Web services REST

2018/2019

2

Introduction

Introduction – Programme – Planning – Validation

Introduction

- Grégory Galli

- Tokidev S.A.S.
  - Consulting, Bureau d’étude, développement informatique.
  - www.tokidev.fr

Tokidev

Avant de commencer

- Posez des questions !
  - Si un point ne vous semble pas clair, n'hésitez pas à me poser des questions.
- Me contacter en dehors du cours
  - Si vous avez besoin d'aide en dehors des horaires de cours, contactez moi par e-mail : greg.galli@tokidev.fr
  - Pour le cours ou vos projets.

Avant de commencer

- Sachez trouver des réponses aux problèmes élémentaires

- Débug
  - Développement web : Inspecteur (Chrome/Firebug)
  - Développement Grails : Débug pas à pas comme en Java

Validation

- Un examen sur la partie cours (1/2 heure)
  - Partie QCM
  - Partie contrôle de connaissances, questions ouvertes
- Une partie pratique à rendre
  - Au moins 6 heures de TD pour avancer dessus
  - A rendre au plus tard le 30 Septembre 2018 à minuit
- Notation du module : 50% examen, 50% projet

Programme

- Présentation Générale
- API REST
  - Principe d’implémentation
  - Verbes HTTP
  - Génération de messages
  - HTTP Status Code
  - Formats d’échange
  - HATEOAS : HAL
  - Exemple détaillé
  - Mécanismes de protection

8

Présentation Générale

REST, c’est quoi

- Né autour des années 2000
- **RE**presentational **S**tate **T**ransfer
- Repose sur HTTP
- Accès à des ressources (URI)
- Utilisation de verbes HTTP
- REST ⇔ RESTful

REST vs SOAP (Simple Object Access Protocol)

- REST
  - Principes architecturaux
- SOAP
  - Spécification d’un protocole de communication standard

REST vs SOAP (Simple Object Access Protocol)

- Deux architectures pour fournir des web services
- Différences importantes
  - Implémentation et de mise en œuvre
  - Lisibilité (enveloppe XML de SOAP)
  - WSDL
    - Format de messages requis
    - Méthodes accessibles
    - Localisation du service

REST vs SOAP – Pros & Cons

- REST
  - (+) Caching (GET HTTP)
  - (+) (x) Stateless
  - (x) Accord obligatoire sur le format
  - (+) Pas de surcharge de header
  - (+) Facile à consommer (XML/JSON)
  - (+) Globalement simple

REST vs SOAP – Pros & Cons

- SOAP
  - (+) WSDL
  - (+) Plus de protocoles supportés
  - (×) Mise en œuvre moins évidente
  - (×) Surcharge globale supérieurs
  - (×) Plus complexe à consommer

REST – Principes / Contraintes

- Segmentation Client (interfaces) – Serveur (données)
- Stateless : Maintien de l'état à la charge du client
- Cache : définition explicite ➔ performances
- Hiérarchisé par couches: répartition et cache partagé
- COD (opt) : Amélioration de la flexibilité
- Interface Uniforme
  - Identification des ressources de manière uniforme
  - Message auto descriptif
  - Manipulation des ressources par des représentations
  - HATEOAS

REST – Avantages

- Segmentation : facilité de maintenance
- Stateless :
  - Indépendance Client – Serveur
  - Pas de congestion via monopolisation par un client
  - Distribution facile à mettre en place
- Exploitation de HTTP (enveloppes / en-têtes), rien à rajouter
- Auto descriptif, facilite la gestion de caches

REST – Inconvénients

- Stateless
  - Gestion Client plus lourde (maintien de l'état)
- Ressources individuelles
  - Plus de requêtes
  - Plus de bande passante

17

API REST – Principe d’implémentation

API REST – Principe d’implémentation

- Définition des ressources et collections de ressources
- Création des ressources
  - Attributs
  - Formats à respecter / Contraintes
- Définition du format d’échange
  - Format unique
  - API multi formats

API REST – Principe d'implémentation

- Sémantique des messages liée aux messages de HTTP
- Les verbes HTTP RFC2616
  - GET : Récupération de la représentation d'une ressource (ou liste)
  - POST : Création d'une ressource (instance)
  - PUT : Mise à jour d'une ressource (instance)
  - DELETE : Effacer une ressource (instance)
  - HEAD : Informations sur une ressource (~GET)
    - NE DOIT PAS avoir de corps de réponse
    - Uniquement des informations META

API REST – Principe d'implémentation

- Utilisation possible de la méthode OPTIONS
  - Partie sous estimée du protocole HTTP
  - Utilisable pour améliorer l'interconnexion des services
  - Rôle : RFC2616

> « This method allows the client to determine the options and/or requirements associated with a resource, or the capabilities of a server, without implying a resource action or initiating a resource retrieval. »

API REST – Echange

- Le Client doit demander un format d'encodage via l'entête « Accept »
- Le Serveur doit confirmer le « Content-type » directement via le header HTTP du même nom

API REST – Verbes HTTP

GET

- Utiliser pour la récupération d'une ressource
  - Cachable
  - Bookmarkable
- Ne jamais utiliser pour modifier quoi que ce soit côté serveur

POST

- Utiliser pour la création d'une nouvelle ressource
  - Peut être utilisé à la place d'un GET si la requête est trop grande (paramètres)
- Ne jamais utiliser pour récupérer une information
  - N'est jamais mis en cache
  - Non bookmarkable

PUT

- Utiliser pour la modification d'une ressource existante
  - Sensé être idempotent
  - Doit être préféré à PATCH même s'il est sensé être plus approprié dans les mises à jour partielles
- Ne jamais utiliser pour récupérer une information
  - N'est jamais mis en cache
  - Non bookmarkable

DELETE

- Utiliser pour supprimer une ressource existante
- Ne jamais utiliser pour autre chose qu'une suppression de ressource
  - N'est jamais mis en cache
  - Non bookmarkable

OPTIONS

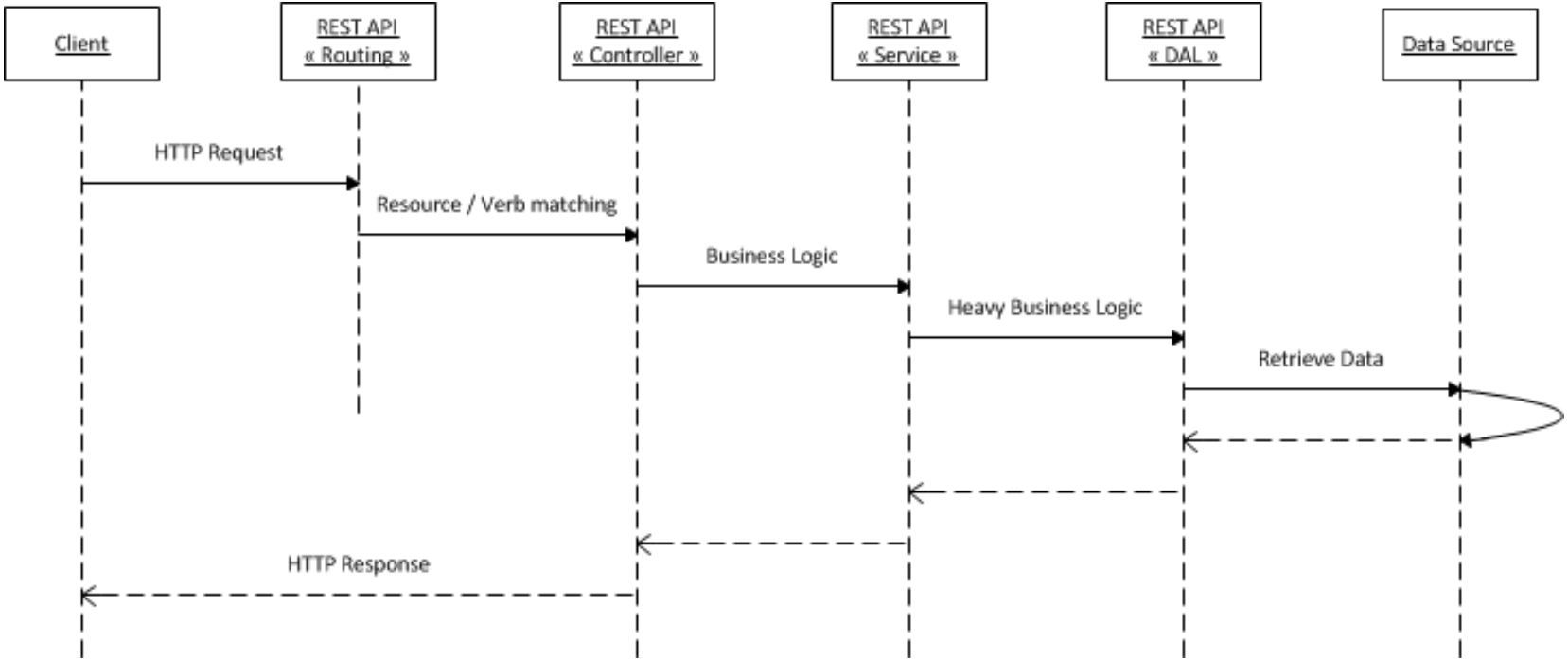
- Doit au minimum retourner une réponse HTTP 200 OK
- Présenter un header « Allow » listant les méthodes utilisables sur la ressource ciblée
- Exemple :

```
200 OK
Allow : HEAD, GET, POST, PUT, DELETE, OPTIONS
```

- Peut aussi retourner d’autres informations

API REST – Les étapes d’un échange

Séquence complète



1. Client – Génération d'un message

- Définition de la ressource demandée
  - Entité
  - Collection d'entités
- Définition de l'action et donc le verbe HTTP à utiliser
  - GET / POST / PUT / DELETE

2. Serveur – Réception et Réponse

- Identification de la ressource demandée
- Identification de la méthode demandée (verbe)
- (Optionnel) Vérification des droits & autorisations
  - Possibilité d'utiliser un token
- Résolution de l'action
- Génération de la réponse
  - Représentation de la ressource (Optionnel)
  - Code d'état (résultat / erreur)

3. Client – Lecture de la réponse

- Décodage du code d'état
  - Code de résultat
  - Code d'erreur
- Si positif (200/201)
  - GET : Analyse de la ressource
  - POST : Validation de création
  - PUT : Validation de modification
  - DELETE : Validation de suppression
- Si négatif (4xx/5xx)
  - Information sur la raison de l'échec si disponible

Exemple d'échange (1)


```
GET /book/1 HTTP/1.1
Accept: text/xml
```

- Echange Client ➔ Serveur
  - Ressource visée : `/book/1`
  - Action demandée : `GET`
  - Encodage demandé : `XML`

Exemple d’échange (2)


```
GET /book/1 HTTP/1.1
Accept: text/xml
```

- Le Client reçoit une réponse du Serveur
  - Décodage du code de résultat
  - Si positif, analyse de la représentation de la ressource dans la réponse

Conclusion – Echange

- Très proche des échanges classiques HTTP
- Facile à implémenter
  - Souvent dans un environnement mobile
    - JSON / XML
    - Format hors standard possible
- Facile à interpréter & debugger
  - Utilisation des outils de développement des navigateurs
  - En console via « curl »
  - Postman

Conclusion – HTTP

- Connaissance vitale de HTTP et sa norme dans son ensemble
  - RFC2616
  - Liste complète
- Codes d'état
  - 200 / 400 / 401 / 403 / 404 / 405 / 500
- Mécanismes de redirection
  - 301

HTTP – Status Codes

Status Codes

- 200 OK / 201 CREATED
  - Information retournée fonction de la méthode utilisée pour la requête
    - GET : L'entitée correspondante est renvoyée dans la réponse
    - POST : Une entitée décrivant ou contenant le résultat de l'action

Status Codes

- 400 Bad Request
  - Requête incompréhensible par le serveur à cause d'une syntaxe incorrecte
- 401 Unauthorized
  - Le service requière une authentification via le header WWW-Authenticate
- 403 Forbidden
  - La requête est correcte mais l'accès à la ressource est interdit

Status Codes

- 405 Method Not Allowed
  - La méthode utilisée n’est pas autorisée pour la ressource identifiée, retourne un header « Allow » avec la liste des méthodes valides
- 500 Internal Server Error
  - Le serveur a rencontré une erreur l’empêchant de traiter la requête
- 404 Not Found
  - Le serveur n’a pas trouvé la ressource demandée

Redirections

- Plusieurs « types » de redirections
- Utilisation courante de redirection 301
  - http://server/library
  - http://server/library/
  - Redirection de l'un vers l'autre afin d'avoir un unique point d'entrée
  - Favorable pour le référencement
  - Plus généralement une bonne pratique

API REST – Formats d’échange

Formats d’échange

- Encodages les plus utilisés
  - XML
    - Plus verbeux
    - Moins facile à lire (humain)
  - JSON
    - Moins facile à produire sans sérialisation (rare)
    - Deux types de structure
      - Objet : Map clef-valeur
      - Collection : Liste de valeurs / Objets

Formats d’échange

- XML

```
▼<library id="1">
   <address>@ home</address>
  ▼<books>
     <book id="3"/>
     <book id="2"/>
     <book id="1"/>
   </books>
   <lat>43.7032472</lat>
   <lng>7.2177977</lng>
   <name>My Library</name>
 </library>
```

- JSON

```
{
    "id": 1,
    "address": "@ home",
    "books":
    [{"id": 3},{"id": 1},{"id": 2}],
    "lat": 43.7032472,
    "lng": 7.2177977,
    "name": "My Library"
}
```

- JSON est plus lisible
- JSON est plus court d’environ 25%
- JSON typé
- Strings uniquement en XML

Formats d’échange – JSON

- Consommation facile
  - Framework JS
- Analyseurs syntaxiques dans les browsers
- Attention aux différences entre navigateurs
  - Utilisation de librairies lissent ces différences

HATEOAS, HAL – Hypertext Application Language

Concept Général : HATEOAS

- Hypermedia As The Engine Of Application State
- Contrainte de REST
- Le client n’a pas besoin de connaissance de l’application déployée
- Segmentation forte du couple client – serveur
  - Evolutions indépendantes

Concept Général : HATEOAS

- Appel à une API REST classique

```
HTTP/1.1 200 OK
Content-Type: application/xml
Content-Length: ...

<?xml version="1.0"?>
<account>
   <account_number>12345</account_number>
   <balance currency="usd">100.00</balance>
</account>
```

Concept Général : HATEOAS

- Appel à une API REST implémentant la contrainte HATEOAS

```
HTTP/1.1 200 OK
Content-Type: application/xml
Content-Length: ...

<?xml version="1.0"?>
<account>
   <account_number>12345</account_number>
   <balance currency="usd">100.00</balance>
   <link rel="deposit" href="https://somebank.org/account/12345/deposit" />
   <link rel="withdraw" href="https://somebank.org/account/12345/withdraw" />
   <link rel="transfer" href="https://somebank.org/account/12345/transfer" />
   <link rel="close" href="https://somebank.org/account/12345/close" />
 </account>
```

HAL, Hypertext Application Language

- Une implémentation parmi d’autres
- HATEOAS implémentable sans forcément respecter HAL
- Voir comme un exemple de mise en œuvre

Principe

- Format simple pour lier des ressources
- Rend l'API explorable
- Plus difficile à mettre en place à première vue
  - Nombreuses librairies pour produire et consommer du HAL

Description

- Ensemble de conventions pour décrire des ressources en JSON ou XML
- Objectifs
  - Passer moins de temps sur le design du format
  - Se concentrer sur l'implémentation et la documentation
- Globalement comme un descripteur d'API via des liens
- On pourrait presque se passer de la documentation et découvrir l'API en l'explorant

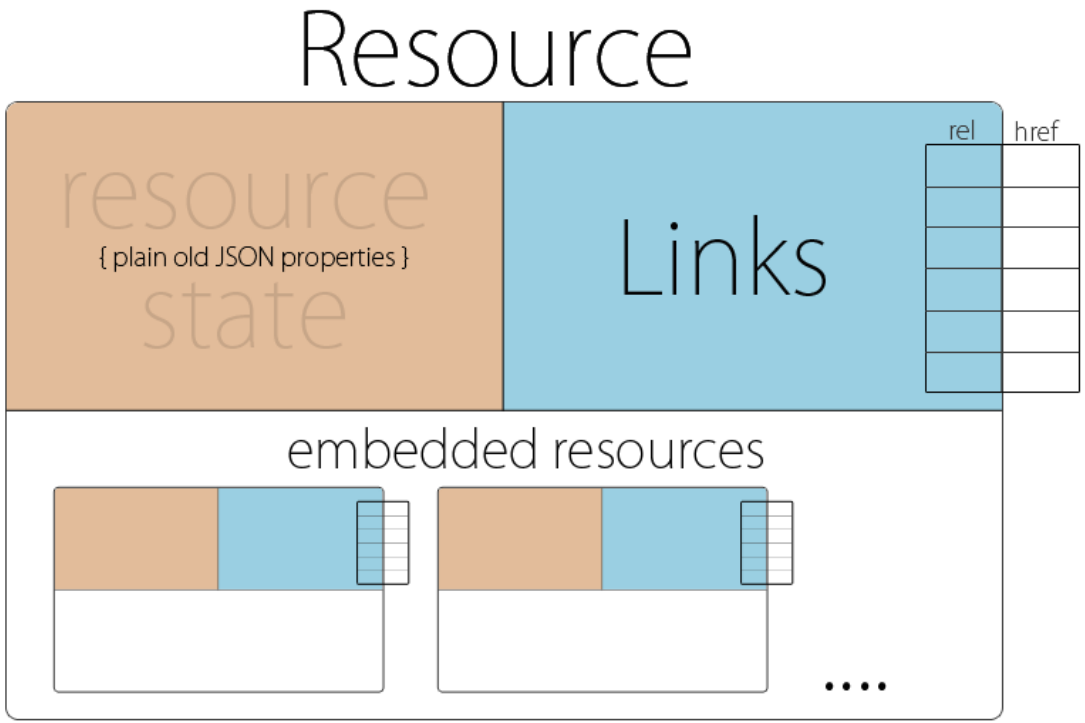
Objectifs

- Fait pour la construction d'API « explorable » via les liens édités
- Les relations doivent permettre d'identifier clairement les ressources disponible et les interactions possible
- Donner un/des point(s) d'entrée au développeur qui va découvrir les liens

Modèle

- Basé sur la représentation de deux éléments
  - Ressource
    - Contient des liens
    - Défini d'autres (sous)ressources
    - Possède un état
  - Lien
    - Désigne une cible (URI)
    - Liée à une relation (rel)
    - Contient des propriétés optionnelles pour la gestion de version et des formats

Modèle


Resource
resource
{ plain old JSON properties }
state
Links
rel
href
embedded resources
....

Exemple

- Ensemble de liens liés à la ressource demandée (self, next, …) (1)
- Propriétés de la collection demandée (2)
- Les « sous-ressources » et leurs liens (3)


```
{
    "_links": {
        "self": { "href": "/orders" },
        "curies": [{ "name": "ea", "href": "http://example.com/docs/rels/{rel}", "templated": true }],
        "next": { "href": "/orders?page=2" },
        "ea:find": {
            "href": "/orders{?id}",
            "templated": true
        },
        "ea:admin": [{
            "href": "/admins/2",
            "title": "Fred"
        }, {
            "href": "/admins/5",
            "title": "Kate"
        }]
    },
    "currentlyProcessing": 14,
    "shippedToday": 20,
    "_embedded": {
        "ea:order": [{
            "_links": {
                "self": { "href": "/orders/123" },
                "ea:basket": { "href": "/baskets/98712" },
                "ea:customer": { "href": "/customers/7809" }
            },
            "total": 30.00,
            "currency": "USD",
            "status": "shipped"
        }, {
            "_links": {
                "self": { "href": "/orders/124" },
                "ea:basket": { "href": "/baskets/97213" },
                "ea:customer": { "href": "/customers/12369" }
            },
            "total": 20.00,
            "currency": "USD",
            "status": "processing"
        }]
    }
}
```


1

2

3

CURIEs


```
"_links": {
    "self": { "href": "/orders" },
    "curies": [{ "name": "ea", "href": "http://example.com/docs/rels/{rel}", "templated": true }],
    "next": { "href": "/orders?page=2" },
    "ea:find": {
        "href": "/orders{?id}",
        "templated": true
    },
```

- Raccourcis pour les liens
- On peux en déclarer plusieurs
- Le « placeholder » {rel} est obligatoire
- L'appel à la méthode « find » se ferait sur
  - `http://example.com/docs/rels/orders{?id}`

Content Type

- application/hal+json
- application/hal+xml

API REST – Exemple détaillé

Modèle utilisé

- Scénario classique avec deux entités
  - Library
  - Book

```
class Library {

    String name
    String address
    Double lat
    Double lng

    static hasMany = [books:Book]
```

```
class Book {

    String name
    String author
    Date dateCreated
    Date lastUpdated

    static belongsTo = [library:Library]
```

Données d’exemple

- Une « Library » et trois « Book »

```
new Library(name: "My Library", address: "@ home", lat: 43.7032472, lng: 7.2177977 )
.addToBooks(new Book(name: "My first book", author: "Serge Miranda"))
.addToBooks(new Book(name: "My second book", author: "Gabriel Mopolo"))
.addToBooks(new Book(name: "My third book", author: "Harris Dahon")).save(flush: true, failOnError: true)
```

Données d’exemple

- Library : identifié via une schéma d’URI du type
  - http://server/library/id
    - ID étant la clef d’identification unique de la ressource
- Ensemble de mes « Library »
  - http://server/library/
  - http://server/libraries/ (plus juste)

Actions possible sur ces ressources

URI	Description	Réponses
GET http://server/library/ GET http://server/libraries/	Récupération de la liste de mes « Library »	200 OK + Représentation de ma collection Autre code pertinent au cas
POST http://server/library/ http://server/libraries/	Création d'une « Library »	201 OK Autre code pertinent au cas
GET http://server/library/id	Récupération de ma ressource par son ID	200 OK + Représentation de ma ressource 404 Resource désignée n'existe pas 401 / 403 / 405 / …
PUT http://server/library/id	Modification de ma « Library »	200 OK 404 Resource désignée n'existe pas 401 / 403 / 405 / …
DELETE http://server/library/id	Suppression de ma « Library »	200 OK 404 Resource désignée n'existe pas 401 / 403 / 405 / …

Actions possible sur ces ressources

URI	Description	Réponses
GET http://server/library/ GET http://server/libraries/	Récupération de la liste de mes « Library »	200 OK + Représentation de ma collection Autre code pertinent au cas

- GET http://server/library/
  - Considéré par certains comme « faux » ou « pas assez précis »
- GET http://server/libraries/
  - Respecte mieux les principes architecturaux
- Certains utilisent indifféremment les deux

Actions possible sur ces ressources

URI	Description	Réponses
GET http://server/library/ GET http://server/libraries/	Récupération de la liste de mes « Library »	200 OK / 404 + Représentation de ma collection
POST http://server/library/ http://server/libraries/	Création d'une « Library »	200 OK 404 Autre code pertinent au cas

- Ces deux premières lignes ciblent une autre « Resource » par rapport aux suivantes car on parle de « l'entité », « la collection » de ressource

Résultats rendus

- GET http://server/libraries/

```
▼<list>
  ▼<library id="1">
      <address>@ home</address>
    ▼<books>
        <book id="3"/>
        <book id="2"/>
        <book id="1"/>
      </books>
      <lat>43.7032472</lat>
      <lng>7.2177977</lng>
      <name>My Library</name>
    </library>
</list>
```

```
▼<library id="1">
    <address>@ home</address>
  ▼<books>
      <book id="1"/>
      <book id="3"/>
      <book id="2"/>
    </books>
    <lat>43.7032472</lat>
    <lng>7.2177977</lng>
    <name>My Library</name>
</library>
```

- GET http://server/library/1

API REST – Mise en œuvre et outils

CURL

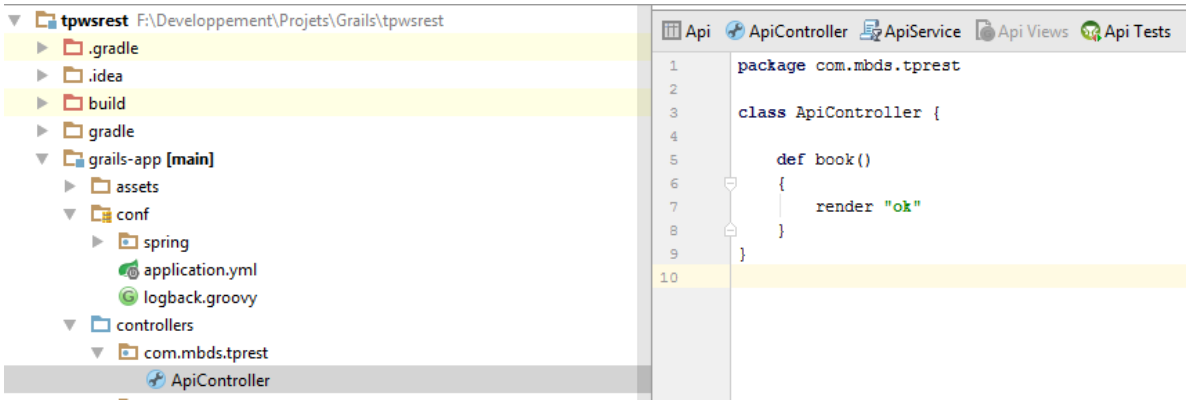
- Client Url Request Library
- Options importantes
  - -X : Verbe
  - -I : Header de la réponse
  - -H : Spécifie le header de la requête
  - -d : Data à envoyer

```
-d '{"title":"Along Came A Spider"}'
```

  - -v : Verbose

Premiers tests

- Création d'un contrôleur « ApiController »
- Création d'une méthode « book » qui renvoi du texte brut



Premiers tests

- On exécute une requête sur notre web service nouvellement créé

```
$ curl -X POST http://127.0.0.1:8081/tpwsrest/api/book/
ok
```

Premiers tests

- On ajuste le code métier afin qu'il réponde à nos besoins

```
def book()
{
    switch (request.getMethod())
    {
        case "POST":
            if (!Library.get(params.library.id)) {
                render (status: 400, text: "Cannot attach a book to a non existent library (${params.library.id})")
                return
            }
            def bookInstance = new Book(params)
            if (bookInstance.save(flush:true))
                response.status = 201
            else
                response.status = 400
        break;
        // Handle other cases here
        default:
            response.status = 405
        break;
    }
}
```

Premiers tests

- On teste ensuite les cas gérés et les cas d'erreur

```
$ curl -X GET http://127.0.0.1:8081/tpwsrest/api/book/ -I
HTTP/1.1 405
X-Application-Context: application:development:8081
Content-Length: 0
Date: Mon, 17 Oct 2016 11:25:29 GMT
```

```
Gav@Gav-Desk ~
$ curl -X POST -I -G "http://127.0.0.1:8081/tpwsrest/api/library/" -d "name=NomDeTest&address=AdresseDeTest&yearCreated=2016"
HTTP/1.1 201
X-Application-Context: application:development:8081
Content-Length: 0
Date: Mon, 17 Oct 2016 12:54:06 GMT


Gav@Gav-Desk ~
$ curl -X POST -I -G "http://127.0.0.1:8081/tpwsrest/api/library/" -d "name=NomDeTest&address=AdresseDeTest&yearCreated=TEST"
HTTP/1.1 400
X-Application-Context: application:development:8081
Content-Length: 0
Date: Mon, 17 Oct 2016 12:54:13 GMT
Connection: close


Gav@Gav-Desk ~
$ curl -X PATCH -I -G "http://127.0.0.1:8081/tpwsrest/api/library/" -d "name=NomDeTest&address=AdresseDeTest&yearCreated=TEST"
HTTP/1.1 405
X-Application-Context: application:development:8081
Content-Length: 0
Date: Mon, 17 Oct 2016 12:54:25 GMT
```

```
$ curl -X POST  -G "http://127.0.0.1:8081/tpwsrest/api/book/" -d "name=NomDeTest&isbn=12AFQF12844636QSD&releaseDate=2014-11-11T01:01:01.00Z&author=TEST&library.id=8"
Cannot attach a book to a non existent library (8)
```

Gestion du format demandé

- Via des méthodes faites pour

```
def libraryInstance = Library.get(params.id)
if (libraryInstance)
{
    withFormat {
        json { render libraryInstance as JSON }
        xml { render libraryInstance as XML }
    }
}
```

- A la main

```
switch (request.getHeader("Accept"))
{
    case "json":
        render libraryInstance as JSON
    break;
    case "xml":
        render libraryInstance as XML
    break;
}
```

Génération automatique de la couche REST

- Via les « Annotations » dédiées

```
@Resource(uri='/books')
class Book {
```

```
@Resource(uri='/libraries')
class Library {
```

```
$ curl -X GET -G "http://127.0.0.1:8081/tpwsrest/libraries/1" -H "Accept: application/json"
{"id":1,"address":"Quai François Mauriac, 75706 Paris","books":[{"id":1},{"id":2}],"name":"François-Mitterrand","yearCreated":1994}
Gav@Gav-Desk ~
$ curl -X GET -G "http://127.0.0.1:8081/tpwsrest/libraries/1" -H "Accept: application/xml"
<?xml version="1.0" encoding="UTF-8"?><library id="1"><address>Quai François Mauriac, 75706 Paris</address><books><book id="1" /><boo
Gav@Gav-Desk ~
$ curl -X GET -G "http://127.0.0.1:8081/tpwsrest/libraries/1.json"
{"id":1,"address":"Quai François Mauriac, 75706 Paris","books":[{"id":1},{"id":2}],"name":"François-Mitterrand","yearCreated":1994}
Gav@Gav-Desk ~
$ curl -X GET -G "http://127.0.0.1:8081/tpwsrest/libraries/1.xml"
<?xml version="1.0" encoding="UTF-8"?><library id="1"><address>Quai François Mauriac, 75706 Paris</address><books><book id="1" /><boo
```

Restriction de formats, Mapping & Ressources liées

- Dans les classes du modèle

```
@Resource(formats=['json', 'xml'])
class Book {
```

```
@Resource(formats=['json', 'xml'])
class Library {
```

```
class UrlMappings {

    static mappings = {
        "/books"(resources: 'book')
        "/libraries" (resources: "library")
        {
            "/books" (resources: 'book')
        }
```

Définition des mimes type

- Dans le fichier de config
  - application.yml

```yaml
grails:
    mime:
        disable:
            accept:
                header:
                    userAgents:
                        - Gecko
                        - WebKit
                        - Presto
                        - Trident
        types:
            all: '*/*'
            atom: application/atom+xml
            css: text/css
            csv: text/csv
            form: application/x-www-form-urlencoded
            html:
              - text/html
              - application/xhtml+xml
            js: text/javascript
            json:
              - application/json
              - text/json
            multipartForm: multipart/form-data
            pdf: application/pdf
            rss: application/rss+xml
            text: text/plain
            hal:
              - application/hal+json
              - application/hal+xml
            xml:
              - text/xml
              - application/xml
```

API REST – Mécanismes de protection

Mécanismes de protection

- L’un des principaux problèmes à gérer lorsque l’on développe une API REST
- Plusieurs solutions, quelques notions à comprendre
  - Authentification
  - Token
  - Signature
  - Requête signée à usage unique
  - HTTPS

Hash

- Empreinte d’un fichier / chaine / etc
- Taille fixe
- Originalement pour les vérifications d’intégrité
- Recalculé à chaque requête d’authentification

Hash - Faiblesses

- Dictionnaires
  - Correspondance mot <> hash
  - 500 mots de passe les plus courants > 75% des utilisateurs
- Brut-force
  - Hash de toutes les combinaisons de caractères possible
  - Quelques secondes si
    - Moins de 6 caractères (lettres, chiffres, caractères spéciaux)

Hash - Faiblesses

- Rainbow tables
  - Correspondances mots de passe <> hash sur des paramètres choisis
    - Nombre max de caractères
    - Caractères spéciaux
    - Chiffres
    - Combinaisons obligatoires

Authentification

- Répond au besoin d’identifier l’émetteur de la requête
- Authentification classique
  - Passer login + mot de passe (hashé) dans la requête
  - Toujours utiliser un « Salt » améliorer la sécurité
- Problèmes
  - Existence de dictionnaires de Hash
  - Si la requête est interceptée on peux s’amuser sur l’API avec les identifiants récupérés

Token

- Principe simple à mettre en place pour limiter les échanges contenant les identifiants
  - On envoi les identifiants pour récupérer un token ayant une durée de vie limitée
  - On utilise ensuite le « Token » pour jouer le rôle d'identifiant
  - Le token peut n'avoir aucun lien avec les identifiants
    - On limite les risques
- Problèmes
  - Si le « Token » est intercepté, on pourra toujours utiliser l'API le temps du timeout du token

Signature

- Consiste à rajouter un paramètre supplémentaire afin de garantir notre identité ainsi que l'intégrité de notre requête
  - Utilisation d'algorithmes de type HMAC
  - Calculé via un « Hash » en combinaison avec une clef secrète
- En cas d'interception, impossible de recréer une signature valide pour une nouvelle requête
- Problème
  - Si la requête est interceptée, elle peut être rejouée

Requête Signée à usage unique

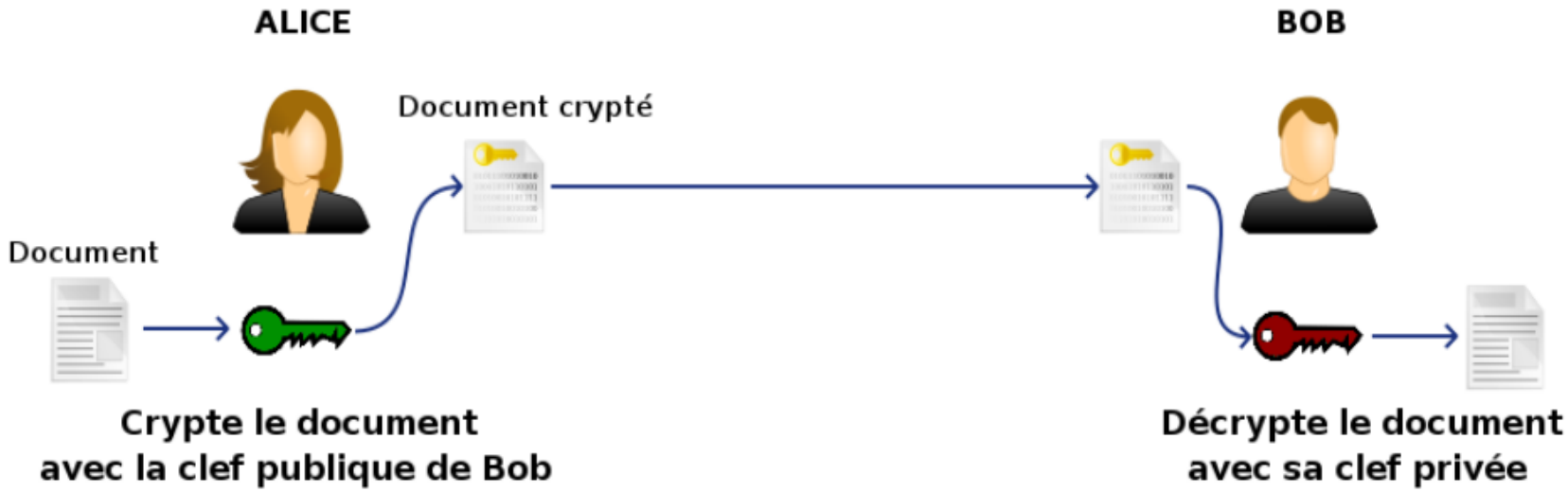
- Utilisé par la majorité des fournisseurs d'API REST
- Stockage supplémentaire du « Timestamp » de la dernière requête côté Client et Serveur et utilisation de ce dernier pour la création de la signature
- Permet de pallier au dernier problème subsistant : même si la requête est interceptée, elle ne pourra pas être rejouée

HTTPS

- Sécurisation des échanges
- Assurance pour le serveur que la requête reçue est identique à celle envoyée
- Assurance que l'émetteur est bien la personne disant avoir transmis la requête
- Chiffrage de bout en bout
  - Man in the middle inutile
- Combiné aux autres méthodes pour plus de sécurité

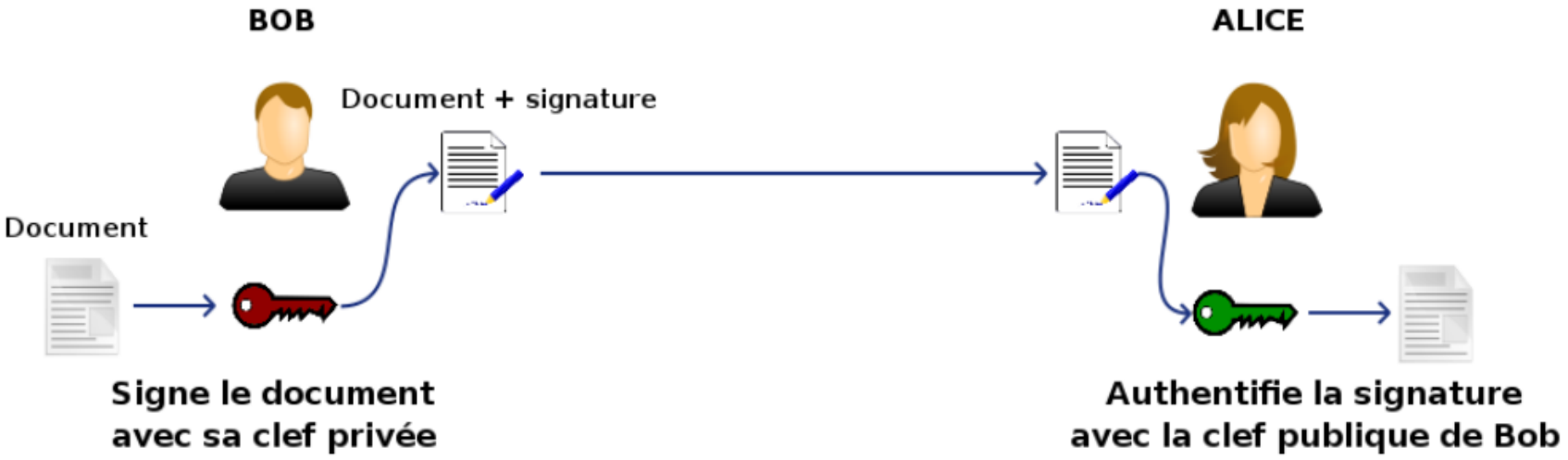
HTTPS

Encryption/Decryption: Alice souhaite envoyer un document à Bob. Elle dispose de la clef publique de Bob. Bob a également cette clef, ainsi que la clef privée qui lui correspond (et qu'il est le seul à avoir).



HTTPS

Signature/vérification: Bob souhaite envoyer un document à Alice. Alice pourra le vérifier et garantir qu'il vient bien de Bob.



Liens

- https://tools.ietf.org/html/draft-kelly-json-hal-08
  - Work in progress sur la spécification
- http://stateless.co/hal_specification.html
  - Condensé des spécifications en cours de validation
- https://en.wikipedia.org/wiki/HATEOAS
  - Définition HATEOAS