Distributed Systems Project Report
2nd Goal

André Ferreira
uc2021233398

May 16, 2024

Contents

1 Introduction

Moving on to this part of the project, Gradle was the selected build tool. To integrate the frontend with the existing project (backend), the chosen method of organization was to separate the backend and frontend into two different folders, each with their own Gradle configuration file.

2 Dependencies

2.1 Backend

JSoup
JUnit5

2.2 Frontend

OkHTTP3
FasterXML Jackson
SpringBoot

3 Structure

3.1 Frontend

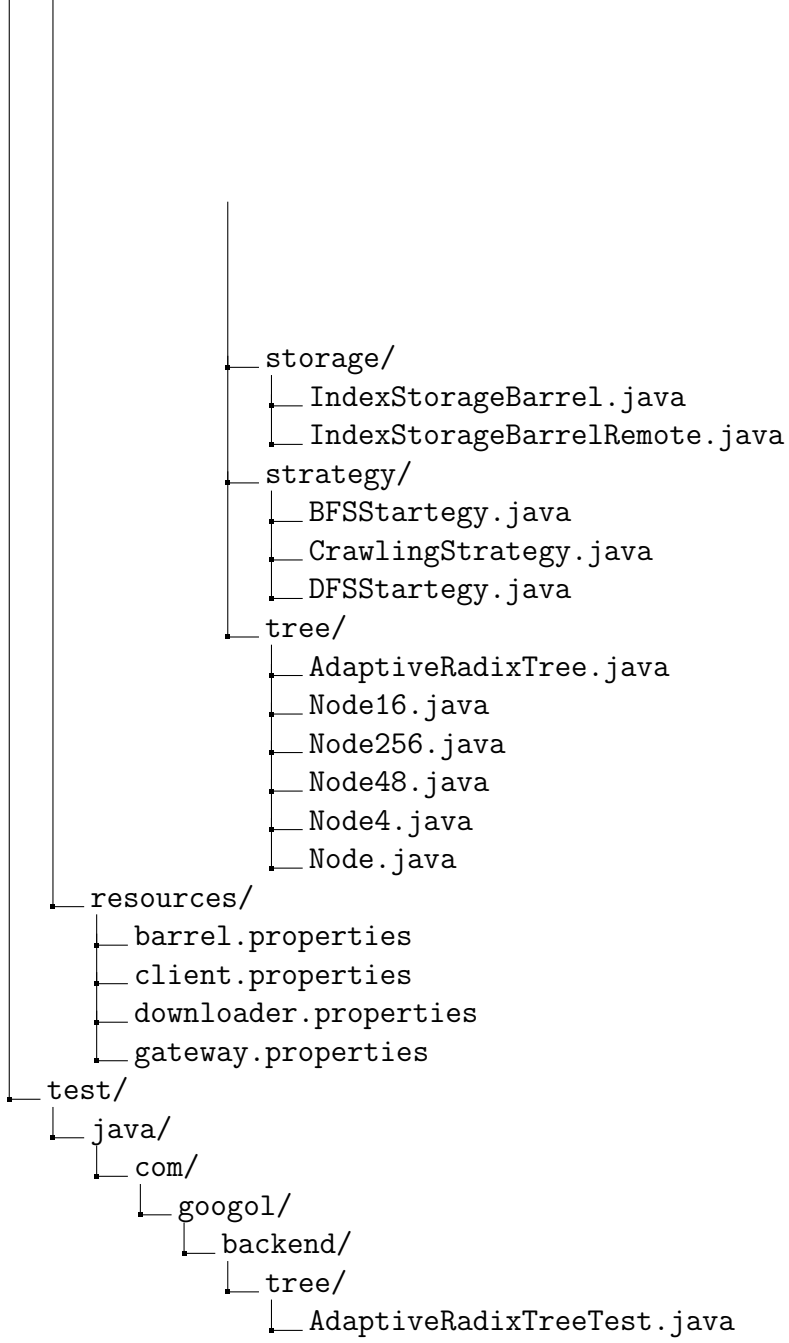
```
frontend
├── build.gradle
└── src/
```

```
└── main/
    ├── java/
    │   └── com/
    │       └── googol/
    │           └── frontend/
    │               ├── App.java
    │               ├── config/
    │               │   ├── HttpsRedirectConfig.java
    │               │   └── WebSocketConfig.java
    │               ├── controller/
    │               │   ├── AIAnalysisController.java
    │               │   ├── ConfigController.java
    │               │   ├── FatherURLsController.java
    │               │   ├── HackerNewsController.java
    │               │   ├── HomeController.java
    │               │   ├── SearchController.java
    │               │   ├── URLFathersController.java
    │               │   └── URLIndexationController.java
    │               ├── handler/
    │               │   └── WebSocketHandler.java
    │               ├── model/
    │               │   └── Story.java
    │               ├── rmi/
    │               │   ├── Gateway.java
    │               │   └── UpdateCallbackImpl.java
    │               └── service/
    │                   ├── GoogleAIService.java
    │                   ├── HackerNewsService.java
    │                   └── OpenAIService.java
    └── resources/
        ├── application.properties
        ├── googol-keystore.p12
        ├── keys.properties
        ├── mycert.crt
        ├── mykey.key
        └── static/
            └── css/
                └── admin.css
```

```
        ├── imgs/
        ├── js/
        │   ├── admin.js
        │   ├── index.js
        │   └── operation.js
        └── templates/
            ├── admin.html
            ├── footer.html
            ├── index.html
            └── search-results.html
    ├── webapp/
    │   └── WEB-INF/
    │       └── views/
└── test/
    └── java/
```

3.2 Backend

```
backend
├── build.gradle
└── src/
    ├── main/
    │   └── java/
    │       └── com/
    │           └── googol/
    │               └── backend/
    │                   ├── client/
    │                   │   └── Client.java
    │                   ├── downloader/
    │                   │   └── Downloader.java
    │                   ├── gateway/
    │                   │   ├── Gateway.java
    │                   │   ├── GatewayRemote.java
    │                   │   └── UpdateCallback.java
    │                   └── model/
    │                       ├── ParsedUrlIdPair.java
    │                       ├── ParsedUrl.java
    │                       └── RawUrl.java
```



4 Implementation

4.1 RMI

It was implemented RMI with callback, so that the backend can notify the frontend users/sessions when something changes (related to the system status). If the RMI connection was not successful, the frontend will proceed without any errors (apart from the console logging) and work without it. When executing some operation on the frontend that needs to interact with the backend through RMI and if the RMI failed to connect before, there

will be an attempt to reconnect. If it is successful, the program continues as expected, otherwise no results are shown on the operation. This is made possible with the use of the Optional Java util and Nullable Spring annotation:

```
@Nullable
private GatewayRemote gatewayRemote;

@Bean
public Optional<GatewayRemote> gatewayRemote() {
    return Optional.ofNullable(gatewayRemote);
}

public Optional<GatewayRemote> getOrReconnect(){
    if(gatewayRemote == null){
        gatewayRemote = connectToGatewayRMI();
    }
    return Optional.ofNullable(gatewayRemote);
}
```

4.1.1 Backend Changes

These functions will be used by the frontend when connecting to the Gateway RMI. The frontend will create a new UpdateCallbackImpl object and register it in the backend, so that the backend can keep track of all the ongoing sessions. It works the same way when the session is unregistered. When the backend has a change and need to notify the frontend clients, notifyClients() will be used, iterating every active session and transimiting it a message, using the onUpdate() function that is declared on the UpdateCallback interface on the backend and defined on the frontend with the actual code that will transmit the message.

```
# Gateway functions
private static final ArrayList<UpdateCallback> clients = new ArrayLi

    @Override
    public synchronized void registerUpdateCallback(UpdateCallback c
        clients.add(client);
    }
```

```
    @Override
    public synchronized void unregisterUpdateCallback(UpdateCallback
        clients.remove(client);
    }

    // notify all registered clients
    private static void notifyClients(ArrayList<ArrayList<String>> m
        synchronized (clients) {
            for (UpdateCallback client : clients) {
                try {
                    client.onUpdate(message);
                } catch (RemoteException e) {
                    System.out.println("Error notifying client: " +
                }
            }
        }
    }
```

```
# RMI update callback interface
package com.googol.backend.gateway;

import java.rmi.Remote;
import java.rmi.RemoteException;
import java.util.ArrayList;

public interface UpdateCallback extends Remote {
    void onUpdate(ArrayList<ArrayList<String>> message) throws Remot
}
```

4.1.2 Frontend Implementation

The Gateway remote interface is created and the RMI looked up. After this, the UpdateCallbackImpl object, implementing the backend UpdateCallback object, is created (containing the overrided onUpdate function) and registered on the backend.

```
# Gateway RMI init (connecting + registering session)
@PostConstruct
    public void init(){
        gatewayRMIURL = "//" + gatewayRMIHost + ":" + gatewayRMIPort
        gatewayRemote = connectToGatewayRMI();

        // Register the callback
        if (gatewayRemote != null) {
            try {
                clientCallback = new UpdateCallbackImpl();
                gatewayRemote.registerUpdateCallback(clientCallback)
                System.out.println("Registered callback successfully
            } catch (Exception e) {
                System.err.println("[ERROR] Failed to register the c
            }
        }
    }
```

The onUpdate function is overrided and when it is called in the backend to notify the clients, it will broadcast a message to the web socket:

```
public class UpdateCallbackImpl extends UnicastRemoteObject implemen
    public UpdateCallbackImpl() throws RemoteException {
        super();
    }

    @Override
    public void onUpdate(ArrayList<ArrayList<String>> message) throw
        WebSocketHandler.broadcast(message);
    }
}
```

4.2 Websockets

The websocket in this project has the purpose of broadcasting the system information when it changes (i.e. when a new barrel connects) without using a polling strategy and avoiding wasting CPU cycles.

4.2.1 Configuration

The websocket is registered at the /ws endpoint, but this is configurable in the Spring application.properties:

```
@Value("${websocket.endpoint}")
private String websocketEndpoint;

@Autowired
private WebSocketHandler webSocketHandler;

@Override
public void registerWebSocketHandlers(WebSocketHandlerRegistry r
    registry.addHandler(webSocketHandler, "/" + websocketEndpoin
}
```

4.2.2 Handling

When the websocket connection is established, the session is added to the ongoing sessions and the frontend immediately tries to retrieve the backend system info to populate the HTML admin panel:

```
private static final Set<WebSocketSession> sessions = Collection

@Override
public void afterConnectionEstablished(WebSocketSession session)
    sessions.add(session);

    Optional<GatewayRemote> gatewayRemote = gateway.getOrReconne
    ArrayList<ArrayList<String>> systemInfo = gatewayRemote.map(
        try{
            return remote.getSystemInfo();
        } catch (Exception e){
            System.out.println("[ERROR] Failed to get system inf
            return null;
        }
    }).orElseGet(() -> null);

    if(systemInfo == null) return;
```

```
        String jsonMessage = buildJSONFromMessage(systemInfo);
        try {
            // Send the JSON message to the client
            session.sendMessage(new TextMessage(jsonMessage));
        } catch (Exception e) {
            System.out.println("Failed␣to␣send␣message:␣" + e.getMes
        }
    }
```

When the connection is closed, the session is removed from the ongoing sessions:

```
    @Override
    public void afterConnectionClosed(WebSocketSession session, Clos
        sessions.remove(session);
    }
```

Finally, when there is a message broadcast (sent from the backend with the updated system info), the message, which comes as an ArrayList¡ArrayList¡String¿¿ is parsed into a JSON to then send to each ongoing session:

```
    public static void broadcast(ArrayList<ArrayList<String>> messag
        String jsonMessage = buildJSONFromMessage(message);

        synchronized (sessions) {
            for (WebSocketSession session : sessions) {
                try {
                    // Send the JSON message to the client
                    session.sendMessage(new TextMessage(jsonMessage)
                } catch (Exception e) {
                    System.out.println("Failed␣to␣send␣message:␣" +
                }
            }
        }
    }
```

4.2.3 Use in JavaScript

This is the JavaScript code used to connect to the web socket and what to do when it opens, when it broadcasts a message, when it closes and when it errors:

```
socket.onopen = () => {
        console.log("WebSocket␣connection␣established.");
    };

    socket.onmessage = (event) => {
        const message = event.data;
        processWebsocketMessage(message);
    };

    socket.onclose = () => {
        console.log("WebSocket␣connection␣closed.");
    };

    socket.onerror = (error) => {
        console.error('WebSocket␣error:', error);
    };
```

It simply logs when the connection opens and closes. It errors to the console when it receives an error and it sends the message received to a helper function when it receives a message, to display it on the HTML page.

4.3 REST

4.3.1 Hacker News

A simple controller was created to expose an endpoint and interact with the service:

```
@RestController
@RequestMapping("/api/hacker-news")
public class HackerNewsController {
    @Autowired
    private HackerNewsService hackerNewsService;

    @GetMapping
```

```
    public ArrayList<Story> getMatchingStories(@RequestParam("query"
        ArrayList<String> searchTerms = new ArrayList<>(Arrays.asLis
        return hackerNewsService.getMatchingStories(searchTerms);
    }
}
```

The service gets all the top stories from the API endpoint that returns the top stories IDs. With the IDs, requests are amde to another endpoint that returns the details of each story including the url. These request are made asynchronously to speed up processing. With the url, the page content is fetched using OkHTTP. After each story is processed, it is added to a concurrent hash map to speed up subsequent stories processing. Each story is put in an object with all that story details (including its unique words). When a search is made, this process happens and the search terms are weighed against the unique words from each story, and if they match then it is returned be indexed if the user wants to do so.

4.3.2 AI Generation

In this prospect, it was advised to use OpenAI API, but as it was giving errors, the Google AI API was used. The code doesn't change one bit apart from the API endpoint and the API key.
Similarly to the Hacker News structure, here a Controller is also used to expose the endpoints and interact with the services (separate endpoint and service for OpenAIAPI and GoogleAIAPI).
The services simply uses OkHTTP to make a request to the API endpoint and returns the response.
Here a prompt is needed to guide the bot in the right direction, so it is in the application.properties file and is loaded by the services and used when making the requests.

4.4 Relevant Features

4.4.1 HTTPS

The project is set up to work with HTTPS. This was done by creating a certificate and key locally. Even though this is okay for development, it is not so for production. For that purpose a certificate must be obtained from a reputable source.

Also, a Config class exists in the project to redirect to HTTPS when an HTTP connection is being made.

4.4.2 Config Controller

To reduce hardcoded configs and variables and improve the project security and make it more robust, the Spring application.properties file was used extensively. This means every possible configuration goes in it, like for example the host, port, websocket endpoint, API endpoints, API keys. Sensitive configurations, such as API keys are stored in a secondary properties file that is ignored by git, thus making it safe to put the API keys in there.

While Spring can easily access these properties using the @Value annotation, the same does not apply to the JavaScript, where some configurations are needed. To mitigate this, a ConfigController was created. This controller loads all the necessary properties that JavaScript needs and exposes a GET endpoint where it can fetch those configurations. This approach is not totally secure, because these endpoints are exposed to everyone, so it would need some kind of authentication.

One example of this is the web socket endpoint building in JavaScript:

```
const hostResponse = await fetch('/api/config/host');
if (!hostResponse.ok) {
    throw new Error('Network␣response␣was␣not␣ok␣(host).');
}
host = await hostResponse.text();

const portResponse = await fetch('/api/config/port');
if (!portResponse.ok) {
    throw new Error('Network␣response␣was␣not␣ok␣(port).');
}
port = await portResponse.text();

const websocketEndpointResponse = await fetch('/api/config/websocket
if (!websocketEndpointResponse.ok) {
    throw new Error('Network␣response␣was␣not␣ok␣(websocket␣endpoint
}
websocketEndpoint = await websocketEndpointResponse.text();
```

```
const socket = new WebSocket(`wss://${host}:${port}/${websocketEndpo
```

5 Build & Run

5.1 Compiling the Backend

To compile the backend, execute the following commands:

```
./gradlew build
./gradlew pack
```

5.2 Compiling the Frontend

To compile the frontend, use the command:

```
./gradlew bootJar
```

5.3 Running the Backend

Run the compiled backend JAR files located in `backend/build/libs` directory using the following commands:

```
java -jar Gateway-0.1.0.jar
java -jar IndexStorageBarrel-0.1.0.jar
java -jar Downloader-0.1.0.jar
java -jar Client-0.1.0.jar
```

5.4 Running the Frontend

Run the compiled frontend JAR file located in `frontend/build/libs` directory with the command:

```
java -jar frontend-0.1.0.jar
```

5.5 Accessing the Website

You can access the website using one of the following URLs:

- `https://localhost:8443`
- `http://localhost:8080`