Cahier de TP « Design Principles »

Outils utilisé lors de :

- Bonne connexion Internet
- Système d'exploitation recommandé : Linux ou Windows 10
- JDK8 et Git
- Docker
- IDE Recommandé : Spring Tools Studio

TP1 : Design Patterns

Pré-requis :

- JDK 8
- IDE recommandé : STS 4

Dans un premier temps, il est demandé aux stagiaires d'identifier les patterns décrits par les 3 problème suivants.

Ensuite, compléter le code fourni

Description de 3 problèmes :

Problème n°1 :

Gérer la réutilisation d'objets lorsqu'un type d'objet est coûteux à instancier ou lorsque le nombre d'instance est limité

Problème n°2 :

Permettre à une implémentation existante d'être accessible via un interface sans que cette implémentation implémente l'interface

Problème n°3 :

Permettre à un objet d'envoyer une commande sans connaître quels objets la recevront. Chaque objet dans la chaîne traite la commande et la passe au suivant

Optionnel

Problème n° 4 :

Minimiser l'usage mémoire en partageant des objets représentatif de données utilisés à de nombreux endroits.

TP2 : AOP avec AspectJ

Pré-requis :

- JDK 8
- IDE recommandé : STS 4

Nous utilisons le framework *SpringBoot* et *AspectJ* pour illustrer l'AOP.

Reprendre les sources fournis (projet Maven)

Un exemple complet d'implémentation d'un aspect via une annotation est fourni avec les classes

org.formation.aspectj.Secured et *org.formation.aspectj.SecuredAspect*

Comprendre l'exemple et s'en inspirer pour implémenter un aspect qui lorsqu'il sera appliqué affichera le temps d'exécution d'une méthode.

Exécuter le test fourni pour visualiser les effets des annotations

TP3 : Programmation réactive

Pré-requis :

- Navigateur Chrome ou Firefox

Aller sur *stackblitz.com* et créer un projet RxJS/TypeScript.

Visualiser le projet exemple

Reprendre les sources fournis et les comprendre

Modifier le fichier index.html en ajoutant un champ de saisie.

Créer un observable sur l'événement ***keypress***.

Sur chaque caractère saisi :

- Filtrer afin que d'accepter les lettres de l'alphabet
- Passer en Majuscule
- Ignorer la saisie de 2 lettres identiques à la suite
- Afficher la chaîne complète saisie

TP4 : Programmation cliente

Pré-requis :

- JDK 8
- IDE recommandé : STS 4

Récupérer les jars fournis et les exécuter :

- *java -jar annuaire.jar*
- *java -jar notification-service.jar*
- *java -jar fakeSMTP2.0.jar*

Les programmes démarrent respectivement un service d'annuaire Eureka écoutant sur le port 1111, un service applicatif de notification qui s'enregistre dans l'annuaire et qui communique avec un faux serveur de mail sur le port 2525

Cela nécessite que les adresses IP ***annuaire*** et ***fake-smtp*** soit résolues en *localhost*

Récupérer les sources fournis dans votre IDE. L'objectif est de compléter la méthode test en :

- Effectuant un lookup vers l'annuaire pour localiser le service de notification (Utiliser l'objet ***discoveryClient***)
- Effectuer un appel REST en utilisant l'objet *RestTemplate* fournit par Spring

Démarrer un 2ème service de notification sur un autre port de la façon suivante :

- *java -jar notification-service.jar --server.port=9091*

Qu'en est il de la répartition de charge ?

TP5 : Serveur : Pool de threads vs Non-Blocking Event-Loop

Pré-requis :

- JDK 8
- IDE recommandé : STS 4
- JMeter (sur Linux)

L'objectif de cette partie est de comparer les performances et le scaling du modèle bloquant vis à vis du modèle non bloquant

Récupérer les 2 projets Maven Web fourni :

- 5_imperative-controller : Modèle bloquant
- 5_reactive-controller : Modèle non-bloquant

Démarrer JMeter avec *$JMETER_HOME/bin/jmeter &*

Utiliser le script JMeter fourni, *File → Open* , choisir *LoadTest.jmx*

Effectuer des tirs avec les paramètres suivants : NB_USERS=150, PAUSE=1000

A la fin du résultat, notez :

- Le temps d'exécution total du test
- Le débit
- Le Max

Effectuez plusieurs tirs en augmentant le nombre d'utilisateurs par pallier de 100

Lors des tirs observer les threads créés (via *jvisulavm* par exemple ou via la commande en ligne *jstack <pid> | grep -c http-nio*)

Attention dans un environnement Windows :
http://twit88.com/blog/2008/07/28/jmeter-exception-javanetbindexception-address-already-in-use-connect/

Alternative docker :

Dans le répertoire docker visualisez les fichiers docker-compose

Exécuter `docker-compose up -d`

Attacher sur le conteneur jmeter via :

```
docker exec -it docker_jmeter_1 /bin/sh
```

Exécuter les tests avec des commandes :

```
jmeter -n -t /LoadTest.jmx -l /LoadTest.jtl -JSERVER=imperative -JNB_USERS=1000
```

Architectures distribuées

TP6 : Services Backend et OpenAPI

Pré-requis :

- JDK 8
- IDE recommandé : STS 4
- Postgres DB

Objectifs de l'atelier : Passer en revue une application back-end avec découpage classique en package offrant une API Rest documentée.

Le projet est un nouveau projet Maven (Spring Boot) composé de 3 packages:

- Package model : Modèle de données persistantes d'une base de produits
- Package service : Un service d'importation de produits
- Package dto : Classes de transfert de données
- Package controller : Couche web

Travail demandé :
Importer le projet Maven fourni.

- Annoter la couche contrôleur afin de fournir une API Rest la plus intuitive possible.
  Les annotations à disposition dans SpringBoot sont :
  - *@RequestMapping*
  - Sa déclinaison en *@GetMapping*, *@PostMapping*, *@PutMapping*, ….
    Voir https://www.baeldung.com/spring-new-requestmapping-shortcuts
- Vérifier la génération via Swagger de la documentation
- Affiner les annotations swagger afin parfaire l'interface de *swagger-ui*

Pour construire et exécuter l'application :
./mvnw spring-boot:run
OU
./mvnw package
java jar target/product-service.jar

TP7. Streaming temps réel avec Spring Boot Stream et Kafka

Pré-requis :

- JDK 8
- IDE recommandé : STS 4
- Docker

- Téléchargement images docker Kafka/Zookeeper

Objectifs : Montrer l'évolutivité d'une architecture de type Stream

Un flux de données de positons sont envoyées sur une URL, chaque donnée comporte un ID identifiant l'objet localisé. Un premier micro-service se contente de récupérer ce flux et de le stocker dans un topic Kafka.

A posteriori, on décide de rajouter un autre micro-services dont la seule responsabilité est de calculer pour chaque objet sa position moyenne pour chaque tranche de minute. Le micro-service récupérera les anciens évènements

1. Démarrage Kafka

Un fichier docker-compose permettant de démarrer kafka et zookeeper est fourni.

Il nécessite la déclaration de la machine kafka vers localhost (/etc/hosts)

2. Première application SpringBoot-CloudStream

Importer le projet Maven ***14_cloud-stream*** dans Spring Tools Suite et démarrer l'application. Vérifier la bonne connexion à Kafka.

Démarrer JMeter et ouvrir le fichier JMX fourni.

Exécuter le script et vérifier le bon envoi de message dans le topic Kafka

3. Mise en place d'une seconde application

Importer le projet Maven ***14_cloud-stream-average*** dans Spring Tools Suite, comprendre et compléter le code (Classes : *AverageStream*, *AverageService* et *PositionsListener*)

Démarrer l'application *SpringBoot* est vérifier le traitement des messages à posteriori

TP8 : Architecture MicroServices

Pré-requis :

- JDK 8
- IDE recommandé : STS 4 avec Lombok

Dans cet atelier, nous allons bâtir une architecture micro-service composé :

- De trois micros-services métier :
  - *product-service avec BD relationelle*
  - *order-service avec BD relationelle*

    - *delivery-service avec Support Kafka*
- D'un micro-service de bas-niveau :
    - *notification-service* avec serveur *smtp*
- De micro-services d'infrastructure :
    - Configuration centralisée
    - Discovery : Eureka
    - Proxy

Pour faciliter le développement, nous mettons à jour notre fichier host avec que les noms suivants pointent sur localhost :
config, annuaire, fake-smtp

Mise en place :
Dézipper l'archive fournie et faire pointer Spring Tools Suite vers ce nouveau workspace

1ère partie démarrage des micro-servics d'infrastructure et vérification des inscriptions dans l'annuaire
Importer les projets Maven suivant :

- *config*
- *product-service*
- *order-service*

Démarrer dans l'ordre :

1. Démarrer le service de config
   Visualiser les URLs
   http://config:8888/application/default
   http://config:8888/product-service/default
   http://config:8888/order-service/default
2. Démarrer le service de discovery : ***java -Xmx128m annuaire.jar***
   Consulter :
   http://annuaire:1111
3. Démarrer *product-service* à partir de votre IDE et vérifier son inscription dans l'annuaire
4. Faire de même avec *order-service*

Visualiser les interfaces REST des 2 micro-services via leur interface Swagger

2ème partie : Equilibrage de charge avec un nouveau micro-service
Importer le projet *notification-service* et le démarrer, vérifier son inscription dans l'annuaire.
Démarrer le faux serveur smtp (*fake-SMTP.jar*), déclarer le host *fake-smtp* comme pointant sur *localhost* et tester l'envoi d'un mail via la commande *curl* fournie dans le projet notification-service.

On veut désormais envoyer un email lorsqu'un Client crée une commande. On s'appuie sur le nouveau micro-service notification-service.
De plus si plusieurs instances de *notification-service* sont démarrés, nous voulons équilibrer la charge

Ajouter la classe de configuration suivante :

```
@Configuration
public class ClientConfiguration {
```

```
	@Autowired
	RestTemplateBuilder builder;

	@Bean
	@LoadBalanced
	RestTemplate restTemplate() {
		return builder.build();
	}
}
```

Ajouter également la propriété suivante :
```
spring.cloud.loadbalancer.ribbon.enabled: false
```

Ajouter un nouveau bean *@Service* proposant une méthode utilisant la bean *RestTemplate* et effectuant la requête nécessaire pour envoyer un mail via le service *notification-service*

Tester la répartition de charge via le script JMeter *Load.jmx* fourni

Que se passe t il si aucun service notification-service n'est démarré ?

3ème partie : Design pattern Circuit-breaker

Nous voulons désormais répondre à une création de commande même si aucun service notification-service n'est disponible.
Nous appliquons le pattern Circuit-breaker en utilisant une librairie spécialisée : Resilience4J
Ajouter la dépendance :
```
<dependency>
	<groupId>org.springframework.cloud</groupId>
	<artifactId>spring-cloud-starter-circuitbreaker-resilience4j</artifactId>
</dependency>
```

Inspirer vous de https://spring.io/projects/spring-cloud-circuitbreaker pour implémenter une méthode de fall-back

Tester le fall-back

4ème partie : Mise en place d'un proxy

Importer le projet proxy, vérifier sa configuration afin qu'il permette l'accès aux micro-services métier ***product-service*** et ***order-service***

4ème partie : Docker

Mise en place d'un docker-compose pour lancer toute la stack

5ème partie (Optionnel) : Saga

La création d'une commande consiste en une saga de transaction locales :

- Insertion de la commande dans la BD
- Vérification de la disponibilité des produits dans la base produits

La vérification de la disponibilité du produit peut conduire à rejeter la commande.

Implémenter sous forme d'une SAGA cette séquence de transactions

Réimporter les projets ***product-service*** et ***order-service*** fournis (le *pom.xml* est compatible avec Kafka) pour effectuer cet exercice.

Regarder le service *product-service* et en particulier la méthode *checkAvailability()*
Compléter le service *order-service*

TP9 : Interface Utilisateur Angular

Pré-requis :

- Node js
- Angular CLI

Description du projet
La page Angular que nous allons créer consiste à affihcer les commandes du Client avec l'id 1

Travail demandé

- Installation Angular CLI et création de projet
  ```
  npm install -g @angular/cli
  ng –version
  ng new angular
  cd angular
  ng serve
  ```
- Créer le composant d'affichage de la liste
  ```
  ng generate component order-list
  ```

Compléter le code fourni

TP10 : Gestion des flux avec ElasticStack

Pré-requis :

- JDK 8
- IDE recommandé : STS 4
- ElasticStack
- Images docker Kafka/Zookeeper
- JMeter

Objectifs de l'atelier : S'initier à la suite ElasticStack et à chacun de ses composants : Ingestion de données, Indexation et Visualisation en temps-réel.

Travail préliminaire :

- Démarrer la stack du Tps précédents :
  - Kafka/Zookeper via docker-compose
  - Les 2 micro-services traitant les messages. On pourra utiliser le jar généré par Maven et démarrer ce programme Java avec seulement 128m
- Solliciter les micro-services via le nouveau script JMeter fourni

Elastic Stack :

-
- Installer *ElasticSearch* et le démarrer, vérifier le bon lancement en accédant à

http://localhost:9200

- Installer *Kibana* et démarrer le serveur, accéder à http://localhost:5601
  Accéder à la Dev Console et exécuter la requête :

```
PUT /positions
{ "mappings": { "doc": {
    "properties": {
      "location": { "type": "geo_point" }
     }
} } }
```

Cette requête créée un index nommé *positons* contenant un champ *location* de type *geo_point*

- Installer *logstash* et récupérer le fichier de configuration fourni. Éditer le et comprendre les différentes étapes de la pipeline
  Tester le avec la commande :
  bin/logstash -t -f ../logstash_pipeline.conf
  Démarrer l'ingestion avec
  bin/logstash -r -f ../logstash_pipeline.conf
- Pendant l'ingestion, accéder à kibana et configurer un index pattern vers l'index *positions*
- Aller dans le menu *Discover* et observer l'arrivée des données en temps réel
- Aller dans *Visualize* et créer une visualisation de type « Coordinate Maps »

TP11 : Réseaux de neurones avec TensorFlow

Pré-requis : :

- Python 2 ou 3
- pip

Objectifs de l'atelier : Appréhender la méthodologie de mise au point de système ML, avoir un apperçu du framework TensorFlow.

TensorFlow : `pip install tensorflow`

Librairies gestion d'images :

```
apt-get install -y python-tk
pip install scikit-image
```

Objectif du système : Classifier des images de panneaux de signalisation routière.

Etape 1 : appréhender les données

Nous disposons de 2 ensembles étiquetés. Un pour l'apprentissage, un pour la validation.

Dézipper et parcourir ces 2 ensembles

Utilisez le script Python fourni *distribution.py* pour bien appréhender les données d'apprentissage

Etape 2 : Préparer les données

Afin de pouvoir alimenter le réseau de neurones avec le même nombre de caractéristique d'entrée, il

est nécessaire de redimensionner les images exemples à la même dimension. D'autre part, la couleur n'est pas importante pour les problèmes de classification, nos les transformons en échelle de gris.

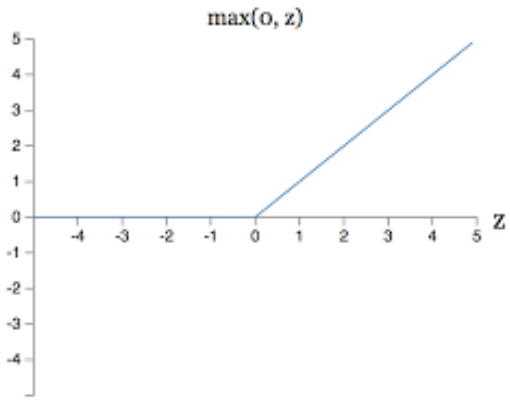
Utiliser le script fourni *plot-random-transform.py* pour voir les effets des transformations

Etape 3 : Créer un modèle, l'entraîner, le valider

La troisième étape consiste à construire un modèle en utilisant l'API *TensorFlow*, l'entraîner avec l'ensemble d'apprentissage est de vérifier son acuité avec l'ensemble de validation.

Le modèle est constitué de 3 couches :

- Une première couche qui met à plat les matrices de pixels en une entrée de 28x28 nœuds
- Une seconde couche constituée de 128 nœuds utilisant une fonction d'activation *RLU (Rectified Linear Unit)* qui introduit de la non-linéarité



- Une dernière couche avec 62 nœuds de sortie correspondant aux différentes étiquettes utilisant la fonction d'activation *softmax*. Ainsi chaque nœud de sortie contient un score correspondant à la probabilité de l'étiquette

Lors de la compilation avec Tera, on indique 3 autres caractéristiques relatif à l'apprentissage :

- L'optimiser : La méthode utilisée pour la mise à jour des paramètres du modèle pour minimiser la perte
- La perte : L'algorithme utilisé pour le calcul de la perte. La valeur à minimiser
- Metric : Un métrique pour surveiller l'apprentissage. Ici, le pourcentage d'image qui sont correctement évaluées

Ensuite, le modèle est entraîné puis évalué :

- La perte et le pourcentage de bonnes prédictions sont affichées pour l'ensemble de validation
- 25 exemples pris au hasard sont affichées et on indique si la prédiction est bonne ou pas.

Comprendre le script fourni *model.py*, l’exécuter et éventuellement jouer avec certains paramètres :

- Nombre d’itérations
- Algorithme d’optimisation
- Ajout d’une couche
- ...