Restlet Framework - Tutorial

0.1

Thierry Templier (Restlet)

I. Introduction 1
II. Providing RESTful applications 2
1. Implementing application foundations 3
1.1. Application class 3
1.2. Configuring and organizing routing 3
1.3. Configuring template engines 4
1.4. Configuring static content serving 5
1.5. Linking with application layers 5
2. Organizing routing 7
3. Implementing security 8
4. Implementing resources 9
4.1. A simple server resource 9
4.2. Adding handling methods 9
4.3. Using server resource request methods 10
4.4. Using server resource response methods 10
5. Modularizing resources 11
5.1. Using generics 11
6. Working with JSON 12
7. Working with XML 13
8. Working with templates 14
9. Working with beans 15
10. Deploying applications as standalone server 16
10.1. Choosing the underlying server 16
10.2. Creating the launch processing 16
10.3. Launching the server 18
11. Deploying applications within servlet containers 19
11.1. Organizing the Web application 19
11.2. Configuring the Web application 19
11.3. Deploying the Web application 21
12. Working with headers 22
13. Optimize your Restlet server applications 23
13.1. Using caching 23
13.2. Compressing content 24
13.3. Configuring specific converters 24
14. Troubleshooting 25
14.1. I use a client protocol but it doesn't seem to work correctly 25
14.2. I try to read twice a representation and Restlet throws an exception 25
15. Implementing a simple web API 26
16. Implement a web application 27
III. Accessing RESTful applications 28

Part I. Introduction

TODO: write part intro

Part II. Providing RESTful applications

TODO: write part intro

Chapter 1. Implementing application foundations

In this section, we will implement foundations of your Restlet server applications.

1.1. Application class

The central class of all Restlet applications is the class that extends the Application one from Restlet what the chosen deployment (standalone, servlet container and so). This class mainly defines the way to access provided resources and acts as an entry point.

Example 1.1.

```
public class TestAppApplication extends Application {
    public Restlet createInboundRoot() {
        (...)
    }
}
```

1.2. Configuring and organizing routing

Routing configuration is done within the createInboundRoot method of your Restlet application class and mainly defined using the Router class. Following code describes a simple use case of routing configuration:

Example 1.2.

```
public Restlet createInboundRoot() {
    Router router = new Router(getContext());

    router.attach("login", LoginServerResource.class);
    router.attach("logout", LogoutServerResource.class);
    router.attach("myEntities/", NewsListServerResource.class);
    router.attach("myEntities/{id}", NewsServerResource.class);

    (...)

    return router;
}
```

The sample above corresponds to a very simple use case. In more real and large applications, you have to define a more structing. As a matter of fact, the processing chain can be different according to resources. Let's take a concrete sample. You probably want to have your resources secured and it's not perhaps the case for static content.

For this particularly use case, a good practice consists in leveraging the matching mode of Restlet when attaching resources on routers. This allows defining inner routes for a particular paths. For example, all routes that starts with a pattern will match and handle by the sub route.

An important point to keep in mind is that matching is done sequentially and if a route matches, remaining routes won't be tested.

Following code describes how to define inner routes and to organize routing within your Restlet application:

Example 1.3.

```
public Restlet createInboundRoot() {
    this.configuration = configureFreeMarker(getContext());

    Router router = new Router(getContext());

    router.attach("/static", createStaticApplication())
            .setMatchingMode(Template.MODE_STARTS_WITH);

    router.attach("/", createGlobalApplication())
            .setMatchingMode(Template.MODE_STARTS_WITH);

    return router;
}
```

Here are the skeletons of the createStaticApplication and createGlobalApplication methods:

Example 1.4.

```
    private Restlet createStaticApplication() {
        Router router = new Router(getContext());
        (...)
        return router;
    }

    private Restlet createGlobalApplication() {
        Router router = new Router(getContext());
        (...)
        return router;
    }
```

You can notice that these classes haven't necessary to return a Router instance. An authenticator, a filter and so on are also possible.

1.3. Configuring template engines

You can choose to use template engines, like Freemarker or Velocity, to produce your representation content. This approach is particularly suitable when your representations are HTML pages.

In the case of Freemarker, you have to initialize the template configuration that mainly configures the place to find out template files. This initialization has to be specified within the application class since it must be executed once. You can access then from the application class within resource classes.

Following code shows a method that initializes a Configuration object stores as class variable within the application class:

Example 1.5.

```
    private Configuration configuration;

    public static Configuration configureFreeMarker(Context context) {
        Configuration configuration = new Configuration();
        ClassTemplateLoader loader = new ClassTemplateLoader(
                TestAppApplication.class,
                "/org/restlet/tutorial/server/templates/");
```

```
        configuration.setTemplateLoader(loader);
        // configuration.setCacheStorage(new StrongCacheStorage());
        return configuration;
    }

    public Configuration getConfiguration() {
        return configuration;
    }
```

You can notice that the configureFreeMarker method must be called from the createInboundRoot method when the application is actually initialized, as described below:

Example 1.6.

```
    public Restlet createInboundRoot() {
        this.configuration = configureFreeMarker(getContext());
        (...)
    }
```

1.4. Configuring static content serving

Restlet allows easily serving static content using the Directory class. An instance of this class can be attached to a router for a specific path. The directory can serve all files contained in a specific folder with any folder depth. You can define this folder using supported protocols of Restlet like file and clap.

In the following code, you use a folder contained in our Restlet application from classpath using the CLAP protocol:

Example 1.7.

```
    Router router = new Router(getContext());
    (...)

    Directory directory = new Directory(getContext(),
                "clap://application/org/restlet/tutorial/server/static/");
    directory.setDeeplyAccessible(true);
    router.attach("/static", directory);
```

You must be sure here to have registered the CLAP protocol. See the deployment sections to see how to register client protocols azccording to your target environment. You can also see the troubleshooting section if you have problems when using a client protocol.

You can notice that you can also reference static folder with an absolute path using the file protocol of Restlet. This approach isn't convenient for deployment that uses archive with content like Web archive (WAR) and OSGi bundle.

1.5. Linking with application layers

Restlet corresponds to the Web layers and are based on business and data access layers for processing. The link should be done within the application class. As a matter of fact, entities of such layers are commonly singleton and it's also the case for the application. Restlet keeps a single instance for your application. Following code

describes how to configure a dao within a Restlet application class:

Example 1.8.

```
    public class TestAppApplication extends Application {
        (...)
        private MyEntityDao myEntityDao;
        (...)

        public TestAppApplication() {
            myEntityDao = new MyEntityDaoImpl();
            (...)
        }

        (...)

        public MyEntityDao getMyEntityDao() {
            return myEntityDao;
        }
    }
```

In Restlet, resources aren't singletons and a best practice consists in using the application to get instances from business or data access layers.

Example 1.9.

```
    public class MyResource extends ServerResource {

        @Get
        public Representation getElements() {
            MyEntityDao dao = ((TestAppApplication)getApplication()).getMyEntityDao();
            (...)
        }
    }
```

You can note that if you use IoC container, you can configure injection within the container itself since they commonly support injection of singletons within prototype entities.

Chapter 2. Organizing routing

Chapter 3. Implementing security

Chapter 4. Implementing resources

Server resources are the elements that provide processing related to paths.

4.1. A simple server resource

Restlet provides the ServerResource class to implement server resource. A server resource is simply a sub-class of the ServerResource class. One specificity in Restlet is that a server resource instance is created for each REST request. So you can define class variables in it but they will be available for the request.

Following code describes a simple custom server resource:

Example 4.1.

```
public class MyServerResource extends ServerResource {
    (...)
}
```

The MyServerResource class can be attached to a path for the application, as described in the "Implementing application foundations" section.

4.2. Adding handling methods

We can add now methods to handle HTTP methods for the path(s) the server resource is attached with. Restlet uses annotations for that, as listed below:

- Get: the associated method handles an HTTP GET method
- Put: the associated method handles an HTTP PUT method
- Post: the associated method handles an HTTP POST method
- Delete: the associated method handles an HTTP DELETE method
- Head: the associated method handles an HTTP HEAD method

The following code describes how to add annotated methods to a server resource class:

Example 4.2.

```
public class MyServerResource extends ServerResource {
    @Get
    public Representation handleGetMethod() {
        (...)
    }

    @Put
    public Representation handlePutMethod() {
        (...)
    }

    @Delete
```

```
    public void handleDeleteMethod() {
        (...)
    }
}
```

You can notice that the method signature defers according to the associated annotation. For example, we expect a GET method to send back a representation, a PUT one to receive and send back representations, a DELETE one to exchange no representation.

4.3. Using server resource request methods

When attaching server resources for a particular path, you can commonly define attributes. Accessing the request attributes Accessing the request query

4.4. Using server resource response methods

Handling redirects

Chapter 5. Modularizing resources

DRY (Don't Repeat Yourself (see http://en.wikipedia.org/wiki/DRY) is a common principle of sofware development that aimed at reducing repetition of information of all kinds. In this section, we will focus how aspects that helps to improve resource processing.

5.1. Using generics

Restlet supports generics to modularize common processing. Let's tackle such aspects when implementing CRUD processing for resources. As described previously in section XX, "", we implement list / add / update / delete operations based on two Restlet server resources:

- The one that handles element list and adding
- The one that handles particular element

Chapter 6. Working with JSON

Chapter 7. Working with XML

Chapter 8. Working with templates

Chapter 9. Working with beans

Chapter 10. Deploying applications as standalone server

Now you have a Restlet applications implemented based on previous described use cases:

- Implementing a simple Web API
- Implementing a Web application

it's time to see deploy and execute them within a server. Restlet provides several approaches for this issue and we will focus here on the one based on a standalone server.

10.1. Choosing the underlying server

Restlet integrates a smart autodetect processing of available extensions. You simply have to put them in the classpath to make them available. This also applies for server connectors usable for a standalone Restlet server:

- Internal NIO server
- Simple framework

By default, meaning no additional extension provided a server connector is available, the Restlet internal NIO server is used.

If you put the simple extension, the Simple Framework server will automatically use as underlying HTTP server. In this case, you will see messages like that in the traces:

```
1 nov. 2012 11:35:05 org.restlet.ext.simple.SimpleServerHelper start
INFO: Starting the Simple [HTTP/1.1] server on port 8182
```

10.2. Creating the launch processing

There are two approaches when implementing the launch processing:

- At application level
- At component level

The first one is the simplest but the most restricted. It simply allows attaching a Restlet application to a server to execute it. But you can't set more configurations here. Following code describes how to implemente this approach:

Example 10.1.

```
Server server = new Server(Protocol.HTTP, 8182);
server.setNext(new MyApplication());
```

The second on is the recommended one since it provides flexility when configuring application configuration for execution. Following lists gives a non-exhaustive list of supported elements for configuration:

- Server connectors
- Client connectors
- Virtual hosts
- Internal routing
- Restlet services

You first need to create a component and eventually set some hints about it as described below:

Example 10.2.

```
Component component = new Component();
component.setName("My component");
component.setDescription("My component description");
component.setOwner("Restlet");
component.setAuthor("The Restlet Team");
```

The second important point to configure now is connectors for both server(s) and client(s). Since we want to make available a server application, a server protocol is mandatory, an HTTP one in our case. If you want to access other resources using Restlet, you need to add client protocols like *FILE* or *HTTP*. Following code describes how to configure such aspects:

Example 10.3.

```
getServers().add(Protocol.HTTP, 8182);
getClients().add(Protocol.FILE);
getClients().add(Protocol.HTTP);
getClients().add(Protocol.HTTPS);
```

The final thing to do to configure our component is to define how to make available our application through the component. Several possibilities are supported within Restlet:

- Using the default host:

Example 10.4.

```
component.getDefaultHost().attach("/app", new MyApplication());
```

- Using the internal router to only make available the application within a same process using the RIAP protocol:

Example 10.5.

```
component.getInternalRouter().attach("/app", new MyApplication());
```

- Using a virtual host to precisely control the domain to access the application:

Example 10.6.

```
VirtualHost virtualHost = new VirtualHost();
virtualHost.setHostDomain("www.myapp.org");
virtualHost.setHostPort("80|8182");
component.getHosts().add(virtualHost);
```

Let's finally see how startup and shutdown you server.

10.3. Launching the server

For both approaches, *start* and *stop* methods are available to start and stop the server

Example 10.7.

```
// Application level approach
Server server = (...)
server.start();

// Component level approach
Component component = (...)
component.start();
```

You can notice that the *start* method isn't blocking so you can consider using an stdin read line to detect when stopping the server, as described below:

Example 10.8.

```
(...)
System.out.println("Press a key to stop");
System.in.read();

// Application level approach
if (server!=null) {
    server.stop();
}
// Component level approach
if (component!=null) {
    component.stop();
}
```

Chapter 11. Deploying applications within servlet containers

Now you have a Restlet applications implemented based on previous described use cases:

- Implementing a simple Web API
- Implementing a Web application

it's time to see deploy and execute them within a server. Restlet provides several approaches for this issue and we will focus here on the one based on a standalone server.

11.1. Organizing the Web application

There is nothing specific here to Restlet but you need to follow the organization of Web applications of Java EE, as below:

```
WebAppRoot
    +- WEB-INF
        +- classes
        +- lib
        -- web.xml
```

The `WEB-INF/classes` folder must contain the classes of your Web application, `WEB-INF/lib` the jar files of the libraries and frameworks used and the `web.xml`, the configuration of the Web application.

You can notice that, in the case of GAE (Google App Engine), you need to add an additional file named appengine-web.xml within the `WEB-INF` folder for configurations related to the GAE environment.

11.2. Configuring the Web application

Restlet provides a servlet adapter for servlet container and Restlet applications must be configured in this case within the `web.xml` file. You will find all elements that can be configured on a Restlet component since this servlet corresponds to a wrapper upon it.

The following code describes the minimal content of the `web.xml` file to configure Restlet within a servlet environment:

Example 11.1.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xmlns="http://java.sun.com/xml/ns/javaee"
         xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
         xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
                  http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
         id="myApplication" version="2.5">
    <display-name>My Application</display-name>

    <context-param>
        <param-name>org.restlet.application</param-name>
        <param-value>org.restlet.tutorial.MyApplication</param-value>
    </context-param>
```

```
    <servlet>
        <servlet-name>ServerServlet</servlet-name>
        <servlet-class>org.restlet.ext.servlet.ServerServlet</servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>ServerServlet</servlet-name>
        <url-pattern>/*</url-pattern>
    </servlet-mapping>
</web-app>
```

You can enhance this configuration by adding the client protocols you want to use with the `org.restlet.clients` attribute of the servlet, as described below:

Example 11.2.

```
<web-app (...)>
    <servlet>
        (...)
        <init-param>
            <param-name>org.restlet.clients</param-name>
            <param-value>CLAP,HTTP</param-value>
        </init-param>
    </servlet>
</web-app>
```

You probably have all you need with this approach but you also can choose to define your own component object and configure it within the `web.xml` file, as described below. In this case, initialization will take it rather than initialization parameters.

Example 11.3.

```
<web-app (...)>
    <servlet>
        (...)
        <init-param>
            <param-name>org.restlet.component</param-name>
            <param-value>org.restlet.tutorial.MyComponent</param-value>
        </init-param>
    </servlet>
</web-app>
```

In the case of deployment for GAE, the appengine-web.xml file contains configurations related to GAE and there is nothing regarding Restlet. The following code describes a minimal content for this file and you don't provide here more details. See the GAE documentation if necessary.

Example 11.4.

```
<?xml version="1.0" encoding="utf-8"?>
<appengine-web-app xmlns="http://appengine.google.com/ns/1.0">
    <application>myapplication</application>
    <version>1</version>
    <threadsafe>true</threadsafe>
    <sessions-enabled>false</sessions-enabled>

    <system-properties>
        <property name="java.util.logging.config.file" value="WEB-INF/logging.properties"/>
    </system-properties>
```

```
    <env-variables>
        <env-var name="DEFAULT_ENCODING" value="ISO-8859-1" />
    </env-variables>
</appengine-web-app>
```

Let's deploy now the application.

11.3. Deploying the Web application

Deployment of the application follows mechanisms provided by servlet container and different execution environment. For more details, have a look at the corresponding documentations.

Chapter 12. Working with headers

Chapter 13. Optimize your Restlet server applications

This section describes some hints to optimize Restlet server applications.

13.1. Using caching

One possible optimization is not to serve related resources (like images, css...) when loading a particular resource with HTML content. An approach can be to use cache support provided by HTTP.

We describe here how to apply browser caching for all static elements loaded from a path with a subfolder called nocache. For these elements, headers for caching will be automatically added. For others, an expiration date of one month will be specified in headers.

This feature can be simply added with Restlet using filters within the method createInbountRoot of your application class. A filter containing caching stuff can to be added in front of the Restlet Directory that serves static content, as described below:

Example 13.1.

```
router.attach("/static", new Filter(getContext(), new Directory(
            getContext(), (...))) {
    protected void afterHandle(Request request, Response response) {
        super.afterHandle(request, response);
        [adding caching stuff here]
    }
});
```

Once the filter is added in the processing chain, we have to handle caching headers based on the Representation and Response objects. The noCache method of the Response automatically adds the related headers for no cache. For expiration date, the setExpirationDate method of the Representation allows defining the laps of time before reloading the element content. Following code describes the complete code:

Example 13.2.

```
router.attach("/static", new Filter(getContext(), new Directory(
            getContext(), (...))) {
    protected void afterHandle(Request request, Response response) {
        super.afterHandle(request, response);
        if (response.getEntity() != null) {
            if (request.getResourceRef().toString(false, false)
                    .contains("nocache")) {
                response.getEntity().setModificationDate(null);
                response.getEntity().setExpirationDate(null);
                response.getEntity().setTag(null);
                response.getCacheDirectives().add(
                            CacheDirective.noCache());
            } else {
                response.setStatus(Status.SUCCESS_OK);
                Calendar c = new GregorianCalendar();
                c.setTime(new Date());
                c.add(Calendar.DAY_OF_MONTH, 1);
                response.getEntity().setExpirationDate(c.getTime());
                response.getEntity().setModificationDate(null);
            }
        }
```

```
    }
});
```

13.2. Compressing content

Modern browsers support compression for received content. This allows to reduce payload of exchanged data. Restlet supports this feature for server-side application using the Encoder class. The latter can take place within the processing chain like router, authenticator and filter. You simply need to configure it within the method createInbountRoot of your application class, as described below:

Example 13.3.

```
Encoder encoder = new Encoder(getContext(), false, true, getEncoderService());
encoder.setNext(router);
return encoder;
```

13.3. Configuring specific converters

Example of Jackson converter.

Example 13.4.

```
private JacksonConverter getRegisteredJacksonConverter() {
    JacksonConverter jacksonConverter = null;
    List<ConverterHelper> converters = Engine.getInstance()
            .getRegisteredConverters();
    for (ConverterHelper converterHelper : converters) {
        System.out.println(converterHelper.getClass());
        if (converterHelper instanceof JacksonConverter) {
            jacksonConverter = (JacksonConverter) converterHelper;
            break;
        }
    }
}
```

Getting the ObjectMapper and configures it for data mapping.

Example 13.5.

```
private void configureJacksonConverter() {
    JacksonConverter jacksonConverter = getRegisteredJacksonConverter();

    if (jacksonConverter != null) {
        ObjectMapper objectMapper = jacksonConverter.getObjectMapper();
        objectMapper.configure(
                SerializationFeature.WRITE_DATES_AS_TIMESTAMPS, false);
        // objectMapper.configure(DeserializationFeature.WRITE_DATES_AS_TIMESTAMPS,
        // false);
        objectMapper.setDateFormat(
                new SimpleDateFormat("yyyy-MM-dd"));
    }
}
```

Chapter 14. Troubleshooting

This section describes how to solve common server problems with Restlet.

14.1. I use a client protocol but it doesn't seem to work correctly

TODO:

14.2. I try to read twice a representation and Restlet throws an exception

TODO:

Chapter 15. Implementing a simple web API

Chapter 16. Implement a web application

Part III. Accessing RESTful applications

TODO: write part intro