SOUCHET Mathéo

MILLET Arthur

PRESTI Louis

COSSEC Alban

LEFEVRE Lilian

RAPPORT DE SAE: R4.02: Qualité de Dev

Introduction:

Lors de notre implémentation, nous avons évidemment été amené à devoir tester de nombreuses fonctions afin d'assurer la pertinence des algorithmes ainsi que la validé des données reçue depuis la/les API utilisées.

Dans ce Rapport, nous allons mettre en l'oeuvre l'explication de deux fonctions: une en Back-End et une en Front-End.

Nous avons évidemment testé le maximum de fonctions possibles (se trouvant dans les DAOs, les models, les routes...)

Contexte:

Afin de comprendre l'enjeux et surtout l'utilités des fonctions qui seront expliqués plus tard, voici quelques informations importante qui décrirons paritellement notre site Web.

- Un site Web avec une carte qui affiche des toilettes et des stations de Vélos.
- Les toilettes ont **Des Avis (note sur 5 étoiles)**. et les vélos proviennent d'une base de données très régulièrement mise à jour qui indique les vélos restants dans une station de vélos ainsi que sa capacité d'accueil.
- Les toilettes ont donc une **Moyenne** qui rassemble l'avis de tous les utilisateurs ayant mis un avis sur ce toilette sur notre site.
- Lorsque l'utilisateur partage sa localisation, le site lui permet de trouver le lieu le plus proche de ce dernier, ainsi même que l'iténeraire emprunté.

Pour compléter, voici les models des classes "Bike", "Toilette", "User":

Bike:

```
export default class BikeModel {
        identifiant
        nom
    addresse
    nombreDeVeloDisponibles
```

```
    nombreDeBorneDisponible
    latitude
    longitude

    constructor(obj) {
        Object.assign(this, obj)
    }
}
```

Toilette:

```
export default class ToilettesModel {
    identifiant
    nom
    latitude
    longitude
    ranking=-1
    nbr_avis=0

    constructor(obj) {
        Object.assign(this, obj)
    }
```

User:

```
export default class User {
    login
    password
    avisdonnees
    constructor(obj) {
        this.login = obj.login
        this.password = obj.password
        this.avisdonnees=obj.avisdonnees
    }
}
```

Complément:

Pour pouvoir tester correctement ces deux fonctions, nous avons mis en place un système de testabilité (vu en cours) permettant de tester au mieux nos fonctions:

- Les tests structurels.
- Les tests Fonctionnels.
- les tests de mutations (tester des tests.)

Première Fonction [Méthode]: CalculMoyenne():

La fonction "CalculMoyenne()" est une **méthode** spéficique aux Toilettes. Cette fonction permettera de savoir l'avis global d'un toilette sur 5 (avec des étoiles au même titre que des applications connues d'avis).
Un utilisateur ne peut donner qu'un avis pour un toilette associé.

Voici le code de cette dite fonction:

```
async calculMoyenne(){
        let datauser = await userDAO.findAll()
        let occurence = 0
        let totalnote = 0


        for (let i = 0; i < datauser.length; i++) {
            for (let j = 0; j < datauser[i].avisdonnees.length; j++) {

                if (datauser[i].avisdonnees[j].idToilette ==
this.identifiant) {

                    totalnote += datauser[i].avisdonnees[j].ranking
                    occurence++
                }

            }
        }

        this.ranking=totalnote==0?-1:totalnote/occurence
        this.nbr_avis=occurence
    }
```

Détail de la fonction:

Cette fonction va d'abord aller chercher dans la base de données tous les utilisateurs present dans cette dernière. On va ensuite faire une première boucle qui parcoura tous les users et une seconde qui parcourera le tableau d'avis du User.
Un Avis contient un idtoilette ainsi que la note donnée par l'utilisateur.

On redéfini ensuite l'attribut ranking du toilette par : 0 si aucun avis a été trouvé, totalnote/occurence sinon. le nbr_d'avis est aussi retourné afin de permet en Front-End d'exploiter facilement l'affichage du nombre d'avis de toilette.

Premières Remarques:

- Notre fonction dépend d'un appel asynchrome "userDao.findAll()". Notre Première idée est de pouvoir Réaliser un "mockk" qui permet de copier le DAO du User.
- Finalement nous avons décidé d'exploiter le Serveur en mémoire proposé par Mongoose. Cela nous permet à chaque test de pouvoir insérer les données voulues en partant obligatoirement d'une base de donnée contenant le minimum. C'est à dire tous les toilettes et tous les Bikes, car ce sont des données qui sont récupérées, depuis une source qui assure la cohérence des données.

- La méthode était anciennement une fonction qui se trouvait dans le Dao des Toilettes, mais a été finalement déplacé dans le model pour plus de cohérence.

Tests Fonctionnels

Nous avons d'abord pensé à tort que le findAll() pouvait parfois renvoyer rien , un , ou plusieurs toilettes. Mais ces données sont récupérées d'une base de données

Alors il possède un unique état possible: (une seule plage possible) findAll() reverra toujours Une liste d'objets toilettes. Ce tableau contiendra exactement le même nombre de toilette que celui de la base.

Ensuite l'idToilette est théoriquement forcément exampté d'erreur également car cette fonction est appellé depuis une récupération des toilettes (toilettes.findAll()). Malgré tout, lors des tests nous avons isolés cela afin de tester la fonction en dehors de notre contexte (qui pourrait être changeant.)

Ainsi voici donc une analyse partionelle.

L'analyse partionnelle:

type: idtoilette=integer(32bit)

plage Normale/Exeception nelle:

- [min;0]
- findById(idToilette)==null -findById(idToilette)==Toilette

(La plage choisie est sort un peu des plages habituelles mais un id est considéré comme bon que lorsque qu'il est dans la base de données, donc lorsque la fonction findByiD ne renvoit pas "null")

DT	IdToilette	//	//	//
//	[Min;0]	X		
//	findById(toilette)==null		X	
//	findByid(idToilette)==Toilette			X
ORACLE	//	//	//	//
//	Erreur	X	X	
//	OK			X

Les tests générés

On considère que : User1(toto,pass1,[avis(5,4.2)]) User2(tata,pass2,[avis(5,2.3)])

sont dans la base.

Et que les Id toilettes= 1,2,5 existe dans la base.

CT1(DT1(-6),Erreur)

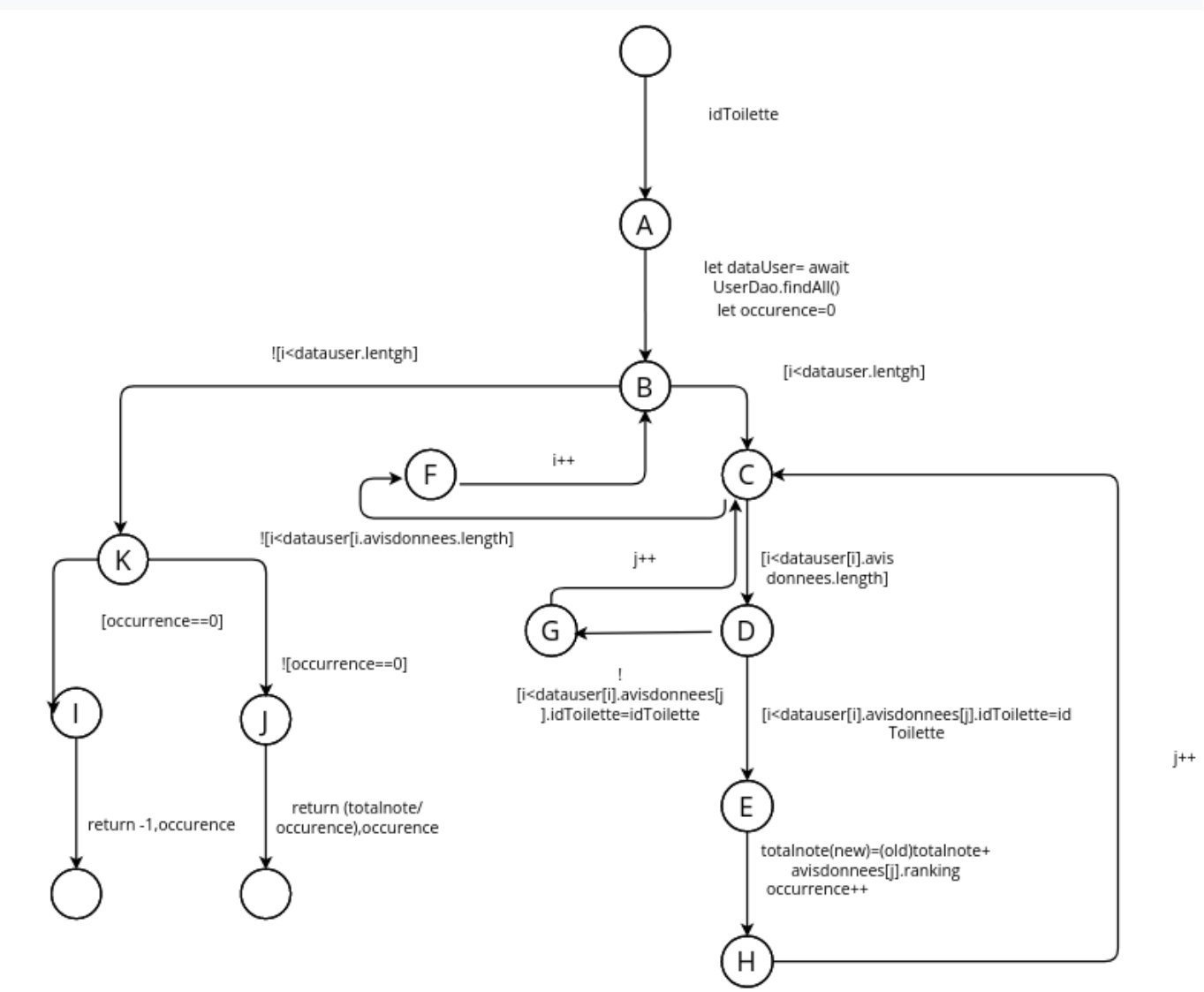
CT2(DT2(3),Erreur)

CT3(DT3(5),2,3.2)

Les tests Fonctionnels sont difficilement suffisant. C'est pourquoi les tests structurels sont ici les plus important pour ce type de fonctions.

Test structurels:

Graphe de flots de contrôles:



Expression:

Expr=[AB((CEB) *)(DGC|DEHC) *) * K (I|J)]

Tous les noeuds:

- ABCDEHCDGCIBKJ,ABKI.

Attention ! On remarque que ABKI est en fait un chemin IMPOSSIBLE, tous simplement car il nécessite que L'occurence soit >0 sauf qu'elle n'est jamais incrémentée dans ce chemin.

Tous les Arcs: Idems.

Tous les 1-Chemins:

- ABCIBKI
- ABCDGCIBKI
- ABCDEHCIBJ
- ABCDEHCIBCDEHCIBKJ
- ABKJ
- ABKJ (impossible)

à Noter qu'il est évidemment impossible de tester pour tous les chemins, car la boucle peut être infinie.

Tests Générés:

(pour des raisons pratique, les users seront considérés comme des données de tests même si dans la vraie vie cela n'est pas vrai. Malgré tout il est nécessaire de redéfinir nos users afin de pouvoir assurer le maximum d'enventualité.)

Finalement les tests **FONCTIONNELS** ne seront pas implémenté, car généré l'erreur lorsque l'idToilette n'existe pas n'est pas utile pour nous et l'envois de l'erreur pourrait nous créer des problèmes.

CT1(DT1(User(a,b,[]),id=12),[-1, 0])

CT2(DT2(User(a,b,[avis(8,4.5)]),id=12),[-1, 0])

CT3(DT3(User(a,b,[avis(12,4.5)]),id=12),[1, 4.5])

CT4(DT4(User1(a,b,[avis(12,4.5)]), User2(c,d,[avis(12,2.5)]), id=12), [2, 3.5])

CT5(DT5([],[-1, 0]))

Vous trouverez les tests implémentés dans le projet Git du groupe dans le chemin suivant: Web/test/DAO_Toilette.mjs

```
  CalculMoyenne()
    ✔ no user
    ✔ one user w/out review (70ms)
    ✔ one user w/ one review [no match] (54ms)
    ✔ one user w/ one review [match] (54ms)
    ✔ two users w/ one review [match] (103ms)

  findAll()
    ✔ findall toilet (48ms)

  findbyID()
null
    ✔ WC introuvable
{
  identifiant: 2,
  nom: 'Jean_pierre',
  latitude: -1,
  longitude: -1,
  ranking: -1,
  nbr_avis: 0
}
    ✔ WC trouvable


  8 passing (581ms)
```

Tests de mutations.

Pour pouvoir tester les tests, nous avons eu, pendant les cours, la possibilité d'utiliser pitest. Malheureusement, pitest n'est pas pris en charge en Javascript. Nous avons donc décider de utiliser "Stryker" un module téléchargeable avec npm. Il est considéré comme un détecteur de mutants.

Malgré tout, lorsque nous avons testé pour la première fois nos tests, il s'avère que le score de 100% a été imédiatemment atteint. le code a été totalement recouvert et les mutants tous tués.

Seconde Fonction : findNearestToilet():

Cette fonction permettera de situer le toilette le plus proche des coordonnées de l'utilisateurs.

Détail de la fonction:

Cette fonction va d'abord faire une requête afin de trouver tous les toilettes. on va ensuite parcours tous les toilettes pour trouver le plus proche des coordonnées de l'utilisateur, grâce à la fonction distance() qui calcule pour nous la distance en km entre les deux coordonnées.

la fonction distance :

```
const distance = (lat1, lon1, lat2, lon2) => {
    const R = 6371; // Rayon de la Terre en kilomètres.
    const dLat = (lat2 - lat1) * (Math.PI / 180); // Différence de latitude
convertie en radians.
    const dLon = (lon2 - lon1) * (Math.PI / 180); // Différence de
longitude convertie en radians.
    const a =
        Math.sin(dLat / 2) * Math.sin(dLat / 2) +
        Math.cos(lat1 * (Math.PI / 180)) * Math.cos(lat2 * (Math.PI / 180))
*
        Math.sin(dLon / 2) * Math.sin(dLon / 2); // Calcul de la formule de
la distance haversine.
    const c = 2 * Math.atan2(Math.sqrt(a), Math.sqrt(1 - a)); // Calcul de
l'angle central.
    const d = R * c; // Distance calculée en multipliant l'angle central
par le rayon de la Terre.
    return d; // Retourne la distance en kilomètres.
}
```

On renvoit le toilette correspondant.

Premières Remarques:

- Notre fonction est testé dans la partie Front-end de l'application. En effet, les données de l'utilistateurs étant récupéré depuis cette dernière ne rends pas vraiment possible la migration de cette fonction dans le back-end (compliqué inutile).

Tests Fonctionnels

Ainsi voici donc une analyse partionelle.

L'analyse partionnelle:

lat,long= Float

plages:

lat= [Min;-90[,[-90;90],]90,max] long= [Min;-180[,[-180;180],]180,max]

DT	lat	//	//	//		
//	[Min;-90]	X			-	-
//	[-90;90]		X		-	-
//	[90;max]			X	-	-
//	Long	//	//	//	//	//
//	[Min;-180]	-		-	X	
//	[-180;180]	-	X	-		
//	[180;max]	-		-		X
ORACLE	//	//	//	//	//	//
//	ArithmeticException	X		X	X	X
//	OK		X			

Les tests générés

donné du toilette calculé: lat=38.89,long=-77.032

CT1(DT1(-90,45,),ArithmeticExeception)

CT2(DT2(40.76,-73.984),333.2071)

CT3(DT3(110,90),ArithmeticExeception)

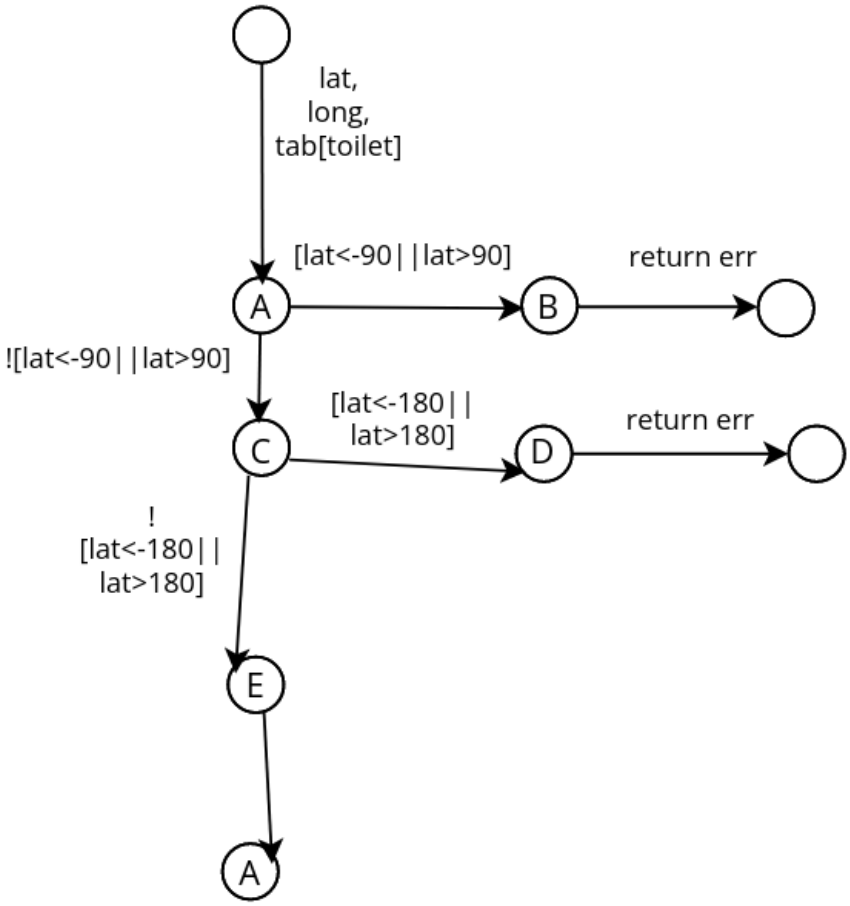
CT4(DT4(-50,-181),ArithmeticExeception)

CT5(DT4(-50,182),ArithmeticExeception)

(Pour économiser des tests, on décide de prendre des valeurs où cela posse problème pour la latitude mais pas pour la longitude, exemple: lat=110)

Test structurels:

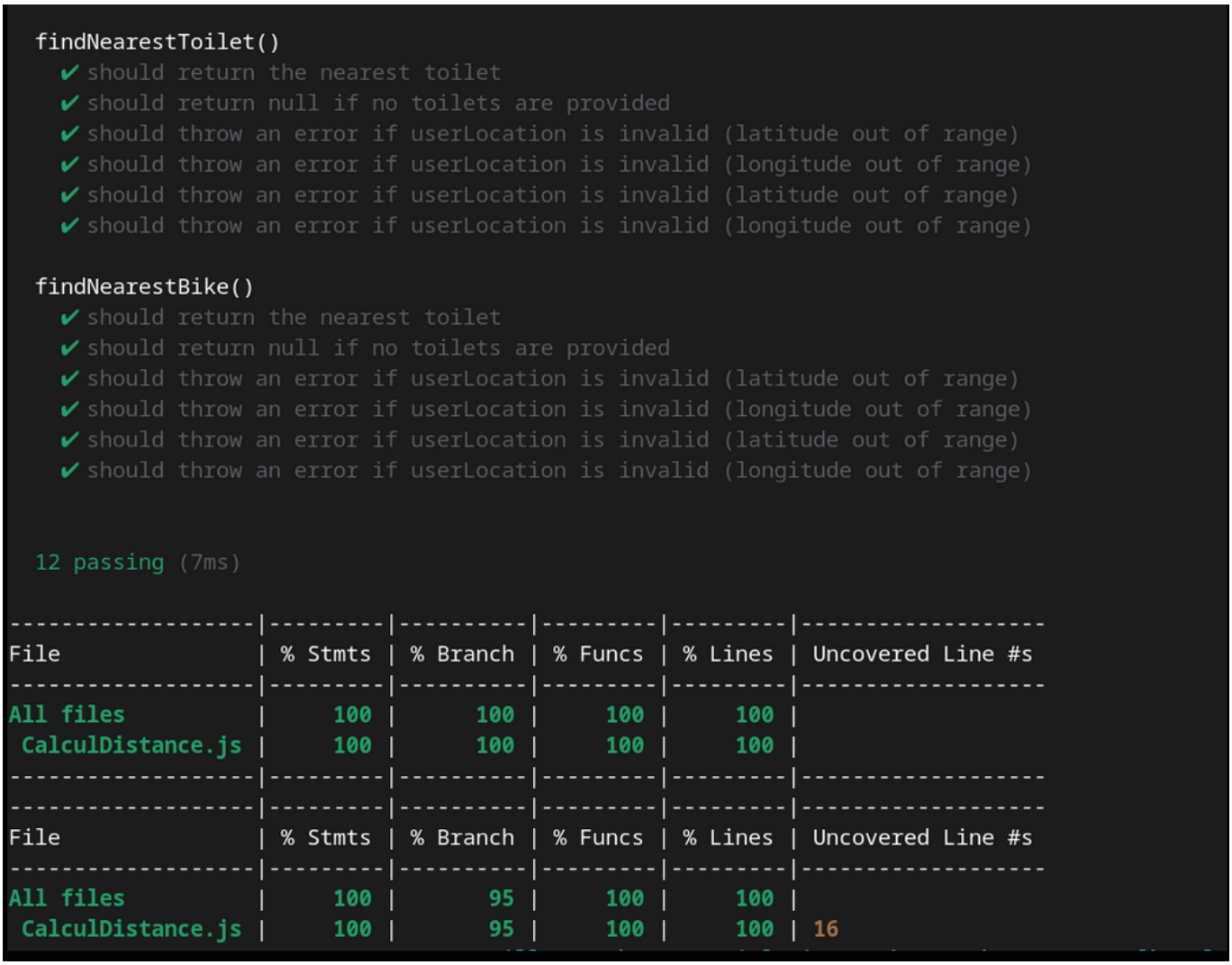
Graphe de flots de contrôles:



Par manque de temps, nous n'avons pas eu le temps de finir ce graphe.

Par conséquent, nous n'avons utilisé que les Tests fonctionnels. (qui se sont averés finalement largement suffisants)

Tests Générés:



Vous trouverez les tests implémentés dans le projet Git du groupe dans le chemin suivant: my-app/test/