The REBOL Documentation Project

-- FR - Documentation REBOL - Tutoriels --

Tutoriels

REBOL/Services pour les nuls (4) : Analyse de l'évaluation d'une requête

REBOLtof

Première publication : 26 novembre 2005, et mis en ligne le vendredi 25 novembre 2005

Résumé :

Souhaitant, pour des raisons professionnelles, comprendre les mécanismes internes à REBOL/services, je n'ai pas voulu garder mes découvertes pour moi, et je les propose sous la forme de cet article.

Souhaitant, pour des raisons professionnelles, comprendre les mécanismes internes à REBOL/services, je n'ai pas voulu garder mes découvertes pour moi, et je les propose sous la forme de cet article. N'hésitez pas à me faire part de vos remarques ou corrections !

Version : 1.0
Date : 18-nov-2005

Historique

Date	Version	Commentaires	Auteur	Email
18-nov-2005	1.0	initial	REBOLtof	reboltof
26-nov-2005	1.1	correction du code du service : j'avais copy/passed une ancienne version :-(	REBOLtof	reboltof

1 Préliminaires

Soit le service :

```
rebol []
name: 'shell
title: "Shell Server"
commands: [
    'do-it
    "execute passed command"
    arg: block!
    (result: do arg/1)
]
```

le serveur qui l'implémente :

```
rebol []
do %server.r
service: start-service/options tcp://:8000 [
    services: [%shell-sv.r]
]
wait []
```

et le client qui lance la requête :

```
rebol []
do %client.r
port: open-service tcp://localhost:8000
result: do-service port [shell/do-it [1 + 1] ]
probe result
```

2 Réception de la requête par le serveur

Le serveur implémente des handlers pour tous les protocoles supportés. Nous allons nous intéresser plus particulièrement au protocole TCP, afin de réaliser une architecture Client/Serveur.

Au lancement du serveur, la fonction **'opener** ouvre un port en mode asynchrone, sans buffer. Un afflux de données sur ce port invoque la fonction **'on-connect**.

Cette fonction **'on-connect** reste à l'écoute du port ouvert. Une activité sur le port déclenche alors la fonction **'on-data**.

'on-data va examiner le contenu du port et éventuellement déclencher une erreur en cas de problème. Si tous se passe bien, les données trouvées sur le port (donc la requête) sont transmises à la fonction en charge du traitement des requêtes : **'do-request** :

```
data: do-request port/user-data data
```

3 Traitement de la requête

La fonction **'do-request** attend deux arguments : **'session** et **'request** :

Le premier, **'session** est un objet contenant tous les paramètres de la session en cours, y compris, dans **'session/config**, les services chargé lors du lancement du serveur. Parmi ces services, nous y retrouverons le service par défaut (home) mais également le(s) service(s) que nous avons créé(s) (shell dans notre cas) :

```
id              integer!  1
seq             integer!  1
config          object!   [flags service-info services  service-l...
flags           block!    length: 0
changed         block!    length: 0
remote-ip       tuple!    127.0.0.1
port            port!     make object! [ scheme: 'tcp host: 127....
recv-buf        binary!   #{}
port-timeout    integer!  20
send-queue      block!    length: 0
sent-queue      block!    length: 0
```

```
callback        none!     none
header          none!     none
request         block!    length: 2
results         block!    length: 0
this-service    none!     none
max-header      integer!  2000
max-request     integer!  100000
max-payload     integer!  8000000
user-id         integer!  0
encode          integer!  1
challenge       none!     none
crypt-key       binary!   #{08395286B702BCDD3AD79FF7FA4DDADE382AB189}
crypt-len       integer!  0
rsa-key         none!     none
encrypt-port    none!     none
decrypt-port    none!     none
language        none!     none
start-date      date!     18-Nov-2005/14:02:56+1:00
access-date     date!     18-Nov-2005/14:02:56+1:00
login-date      none!     none
```

Le second, **'request**, est un block contenant un header (exigé par le protocole de communication utilisé par /Service) et notre requête :

```
[[request seq 1] [shell/do-it [1 + 1]]
```

'do-request va parcourir ce block afin d'en extraire les commandes transmises, c'est-à-dire chacun des blocs contenus dans la requête, sauf le premier.

Chaque commande trouvée est ensuite transmise à la fonction **'do-command**, pour exécution :

```
catch [result: do-command session command none]
```

4 Evaluation de la commande

'do-command est chargé d'évaluer la commande et de retourner le résultat ou de déclencher, le cas échéant, une erreur. Cette fonction est importante pour la bonne compréhension des mécanismes qui régissent /Services. Nous allons donc en analyser les détails.

La fonction prend deux arguments : **'session**, qui est un objet comprenant les paramètres de la session (voir point précédent) et **'command** qui est un bloc contenant notre commande :

```
[shell/do-it [1 + 1]]
```

Dans un premier temps, les services par défaut dans le contexte de nom **'home**, vont être chargés dans un mot **'servs** :

```
servs: reduce ['home find-service session/config 'home]
```

Après réduction, le bloc **'servs** devient :

```
[home make object! [
   name: 'home
   title: "Home Service"
   description: {Implements the home service (it is always available).}
   dispatch: func [session command /local args result][
       (...)
   ]
   (...)
]
```

soit le nom du contexte suivi de la définition du service...

Nous allons alors établir un mapping entre les variables de travail **'name** et **'command** et les valeurs que l'on retrouve dans le path ! de notre commande :

```
if path? command/1 [
    name: command/1/1
    change command command/1/2
]
```

Dans notre cas, nous obtiendrons :

```
NAME is a word of value: shell
COMMAND is a block of value: [do-it [1 + 1]]
```

Une valeur est donc attribuée au mot **'name**. Cela permet de valider le test suivant, et d'en exécuter le code :

```
if name [
    spec: find-service session/config name
    if not spec [bad-service name]
    insert servs reduce [name spec]
]
```

'spec va donc contenir la définition de notre service, sous forme d'objet, et le nom du service, suivi de ces spécifications, va être inséré avant le service **'home** par défaut, dans le bloc contenu dans variable **'servs**. Nous retrouverons donc dans **'servs** :

```
[shell make object! [
    name: 'shell
    title: "Shell Server"
    description: "An empty service."
    dispatch: func [session command /local args result][
        debug service ['dispatch name command]
        if not rules [
            rules: make-service-rules commands
            bind rules 'args
```

```
        ]
        if not parse command rules [throw 'bad-command]
        result
    ]
    rules: none
    allow: none
    commands: [
        'do-it
        "execute passed command"
        arg: block!
        (result: do arg/1)
    ]
]
home make object! [
    name: 'home
    title: "Home Service"
    description: {Implements the home service (it is always available).}
    dispatch: func [session command /local args result][
        (...)
    ]
    (...)
 ]
```

Nous remarquons que lors du chargement du service au démarrage du serveur, la fonction **'dispatch** lui a été ajoutée. Cette fonction va servir à la création d'un bloc de règles d'analyse pour le parsing de notre commande. Examinons cela en détail.

5 Création des règles d'analyse

Dans la suite du code de la fonction **'do-command**, nous trouvons :

```
fail: catch [result: spec/dispatch session command none]
```

Nous allons maintenant faire appel à la fonction **'dispatch** de notre service, en lui passant deux arguments : notre **'session** et la **'command**. Remarquez que le **'none** n'est là que pour s'assurer que dans tous les cas, une valeur sera attribuée au mot 'result, et donc qu'une erreur non traitée ne sera pas déclenchée.

Si aucune règle d'analyse n'existe, ce qui est le cas lors du premier usage du service, le code suivant est exécuté, faisant appel à la fonction **'make-service-rules** :

```
if not rules [
    rules: make-service-rules commands
    bind rules 'args
]
```

Cette fonction assure la mise en forme des règles d'analyse qui seront appliquées. Examinons-la :

```
make-service-rules: func [
    "Create parse rules from main command dialect"
    commands [block!]
    /local val blk
] [
    blk: make block! (length? commands) / 2
    parse commands [
        some [
            val:
            #doc block!
            | #need [word! | block!]
            (append/only blk to-paren compose/only [need-permission session (val/2)])
            | string!
            | block! (append/only blk make-service-rules first val)
            | skip (append/only blk first val)
        ]
    ]
    blk
]
```

La création des régles d'analyse est donc issue elle-même d'une analyse (parsing) ! Les régles utilisées dans la fonction spécifient, dans l'ordre :

- si l'on rencontre la balise "#doc" suivi d'un block !, ne rien faire, OU
- si l'on rencontre la balise "#need" suivi d'un word ! OU d'un block !, rajouter au bloc **'blk** le résultat issu de la fonction 'need-permission (nous n'entrerons pas dans les détails ici) OU
- si l'on rencontre une string ! (issue de la documentation que l'on peut rajouter au commandes), ne rien faire, OU
- si l'on rencontre un block !, appliquer la même règle de manière récursive, OU
- dans tous les autres cas, rajouter l'élément trouvé à notre bloc **'blk**

Dans notre cas, seul le point 5 va être appliqué, afin d'obtenir un **'blk** contenant :

```
[
    'do-it
    arg: block!
    (result: do arg/1)
]
```

Ce bloc est retourné à la fonction appelante, et il est placé dans le mot **'rules** défini dans la fonction **'dispatch** (voir plus haut).

A la ligne suivante de la fonction **'dispatch**, un binding est réalisé :

```
bind rules 'args
```

Le but est ici de lier les régles d'analyse que nous venons de produire, au contexte local de la fonction ('args est déclaré comme /local à la fonction).

4 Evaluation de la commande

Nous disposons à présent de la commande à exécuter, et des régles de parsing. La ligne suivante de la fonction **'dispatch** assure alors l'évaluation de la commande :

```
if not parse command rules [throw 'bad-command]
```

Dans notre cas nous obtiendrons donc le parsing suivant :

```
parse [do-it [1 + 1]] [
    'do-it
    arg: block!
    (result: do arg/1)
]
```

Ce parsing est à interpréter comme suis :

- si l'on rencontre le mot **'do-it**
- suivi d'un block !
- mettre cette valeur dans la variable **'arg**
- et enfin exécuter le code entre parenthèses

L'exécution de ce code aura pour effet de mettre le résultat de l'évaluation (soit : 2) dans la variable **'result**, qui est retournée à la fonction appelante : **'do-command**.

7 Transmission du résultat de l'évaluation

Aucune erreur n'ayant été déclenchée dans **'do-command**, c'est également la valeur contenue dans **'result** (soit toujours : 2) qui est transmise à la fonction appelante **'do-request**... Rappelez-vous le code en question :

```
result: do-command session command none
```

Quelques lignes plus loin, nous trouvons :

```
append-reply session 'ok cmd result
```

La fonction **'append-reply** va donc former la réponse à notre requête, et placer celle-ci dans notre session, pour transmission vers le client.

8 Réception par le client

Enfin, à l'autre bout du fil du réseau, notre client reçoit le résultat de la commande :

```
[
    done [reply seq 1 service "Generic REBOL Service" commands 1 time 0:00:00.01]
    ok [shell/do-it 2]
```

```
]
```

où nous retrouvons deux paires word !/block !

La première :

```
done [reply seq 1 service "Generic REBOL Service" commands 1 time 0:00:00.01]
```

est un résumé de la requête. Le mot **'done** indique qu'aucune erreur ne s'est produite et que la requête a été complètement traitée. Nous y retrouvons différentes informations, ainsi que le temps mis par la commande pour être exécutée.

La seconde paire :

```
ok [shell/do-it 2]
```

nous indique que la commande s'est bien exécutée ("ok"), rappelle le nom de la commande ("shell/do-it") et donne le résultat obtenu ("2").

Une alternative aurait été, en imaginant que nous n'avions pas les permissions nécessaires à l'exécution de la commande :

```
fail [shell/do-it not-allowed]
```

J'espère que ces quelques lignes ont pu vous aider dans la voie de la compréhension de ces magnifiques /Services :-)

Merci de m'avoir lu !