Smart Building Educational Platform source code documentation

Andres Orellana

July 2025

1 Introduction and Installation Guide

1.1 Project Overview

This project is a modular, containerized educational platform for simulating and optimizing smart buildings. It allows users to modify EnergyPlus models (IDF files), convert them to FMU for co-simulation, train reinforcement learning agents, and visualize results using Grafana and Jupyter Notebooks. The entire architecture is designed for flexibility, reproducibility, and extensibility, making it ideal for both research and teaching purposes.

1.2 Project Structure

The system is composed of three major components:

- **Frontend (React)** – Provides an intuitive web interface for editing models, configuring simulations, and launching notebooks.
- **Backend (Flask/CherryPy)** – Exposes a RESTful API for managing IDF objects, converting to FMU, launching simulations, and controlling the environment.
- **Docker Services** – Include:
  - `InfluxDB` for time-series storage.
  - `Grafana` for visualization.
  - `Mosquitto (MQTT)` for device simulation and control.
  - `OpenHAB` for home automation integration.
  - Multiple Jupyter environments for simulation and training.

The entire project is orchestrated via Docker Compose, which ensures that all services are launched and connected correctly with a single command.

1.3 Installation Instructions

To install and launch the project on a development machine, follow the steps below.

1.3.1 1. Clone the repository

Clone the Git repository to your local machine:

```
git clone <repository-url>
cd smart-building-platform
```

1.3.2 2. Ensure Docker and Docker Compose are installed

You need Docker Engine and Docker Compose. On Ubuntu:

```
sudo apt update
sudo apt install docker.io docker-compose
sudo usermod -aG docker $USER
newgrp docker
```

1.3.3 3. Build the services

Run the following command to build all Docker images:

```
docker-compose build
```

1.3.4 4. Start the platform

Start all containers in the background (detached mode):

```
docker-compose up -d
```

1.3.5 5. Access the application

- **Frontend (React UI):** `http://localhost:80`
- **Backend (API):** `http://localhost:5000`
- **Grafana Dashboard:** `http://localhost:3000`
- **OpenHAB Interface:** `http://localhost:8080`
- **Jupyter Notebooks:** Various ports (e.g., 8887, 8889, 8890)

1.3.6 6. Stop the application

To stop all running containers:

```
docker-compose down
```

1.3.7 7. Clean up (optional)

To remove all containers, images, and volumes:

```
docker-compose down --rmi all --volumes
```

1.4 Developer Recommendations

- Ensure ports used by Docker services (80, 5000, 3000, 8080, etc.) are not blocked or already in use.
- When changing the frontend or backend, rebuild only the affected containers to save time:

  ```
  docker-compose build backend
  docker-compose up -d backend
  ```

- Use the provided `uploads/` directory to persist user files between sessions.

2 Backend Documentation

This section details the backend structure of the project, implemented using Flask, Flask-CORS, and SQLAlchemy, intended for developers aiming to expand or customize functionalities.

2.1 Backend Overview

The backend provides RESTful services to manage, edit, and convert IDF files (EnergyPlus input files), execute EnergyPlus simulations, and facilitate surrogate modeling. The backend uses Flask, Flask-CORS for cross-origin support, and SQLAlchemy for ORM database interactions.

2.2 Project Structure

- `config.py`: Application configurations, database URI, session management.
- `main.py`: Entry point, registers blueprints and initializes the Flask application.
- `methods.py`: Utility methods for IDF handling, EnergyPlus simulation, and data parsing.
- `models.py`: SQLAlchemy database models.
- `model_editor.py`, `model_converter.py`, `surrogate_model.py`: Blueprints.
- `inject_data.py`: Script to inject user-uploaded files into Jupyter Notebooks.

2.3 Initial Configuration (config.py)

The configuration file initializes Flask, database connections, and session handling:

- Database URI: SQLite (`smart_building_educational_platform.db`)
- Session management stored in filesystem (`flask_session`)
- Upload folder: `uploads/`

2.4 Blueprints

The backend is organized into Blueprints for modularity:

Existing Blueprints:

- **Model Editor** (`/model_editor`): IDF object management.
- **Model Converter** (`/model_converter`): Converts IDF files into FMU format, handles EnergyPlus simulations, and data export.
- **Surrogate Modeling** (`/surrogate_modeling`): Handles surrogate modeling tasks, providing JSON data about IDF models and simulations.

2.5 How to Add Support for New IDF Objects

To add support for new IDF objects:

1. Define the object structure in `models.py` using SQLAlchemy.
2. Create parsing methods in `methods.py` to read/write new IDF objects.
3. Add new Flask endpoints within the appropriate Blueprint (e.g., `model_editor.py`) for CRUD operations.
4. Ensure to update database migrations and apply them.

Example (models.py):

```
class idf_NewObject(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    attribute_1 = db.Column(db.String, nullable=False)
    attribute_2 = db.Column(db.String, nullable=False)

    def to_json(self):
        return {"id": self.id, "attribute1": self.attribute_1,
        "attribute2": self.attribute_2}
```

2.6 How to Create and Register New Blueprints

To add new blueprints:

1. Create a new Python file (e.g., `new_blueprint.py`).
2. Define a Blueprint and its endpoints.
3. Register the blueprint in `main.py`:

```
from new_blueprint import new_blueprint_bp
app.register_blueprint(new_blueprint_bp, url_prefix='/new_endpoint')
```

2.7 Adding New Methods

New functionalities should follow these guidelines:

- Implement reusable functions in `methods.py`.
- Include clear documentation and exception handling within each method.
- Integrate methods within relevant Blueprints.

Example method definition in `methods.py`:

```
def new_method(param1, param2):
    try:
        # Method logic here
        return result
    except Exception as e:
        current_app.logger.error(f"Error in new_method: {str(e)}")
        raise
```

2.8 Recommendations for Future Backend Expansion

- Maintain consistent naming and structure across files.
- Clearly document each endpoint and method functionality.
- Ensure thorough logging and error handling for debugging and maintainability.
- Regularly update documentation and database schemas alongside code changes.

3 Frontend Documentation

This section provides a comprehensive guide to the React-based frontend of the platform. It is intended for developers aiming to understand the architecture, expand the application, and integrate new features with the backend.

3.1 Overview

The frontend is developed in React using modern practices such as:

- React Router DOM for navigation.
- Context API for global state management.
- Bootstrap for styling and responsiveness.
- Axios and Fetch APIs for HTTP communication with the Flask backend.

The application is designed for modularity and extensibility, making it straightforward to add new object types, routes, pages, and modals.

3.2 Project Structure and Key Components

- `App.jsx` – Entry point where all routes are declared.
- `main.jsx` – Renders the root React component using `ReactDOM`.
- `AppContext.jsx` – Global context provider, manages state such as uploaded files, fetched IDF objects, and selected variables.
- `NavBar.jsx` – Top-level navigation menu linking to all pages.
- `FileUploadBar.jsx` – Handles IDF file uploads and triggers context updates.
- `ModelEditor.jsx` – UI for exploring, viewing, and adding EnergyPlus/EMS objects.
- `ModalComponent.jsx` – Dynamically renders a modal window with appropriate forms based on object type.
- `IdfObjectGrid.jsx` – Displays IDF objects in a structured view with support for deletion and editing.
- `ModelConverter.jsx` – Interface for converting IDF to FMU and generating the simulation config.
- `Simulator.jsx`, `Controller.jsx`, etc. – Pages for launching co-simulation Jupyter notebooks.
- `NotebookIframes.jsx` – Manages iframe rendering for each notebook.
- `SurrogateModeling.jsx`, `SurrogateObjectGrid.jsx` – Special pages for viewing surrogate modeling objects.

3.3 Routing and Adding New Pages

Routing is defined in `App.jsx` using `react-router-dom`. Each route renders a specific page component.

To add a new page:

1. Create a new component in `src/pages/` (e.g., `NewFeature.jsx`).
2. Add a `<Route>` in `App.jsx`:

   ```
   <Route path="/new_feature" element={<NewFeature />} />
   ```

3. Register the link in `NavBar.jsx` using:

   ```
   <Link to="/new_feature" className="nav-link">New Feature</Link>
   ```

3.4 Global State and AppContext

The `AppContext.jsx` defines and provides global variables such as:

- `uploadedFiles` – Tracks uploaded IDF and JSON files.
- `idfObjects` – Stores parsed IDF data from the backend.
- `selection`, `actionMinMax`, etc. – Used for configuring simulations.

To expand the AppContext:

1. Define a new value and setter using `useState()`.
2. Add the value and setter to the `AppContext.Provider value={}`.
3. Import and use it in any component via `useContext(AppContext)`.

3.5 Adding New IDF Object Modals

The modal system is centralized in `ModalComponent.jsx`. It renders different forms based on the selected object `type`.

To support a new object type:

1. Add the new `type` in `ModelEditor.jsx` inside the `gridTitles` object.
2. Extend `ModalComponent.jsx`:
   - Add a new entry in the `endpointMap`.
   - Add a new `case` in the `handleSubmit` switch statement to prepare the payload.
   - Render new form fields conditionally with `type === 'New:Object'`.
3. Ensure the backend has a matching endpoint (e.g., `/model_editor/add_new_object`).

Example endpoint mapping:

```
'EnergyManagementSystem:Sensor':
  'http://localhost:5000/model_editor/add_ems_sensor',
```

Example payload preparation:

```
case 'EnergyManagementSystem:Sensor':
  payload = {
    name,
    outputVariableOrOutputMeterIndexKeyName,
    outputVariableOrOutputMeterName
  };
  break;
```

3.6 Notebook Integration with Iframes

The `NotebookIframes.jsx` component dynamically displays embedded Jupyter notebooks depending on the current URL path. Each notebook corresponds to a simulation or modeling tool running on a specific port (e.g., 8889 for Surrogate Modeling).

To add a new notebook:

1. Register the new route in `App.jsx` and `NavBar.jsx`.
2. Add a conditional section to `NotebookIframes.jsx` that renders the iframe pointing to the correct Jupyter URL.

3.7 Surrogate Modeling Grid

The surrogate object view is managed in `SurrogateModeling.jsx` and rendered using `SurrogateObjectGrid.jsx`.

Each object type is displayed in a collapsible section and formatted using specific functions (e.g., `formatMaterial`, `formatConstruction`).

To support new surrogate types:

1. Add the object to the `surrogateTypes` array in `SurrogateModeling.jsx`.
2. Define a new formatting function in `SurrogateObjectGrid.jsx`.
3. Map the type key to the function in the `formatters` dictionary.

3.8 Best Practices for Expansion

- Reuse the Context API to minimize prop drilling.
- Keep all component logic inside dedicated files (e.g., use one file per modal).
- Align naming of frontend types and backend endpoints.

- Use hooks like `useEffect` to fetch data upon component mount.
- Ensure every backend integration has graceful error handling and feedback (e.g., `alert(result.message)`).

4 Docker Compose and Dockerfiles

This section provides a comprehensive overview of the Dockerized environment used in this project, detailing the exact configuration, including ports, services, dependencies, and volumes defined within the `docker-compose.yml` file.

4.1 Docker Compose

Docker Compose orchestrates multiple Docker containers, each providing a distinct service. Below are the precise definitions and mappings for each service:

Services in `docker-compose.yml`:

- **Frontend (React Application)**
  - Port Mapping: `80:80`
  - Dependencies: backend
- **Backend (Python Flask)**
  - Port Mapping: `5000:5000`
  - Dependencies: energyplus, eptofmu, influxdb, mosquitto, openhab
  - Environment variables include connection details for InfluxDB, MQTT, and OpenHAB.
  - Docker socket (`/var/run/docker.sock`) mounted for internal Docker interactions.
- **InfluxDB (Time-series Database)**
  - Port Mapping: `8086:8086`
  - Persistent data storage via Docker volumes.
  - Admin credentials: `admin/admin123`
  - Initializes database `base_building` at startup.
- **Grafana (Visualization Platform)**
  - Port Mapping: `3000:3000`
  - Linked directly to InfluxDB.
  - Admin credentials: `admin/admin`
- **Mosquitto (MQTT Broker)**

  - Port Mapping: `1883:1883`
  - Custom configuration file `mosquitto.conf` mounted.
  - Persistent storage for logs and data.

- **OpenHAB (Automation Server)**
  - Port Mapping: `8080:8080`, `8443:8443`
  - Persistent volumes for user data, configurations, persistence, and add-ons.

- **Jupyter Simulation Environments**
  - Jupyter Simulation: Port `8887:8888`
  - Jupyter Surrogate Modeling: Port `8889:8888`
  - Controlled Simulation: Port `8890:8888`
  - Base Simulation: Port `8891:8888`
  - Controller: Port `8892:8888`
  - Each has specific volumes mounted for notebooks, data, and EnergyPlus.

- **EnergyPlus**
  - No exposed port (service used internally).
  - Persistent volume with simulation outputs and uploads.

- **EPTOfmu (EnergyPlus to FMU converter)**
  - No exposed port (internal service).
  - Persistent volumes for uploads and user files.

- **Subscribers (MQTT to InfluxDB connectors)**
  - No exposed ports (internal MQTT subscribers).
  - Topics: `simulationRL/#` and `simulationBase/#`.
  - Connected directly to Mosquitto and InfluxDB.

Networks and Volumes:
- Defined internal Docker network for service intercommunication. - Persistent volumes for:

- Grafana data (`grafana_data`)
- OpenHAB userdata (`openhab_userdata`)
- EnergyPlus data (`energyplus_data`)

4.2 Dockerfiles

Each Docker service has a corresponding Dockerfile that defines the runtime environment, dependencies, and startup behavior:

- **Frontend Dockerfile**
  - Node.js base image (`node:lts-alpine`).
  - React build served by a static web server (usually on port 80).
- **Backend Dockerfile**
  - Python image (`python:3.x-slim`), installs from `requirements.txt`.
  - Flask or CherryPy REST API served on port 5000.
- **EPTOfmu Dockerfile**
  - Customized image to convert EnergyPlus IDFs to FMUs.
  - Utilizes a Python-based environment with specific dependencies.
- **Subscriber Dockerfiles**
  - Python-based MQTT clients pushing data to InfluxDB.
  - Environment variables configure MQTT and InfluxDB connectivity.
- **Jupyter Dockerfile (Unified Jupyter Environment)**
  - Customized Jupyter setup with EnergyPlus support and notebooks.
  - Conda environments built from a specific Dockerfile for simulations.

4.3 Deployment Instructions

To deploy the complete project, perform these steps:

1. Clone the repository and navigate to the project folder:

   ```
   git clone <repo_url>
   cd project
   ```

2. Build the images from the Dockerfiles:

   ```
   docker-compose build
   ```

3. Run all services:

   ```
   docker-compose up -d
   ```

4. Verify the running containers:

   ```
   docker-compose ps
   ```

4.4 Common Docker Management Commands

- Checking logs:

```
docker-compose logs <service_name>
```

- Stop all services:

```
docker-compose stop
```

- Remove all services and volumes (complete reset):

```
docker-compose down -v --rmi all
```

4.5 Recommendations for Future Developers

- Maintain and update Dockerfiles clearly and document new environment variables or configurations. - Ensure persistent volumes are consistently used to prevent data loss. - Clearly define new service dependencies within the `docker-compose.yml`. - Regularly update the documentation whenever Docker configurations change.