INSTRUCTIONS TO THE COMPETITOR

Module C – REST API

In this module, you must develop a backend REST API that provides live monitoring and control capabilities for an offshore wind farm. You are tasked with building an API for use by a frontend application (Module D).

The backend must consume turbine data from an external API, process it, cache it, and provide a clean and frontend-ready version. Additionally, it must support alerts, turbine control, and assigning roles.

Competitor Information

- The backend will be tested using HTTP clients and a test suite that will interact with your API.
  - Note: The test suite will be provided in the competitor handout.
- A mock server that simulates the external API is provided.
  - Note: The competitor handout will include the mock server URL and details.
- The API must be secure and reject unauthorized requests.
- The backend must provide the API specified in the provided OpenAPI specification.
- The use of a relational database is required; an in-memory or file-based store is not acceptable.
- The backend must implement all domain logic as described.

Scenario

You are building the backend for a wind farm off the coast of Denmark. A third-party system provides raw turbine data via an external API. Your backend acts as a middle layer: it fetches, validates, caches, interprets, and serves this data to a frontend. You also expose control, alert, and role-assigning functionality through your own API.

Requirements

1. Authentication and Access Control

Roles:

The API must support three roles with different access levels:

- anonymous: no access to protected endpoints. No authentication required.
- operator: read + control + acknowledge alerts. Requires authentication.
- admin: full access, including assigning roles. Requires authentication.

Login: POST `/auth/login`

- Input: `{ "username": string, "password": string }`
- Output: `{ "token": string, "role": string }`
- Validate token for subsequent requests
  - Validate token on each request to be in the `Authorization` header as `Bearer <token>`
  - The token string must sufficiently sophisticate (≥32 characters) to prevent brute-force attacks (i.e. don't just use the username as token)

Test Users:

You must provide test users for authentication.

See the **Database** section below for details.

Role Access:

- Public (no auth required) endpoints:
  - POST `/auth/login` — authenticate user and return token
  - GET `/turbines` — read all turbines (id, name, location, status)
  - GET `/turbines/:id/status` — read turbine status
  - GET `/turbines/:id/actions` — read triggered actions for a turbine
- Protected endpoints (require auth & role):
  - GET `/alerts` — list active alerts
  - POST `/alerts/:id/ack` — acknowledge an alert
  - POST `/turbines/:id/control` — control turbine pitch and yaw
  - POST `/turbines/:id/start` — start turbine
  - POST `/turbines/:id/shutdown` — shutdown turbine
  - POST `/turbines/:id/maintenance` — enter maintenance mode
  - GET `/turbines/:id/logs` — get turbine logs
  - POST `/auth/assign-role` (admin only)

2. External API Integration

Turbine status is not stored locally. You must fetch live data from an external API when the frontend requests it.

You do not need to implement a polling mechanism; the frontend will request data as needed.

External API Info:

Two URLs are provided during the competition:

- Mock API base URL – use this in code for all programmatic calls: GET `/turbines`, GET `/turbines/:id/logs`, POST `/turbines/:id/control`, etc.
- Control-panel URL - a browser-only UI for manually simulating scenarios (empty data, partial data, errors). Do not call it from your backend.

Mock API Details:

- Authentication: Send `Authorization: Bearer <token>` in headers (token will be provided during the competition).
- Endpoint: GET `/turbines`
- Returns:

```
{
    "timestamp": "2025-06-21T10:12:00Z",
    "data": [
        {
            "id": 1,
            "name": "Turbine A1",
            "location": {
                "lat": 56.4501,
                "lng": 8.3465
            },
            "rpm": 47,
            "powerMw": 1.9,
            "yaw": 270,
            "pitch": 22,
            "temperature": 35.2,
            "status": "started"
        },
        ...
    ]
}
```

An OpenAPI specification will be provided during the competition.

It would not make sense to work with the real external API during the competition, you will be provided with a mock server that simulates the external API. This mock server also includes a control panel where you can simulate scenarios, such as empty data, partial data, missing properties or errors.

Each turbine has the following properties.

Property	Type	Description
id	Integer	Unique turbine identifier
name	string	Turbine name
location	object	`{ lat: float, lng: float }`
rpm	integer	Rotations per minute (dynamic)
powerMw	float	Power output in megawatts (dynamic)
yaw	integer	Yaw angle in degrees (0–360)
pitch	integer	Blade pitch angle in degrees (-90–90)
temperature	float	Temperature in °C (dynamic)
status	string	One of: `"started"`, `"maintenance"`, `"shutdown"`

Special Cases:

The external API may return:

- A timestamp indicating when the data was last updated.
- Empty data: `"data": []`
- Partial turbines: `"data": [{...}, {...}]` (some turbines may be missing)
- Partial properties: Some turbines may not have all properties filled in (e.g., `temperature` may be `null`).
- Errors (e.g. network timeouts or 500s)

You must handle these cases gracefully:

- The response you serve to the frontend must always include a `freshness` field for each turbine and also for each turbine property, indicating whether the data is `live`, `cached`, or `missing`.
- The response must also include a `lastUpdated` timestamp for each turbine and each property, indicating when it was last updated.

3. Alerts

You must implement the specified alert rules. The rules are evaluated on fetch of the turbine data from the external API (no polling required).

Alerts must be stored in a database and can be acknowledged by the operator. Your logic must deduplicate alerts and ensure that each alert is only triggered once per turbine. A new alert can only be triggered if the previous alert is resolved (the rule was evaluated to `false`).

For now, only one alert needs to be implemented:

- High RPM Alert: If `rpm` exceeds 60, trigger an alert.

Alert status:

- Firing state: `firing` (active alert) or `resolved` (alert rule is no longer true)
- Acknowledged state: `acknowledged` (alert has been acknowledged by the operator) or `unacknowledged` (alert has not been acknowledged)

The alerts can be retrieved via endpoint GET `/alerts` and acknowledged via POST `/alerts/:id/ack`.

4. Turbine Control

Control actions:

- Set pitch and yaw angles via POST `/turbines/:id/control`
    - Input: `{ "pitch": integer, "yaw": integer }`
    - Valid ranges: `pitch: -90 to 90`, `yaw: 0 to 360`
    - Response: `{ "status": "success" }` or error message
- State transitions with the turbine: POST `/turbines/:id/:action` (`action` can be `start`, `shutdown`, or `maintenance`)
    - Response: `{ "status": "success" }` or error message

- The transition must be valid based on the current turbine state (see state transitions below).

Allowed Transitions:

1. Started → Shutdown: Shutdown, e.g., for low wind or grid issues
2. Shutdown → Started: Restart after shutdown
3. Shutdown → Maintenance: Maintenance can only be performed during downtime
4. Maintenance → Shutdown: Maintenance completed, turbine shuts down, making it ready for restart

The user actions must be saved in the database and can be retrieved via the unauthenticated endpoint `GET /turbines/:id/actions`.

- Each action includes a timestamp and the user who performed the action.
- The action type is one of `control`, `start`, `shutdown`, `maintenance`.
- A `control` action also includes the pitch and yaw values.

5. Turbine Logs

You must implement a logging system for each turbine. The logs can be retrieved from the external API via the endpoint `GET /turbines/:id/logs` which returns a `plaintext` response with the following format:

```
2025-06-21T10:12:00Z [Info] Turbine started using config /etc/turbine-a1.json
2025-06-21T10:15:00Z [Warning] Turbine start delayed due to low wind conditions
2025-06-21T10:20:00Z [Error] Turbine lost satellite fallback connection. Details:
multi line error message belonging to the sensor failure
second line of the error message still belongs to the sensor failure
2025-06-21T10:25:00Z [Info] Maintenance mode activated
```

The logs must be parsed to include:

- Turbine ID: not embedded in each log line; use the `:id` path parameter from the request
- Timestamp: parsed from the log line
- Log level: extracted from the log line and normalized to lowercase (`info`, `warning`, `error`)
- Message with new lines preserved: the rest of the log line after the timestamp and level until a new line and another log line with timestamp, level, and message starts

You will have to develop a parser that converts the plaintext logs and stores them in a structured format in the database.

The external API will return the last 1000 log entries for a turbine, and you must ensure you do not store duplicate logs based on the timestamp and message. Logs are only fetched when the frontend requests them, so you do not need to implement a polling mechanism. The external log endpoint can

be unavailable or return an error, in which case you must handle this gracefully and return the cached logs if available.

The endpoint to retrieve the logs is `GET /turbines/:id/logs`. The frontend will use this endpoint to display the logs in a user-friendly format. The sort order of the logs must be from oldest to newest, and only the newest 1000 logs must be returned at a time. No pagination is required.

The frontend also requires a search functionality to filter logs by level and message substring.

These shall be implemented in the backend as query parameters:

- `levels`: Filter logs by log level (e.g., `info`, `warning`, `error`). A comma separated list of levels can be provided.
- `message`: Filter logs by a substring in the message

6. Role Management

You must implement a role management system that allows the admin to assign roles to users. The admin can assign roles to users via the endpoint `POST /auth/assign-role`. The request must include the username and the role to assign.

A user with admin role cannot remove their own admin role.

7. Database

You are free to design the database schema as you see fit.

You must provide a SQL dump file with both the **structure** and **initial data** to seed the database.

It must be committed to the Git repository in the root directory as `seed.sql`.

The SQL dump must include the following data:

- user without role: `username: user`, `password: user12345`
- operator: `username: bob`, `password: bob12345`
- admin: `username: alice`, `password: alice12345`

The `seed.sql` file must not contain the plaintext passwords, but rather the hashed passwords.

You may include additional seed data as long as the required rows above remain intact and tests still pass.

8. API Specification for Frontend

An OpenAPI specification will be provided during the competition. It will include all endpoints, request and response schemas, examples, and authentication details. You must ensure your API adheres to this specification.

Error responses, such as authentication errors or validation errors, will be detailed in the OpenAPI specification.