TLM2008 LAB 2 - A SIMPLE WEB SERVER

In this lab, we will develop a simple web server which process GET/POST request and send back the requested content if available. By the end of the lab, you should be able to demonstrate that your Web Server is capable of delivering your home page to a web browser.

1 BEFORE YOU START

We are going to implement version 1.0 of HTTP, as defined in RFC 1945, where separate HTTP requests are sent for each component of the Web page. The server will be able to handle multiple simultaneous service requests in parallel. This means that the Web server is multi-threaded. In the main thread, the server listens to a fixed port, let say 8080. When it receives a TCP connection request, it sets up a TCP connection through another port and services the request in a separate thread. To simplify this programming task, we will develop the code in two stages. In the first stage, you will write a multi-threaded server that simply displays the contents of the HTTP request message that it receives. After this program is running properly, you will add the code required to generate an appropriate response.

Being a HTTP Server, you can connect to it using your browser e.g. Chrome, Firefox, or IE. There is a code template which is given to you so you can start easily. In the code template, there is a /www folder containing a static website (template) that is used to display on the client's web browser.

2 SIMPLE DATE SERVER

In the given template, the runServer() method is called when an instance of the WebServer is created.

2.1 CREATE A SOCKET

Add the following code to the function:

```
public void runServer() throws IOException {
    final ServerSocket server = new ServerSocket(8080);
    System.out.println("Listening for connection on port 8080 ....");
    while (true) {
        final Socket client = server.accept();
        // 1. Read HTTP request from the client socket
        // 2. Prepare an HTTP response
        // 3. Send HTTP response to the client
        // 4. Close the socket
    }

}
```

The first step is to create a network socket that can accept connection on certain TCP port. In this example, the program will be listening at 8080. We are also using ServerSocket class in Java to create a server which can accept request. Resolve any error, missing library and exception handling as per suggestion by NetBeans.

The code above is the basis of creating a webserver in Java. The server is now ready and accepting incoming connection on port 8080. You can try to compile and run the program. After that connect to `http://localhost:8080` from your browser, the connection will be established and browser will wait forever.

This is also the standard HTTP Server, which is stateless server. The server does not need to remember previous connection but only focus on the incoming connection.

2.2 READ HTTP REQUEST

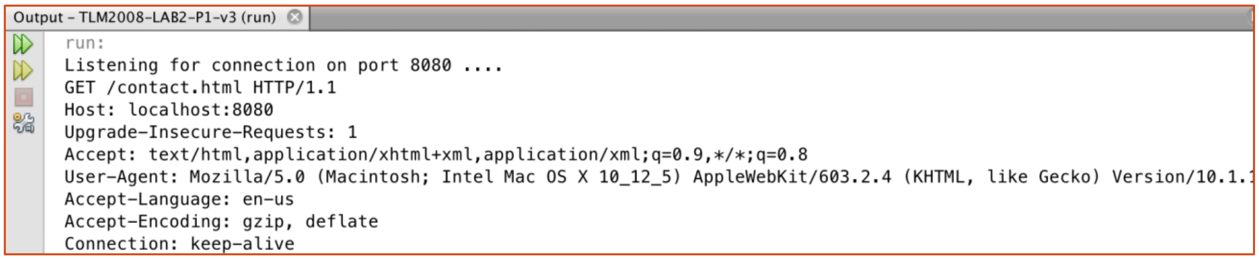
When connecting to `http://localhost:8080`, your browser will send a GET HTTP request to the server. In this session, we will read the content of request using InputStream opened from the client socket.

```java
private void readHttpRequest(Socket clientSocket) throws IOException{
    InputStreamReader isr = new InputStreamReader(clientSocket.getInputStream());
    BufferedReader reader = new BufferedReader(isr);
    String line = reader.readLine();
    while (!line.isEmpty()) {
       //TODO: Enter your own code
    }

}
```

As the browser will send multiple line, that is why we need to use BufferedReader to buffer the day. After that, we can print out all the lines in the request sent.

Fill in the missing code such that the program can print out the request to the output screen (System.out) as follow (if you are running from Netbeans). Remember to add this function to runServer().


Output – TLM2008-LAB2-P1-v3 (run)

```
run:
Listening for connection on port 8080 ....
GET /contact.html HTTP/1.1
Host: localhost:8080
Upgrade-Insecure-Requests: 1
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_12_5) AppleWebKit/603.2.4 (KHTML, like Gecko) Version/10.1.
Accept-Language: en-us
Accept-Encoding: gzip, deflate
Connection: keep-alive
```

2.3 PREPARE AND SEND RESPONSE

To keep our server simple, we will just send today's date to the client. Below is the code to do that:

```java
private void prepareAndSendResponse(Socket socket) throws IOException{
    Date today = new Date();
    String httpResponse = "HTTP/1.1 200 OK\r\n\r\n" + today;
    socket.getOutputStream().write(httpResponse.getBytes("UTF-8"));
}
```

In order to send the response, we need to get an output stream from the socket. The second line prepares the response by form the HTTP response code OK and today's date. The third line sends the response to the output stream.

Add all the function calls to the runServer(), compile and run the server. Try to connect through your browser, you should be able to see the date/time printed.

```
public void runServer() throws IOException {
    final ServerSocket server = new ServerSocket(8080);
    System.out.println("Listening for connection on port 8080 ....");
    while (true) {
        final Socket client = server.accept();
        // 1. Read HTTP request from the client socket
        this.readHttpRequest(client);
        // 2. Prepare an HTTP response
        // 3. Send HTTP response to the client
        this.prepareAndSendResponse(client);
        // 4. Close the socket
        client.close();
    }

}
```

3 MULTI-THREADED HTTP SERVER

The server created in section 2 is not able to handle multiple client request. This is because there is only one loop accept the connections. Once the server is serving a connection, others need to wait. In this section, we will design a multi-thread server that process client GET and POST request and send back appropriate content.

In this section, you can use the ___ template provided. In this section, the server will send back the requested file (if any) in the GET request received from the browser. In the second part of this exercise, you will be writing your own code to handle a submit form request and display the appropriate web page.

3.1 SERVER

The server handles each client request in a separate thread:

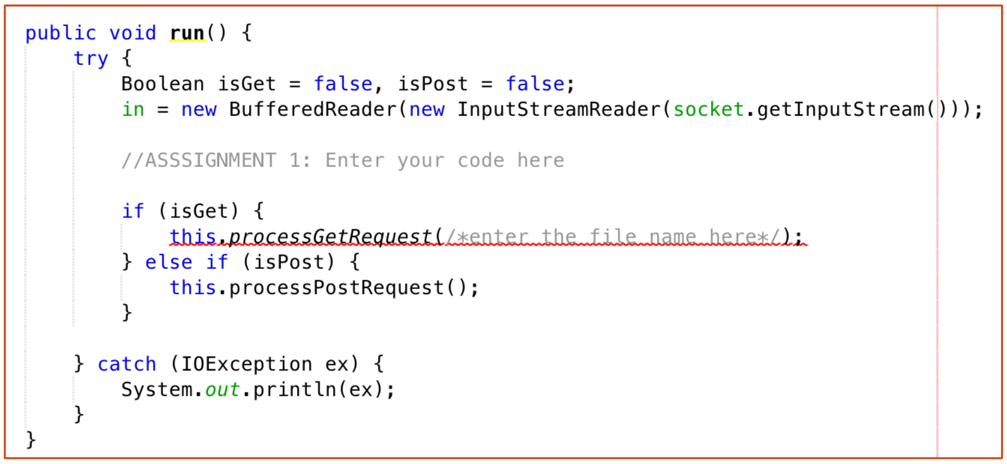
```
public static void main(String[] args) throws IOException {
    serverSocket = new ServerSocket(8080);  // Start, listen on port 80
    while (true) {
        try {
            Socket s = serverSocket.accept();  // Wait for a client to connect
            new ClientHandler(s);  // Handle the client in a separate thread
        } catch (IOException x) {
            System.out.println(x);
        }
    }
}
```

You can see that for each incoming connection, there is a new instance of the ClientHandler created. As such, the server can handle multiple request at the same time. You will be developing the ClientHandler in the subsequent session.

3.2 IDENTIFY REQUEST TYPE

Based on the simple date server in section 2, you should be able to print out and study the content of an Http request. Based on your understanding, complete the code below to identify if the incoming request is a GET or POST request:

3.2.1 ASSIGNMENT 2A



```
public void run() {
    try {
        Boolean isGet = false, isPost = false;
        in = new BufferedReader(new InputStreamReader(socket.getInputStream()));

        //ASSSIGNMENT 1: Enter your code here

        if (isGet) {
            this.processGetRequest(/*enter the file name here*/);
        } else if (isPost) {
            this.processPostRequest();
        }

    } catch (IOException ex) {
        System.out.println(ex);
    }
}
```

If you look at the code template, there are 2 empty methods processGetRequest() and processPostRequest(). In subsequent sections, we fill add code to these function.

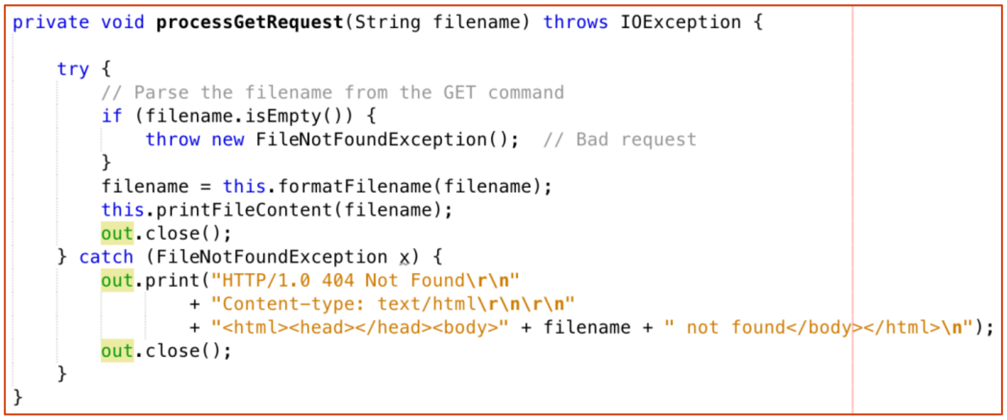
You need to insert your own code to:

1. Read the input from the input stream
2. Identify if the request of GET or POST by setting the pre-defined isGet and isPost value.
3. If it is a GET request, identify the name of the file being requested and pass the filename to
   `this.processGetRequest(/*enter the file name here*/);`

3.3 PROCESS GET REQUEST

When sending a GET request, the browser includes a file name that it requests for. In this section, we will add code to parse the file name, check if the file name is valid, check if the file is available and if so, send the file content to the browser.

Insert this code segment into the processGetRequest function:



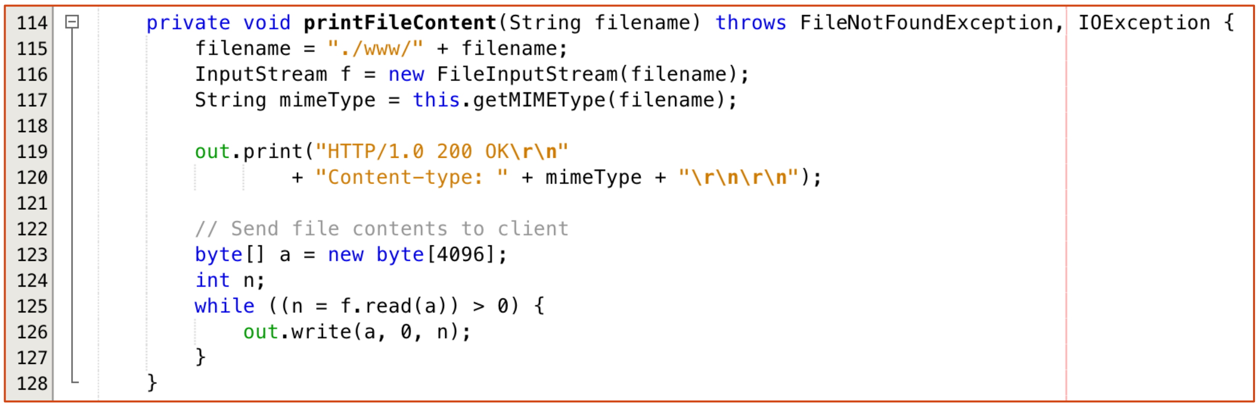
```
private void processGetRequest(String filename) throws IOException {

    try {
        // Parse the filename from the GET command
        if (filename.isEmpty()) {
            throw new FileNotFoundException();  // Bad request
        }
        filename = this.formatFilename(filename);
        this.printFileContent(filename);
        out.close();
    } catch (FileNotFoundException x) {
        out.print("HTTP/1.0 404 Not Found\r\n"
                + "Content-type: text/html\r\n\r\n"
                + "<html><head></head><body>" + filename + " not found</body></html>\n");
        out.close();
    }
}
```

This code segment first check if the filename is empty. If so, it throws a FileNotFoundException. Otherwise, it passes the filename to the formatFilename method. **Study the code template, you should be able to understand the various formating and standarding operations applied to a file name.**

Once the file is formatted, it is send to the method printFileContent to load and send the file content to the browser through the output stream. After that, we close the output stream. Next, we will be adding code to the printFileContent method.

3.4 PRINT FILE CONTENT TO BROWSER

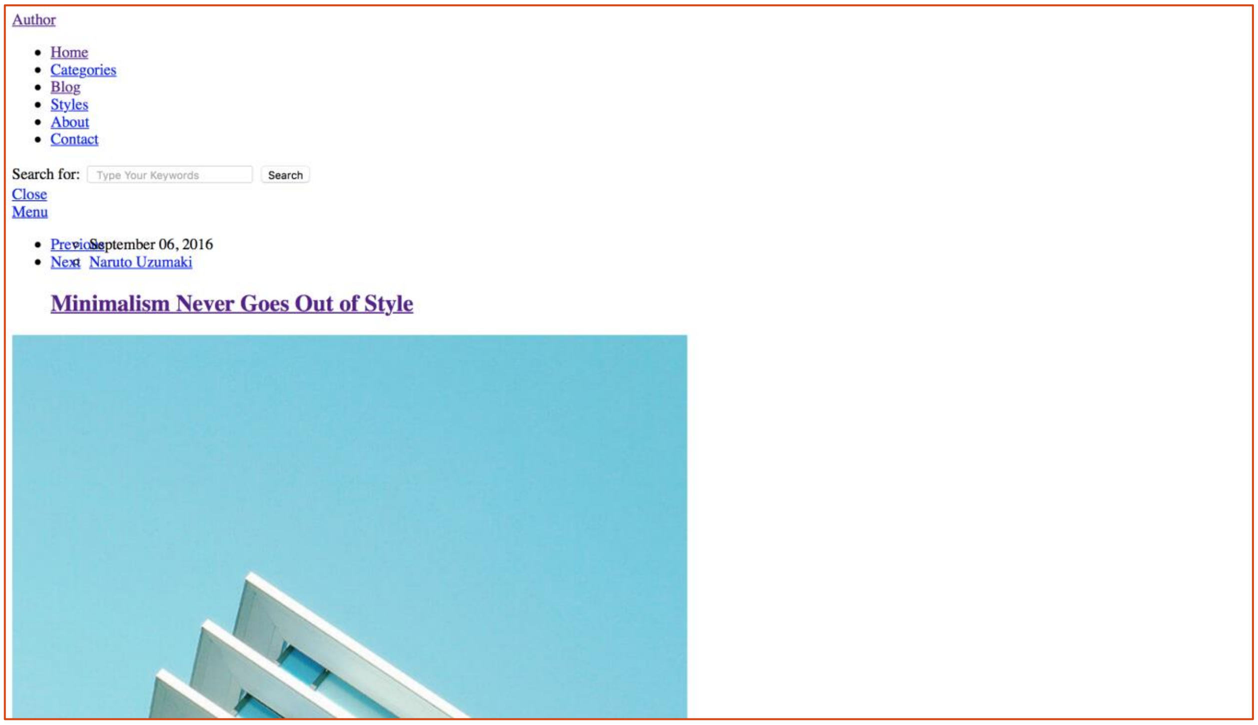


```
114     private void printFileContent(String filename) throws FileNotFoundException, IOException {
115         filename = "./www/" + filename;
116         InputStream f = new FileInputStream(filename);
117         String mimeType = this.getMIMEType(filename);
118 
119         out.print("HTTP/1.0 200 OK\r\n"
120                 + "Content-type: " + mimeType + "\r\n\r\n");
121 
122         // Send file contents to client
123         byte[] a = new byte[4096];
124         int n;
125         while ((n = f.read(a)) > 0) {
126             out.write(a, 0, n);
127         }
128     }
```

Line 117 determines the MIME type for the requested type. "The **MIME type** is the mechanism to tell the client the variety of document transmitted: the extension of a file name has no meaning on the web. It is, therefore, important that the server is correctly set up, so that the correct MIME type is transmitted with each document. Browsers often use the MIME-type to determine what default action to do when a resource is fetched." (Reference)

Line 119 add the HTTP header, mimeType and send it to the socket output stream.

Add the above code to your program, compile and run your application. You should be able to see the following content on the browser:


Author

- Home
- Categories
- Blog
- Styles
- About
- Contact

Search for: Type Your Keywords Search
Close
Menu

- PreviousSeptember 06, 2016
- Next Naruto Uzumaki

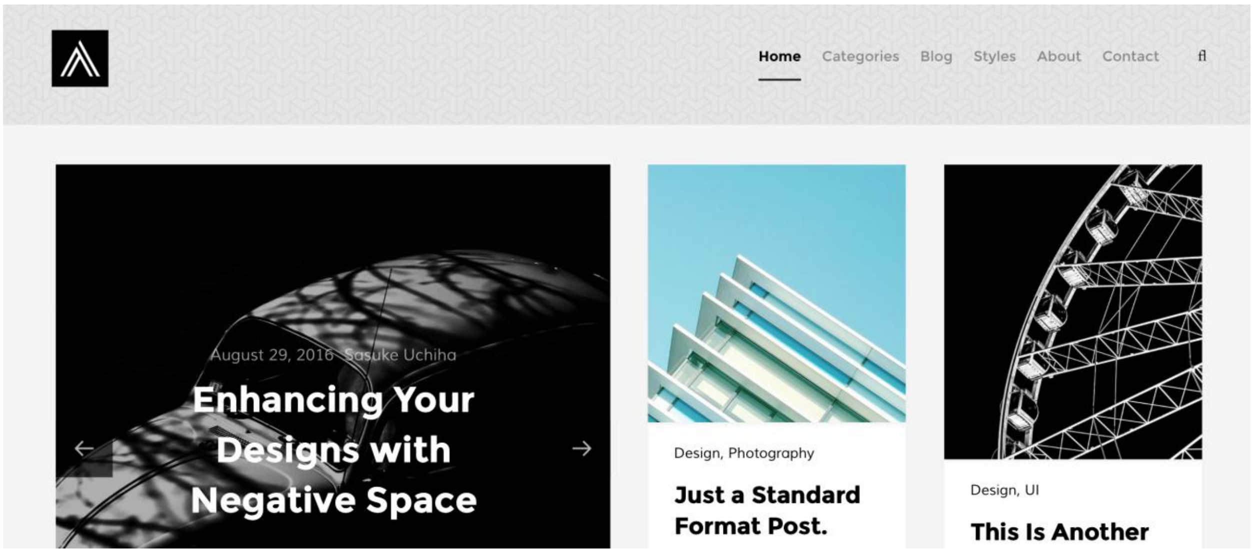
Minimalism Never Goes Out of Style

The content of the page is not formatted. This is because the css file is not property sent to the browser.

3.4.1 EXERCISE 2A

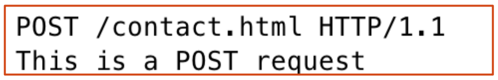
Study the getMIMEType method you should be able to see that the method is now written to handle text, html, jpeg, gif, class MIME types.

Add your own code so that css type is also handled. After that, you should be able to see the following webpage on the web browser:


Home
Categories
Blog
Styles
About
Contact
August 29, 2016 Sasuke Uchiha
Enhancing Your Designs with Negative Space
Design, Photography
Just a Standard Format Post.
Design, UI
This Is Another

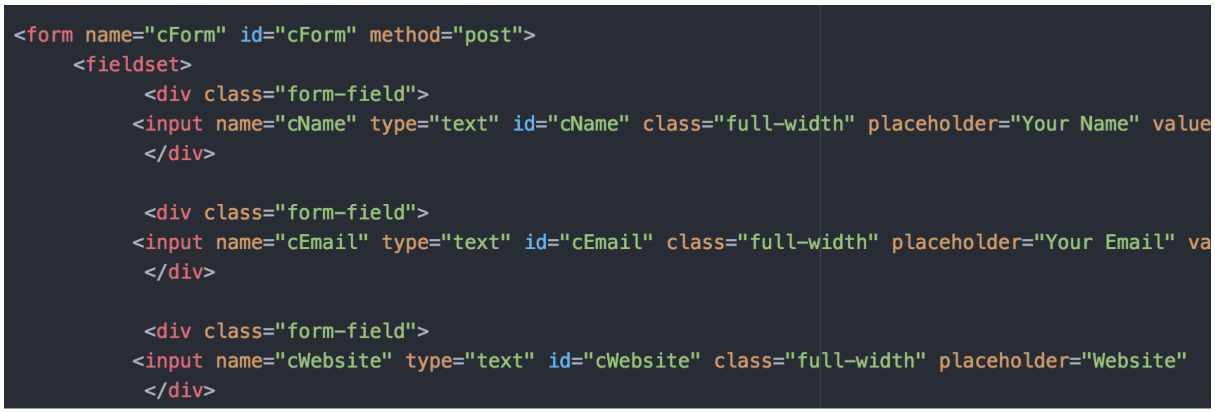
3.5 PROCESS POST REQUEST

Feel free to browse through the web template. Click on the contact tab, fill in the contact form and click Submit. You should be able to see under the Netbeans output screen that this is a POST request.



```
POST /contact.html HTTP/1.1
This is a POST request
```

To understand more about the reason that this is a post request, we need to look at the HTML form design for the page contact.html:



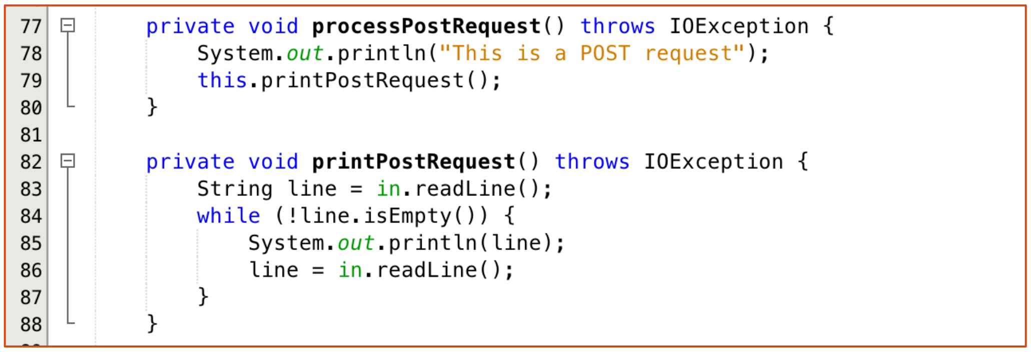
```
<form name="cForm" id="cForm" method="post">
     <fieldset>
           <div class="form-field">
          <input name="cName" type="text" id="cName" class="full-width" placeholder="Your Name" value
           </div>

           <div class="form-field">
          <input name="cEmail" type="text" id="cEmail" class="full-width" placeholder="Your Email" va
           </div>

           <div class="form-field">
          <input name="cWebsite" type="text" id="cWebsite" class="full-width" placeholder="Website"
           </div>
```

As can be seen from the above picture, the form indicates that the request method is "post". The HTTP Post request sends data to the server. As such, when you submit the form, the value of cName ("Your Name" field), cEmail ("Your Email" field), etc will be sent together with the POST request.

Enter the following code into your program:



```
    private void processPostRequest() throws IOException {
        System.out.println("This is a POST request");
        this.printPostRequest();
    }

    private void printPostRequest() throws IOException {
        String line = in.readLine();
        while (!line.isEmpty()) {
            System.out.println(line);
            line = in.readLine();
        }
    }
```

The code above use the technique in the Simple Date Server to print out the content of the POST request on your Netbeans output screen. Run your server again, fill in the contact form, submit the form again, you should be able to see the following output on the screen without the last line:

```
This is a POST request
Host: localhost:8081
Content-Type: application/x-www-form-urlencoded
Origin: http://localhost:8081
Accept-Encoding: gzip, deflate
Connection: keep-alive
Upgrade-Insecure-Requests: 1
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_12_5) AppleWebKit/603.2.4 (KHTML, like Gecko) Version/10.
Referer: http://localhost:8081/contact.html
Content-Length: 56
Accept-Language: en-us

cName=Alvin&cEmail=Alvin%40gmail.com&cWebsite=&cMessage=
```

3.5.1 ASSIGNMENT 2B

As you can see that the technique used in the Simple Date Server is only suitable for GET request. As for POST request, additional post data cannot be read by this approach. In this assignment, you are asked to modify/add/change the code of the method such that the last line can be read and returned so that the post data can be used in subsequent part. As such, the signature of the printPostRequest should be changed to:

```
private String printPostRequest()
```

3.5.2 EXERCISE 2B

Extract email address from the post request. In this assignment, you are asked to add your own code to the following function to extract the email address entered through the web form and print the email address to Netbeans output screen.

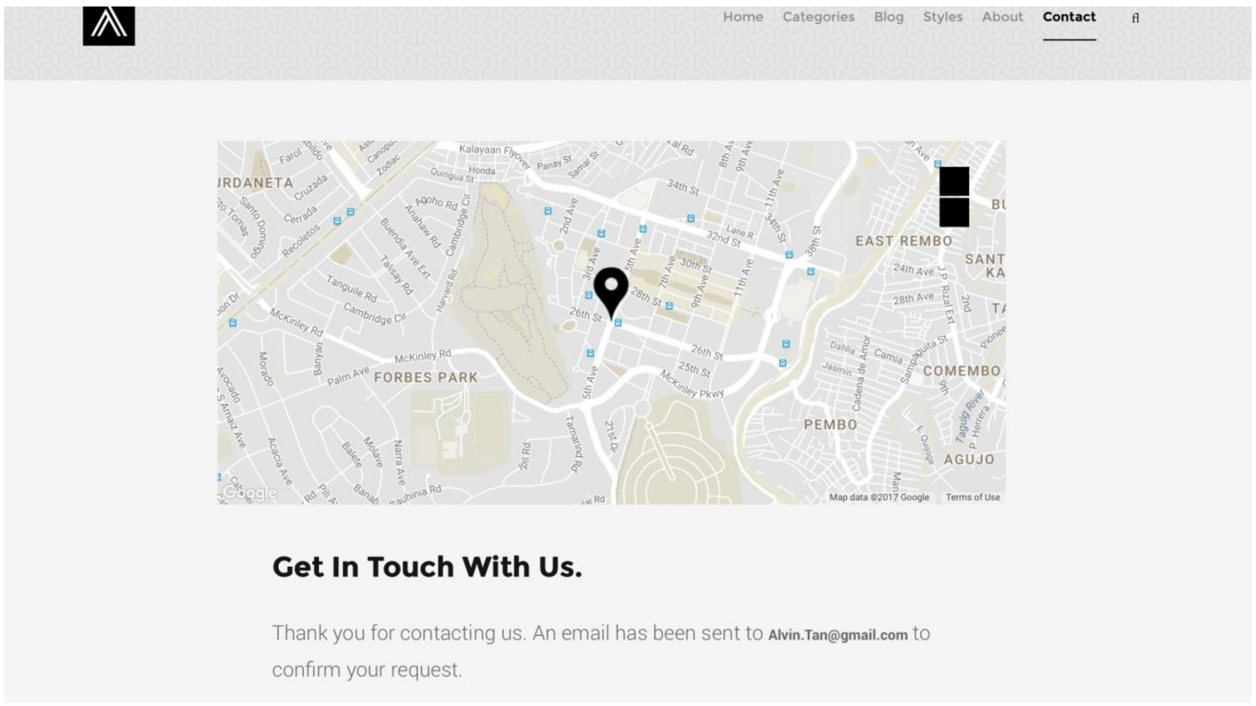
```
private void processPostRequhereest() throws IOException {
    System.out.println("This is a POST request");
    String postData = this.printPostRequest();
    String email = this.getEmailAddress(postData);

}

private String getEmailAddress(String postData){
    String email = "";
    //TODO: Enter your code
    System.out.println(email);
    return email;
}
```

3.6 REPONSE TO POST REQUEST

In this section, you will be extracting email submitted by the user from the form and send a contact confirmation page to the user's browser as the one below.

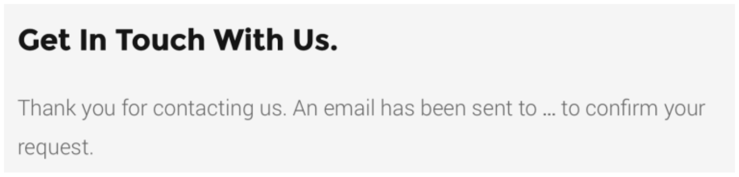


Get In Touch With Us.

Thank you for contacting us. An email has been sent to Alvin.Tan@gmail.com to confirm your request.

3.6.1 EXERCISE 2C (OPTIONAL)

In this assignment, you are required to add your own code so that the page as above is sent to the browser. You are provided with the page confirmContact.html. When opening the page, you should see the following:



Get In Touch With Us.

Thank you for contacting us. An email has been sent to ... to confirm your request.

You can make use of this page and insert the email address extracted from the form to the file and then send the content of the file to the client's browser.

One easy way to do implement this is to separate the confirmContact.html into the following 2 files:

- confirmContact1.html: Contains content from the beginning to "sent to"
- confirmContact2.html: Contains content from "to confirm your request." to the end.

After that you will have to write the content of confirmContact1.html, follow by the extracted email address, follow by the content of confirmContact2.html to the output stream through the socket to the client browser. Section 3.4 should give you some good idea on how to do this.