Département de génie informatique et de génie logiciel

LOG3430

Méthodes de tests et validations

TP1

Tests unitaires

Soumis par :

Jean-Frédéric Fontaine (1856632)

Sébastien Cadorette (1734603)

2 octobre 2018

Soumis à : Hiba Bagane

Polytechnique Montréal

Introduction

Le présent rapport a pour objectif d'expliquer en détail le processus de développement suivi lors de la création des tests dans le cadre du TD1 du cours LOG3430. Plus précisément, nous débuterons par une brève explication des différentes techniques permettant d'isoler la logique interne des fonctions à tester, c'est-à-dire les Mocks, les Stubs et les Spies. Par la suite, nous expliquerons dans quel contexte nous avons utilisé ces outils. Enfin, nous répondrons aux différentes questions proposées par l'énoncé.

Mocks, Stubs & Spies

La distinction entre les Mocks et les Stubs n'est pas toujours clair. Pour bien comprendre celle-ci il faut se rappeler la motivation derrière leur existence. Comme nous l'avons vue dans le cours , les tests unitaires requièrent souvent d'isoler la logique du code à tester. Bien qu'une isolation complète ne soit pas forcément signe d'un bon test, il est fréquent qu'une isolation partielle soit essentielle afin d'éviter les effets secondaires *(side effects).* Par exemple, les méthodes que nous avons à tester dans le cadre de ce TP effectuent régulièrement des requêtes HTTP à un serveur distant. Le temps que peuvent prendre ces requêtes est dépendant de plusieurs facteurs qui n'ont rien à voir avec le code à tester. Ainsi, il serait préférable de simuler ce serveur à l'aide d'un mécanisme comme un Mock ou un Stub. Mais lequel choisir ? Ces deux outils ont le même objectif mais leur niveau d'implémentation diffère. En effet, un Stub consiste en une construction minimaliste du module à simuler. Un Stub n'effectue pas de transformation ou d'opérations logiques, il n'a qu'une seule raison d'être ; retourner ce qu'on lui demande de retourner. Par exemple, les requêtes HTTP décris ci-haut sont réalisées par la méthode *Fetch* de la librairie *Util*.

En ce qui concerne les *Mock* de *Sinon.js*, ceux-ci permettent de faire plusieurs *expects* d'un seul coup en utilisant la méthode *Verify*. Aussi, un *Mock* est généralement associé à un type de *Stub* dans lequel on insère de la logique et des états. Dans le contexte de ce TP, nous n'avons pas fait usage des *Mocks*. Ainsi, nous avons utilisé des Stubs afin d'intercepter les requêtes à l'API. La librairie *Sinon.js* nous permet de rediriger l'appel à la méthode *Fetch* vers un Stub qui répondra

avec une réponse prédéterminée. Aussi, étant donné que le comportement internes de la méthode Fetch nous n'était d'aucun intérêt, nous n'avons pas développé de classe maison de type *Fake*.

Les Spies (ou espions), ont une raison d'être complètement différente. Les Spies n'ont pas pour objectif d'interagir directement avec le code à tester. Les Spies servent plutôt à observer les interactions et le comportement de la méthode qui est testé ainsi que les arguments avec lesquelles celle-ci a été appelée. Par exemple, on pourrait espionner la méthode *Fetch* afin de s'assurer que celle-ci est appelé. Il est a noté que les fonctionnalités associées aux *Spies* sont disponibles tant pour les *Stub* que pour les *Mocks*. Ainsi, il est possible d'étudier le nombre de fois qu'un *Stub* a été appelé.

Questions

Quelle méthode avez-vous choisi pour empêcher la classe JsonClient d'exécuter les vrais appels API durant vos tests unitaires?

Comme mentionné ci-haut, les appels à l'API effectué par la classe *JsonClient* consistent en des requêtes HTTP. Il est impératif de ne pas dépendre du protocole TCP/IP lorsqu'on effectue des tests unitaires. De plus, il n'est pas nécessaire de vérifier les manipulations qui sont fait du coté serveur. Ainsi, nous avons utilisé la librairie *Sinon.js* afin de générer un *Stub*. Nous avons utilisé les *Json* fournit avec le TP afin de simuler les réponses du serveur.

Quelle méthode avez-vous choisi pour empêcher la classe Client d'exécuter les vrais appels API contenus dans les méthodes de la classe JsonClient durant vos tests unitaires?

Idem que pour la question précédente. Nous avons fait usage des *Stubs* offert avec la librairie *Sinon.js* ainsi que les exempes *Json* fournit avec l'énoncé.

Est-ce que vos tests unitaires ont découvert une ou plusieurs défaillances? Si oui, expliquez ce que vous avez trouvé ?

Oui, une erreur a été trouvé dans la méthode *toJson* de la classe *Sharedbox*. Tout d'abord, tous les attributs de la classe ainsi que leur valeur sont enveloppés dans le *JSON* par une clé *Sharedbox*. Lors des tests, il nous a fallu plusieurs minutes avant de réaliser que nous devions prendre le résultat retourné par la fonction et appeler le *.sharedbox* avant de pouvoir récupérer les éléments.

```
const sharedboxJson = sharedbox.toJson();
const guid = sharedboxJson.sharedbox.guid;
```

De plus, lorsqu'on transfert en *JSON* avec la méthode *toJson*, qu'ensuite on *parse* à l'aide de la méthode *JSON.parse(/*Item to parse*/),* et que finalement on crée un objet avec le *JSON* parsé, certains attributs ne suivent pas, comme par exemple les options de sécurité. Dans le fichier *Sharedbox.spec.js*, un *.skip* est ajouté après chaque *it* qui test un attribut erroné.

Le projet implémente un type d'exceptions personnalisé SharedBoxException qui étend le type standard Error. Lorsque vous avez testé les exceptions qui peuvent être lancées par la classe Client, était-il possible de vérifier le type de l'exception? Si non, expliquez pourquoi et proposez une solution.

Oui, nous avons réussi à faire ce test. Nous avons simplement utilisé les outils disponibles avec le framework *Chai:*

```
assert.throws(function () { client.submitSharedBox(sharedbox); }, SharedBoxException);
```

Toutefois, deux fonctions de la classe *Client* y font exception, soient les fonction *addRecipient* et *closeSharedbox*. Il manque un bloc *.catch* après le *.then*, ce qui permettrait d'attraper l'erreur lorsque nous faisons nos tests. Il ne suffirait que d'ajouter un *.catch* après le *.then*, comme dans les autres fonctions. Prenons l'exemple de la fonction *submitSharedBox* qui a un bloc *.then* suivi d'un bloc *.catch* afin d'attraper les erreurs et de bien la relancer.

```
return this.jsonClient.submitSharedBox(sharedbox.toJson())
    .then(result => {
      /* Des instructions… */
    })
    .catch(error => {
      throw error;
```

```
});
```