Batch and Stream Processing: Realtime Analysis of Big Data

Marcel Stolin
marcelpascal.stolin@studenti.unitn.it

Abstract

Since the beginning of Big Data, batch processing was the most popular choice for processing large amounts of generated data. These existing processing technologies are not suitable to process the large amount of data we face today. Research works developed a variety of technologies that focus on stream processing. Stream processing technologies bring significant performance improvements and new opportunities to handle Big Data. In this paper, we discuss the differences of batch and stream processing and we explore existing batch and stream processing technologies. We also explain the new possibilities that stream processing make possible.

1 Part 1

1.1 Introduction

In the second lecture of the course, an implementation of a basic HTTP server with the name *TinyHttpd* was introduced. The functionality of *TinyHttpd* is limited to opening `.html` files and delivering the content via the HTTP 1.1 protocol to the client.

The task of part 1 of the first assignment, is to extend the *TinyHttpd* implementation by implementing the functionality to launch an external process. Therefore, the client sends a request via the URL `http://localhost:8000/process/reverse?par1=ROMA`. Then, the server executes an external Java process, which reverses the string `ROMA`, given in the query `?par1=ROMA`, and responses the result (`AMOR`) of the external Java process to the client via HTTP 1.1.

Given the above mentioned task description, the following steps have to be implemented:

1. Create a Java application, called *StringReverser*, which takes a valid String as input and returns the reversed string
2. Extend the *TinyHttpd* implementation to launch external processes when requested by client via the URL `http://localhost:8000/process/PROCESS_NAME?PROCESS_PARAMETERS`

3. Execute the requested process on the server and deliver the process output to the client via the HTTP protocol.

1.2 Conceptual Design

Given the problem statement introduced in Section 1.1, a new application called *StringReverser* needs to be implemented, and the *TinyHttpd* server has to be extended in a way to launch an external Java process.

1.2.1 StringReverser

The *StringReverser* application is a simple terminal application. It takes any valid String as an input and returns the reversed String as the output. It can be executed via the console. For example the command `$ java -jar StringReverser.jar ROMA` should responses the string *AMOR*.

1.2.2 TinyHttpd

Whenever the client makes a request via the URL `http://localhost:8000/process/PROCESS_NAME?PROCESS_P` the server is supposed to start an external Java process, waits for the output of the process, and responses the process output to the user. The client has the possibilities to specify which process has to be executed. Given the URL `http://localhost:8000/process/reverse?par1=ROMA`, the user explicitly requests to launch the *reverse* process with the given query `par1=ROMA` as the process input. It is important to mention, that each process takes individual parameters as input. For the above mentioned `StringReverser`, only the value of the first parameter in the given query is important. All other parameters, and the parameter key, are therefore ignored.

1.3 Implementation

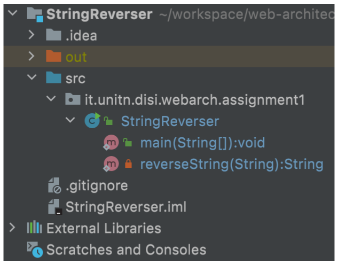
To implement the given conceptual design mentioned in Section 1.2, the OpenJDK 17[1] is used.

1.3.1 StringReverser

The application *StringReverser* is a simple Java project. It is composed of a single Java class called `StringReverser` as shown in Figure 1. The source code is shown in Listing 1, and it consists of a `main` method and a method called `reverseString`. The main method will be executed, when the application is launched via the terminal. Additionally, it checks if the given input string is valid. Otherwise, it will return an error message and exits with system code 0. If the input is a valid string, it will call the `reverseString` method, and returns

[1] JDK 17 - `https://openjdk.java.net/projects/jdk/17/` (Accessed: 02/10/2021)

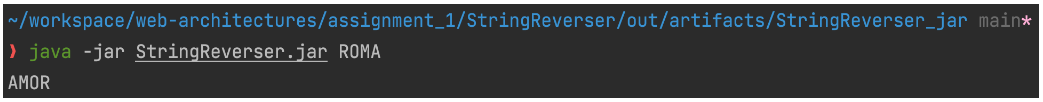
the result as the output. The `reverseMethod` is responsible to reverse the given input. To achieve this, it uses the `StringBuilder`[2] class to reverse the String.



```
StringReverser ~/workspace/web-architec
  .idea
  out
  src
    it.unitn.disi.webarch.assignment1
      StringReverser
        main(String[]):void
        reverseString(String):String
  .gitignore
  StringReverser.iml
External Libraries
Scratches and Consoles
```

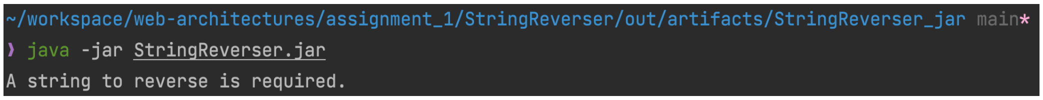
Figure 1: Project structure of the `StringReverser` application

Figure 2 shows the execution of the *StringReverser* application. The `.jar` artifact was created using the IntelliJ IDEA[3]. If the input is invalid, the `StringReverser` will print an error message to the terminal and exits with system code 0, as shown in Figure 3.



```
~/workspace/web-architectures/assignment_1/StringReverser/out/artifacts/StringReverser_jar main*
❯ java -jar StringReverser.jar ROMA
AMOR
```

Figure 2: Successful execution of the *StringReverser* application



```
~/workspace/web-architectures/assignment_1/StringReverser/out/artifacts/StringReverser_jar main*
❯ java -jar StringReverser.jar
A string to reverse is required.
```

Figure 3: Execution of the *StringReverser* application without an input

1.3.2 TinyHttpd

The foundation of the *TinyHttpd* project is provided by the Professor. It has to be extended to launch an external Java process, and deliver the process output to the client as a HTTP response. To implement the conceptual design mentioned in Section 1.2.2, the *TinyHttpd* project has to be extended in the following way:

1. Parse the HTTP request to check if a process has been requested

[2] StringBuilder - `https://docs.oracle.com/javase/7/docs/api/java/lang/StringBuilder.html` (Accessed: 02/10/2021)

[3] Create your first Java application - `https://www.jetbrains.com/help/idea/creating-and-running-your-first-java-application.html` (Accessed: 02/10/2021)

2. Generate the command to execute the requested process
3. Launch an external process, using the generated command
4. Response the process output to the client via HTTP

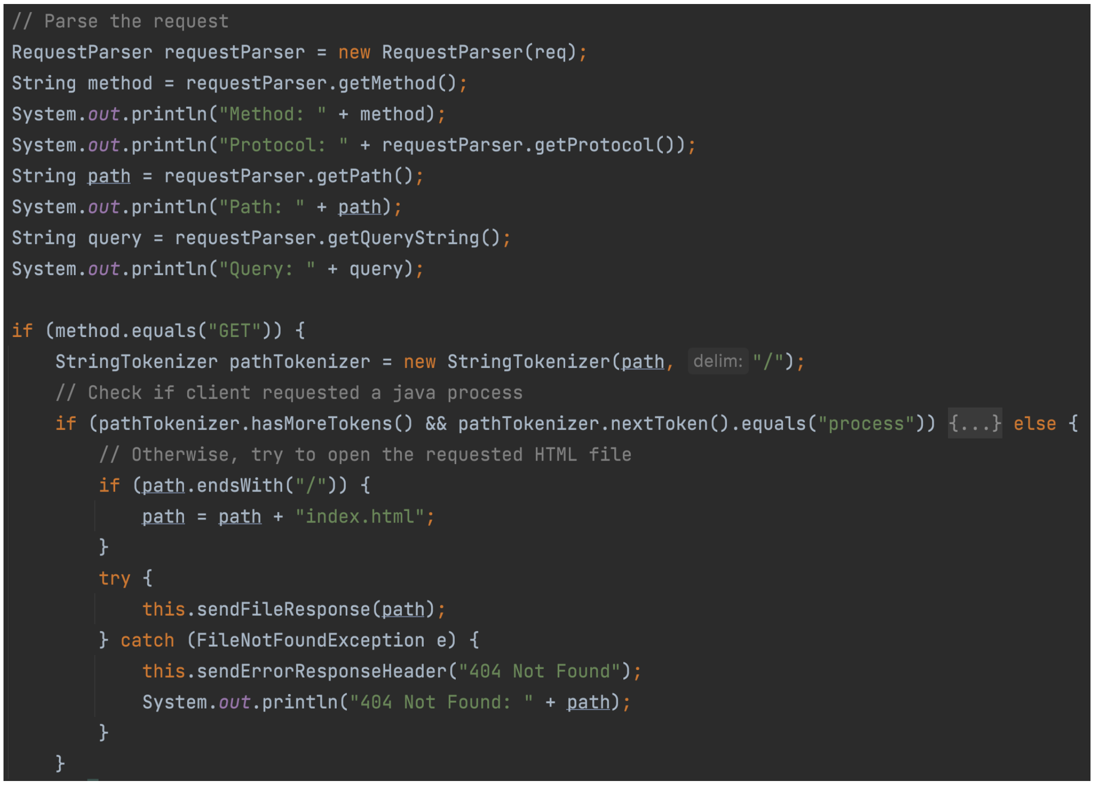
In the following, the single steps are described in detail.

Step 1: The first step of the implementation, is to check if the client requested the execution of a process. Therefore, the HTTP request has to be parsed accordingly. For this task, a new Java class called `RequestParser` is introduced to the *TinyHttpd* project. The `RequestParser` class is able to parse a HTTP request into its parts. For example, the request `GET /process/reverse?param=roma HTTP/1.1` is composed of the HTTP method (`GET`), the path (`/process/reverse?param=roma`), and the HTTP protocol version (`HTTP/1.1`). Furthermore, the path has an additional query attached (?param=roma). The implementation of the `RequestParser` class is attached at Listing 2.

After the `RequestParser` has successfully parsed the clients request, it is possible to check, by using an `if` statement, if the user requested the path */request/PROCESS_NAME*. If yes, the server executes the requested process, and sends a response accordingly. Otherwise, if the client has not requested to perform a Java process, the server tries to open the HTML file according to the given path and responses the HTML content to the client. The implementation of this procedure is shown in Figure 4.

Step 2: If the client has send a valid request for a process, the next step is to generate the command to launch the requested process. To keep the *TinyHttpd* implementation extensible, a new class called `CommandFactory` is added to the project, which is implemented using the Factory pattern. The implementation of this class is attached at Listing 3. The `CommandFactory` class is responsible to generate the command for the given request. It has a public static method called `generateCommand`, which takes the requested process name and the query of the request path as arguments. According to the given process name, it generates the command to execute the `.jar` artifact with the given query as input parameter. As example for the given path `/process/reverse?param=roma`, the generated command is `java -jar /Users/marcel/workspace/web-architectures/assignment_1/MiniHTTPD/jars/StringReverser.jar "roma"`. In addition, the `CommandFactory` is also responsible for checking, if the requested process is available and if the given query is a valid parameter for the process. If not, it will throw an exception.

Step 3: After the command has been generated, it needs to be executed in a shell. To achieve this, the `TinyHttpd` class is extended with a new new method called `launchProcess`. The implementation of this method is available at Listing 4. This method takes the generated command as an argument and executes

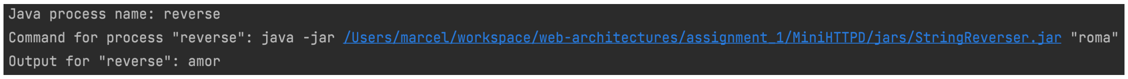


```
// Parse the request
RequestParser requestParser = new RequestParser(req);
String method = requestParser.getMethod();
System.out.println("Method: " + method);
System.out.println("Protocol: " + requestParser.getProtocol());
String path = requestParser.getPath();
System.out.println("Path: " + path);
String query = requestParser.getQueryString();
System.out.println("Query: " + query);

if (method.equals("GET")) {
    StringTokenizer pathTokenizer = new StringTokenizer(path,  delim: "/");
    // Check if client requested a java process
    if (pathTokenizer.hasMoreTokens() && pathTokenizer.nextToken().equals("process")) {...} else {
        // Otherwise, try to open the requested HTML file
        if (path.endsWith("/")) {
            path = path + "index.html";
        }
        try {
            this.sendFileResponse(path);
        } catch (FileNotFoundException e) {
            this.sendErrorResponseHeader("404 Not Found");
            System.out.println("404 Not Found: " + path);
        }
    }
```

Figure 4: Execution of the *StringReverser* application without an input

it using the `ProcessBuilder`[4] class. The method executes the given command in the bash shell using `bash -c COMMAND` and returns the process output as a string. It is important, that the process output is returned as a string, instead of printing it directly to the HTTP output stream. The reason is, that the HTTP header of the server response needs the length of the process output. Figure 6 shows the execution of the `launchProcess` method and Figure 5 shows the output to the console, after the process has been executed successfully.

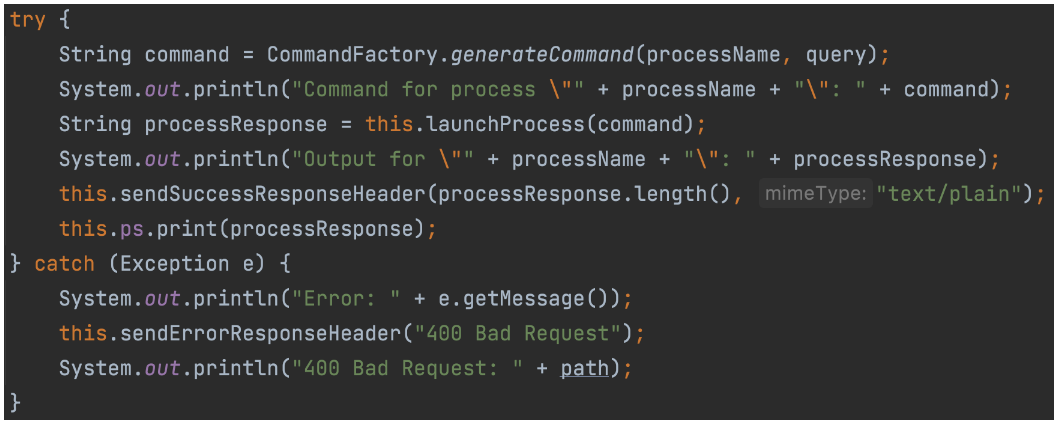


```
Java process name: reverse
Command for process "reverse": java -jar /Users/marcel/workspace/web-architectures/assignment_1/MiniHTTPD/jars/StringReverser.jar "roma"
Output for "reverse": amor
```

Figure 5: Console output after the generated command has been launched successfully

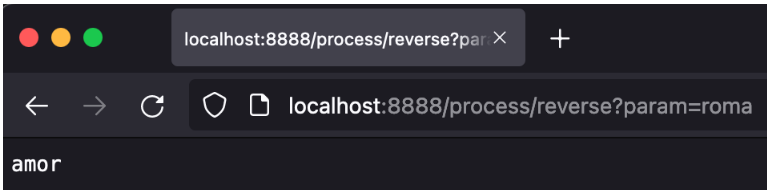
[4] ProcessBuilder - `https://docs.oracle.com/javase/7/docs/api/java/lang/ProcessBuilder.html` (Accessed: 02/10/2021)

Step 4: The last step, is to response the process output as a HTTP response to the client. A method called `sendSuccessResponseHeader` is added to the *TinyHttpd* class, which is responsible to send a `200 OK` HTTP response. Additionally, the method takes the length of the output as an argument, which was mentioned before, as well as the MIME type of the response content. The implementation of the `sendSuccessResponseHeader` is attached at Listing 5. Figure 6 shows the whole process of generating the process command, launching the command, and sending the reponse to the client. First a header with the process output length and the MIME type `text/plain` is written to the output stream, then the process output. Figure 7 shows the response in the browser for a successful process request.



```
try {
    String command = CommandFactory.generateCommand(processName, query);
    System.out.println("Command for process \"" + processName + "\": " + command);
    String processResponse = this.launchProcess(command);
    System.out.println("Output for \"" + processName + "\": " + processResponse);
    this.sendSuccessResponseHeader(processResponse.length(), mimeType: "text/plain");
    this.ps.print(processResponse);
} catch (Exception e) {
    System.out.println("Error: " + e.getMessage());
    this.sendErrorResponseHeader("400 Bad Request");
    System.out.println("400 Bad Request: " + path);
}
```

Figure 6: Implementation of the steps 2 to 4



localhost:8888/process/reverse?par
localhost:8888/process/reverse?param=roma

```
amor
```

Figure 7: A successful process request

1.4 Conclusion

2 Part 2

2.1 Introduction

In addition to the *TinyHttpd*, dynamic pages have been introduced in the lecture. In detail, the Common Gateway Interface (CGI) has been introduced, which is able to spawn processes on the server, and send the result as a HTTP response. The task of this part of the assignment, is to create a script, that performs the *StringReverser* application, as introduced in Section 1.2.1, and responses the result via HTTP. This task is almost equally to the first task (introduced in Section 1.1), but rather than extending an existing Java project, a script has to be created which will be launched within the `cgi-bin` folder of the Apache Web Server. For this task, the Apache Web Server is an requirement and the version XAMPP 7.4.23 for Mac is being used.

2.2 Conceptual Design

Given the previously introduced problem statement in Section 2.1, a script will be implemented using bash. Therefore, the environment variables can be used, to read details about the HTTP request. Overall, the script has the same requirements, as the modification of the first task. However, the client can send a request via the url `http://localhost:80/cgi-bin/run_reverse_process.sh?param=ROMA`. Therefore, the use case is limited to the reverse process.

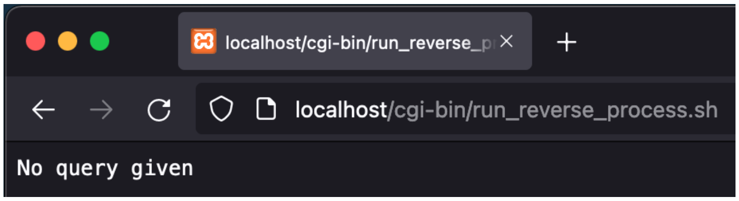
2.3 Implementation

The implementation of the script is available at Listing 6. The script is implemented in accordance to the following steps:

1. Check if the request was send via GET
2. Verify if a query is given
3. Validate the given query
4. If the given query is valid, perform the *StringReverser* Java process and send the process output to the client via HTTP

A simple if statement is used to check if the request was send via the GET method, otherwise an error message is sent. The request method is available via the environment variable `$REQUEST_METHOD`.

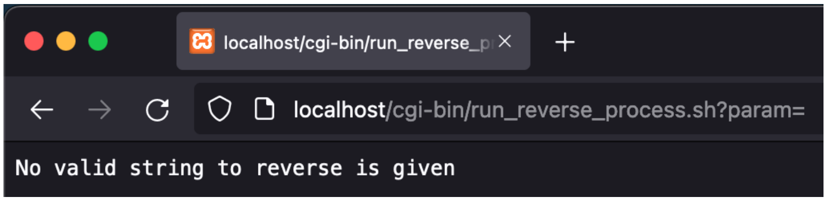
In the next step, the query has to be validated, which is saved in the environment variable `$QUERY_STRING`. If `$QUERY_STRING` is either not set or empty, an error message is sent by the script, which is illustrated in Figure 8.


localhost/cgi-bin/run_reverse_p
localhost/cgi-bin/run_reverse_process.sh

```
No query given
```

Figure 8: The error message if no query is provided

After the query string has been validated accordingly, the value of the first parameter has to be extracted. This is done by splitting the query string into its single parts. Next, the script checks if the value of the first parameter is valid. If not, an error message is sent to client, as illustrated in Figure 9.

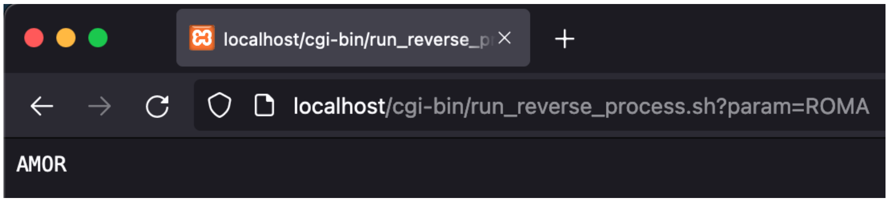


localhost/cgi-bin/run_reverse_process.sh?param=

```
No valid string to reverse is given
```

Figure 9: The error message if the query is invalid

The last step, is to execute the *StringReverser* application by executing it via the command `java -jar ARTIFACT STRING_TO_REVERSE`. Therefore, the generated `.jar` artifact is placed in the `cgi-bin/` directory of the Apache installation. After the process has been executed successfully, the output is send to the client as a HTTP response with the MIME type `text/plain`. This is shown in Figure 10.



localhost/cgi-bin/run_reverse_process.sh?param=ROMA

```
AMOR
```

Figure 10: A successful process request using the cgi-bin script

2.4 Conclusion

Was war einfacher? PArt1 oder Part 2

A Part 1

A.1 Implementation

A.1.1 StringReverser

Listing 1: CLI command to start a GitLab runner in a Docker container

```
package it.unitn.disi.webarch.assignment1;

public class StringReverser {

    public static void main(String[] args) {
        if (args.length >= 1) {
            String text = args[0];

            if (text != null) {
                String reversedString = reverseString(text);
                System.out.println(reversedString);
            }
        } else {
            System.out.println(”A␣string␣to␣reverse␣is␣required.”);
            System.exit(0);
        }
    }

    /**
     * This method reverses the given string.
     *
     * @param text
     * @return Reversed text
     */
    private static String reverseString(String text) {
        StringBuilder stringBuilder = new StringBuilder(text);
        stringBuilder.reverse();
        return stringBuilder.toString();
    }

}
```

A.1.2 TinyHttpd

Listing 2: CLI command to start a GitLab runner in a Docker container

```
package it.unitn.disi.webarch.tinyhttpd.utils;

import java.util.StringTokenizer;
```

```java
public class RequestParser {

    private String request;
    private String method;
    private String protocol;
    private String fullPath;
    private String path;
    private String queryString;

    /**
     * The RequestParser is able to parse an HTTP request
     * into its single parts.
     *
     * @param request
     */
    public RequestParser(String request) throws Exception {
        this.request = request;
        this.parseRequest();
        this.parsePath();
    }

    private void parseRequest() throws Exception {
        StringTokenizer tokenizer = new StringTokenizer(this.request);
        if (tokenizer.countTokens() < 3) {
            throw new Exception("The request \"" + this.request + "\" is invalid
        }

        this.method = tokenizer.nextToken();
        this.fullPath = tokenizer.nextToken();
        this.protocol = tokenizer.nextToken();
    }

    private void parsePath() {
        StringTokenizer tokenizer = new StringTokenizer(this.fullPath, "?");

        this.path = tokenizer.nextToken();
        if (tokenizer.hasMoreTokens()) {
            this.queryString = tokenizer.nextToken();
        }
    }

    public String getMethod() {
        return this.method;
    }
    public String getProtocol() {
        return this.protocol;
    }

    public String getPath() {
        return this.path;
    }

    public String getQueryString() {
        return this.queryString;
    }
}
```

Listing 3: CLI command to start a GitLab runner in a Docker container

```
package it.unitn.disi.webarch.tinyhttpd.utils;

import java.util.StringTokenizer;

public class CommandFactory {

    /**
     * Generates the command for the given process name.
     *
     * @param process Name of the requested process
     * @param parameters Probably the query string
     * @return Command as string
     * @throws Exception If the process name is unknown
     */
    public static String generateCommand(String process, String parameters) thro
        if (process.equals("reverse")) {
            return generateReverseProcessCommand(parameters);
        } else {
            throw new Exception("No␣process␣called␣\"" + process + "\"␣available
        }
    }

    private static String generateReverseProcessCommand(String parameters) throw
        StringTokenizer paramTokenizer = new StringTokenizer(parameters, "&");
        // first check if there are any parameters
        if (!paramTokenizer.hasMoreTokens()) {
            throw new Exception("No␣parameters␣given.␣The␣reverse␣process␣needs␣
        }
        String firstParam = paramTokenizer.nextToken();
        String[] paramKeyValue = firstParam.split("=");
        if (paramKeyValue.length >= 2) {
            // We need at least 2 elements
            String textToReverse = paramKeyValue[1];
            String artifactPath = System.getProperty("user.dir") + "/jars/String
            String command = "java" + "␣-jar␣" + artifactPath + "␣\"" + textToRe

            return command;
        } else {
            throw new Exception("No␣valid␣parameter␣input␣given␣for␣parameters␣\
        }
    }

}
```

Listing 4: CLI command to start a GitLab runner in a Docker container

```
private String launchProcess(String command) throws Exception {
    ProcessBuilder processBuilder = new ProcessBuilder();
    processBuilder.command("bash", "-c", command);
    Process process = processBuilder.start();

    StringBuilder output = new StringBuilder();
    BufferedReader reader = new BufferedReader(
            new InputStreamReader(process.getInputStream()));

    String line;
    while ((line = reader.readLine()) != null) {
        output.append(line + "\n");
    }

    return output.toString();
}
```

Listing 5: CLI command to start a GitLab runner in a Docker container

```
private void sendSuccessResponseHeader(int responseLength, String mimeType) {
    ps.print("HTTP/1.1␣200␣OK\r\n");
    ps.print("Content-Length:␣" + responseLength + "\r\n");
    ps.print("Content-Type:␣" + mimeType + "\r\n");
    ps.print("\r\n");
}
```

B Part 2

B.1 Implementation

Listing 6: CLI command to start a GitLab runner in a Docker container

```
#!/bin/sh
```

```
send_response () {
  echo "Content-type:␣text/plain;␣charset=iso-8859-1"
  echo "Content-Length:␣${#1}"
  echo
  echo $1
}


# Check if the request method is GET
if [ $REQUEST_METHOD == "GET" ]; then
  # Check if query is given
  if [ ! $QUERY_STRING ]; then
    send_response "No␣query␣given"
  else
    # Parse query
    saveIFS=$IFS
    IFS='=&'
    params=($QUERY_STRING)
    IFS=$saveIFS
    STRING_TO_REVERSE=${params[1]}

    # Check if the given parameter is valid
    if [ ! $STRING_TO_REVERSE ]; then
      send_response "No␣valid␣string␣to␣reverse␣is␣given"
    else
      REVERSE_JAVA_ARTIFACT="$(pwd)/StringReverser.jar"
      REVERSED_STRING=$(java -jar $REVERSE_JAVA_ARTIFACT $STRING_TO_REVERSE)

      send_response $REVERSED_STRING
    fi
  fi
else
  send_response "Only␣GET␣requests␣are␣allowed"
fi
```