Assignment 2: RESTful Microservices Architectures

Web Services and Cloud Based System 2020
Lecturer: dr. Adam Belloum, Group 2

Hasine Efetürk	Furong Guo	Yaping Ren
VU ID: 2527299	UvA ID: 12577790	UvA ID: 12985090

April 24, 2020

2.1 User registration and login

Design

On top of previous assignment, we still use Flask for this assignment, adding user authentication step. And library jwt is imported to generate JWT, which will look at username, password, and able to set expiration time length. For user registration and user login there should be 2 separate routes, "/users" and "/users/login" respectively. And this micro service runs on port 5001, while URL shortener service still runs on port 5000.

Implementation

"/users" - POST
We created a dictionary to store username and password in memory. For every new entered username and password pair, they're stored in the dictionary. Since username check is not required, and the dictionary gets cleaned after service reset, no additional check is deployed.

"/users/login" - POST
Getting entered username and password, we could look them up in dictionary and see if there's a match. If not, the function would return "forbidden" with code 403. If there is a match, the function would generate the JWT. Expiration time length is set 10 minutes with the help to datetime.timedelta(). And the token is returned in json format using jsonify.

2.2 Update URL-shortener

a. Retrieve JWT

We added the JWT secret token in this file as well to avoid the trouble of asking for secret key from the other port. We use the returned token from the successfully logged in return value, and put in header "x-access-token" value for verification. Library jwt would use same algorithm and same secret key to decode the token.

b. Verify JWT

Since there is no need to verify JWT for any "GET" requirements, the verification step is added after after that. If the return value of authentication check is false, then the function will return "forbidden" with code 403. Else will go on with processing the requests.

2.3 Single entry point

To enable two or multiple microservices have one entry point port instead of running separately on different ports, an API gateway approach is considered. This provides a single entry point for a group of microservices.

It acts as a reverse proxy, routing requests from clients to services. It can also provide additional cross-cutting features such as authentication, SSL termination, and cache[1]. When using HTTP and RESTful services, the API consists of the URLs and the request and response JSON formats.

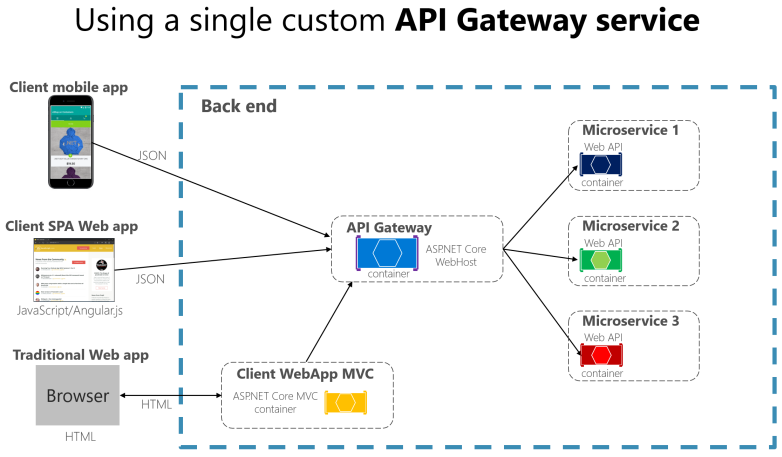

Figure 1: Azure API Gateway as a custom service

Instead of having a suitable solution for every client, a different API for each client to fit its requirements is preferable. The design of this approach will enable each kind of client with a separate API gateway.

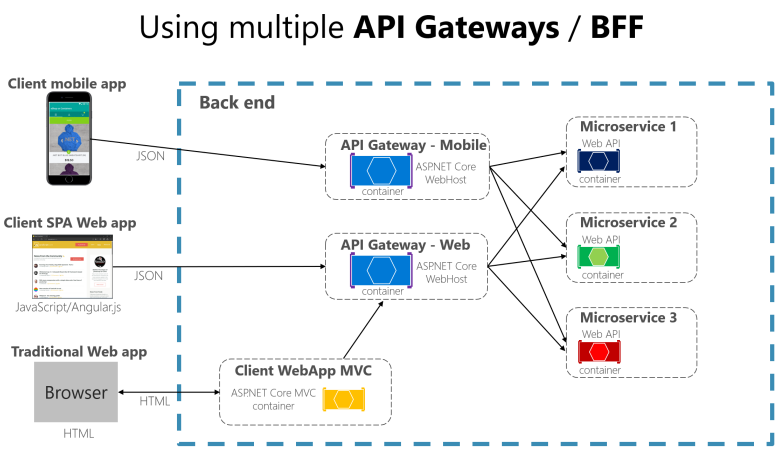

Figure 2: Multiple API Gateways

2.4 Scaling up services

Offloading services during peak traffic times can be realized by scaling up services independently of each other. An orchestrator is essential if the client relies on services and needs to ensure the functionality to scale up and down.

Each service instance needs to be contained in a single container also seen as "units of deployment", a single instance of Docker. Those containers are handled by a host. Docker is suitable for this. Docker engine in a single Docker hosts manage simple image instances. But for the management of multiple containers a more complex orchestrator like Kubernetes is needed. Kubernetes enables the deployment of multiple containers into the cluster and scale-out with any number of container instances. It automatically starts containers, scale-out containers with multiple instances per image, suspend them or shut them down when needed, and ideally also control how they access resources like the network and data storage[2].

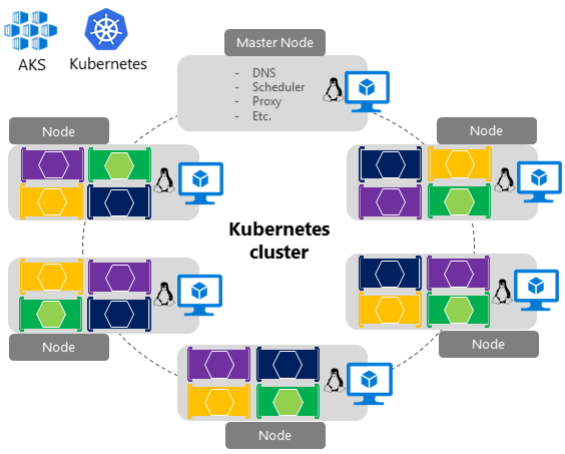

Figure 3: Kubernetes cluster topology

2.5 Track services

In a microservice architecture, many web services can be distributed over several backend servers. To keep track of all the services, you can use a management orchestrator like Kubernetes. This will grant access into the health of each service. An Resource Manager Template is a powerful method to keep track of all the deployed services or to automate deployment. The template is a JSON file that defines the infrastructure, location and configuration of the service.

```
    "resources": [
  {
    "type": "Microsoft.Storage/storageAccounts",
    "apiVersion": "2016-01-01",
    "name": "mystorageaccount",
    "location": "westus",
    "sku": {
      "name": "Standard_LRS"
    },
    "kind": "Storage",
    "properties": {}
  }
]
```

The resource manager converts the JSON template into REST API operations[3].

References

[1] The api gateway pattern versus the direct client-to-microservice communication. https://docs.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-applications/direct-client-to-microservice-communication-versus-the-api-gateway-pattern. Accessed: 2020-04-21.

[2] Orchestrate microservices and multi-container applications for high scalability and availability. https://docs.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-applications/scalable-available-multi-container-microservice-applications. Accessed: 2020-04-23.

[3] What are arm templates? https://docs.microsoft.com/en-us/azure/azure-resource-manager/templates/overview. Accessed: 2020-04-23.