Project Documentation

Contents

title: ”Project Documentation” author: ”DHBW Heidenheim INF2023 Semester 4” date: ”2025-09-04”

Documentation Table of Contents

README.md

altbauVsNeubau

DHBW Heidenheim INF2023 Semester 4

Use Case

This project compares indoor air quality between the old building (Marienstraße) and the new building (Hanns Voith Campus) of the Duale Hochschule Heidenheim. For this purpose, temperature and air quality are continuously measured in both buildings, centrally collected, and evaluated.

Goal

- Measurement of temperature and air quality in two different building sections
- Comparison of collected data via a central application
- Visualization of data for assessing room quality

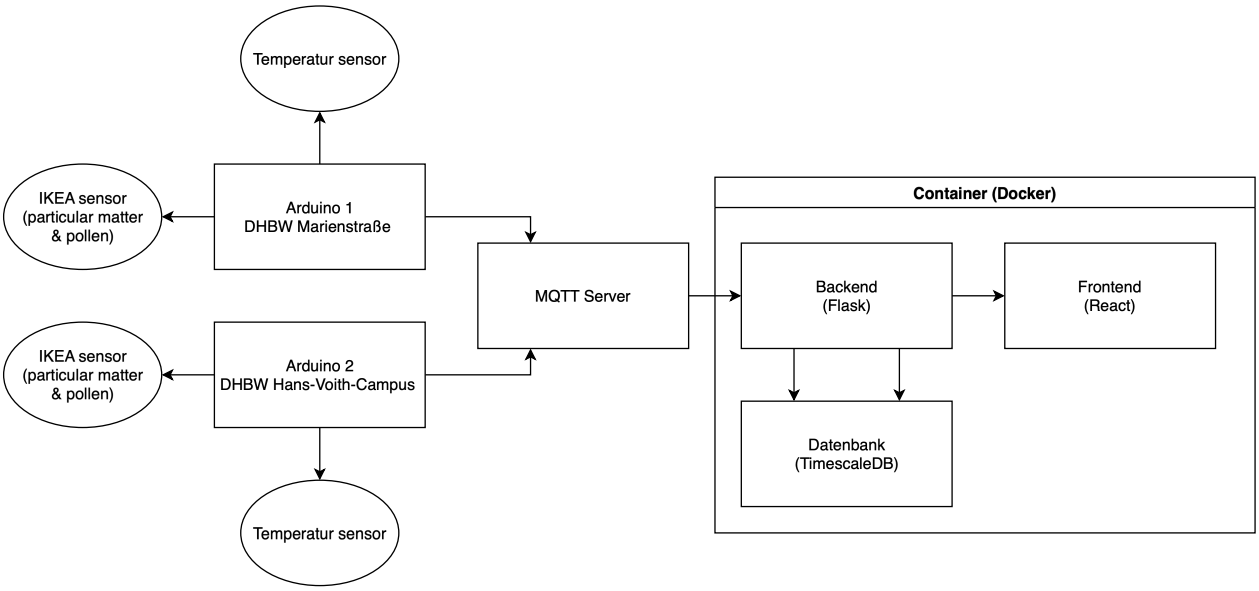
Technical Workflow

1. Two Arduino boards (one in the old building and one in the new building) regularly collect sensor data.
2. The data is sent via MQTT to a central MQTT broker.
3. A server application receives the data, validates it, and stores it in a database.
4. A web interface or dashboard graphically displays the measurement series for comparison.

Test Coverage

To ensure the functionality and reliability of the system, targeted test coverage through unit tests will be implemented. Tests will be automatically initiated via GitHub Actions with each commit to ensure code quality and stability.

Architecture



Setup Instructions

Prerequisites

- Docker and Docker Compose installed
- Git installed
- Recommended: VS Code or another code editor

Clone the Repository

```
git clone https://github.com/SWProjectTin23/altbauVsNeubau.git
cd altbauVsNeubau
```

Configuration

1. Copy the example environment file and adjust secrets/credentials as needed:

   ```
   cp .env.example .env
   ```

   Edit `.env` to set database credentials, MQTT broker info, and SMTP settings for alerting.

2. (Optional) Review and adjust `docker-compose.yml` for your environment.

Start the System

```
docker compose up -d
```

- This will start all required services:
  - TimescaleDB (PostgreSQL)
  - Backend API (Flask)
  - MQTT Handler
  - React Frontend
  - Monitoring (Prometheus, Grafana, Uptime Kuma, Loki)
  - Sensor Exporter

Access the Application

- **Frontend Dashboard:** http://localhost:3000
- **Grafana Monitoring:** http://localhost:3003
- **Prometheus:** http://localhost:9090
- **Uptime Kuma:** http://localhost:3002

Database Initialization

- The database is initialized via `db/init.sql` and `db/availability_sensor.sql` **only on first container creation**.
- If you change the schema later, you must manually apply changes using SQL tools (e.g., `psql`).

Running Tests

- Unit and integration tests are located in the `backend/tests/` directory.
- Run backend tests:
  `docker compose exec backend-api pytest`

Documentation

- Branching Strategy
- Requirements
- CI Workflows
- Frontend Documentation
- Backend Documentation
- Database Documentation
- ADR (Architecture Decision Records)

Troubleshooting

- If a service does not start, check logs with:
  `docker compose logs <service-name>`
- For database schema changes, see Database Documentation.

---

adr/0001-use-flask-for-backend.md

Use Flask for backend

Date: 2025-07-15

Status

Accepted

Context

We need to implement a lightweight, maintainable backend service that exposes APIs to receive sensor data (e.g. from MQTT), store them in a database, and provide endpoints for the frontend to retrieve data and system states. The team is familiar with Python, and many sensor and scientific data tools integrate well with the Python ecosystem. Therefore, a Python-based web framework is preferred.

Alternatives Considered

We are choosing between popular Python frameworks like Flask, FastAPI, and Django. While Django provides many features out of the box, it may be too heavy for our use case. FastAPI offers async features and automatic docs, but the team has more experience with Flask and prefers its simplicity for this project's initial scope.

Decision

We will use **Flask** as the backend framework.

Flask allows us to:

- Build REST APIs quickly with minimal boilerplate.
- Keep the project structure lean and focused.
- Easily integrate with libraries for MQTT and PostgreSQL.

Consequences

- The backend will be synchronous by default, which is acceptable for our current scale.
- If performance becomes an issue, we may revisit the choice or offload heavy processing to background tasks.
- We will write modular Flask blueprints to maintain clarity and allow future refactoring.

---

adr/0002-use-react-for-frontend.md

Use React for Frontend

Date: 2025-07-15

Status

Accepted

Context

We need to build an interactive and responsive web interface that displays sensor data (e.g., air quality, temperature, humidity) and provides a user-friendly experience for exploring and comparing time-series data. The frontend must be modular, easy to maintain, and capable of integrating with REST APIs provided by the backend.

The team already has solid experience with React, and we value a fast development cycle, reusable components, and a large ecosystem of community tools and libraries.

Alternatives Considered

We considered several frontend options:

- Vanilla JS / HTML / CSS — too low-level and inefficient for building a scalable UI.
- Vue.js — simpler than React but unfamiliar to most team members.
- Angular — powerful, but has a steep learning curve and introduces more complexity than needed.
- React — balances flexibility and structure, and is already well known by the team.

Decision

We will use React as the frontend framework.

React enables us to:

- Build reusable, component-based UIs efficiently.
- Leverage existing team expertise to accelerate development.
- Integrate third-party libraries (e.g., charts, maps) easily.
- Maintain flexibility in how we structure and scale the frontend.

Consequences

- We can start quickly without a steep learning curve.
- Our stack will require additional decisions (e.g., state management, routing), which we will document in separate ADRs.
- Build and deployment will be handled via Docker and integrated into our docker-compose setup.

---

adr/0003-utilizing-university-infrastructur-with-docker-for-project-deployment.md

Utilizing University Infrastructur with Docker for Project Deployment

Date: 2025-07-15

Status

Expired

Context

Our project consists of multiple components that require a consistent and reproducible environment for presentations and demonstrations. Currently, manually deploying the project on presentation machines is error-prone and time-consuming. We need a solution that ensures the project environment is always identical and easily accessible. The university offers a General Informatics server that could serve as a central deployment platform.

Alternatives Considered

We evaluated several approaches for project deployment:

Manual installation on presentation machines: This approach is highly error-prone, inefficient, and inconsistent. Each presentation would require re-configuration, potentially leading to version conflicts and ”it works on my machine” issues. This was rejected due to high maintenance overhead and unreliability.

Direct deployment of application files on the university server without containerization: This would require direct installation of application dependencies on the server. It’s prone to conflicts with other applications on the server and makes it difficult to ensure a consistent runtime environment that matches the development environment. This was rejected as it doesn’t sufficiently guarantee reproducibility and isolation.

Using external cloud services for hosting: While cloud services offer high availability, they would incur additional costs and potentially complicate data transfer and adherence to university internal policies. This was rejected as the internal university infrastructure is sufficient for our purposes and more cost-effective.

Decision

We will leverage Docker to deploy our project on the university’s General Informatics server. The application and its dependencies will be encapsulated in Docker containers, ensuring an isolated and reproducible runtime environment.

The deployment process will involve building Docker images and then deploying them to the university server. This will enable us to present the project at any time in a defined state and ensure that we always access the same, tested version.

Consequences

- **Consistent and Reproducible Environments:** Docker ensures that the runtime environment on the server is identical to the development environment, eliminating ”it works on my machine” problems.
- **Simplified Deployment:** Once Docker images are built, deploying them to the server is a straightforward process, minimizing manual configuration steps.
- **Efficient Resource Utilization:** Utilizing existing university infrastructure avoids the need to procure and maintain our own servers, saving costs and effort.
- **Centralized Availability:** The project will be centrally available on the university server at all times, simplifying access for presentations and team members.
- **Dependency on University Resources:** The availability and maintenance of the university server will impact the accessibility and reliability of our deployment.
- **Required Docker Knowledge:** The team will need basic knowledge of creating and managing Docker containers.

---

adr/0004-github-actions-with-self-hosted-runner-for-automated-deployment-workflow.md

GitHub Actions with Self-Hosted Runner for Automated Deployment Workflow

Date: 2025-07-22

Status

Expired

Context

Our application (comprising backend, frontend, database) is orchestrated using Docker Compose. To ensure an automated, reliable, and consistent deployment of the application to our local, internal university server whenever code changes are pushed to the main branch, we require a robust CI/CD process. Manual deployment steps are prone to errors and time-consuming.

We use GitHub as our primary platform for code development and version control. The team prefers an integrated CI/CD solution that offers direct control over the deployment environment and minimizes external dependencies, especially given the internal nature of the target server (the university server, as decided in [ADR 0003: Utilizing University Infrastruktur with Docker for Project Deployment]).

Alternatives Considered

We considered several options for automating deployment:

- **Manual Deployment:** This approach is highly error-prone, inefficient, and inconsistent. It was rejected due to its high maintenance overhead and unreliability.
- **GitHub-hosted Runners with SSH/SCP:** This alternative would require complex SSH connections, secure key management, and file transfers from an external GitHub-hosted runner to the internal target server. This would significantly increase complexity compared to direct execution on the server and expand the attack surface. It was rejected.
- **Other CI/CD Tools (e.g., Jenkins, GitLab CI):** While capable, these tools would introduce an additional, separate system outside of our GitHub-centric workflow. This would increase overhead and complexity. They were rejected as GitHub Actions offers sufficient functionality for this use case and integrates well with our existing development environment.

Decision

We will use GitHub Actions in conjunction with a self-hosted runner installed directly on the internal university server (the General Informatics server) to automatically deploy our Docker Compose setup.

The workflow will be triggered on every push to the main branch and on every pull request targeting main. It will include steps to check out the repository, gracefully stop and remove existing Docker Compose services, pull the latest Docker images, and start the services in detached mode (docker compose up -d --build).

Consequences

- **Automated & Consistent Deployments:** This ensures reliable and repeatable deployments, significantly reducing manual effort and potential errors.
- **Direct Server Control:** The self-hosted runner provides direct access to Docker commands and the host's file system, enabling efficient and precise deployment actions without complex remote access layers. This is particularly advantageous for an internal server setup.
- **Adaptation to Internal Infrastructure:** This solution integrates seamlessly with the characteristics of an internal server, avoiding the need for external services to have direct access to our internal network.
- **Dependency on Server Health:** The deployment process directly depends on the availability and health of the self-hosted runner and the target server.
- **Permissions Management:** The runner user must have appropriate permissions for Docker operations and file system access. Resolving issues like "permission denied" (e.g., from root-owned Docker volume remnants) may require administrative intervention if sudo privileges are unavailable to the runner user.
- **Runner Maintenance:** The self-hosted runner software will need to be maintained and kept running (e.g., as a systemd service) on the target server.

---

adr/0005-use-aicon-server-of-dhbw-as-hosting-infrastructure.md

Use AICON Server of DHBW Heidenheim as Hosting Infrastructure

Date: 2025-07-15

Status

expired

Context

For hosting our application, we require a reliable server infrastructure that supports Docker Compose and is accessible to our development team. Renting and maintaining a dedicated server would incur costs and administrative overhead. As students of DHBW Heidenheim, we have access to the AICON server, which is provided by the university for student projects.

The AICON server is available free of charge and is already equipped with the necessary resources and network configuration for our project. However, it is a shared resource, used by multiple student teams, which may lead to resource contention or configuration conflicts. Additionally, the server is only accessible from within the university network; external access requires a VPN connection.

Decision

We will use the AICON server provided by DHBW Heidenheim as the hosting infrastructure for our project.

- The server is cost-free and maintained by the university IT department.

- It supports Docker and Docker Compose, meeting our technical requirements.
- The infrastructure is shared with other student projects, so we must be mindful of resource usage and potential conflicts.
- Access to the server is restricted to the university network; remote access from outside requires a VPN connection.

Consequences

- **No Hosting Costs:** We avoid expenses and administrative effort associated with renting and maintaining a private server.
- **Shared Resources:** Resource contention or accidental interference with other student projects is possible. Coordination and communication with other teams may be necessary.
- **Limited Accessibility:** The server is only reachable from the university network or via VPN, which may complicate remote development and monitoring.
- **University IT Support:** The server is maintained by the university, reducing our operational burden but also limiting our control over hardware and base configuration.
- **Security & Privacy:** Data and services are hosted in a controlled, academic environment, but privacy and security depend on the shared nature of the infrastructure.

Summary:
Using the AICON server is a pragmatic and cost-effective solution for our project, leveraging university resources while accepting the limitations of shared infrastructure and restricted access.

---

adr/0006-use-rechart-tailwind-for-frontend.md

Use Tailwind CSS and Recharts for Styling and Charting

Date: 2025-07-24
Status: Accepted

Context

The web application must display real-time sensor data (e.g., temperature, humidity, pollen levels, particulate matter) and provide interactive time-series visualizations. The UI should be modern, responsive, and user-friendly. Users should also be able to:

- Compare data across different buildings (e.g., Altbau vs. Neubau).
- Define custom alert thresholds.
- Visualize historical data across configurable time ranges.
- Receive clear feedback on system errors.

Technical requirements emphasize responsive styling, seamless React integration, fast load times, and smooth transitions between views.

Functional Requirements Recap

1. **Display of current values** (for temperature, humidity, pollen, particulate matter)
2. **Time-series visualizations** with configurable time ranges and building comparison
3. **Custom warning thresholds** and visual alerts on threshold breaches

4. **Local storage of user preferences**
5. **Clear error messages** in case of network or system issues

Technical Requirements

- Use **Recharts** for visualizations
- Use **Tailwind CSS** for modern, responsive UI
- Ensure fast performance and smooth UI transitions

Alternatives Considered

- **Chart.js** – Popular and powerful, but not optimized for React.
- **Styled Components / SCSS / plain CSS** – Offers full control but requires more setup, maintenance, and consistency enforcement.
- **Recharts** – Built for React, supports declarative syntax, and is easy to integrate.
- **Tailwind CSS** – Utility-first framework that fits naturally with React's component-driven approach and ensures consistent styling.

Decision

We will use:

- **Tailwind CSS** for UI styling
- **Recharts** for time-series data visualization

These tools offer seamless integration with React, promote maintainability, and support our requirements for performance, customization, and developer productivity.

Consequences

Tailwind CSS:

- Enables rapid UI development using utility classes directly in JSX
- Reduces need for separate CSS files and avoids naming conflicts
- Provides responsive design patterns out-of-the-box
- Customizable through a central Tailwind configuration

Recharts:

- Native React integration (no wrappers needed)
- Declarative chart configuration makes logic easy to follow
- Supports dynamic updates and comparisons (e.g., by building or time range)
- Simple integration of custom highlights and threshold indicators

---

adr/0007-frontend-authentication.md

No Authentication for the Web Application

Date: 2025-07-24
Status: Accepted

Context

The web frontend is designed to display environmental sensor data such as temperature, humidity, pollen levels, and particulate matter. While the application allows basic user interactions—like setting local threshold alerts and selecting time ranges—it does not involve any sensitive, personal, or private data.

There are no user-specific features that require persistent identity, and no operations that could compromise the system or data integrity. The main functionality is read-only access to public environmental data.

Implementing authentication would require significant effort (backend logic, token management, secure storage, error handling) without a clear benefit to the user or system.

Alternatives Considered

- **Implement authentication via OAuth or custom login system** – Secure, but adds complexity, infrastructure overhead, and unnecessary friction for the user.
- **Anonymous access with optional settings stored locally** – Simple, fast, and meets current functional needs.

Decision

We will **not implement any authentication mechanism** in the frontend application at this stage.

All data access will remain anonymous and unrestricted. Local settings (e.g., custom thresholds or selected views) will be stored in the database.

Consequences

- **Simplified development** – No need for user account handling, token management, session expiration, or login UI.
- **Faster user access** – Users can access the application instantly without sign-in barriers.
- **Limited personalization** – Only basic preferences are supported.
- **Potential future requirement** – If more advanced user-specific functionality (e.g., saving settings to a backend, sharing views) is introduced, authentication may need to be revisited in a future ADR.

---

adr/0008-use-timescaledb-for-db.md

use timescaledb for db

Date: 2025-07-22

Status

Accepted

Context

Our project processes sensor data that is sent at regular short intervals (e.g., temperature, humidity, particulate matter). These are time-series data, which require efficient storage,

querying, and support for operations like aggregations, range filtering, and long-term archiving.

Decision

We decided to use **TimescaleDB** as our primary database system, as our application deals with high-frequency, time-stamped sensor data. This type of workload is best handled by a time-series database, which is specifically designed for efficient ingestion, storage, and querying of time-based data.

Why a Time-Series Database

Compared to a traditional relational database, a time series database offers significant advantages: it supports fast inserts at scale, efficient storage through compression, and powerful time-based query capabilities such as range filtering and aggregations.

Why TimescaleDB

- Optimized for time-series data: TimescaleDB extends PostgreSQL with native support for time-series features like hypertables, automatic partitioning, compression, and continuous aggregates.
- PostgreSQL-compatible: Because TimescaleDB is a PostgreSQL extension, we can still use the broader PostgreSQL ecosystem (e.g., SQLAlchemy, pgAdmin, psycopg2).
- Efficient storage and performance: Compared to traditional relational databases, TimescaleDB handles large volumes of time-series data much more efficiently.

Alternatives Considered

- PostgreSQL without Timescale extension: Functional but lacks efficiency for large-scale time-series data handling.
- InfluxDB: Designed for time-series, but less flexible for complex relational queries and lacks full SQL support.

Consequences

- We will set up a PostgreSQL instance with TimescaleDB extension enabled via Docker.
- All sensor data will be stored in hypertables to enable time-based queries.
- Our query structure will follow TimescaleDB best practices for time-series analysis.

---

adr/0009-MQTT-communication-protocol.md

Use MQTT as Communication Protocol

Date: 2025-07-25

Status

Accepted (externally specified)

Context

In this project, microcontrollers (e.g., Arduino MKR WiFi 1010) send sensor data to a central backend. The transmission uses the MQTT protocol and a predefined JSON format containing timestamped measurement values and metadata such as firmware version and startup time:

```
{
  "timestamp": "1753098733",
  "value": 23.4,
  "sequence": 123,
  "meta": {
    "firmware": "v1.2.3",
    "startup": "1753098730"
  }
}
```

The MQTT topic structure is also specified externally and follows this pattern:

`dhbw/ai/si2023/<group-number>/<sensor-type>/<sensor-id>`

The use of MQTT was mandated by external stakeholders and is considered a project requirement.

Alternatives Considered

No internal alternatives were evaluated, as MQTT was preselected and imposed externally.

Decision

We will use **MQTT** as the communication protocol between Arduino-based sensors and the backend system.

Consequences

- MQTT is lightweight, well-suited for IoT scenarios, and broadly supported in the Python ecosystem.
- The predefined message format and topic structure simplify integration on both sender and receiver sides.
- Security considerations (such as authentication and TLS encryption) may need to be added separately, as MQTT does not enforce them by default.

---

adr/0010-migration-to-own-server-infrastructure.md

Migration to Own Server Infrastructure Instead of AICON Server

Date: 2025-07-23

Status

Accepted

Context

Previously, we decided to use the AICON server provided by DHBW Heidenheim as our hosting infrastructure (see ADR-5). While this solution offered cost savings and university IT support, several limitations have impacted our development and operational efficiency:

- **Limited External Access:** The AICON server is only accessible from within the university network or via VPN, complicating remote development, monitoring, and collaboration.
- **Restricted Configuration:** We have limited control over server configuration, which hinders optimization, troubleshooting, and the ability to install or update required software.
- **Resource Constraints:** The server resources are shared among multiple student projects, leading to potential performance issues and resource contention.
- **Reliability Issues:** The shared nature and limited oversight have resulted in increased error rates and downtime, affecting the stability of our application.

Decision

We will migrate our project from the AICON server to our own dedicated server infrastructure.

- The new infrastructure will be fully under our control, allowing unrestricted configuration and optimization.
- We will ensure reliable external access for all team members, regardless of location.
- Dedicated resources will improve performance and scalability.
- We will be able to implement robust monitoring, backup, and security measures tailored to our needs.

Consequences

- **Increased Costs:** We will incur expenses for server rental, maintenance, and administration.
- **Full Control:** We gain complete control over hardware, software, and network configuration.
- **Improved Accessibility:** Team members can access the infrastructure from anywhere without VPN restrictions.
- **Enhanced Reliability:** Dedicated resources and oversight will reduce downtime and error rates.
- **Operational Responsibility:** We are responsible for server maintenance, updates, and security.

Summary:
Migrating to our own server infrastructure addresses the limitations of the AICON server, providing greater flexibility, reliability, and accessibility for our project, at the cost of increased operational responsibility and expenses.

---

adr/0011-use-Grafana-loki-for-logging.md

Use Grafana Loki for logging

Date: 2025-08-08

Status

Accepted

Context

Our project consists of a React frontend, a Flask backend, and a PostgreSQL database. As we prepare for production deployment, we want to introduce centralized logging to capture errors, warnings, and important system events in a consistent and queryable manner.

The goal is to implement a lightweight, extensible logging solution that can run both locally and in the cloud. The solution should collect logs from the Flask backend and provide centralized search and visualization capabilities.

Alternatives Considered

1. Sentry (Self-hosted):
Sentry offers powerful error tracking and performance monitoring. However, the self-hosted version is relatively heavy (multiple services, large Docker images, complex setup), making it an overkill for our current needs.

2. ELK Stack (Elasticsearch, Logstash, Kibana):
A well-established logging stack with powerful capabilities. However, it is resource-intensive and complex to operate. Elasticsearch requires significant memory and storage, and Logstash adds additional overhead.

3. JSON Logging + External Tools:
Structured logging with JSON enables better machine parsing, but for our current needs (quick setup, human-readable logs, small team), plain text is more practical. JSON logging remains a viable upgrade path if structured querying becomes necessary.

Decision

We will implement a logging stack based on:

- **Flask** writing log messages to a local file (`app.log`) in **plain text format**
- **Promtail**, which reads the log file and attaches metadata (labels)
- **Grafana Loki**, which receives and stores the logs for indexed, time-based querying
- **Grafana**, used to visualize and explore the logs via a web UI

Consequences

- **Low Complexity:** The setup is lightweight, easy to run locally, and does not require complex infrastructure like Elasticsearch or Kubernetes.
- **Fast Log Search:** Loki enables fast text-based search across logs using labels and timestamps directly within Grafana.
- **Limited Structure:** Plain text logs are easy for developers to read but provide limited capabilities for structured analysis. This can be mitigated later by switching to JSON.
- **Scalability:** The system can be extended to include additional services or logging sources without major architectural changes.
- **Unified Monitoring Platform:** Since we are already using Grafana, we can later integrate metrics and tracing alongside logs in a single interface.

Extension: Frontend Logging (React)

Although the initial logging stack covers only the Flask backend, it's also important to capture logs from the React frontend.

Goals for Frontend Logging

- Capture JavaScript runtime errors
- Log user actions or client-side events (optional)
- Report performance issues (optional)

Decision

The React frontend will send relevant logs to the Flask backend via HTTP. Flask will then write these logs to the same local log file (`app.log`) used by backend logs. Promtail will continue to read from this file and forward all logs to Loki.

- A simple logging function in React will capture errors (e.g. via `window.onerror` or a custom logger).
- Logs are sent to an endpoint like `POST /frontend-log` on the Flask server.
- The Flask endpoint will parse and log the message (with proper labels like `[FE]` or `source=frontend`).
- Promtail will include these in its scrape targets, so frontend logs appear in Grafana.

Alternatives Considered

Direct Loki Push from React: Requires Loki to be exposed or proxied, plus authentication, which adds security and complexity concerns. This was rejected in favor of a simpler backend relay.

Using an external tool (e.g. Sentry, LogRocket): More powerful for production debugging, but heavier and often cloud-bound. Our stack prioritizes self-hosting and simplicity.

Consequences

- **Unified Logging:** Both backend and frontend logs will be searchable in the same Grafana interface.
- **Simple Setup:** No additional infrastructure required.
- **Extensible:** If needed, we can later structure frontend logs as JSON or include user/session metadata.

---

adr/0012-useuptime-kuma.md

Introduction of Uptime Kuma and Uptime Robot for Service Availability Monitoring

Date: 2025-08-07

Status

Accepted

Context

Currently, there is no consistent or centralized method in place for monitoring the availability of internal and external services (e.g., APIs, frontend, MQTT, database). Outages may go unnoticed or are only discovered reactively. A reliable monitoring solution is required to detect downtimes early, notify the team, and provide historical uptime tracking.

The chosen solution should:

- Be easy to deploy and maintain
- Support internal and external monitoring
- Include alerting and a web-based status overview
- Be lightweight and suitable for integration into our Dockerized infrastructure

Alternatives Considered

1. Elasticsearch
Elasticsearch was considered as a logging and monitoring backend by storing availability checks and analyzing them via queries or dashboards. However, Elasticsearch is a heavyweight solution that consumes significant resources (CPU, memory, disk), especially in smaller deployments. For the relatively simple use case of availability monitoring, it is unnecessarily complex and resource-intensive.

2. JMeter with Grafana
JMeter is a powerful tool for performance and load testing. It could be scheduled to perform uptime checks and push results into a time-series database (e.g., InfluxDB), visualized through Grafana. However, JMeter is not designed for continuous availability monitoring. It's best suited for short-term performance testing. The setup would also require additional components, increasing maintenance overhead.

3. Uptime Robot (alone)
Uptime Robot is a reliable external SaaS uptime monitoring service. It works well for public endpoints but cannot access services running on local networks or behind firewalls. Therefore, it is insufficient as a standalone monitoring solution.

Decision

We will use **Uptime Kuma** for internal monitoring and **Uptime Robot** for external validation.

- **Uptime Kuma** is a self-hosted, Docker-based uptime monitoring tool that supports HTTP(s), TCP, ping, and more. It will be deployed internally and used to monitor all local services.
- **Uptime Robot** will be used in parallel to perform external monitoring of publicly reachable endpoints (e.g., production APIs, website).
- Alerts from both systems will be configured via email and Telegram where possible.

Consequences

- **Full Coverage:** We gain visibility into both internal and external service availability.
- **Simple Maintenance:** Uptime Kuma is lightweight and runs as a single Docker container with persistent storage (`kuma.db`).
- **Low Overhead:** No need for complex data pipelines, custom scripting, or multi-container observability stacks.
- **Alerting and Dashboard:** Web UI and real-time alerts are available without additional tools.
- **Security Considerations:** Internal access to Uptime Kuma will be restricted to trusted networks or secured via reverse proxy.

---

adr/0013-jmeter-for-frontend-testing.md

Use JMeter for Load and Performance Testing

Date: 2025-08-13

Status

Accepted

Context

In this project, we need to verify that the frontend and backend API can handle expected traffic levels and meet specific Service Level Objectives (SLOs) for response times.
The test setup should be able to:

- Simulate realistic user behavior (e.g., loading the homepage, clicking through data intervals like 3h, 1d, 1w, 1m).
- Measure performance metrics such as Time-To-Last-Byte (TTLB) and API response times.
- Generate detailed reports (HTML dashboards, raw data files) for analysis.
- Integrate seamlessly into both local developer workflows and automated CI/CD pipelines.

Apache JMeter was evaluated and selected because it meets all these requirements and fits well with our Docker-based infrastructure.

Alternatives Considered

- **k6**
  Modern and scriptable in JavaScript, but lacks built-in GUI for test plan creation and has less native CSV/HTML reporting without additional tooling.
- **Custom Python scripts**
  Flexible and Pythonic, but would require building reporting and CI integration manually, as well as re-creating complex browser/resource loading logic.

Decision

We will use **Apache JMeter** for load and performance testing because:

- It has a mature ecosystem with strong community support.
- It supports both GUI-based and headless (CLI) execution, allowing easy local design and automated CI/CD runs.
- It produces detailed HTML reports and raw JTL data for further processing.
- It integrates well with Docker for reproducible and isolated test environments.
- It supports parameterization via environment variables, making it CI-friendly.

Consequences

- JMeter test plans (`.jmx` files) can be complex XML and may require knowledge of its structure for advanced modifications.
- Test execution will require Docker or a JMeter installation in the environment.

- Existing team members can leverage JMeter's GUI to modify test scenarios without writing code.
- Easy integration with GitHub Actions enables automatic SLO validation on pull requests.

---

adr/0014-use-grafana-for-sensor-uptime-monitoring.md

Use Grafana Dashboards for Sensor Data Availability Monitoring

Date: 2025-08-20

Status

Accepted

Context

Our IoT monitoring stack collects sensor data via MQTT and stores it in TimescaleDB.
To ensure system reliability, we must verify that expected sensor readings are consistently ingested into the database and detect outages or communication gaps early.

The goal is to provide a **clear visibility layer** for sensor availability using Grafana dashboards. This should highlight:

- How many data points were **expected** vs. how many were **actually ingested**
- The **availability percentage** per device
- Any **gaps in data transmission**

Grafana is already part of our stack, which makes it a natural choice for visualization.

Alternatives Considered

1. Raw SQL Queries via pgAdmin:
Direct SQL queries in pgAdmin can reveal missing or delayed sensor data. However, this approach is manual, requires database expertise, and lacks visual clarity for stakeholders.

2. Custom React Dashboard:
Building a custom frontend to visualize expected vs. actual values offers full flexibility but duplicates Grafana functionality. It requires significant development effort and ongoing maintenance.

Decision

We will implement availability monitoring in **Grafana dashboards**, based on TimescaleDB views.

- Use PostgreSQL datasource (`timescaledb`) pre-provisioned in Grafana.
- Build panels to display:
  - **Expected vs. Actual Totals** (Stat panels per device)
  - **Availability Percentage** (Gauge visualization with thresholds)
  - **Data Gaps** (Table listing gaps longer than 10 minutes)
- Leverage TimescaleDB views (`v_totals_since_start_by_device`, `v_sensor_gaps`) for efficient queries.

- Refresh dashboards every 30s for near real-time visibility.

Consequences

- **Low Effort, High Value:** Reuses Grafana, which is already deployed in the stack.
- **Immediate Visibility:** Operators can quickly see if sensors are dropping data.
- **Alerting Ready:** Grafana Alerting can be added later to notify when availability falls below thresholds.

---

adr/0015-use-grafana-alerting-for-sensor-availability-notifications.md

Use Grafana Alerting for Sensor Availability Notifications

Date: 2025-08-20

Status

Accepted

Context

Continuous monitoring of sensor data availability is essential for system reliability. While Grafana dashboards (see ADR-14) provide real-time visibility, operators also need **proactive notifications** when sensors go offline or data gaps occur.

Sensor availability is determined by an **external service** (`exporter.py`), which exposes Prometheus metrics (`sensor_seconds_since_last_data`) for each individual sensor. This allows us to monitor the availability of every single sensor in the system.

Several alerting solutions were considered:

Alternatives Considered

1. Custom Notification Service:
A bespoke service could monitor TimescaleDB and send alerts via email or messaging platforms. This approach offers flexibility but requires additional development, integration, and maintenance.

2. Prometheus Alertmanager:
Prometheus Alertmanager is widely used for metric-based alerting. However, our main sensor data source is TimescaleDB, and integrating Alertmanager would add complexity.

3. Grafana Unified Alerting:
Grafana's built-in alerting system supports multiple datasources, including Prometheus and PostgreSQL. It allows rule-based notifications, integrates with dashboards, and supports email, Slack, and other channels.

Decision

We will use **Grafana Unified Alerting** for sensor availability notifications.

- Sensor availability is monitored by the external service `exporter.py`, which provides Prometheus metrics for each sensor.
- Alert rules are defined in Grafana based on these metrics (e.g., `sensor_seconds_since_last_data`).

- The availability of **each individual sensor** is checked, and alerts are triggered if any sensor becomes unavailable.
- Configure email notifications for responsible operators.
- Integrate alert summaries directly into dashboard panels for context.
- Provision alerting resources via code for reproducibility and version control.

This approach leverages our existing Grafana deployment (see ADR-14), minimizes operational overhead, and provides a unified monitoring and alerting experience.

Consequences

- **Consistent User Experience:** Operators use a single interface for both dashboards and notifications.
- **Scalable:** Alerting rules and contact points can be extended as requirements evolve.
- **Maintainable:** Alerting configuration is versioned and provisioned alongside dashboards.
- **Granular Monitoring:** The availability of every individual sensor is tracked and alerted on, improving reliability and transparency.

---

adr/0016-use-backend-alerting-for-threshold-Violations.md

Use Backend-Based Alerting for Sensor Threshold Notifications

Date: 2025-08-28

Status

Accepted

Context

Timely notification when sensor values exceed defined thresholds is critical for air quality monitoring and operational safety. While Grafana offers powerful alerting capabilities (see ADR-14), its configuration for value-based threshold alerting—especially with dynamic, per-device rules—proved to be overly complex and difficult to maintain.

Several alternatives were considered:

Alternatives Considered

1. Grafana Unified Alerting:
Grafana's alerting system is well-suited for metric-based and availability notifications. However, implementing fine-grained, per-device threshold alerts for air quality metrics (e.g., temperature, humidity, pollen, particulate matter) would require extensive dashboard and rule configuration. Managing dynamic thresholds and custom notification logic for each sensor in Grafana was found to be impractical and error-prone.

2. Backend-Based Alerting:
Integrating alerting logic directly into the backend API allows for flexible, programmatic threshold checks and notification workflows. The backend can evaluate incoming sensor data

against configured thresholds, manage alert cooldowns, and send context-rich email notifications to operators. This approach centralizes alerting logic, simplifies threshold management, and enables stateful alerting (e.g., cooldown resets when values return to normal).

Decision

We will implement **backend-based alerting** for sensor threshold notifications.

- The backend API receives all sensor values and evaluates them against configured thresholds.
- When a threshold is exceeded, the backend sends an alert email to the responsible operator.
- A state-based cooldown prevents repeated notifications while the value remains outside the threshold.
- The cooldown is reset only when the value returns to the normal range, allowing new alerts for subsequent threshold violations.
- Alerting configuration (thresholds, email addresses) is managed via API endpoints.

This approach avoids the complexity of managing dynamic alerting rules in Grafana and ensures maintainable, scalable, and context-aware notifications.

Consequences

- **Simplified Alerting Logic:** Threshold checks and notifications are handled programmatically in the backend, reducing operational overhead.
- **Maintainable and Scalable:** Alerting rules and thresholds can be updated via API, supporting future expansion and customization.
- **Context-Rich Notifications:** Emails include device, metric, value, threshold details, and actionable recommendations.
- **Unified Data Flow:** All sensor data and alerting logic are managed in one place, improving reliability and traceability.
- **Reduced Grafana Complexity:** Grafana remains focused on visualization and availability monitoring, while the backend handles value-based alerts.

---

adr/0017-auto-deployment.md

Automated Deployment via GitHub CI/CD

Date: 2025-09-02

Status

Accepted

Context

As described in ADR 0010: Migration to Own Server Infrastructure Instead of AICON Server, we moved our hosting from university-managed servers to our own dedicated infrastructure. This change provided us with full control, improved accessibility, and enhanced reliability.

Decision

We implemented an **automated deployment process** using a GitHub Actions CI/CD pipeline:

- On every push to the main branch, the pipeline builds and tests the application.
- Successful builds are deployed to our own server infrastructure using Docker and Docker Compose.
- The pipeline ensures that the servers always run the latest tested version of the application in a reproducible environment.

Consequences

- **Consistent Deployment:** The GitHub CI/CD pipeline guarantees reproducible and up-to-date deployments.
- **Full Control:** We manage the deployment infrastructure and process independently.
- **Operational Responsibility:** Maintenance, updates, and security of the servers remain our responsibility.

---

adr/0018-redundant-MQTT-brokers.md

Redundant MQTT Brokers for Increased Reliability

Date: 2025-09-02

Status

Accepted

Context

In ADR 0017: Migration to Private Servers with GitHub CI/CD Pipeline for Automated Deployment we moved our deployment infrastructure from the university server to two privately managed servers.
In ADR 0009: Use MQTT as Communication Protocol it was decided that all microcontrollers (e.g., Arduino MKR WiFi 1010) communicate with the backend using **MQTT** and a predefined JSON format with externally specified topic structure.

Initially, a single MQTT broker instance was used. While this fulfilled the communication requirements, it introduced a **single point of failure**: if the broker became unavailable, all sensor data transmission and backend communication would stop.

To mitigate this risk and increase availability, we evaluated the use of multiple MQTT brokers.

Decision

We migrated from a single MQTT broker to **two MQTT brokers running on different private servers**.

Both brokers use the same configuration regarding authentication, topic structure, and message format.
Clients (microcontrollers and backend consumers) are configured to support multiple broker endpoints, ensuring continued operation if one broker fails.

This setup provides fault tolerance and improved reliability without changing the externally mandated MQTT protocol or topic/message format.

Consequences

- **Improved Reliability:** System remains operational even if one broker/server fails.
- **Load Distribution:** Depending on client configuration, the two brokers can share traffic, preventing overload.
- **Configuration Overhead:** Clients must be configured with multiple broker endpoints, which adds minor complexity.
- **Operational Costs:** Running two brokers increases resource usage and administrative overhead.
- **Consistency Management:** If both brokers operate independently (no clustering), ensuring synchronized state (e.g., retained messages) may require additional mechanisms.

---

adr/0019-use-css-instead-of-tailwind.md

Use plain CSS instead of Tailwind

Date: 2025-09-03

Status

Accepted

Context

For styling our frontend, several options were considered. Initially, we evaluated **Tailwind CSS** because it offers fast development with utility classes and ensures consistent design.

However, during setup we encountered configuration issues that would require extra effort to resolve. Additionally, the benefits of Tailwind for our current project are limited, as we do not require highly complex or design-heavy interfaces.

The team already has solid experience with plain CSS, making it the simpler and more direct solution.

Alternatives Considered

- **Tailwind CSS** — modern utility-first approach, but caused configuration problems and introduced unnecessary overhead.
- **Plain CSS (possibly modularized)** — straightforward, no setup overhead, already familiar to the team.

Decision

We will use **plain CSS** for styling the frontend.

This enables us to:

- Start immediately without additional tooling hurdles.
- Leverage the team's existing knowledge.
- Maintain full control over styles without framework dependencies.

Consequences

- Fewer built-in design utilities compared to Tailwind.
- We must ensure style consistency (spacing, colors, components) ourselves.
- No additional build or configuration issues from Tailwind.
- The project remains leaner and easier to maintain.

---

backend/alerts.md

Alerting Documentation

Overview

This project uses Grafana's unified alerting system to monitor the availability of **all sensors** (not just Arduino devices). Alerts are triggered when any sensor has not sent data for a defined period. Notifications are sent via email to the configured recipients.

Alert Rule

- **Location:** `grafana/provisioning/alerting/alert-rules.yaml`
- **Rule Name:** Sensor-offline
- **Condition:** The alert triggers if `sensor_seconds_since_last_data` for any sensor (any device and any sensor type) exceeds **600 seconds** (10 minutes).
- **Interval:** The rule is evaluated every 30 seconds.
- **Summary Message:**
  The alert email contains the device ID, sensor type, and the number of seconds since the last data was received:
  ```
  Folgende Sensoren sind offline:
    Gerät: {{ .labels.device_id }}, Sensor: {{ .labels.sensor_type }}, Zeit: {{ .value }} Sekunden
  ```
  This ensures the exact offline duration for each sensor is shown in the notification.

Contact Points

- **Location:** `grafana/provisioning/alerting/contact-points.yaml`
- **Default Email Receiver:**
  - Recipients:
    - hirschmillert.tin23@student.dhbw-heidenheim.de
    - geroldj.tin23@student.dhbw-heidenheim.de
  - Multiple recipients are supported.
  - Notifications are sent for both firing and resolved alerts.

Data Sources

- **Prometheus:**
  Used for querying the metric `sensor_seconds_since_last_data` for all sensors.

Dashboard Integration

- **Location:** `grafana/provisioning/dashboards/ArduinoAvailability.json`
- **Panels:**
  - Stat and time series panels visualize the offline time for each sensor and device.
  - Thresholds are set to highlight sensors offline for more than 600 seconds.

Customization

- **Threshold:**
  Adjust the `params` value in the alert rule to change the offline threshold (default: 600 seconds).
- **Notification Message:**
  Edit the `summary` field in the alert rule to customize the email content.

Troubleshooting

- Ensure Prometheus is scraping the correct metrics for all sensors.
- Check that the email addresses in `contact-points.yaml` are valid.
- Verify that the alert rule is enabled and not paused.

References

- Grafana Alerting Documentation
- Provisioning Alerting Resources

---

backend/api.md

API Overview

This document describes the available REST API endpoints for the sensor backend.

Endpoints are under development and may change as needed.

Base URL

`http://localhost:5001/api/`

Endpoints

1. Get available time range

GET `/devices/range`

- Returns the earliest and latest available timestamps for all devices.

Example: `http://localhost:5001/api/devices/range`

Success Response:

```
{
  "status": "success",
  "data": [
    {
      "device_id": 1,
      "start": 1721736000,
      "end": 1721745000
    },
    {
      "device_id": 2,
      "start": 1721736300,
      "end": 1721744700
    }
  ]
}
```

No Data Response:

```
{
  "status": "success",
  "message": "No time ranges found for any devices.",
  "data": []
}
```

Error Response:

```
{
  "status": "error",
  "message": "A database error occurred while processing your request."
}
```

---

2. Get sensor data by device ID and time range

GET `/devices/<int:device_id>/data`

- Returns all available data for a specific device.
- Optional: time filters via query parameters.

Path Parameter:

- `device_id`: integer ID of the sensor device

Query Parameters:

- `start` *(optional)*: start timestamp in UNIX format
- `end` *(optional)*: end timestamp in UNIX format

Example: `http://localhost:5001/api/devices/1/data?start=1721736000&end=1721745660`

Success Response:


```
{
  "device_id": 1,
  "start": 1721736000,
  "end": 1721745660,
  "status": "success",
  "data": [
    {
      "device_id": 1,
      "humidity": 45.2,
      "particulate_matter": 28,
      "pollen": 10,
      "temperature": 22.1,
      "timestamp": 1721745600
    },
    {
      "device_id": 1,
      "humidity": 45.5,
      "particulate_matter": 30,
      "pollen": 11,
      "temperature": 22.3,
      "timestamp": 1721745660
    }
  ],
  "message": null
}
```


No Data Response:


```
{
  "device_id": 1,
  "start": 1721736000,
  "end": 1721745660,
  "status": "success",
  "data": [],
  "message": "No data available for device 1 in the specified range."
}
```


Device Not Found Response:


```
{
  "status": "error",
  "message": "Device with ID 999 does not exist."
}
```


Error Response:


```
{
  "status": "error",
  "message": "A database error occurred while processing your request."
}
```


---

3. Get Latest Data for a Device

GET `/devices/<int:device_id>/latest`

- Returns the most recent data entry for a specific device.

Path Parameter:

- `device_id`: integer ID of the sensor device

Example: `http://localhost:5001/api/devices/1/latest`

Success Response:

```
{
  "status": "success",
  "data": {
    "device_id": 1,
    "humidity": 45.5,
    "particulate_matter": 30,
    "pollen": 11,
    "temperature": 22.3,
    "timestamp": 1721745660
  },
  "message": null
}
```

No Data Response:

```
{
  "status": "success",
  "data": [],
  "message": "No data available for device 1."
}
```

Device Not Found Response:

```
{
  "status": "error",
  "message": "Device with ID 999 does not exist."
}
```

Error Response:

```
{
  "status": "error",
  "message": "A database error occurred while processing your request."
}
```

---

4. Comparison Between Devices over Time Range (Aggregated)

GET `/comparison`

- Returns the selected metric of two devices over a given time range.

- The data is aggregated in buckets (e.g., 100 average values per device) for efficient charting.

Note: If a device currently sends no data for the requested time range, its array will be empty. In that case no buckets with "value" entries are returned for that device (the frontend must handle empty arrays).

Query Parameters

- `device_1`: ID of first device (e.g., 1)
- `device_2`: ID of second device (e.g., 2)
- `metric`: one of `temperature`, `humidity`, `pollen`, `particulate_matter`
- `start`: Unix timestamp (optional)
- `end`: Unix timestamp (optional)
- `buckets`: *(optional, default: 300)* Number of buckets (average values) to return per device

Example: `http://localhost:5001/api/comparison?device_1=1&device_2=2&metric=pollen&start=17217`

Success Response:

```
{
  "device_1": [
    { "timestamp": 1721745600, "value": 10 },
    { "timestamp": 1721745660, "value": 11 }
  ],
  "device_2": []],
  "metric": "pollen",
  "start": 1721745600,
  "end": 1721745660,
  "status": "success",
  "message": null
}
```

- Each array contains up to `buckets` entries, each representing the average value for that time bucket.

No Data Response:

```
{
  "device_1": [],
  "device_2": [],
  "metric": "pollen",
  "start": 1721745600,
  "end": 1721745660,
  "status": "success",
  "message": "No data found for the specified devices and metric."
}
```

Error Responses:

```
{
  "status": "error",
  "message": "Metric must be specified."
}
```

```
{
  "status": "error",
  "message": "Both device IDs must be provided."
}
```

```
{
  "status": "error",
  "message": "Invalid time range: <error description>"
}
```

```
{
  "status": "error",
  "message": "An unexpected error occurred while processing your request."
}
```

---

5. Manage Thresholds

This endpoint allows you to retrieve and update the soft and hard thresholds for different sensor metrics (temperature, humidity, pollen, particulate matter).

GET `/thresholds` - Returns the currently configured thresholds.

Example `http://localhost:5001/api/thresholds`

Success Response:

```
{
  "status": "success",
  "data": {
    "temperature_min_soft": 12.0,
    "temperature_max_soft": 25.0,
    "temperature_min_hard": 15.0,
    "temperature_max_hard": 30.0,
    "humidity_min_soft": 10.0,
    "humidity_max_soft": 70.0,
    "humidity_min_hard": 20.0,
    "humidity_max_hard": 80.0,
    "pollen_min_soft": 5,
    "pollen_max_soft": 50,
    "pollen_min_hard": 10,
    "pollen_max_hard": 100,
    "particulate_matter_min_soft": 1,
    "particulate_matter_max_soft": 50,
    "particulate_matter_min_hard": 5,
    "particulate_matter_max_hard": 100
  },
  "message": "Thresholds retrieved successfully."
}
```

No Data Response:

```
{
  "status": "success",
  "data": [],
```

```
  "message": "No thresholds available."
}
```

Error Response:

```
{
  "status": "error",
  "message": "A database error occurred while processing your request."
}
```

```
{
  "status": "error",
  "message": "An unexpected error occurred while processing your request."
}
```

**POST `/thresholds`

The request body should be an JSON object containing the threshold values. All keys are required.

- `temperature_min_soft` (float): Soft minimum temperature threshold.
- `temperature_max_soft` (float): Soft maximum temperature threshold.
- `temperature_min_hard` (float): Hard minimum temperature threshold.
- `temperature_max_hard` (float): Hard maximum temperature threshold.
- `humidity_min_soft` (float): Soft minimum humidity threshold.
- `humidity_max_soft` (float): Soft maximum humidity threshold.
- `humidity_min_hard` (float): Hard minimum humidity threshold.
- `humidity_max_hard` (float): Hard maximum humidity threshold.
- `pollen_min_soft` (int): Soft minimum pollen threshold.
- `pollen_max_soft` (int): Soft maximum pollen threshold.
- `pollen_min_hard` (int): Hard minimum pollen threshold.
- `pollen_max_hard` (int): Hard maximum pollen threshold.
- `particulate_matter_min_soft` (int): Soft minimum particulate matter threshold.
- `particulate_matter_max_soft` (int): Soft maximum particulate matter threshold.
- `particulate_matter_min_hard` (int): Hard minimum particulate matter threshold.
- `particulate_matter_max_hard` (int): Hard maximum particulate matter threshold.

Validation Rules:

- For each metric, the `min_soft` value must be less than `max_soft`.
- For each metric, the `min_hard` value must be less than `max_hard`.
- For each metric, the `min_hard` value must be less than or equal to `min_soft`.
- For each metric, the `max_hard` value must be greater than or equal to `max_soft`.

Example Request Body:

```
{
  "temperature_min_soft": 12.0,
  "temperature_max_soft": 25.0,
  "temperature_min_hard": 10.0,
  "temperature_max_hard": 28.0,
  "humidity_min_soft": 30.0,
  "humidity_max_soft": 60.0,
  "humidity_min_hard": 25.0,
  "humidity_max_hard": 65.0,
```

```
  "pollen_min_soft": 10,
  "pollen_max_soft": 70,
  "pollen_min_hard": 5,
  "pollen_max_hard": 80,
  "particulate_matter_min_soft": 5,
  "particulate_matter_max_soft": 40,
  "particulate_matter_min_hard": 2,
  "particulate_matter_max_hard": 50
}
```

Success Response:

```
{
  "status": "success",
  "message": "Thresholds updated successfully."
}
```

Error Responses:

```
{
  "status": "error",
  "message": "Invalid input data. Expecting a Dictionary."
}
```

```
{
  "status": "error",
  "message": "Missing required key: 'temperature_min_soft'."
}
```

```
{
  "status": "error",
  "message": "Value for 'temperature_min_soft' cannot be None."
}
```

```
{
  "status": "error",
  "message": "Invalid value for 'temperature_min_soft': 'abc'. Expected type float."
}
```

```
{
  "status": "error",
  "message": "Minimum value for 'temperature_min_soft' must be less than maximum value for 'tem
}
```

```
{
  "status": "error",
  "message": "Hard threshold 'temperature_max_hard' must be greater than soft threshold 'temper
}
```

```
{
  "status": "error",
  "message": "A database error occurred while processing your request."
}
```

```
{
  "status": "error",
  "message": "An unexpected error occurred while processing your request."
}
```

6. Aler Email Management

GET `/alert_email`

Returns the currently configured alert email address for the thresholds.

Example: `http://localhost:5001/api/alert_email`

Success Response:

```
{
  "status": "success",
  "email": "your-alert@example.com"
}
```

Error Response:

```
{
  "status": "error",
  "message": "No mail found."
}
```

POST `/alert_email`

Sets or updates the alert email address for thresholds

Request Body:

```
{
  "alert_email": "your-alert@example.com"
}
```

Success Response:

```
{
  "status": "success",
  "message": "Alert email saved."
}
```

Error Response:

```
{
  "status": "error",
  "message": "Mail is missing."
}
```

---

7. Send Alert Mail (Threshold Alerting)

POST `/send_alert_mail`

Triggers the alerting logic for a given metric and device.
This endpoint is called by the frontend whenever a new sensor value is received.

Request Body:

```
{
  "metric": "Temperatur",
  "value": 35,
```

```
  "thresholds": {
    "Temperatur": {
      "redLow": 10,
      "yellowLow": 15,
      "yellowHigh": 30,
      "redHigh": 32
    }
  },
  "device": "Altbau"
}
```

Success Responses:

- If a new alert is triggered:
  ```
  {
    "status": "success",
    "message": "hart-Mail sent"
  }
  ```
- If the alert is still active (cooldown):
  ```
  {
    "status": "success",
    "message": "hart-Mail already active"
  }
  ```
- If the value is back in the normal range (cooldown reset):
  ```
  {
    "status": "success",
    "message": "No Threshold Exceeded"
  }
  ```

Error Response:

```
{
  "status": "error",
  "message": "Missing parameters"
}
```

---

backend/arduino.md

Arduino Code Documentation

This document describes the structure and functionality of the Arduino scripts in the **altbau-VsNeubau** project.

Overview

The Arduino scripts are responsible for collecting sensor data and sending it to the central backend via the MQTT protocol.

Main Functions

- **Sensor Initialization:** All connected sensors are initialized in the `setup()` block (`tempsensorStartup()`, `humiditySensorStartup()`).

- **WiFi Connection:** The board connects to the configured WiFi network.
- **MQTT Connection:** Establishes and maintains a connection to the MQTT broker.
- **Data Acquisition:** In the `loop()` block, sensor values are read regularly (fine dust, temperature, humidity).
- **Message Format:** The sensor values are formatted as a JSON object (see ADR 0008).
- **Publishing:** The JSON message is published to a project-specific MQTT topic.

Example MQTT Message

```
{
  "timestamp": "1753098733",
  "value": 23.4,
  "sequence": 123,
  "meta": {
    "firmware": "v1.2.3",
    "startup": "1753098730"
  }
}
```

Important Files

- `arduino_sensor.ino`: Main program, contains setup and loop logic.
- `sensor_reader.cpp`: Sensor reading and averaging logic for fine dust, temperature, and humidity.
- `WifiUtils.cpp` and `MQTTUtils.cpp`: Handle WiFi and MQTT communication.
- `secrets.h`: Contains credentials for WiFi and MQTT (excluded from the repository via `.gitignore`).

Notes

- **Credentials:** WiFi and MQTT credentials should never be published in the repository. The `secrets.h` file is excluded via `.gitignore`.
- **Adding New Sensors:** Please follow the existing structure and message format when integrating new sensors.

Example `secrets.h`:

```
#ifndef SECRETS_H
#define SECRETS_H

#define WIFI_SSID "WIFI_SSID"
#define WIFI_PASS "WIFI_PASS"
#define MQTT_SERVER "MQTT_SERVER"
#define MQTT_PORT "MQTT_PORT"
#define MQTT_SERVER2 "MQTT_SERVER2"
#define MQTT_PORT2 "MQTT_PORT2"
#define DEVICE_ID "DEVICE_ID"
#endif
```

---

backend/backend.md

Backend Overview

This backend powers the AltbauVsNeubau project. It consists of:

- API service (Flask + Flask-RESTful) exposing endpoints consumed by the frontend
- MQTT ingester that validates sensor messages and writes metrics to the database
- Database access layer (PostgreSQL/TimescaleDB) and alerting support
- Structured logging and error handling

Key components in the repository:

- API app entry: `backend/run.py`, `backend/api/__init__.py`
- MQTT ingester: `backend/mqtt_client/main_ingester.py`
- MQTT/DB config: `backend/mqtt_client/mqtt_config.py`
- Common logging: `backend/common/logging_setup.py`
- Common exceptions: `backend/common/exceptions.py`
- Database helpers: `backend/api/db/`
- Dependencies: `backend/requirements.txt`

API Service

In `backend/api/__init__.py` the Flask app is created with `create_app` and routes are also registered in the same file. Default dev port is 5001 (see `backend/run.py`).

CORS is enabled for specific origins to allow the frontend to call the API.

Endpoints

Method	Path	Purpose
GET	`/api/devices/<device_id>/data`	Time-series data for device; optional `start`, `end`, `metric`
GET	`/api/range`	Earliest/latest timestamps per device
GET	`/api/devices/<device_id>/latest`	Latest datapoint for device
GET	`/api/comparison`	Compare two devices over time; `metric`, `device_1`, `device_2`
GET	`/api/thresholds`	Read thresholds
POST	`/api/thresholds`	Update thresholds
GET	`/api/alert_email`	Get configured alert email
POST	`/api/alert_email`	Set alert email (sends confirmation mail)
POST	`/api/confirm_email`	Confirm alert email with token
POST	`/api/send_alert_mail`	Send threshold alert mail and manage cooldown
GET	`/health`	Health check

Additional API docs:

- High-level API notes: `docs/Backend/api.md`
- Alerts and thresholds: `docs/Backend/alerts.md`, `docs/Backend/thresholds_alerting.md`

MQTT Ingestion

The ingester subscribes to sensor topics, validates messages, maps them to metrics, and writes to the DB.

- Transport/topic parsing and JSON decoding: `backend/mqtt_client/main_ingester.py`
- Payload validation and DB writes: `backend/mqtt_client/handler.py`, `backend/mqtt_client/db_writer.`
- Configuration (env variables): `backend/mqtt_client/mqtt_config.py`
- Detailed ingestion documentation: `docs/Backend/mqtt.md`

Expected topic format, payload schema, metric mapping, and error handling are described in `mqtt.md` and implemented across the files above.

Database Layer

The API uses a thin DB layer under `backend/api/db/` with consistent error mapping and structured logging.

- Connection and config checks: `connection.py`
- Validation helpers: `validation.py`
- Devices: `devices.py`
- Time ranges: `time_ranges.py`
- Device data: `device_data.py`
- Latest data: `device_latest.py`
- Comparison/aggregation: `comparison.py`
- Thresholds CRUD: `thresholds.py`
- Alert email storage/cooldowns: `alertMail.py`, `sendAlertMail.py`
- Row serialization: `serialization.py`

Schema and initialization:

- DB schema overview: `docs/DB/db.md`
- Init scripts: `db/init.sql`, availability helper: `db/availability_sensor.sql`
- ADR: TimescaleDB decision: `docs/adr/0008-use-timescaledb-for-db.md`

Logging and Error Handling

- Structured JSON logging via Loguru sink: `backend/common/logging_setup.py`
- Unified exceptions for API, DB, MQTT, and ingestion domains: `backend/common/exceptions.py`
- Operational logging and monitoring decisions: `docs/software-quality/logging-monitoring.md`, `docs/software-quality/logging.md`

Configuration

Configuration is sourced from environment variables. Notable keys:

- Database: `DB_HOST`, `DB_PORT`, `DB_NAME`, `DB_USER`, `DB_PASSWORD` (see `connection.py`)
- MQTT: `MQTT_BROKER`, `MQTT_PORT`, optional `MQTT_BROKER_BACKUP`, `MQTT_PORT_BACKUP`, `MQTT_BASE_TOPIC`, `MQTT_QOS` (see `mqtt_config.py`)
- Alert mail (Grafana SMTP relays): `GF_SMTP_HOST`, `GF_SMTP_USER`, `GF_SMTP_PASSWORD`, `GF_SMTP_FROM`, `GF_SMTP_FROM_NAME` (see `sendAlertMail.py`)
- Frontend URL for confirmation links: `FRONTEND_URL` (see `alertMail.py`)

`.env` is supported locally by the MQTT ingester; containers typically use environment variables (see `USE_DOTENV` in `mqtt_config.py`).

Running and Deployment

- Local dev API: `python backend/run.py` (listens on port 5001 by default)
- Docker images:
  - API service Dockerfile: `backend/Dockerfile.api`
  - MQTT ingester Dockerfile: `backend/Dockerfile.mqtt`

Testing

Backend tests live under `backend/tests/`. See also `docs/Backend/tests.md`.

Related Documentation

- Frontend overview: `docs/Frontend/frontend.md`
- Backend API notes: `docs/Backend/api.md`
- MQTT ingestion: `docs/Backend/mqtt.md`
- Alerts and thresholds: `docs/Backend/alerts.md`, `docs/Backend/thresholds_alerting.md`
- Architecture diagrams: `docs/architecture/`, images under `docs/images/`
- ADRs (decisions): `docs/adr/`

---

backend/mqtt.md

Sensor data validation (MQTT ingestion)

This document summarizes how incoming MQTT messages are validated and written to the database by the ingester.

Topic format

- Expected: `dhbw/ai/si2023/<group>/<sensor-type>/<sensor-id>`
- Example: `dhbw/ai/si2023/01/temperature/01`
- Mapping to metric (from `backend/mqtt_client/main_ingester.py`):
  - `ikea/01` → `pollen`
  - `ikea/02` → `particulate_matter`
  - `temperature/01` → `temperature`
- If the topic has too few segments or no mapping exists, the message is skipped and a warning log with reason `schema_mismatch` is emitted.

Payload schema

- JSON body example:

```
{
  "value": 22.5,
  "timestamp": "1722945600",
  "meta": { "device_id": 1 }
}
```

- Required fields:
  - `meta.device_id` (integer, positive)

  - `timestamp` (epoch seconds as string/integer; must parse to an integer)
  - `value` (numeric)
- Failures emit a warning log with reason `schema_mismatch` and the message is skipped.

Metric handling and value types

- Supported metrics and ranges (inclusive):
  - `temperature`: 1–40 (float allowed)
  - `humidity`: 1–100 (float allowed)
  - `pollen`: 1–700 (integer only; `12.0` is accepted and cast to `12`, `12.3` is rejected)
  - `particulate_matter`: 1–700 (integer only; `12.0` ok, `12.3` rejected)
- Non-numeric `value` → warning with reason `schema_mismatch`.
- Out-of-range `value` → warning with reason `min_max_check`.

Timestamp normalization

- The `timestamp` is parsed from epoch seconds to a `datetime` and logged in ISO-8601 UTC `...Z` format.

Database write behavior

- Only the current metric is passed to the DB; other fields are `None` for this write.
- `insert_sensor_data` performs an upsert with `COALESCE`, preserving existing values when `None` is provided.
- On success, a single info log `msg_processed` is emitted.

Error mapping (DB)

- Transient/driver messages containing "timeout/timed out" → `DatabaseTimeoutError` → error log `db_write_failed` with reason `timeout`.
- Generic DB failures → `DatabaseError` → error log `db_write_failed` with reason `db_error`.
- Rollback errors are swallowed; the original DB error mapping is preserved.

Examples

- Valid temperature message:

```
{"value": 23.4, "timestamp": "1722945600", "meta": {"device_id": 2}}
```

- Invalid value type (rejected):

```
{"value": "NaN", "timestamp": "1722945600", "meta": {"device_id": 2}}
```

- Invalid range (rejected):

```
{"value": 1000, "timestamp": "1722945600", "meta": {"device_id": 2}}
```

- Integer metric with non-integer float (rejected):

```
{"value": 12.3, "timestamp": "1722945600", "meta": {"device_id": 2}}
```

Responsibilities: main_ingester vs handler

- `backend/mqtt_client/main_ingester.py` (transport-layer checks)
  - Validate/parse topic: check segment count; map `<sensor-type>/<sensor-id>` to a supported metric.

  - Decode JSON: if decoding fails, skip the message and log WARNING with reason `schema_mismatch`.
  - Pass `topic` and decoded `payload` to the handler for domain validation and persistence.

- `backend/mqtt_client/handler.py` (domain validation and persistence)
  - Parse and validate required payload fields: `meta.device_id`, `timestamp`, `value`.
  - Type and range rules:
    * Floating-point metrics (`temperature`, `humidity`) accept floats and must be within range.
    * Integer metrics (`pollen`, `particulate_matter`) accept values like `12.0` (cast to `12`) but reject `12.3`; must be within range.
  - Timestamp normalization: epoch → `datetime`; logs use ISO-8601 UTC.
  - Database write: write only the current metric; pass `None` for others so DB-side `COALESCE` preserves previous values.
  - Error mapping and logging: map driver/timeout errors to domain errors and emit structured logs; rollback failures are swallowed while preserving the original error mapping.

Related implementation files

- Topic/JSON validation and routing: `backend/mqtt_client/main_ingester.py`
- Payload parsing, type/range validation, DB write: `backend/mqtt_client/handler.py`
- Upsert/COALESCE details: `backend/mqtt_client/db_writer.py`

---

backend/tests.md

Backend Tests Overview

This document summarizes how backend unit tests are structured and executed.

Goals

- Validate HTTP API behaviors (status codes, response shapes, messages)
- Validate MQTT ingestion logic (payload validation, metric mapping, DB write calls)
- Keep tests fast and deterministic by mocking external systems (DB, SMTP, MQTT broker)
- Assert structured logging for key paths (success, warnings, errors)

Technology

- Test runner: pytest (see `backend/pytest.ini`)
- Mocking: `pytest-mock` via the built-in `mocker` fixture
- Coverage: `pytest-cov`

Layout

- API tests: `backend/tests/api/`
  - Examples: `test_device_data.py`, `test_comparison.py`
- MQTT tests: `backend/tests/mqtt_client/`
  - Examples: `test_handler.py`, `test_db_writer.py`, `test_main_ingester.py`
- Shared fixtures: `backend/tests/conftest.py`

Fixtures and Isolation

- App and client fixtures create a fresh Flask app per test and expose a test client:
  - See `conftest.py` for `app` and `client`
- DB interactions are mocked at the module boundary (e.g., `api.device_data.get_device_data_from_db`), so no real database is needed
- Logs are asserted by patching `log_event` and inspecting calls
- SMTP and other I/O are patched similarly in alert-related tests

Typical Assertions

- HTTP: response status, JSON structure, error messages
- MQTT handler: correct coercion/validation; DB write function called/not called; log events include expected `level` and `event`
- Error mapping: domain errors map to expected HTTP codes or logged reasons

CI

Tests are designed to run in CI (see `docs/workflows/ci.md`). Use `pytest -q` with coverage flags as needed.

---

backend/thresholds_alerting.md

Alerting Logic Documentation

Overview

The alerting system notifies users via email when sensor values exceed configured thresholds. To prevent spam, a **state-based cooldown** is implemented:

- Only one alert email is sent while the value remains outside the threshold.
- The cooldown is reset only when the value returns to the normal range.

How It Works

1. **Frontend triggers `/send_alert_mail`** for every new value.
2. **Backend checks thresholds:**
   - If the value exceeds a ”hard” or ”soft” threshold, it checks if an alert is already active for that metric and device.
   - If no alert is active, an email is sent and the alert is marked as active (in the database).
   - If an alert is already active, no email is sent (cooldown).
3. **Cooldown reset:**
   - When the value returns to the normal range (i.e., within all thresholds), the alert is reset.
   - The next time the value exceeds a threshold, a new email will be sent.

Email Confirmation

- **Double-Opt-In:**
  Before any alert emails are sent, the user must confirm their email address via a confirmation link sent to their inbox.
  - When a new email is entered, a confirmation email is sent.
  - Alerts are only sent to confirmed email addresses.

Email Content

- **Subject:**
  `[HARD] Alert: Device Altbau - Temperature`
- **Body:**

```
ALERT (HARD)

Affected Device: Altbau
Sensor/Metric: Temperature

Current Value: 35 °C

Thresholds:
  Red Low: 10°C
  Yellow Low: 15°C
  Yellow High: 30°C
  Red High: 32°C

The current value for Temperature has exceeded the HARD threshold.
Please check the air quality and ventilate the rooms or take further action if necessary.
```

Example Workflow

1. Value rises above `redHigh` (e.g., 35 > 32):
   → Email is sent, alert is active.
2. Value remains above `redHigh` (e.g., 36):
   → No email sent, cooldown active.
3. Value drops below `redHigh` (e.g., 30):
   → Cooldown is reset.
4. Value rises above `redHigh` again (e.g., 33):
   → New email is sent.

Logging

The backend logs every alert event:

- When an email is sent: `alert_mail.sent`
- When the cooldown is active: `alert_mail.cooldown_active`
- When the cooldown is reset: `alert_mail.reset`
- On errors or missing parameters: `alert_mail.missing_parameters`, `alert_mail.unknown_metric`, `alert_mail.process_error`
- When a confirmation email is sent: `alert_email.confirmation_sent`
- When an email is confirmed: `alert_email.confirmed`

Notes

- The alerting logic is **state-based**, not time-based.
- The frontend should call `/send_alert_mail` for every value update to ensure correct cooldown handling.
- All thresholds and alert email addresses can be managed via the API.
- Alert emails are only sent to confirmed addresses (double-opt-in).

---

db/db.md

Database Architecture Documentation

Overview

This document outlines the database schema designed to store and manage environmental sensor data and their corresponding warning thresholds. The architecture leverages TimescaleDB, an extension for PostgreSQL, to efficiently handle time-series data, making it ideal for continuous sensor readings.

The database consists of several primary tables: `sensor_data` for storing raw sensor readings over time, `thresholds` for defining configurable warning and critical limits, and additional tables for alerting logic (`alert_emails`, `alert_cooldowns`).

Initialization & Docker Compose Integration

The database is provisioned using the official TimescaleDB Docker image.
In the `docker-compose.yml`, the database service is defined as follows:

```yaml
db:
  image: timescale/timescaledb:latest-pg14
  container_name: altbau_vs_neubau_db
  environment:
    - POSTGRES_USER=${DB_USER}
    - POSTGRES_PASSWORD=${DB_PASSWORD}
    - POSTGRES_DB=${DB_NAME}
  ports:
    - "5432:5432"
  volumes:
    - ./db/init.sql:/docker-entrypoint-initdb.d/01_init.sql:ro
    - ./db/availability_sensor.sql:/docker-entrypoint-initdb.d/02_availability_sensor.sql:ro
    - db_data:/var/lib/postgresql/data
  networks:
    - pg-network
  healthcheck:
    test: ["CMD-SHELL", "psql -U ${DB_USER} -d ${DB_NAME} -c 'SELECT 1;' || exit 1"]
    interval: 5s
    timeout: 3s
    retries: 10
    start_period: 60s
```

Important:
The `init.sql` script is only executed **once** when the database container is created for the first time (i.e., when the volume `db_data` is empty).
If you restart or recreate the container and the volume already exists, PostgreSQL will **not** re-run the initialization scripts.
Any schema changes or new tables must be created manually using SQL commands.

1. sensor_data Table

This table is the core of the time-series data storage. It's designed to capture readings from various devices at specific timestamps.

Purpose

Stores historical and real-time sensor measurements, including temperature, humidity, pollen count, and particulate matter levels, associated with a specific device and timestamp.

```
CREATE TABLE IF NOT EXISTS sensor_data (
    device_id INT NOT NULL,
    timestamp TIMESTAMPTZ NOT NULL,
    temperature DECIMAL(5, 2),
    humidity DECIMAL(5, 2),
    pollen INT,
    particulate_matter INT,
    PRIMARY KEY (device_id, timestamp)
);

-- Convert to hypertable for TimescaleDB
SELECT create_hypertable('sensor_data', 'timestamp', if_not_exists => TRUE);
```

Key Characteristics

- `device_id` (INT, NOT NULL): Unique identifier for the sensor device. Part of the composite primary key.
- `timestamp` (TIMESTAMPTZ, NOT NULL): The exact time the sensor reading was taken, including timezone information. Also part of the composite primary key and the time-series dimension for TimescaleDB.
- `temperature`, `humidity` (DECIMAL(5, 2)): Sensor readings with two decimal places.
- `pollen`, `particulate_matter` (INT): Integer sensor readings.
- Primary Key: Composite on `(device_id, timestamp)` ensures one reading per device per moment.

2. thresholds Table

This table provides a flexible mechanism to define and manage warning and critical limits for the sensor readings.

Purpose

Stores configurable ”soft” (warning) and ”hard” (critical) thresholds for each environmental parameter. These values can be used by the application to trigger alerts or visualize data against defined limits.

Schema

```
CREATE TABLE IF NOT EXISTS thresholds (
    temperature_min_soft DECIMAL(5, 2),
    temperature_max_soft DECIMAL(5, 2),
    temperature_min_hard DECIMAL(5, 2),
    temperature_max_hard DECIMAL(5, 2),
    humidity_min_soft DECIMAL(5, 2),
    humidity_max_soft DECIMAL(5, 2),
    humidity_min_hard DECIMAL(5, 2),
    humidity_max_hard DECIMAL(5, 2),
    pollen_min_soft INT,
    pollen_max_soft INT,
    pollen_min_hard INT,
```

```
    pollen_max_hard INT,
    particulate_matter_min_soft INT,
    particulate_matter_max_soft INT,
    particulate_matter_min_hard INT,
    particulate_matter_max_hard INT,
    last_updated TIMESTAMPTZ DEFAULT CURRENT_TIMESTAMP
);
```

Key Characteristics

- *_min_soft / *_max_soft: Lower and upper bounds for "soft" warnings.
- *_min_hard / *_max_hard: Lower and upper bounds for "hard" critical alerts.
- `last_updated`: Timestamp of last modification.

3. alert_emails Table

Stores email address for alert notifications, including confirmation status and tokens for double-opt-in.

```
CREATE TABLE IF NOT EXISTS alert_emails (
    email VARCHAR(255) NOT NULL,
    confirmed BOOLEAN DEFAULT FALSE,
    confirmation_token VARCHAR(64),
    PRIMARY KEY (email)
);
```

4. alert_cooldowns Table

Tracks cooldowns for alert notifications to avoid spamming users.

```
CREATE TABLE IF NOT EXISTS alert_cooldowns (
    device VARCHAR(50) NOT NULL,
    metric VARCHAR(50) NOT NULL,
    mail_type VARCHAR(10) NOT NULL,
    last_sent TIMESTAMPTZ NOT NULL,
    PRIMARY KEY (device, metric, mail_type)
);
```

Data Initialization

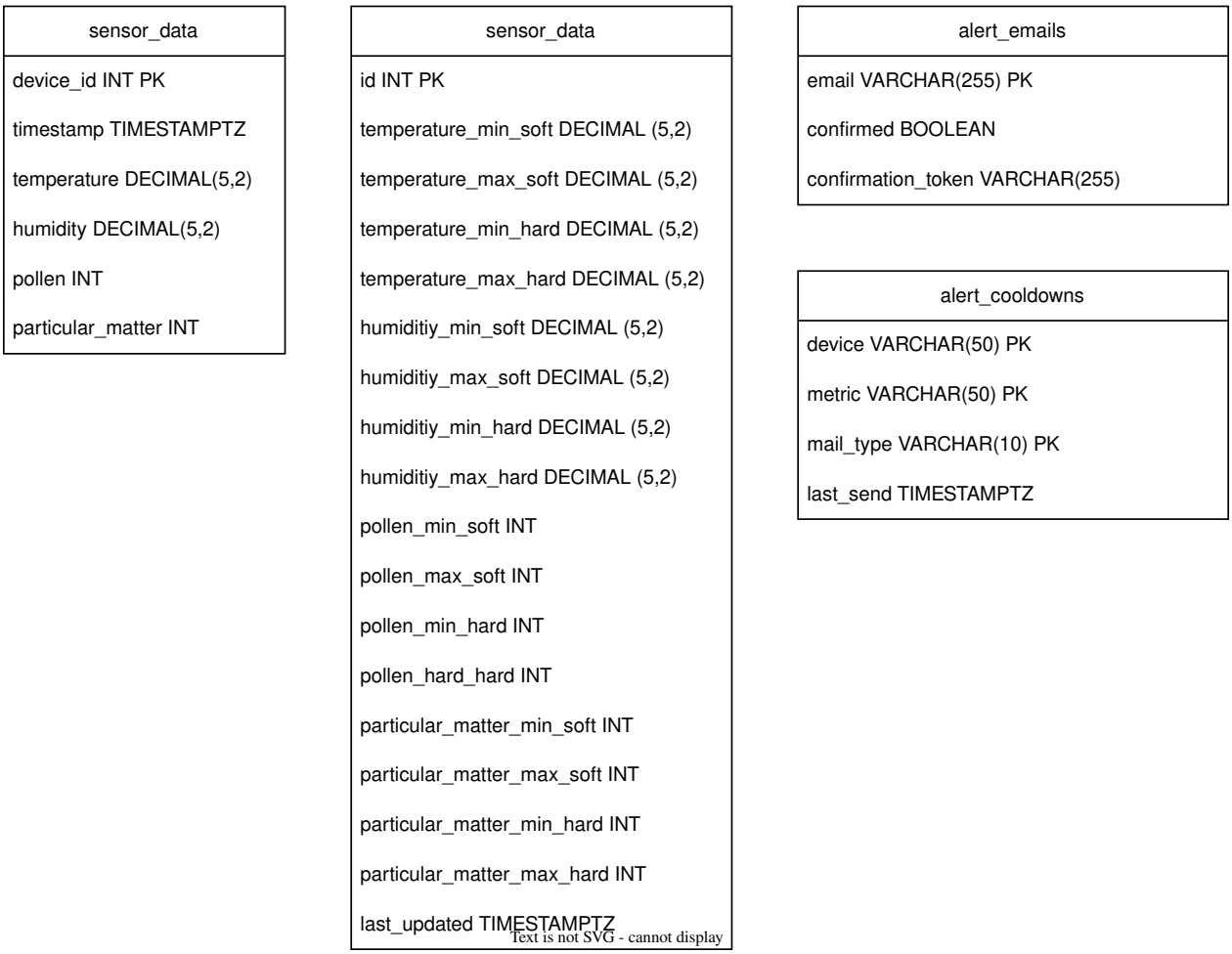
The `init.sql` script also inserts default threshold values:

```
DELETE FROM thresholds;
INSERT INTO thresholds (temperature_min_hard, temperature_min_soft, temperature_max_soft, tempe
                        humidity_min_hard, humidity_min_soft, humidity_max_soft, humidity_max_h
                        pollen_min_hard, pollen_min_soft, pollen_max_soft, pollen_max_hard,
                        particulate_matter_min_hard, particulate_matter_min_soft, particulate_m
(15.00, 18.00, 26.00, 32.00,
 30.00, 40.00, 60.00, 80.00,
 0, 5, 30, 80,
 0, 20, 50, 70);
```

Manual Schema Changes

Note:
If you need to add, remove, or modify tables after the initial container creation, you must do so manually using SQL commands (e.g., via `psql` or a database GUI).
Re-running the container will **not** apply changes from `init.sql` unless the database volume is deleted and recreated.

Visual Representation of the Database Structure



Summary of Docker Compose Integration

- The database is deployed as a TimescaleDB container.
- Initialization scripts are only run once, on first volume creation.
- All other services (API, MQTT, Frontend, Monitoring) connect to the database via the defined Docker network (`pg-network`).
- Persistent data is stored in the `db_data` volume.
- Healthchecks ensure the database is ready before dependent services start.

---

frontend/frontend.md

Altbau vs Neubau – Frontend Documentation

Overview

This React frontend visualizes sensor data and warning thresholds for the ”Altbau vs Neubau” project. It provides a dashboard for live and historical metrics, as well as a page for viewing and managing warning values.

Detailed Page Documentation

- Dashboard Page
- Warnings / Thresholds Page
- Confirm Email Page

Structure

- **Main Entry:** `src/App.js`
- **Pages:**
  - `src/pages/Dashboard.jsx` – Main dashboard with charts and metric selection
  - `src/pages/Warnings.jsx` – Warning thresholds overview and management
  - `src/pages/ConfirmEmail.jsx`- Page for email confirmation (Double opt in)
- **Assets:** Images and static files in `src/assets/`
- **Components:** Components for Pages in `src/components/`
- **Styling:** Global styles in `src/App.css` and component-specific classes

Routing

Uses react-router-dom for navigation:

Path	Component	Description
`/`	Dashboard	Main dashboard view
`/warnings`	Warnings	Warning values management
`/confirm-email`	ConfirmEmail	Confirm Email for alerting

Main Features

- **Header:** Displays project title and logo.
- **Dashboard:** Interactive charts for metrics (temperature, humidity, pollen, particulate matter) with interval selection and per-chart line toggling.
- **Warnings:** Show and set current warning thresholds.
- **Footer:** Contains copyright and contact-link

Data Flow

- Sensor and threshold data are fetched from the backend API.
- State management is handled via React hooks.
- Changes to thresholds or email addresses are validated and sent to the backend.

Key Components

- `CurrentValuesTable`: Shows latest sensor readings.
- `ChartCard` & `ChartModal`: Interactive charts for historical data.
- `IntervalButtons`: Select time intervals for charts.
- `WarningsForm`: Edit and save warning thresholds.
- `ErrorMessage` & `LoadingIndicator`: User feedback and loading states.

Screenshots

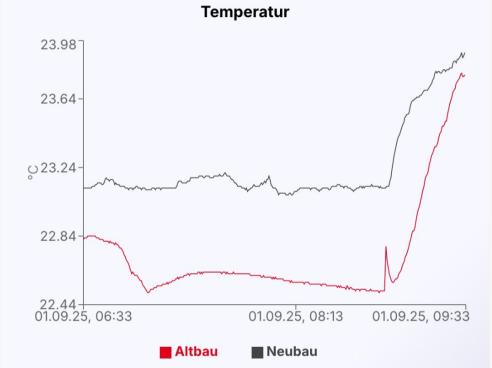
Aktuelle Messwerte

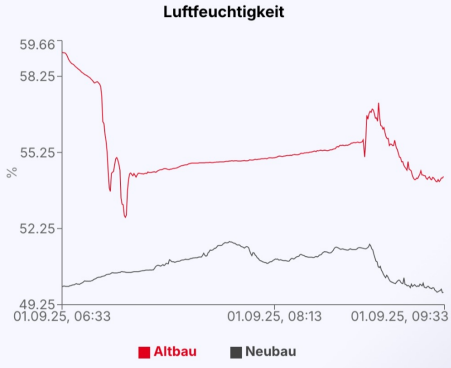
Metrik	Altbau	Neubau
Temperatur	23.78 °C	23.91 °C
Luftfeuchtigkeit	54.3 %	49.73 %
Pollen	—	7 µg/m³
Feinstaub	—	35 µg/m³
Zeitstempel	01.09.25, 09:33	01.09.25, 09:33

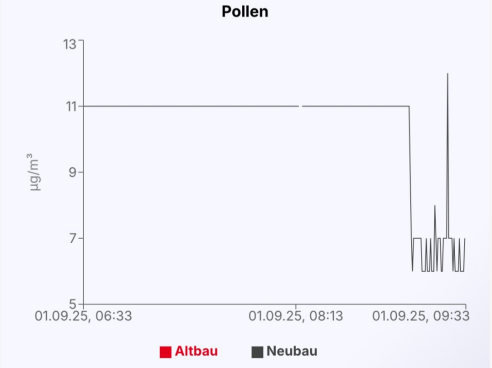
Warnungen ändern

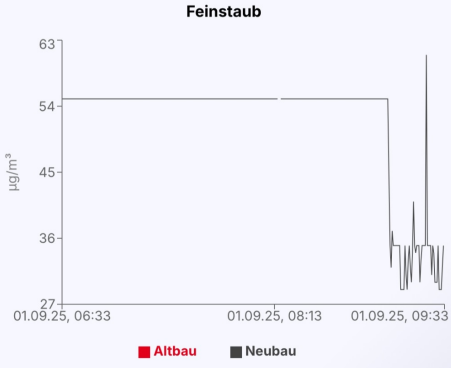
Verlauf

30 Minuten | 1 Stunde | 3 Stunden | 6 Stunden | 12 Stunden | 1 Tag | 1 Woche | 1 Monat









Development Notes

- All pages and components are modular and easy to extend.
- Use `src/components/` for reusable UI elements.
- For new features, add components and update routing in `App.js`.

---

frontend/frontend_confirm-email.md

Confirm Email Page Documentation

Overview

The **Confirm Email Page** is a dedicated interface for users to complete the double-opt-in process for alert notifications.
When a user enters a new alert email address on the Warnings/Thresholds page, a confirmation email is sent containing a unique token.
The user must visit the Confirm Email page (usually via a link in the email) to verify their address and activate alert notifications.

Main Features

- **Token Verification:**
  The page reads the confirmation token from the URL query parameters and sends it to the backend API for validation.
- **Status Feedback:**
  The page displays clear feedback based on the confirmation result:
  - Pending: ”Bestätige deine E-Mail-Adresse...”
  - Success: ”Deine E-Mail-Adresse wurde erfolgreich bestätigt. Du erhältst nun Alerts.”
  - Error: ”Bestätigung fehlgeschlagen.” (e.g., invalid or expired token)
- **Navigation:**
  After confirmation, users can return to the dashboard with a single click.

Technical Details

- **Component:**
  - `src/pages/ConfirmEmail.jsx`
- **Logic:**
  - Uses React hooks (`useEffect`, `useState`) to handle token extraction and API communication.
  - Utilizes `useSearchParams` from `react-router-dom` to read the token from the URL.
  - Calls the backend endpoint `/confirm_email` via POST with the token.
  - Displays status messages based on the API response.
- **Styling:**
  - Uses global styles and a visually distinct info box for feedback.
  - Button for navigation back to the dashboard.

User Experience

- **Guided Confirmation:**
  Users are informed about the confirmation status at every step.
- **Error Handling:**
  Invalid or missing tokens result in clear error messages.
- **Simple Navigation:**
  After confirmation, users can easily return to the main dashboard.

Example Workflow

1. User enters a new email address on the Warnings page.
2. System sends a confirmation email with a link:
   `https://your-domain/confirm-email?token=abc123`
3. User clicks the link and lands on the Confirm Email page.
4. The page verifies the token and displays the result.
5. On success, alerts will be sent to the confirmed email address.

Summary

The Confirm Email Page ensures that only verified email addresses receive alert notifications, improving security and user trust.
It is a key part of the double-opt-in flow and provides a clear, user-friendly confirmation experience.

For more details on the frontend structure, see frontend.md.

---

frontend/frontend_dashboard.md

Dashboard Page Documentation

Overview

The **Dashboard Page** is the central hub of the ”Altbau vs Neubau” frontend.
It provides users with a clear, interactive overview of all relevant sensor data, including current values and historical trends for both the ”Altbau” and ”Neubau” buildings.
The dashboard is designed for quick status checks, in-depth analysis, and direct access to warning threshold management.

Main Features

1. Current Values Table

- Displays the latest sensor readings for each metric (Temperature, Humidity, Pollen, Particulate Matter) for both buildings.
- Highlights values that exceed warning or critical thresholds using color-coded cells.
- Shows the timestamp of the most recent measurement per building.
- If a sensor is unavailable, the table indicates this clearly.

2. **Interactive Charts**

- For each metric, a line chart visualizes historical data for both buildings over a selectable time interval.
- Users can toggle the visibility of each building's data series via the legend.
- Charts automatically insert gaps if data is missing, making outages or sensor failures visible.
- Tooltips provide precise values and timestamps on hover.
- Clicking a chart opens a modal with a larger, more detailed view.

3. **Interval Selection**

- Users can choose the time range for historical data (e.g., 30 minutes, 1 hour, 1 day, 1 week, 1 month).
- Interval selection updates all charts and data accordingly.

4. **Warning Thresholds Integration**

- Thresholds are fetched from the backend and used to color-code both the current values and chart lines.
- A button provides direct navigation to the warning management page.

5. **Error Handling & Feedback**

- If data cannot be loaded, the dashboard displays clear error messages.
- Loading states are shown while data is being fetched.

Technical Details

- **Component Structure:**
  - `Dashboard.jsx` orchestrates data fetching, state management, and layout.
  - `CurrentValuesTable` renders the latest sensor values.
  - `ChartCard` and `ChartModal` handle chart rendering and modal display.
  - Utility functions in `dashboardUtils.js` manage data mapping, interval calculation, and formatting.
  - `IntervalButtons`, `CustomLegend`, and `CustomTooltip` provide interactive UI elements.
- **Data Flow:**
  - Fetches current and historical sensor data from the backend API.
  - Fetches warning thresholds and applies them for visual feedback.
  - Sends alert mail requests when current values exceed thresholds.
- **Styling:**
  - Uses global and page-specific CSS for layout, responsiveness, and color coding.
  - Charts are rendered with Recharts.

User Experience

- **Quick Status:**
  Instantly see if any metric is in a warning or critical state.

- **Analysis:**
  Explore historical trends and compare buildings side-by-side.
- **Action:**
  Navigate to warning management to adjust thresholds as needed.

Summary

The Dashboard Page is designed to be the main entry point for users, combining real-time monitoring, historical analysis, and actionable insights in a single, intuitive interface.
It supports the overall goal of the project: making building sensor data transparent, actionable, and easy to manage.

For more details on the frontend structure, see frontend.md.

---

frontend/frontend_requirements.md

Frontend Requirements

Functional Requirements

1. **Display of Current Measurements**
   - Show current values for the following metrics:
     - Temperature
     - Humidity
     - Pollen load
     - Fine dust (particulate matter)
   - Values are updated at short intervals.
2. **Visualization of Historical Data**
   - Display time series data as line charts.
   - Select different time ranges (e.g., last hour, 24 h, week, month).
   - Compare two devices (Arduino with three sensors)
3. **Customizable Warning Thresholds**
   - Users can define individual thresholds.
   - Exceeding thresholds is visually highlighted (e.g., color change, warning icon).
   - The user can register his email address to receive alert on thresholds via email.
4. **Local Storage of Settings**
   - User-defined warning values and time settings are stored in the database.
   - These are automatically loaded when the application is reopened or every 30 seconds.
5. **Error Messages for System Issues**
   - User-friendly display for:
     - Application unavailability (e.g., network error, backend not available)
     - Errors loading current data
   - Clear indications of the cause and possible solutions.

Technical Requirements

- **Data Visualization:**
  - Use **Recharts** (preferred over Chart.js) for line chart visualization.
- **Authentication:**
  - No authentication will be implemented.
- **API**
  - Easy Access to the sensor data from the database

Wireframe

Home screen

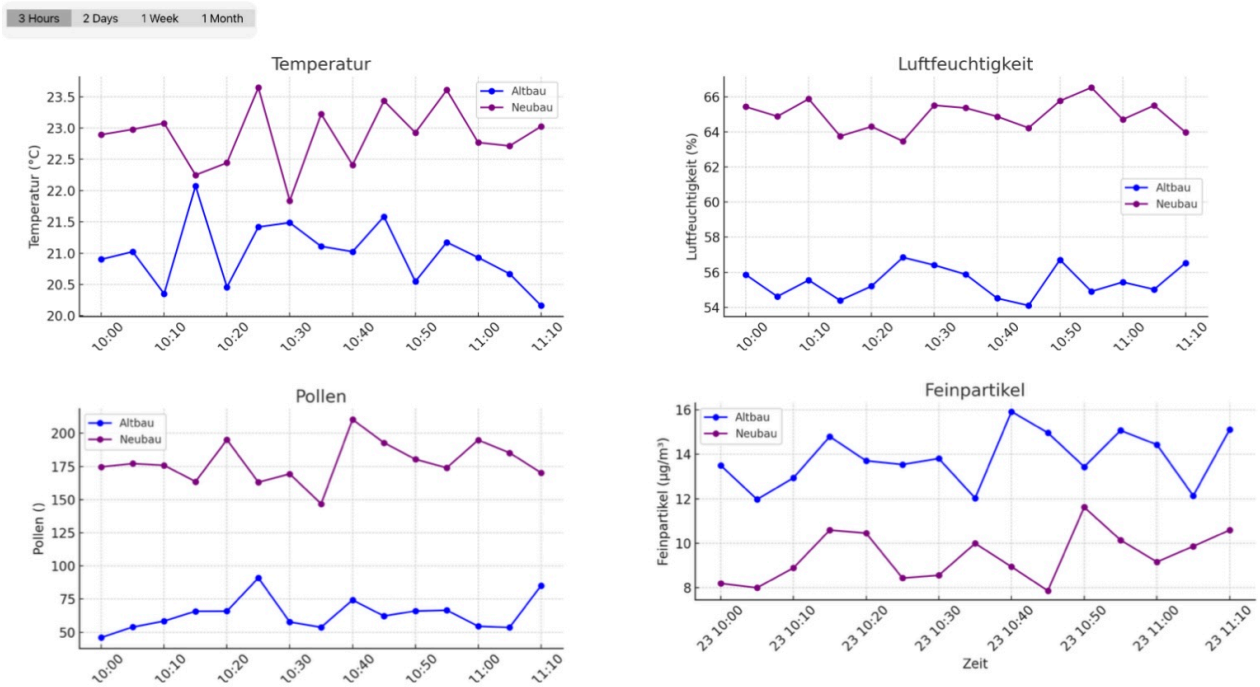
Altbau vs Neubau

Aktuelle Messwerte

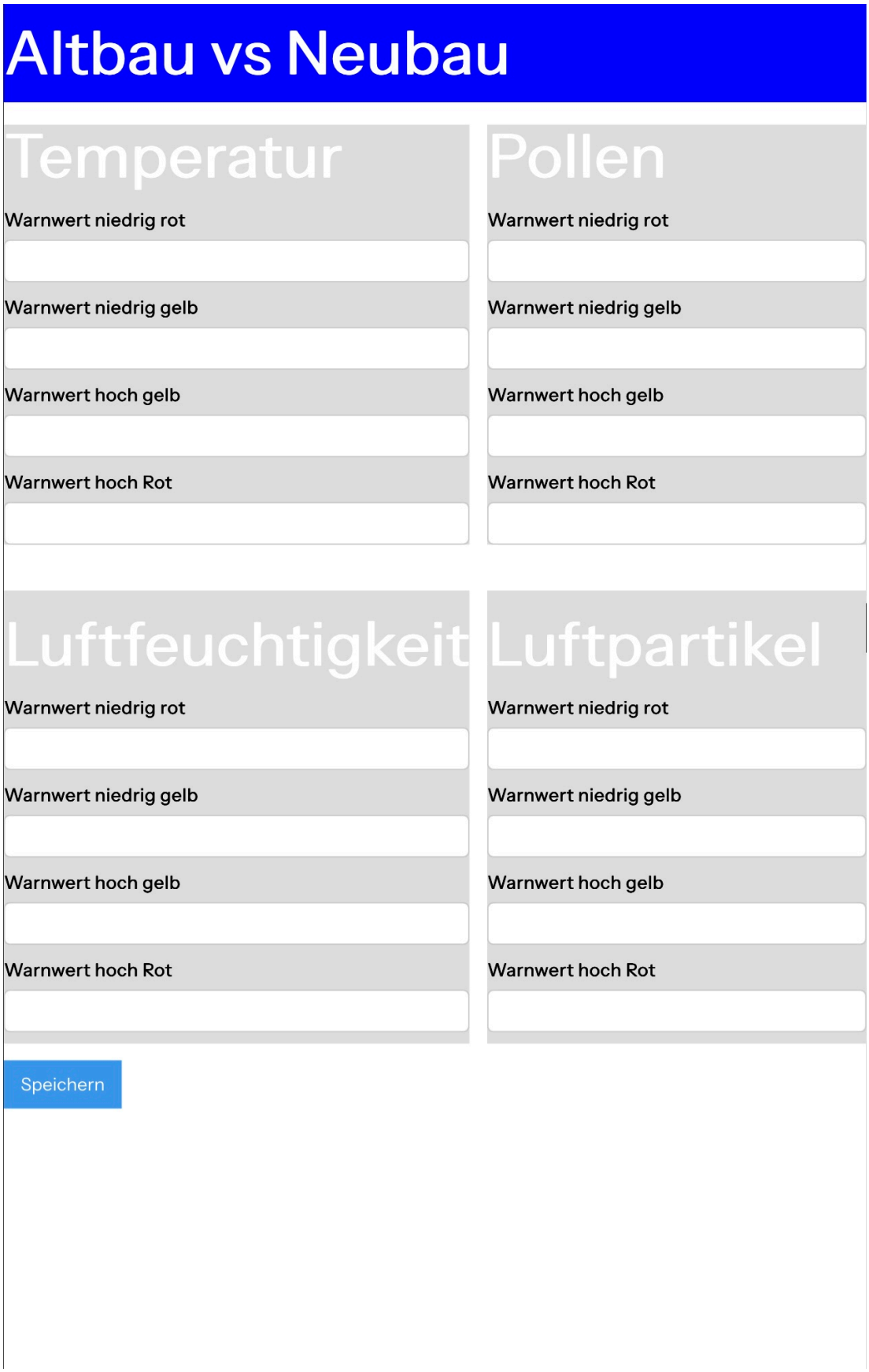
Metriken	Altbau	Neubau
Temperatur	20°	22°
Luftfeuchtigkeit	55 %	65 %
Pollen	50	200
Feinpartikel	15 µg/m³	8 µg/m³

Ändere Warnungswerte

Verlauf



Threshold screen



---

frontend/frontend_warnings.md

Warnings / Thresholds Page Documentation

Overview

The **Warnings Page** (also called the Thresholds Page) is the central interface for managing alert thresholds and notification settings in the ”Altbau vs Neubau” frontend.
It allows users to view, adjust, and save warning and critical limits for all monitored metrics (Temperature, Humidity, Pollen, Particulate Matter) and to configure the alert email address for notifications.

Main Features

1. Threshold Management Form

- Displays all relevant metrics with their current warning (”yellow”) and critical (”red”) thresholds.
- Each metric (e.g., Temperature, Humidity) shows four editable fields:
  - Red Low: Critical lower limit
  - Yellow Low: Warning lower limit
  - Yellow High: Warning upper limit
  - Red High: Critical upper limit
- Values can be adjusted using input fields with increment/decrement buttons for convenience.
- Validation ensures logical consistency (e.g., red low < yellow low < yellow high < red high).

2. Alert Email Configuration

- Users can enter or update the email address to receive alert notifications.
- Implements double-opt-in: When a new email is entered, a confirmation mail is sent and alerts are only sent to confirmed addresses.
- The page provides clear feedback if confirmation is required.

3. Form State & Feedback

- The form tracks unsaved changes (”dirty” state) and disables navigation until changes are saved or discarded.
- Error messages are shown for invalid input, failed saves, or unsaved changes.
- Informational messages guide the user through confirmation steps and successful saves.
- Loading indicators are displayed while thresholds or email settings are being fetched.

4. Navigation

- Includes a button to return to the Dashboard, which is only enabled when there are no unsaved changes.

Technical Details

- **Component Structure:**
  - `Warnings.jsx` manages data fetching, state, validation, and save logic.
  - `WarningsForm.jsx` renders the form and handles user input.
  - `WarningCard.jsx` displays threshold fields for each metric.
  - `NumberInputWithButtons.jsx` provides enhanced input controls.
  - Utility functions in `warningsUtils.jsx` handle mapping, validation, and formatting.
  - `ErrorMessage.jsx` and loading components provide user feedback.
- **Data Flow:**
  - Fetches current thresholds and alert email from the backend API on load.
  - Maps API data to UI format and vice versa for editing and saving.
  - Validates input before saving; only sends confirmation mail if the email address has changed.
  - Saves thresholds and email settings via API calls.
- **Styling:**
  - Uses global and page-specific CSS for layout, responsive design, and color coding.
  - Error and info messages are visually distinct for clarity.

User Experience

- **Intuitive Editing:**
  Users can quickly adjust thresholds with clear labels and input controls.
- **Safe Changes:**
  Unsaved changes are tracked, preventing accidental navigation.
- **Guided Alerts Setup:**
  The email confirmation process is transparent and user-friendly.
- **Immediate Feedback:**
  Errors and info messages keep users informed at every step.

Summary

The Warnings / Thresholds Page empowers users to customize alerting behavior and notification settings, ensuring the system matches their needs and preferences.
It is designed for clarity, safety, and ease of use, supporting the project's goal of actionable, user-driven building monitoring.

For more details on the frontend structure, see frontend.md.

---

software-quality/branchingStrategy.md

Git Branching Strategy

This document proposes a Git branching strategy for structured collaboration within the project.

---

1. Motivation

In multi-module projects with separate responsibilities (e.g. Arduino, Backend, Frontend, Database), a clearly defined branching strategy helps to:

- avoid merge conflicts,
- maintain a clean and stable `main` branch,
- improve collaboration via clear workflows.

---

2. Strategy selection

The following models were evaluated:

- **Git Flow**: full-featured but complex; more suitable for long release cycles. https://www.atlassian.com/gi workflows/gitflow-workflow
- **GitHub Flow**: lightweight and CI/CD-friendly. https://docs.github.com/en/get-started/using-github/github-flow
- **Trunk-Based Development**: promotes short-lived branches and continuous integration. https://trunkbaseddevelopment.com/

Among these, **a combination of GitHub Flow and Trunk-Based Development** is proposed, as it allows:

- short feature branches for parallel development,
- the ability of integrating with CI tools
- clear merge and review processes.

Rules:

- All development is done on short-lived branches derived from `main`.
- Each change is reviewed and merged via Pull Request.
- CI checks (tests, linters) must pass before merging.

---

3. Branch naming

Type/action-module

Type	Example	Use case
`feature`	`feature/add-backend-api`	New functionality
`fix`	`fix/mqtt-reconnect`	Bug fix
`docs`	`docs/update-readme`	Documentation
`test`	`test/add-backend-tests`	Testing-related change
`chore`	`chore/update-gitignore`	Setup or config change

Use lowercase and dashes.

---

4. Developer workflow

1. Create a feature branch from `main`:

```
git checkout main
git pull origin main
git checkout -b feature/<task-name>
```

2. Git commit and push the branch to GitHub:

```
git commit
git push origin feature/<task-name>
```

3. Create a Pull Request in github
4. Code Review
5. After successful review and test, merge and delete the branch:

```
git branch -d feature/<task-name>
git push origin --delete feature/<task-name>
```

---

5. Merge and CI

- `main` should be a protected branch (no direct pushes)
- All changes must go through PRs
- Merges are only allowed after successful CI
- ”Squash and merge” is recommended to keep history clean

---

software-quality/exceptions.md

Error & Exception Handling Guidelines (Foundational)

Purpose

Establish a coherent strategy for classifying, raising, propagating, and presenting errors, aligned with the custom exception hierarchy in `backend/exceptions.py`.

Objectives

- Provide stable semantic categories (Validation, Not Found, Database, MQTT, Timeout, Generic).
- Preserve internal diagnostic fidelity while exposing sanitized responses.
- Facilitate predictable frontend handling via `error_type`.
- Enable targeted remediation and monitoring.

Taxonomy Principles

- One class per distinct semantic meaning, not per endpoint.
- Hierarchy depth minimized to avoid cognitive load.
- Shared base (`AppError`) supplies status code and message contract.
- Specialized subclasses refine context without leaking implementation specifics.

Classification Rules

- ValidationError: Input structure, type, range, or format issues detected early.
- NotFoundError: Definitive absence of a requested resource (no ambiguity).
- Database* Errors: Failures at persistence boundary (connectivity, timeout, generic).
- MQTT* Errors: Broker connectivity or delivery path issues.
- Timeout Errors: True exceeded deadline scenarios only (not generic slowness).
- Generic ApiError: Unexpected but domain-level; still controlled shape.
- AppError fallback: Internal catch-all prior to generic server response.

Raising Strategy

- Validate early; raise specific exceptions before side effects.
- Wrap third-party/library exceptions once at the boundary (translate to domain exception).
- Maintain original exception context internally (logging) but not in outward message.
- Avoid using generic exceptions for convenience when a specific class exists.

Propagation

- Allow domain exceptions to bubble to a single centralized handler.
- Do not intercept and re-wrap with equal semantics (avoid semantic erosion).
- Enforce single response emission (no partial writes followed by raise).

Response Contract

- status: ”error”
- error_type: Canonical class name
- message: Concise, user-comprehensible, action-oriented if applicable
- Optional (future): correlation_id, timestamp, documentation link

Message Guidelines

- Human-readable, bounded length, neutral tone.
- No stack traces, SQL fragments, broker internals, or library identifiers.
- Provide actionable hint for ValidationError (e.g., bounds, expected types).
- Avoid implying existence of other resources in NotFound scenarios.

Logging Interaction

- Exceptions may self-log at instantiation (current design) OR be centrally logged—never duplicate severity lines.
- Severity alignment derives from exception class category.

Frontend Consumption

- Branch logic strictly on `error_type`.
- Provide default fallback UX for unknown future classes (forward compatibility).
- Avoid brittle parsing of message text.

Testing Requirements

- Unit tests assert mapping from raised class to status code and serialized `error_type`.
- Negative tests confirm absence of internal implementation leakage.

- Regression tests ensure new exceptions integrate without altering existing contracts unintentionally.

Evolution & Governance

Adding a new exception requires:

1. Justification (new semantic gap).
2. Assigned and documented status code (avoid overloading existing ones).
3. Update of hierarchy diagram and this guideline.
4. Relevant test coverage and changelog note.

Operational Metrics (Optional Future)

Track counts per `error_type` for trend analysis (e.g., ratio of ValidationError vs DatabaseError) to drive quality improvements.

Anti-Patterns

- Over-granular proliferation (micro-classes with no distinct handling).
- Repurposing a class for unrelated semantics.
- Encoding status codes in message strings.
- Swallowing exceptions silently after logging.
- Using exceptions for nominal control flow.

Success Indicators

- Frontend can adapt display purely from `error_type`.
- Operational dashboards show stable, interpretable error distribution.
- Minimal need to inspect raw logs for routine client-facing issues.
- Low churn in public error contract

Using the Exceptions (Backend)

1. API Resource Layer

Raise domain exceptions directly; do not craft ad-hoc JSON.

```
from exceptions import ValidationError, NotFoundError
from flask_restful import Resource, reqparse

class DeviceLatest(Resource):
    def get(self, device_id: int):
        if device_id <= 0:
            raise ValidationError("device_id must be positive")
        if not device_exists(device_id):
            raise NotFoundError(f"Device {device_id} not found")
        return {"status": "success", "data": get_latest(device_id)}
```

2. Service / Domain Layer

Translate library errors once; propagate typed exception upward.

```
import psycopg2
from exceptions import DatabaseTimeoutError, DatabaseConnectionError, DatabaseError

def run_query(conn, sql, params):
    try:
        with conn.cursor() as cur:
            cur.execute(sql, params)
            return cur.fetchall()
    except psycopg2.errors.QueryCanceled:
        raise DatabaseTimeoutError("Query exceeded time limit")
    except psycopg2.OperationalError as e:
        raise DatabaseConnectionError(str(e))
    except psycopg2.Error as e:
        raise DatabaseError(e.pgerror or str(e))
```

3. MQTT Ingestion

Validate payload early; raise ValidationError; let outer loop decide retry / drop.

```
from exceptions import ValidationError, DatabaseError

def parse_payload(p):
    if "meta" not in p or "device_id" not in p["meta"]:
        raise ValidationError("Missing device_id")
    if "timestamp" not in p or "value" not in p:
        raise ValidationError("Missing timestamp or value")
    return p["meta"]["device_id"], int(p["timestamp"]), p["value"]
```

4. Background / Worker Loop

Catch only high-level base to implement retry/backoff; re-raise unexpected if policy demands.

```
from exceptions import AppError

def process_message(raw):
    try:
        did, ts, val = parse_payload(raw)
        store(did, ts, val)
    except AppError:
        # Already logged; maybe increment metric / discard
        raise
```

5. Global Flask Handler

Single serialization point; no per-endpoint duplication.

```
from exceptions import AppError
from flask import jsonify

@app.errorhandler(AppError)
def handle_app_error(err: AppError):
    return jsonify({
        "status": "error",
        "error_type": err.__class__.__name__,
        "message": str(err)
    }), getattr(err, "status_code", 500)
```

Fallback for uncaught:

```
@app.errorhandler(Exception)
def handle_unexpected(err):
    app.logger.exception("Unhandled exception")
    return jsonify({
        "status": "error",
        "error_type": "InternalServerError",
        "message": "An unexpected error occurred"
    }), 500
```

6. Frontend Handling Pattern

Map `error_type` to user messages; keep default fallback.

```
function mapBackendError(json) {
  if (json.status !== "error") return null;
  switch (json.error_type) {
    case "ValidationError": return "Input invalid.";
    case "NotFoundError": return "Resource not found.";
    case "DatabaseError": return "Temporary backend issue.";
    default: return json.message || "Unknown error.";
  }
}
```

7. Testing Usage

- Mock lower layer to raise each custom exception.
- Assert response status + JSON `error_type`.
- Ensure no stack trace or internal identifiers leak.

```
def test_validation_error(client, mocker):
    mocker.patch("api.device_data.device_exists", side_effect=ValidationError("bad"))
    r = client.get("/api/devices/1/latest")
    assert r.status_code == 400
    assert r.get_json()["error_type"] == "ValidationError"
```

Adding a New Exception (Python)

1. Define

```
class RateLimitError(ApiError):
    """Too many requests in a given window."""
    def __init__(self, message: str = "Too many requests"):
        super().__init__(message, status_code=429)
```

Guidelines:

- Name ends with `Error`.
- Concise one-line docstring.
- Explicit status if diverging from parent.
- Message free of secrets / internal identifiers.

2. (Optional) Custom Logging

Override `__init__` only if log level differs from parent.

3. Raise at Boundary

Translate from external exception at the outermost integration layer only.

4. Serialization

Rely on global handler (do not manually JSONify).

5. Update

Docs + tests + (optional) metrics.

Checklist

- ☐ Correct base class
- ☐ Justified status code
- ☐ Clear message (no sensitive data)
- ☐ Single translation point
- ☐ Tests added
- ☐ Docs updated

Frontend Extension (React)

Goals

- Mirror backend semantic categories.
- Decouple network / parsing failures from UI components.
- Provide consistent user messaging & developer diagnostics.

Frontend Taxonomy

```
Error (native)
└─ AppError (base)
   ├─ ApiError (HTTP/status + backend error_type)
   │   ├─ ValidationError
   │   ├─ NotFoundError
   │   ├─ DatabaseError
   │   ├─ DatabaseTimeoutError
   │   ├─ RateLimitError (optional)
   │   └─ GenericApiError
   ├─ NetworkError (connectivity / CORS / abort)
   ├─ ParseError (invalid JSON / schema mismatch)
   ├─ TimeoutError (client-side abort controller)
   └─ UIStateError (unexpected local state invariants)
```

Class Skeleton

```
// src/errors.js
export class AppError extends Error {
```

```
  constructor(message, meta = {}) {
    super(message);
    this.name = this.constructor.name;
    this.meta = meta;
  }
}

export class NetworkError extends AppError {}
export class TimeoutError extends AppError {}
export class ParseError extends AppError {}

export class ApiError extends AppError {
  constructor(message, status, errorType, meta = {}) {
    super(message, { status, errorType, ...meta });
    this.status = status;
    this.errorType = errorType;
  }
}

export class ValidationError extends ApiError {}
export class NotFoundError extends ApiError {}
export class DatabaseError extends ApiError {}
export class DatabaseTimeoutError extends ApiError {}
export class GenericApiError extends ApiError {}
```

Mapping Backend -> Frontend

Backend error_type	Frontend class
ValidationError	ValidationError
NotFoundError	NotFoundError
DatabaseError	DatabaseError
DatabaseTimeoutError	DatabaseTimeoutError
(others / unknown)	GenericApiError

Fetch Wrapper

Centralize translation; components never call `fetch` directly.

```
// src/api/client.js
import {
  NetworkError, TimeoutError, ParseError,
  ValidationError, NotFoundError,
  DatabaseError, DatabaseTimeoutError,
  GenericApiError
} from "../errors";

const DEFAULT_TIMEOUT_MS = 15000;

function classify(status, errorType, message) {
  switch (errorType) {
    case "ValidationError": return new ValidationError(message, status, errorType);
    case "NotFoundError": return new NotFoundError(message, status, errorType);
    case "DatabaseTimeoutError": return new DatabaseTimeoutError(message, status, errorType);
```

```
    case "DatabaseError": return new DatabaseError(message, status, errorType);
    default:
      if (status === 404) return new NotFoundError(message || "Not found", status, "NotFoundErr
      if (status === 400) return new ValidationError(message || "Invalid input", status, "Valid
      return new GenericApiError(message || "Unexpected error", status, errorType || "GenericAp
  }
}

export async function apiRequest(path, options = {}) {
  const controller = new AbortController();
  const timeout = setTimeout(() => controller.abort(), options.timeout || DEFAULT_TIMEOUT_MS);
  const base = import.meta.env.VITE_API_URL ;
  let res;
  try {
    res = await fetch(base + path, { ...options, signal: controller.signal });
  } catch (e) {
    clearTimeout(timeout);
    if (e.name === "AbortError") throw new TimeoutError("Client-side timeout");
    throw new NetworkError("Network request failed");
  }
  clearTimeout(timeout);

  let body;
  try {
    body = await res.json();
  } catch (e) {
    throw new ParseError("Response JSON invalid");
  }

  if (body.status === "error" || !res.ok) {
    throw classify(res.status, body.error_type, body.message);
  }
  return body;
}
```

Component Usage Pattern

```
async function loadThresholds() {
  setState(s => ({ ...s, loading: true, error: null }));
  try {
    const json = await apiRequest("/thresholds");
    // process json.data
  } catch (e) {
    if (e instanceof ValidationError) setUserMessage("Eingaben ungültig.");
    else if (e instanceof NotFoundError) setUserMessage("Nicht gefunden.");
    else setUserMessage("Technischer Fehler.");
    setDevError(e); // optional debug state
  } finally {
    setState(s => ({ ...s, loading: false }));
  }
}
```

UI Messaging Guidelines

Class	User Message Style
ValidationError	Actionable (what to correct)
NotFoundError	Neutral absence message
NetworkError	Connectivity hint / retry
TimeoutError	Retry suggestion
DatabaseError	Generic technical issue
GenericApiError	Neutral fallback

Error Boundary

Wrap high-level layout to capture render-time UIStateError or unexpected exceptions.

```
class AppErrorBoundary extends React.Component {
  state = { hasError: false };
  static getDerivedStateFromError() { return { hasError: true }; }
  componentDidCatch(err) { console.error("[UI]", err); }
  render() { return this.state.hasError ? <Fallback /> : this.props.children; }
}
```

State Invariant Checks

Throw `UIStateError` (derived from `AppError`) for impossible client-only states; boundary captures and surfaces generic fallback.

Do / Avoid

- Do: Centralize translation (single fetch wrapper).
- Do: Differentiate network vs backend semantic errors.
- Avoid: Branching on raw `status` codes in components.
- Avoid: Surfacing raw backend messages without sanitization (optional refinement layer).
- Avoid: Silent catch without user feedback.

Versioning & Evolution

Add new frontend error subclasses only when backend introduces stable new semantic categories or client-only invariants need distinct handling.

---

software-quality/frontend-loadtest.md

Frontend JMeter Smoke Test Guide

This guide explains how to run the JMeter **smoke test** for the frontend.
It focuses on validating the **Initial Page TTLB (Time-To-Last-Byte)** against an SLO threshold.

---

1. Test Files Overview

File	Purpose
`.env`	Environment variables (target URLs, number of users, ramp-up time, etc.)
`run-jmeter.sh`	Bash script that executes the JMeter test inside Docker
`plan.jmx`	The JMeter test plan (defines test steps, requests, and data collection)
`results/`	Folder where test results and HTML reports are saved
`frontend-test.yml`	GitHub Actions workflow for automated smoke test

---

2. Local Execution

Step 1 — Configure `.env`

Default values:

```
TARGET_URL=http://hrschmllr.de:3000     # Frontend service URL
API_BASE=http://hrschmllr.de/api        # API base URL
USERS=20                                # Number of virtual users
RAMP=30                                 # Ramp-up time in seconds
COMPOSE_NETWORK=frontend_pg-network     # Docker network (optional)
```

Step 2 — Start Your Application

```
docker compose up -d
```

Check reachability:

```
curl http://hrschmllr.de:3000/
curl http://hrschmllr.de/api/health
```

Step 3 — Run the Test

```
cd frontend-loadtest
chmod +x run-jmeter.sh
./run-jmeter.sh
```

This will:

- Clean previous results
- Run JMeter inside Docker in non-GUI mode
- Save results to `results/` and generate an HTML report

Step 4 — View Results

```
xdg-open results/report/index.html
```

---

3. CI/CD Execution (GitHub Actions)

The GitHub Actions workflow runs the smoke test:

- On pull requests to `main`
- On manual trigger via **Actions → Run workflow**

Workflow Steps

1. Checkout repository code
2. Start app stack in Docker Compose
3. Normalize line endings
4. Run JMeter smoke test
5. Upload results (HTML report, JTL, logs)
6. Evaluate **TTLB SLO only**
7. Show logs on failure
8. Tear down the Docker stack

---

4. Key Variables

Variable	Description	Example Value
`USERS`	Number of concurrent virtual users	`20`
`RAMP`	Ramp-up time in seconds	`30`
`TARGET_URL`	Base URL of frontend	`http://hrschmllr.de:3000`
`API_BASE`	Base URL of API	`http://hrschmllr.de/api`
`COMPOSE_NETWORK`	Docker network name (optional)	`pg-network`

---

5. SLO Evaluation Logic

The workflow checks the **95th percentile of Initial Page TTLB**:

Test Label	Threshold
Initial Page TTLB	`< 2000 ms`

If the measured p95 latency exceeds the threshold, the build fails.

---

6. GitHub Actions Workflow

This project includes an automated **smoke test** that runs on every **pull request**.
The test checks the **Initial Page TTLB (Time-To-Last-Byte)** and enforces a strict **Service Level Objective**:

- **p95 latency must stay below 2000 ms (2 seconds)**
- If the threshold is exceeded, the workflow fails and the pull request cannot be merged until performance is improved.

---

software-quality/grafana.md

Documentation: Sensor Data Monitoring with Grafana & TimescaleDB

1. Overview

This project provides a monitoring stack for IoT sensor data. It ingests sensor readings via MQTT, stores them in a TimescaleDB (PostgreSQL with time-series extensions), and visualizes data in Grafana.

The focus of the dashboards is on **availability analysis** – comparing the expected number of sensor messages with the actual count and highlighting data gaps.
Additionally, the system shows **per-sensor availability** (temperature, humidity, pollen, particulate matter) per device.

---

2. Data Source Configuration

Grafana is preconfigured with a PostgreSQL datasource pointing to TimescaleDB.

grafana/provisioning/datasources/datasource.yaml:

```yaml
apiVersion: 1

datasources:
  - name: TimescaleDB
    type: postgres
    access: proxy
    url: db:5432
    user: ${DB_USER}
    secureJsonData:
      password: ${DB_PASSWORD}
    jsonData:
      database: ${DB_NAME}
      sslmode: disable
      postgresVersion: 14
      timescaledb: true
```

- **Datasource name**: `TimescaleDB`
- **UID**: `timescaledb` (used inside dashboards)
- **Authentication**: via environment variables `${DB_USER}` and `${DB_PASSWORD}`

---

3. Availability Dashboard

The `Availability.json` file defines a Grafana dashboard that tracks sensor data availability.

Panels

1. **Expected Values (per device)**

   ```sql
   SELECT expected_total AS value
   FROM v_totals_since_start_by_device
   WHERE device_id = <id>;
   ```

- Shows how many messages *should* have been received since the start.

2. **Actual Values (per device)**

```
SELECT actual_total AS value
FROM v_totals_since_start_by_device
WHERE device_id = <id>;
```

   - Displays the number of messages that were actually ingested.

3. **Availability Gauge**

```
SELECT availability_pct AS value
FROM v_totals_since_start_by_device
WHERE device_id = <id>;
```

   - Visualizes the percentage of received messages (`actual / expected`).
   - Thresholds:
     - <70% = red
     - ≥70% = green

4. **Per-Sensor Table (per device & sensor)**

```
SELECT device_id, sensor, expected_total, actual_count, availability_pct
FROM v_counts_since_start_by_device_and_sensor
ORDER BY device_id, sensor;
```

   - Shows expected vs. actual counts and availability percentage for each individual sensor column.

5. **Gap Table**

```
SELECT device_id, gap_start, gap_end, gap_duration
FROM v_sensor_gaps
ORDER BY device_id, gap_start;
```

   - Lists all detected communication gaps longer than 10 minutes.
   - `gap_duration` is displayed in seconds.

Layout

- Top rows: Device 1 & Device 2 – Expected, Actual, Availability
- Middle: Per-Sensor availability table
- Bottom: Table with gap events across all devices

---

4. Database Views

The dashboard depends on the following **database views** (must be created in SQL scripts):

- **`v_totals_since_start_by_device`**
  Provides `expected_total`, `actual_total`, and `availability_pct` per device.
  - `expected_total`: Expected number of rows based on device frequency.
  - `actual_total`: Count of rows actually inserted.
  - `availability_pct`: `(actual_total / expected_total) * 100`.
- **`v_counts_since_start_by_device_and_sensor`**
  Provides `expected_total`, `actual_count`, and `availability_pct` per sensor column (temperature, humidity, pollen, particulate_matter).

- **`v_sensor_gaps`**
  Lists communication gaps longer than 10 minutes with start/end timestamps and duration.

---

5. Usage Instructions

1. Start the stack:
   `docker compose up -d`
2. Open Grafana: http://localhost:3003
3. Open the dashboard **“Sensor Availability”**.
4. Observe per-device and per-sensor availability and data gaps.

---

6. Maintenance Notes

- Ensure TimescaleDB hypertables and indexes are created:
  ```
  SELECT create_hypertable('sensor_data', 'ts', if_not_exists => TRUE);
  CREATE INDEX IF NOT EXISTS idx_sensor_ts ON sensor_data (ts DESC);
  ```
- Check service health in **Uptime Kuma** at http://localhost:3002.

---

software-quality/logging-monitoring.md

Loki & Promtail Logging Setup

This document describes how to set up a **Loki + Promtail logging stack** using **Docker Compose**.
Additionally, a pre-configured **Grafana dashboard (`logging-overview.json`)** is provided to visualize log levels and messages.

Services Overview

Loki

Loki is a log aggregation system by Grafana, optimized for storing and querying logs.

Docker Compose Service:

```yaml
loki:
  image: grafana/loki:2.9.6
  command: -config.file=/etc/loki/config.yml
  ports:
    - "3100:3100"
  volumes:
    - ./loki/loki-config.yml:/etc/loki/config.yml:ro
    - loki-data:/loki
  networks:
    - pg-network
  restart: unless-stopped
```

Details:

- **Image:** `grafana/loki:2.9.6`
- **Port:** Exposes `3100` → http://localhost:3100
- **Config file:** Mounted from `./loki/loki-config.yml`
- **Storage:** Uses `loki-data` volume for persistence
- **Network:** Connected to `pg-network`
- **Restart policy:** Restart unless stopped

Promtail

Promtail is an agent that collects logs and ships them to Loki.

Docker Compose Service:

```yaml
promtail:
  image: grafana/promtail:2.9.6
  command: -config.file=/etc/promtail/config.yml
  volumes:
    - ./loki/promtail-config.yml:/etc/promtail/config.yml:ro
    - /var/run/docker.sock:/var/run/docker.sock:ro
    - promtail-data:/tmp
  depends_on:
    - loki
  networks:
    - pg-network
  restart: unless-stopped
```

Details:

- **Image:** `grafana/promtail:2.9.6`
- **Config file:** Mounted from `./loki/promtail-config.yml`
- **Docker socket:** `/var/run/docker.sock` → reads logs from all containers
- **Temporary data:** Uses `promtail-data` volume
- **Dependency:** Starts only after Loki is running
- **Restart policy:** Restart unless stopped

Grafana Dashboard: Logging Overview

The provided `Logging.json` is a **Grafana dashboard** that visualizes logs from Loki.

Features

- **Log Level Counters:**
  - Critical
  - Error
  - Warning
  - Info
- **Totals Table:** Displays aggregated log level counts
- **Log Explorer:** Lists detailed Critical / Error / Warning messages with labels
- **Environment Selector (`$env`):** Filter logs by environment
- **Auto-refresh:** Updates every 10 seconds

Import Instructions

1. Open Grafana → http://localhost:3000
2. Navigate to **Dashboards → Import**

Conclusion

With this setup you get:

- A **Loki instance** for log storage
- A **Promtail agent** collecting container logs
- A **Grafana dashboard** for monitoring & exploration

This provides a robust logging infrastructure for your Docker environment.

---

software-quality/logging.md

Logging Specification

Lightweight baseline (v0) *No code usage in this document; this is a format/operational spec.*

---

1. Purpose & Scope

Establish a **uniform, structured logging standard** so that:

- All services emit machine-parsable logs with the **same core fields**.
- Containers write **JSON to stdout** (ready for later centralization).
- Each service can add **domain-specific fields** without breaking the baseline.

---

2. Operating Principles

- **Single format:** JSON only, written to **stdout** from every container.
- **Single baseline:** Common required fields across all services.
- **Per-service extensions:** Add fields that fit the domain (API, MQTT ingestor, backend).
- **Event names:** Short, stable, `snake_case` (e.g., `request_ok`, `msg_processed`).
- **Durations:** When an event represents an operation, include `duration_ms` (elapsed time in milliseconds).
- **Keep payloads out:** Log identifiers, sizes, and reasons. Not full raw payloads for security reasons.

---

3. Unified JSON Schema (Baseline)

Every log line **must** include these keys:

Field	Type	Description
`timestamp`	string	ISO-8601 UTC (`YYYY-MM-DDTHH:mm:ss.sssZ`).

Field	Type	Description
`level`	string	`DEBUG` | `INFO` | `WARNING` | `ERROR` | `CRITICAL`.
`service`	string	Logical service name: `api`, `ingestor`, …
`module`	string	Component or source within the service.
`event`	string	Short event name (not a prose sentence), e.g., `request_ok`.
`env`	string	Deployment environment: `dev` | `staging` | `prod`.
`duration_ms`	number	**When applicable**: elapsed time of the operation (ms).

Duration definition: Time between the operation's start and end (e.g., request handling, message processing, DB write), recorded in **milliseconds**.

---

4. Per-Service Extensions (Examples)

4.1 API (Flask)

Add:

- `request_id` — Correlates a request across components.
- `method` — HTTP method.
- `path` — Request path.
- `status_code` — HTTP status code.

Example

```
{
  "timestamp": "2025-08-12T14:35:12.512Z",
  "level": "INFO",
  "service": "api",
  "module": "routes",
  "event": "request_ok",
  "env": "staging",
  "request_id": "abc123",
  "method": "GET",
  "path": "/api/devices",
  "status_code": 200,
  "duration_ms": 25
}
```

4.2 MQTT Ingestor

Use domain-specific fields (no HTTP status code):

- `device_id` — Sensor/device identifier.
- `metric` — e.g., `temperature`, `humidity`, `pollen`.
- `msg_ts` — Sensor message timestamp (ISO-8601 or epoch).
- `payload_size` — Size in bytes of the received payload.
- `result` — `ok` | `failed`.
- `reason` — Short reason on warnings/errors (e.g., `min_max_check`, `schema_mismatch`).
- `error_type` — Exception/failure type (on errors).
- (Optional) `qos`, `broker` for connection lifecycle events.

Examples

```
{
  "timestamp": "2025-08-12T14:36:00.101Z",
  "level": "INFO",
  "service": "ingestor",
  "module": "handler",
  "event": "msg_processed",
  "env": "prod",
  "device_id": 1,
  "metric": "temperature",
  "msg_ts": "2025-08-12T14:35:59.900Z",
  "payload_size": 128,
  "result": "ok",
  "duration_ms": 7
}
```

```
{
  "timestamp": "2025-08-12T14:36:05.003Z",
  "level": "WARNING",
  "service": "ingestor",
  "module": "validator",
  "event": "value_out_of_range",
  "env": "prod",
  "device_id": 1,
  "metric": "temperature",
  "result": "failed",
  "reason": "min_max_check",
  "payload_size": 124
}
```


---

5. Logging Event Reference (Minimal Set)

API

- `request_ok` (INFO): successful request completed.
- `validation_failed` (WARNING): client-side validation issue.
- `unhandled_exception` (ERROR): unexpected server-side error.

Ingestor

- `mqtt_connected` / `mqtt_reconnected` / `mqtt_disconnected` (INFO).
- `msg_processed` (INFO): message parsed/handled successfully.
- `value_out_of_range` (WARNING): domain validation failed.
- `db_write_failed` (ERROR): persistence failed.

---

6. Payload Policy

- **Do not log raw payloads** in production logs.
- Log **identifiers and metadata** instead: `payload_size`, `device_id`, `metric`, `reason`, and (when relevant) `invalid_payload_file`.
- **Invalid payload storage (local file):**
  - Store the full invalid payload in `/var/log/app/invalid_payloads/` (mounted volume).

  - File naming: timestamp + stable identifier (e.g., `2025-08-12T14-36-05Z_req-abc123.json`).
  - In the log entry, include the path as `invalid_payload_file` so incidents can be correlated without embedding the payload.
  - Apply lifecycle management (e.g., delete files older than 7 days) and restrict access.
- **Development-only exception:** In non-production environments, truncated snippets (first N characters) may be captured for debugging. Avoid in shared/staging/prod.

---

software-quality/requirements.md

Functional Requirements

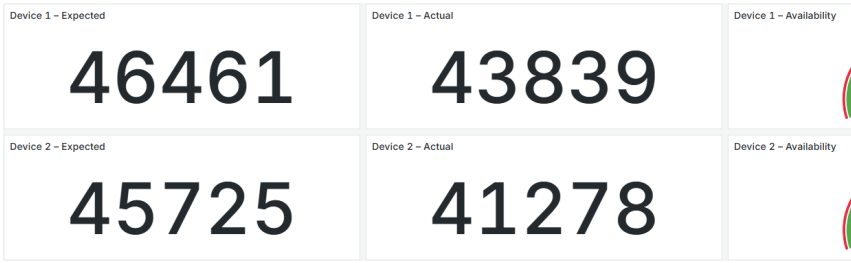
- **FR-01:**
  The software must record temperature and particulate matter values every 10 seconds and automatically send them to the data platform.
- **FR-02:**
  All sensor data must have a timestamp and a precise assignment to the location (old or new building) and be stored in a central database to enable time series comparisons between locations.
- **FR-03:**
  Warning thresholds (temperature too high, poor air quality) must be defined and taken into account.
- **FR-04:**
  The user interface should allow graphical display of temperature and particulate matter data for both locations.
- **FR-05:**
  The system must automatically generate a warning when measured environmental values exceed defined thresholds.
  The warning must be visually displayed in the dashboard and the user should receive a push notification.

Non-Functional Requirements

- **NFR-01:** The system must ensure that at least **70.00%** of sensor data per day is successfully transmitted and stored.
  If a sensor does not send data for more than 10 minutes, the system must automatically detect and log this.

  Measurement Method:

  - **Log Analysis:**
    * Calculate how many measurements per sensor per day are expected (target values).
    * Query actually stored values in TimescaleDB (actual values).

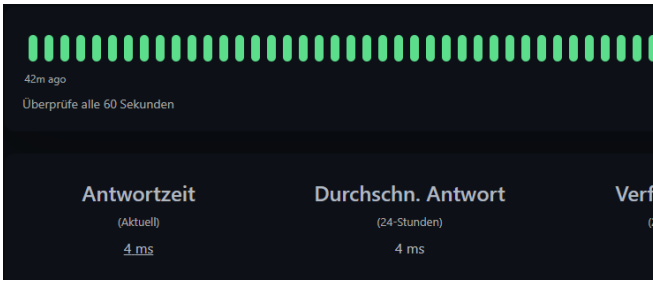


    * `Availability = actual/target`
  - **Timeout Monitoring:**
    * Backend logs when a sensor does not send data for >10 minutes.

- **NFR-02:** The web interface must guarantee an availability of at least **95%**.
  Errors must be clearly indicated to users through understandable error messages.

  Measurement Method:

  - **Monitoring:**



    * External monitoring tool (possibly self-hosted)

- **NFR-03:** Visualization of room quality data in the dashboard must be fully loaded within **2 seconds** after a user request, even with simultaneous use by up to 20 users.

  Measurement Method:

  - **Monitoring:**
    * External monitoring tool (possibly self-hosted)
    * Load test with 20 parallel users

Our Test Statistics

The load test was executed with **20 virtual users** and a **30-second ramp-up**.
The configured **total test duration was 180 seconds (3 minutes)**. Given an observed per-endpoint throughput of **~6.9 req/s**, this yields about
1,200 requests per endpoint (≈ 1,233 samples). The overall scenario produced **8,646 requests**.

Label	Samples	Errors	Error %	Avg (ms)	Min	Max	p50	p90
Total	8646	0	0.00%	148.24	0	2581	58.0	467.0
GET http://hrschmllr.de:3000/	1233	0	0.00%	516.49	354	1765	494.0	600.6
GET http://hrschmllr.de:3000/-0	1233	0	0.00%	45.91	26	388	40.0	61.0
GET http://hrschmllr.de:3000/-1	1233	0	0.00%	75.56	29	414	71.0	105.0
GET http://hrschmllr.de:3000/-2	1233	0	0.00%	52.24	29	568	43.0	67.0
GET http://hrschmllr.de:3000/-3	1233	0	0.00%	46.53	28	319	42.0	64.0
GET http://hrschmllr.de:3000/-4	1233	0	0.00%	247.65	159	1153	235.0	298.0
GET http://hrschmllr.de:3000/-5	1233	0	0.00%	42.12	27	155	38.0	58.0

Label	Samples	Errors	Error %	Avg (ms)	Min	Max	p50	p90
TTLB Dashboard	1240	0	0.00%	2004.27	0	3298	2008.5	2435.0

- **NFR-04:** The system's source code must be modular and documented so that changes to individual components (e.g., sensor protocol, visualization) can be made without affecting other parts. Additionally, there must be at least one automated component test for each main component.

  Measurement Method:

  - **Backend structure in Flask** `-mqtt/` (MQTT handler) `-storage/`(DB access) `-api/`(REST endpoints) `-test/`(Unit and integration tests)
  - **GitHub Actions CI/CD**
    - Automatic test run for every pull request to `main`
  - **Documentation**
    - OpenAPI schema for REST
    - README + short developer documentation

- **NFR-05:** The system must be designed so that it remains operational in the event of sensor failures or connection interruptions and automatically attempts to reconnect within **5 seconds**.
  Failed data transmissions must not cause a system crash but must be logged and, if necessary, temporarily stored in a queue.

  Measurement Method:

  - **MQTT client in Flask**
    - Reconnect logic every 5 seconds if broker is lost
  - **System remains operational** even without all sensors --> Frontend shows "Sensor offline"

- **NFR-06:** The software components must be provided in containers (e.g., using Docker) to enable easy platform switching (e.g., from a local server to a cloud VM) without code changes. Additionally, the system must be successfully installed and operated on at least two different operating systems (e.g., Linux and Windows).

  Measurement Method:

  - Docker setup with: `-backend-api` `-backend-mqtt` `-react-frontend` `-timescaledb`
  - Docker Compose for local development + deployment
  - CI/CD test on Linux + Windows via Github Actions
  - Environment variables for all paths and secrets instead of hardcoding

---

System Element	**Function**	**Possible Failure**
Arduino	Collect sensor data	Sensor does not provide data
Temperature sensor	Measure temperature	Temperature not measured correctly
Air quality sensor	Measure air quality	No or incorrect measurements
MQTT connection	Transmit data from Arduino to server	Connection drops
MQTT broker	Receive measurement data	Broker unreachable
Database	Store data	Data not stored

System Element	Function	Possible Failure
Website	Display measurement data	Site unavailable, connection to backend fails
Web server	Host website	Web server offline

Last updated: 14.08.2025

Quality Attributes

- Reliability
- Fault tolerance
- Performance

---

software-quality/uptime-monitoring.md

Uptime Kuma Setup & Monitor Import

Overview

This document explains how to set up Uptime Kuma in an environment where:

- The database is **not mounted** (data is not persistent).
- No status page is used.
- Monitor configuration is restored manually from a JSON backup (`Uptime_Kuma.json`).

Step 1: Start the Container

Deploy Uptime Kuma (e.g., via Docker or docker-compose). Since the database is not persisted, each container restart starts with a clean state.

Step 2: Create Admin User

1. Open the Uptime Kuma web interface (default: `http://localhost:3001` or mapped port).
2. On first launch, you will be prompted to **create a new user** (username + password).
3. Save these credentials in a secure location (e.g., password manager, team vault).
   Without persistence, this step is required every time you restart the container.

Step 3: Import Monitor Configuration

1. Log in with the user created above.
2. In the left sidebar, click on your **user name**.
3. Select **Settings**.
4. Open the **Backup** tab.
5. Click **Restore Backup**, then upload the JSON file:

- `Uptime_Kuma.json`

This file contains all defined monitors.

Step 4: Verify Monitors

After import, you should see the following monitors in the dashboard:

Monitor Name	Type	Target
Frontend	HTTP	`http://frontend:80`
Backend-API	HTTP	`http://backend-api:5000/`
Database	TCP	`db:5432`
MQTT	TCP	`isd-gerold.de:1883`
Grafana	HTTP	`http://grafana:3000`
Prometheus	HTTP	`http://prometheus:9090`
MQTT Backup	TCP	`hrschmllr.de:1883`
Loki	HTTP	`http://loki:3100/ready`

Step 5: Start Monitoring

- Once imported, all monitors can be started.
- You can view real-time status, uptime history, and response times in the dashboard.
- If needed, adjust intervals, retries, or notification settings manually for each monitor.

Notes

- Since the database is not persisted:
  - User credentials and monitor configuration must be restored after **every restart**.
  - Always keep the latest backup JSON available.

---

workflows/ci.md

Docker Compose Service Test Workflow

Purpose

This workflow builds and starts all main services (backend, frontend, database, MQTT broker) using Docker Compose, then performs health checks to verify that each service is running correctly.

Triggers

- On pull requests targeting the `main` branch

Steps

1. **Checkout Code:**
   Retrieves the latest code from the repository.
2. **Set up Docker Buildx:**
   Prepares the runner for efficient Docker builds.
3. **Build and Start Services:**
   Uses Docker Compose to build images and start services in the background.
4. **Service Status Output:**
   Prints the status of all running services.
5. **Test Backend Service:**
   Performs a health check on the backend API (`/health` endpoint).
6. **Test Frontend Service:**
   Performs a health check on the frontend (`/` endpoint).
7. **Get Service Logs:**
   Always outputs logs from backend, MQTT, frontend, and database containers for debugging.
8. **Clean Up:**
   Stops all services and removes networks and volumes to free up resources.

Environment Variables

- Database credentials
- MQTT broker configuration
- SMTP settings for Grafana

Notes

- All steps run on Ubuntu GitHub-hosted runners.
- Logs are always shown for easier debugging.

---

workflows/deploy.md

CD Workflow (Continuous Deployment)

Docker Compose Deployment with GHCR Images

Purpose:
This workflow automates the deployment of your Docker Compose application using GitHub Actions.
It builds and pushes Docker images to the GitHub Container Registry (GHCR) and then deploys the latest version to your production server via SSH.

Triggers:

- **Push to `main` branch:** Automatically builds, pushes, and deploys the latest code and Docker images whenever changes are pushed to `main`.

Runner:

- **GitHub-hosted runner:** The workflow runs on GitHub's infrastructure (e.g., `ubuntu-latest`).
  The deployment step connects to your server via SSH and executes Docker Compose commands remotely.

Workflow Steps Explained

1. **Build and Push Docker Images to GHCR:**
   On every push to `main`, GitHub Actions builds the Docker images for all services (`backend-api`, `backend-mqtt`, `frontend`, `sensor-exporter`) and pushes them to the GitHub Container Registry (GHCR).
   See `.github/workflows/docker-ghcr.yml` for details.
2. **Deploy to Production Server via SSH:**
   After building and pushing, the workflow connects to your server using SSH (`appleboy/ssh-action`).
   It pulls the latest code and images, then starts the services using `docker compose -f docker-compose-prod.yml up -d`.

Important Notes

- **Persistent Deployment:**
  This workflow **does** deploy your application to a permanent server.
  Make sure your server is reachable via SSH and has Docker Compose installed.
- **Secrets:**
  The workflow uses GitHub Secrets for authentication (`SERVER_HOST`, `SERVER_USER`, `SSH_PRIVATE_KEY`, `GHCR_TOKEN`).
  These must be configured in your repository settings.
- **`docker-compose-prod.yml` location:**
  The workflow assumes your production Compose file is named `docker-compose-prod.yml` and is located in the project root on your server.
- **Manual Execution:**
  The workflow is triggered automatically by pushes to the `main` branch.
  For manual execution, add a `workflow_dispatch` trigger to the workflow YAML.

---

workflows/frontend-test.md

Frontend JMeter Smoke Test Workflow

Purpose

This workflow runs JMeter-based smoke tests against the frontend to ensure basic performance and availability. It checks initial page load times and interval click response times against defined Service Level Objectives (SLOs).

Triggers

- On pull requests targeting the `main` branch
- Manual execution via the Actions tab (`workflow_dispatch`)

Steps

1. **Checkout Repository:**
   Retrieves the latest code from the repository.
2. **Set up Docker Buildx:**
   Prepares the runner for advanced Docker builds.
3. **Start App Stack:**
   Uses Docker Compose to start backend, frontend, and database services.
4. **Normalize Line Endings:**
   Ensures scripts use LF endings to avoid CI issues.
5. **Reachability Checks:**
   Verifies that the frontend and backend are reachable via HTTP.
6. **Run JMeter Smoke Test:**
   Executes the JMeter test script in the `frontend-loadtest` directory.
7. **Upload JMeter Artifacts:**
   Stores JMeter HTML reports and raw results as workflow artifacts.
8. **Evaluate SLOs:**
   Checks if the 95th percentile latency for page loads and interval clicks meets the defined thresholds. Fails the build if not.
9. **Show Service Logs:**
   Displays logs from backend, frontend, and database containers for debugging.
10. **Tear Down:**
    Stops and removes all Docker Compose services and volumes.

Environment Variables

- SLO thresholds for performance checks
- Docker Compose secrets for database and MQTT configuration
- JMeter test runner variables

Notes

- All steps run on Ubuntu GitHub-hosted runners.
- Artifacts are available for download after each run.
- Performance failures will fail the build.

---

workflows/python-backend-test.md

Python Backend Test Workflow

Purpose

This workflow runs automated tests for the Python backend using pytest and generates a coverage report.

Triggers

- On pull requests targeting the `main` branch

Steps

1. **Checkout Repository:**
   Retrieves the latest code.
2. **Setup Python:**
   Installs the specified Python version (e.g., 3.13).
3. **Export Environment Variables:**
   Loads secrets for database and MQTT configuration into the environment.
4. **Install Dependencies:**
   Installs Python dependencies from `backend/requirements.txt`.
5. **Debug Directory Structure:**
   Prints the current directory and lists files for debugging.
6. **Set PYTHONPATH:**
   Ensures the backend code is discoverable by pytest.
7. **Run Tests with Coverage:**
   Executes pytest, fails on the first error, and generates a coverage report.
8. **Upload Coverage Report:**
   Stores the HTML coverage report as a workflow artifact.

Notes

- Coverage reports are available for download after each run.
- All steps run on Ubuntu GitHub-hosted runners.