

RAPPORT DE SAE: R4.02: Qualité de Dev

Introduction:

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

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

Bien-sûr, nous avons testé le maximum de fonctions possibles (se trouvant dans les DAOs, les models, les routes...), bien qu'elles ne soient pas aussi poussées que les deux qui suivront.

Contexte:

Afin de comprendre l'enjeu et surtout l'utilité des fonctions qui seront expliquées plus tard, voici quelques informations importantes qui décrivent partiellement 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'itinéraire emprunté.

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

Bike:

```
export default class BikeModel { identifiant  
  
  nom  
  
  adresse  
  
  nombreDeVeloDisponibles  
  
  nombreDeBorneDisponible  
  
  latitude  
  
  longitude  
  
  constructor(obj) {  
    Object.assign(this, obj)  
  }  
}
```

Toilette:

```
export default class ToiletttesModel {  
  
  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** spécifique aux Toilettes. Cette fonction permettra 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 une 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 présent 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 redéfinit 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 asynchrone "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. (Qui était en réalité une erreur car un appel de dao ne devrait pas être appelé dans le model !)

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 exempté d'erreur également car cette fonction est appelé 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/Execeptionnelle:

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

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

<u>DT</u>	<u>IdToilette</u>	//	//	//
//	[Min;0]	X		
//	<u>findById(toilette)==null</u>		X	
//	<u>findById(idToilette)==Toilette</u>			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)

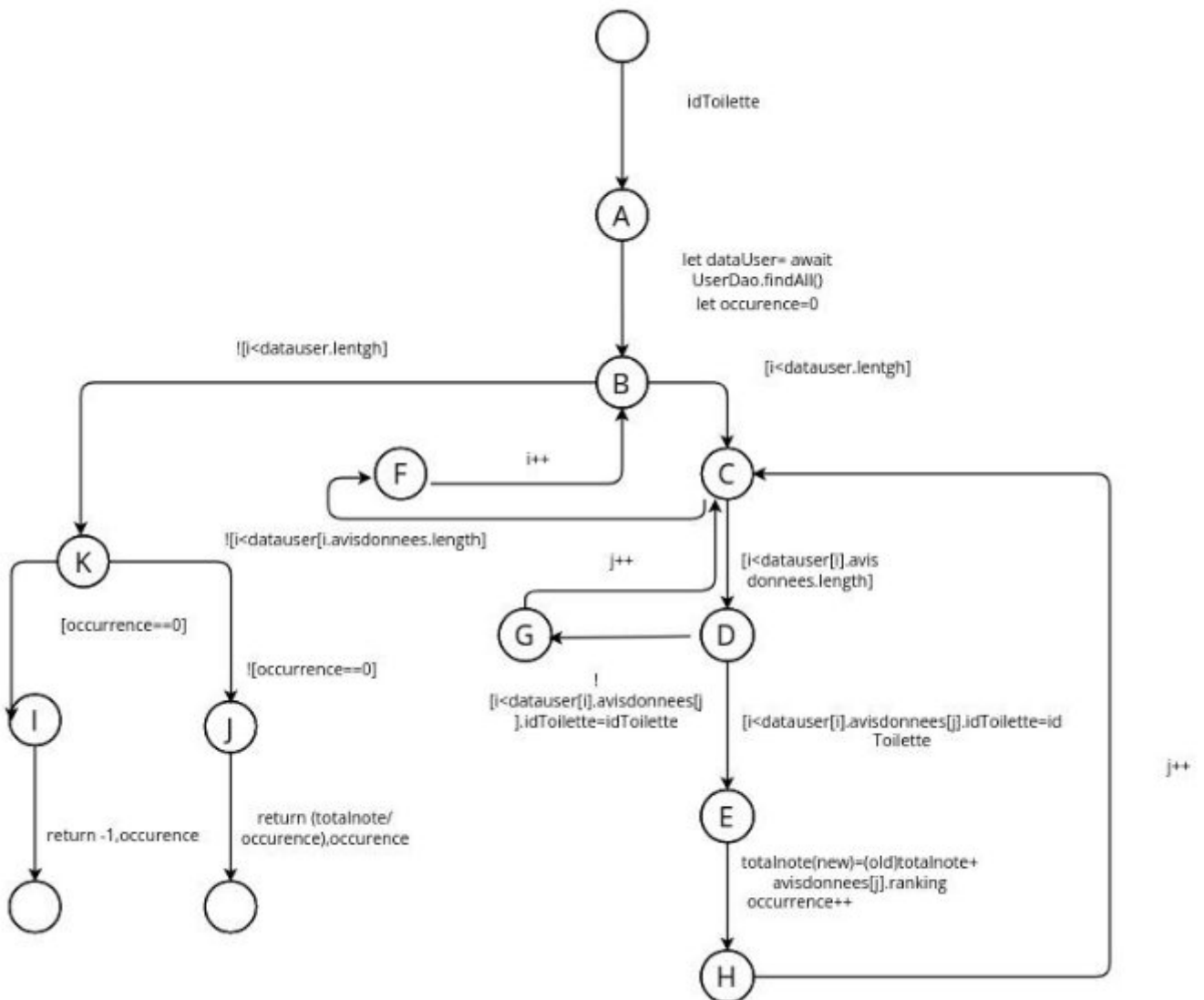
CT2(DT2(3),Erreur)

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 (|J)]

Tous les noeuds:

ABCDEHCDGCIBKJ,ABKI.

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

Tous les Arcs:

Idems.

Tous les 1-Chemins:

ABCIBKI

ABCDGCIBKI

ABCDEHCIBJ

ABCDEHCIBBCDEHCIBKJ

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 pratiques, les users seront considérés comme des données de tests même si dans la vraie vie cela n'est pas possible.

Finalement les tests **FONCTIONNELS** ne seront pas implémentés, car générer l'erreur lorsque l'idToilette n'existe pas n'est pas utile pour nous et l'envoi 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 décidé 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 immédiatement atteint. le code a été totalement recouvert et les mutants tous tués.

Seconde Fonction : findNearestToilet():

Cette fonction permettra de situer la toilette la plus proche des coordonnées de l'utilisateur.

Détail de la fonction:

Cette fonction va d'abord faire une requête afin de trouver toutes les toilettes. On va ensuite parcourir toutes les toilettes pour trouver la 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 renvoie la toilette correspondant.

Premières Remarques:

Notre fonction est testée dans la partie Front-end de l'application. En effet, les données de l'utilisateur étant récupérées depuis cette dernière ne rend pas vraiment possible la migration de

cette fonction dans le back-end (compliqué et inutile).

Tests Fonctionnels

Ainsi voici donc une analyse partionelle.

L'analyse partionnelle:

lat,long= Float

pages:

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

Explications:

Nous avons trouvé ces limites en se renseignant sur le protocole pour appliquer des coordonnées à un endroit précis de la terre. La latitude se trouve forcément entre -90 et 90 degrés (En réalité 90 degré Nord et 90 degré Sud) et -180 et 180 degrés pour la longitude (En réalité 180 Ouest, 180 Est).

<u>DT</u>	lat	//	//	//		
//	[Min;-90]	X			-	-
//	[-90;90]		X		-	-
//	[90;max]			X	-	-
//	Long	//	//	//	//	//
//	[Min;-180]	-		-	X	
//	[-180;180]	-	X	-		
//	[180;max]	-		-		X
ORACLE	//	//	//	//	//	//
//	<u>ArithmeticException</u>	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)

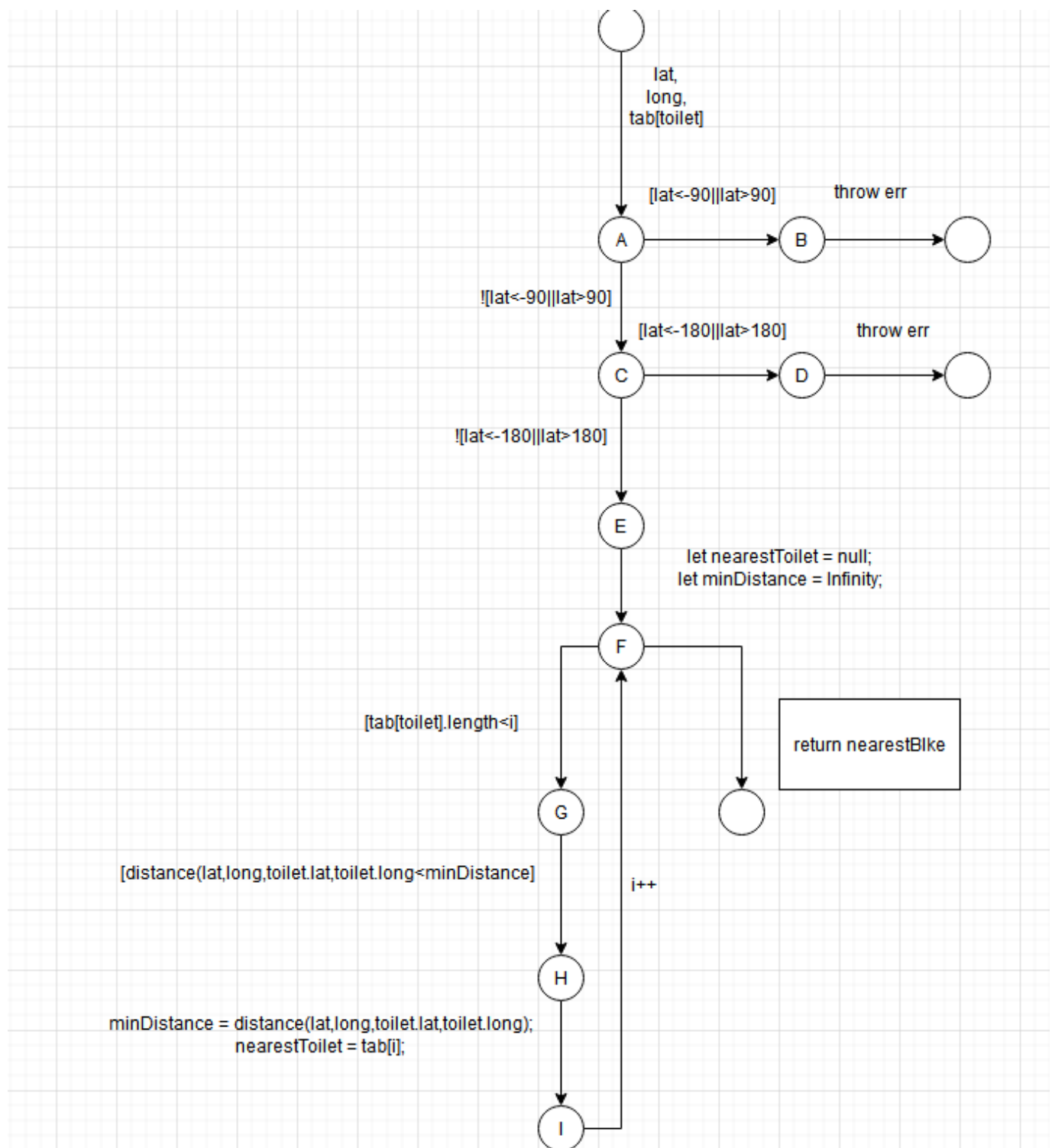
CT4(DT4(-50,-181),ArithmeticExeception)

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

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

Test structurels:

Graphe de flots de contrôles:



Expression: A (B|C (D|EF(GHIF)*)))

Tous les noeuds:

- AB
- ACD
- ACEFGHIF

Tous les chemins:

Idem

Tous les 1-chemins:

- AB
- ACD
- ACEFGHIF
- ACEF

("Tous les chemins "étant impossible dû à la présence d'une boucle)

Tests Générés:

Les tests aux limites étant déjà fait également pour le test d'une toilette trouvée et d'un e toilette non trouvée, il ne reste donc plus aucun test à implémenter.

Précision sur le test toilette non trouvée:

Tester si la fonction renvoie bien null si elle ne passe pas dans la boucle (Chemin ACEF). Malgré tout, le tableau de toilettes ne SERA JAMAIS VIDE (car on récupère de tout façon les données avant toute chose, même dans un serveur mongodb en mémoire).

```
findNearestToilet()
```

- ✓ should return the nearest toilet
- ✓ should return null if no toilets are provided
- ✓ should throw an error if userLocation is invalid (latitude out of range)
- ✓ should throw an error if userLocation is invalid (longitude out of range)
- ✓ should throw an error if userLocation is invalid (latitude out of range)
- ✓ should throw an error if userLocation is invalid (longitude out of range)

```
findNearestBike()
```

- ✓ should return the nearest toilet
- ✓ should return null if no toilets are provided
- ✓ should throw an error if userLocation is invalid (latitude out of range)
- ✓ should throw an error if userLocation is invalid (longitude out of range)
- ✓ should throw an error if userLocation is invalid (latitude out of range)
- ✓ should throw an error if userLocation is invalid (longitude out of range)

```
12 passing (7ms)
```

File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	100	100	100	100	
CalculDistance.js	100	100	100	100	
File	% Stmts	% Branch	% Funcs	% Lines	Uncovered Line #s
All files	100	95	100	100	
CalculDistance.js	100	95	100	100	16

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

Tests de mutations:

Cette fois-ci quelques mutants ont été trouvés.

```

const findNearestToilet = (userLocation, toilets) => {
  if (userLocation[0] < -90 || userLocation[0] > 90 || userLocation[1] < -180 || userLocation[1] > 180){
    throw Error("userLocation Invalid")
  }
  let nearestToilet = null; // Initialisation de la toilette la plus proche.
  let minDistance = Infinity; // Initialisation de la distance minimale avec une valeur infinie.

  // Parcours de la liste des toilettes.
  toilets.forEach(toilet => {
    // Calcul de la distance entre la position de l'utilisateur et la toilette actuelle.
    const d = distance(userLocation[0], userLocation[1], toilet.latitude, toilet.longitude);
    // Vérification si la distance calculée est inférieure à la distance minimale.
    if (d < minDistance) {
      minDistance = d; // Mise à jour de la distance minimale.
      nearestToilet = toilet; // Mise à jour de la toilette la plus proche.
    }
  });

  return nearestToilet; // Retourne les informations de la toilette la plus proche ou null.
}

```

Les mutants sont situés sur les opérateurs, qui sont systématiquement remplacés par des \leq ou des \geq . Il faudrait donc faire des tests aux limites (Valeurs d'essais 90,-90,180,-180).

De plus, dans notre cas, nous avons besoin que les valeurs -90,90,-180,180 soient attribuable. Hors ce n'est pas le cas.

Il faudrait au total donc ajouter au moins 2 tests pour les limites de la latitude et la longitude (deux valeurs peuvent êtres testées en même temps).

Ainsi qu'un test où l'on vérifiera si $d = \text{minDistance}$.

Ce dernier test peut être option car il n'aura aucun effet si d prend une valeur égale à lui même.

Bilan:

Les tests sont un aspect très important de la programmation, ils permettent d'assurer un code robuste, de neutraliser le code mort ainsi que des variables mal gérées sur leurs limites.

Les types de tests sont multiples et chacun en complète un autre.

Et malgré tout cela, nous avons tout de même utilisé une extension permettant de juger la qualité de nos tests qui après nos analyses théoriques, manquaient encore de couverture !