ACIT 3855 – Lab 5 – Processing Service

Instructor	Mike Mulder (mmulder10@bcit.ca) – Sets A and C Tim Guicherd (tguicherd@bcit.ca) – Set B
Total Marks	10
Due Dates	Demo and submission by the end of next class: • Feb. 15th for Set C • Feb. 17th for Sets A and B

Purpose

- Create a new Processing Service that leverages periodic processing, logging and external configuration.
- The Processing Service will store its data in a SQLite database file.
- Add GET endpoints to both the Storage and Processing services.

Part 1 – Storage Service Updates

Add two new GET endpoints to your Storage Service, each with a timestamp parameter:

1. GET /<Event 1 Type>?start_timestamp=<datetime string>&end_timestamp=<datetime string>
2. GET /<Event 2 Type>?start_timestamp=<datetime string>&end_timestamp=<datetime string>

Each should return a JSON array of all events of the given type whose date_created value is on or after the values provided in the timestamp parameter. If there are none, an empty array should be returned.

You will need to:

- Update your openapi.yml file with a GET endpoint for each event.
    - Parameter – start_timestamp
    - Parameter – end_timestamp
    - Response – Array of your event objects.

Here is an example of what you need to add to the path for each event type (but modified to reflect your event type):

```yaml
get:
  tags:
    - devices
  summary: gets new blood pressure readings
  operationId: app.get_blood_pressure_readings
  description: Gets blood pressure readings added after a timestamp
  parameters:
    - name: start_timestamp
      in: query
      description: Limits the number of readings returned
      schema:
        type: string
        format: date-time
        example: 2016-08-29T09:12:33.001Z
    - name: end_timestamp
      in: query
      description: Limits the number of readings returned
      schema:
        type: string
        format: date-time
        example: 2016-08-29T09:12:33.001Z
```

```
  responses:
    '200':
      description: Successfully returned a list of blood pressure events
      content:
        application/json:
          schema:
            type: array
            items:
              $ref: '#/components/schemas/BloodPressureReading'
    '400':
      description: Invalid request
      content:
        application/json:
          schema:
            type: object
            properties:
              message:
                type: string
```

- Create two new functions in your app.py that are mapped to your GET endpoints.

Here is an example of what your code could look like (you will need to import datetime):

```
def get_blood_pressure_readings(start_timestamp, end_timestamp):
    """ Gets new blood pressure readings between the start and end timestamps """

    session = DB_SESSION()

    start_timestamp_datetime =
        datetime.datetime.strptime(start_timestamp, "%Y-%m-%dT%H:%M:%S")
    end_timestamp_datetime =
        datetime.datetime.strptime(end_timestamp, "%Y-%m-%dT%H:%M:%S")


    results = session.query(BloodPressure).filter(
        and_(BloodPressure.date_created >= start_timestamp_datetime,
             BloodPressure.date_created < end_timestamp_datetime))

    results_list = []

    for reading in readings:
        results_list.append(reading.to_dict())

    session.close()

    logger.info("Query for Blood Pressure readings after %s returns %d results" %
                (start_timestamp, len(results_list)))

    return results_list, 200
```

Make sure your to_dict function in your SQLAlchemy declaratives (i.e., the event object) produce data that matches your event schemas in the OpenAPI specification otherwise your service will return 500 response codes.

Part 2 – Stub Out New Processing Service

Use your previous labs as a reference. This Processing Service will get the newest events from your Storage Service and generate and store statistics on the events. You must have **four statistics**, two of which can be the number of Event1 and Event2 type events received and the other two based on the numeric values in your APIs.

- Create a new project for Lab 5 in your IDE, but also keep the projects for Lab 2 (Receiver Service) and Lab 3 (Storage Service) open as you will need it for overall testing.

- Copy over the openapi.yml and app.py files from your one of your existing services as your starting point.
- You'll need to install the following packages:
    - connexion
    - swagger-ui-bundle
    - requests
    - apscheduler-bundle
- Modify the openapi.yml for your new processing service. It should have one GET endpoint that returns your statistics. Here is an example:

```yaml
openapi: 3.0.0
info:
  description: This API provides event stats
  version: "1.0.0"
  title: Stats API
  contact:
    email: mmulder10@bcit.ca

paths:
  /stats:
    get:
      summary: Gets the event stats
      operationId: app.get_stats
      description: Gets Blood Pressure and Heart Rate processsed statistics
      responses:
        '200':
          description: Successfully returned a list of blood pressure events
          content:
            application/json:
              schema:
                type: object
                items:
                  $ref: '#/components/schemas/ReadingStats'
        '400':
          description: Invalid request
          content:
            application/json:
              schema:
                type: object
                properties:
                  message:
                    type: string

components:
  schemas:
    ReadingStats:
      required:
      - num_bp_readings
      - max_bp_sys_reading
      - max_bp_dia_reading
      - num_hr_readings
      - max_hr_reading
      properties:
        num_bp_readings:
          type: integer
          example: 500000
        max_bp_sys_reading:
          type: integer
          example: 200
        max_bp_dia_reading:
          type: integer
          example: 180
        num_hr_readings:
          type: integer
          example: 500000
```

```
    max_hr_reading:
      type: integer
      example: 250
  type: object
```

You can use SwaggerHub to edit and validate the file before copying into your Lab 5 project.

- Modify the app.py as follows:
  - Remove any obsolete methods and add a methods for the new GET method
  - Change the port used for this service. The Receiver Service should use port 8080, the Data Storage Service port 8090 and this service port 8100. This allows them to all run concurrently on your laptop without conflicting on ports.

Part 3 – Configuration

Create a new YAML file in your Lab 5 project called app_conf.yml. It should look something like this:

```
version: 1
datastore:
  filename: stats.sqlite
scheduler:
  period_sec: 5
eventstore:
  url: http://localhost:8090
```

Load the configuration into your app.py file as follows:

```
with open('app_conf.yaml', 'r') as f:
    app_config = yaml.safe_load(f.read())
```

Your configuration is now available in app_config which is a Python dictionary.

Make sure you have imported yaml into your app.py file.

Part 4 – Logging

Create a new YAML file in your Lab 5 project called log_conf.yml (or copy from an existing service). It should look something like this:

```
version: 1
formatters:
  simple:
    format: '%(asctime)s - %(name)s - %(levelname)s - %(message)s'
handlers:
  console:
    class: logging.StreamHandler
    level: DEBUG
    formatter: simple
    stream: ext://sys.stdout
  file:
    class: logging.FileHandler
    level: DEBUG
    formatter: simple
    filename: app.log
```

```
loggers:
  basicLogger:
    level: DEBUG
    handlers: [console, file]
    propagate: no
root:
  level: DEBUG
  handlers: [console]
```

Load the configuration into your app.py file as follows:

```
with open('log_conf.yaml', 'r') as f:
    log_config = yaml.safe_load(f.read())
    logging.config.dictConfig(log_config)
```

Create a logger from the basicLogger defined in the configuration file.

```
logger = logging.getLogger('basicLogger')
```

Use this to create log messsages. All log message on logger will be written to the file app.log.

Make sure you have imported yaml and logging.config in your app.py file.

Part 5 – Periodic Processing

Stub out a method called populate_stats() in app.py:

```
def populate_stats():
    """ Periodically update stats """
    pass
```

Schedule it to be called periodically based on your 'periodic_sec' interval from your configuration file:

```
def init_scheduler():
    sched = BackgroundScheduler(daemon=True)
    sched.add_job(populate_stats,
                  'interval',
                  seconds=app_config['scheduler']['period_sec'])
    sched.start()
```

Make sure to call this function here:

```
if __name__ == "__main__":
    # run our standalone gevent server
    init_scheduler()
    app.run(port=8100, use_reloader=False)
```

The populate_stats method will now be called every 5 seconds. Test to make sure this works. Replace the 'pass' with a log message, i.e., logger.info("Start Periodic Processing").

Implement populate_stats as per the following pseudo code:

- Log an INFO message indicating periodic processing has started

- Read in the current statistics from the SQLite database (filename defined in your configuration)
    - If no stats yet exist, use default values for the stats
- Get the current datetime
- Query the two GET endpoints from your Data Store Service (using requests.get) to get all new events from the last datetime you requested them (from your statistics) to the current datetime
    - Log an INFO message with the number of events received
    - Log an ERROR message if you did not get a 200 response code
- Based on the new events from the Data Store Service:
    - Calculate your updated statistics
    - **Log a DEBUG message for each event processed that includes the trace_id**
    - Write the updated statistics to the SQLite database file (filename defined in your configuration)
    - Log a DEBUG message with your updated statistics values
- Log an INFO message indicating period processing has ended

The table in your SQLite database (stats.sqlite) should have a row added each time the populate_stats function is called. The row should include a column for each statistics your are tracking plus the last_update datetime stamp. See the notes at the end of this lab for what your create_tables.py script and SQLAlchemy declaritive might look like.

Note that values in the SQLite database for the row containing the latest stats should correspond to the values in the JSON response from your GET /stats endpoint.

Make sure to import the BackgroundScheduler (i.e., from apscheduler.schedulers.background import BackgroundScheduler), json, requests and datetime modules into your app.py file.

Part 6 – GET API Implementation

Implement the GET endpoint for your /events/stats resource in your Lab 5 app.py file as per the following pseudo code. Note that the function name must match the name you defined in your openapi.yml file (i.e., get_stats)

- Log an INFO message indicating request has started
- Read in the current statistics from the SQLite database (i.e., the row with the most recent last_update datetime stamp.
    - If no stats exist, log an ERROR message and return 404 and the message “Statistics do not exist” OR return empty/default statistics
- Convert them as necessary into a new Python dictionary such that the structure matches that of your response defined in the openapi.yaml file.
- Log a DEBUG message with the contents of the Python Dictionary
- Log an INFO message indicating request has completed
- Return the Python dictionary as the context and 200 as the response code

Part 6 – Testing

Run all three of your services: Receiver, Storage and Processing.

Use your jMeter test script to load events into the Storage Service through the Receiver Service. Verify that your Processing Service is periodically updating its stats while the test is running:

- View the contents of the SQLite database using the DB Browser for SQLite.
- Exercise the GET endpoint of your Processing Service to get the current statistics

You will demo this next class.

Grading and Submission

Submit the following to the Lab 5 Dropbox on D2L:

- A zipfile containing the code for your processing service.

Demo the following to your instructor before the end of next class to receive your marks:

Processing Service (1 mark each) • openapi.yaml file • app_conf.yaml file • log_conf.yaml file • app.py file	4 marks
Storage Service • Two new GET endpoints	1 mark
Run all three services (Receiver, Storage, Processing). Exercise your jMeter test against the Receiver service. While the jMeter test is running, demo the statistics processed by the Processing Service in the following ways: • SQLite file on disk • Through the GET endpoint	5 marks
Total	**10 marks**

Notes for Setting up the SQLite Database and Declarative

Here is a sample create_tables.py script to create a SQLite database with a stats table:

```
import sqlite3

conn = sqlite3.connect('stats.sqlite')

c = conn.cursor()
c.execute('''
          CREATE TABLE stats
          (id INTEGER PRIMARY KEY ASC,
           num_bp_readings INTEGER NOT NULL,
           num_hr_readings INTEGER NOT NULL,
```

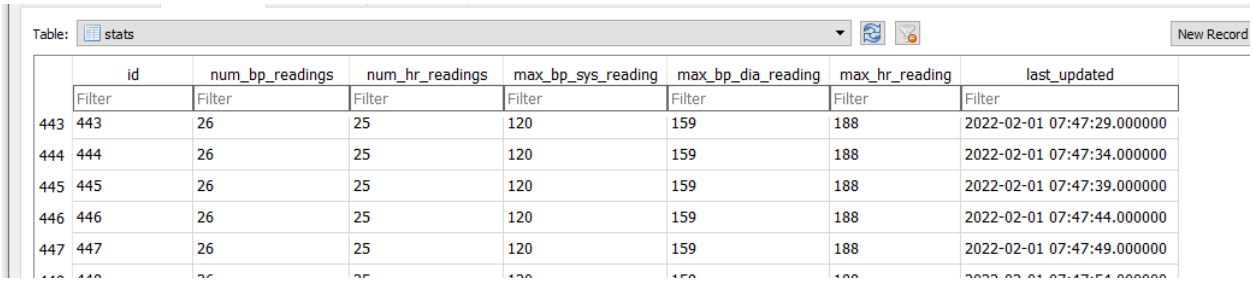
```
            max_bp_sys_reading INTEGER,
            max_bp_dia_reading INTEGER,
            max_hr_reading INTEGER,
            last_updated VARCHAR(100) NOT NULL)
           ''')

conn.commit()
conn.close()
```

Here is a sample Stats Declartive class (stats.py):

```
from sqlalchemy import Column, Integer, String, DateTime
from base import Base
class Stats(Base):
    """ Processing Statistics """

    __tablename__ = "stats"

    id = Column(Integer, primary_key=True)
    num_bp_readings = Column(Integer, nullable=False)
    num_hr_readings = Column(Integer, nullable=False)
    max_bp_sys_reading = Column(Integer, nullable=True)
    max_bp_dia_reading = Column(Integer, nullable=True)
    max_hr_reading = Column(Integer, nullable=True)
    last_updated = Column(DateTime, nullable=False)

    def __init__(self, num_bp_readings, num_hr_readings,
max_bp_sys_reading, max_bp_dia_reading, max_hr_reading,
last_updated):
        """ Initializes a processing statistics objet """
        self.num_bp_readings = num_bp_readings
        self.num_hr_readings = num_hr_readings
        self.max_bp_sys_reading = max_bp_sys_reading
        self.max_bp_dia_reading = max_bp_dia_reading
        self.max_hr_reading = max_hr_reading
        self.last_updated = last_updated

    def to_dict(self):
        """ Dictionary Representation of a statistics """
        dict = {}
        dict['num_bp_readings'] = self.num_bp_readings
        dict['num_hr_readings'] = self.num_hr_readings
        dict['max_bp_sys_reading'] = self.max_bp_sys_reading
        dict['max_bp_dia_reading'] = self.max_bp_dia_reading
        dict['max_hr_reading'] = self.max_hr_reading
        dict['last_updated'] = self.last_updated.strftime("%Y-
%m-%dT%H:%M:%S")
```

```
        return dict
```

Here is what the corresponding database table might look like in the DB Browser for SQLite:



	id	num_bp_readings	num_hr_readings	max_bp_sys_reading	max_bp_dia_reading	max_hr_reading	last_updated
	Filter	Filter	Filter	Filter	Filter	Filter	Filter
443	443	26	25	120	159	188	2022-02-01 07:47:29.000000
444	444	26	25	120	159	188	2022-02-01 07:47:34.000000
445	445	26	25	120	159	188	2022-02-01 07:47:39.000000
446	446	26	25	120	159	188	2022-02-01 07:47:44.000000
447	447	26	25	120	159	188	2022-02-01 07:47:49.000000

You'll need to create a database engine and sessionmaker in your app.py so you can create database sessions to query the database:

```
DB_ENGINE = create_engine("sqlite:///%s" %
app_config["datastore"]["filename"])
Base.metadata.bind = DB_ENGINE
DB_SESSION = sessionmaker(bind=DB_ENGINE)
```

Here is a sample query to get all the Stats objects from the database in descending order (from newest to oldest). Note that the first would be the most recent in this case:

```
session = DB_SESSION()

results =
session.query(Stats).order_by(Stats.last_updated.desc())

session.close()
```

Here is sample code to write a new Stats object to the database:

```
session = DB_SESSION()

stats = Stats(stats["num_bp_readings"],
              stats["num_hr_readings"],
              stats["max_bp_sys_reading"],
              stats["max_bp_dia_reading"],
              stats["max_hr_reading"],
              datetime.datetime.strptime(stats["last_updated"],
"%Y-%m-%dT%H:%M:%S"))

session.add(stats)
```

```
session.commit()
session.close()
```