Exercices API RESTful : conception et implémentation (avec node.js et Express.js)

Paul Schuhmacher

Octobre 2023

Module: API

Exercice 1: Contraintes REST, fondamentaux - Mieux comprendre le cache HTTP

> *Notions abordées : setup environnement node, node.js et express (intro), cache HTTP, utiliser l'onglet Réseau du devtools du navigateur*

Préparation de l'environnement

1. Installer l'environnement d'exécution nodejs
2. Créer un dossier `exercice1` (racine du projet) et créer un dossier `public`. Placez-y un fichier `index.html` contenant la chaîne de caractères `"Hello world"`.
3. A la racine du projet, créer un fichier `package.json` : `npm init`, puis installer express : `npm install express`
4. Créer un fichier `index.js` :

```
const express = require('express')
const app = express()
const port = 3000
//Servir des ressources statiques avec express
app.use(express.static('public', { index: 'index.html' }))
app.listen(port, () => {
    console.log(`Demo REST, servie à l'url: http://localhost:${port}`)
})
```

5. Lancer le serveur :

```
node index.js
```

Rendez-vous sur l'URL `http://localhost:3000` pour tester. Forcer ensuite la suppression du cache.

> *Lorsque vous modifiez votre code javascript côté serveur, pensez à redémarrer le serveur pour prendre les modifications en compte, ou utiliser nodemon pour relancer automatiquement le serveur au changement des sources.*

Manipuler le cache HTTP

1. **Effectuer** une requête pour demander la ressource pointée par `/` à l'aide de votre navigateur favori. Faire la même chose en demandant la ressource sur l'URL `/foo`. **Noter** les *code status* à chaque fois. Requêter à nouveau `/`. **Observer** le *code status*. Que remarquez-vous ?

   Étudier les requêtes avec les dev tools de votre navigateur favori, onglet Réseau (ou Network). Vous pouvez inspecter les requêtes HTTP et les réponses HTTP dans le détail.

2. Qu'est ce que le cache de manière générale ? Qu'est ce que le cache HTTP ?

3. **Parcourir** la page HTTP caching de la MDN qui synthétise la RFC9111 - HTTP Caching (2022) du protocole HTTP/1.1. Où peut-on trouver du cache sur le web ? A quoi servent les headers `Date`, `Last-modified`, `Cache-control` ? **Étudier** les concepts de cache *fresh* (frais) et *stale* (vicié) et de *revalidation*. Qu'est-ce qu'un cache "frais" (fresh) ? Qu'est ce que la validation (ou revalidation) du cache ? Est ce qu'une réponse "viciée/périmée" (*stale*) est forcément *invalide* (nécessite une nouvelle requête pour être mise à jour) ? Quelles sont les deux manières de revalider le cache ?

4. **Requêter** à nouveau l'URL `/` et observer le contenu de la requête HTTP, notamment l'en-tête `Cache-Control`. Que constatez-vous ? Comment expliquer que la réponse soit servie depuis le cache ?

5. **Modifier** le fichier `index.html` (par exemple le texte), et effectuer à nouveau une requête HTTP. Que se passe-t-il ? Inspecter le header `Last-Modified`.

6. **Forcer** le rechargement de la page (*hard reload*) (souvent `Ctr+Maj+r`). Que se passe-t-il ?

7. Citer les deux headers de requête qui permettent de faire de la revalidation (requête conditionnelle). Côté serveur, désactiver le header `Last-Modified`. L'étape de revalidation est-elle encore possible ? Comment ?

8. Utiliser la configuration serveur suivante :

```
app.use(express.static('public', {
    index: 'index.html', maxAge: 20000, etag: false, lastModified: false
}))
```

On s'attendrait à ce que la réponse soit mise en cache pendant 20 secondes, mais ce n'est pas le cas. Lorsqu'on modifie le fichier index.html par exemple et qu'on recharge immédiatement l'onglet du navigateur, le nouveau contenu est servi. Pourquoi (diable) ?

9. Voici une nouvelle configuration :

```
app.use(express.static('public', {
    index: 'index.html', setHeaders: function (res, path, stat) {
```

```
        res.set('Cache-Control', 'no-store')    }   }))
```

Quel résultat a ce header côté client ? Dans quel cas cela pourrait-il être utile ?

10. Quel *caching pattern* (combinaisons et valeurs de Header) utiliser pour servir au client des ressources toujours à jour ?

11. *Bonus* : écrire une fonction middleware avec Express pour écrire un log sur la sortie standard lorsqu'une requête est traitée. Indiquer lorsque cette requête est une requête conditionnelle. *Note: une requête conditionnelle est caractérisée par la présence d'un header If-Modified-Since et/ou If-None-Match*

12. *Bonus* : Trouver un moyen d'afficher l'heure courante mise à jour toutes les 20 secondes uniquement (peu importe les requêtes effectuées entre temps). Par exemple, la réponse du serveur doit être `13:00:00` jusqu'à `13:00:20`, même si vous effectuez une nouvelle requête à `13:00:07`. Réaliser cela en utilisant *uniquement les headers*. Votre code doit *toujours* calculer l'heure courante. Vérifier votre configuration en inspectant les headers et les log de la fonction crée à la question 11.

Voici un point de départ

```
///Retourne l'heure courante dans un format hh:mm:ss
function currentTimeFormatted() {
    const now = new Date();
    const hoursClock = now.getHours().toString().padStart(2, '0');
    const minutesClock = now.getMinutes().toString().padStart(2, '0');
    const secondsClock = now.getSeconds().toString().padStart(2, '0');
    return `${hoursClock}:${minutesClock}:${secondsClock}`;
}
app.get('/funny-clock', (req, res) => {
    /// ??
    res.send(currentTimeFormatted())
})
```

Documentation utile : express.static(root, [options])

Exercice 2 - Design d'une API RESTful

On désire mettre en ligne un service de réservation de billets de concert. Le service ne gère pas de base de données des utilisateurs : un·e utilisateur·ice est simplement identifié·e par un pseudo au moment de la réservation.

Les cas d'utilisation définis sont :

1. L'utilisateur·ice consulte la liste des concerts disponibles
2. L'utilisateur·ice consulte les informations d'un concert
3. L'utilisateur·ice réserve une place pour un concert avec un pseudo
4. L'utilisateur·ice annule sa réservation

5. L’utilisateur·ice confirme sa réservation
6. Le gestionnaire du site consulte la liste des réservations confirmées pour un concert.

Attention, **un utilisateur qui a confirmé sa réservation ne peut plus l’annuler !**

Décrire une API Web RESTful **par des exemples de requêtes/réponses HTTP** permettant de réaliser les cas d’utilisation ci-dessus.

1. Déterminer l’ensemble de données
2. Décomposer l’ensemble de données en ressources
3. Pour chaque ressource:
4. La nommer avec des URI et préciser l’archétype de ressource
5. Implémenter un sous-ensemble de l’interface uniforme (GET, POST, DELETE, PUT)
6. Concevoir la ou les représentations acceptées par les clients, en utilisant la spécification HAL.
7. Concevoir la ou les représentations à mettre à disposition des clients (*formulaires*) sous la forme de pseudo requêtes HTTP

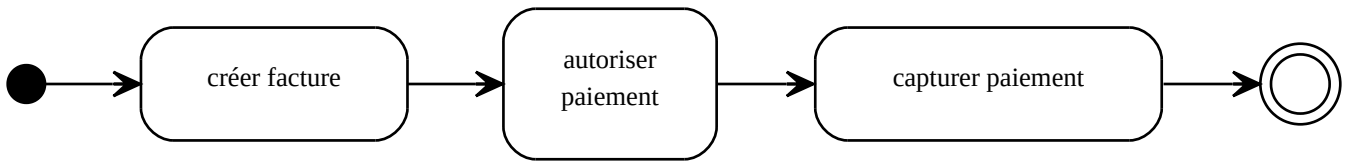
```
POST /login HTTP/1.1

pseudo=ed
password=password
```

4. Envisager la progression typique des évènements: qu’est-ce qui est censé se produire ? Définir les code retours pour chaque requête HTTP
5. Considérer les cas d’erreurs: qu’est-ce qui peut mal se passer ?

Exercice 3 - Design d’une API RESTful

Étant donné le workflow suivant, décrivant le paiement d’une facture :



Les activités sont :

- *1. Créer facture* Création d’un document identifiant les coordonnées bancaires du créditeur et le montant à payer.
- *2. Autoriser paiement* Le débiteur indique ses informations de paiement (numéro de carte de paiement, date limite de validité de la carte et cryptogramme de sécurité). Le client effectue une demande auprès de l’organisme bancaire pour savoir si la transaction est autorisée. L’organisme bancaire fournit un numéro d’autorisation permettant d’identifier la transaction.
- *3. Capturer paiement* Si l’autorisation s’est bien passée, le client réalise une capture, c’est-à-dire qu’il demande à l’organisme bancaire d’enregistrer le paiement et d’effectuer le

transfert du compte du débiteur vers le compte du créditeur.

1. **Proposer une description** d'une API Web RESTful répondant à ces spécifications **par des exemples de requêtes/réponses HTTP**.
2. **Décrire** également les scénarios alternatifs suivants :

- L'autorisation échoue car les informations bancaires fournies sont incorrectes
- L'autorisation échoue car le client ne dispose pas du crédit suffisant sur son compte.

Exercice 4 - Implémentation d'une API RESTful de billetterie de concerts

1. **Implémenter** l'API de l'exercice 2 avec node.js et Express.js. **Concevoir** le schéma de base de données et implémenter la base relationnelle.
2. *Bonus* : **Développer** un ensemble de *ressources* pour qu'un agent *humain* puisse réaliser les cas d'utilisation exposés par l'API (via des pages web)

   Pour réaliser cet exercice, utiliser le starter-pack mis à votre disposition à cette adresse.

Exercice 5 - Implémentation d'une API RESTful de facturation, avec authentification et autorisation par JWT

1. **Implémenter** l'API de l'exercice 3 avec node.js et Express.js. **Concevoir** et utiliser le schéma de base de données relationnelles.
2. **Développer** un ensemble de *ressources* pour qu'un agent *humain* puisse réaliser les cas d'utilisation exposés par l'API (via des pages web)
3. **Mettre en place un système d'authentification et d'autorisation** par **JSON Web Token**. On créera le compte utilisateur à la main *directement en base* via une requête SQL (inutile d'exposer une ressource */sign-up* sur l'API pour le faire)

   Pour réaliser cet exercice, utiliser le starter-pack mis à votre disposition à cette adresse.

Exercice 6 - API RESTful d'un carnet d'adresses

Cet exercice est tiré de l'ouvrage Bien architecturer une application REST (voir ressoures)

Vous êtes en charge du développement d'un service web de carnet d'adresses. Au sein de cette application, les clients doivent pouvoir lire des cartes de visites (coordonnées d'une personne), en ajouter, les modifier ou les supprimer. Il faut également pouvoir créer des *groupes* pour regroupes des fiches (professionnel, famille, amis, etc.). Le service doit permettre d'accéder à une fiche individuelle ou à un groupe de fiches et d'effectuer une recherche par *nom de famille*.

Les *coordonnées* d'une personne sont définies par :

- le prénom
- le nom de famille
- le genre
- le numéro de téléphone
- l'adresse

Ce service doit pouvoir être consommé par des clients *REST*. Par client *REST*, nous entendons un programme, écrit dans un langage quelconque, qui interrogera des URL via le protocole HTTP pour accéder aux données du carnet d'adresses, dans un format à définir (HTML, XML, JSON, etc.). Par défaut, l'API renverra des données au format `application/hal+json` et respectera la convention HAL.

Vous devez implémenter cette API *RESTful* en utilisant l'environnement d'exécution `node.js` et `express.js`.

1. **Déterminer** l'ensemble des données qui seront nécessaires au développement du service. Le *dictionnaire des données* ainsi constitué sera présenté sous la forme d'un tableau avec les colonnes suivantes : *libellé*, *désignation*, *type*, *taille*, *remarques/contraintes*. Compléter donc le tableau suivant :

   > *Le dictionnaire des données sert à la conception de la base de données. Les données recensées y sont donc atomiques (données qui ne peuvent plus être décomposées en données plus petites sans perte d'information).*

Libellé	Désignation	Type	Taille	Obligatoire ?	Remarques/Contraintes
`first_name`	Le prénom d'un contact	A	70	Oui	Aucune
…	…	…	…	…	…

Légende:

- AN : alphanumérique
- N: numérique
- A: alphabétique
- D: Date (et heure)
- B: Booléen

La *taille* s'exprime en nombre de caractères maximum ou de chiffres. Pour une date on compte également le nombre de caractères. Pour les booléens, inutile de préciser la taille.

2. **Décomposer** les données en ressources et **identifier** les relations.
3. **Nommer** les ressources avec des URI et un libellé.
4. **Définir** un sous-ensemble de l'interface uniforme (GET, POST, PUT, DELETE) pour chaque ressource identifiée.

5. On rappelle que l'API doit renvoyer par défaut des données au format `application/hal+json`, en suivant la spécification HAL. **Définir** la ou les représentations acceptées par les clients REST. **Donner** un exemple de données JSON au format `application/hal+json` pour chaque représentation et le code de retour HTTP .
6. **Définir** les représentations acceptées par le serveur pour *modifier* les ressources. Le client REST envoie sa représentation au format `application/x-www-form-urlencoded` (format des données soumises via un formulaire via une balise `<form>`), soit de simples `clef=valeur` dans le corps de la requête HTTP. Pour chaque représentation, fournir une pseudo requête HTTP en utilisant le template suivant :

```
METHODE URL HTTP/1.1

clef=valeur
```

7. **Représenter** le *ressource state* sous forme d'un graphe (l'ensemble des ressources disponibles via leurs URL) (cf cours)

8. **Écrire les pseudo-requêtes/réponses HTTP (avec les code status)** pour

    1. Acceder à une carte du carnet, par exemple celle de Hank Williams
    2. Créer le groupe *Country Legends*
    3. Acceder a un groupe de fiches, par exemple le groupe *Country Legends*
    4. Modifier la fiche d'Hank Williams pour l'ajouter dans le groupe *Country Legends*
    5. Supprimer le groupe *Country Legends*
    6. Supprimer un groupe qui n'existe pas
    7. Créer une carte avec une représentation incompréhensible
    8. Erreur du serveur lors de la demande de la carte d'Hank Williams (par exemple limite de place sur l'espace disque)

9. A partir de votre travail sur le dictionnaire des données et sur les ressources, **concevoir** le schéma de la base de données relationnelle sous la forme d'un modèle conceptuel des données (MCD). Identifier les relations et associations.

    Parmi les opérations sur les ressources exposées par votre API, lesquelles ne sont pas idempotentes ? Pourquoi ?

Implémentation

1. Traduire le MCD en requêtes SQL et **implémenter** la base de données.
2. À l'aide de votre travail de conception, **proposer** une implémentation de l'API répondant au cahier des charges. Une attention particulière sera donnée aux codes de retour HTTP utilisés en cas de succès ou d'erreur (représentation incorrecte, accès à une ressource inexistante). Cette implémentation sera faite avec node.js et express.js. (5pt)

Sécurité et contrôle

- Implémenter l'authentification via JSON web token (JWT) en utilisant la librairie jsonwebtoken et **protéger** les ressources pour modifier ou supprimer une carte de visite.

Pour émuler un compte utilisateur, mettre en dur ses credentials dans le code.

- Sécuriser et limiter les usages de l'API :
  - Un·e utilisateur·ice ne peut pas avoir plus de 1000 contacts
  - Un·e utilisateur·ice ne peut pas effectuer plus de **100 requêtes par minute**
  - Un client non authentifié est banni (identifié par son adresse IP) s'il effectue trois tentatives infructueuses d'authentification
  - Sécuriser le JWT en ajoutant un délai d'expiration
  - Sécuriser le JWT en ajoutant un nombre d'utilisation maximal

Exercice 7 : prise en main de cURL

> *Notions abordées: cURL, lecture et navigation de documentation, cache HTTP*

Pré-requis : installer cURL

Pour l'exercice, nous allons utiliser l'API publique {JSON} Placeholder

1. En une seule instruction avec cURL, récupérer la liste des *posts*. Stocker le résultat dans un fichier `posts.json`. En consultant la documentation, trouver un moyen d'enregistrer à la fois le corps de la réponse (comme précédemment) ainsi que les headers dans le fichier `response.txt` en demandant la ressource `users/1`
2. En une seule instruction avec cURL, récupérer les `users` 7, 8, 9 et 10
3. En une seule instruction avec cURL, récupérer les vingt premières `photos`
4. Écrire une instruction avec cURL qui vous permet de connaître la technologie serveur utilisée par l'API
5. En explorant la documentation de cURL, trouver le moyen d'imprimer le code status de la réponse HTTP
6. Avec cURL, créer une nouvelle ressource de type `post` avec un titre `'Foo'`. Vérifier que la requête a réussi
7. Avec cURL, supprimer la ressource user avec l'id `8`. Inspecter le code status ? Que remarquez-vous ?
8. Comment ajouter un header dans une requête HTTP avec cURL ?
9. En explorant la documentation de cURL, trouver un moyen d'enregistrer l'`ETag` de la réponse HTTP suite à la requête de la ressource *users*. Écrire ensuite une requête conditionnelle utilisant l'`ETag` enregistré précédemment pour ajouter un header `If-None-Match` à la requête HTTP. Que remarquez-vous ? Afficher le code status. Que remarquez-vous ? Que se passerait-il si on modifiait l'ETag stocké précédemment ?

Exercice 7 : cURL et fondamentaux du web humain et programmable

Dans cet exercice, nous allons développer un mini-site web permettant la publication d'objets à vendre (petites annonces). Pour soumettre un objet à la vente, l'utilisateur doit être authentifié. Ce système sera accessible à la fois via un navigateur web et en utilisant l'outil de ligne de commande "cURL."

Étape 1 : Configuration de la Page d'Authentification

Sur un serveur local, **configurez** une page HTML à l'URL `/` qui affiche un formulaire avec les champs "pseudo" et "mot de passe." La page affiche également la liste actuelle des objets en vente. Pour simuler une base de données, nous utiliserons deux fichiers JSON : `users.json` pour les informations des utilisateurs et `items.json` pour les données des objets en vente.

Créez un utilisateur par défaut avec le nom "foo" et le mot de passe "bar."

Étape 2 : Mise en Place de la Gestion de Session

Lorsqu'un utilisateur est authentifié avec succès, maintenez sa session en utilisant un cookie côté client pour le suivi de l'authentification.

Étape 3 : Publication d'un Objet en Vente

Un utilisateur authentifié peut accéder à l'URI `/sell`, où un formulaire permet de saisir les détails d'un objet à vendre. Les champs du formulaire incluent le nom, la catégorie, la description, une image, et le prix en euros. La catégorie peut être choisie parmi un ensemble prédéfini de valeurs (sélection multiple).

L'utilisateur authentifié doit pouvoir soumettre le formulaire pour mettre un objet en vente. Le serveur doit répondre avec une page indiquant "Objet mis en vente avec succès!" et enregistrer les informations dans le fichier JSON `items.json`. Les images doivent être stockées dans un dossier nommé "uploads" à la racine du serveur.

Étape 4 : Récupération de la Liste de Produits en Vente avec curl

Utilisez l'outil "curl" pour écrire une requête qui récupère la liste des produits en vente.

Étape 5 : Authentification avec curl

Écrivez une requête curl pour l'authentification.

Étape 6 : Publication d'une Annonce de Vente avec curl

Écrivez une requête curl pour publier une annonce de vente.

Étape 7 (Bonus) : Publication Automatisée

Écrivez un script utilisant curl qui peut poster 1000 demandes de vente en une seule exécution.

Étape 8 : Sécurisation et amélioration de l'API

Discutez des solutions potentielles pour protéger l'API contre des clients malveillants.

Que faudrait-il faire pour rendre notre site web plus convivial pour les agents non humains ?

Annexe: node.js et express.js, Getting started

```
npm init
npm install express --save
```

Créer le code pour démarrer un serveur

```
// index.js
const express = require('express')
const app = express()
const port = 3000

app.get('/', (req, res) => {
    res.send('Hello World!')
})

app.listen(port, () => {
      console.log(`Demo REST, servie à l'url: http://localhost:${port}`)
})
```

Lancer le serveur

```
node index.js
```

Ressources

Node, Express et libs

- Source des exercices
- nodejs
- Express, framework web minimal pour les applications node.js
- Express, *Hello world*
- Générateur d'application Express
- Routage Express
- En-tête HTTP `Cache-Control`
- Cache headers in Express js app, un bon article qui explique la gestion du cache dans des applications express
- Express, static files
- Express, static files, demo
- Express: meilleures pratiques en production : performances et fiabilité
- mysql.js: Escaping query values

Protocole HTTP

- HTTP Code status 304
- HTTP Caching, une synthèse sur l'implémentation du cache du protocole HTTP. Attention tous les navigateurs n'implémentent pas le standard au même point.
- Un tutoriel de la mise en cache, un très bon tutoriel en français sur la mise en cache du protocole HTTP

Design API

- REST APIs must be hypertext-driven, billet de blog de Roy T. Fiedling très intéressant sur le fait qu'une API RESTful doit être orientée *hypertexte* (ou de manière générale par les *hypermédia*). Concepts fondamentaux à suivre.
- JSON Hypertext Application Language draft-kelly-json-hal-08, HAL representation pour les modèles de données. Une proposition de standard
- API RESTful, spécification des schémas de données *HAL*, les différents types d'hypermédia définis pour le protocole HTTP et pour construire des API plus robustes. Le livre de l'auteur Building Hypermedia APIs with HTML5 and Node, Amundsen, a l'air très intéressant
- API RESTful, spécification des schémas de données *JSON-LD 1.1, A JSON-based Serialization for Linked Data*, une autre spécification des données renvoyées par une API, soutenue et recommandée par le W3C
- Schema.org, *Schema.org is a collaborative, community activity with a mission to create, maintain, and promote schemas for* structured data* on the Internet*. Propose une liste de schémas à suivre pour différents modèles de données
- Microformats wiki, un wiki qui décrit des specifications de structure de données interopérables
- Hydra, Hydra is an effort to simplify the development of interoperable, hypermedia-driven Web APIs. The two fundamental building blocks of Hydra are JSON-LD and the Hydra Core Vocabulary.
- Zalando RESTful API and Event Guidelines
- Bien architecturer une application REST, par Olivier Gutknecht, avec la contribution de Jean Zundel, Eyrolles, 2009

JWT

- JSON Web Token (JWT), la rfc du standard
- RFC 9068: JWT Profile for OAuth 2.0 Access Tokens
- Introduction to JSON Web Tokens, une introduction aux JWT
- Décoder le JWT, une application web pour décoder le contenu d'un JWT

Conception base de données relationnelles

- Le Dictionnaire de Données, établir un dictionnaire de données est une étape fondamentale de tout travail de conception d'une base de données