Tutorial

Table of contents

1. Registering an implementation

The Restlet framework is composed of two parts. First, there is the "Restlet API", a neutral API supporting the concepts of REST and a mechanism called Restlet facilitating the handling of REST uniform calls. This API must be supported by a "Restlet Implementation" before it can effectively be used. Multiple implementations could be provided (open source projects or commercial products).

This separation between the API and the implementation is similar to the one between the Servlet API and Web containers like Jetty or Tomcat, or between the JDBC API and actual JDBC drivers. Currently, the "Noelios Restlet Engine" (NRE) is available and acts as the reference implementation. When you download the Restlet framework, the API and the NRE come bundled together, ready to be used. If you need to use a different implementation just add the implementation JAR file to the classpath and remove the NRE JAR file named com.noelios.restlet.jar by default.

The registration is done automatically. See the JAR specification for details. When an implementation is loaded, it automatically calls back the `org.restlet.Factory.setInstance()` method.

2. Retrieving the content of a Web page

As we mentioned in the introduction paper, the Restlet framework is at the same time a client and a server framework. For example, NRE can easily work with remote resources using its HTTP client connector. A connector in REST is a software element that enables the communication between components, typically by implementing one side of a network protocol. Here we will get the representation of an existing resource and output it in the JVM console:

```
// Outputting the content of a Web page
Client client = new GenericClient(Protocol.HTTP);
client.get("http://www.restlet.org").getOutput().write(System.out);
```

Note that the example above uses a simplified way to issue calls via the HttpClient. A more flexible way is to create a new REST

call and to ask the HTTP client to handle it. The example below illustrate how to set some preferences in your call, like a referrer URI or the languages and media types you prefer to receive as a response:

```
// Prepare the REST call
Call call = new Call(Method.GET, "http://www.restlet.org");
call.setReferrerRef("http://www.mysite.org");

// Ask to the HTTP client connector to handle the call
Client client = new GenericClient(Protocol.HTTP);
client.handle(call);

// Output the result representation on the JVM console
Representation output = call.getOutput();
output.write(System.out);
```

3. Listening to Web browsers

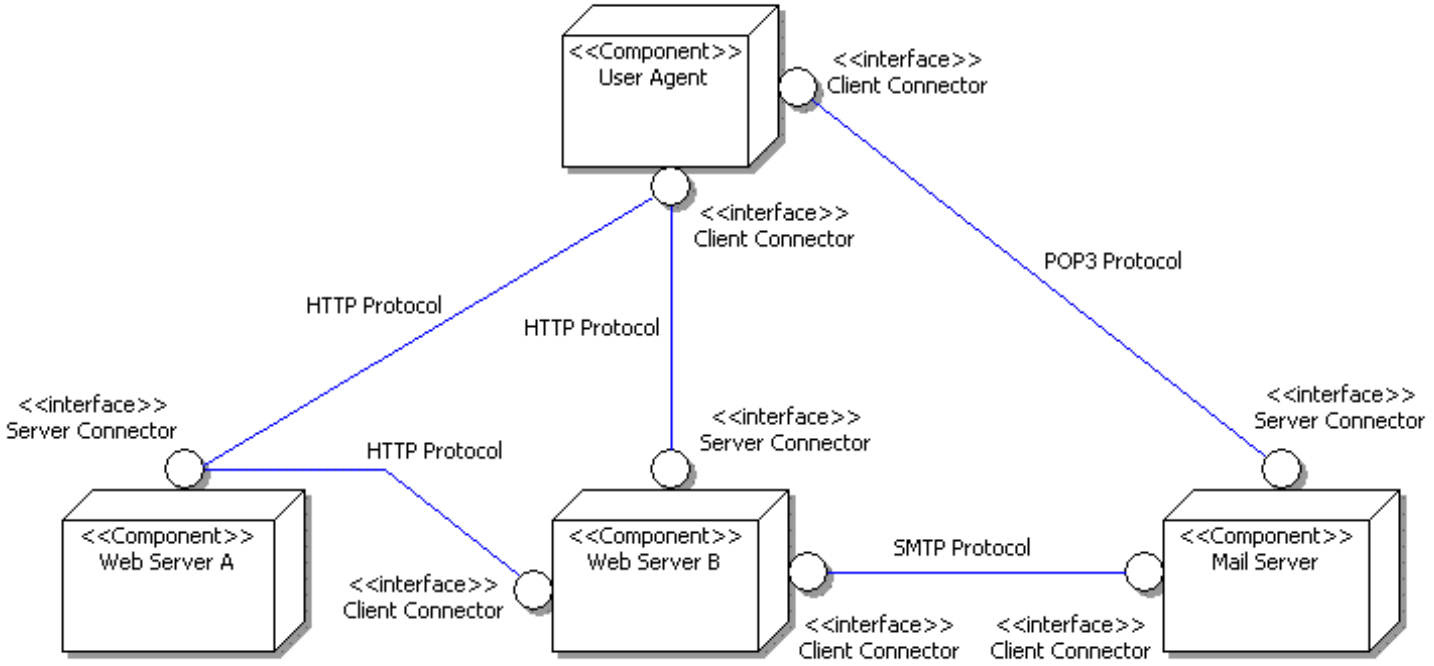
Now, we want to see how the Restlet framework can listen to client requests and reply to them. We will use the NRE HTTP server connector (supported by Mortbay's Jetty HTTP listener) and return a simple string representation "Hello World!" as plain text. Note that in a more realistic application, we would probably create a separate class extending the AbstractRestlet instead of relying on an anonymous inner class.

```
// Creating a minimal Restlet returning "Hello World"
Restlet handler = new Restlet()
{
   public void handleGet(Call call)
   {
      call.setOutput(new StringRepresentation("Hello World!"));
   }
};

// Create the HTTP server and listen on port 8182
new GenericServer(Protocol.HTTP, 8182, handler).start();
```

If you run this code and launch your server, you can open a Web browser and hit the http://localhost:8182. Actually, any URI will work, try also http://localhost:8182/test/tutorial. Note that if you test your server from a different machine, you need to replace "localhost" by either the IP address of your server or its domain name. The handler that we created is very primitive. We will see later how to take more advantage of the Restlet framework.

4. Overview of a REST architecture

Let's step back a little bit and consider typical web architectures from a REST point of view. In the diagram below, circles represent the connector that enables the communication between components which are represented by the boxes. The blue links represents the network communication using a particular protocol (HTTP, SMTP, etc.).



Note that the same component can have any number of client and server connectors attached to it. Web Server B, for example, has both a server connector to respond to requests from the User Agent component, and client connectors to send requests to the Database Server, the Mail Server and the Web Server A.

5. Restlets servers and containers

In addition to supporting the standard REST software architecture elements as presented before, the Restlet framework also provides a set of components that facilitate the handling of REST calls. The goal is to provide a powerful and RESTful alternative to existing Servlet frameworks.

First, we can use Restlets which are simple call handlers living inside a parent Restlet container. Restlets are selected when the URI of the requested resource matches a given URI pattern. In order to see how to reply to client requests, we will attach a Restlet (myRestlet) to a parent container (myContainer). In order for a Web client to invoke the Restlet, an URI like this one will have to be entered: http://localhost:8182/trace/abc/def?param=123.

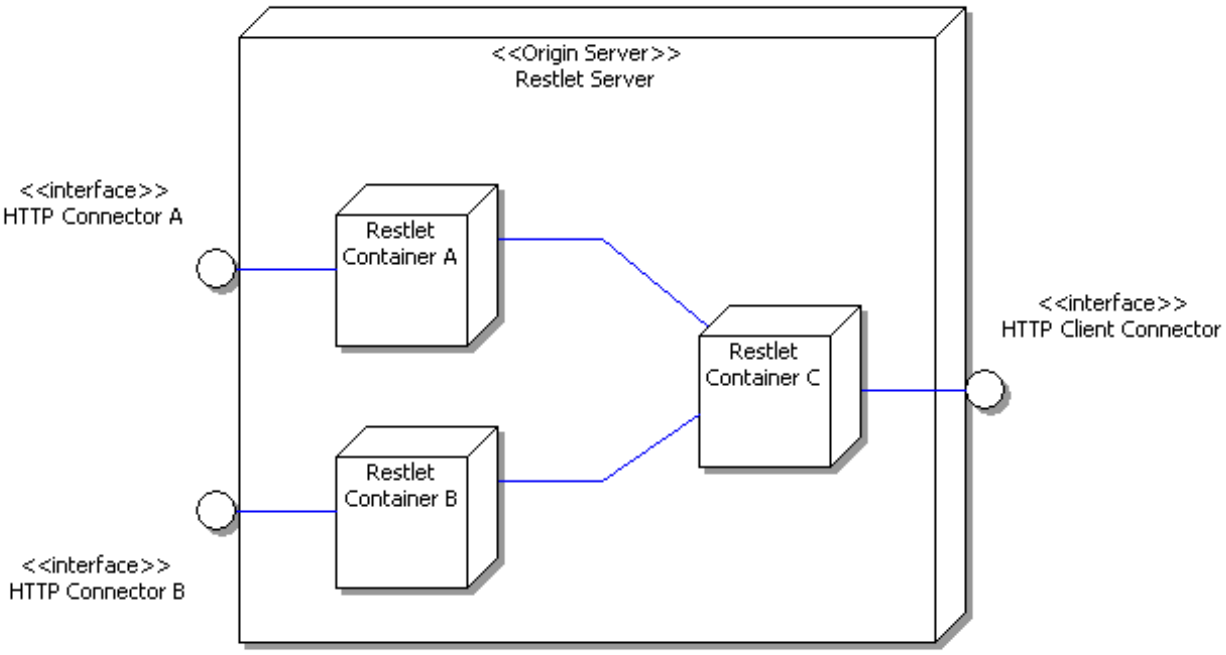
```
// Create a new Restlet container
Container myContainer = new Container();
Context myContext = myContainer.getContext();

// Create the HTTP server connector, then add it to the container.
// Note that the container will act as the initial Restlet call's handler.
myContainer.getServers().add(Protocol.HTTP, 8182);

// Create a host router matching calls to the server
HostRouter host = new HostRouter(myContext, 8182);
myContainer.setRoot(host);

// Create a new Restlet that will display some path information.
Restlet myRestlet = new Restlet(myContext)
   {
      public void handleGet(Call call)
      {
         // Print the requested URI path
```

```
                String output = "Resource URI:  " + call.getResourceRef() + '\n' +
                                "Base URI:      " + call.getBaseRef() + '\n' +
                                "Relative path: " + call.getRelativePart() + '\n' +
                                "Query string:  " + call.getResourceRef().getQuery
();
                call.setOutput(new StringRepresentation(output));
            }
        };

        // Then attach it to the host router.
        host.getScorers().add("/trace", myRestlet);

        // Now, let's start the container!
        // Note that the HTTP server connector is also automatically started.
        myContainer.start();
```

Here is the result that you will get if you test the example with the proposed URI:

```
Resource URI:  http://localhost:8182/trace/abc/def?param=123
Base URI:      http://localhost:8182/trace
Relative path: /abc/def
Query string:  param=123
```

Now, if your web application is becoming more complex, you may want to use multiple Restlet containers, all living within a parent Restlet server. For example, this is very useful if you want to separate you domain resources (managed by container C) from your web server resources (managed by container A) and your web services resources (managed by container B):



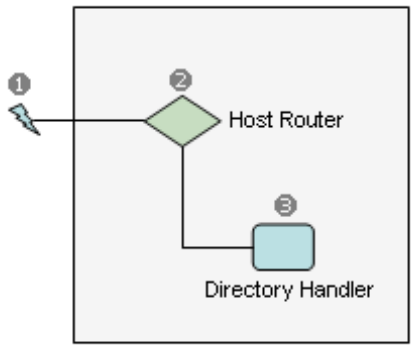
Also, note that servers and containers are REST components themselves. This means that if your Restlet server has only one container, you can simply use it as your origin server and attach connectors normally to it.

6. Serving context resources

Do you have a part of your web application that serves static pages like Javadocs? Well, no need to setup an Apache server just

for that, the Noelios Restlet Engine provides a dedicated Restlet called DirectoryRestlet. See how simple it is to use it:

```
        // Create a host router matching calls to the server
        HostRouter host = new HostRouter(myContainer.getContext(), 8182);
        myContainer.setRoot(host);

        // Create a directory Restlet able to return a deep hierarchy of Web files
        // (HTML pages, CSS stylesheets or GIF images) from a local directory.
        DirectoryHandler directory = new DirectoryHandler(myContainer.getContext(),
ROOT_URI, "index.html");

        // Then attach the Restlet to the container.
        host.getScorers().add("/", directory);
```



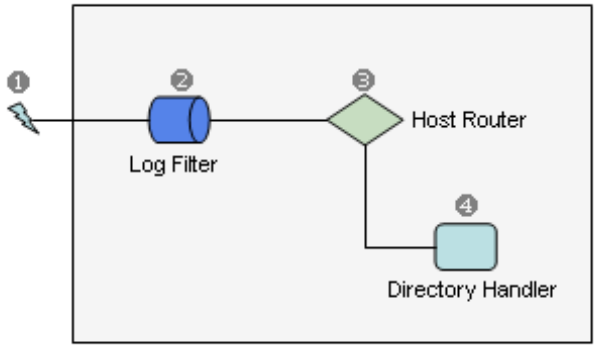
Note that no external configuration file is needed. Here, the three extensions are declaring some media types, but the same mechanism can also be used to declare other metadata like languages.

7. Filters and call logging

Being able to properly log the activity of a Web application is a common requirement. NRE has a class called the LogFilter that knows how to generate Apache-like logs. By taking advantage of the logging facility built in the JDK, the logger can be configured like any standard JDK log to filter messages, reformat them and specify where to send them. Rotation of logs is also supported; see the java.util.logging package for details.

Filters are specialized Restlets that filter calls passed to them, without modifying the requested resource URI. If you are familiar with the Servlet API, the concept is similar to the Filter interface. Below is a simple example adding the logging facility to the previous example:

```
        // Attach a log Filter to the container
        LogFilter log = new LogFilter(myContext, "com.noelios.restlet.example");
        myContainer.setRoot(log);

        // Create a host router matching calls to the server
        HostRouter host = new HostRouter(myContext, 8182);
        log.setNext(host);

        // Create a directory Restlet able to return a deep hierarchy of Web files
        DirectoryHandler directory = new DirectoryHandler(myContext, ROOT_URI,
"index.html");

        // Then attach the Restlet to the log Filter.
        host.getScorers().add("/", directory);
```



Note that "com.noelios.restlet.tutorial" is the log name as given to the java.util.logging classes. You will need to set the configuration location, for example by calling:

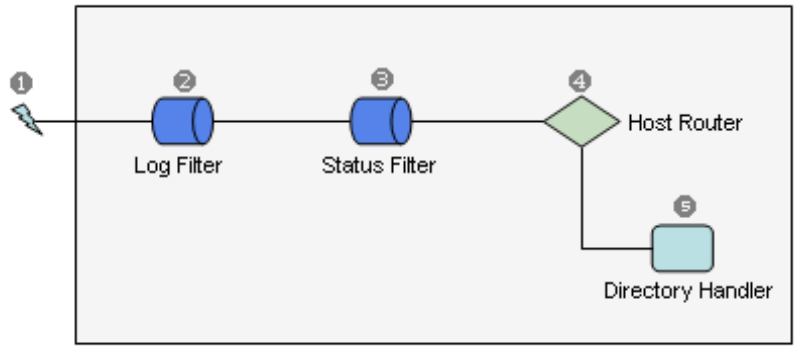
```
        System.setProperty("java.util.logging.config.file", "/your/path/logging.
config");
```

For details on the configuration file format, please check the JDK's LogManager class.

8. Displaying error pages

Another common requirement is the ability to customize the status pages returned when something didn't go as expected during the call handling. Maybe a resource was not found or an acceptable representation isn't available? In this case, or when any unhandled exception must be intercepted, the StatusFilter can be used. Here we will simply attach one between the logger Filter and the directory Restlet. Try to get a non-existing resource to see a status message:

```
        // Attach a log Filter to the container
        LogFilter log = new LogFilter(myContext, "com.noelios.restlet.example");
        myContainer.setRoot(log);

        // Attach a status Filter to the log Filter
        StatusFilter status = new StatusFilter(myContext, true, "webmaster@mysite.
org", "http://www.mysite.org");
        log.setNext(status);

        // Create a host router matching calls to the server
        HostRouter host = new HostRouter(myContext, 8182);
        status.setNext(host);

        // Create a directory Restlet able to return a deep hierarchy of Web files
        DirectoryHandler directory = new DirectoryHandler(myContext, ROOT_URI,
"index.html");

        // Then attach the directory Restlet to the host router.
        host.getScorers().add("/", directory);
```



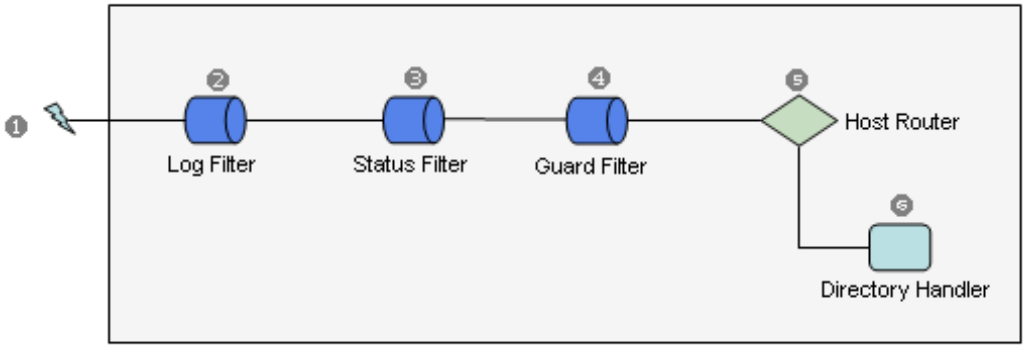
In order to customize the default messages, you will simply need to subclass the StatusFilter class and override the getRepresentation(status, call) method.

9. Guarding access to sensitive resources

When you need to secure the access to some Restlets, several options are available. The most common way is to rely on cookies to identify clients (or client sessions) and to check a given user ID or session ID against your application state to determine if access should be granted. Restlets natively support cookies via the Cookie and CookieSetting objects accessible from a Call.

Also, there is another way based on the standard HTTP authentication mechanism. The Restlet framework currently supports the basic HTTP authentication scheme via the GuardFilter. See below how we would modify the previous example to secure the access to the directory restlet:

```
        // Attach a guard Filter to the container
        GuardFilter guard = new GuardFilter(myContext, "com.noelios.restlet.
example", true, ChallengeScheme.HTTP_BASIC , "Restlet tutorial", true);
        guard.getAuthorizations().put("scott", "tiger");
        status.setNext(guard);
```



Note that the authorization decision is fully customizable via the authorize method. Any mechanism can be used to check whether the given credentials are valid. Here we simply hard-coded the only valid user and password. In order to test this authentication mechanism, let's use the client-side Restlet API:

```
        // Prepare the REST call
        Call call = new Call(Method.GET, "http://localhost:8182/");

        // Add the client authentication to the call
        ChallengeResponse authentication = new ChallengeResponse(ChallengeScheme.
HTTP_BASIC, "scott", "tiger");
        call.getSecurity().setChallengeResponse(authentication);
```

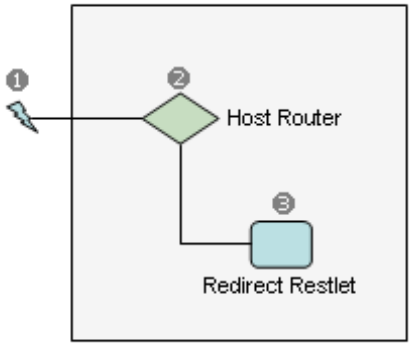
```
        // Ask to the HTTP client connector to handle the call
        Client client = new GenericClient(Protocol.HTTP);
        client.handle(call);

        if(call.getStatus().isSuccess())
        {
           // Output the result representation on the JVM console
           Representation output = call.getOutput();
           output.write(System.out);
        }
        else if(call.getStatus().equals(Status.CLIENT_ERROR_UNAUTHORIZED))
        {
           // Unauthorized access
           System.out.println("Your access was not authorized by the server, check
your credentials");
        }
        else
        {
           // Unexpected status
           System.out.println("An unexpected status was returned: " + call.getStatus
());
        }
```

You can change the user ID or password sent by this test client in order to check the response returned by the server. Remember to launch the previous Restlet server before starting your client. Note that if you test your server from a different machine, you need to replace "localhost" by either the IP address of your server or its domain name when typing the URI in the browser. The server won't need any adjustment due to the usage of the HostRouter which accepts all types of URI by default.

10. URI rewriting and redirection

One of the advantages of the Restlet framework is the built-in support for cool URIs. Another good description of the importance of proper URI design is given by Jacob Nielsen in his AlertBox. The first tool available is the RedirectRestlet, which allows the rewriting of a cool URI to another URI, followed by an automatic redirection. Several types of redirection are supported, the external redirection via the client/browser, the internal redirection at the container level and the connector redirection for proxy-like behavior. In the example below, we will define a search service for our web site (named "mysite.org") based on Google. The "/search" relative URI identifies the search service, accepting some keywords via the "query" parameter. :

```
        // Create a redirect Restlet then attach it to the container
        String target = "http://www.google.com/search?q=site:mysite.org+${query
('query')}";
        RedirectRestlet redirect = new RedirectRestlet(myContext, target,
RedirectRestlet.MODE_CLIENT_TEMPORARY);
        host.getScorers().add("/search", redirect);
```



Note that the RedirectRestlet needs two parameters only. The first one defines how the URI rewriting should be done, based on a URI template. This template will be processed by the StringTemplate class and the URI data model provided by the RestletCallModel class. The second parameter defines the type of redirection; here we chose the client redirection, for simplicity purpose.

11. Routers and hierarchical URIs

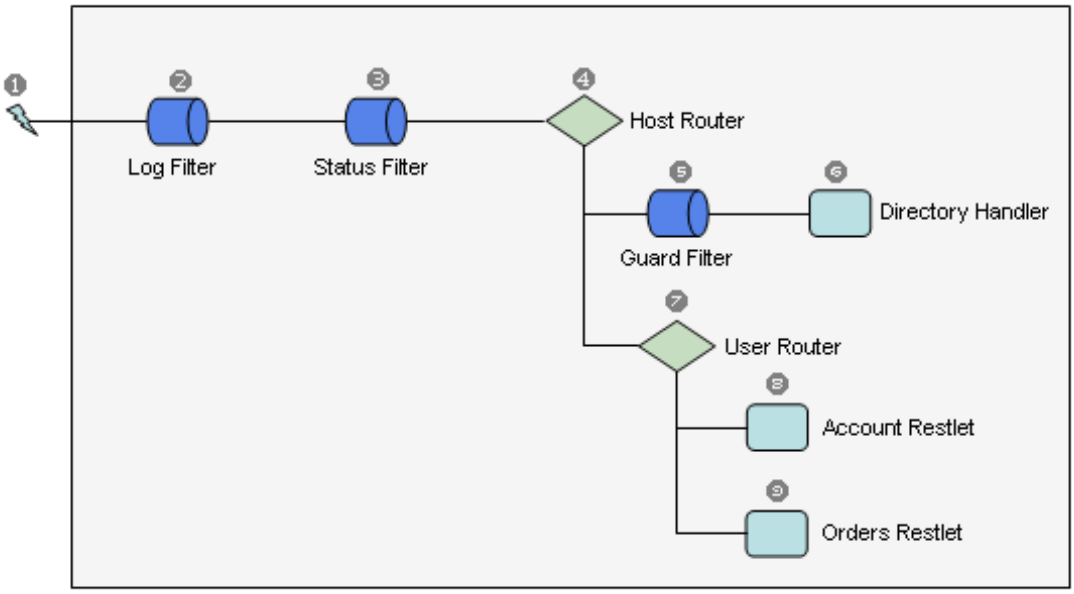
In addition to the RedirectRestlet, we have another tool to manage cool URIs: Routers. They are specialized Restlets that can have other handlers (Restlets, Routers or Filters) attached to them and that can automatically delegate calls based on a URI pattern (a regular expression). Note that Restlet containers are also Routers, so you could directly attach Restlets to "myContainer" using an adequate URI pattern. In general, it is easier to attach Filters followed by a HostRouter to the container. Then you can attach your application Restlets or Routers to this root HostRouter.

Here we want to explain how to handle the following URI patterns:

1. /docs/ to display static files
2. /users/[a-z]+ to display a user account
3. /users/[a-z]+/orders to display the orders of a particular user

The fact that these URIs contain some dynamic parts and that no file extension is used makes it harder to handle them in a typical Web container. Here we will simply rely on a tree of Routers. In the processing flow below, you can see that when a call is received by the container (1), it is first filtered by two Filters (2) and (3), and then it reaches the root Host Router. If the URI starts with "/docs/" then the GuardFilter will get the call (5) and pass it to the DirectoryRestlet (6) if the user credentials are authorized. Instead, if the URI starts with "/users/[a-z]+" (where "[a-z]+" is representing a valid user name) then it is the User Router that gets the call, otherwise an error is returned to the client (404, not found). Please, note that the User Router is directly attached to its parent Host Router and not to the top level container!

When a call reaches the User Router (7) we try to match the remaining part of the URI against the "$" pattern, meaning the the remaing resource path should be empty. If a match is found, then the Account Restlet is invoked (8). At this point it is possible to get the login name by getting the last URI segment. Otherwise, if the "/orders$" pattern matches, the Orders Restlet is invoked (9), where we can still get the login name from the list of URI segments. Note that again we use the '$' character at the end of the pattern in order to make sure that we match "/users/foo/orders" but not "/users/foo/orders123".



See the implementation code below. In a real application, you will probably want to create separate subclasses instead of the anonymous ones we use here:

```
        // Create a new Restlet container
        Container myContainer = new Container();
        Context myContext = myContainer.getContext();

        // Add an HTTP server connector to the Restlet container.
        // Note that the container is the call restlet.
        myContainer.getServers().add(Protocol.HTTP, 8182);

        // Attach a log Filter to the container
        LogFilter log = new LogFilter(myContext, "com.noelios.restlet.example");
        myContainer.setRoot(log);

        // Attach a status Filter to the log Filter
        StatusFilter status = new StatusFilter(myContext, true, "webmaster@mysite.
org", "http://www.mysite.org");
        log.setNext(status);

        // Create a host router matching calls to the server
        HostRouter host = new HostRouter(myContext, 8182);
        status.setNext(host);

        // Attach a guard Filter to secure access the the chained directory Restlet
        GuardFilter guard = new GuardFilter(myContext, "com.noelios.restlet.
example", true, ChallengeScheme.HTTP_BASIC , "Restlet tutorial", true);
        guard.getAuthorizations().put("scott", "tiger");
        host.getScorers().add("/docs/", guard);

        // Create a directory Restlet able to return a deep hierarchy of Web files
        DirectoryHandler directory = new DirectoryHandler(myContext, ROOT_URI,
"index.html");
        guard.setNext(directory);

        // Create the user router
        Router user = new Router(myContext);
```

```
        host.getScorers().add("/users/[a-z]+", user);

        // Create the account Restlet
        Restlet account = new Restlet()
           {
              public void handleGet(Call call)
              {
                 // Print the requested URI path
                 String output = "Account of user named: " + call.getBaseRef().
getLastSegment();
                 call.setOutput(new StringRepresentation(output, MediaType.
TEXT_PLAIN));
              }
           };
        user.getScorers().add("$", account);

        // Create the orders Restlet
        Restlet orders = new Restlet(myContext)
           {
              public void handleGet(Call call)
              {
                 // Print the user name of the requested orders
                 List segments = call.getBaseRef().getSegments();
                 String output = "Orders of user named: " + segments.get(segments.
size() - 2);
                 call.setOutput(new StringRepresentation(output, MediaType.
TEXT_PLAIN));
              }
           };
        user.getScorers().add("/orders$", orders);

        // Now, let's start the container!
        myContainer.start();
```

12. Simplify configuration with fluent builders

At this point, you may feel like the configuration of a large application may not be as simple as it should be. It's especially difficult to keep track of the flow of Filters and Routers when looking the the code of Tutorial 11. One solution would be to rely on a compact XML configuration file, similar to the J2EE application descriptors. The drawbacks is that you loose the benefits of Java's strong typing, you need to learn another XML language and the result application is harder to debug and to manage over time.

This is were came the idea of Yuri de Wit and Lars Heuer to use the Fluent Interface design pattern. This pattern, formalized by Martin Fowler, aims to fluidify the combination of method calls. Here we use the pattern to progressively create new components, connectors, Restlets and to attach them to each other. The result is surprisingly compact and readable, especially if you organize your method calls into a tree. Here is a reimplementation of Tutorial 11 based on fluent builders:

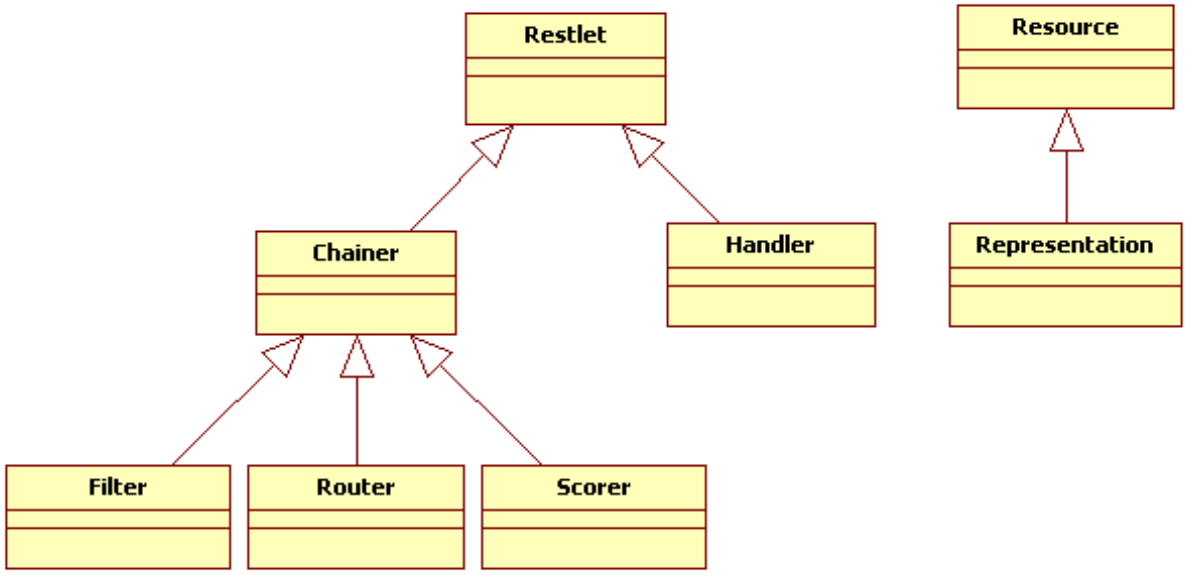
```
        // Build and start the container
        Builders.buildContainer()
           .addServer(Protocol.HTTP, 8182)
           .attachLog("com.noelios.restlet.example")
              .attachStatus(true, "webmaster@mysite.org", "http://www.mysite.org")
                 .attachHost(8182)
                    .attachGuard("/docs/", "com.noelios.restlet.example", true,
ChallengeScheme.HTTP_BASIC , "Restlet tutorial", true)
                       .authorize("scott", "tiger")
```

```
            .attachDirectory(ROOT_URI, "index.html").upRouter()
        .attachRouter("/users/[a-z]+")
              .attach("$", userRestlet).upRouter()
              .attach("/orders$", ordersRestlet).owner().start();
```

Here we are essentially creating a tree of builders. Each builder wrap a node (Restlet, Components, etc.) and provides method facilitating its configuration (setting of properties, creating of children and automatic attachment). Each time an attach*() method is invoked, the current builder is moved down to a builder wrapping the newly attached node. If you need to go up the tree of builders in order to attach more children, just use the up() or up(levels) or owner() or root() methods available. The to*() methods can be necessary in order to use methods specific to a certain type of builder. Another benefit is the possibility to use code completion in your IDE to assist you in using these builders. Have a look at the com.noelios.restlet.build package for more details.

Conclusion

We already covered many aspects of the framework. Before you move on by yourself, let's take a step back and look at this hierarchy diagram showing the main concepts covered in this tutorial and their relationships.



Beside this tutorial, your best source of information will be the Javadocs available for both the API and the NRE. You can also post your questions and help others in our discussion list.

Jérôme Louvel (blog)

Notes

- I encourage you to run the examples. The full source code is available in the latest release.
- Thanks to Jean-Paul Figer, Christian Jensen, Jim Ancona, Roger Menday, John D. Mitchell, Jérôme Bernard and Dave Pawson for the feed-back on this tutorial.

Version 4.7, last modified on 2006-08-18