Développement API - Exercices : Conception et Implémentation (avec Node.js/Express.js)

Paul Schuhmacher

Octobre 2024

Module: Développement API

- Protocole HTTP
- Design API
- Json Web Token (JWT)
- Conception de base de données relationnelles

Exercice 1: Contraintes REST, fondamentaux - Comprendre et utiliser 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 Node.js.
2. Créer un dossier `exercice1` (racine du projet) et, à l'intérieur, créer un dossier `public`. Placez-y un fichier `index.html` contenant la chaîne de caractères `"Hello, world"`.
3. A la racine du projet, initialiser un projet Node.js avec `npm init -y`, puis installer Express.js : `npm install express`.
4. Créer un fichier `index.js` avec le code suivant :

```
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}`)
})
```

> *Nous verrons en détail le fonctionnement d'Express.js dans le Module 3.*

5. Lancer le serveur http :

```
node index.js
```

Rendez-vous sur l'URL `http://localhost:3000` pour tester.

> *Lorsque vous modifiez votre code javascript côté serveur, pensez à redémarrer le serveur pour prendre les modifications en compte, ou utiliser `node --watch` ou nodemon pour relancer automatiquement le serveur au changement des sources (hot reload)*

Manipuler le cache HTTP

1. **Effectuer** une requête `GET` pour demander la ressource à l'URL racine de votre site web `/` 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 ? Que signifie ce code status ?

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 ? Comment est-il géré par votre navigateur ?

3. **Prenez le temps de parcourir** la page HTTP caching de la MDN qui synthétise la RFC9111 - HTTP Caching (2022) du protocole HTTP/1.1. Voir la version traduite en français. Répondez à ce questions :

- Où peut-on trouver du cache sur le web ?
- A quoi servent les headers `Date`, `Last-modified`, `Cache-control` ?
- 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) ? Quels sont les deux mécanismes mis à disposition par le protocole HTTP pour revalider le cache ?

Une mise en commun sera faite lors de la correction de l'exercice.

4. **Requêter** à nouveau l'URL `/` et observer le contenu de la requête HTTP, notamment la valeur de 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 ? Noter la valeur du header `Cache-Control`.

7. Rappeler les deux headers de requête qui permettent de faire de la validation de cache (via des requêtes conditionnelles). Côté serveur, désactiver le header `Last-Modified`. L'étape de revalidation est-elle encore possible ? Si oui, 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 à présent 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) ? *Conseil: vous comprenez bien le cache, c'est le navigateur qui est responsable*

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. Un·e utilisateur·ice est simplement *identifié·e* par un pseudo au moment de la réservation.

Les spécifications du système sont les suivantes :

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 de concert avec son pseudo
4. Le système ne permet de délivrer qu’une seule place (nominale) par personne
5. L’utilisateur·ice annule sa réservation
6. L’utilisateur·ice confirme sa réservation
7. 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 !**

1. Déterminer l’ensemble de données et réaliser un dictionnaire de données
2. Décomposer l’ensemble de données en ressources (ce qui sera exposé par l’API)
3. Pour chaque ressource :
4. La nommer avec des URI et préciser l’*archétype* de la 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. Pour cela, illustrer la réponse avec un document JSON exemple (concert donné, utilisateur·ice donné·e, réservation donné·e);
7. Concevoir la ou les représentations à mettre à disposition des clients (*formulaires*) sous la forme de pseudo requêtes HTTP de la forme:

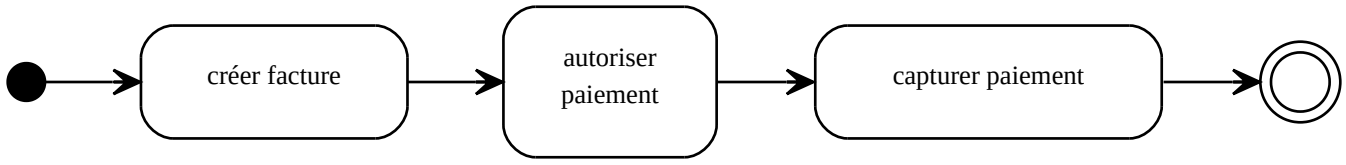
```
METHODE /login HTTP/1.1

clef=valeur
clef2=valeur2
```

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 ? Trouver une solution pour les résoudre.

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. Ignorer dans un premier temps le cas d'utilisation *Le gestionnaire du site consulte la liste des réservations confirmées pour un concert*.
2. *Bonus* : Implémenter le cas d'utilisation *Le gestionnaire du site consulte la liste des réservations confirmées pour un concert*. Pour cela, créer la ressource *La liste des réservations pour un concert* et la protéger par authentification avec un Json Web Token (JWT). Modifier le schéma de votre base de données (si besoin) pour ajouter un mot de passe. Créer un nouvelle ressource *S'authentifier* (`/login`) qui délivrera un JWT **valable 1h**. Pour l'exercice, seul le gestionnaire du site a besoin d'un mot de passe. En base de données, créer un utilisateur gestionnaire de site dont le pseudo est `ed` et possédant le mot de passe `astrongpassword`. Créer le directement en base de données (inutile d'ajouter une ressource *Se créer un compte* ou *Créer un compte pour un admin*). L'accès à la ressource doit être réservée au gestionnaire de site authentifié par le système. Pour mettre en place l'authentification par JWT, utiliser le module jsonwebtoken et des fonctions middleware d'Express
3. *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 cela, déplacer l'ensemble des ressources développées précédemment sur le path `api` et utiliser un moteur de templates d'Express.js (pug, Twig, etc.). Ainsi, le service de billetterie sera utilisable facilement par des agents humains (`/concerts` retourne une page web avec la liste des concerts) *et* par des programmes (`/api/concerts` retourne la liste des concerts sous forme de document JSON)

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

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.*

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.

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 d'authentification infructueuses
    - Sécuriser le JWT en ajoutant un délai d'expiration de 60 minutes
    - Sécuriser le JWT en ajoutant un nombre d'utilisation maximal de 20

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

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 8 : 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 pour un agent humain, et en utilisant cURL comme client HTTP pour un agent non humain (et donc programmable).

Configuration de la Page d'Authentification

1. 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."

Mise en Place de la Gestion de Session

2. Lorsqu'un utilisateur est authentifié avec succès, maintenez sa session en utilisant un cookie côté client pour le suivi de l'authentification. Dans un service web RESTful, pourquoi doit-on maintenir la session dans un cookie ?

Publication d'un objet en vente

3. 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.

Récupération de la liste de Produits en vente

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

Authentification

5. Écrivez une requête curl pour l'authentification.

Publication d'une Annonce de Vente

6. Écrivez une requête curl pour publier une annonce de vente.

Étape 7 (Bonus) : Publication Automatisée

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

Sécurisation et amélioration de l'API

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

9. Que faudrait-il faire pour rendre notre site web plus simple à utiliser pour les agents non humains ?

Exercice 9 : Utiliser une API RESTful officielle

On souhaite écrire un programme client de l'API RESTful de GitHub.

1. Écrire un script Node.js qui permet de :

- Créer un nouveau dépôt `restful-api-exercice9`;
- Lister les commits sur ce dépôt (réaliser des commits sur le README du dépôt avant);
- Ajouter un commentaire sur un commit;
- Comparer deux commits;
- Créer et merger une Pull Request;
- Supprimer le dépôt crée `restful-api-exercice9`;

2. Refaire la même chose en utilisant cette fois cURL.

Exercice 10 : Archétypes et relations entre ressources

Nous souhaiterions créer un service de registre d'état civil enregistrant l'état civil (naissance, décès, mariage et divorce) des personnes résidentes dans une commune.

1. Après un travail d'analyse, nous devons répondre aux spécifications suivantes:
    1. Le système permet d'accéder à l'état civil d'une personne;
    2. Le système permet d'accéder à la liste des états civils;
    3. Le système permet de déclarer la naissance d'une personne;
    4. Le système permet de déclarer le décès d'une personne;
    5. Le système permet de fournir la liste des personnes nées la même année;

Identifier les ressources exposées par le service pour répondre aux spécifications. Pour chaque ressource, **proposer** :

- Son archétype (Document, Controller, Store, Collection);
- Son URL et ses paramètres de requête d'URL si nécessaire
- Son sous-ensemble de l'interface uniforme (GET, POST, DELETE, PUT), i.e méthodes implémentées;

2. Le système permet également d'enregistrer un mariage. Comment représenter cette ressource dans mon système ? De la même manière, comment gérer le cas du divorce ? A-t-on besoin d'une (ou plusieurs) ressource(s) de type *Controller* ?

3. Implémenter le service avec Node.js/Express.js. Utiliser un simple fichier JSON en guise de base de données.

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
```

Lancer le serveur avec un watch des sources (*hot reload*), pratique en développement:

```
node --watch index.js
```

Liens utiles

Node, Express et modules Node.js

- Node.js, le site officiel et sa documentation
- 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

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
- 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
- Source de certains exercices

Json Web Token (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 de 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