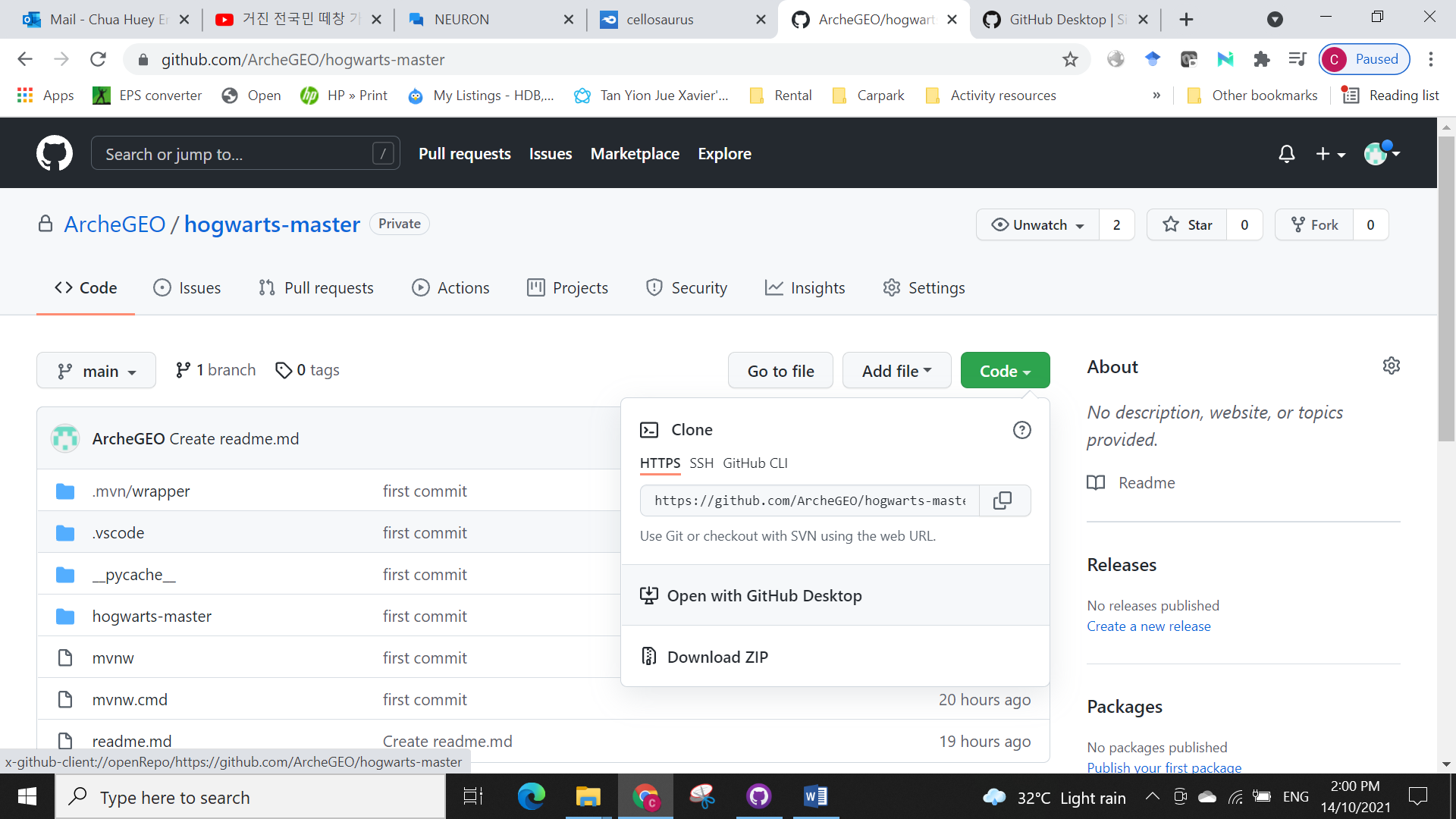
**ArcheGEO online pipeline (Hogwarts-master)**

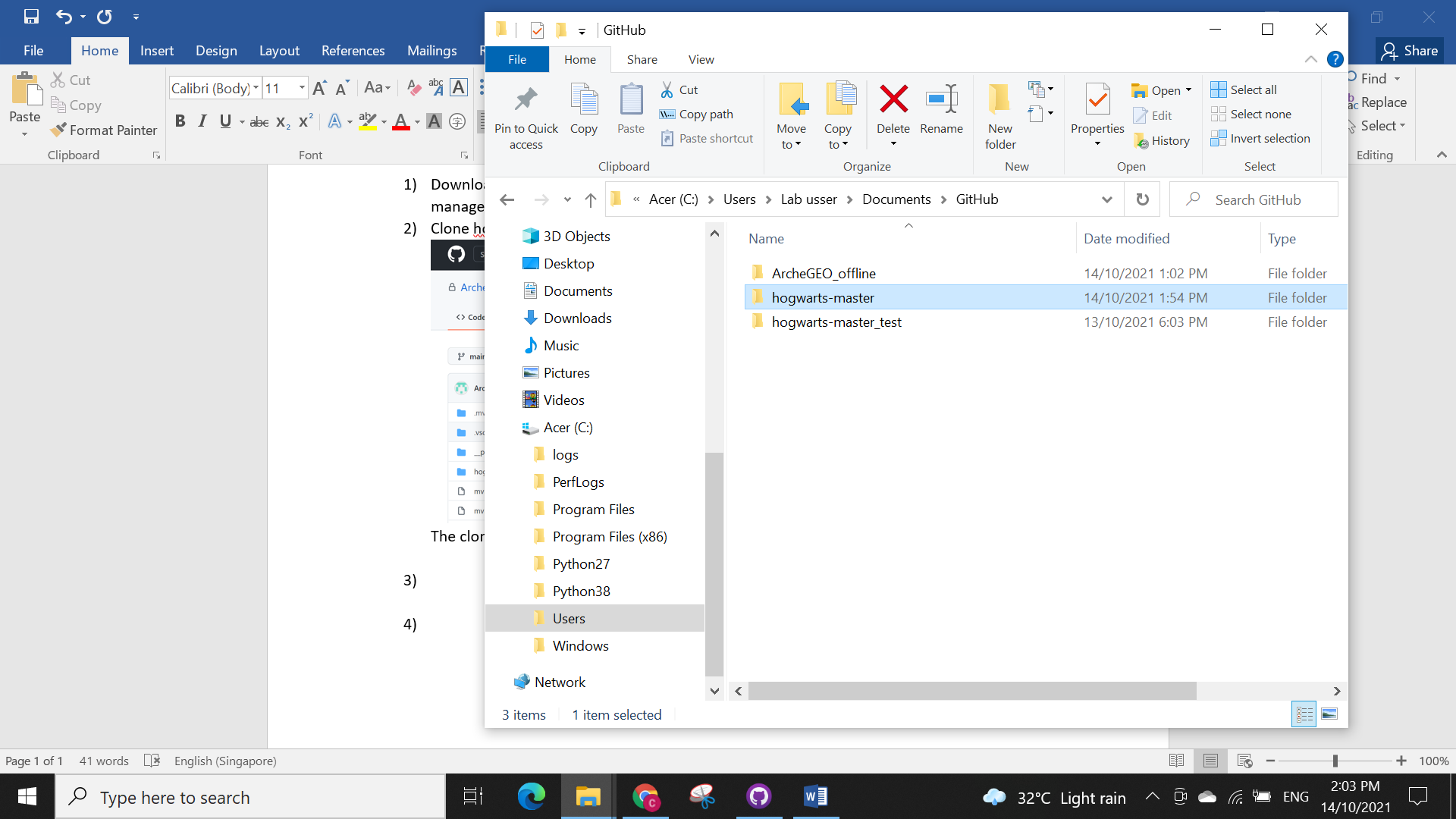
The online pipeline handles the web service of ArcheGEO. It utilizes the PostgreSQL database set up by ArcheGEO\_offline pipeline. The online pipeline is a springboot-vue project and is developed from Jonas Hecht’s example Github project (<https://github.com/jonashackt/spring-boot-vuejs>). Springboot (java) is the backend and Vue is the frontend.

**Cloning the project from Github**

1. Download and install Github desktop (<https://desktop.github.com/>). It allows easy management of the cloned project.
2. Clone hogwarts-master from Github.



The cloned directory can be found in the Github folder.



**Building the project using Visual Studio Code**

1. Open the project using Visual Studio Code (<https://code.visualstudio.com/>).

Check the details of the PostgreSQL database setup in application.properties file found in ..hogwarts-master/backend/src/main/resources/ and make sure they correspond to the database setup by ArcheGEO offline pipeline.

spring.datasource.platform=postgres

spring.datasource.url=jdbc:postgresql://localhost:5432/hogwarttest

spring.datasource.username=postgres

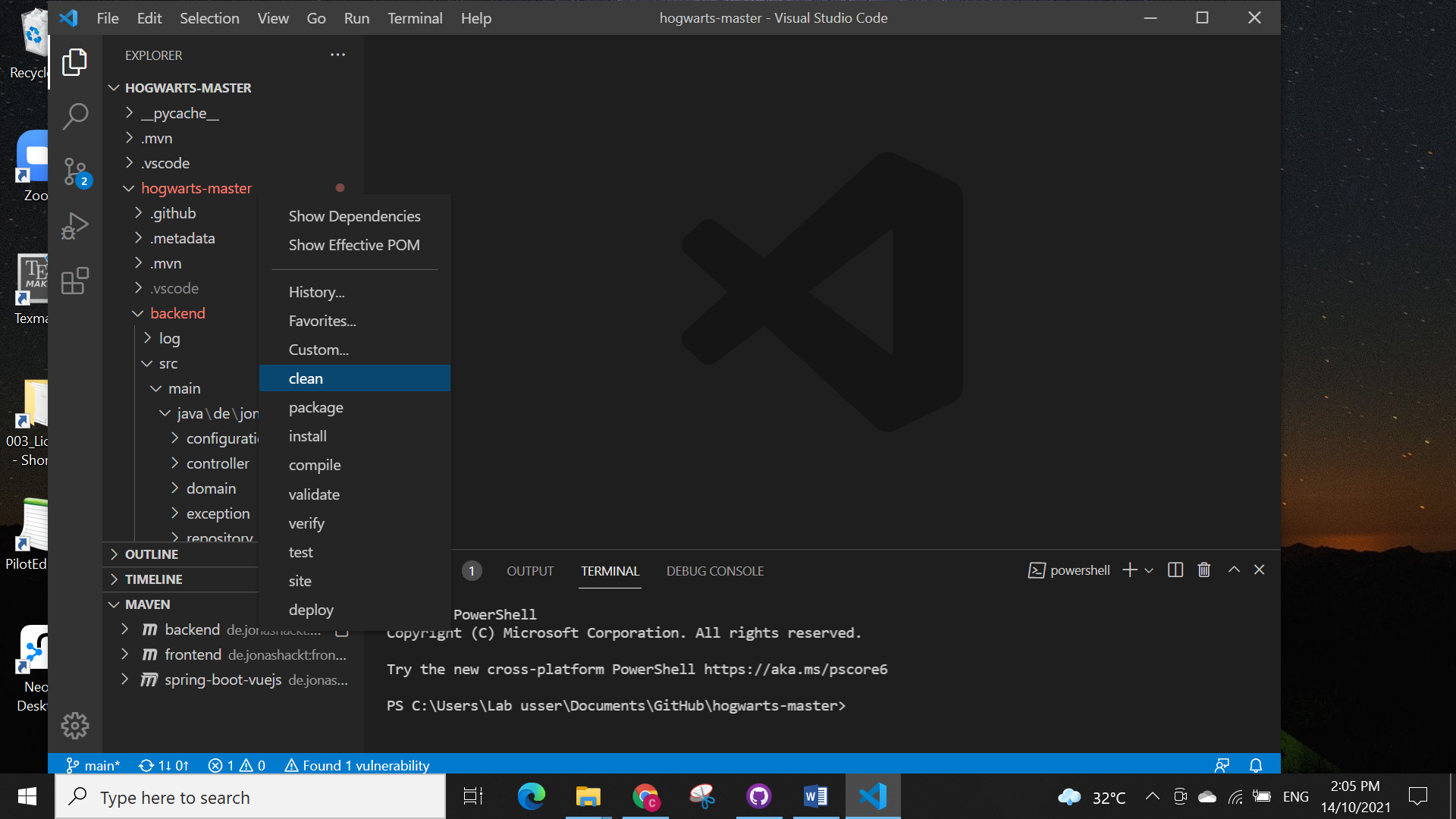
spring.datasource.password=trove

Note that you can also modify the folder for ArcheGEO web service log by modifying

logging.path=../hogwarts-master/backend/log

logging.file.name=${logging.path}/archegeo.log

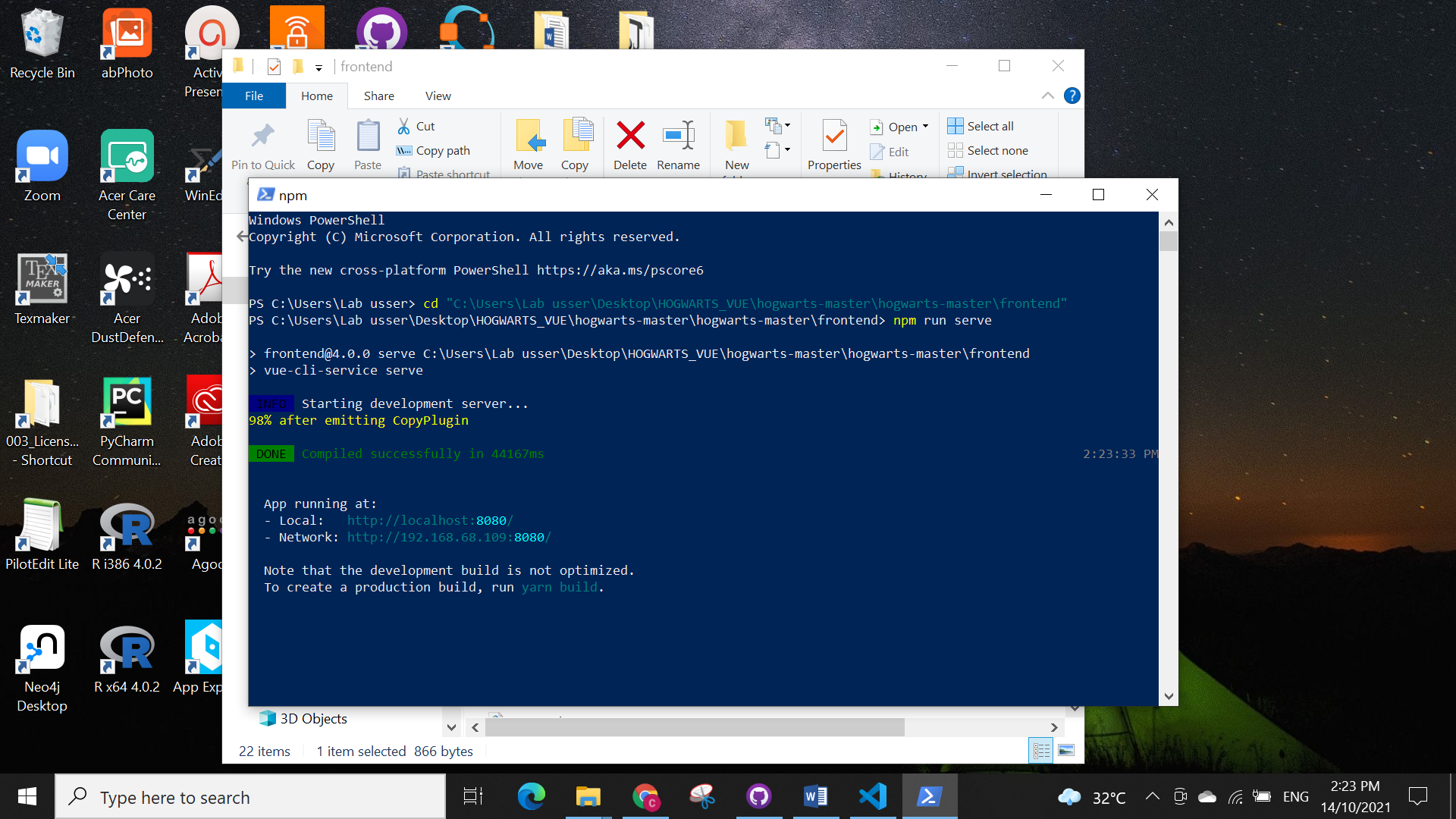
1. Build the frontend and backend code using maven pom.xml. Right click on backend in MAVEN and select clean if this is not your first time building this project. Then, click on install to install the backend project from pom.xml. Do the same for the frontend.



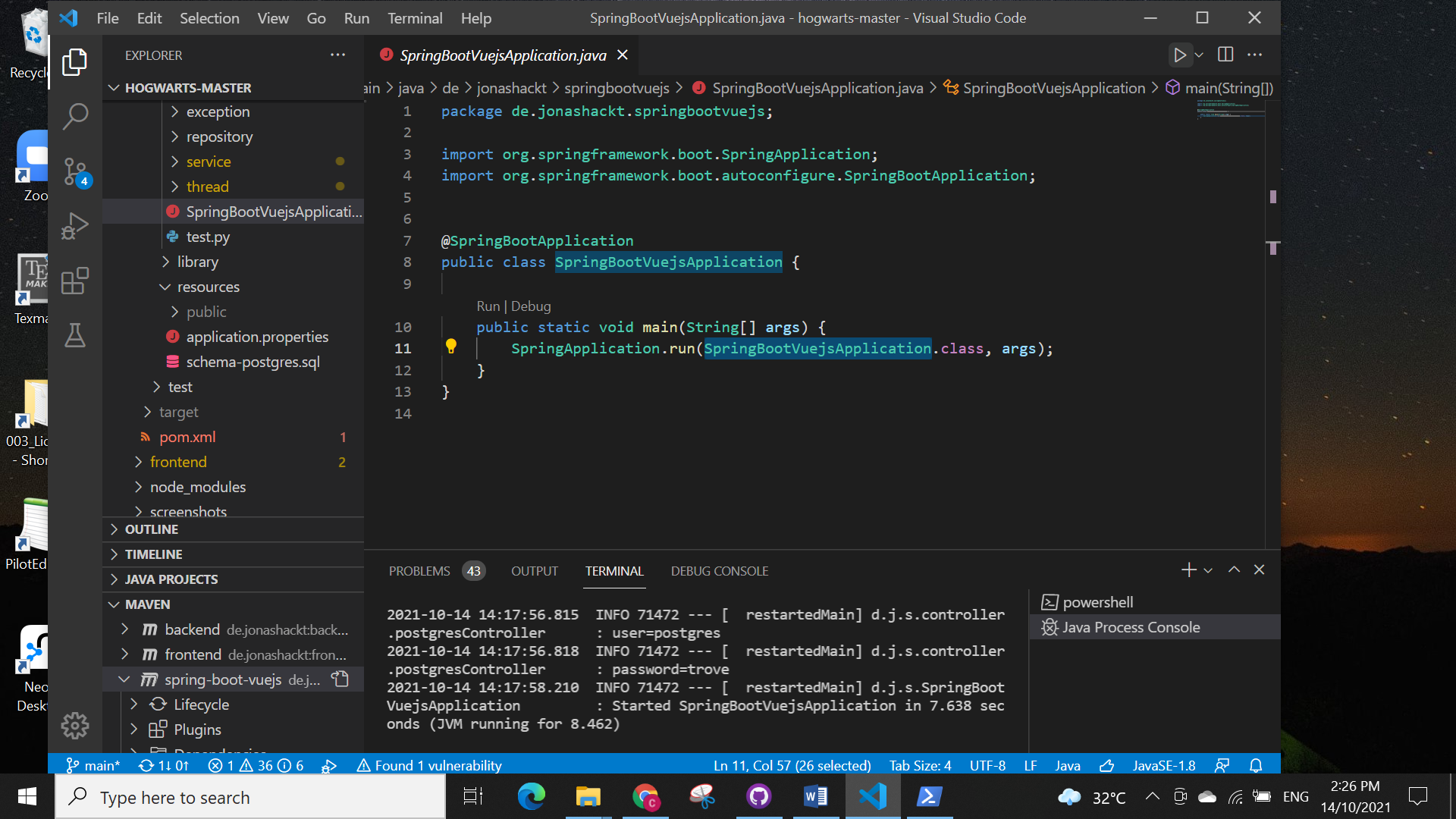
**Running the project**

1. Once the project builds without errors, you can run them as follows:

* Frontend using Windows PowerShell by going to the frontend directory and using command ‘npm run serve’:

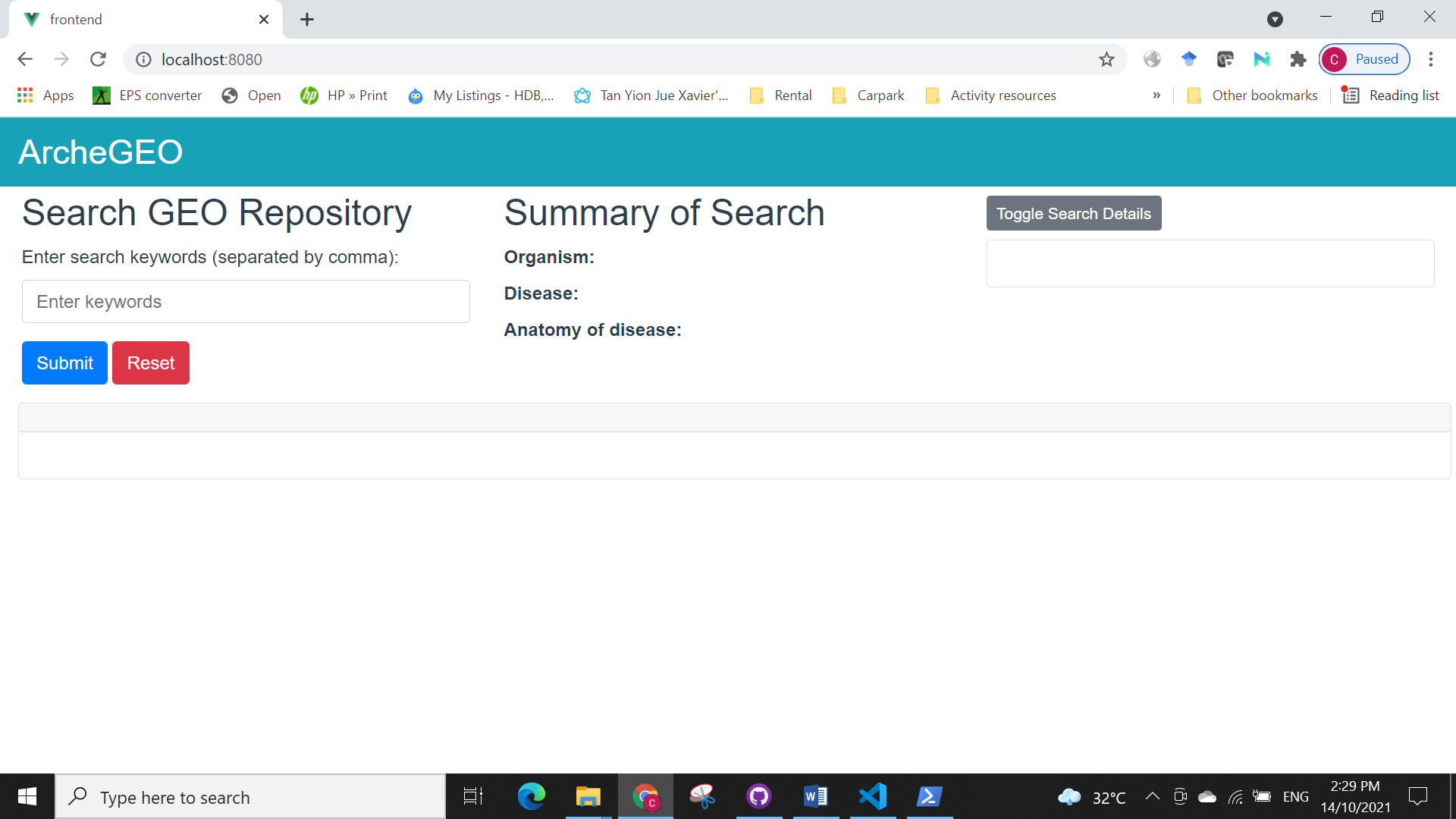


* Backend using Visual Studio Code by doing a “Run” (circled in red) on the SpringBootVuejsApplication.java:

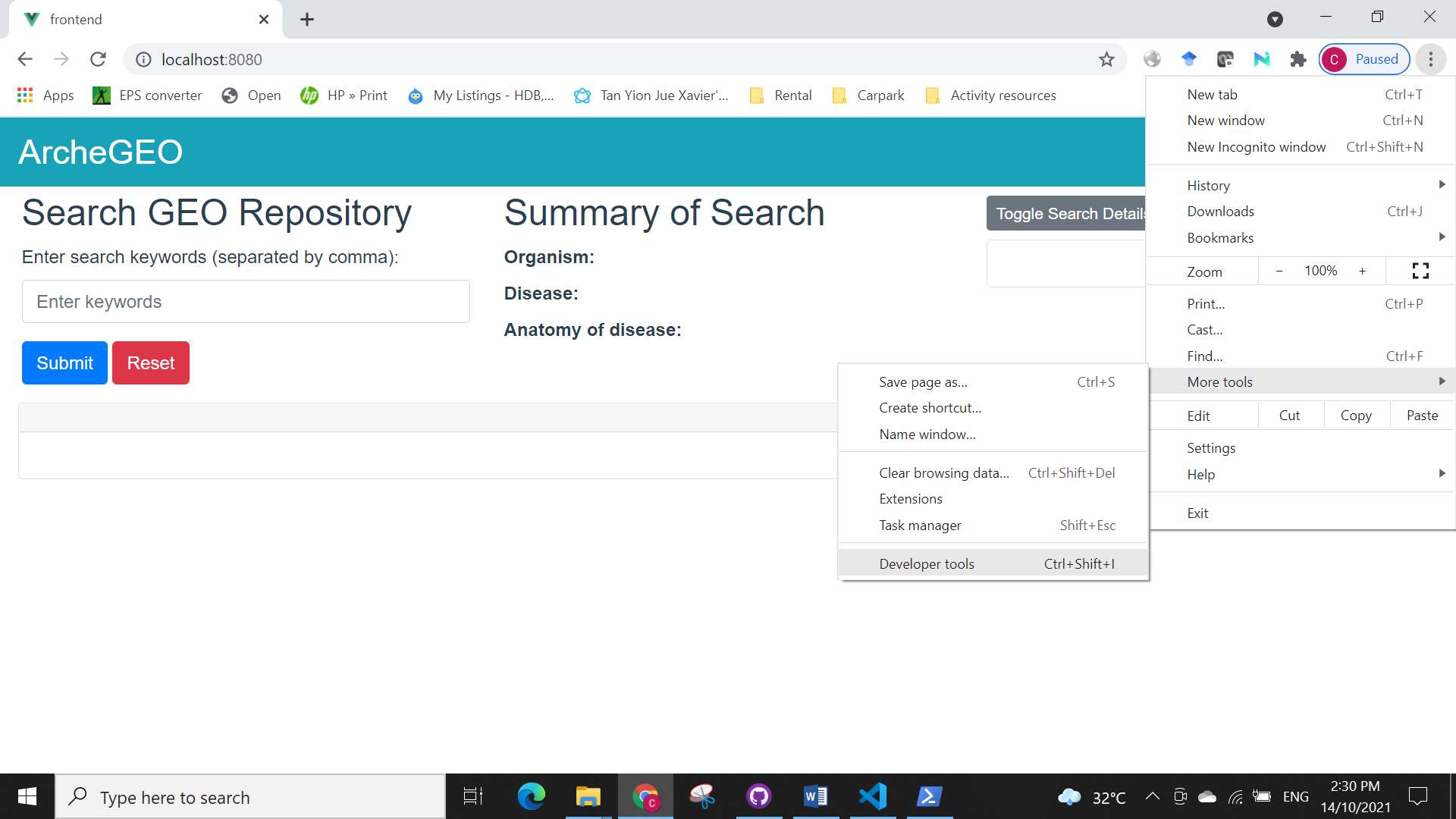


When the backend runs successfully, you should see “JVM running for …” in the terminal (green box).

1. Launch the service in your web browser (Chrome is preferred) by going to localhost:8080.



1. Debugging of the web browser can be done by using the Developers Tool.



**Frontend-Backend Communication**

The frontend is coded using Vue (BootstrapVue). The web browser main page is found in MainPage.vue. MainPage.vue uses some styling of the page that is specified in MainPage.css (found in ../assets/css/ folder). Vue communicates with the backend using AXIOS (e.g., get, post message) that is found in backend-api.js. Objects and datatypes are passed through AXIOS by encoding them as JSON object. Multiple parameters can be passed via a single AXIOS call by creating a new JSON object. Example:

let paramInput = {

          ruleListArray: jsonRulelist,

          validItemsArray: this.validItems,

          validTabArray: jsonValidResultTab

        }

        api.getNewTabRecords(JSON.stringify(paramInput)).then(response => {

          console.log("done transferRulelist...")

          this.validItems.push(response.data)

          this.loading=false

        })

backend-api.js receives the call from MainPage.vue and processes it to pass to BackendController.java

getNewTabRecords(params) {

        console.log('getNewTabRecords');

        return AXIOS.post(`/getNewTabRecords`, params, {

            headers: {

              // Overwrite Axios's automatically set Content-Type

              'Content-Type': 'application/json'

            }

        });

    },

In the frontend, console.log are used to output debug messages to aid debugging.

In the backend, BackendController.java receives the AXIOS messages and processes them by leveraging annotations such as @ResponseBody, @RequestMapping. The JSON object have to be decoded in BackendController.java and this is done with the GSON library. Note that after processing in BackendController.java, some API calls may need to return variables back to the frontend. To do this, we can do a return of a JSON object again.

@ResponseBody

    @RequestMapping(path="/getNewTabRecords", method=RequestMethod.POST)

    public String getNewTabRecords(@RequestBody String jsonParams) {

        LOG.info("backendcountroller.java - getNewTabRecords ");

        ArrayList<gdsrecordmini> newTabRecords = new ArrayList<gdsrecordmini>();

        tabRecordParam params = gson.fromJson(jsonParams, new TypeToken<tabRecordParam>(){}.getType());

        ArrayList<rulelist> rulelistArray = params.getRuleListArray();

        ArrayList<ArrayList<gdsrecordmini>> validitemsArray = params.getValidItemsArray();

        ArrayList<String> validtabArray = params.getValidTabArray(); //tell us what the original valid result tabs are. Only those are selectable for combination results view

        ...<these contents have been removed>...

        pgController.GDS\_removeAllNewTabTempTables(tempTableList);

        pgController.GDS\_removeAllNewTabTempTables(negateTempTableList);

        return gson.toJson(newTabRecords);

    }

For proper decoding of the JSON objects, a class is defined in the domain package of the backend that corresponds exactly to that in the frontend. Note that the name of the variables in this class should correspond exactly to that encoded in the frontend JSON object.

package de.jonashackt.springbootvuejs.domain;

import java.util.ArrayList;

public class tabRecordParam {

    ArrayList<rulelist> ruleListArray = new ArrayList<rulelist>();

    ArrayList<ArrayList<gdsrecordmini>> validItemsArray = new ArrayList<ArrayList<gdsrecordmini>>();

    ArrayList<String> validTabArray = new ArrayList<String>();

    public tabRecordParam() {

    }

    public ArrayList<rulelist> getRuleListArray(){

        return ruleListArray;

    }

    public ArrayList<ArrayList<gdsrecordmini>> getValidItemsArray(){

        return validItemsArray;

    }

    public ArrayList<String> getValidTabArray(){

        return validTabArray;

    }

In some cases, these returned JSON objects contained variables that should be mapped back to corresponding objects in the frontend. We will need to perform mapping of the “response.data” to these objects.

api.setInvalidRecordsAsValid(JSON.stringify(jsonObject)).then(response => {

        console.log("done confirmSetAsValidRows")

        let resData = response.data;

        console.log("response.data length = "+response.data.length+" resData.length="+resData.length)

        let itemArray = [];

        if(resData.length>0){

          resData.map((data, index)=> {

            console.log("data length = "+data.length+" index="+index)

            console.log("data = "+data)

            if(data.length>0){

              //update the invalid records entries to corresponding valid tabs

              itemArray.push(data);

              //update message details regarding number of valid records

              console.log("gdssearch message="+this.gdssearch.message)

LOGGER messages are used for debugging purpose in the backend code. An example is:

LOG.info("processKeyword keyword ="+keyword);

The message will be shown in the terminal panel of Visual Studio Code.

**Backend-PostgreSQL Communication**

For other example, you can refer to <https://zetcode.com/springboot/postgresql/>.

We use several annotations:

* @Entity: the class is an entity and is mapped to a database table
* @Table: name of the database table used for mapping
* @Id: specifies the primary key of an entity
* @GeneratedValue: specify the generation strategies for generating unique primary keys.

In some cases, we might have to deal with composite primary key. To do so, create a serialization class in the same package that you store your entity class (e.g., package domain). Each column that forms the composite primary key (e.g., gdsid, inputorganism and inputdisease) and specified in that serializable class (i.e.g, gdsrecordviewId)

package de.jonashackt.springbootvuejs.domain;

import java.io.Serializable;

public class gdsrecordviewId implements Serializable{

    private int gdsid;

    private String inputorganism;

    private String inputdisease;

    // default constructor

    public gdsrecordviewId(){

    }

    public gdsrecordviewId(int gdsid, String inputorganism, String inputdisease) {

        this.gdsid = gdsid;

        this.inputorganism = inputorganism;

        this.inputdisease = inputdisease;

    }

    // equals() and hashCode()

}

Use this serializable class in the entity class (i.e., @IdClass(gdsrecordviewId.class)). Note that all columns should also be annotated with @Id in addition to @Column.

package de.jonashackt.springbootvuejs.domain;

import java.util.Objects;

import javax.persistence.\*;

@Entity

@IdClass(gdsrecordviewId.class)

@Table(name = "gdsrecordview")

public class gdsrecordview {

    @Id

    @Column(name="gdsid")

    private int gdsid;

    ...<contents have been removed>...

    @Column(name="platform")

    private String platform;

    @Column(name="samplenum")

    private int samplenum;

    @Column(name="fname")

    private String fname;

    @Id

    @Column(name="inputorganism")

    private String inputorganism;

    @Id

    @Column(name="inputdisease")

    private String inputdisease;

    @Column(name="validrecord")

    private boolean validrecord;

    @Column(name="organismmismatch")

    private boolean organismmismatch;

    ...<contents have been removed>...

    public gdsrecordview() {

    }

    public void setValidRecord(boolean validrecord) {

        this.validrecord = validrecord;

    }

    public boolean getOrganismMismatch() {

        return organismmismatch;

    }

    ...<contents have been removed>...

}

Sometimes, we may need to generate a running index as primary key for a table. To do this, use the @GeneratedValue annotation. You will also need @SequenceGenerator to inform PostgreSQL how this index should be generated which is specified in the strategy variable in @GeneratedValue. Please refer to <https://www.doctrine-project.org/projects/doctrine-orm/en/2.6/reference/basic-mapping.html#identifier-generation-strategies> for more details on various strategies available. You can also specify the initial value (i.e., initialValue in @SequenceGenerator) and also the increment (i.e., allocationSize in @SequenceGenerator)

package de.jonashackt.springbootvuejs.domain;

import java.util.Objects;

import javax.persistence.\*;

@Entity

@Table(name = "gdsrecordview")

public class gdsrecordview {

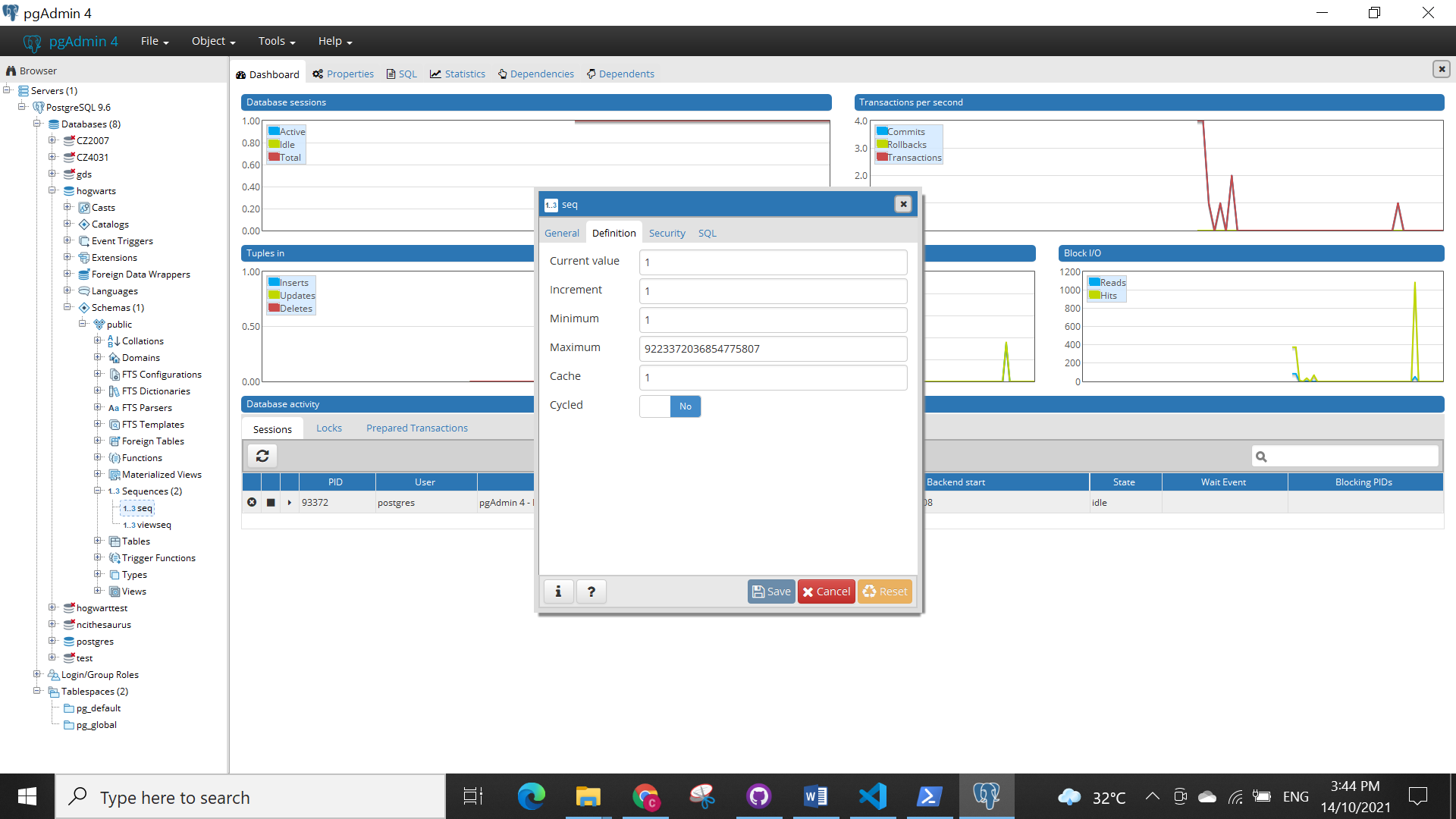
    @Id

    @SequenceGenerator(name="gds\_sequence",sequenceName="viewseq", initialValue=1, allocationSize=1)

    @GeneratedValue(strategy=GenerationType.SEQUENCE, generator="gds\_sequence")

    private long id;

In PostgreSQL, the sequence generated can be viewed in the pgAdmin.



1. In the case of PostgreSQL as RDBMS, when the springboot-vue application is started, the schema-postgres.sql script is executed provided that the automatic schema creation is turned off. This script allows you to initialize your database.
2. For this project, the main processing of PostgreSQL related tasks such as creation of tables, updates, insertion and deletion is handled by postgresController.java. In PostgreSQL, the tables are either related to thesaurus (NCI thesaurus, NCI metathesaurus and Cellosaurus) or related to GDS records. Functions in postgresController.java is differentiated based on TH\_ prefix and GDS\_prefix which refer to functions associated to thesaurus-related tables and GDS-related tables, respectively.

**Backend Code Structure**

The backend consists of several main packages (configuration, controller, domain, exception, repository, service and thread).

1. Configuration Package
   * gdsrecordAsyncConfiguration.java: This file is responsible for enabling asynchronous retrieval of the gdsrecord.
2. Controller Package
3. Domain Package
4. Exception Package
5. Repository Package
6. Service Package
7. Thread Package