

Centro Federal de Educação Tecnológica de Minas Gerais
Campus VII - Unidade Timóteo - Engenharia da Computação
Trabalho Prático 3

Trabalho Prático III: Implementação de um chat web

Arthur Moraes Pimentel

Professor: Lucas Pantuza Amorim

Timóteo, Novembro de 2022

1 Introdução

Neste trabalho foi implementado um chat web apartir de dois projetos em linguagens distintas (Java e JavaScript), afim de exercitar o conceito de web services estudado em sala de aula. Foi levado em consideração o modelo de arquitetura REST, implementando assim duas API RESTful, que utilizam do protocolo de comunicação HTTP, enviando e recebendo arquivos no modelo JSON.

Foi optado estas duas linguagens de programação, por já haver uma familiaridade com as mesmas, tanto no âmbito educacional quanto profissional.

A captura dos pacotes trafegados foi realizado com o software Wireshark.

2 Metodologia

Para desenvolvimento e aplicabilidade dos projetos foi tomado dois caminhos para construção e elaboração dos mesmos, um para o projeto em Java e outro para o projeto em JavaScript.

2.1 Java

Para o projeto em Java foi optato um framework bastante conhecido no ramo de desenvolvimento web para esta linguagem, o springboot, ele traz consigo algumas facilidades quanto a criação de um serviço web, pelo fato de disbonibilizar algumas depedências que torna o desenvolvimento mais fluido e dinâmico. Para criação do projeto Spring, foi utilizado [Spring Initializr](#), um sistema capaz de gerar o esqueleto do projeto de acordo com as propriedades fornecidas.

As propriedades usadas para gerar o projeto SpringBoot foi:

Project: Maven, Language: Java, Spring Boot: 2.7.5(a versão mais recente disponivel), descrições do projeto, Packaging: Jar, Java: 17, Dependencies: [Spring web, DevTools, Lombok]

A IDE utilizada para o projeto java foi a [IntelliJIDEA](#), e para executar o código basta dá um Run na classe principal, a propria IDE já oferece recursos bem claros e precisos para execução do projeto.

Foi criado uma interface gráfica através da lib JFrame, própria da linguagem Java, onde a mesma proporciona uma melhor interação do sistema com o usuário, e oferece uma melhor visibilidade das funcionalidades pautadas. De acordo com a figura capturada no início da execução do sistema, temos a seguinte imagem ilustrativa 1.

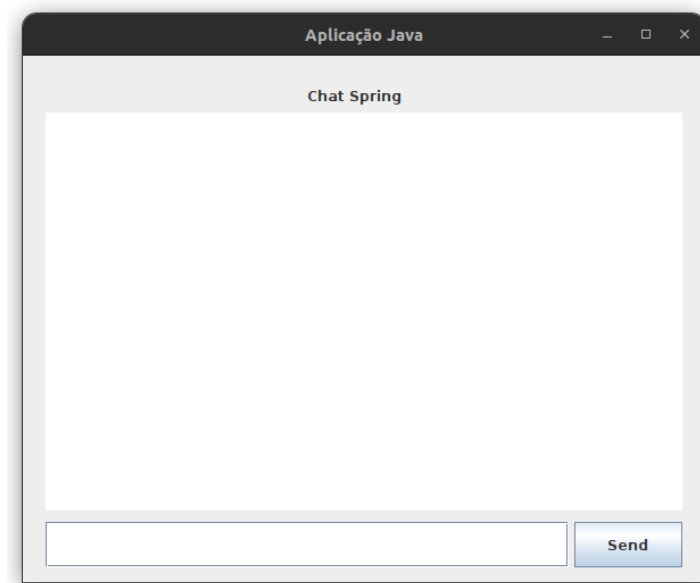


Figura 1: Interface Java

A tela iterativa por parte do projeto Java é basicamente esta apresentada acima, a mesma é responsável por mostrar as conversas na área reservada, e apresenta um botão "Send", que quando pressionado envia para o outro projeto JS a mensagem escrita ao lado do botão.

2.2 JavaScript

Para o projeto em JavaScript foi preciso utilizar o Node.js, um software que permite a execução de códigos JavaScript fora de um navegador web, o Node apresenta uma documentação bem sucinta e clara, até mesmo para criação de um servidor JS.

As dependências utilizadas no projeto Node foram:

Express, um framework que fornece recursos e suporte para construção do servidor web, foi usado principalmente para comunicação com o servidor Java. Http e socket.io, foram importantes, pois como o Node é um software que atua no backend, com isso a solução para executar um html e o mesmo comunicar com nosso servidor JS foi a de usar o Http que também trata e fornece aplicabilidades de

servidor, e o socket.io que permite a comunicação bidirecional em tempo real, utiliza de web socket, e nos permitiu a comunicação de nosso servidor NodeJS com um arquivo Html que foi usado para exibição de uma tela iterativa coforme a figura 2.

Chat node

Send

Figura 2: Interface Node

A tela iterativa por parte do projeto Node é basicamente esta apresentada acima, responsável por mostrar as conversas na área reservada abaixo do botão send, que quando precionado envia para o outro projeto Java a mensagem escrita no campo reservado em cima do botão.

Foi utilizado o Bootstrap, um framework capaz de fornecer a estilização para nosso html.

O projeto node foi criado utilizando o editor de texto VSCode, e para execução do mesmo basta abrir um terminal pelo proprio editor e digitar o seguinte comando: npm start, caso seja a primeira vez é preciso executar o comando npm install, que irá instalar todas as dependências usadas no projeto e citadas no arquivo package.json.

3 Resultados

Nos projetos criados para realização deste trabalho buscou-se implementar interfaces o mais simples possível, para contemplar a funcionalidade de um chat. No projeto Java foi abordado o uso da lib JFrame, a mesma pode ser visualizada quando executar o projeto.

Podemos ver mais detalhes na figura 3

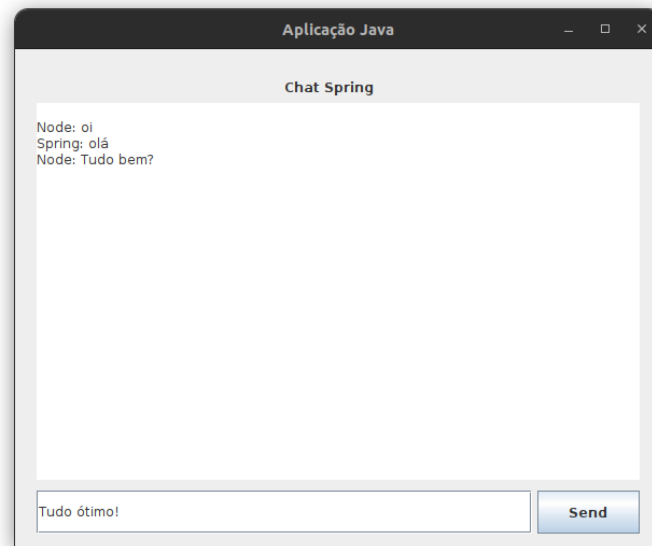


Figura 3: Chat Java

Já no projeto JavaScript foi abordado o uso do Bootstrap, que permitiu ter uma janela mais customizada. A janela deste projeto por sua vez pode ser visualizado pelo navegador ao acessar o endereço <http://localhost:3000>, após ter executado o projeto. Podemos visualizar a página pela figura 4

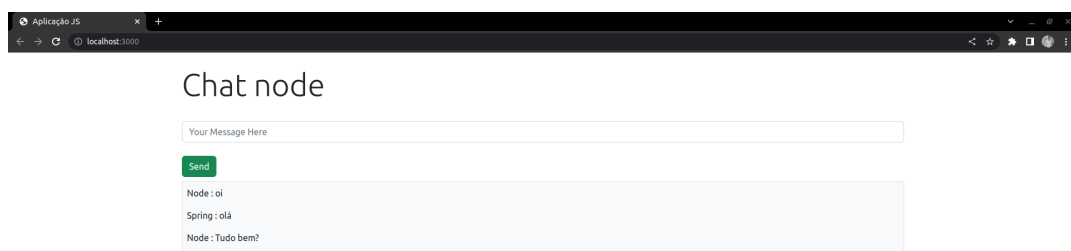


Figura 4: Chat JS

Após ser feita a comunicação desejada, para encerrar os programas basta fechar a JFrame no caso do projeto Java, e no projeto JavaScript basta ir no terminal que subiu a aplicação e digitar Ctrl+c.

4 Análise

No contexto analítico, podemos observar que um serviço web não depende diretamente de uma linguagem de programação para poder se comunicar a outros serviços disponíveis na web, tanto que esse trabalho buscou explorar este ponto, uma vez que foi elaborado dois projetos cuja as suas linguagens foram distintas.

O serviço web aqui estudado e implementado foi em cima do protocolo HTTP, cada projeto teve a sua forma de usa-ló, mas que no final das contas os dois vieram a usar dos métodos de requisição responsáveis por indicar a ação a ser executada para um dado recurso.

O método HTTP usado foi o POST, que tem como fundamento enviar dados ao servidor, por forma de uma requisição, como a submissão de um formulário. A requisição neste sentido é o pedido que um cliente envia a um servidor, nessa requisição há dados específicos, que no cenário aqui aplicado é a mensagem transferida em um arquivo JSON, no seguinte modelo:

```
{ name: nome da aplicação, message: conteudo da mensagem } .
```

Nas interfaces, as requisições de um servidor a outro é feita ao clicar no botão "send", o mesmo tem a funcionalidade de direcionar o código a uma função que irá realizar um requisição ao servidor oposto. Como este é um trabalho cujo a implementação se viu utilizar dois projetos, tem-se como ponto de partida que os dois projetos se comunicam dado o endereço de ambos, ou seja, o projeto Java tem ciência do endereçamento que o servidor JavaScript está operando, e o mesmo se dá para o servidor JavaScript, uma vez identificado e passado para ambos o endereço de cada um.

Para análise dos dados trafegados, foi utilizado o software Wireshark, que é um programa que analisa o tráfego da rede, e disponibiliza um relatório organizado por protocolos. Abaixo segue as subseções ilustrando os pacotes capturados ao ser efetuado a troca de mensagens entre os dois serviços web.

4.1 Análise Pacotes JavaScript

Nesta subseção será abordado os pacotes enviados pelo servidor JavaScript, capturados pelo Wireshark.

Na figura 5, podemos visualizar uma requisição feita pelo serviço JavaScript ao servido Java, pode ser visualizado que o protocolo é o HTTP e o método utilizado foi o POST, bem como o endpoint do serviço Java que foi acessado, "chat_spring".

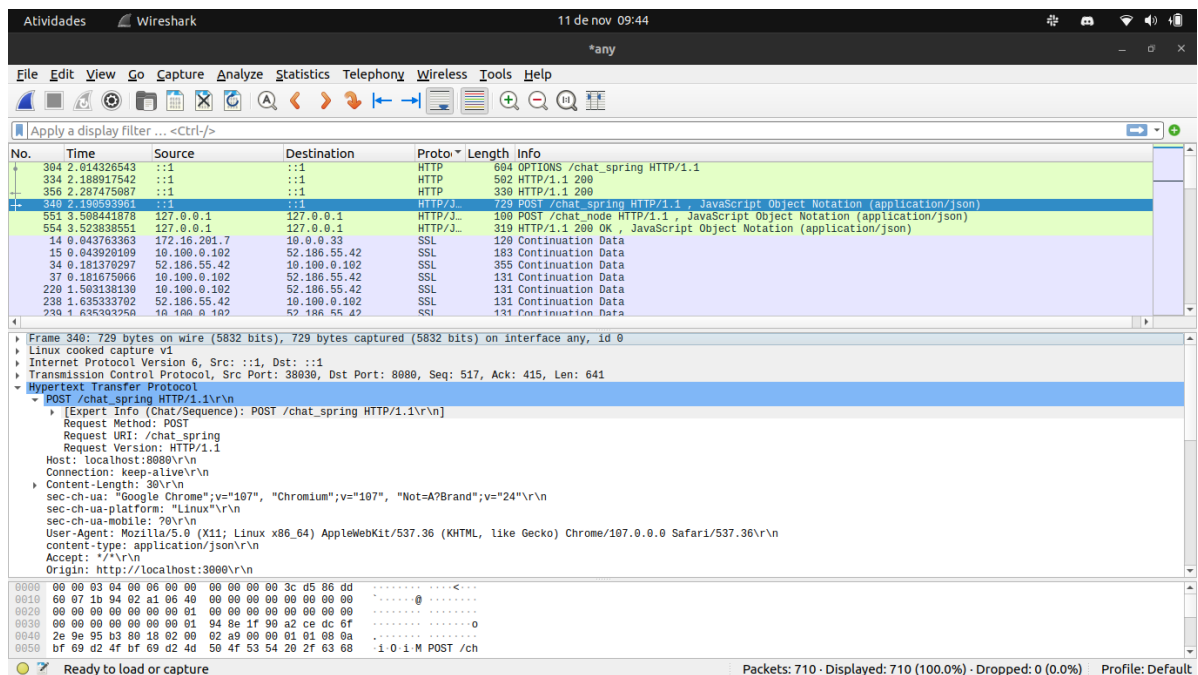


Figura 5: Pacote JavaScript

Na figura 6, podemos visualizar o mesmo pacote abordado na figura 5, porém aqui pode ser contemplado o arquivo json trafegado no pacote. Temos um Object que contém as chaves com seus valores, como é formado um arquivo JSON.

Podemos visualizar a chave "name" com seu valor sendo "Node", o que foi usado como um identificador dos serviços. Pode ser visualizado também a chave "message" com seu valor sendo de fato a mensagem, "oi".

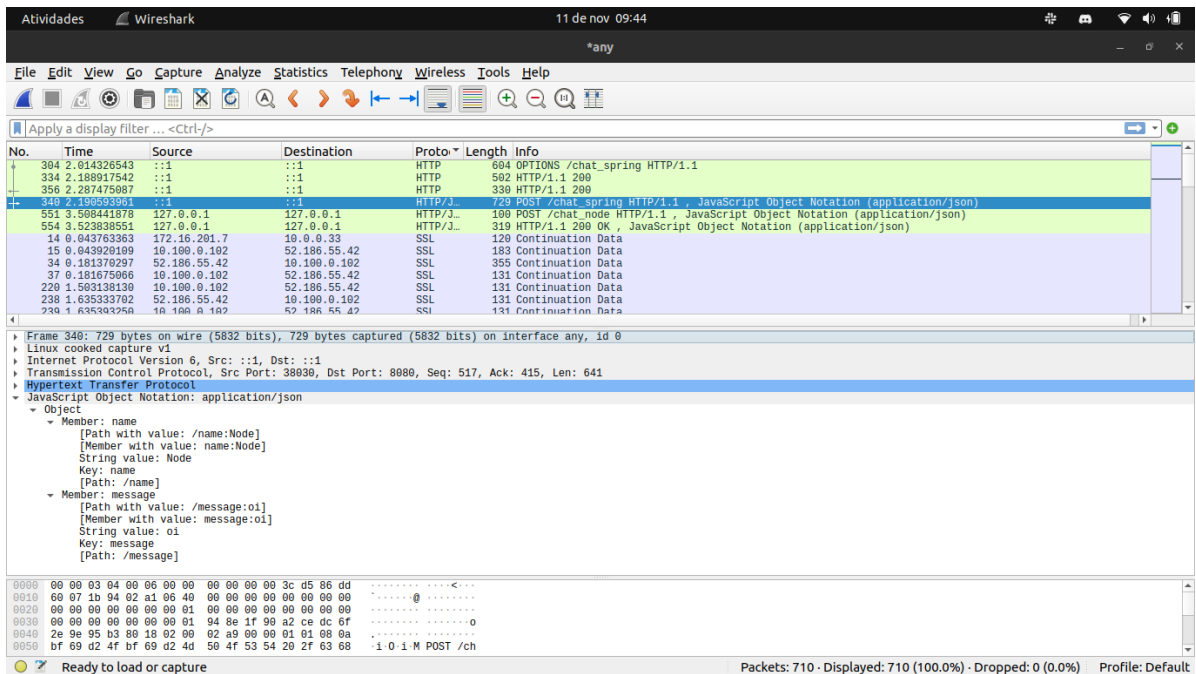


Figura 6: Mensagem do pacote JavaScript

4.2 Análise Pacotes Java

Nesta subseção será abordado os pacotes enviados pelo servidor Java, capturados pelo Wireshark.

Na figura 7, podemos visualizar uma requisição feita pelo serviço Java ao servido JavaScript, pode ser visualizado que o protocolo é o HTTP e o método utilizado foi o POST, bem como o endpoint do serviço JavaScript que foi acessado, "chat_node".

Na figura 8, podemos visualizar o mesmo pacote abordado na figura 7, porém aqui pode ser contemplado o arquivo json trafegado no pacote. Temos um Object que contém as chaves com seus valores, como é formado um arquivo JSON.

Podemos visualizar a chave "name" com seu valor sendo "Spring", o que foi usado como um identificador dos serviços. Pode ser visualizado também a chave "message" com seu valor sendo de fato a mensagem, "oi".

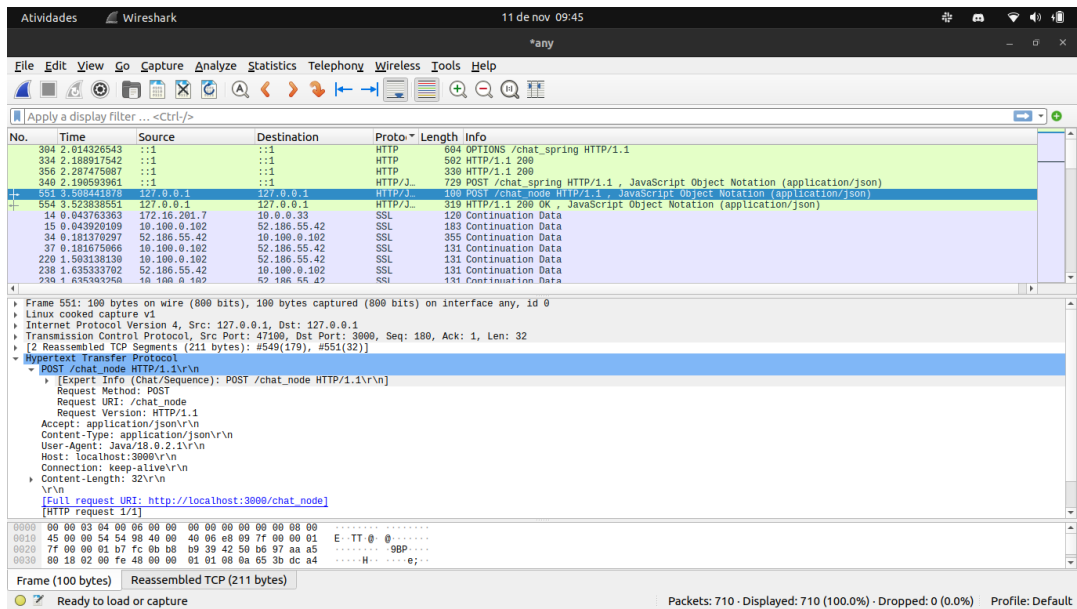


Figura 7: Pacote Java

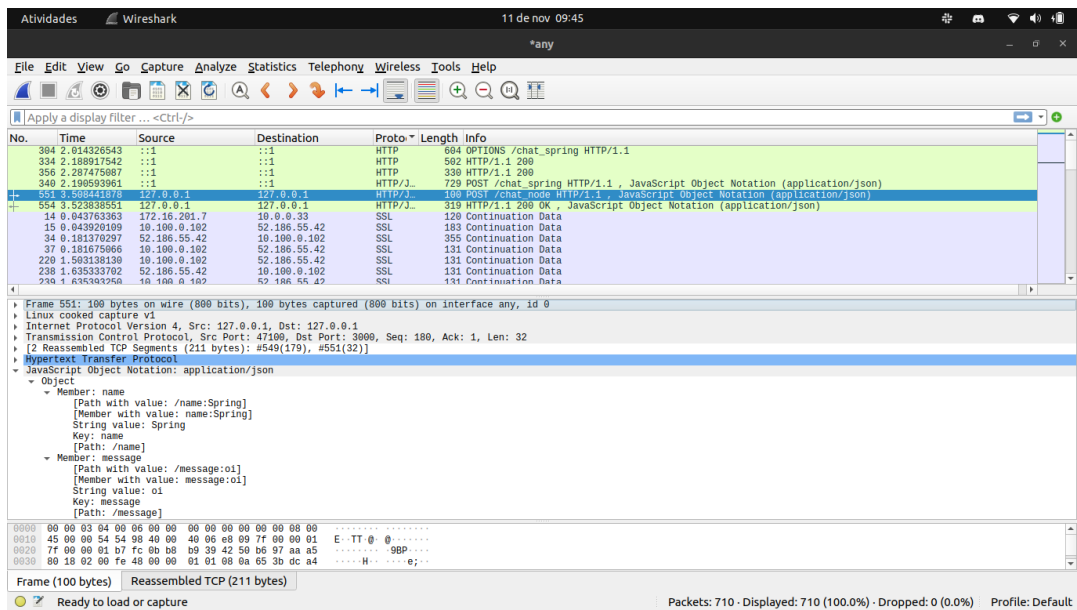
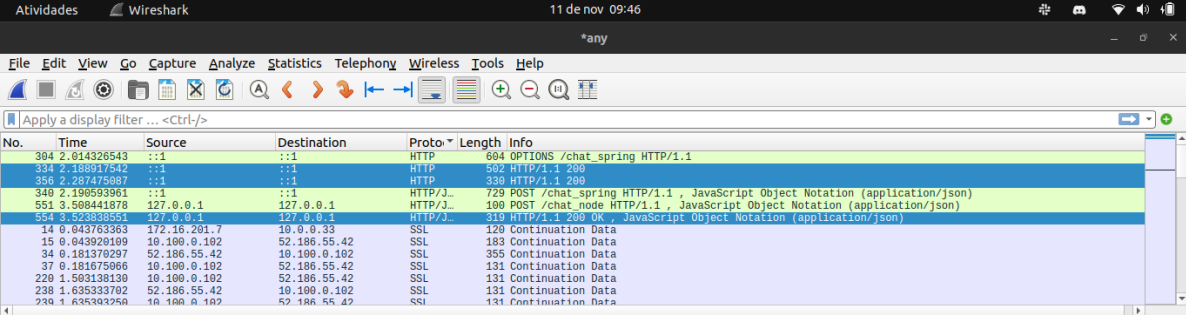


Figura 8: Mensagem do pacote Java

4.3 Resposta do protocolo http

Uma requisição http, possui uma resposta como uma forma de confirmar oque aconteceu com a requisição recebida por parte do servidor. Deste modo foi aplicado para o caso de sucesso um status code (200 - OK) para a requisição recebida. Caso ocorra alguma falha ou erro inesperado a resposta irá retornar algum outro status representando o erro contemplado.

Podemos visualizar na figura 9 as respostas por parte dos servidores marcados no wireshark, observa que o status é 200, ou seja as requisições por parte dos servidores foram bem sucedidas.



The image shows a Wireshark network traffic capture. The top bar indicates the date and time as '11 de nov 09:46'. The interface includes a menu bar (File, Edit, View, Go, Capture, Analyze, Statistics, Telephony, Wireless, Tools, Help) and a toolbar. Below the toolbar is a display filter set to '*any'. The main packet list table shows several captured packets. The selected packet is number 554, which is an HTTP response with status 200 OK. The table columns are No., Time, Source, Destination, Protocol, Length, and Info.

| No. | Time | Source | Destination | Protocol | Length | Info |
|-----|-------------|--------------|--------------|----------|--------|--|
| 304 | 2.014326543 | :::1 | :::1 | HTTP | 604 | OPTIONS /chat_spring HTTP/1.1 |
| 334 | 2.189917542 | :::1 | :::1 | HTTP | 502 | HTTP/1.1 200 |
| 350 | 2.281476507 | :::1 | :::1 | HTTP | 330 | HTTP/1.1 200 |
| 340 | 2.190593961 | :::1 | :::1 | HTTP/J | 729 | POST /chat_spring HTTP/1.1 , JavaScript Object Notation (application/json) |
| 551 | 3.508441878 | 127.0.0.1 | 127.0.0.1 | HTTP/J | 100 | POST /chat_node HTTP/1.1 , JavaScript Object Notation (application/json) |
| 554 | 3.528938551 | 127.0.0.1 | 127.0.0.1 | HTTP/J | 319 | HTTP/1.1 200 OK , JavaScript Object Notation (application/json) |
| 14 | 0.043763363 | 172.16.201.7 | 10.0.0.33 | SSL | 129 | Continuation Data |
| 15 | 0.043929109 | 10.100.0.102 | 52.186.55.42 | SSL | 183 | Continuation Data |
| 34 | 0.181370297 | 52.186.55.42 | 10.100.0.102 | SSL | 355 | Continuation Data |
| 37 | 0.181675066 | 10.100.0.102 | 52.186.55.42 | SSL | 131 | Continuation Data |
| 220 | 1.503136130 | 10.100.0.102 | 52.186.55.42 | SSL | 131 | Continuation Data |
| 238 | 1.635333702 | 52.186.55.42 | 10.100.0.102 | SSL | 131 | Continuation Data |
| 230 | 1.635303250 | 10.100.0.102 | 52.186.55.42 | SSL | 131 | Continuation Data |

At the bottom of the window, the status bar shows: 'Ready to load or capture', 'Packets: 710 · Displayed: 710 (100.0%) · Selected: 3 (0.4%) · Dropped: 0 (0.0%)', and 'Profile: Default'.

Figura 9: Response

5 Conclusão

Neste trabalho foi possível rever alguns dos conceitos que envolve um serviço web, como implementá-los nos projetos desenvolvidos. O protocolo HTTP responsável pela comunicação foi fundamental para este trabalho, no requisito de que foi possível criar duas aplicações totalmente distintas mais que se

comunicaram idenpendente da linguagem utilizada, bastando apenas acessar um método do protocolo http implemenado e disponibilizado em um endpoint pelos serviços.

Com a realização do trabalho apresentado, foi possível entender na prática um pouco melhor sobre os protocolos de transporte estudados em sala de aula, bem como ter a experiência de capturar os dados trafegados em rede pelo Wireshark.

6 Código

Todo o código fonte se encontra disponível em [GitHub](#).

Segue o código comentado:

6.1 Projeto Java

6.1.1 Main

Onde será inicializado a aplicação

```
1 package br.com.java.Spring;
2
3 import br.com.java.Spring.view.Window;
4 import org.springframework.beans.factory.annotation.Autowired;
5 import org.springframework.boot.SpringApplication;
6 import org.springframework.boot.autoconfigure.SpringBootApplication;
7 import org.springframework.boot.builder.SpringApplicationBuilder;
8 import org.springframework.context.ConfigurableApplicationContext;
9
10 @SpringBootApplication
11 public class Application {
12
13     @Autowired
14     private Window window;
15
16     public static void main(String[] args) {
17
```

```
18 // Initializes the application and allows using the jframe
19 SpringApplicationBuilder builder = new SpringApplicationBuilder(
20     Application.class);
21 builder.headless(false);
22 ConfigurableApplicationContext context = builder.run(args);
23 }
24
25 }
```

codes/Java/Application.java

6.1.2 Config

Foi necessário criar um arquivo de configuração por parte do servidor java, para que o seu endpoint implementado pudesse ser acessado pela api do javascript.

```
1 package br.com.java.Spring.config;
2
3 import org.springframework.context.annotation.Configuration;
4 import org.springframework.web.servlet.config.annotation.CorsRegistry;
5 import org.springframework.web.servlet.config.annotation.EnableWebMvc;
6 import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;
7
8 @Configuration
9 @EnableWebMvc
10 public class WebConfig implements WebMvcConfigurer {
11
12     @Override
13     public void addCorsMappings(CorsRegistry registry) {
14         registry.addMapping("/**");
15     }
16 }
```

codes/Java/config/WebConfig.java

6.1.3 Controller

Esta classe no Spring Boot é uma classe que tem como função realizar um controle de métodos para a rota "chat_spring", no cenário abordado foi necessário apenas um método que foi o post.

```
1 package br.com.java.Spring.controller;
2
3 import br.com.java.Spring.model.Message;
4 import br.com.java.Spring.view.Window;
5 import lombok.RequiredArgsConstructor;
6 import org.springframework.http.HttpStatus;
7 import org.springframework.web.bind.annotation.*;
8
9 @RestController
10 @RequestMapping("/chat_spring")
11 @RequiredArgsConstructor
12 public class ChatController {
13
14     private final Window window;
15
16     /*
17     * Route java
18     */
19     @PostMapping()
20     @ResponseStatus(HttpStatus.OK)
21     public void receiveMessage(@RequestBody Message message){
22         window.createMessage(message);
23     }
24 }
```

codes/java/controller/ChatController.java

6.1.4 Model

A classe Message tem o proposito de oferecer melhor visibilidade dos dados trafegados por parte do serviço java. As mensagem enviadas e recebidas terão o corpo baseado nesta classe.

```
1 package br.com.java.Spring.model;
2
3
4 import lombok.Data;
5
6 /*
7 * message template
8 *
9 */
10 @Data
11 public class Message {
12     private String name;
```

```
13     private String message;  
14 }
```

codes/java/model/Message.java

6.1.5 View

A classe View implementa a JFrame, que tem como funcionalidade gerar e renderizar a tela para uso do chat. É nesta mesma classe que é implementado a função para acessar o método do servidor JavaScript. A função actionPerformed é invocada quando é realizado um evento de click no botão "send", desta forma é estruturado um corpo para mensagem com base na classe modelo e então é realizado as configurações necessárias para realizar a requisição no outro server.

```
1 package br.com.java.Spring.view;  
2  
3 import br.com.java.Spring.model.Message;  
4 import org.springframework.context.annotation.Scope;  
5 import org.springframework.http.*;  
6 import org.springframework.stereotype.Component;  
7 import org.springframework.web.client.RestTemplate;  
8  
9 import javax.swing.*;  
10 import java.awt.event.ActionEvent;  
11 import java.awt.event.ActionListener;  
12 import java.util.Collections;  
13  
14 @Component  
15 @Scope("singleton")  
16 public class Window implements ActionListener {  
17  
18     JFrame jFrame = new JFrame();  
19  
20     private String text = "";  
21  
22     JLabel title = new JLabel("Chat Spring");  
23  
24     JTextArea fieldChat = new JTextArea("");  
25  
26     JTextField input = new JTextField();  
27  
28     JButton send = new JButton("Send");  
29  
30     public Window() {  
31         setConfigsWindow();  
32     }  
33 }
```

```
32
33     JFrame.add(title);
34     title.setBounds(250, 20, 100, 30);
35
36     JFrame.add(fieldChat);
37     fieldChat.setBounds(20, 50, 560, 350);
38     fieldChat.setEditable(false);
39
40     JFrame.add(input);
41     input.setBounds(20, 410, 460, 40);
42
43     JFrame.add(send);
44     send.setBounds(485, 410, 95, 40);
45     send.addActionListener(this);
46 }
47
48 private void setConfigsWindow(){
49     JFrame.setTitle("Aplicação Java");
50     JFrame.setLayout(null);
51     JFrame.setSize(600,500);
52     JFrame.setVisible(true);
53     JFrame.setResizable(true);
54     JFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
55     JFrame.setLocationRelativeTo(null);
56 }
57
58
59 public void createMessage(Message message){
60     text += "\n"+message.getName()+": "+message.getMessage();
61     fieldChat.setText(text);
62 }
63
64 /*
65  * Function that is executed from an event
66  * send the message to the js api
67  */
68 @Override
69 public void actionPerformed(ActionEvent e) {
70     Message message = new Message();
71     message.setName("Spring");
72     message.setMessage(input.getText());
73
74
75     // URL Service Node
76     String url = "http://localhost:3000/chat_node";
77
78     try{
79         HttpHeaders headers = new HttpHeaders();
```

```
80     headers.setContentType(MediaType.APPLICATION_JSON);
81     headers.setAccept(Collections.singletonList(MediaType.APPLICATION_JSON));
82
83     HttpEntity<Message> entity = new HttpEntity<>(message, headers);
84
85     RestTemplate restTemplate = new RestTemplate();
86
87     ResponseEntity<Message> response = restTemplate.postForEntity(url, entity, Message.class);
88
89     if(response.getStatusCode().equals(HttpStatus.OK)){
90         createMessage(message);
91         input.setText("");
92     }
93 } catch (Exception error){
94     error.printStackTrace();
95 }
96
97 }
98 }
99 }
```

codes/Java/view/Window.java

6.2 Projeto JavaScript

6.2.1 App

O arquivo app é o arquivo principal da aplicação js, que será executado assim que o projeto for iniciado. É feita a parte de configuração do servidor js, como a configuração do mini servidor socket que atua como comunicação entre o serviço javascript e o html renderizado.

O método http aqui implementado também é o post, e o mesmo ao receber uma requisição irá emitir uma mensagem para o html através da conexão via socket.

```
1 // Declarations
2 const express = require('express');
3 const app = express();
4 const bodyParser = require('body-parser');
```

```
5 const http = require('http');
6 const socket = require('socket.io');
7
8
9 // Configs
10
11 // Body Parser
12 app.use(bodyParser.urlencoded({ extended: false }));
13 app.use(bodyParser.json());
14
15 // Static File
16 app.use(express.static(__dirname + '/src/view'));
17
18 // Socket
19 const httpServer = http.createServer(app);
20 const io = socket(httpServer, {
21   path: '/socket.io'
22 })
23
24
25 // Client Socket
26 let clientSocket;
27
28 // Route
29
30 app.post('/chat_node', (req, res) => {
31   if(clientSocket) clientSocket.emit('message', req.body);
32
33   res.setHeader('content-type', 'application/json');
34   res.send({ message: "OK" });
35 })
36
37 // Initialize
38
39 // Connection Socket
40 io.on('connection', (client) => {
41   console.log('Client ${client.id} connected');
42   clientSocket = client;
43
44   client.on('disconnect', () => {
45     console.log('Client ${client.id} disconnect');
46     clientSocket = null;
47   })
48 })
49
50 // Server
51 httpServer.listen(3000, function () {
52   console.log('Servidor inicializado em: http://localhost:3000');
```

53 | });

codes/JavaScript/app.js

6.2.2 View

Neste projeto JS a parte da janela iterativa para uso do chat é feita por esse arquivo html. Foi utilizado o bootstrap para melhorar e facilitar a estilização do componente, bem como a lib de socket io, para que o html pudesse comunicar com o serviço JS.

É neste mesmo arquivo html que é implementado a função responsável por realizar uma requisição na api do java, levando como ideia a mesma ideia do projeto java, no que diz respeito a chamada da função, que seria quando for efetuado o evento de click no botão send.

```
1 <!DOCTYPE html>
2 <html lang="pt-br">
3   <head>
4     <meta charset="UTF-8" />
5     <meta http-equiv="X-UA-Compatible" content="IE=edge" />
6     <meta name="viewport" content="width=device-width, initial-scale=1.0" />
7     <title>Aplica o JS</title>
8
9     <!-- Bootstrap -->
10    <link
11      href="https://cdn.jsdelivr.net/npm/bootstrap@5.2.2/dist/css/
12        bootstrap.min.css"
13      rel="stylesheet"
14      integrity="sha384-Zenh87qX5JnK2JlOvWa8Ck2rdkQ2Bzep5IDxbcnCeuOxjzrPF/
15        et3URy9Bv1WTRi"
16      crossorigin="anonymous"
17    />
18
19    <!-- Socket IO -->
20    <script
21      src="https://cdn.socket.io/4.5.3/socket.io.min.js"
22      integrity="sha384-
23        WPFUvHkB1aHA5TDSZi6xtDgkF0wXJcIlxXhC6h8OT8EH3fC5PWro5pWJ1THjcfEi"
24      crossorigin="anonymous"
25    ></script>
26  </head>
27  <body>
```

```
25 <div class="container">
26   <br />
27   <div class="jumbotron">
28     <h1 class="display -4">Chat node</h1>
29     <br />
30     <input
31       id="message"
32       class="form-control"
33       placeholder="Your Message Here"
34     />
35     <br />
36     <button id="send" class="mb-2 btn btn-success" onclick="
37       sendMessage()">Send</button>
38     <div id="messages" class="p-2 bg-light border"></div>
39   </div>
40 </div>
41
42 <script>
43   // Initialize service socket
44   const socket = io();
45
46   // Receive the messages
47   // and execute the function provided in the second parameter
48   socket.on("message", addMessages);
49
50   // function that receives the message and rederizes it in the html
51   function addMessages(newMessage) {
52     const element = document.getElementById("messages");
53     const message = document.createElement("p");
54     message.textContent = `${newMessage.name} : ${newMessage.message}
55     `;
56     element.appendChild(message)
57   }
58
59   // function that when captured a click is executed
60   function sendMessage() {
61     const input = document.getElementById("message");
62     const message = {
63       name: "Node",
64       message: input.value
65     };
66
67     let headers = new Headers();
68     headers.append('Content-Type', 'application/json');
69
70     // access the method provided in the java api
71     fetch("http://localhost:8080/chat_spring", {
```

```
71     method: "POST",
72     headers: headers,
73     body: JSON.stringify(message)
74   })
75   .then((result) => {
76     if(result.status == 200){
77       addMessages(message);
78       input.value = "";
79     }else{
80       alert("Falha na requisi o!");
81     }
82   })
83   .catch((error) => {
84     console.log(error)
85     alert("Falha ao enviar a mensagem!")
86   })
87   }
88   </script>
89 </body>
90 </html>
```

codes/JavaScript/view/index.html
