

La connaissance de l'architecture applicative de microservices

Table des matières

I. Architecture de microservices	3
II. Exercice : Quiz	6
III. Mise en place d'une architecture de microservices avec Docker	7
IV. Exercice : Quiz	14
V. Essentiel	15
VI. Auto-évaluation	15
A. Exercice	15
B. Test	15
Solutions des exercices	16

I. Architecture de microservices

Durée : 1 h

Environnement de travail : PC connecté à Internet

Contexte

En 2008, Netflix a échappé de justesse au risque de perdre la confiance de ses utilisateurs. En réalité, au cours de cette année, plusieurs utilisateurs de la plateforme ont été incapables d'accéder à leur espace d'abonnement. L'explication donnée par les développeurs est qu'une virgule mal placée dans le code a compromis l'ensemble des services. Cette situation aurait pu être évitée si la plateforme n'était pas basée sur une architecture monolithique.

Alors, pour pallier ce danger à l'avenir, Netflix a décidé d'opérer une migration vers le cloud public avec en fond le déploiement de ses services selon une méthodologie de type SOA. En clair, il était question d'opérer une rupture progressive avec le monolithique pour passer à l'architecture de microservices.

C'est un *pattern* qui décentralise la composition d'une application. Il consiste à donner plus de visibilité aux codes, les isoler et les embarquer dans des paquets autonomes, interopérables et capables de réaliser une tâche spécifique. En clair, c'est le type de modèle qui facilite la prise en main d'une maintenance et d'une migration par les développeurs.

C'est là que les conteneurs entrent en jeu, car ils sont connus pour répondre au même besoin que l'approche des microservices. C'est la jonction entre la méthodologie et l'outil le plus approprié pour la création des applications décentralisées.

Mais bien qu'elle soit avantageuse, la conteneurisation des microservices est un exercice complexe. Il ne suffit pas de connaître les étapes techniques d'utilisation d'une plateforme Docker. Il faut nécessairement cerner les défis auxquels font face les microservices, les principes, et il faut maîtriser le choix des modèles.

Dans ce cours, nous vous aidons à réaliser avec succès vos projets de déploiement ou de migration d'applications évolutives.

Définition Les microservices

Les microservices, ou l'architecture de microservices, correspondent à un modèle architectural et organisationnel exploité dans la conception des applications. Il se distingue par son approche basée sur la décomposition des services ou applications en de petit blocs ou modules. Chaque module est un exécutable disposant de sa propre interface. Il permet d'accomplir une tâche spécifique et simple. Il est également doté des API REST qui lui permettent de communiquer avec d'autres blocs indépendants. Les microservices forment ainsi un ensemble volumineux et fonctionnel qu'on appellera « *application* ». En clair, c'est une méthodologie qui partage les caractéristiques des conteneurs applicatifs.

Définition Les conteneurs applicatifs et la conteneurisation

Un conteneur est un composant de développement informatique qui permet de virtualiser des applications dans un environnement système. À la différence des machines virtuelles et de leur hyperviseur, il embarque les codes, bibliothèques et dépendances des applications dans différents paquets exécutables, légers et autonomes.

Il offre aux applications un environnement d'exécution transportable et moins gourmand en matière de consommation des ressources machines. Il contribue aujourd'hui à accélérer les processus DevOps et réduit les coûts supportés par les entreprises.

Leur création, déploiement, maintenance sont regroupés dans un processus appelé « *conteneurisation* ». Sa mise en œuvre est simplifiée par des outils logiciels appelés :

- Plateforme de conteneurisation (exemple : Docker),
- Plateforme d'orchestration ou de mise à l'échelle de conteneur (exemple : Kubernetes).

Histoire de l'architecture de microservices

Le principe des microservices n'est pas si récent. Il trouve ses fondements dans la méthode de conception **SOA** (Architecture Orienté Service) vulgarisée par Martin Fowler en 1997. D'ailleurs, l'architecture de microservices est aujourd'hui considérée comme une variante de cette méthodologie de découpage des applications en plusieurs petits services faiblement couplés.

Si son principe est ancien, l'essor de l'architecture de microservices ne s'est produit qu'en 2012. C'est l'année où une célèbre plateforme de streaming a choisi cette approche pour un déploiement de ses services sur le cloud. Ce pas de géant a depuis lors incité un grand nombre d'entreprises à revoir leur architecture.

Fonctionnement des microservices

Une application par microservices est constituée de plusieurs services. Chacun de ses composants est caractérisé par un processus, une base de données et des tâches propres. Par exemple, dans l'application, un microservice peut gérer l'authentification tandis qu'un autre peut assurer l'affichage des interfaces. Il s'agit donc d'un fonctionnement et d'une organisation décentralisés de l'architecture des applications.

Les composants de base de l'architecture de microservices

De façon standard, l'architecture de microservices est constituée de plusieurs éléments. Certains de ces composants sont plus importants que d'autres, mais en général nous pouvons retenir :

- **Service autonome** : il s'agit de l'élément principal de l'approche microservices. Comme évoqué plus haut, ce sont des services indépendants, faiblement couplés et interopérables. Ils sont écrits dans des langages informatiques divers.
- **Clients** : ce sont des centres d'opération au niveau desquels les appels aux différents services sont pris en charge.
- **Conteneurs** : tel qu'on les connaît déjà, ce sont des paquets exécutables qui servent à encapsuler, isoler et autonomiser les services. Ils sont utiles dans la mesure où ils offrent aux applications un environnement d'exécution indépendant de la machine ou de son infrastructure. En leur sein, on retrouve du code d'application, des bibliothèques et des dépendances.
- **Maillage de service** : il s'agit des configurations qui permettent de relier les services. Elles apportent plus de sécurité au processus de développement et favorisent la création d'applications sans passer par le code.
- **Base de données** : comme on peut déjà l'imaginer, il s'agit des structures de données privées mises à disposition de chaque service.
- **Passerelle API** : elle constitue un point de connexion ou de contact des microservices avec d'autres entités, par exemple un utilisateur ou un programme. À elles seules, elles permettent la composition, la traduction et le routage des requêtes.
- **Contenu statique** : il est responsable du déploiement du contenu sous une forme statique sur des serveurs cloud. Ce contenu est par la suite servi aux utilisateurs *via* un réseau de partage.
- **Fournisseurs d'identité** : il s'agit des éléments qui contrôlent l'accès à la base des informations d'identité des utilisateurs. Ils sont parmi les composants les plus légers de l'architecture de microservices.
- **La gestion** : il s'agit des fonctionnalités qui permettent aux utilisateurs de paramétrer un processus exécuté par l'architecture de microservices.

- **Formats de messagerie** : il s'agit des protocoles qui mettent en œuvre des schémas de communication synchrone ou asynchrone dans les services.
- **Découverte de service** : il s'agit des éléments qui permettent de repérer la position des instances de services dans l'architecture. Ils sont nécessaires car la position de l'instance n'est pas statique, elle varie au cours de l'exécution des processus.

Les avantages de l'architecture applicative des microservices

L'architecture des microservices est avantageuse à plusieurs égards. De manière succincte, elle permet de :

- **Réduire les efforts et améliorer l'agilité en phase de mise à l'échelle** : Pour faire évoluer une application microservices, les développeurs peuvent se séparer en plusieurs groupes. Chaque équipe peut se pencher sur un microservice particulier et permettre ainsi d'accélérer le projet sans craindre de devoir faire plusieurs tests.
- **Déployer l'application indépendamment de l'ensemble de ses microservices** : le fonctionnement décentralisé et autonome des microservices permet d'exclure certains services problématiques en phase de déploiement. Cela se fera sans aucune crainte de devoir revoir tout le code et de refaire de multiples tests. La composante retirée peut être corrigée pour faire l'objet d'une mise à jour ultérieure.
- **Mieux isoler les pannes** : en phase de production, l'architecture de microservices permet d'éviter les dysfonctionnements complets des applications. Lorsqu'une défaillance survient, cela n'affecte que les services concernés. Les utilisateurs peuvent continuer à bénéficier des autres fonctionnalités de l'application. De plus, ce fonctionnement permet de vite identifier le bout de code en cause dans cette situation.
- **Éviter la dépendance vis-à-vis des piles technologiques** : les microservices sont autonomes et chacun est bâti sur une technologie spécifique. Lorsqu'ils sont déployés ensemble, ils réduisent la dépendance vis-à-vis d'un langage ou d'une base de données, par exemple. Cela permet de rendre l'application plus flexible et de l'optimiser pour les enjeux d'évolutivité.
- **Concevoir des applications moins gourmandes** : associée au processus de conteneurisation et à la technologie du cloud, l'approche microservices permet de réduire les sollicitations des ressources machine. C'est donc une solution qui permet aux entreprises d'optimiser leurs coûts.
- **Disposer d'un code réutilisable** : la fragmentation du code, lorsqu'elle est bien opérée, permet son utilisation ultérieure. Par exemple, un service d'authentification des utilisateurs peut être reconduit dans un autre projet de développement. On peut donc créer de nouvelles applications en partant de zéro et sans toucher profondément au code.

Usage de l'architecture de microservices

La tendance est à l'architecture de microservices. Les entreprises l'exploitent pour accélérer et optimiser la cible de développement des différents outils logiciels. Cependant, les microservices reviennent le plus souvent lorsqu'il est question d'usages particuliers. Il s'agit par exemple :

- **Des traitements de données** : la mise à disposition d'une base propre pour chaque service élargit le volume et le panel d'informations exploitées par l'application. Il en est de même pour le nombre de requêtes traitées par l'application de façon simultanée. Tout ceci fait d'une telle application une solution de traitement rapide et performante des données.
- **La migration web** : les microservices permettent d'éviter les pertes de temps durant des projets de migration. En effet, cela évite de devoir faire un réajustement qui concerne l'ensemble du site. Les travaux nécessaires sont ciblés et cantonnés à quelques composants, comme ceux opérant avec le nom de domaine ou les interfaces utilisateurs.
- **Contenu multimédia** : tout comme Netflix, les entreprises spécialisées dans la distribution de contenu multimédia à grande échelle ont beaucoup à gagner en adoptant l'architecture de microservices. Grâce à ses multiples API, elle constitue une solution adéquate pour répondre aux nombreuses sollicitations quotidiennes de connexion à des sous-domaines.

- **Transaction et factures** : l'architecture de microservices est adoptée de plus en plus souvent dans les projets de développement de solution de facturation ou de transaction. La raison est simple : elle garantit l'effectivité des opérations et évite des pertes conséquentes pour les utilisateurs. Ceci est dû à une bien meilleure tolérance aux pannes. Avec la solution monolithique, les importantes charges de travail et le caractère homogène sont susceptibles d'exposer l'application à un risque de panne générale, qui peut causer la perte des données.

Remarque Microservices et architecture monolithique

La montée en puissance des microservices fait suite aux difficultés des entreprises à assurer la croissance de leur service avec l'approche monolithique. En réalité, avant les microservices, l'architecture monolithique était le standard utilisé dans l'univers du DevOps.

C'est une approche qui prône l'unification et la centralisation des codes, des bibliothèques et des dépendances. Ceci dit, il permet de construire l'application comme une unité volumineuse qui fournit une panoplie de services. Cette méthodologie a été pendant longtemps plébiscitée pour plusieurs raisons :

- La charge cognitive du développeur est grandement réduite,
- La simplification et l'accélération du processus de développement,
- La facilité de la prise en main et du déploiement d'un fichier exécutable unique,
- Une seule API pour échanger avec d'autres applications,
- La gestion rapide des tests et des travaux de débogage, qui sont plus centralisés.

Cependant, avec des besoins de mise à l'échelle de plus en plus importants, ces avantages sont devenus les limites de la méthodologie. Il s'est avéré qu'un système centralisé et volumineux est beaucoup plus difficile à maintenir ou à mettre à jour. Par exemple, rechercher une erreur dans ses codes reviendrait à trouver une aiguille dans une botte de foin. Il faut du temps pour réaliser de nombreux tests. Enfin, à chaque modification, il est nécessaire de déployer une nouvelle mise à jour.

Face à tous ces problèmes, l'approche par microservices se positionne comme une alternative envisageable.

Exercice : Quiz

[solution n°1 p.17]

Question 1

Qu'est-ce qu'un microservice ?

- ☐ Un morceau de code
- ☐ Un petit programme qui représente une logique discrète qui s'exécute dans une limite bien définie sur du matériel dédié
- ☐ Un style de conception utilisé dans la programmation orientée objet et fonctionnelle
- ☐ Un style de conception pour les systèmes d'entreprise basée sur une architecture de composants faiblement couplés

Question 2

Laquelle des réponses suivantes est un avantage des microservices ?

- ☐ Ils ne nécessitent pas beaucoup d'expertise pour programmer
- ☐ Tout composant de microservice peut changer indépendamment des autres composants
- ☐ Ils sont si petits que les développeurs peuvent généralement en écrire de très puissants avec quelques lignes de texte
- ☐ Ils sont faciles à gérer

Question 3

Parmi les réponses suivantes, laquelle est un inconvénient des microservices ?

- ☐ Les microservices nécessitent beaucoup de surveillance pour fonctionner efficacement
- ☐ Les microservices sont très difficiles à gérer à grande échelle
- ☐ Les 2
- ☐ Aucune des 2

Question 4

Les microservices sont étroitement liés à :

- ☐ SEO
- ☐ Paas
- ☐ API
- ☐ AWS

Question 5

Dans quel cas une architecture de microservices serait-elle favorable pour une entreprise ?

- ☐ Si une architecture cloud nécessite une évolutivité
- ☐ Si une application bénéficie d'un faible couplage
- ☐ Les 2
- ☐ Aucune de ces réponses

III. Mise en place d'une architecture de microservices avec Docker

Il existe un grand nombre de pièges dans le processus de développement d'un microservice. Ces facteurs de risque sont pour la plupart liés aux défis qui se posent au microservice. Il est important de connaître ces éléments, mais aussi les solutions pour y répondre, à savoir : la maîtrise des principes, des modèles et surtout de la méthodologie propre à l'outil de conception choisi.

Les principes d'une bonne architecture de microservices

Une architecture de microservices efficiente doit répondre à certains principes essentiels. Sans leur respect, on risque de passer à côté des avantages qu'un microservice peut apporter dans une migration ou dans le développement de nouvelles solutions. Les principes les plus importants sont :

- L'évolutivité,
- La décentralisation,

- La flexibilité,
- La résilience des services,
- La disponibilité,
- L'isolation des bugs,
- L'intégration des API,
- L'équilibrage des charges,
- La livraison continue DevOps.

Les modèles d'une architecture de microservices

Il existe une longue liste de modèles de microservices utilisés par les entreprises. Mais les plus importants à connaître sont les suivants.

Les bases de données microservices

L'architecture de microservices propose 2 modèles d'organisation des données. Le premier est dénommé « *base de données par service* ». Ce modèle prône la mise à disposition d'une base de données pour chaque service. Cette base est propre à ce service et n'est accessible que par les processus de ce dernier. La séparation des données permet une meilleure compréhension des microservices dans leur ensemble et offre une plus prompt entrée en matière pour les nouveaux membres de l'équipe DevOps. Conclusion : cela accélère le développement et la mise à l'échelle. Le seul point négatif est le risque de défaillance qui peut survenir lorsque la base ne répond pas aux requêtes.

Le deuxième modèle est dit de « *base de données partagées* », c'est l'opposé du modèle précédent. Cette option permet de concentrer les données dans un même système. Elles seront donc prises en charge par un seul et même système de gestion. Cependant, ce choix revient à renoncer à ce qui garantit l'évolutivité, l'autonomie et la robustesse des microservices.

Approvisionnement en événement

D'après ce modèle, chaque changement opéré sur l'application doit concourir à l'enregistrement d'un événement. Ces derniers intègrent une base de données qui permet au développeur de cerner les besoins d'évolution ou de maintenance des microservices.

L'agrégation

Selon ce modèle, il faut mettre en place un composant appelé « *agregator* » qui combine les données des différentes bases pour construire une réponse à une requête. De ce fait, chaque variation d'état de l'application est chargée de nouvelles informations provenant de différentes sources.

Les passerelle API

C'est un modèle qui permet de créer des microservices autonomes tout en proposant une interopérabilité de ces derniers. Il prône le routage des requêtes d'un service vers les autres pour permettre un effort commun de résolution de problèmes. D'autre part, la passerelle offre une porte d'entrée pour la réception et la transmission des requêtes.

Chaîne de responsabilité

Il s'agit ici de mettre les services en chaîne et d'ainsi créer une sortie unique et consolidée des requêtes. À titre d'exemple, une requête émise à l'endroit d'un service spécifique est d'abord reçue par un service au bout de la chaîne. Ce dernier doit par la suite la transmettre au service qui le suit, ainsi de suite jusqu'à ce qu'il parvienne au service ciblé par la requête. À chaque transmission, la chaîne de service émettrice s'associe pour fournir une version plus consolidée de la requête.

L'étrangleur

Avec ce modèle, les applications peuvent être mises l'une à côté de l'autre avec la même adresse. Mais le domaine de l'une prédominera sur l'autre. Résultat : l'application prédominante remplace la seconde application, voire l'étrangle. C'est une option utilisée pour remplacer progressivement une architecture monolithique.

Le disjoncteur

C'est une approche qui permet de stopper un processus lors que le service qui doit fournir la réponse ne répond pas. Une barrière est donc déployée pour empêcher l'envoi continu de requête inutile et qui encombre le réseau.

Méthode

Si les principes de base et les modèles sont bien maîtrisés, alors le reste devrait paraître simple. Il suffit de suivre les 6 étapes appropriées que nous avons définies ici.

Étape 1 :

La première action à mener est l'installation d'une version de Docker adaptée au système d'exploitation installé sur votre machine.

Étape 2 :

Après l'installation de Docker, il faut passer à l'installation du Docker Compose. Cela passe par le téléchargement, l'exécution et la définition des autorisations de fichier.

Étape 3 :

Il faut préparer son environnement en créant un répertoire pour le microservice et en définissant sa structure.

Étape 4 :

C'est le moment le plus fastidieux. Il consiste à créer les différents fichiers Dockerfile des images de conteneur (Nginx, PostgreSQLName, Web) puis à les configurer.

Étape 5 :

Dès à présent, on utilise Docker Compose pour définir et configurer les connexions pour rendre le microservice opérationnel.

Étape 6 :

Pour finaliser le tout, il faut tester le travail afin de s'assurer du fonctionnement effectif de l'application.

Exemple

Dans notre exemple, nous voulons créer et déployer un microservice. Pour l'exercice, nous utiliserons Docker ainsi que son composé. Pour nos commandes, nous allons procéder avec les privilèges d'un utilisateur classique. Cependant, nous mettrons à profit la commande « `sudo` » pour réaliser des opérations qui nécessitent des droits plus importants.

Nous allons maintenant commencer par le téléchargement et l'installation de Docker CE. Dans notre entreprise, nous installerons l'outil de conteneurisation sur un environnement Ubuntu. Une fois cette première installation terminée, nous allons passer à Docker Compose. Il suffit de se rendre sur GitHub pour obtenir la dernière version, ou utiliser la ligne de commande suivante. Après l'opération, nous allons définir les autorisations en nous servant de ce code :

```
1 sudo chmod +x /usr/local/bin/docker-compose
```

Juste après les installations de base, nous passons maintenant à la configuration de notre environnement. Pour ce faire, nous allons mettre en place notre répertoire en usant de la ligne de commande qui suit :

```
1 mkdir flask-microservice
```

Nous allons organiser le dossier précédemment créé avec des sous-dossiers pour chacun des composants de notre microservice. Voici la commande à utiliser :

```
1 cd flask-microservice
2 mkdir nginx postgres web
```

Nous allons maintenant déployer et configurer nos 3 images de conteneur Nginx, postgres et Web.

Déjà du côté du serveur web Nginx, il nous faut en premier créer le sous-dossier « *nginx* ». À l'intérieur de ce répertoire, nous allons créer le fichier Dockerfile de l'image du serveur web. Voici les 2 lignes de code que nous utilisons :

```
1 from nginx:alpine
2 COPY nginx.conf /etc/nginx/nginx.conf
```

Toujours par rapport à l'image de Nginx, nous allons créer le fichier « *nginx.cnf* » dans le précédent puis y insérer les lignes de code suivantes :

```
1 user  nginx;
2 worker_processes 1;
3 error_log  /dev/stdout info;
4 error_log off;
5 pid       /var/run/nginx.pid;
6
7 events {
8     worker_connections 1024;
9     use epoll;
10    multi_accept on;
11 }
12
13 http {
14     include      /etc/nginx/mime.types;
15     default_type  application/octet-stream;
16
17     log_format  main  '$remote_addr - $remote_user [$time_local] "$request" '
18                      '$status $body_bytes_sent "$http_referer" '
19                      '"$http_user_agent" "$http_x_forwarded_for"';
20
21     access_log  /dev/stdout main;
22     access_log off;
23     keepalive_timeout 65;
24     keepalive_requests 100000;
25     tcp_nopush on;
26     tcp_nodelay on;
27
28     server {
29         listen 80;
30         proxy_pass_header Server;
31
32         location / {
33             proxy_set_header Host $host;
34             proxy_set_header X-Real-IP $remote_addr;
35
36             # app vient de /etc/hosts, Docker l'a ajouté pour nous !
37             proxy_pass http://flaskapp:8000/;
38         }
39     }
40 }
```

Après l'exécution des lignes de code précédentes, nous allons nous pencher sur le cas de l'image PostgreSQL. Ici, il n'est pas nécessaire de configurer un fichier Dockerfile. Nous récupérons juste la copie « *postgresql* » depuis le référentiel Docker Hub. Par la suite, nous créons un fichier de configuration « *ini.sql* » dans le dossier « *postgres* » précédemment préparé. Voici les lignes de code que nous utilisons pour configurer ce fichier :

```
1 SET statement_timeout = 0;
2 SET lock_timeout = 0;
3 SET idle_in_transaction_session_timeout = 0;
4 SET client_encoding = 'UTF8';
5 SET standard_conforming_strings = on;
6 SET check_function_bodies = false;
7 SET client_min_messages = warning;
8 SET row_security = off;
9 CREATE EXTENSION IF NOT EXISTS plpgsql WITH SCHEMA pg_catalog;
10 COMMENT ON EXTENSION plpgsql IS 'PL/pgSQL procedural language';
11 SET search_path = public, pg_catalog;
12 SET default_tablespace = '';
13 SET default_with_oids = false;
14 CREATE TABLE visitors (
15     site_id integer,
16     site_name text,
17     visitor_count integer
18 );
19 ALTER TABLE visitors OWNER TO postgres;
20 COPY visitors (site_id, site_name, visitor_count) FROM stdin;
21 1   linodeexample.com   0
22 \.
```

La prochaine action de configuration concerne l'image web. Afin de rendre le parquet fonctionnel, nous ajoutons au répertoire web différents fichiers. Nous nommons le premier d'entre eux « *python-version* ». Il contient juste la ligne suivante :

```
1 echo "3.6.0" >> web/.python-version
```

Cette ligne de code signale l'utilisation de la version 3.6 du langage Python. Le deuxième fichier à ajouter est le Dockerfile. Nous y plaçons le code suivant :

```
1 from python:3.6.2-slim
2 RUN groupadd flaskgroup && useradd -m -g flaskgroup -s /bin/bash flask
3 RUN echo "flask ALL=(ALL) NOPASSWD:ALL" >> /etc/sudoers
4 RUN mkdir -p /home/flask/app/web
5 WORKDIR /home/flask/app/web
6 COPY requirements.txt /home/flask/app/web
7 RUN pip install --no-cache-dir -r requirements.txt
8 RUN chown -R flask:flaskgroup /home/flask
9 USER flask
10 ENTRYPOINT ["/usr/local/bin/gunicorn", "--bind", ":8000", "linode:app", "--reload", "--workers", "16"]
11 Ensuite vient un troisieme fichier nommé linode.py qui contiendra le script que voici :
12 from flask import Flask
13 import logging
14 import psycpg2
15 import redis
16 import sys
17 app = Flask(__name__)
18 cache = redis.StrictRedis(host='redis', port=6379)
19 # Configurer la journalisation
20 app.logger.addHandler(logging.StreamHandler(sys.stdout))
21 app.logger.setLevel(logging.DEBUG)
```

```

22 def PgFetch(query, method):
23     # Se connecter à une base de données existante
24     conn = psycopg2.connect("host='postgres' dbname='linode' user='postgres'
25                             password='linode123'")
26     # Ouvrir un curseur pour effectuer des opérations de base de données
27     cur = conn.cursor()
28     # Interroger la base de données et obtenir des données sous forme d'objets Python
29     dbquery = cur.execute(query)
30     if method == 'GET':
31         result = cur.fetchone()
32     else:
33         result = ""
34     # Rendre les modifications apportées à la base de données persistantes
35     conn.commit()
36     # Communication étroite avec la base de données
37     cur.close()
38     conn.close()
39     return result
40 @app.route('/')
41 def hello_world():
42     if cache.exists('visitor_count'):
43         cache.incr('visitor_count')
44         count = (cache.get('visitor_count')).decode('utf-8')
45         update = PgFetch("UPDATE visitors set visitor_count = " + count + " where site_id =
46 1;", "POST")
47     else:
48         cache_refresh = PgFetch("SELECT visitor_count FROM visitors where site_id = 1;",
49 "GET")
50         count = int(cache_refresh[0])
51         cache.set('visitor_count', count)
52         cache.incr('visitor_count')
53         count = (cache.get('visitor_count')).decode('utf-8')
54     return 'Hello Linode! This page has been viewed %s time(s).' % count
55 @app.route('/resetcounter')
56 def resetcounter():
57     cache.delete('visitor_count')
58     PgFetch("UPDATE visitors set visitor_count = 0 where site_id = 1;", "POST")
59     app.logger.debug("reset visitor count")
60     return "Successfully deleted redis and postgres counters"

```

Nous finissons la liste des fichiers à ajouter au répertoire web avec le document « *requirement.txt* », voici le code qu'il contient :

```

1 flask
2 gunicorn
3 psycopg2-binary
4 redis

```

Nous en venons maintenant au paramétrage de Docker Compose. Le but d'une telle manœuvre est d'établir une passerelle de communication entre conteneurs. Pour ce faire, nous allons ajouter un fichier nommé « *docker-compose.yml* » dans le dossier « *flask-microservice* ». Dans ce dernier fichier, nous ajoutons ce script que nous avons préparé :

```

1 version: '3'
2 services:
3   # Définir l'application Web Flask
4   flaskapp:
5     # Construisez le Dockerfile qui se trouve dans le répertoire web
6     build: ./web
7     # Redémarrez toujours le conteneur quel que soit l'état de sortie ; essayez de redémarrer
    le conteneur indéfiniment

```

```

8  restart: always
9  # Exposez le port 8000 à d'autres conteneurs (pas à l'hôte de la machine)
10 expose:
11   - "8000"
12 # Montez le répertoire Web dans le conteneur à /home/flask/app/web
13 volumes:
14   - ./web:/home/flask/app/web
15 # Ne créez pas ce conteneur tant que les conteneurs redis et postgres (ci-dessous) n'ont
pas été créés
16 depends_on:
17   - redis
18   - postgres
19 # Liez les conteneurs redis et postgres ensemble afin qu'ils puissent se parler
20 links:
21   - redis
22   - postgres
23 # Passez les variables d'environnement au conteneur de flask (ce niveau de débogage vous
permet de voir des informations plus utiles)
24 environment:
25   FLASK_DEBUG: 1
26 # Déployer avec trois répliques en cas de défaillance de l'un des conteneurs (uniquement
dans Docker Swarm)
27 deploy:
28   mode: replicated
29   replicas: 3
30 # Définir le conteneur Redis Docker
31 redis:
32   # utilisez l'image redis:alpine : https://hub.docker.com/_/redis/
33   image: redis:alpine
34   restart: always
35   deploy:
36     mode: replicated
37     replicas: 3
38 # Définir le conteneur de proxy de transfert redis NGINX
39 nginx:
40   # construire le nginx Dockerfile : http://bit.ly/2kuYaIv
41   build: nginx/
42   restart: always
43   # Exposez le port 80 à la machine hôte
44   ports:
45     - "80:80"
46   deploy:
47     mode: replicated
48     replicas: 3
49   # L'application Flask doit être disponible pour que NGINX fasse des demandes de proxy
réussies
50   depends_on:
51     - flaskapp
52 # Définir la base de données postgres
53 postgres:
54   restart: always
55   # Utilisez l'image alpine postgres : https://hub.docker.com/_/postgres/
56   image: postgres:alpine
57   # Monter un script d'initialisation et le volume de données postgresql persistant
58   volumes:
59     - ./postgres/init.sql:/docker-entrypoint-initdb.d/init.sql
60     - ./postgres/data:/var/lib/postgresql/data
61   # Passer les variables d'environnement postgres

```

```

62  environment:
63      POSTGRES_PASSWORD: l1node123
64      POSTGRES_DB: l1node
65      # Exposer le port 5432 à d'autres conteneurs Docker
66  expose:
67      - "5432"

```

Après toutes les précédentes configurations, nous voici à la phase de test de notre microservice. Pour nous assurer que tout fonctionne, nous nous rendons dans le terminal et lançons le code suivant :

```
1 cd flask-microservice/ && docker-compose up
```

Comme réplique, nous observons alors un démarrage des services dans l'interface de sortie du terminal.

Exercice : Quiz

[solution n°2 p.18]

Question 1

Quelles sont les technologies couramment utilisées pour mettre en œuvre des microservices ?

- ☐ Docker
- ☐ Kubernetes
- ☐ Les 2
- ☐ Aucune de ces réponses

Question 2

Comment le traçage distribué est-il utilisé dans les microservices ?

- ☐ En tant que mécanisme permettant d'observer le comportement d'appels systèmes distincts entre et au sein des microservices
- ☐ En tant que mécanisme garantissant que les microservices défaillants sont correctement ressuscités
- ☐ En tant que mécanisme permettant de modifier le comportement d'un microservice lors de l'exécution

Question 3

L'utilisation du modèle base de données par service ne comporte aucun aspect négatif.

- ☐ Vrai
- ☐ Faux

Question 4

Le modèle base de données partagées qui possède un fonctionnement opposé à celui de données par service à cette particularité de concentrer les données dans une configuration spécifique qui est :

- ☐ Dans différents système
- ☐ Dans un même système
- ☐ Dans deux systèmes uniquement
- ☐ Uniquement dans un seul système différent

Question 5

La passerelle API est un modèle qui ne permet pas de créer des microservices autonomes.

- ☐ Vrai
- ☐ Faux

V. Essentiel

En définitive, une architecture de microservices est un pattern architectural d'organisation des composants d'une application. Il s'agit d'une approche qui se positionne comme la solution aux nombreuses difficultés rencontrées avec le modèle monolithique. Elle consiste à découper le code d'une application en plusieurs blocs exécutables, autonomes et faiblement couplés. Ils permettent d'exécuter des tâches spécifiques. Mais ces microservices sont aussi interopérables et communiquent les uns avec les autres *via* des API. Combinés, ils forment un tout fonctionnel qui permet à l'utilisateur de résoudre un problème.

Entre autres, les microservices favorisent une évolutivité rapide, efficace et efficiente. Mieux, ils apportent plus de flexibilité à l'application. Même quand la quantité des données est en pleine croissance, on peut toujours modifier l'application sans risquer de mettre à mal l'ensemble de son fonctionnement. Face à la panne, cela s'impose comme une solution salutaire. En effet, en cas de dysfonctionnement, seul le service concerné sera hors d'état de fonctionnement. De plus, cette architecture permet d'isoler facilement les erreurs puis de leur trouver des solutions correctives plus adéquates et de mieux réfléchir.

Derrière tous ces avantages se trouvent d'importants défis quant à la mise en place. En effet, c'est une opération qui nécessite de longues périodes de réflexion et de tentatives de dissociation des codes. Ce qui équivaut également à de nombreux tests durant le développement. Elle nécessite également des expertises de différentes natures, puisque les services ne seront pas basés sur les mêmes technologies. De plus, l'équipe DevOps doit avoir une maîtrise parfaite des principes, de la méthodologie et des outils de conception.

En clair, il faut du temps, de l'argent et des compétences pour mettre en place une architecture de microservices. Si ces ressources venaient à manquer, vous pouvez vous retrouver avec une application défailante. D'où l'importance de la mettre en place progressivement dans le cadre d'une rupture avec l'approche monolithique.

VI. Auto-évaluation

A. Exercice

Vous êtes un expert DevOps recruté pour accompagner la plateforme d'échange de cryptomonnaie CoinNet. De son lancement à ce jour, la plateforme a connu un accroissement fulgurant de son trafic. L'évolution de la base d'utilisateurs a évolué de telle sorte que la performance du système commence à en pâtir. Le système tombe désormais régulièrement en panne. Pour résoudre cette situation, l'entreprise est prête à investir les ressources qu'il faut pour renforcer l'architecture monolithique du site ou pour procéder à une migration vers les microservices.

Question 1

[solution n°3 p.19]

Quelle solution conseillez-vous entre l'approche monolithique et celle des microservices ? Quels sont les arguments qui justifient la pertinence de vos propos ?

Vous êtes un freelance spécialisé dans le développement avec Node.js. Vous venez d'être recruté pour la dockerisation des microservices *back-end* d'une application de gestion.

Question 2

[solution n°4 p.20]

Quelle est la démarche à suivre pour réaliser l'opération en local ?

B. Test

Exercice 1 : Quiz

[solution n°5 p.20]

Question 1

Quand les développeurs utilisent-ils les microservices ?

- ☐ Quand ils travaillent avec la nanotechnologie éphémère
- ☐ Quand ils veulent écrire des applications pour téléphone portable qui s'exécutent rapidement
- ☐ Lorsqu'ils doivent créer de grandes applications de niveau entreprise qui sont sujettes à des modifications fréquentes

Question 2

L'architecture de microservice permet efficacement de palier aux déficiences du pattern monolithique.

- ☐ Vrai
- ☐ Faux

Question 3

Les microservices ne sont destinés qu'aux petites entreprises.

- ☐ Vrai
- ☐ Faux

Question 4

Qu'est-ce qu'une API en architecture de microservices ?

- ☐ Un service qui expose les fonctionnalités d'une architecture à un minimum de 2 applications
- ☐ Un sous-composant d'une application qui fonctionne indépendamment
- ☐ Une interface qui fonctionne entre 2 applications indépendantes et leur permet de communiquer

Question 5


La SOA est définie comme un composant de service indépendant qui permet à 2 applications de communiquer.

- ☐ Vrai
- ☐ Faux

Solutions des exercices


Exercice p. 6 Solution n°1**Question 1**

Qu'est-ce qu'un microservice ?

- ☐ Un morceau de code
- ☐ Un petit programme qui représente une logique discrète qui s'exécute dans une limite bien définie sur du matériel dédié
- ☐ Un style de conception utilisé dans la programmation orientée objet et fonctionnelle
- ☒ Un style de conception pour les systèmes d'entreprise basée sur une architecture de composants faiblement couplés
-  Tel qu'il est défini, un microservice rassemble des composants de petite taille et faiblement couplés qui, ensemble, fournissent une panoplie de fonctionnalités applicatives.


Question 2

Laquelle des réponses suivantes est un avantage des microservices ?

- ☐ Ils ne nécessitent pas beaucoup d'expertise pour programmer
- ☒ Tout composant de microservice peut changer indépendamment des autres composants
- ☐ Ils sont si petits que les développeurs peuvent généralement en écrire de très puissants avec quelques lignes de texte
- ☐ Ils sont faciles à gérer
-  Les composants d'un microservice sont faiblement couplés et facilitent ainsi les travaux de retouche du code, sans pour autant créer des répercussions sur les autres composants.


Question 3

Parmi les réponses suivantes, laquelle est un inconvénient des microservices ?

- ☐ Les microservices nécessitent beaucoup de surveillance pour fonctionner efficacement
- ☒ Les microservices sont très difficiles à gérer à grande échelle
- ☐ Les 2
- ☐ Aucune des 2
-  La décentration de l'architecture d'un programme génère des charges de travail cognitives supplémentaires. Il faut un expert aguerri à sa mise en œuvre pour s'en occuper. Dans le cas contraire, on risque commettre des erreurs.


Question 4

Les microservices sont étroitement liés à :

- ☐ SEO
- ☐ Paas
- ☒ API
- ☐ AWS
-  L'utilisation d'une architecture de microservices implique obligatoirement la mise en place de passerelles de type API. C'est ainsi qu'une relation s'est établie ces dernières années entre les 2 notions.

Question 5


Dans quel cas une architecture de microservices serait-elle favorable pour une entreprise ?

- ☐ Si une architecture cloud nécessite une évolutivité
- ☐ Si une application bénéficie d'un faible couplage
- ☒ Les 2
- ☐ Aucune de ces réponses
-  On connaît les microservices comme étant basés sur un faible couplage des services. Cela permet d'assurer non seulement l'autonomie des services, mais aussi leur évolutivité.

Exercice p. 14 Solution n°2


Question 1

Quelles sont les technologies couramment utilisées pour mettre en œuvre des microservices ?

- ☐ Docker
- ☐ Kubernetes
- ☒ Les 2
- ☐ Aucune de ces réponses
-  Les conteneurs répondent parfaitement au principe de décentralisation, de maillage et de routage de requête *via* les passerelles, comme proposé dans l'architecture de microservices.

Question 2

Comment le traçage distribué est-il utilisé dans les microservices ?


- ☒ En tant que mécanisme permettant d'observer le comportement d'appels systèmes distincts entre et au sein des microservices
- ☐ En tant que mécanisme garantissant que les microservices défaillants sont correctement ressuscités
- ☐ En tant que mécanisme permettant de modifier le comportement d'un microservice lors de l'exécution
-  On connaît les outils de traçage pour leur capacité à cartographier les données et à aider à suivre les requêtes de bout en bout.

Question 3

L'utilisation du modèle base de données par service ne comporte aucun aspect négatif.

☐ Vrai

☒ Faux

 Ce modèle, bien que comportant de nombreux aspects utiles et performants, possède un risque de défaillance qui peut survenir lorsque la base ne répond pas aux requêtes, donc son utilisation contient au moins un aspect qui peut être considéré comme négatif.

Question 4


Le modèle base de données partagées qui possède un fonctionnement opposé à celui de données par service à cette particularité de concentrer les données dans une configuration spécifique qui est :

☐ Dans différents système

☒ Dans un même système

☐ Dans deux systèmes uniquement

☐ Uniquement dans un seul système différent


 Ce qui veut dire que le modèle base de données partagées permet de prendre en charge les données par un seul et même système de gestion, ce qui est un avantage mais comportant cependant l'inconvénient de ne pas avoir l'évolutivité, l'autonomie et la robustesse des microservices.

Question 5

La passerelle API est un modèle qui ne permet pas de créer des microservices autonomes.

☐ Vrai

☒ Faux

 Les passerelles API sont des modèles qui possèdent cette caractéristique que de créer des microservices autonomes en ayant la possibilité d'une interopérabilité de ces derniers, le routage des requêtes d'un service vers les autres est mis en avant pour une résolution de problème résolu grâce à un effort commun.

p. 15 Solution n°3

Nous conseillons l'implémentation d'une architecture de microservices. Mais nous prôtons une approche progressive et non de rupture brusque avec l'architecture monolithique.

Notre choix se porte sur les microservices pour plusieurs raisons. Tout d'abord, il s'agit d'une architecture qui simplifie l'évolutivité des applications. Avec des microservices autonomes faiblement couplés, il devient plus facile de comprendre et de retoucher le code d'une application. À l'avenir, la plateforme pourra être pensée par plusieurs petits groupes de développeurs qui interviendront sur différents blocs de services. Lorsqu'une panne surviendra, elle pourra très vite être localisée et corrigée en raison de l'isolement des services.

De plus, l'approche par microservices permet de se libérer de l'emprise d'une technologie unique comme cela aurait été le cas avec la démarche monolithique. Pour finir, c'est une solution qui garantit une croissance sereine pour les applications. En effet, les cas de dysfonctionnement ne sont plus à redouter car elle n'affecte qu'un petit bloc du système. On peut alors sereinement déployer des solutions plus appropriées en s'appuyant sur la maîtrise de l'expérience, sans être pressé par l'urgence.

Mais notons que tout l'argumentaire précédent est basé sur un cas où on parvient à mettre en place une architecture de services. Cela est fait intentionnellement, car il s'agit d'une méthodologie exigeante surtout dans sa phase de mise en œuvre. En effet, elle nécessite une maîtrise profonde des processus métier et service pour

réussir la division des codes. Il faut mettre en place plusieurs API plutôt qu'une seule. Il est nécessaire de réunir des compétences variées en raison du besoin d'utiliser des technologies différentes. Le déploiement aussi peut être problématique, car il est plus difficile d'installer plusieurs modules qu'un seul fichier exécutable.

En conclusion, si la transition est bâclée, nous risquons de mettre à mal le bon fonctionnement de l'application. Raison pour laquelle nous conseillons de déployer progressivement des microservices à côté des infrastructures monolithiques.

p. 15 Solution n°4

En prélude aux travaux, nous allons installer Docker Conteneur, le client Docker et le serveur du même nom en local sur notre machine. Nous installons également la version 110 de Node.js ou une version plus évoluée afin de profiter des meilleures performances du système.

Dans le cadre de ce projet nous ne commencerons pas de zéro. Dans le référentiel, nous allons récupérer et cloner un noyau de base. Il s'agit d'un paquet exécutable écrit en Node.js qui permet d'établir une communication avec API.

Nous allons maintenant dockeriser le précédent clone. Pour y arriver, il nous faut créer un répertoire Dockerfile. Dans ce dossier, nous allons créer un fichier du même nom et contenant le code de création de l'image.

Pour commencer, on met à votre disposition un noyau de base en Node.js de l'application finale souhaitée. Voici le code du fichier :

```
1 FROM node:14-alpine
2 WORKDIR /usr/src/app
3 COPY ["package.json", "package-lock.json", "./"]
4 RUN npm install
5 COPY . .EXPOSE 3001
6 RUN chown -R node /usr/src/app
7 USER node
8 CMD ["npm", "start"]
```

Nous allons maintenant placer notre Dockerfile dans le même dossier que l'application clonée. Désormais, il ne nous reste plus qu'à franchir 3 autres étapes.

La première concerne la création de l'image de conteneur. Pour le faire, nous utilisons la ligne de commande qui suit :

```
1 docker build -t nodejs-backend-application:0.1.0
```

Une fois l'image du conteneur créée, nous allons exécuter ce dernier avec la ligne de commande suivante :

```
1 # docker run -p <host_port>:<container_port> nodejs-backend-application:0.1.0
2 docker run -p 3001:3001 nodejs-backend-application:0.1.0
```

Voilà, nous sommes à la fin. C'est le moment de vérifier si nous parvenons à accéder à notre application depuis le navigateur. Pour cela, nous utilisons la commande de testing « curl » comme spécifiée dans la ligne suivante :


```
1 curl http://127.0.0.1:3001
```

Nous venons de finaliser la conteneurisation de notre application back-end.

Exercice p. 15 Solution n°5


Question 1

Quand les développeurs utilisent-ils les microservices ?

- ☐ Quand ils travaillent avec la nanotechnologie éphémère
- ☐ Quand ils veulent écrire des applications pour téléphone portable qui s'exécutent rapidement
- ☒ Lorsqu'ils doivent créer de grandes applications de niveau entreprise qui sont sujettes à des modifications fréquentes
-  L'une des choses pour lesquelles l'architecture de microservices est certainement utile, c'est de faciliter l'évolutivité des applications. Ceci est permis grâce à la décentralisation des processus et à l'autonomie des composants.


Question 2

L'architecture de microservice permet efficacement de palier aux déficiences du pattern monolithique.

- ☒ Vrai
- ☐ Faux
-  L'architecture de microservice est une alternative efficace pour pallier les déficiences du pattern monolithique. Cependant, elle nécessite plus de travail et surtout une expertise sur la question. Lorsque sa mise en place est bâclée, le résultat peut être catastrophique.


Question 3

Les microservices ne sont destinés qu'aux petites entreprises.

- ☐ Vrai
- ☒ Faux
-  Depuis son apparition l'approche par microservices est largement adoptée par les grandes entreprises qui trouvent en elle la solution pour opérer des mises à l'échelle efficaces et efficientes. C'est le cas par exemple avec Amazon ou encore Netflix.


Question 4

Qu'est-ce qu'une API en architecture de microservices ?

- ☐ Un service qui expose les fonctionnalités d'une architecture à un minimum de 2 applications
- ☐ Un sous-composant d'une application qui fonctionne indépendamment
- ☒ Une interface qui fonctionne entre 2 applications indépendantes et leur permet de communiquer
-  Les API sont utilisées dans l'architecture de microservices pour permettre à 2 services de s'échanger des données. Elles font partie intégrante des composants de chaque service.

Question 5

La SOA est définie comme un composant de service indépendant qui permet à 2 applications de communiquer.

- ☐ Vrai
- ☒ Faux
-  Il s'agit d'un design qui organise les composants d'une application dans une architecture décentralisée avec des nœuds autonomes.