L'architecture REST

Benjamin Canou - Christian Queinnec

Cours 3 du 3/12/2012

Architecture des Applications Réticulaires

Rappels sur HTTP

Caractéristiques
Formats des requêtes et réponses
Les méthodes et en-têtes

Architecture des Applications Réticulaires

Rappels de base sur HTTP

Caractéristiques du protocole :

- Textuel
- Non connecté (à ne pas confondre avec asynchrone)
- Asymétrique (client $\neq$ serveur)

Historique :

- Inventé par Tim Berners-Lee et son équipe (au CERN)
- Né en même temps qu'HTML
- Projet World Wide Web : rencontre d'Internet et de l'hypertexte
- Première documentation : 1991

Requêtes

Structure :

- Commande : <méthode>_<URI>_HTTP/<version>
- En-tetes : <en-tête>:_<valeur> (pour chaque ligne)
- Ligne vide
- Corps (si approprié)

Exemple :

```
1 : GET /getcookie?size=big&flavor=chocolate HTTP/1.1
2 : User-Agent: Mozilla/5.0 \hilight{...}
3 : Accept: text/html,application/xhtml+xml, \hilight{...}
4 : Accept-Encoding: gzip, deflate
5 : Accept-Language: fr,fr-fr;q=0.8,en-us;q=0.5,en;q=0.3
6 : Connection: keep-alive
7 : Cookie: client-number: 1234: client-name: bob
8 : If-Modified-Since: Sun, 25 Nov 2012 21:49:55 GMT
9 : Cache-Control: max-age=0
10 :
11 : EOF
```

Réponses

Structure :

- Commande : HTTP/<version>_<code>_<texte>
- En-tetes : <En-tête>:_<valeur> (pour chaque ligne)
- Ligne vide
- Corps (si approprié)

Exemple :

```
1 : HTTP/1.1 200 OK
2 : Date: Sat, 01 Dec 2012 21:39:54 GMT
3 : Content-Type: text/html; charset=UTF-8
4 : Set-Cookie: flavor=chocolate; size=small
5 : Cache-Control: no-cache
6 : Expires: -1
7 :
8 : <html>
9 :   <h1>Sorry bob, no more big cookies :(</h1>
10 :   Here, have a small one for free !
11 : </html>
12 : EOF
```

Méthodes

Les bien connues :

- GET : accès à une URI avec paramètres
- POST : accès à une URI avec paramètres et corps
- HEAD : comme GET sans le corps de la réponse
- OPTIONS : donne les méthodes disponibles pour l'URI

Les moins connues mises à profit par REST :

- PUT : affecte le document à l'URI
- DELETE : supprime le document à l'URI
- PATCH : modifie le document à l'URI

D'autres :

- CONNECT : proxy HTTP
- TRACE : écho
- Extensions spécifiques (ex. WebDAV)

En fait, souvent seulement GET, HEAD et POST.

En-têtes

De requête :

- Format attendu : Accept, Accept-Charset, Accept-Encoding, Accept-Language
- Cache : Accept-Datetime, Cache-Control, If-Modified-Since, If-Unmodified-Since
- Proxy : Host, Max-Forwards, Proxy-Authorization, Via
- Meta : Content-Type, Content-MD5, Date, User-Agent, Referer, Warning
- Session : Cookie, Authorization, DNT, Connection

De réponse :

- Format : Content-Encoding, Content-Language, Content-Length, Content-Location, Content-Type
- Cache : Cache-Control, ETag, Expires, Last-Modified,
- Proxy : Age, Proxy-Authenticate, Via
- Meta : Allow, Date, Server, Warning
- Session : Set-Cookie, Connection, Location, Refresh, Retry-After

Codes de retour

- 200 OK
- 201 Created (+ Location pour URI canonique)
- 202 Accepted (+ Location pour URI de contrôle)
- 204 No Content (la représentation est vide)
- 301 Moved Permanently (+ Location)
- 303 See Other (+ Location pour le contenu de la réponse)
- 304 Not Modified
- 307 Temporary Redirect (+ Location pour re-soumettre la requête)
- 400 Bad Request
- 404 Not Found
- 405 Method Not Allowed
- 406 Not Acceptable (pas de représentation acceptable)
- 409 Conflict
- 410 Gone
- 415 Unsupported Media Type
- 500 Internal Server Error
- 503 Service Unavailable

Le modèle REST

Historique & exemple
Principes fondamentaux
Comparaison avec l'approche services

Architecture des Applications Réticulaires

Généralités

Point de départ :

- Thèse de Roy Fielding (2000) : pourquoi ne pas s'appuyer plus sur HTTP ?
  Architectural Styles and the Desing of Network-Based Software Architectures
- Plus de sémantique dans l'URL: une URI désigne une ressource
- Pas d'enveloppes (aller ou retour): l'URL est l'enveloppe d'émission
- Outre GET, usage des commandes HEAD, POST, PUT, DELETE
- Adoption du modèle déconnecté d'HTTP

Les grands exemples d'architecture REST:

- Produits : Amazon, Google, Atom
- Technologies : Ruby on Rails,django

Exemple: Amazon S3

Serveur programmatique de stockage

- GET / liste les aires
- GET /bucket liste les objets contenus dans une aire
- GET /bucket/object renvoie l'objet
- HEAD /bucket/object ne renvoie que les méta-données concernant l'objet
- PUT /bucket/object + body crée/modifie l'objet
- POST /bucket + body crée un objet anonyme et renvoie son URI (via Location)
- DELETE /bucket/object supprime l'objet
- DELETE /bucket supprime l'aire
- …

Principes REST

- Découplage client /serveur
  - Client = interface
  - Serveur = données (ressources)
  - Le serveur ne stocke rien sur le client (stateless)
- Clarté du Nommage :
  - Toute URI désigne une unique ressource
  - Une même ressource peut avoir plusieurs noms /software/1.0.3.tgz & /software/latest.tgz
  - Valable à un instant donné
- Découplage entre ressource et représentations :
  - Dans les entêtes HTTP: Accept, Accept-Encoding etc.
  - Dans l'URI (`GET/id/xml/`)
- Compatible avec le mécanisme de cache d'HTTP

Modèle CRUD

Quatre opérations, implantées par quatre méthodes HTTP

- Create (POST) : alloue une nouvelle ressource
- Read (GET) : accède à une ressource existante
- Update (PUT) : affecte une ressource (existante ou non)
- Delete (DELETE) : supprime une ressource

Échec / réussite : code de retour HTTP

Contraintes fortes sur le serveur (déjà présentes dans HTTP) :

- GET et HEAD sont sans effet
- PUT et DELETE sont idempotentes
- POST est sans contrainte
- Gestion parfaite du cache (en-têtes If-Modified-Since, Expires, ETags, etc.)

Représentations

Désignation de la représentation :

- Dans l'en-tete avec le type MIME : text/xml, text/plain, text/json, etc.
- Dans l'URI : /xml/, /json/, etc.

Formats classiques :

- Représentation brute pour les machines : XML, JSON, YAML
- Représentation enrobée et hypertexte pour l'humain : HTML
- Représentations natives pour les images, sons, textes

Comment répondre à une requête ?

- L'URI est analysée pour déterminer la ressource concernée.
  pour S3: `/{bucket}/{object}`
- la commande est déterminée (GET, PUT, POST, DELETE)
- Si retour il y a, la représentation de la ressource ou de l'anomalie est déterminée
- les méta-données décrivant la représentation accompagnent celle-ci.

La forme usuelle est d'avoir une arborescence de contrôleurs/actions:

$$(\text{patronURL} \times \text{variables}) \rightarrow \text{ressource}$$
$$\text{ressource} \rightarrow (\text{méthode} \rightarrow \text{représentation})$$

Par exemple :

```
1 : GET /person/{nom}/{prenom} -> fun(nom prenom) ressource
2 : PUT /person/{id}/age/{age} -> fun(id age) ressource
```

Multiples variantes différemment Curryfiées ou empaquetées:

```
1 : /person/{nom}/age/{age} ->
2 :     function(%parametres) ->
3 :         GET -> ressource
```

Difficultés et limites

- Actions binaires:
  - Demander si une personne fait partie d'un groupe ?
  - Demander si un groupe contient une personne ?
  - Les deux ? Faire apparaitre le format interne ?
  - Plus dur : pour la création ?
- Transactionnel
  - Décrire un virement bancaire ?
  - Créer une ressource incrémentalement ?
- Les URI doivent-elles être évidentes ou opaques ?
  - Donnée : /PDP11/home/~maurice/gopher/ (joli, mnémotechnique)
  - Identifiant : /homedir/23eab89c/ (résistant au changement)
- Sécurité
  - Authentification HTTP simple
  - Utilisation d'HTTPS, clefs de cryptage échangées par un autre moyen, etc.

Modèle d'interaction :

- SOAP : Échanges
  - Le serveur conserve des données sur la session
  - Les messages ne contiennent que ce qu'ils expriment
- REST : Opérations indépendantes
  - Serveur sans état
  - Les messages doivent embarquer le contexte

Cible :

- SOAP : plutot des services transactionnels
- REST : plutôt des échanges de données/documents

Protocole :

- SOAP : subit HTTP
  - Indépendant du transport donc d'HTTP
    mais en réalité, la large majorité des échanges passe sur HTTP
  - Propre modèle de sécurité
  - Propre retour des erreurs
  - Propre stratégie de cache ou d'idempotence
- REST : épouse HTTP

Formats :

- REST : s'adapte aux capacités du client
- WSDL : définit finement les formats des données échangées

Documentation :

- REST : pas de norme, mais facile à décrire (patrons)
- WSDL : définit finement (mais verbeusement) les échanges

Bibliothèques et outils

Architecture des Applications Réticulaires

Implantations

Bibliothèques :

- Perl bas niveau : REST::Resource, REST::Application
- Perl MVC : Catalyst :: Action :: REST
- Java : Restlet, Spring 3.0 et RestTemplate

Exécution :

- Indépendante, derrière nginx, intégréé dans Apache ou Tomcat.
- Avec Apache, directive AcceptPathInfo utile

Perl :: REST :: Resource

1. sous-classer REST::Resource
2. indiquer les fonctions à invoquer pour chaque méthode HTTP. Ces fonctions ont la signature (webapp requête) $\rightarrow$ ...
3. écrire ces fonctions qui construisent la réponse HTTP

```
1 : package My::REST::Resource;
2 : use base "REST::Resource";
3 : sub new {
4 :   my ($class) = @_;
5 :   my $webapp = $class->SUPER::new(@_);
6 :   $webapp->method('GET', \&getFunction, $comment);
7 : }
8 : sub getFunction {
9 :   my ($webapp, $request) = @_;
10 :   ...
11 : }
```

Perl :: REST :: Application

1. sous-classer REST::Application
2. spécialiser setup() pour ajouter des couples *URI-regexp* $\rightarrow$ fonction(webapp regexp-parts) ...
3. écrire les fonctions prenant des morceaux de l'URI (les informations supplémentaires sont dans la webapp et construisant la réponse HTTP

```
 1 : my My::REST::Application;
 2 : use base 'REST::Application';
 3 : sub setup {
 4 :   my ($webapp) = @_;
 5 :   $webapp->resourceHooks(
 6 :       m|^/city/(\w+)/station/(\d+)$| => \&stationFonc);
 7 : }
 8 : sub stationFonc {
 9 :   my ($webapp, $city, $stationId) = @_;
10 :   ...
11 : }
```

Bibliothèque MVC :

```
1 : % catalyst.pl MyApp
2 : % ### add models, views, controllers
3 : % script/myapp_create.pl model MyDatabase DBIC::Schema \
4 : >   create=dynamic dbi:SQLite:/path/to/db
5 : % script/myapp_create.pl view MyTemplate TT
6 : % script/myapp_create.pl controller Search
7 : % ### built in testserver
8 : % script/myapp_server.pl
```

Comme en RoR, django, le modèle est dérivé automatiquement de la BD.

```
1 : ### in lib/MyApp.pm
2 : use Catalyst qw/-Debug/; # include plugins here as well
3 :
4 : ### In lib/MyApp/Controller/Root.pm (autocreated)
5 : sub foo : Global { # called for /foo, /foo/1, /foo/1/2, etc.
6 :     # args are qw/1 2/ for /foo/1/2
7 :     my ( $self, $c, @args ) = @_;
8 :     $c->stash->{template} = 'foo.tt'; # set the template
9 :     # lookup something from db
10 :    # stash vars are passed to TT
11 :    $c->stash->{data} =
12 :      $c->model('Database::Foo')->search({
13 :            country => $args[0] });
14 :    if ( $c->req->params->{bar} ) {
15 :      # access GET or POST parameters
16 :      $c->forward( 'bar' ); # process another action
17 :      # do something else after forward returns
18 :    }
19 : }
```

```
1 : # The foo.tt TT template can use the stash data from
2 : # the database
3 : [% WHILE (item = data.next) %]
4 :   [% item.foo %]
5 : [% END %]
```

Lire le tutoriel sur le site http://www.restlet.org/

```
1 : // Create a component
2 : Component component = new Component();
3 : component.getServers().add(Protocol.HTTP, 8182);
4 : component.getClients().add(Protocol.FILE);
5 :
6 : // Create an application
7 : Application application =
8 :  new Application(component.getContext()) {
9 :   @Override
10 :   public Restlet createRoot() {
11 :     // Create a root router
12 :     Router router = new Router(getContext());
13 :     // Attach a guard to secure access to the directory
14 :     Guard guard = new Guard(getContext(),
15 :       ChallengeScheme.HTTP_BASIC, "Restlet tutorial");
16 :     guard.getSecrets().put("scott", "tiger".toCharArray());
17 :     router.attach("/docs/", guard);
```

```
1 : // Create a directory able to expose a hierarchy of files
2 : Directory directory = new Directory(getContext(),
3 :                                    ROOT_URI);
4 : guard.setNext(directory);
5 : // Create the account handler
6 : Restlet account = new Restlet() {
7 :   @Override
8 :   public void handle(Request request, Response response) {
9 :     // Print the requested URI path
10 :    String message = "Account␣of␣user␣\""
11 :      + request.getAttributes().get("user") + "\"";
12 :    response.setEntity(message, MediaType.TEXT_PLAIN);
13 :  }
14 : };
15 : // Attach the handlers to the root router
16 : router.attach("/users/{user}", account);
17 : // Return the root router
18 : return router;
```

```
1 : host = new VirtualHost(getContext());
2 : host.setHostDomain("restlet.org|restlet.net|restlet.com|"
3 :         + "www.restlet.net|www.restlet.com");
4 : host.setHostPort("80|" + Integer.toString(port));
5 : host.attach(new RedirectApplication(getContext(),
6 :         "http://www.restlet.org{rr}", true));
7 : getHosts().add(host);
8 :
9 : // Attach the application to the component and start it
10 : component.getDefaultHost().attach(application);
11 : component.start();
```

Utilisation avancée :

- Liens entre méthodes Java et URIs par annotations
- Envoi d'objets Java sérialisés (pourquoi pas ?)
- Couche client : une sorte de RMI via REST
    - Description des ressources via des interfaces
    - Réception d'objets Java
    - Implantations Java, GWT, Android

Autres implantations :

- Jersey,implantation de la JSR 311
- Spring / RestTemplate

From Scratch

Moteur basique en quelques centaines de lignes
Comment typer le modèle REST

Architecture des Applications Réticulaires

Revue de code

Revue d'un micro serveur HTTP :

- Quelques dizaines de lignes pour un serveur basique
- Quelques centaines pour un serveur complet
- Un peu plus pour un serveur robuste et léger
- À faire une fois dans sa vie !

En définissant proprement les patrons, on gagne :

- La vérification automatique des types des paramètres
- La vérification statique de l'adéquation des actions aux patrons
- La compatibilité entre tous les patrons
- La vérification du traitement exhaustif des possibilités d'appels

L'architecture REST se prête très bien au typage et à la vérification statique

Conclusion

REST est un style architectural :

- Structurant: à cause du nommage des URI
- Efficace: utilise bien la machinerie HTTP
- Évolutif: en termes de nouvelles URIs ou représentations
- Indépendant des mises en œuvre (jsp, restlet, etc.)