Architecture REST

François Guibault, Konstantinos Lambrou-Latreille
École Polytechnique de Montréal

Principes architecturaux de REST

- REST=Representational State Transfer

Traduction libre: *Transfert d'une représentation de l'état (d'une ressource ou application).*

- Principes de base:
  - Utilisation explicite des méthodes HTTP
  - Sans conservation d'état
  - Présente une structure d'URI s'apparentant à une hiérarchie de répertoires
  - Transferts XML, JSON ou les deux (représentation)

Principes architecturaux de REST

- Utilisation explicite des méthodes HTTP
  - Respect de la sémantique de chaque méthode
  - Utilisation des entêtes et des codes de réponses prévus dans le protocole HTTP comme mécanisme de signalisation
  - Utilisation du corps des méthodes (*body ou payload*) pour transporter l'état des ressources et de l'application

Principes architecturaux de REST

- Sans conservation d'état sur le serveur
  - Élimination de la gestion des données liées à l'application sur le serveur
  - Maximisation de la capacité de réponse du côté serveur
  - Possibilité de répartir les requêtes sur plusieurs serveurs
  - À chaque requête, transfert du client vers le serveur de l'information nécessaire pour satisfaire la requête
  - Impose au client de conserver l'état de l'application et de coordonner les services

Principes architecturaux de REST

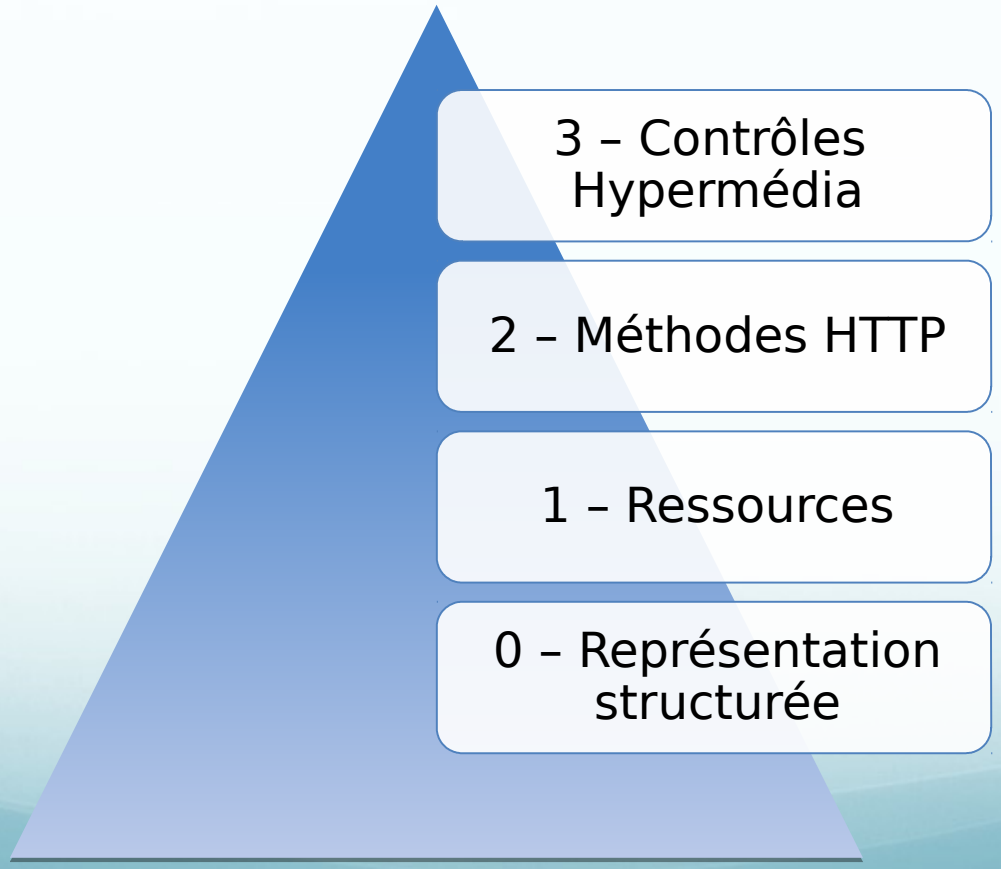
- Présentation d’une structure arborescente de ressources
  - Nomenclature aussi intuitive que possible des ressources
  - Racine unique exposant au premier sous-niveau les ressources principales gérées par le service
  - Représentation des liens entre les ressources comme des niveaux hiérarchiques intermédiaires
  - Hiérarchie indépendante de la structure interne de représentation des données sur le serveur
  - Exemple de ressources :
  - /books, /books/:id/chapters, /users

Principes architecturaux de REST

- Transfert d’état avec XML ou JSON
  - Simplification des modes de représentation de l’état des ressources
  - Permet l’interprétation de la représentation par un humain
  - Indépendance face aux langages de programmation du client et du serveur
  - Tendance actuelle à remplacer XML avec JSON

REST – Niveaux de maturité

Richardson propose un modèle à 4 niveaux de l'utilisation des concepts REST dans un projet:



REST – Niveau 0

- Représentation structurée (XML/JSON/…)
  - Utilisation de HTTP comme mécanisme de transport d'information
  - Permet d'encapsuler des invocations de méthodes distantes (*RPC*) dans un protocole standard
  - Le point d'entrée (URI) sur le serveur (souvent unique) ne représente pas nécessairement une ressource
  - La sémantique des requêtes est interne au serveur et doit être connue des clients

REST – Niveau 0

Réservation d'un rendez-vous avec un médecin

- On commence par obtenir les plages d'horaires du médecin

```
POST /appointmentService HTTP/1.1
[plusieurs autres entêtes]

<openSlotRequest date = "2010-01-04" doctor = "mjones"/>


HTTP/1.1 200 OK
[plusieurs autres entêtes]

<openSlotList>
  <slot start = "1400" end = "1450">
    <doctor id = "mjones"/>
  </slot>
  <slot start = "1600" end = "1650">
    <doctor id = "mjones"/>
  </slot>
</openSlotList>
```

REST – Niveau 0

- On réserve une plage d'horaire du médecin

```
POST /appointmentService HTTP/1.1
[plusieurs autres entêtes]

<appointmentRequest>
  <slot doctor = "mjones" start = "1400" end = "1450"/>
  <patient id = "jsmith"/>
</appointmentRequest>
```

- La réservation est effectuée avec succès

```
HTTP/1.1 200 OK
[plusieurs autres entêtes]

<appointment>
  <slot doctor = "mjones" start = "1400" end = "1450"/>
  <patient id = "jsmith"/>
</appointment>
```

REST – Niveau 0

- Si le médecin n’est pas disponible pendant cette plage

```
HTTP/1.1 200 OK
[plusieurs autres entêtes]

<appointmentRequestFailure>
  <slot doctor = "mjones" start = "1400" end = "1450"/>
  <patient id = "jsmith"/>
  <reason>Slot not available</reason>
</appointmentRequestFailure>
```

REST – Niveau 1

- Ressources
  - Plutôt que d’exposer un point d’entrée unique, le serveur présente une hiérarchie de routes correspondant chacune à une ressource
  - Élimination des paramètres fournis dans l’URL des requête/réponses
  - Remplacement par une arborescence de points d’entrée sur le serveur

REST – Niveau 1

- On commence par obtenir les plages d'horaires du médecin

```
POST /doctors/mjones HTTP/1.1
[plusieurs autres entêtes]

<openSlotRequest date = "2010-01-04"/>
```

- On reçoit les disponibilités du médecin

```
HTTP/1.1 200 OK
[plusieurs autres entêtes]

<openSlotList>
  <slot id = "1234" doctor = "mjones" start = "1400" end = "1450"/>
  <slot id = "5678" doctor = "mjones" start = "1600" end = "1650"/>
</openSlotList>
```

REST – Niveau 1

- On peut réserver une plage d'horaire du médecin

```
POST /slots/1234 HTTP/1.1
[plusieurs autres entêtes]

<appointmentRequest>
  <patient id = "jsmith"/>
</appointmentRequest>
```

- On reçoit les données sur la nouvelle réservation

```
HTTP/1.1 200 OK
[plusieurs autres entêtes]

<appointment>
  <slot id = "1234" doctor = "mjones" start = "1400" end = "1450"/>
  <patient id = "jsmith"/>
</appointment>
```

REST – Niveau 2

- Méthodes HTTP
  - Respect de la sémantique de chaque méthode HTTP
  - Mapping des méthodes HTTP à la sémantique CRUD
  - Élimination des opérations fournis dans l'URL des requête/réponses
  - Utilisation des entêtes et des codes de réponses prévus dans le protocole HTTP comme mécanisme de signalisation
  - Utilisation du corps des méthodes (*body ou payload*) pour transporter l'état des ressources et de l'application

REST – Niveau 2

- Méthode GET
  - Récupérer la représentation d'une ressource (lecture)
  - La réponse normale est un code 200 (OK)
  - En cas d'erreur, le serveur retourne généralement un code 404 (Not Found) ou 400 (Bad Request)
  - L'état de la ressource est envoyée dans le corps de la réponse en XML ou JSON
  - Une requête GET ne devrait jamais modifier une ressource

REST – Niveau 2

- Méthode POST
  - Requête visant à créer une nouvelle ressource, particulièrement des ressources subordonnées à une ressource existante
  - La réponse normale est un code 201 (Created) et une entête Location contenant un lien vers la ressource créée
  - En cas d'erreur, des réponses courantes sont les codes 404 (Not Found) et 409 (Conflict) si la ressource existe déjà
  - Requête non-idempotente, deux requêtes identiques consécutives produisent généralement la création de deux ressources avec la même information

REST – Niveau 2

- Méthode PUT
  - Mise-à-jour de l'état d'une ressource existante
  - Le corps de la requête contient un représentation mise à jour de la ressource
  - La réponse normale est un code 200 (Ok) si l'état de la ressource est retournée ou 204 (No Content) sinon.
  - En cas d'erreur, la réponse habituelle est le code 404 (Not Found)
  - Requête idempotente, deux requêtes identiques consécutives devraient produire le même résultat
  - Si une requête a un effet de bord, il est préférable d'utiliser POST

REST – Niveau 2

- Méthode PATCH
  - Modification de l’état d’une ressource existante
  - Le corps de la requête contient des instruction de modification de la ressource
  - La réponse normale est un code 200 (Ok) si l’état de la ressource est retournée ou 204 (No Content) sinon.
  - En cas d’erreur, la réponse habituelle est le code 404 (Not Found)

REST – Niveau 2

- Méthode PATCH
  - Requête non-idempotente, deux requêtes identiques consécutives ne produisent pas nécessairement le même résultat, mais peuvent être rendues idempotente si formulées de façon à ne pas dépendre de l'état initial de la ressource
  - Si des modifications non-idempotentes sont demandées, une requête conditionnelle devrait être utilisée pour vérifier que la ressource n'a pas été modifiée depuis le dernier accès (utilisation de l'entête ETag)

REST – Niveau 2

- Méthode DELETE
  - Élimination d’une ressource
  - La réponse normale est un code 200 (Ok) si l’état de la ressource est retournée ou 204 (No Content) sinon.
  - En cas d’erreur, la réponse habituelle est le code 404 (Not Found)
  - Requête idempotente, deux requêtes identiques consécutives produisent le même résultat, la ressource est éliminée. Souvent une erreur est générée en réponse à la seconde requête

REST – Niveau 2

- On veut obtenir les plages de disponibilités du médecin

```
GET /doctors/mjones/slots?date=20100104&status=open HTTP/1.1
Host: royalhope.nhs.uk
```

- On reçoit les disponibilités (même que celle du niveau 1)

```
HTTP/1.1 200 OK
[plusieurs autres entêtes]

<openSlotList>
  <slot id = "1234" doctor = "mjones" start = "1400" end = "1450"/>
  <slot id = "5678" doctor = "mjones" start = "1600" end = "1650"/>
</openSlotList>
```

REST – Niveau 2

- On veut réserver une plage d'horaire du médecin

```
POST /slots/1234 HTTP/1.1
[plusieurs autres entêtes]

<appointmentRequest>
  <patient id = "jsmith"/>
</appointmentRequest>
```

- La réservation est effectuée avec succès

```
HTTP/1.1 201 Created
Location: slots/1234/appointment
[plusieurs autres entêtes]

<appointment>
  <slot id = "1234" doctor = "mjones" start = "1400" end = "1450"/>
  <patient id = "jsmith"/>
</appointment>
```

REST – Niveau 2

- Si quelqu’un d’autre a réservé un peu avant nous

HTTP/1.1 409 Conflict
[plusieurs autres entêtes]

```
<openSlotList>
  <slot id = "5678" doctor = "mjones" start = "1600" end = "1650"/>
</openSlotList>
```

REST – Niveau 3

- Contrôles Hypermédia
  - Introduction de mécanismes de guidage de l'application client via des réponses riches fournies par le serveur
  - Dans sa réponse, le serveur fournit au client de l'information sur les actions possibles suite à la requête
  - Le client présente à l'usager ou choisit automatiquement les futures requêtes à effectuer
  - Acronyme HATEOAS (*Hypertext As The Engine Of Application State*)

REST – Niveau 3

- On veut obtenir les plages de disponibilités du médecin

```
GET /doctors/mjones/slots?date=20100104&status=open HTTP/1.1
Host: royalhope.nhs.uk
```

- On reçoit les disponibilités (même que celle du niveau 1)

```
HTTP/1.1 200 OK
[plusieurs autres entêtes]

<openSlotList>
  <slot id = "1234" doctor = "mjones" start = "1400" end = "1450">
     <link rel = "/linkrels/slot/book" uri = "/slots/1234"/>
  </slot>
  <slot id = "5678" doctor = "mjones" start = "1600" end = "1650">
     <link rel = "/linkrels/slot/book" uri = "/slots/5678"/>
  </slot>
</openSlotList>
```

REST – Niveau 3

- On veut réserver une plage d’horaire du médecin

```
POST /slots/1234 HTTP/1.1
[plusieurs autres entêtes]

<appointmentRequest>
  <patient id = "jsmith"/>
</appointmentRequest>
```

REST – Niveau 3

- La réservation est effectuée avec succès
- On obtient les actions possibles

HTTP/1.1 201 Created
Location: http://royalhope.nhs.uk/slots/1234/appointment
[plusieurs autres entêtes]

```
<appointment>
  <slot id = "1234" doctor = "mjones" start = "1400" end = "1450"/>
  <patient id = "jsmith"/>
  <link rel = "/linkrels/appointment/cancel" uri = "/slots/1234/appointment"/>
  <link rel = "/linkrels/appointment/addTest" uri = "/slots/1234/appointment/tests"/>
  <link rel = "self" uri = "/slots/1234/appointment"/>
  <link rel = "/linkrels/appointment/changeTime"
        uri = "/doctors/mjones/slots?date=20100104@status=open"/>
  <link rel = "/linkrels/appointment/updateContactInfo"
        uri = "/patients/jsmith/contactInfo"/>
  <link rel = "/linkrels/help" uri = "/help/appointment"/>
</appointment>
```

REST – Niveau 3

- Avantages
  - Permet au serveur de changer ses URI sans briser les clients
  - Permet aux développeurs d'explorer le protocole
    - Une forme de documentation
    - Par contre, certaines ressources sont ambiguës. On ne sait pas quelle méthode HTTP utiliser
  - Permet aux développeurs du serveur de promouvoir des nouvelles fonctionnalités en ajoutant de nouveaux liens dans les réponses
    - Surtout si le client reste à l'affut de nouveaux liens pour les explorer
- Désavantages
  - Aucun standard de comment représenter les contrôles hypermédias
    - Dans les exemples précédents, les hyperliens sont représentés en utilisant la convention de ATOM (RFC 4287)
  - Réflexion profonde sur la représentation

REST – Un exemple d’API

- Conception d’une application RESTful en une page unique
- https://closebrace.com/tutorials/2017-03-02/creating-a-simple-restful-web-app-with-nodejs-express-and-mongodb

Lignes directrices pour la conception d'une API RESTful

Librement adapté de « **Best Practices for Designing a Pragmatic RESTful API** »

- http://www.vinaysahni.com/best-practices-for-a-pragmatic-restful-api
- Basé sur l'expérience de développement d'une application web de support à la clientèle (Enchant)
- Application permettant aux entreprises de gérer les contacts avec leur clientèle, de fournir du support et d'assurer le suivi des commandes, retours, plaintes, etc.

Conception d'une API RESTful 1 – URL et actions

- Comment identifier les ressources ?
  - Utiliser des **noms** qui ont un sens dans la perspective d'une interface client
  - Standardiser la nomenclature, utiliser le pluriel
  - Ne pas laisser transparaître de détails inutiles sur l'implémentation du service
  - Dans le contexte de Enchant, des noms de ressources appropriés sont: tickets, users et groups

Conception d'une API RESTful 1 – URL et actions

- Comment identifier les actions ?
  - Pour chaque ressource, se demander quelles sont les actions possibles offertes par le service
  - Les principes RESTful fournissent des stratégies pour gérer les actions CRUD à l'aide de méthodes HTTP
  - Par exemple:
    - GET /tickets - Récupère une liste de tickets
    - GET /tickets/12 - Récupère un ticket spécifique
    - POST /tickets - Crée un nouveau ticket
    - PUT /tickets/12 - Met à jour le ticket #12
    - PATCH /tickets/12 - Modifie partiellement le ticket #12
    - DELETE /tickets/12 – Détruit le ticket #12

Conception d'une API RESTful 1 – URL et actions

- Comment traiter les relations entre ressources?
  - Si une relation n'existe seulement qu'à l'intérieur d'une autre ressource, on utilise une hiérarchie de ressources
  - Dans Enchant, un ticket contient une série de messages, cette relation peut être intégrée à la ressource tickets:
    - GET /tickets/12/messages - Récupère la liste des messages du ticket #12
    - GET /tickets/12/messages/5 - Récupère le message #5 du ticket #12
    - POST /tickets/12/messages – Crée un nouveau message pour le ticket #12
    - PUT /tickets/12/messages/5 - Met-à-jour le message #5 du ticket #12
    - PATCH /tickets/12/messages/5 - Modifie le message #5 du ticket #12
    - DELETE /tickets/12/messages/5 - Élimine le message #5 du ticket #12

Conception d'une API RESTful 1 – URL et actions

- Comment traiter les relations entre ressources?
  - Si une relation existe indépendamment d'une ressource, on utilise un identificateur dans la représentation de la ressource
    - Ce choix impose au client de faire une requête spécifique pour obtenir l'information associée à la relation
    - Cette information peut être incluse dans la représentation de la ressource si les deux sont fréquemment récupérées ensemble

Conception d'une API RESTful 1 – URL et actions

- Comment traiter les actions qui n'ont pas d'équivalent CRUD?
  - Pas de réponse universelle
  - Les actions sans paramètres peuvent être représentées comme un champ d'une ressource. Ex.: activate peut être représentée par un booléen « activated » modifié par une requête de type PATCH
  - Utiliser une sous-ressource modifiée par des requêtes CRUD
  - Créer une pseudo-ressource portant le nom de l'action. Ex. pour fournir une action de recherche multi-paramètres, on peut créer une ressource /search

Conception d’une API RESTful 2 – Authentification/encryption

- En production, SSL devrait toujours être utilisé
- Simplifie l’authentification, qui peut reposer sur un échange de jetons d’accès
- Ne pas accepter de connexion sur des URL non encryptées

Conception d’une API RESTful 3 – Documentation

- Rendre la documentation facilement accessible publiquement
- Fournir des exemples complets de cycles requêtes/réponses
- Avertir à l’avance lorsque des changements à l’interface sont prévus. Communiquer via un blog ou une liste de courriel.

Conception d'une API RESTful 4 – Versions

- Toujours associer des numéros de version à une API
- Permet d'itérer plus rapidement et prévient les requêtes invalides à des URI qui ont été mises à jour
- Facilite les transitions lors de changements majeurs en permettant de supporter temporairement les anciennes versions
- Le numéro de version devrait-il faire partie de l'URL ou de l'entête ?
  - Plus transparent dans l'entête
  - Facilite l'exploration à l'aide d'un fureteur dans l'URL
  - Solution mixte: version principale dans l'URL, version mineures dans l'entête

Conception d'une API RESTful 5 – Filtrage, tri et recherche

- Garder les URL de base aussi simples que possible
- Utiliser des paramètres de recherche pour chercher, filtrer et trier les résultats
- Filtrage: utiliser un identificateur unique pour chaque champ supportant un filtre.
  - Ex.: GET /tickets?state=open - Récupère seulement les tickets qui sont ouverts.
- Tri: utiliser un paramètre sort pour décrire les règles de tri.
  - GET /tickets?sort=-priority - Récupérer une liste de tickets en ordre décroissant de priorité
  - GET /tickets?sort=-priority,created_at - Récupérer une liste liste de tickets en ordre décroissant de priorité. Pour une priorité spécifique, les tickets les plus vieux sont placés en premier

Conception d’une API RESTful 5 – Filtrage, tri et recherche

- Recherche: certaines applications exigent des fonctionnalités avancées de recherche et intègrent des technologie tierces-parties
- Lorsqu’utilisées pour récupérer des ressources, ces technologies peuvent être rendues disponible par un paramètre de recherche (p.ex. ‘q’)
  - Ex.: GET /tickets?q=return&state=open&sort=-priority,created_at - Récupérer les tickets ouverts ayant la plus haute priorité qui mentionnent le mot 'return'
- Les résultats de recherche devraient être retournés dans le même format que les requêtes de ressources standards

Conception d’une API RESTful 5 – Filtrage, tri et recherche

- Pour simplifier l’utilisation de l’API, certaines recherches courantes peuvent être encapsulées dans des routes spécifiques sous forme d’URL:
  - La requête

    GET /tickets?state=closed&sort=-updated_at

    qui récupère les tickets fermés récemment, peut être encapsulée dans une route simplifiée

    GET /tickets/recently_closed

Conception d’une API RESTful 5 – Filtrage, tri et recherche

- Permettre de sélectionner les champs retournés dans une ressource
  - Un paramètre de requête fields associé à une liste de noms de champs séparés par des virgules permet au client de sélectionner les champs de la ressource qui doivent être retournés
  - Ex.:
  - GET /tickets?
    fields=id,subject,customer_name,updated_at
  - &state=open&sort=-updated_at

Conception d’une API RESTful 6 – Création et mise à jour

- Les requêtes de création (POST) et de mise à jour (PUT et PATCH) devraient retourner le nouvel état de la ressource
- Une mise à jour peut impliquer des changements dans des champs internes de la ressource (created_at ou updated_at)
- Évite que le client doivent faire une nouvelle requête pour récupérer l’état
- Dans le cas d’une requête POST qui a créé une nouvelle ressource, le code de réponse devrait être 201 et l’entête devrait contenir un champ Location contenant un lien vers la ressource créée

Conception d’une API RESTful 7 – Approche HATEOAS ?

- Le web est basé sur le principe HATEOS, où l’usager décide de sa prochaine interaction avec un site en se basant sur les liens qui lui sont présentés dans la page courante
- Au niveau des API, le principe HATEOS veut que ce soit le serveur qui génère les routes possibles suite à une requête et les transmette au client dans la réponse
- Le client peut alors choisir dynamiquement la prochaine action en se basant sur les routes proposées par le serveur
- Actuellement, il est difficile pour le client de choisir automatiquement les routes, qui sont généralement déterminée au moment d’écrire le code
- Les avantages liés à une génération dynamique des routes ne sont donc pas largement exploités

Conception d’une API RESTful 8 – Format des réponses

- XML est en voie d’être abandonné au profit de JSON comme format de représentation des ressources

Conception d'une API RESTful 9 – Format des requêtes

- Plusieurs API utilisent le même format d'encodage du corps des requêtes que celui utilisé pour encoder les URL
  - Encodage simple, largement supporté
  - N'inclut pas de notion de type et force la conversion des entiers et booléens en chaînes de caractères
  - N'inclut pas de notion de structure hiérarchique
  - Peux suffire dans le cas d'un API simple
- Pour les API plus complexes, le corps des requêtes devrait être formaté en JSON
- Une API qui accepte des requêtes JSON pour les méthodes POST, PUT et PATCH devrait aussi exiger une entête `Content-Type: application/json` ou

Conception d'une API RESTful 10 – Chargement automatique de ressources reliées

- Pour plusieurs requêtes, d'autres ressources, reliées à la ressource demandée, doivent être chargées par le client
- Pour éviter de multiples aller-retour, il peut être tentant de retourner plusieurs ressources dans une seule réponse
- Cette approche n'est pas considérée une bonne pratique
- Pour éviter de déroger aux principes REST, on peut inclure des ressources sur demande explicite du client (requête avec `embed` ou `expand`
  - Ex. `GET /tickets/12?embed=customer.name,assigned_user`
  - Retournerait:

Conception d'une API RESTful 10 – Chargement automatique de ressources reliées

- Pour éviter de déroger aux principes REST, on peut inclure des ressources sur demande explicite du client (requête avec `embed` ou `expand`)
  - Ex. GET /tickets/12?embed=customer.name,assigned_user
  - Retournerait:

```
{
  "id" : 12,
  "subject" : "I have a question!",
  "summary" : "Hi, ....",
  "customer" : {
    "name" : "Bob"
  },
  assigned_user: {
   "id" : 42,
   "name" : "Jim",
  }
}
```

Conception d'une API RESTful 11 – Cache

- HTTP fournit directement des mécanismes de mise en cache, il suffit d'inclure les entêtes appropriées dans les réponses. Deux approches disponibles: ETag et Last-Modified
  - ETag: Inclure dans la réponse une entête ETag contenant un hash ou une somme de la représentation.
  - Doit changer chaque fois que la représentation change.
  - Si une requête contient l'entête If-None-Match avec le hash ou la somme, et que la ressource n'a pas changé, le serveur doit retourner un code `304 Not Modified`
  - `Last-Modified` fonctionne comme le ETag, mais en se basant sur un marqueur de temps, suite à une requête contenant `If-Modified-Since`