Tests unitaires

TP1

Billy Bouchard

Jacob Dorais

Un Travail présenté à :

Hiba Bagane

Département Génie informatique et Logiciel

Polytechnique Montreal

le 2 Octobre 2018

Stub, Spy et Mock

Malgré que ces 3 éléments permettent de créer de fausses fonctions et de faux objets lors des différents tests, il devient très rapidement difficile d'identifier lesquels servent dans quelles cas. En effet, ces trois objets ont différents buts dans les tests et il est important de connaitre leur rôle. Le spy permet d'observer le comportement d'une fonction, Le stub permet d'en empêcher l'exécution et le mock permet même de planifier d'avance les appels voulus.

Spy

Le spy permet d'observer le comportement d'une méthode sans en modifier l'exécution. Cela peut rapidement devenir utile lorsqu'on veut connaitre les arguments de la fonction utilisée ou encore tout simplement savoir combien de fois cette dernière a été appelée. Malgré qu'il ne faut pas utiliser de spy lorsqu'on a l'intention de vouloir modifier le fonctionnement d'une méthode ou d'une fonction, on peut quand même observer tous les appels qui lui ont été faits. Il s'agit donc ici d'un "wrapper" qui enregistre toutes les entrée et sortie de la fonction ainsi que les appels, mais qui ne change pas son comportement. un spy est utilisable de cette façon :

```
var spy = sinon.spy(Object, 'methodName');
// pour verifier le spy a ete appeler avec le parametre arg
expect(spy.calledWith(arg)).to.be.true;
// pour simplement verifier s'il a   t   appel
expect(spy.called()).to.be.true
```

Stub

Le stub permet de faire la même chose que le spy, mais en plus il empêche l'exécution de la méthode. En fait, il permet plutôt de remplacer l'implémentation de base de la méthode par une plus simple qui fera exactement ce que l'on veut. Les stub sont donc des spy qui au lieu de "wrapper", remplacent simplement la fonction(ou méthode) par une autre implémentation. Ils sont extrêmement utiles lorsque l'on veut éviter de tester une méthode ou fonction dont l'implémentation cascade sur une dizaine de méthodes. On peut donc juste forcer le résultat de la première fonction plutôt que de trouver quelle valeur envoyer. Il est important de noter que toutes les méthodes existantes dans un spy sont disponibles dans un stub. La façon de créer un stub est la suivante :

```
var stub = sinon.stub(Object, 'methodName');
// Pour l'utiliser avec des promesses
stub.resolves(/*Objet a retourner en cas de resolution*/);
// ou
stub.rejects(/*Objet (souvent erreures) a retourner en cas de reject*/)
// sinon il est possible de simplement choisir un retour
```

```
stub.returns(/*Objet a retourner en tout cas*/)
```

Mock

Le Mock est le plus complet des modificateurs utilisés pour les tests unitaires. Ce dernier permet non seulement d'observer le comportement d'une méthode, mais aussi de lui donner le comportement désiré de l'appelant envers elle. Il devient donc possible de modifier le retour du mock en fonction des arguments qui lui sont passer. Contrairement au spy, le test échouera automatiquement si les paramètres attendus ne sont pas respectés lors des appels. Le Mock est donc un stub dans lequel on ajoute des attentes notamment quant au nombre d'appels et aux arguments avec lesquels les appels sont faits. Ainsi, le Mock est très utile lorsqu'on veut contrôler comment la méthode tester utilise notre mock.

Question

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

Nous avons choisi d'utiliser un stub de la méthode fetch de l'objet nodeFetch. Il s'agit d'une modification de la méthode qui remplace la définition utilisée dans le code de base. Un stub permet donc de remplacer complètement l'implémentation de la méthode fetch de l'objet nodeFetch. On peut alors choisir de renvoyer les réponses que l'on veut, que ce soit une promesse ou une simple valeur de retour. Dans le cas du programme actuel, on utilise le stub pour envoyer des promesses et éviter de faire de vraie requête http. On peut alors contrôler les paramètres des promesses afin de s'assurer que les résultats voulus soient appelés. On peut aussi choisir de faire différents retours en fonction du nombre de fois que la fonction est appelée. Ainsi, on peut choisir de retourner certaines données lors du premier appel et d'autres données lors du second. Le tout nous permet d'obtenir des tests encore plus précis sur certaines méthodes.

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 ?

Pour la majorité des appels, nous avons fait la même chose que pour la classe JsonClient. Nous avons donc utilisé un Stub sur certaines méthodes de la classe JsonClient. La fonction stub changeait souvent en fonction des fonctions testées, par conséquent elle était souvent différente. Cependant, pour certaines des fonctions, il est devenu plus utile de faire des Spy sur ces dernières. En effet, puisqu'il peut y avoir plus qu'un appel de ses sous méthodes, il est devenu important d'observer le nombre de fois où elles ont été appelées et les arguments avec lesquels elles ont été appelées. Ainsi, on ne change pas le fonctionnement de la méthode, mais on peut au

moins savoir si elle a été appelée avec les bons arguments ou encore si elle a été appelée le bon nombre de fois. Un exemple d'utilisation que nous avons fait a été pour vérifier si une des méthodes testées lançait une erreur.

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

Oui, ils ont trouvé un une défaillance dans 2 méthodes. Dans chacune des classes Sharedbox et Recipient, une méthode toJson est implémentée pour obtenir une chaîne de caractères de style JSON qui normalement nous permet de recréer l'objet en entier et retrouver tous ses attributs. Pour la tester, prenons Sharedbox comme exemple, nous avons donc, créer un objet Sharedbox, nous avons appelé la méthode toJson et se sont assuré que la valeur retournée est une chaîne de caractère. Ensuite, nous avons utilisé la fonction parse de Json qui sert de retransformer une chaîne de caractère Json en objet. Nous nous sommes assuré que c'était bien un objet qui était retourné. Nous avons créé un autre Sharedbox à partir de cet objet et avons comparé chaque attribut du nouveau Sharedbox avec l'ancien. Après avoir comparé, certains attributs n'étaient pas définis dans la nouvelle sharedbox. Pour la sharedbox, userId, status, previewUrl, createdAt, updateAt, expiration, closedAt, attachments ainsi que les security options étaient manquants. Pour le récipient, l'Id et les options étaient manquants.

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, il était même très facile de vérifier le type ! En effet, les librairies utilisées, sinon et chai, permettent facilement de vérifier si les fonctions utilisées 'throw'. De plus, dans le cas des promesses, on peut facilement choisir de simuler un 'reject' de la promesse. Toutes ces fonctions permettent de tester les erreurs. La fonction throw() de chai peut recevoir le type d'erreur en paramètre et donc il est possible de vérifier s'il s'agit de la bonne exception. Ainsi, on peut s'assurer qu'il s'agissait bien d'une SharedBoxException qui a fait échouer la fonction. De plus, puisque nous avons utilisé un stub sur les fonctions de jsonClient„ il était possible de forcer un certain type d'erreur et de voir si ce type suivait.