

Company:

IGM Technology

2 Bloor St W, Toronto, ON M4W 3E2, Canada

Candidate name: Hans Garcia

Task: Spring Boot Test Task (SBTT)

Date: 23/09/2023

## Spring Boot Test Task Solution

### 1. Project Spring Boot and Java version

The Spring Boot version chosen for the development of the task is Spring Boot 3.1.4. as it is the latest version as of September 21st of 2023 considering doc.spring.io requirements the Java version used in this project is 17.

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>3.1.4</version>
  <relativePath/> <!-- lookup parent from repository -->
</parent>
<groupId>IGM.Technology</groupId>
<artifactId>sbtt</artifactId>
<version>1.0.0</version>
<name>SBTT</name>
<description>Spring Boot Test Task project assigned to and created by Hans Garcia</description>
<properties>
  <java.version>17</java.version>
</properties>
```

**Figure 1.** Project pom.xml file.  
IntelliJ.

### 2. Maven usage for the project

As it is asked the building tool used for the project is Maven in its latest version 3.9.4 as of august 3rd of 2023.

```
Apache Maven 3.9.4 (dfbb324ad4a7c8fb0bf182e6d91b0ae20e3d2dd9)
Maven home: C:\Program Files\Maven\apache-maven-3.9.4
Java version: 17.0.4.1, vendor: Oracle Corporation, runtime: C:\Program Files\Java\jdk-17.0.4.1
Default locale: es_CO, platform encoding: UTF-8
OS name: "windows 11", version: "10.0", arch: "amd64", family: "windows"
```

**Figure 2.** Maven mvn -v command as seen in console.  
IntelliJ.

### 3. REST endpoints

**Note:** Every path uses localhost:8080 as domain. The subsequent services that are inside the project use ports such as 8181 for HTML service, 8282 for third party API and 5005 for debugging. This last endpoint is set to debug with VSCode using launch.json file configuration.

**Note 2:** Whenever a call is made within the service please check IDE, Linux, and/or Docker logs to see the complete behavior of the project. E.g. When accessing data from third party API retry attempts are seen in IntelliJ IDE logs, for Docker under logs tab, for Linux when executing docker run command and is set as console for the project running in Spring Boot.

#### 3.a. Retrieving a HTML file – Parallelization – Endpoint

**Path:** “/sbtt/site”

As it is asked in the 3.a. section an endpoint is developed to retrieve an HTML file based on the logic proposed. The development of this endpoint required to mock the method for generating the HTML file this is accomplished using WireMock utility for Spring Boot. As stated in section 3.a.ii the idea of this endpoint is to parallelize generation. Thus, the development of this endpoint accomplishes this goal. As an extra or plus, and as it is for interest in computer science, this endpoint was developed to accomplish concurrency too. We will see this at the end of this section.

As is common within the development of spring boot projects, the development of the project was done with this logic and with OOP logic. First, we have our controller who is in charge of defining the two endpoints needed for the whole task. For the HTML file generation endpoint, we have under the general path “/sbtt” the path “/site”. This endpoint uses the Get HTTP method to retrieve the file from the mocked method.

```
@GetMapping("/site")
public ResponseEntity<String> generateNewHtml() {
    try {
        ResponseEntity<String> htmlContentFile = HTMLService.retrieveHTML();

        HttpStatus statusCode = htmlContentFile.getStatusCode();
        String responseBody = htmlContentFile.getBody();

        System.out.println("Status Code: " + statusCode);
        System.out.println("Response Body: " + responseBody);

        return htmlContentFile;
    } catch (Exception e) {
        String errorMessage = "Failed to retrieve HTML. Please try again.";
        return ResponseEntity.status(HttpStatus.INTERNAL_SERVER_ERROR).body(errorMessage);
    }
}
```

**Figure 3.** Project controller.  
IntelliJ.

Variable HTMLService is of type HTMLClientService. As we can see here services are made too following spring boot project structure. This service is in charge of making a call to another service

under port 8081, i.e., the port for the mocked method and bring ResponseEntity the response for this call. Path for this call is set as "html/generate".

```
public ResponseEntity<String> retrieveHTML() throws Exception {
    try {
        ResponseEntity<String> response = restTemplate.getForEntity( url: "http://localhost:8081/html/generate", String.class);
        System.out.println("Successful request");
        return response;
    } catch (Exception ex) {
        throw new Exception("Failed to retrieve HTML", ex);
    }
}
```

**Figure 4.** retrieveHTML method.

IntelliJ.

Then here the following steps work under WireMock logic to retrieve the HTML file. First here we develop three classes falling under all the endpoint logic. Those are the HTMLServerInitilizer.java, HTMLServerMappings.java and HTMLGeneratorTransformer.java classes. Here the main component for the whole purpose of the endpoint is in the HTMLGeneratorTransformer.java class. To parallelize generation, the concepts of Future and Executors are used to make our system work with a number of tasks depending on the available CPU cores of the machine. Logic followed to develop the main component is the following:

```
HTMLGenerationTransformer.java
// read all lines of the file
setCount(Files.lines(file).count());
} catch (Exception e) {
    e.printStackTrace();
}

setNumberOfHTMLLinesPerPiece(getCount() / numTasks);

ExecutorService executor = Executors.newScheduledThreadPool(numTasks);

// Create a list of CompletableFuture tasks for parallel generation
List<CompletableFuture<String>> tasks = IntStream.range(1, numTasks + 1) .IntStream
    .mapToObj(i -> CompletableFuture.supplyAsync(() -> generateHTMLPart(i), executor)) .Stream<CompletableFuture<String>>
    .collect(Collectors.toList());

// Wait for all tasks to complete and combine the HTML parts
CompletableFuture<Void> allOf = CompletableFuture.allOf(tasks.toArray(new CompletableFuture[0]));
allOf.join();

executor.shutdown();

String generatedHTML = tasks.stream() .Stream<CompletableFuture<String>>
    .map(CompletableFuture::join) .Stream<String>
    .collect(Collectors.joining());

return generatedHTML;
```

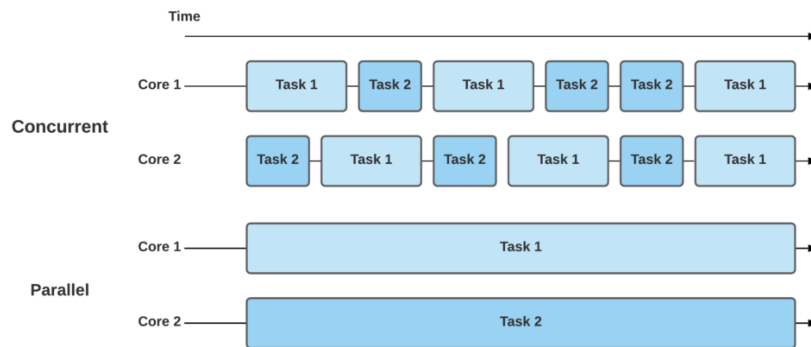
**Figure 5.** HTMLGeneratorTransformer.java class.

IntelliJ.

- Read and define number of tasks depending on available resources i.e., available number of cores (In example provided here we will work under the number of 16 available cores, taking into account that, as we will see, those are in fact our threads).
- Read a provided HTML and set the number of lines that file has (source: TemplateMo).
- Define the number of lines each task will be responsible for working with.
- Use iterably (repeated times) generateHTMLPart method. Here the code will generate our Futures (CompletableFutures) with every part of the HTML file generated.
- Wait until all our CompletableFutures end their tasks.
- Combine all the HTML file parts under the CompletableFutures.

- Generate the HTML File.

The result is retrieved then by the service and shown as Response Body for our endpoint. Here it is important to note that the code will work under the creation of a thread pool (`newScheduledThreadPool()` method) i.e., to achieve parallelization every task will work under one resource at the same time (this is parallelization by definition). In other words, for every resource available in our machine only one task will be completed.



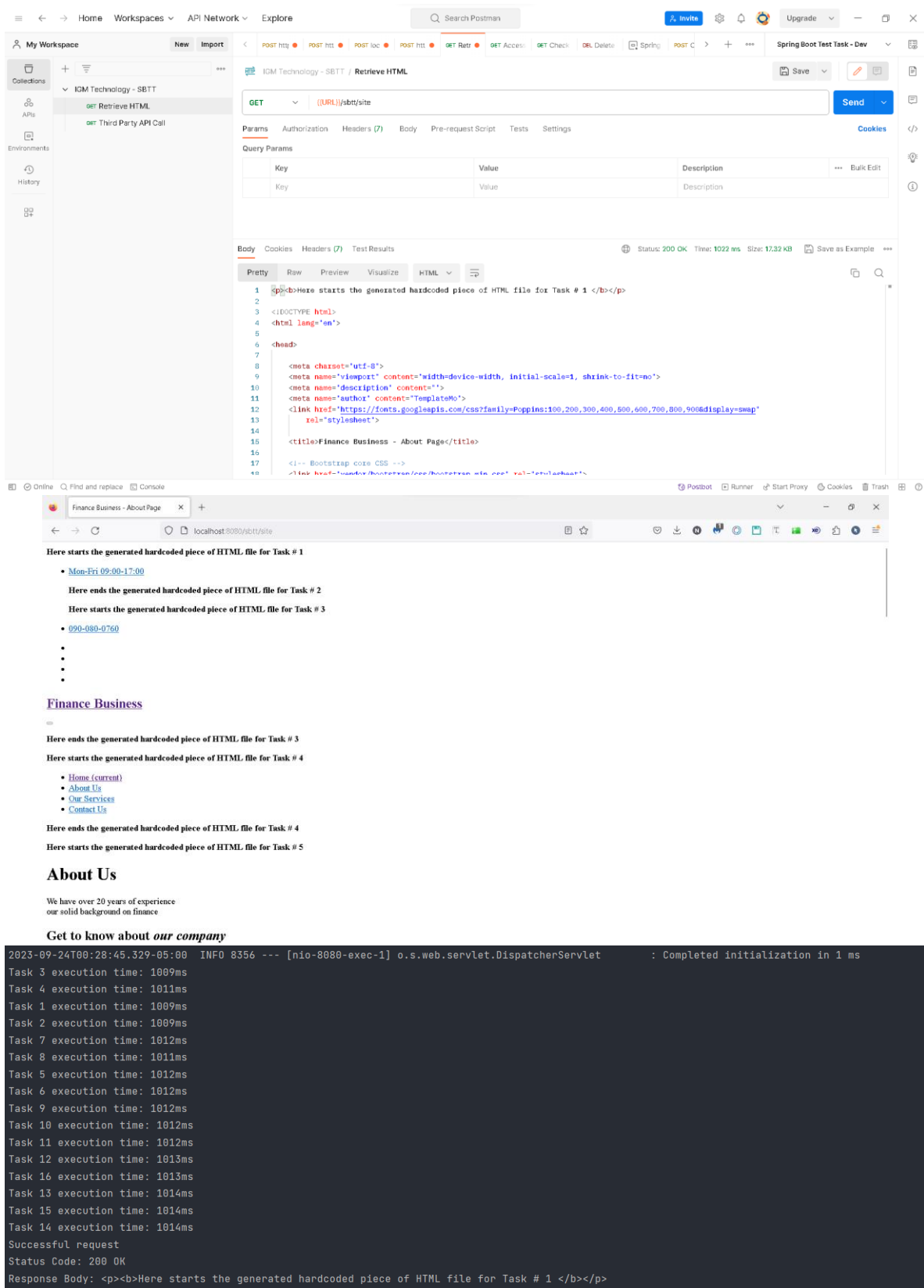
**Figure 6.** Parallelization vs Concurrency.

**Source:** Baeldung.

Also, that for every part of the HTML file that is generated a number of 1000 milliseconds in **delay** are set for every HTML file piece that is generated. A reader (`BufferedReader`) will be in charge of reading the file every time a part is generated, here in accordance to the task that is being executed the system will generate the part (or piece) according to the distribution of number of lines for each task that was previously set, this accomplishing that every task has the same or less amount of work depending on the extension of the HTML file.

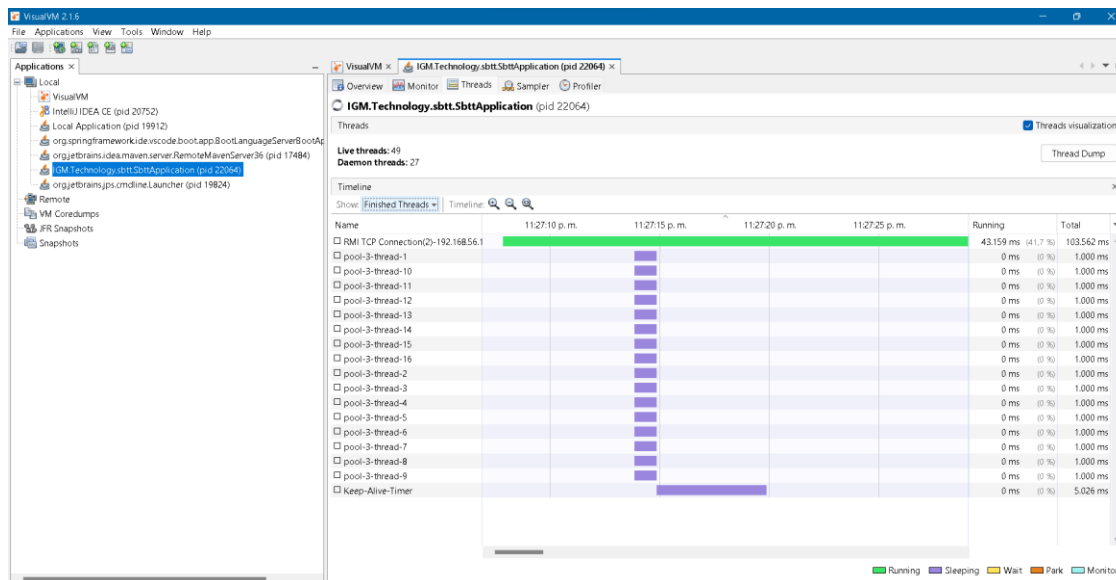
Summing up, using “sbtt/site” path the results are the following:

The results can be tracked when an API platform or browser loads the HTML generated by the Get method call as we can see in the following picture the results for Postman and Firefox tools.



**Figure 7.** HTML file retrieve endpoint results. Get call to `localhost:8080/sbtt/site`.  
Postman. Firefox. IntelliJ.

To test that we are under parallelization logic the VM environment is test under Oracle's VisualVM.



**Figure 8. Parallelization.**

HTML file retrieve endpoint results. Get call to localhost:8080/sbtt/site.

Testing parallelize generation of HTML file.

VisualVM.

As we can see at the same time every resource executed exactly just one task. As every task delays 1 second (1000 ms) we can confirm that every resource (core/thread) is doing just one task, thus by definition we accomplish parallelization. In accordance with the logic of the method and as it is set (without modifications) parallelization is guaranteed as HTML files will be split into the number of available cores, in this example the application worked with 16 cores.

### Extra - Concurrency

If we wanted to accomplish concurrency, as it is desired not only to execute multiple tasks each with one resource but to do multiple tasks with the same resource in separate times (definition of concurrency) we can achieve this by increasing the number of tasks a resource can do. If we do this with our code as it is set, we not only achieve concurrency but both parallelization and concurrency at the same time. Let us see:

Method `newScheduledThreadPool()` allows us to create multiple threads, if for every thread we increase the number of tasks it can handle we can achieve concurrency, as tasks are done by the parallelization configuration previously set, task will be synchronous and not asynchronous for pure concurrency. In the following changes we set the number of tasks in 32 tasks and the available Cores the remain the same as 16 cores.

```

public String generateHTML() throws ExecutionException, InterruptedException {
    // Set number of availableCores
    int availableCores = Runtime.getRuntime().availableProcessors();
    // Define the number of tasks based on CPU cores
    int numTasks = 32;

    String filePath = "src/main/resources/templates/about.html";

    try {
        // make a connection to the file
        Path file = Paths.get(filePath);
        // read all lines of the file
        setCount(Files.lines(file).count());
    } catch (Exception e) {
        e.printStackTrace();
    }

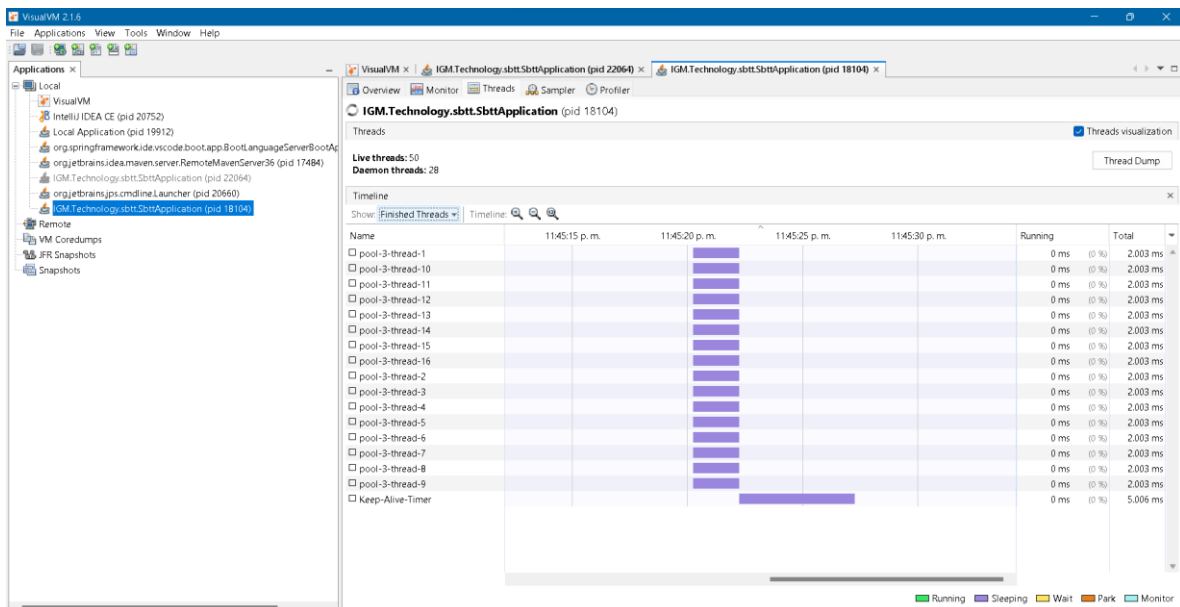
    setNumberOfHTMLLinesPerPiece(getCount() / numTasks);

    ExecutorService executor = Executors.newScheduledThreadPool(availableCores);
}

```

**Figure 9.** Concurrency.  
Changes to code to achieve concurrency too.  
IntelliJ.

As we can see concurrency is achieved when multiple tasks are done by every CPU core, we have available. To confirm this in the following picture we can note that for every CPU core exactly two tasks were executed (note that the total time of accomplishing 1 task has a delay of exactly 1 second or 1000 milliseconds).



```

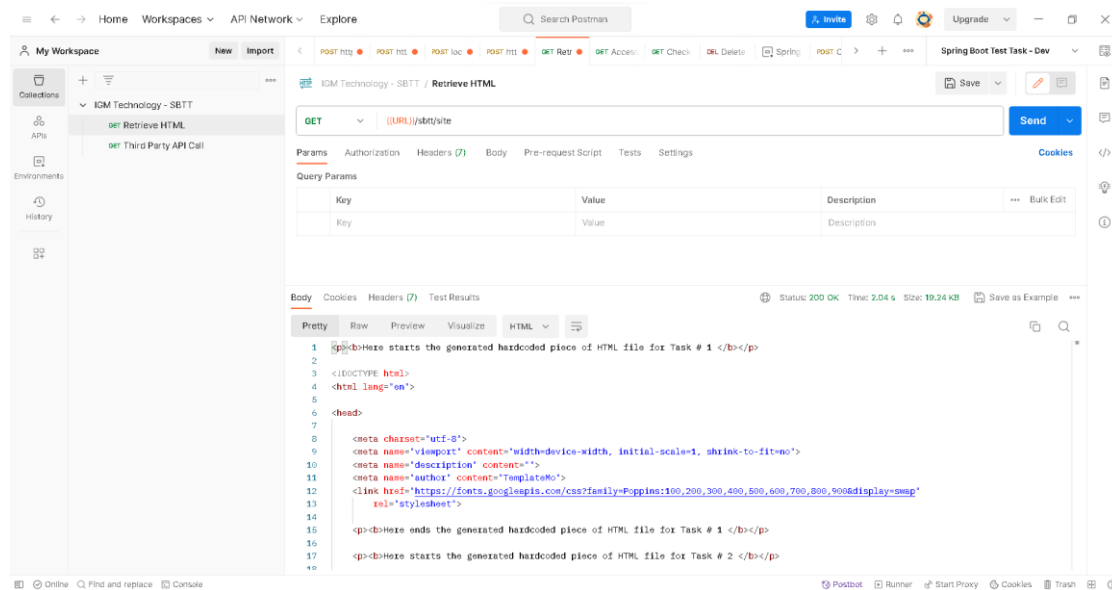
long startTime = System.currentTimeMillis();

try {
    Thread.sleep( millis: 1000); // Simulate delay
} catch (InterruptedException e) {
    Thread.currentThread().interrupt();
}

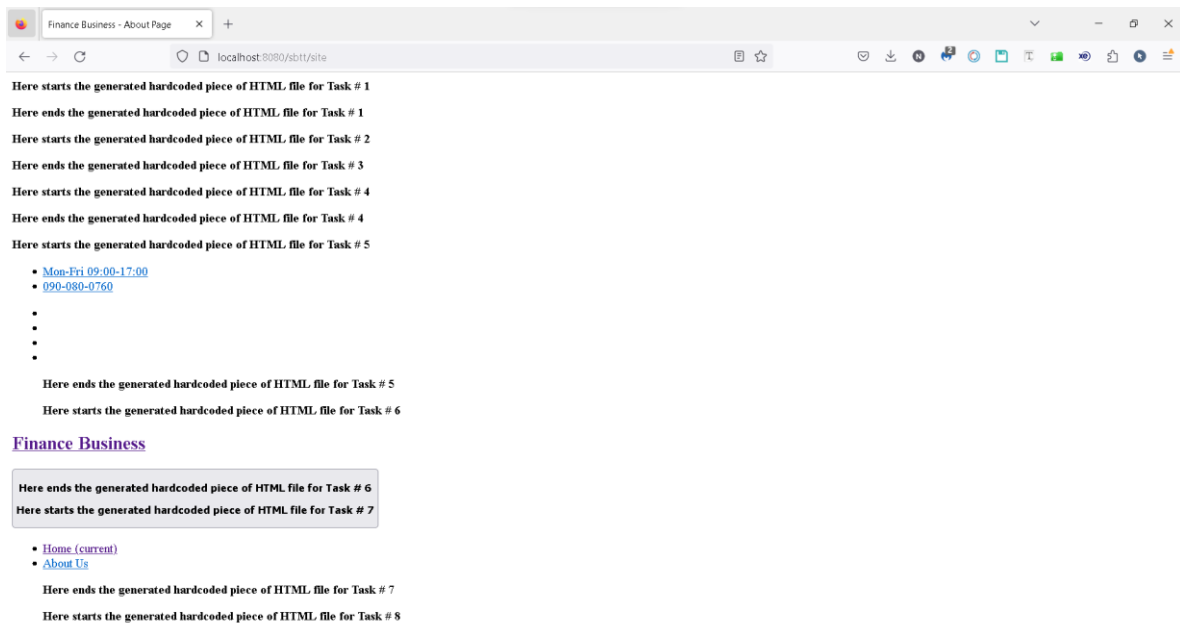
```

**Figure 10.** Concurrency.  
Changes to code to achieve concurrency too.  
Delay time for every task completed.  
VisualVM.

The results can be tracked when an API platform or browser loads the HTML generated by the Get method call as we can see in the following picture the results for Postman and Firefox tools.





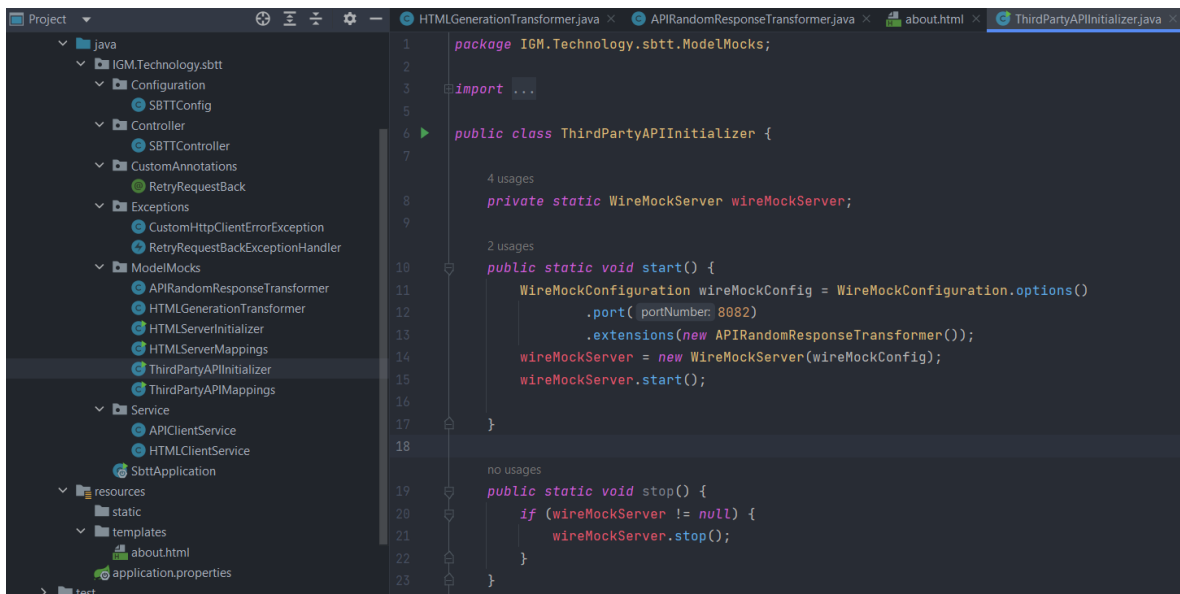


**Figure 11.** HTML file retrieve endpoint results for achieving concurrency.  
Get call to localhost:8080/sbtt/site.  
Postman. Firefox.

### 3.b. Third party API data retrieve - Endpoint

Path: “/sbtt/fetch-some-data”

As it is asked in section 3.b. an endpoint is developed to communicate through a request with an API for accessing some data. The data that this endpoint accesses is data from a hypothetical user from some API. Here to mock the method that will simulate the third-party API will be achieved using WireMock again.



**Figure 12.** Third Party API mock method classes. Project Structure.  
IntelliJ.

Under the following classes our API will work: ThirdPartyAPIInitializer.java and ThirdPartyAPIMappings.java, and APIRandomResponseTransformer.java.

To meet the requirements asked in sections 3.b.i, 3.b.ii and 3.b.iii the main logic is carried out by APIRandomResponseTransformer.java class. As it is required with certain frequency the API returns status 429 to simulate that third party API has rate limiter. This by setting a probability of 60% to get a 429 status and a probability of 40% to get a 200 status. This is to see easily how the retry and backoff mechanism will work.

```
@Override
public Response transform(Request request, Response response, FileSource files, Parameters parameters) {
    int randomValue = random.nextInt( bound: 100);

    final SimpleDateFormat TSPFormat = new SimpleDateFormat( pattern: "yyyy-MM-dd'T'HH:mm:ss.SSSXXX");
    Timestamp timestamp = new Timestamp(System.currentTimeMillis());
    Timestamp timeElapsed = new Timestamp(System.currentTimeMillis() + 2000);

    if (randomValue < 40) { // 40% probability for a 200 response
        return Response.Builder.like(response)
            .status(200)
            .body("{\"metadata\": {\"request_time\": \"\" + TSPFormat.format(timestamp) + "\", \"response_time\": \"\" +
            .build();
    } else { // 60% probability for a 429 response
        return Response.Builder.like(response)
            .status(429)
            .body("{\"error\": [{\"metadata\": {\"request_time\": \"\" + TSPFormat.format(timestamp) + "\", \"response_
            .build();
    }
}
```

**Figure 13.** Third party API mechanism for 429 status returning.  
IntelliJ.

The custom annotation RetryRequestBack and its exceptions CustomHttpClientErrorException.java and RetryRequestBackExceptionHandler.java are created to accomplish retrying requests with a backoff mechanism. The maximum number of retry attempts is set to 3 attempts. The backoff is set to make a retry with a delay of 1000 milliseconds with a multiplier of 2 and a maximum delay of 4000 milliseconds. So, every attempt will increase the time by a factor of 2 to make another retry.

```

1 package IGM.Technology.sbtt.CustomAnnotations;
2
3 import ...
4
5 2 usages
6
7 @Target(ElementType.METHOD)
8 @Retention(RetentionPolicy.RUNTIME)
9 @Retryable(
10     maxAttempts = 3,           // Maximum number of retry attempts
11     backoff = @Backoff(
12         delay = 1000,          // Initial delay in milliseconds
13         multiplier = 2,        // Delay multiplier for exponential backoff
14         maxDelay = 4000        // Maximum delay in milliseconds
15     ),
16     exclude = {CustomHttpClientErrorException.class},
17     include = {ExhaustedRetryException.class}
18 )
19
20 public @interface RetryRequestBack {
21     no usages
22     @AliasFor(annotation = Retryable.class)
23     Class<? extends Throwable>[] value() default {};
24 }

```

**Figure 14.** Custom annotation `RetryRequestBack.java`.  
IntelliJ.

To get every response to the requests made to the API a JSON format is used for each request: Whether we get a 200 status or a 429-status response. Error handling is made so that in services such as Postman we can test the API responses.

**Note:** Error handling and response for successful calls is defined so we follow a structure of headers, metadata, and body structure.

Let us see the results of the development. Here we will make a Get call to the third-party API using “sbtt/fetch-some-data” path.

The results can be tracked when an API platform or browser loads the request to the third-party API by the Get method call as we can see in the following picture the results for Postman and Firefox tools.

### Successful data access – 200 status

Here we can see the access to data from a hypothetical user called John Doe. As stated, it follows a JSON structure where when building up is shown in the following way as pictured here.

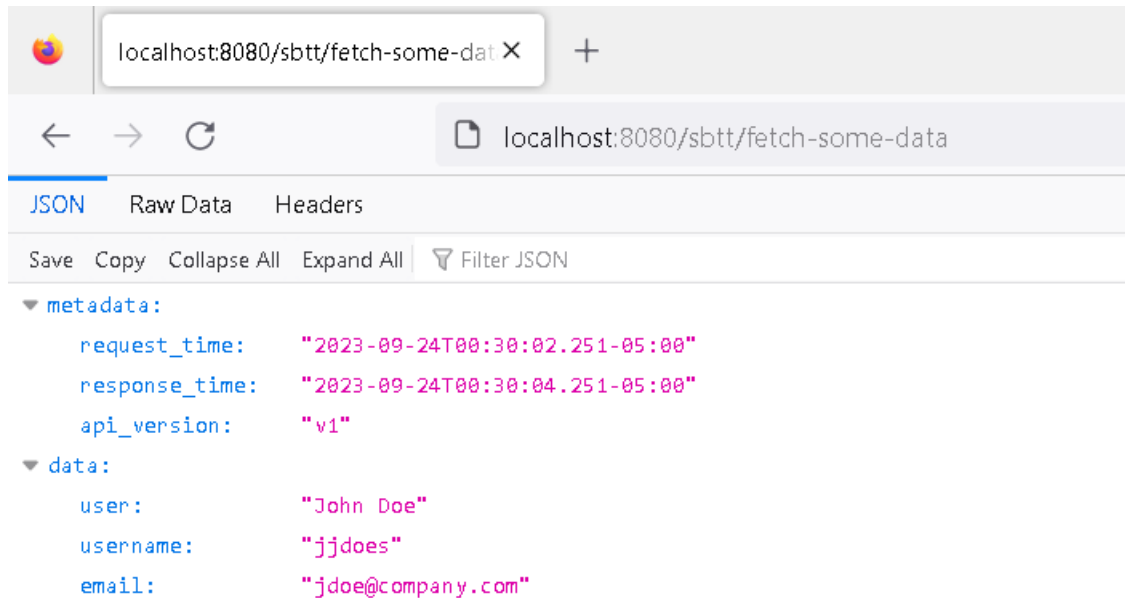
Body Cookies Headers (7) Test Results Status: 200 OK Time: 1014 ms Size: 459 B Save as Example

Pretty Raw Preview Visualize JSON

```

1 {
2   "metadata": {
3     "request_time": "2023-09-24T00:35:22.107-05:00",
4     "response_time": "2023-09-24T00:35:24.107-05:00",
5     "api_version": "v1"
6   },
7   "data": {
8     "user": "John Doe",
9     "username": "jjdoes",
10    "email": "jdoe@company.com"
11  }
12 }

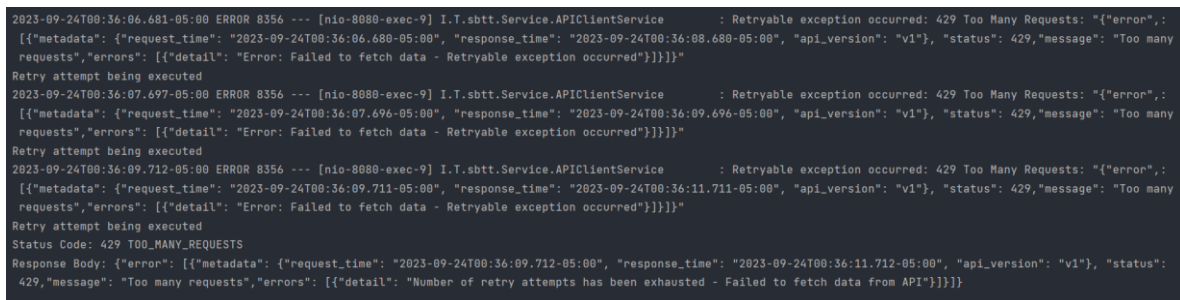
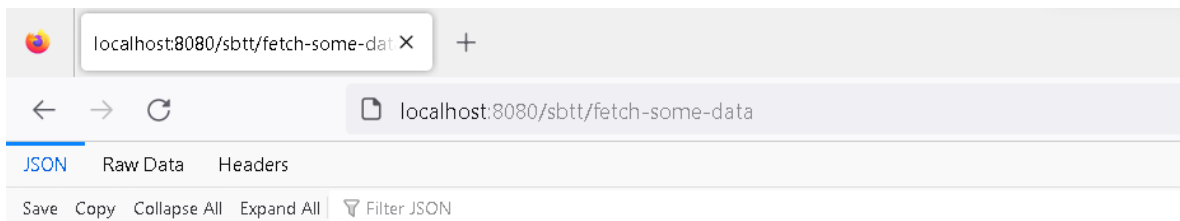
```



**Figure 15.** Request for accessing some data from third party API. Successful response. 200 status.  
Get call to localhost:8080/sbtt/site.  
Postman. Firefox.

### Unsuccessful data access – 429 status – rate limiter

Now here we can see the access to data from a hypothetical user called John Doe when it is unsuccessful as the API with certain frequency returns 429 status. As stated, it follows a JSON structure where when building up is shown in the following way as pictured here. Here we can see the exhaustion of all the three retry attempts. It is important to note that every retry attempt has a 40% chance of getting a successful response and 60% chance of getting an unsuccessful response as result of the rate limiter. Timestamps are set to match current time. Hypothetical response time is set to match 2000 milliseconds after response time do not confuse with the time of backoff mechanism, it is set merely for presentation and etiquette purposes.



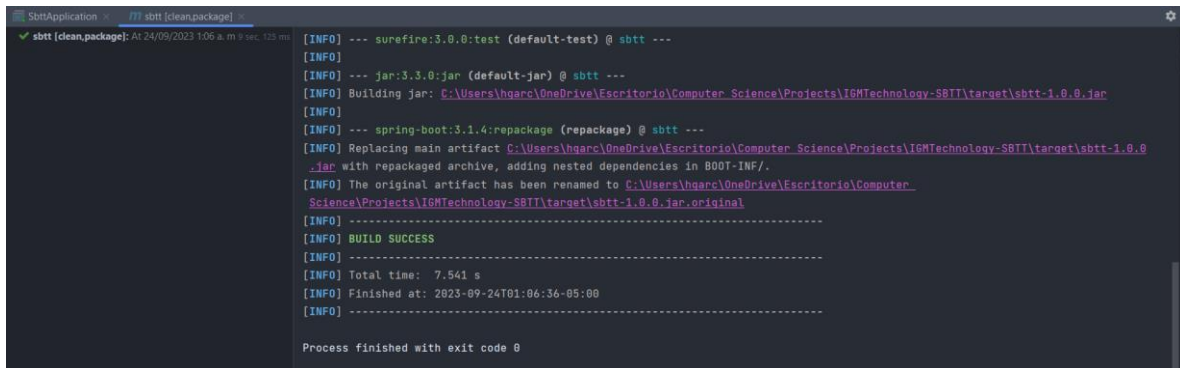
**Figure 16.** Request for accessing some data from third party API. Unsuccessful response. 429 status. Too many requests. Complete max number of attempts made.

Get call to localhost:8080/sbtt/site.

Postman. Firefox. IntelliJ.

#### 4. Docker support

Now that we have gone under code and building stages (using Java, Spring Boot and Maven), let us say testing and release stages were made with tools for instance such as JUnit and Jenkins, we can now go under deployment stage, so let us use Docker to containerize our application.



```
✓ sbtt [clean:package] At 24/09/2023 1:06 a. m 4 sec 123 ms
[INFO] --- surefire:3.0.0:test (default-test) @ sbtt ---
[INFO] --- jar:3.3.0:jar (default-jar) @ sbtt ---
[INFO] Building jar: C:\Users\hgarc\OneDrive\Escritorio\Computer Science\Projects\IGHTechnology-SBTT\target\sbt-1.0.0.jar
[INFO] --- spring-boot:3.1.4:repackage (repackage) @ sbtt ---
[INFO] Replacing main artifact C:\Users\hgarc\OneDrive\Escritorio\Computer Science\Projects\IGHTechnology-SBTT\target\sbt-1.0.0.jar with repackaged archive, adding nested dependencies in BOOT-INF/.
[INFO] The original artifact has been renamed to C:\Users\hgarc\OneDrive\Escritorio\Computer Science\Projects\IGHTechnology-SBTT\target\sbt-1.0.0.jar.original
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 7.541 s
[INFO] Finished at: 2023-09-24T01:06:36-05:00
[INFO] -----
Process finished with exit code 0
```

**Figure 17.** Maven. Packaging and building of project. IntelliJ.

First, we create a Dockerfile for the project. Here we define the instructions that will allow our services to run in a containerized environment. First, we use an image of OpenJDK to work with our current Java version (17), next we can copy the contents of our packaged and built application to the container. This can be achieved by using the COPY command in the docker file. Then we can set CMD parameters to run our java services such as "java", "-jar", and the name of our packaged application .jar. Finally we can EXPOSE the port in which our service will run, for the purpose of showing all the services that run in the application the docker file is set to expose more than one port, so port 8080 is set for the service, 8181 for the HTML built-in generation service and 8282 for the third party API service.

In addition to these services to debug the container, I expose port 5005 and use ARG and ENV command parameters to set JAVA\_OPTS for debugging in Visual Studio Code. In this way when something goes wrong with the up and running container, we can debug it so we can fix it whenever it is needed.

Considering this, this is our Docker file:

```
Dockerfile x launch.json x
1 FROM openjdk:17
2 ARG JAVA_OPTS
3 ENV JAVA_OPTS=$JAVA_OPTS
4 ENV JAVA_TOOL_OPTIONS -agentlib:jdwp=transport=dt_socket,address=0.0.0.0:5005,server=y,suspend=n
5 WORKDIR /app
6 COPY src/main/resources src/main/resources
7 COPY ./target/sbtt-1.0.0.jar /app
8 CMD ["java", "-jar", "sbtt-1.0.0.jar"]
9 EXPOSE 8080 8081 8082
10 EXPOSE 5005
```

Figure 18. Docker. Docker file structure.  
IntelliJ.

The following pictures denote the process of building the image of our service and running the container with the intended ports. At the end we can see the debug support for the container included with VSCode IDE. Commands docker build and docker run are set under Linux OS where docker containers work.

```
h@hpc:~/OneDrive/Escritorio/Computer Science/Projects/IONechnology-SBTT$ docker build -t sbtt-application .
[+] Building 2.1s (9/9) FINISHED
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 362B
=> [internal] load .dockerignore
=> => transferring context: 2B
=> [internal] load metadata for docker.io/library/openjdk:17
=> [1/4] FROM docker.io/library/openjdk:17
=> [internal] load build context
=> => transferring context: 49.77MB
=> CACHED [2/4] WORKDIR /app
=> CACHED [3/4] COPY src/main/resources src/main/resources
=> [4/4] COPY ./target/sbtt-1.0.0.jar /app
=> exporting to image
=> => exporting layers
=> => writing image sha256:9912e3dc618c7a8e26852925b7b8f8abdb3e1b6cf377be7f2507371b7fa75c9
=> => naming to docker.io/library/sbtt-application
```

Images [Give feedback](#)

Local Hub Artifactory **EARLY ACCESS**

0 Bytes / 533.32 MB in use 3 images Last refresh: 6 hours ago

<input type="checkbox"/>	Name	Tag	Status	Created	Size	Actions
<input type="checkbox"/>	<a href="#">sbtt-application</a> 9912e3dc618c	latest	Unused	1 second ago	521.24 MB	<a href="#">▶</a> <a href="#">⋮</a> <a href="#">🗑</a>
<input type="checkbox"/>	<a href="#">docker/logs-explorer-extension</a> 6264a9fd6396	0.2.3	Unused	4 months ago	12.07 MB	<a href="#">▶</a> <a href="#">⋮</a> <a href="#">🗑</a>
<input type="checkbox"/>	<a href="#">openjdk</a> 5e28bs2b4cdb	17	Unused	1 year ago	471.46 MB	<a href="#">▶</a> <a href="#">⋮</a> <a href="#">🗑</a>

```
root@b811:~# docker run --name sbttapp -p 8080:8080 -p 8181:8181 -p 5282:5282 -p 5005:5005 sbtt-application
Picked up JAVA_TOOL_OPTIONS: -agentlib:jdwp=transport=dt_socket,address=0.0.0.0:5005,server=y,suspend=n
Listening for transport dt_socket at address: 5005

=====
:: Spring Boot ::
(v3.1.4)

2023-09-24T06:08:10.713Z INFO 1 --- [main] IGM.Technology.sbtt.SbttApplication : Starting SbtApplication v1.0.0 using Java 17.0.2 with PID 1 (/app/sbtt-1.0.0.jar started by root in /app)
2023-09-24T06:08:10.716Z INFO 1 --- [main] IGM.Technology.sbtt.SbttApplication : No active profile set, falling back to 1 default profile: "default"
2023-09-24T06:08:12.036Z INFO 1 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 8080 (http)
2023-09-24T06:08:12.046Z INFO 1 --- [main] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2023-09-24T06:08:12.047Z INFO 1 --- [main] o.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/10.1.13]
2023-09-24T06:08:12.126Z INFO 1 --- [main] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext
2023-09-24T06:08:12.129Z INFO 1 --- [main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 1339 ms
2023-09-24T06:08:12.576Z INFO 1 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8080 (http) with context path ''
2023-09-24T06:08:12.697Z INFO 1 --- [main] IGM.Technology.sbtt.SbttApplication : Started SbtApplication in 2.511 seconds (process running for 3.14)
2023-09-24T06:08:13.387Z INFO 1 --- [main] org.eclipse.jetty.server.Server : jetty-11.0.16; built: 2023-08-25T19:43:30.438Z; git: bedff458c4dd1a716d59e17b8cb0d2842eeab291; jvm 17.0.2+8-B8
2023-09-24T06:08:13.436Z INFO 1 --- [main] o.e.j.s.h.ContextHandler$Root : Started o.e.j.s.ServletContextHandler@106faf11(/_admin,null,AVAILABLE)
2023-09-24T06:08:13.440Z INFO 1 --- [main] o.e.j.s.h.ContextHandler$Root : RequestHandlerClass from context returned com.github.tomakehurst.wiremock.http.StubRequestHandler. Normalized map
ped under returned 'null'
2023-09-24T06:08:13.446Z INFO 1 --- [main] o.e.j.s.h.ContextHandler$Root : Started o.e.j.s.ServletContextHandler@4b770e40(/,null,AVAILABLE)
2023-09-24T06:08:13.458Z INFO 1 --- [main] o.e.j.s.h.ContextHandler$Root : Started NetworkTrafficServerConnector@aec50a1(HTTP/1.1, (http/1.1, h2c))[0.0.0.0:8081]
2023-09-24T06:08:13.462Z INFO 1 --- [main] org.eclipse.jetty.server.Server : Started Server@1af7f54a[STARTING][11.0.16,sto=1000] @9006ms
2023-09-24T06:08:13.788Z INFO 1 --- [qtp795273218-50] o.e.j.s.h.ContextHandler$Root : RequestHandlerClass from context returned com.github.tomakehurst.wiremock.http.AdminRequestHandler. Normalized map
ped under returned 'null'
2023-09-24T06:08:13.978Z INFO 1 --- [main] org.eclipse.jetty.server.Server : jetty-11.0.16; built: 2023-08-25T19:43:30.438Z; git: bedff458c4dd1a716d59e17b8cb0d2842eeab291; jvm 17.0.2+8-B8
2023-09-24T06:08:13.978Z INFO 1 --- [main] o.e.j.s.h.ContextHandler$Root : Started o.e.j.s.ServletContextHandler@4fe81805(/_admin,null,AVAILABLE)
2023-09-24T06:08:13.978Z INFO 1 --- [main] o.e.j.s.h.ContextHandler$Root : RequestHandlerClass from context returned com.github.tomakehurst.wiremock.http.StubRequestHandler. Normalized map
ped under returned 'null'
2023-09-24T06:08:13.978Z INFO 1 --- [main] o.e.j.s.h.ContextHandler$Root : Started o.e.j.s.ServletContextHandler@95120f99(/,null,AVAILABLE)
2023-09-24T06:08:13.978Z INFO 1 --- [main] o.e.j.s.h.ContextHandler$Root : Started NetworkTrafficServerConnector@6813a331(HTTP/1.1, (http/1.1, h2c))[0.0.0.0:8082]
2023-09-24T06:08:13.980Z INFO 1 --- [main] org.eclipse.jetty.server.Server : Started Server@6d28bcd5[STARTING][11.0.16,sto=1000] @9423ms
2023-09-24T06:08:13.980Z INFO 1 --- [qtp855501888-59] o.e.j.s.h.ContextHandler$Root : RequestHandlerClass from context returned com.github.tomakehurst.wiremock.http.AdminRequestHandler. Normalized map
ped under returned 'null'
```

Containers [Give feedback](#)

Container CPU usage 📊





0.16% / 1000% (10 cores allocated)


Container memory usage 📊

337.4MB / 30.31GB

[Show charts](#) ⌵

☰ 🔌 Only show running containers

<input type="checkbox"/>	Name	Image	Status	CPU (%)	Port(s)	Last started	Actions
<input type="checkbox"/>	<div><div> sbttapp</div><div>a45061694bc2 </div></div>	sbtt-application	Running	0.16%	<a href="#">5005:5005</a>  <a href="#">Show all ports (4)</a>	32 seconds ago	<div><span>⌵</span> <span>⋮</span> </div>



localhost:8080/ ✕ +

⬅ ➡ 🔄

🔒 📄 localhost:8080

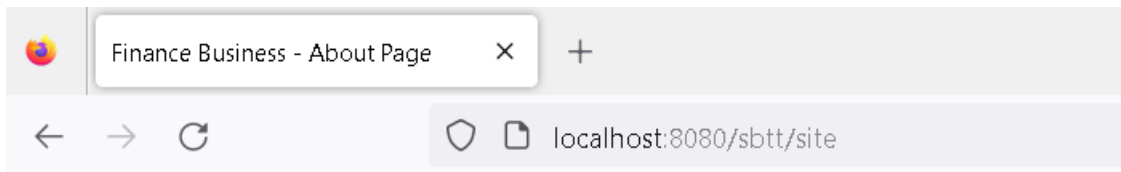
# Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as a fallback.

Sun Sep 24 06:08:59 UTC 2023

There was an unexpected error (type=Not Found, status=404).





**Here starts the generated hardcoded piece of HTML file for Task # 1**

- [Mon-Fri 09:00-17:00](#)

**Here ends the generated hardcoded piece of HTML file for Task # 2**

**Here starts the generated hardcoded piece of HTML file for Task # 3**

- [090-080-0760](#)

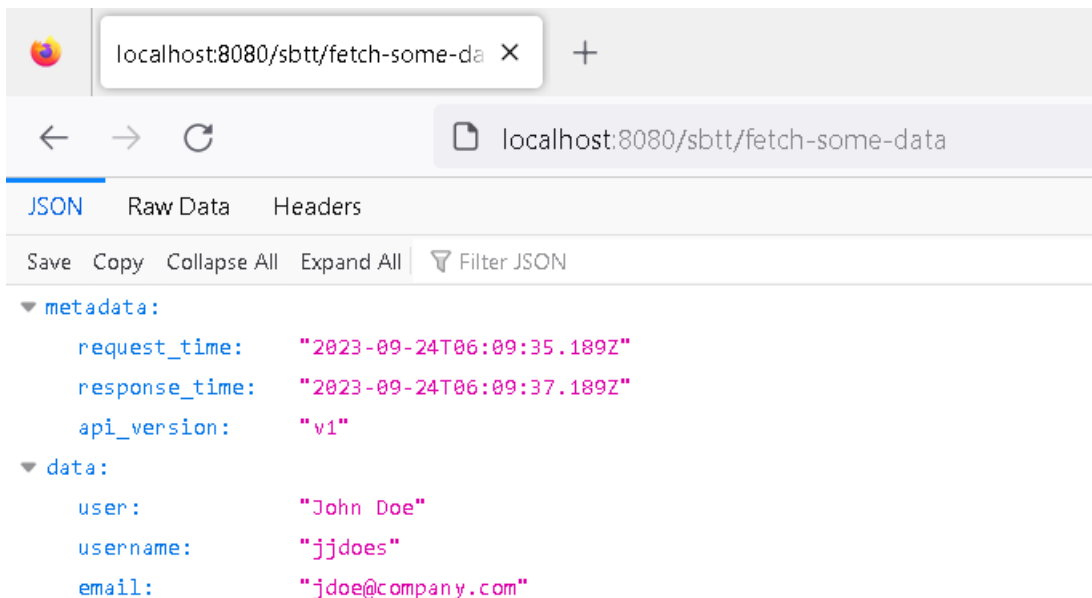
- 
- 
- 
- 

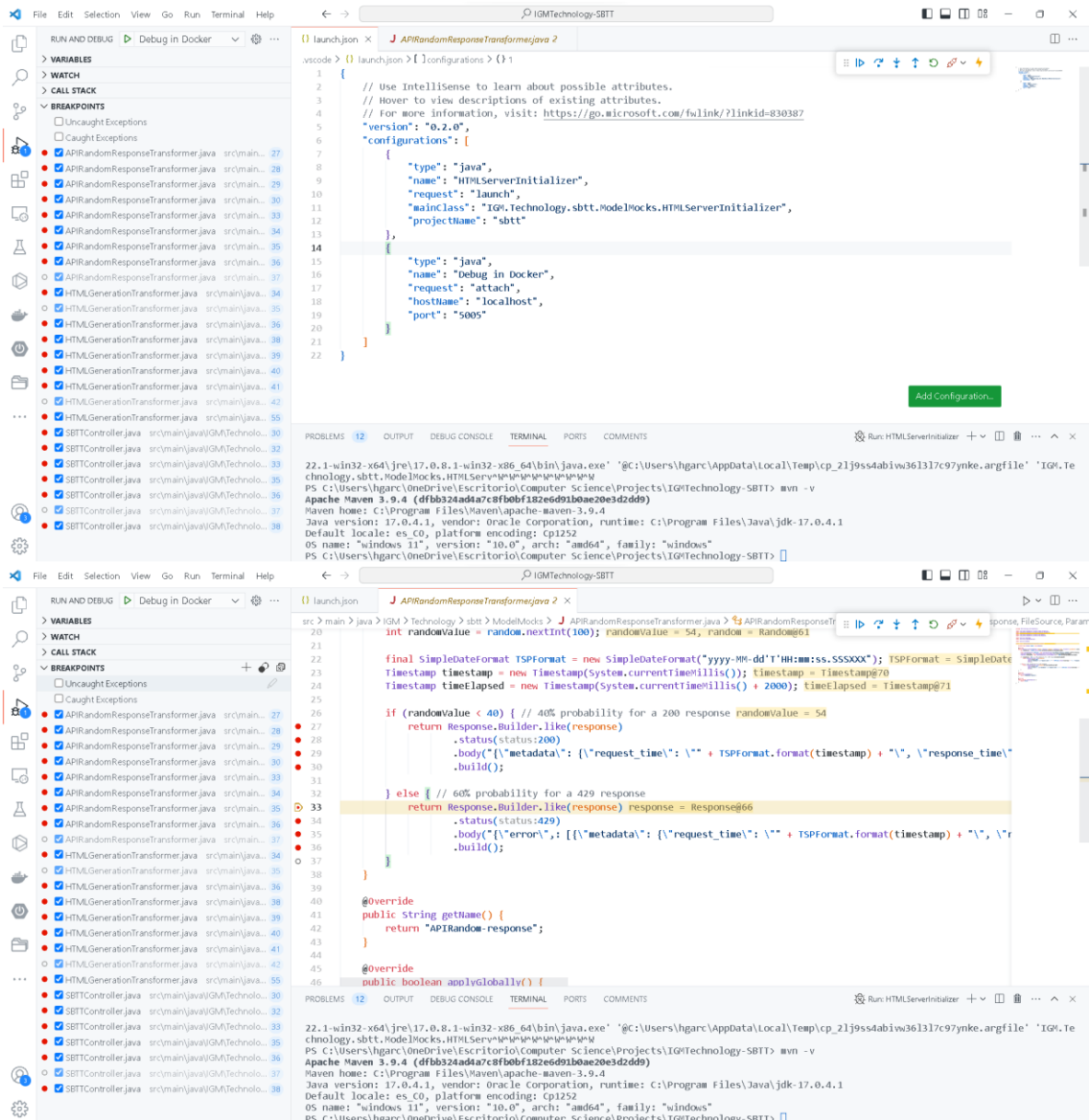
## Finance Business

—

**Here ends the generated hardcoded piece of HTML file for Task # 3**

**Here starts the generated hardcoded piece of HTML file for Task # 4**





**Figure 20.** Docker support. Process of building and running application image and container.

Support for VSCode debugging included.

PowerShell. Linux. Docker. Firefox. IntelliJ. VSCode.