EDA095
URLConnections

Pierre Nugues

Lund University
http://cs.lth.se/home/pierre_nugues/

March 27, 2014

Covers: Chapter 7, *Java Network Programming*, 4th ed., Elliotte Rusty Harold

URLConnection

`URLConnection` represents a communication link between the application and a URL.
It is created from a URL object that calls `openConnection()`
We have seen how to open a stream from a URL:

```
URL myDoc = new URL("http://cs.lth.se/");
InputStream is = myDoc.openStream();
```

These lines are equivalent to:

```
URL myDoc = new URL("http://cs.lth.se/");
URLConnection uc = myDoc.openConnection();
InputStream is = uc.getInputStream();
```

They are more complex than `openStream()` but more flexible.

Reading a URL

```
try {
  URL myDoc = new URL("http://cs.lth.se/");
  URLConnection uc = myDoc.openConnection();
  InputStream is = uc.getInputStream();
  BufferedReader bReader =
    new BufferedReader(new InputStreamReader(is));
  String line;
  while ((line = bReader.readLine()) != null) {
    System.out.println(line);
  }
} catch (Exception e) { e.printStackTrace(); }
```

//ReadURL.java
Nearly the same as in ViewHTML.java except that we have a `URLConnection` object instead of an `InputStream` object.

URLConnection

`URLConnection` enables the programmer to have more control over a connection:

- Access the header fields
- Configure the client properties
- Use more elaborate commands (`POST`, `PUT` for HTTP)

Reading the Header

The header is part of the HTTP protocol and consists of a list of pairs: parameter/value. There are two main methods to read it:

- `String getHeaderFieldKey(int n)` // the parameter name of the nth header
- `String getHeaderField(int n)` // the parameter of the nth header

Headers have typical parameters: Date, Content type, etc.
There are shortcuts to access them:

- `String getContentEncoding()`
- `String getContentType()`
- `long getDate()`, etc.

Reading the Header (I)

Extracting the complete list:

```
try {
  URL myDoc = new URL("http://cs.lth.se/");
  URLConnection uc = myDoc.openConnection();
  for (int i = 0; ; i++) {
    String header = uc.getHeaderField(i);
    if (header == null) break;
    System.out.println(uc.getHeaderFieldKey(i) + ": "
      + header);
  }
} catch (Exception e) {
  e.printStackTrace();
}
```

// ReadHeader.java

Reading the Header (II)

Extracting selected parameters:

```
try {
  URL myDoc = new URL("http://cs.lth.se/");
  URLConnection uc = myDoc.openConnection();
  System.out.println("Date: " + new Date(uc.getDate()));
  System.out.println("Content type: " + uc.getContentType());
  System.out.println("Content encoding: " +
    uc.getContentEncoding());
  System.out.println("Last modified: " + uc.getLastModified()
} catch (Exception e) {
  e.printStackTrace();
}
```

// ReadHeader2.java

MIME

The MIME (Multipurpose Mail Internet Extensions) is a tag to identify the content type. RFC 2045 and 2046 (`http://tools.ietf.org/html/rfc2045`)
MIME defines a category and a format: a type and a subtype
Useful MIME types are `text/html`, `text/plain`, `image/gif`, `image/jpeg`, `application/pdf`, and so on.
In ReadHeader.java, let's replace

```
URL myDoc = new URL("http://cs.lth.se/");
```

with

```
URL myDoc = new URL("http://fileadmin.cs.lth.se/cs/Bilder/
  Grundplatta-3.jpg");
```

HTTP servers should send a content type together with data. It is not always present however.
Sometimes, the client has to guess using `URLConnection.guessContentTypeFromStream()`

Downloading Text (and Ignoring the Rest)

```
public ArrayList<URL> readURL(URL url) {
  LinkGetter callback = null;
  try {
    URLConnection uc = url.openConnection();
    String type = uc.getContentType().toLowerCase();
    // We read only text pages
    if ((type != null) && !type.startsWith("text/html")) {
      System.out.println(url + " ignored. Type " + type);
      return null;
    }
    ParserGetter kit = new ParserGetter();
    HTMLEditorKit.Parser parser = kit.getParser();
...
}
```

Caches

When a web page is viewed multiple times, the images it can contain are downloaded once and then retrieved from on a cache on the local machine. The cache can be controlled by HTTP headers.

- Expires (HTTP 1.0)
- Cache-control (HTTP 1.1): max-age, no-store
- Last-modified
- ETag: A unique identifier sent by the server. The identifier changes when the resource changes.

It is possible to write a cache manager in Java, the idea is: If you access a URL, check if it is in the cache before downloading it.
See the book, pages 203–208.

Configuring the Parameters

A set of `URLConnection` methods enables a program to read and modify the connection's request parameters:

- `protected boolean connected` //false
- `protected boolean doInput` // true
- `protected boolean doOutput` // false
- `protected URL url`, etc.

The methods to read and modify the connection are:

- `URL getURL()`
- `boolean getDoInput()`
- `void setDoInput(boolean)`
- `String getRequestProperty(String key)`
- `void setRequestProperty(String key, String value)`

etc.

Getting the Parameters

```
try {
  URL myDoc = new URL("http://cs.lth.se/");
  URLConnection uc = myDoc.openConnection();
  System.out.println("URL: " + uc.getURL());
  System.out.println("Do Input: " + uc.getDoInput());
  System.out.println("Do Output: " + uc.getDoOutput());
  uc.setDoOutput(true);
  System.out.println("Do Output: " + uc.getDoOutput());
} catch (Exception e) {
  e.printStackTrace();
}
```

// ReadParameters.java

Google

The input box of the Google page

```
<form action=/search name=f>...
  <input type=hidden name=hl value=sv>
  <input maxLength=256 size=55 name=q value="">
  <input type=submit value="Google-sökning" name=btnG>
  <input type=submit value="Jag har tur" name=btnI>
  <input id=all type=radio name=meta value="" checked>
    <label for=all> webben</label>
  <input id=lgr type=radio name=meta value="lr=lang_sv" >
    <label for=lgr> sidor på svenska</label>
  <input id=cty type=radio name=meta value="cr=countrySE" >
    <label for=cty>sidor från Sverige</label>...
</form>
```

Queries

The query *Nugues* to Google is a sequence of pairs (name, value)

`http://www.google.com/search?source=ig&hl=fr&rlz=&q=nugues`

`URISplitter` extracts the query:

`source=ig&hl=fr&rlz=&q=nugues`

We can create a `GET` request using the URL constructor and send it to Google using `openStream()`
Google returns a 403 error: Forbidden. AltaVista is nicer.
//GoogleQuery.java
POST would send

```
Content-Type: application/x-www-form-urlencoded
Content-Length: 29
Connection: close

source=ig&hl=fr&rlz=&q=nugues
```

Faking the Lizard

A naïve Google query from Java fails miserably:

```
Server returned HTTP response code: 403 for URL:
source=ig&hl=fr&rlz=&q=nugues
```

Google sets constraints on the user agent.
It is possible to remedy this.
Just set a user agent corresponding to a known browser


```
uc.setRequestProperty("User-Agent", "Mozilla/5.0");
```

Using the POST Command

The two main commands of a HTTP client are GET and POST
GET sends parameters as an extension of a URL address
They are visible to everybody
How to send data with POST?
Programs FormPoster.java and QueryString.java from Elliotte Rusty Harold, *Java Network Programming*, pages 220–222 and page 153 are examples of it.
(http://www.cafeaulait.org/books/jnp3/examples/15/)
They form a client that works in conjunction with a server and formats a query.
The query is sent back by the server

Formatting a Query

QueryString.java encodes and formats a query.
The pairs of parameter names (keys) and values are separated with &
In the example, we send:

- Name: Elliotte Rusty Harold
- Email: elharo@metalab.unc.edu

Using the POST Command

Switching from GET to POST is done implicitly through setDoOutput()

```
URLConnection uc = url.openConnection();
uc.setDoOutput(true);
OutputStreamWriter out =
  new OutputStreamWriter(uc.getOutputStream(), "UTF-8");
out.write(query.toString());
out.write("\r\n");
out.flush();
out.close();
```

The client header is sent automatically

Using the POST Command (II)

HttpURLConnection is a subclass designed to carry out HTTP interaction
The POST method is more explicit with it and the setRequestMethod()

```
HttpURLConnection uc =
  (HttpURLConnection) url.openConnection();
uc.setRequestMethod("POST");
uc.setDoOutput(true);
```

(FormPoster2.java)

HttpURLConnection

Possible requests with `HttpURLConnection` are:

- `GET` // default download
- `POST`
- `PUT` // Upload a file
- `DELETE` // delete a file
- `HEAD` //same as `GET` but return the header only
- `OPTIONS` //lists the possible commands
- `TRACE` // send back the header

This makes provision to manage HTTP protocol codes directly at the API level.

DELETE

```
HttpURLConnection uc =
  (HttpURLConnection) url.openConnection();
uc.setRequestMethod("DELETE");
uc.setDoOutput(true);
OutputStreamWriter out =
  new OutputStreamWriter(uc.getOutputStream(), "UTF-8");

// The DELETE line, the Content-type header,
// and the Content-length headers are sent by the
// URLConnection.
// We just need to send the data
out.write("/none");
out.write("\r\n");
```

(Poster.java)

FTP

```
//URL url = new URL("ftp://username:password@ftp.whatever.com/

URL url =
new URL("ftp://anonymous:Pierre.Nugues%40cs.lth.se@ftp.sri.com
URLConnection uc = url.openConnection();
InputStream is = uc.getInputStream();
BufferedReader bReader =
  new BufferedReader(new InputStreamReader(is));
String line;
while ((line = bReader.readLine()) != null) {
    System.out.println(line);
}
```

//ReadURLftp.java

REST Architecture

REST – representation state transfer – An a posteriori model of the web: clients, servers, and HTTP

RESTful architecture implicitly means: the client-server transactions based on three standards:

- HTTP:
  - Transfer protocol of the web
  - On top of TCP/IP
  - Pairs of requests from clients and responses from servers
- URI/URLs:
  - A way to name and address objects on the net
- HTML/XML

REST Methods

Most web servers use databases to store data.
REST transactions are essentially database operations.
In the context of REST, we reuse HTTP methods with a different meaning.
This defines the interaction protocol or API.
CRUD is another name of the same concept.
The CRUD operations are mapped onto HTTP methods.

CRUD names	HTTP methods
Create	POST
Read (Retrieve)	GET
Update	PUT
Delete	DELETE

See: `http://en.wikipedia.org/wiki/Create,_read,_update_and_delete`.

REST Methods: POST

Doodle is a popular planning service for meetings: `http://doodle.com/`
Doodle uses a REST API:
`http://doodle.com/xsd1/RESTfulDoodle.pdf`
We create a meeting (poll in Doodle) with POST

Client		Server
`POST /polls`	$\longrightarrow$	
Data in the message body		
	$\longleftarrow$	201 Created
		Content-Location:lschgtq77kunavkcu

REST Methods: GET

We retrieve a meeting with GET

Client		Server
`GET /polls/Ischgtq`	$\longrightarrow$	
	$\longleftarrow$	The poll encoded in XML according to schema poll.xsd.

REST Methods: PUT

We update a meeting with PUT

Client		Server
`PUT /polls/Ischgtq`	$\longrightarrow$	
Data in the message body		
	$\longleftarrow$	200 OK

REST Methods: DELETE

We delete a meeting with DELETE

Client		Server
`DELETE /polls/Ischgtq`	$\longrightarrow$	
	$\longleftarrow$	204 No Content

HTTP Methods and REST

Amazon S3 is another example (from RESTful web services, Chap. 3, O'Reilly)
It uses two types of objects: buckets (a folder or a collection) and objects and four methods: GET, HEAD, PUT, and DELETE

	GET	HEAD	PUT	DELETE
Bucket	List content		Create bucket	Delete bucket
Object	Get data and metadata	Get metadata	Set values and metadata	Delete object

One example among others...

REST and Sesame

Sesame extends the REST protocol to manage graphs:

- GET: Fetches statements from the repository.
- PUT: Updates data in the repository, replacing any existing data with the supplied data. The data supplied with this request is expected to contain an RDF document in one of the supported RDF formats.
- DELETE: Deletes statements from the repository.
- POST: Performs updates on the data in the repository. The data supplied with this request is expected to contain either an RDF document or a special purpose transaction document. In case of the former, the statements found in the RDF document will be added to the repository. In case of the latter, the updates specified in the transaction document will be executed.

REST Examples

Get all repositories (tuple query):

```
curl -X GET -H "Accept: application/sparql-results+xml"
  http://asimov.ludat.lth.se/openrdf-sesame/repositories
```

Delete all statements in the repository:

```
curl -X DELETE
  http://asimov.ludat.lth.se/openrdf-sesame/repositories/sandb
```

SPARQL queries is also straightforward.
A SELECT query: `SELECT ?s ?p ?o WHERE {?s ?p ?o}` (tuple query):

```
curl -X GET -H "Accept: application/sparql-results+json"
http://asimov.ludat.lth.se/openrdf-sesame/repositories/sandbo
?query=SELECT+%3fs+%3fp+%3fo+WHERE+%7b%3fs+%3fp+%3fo%7d
```

REST In Practice

Few programmers would build a REST application from scratch. There are plenty of tools available:

- Reference:

  JSR 311, JAX-RS: The Java API for RESTful Web Services, http://jsr311.java.net/

- Tools:

  cURL, a command line tool, http://curl.haxx.se/

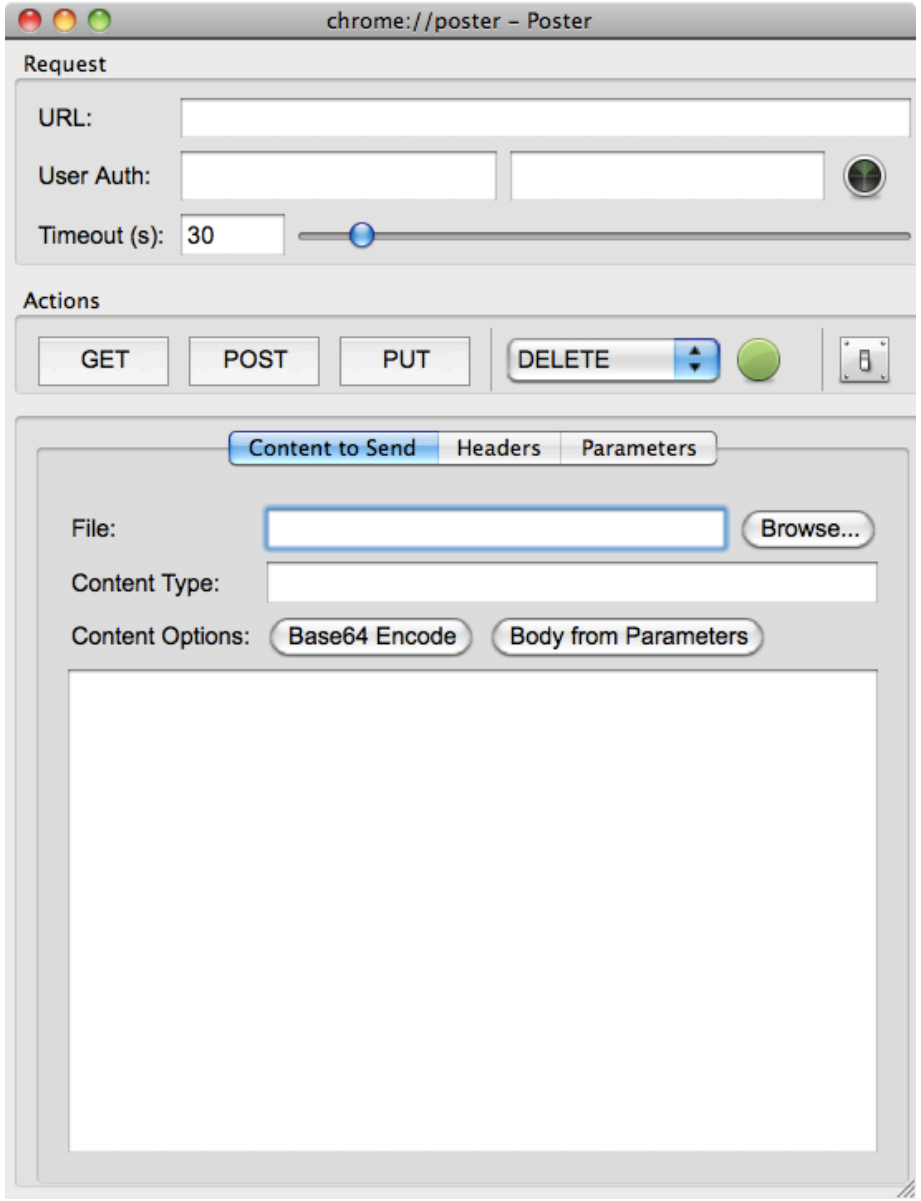
  Poster, a plugin module for Firefox, https://addons.mozilla.org/en-US/firefox/addon/poster/

  soapUI, http://www.soapui.org/

- Implementation:

  Jersey, http://jersey.java.net/

Poster


chrome://poster - Poster
Request
URL:
User Auth:
Timeout (s):
30
Actions
GET
POST
PUT
DELETE
Content to Send
Headers
Parameters
File:
Browse...
Content Type:
Content Options:
Base64 Encode
Body from Parameters