LAB 5: ORCHESTRATION AND SCALING

In the previous lab session, you learned about building images and starting containers. You also learned to write Docker compose files to manage multi-container applications on a single node.

The goal of this lab session is to provide you with an introduction on deploying and managing multiple instances of services across a multi-node cluster. You will also learn how to auto-scale a service and how to reduce the likelihood of a system-wide failure when only a single host or micro-service is unavailable.

We will continue working on the Hospital application from the previous Lab sessions (shown in Figure 1), but rather than having a single instance of each service, you will deploy and manage multiple instances. Although we will only be using a single node (i.e. your local PC), everything we teach you in the Lab session is applicable for multiple nodes.

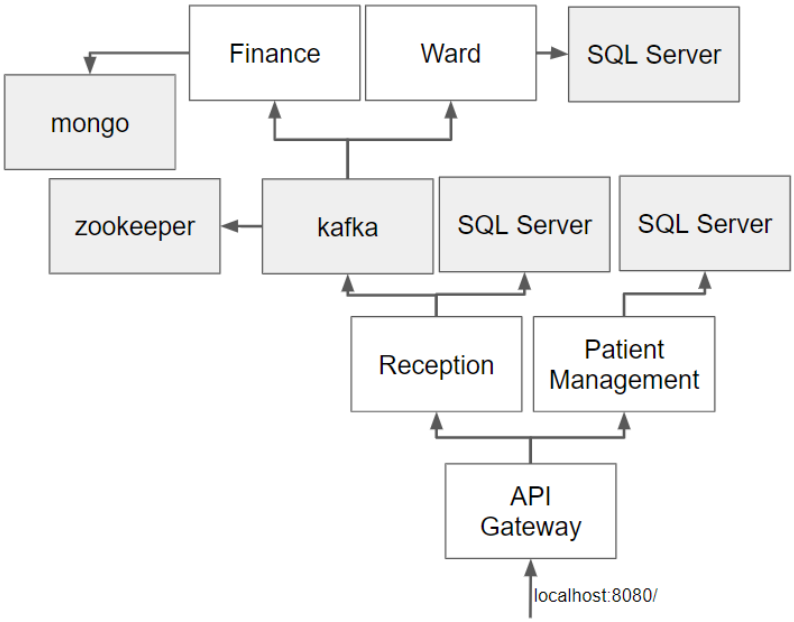

Figure 1 Containers you deployed in the previous Lab session.

1 Getting started with Docker swarms

Docker Swarm is an orchestration tool for multi-container applications. Its concepts are very similar to Kubernetes, which was discussed in the theory lectures. In this section you will learn about the basic commands and key concepts of Docker swarm. You will use this knowledge in the following sections to deploy the Hospital application.

For information about the key concepts of Docker swarm, you can consult: https://docs.docker.com/engine/swarm/key-concepts/

First, we need to initialize our swarm. A swarm has one master, all other nodes are worker nodes. In this Lab session you will be working with just a master node. Initialize the swarm:

- `docker swarm init`
- The output of `docker swarm init` should provide you with a token that you can use to add other machines to your swarm as worker nodes. You can also retrieve the token using: `docker swarm join-token worker`

1.1 Services, tasks and containers

To investigate the difference between services, tasks and containers, we will create a simple service and inspect what containers are running on our swarm nodes. You can view a list of the different `docker service` commands at

https://docs.docker.com/engine/reference/commandline/service/.

- Create a service from the MySQL image which has 2 replica tasks and is called mysql
  - You will also need to specify the MYSQL_ROOT_PASSWORD
- Check that your service is running:
  - `docker service ls`
- Find out which nodes the tasks are running on:
  - `docker service ps` mysql

Docker is monitoring what service tasks are running and will check that the correct number of replica tasks are being run within the swarm. To test this out try the following steps:

- Kill one of the containers running the mysql image: `docker kill <container name>`
  - re-run `docker service ps mysql`
  - What do you notice?

Respond to the following question: what is the difference between services, tasks and containers?

Once you have completed this section, remove your service:

- `docker service rm mysql`

2 Hospital Application

Multiple containers will be used to run the Finance, Ward, Reception, Patient and API Gateway services shown in Figure 1. These will use Docker's built in ingress network to perform service discovery and load balancing. If one of the containers fails, requests can be routed to another container running the same service. For more information about the ingress network see: https://docs.docker.com/engine/swarm/ingress/

We will start with a basic application only containing our Patient, patient MySQL database and API gateway containers.

2.1 Setup

Download the solution from the last lab session and check it runs.

- You will need to create a JAR file for each Spring boot application
- Run `docker-compose up --build` to run the solution.

2.2 Docker compose for swarms

In this section we will convert the Docker compose file into one which is compatible with Docker swarm. We will also try out some of the different configuration options. Information on how to write a docker-compose file can be found at: https://docs.docker.com/compose/compose-file

- Create a new docker-compose file.
  - You should use docker-compose version 3.
- Copy across the `patient_db` service from the previous Lab session and modify it to work with Docker swarm.
  - You need to supply an image name.
  - We only want to have a single instance of this service. Why?
- Copy across the `patient` service and modify it to work with Docker swarm.
  - Use 2 replicas
- Copy across the `api_gateway` service and modify it to work with Docker swarm.
  - Use the Global mode option.
  - What is the difference between a replicated and global service and why are they useful?

The command to start a Docker swarm from a compose file is:

- `docker stack deploy –compose-file <file-name> <stack-name>`

However, this command does not build the images for you (like the docker-compose command can). To save time, rather than building all your images manually you can use the docker-

compose file from the previous Lab session to build the images by running: `docker-compose -f <compose-file-name> build`

Start your swarm and try visiting: http://localhost/patient

- Which one of the Patient containers did the command run in?
- Try killing one of the Patient containers. All commands should now go to the other one.

2.3 Circuit Breaker

The circuit breaker pattern can be used to prevent the entire application from failing when a single service is unavailable, and to monitor the number of (failed) requests. Currently if our `Patient` service is not running, a 500 error message is shown to the user when they attempt to visit `http://localhost/patient`. Also, the `api_gateway` service will wait until it receives a response from the `Patient` service; therefore any response delays will affect the user.

Test this out:

- Stop your application if it is currently running: `docker stack rm <stack name>`
- Start only the API Gateway application.
  - `docker run -d -p 80:8080 -v "$(pwd)"/hospital-app-api-gateway:/src <api-gateway image name>`
- Visit http://localhost/patient

We will use a circuit breaker to provide a friendly message to the user when the Patient or Reception services is unavailable. A tutorial on how to do this using Hystrix can be found at: https://spring.io/guides/gs/gateway/ . Modify the API Gateway application:

- Add a dependency for `spring-boot-starter-netflix-hystrix`
- Use a Hystrix filter to set a FallbackUri for the request to the patient service.
- Write a fallback method which returns: “Patient information is currently unavailable”.
- Do the same for the request to the Reception service.

Run and test your API Gateway service. Don’t forget to recreate the JAR and Docker image!

3 Docker swarm: Messaging services

In the section we will investigate how to use multiple instances of services when they communicate using messaging. Start by adding the remaining services to the Docker compose file:

- Add all the remaining services (i.e. reception, reception_db, finance, mongo, ward, ward_db, zookeeper and kafka) to your Docker swarm compose file.
- For the ward and finance services use 2 replicas

- Use only a single instance of the reception service.

Build and run your application. Test it by visiting:
http://localhost/reception/check_in_patient?patientId=1

- Which instances of your services received which messages and why?
  - Hint: Look at the logs for both the finance containers.

In the `application.properties` file you will notice that a `group` has been set for the `open_invoice` message but has not been set for `check_in_completed`. Only 1 member of a group consumes a message. If the group option has not been set all instance will consume the message. Fix this issue so that only one instance of each service consumes a message:

- Add the group property to all messages for all services.

Build, run and test your application.

4 Monitoring and managing your application

To automatically scale our application when there is a high demand we need to first setup producing metrics. We will modify the Patient application to output the number of seconds it takes performing HTTP requests. If the rate of this value becomes too high the number of instances of the Patient services should be increased. To simulate many requests being made you will use JMeter.

The services you will deploy are shown in Figure 2. You may want to use a docker-compose file which only has these services - so you don't have to wait for the others to start-up.

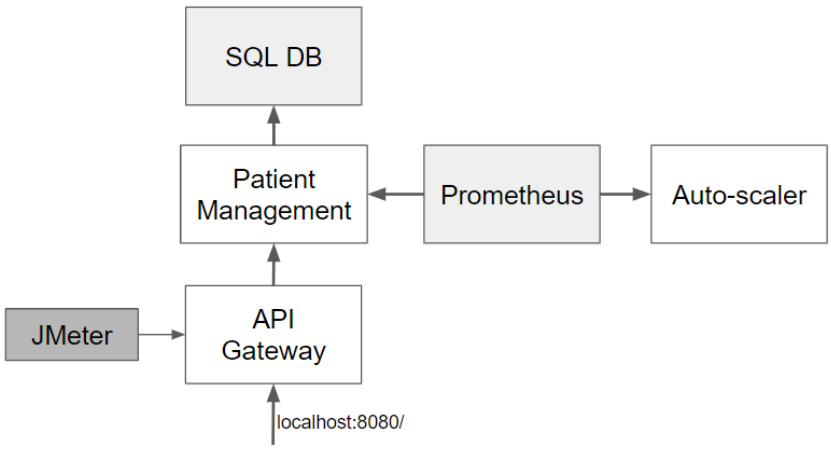

Figure 2 Containers you will deploy in this session. JMeter is used for testing, to simulate many users accessing the services; thus is not added to the docker-compose file.

4.1 Monitoring

Spring Boot can provide metrics about our application, such as how many HTTP requests are received. This allows developers to monitor the performance and reliability of a system. You will be using the metric stream to upscale and downscale your application depending on the number of requests. In Spring Boot metrics can be provided by adding a dependency and setting some properties.

Modify the Patient service:

- Add a dependency for `spring-boot-starter-actuator`

Actuator alone provides us with many metrics, including what HTTP requests have been made and the health of the service. However, we want to be create a stream of metrics and use a Prometheus service to aggregate the metrics for all of our containers.

- Add dependencies for `micrometer-core` and `micrometer-registry-prometheus`
- Add the following settings to the `applications.properties` file:

```
management.security.enabled=false
management.endpoints.web.exposure.include=*
management.endpoint.metrics.enabled=true
management.endpoint.prometheus.enabled=true
management.metrics.export.prometheus.enabled=true
```

Rather than having to re-start all your services to apply your changes to the Patient service, let's perform a rolling update.

4.1.1 Rolling update

A rolling update will stop containers one at a time and re-launch them with the new version of the image.

Run the command `docker service ps <stack_name>_<service_name>`. You will notice that all instances of the Patient services are currently running the `hospital-app-patient-management:latest` image. We will update these to a new image.

- Build the `hospital-app-patient-management` image but this time provide a version tag, e.g. use `hospital-app-patient-management:v2`
    - If you haven't called your image "`hospital-app-patient-management`" then use whatever your image name is.

Update your `Pateint` service to the `hospital-app-patient-management:v2` image using the `service update` command with the following options (https://docs.docker.com/engine/reference/commandline/service_update/#options):

- The delay between updates set to 2 seconds
- If the update fails the image should be rolled back

Test your update has worked. Execute the following curl command from within the Patient container: `curl 0.0.0.0:2222/actuator.` This should provide you with the list of available actuator endpoints.

4.1.2 Aggregating metrics

Prometheus is a systems and service monitoring system. It collects metrics from configured targets, provides a means of querying and visualizing and enables alerts to be triggered e.g. when a services is unavailable or too many requests are made.

We will use the Prometheus Docker image: https://hub.docker.com/r/prom/prometheus/. First we need to write a Prometheus configuration file to tell it to monitor the Patient service. How to write the configuration file can be found at: https://prometheus.io/docs/prometheus/latest/configuration/configuration/

Create a `prometheus.yml` file, and add the following configuration options:

- Scrapes all targets every 10 seconds
- Scrapes the Patient service's metrics – configure this using the `scrape_config` option.
    - The metric path is '/actuator/prometheus'
    - To find all the instances of the Patient services you need to use a `dns_ds_config`
        - Use type A
        - For Docker services the name parameter should be set to `'tasks.<service name>'.` Thus, for your patient service should be: `'tasks.patient'`
        - Set the port to 2222

Add a Prometheus service to your Docker compose file:

- For the image you should use: https://hub.docker.com/r/prom/prometheus/
- It should depend on patient
- Port 9090 needs to be externally accessible
- Use the volumes option to make your `prometheus.yml` file available to the container at: `/etc/prometheus/prometheus.yml`.

Run the stack deploy command, **without** removing the running stack first. Docker will check if any of the services need to be updated and will deploy any new services for you.

Check Prometheus has successfully connected to your Patient service by visiting http://localhost:9090/targets

Click on Graph and try out a few queries.

- What metrics are available?
- In total how many seconds has your Patient service carried out requests for?
    - You need to visit http://localhost/patient to start receiving metrics about the number of requests to this page.
- What does the following query do?
    - `avg(rate(http_server_requests_seconds_count{uri=~".*patient.*"}[1m]))`
    - If you are not sure try breaking it up. e.g. first execute: `http_server_requests_seconds_count{uri=~".*patient.*"}`

4.2 Load generation

Tools such as JMeter can be used to test how well a system works under heavy loads. In this section you will create and run a JMeter test plan to test how well your web shop application works when accessed by multiple users. Inside your local VM:

- Download JMeter http://jmeter.apache.org/download_jmeter.cgi
- Unzip it and run `./bin/jmeter.sh`
- Using the instructions found at http://jmeter.apache.org/usermanual/build-web-test-plan.html create a test plan which:
    - has 50 users
    - sends requests to `localhost/patient`

Run a container from the `cirit/jmeter:master` docker image which:

- uses the –v option to make the directory containing your `.jmx` available to the container
- passes `-t <your_jmeter_test.jmx>` to the container's entrypoint
- uses detached mode
- set the network to `host`

Test the load has been correctly generated using Prometheus.

4.3 Automatic scaling

Unexpected high demand can damage an application's performance, and risk a company losing customers. However, continuously running large numbers of servers and containers can be costly. Auto-scaling services based on demand is thus needed. In this Lab session we will just perform upscaling.

4.3.1 Prometheus alert

We will configure Prometheus to trigger an alert when a large number of requests are being sent to the Patient services. For how to set up an alert see: https://prometheus.io/docs/prometheus/latest/configuration/alerting_rules/

- Create an `alert_rules.yml` file and write an alter with the following options:
  - A sensible name, e.g. patient_request_rate_error
  - The alert should be triggered if `avg(rate(http_server_requests_seconds_count{uri=~".*patient.*"}[1m]))` exceeds 0.15
  - Provide an annotation containing a description of the alert.
- Add `alert_rules.yml` to a list of `rule_files` in your `prometheus.yml` file
- Add a volume to the Prometheus service in your Docker compose file, so it can use your `alert_rules.yml` file.

Test the alert works by (re-)deploying your stack and running the JMeter container as before. In the Prometheus UI you should be able to see your alert being triggered from the Alerts page.

4.3.2 Scale your application

Once the alert has been triggered we need to scale the Patient service. We have provided you with a Dockerfile, package.json and the start of a Javascript application that will be used to scale the Patient service.

- Download the `auto_scale` directory from Minerva.
- Complete the index.js file. The instructions can be found within the comments.
- Add the auto-scale application as a service to your Docker compose file.
  - Add the following volume: `/var/rub/docker.sock:/var/run/docker.sock`
    - This gives the container access to the instance of Docker running on your host machine.

Prometheus needs to know where to send the alert too.

- Add the following lines to your `prometheus.yml` file:

```
alerting:
    alertmanagers:
        - scheme: http
            path_prefix: /alert
            static_configs:
        - targets: ['<name of your scaling service>:9093']
```

- In the Docker compose file the Prometheus services needs to be linked to your auto-scale service.

Test your automatic scaling works. Deploy your stack, run the JMeter container as before and check how many instances of the Patient service are running.