ACIT 3855 – Service Oriented Architectures – Fall 2021

Data Storage in Microservices

Most services within a microservices based system need to store data. This data storage could be in a cache (if the data is transient), a data file (i.e., text, json or other format) or most likely a database (MySQL, PostgreSQL, SQL Server, Mongo, etc.). Please review Fowler's discussion on decentralized data management in microservices:

- https://martinfowler.com/articles/microservices.html#DecentralizedDataManagement

A monolithic application will have a single database for the entire application. In a microservices architecture, each service has its own independent data store following the Database Per Service Pattern. Please review this pattern:

- https://microservices.io/patterns/data/database-per-service.html

Note: If you remember our definition of a design pattern from ACIT 2515, roughly: "A design pattern is a general, reusable solution to a commonly occurring problem in software design."

Likewise, we will discuss various architectural patterns for a microservices architecture which can be defined as a "general, reusable solution to a commonly occuring problem in software architecture within a given context."

Data Storage in the Week 3 Lab

We will be initially using a file based database for Lab 3 (i.e., SQLite) and migrating to a database server in Lab 4 (i.e., MySQL).

Please consider these questions:

- What is the key difference between sqlite and MySQL?
- What are the advantages and disadvantages of using a database like sqlite?
- What are the advantages and disadvantages of using a database like MySQL?

We will also be using an Object Relational Mapper (ORM), specifically SQLAlchemy to interact with our database. An ORM allows us to maps Python objects to database tables and interact with those objects as we would any other object in Python. It also allows us to be database agnostic since the ORM provides services to create, update, delete and query objects from the database. This will make it easy to transition from a SQLite to a MySQL database.

You may or may not have covered this in ACIT 2515. Here is a brief tutorial on SQLAlchemy.

Let's say we have a table for a Heart Rate reading:

heart_rate					
id	patient_id	device_id	heart_rate	timestamp	date_created

Created in a SQLite database by the following Python code:

```
import sqlite3

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

c = conn.cursor()
c.execute('''
          CREATE TABLE heart_rate
          (id INTEGER PRIMARY KEY ASC,
           patient_id VARCHAR(250) NOT NULL,
           device_id VARCHAR(250) NOT NULL,
           heart_rate INTEGER NOT NULL,
           timestamp VARCHAR(100) NOT NULL,
           date_created VARCHAR(100) NOT NULL)
          ''')

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

SQLAlchemy Declaratives

A SQLAlchemy declarative is mapping of a Python object to a row in a database table.

It looks like this:

```
from sqlalchemy import Column, Integer, String, DateTime
from base import Base
import datetime


class HeartRate(Base):
    """ Heart Rate """

    __tablename__ = "heart_rate"

    id           = Column(Integer, primary_key=True)
    patient_id   = Column(String(250), nullable=False)
    device_id    = Column(String(250), nullable=False)
    heart_rate   = Column(Integer, nullable=False)
    timestamp    = Column(String(100), nullable=False)
    date_created = Column(DateTime, nullable=False)

    def __init__(self, patient_id, device_id, timestamp, heart_rate):
        """ Initializes a heart rate reading """
        self.patient_id = patient_id
        self.device_id = device_id
        self.timestamp = timestamp
        self.date_created = datetime.datetime.now() # Sets the
                                    # date/time the record is created
```

HeartRate extends a SQLAlchemy Base class

Table Mapping

Mapping to the heart_rate table

Column Mapping

Mapping to the columns in the heart_rate table

Constructor

```
        self.heart_rate = heart_rate

def to_dict(self):
        """ Dictionary Representation of a heart rate reading """
        dict = {}
        dict['id'] = self.id
        dict['patient_id'] = self.patient_id
        dict['device_id'] = self.device_id
        dict['heart_rate'] = self.heart_rate
        dict['timestamp'] = self.timestamp
        dict['date_created'] = self.date_created

        return dict
```

Custom Method

Helper method to put the object's attributes into a Python dictionary. This Python dictionary can later be serialized into JSON.

SQLAlchemy Sessions

We need a database session to add or query HeartRate objects to/from the database. Here are some examples:

```
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker
from heart_rate import HeartRate

…

# Example Add a Heart Rate Reading
session = DB_SESSION()

hr = HeartRate(patient_id,
               device_id,
               timestamp,
               heart_rate)


session.add(hr)


session.commit()
session.close()

…

# Example Query for All Heart Rate Readings
session = DB_SESSION()

all_readings = session.query(HeartRate).all()
one_reading = session.query(HeartRate).filter(
                HeartRate.id = query_id).first()
```

Create a database session

Create a new HeartRate object (calls the Constructor).

Adds the HeartRate object to the database session

Commits the HeartRate object to the database and closes the session.

Create a database session

Queries for all HeartRate objects from the database.

```
session.close()
…

# Can now access public attributes and methods on the
# HeartRate objects
print("Heart Rate is %d from device" % (one_reading.heart_rate,
                                        one_reading.device_id))

# Create a list of Python dictionaries for all readings
all_readings_list = []

for reading in all_readings:
    all_readings_list.append(reading.to_dict())

…
```

Close the session

Mapped columns are public instance variables

Useful to convert to Python objects when returned as a response message from a connexion endpoint