REDIS

Assignment 3

Papadatos Ioannis (t8190314)
Panourgia Evangelia (t8190130)
Professor: Chatziantoniou Damianos

Latest Update: May 14, 2022

School of Management Science and Technology,
Athens University of Economics and Business

Contents

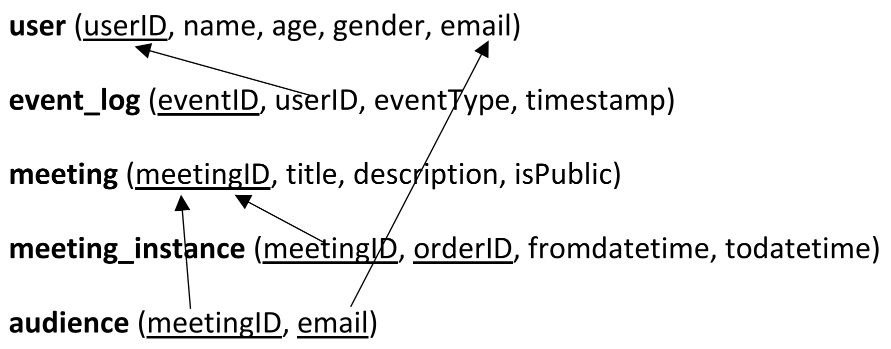
Introduction

Redis is an open source, in-memory key-value data store used by millions of developers as a database, cache, streaming engine, and message broker. In this assignment we will use Redis to create the API for an imaginary real-time collaboration application.

We utilized docker to quickly setup the environment needed for the assignment. We used MySQL as the relational database to store the data that are not accessed very frequently, we used Redis to store the data of the active meetings that require low latency (Redis docker image has durability enabled by default) & we used the FastAPI which is a modern, fast (high-performance), web framework for building APIs with Python, to build the API.

Data Modeling

Relational model:



user (userID, name, age, gender, email)

event_log (eventID, userID, eventType, timestamp)

meeting (meetingID, title, description, isPublic)

meeting_instance (meetingID, orderID, fromdatetime, todatetime)

audience (meetingID, email)

We used the SQLAlchemy library to create the DB tables:

```
user = Table(
  'user', meta,
  Column('userID', Integer, primary_key=True, autoincrement=True),
  Column('name', String(255)),
  Column('age', Integer),  # Normally DoB is prefered - computed field
  Column('gender', Enum(Gender)),
  Column('email', String(255), unique=True, index=True)
)
```

```
event_log = Table(
  'event_log', meta,
  Column('eventID', Integer, primary_key=True, autoincrement=True),
  Column('userID', Integer, ForeignKey('user.userID')),
  Column('eventType', Enum(EventType)),
  Column('timestamp', DateTime(timezone=True), server_default=func.now())
)
```

```
meeting = Table(
  'meeting', meta,
  Column('meetingID', Integer, primary_key=True, autoincrement=True),
  Column('title', String(255)),
  Column('description', Text),
  Column('isPublic', Boolean)
)
```

```
meeting_instance = Table(
  'meeting_instance', meta,
  Column('meetingID', Integer, ForeignKey('meeting.meetingID'), primary_key=True),
  Column('orderID', Integer, primary_key=True),
  Column('fromdatetime', DateTime(timezone=True)),
  Column('todatetime', DateTime(timezone=True))
)
```

```
audience = Table(
  'audience', meta,
  Column('meetingID', Integer, ForeignKey('meeting.meetingID'), primary_key=True),
  Column('email', String(255), ForeignKey('user.email'), primary_key=True)
)
```

Redis:

Redis Hashes are maps between string fields and string values, so they are the perfect data type to represent objects. Also, a hash with a few fields is stored in a way that takes very little space.

We used hashes to store the details of the users:

```
user:{userID}  --->  name => {name}
                     age => {age}
                     gender => {gender}
                     email => {email}
```

We used hashes to store the details of the meetings. These hashes contain the details from both meeting & meeting_instance of the relational model, in order to reduce the number of round trips between our application and the Redis server:

```
meeting:{meetingID}:order:{orderID}  --->  title => {title}
                                           description => {description}
                                           isPublic => {isPublic}
                                           fromdatetime => {fromdatetime}
                                           todatetime => {todatetime}
```

We used hashes to store the details of the logs:

```
log:{eventID}  --->  userID => {userID}
                     eventType => {eventType}
                     timestamp => {timestamp}
```

We used hashes to store the details of the messages:

```
message:{messageID}  --->  userID => {userID}
                           body => {body}
                           timestamp => {timestamp}
```

We used a set to store the keys of the active meetings:

```
active_meetings  --->  { meeting:{meetingID}:order:{orderID}, ... }
```

We used a list to store the emails of the users that are allowed to attend the meeting (if it is not public):

meeting:{meetingID}:order:{orderID}:audience ---> [email, ...]

We used a list to store the userIDs of the participants (users that are attending the meeting):

meeting:{meetingID}:order:{orderID}:participants ---> [userID, ...]

We used a list to store the messageIDs of the messages that were posted in the meeting:

meeting:{meetingID}:order:{orderID}:messages ---> [messageID, ...]

We used a list to store the eventIDs of the logs that are related to the meeting:

meeting:{meetingID}:order:{orderID}:logs ---> [eventID, ...]

Initial Data

```
INSERT INTO `user` (`userID`, `name`, `age`, `gender`, `email`) VALUES
(1, 'john', 21, 'male', 'johnpapadatos77@hotmail.com'),
(2, 'hamid', 22, 'male', 'hamidmeh9@hotmail.com'),
(3, 'haled', 23, 'male', 'haledmeh10@hotmail.com');
```

```
INSERT INTO `meeting` (`meetingID`, `title`, `description`, `isPublic`) VALUES
(1, 'BDMS Course', 'This course provides an introduction to the following systems:
Hadoop, Redis, MongoDB, Neo4J, Azure Streams', 0),
(2, 'DSA Course', 'This course covers the fundementals of data structures &
algorithms.', 0),
(3, 'Seminar', 'This is an open HR seminar about recruitment.', 1);
```

```
INSERT INTO `audience` (`meetingID`, `email`) VALUES
(2, 'haledmeh10@hotmail.com'),
(2, 'hamidmeh9@hotmail.com'),
(1, 'johnpapadatos77@hotmail.com');
```

```
INSERT INTO `meeting_instance` (`meetingID`, `orderID`, `fromdatetime`,
`todatetime`) VALUES
(1, 1, '2022-04-15 13:00:00', '2022-04-15 17:00:00'),
(1, 2, '2022-04-22 13:00:00', '2022-04-22 17:00:00'),
(1, 3, '2022-04-29 13:00:00', '2022-04-29 17:00:00'),
(1, 4, '2022-05-06 13:00:00', '2022-05-06 17:00:00'),
(2, 1, '2022-04-29 13:00:00', '2022-04-29 15:00:00'),
(2, 2, '2022-05-06 13:00:00', '2022-05-06 15:00:00'),
(3, 1, '2022-05-06 13:00:00', '2022-05-06 21:00:00');
```

Functions

The load_users() function is executed once before the application starts. This function performs a query to retrieve the users from the database and then for each user it creates his/her hash in Redis. (In a real application we would only store active users in Redis).

```
@app.on_event("startup")
def load_users():
  try:
    with engine.connect() as conn:
      users = conn.execute(text("SELECT * FROM user"))

      for userID, name, age, gender, email in users:
        cache.hmset(f'user:{userID}', {
          "name": name,
          "age": age,
          "gender": gender,
          "email": email
        })
  except Exception as exc:
    exception_logger.exception(exc)
```

Function 1: a meeting instance becomes active

Function 2: when a meeting ends, all participants must leave

The activate_meetings() function is periodically called by the server every 15 seconds. Every time it is called, it performs a query to retrieve the meetings from the database and then for each meeting checks if it exists in the active_meetings.

If it does not exist but it is time to activate it (Function 1):

- We add the meeting_instance_key to the set that contains the keys of the active meetings
- We create the meeting's hash
- We create the list with the emails of the users that are allowed to attend the meeting (if it is not public)

If it does exist but is time to terminate it (Function 2):

- We remove the meeting_instance_key from the set that contains the keys of the active meetings
- We delete the meeting's hash
- We delete the list with the emails of the users that are allowed to attend the meeting (if it is not public)
- We delete the list with the eventIDs of the logs related to this meeting and for each eventID we also delete its hash, however we persist all of the logs in the database
- We delete the list with the userIDs of the meeting's participants and for each participant we create a leave log in the database, with departure time the finish_time (todatetime) of the meeting
- We delete the list with the messageIDs of the messages related to this meeting and for each messageID we also delete its hash

```python
@app.on_event("startup")
@repeat_every(seconds=15)
def activate_meetings():
  try:
    current_time = datetime.datetime.now()

    with engine.connect() as conn:
      meetings = conn.execute(text("""
        SELECT m.meetingID, orderID, title, description, isPublic, fromdatetime, todatetime
        FROM meeting as m, meeting_instance as mi
        WHERE m.meetingID = mi.meetingID
      """))

    for meetingID, orderID, title, description, isPublic, start_time, finish_time in meetings:
      meeting_instance_key = f'meeting:{meetingID}:order:{orderID}'

      if cache.sismember('active_meetings', meeting_instance_key):
        if current_time > finish_time:
          cache.srem('active_meetings', meeting_instance_key)
          cache.delete(meeting_instance_key)
          cache.delete(f'{meeting_instance_key}:audience')
          delete_logs(meeting_instance_key)
          delete_participants(meeting_instance_key, str(finish_time))
          delete_messages(meeting_instance_key)
      else:
        if start_time < current_time < finish_time:
          cache.sadd('active_meetings', meeting_instance_key)
          cache.hmset(meeting_instance_key, {
            "title": title,
            "description": description,
            "isPublic": isPublic,
            "fromdatetime": str(start_time),
            "todatetime": str(finish_time)
          })

          if not(isPublic):
            create_audience(meeting_instance_key, meetingID)
  except Exception as exc:
    exception_logger.exception(exc)
```

```
def delete_logs(meeting_instance_key: str):
  with engine.connect() as conn:
    logs = cache.lrange(f'{meeting_instance_key}:logs', 0, -1)
    for eventID in logs:
      userID, eventType, timestamp = \
        cache.hmget(f'log:{eventID}', ["userID", "eventType", "timestamp"])

      conn.execute(text(f"""
        INSERT INTO event_log (userID, eventType, timestamp)
        VALUES ({userID}, "{eventType}", "{timestamp}")
      """))

      cache.delete(f'log:{eventID}')
    cache.delete(f'{meeting_instance_key}:logs')


def delete_participants(meeting_instance_key: str, finish_time: str):
  with engine.connect() as conn:
    participants = cache.lrange(f'{meeting_instance_key}:participants', 0, -1)
    for userID in participants:
      conn.execute(text(f"""
        INSERT INTO event_log (userID, eventType, timestamp)
        VALUES ({userID}, '{EventType.leave}', '{finish_time}')
      """))
    cache.delete(f'{meeting_instance_key}:participants')


def delete_messages(meeting_instance_key: str):
  with engine.connect() as conn:
    messages = cache.lrange(f'{meeting_instance_key}:messages', 0, -1)
    for messageID in messages:
      cache.delete(f'message:{messageID}')
    cache.delete(f'{meeting_instance_key}:messages')


def create_audience(meeting_instance_key: str, meetingID: int):
  with engine.connect() as conn:
    audience = conn.execute(text(f"""
      SELECT email
      FROM meeting as m, audience as a
      WHERE m.meetingID = {meetingID}
      AND m.meetingID = a.meetingID
    """))

    for email, in audience:
      cache.rpush(f'{meeting_instance_key}:audience', email)
```

Function 3: a user joins an active meeting instance – if allowed, i.e. his email is in audience

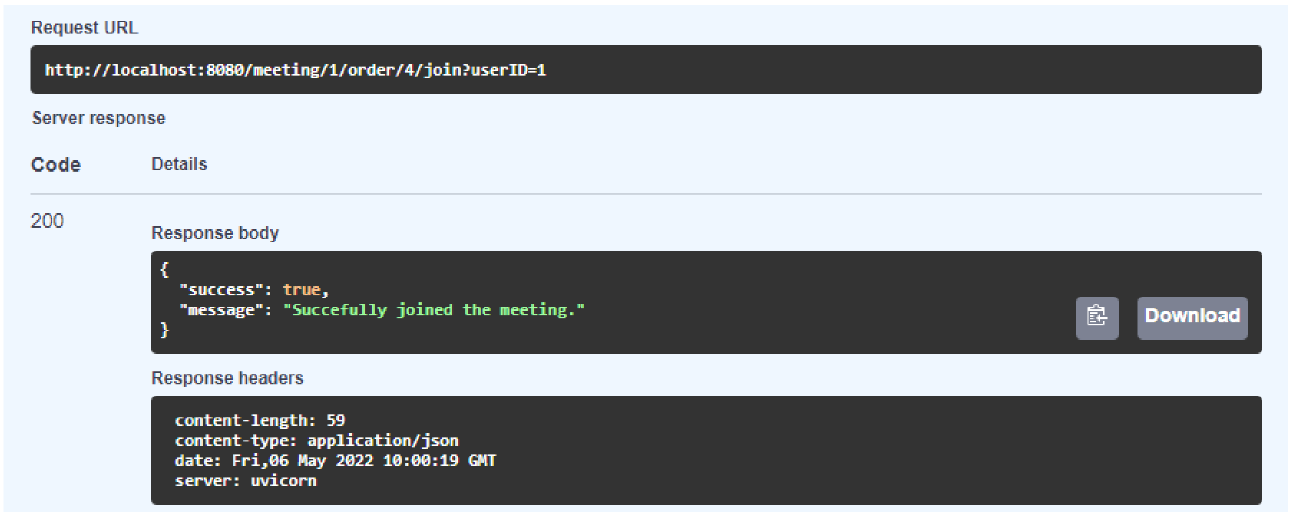
The join_meeting_instance(meetingID: int, orderID: int, userID: int) function is executed whenever someone sends a GET request to the '/meeting/{meetingID}/order/{orderID}/join' endpoint, where meetingID & orderID are path parameters that uniquely identify a meeting instance and the userID is a query parameter which represents the ID of the user that wants to join the meeting.

The function first checks if the meeting exists, if the user is already attending the meeting or if the user is forbidden to join the meeting (his email is not included in the audience list) and if any of those checks raises an error, it responds with the appropriate error message. Otherwise, we append the ID of the user in the meeting's participants list, we create a new log for the user that joined the meeting and we append the eventID of that log to the meeting's logs list.

```
@function.get('/meeting/{meetingID}/order/{orderID}/join')
def join_meeting_instance(meetingID: int, orderID: int, userID: int):
  try:
    meeting_instance_key = f'meeting:{meetingID}:order:{orderID}'

    check_if_meeting_exists(meeting_instance_key)
    check_if_user_already_in_meeting(meeting_instance_key, userID)
    check_if_user_forbidden_to_join_meeting(meeting_instance_key, userID)

    cache.rpush(f'{meeting_instance_key}:participants', userID)
    create_log(meeting_instance_key, userID, EventType.join)

    return {
      "success": True,
      "message": "Succefully joined the meeting."
    }
  except MeetingNotExistOrInactive as exc:
    raise HTTPException(400, str(exc))
  except UserForbiddenToJoinMeeting as exc:
    raise HTTPException(403, str(exc))
  except UserAlreadyInMeeting as exc:
    raise HTTPException(409, str(exc))
  except Exception as exc:
    exception_logger.exception(exc)
    raise HTTPException(500, "Failed to join the meeting.")
```

```python
def check_if_meeting_exists(meeting_instance_key: str):
  exists = cache.exists(meeting_instance_key)
  if not(exists):
    raise MeetingNotExistOrInactive(
      'This meeting either does not exist or it is inactive.')

def check_if_user_already_in_meeting(meeting_instance_key: str, userID: int):
  participants = cache.lrange(f'{meeting_instance_key}:participants', 0, -1)
  if str(userID) in participants:
    raise UserAlreadyInMeeting('You have already joined in the meeting!')

def check_if_user_forbidden_to_join_meeting(meeting_instance_key: str, userID: int):
  isPublic = int(cache.hget(meeting_instance_key, 'isPublic'))
  if not(isPublic):
    email = cache.hget(f'user:{userID}', "email")
    audience = cache.lrange(f'{meeting_instance_key}:audience', 0, -1)
    if email not in audience:
      raise UserForbiddenToJoinMeeting('You are not allowed to join this meeting!')

def create_log(meeting_instance_key: str, userID: int, eventType: EventType):
  eventID = cache.incr('eventID')
  cache.hmset(f'log:{eventID}', {
    "userID": userID,
    "eventType": eventType,
    "timestamp": str(datetime.datetime.now())
  })
  cache.rpush(f'{meeting_instance_key}:logs', eventID)
```

Successful request to join a meeting:

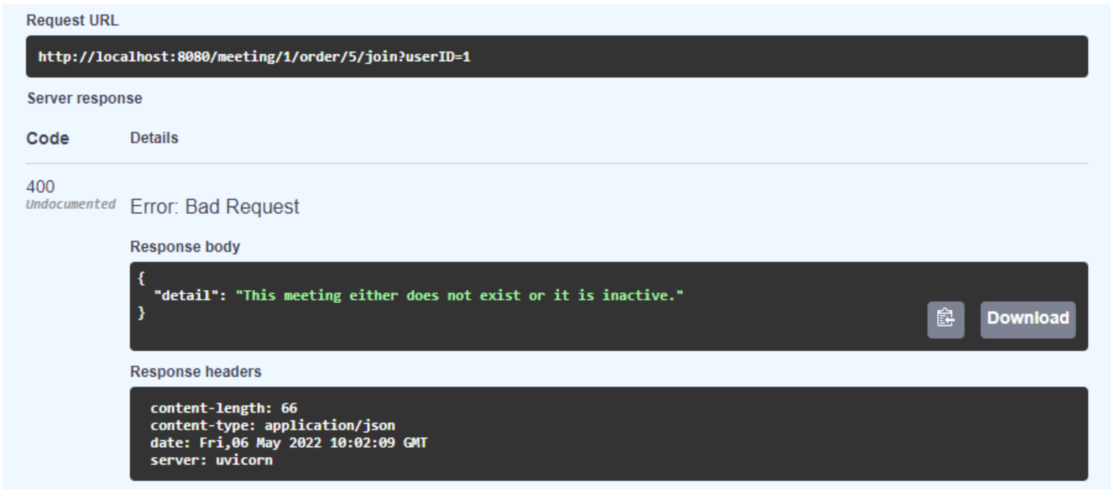


```
http://localhost:8080/meeting/1/order/4/join?userID=1
```

```
{
  "success": true,
  "message": "Succefully joined the meeting."
}
```

```
 content-length: 59
 content-type: application/json
 date: Fri,06 May 2022 10:00:19 GMT
 server: uvicorn
```

Unsuccessful request to join a meeting, because the meeting either does not exist or it is not active:


Request URL

```
http://localhost:8080/meeting/1/order/5/join?userID=1
```

Server response

Code Details

400 *Undocumented* Error: Bad Request

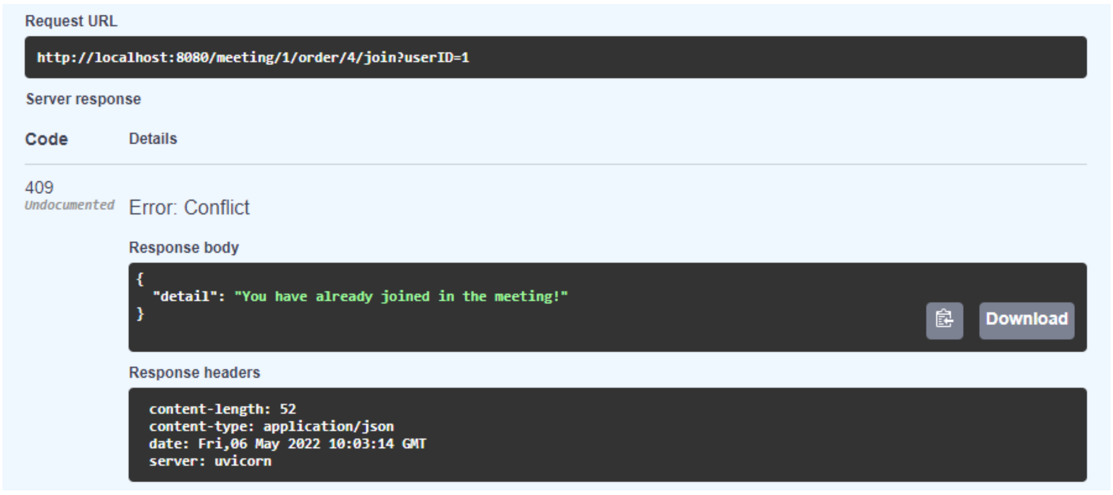
Response body

```
{
  "detail": "This meeting either does not exist or it is inactive."
}
```

Response headers

```
content-length: 66
content-type: application/json
date: Fri,06 May 2022 10:02:09 GMT
server: uvicorn
```

Unsuccessful request to join a meeting, because the user is already attending the meeting:


Request URL

```
http://localhost:8080/meeting/1/order/4/join?userID=1
```

Server response

Code Details

409 *Undocumented* Error: Conflict

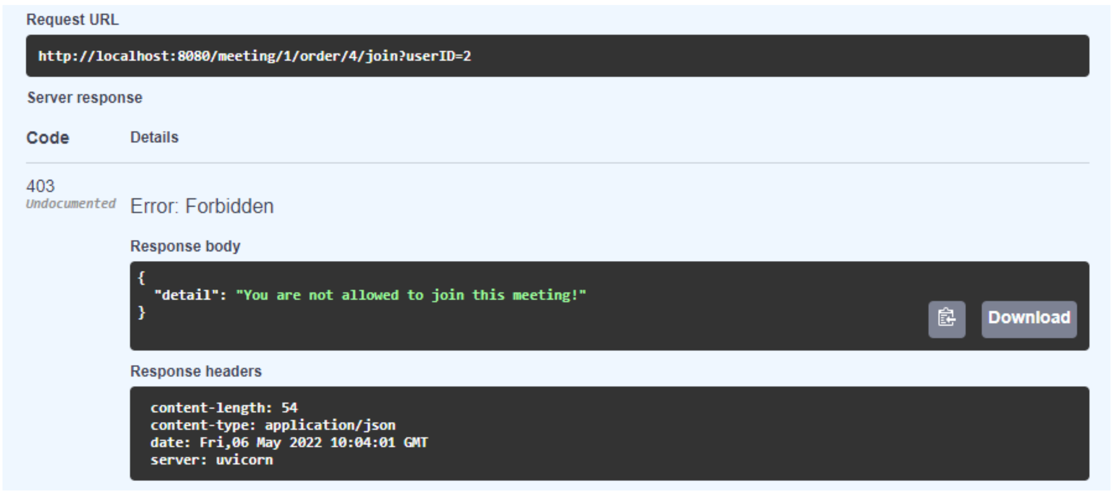
Response body

```
{
  "detail": "You have already joined in the meeting!"
}
```

Response headers

```
content-length: 52
content-type: application/json
date: Fri,06 May 2022 10:03:14 GMT
server: uvicorn
```

Unsuccessful request to join a meeting, because the user is not allowed to join the meeting:


Request URL

```
http://localhost:8080/meeting/1/order/4/join?userID=2
```

Server response

Code Details

403 *Undocumented* Error: Forbidden

Response body

```
{
  "detail": "You are not allowed to join this meeting!"
}
```

Response headers

```
content-length: 54
content-type: application/json
date: Fri,06 May 2022 10:04:01 GMT
server: uvicorn
```

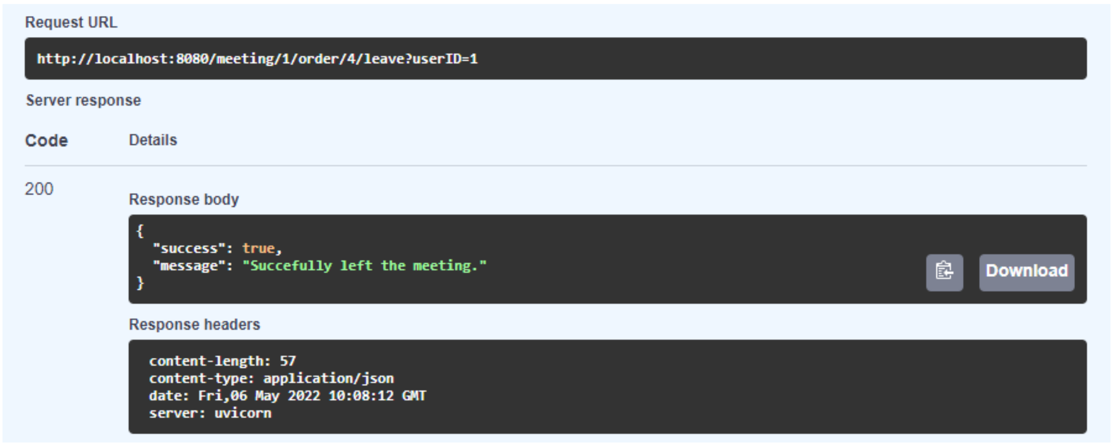
Function 4: a user leaves a meeting that has joined

The leave_meeting_instance(meetingID: int, orderID: int, userID: int) function is executed whenever someone sends a GET request to the '/meeting/{meetingID}/order/{orderID}/leave' endpoint, where meetingID & orderID are path parameters that uniquely identify a meeting instance and the userID is a query parameter which represents the ID of the user that wants to leave from the meeting.

The function first checks if the meeting exists or if the user is not attending the meeting and if any of those checks raises an error, it responds with the appropriate error message. Otherwise, we remove the ID of the user from the meeting's participants list, we create a new log for the user that leaved the meeting and we append the eventID of that log to the meeting's logs list.

```
@function.get('/meeting/{meetingID}/order/{orderID}/leave')
def leave_meeting_instance(meetingID: int, orderID: int, userID: int):
  try:
    meeting_instance_key = f'meeting:{meetingID}:order:{orderID}'

    check_if_meeting_exists(meeting_instance_key)
    check_if_user_not_in_meeting(meeting_instance_key, userID)

    cache.lrem(f'{meeting_instance_key}:participants', 1, userID)
    create_log(meeting_instance_key, userID, EventType.leave)

    return {
      "success": True,
      "message": "Succefully left the meeting."
    }
  except MeetingNotExistOrInactive as exc:
    raise HTTPException(400, str(exc))
  except UserNotInMeeting as exc:
    raise HTTPException(409, str(exc))
  except Exception as exc:
    exception_logger.exception(exc)
    raise HTTPException(500, "Failed to leave the meeting.")



def check_if_user_not_in_meeting(meeting_instance_key: str, userID: int):
  participants = cache.lrange(f'{meeting_instance_key}:participants', 0, -1)
  if str(userID) not in participants:
    raise UserNotInMeeting('You are not attending this meeting.')
```

Successful request to leave from the meeting:


Request URL

```
http://localhost:8080/meeting/1/order/4/leave?userID=1
```

Server response

Code Details

200

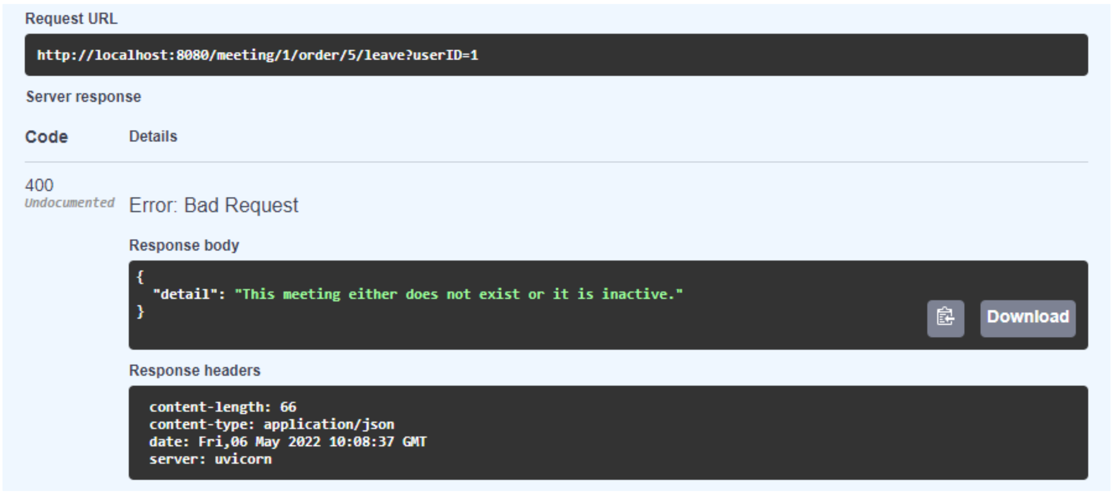
Response body

```
{
  "success": true,
  "message": "Succefully left the meeting."
}
```

Response headers

```
content-length: 57
content-type: application/json
date: Fri,06 May 2022 10:08:12 GMT
server: uvicorn
```

Unsuccessful request to leave from the meeting, because the meeting either does not exist or it is not active:


Request URL

```
http://localhost:8080/meeting/1/order/5/leave?userID=1
```

Server response

Code Details

400
Undocumented Error: Bad Request

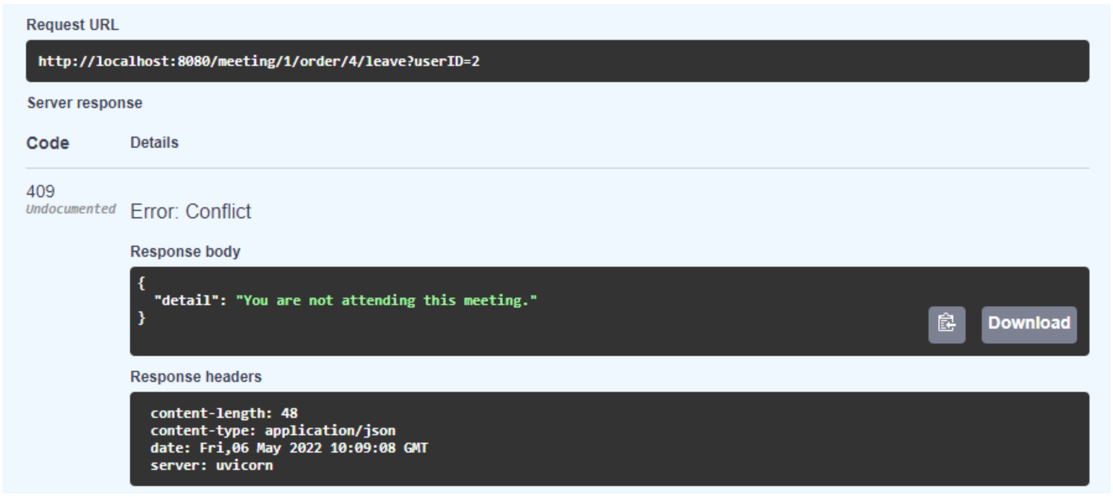
Response body

```
{
  "detail": "This meeting either does not exist or it is inactive."
}
```

Response headers

```
content-length: 66
content-type: application/json
date: Fri,06 May 2022 10:08:37 GMT
server: uvicorn
```

Unsuccessful request to leave the meeting, because the user is not attending the meeting:


Request URL

```
http://localhost:8080/meeting/1/order/4/leave?userID=2
```

Server response

Code Details

409
Undocumented Error: Conflict

Response body

```
{
  "detail": "You are not attending this meeting."
}
```

Response headers

```
content-length: 48
content-type: application/json
date: Fri,06 May 2022 10:09:08 GMT
server: uvicorn
```

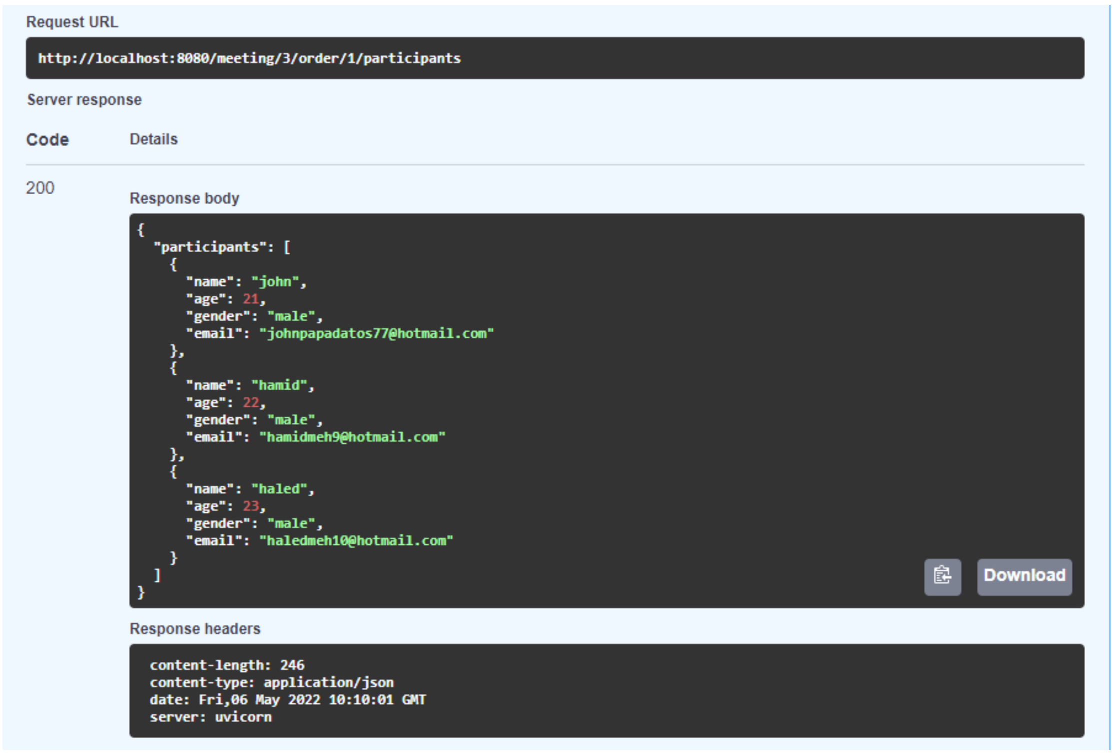
Function 5: show meeting’s current participants

The show_meeting_instance_participants(meetingID: int, orderID: int) function is executed whenever someone sends a GET request to the '/meeting/{meetingID}/order/{orderID}/participants' endpoint, where meetingID & orderID are path parameters that uniquely identify a meeting instance.

The function first checks if the meeting exists and if it raises an error, it responds with the appropriate error message. Otherwise, it creates the response with the details of the meeting’s participants and returns it to the client.

```python
@function.get('/meeting/{meetingID}/order/{orderID}/participants')
def show_meeting_instance_participants(meetingID: int, orderID: int):
  try:
    meeting_instance_key = f'meeting:{meetingID}:order:{orderID}'

    check_if_meeting_exists(meeting_instance_key)

    response = []
    participants = cache.lrange(f'{meeting_instance_key}:participants', 0, -1)
    for userID in participants:
      name, age, gender, email = \
        cache.hmget(f'user:{userID}', ["name", "age", "gender", "email"])

      response.append({
        "name": name,
        "age": int(age),
        "gender": gender,
        "email": email
      })

    return { "participants": response }
  except MeetingNotExistOrInactive as exc:
    raise HTTPException(400, str(exc))
  except Exception as exc:
    exception_logger.exception(exc)
    raise HTTPException(500, "Failed to show the participants of the meeting.")
```

Successful request to get the current participants of the meeting:



Request URL

```
http://localhost:8080/meeting/3/order/1/participants
```

Server response

Code Details

200

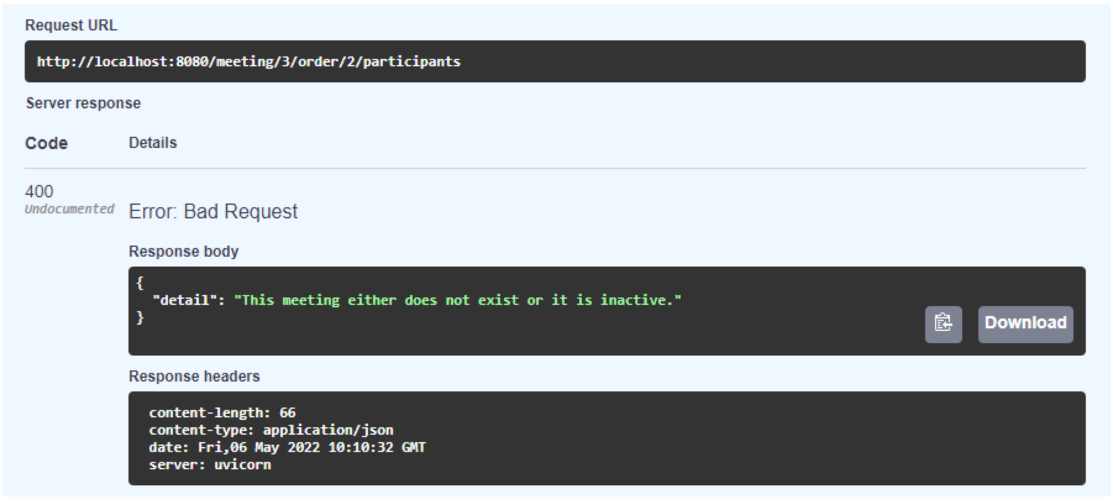
Response body

```
{
  "participants": [
    {
      "name": "john",
      "age": 21,
      "gender": "male",
      "email": "johnpapadatos77@hotmail.com"
    },
    {
      "name": "hamid",
      "age": 22,
      "gender": "male",
      "email": "hamidmeh9@hotmail.com"
    },
    {
      "name": "haled",
      "age": 23,
      "gender": "male",
      "email": "haledmeh10@hotmail.com"
    }
  ]
}
```

Response headers

```
content-length: 246
content-type: application/json
date: Fri,06 May 2022 10:10:01 GMT
server: uvicorn
```

Unsuccessful request to get the current participants of the meeting, because the meeting either does not exist or it is not active:



Request URL

```
http://localhost:8080/meeting/3/order/2/participants
```

Server response

Code Details

400
Undocumented

Error: Bad Request

Response body

```
{
  "detail": "This meeting either does not exist or it is inactive."
}
```

Response headers

```
content-length: 66
content-type: application/json
date: Fri,06 May 2022 10:10:32 GMT
server: uvicorn
```

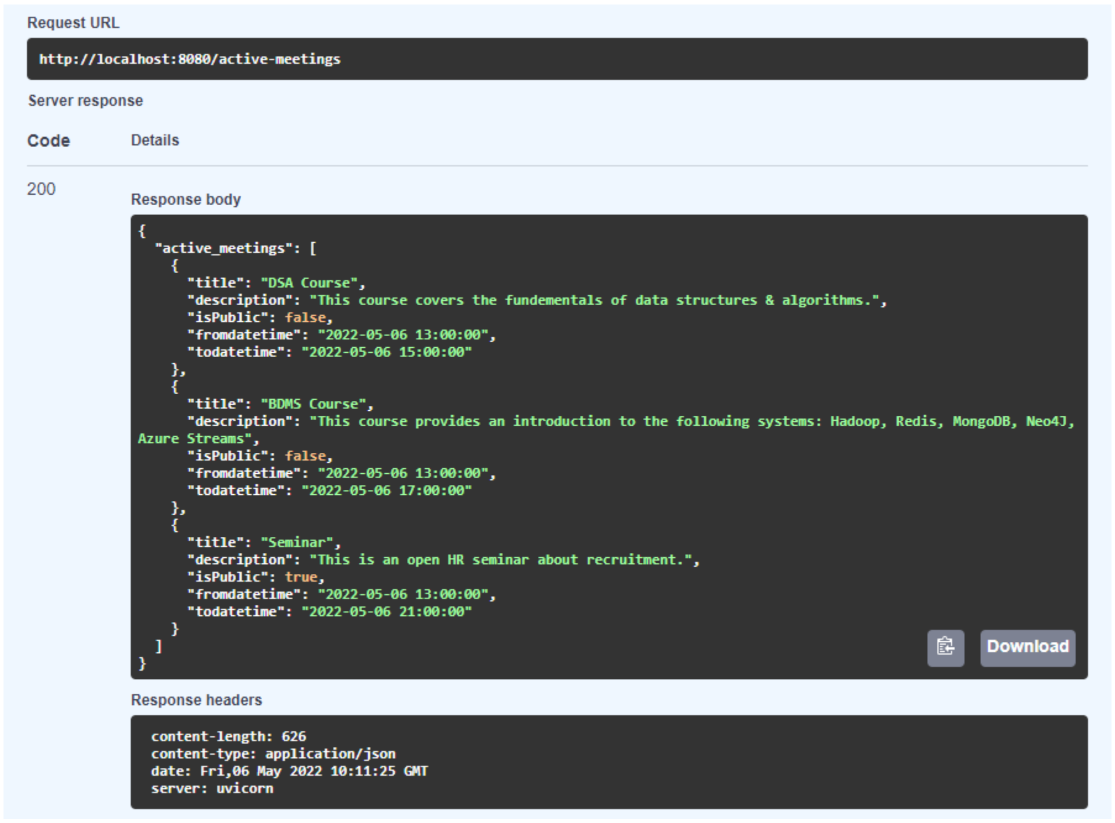
Function 6: show active meetings

The show_active_meetings() function is executed whenever someone sends a GET request to the '/active-meetings' endpoint.

The function creates the response with the details of the active meetings and returns it to the client.

```
@function.get('/active-meetings')
def show_active_meetings():
  try:
    response = []
    active_meetings = cache.smembers('active_meetings')
    for meeting_instance_key in active_meetings:
      title, description, isPublic, start_time, finish_time = \
        cache.hmget(
          meeting_instance_key,
          ["title", "description", "isPublic", "fromdatetime", "todatetime"]
        )

      response.append({
        "title": title,
        "description": description,
        "isPublic": True if int(isPublic) else False,
        "fromdatetime": start_time,
        "todatetime": finish_time
      })

    return { "active_meetings": response }
  except Exception as exc:
    exception_logger.exception(exc)
    raise HTTPException(500, "Failed to show the active meetings.")
```

Successful request to get the active meetings:



```
http://localhost:8080/active-meetings
```

200

```
{
  "active_meetings": [
    {
      "title": "DSA Course",
      "description": "This course covers the fundementals of data structures & algorithms.",
      "isPublic": false,
      "fromdatetime": "2022-05-06 13:00:00",
      "todatetime": "2022-05-06 15:00:00"
    },
    {
      "title": "BDMS Course",
      "description": "This course provides an introduction to the following systems: Hadoop, Redis, MongoDB, Neo4J,
Azure Streams",
      "isPublic": false,
      "fromdatetime": "2022-05-06 13:00:00",
      "todatetime": "2022-05-06 17:00:00"
    },
    {
      "title": "Seminar",
      "description": "This is an open HR seminar about recruitment.",
      "isPublic": true,
      "fromdatetime": "2022-05-06 13:00:00",
      "todatetime": "2022-05-06 21:00:00"
    }
  ]
}
```

```
content-length: 626
content-type: application/json
date: Fri,06 May 2022 10:11:25 GMT
server: uvicorn
```

Function 7: a user can posts a chat message

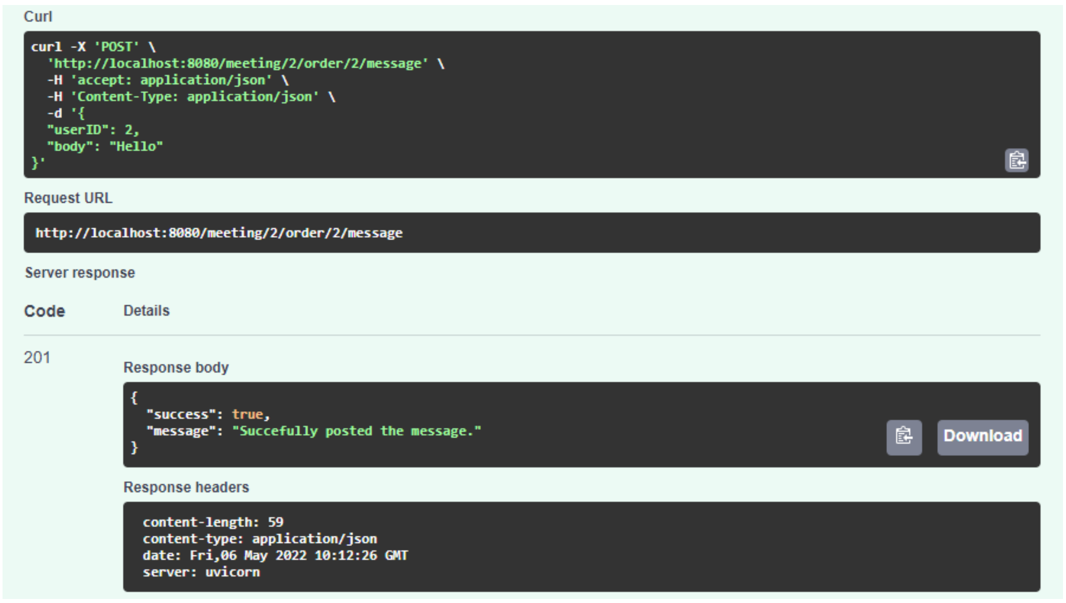
The post_message(message: Message, meetingID: int, orderID: int) function is executed whenever someone sends a POST request to the '/meeting/{meetingID}/order/{orderID}/message' endpoint, where meetingID & orderID are path parameters that uniquely identify a meeting instance. The request body contains the userID of the user that wants to post the message and the body of the message.

The function first checks if the meeting exists or if the user is not attending the meeting and if any of those checks raises an error, it responds with the appropriate error message. Otherwise, we create a new message from the specified user and we append the messageID of that message to the meeting's messages list.

```
@function.post('/meeting/{meetingID}/order/{orderID}/message', status_code=201)
def post_message(message: Message, meetingID: int, orderID: int):
  try:
    meeting_instance_key = f'meeting:{meetingID}:order:{orderID}'

    check_if_meeting_exists(meeting_instance_key)
    check_if_user_not_in_meeting(meeting_instance_key, message.userID)

    create_message(meeting_instance_key, message)

    return {
      "success": True,
      "message": "Succefully posted the message."
    }
  except MeetingNotExistOrInactive as exc:
    raise HTTPException(400, str(exc))
  except UserNotInMeeting as exc:
    raise HTTPException(409, str(exc))
  except Exception as exc:
    exception_logger.exception(exc)
    raise HTTPException(500, "Failed to post the message.")
```

```python
def create_message(meeting_instance_key: str, message: Message):
  messageID = cache.incr('messageID')
  cache.hmset(f'message:{messageID}', {
    "userID": message.userID,
    "body": message.body,
    "timestamp": str(datetime.datetime.now())
  })
  cache.rpush(f'{meeting_instance_key}:messages', messageID)
```

Successful request to post a message to a meeting:



Curl

```
curl -X 'POST' \
  'http://localhost:8080/meeting/2/order/2/message' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
  "userID": 2,
  "body": "Hello"
}'
```

Request URL

```
http://localhost:8080/meeting/2/order/2/message
```

Server response

Code Details

201

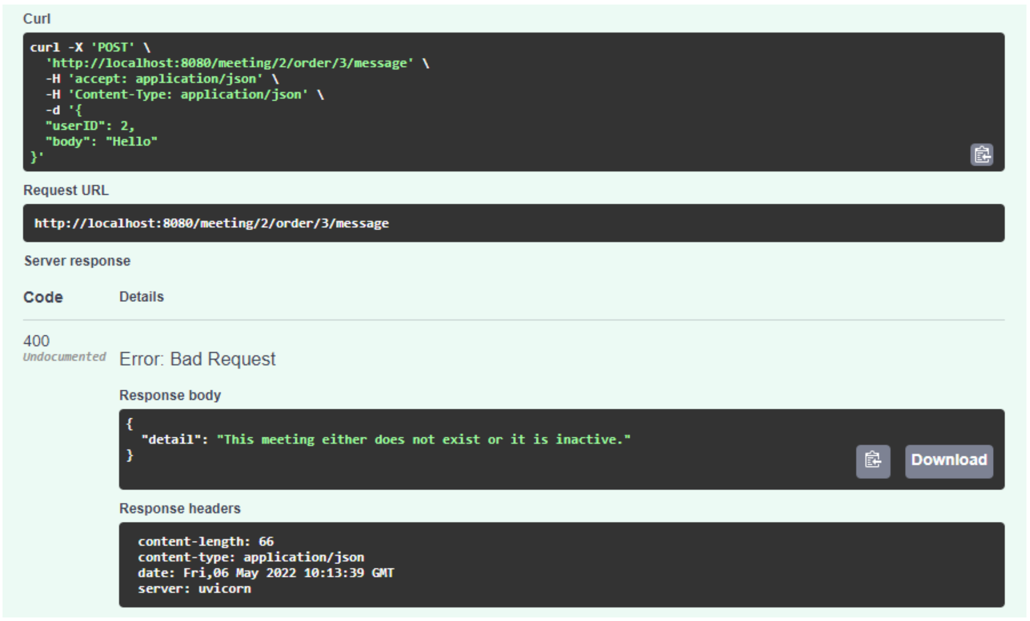
Response body

```
{
  "success": true,
  "message": "Succefully posted the message."
}
```

Response headers

```
content-length: 59
content-type: application/json
date: Fri,06 May 2022 10:12:26 GMT
server: uvicorn
```

Unsuccessful request to post a message to a meeting, because the meeting either does not exist or it is not active:



Curl

```
curl -X 'POST' \
  'http://localhost:8080/meeting/2/order/3/message' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
  "userID": 2,
  "body": "Hello"
}'
```

Request URL

```
http://localhost:8080/meeting/2/order/3/message
```

Server response

Code Details

400 *Undocumented* Error: Bad Request

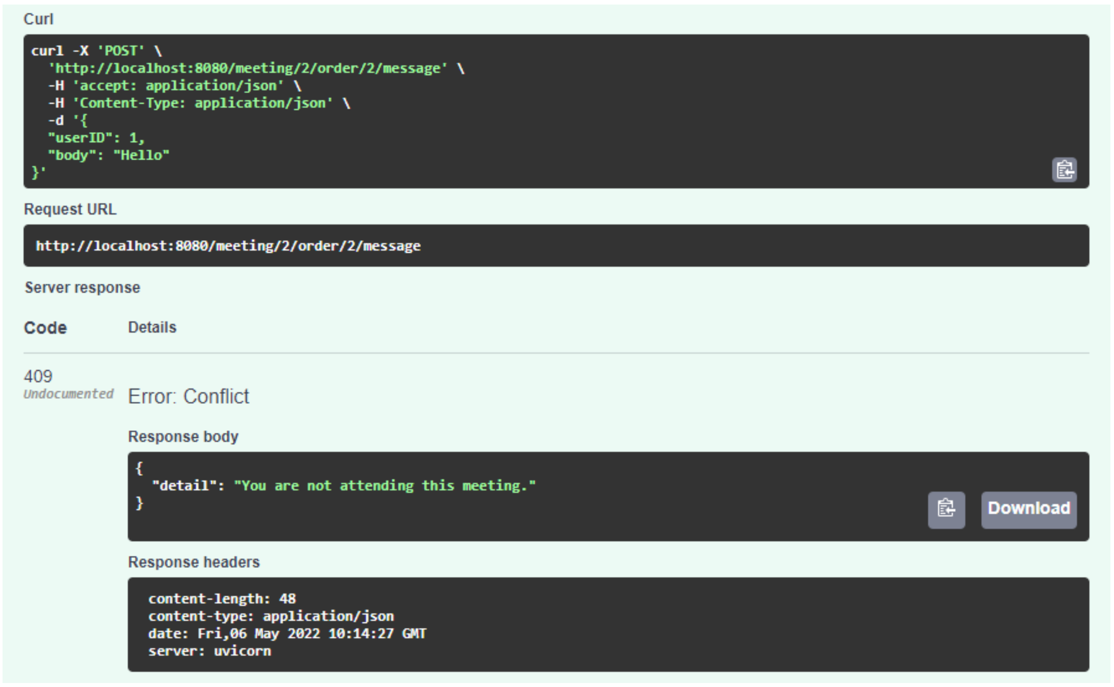
Response body

```
{
  "detail": "This meeting either does not exist or it is inactive."
}
```

Response headers

```
content-length: 66
content-type: application/json
date: Fri,06 May 2022 10:13:39 GMT
server: uvicorn
```

Unsuccessful request to post a message to a meeting, because the user is not attending the meeting:


Curl

```
curl -X 'POST' \
  'http://localhost:8080/meeting/2/order/2/message' \
  -H 'accept: application/json' \
  -H 'Content-Type: application/json' \
  -d '{
  "userID": 1,
  "body": "Hello"
}'
```

Request URL

```
http://localhost:8080/meeting/2/order/2/message
```

Server response

Code Details

409
Undocumented

Error: Conflict

Response body

```
{
  "detail": "You are not attending this meeting."
}
```

Response headers

```
 content-length: 48
 content-type: application/json
 date: Fri,06 May 2022 10:14:27 GMT
 server: uvicorn
```

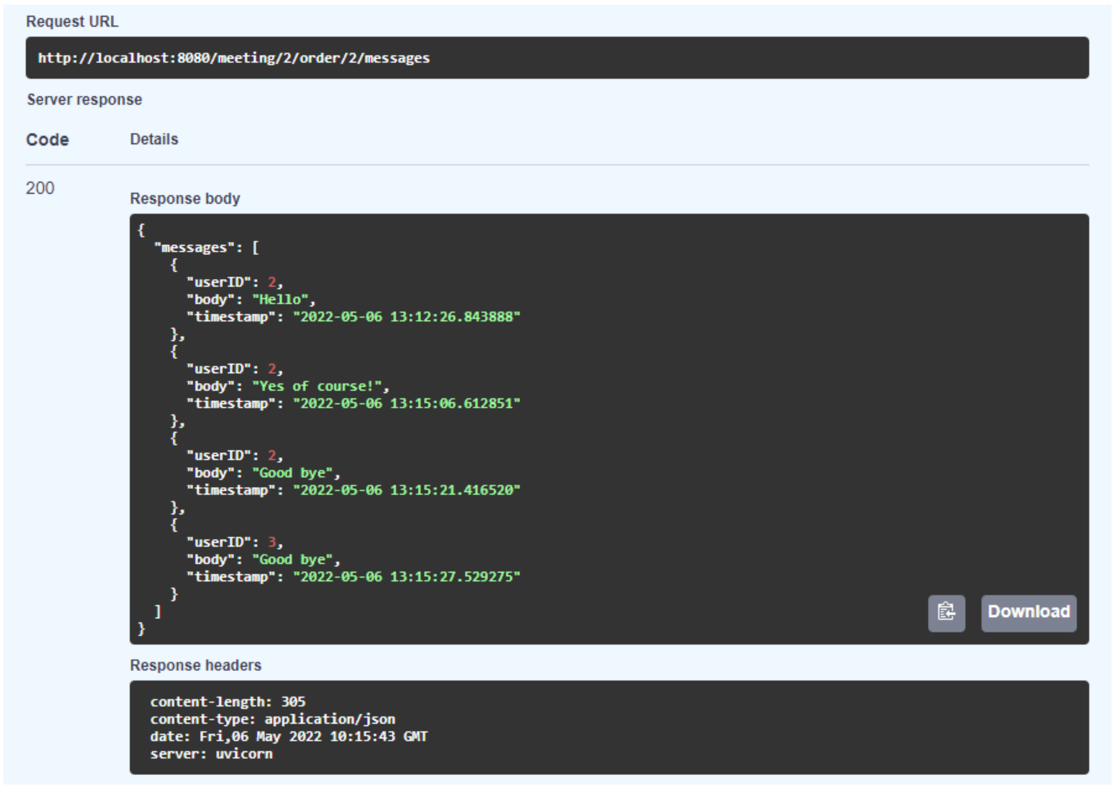
Function 8: show meeting’s chat messages in chronological order

The show_meeting_instance_messages(meetingID: int, orderID: int) function is executed whenever someone sends a GET request to the '/meeting/{meetingID}/order/{orderID}/messages' endpoint, where meetingID & orderID are path parameters that uniquely identify a meeting instance.

The function first checks if the meeting exists and if it raises an error, it responds with the appropriate error message. Otherwise, it creates the response with the details of the meeting’s messages and returns it to the client.

```
"""
The chronological order requirement is achieved automatically because lists are
ordered collections (they maintain the insertion order) & we are only appending
elements at the end of the list.
"""
@function.get('/meeting/{meetingID}/order/{orderID}/messages')
def show_meeting_instance_messages(meetingID: int, orderID: int):
  try:
    meeting_instance_key = f'meeting:{meetingID}:order:{orderID}'

    check_if_meeting_exists(meeting_instance_key)

    response = []
    messages = cache.lrange(f'{meeting_instance_key}:messages', 0, -1)
    for messageID in messages:
      userID, body, timestamp = \
        cache.hmget(f'message:{messageID}', ["userID", "body", "timestamp"])

      response.append({
        "userID": int(userID),
        "body": body,
        "timestamp": timestamp
      })

    return { "messages": response }
  except MeetingNotExistOrInactive as exc:
    raise HTTPException(400, str(exc))
  except Exception as exc:
    exception_logger.exception(exc)
    raise HTTPException(500, "Failed to show the messages of the meeting.")
```

Successful request to get the messages of the meeting:



Request URL

```
http://localhost:8080/meeting/2/order/2/messages
```

Server response

Code Details

200

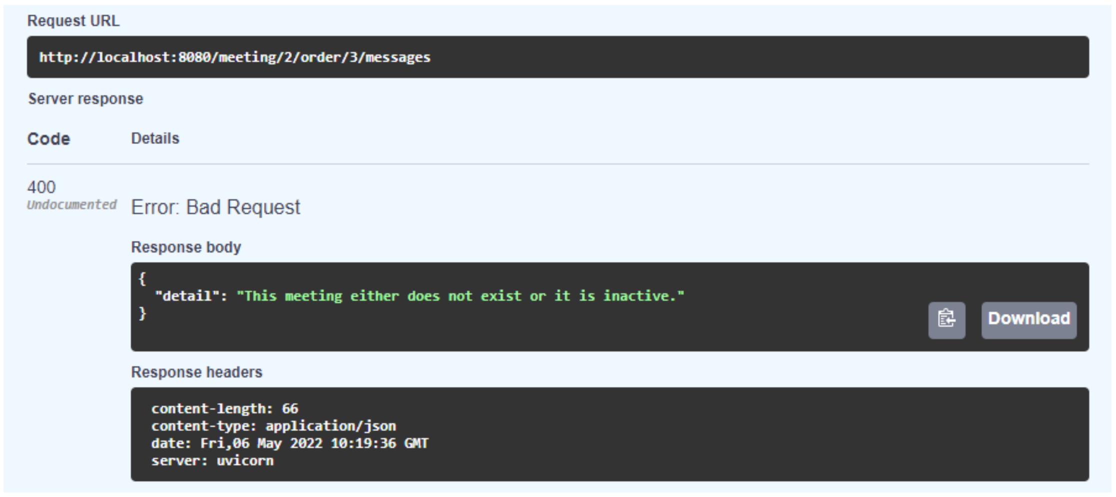
Response body

```
{
  "messages": [
    {
      "userID": 2,
      "body": "Hello",
      "timestamp": "2022-05-06 13:12:26.843888"
    },
    {
      "userID": 2,
      "body": "Yes of course!",
      "timestamp": "2022-05-06 13:15:06.612851"
    },
    {
      "userID": 2,
      "body": "Good bye",
      "timestamp": "2022-05-06 13:15:21.416520"
    },
    {
      "userID": 3,
      "body": "Good bye",
      "timestamp": "2022-05-06 13:15:27.529275"
    }
  ]
}
```

Response headers

```
content-length: 305
content-type: application/json
date: Fri,06 May 2022 10:15:43 GMT
server: uvicorn
```

Unsuccessful request to get the messages of the meeting, because the meeting either does not exist or it is not active:



Request URL

```
http://localhost:8080/meeting/2/order/3/messages
```

Server response

Code Details

400
Undocumented

Error: Bad Request

Response body

```
{
  "detail": "This meeting either does not exist or it is inactive."
}
```

Response headers

```
content-length: 66
content-type: application/json
date: Fri,06 May 2022 10:19:36 GMT
server: uvicorn
```

Function 9: show for each active meeting when (timestamp) current participants joined

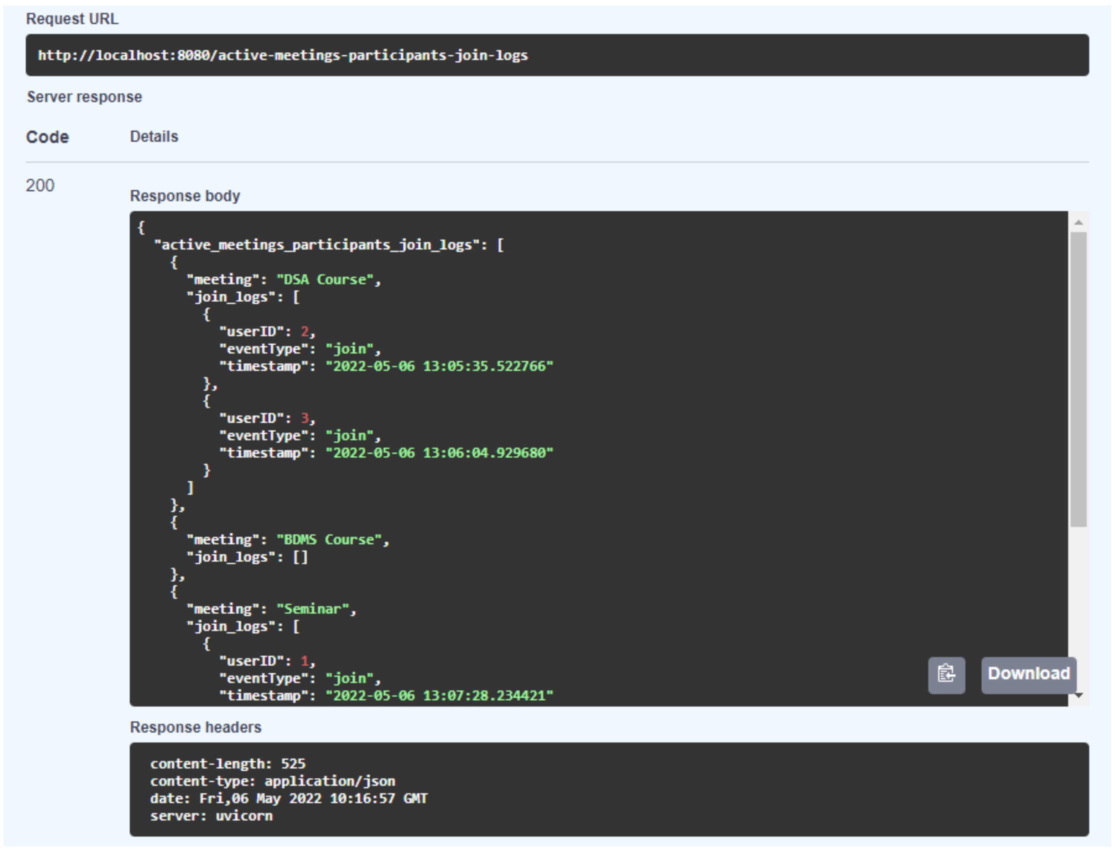
The show_active_meetings_participants_join_logs() function is executed whenever someone sends a GET request to the '/active-meetings-participants-join-logs' endpoint.

The function creates the response with the title of each active meeting along with the details of the join logs of its participants and returns it to the client.

```
@function.get('/active-meetings-participants-join-logs')
def show_active_meetings_participants_join_logs():
  try:
    response = []
    active_meetings = cache.smembers('active_meetings')
    for meeting_instance_key in active_meetings:
      title = cache.hget(meeting_instance_key, "title")
      participants = cache.lrange(f'{meeting_instance_key}:participants', 0, -1)

      join_logs = []
      logs = cache.lrange(f'{meeting_instance_key}:logs', 0, -1)
      for eventID in logs:
        userID, eventType, timestamp = \
          cache.hmget(f'log:{eventID}', ["userID", "eventType", "timestamp"])

        if userID in participants and eventType == EventType.join:
          join_logs.append({
            "userID": int(userID),
            "eventType": eventType,
            "timestamp": timestamp
          })

      keep_latest_join_log_for_each_user(join_logs)

      response.append({
        "meeting": title,
        "join_logs": join_logs
      })

    return { "active_meetings_participants_join_logs": response }
  except Exception as exc:
    exception_logger.exception(exc)
    raise HTTPException(
      500, "Failed to show the join logs of the participants in the active meetings.")
```

```python
def keep_latest_join_log_for_each_user(join_logs: List[Dict[str, Union[int, str]]]):
  idx_to_del = []
  userID_to_idx = dict()
  for idx, join_log in enumerate(join_logs[:]):
    userID = join_log["userID"]
    if userID in userID_to_idx:
      idx_to_del.append(userID_to_idx[userID])
    userID_to_idx[userID] = idx

  idx_to_del.sort(reverse=True)
  for idx in idx_to_del:
    del join_logs[idx]
```

Successful request to get the join logs from the participants of the active meetings:



Request URL

```
http://localhost:8080/active-meetings-participants-join-logs
```

Server response

Code Details

200

Response body

```
{
  "active_meetings_participants_join_logs": [
    {
      "meeting": "DSA Course",
      "join_logs": [
        {
          "userID": 2,
          "eventType": "join",
          "timestamp": "2022-05-06 13:05:35.522766"
        },
        {
          "userID": 3,
          "eventType": "join",
          "timestamp": "2022-05-06 13:06:04.929680"
        }
      ]
    },
    {
      "meeting": "BDMS Course",
      "join_logs": []
    },
    {
      "meeting": "Seminar",
      "join_logs": [
        {
          "userID": 1,
          "eventType": "join",
          "timestamp": "2022-05-06 13:07:28.234421"
```

Download

Response headers

```
content-length: 525
content-type: application/json
date: Fri,06 May 2022 10:16:57 GMT
server: uvicorn
```

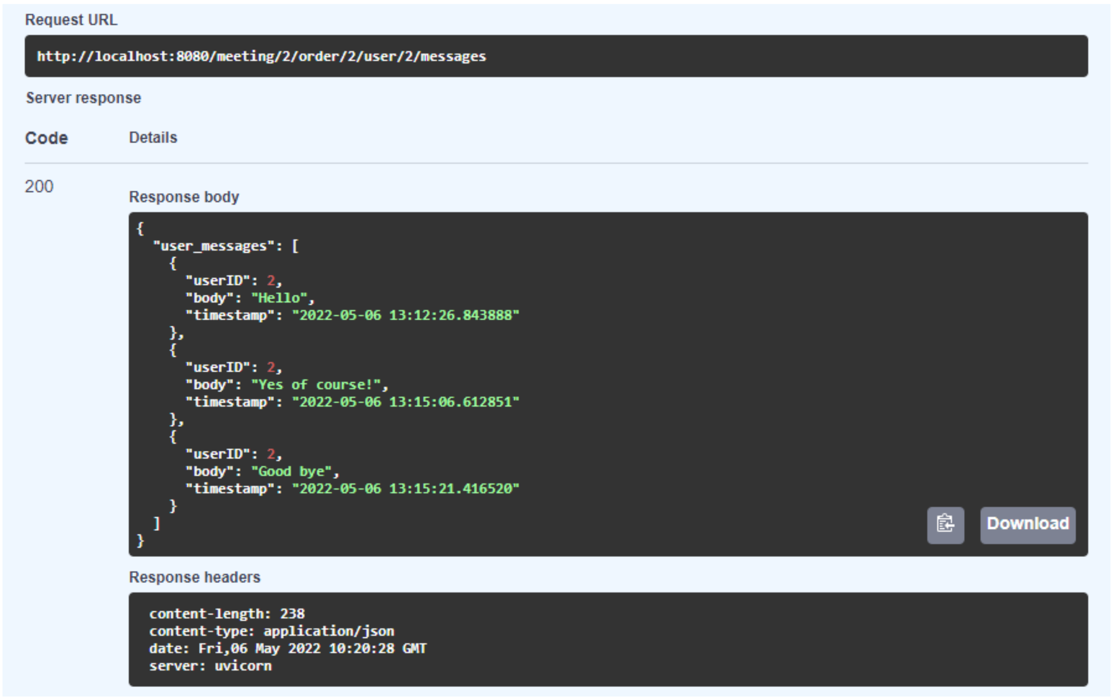
Function 10: show for an active meeting and a user his/her chat messages

The show_meeting_instance_user_messages(meetingID: int, orderID: int, userID: int) function is executed whenever someone sends a GET request to the '/meeting/{meetingID}/order/{orderID}/user/{userID}/messages' endpoint, where meetingID & orderID are path parameters that uniquely identify a meeting instance & userID is also a path parameter which represents the ID of the user whose messages we want to show.

The function first checks if the meeting exists and if it raises an error, it responds with the appropriate error message. Otherwise, it creates the response with the details of the meeting’s messages posted by the specified user and returns it to the client.

```
@function.get('/meeting/{meetingID}/order/{orderID}/user/{userID}/messages')
def show_meeting_instance_user_messages(meetingID: int, orderID: int, userID: int):
  try:
    meeting_instance_key = f'meeting:{meetingID}:order:{orderID}'

    check_if_meeting_exists(meeting_instance_key)

    response = []
    messages = cache.lrange(f'{meeting_instance_key}:messages', 0, -1)
    for messageID in messages:
      msg_userID, body, timestamp = \
        cache.hmget(f'message:{messageID}', ["userID", "body", "timestamp"])

      if str(userID) == msg_userID:
        response.append({
          "userID": int(msg_userID),
          "body": body,
          "timestamp": timestamp
        })

    return { "user_messages": response }
  except MeetingNotExistOrInactive as exc:
    raise HTTPException(400, str(exc))
  except Exception as exc:
    exception_logger.exception(exc)
    raise HTTPException(500, "Failed to show the messages of the user in the meeting.")
```

Successful request to get the messages posted by the specified user to the meeting:



Request URL

```
http://localhost:8080/meeting/2/order/2/user/2/messages
```

Server response

Code Details

200

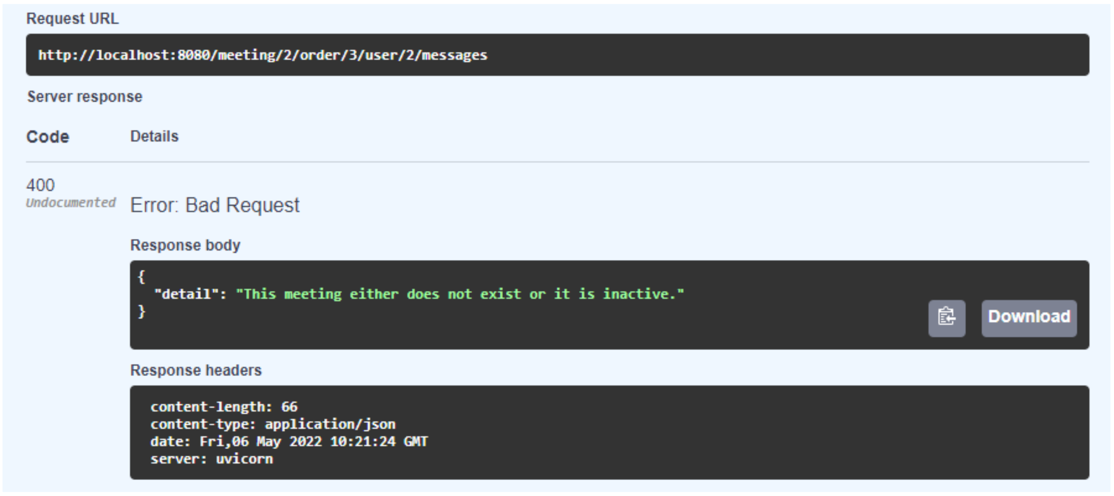
Response body

```
{
  "user_messages": [
    {
      "userID": 2,
      "body": "Hello",
      "timestamp": "2022-05-06 13:12:26.843888"
    },
    {
      "userID": 2,
      "body": "Yes of course!",
      "timestamp": "2022-05-06 13:15:06.612851"
    },
    {
      "userID": 2,
      "body": "Good bye",
      "timestamp": "2022-05-06 13:15:21.416520"
    }
  ]
}
```

Response headers

```
content-length: 238
content-type: application/json
date: Fri,06 May 2022 10:20:28 GMT
server: uvicorn
```

Unsuccessful request to get the messages posted by the specified user to the meeting, because the meeting either does not exist or it is not active:



Request URL

```
http://localhost:8080/meeting/2/order/3/user/2/messages
```

Server response

Code Details

400 *Undocumented* Error: Bad Request

Response body

```
{
  "detail": "This meeting either does not exist or it is inactive."
}
```

Response headers

```
content-length: 66
content-type: application/json
date: Fri,06 May 2022 10:21:24 GMT
server: uvicorn
```

References

- https://redis.io/

- https://www.mysql.com/

- https://fastapi.tiangolo.com/

- https://www.docker.com/