

# **1. SYSTEM IMPLEMENTATION AND DEVELOPMENT**

## Introduction

In this chapter, we explain the actual development and integration of the proposed IIoT-based monitoring and maintenance system. We divided the development into two main layers: edge data acquisition, and the Django-based web application. Each component played a key role in enabling real-time monitoring, fault reporting, and maintenance management.

### 1.1. Environment Setup

#### 1.1.1. virtual environment

In order to maintain a clean and organized development setup, we used a virtual environment to create an isolated Python environment for our project. This allowed us to manage our dependencies separately from other projects or the system-wide Python installation. It was especially useful since we were working on multiple tools that might require different package versions.

```
# we changed the directory to our project folder
cd capstone_project\
# We created a virtual environment named 'venv'
python -m venv venv
# This how we activate the virtual environment each time
venv\Scripts\activate
# Install Django
# We installed Django , Snap7 & Requirments packages
pip install django
pip install python-snap7
pip install requirements.txt -r
# When we're done, we deactivate the virtual environment with
deactivate
```

#### 1.1.2. Requirments

- psycopg2-binary
  - Why we need it: It allows Django to communicate with a PostgreSQL database.
  - Used for: ORM queries, migrations, database access.
- sqlparse
  - Why you need it: Used internally by Django to format and parse SQL statements.
  - Used for: Debugging SQL queries, Django's sqlmigrate command, and other internal formatting tasks.

## 1.2. User interface

Through our implementation process, we designed user-friendly interfaces for the technicians and the administrators to use while interacting with the CMMS system. The logic of every form and dashboard is governed by Django views and templates, dynamically rendered based on the user's role.

*For more comprehensive coverage, consult Appendix H*

### 1.2.1. Authentication

In this section, we designed a user-friendly authentication view as shown in Figure 1 to serve as the entry point to our web application. We implemented a simple and responsive interface that allows users to log in securely using their credentials. Through this view, we control access to the system and direct users to their respective dashboards based on their roles, whether they are technicians or administrators.

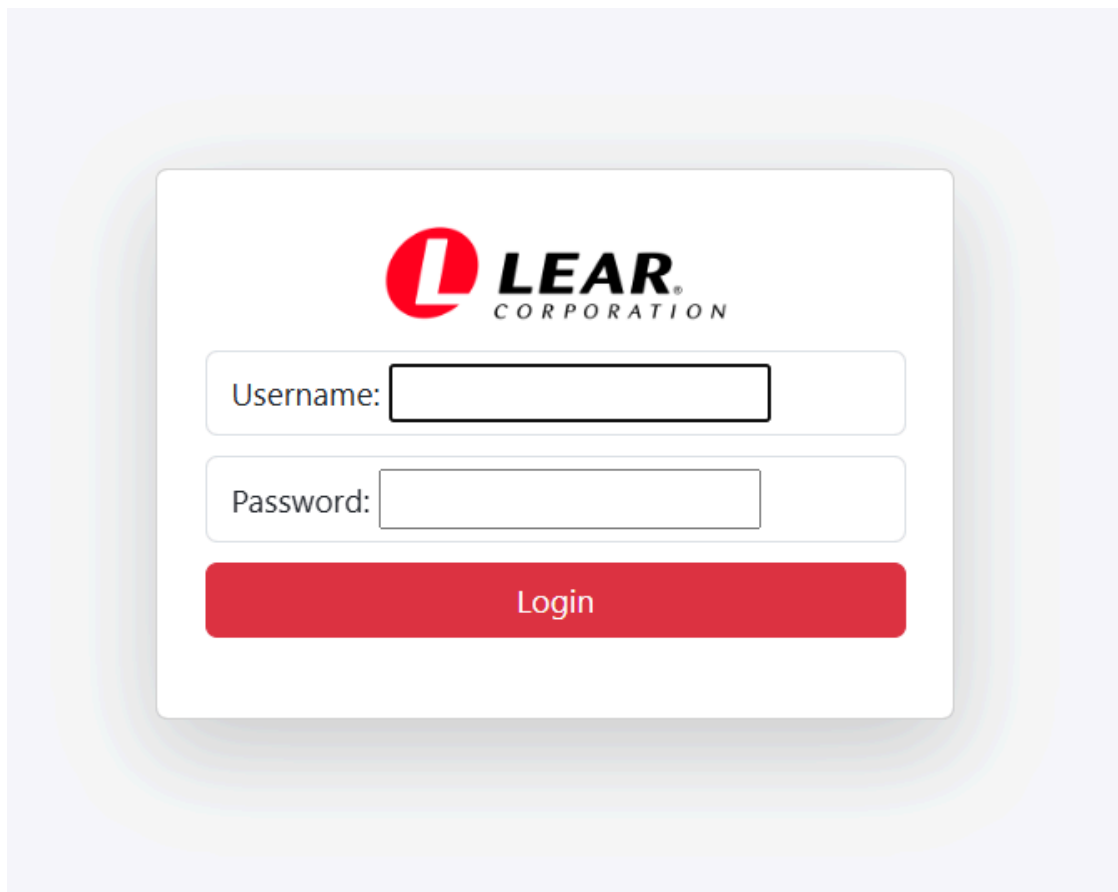
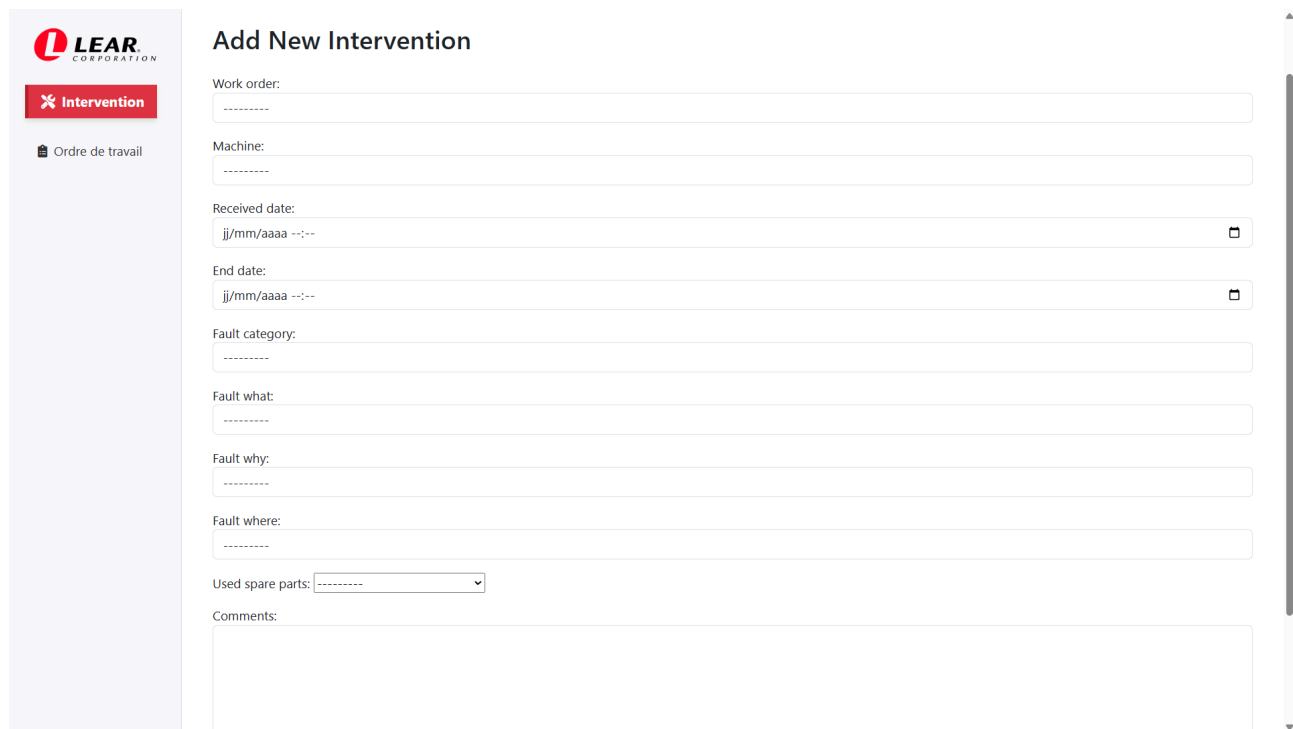


Figure 1: the Login page

### 1.2.2. Technicien Interface

We created a dedicated screen where technicians can submit intervention reports directly from their dashboard. This form is pre-filled with the technician's name (based on the logged-in user), and allows them to input key information such as fault category, start/end time, and select used spare parts from a dropdown list. Django Forms were used to generate and validate the form, with Bootstrap ensuring a clean and mobile-friendly layout. The folder structure of the

Figure 2 shows the intervention form interface used by technicians.



The screenshot displays the 'Add New Intervention' form. On the left is a sidebar with the LEAR CORPORATION logo and a menu containing 'Intervention' (highlighted with a red background) and 'Ordre de travail'. The main form area has the title 'Add New Intervention' and contains the following fields: 'Work order:' (text input), 'Machine:' (text input), 'Received date:' (date picker showing 'jj/mm/aaaa --:--'), 'End date:' (date picker showing 'jj/mm/aaaa --:--'), 'Fault category:' (text input), 'Fault what:' (text input), 'Fault why:' (text input), 'Fault where:' (text input), 'Used spare parts:' (dropdown menu), and 'Comments:' (text area). A vertical scrollbar is visible on the right side of the form.

Figure 2: The Technicien interface

### 1.2.3. Admin Interface

For administrators, we implemented a comprehensive dashboard that visualizes maintenance activity in real time. Using Django templates and JavaScript chart libraries `Chart.js`, we displayed various statistics, such as:

- Number of work orders by status (open, in progress, completed)
- Downtime by location or fault category
- Spare part usage trends

in the code below is a simplified snippet that show faults by machine:

```

new Chart(document.getElementById("timelineChart"), {
  type: 'line',
  data: {
    labels: data.timeline.labels,
    datasets: [{
      label: "Interventions Over Time",
      backgroundColor: "dc3545",
      borderColor: "dc3545",
      data: data.timeline.values,
      fill: true
    }]
  }
});

```

Cards and alert boxes were styled using Bootstrap’s “danger” theme to highlight urgent issues, with each card dynamically updated from the database. Figure 3 illustrates the administrator dashboard with visual indicators and real-time statistics. These dashboards help maintenance managers make data-driven decisions and quickly assess factory health.

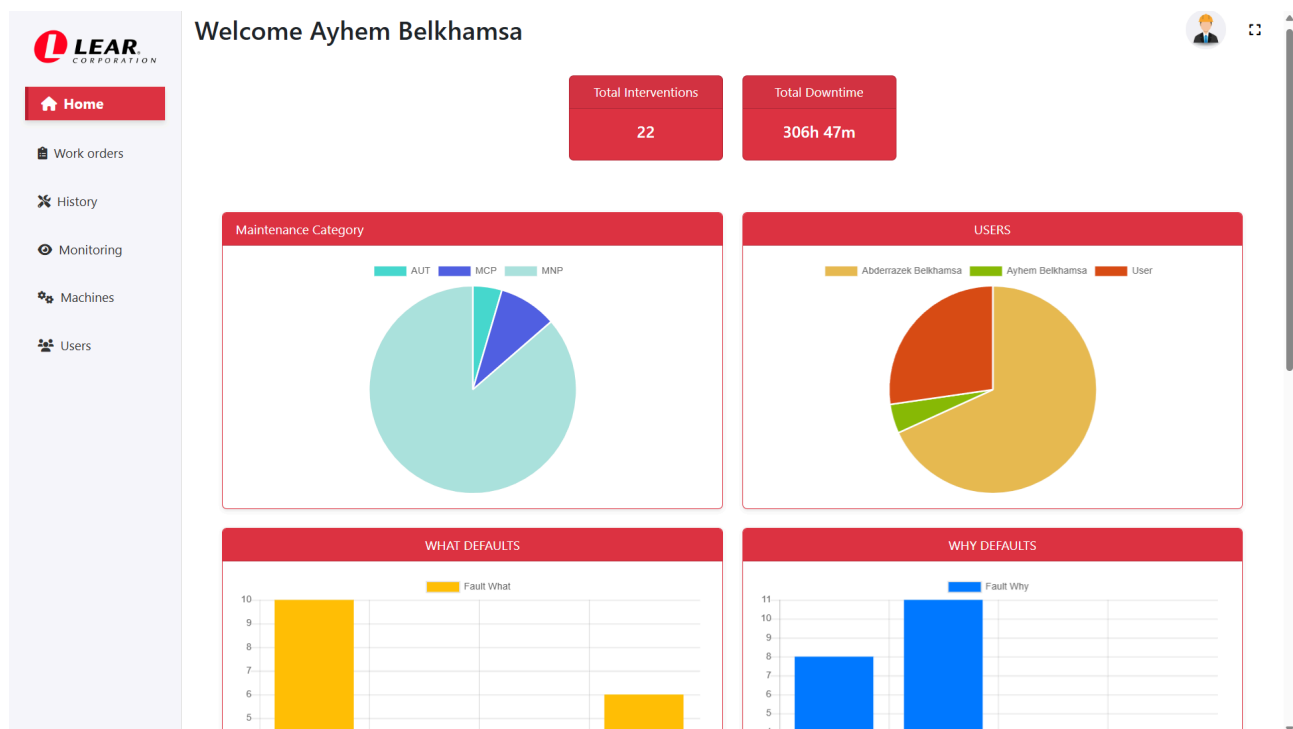


Figure 3: Admin's Dashboard

## Navigation

The navigation bar allows users to move between modules such as “Work Orders”, “Interventions”, and “Spare Parts”, with the interface adapting to their permissions. Figure 4 displays the navigation layout for an admin user.

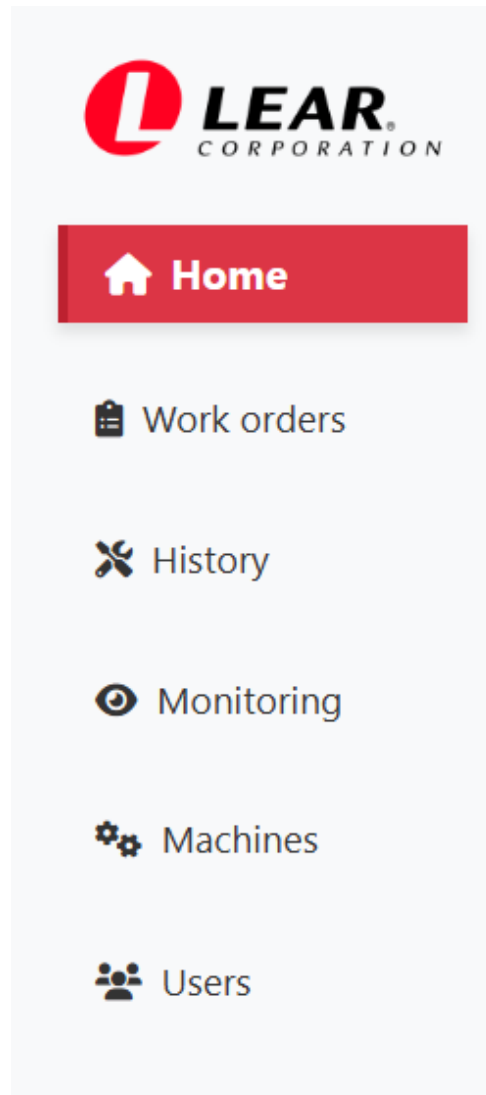


Figure 4: Admin's Navbar

## 1.3. Backend Development

The backend is the inner engine that handles the management of logic, processing of user requests, data transactions, and communication with the PLC. The goal was to develop a robust and modular backend capable of seamlessly integrating the database, user interface, and industrial automation system.

We crafted the backend using Django's MVT framework, in which we could effectively separate concerns while maintaining flexibility and scalability. This structure made it straightforward to deal with users, save intervention logs, manage session authentication, and incorporate external services like Snap7. we organized our project by separating concerns into distinct files and apps. This approach allowed us to scale features easily and maintain a clean codebase throughout the development process.

### 1.3.1. Apps

When building a Django project, the `startapp` command is used to create a new component (app) with its own structure (models, views, templates, etc.). Each app is designed to handle a specific piece of the overall system. In our case, we executed:

```
python manage.py startapp maintenance_app
```

This created a new directory with the necessary files:

- maintenance\_app
  - static/
  - templates/
  - views.py
  - urls.py
  - forms.py
  - models.py

### 1.3.2. Static files

In our system, static files played a central role in both the user dashboard and the administrative interface:

- CSS files, particularly through the use of Bootstrap, ensured visual coherence and responsive layout design across all screen sizes and devices.
- JavaScript files powered dynamic client-side behaviors such as chart rendering, modal pop-ups, and real-time UI updates.
- Images and icons served as intuitive visual indicators, helping users quickly interpret machine status and navigation elements.

### Organization and Best Practices

To maintain a scalable and modular structure, static assets were organized within each Django app's `static/` directory, with a central static directory configured for shared resources. This setup was declared explicitly in the `STATICFILES_DIRS` setting of the `settings.py` file.

Within the templates, static assets were integrated using Django's `{% load static %}` directive and referenced using `{% static '/img/figure_7.png' %}` for example. This approach ensured that files were properly resolved during development and deployment.

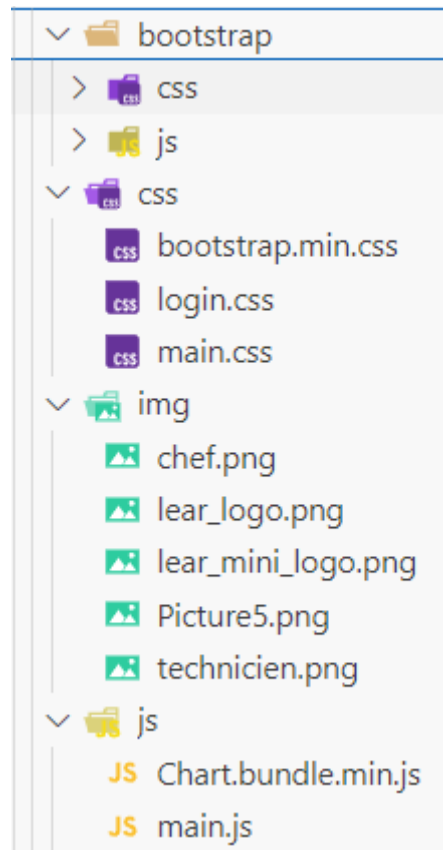


Figure 5: Static Folder

### 1.3.3. Templates

To achieve an easy-to-use interface, we relied on Django's template engine to render dynamic HTML based on data from the backend. Django templates allowed us to maintain a clean separation of the presentation and logic layers. We developed all the templates using Bootstrap elements.

We organized the templates into logically arranged sections to suit the user roles and application function. For instance, technicians follow a fault reporting form, and administrators are presented with a dashboard view containing intervention history and machine status.

The Figure 6 shows the templates folder

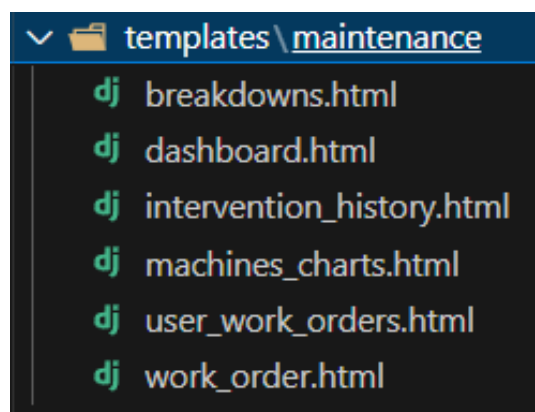


Figure 6: Templates folder



## Template Inheritance

We used a base template `admin_layout.html` to keep our structure consistent across pages: we code the main layout that is present in every page and then each page extends it like this:

```
{% extends "admin_layout.html" %}
{% block title %}Technician Dashboard{% endblock %}
{% block content %}
    #Current page content
{% endblock %}
```

### 1.3.4. views.py

Our `views.py` files handled the logic that connects the frontend with the database:

- For **technicians**, we developed views that render the intervention for and process submissions.
- For **admins**, we created views to list interventions, filter them by date or machine.

```
@login_required
def admin(request):
    return render(request, "users/admin.html")

def user_login(request): # I Renamed the function to avoid conflict with built-in
    login function
    if request.method == "POST":
        form = AuthenticationForm(data=request.POST)
        if form.is_valid():
            user = form.get_user()
            auth_login(request, user) # I Used auth_login instead of login
            if user.is_superuser:
                return redirect("users:admin")
            else:
                return redirect("users:users")
        else:
            form = AuthenticationForm()
    return render(request, "users/login.html", {"form": form})
```

### 1.3.5. urls.py

We organized the routes in `urls.py` to keep navigation and access clear:

```
from django.urls import path
from .views import
intervention_history, intervention_data, machines_charts, work_order, breakdowns, user_work_order,
app_name = 'maintenance'
urlpatterns = [
```

```

path('intervention-history/', intervention_history,
name='intervention_history'),
path('intervention-data/', intervention_data,name='intervention_data'),
path('machines_charts/', machines_charts, name='machines_charts'),
path('work_order/', work_order, name='work_order'),
path('breakdowns/', breakdowns, name='breakdowns'),
path('user_work-orders',user_work_order, name='user_work_order'),
path('add_intervention/', add_intervention, name='add_intervention'),
path('update-inputs/', update_inputs),
path('update-inputs/', update_inputs),
path('get-inputs/', get_inputs),
path('dashboard/', dashboard,name='dashboard'),
path('write-memory-bit/', write_memory_bit),]

```

### 1.3.6. forms.py

To validate forms and make user input easier, we utilized Django's pre-built forms module. This allowed us to rapidly develop safe and organized HTML forms and take advantage of Django's built-in validation support. One of the most important forms we developed was the reporting of machine faults form that enabled the technicians to report faults straightaway via the web interface. Below is an excerpt from `forms.py`, which defines the `InterventionForm`. This form maps directly to the `Intervention` model, ensuring data consistency and simplifying form rendering in templates:

```

class InterventionForm(forms.ModelForm):
    class Meta:
        model = Intervention
        fields = [
            "work_order",
            "Machine",
            "received_date",
            "end_date",
            "fault_category",
            "fault_what",
            "fault_why",
            "fault_where",
            "used_spare_parts",
            "comments",
        ]
        widgets = {
            "work_order": forms.Select(attrs=
{"class": "form-control", "readonly": True}),
            "received_date": forms.DateTimeInput(attrs=
{"type": "datetime-local", "class": "form-control"}),

```

```

        "end_date": forms.DateTimeInput(attrs=
        {"type": "datetime-local", "class": "form-control"}),
        "fault_category": forms.Select(attrs=
        {"class": "form-control"}),
        "fault_what": forms.Select(attrs=
        {"class": "form-control"}),
        "fault_why": forms.Select(attrs
        ={"class": "form-control"}),
        "fault_where": forms.Select(attrs=
        {"class": "form-control"}),
        "spare_parts": forms.Select(attrs={"class": "form-control"}),
        "Machine": forms.Select(attrs=
        {"class": "form-control"}),
        "comments": forms.Textarea(attrs=
        {"class": "form-control"}),
    }

```

### 1.3.7. Models.py

After designing the database schema, we implemented the models in Django using the `models.py` file. Below is the implementation of the Machine model, which plays a central role in tracking equipment status and managing fault reporting workflows.

```

class Machine(models.Model):
    STATUS_CHOICES = [
        ('Working', 'Working'),
        ('in_progress', 'In Progress'),
        ('fixed', 'Fixed Now'),
        ('not_fixed', 'Not Fixed'),
    ]
    name = models.CharField(max_length=255)
    # Unique identifier for the machine
    machine_id = models.IntegerField(unique=True)
    description = models.TextField(blank=True)
    status = models.CharField(max_length=20, choices=STATUS_CHOICES,
    default='default')

    def __str__(self):
        return f"{self.name} ({self.machine_id}) - {self.description}"

```

### 1.3.8. Communication with the Database (Django ORM)

Django uses an Object-Relational Mapper (ORM) to interact with the database. This means we write Python code, and Django handles the SQL behind the scenes as shown in Figure 7.

**How it works:**

Models (defined in `models.py`) are Python classes that represent tables in the database. So when we call `Machine.objects.all()` in a view, Django generates a SQL query like:

```
SELECT * FROM machine;
```

And when we save a form or create a new object Django converts that into an `INSERT INTO` SQL statement. This abstraction layer provides:

- Data validation via model field types.
- Referential integrity through ForeignKey and choices.
- Automatic migrations that evolve the database schema with model changes.

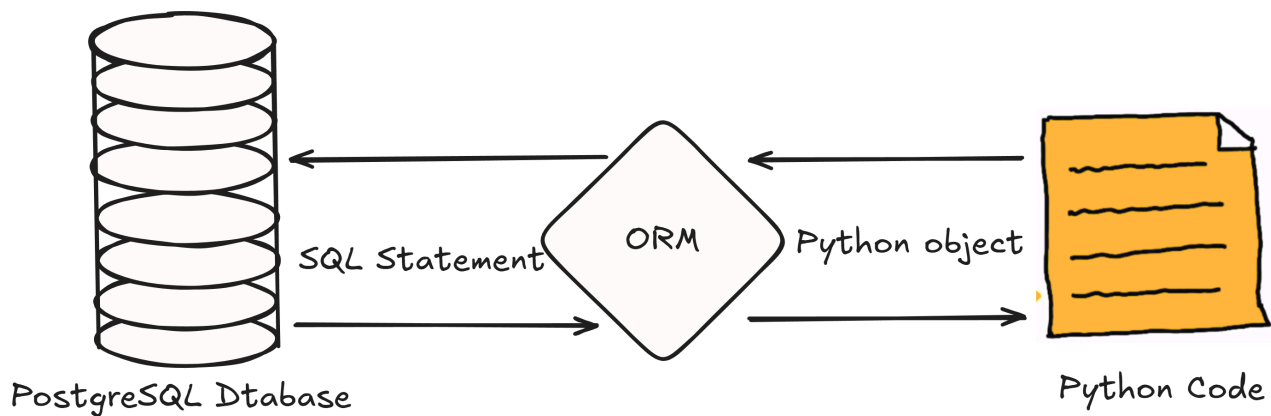


Figure 7: Django ORM

**1.4. Snap7 implementation**

To establish real-time communication between the web-based monitoring system and the PLC, we implemented the Snap7 client within the Django application.

The implementation followed a modular and fault-tolerant approach, allowing us to query the PLC's internal memory for machine statuses, operating states, and potential fault codes. These data points were then interpreted and recorded in the system's database, triggering corresponding updates on the user interface and generating appropriate maintenance records when needed. Refer to Figure 8 for a visual representation

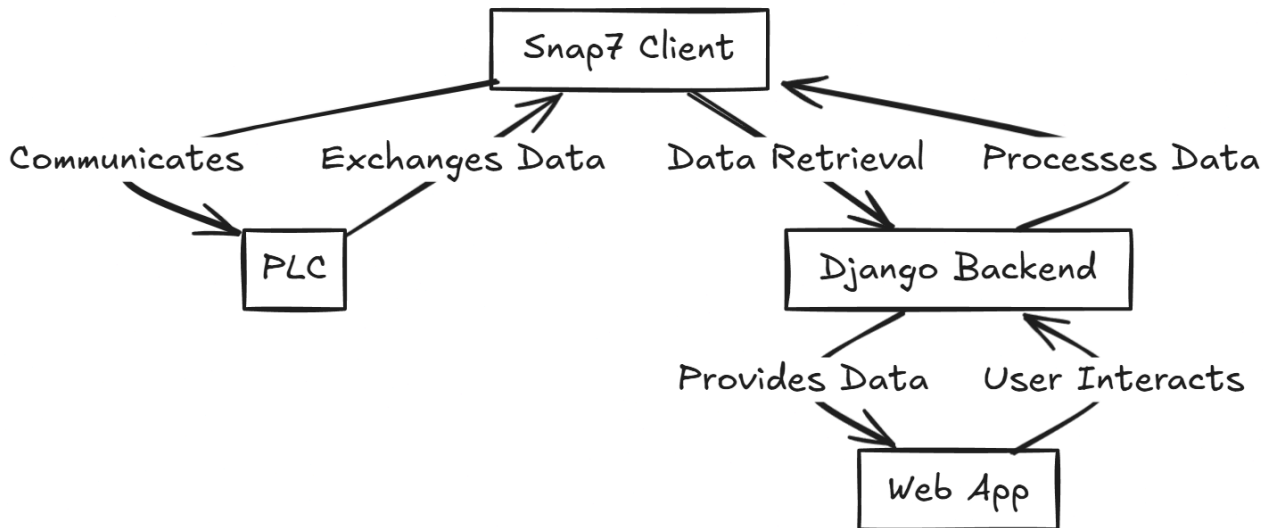


Figure 8: visual presentation of interaction with Snap7' client

#### 1.4.1. Plc configuration

On the Siemens side, we defined a dedicated DB to store machine status flags and error codes. These variables were updated automatically by the PLC's internal logic and served as the primary source of truth for the system's operational state. we also need to configure the DB as seen in Figure 9 and Figure 10

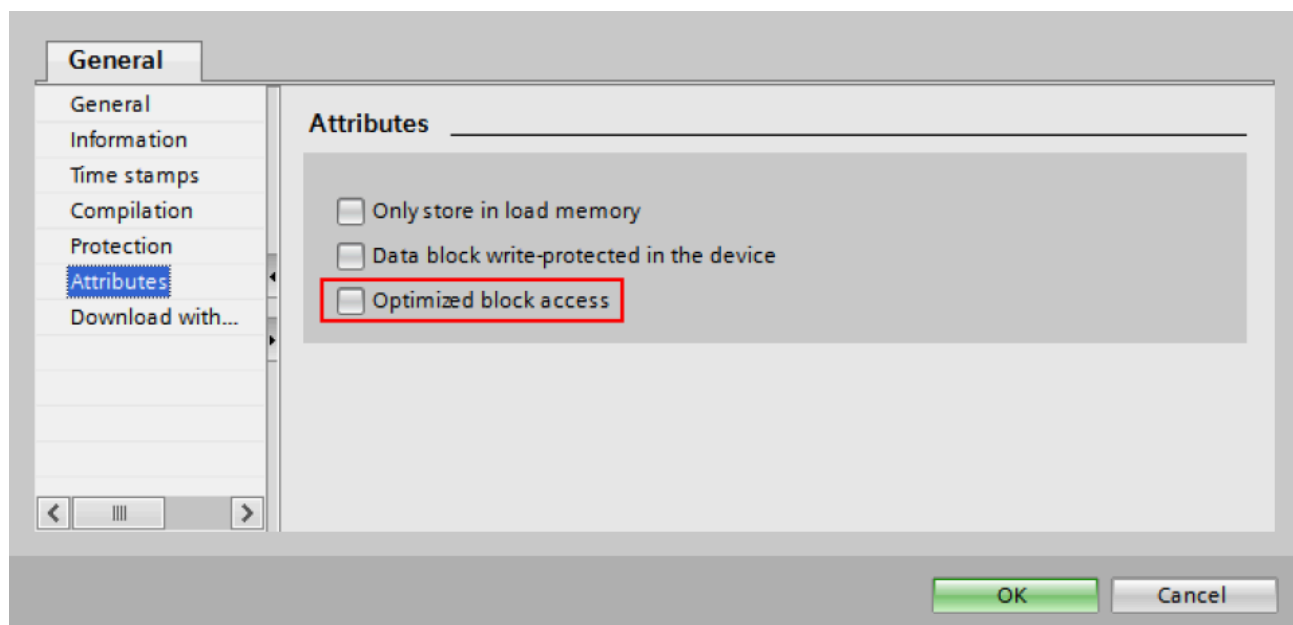


Figure 9: Disabling Optimized access

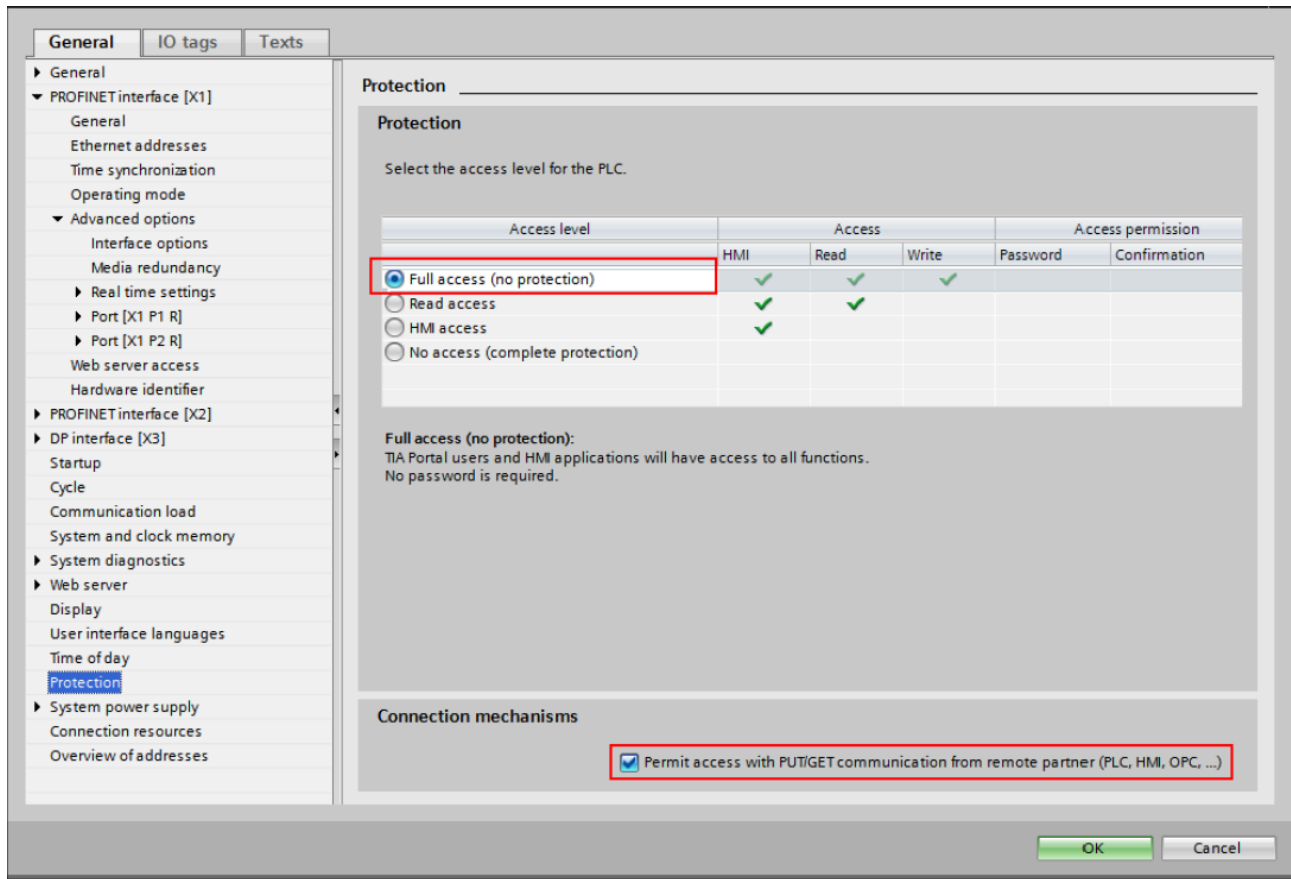


Figure 10: PUT/GET permission

### 1.4.2. Snap7 Client

After completing the configuration of the PLC, we proceeded with implementing the communication interface. We organized the communication logic to create a continuous client connection to it with Snap7's Client object. The client will periodically query specific memory addresses as implemented in the code below to read important machine parameters and fault flags.

```
def __init__(self, plc_ip="192.168.10.5", django_url="http://127.0.0.1:8000/update-
inputs/"):
    self.client = snap7.client.Client()
    self.plc_ip = plc_ip
    self.django_url = django_url
    self.running = False
    self.write_queue = Queue()
    self.inputs = {} # Live copy of current input states
```

### 1.4.3. Intergration to Django

We used a custom Python module in our Django app as demonstrated in the implementation below to facilitate Snap7 interactions. The module instantiates a client object which links the PLC through its IP address and related rack and slot setup. Methods, like `read_area()`, were wrapped to retrieve specific

data blocks or input/output addresses which correspond to machine statuses and fault signals. These methods were invoked by Django views or queued background jobs, based on the specific use case.

```
def update_inputs(request):  
    global last_inputs  
    if request.method == "POST":  
        data = json.loads(request.body)  
        last_inputs.update(data)  
        return JsonResponse({'status': 'updated'})  
    return JsonResponse({'error': 'Invalid request'}, status=400)
```

## 1.5. Intervention Logging and Machine Status Updates

In order to offer seamless maintenance tracking and accurate system monitoring, we created a structured workflow of storing intervention data and updating machine status

### 1.5.1. Workflow Description

Upon detection of a fault in a machine by a technician, the intervention form is opened through the dashboard interface. The form is rendered using Django's template engine and is supported by a Django ModelForm class associated with the Intervention model. When the form is submitted, the entered data is validated, and the new record is then saved to the PostgreSQL database. Refer to Figure 11 for a visual representation of the workflow

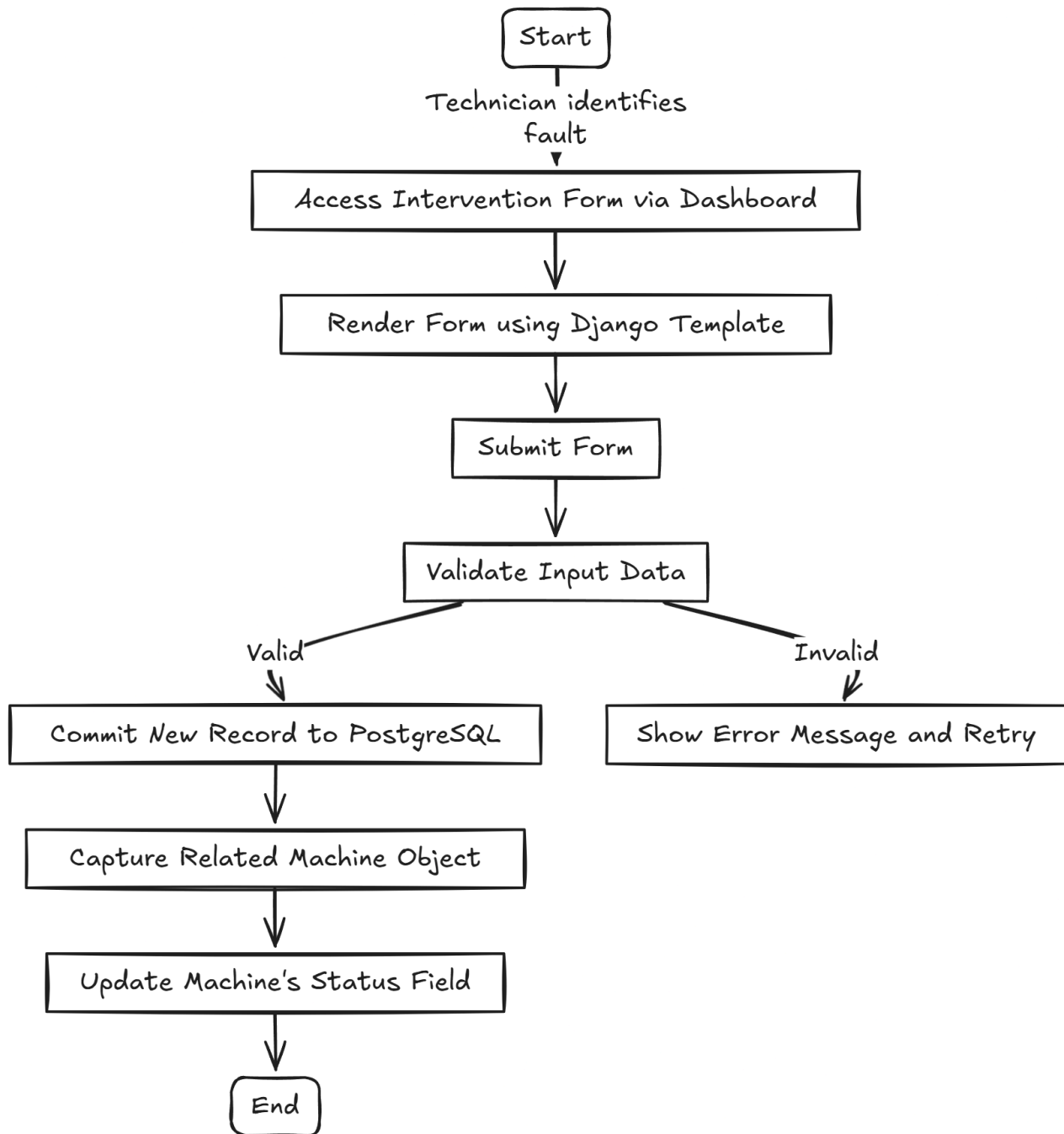


Figure 11: The workflow

### 1.5.2. Code Implementation Summary

The following is the backend logic utilized for storing interventions and the corresponding machine status update:

1. Form Submission: When the intervention form is submitted via POST, the Django view function processes the form.
2. Validation and Saving: If the form is valid, it creates a new instance of Intervention.



3. Status Update: Before the preservation of the intervention, the concerned machine object is retrieved and its status is updated. Database Commitment: The updated machine, together with the newly created intervention record, is retained in the database.

```
def add_intervention(request):
    work_order_id = request.GET.get('work_order_id')
    work_order = get_object_or_404(WorkOrder, pk=work_order_id)

    if request.method == 'POST':
        form = InterventionForm(request.POST)
        if form.is_valid():
            intervention = form.save(commit=False)
            intervention.technicien = request.user
            intervention.work_order = work_order
            intervention.work_order.status="in_progress"
            intervention.work_order.save()
            intervention.Machine.status = "in_progress"
            intervention.Machine.save()
            intervention.save()

            # Update machine status if needed
            if intervention.Machine and intervention.end_date:
                intervention.Machine.status = "fixed"
                intervention.work_order.status="fixed"
                intervention.work_order.save()
                intervention.Machine.save()

            return redirect('users:users') # Redirect after success
        else:
            form = InterventionForm(initial={'work_order': work_order})

    return render(request, 'users/users.html', {
        'form': form,
        'work_order': work_order,
    })
```

This implementation guarantees referential integrity between the record of intervention and the machine status. Moreover, it enables the administrator to see real-time information on the system dashboard, for example, how many issues are active and their corresponding status of handling.