Formation APIREST

Ihab ABADI / UTOPIOS

SOMMAIRE

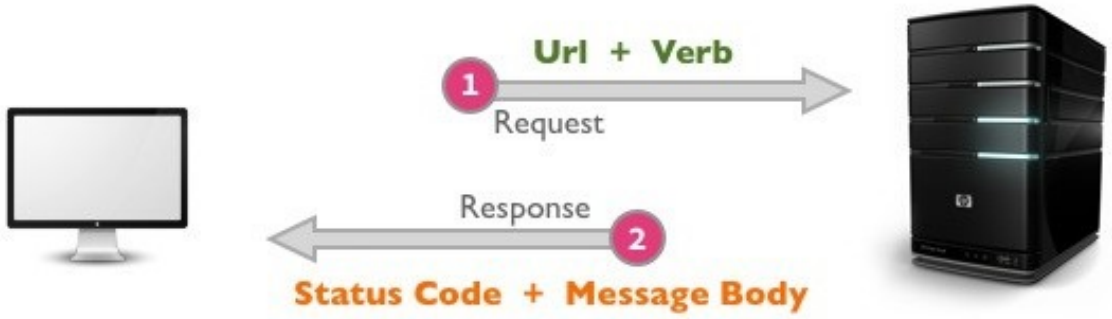
- La communication client/serveur.
- Le Protocol HTTP.
- Les Verbes.
- Les codes de statut.
- Qu’est-ce qu’une Api REST ?
- Les types de retours.
- Une API REST HATEOS
- Modèle de maturité de Richardson.
- Les ressources d’une API REST.
- Les endPoints d’une API REST.
- La sécurisation d’une API REST.
- Comment test une API REST

Communication client/serveur

HTTP : les concepts

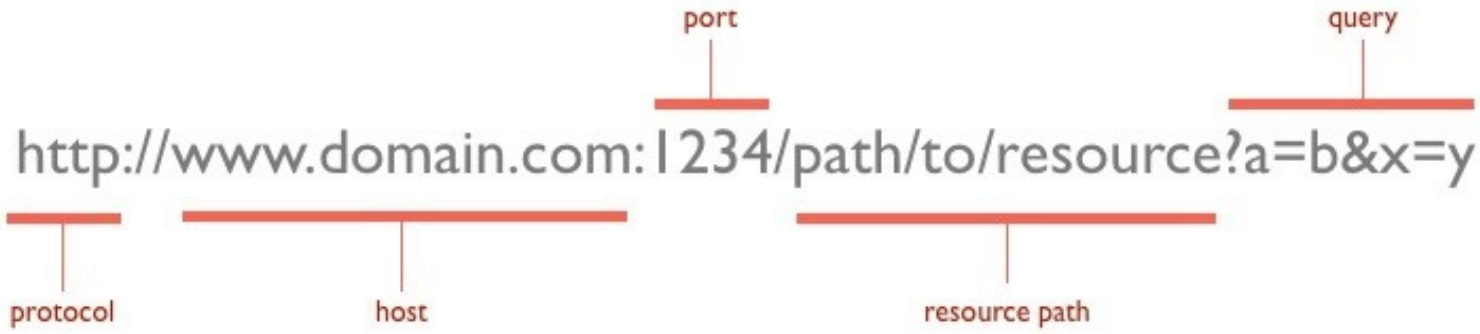
HTTP s'appuie sur 4 concepts fondamentaux :

- le binôme requête/réponse
- les URLs
- les verbes
- les codes de statut.



Communication client/serveur

HTTP : les URLs



Les urls sont à la base du fonctionnement de http car elles permettent d'identifier une ressource :

- **protocol** : le protocole utilisé (http, https, ftp, news, ssh...)
- **host** : nom de domaine identifiant le serveur (FQDN)
- **port** : le port utilisé (80 pour http, 443 pour https, 21 pour ftp)
- **ressource path** : identifiant de la ressource sur le serveur
- **query** : paramètres de la requête.

Communication client/serveur HTTP les verbes

Les **verbes** permettent de manipuler les ressources identifiées par les URLs. Ceux principalement utilisés sont :

- **GET** : le client demande à lire une ressource existante sur le serveur
- **POST** : le client demande la création d'une nouvelle ressource sur le serveur
- **PUT** : le client demande la mise à jour d'une ressource déjà existante sur le serveur.
- **PATCH** : le client demande la mise à jour d'une partie d'une ressource déjà existante sur le serveur.
- **DELETE** : le client demande la suppression d'une ressource existante sur le serveur.

Ils sont invisibles pour l'utilisateur mais sont envoyés lors des échanges réseaux. Chaque requête est accompagnée d'un verbe pour indiquer l'action à effectuer sur la ressource ciblée.

```
GET http://welcome.com.intra/

GET http://www.monsite.fr/index.php
POST http://api.utopios.net/monappli/users/
```

Communication client/serveur
HTTP : les codes de statut

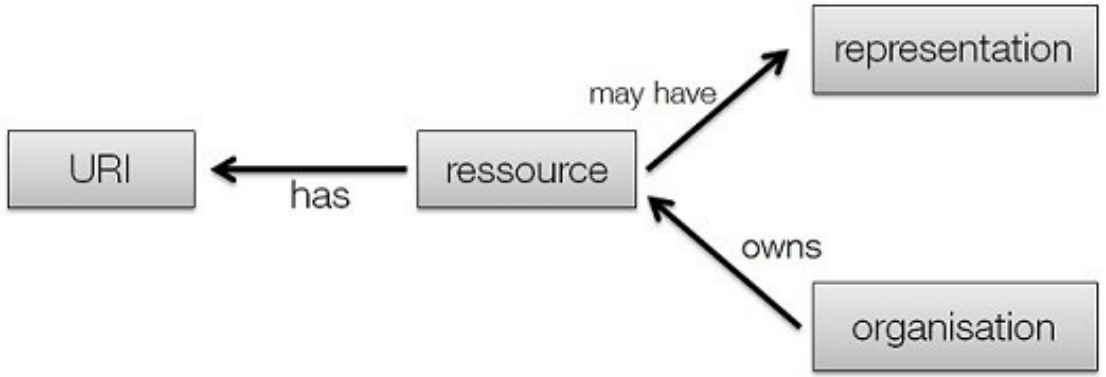
Chaque requête de la part d'un client reçoit une réponse de la part du serveur, comportant un **code de statut**, pour informer le client du bon déroulement ou non du traitement demandé.

Ces codes de statut sont rangés par **plages numériques** :

- **1xx** : message d'information provisoire
- **2xx** : requête reçue, interprétée, acceptée et traitée avec succès
- **3xx** : message indiquant qu'une action complémentaire de la part du client est nécessaire (exemple : redirection vers une autre url)
- **4xx** : erreur du serveur du fait des données en entrée envoyées par le client (exemple : authentification, autorisations, paramètres d'entrée)
- **5xx** : erreur du serveur du fait d'un motif interne au serveur (exemple : indisponibilité d'un composant du serveur, erreur inattendue).

qu'est-ce qu'une API REST ?

Une **API REST (REpresentational State Transfer)** permet à une application d'exposer les services qu'elle offre aux autres applications (pourvues d'une IHM ou pas).



REST s'articule autour de la notion de **ressource** :

- une ressource représente n'importe quel concept (une commande, un client, un message...)
- une représentation est un document qui capture l'état actuel d'une ressource (au format Json, XML, pdf...)
- une ressource appartient à une organisation (une entreprise, un service public...)
- une ressource est accessible via une URI.

Qu'est-ce qu'une API REST ?

HATEOAS

HATEOAS (Hypermedia As The Engine Of Application State) est un pilier de REST, permettant la **découvrabilité (discoverability)** de l'API à partir d'un point d'entrée unique.

Lorsque le serveur envoie sa réponse (la représentation d'une ressource) au client, il doit également ajouter les liens qui permettront au client de **modifier l'état de la ressource** en question ou de **naviguer** vers d'autres ressources.

Conséquences :

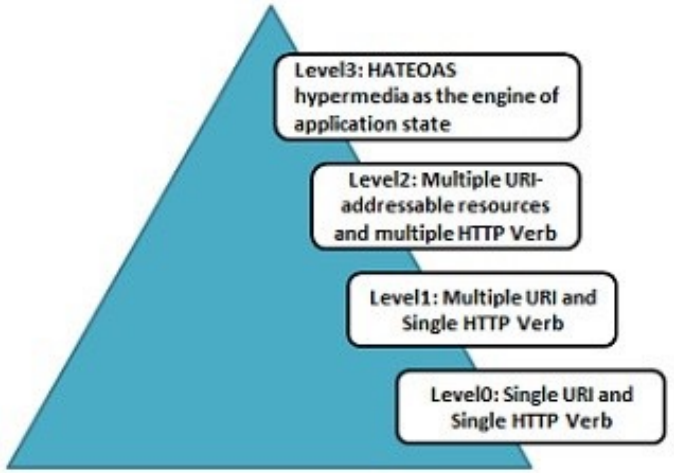
- –plus le message est pauvre (représentation sans hyperlien), plus le client doit être intelligent (connaître ce qu'il peut faire à partir de tel état)
- – plus le message est riche (avec hyperliens), moins le client doit être intelligent car il n'a qu'à suivre ce que lui indique le serveur.

Un site web respecte cette logique avec des liens envoyés par le serveur pour naviguer entre les pages (ressources), dans un format (HTML CSS, images...) lisible facilement par un humain. Entre machines, seules les informations métiers (au format Json par exemple) sont utiles, avec les liens qui les unient.

Qu'est-ce qu'une API REST ?

modèle de maturité de Richardson

Le modèle de Richardson permet de mesurer le degré de maturité d'une API :



Richardson Maturity Model

Qu'est-ce qu'une API REST ?

modèle de maturité de Richardson

- **niveau 0** :
  - utilisation de *HTTP* servant de transport uniquement
  - verbe, URL et code retour uniques

  exemple : webservices SOAP

- **niveau 1** :
  - niveau 0 + *URLs différentes* pour identifier les ressources

  exemple : navigation web

- **niveau 2** :
  - niveau 1 + *verbes HTTP* pour manipuler les ressources + *codes retour* pertinents

  exemple : APIs REST classiques

- **niveau 3** :
  - niveau 2 + HATEOAS (liens)

  vraie API REST idéalement

exemple d'API: ToDoList

1/ identifier les ressources qui constitueront l'API

La plupart du temps on retrouve :

- des ressources **entités** : concepts manipulés par l'API
- des ressources **composites** : agrégation de plusieurs ressources entités en une seule
- des ressources **collections** d'entités ou de composites.

Dans le cas d'une API de **ToDoList**, on peut imaginer avoir les ressources suivantes :

- entités : **ToDoList** et **ToDoItem**
- collections : **ToDoLists** et **ToDoItems**.

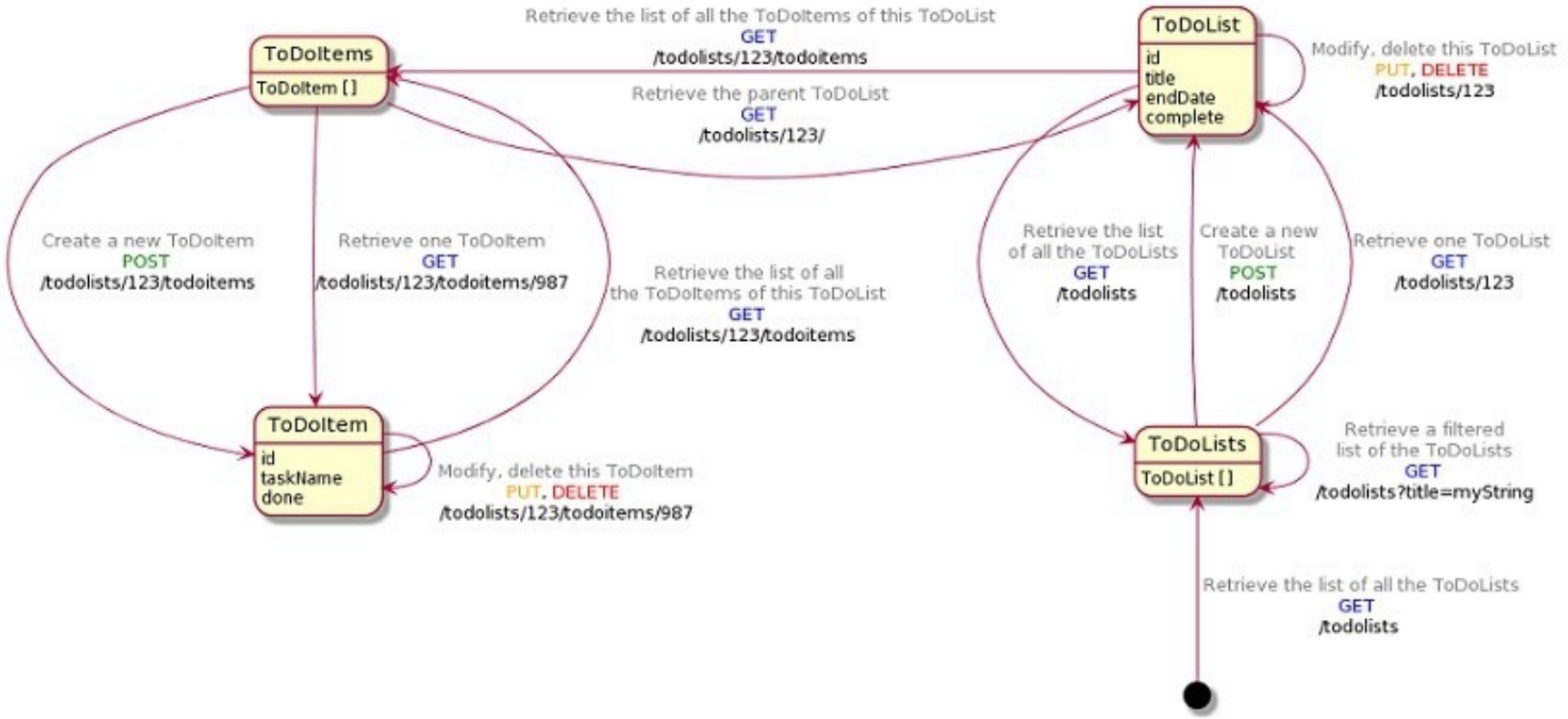
2/ déterminer les URLs de ces ressources et les liens activables

Pour chaque ressource, il faut déterminer :

- comment y accéder (leur URL)
- les actions autorisées :
  - changement d'état de la ressource
  - navigation vers une autre ressource.

exemple d'API: ToDoList

Pour représenter plus clairement les ressources et leur *cinématique*, on peut utiliser un **diagramme d'interactions** :



exemple d'API: ToDoList

exemple de représentation JSON / HATEOAS - HAL

Le client fait la requête suivante (point d'entrée de l'application) pour avoir la liste des **ToDoList** :

```
GET    /todolists
```

Le serveur reçoit la requête, la traite (connexion base de données...), et envoie par exemple, pour chaque **ToDoList** :

- ses informations
- ses liens activables :
  - de changement d'état (modify, delete)
  - de navigation (self, items...).

> ***Note** : par choix de design, les sous-ressources (**ToDoItems**) peuvent n'être envoyées qu'à la demande, lorsque l'on consulte l'une des **ToDoList**. Cela évite de charger une grappe d'objets trop importante.*

exemple d'API: ToDoList

exemple de représentation JSON / HATEOAS - HAL

```
[ # tableau de ToDoLists
  { # première ToDoList de la liste
      # infos complètes ou partielles de la ressource ToDoList
    "id": 123,
    "title": "vacances",
    "endDate": "12/10/2018",
    "complete": false,
      # liens activables pour cette ToDoList (HATEOAS)
    "_links":{
      "self":   {"href": "/todolists/123"},
      "all":    {"href": "/todolists{?title,complete}"},
      "modify": {"href": "/todolists/123"},
      "delete": {"href": "/todolists/123"},
      "items":  {"href": "/todolists/123/todoitems"}
    },
      # sous ressources éventuelles
    "_embedded": {
      "todoitems":[ ... ]
    }

  },
    # deuxième ToDoList de la liste
  { ... }
]
```

exemple d'API: ToDoList

exemple de représentation JSON / HATEOAS - HAL

Le client suit le lien **self** de l'une des **ToDoList** de la liste pour avoir le détail de cette ressource :

```
GET      /todolists/123
```

Le serveur envoie la représentation complète :

- les informations complètes de la **ToDoList**
- les liens activables sur cette **ToDoList** :
  - de changement d'état (modify, delete)
  - de navigation (self, parent, items...)
- éventuellement les informations des sous-ressources (**ToDoItems**) partielles ou complètes
  - éventuellement les liens activables pour ces **ToDoItems**.

> ***Note** : en tant que fournisseur d'API, il faut réfléchir sur les différentes façons d'exposer ses ressources en fonction des possibles usages des clients :*
>
> - *obtenir le détail d'une ressource ou seulement un résumé*
> - *paginer les données*
> - *trier/filtrer selon certains champs*
> - *...*

exemple d'API: ToDoList

exemple de représentation JSON / HATEOAS - HAL

```
# une seule ToDoList
{
  # informations complètes de la ToDoList
  "id": 123,
  "title": "vacances",
  "endDate": "12/10/2018",
  "complete": false,

  # liens activables pour cette ToDoList (HATEOAS)
  "_links": {
    "self":   { "href": "/todolists/123" },
    "all":    { "href": "/todolists{?title}" },
    "modify": { "href": "/todolists/123" },
    "delete": { "href": "/todolists/123" },
    "items":  { "href": "/todolists/123/todoitems" }
  },
  # tableau de ToDoItems (voir page suivante)
  "_embedded": {
    ...
  }
}
```

exemple d'API: ToDoList

exemple de représentation JSON / HATEOAS - HAL

```
    ...
    "_embedded": {
       "todoitems": [ # tableau de ToDoItems
          # premier ToDoItem
          {   # infos et liens activables pour ce ToDoItem
             "id": 987,
             "taskName": "acheter une casquette",
             "done": false,
             "_links": {
                "self": {"href": "/todolists/123/todoitems/987"},
                ...
             }
          },
          # ToDoItem suivant
          {   # infos et liens activables pour ce ToDoItem
             "id": 988,
             "taskName": "prendre la creme solaire",
             "done": true,
             "_links": {...}
          }
       ]
    }
}
```

Sécurisation d’une API Rest

Les 4 principaux concepts de la sécurité sont :

- Confidentialité : rejet des accès non autorisés.
- Intégrité : rejet des modifications non autorisées.
- Disponibilité : lutte contre les dénis de service.
- Non répudiation : capacité à fournir des preuves

Sécurisation d'une API Rest – Identification, Authentification et Autorisation

- Identification : On identifie une entité sans pouvoir en vérifier l'authenticité.
- Authentification : Il est possible de vérifier l'authenticité d'un message, d'une action etc...
  Cela n'implique pas forcément une identification.
  Ex. : Enregistrement d'un pseudo sur IRC.
  Ex. : Facebook's Anonymous Login.
  Ex. : Clés SSH.
- Autorisation : Détermine si une entité a accès à une ressource en fonction des règles définis dans les A.C.L. (Access Control Lists).
  Ex. : Accès autorisé / refusé à une ressource sur une API.
  Ex. : Accès autorisé / masqué / refusé à une propriété d'une ressource.
- Les règles A.C.L. ne sont pas forcément associées à une entité.
  Le porteur d'un "token" n'est donc pas forcément identifié.
  Ex. : Un "token" temporaire partagé avec plusieurs utilisateurs pour accéder à un document

Sécurisation d’une API Rest – Identification, Authentification et Autorisation – Vulnérabilités

- Authentification et autorisation absentes ou insuffisantes.
- Authentification sans autorisation.
- Autorisations trop permissives sur les propriétés en lecture ou en écriture

CSRF – Cross-Site Request Forgery

4 - Under the hood GET /POST /… requests
The browser adds the smurf.com Cookie.

```
DELETE /mushrooms
Cookie:
```

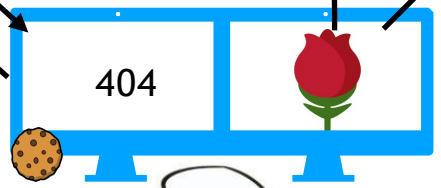
smurf.com

gargamel.com

1 - Log In

2 - Set-Cookie :

3 - Open app
(Smurfette is still signed in to smurf.com even if she closes smurf.com windows)



Sécurisation d'une API Rest – Identification, Authentification et Autorisation – Vulnérabilités

- "Origin" : scheme + FQDN + port.
  https://www.wishtack.com(:443)

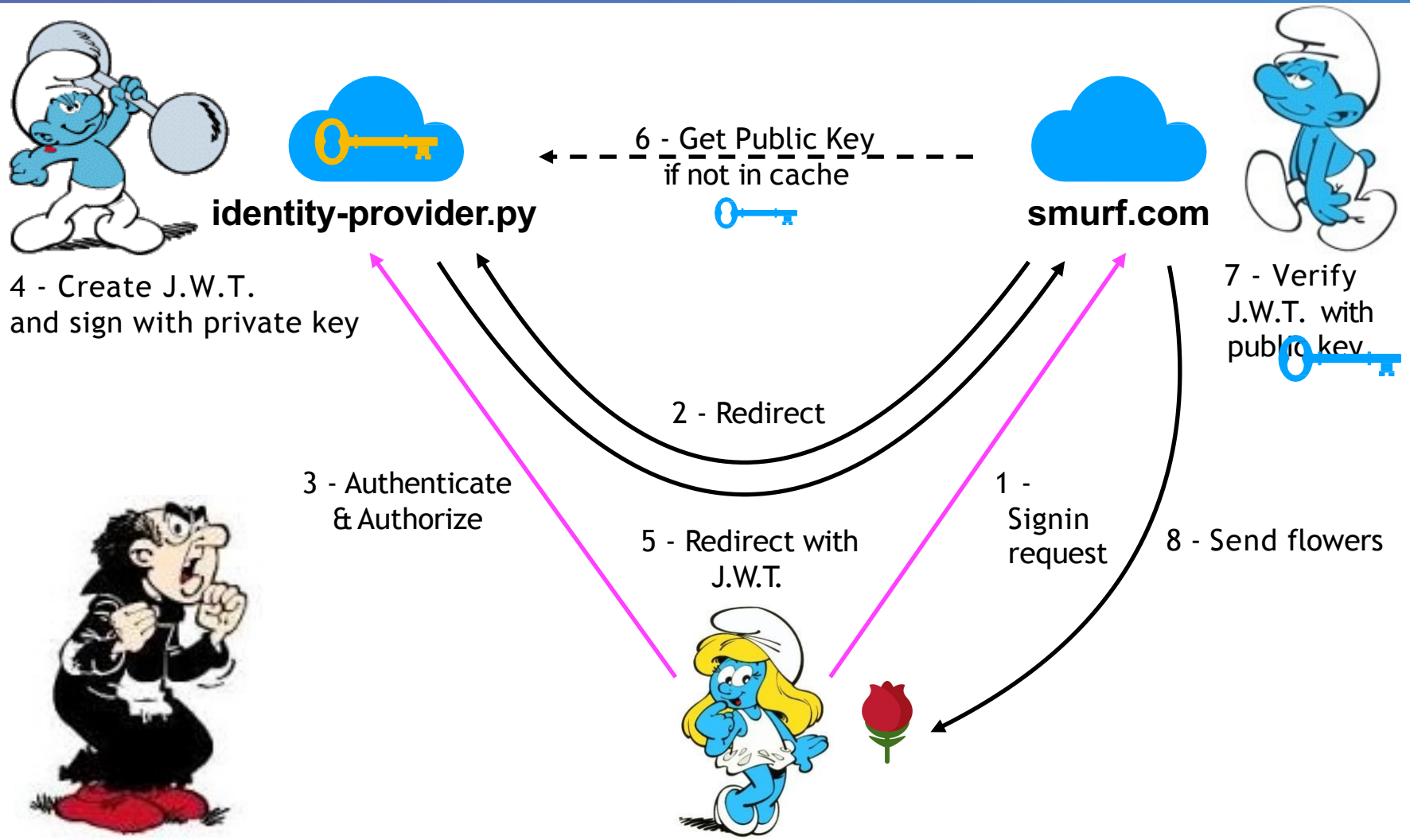
- Le client envoie une preflight request (méthode OPTIONS) indiquant l'"origin", la méthode, les headers **si nécessaire** :
  - Méthode autre que GET / HEAD / POST.
  - POST avec un media type autre que text/plain ou application/x-www-form-urlencoded ou multipart/ form-data.
  - "Headers" modifiés autres que Accept / Accept-Language / Content-Type (Cf. condition précédente) / Content-Language.

- Le serveur répond avec les headers :
  Access-Control-Allow-Origin Access-Control-Allow-Methods Access-Control-Allow-Headers
  Access-Control-Allow-Credentials

J.W.T. JSON Web Token

- J.O.S.E. : Un framework pour échanger des “claims” de manière sécurisée.
  - J.W.K. : JSON Web Key.
  - J.W.E. : JSON Web Encryption.
  - J.W.S. : JSON Web Signature.
  - J.W.T. : JSON Web Token.

J.W.T. Exemple d'utilisation

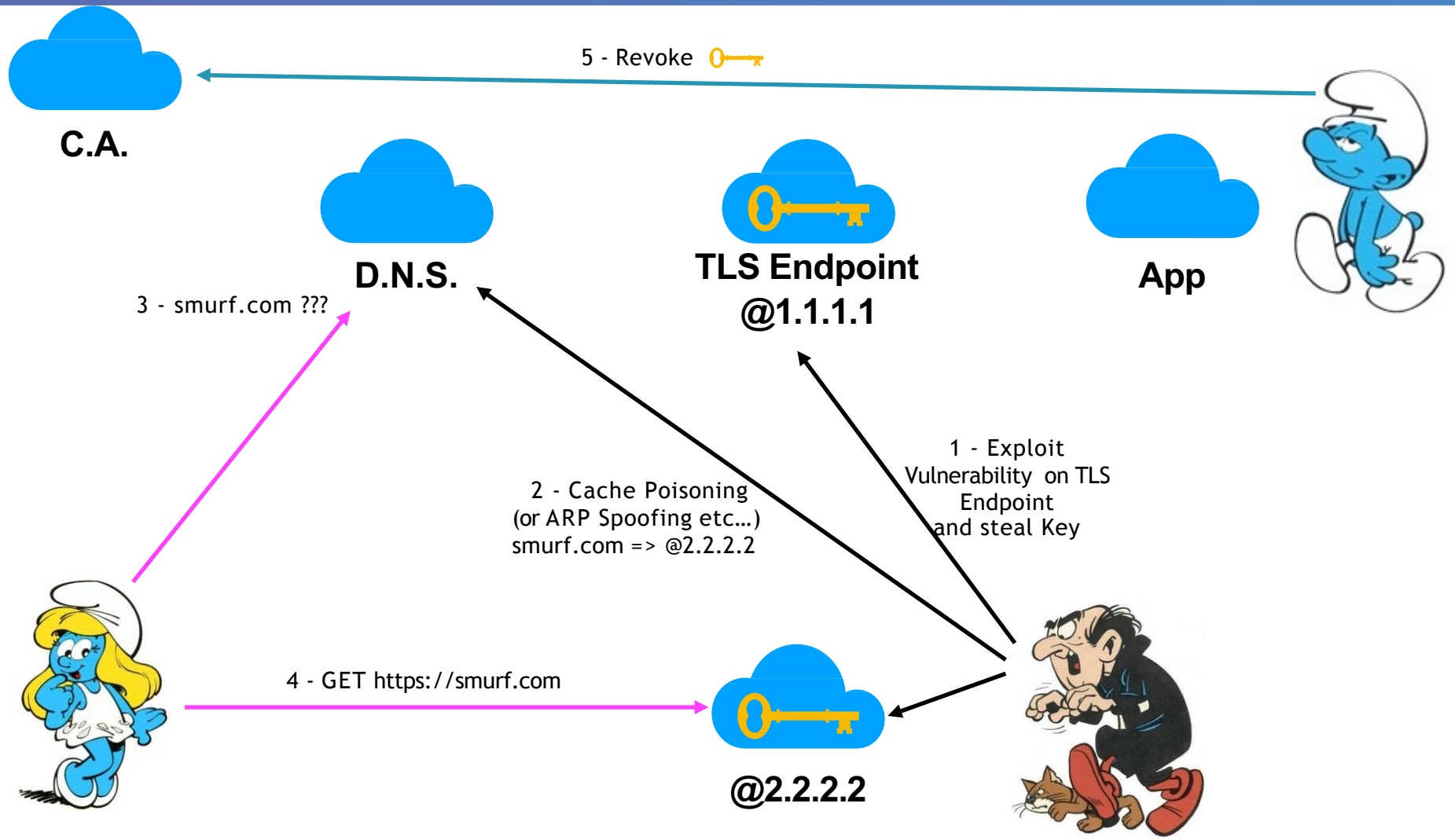


J.W.T. ou cryptographie sans “Key Management”

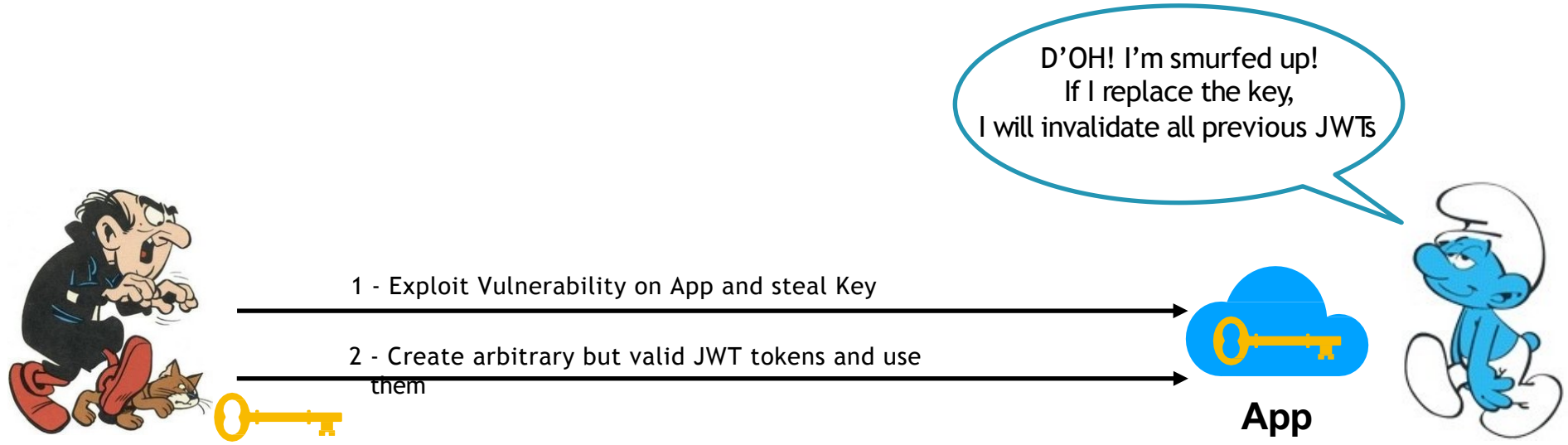
- La cryptographie sans “Key Management”, c’est comme une voiture sans freins, tant qu’on ne cherche qu’à rouler en ligne droite sans s’arrêter, ça répond plutôt bien au besoin…

- Analysons d’abord la sécurité de nos clés TLS.

Quand on vole une clé T.L.S.



Quand on vole une clé J.W.T.



J.W.T. Vulnérabilités classiques

- Stockage non sécurisé.
- `data = jwt.decode(token, key)`
  vs.
  `data = jwt.verify(token, key)`
- Signature stripping: `{alg: 'none'}`
- ~~a~~symétrique ?
- HS256 nécessite une clé de longueur supérieure ou égale à :
  256 bits / 32 bytes / 64 caractères hexadécimaux.
- HS256 vs RS256.
- Selon NIST :
  1024 : RIP 2006
  2048 : RIP 2030
  4096 : ???

J.W.T. Exemple RS256

- Et hop, 1/2 kilo de token :
  eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ9.eyJzdWIiOiJqb2
  huZG9lIiwiaWF0IjoxMzk3MzgzOTA4LCJrZXlpZCI6IjEyMyJ9.
  b0cBt45NzbdAi21VNv_mVtGwYNWRJBkkWNyk_IQDTt6las
  PvXUmbPU-6ou1Yj9FElRCDr7aqjvm7IaQHs0cx0aU5CmBY
  EcvbTo7kNxJkhOtWl2XRU7Mk2zCNJgNK2eEEOHDTT48_U
  MCkHcCGADYzJ5H9mPySeMGTYq4cHGHVbw5v6LRjXYaB
  Xa1jgDfqjTqy5RL2hS19YYaKsoCm5Vsk1tsHAyz4TdqM-
  Ctbuk6AnA57TL_-
  zcx2XbXqv_ztTpdZSA9hLzXVJyFQOj-8hDmNHpZTTFLKfeC
  Ha7HSZfQ1kUGD6AX-
  Kn5Gk3nmaRv7Ox0Dtl-2v15BELiFCLAOFp1acw

J.W.T. Key Rotation

- Les clés doivent être générées et remplacées dynamiquement et régulièrement.
- Les clés ne peuvent pas avoir une durée de vie inférieure à celles des tokens générés.
- Chez Google, une durée moyenne de 3 jours : https://www.googleapis.com/oauth2/v3/certs

Formalisation des APIs avec OpenAPI

spécification d'API : fichier OpenAPI json/yaml

OpenAPI est un métalangage standardisé de description des APIs REST, issu du projet **Swagger**. Un fichier OpenAPI (en JSON ou en YAML) contient la description complète d'une API :

- informations générales sur l'API : version, titre, description
- les protocoles autorisés (HTTP, HTTPS...)
- les représentations des ressources autorisées (JSON, XML...)
- les URLs des ressources, avec pour chacune d'elles :
  - les verbes autorisés
  - une description de l'opération (son but, son usage...)
  - les paramètres d'entrée
  - les réponses possibles avec leurs codes de statut
- la définition des ressources elles-mêmes (objets échangés).

Un ensemble d'outils Swagger s'appuyant sur OpenAPI permettent de générer la documentation, des tests, le squelette de code client/serveur...

Formalisation des APIs avec OpenAPI

écriture d'API : Swagger Editor

Swagger Editor est un outil accessible sur le web. Il est également possible de l'installer en local. Il permet

:

– d'écrire directement le fichier OpenAPI (description de l'API en JSON ou YAML)
– de vérifier sa validité syntaxique
– d'avoir un affichage très lisible et en temps réel, en miroir du code OpenAPI.

> *Note : un fichier OpenAPI étant un fichier texte, il est possible d'utiliser tout éditeur de texte, mais vous n'aurez pas de validation syntaxique ni l'affichage miroir temps réel.*

Formalisation des APIs avec OpenAPI

consultation et test d'API : Swagger UI

Swagger UI est un outil accessible sur le web.

Il permet d'afficher la définition d'une API OpenAPI sous une forme graphique, claire et conviviale. Il suffit de renseigner l'URL d'accès au fichier OpenAPI déployé préalablement sur un serveur. L'outil **Swagger uploader** du programme API permet de le faire facilement.

Le but est de faciliter la compréhension et l'adoption de l'API par les clients.

Swagger UI sert d'**interface entre le fournisseur et le client** de l'API :

- le fournisseur y **publie** son API (fichier OpenAPI) et la fait pointer sur une plateforme de test ou un serveur de bouchons (exemples : Mock-server, SoapUI)
- le client la **consulte**, et peut **tester** son comportement (paramètres d'entrée, données échangées, statuts de retour...).

Formalisation des APIs avec OpenAPI

inspection d'API : Swagger Inspector

Swagger Inspector est un outil accessible sur le web. Il a 2

fonctionnalités principales :

- servir de client HTTP pour tester une API (comme POSTMAN, RESTer, RESTClient...)
- générer un fichier OpenAPI à partir des API testées :
  - lancer les requêtes HTTP souhaitées pour tester l'API
  - ces requêtes sont stockées dans l'historique de l'outil et sont facilement accessibles
  - sélectionner dans cet historique les requêtes qui seront prises en compte pour la génération du fichier OpenAPI.

Formalisation des APIs avec OpenAPI

génération de code client/serveur : Swagger Codegen

Swagger Codegen permet de générer du code client et/ou serveur (comme WSDL2Java pour les webservices Soap) à partir du fichier OpenAPI. Cet outil est utilisable :

- via Swagger Editor, menu **Generate Server** ou **Generate Client**
- via un plugin Maven.

Swagger Codegen supporte plusieurs frameworks Java de gestion d'API, tels que Jersey, Spring Web MVC, CXF. D'autres langages sont également disponibles (PHP, Node.js, Scala, Ruby, Python, Typescript Angular 2...).

La suite de la présentation ne s'appuiera pas sur Swagger Codegen, afin de montrer comment écrire du code Spring MVC pour exposer des services REST.

Tests d'API REST

exemple de réponse de l'API ToDoList

Voici un exemple de réponse retournée en JSON par un **GET /todolists/123** :


```
{    "id": 123,
"title": "vacances",  "endDate": "12/06/2016", "complete": false,
"links": [
    {"rel":"self", "href":"http://localhost/api/v1/todolists/123"},
    {"rel":"all", "href":"http://localhost/api/v1/todolists"},
    ...
],
"todoitems": [
    {"id": 987,
    "taskName": "acheter une casquette", "done": true,
    "links": [
        {"rel":"self", "href":".../api/v1/todolists/123/todoitems/987"},
        {"rel":"allOfList","href":".../api/v1/todolists/123/todoitems"},
        ...
    ]
    }
]
```

tests d'intégration d'API REST