Asynchronous Messaging and Conditional GET

Project 4: Group 12

CPSC 449-01 - Fall 2023

Group Members

Micah Baumann, Huy Hoang, Jorge Mejia, Sanjyot Satvi, Arjit Saxena, Ethan Stupin

Project 4 - Project Document

CPSC 449-01 - Fall 2023

Installation

Installing Redis:

```
(.venv) student@tuffix-vm:~/Documents/StudentEnrollment$ sudo apt install --yes redis
[sudo] password for student:
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
```

```
sudo apt update
sudo apt install –yes redis
python -m pip install redis[hiredis]
```

Installing AWS Cli:

```
(.venv) student@tuffix-vm:~/Documents/StudentEnrollment$ curl "https://awscli.amazonaws.com/awscli-exe-linux-x86_64.zip" -o "awscliv2.zip"
unzip awscliv2.zip
sudo ./aws/install
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 56.6M  100 56.6M    0     0  4972k      0  0:00:11  0:00:11 --:--:-- 5848k
Archive:  awscliv2.zip
```

```
curl "https://awscli.amazonaws.com/awscli-exe-linux-x86_64.zip" -o
"awscliv2.zip"
unzip awscliv2.zip
sudo ./aws/install
```

Installing the AWS SDK for Python:

```
(.venv) student@tuffix-vm:~/Documents/StudentEnrollment$ python -m pip install boto3
Requirement already satisfied: boto3 in /home/student/.venv/lib/python3.10/site-packages (1.29.6)
Requirement already satisfied: botocore<1.33.0,>=1.32.6 in /home/student/.venv/lib/python3.10/site-packages (from boto3) (1.32.6)
Requirement already satisfied: jmespath<2.0.0,>=0.7.1 in /home/student/.venv/lib/python3.10/site-packages (from boto3) (1.0.1)
Requirement already satisfied: s3transfer<0.8.0,>=0.7.0 in /home/student/.venv/lib/python3.10/site-packages (from boto3) (0.7.0)
```

```
python -m pip install boto3
```

Installing Java Runtime:

```
(.venv) student@tuffix-vm:~/Documents/StudentEnrollment$ sudo apt install –yes openjdk-19-jre-headless
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
```

```
sudo apt update
sudo apt install –yes openjdk-19-jre-headles
```

Install and start RabbitMQ using the following commands:

```
(.venv) student@tuffix-vm:~/Documents/StudentEnrollment$ sudo rabbitmqctl ping
Will ping rabbit@tuffix-vm. This only checks if the OS process is running and registe
red with epmd. Timeout: 60000 ms.
Ping succeeded
```

```
sudo apt update
sudo apt install --yes rabbitmq-server
```

Verify that RabbitMQ is up and running by executing the following command and checking that it succeeds:

```
sudo rabbitmqctl ping
```

Project 4 - Project Document

CPSC 449-01 - Fall 2023

Enrollment notification service

Subscribe to notifications for a new course

```python
@app.get("/subscribe/{studentid}/{classid}")
def example(studentid: int,
            classid: int,
            email_header : Annotated[str | None, Header(convert_underscores=False)] = "None",
            callback_header : Annotated[str | None, Header(convert_underscores=False)] = "None",
            r = Depends(get_redis)):

    subscription_key = f"subscription:{studentid}_{classid}"

    r.hset(subscription_key,
           mapping={
               "studentid" : studentid,
               "classid" : classid,
               "email_header" : email_header,
               "callback_header" : callback_header
           })

    #Left this here just in case you want to see the entry that was just created
    # created_subscription = r.hgetall(subscription_key)
    # return {"created_subscription" : created_subscription}

    return {"message" : "Subscription succesful"}
```

```
{
    "endpoint": "/student/subscribe/{classid}",
    "input_headers": [
        "email_header",
        "callback_header"
    ],
    "method": "POST",
    "output_encoding": "no-op",
    "backend": [
    {
        "url_pattern": "/subscribe/{JWT.jti}/{classid}",
        "method": "POST",
        "host": [
            "http://localhost:5600"
        ],
        "encoding": "no-op",
        "extra_config": {
            "backend/http": {
                "return_error_details": "backend_alias"
            }
        }
    }
    ],
    "extra_config": {
        "auth/validator": {
            "alg": "RS256",
            "roles": ["Student"],
            "jwk_local_path": "./etc/public.json",
            "disable_jwk_security": true,
            "operation_debug": true
        }
    }
},
```

This endpoint is designed to facilitate class subscription for users. By supplying their student ID and class ID, a user can easily subscribe to their desired class. Additionally, the endpoint offers the option to provide email and callback information through headers for enhanced communication. Once the subscription process is complete, the details are securely stored within a Redis database. To confirm a successful subscription, the user is promptly sent a confirmation message.

Project 4 - Project Document

CPSC 449-01 - Fall 2023

List their current subscriptions

```python
@app.get("/subscriptions/{student_id}")
def get_subscriptions(student_id: int, r = Depends(get_redis)):
    pattern = f'subscription:{student_id}_*'
    subscriptions = []
    subscribed_courses=[]
    cursor = '0'
    while cursor != 0:
        cursor, keys = r.scan(cursor=cursor, match=pattern)
        for key in keys:
            subscription_data = r.hgetall(key)
            subscriptions.append(subscription_data.get(b'classid').decode())

        if cursor == 0:
            break
        cursor = int(cursor)


    for class_id in subscriptions:
        response = classes_table.get_item(
            Key={
                "ClassID": int(class_id)
            }
    )

        item = response.get("Item")
        if item:
            class_name = item.get("ClassName")
            if class_name:
                subscribed_courses.append(class_name)



    return {"subscriptions": subscribed_courses}
```

```json
{
    "endpoint": "/student/subscriptions",
    "method": "GET",
    "output_encoding": "no-op",
    "backend": [
    {
        "url_pattern": "/subscriptions/{JWT.jti}",
        "method": "GET",
        "host": [
            "http://localhost:5600"
        ],
        "encoding": "no-op",
        "extra_config": {
            "backend/http": {
                "return_error_details": "backend_alias"
            }
        }
    }
    ],
    "extra_config": {
        "auth/validator": {
            "alg": "RS256",
            "roles": ["Student"],
            "jwk_local_path": "./etc/public.json",
            "disable_jwk_security": true,
            "operation_debug": true
        }
    }
}
```

This endpoint is specifically designed to enable students to access a list of classes to which they are currently subscribed. By inputting their student ID, students can initiate a request to retrieve this list. The system efficiently pulls the subscription data from Redis and complements it with further class details sourced from an additional database table. By adding all of these together, we get the delivery of the requested information in a well-organized JSON format, ensuring that students receive a clear and structured overview of their class subscriptions.

Project 4 - Project Document

CPSC 449-01 - Fall 2023

Unsubscribe from a course

```python
@app.get("/unsubscribe/{studentid}/{classid}")
def example(studentid: int,
            classid: int,
            email_header : Annotated[str | None, Header(convert_underscores=False)] = "None",
            callback_header : Annotated[str | None, Header(convert_underscores=False)] = "None",
            r = Depends(get_redis)):

    subscription_key = f"subscription:{studentid}_{classid}"

    all_keys = list(r.hgetall(subscription_key).keys())
    r.hdel(subscription_key, *all_keys)

    #Left this here just in case you want to check if key was actually deleted
    # created_subscription = r.hgetall(subscription_key)
    # print(created_subscription)

    return {"message" : "Unsubscription successful"}
```

```
{
    "endpoint": "/student/unsubscribe/{classid}",
    "method": "DELETE",
    "output_encoding": "no-op",
    "backend": [
    {
        "url_pattern": "/unsubscribe/{JWT.jti}/{classid}",
        "method": "DELETE",
        "host": [
            "http://localhost:5600"
        ],
        "encoding": "no-op",
        "extra_config": {
            "backend/http": {
                "return_error_details": "backend_alias"
            }
        }
    }
    ],
    "extra_config": {
        "auth/validator": {
            "alg": "RS256",
            "roles": ["Student"],
            "jwk_local_path": "./etc/public.json",
            "disable_jwk_security": true,
            "operation_debug": true
        }
    }
},
```

This endpoint is used to streamline the process of unsubscribing from a class. Users can initiate unsubscription by submitting their student ID and class ID. Upon receiving these details, the endpoint efficiently removes the corresponding subscription data from the Redis database. To complete the process, it sends a confirmation message to the user, ensuring they are informed of the successful unsubscription. This functionality provides a hassle-free way for users to manage their class subscriptions.

Project 4 - Project Document

CPSC 449-01 - Fall 2023

Redis

```
21      def get_redis():
22          yield redis.Redis()
23
```

```
        r = Depends(get_redis)):

subscription_key = f"subscription:{studentid}_{classid}"

r.hset(subscription_key,
       mapping={
           "studentid" : studentid,
           "classid" : classid,
           "email_header" : email_header,
           "callback_header" : callback_header
       })

#Left this here just in case you want to see the entry that was just created
# created_subscription = r.hgetall(subscription_key)
# return {"created_subscription" : created_subscription}

return {"message" : "Subscription succesful"}
```

Since there may be multiple clients and updates, it will need to continue after a service restart, so then the stored subscriptions should be stored in Redis. Its sheer speed will allow us to handle high traffic since this will be one busy endpoint. The use of Redis ensures that no subscription is lost in the event of a service disruption, allowing the system to resume operations without missing a beat, thus preserving the integrity and reliability of the communication process.

Project 4 - Project Document

CPSC 449-01 - Fall 2023

Producing enrollment notifications

```
# Add to MQ
mq_msg = {
    "action": "enrolled",
    "uid": next_on_waitlist,
    "class": classid
}
mq.basic_publish(exchange='notify', routing_key='', body=json.dumps(mq_msg))
```

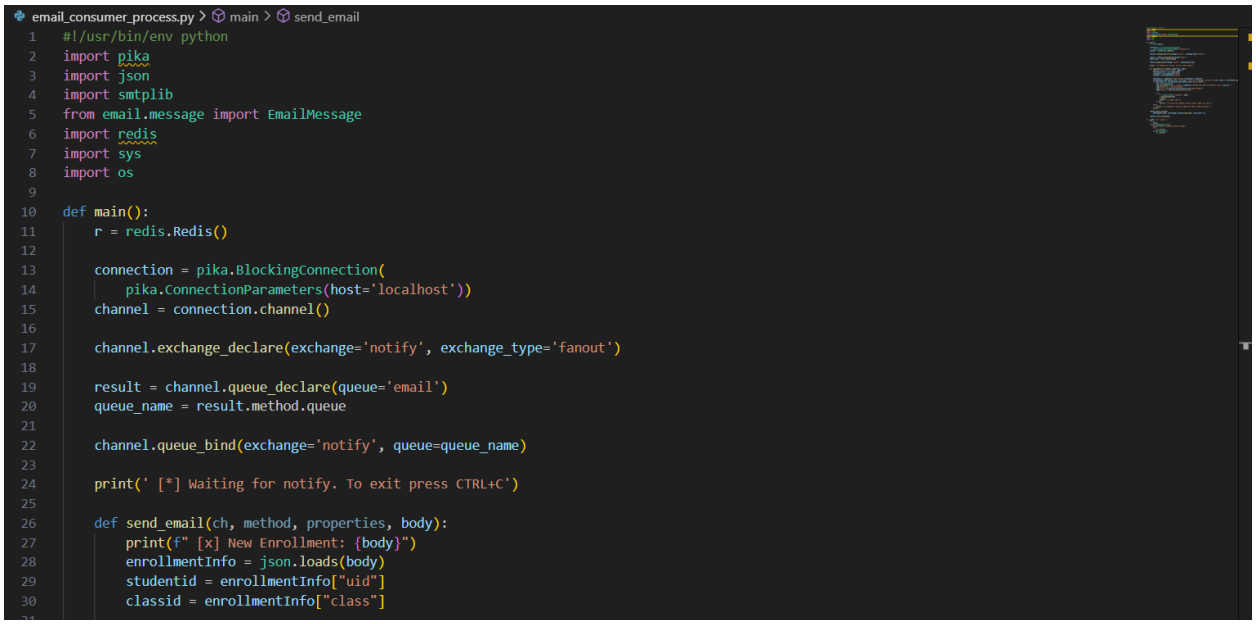
```
class RabbitMQ:
    def __init__(self):
        self.connection = pika.BlockingConnection(
            pika.ConnectionParameters(host='localhost')
        )
        self.channel = self.connection.channel()
        self.channel.exchange_declare(exchange='notify', exchange_type='fanout')

    def close(self):
        self.connection.close()

def get_mq():
    rabbitmq = RabbitMQ()
    try:
        yield rabbitmq.channel
    finally:
        rabbitmq.close()
```

This process is where a message is dispatched to a message queue through a fanout exchange mechanism whenever a student is automatically enrolled in a class from the waitlist. To facilitate this message sending, it is necessary to initialize RabbitMQ within the enrollment service's API file. This setup ensures that the enrollment system can communicate effectively and manage the enrollment workflow efficiently.

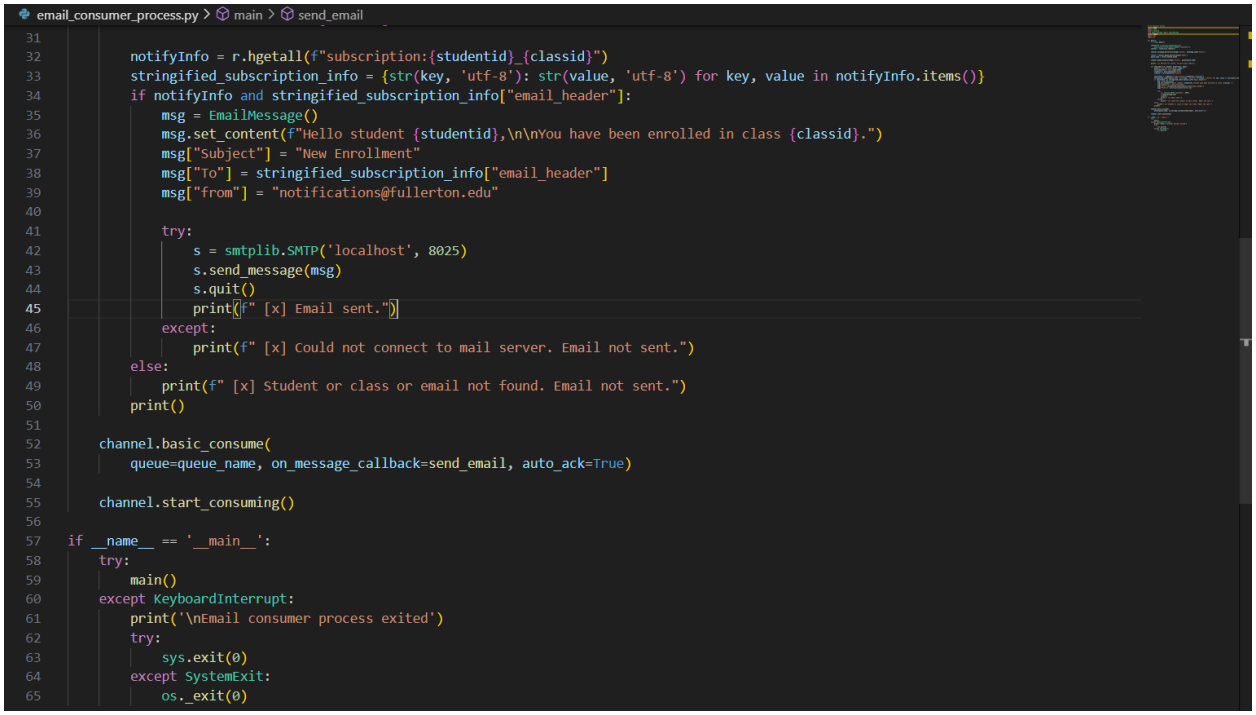
Project 4 - Project Document

CPSC 449-01 - Fall 2023

Consuming enrollment notifications



```python
#!/usr/bin/env python
import pika
import json
import smtplib
from email.message import EmailMessage
import redis
import sys
import os

def main():
    r = redis.Redis()

    connection = pika.BlockingConnection(
        pika.ConnectionParameters(host='localhost'))
    channel = connection.channel()

    channel.exchange_declare(exchange='notify', exchange_type='fanout')

    result = channel.queue_declare(queue='email')
    queue_name = result.method.queue

    channel.queue_bind(exchange='notify', queue=queue_name)

    print(' [*] Waiting for notify. To exit press CTRL+C')

    def send_email(ch, method, properties, body):
        print(f" [x] New Enrollment: {body}")
        enrollmentInfo = json.loads(body)
        studentid = enrollmentInfo["uid"]
        classid = enrollmentInfo["class"]
```



```python
        notifyInfo = r.hgetall(f"subscription:{studentid}_{classid}")
        stringified_subscription_info = {str(key, 'utf-8'): str(value, 'utf-8') for key, value in notifyInfo.items()}
        if notifyInfo and stringified_subscription_info["email_header"]:
            msg = EmailMessage()
            msg.set_content(f"Hello student {studentid},\n\nYou have been enrolled in class {classid}.")
            msg["Subject"] = "New Enrollment"
            msg["To"] = stringified_subscription_info["email_header"]
            msg["from"] = "notifications@fullerton.edu"

            try:
                s = smtplib.SMTP('localhost', 8025)
                s.send_message(msg)
                s.quit()
                print(f" [x] Email sent.")
            except:
                print(f" [x] Could not connect to mail server. Email not sent.")
        else:
            print(f" [x] Student or class or email not found. Email not sent.")
        print()

    channel.basic_consume(
        queue=queue_name, on_message_callback=send_email, auto_ack=True)

    channel.start_consuming()

if __name__ == '__main__':
    try:
        main()
    except KeyboardInterrupt:
        print('\nEmail consumer process exited')
        try:
            sys.exit(0)
        except SystemExit:
            os._exit(0)
```

This process is designed to wait attentively for the arrival of messages within a message queue, and the email serves as an alert to a user, indicating that a new message awaits their attention. The consumer process ensures that no message goes unnoticed and that the user is kept informed in a timely manner, thus streamlining the flow of information and maintaining an effective line of communication.

Project 4 - Project Document

CPSC 449-01 - Fall 2023

Consuming Webhook Callback Notifications

webhook_consumer_process.py 2

webhook_consumer_process.py > ...

```python
#!/usr/bin/env python
import pika
import httpx
import sys
import os
import redis
import json

def main():
    r = redis.Redis()

    connection = pika.BlockingConnection(
        pika.ConnectionParameters(host='localhost'))
    channel = connection.channel()

    channel.exchange_declare(exchange='notify', exchange_type='fanout')

    result = channel.queue_declare(queue='webhook')
    queue_name = result.method.queue

    channel.queue_bind(exchange='notify', queue=queue_name)

    print(' [*] Waiting for notify. To exit press CTRL+C')

    def callback(ch, method, properties, body):
        enrollmentInfo = json.loads(body)
```

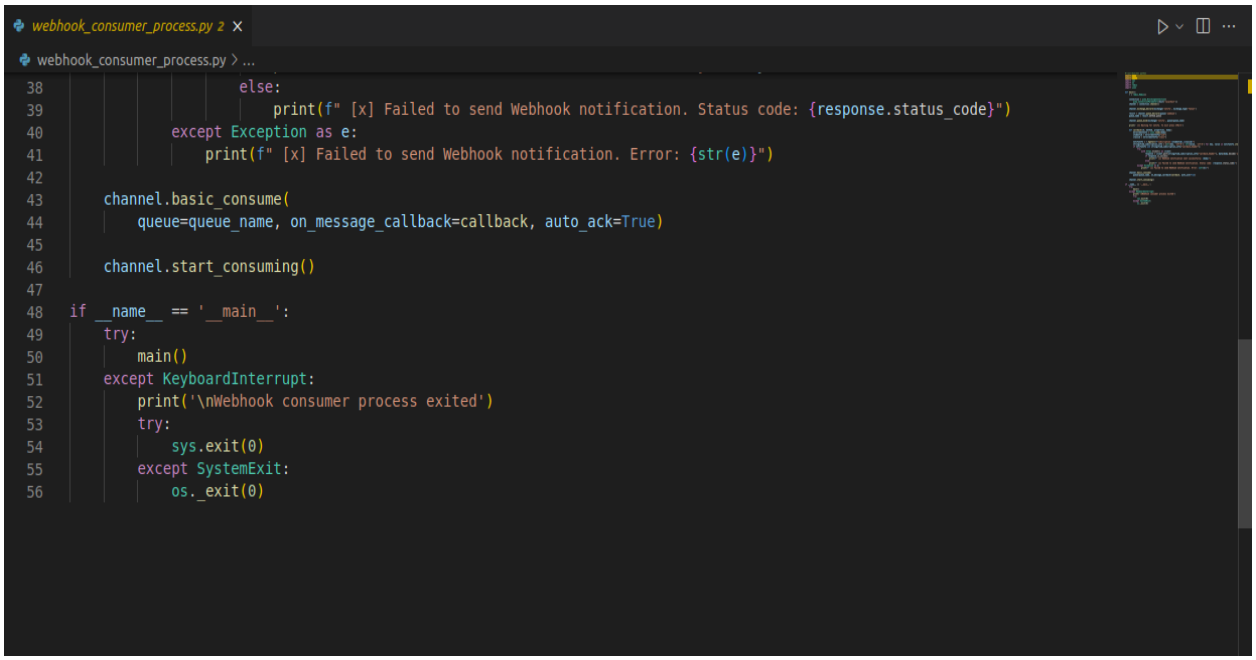
webhook_consumer_process.py 2

webhook_consumer_process.py > ...

```python
    def callback(ch, method, properties, body):
        enrollmentInfo = json.loads(body)
        studentid = enrollmentInfo["uid"]
        classid = enrollmentInfo["class"]

        notifyInfo = r.hgetall(f"subscription:{studentid}_{classid}")
        stringified_subscription_info = {str(key, 'utf-8'): str(value, 'utf-8') for key, value in notifyInfo.items()}
        if notifyInfo and stringified_subscription_info["callback_header"]:
            try:
                with httpx.Client() as client:
                    response = client.post(stringified_subscription_info["callback_header"], data=body.decode('utf-8'))
                    if response.is_success:
                        print(f" [x] Webhook notification sent successfully: {body}")
                    else:
                        print(f" [x] Failed to send Webhook notification. Status code: {response.status_code}")
            except Exception as e:
                print(f" [x] Failed to send Webhook notification. Error: {str(e)}")

    channel.basic_consume(
        queue=queue_name, on_message_callback=callback, auto_ack=True)

    channel.start_consuming()

if __name__ == '__main__':
    try:
        main()
```

Project 4 - Project Document

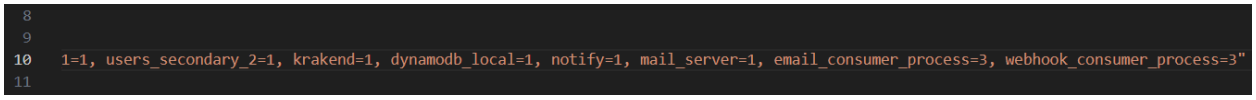
CPSC 449-01 - Fall 2023



```
webhook_consumer_process.py 2 ×
webhook_consumer_process.py > ...
                    else:
                        print(f" [x] Failed to send Webhook notification. Status code: {response.status_code}")
                except Exception as e:
                    print(f" [x] Failed to send Webhook notification. Error: {str(e)}")

    channel.basic_consume(
        queue=queue_name, on_message_callback=callback, auto_ack=True)

    channel.start_consuming()

if __name__ == '__main__':
    try:
        main()
    except KeyboardInterrupt:
        print('\nWebhook consumer process exited')
        try:
            sys.exit(0)
        except SystemExit:
            os._exit(0)
```

This webhook consumer script is designed to integrate with a RabbitMQ message broker and a Redis database to receive notifications via a RabbitMQ message broker and send them as HTTP POST requests to subscribed endpoints.

```
Procfile
users_primary: ./bin/litefs mount -config ./users/etc/primary.yml
users_secondary_1: ./bin/litefs mount -config ./users/etc/secondary_1.yml
users_secondary_2: ./bin/litefs mount -config ./users/etc/secondary_2.yml
enroll: uvicorn --port $PORT enroll.api:app --reload
krakend: echo krakend.json | entr -nrz krakend run --port $PORT --config krakend.json
dynamodb_local: java -Djava.library.path=./bin/DynamoDBLocal_lib -jar ./bin/DynamoDBLocal.jar -sharedDb -port $PORT
notify: uvicorn --port $PORT notify.notify:app --reload
mail_server: python -m aiosmtpd -n -d
email_consumer_process: python email_consumer_process.py
webhook_consumer_process: python webhook_consumer_process.py
```



```
1=1, users_secondary_2=1, krakend=1, dynamodb_local=1, notify=1, mail_server=1, email_consumer_process=3, webhook_consumer_process=3"
```

We edited our procfile with the inclusion of webhook and email consumer processes. This addition is aimed at enabling the initiation of multiple instances of each process by utilizing the formation flag. This configuration is crucial for scaling the application's capabilities and ensuring efficient process management.

Project 4 - Project Document

CPSC 449-01 - Fall 2023

Testing

In order to test the ability to send email, use the following commands to install the aiosmtpd SMTP server and run it in debugging mode:

```
python -m pip install aiosmtpd
python -m aiosmtpd -n -d
```

The server will listen on `localhost` port 8025, and the default handler will print the contents of received messages to `stdout`.

Project 4 - Project Document

CPSC 449-01 - Fall 2023

Cache waiting list positions

HTTP Conditional Requests

```python
def update_cache_if_exists(classid :int,r):
    cache_key = f"waitlist_{classid}"
    if r.exists(cache_key):
        byte_waitlist = r.lrange(f"waitClassID_{classid}", 0, -1)
        waitlist = [student_id.decode('utf-8') for student_id in byte_waitlist]
        cache_value = json.dumps({'data': waitlist, 'last_modified': datetime.utcnow().timestamp()})
        r.set(cache_key, cache_value)
        print("Waitlist Updated ")
        return True
    else:
        return False
```

Within our system, there is a dedicated function designed to update the waitlist as needed. This function is automatically triggered in response to any modifications within the waitlist. Such changes may include a student voluntarily withdrawing from a class or waitlist, or an instructor removing a student from the list. The primary purpose of this function is to facilitate the rapid updating and retrieval of waitlist data, ensuring that subsequent requests for information are both swift and accurate.

```python
view_waitlist_position(studentid: int, classid: int, username: str, email: str,request:Request, r = Depends(get_redis)):
"""API to view a student's position on the waitlist.
Args: ...
Returns: ...
"""
check_user(studentid, username, email)
redisCacheKey = f"waitlist_{classid}"
cachedData = r.get(redisCacheKey)
#print(f'Cached Data ${cachedData}')

#if we find cached data lastModified is earlier that modifiedsince time we give 304
if cachedData:
    cachedData = json.loads(cachedData)
    #print(f'Cached Data${cachedData}')
    allStudentIds = cachedData['data']
    #print(f"All student data${allStudentIds}")
    student_exists = str(studentid) in allStudentIds
    if student_exists:
        lastModifiedTime = datetime.fromtimestamp(cachedData['last_modified'])
        #print(f'Last Modified Time ${lastModifiedTime}')
        modifiedSince = request.headers.get('If-Modified-Since')
        #print(f'modified since ${modifiedSince}')
        if modifiedSince:
            modifiedSinceTime = datetime.strptime(modifiedSince, '%a, %d %b %Y %H:%M:%S GMT')
            #print(f'modified since ${modifiedSinceTime}')
            if lastModifiedTime <= modifiedSinceTime:
                return Response(content=None, status_code=304)
position = r.lpos(f"waitClassID_{classid}", studentid)
#print(f'position {position}')

if position is not None:
    message = f"Student {studentid} is on the waitlist for class {classid} in position {position+1}"
else:
    message = f"Student {studentid} is not on the waitlist for class {classid}"
    raise HTTPException(
        status_code=404,
        detail=message,
    )
byte_waitlist = r.lrange(f"waitClassID_{classid}", 0, -1)
waitlist = [student_id.decode('utf-8') for student_id in byte_waitlist]
cacheValue = json.dumps({'data':waitlist,'last_modified':datetime.utcnow().timestamp()})
#print(f'cached Value ${cacheValue}')
response = JSONResponse(content={message:position+1})
response.headers["If-Modified-Since"] = (datetime.utcnow() + timedelta(seconds=1)).strftime("%a, %d %b %Y %H:%M:%S GMT")

# Cache the response
r.set(redisCacheKey, cacheValue)
return response
```

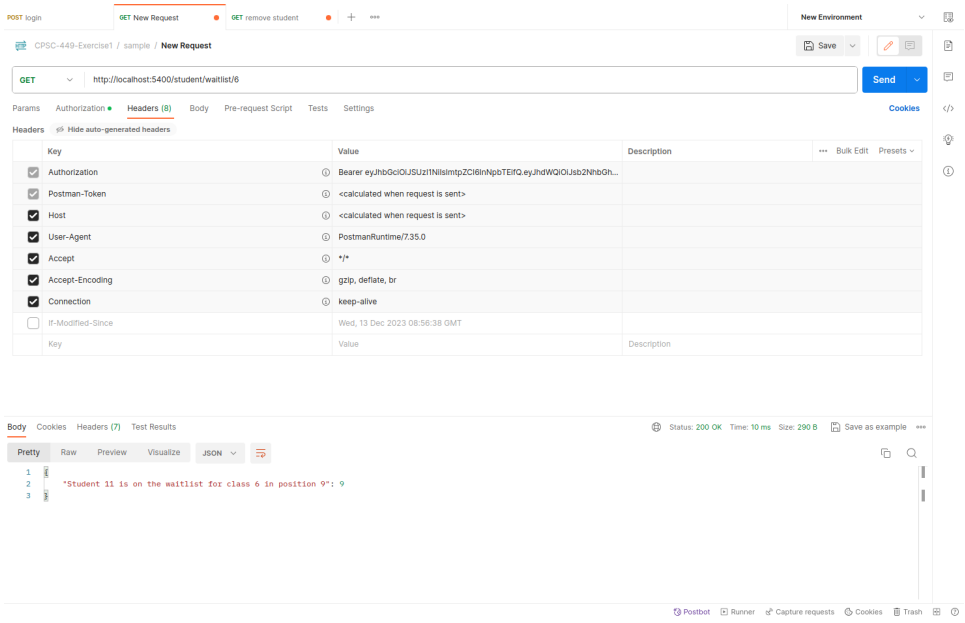
The view_waitlist_position serves as a dynamic view for the waitlist position, designed to reflect real-time updates. It ensures that any alterations in the waitlist are captured and updated, providing users with the most current information regarding their position. This feature is particularly useful for users who need to track changes to their status on the waitlist, allowing them to stay informed and make decisions accordingly.

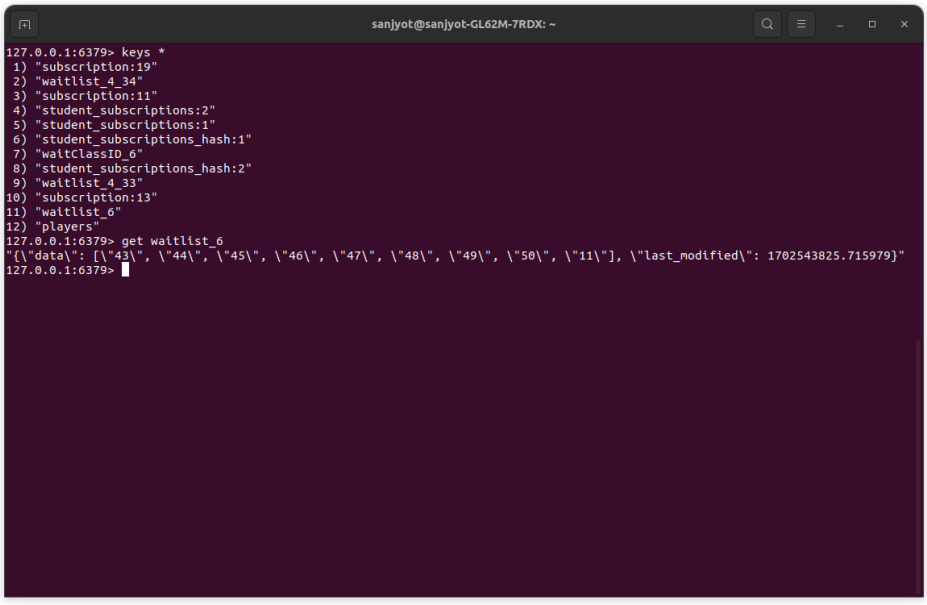
Project 4 - Project Document

CPSC 449-01 - Fall 2023

Testing

Initialize Postman to communicate with the project.

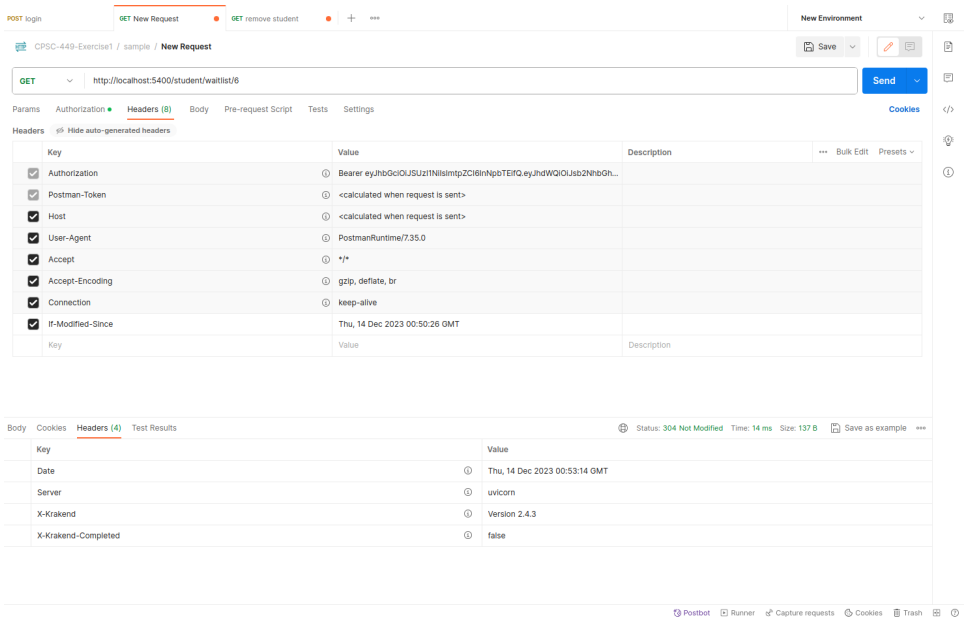




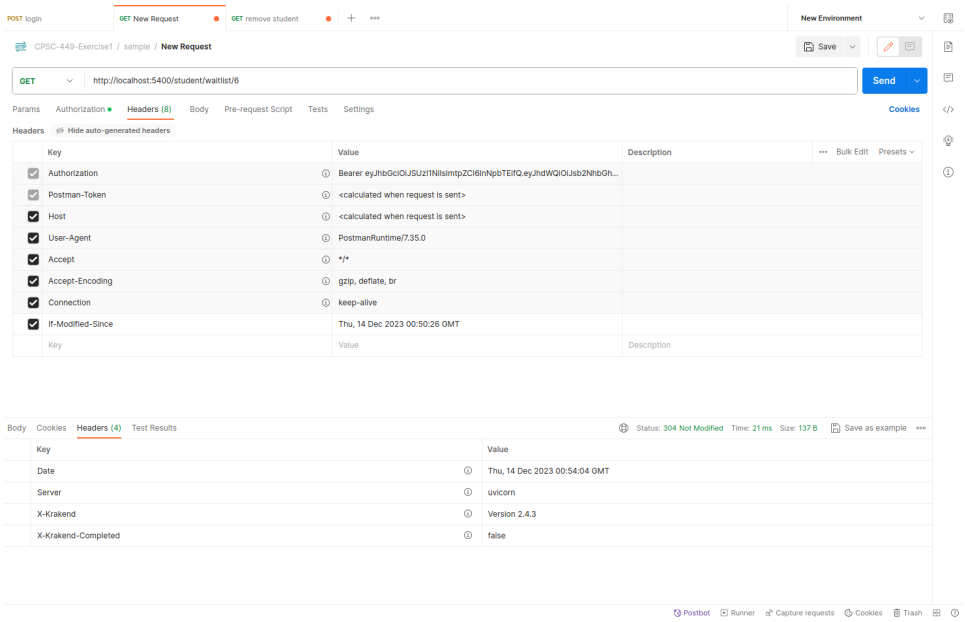
We then created a waitlist cache that includes a last_modified timestamp to track updates.This cache will allow efficient retrieval of waitlist entries and ensure that the most recent changes are reflected with the timestamp now indicating the last modification date and time.

Project 4 - Project Document

CPSC 449-01 - Fall 2023

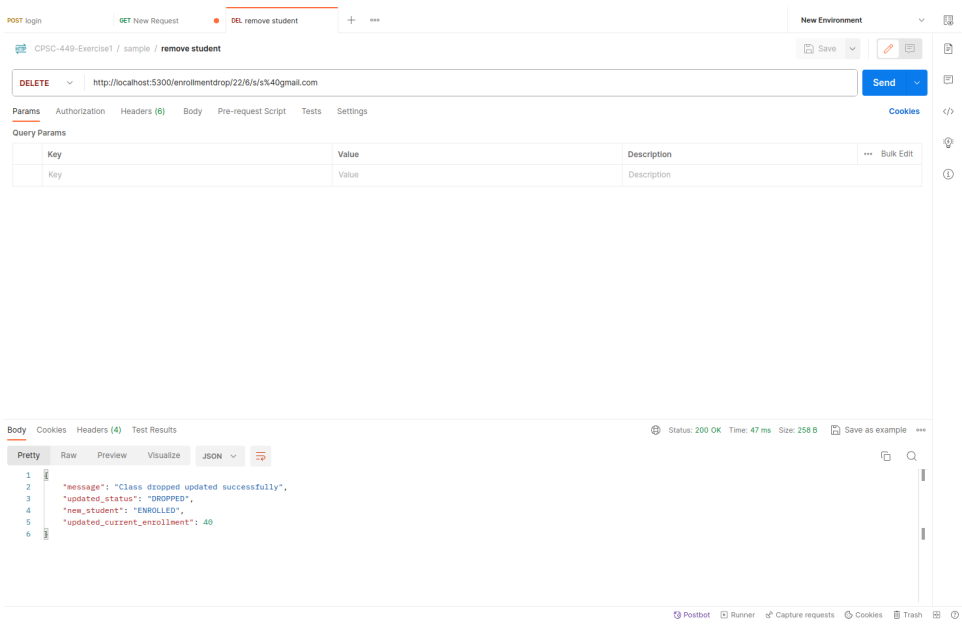


The If-Modified-Since header plays a crucial role in optimizing data transfer during HTTP requests. When included in a request, this header acts as a conditional check to determine if the requested resource has undergone any updates since the specified timestamp. If the resource has been modified, it will be retrieved; otherwise, the transfer is bypassed.

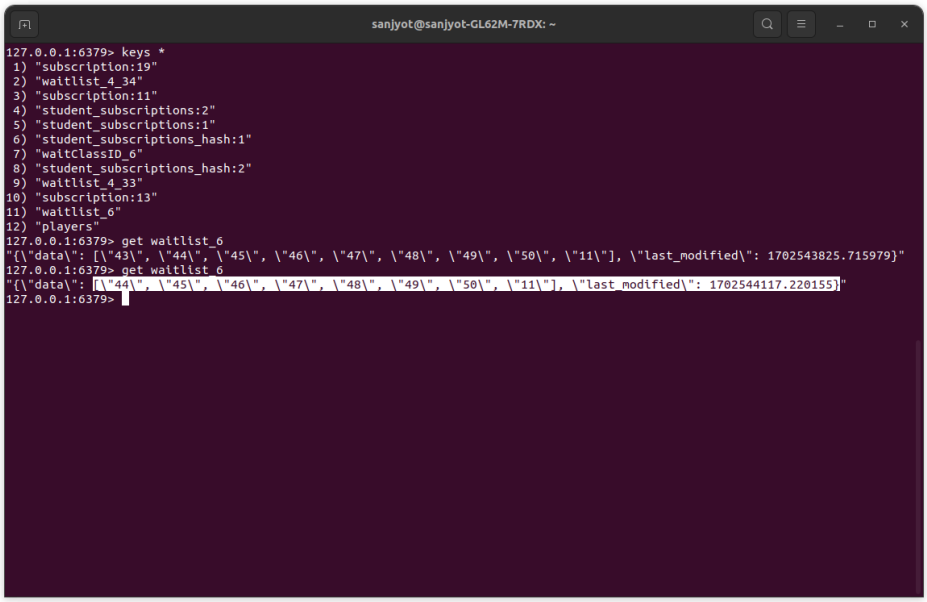


When multiple requests yield a 304 status code, it signifies that there has been no recent change in the class waitlist, and the last_updated timestamp predates the time indicated in the If-Modified-Since header. This HTTP response code is instrumental in optimizing server-client interactions by confirming that the content remains unaltered since the last known update.

Consequently, clients can confidently rely on the cached version of the waitlist, eliminating the need for redundant data retrieval from the server.



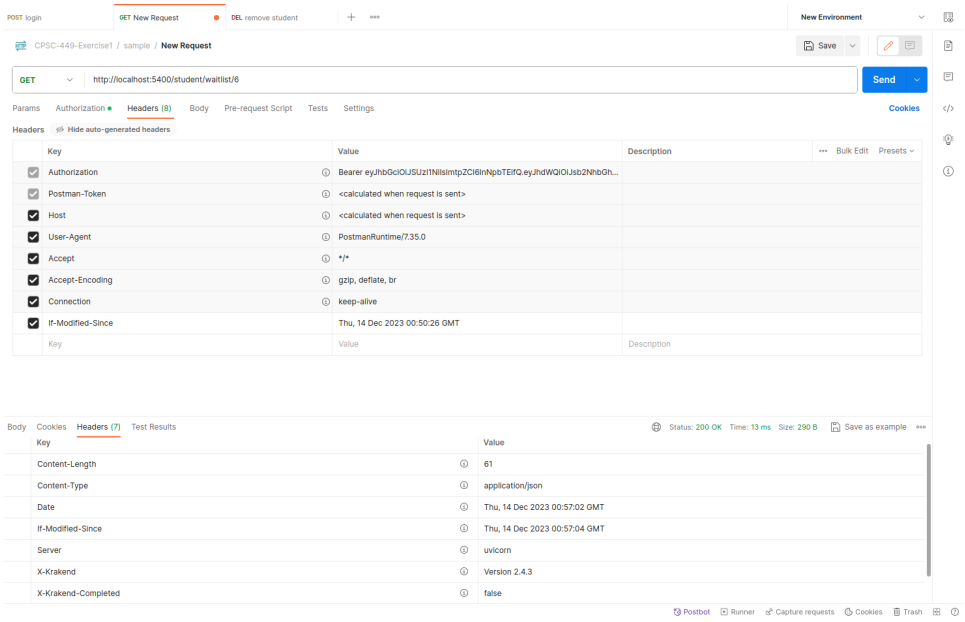
The system is designed to maintain an accurate and current waitlist for class enrollments. When a student opts to drop from a class, the waitlist cache is promptly updated to reflect this change, and the last_modified timestamp is accordingly refreshed. This ensures that the waitlist remains a reliable source of information, mirroring the latest enrollment status. Additionally, the updated last_modified timestamp plays a critical role in handling conditional requests, allowing for efficient content change management. By keeping the waitlist and timestamps up-to-date, the system effectively supports the dynamic nature of class enrollments and waitlist management.



Project 4 - Project Document

CPSC 449-01 - Fall 2023

Upon a student's withdrawal from a class, the waitlist cache is revised to mirror the updated class capacity. This modification allows for the retrieval of waitlist data for subsequent inquiries, ensuring that the information provided is both swift and current. It also includes the newly vacated position which offers other students the chance to register for the class.



When the last_modified timestamp is updated to a later time than the If-Modified-Since timestamp, it's an indication that there have been changes. As a result, the server fetches, returns, and caches the new portion of the content. It ensures that the client receives the latest update made to the waitlist.