Synertics Project Documentation

André Rodrigues-rodrigues.n.andre46@gmail.com

May 11, 2025

Contents

1 Project Overview

Synertics is a Django-based web application that performs automated data scraping from greek electricity futures prices, processes the data, and provides it through a REST API and web interface. The system includes scheduled tasks, email notifications, and a modern web interface.It meets all requirements in the guide, including the bonus ones.

1.1 Key Features

- Automated daily data scraping: The system gathers fresh data every day without manual effort(also manually activated if needed).
- REST API: Easy acess to data scraped via Django Rest Framework.
- Real-time data processing: Data scraped will go into a PostgreSQL database, and processed for up-to-date insights.
- Email notifications: Errors in data scraping will send a email notification .
- Progressive Web App : Acess the platform like a native app in all devices.
- Docker containers:The project is done in containers to ensure isolated and reproducible development and deployment environments.

2 System Architecture

2.1 Django Web Framework

The core of the application is built using the Django web framework, which provides robust support for:

- Core application logic, organizing views, models, and business processes
- URL routing to direct requests to the appropriate views
- Template rendering for dynamic HTML generation
- Built-in admin interface for managing data

2.2 Celery Task Queue

To handle background operations and improve performance, the project integrates Celery, a distributed task queue. It enables:

- Scheduled data scraping tasks executed daily
- Redis is used as the message broker to manage task communication efficiently

2.3 PostgreSQL database

PostgreSQL serves as the project's relational database, providing:

- Reliable and structured data storage
- Robust transaction management to ensure data consistency

2.4 REST API

The application exposes a RESTful API to enable external access to its data and services, including:

- Well-defined data endpoints to get data
- JSON as the standard response format for client compatibility
- CORS support to allow secure cross-origin requests

2.5 Directory Structure

The main directories for this project are:

```
Synertics/
|-- core/                  # Main application
|   |-- api/               # REST API endpoints
|   |-- migrations/        # Database migrations
|   |-- static/            # Static files
|   |-- templates/         # HTML templates
|   |-- admin.py           # Admin configuration
|   |-- models.py          # Database models
|   |-- tasks.py           # Celery tasks
|   |-- urls.py            # URL routing
|   `-- views.py           # View logic
|-- Synertics/             # Project settings
|   |-- settings.py        # Project configuration
|   |-- urls.py            # Main URL routing
|   `-- wsgi.py            # WSGI configuration
|-- static/                # Project-wide static files
|-- logs/                  # Application logs
|-- manage.py              # Django management script
|-- Dockerfile             # Instructions to build container
`-- docker-compose.yaml    # Creates the many containers
```

3 Configuration guide

3.1 Pre-requisites

The system needs Docker installed to run, since this application is built on docker containers. These containers have all the installations needed.

The system only needs a small configuration for the notification system. All the rest is already ready to use. In Synertics/settings.py, the user needs to write their email information so it gets notified when there's an error in webscraping. The lines are:
EMAIL_HOST_USER = 'your-email@gmail.com' # Your Gmail address
EMAIL_HOST_PASSWORD = 'your-app-password' # Your Gmail app password
DEFAULT_FROM_EMAIL = 'your-email@gmail.com' # Same as EMAIL_HOST_USER
ADMIN_EMAIL = 'admin@yourdomain.com' # Where to send error notifications
Users should change these credentials to their own credentials to make the notification system work. After this, run the command "docker-compose up –build" to create all the docker containers , and the system will be ready to use.

4 Routes

The system has 3 routes:

- "/admin/", to use the built in django admin interface.
- "",to see the chart of data from the 7 previous days.
- "/call_scraper/", to manually activate the scraper function for the current day.

5 Interface

The interface is as close as possible to the design provided in the guide for this project.

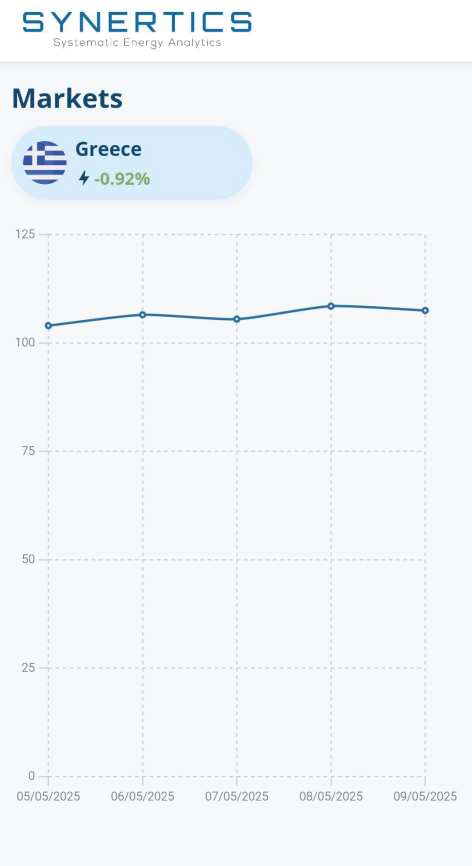

Figure 1: Mobile Interface

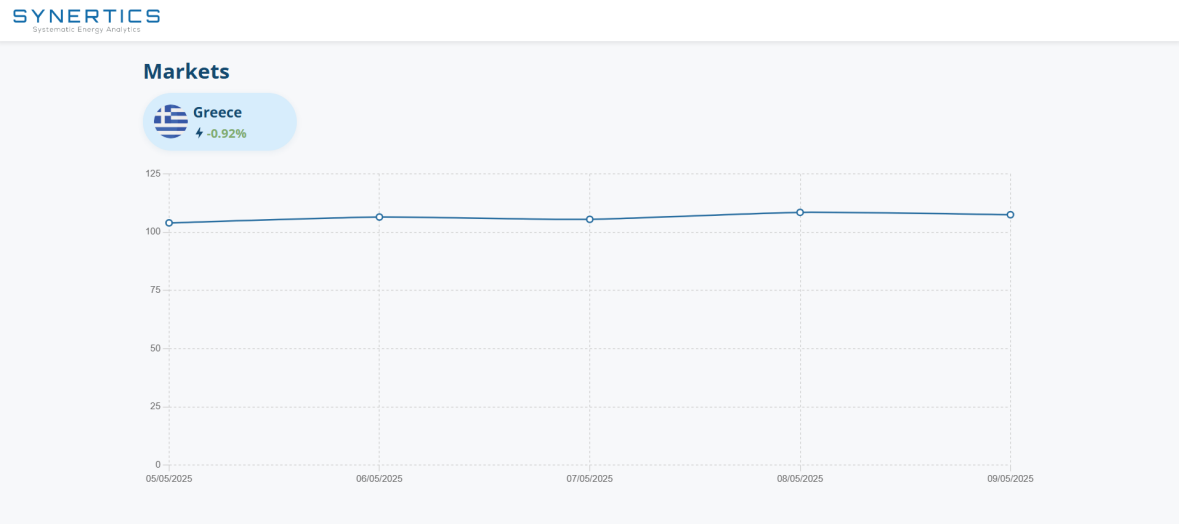

Figure 2: Desktop interface

6 API Endpoints

Every endpoint will return data in Json.

- Get all routes:"/api/", returns a list of available api endpoints.

- Get chart data:"/api/chartdata/", returns the data contained in the chart.
- Get percentage:"/api/percentage/", returns the percentage variation of today and yesterday data.
- Get all prices:"/api/allprices/", returns all "BY" prices and dates for every day recorded.
- Get Trades:"/api/trades/", returns all Trade data, including every column in the xlsx file.
- Get Trade by Id:"/api/trade/:id", returns the Trade object with that Id.
- Get Trades by Date:"/api/trade/date/YYYY-MM-DD", returns all trades from that day.

7 Requirements

7.1 Chart with market trends

The system uses chart.js to create the line chart to represent energy market trends.In core/views.py, when a user goes to the chart route, the system makes the necessary data processing to search into the database for the dates and corresponding average price or orders, and sends the result to the html page, serializing it to JSON.The javascript code transforms the JSON strings in js objects, and creates the chart with the data.
To process the data, we used django models functionalities to filter data only from the last 7 days, as required, and only the instruments containing "BY", meaning the yearly baseloads.This process has logging so the administrator can see information from the data selection.If there's no data in a day, the code will just skip it and not add it to the chart.

7.2 Responsive design for desktop and mobile

The system implements django-pwa, to create a progressive web app. This implementions need some configuration on settings.py, including the app names and icons.A path to pwa also needs to be added in urls.py.
To be mobile responsive, the javascript code has a function to check if a device is mobile, and changes the html page accordingly.The code was made for the page to be as close to the design as possible.

7.3 Price variation

This variation percentage is calculated alongside with the data from the graph,with the formula:

$$variation = \frac{price_{today} - price_{yesterday}}{price_{yesterday}} * 100 \tag{1}$$

That information is sent to the html page who shows it above the chart.

7.4 Daily scraping

The system scrapes data from the required link every day at 9 pm (Lisbon time)(the ENEX server is updated at 6 pm, so this should be safe),so users can see the recent market information. For this, the system uses celery-beat and redis as a broker for celery, to create a scheduled automated task to scrape from the link.These need some configurations in settings.py , mainly setting up the time for doing the task.A file celery.py was also added for django configuration. The task, on tasks.py, checks the current day, and with that information scrapes the link for information of that day.Using requests import, gets the response from the link and with pandas transforms the xlsx file into a pandas dataframe.The link doesn't have data everyday, so the system is able to deal with no responses, by skipping the day.The information in the dataframe is then added to the database, which will be described in the next section. This scraping can also be manually activated as described before.

7.5 Data storing

The system has a django model created for the object Trade, which contains every attribute from a line from the excel file.This creates the SQL table for this objects.
In the tasks.py described before, when the data is scraped and transformed into a pandas dataframe, we use the create method from the django models to add every line in the excel to add the Trade object in the database, with all the attributes in the columns. The database is implemented in PostgreSQL,which needs some configuration on settings.py and .env file.

7.6 REST API

The system has a REST API, which uses the django REST framework.This was created in folder /api.The views.py presents all the GET requests described previously, which helps to integrate with other frontend components.The output from the get requests is in JSON format,using ModelSerializer the rest framework provides.

7.7 Percentage change

The variation calculated for the page is also presented in JSON format in API to display derived metrics.

7.8 Code documentation

This pdf explains the project, but the code itself has various comments to help understand what the code does.

7.9 Notifications

The system will send email notifications in case the web scraping process fails. This was configured in settings.py(and needs further configuration from the user as explained before).This is done in tasks.py, where if the scraping fails, a catch will send a notification to user email.

7.10 Scraping Logs

The scraping process has various logs, and adds those logs into the logs/scrape.log file. This logging uses the package logging, and needed configuration in settings.py. The system logs when the scraping starts,if the data was successfully scraped or not, and saving to the database.

7.11 Extra Requirements

- The system uses Django default authentication system, but some configuration on settings.py were added to make sure the dashboard was accessible to everyone.Installation django-cors-headers and configuration is needed to allow access to a frontend component requests data from the API.The REST API also has no restrictions.
- The system was implemented in Docker, so it runs on an isolated environment, and works with another host device.
- The challenge will be sent in both a zip file and in a Git repository.