TP2. Des Tests Unitaires vers les Tests Système

Dans ce TP, nous continuons sur l'utilisation de JUnit, étendu à de nouveaux contextes, pour la définition de tests d'intégration système. Il s'agit ici de tester l'application dans un environnement d'exécution, plutôt que de ne s'intéresser qu'à des interactions des composants dans l'environnement de développement. En particulier, on va s'intéresser au test d'applications client-serveur développés en Java suivant le framework SpringBoot, permettant le chargement d'un contexte d'application à son démarrage. Les tests systèmes interviennent sur cette application démarrée sur le serveur. En premier lieu nous nous intéresserons à des tests de la partie backend des applications au travers de l'api qu'elles fournissent, avant de tester le système de bout en bout, en mettant en place des interactions avec le front-end (ou partie client de l'application).

Pour celà, nous travaillerons sur deux application à disposition sur le moodle de l'UE, la première pour prendre en main les différents concepts, la seconde pour les appliquer dans un contexte client-serveur plus réaliste (mais plus complexe) :

- Calculator : application client-serveur simple proposant une api d'opérations arithmétiques de base entre deux nombres saisis par un utilisateur [1] ;
- Blog : application Web de type Blog [2], avec accès sécurisé et manipulation de bases de données. Les tests avec cette application se feront en utilisant une base de données de type H2 chargée en mémoire au chargement de l'application, qui pourrait aisément être remplacée par toute forme de base de données SQL persistante lors de son déploiement final.

Les deux applications s'exécutent sous le framework SpringBoot, qui est un environnement Java basé sur un système modulaire de dépendances et un principe de configuration automatique, très populaire pour le déploiement d'applications Web. Springboot met en œuvre un principe d'injection de dépendance permettant de diminuer le couplage entre les objets manipulés. Il se base sur un contexte d'application, destiné à contenir les objets utiles à son fonctionnement. Ces objets - appelés Beans - sont chargés à leur création et utilisés par les différents services qui en ont besoin. Les Beans sont déclarés par des annotations spécifiques dans le code de leur déclaration. Pour plus de détails sur le framework Spring, un bon tutoriel est donné à l'adresse : `https://gayerie.dev/docs/spring/spring/introduction.html`. Pour travailler avec Spring Boot sous Eclipse, il s'agit de se rendre dans help>Eclipse Marketplace, puis installer "Spring Tools 4 (aka Spring Tool Suite 4)".

Les deux applications se lancent via la commande (après compilation avec mvn compile et éventuellement nettoyage via mvn clean) : `mvnw spring-boot:run` à partir du répertoire où se trouve le fichier pom.xml.

Pour l'application de Blog dont la partie frontEnd est codée séparément en React, il s'agira de la compiler via `npm install` puis de la lancer via `npm run start` à partir du repertoire FrontEnd, ce qui ouvrira l'application client à l'adresse `http://localhost:3000` de votre navigateur par défaut.

1. Disponible à l'adresse : `https://github.com/geoffreyarthaud/oc-testing-java-cours/`
2. Disponible à l'adresse : `https://github.com/keumtae-kim/spring-boot-react-blog`

Tests d'Intégration Système

Les tests d'intégration système partent d'une démarche similaire aux tests d'intégration composants, à savoir assembler des composants entre eux, mais de manière plus large, avec plusieurs services, une configuration d'application, et des liens éventuels avec des composants extérieurs à l'application, simulés ou en production.

Exercice 1 – Illustration avec Calculator

Spring peut lancer un environnement simulé qui laisse agir les tests comme si on avait un serveur web fonctionnel. Pour cela, on utilisera l'annotation @SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT) indiquant à Spring de charger l'application et la déployer sur un port aléatoire que l'on utilisera pour nos tests. L'exemple ci-dessous utilise ce mécanisme pour mettre en place les conditions d'un test d'intégration du module de calcul dans son contexte d'application :

```
1  @ExtendWith(SpringExtension.class)
2  @SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
3  public class CalculatorSIT {
4
5      @Autowired
6      private ApplicationContext context;
7
8      @Test
9      public void testAddTwoPositiveNumbers() {
10         int res=context.getBean(Calculator.class).add(3,4);
11         assertEquals(res,7);
12     }
13
14 }
```

Bien que cet exemple ne teste qu'un composant (l'objet Calculator), nous pouvons le qualifier de test d'intégration Système (assez arbitrairement, la frontière entre test système et test composant étant assez floue / subjective) car on le teste via récupération selon le contexte de l'application déployée (ce qui justifie le suffixe de la classe de test en SIT - S pour Système). Dans cet exemple on récupère le contexte d'application via la ligne 6, puis on s'en sert pour récupérer le Bean Spring qui nous intéresse : une instance de la classe Calculator. L'annotation @Autowired (équivalent à @Inject) utilisée à la ligne 5 indique à Spring de fournir une instance d'une classe et de gérer sa création à partir d'une classe ou méthode Bean issue de son contexte d'application. Pour que Spring puisse créer cette instance, son contexte doit comporter un constructeur de la classe d'objet demandée, désignée avec @Service, @Component, @Controller, @Bean ou encore @Named.

Q 1.1 Expérimenter ce code pour en comprendre le fonctionnement sur votre environnement de développement.

Un aspect plus intéressant à considérer dans nos tests d'intégration système est le bon fonctionnement des APIs que l'application fournit via ses contrôleurs. Il s'agira de simuler l'envoi de requête HTTP à ses contrôleurs et d'en vérifier le bon traitement. Cela peut se faire un objet MockMvc comme dans l'exemple ci-dessous :

```
@ExtendWith(SpringExtension.class)
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
@AutoConfigureMockMvc
public class CalculatorControllerSIT {

    @Inject
    private MockMvc mockMvc;

    @Test
    public void givenACalculatorApp_whenRequestToAdd_thenSolutionIsShown() throws
     Exception {
        final MvcResult result = mockMvc.perform(
                MockMvcRequestBuilders.post("/calculator")
                        .param("leftArgument", "2")
                        .param("rightArgument", "3")
                        .param("calculationType", "ADDITION"))
                .andExpect(MockMvcResultMatchers.status().is2xxSuccessful())
                .andReturn();

        assertThat(result.getResponse().getContentAsString())
                .contains("id=\"solution\"")
                .contains(">5</span");
    }
```

Dans l'exemple ci-dessus, on teste le fonctionnement du contrôleur CalculatorController, en charge de recevoir les requêtes de calcul et de retourner le résultat dans une réponse HTTP. L'objet MockMVC, auto-configuré grâce à l'annotation @AutoConfigureMockMvc, est en charge de produire une requête HTTP à envoyer via un protocole POST à ce contrôleur, dont l'API est déployée à l'adresse relative "/calculator". On teste ensuite le statut de la réponse HTTP obtenue et on vérifie la correction de la solution retournée (présentée dans un format HTML dans ce cas).

Q 1.2 Expérimenter ce code pour en comprendre le fonctionnement sur votre environnement de développement.

Pour tester la levée d'exceptions, par exemple dans le cas d'une division par zero dans notre cas, nous avons ajouté un gestionnaire d'exceptions Spring dans le contrôleur testé, dont voici le code :

```
@ExceptionHandler(IllegalArgumentException.class)
    public ResponseEntity<?> illegalArgumentException(IllegalArgumentException ex
    ) {
        return new ResponseEntity<>(ex.getMessage(), HttpStatus.BAD_REQUEST);
    }
```

Son rôle est de capturer l'exception et de retourner une reponse HTTP appropriée.

Q 1.3 S'en servir pour tester la levée d'exception dans le cas d'une division par zero (vérifier que l'on récupère un statut de réponse http "bad request").

Q 1.4 MockBean : Jusqu'ici on a testé le contrôleur en utilisant l'ensemble du système déployé, ce qui peut rendre l'interprétation des résultats difficiles. Bien que cela soit finalement nécessaire, il est recommandé de réaliser également des tests préliminaires en isolation, c'est à dire excluant un certain nombre de composants du test, pour ne tester qu'une partie du système. Ceci est rendu possible dans ce cadre via l'annotation @MockBean permettant de simuler des Beans, à la manière de ce que l'on faisait au TP précédent sur les objets classiques : C'est la même chose que le @Mock de Mockito, sauf que Spring utilise ces objets comme des beans Spring utilisables par d'autres beans Spring. Par exemple, l'attribut de type CalculatorService nécessite qu'une instance de calculateur soit passée à son constructeur. En annotant Calculator avec @MockBean, Spring lui injectera le mock de calculateur à la place d'un calculateur réel. Expérimenter ce procédé sur le test de la question 1.1, en mockant les Bean Calculator et SolutionFormatter. On testera le nombre d'appels et les arguments des méthodes appelées sur ces objets simulés (via verify).

Q 1.5 WebMvcTest : On note enfin que `@SpringBootTest` déploie la totalité du contexte de l'application, ce qui peut rendre la suite de tests très lente, alors que nous ne sommes parfois intéressés que par l'API fournie par le contrôleur. Pour ne charger que le contrôleur testé (et certains composants associés), il est possible de remplacer cette annotation par `@WebMvcTest` , permettant de spécifier pour notre classe de test les seuls éléments du contexte à charger, imposant à l'ensemble des autres composants d'être mockés via MockBean. Remplacer dans votre code de la question précédente `@SpringBootTest` par `@WebMvcTest(controllers = {composants})`, avec composants la liste des classes de composants réels à charger dans le contexte (rien d'autre ne devrait être à modifier dans le code de la question précédente pour que le test passe, si les bons composants ont été spécifiés).

Exercice 2 – Implémentation des tests systèmes sur l'application de Blog

Il s'agit ici d'implémenter ces tests système sur notre application de blog. On commence par des tests sur les services de l'API accessibles via des requêtes GET, ne nécessitant pas d'authentification.

Q 2.1 Tester le service de récupération d'un post particulier, en interrogeant l'API à l'adresse "/api/posts/id", où *id* correspond à l'identifiant du post à récupérer (par exemple 1). On testera que le texte du post obtenu correspond bien à celui du post possédant cet identifiant dans la base de données (les données insérées dans la base à son démarrage sont spécifiées dans src/main/ressources/data.sql). On testera également le service avec un identifiant inexistant dans la base.
On s'intéresse maintenant au service d'authentification via le contrôleur AuthController.

Q 2.2 Tester l'authentification avec les identifiants email="admin@mail.com", password="admin", puis avec des identifiants invalides. Les identifiants de connexion doivent être passés dans le corps de la requête http en JSON, via la fonction content(). On teste ici seulement le statut de la réponse.
Le service d'authentification retourne dans le corps de la réponse un champ token, qui est utilisé pour toutes les autres requêtes de type POST soumises à l'API de l'application.

Q 2.3 Après avoir récupéré le token d'authentification, tester que l'utilisateur connecté est bien

le bon en utilisant le service getUser offert à l'adresse "/auth/user". Le token devra être inséré dans le header de la requête http via la fonction header(). On pourra tester avec un token valide (tester que l'utilisateur a bien la bonne adresse mail) et avec un token invalide (tester le statut non-autorisé de la réponse).
On peut donc accéder à tous les services sécurisés, en simulant une authentification, en récupérant le token et en l'utilisant pour lancer la requête ciblée. Cela est cependant un peu lourd. Une manière plus légère de réaliser les tests de ces services sécurisés est d'utiliser l'annotation @WithUserDetails(value=VALUE, userDetailsServiceBeanName=SERVICE), avec VALUE le nom d'utilisateur avec lequel le test doit être réalisé (ici son mail) et SERVICE le nom du service de gestion des utilisateurs dans l'application (ici "customUserDetailsService", injecté automatiquement à partir de la classe WebSecurityConfig.class).

Q 2.4 Tester le service d'ajout de post via le PostController, en utilisant cette annotation @WithUserDetails plutôt qu'en récupérant le token d'une authentification préalable. On testera que la reponse contient bien un post avec le bon titre et le bon texte.

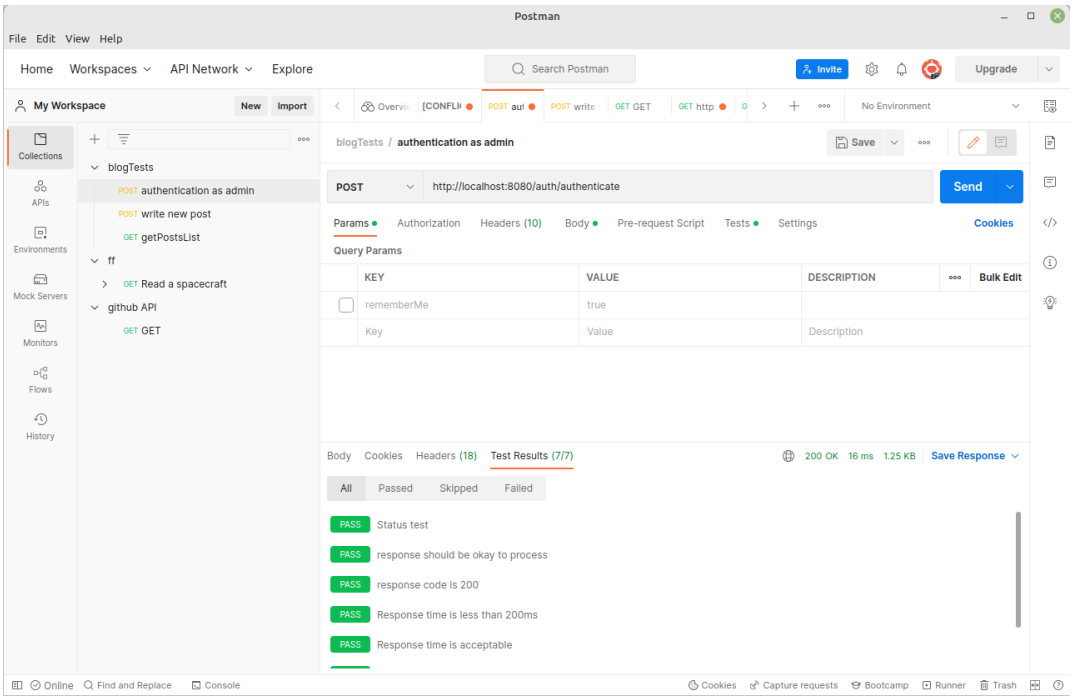
Q 2.5 Plutôt que de simplement tester la reponse HTTP, une autre possibilité est de vérifier que le post inséré a bien été ajouté en base. Pour cela on peut utiliser le contexte de l'application en déclarant une variable d'instance ApplicationContext context à la classe de test (précédée de l'annotation @Autowired). Dans la méthode de test, on peut récupérer le service qui gère la base de Posts, via context.getBean("postService"). Compléter le test précédent en utilisant ce service pour tester que le post inséré via le contrôleur existe bien en base.

Exercice 3 – Tests d'API via Postman

Une alternative aux tests d'API via mockMVC est l'utilisation d'outils externes tels que PostMan ou Swagger. Postman est un client API utilisé pour développer, tester, partager et documenter des API. Il est utilisé pour les tests de backend où nous entrons l'URL du point final, il envoie la requête au serveur et reçoit la réponse du serveur. Tout ce qui est fait via Postman peut l'être via les Junit et MockMVC vus aux sections précédentes, mais Postman est largement utilisé en entreprise pour sa facilité d'accès. Bien que plus limité pour faire de l'intégration continue, il permet de mettre en place rapidement des premiers tests de manière exploratoire. En outre, il permet un dialogue facilité avec des APIs tierces.

L'installation de Postman se fait sous linux via le gestionnaire de paquets snap via `snap install postman`. Si ce gestionnaire de paquets n'est pas installé sur votre environnement, la procédure pour l'installer est la suivante :

```
// 1iere ligne optionnelle sous certains environnements
sudo mv /etc/apt/preferences.d/nosnap.pref ~/Documents/nosnap.backup
sudo apt update && apt upgrade
sudo apt install snapd
// Pour tester l'install (dans un nouveau bash ou apres reboot) :
snap install hello-world
hello-world
```



Bien que Postman embarque bien d'autres fonctionnalités (il peut être utilisé pour déployer une API), nous nous concentrerons ici sur la création de collections de tests. On commence par créer une collection dans le workspace, à l'intérieur de laquelle il est possible de définir des requêtes http avec des tests associés.

Q 3.1 Définir une requête GET permettant de récupérer la liste de tous les posts du serveur et observer le résultat obtenu.
Postman associe à chaque requête un onglet Tests permettant de définir un script de tests concernant cette réponse reçue. L'exemple ci-dessous teste si la requête a obtenu un statut de succès, qu'elle a été exécuté en un temps raisonnable et qu'elle a retourné une liste d'objets dont le premier possède bien le texte attendu en tant que titre :

```
tests["response code is 200"] = responseCode.code === 200;
tests["Response time is less than 200ms"] = responseTime < 200;
var jsonData = JSON.parse(responseBody);
tests["get post 1"] = jsonData[0].title === "The standard Lorem Ipsum passage";
```

Q 3.2 Lancer ce test et observer le résultat. Ajouter un test sur le nombre de posts contenus dans la liste.

Q 3.3 Définir maintenant une requête POST qui permet l'authentification de l'utilisateur admin@mail.com dont le mot de passe est "admin". Les identifiants de l'utilisateur sont à définir dans Body, sous la forme d'un texte raw au format JSON. Tester la réponse, notamment que celle-ci rencontre bien un succès et qu'elle contienne un token d'authentification. On en profitera pour sauvegarder ce token dans une variable de la collection, via pm.collectionVariables.set(key,value), pour usage dans futur.

Q 3.4 Définir une requête permettant d'enregistrer un nouveau post sur le serveur, en utilisant le token sauvegardé à la question précédente. Ce token est à ajouter dans le header de la requête (les

variables sauvegardées sont accessibles dans le formulaire via {{nom_variable}}), les informations sur le post (title et body) sont à ajouter dans le corps de la requête sous le format JSON raw. Tester le succès de la requête, ainsi que la réponse contient bien le bon post.

Q 3.5 On testera ensuite qu'un appel au service retournant la liste des posts contient bien le nouveau post ajouté à la question précédente, avec l'identifiant le plus élevé.

Q 3.6 Enfin, il est d'usage de nettoyer les modifications effectuées pour garantir au tests futurs qu'ils se déroulent correctement. On en profite alors pour tester également le service de suppression, que l'on utilisera pour effacer de la base le nouveau post ajouté précédemment.

Tests de bout en bout

Alors que la section précédente se limitait à des tests sur la partie backend des applications, nous allons maintenant étudier la manière de procéder pour tester le système de bout en bout, en y incluant l'interface utilisateur fournie par la partie frontend de l'application déployée. Ceci se fait sous Java via un WebDriver fourni par la librairie Selenium, qui est en charge de lancer un navigateur et simuler les actions d'un utilisateur sur ce navigateur. Il existe des WebDriver pour les différents types de navigateur existants, nous nous limiterons dans ce TP à l'utilisation du WebDriver permettant d'interagir avec Firefox (s'assurer que Firefox soit bien installé sur votre machine), bien qu'il faudrait en pratique les considérer tous pour garantir un bon fonctionnement sur les différents environnements.

Exercice 4 – Illustration avec Calculator

L'exemple ci-dessous teste de bout en bout l'utilisation de l'application de calcul pour réaliser une opération de multiplication via son interface web :

```
1  @ExtendWith(SpringExtension.class)
2  @SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
3  public class StudentMultiplicationJourneyE2EIT {
4
5      @LocalServerPort
6      private Integer port;
7      private WebDriver webDriver = null;
8      private String baseUrl;
9
10     @BeforeAll
11     public static void setUpFirefoxDriver() {
12         WebDriverManager.firefoxdriver().setup();
13     }
14
15     @BeforeEach
16     public void setUpWebDriver() {
17         webDriver = new FirefoxDriver();
18         baseUrl = "http://localhost:" + port + "/calculator";
19     }
20
21     @AfterEach
22     public void quitWebDriver() {
23         if (webDriver != null) {
24             webDriver.quit();
```

```
25          }
26      }
27
28      @Test
29      public void aStudentUsesTheCalculatorToMultiplyTwoBySixteen() {
30          webDriver.get(baseUrl);
31
32          // Récupération des éléments de la page
33          final WebElement leftField = webDriver.findElement(By.id("left"));
34          final WebElement typeDropdown = webDriver.findElement(By.id("type"));
35          final WebElement rightField = webDriver.findElement(By.id("right"));
36          final WebElement submitButton = webDriver.findElement(By.id("submit"));
37
38          // Envoi d'informations au formulaire
39          leftField.sendKeys("2");
40          typeDropdown.sendKeys("x");
41          rightField.sendKeys("16");
42          submitButton.click();
43
44          // Attente jusqu'à ce que la solution soit visible puis récupération de l
    'élément
45          final WebDriverWait waiter = new WebDriverWait(webDriver, 5);
46          final WebElement solutionElement = waiter.until(
47                  ExpectedConditions.presenceOfElementLocated(By.id("solution")));
48
49          // Verification du resultat
50          final String solution = solutionElement.getText();
51          assertThat(solution).isEqualTo("32"); // 2 x 16
52
53      }
54  }
```

Dans cet exemple, après avoir configuré le driver, on le branche sur l'url du service à tester (en utilisant le port attribué pour le déploiement de l'application), puis on s'en sert pour récupérer les éléments de l'arbre DOM de la page affichée. Ici on récupère les éléments web via leur id selon *By.id*. Il est également possible de les repérer via leur nom de classe selon *By.className* ou via xpath (langage de requête pour localiser une portion d'un document XML) selon *By.xpath*. Une fois les éléments de formulaire récupéré, le driver leur envoie les données utiles au test et soumet de le formulaire. Enfin, une fois que l'élément de solution est visible sur la page, on en vérifie le contenu en fonction de l'attendu.

Q 4.1 Expérimenter ce code sur votre environnement et observer le comportement.

Exercice 5 – Implémentation des tests de bout en bout sur l'application de Blog

Q 5.1 Test de la page principale : On souhaite vérifier que le 1er post affiché dans la liste a bien pour titre "The standard Lorem Ipsum passage". Indication : l'élément Web correspondant peut se retrouver par le xpath : ".//h2//a[@href='/posts/1']".

Q 5.2 Test affichage commentaires : On souhaite vérifier que les commentaires associés à un post s'affichent bien lorsque l'on clique sur son titre. En démarrant de la page principale, tester que le 1er commentaire du premier post est bien le bon (selon ce que l'on a chargé via data.sql).

Q 5.3 Test d'authentification : En démarrant de la page d'accueil, tester la procédure d'authentification (par exemple en vérifiant que le lien de création d'un nouveau post s'affiche après la soumission du formulaire d'authentification si les identifiants sont corrects). Tester également avec des identifiants incorrects. Pour cela, on pourra s'assurer du dépassement du temps d'attente de l'affichage du lien d'ajout d'un nouveau post, via :

```
1 assertThrows( org.openqa.selenium.TimeoutException.class , () -> {new
      WebDriverWait(webDriver , 3).until(ExpectedConditions.
      visibilityOfElementLocated(By.xpath(".//a[@href='/editor']"))));});
```

Q 5.4 Test d'insertion d'un nouveau post :

- S'identifier (on peut démarrer de /login si on le souhaite)
- Clic sur "New Post"
- Saisie du titre et du texte, puis clic sur "Save" (Indications : l'élément de saisie du titre peut se récupérer via le xpath ".//input[@type='text']", l'élément de saisie du corps du post via "//div[@class='notranslate public-DraftEditor-content' and @role='textbox']" et le bouton de soumission via ".//button[@class='pull-left btn btn-info']")
- Vérification qu'un post s'affiche bien avec le bon titre et le bon texte

Q 5.5 Test d'insertion d'un nouveau post (suite) : on aimerait laisser la base telle qu'à l'origine pour ne pas gêner d'éventuels tests suivants. On souhaite alors supprimer le post nouvellement créé. En fin de test d'ajout on ajoutera donc un Clic sur le bouton "Delete", dont on pourra vérifier le bon fonctionnement en s'assurant que le premier post de la liste affichée est toujours le premier original (celui d'identifiant 1)... Mais çà coince ! Qu'observez vous ? Quel est le problème ? Proposer une solution.

Q 5.6 Faire la même chose mais en testant directement la présence puis l'absence du post dans la base de donnée via le service (histoire de mixer les plaisirs :-))