ECM3408 - Enterprise Computing – Alleys

1. **Microservice Identification**



Alleys provides a driver service like Uber using a REST microservice architecture. Microservices are implemented with a stateless, client-server motivation in mind. Manipulation of resources is performed through CRUD with HTTP methods. The user is unaware of the individual services behind the endpoint which they interact with.

Alleys comprises four main microservices, organised with the orchestration pattern. The Rider service focuses on acting as an endpoint where the rider requests the cheapest rate for their journey. This service makes requests to the other services. Rider exists only to request these other services. It maintains the state associated with each stage in the process and does not maintain application data. It contains a GET method to retrieve the cheapest driver and fare. The whole process is idempotent as resources are read and the result will be identical provided the parameters are the same. The logic and data itself lives in the microservices that are responding to the rider. Each service can be called by the orchestrater with an identical payload many times, but the service will respond with an identical result each time. The only exception is the non-idempotent POST in the Roster service, as this creates a new resource. Therefore, asynchronous calls to other services are not a problem.

The orchestration pattern works well on well-defined small services like in Alleys, however, if tens or hundreds of services were required in a large company then a system like Choreography may be more suitable. This is because Choreography avoids bloated composer services. Heterogeneity is also slightly improved with Choreography. The endpoint service does not request other services directly, but it publishes an event which they can consume. This means   
that, provided the technology used for a service can consume that event, it can be implemented trivially. One primary reason to use Orchestration over Choreography is the difference in complexity. Choreography is not as appropriate in a small system as some form of message queueing system like RabbitMQ is needed for services to publish events to. If the services are dancers, then this makes the message queue the music, however, a format for these messages or the “notes” is also needed [1].

The Mapping Service implements a GET method which focuses on getting directions data from the Google Maps API and parsing the result. This is idempotent, it causes no storage side effects. The SurgePricing service also implements an idempotent GET. This service focuses on calculating the final rate using the surge pricing algorithm and sending it back to the client. Finally, the Roster is an example of storage as a service. It implements GET, POST, PUT, and DELETE methods. As the Roster uses POST to create a resource in the mongo database, it is not fully idempotent. The PUT method is idempotent if the driver exists, but not idempotent if the driver does not already exist as it will create the driver instead []. The Roster focuses on driver management. A driver may be created, updated and deleted. A GET method is also implemented which gets the cheapest driver and the count of drivers in the roster.

Each of the services presented above exposes a fixed API on a separate process. They listen automatically for requests until a client contacts them. Thus, one can request data from each service individually and it will return that data without having to know about other services. The only exception in such a pattern in the rider service which needs to request data from the other services. Alleys is loosely coupled. Provided the fixed API is followed, changes to a service’s code will not affect others. For instance, if the surge pricing algorithm now quadrupled the rate on A roads, this change would not impact on the Mapping, Roster, or Rider services. Though the user may now be annoyed. Alleys is highly cohesive too. Each service focuses on one part of Alleys, whether it be mapping, price calculation or data manipulation. Due to this microservice implementation, failure of a service need not be critical. Replicas can easily be provided where user traffic is diverted in the case of the original failing. The original can then be restarted as if nothing happened.

1. **Microservice Implementation – Node js**
2. **Microservice Deployment – Docker**
3. *No Docker-Compose*

Each microservice that comprises Alleys is run in a networked docker container. A dockerfile to build the required image is presented below for each service. In this format, each microservice has its own subdirectory which contains the Node js file and corresponding docker file. In development the docker-compose mechanic was used instead. Please see section “b)” to see the dockerfiles for docker-compose and the yaml file required.



Without docker-compose a docker network must be created manually for the containers to attach to and be able to communicate with each other. This can be achieved with the command:

Then the images must be built from the Dockerfiles by changing to the relevant microservice directory:

Lastly, attach the required containers to the network:

To attach a mongo database to the network that the roster service can use please use this command:

Exposing the ports in the command enables easy connection from a DBMS GUI like Mongo Compass; use docker-machine ls to discover the IP required to connect. Once all the containers are live and, in the network, one can use the curl commands provided in the last section to test Alleys.

1. *With Docker-Compose*

The whole of Alleys can be provisioned and run as separate microservices in containers using a single docker-compose command. Images are built automatically and used to run containers in a network that is also created automatically. The blueprints for this process are written in a   
docker-compose.yml file. Both the dockerfiles for the services and the docker-compose file can be found below.



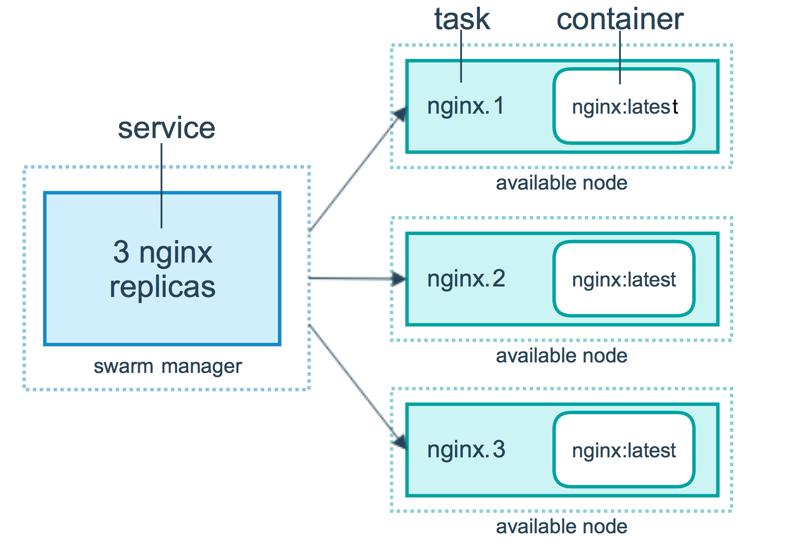
The docker-compose file allows you to define services that will be run as containers. Each service is a group of containers where one can define replicas of the same microservice [2]. For instance, if I define a mapping microservice and state I would like four replicas then four identical containers of the same microservice are run as a docker “service”. In production one can use the docker swarm manager to define a service and run many replicas of it as tasks. With Alleys this could mean having multiple HTTP servers listening on different swarm nodes. The swarm manager will subsequently schedule these tasks to run the containers which implement them. If a container fails, or does not pass a pre-defined health check, it is automatically replaced by a replica which the swarm builds. Figure x shows the swarm manager in action.

Fig. x. An example of the docker swarm manager creating replicas of an nginx web server. If one fails it is replaced automatically. Taken from: https://docs.docker.com/engine/swarm/how-swarm-mode-works/services/#services-tasks-and-containers

The services are named fields in the compose file. For each service one specifies a build context and dockerfile, just as with the usual docker build command. Then any commands to be run can be provided, in this case a node server is started. The port mapping that is usually specified in the run command is set next. The environment field allows one to define an environment variable, in this case I expose the service port as an environment variable. This means that in my Node JS code I can access the ports I need dynamically.

The volumes field is like the dockerfile copy but it also mounts the locations provided. Thus, one can change a microservice file and restart it with a single command without having to affect other services. The depends\_on field tells docker to start the listed services before starting the current service. Finally, the networks field of each service tells docker which network to attach the service to. The final separate networks field in the compose file instructs docker to create a bridged network to which the services will all connect.

When first running the services, you must use this command in the directory where the compose file is located:



This provisions all the images, interprets the dockerfiles, creates the network and attaches all the service containers to it. To take it down again use the same command but change “up” to “down” and omit the --build flag. If you do not change the dockerfiles or compose file you may start the containers in subsequent runs by using the original command and omitting --build. After the build command is run docker-compose caches the container recipes making subsequent up commands much faster.

The command to restart a service while all the services are live is:



This command emphasises the Ease of Deployment characteristic of microservices as it simplifies the equivalent process for deployment without docker-compose.

The command window where docker-compose is running can get clogged up with logging from different services. To see output and logging from a single service use the following command:

**IIII. Microservice Testing – Curl Commands**

Hello World.

Afterwards, try any number of these CURL commands I have provided:

If you change a docker file with compose use docker-compose up –build

Only up every other time

Docker- compose down

Docker-compose restart service

Docker-compose logs -ft service

{

"name":"MongoError",

"err":"E11000 duplicate key error index: test.test.$country\_1 dup key: { : \"XYZ\" }",

"code":11000,

"n":0,

"connectionId":10706,

"ok":1

}

<https://nordicapis.com/asynchronous-apis-in-choreographed-microservices/>

<https://docs.docker.com/docker-cloud/getting-started/your_first_service/#create-and-deploy>

<https://docs.docker.com/engine/swarm/how-swarm-mode-works/services/>