TECHNICAL UNIVERSTY OF KOSICE

FACULTY OF ELECTRICAL ENGINEERING AND INFORMATICS

Full stack To-do List Application

React with Django

Kosice, 2024

SARUKESH BOOMINATHAN

TECHNICAL UNIVERSTY OF KOSICE

FACULTY OF ELECTRICAL ENGINEERING AND INFORMATICS

Documentation

Django with React Project

Study programme:	Bachelors in Informatics
Department:	Department of Informatics
Subject:	Programming
Supervisor:	Ing. Tomáš Kormaník
	Ing. Marek Horváth

Abstract:

The project is a comprehensive task management web application built using React.js for the frontend and PostgreSQL for the backend database. It offers a modern and intuitive user interface for efficiently managing tasks. Users can sign up securely, log in, and access personalized task management features. The application leverages RESTful APIs to interact with the PostgreSQL database, providing functionalities such as task creation, deletion, search, and data retrieval. Additionally, users can download task data and import tasks from external files, enhancing flexibility and usability. With its responsive design and seamless integration of frontend and backend technologies, the project aims to streamline task management processes and enhance productivity for individuals and teams.

Acknowledgement:

I extend my sincere appreciation to the individuals who have contributed to the development of this project:

I would like to acknowledge the support and assistance provided by my colleagues and staff members throughout the project. Their collaboration and dedication have been invaluable in overcoming challenges and achieving milestones.

Special thanks to my mentors and advisors for their guidance, feedback, and encouragement. Their expertise and insights have been instrumental in shaping the project's direction and improving its quality.

I would also like to express my gratitude to myself for the hard work, perseverance, and commitment invested in completing this project. Despite the hurdles faced, I remained focused and dedicated to delivering a successful outcome.

This project is a testament to the collective effort and collaboration of all those involved, and I am grateful for the opportunity to have worked with such a supportive team.

1st Problem set:

Project Overview

This full-stack web application, crafted by Sarukesh Boominathan, seamlessly blends frontend and backend components to deliver a holistic user experience. Here's a detailed examination of each element:

Frontend Components:

1. Header Component:

- Presents the project title and an accompanying image.
- Emphasizes Sarukesh Boominathan's name as a focal point.

2.Navbar Component:

- Furnishes navigation links for effortless exploration within the application.
- Incorporates a direct link to an "About Me" section.
- Offers buttons for data retrieval and documentation download.

3. Input Form Component:

- Facilitates user submission of personal details: name, mobile number, and email address.
- User-centric design ensures ease of data input.

Backend Components:

1. URL Configuration:

- Specifies routing for various endpoints within the application.

2.Views:

- Hosts functions dedicated to request handling and response generation.
- Encompasses operations like rendering HTML templates, processing form data, and serving API endpoints.

3. Settings:

- Houses configuration parameters pertinent to the Django project.
- Manages settings for static files, internationalization, and other project-specific configurations.

This project overview encapsulates the core functionalities provided by Sarukesh Boominathan, accentuating both frontend and backend components, along with their respective roles and functionalities.

Introduction

Welcome to the full-stack web application crafted by Sarukesh Boominathan, which seamlessly integrates React.js and Django frameworks. This project serves as a comprehensive platform for users to interact with various features, leveraging frontend technologies for user input collection and backend technologies for data storage and retrieval.

In this application, React.js is employed on the frontend to provide an intuitive user interface for seamless interaction. Users can conveniently inputs, including id, title, and due date, via a user-friendly form. React.js ensures a smooth and responsive experience, enhancing user engagement and satisfaction.

On the backend, Django plays a pivotal role in handling data storage and management. Instead of employing a traditional database system, Django is configured to store user data in a text file. This approach offers simplicity and efficiency, eliminating the need for complex database configurations while ensuring data persistence and accessibility.

Additionally, the application features a convenient **documentation download** functionality. Users can effortlessly access project documentation by clicking on the designated button, simplifying the process of acquiring essential information about the application's features and functionalities.

With the combination of React.js and Django, this web application delivers a robust and user-centric solution, catering to the diverse needs of users while maintaining simplicity and efficiency in data management and documentation access.

Starting a Project:

Create a Folder for your new project

Click the file path and type “cmd” to open Command prompt inside your folder

I have used a code editor called “Visual Studio Code” for my project. So I am using a command called “code .” to open Visual Studio Code inside my folder.

Now click on “Terminal” in navigation bar and click “New Terminal”

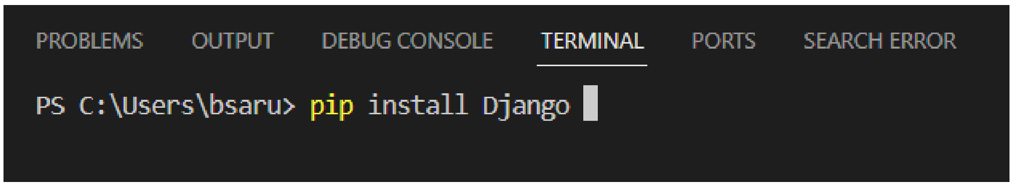
Use the terminal to navigate to the directory where you want to create your Django project.

Now follow the below mentioned steps to start you Full Stack project .

Django Installation

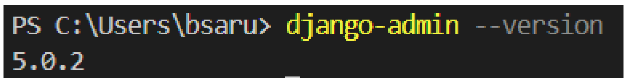
Install Django

Once you ensure that you are in the correct directory, you can install Django pip:



```
PS C:\Users\bsaru> pip install Django
```

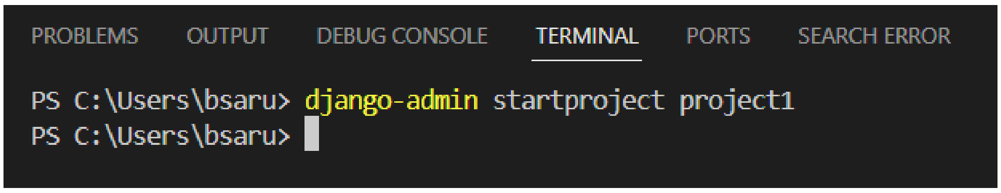
You can verify that Django has been installed correctly by checking its version:



```
PS C:\Users\bsaru> django-admin --version
5.0.2
```

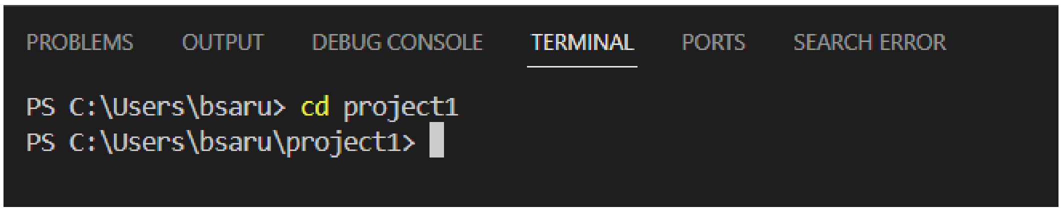
Starting a Project in Django

To start a project in Django, Use the “Django-admin” command. This step must be continued on the same directory:



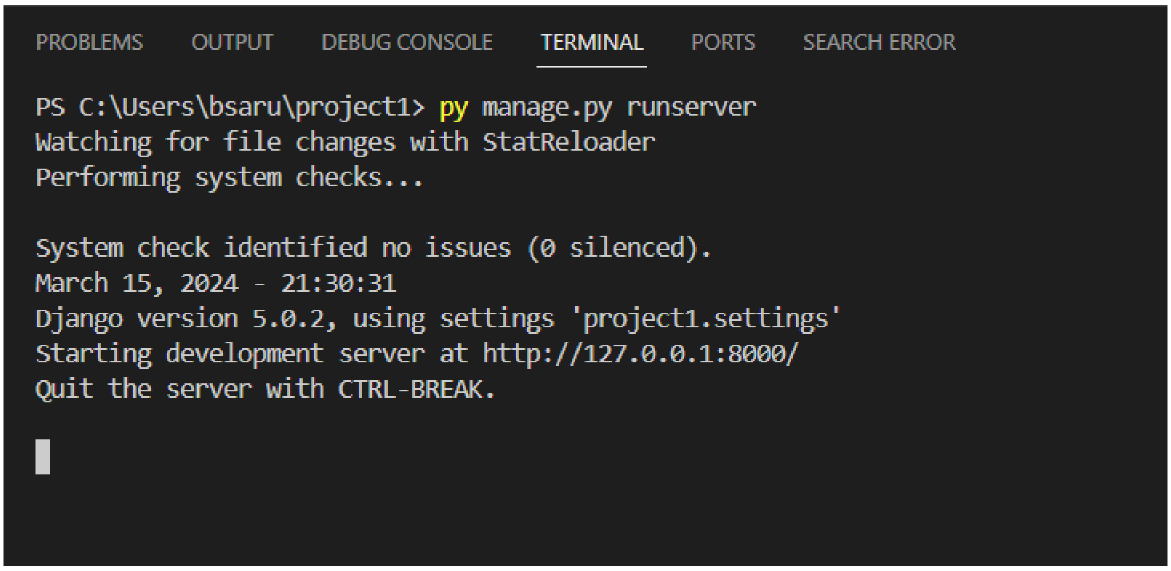
```
PS C:\Users\bsaru> django-admin startproject project1
PS C:\Users\bsaru>
```

Once it is installed you can navigate to your project folder using:



```
PS C:\Users\bsaru> cd project1
PS C:\Users\bsaru\project1>
```

now I am inside my project folder. Now to start Django development server to see your project in action:



```
PS C:\Users\bsaru\project1> py manage.py runserver
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).
March 15, 2024 - 21:30:31
Django version 5.0.2, using settings 'project1.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
```

Files of Django

Manage.py

manage.py is a command-line utility provided by Django to perform various administrative tasks within a Django project. Here's a concise explanation:

Management Commands:

- Executes management commands such as creating applications, running development servers, and applying database migrations.

Project Configuration:

- Sets up the Django project environment, including settings, database connections, and installed applications.

Development Server:

- Starts the development server to run the Django project locally during development.

Database Migrations:

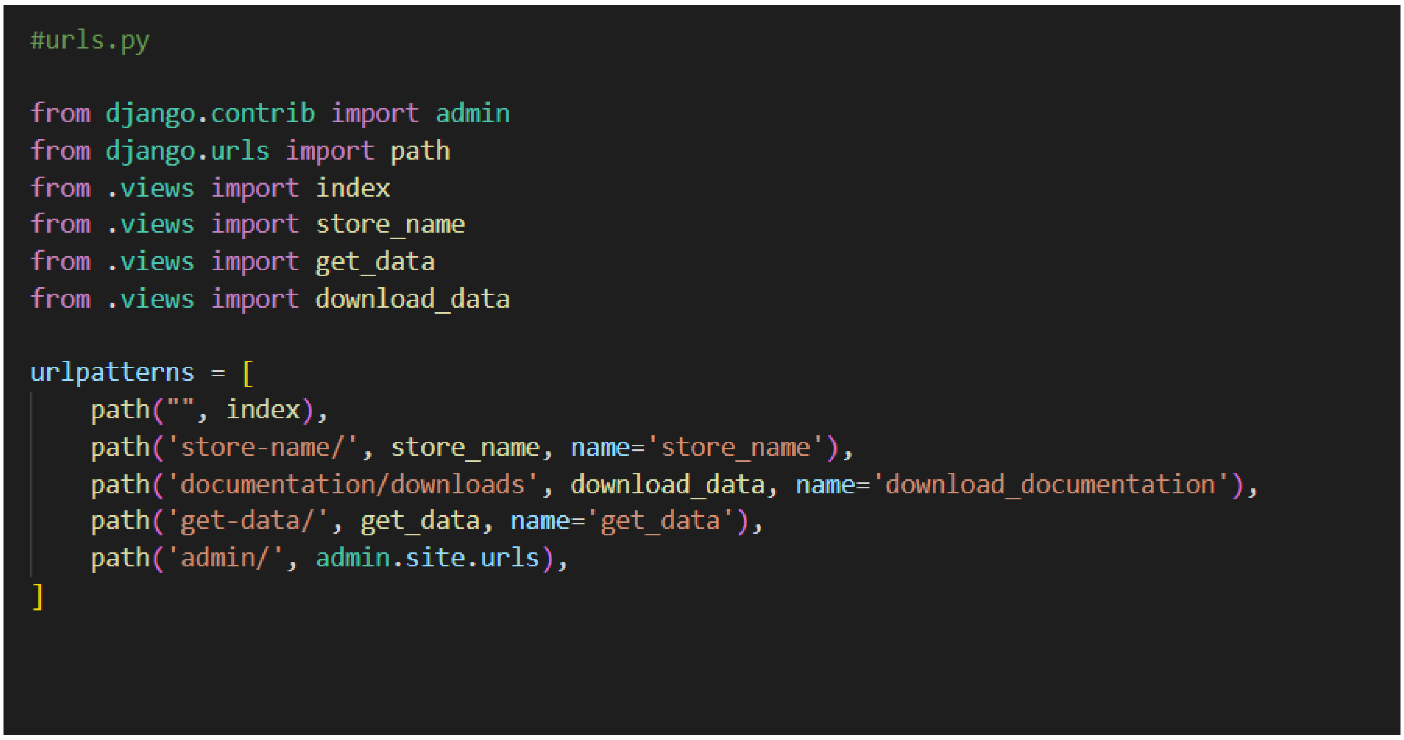
- Manages database schema changes and data migrations using Django's built-in migration system.

Utility Functions:

- Provides utility functions for tasks like creating superusers, running tests, and managing static files.

Urls.py

In Django, `urls.py` serves as a configuration file where you define URL patterns for your web application. It acts as a bridge between incoming URLs and the views that handle them. Each URL pattern is associated with a specific view function or class-based view, allowing you to map URLs to the appropriate functionality within your application. `urls.py` enables you to organize and manage your application's URL structure, providing a clear and structured approach to handling incoming requests.



```python
#urls.py

from django.contrib import admin
from django.urls import path
from .views import index
from .views import store_name
from .views import get_data
from .views import download_data

urlpatterns = [
    path("", index),
    path('store-name/', store_name, name='store_name'),
    path('documentation/downloads', download_data, name='download_documentation'),
    path('get-data/', get_data, name='get_data'),
    path('admin/', admin.site.urls),
]
```

This `urls.py` file configures the URL patterns for a Django project:

1. It imports necessary modules (`admin` and `path`) from Django.

2. It maps URLs to corresponding view functions (“index”, ”store_name”, “get_data”, “download_data”).

3. Each URL pattern is defined using the `path()` function.

4. The `name` parameter assigns unique names to URL patterns for easy referencing.

5. The `'admin/'` URL is reserved for Django's admin interface.

In essence, this file defines how incoming URLs are handled and which views should respond to them within the Django project.

Views.py

The `views.py` file contains Python functions that handle HTTP requests and generate HTTP responses in a Django project:

It imports necessary modules for handling requests and generating responses.

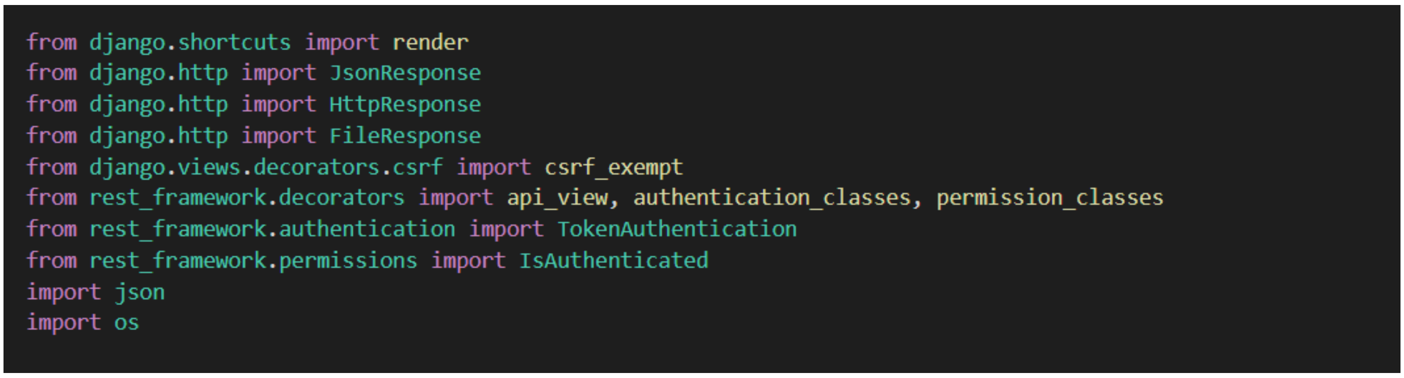
Each function represents a specific view or endpoint in the application.

The functions process incoming requests, perform necessary operations (such as data retrieval or manipulation), and return appropriate responses.

Views can render HTML templates, serve API endpoints, or perform other tasks based on the application's requirements.

View functions are linked to URL patterns defined in `urls.py`, determining which function should be invoked for a particular URL.

This `views.py` file contains Python functions used to handle HTTP requests and generate responses in a Django project:



```python
from django.shortcuts import render
from django.http import JsonResponse
from django.http import HttpResponse
from django.http import FileResponse
from django.views.decorators.csrf import csrf_exempt
from rest_framework.decorators import api_view, authentication_classes, permission_classes
from rest_framework.authentication import TokenAuthentication
from rest_framework.permissions import IsAuthenticated
import json
import os
```

- “HttpResponse”: Returns basic HTTP responses.

- “FileResponse”: Sends files as HTTP responses.

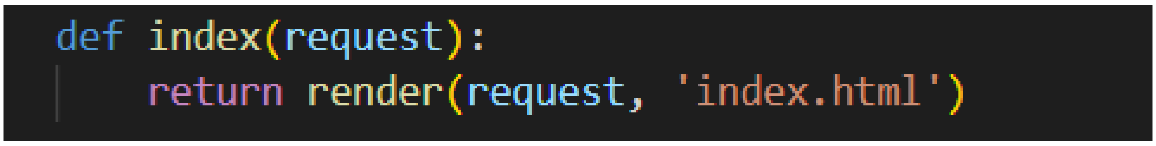
- “csrf_exempt”: Exempts views from CSRF protection.

- “api_view”, “authentication_classes”, “permission_classes’: Decorators for API views.

- “TokenAuthentication”, “IsAuthenticated”: Components for authentication and permissions.

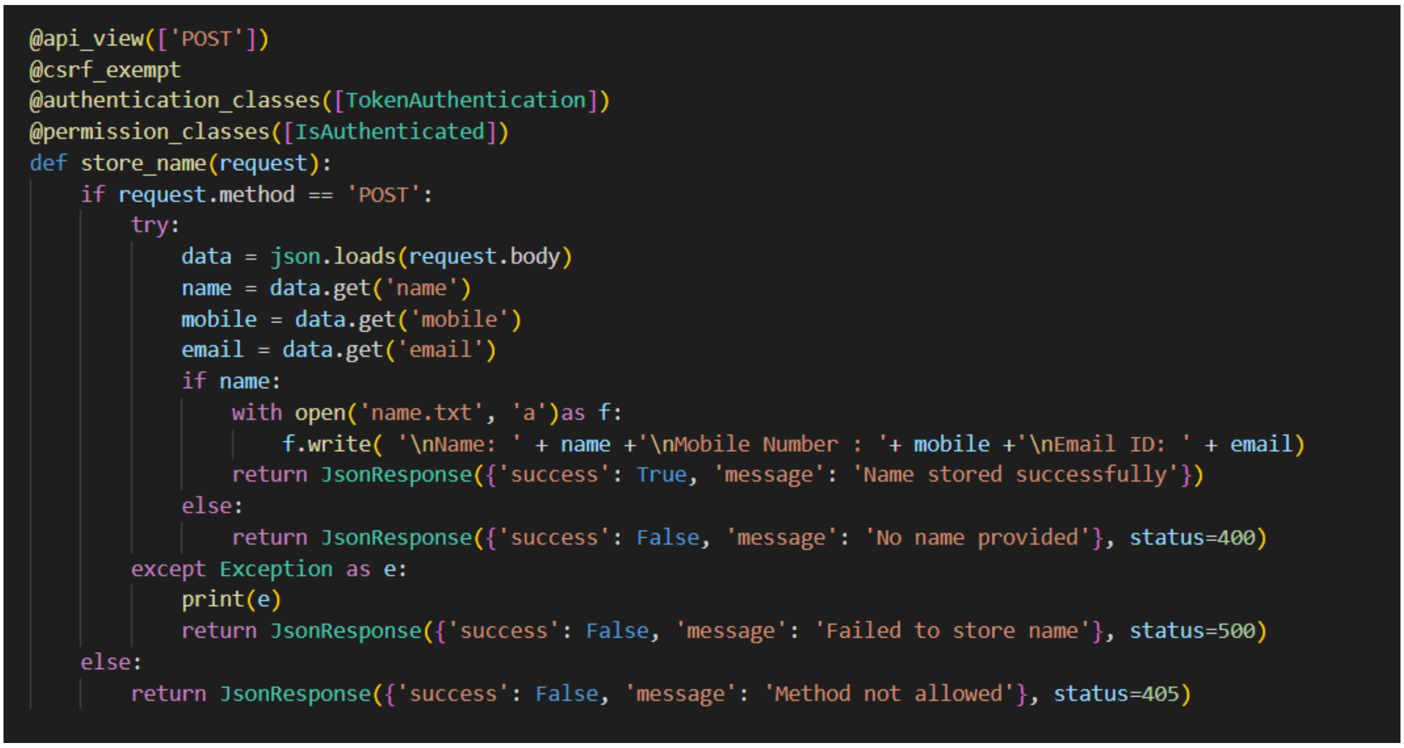
- “json”: Module for JSON serialization.

- “os”: Module for interacting with the operating system.



```python
def index(request):
    return render(request, 'index.html')
```

The `index` function renders the `index.html` template when a user visits the root URL of the application. “Index.html” serves as a static file which contain frontend code.



```
@api_view(['POST'])
@csrf_exempt
@authentication_classes([TokenAuthentication])
@permission_classes([IsAuthenticated])
def store_name(request):
    if request.method == 'POST':
        try:
            data = json.loads(request.body)
            name = data.get('name')
            mobile = data.get('mobile')
            email = data.get('email')
            if name:
                with open('name.txt', 'a')as f:
                    f.write( '\nName: ' + name +'\nMobile Number : '+ mobile +'\nEmail ID: ' + email)
                return JsonResponse({'success': True, 'message': 'Name stored successfully'})
            else:
                return JsonResponse({'success': False, 'message': 'No name provided'}, status=400)
        except Exception as e:
            print(e)
            return JsonResponse({'success': False, 'message': 'Failed to store name'}, status=500)
    else:
        return JsonResponse({'success': False, 'message': 'Method not allowed'}, status=405)
```

This “**store_name**” function is a view in the Django project, designed to handle POST requests to store user information. Here's a concise explanation of its functionality:

Decorator:

- @api_view(['POST']): Marks the function as an API view that only accepts POST requests.

CSRF Exemption:

- @csrf_exempt: Exempts the view from CSRF (Cross-Site Request Forgery) protection.

Authentication and Permission Classes:

- @authentication_classes([TokenAuthentication]): Specifies token-based authentication for the view.

- @permission_classes([IsAuthenticated]): Requires users to be authenticated to access the view.

Request Handling:

- Checks if the request method is POST.

- Parses the JSON data from the request body to extract name, mobile, and email.

- Stores the user information in a text file named name.txt.

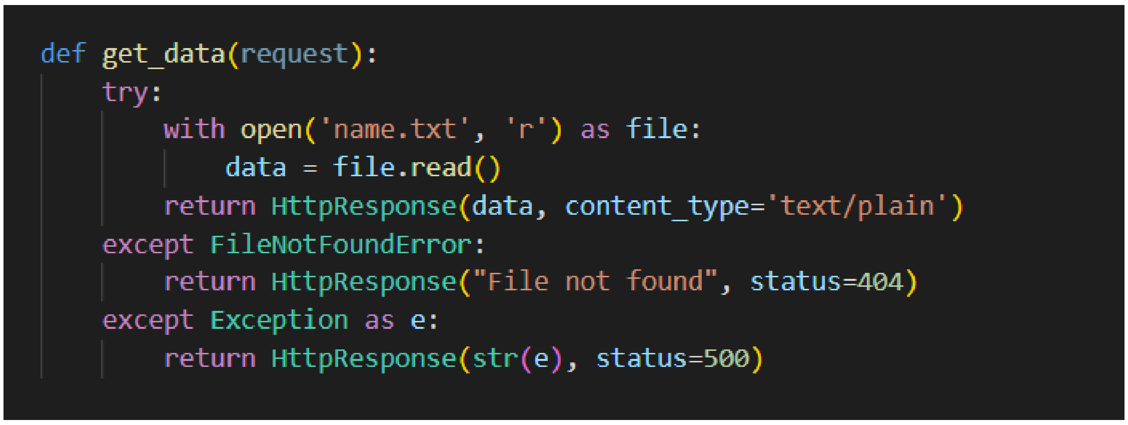
- Returns a JSON response indicating whether the operation was successful or not.

Error Handling:

- Handles exceptions that may occur during the process, such as JSON parsing errors or file writing errors.

- Returns appropriate error responses with status codes.

In summary, the “**store_name**” view function processes POST requests, extracts user data, stores it in a text file, and returns JSON responses to indicate the success or failure of the operation. It also includes measures for authentication, permission, and error handling.



```python
def get_data(request):
    try:
        with open('name.txt', 'r') as file:
            data = file.read()
        return HttpResponse(data, content_type='text/plain')
    except FileNotFoundError:
        return HttpResponse("File not found", status=404)
    except Exception as e:
        return HttpResponse(str(e), status=500)
```

The **get_data** function in the Django project serves as a view to retrieve stored data from a text file named **name.txt**. Below is a simplified explanation of its functionality.

- **File Reading:**

  - Attempts to open the name.txt file in read mode (“**r**”) using a **with statement.**

  - Reads the contents of the file into the **data** variable.

HTTP Response:

  - Returns an HTTP response containing the data read from the file.

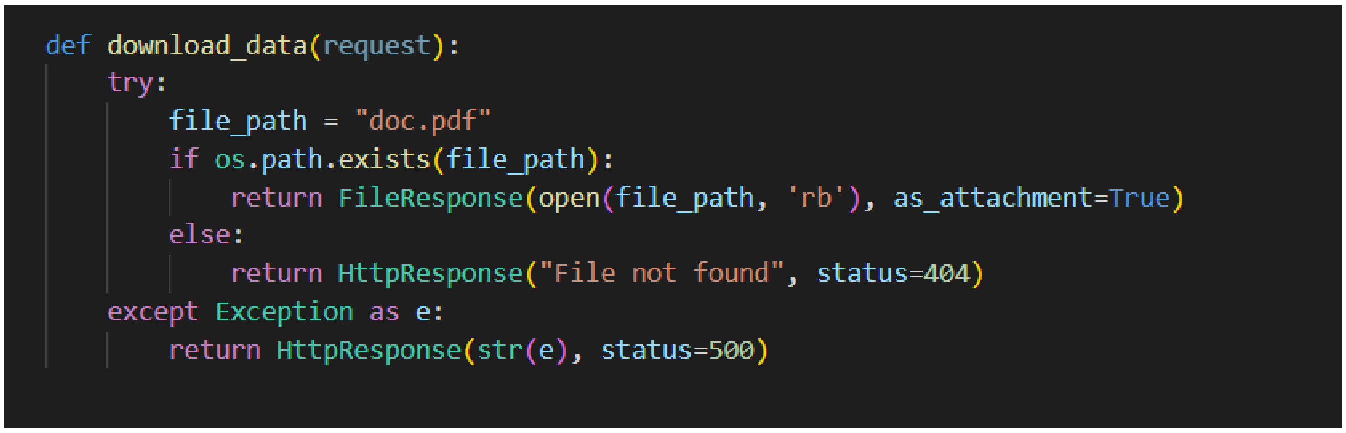
  - The **content_type** parameter specifies the MIME type of the response, set to **'text/plain'**.

Error Handling:

  - If the **name.txt** file is not found, returns an HTTP response with status code 404 (File Not Found).

- Handles any other exceptions that may occur during file reading and returns an HTTP response with status code 500 (Internal Server Error), along with the error message.

In summary, the **get_data** function attempts to read the contents of the **name.txt** file and returns them as a plain text HTTP response. It includes error handling to manage situations where the file is not found or if any other error occurs during the file reading process.



```python
def download_data(request):
    try:
        file_path = "doc.pdf"
        if os.path.exists(file_path):
            return FileResponse(open(file_path, 'rb'), as_attachment=True)
        else:
            return HttpResponse("File not found", status=404)
    except Exception as e:
        return HttpResponse(str(e), status=500)
```

The **download_data** function serves as a view in the Django project to allow users to download a PDF file named **doc.pdf**. Below is a simplified explanation of its functionality:

File Path:

- Defines the file path variable **file_path** pointing to the location of the PDF file.

File Existence Check:

- Checks if the file exists at the specified path using **os.path.exists()**.

- If the file exists:

  - Returns a **FileResponse** containing the PDF file in binary mode (**rb**), with the **as_attachment** parameter set to `True` to prompt the user to download the file.

- If the file does not exist:

  - Returns an HTTP response with status code 404 (File Not Found) and a message indicating that the file was not found.

- Error Handling:

  - Handles any exceptions that may occur during the process, such as file access errors.

  - Returns an HTTP response with status code 500 (Internal Server Error) along with the error message if any exception occurs.

In summary, the **download_data** function checks for the existence of the PDF file **doc.pdf** and allows users to download it. It includes error handling to manage situations where the file is not found or if any other error occurs during the process.

5.React Installation

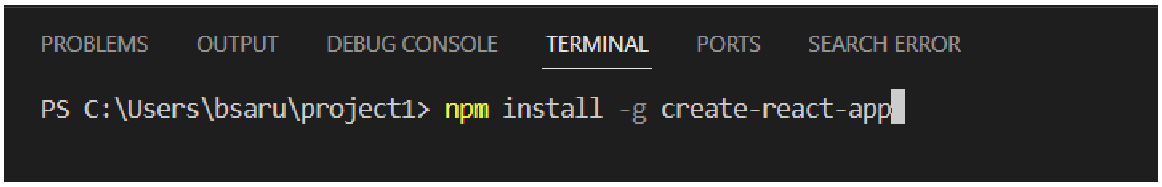
To install React, follow these steps inside the folder which we created for our Django project. React must be inside Django's root directory.

1. **Node.js Installation:** - Ensure you have Node.js installed on your system. You can download and install Node.js from the official website: [Node.js Download](https://nodejs.org/).

2. **Create React App**:

- Open your terminal

- Run the following command to install the Create React App tool globally:



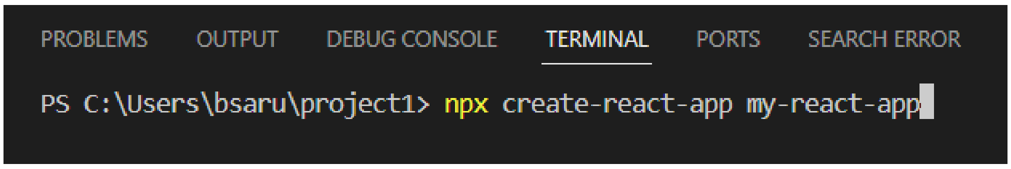
```
PS C:\Users\bsaru\project1> npm install -g create-react-app
```

- This tool simplifies the process of creating and managing React applications.

3. **Create a New React Project**:

- Navigate to the directory where you want to create your React project.

- Run the following command to create a new React project:

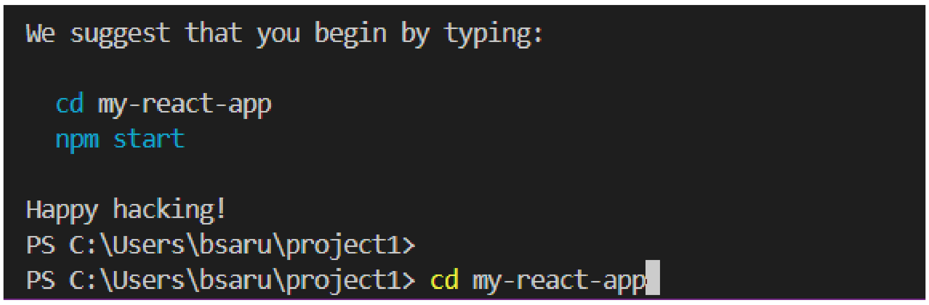


```
PS C:\Users\bsaru\project1> npx create-react-app my-react-app
```

- Replace my-react-app with the name of your project.

4. **Navigate to Project Directory:**

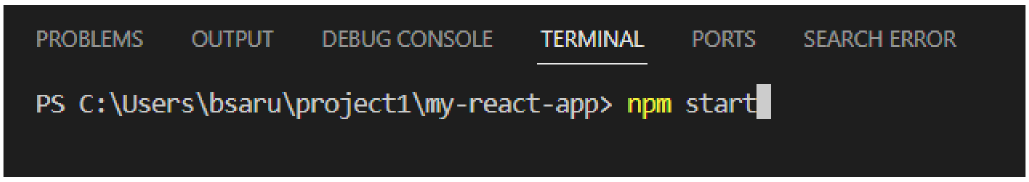
- Change into the project directory:



```
We suggest that you begin by typing:

  cd my-react-app
  npm start

Happy hacking!
PS C:\Users\bsaru\project1>
PS C:\Users\bsaru\project1> cd my-react-app
```

5. **Start Development Server:**

- Once inside the project directory, start the development server by running:

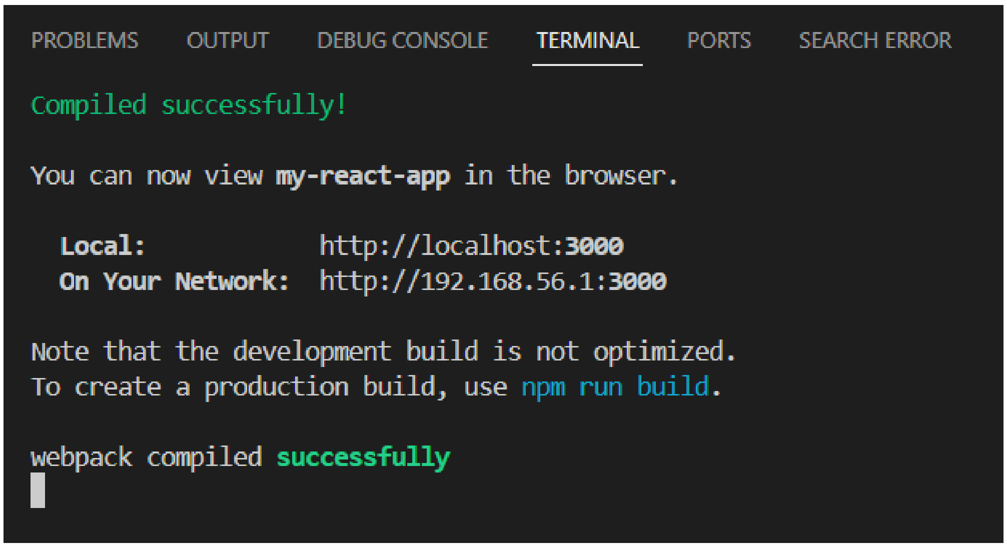


```
PS C:\Users\bsaru\project1\my-react-app> npm start
```

- This command will start the development server and open your default web browser to view the React application.

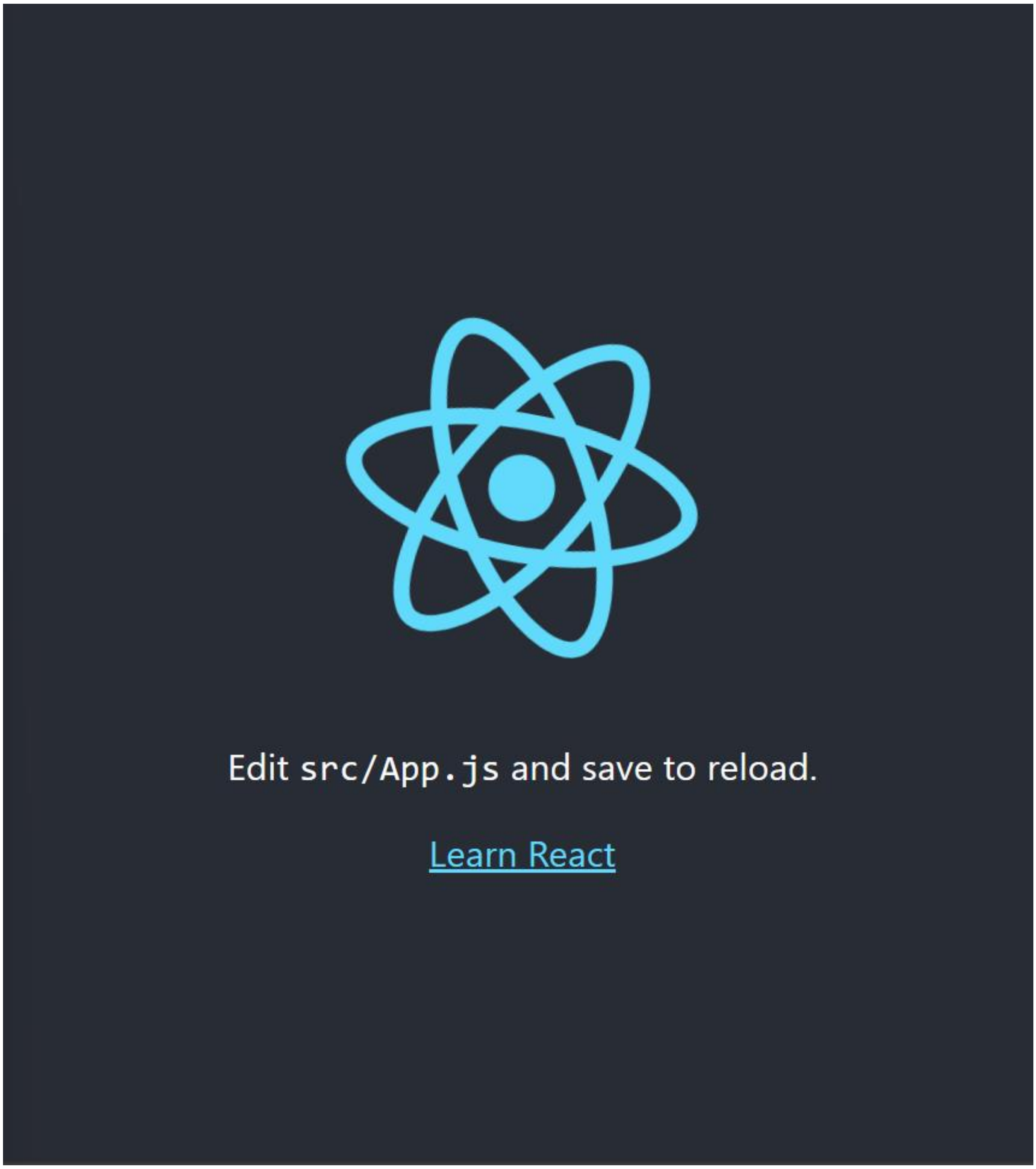
6. **Verify Installation:**

- Open your web browser and navigate to **“http://localhost:3000”** to see the React application running.



```
Compiled successfully!

You can now view my-react-app in the browser.

  Local:            http://localhost:3000
  On Your Network:  http://192.168.56.1:3000

Note that the development build is not optimized.
To create a production build, use npm run build.

webpack compiled successfully
```

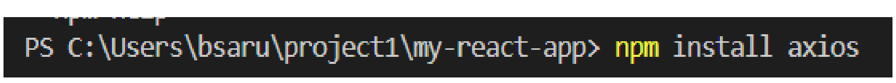
Open your browser and enter the address that your development server shows. You will have a window like this if your installation is a success.



That's it! You've successfully installed React and created a new React project. You can now begin developing your React application.

Additional Package

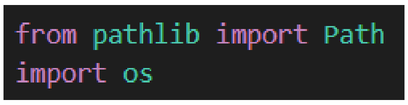
To make HTTP request we need to download “**axios**”



```
PS C:\Users\bsaru\project1\my-react-app> npm install axios
```

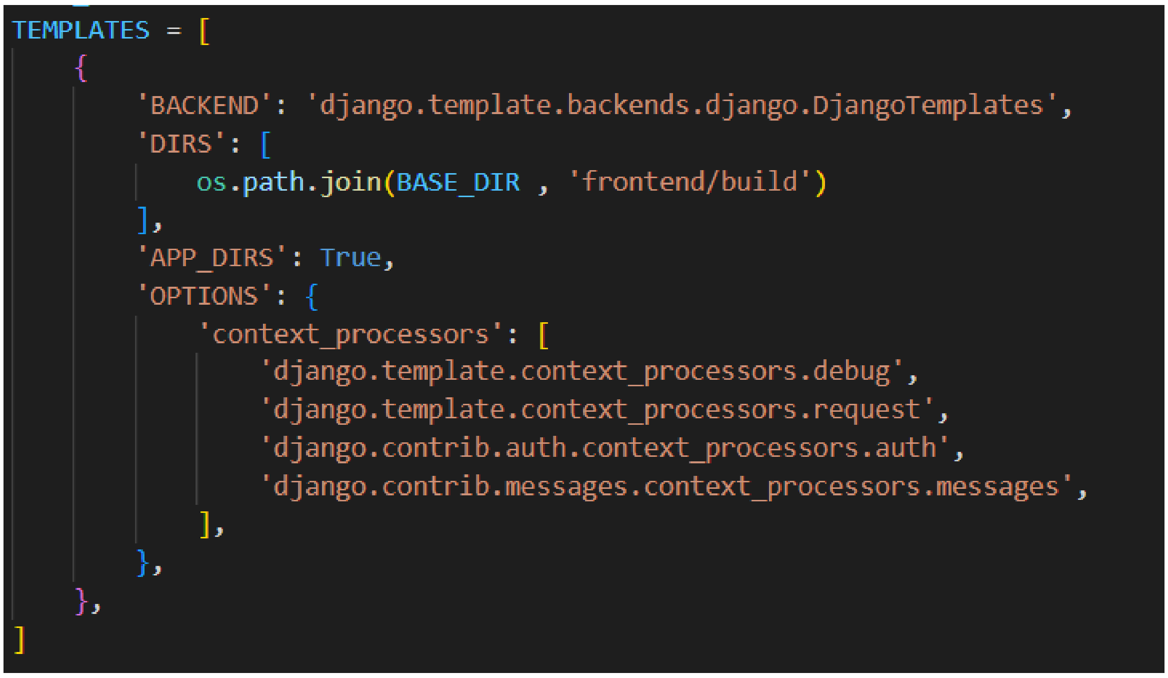
6.Connecting React with Django

-Open <u>**settings.py**</u> add the following code below:

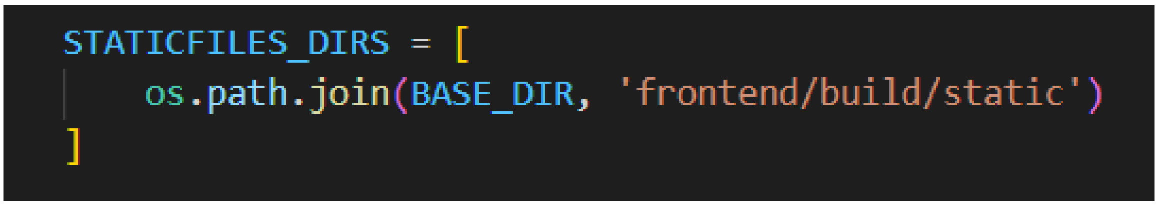


```
from pathlib import Path
import os
```

-Add **import os**



```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [
            os.path.join(BASE_DIR , 'frontend/build')
        ],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',
                'django.contrib.messages.context_processors.messages',
            ],
        },
    },
]
```



```
STATICFILES_DIRS = [
    os.path.join(BASE_DIR, 'frontend/build/static')
]
```

-By adding these code we will successfully connect React to Django. We are rendering the static files from react.

7.Files of React

1. **index.html**:

   - Entry point HTML file where the React application is rendered.

   - Contains a root **<div>** element with an **id** where the React app is mounted, usually **id="root"**.

2. **index.js**:

   - JavaScript file that serves as the entry point for the React application.

   - Renders the root component (`App`) into the DOM using ReactDOM.

3. **App.js**:

   - Main component file where the root component ("App") is defined.

   - Contains the structure of the entire application and renders other components.

4. **App.css:**

   - CSS file for styling the App component.

   - Contains styles specific to the App component and its children.

5. **src/ directory:**

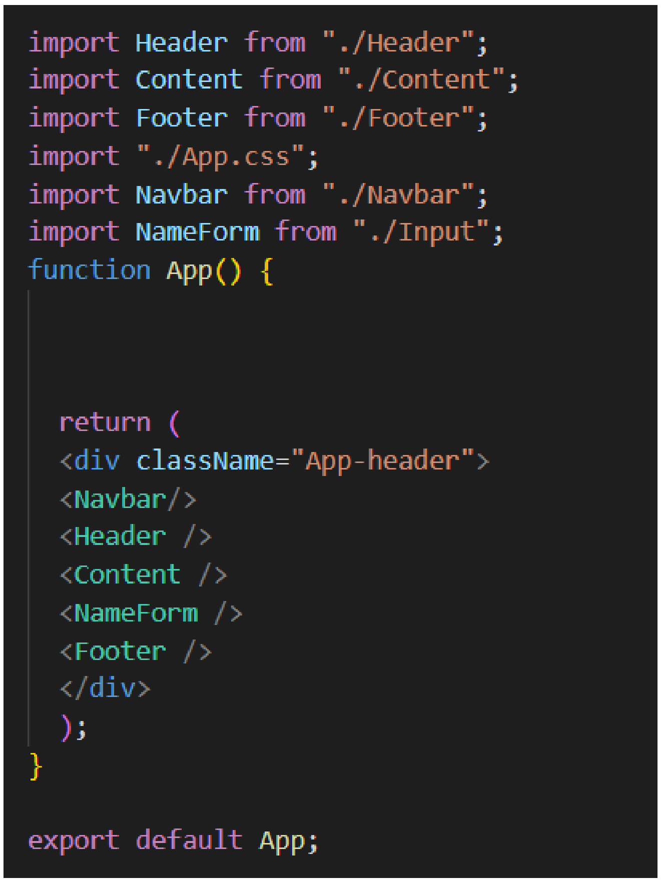
   - Directory where most of the application's source code resides.

- Contains JavaScript files for React components, CSS files for styling, and other assets.

6. **Components**:

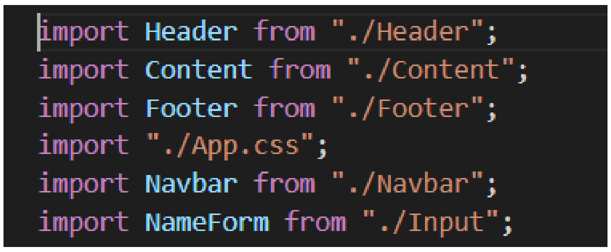
   - Individual component files (e.g., Navbar.js, Header.js, Footer.js).

- Each component typically consists of a JavaScript file defining the component's logic and a CSS file for styling.



```
import Header from "./Header";
import Content from "./Content";
import Footer from "./Footer";
import "./App.css";
import Navbar from "./Navbar";
import NameForm from "./Input";
function App() {



  return (
  <div className="App-header">
  <Navbar/>
  <Header />
  <Content />
  <NameForm />
  <Footer />
  </div>
  );
}

export default App;
```

-This is **App.js** file. This file connects all the UI designs. Each element is made in separate files and connected to the **main.js**

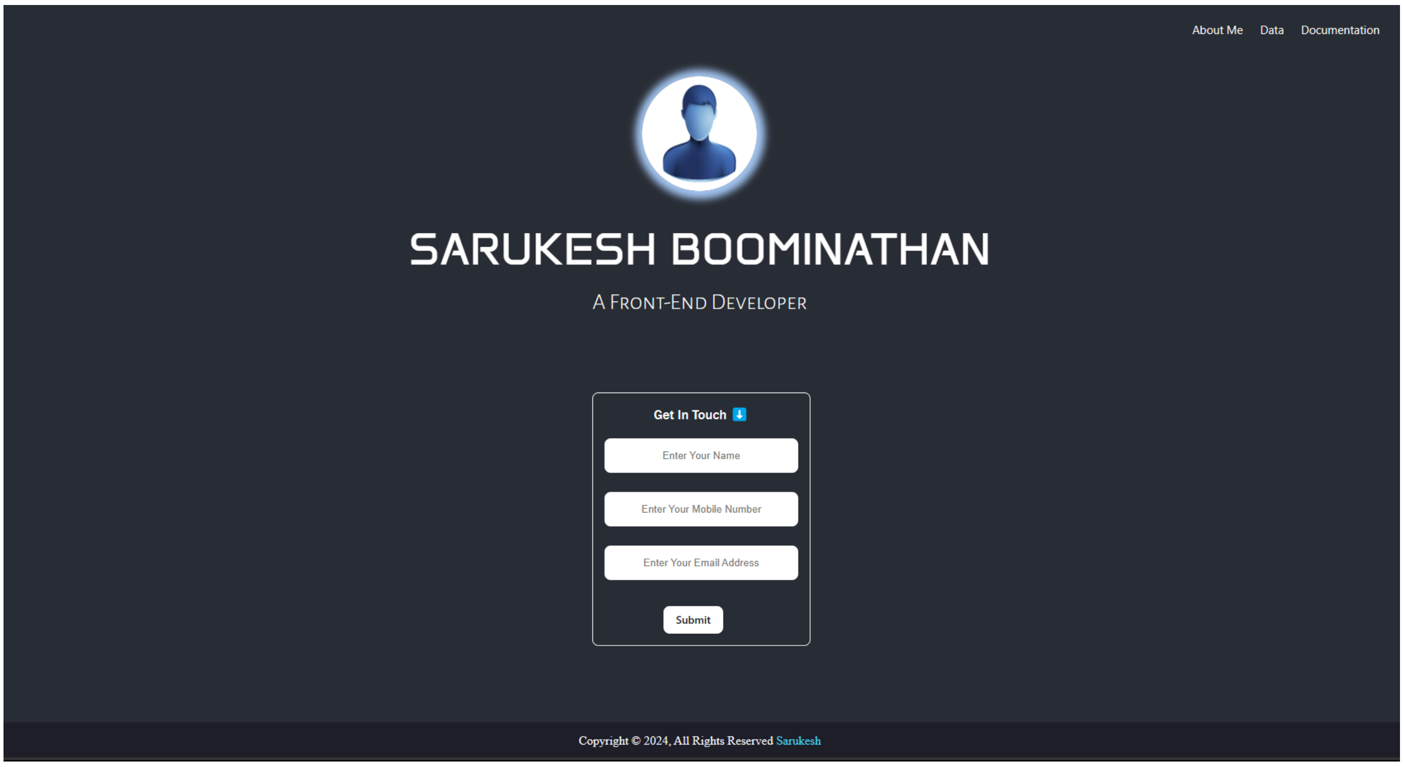


```
import Header from "./Header";
import Content from "./Content";
import Footer from "./Footer";
import "./App.css";
import Navbar from "./Navbar";
import NameForm from "./Input";
```

-This code connects all the components. It imports the code from external file to the **App.js.**

8.Output:

The project is a full-stack web application developed using Django for the backend and React for the frontend. It serves as a platform where users can interact with the application by providing their id, title, and due date. The backend stores this information in a text file and provides endpoints for retrieving the stored data and downloading documentation.



9.Conclusion:

Key Components:

1. Backend (Django):

- Utilizes Django framework to handle HTTP requests, manage data, and serve API endpoints.
- Defines views for handling requests such as storing user information and providing data retrieval functionality.
- Uses Django REST Framework for building RESTful APIs and integrating authentication classes for securing endpoints.
- Stores user information in a text file named `name.txt`.
- Provides endpoints for storing user data, retrieving stored data, and downloading documentation.

2. **Frontend (React):**

- Developed using React library for building user interfaces.
- Consists of components such as Navbar, Header, Content, NameForm (input form), and Footer.
- Uses Axios library to make HTTP requests from the frontend to the backend.

- Allows users to input their name, mobile number, and email address through the NameForm component.
- Integrates with the backend to store user data and retrieve stored information.

Communication Between Frontend and Backend:

- Frontend communicates with the backend by making HTTP requests using Axios library.
- Backend processes incoming requests, performs necessary operations, and sends appropriate responses back to the frontend.
- This communication enables data exchange between the frontend and backend, allowing users to interact with the application seamlessly.

In conclusion, the project demonstrates the integration of Django and React to build a full-stack web application. It showcases the implementation of CRUD (Create, Read, Update, Delete) operations using Django's backend functionalities and React's dynamic user interface. By connecting the frontend and backend, the application enables users to input their information, store it securely, and retrieve it when needed. With its modular architecture and responsive design, the project serves as a robust foundation for further development and expansion of features.

2nd Problem set

Objective:

1. Persistence with File-based Storage
2. Implementing API authentication
3. Advanced Features - Search and Filtering

Abstract:

The task management system is an advanced platform built using Django and DRF, aimed at providing users with a seamless experience in handling tasks. Key features include user authentication through token-based authentication, allowing secure access to task data, and integration with PostgreSQL for robust data storage and retrieval. The system offers RESTful APIs for CRUD operations on tasks, ensuring efficient task management. Additionally, error handling mechanisms and logging functionalities have been enhanced for improved debugging and maintenance. The system also includes documentation retrieval functionality, enabling users to access project documentation easily.

Introduction:

The project represents a significant evolution in the realm of task management systems, leveraging Django and Django REST Framework (DRF) to create a robust platform. Designed with a focus on functionality, security, and scalability, this system offers a comprehensive solution for managing tasks efficiently. Users can perform various operations such as task creation, retrieval, update, and deletion through RESTful APIs. With user authentication mechanisms in place, including token-based authentication, the system ensures secure access to task data. Furthermore, the integration of PostgreSQL as the database backend enhances performance and scalability, making it suitable for handling large volumes of task-related information.

Advancements in the Project:

The project has undergone a comprehensive overhaul with several notable changes aimed at enhancing functionality, security, and maintainability:

1. Migration to Django REST Framework (DRF):

   - Replaced traditional Django views with DRF's view sets and serializers for building RESTful APIs.

   - Utilized DRF's authentication classes for implementing user token authentication, enhancing security.

2. User Authentication Enhancement:

   - Integrated token-based authentication using Django's built-in authentication system and DRF's TokenAuthentication class.

   - Implemented signup and login endpoints to facilitate user registration and authentication.

3. Database Integration with PostgreSQL:

   - Transitioned from SQLite to PostgreSQL for the database backend, offering improved scalability, performance, and reliability.

   - Configured Django settings to connect to PostgreSQL database, ensuring seamless data storage and retrieval.

4. Model Enhancement and Validation:

   - Introduced Task model using Django's ORM, now utilizing PostgreSQL's advanced features such as JSONField for storing task data.

   - Enhanced Task model with validations and constraints using Django model fields and Meta options, ensuring data integrity.

5. Refactored Codebase:

- Restructured codebase for improved modularity, readability, and adherence to Django and DRF conventions.

- Separated views into separate modules and organized project files into logical directories for better organization.

6. Improved Error Handling and Logging:

- Enhanced error handling mechanisms to provide informative responses and log exceptions for debugging purposes.

- Implemented logging using Python's logging module to record errors and application events.

7. Documentation and Documentation Download:

- Included a dedicated function to read data from a file, expanding the system's functionality to handle external data sources.

- Provided endpoints for downloading documentation, enabling users to access project documentation easily.

8. Enhanced Task Management Functionality:

- Implemented CRUD (Create, Read, Update, Delete) operations for tasks using DRF's ModelViewSet and serializer classes.

- Added functionality to search for task details by ID, providing users with efficient access to specific task information.

These changes collectively contribute to a more robust, scalable, and feature-rich task management system, ensuring improved user experiences and easier maintenance of the project in the future. The integration of PostgreSQL further strengthens the system's backend infrastructure, offering enhanced performance and scalability for managing task data effectively.

Important Commands to know:

1. This command is the most often used command. This command starts our Django development server

```
> python manage.py runserver
```

2. To convert the Python code written for the model classes (which further represents tables in the database) into database queries. And it becomes necessary to run this command whenever we make any kind of changes to our database class models.

```
> python manage.py makemigrations
```

3. We need to run this command to create tables in the specified database based on the defined Python class models. This command is responsible for applying or un-applying migrations.

```
> python manage.py migrate
```

4. A Django project is a collection of apps and configurations for a website. A project can have multiple apps inside it and an app can be included in several Django projects.

```
> python manage.py startapp <app_name>
```

5. It is an essential and necessary command to log in to the default admin interface panel provided by the Django framework. This command is

required to create a superuser for the Admin interface who has the username, password, and all other necessary permissions to access and manage the Django website

```
> python manage.py createsuperuser
```

Additional Functionality I learned :

1. “.gitignore” file:

.gitignore tells git which files (or patterns) it should ignore. It's usually used to avoid committing transient files from your working directory that aren't useful to other collaborators, such as compilation products, temporary files IDEs create, etc.

<u>Examples</u>

- log files
- temporary files
- hidden files
- personal files

2. Logger function:

Logger Initialization:

```
logger = logging.getLogger(__name__)
```

In the project, logging plays a crucial role in capturing and recording various events and errors that occur during the execution of the application.

The line logger = logging.getLogger(__name__) is a critical initialization step that sets up a logger object specific to the module where it is placed.

The initialization of the logger using

logger = logging.getLogger(__name__) is a crucial step in setting up a robust logging infrastructure within the project, enabling effective monitoring and debugging of the application's behavior.

Creating an App in Django:

I'm address a common mistake in Django project organization and demonstrate the correct approach by creating a new Django app. Initially, views were placed directly within the project directory, which is not considered a best practice in Django development. To rectify this, we'll create a new Django app and migrate the views into it, ensuring better organization, modularity, and scalability of the project.

Issue with having views.py in project directory:

Placing views directly within the project directory violates the principle of modularity and separation of concerns in software development. This approach can lead to cluttered project structure, difficulty in maintaining codebase, and reduced scalability as the project grows. By moving views to a dedicated Django app, we adhere to Django's design philosophy, promoting better code organization and maintainability.

Create a New App:

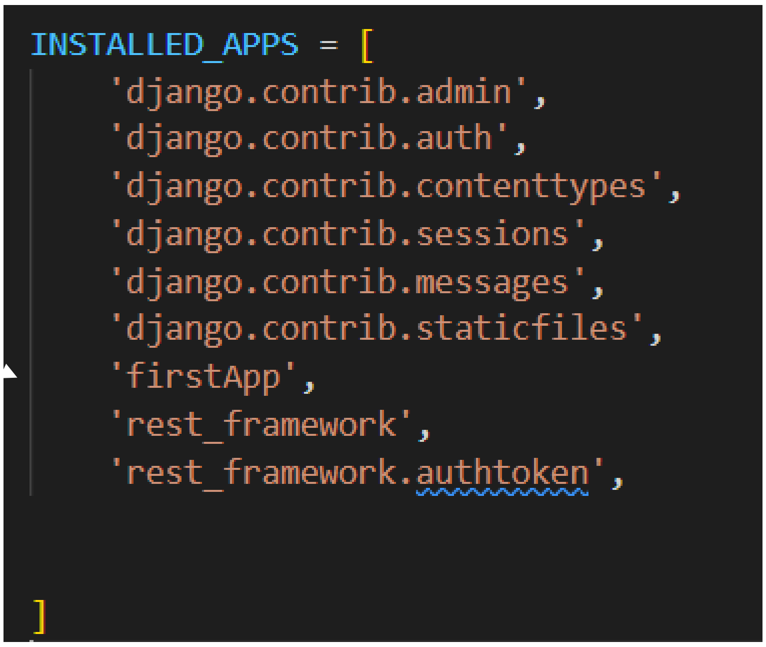
Use the following command to create a new Django app. Replace firstApp with the desired name for your app.

```
> python manage.py startapp <app_name>
```

This command will generate a new directory named firstApp containing the necessary files and folders for the Django app.

Register the App:

After creating the app, you need to register it with the Django project. Open the settings.py file located in your project directory, and add 'firstApp' to the INSTALLED_APPS list:



```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'firstApp',
    'rest_framework',
    'rest_framework.authtoken',


]
```

Moving Views to the App:

Copy Views File:

Locate the views.py file in your project directory where your views are currently defined. Copy this file.

Paste Views File into App:

Navigate to the firstApp directory that was created earlier. Paste the views.py file into this directory.

Remove Views from Project Directory:

Once you've verified that the views have been successfully moved to the app directory and all imports are updated, you can remove the original views.py file from the project directory.

Files inside my new app:

admin.py:

- This file is used to register models with the Django admin interface.

- You can customize how models are displayed and managed in the admin interface by defining corresponding admin classes.

apps.py:

- This file defines the configuration class for the app.
- It can be used to configure various aspects of the app, such as the app name and any app-specific settings.

models.py:

- This file is where you define your Django models, which represent the data structure of your app.
- Models are typically Python classes that inherit from `django.db.models.Model` and define fields and methods for interacting with the data.

test.py

- This file is used for writing test cases for your app.
- You can define test functions or classes to ensure that your app behaves as expected under various scenarios.

Views.py

- This file contains the views (or controller functions) of your app.
- Views are responsible for processing incoming requests, performing any necessary business logic, and returning HTTP responses.

Urls.py

- This file defines the URL patterns for your app.

- It maps URLs to corresponding views, allowing Django to route incoming requests to the appropriate view functions.

migrations/ :

- This directory contains database migration files generated by Django's migration system.
- Migrations are used to propagate changes to your models (such as adding or modifying fields) to your database schema.

tests/:

- This directory is used for storing test files for the app.
- Test files contain test cases written using Django's testing framework to ensure that the app behaves correctly under different conditions.

Custom Files:

templates/:

- This directory is used to store HTML templates used by the app's views.
- Templates are used to generate dynamic HTML content, which is then returned as part of the HTTP response by the views.

| In my project this folder is not necessary because I used React jS as my frontend. So, I am rendering my static files directly from React **|**

Serializers.py:

The serializers.py file is used to define serializers, which are classes responsible for converting complex data types (such as Django models) into native Python data types suitable for rendering into JSON, XML, or other content types.

Using Databases in Django:

When developing web applications with Django, choosing the appropriate database management system (DBMS) is crucial. Let's compare the use of databases in Django and then highlight the main advantages of PostgreSQL and SQLite3:

Comparing PostgreSQL and SQLite3:

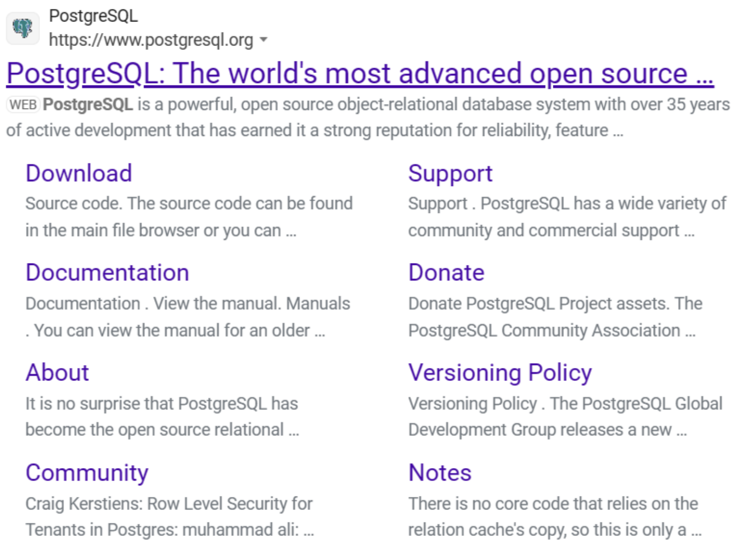
PostgreSQL:

Scalability and Performance: PostgreSQL is well-suited for large-scale applications and environments requiring high concurrency and robust performance. It offers advanced optimization features, support for complex queries, and transactional integrity, making it ideal for handling heavy workloads.

-Advanced Features: PostgreSQL provides advanced features such as JSONB data type for storing and querying JSON data, support for full-text search, advanced indexing options, and extensibility through user-defined functions and custom data types.

Setting up my Database:

Open your browser and search for “PostgresSql” click the correct link as shown below.



PostgreSQL
https://www.postgresql.org

PostgreSQL: The world's most advanced open source ...

WEB **PostgreSQL** is a powerful, open source object-relational database system with over 35 years of active development that has earned it a strong reputation for reliability, feature ...

Download
Source code. The source code can be found in the main file browser or you can ...

Support
Support . PostgreSQL has a wide variety of community and commercial support ...

Documentation
Documentation . View the manual. Manuals . You can view the manual for an older ...

Donate
Donate PostgreSQL Project assets. The PostgreSQL Community Association ...

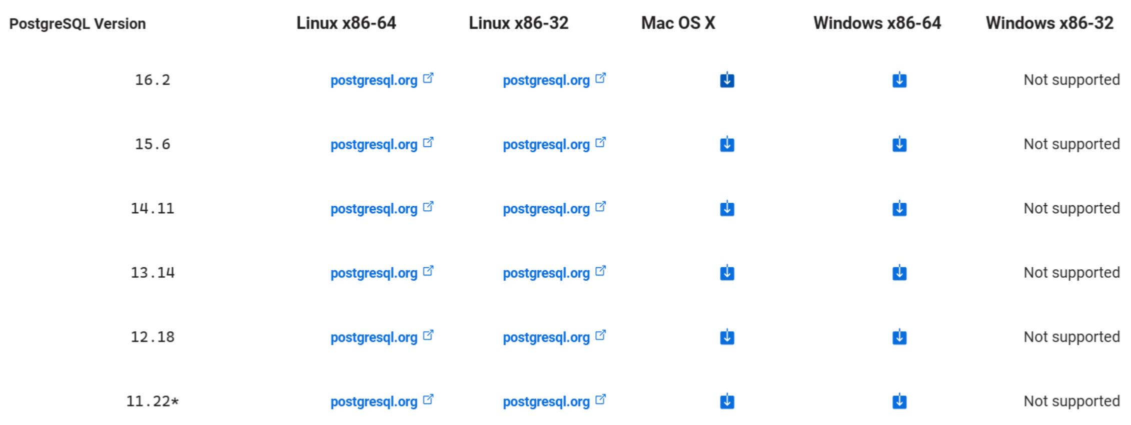
About
It is no surprise that PostgreSQL has become the open source relational ...

Versioning Policy
Versioning Policy . The PostgreSQL Global Development Group releases a new ...

Community
Craig Kerstiens: Row Level Security for Tenants in Postgres: muhammad ali: ...

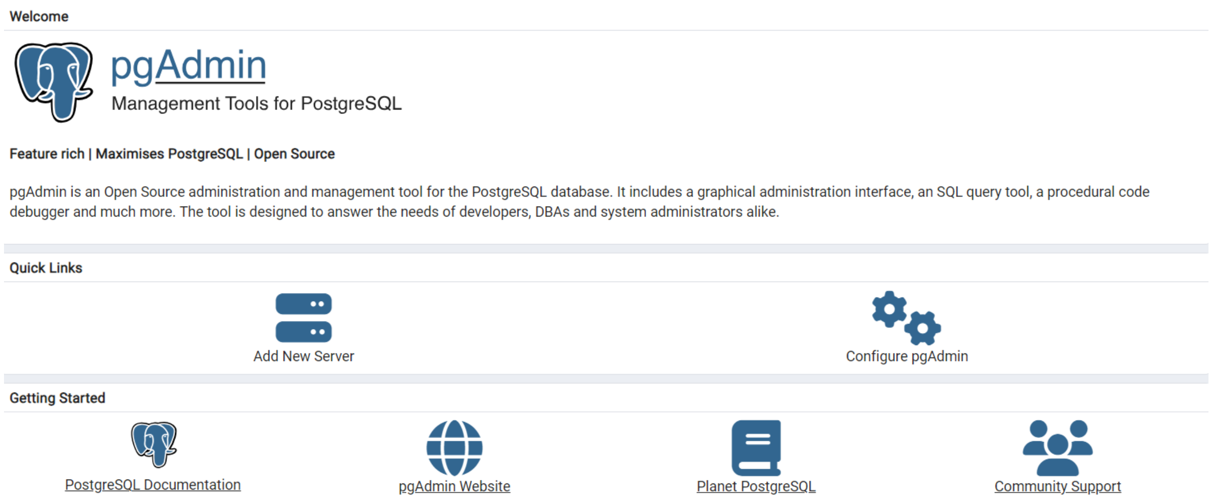
Notes
There is no core code that relies on the relation cache's copy, so this is only a ...

Now, navigate to the download page and download the latest version and choose your appropriate operating system



PostgreSQL Version	Linux x86-64	Linux x86-32	Mac OS X	Windows x86-64	Windows x86-32
16.2	postgresql.org	postgresql.org	⬇	⬇	Not supported
15.6	postgresql.org	postgresql.org	⬇	⬇	Not supported
14.11	postgresql.org	postgresql.org	⬇	⬇	Not supported
13.14	postgresql.org	postgresql.org	⬇	⬇	Not supported
12.18	postgresql.org	postgresql.org	⬇	⬇	Not supported
11.22*	postgresql.org	postgresql.org	⬇	⬇	Not supported

Open the software and login with your credentials which you gave while setup process



Welcome

pgAdmin

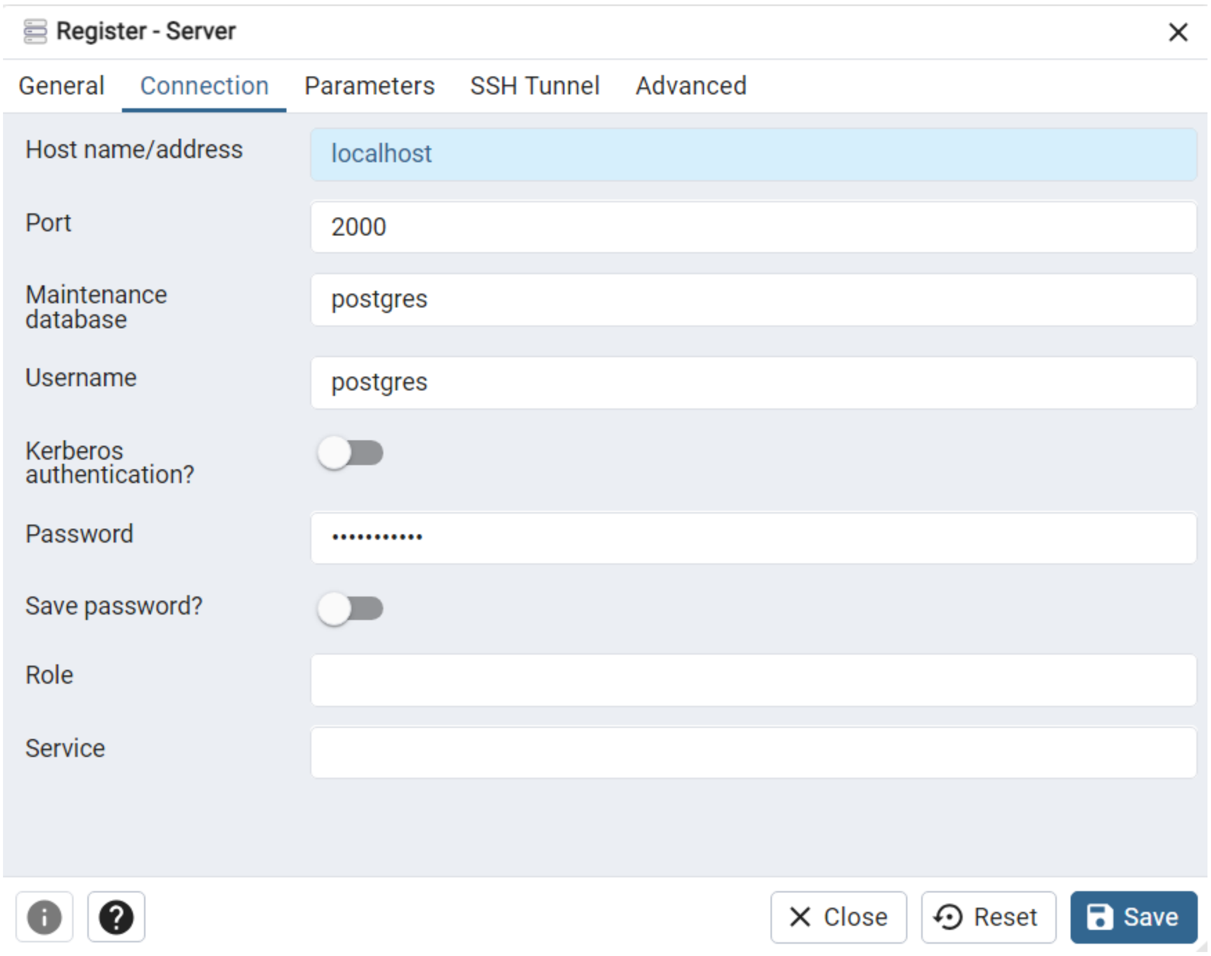
Management Tools for PostgreSQL

Feature rich | Maximises PostgreSQL | Open Source

pgAdmin is an Open Source administration and management tool for the PostgreSQL database. It includes a graphical administration interface, an SQL query tool, a procedural code debugger and much more. The tool is designed to answer the needs of developers, DBAs and system administrators alike.

This the page you will get after you login.

Click on “Add new server”


Register - Server

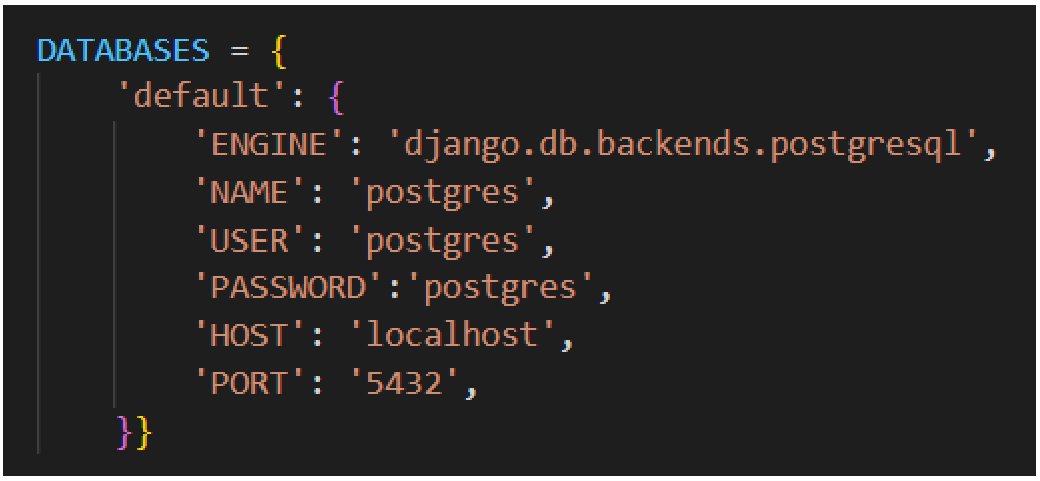
General | Connection | Parameters | SSH Tunnel | Advanced

Host name/address	localhost
Port	2000
Maintenance database	postgres
Username	postgres
Kerberos authentication?	○
Password	•••••••••••
Save password?	○
Role	
Service	

Hostname/address should a valid address. You can give "localhost".

Port number is an important parameter which should be given carefully. Because each application has its unique port number. It is better to use the default port address "5432"

Once your application is downloaded and all basic setup is done, its time to connect your database to our project like shown below.


```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'postgres',
        'USER': 'postgres',
        'PASSWORD':'postgres',
        'HOST': 'localhost',
        'PORT': '5432',
    }}
```

Now database has be successfully connected to your project. Its time to create tables inside your database.

```
> python manage.py makemigrations
```

Run this command to create tables in your database. This is convert your model class in query and create tables into your database.

Now run this code to apply migrations.

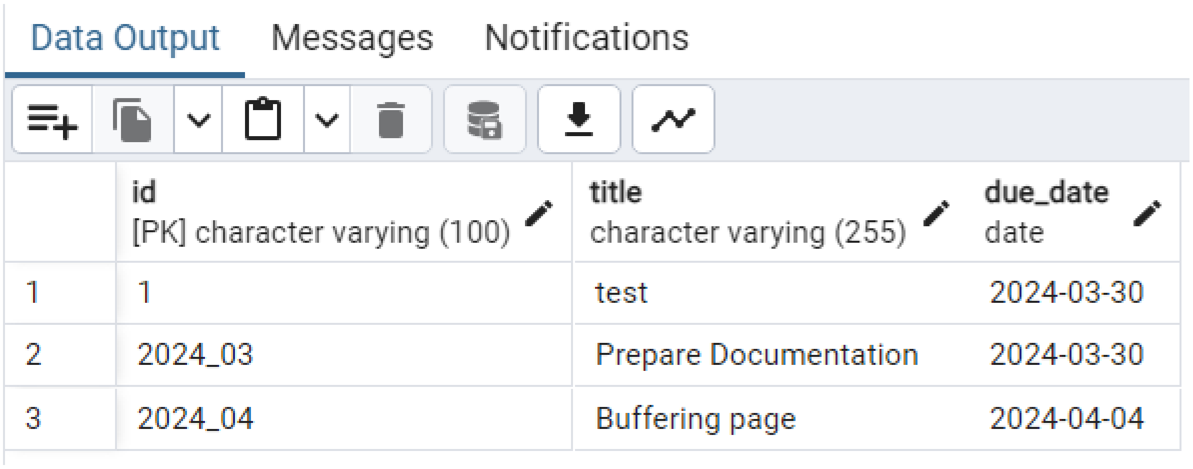
```
> python manage.py migrate
```

Now you have created your tables inside your database and applied all changes. Whenever you create a new table run **"makemigratations"** command. And to apply changes to your table run **"migrate"** command.

To see your table in postgres open the software and login using your credentials for your database and do as follow:

server>databases>(your database name)>schemas>tables

right click on tables and click view to view you data inside tables



	id [PK] character varying (100)	title character varying (255)	due_date date
1	1	test	2024-03-30
2	2024_03	Prepare Documentation	2024-03-30
3	2024_04	Buffering page	2024-04-04

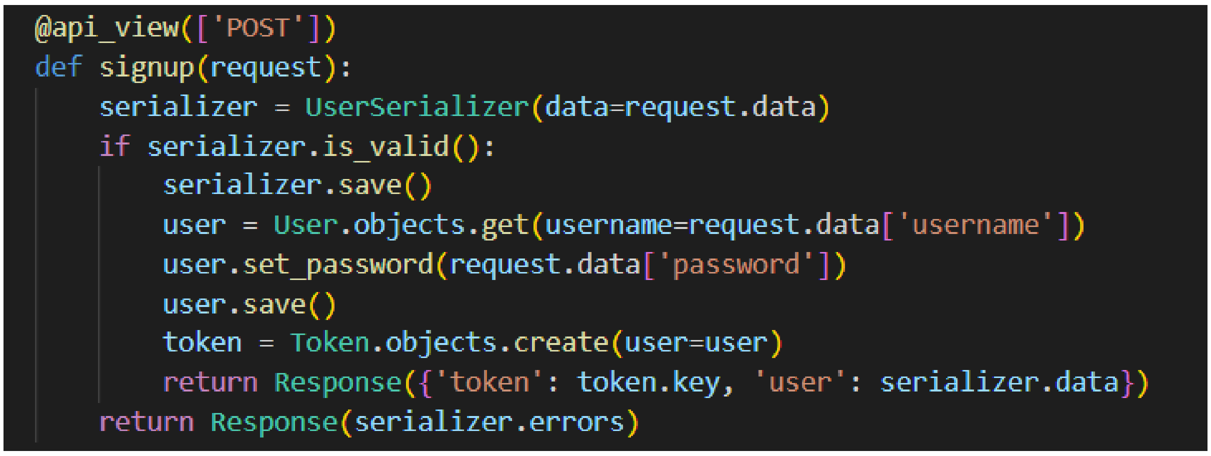
I have done all the basic configuration and setup my app.

Backend Code Explanation:

Views.py:

The views.py file manages various functions within your Django application. It handles tasks like displaying the homepage (index(request)), managing user sign-up (signup(request)) and login (login(request)), storing task data received through POST requests (store_name(request)), deleting task data based on provided IDs (delete_data(request)), searching for task details (search_task_details(request)), retrieving all task data (get_data(request)), and downloading files (download_data(request)). Additionally, it includes a function (read_data_from_file()) to read data from a file named `'name.txt'`. Each function serves a specific purpose in handling different aspects of the application's functionality.

signup(request):



```
@api_view(['POST'])
def signup(request):
    serializer = UserSerializer(data=request.data)
    if serializer.is_valid():
        serializer.save()
        user = User.objects.get(username=request.data['username'])
        user.set_password(request.data['password'])
        user.save()
        token = Token.objects.create(user=user)
        return Response({'token': token.key, 'user': serializer.data})
    return Response(serializer.errors)
```

- This function is a view for handling user signup requests. It's decorated with `@api_view(['POST'])`, indicating that it only accepts POST requests.

- Upon receiving a POST request, it instantiates a `UserSerializer` object with the data from the request body (`request.data`).

- It checks if the serializer is valid using `serializer.is_valid()`. If valid:

  - It saves the user data using `serializer.save()`.

- It retrieves the user object from the database based on the provided username and sets the password for the user obtained from the request data.

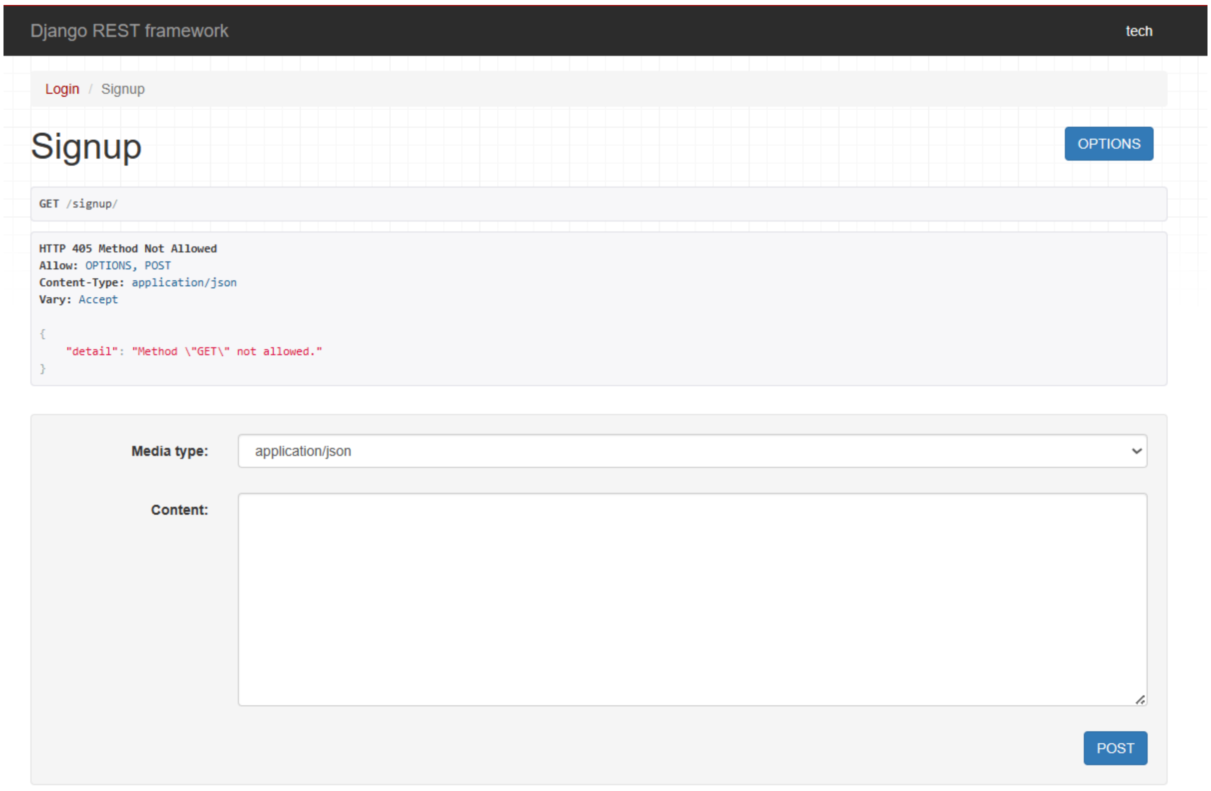
- It saves the updated user object.

- It creates an authentication token for the user using `Token.objects.create(user=user)`.

- It returns a response with a JSON object containing the authentication token (`token.key`) and the serialized user data (`serializer.data`).

- If the serializer is not valid, it returns a response containing the validation errors.

UI:

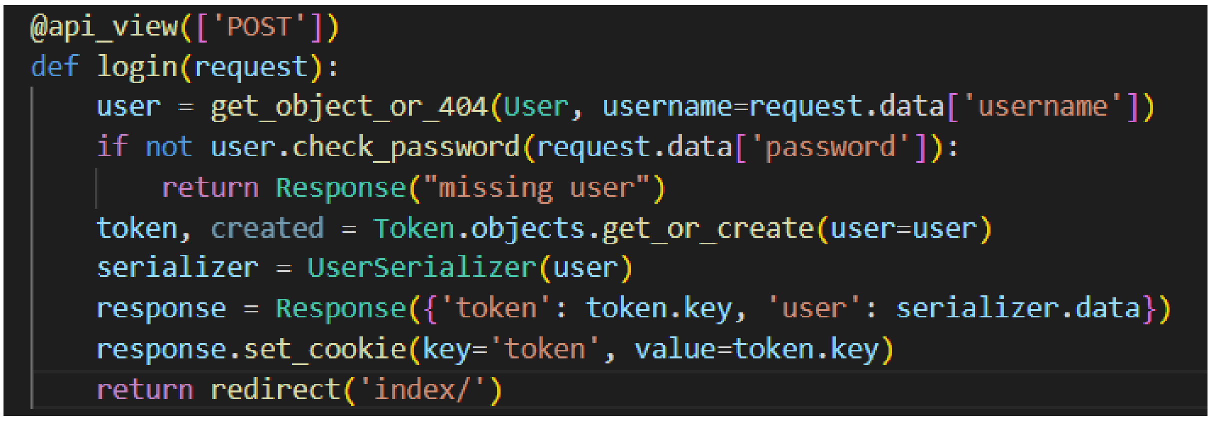


```
GET /signup/
```

```
HTTP 405 Method Not Allowed
Allow: OPTIONS, POST
Content-Type: application/json
Vary: Accept

{
    "detail": "Method \"GET\" not allowed."
}
```

This user interface is a default page which is rendered by Django rest API

login(request)



```
@api_view(['POST'])
def login(request):
    user = get_object_or_404(User, username=request.data['username'])
    if not user.check_password(request.data['password']):
        return Response("missing user")
    token, created = Token.objects.get_or_create(user=user)
    serializer = UserSerializer(user)
    response = Response({'token': token.key, 'user': serializer.data})
    response.set_cookie(key='token', value=token.key)
    return redirect('index/')
```

- This function is a view for handling user login requests. Like `signup(request)`, it's decorated with `@api_view(['POST'])`, indicating that it only accepts POST requests.

- Upon receiving a POST request, it attempts to retrieve the user object from the database based on the provided username using `get_object_or_404(User, username=request.data['username'])`.

- If the user object is found, it checks if the provided password matches the user's password using `user.check_password(request.data['password'])`. If not, it returns a response indicating a failed login attempt.

- If the password is correct or the user is found, it creates an authentication token for the user using `Token.objects.get_or_create(user=user)`.

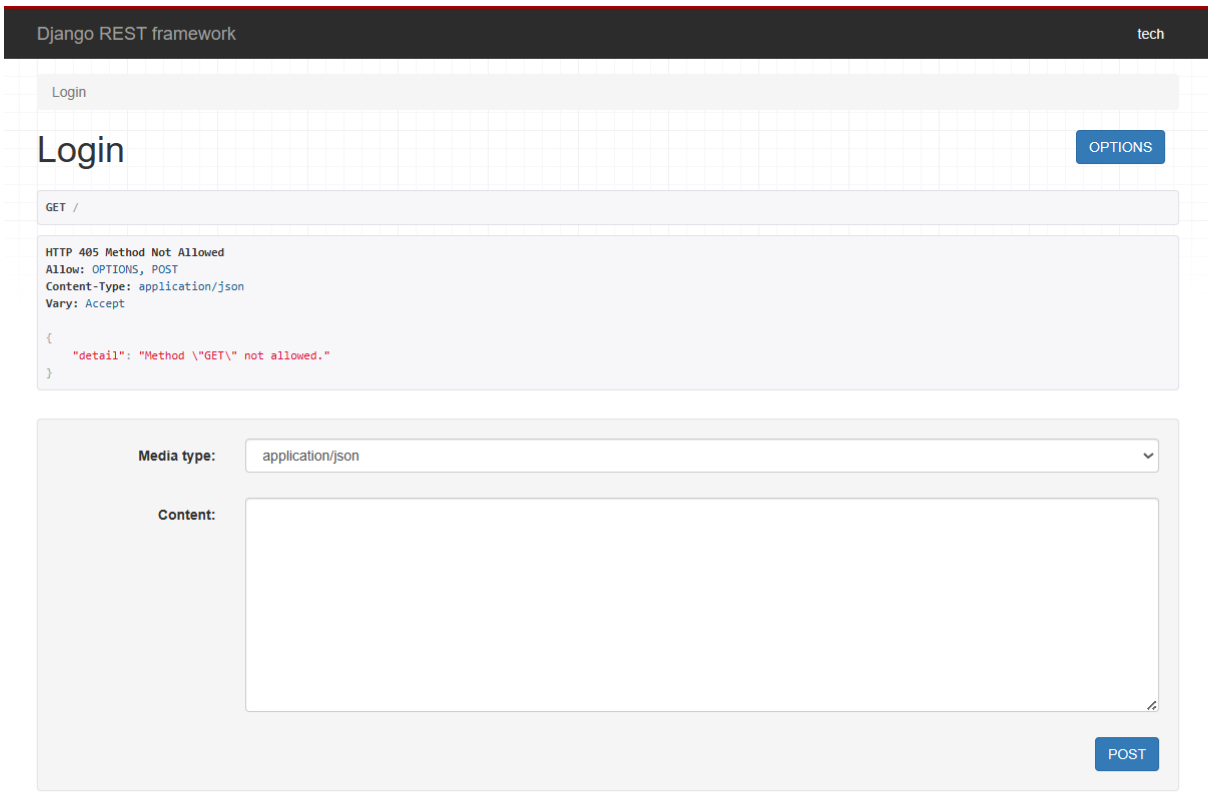
- It instantiates a `UserSerializer` object with the user data.

- It creates a response containing the authentication token (`token.key`) and the serialized user data (`serializer.data`).

- Additionally, it sets a cookie named `'token'` with the value of the authentication token.

- Finally, it redirects the user to the homepage using redirect('index/').

UI:



Login

```
GET /
```

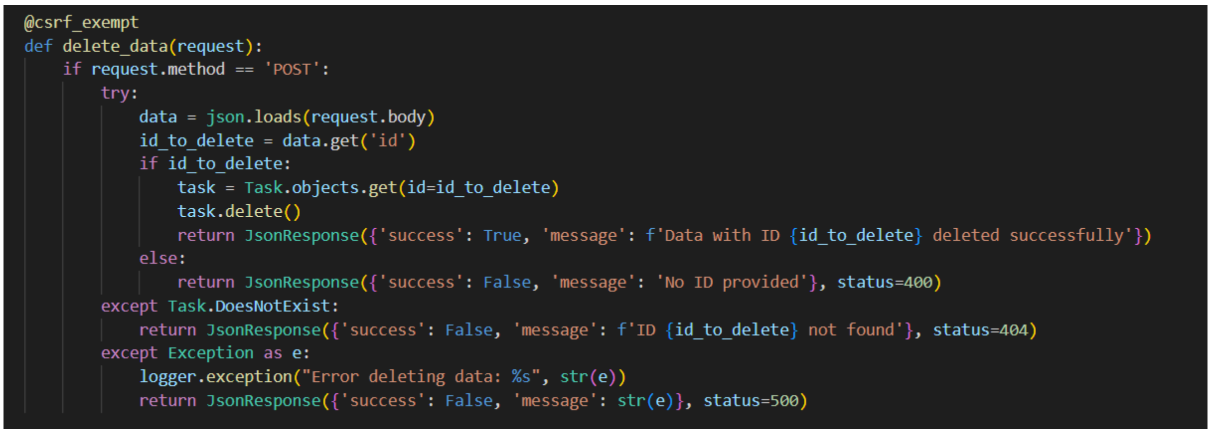
```
HTTP 405 Method Not Allowed
Allow: OPTIONS, POST
Content-Type: application/json
Vary: Accept

{
    "detail": "Method \"GET\" not allowed."
}
```

Media type: application/json

Content:

Enter created credentials in **Json format**

delete_data(request):



```
@csrf_exempt
def delete_data(request):
    if request.method == 'POST':
        try:
            data = json.loads(request.body)
            id_to_delete = data.get('id')
            if id_to_delete:
                task = Task.objects.get(id=id_to_delete)
                task.delete()
                return JsonResponse({'success': True, 'message': f'Data with ID {id_to_delete} deleted successfully'})
            else:
                return JsonResponse({'success': False, 'message': 'No ID provided'}, status=400)
        except Task.DoesNotExist:
            return JsonResponse({'success': False, 'message': f'ID {id_to_delete} not found'}, status=404)
        except Exception as e:
            logger.exception("Error deleting data: %s", str(e))
            return JsonResponse({'success': False, 'message': str(e)}, status=500)
```

Function Logic:

- The function checks if the incoming request method is POST.

- It attempts to load JSON data from the request body using `json.loads(request.body)`.

- It extracts the ID to delete from the JSON data (`id_to_delete = data.get('id')`).

- If an ID is provided (`id_to_delete` is not None), it attempts to retrieve the corresponding task object from the database using `Task.objects.get(id=id_to_delete)`.

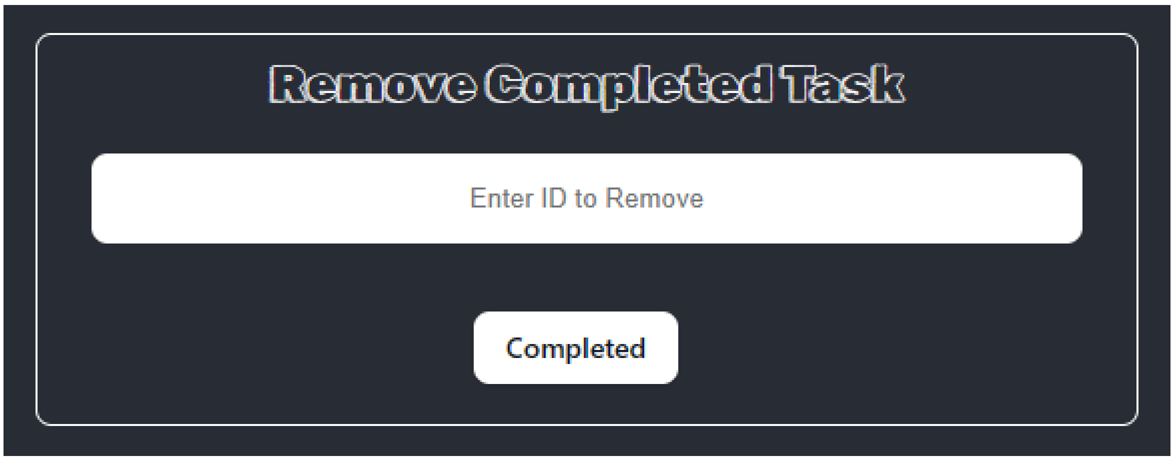
  - If the task exists, it deletes the task using `task.delete()` and returns a JSON response indicating success, along with a message confirming the deletion.

  - If no ID is provided, it returns a JSON response indicating failure and a message stating that no ID was provided.

  - If the task with the provided ID does not exist, it returns a JSON response indicating failure and a message stating that the ID was not found.

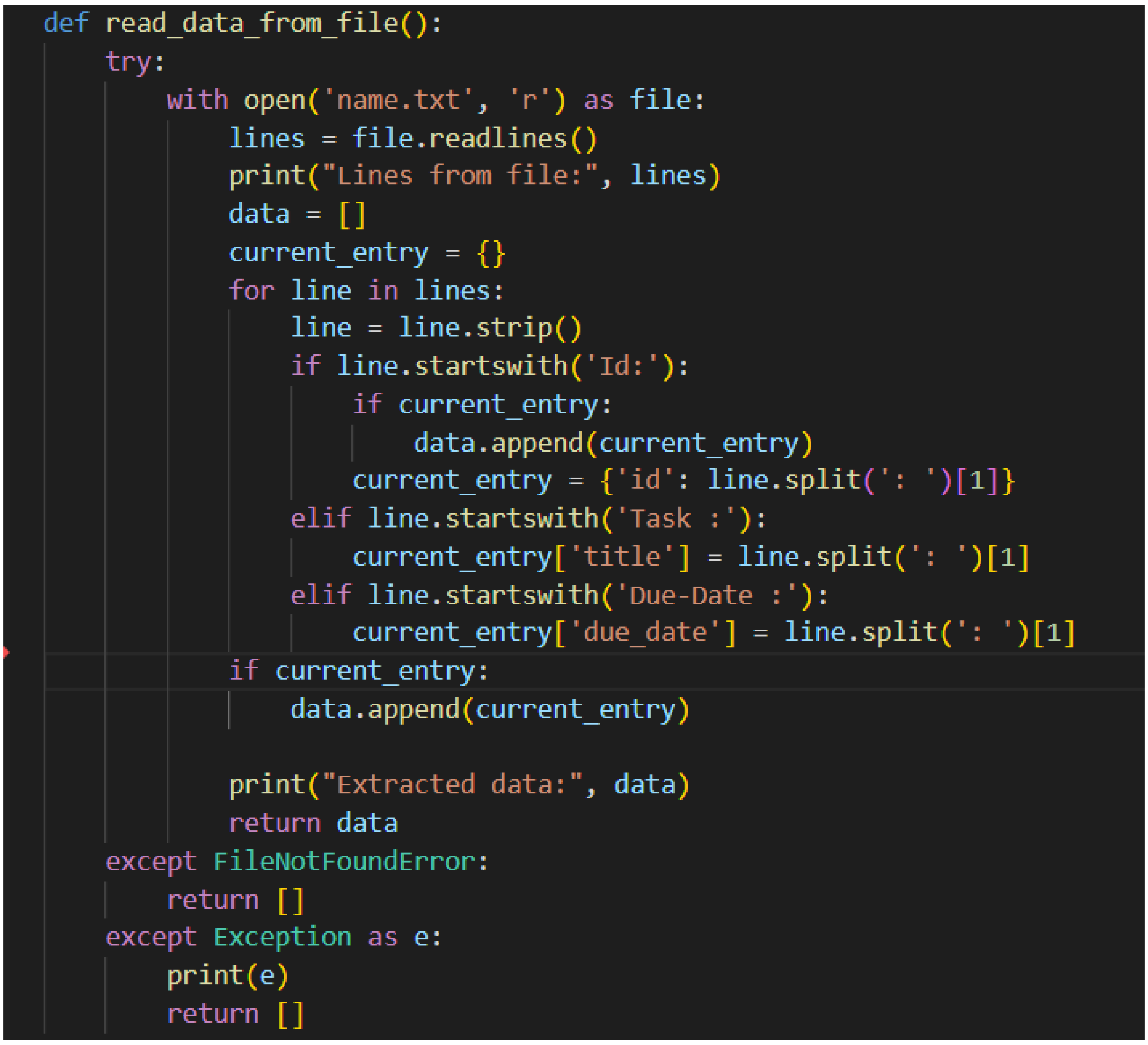
  - If any other exception occurs during the process, it logs the error using `logger.exception()` and returns a JSON response indicating failure along with the error message.

UI:



This function serves as an endpoint for deleting data, specifically tasks, from the database. It ensures that the request method is POST and handles various scenarios such as invalid or missing IDs and database errors, providing appropriate responses accordingly.

read_data_from_file():



```python
def read_data_from_file():
    try:
        with open('name.txt', 'r') as file:
            lines = file.readlines()
            print("Lines from file:", lines)
            data = []
            current_entry = {}
            for line in lines:
                line = line.strip()
                if line.startswith('Id:'):
                    if current_entry:
                        data.append(current_entry)
                    current_entry = {'id': line.split(': ')[1]}
                elif line.startswith('Task :'):
                    current_entry['title'] = line.split(': ')[1]
                elif line.startswith('Due-Date :'):
                    current_entry['due_date'] = line.split(': ')[1]
            if current_entry:
                data.append(current_entry)

            print("Extracted data:", data)
            return data
    except FileNotFoundError:
        return []
    except Exception as e:
        print(e)
        return []
```

1. File Reading:

- The function attempts to open a file named `'name.txt'` in read mode using open('name.txt', 'r').

- It reads all lines from the file using `file.readlines()` and stores them in the `lines` variable.

2. Data Extraction:

- It initializes an empty list `data` to store extracted data and an empty dictionary `current_entry` to represent each entry in the file.

- It iterates through each line in the `lines`.

- For each line, it removes leading and trailing whitespace using `line.strip()`.

- If a line starts with `'Id:'`, it extracts the ID from the line and initializes a new `current_entry` dictionary with the ID.

- If a line starts with `'Task :'`, it extracts the task title from the line and adds it to the `current_entry` dictionary.

- If a line starts with `'Due-Date :'`, it extracts the due date from the line and adds it to the `current_entry` dictionary.

3. Data Storage:

- After processing each line, if `current_entry` is not empty (indicating that at least one entry has been processed), it appends `current_entry` to the `data` list.

- Finally, if there is any remaining `current_entry`, it appends it to the `data` list.

4. Error Handling:

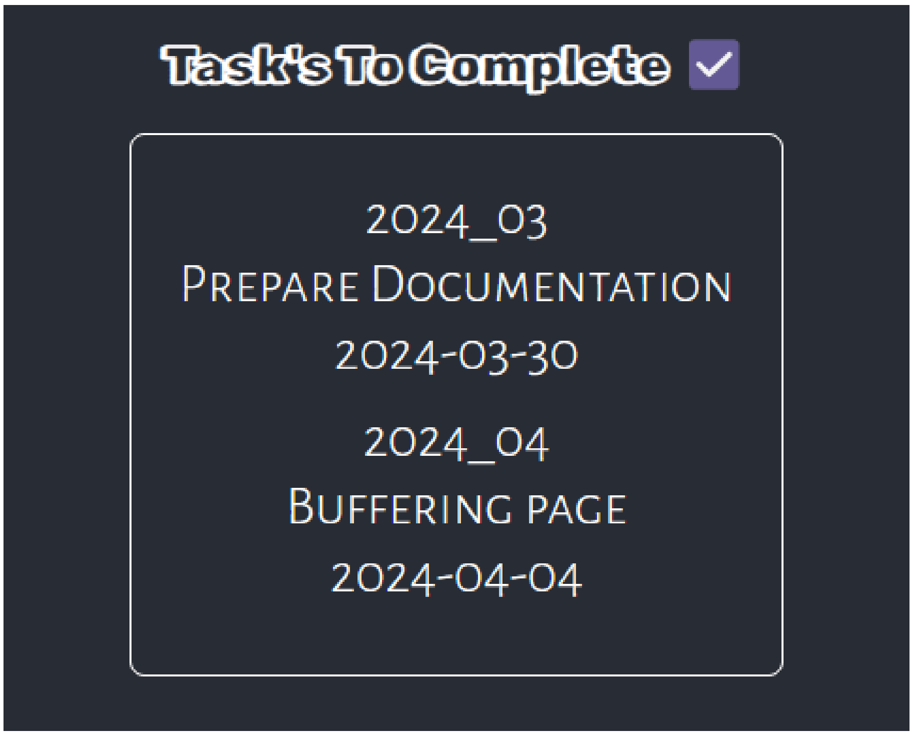
- If the file `'name.txt'` is not found, it returns an empty list `[]`.

- If any other exception occurs during the process, it prints the error using `print(e)` and returns an empty list `[]`.

5. Return:

- It prints the extracted data and returns it as a list of dictionaries representing each entry in the file.

UI:



Task's To Complete ✓

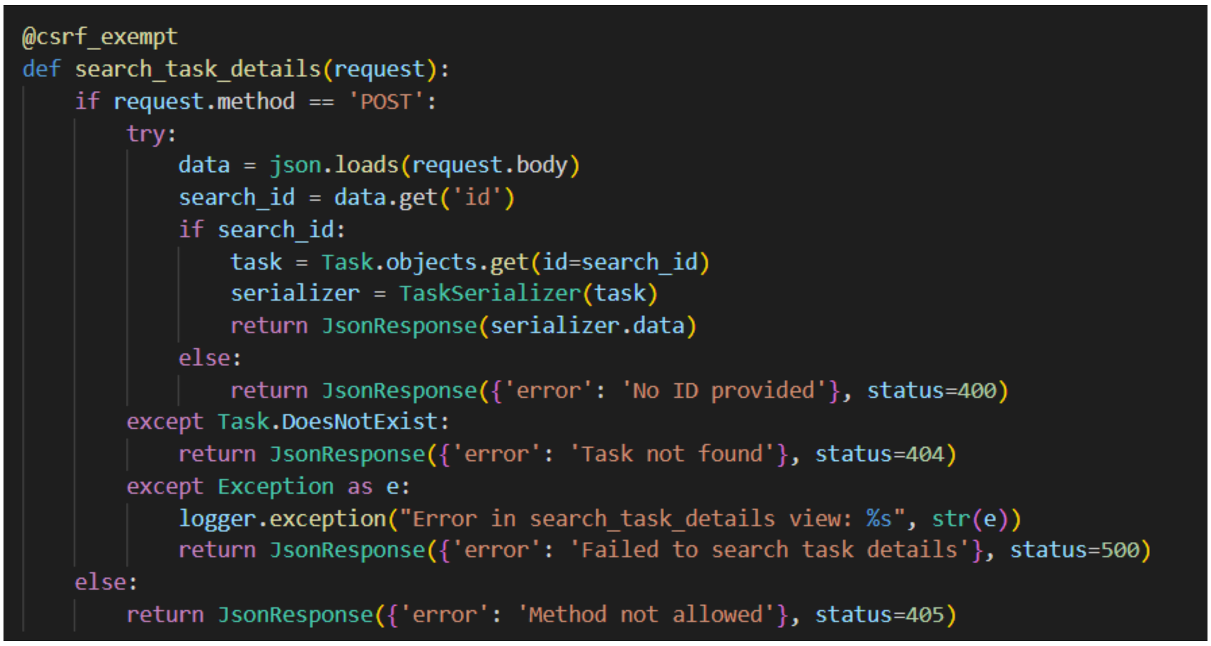
2024_03
PREPARE DOCUMENTATION
2024-03-30

2024_04
BUFFERING PAGE
2024-04-04

This function reads data from a file named name.txt, extracts information formatted with specific tags

(`'Id:'`, `'Task :'`, `'Due-Date :'`), and returns it as a list of dictionaries. It handles file not found and other exceptions gracefully by returning an empty list in case of errors.

search_task_details(request):



```
@csrf_exempt
def search_task_details(request):
    if request.method == 'POST':
        try:
            data = json.loads(request.body)
            search_id = data.get('id')
            if search_id:
                task = Task.objects.get(id=search_id)
                serializer = TaskSerializer(task)
                return JsonResponse(serializer.data)
            else:
                return JsonResponse({'error': 'No ID provided'}, status=400)
        except Task.DoesNotExist:
            return JsonResponse({'error': 'Task not found'}, status=404)
        except Exception as e:
            logger.exception("Error in search_task_details view: %s", str(e))
            return JsonResponse({'error': 'Failed to search task details'}, status=500)
    else:
        return JsonResponse({'error': 'Method not allowed'}, status=405)
```

2. Function Logic:

  - The function first checks if the incoming request method is POST.

  - It attempts to load JSON data from the request body using `json.loads(request.body)`.

  - It retrieves the ID to search for from the JSON data (`search_id = data.get('id')`).

  - If a search ID is provided (`search_id` is not None), it attempts to retrieve the corresponding task object from the database using `Task.objects.get(id=search_id)`.

  - If the task is found, it serializes the task object using `TaskSerializer` to convert it into JSON format.

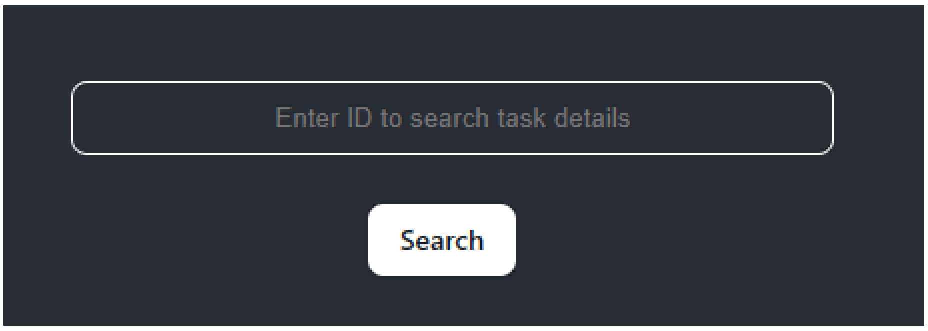
  - It returns a JSON response containing the serialized task data.

- If no search ID is provided, it returns a JSON response indicating an error with the message 'No ID provided' and a status code of 400 (Bad Request).

- If the task with the provided ID does not exist in the database, it returns a JSON response indicating an error with the message 'Task not found' and a status code of 404 (Not Found).

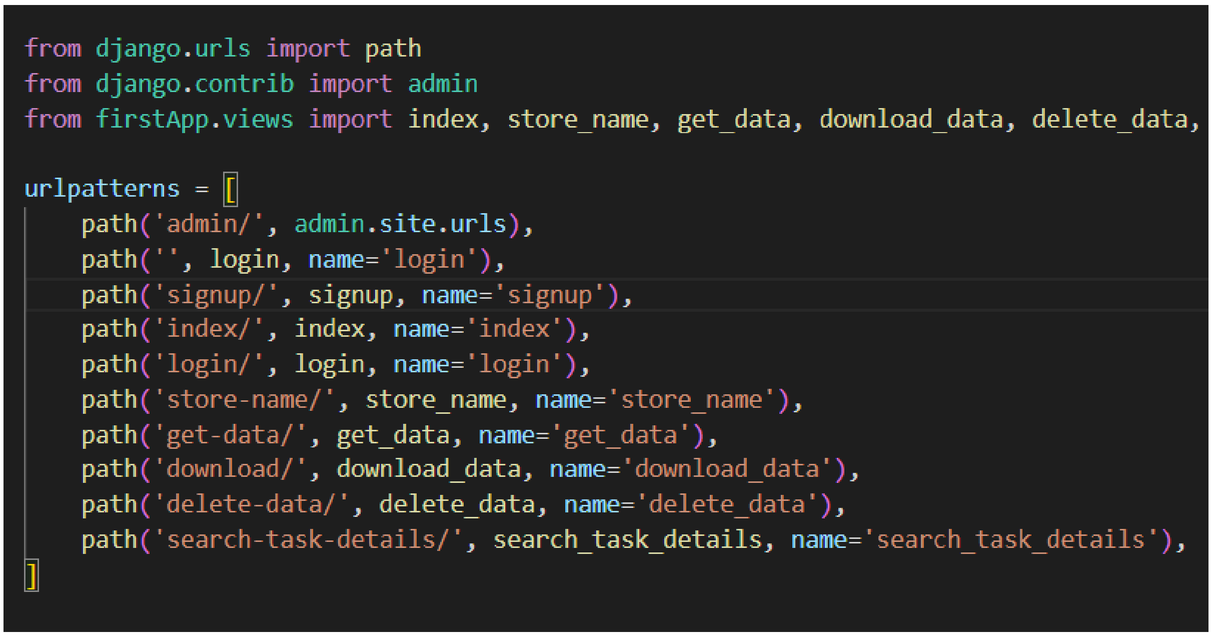
- If any other exception occurs during the process, it logs the error using `logger.exception()` and returns a JSON response indicating an error with the message 'Failed to search task details' and a status code of 500 (Internal Server Error).

UI:



This function serves as an endpoint for searching task details based on the provided ID. It ensures that the request method is POST and handles various scenarios such as missing IDs, invalid IDs, and database errors, providing appropriate responses accordingly.

Urls.py



```
from django.urls import path
from django.contrib import admin
from firstApp.views import index, store_name, get_data, download_data, delete_data,

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', login, name='login'),
    path('signup/', signup, name='signup'),
    path('index/', index, name='index'),
    path('login/', login, name='login'),
    path('store-name/', store_name, name='store_name'),
    path('get-data/', get_data, name='get_data'),
    path('download/', download_data, name='download_data'),
    path('delete-data/', delete_data, name='delete_data'),
    path('search-task-details/', search_task_details, name='search_task_details'),
]
```

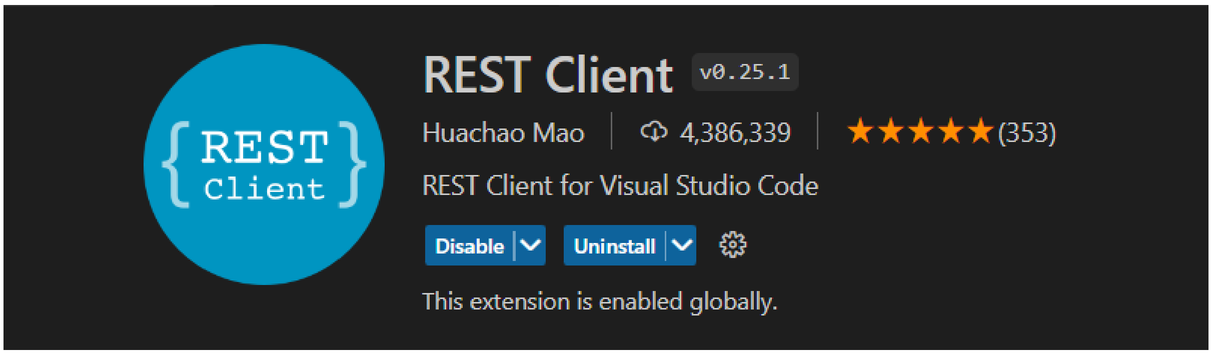
I have connected all endpoints to the urls.py file. Since we have out views.py file inside firstApp I am importing views from it.

"from firstApp.views import ……..."

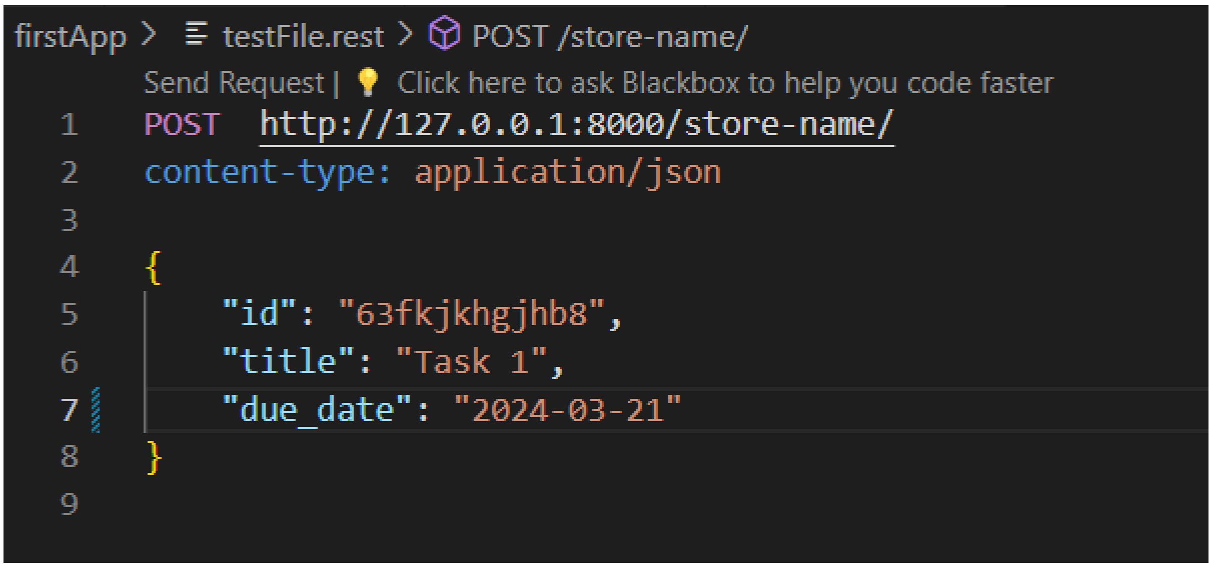
Testing endpoint inside your development environment:

We use tools like postman to test our endpoints. But I use an extension called "REST CLIENT" which test our endpoints by few lines of code.

First to do that we need to download an extension inside your IDE. I use Visual Studio Code.


REST Client
v0.25.1
Huachao Mao
4,386,339
★★★★★(353)
REST Client for Visual Studio Code
Disable
Uninstall
This extension is enabled globally.

After installing create an file called "test.rest" we use ".rest" extension for it. After creating open it and code as follow:


firstApp > testFile.rest > POST /store-name/
Send Request | Click here to ask Blackbox to help you code faster

```
POST  http://127.0.0.1:8000/store-name/
content-type: application/json

{
    "id": "63fkjkhgjhb8",
    "title": "Task 1",
    "due_date": "2024-03-21"
}

```

In this I make a POST request to the address http://127.0.0.1:8000/store-name/

with the following json data:

{

“id”:”1”

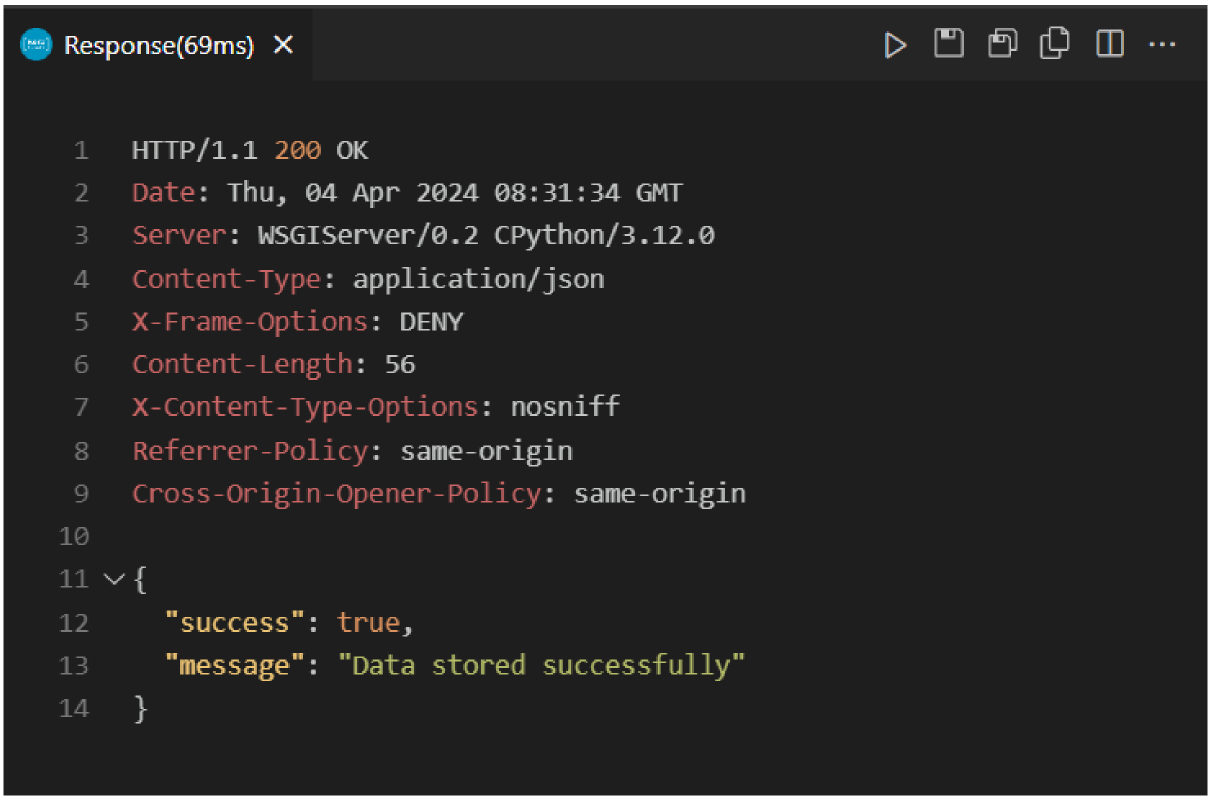
“title”:”task-1”

“due_date”:”2024-05-01”

}

Click **Send Request** button

Now, the rest client send this request containing Json data to the server and get response from it



```
HTTP/1.1 200 OK
Date: Thu, 04 Apr 2024 08:31:34 GMT
Server: WSGIServer/0.2 CPython/3.12.0
Content-Type: application/json
X-Frame-Options: DENY
Content-Length: 56
X-Content-Type-Options: nosniff
Referrer-Policy: same-origin
Cross-Origin-Opener-Policy: same-origin

{
  "success": true,
  "message": "Data stored successfully"
}
```

This is no essential but I faced some issues in installing postman. So I found this way.

Conclusion:

In this project, I've developed a Django application for managing tasks. The application provides functionalities such as user authentication, task creation, deletion, search, and data retrieval. We've utilized Django's built-in functionalities along with Django REST Framework for creating RESTful APIs. By implementing views, serializers, and models, we've established a robust backend structure to handle various operations seamlessly. Additionally, we've incorporated logging to capture and handle errors effectively. Overall, the project demonstrates the power and versatility of Django in building scalable web applications.

Output:

Upon running the application, users can interact with the following functionalities:

1. User Signup and Login: Users can sign up for an account and log in securely.

2. Task Management: Users can create tasks, delete tasks by ID, search for task details using task IDs, and retrieve all tasks.

3. File Handling: The application can read data from Database and a file named \`'name.txt'\` and extract task information.

4. Error Handling: Errors are handled gracefully, and appropriate error messages are returned to users.

5. Security: CSRF protection is implemented for secure form submissions, and authentication tokens are generated for user sessions.

Overall, the application provides a seamless experience for managing tasks, ensuring data integrity and security throughout the process.

Errors I Faced during this Project:

In this I have added error that I encountered both in front-end (React JS) and backend (Django).

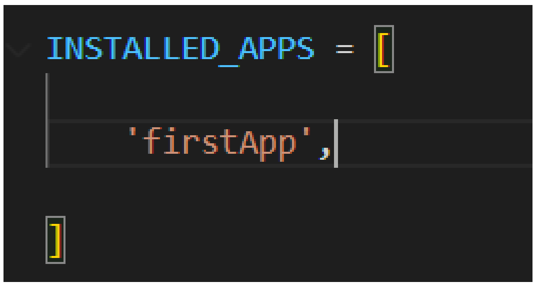
Blank White Screen: When you face this problem it mean =s the problem is with frontend code. The file will be running no errors will be shown by terminal but the UI will be blank so check you last modification in frontend.

Error code 404: This error commonly occurs when a file or packages is missing. So to resolve this check the terminal. It will give you the exact information about the missing file. I faced this because I forget to install rest_framework after moving my file to a different location

Backend Not working: If your backend code dosen't work the best way to debug is check your terminal for error code or to get more detail about the error use loggers which I have used. This will print a detailed info about the error in the terminal or use postman or rest client for testing endpoints. This will give an idea to start debugging.

Not Found: Manifest.json: What I learned form this error is just ignore it. I still face this issue but by code is working fine. Keep in mind if you see this in terminal and still your code runs, never disturb it. I will update a solution for it in future.

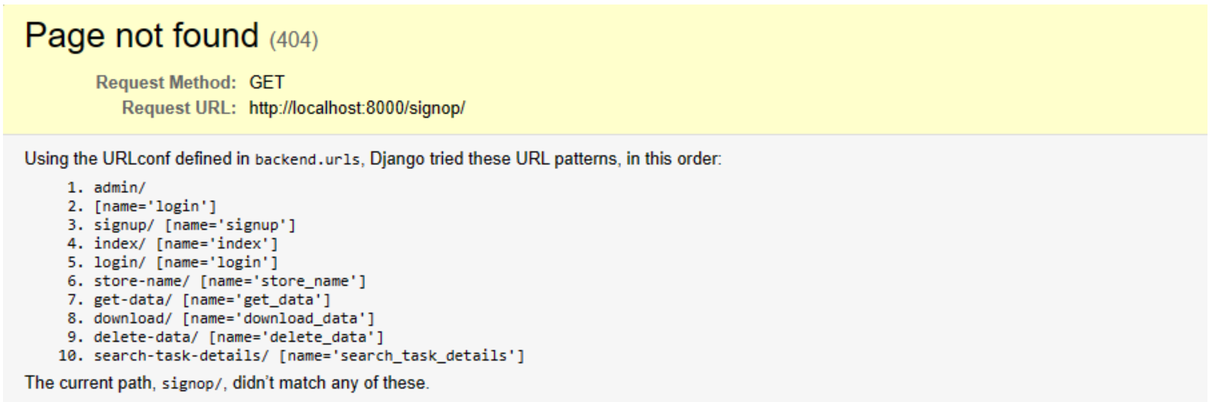
RuntimeError: Model class firstApp.models.Task doesn't declare an explicit app_label and isn't in an application in INSTALLED_APPS: This looks like a serious error but this is as simple as it is. The problem is when you create an App you should declare to your project directory that you have created an app like this



```
INSTALLED_APPS = [

    'firstApp',

]
```

Database not migrating: when connecting your database double check the information that you enter in settings.py. because I missed a letter in my password and I was searching for the error for whole day.

Page not found (404):



```
Page not found (404)
        Request Method:  GET
           Request URL:  http://localhost:8000/signop/

Using the URLconf defined in backend.urls, Django tried these URL patterns, in this order:
     1. admin/
     2. [name='login']
     3. signup/ [name='signup']
     4. index/ [name='index']
     5. login/ [name='login']
     6. store-name/ [name='store_name']
     7. get-data/ [name='get_data']
     8. download/ [name='download_data']
     9. delete-data/ [name='delete_data']
    10. search-task-details/ [name='search_task_details']
The current path, signop/, didn't match any of these.
```

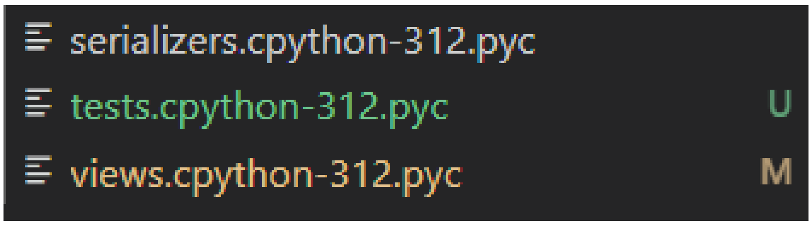
If you see this page It means that the url which you gave is wrong or the url is not defined. So check the spelling once or check your urls.py file to sort out the issue.

Git related issues:

Fatal: not a git repo- : if you face this issue you are inside a wrong directory which dosent have git access. Make sure git open a correct directory. To go g=back to a directory in terminal use "cd .." make sure you give space. To go into a dir use "cd <dir name>"

Understanding the colours:

I was totally confused why my files are in red, green and yellow colour then later I came to know that it Is because of git.



Yellow colour:

Your file name will be highlighted in yellow colour when you make changes to a file which is pushed into git. If the file is modified, your file will be highlighted in yellow colour. This will be solved once you make commit.

Green Colour:

This means the file is created now. And if your file is highlighted with green colour, it means the file is not added to your git. So after you use “git add .” or “git add <file name>” the file will turn white.

3rd Problem set

Topics Covered:

- Data Protection- Implementing Encryption
- Automated Testing for APIs
- Creating a Command-Line Interface for API Interaction

Abstract:

In today's digital world, ensuring data protection is paramount for any application or system. Encryption serves as a fundamental technique to safeguard sensitive information from unauthorized access. In this project, we explore the implementation of encryption to protect data stored in files. Additionally, we delve into automated testing for APIs, leveraging Selenium for web automation testing. Furthermore, we enhance user interaction with the API by developing a Command-Line Interface (CLI) using argparse in Django. This CLI empowers users to perform essential tasks such as creating, deleting, and listing saved tasks seamlessly.

Introduction:

With the increasing prevalence of data breaches and cyber threats, the need for robust data protection mechanisms has become more critical than ever. Encryption stands as one of the most effective methods to safeguard data, ensuring confidentiality and integrity. In this project, we focus on implementing encryption to secure data stored in files. By encrypting sensitive information, we mitigate the risk of unauthorized access and data leakage.

Moreover, ensuring the reliability and functionality of APIs is essential in modern software development. Automated testing plays a pivotal role in this aspect, enabling developers to validate API behavior efficiently and consistently. We explore automated testing for APIs using Selenium, a powerful tool for web automation testing. Through automated testing, we can ensure the correctness and reliability of API endpoints, enhancing the overall quality of the software.

Furthermore, to enhance user interaction with the API, we introduce a Command-Line Interface (CLI) using argparse in Django. This CLI provides users with a convenient and intuitive way to interact with the API directly from the command line. Users can perform various tasks, such as creating, deleting, and listing saved tasks, without the need for a web client or graphical interface. By leveraging the CLI, users can streamline their workflow and efficiently manage tasks within the application.

Overall, this project aims to address the critical aspects of data protection, automated testing for APIs, and enhancing user interaction through a Command-Line Interface. By implementing encryption, conducting automated testing, and developing a CLI, we ensure the security, reliability, and usability of the application, thereby meeting the evolving needs and expectations of users in today's digital landscape.

Installing Required Packages:

1 . This Command is used to install the packages required for Cryptography.

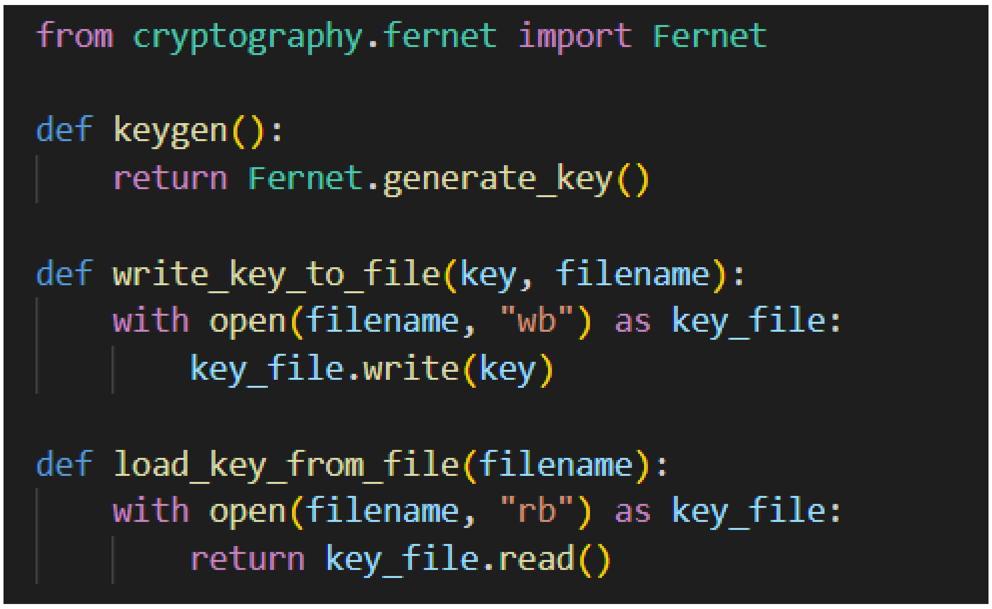
```
pip install django-cryptography
```

2 . We use this command to install selenium drivers. This is the initial step to install selenium

```
pip install selenium
```

Data Protection- Implementing Encryption

In this I have a text file for entering the data. The data entered inside the file will be encrypted and Decrypted using a key which is generated by my code.



```python
from cryptography.fernet import Fernet

def keygen():
    return Fernet.generate_key()

def write_key_to_file(key, filename):
    with open(filename, "wb") as key_file:
        key_file.write(key)

def load_key_from_file(filename):
    with open(filename, "rb") as key_file:
        return key_file.read()
```

1. `from cryptography.fernet import Fernet`: This line imports the `Fernet` class from the `cryptography.fernet` module. Fernet is a symmetric encryption algorithm used for securing data.

2. `def keygen():`: This line starts the definition of a function named `keygen`, which is responsible for generating encryption keys.

3. `return Fernet.generate_key()`: This line calls the `generate_key()` method of the `Fernet` class, which generates a random encryption key. The key is then returned by the function.

4. `def write_key_to_file(key, filename):`: This line defines a function named `write_key_to_file` that takes two parameters: `key` (the encryption key) and `filename` (the name of the file to write the key to).

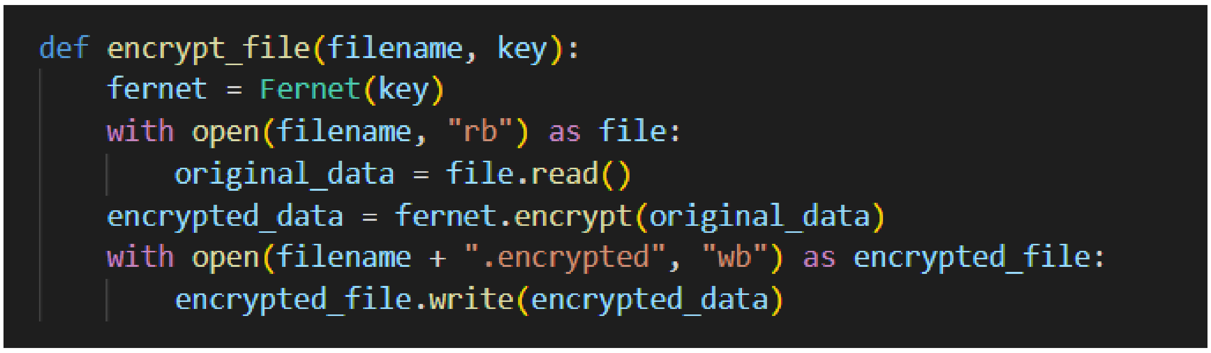
5. `with open(filename, "wb") as key_file:`: This line opens the specified file (`filename`) in binary write mode (`"wb"`) and assigns it to the variable `key_file`. The `with` statement is used to ensure that the file is properly closed after its suite (indented block) finishes executing.

6. `key_file.write(key)`: This line writes the encryption key (`key`) to the opened file (`key_file`).

7. `def load_key_from_file(filename):`: This line defines a function named `load_key_from_file` that takes one parameter: `filename` (the name of the file containing the encryption key).

8. `with open(filename, "rb") as key_file:`: This line opens the specified file (`filename`) in binary read mode (`"rb"`) and assigns it to the variable `key_file`. Similar to before, the `with` statement ensures proper handling of the file.

9. `return key_file.read()`: This line reads the contents of the opened file (`key_file`) and returns it. In this case, it reads the encryption key stored in the file.



```python
def encrypt_file(filename, key):
    fernet = Fernet(key)
    with open(filename, "rb") as file:
        original_data = file.read()
    encrypted_data = fernet.encrypt(original_data)
    with open(filename + ".encrypted", "wb") as encrypted_file:
        encrypted_file.write(encrypted_data)
```

1. `def encrypt_file(filename, key):`: This line defines a function named `encrypt_file` that takes two parameters: `filename` (the name of the file to be encrypted) and `key` (the encryption key).

2. `fernet = Fernet(key)`: This line creates an instance of the `Fernet` class with the provided encryption `key`. This instance will be used to perform encryption and decryption operations.

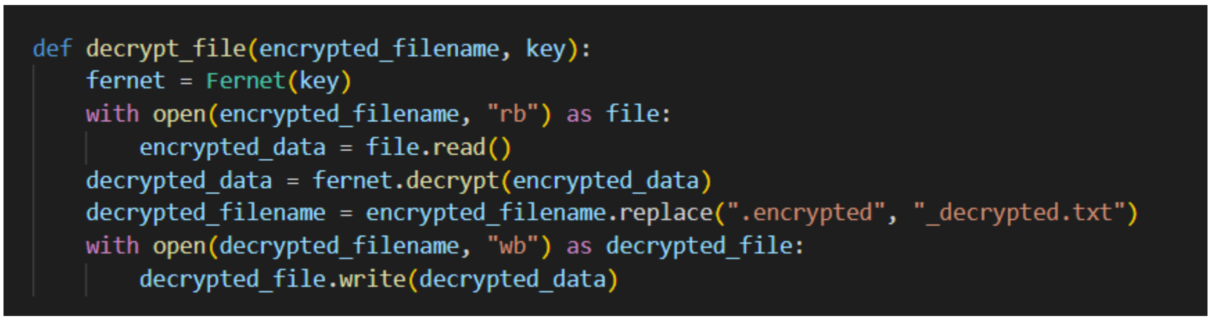
3. `with open(filename, "rb") as file:`: This line opens the specified file (`filename`) in binary read mode (`"rb"`) and assigns it to the variable `file`. The `with` statement ensures that the file is properly closed after its suite (indented block) finishes executing.

4. `original_data = file.read()`: This line reads the contents of the opened file (`file`) and assigns them to the variable `original_data`. This data will be encrypted.

5. `encrypted_data = fernet.encrypt(original_data)`: This line encrypts the `original_data` using the `encrypt` method of the `Fernet` instance (`fernet`). The encrypted data is assigned to the variable `encrypted_data`.

6. `with open(filename + ".encrypted", "wb") as encrypted_file:`: This line opens (or creates) a new file with the name `filename + ".encrypted"` in binary write mode (`"wb"`) and assigns it to the variable `encrypted_file`. This is where the encrypted data will be written. Again, the `with` statement ensures proper handling of the file.

7. `encrypted_file.write(encrypted_data)`: This line writes the encrypted data (`encrypted_data`) to the opened file (`encrypted_file`). The data is written in binary format, as specified by the mode `"wb"`.



```python
def decrypt_file(encrypted_filename, key):
    fernet = Fernet(key)
    with open(encrypted_filename, "rb") as file:
        encrypted_data = file.read()
    decrypted_data = fernet.decrypt(encrypted_data)
    decrypted_filename = encrypted_filename.replace(".encrypted", "_decrypted.txt")
    with open(decrypted_filename, "wb") as decrypted_file:
        decrypted_file.write(decrypted_data)
```

1. `def decrypt_file(encrypted_filename, key):`: This line defines a function named `decrypt_file` that takes two parameters: `encrypted_filename` (the name of the file to be decrypted) and `key` (the decryption key).

2. `fernet = Fernet(key)`: This line creates an instance of the `Fernet` class with the provided decryption `key`. This instance will be used to perform decryption operations.

3. `with open(encrypted_filename, "rb") as file:`: This line opens the specified encrypted file (`encrypted_filename`) in binary read mode (`"rb"`) and assigns it to the variable `file`. The `with` statement ensures proper handling of the file.

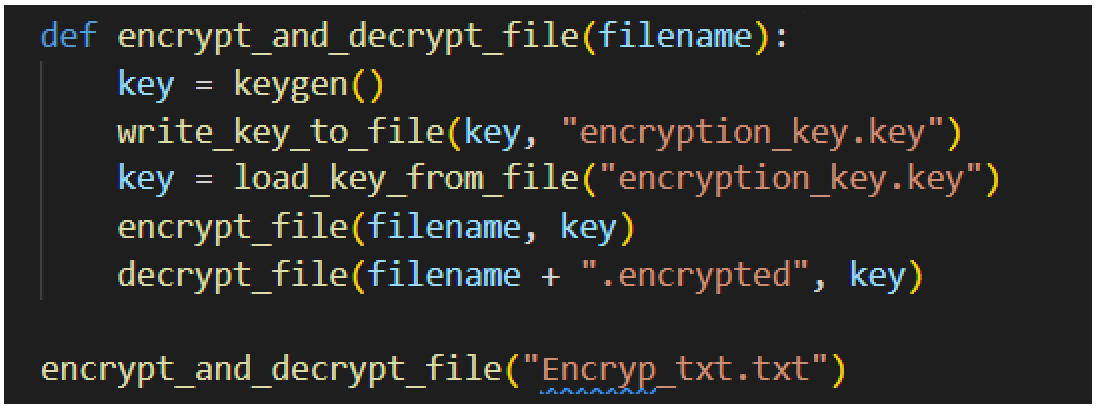
4. `encrypted_data = file.read()`: This line reads the contents of the opened encrypted file (`file`) and assigns them to the variable `encrypted_data`. These are the bytes representing the encrypted data.

5. `decrypted_data = fernet.decrypt(encrypted_data)`: This line decrypts the `encrypted_data` using the `decrypt` method of the `Fernet` instance (`fernet`). The decrypted data is assigned to the variable `decrypted_data`.

6. `decrypted_filename = encrypted_filename.replace(".encrypted", "_decrypted.txt")`: This line creates a new filename for the decrypted file by replacing the ".encrypted" extension in the original filename (`encrypted_filename`) with "_decrypted.txt". This assumes that the original filename follows the convention of having the ".encrypted" extension.

7. `with open(decrypted_filename, "wb") as decrypted_file:`: This line opens (or creates) a new file with the decrypted filename (`decrypted_filename`) in binary write mode (`"wb"`) and assigns it to the variable `decrypted_file`. The `with` statement ensures proper handling of the file.

8. `decrypted_file.write(decrypted_data)`: This line writes the decrypted data (`decrypted_data`) to the opened decrypted file (`decrypted_file`). The data is written in binary format, as specified by the mode `"wb"`.



```
def encrypt_and_decrypt_file(filename):
    key = keygen()
    write_key_to_file(key, "encryption_key.key")
    key = load_key_from_file("encryption_key.key")
    encrypt_file(filename, key)
    decrypt_file(filename + ".encrypted", key)

encrypt_and_decrypt_file("Encryp_txt.txt")
```

1. `def encrypt_and_decrypt_file(filename):`: This line defines a function named `encrypt_and_decrypt_file` that takes one parameter: `filename` (the name of the file to be encrypted and decrypted).

2. `key = keygen()`: This line generates a new encryption key using the `keygen` function and assigns it to the variable `key`.

3. `write_key_to_file(key, "encryption_key.key")`: This line writes the generated encryption key to a file named "encryption_key.key" using the `write_key_to_file` function. This file will be used later to load the key during decryption.

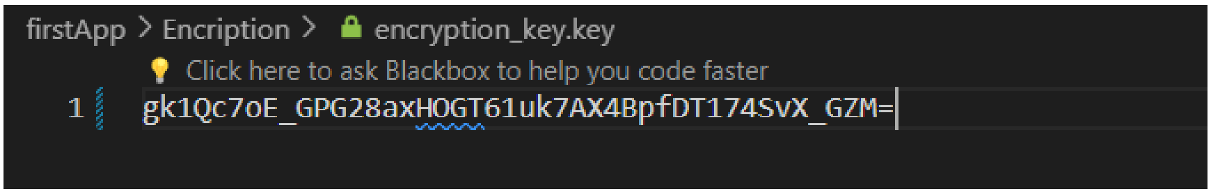
4. `key = load_key_from_file("encryption_key.key")`: This line loads the encryption key from the "encryption_key.key" file using the `load_key_from_file` function and assigns it to the variable `key`.

5. `encrypt_file(filename, key)`: This line encrypts the contents of the specified file (`filename`) using the encryption key (`key`) and the `encrypt_file` function.

6. `decrypt_file(filename + ".encrypted", key)`: This line decrypts the encrypted file (`filename + ".encrypted"`) using the decryption key (`key`) and the `decrypt_file` function.

7. `encrypt_and_decrypt_file("Encryp_txt.txt")`: This line calls the `encrypt_and_decrypt_file` function with the filename "Encryp_txt.txt", which starts the encryption and decryption process for that file.

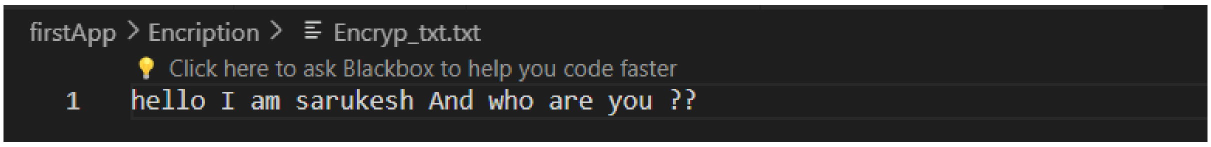
Generated key:


firstApp > Encription > encryption_key.key

Click here to ask Blackbox to help you code faster

```
gk1Qc7oE_GPG28axHOGT61uk7AX4BpfDT174SvX_GZM=
```

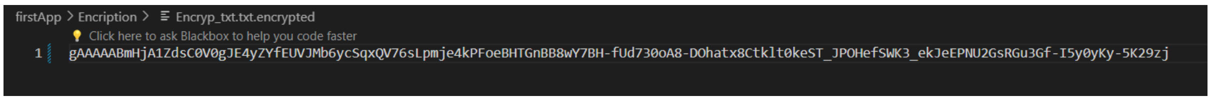
Input Text:


firstApp > Encription > Encryp_txt.txt

Click here to ask Blackbox to help you code faster

```
hello I am sarukesh And who are you ??
```

Encrypted Output:


firstApp > Encription > Encryp_txt.txt.encrypted

Click here to ask Blackbox to help you code faster

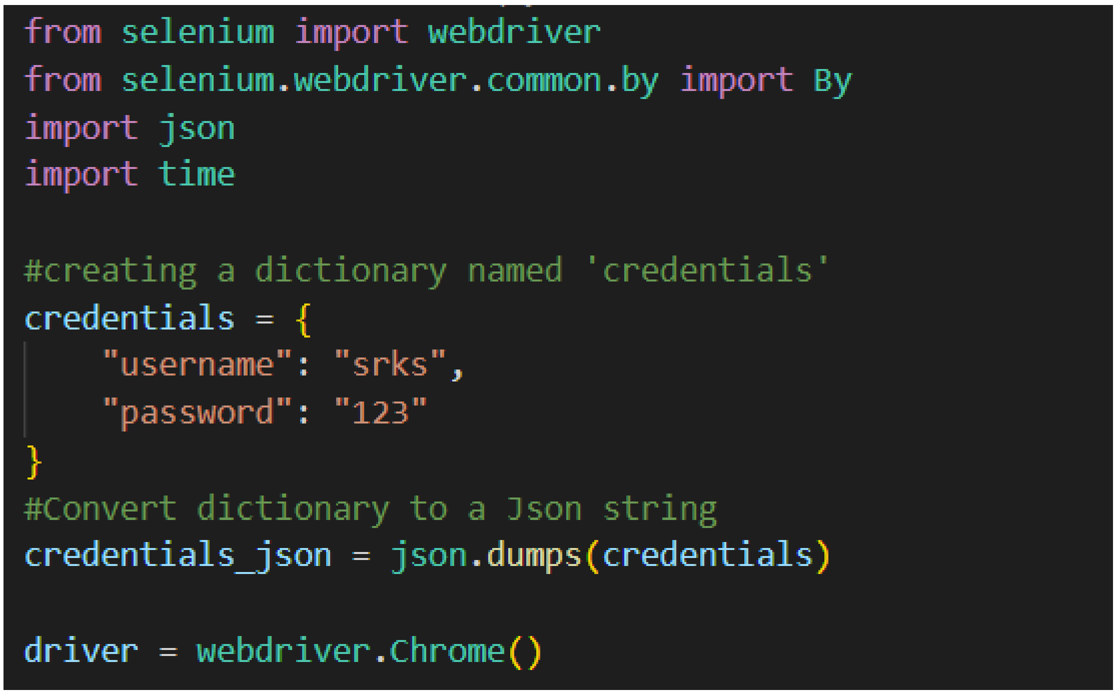
```
gAAAAABmHjA1ZdsC0V0gJE4yZYfEUVJMb6ycSqxQV76sLpmje4kPFoeBHTGnBB8wY7BH-fUd730oA8-DOhatx8Ctklt0keST_JPOHefSWK3_ekJeEPNU2GsRGu3Gf-I5y0yKy-5K29zj
```

Automated Testing for APIs

In this project I have used selenium for testing my web application. Selenium is used for testing web applications and for unit testing. I t can be automated too. I used selenium web driver for my unit testing. My selenium code tests all the functions and print the status in the terminal. It first run the url then maximize the window. It login the page with given credentials. Then it will create an task and search the same task and also delete the created task using id. After testing every functions it closes the window.

Now as I mentioned above (page no: 60) use the command and install the necessary packages.

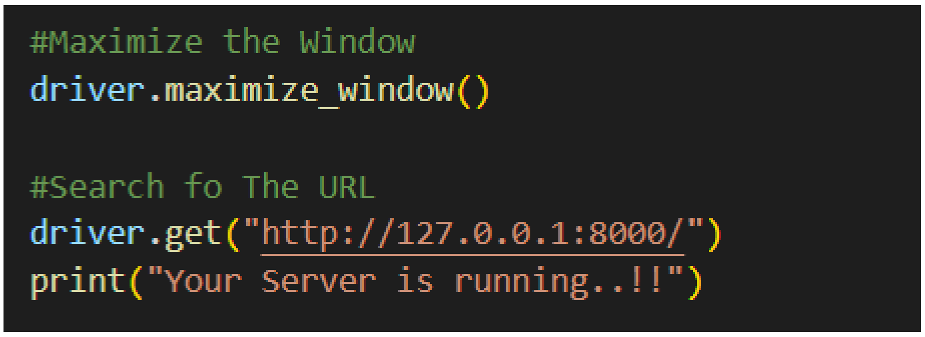
After initial installation we are importing the necessary modules for our selenium web driver



```python
from selenium import webdriver
from selenium.webdriver.common.by import By
import json
import time

#creating a dictionary named 'credentials'
credentials = {
    "username": "srks",
    "password": "123"
}
#Convert dictionary to a Json string
credentials_json = json.dumps(credentials)

driver = webdriver.Chrome()
```

And I store my login credentials inside a variable called Credentials in json format so that I can call the variable whenever I want.

In the last line “driver = webdriver.Chrome()” I am telling it to install drivers of chrome. This will oprn the url in chrome brower. U can use your desired browser as you wish

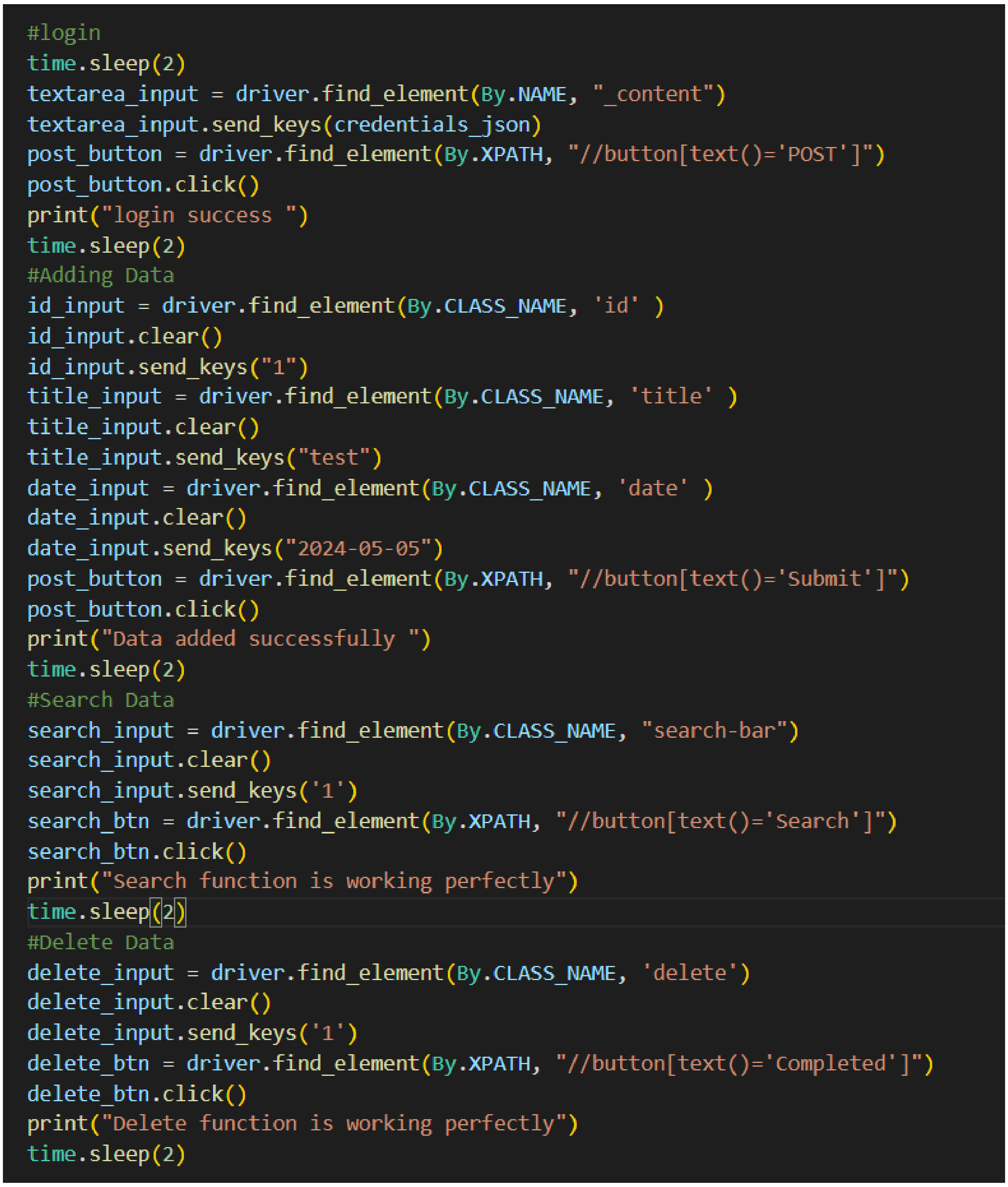


```python
#Maximize the Window
driver.maximize_window()

#Search fo The URL
driver.get("http://127.0.0.1:8000/")
print("Your Server is running..!!")
```

In this the code opens the chrome browser and maximize the window and search for the url:

http://127.0.0.1:8000/

|NOTE: THE DJANGO DEVELOPMENT SERVER MUST BE RUNNING |



```
#login
time.sleep(2)
textarea_input = driver.find_element(By.NAME, "_content")
textarea_input.send_keys(credentials_json)
post_button = driver.find_element(By.XPATH, "//button[text()='POST']")
post_button.click()
print("login success ")
time.sleep(2)
#Adding Data
id_input = driver.find_element(By.CLASS_NAME, 'id' )
id_input.clear()
id_input.send_keys("1")
title_input = driver.find_element(By.CLASS_NAME, 'title' )
title_input.clear()
title_input.send_keys("test")
date_input = driver.find_element(By.CLASS_NAME, 'date' )
date_input.clear()
date_input.send_keys("2024-05-05")
post_button = driver.find_element(By.XPATH, "//button[text()='Submit']")
post_button.click()
print("Data added successfully ")
time.sleep(2)
#Search Data
search_input = driver.find_element(By.CLASS_NAME, "search-bar")
search_input.clear()
search_input.send_keys('1')
search_btn = driver.find_element(By.XPATH, "//button[text()='Search']")
search_btn.click()
print("Search function is working perfectly")
time.sleep(2)
#Delete Data
delete_input = driver.find_element(By.CLASS_NAME, 'delete')
delete_input.clear()
delete_input.send_keys('1')
delete_btn = driver.find_element(By.XPATH, "//button[text()='Completed']")
delete_btn.click()
print("Delete function is working perfectly")
time.sleep(2)
```

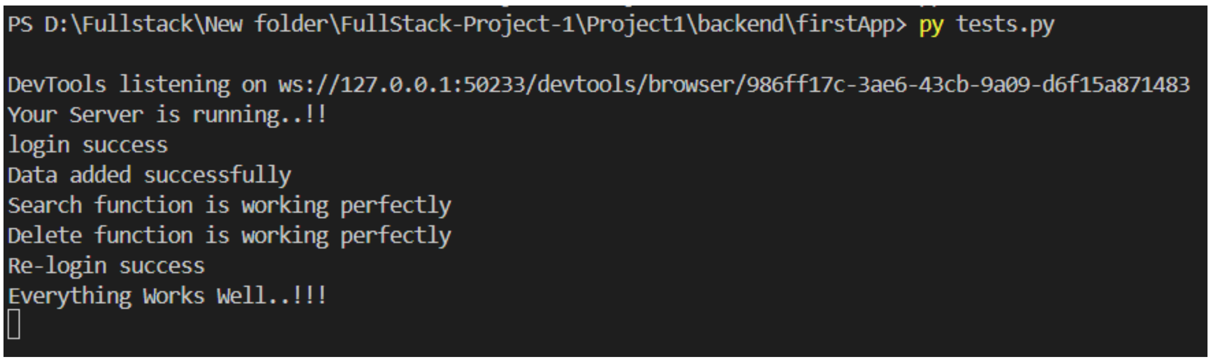
In this I have created my tasks that should selenium do. For example, after running the url it should wait for 2 sec and then I the second line, it will search for element with name “_content”

After finding the text area it will send key. It will print the json data which I have already declared inside a variable called credentials.

And then it will now search for the button names POST and click the button.

This what happens for every other part of code at last after executing my code I made it to quit the window. Meanwhile all the print statements will be printed one by one when it gets executed.

It looks like:

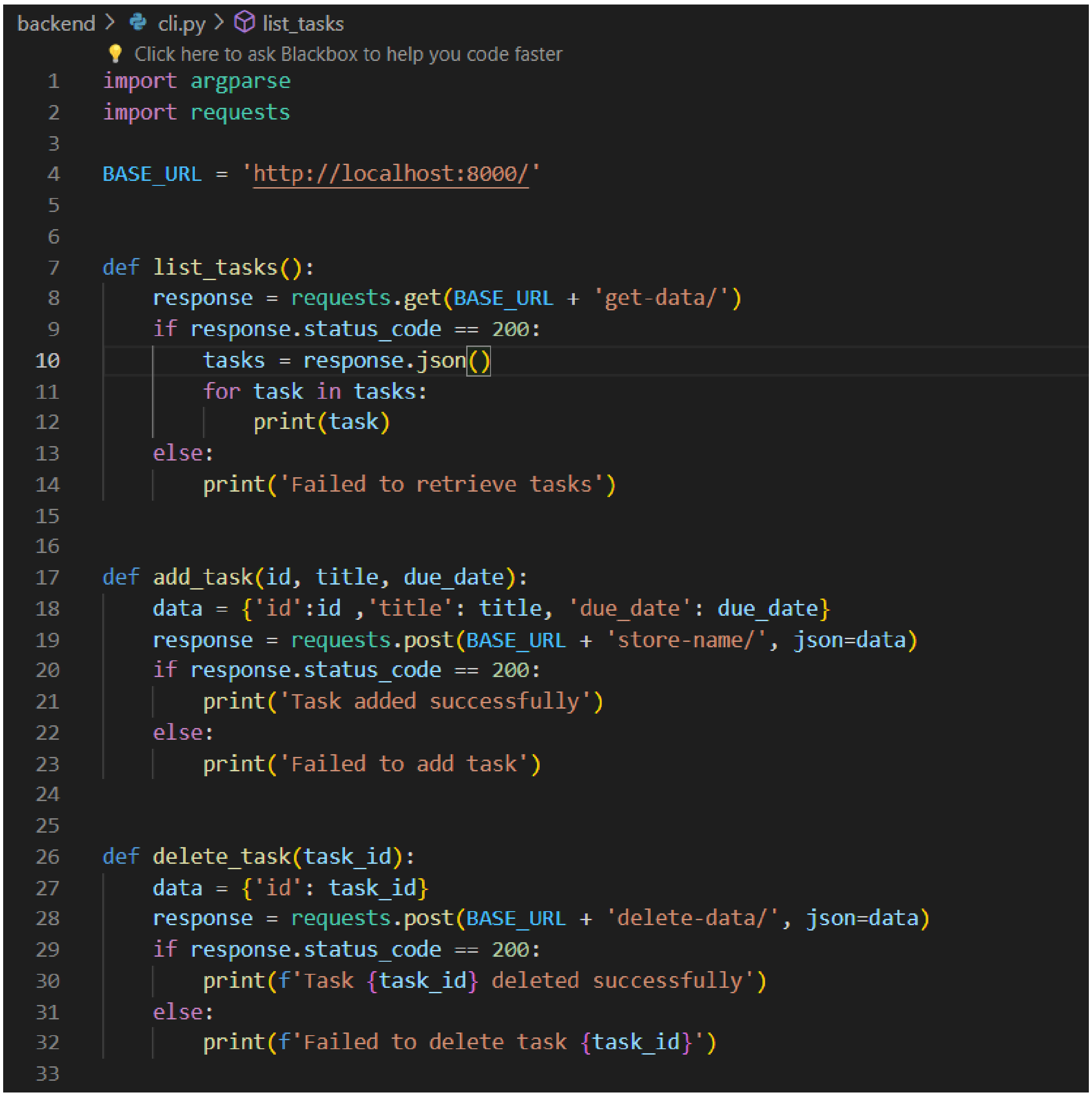


```
PS D:\Fullstack\New folder\FullStack-Project-1\Project1\backend\firstApp> py tests.py

DevTools listening on ws://127.0.0.1:50233/devtools/browser/986ff17c-3ae6-43cb-9a09-d6f15a871483
Your Server is running..!!
login success
Data added successfully
Search function is working perfectly
Delete function is working perfectly
Re-login success
Everything Works Well..!!!
```

After this the window closes.

Creating a Command-Line Interface for API Interaction

By using this we can perform all the actions in command line. We can add tasks, delete tasks and list all the saved tasks.


backend > cli.py > list_tasks
Click here to ask Blackbox to help you code faster

```python
import argparse
import requests

BASE_URL = 'http://localhost:8000/'


def list_tasks():
    response = requests.get(BASE_URL + 'get-data/')
    if response.status_code == 200:
        tasks = response.json()
        for task in tasks:
            print(task)
    else:
        print('Failed to retrieve tasks')


def add_task(id, title, due_date):
    data = {'id':id ,'title': title, 'due_date': due_date}
    response = requests.post(BASE_URL + 'store-name/', json=data)
    if response.status_code == 200:
        print('Task added successfully')
    else:
        print('Failed to add task')


def delete_task(task_id):
    data = {'id': task_id}
    response = requests.post(BASE_URL + 'delete-data/', json=data)
    if response.status_code == 200:
        print(f'Task {task_id} deleted successfully')
    else:
        print(f'Failed to delete task {task_id}')

```

1. `list_tasks()`: This function sends a GET request to the API endpoint `BASE_URL + 'get-data/'` to retrieve a list of tasks. If the response status code is 200 (OK), it prints each task. Otherwise, it prints an error message.

2. `add_task(id, title, due_date)`: This function sends a POST request to the API endpoint `BASE_URL + 'store-name/'` to add a new task. It takes three

arguments: `id`, `title`, and `due_date`, and sends them as JSON data in the request body. If the response status code is 200, it prints a success message; otherwise, it prints an error message.

3. `delete_task(task_id)`: This function sends a POST request to the API endpoint `BASE_URL + 'delete-data/'` to delete a task with the specified `task_id`. It sends the `task_id` as JSON data in the request body. If the response status code is 200, it prints a success message; otherwise, it prints an error message.

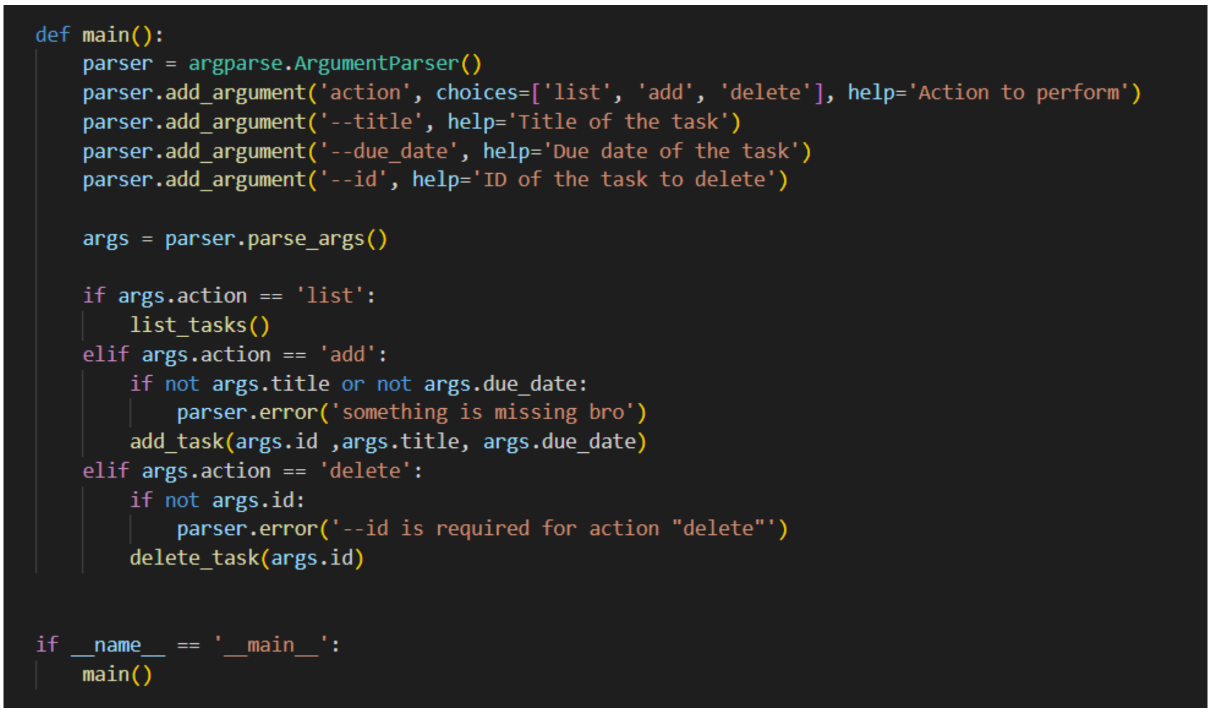
This `main()` function is the entry point of the script. It uses the `argparse` module to parse command-line arguments and determine the action to perform.

1. `parser = argparse.ArgumentParser()`: Creates an ArgumentParser object to define the command-line interface.

2. `parser.add_argument('action', choices=['list', 'add', 'delete'], help='Action to perform')`: Adds a positional argument named `action` that accepts values from the choices list ['list', 'add', 'delete']. This argument specifies the action to be performed.

3. `parser.add_argument('--title', help='Title of the task')`: Adds an optional argument named `--title` to specify the title of the task to add.

4. `parser.add_argument('--due_date', help='Due date of the task')`: Adds an optional argument named `--due_date` to specify the due date of the task to add.


```python
def main():
    parser = argparse.ArgumentParser()
    parser.add_argument('action', choices=['list', 'add', 'delete'], help='Action to perform')
    parser.add_argument('--title', help='Title of the task')
    parser.add_argument('--due_date', help='Due date of the task')
    parser.add_argument('--id', help='ID of the task to delete')

    args = parser.parse_args()

    if args.action == 'list':
        list_tasks()
    elif args.action == 'add':
        if not args.title or not args.due_date:
            parser.error('something is missing bro')
        add_task(args.id ,args.title, args.due_date)
    elif args.action == 'delete':
        if not args.id:
            parser.error('--id is required for action "delete"')
        delete_task(args.id)


if __name__ == '__main__':
    main()
```

5. `parser.add_argument('--id', help='ID of the task to delete')`: Adds an optional argument named `--id` to specify the ID of the task to delete.

6. `args = parser.parse_args()`: Parses the command-line arguments provided by the user.

7. Based on the value of `args.action`, it calls the corresponding function: `list_tasks()`, `add_task()`, or `delete_task()`. If `args.action` is 'list', it calls `list_tasks()`. If it's 'add', it calls `add_task()` with the provided title and due date. If it's 'delete', it calls `delete_task()` with the provided task ID.

8. Error handling: It checks if required arguments are missing and displays an error message using `parser.error()` if necessary.

9. Finally, `main()` is called if the script is executed directly (`if __name__ == '__main__':`). This ensures that `main()` is executed when the script is run from the command line.

References:

Programming: https://kurzy.kpi.fei.tuke.sk

Django: www.djangoproject.com

Django-rest-framework: www.django-rest-framework.org

React: www.react.com

Stack-overflow: www.stackoverflow.com

W3 Schools: www.w3schools.com

Chatgpt: chat.open.ai

Black Box AI: vs code extension

Selenium: www.selenium.dev

Postgresql: postgrsqltutorial.com

Contact:

Name: Sarukesh Boominathan

Subject: Programming

Study programme: Bachelors in Informatics

Department: Department of Informatics

Email: sarukesh.boominatha@student.tuke.sk

Website: www.sarukesh.com