REST

LicencePro 2014

Olivier Perrin
Université de Lorraine

Un peu d'histoire…

Retour en arrière

Le Web a été créé à l'origine pour permettre à des physiciens d'échanger et de gérer leurs articles

XML a été créé pour pouvoir récupérer les documents SGML sur le Web

Deux technologies importantes

Pourquoi ?

Elles ont toutes les deux une logique simple et optimisée pour faire la bonne chose au bon moment

Le Web rend (doit rendre) accessible tous les documents via les URIs (axiome 0, T. Berners-Lee Web Design)

XML rend accessible tous les composants d'un document via les éléments

SOAP 1.2

SOAP est le protocole de base des services Web

SOAP 1.2 est flexible

- nombreux MEP (*Message Exchange Patterns*)
  - synchrone bidirectionnel
  - asynchrone unidirectionnel
  - …
- nombreux protocoles de transport
  - HTTP, SMTP, TCP, MQSeries, …
- nombreuses méthodes (verbes)
- nombreux modèles de nommage/adressage
- nombreuses catégories de données (noms)

Réflexion

Si le Web avait été inventé par Microsoft/IBM/Oracle/Sun/HP

- il n'y aurait pas un mais plusieurs schémas d'adressage
- il supporterait tous les protocoles (dont ceux qui existaient avant) mais on ne pourrait pas naviguer comme on le fait actuellement
- il n'y aurait pas un mais une douzaine de langage de présentation (ML++, .NetML, OraML, JavaML,…)

Question: les services Web ne prennent-ils pas cette direction ?

Par comparaison

Des protocoles simples que tout le monde utilise: HTTP/SMTP/…

Pas très flexibles, mais interopérables

- utilisent des MEPs connus
- fonctionnent tous au-dessus de TCP (ou quelque chose de TCP-like)
- les méthodes/adresses sont définies globalement

Développé par des universitaires/hackers, et non par des entreprises/vendeurs

Solution: standardisé un sous-ensemble SOAP

On pourrait se mettre d'accord sur

- l'utilisation des URIs pour l'adressage
- l'utilisation de XML pour l'encodage des messages
- l'utilisation de SOAP au-dessus de HTTP comme protocole standardisé

Mais une question demeure: comment se mettre d'accord sur la représentation des ressources (*noms*) et des opérations (*verbes*) ?

Standardiser les noms

Il est maintenant démontré qu'il est impossible de mettre tout le monde d'accord sur un vocabulaire commun

Nike n'utilise pas les mêmes types de données que JPMorgan

Les informations financières ne sont pas structurées de la même manière que celles d'un flux RSS destinées aux actualités

Il n'existe pas un seul schéma universel valide pour tout représenter

Standardiser localement les noms

On ne peut pas standardiser de manière universelle, donc

- on va construire des annuaires (*registries/repositories*)
- on va utiliser des transformations (XSLT par exemple)
- on va standardiser la sémantique là où cela est possible (XLink, RDF)

C'est un processus coûteux, mais c'est la seule possibilité

On peut appeler cela une *standardisation locale*

Standardisation des verbes

Intuitivement, il semble qu'il ne soit pas possible de standardiser les verbes

On pourrait alors standardiser localement, i.e. construire

- des annuaires d'interfaces de services
- de transformations entre ces interfaces

Puis standardiser quelques conventions, comme le routage

À nouveau, c'est coûteux, mais indispensable

Oui mais, cette intuition est erronée !

Exemple: les SGBDs relationnels

Que l'on conçoive une base pour la finance, les voyages, les bibliothèques

L'interface entre votre application et les données est basée sur 4 opérations (*CRUD*)

- INSERT/SELECT/UPDATE/DELETE

Vous pouvez construire des abstractions au-dessus, mais ces 4 opérations sont génériques et fixées

Exemple: les systèmes de gestion de fichiers

Que vous soyez

- un artiste qui travaille sur Photoshop
- un mathématicien qui travaille sur les équations diophantiennes dans Mathématica
- un trader qui travaille sur des feuilles de calculs dans Excel

Votre application utilise les mêmes primitives pour lire et écrire les données sur le système de gestion de fichiers

Revenons au Web

Et sur le Web ?

Et bien, on utilise les mêmes *verbes* pour

- consulter un itinéraire sur Mappy
- envoyer des cartes de vœux
- commander des livres
- vendre des CDs

Mais, quels verbes ?

- GET
- POST
- PUT
- DELETE

Services Web basés sur le Web

Que se passe-t-il si on applique les recettes du Web aux problèmes posés par les services Web ?

On pourrait utiliser le protocole HTTP tel qu'il est, et pas comme une couche transport pour un autre protocole (par ex. SOAP)

On pourrait utiliser des opérations standards et universelles, comme GET/PUT/POST/DELETE

Quelles sont les techniques du Web ?

On peut très bien développer sur le Web sans connaître les principes de son architecture

Malgré tout, c'est mieux de les connaître

Quelques pointeurs

- "*Design Issues*", par Tim Berners-Lee
  - http://www.w3.org/DesignIssues/Overview.html
- W3C Technical Architecture Group
  - http://www.w3.org/2001/tag/
- REST, par Roy Fielding
  - http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

HTTP, URL

Vue générale

HyperText Transfer Protocol

- à l'origine, protocole permettant de publier et de retrouver des pages

Version actuelle: HTTP/1.1

Construit au-dessus de TCP, port 80 par défaut (ou 443 en SSL)

Protocole requête/réponse ***sans état***

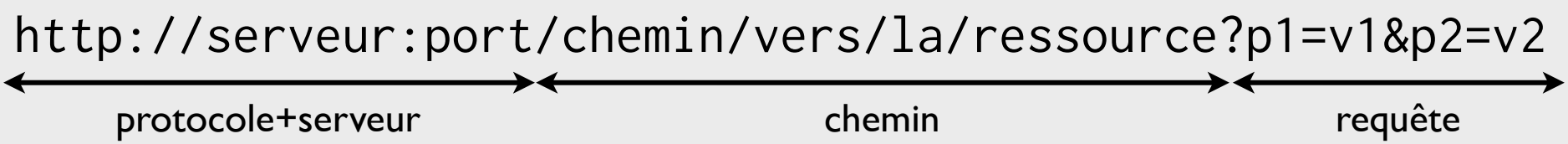
Terminologie

Le serveur est appelé *origin server*

Le client est appelé *user-agent*

Le serveur offre des *resources* qui peuvent être accédées grâce aux URIs

Format des URLs



Requêtes et chemins

Un gestionnaire de requêtes côté serveur (par ex. un servlet) peut gérer plusieurs URLs (éventuellement virtuels)

Exemple (cf slide 118 du cours Servlet)

```
/clients/Londres/MrBean
```

Chemin du servlet (*servlet path*)

```
/clients/*
```

Information (*path info*)

```
Londres/MrBean
```

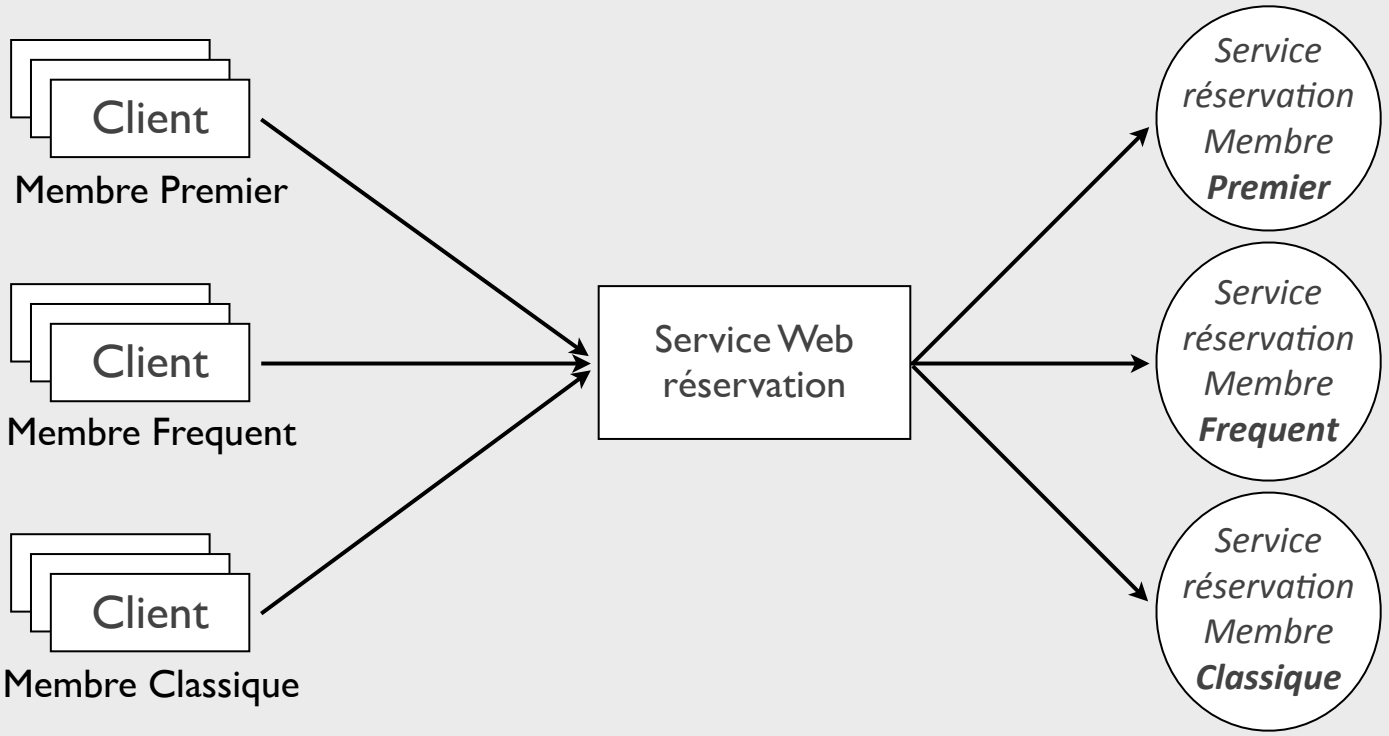
URLs lisibles, largement meilleur que

```
/clients?ville=Londres&nom=MrBean
```

Requêtes et chemins (cont.)

On suppose un service de réservation d'une compagnie aérienne

Approche 1 ([R. Costello])



Requêtes et chemins (cont.)

Inconvénients de cette approche

- les règles changent d'un service à un autre. Le client doit apprendre les règles, et le service doit être écrit pour comprendre les règles
- le service est un point névralgique
- elle viole l'axiome 0 de T. Berners-Lee
- les URLs sont compliqués

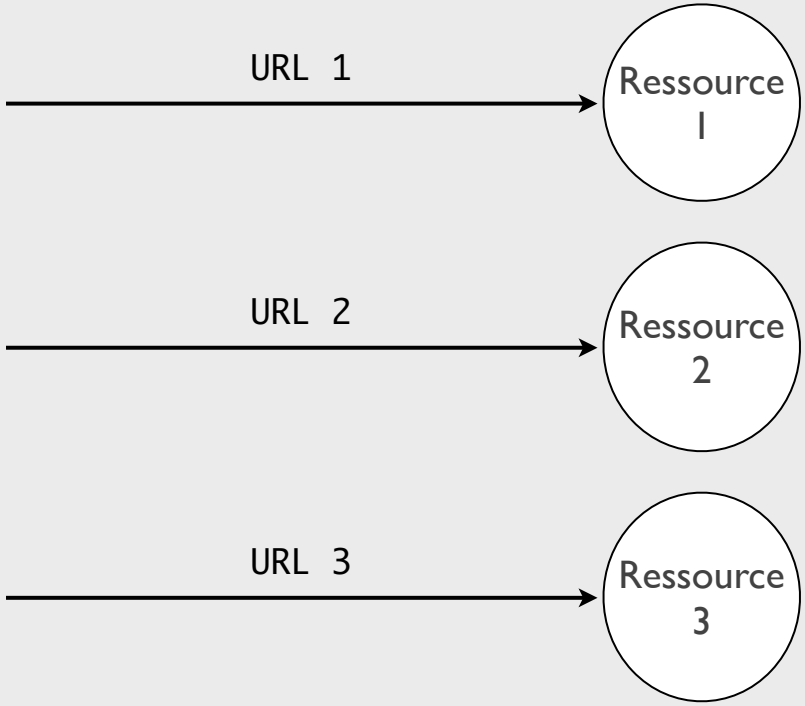
```
www.airCie.com/idClient=45GT76&action=reserv&status=frequent

www.airCie.com/idClient=45XF36&action=reserv&status=premier
```

Requêtes et chemins (cont.)

Web Design (T. Berners-Lee), Axiome 0

"all resources on the Web must be uniquely identified with an URI"



Requêtes et chemins (cont.)

Approche 2: la compagnie aérienne fournit plusieurs URLs, un pour chaque catégorie

Requêtes et chemins (cont.)

Les URLs peuvent être découverts par les moteurs de recherche (ou les annuaires UDDI)

Simple à comprendre (comme le Web)

Pas besoin de règles. Le client sait ce qu'il veut, et sait comment y accéder

Équilibrage facile: un serveur rapide pour les membres premiers

Pas de point névralgique

Cohérent avec l'axiome 0

HTTP

HyperText Transfer Protocol

- à l'origine, protocole permettant de publier et de retrouver des pages

Version actuelle: HTTP/1.1

Construit au-dessus de TCP, port 80 par défaut (ou 443 en SSL)

Protocole requête/réponse ***sans état***

HTTP est *stateless*

- cookies (*client-side*)
- session (*server side*)

HTTP: format requête/réponse

Basé sur du texte

Requête

- Méthode + chemin + version HTTP
- Hôte
- Ligne vide
- Contenu optionnel

Réponse

- Version HTTP + code d'état
- Entêtes
- Ligne vide
- Contenu optionnel

HTTP: exemple requête/réponse

Requête

```
GET /index.html HTTP/1.1
Host: www.example.com
```

Réponse

```
HTTP/1.1 200 OK
Date: Mon, 10 November 2008 08:38:34 GMT
Server: Apache/1.3.27 (Unix)  (Red-Hat/Linux)
Last-Modified: Wed, 02 Jan 2008 23:11:55 GMT
Accept-Ranges: bytes
Content-Length: 10
Connection: close
Content-Type: text/html; charset=UTF-8

<html><body>Hello world!</body></html>
```

HTTP: méthodes

Quelles sont les méthodes HTTP ?

- pour récupérer la représentation d'une ressource: GET
- pour créer une nouvelle ressource: PUT à un nouvel URI, POST à un URI existant
- pour modifier une ressource existante: PUT à un URI existant
- pour supprimer une ressource existante: DELETE
- pour obtenir les méta-données d'une ressource existante: HEAD
- pour connaître les verbes que la ressource comprend: OPTIONS

HTTP: codes d'état

Quels sont les codes d'état de HTTP ?

- 1xx : méta-données
- 2xx : tout va bien
- 3xx : redirection
- 4xx : le client a fait quelque chose de pas correct
- 5xx : le serveur a fait quelque chose de pas correct

HTTP: codes d'état (cont.)

100 – Continue

200 – OK, 201 – Created, 202 – Accepted, 203 – Non Authoritative Information, 204 – No Content, 206 – Partial Content

300 – Multiple Choices, 301 – Moved Permanently, 303 – See Other, 304 – Modified, 307 – Temporary Redirect,

400 – Bad Request, 401 – Unauthorized, 403 – Forbidden, 404 – Not Found, 405 – Method Not Allowed, 406 – Not Acceptable, 409 – Conflict, 410 – Gone, 411 – Length Required, 412 – Precondition Failed, 413 – Request Entity Too Large, 414 – Request URI Too Long, 415 – Unsupportable Media Type

500 – Internal Server Error, 501 – Not Implemented, 502 – Bad Gateway, 503 – Service Unavailable

HTTP: headers

Les headers HTTP à connaître

- Authorization: contient les certificats de sécurité (basic, digest, …)
- Content-Length: longueur du contenu
- Content-Type: le format de représentation de la ressource
- Etag/If-None-Match: checksum, pour les GET conditionnels
- If-Modified-Since/Last-Modified: pour les GET conditionnels
- Host: domaine de l'URI
- Location: localisation d'une ressource créée/déplacée
- WWW-Authenticate: authentification attendue
- Date: obligatoire, estampille pour les requêtes et les réponses

HTTP: headers (cont.)

Les headers HTTP utiles

- Accept: le client précise au serveur quel format il désire
- Accept-Encoding: le client précise au serveur quel format de compression il comprend
- Content-Encoding: équivalent à Accept-Encoding côté serveur
- Allow: le serveur envoie au client les verbes permis
- Cache-Control: transmet aux caches la façon de cacher (ou non) la représentation de la ressource
- Content-MD5: checksum cryptographique du contenu

HTTP: headers (cont.)

- Expect: le client demande s'il peut continuer en attendant un 100-Continue
- Expires: le serveur précise au client que la représentation peut être placée dans un cache jusqu'à une certaine date
- If-Match: utilisé pour les comparaisons de ETag
- If-Unmodified-Since: utilisé pour les PUT/POST conditionnels
- Range: spécifie la partie d'une représentation que le client désire (GET conditionnel)
- Retry-After: la ressource ou le serveur sont indisponibles pendant le délai spécifié
- Content-Location: le header donne l'URI canonique de la ressource

HTTP: extensions

HTTP/1.1 permet d'inclure des extensions qui utilisent de nouvelles méthodes

Par exemple, WebDav expose les dossiers grâce à HTTP

REST

Representational state transfer (REST) is an architectural style consisting of a coordinated set of architectural constraints applied to components, connectors, and data elements, within a distributed *hypermedia* system. REST ignores the details of component implementation and protocol syntax in order to focus on the roles of components, the constraints upon their interaction with other components, and their interpretation of significant data elements.

http://en.wikipedia.org/wiki/Representational_state_transfer

Introduction sur REST

REpresentational State Transfer

- Roy Fielding, thèse soutenue en 2000
- http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

C'est un style d'architecture

Ce n'est pas un standard

C'est une émanation de la façon dont fonctionne le Web

C'est une alternative à SOAP

Composition

Data elements

- ressources et identifiants, représentations, méta-donnée, données de contrôle

Connectors

- client, serveur, cache, resolver, tunnel

Components

- user agent, origin server, gateway, proxy

REST et les standards

REST utilise les standards existants

- HTTP
- URIs/URLs
- XML, HTML, PNG, JSON… (formats standards de représentation des ressources)
- application/xml, text/html, image/png,… (types MIME – Multipurpose Internet Mail Extensions)

HTTP est utilisé comme protocole applicatif et non protocole de transport !

Instanciation sur le Web

REST data elements	Instanciation sur le Web
Resource identifier	URL
Representation	HTML, JPG,…
Representation metadata	Media-type, last-modified,…
Control data	if-modified-since, cache-control

Instanciation sur le Web (cont.)

REST connector	Instanciation sur le Web
Client	libwww
Server	Apache API
Cache	Cache du navigateur
Resolver	Bind (DNS)
Tunnel	SSL

Instanciation sur le Web (cont.)

REST components	Instanciation sur le Web
Origin server	Apache httpd
User agent	Firefox, Chrome, Safari, IE
Gateway	Squid, CGI
Proxy	CERN proxy

Les principes généraux de REST

Adressabilité

- donner un ID à tout !

Uniformité de l'interface

- utiliser les standards

Communication stateless

- le serveur ne mémorise pas l'état d'une conversation avec un client

Transfert d'état en utilisant les possibilités hypermédia

- on relie les choses entre elles

Adressabilité

Architecture client/serveur

Les ressources sont

- identifiées par des URIs
- manipulées grâce à leurs représentations, et non directement

Les URIs permettent d'exposer

- les données
- l'état des données

Parenthèse: qu'est-ce qu'une ressource ?

Une ressource est une entité, un concept, une information qui peut être nommée

On accède à une ressource grâce à une de ses représentations

Par exemple

- une page Web est une représentation d'une ressource
- un URI permet de dire à un client qu'une ressource existe quelque part
- le client peut alors récupérer une des représentations de la ressource parmi celles disponibles au niveau du serveur

Interface uniforme

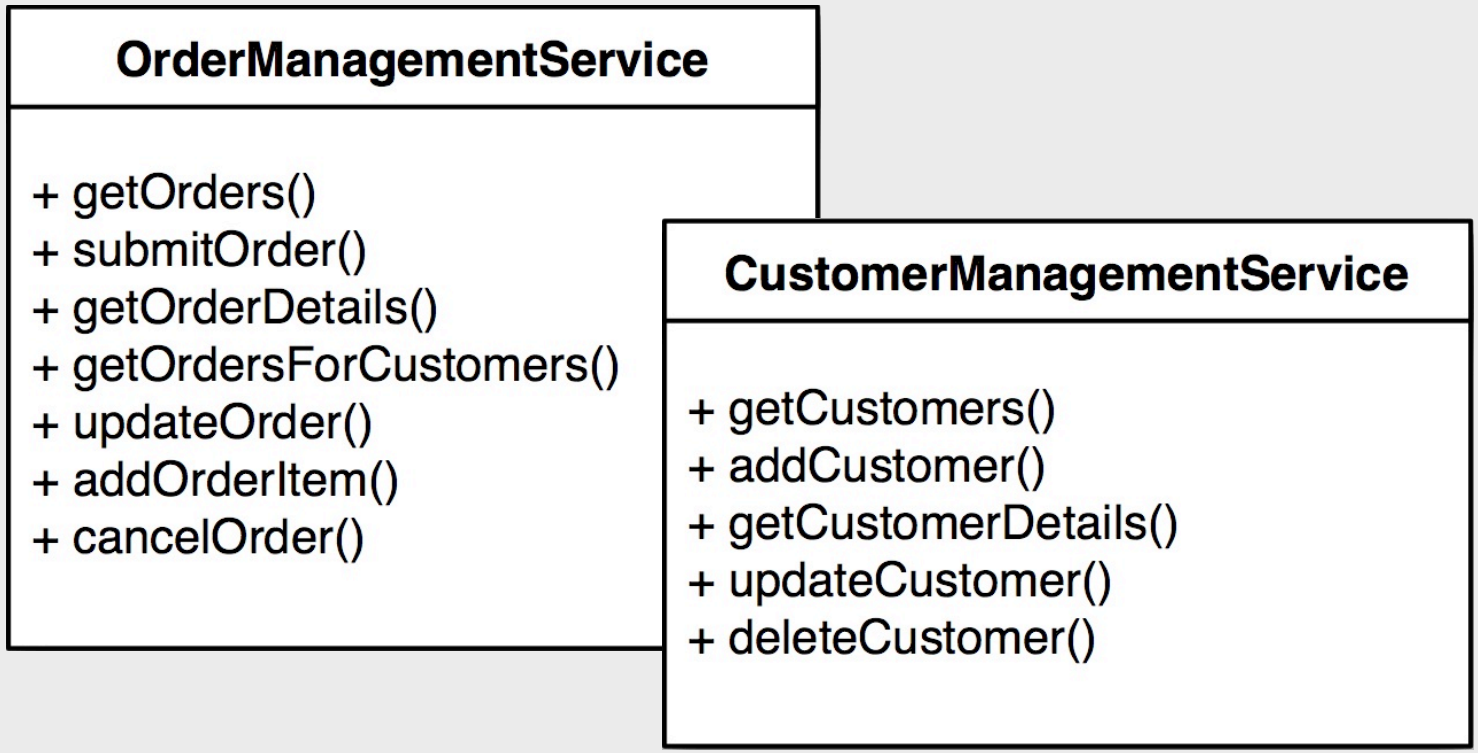
Interface uniforme pour accéder et manipuler les ressources

Sémantique uniforme et bien définie: celle de HTTP

Verbe (méthode)	Sémantique	Propriété
GET	récupère une représentation	idempotent
PUT	met à jour une représentation	idempotent
DELETE	supprime une ressource	idempotent
POST	crée une nouvelle ressource	-

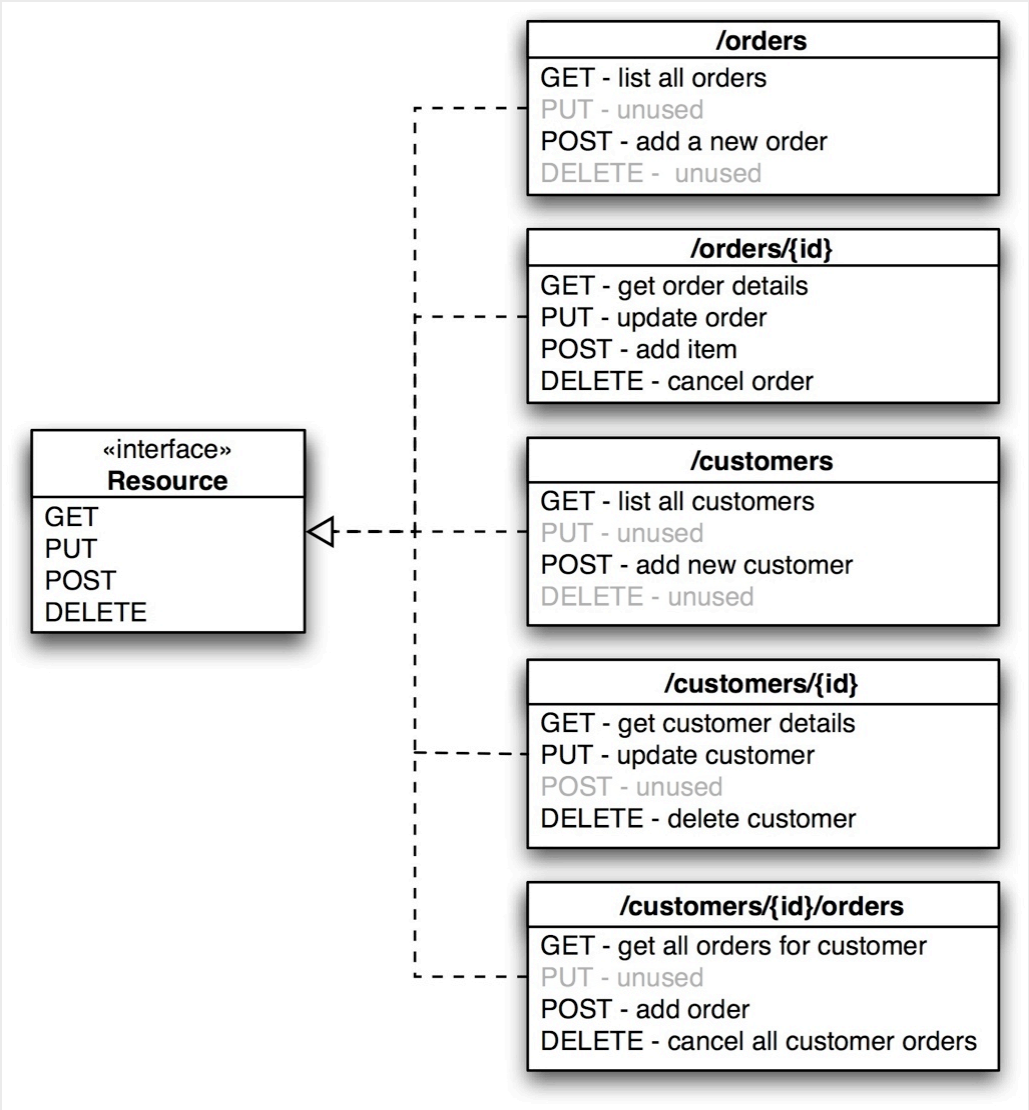
Permet l'utilisation d'un cache, possibilité d'avoir des couches supplémentaires (authentification,…)

Interface uniforme (cont.)



http://www.infoq.com/articles/rest-introduction [S. Tilkov]

Interface uniforme (cont.)



http://www.infoq.com/articles/rest-introduction [S.Tilkov]

Communication stateless

Pas de gestion des états

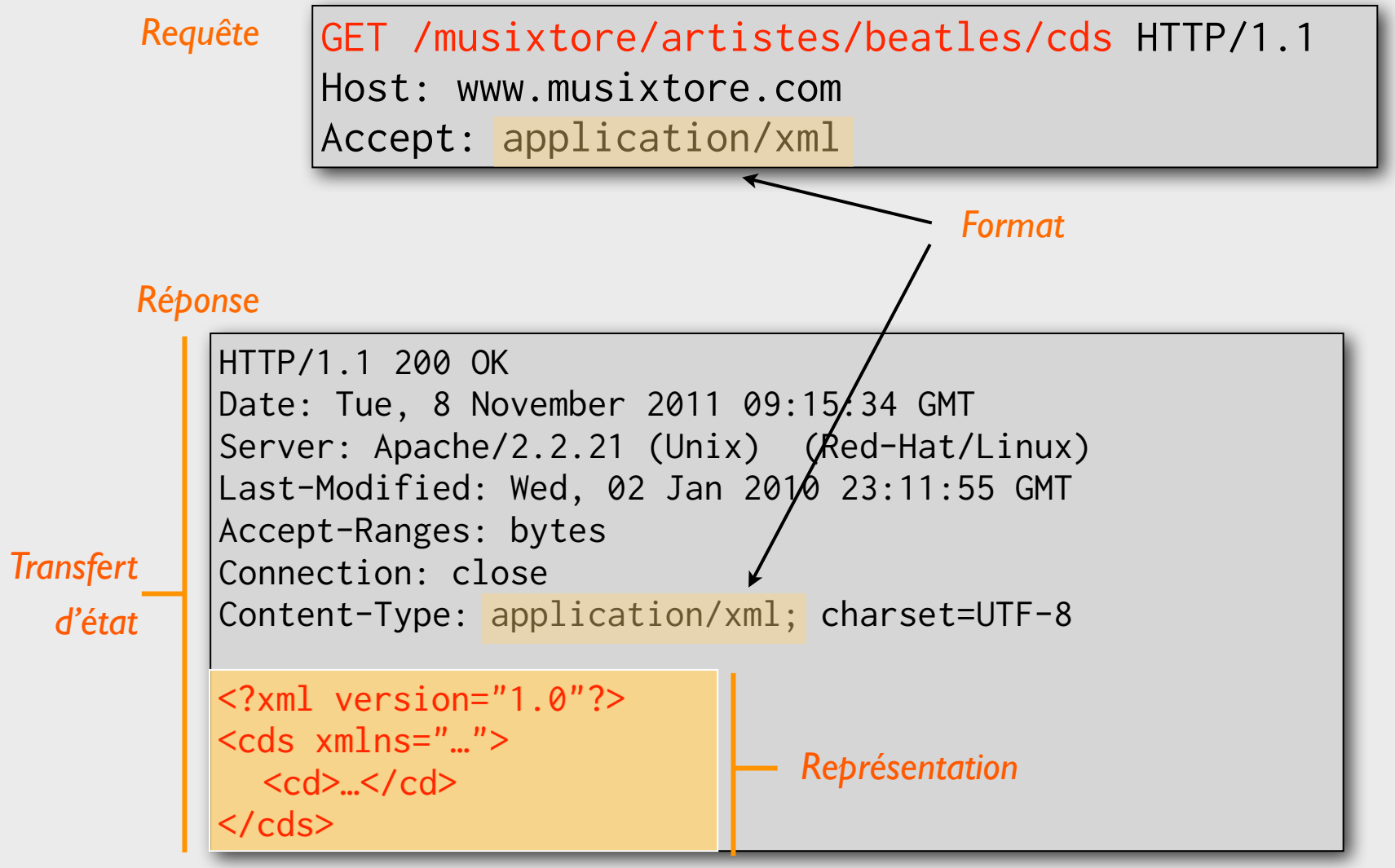
- pas de stockage du contexte au niveau du serveur
- les messages sont sans états autonomes (self-descriptive)
  - chaque requête comporte toutes les informations (l'état est conservé dans les ressources)
  - les états du serveur sont exposés comme une ressource

Utilisation des possibilités hypermédia

HATEOAS, an abbreviation for **Hypermedia as the Engine of Application State**, is a constraint of the ***REST application architecture*** that distinguishes it from most other network application architectures. The principle is that a client interacts with a network application entirely through ***hypermedia*** provided dynamically by application servers. A REST client needs no prior knowledge about how to interact with any particular application or server beyond a generic understanding of hypermedia. In a ***service-oriented architecture*** (SOA), clients and servers interact through a fixed ***interface*** shared through documentation or an ***interface description language*** (IDL).

https://en.wikipedia.org/wiki/HATEOAS

Fonctionnement



État et transfert d'état

État

- un état se rapporte à l'état de l'application ou de la session
- maintenu comme une partie du contenu transféré du client vers le serveur et vice-versa
- un serveur peut potentiellement continuer une transaction à partir de l'état dans lequel il l'avait laissé

Transfert

- les Connectors (client, serveur, cache, resolver, tunnel) ne sont pas concernés par les sessions
- l'état est maintenu en étant transféré des clients vers les serveurs et vice-versa

HTTP comme un protocole applicatif

HTTP offre

- le transport
- les méta-données
- un modèle de gestion d'erreurs
- un modèle de composant
- un environnement d'exécution

Les opérations

Méthodes HTTP comparables à CRUD pour des ressources accessibles via des URIs (analogie avec les BD)

- la ressource est créée avec POST
- elle est lue avec GET
- et mise à jour avec PUT
- pour finalement être supprimée en utilisant DELETE

Mais, certains firewalls filtrent PUT et DELETE !

- utilisation de X-HTTP-Method-Override
- entête DELETE au-dessus d'une méthode POST

Attention !

REST utilise les URI pour externaliser l'état du protocole

On passe d'un URI à un autre URI (d'un état à un autre)

REST rend les états du protocole explicites et accessibles grâce aux URI

Restreindre REST à l'insertion/mise à jour de ressources limite l'intérêt (vision uniquement données)

Par contre, utiliser REST pour passer d'un état à un autre (logique métier du service) est vraiment adapté

Principe

On crée un URL pour chaque ressource

Les ressources doivent être des noms, pas des verbes

Exemple

- http://www.airCie.com/getReservation?idClient=00345
  - getReservation est un verbe
- on peut utiliser un nom
  - http://www.airCie.com/reservations/00345

Comment ça fonctionne ?

La requête pour enregistrer un CD

```
POST /creation_cd HTTP/1.1

<?xml version="1.0" encoding="utf-8"?>
<cd xmlns="…">
 <auteur>
   <nom>John Lennon</nom>
     …
 </auteur>
…
</cd>
```

Comment ça fonctionne ? (cont.)

La réponse

```
HTTP/1.1 201 Created
Location: http://www…/cd_1

<?xml version="1.0" encoding="utf-8"?>
<CD-URI xmlns="…">http://www…/cd_1</CD-URI>
```

Comment ça fonctionne ? (cont.)

La requête pour afficher le CD

```
GET /cds/cd_1 HTTP/1.1

```

Comment ça fonctionne ? (cont.)

La réponse

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8

<?xml version="1.0" encoding="utf-8"?>
<cd>
    <auteur>
      <nom>John Lennon</nom> …
    </auteur>
…
</cd>
```

Comment ça fonctionne ? (cont.)

La requête pour afficher la liste des CDs

```
GET /cds HTTP/1.1

```

Comment ça fonctionne ? (cont.)

La réponse

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8

<?xml version="1.0" encoding="utf-8"?>
<cds xmlns="…">
  <cd>http://www…/cd_1</cd>
  <cd>http://www…/cd_2</cd>
</cds>
```

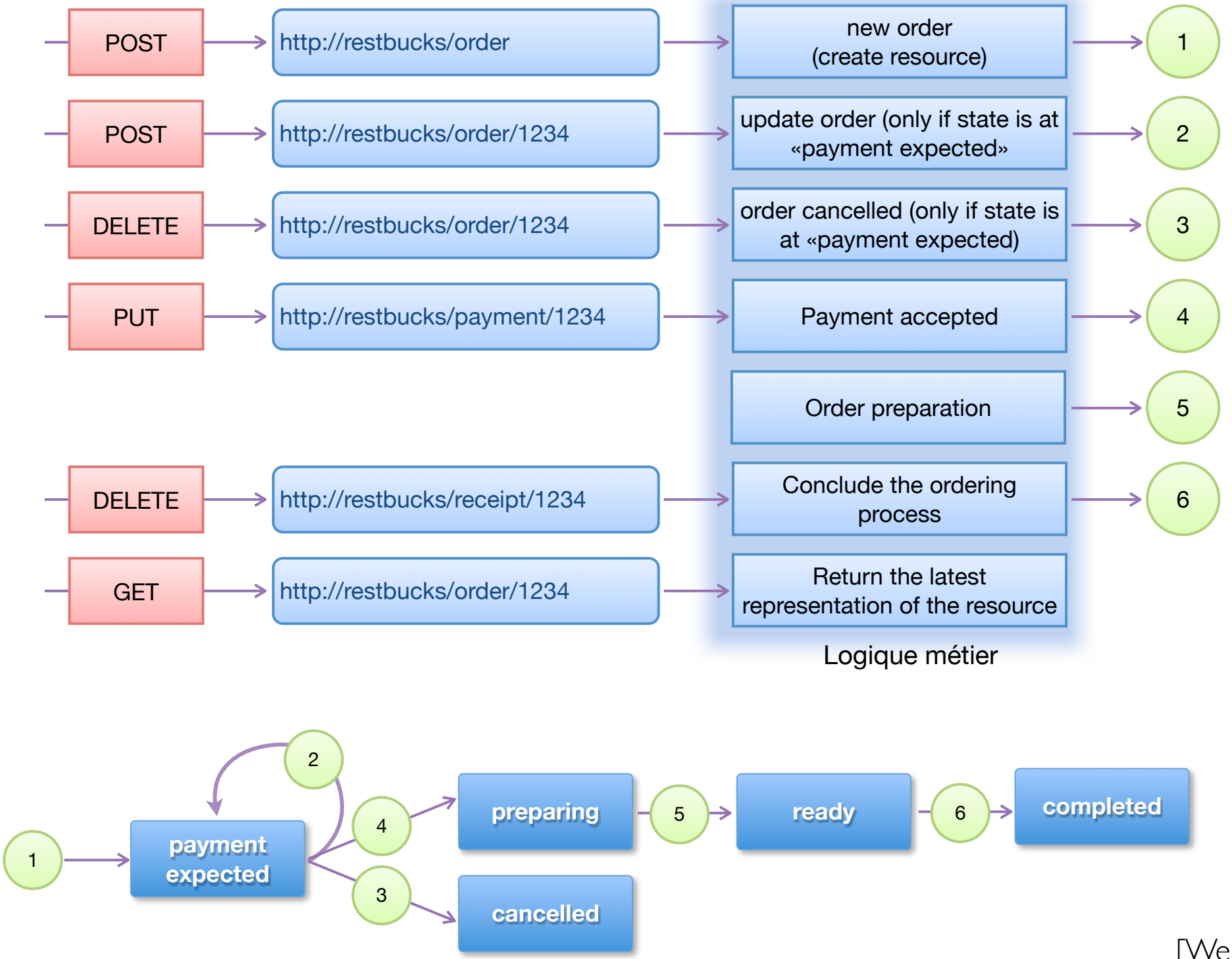
Les liens

La représentation d'une ressource peut contenir d'autres URIs

Les liens agissent comme des transitions entre états

L'état d'une conversation entre client et serveur est mémorisé en termes d'états

Exemple


POST
http://restbucks/order
new order
(create resource)
1
POST
http://restbucks/order/1234
update order (only if state is at «payment expected»
2
DELETE
http://restbucks/order/1234
order cancelled (only if state is at «payment expected)
3
PUT
http://restbucks/payment/1234
Payment accepted
4
Order preparation
5
DELETE
http://restbucks/receipt/1234
Conclude the ordering process
6
GET
http://restbucks/order/1234
Return the latest representation of the resource
Logique métier
1
payment expected
2
3
4
preparing
cancelled
5
ready
6
completed

[Webber]

JSON

JavaScript Object Notation

- utilise la syntaxe JavaScript

Paires clé-valeur, plus tableaux

Plus compact que XML

Peut être chargé en utilisant la fonction Javascript eval()

Bibliothèques disponibles pour de nombreux langages

- http://www.json.org/

REST/JSON est de plus en plus populaire

Mais REST, ce n'est pas uniquement…

Des URIs bien formés

Du XML ou du JSON

Des applications AJAX

Intérêt de REST

Passage à l'échelle

Tolérance aux fautes

Recouvrable

Sécure

Couplage faible

Exactement ce que l'on attend des systèmes distribués métiers !

Passage à l'échelle

Le Web est à l'échelle d'Internet

- couplage faible
  - les changements ici n'ont pas de répercussion là-bas
- interface uniforme
  - HTTP définit une interface standard pour tout le monde
  - la réplication et la gestion du cache sont d'origine dans le modèle
- modèle stateless
  - supporte le passage à l'échelle horizontal
- cache
  - pour le passage à l'échelle vertical

Tolérance aux fautes

Le Web est stateless

- toute l'information nécessaire pour le traitement d'une requête est dans la requête

Réplication plus facile

- un serveur Web est remplaçable par un autre
- passage à l'échelle horizontal, basculement (fail-over) facilité

Recouvrable

Le Web prend en compte la restauration de l'information

- GET, PUT et DELETE sont idempotents
- dans le cas d'un échec, on peut refaire un GET/PUT/DELETE sur la ressource

Les verbes HTTP sont sûrs

- les codes erreurs de HTTP sont bien définis
- certains verbes (PUT, DELETE) sont sûrs vis-à-vis de la duplication

Sécure

HTTPs est relativement mature

- HTTP Basic et Digest Authentication
- SSL/TLS pour les échanges point-à-point (évite les attaques man-in-the-middle)

On peut également crypter certaines parties de la ressource

- du coup, on peut utiliser le cache !

Plus de détail dans le cours Sécurité: clé publique, CA, OAuth, OpenID

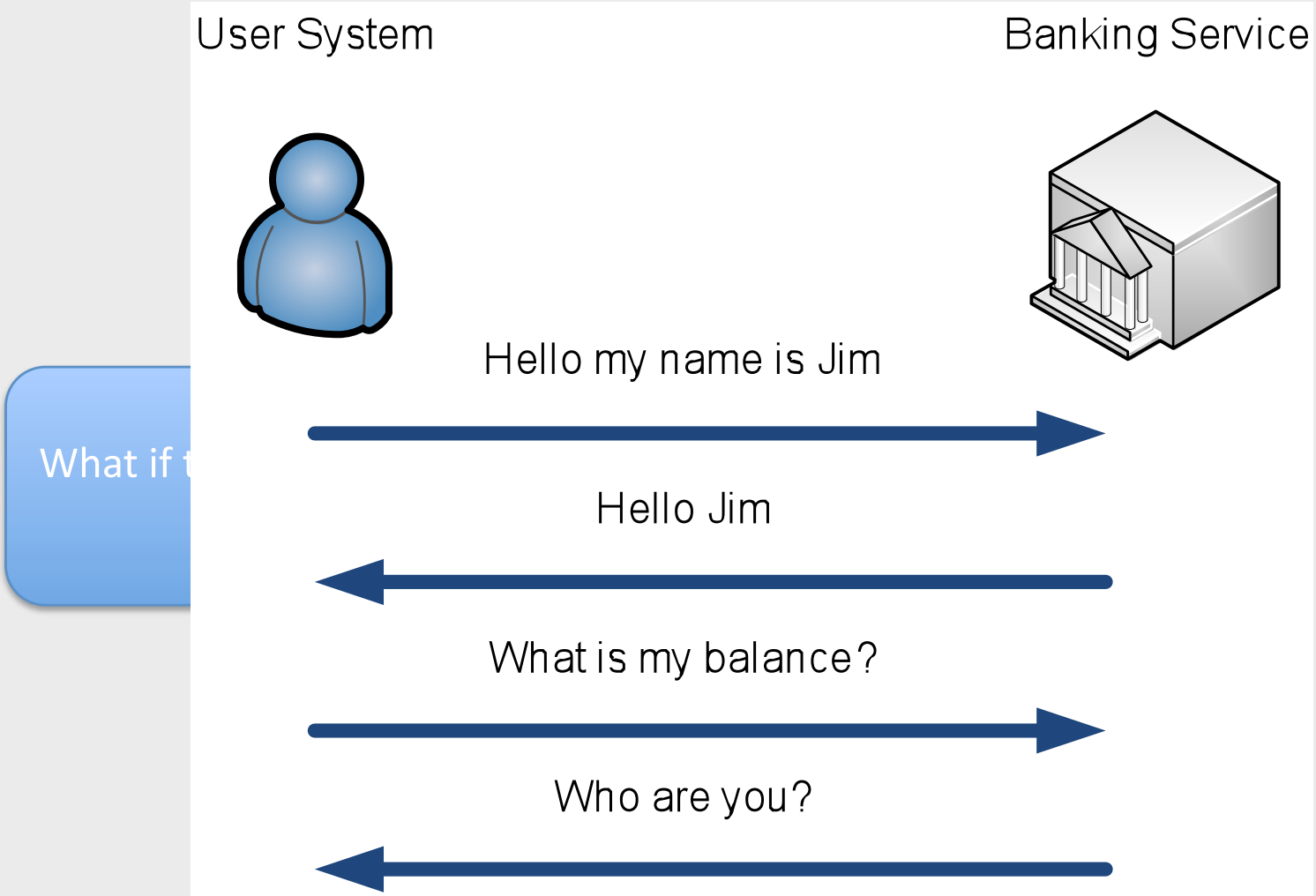
Couplage faible

Ajouter une opération métier ne modifie pas les autres opérations métier

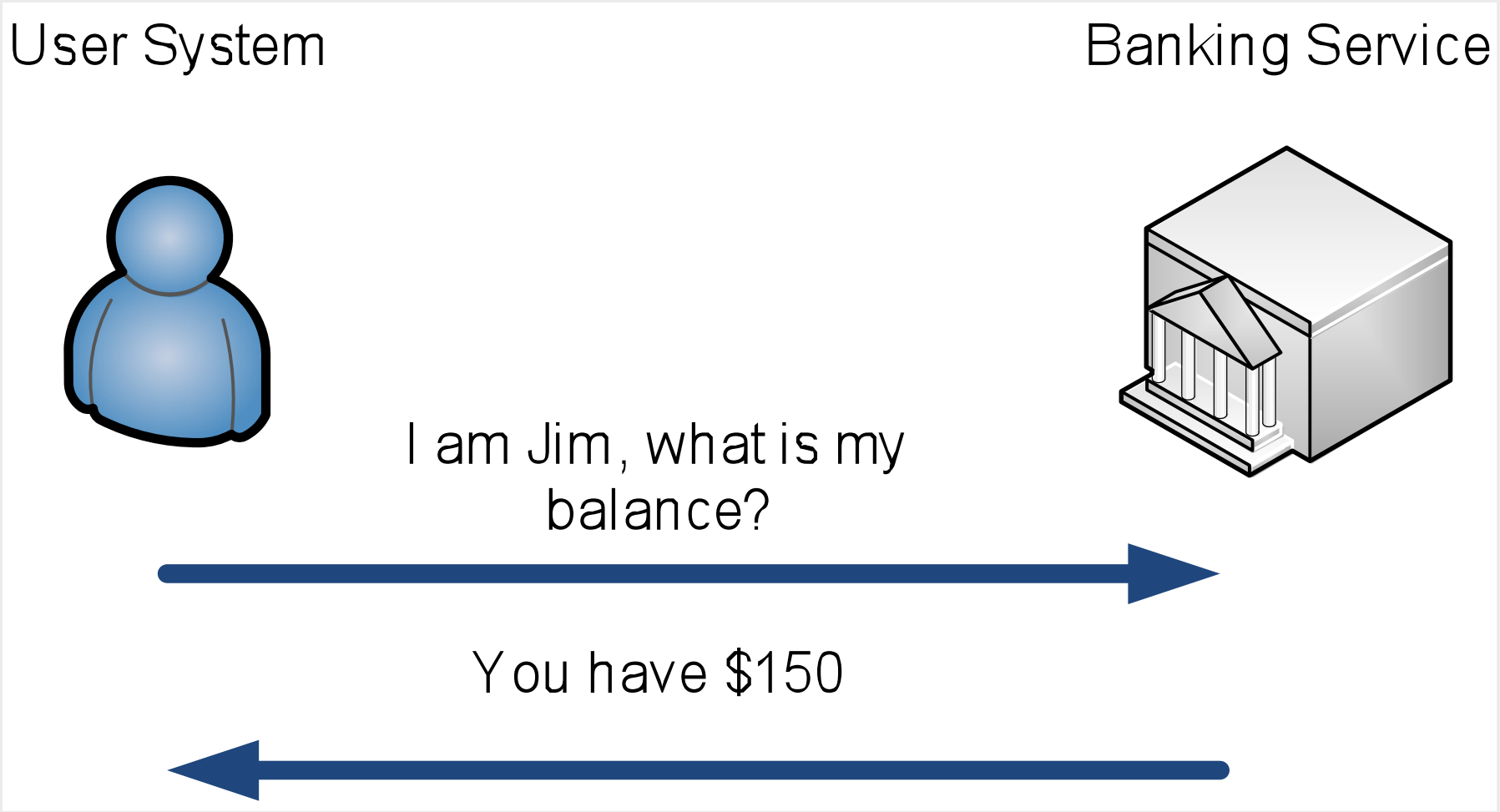
Toutes les opérations utilisent la même interface

Stateless vs. stateful

[Webber]



Stateless vs. stateful (cont.)



SOAP ou REST ?

Emacs ou vi ?

Critiques faites par les partisans de chaque communauté

- contre les WS
  - « on réinvente le Web avec HTTP POST »
  - « c'est trop complexe »
  - « on perd les avantages du Web »
- contre REST
  - « impossible de tout modéliser avec GET, POST, PUT et DELETE »
  - « comment fait-on quand on veut des transactions, de la sécurité,... ? »

Sous un angle différent

Les Web services sont très ambitieux

- SOAP: extensibilité
- extensions: non synchrone, fiabilité, intégrité, sécurité, confidentialité…
- indépendant du protocole
- description des services analysable par une machine
- description des compositions de services analysables par une machine (par ex. WSBPEL)

REST est simple, et essentiellement au niveau technologique

- rien sur fiabilité, intégrité, sécurité,… car on utilise HTTP
- beaucoup de choses à faire par le développeur
- modélisation du système comme une ressource

Quand utiliser REST ?

Si vos services sont entièrement stateless

Si vos services renvoient des données qui peuvent être mises dans un cache

Si le fournisseur et le client du service ont une compréhension commune du contexte

Si la bande passante est importante pour vous

- REST est performant même avec des configurations limitées
- REST ne nécessite pas d'infrastructure particulière

Si vous voulez composer des services existants (on peut construire des applications en utilisant AJAX)

Quand utiliser SOAP ?

Si vous estimez nécessaire d'avoir un contrat entre le service et le client à travers une interface (WSDL)

Si vous avez des besoins non fonctionnels (adressage, fiabilité, sécurité, coordination,…) et que vous ne voulez pas les coder vous-même

Si vous avez besoin d'autres protocoles que HTTP, en particulier sur des protocoles permettant des communications asynchrones (SMTP, JMS,…)

Conclusion sur Java Avancé

Qu'est-ce qu'on a vu ?

JavaEE

- servlets, JSP, JPA, patrons de conception

Java RMI

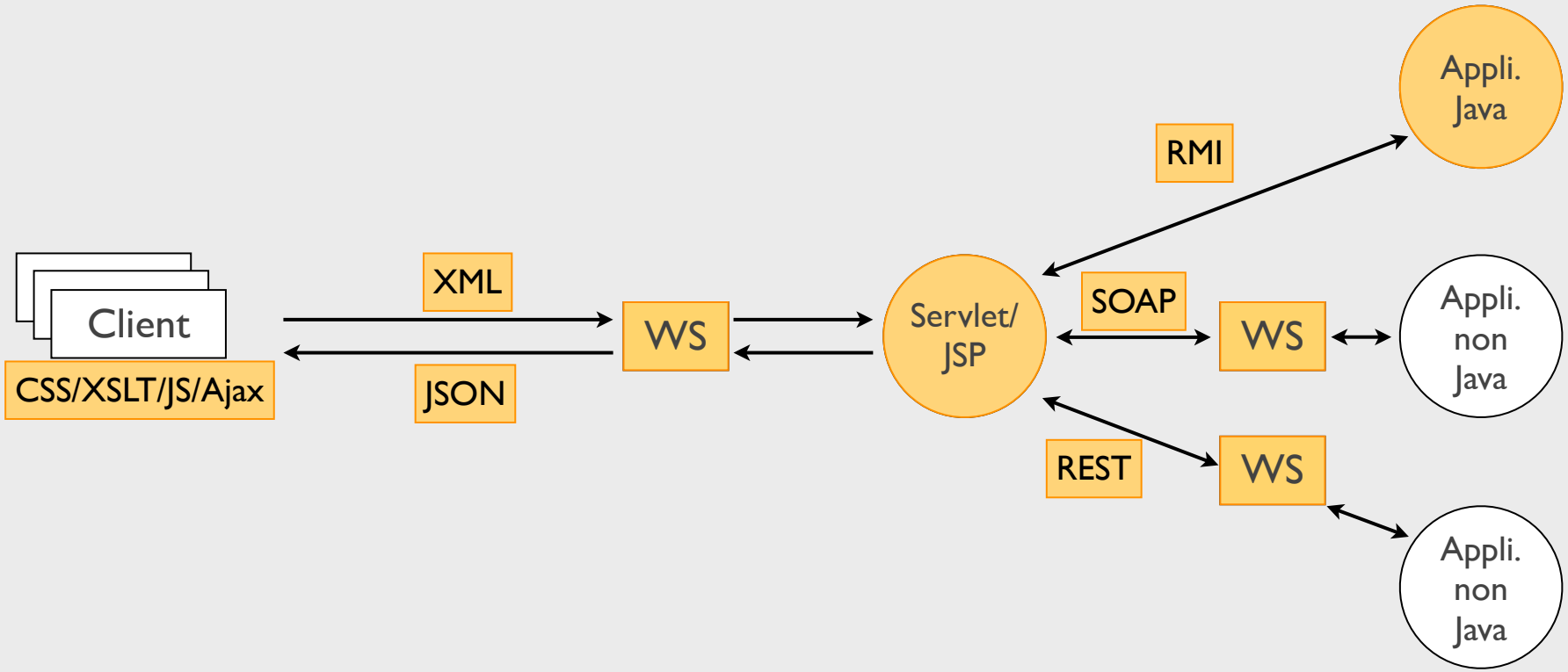
- comment se connecter à des applications Java distantes

Services Web

- comment se connecter à des applications non Java distantes
- SOAP
- REST

Plus les autres cours: Conception, CSS, DOM/JS, XML, XSLT, IU,…

Comment assembler les morceaux ?



Interview de Werner Vogels (CTO Amazon)

«If you hit the Amazon.com gateway page, the application calls more than ***100 services*** to collect data and construct the page for you.
The big architectural change that Amazon went through in the past five years was to move from a ***two-tier monolith to a fully-distributed, decentralized, services platform*** serving many different applications.»

JG How many of the current buzzwords, such as SOA, WSDL, SOAP, WS-security, are relevant to you?
WV … The second category is the interface with our retail partners, which has strict descriptions for XML feed processing, service interfaces, etc., and where we leverage as many standard technologies as possible.
The third category is our public Amazon Web Services, which builds on the platform services and provides REST-like as well as SOAP interfaces. If we look at how developers use these interfaces, in general the REST version is used by small libraries in Perl or PHP as part of a LAMP stack, and the SOAP calls are mainly done by applications that have been built on Java or .NET platforms by consuming our WSDL files and generating proxy objects.

WV The mobile space is an example. This world is so much in flux, with respect to form factors, network speeds, and input methods, that it is difficult to build applications that give a good user experience across the board. We have made sure in our Web-services interfaces that developers can specify external XSLT (extensible stylesheet language transformation) stylesheets so that data can get post-processed at our servers and that what is returned to the consumer is in a format that is optimal for the device and application that requested it.