



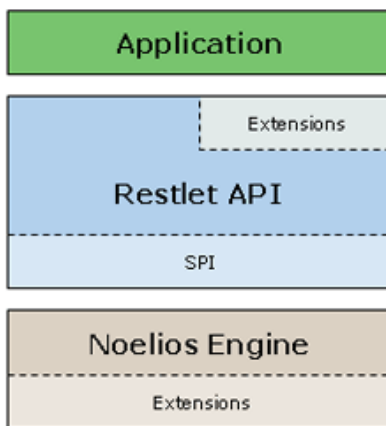
Tutorial

Table of contents

1. [Registering an implementation](#)
2. [Retrieving the content of a Web page](#)
3. [Listening to Web browsers](#)
4. [Overview of a REST architecture](#)
5. [Components, virtual hosts and applications](#)
6. [Serving static files](#)
7. [Access logging](#)
8. [Displaying error pages](#)
9. [Guarding access to sensitive resources](#)
10. [URI rewriting and redirection](#)
11. [Routers and hierarchical URIs](#)
12. [Reaching target Resources](#)
13. [Conclusion](#)

1. Registering an implementation

The Restlet framework is composed of two main parts. First, there is the "[Restlet API](#)", a neutral API supporting the concepts of REST and facilitating the handling of calls for both client-side and server-side applications. 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 concrete JDBC drivers. Currently, the "[Noelios Restlet Engine](#)" (NRE) is available and acts as the reference implementation. When you download the Restlet distribution, 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`.

The registration is done automatically. See the [JAR specification](#) for details. When an implementation is loaded, it automatically calls back the `org.restlet.util.Engine.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 Client(Protocol.HTTP);
client.get("http://www.restlet.org").getEntity().write(System.out);
```

Note that the example above uses a simplified way to issue calls via the generic Client class. A more flexible way is to create a new Request object and to ask the Client to handle it. The example below illustrates how to set some preferences in your call, like a referrer URI. It could also be the languages and media types you prefer to receive as a response:

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

// Handle it using an HTTP client connector
Client client = new Client(Protocol.HTTP);
Response response = client.handle(request);

// Write the response entity on the console
Representation output = response.getEntity();
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 one of the NRE HTTP server connectors (such as the one based on Mortbay's Jetty) 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 Restlet instead of relying on an anonymous inner class.

The Restlet class is similar to a Servlet and has provides somewhat limited help when handling calls in a RESTful applications. We will see later one that the framework provides many specialized subclasses as well as a Resource class that will be able to greatly abstract and simplify the handling process. For now, let's stick to a simple example:

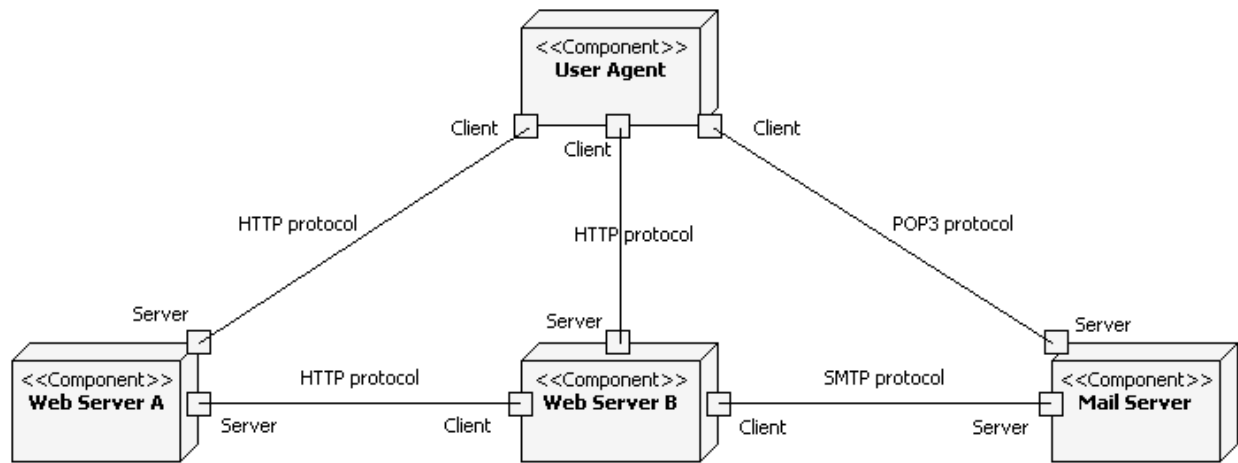
```
// Creating a minimal Restlet returning "Hello World"
Restlet restlet = new Restlet() {
    @Override
    public void handle(Request request, Response response) {
        response.setEntity("Hello World!", MediaType.TEXT_PLAIN);
    }
};

// Create the HTTP server and listen on port 8182
new Server(Protocol.HTTP, 8182, restlet).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 if it has one defined.

4. Overview of a REST architecture

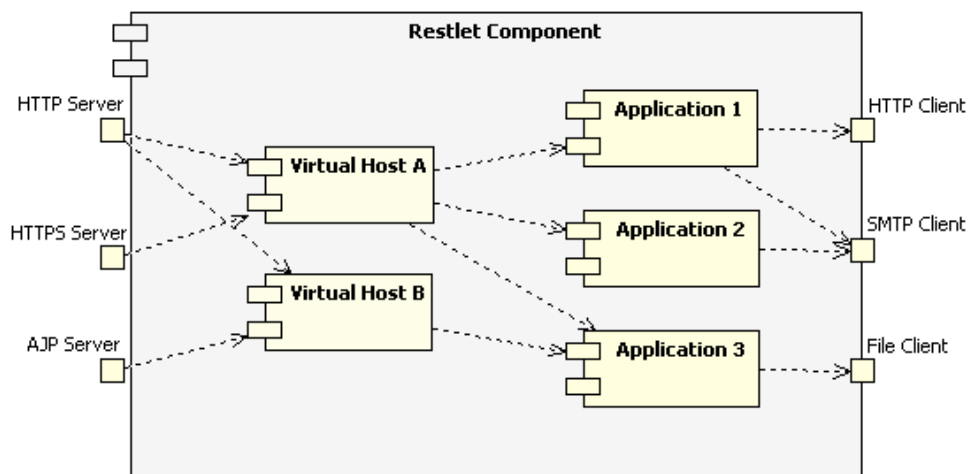
Let's step back a little and consider typical web architectures from a REST point of view. In the diagram below, ports represent the connector that enables the communication between components which are represented by the larger boxes. The links represents the particular protocol (HTTP, SMTP, etc.) used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



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. Components, virtual hosts and applications

In addition to supporting the standard REST software architecture elements as presented before, the Restlet framework also provides a set of classes that greatly simplify the hosting of multiple applications within a single JVM. The goal is to provide a RESTful, portable and more flexible alternative to the existing Servlet API. In the diagram below, we can see the three types of Restlets that are provided in order to manage these complex cases. Components can manage several Virtual Hosts and Applications. Virtual Hosts support flexible configuration where, for example, the same IP address is shared by several domain names, or where the same domain name is load-balanced across several IP addresses. Finally, we use Applications to manage a set of related Restlets, Resources and Representations. In addition, Applications are ensured to be portable and reconfigurable over different Restlet implementations and with different virtual hosts. In addition, they provide important services like access logging, automatic decoding of request entities, configurable status page setting and more!



In order to illustrate these classes, let's examine a simple example. Here we create a Component, then add an HTTP server connector to it, listening on port 8182. Then we create a simple trace Restlet and attach it to the default VirtualHost of the Component. This default host is catching any request that wasn't already routed to a declared VirtualHost (see the Component.hosts property for details). In a later example, we will also introduce the usage of the Application class. Note that for now you don't see any access log displayed in the console.

```
// Create a new Restlet component and add a HTTP server connector to it
Component component = new Component();
component.getServers().add(Protocol.HTTP, 8182);

// Create a new tracing Restlet
Restlet restlet = new Restlet() {
    @Override
```

```

    public void handle(Request request, Response response) {
        // Print the requested URI path
        String message = "Resource URI   : " + request.getResourceRef()
            + '\n' + "Root URI       : " + request.getRootRef()
            + '\n' + "Routed part  : "
            + request.getResourceRef().getBaseRef() + '\n'
            + "Remaining part: "
            + request.getResourceRef().getRemainingPart();
        response.setEntity(message, MediaType.TEXT_PLAIN);
    }
};

// Then attach it to the local host
component.getDefaultHost().attach("/trace", restlet);

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

```

Now let's test it by entering <http://localhost:8182/trace/abc/def?param=123> in a Web browser. Here is the result that you will get:

```

Resource URI   : http://localhost:8182/trace/abc/def?param=123
Root URI       : http://localhost:8182/trace
Routed part    : http://localhost:8182/trace
Remaining part : /abc/def?param=123

```

6. Serving static files

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, use instead the dedicated Directory class. See how simple it is to use it:

```

// Create a component
Component component = new Component();
component.getServer().add(Protocol.HTTP, 8182);
component.getClient().add(Protocol.FILE);

// Create an application
Application application = new Application(component.getContext()) {
    @Override
    public Restlet createRoot() {
        return new Directory(getContext(), ROOT_URI);
    }
};

// Attach the application to the component and start it
component.getDefaultHost().attach("", application);
component.start();

```

In order to run this example, you need to specify a valid value for `ROOT_URI`, depending on the location of your Restlet installation. By default, it is set to `file:///D:/Restlet/www/docs/api/`. Note that no additional configuration is needed. If you want to customize the mapping between file extensions and metadata (media type, language or encoding) or if you want to specify a different index name, you can use the Application's ["metadataService"](#) property.

7. Access logging

Being able to properly log the activity of a Web application is a common requirement. Restlet Components know by default how to generate Apache-like logs or even custom ones. 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 or specify where to send them. Rotation of logs is also supported; see the [java.util.logging](#) package for details.

Note that you can customize the logger name given to the `java.util.logging` framework by modifying the Application's `"logService"` property. In order to fully configure the logging, you need to declare a configuration file by setting a system property like:

```
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 is intercepted, the Application or the Component will automatically provide a default status page for you. This service is associated to the `org.restlet.util.StatusService` class, which is accessible as an Application and Component property called "statusService".

In order to customize the default messages, you will simply need to create a subclass of `StatusService` and override the `getRepresentation(Status, Request, Response)` method. Then just set an instance of your custom service to the appropriate "statusService" property.

9. Guarding access to sensitive resources

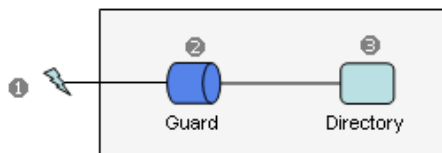
When you need to secure the access to some Restlets, several options are available. A common way is to rely on cookies to identify clients (or client sessions) and to check a given user ID or session ID against your application state to determine if access should be granted. Restlets natively support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

There is another way based on the standard HTTP authentication mechanism. The Noelios Restlet Engine currently accepts credentials sent and received in the Basic HTTP scheme and also the credentials sent in the Amazon Web Services scheme.

When receiving a call, developers can use the parsed credentials available in `Request.challengeResponse.identifier/secret` via the Guard filter. Filters are specialized Restlets that can pre-process a call before invoking and attached Restlet or post-process a call after the attached Restlet returns it. If you are familiar with the Servlet API, the concept is similar to the [Filter](#) interface. See below how we would modify the previous example to secure the access to the Directory:

```
// Create a Guard
Guard guard = new Guard(getContext(),
    ChallengeScheme.HTTP_BASIC, "Tutorial");
guard.getSecrets().put("scott", "tiger");

// Create a Directory able to return a deep hierarchy of files
Directory directory = new Directory(getContext(), ROOT_URI);
guard.setNext(directory);
return guard;
```



Note that the authentication and authorization decisions are fully customizable via the `authenticate()` and `authorize()` methods. Any custom mechanism can be used to check whether the given credentials are valid and whether the authenticated user is authorized to continue to the attached Restlet. Here we simply hard-coded a single user and password couple. In order to test, let's use the client-side Restlet API:

```
// Prepare the request
Request request = new Request(Method.GET, "http://localhost:8182/");

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

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

if (response.getStatus().isSuccess()) {
    // Output the response entity on the JVM console
```

```

        response.getEntity().write(System.out);
    } else if (response.getStatus()
        .equals(Status.CLIENT_ERROR_UNAUTHORIZED)) {
        // Unauthorized access
        System.out
            .println("Access authorized by the server, check your credentials");
    } else {
        // Unexpected status
        System.out.println("An unexpected status was returned: "
            + response.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 a VirtualHost which accepts all types of URIs by default.

10. URI rewriting and redirection

Another advantage of the Restlet framework is the built-in support for [cool URIs](#). A good description of the importance of proper URI design is given by Jacob Nielsen in his [AlertBox](#).

The first tool available is the Redirector, 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 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 "kwd" parameter:

```

// Create an application
Application application = new Application(component.getContext()) {
    @Override
    public Restlet createRoot() {
        // Create a Redirector to Google search service
        String target = "http://www.google.com/search?q=site:mysite.org+{keywords}";
        return new Redirector(getContext(), target,
            Redirector.MODE_CLIENT_TEMPORARY);
    }
};

// Attach the application to the component's default host
Route route = component.getDefaultHost().attach("/search", application);

// While routing requests to the application, extract a query parameter
// For instance :
// http://localhost:8182/search?kwd=myKeyword1+myKeyword2
// will be routed to
// http://www.google.com/search?q=site:mysite.org+myKeyword1%20myKeyword2
route.extractQuery("keywords", "kwd", true);

```

Note that the Redirector needs three parameters only. The first is the parent context, the second one defines how the URI rewriting should be done, based on a URI template. This template will be processed by the [Template](#) class. The second parameter defines the type of redirection; here we chose the client redirection, for simplicity purpose.

Also, we are relying on the Route class to extract the query parameter "kwd" from the initial request while the call is routed to the application. If the parameter is found, it is copied into the request attribute named "keywords", ready to be used by the Redirector when formatting its target URIs.

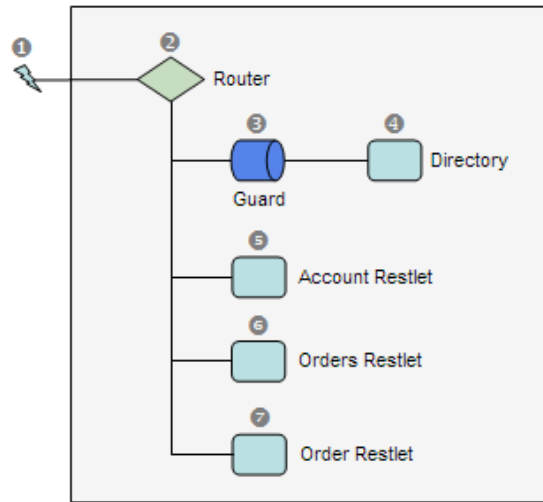
11. Routers and hierarchical URIs

In addition to the Redirector, we have another tool to manage cool URIs: Routers. They are specialized Restlets that can have other Restlets (Finders and Filters for example) attached to them and that can automatically delegate calls based on a [URI template](#). In general, you will set a Router as the root of your Application.

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

1. /docs/ to display static files
2. /users/{user} to display a user account
3. /users/{user}/orders to display the orders of a particular user
4. /users/{user}/orders/{order} to display a specific order

The fact that these URIs contain variable parts (between accolades) and that no file extension is used makes it harder to handle them in a typical Web container. Here, you just need to attach target Restlets to a Router using the URI template. At runtime, the route that best matches the request URI will receive the call and be able to invoke its attached Restlet. At the same time, the request's attributes map will be automatically updated with the value of the URI template variables!



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 component
Component component = new Component();
component.getServers().add(Protocol.HTTP, 8182);
component.getClients().add(Protocol.FILE);

// Create an application
Application application = new Application(component.getContext()) {
    @Override
    public Restlet createRoot() {
        // Create a root router
        Router router = new Router(getContext());

        // Attach a guard to secure access to the directory
        Guard guard = new Guard(getContext(),
            ChallengeScheme.HTTP_BASIC, "Restlet tutorial");
        guard.getSecrets().put("scott", "tiger");
        router.attach("/docs/", guard);

        // Create a directory able to expose a hierarchy of files
        Directory directory = new Directory(getContext(), ROOT_URI);
        guard.setNext(directory);

        // Create the account handler
        Restlet account = new Restlet() {
            @Override
            public void handle(Request request, Response response) {
                // Print the requested URI path
                String message = "Account of user \""
                    + request.getAttributes().get("user") + "\"";
                response.setEntity(message, MediaType.TEXT_PLAIN);
            }
        };
    }
};
```

```

// Create the orders handler
Restlet orders = new Restlet(getContext()) {
    @Override
    public void handle(Request request, Response response) {
        // Print the user name of the requested orders
        String message = "Orders of user \""
            + request.getAttributes().get("user") + "\"";
        response.setEntity(message, MediaType.TEXT_PLAIN);
    }
};

// Create the order handler
Restlet order = new Restlet(getContext()) {
    @Override
    public void handle(Request request, Response response) {
        // Print the user name of the requested orders
        String message = "Order \""
            + request.getAttributes().get("order")
            + "\" for user \""
            + request.getAttributes().get("user") + "\"";
        response.setEntity(message, MediaType.TEXT_PLAIN);
    }
};

// Attach the handlers to the root router
router.attach("/users/{user}", account);
router.attach("/users/{user}/orders", orders);
router.attach("/users/{user}/orders/{order}", order);

// Return the root router
return router;
}
};

// Attach the application to the component and start it
component.getDefaultHost().attach("", application);
component.start();

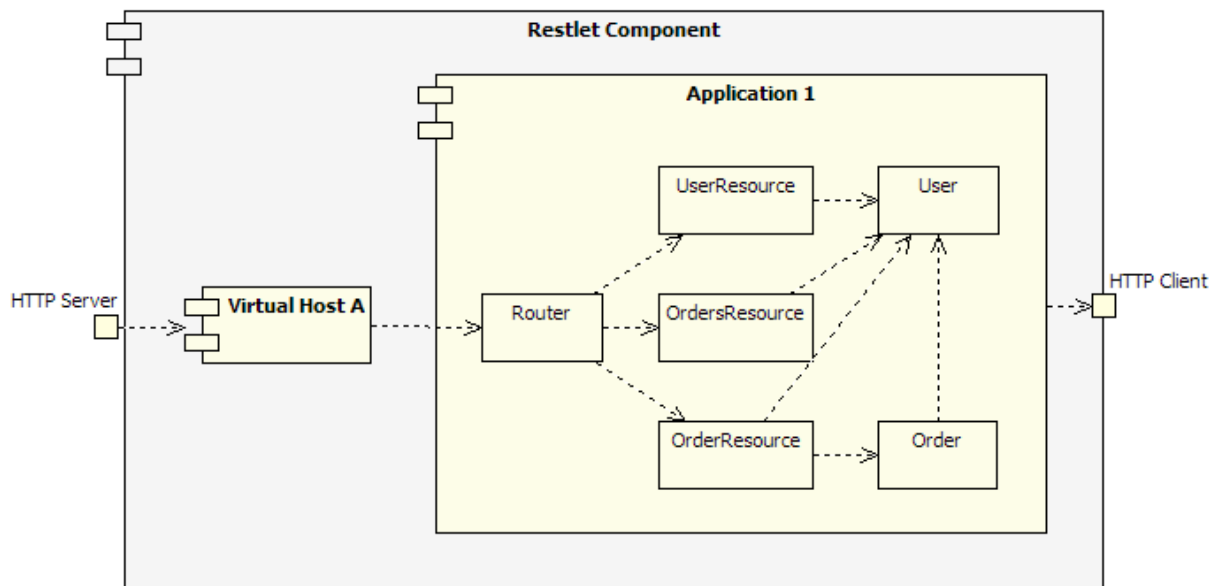
```

12. Reaching target Resources

In the previous example, we took advantage of the flexible routing features of the framework to route the requests while extracting interesting parts from the target URI. But, we didn't pay attention to the request method, nor to the client preferences regarding the response that he expects. Also, how do we connect our Restlet handlers with the backend systems, the domain objects?

So far, we introduced features that go beyond the traditional Servlet API but we didn't justify the REST part in our Restlet name! If you haven't done so already, I recommend that you learn more about the REST architecture style and the best practices to follow when applying it to a Web application. There is a related [FAQ entry](#) that will give you some starting pointers and we also maintain a [REST search engine](#) (based on Google) that can be useful. If you have some experience with a traditional MVC framework, you can read more about the relationship to Restlets in this other [FAQ entry](#).

To summarize, a request contains an URI that identifies the target resource that is the subject of the call. This information is stored in the Request resourceRef property and serves as the basis of the routing as we saw. So the first goal when handling a request is to find the target resource which is in the framework... an instance of the Resource class or more precisely one of its subclasses. To help us in this task, we can use the dedicated Finder, a subclass of Restlet, which takes a Resource class reference as an argument and which will automatically instantiate it when a request comes in. Then it will dynamically dispatch the call to the newly created instance, actually to one of its handle*() methods (handleGet, handleDelete, etc.) depending on the request method. Of course, this behavior can be customized. There is even an attach() method on Router that can take two arguments, an URI template and a Resource class and that transparently creates the Finder for you! Now, let's have a look at this overall diagram, showing the relationship between the main framework classes involved in this example:



Back to the code, here is our refactored `Application.createRoot()` method. For simplicity purpose, we didn't keep the `Directory` serving static files as this part wouldn't change. You can notice the way that resource classes are directly attached to the router.

```
// Create a router
Router router = new Router(getContext());

// Attach the resources to the router
router.attach("/users/{user}", UserResource.class);
router.attach("/users/{user}/orders", OrdersResource.class);
router.attach("/users/{user}/orders/{order}",
    OrderResource.class);

// Return the root router
return router;
```

We will finally review one of the resource classes, the `UserResource` class. This class derives from `org.restlet.resource.Resource` and therefore has to override the three arguments constructor. This will initialize the useful "context", "request" and "response" properties on our instances. Then, we use the attribute "user" that is automatically extracted from the `/users/{user}` URI template and store its value in a convenient member variable. At this point, in a full application, we would lookup our associated "user" domain object. Finally, we declare which representation variants we want to expose, in our case only plain text. This will be used at runtime to transparently do some content negotiation, in order to select the preferred variant for each request handled, all this transparently.

```
public class UserResource extends Resource {
    String userName;

    Object user;

    public UserResource(Context context, Request request, Response response) {
        super(context, request, response);
        this.userName = (String) request.getAttributes().get("user");
        this.user = null; // Could be a lookup to a domain object.

        // Here we add the representation variants exposed
        getVariants().add(new Variant(MediaType.TEXT_PLAIN));
    }

    @Override
    public Representation getRepresentation(Variant variant) {
        Representation result = null;
        if (variant.getMediaType().equals(MediaType.TEXT_PLAIN)) {
            result = new StringRepresentation("Account of user \"\"
                + this.userName + "\"");
        }
    }
}
```

```

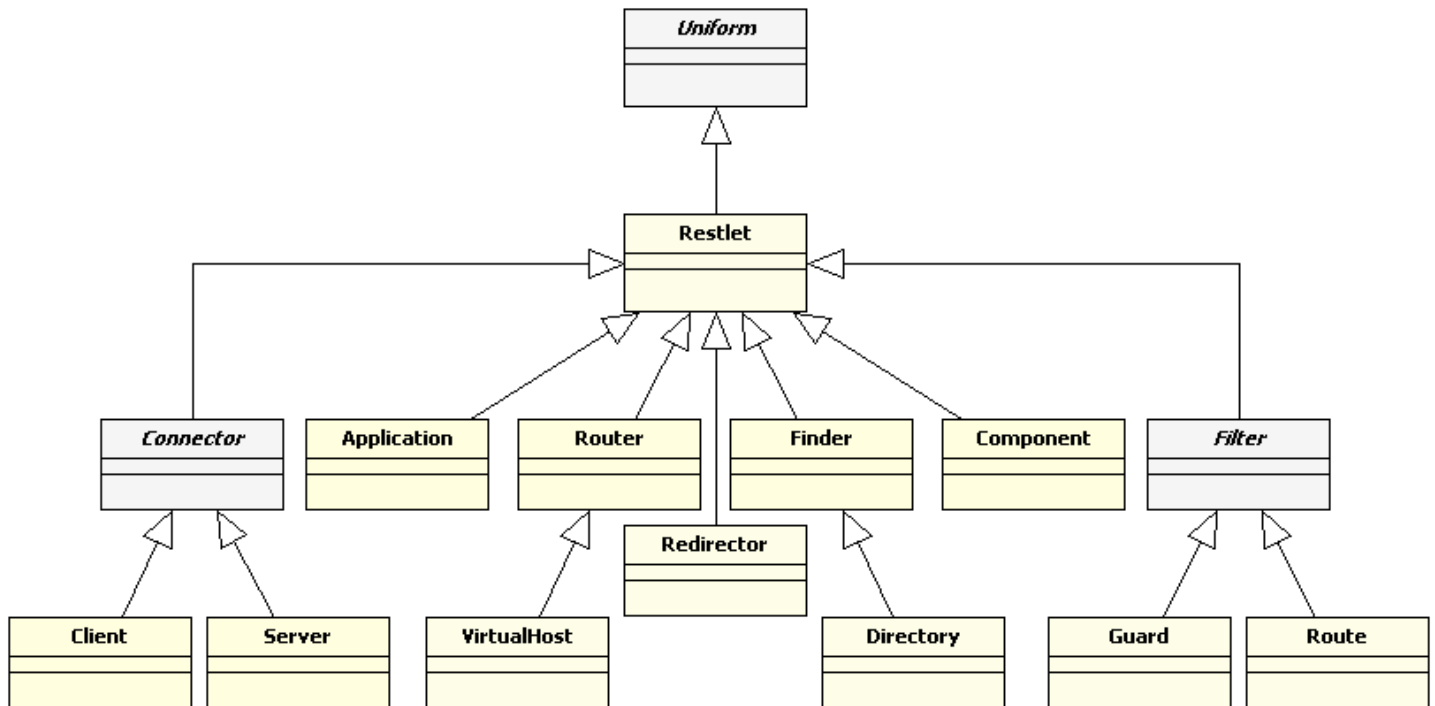
        return result;
    }
}

```

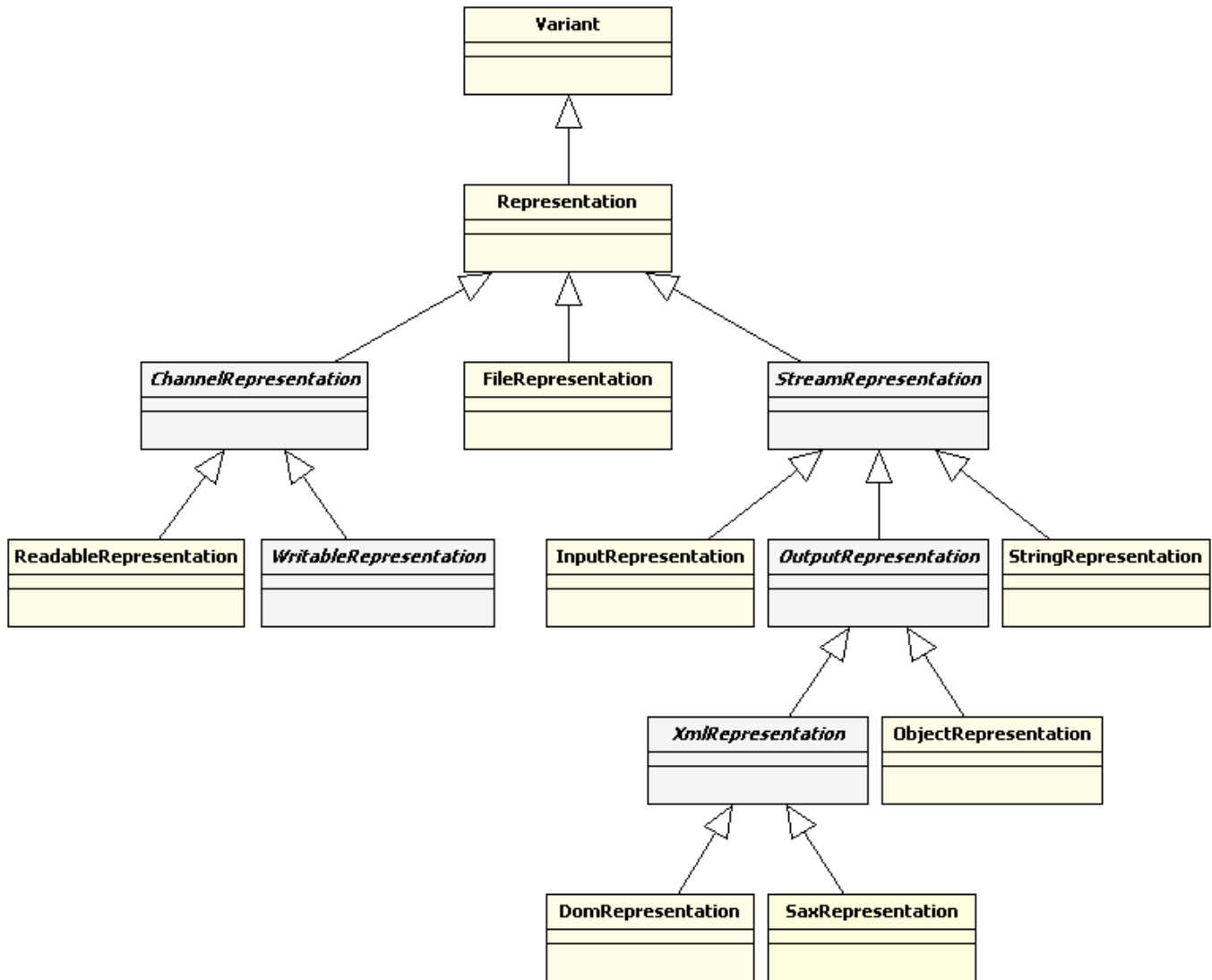
You can have a look at the rest of the code in the tutorial package and test the application. You will obtain the same behavior as in Part11, with the difference that only GET requests will be accepted. If you want to enable PUT for example, you have to create an "allowPut()" method in UserResource that simply returns 'true' then add a "put(Representation)" method to process the call. You can check the Javadocs for further details.

Conclusion

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



Now, here is the hierarchy with the core Representation classes:



Beside this tutorial, your best source of information will be the Javadocs available for the [Restlet API](#), the [Restlet Extensions](#) 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, Dave Pawson and Peter Murray for the feed-back on this tutorial.