DEV Challenge

Thanks for considering my application for this position. This document highly recorded my effort to play with the interview coding challenge.

1. Overview

In this exercise, these technologies would be applied:

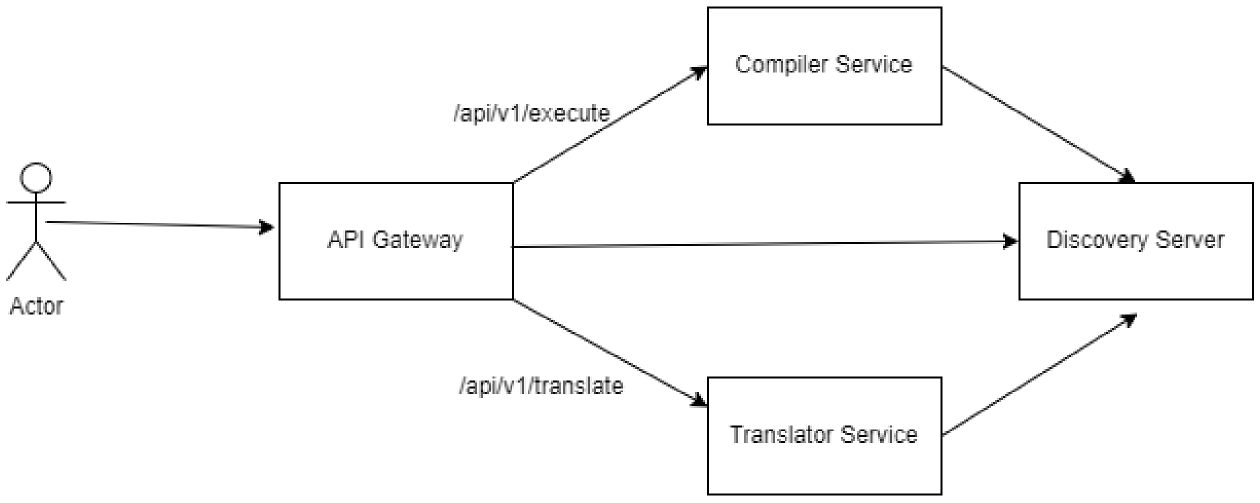
- Backend: Java Spring Boot.
- Frontend: ReactJS.

The microservice pattern also be provided in this assignment.

The port numbers for each service:

Service	Port
API-GATEWAY	8080
COMPILER-SERVICE	8081
TRANSLATOR-SERVICE	8082
Web UI	3000 (in development) 80 (run with docker-compose)

Here is the backend architecture:



Here, the API Gateway acts as a gatekeeper and also be the entry point to access our services. Therefore we can access both compiler service and translator service via port 8080.

To run the application on your machine, please navigate to **backend/dev-challenge** folder and run these commands:

- **mvn compile jib:dockerBuild**
- **docker-compose up -d**

2. Code Interpreter

For the code interpreter, we can utilize the existed API to request for code execution.

For further details, please navigate to this repo: https://github.com/Jaagrav/CodeX-API and the issue link: https://github.com/Jaagrav/CodeX-API/issues/55

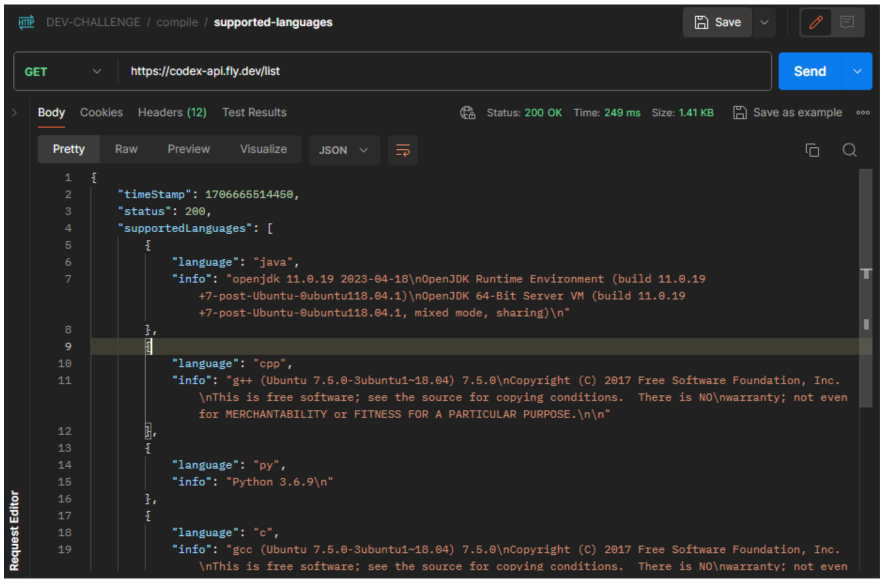
So, besides Java or Kotlin, other languages like C++, Python, Golang or JavaScript are also be supported. Therefore, we can expand the diversity of our code editor.

Before deep diving into writing code for compiler-service, let expore what we can get from this API repo.

Due to the issue "API stopped working" so the API endpoint we use to investigate is: https://codex-api.fly.dev/

2.1. Exploring third-party APIs

- **Get all languages supported**

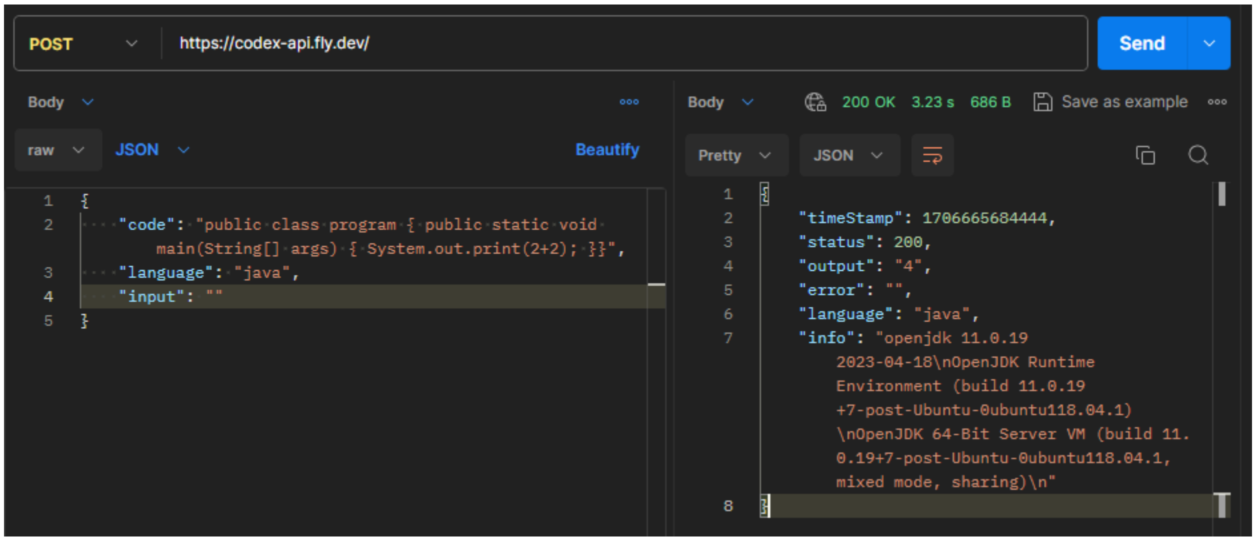


```
GET https://codex-api.fly.dev/list
```

```
{
    "timeStamp": 1706665514450,
    "status": 200,
    "supportedLanguages": [
        {
            "language": "java",
            "info": "openjdk 11.0.19 2023-04-18\nOpenJDK Runtime Environment (build 11.0.19
                +7-post-Ubuntu-0ubuntu118.04.1)\nOpenJDK 64-Bit Server VM (build 11.0.19
                +7-post-Ubuntu-0ubuntu118.04.1, mixed mode, sharing)\n"
        },
        {
            "language": "cpp",
            "info": "g++ (Ubuntu 7.5.0-3ubuntu1~18.04) 7.5.0\nCopyright (C) 2017 Free Software Foundation, Inc.
                \nThis is free software; see the source for copying conditions.  There is NO\nwarranty; not even
                for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.\n\n"
        },
        {
            "language": "py",
            "info": "Python 3.6.9\n"
        },
        {
            "language": "c",
            "info": "gcc (Ubuntu 7.5.0-3ubuntu1~18.04) 7.5.0\nCopyright (C) 2017 Free Software Foundation, Inc.
                \nThis is free software; see the source for copying conditions.  There is NO\nwarranty; not even
```

Here, we can see many languages are supported including C, C++, Java, Python and so on.

- **Execute code**


POST https://codex-api.fly.dev/

```
{
    "code": "public class program { public static void main(String[] args) { System.out.print(2+2); }}",
    "language": "java",
    "input": ""
}
```

200 OK 3.23 s 686 B

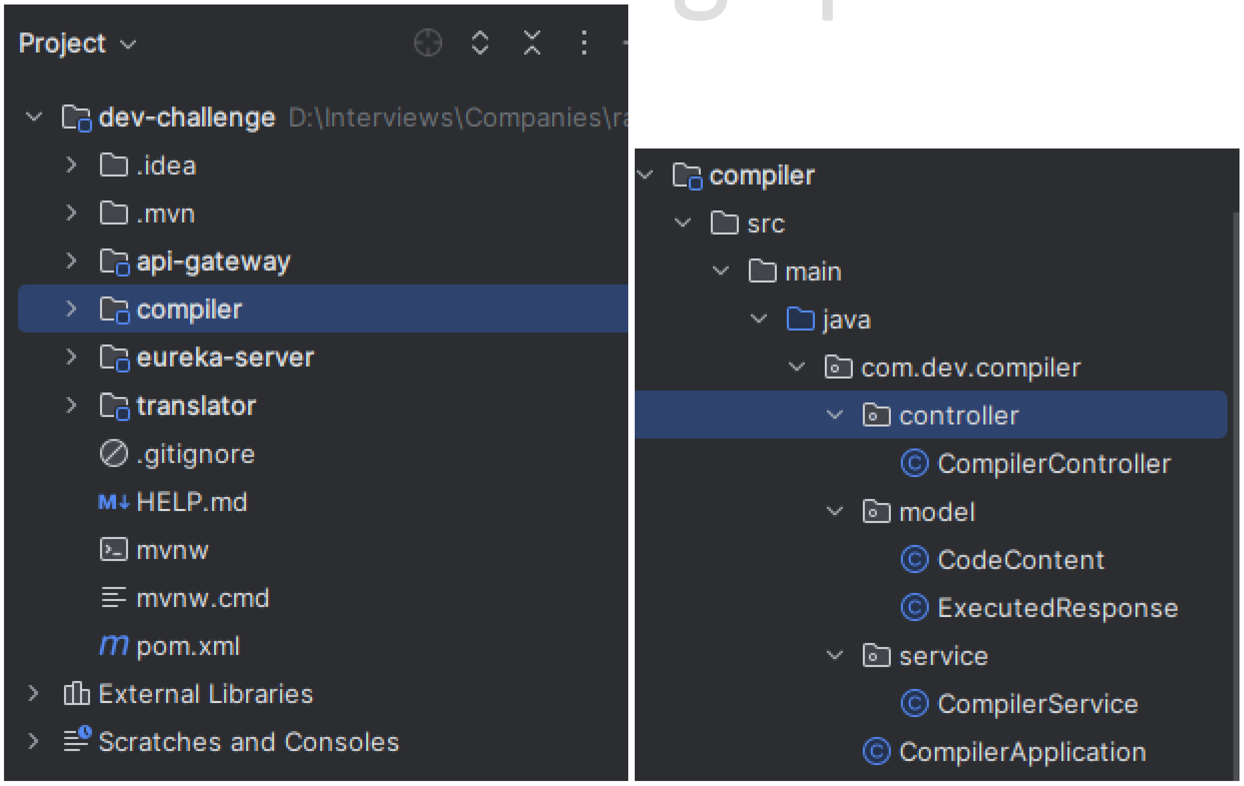
```
{
    "timeStamp": 1706665684444,
    "status": 200,
    "output": "4",
    "error": "",
    "language": "java",
    "info": "openjdk 11.0.19 2023-04-18\nOpenJDK Runtime Environment (build 11.0.19+7-post-Ubuntu-0ubuntu118.04.1)\nOpenJDK 64-Bit Server VM (build 11.0.19+7-post-Ubuntu-0ubuntu118.04.1, mixed mode, sharing)\n"
}
```

We can try the provided simple lines of code, in this case, we use Java and try to calculate the sum of 2 + 2. The output is 4 in the response.

After exploring these APIs we now can build our service to consume them and return response for clients.

2.2. Build Backend Spring Boot service

We will build the microservice project including API Gateway, Discovery server and some of our services. To use the interpreter/compiler, we build the **compiler** service



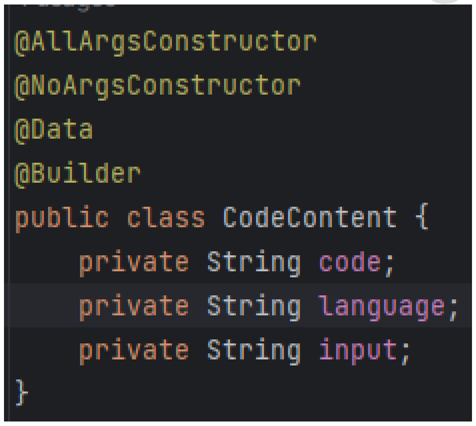
Definition of application configuration:

```
application.executing-endpoint=https://codex-api.fly.dev/

spring.application.name=compiler-service
server.port=8081

eureka.client.service-url.defaultZone=http://localhost:8761/eureka
eureka.client.fetch-registry=true
eureka.client.register-with-eureka=true
eureka.client.enabled=true
eureka.instance.hostname=localhost
eureka.instance.prefer-ip-address=true
```

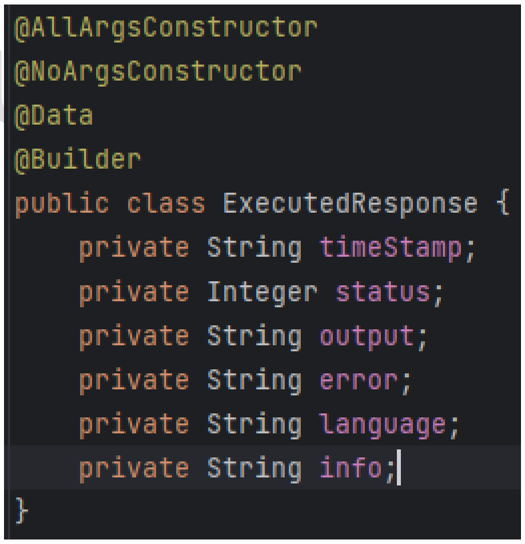
Inside the compiler, we have:

Packages	**Functionality**
controller	Listens to requests and return responses
model	Contains request and response DTOs
service	Encapsulates the business logic – utilize CodeX APIs to excuted code

The snippet codes for the data transferred objects:



```
@AllArgsConstructor
@NoArgsConstructor
@Data
@Builder
public class CodeContent {
    private String code;
    private String language;
    private String input;
}
```



```
@AllArgsConstructor
@NoArgsConstructor
@Data
@Builder
public class ExecutedResponse {
    private String timeStamp;
    private Integer status;
    private String output;
    private String error;
    private String language;
    private String info;
}
```

Then, define the service class to use CodeX Apis with the supported OkHttp library:

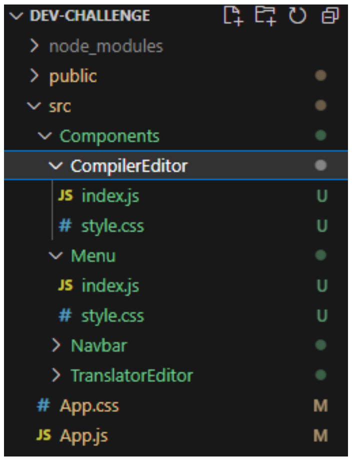
```
@Service
@RequiredArgsConstructor
public class CompilerService {
    private final ObjectMapper mapper;

    @Value("${application.executing-endpoint}")
    private String endpoint;

    public ExecutedResponse execute(CodeContent codeContent) throws
IOException {
```

```java
        OkHttpClient client = new OkHttpClient().newBuilder()
                .build();
        MediaType mediaType = MediaType.parse("application/json");

        String codeContentJson = mapper.writeValueAsString(codeContent);
        RequestBody body = RequestBody.create(mediaType, codeContentJson);
        Request request = new Request.Builder()
                .url(endpoint)
                .method("POST", body)
                .addHeader("Content-Type", "application/json")
                .build();

        String result;
        try (Response response = client.newCall(request).execute()) {
            assert response.body() != null;
            result = response.body().string();
        }

        return mapper.readValue(result, ExecutedResponse.class);
    }
```

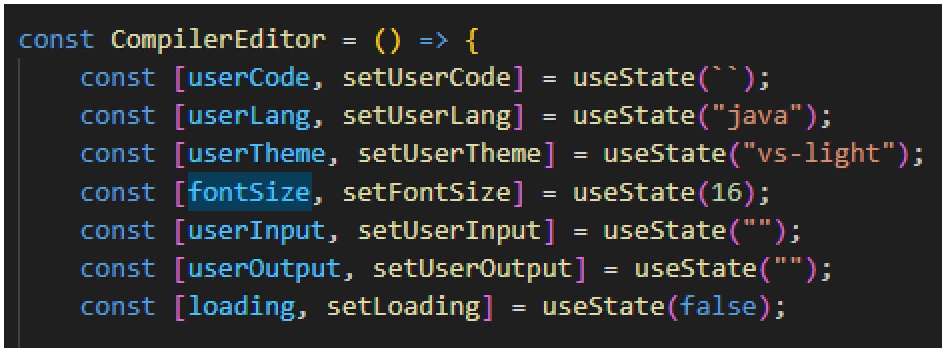
2.3. Build user interface

We can applied react library to build the website UI contains 2 separate fields to excute code and get back the results.

We organize the React app into multiple components. In this section, we build the CompileEditor component.



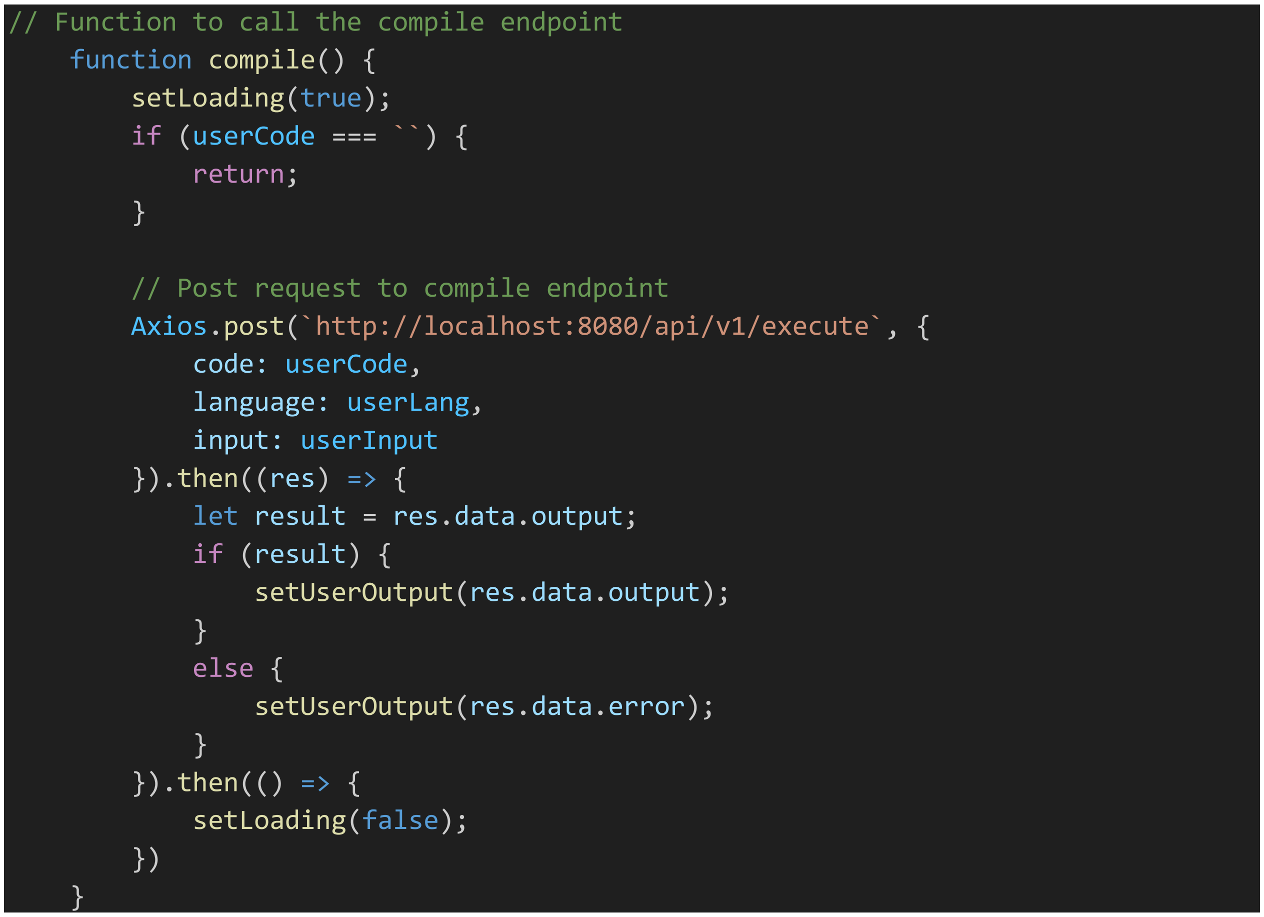
```
DEV-CHALLENGE
  > node_modules
  > public                 ●
  v src                    ●
    v Components           ●
      v CompilerEditor     ●
          index.js         U
          style.css        U
      v Menu               ●
          index.js         U
          style.css        U
      > Navbar             ●
      > TranslatorEditor   ●
    App.css                M
    App.js                 M
```



```
const CompilerEditor = () => {
    const [userCode, setUserCode] = useState(``);
    const [userLang, setUserLang] = useState("java");
    const [userTheme, setUserTheme] = useState("vs-light");
    const [fontSize, setFontSize] = useState(16);
    const [userInput, setUserInput] = useState("");
    const [userOutput, setUserOutput] = useState("");
    const [loading, setLoading] = useState(false);
```

And we also manage the states of this component, which would be user code, user input, language that user uses (default is Java) and so on.

The key point here is that applying Axios library can help us make request to backend to execute the code and then get the result to be the user's output:



```
// Function to call the compile endpoint
    function compile() {
        setLoading(true);
        if (userCode === ``) {
            return;
        }

        // Post request to compile endpoint
        Axios.post(`http://localhost:8080/api/v1/execute`, {
            code: userCode,
            language: userLang,
            input: userInput
        }).then((res) => {
            let result = res.data.output;
            if (result) {
                setUserOutput(res.data.output);
            }
            else {
                setUserOutput(res.data.error);
            }
        }).then(() => {
            setLoading(false);
        })
    }
```

3. Language Translator

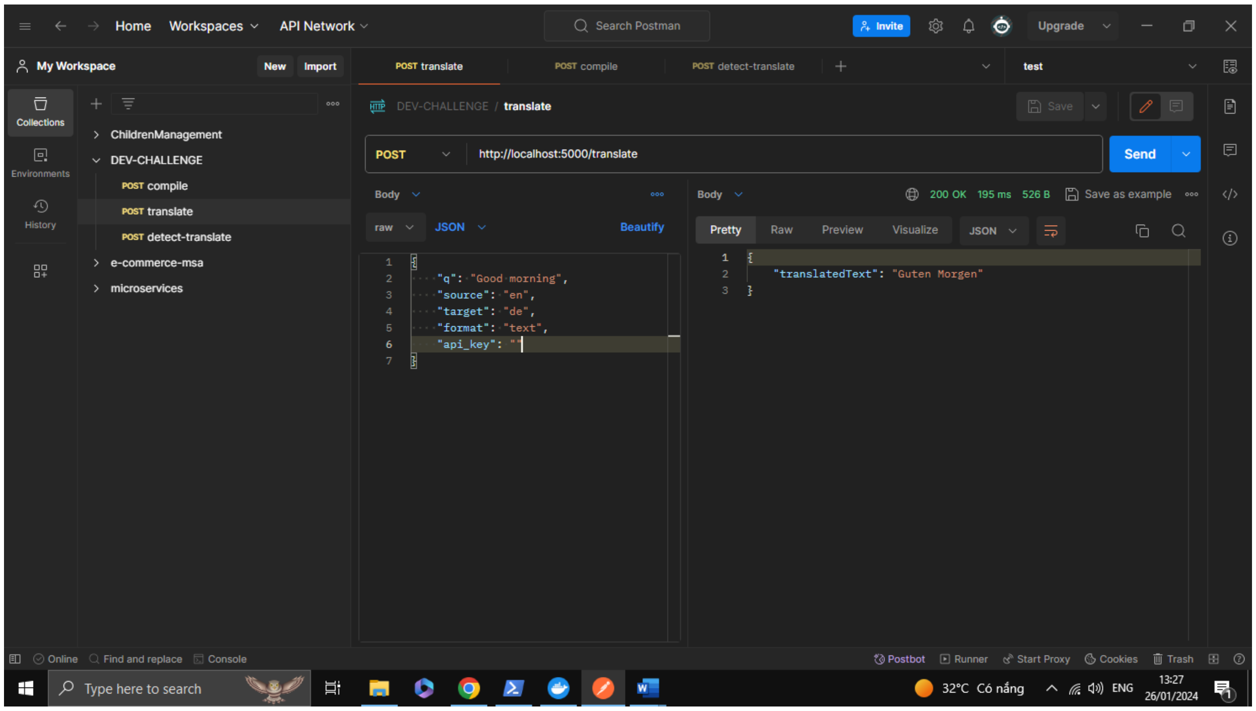
For language translation, beside translate between English and German, we can translate more languages to each other thanks for **LibreTranslate**: https://github.com/cmooredev/LibreTranslate/tree/main

LibreTranslate also gives us a bunch of APIs to get supported languages and translate languages.

We can use this endpoint https://libretranslate.de/ for the simplicity for making API call. However, if this endpoint sometimes faced with timeout exception, we call run docker container locally via command ***docker run -ti --rm -p 5000:5000 libretranslate/libretranslate*** and make request calls to this container.

3.1. Explore APIs

3.1.1. Use translator with defined source language:

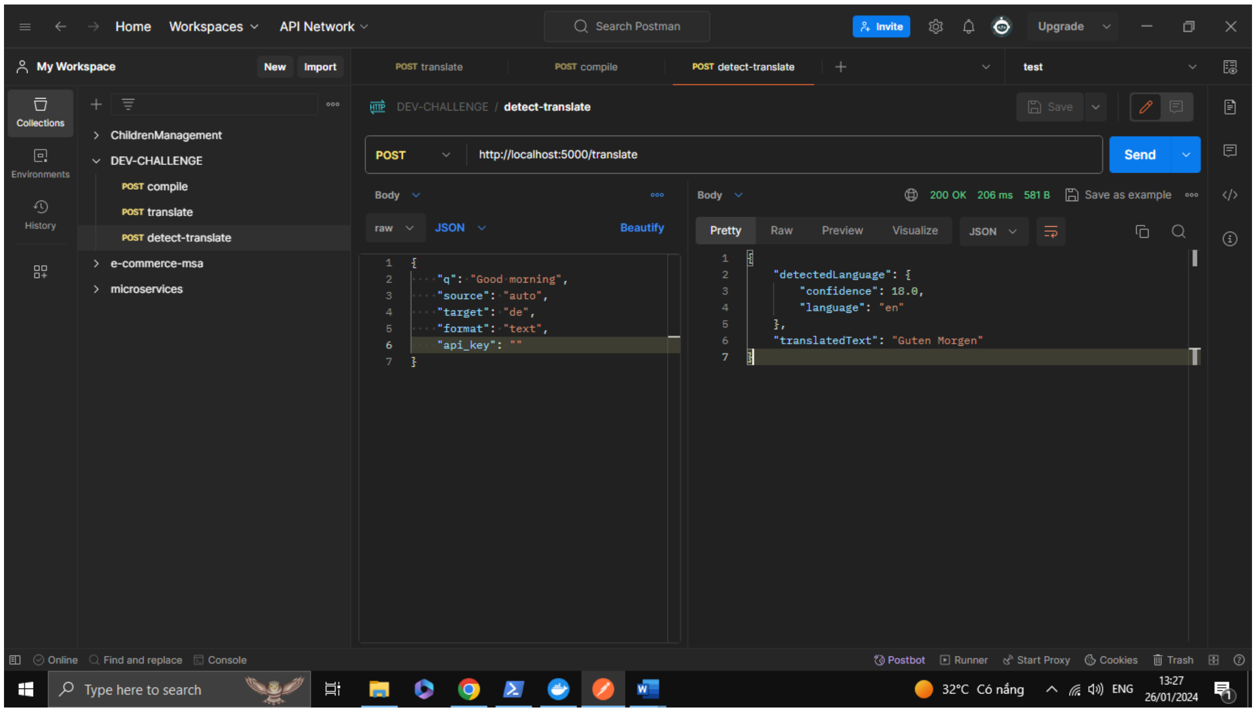


```
POST http://localhost:5000/translate
```

```
{
    "q": "Good morning",
    "source": "en",
    "target": "de",
    "format": "text",
    "api_key": ""
}
```

```
{
    "translatedText": "Guten Morgen"
}
```

3.1.2. Use translator API with auto-detect source language:



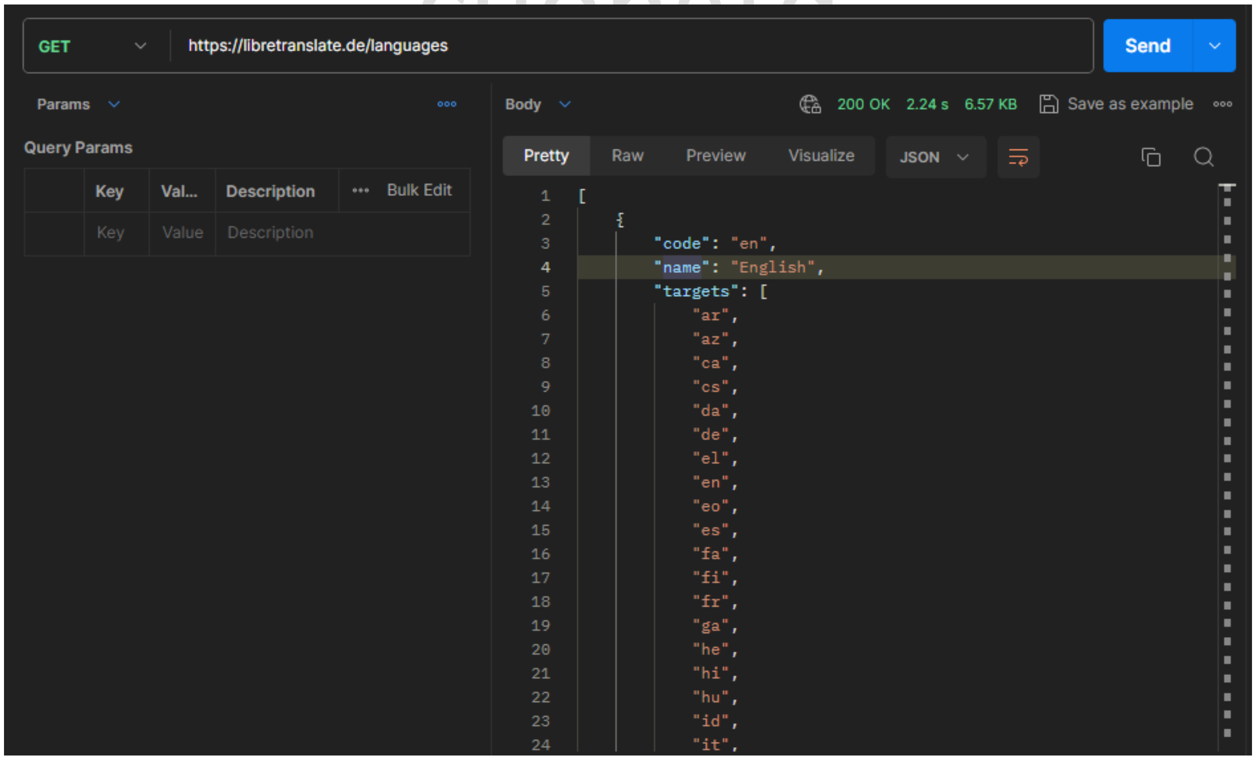
POST http://localhost:5000/translate

```
{
    "q": "Good morning",
    "source": "auto",
    "target": "de",
    "format": "text",
    "api_key": ""
}
```

```
{
    "detectedLanguage": {
        "confidence": 18.0,
        "language": "en"
    },
    "translatedText": "Guten Morgen"
}
```

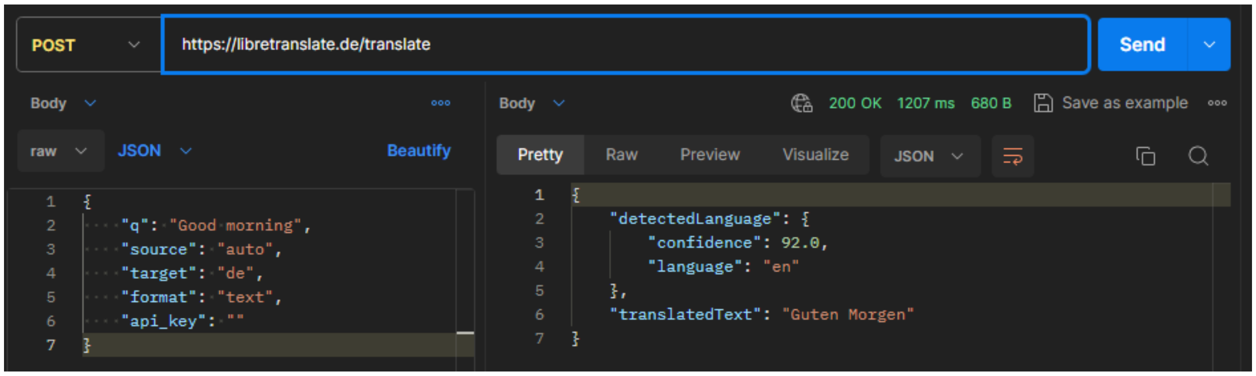
We can also receive the detected language information in the response.

In addition, https://libretranslate.de can also be used for simplicity.



GET https://libretranslate.de/languages

```
[
    {
        "code": "en",
        "name": "English",
        "targets": [
            "ar",
            "az",
            "ca",
            "cs",
            "da",
            "de",
            "el",
            "en",
            "eo",
            "es",
            "fa",
            "fi",
            "fr",
            "ga",
            "he",
            "hi",
            "hu",
            "id",
            "it",
```


POST https://libretranslate.de/translate

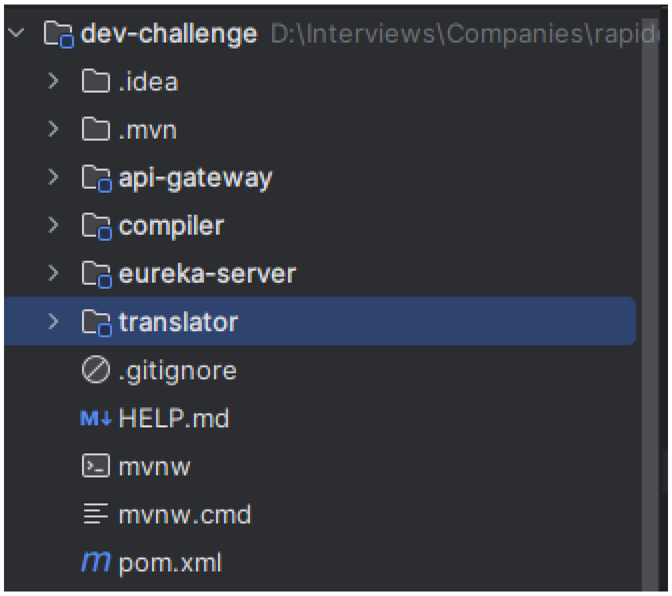
Body raw JSON

```
{
    "q": "Good morning",
    "source": "auto",
    "target": "de",
    "format": "text",
    "api_key": ""
}
```

200 OK 1207 ms 680 B

```
{
    "detectedLanguage": {
        "confidence": 92.0,
        "language": "en"
    },
    "translatedText": "Guten Morgen"
}
```

3.2. Build Backend Spring Boot service



```
dev-challenge D:\Interviews\Companies\rapid
  .idea
  .mvn
  api-gateway
  compiler
  eureka-server
  translator
  .gitignore
  HELP.md
  mvnw
  mvnw.cmd
  pom.xml
```

The **translator** service will take responsibility for translating languages.

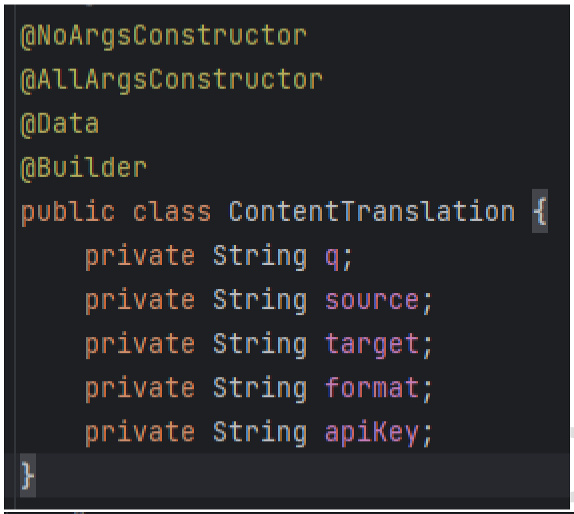
Definitions of application configurations:

```
#application.executing-endpoint=https://libretranslate.de

# Sometimes, the above endpoint will be down, start docker and use this
endpoint as alternative approach
application.executing-endpoint=http://localhost:5000

spring.application.name=translate-service
server.port=8082

eureka.client.service-url.defaultZone=http://localhost:8761/eureka
eureka.client.fetch-registry=true
eureka.client.register-with-eureka=true
eureka.client.enabled=true
eureka.instance.hostname=localhost
eureka.instance.prefer-ip-address=true
```

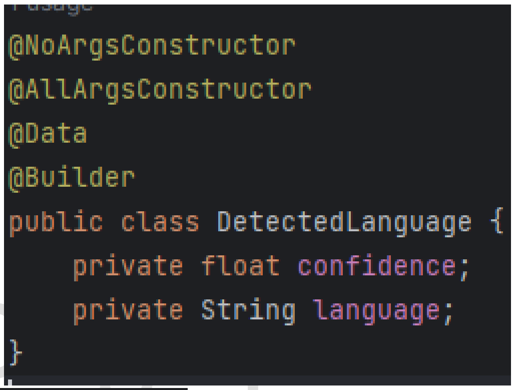
Similar to the compile service, we also structure our service into the layers like controller, service and model:

Packages	Functionality
controller	Listens to requests and return responses
model	Contains request and response DTOs
service	Encapsulates the business logic – utilize LibreTranslate APIs to excuted code

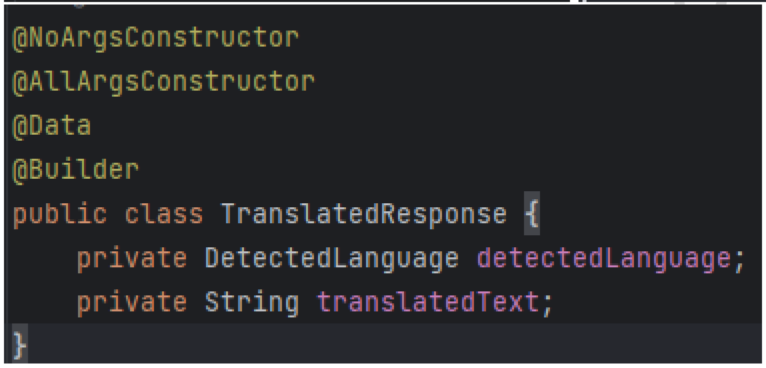
The snippet codes for the data transferred objects:



```java
@NoArgsConstructor
@AllArgsConstructor
@Data
@Builder
public class ContentTranslation {
    private String q;
    private String source;
    private String target;
    private String format;
    private String apiKey;
}
```



```java
@NoArgsConstructor
@AllArgsConstructor
@Data
@Builder
public class DetectedLanguage {
    private float confidence;
    private String language;
}
```



```java
@NoArgsConstructor
@AllArgsConstructor
@Data
@Builder
public class TranslatedResponse {
    private DetectedLanguage detectedLanguage;
    private String translatedText;
}
```

Let define the translate method inside **TranslateService** class to consume LibreTranslate API and process translation:

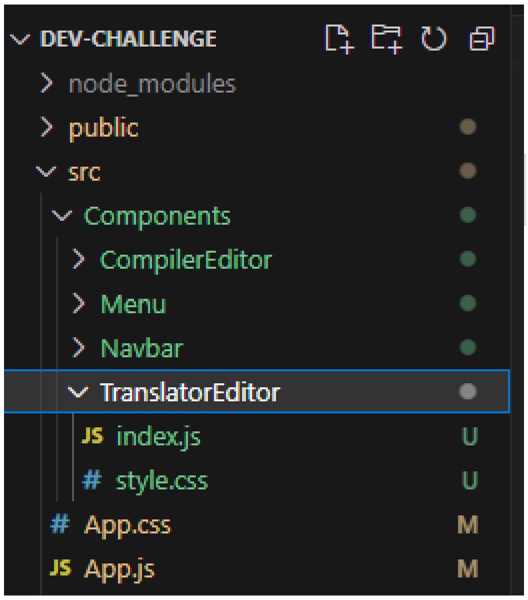
```java
public TranslatedResponse translate(ContentTranslation contentTranslation)
throws IOException {
    String translateUrl = endpoint + "/translate";
    MediaType mediaType = MediaType.parse("application/json");

    String codeContentJson = mapper.writeValueAsString(contentTranslation);
    RequestBody body = RequestBody.create(mediaType, codeContentJson);
    Request request = new Request.Builder()
            .url(translateUrl)
            .method("POST", body)
            .addHeader("Content-Type", "application/json")
            .build();

    String result;
    try (Response response = client.newCall(request).execute()) {
        assert response.body() != null;
        result = response.body().string();
    }

    return mapper.readValue(result, TranslatedResponse.class);
}
```

We can also define a similar method for getting all supported languages.

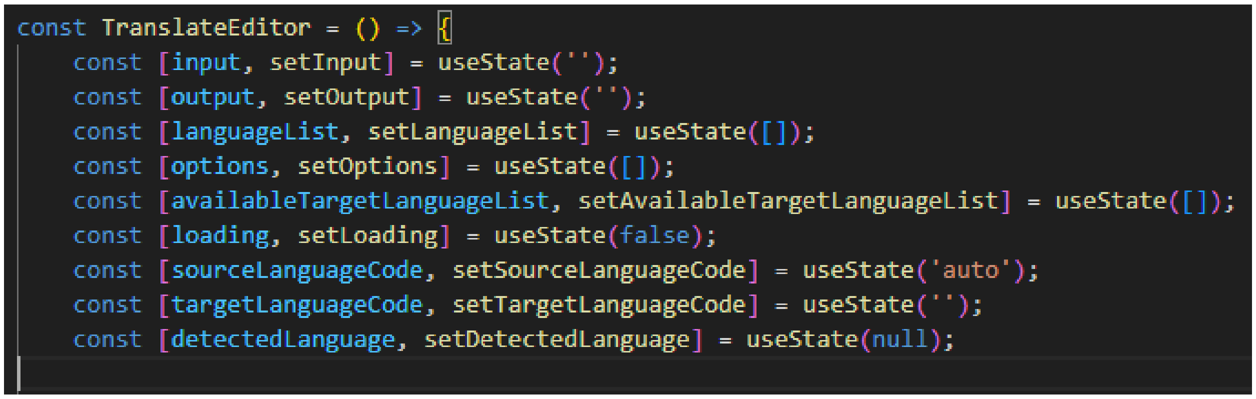
3.3. Build user interface

We can applied react library to build the website UI contains 2 separate fields to input source text and translate it to the target langugage.

We organize the React app into multiple components. In this section, we build the **TranslatorEditor** component.



And we also manage the states of this component, which would be input, output text, source language, target language and so on.



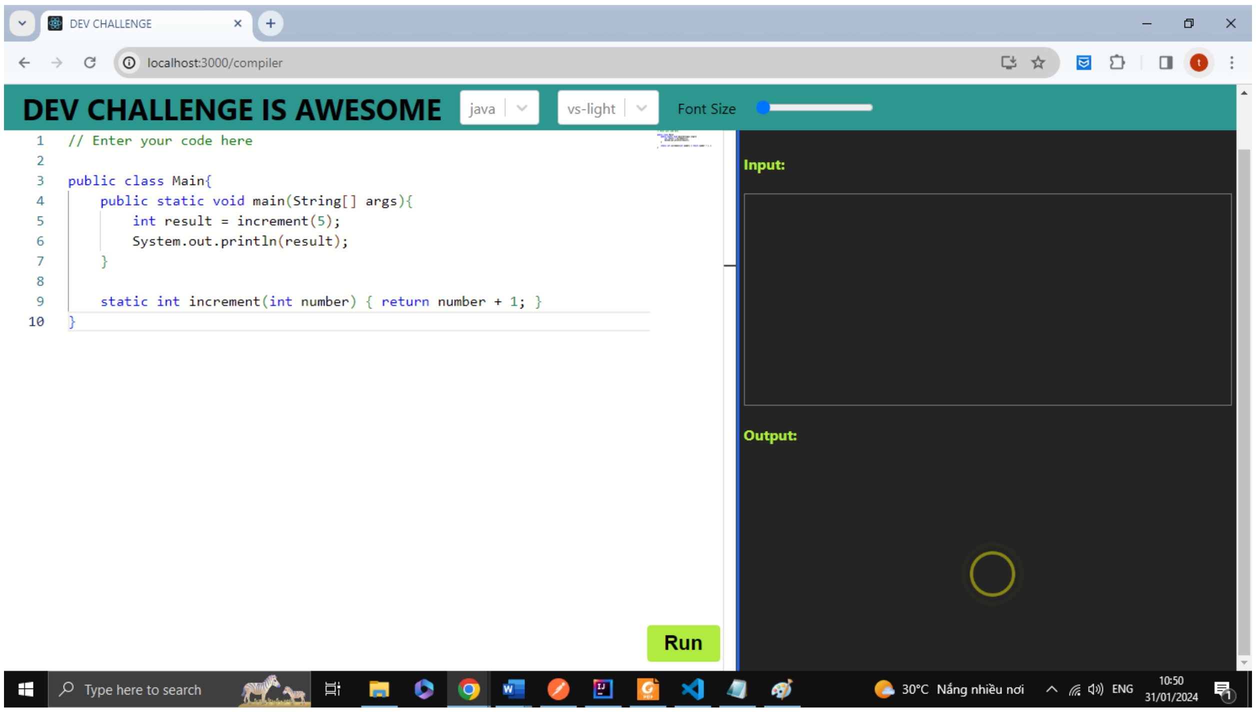
```
const TranslateEditor = () => {
    const [input, setInput] = useState('');
    const [output, setOutput] = useState('');
    const [languageList, setLanguageList] = useState([]);
    const [options, setOptions] = useState([]);
    const [availableTargetLanguageList, setAvailableTargetLanguageList] = useState([]);
    const [loading, setLoading] = useState(false);
    const [sourceLanguageCode, setSourceLanguageCode] = useState('auto');
    const [targetLanguageCode, setTargetLanguageCode] = useState('');
    const [detectedLanguage, setDetectedLanguage] = useState(null);
```

By making use of Axios library, we can simply make request to backend to execute the code and then get the result to be the translated text:

```
const translate = (text) => {
        if (text) {
            setLoading(true);
            Axios.post(`http://localhost:8080/api/v1/translate`, {
                q: input,
                source: sourceLanguageCode,
                target: targetLanguageCode,
                format: "text",
                api_key: ""
            }).then((res) => {
                let result = res.data.translatedText;
                setOutput(result);
                let detectedLanguage = res.data.detectedLanguage;
                if(detectedLanguage){
                    setDetectedLanguage(res.data.detectedLanguage);
                }
            }).then(() => {
                setLoading(false);
            })
        }
    }
```

4. Demo

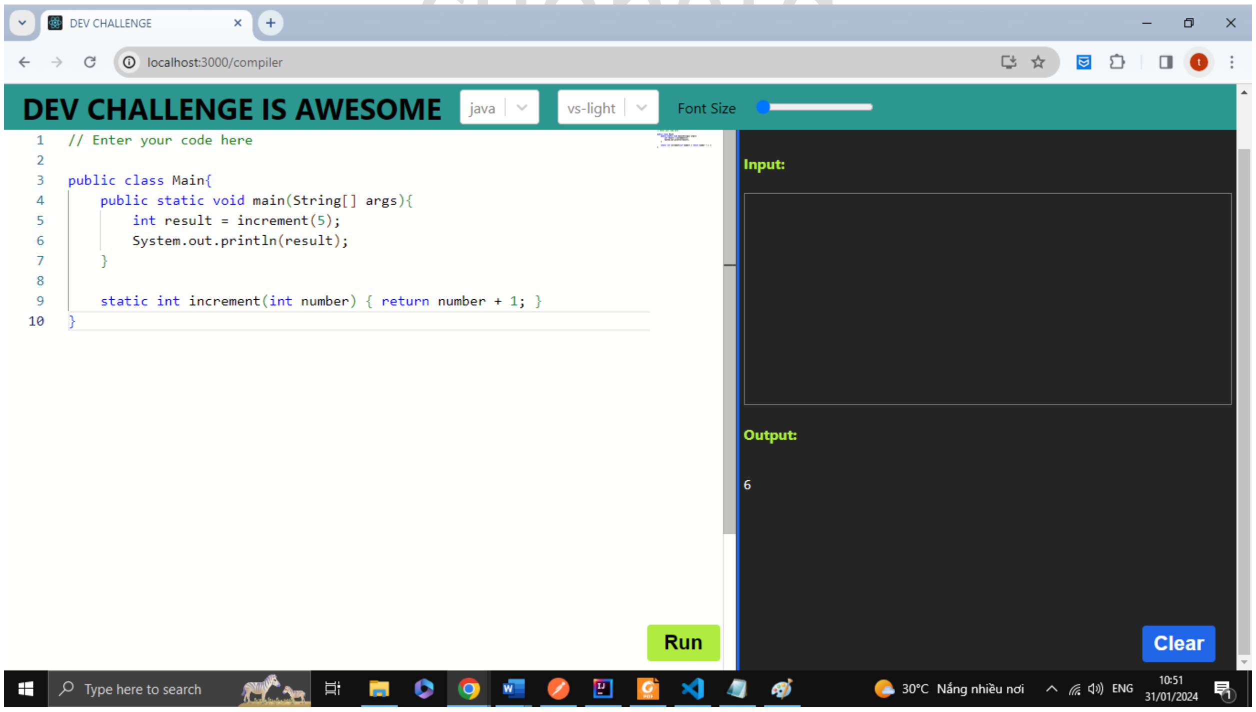
4.1. Code Interpreted/Compiled:



```java
// Enter your code here

public class Main{
    public static void main(String[] args){
        int result = increment(5);
        System.out.println(result);
    }

    static int increment(int number) { return number + 1; }
}
```

Hit the Run button, it appears the waiting screen in the right side and then the result will be given.

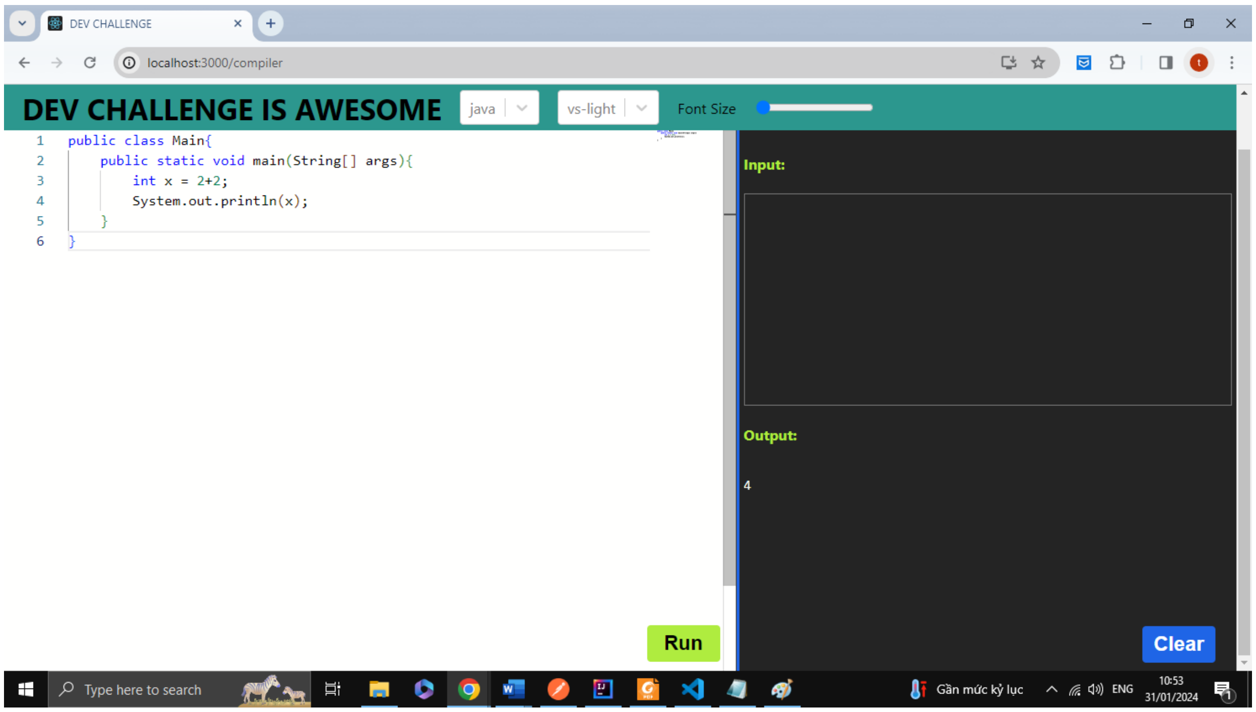
The result of this program is 6:



```java
// Enter your code here

public class Main{
    public static void main(String[] args){
        int result = increment(5);
        System.out.println(result);
    }

    static int increment(int number) { return number + 1; }
}
```

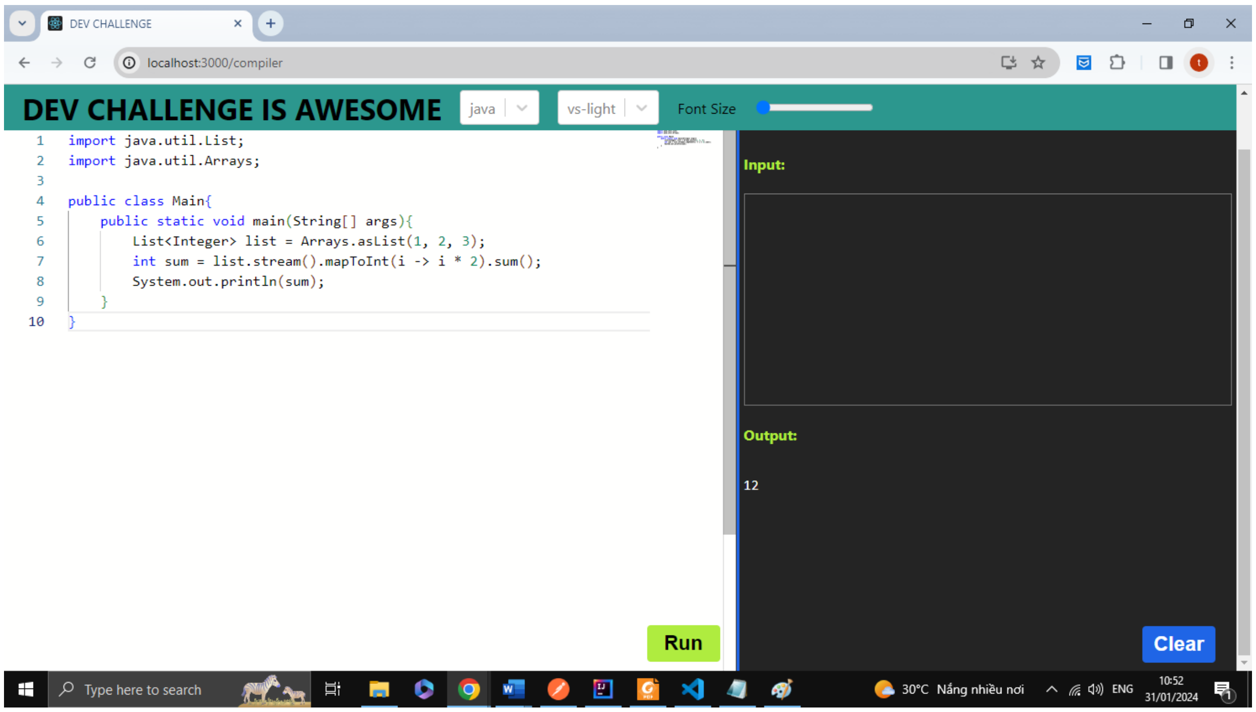
```
6
```

We can try more methods:



```java
public class Main{
    public static void main(String[] args){
        int x = 2+2;
        System.out.println(x);
    }
}
```

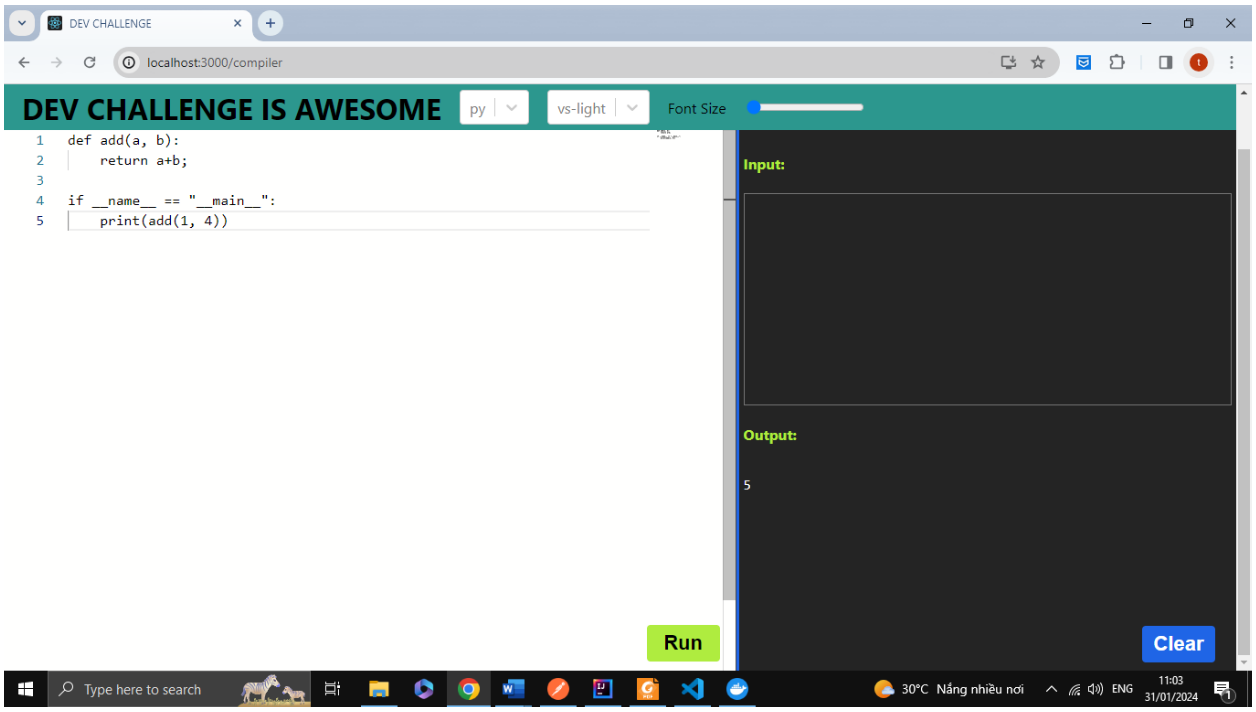
```
4
```



```java
import java.util.List;
import java.util.Arrays;

public class Main{
    public static void main(String[] args){
        List<Integer> list = Arrays.asList(1, 2, 3);
        int sum = list.stream().mapToInt(i -> i * 2).sum();
        System.out.println(sum);
    }
}
```

```
12
```

We can even try with another language. Let try with Python as a typical example:


DEV CHALLENGE IS AWESOME

py | vs-light | Font Size

```
def add(a, b):
    return a+b;

if __name__ == "__main__":
    print(add(1, 4))
```

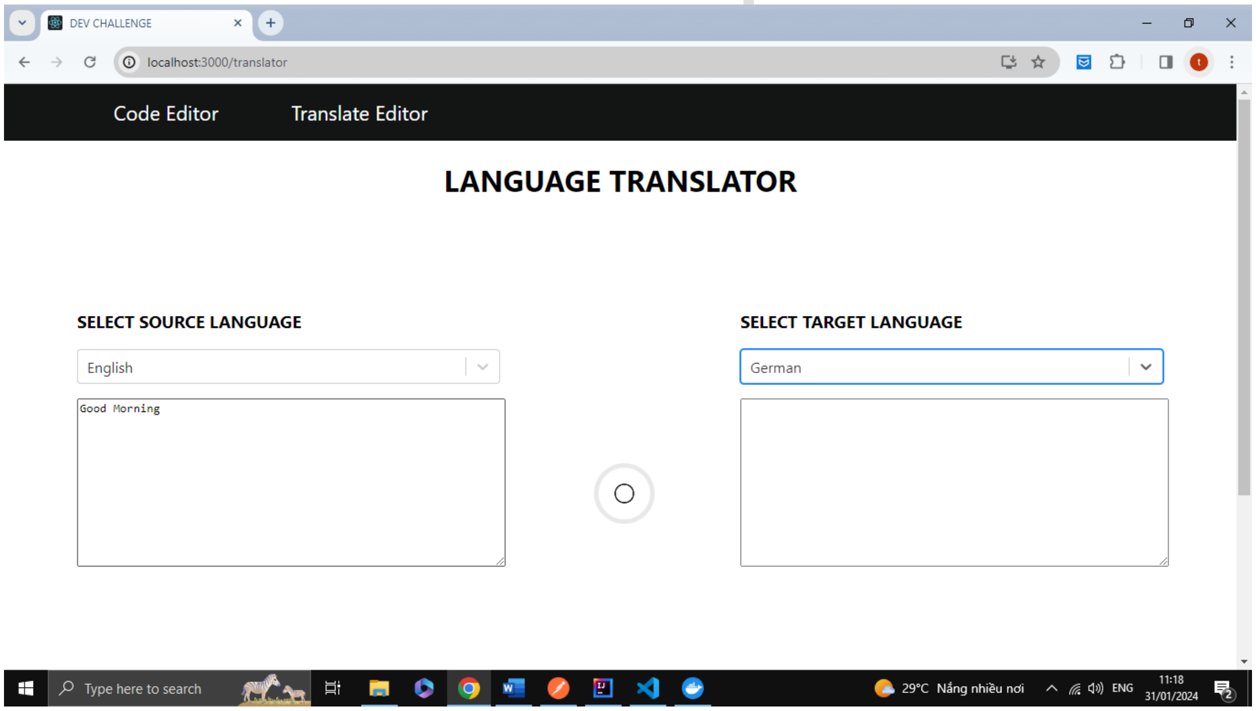
Input:

Output:

```
5
```

4.2. Language Translator

4.2.1. Translate with choosen source language



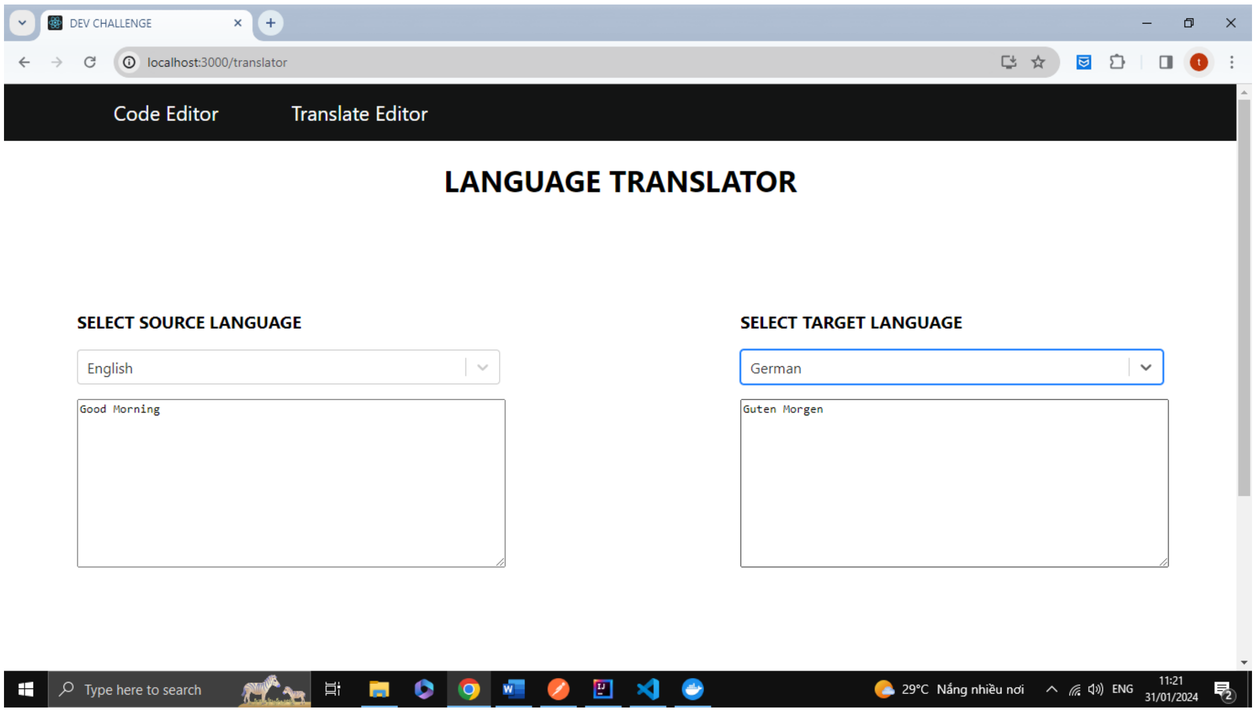
LANGUAGE TRANSLATOR

SELECT SOURCE LANGUAGE: English

```
Good Morning
```

SELECT TARGET LANGUAGE: German

When we type a text and choose the source and target languages, the waiting screen will appear and then we can get the translated text:



LANGUAGE TRANSLATOR

SELECT SOURCE LANGUAGE

English

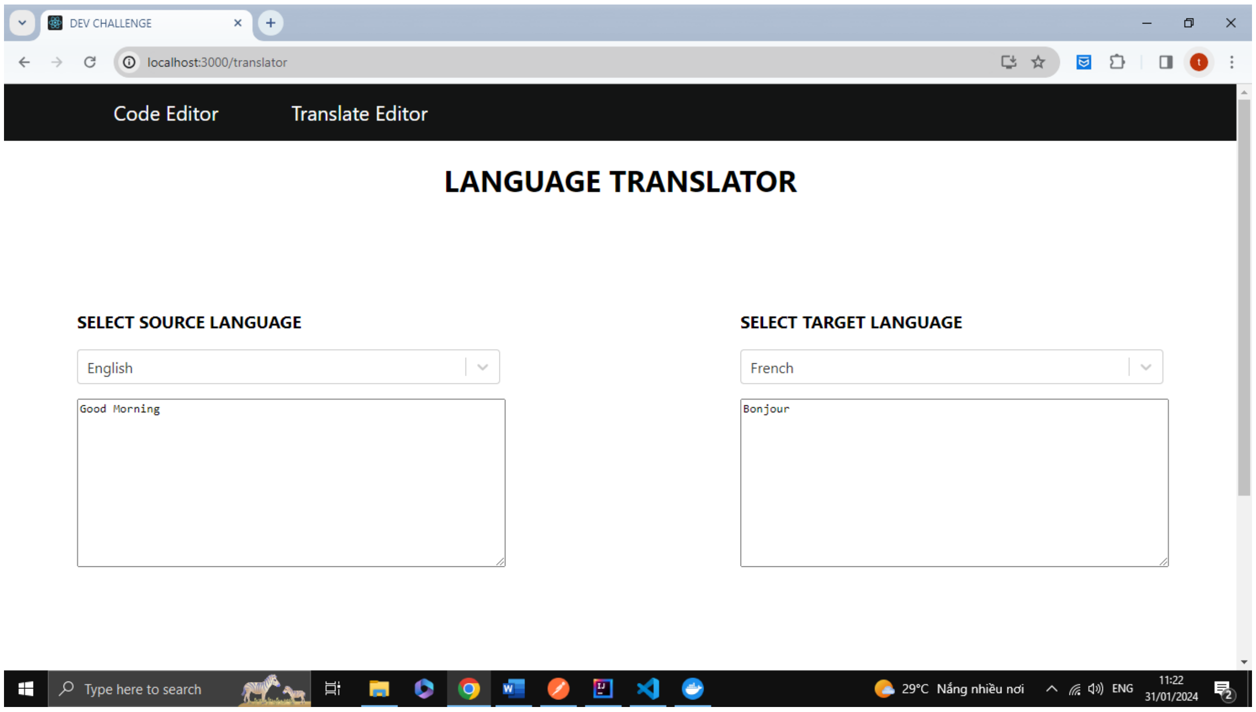
Good Morning

SELECT TARGET LANGUAGE

German

Guten Morgen

We can also try to translate from English to another language like French:



LANGUAGE TRANSLATOR

SELECT SOURCE LANGUAGE

English

Good Morning

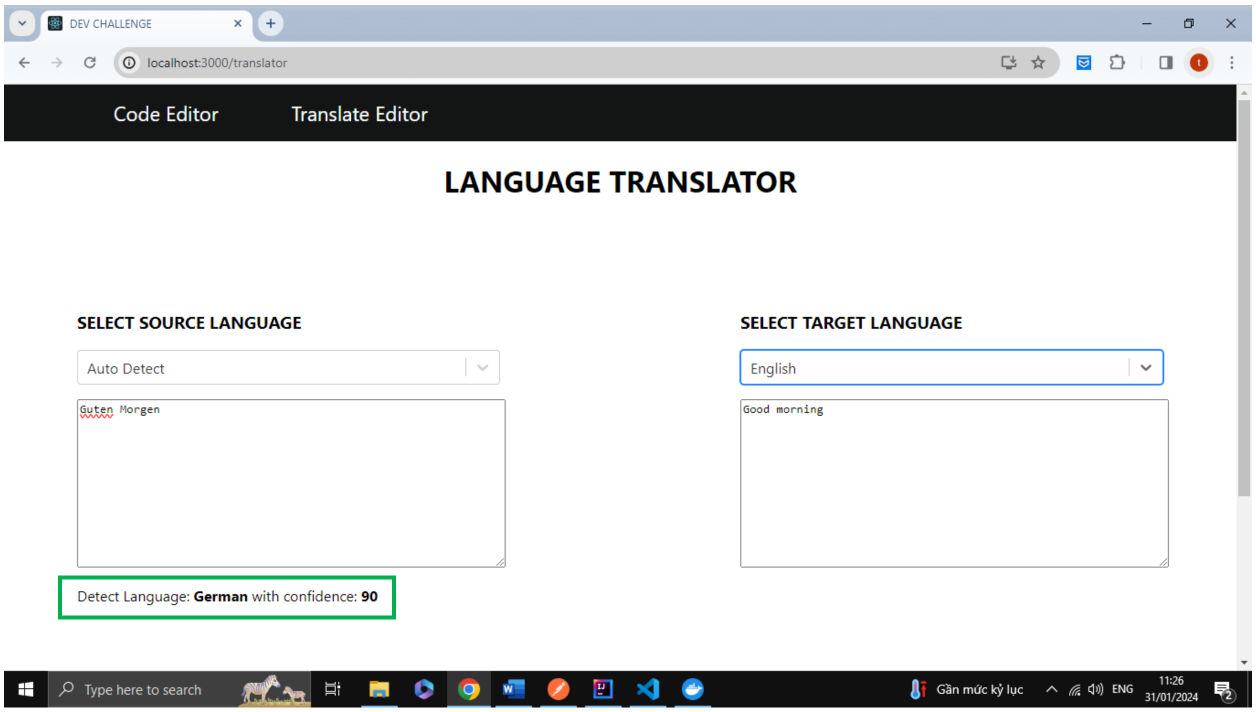
SELECT TARGET LANGUAGE

French

Bonjour

4.2.2. Translate with auto-detect source language

Rather than choose specific language, we can choose the “Auto Detect” option in the Source language select:



```
Guten Morgen
```

```
Good morning
```

Detect Language: **German** with confidence: **90**

In this case, we choose auto detect a source language and translate it to English. After that we can also see the detected language information and its confidence of detection.

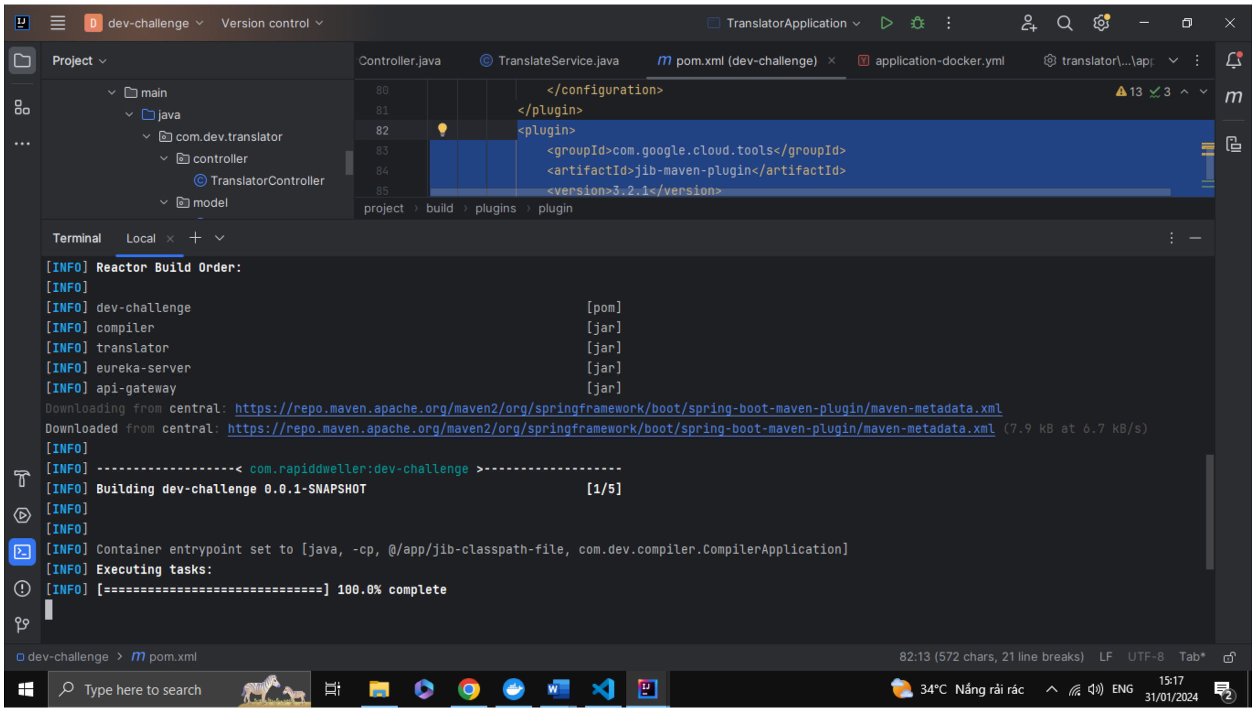
5. Dockerized the application

For Spring Boot project, we can use jib plugin from Google to build our service images: https://cloud.google.com/java/getting-started/jib

Add this plugin into the global pom.xml:

```
<plugin>
             <groupId>com.google.cloud.tools</groupId>
             <artifactId>jib-maven-plugin</artifactId>
             <version>3.2.1</version>
             <configuration>
                <from>
                   <image>eclipse-temurin</image>
                </from>
                <to>
                   <image>dev-challenge/${project.artifactId}</image>
                </to>
             </configuration>
             <executions>
                <execution>
                   <phase>package</phase>
                   <goals>
                      <goal>dockerBuild</goal>
                   </goals>
                </execution>
             </executions>
          </plugin>
```

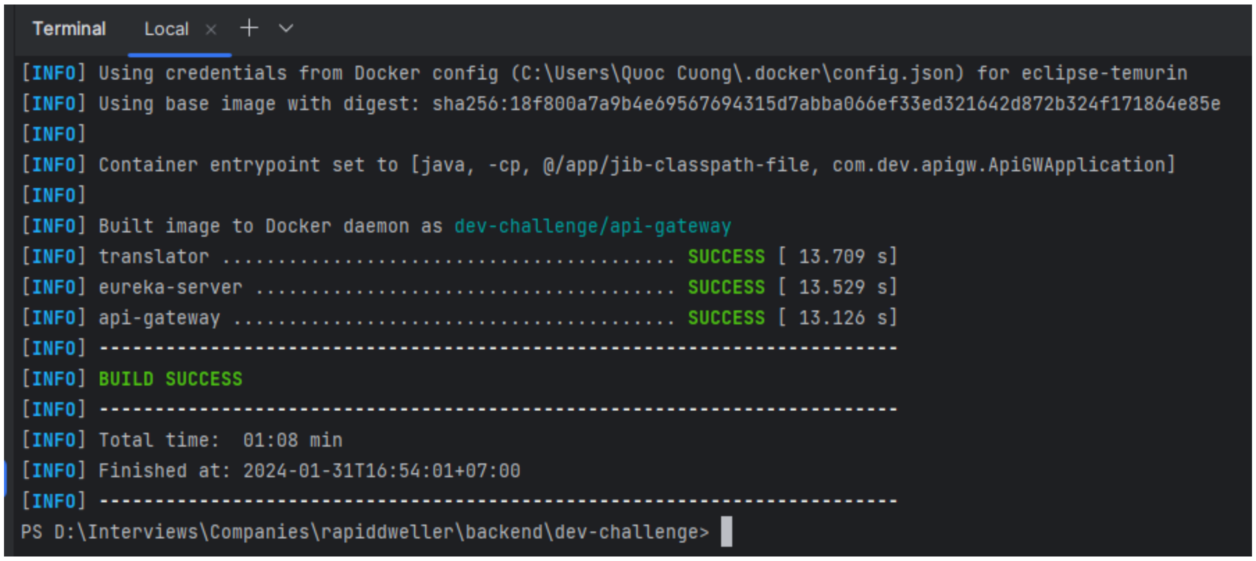
To build the image locally, use this command: **mvn clean compile jib:dockerBuild**



```
</configuration>
</plugin>
<plugin>
    <groupId>com.google.cloud.tools</groupId>
    <artifactId>jib-maven-plugin</artifactId>
    <version>3.2.1</version>
```

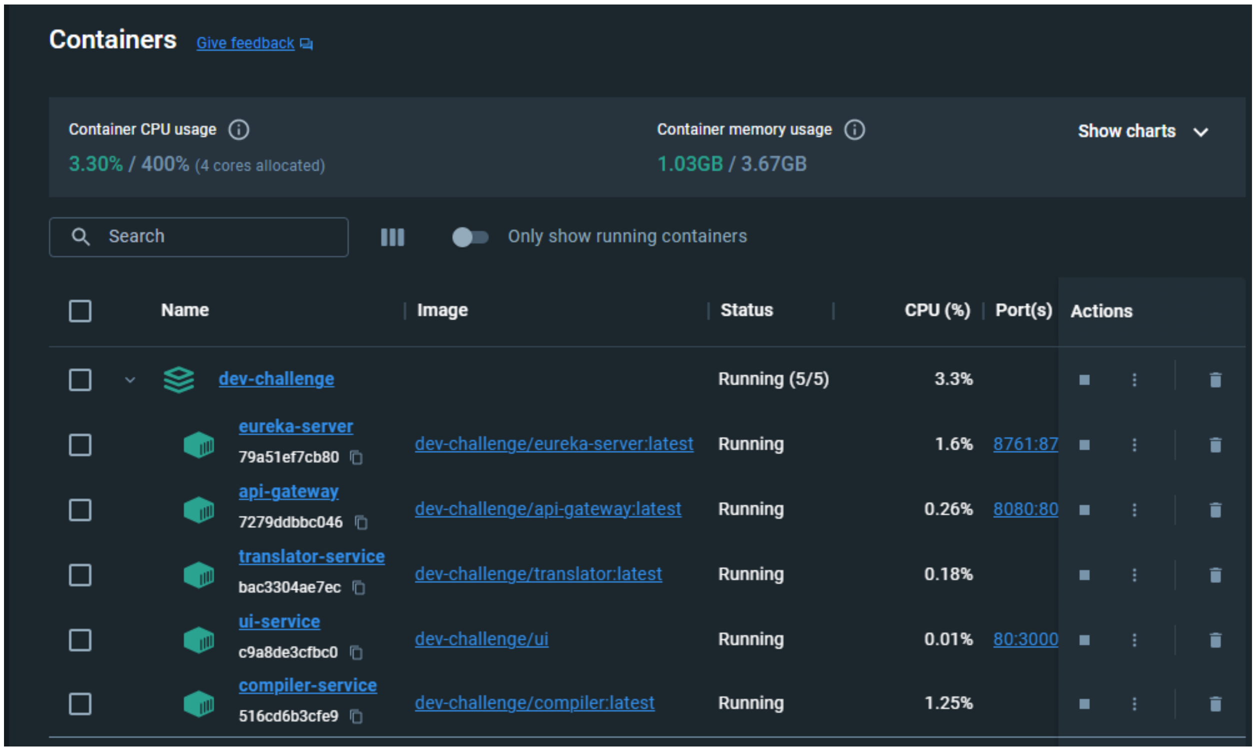
```
[INFO] Reactor Build Order:
[INFO]
[INFO] dev-challenge                                                      [pom]
[INFO] compiler                                                           [jar]
[INFO] translator                                                         [jar]
[INFO] eureka-server                                                      [jar]
[INFO] api-gateway                                                        [jar]
Downloading from central: https://repo.maven.apache.org/maven2/org/springframework/boot/spring-boot-maven-plugin/maven-metadata.xml
Downloaded from central: https://repo.maven.apache.org/maven2/org/springframework/boot/spring-boot-maven-plugin/maven-metadata.xml (7.9 kB at 6.7 kB/s)
[INFO]
[INFO] -------------------< com.rapiddweller:dev-challenge >-------------------
[INFO] Building dev-challenge 0.0.1-SNAPSHOT                              [1/5]
[INFO]
[INFO]
[INFO] Container entrypoint set to [java, -cp, @/app/jib-classpath-file, com.dev.compiler.CompilerApplication]
[INFO] Executing tasks:
[INFO] [==============================] 100.0% complete
```

Wait till the build success:



```
[INFO] Using credentials from Docker config (C:\Users\Quoc Cuong\.docker\config.json) for eclipse-temurin
[INFO] Using base image with digest: sha256:18f800a7a9b4e69567694315d7abba066ef33ed321642d872b324f171864e85e
[INFO]
[INFO] Container entrypoint set to [java, -cp, @/app/jib-classpath-file, com.dev.apigw.ApiGWApplication]
[INFO]
[INFO] Built image to Docker daemon as dev-challenge/api-gateway
[INFO] translator ......................................... SUCCESS [ 13.709 s]
[INFO] eureka-server ...................................... SUCCESS [ 13.529 s]
[INFO] api-gateway ........................................ SUCCESS [ 13.126 s]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  01:08 min
[INFO] Finished at: 2024-01-31T16:54:01+07:00
[INFO] ------------------------------------------------------------------------
PS D:\Interviews\Companies\rapiddweller\backend\dev-challenge>
```

Then, we can run docker compose file:



Containers Give feedback

Container CPU usage 3.30% / 400% (4 cores allocated)

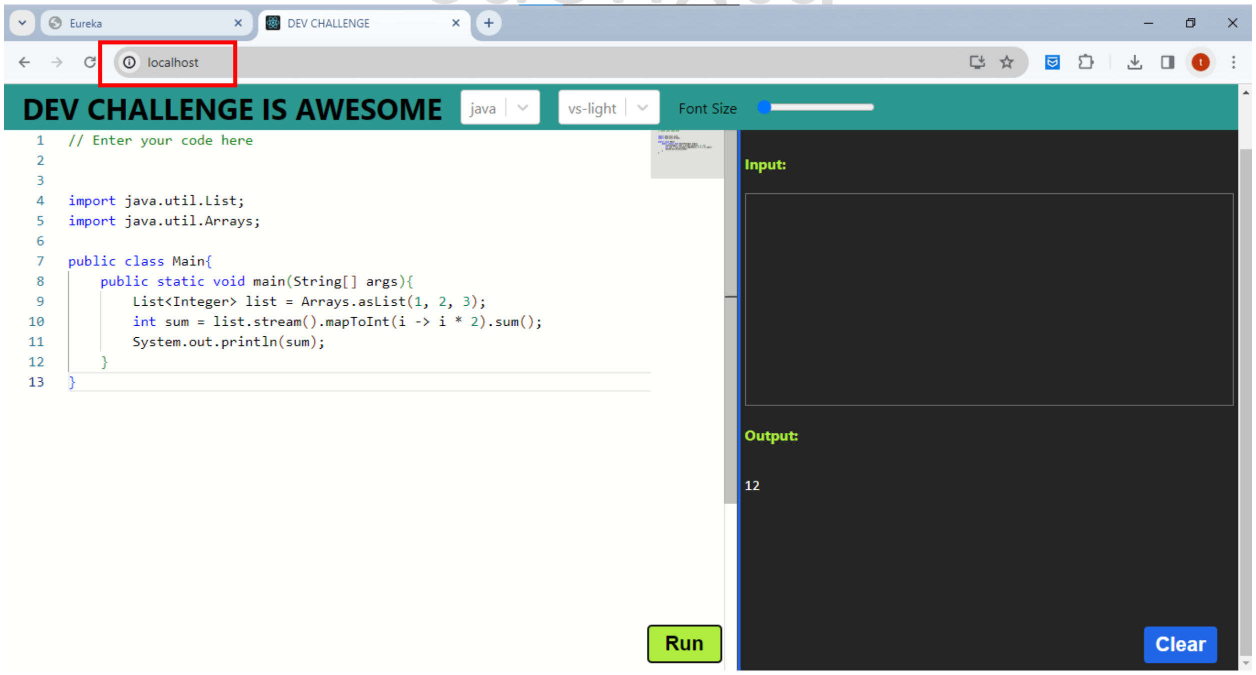
Container memory usage 1.03GB / 3.67GB

Show charts

Only show running containers

Name	Image	Status	CPU (%)	Port(s)
dev-challenge		Running (5/5)	3.3%	
eureka-server 79a51ef7cb80	dev-challenge/eureka-server:latest	Running	1.6%	8761:87
api-gateway 7279ddbbc046	dev-challenge/api-gateway:latest	Running	0.26%	8080:80
translator-service bac3304ae7ec	dev-challenge/translator:latest	Running	0.18%	
ui-service c9a8de3cfbc0	dev-challenge/ui	Running	0.01%	80:3000
compiler-service 516cd6b3cfe9	dev-challenge/compiler:latest	Running	1.25%	

Now we can send the requests as usual but this time we can use localhost without explicitly specify port number:



localhost

DEV CHALLENGE IS AWESOME

java | vs-light | Font Size

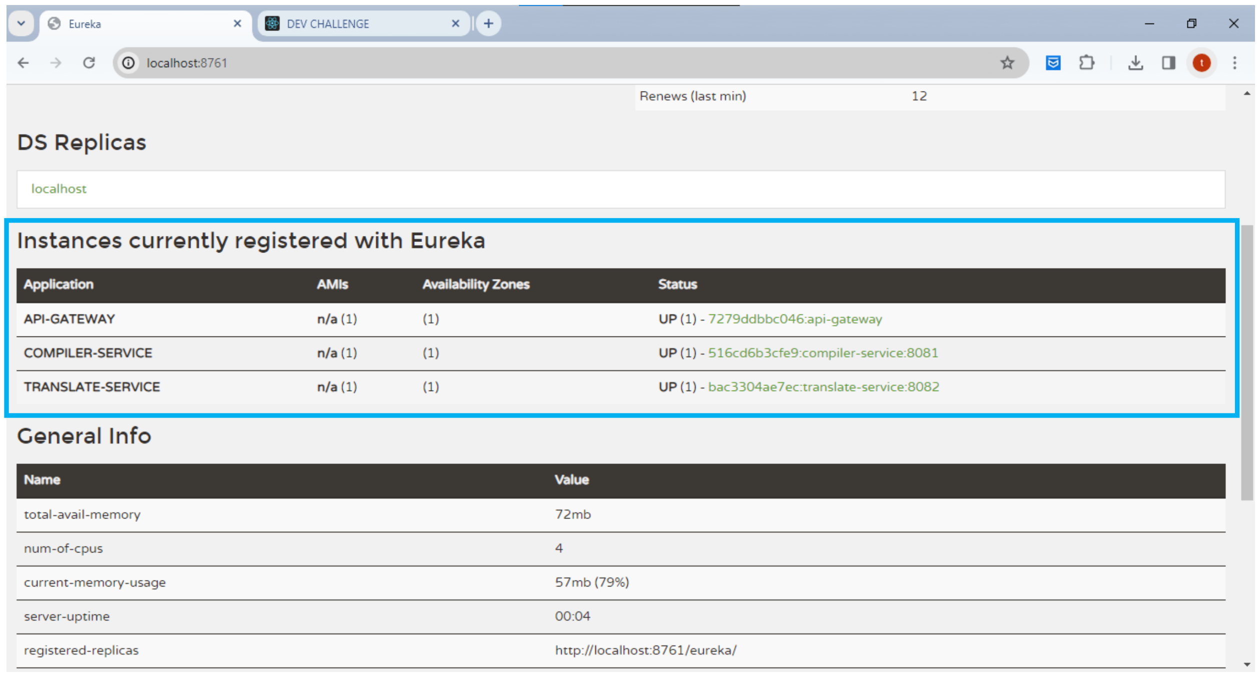
```
// Enter your code here


import java.util.List;
import java.util.Arrays;

public class Main{
    public static void main(String[] args){
        List<Integer> list = Arrays.asList(1, 2, 3);
        int sum = list.stream().mapToInt(i -> i * 2).sum();
        System.out.println(sum);
    }
}
```

Input:

Output:

```
12
```

And kindly check our services registered in discovery server:



Renews (last min)	12

DS Replicas

localhost

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
API-GATEWAY	n/a (1)	(1)	UP (1) - 7279ddbbc046:api-gateway
COMPILER-SERVICE	n/a (1)	(1)	UP (1) - 516cd6b3cfe9:compiler-service:8081
TRANSLATE-SERVICE	n/a (1)	(1)	UP (1) - bac3304ae7ec:translate-service:8082

General Info

Name	Value
total-avail-memory	72mb
num-of-cpus	4
current-memory-usage	57mb (79%)
server-uptime	00:04
registered-replicas	http://localhost:8761/eureka/

Everything works perfectly!

We can know make use of this docker compose file for deployment somewhere else.

Thanks for reading my solution. If you have any comments or feedbacks as well as ideas to enhance my solution, do not hesitate to let me know.

Have a nice day!