UNIVERSITY OF PISA

Master in Computer Engineering
Project for Electronic systems

REST application using container

Author:
Tommaso Burlon

Academic year
2020/2021

Summary

1 Introduction

For this project I've design and develop a small REST application using Docker container and kubernetes. The application is a very simple forum application where user can start threads of discussion and every user can comment a thread or watch the comment of other users.

2 Rest method implementation

This application expose a REST interface from the port 15000. The REST method implemented are the follows

Method	URI	Description
GET	/user/<id>	get the user whoose id is equal to <id>
POST	/user	Add and user to the system
PUT	/user/<id>	Update an user with new information
DELETE	/user/<id>	Remove an user
GET	/users/<page>	Retrieve a list of users
GET	/thread/<id>	Get the thread which id is equal to <id>
POST	/thread	Create a new thread
PUT	/thread/<id>	Update a thread (only the title)
DELETE	/thread/<id>	Remove a thread
GET	/threads/<page>	Get a list of threads
GET	/comment/<id>	Get the comment which id is equal to <id>
POST	/comment	Create a new comment
PUT	/comment/<id>	Update a comment (only the body)
DELETE	/comment/<id>	Remove a comment
GET	/lastComments/<page>	Get a list of comments
GET	/threadComments/<thread_id>/<page>	Get a list of comment from thread <thread_id>

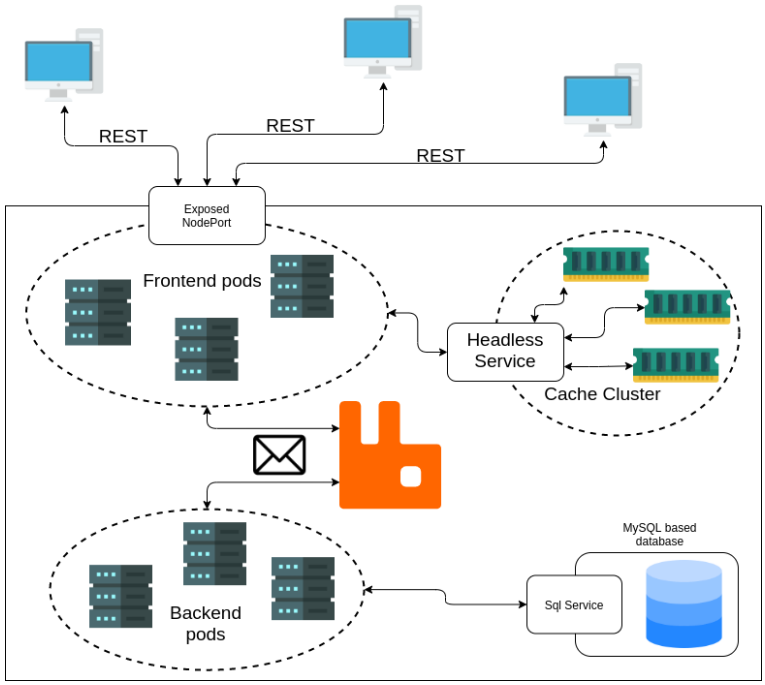
All the data exchange is made using a JSON format. The field of the entities

are the follow:

1. User
    (a) id [integer]
    (b) username [string]
    (c) password [string]
2. Thread
    (a) id [integer]
    (b) user_id [integer] (user that create the thread)
    (c) title [string]
3. Comment
    (a) id [integer]
    (b) body [string]
    (c) user_id [integer]
    (d) thread_id [integer]
    (e) postDate [Datetime]

3 Project structure

The project was develop using container. The project is divided into five main components: frontend, backend, message queue (RabbitMQ), caches, database. The frontend is a conteiner that communicate directly with: the cache, the client with a REST-interface ad with the message queue hiding the implementation detail of the database and of the backend. The backend can communicate directly with the database and with the message queue. The backend act like a server answering to the jobs that the frontend pubblish. The cache is implemented using a container of the popular memcached, the direct connection of the frontend with the cache is not separated by the message queue for performance reason (the cache should be near the frontend) however every container is inside a different pod.



4 Implementation detail

The frontend is implemented in Python using the Flask library to handle REST requests. The backend is implemented also in Python. The db used in this project was mySQL. The message exchanged by the frontend and backend use a JSON structure containing the payload and the return code of the message while the action that the backend has to complete it is derived form the queue in which the command was sent, the possible action (and so the queues) are 5: "GET", "POST", "PUT", "DELETE", "GET LIST", this approach was used to maintain the application highly modular (it is possible to create different images to handle different requests but I have not implemented this solution, I believe that for this case a single backend image was sufficient in terms of scalability and efficiency). The project was deployed in a kubernetes environment (In my case i used the minikube program to build the environment). With kubernetes I can also access to a builtin DNS server to traduce the hostname of the message queue, of the cache servers and of the databse so the application require no configuration. I've not implemented an auto-scalar so in case of

poor performance the .yaml file has to be replaced with other files with a different number of replicas of the frontend, backend and cache server. The mySQL db is fairly simple, contains only three table with the same kind of record of the ones described in the second paragraph

5 Cache use

The cache servers have been used in the following way: every GET method to retrieve a single record (/user/<id>, /comment/<id> and /thread/<id>) save the result to the cache and the relative PUT, DELETE or POST methods update the value in the cache (in this way there should be no inconsistency). For the GET methods that return a list of record (/users/<page-id>, /thread/<page-id>, /lastComments/<page-id> and /threadComments/<thread-id>/<page-id>) the situation is different, in fact when a POST, PUT or DELETE occur every page is changed in this way every page should be flushed or updated in the cache. Instead my approach was to save only the first pages (that are usually the more important in this kind of applications) and every time an UPDATE methods is called these records are flushed from the cache reducing some works from frontend pod.