Cahier de TP « Design Principles »

Pré-requis :

- Bonne connexion Internet
- Système d'exploitation recommandé : Linux
- JDK 8
- PostgreSQL
- Eclipse Mars JavaEE+ JbossTools
- Spring Tools Suite 3.8.1+
- Serveur Wildfly 10.0
- Docker et Docker Compose
- Git
- Npm et Node.js
- SoapUI
- Bon éditeur

TP1 : Design Patterns

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

Ensuite, le code source est récupéré et doit être complété. Le travail s'effectue en binôme pour favoriser les discussions.

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

TP2 : Architecture n-tiers

Objectifs de l'atelier : Passer en revue les apports d'un serveur applicatif, revoir l'architecture logicielle classique pour le développement d'applications Web.
En particulier :
- Le format EAR
- Maven
- JPA
- Technologies EJB3 stateless

- CDI
- Pattern MVC de la couche de présentation
- Comportement Ajax et Encapsulation du javascript dans les balises JSF

Description du service en ligne
L'application consiste en une base documentaire en ligne. Des utilisateurs peuvent s'enregistrer en ligne sur le service. Ils peuvent alors uploader des documents sur un serveur.

Les sources fournies sont un projet Maven composé de 4 modules :
- **Module modèle** : Contenant les classes du modèle. (Classes User et Document)
- **Module ejb** : Implémentant la couche service sous forme d'EJBs stateless, cette couche utilise JPA. Les services métiers y sont spécifiés via l'interface *UserDocumentServiceIF*
- **Module web** : Implémentation JSF de la couche présentation utilisant jquery et bootstrap.css. La cinématique de l'application est gérée par les classes contrôleurs. Les vues JSF effectuent les binding entre les classes contrôleurs et la vue
- **Module ear** : Nécessaire pour le packaging en ear

Travail demandé :
1. Créer une base de données PostgreSQL
2. Démarrer le serveur Wildfly
3. Déployer le driver Postgresql fourni en le copiant dans le répertoire *standalone/deployements* de Wildfly
4. Accéder à la console d'administration (localhost:9990) et créer une source de données avec le nom JNDI suivant : « java:jboss/datasources/PostgresqlDS »
5. Importer le projet Maven *2_UserDocuments3Tiers* dans Eclipse
6. Déclarer le serveur Wildfly 10.x et associer le projet au serveur
7. Effectuer un premier déploiement et modifier la configuration afin qu'Hibernate créée automatiquement les tables à partir du modèle objet.

Comprendre et compléter le code fourni (EJB, Contrôleurs et vue JSF) :
En particulier, les fichiers suivants sont à compléter :
- Module EJB : *UserDocumentService.java*
- Module WEB : *LoginController.java*, *DocumentsController.java*, *login.xhtml*, *documents.xhtml*

TP3 : Services Web

Objectifs de l'atelier : Comprendre les implications d'offrir ses services métiers via le web. Appréhender les technologies SOAP et REST

Le projet est un nouveau projet Maven composé de 3 modules :
- Module modèle : Identique au précédent projet
- Module EJB : Identique au précédent projet
- Module parent : Module parent incluant les 2 sous modules et les packageant sous forme de war

Travail demandé :
- Mise en place projet : Importer le projet Maven fourni, l'associer au serveur Wildfly
- API Soap : Ajouter les annotations afin que les méthodes de l'EJB soient accessibles via SOAP, Déployer, accéder au WSDL, Effectuer une requête SOAP avec SoapUI par exemple
- API REST :

- Ajouter les annotations jax-rs à la classe *UserDocumentServiceRest.java*. Accéder à l'interface REST
- Générer la documentation de l'API via Swagger

Voir https://github.com/swagger-api/swagger-core/wiki/Swagger-Core-JAX-RS-Project-Setup-1.5.X pour la mise en place et https://github.com/swagger-api/swagger-core/wiki/Annotations pour les annotations disponibles

TP4 : Gestion des flux avec ElasticStack

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 demandé :

- Installer *ElasticSearch* et le démarrer, vérifier le bon lancement en accédant à *http://localhost:9200*
- Installer *Kibana* et démarrer le serveur, accèder à l'URL affichée sur la console
- Installer *logstash* et récupérer le fichier de configuration fourni. Éditer le et l'adapter à votre environnement.
  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 et configurer Kibana
- L'application précédente peut être accéder simultanément afin d'observer l'aspect « temps-réel »

TP5 : Architecture MicroServices

Dans cet atelier, nous allons décomposer l'architecture monolithique (un seul war ou ear) des précédents TPs en une architecture micro-service. Dans un premier temps, nous allons décomposer notre projet en 2 micro-services offrant une interface RESTFul. Ces 2 micro-services s'inscriront au démarrage dans un annuaire de service afin de pouvoir être découvert automatiquement.
Les technologies utilisées sont cette fois-ci basées sur les outils Spring. En particulier :

- *Spring-Boot*, *Spring-Cloud* : Pour les aspects micro-services
- *Spring-REST*, *Spring MVC* pour les couches d'accès REST
- *Spring-core* pour les annotations et injection de dépendances
- *Spring-JPA*, *Spring-Repository* pour le tiers de persistance
- *Spring-Tool-Suite* pour l'IDE

Description du projet
Le projet est un projet Maven composé de 3 sous-modules :

- Module *annuaire* fournit l'application Spring-Boot permettant de démarrer un serveur Eureka qui enregistre les services. Les sources fournies sont complets
- Module *userService* fournit l'application SpringBoot qui implémente le service de gestion des utilisateurs. Les classes principales du service sont :
  - *MemberRepository* : Interface d'accès aux services de stockage des utilisateurs
  - *MembersController* : Contrôleur Rest fournissant l'interface REST au services

  Ces classes doivent être complétées. Les autres classes sont complètes.

- Module *documentService*. Il est construit de la même façon que *userService*

Travail demandé
Installer Spring Tool-suite
Importer le projet Maven fourni
Visualiser les annotations utilisées et les configuration *.yml
Compléter les sources
Vérifier l'inscription des services dans l'annuaire (http://localhost:1111)
Tester les interfaces REST des 2 micro-services

TP6 : Frameworks clients, l'exemple d'Angular 1

Cet atelier poursuit l'atelier précédent. L'architecture micro-service est augmenté d'un nouveau micro-service d'infrastructure. Un service de proxy permettant d'offrir une interface unique pour l'accès à tous les services. L'agrégation des services est effectué côté client et utilise le framework AngularJS

Description du projet
Le projet est le projet Maven précédent augmenté du sous-module proxy. Le sous-module utilise Zuul de SpringBoot et fournit également l'interface Angular.js. L'interface est composée de 2 vues :

- *partials/login.html* : formulaire de login permettant d'appeler le service ***userService*** via le proxy pour l'authentification
- *partials/documents.html* : la liste des documents de l'utilisateur loggé. La liste est obtenue en appelant le service ***documentService*** via le proxy

Travail demandé
Activer le sous-module proxy en dé-commentant les lignes correspondantes dans le *pom.xml* du projet précédent
Visualiser la configuration du proxy (bootstrap.yml)
Compléter *service.js* et *controller.js* pour effectuer les bons appels de service

TP6B : Angular2

Description des projets
2 projets sont utilisés pour cet atelier :

- ***proxy*** : Un projet Maven dans l'environnement Spring Tools Suite
- ***angular2*** : Un projet *npm* qui implémente l'interface avec Angular2
  Ce projet est composé de 3 composants
  - app : Le composant parent englobant les 2 autres
  - login : Le formulaire permettant de s'authentifier
  - documents : La liste des documents d'un utilisateur donné

  Il contient également un service qui effectue les appels backend vers le service proxy fourni par SpringBoot

Travail demandé

- Micro-service SpringBoot

Activer le sous-module proxy en dé-commentant les lignes correspondantes dans le *pom.xml* du projet précédent

Visualiser la configuration du proxy (*bootstrap.yml*)
Démarrer le proxy et essayer d'atteindre les service back-end *UserService* et *DocumentService*

- Angular2

Commencer par installer les dépendances : *npm install*
Visualiser les fichiers TypeScript fournis, certains sont à compléter
Démarrer l'application avec *npm start*
Exécuter l'application avec Chrome et les outils de debugging

TP7 : Containerisation avec Docker

Dans cet atelier, nous allons construire et exécuter les images Docker des précédents micro-services. La containerisation des services va nous permettre de facilement démarrer plusieurs répliques du même service et donc d'introduire de la haute-disponibilité dans notre architecture. La répartition des requêtes clientes s'effectuera via SpringBoot et Ribbon.
Docker compose et démarrage des différents containers

Description du projet :
Le précédent Maven contenait des fichiers Docker file définissant les images docker pour les différents services. Il contient également un fichier *docker-compose.yml* qui configure docker-compose pour démarrer l'ensemble de l'architecture.

Travail demandé :
Compléter les fichiers *Dockerfile* fournis, construire les images et les exécuter manuellement
Identifier dans le projet proxy la configuration Ribbon et Hystrix
Compléter le fichier *docker-compose.yml* afin de lancer plusieurs répliques des services back-end
Tester la haute disponibilité de l'application

TP8 : Intégration continue

Dans cette atelier, nous mettons en place un plusieurs jobs Jenkins formant une pipeline qui simule une pipeline de livraison continue :

- Le premier job construit l'application, effectue les tests unitaires, construit les images docker et démarre des serveurs d'intégration. Le job est déclenché par un commit dans un repository git local
- Le second job démarre si le premier a réussi. Il effectue des tests de performance
- Le troisième job est le job qui déploie. Il est déclenché manuellement

Les plugins Jenkins utilisés sont Git, Pipeline et Performance Plugin

Description du projet
Le projet est constitué :

- D'une image docker correspondant à une installation Jenkins avec les plugins Git
- D'un repository Git en local (1 repository par stagiaire)

Travail demandé :
Clone de repository Github dans un repository local https://github.com/dthibau/design-principles-TP7.git

Construction, puis exécution de l'image Docker Jenkins :

docker build -t jenkins_docker_plugin .

docker run -d -v /var/run/docker.sock:/var/run/docker.sock -v $(which docker):/usr/bin/docker -v /your-work/jenkins_home:/var/jenkins_home -v /your-git-repo:/user/local/userDocument -p 8080:8080 jenkins_docker_plugin

Tester le bon fonctionnement de Docker dans Jenkins (dans Docker) en créant un job shell exécutant docker *sudo docker run hello-world*

Création du premier job Jenkins déclenché par un commit sur le repository local

- Effectuant le build Maven *mvn package*
- Construisant les images Docker pour les différents services et les poussant dans le repository local
- Démarrant les différentes images Docker avec docker-compose

Tester

Configuration du plugin pipeline (Ajouter une vue)

Création d 'un second job LoadTest déclenché à la suite du premier

- Effectuant le build Maven *mvn verify*
- 2 Post-build-actions
  - Publier les résultats de performance
  - Ajouter une étape manuelle pour démarrer le 3ème job « Deploy »

Le job « Deploy » simulera un déploiement

TP8B : Intégration continue (BlueOcean et JenkinsFile)

Objectifs de l'atelier : Adopter une approche DevOps en intégrant le fichier de configuration de la pipeline dans les sources projets.

Travail demandé :
Installer le plugin BlueOcan
S'inspirer du fichier JenkinsFile fourni pour mettre en place la pipeline pour nos micro-services

TP9 : Microservices avec Jhipster

Dans cette atelier, nous générons une application avec une architecture micro-service avec JHipster afin de comparer les résultats obtenus avec un générateur de code avec le réssultat des Tps précédents.

Pré-requis

- Installation JHipster
- Téléchargement JHipster registry

Travail demandé :

1. Démarrage de JHipster registry

2. Création d’une application microservice avec *JHipster*, nommée *micro_doc*, création d’une entité ***doc***, démarrage du microservice. Vérification de l’inscription dans *JHipster Registry*
3. Création d’une application microservice avec *JHipster*, nommée *micro_membre*, création d’une entité ***membre*** en relation avec ***user*** (*JHipster*) et ***doc***, démarrage du microservice. Vérification de l’inscription dans *JHipster Registry*
4. *Création d’une application gateway, création des 2 entités membre et doc*