

# **TD4 - GraphQL Directus - Compte rendu**

Auteur : Carette Robin

Depot : <https://github.com/CaretteRobin/nouveaux-paradigmes-TD3-4.git>

## **Introduction**

Ce compte rendu presente la mise en oeuvre complete du TD4 autour de GraphQL avec Directus. L'objectif est de documenter un service exploitable par un client ou un chef de projet, en montrant la securisation des acces, les requetes de lecture, et les mutations de modification. Chaque capture ecran est commente pour justifier les points importants du sujet.

## **Objectifs du TD**

- Interroger l'API GraphQL de Directus (filtres, alias, fragments, variables).
- Mettre en place les autorisations (role, policy, utilisateurs).
- Verifier les acces avec et sans authentification.
- Executer les mutations demandees (creation, rattachement, suppression).

## **Contexte technique**

- Endpoint GraphQL : `http://localhost:8055/graphql`
- Methode : POST (requete GraphQL dans le body)
- Authentification : `Authorization: Bearer <TOKEN>`

Note : les identifiants et tokens sont volontairement masques dans ce rendu.

## **Securite et authentification**

Les collections sensibles `motif_visite` et `moyen_paiement` ne sont plus accessibles au role `Public`. Un role dedie (`td4_reader`) est cree avec des droits de lecture sur les collections utiles au TD, puis deux utilisateurs sont associes a ce role : un utilisateur avec token statique et un utilisateur utilise pour obtenir un JWT.

Ce rôle regroupe les droits de lecture sur l'ensemble des collections métier. Il isole les accès GraphQL par rapport au rôle Public et permet de tester l'authentification.

Deux comptes sont présentes : un utilisateur “statique” pour les tests avec token fixe et un utilisateur “jwt” pour les tests via login. Cela couvre les deux mécanismes demandés.

The screenshot shows a GraphQL playground interface with a 'Headers' tab selected. The 'Authorization' header is checked, and its value is empty. The response pane displays an error message in JSON format:

```

1 : {
2   "errors": [
3     {
4       "message": "GraphQL validation error.",
5       "extensions": {
6         "code": "GRAPHQL_VALIDATION",
7         "errors": [
8           {
9             "message": "Cannot query field \\"structure\\" on type \\"Query\\".",
10            "locations": [
11              {
12                "line": 2,
13                "column": 3
14              }
15            ]
16          }
17        ]
18      }
19    ]
20  }
21

```

Cette capture montre un refus d'accès sans token : la requête échoue quand le rôle Public ne possède plus les droits. C'est la preuve que les restrictions sont effectives.

## Requêtes GraphQL (1 à 10)

Les requêtes suivantes utilisent les collections `praticien`, `specialite`, `structure` et les relations créées au TD3. Adapter les noms des relations inverses si besoin (ex. `praticiens`).

### 1) Liste des praticiens (id, nom, prenom, telephone, ville)

```
query {
  praticien {
    id
    nom
    prenom
    telephone
    ville
  }
}
```

Retourne la liste des praticiens avec les champs demandés.

### 2) Idem + libelle de la spécialité

```
query {
  praticien {
    id
    nom
    prenom
    telephone
  }
}
```

```

    ville
    specialite {
      libelle
    }
  }
}

```

Ajout d'une relation M2O pour exposer le libelle de la specialite associee.

### 3) Idem + filtre ville = “Paris”

```

query {
  praticien(filter: { ville: { _eq: "Paris" } }) {
    id
    nom
    prenom
    telephone
    ville
    specialite {
      libelle
    }
  }
}

```

Filtre cote serveur sur un champ simple.

### 4) Idem + structure (nom, ville)

```

query {
  praticien {
    id
    nom
    prenom
    telephone
    ville
    structure {
      nom
      ville
    }
  }
}

```

Ajout d'une seconde relation pour recuperer la structure d'appartenance.

### 5) Idem + filtre email contenant “.fr”

```

query {
  praticien(filter: { email: { _contains: ".fr" } }) {

```

```

    id
    nom
    prenom
    email
    ville
}
}

```

Filtre de type “contains” sur une chaine.

#### 6) Praticiens rattachés à une structure dont la ville est “Paris”

```

query {
  praticien(filter: { structure: { ville: { _eq: "Paris" } } }) {
    id
    nom
    prenom
    ville
    structure {
      nom
      ville
    }
  }
}

```

Filtre sur un champ d'une relation (structure.ville).

#### 7) Deux listes via alias (Paris et Bourdon-les-Bains)

```

query {
  praticiens_paris: praticien(filter: { ville: { _eq: "Paris" } }) {
    id
    nom
    prenom
  }
  praticiens_bourdon: praticien(filter: { ville: { _eq: "Bourdon-les-Bains" } }) {
    id
    nom
    prenom
  }
}

```

Alias utilisés pour retourner deux listes distinctes dans une seule réponse.

#### 8) Même requête avec fragment

```

fragment PracticienFields on praticien {
  id
}

```

```

    nom
    prenom
    telephone
    ville
}

query {
  praticiens_paris: praticien(filter: { ville: { _eq: "Paris" } }) {
    ...PracticienFields
  }
  praticiens_bourdon: praticien(filter: { ville: { _eq: "Bourdon-les-Bains" } }) {
    ...PracticienFields
  }
}

```

Le fragment factorise la selection de champs.

### 9) Requete parametree par variable

```

query PracticiensParVille($ville: String) {
  praticien(filter: { ville: { _eq: $ville } }) {
    id
    nom
    prenom
    telephone
    ville
  }
}

```

Variables :

```
{ "ville": "Paris" }
```

La valeur de la ville est externalisee pour reutiliser la requete.

### 10) Structures + praticiens + specialites

```

query {
  structure {
    nom
    ville
    praticiens {
      nom
      prenom
      email
      specialite {
        libelle
      }
    }
  }
}

```

```

        }
    }
}
```

Requete imbriquée pour remonter les praticiens et leur spécialité par structure.

```

query {
  structure { nom ville praticiens { nom prenom email specialite { libelle } } }
}

```

```

{
  "structure": [
    {
      "nom": "Cabinet Medical du Centre",
      "ville": "Villers-sur-Mer",
      "praticiens": [
        {
          "nom": "Durand",
          "prenom": "Luc",
          "email": "luc.durand@example.org",
          "specialite": {
            "libelle": "Dermatologie"
          }
        },
        {
          "nom": "Petit",
          "prenom": "Sarah",
          "email": "sarah.petit@example.org",
          "specialite": {
            "libelle": "Oncologie"
          }
        },
        {
          "nom": "Maison de Sante",
          "ville": "Lourville",
          "praticiens": [
            {
              "nom": "Petit",
              "prenom": "Sarah",
              "email": "sarah.petit@example.org",
              "specialite": {
                "libelle": "Oncologie"
              }
            }
          ]
        }
      ]
    }
  ]
}

```

Cette

capture confirme l'exécution des requêtes GraphQL avec des relations imbriquées. Elle sert de preuve d'exécution pour les requêtes 1 à 10, en particulier la requête 10.

## Tests d'autorisation (exigés par le sujet)

Les requêtes suivantes doivent échouer sans authentification, puis fonctionner avec les comptes du rôle `td4_reader`.

### Sans authentification (attendu : échec)

```

query {
  moyen_paiement {
    id
    libelle
  }
}
```

### Avec token statique (`moyen_paiement`)

```

query {
  moyen_paiement {
    id
    libelle
  }
}
```

Resultat : liste des moyens de paiement.

#### Avec JWT (specialites + motifs de visite)

```
query {
  specialite {
    id
    libelle
    motif_visite {
      id
      libelle
    }
  }
}
```

Resultat : chaque specialite expose ses motifs associes.

### Mutations GraphQL

Les mutations suivantes repondent a la liste demandee. Les identifiants sont a adapter selon les donnees presentes dans la base.

#### 1) Creer la specialite “cardiologie”

```
mutation {
  create_specialite_item(data: { libelle: "cardiologie" }) {
    id
    libelle
  }
}
```

#### 2) Creer un praticien (nom, prenom, ville, email, telephone)

```
mutation {
  create_praticien_item(
    data: {
      nom: "Dupont"
      prenom: "Alice"
      ville: "Paris"
      email: "alice.dupont@example.fr"
      telephone: "0102030405"
    }
  ) {
    id
    nom
    prenom
  }
}
```

```
    }
}
```

### 3) Rattacher ce praticien a “cardiologie”

```
mutation RattacherPraticienCardio($praticienId: ID!, $specialiteId: ID!) {
  update_praticien_item(id: $praticienId, data: { specialite: $specialiteId }) {
    id
    nom
    specialite {
      id
      libelle
    }
  }
}
```

Variables :

```
{ "praticienId": 1, "specialiteId": 10 }
```

### 4) Creer un praticien rattaché a “cardiologie”

```
mutation CreerPraticienCardio($specialiteId: ID!) {
  create_praticien_item(
    data: {
      nom: "Martin"
      prenom: "Leo"
      ville: "Nancy"
      email: "leo.martin@example.fr"
      telephone: "0611223344"
      specialite: $specialiteId
    }
  ) {
    id
    nom
    prenom
    specialite {
      libelle
    }
  }
}
```

Variables :

```
{ "specialiteId": 10 }
```

5) Creer un praticien et sa specialite “chirurgie”

```
mutation {
  create_praticien_item(
    data: {
      nom: "Bernard"
      prenom: "Claire"
      ville: "Lyon"
      email: "claire.bernard@example.fr"
      telephone: "0677889900"
      specialite: { libelle: "chirurgie" }
    }
  ) {
    id
    nom
    prenom
    specialite {
      id
      libelle
    }
  }
}
```

6) Ajouter un praticien a la specialite “chirurgie”

```
mutation RattacherPraticienChirurgie($praticienId: ID!, $specialiteId: ID!) {
  update_praticien_item(id: $praticienId, data: { specialite: $specialiteId }) {
    id
    nom
    specialite {
      id
      libelle
    }
  }
}
```

Variables :

```
{ "praticienId": 2, "specialiteId": 11 }
```

7) Rattacher le premier praticien a une structure existante

```
mutation RattacherPraticienStructure($praticienId: ID!, $structureId: ID!) {
  update_praticien_item(id: $praticienId, data: { structure: $structureId }) {
    id
    nom
    structure {
      id
    }
  }
}
```

```
    nom  
    ville  
}  
}  
}
```

## Variables :

```
{ "praticienId": 1, "structureId": 3 }
```

8) Supprimer les deux derniers praticiens crees

```
mutation SupprimerPraticiens($ids: [ID!]!) {
  delete_praticien_items(ids: $ids) {
    ids
  }
}
```

## Variables :

```
{ "ids": [2, 3] }
```

The screenshot shows the MongoDB Compass interface with the following details:

- Project:** My Workspace
- Collection:** test
- API:** dev web server
- Method:** POST API
- URL:** POST http://localhost:8080/api/graph
- Body:** JSON (shown as a query document)

```
1 mutation {  
2   create_specialite_item(data: { libelle: "cardiologie" }) {  
3     id  
4     libelle  
5   }  
6 }  
7
```

- Response:** JSON (200 OK, 99ms, 7B)

```
1   "data": {  
2     "createSpecialiteItem": {  
3       "id": "407",  
4       "libelle": "cardiologie"  
5     }  
6   }  
7 }
```

Cette capture prouve l'exécution des mutations en contexte admin et la création d'une spécialité. Elle valide le bon fonctionnement des opérations d'écriture.

## Conclusion

Le TD4 est réalisé de bout en bout : création du rôle et des utilisateurs, vérification des droits, exécution des requêtes GraphQL (filtres, alias, fragments, variables) et exécution des mutations. Les captures apportent une preuve claire des actions et résultats. Le service est ainsi documenté de manière professionnelle et directement exploitable.