

# PROJET API

Architecture des systèmes  
d'informations

THIRIOT Anaïs

M2 ACSI

2018/2019

# Table des matières

Introduction .....	1
1. Analyse des besoins .....	2
2. Conception.....	3
3. Étapes d'implémentation et de réalisation.....	5
3.1. Service Tâche simple.....	5
3.1.1. Architecture.....	5
3.1.2. Méthodes REST.....	6
3.2. Service Participant simple.....	11
3.2.1. Architecture.....	11
3.2.2. Méthodes REST et tests .....	11
3.3. Mise en place d'un serveur Eureka.....	12
3.3.1. Architecture.....	12
3.3.2. Serveur Eureka .....	12
3.3.3. Clients .....	12
3.4. Mise en place d'attributs communiquant .....	13
3.5. Mise en place d'une communication entre les services .....	13
3.5.1. Préparation de la communication du service Participant .....	13
3.5.2. Communication du service Tache.....	15
3.6. Aspect sécurité .....	17
3.7. Hateoas .....	18
3.8. Tests automatiques .....	18
4. Architecture et implémentation finale.....	19
4.1. Requêtes disponibles .....	20
4.2. Utilisation et tests unitaires des requêtes .....	21
4.3. Implémentation des requêtes.....	29

## INTRODUCTION

---

Ce projet consiste en à la mise en place d'une API répondant au sujet suivant :

« Une tâche est définie par les informations suivantes :

- *id de la tâche (généré par l'API, en respectant UUID)*
- *nom de la tâche*
- *nom du responsable de la tâche*
- *participante de la tâche*
- *date de début (celle de la création)*
- *date d'échéance de la tâche*
- *état courant de la tâche (créée, en cours, achevée, archivée)*

Le processus est le suivant :

- *un utilisateur peut créer une tâche. Si la tâche est créée avec un participant, alors elle est immédiatement dans l'état en cours. Sinon, si une tâche est créée sans participant, elle est dans l'état créé, et elle basculera dans l'état en cours à partir du moment où au moins un participant sera ajouté.*
- *Un utilisateur peut ajouter/retirer des participants à la tâche, quand il le souhaite, à condition que la tâche ne soit pas achevée.*
- *Un utilisateur peut modifier l'état de la tâche (état achevé), par exemple si celle-ci se termine avant la date d'échéance.*

*On vous demande également de traiter les aspects sécurité. Lorsqu'une personne dépose une tâche (inutile de se connecter), elle reçoit un token unique, qu'elle doit ensuite transmettre à chaque requête qu'elle enverra pour avoir des informations sur sa tâche. »*

## 1. ANALYSE DES BESOINS

---

Compte tenu du sujet transmis, l'objectif est de représenter différentes tâches, ainsi que les participants affectés à cette tâche. Il s'agit donc de représenter ces tâches ainsi que leurs caractéristiques, mais également leurs participants. Il nous intéresse également de connaître l'état actuel de la tâche.

Il a été validé comme quoi un participant ne peut être affecté qu'à une unique tâche, ainsi qu'un responsable pouvait être géré à partir d'une chaîne de caractère, et non pas une entité permettant de gérer une personne.

Voici différentes fonctionnalités demandées :

- Un utilisateur doit avoir la possibilité de créer une tâche, qui sera dans l'état « créée ».
- Un utilisateur peut ajouter des participants à une tâche et celle-ci sera dans un état « en-cours », mais aucun ajout ne pourra être effectué si la tâche est achevée.
- Un utilisateur peut également retirer des participants d'une tâche. Il a été décidé comme quoi un utilisateur ne pouvait pas retirer le dernier participant d'une tâche pour ne pas que celle-ci se retrouve sans aucun participant alors que son état est « en-cours », le tout en s'assurant que la tâche n'est pas achevée.
- Un utilisateur a la possibilité de modifier une tâche de façon à l'achever avant sa date d'échéance

Concernant la sécurité, aucune notion de connexion ne doit être mise en place, mais un simple passage de token d'identification de la tâche, afin de s'assurer que l'accès à la tâche est autorisé.

## 2. CONCEPTION

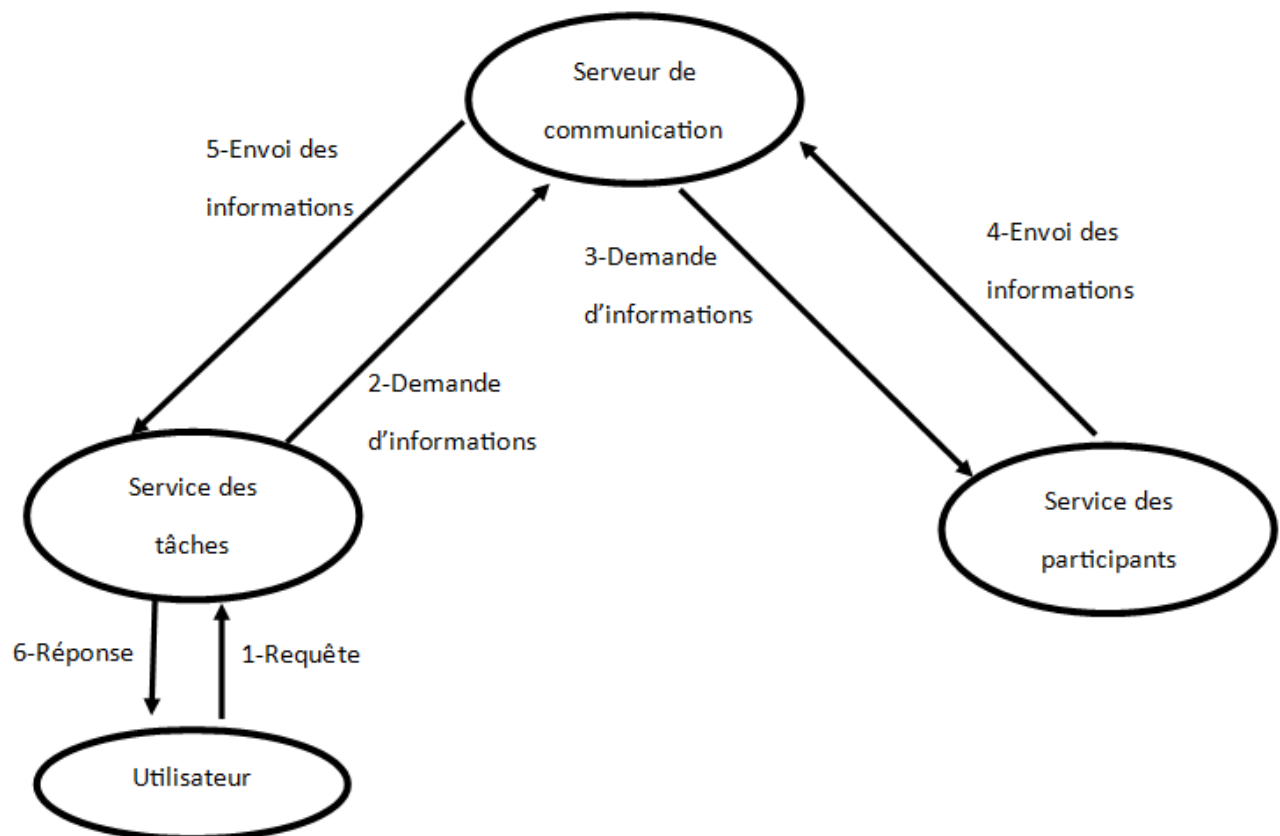
---

Concernant la conception, il a été choisi de mettre en place une architecture micro service permettant de faire communiquer deux services :

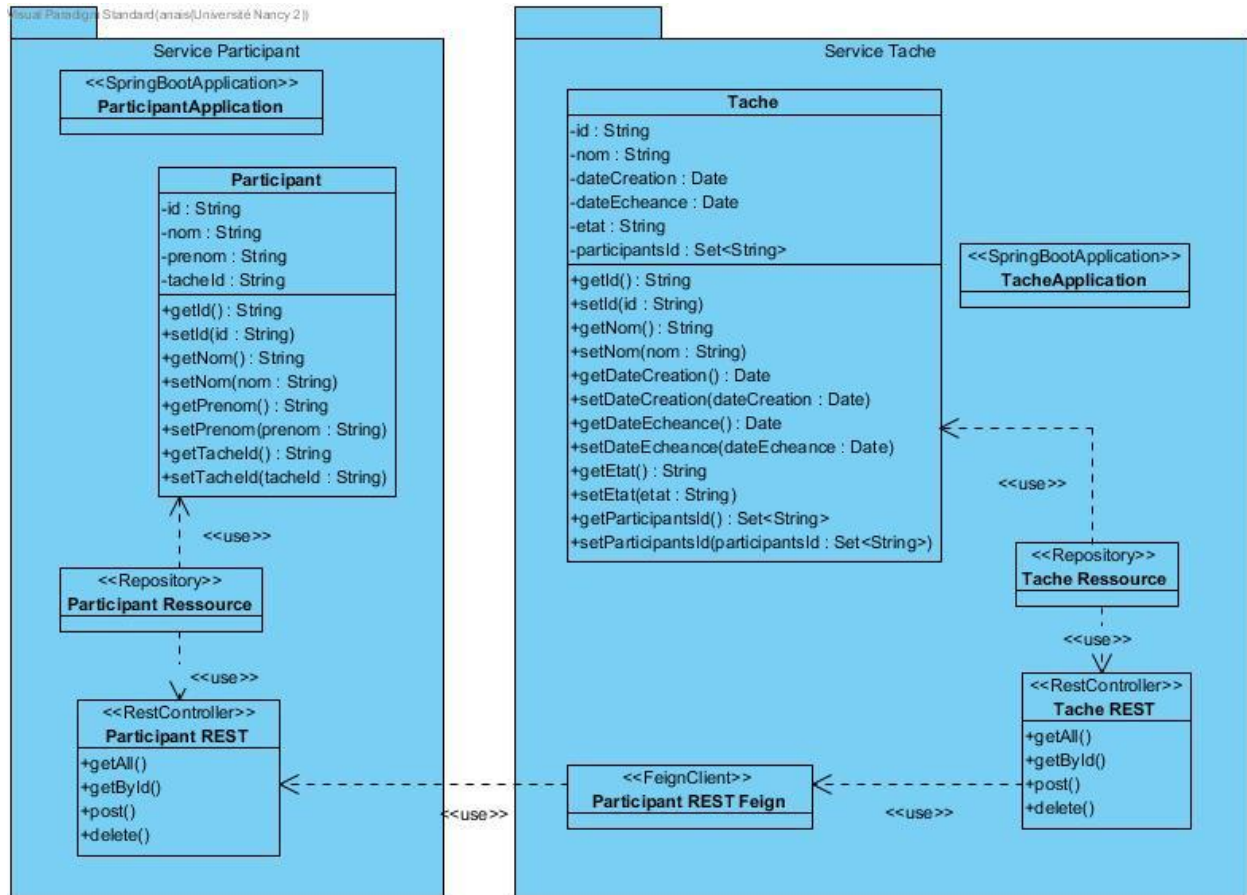
- Un service gérant les tâches
- Un service gérant les participants

L'objectif final est que le service des tâches puisse récupérer les participants nécessaires dans le service des participants.

Voici une représentation de la dynamique attendue dans cette architecture micro service.



Voici également une ébauche de diagramme de classe sur cette architecture :



### 3. ÉTAPES D'IMPLÉMENTATION ET DE RÉALISATION

---

Pour réaliser ce projet, j'ai choisi d'utiliser une méthode itérative, afin de produire une suite d'éléments fonctionnels. J'ai préféré réaliser le projet de façon incrémentale afin de le réduire à des petites étapes simples et fonctionnelles, plutôt qu'un ensemble complexe ; mais également pour montrer les réelles étapes de mise en œuvre.

#### 3.1. Service Tâche simple

J'ai tout d'abord décidé de m'intéresser uniquement aux tâches, ainsi que les attributs correspondants, mais sous forme d'attributs de type String. Cela permet de réaliser les différentes fonctions de requête HTTP sur un objet simple, sans se préoccuper de la complexité des types.

##### 3.1.1. Architecture

Voici l'architecture qui a été mise en place pour gérer les tâches sous forme d'API :

- Classe **Tache - Entity** : Cette classe permet de représenter un objet de type Tache. Voici la structure principale de cette classe :

```
public class Tache {  
    private String id;  
    private String nomTache ;  
    private String nomResponsable;  
    private String participants;  
    private String dateCreation;  
    private String dateEcheance;  
    private String etat;  
    public Tache(String nomTache, String nomResponsable, String participants, String dateCreation,  
        String dateEcheance, String etat);  
}
```

On y retrouve également les différents accesseurs associés.

- Classe **TacheRepresentation – RestController** : Cette classe permet cette fois-ci de créer un objet utilisable par une API RestFulWeb. Cette classe permet ainsi d'implémenter toutes les méthodes utilisables sur l'API RestFul créée.
- Classe **TacheRessource - Repository** : Cette classe permet de créer des points de terminaison RESTFul sur la classe Tache, et peut être utilisée pour modifier par la suite des détails d'exportations sur les objets : par exemple, en effectuant des requêtes personnalisées sur les objets.
- Classe **ProjetAPIApplication** : cette classe permet de définir une application de type Spring de façon automatique grâce à l'annotation @SpringBootApplication.

### 3.1.2. Méthodes REST

#### Méthodes Post et get

Les premières méthodes que j'ai choisi d'implanter sont les méthodes post et get, qui sont essentielles au fonctionnement courant d'une API. Voici donc les fonctions mises en places

voici la méthode permettant de récupérer l'ensemble des tâches :

```
@GetMapping
public ResponseEntity< ? > getAllTaches(){
    Iterable<Tache> allTaches = tr.findAll() ;
    return new ResponseEntity<>(tacheToResource(allTaches), HttpStatus.OK);
}
```

Et la méthode permettant de récupérer une tâche selon son id :

```
@GetMapping(value =("/{tacheId}")
public ResponseEntity<?> getTache(@PathVariable("tacheId") String id) {
    return Optional.ofNullable(tr.findById(id))
        .filter(Optional::isPresent)
        .map(tache -> new ResponseEntity<>(tacheToResource(tache.get(), true), HttpStatus.OK))
        .orElse(new ResponseEntity<>(HttpStatus.NOT_FOUND));
}
```

Voici également la méthode initiale permettant de sauvegarder une tâche grâce à la requête post.

```
@PostMapping
public ResponseEntity< ? > saveTache(@RequestBody Tache tache) {
    tache.setId(UUID.randomUUID().toString());
    Tache saved = tr.save(tache);
    HttpHeaders responseHeaders = new HttpHeaders();
    responseHeaders.setLocation(linkTo(TacheRepresentation.class).slash(saved.getId()).toUri());
    return new ResponseEntity<>(null, responseHeaders, HttpStatus.CREATED);
}
```

#### Méthode delete

La méthode implémentée par la suite est la méthode de suppression. Cette méthode fonctionne également en utilisant l'id de la tâche.

```
@DeleteMapping(value =("/{tacheId}")
public ResponseEntity<?> deleteIntervenant(@PathVariable("tacheId") String id) {
    Optional<Tache> tache = tr.findById(id);
    if (tache.isPresent()) {
        tr.delete(tache.get());
    }
    return new ResponseEntity<>(HttpStatus.NO_CONTENT);
}
```

#### Méthode put

La dernière méthode implémentée dans cette première étape a été la méthode put : cette méthode consiste à modifier l'intégralité d'une tâche. Voici ici la méthode réalisée.

```
@PutMapping(value =("/{tacheId}")
public ResponseEntity<?> updateInscription(@RequestBody Tache tache, @PathVariable("tacheId")
String id) {
```



```

Optional<Tache> body = Optional.ofNullable( tache);
if (!body.isPresent()) {
    return new ResponseEntity<>(HttpStatus.BAD_REQUEST);
}
if (!tr.existsById(id)) {
    return new ResponseEntity<>(HttpStatus.NOT_FOUND);
}
tache.setId(id);
Tache result = tr.save(tache);
return new ResponseEntity<>(HttpStatus.NO_CONTENT);
}

```

### Tests manuels

Différents tests manuels ont été mis en place afin d'étudier le fonctionnement de l'application

La requête *Get* <http://localhost:8082/taches> permet d'afficher la liste des tâches contenues : le résultat est une liste vide puisqu'aucune donnée n'est ajoutée.

On peut donc ajouter des données, en utilisant la requête *Post* <http://localhost:8082/taches> :

```

{"id": "1", "nomTache": "A", "nomResponsable": "A", "participants": "A", "dateCreation": "A", "dateEcheance": "A", "etat": "A"}
{"id": "2", "nomTache": "B", "nomResponsable": "B", "participants": "B", "dateCreation": "B", "dateEcheance": "B", "etat": "B"}
{"id": "3", "nomTache": "C", "nomResponsable": "C", "participants": "C", "dateCreation": "C", "dateEcheance": "C", "etat": "C"}

```

Le résultat suite à l'ajout de ces 3 données est bien « 201 created ».

On peut ensuite réexécuter la méthode de recherche *Get* <http://localhost:8082/taches>, qui affiche l'ensemble de ces données :

```

Embedded: {
  "taches": [
    {
      "nomTache": "B",
      "nomResponsable": "B",
      "participants": "B",
      "dateCreation": "B",
      "dateEcheance": "B",
      "etat": "B",
      "_links": {
        "self": {
          "href": "http://localhost:8082/taches/acf99406-13ea-41ba-899b-7120d7e63b21"
        }
      }
    },
    {
      "nomTache": "C",
      "nomResponsable": "C",
      "participants": "C",
      "dateCreation": "C",
      "dateEcheance": "C",
      "etat": "C",
      "_links": {
        "self": {
          "href": "http://localhost:8082/taches/94bee835-20bf-4f1b-8361-0e848a6eaeac"
        }
      }
    }
  ]
}

```

```

{
  "nomTache": "A",
  "nomResponsable": "A",
  "participants": "A",
  "dateCreation": "A",
  "dateEcheance": "A",
  "etat": "A",
  "_links": {
    "self": {
      "href": "http://localhost:8082/taches/7a750528-3d42-406c-b494-090c5ccca82e"
    }
  }
}

```

On peut également effectuer une recherche sur une tâche particulière, en utilisant son identifiant, par exemple : <http://localhost:8082/taches/f7bb416f-5fb4-443a-9fee-24eee2501a7c>.

La fonction de suppression a elle aussi testé manuellement :

Les tâches existantes sont tout d'abord visualisées via la requête : *GET* <http://localhost:8082/taches/> afin de s'assurer l'existence d'au moins une tâche. L'identifiant d'une de ces données est utilisé dans une requête de suppression : *DELETE* <http://localhost:8082/taches/39e8dc10-c767-4edc-bf9d-dccbca4616e9>.

Le résultat de cette requête est « 204 No Content ». On peut ensuite vérifier la suppression via une requête de recherche : *GET* <http://localhost:8082/taches/39e8dc10-c767-4edc-bf9d-dccbca4616e9>.

Cette fois-ci, aucun résultat n'est affiché, et le message « 404 Not Found » est affiché puisqu'aucun contenu correspondant n'est trouvé.

Concernant les tests manuels réalisés pour la méthode PUT, voici la démarche mise en place : un id d'une tâche existante est récupérée afin de l'utiliser dans une méthode PUT :

Voici par exemple une tâche existante :

```
{
  "_embedded": {
    "taches": [
      {
        "nomTache": "A",
        "nomResponsable": "A",
        "participants": "A",
        "dateCreation": "A",
        "dateEcheance": "A",
        "etat": "A",
        "_links": {
          "self": {
            "href": "http://localhost:8082/taches/f7bb416f-5fb4-443a-9fee-24eee2501a7c"
          }
        }
      }
    ]
  }
}
```

La requête de modification est exécutée, avec le corps de requête qui suit : *PUT* <http://localhost:8082/taches/f7bb416f-5fb4-443a-9fee-24eee2501a7c>

```
{ "id" : "21", "nomTache": "New", "nomResponsable": "New", "participants": "New", "dateCreation": "New",
  "dateEcheance": "New", "etat": "New" }
```

On visualise ensuite cette même tâche selon son id : *Get* <http://localhost:8082/taches/f7bb416f-5fb4-443a-9fee-24eee2501a7c>

```
{
  "nomTache": "New",
  "nomResponsable": "New",
  "participants": "New",
  "dateCreation": "New",
  "dateEcheance": "New",
  "etat": "New",
  "_links": {
    "self": {
      "href": "http://localhost:8082/taches/f7bb416f-5fb4-443a-9fee-24eee2501a7c"
    },
    "collection": {
      "href": "http://localhost:8082/taches"
    }
  }
}
```

On remarque donc que la donnée a bien évolué avec les changements mis en place.

### Tests automatiques

Des tests automatiques ont également été mis en place afin de faciliter la vérification en cas de modification des différentes implémentations de fonction, ou en cas d'ajouts. Voici un exemple de méthode de test qui a été mis en place :

```
@Test
public void getOneTache() {
    Tache tache = new Tache("TacheA", "MonsieurA", "ParticipantA", "dateCreationA",
    "dateEcheanceA", "etatA");
    tache.setId(UUID.randomUUID().toString());
    tr.save(tache);
    ResponseEntity<String> response = restTemplate.getForEntity("/taches/" + tache.getId(),
    String.class);
    assertThat(response.getStatusCode()).isEqualTo(HttpStatus.OK);
    assertThat(response.getBody()).contains("TacheA");
    assertThat(response.getBody()).contains("ParticipantA");
}
```

Cette méthode consiste à ajouter une tâche, pour ensuite vérifier son bon ajout en vérifiant que le résultat de la méthode GET retourne les informations attendues.

D'autres tests ont été implémentés pour la méthode POST, et la version GETALL.

Deux tests automatiques ont été mis en place concernant la suppression ou absence de données :

Cette première méthode permet de tester la suppression d'une tâche : on ajoute une tâche, on récupère ensuite son id pour pouvoir la supprimer. On s'assure ensuite que la recherche sur cet id ne retourne aucun résultat.

```
@Test
public void deleteAPI() throws Exception {
    Tache tache = new Tache("TacheA", "MonsieurX", "ParticipantY",
        "dateCreationXXX", "dateEcheanceYYY", "etatAAA");
    tache.setId(UUID.randomUUID().toString());
    tr.save(tache);
    restTemplate.delete("/taches/" + tache.getId());
    ResponseEntity<?> response =
        restTemplate.getForEntity("/taches/" + tache.getId(), String.class);
    assertThat(response.getStatusCode()).isEqualTo(HttpStatus.NOT_FOUND);
}
```

Cette seconde méthode permet de tester le simple cas d'absence de données : on effectue une recherche sur une tâche non présente dans la base pour s'assurer que le résultat donné est bien NOT\_FOUND.

```
@Test
public void notFoundAPI() throws Exception {
    ResponseEntity<String> response = restTemplate.getForEntity("/taches/12", String.class);
    assertThat(response.getStatusCode()).isEqualTo(HttpStatus.NOT_FOUND);
}
```

Un test automatique a également été mis en place afin de faciliter les vérifications en cas de modifications. Voici le test mis en place :

```
@Test
public void putAPI() throws Exception {
    Tache tache = new Tache("TacheA", "MonsieurX", "ParticipantY", "dateCreationXXX",
        "dateEcheanceYYY", "etatAAA");
    tache.setId(UUID.randomUUID().toString());
    tr.save(tache);
    tache.setNomResponsable("nouveauResponsable");
    .setDateEcheance("nouvelleDate");
    HttpHeaders headers = new HttpHeaders();
    headers.setContentType(MediaType.APPLICATION_JSON);
    headers.setAccept(Arrays.asList(MediaType.APPLICATION_JSON));
    HttpEntity<String> entity = new HttpEntity<>(this.toJsonString(tache), headers);

    restTemplate.put("/taches/" + tache.getId(), entity);

    ResponseEntity<String> response = restTemplate.getForEntity("/taches/" + tache.getId(),
        String.class);
}
```

```
assertThat(response.getStatusCode()).isEqualTo(HttpStatus.OK);
assertThat(response.getBody()).contains("nouveauResponsable");

String code = JsonPath.read(response.getBody(), "$.dateEcheance");
assertThat(code).isEqualTo(tache.getDateEcheance());
}
```

On crée ici une première tâche que l'on modifie ensuite pour pouvoir l'ajouter à la base. On effectue ensuite une requête selon l'id pour vérifier que les changements ont bien été pris en compte.

### 3.2. Service Participant simple

Après avoir représenté les tâches dans un service, j'ai choisi de mettre en place un autre service totalement indépendant permettant de représenter les participants.

#### 3.2.1. Architecture

Ce service possède la même architecture que le service tâches :

- Classe **Participant - Entity** : Cette classe permet de représenter un objet de type Participant, avec son id, son nom, et son prénom.
- Classe **ParticipantRepresentation - RestController** : Cette classe permet cette fois-ci de créer un objet utilisable par une API RestFulWeb. Cette classe permet ainsi d'implémenter toutes les méthodes utilisables sur l'API RestFul créée.
- Classe **ParticipantRessource - Repository** : Cette classe permet de créer des points de terminaison RESTFul sur la classe Participant, et peut être utilisée pour modifier par la suite des détails d'exportations sur les objets : par exemple, en effectuant des requêtes personnalisées sur les objets.
- Classe **ProjetAPIApplication** : cette classe permet de définir une application de type Spring de façon automatique grâce à l'annotation `@SpringBootApplication`.

#### 3.2.2. Méthodes REST et tests

Les mêmes méthodes que pour l'application gérant les tâches ont été implémentées :

- Méthode post,
- Méthode get,
- Méthode delete,
- Méthode put,

Toutes accompagnées de tests automatiques et de tests manuels sous la même forme que le service gérant les tâches.

### 3.3. Mise en place d'un serveur Eureka

Suite à l'implémentation des deux services des Taches et des Participants, ceux-ci ne communiquaient pas et n'avaient aucun lien entre eux.

Afin de pouvoir relier les deux services, j'ai choisi de mettre en place un serveur Eureka : ce serveur permet aux différents services connectés de se rechercher et communiquer entre eux. Cette étape ne consiste pas en l'implémentation d'une communication entre les deux services, mais simplement en la mise en place d'un serveur permettant de reconnaître différents services.

#### 3.3.1. Architecture

Pour cette étape d'ajout d'un serveur de communication, voici l'architecture correspondante :

- Un registre de service : Eureka
- Un service REST Client : Tache
- Un service REST Client : Participant

Ainsi, les services Tache et Participant auront la possibilité d'être détectés par le serveur Eureka.

#### 3.3.2. Serveur Eureka

Implémenter un serveur Eureka consiste en l'utilisation de la dépendance `spring-cloud-starter-netflix-eureka-server`, puis en la mise en place d'une application `@SpringBootApplication`, annotée de `@EnableEurekaServer` :

```
@SpringBootApplication
@EnableEurekaServer
public class EurekaServiceApplication {

    public static void main(String[] args) {
        SpringApplication.run(EurekaServiceApplication.class, args);
    }
}
```

Une fois configuré, ce serveur est visualisable à l'adresse : <http://localhost:8761>

#### 3.3.3. Clients

Afin d'être visibles au serveur, les clients doivent être également configurés :

Dans les deux clients Tache et Participant, les dépendances ont dû être complétées par l'ajout de la dépendance `spring-cloud-starter-netflix-eureka-client`

Grâce à cet ajout, les deux services clients sont rendus visibles du serveur Eureka :

Instances currently registered with Eureka			
Application	AMIs	Availability Zones	Status
PROJET-API-PARTICIPANT	n/a (1)	(1)	UP (1) - LAPTOP-MCM4E5DO.lan:projet-api-participant:8080
PROJET-API-TACHE	n/a (1)	(1)	UP (1) - LAPTOP-MCM4E5DO.lan:projet-api-tache:8082

### 3.4. Mise en place d'attributs communiquant

Avant de pouvoir faire communiquer les deux services de tâches et participants, il est indispensable que les instances des deux objets tâches et participants puissent être reliés, et cela par des attributs.

Il a été convenu qu'un participant ne pouvait être affecté qu'à une unique tâche, tandis qu'une tâche pouvait posséder plusieurs participants.

Ainsi, pour faciliter la communication entre les deux entités, une double liaison a été mise en place :

- Un attribut « `tacheId` » de type `String` a été ajouté à l'objet `Participant` du service participant afin de pouvoir faire un lien directement avec la tâche correspondante.
- Un attribut « `participantsId` » de type `Set<String>` a été ajouté à l'objet `Tache` du service des tâches, afin de pouvoir associer une tâche à l'ensemble de ces participants, ou du moins leur id. Il s'agit d'un attribut ensembliste compte tenu de l'association n-aire reliant les tâches aux participants.

Ces attributs ne sont pas essentiels des deux côtés, mais ils ont été mis en place pour faciliter la liaison et la mise en place des requêtes communicantes.

### 3.5. Mise en place d'une communication entre les services

L'étape suivante consiste en relier le service `Tache` avec le service `Participant` de façon à ce que les tâches puissent être associées aux informations des participants correspondants, et non pas uniquement leur id.

L'objectif est de pouvoir afficher les informations des participants au moment de l'affichage des tâches, mais également de réaliser différentes requêtes http comme l'ajout ou la suppression d'un participant à une tâche, via le service de tâche.

#### 3.5.1. Préparation de la communication du service Participant

Avant de pouvoir communiquer du service des tâches au service des participants, une préparation de la communication doit être instaurée dans le service participant. Cela signifie que pour pouvoir faire des requêtes de tâche vers participant, les requêtes doivent être préparées dans participant.

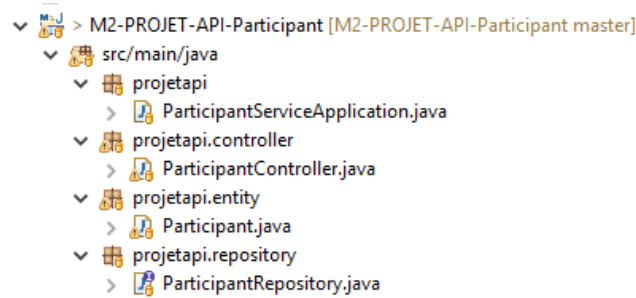
#### Architecture

Avant de mettre en place les différentes méthodes de requêtes, un refactor a été réalisé au niveau des packages et des classes, de façon à se rapprocher d'une architecture MVC, pour distinguer de façon plus précise le rôle de chacune des classes.

Voici la structure ainsi mise en place dans le service Participant :

- Un package **controller** contenant toutes les entités de contrôle du service
  - o Classe **ParticipantController** faisant office de contrôleur pour le service participant, c'est-à-dire office de `RestController`. C'est cette classe qui permet d'exposer la ressource sous forme d'API, en définissant les différentes requêtes de mapping du service.
- Un package **entity** contenant tous les objets métiers du service
  - o Classe **Participant** permettant de représenter l'objet Participant géré par le service.
- Un package **repository** permettant de gérer l'accès aux données
  - o Classe **ParticipantRepository**, permettant d'utiliser l'ensemble des méthodes crud pour la source de données sous-jacente, comme `save`, `findOne`, `findAll` ; mais permettant également de personnaliser des requêtes à partir des attributs des objets et des différents mots-clés fournis par `SpringDataRepository`, qui seront automatiquement interprétés sous forme de requête.

- Une classe **ParticipantServiceApplication** permettant le lancement du service.



## Requêtes

### *ParticipantController*

Comme indiqué précédemment, la classe ParticipantController permet de recenser l'ensemble des requêtes mises à disposition par le service.

On retrouve tout d'abord des requêtes classiques pour le service participant comme :

- getAllParticipants()
- getParticipant(participantId)

Mais d'autres requêtes ont été mises en place pour une communication avec les tâches, par exemple les requêtes suivantes :

- getParticipantsByTache(tacheId), permettant de récupérer tous les participants d'une tâche
- getParticipantByTacheAndId(tacheId, participantId), permettant de récupérer un participant précis d'une tâche précise
- newParticipant(tacheId) permettant d'enregistrer un participant référencer dans le body de la requête comme participant de la tâche donnée

Il s'agit de requête orientée communication avec les tâches.

### *ParticipantRepository*

Mais ces requêtes nécessitent des fonctions du repository non natives. Certaines ont donc dû être ajoutées :

- List<Participant> findByTacheid(String tacheId)
- Optional<Participant> findByTacheidAndId(String tacheId, String id)
- Optional<Participant> findById(String id)

Ainsi, le contrôleur peut accéder à ces fonctions pour les requêtes telles que getParticipantsByTache.



### 3.5.2. Communication du service Tache

#### Architecture

Un refactor a également été réalisé au niveau des packages et des classes, de façon à se rapprocher d'une architecture MVC, pour distinguer de façon plus précise le rôle de chacune des classes.

Voici la structure ainsi mise en place dans le service Tache :

- Un package **controller** contenant toutes les entités de contrôle du service
  - o Classe **TacheController** faisant office de contrôleur pour le service tache, c'est-à-dire une classe RestController permettant d'exposer les ressources du service sous forme d'API. Cette classe va ainsi utiliser les TacheService et ParticipantService où les différentes méthodes seront implémentées, de façon à distinguer la partie REST de la partie usage des services. Cette classe permet par la suite de gérer par exemple les autorisations d'accès, pour les distinguer du traitement sur les entités.
- Un package **entity** contenant tous les objets métiers du service
  - o Classe **Tache** permettant de représenter l'objet Tache géré par le service.
  - o Classe **Participant** permettant de représenter l'objet Participant provenant du service Participant, mais que ce service Tache va utiliser.
- Un package **repository** permettant de gérer l'accès aux données
  - o Classe **TacheRepository**, permettant d'utiliser l'ensemble des méthodes crud sur l'objet tâche, mais permettant également de personnalisé des requêtes.
- Un package **service** pour centraliser les différents services, en distinguant au maximum du contrôleur
  - o Classe **TacheService** regroupant l'ensemble des fonctions propre au service de tâches, par exemple les méthodes pour récupérer une ou les tâches.
  - o Classe **ParticipantService** regroupant l'ensemble des fonctions propre aux traitements des participants, comme l'ajout d'un participant à une tâche. Cette classe va pour cela utiliser la classe ParticipantServiceProxy.
  - o Classe **ParticipantServiceProxy**, classe de connexion à l'API des participants et de gestion des services qui y sont proposés. Il s'agit de façon simpliste d'une copie des signatures des requêtes de mapping du service Participant. C'est cette classe qui relie les tâches aux services.
- Une classe **TacheServiceApplication** permettant le lancement du service.

#### Requêtes

Voici donc certaines requêtes qui sont disponibles dans la classe TacheController, et qui sont donc exposées sous forme d'API.

On retrouve des requêtes simples sur les tâches comme :

- getAllTaches()
- getTache(id)
- saveTache(id)
- deleteTache(id)
- ...

Mais également des requêtes plus complexes pour la communication avec le service Participant par exemple :

- getParticipantsByTache(tacheId) permettant de récupérer tous les participants d'une tâche
- getParticipantByTache(tacheId, participantId) permettant de récupérer un participant particulier d'une tâche
- getTacheByEtat(statut) permettant de récupérer les tâches dans un certain état
- getTacheByResponsable(responsable) permettant de récupérer les participants selon un responsable

- newParticipantTache(tacheId) permettant d'ajouter un participant à une tâche
- deleteParticipantTache(tacheId, tacheParticipant) permettant de supprimer un participant d'une tâche

```
▼ M2-PROJET-API-Tache [M2-PROJET-API-Tache master]
  ▼ src/main/java
    > projetapi
    ▼ projetapi.controller
      > TacheController.java
    ▼ projetapi.entity
      > Participant.java
      > Tache.java
    > projetapi.repository
    ▼ projetapi.service
      > ParticipantServiceProxy.java
      > TacheService.java
```

### 3.6. Aspect sécurité

Il était demandé de traiter les aspects sécurité de l'API : lorsqu'une personne dépose une tâche, un token unique lui est attribué, et doit être transmis pour avoir des informations sur cette requête.

Pour cela, un attribut a été ajouté à l'entité Tache : tokenconnexion. À chaque ajout d'une tâche à la base de données, un token aléatoire lui sera attribué, et réduit en 20 caractères afin de faciliter son utilisation.

```
tache.setTokenConnexion((UUID.randomUUID().toString() +
UUID.randomUUID().toString()).substring(20, 40));
```

Cela permet d'avoir des tokens unique pour chacune des tâches, mais également de taille réduite pour l'utilisation.

Niveau utilisation, pour chaque tâche propre à une tâche, le token sera demandé à l'utilisateur : Ce token devra obligatoirement être indiqué dans le header de la requête, sous la forme suivante :

« key : value »

Avec comme clé « token », et comme valeur le token de la tâche à consulter.

Params	Authorization	Headers (2)	Body	Pre-request Script	Tests	Cookies	Code	Comments (0)
Query Params								
	KEY	VALUE	DESCRIPTION	***	Bulk Edit			
<input checked="" type="checkbox"/>	token	ancfbb5f2a8c4t8a2cf7						
	Key	Value	Description					

Ainsi, cette démarche est mise en place pour l'ensemble des requêtes sur une tâche.

Par contre, aucune sécurité n'a été instaurée pour la méthode récupérant l'ensemble des tâches, ou l'ensemble des tâches dans un état précis, puisque l'utilisateur devrait fournir les tokens de l'ensemble des tâches. Puisqu'aucun système de connexion n'est demandé, il n'a pas été instauré ici de solution pour autoriser.

Il s'agit dans le cadre du projet d'une « faille » puisqu'en affichant l'ensemble des tâches, les tokens sont accessibles à tous. En pratique réelle, il faudrait masquer le champ sur le token pour n'afficher que les informations autorisées ; mais ici, aucune information n'est masquée, puisque pour tester manuellement, il est indispensable de pouvoir visualiser à un moment les différents tokens des tâches disponibles au lancement du service.

Une autre solution aurait été de stocker les tokens d'accès dans un service distinct, pour effectuer des requêtes sur ce service uniquement pour vérifier les accès. Ainsi, en cas d'affichage de l'ensemble des tâches, aucune requête sur ce service de connexion n'aurait été réalisée, et les tâches affichées n'auraient pas disposés de leur token d'accès.

Concernant la vérification de cet accès sécurisé, à chaque demande d'informations ou modification sur une tâche, l'utilisateur devra saisir, comme indiqué, le token dans le header de sa requête, et celui-ci sera étudié.

```

/**
 * Méthode permettant de vérifier si l'accès à une tâche est autorisé
 * @param tacheId
 * @param token
 * @return une autorisation d'accès
 */
public AutorisationAcces verificationAutorisationAcces(String tacheId, String token) {
    Optional<Tache> tacheOptional = tacheRepository.findById(tacheId);
    if (tacheOptional.isPresent()) {
        Tache tache = tacheOptional.get();
        if (tache.getTokenconnexion().equals(token)) {
            return AutorisationAcces.AUTORISE;
        }
        else { return AutorisationAcces.REFUSE; }
    }
    else {
        return AutorisationAcces.INCONNU;
    }
}

```

Cette méthode sera appelée à chaque requête sur une tâche particulière, de façon à s'assurer que la tâche existe, et que le token est valide. Ainsi une autorisation d'accès sera retournée ; chaque autorisation d'accès ayant son propre message et HttpStatus.

```

AUTORISE("Acces autorisé",HttpStatus.OK),
REFUSE("Acces refusé",HttpStatus.FORBIDDEN),
INCONNU("Tache inconnue",HttpStatus.NOT_FOUND);

```

### 3.7. HATEOAS

Il faut noter que dans cette mise en place d'une communication entre services, l'ensemble des ressources disposent d'une référence HATEOAS permettant d'être renvoyé sur chacun des objets.

```

{
  "_embedded": {
    "participants": [
      {
        "nom": "Mansuy",
        "prenom": "Claire",
        "tacheId": "2",
        "_links": {
          "self": {
            "href": "http://localhost:8080/participants/1"
          }
        }
      },
      {
        "nom": "Thiriot",
        "prenom": "Anaïs",
        "tacheId": "1",
        "_links": {
          "self": {
            "href": "http://localhost:8080/participants/2"
          }
        }
      },
      {
        "nom": "Fernandes",
        "prenom": "Charlotte",
        "tacheId": "2",
        "_links": {
          "self": {
            "href": "http://localhost:8080/participants/3"
          }
        }
      }
    ]
  }
},
{
  "nomtache": "Football",
  "nomresponsable": "Loïc",
  "datecreation": "2019-01-01",
  "datecheance": "2020-01-01",
  "etat": "EN COURS",
  "tokenconnexion": "741dopmelanc54faqihn",
  "participants-id": [
    "4",
    "5",
    "6"
  ],
  "_links": {
    "self": {
      "href": "http://localhost:8082/taches/3"
    }
  }
},
{
  "nomtache": "Stage",
  "nomresponsable": "Monsieur Tabonne",
  "datecreation": "2019-04-01",
  "datecheance": "2019-09-06",
  "etat": "CREEE",
  "tokenconnexion": "nci455vna5kl2palc7ka",
  "participants-id": [],
  "_links": {
    "self": {
      "href": "http://localhost:8082/taches/4"
    }
  }
}

```

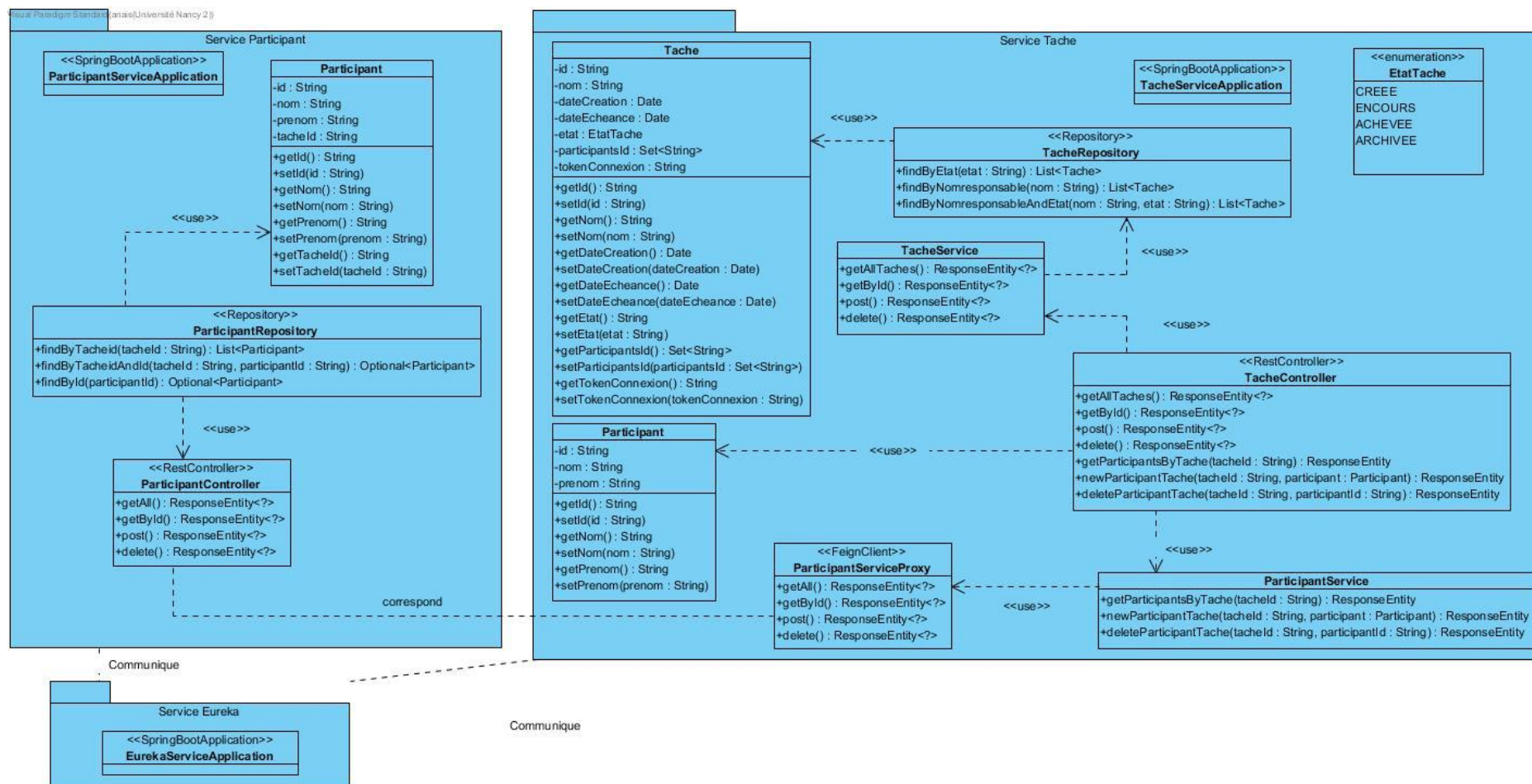
L'ensemble des objets utilisés sont ici accessibles selon leur référence.

### 3.8. Tests automatiques

Différents tests automatiques ont été implémentés dans les deux services gérant les tâches et les participants.

## 4. ARCHITECTURE ET IMPLÉMENTATION FINALE

Voici le diagramme de classe finale de l'implémentation réalisée. L'ensemble des opérations n'ont pas été ajoutées, mais celui-ci a été réalisé afin de comprendre la communication entre les différentes classes des services.



## 4.1. Requêtes disponibles

L'ensemble des requêtes disponibles dans le service Participant ne sont pas listées ici puisque le but n'est pas de les utiliser directement sur l'API Participant, mais plutôt d'être utilisé pour une communication via le service tâches. Voici donc ci-dessous l'ensemble des opérations utilisables sur l'API des tâches :

### *GET tâches*

→ permet de récupérer l'ensemble des tâches et leurs informations. Cette requête ne donne pas d'informations sur les participants, hormis leur id.

### *GET tâches/{tacheId}*

*\* token de connexion à fournir*

→ permet de récupérer une tâche particulière selon son id. Cette fois encore la requête ne donne pas d'informations sur les participants, hormis leur id.

### *POST tâches*

→ permet d'ajouter une tâche au service. La tâche est forcément dans l'état créé, et la date de fin est obligatoirement postérieure à la date de création, soit la date courante.

### *DELETE tâches/{tacheId}*

*\* token de connexion à fournir*

→ permet de clôturer une tâche en cours, ou créée en passant son état à l'état « achevé » ; ou de l'archiver si la tâche est « achevée ».

### *PUT tâches/{tacheId}?dateFin={nouvelleDateFin}*

*\* token de connexion à fournir*

→ cette méthode permet de mettre à jour la date de fin d'une tâche.

*Contrairement à une méthode PUT classique, celle-ci ne nécessite pas de body de mise à jour. Il s'agit d'une adaptation mise en place afin de modifier uniquement la date de fin de la requête. La méthode patch aurait pu être utilisée pour une mise à jour partielle, mais sa mise en œuvre n'a pas été aboutie.*

### *GET tâches?statut = {statut}*

→ permet de rechercher l'ensemble des tâches dans un statut particulier.

### *GET tâches?responsable={responsable}*

→ permet de rechercher l'ensemble des tâches d'un certain responsable.

### *GET tâches/{tacheId}/participants*

*\* token de connexion à fournir*

→ permet de lister l'ensemble des participants d'une tâche

### *GET tâches/{tacheId}/participants/{participantId}*

*\* token de connexion à fournir*

→ permet de récupérer un participant particulier d'une tâche

### *POST tâches/{tacheId}/participants*

*\* token de connexion à fournir*

→ permet d'ajouter un nouveau participant à une tâche, selon le participant fourni dans le body de la requête. Le participant est donc automatiquement ajouté au service participant, et l'id du participant ajouté dans la liste des participants de la tâche.

### *DELETE tâches/{tacheId}/participants/{participantId}*

*\* token de connexion à fournir*

→ permet de supprimer un participant particulier d'une tâche, à condition que la tâche ne soit pas achevée et possède au minimum un participant après suppression. Il s'agit d'un choix de conserver un participant au minimum dans la tâche plutôt que de changer l'état pour « créé », sachant que la tâche a déjà démarré. Cela évite de se retrouver avec des tâches en cours sans aucun participant.

## 4.2. Utilisation et tests unitaires des requêtes

Cette partie vise à présenter les tests unitaires réalisés sur l'ensemble des requêtes, expliquant ainsi de façon plus approfondie l'utilisation des fonctionnalités.

### *GET taches*

Cette requête ne nécessite aucune manipulation de sécurité, et permet d'afficher l'ensemble des tâches disponibles.

```

1 {
2   "_embedded": {
3     "taches": [
4       {
5         "nomtache": "Projet API",
6         "nomresponsable": "Monsieur Perrin",
7         "datecreation": "2018-11-30",
8         "datecheance": "2020-12-31",
9         "etat": "EN COURS",
10        "tokenconnexion": "ancf5f2a8c4t8a2cf7",
11        "participants-id": [
12          "2"
13        ],
14        "_links": {
15          "self": {
16            "href": "http://localhost:8082/taches/1"
17          }
18        }
19      },
20      {
21        "nomtache": "Projet Bibliotheque",
22        "nomresponsable": "Monsieur Benali",
23        "datecreation": "2018-12-14",
24        "datecheance": "2019-03-09",
25        "etat": "ACHEVEE",
26        "tokenconnexion": "a5vr8a1ciypzav5ahcoe",
27        "participants-id": [
28          "1",
29          "3"
30        ]
31      }
32    ]
33  }
34 }

```

Ce test peut être réalisé afin de récupérer le token d'une tâche.

### *GET taches/{tacheId}*

*\* token de connexion à fournir*

Dans le cas de l'accès à une tâche, si le token d'accès est invalide ou non renseigné, l'accès à la tâche ne pourra pas être autorisé :

KEY	VALUE	DESCRIPTION
<input checked="" type="checkbox"/> Content-Type	application/json	
<input checked="" type="checkbox"/> token		
Key	Value	Description

Body Cookies Headers (3) Test Results Status: 403 Forbidden Time: 414 ms Size: 142 B Download

```

1 x 1 Accès refusé

```

De plus, si la tâche n'est pas connue, celle-ci ne peut pas être affichée :



GET http://localhost:8082/taches/12 Send Save

Params Authorization **Headers (2)** Body Pre-request Script Tests Cookies Code Comments (0)

KEY	VALUE	DESCRIPTION
<input checked="" type="checkbox"/> Content-Type	application/json	
<input checked="" type="checkbox"/> token		
Key	Value	Description

Body Cookies Headers (3) Test Results Status: 404 Not Found Time: 194 ms Size: 143 B Download

Pretty Raw Preview JSON ⌵

1 Tâche inconnue

Enfin, si la tâche est bien connue, et que le token est valide, le contenu de la tâche pourra être donné :

GET http://localhost:8082/taches/2 Send Save

Params Authorization **Headers (2)** Body Pre-request Script Tests Cookies Code Comments

KEY	VALUE	DESCRIPTION
<input checked="" type="checkbox"/> Content-Type	application/json	
<input checked="" type="checkbox"/> token	a5vr8a1ciypzav5ahcoe	
Key	Value	Description

Body Cookies Headers (3) Test Results Status: 200 OK Time: 60 ms Size: 542 B Download

Pretty Raw Preview JSON ⌵

```

1 {
2   "nomtache": "Projet Bibliotheque",
3   "nomresponsable": "Monsieur Benali",
4   "datecreation": "2018-12-14",
5   "datecheance": "2019-03-09",
6   "etat": "ACHEVEE",
7   "tokenconnexion": "a5vr8a1ciypzav5ahcoe",
8   "participants-id": [
9     "1",
10    "3"
11  ],
12  "_links": {
13    "self": [
14      {
15        "href": "http://localhost:8082/taches/2"
16      }
17    ]
18  }
19 }

```

### POST taches

De par l'utilisation de cette requête, il est possible d'ajouter une tâche :

POST http://localhost:8082/taches/ Send Save

Params Authorization Headers (2) **Body** Pre-request Script Tests Cookies Code Comments

none form-data x-www-form-urlencoded raw binary JSON (application/json) Beautify

```

1 {\"nomtache\": \"Réalisation des tests\", \"nomresponsable\": \"Anaïs\", \"datecheance\": \"2019-05-01\"}
2

```

Il ne suffit de préciser que ces informations puisque les autres informations sont initialisées par défaut : un id aléatoire et la date de création initialisée à la date courante. En ce qui concerne les participants, il n'est ici pas possible d'ajouter une tâche directement avec ses participants. Une étape supplémentaire d'ajout de participants doit être réalisée. Selon ce principe, l'état de la tâche est initialisé à « créée ».

Voici le résultat retourné par la requête de création :





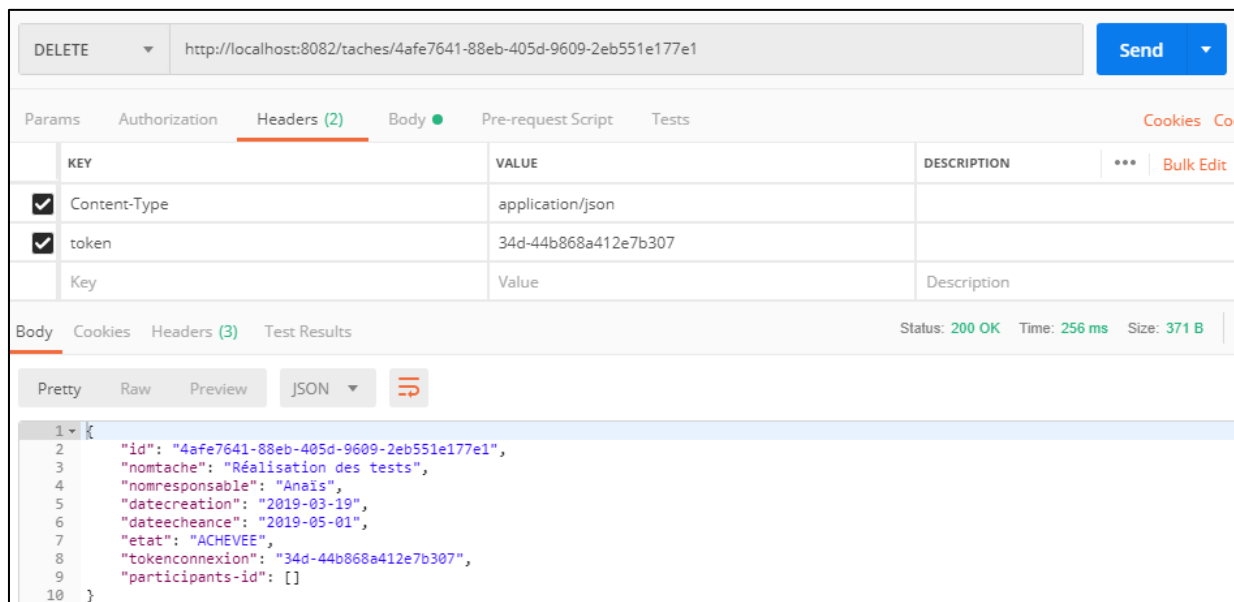
On peut bien visualiser l'ensemble des informations, ainsi que la liste de participants vide, et l'état « créée ».

### *DELETE taches/{tacheId}*

*\* token de connexion à fournir*

De même que pour une recherche de tâche, l'accès n'est autorisé que si la tâche existe, et si le token d'accès est valide.

Si ces conditions sont remplies, il est possible d'achever une tâche via cette requête delete, dans le cas où celle-ci est en cours ou créée. SI la tâche est déjà achevée, elle sera archivée. Si elle est déjà archivée, aucune modification ne sera possible.



Ainsi, il est possible de constater que la tâche a bien été achevée.

`PUT taches/{tacheId}?dateFin={nouvelleDateFin}`

*\* token de connexion à fournir*

Comme indiqué, il n'est pas possible de modifier la date de fin d'une tâche achevée, comme le montre cet exemple, reprenant la même tâche que dans le cas précédent :

The screenshot shows a REST client interface with the following details:

- Method:** PUT
- URL:** `http://localhost:8082/taches/4afe7641-88eb-405d-9609-2eb551e177e1?dateFin=01-04-2019`
- Headers (2):**
  - Content-Type:** application/json
  - token:** 34d-44b868a412e7b307
- Status:** 400 Bad Request
- Time:** 317 ms
- Size:** 192 B
- Message:** Impossible de modifier une tâche achevée

Par ailleurs, si la tâche n'est pas achevée, il est possible de modifier la date d'échéance de la tâche.

The screenshot shows a REST client interface with the following details:

- Method:** PUT
- URL:** `http://localhost:8082/taches/1?dateFin=2019-05-01`
- Headers (2):**
  - Content-Type:** application/json
  - token:** ancfhb5f2a8c4t8a2cf7
- Status:** 200 OK
- Time:** 1101 ms
- Size:** 337
- Body (JSON):**

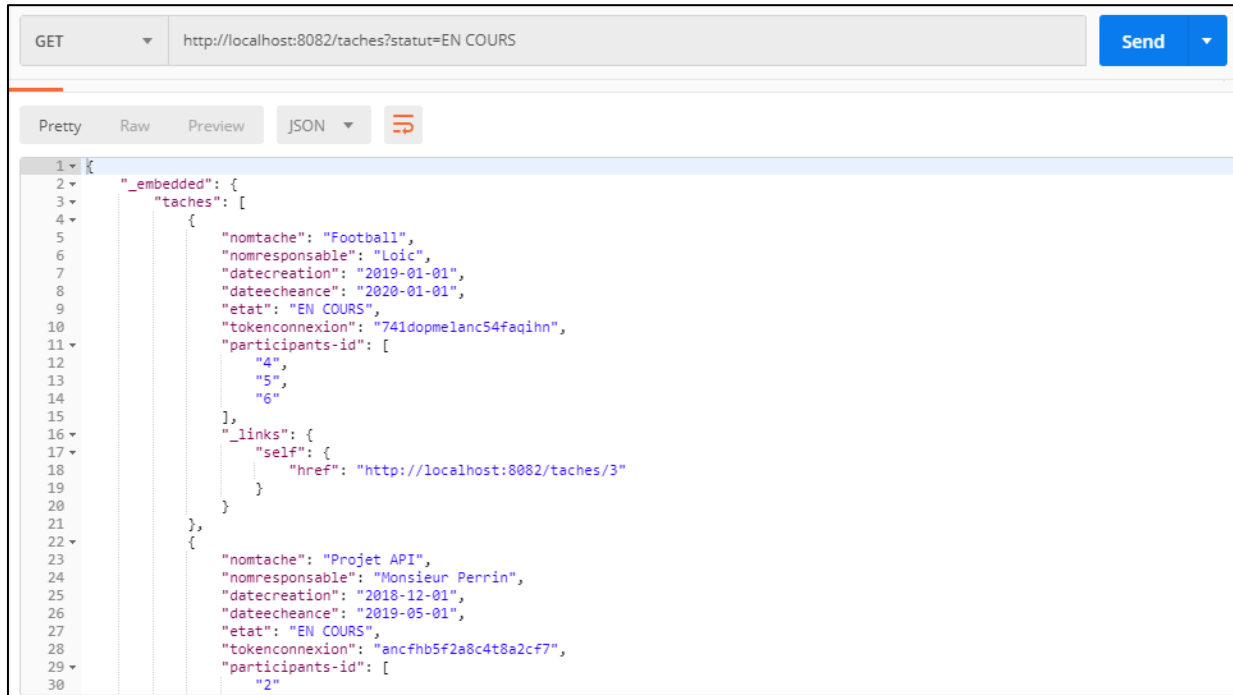
```
{
  "id": "1",
  "nomtache": "Projet API",
  "nomresponsable": "Monsieur Perrin",
  "datecreation": "2018-12-01",
  "dateecheance": "2019-05-01",
  "etat": "EN COURS",
  "tokenconnexion": "ancfhb5f2a8c4t8a2cf7",
  "participants-id": [
    "2"
  ]
}
```

On remarque ici que la date était bien « en-cours » pour autoriser la modification, et que la date d'échéance a bien été modifiée.

*Contrairement à une méthode PUT classique, celle-ci ne nécessite pas de body de mise à jour. Il s'agit d'une adaptation mise en place afin de modifier uniquement la date de fin de la requête. La méthode patch aurait pu être utilisée pour une mise à jour partielle, mais sa mise en œuvre n'a pas été aboutie.*

*GET taches?statut={statut}*

Cette méthode permet, en spécifiant l'un des statuts CREEE, EN COURS, ACHEVEE ou ARCHIVEE, d'affiche la liste des tâches correspond à ce statut.



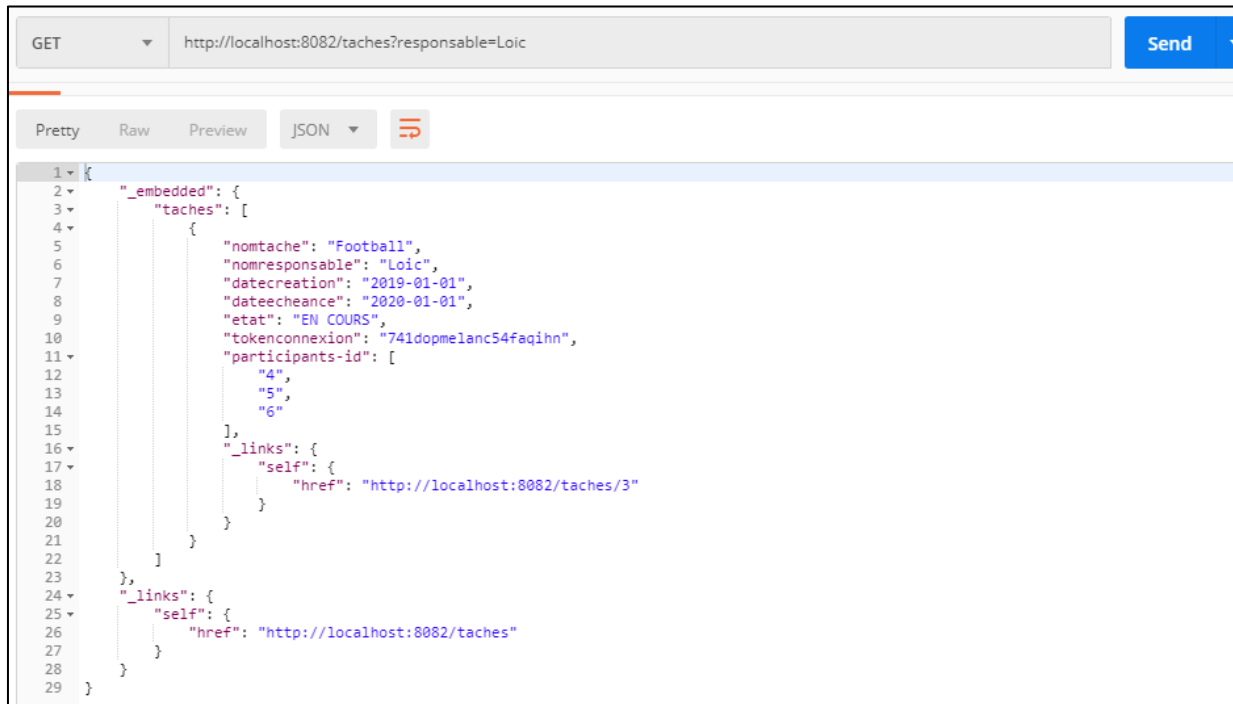
```
GET http://localhost:8082/taches?statut=EN COURS Send

Pretty Raw Preview JSON

1 {
2   "_embedded": {
3     "taches": [
4       {
5         "nomtache": "Football",
6         "nomresponsable": "Loic",
7         "datecreation": "2019-01-01",
8         "datecheance": "2020-01-01",
9         "etat": "EN COURS",
10        "tokenconnexion": "741dopmelanc54faqihn",
11        "participants-id": [
12          "4",
13          "5",
14          "6"
15        ],
16        "_links": {
17          "self": {
18            "href": "http://localhost:8082/taches/3"
19          }
20        }
21      },
22      {
23        "nomtache": "Projet API",
24        "nomresponsable": "Monsieur Perrin",
25        "datecreation": "2018-12-01",
26        "datecheance": "2019-05-01",
27        "etat": "EN COURS",
28        "tokenconnexion": "ancfbb5f2a8c4t8a2cf7",
29        "participants-id": [
30          "2"
31        ]
32      }
33    ]
34  }
35 }
```

*GET taches?responsable={responsable}*

De même que pour le statut, il est possible de spécifier un nom de responsable pour retourner l'ensemble des tâches ayant cette information pour responsable.



```
GET http://localhost:8082/taches?responsable=Loic Send

Pretty Raw Preview JSON

1 {
2   "_embedded": {
3     "taches": [
4       {
5         "nomtache": "Football",
6         "nomresponsable": "Loic",
7         "datecreation": "2019-01-01",
8         "datecheance": "2020-01-01",
9         "etat": "EN COURS",
10        "tokenconnexion": "741dopmelanc54faqihn",
11        "participants-id": [
12          "4",
13          "5",
14          "6"
15        ],
16        "_links": {
17          "self": {
18            "href": "http://localhost:8082/taches/3"
19          }
20        }
21      }
22    ]
23  },
24  "_links": {
25    "self": {
26      "href": "http://localhost:8082/taches"
27    }
28  }
29 }
```

*GET taches/{tacheId}/participants*

*\* token de connexion à fournir*

Toujours avec le même système de token d'accès, il est possible d'accéder à la liste des participants d'une tâche :

GET <http://localhost:8082/taches/2/participants>

☒ token a5vr8a1ciypzav5ahcoe

Key Value Description

Body Cookies Headers (4) Test Results Status: 200 OK

Pretty Raw Preview JSON

```

1 {
2   "_embedded": {
3     "participants": [
4       {
5         "nom": "Mansuy",
6         "prenom": "Claire",
7         "tacheId": "2",
8         "_links": {
9           "self": {
10            "href": "http://LAPTOP-MCM4E5D0:8080/participants/1"
11          }
12        }
13      },
14      {
15        "nom": "Fernandes",
16        "prenom": "Charlotte",
17        "tacheId": "2",
18        "_links": {
19          "self": {
20            "href": "http://LAPTOP-MCM4E5D0:8080/participants/3"
21          }
22        }
23      }
24    ]
25  }
26 }

```

*GET taches/{tacheId}/participants/{participantId}*

*\* token de connexion à fournir*

Comme ci-dessous, il est possible d'accéder à la liste des participants d'une tâche, et de restreindre l'accès à un participant particulier.

GET <http://localhost:8082/taches/2/participants/1> Send

Params Authorization Headers (1) Body Pre-request Script Tests Cookies

☒ token a5vr8a1ciypzav5ahcoe

Key Value Description

Body Cookies Headers (4) Test Results Status: 200 OK Time: 1876 ms Size: 332

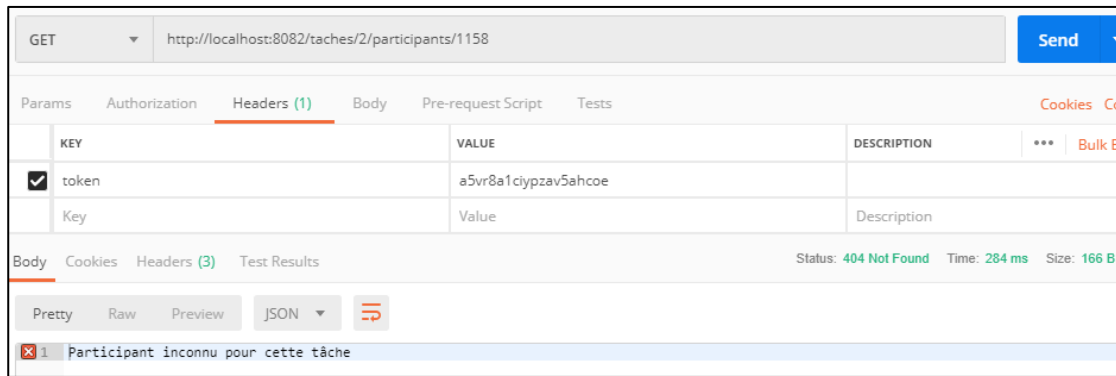
Pretty Raw Preview JSON

```

1 {
2   "nom": "Mansuy",
3   "prenom": "Claire",
4   "tacheId": "2",
5   "_links": {
6     "self": [
7       {
8         "href": "http://LAPTOP-MCM4E5D0:8080/participants/1"
9       }
10      ],
11      "tache": {
12        "href": "http://LAPTOP-MCM4E5D0:8080/participants"
13      }
14    }
15  }

```

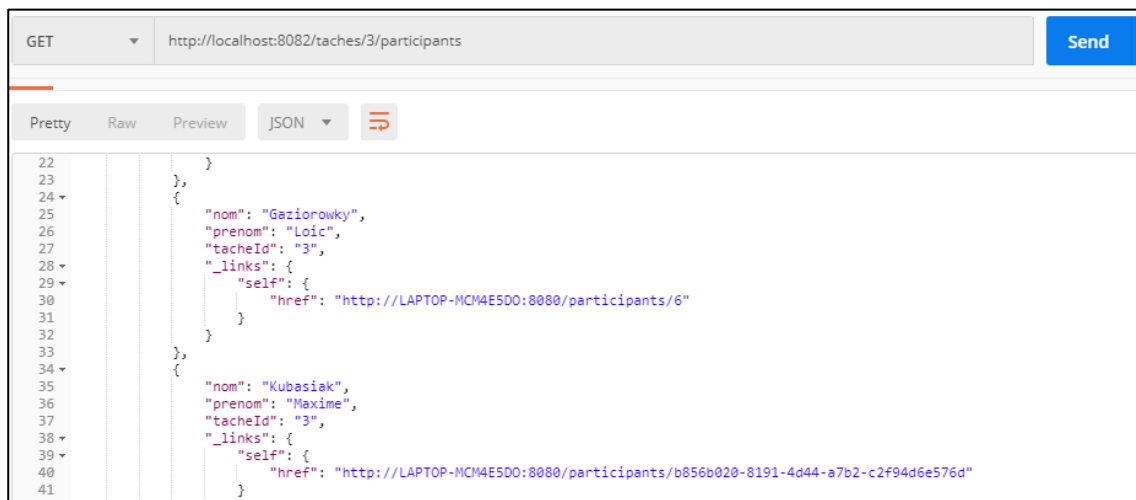
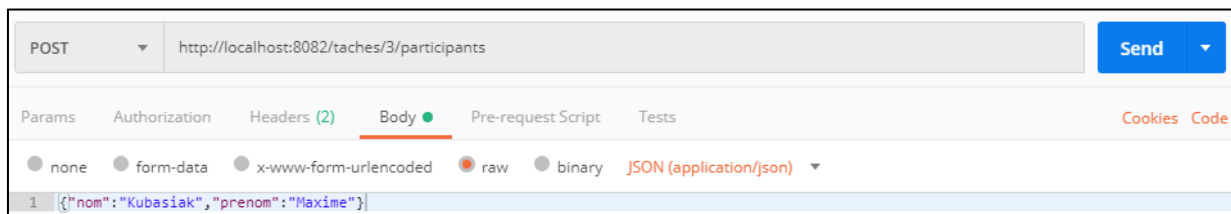
si une demande sur un participant n'ont existant est effectué, cela sera signalé à l'utilisateur.



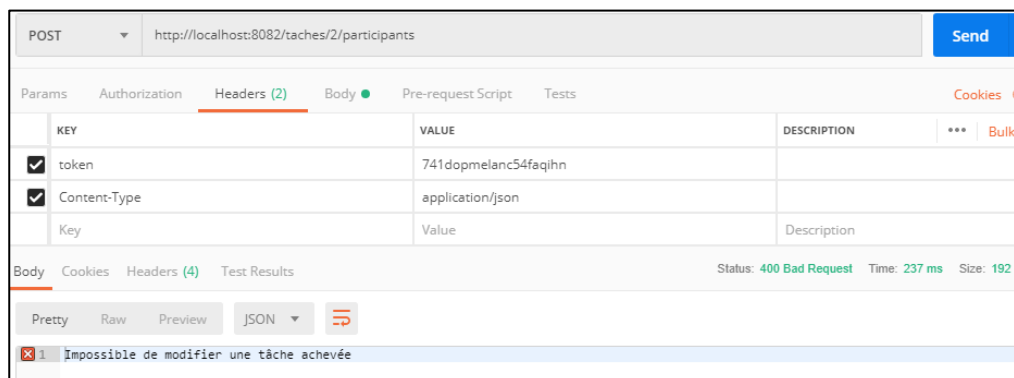
### POST taches/{tacheId}/participants

*\* token de connexion à fournir*

Cette fois encore, il est nécessaire d'utiliser le token d'accès pour ajouter un nouveau participant à une tâche. Il suffit pour cela de spécifier le participant à ajouter dans le body de la requête, pour que le participant soit ajouté à la tâche.



Dans le cas où la tâche est achevée, aucun ajout de participant ne peut être effectué :



**DELETE** `taches/{tacheId}/participants/{participantId}`

*\* token de connexion à fournir*

Pour finir, il est possible de supprimer un participant d'une tâche, de nouveau à condition de fournir le token d'accès, et sous réserve que la tâche ne soit pas achevée ou archivée ; ou encore qu'il reste au minimum un participant dans la tâche.

DELETE		http://localhost:8082/taches/3/participants/6		Send
Params	Authorization	Headers (2)	Body	Pre-request Script Tests
	KEY	VALUE	DESCRIPTION	...
<input checked="" type="checkbox"/>	token	741dopmelanc54faqihn		
<input checked="" type="checkbox"/>	Content-Type	application/json		
	Key	Value	Description	
Body Cookies Headers (2) Test Results				Status: 200 OK Time

GET		http://localhost:8082/taches/3/participants/6		Send
Params	Authorization	Headers (2)	Body	Pre-request Script Tests
	KEY	VALUE	DESCRIPTION	...
<input checked="" type="checkbox"/>	token	741dopmelanc54faqihn		
<input checked="" type="checkbox"/>	Content-Type	application/json		
	Key	Value	Description	
Body Cookies Headers (3) Test Results				Status: 404 Not Found Time: 133 ms Size: 166 B
Pretty Raw Preview JSON				
<div> <div>1</div> <div>Participant inconnu pour cette tâche</div> </div>				

### 4.3. Implémentation des requêtes

Cette partie vise à expliquer plus précisément le fonctionnement de chacune des requêtes, ainsi que les contraintes sur ces requêtes.

#### *GET taches*

Cette méthode effectue un simple findAll sur le repository des tâches pour ensuite afficher l'ensemble des résultats obtenus ainsi qu'un code OK.

```
/**
 * Requete d'accès à l'ensemble des tâches
 * @return ResponseEntity
 */
public ResponseEntity<?> getAllTaches() {
    Iterable<Tache> allTaches = tacheRepository.findAll();
    return new ResponseEntity<>(tacheToResource(allTaches), HttpStatus.OK);
}
```

*Service Tache – TacheService*

#### *GET taches/{tacheId}*

*\* token de connexion à fournir*

Comme indiqué précédemment, cette méthode permet de récupérer les informations d'une tâche particulière. Le token d'accès à la tâche est tout d'abord vérifié pour autoriser ou non l'accès à la tâche. Deux types d'interdiction sont ici utilisés :

- Une interdiction pour token invalide
- Une interdiction pour tâche inexistante

```
/**
 * Requete qui retourne une tâche particulière
 * @param id identifiant de la tâche
 * @param token token de vérification de l'accès à la tâche
 * @return ResponseEntity
 */
@GetMapping(value =("/{tacheId}")
public ResponseEntity<?> getTache(@PathVariable("tacheId") String id,@RequestHeader(value="token") String token) {

    AutorisationAcces autorisation = tacheService.verificationAutorisationAcces(id,token);
    if(!autorisation.equals(AutorisationAcces.AUTORISE)) {
        return new ResponseEntity<>(autorisation.getMessage(),autorisation.getHttpStatus());
    }

    return tacheService.getTache(id);
}
```

*Service Tache - TacheController*

Après avoir autorisé l'accès à la tâche, celle-ci est recherchée sur le repository tout en s'assurant de nouveau que celle-ci est présente. Cette vérification de présence aurait pu être retirée, mais a été conservée afin de pouvoir utiliser cette méthode getTache sans forcément avoir de vérification d'accès et donc d'existence au préalable.

```
/**
 * Requete d'accès à une tâche
 * @param id identifiant de la tâche à rechercher
 * @return ResponseEntity
 */
public ResponseEntity<?> getTache(@PathVariable("tacheId") String id) {
    return Optional.ofNullable(tacheRepository.findById(id)).filter(Optional::isPresent)
        .map(tache -> new ResponseEntity<>(tacheToResource(tache.get(), true), HttpStatus.OK))
        .orElse(new ResponseEntity<>(HttpStatus.NOT_FOUND));
}
```

*Service Tache - TacheService*

**POST taches**

Cette méthode récupère les informations sur la tâche passée dans le body de la requête, afin de lui attribuer un id unique, mais également un token selon la politique de sécurité établie plus haut.

Des vérifications sont réalisées sur la tâche, comme le fait que la date de fin doit forcément être après la date de création, sinon celle-ci ne pourra pas être sauvegardée.

Par la suite, on peut apporter le bon état « créée » à la tâche puisque celle-ci ne peut être créée que sans participant par l'intermédiaire de cette méthode.

```
/**
 * Requete de sauvegarde d'une tache dans le service
 * @param tache tache a enregistrer
 * @return ResponseEntity
 */
public ResponseEntity<?> saveTache(@RequestBody Tache tache) {
    tache.setId(UUID.randomUUID().toString());
    tache.setDatecreation(new Date());
    tache.setTokenConnexion((UUID.randomUUID().toString() + UUID.randomUUID().toString().substring(20, 40));

    // On vérifie que la date de fin est postérieure à la date de début
    if (tache.getDatecreation().compareTo(tache.getDatecheance()) >= 0) {
        // Date de création supérieure à l'échéance
        return new ResponseEntity<>("La date d'échéance de la date doit être ultérieure à la date du jour",
            HttpStatus.BAD_REQUEST);
    } else {
        tache.setEtat(EtatTache.CREEE.getEtat()); // On instancie au 1er état, créé
        Tache saved = tacheRepository.save(tache);
        HttpHeaders responseHeaders = new HttpHeaders();
        responseHeaders.setLocation(LinkTo(TacheController.class).slash(saved.getId()).toUri());
        return new ResponseEntity<>(null, responseHeaders, HttpStatus.CREATED);
    }
}
```

Service Tache – TacheService

**DELETE taches/{tacheId}**

*\* token de connexion à fournir*

Pour cette méthode de clôture de tâche, l'accès est sécurisé est au préalable vérifié, pour ensuite pouvoir modifier la requête. Il s'agit d'un simple changement d'état de la tâche afin de l'achever.

```
/**
 * Requete de cloture d'une tache du service
 * @param id identifiant de la tache a cloturer
 * @return ResponseEntity
 */
public ResponseEntity<?> deleteTache(@PathVariable("tacheId") String id) {
    Optional<Tache> tacheOptional = tacheRepository.findById(id);
    if (tacheOptional.isPresent()) {
        // tacheRepository.delete(tache.get());
        // On ne la delete pas vraiment mais uniquement un changement d'état
        Tache tache = tacheOptional.get();
        tache.setEtat(EtatTache.ACHEVEE.getEtat());
        tacheRepository.save(tache);
    }
    return new ResponseEntity<>(HttpStatus.OK); // OK et non pas NO_CONTENT comme un delete
}
```

Service Tache - TacheService



*PUT taches/{tacheId}?dateFin={nouvelleDateFin}*

*\* token de connexion à fournir*

Comme pour l'ensemble des méthodes propres aux tâches, l'accès à cette requête est vérifié grâce au token.

La tâche à modifier est récupérée afin de s'assurer que celle-ci n'est pas achevée, entraînant l'interdiction de la modification. De plus, si celle-ci est bien en cours, la nouvelle date d'échéance doit respecter des contraintes, comme être postérieure à la date courante ou la date de création de la tâche. La tâche peut ensuite être modifiée et enregistrée.

```
/**
 * Methode permettant de mettre a jour la date de fin d'une tache
 * @param nouvelleDate nouvelle date a prendre en compte
 * @param id identifiant de la tache a modifier
 * @return ResponseEntity
 */
public ResponseEntity<> updateTacheDateFin(Date nouvelleDate, String id){
    if (!tacheRepository.existsById(id)) {
        return new ResponseEntity<>(HttpStatus.NOT_FOUND);
    }
    else {
        Tache tache = tacheRepository.getOne(id);

        if(tache.getEtat().equals(EtatTache.ACHEVEE.getEtat()) || tache.getEtat().equals(EtatTache.ARCHIVEE.getEtat())) {
            return new ResponseEntity<>("Impossible de modifier une tâche achevée ou archivée", HttpStatus.BAD_REQUEST);
        }
        else if(nouvelleDate.before(tache.getDatecreation()) || nouvelleDate.before(new Date())) {
            return new ResponseEntity<>("La date est invalide. Impossible d'avoir une date de "
                + "fin antérieure à la date de création ou à la date du jour",HttpStatus.BAD_REQUEST);
        }
        else {
            tache.setDateecheance(nouvelleDate);
            tacheRepository.save(tache).getDateecheance();
            return new ResponseEntity<>(tache,HttpStatus.OK);
        }
    }
}
```

*Service Tache – TacheService*

*GET taches?Statut = {statut}*

Cette méthode utilise une simple méthode de recherche du repository retournant une liste de tâches répondant à la contrainte.

```
/**
 * Requete d'accès aux taches selon leur etat
 * @param etat etat recherche
 * @return ResponseEntity
 */
public ResponseEntity<> getTacheByEtat(String etat) {
    List<Tache> allTaches = tacheRepository.findByEtat(etat);
    return new ResponseEntity<>(tacheToResource(allTaches), HttpStatus.OK);
}
```

*Service Tache – TacheService*

*GET taches?responsable={responsable}*

Cette méthode utilise une simple méthode de recherche du repository retournant une liste de tâches répondant à la contrainte.

```
/**
 * Requete d'accès aux taches selon leur responsable
 * @param responsable responsable a prendre en compte
 * @return ResponseEntity
 */
public ResponseEntity<> getTacheByNomresponsable(String responsable) {
    List<Tache> allTaches = tacheRepository.findByNomresponsable(responsable);
    return new ResponseEntity<>(tacheToResource(allTaches), HttpStatus.OK);
}
```

*Service Tache – TacheService*

*GET taches/{tacheId}/participants*

*\* token de connexion à fournir*

Cette requête réalise tout d'abord une vérification sur le token d'accès à la tâche. Elle effectue ensuite un appel sur une requête exposée par le service des participants :

```
/**
 * Requete permettant de recuperer l'ensemble des participants d'une tache
 * @param tacheId tache recherchee
 * @return ResponseEntity
 */
public ResponseEntity<> getParticipantsByTache(String tacheId){
    return participantServiceProxy.getParticipantsByTache(tacheId);
}
```

*Service Tache - TacheService*

```
/**
 * Requete de recherche des participants d'une tache
 * @param tacheId identifiant de la tache a prendre en compte
 * @return ResponseEntity
 */
@RequestMapping(method = RequestMethod.GET, value = "tache/{tacheId}")
ResponseEntity<> getParticipantsByTache(@PathVariable("tacheId") String tacheId);
```

*Service Tache - ParticipantServiceProxy*

Voici l'implémentation de cette méthode dans le service participant :

```
@GetMapping(value = "tache/{tacheId}")
public ResponseEntity<> getParticipantsByTache(@PathVariable("tacheId") String tacheId){
    Iterable<Participant> allParticipants = participantRepository.findByTacheid(tacheId);
    return new ResponseEntity<>(participantToResource(allParticipants), HttpStatus.OK);
}
```

*ServiceParticipant - ParticipantController*

Celle-ci utilise le repository des participants afin de récupérer l'ensemble des participants d'une tâche particulière. Ces informations sont ensuite transmises grâce à la classe de connexion entre les deux services, en @FeignClient.

*GET* `taches/{tacheId}/participants/{participantId}`

*\* token de connexion à fournir*

Cette méthode utilise une des requêtes exposées par le service des participants afin de récupérer un participant particulier d'une tâche précise.

```
/**
 * Requete permettant de recuperer un participant d'une tache
 * @param tacheId tache recherchee
 * @param participantId participant recherche
 * @return ResponseEntity
 */
public ResponseEntity<> getParticipantByTache(String tacheId, String participantId){
    ResponseEntity<> response;
    try {
        response = participantServiceProxy.getParticipantByTacheAndId(tacheId, participantId);
    } catch (feign.FeignException e) {
        response = new ResponseEntity<>("Participant inconnu pour cette tâche",HttpStatus.NOT_FOUND);
    }
    return response;
}
```

*Service Tache - TacheService*

```
/**
 * Requete de recherche des participants d'une tache
 * @param tacheId identifiant de la tache a prendre en compte
 * @return ResponseEntity
 */
@RequestMapping(method = RequestMethod.GET, value = "tache/{tacheId}")
ResponseEntity<> getParticipantsByTache(@PathVariable("tacheId") String tacheId);
```

*Service Tache - ParticipantServiceProxy*

```
@GetMapping(value = "{participantId}/tache/{tacheId}")
public ResponseEntity<> getParticipantByTacheAndId(@PathVariable("tacheId") String tacheId, @PathVariable("participantId") String participantId) {
    return Optional.ofNullable(participantRepository.findByTacheIdAndId(tacheId, participantId))
        .filter(Optional::isPresent)
        .map(i -> new ResponseEntity<>(participantToResource(i.get(), true), HttpStatus.OK))
        .orElse(new ResponseEntity<>(HttpStatus.NOT_FOUND));
}
```

*Service Participant – ParticipantController*

On remarque ici que la requête du service participant peut ne pas retourner de participant, et donc un NOT\_FOUND. Pour une raison inexpliquée, ce responseEntity du service des participants est interprété comme une exception au moment de la réception par le service des tâches. Pour cela, un catch sur l'exception correspondante est réalisé afin de savoir quand la requête n'a pas abouti à un résultat.

*POST* `taches/{tacheId}/participants`

*\* token de connexion à fournir*

Pour l'ajout d'un nouveau participant à une tâche, le token d'accès est tout d'abord vérifié, avant de vérifier par la suite que la tâche n'est pas achevée, pour autoriser sa modification.

```
/**
 * Requete d'ajout d'un participant à une tache
 * @param tacheId identifiant de la tache pour laquelle ajouter un participant
 * @param participant participant a ajouter a la tache
 * @return ResponseEntity
 */
@RequestMapping(method = RequestMethod.POST, value = "{tacheId}/participants")
protected ResponseEntity<> newParticipantTache(@PathVariable("tacheId") String tacheId, @RequestBody Participant participant,
    @RequestHeader(value="token") String token) throws JsonParseException, JsonMappingException, IOException {

    AutorisationAcces autorisation = tacheService.verificationAutorisationAcces(tacheId,token);
    if(!autorisation.equals(AutorisationAcces.AUTORISE)) {
        return new ResponseEntity<>(autorisation.getMessage(),autorisation.getHttpStatus());
    }

    Tache tache = tacheService.tacheRepository.getOne(tacheId);
    if(tache.getEtat().equals(EtatTache.ACHEVEE.getEtat())) {
        return new ResponseEntity<>("Impossible de modifier une tâche achevée", HttpStatus.BAD_REQUEST);
    }
    else {
        ResponseEntity<> response = participantService.newParticipantTache(tacheId, participant);
        Participant saved = Participant.StringToParticipant(response.getBody().toString());
        tacheService.ajoutParticipantTache(tacheId, saved.getId());
        return new ResponseEntity<>( HttpStatus.CREATED);
    }
}
```

*Service Tache –TacheController*

On peut ensuite ajouter un nouveau participant au service participant, puis ajouter son id à la tâche :

```
@PostMapping("/{tacheid}")
public Participant newParticipant(@PathVariable("tacheid") String tacheid, @RequestBody Participant participant) {
    participant.setId(UUID.randomUUID().toString());
    participant.setTacheId(tacheid);
    Participant saved = participantRepository.save(participant);
    return saved;
}
```

*Service Participant – ParticipantController*

```
/**
 * Methode permettant l'ajout d'une reference d'un participant a une tache
 * @param tacheId tache concernee
 * @param participantId id du participant a ajouter
 */
public void ajoutParticipantTache(String tacheId, String participantId) {
    Tache tache = tacheRepository.getOne(tacheId);
    Set<String> idParticipants = tache.getIdParticipants();
    idParticipants.add(participantId);
    tache.setIdParticipants(idParticipants);
    tache.setEtat(EtatTache.ENCOURS.getEtat());
    tacheRepository.save(tache);
}
```

*Service Tache – TacheService*

*DELETE taches/{tacheId}/participants/{participantId}*

*\* token de connexion à fournir*

La même démarche que pour l'ajout d'un participant est réalisée.