

2023-2024 ACADEMIC YEAR

SPRING SEMESTER

CENG 304/314  
COMPUTER NETWORKS

FINAL PROJECT REPORT

Dashboard with Real-Time  
Monitoring

Prepared By  
Melike Kara  
200201013

## CONTENTS

ABSTRACT

1. INTRODUCTION

2. DASHBOARD DEVELOPMENT THAT PROVIDES REAL-TIME MONITORING

2.1. PROJECT SETUP

2.1.1. DJANGO PROJECT SETUP

2.1.2. CREATING THE DJANGO APPLICATION

2.2. DATABASE MODELING AND MANAGEMENT

2.2.1. MODEL DEFINITION

2.2.2. ADMIN INTERFACE SETUP

2.3. DATABASE MIGRATIONS AND MANAGEMENT

2.3.1. MIGRATIONS

2.4. CUSTOM MANAGEMENT COMMANDS

2.4.1. CREATION OF CUSTOM MANAGEMENT COMMANDS

2.4.2. TESTING AND EXECUTION

3. PROPOSED APPROACH FOR THE PROJECT

3.1. TECHNOLOGY SELECTION

3.2. PROJECT STRUCTURE

3.3. DATA UPDATE AND MONITORING

3.4. EXECUTABLE COMMAND SCRIPT

3.5. APPLICATION STRUCTURE

4. RESULTS AND DISCUSSION

4.1. FUNCTIONALITY

4.2. EFFICIENCY

4.3. USER FEEDBACK

4.4. CHALLENGES ENCOUNTERED AND LESSONS LEARNED

4.5. FUTURE DEVELOPMENTS AND ADDITIONS

5. CONCLUSION

REFERENCES

APPENDICES

## **ABSTRACT**

This project aims to develop a real-time monitoring system for IoT devices using Django, a high-level Python web framework. Real-time monitoring is a method used to monitor the instantaneous status of IoT devices and make the necessary interventions in a timely manner. The system integrates real-time monitoring capabilities into a control panel to provide instant information about the status of IoT devices. The control panel visualizes measurements such as device connectivity, sensor readings, and operational status using data streaming and visualization techniques.

The development process includes the following steps:

- Firstly, a new Django project named "iot\_dashboard" is created, followed by initiating a Django application named "device\_monitor" within the project directory.
- Database models are defined to store information about IoT devices, including device ID, connection status, last seen timestamp, temperature, and humidity.

The Django admin interface is configured for easy management of device data, and database migrations are performed to update the database schema according to the defined models. Custom management commands are created for periodic updating of device data and monitoring device data in the terminal.

An executable command file named "run\_commands.bat" is developed to automate Django commands. The entire setup is verified by adding device data through the Django admin panel and observing updates and monitoring outputs in the terminal.

In conclusion, this project provides an abstract real-time monitoring and management system for IoT devices with the use of Django.

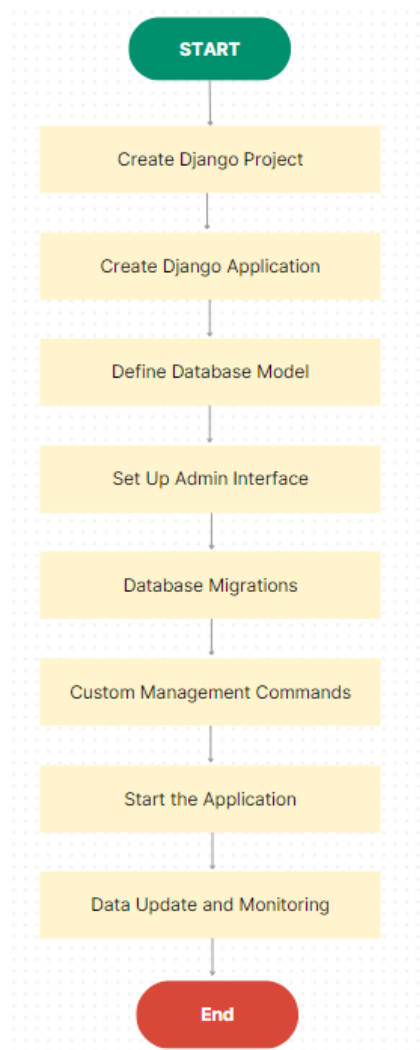
## 1. INTRODUCTION

Today, **Internet of Things (IoT)** technology has emerged as a transformative technology, enabling data exchange and automation through the connection and communication of various devices. In IoT ecosystems, **real-time monitoring** of devices is of critical importance to ensure efficient operation and timely interventions. However, developing effective real-time monitoring systems for IoT devices brings along many challenges, including data collection, processing, visualization, and management.,

To overcome these challenges, this project aims to develop a real-time monitoring system for IoT devices using Django, a powerful web framework for building scalable and sustainable web applications.

In this report, we present the methodology and implementation details of the developed system. The project includes installation, database modeling, administration interface setup, migration, and database setup, creation of custom management commands, testing, and execution.

The flow chart applied in the project is as follows:



## **2. DASHBOARD DEVELOPMENT THAT PROVIDES REAL-TIME MONITORING**

This control panel, created using Django, provides users with the information they need to make informed decisions by instantly displaying the status of IoT devices. Key steps of the project include creating the Django project and application, database modeling and management, and creating and testing custom administrative commands.

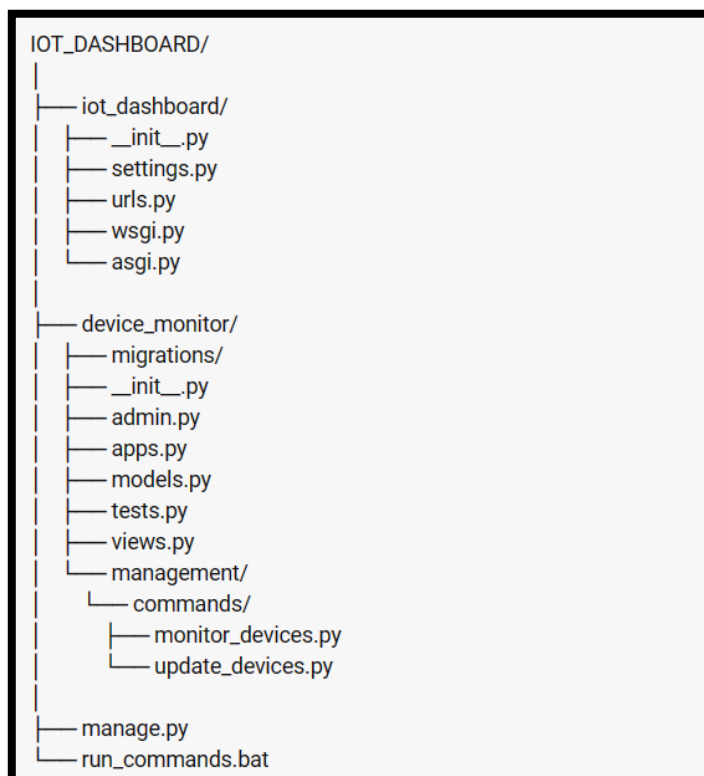
### **2.1. PROJECT SETUP**

#### **2.1.1. DJANGO PROJECT SETUP**

The creation of the Django project marks the starting point of the project and was accomplished through the following steps:

- The following command was executed in the terminal:  
"django-admin startproject iot\_dashboard".
- This command created a new Django project named iot\_dashboard.

The directory structure of the project is as follows:



The tasks of the created files are as follows:

- `iot_dashboard/`: Main project directory
- `__init__.py`: Marks this directory as a Python package.
- `settings.py`: Contains the project settings, including configurations such as database, templates, static files, etc.
- `urls.py`: Contains the URL configuration for the project, specifying which views are mapped to which URLs.
- `wsgi.py`: Determines how the project will be served with WSGI-compliant servers.
- `manage.py`: Command-line interface used to manage commands related to the Django project.

### 2.1.2. CREATING THE DJANGO APPLICATION

➤ The creation of the Django application was performed to provide the fundamental functionality of the project, following the steps below:

- The following command was executed in the terminal:

```
"cd iot_dashboard  
python manage.py startapp device_monitor"
```

➤ This command created a new Django application named `device_monitor`. Below, we describe the tasks of the generated files:

- **`device_monitor/`**: Application directory
- **`__init__.py`**: Marks this directory as a Python package.
- **`admin.py`**: Configures model registrations for the Django admin interface.
- **`apps.py`**: Contains the configuration of the application.
- **`models.py`**: Defines the database model.
- **`tests.py`**: Contains the application tests.
- **`views.py`**: Generates and presents web page views.
- The Django application is configured to provide the necessary model, views, and functionality for monitoring and managing IoT devices.

➤ Adding 'device\_monitor' to the `INSTALLED_APPS` section in the `settings.py` file allows Django to recognize this application and make it available. This allows the 'device\_monitor' application to be included in the project and makes its models, views, management interface and other features available.



```
INSTALLED_APPS = [  
    "django.contrib.admin",  
    "django.contrib.auth",  
    "django.contrib.contenttypes",  
    "django.contrib.sessions",  
    "django.contrib.messages",  
    "django.contrib.staticfiles",  
    'device_monitor'  
]
```

## 2.2. DATABASE MODELING AND MANAGEMENT

### 2.2.1. MODEL DEFINITION

Database modeling forms the foundation of the project and is necessary to store the characteristics of IoT devices. In this step, the Device model has been created. The relevant files and their tasks are explained below:

- **models.py (device\_monitor/):** This file contains Django model definitions. A class named Device is defined, which includes fields representing the properties of each IoT device.

- The models.py file is as follows:

```
device_monitor > models.py > ...
1  from django.db import models # Importing the models module from Django.
2
3  class Device(models.Model): # Defining a model named Device, derived from models.Model.
4      device_id = models.CharField(max_length=100, unique=True) # Creating a unique character field named device_id.
5      is_connected = models.BooleanField(default=False) # Creating a Boolean field named is_connected with a default value of False.
6      last_seen = models.DateTimeField(auto_now=True) # Creating a DateTime field named last_seen that automatically updates to the current time.
7      temperature = models.CharField(max_length=100, null=True, blank=True) # Creating an optional character field named temperature.
8      operational_status = models.BooleanField(default=False) # Creating a Boolean field named operational_status with a default value of False.
9      humidity = models.CharField(max_length=100, null=True, blank=True) # Creating an optional character field named humidity.
10
11  def __str__(self): # Defining a special method to determine the string representation of the model.
12      status = 'Connected' if self.is_connected else 'Disconnected' # Checking if the device is connected or disconnected.
13      return f"{self.device_id} - {status}, Temp: {self.temperature}, Operational: {'Yes' if self.operational_status else 'No'}, Humidity: {self.humidity}"
14  # Creating the string representation of the model, including device ID, connection status, temperature, operational status, and humidity.
```

- These fields are as follows:

- **device\_id:** A character field representing the unique identifier of the device.
- **is\_connected:** A boolean field indicating whether the device is connected or not.
- **last\_seen:** A date/time field representing the last time the device was seen.
- **temperature:** A character field to store temperature data of the device.
- **operational\_status:** A boolean field indicating the operational status of the device.
- **humidity:** A character field to store humidity data of the device.

This model defines the properties of each IoT device and enables them to be stored in the database.

### 2.2.2. ADMIN INTERFACE SETUP

The admin interface allows easy management of the Device model in the Django admin panel. The relevant file and configuration are explained below:

- **admin.py (device\_monitor/):** This file contains configurations for the Django admin interface. The Device model is registered with the `admin.site.register()` function and configured with the `DeviceAdmin` class. The `list_display` attribute specifies the fields used to list the devices in the admin panel.
- This configuration enables administrators to easily add, edit, and delete devices, as well as view a list of devices and their properties in the admin panel.

- The admin.py file is

```
device_monitor > admin.py > ...
1 from django.contrib import admin # Importing the admin module from Django.
2 from .models import Device # Importing the Device model from the current package.
3
4 @admin.register(Device) # Registering the Device model with the admin site.
5 class DeviceAdmin(admin.ModelAdmin): # Defining a custom admin class for the Device model.
6     list_display = ['device_id', 'is_connected', 'last_seen', 'temperature', 'operational_status', 'humidity']
7     # Defining the fields to be displayed in the admin list view for Device instances.
```

## 2.3. DATABASE MIGRATIONS AND MANAGEMENT

### 2.3.1. MIGRATIONS

- Database migrations are used to apply model changes to the database. In this step, how migrations are created and applied is explained below:
- The following commands were executed in the terminal:  
`python manage.py makemigrations`  
`python manage.py migrate`
- The makemigrations command detects model changes in the device\_monitor application and creates migration files. Then, the migrate command applies these migrations to the database.
- These steps ensure that changes made to the Device model are reflected in the database.

## 2.4. CUSTOM MANAGEMENT COMMANDS

### 2.4.1. CREATION OF CUSTOM MANAGEMENT COMMANDS

In the project, custom management commands have been developed to update and monitor the data of IoT devices. These are named `update\_devices.py` and `monitor\_devices.py`, providing the following functionality:

- The `update\_devices.py` code for this file is shown below:

```
device_monitor > management > commands > update_devices.py > ...
1 import random # Importing the random module to generate random data.
2 from django.core.management.base import BaseCommand # Importing BaseCommand class from django.core.management.base.
3 from device_monitor.models import Device # Importing the Device model from device_monitor app.
4 from django.utils import timezone # Importing timezone module from django.utils.
5 import time # Importing the time module.
6
7 class Command(BaseCommand): # Defining a custom management command named Command, derived from BaseCommand.
8     help = 'Updates device data periodically' # Help text for the management command.
9
10     def handle(self, *args, **options): # Defining the handle method to execute the command.
11
12         try: # Handling KeyboardInterrupt to stop the infinite loop gracefully.
13             while True: # Running indefinitely.
14                 self.update_devices() # Calling the update_devices method to update device data.
15                 time.sleep(2) # Waiting for 2 seconds before the next update.
16         except KeyboardInterrupt: # Handling KeyboardInterrupt when the user interrupts the command.
17             self.stdout.write("Stopping device data updates...") # Displaying a message indicating the command is stopped.
18
19     def update_devices(self): # Defining a method to update device data.
20         devices = Device.objects.all() # Retrieving all Device instances from the database.
21         for device in devices: # Iterating over each device.
22             device.is_connected = random.choice([True, False]) # Setting is_connected field randomly.
23             device.temperature = f"{random.randint(18, 35)}°C" # Generating a random temperature value.
24             device.operational_status = random.choice([True, False]) # Setting operational_status field randomly.
25             device.humidity = f"{random.randint(40, 60)}%" # Generating a random humidity value.
26             device.last_seen = timezone.now() # Updating the last_seen field with the current time.
27             device.save() # Saving the changes to the device instance in the database.
```



- **update\_devices.py File:** This file is used to periodically update the data of IoT devices.
  - The `random` module is used to generate random values.
  - The `BaseCommand` class is the base class for creating Django management commands.
  - The `Device` model is imported from the `device_monitor` application and used to update the data of devices.
  - The `timezone` module provides timezone support for time operations in Django.
  - The `time` module is a built-in module in Python used for time operations.
  - **handle method:** This method contains the main functionality of the management command. It initiates a loop and continuously calls the `update_devices` method. The loop terminates when a KeyboardInterrupt (Ctrl+C) combination is detected.
  - **update\_devices method:** This method takes all devices and updates their data by generating random values for each device. Changes are saved to the database using the `device.save()` method.

- The **monitor\_devices.py** code for this file is shown below:

```
device_monitor > management > commands > monitor_devices.py > Command > handle
1 from django.core.management.base import BaseCommand # Importing BaseCommand class from django.core.management.base.
2 from device_monitor.models import Device # Importing the Device model from device_monitor app.
3 import time # Importing the time module.
4
5 class Command(BaseCommand): # Defining a custom management command named Command, derived from BaseCommand.
6     help = 'Monitors the status of IoT devices' # Help text for the management command.
7
8     def handle(self, *args, **options): # Defining the handle method to execute the command.
9         print("Starting device monitoring...") # Displaying a message indicating the start of device monitoring.
10        try: # Handling KeyboardInterrupt to stop the infinite loop gracefully.
11            while True: # Running indefinitely.
12                devices = Device.objects.all() # Retrieving all Device instances from the database.
13                print(f"Time: {time.strftime('%X')}") # Printing the current time.
14                for device in devices: # Iterating over each device.
15                    # Printing the status of each device including device ID, connection status, operational status, temperature, and last seen time.
16                    print(f"Device ID: {device.device_id}, Status: {'Connected' if device.is_connected else 'Disconnected'},
17                          Operational Status: {'Working' if device.operational_status else 'Not Working'}, Temperature: {device.temperature}, Last Seen: {device.last_seen}")
18                time.sleep(1) # Waiting for 1 second before refreshing data.
19                print("-" * 40) # Printing a separator line.
20        except KeyboardInterrupt: # Handling KeyboardInterrupt when the user interrupts the command.
21            print("Monitoring stopped.") # Displaying a message indicating the monitoring is stopped.
```

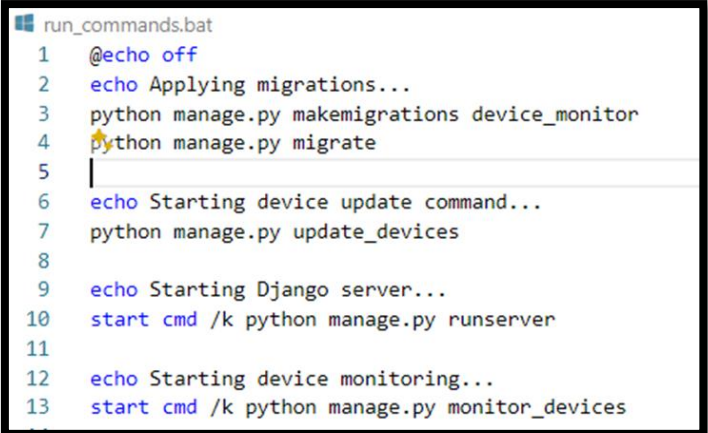
- **monitor\_devices.py File:** This file is used to monitor the status of IoT devices.
  - The `BaseCommand` class is the base class for creating Django management commands.
  - The `Device` model is imported from the `device_monitor` application and used to monitor the data of devices.
  - The `time` module is a built-in module in Python used for time operations.
  - **handle method:** This method contains the main functionality of the management command. It initiates a loop and continuously checks the status of all devices. The loop terminates when a KeyboardInterrupt (Ctrl+C) combination is detected.

These custom management commands, created in the `device_monitor/management/commands/` directory, provide essential functionality that enables the project to run automatically.

## 2.4.2. TESTING AND EXECUTION

To ensure the automatic execution of Django commands, we created a shell script named `run_commands.bat`. This script starts the Django server and executes our custom management commands.

The content of the script is as follows:

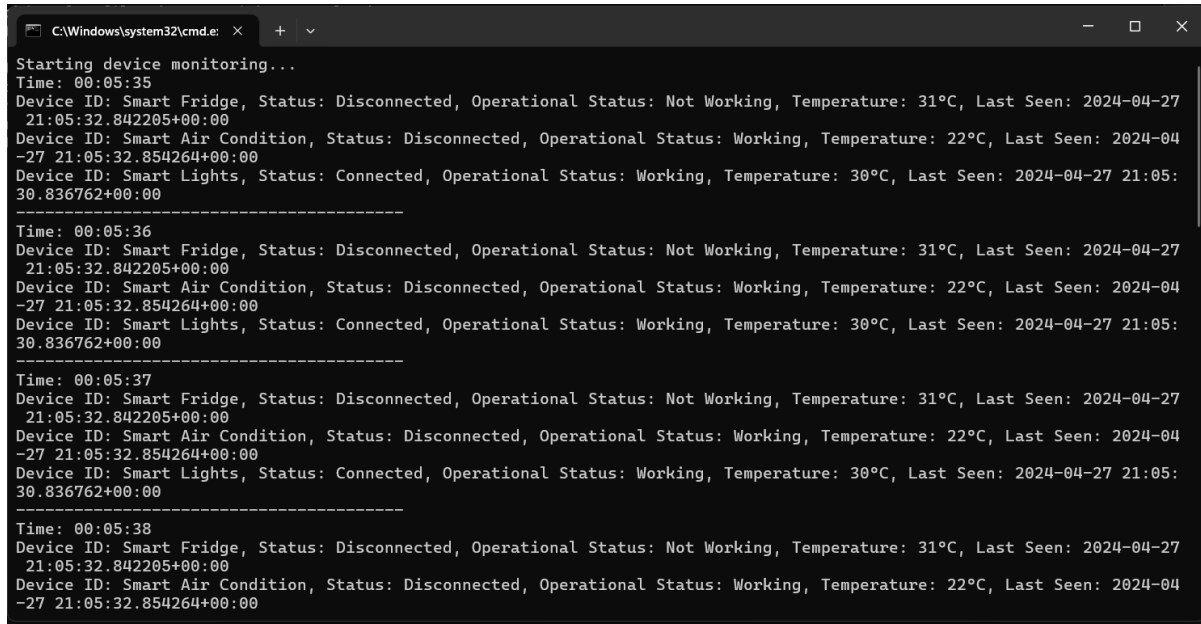


```
run_commands.bat
1  @echo off
2  echo Applying migrations...
3  python manage.py makemigrations device_monitor
4  python manage.py migrate
5
6  echo Starting device update command...
7  python manage.py update_devices
8
9  echo Starting Django server...
10 start cmd /k python manage.py runserver
11
12 echo Starting device monitoring...
13 start cmd /k python manage.py monitor_devices
```

### Script Description:

- **@echo off:** This command turns off the display of commands in the console window while the script is running. This way, only the output of the specified operations is displayed.
- **echo Applying migrations...:** This line prints "Applying migrations..." text to the console window. Here, migrations are created and applied to implement changes in the database.
- **python manage.py makemigrations device\_monitor:** This command detects model changes in the `device\_monitor` application of the Django project and creates a migration file.
- **python manage.py migrate:** This command applies the created migration files to implement changes in the database.
- **echo Starting device update command...:** This line prints "Starting device update command..." text to the console window. At this step, a custom management command for updating devices is initiated.
- **python manage.py update\_devices:** This command executes a custom management command named `update\_devices`, defined in the Django project. This command performs tasks such as updating devices or adding new devices to the database.
- **echo Starting Django server...:** This line prints "Starting Django server..." text to the console window. At this step, the Django server is started.
- **start cmd /k python manage.py runserver:** This command starts the `python manage.py runserver` command in the background by opening a new command prompt (cmd). This way, the Django server continues to run, and the terminal window is freed.
- **echo Starting device monitoring...:** This line prints "Starting device monitoring..." text to the console window. At this step, a custom management command for monitoring devices is initiated.
- **start cmd /k python manage.py monitor\_devices:** This command starts the `python manage.py monitor\_devices` command in the background by opening a new command prompt (cmd). This command is used to regularly monitor the status of devices.

- This script automates specific operations in the Django project, making the development process easier and more manageable.
- We made the script executable using the ``chmod +x run_commands.bat`` command and then executed it using the ``./run_commands.bat`` command. Ctrl+C tuşlarına basılarak tetiklenir.
- **The terminal screen is as follows:**



```
C:\Windows\system32\cmd.exe
Starting device monitoring...
Time: 00:05:35
Device ID: Smart Fridge, Status: Disconnected, Operational Status: Not Working, Temperature: 31°C, Last Seen: 2024-04-27 21:05:32.842205+00:00
Device ID: Smart Air Condition, Status: Disconnected, Operational Status: Working, Temperature: 22°C, Last Seen: 2024-04-27 21:05:32.854264+00:00
Device ID: Smart Lights, Status: Connected, Operational Status: Working, Temperature: 30°C, Last Seen: 2024-04-27 21:05:30.836762+00:00
-----
Time: 00:05:36
Device ID: Smart Fridge, Status: Disconnected, Operational Status: Not Working, Temperature: 31°C, Last Seen: 2024-04-27 21:05:32.842205+00:00
Device ID: Smart Air Condition, Status: Disconnected, Operational Status: Working, Temperature: 22°C, Last Seen: 2024-04-27 21:05:32.854264+00:00
Device ID: Smart Lights, Status: Connected, Operational Status: Working, Temperature: 30°C, Last Seen: 2024-04-27 21:05:30.836762+00:00
-----
Time: 00:05:37
Device ID: Smart Fridge, Status: Disconnected, Operational Status: Not Working, Temperature: 31°C, Last Seen: 2024-04-27 21:05:32.842205+00:00
Device ID: Smart Air Condition, Status: Disconnected, Operational Status: Working, Temperature: 22°C, Last Seen: 2024-04-27 21:05:32.854264+00:00
Device ID: Smart Lights, Status: Connected, Operational Status: Working, Temperature: 30°C, Last Seen: 2024-04-27 21:05:30.836762+00:00
-----
Time: 00:05:38
Device ID: Smart Fridge, Status: Disconnected, Operational Status: Not Working, Temperature: 31°C, Last Seen: 2024-04-27 21:05:32.842205+00:00
Device ID: Smart Air Condition, Status: Disconnected, Operational Status: Working, Temperature: 22°C, Last Seen: 2024-04-27 21:05:32.854264+00:00
```

- Finally, to verify the entire project configuration, we added device data through the Django admin panel and observed the updates and monitoring outputs in the terminal.

It outlines the steps to create a Django-based system for monitoring IoT devices. These steps include project setup, model definition, migration management, and implementation of functionality to periodically update and monitor device data.

### **3. PROPOSED APPROACH FOR THE PROJECT**

#### **3.1. TECHNOLOGY SELECTION**

The reasons for choosing Django in the project can be explained in more detail. Django accelerates the development process by providing a robust web framework, while also offering significant advantages such as security and scalability. Particularly, by using ORM (Object-Relational Mapping), database operations can be easily managed, and security measures can be implemented. Additionally, with the convenience Django provides, specific configurations of the project can be accomplished more comfortably.

#### **3.2. PROJECT STRUCTURE**

The project is created as a Django project named "iot\_dashboard." Within this project, there is an application named "device\_monitor." The application is configured to provide the model, view, and functionality necessary for monitoring and managing IoT devices. The project includes a mechanism that assigns random values at specified intervals when access to a particular device for live reading from sensors or the database is not available. This mechanism allows users to test the project without real-world scenarios and to verify functionality during the project development process.

#### **3.3. DATA UPDATE AND MONITORING**

Special management commands are used for real-time data update process. The "update\_devices.py" file comes into play when access to data from a device's sensor or the database is not available live. In such cases, random values are generated at specific intervals, and these values are assigned to device data. For example, new random value is generated and written to the database every 2 seconds within the specified value ranges for measurements like temperature and humidity. This continuous data update process enables real-time observation of data in the terminal interface. This method is useful for testing the system's functionality without real device data and plays a significant role in the project development process.

#### **3.4. EXECUTABLE COMMAND SCRIPT**

An executable command script named "run\_commands.bat" is created to automate the project's operation. This script starts the Django server, applies database migrations, and initiates the project by running special management commands.

#### **3.5. APPLICATION STRUCTURE**

The application is designed to allow users to interact via a simple terminal interface. Users can start the server, update device data, and monitor devices using simple commands. This ensures that the application is user-friendly and accessible.

## **4. RESULTS AND DISCUSSION**

In this section, we discuss the results of the terminal-based Django project for real-time monitoring of IoT devices and focus on the functionality, performance, user feedback, encountered challenges, and future development potential of the project.

### **4.1.FUNCTIONALITY**

The terminal application provides a successful solution for real-time monitoring of IoT devices. Users can directly access important data such as device connectivity, sensor readings (temperature, humidity), and operational status through the terminal. Additionally, the application allows users to easily start the server, update device data, and monitor devices using simple commands. In this project, a mechanism is provided to assign random values at specified data intervals for real-time monitoring of devices via a terminal application. This mechanism enables users to observe device data through a specific simulation without requiring access to a sensor or database for live device readings.

### **4.2. EFFICIENCY**

The performance of the application is satisfactory, and data updates occur at regular intervals. Django's management commands optimize data acquisition and processing, while the lightweight design of the application minimizes resource consumption.

### **4.3. USER FEEDBACK**

Initial user feedback is positive. Users appreciate the ease and simplicity of accessing real-time device data directly from the terminal. The clear and concise presentation of information and ease of use are particularly emphasized by users.

### **4.4. CHALLENGES ENCOUNTERED AND LESSONS LEARNED**

Various challenges were encountered during the development process, but they were successfully overcome through iterative testing and debugging processes. Challenges related to data management and command execution were overcome with comprehensive error handling mechanisms.

## 4.5. FUTURE DEVELOPMENTS AND ADDITIONS

Future developments for the terminal application may involve adding more complex scenarios to test the system under diverse conditions, such as fluctuating network connectivity, intermittent sensor failures, or varying environmental conditions. Enhancing data analysis and visualization capabilities could provide deeper insights from collected data, possibly through advanced analytics algorithms or integrating with data visualization libraries. Improving the user interface for a more intuitive experience, expanding device management functions for greater control over IoT devices, and simulating real-world scenarios, including anomalies and errors, could enhance the application's reliability and suitability. These enhancements could further improve the functionality and usability of the Django project for real-time IoT device monitoring.

## 5. CONCLUSION

This article has discussed the methodology, application details, and future development potential of a Django-based IoT project used to develop real-time monitoring systems. The project provides a web-based control panel aimed at monitoring the status of IoT devices. By utilizing data flow and visualization techniques, users can view important measurements such as device connectivity, sensor readings, and operational status in real-time.

The development process of the project has been explained step by step, with techniques and tools used at each step being thoroughly examined. The robust structure of Django and the opportunities it provides have been a significant advantage in the development process. In particular, Django's ORM system and management commands have accelerated development by facilitating and automating database operations.

The future development potential of the project is extensive. Adding more complex scenarios, enhancing data analysis and visualization capabilities, improving the user interface, and expanding device management functions could further strengthen the project. Additionally, simulating real-world scenarios and testing the performance of the application in various environments are crucial steps.

In conclusion, this project is a demonstration of an effective solution for real-time monitoring of IoT devices using Django and showcases the development of scalable and sustainable web applications. With further improvements, the project could be used in various industrial and home automation applications, contributing to the widespread adoption of IoT technology.

## REFERENCES

- <https://chat.openai.com>
- [IoT devices - what are they, who uses them and what are the risks \(itchronicles.com\)](https://itchronicles.com)
- [Real-Time Dashboard for IoT Device Management - Moments Log](#)
- [https://www.oracle.com/tr/internet-of-things/what-is-iot/#:~:text=Nesnelerin%20interneti%20\(IoT\)%2C%20internet,olan%20fiziksel%20nesnelerin%20ağı,nı%20açıklar.](https://www.oracle.com/tr/internet-of-things/what-is-iot/#:~:text=Nesnelerin%20interneti%20(IoT)%2C%20internet,olan%20fiziksel%20nesnelerin%20ağı,nı%20açıklar.)
- <https://webbylab.com/blog/iot-monitoring-dashboard-development-for-iot-devices/>
- <https://intellisoft.io/how-to-build-an-iot-dashboard-benefits-challenges-and-process/>
- <https://powerbi.microsoft.com/en-in/blog/using-power-bi-real-time-dashboards-to-display-iot-sensor-data-a-step-by-step-tutorial/>
- [How to generate random numbers in django - Stack Overflow](#)
- [Building a High-Performance Real-Time Analytics Platform with Django: A Comprehensive Guide | by Aditya Sharma | Medium](#)

(SIGNATURE)

Melike Kara  
200201013

## APPENDICES:

- 1- Inventory of Source Code (... pages)
- 2- Screenshots of Application GUI and/or Screen Outputs (... pages)

## APPENDICES

### 1. Inventory of Source Code (... pages)

- models.py

Defines Django model classes that represent database tables and define the data structure of the application.

```
device_monitor > models.py > ...
1  from django.db import models # Importing the models module from Django.
2
3  class Device(models.Model): # Defining a model named Device, derived from models.Model.
4      device_id = models.CharField(max_length=100, unique=True) # Creating a unique character field named device_id.
5      is_connected = models.BooleanField(default=False) # Creating a Boolean field named is_connected with a default value of False.
6      last_seen = models.DateTimeField(auto_now=True) # Creating a DateTime field named last_seen that automatically updates to the current time.
7      temperature = models.CharField(max_length=100, null=True, blank=True) # Creating an optional character field named temperature.
8      operational_status = models.BooleanField(default=False) # Creating a Boolean field named operational_status with a default value of False.
9      humidity = models.CharField(max_length=100, null=True, blank=True) # Creating an optional character field named humidity.
10
11  def __str__(self): # Defining a special method to determine the string representation of the model.
12      status = 'Connected' if self.is_connected else 'Disconnected' # Checking if the device is connected or disconnected.
13      return f'{self.device_id} - {status}, Temp: {self.temperature}, Operational: {'Yes' if self.operational_status else 'No'}, Humidity: {self.humidity}'
14  # Creating the string representation of the model, including device ID, connection status, temperature, operational status, and humidity.
```

- admin.py

Configures models to be displayed in the Django admin interface.

```
device_monitor > admin.py > ...
1  from django.contrib import admin # Importing the admin module from Django.
2  from .models import Device # Importing the Device model from the current package.
3
4  @admin.register(Device) # Registering the Device model with the admin site.
5  class DeviceAdmin(admin.ModelAdmin): # Defining a custom admin class for the Device model.
6      list_display = ['device_id', 'is_connected', 'last_seen', 'temperature', 'operational_status', 'humidity']
7  # Defining the fields to be displayed in the admin list view for Device instances.
```

- monitor\_devices.py

Contains a custom management command that regularly checks and updates the status of IoT devices.

```
device_monitor > management > commands > monitor_devices.py > Command > handle
1  from django.core.management.base import BaseCommand # Importing BaseCommand class from django.core.management.base.
2  from device_monitor.models import Device # Importing the Device model from device_monitor app.
3  import time # Importing the time module.
4
5  class Command(BaseCommand): # Defining a custom management command named Command, derived from BaseCommand.
6      help = 'Monitors the status of IoT devices' # Help text for the management command.
7
8  def handle(self, *args, **options): # Defining the handle method to execute the command.
9      print("Starting device monitoring...") # Displaying a message indicating the start of device monitoring.
10     try: # Handling KeyboardInterrupt to stop the infinite loop gracefully.
11         while True: # Running indefinitely.
12             devices = Device.objects.all() # Retrieving all Device instances from the database.
13             print(f"Time: {time.strftime('%X')}") # Printing the current time.
14             for device in devices: # Iterating over each device.
15                 # Printing the status of each device including device ID, connection status, operational status, temperature, and last seen time.
16                 print(f"Device ID: {device.device_id}, Status: {'Connected' if device.is_connected else 'Disconnected'},
17                     Operational Status: {'Working' if device.operational_status else 'Not Working'}, Temperature: {device.temperature}, Last Seen: {device.last_seen}")
18             time.sleep(1) # Waiting for 1 second before refreshing data.
19             print("-" * 40) # Printing a separator line.
20     except KeyboardInterrupt: # Handling KeyboardInterrupt when the user interrupts the command.
21         print("Monitoring stopped.") # Displaying a message indicating the monitoring is stopped.
```



- `update_devices.py`

Contains a custom management command used to update the data of IoT devices.

```
device_monitor > management > commands > update_devices.py > ...
1  import random # Importing the random module to generate random data.
2  from django.core.management.base import BaseCommand # Importing BaseCommand class from django.core.management.base.
3  from device_monitor.models import Device # Importing the Device model from device_monitor app.
4  from django.utils import timezone # Importing timezone module from django.utils.
5  import time # Importing the time module.
6
7  class Command(BaseCommand): # Defining a custom management command named Command, derived from BaseCommand.
8      help = 'Updates device data periodically' # Help text for the management command.
9
10     def handle(self, *args, **options): # Defining the handle method to execute the command.
11
12         try: # Handling KeyboardInterrupt to stop the infinite loop gracefully.
13             while True: # Running indefinitely.
14                 self.update_devices() # Calling the update_devices method to update device data.
15                 time.sleep(2) # Waiting for 2 seconds before the next update.
16         except KeyboardInterrupt: # Handling KeyboardInterrupt when the user interrupts the command.
17             self.stdout.write("Stopping device data updates...") # Displaying a message indicating the command is stopped.
18
19     def update_devices(self): # Defining a method to update device data.
20         devices = Device.objects.all() # Retrieving all Device instances from the database.
21         for device in devices: # Iterating over each device.
22             device.is_connected = random.choice([True, False]) # Setting is_connected field randomly.
23             device.temperature = f"{random.randint(18, 35)}°C" # Generating a random temperature value.
24             device.operational_status = random.choice([True, False]) # Setting operational_status field randomly.
25             device.humidity = f"{random.randint(40, 60)}%" # Generating a random humidity value.
26             device.last_seen = timezone.now() # Updating the last_seen field with the current time.
27             device.save() # Saving the changes to the device instance in the database.
28
```

- `run_commands.bat`

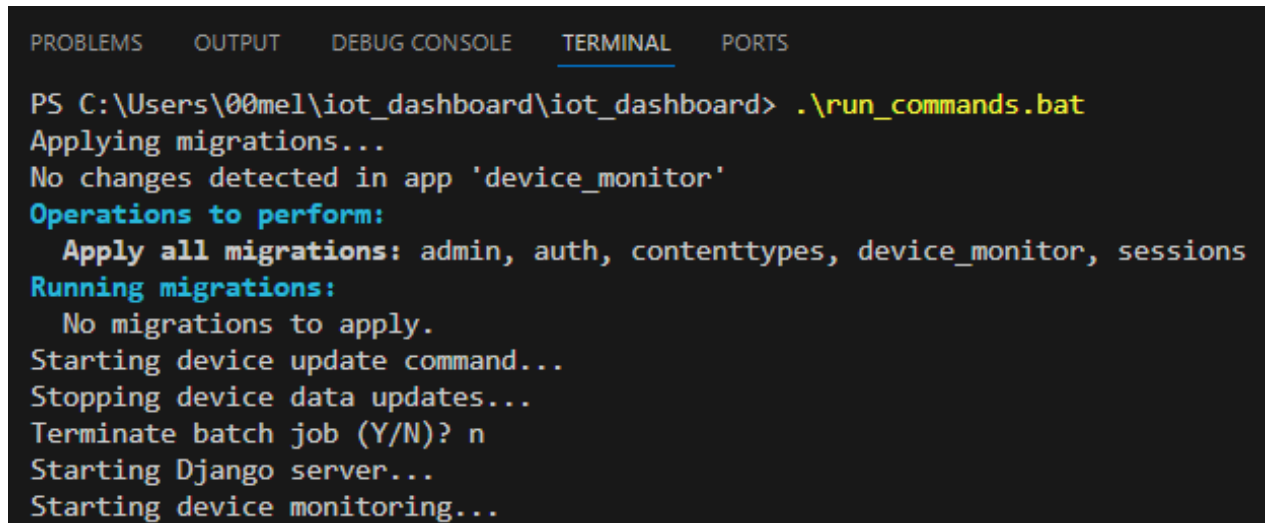
A Windows batch file that automates basic project operations.

```
run_commands.bat
1  @echo off
2  echo Applying migrations...
3  python manage.py makemigrations device_monitor
4  python manage.py migrate
5
6  echo Starting device update command...
7  python manage.py update_devices
8
9  echo Starting Django server...
10 start cmd /k python manage.py runserver
11
12 echo Starting device monitoring...
13 start cmd /k python manage.py monitor_devices
..
```

## 2- Screenshots of Application GUI and/or Screen Outputs (... pages)

### STEPS TO RUN THE PROJECT:

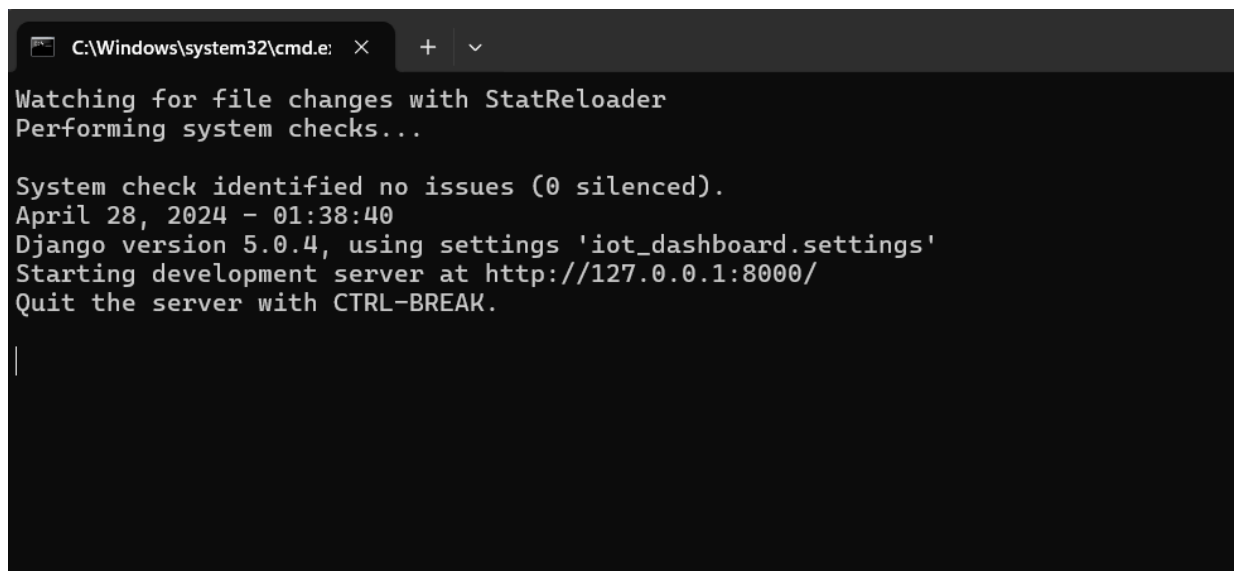
- Follow the steps below to run the script in terminal.
- Open a new terminal window in Visual Studio or your computer's command prompt or Windows powershell and navigate to the directory where the 'run\_commands.bat' file is located using the 'cd' command.
- Once you are in the correct directory, simply type `./run_commands.bat` and press **Enter** and **ctrl+c**.



```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS C:\Users\00mel\iot_dashboard\iot_dashboard> .\run_commands.bat
Applying migrations...
No changes detected in app 'device_monitor'
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, device_monitor, sessions
Running migrations:
  No migrations to apply.
Starting device update command...
Stopping device data updates...
Terminate batch job (Y/N)? n
Starting Django server...
Starting device monitoring...
```

- This will run the batch file, which will run all the necessary Python code.



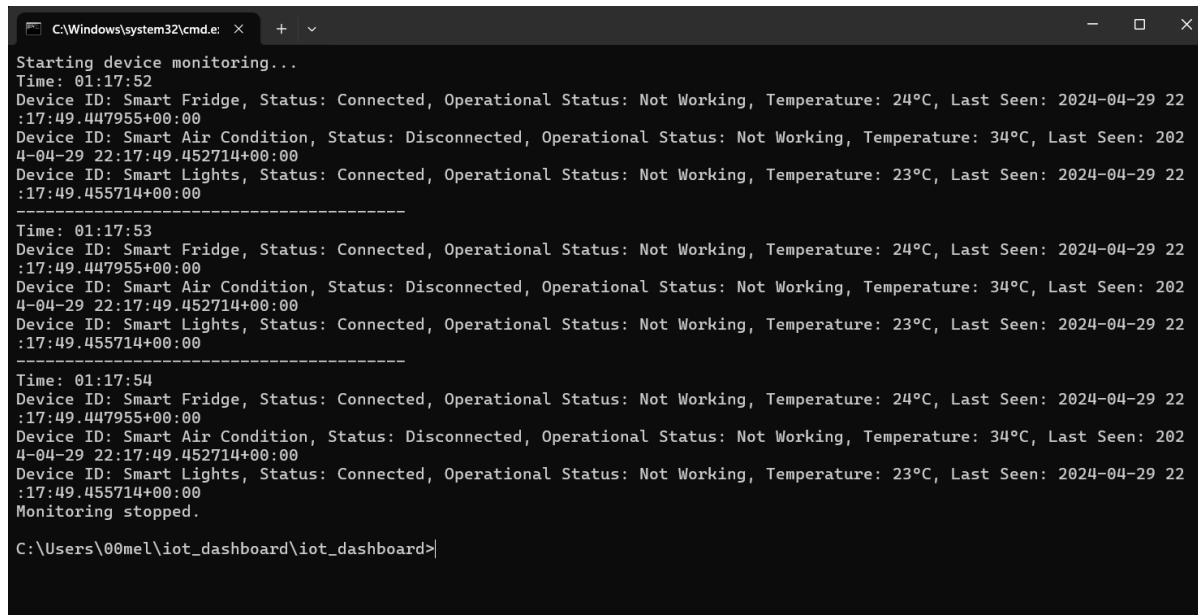
```
C:\Windows\system32\cmd.exe  X  +  v

Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).
April 28, 2024 - 01:38:40
Django version 5.0.4, using settings 'iot_dashboard.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.

|
```

- You will then see a live data stream in the terminal, displaying information about the devices in real time.



```
C:\Windows\system32\cmd.exe
Starting device monitoring...
Time: 01:17:52
Device ID: Smart Fridge, Status: Connected, Operational Status: Not Working, Temperature: 24°C, Last Seen: 2024-04-29 22:17:49.447955+00:00
Device ID: Smart Air Condition, Status: Disconnected, Operational Status: Not Working, Temperature: 34°C, Last Seen: 2024-04-29 22:17:49.452714+00:00
Device ID: Smart Lights, Status: Connected, Operational Status: Not Working, Temperature: 23°C, Last Seen: 2024-04-29 22:17:49.455714+00:00
-----
Time: 01:17:53
Device ID: Smart Fridge, Status: Connected, Operational Status: Not Working, Temperature: 24°C, Last Seen: 2024-04-29 22:17:49.447955+00:00
Device ID: Smart Air Condition, Status: Disconnected, Operational Status: Not Working, Temperature: 34°C, Last Seen: 2024-04-29 22:17:49.452714+00:00
Device ID: Smart Lights, Status: Connected, Operational Status: Not Working, Temperature: 23°C, Last Seen: 2024-04-29 22:17:49.455714+00:00
-----
Time: 01:17:54
Device ID: Smart Fridge, Status: Connected, Operational Status: Not Working, Temperature: 24°C, Last Seen: 2024-04-29 22:17:49.447955+00:00
Device ID: Smart Air Condition, Status: Disconnected, Operational Status: Not Working, Temperature: 34°C, Last Seen: 2024-04-29 22:17:49.452714+00:00
Device ID: Smart Lights, Status: Connected, Operational Status: Not Working, Temperature: 23°C, Last Seen: 2024-04-29 22:17:49.455714+00:00
Monitoring stopped.

C:\Users\00mel\iot_dashboard\iot_dashboard>
```

- If you need to stop tracking data, you can do so by pressing **Ctrl+C** in the terminal window. Writes “**Monitoring stopped.**” on the screen.

(SIGNATURE)

Melike Kara  
200201013