Polytech INFO4, F. Boyer, O. Gruber

TP – Implementing your own Web Server

Organisation	Binome
Evaluation	Code-based
Advised time	8-10h per binome
Tools	JDK ==1.8, Eclipse
Return	**Eclipse Projet**, named **LastName1.LastName2**. In compressed format, either zip or .tar (**nothing else**), **on moodle**
Contact	Fabienne.Boyer@univ-grenoble-alpes.fr Olivier.Gruber@univ-grenoble-alpes.fr

1. Objective

This practical work aims at programming a Web Server over TCP/IP. You will have to design and implement a Web Server able to serve static HTML pages as well as dynamic requests that involve *ricmlets*. Ricmlets are very closed to regular HTTP servlets, just a little bit simpler. Ricmlets provide clients with the ability to run Java code, server side, following a well-defined API.

To be usable from any browser, your Web Server will follow the specifications of the HTTP protocol. However **for time reasons, we will only consider serving the GET requests of the HTTP protocol.** As you all know, GET requests allow to get either static pages or dynamic ones, as recalled below.

2. HTTP Protocol Basics

Let's consider the basics of the HTTP protocol. The protocol recognizes HTTP requests and HTTP responses, both using the same generic message format (RFC7230). We will consider a basic version described below:

```
HTTP-message grammar = start-line
                       *( header-field CRLF )
                       CRLF
                       [ message-body ]
```

Let's discuss the different parts:

- A start-line present in any HTTP message
- Zero or more header fields, each header field being terminated by the two characters CR and LF
- An empty line marking the end of the header
- Optionally, a message body as a sequence of bytes

Overall, the message contains US-ASCII characters. Special characters are CR, LF, SP. The character **CR** is the US-ASCII value 13 (0xD), the character **LF** is the US-ASCII value 10 (0xA). **SP** (space) is the ASCII value 32 (0x20).

The start-line corresponds to either a request-line (for requests) or a status-line (for responses).

start-line = request-line | status-line

request-line = method SP resource-id SP HTTP-version CRLF (`ex: GET /hello.html HTTP/1.1`)

status-line = HTTP-version SP status-code SP reason-phrase CRLF (`ex: HTTP/1.1 200 OK`)

Header fields are lines beginning with a field name, followed by a colon (":"), followed by a field body, and terminated by CRLF.

header-field = field-name : field-body. (`ex: Content-type: text/html`)

STEP1 - Serving static Pages

Client-Side Protocol:

- The client connects to the server, via an TCP/IP socket
- The client sends a request with a start-line indicating a method and a resource identifier. The resource identifier may be followed by the version of the HTTP protocol used by the client. For instance, to get a static page named hello.html server side, the start-line may be:

```
GET /hello.html HTTP/1.1
```

Server-Side Protocol:

- The server accepts on a given port for connections from remote clients.
- For any accepted connection, the server creates a dedicated worker thread (for simplicity, we advise using a basic multi-threaded design, but feel free to decide otherwise).
- When receiving a request, the server looks for a file corresponding to the requested page.
- If the file is found, the server returns a response composed of a start-line indicating a success, followed by a header indicating the length and the type of the body. After the header, the server inserts an empty line and then the file content.

```
HTTP/1.1 200 OK
Date: …
Server: Ricm4HttpServer
Content-length: 45876
Content-type: text/html
```

- To build the response header, you may get the content type through the suffix of the resource identifier given in the url (html, txt, gif, jpg, etc). Just use the static method ***getContentType(String)*** **that is given to you** in the class *HttpRequest*.
- In case the requested file is not found at server side, the server must reply with an error code

```
HTTP/1.0 404 File not found
```

Work to do:

1. **Understand** the interfaces *HttpRequest* and *HttpResponse* in the package *httpserver.itf, (HttpRequest* being an interface defined through an abstract class).

2. U**nderstand** the class *HTTPServer* in the Java package *httpserver.itf.impl* and how it uses the classes *HttpRequest* and *HttpResponseImpl*. Together, these classes implement a web server who serves HTTP GET requests and returns the corresponding static HTML pages.

3. Then **complete the class *HttpStaticRequest*** that handles static HTTP requests.

4. Check your implementation with a local browser: first, start your HttpServer (you may start it with '.' as default folder). Then try to get the url *localhost:<port>/FILES/hello.html*. Try to get the other resources given in the File directory to check that your server works well with different resource types (*gif, jpeg*, ..). Of course, check also that the url *localhost:<port>/FILES* (as well as *localhost:<port>/FILES/*) displays the index.html page that is under the FILES directory on the server.

5. Check also your HTTP server from a terminal, through executing the *wget* command (install *wget* if required). The command is of the form: wget <url>

6. If you have time, check your HTTP server from a remote browser through the url *<hostname>:<port>/hello.html*.

STEP2 - Dynamic Pages

We now consider dynamic pages, meaning that clients requests can trigger the execution of server-side classes (that we call *ricmlets*) that return dynamically computed HTML pages. As for Java servlets, a ricmlet should define a method *doGet(..)* that takes as argument request and response objects. The protocol is the following.

Protocol at client side

- The client connects to the server.
- The client sends a request with a start-line indicating a method and a resource identifier. The resource starts by */ricmlets* and indicates the name of the ricmlet to execute. For instance, to execute a ricmlet corresponding to the class *examples.HelloRicmlet* , the start-line will be:

```
GET /ricmlets/examples/HelloRicmlet HTTP/1.1
```

- In case your ricmlet expects arguments, these are sent along with the resource identifier, as with regular servlets:

```
GET /ricmlets/examples/HelloRicmlet?name=Bob&surname=Marley HTTP/1.1
```

Protocol at server side

- The server waits on a given port for connections from remote clients.
- When receiving a request, the HTTP server determines if it is a static or dynamic request. In case of dynamic request, **it looks for the class corresponding to the requested ricmlet under the *ricmlets* package**.
- If the class is found, the server instantiates it if it is not already instantiated. It is very important to respect this lifecycle rule: we want that each ricmlet behaves as a singleton, meaning that there should exist at most one instance per ricmlet class.
- Once the ricmlet instance has been found, the server executes the *doGet()* method, giving as arguments two objects: one representing the request and another one representing the response. Among others, the *request* object allows to get the arguments, the *response* object allows to get the stream on which the response body should be sent. See the class *HelloRicmlet.java* for an example of a ricmlet class.
- In case the Ricmlet class is not found at server side, the server replies with an error code

```
HTTP/1.0 404 Ricmlet not found
```

Notice that ricmlet classes do not set up the *content-length* header field in their response. They produce HTML responses dynamically and incrementally, the bytes composing the response being directly written on the server output stream for efficiency reasons. It is thus difficult to predetermine the length of the response before producing the response itself. Hopefully, not indicating the length of the response is not an issue for HTML responses, because browsers have the ability to detect when the full response has been received through HTML tags.

Work to do

- Complete your HttpServer class to support dynamic requests (in a very first step, you may not consider arguments). We ask you to define two additional classes: *HttpRicmletRequestImpl* and *HttpRicmletResponseImpl*, that respectively extends *HttpRicmletRequest* and *HttpRicmletResponse*.

  Remind that you can instantiate a class whose name is stored in a string as follow (assuming the class has a no-argument public constructor):

  String clsname = "a.b.c.Foo";
  Class<?> c = Class.forName(clsname); c.getDeclaredConstructor().newInstance();

- Check your implementation from a browser: try to get the url localhost:<port>/ricmlets/examples/HelloRicmlet.
- Check that your ricmlet lifecycle management is correct. To this end, use the ricmlet *CountRicmlet* that delivers a page indicating the number of times it has been processed. Check that getting the url localhost:<port>/ricmlets/examples/CountRicmlet several times actually increments this number.
- Then consider managing arguments, through implementing the *getArgs()* method in the class HttpRicmletRequest. Check with your browser that the url <hostname>:<port>/ricmlets/examples/HelloRicmlet?name=Bob&surname=Marley delivers a page including a *Hello Bob Marley* message.

STEP3 - Introducing Cookies

We now consider *cookies*, a concept of the Http protocol allowing to keep some data at client-side that will be automatically integrated into Http requests and responses.
Cookies sent by the browser can be get from the *HttpRicmletRequestImpl* object (*getCookie(..)* method). Cookies returned by the server to the browser can be set in the *HttpRicmletResponseImpl* object (*setCookie(..)* method). By default, a cookie that is not sent back to the browser will remain unchanged at browser side.

The Http server should automatically integrate the cookies that have been added to the *HttpRicmletResponseImpl* object in the response made to the browser. This is achieved through the `Set-Cookie` directive (see below). A response header may include several *Set-Cookie* directives. Moreover, a *Set-cookie* directive may define several cookies, as illustrated below.

```
HTTP/4.1 200 OK
…
Set-Cookie: myFirstCookie=123;mySecondCookie=Hello
Set-Cookie: anotherCookie=45678
Server: Ricm4HttpServer
Content-length: 45876
Content-type: text/html
```

When a browser receives an Http response, it automatically adds/modify the defined cookies to its local space of cookies (if a cookie was previously defined, its value may be overwritten). If a cookie is assigned a null value and a max-age of 0 (ex : `anotherCookie=; Max-Age=0`), the browser will delete it from its local space. Otherwise, once created, a cookie will stay alive as long as it expiration date (if any) authorizes it.

Work to do

- Complete your classes to support cookies. First, in the constructor of the class *HttpRicmletRequestImpl*, parse the remaining request header and build a data-structure keeping the defined cookies. You may change the constructor interface to get this remaining header (e.g., add a *BufferedReader* argument).

- Define) the method *getCookie(String name)* in the class *HttpRicmletRequestImpl* and the method *setCookie(String name, String value)* and *setCookie(String name, String value)* in the class *HttpRicmletResponseImpl* . These two methods can be used by the application layer (the first to get the cookies sent by the browser, the second to set the cookies to return back to the browser).

- Manage the fact that cookies that have been set in the *HttpRicmletResponseImpl* object shall be integrated in the Http response (adding `Set-Cookie` directives in the response output stream).

- Check your implementation through defining a class *MyFirstCookieRicmlet* that gets a cookie named "MyFirstCookie", creating it if it does not exist, and change its value each time it is called.

STEP4 - Session Management

We now consider *sessions*, a concept discussed during the lecture on HTTP. We will consider non-persistent sessions, meaning sessions that do not survive a kill-restart cycle on your *HttpServer*.

A session corresponds to data associated to a given client but kept at server-side. This data is structured as a set of *(Object key, Object value)* pairs. The *doGet(..)* method of an ricmlet has the ability to get / modify / create session data, through getting the reference of the *Session* object associated to the current client and calling the methods *set/get* on this object (see interface *HttpSession*).
Thus, your Http server should be able to get the session object associated to a client when receiving a request, creating this object if it does not already exist. To this end, we suggest that you do as many Web servers do: use a dedicated cookie (named for example *session-id*) to store at client-side a key allowing to get the session object associated to the client. The Http server will automatically integrate this cookie in any response and the client browser will automatically integrate this cookie in any request sent to the Http server.

Session destructions work as follow. The Http server will be in charge of automatically destruct any session object in case no interaction from a client occurred for a given time.

Work to do

- Complete your classes to support sessions. In a first step, do not manage of session destructions, just focus on session creation. Define a class *Session* that implements the given *HttpSession* interface. Then instantiate this class at the right moment in your Server implementation.

- Check your implementation through the ricmlet *CountBySessionRicmlet* that counts the number of times it has been processed per session. To this end, fetch the following url from two distinct client sessions, checking that each session gets its proper counter:
  http://localhost:<port>/ricmlets/examples/CountBySessionRicmlet

  To act as two users such as to get two distinct sessions, open two browsers (e.g., *firefox* and *chrome*) on your machine. If your configuration forces these two browsers to share a same cookie space, you may use a local browser and a remote one. You may also use a local browser and a local terminal running the *wget* command.

- [OPTIONAL] Then manage session destructions. Remember that a session gets automatically destroyed in case the client does not issue any request for a given duration.

STEP5 - Application Management [OPTIONAL]

We finally extend our Web Server with the notion of *applications* to give a way to modularize ricmlet classes. We will consider that an application defines a set of ricmlet classes through a Java archive file. All archive files will be put at a given folder at server-side. Then, urls used to trigger the execution of ricmlets will follow the pattern given below:
<hostname>:<port>/ricmlets/<appName>/<fullRicmletClassName>.

Upon such request, the server will look for the class <ricmletClassName> in the <appName>.jar file. Then it will get the *classloader* associated to the application appName and request it to instantiate this class if not yet done. As we have seen together during the lecture, using a classloader per application is the only way to ensure that an application cannot instantiate a class belonging to another application.

Work to do

- Define a class *Application* that provides a method *getInstance(String className, String appName, ClassLoader parent)* allowing to get the instance of the ricmlet associated to the class *className*, for the account of the application *appName*.

- Adapt your *HttpServer* classes to take into account applications.

- Check your implementation. For this, make a jar file (e.g., *app1.jar)*, containing the ricmlet *CountRicmlet* for instance. Then, fetch the url http://localhost:<port>/app1/CountRicmlet from your browser and check that the count printed by CountRicmlet increases each time you fetch the url.

- Then make another jar file named *app2.jar*, containing the same ricmlet classes. Then, fetch the url http://localhost:<port>/app2/CountRicmlet from your browser. You should observe that the count printed by *CountRicmlet* starts at 0.