Technical University of Moldova

Faculty of Computers, Informatics and Microelectronics

Department of Software Engineering and Automation

Web Programming

Laboratory work #2

HTTP, caching and content negotiation

Author:
Liviu Mocanu
std. gr. FAF-203

Supervisor:
Alexei Șerșun

Chișinău 2023

1 Task

Our task for this laboratory work was to:

1. Write a command line program, using go2web executable as a starting point.
2. The program should implement at least the following CLI:
   - go2web -u <URL> # make an HTTP request to the specified URL and print the response
   - go2web -s <search-term> # make an HTTP request to search the term using your favorite search engine and print top 10 results
   - go2web -h # show this help
3. The responses from request should be human-readable (e.g. no HTML tags in the output)

We were not allowed to use any third-party libraries for making HTTP requests.

2 Results

In order to complete this laboratory work I used Java and the only third-party libraries besides core Java were *Jsoup* for parsing and presenting HTML as well as *JsonNode*, *ObjectMapper* and *SerializationFeature* for dealing with JSON when scrapping. I used *JSONArray* and *JSONObject* to handle the response when searching for a query in **Google** as the chosen search engine. I also used SSLSocket to establish the connection.

2.1 Executable

When ./go2web with argument -u is used like we see in Listing 1, the `Scrapper` from package utm is called from within the jar, with the *url* and *location* where the Java project is located as arguments, in order to be able to cash the result.

```
case $opt in
u)
  url=$OPTARG
  java -cp "$(pwd)/target/pw-2-1.0-SNAPSHOT-jar-with-dependencies.jar" utm.Scrapper
  "$url" "$(pwd)"
  ;;
```

Listing 1: Implementation of ./go2web -u command

When ./go2web with argument -s is used like we see in Listing 2, the `Searcher` from package utm is called from within the jar, with the only argument being the *query*. As previously mentioned, I used the Google search engine. `shift` and `query="$*"` were used to allow multiple words as prompt other than just one.

```
case $opt in
s)
  # Use shift to remove the '-s' option from the argument list
  shift
  # Join all the remaining arguments into a single string separated by spaces
  query="$*"
  java -cp "$(pwd)/target/pw-2-1.0-SNAPSHOT-jar-with-dependencies.jar" utm.Searcher
  "$query"
  ;;
```

Listing 2: Implementation of ./go2web -s command

The overall configuration of the go2web executable is showcased in Listing 3. Besides the help, I decided to add a message in case of an Invalid option or if the option is valid but no arguments were added even when required as demonstrated in [1].

```
#! /usr/bin/env bash

while getopts "u:s:h" opt; do
  case $opt in
  u)
    url=$OPTARG
    java -cp "$(pwd)/target/pw-2-1.0-SNAPSHOT-jar-with-dependencies.jar" utm.Scrapper "$url" "$(pwd)"
    ;;
  s)
    shift
    query="$*"
    java -cp "$(pwd)/target/pw-2-1.0-SNAPSHOT-jar-with-dependencies.jar" utm.Searcher "$query"
    ;;
  h)
    echo "go2web -u <URL>         # make an HTTP request to the specified URL and print the response"
    echo "go2web -s <search-term> # make an HTTP request to search the term using your favorite search engine and print top 10 results"
    echo "go2web -h               # show this help"
    exit 1
    ;;
  \?)
    echo "Invalid option: -$OPTARG" >&2
    exit 1
    ;;
  :)
    echo "Option -$OPTARG requires an argument." >&2
    exit 1
    ;;
  esac
done
```

Listing 3: The executable go2web

2.2 Scrapper

The `Scrapper` class has 4 important functions: `main`, `requestResponse`, `establishConnection` and the `getResponseBodyAsString` function as well as two other helper functions to get the response code and the content type header value.

Besides some validation calling the `requestResponse` function, `main` has actually the caching functionality as well.

`Main` first creates a **cache** directory by calling `Files.createDirectories()` with a `Path` object created from string concatenatenating `cacheDirectoryPath` and `"cache"`. It then creates a cache file by creating a File object within the cache directory, using the URL string with all *non-alphanumeric* characters removed as the filename.

If the cache file already exists and is a regular file (not a directory), the code reads the cached response from the file using `Files.readString()` after which requests a new response by calling the `requestResponse()` method with the URL string. If the new response is not null and is different from the cached response, the code updates the cache file by overwriting it with the new response using a `BufferedWriter`.

Otherwise, the code requests a new response using the `requestResponse()` method with the URL string. If the new response is not null, the code saves it to the cache file.

Then, the code either prints the cached response or the new response to the console, depending on whether a cached response was retrieved or a new response was requested. If there is an `IOException` while performing any of these operations, the code catches the exception, prints an error message to the console, and does not return any response as seens in Listing 5:

```
public static void main(String[] args) throws IOException {
    // ...
    String cacheDirectoryPath = args[1];

    File cacheDirectory = Files.createDirectories(Paths.get(cacheDirectoryPath, "cache")).toFile();
    File cacheFile = new File(cacheDirectory, urlString.replaceAll("[^a-zA-Z0-9]", "") + ".txt");

    // if request is already cached
    if (cacheFile.exists() && !cacheFile.isDirectory()) {
        String cachedResponse = Files.readString(cacheFile.toPath());
        String newResponse = requestResponse(urlString);
        if (newResponse != null && !newResponse.equals(cachedResponse)) {
            // update cache
            try (BufferedWriter writer = new BufferedWriter(new FileWriter(cacheFile))) {
                writer.write(newResponse);
            } catch (IOException e) {
                System.err.println("Error: " + e.getMessage());
            }
        }
        System.out.println(cachedResponse);
    } else {
        String newResponse = requestResponse(urlString);
        if (newResponse != null) {
            // save to cache
            try (var writer = new BufferedWriter(new FileWriter(cacheFile))) {
                writer.write(newResponse);
            } catch (IOException e) {
                System.err.println("Error: " + e.getMessage());
            }
            System.out.println(newResponse);
        }
```

```
    }
}
```

Listing 4: Scrapper main function

`requestResponse` takes urlString as an input parameter and returns a String representing the formatted response obtained from the SSLSocket connection.

The method works by first calling the `establishConnection` method to establish a SSLSocket connection to the URL specified by the urlString input parameter. It then retrieves the content type of the SSLSocket response using the `getHeader()` method of the Response object, and reads the response body into a String using the `getResponseBodyAsString` method using Response again.

Next, the method checks the content type of the HTTP response. It is exactly here that **content negotiation** was implemented:

If the content type is **"application/json"**, it uses the Jackson JSON library to parse the response body into a `JsonNode` object, formats the JSON using the `ObjectMapper` class, and converts it to a String using the `writeValueAsString` method of the ObjectMapper class.

If the content type is **"text/html"**, the method uses the `Jsoup` library to parse the response body into an HTML document and extract the text content using the `text()` method.

If the content type is not supported, the method prints an error message to the console.

The method closses the socket connection and returns the formatted response String. If there is an IOException while performing any of these operations, the method catches the exception, prints an error message to the console, and returns null as shown in Listing 5:

```
private static String requestResponse(String urlString) {
    try {
        SSLSocket socket = establishConnection(urlString);
        String contentType = getHeader(in, "Content-Type");
        String responseString = getResponseBodyAsString(in);
        String formattedResponse = null;

        if (contentType.contains("application/json")) {
            ObjectMapper objectMapper = new ObjectMapper();
            objectMapper.enable(SerializationFeature.INDENT_OUTPUT);
            JsonNode jsonNode = objectMapper.readTree(responseString);
            formattedResponse = objectMapper.writeValueAsString(jsonNode);
        } else if (contentType.contains("text/html")) {
            formattedResponse = Jsoup.parse(responseString).text();
        } else {
            System.out.println("Content type not supported.");
        }

        socket.close();
        return formattedResponse;
    } catch (IOException e) {
        System.err.println("Error: " + e.getMessage());
        return null;
    }
}
```

Listing 5: Scrapper requestResponse function

`establishConnection` takes a String `urlString` as an input parameter and returns an `SSLSocket` object representing the established connection.

The method works by first creating a new URL object from `urlString`. Then, it enters a loop where it opens a new `SSLSocket` using the `Response.establish()` method, passing the host and path of the URL object. It also retrieves the status code of the response using the `getResponseStatus()` helper method, which takes a `BufferedReader` object as an input.

It is exactly here that the function deals with **HTTP request redirects**. If the status code indicates that the server has issued a redirect (`HTTP_MOVED_TEMP`, `HTTP_MOVED_PERM`, `or HTTP_SEE_OTHER`), the method retrieves the URL for the redirect from the Location header field of the HTTP response using a custom `getHeader()` method, which takes a `BufferedReader` object and the header field name as inputs. It then closes the current socket and creates a new URL object from the redirect URL.

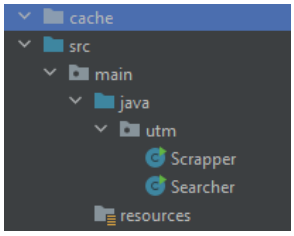
The loop continues until the status code does not indicate a redirect. The method then returns the `SSLSocket` object representing the established connection.

The method throws an `IOException` if there is an error establishing the connection as shown in Listing 6:

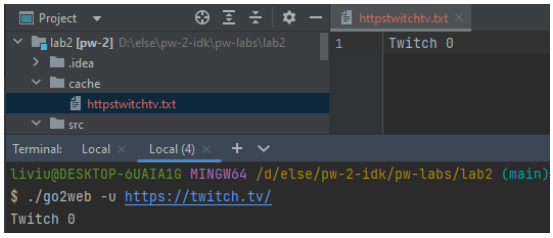
```
private static SSLSocket establishConnection(String urlString) throws IOException {
    URL url = new URL(urlString);
    SSLSocket socket;
    int status;
    do {
        socket = Response.establish(url.getHost(), url.getPath());

        in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
        status = getResponseStatus(in);
        if (status == HTTP_MOVED_TEMP || status == HTTP_MOVED_PERM || status ==
    HTTP_SEE_OTHER) {
            String redirectUrl = getHeader(in, "Location");
            socket.close();
            url = new URL(redirectUrl);
        }
    } while (status == HTTP_MOVED_TEMP || status == HTTP_MOVED_PERM || status ==
    HTTP_SEE_OTHER);

    return socket;
}
```

Listing 6: Scrapper establishConnection function

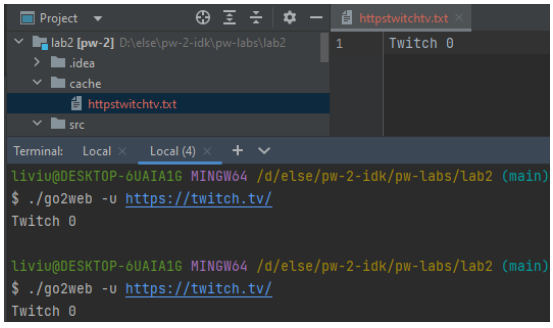
An example of using cache can be seen in Figure 1 by sending an HTTP request to *https://twitch.tv/*:



(a) No cache files before any requests



(b) Cache directory after a request



(c) Reusing cache

Figure 1: Cache directory after ./go2web -u https://twitch.tv/ command

Unfortunately in cases such as with http://worldtimeapi.org/api/ip/, cache won't help too much since every request will be different as seen in Figure 2:

```
liviu@DESKTOP-6UAIA1G MINGW64 /d/else/pw-2-idk/pw-labs/lab2 (main)
$ ./go2web -u http://worldtimeapi.org/api/ip/
{
  "day_of_year" : 79,
  "dst" : false,
  "dst_from" : null,
  "dst_offset" : 0,
  "dst_until" : null,
  "raw_offset" : 7200,
  "timezone" : "Europe/Chisinau",
  "unixtime" : 1679278447,
  "utc_datetime" : "2023-03-20T02:14:07.919038+00:00",
  "utc_offset" : "+02:00",
  "week_number" : 12
}

liviu@DESKTOP-6UAIA1G MINGW64 /d/else/pw-2-idk/pw-labs/lab2 (main)
$ ./go2web -u http://worldtimeapi.org/api/ip/
{
  "abbreviation" : "EET",
  "client_ip" : "89.149.111.34",
  "datetime" : "2023-03-20T04:14:25.574334+02:00",
  "day_of_week" : 1,
  "day_of_year" : 79,
  "dst" : false,
  "dst_from" : null,
  "dst_offset" : 0,
  "dst_until" : null,
  "raw_offset" : 7200,
  "timezone" : "Europe/Chisinau",
  "unixtime" : 1679278465,
  "utc_datetime" : "2023-03-20T02:14:25.574334+00:00",
  "utc_offset" : "+02:00",
  "week_number" : 12
}
```

Figure 2: Cache will change if the request is different

However before moving on to the Searcher, we have to look into the helper class Response that we already used several times in the Scrapper, as it is essentially responsible for the connection and retrieval of information.

2.3 Response

The `Response` class contains several utility methods for working with HTTPS connections. The main functionality includes establishing an SSL connection, parsing the status of an HTTP response, extracting specific headers from the response, and retrieving the response body as a string.

establish(String host, String path): This method takes a `host` and a `path` as input parameters and returns an `SSLSocket` object representing the established SSL connection. It starts by creating an `SSLSocketFactory` and an `SSLSocket` for the given host and port 443 (HTTPS). It then initiates the SSL handshake and sends an HTTP GET request to the server using the specified path. The method returns the `SSLSocket` object as seen in Listing 7:

```
public static SSLSocket establish(String host, String path) throws IOException {
    SSLSocketFactory factory = (SSLSocketFactory) SSLSocketFactory.getDefault();
    SSLSocket socket = (SSLSocket) factory.createSocket(host, 443);
    socket.startHandshake();

    OutputStream os = socket.getOutputStream();
    String request = "GET " + path + " HTTP/1.1\r\n" +
            "Host: " + host + "\r\n" +
            "Connection: close\r\n\r\n";
    os.write(request.getBytes());
    os.flush();

    return socket;
}
```

Listing 7: Response establish function

getResponseStatus(BufferedReader in): This method takes a `BufferedReader` object as an input parameter and returns the status code of the HTTP response as an integer. It reads the status line from the input, splits it into tokens, and then returns the second token, which is the status code. The code for the function is found in Listing 8.

```
public static int getResponseStatus(BufferedReader in) throws IOException {
    String statusLine = in.readLine();
    String[] statusTokens = statusLine.split(" ");
    return Integer.parseInt(statusTokens[1]);
}
```

Listing 8: Response getResponseStatus function

getHeader(BufferedReader in, String headerName): This method takes a `BufferedReader` object and a `headerName` as input parameters and returns the value of the specified header field in the HTTP response. It reads lines from the input until it finds a line that starts with the specified header name. When found, it extracts and returns the value of the header field. If the header is not found, the method returns `null` as can be observed in Listing 9.

```
public static String getHeader(BufferedReader in, String headerName) throws
    IOException {
    String headerLine;
    while ((headerLine = in.readLine()) != null) {
        if (headerLine.toLowerCase().startsWith(headerName.toLowerCase() + ":")) {
            return headerLine.substring(headerLine.indexOf(":") + 1).trim();
        }
    }
    return null;
}
```

Listing 9: Response getHeader function

getResponseBodyAsString(BufferedReader in): This method takes a `BufferedReader` object as an input parameter and returns the HTTP response body as a string. It reads and discards

the headers from the input, then reads the response body line by line, appending each line to a `StringBuilder`. The method returns the resulting string as can be seen in Listing 10.

```
public static String getResponseBodyAsString(BufferedReader in) throws IOException {
    StringBuilder responseBuilder = new StringBuilder();
    String inputLine;

    // Read and discard the headers
    while ((inputLine = in.readLine()) != null) {
        if (inputLine.trim().isEmpty()) {
            inputLine = in.readLine();
            if (inputLine.startsWith("{")) {
                responseBuilder.append(inputLine);
            }
            break;
        }
    }

    // Read the response body
    while ((inputLine = in.readLine()) != null) {
        responseBuilder.append(inputLine);
        responseBuilder.append("\n");
    }

    return responseBuilder.toString();
}
```

Listing 10: Response getResponseBodyAsString function

These methods can be used together to establish a connection to a server, read the response status, extract specific headers, and retrieve the response body, whilst entirely based on SSL Sockets.

2.4 Searcher

Moving onto the `Searcher` class, we have 2 other important functions: `main`, `getResults`, and 2 helper functions, `getResponseCode` and `getResponseBodyAsString` function.

The function `main` performs a Google search using the Google Custom Search API and prints the top `MAX_RESULTS` (In our case 10) first results.

For that however, we had to set up our own `API_KEY` and `CX` by going to Google Developers Console and setting up a new Custom Search API project. In my case they can be seen in Listing 11:

```
private static final String API_KEY = "AIzaSyBKAkng_5Qu43Yg4aBlvEHKb-A2V6l2UBI";
private static final String CX = "75e13f83e334443b6";
```

Listing 11: Setting up a new Custom Search API project

Firstly, we encode the query passed as an argument using `URLEncoder.encode()` and the `UTF_8` charset.

Then we create a URI string using `String.format()`, with placeholders for the `GOOGLE_URL` constant, `API_KEY` and `CX`, and query variables. We create a URI object using the resulting string and establish a `SSLSocket` connection by calling the `Response.establish` method with the host and the path with the query of the created URI.

Afterwards, we create a `BufferedReader` to read from the socket's input stream and get the response code using the `getResponseCode` method.

If the response code is `HTTP_OK` (200), the code reads the response body using

`getResponseBodyAsString()` and extracts the search results using `getResults()` after which printing the search query and the top `MAX_RESULTS` first results.

Otherwise, if the response code is not `HTTP_OK`, the code prints an error message to the console indicating that the search results could not be retrieved.

The code then closes the `SSLSocket`. If any exception occurs, we print the stack trace to the console.

```
public static void main(String[] args) {
    // first verify whether the Searcher is called correctly
    if (args.length < 1) {
        System.err.printf("Searcher expectations not met:%n" +
                "Provide only the query%n");
        System.exit(2);
    }
    System.out.println(args[0]);

    try {
        String query = URLEncoder.encode(args[0], StandardCharsets.UTF_8);
        URI uri = URI.create(String.format("%s?key=%s&cx=%s&q=%s", GOOGLE_URL,
    API_KEY, CX, query));

        SSLSocket socket = Response.establish(uri.getHost(), uri.getPath() + "?" +
    uri.getQuery());

        BufferedReader in = new BufferedReader(new InputStreamReader(socket.
    getInputStream()));
        int responseCode = getResponseCode(in);
        if (responseCode == HTTP_OK) {
            String response = getResponseBodyAsString(in);
            List<String> searchResults = getResults(response);
            System.out.println("'" + args[0] + "' provided the following results:");
            for (int i = 0; i < searchResults.size(); i++) {
                String result = searchResults.get(i);
                System.out.println((i + 1) + ". " + result);
            }
        } else {
```

```
            System.err.println("Failed to get search results. Response code: " +
    responseCode);
        }

        socket.close();
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

Listing 12: Searcher `main` function

`getResults` takes a string responseString as input and returns a list of strings which represent the results to the query by:

- Creating a new `JSONObject` by parsing the responseString parameter.
- Retrieving the items JSON array from the `JSONObject`.
- Using `IntStream.range` to create a stream of integers from 0 to `MAX_RESULTS` and then `mapToObj` to map each integer to the corresponding `JSONObject` in the items array.
- Using map to convert each `JSONObject` to a string that concatenates the values of the *title* and *link* fields, separated by a hyphen.
- Collecting the resulting strings into a list and returning that list.

Thus we extract the title and link fields from the items array of a JSON object and return them as a list of strings. The entirety of the proccess takes place in a try-catch block, such that in case of not finding any results, we print the cause of the error/lack of results as seen in Listing 13.

```
private static List<String> getResults(String responseString) {
    try {
        JSONObject json = new JSONObject(responseString);
        JSONArray items = json.getJSONArray("items");

        return IntStream.range(0, Math.min(items.length(), MAX_RESULTS))
                .mapToObj(items::getJSONObject)
                .map(item -> item.getString("title") + " - " + item.getString("link")
    )
                .collect(Collectors.toList());
    } catch (JSONException e) {
        System.err.println("Not enough results found.");
        return null;
    }
}
```

Listing 13: Searcher `getResults` function

The `Searcher` can be seen in action in Figure 3, where we can observe that not only we could do a query with more than one word, but the top 10 links provided are also clickable.

As mentioned the other two helper functions the `Searcher` is using are simple calls to the `Response` class, similar to how the `Scrapper` is calling it.

3 Conclusion

This laboratory work for Web Programming involved building a Command Line program that made HTTP requests to specific URLs, including search engines, and printed the responses in a human-readable format.

The project also included implementing advanced features, such as :

```
$ ./go2web -s what is pw
what is pw
'what is pw' provided the following results:
1. P.W. - Urban Dictionary - https://www.urbandictionary.com/define.php?term=P.W.
2. I am a faculty member. A student withdrew from my course and the ... - https://gsu.my.site.com/support/s/article/I-am-a-fac
x-on-PAWS-is-PW-What-is-PW
3. P.W. | English meaning - Cambridge Dictionary - https://dictionary.cambridge.org/dictionary/english/pw
4. Pw definition and meaning | Collins English Dictionary - https://www.collinsdictionary.com/us/dictionary/english/pw
5. DOR Pass-Through Entity Withholding - https://www.revenue.wi.gov/Pages/OnlineServices/pw-home.aspx
6. PW Pizza - https://pwpizza.com/
7. Instructions for 2019 Form PW-1 - https://www.revenue.wi.gov/TaxForms2017through2019/2019-FormPW-1-Inst.pdf
8. Pw - What is pw short for? - https://slang.net/meaning/pw
9. Home - P.W. Grosser Consulting, Inc. - https://pwgrosser.com/
10. PAPBS > Program wide PBIS - https://papbs.org/Program-wide-PBIS
```

Figure 3: Top 10 search results for prompt 'what is pw' using Google

- **HTTP request redirects**, which are important for ensuring that the program can follow links and request the correct resources.
- **HTTP cache mechanism**, which can help reduce network traffic and improve performance.
- **Content negotiation**, which is important for building modern web applications that can work with various types of data.

By completing this laboratory work, I gained valuable experience in working with HTTP requests and responses, as well as got hands-on experience on advanced web development concepts that are essential for developing advanced web-based applications.

References

[1] Small getopts tutorial, *https://wiki.bash-hackers.org/howto/getopts_tutorial*
Accessed on March 4, 2023.

[2] Do a Simple HTTP Request in Java, *https://www.baeldung.com/java-http-request*
Accessed on March 4, 2023.

[3] How to read write this in utf-8?, *https://stackoverflow.com/questions/13350676/how-to-read-write-this-in-utf-8*
Accessed on March 5, 2023 .

[4] How To Get Google Api Key & Search Engine ID Google CSE, *https://www.youtube.com/watch?v=Bxy8Yqp5XX0*
Accessed on March 5, 2023 .

[5] Introduction to SSL in Java, *https://www.baeldung.com/java-ssl*
Accessed on March 18, 2023.