DHBW Stuttgart

WWI2021F 4th semester – Distributed Systems portfolio examination: program design

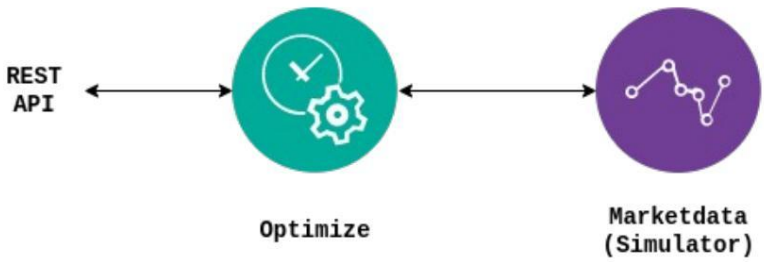
story

You get your electricity from an electricity provider with a flexible tariff. That means you Electricity price changes every hour on the hour, depending on the current electricity market situation. Your electricity provider makes the price available via a REST API (“Marketdata API”). You want to take advantage of this by shifting larger consumptions to the optimal, ie cheapest, time. Suppose you can also automatically determine how many hours, for example, your washing machine runs or your e-scooter / pedelec needs to be charged. Then the cheapest / cheapest period for this could be by software

electrical consumption can be determined.

tasks

1. Develop an “Optimize” microservice that uses the data from the Marketdata API to determine the most favorable continuous time period (in full hours, eg 1 hour or 3 hours). The microservice should be written in Python, Node.js or Java.



2. Deploy your microservice in Docker.
3. Deploy your microservice to Kubernetes.
4. Document your solution to the task.

Details on the Marketdata Simulator, how to put it into operation and on the Marketdata API can be found in ÿ Appendix: Marketdata API.

Part 1: Optimize Microservice

Two API endpoints are to be implemented:

1. Data endpoint

Determine the most favorable period from the number of hours (integer) passed by using the Marketdata API of the simulator is called ÿ The Marketdata API

Operation: GET, Path: /api/v1/get_optimal, Parameters: q = number of hours of call, eg curl -X 'GET' 'http://0.0.0.0:8001/api/v1/get_optimal?q=3 '

Result as JSON object:

{"start":"05.01.2023 06:00:00","duration":3,"avg_price":0.2259}

- "start" is the start of the cheapest period (in a human-readable form). • "duration" is the number of hours passed in the call. • "avg_price" is the average electricity price in €/kWh. The Marketdata API delivers
  Electricity prices in €/MWh, which is unusual for end consumers. Average given in the example: (price at 6:00 + price at 7:00 + price at 8:00) / 3 / 1000

If the number of hours q requested is greater than the number of hours for which the Market API provides price data, an error should be returned:

HTTP code: 400

Response: {"Client Error": "Requested number of hours not satisfiable"}

Note 1: ÿ The Marketdata API provides electricity price data for a minimum of 12 and a maximum of 24 hours, depending on the time of day of the call.

Note 2: Marketdata API URL and API Key might change...

2. Health check endpoint

To test whether the microservice is available.

Operation: GET, Path: /health

Call eg curl -X 'GET' 'http://0.0.0.0:8001/health'

Result eg: {"status": "UP"}

implementation

- Programming language: Python, Java or Node.js
- Implement a minimum of error tolerance for calling the Marketdata API, eg if the service is not available (HTTP code 500, internal server error) or the API key is incorrect (HTTP code 401, unauthorized).
- **Document your code and custom API.**
- Remember the 12 Factor Apps and the requirement to separate code and configuration (Marketdata API URL, API Key).

Part 2: Docker Deployment

Create a Docker image for your microservice from part 1 that meets the requirements of the 12 factor apps (external configuration).

If you wrote your microservice in Java, be sure to use a **multi stage build** for Java Compile and Docker Image Build.

https:// codefresh.io/ docker-tutorial/ create-docker-images-for-java/
Section: Multi-stage Build (The ideal way)

In your documentation, describe:

1. the Dockerfile
2. How to build an image with the Dockerfile ('docker build') 3. How to start this image correctly in Docker ('docker run') 4. How to call your service externally, eg with curl.

Part 3: Kubernetes Deployment

Develop the configuration for:

- a Kubernetes deployment for your container image • a Kubernetes service (of the type: Nodeport) • a Kubernetes object for the configuration (configmap and/or secret)

ÿ 3. Kubernetes describes how to deploy a Marketdata Simulator instance in Kubernetes.

In your documentation, describe:

1. the essential details of your Kubernetes configuration 2. how the configuration is deployed in Kubernetes
3. How to call the service externally, eg with curl.

Part 4: Documentation

Your documentation should not be a scientific elaboration. However, someone unfamiliar with this exam task and your implementation should be able to understand your solution and get your code working locally, in containers, and in Kubernetes.

levy

Part 1	Complete documentation/description of the microservice to a reasonable extent (no academic elaboration!) including architecture and installation and/or compilation instructions as well as Instructions for starting the service.
Part 2	code with all required files and directories. Comment your Microservice, best with the help of Javadoc, JSDoc, or similar for Python.
Part 3	1. Document the Dockerfile and correct 'docker build' and 'docker run' command. **Java: please multi-stage build!** 2. Place Dockerfile in the root directory of your microservice code. Please do not hand in the Docker image!
Part 4	1. Document the Kubernetes deployment, service, etc., the necessary kubectl commands and the access to the service from outside the Kubernetes cluster, eg as a 'curl' call. 2. YAML file(s) in a directory of your microservice code (eg /deployment).

Contents: One (1) packed ZIP file containing the complete code and your documentation. Please include your documentation **either as Markdown (README.md) or as PDF** with the code pack. Please check the ZIP file before uploading it to Moodle, in particular whether it is really contains data!

The submitted ZIP file should contain your name, eg Doe_Jane_Abgabe.zip.

Submission via Moodle submission link by Thursday, March 16th, 2023, 11:59 pm

Maximum score: 48

Late submission or incorrect or empty ZIP files are rated with 0 points.

Appendix: Marketdata API

background

The idea for this task is based on the offer from aWATTar Deutschland GmbH, an electricity provider in Munich who "Allows electricity consumption to be shifted to the greenest and cheapest hours through time-variable tariffs".

The provider offers an API for a data feed for "tinkerers and developers", about the the Future electricity prices can be queried. The prices come from the EPEX SPOT Day Ahead electricity market. This market works via a blind auction, exchange members enter their bids in an order book. Once a day, based on supply and demand, EPEX SPOT calculates a market price for each hour of the following day. The results will be published from 12.55 pm (for the German market).

I have no ties or obligations to aWATTar, but found the idea very interesting as a basis for an exam paper.

There is the following notice on the homepage:

> Become an aWATTar customer and use our data feed free of charge according to the fair use principle (60 queries per minute). You currently do not need an access token!

Since we're probably not all customers and I don't want to risk suddenly requiring an access token or otherwise blocking access to the API while you're working on your exam assignment, I've recreated the API and made it available in the form of a container image placed.

The Marketdata API

- The API returns a JSON object with electricity prices that are valid for a full hour, eg from 2:00 p.m. to 3:00 p.m
- From 1 pm the prices for the next day are available. This means that between 1:00 pm and 2:00 pm today, the Prices for the 24 hours tomorrow plus the remaining 11 hours today are available. That would be 35 price records. But:
- The API only provides data for a maximum of 24 hours in advance.
- Ie from midnight the number of records decreases every hour until the minimum of 12 records for 12 hours between 12 pm and 1 pm of the next day is reached. From 1 pm, prices for 24 hours are available again and the process of decreasing the number of data records is repeated the next morning from 0:00 am • The simulated API behaves identically to the "real" API, with the exception of the price data, which does not change in the simulated API, ie the prices for a call today at 1:08 pm are identical to the prices for a call tomorrow at 1 pm :08. However, the timestamps are correct for the current time.

- As an additional obstacle, “my” Marketdata API requires the submission of a API key (password). details see below

The container image is in my public Docker Hub repository haraldu/marketdatasim

The Marketdata simulator can be started with this command:

```
$ docker run --name marketdata -d -p 8080:8080 haraldu/marketdatasim:1
```

Ie the service is then via http://localhost:8080 reachable.

If you open this URL in the browser, you will be redirected to the API Explorer ("Swagger"), in which you can test the API.

The Marketdata API is on the path /v1/marketdata, the API key is always 12345.

Ie the API call is complete:

http://localhost:8080/v1/marketdata?apikey=12345

The result is a JSON object: {


```
"object": "list", "data":
[ {

        "start_timestamp": 1672153200000,
        "end_timestamp": 1672156800000,
        "marketprice": 388.97, "unit":
        "Eur/MWh" },

...
    {
        "start_timestamp": 1672236000000,
        "end_timestamp": 1672239600000,
        "marketprice": 250, "unit":
        "EUR/MWh"


} ], "url": "/de/v1/marketdata"
}
```

The essential part is the "data" array, which, as described above, contains between 12 and 24 entries, depending on the time of day, with the electricity price ("market price" in euros per MWh) for a full hour ("start_timestamp " to " end_timestamp").

The two timestamps are Unix epoch timestamps (an online converter is eg here: https://www.unixtimestamp.com/index.php), these differ from eg Python timestamps, the Unix epoch timestamp contains the milliseconds!

If the API key is missing completely, the simulator returns an HTTP code 422 because the API key is in required parameter is:

```
HTTP Response Code 422 Unprocessable Entity,
Response: {"detail":[{"loc":["query","apikey"],"msg":"field required","type":"value_error.missing"}] }
```

If the API key is wrong, the answer is:

```
HTTP Response Code 401 Unauthorized,
Response: {"detail":"ERROR 401: Invalid API key."}
```

Execution Environments

You will most likely develop your own microservice locally first, then build a container image from it and start the service as a container, and finally deploy the container as a pod in Kubernetes. Then start the Marketdata Simulator as follows:

1.Local

```
$ docker run --name marketdata -d -p 8080:8080 haraldu/marketdatasim:1
```

API call: http://localhost:8080/v1/marketdata?apikey=12345

2.Containers

If your microservice runs in the container, it must be in the same container network as the simulator, otherwise the two cannot communicate.

Create a suitable Docker network named “dhbw”:

```
$ docker network create dhbw
```

Start of the simulator in this network:

```
$ docker run --name marketdata -d -p 8080:8080 --network dhbw \
        haraldu/marketdatasim:1
```

Your own container must then also be started in this network ("dhbw"), then the API is called with the network name "marketdata":

```
http://marketdata:8080/v1/marketdata?apikey=12345
```

3. Kubernetes

The simulator is deployed in Kubernetes using the marketdata.yaml file:

```
$ kubectl apply -f marketdata.yaml
```

The marketdata.yaml file is provided to you with this documentation.

This creates a service and a deployment (and therefore a pod).

API call: http://marketdata:8080/v1/marketdata?apikey=12345

This call only works within Kubernetes. Outside of Kubernetes you can

the API URL can be displayed in the case of Minkube with this command:

```
$ minikube service marketdata --url
```

Attachment: marketdata.yaml

```
apiVersion: v1 kind:
Services metadata:

    name: marketdata
    labels:
        app: marketdata
specs:
    type: NodePort ports:
    - ports:
        8080 name: http
            selector: app:
    marketdata

---
apiVersion: apps/v1 kind:
Deployment metadata:

    name: marketdata
    labels:
        app: marketdata
specs:
    replicas: 1
    selector:
        matchLabels: app:
            market data template:
    metadata:

        labels:
            app: marketdata
    spec:
        containers:
        - image: haraldu/marketdatasim:1 name:
            marketdata ports:
            -
            containerPort: 8080 readinessProbe:
            httpGet: path: /health
                port: 8080
                initialDelaySeconds: 5
                periodSeconds:
                10
```