Chapter 15 - Microservices and Patterns For The Cloud

September 18, 2021

0.1 Overview

- microservices fall into `Modern Architecture-style` design patterns
- the idea is that we can build an application as a set of loosely coupled, collaborating services
- increasing number of applications are deployed in the cloud and must be designed upfront with this type of environment and its constraints in mind
- the microservice patterns we will focus on are: `Retry`, `Circuit Breaker` and `Cache-Aside` and `Throttling`

0.2 Microservice Pattern

problems with Monolithic Model - sicne a signle code base is used, the development team has to work on maintaining the whole code base simultaneously - it is more difficult to organize testing and reproduce and fix bugs - tests and development becomes difficult to manage as the application and its user base grows and its constraints increase

0.2.1 Use Cases

microservice is a good choice when: - the requirement to support different clients, including desktop and mobile - there is an API for third parties to consume - we have to communicate with other applications using messaging - we serve requests by accessing a database, communicating with other systems, and returning the right type of response (`JSON`, `XML`, `HTML` or even `PDF`) - there are logical components corresponding to different functional areas of the application

0.2.2 Implementation

- we will have to use docker because you need container when running multiple services
- to use `RabbitMQ` using `Docker`
  - `docker run -d -p 5672:5672 -p 15672:15672 --name rabbitmq rabbitmq`

[1]:
```
def wrapper():

    fake = Faker()

    class PeopleListService:
        name = 'peoplelist'

        @rpc
        def populate(self, number=20):
```

```
            names = []
            for _ in range(0, number):
                n = fake.name()
                names.append(n)
            return names
```

[2]:
```
def test_wrapper():
    from nameko.testing.services import worker_factory
    from service_first import PeopleListService
    def test_people():
        service_worker = worker_factory(PeopleListService)
        result = service_worker.populate()
        for name in result:
            print(name)
    if __name__ == "__main__":
        test_people()
```

0.3 Retry Pattern

- `Retrying` is an approach that is more and more needed with microservices
- there might be networking issues and we need to retry

0.3.1 Real-World Examples

- in python the `Retrying` library is available to simplify the taks of adding retry behavior to our functions

0.4 Use Cases

- `retrying` approach is not recommened for handling failures such as internal exceptions caused by errors in the application logic itself
- also if there is frequent `retrying`, there might be scaling issues with the application

[7]:
```
import time
import sys
import os

def create_file(filename, after_delay=5):
    time.sleep(after_delay)

    with open(filename, 'w') as f:
        f.write('A file creation test')

def append_data_to_file(filename):
    if os.path.exists(filename):
        with open(filename, 'a') as f:
            f.write(' ...Updating the file')
    else:
```

```
        raise OSError
```

- in the main part of the code, we use the right function depending on what is passed

[9]:
```
def wrapper():
    FILENAME = 'file1.txt'
    if __name__ == "__main__":
        args = sys.argv

        if args[1] == 'create':
            create_file(FILENAME)
            print(f"Created file '{FILENAME}'")
        elif args[1] == 'update':
            while True:
                try:
                    append_data_to_file(FILENAME)
                    print("Success! We are done!")
                    break
                except OSError as e:
                    print("Error... Try again")
```

- we can also do this with a third party module

[11]:
```
def wrapper():

    import time
    import sys
    import os
    from retrying import retry

    def create_file(filename, after_delay=5):
        time.sleep(after_delay)

        with open(filename, 'w') as f:
            f.write('A file creation test')

    @retry
    def append_data_to_file(filename):
        if os.path.exists(filename):
            print("got the file... let's proceed!")
            with open(filename, 'a') as f:
                f.write(' ...Updating the file')
            return "OK"
        else:
            print("Error: Missing file, so we can't proceed. Retrying...")
            raise OSError
```

- the `retry` module is not maintained
- we can use the `tenacity module`

[14]:
```python
def wrapper():
    import tenacity

    #@tenacity.retry(wait=tenacity.wait_exponential())
    @tenacity.retry(wait=tenacity.wait_fixed(2))
    def append_data_to_file(filename):
        # code that could raise an exception
        pass
```

0.5 Circuit Breaker Pattern

- when a failure to due to communication with an external component is likely to be long-lasting, using a retry mechanism can affect the responsiveness of the application
- with `circuit breaker`, you wrap a fragile function call in a special object, which monitors for failure
- once the failures reach a certain threshold, the circuit breaker `trips` and all further call to the circuit breaker return with an error
- in python we use `pybreaker`

0.5.1 Implementation

[16]:
```python
def wrapper():

    import pybreaker
    from datetime import datetime
    import random
    from time import sleep


    breaker = pybreaker.CircuitBreaker(fail_max=2, reset_timeout=5)

    @breaker
    def fragile_function():
        if not random.choice([True, False]):
            print(' / OK', end='')
        else:
            print(' / FAIL', end='')
            raise Exception('This is a sample Exception')


    if __name__ == "__main__":
        while True:
            print(datetime.now().strftime('%Y-%m-%d %H:%M:%S'), end='')

        try:
            fragile_function()
```

```
        except Exception as e:
            print(' / {} {}'.format(type(e), e), end='')
        finally:
            print('')
            sleep(1)
```

0.6 The Cache-Aside Pattern

- a situation where data is more frequently read than updated, applications use a cache to optimize repeated access to information stored in a database or data store

  Load data on demand into a cache from a data store, as an attempt to improve performance, while maintaining consistency between data held in the cache and the data in the underlying data store

0.6.1 Realworld-examples

- realworld examples are `Memcached`; which is a popular in-memeory `key-value` store for small chunks of data
- `Redis` is another server solution
- Amazons `ElastiCache`, according to the documentation is a web service that makes it easy to set up, manage and scale distributed in-memeory data store or cache environment in the cloud

0.6.2 Use cases

- useful for data that does not change often and for data storage that doesn't depend on consistency of a set of entries in the storage
- pattern might not be suitable in the case where cached data set is static or for caching session state information in a web application hosted in a web farm

0.6.3 Implementation

Case 1: - when we want to fetch a data item: return the item from cache if found in it. If not found in cache, read the data from the database. Put the read item in the cache and return it
Case 2: - when we want to update a data item: write the item in the database and remove the corresponding entry from the cache

[21]:
```
def wrapper():

    import sys
    import sqlite3
    import csv
    from random import randint
    from faker import Faker
    fake = Faker()

    def setup_db():
        try:
```

```
            db = sqlite3.connect('data/quotes.sqlite3')
            # Get a cursor object
            cursor = db.cursor()
            cursor.execute('''
                CREATE TABLE quotes(id INTEGER PRIMARY KEY, text TEXT)
            ''')
            db.commit()
        except Exception as e:
            print(e)
        finally:
            db.close()

    def add_quotes(quotes_list):
        quotes = []
        try:
            db =  db = sqlite3.connect('data/quotes.sqlite3')
            cursor = db.cursor()

            quotes = []
            for quote_text in quotes_list:
                quote_id = randint(1, 100)
                quote = (quote_id, quote_text)
                try:
                    cursor.execute('''INSERT INTO quotes(id, text) VALUES(?, ?
↪)''', quote)
                    quotes.append(quotes)
                except Exception as e:
                    print(f"Error with quote id {quote_id}: {e}")
            db.commit()
        except Exception as e:
            print(e)
        finally:
            db.close()
        return quotes

def main():
    args = sys.argv
    if args[1] == 'init':
        setup_db()

    elif args[1] == 'update_db_and_cache':
        quotes_list = [fake.sentence() for _ in range(1, 11)]
        quotes = add_quotes(quotes_list)
        print("New (fake) quotes added to the database:")
        for q in quotes:
            print(f"Added to DB: {q}")
```

```
        # Populate the cache with this content
        with open('data/quotes_cache.csv', "a", newline="") as csv_file:
            writer = csv.DictWriter(csv_file,
                                    fieldnames=['id', 'text'],
                                    delimiter=";")
            for q in quotes:
                print(f"Adding '{q[1]}' to cache")
                writer.writerow({'id': str(q[0]), 'text': q[1]})
    elif args[1] == 'update_db_only':
        quotes_list = [fake.sentence() for _ in range(1, 11)]
        quotes = add_quotes(quotes_list)
        print("New (fake) quotes added to the database ONLY:")
        for q in quotes:
            print(f"Added to DB: {q}")

# if __name__ == "__main__":
#     main()
```

[26]:
```
def wrapper_cache_aside():
    import sys
    import sqlite3
    import csv

    cache_key_prefix = "quotes"

    class QuoteCache:
        def __init__(self, filename=""):
            self.filename = filename

        def get(self, key):
            with open(self.filename) as csv_file:
                items = csv.reader(csv_file, delimiter=';')
                for item in items:
                    if item[0] == key.split('.')[1]:
                        return item[1]
        def set(self, key, quote):
            existing = []
            with open(self.filename) as csv_file:
                items = csv.reader(csv_file, delimiter=';')
                existing = [cache_key_prefix + "." + item[0] for item in items]

            if key in existing:
                print("This is weird. The key already exists.")
            else:
                # save the new data
                with open(self.filename, "a", newline="") as csv_file:
                    writer = csv.DictWriter(csv_file,
```

```
                                        fieldnames=['id', 'text'],
                                        delimiter=";")
                    writer.writerow({'id': key.split('.')[1], 'text': quote})

        cache = QuoteCache('data/quotes_cache.csv')

        def get_quote(quote_id):
            quote = cache.get(f"quote.{quote_id}")
            out = ""

            if quote is None:
                try:
                    db = sqlite3.connect('data/quotes.sqlite3')
                    cursor = db.cursor()
                    cursor.execute(f"SELECT text FROM quotes WHERE id =␣
↪{quote_id}")
                    for row in cursor:
                        quote = row[0]
                        print(f"Got '{quote}' FROM DB")
                except Exception as e:
                    print(e)
                finally:
                    # Close the db connection
                    db.close()

                # and add it to the cache
                key = f"{cache_key_prefix}.{quote_id}"
                cache.set(key, quote)
            if quote:
                out = f"{quote} (FROM CACHE, with key 'quote.{quote_id}')"
            return out

    if __name__ == "__main__":
        args = sys.argv

        if args[1] == 'fetch':
            while True:
                quote_id = input('Enter the ID of the quote: ')
                q = get_quote(quote_id)
                if q:
                    print(q)
```

0.7 Throttling

- throttling is based on limiting the number of requests a user can send to a given web service in a given amount of time

0.7.1 Implementation

- differnet types of limiters limit based on `IP` or `session` or `account`

[27]:
```python
def wrapper():
    from flask import Flask
    app = Flask(__name__)

    limiter = Limiter(
        app,
        key_func=get_remote_address,
        default_limits=["100 per day", "10 per hour"]
    )

    @app.route("/limited")
    def limited_api():
        return "Welcome to our API!"

    @app.route("/more_limited")
    @limiter.limit("2/minute")
    def more_limited_api():
        return "Welcome to our expensive, thus very limited, API!"
```