DevOps

Examen Final – Mai 2017

UFR IM2AG – M1 INFO – 2016-2017

Les consignes sont les suivantes :

- Durée de l'examen : **2 heures**
- Nombre de pages dans l'énoncé : **4**
- Documents autorisés : **Tous documents**
- Le barème est donné à titre indicatif.

1. L'approche DevOps (5 points)

1.1 L'approche DevOps séduit de plus en plus d'entreprises dans le monde du logiciel. Donnez 4 intérêts majeurs de l'approche DevOps qui peuvent selon vous expliquer son attrait. Justifiez votre réponse.

1.2 Les architectures logicielles sont de plus en plus souvent construites selon le modèle des microservices. Plutôt que de concevoir de grosses applications monolithiques, celles-ci sont construites comme un ensemble de services utilisant un protocole simple pour communiquer. Donnez 3 arguments en faveur de l'adoption d'une architecture microservices dans le cadre de la mise en place d'une approche DevOps. Justifiez vos réponses en quelques mots.

1.3 À quelques heures d'une réunion importante avec des clients, un collègue se rend compte d'un bogue important dans le logiciel qu'il doit leur présenter. Il a réussi à identifier le bogue en question : il est lié à une des bibliothèques qu'il utilise (`libdosomestuff`).

Pour le moment il utilise la version 2.4.7 de la bibliothèque `libdosomestuff`, et il a vu sur Internet qu'il en existe deux nouvelles versions : une version 2.4.8 et une version 3.0.0.

(a) Il ne comprend pas bien le principe des numéros de version. je le cite : "Mais pourquoi ça ne passe pas de 2.4.8 à 2.4.9 ? !" Que pouvez-vous répondre à sa question ?

(b) En consultant les informations fournies sur ces nouvelles versions, il constate que la version 3.0.0 apporte des améliorations majeures en terme de performances, ce qui l'intéresse beaucoup. Quelle version lui conseillez vous d'intégrer en vue de sa réunion ? Justifiez.

2. Les builders (5 points)

2.1 Un Makefile La figure 1 décrit un exemple de fichier `Makefile`. Expliquez ce qu'il se passe lors d'un appel à :

(a) `make clean`

(b) `make install`

(c) `make`

Pour chacun de ces cas, vous donnerez une réponse détaillée, c'est-à-dire que vous expliciterez :

- l'ensemble des règles exécutées lors de l'exécution du Makefile (On supposera toujours que les dépendances ont été modifiées depuis la dernière exécution d'une règle) ;
- le contenu de chaque variable utilisée dans une règle ;
- tout autre point qu'il vous semble important de mentionner.

Cependant si une règle ou une variable a déjà été discutée dans une réponse précédente, il n'est pas nécessaire de détailler à nouveau.

```
1  CC=gcc
2  CFLAGS=
3  LDFLAGS= -lm
4
5  MAIN=myprog.c
6  SOURCES=$(wildcard src/*.c)
7
8  TEST_PROG=run_test.run
9  TARGET=$(MAIN:.c=.run)
10
11 OBJECTS=$(patsubst %.c, %.o, $(SOURCES))
12
13 all: install test
14
15 install: $(TARGET)
16         mv $(TARGET) /usr/bin
17
18 test: $(TEST_PROG)
19         @echo "Running␣tests␣..."
20         ./$(TEST_PROG)
21
22 %.run: %.o $(OBJECTS)
23         $(CC) $^ -o $@ $(LDFLAGS)
24
25 %.o: %.c
26         $(CC) -c $< $(CFLAGS)
27
28 clean:
29         rm -rf $(TEST_PROG) $(TARGET) $(OBJECTS) *.o
```

FIGURE 1 – Un Makefile

2.2 Maven Dans le cadre d'un projet de développement logiciel constitué de plusieurs modules, une bonne manière de structurer son projet est d'avoir un répertoire par module. Si Maven est utilisé dans le cadre d'un tel projet, il est alors courant d'avoir un fichier `pom.xml` par module et un fichier `pom.xml` à la racine du projet.

Citez deux mécanismes de Maven qui peuvent être particulièrement utiles dans ce contexte. Pour chacun d'entre eux, expliquez brièvement son fonctionnement et son utilité.

3. Docker (5 points)

3.1 Docker utilise le concept d'*image* et de *conteneur*. Définissez ces deux concepts et explicitez la relation entre les deux.

3.2 Vous avez pu constater en utilisant Docker que la taille des images est généralement petite. Expliquez pourquoi dans les deux cas suivants :

(a) L'image Docker `ubuntu:latest`, c'est à dire l'image correspondant à la dernière version stable de la distribution Ubuntu, ne fait que 130 MB.

- Remarque : la première ligne du fichier `Dockerfile` définissant cette image est "`FROM scratch`", `scratch` correspondant à une image vide.

(b) L'image Docker de `Memcached`, qui est un service concurrent de Redis, ne fait que 87 MB.

- Remarque : la première ligne du fichier `Dockerfile` définissant cette image est "`FROM ubuntu:latest`".

3.3 Nous avons vu lors du TP sur Docker qu'il existe un moyen simple pour accéder en écriture et en lecture à un répertoire du système d'exploitation hôte depuis un conteneur Docker. Ainsi les modifications appliquées à un fichier de ce répertoire lors de l'exécution du conteneur sont encore visibles après la fin du conteneur.

(a) Quel est le mécanisme qui permet de faire ceci ?

(b) Quelle commande permet de l'utiliser ?

3.4 A propos de la construction d'images :

(a) Écrire le fichier `Dockerfile` permettant de créer l'image `devopsexam` répondant aux contraintes suivantes :

- L'image doit être basé sur l'image `debian:latest`
- Le répertoire `/tmp` doit être défini comme répertoire de travail
- La commande `rev` doit être installée (installation avec `apt-get install -y rev`). Cette commande inverse toutes les lettres des chaînes de caractères qui lui sont données en entrée.
- Le fichier `my_palindrome.txt` doit être copié depuis l'hôte vers le répertoire de travail
- La commande par défaut à associer à l'image est :

```
rev < my_palindrome.txt
```

(b) A quoi faut-il fait attention au sujet du fichier `my_palindrome.txt` pour que la création de l'image se passe correctement ?

4. Tests (5 points)

4.1 Nous avons évoqué en cours les *principles of testing* qui sont un ensemble de points à avoir à l'esprit lors du test d'un logiciel. Le premier d'entre eux dit que "*Les tests mettent en évidence des problèmes*". Expliquez le problème que souligne ce principe. Donnez un exemple pour illustrer votre propos.

4.2 Dans cet exercice nous vous demandons d'écrire une classe de tests en Java en utilisant JUnit4.

Nous considérons la classe `SortedList` qui met en œuvre une liste triée d'entiers. Nous nous intéressons plus particulièrement à la méthode `insert(Integer A)` qui insère l'entier A dans la liste. Cette méthode a la spécification suivante :

- La méthode `insert(Integer A)` insère l'entier A dans la liste de telle manière à ce que la liste reste triée.
- Si l'entier A est déjà présent dans la liste, une exception de type `AlreadyInsertedException` est levée.
- Si l'entier A a une valeur négative, une exception de type `InvalidArgException` est levée.

Donnez le code Java de la classe `SortedListTest` en y incluant tous les tests qui vous semblent nécessaires pour tester correctement la méthode `insert(Integer A)`.

Remarque 1 : La qualité de vos tests sera évaluée en fonction du taux de couverture mais aussi en fonction de la qualité de votre code du point vue de l'utilisation de JUnit.

Remarque 2 : Ne vous souciez pas des packages à importer.

Remarque 3 : Vous pouvez supposer que d'autres méthodes ont été définies pour la classe `SortedList` si vous en avez besoin pour vos tests. Donnez leur un nom suffisamment précis pour que votre code reste lisible.

Remarque 4 : Pour tester le contenu d'une `SortedList`, pensez à utiliser la méthode `equals(Object o)` pour comparer son contenu avec celui d'une autre liste. La méthode `equals` définie dans l'interface `List` renvoie `vrai` si et seulement si, l'objet considéré est une liste, les deux listes ont la même taille, et tous les éléments des listes sont égaux deux à deux.