SAE4 - Qualité de développement

Membres du groupe

Maël ANGOT
Nolan BLOURDE
Amédéo CALCAGNI
Quentin CHAUVELON
Arthur OSSELIN

Présentation du projet

Dans le cadre de la SAE4, nous souhaitions créer un emploi du temps amélioré, qui permettrait non seulement de consulter son emploi du temps, mais aussi de pouvoir trouver une salle libre à une heure donnée, de savoir où trouver un professeur un jour donné ou encore de pouvoir synchorniser une API de direction (Google Maps) avec notre emploi du temps pour savoir comment se rendre à un certains cours et à quelle heure partir.
Nous avons donc réalisé 3 applications en lien avec ce projet :

- Une API
- Une application web (utilisant l'API)
- Une application mobile (utilisant l'API)

Execution des tests

Pour lancer les tests, il vous faut récupérer le dossier `/API` disponible sur notre dépôt gitlab.
Une fois cela fait, modifiez les variables `port` et `host` dans la configuration du serveur dans le fichier

server.mjs.

```
const server = Hapi.server({
    port: 443 // Modifiez cette variable pour indiquer le port (ex : 3030),
    host: '172.26.82.56' // Modifiez cette variable pour indiquer l'hôte (ex : localho
    // Reste du code
});
```

Après avoir configuré, le serveur, il ne vous reste plus qu'à installer les modules en exécutant la commande :

```
npm install
```

Une fois que tous les modules sont installés, vous pouvez lancer les tests en exécutant l'une des deux commandes suivantes :

```
npm test
```

ou :

```
node_modules/.bin/c8 node_modules/.bin/nyc node_modules/.bin/mocha --timeout 10000 --
```

Vous obtiendrez, après quelques secondes, un rapport des tests qui viennent d'être exécutés.

Tests

Dans tout projet informatique, le test est une partie essentielle permettant de s'assurer du bon développement de l'application. C'est pourquoi, nous avons vigoureusement testé toutes nos applications. Cependant, pour l'application web et l'application mobile, nous n'avons pas pu coder de tests par manque de temps mais aussi car ce sont des applications plutôt visuelles pour lesquelles toutes les données étaient fournies par l'API, il s'agissait donc d'un simple affichage.

Nous avons effectué des tests structurels (en boîte blanche), c'est à dire en écrivant d'abord le code, puis en concevant les tests nécessaires pour vérifier ce code.

Notre API était constitué de routes indépendantes pour lesquels nous devions nous assurer que pour une certaine entrée, nous obtenions bien le résultat escompté. Ainsi, nous pouvions facilement tester chaque route indépendamment des autres (sans problèmes de dépendances) et s'assurer, avec des tests assez complets, de la non-régression tout au long du développement.

Nous avons suivi un cycle de développement itératif, c'est à dire en découpant le projet en petite parties, qui sont testées, puis en ajoutant des fonctionnalités au fur et à mesure (de manière itérative).

Exemple d'itérations des tests :

Pour montrer l'évolution du code et des tests en deux itérations, nous allons nous intéresser à la route `/populate/{type}` dans le fichier `server.mjs`. Cette route permet de peupler la base de données à partir

d'un fichier qui correspond au type donné en paramètre (teachers, rooms ou groups). Cependant, à l'origine, cette route ne peuplait que la base de données la salle (les groupes ou professeurs étant arrivés plus tard dans le développement). Le code de la route ressemblait donc à cela lors de la 1ère itération :

```
{
    method: 'GET',
    path: '/populate/rooms',
    handler: async (request, h) => {
        try {
            const rooms = await controller.populate("rooms")

            if (rooms != null) {
                    return h.response(rooms).code(200)
                } else {
                    return h.response({message: 'not found'}).code(404)
                }
        } catch (e) {
            return h.response(e).code(400)
        }
    }
},
```

qui appelait la méthode `controller.populate` qui ressemblait à ça :

```
populate : async(fileName) => {
    try {
        switch (fileName) {
            case "rooms":
                return await roomDao.populate("./data/rooms.csv")

            default:
                return h.response({message: 'not found'}).code(404);
        }

    } catch (e) {
        return Promise.reject({message : "error"})
    }
},
```

Nous avons donc conçu les tests de cette route :

```
describe('/populate/rooms', () => {
    let server;

    beforeEach(async () => {
        server = await init();
    });
```

```
    afterEach(async () => {
        await server.stop();
    });


    it('must show rooms', async () => {

        const res = await server.inject({
            method: 'get',
            url: '/populate/rooms'
        });
        chai.expect(res.statusCode).to.equal(200);
        chai.expect(res.result).to.be.eql([
            {
                "id": 1299,
                "name": "C0/01",
                "computerRoom": true
            },
            // [...] Tableau de retour simplifié car trop long (renvoie toutes les sal
            {
                "id": 89808,
                "name": "C1/14",
                "computerRoom": true
            }
        ])
    });

    it("must show 404 - not found", async () => {
        const res = await server.inject({
            method: 'get',
            url: '/populate/fdsjfkdqlms'
        });
        chai.expect(res.statusCode).to.equal(404);
        chai.expect(res.result).to.be.eql({message: 'not found'})
    });
});
```

Puis lors de la 2ème itération, nous avons ajouté la possibilité d'appeler la route avec une autre valeur que `room` (`groups` et `teachers`). Nous avons donc transformé le code de la route ainsi :

```
{
    method: 'GET',
    path: '/populate/{type}',
    handler: async (request, h) => {
        try {
            const classes = await controller.populate(request.params.type)

            if (classes != null) {
```

```
                return h.response(classes).code(200)
            } else {
                return h.response({message: 'not found'}).code(404)
            }
        } catch (e) {
            return h.response(e).code(400)
        }
    }
},
```

et `controller.populate` comme ceci :

```
populate : async(fileName) => {
    try {
        switch (fileName) {
            case "groups":
                return await groupDao.populate("./data/groups.csv")

            case "rooms":
                return await roomDao.populate("./data/rooms.csv")

            case "teachers":
            return await teacherDao.populate("./data/teachers.csv")

            default:
                return null;
        }

    } catch (e) {
        return Promise.reject({message : "error"})
    }
},
```

Nous avons fait évolué nos tests pour couvrir les nouvelles fonctionnalités que nous venions d'implémenter tout en nous assurant que `populate/rooms` fonctionne toujours :

```
describe('/populate/{type}', () => {
    let server;

    beforeEach(async () => {
        server = await init();
    });

    afterEach(async () => {
        await server.stop();
    });
```

```
    it('must show rooms', async () => {

        const res = await server.inject({
            method: 'get',
            url: '/populate/rooms'
        });
        chai.expect(res.statusCode).to.equal(200);
        chai.expect(res.result).to.be.eql([
            {
                "id": 1299,
                "name": "C0/01",
                "computerRoom": true
            },
            // [...] Tableau de retour simplifié car trop long (renvoie toutes les sal
            {
                "id": 89808,
                "name": "C1/14",
                "computerRoom": true
            }
        ])
    });

    it("must show groups", async () => {
        const res = await server.inject({
            method: 'get',
            url: '/populate/groups'
        });
        chai.expect(res.statusCode).to.equal(200);
        chai.expect(res.result).to.be.eql([
            {
                "id": 3163,
                "name": "INFO 1 TP 1-1"
            },
            // [...] Tableau de retour simplifié car trop long (renvoie toutes les gro
            {
                "id": 3192,
                "name": "INFO 2 TP 4-2"
            }
        ])
    });

    it('must show teachers', async () => {

        const res = await server.inject({
            method: 'get',
            url: '/populate/teachers'
        });
        chai.expect(res.statusCode).to.equal(200);
        chai.expect(res.result).to.be.eql([
            {
```

```
                "id": 2232,
                "name": "Jean-Marie MOTTU"
            },
            // [...] Tableau de retour simplifié car trop long (renvoie toutes les pro
            {
                "id": 928742,
                "name": "Ali BENJILANY"
            }
        ])
    });

    it("must show 404 - not found", async () => {
        const res = await server.inject({
            method: 'get',
            url: '/populate/fdsjfkdqlms'
        });
        chai.expect(res.statusCode).to.equal(404);
        chai.expect(res.result).to.be.eql({message: 'not found'})
    });
});
```

Enfin, lors d'une 3ème itération, nous avons ajouté la validation des entrées et des sorties en utilisant Joi (par exemple, nous pouvions préciser à la route que l'arguement `type` qu'elle attendait était de type `string`).

Nous avons donc modifié notre `server.mjs` comme suit :

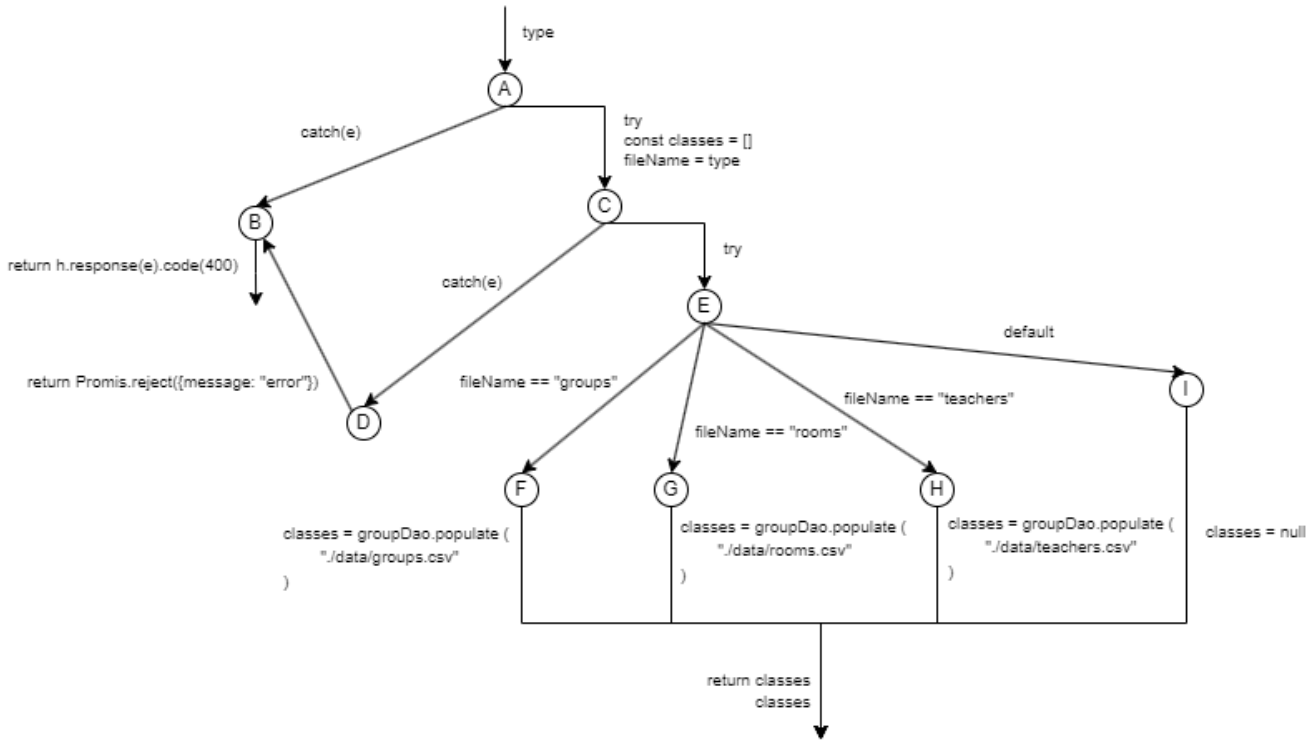
```
{
    method: 'GET',
    path: '/populate/{type}',
    options : {
        description : "Lit le csv corresopndant au type donné et peuple la base de dor
        notes : "Lit le csv corresopndant au type donné et peuple la base de données",
        tags : ['api'],
        validate: {
            params: Joi.object({
                type : Joi.string().description("Le type de données à peupler (teacher
            }),
        },
        response: {
            status: {
                200 : Joi.array().items(Joi.object()).description("Une liste de JoiRoc
                404 : notFound,
                400 : Joi.object()
            }
        }
    },
    handler: async (request, h) => {
        try {
            const classes = await controller.populate(request.params.type)
```

```
            if (classes != null) {
                return h.response(classes).code(200)
            } else {
                return h.response({message: 'not found'}).code(404)
            }
        } catch (e) {
            return h.response(e).code(400)
        }
    }
},
```

Après avoir ajouté cette validation, nos tests n'ont pas vraiment changé, nous avons surtout dû nous assurer de la non-régression du code.
Nous n'avons pas eu besoin d'ajouter de tests puisque la route attendait un `string` en entrée et même si un autre type était envoyé (int, bool), il aurait quand même pu être interprété comme un `string` et n'aurait donc pas causé de problème (la route aurait simplement renvoyé `{message : "not found"}` comme dans le dernier test présenté au-dessus).

Conception des tests

Pour s'assurer que la couverture des tests pour la route `/populate/{type}` est maximale, nous pouvons commencer par réaliser un graphe de flot de contrôle :



A noter que dans le graphe ci-dessus, nous n'avons pas développer le code de groupDao.populate(...) après les noeuds F, G H et I, car celui-ci fait simplement de la lecture de fichier csv et de l'ajout dans une base de données. Il n'aurait donc pas ajouté de nouvelles branches au graphe (autre que des try catch), et n'aurait donc pas ajouté de cas de test. Par conséquent, nous avons préféré ne pas le rapporter entièrement afin de ne pas rendre le graphe plus illisible.

On peut voir dans le graphe ci-dessus que les différentes routes à tester sont `ACEF`, `ACEG`, `ACEH` et `ACEI`. Quant à `AB` et `ACDB`, ce sont des chemins qui ne pourront être exécutés que dans le cas où il n'y aurait une erreur imprévue, ce que l'on ne peut donc pas tester.

Nos routes n'étant pas très complexes et n'ayant pas beaucoup de chemins différents (assez peu de conditions et de boucles, mais principalement de l'appel de méthodes d'autres fichiers et de base de données), nous n'avons pas réalisé d'analyse de mutation.

Résultat des tests

Lors de l'exécution des tests en utilisant les commandes mentionnée dans la section Execution des tests. nous obtenons un rapport nous indiquant quels tests sont passés et lesquels ont échoués :

```
default route
✓ expected default route  (77ms)

/populate/{type}
✓ must show 404 - not found
/populate/rooms
    ✓ must show rooms (3304ms)
/populate/groups
    ✓ must show groups (754ms)
/populate/teachers
    ✓ must show teachers (5942ms)

room/{id}/{time}
✓ should show room class of C0-05 at 30/03/2023 at 8:30 am and it has no class (2017m
✓ should show room class of C0-04 at 30/03/2023 at  15:20 and it has class at this ti
✓ should show error because the date is not well formated
✓ should show not found because the id does not exist
✓ should show empty because the date is negative
✓ should show empty because the date is well formated but is recognized as 0230/03/30

/rooms/{computerRoomOnly}/{time}
✓ should show all empty non-computer rooms on the given date (4570ms)
✓ should show all empty computer rooms on the given date (142ms)
✓ should show empty because there are no empty rooms (132ms)
✓ should show error because the date is not well formated

/schedule
/schedule/day/{id}/{date}
    ✓ should show 404 because the group does not exist
    ✓ should show date invalide
    ✓ should show empty table for the schedule of the group  (369ms)
    ✓ should show the schedule of the day for the group
/schedule/week/{id}/{date}
    ✓ should show 404 because the group does not exist (40ms)
    ✓ should show empty tables for the schedule of the group
    ✓ should show date invalide
```

```
    ✓ should show the schedule of the week for the group

/teacher/{id}/{date}
✓ should show classes of Mr.Berdjugin at 30/03/2023 at 8:30 am (499ms)
✓ should show classes of Mr.Berdjugin at 30/03/2023 at 8:30 am and he has no classes
✓ should show error because the date is not well formated
✓ should show not found because the id does not exist

/groups
✓ must show groups (373ms)

/teachers
✓ must show teachers (743ms)

/user
/user/register
    ✓ add user ok (166ms)
    ✓ add existing user
    ✓ wrong payload
/user/login
    ✓ user login (278ms)
    ✓ user not found
    ✓ wrong payload
/user/favoriteSchedule
    ✓ user favoriteSchedule (148ms)
    ✓ user not found  (62ms)
    ✓ wrong payload
/user/favoriteSchedule/{token}
    ✓ user favoriteSchedule (180ms)
    ✓ user not found  (43ms)
/user/favoriteAddress
    ✓ user favoriteAddress (215ms)
    ✓ user not found
    ✓ wrong payload
/user/favoriteAddress/{token}
    ✓ user favoriteAddress (92ms)
    ✓ user not found
/user/favoriteTransitMode
    ✓ user favoriteTransitMode (94ms)
    ✓ user not found
    ✓ wrong payload
/user/favoriteTransitMode/{token}
    ✓ user favoriteTransitMode (87ms)
    ✓ user not found

/directions/{origin}/{arrivalTime}/{transitMode?}
    ✓ get directions
```

```
51 passing (21s)
0 failing
```

Dans le cas où un test ne passe pas, on obtient le détail de pourquoi il a échoué :

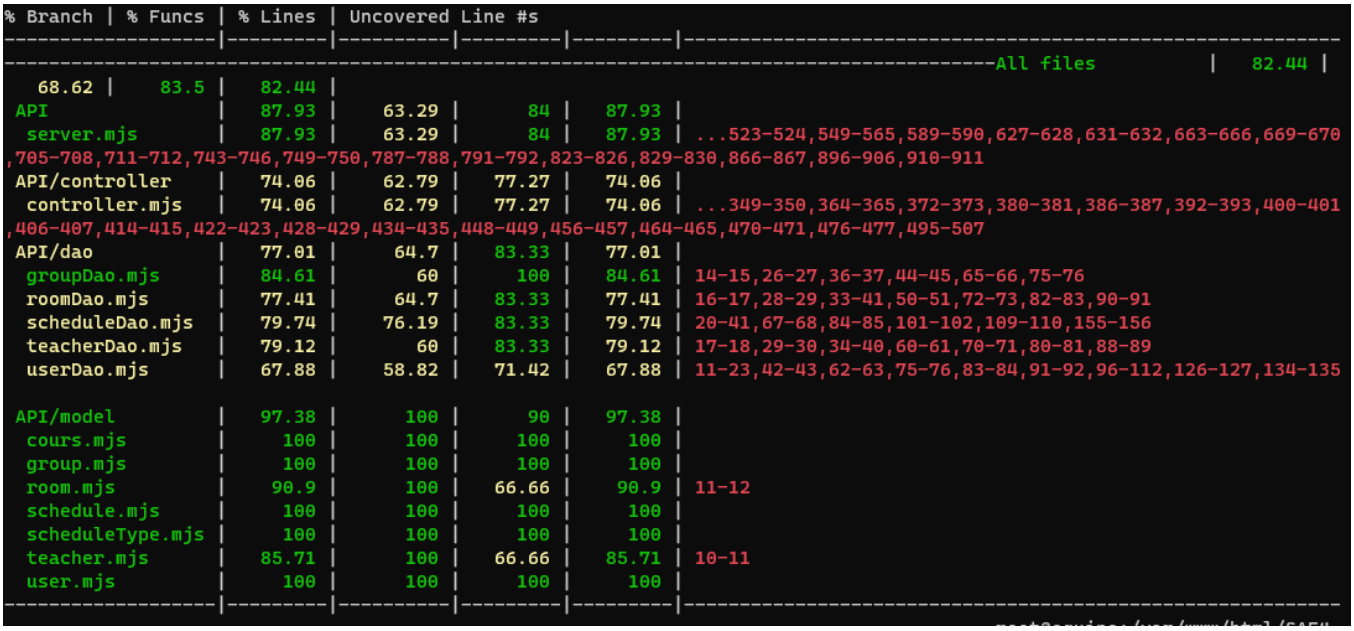
```
3) /schedule
    /schedule/week/{id}/{date}
        should show empty tab and the schedule of the room :

    AssertionError: expected 400 to equal 200
    + expected - actual

    -400
    +200

    at Context.<anonymous> (file:///var/www/html/SAE4/API/test/test-chai.mjs:453:44)
    at process.processTicksAndRejections (node:internal/process/task_queues:95:5)
```

De plus, la commande affiche aussi un tableau de couverture du code par les tests :



% Branch	% Funcs	% Lines	Uncovered Line #s
68.62	83.5	82.44	All files
API	87.93	63.29	84	87.93	
server.mjs	87.93	63.29	84	87.93	...523-524,549-565,589-590,627-628,631-632,663-666,669-670,705-708,711-712,743-746,749-750,787-788,791-792,823-826,829-830,866-867,896-906,910-911
API/controller	74.06	62.79	77.27	74.06	
controller.mjs	74.06	62.79	77.27	74.06	...349-350,364-365,372-373,380-381,386-387,392-393,400-401,406-407,414-415,422-423,428-429,434-435,448-449,456-457,464-465,470-471,476-477,495-507
API/dao	77.01	64.7	83.33	77.01	
groupDao.mjs	84.61	60	100	84.61	14-15,26-27,36-37,44-45,65-66,75-76
roomDao.mjs	77.41	64.7	83.33	77.41	16-17,28-29,33-41,50-51,72-73,82-83,90-91
scheduleDao.mjs	79.74	76.19	83.33	79.74	20-41,67-68,84-85,101-102,109-110,155-156
teacherDao.mjs	79.12	60	83.33	79.12	17-18,29-30,34-40,60-61,70-71,80-81,88-89
userDao.mjs	67.88	58.82	71.42	67.88	11-23,42-43,62-63,75-76,83-84,91-92,96-112,126-127,134-135
API/model	97.38	100	90	97.38	
cours.mjs	100	100	100	100	
group.mjs	100	100	100	100	
room.mjs	90.9	100	66.66	90.9	11-12
schedule.mjs	100	100	100	100	
scheduleType.mjs	100	100	100	100	
teacher.mjs	85.71	100	66.66	85.71	10-11
user.mjs	100	100	100	100	

On peut donc ici voir que la couverture de nos tests est assez bonne et que la plupart des lignes par lesquels ne sont pas passés les tests sont la documentation des routes avec Swagger :

```
options : {
    description : 'Supprime un utilisateur',
    notes : 'Supprime un utilisateur en utilisant le token donné',
    tags : ['api'],
    validate: {
        params: Joi.object({
            token : Joi.string().description("Le token lié au compte (obtenu lors de l
        }),
    },
    response: {
        status: {
```

```
            200 : joiUserSansMdp.description("Le compte qui vient d'être supprimé (le
            404 : notFound,
            400 : Joi.object()
        }
    }
},
```

et les catch (d'où le fait que les lignes où les tests ne passent pas dans les fichiers autre que `server.mjs` sont des blocs de 2 lignes (ex : 11-12, 26-27...)) :

```
} catch(e) {
    return Promise.reject(e)
}
```

Correction du code

Les tests nous ont permis de repérer de nombreux bugs, allant de bugs majeurs nuisant au bon fonctionnement de l'application à des bugs mineurs (principalement des erreurs sur des codes de retour (ex : 404 au lieu de 400)). Nous allons pas lister tous les problèmes que nous avons pu régler, mais nous allons vous en présenter deux :

Le premier problème que nous ont permis de régler les tests était la validation de l'id pour récupérer un emploi du temps. En effet, nous avons une route (/schedule/day/{id}/{date}) qui permet de récupérer l'emploi du temps de l'id donné. Cependant, si j'indiquais un id qui n'existait, au lieu d'obtenir un `{message : "not found"}`, j'obtenais un tableau de cours vide. Ce n'était pas un problème qui nous empêchait d'utiliser l'API, mais cela faisait moins professionnel. La méthode `findByDay` du fichier `controller.mjs` est donc passé de :

```
findByDay : async(id, date, scheduleType) => {
        try {

            clearDatabaseIfNotUpdatedToday()

            let schedule = await scheduleDao.find(id);
            if (schedule == null) {
                schedule = await scheduleDao.save(id, scheduleType)
            }

            if (schedule == null) {
                return null
            }

            return await scheduleDao.findByDay(schedule, date)

        } catch (e) {
            return Promise.reject({message : "error"})
```

```
        }
    },
```

à :

```
findBy Day : async(id, date, scheduleType) => {
        try {

            clearDatabaseIfNotUpdatedToday()

            // [ Ajout (on vérifie que l'id est bien celui d'un professeur ou d'un gro
            const teacher = await teacherDao.find(id);
            const group = await groupDao.find(id);
            if (teacher == null && group == null) {
                return null;
            }
            // ]

            let schedule = await scheduleDao.find(id);
            if (schedule == null) {
                schedule = await scheduleDao.save(id, scheduleType)
            }

            if (schedule == null) {
                return null
            }

            return await scheduleDao.findByDay(schedule, date)

        } catch (e) {
            return Promise.reject({message : "error"})
        }
    },
```

qui a pu être corrigé grâce au test suivant :

```
it('should show 404 because the group does not exist', async () => {
    const res = await server.inject({
        method: 'get',
        url: '/schedule/day/0000/20230330T114200000Z'
    });
    chai.expect(res.statusCode).to.equal(404);
    chai.expect(res.result).to.be.eql({message: "not found"})
});
```

Le 2ème bug que nous avons pu corrigé était cette fois beaucoup plus critique et a la majorité de nos routes.

Ce bug est apparu lorsque nous avons ajouté la validation de objets avec Joi (comme mentionné un peu plus haut dans ce rapport) où nous avions défini les dates comme des string. Cependant, pour ces routes, la date peut être omise, auquel cas, c'est la date du jour qui est prise, ce qui faisait qu'un chaîne de caractères vide était donc passé comme paramètre de la route. Le problème étant que Joi ne considère pas une chaîne de caractères vide comme un string et nous renvoyait donc une erreur 400 : BAD PARAMS.
Le code est donc passé de :

```
date : Joi.string().description("La date ou vide (par défaut, la date du jour est uti
```

à :

```
date : Joi.string().allow(null, "").description("La date ou vide (par défaut, la date
```

qui a pu être corrigé grâce à tous les tests que nous avions déjà conçus.

Amélioration

Notre API dépendant en majorité de données pouvant changer, que ce soit les emplois du temps (qui peuvent être modifiés à tout moment) ou le calcul de trajet (qui peut être altéré en cas de travaux, bouchons...). Il aurait été intéressant, avec un peu plus de temps, d'implémenter des stubs (doublures).

Conclusion

Etant donné que l'application web et l'application mobile était tous deux basés sur l'API, il était indispensable que celle-ci soit fiable. Les tests nous ont donc permis de nous assurer de sa fiabilité et de pouvoir l'utiliser avec une plus grande confiance.