http://www.yegor256.com/2014/04/11/jcabi-http-intro.html

Fluent Java HTTP Client

11 April 2014 modified on 29 October 2014 Yegor Bugayenko

In the world of Java, there are plenty of HTTP clients from which to choose. Nevertheless, I decided to create a new one because none of the other clients satisfied fully all of my requirements. Maybe, I'm too demanding. Still, this is how my jcabi-http client interacts when you make an HTTP request and expect a successful HTML page in return:

```
String html = new JdkRequest("https://www.google.com")
  .uri().path("/users").queryParam("id", 333).back()
  .method(Request.GET)
  .header("Accept", "text/html")
  .fetch()
  .as(RestResponse.class)
  .assertStatus(HttpURLConnection.HTTP_OK)
  .body();
```

I designed this new client with the following requirements in mind:

Simplicity

For me, this was the most important requirement. The client must be simple and easy to use. In most cases, I need only to make an HTTP request and parse the JSON response to return a value. For example, this is how I use the new client to return a current EUR rate:

```
String rate = new JdkRequest("http://www.getexchangerates.com/api/la
  .header("Accept", "application/json")
  .fetch()
```

```
  .as(JsonResponse.class)
  .json().readArray().getJsonObject(0)
  .getString("EUR");
```

I assume that the above is easy to understand and maintain.

Fluent Interface

The new client has to be fluent, which means that the entire server interaction fits into one Java statement. Why is this important? I think that fluent interface is the most compact and expressive way to perform multiple imperative calls. To my knowledge, none of the existing libraries enable this type of fluency.

Testable and Extendable

I'm a big fan of interfaces, mostly because they make your designs both cleaner and highly extendable at the same time. In jcabi-http, there are five interfaces extended by 20 classes.

`Request` is an interface, as well as `Response`, `RequestURI`, and `RequestBody` exposed by it.

Use of interfaces makes the library highly extendable. For example, we have `JdkRequest` and `ApacheRequest`, which make actual HTTP calls to the server using two completely different technologies: (JDK `HttpURLConnection` and Apache Http Client, respectively). In the future, it will be possible to introduce new implementations without breaking existing code.

Say, for instance, I want to fetch a page and then do something with it. These two calls perform the task differently, but the end results are the same:

```
String uri = "http://www.google.com";
Response page;
page = new JdkRequest(uri).fetch();
page = new ApacheRequest(uri).fetch();
```

XML and JSON Out-of-the-Box

There are two common standards that I wanted the library to support right out of the box. In most cases, the response retrieved from a server is in either XML or JSON format. It has always been a hassle, and extra work, for me to parse the output to take care of formatting issues.

jcabi-http client supports them both out of the box, and it's possible to add more formats in the future as needed. For example, you can fetch XML and retrieve a string value from its element:

```
String name = new JdkRequest("http://my-api.example.com")
  .header("Accept", "text/xml")
  .fetch()
  .as(XmlResponse.class)
  .xml().xpath("/root/name/text()").get(0);
```

Basically, the response produced by `fetch()` is decorated by `XmlResponse`. This then exposes the `xml()` method that returns an instance of the `XML` interface.

The same can be done with JSON through the Java JSON API (JSR-353).

None of the libraries that I'm aware of or worked with offer this feature.

Immutable

The last requirement, but certainly not the least important, is that I need all interfaces of the library to be annotated with `@Immutable`. This is important because I need to be able to encapsulate an instance of `Request`

in other immutable classes.

ps. A short summary of this article was published at JavaLobby