# TM n°2

L'objectif du TP est de programmer un serveur REST pour la base de données de l'exemple de MCD Médecin/Patient (TD 2, Ex 3). La base de données SQL est celle issue du TM n°1. Le travail va donc consister à fournir une interface (API) vers cette base données. Pour cela, une architecture REST va être définie.

## 1 Comprendre l'esprit des architectures REST

REST [1] (*Representational State Transfer*) propose un ensemble de préconisations pour élaborer un service Web léger et modulaire. Chaque ressource mise à disposition par le serveur est identifiée de façon homogène par un URI [2] (*Uniform Resource Identifier*). Par exemple, dans notre serveur, un médecin sera accessible par l'URI

`http://localhost:8887/medecins/123456789`

qui précise, dans l'ordre,
- le protocole d'accès (ici HTTP),
- l'hôte (`localhost`),
- le port (8887), et
- le chemin d'accès (`/medecins/123456789`).

L'hôte et le port sont anecdotiques dans ce TP. Votre machine, dont le nom en local est en général `localhost`, hébergera le serveur dont le service sera exposé sur le port 8887 choisi arbitrairement. Détaillons plus particulièrement les deux autres éléments.

**Les chemins d'accès et types de ressource.** L'un des enjeux importants lors de la conception d'un serveur Web REST est l'élaboration des chemins d'accès. REST distingue 4 types de ressources : les documents, les collections, les *storages* et les opérations. Seules les 2 premiers nous intéresserons ici.

**Document :** Il représente un objet individuel fournit par le serveur. Ces documents ne sont pas accessibles directement mais le serveur permet d'en obtenir une *représentation*. Cette représentation comprend généralement à la fois des champs avec des valeurs et des liens vers d'autres ressources connexes. Les documents ne sont pas figés et leur état peut évoluer ; si le serveur le permet, le client peut demander la modification de champs et faire vivre le document. Il peut également demander la suppression du document. Dans notre cas, les documents correspondent aux différentes entités et associations représentées telles que les médecins, les patients, les spécialités, etc. Voici quelques exemples de chemins d'accès vers des documents de ce TP.

```
/patients/214119912345633
/specialistes/cardiologie/123456789
/suivis/214119912345633/cardiologie/123456789/actes/2025-09-15
```

1. `https://fr.wikipedia.org/wiki/Representational_state_transfer`
2. `https://fr.wikipedia.org/wiki/Uniform_Resource_Identifier`

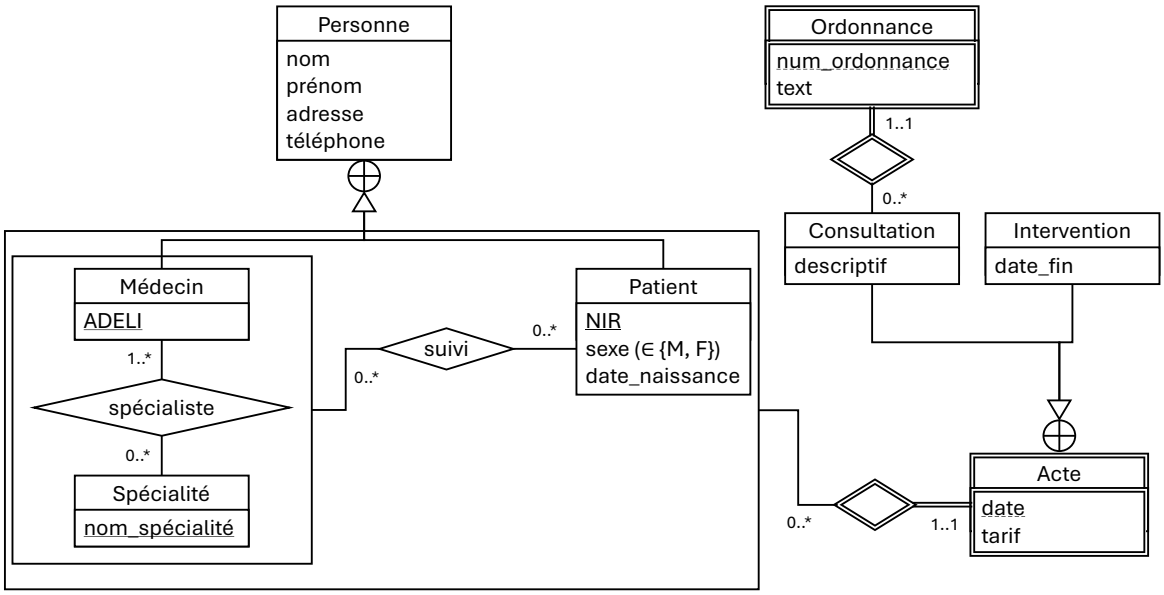


FIGURE 1 – Schéma E/A

**Collection :** Il s'agit d'un répertoire de ressources gérées par le serveur. La collection permet aux clients d'accéder aux ressources qu'elle contient mais également d'en créer de nouvelles. En général, une collection regroupe des ressources de même type. Dans notre cas, les collections correspondent aux classes d'entités et aux classes d'associations. Voici quelques exemples de chemins d'accès vers des collections de ce TP.

```
/personnes/
/specialites/cardiologie/specialistes/
/suivis/214119912345633/cardiologie/123456789/interventions/
```

Les chemins d'accès seront élaborés à partir du schéma E/A de la Figure 1. Ils seront fournis au fur et à mesure du développement du serveur.

**Le protocole HTTP en REST.** Les architectures REST utilisent en général le protocole de communication HTTP. HTTP (*HyperText Transfer Protocol*) est le protocole de communication permettant d'échanger des données entre un client et un serveur web. Chaque échange est constitué d'une requête (*request*) émise par le client et de la réponse (*response*) du serveur (voir figure 2). Le protocole définit comment les requêtes et les réponses sont formatées et transmises. REST fait un usage spécifique de ce protocole.

**Requête :** Dans une requête, le client spécifie une ressource (à travers un URI) et une opération, appelée *méthode*, précisant l'action à effectuer. Le protocole définit un ensemble fixe de méthodes ; seules 4 d'entre elles seront utilisées : `GET`, `POST`, `PUT` et `DELETE`. Au besoin, des informations supplémentaires peuvent être fournies. En particulier, une zone de données libre, appelées le corps de la requête (ou *body*), permet de transmettre un contenu dans un format précisé par le client. Par exemple, lors de la création d'une nouvelle ressource, on pourra préciser dans le corps le représentation de l'état de cette ressource.

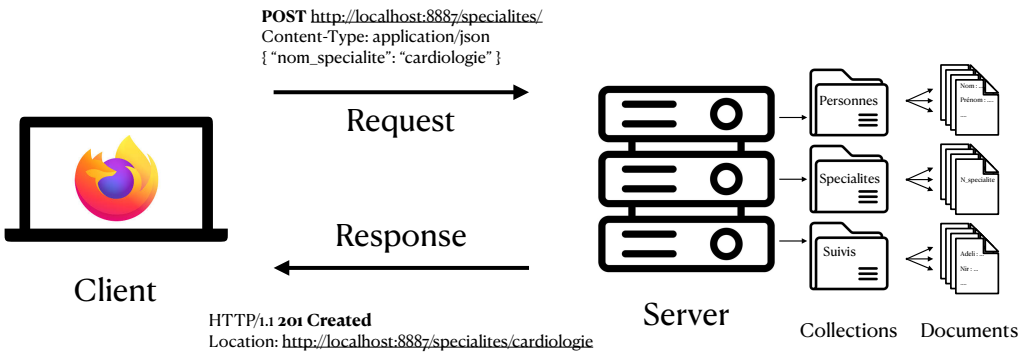


FIGURE 2 – Exemple d'interaction client-server via HTTP.

```
POST http://localhost:8887/specialites/
Content-Type: application/json
{ "nom_specialite": "cardiologie" }
```

Cette requête permettra de créer la spécialité *cardiologie*. Comme vous pouvez le constater, le corps est formaté en JSON (*JavaScript Object Notation*), un format de données textuel dérivé de la notation des objets du langage JavaScript.

**Réponse :** A la réception d'une requête, le serveur va tenter d'effectuer l'opération demandée, suite à quoi il retournera une réponse spécifiant si l'opération a été réalisée avec succès ou non, et les données demandées le cas échéant. Pour cela, la réponse contient un statut, c'est-à-dire un numéro standardisé indiquant comment les choses se sont déroulées. Parmi les codes statut standards, nous utiliserons les suivants.

| statut | signification |
|---|---|
| 200 | OK |
| 201 | Created |
| 204 | No Content |
| 404 | Not Found |
| 409 | Conflict |
| 412 | Precondition Failed |
| 500 | Internal Server Error |

Pour les informations de retour, tout comme la requête, la réponse dispose d'un corps. On peut également utiliser des champs particuliers, tels que `Location` dont la valeur est un URI, dans le cas de mises-à-jour ou de créations de ressources par exemple. Ainsi, pour la requête précédente, un exemple de réponse, au cas où la ressource a bien été créée, serait le suivant.

```
HTTP/1.1 201 Created
Location: http://localhost:8887/specialites/cardiologie
```

indiquant en champ `Location` l'URI de la ressource créée.

```
api/
|-- db/
|   |-- db.js
|   |-- schema.sql
|-- public/
|   |-- client.js
|   |-- index.html
|-- routes/
|   |-- domains.js
|   |-- patient.js
|-- nodemon.json
|-- server.js
```

FIGURE 3 – Arborescence de l'archive.

# 2 Mise en place de l'environnement de travail

— Décompresser l'archive `cbd_tp2.zip` dans le répertoire de votre choix. L'archive contient l'arborescence donnée Figure 3. Ces fichiers contiennent un commencement de serveur Web.

`server.js` : il s'agit du fichier principal du serveur. Il contient son initialisation et la mise en place des routes de la racine. On utilise la bibliothèque `express`.

**Vous aurez à modifier en partie ce fichier.**

`routes/` : ce répertoire contient les *router*s de votre serveur. Pour le moment, ce répertoire contient le fichier `patient.js` qui a la gestion des routes de la forme `/patients/...` (cf. fichier `server.js`). Durant le TP, des routers seront ajoutés dans ce répertoire. On remarquera également la présence du fichier `domains.js` qui contient quelques fonctions permettant de vérifier qu'une donnée est bien formatée. Par exemple, la fonction `checkIsNIR` qui y est définie, vérifie que son paramètre est bien une chaîne de caractères codant un numéro de sécurité sociale.

**Vous aurez à créer ou modifier des fichiers dans ce répertoire.**

`public/` est un répertoire de fichiers mis à disposition côté client pour interagir avec le serveur. Le fichier `index.html` est le fichier d'accueil du serveur, servi sur la route `/index.html` (cf. `server.js`). Le fichier `client.js` fournit quelques fonctions qui permettront de lancer des requêtes depuis le client (dans notre cas, la console de votre navigateur Web). Il est servi sur la route `/client.js` et chargé par `index.html`.

`db/` est le répertoire contenant les fichiers de la base de données sqlite3. En particulier, `schema.sql` contient le code SQL pour la création de la base (en d'autres termes, la correction du TP 1). Le fichier `db.js` contient les éléments NodeJS pour créer et charger la base de données. On utilise la bibliothèque `better-sqlite3`.

`nodemon.json` est un fichier de configuration pour la bibliothèque `nodemon` permettant de monitorer le serveur, et notamment de le relancer lorsque des fichiers sources sont modifiés.

— Se placer dans le répertoire `api` et lancer les commandes suivantes.

```
npm init -y
npm install express better-sqlite3 nodemon
npx nodemon --legacy-watch
```

Les deux premières initialisent votre projet NodeJS et installe localement les bibliothèques utilisées. La dernière ligne lance `nodemon`, lui-même lançant le serveur. **Ne pas fermer ce terminal et le laisser tourner sans y toucher.** C'est ici que les erreurs apparaîtront lors de l'édition de vos fichiers.

— Ouvrir `firefox` et charger la page `http://localhost:8887/index.html`. On vous invite à ouvrir la console JavaScipt de votre navigateur pour interagir avec le serveur. En y écrivant,

```
await client_read("/");
```

vous obtenez en réponse un objet de la forme

```
{ read: "200 (OK)", body: {
    _links: [
        { rel: "personnes", href: "/personnes/" },
        { rel: "médecins", href: "/medecins/" },
        { rel: "patients", href: "/patients/" },
        ...
    ]
}}
```

qui s'affiche dans la console. Vous retrouvez ici le retour de la route `/` définie dans `server.js` par les lignes suivantes.

```
app.get('/', (req, res) => res.json({
  _links: [
    { rel: 'personnes', href: '/personnes/' },
    { rel: 'médecins', href: '/medecins/' },
    { rel: 'patients', href: '/patients/' },
    ...
  ]
}));
```

En effet, cette simple instruction spécifie que lors d'une requête `GET` vers la ressource `/`, le serveur envoie une réponse au format JSON (`res.json(...)`) dont le contenu précise l'ensemble des liens que le serveur est disposé à servir. Pour le moment, seule la route `/patients/` est opérationnelle ; exécuter la commande

```
await client_read("/patients/");
```

pour vous en apercevoir. En revanche, la commande

```
await client_read("/medecins/");
```

retournera une erreur 404.

# 3 Des patients... et des médecins

L'objectif de cette section est de comprendre le code du fichier `routes/patient.js` et de créer le fichier `routes/medecin.js`. Le fichier `routes/patient.js` a la structure suivante.

```
const express = require('express');
const db = require('../db/db.js');
const domains = require('./domains');

// Création du router
const router = express.Router();

// Configuration du router
router.get("/", (req, res) => { ... });
router.get("/:nir", (req, res) => { ... });
router.delete("/:nir", (req, res) => { ... });
router.post("/", (req, res) => { ... });
router.put("/:nir", (req, res) => { ... });

// Export du router
module.exports = router;
```

Après le chargement de la bibliothèque `express`, de la base de données `db`, et du module `domains` pour vérifier les domaines, un router `express` est créé, configuré, et enfin exporté. Ce router sera utilisé dans le fichier `server.js` à la ligne

```
app.use('/patients', require('./routes/patient'));
```

qui se lit littéralement : *Pour tout chemin d'accès commençant par `/patients`, utiliser le router exporté par le fichier `routes/patient.js`.*

**Question 1.** Créer un fichier `routes/medecin.js` identique à `routes/patient.js` sans la partie configuration du router.

**Question 2.** Ajouter à `server.js` la ligne suivante.

```
app.use('/medecins', require('./routes/medecin'));
```

La partie configuration des fichiers permet de spécifier les routes mises à disposition et les opérations disponibles sur ces ressources. Il y a dans le cas des patients 2 chemins d'accès :

1. `/patients/` : URI de la collection de tous les patients
2. `/patients/:nir` : URI d'un patient (la partie `:nir` sera remplacée par le NIR du patient).

Cinq fonctionnalités sont mises à disposition au total, 2 sur la collection des patients et 3 pour un patient donné. Elles implémentent les fameuses opérations **CRUD** (*Create*, *Read*, *Update*, *Delete*) pour les patients, c'est-à-dire la possiblité de créer un nouveau patient dans la base, de récupérer les données des patients, de les mettre à jour, et de les supprimer.

**Opération *Read* (List).** Il s'agit de récupérer les données des patients. Cette opération est associée dans les API REST à la méthode HTTP GET. On distingue ici deux cas, celui où l'on cherche à récupérer la liste des patients et celui où l'on cherche à récupérer les informations d'un patient donné. Nous nous intéressons ici au premier cas.

L'opération `GET /patients/` effectue la requête SQL

```
SELECT nir FROM Patient
```

pour récupérer la liste des NIR de tous les patients et retourne au format JSON la liste des URIs sous la forme `/patients/:nir` avec le statut 200. La donnée envoyée en cas de succès a donc la forme suivante.

```
[ '/patients/196078901234567',
  '/patients/200112345678901',
  '/patients/284056789012345' ]
```

En cas d'erreur, le statut 500 est retourné.

**Question 3.** Tester la route `/patients/` dans la console.

```
await client_read('/patients/');
```

**Question 4.** En s'inspirant du code de `patient.js`, ajouter dans `medecin.js`

```
router.get("/", (req, res) => { ... });
```

pour définir la méthode GET sur la route `/medecins/`.

**Question 5.** Tester le code en requêtant dans la console

```
await client_read('/medecins/');
```

Le retour attendu est le suivant.

```
[ '/medecins/123456789',
  '/medecins/456789123',
  '/medecins/987654321' ]
```

**Opération *Read* (One).** On cherche mainteneant à récupérer les informations d'un patient donné. L'opération `GET /patients/:nir` effectue la requête SQL

```
SELECT nir, nom, prenom, adresse, telephone, date_naissance, sexe
FROM Patient WHERE nir = ?
```

où le point d'interrogation prend la valeur du `:nir` fourni. Une fois exécutée les attributs sont regroupés dans un objet JavaScript auquel est ajouté le champ `_links` retournant une collection de liens contenant (pour le moment) simplement l'URI actuel nommé `self`. La donnée envoyée en cas de succès (statut 200) a donc la forme suivante.

```
{ nir: '196078901234567',
  nom: 'Dupond',
  prenom: 'Jean',
  adresse: '8 avenue de la République, 69003 Lyon',
  telephone: '0623456789',
  date_naissance: '1960-07-23',
  sexe: 'M',
  _links: {
    self: { href: '/patients/196078901234567' }
  }
}
```

Si la ressource n'existe pas, un statut 404 est levé. Pour toute autre erreur, le statut 500 est retourné.

**Question 6.** Tester la route `/patients/196078901234567` dans la console.

```
await client_read('/patients/196078901234567');
```

**Question 7.** En s'inspirant du code de `patient.js`, ajouter dans `medecin.js`

```
router.get("/:adeli", (req, res) => { ... });
```

pour définir la méthode GET sur la route `/medecins/:adeli`.

**Question 8.** Tester le code en requêtant dans la console

```
await client_read('/medecins/456789123');
await client_read('/medecins/999999999');
```

Le premier cas doit retourner un résultat suivant alors que le second génère une erreur 404.

```
{ adeli: '456789123',
  nom: 'Moreau'
  prenom: 'Isabelle'
  adresse: '5 place du Capitole, 31000 Toulouse'
  telephone: '0561782345'
  _links: {
    self: { href: '/medecins/456789123' }
  }
}
```

**Opération *Delete*.** Il s'agit de supprimer un élément de la base. Cette opération est associée dans les API REST à la méthode HTTP DELETE. Dans notre cas, l'opération n'est pas autorisée sur une collection. En revanche, il est possible de supprimer un patient donné. L'opération `DELETE /patients/:nir` effectue la requête SQL

```
DELETE FROM Patient WHERE nir = ?
```

où le point d'interrogation prend la valeur du `:nir` fourni. En cas de succès, le status de retour est 204. Si la ressource n'existe pas, un statut 404 est levé. Pour toute autre erreur, le statut 500 est retourné.

**Question 9.** Tester la route `/patients/200112345678901` dans la console.

```
await client_delete('/patients/200112345678901');
```

**Question 10.** En s'inspirant du code de `patient.js`, ajouter dans `medecin.js`

```
router.delete("/:adeli", (req, res) => { ... });
```

pour définir la méthode DELETE sur la route `/medecins/:adeli`.

**Question 11.** Tester le code en requêtant dans la console

```
await client_delete('/medecins/456789123');
await client_read('/medecins/456789123');
await client_delete('/medecins/999999999');
```

Le premier cas doit retourner statut 204, les deux autres génèrent des erreurs 404.

**Opération *Create*.** Il s'agit de créer un nouvel élément dans la base. Cette opération est associée dans les API REST à la méthode HTTP POST. L'opération s'effectue sur une collection afin d'y créer un nouveau membre. La représentation de l'élément créé est donné au format JSON dans le corps de la requête. L'opération

```
POST /patients/
Content-Type: application/json
{ "nir": "299054321098765",
  "nom": "Lefevre",
  "prenom": "Camille",
  "adresse": "18 rue des Fleurs, 44000 Nantes",
  "telephone": "0678123456",
  "date_naissance": "1999-05-12",
  "sexe": "M"
}
```

récupère la valeur des champs dans le corps, vérifie leurs formats et leurs validités, puis effectue la requête SQL

```
INSERT INTO Patient (nir, nom, prenom, adresse, telephone,
    date_naissance, sexe)
VALUES (?, ?, ?, ?, ?, ?, ?)
```

où les points d'interrogation prennent les valeurs des champs lus depuis le corps. En cas de succès, le champ `Location` de la réponse est initialisé à `/patients/:nir` où `:nir` prend la valeur du NIR récupéré, et le statut 201 est envoyé. Si l'un des champs ne vérifie pas les conditions de format, une erreur 412 est levée. En cas d'impossibilité due à des contraintes du schéma, le serveur lève une erreur 409. Pour toute autre erreur, le statut 500 est retourné.

**Question 12.** Tester la route `/patients/` dans la console.

```
await client_create('/patients/', {
    nir: "299054321098765",
    nom: "Lefevre",
    prenom: "Camille",
    adresse: "18 rue des Fleurs, 44000 Nantes",
    telephone: "0678123456",
    date_naissance: "1999-05-12",
    sexe: "M"
});
```

**Question 13.** En s'inspirant du code de `patient.js`, ajouter dans `medecin.js`

```
router.post("/", (req, res) => { ... });
```

pour définir la méthode POST sur la route `/medecins/`.

**Question 14.** Tester le code en requêtant dans la console

```
await client_create('/medecins/', {
    adeli: "123456789",
    nom: "Rousseau",
```

```
    prenom: "Marc",
    adresse: "10 boulevard Saint-Michel, 75005 Paris",
    telephone: ""
});
```

Cela devrait produire une erreur 412 car le téléphone n'est pas renseigné. Réessayer avec le téléphone `0156782345` ; cette fois-ci l'erreur est 409 car un médecin existe déjà avec ce numéro ADELI. Réessayer avec le numéro ADELI `753246801` pour cette fois créer correctement l'entrée. Observer la valeur du champ `Location` en retour.

**Opération *Update*.** Cette dernière opération permet de modifier l'état d'un élément de la base. Elle est associée dans les API REST à la méthode HTTP PUT. L'opération s'effectue sur un document donné. La mise-à-jour de l'état est donnée au format JSON dans le corps de la requête. L'opération

```
PUT /patients/299054321098765
Content-Type: application/json
{ "sexe": "F" }
```

récupère le NIR du patient, la valeur des champs à mettre à jour dans le corps (remarquer que la clé peut aussi être changée), vérifie leurs formats et leurs validités, puis effectue la requête SQL

```
UPDATE Patient
SET nir = COALESCE(?, nir),
    nom = COALESCE(?, nom),
    prenom = COALESCE(?, prenom),
    adresse = COALESCE(?, adresse),
    telephone = COALESCE(?, telephone),
    date_naissance = COALESCE(?, date_naissance),
    sexe = COALESCE(?, sexe)
WHERE nir = ?
```

où les points d'interrogation prennent les valeurs des champs lus depuis le corps. La fonction `COALESCE` prend pour valeur celle du premier argument s'il n'est pas `NULL`, sinon celle du deuxième. Ainsi, seuls les champs ayant une nouvelle valeur sont affectés par la modification. Remarquer la possibilité de modifier le NIR (la variable `nir` obtenue depuis l'URI et la variable `nir_` obtenue depuis le corps). En cas de succès, le statut 204 est retourné ; si la clé est modifiée, le champ `Location` de la réponse est initialisé avec le nouvel URI. Si l'un des champs à mettre à jour ne vérifie pas les conditions de format, une erreur 412 est levée. Si l'élément à modifier n'existe pas, une erreur 404 est levée (noter la requête `SELECT ...` dans le code pour vérifier l'existence). En cas d'impossibilité due à des contraintes du schéma, le serveur lève une erreur 409. Pour toute autre erreur, le statut 500 est retourné.

**Question 15.** Tester la route `/patients/299054321098765` dans la console.

```
await client_update('/patients/299054321098765', {
    nir: "299054321098764",
    sexe: "F"
});
```

**Question 16.** En s'inspirant du code de `patient.js`, ajouter dans `medecin.js`

```
router.put("/:adeli", (req, res) => { ... });
```

pour définir la méthode PUT sur la route `/medecins/:adeli`.

**Question 17.** Tester le code en requêtant dans la console

```
await client_update('/medecins/753246801', {
    adeli: "123456789",
    prenom: "Jean-Marc"
});
```

Cela devrait produire une erreur 409 car un médecin existe déjà avec le numéro ADELI `123456789`. Réessayer avec le numéro ADELI `753246802` pour cette fois modifier correctement l'entrée. Observer la valeur du champ `Location` en retour.

# 4 ... Et des spécialités

L'objectif est maintenant d'appliquer aux spécialités ce qui a été fait pour les médecins.

**Question 18.** Créer le fichier `routes/specialite.js` et ajouter dans `server.js` la ligne suivante.

```
app.use('/specialites', require('./routes/specialite'));
```

**Question 19.** Ajouter à `routes/specialite.js` l'opération Read (List).

```
router.get("/", (req, res) => { ... });
```

**Question 20.** Ajouter à `routes/specialite.js` l'opération Read (One).

```
router.get("/:nom_specialite", (req, res) => { ... });
```

**Question 21.** Ajouter à `routes/specialite.js` l'opération Delete.

```
router.delete("/:nom_specialite", (req, res) => { ... });
```

**Question 22.** Ajouter à `routes/specialite.js` l'opération Create.

```
router.post("/", (req, res) => { ... });
```

**Question 23.** Ajouter à `routes/specialite.js` l'opération Update.

```
router.put("/:nom_specialite", (req, res) => { ... });
```

# 5 Les spécialistes

Nous allons traiter les associations de la même façon que les entités. Cette section se concentre sur l'association Spécialiste. Pour rappelle, celle-ci relie un médecin à une spécialité. Un spécialiste sera donc identifié par le numéro ADELI du médecin et le nom de la spécialité. Les chemins d'accès seront les suivants :

— `/specialistes/` : URI de la collection des spécialistes.
— `/specialistes/:nom_specialite/:adeli` : URI d'un spécialiste donné.

L'état d'un spécialiste est représenté de la façon suivante en JSON.

```
{ _links: {
    self: { href: '/specialistes/:nom_specialite/:adeli' },
    medecin: { href: '/medecins/:adeli' },
    specialite: { href: '/specialites/:nom_specialite' }
  }
}
```

Comme l'association n'est pas attribuée, aucun attribut n'est présent dans cet objet, seul le champ `_links` est donné. Il n'est pas possible de mettre à jour (les attributs de) ce document : la méthode PUT est donc absente. Concernant le champ `_links`, contrairement aux entités vues jusqu'ici, le lien `self` est accompagné de deux autres liens donnant les URIs du médecin et de la spécialité concernés : ils implémentent les deux branches de l'association Spécialiste.

**Question 24.** Créer le fichier `routes/specialiste.js` et ajouter dans `server.js` la ligne suivante.

```
app.use('/specialistes', require('./routes/specialiste'));
```

**Question 25.** Ajouter à `routes/specialiste.js` l'opération Read (List).

```
router.get("/", (req, res) => { ... });
```

**Question 26.** Ajouter à `routes/specialiste.js` l'opération Read (One).

```
router.get("/:nom_specialite/:adeli", (req, res) => { ... });
```

**Question 27.** Ajouter à `routes/specialiste.js` l'opération Delete.

```
router.delete("/:nom_specialite/:adeli", (req, res) => { ... });
```

**Question 28.** Ajouter à `routes/specialiste.js` l'opération Create.

```
router.post("/", (req, res) => { ... });
```

Le corps de la requête aura la forme suivante.

```
{ nom_specialite: '....', adeli: '....' }
```

# 6 Les suivis

**Question 29.** Créer les suivis en suivant les mêmes principes que pour l'association Spécialiste.
— `/suivis/` : URI de la collection des suivis.
— `/suivis/:nir/:nom_specialite/:adeli` : URI d'un suivi donné. L'état d'un suivi est représenté en JSON de la façon suivante.

```
{ _links: {
    self: { href: '/suivis/:nir/:nom_specialite/:adeli' },
    patient: { href: '/patients/:nir' },
    specialiste:
      { href: '/specialistes/:nom_specialite/:adeli' }
  }
}
```

# 7 Pour finir

Pour l'instant, les entités et les associations classiques ont été implémentées. Il reste encore à travailler sur les héritages et sur les entités faibles. Vous êtes libres de développer ces éléments à votre guise mais en respectant les spécifications qui suivent.

**Entité Personne**

- URI de la collection : `/personnes/`. La seule méthode fournie est GET (pas de création car l'héritage est total).
- URI d'un document donné : `/personnes/:id`. Les méthodes fournies sont GET, DELETE et PUT.
- Représentation de l'état est la suivante. Noter l'absence du champs `id` qui n'est pas sur le MCD de la Fig. 1.

```
{ nom: '...',
  prenom: '...',
  adresse: '...',
  telephone: '...',
  _links: {
    self: { href: '/personnes/:id' },
  }
}
```

**Entité Intervention**

- URI de la collection : `/suivis/:nir/:nom_specialite/:adeli/interventions/`. Les méthodes fournies sont GET et POST.
- URI d'un document donné : `/suivis/:nir/:nom_specialite/:adeli/interventions/:date`. Les méthodes fournies sont GET, DELETE et PUT.
- Représentation de l'état est la suivante.

```
{ date: '...',
  date_fin: '...',
  tarif: '...',
  _links: {
    self: { href: '/suivis/:nir/:nom_specialite/:adeli
    /interventions/:date' },
  }
}
```

**Entité Consultation**

- URI de la collection : `/suivis/:nir/:nom_specialite/:adeli/consultations/`. Les méthodes fournies sont GET et POST.
- URI d'un document donné : `/suivis/:nir/:nom_specialite/:adeli/consultations/:date`. Les méthodes fournies sont GET, DELETE et PUT.
- Représentation de l'état est la suivante.

```
{ date: '...',
  descriptif: '...',
  tarif: '...',
  _links: {
    self: { href: '/suivis/:nir/:nom_specialite/:adeli
    /consultations/:date' },
  }
}
```

**Entité Acte**

- URI de la collection : `/suivis/:nir/:nom_specialite/:adeli/actes/`. La seule méthode fournie est GET (pas de création car l'héritage est total).
- URI d'un document donné : `/suivis/:nir/:nom_specialite/:adeli/actes/:date`. Les méthodes sont GET, DELETE et PUT.
- Représentation de l'état est la suivante.

```
{ date: '...',
  tarif: '...',
  _links: {
    self: { href: '/suivis/:nir/:nom_specialite/:adeli/actes
    /:date' },
  }
}
```

**Entité Ordonnance**

- URI de la collection : `/suivis/:nir/:nom_specialite/:adeli/consultations/:date/ordonnances/`. Les méthodes fournies sont GET et POST.
- URI d'un document donné : `/suivis/:nir/:nom_specialite/:adeli/consultations/:date/ordonnances/:num_ordonnance`. Les méthodes fournies sont GET, DELETE et PUT.
- Représentation de l'état est la suivante.

```
{ num_ordonnance: '...',
  text: '...',
  _links: {
    self: { href: '/suivis/:nir/:nom_specialite/:adeli
    /consultations/:date/ordonnances/:num_ordonnance' },
  }
}
```