Documentation Phase 2

Projet Architecture

Groupe 3 : Groupe

Encadrant : M.PETIT

Younes Dhaby
Mickael Lamotte
Olivia Lenormand
Valentin Pragassam

TABLE DES MATIÈRES

OBJECTIF DE LA PHASE 2

La phase n°2 du projet d'architecture logicielle consiste à reprendre le travail d'un autre groupe, et de choisir une parmi les deux implémentations possible (dans notre cas ici la partie "groupe", il n'y a qu'une unique implémentation possible). Faire un état de l'art de l'existant et le mettre à jour en accord avec les nouvelles exigences fonctionnelles et non fonctionnelle.

ETAT DE L'ART DE L'EXISTANT

Nous avons récupéré le travail effectué par le groupe n°4 sur la partie "Groupe".

Installation de l'existant

- Installer JavaFX :
help -> Eclipse market -> Rechercher "javafx" -> e(fx)clipse
- Importer le projet
- Avoir Java JDK 1.8

Nécessite le jar "sqlite-jdbc-3.27.2.1.jar"
Pour l'intégrer dans le projet il y a deux méthodes :
- soit le copier à la racine du projet , ensuite cliquer droit sur le projet *build path -> configure build path -> libraries -> add Jars* -> choisir le jar qui est à la racine -> *apply and close*
- Soit l'importer depuis un dossier local en cliquant droit sur le projet *build path -> configure build path -> libraries -> add External JARs* -> choisir le jar que vous avez téléchargé sur votre ordinateur.

Architecture utilisée dans l'existant

Modèle MVC(Modèle-Vue-Contrôleur)
Chaque vue possède un modèle et un contrôleur.

Le fichier MainApp.java lance la fênetre ainsi que la 1ère interface de connexion. Le MainApp.java lance également notre base de données locale par l'appel : *DataBaseManager.initialisationBD()* qui va créer un fichier groupe.db à la racine du projet si celui-ci n'existe pas déjà, puis stocker la connexion à cette base dans un attribut de type Connection. Elle va ensuite créer les différentes tables si elles n'existent pas déjà.

Toute la gestion de la base de données se fait grâce à la classe DataBaseManager.
Les fonctions sont séparées en quatre parties qui correspondent aux quatre opérations CRUD.

Environnement matériel & logiciel

Aucune spécification particulière, code compilable sous Windows, Mac OS et Linux.
- Java 8
- JavaFx 8
- SQLite : système de gestion de base de données léger, gratuit, facile à implémenter et fortement documenté. Pour l'utilisation d'une telle base de données nous utilisons sqlite-jdbc.jar situé à la racine du projet()
- Scene Builder : outil interactif de conception d'interface graphique pour JavaFX. Ceci n'est pas indispensable pour le fonctionnement de l'application mais cet outil aide à concevoir des interfaces rapidement et facilement.

Entrées et sorties des composants

Notre application est basé sur une architecture MVC , donc 3 composant principaux :
- **Modèle**:
  Entrée : la requête invoquée
  Sortie : Exécution de la requête
- **Contrôleur**:
  Entrée : Demande de l'utilisateur
  Sortie: Réponse à l'utilisateur , lancement d'une requête(Appel du modèle)
- **Vue** :
  Entrée : données du modèle
  Sortie : l'interface mise à jour

Exigences réalisées

Non fonctionnelle :

- [Req-doc-01] La documentation doit décrire la procédure d'installation.
- [Req-doc-02] La documentation doit décrire la procédure de désinstallation.
- [Req-doc-03] La documentation doit décrire Les entrées et sorties de chaque composant.
- [Req-doc-04] La documentation doit décrire l'environnement matériel nécessaire à l'installation.
- [Req-doc-05] La documentation doit décrire l'environnement logiciel nécessaire à l'installation.
- [Req-doc-06] La documentation doit décrire le cheminement des appels de fonctions pour 2 fonctionnalités.
- [Req-doc-07] La documentation doit lister les exigences satisfaites.
- [Req-doc-08] La documentation doit lister les exigences non satisfaites.
- [Req-doc-09] La documentation doit lister les anomalies.
- [Req-doc-10] La documentation doit décrire la procédure de compilation.
- [Req-exp-01] Le système doit fonctionner sous Linux.

- [Req-exp-02] Le système doit fonctionner sous Windows.
- [Req-exp-03] Le système doit fonctionner sous Mac OS.
- [Req-exp-04] Le système doit être développé en Java 8.
- [Req-liv-01] La livraison doit contenir tous les éléments nécessaires à la génération de la version binaire.
- [Req-liv-02] La livraison doit contenir la version binaire du système.
- [Req-liv-03] La livraison doit contenir toute la documentation.
- [Req-arc-01] Le système est constitué d'un seul exécutable.
- [Req-arc-02] Le programme principal instancie deux objets : l'un implémentant l'IHM et l'autre exposant les fonctionnalités.
- [Req-arc-03] Les communications entre IHM et fonctions passent par une unique interface Java.
- [Req-arc-04] Les requêtes vont uniquement de l'IHM vers l'objet exposant les fonctions.
- [Req-arc-05] Les opérations exposées par l'interface sont de 4 types : lecture, création, modification, suppression d'un objet.
- [Req-arc-07] L'identifiant des objets créés est attribuée par l'IHM.
- [Req-fct-01] Après re-démarrage du système il est dans le même état qu'avant son arrêt (données).
- [Req-fct-02] Il est possible de mettre à jour l'IHM (automatiquement ou à la demande de l'utilisateur)

Fonctionnelle

- [Req-gro-01] L'utilisateur peut créer une Unité d'Enseignement.
- [Req-gro-02] L'utilisateur peut supprimer une Unité d'Enseignement.
- [Req-gro-03] L'utilisateur peut créer un élève.
- [Req-gro-04] L'utilisateur peut supprimer un élève.
- [Req-gro-05] L'utilisateur peut créer un sujet.
- [Req-gro-06] L'utilisateur peut supprimer un sujet.
- [Req-gro-07] L'utilisateur peut créer un groupe (UE – élèves – sujet)
- [Req-gro-08] L'utilisateur peut supprimer un groupe (UE – élèves – sujet)
- [Req-gro-09] L'utilisateur peut créer aléatoirement des groupes pour un ensemble d'élèves.
- [Req-gro-10] L'utilisateur peut changer un élève de groupe.

Exigences non réalisées

[Req-arc-06] Les classes implémentant l'IHM et les fonctionnalités sont packagées dans des jar distincts.

ANALYSE FONCTIONNELLE

Nouvelles exigences fonctionnelles

- **Req-arc-08** : Le système est composé de deux parties : un client présentant l'IHM et un serveur gérant les données (priorité 1)
- **Req-arc-09** : Le client et le serveur peuvent être déployés sur deux machines distinctes (priorité 1)
- **Req-arc-10** : La communication entre client et serveur passe par une API REST (priorité 1)
- **Req-arc-11** : Le port d'écoute du serveur est paramétrable via un fichier dans un format standard (XML, JSON, YAML ou INI) (priorité 2)
- **Req-arc-12** : Le serveur utilisé par le client est paramétrable via un fichier dans un format standard (XML, JSON, YAML ou INI) (priorité 2)
- **Req-arc-13** : Le déploiement du serveur passe par une unique commande (priorité 3).
- **Req-arc-14** : Le démarrage du serveur passe par une unique commande (priorité 3).
- **Req-arc-15** : L'arrêt du serveur passe par une unique commande (priorité 3).
- **Req-arc-16** : L'API REST utilisée est exprimée en OpenAPI (priorité 4).

INSTALLATION

Procédure d'installation

- Installer JavaFX :
help -> Eclipse market -> Rechercher "javafx" -> e(fx)clipse
- Importer le projet
- Avoir Java JDK 1.8
- Créer un projet "Maven"

Nécessite le jar "sqlite-jdbc-3.27.2.1.jar"
Pour l'intégrer dans le projet il y a deux méthodes :
- soit le copier à la racine du projet , ensuite cliquer droit sur le projet *build path -> configure build path -> libraries -> add Jars* -> choisir le jar qui est à la racine -> *apply and close*
- Soit l'importer depuis un dossier local en cliquant droit sur le projet *build path -> configure build path -> libraries -> add External JARs* -> choisir le jar que vous avez téléchargé sur votre ordinateur.

Procédure de désinstallation

Afin de désinstaller le projet et les logiciels, il suffit de placer tous les fichiers et logiciels dans la corbeille et de les supprimer définitivement.

Environnement matériel nécessaire

Aucune spécification particulière, code compilable sous Windows, Mac OS et Linux.

Environnement logiciel nécessaire

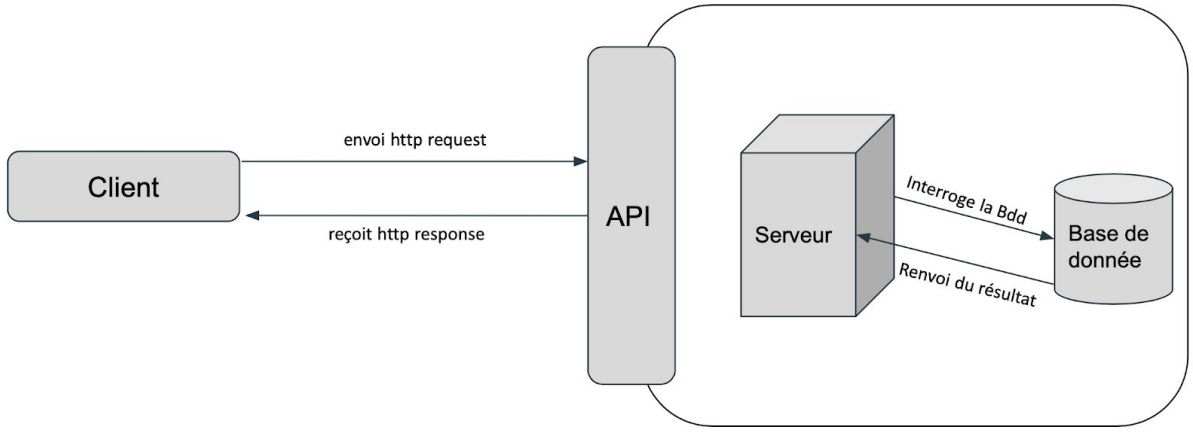
Afin de passer le projet en client-serveur, nous, avons utilisé “SpringBoot” qui est un framework qui facilite le développement d'applications fondées sur Spring en offrant des outils permettant d'obtenir une application packagée en jar , totalement autonome.

Pour générer le code client et serveur permettant d'exposer/utiliser l'API spécifiée, nous avons utilisé l’outil “Swagger.lo”, qui est un framework logiciel open-source soutenu par un large écosystème d'outils qui aide les développeurs à concevoir, construire, documenter et consommer des services web REST. Alors que la plupart des utilisateurs identifient Swagger à l'aide de l'outil d'interface utilisateur Swagger, l'ensemble d'outils Swagger prend en charge la documentation automatisée, la génération de code et la génération de scénario de test.

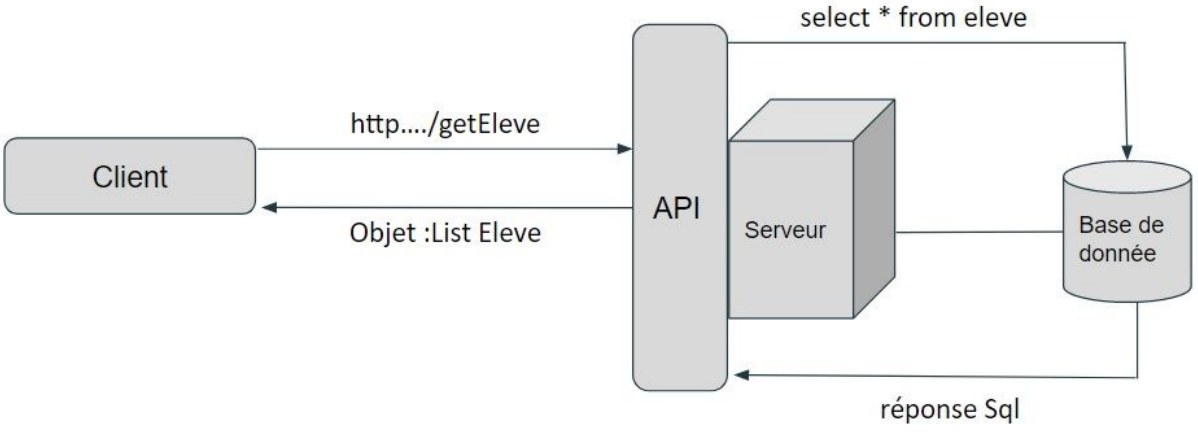
Nous avons utilisé ngrok afin de rendre public notre localhost via le phénomène de “tunneling”. Il suffit d’installer ngrok et de transmettre aux concernés, l’URL fournie par le service qui sera de type : ***http://xxxx.ngrok.com*****.**

Nous rappelons également l'environnement logiciel nécessaire à la première partie du projet, et toujours valable pour cette seconde partie :
- Java 8
- JavaFx 8
- SQLite : système de gestion de base de données léger, gratuit, facile à implémenter et fortement documenté. Pour l’utilisation d’une telle base de données nous utilisons sqlite-jdbc.jar situé à la racine du projet()
- Scene Builder : outil interactif de conception d'interface graphique pour JavaFX. Ceci n’est pas indispensable pour le fonctionnement de l’application mais cet outil aide à concevoir des interfaces rapidement et facilement.

ENTRÉES ET SORTIES DE CHAQUE COMPOSANT



EXEMPLE D'APPEL DE FONCTIONS



EXIGENCES

Les exigences communes aux deux parties du projet sont restées inchangées. Nous nous préoccupons donc des nouvelles exigences.

Exigences réalisées

Voici les exigences réalisées parmi les nouvelles exigences de la phase 2 :

- **Req-arc-08** : Le système est composé de deux parties : un client présentant l'IHM et un serveur gérant les données (priorité 1)
- **Req-arc-09** : Le client et le serveur peuvent être déployés sur deux machines distinctes (priorité 1)

- **Req-arc-10** : La communication entre client et serveur passe par une API REST (priorité 1)
- **Req-arc-11** : Le port d'écoute du serveur est paramétrable via un fichier dans un format standard (XML, JSON, YAML ou INI) (priorité 2)
- **Req-arc-12** : Le serveur utilisé par le client est paramétrable via un fichier dans un format standard (XML, JSON, YAML ou INI) (priorité 2)
- **Req-arc-16** : L'API REST utilisée est exprimée en OpenAPI (priorité 4).

Exigences non réalisées

- **Req-arc-13** : Le déploiement du serveur passe par une unique commande (priorité 3).
- **Req-arc-14** : Le démarrage du serveur passe par une unique commande (priorité 3).
- **Req-arc-15** : L'arrêt du serveur passe par une unique commande (priorité 3).

ANOMALIES

- Sous Mac, l'icône permettant de rafraîchir l'affichage de la base de donnée n'est pas visible mais fonctionne tout de même. Il est donc nécessaire d'avoir connaissance de son emplacement pour pouvoir l'utiliser.
- Malheureusement, nous n'avons pas réussi à exporter la partie serveur. Nous supposons que le problème est dû au fichier jdbc (lié à la base de donnée) qui a été implémenté dans la partie cliente. Si tel était le cas, nous pensons qu'il faudrait faire une dépendance du fichier jdbc dans la partie serveur.

PROCÉDURE DE COMPILATION

La procédure de compilation ne comporte pas de spécificité. Nous utilisons donc la procédure de compilation standard d'Eclipse.
Il faut lancer le serveur avant d'exécuter le client !
Pour lancer le serveur il faut :

- a la racine du projet serveur → aller dans le dossier serveur → ouvrir un terminal dans le dossier et taper la commande :
  ngrok authtoken1UNsvMGBB3yTzgyHXlp389BFspo_885rnVyDm2yxgxNh2u5sg
  seulement a la 1ère execution
- Pour lancer : ngrok http 9090

Pour chaque client :

- a la racine du client ouvrir le fichier config.xml et recopier l'url donnée par ngrok et lancer l'exécutable client
- Pour effectuer un changement de port : aller dans application.yml