# **Let’s Deploy Code Compiler**

**Github repository link:** [**https://github.com/kothapalliyamini91/Code-Compiler**](https://github.com/kothapalliyamini91/Code-Compiler)

# **Overview:**

Building an online code editor and compiler can seem too complicated, but we can break it down into two pieces.

1. API running on the backend server, which will take a piece of code and language as input and output the answer after running the code on the server
2. Frontend code editor- We can choose the language and edit and change the code here. Then we make a post request to the backend API and show output on the website. website.

# **Introduction**

Judge0is a robust, scalable, and open-source online codeexecution system that can be used to build a wide range of applications that need online code execution features. It is used in multiple competitive programming platforms, code editors, and e-learning platforms.

# **Install Docker and Docker Compose**

## [judge0/judge0](https://github.com/judge0/judge0/blob/master/CHANGELOG.md#deployment-procedure)

### [**This is the Changelog for Judge0. The Changelog for Judge0 Extra can be found here. Learn more about their difference…**](https://github.com/judge0/judge0/blob/master/CHANGELOG.md#deployment-procedure)

<https://github.com/judge0/judge0/blob/master/CHANGELOG.md#deployment-procedure>

Install docker and docker-compose from the official page.

Make sure [docker](https://medium.com/javarevisited/top-15-online-courses-to-learn-docker-kubernetes-and-aws-for-fullstack-developers-and-devops-d8cc4f16e773)is running

sudo systemctl status docker

Make docker-compose is running

docker-compose --version

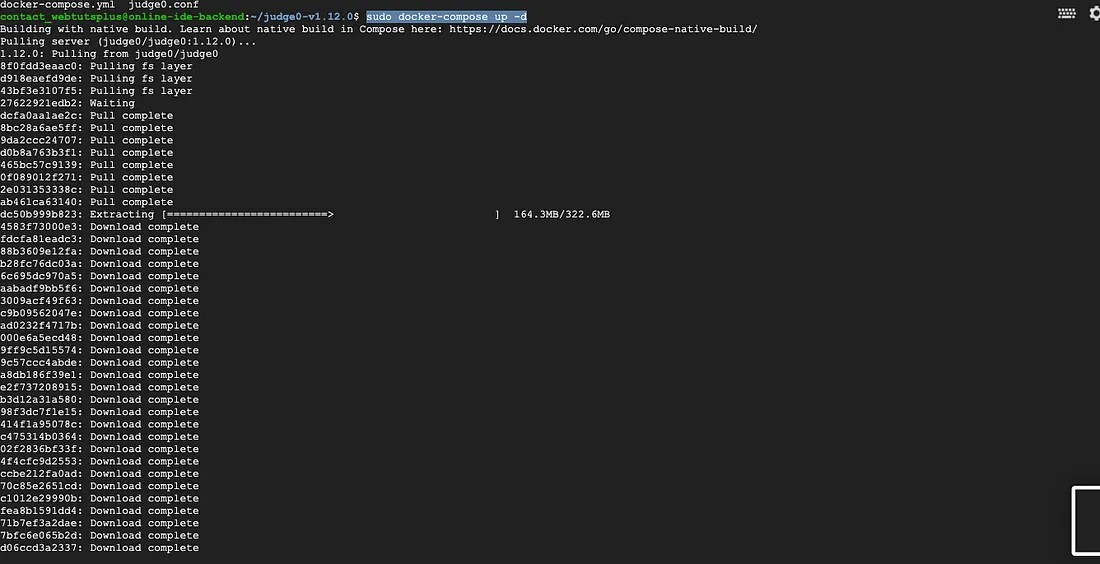
Download and extract release judge0 archive

wget https://github.com/judge0/judge0/releases/download/v1.12.0/judge0-v1.12.0.zip  
unzip judge0-v1.12.0.zip

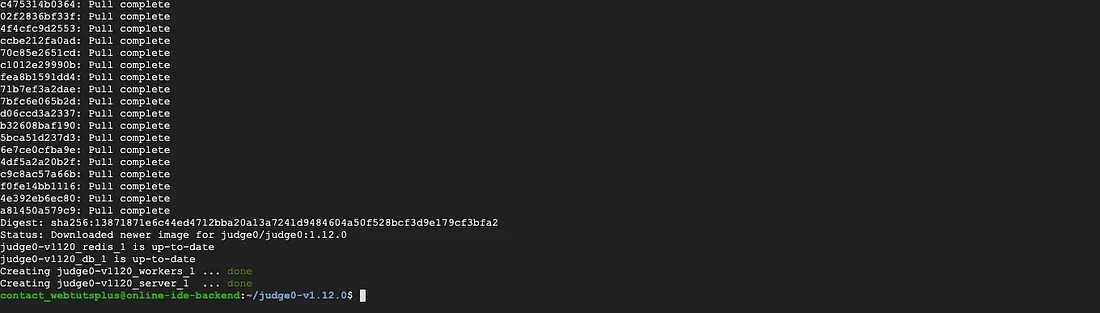
Run all services and wait a few seconds until everything is initialized

cd judge0-v1.12.0  
sudo docker-compose up -d db redis  
sleep 10s  
sudo docker-compose up -d  
sleep 5s

Docker is working



we will see this screen.



An instance of Judge0 v1.13.0 is now available at http://<IP ADDRESS OF YOUR SERVER>

If we open <http://10.0.2.15/dummy-client.html> and execute a sample code

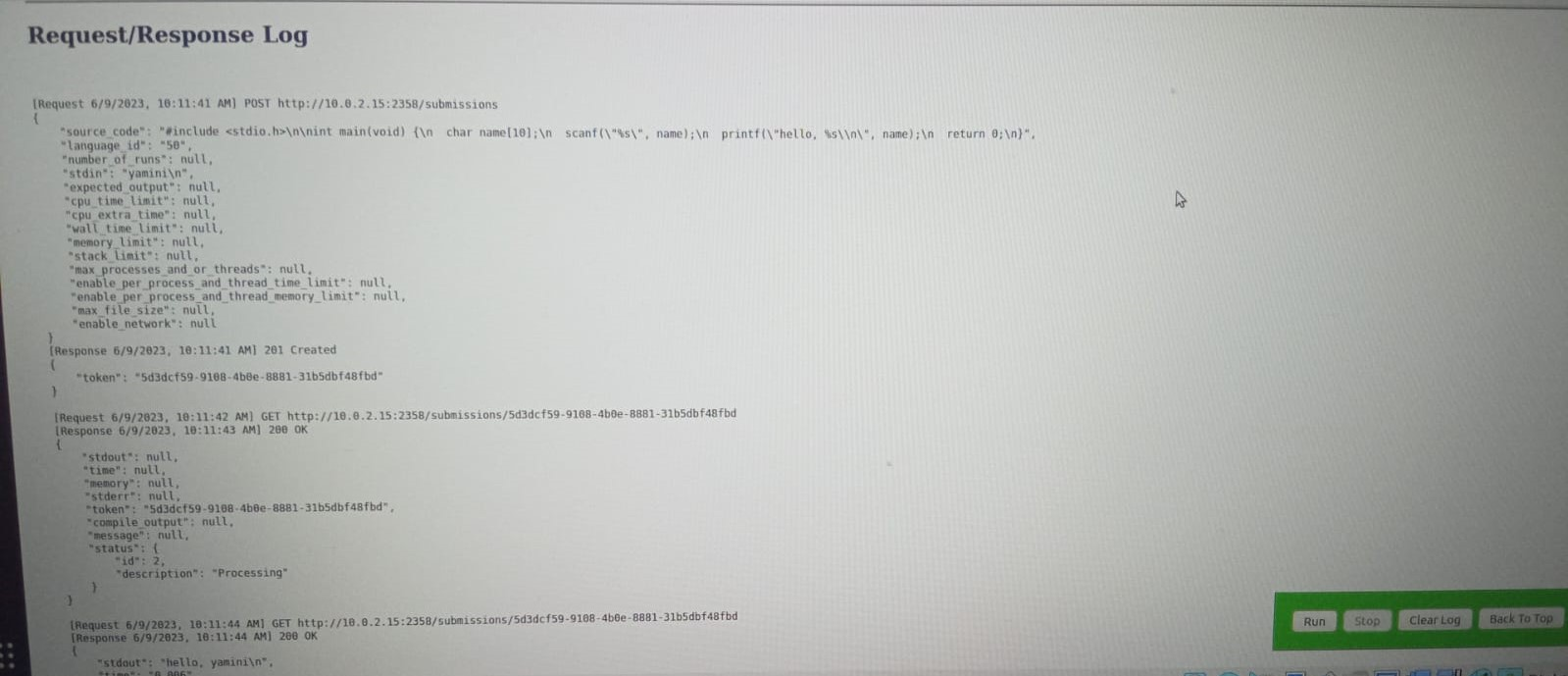
## [Judge0 Dummy Client](http://34.72.83.62/dummy-client.html)

### [**Judge0 dummy client can be used to try and test features of Judge0. Data is not validated on client side. Client first…**](http://34.72.83.62/dummy-client.html)

10.0.2.15



Running Hello World in the online judge



# **Backend API**

Using  [**Judge0**](https://judge0.com/) for code execution.

Judge0 is a robust, scalable, and open-source online codeexecution system that can be used to build a wide range of applications that need online code execution features. It is used in multiple competitive programming platforms, code editors, and e-learning platforms.

When write a code, we pass the code as a **POST**request to http://10.0.2.15/submissions and we get a token as a response. Let’s say we get a response as 3b47cfac-eae9-4412-b845-42e31a551873 . So this is the token that needs to be sent to the server for our next GET request.

We do a **GET**request to http://10.0.2.15/submissions/3b47cfac-eae9-4412-b845-42e31a551873 . The response we get from this network call is our result. We do this network call after some timeout because sometimes our code may need some time to execute.

If the answer is ready then the second API call will give our answer. In our code, we are using a 3-second timeout.



**PARAMETERS:**

**API URL**: The URL endpoint of the API you are trying to access.

**Authentication Header: The name and format of the header used for authentication.**

**Authentication Token**: The value of the authentication token or credentials.

**Authorization Header**: The name and format of the header used for authorization.

**Authorization Token**: The value of the authorization token or credentials.

**Language ID:** The specific programming language ID or name to use executing the code.

**Number of Runs:** The desired number of times the program should be executed.

**CPU Time Limit:** The maximum amount of CPU time allowed for program execution (in seconds).

**CPU Extra Time:** Additional time allowed after the CPU Time Limit is reached (in seconds).

**Wall Time Limit:** The maximum amount of wall clock time (real-world time) allowed for program execution (in seconds).

**Memory Limit**: The maximum amount allocated for the program execution (in bytes).

**Stack Limit**: The maximum stack size allocated for program execution (in bytes).

**Max Processes or Threads**: The maximum number of processes and threads allowed for program execution.

**Enable/Disable Per Process and Thread Time Limit:** A flag indicating whether per process thread time limits should be enabled or disabled.

**LANGUAGE IDS:**

Here are the language IDs for some commonly used programming languages

These language IDs can be used when making API requests to specify the programming language for code execution.

C: 50

C++: 54

Java: 62

Python: 71

JavaScript: 63

Ruby: 72

Go: 68

Swift: 74

PHP: 68

Rust: 79

Build a Code compiler application using React and Django.

[React](https://reactjs.org/) is a front-end JavaScript library that uses components to create user interfaces for single-page applications.

[Django](https://www.djangoproject.com/) is a Python backend web framework used to build scalable and secure website applications.

I am going to create an application that consumes [React](https://reactjs.org/) for the user interface and [Django](https://www.djangoproject.com/) for the API of our application using the Django REST framework (DRF).

### **Step 1: Backend using Django**

From a terminal create a new project directory called django-react-OnlineideApp

mkdir django-react-onlineideApp

cd django-react-OnlineideApp

We must have a virtual environment activated to be able to install Django.

Let’s create a virtual environment and activate it:

pip install pipenv

pipenv shell

Install Django and create a project named backend:

pipenv install django

django-admin startproject backend

Now that we have created our project, let’s go ahead and create an application called Onlineide and migrate the models into the database.

If everything works you should see the “Congratulations” page from Django.

Navigate to backend/settings.py and add Online to the list of INSTALLED\_APPS

INSTALLED\_APPS = [

'django.contrib.admin',

'django.contrib.auth',

'django.contrib.contenttypes',

'django.contrib.sessions',

'django.contrib.messages',

'django.contrib.staticfiles',

'Online',

]

We’ll go ahead and set up our model for Online item fields.

Modify Online/models.py as follows:

from django.db import models

class Online(models.Model):

LANGUAGE\_CHOICES = (

(50, 'c'),

(54, 'cpp'),

(62, 'java'),

(71, 'python'),

)

user\_id = models.CharField(max\_length=100)

source\_code = models.TextField(blank=False)

output = models.TextField(null=True, blank=True)

language\_name = models.IntegerField(choices=LANGUAGE\_CHOICES)

input = models.TextField(null=True, blank=True)

error = models.TextField(null=True, blank=True)

status = models.TextField(null=True, blank=True)

time = models.TextField(null=True, blank=True)

question\_id = models.IntegerField(null=True, blank=True)

questionDescription = models.TextField(null=True, blank=True)

Inputcase1 = models.TextField(null=True, blank=True)

Inputcase2 = models.TextField(null=True, blank=True)

Inputcase3 = models.TextField(null=True, blank=True)

Outputcase1 = models.TextField(null=True, blank=True)

Outputcase2 = models.TextField(null=True, blank=True)

Outputcase3 = models.TextField(null=True, blank=True)

def \_\_str\_\_(self):

return self.user\_id

Let us run migrations to add our model to the database schema.

python manage.py makemigrations

python manage.py migrate

Django comes with a built-in admin interface. The interface allows administrators and authorized users to perform actions directly to the objects defined in the models.

We can add models to our Admin page using the admin.site.register() functions. In the Online app’s admin.py, let’s add the model to our admin page.

from django.contrib import admin

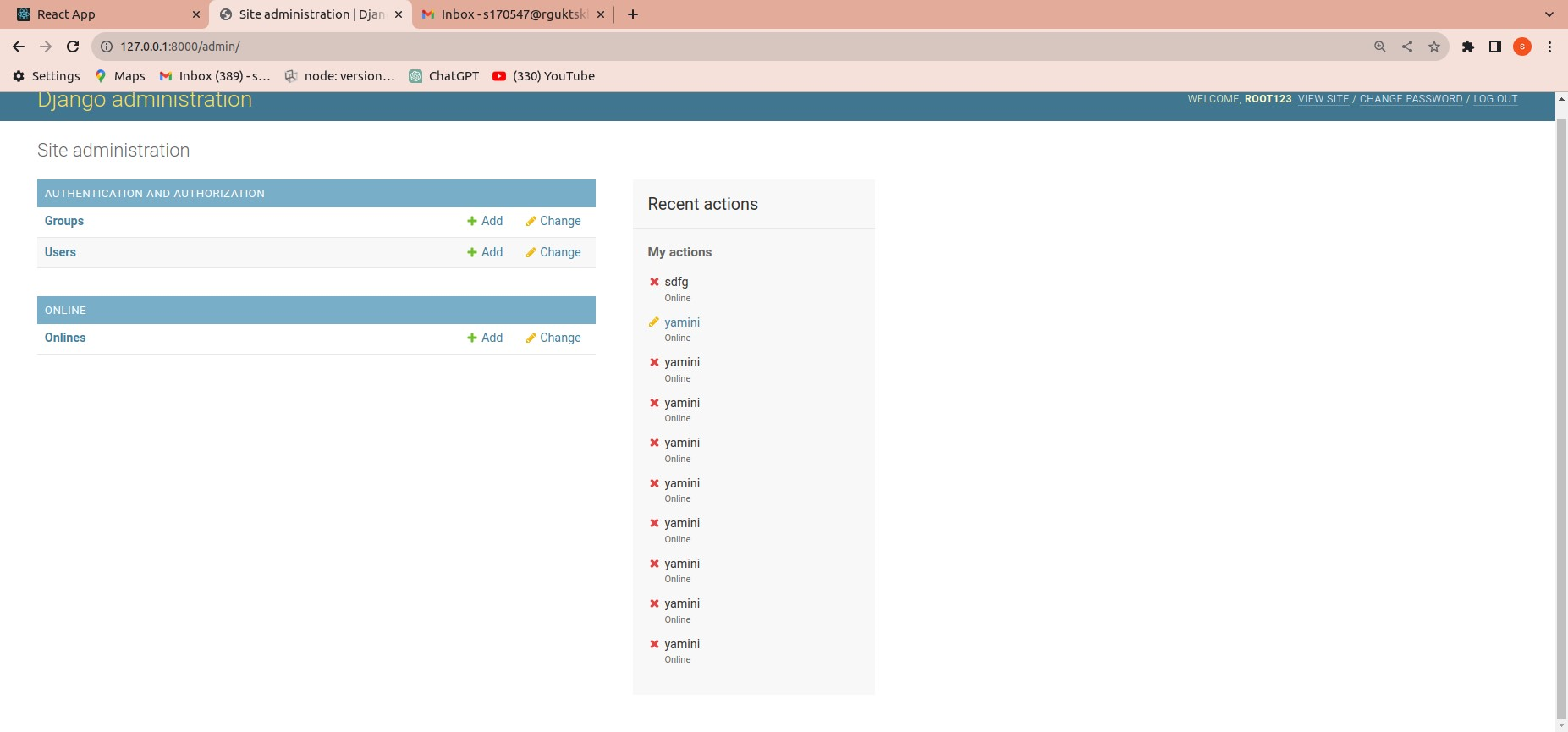
from .models import Online

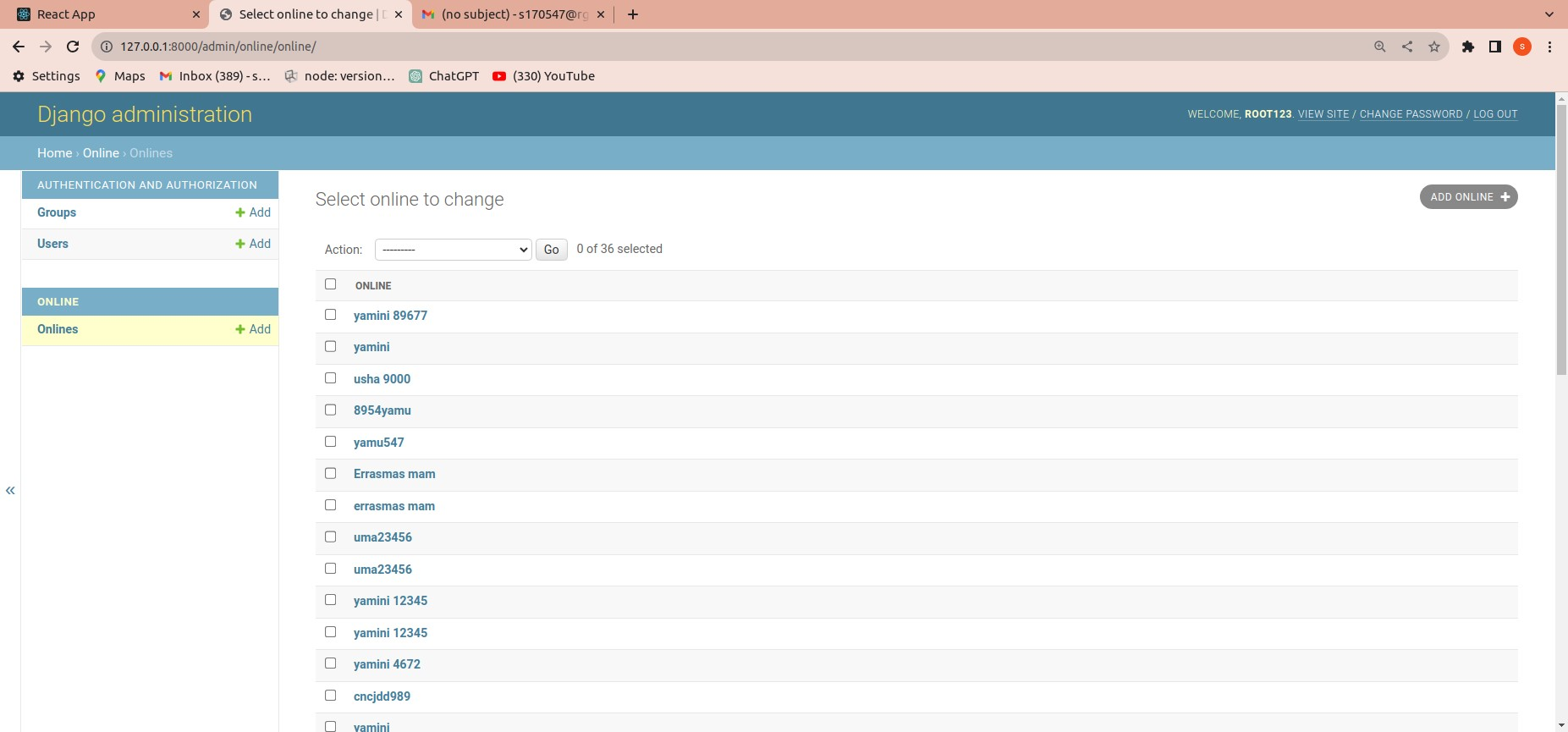
admin.site.register(Online)

Let’s create a superuser for the admin page to login.

python manage.py createsuperuser

This will prompt you to enter the username, email, password, password(again). We can open the admin page using the following link http://localhost:8000/admin.

python manage.py runserver

Now add and delete items from the admin page.   


### **Step 2: Putting in the APIs**.

Install the djangorestframework and django-cors-headers:

pipenv install djangorestframework django-cors-headers

Add rest\_framework and corsheaders to the INSTALLED\_APPS in backend/settings.py file and modifiy the MIDDLEWARE:

INSTALLED\_APPS = [

'django.contrib.admin',

'django.contrib.auth',

'django.contrib.contenttypes',

'django.contrib.sessions',

'django.contrib.messages',

'django.contrib.staticfiles',

'Online',

'corsheaders',

'rest\_framework',

]

MIDDLEWARE = [

'corsheaders.middleware.CorsMiddleware',

'django.middleware.security.SecurityMiddleware',

'django.contrib.sessions.middleware.SessionMiddleware',

'django.middleware.common.CommonMiddleware',

'django.middleware.csrf.CsrfViewMiddleware',

'django.contrib.auth.middleware.AuthenticationMiddleware',

'django.contrib.messages.middleware.MessageMiddleware',

'django.middleware.clickjacking.XFrameOptionsMiddleware',

]

Add this code snippet in backend/settings.py file:

CORS\_ORIGIN\_WHITELIST = [

'http://localhost:8000',

]

Django-cors-headers is an HTTP-header-based that allows a server to indicate any other origins to your Django application. [Cross-origin Resource Sharing (CORS)](https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS). Within the CORS\_ORIGIN\_WHITELIST, localhost:8000 will serve as our port.

Now, let’s create a serializer file.

A serializer is a component that converts Django models to JSON objects and vice-versa.

touch Online/serializers.py

Let’s add this to the serializers.py file:

from rest\_framework import serializers

from .models import \*

class OnlineSerializer(serializers.ModelSerializer):

class Meta:

model = Online

fields = '\_\_all\_\_'

From the rest\_framework package, we import the serializers. We create a class, OnlineSerializer that extends from the ModelSerializer class. We then go ahead and specify the model and fields we want to be returned.

Let’s also update the Online/views.py:

import logging

from .models import Online

from .serializers import OnlineSerializer

from rest\_framework.status import HTTP\_201\_CREATED

from rest\_framework.response import Response

from rest\_framework.views import APIView

log = logging.getLogger(\_\_name\_\_)

class Useradd(APIView):

def post(self, request):

try:

my\_dict = request.data

print(request.data)

user\_info = Online()

user\_info.user\_id = my\_dict['userId']

user\_info.source\_code = my\_dict['sourceCode']

user\_info.language\_name = my\_dict['language']

user\_info.input = my\_dict['input']

user\_info.output = my\_dict['output']

user\_info.status = my\_dict['executionStatus']

user\_info.time = my\_dict['executionTime']

user\_info.error = my\_dict['error\_']

user\_info.question\_id = my\_dict['questionNumber']

user\_info.questionDescription = my\_dict['questionDescription']

user\_info.save()

return Response(True, status=HTTP\_201\_CREATED)

except Exception as e:

log.exception(e)

return Response(False) # Return an appropriate error response

Before creating webpages in Django we must define our URLs. A URL is an address to which a webpage is served. Defining URLs describes what requests are returned from the views when the templates are rendered in the browser.

In the backend/urls.py we define the URL routes for the API:

from django.contrib import admin

from django.urls import path,include

urlpatterns = [

path('admin/', admin.site.urls),

path("server/", include("online.urls")),

]

### python manage.py runserver

### **Step 3: Frontend using React**

To install React we use the following command:

-g stands for global as we are first installing create-react-app globally:

npm install -g create-react-app

While in the parent directory - django-react-OnlineApp- create a React application, frontend:

create-react-app frontend

To start the server:

cd frontend

npm start

You should be able to see the default React app by now.

When we open our index.js file it should resemble the code below.

import React from 'react';

import ReactDOM from 'react-dom/client';

import './index.css';

import App from './App';

import reportWebVitals from './reportWebVitals';

const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(

<React.StrictMode>

<App />

</React.StrictMode>

);

ReactDOM.render() renders a React element into the DOM in the given container element. It takes two arguments. The first is the JSX being rendered, and the second displays the container element on the HTML page.

Substitute the below code in src/App.js:

import OnlineIde from "./Components/OnlineIde";

function App() {

return (

<div>

<OnlineIde/>

</div>

);

}

export default App;

start by rendering a list of items. The list will be artificial data but later we’ll fetch the data from the API we created in the previous steps.

Used the built-in JavaScript **map** functionality in our JavaScript XML (JSX).

It is time to consume the API we created earlier.

cd backend

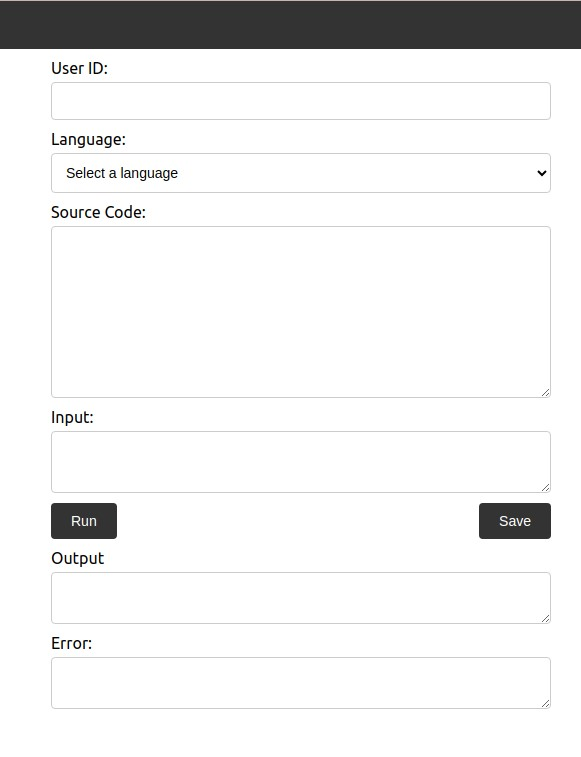
python manage.py runserver

Needs to modify the frontend/package.json by adding proxy. A proxy is used in the development environment to facilitate communication between the server and the UI since the backend and the UI will be running on different ports.

It is time to consume the API we created earlier.

cd backend

python manage.py runserver

Need to modify the frontend/package.json by adding proxy. A proxy is used in the development environment to facilitate communication between the server and the UI since the backend and the UI will be running on different ports.

The **proxy** enables us to use Django’s localhost which handles our API requests.

Let’s go ahead and add it.

"name": "client",

"version": "0.1.0",

"private": true,

"dependencies": {

"@testing-library/jest-dom": "^5.16.5",

"@testing-library/react": "^13.4.0",

"@testing-library/user-event": "^13.5.0",

"axios": "^1.4.0",

"react": "^18.2.0",

"react-dom": "^18.2.0",

"react-scripts": "5.0.1",

"web-vitals": "^2.1.4"

},

To consume our API instead of the artificial data, update the frontend/src/App.js with the snippet below:

import React, { useState } from 'react';

import './ide.css';

import axios from 'axios';

import questionsData from './decription';

const baseURL = 'http://172.18.0.1:2358/submissions';

const baseURL1 = 'http://127.0.0.1:8000/server/ide/';

export default function OnlineIde() {

const [userId, setUserId] = useState('');

const [language, setLanguage] = useState('');

const [sourceCode, setSourceCode] = useState('');

const [input, setInput] = useState('');

const [output, setOutput] = useState('');

const [selectedQuestionIndex, setSelectedQuestionIndex] = useState(0);

const [modalIsOpen, setModalIsOpen] = useState(false);

const [questionNumber, setQuestionNumber] = useState('');

const [questionDescription, setQuestionDescription] = useState('');

const [questions, setQuestions] = useState(questionsData);

const [executionStatus, setExecutionStatus] = useState('');

const [executionTime, setExecutionTime] = useState('');

const [error\_, setError\_] = useState('');

const [savedData, setSavedData] = useState([]);

// const [sampleTestCases, setSampleTestCases] = useState([]);

const [outputs, setOutputs] = useState([]);

const [testCasesPassed, setTestCasesPassed] = useState(0);

const [failedTestCases, setFailedTestCases] = useState([]);

const handleSubmit = async (e) => {

e.preventDefault();

setExecutionTime('');

setError\_('');

const data = {

source\_code: sourceCode,

language\_id: language,

stdin: input,

};

try {

const response = await axios.post(baseURL, data);

const token = response.data.token;

setTimeout(() => {

fetchOutput(token);

}, 4000);

const sampleTestCases = [

{ input: '6', expectedOutput: 'The given number is not prime' },

{ input: '7', expectedOutput: 'The given number is prime' },

{ input: '8', expectedOutput: 'The given number is not prime' },

];

// Define a function to run the code for each test case

const runTestCases = async () => {

let passedCount = 0;

const newOutputs = [];

const failedCases = [];

for (const testCase of sampleTestCases) {

try {

const response = await axios.post(baseURL, {

...data,

stdin: testCase.input,

});

const token = response.data.token;

// eslint-disable-next-line no-loop-func

setTimeout(async () => {

const output1 = await fetchOutput1(token);

console.log('Input:', testCase.input);

console.log('Expected Output:', testCase.expectedOutput);

console.log('Actual Output:', output1);

console.log('---');

newOutputs.push({ input: testCase.input, output1 });

if (output1 === testCase.expectedOutput) {

passedCount++;

} else {

failedCases.push({

input: testCase.input,

expectedOutput: testCase.expectedOutput,

actualOutput: output1,

});

}

}, 4000);

Before we continue let’s install axios. It allows our applications to make a request to external endpoints. We use it to perform CRUD operations on our API.

npm install axios@0.21.1

Certainly, I'll provide an overview of what the code is doing step by step:

**Imports and Initial Setup:**

Import React, useState, and Axios.

Import CSS styles from a file named 'ide.css'.

Define two base URLs, one for executing code submissions (baseURL) and another for saving data (baseURL1).

**State Variables:**

Define state variables to manage various aspects of the code execution process:

**UserId:**  User ID input.

**Language:** Selected programming language.

**Source code:** Source code entered by the user.

**Input:** Input data for code execution.

**Outpu**t: Output generated by code execution.

**ExecutionStatus**: Execution status (e.g., "Accepted," "Compilation Error," etc.).

**Execution time:** Time taken for code execution.

**Error:** Error message, if any.

Handle Submit:

On form submission, an asynchronous function handleSubmit is triggered.

A POST request is made to the base URL with source\_code, language\_id, and stdin (input data).

Upon success, the response token is used to fetch the output after a delay using the fetchOutput function.

Fetch Output:

An asynchronous function fetchOutput is used to get the output using the response token from the previous request.

A GET request is made to baseURL/token.

The output, error message, execution time, and status are updated in the state.

Handle Language Change and Source Code:

When the user selects a language or changes the source code, relevant state variables are updated, and input and output are cleared.

Handle Save:

The handle save function is triggered when the "Save" button is clicked.

The current data is collected and saved in the saved data state array.

A POST request is made to baseURL1 to save the data on the server.

JSX Rendering:

Displays a navigation bar and a form container.

The form includes input fields for User ID, Language, Source Code, Input, and buttons for "Run" and "Save."

Displays the output, error message, execution time, and execution status.

Commented-out code indicates a section where saved data can be displayed.

The code is structured as a React functional component. It provides an interface for users to input code, execute it, see the output, save data, and view execution-related information.

The code interactions are handled through asynchronous HTTP requests and state management using the useState hook.

### **Step 4: Testing**

Let’s start our backend server made using Django by running the following commands:

python3 manage.py runserver

Let’s start our React application by running the following commands:

npm start

### **Conclusion**

I have learned how to integrate a Django application with React serving as the frontend.

