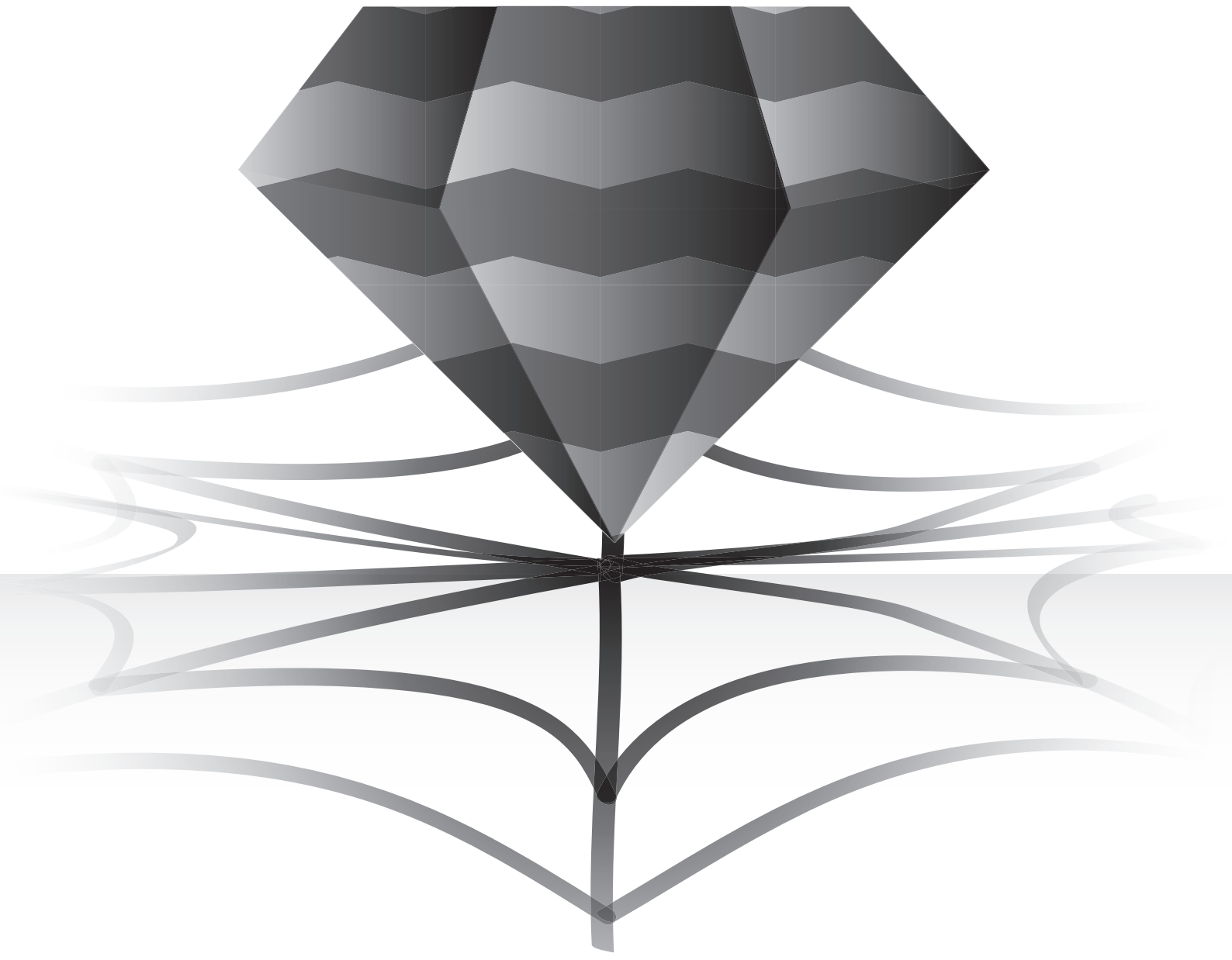
RR-79

REST from Scratch



Caelum
Ensino e Inovação

www.caelumobjects.com

Caelum provides training in software development with its team of highly qualified instructors in Java, Ruby and Rails, Agile and Rest services. Its team is spread in several cities around Brazil, being responsible for hundreds of training classes an year. Caelum has a strong presence on the Brazilian Open Source market, including the creation and official training for Restfulie, VRaptor and Stella.

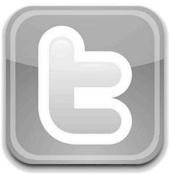
Caelum is currently responsible for GUJ, the largest portuguese language online java forum, the brazilian site of INFOQ.br and several conferences in the country, incuding QCon São Paulo and Caelum Day, with more than 600 attendees in its latest edition.

Caelum's mission is to deliver high quality training and presentations, through updated material and a specific teaching methodology developed to leverage every one previous experiences in a bottom up approach. Caelum also provides training and mentoring in specific projects within companys.

Learn more

Follow our company blog for news on the frameworks, events, theory and much more by accessing its feed:

➡ blog.caelumobjects.com



Follow Restfulie at **Twitter**:

➡ twitter.com/restfulie

Visit Restfulie's web site for extra information on the framework itself:

➡ restfulie.caelumobjects.com

Caelum

"Kill time and you will be killing your cereer"
Bryan Forbes

About Caelum

Caelum acts on the market since 2002, developing systems and as consultant in many areas, always under the Java platform. It was founded by professionals who met in Brazil after experiences on Germany and Italy, developing wide systems, integrated with the most diverse ERPs. Its professionals have already published many articles on Brazilian magazines about Java, as well as academic articles, and are constant presence on technology events.

In 2004, Caelum made a range of courses that rapidly got acceptance in the market. The courses were elaborated by Sun's ex-intructors who would like to bring more dinamism and apply the tools and libraries used on the market, such as Eclipse, Hibernate, Struts, and other opensource technologies, not mencioned by Sun. Caelum material were assembled by Caelum instructors during Java summer courses at University of São Paulo, SP, Brazil, on January, 2004.

In 2007, the company focus its efforts in two great areas:

- Development consulting, mentoring and coaching
- Trainings with formation intention

On the textbook

This Caelum textbook intends teaching Java in a elegant way, showing the very necessary when it is needed, in the right moment, saving the reader from subjects that are usually not interesting to them in specific phases of the learning process.

Caelum hopes you enjoy this material and that it can be of great value to self-thaught people and students. All comments, criticism and sugestions are welcome.

Disclaimer: You will find the textbook version by the end of the table of contents. We do not recommend printing the textbook you might have received from a friend or by e-mail, as we constantly update this material, almost monthly. Go to our website and grab the last version of it.

www.caelumobjects.com.br

Contents

Version: 12.7.31

CHAPTER 1

HTTP

1.1 - REST

The web has proved to be the largest distributed system humankind has built so far, allowing new systems to integrate with older clients. It has also scaled to a global level. What has made the web succeed where other protocols and ideas have failed?

Back in 2000, Roy Fielding published his dissertation where he studied the key aspects of the web and other architectures, deriving a new one, the REST style architecture. Server and client systems built on top of such architecture are believed to leverage from the web in the same way that we, humans, do nowadays.

1.2 - Scalability

Scalability can be generically achieved in two different ways. But what is scaling?

When delivering water to the population, one used to dig deep down and install a pipe which allows water to be sent from an origin to a destination. In the same way, cables allowed us to call relatives by being installed within pipes buried under our streets.

Measuring scalability means measuring flow capacity. How much can we send from our source to different destinations during a specific amount of time? If I have a direct connection between a water processing facility and my users, am I able to cope with 500 using it at the same time?

1.3 - Scaling vertically

Whenever the pipes were too small, one would change it to a larger one, having to pay for costs on creating a new, replacing and destroying the old one. And this would occur every time our system was not able to cope with the required water flow: our system did not scale enough.

One would end up with huge specialized pipes and an amazing amount of wasted labor and material that was used during this process.

If the pipe break in any place, there is a huge number of clients without access to the system. In order to cover that, another huge specialized pipe needs to be put into place, so our system keeps running in case the first set stops working, doubling the amount of money spent in our system. If we need to supply scalability to up to N clients, our company will pay for 2*N in order to provide a fault tolerant system.

Following the same mindset, the water pipe was not able to cope with cables thus requiring a new set of connections between the source and all destinations in order to provide this second type of service.

Again, money was wasted in a parallel system that achieves almost the same thing as the previous one: distributing something over a network of pipes.

Suppose a distributed system following such principles requires 1 million dollars to scale up to 1 million users; in order to provide three fault tolerant services to the same users, a company would spend some 6 million dollars to support.

But what would happen if all services were provided through the same means? In a uniform way? And what if we were able to tackle faults without doubling our costs?

We would be able to support higher levels of faults in our distributed environment with lower costs.

1.4 - Scaling horizontally

This has already happened to the web, partially to electricity, and is happening nowadays to cloud computing. The same process can change costly web services into services that can be distributed in a cheaper, and uniform way.

First, imagine that instead of one huge origin that provides water to the entire population, a huge server that needs to handle all system requests, several smaller origins or several smaller servers are able to understand and answer it.

If any single failure occurs, the failure rate on the first approach is 100%. In order to respond to it, we add a backup origin, and now, with doubled costs, we accept one failure and keep to 0% fail. If two origins present any problem, we come back to 100% failure rate and in order to cope with this scenario, our costs will be tripled.

That's why your country do not rely in one huge energy station, in the world's hugest water processing facilities. Appart from high maintanance costs, any single failure might be fatal to the system and take too long to recover.

Instead, you rely in dozend of power plants and water processing facilities. If any one of those fails, (N-1/N) will still run. The software world is even more advanced that energy or water distribution: if any software can run in any machine, any single failure might be fixed by activating one of the origins that were not activated so far, implying in no extra costs, apart from copying your data from one place to another.

That's why cloud computing is a cheaper and promisses to be even more successful than water processing and energy production.

Both origin - processing - and distribution can be scaled? Distribution is not executed on the source, it happens on the pipes... in our case, over the web.

Let's take a look at that.

1.5 - Client Server

First of all, the web is a client server distributed system. We have mentioned water and energy distribution which provide similar resources to clients with a distinct difference: a water distribution system only provides water, but the web provides any kind of resource, it allows any type of system to connect to its network of pipes and run through it.

Concerns related to data storage and retrieval can be kept on one or multiple servers - connected through hyperlinks - and others on the client: in a REST architecture, this separation allows client and server to evolve independently, through loosily coupling.

1.6 - Stateless

Imagine that whenever one turns on his lights for the first time on a newly built house, the power plant providing this energy records such information in a way that any further needs that this house might have can only be provided by this power plant. This does not scale in a costly effective manner because *each* power plant would have to be built to support thousands in peak hours, although not necessarily true during the rest of the day.

In a system that any server can answer to your demands, processing costs can be shared between them all, requiring a larger number of less powerful servers: higher fault tolerance and lesser costs.

Any information required for server processing shall be made available to every server, so any one of them can answer to this request.

In web words, do not keep session state on your server, although it is fine to keep session state in the client - in many different ways.

1.7 - Cache

Caching is the ability to keep previously retrieved data stored in order to avoid hitting a source whenever its required again.

In water distribution, caching is typically implemented in two different ways. On the way to clients, water is stored in water castles. When source failures occur, we still have a running system for a specific amount of time. Clients are unaware that they are being served through those intermediates.

Water castles, as intermediate caches, adds fault tolerancy to such systems.

Cache over the web, and in REST systems is even better than those water and electricity intermediate repositories. Cached information in REST systems can nurture any amount of clients during a specific amount of time: the cache will only need to be refreshed if its cache expires, not once before that.

Meanwhile, resources - water, energy our web resources - can also be cached within the client. The water distribution system is push based: it pushes water down the flow to your house, where you can store it prior to usage. Web resources are pulled from the server, requested through a message that is sent up the network. In this system, keeping local caches within the clients will save not only bandwidth but response time, because there is not latency when there is no request.

Creating systems where resources can be cached is a keypoint to horizontal scalable systems built on top of the REST architecture.

Stateless systems that provide cacheable resources imply in lesser costs as bandwidth is saved, latency is minimized, fault tolerancy is increased and processing costs is divided between all origins.

1.8 - Uniform interface

But what would happen if all services were provided through the same means? In a uniform way? And what if we were able to tackle faults without doubling our costs?

As soon as a intermediary, a client or a server sees something that is compatible with that uniform interface, it knows what to expect from that.

They all know what that thing is capable of, what its intentions are. And it also knows that, whatever it does with it, while staying compatible with the uniform interface, no one will ever notice.

In REST, resources are identified in a uniform manner, using URIs. Resources are manipulated through its representations, i.e. an atom representation of a collection or an open search document definition.

Messages are self descriptive because all knowledge required to understand it is contained within the message itself.

Finally, application state is changed by following hyperlinks.

Those four constraints compose REST's uniform interface.

1.9 - Layered System

Because an uniform interface is into place, any intermediate might take decisions that affect the result in a way that neither the server nor the client will notice, implementing features, improving response time, allowing scalability etc.

Imagine an old client server system that used to exchange representations in a format, i.e. json. If that system is too complex to be refactored, xml support can be added by simply adding an intermediate which transforms json into xml.

Response time and scalability is improved by implementing caches in intermediate layers, gateways and a reverse proxy on the server itself.

1.10 - Code-On-Demand

The final characteristic of a REST architecture is the code on demand. This topic has not yet been widely debated and there is still room for discussion and improvement. In the human web, code on demand is well accepted as javascript code downloaded and run in our clients.

This Javascript runs in a sandbox, along with other security restrictions, thus clients trust this code.

On the machine to machine world, javascript could be downloaded and execute media, format or even business logic accessing the same server where the code came from, in a similar manner that it happens with browsers. This raises *the same security issues* as with the human web, none extra.

Code on demand allows clients to learn new features, developers to get notified when things changes and less code to be written on clients.

1.11 - Measuring your maturity

In order to teach REST over to developers who are used to web development, Leonard Richardson came up with a model that simplifies this process.

Web services became popular with the use HTTP for tunneling, with only one URI and verb: POST. Servers designed with this style show a lot of tight coupling between itself and the client. Those systems are Level 0.

Level 1 is how most web services were implemented over the last decade: multiple URIs representing different systems, using only one verb: POST, while its posted message defines what is to be executed. This is how web services were sold to companies.

Level 2 adds the use of an uniform interface to act upon resources. Rails has done a great job popularizing this level, by allowing different http verbs to execute their expected actions over resources, i.e. a DELETE verb shall remove a resource. One of the benefits from level 3 over level 2 is the lesser coupling between clients and server, amongst with extra visibility because all intermediates as cache and proxies know what is happening prior to acting.

Level 3 adds hypermedia support, where resources get linked to each other. In html this is done through “a” elements and we have been using it for over two decades. Linked data allows client to discover related resources and act upon them.

It also adds a new layer of scalability possibilities: sharding is easier because some resources might be located on the same server while others in other servers, it is just a matter of where the URI points to.

Those 3 steps brings someone’s APIs closer to REST, trying to minimize the coupling between client and server.

In the next section we shall see how to achieve a full REST architecture, both on the server and client side, and next, how to implement such a system.

1.12 - Rule of thumb

Use HTTP There is a huge infrastructure available for systems that treat the web as it was really build, following the HTTP specification. The web is the broadest, amazingly loosely coupled and most scaled system human system.

Stick to existing HTTP verbs If not, you break the uniform interface, losing visibility and scalability.

Connect your resources If not, an entire system composed of resources without links tight couples the client to your server, by knowing too many URIs.

CHAPTER 2

Getting started

This chapter will go through creating and accessing servers that provide APIs based on HTTP, with different URIs and methods, such as the twitter API.

2.1 - Installing restfulie

First of all, install restfulie with one line of code in your console/terminal:

```
gem install restfulie
```

Rubygems will install restfulie's latest release and all its dependencies:

```
Successfully installed nokogiri-1.4.2
Successfully installed activesupport-2.3.8
Successfully installed rack-1.1.0
Successfully installed actionpack-2.3.8
Successfully installed responders_backport-0.1.2
Successfully installed json_pure-1.4.3
Successfully installed restfulie-0.8.0
7 gems installed
Installing ri documentation for nokogiri-1.4.2...
Installing ri documentation for activesupport-2.3.8...
Installing ri documentation for rack-1.1.0...
Installing ri documentation for actionpack-2.3.8...
Installing ri documentation for responders_backport-0.1.2...
Installing ri documentation for json_pure-1.4.3...
Installing ri documentation for restfulie-0.8.0...
Installing RDoc documentation for nokogiri-1.4.2...
Installing RDoc documentation for activesupport-2.3.8...
Installing RDoc documentation for rack-1.1.0...
Installing RDoc documentation for actionpack-2.3.8...
Installing RDoc documentation for responders_backport-0.1.2...
Installing RDoc documentation for json_pure-1.4.3...
Installing RDoc documentation for restfulie-0.8.0...
```

Because we are going to use some xml based representations, libxml is our basic serialization library:

```
gem install libxml-ruby
```

2.2 - Accessing twitter

Let's parse twitter's public timeline, creating a twitter.rb file:

```
require 'rubygems'
require 'restfulie'

result = Restfulie.at("http://twitter.com/statuses/public_timeline.xml").get
result.statuses.each do |status|
  puts "#{status.user.screen_name}: #{status.text}, #{status.created_at}"
end
```

And let's run it:

```
ruby twitter.rb
```

First of all, Restfulie will retrieve the contents from the public timeline by sending a GET request:

```
GET /statuses/public_timeline.xml
```

After that you will get all messages printed, including the users screen name.

Due to http URIs and verb usage, twitter's web service api is easy to access. But how could it be improved in a REST point of view in two easy steps?

2.3 - Possible improvements

To minimize coupling between client and server, client's would be able to navigate throughout the system.

If you think about the human web, where users discover resources by navigating through links, they are not supposed to type in server specific uri patterns in order to find out what is available.

By linking resources, Twitter's API could link to Facebook user who "retwited" a message. You, as a client software, would get to that user profile without realizing its another server.

Without links, that is not possible without further coupling and client changes.

Understanding the "Accept" header would improving visibility, instead of patterned URIs (.xml, .json and so on).

2.4 - Installing rails

Let's install rails and sqlite3, a lightweight database to support our application with minimal configuration:

```
gem install rails sqlite3-ruby
```

```
Successfully installed activerecord-2.3.5
Successfully installed actionmailer-2.3.5
Successfully installed activeresource-2.3.5
Successfully installed rails-2.3.5
4 gems installed
Installing ri documentation for activerecord-2.3.5...
```

```
Installing ri documentation for actionmailer-2.3.5...
Installing ri documentation for activeresource-2.3.5...
Installing ri documentation for rails-2.3.5...
Installing RDoc documentation for activerecord-2.3.5...
Installing RDoc documentation for actionmailer-2.3.5...
Installing RDoc documentation for activeresource-2.3.5...
Installing RDoc documentation for rails-2.3.5...
```

2.5 - Creating our store

Let's create our store as a typical Rails model and controller:

```
rails store
cd store
script/generate model Item name:string price:decimal
script/generate controller items
rake db:create db:migrate
```

```
> rails store
      create
      ...
> cd store
> script/generate model Item name:string price:decimal
      exists  app/models/
      exists  test/unit/
      exists  test/fixtures/
      create  app/models/item.rb
      create  test/unit/item_test.rb
      create  test/fixtures/items.yml
      create  db/migrate
      create  db/migrate/20100520131931_create_items.rb
> script/generate controller items
      exists  app/controllers/
      exists  app/helpers/
      create  app/views/items
      exists  test/functional/
      create  test/unit/helpers/
      create  app/controllers/items_controller.rb
      create  test/functional/items_controller_test.rb
      create  app/helpers/items_helper.rb
      create  test/unit/helpers/items_helper_test.rb
```

And finally let's add some items to our database:

```
script/console
Item.create :name => "Caelum Rest Training", :price => "400"
Item.create :name => "Water", :price => "2"
Item.create :name => "Rest in veneza", :price => "800"
```

We finish configuring the rails model by mapping the routes at routes.rb:

```
map.resources :items
```

2.6 - Configuring Restfulie

Open your *config/enviroment.rb* file and add the restfulie gem:

```
config.gem "restfulie", :version => ">= 0.8.1"
```

2.7 - Creating our controller

Using a backported feature from Rails 3, we shall list all Items:

```
class ItemsController < ApplicationController

  def index
    @items = Item.all
    respond_with @items
  end

end
```

And notify Rails that we are capable of handling xml and json request:

```
class ItemsController < ApplicationController

  respond_to :xml, :json
    ...

end
```

And the first line of Restfulie code is to actually notify that this controller behave as a Restfulie controller:

```
class ItemsController < ApplicationController

    acts_as_restfulie
    ...

end
```

Now that we are ready to test our list, you can curl or wget your resources, which returns all related headers:

```
> curl -i "http://localhost:3000/items"
HTTP/1.1 200 OK
Etag: "dce691598ce7d64cdd6d1dba1c4025fd"
Connection: Keep-Alive
Content-Type: application/xml; charset=utf-8
Date: Thu, 20 May 2010 13:40:39 GMT
Server: WEBrick/1.3.1 (Ruby/1.8.7/2010-01-10)
X-Runtime: 11
Content-Length: 823
```

```
Cache-Control: private, max-age=0, must-revalidate
```

And the xml representation:

```
<?xml version="1.0" encoding="UTF-8"?>
<items type="array">
  <item>
    <created-at type="datetime">2010-05-20T13:35:09Z</created-at>
    <id type="integer">1</id>
    <name>Caelum Rest Training</name>
    <price type="decimal">400.0</price>
    <updated-at type="datetime">2010-05-20T13:35:09Z</updated-at>
  </item>
  <item>
    <created-at type="datetime">2010-05-20T13:35:09Z</created-at>
    <id type="integer">2</id>
    <name>Water</name>
    <price type="decimal">2.0</price>
    <updated-at type="datetime">2010-05-20T13:35:09Z</updated-at>
  </item>
  <item>
    <created-at type="datetime">2010-05-20T13:35:09Z</created-at>
    <id type="integer">3</id>
    <name>Rest in veneza</name>
    <price type="decimal">800.0</price>
    <updated-at type="datetime">2010-05-20T13:35:09Z</updated-at>
  </item>
</items>
```

Note that content negotiation took place: you did not ask for a specific representation, so Restfulie chose the first one, xml. If you want a json representation, use the Accept header, as the HTTP specification tell us:

```
> curl -i "http://localhost:3000/items" -H "Accept: application/json"
HTTP/1.1 200 OK
Etag: "362c96b10cef571c9b710409fce0e321"
Connection: Keep-Alive
Content-Type: application/json; charset=utf-8
Date: Thu, 20 May 2010 13:42:50 GMT
Server: WEBrick/1.3.1 (Ruby/1.8.7/2010-01-10)
X-Runtime: 10
Content-Length: 380
Cache-Control: private, max-age=0, must-revalidate
```

The Content-Type header indicated an application/json response, which can be seen in its body:

```
[{"item":
{"price":400.0,"name":"Caelum Rest Training",
"created_at":"2010-05-20T13:35:09Z","updated_at":"2010-05-20T13:35:09Z","id":1}},
{"item":{"price":2.0,"name":"Water","created_at":"2010-05-20T13:35:09Z",
"updated_at":"2010-05-20T13:35:09Z","id":2}},{"item":{"price":800.0,
"name":"Rest in veneza","created_at":"2010-05-20T13:35:09Z",
```

```
"updated_at":"2010-05-20T13:35:09Z","id":3}}]
```

2.8 - Restfulie client

It is time for us to access our products through code. Let's execute a GET request, using content negotiation:

```
> require 'rubygems'
> require 'restfulie'
> resource = Restfulie.at('http://localhost:3000/items').get
{"items"=>[{"price"=>#<BigDecimal:187e0b4,'0.4E3',4(8)>, "name"=>"Caelum Rest Training",
"created_at"=>Thu May 20 13:35:09 UTC 2010,
"updated_at"=>Thu May 20 13:35:09 UTC 2010, "id"=>1},
{"price"=>#<BigDecimal:187c728,'0.2E1',4(8)>, "name"=>"Water",
"created_at"=>Thu May 20 13:35:09 UTC 2010,
"updated_at"=>Thu May 20 13:35:09 UTC 2010, "id"=>2},
 {"price"=>#<BigDecimal:187ad9c,'0.8E3',4(8)>,
"name"=>"Rest in veneza", "created_at"=>Thu May 20 13:35:09 UTC 2010,
"updated_at"=>Thu May 20 13:35:09 UTC 2010, "id"=>3}]}

> puts resource.response.headers['content-type']
application/xml; charset=utf-8

> puts resource.items.size
3

> puts resource.items.last.name
Caelum Rest Training
```

Restfulie already unmarshalled our representation because it comes with some media type handlers out of the box. If it did not understand the media type returned it would return the response as raw data so you as a client can decide what to do next.

Let's see how Restfulie would behave with a json representation by adding an *accept('application/json')* to our dsl chaining.

```
require 'rubygems'
require 'restfulie'
resource = Restfulie.at('http://localhost:3000/items').accepts('application/json').get
```

> **Media types**
>
> The json media type work with collections in a slighlty different way than the xml media type. This is natural because representations in different media types will tipically imply in different semantic values. As we will see later, xml without schemas and json might not provide enough information to rest clients.
> You will learn how to support custom or other media types in a later section of the course.

Our returned object contains the response, as usual. According to a response media type, I, as a client, know which structure can be found within the body of the response, therefore in the *application/json* case, one should check the size of the array and access its elements:

```
> puts resource.response.headers['content-type']
application/json; charset=utf-8
```

```
> puts resource.size
3

> puts resource.items.last.name
Caelum Rest Training
```

Because both the xml and json unmarshallers deal, by default, with Hashes, you might discover fields by accessing the typical Hash methods (keys, values, each and so on).

> **Avoid mapping responses to types**
>
> Because Restfulie uses Hashes by default in those two cases, it leverages your system by following the Must-ignore rule: new fields will not kill your client application. As we will see later one can even diminish the binding because in REST application one does not know what happens after following a link.

For now, remember that you can specify your media type and access it as usual objects.

2.9 - Error codes and exceptions

If you invoke methods as *get*, *post*, *delete* and so on, Restfulie will return the response even if it is an error response (i.e. 4XX, 5XX). If you want Restfulie to thrown an exception upon those response codes, simply add a *!* to the method invocation: *get!*.

2.10 - Creating and accessing a product on the server side

By creating the show method as we usually do in Rails application, Restfulie will be capable of serving xml and json representations of your item:

```
def show
  @item = Item.find(params[:id])
  respond_with @item
end
```

Resource creation works almost the same way as it does over the typical human web interface, although instead of returning a redirect response, we will answer with a 201, which means that the resource was created. This is better usage of the HTTP protocol, the protocol we accepted to support our architecture:

```
def create
  @item = Item.create(params[:item])
  render :text => "", :status => 201, :location => item_url(@item)
end
```

We are ready to give it a try, sending an xml representation of our item:

```
curl -i -H "Content-type: application/xml" -d
"<item><name>Caelum Business Processes over REST training</name><price>600
</price></item>" "http://localhost:3000/items"
```

Note that the response notifies us where to find the newly created resource:

```
HTTP/1.1 201 Created
Location: http://localhost:3000/items/4
Connection: Keep-Alive
Content-Type: text/html; charset=utf-8
Date: Thu, 20 May 2010 14:20:53 GMT
Server: WEBrick/1.3.1 (Ruby/1.8.7/2010-01-10)
X-Runtime: 71
Content-Length: 0
Cache-Control: no-cache
```

Background processing

Sometimes a resource creation depends on some batch our background process, in those cases, according to the http specification, return a 202 and a pointer to another URI that can be used to check the request status.

2.11 - Creating and accessing via console

Extracting the item would be a simple get instruction:

```
curl "http://localhost:3000/items/4"
```

```
<?xml version="1.0" encoding="UTF-8"?>
<item>
  <created-at type="datetime">2010-05-20T14:20:53Z</created-at>
  <id type="integer">4</id>
  <name>Caelum Business Processes over REST training</name>
  <price type="decimal">600.0</price>
  <updated-at type="datetime">2010-05-20T14:20:53Z</updated-at>
</item>
```

Now let's create and retrieve it using JSON:

```
> curl -i -H "Content-type: application/json" -d "{ 'item' :
 {'name' : 'REST is not cute URIs book', 'price' : 100 } }" "http://localhost:3000/items"
HTTP/1.1 201 Created
Location: http://localhost:3000/items/5
Connection: Keep-Alive
Content-Type: text/html; charset=utf-8
Date: Thu, 20 May 2010 14:25:02 GMT
Server: WEBrick/1.3.1 (Ruby/1.8.7/2010-01-10)
X-Runtime: 11
Content-Length: 0
Cache-Control: no-cache

> curl "http://localhost:3000/items/5" -H "Accept: application/json"
{"item":{"price":100.0,
"name":"REST is not cute URIs book","created_at":"2010-05-20T14:25:02Z",
"updated_at":"2010-05-20T14:25:02Z","id":5}}
```

2.12 - Creating and accessing a product on the client side

Finally, its time to use Restfulie's API to create and access single products, including following redirections hints as the Location header.

First of all, let's post our content. Remember that one *must* specify the content type whenever posting something as an entry point. This is a requirement by the http protocol:

```
resource = Restfulie.at('http://localhost:3000/items')
.as("application/xml").post!(:item => { :name => "Robin costume", :price => 25.0})
```

A resource is created and Restfulie follows the 201 hint:

```
POST /items
Content-Type: application/xml
Accept: application/xml
"<?xml version=\"1.0\" encoding=\"UTF-8\"?>\n
<item>\n  <price type=\"float\">25.0</price>\n  <name>Robin costume</name>\n</item>\n"

GET /items/7
Accept: application/xml
 => {"item"=>{"price"=>#<BigDecimal:18efa84,'0.25E2',4(8)>,
"name"=>"Robin costume", "created_at"=>Thu May 20 14:49:45 UTC 2010,
"updated_at"=>Thu May 20 14:49:45 UTC 2010, "id"=>7}}
```

If you want to access the resource, do it as you would normally do:

```
> p resource.item.name
"Robin costume"
```

Now we will do the same using json:

```
resource = Restfulie.at('http://localhost:3000/items').as("application/json")
.post!(:item => { :name => "Batman costume", :price => 26.0})

POST /items
Content-Type: application/json
Accept: application/json
"{\"item\":{\"price\":26.0,\"name\":\"Batman costume\"}}"

GET /items/8
Accept: application/json
 => {"item"=>{"name"=>"Batman costume",
"price"=>26.0, "created_at"=>"2010-05-20T14:52:30Z",
"updated_at"=>"2010-05-20T14:52:30Z", "id"=>8}}
```

By default, Restfulie serialize hashes and classes using their own *to_xml* and *to_json* methods.

> **Custom content**
>
> If you want to customize the json or xml yourself, pass a String to the post method. Both json and xml media type handlers will post that content if it is not a Hash, but a String.

2.13 - What is still missing?

Take a look at the entire items collection, if you have it in your hands, how can you add an item to it? This representation does not provide any information on how to interact with it and related resources. In the human web, a list representation would provide hyperlinks (and forms) that allow you to create, remove and access specific elements. Or do you have to type in an URI every time you want to read a product's description at amazon.com? You do not, because amazon - and any other loosely coupled system - did not want you to memorize all that information specific to their servers.

This is *vital* to a REST architecture loose coupling, and we shall do it in the next chapters.

There is a long way to go, but the first thing is to be able to provide even more media types, as atom. Then we shall be able to provide links between resources and finally create truly restful clients by using dsls to describe our intentions. The following two chapters will take care of those steps.

2.14 - Exercises

1) Create a new rails project with support to restfulie
2) Create a new model, called Hotel and add it's name, room count and price.
3) Make its list available through xml and json.
4) Access it through a Restfulie client.
5) Create methods to support hotel creation and accessing.
6) Create a new hotel using Restfulie client.
7) Implement support for product removal through a DELETE request to its URI.

Linking resources

3.1 - Baskets

We will expand our system to support baskets with the basic Rails support to them. First, generate the model, controller and a migration that will add items to our baskets.

```
script/generate model Basket
script/generate controller baskets
script/generate migration add_items_to_baskets
```

Opening the migration file on the text editor, the basket to item mapping table is created in our relational database.

Note that this table does not contain an id column.

```
class AddItemsToBaskets < ActiveRecord::Migration
  def self.up
    create_table :baskets_items, :id=> false, :force => true do |t|
      t.integer :basket_id
      t.integer :item_id
      t.timestamps
    end
  end

  def self.down
    drop_table :baskets_items
  end
end
```

And migrate our system:

```
rake db:migrate
```

Rails keep its routes in the routes.rb file, thus we will map our basket resource on that file:

```
map.resources :baskets
```

3.2 - Customizing our representations

Representations can be customized in a uniform way. If we had to customize the representation for each different media type, it would take a lot of effort from the developers to support them.

Let's customize our items list and add a link to a basket resource. By creating a relation named basket and documenting it, our clients are aware that creating such a resource will create a basket, accessing such a

resource will access a basket.

Restfulie uses an internal template engine called tokamak, which is similar to xml builders on the Ruby platform. Creating a index.tokamak file in the *app/views/items* will notify Restfulie to use it instead of the default rendering.

This *app/views/items* file will describe the *items* collection, therefore:

```
collection(@items) do |collection|
end
```

Describing the collection but not its members will just render an empty collection:

```
> curl http://localhost:3000/items


<?xml version="1.0"?>
<items/>
```

3.3 - Creating links

This is not what we want, we are looking for a representation that will describe all its members and a link to a relation called basket.

In order to add a link, simply invoke the link method, passing the relation name and uri as parameters to it:

```
collection(@items) do |collection|

  collection.link "basket", baskets_url

end
```

> **Extra parameters**
>
> The link method supports extra parameters that will be appended as attributes in the link tag.

Executing the same *GET* request we did earlier, one receives the items collection with a link to the basket:

```
> curl http://localhost:3000/items


<?xml version="1.0"?>
<items>
  <link href="http://localhost:3000/baskets" rel="basket" type="application/xml"/>
</items>
```

The link itself already gave a hint on which media type can be used for further processing.

3.4 - Describing members

We are left with the task to add members to our collection. The *members* method will execute its block for each member within the collection, therefore one can achieve:

```
collection(@items) do |collection|
```

```
  collection.link "basket", baskets_url

  collection.members do |member, item|
  end

end
```

Which will result in a series of empty item nodes:

```
>curl http://localhost:3000/items


<?xml version="1.0"?>
<items>
  <link href="http://localhost:3000/baskets" rel="basket" type="application/xml"/>
  <item/>
  <item/>
  <item/>
  <item/>
  <item/>
</items>
```

Because we want to describe our items, let's invoke the partial rendering method:

```
collection(@items) do |collection|

  collection.link "basket", baskets_url

  collection.members do |member, item|
     partial "show", binding
  end

end
```

And create the *app/views/items/_show.tokamak* view, that adds elements to our representation:

```
member.link :self, item_url(item)
member.values { |values|
  values.id     item.id
  values.name   item.name
  values.price  item.price
}
```

The final result is a series of items, including its member values and links:

```
> curl http://localhost:3000/items


<?xml version="1.0"?>
<items>
  <link href="http://localhost:3000/baskets" rel="basket"
     type="application/xml"/>
  <item>
    <link href="http://localhost:3000/items/1" rel="self" type="application/xml"/>
    <id>1</id>
```

```
    <name>Calpis</name>
    <price>10.0</price>
  </item>
  <item>
    <link href="http://localhost:3000/items/2" rel="self" type="application/xml"/>
    <id>2</id>
    <name>Rest in practice</name>
    <price>40.0</price>
  </item>
  <item>
    <link href="http://localhost:3000/items/3" rel="self" type="application/xml"/>
    <id>3</id>
    <name>Caelum Business Processes over REST training</name>
    <price>600.0</price>
  </item>
  <item>
    <link href="http://localhost:3000/items/4" rel="self" type="application/xml"/>
    <id>4</id>
    <name>REST is not cute URIs book</name>
    <price>100.0</price>
  </item>
  <item>
    <link href="http://localhost:3000/items/5" rel="self" type="application/xml"/>
    <id>5</id>
    <name>Robin costume</name>
    <price>25.0</price>
  </item>
</items>
```

3.5 - Controlling baskets

Because our clients want to know how much they need to pay for their baskets, we will add a price method in our basket model class:

```
class Basket < ActiveRecord::Base

  def price
    items.inject(0) do |total, item|
      total + item.price
    end
  end

end
```

We also need to notify Rails of our relation between baskets and items by invoking has_and_belongs_to_many:

```
class Basket < ActiveRecord::Base

    has_and_belongs_to_many :items

  def price
    items.inject(0) do |total, item|
```

```
      total + item.price
    end
  end

end
```

3.6 - Supporting payments

Our clients want to pay for the items they have selected so far, adding them in baskets, and in order to keep track of those payments, we will create a payment class.

Our payment will contain the card number, holder, the amount being paid and a relation to which basket is being paid by this object:

```
script/generate model Payment cardnumber:string cardholder:string
    amount:decimal basket_id:integer

script/generate controller payments

rake db:migrate
```

As a service provider, we want our clients who have a list of desired items in their hands, a basket, access allow a payment to be made by accessing its payment relation. In order to do so, lets map the basket relation within the Payment class.

```
class Basket < ActiveRecord::Base
    has_many :payments
end
```

A final change to our routes.rb file will map payments within baskets:

```
map.resources :baskets, :has_many => [:payments]
```

3.7 - Basket creation

As we did in the items controller, we first configure the BasketsController to respond both to xml and json requests, and include the Restfulie extension to action controller:

```
class BasketsController < ApplicationController

  acts_as_restfulie

  respond_to :xml, :json

end
```

As with item creation, Restfulie will parse representations and give access to them through simple parameters.

> **Media types**
>
> By now we are using *application/xml* and *application/json* as media types, which do not provide hypermedia definition as a media type themselves. We will later understand how to support vendorized or custom media types simultaneously.

Supposing the basket schema defines a basket root element and a list of items and ids contained within them, we shall add each requested item to the basket:

```
def create
    @basket = Basket.new
    params[:basket][:items].each do |item|
      @basket.items << Item.find(item[:id])
    end
    @basket.save
end
```

It could also be extended to support an item *uri* as its identifier instead of its database id. For our clients there will be no difference because our representations contain a link to *self*. It is a matter of choice and the positive aspect of using *uris* would be *if the media type definition implies that those ids can be accessed as uris*.

We still need to render the resource with the 201 response code, as done earlier with items:

```
render :text => "", :status => 201, :location => basket_url(@basket)
```

After that it is important to create the show method, which is nothing but the traditional find that we implemented earlier:

```
def show
  @basket = Basket.find(params[:id])
  respond_with @basket
end
```

```
require 'rubygems'
require 'restfulie'

# retrieves the list
resource = Restfulie.at("http://localhost:3000/items").accepts("application/xml").get

# picks the first item
basket = {"basket" => {"items" => [{"id" => resource.items.item.first["id"]}]}}

# creates the basket
resource = resource.items.links.basket.post! basket
```

But if you inspect the resource, there is no information from the basket because the default serialization process did not find any fields to serialize - and it did not follow any relationships.

It is our turn to create our representation description using tokamak. Instead of describing a collection, we shall describe a single member, with a link to its payments and itself by creating a file called *show.tokamak* at *apps/views/basket/show.tokamak*.

```
member(@basket) do |member, basket|
  member.link "payment", basket_payments_url(@basket)
  member.link "self", basket_url(@basket)
end
```

Retrieving the basket we have previously created will return us a representation with those links:

```
curl http://localhost:3000/baskets/1
```

```
<?xml version="1.0"?>
<basket>
  <link href="http://localhost:3000/baskets/1/payments" rel="payment"
        type="application/xml"/>
  <link href="http://localhost:3000/baskets/1" rel="self"
        type="application/xml"/>
</basket>
```

Finally, beginning a values block allows us to create *any* element in the xml but invoking it as a method:

```
member.values { |values|
  values.price @basket.price # creates a price element
}
```

The entire *show.tokamak* file is:

```
member(@basket) do |member, basket|
  member.link "payment", basket_payments_url(@basket)
  member.link "self", basket_url(@basket)

  member.values { |values|
    values.price @basket.price
  }
end
```

```
curl http://localhost:3000/baskets/1
```

```
<?xml version="1.0"?>
<basket>
  <link href="http://localhost:3000/baskets/1/payments" rel="payment"
        type="application/xml"/>
  <link href="http://localhost:3000/baskets/1" rel="self"
        type="application/xml"/>
  <price>40.0</price>
</basket>
```

The same request, now with the Accept header set for json, will return a json representation including links:

```
curl http://localhost:3000/baskets/1 -H "Accept: application/json"
```

```
{"price":40.0,"link":[{"href":"http://localhost:3000/baskets/1payments",
"rel":"payment","type":"application/json"},{"href":"http://localhost:3000/baskets/1",
"rel":"self","type":"application/json"}]}
```

3.8 - Accepting payments

Again, payments will be supported through its create and show methods for now. The show method is exactly as we expect from previous experiences:

```
class PaymentsController < ApplicationController

  acts_as_restfulie
```

```
  respond_to :xml, :json

  def show
    @payment = Payment.find(params[:id])
    respond_with @payment
  end

end
```

We are left with the *create* method, that instantiates payment based on its basket. Because the URI identified the basket id - a Rails feature that came out of the box - you can simply load the basket and create a payment with its relation already set:

```
@payment = Basket.find(params[:basket_id]).payments.create(params[:payment])
```

The rendering process is the same as for the item and basket creation:

```
def create
  @payment = Basket.find(params[:basket_id]).payments.create(params[:payment])
  render :text => "", :status => 201,
    :location => basket_payment_url(@payment.basket, @payment)
end
```

On the client side, we can create a payment object

```
# prepares the payment
payment = {"payment" => {
                                "cardnumber" => "4850000000000001",
                                "cardholder" => "guilherme silveira",
                                "amount" => resource.basket.price}}
```

And finally navigate through the payment link:

```
# creates the payment
resource = resource.basket.links.payment.post! payment
```

> **Client data serialization**
>
> When serializing an object into a representation to send back to the server, we have always used the default one. In order to customize the serialization process, pass on a recipe which is as exactly as a tokamak description is.
> You should avoid custom serialization whenever possible on the client side. Whenever the client needs to customize its parameters, it means there is some extra coupling between it and the current media type. Using as less coupling as possible will allow you to switch media types and keep everything running.

And that's all we have to do. What did we earn so far?

3.9 - Winnings so far

If the server only supports xml and starts supporting json, your client will keep working just fine.

If at any day, the server drops support from one of those media types, your client will keep working.

If the server decides to implement support to a new media type, with extra functionalities, you can update each part of your client step by step by introducing a call to *accepts* prior to invoking a link relation.

If any URI but the entry point changes, your client will still work just fine.

By using the proper verbs to do resource retrieval and creation, intermediates as gateways and caches might cache those resource for later use.

As we will see later, cache headers can be automatically inserted, cache channels can be created, the tokamak files might be shared between different implementations of Restfulie and much more.

If a group of friends wants to buy a product for someone, they may make each one part of the payment and when the payment is entirely fulfilled, the product is then sent. This is due to the use of the has_many relationship and http uniform interface for accessing our system.

If another system supports the same set of media types, you can change the entry point and achieve the same result in another system.

There are still a few things to improve and we shall continue doing so step by step to allow server evolution without breaking existing clients.

3.10 - Exercises

1) Create a delete method for an item, load its id and invoke the delete method.

2) Using the console, retrieve the entry point, select the last item, invoke the self relation by sending a delete request.

3) Check that the resource was deleted in the database.

```
> script/console
> puts Item.last.name
```

4) Customize the basket representation by adding a field that indicates whether the basket has not been, partial ou fully paid. A basket has not been paid if the sum of payment amount is 0. Its partially paid if it is less than the basket price but higher than zero. It is fully paid when that sum is bigger than the basket price.

5) Change the test script to send half of the basket cost in two different payments. Check the basket status between them and after the latest payment is done.

6) Add support to hotel removal in your hotel project.

7) Test the hotel removal.

8) Create a new model called Booking, with a many to many relationship to hotels.

9) Support Booking creation and show on the server side.

10) Write a test file that loads all hotels, and use the first two to create a Booking.

11) Support Booking removal and test it.

12) Support Basket removal and test it.

Examples

4.1 - Sample Projects

You can find full examples on how to use Restfulie both on the client and server side at:

http://github.com/caelum/restfulie-full-examples

4.2 - Support

You will find support in each different restfulie mailing lists:

Ruby and Rails: http://groups.google.com/group/restfulie Java: http://groups.google.com/group/restfulie-java C#: http://groups.google.com/group/restfulie-csharp Ruby and Rails developers: http://groups.google.com/group/restfulie-dev

4.3 - Training

Caelum Objects (http://caelumobjects.com) offers training on REST systems and processes, starting from the basic theory to the practical coding on how to save on middleware and use the web as your infrastructure.

CHAPTER 5

Restfulie Java

5.1 - Server side

On the server side, Restfulie is embedded within VRaptor. You can start a new VRaptor project by downloading a blank project at its website at http://vraptor.caelum.com.br/en/downloads.jsp.

You can find more info on vraptor at http://vraptor.caelum.com.br/documentation/

To enable restfulie features, there is only one configuration step, setting up the Restfulie components, which will substitute some dependency injection mechanisms internally to VRaptor and add support to serializing objects into hypermedia aware representaitons.

In order to do so, simply open your web.xml file and add the following lines of code:

```
<context-param>
    <param-name>br.com.caelum.vraptor.packages</param-name>
    <param-value>br.com.caelum.vraptor.restfulie</param-value>
</context-param>
```

Defining URIs and verbs

The basic usage of VRaptor allows you to set which URI pattern and set of HTTP verbs will be answered with a method. A typical ItemsController using VRaptor looks like:

```
@Resource
public class ItemsController {

    @Get @Path("/items")
    public void list() {...}

    @Post @Path("/items")
    public void create(Item item) {...}

    @Get @Path("/items/{id}")
    public void show(int id) {...}

    @Put @Path("/items/{item.id}")
    public void update(Item item) {...}

    @Delete @Path("/items/{id}")
    public void destroy(int id) {...}
}
```

Using resource representations

Since you can't transport your resources through the Web, you must transport its representations. These representations must use an specific media type that both ends understand, i.e. application/xml. While some media types are hypermedia capable, others aren't. It is a tradeoff that is out of the discussion of this course which one is better/worse in each case, but you can find more information about those tradeoffs in the FJ-91 training course.

On VRaptor, if you want to receive a representation from your resource you must add the `@Consumes` annotation:

```
@Post @Path("/items")
@Consumes
public void create(Item item) {...}
```

VRaptor will use the request Content-Type header to decide how to deserialize the request body into an Item. You must have a `Deserializer` for this content type, or VRaptor will fail with a 415 - Unsupported Media Type response.

VRaptor has built-in deserializers for application/xml and application/json formats. You can implement your own deserializers either by implementing Deserializer interface or extending an existing Deserializer. For example, if you want to support an xml-based vendor media type you can extend the VRaptor default xml deserializer:

```
@ApplicationScoped
@Deserializes("application/bookstore+xml")
public class BookstoreDeserializer extends XStreamXMLDeserializer {
   //...
   // you can override the protected methods for customizations
}
```

If you want to send representations you can use the `Result` object, as a typical VRaptor web application:

```
import static br.com.caelum.vraptor.view.Results.representation;
@Resource
public class ItemsController {

    public ItemsController(Database database, Result result) {
        this.database = database;
        this.result = result;
    }

    @Get @Path("/items/{id}")
    public void show(int id) {
        Item item = database.get(id);
        result.use(representation()).from(item).serialize();
    }
    //...
}
```

This way VRaptor will use content negotiation to decide which format will be used to serialize the `item`. VRaptor has built-in serializers for application/xml and application/json, but you can add more serializers by implementing Serialization interface or extending existing serializers. For xml-based types:

```
@Component
public class BookstoreSerialization extends XStreamXMLSerialization {
    //...
    @Override
    public boolean accepts(String format) {
        return "application/bookstore+xml".equals(format);
    }
    // override protected methods for customizations
}
```

Hypermedia

For enabling hypermedia links, your resources should implement the HypermediaResource interface and define transitions, so VRaptor will generate links for these transitions:

```
public class Item implements HypermediaResource {
    private Integer id;
    private String name;
    private Double price;

    @Override
    public void configureRelations(RelationBuilder builder) {
        //VRaptor will generate the URIs if you use the controller path
        builder.relation("self").uses(ItemsController.class).show(id);

        //but you can add the uris as string also
        builder.relation("basket").at("/baskets");
    }

}
```

When you serialize an `Item`, you get an xml as:

```
<item>
    <id>3</id>
    <name>Soap</name>
    <price>1.99</price>
    <atom:link rel="self" href="http://localhost:8080/items/3"
            xmlns:atom="http://www.w3.org/2005/Atom"/>
    <atom:link rel="basket" href="http://localhost:8080/baskets"
            xmlns:atom="http://www.w3.org/2005/Atom"/>
</item>
```

5.2 - Client-side

On the client-side, you can use Restfulie-java client project: http://restfulie.caelumobjects.com

Configuring your REST client

For accessing RESTful services, you must set up the Restfulie RestClient with your application types, and possible representations. The start point is the Restfulie.custom() factory method:

```
RestClient client = Restfulie.custom();
```

At this point the RestClient is ready to access any RESTful service, but can't yet deserialize any representation. For deserialization you must configure the MediaTypes and corresponding java classes. For example, a XML based media types would be:

```
RestClient client = Restfulie.custom();
client.getMediaTypes().register(new XmlMediaType() {
    @Override
    protected List<Class> getTypesToEnhance() {
        // all the types that will participate on your business proccess
        return Arrays.<Class>asList(Item.class, Basket.class, Payment.class);
    }

    @Override
    protected List<String> getCollectionNames() {
        // for collection types, you must return the plurals of your types here
        return Arrays.asList("items", "baskets", "payments");
    }
});
```

Now the client is able to access any service that uses items, baskets and payments and deserializes and serializes their representations.

Getting a resource

When getting a resource, you must specify which content-type will be used to deserialize it.

```
Response response = client
    .at("http://localhost:8080/restfulie/items/2")
    .accept("application/xml")
    .get();
Item item = response.getResource();
```

Sending a resource

When sending a resource to the service, you must also specify which content-type will be used to serialize the resource.

```
Item item = new Item("soap", 1.99);
Response response = client
    .at("http://localhost:8080/restfulie/items")
    .accept("application/xml")
    .as("application/xml")
    .post(item);

item = response.getResource();
```

Navigating through hypermedia

At the point that you received a representation of a resource from the RESTful service, you can use the hypermedia links to execute further actions. The returned resource has all the associated links ready to use and follow.

```
Response response = client.at("http://localhost:8080/restfulie/items").accept(XML).get();
List<Item> items = response.getResource();

//select some items
List<Item> selectedItems = items.subList(0, 2);
```

Now we are ready to follow the *basket* rel link by accessing it:

```
//the resource method allows you to access the resource links
Link link = resource(items).getLink("basket");

//you can now follow the link to create a basket with the selected items
response = link.follow().as(XML).accept(XML).post(new Basket(selectedItems));
```

The above code is missing the following static imports:

```
import static br.com.caelum.restfulie.Restfulie.resource;
import static br.com.caelum.restfulie.mediatype.MediaTypes.XML;
```

One can enhance our example even further by checking the response code and executing the payment. The rest of the example uses part of the API described so far:

```
//you can inspect the response status code to know if the creation was successful
if (response.getCode() == 200) {
    Basket basket = response.getResource();

    //having the basket, we can now procceed to payment
    response = resource(basket).getLink("payment")
        .follow().as(XML).accept(XML).post(new Payment(basket.getCost()));


    Payment payment = response.getResource();
    // and now see if the payment was accepted
    if (payment.getStatus() == Status.ACCEPTED) {
        System.out.println("My job is done!")
    } else {
        //do something else to be able to pay the basket, perhaps removing some items
    }
}
```

Get to know some of our trainings!

FJ-31:
Java EE and Web Services

FJ-91:
Architecture and Design with Java

RR-71:
Agile Web 2.0 Development with Ruby on Rails

FJ-16:
Test, XML and Design Patterns Workshop with Java

RR-75:
Advanced Ruby and Rails: dealing with every day problems

RR-79:
Rest from Scratch

For price inquiries, contact us at contato.sp@caelum.com.br

- ✓ More than 8000 students;
- ✓ Restfulie oficial training;
- ✓ Updated material;
- ✓ Unique teaching methodology;
- ✓ Instructors well known on the Java, Rails and Agile community;
- ✓ Experienced instructors;
- ✓ Free online material.