

МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

**«МИРЭА — Российский технологический университет»**

**РТУ МИРЭА**

|  |
| --- |
| Институт искусственного интеллекта |
| (наименование института, филиала) |
| Кафедра промышленной информатики |
| (наименование кафедры) |

**ОТЧЁТ ПО ПРАКТИЧЕСКОЙ РАБОТЕ №3**

по дисциплине «Разработка автоматизированных систем реального времени»

Тема: «Разработка автоматизированной системы мониторинга параметров производства кабельной продукции в реальном времени»

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Студент группы |  | КВБО-03-21 Беликов Михаил Дмитриевич |  |  |
|  | (Ф.И.О., учебная группа) |  | (подпись студента) |
| Преподаватель |  | Зорина Наталья Валентиновна |  |  |
|  | (Ф.И.О.) |  | (подпись преподавателя) |

Работа выполнена «\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_ 2025 г.

Проверена «\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_ 2025 г.

СОДЕРЖАНИЕ

[3 Разработка 3](#_Toc194934401)

[3.1 Выбор средств ведения разработки 3](#_Toc194934402)

[3.2 Разработка БД 4](#_Toc194934403)

[3.3 Разработка системы мониторинга параметров 6](#_Toc194934404)

[3.4 Разработка интерфейса 8](#_Toc194934405)

[ВЫВОД 11](#_Toc194934406)

[ПРИЛОЖЕНИЯ 12](#_Toc194934407)

[Приложение А 13](#_Toc194934408)

[Приложение Б 14](#_Toc194934409)

# 3 Разработка

## 3.1 Выбор средств ведения разработки

Основным языком разработки для системы мониторинга является Python 3, он обладает обильной базой различных библиотек для работы с данными и веб-разработки.

Для хранения данных используется PostgreSQL — популярная и надёжная реляционная СУБД с открытым исходным кодом. PostgreSQL обеспечивает транзакционную целостность данных, поддержку JSON для хранения сложных структур данных датчиков и высокую производительность при обработке больших объёмов информации. Взаимодействие с базой данных осуществляется через асинхронный драйвер asyncpg и ORM SQLAlchemy.

В качестве фреймворка для создания веб-интерфейса и API выбран FastAPI — современный, высокопроизводительный веб-фреймворк с поддержкой асинхронных операций. FastAPI обеспечивает автоматическую генерацию документации API, валидацию данных и типизацию, что значительно снижает количество ошибок на этапе разработки. Выбор FastAPI позволил реализовать как API для взаимодействия с системой, так и веб-интерфейс для мониторинга производства и управления параметрами.

Архитектура системы основана на принципах событийно-ориентированного программирования и использует брокеры сообщений для обеспечения надёжного обмена данными между компонентами. Для получения данных с датчиков применяется протокол MQTT — легковесный протокол обмена сообщениями, который идеально подходит для устройств Интернета вещей и промышленных датчиков. В качестве MQTT брокера используется Mosquitto. Обработка потоков данных осуществляется с помощью Apache Kafka, обеспечивающей надёжную доставку сообщений и масштабируемость при растущих объёмах данных.

Интерфейс пользователя построен с использованием современных веб-технологий HTML5, CSS3 и JavaScript. Для улучшения внешнего вида и создания адаптивного интерфейса применяется Bootstrap, обеспечивающий единообразие элементов управления и правильное отображение на различных устройствах.

В системе реализована аутентификация на основе JWT (JSON Web Tokens). Это позволяет обеспечить безопасный доступ к функциям системы в соответствии с ролями пользователей (администратор, оператор, технический специалист, менеджер).

Для связи между Backend и Fronted частями используется протокол Websocket, который устанавливает двунаправленное соединение и позволяет не опрашивать по HTTP сервер каждую секунду.

Программная архитектура системы построена по модульному принципу и состоит из следующих основных компонентов:

1. Модуль сбора данных датчиков — отвечает за получение информации с физических датчиков или симулятора через протокол MQTT.
2. Модуль обработки и анализа данных — выполняет первичную обработку данных, проверяет соответствие показаний допустимым значениям и генерирует оповещения при отклонениях от нормы.
3. Модуль хранения данных — обеспечивает сохранение информации в базе данных PostgreSQL и предоставляет интерфейс для выполнения запросов.
4. Модуль оповещений — формирует уведомления о критических ситуациях и отклонениях в работе оборудования.
5. Модуль аутентификации и авторизации — управляет процессами идентификации пользователей и контролирует доступ к функциям системы.
6. Веб-модуль — предоставляет интерфейс пользователя для мониторинга и управления производственным процессом.
7. API модуль — обеспечивает программный интерфейс для интеграции с другими системами.

## 3.2 Разработка БД

Центральным элементом модели является таблица датчиков (sensors), которая содержит информацию о всех установленных на производстве измерительных устройствах. Датчики расположены в определенных местах производственной линии, что отражено в связи с таблицей местоположений (productionLine).

Таблица показаний датчиков (currentValues) представляет собой основное хранилище данных, поступающих от датчиков в реальном времени. При отклонении показаний от нормы система генерирует события, которые сохраняются в таблице событий (events).

Доступ к системе контролируется через таблицы пользователей (users) и ролей (roles), обеспечивая разграничение прав в соответствии с должностными обязанностями пользователей.

На Листинге 3.1 показан SQL-скрипт для создания таблиц users, role, sensors, currentValues, productionLine и events.

|  |
| --- |
| Листинг 3.1 — Создание таблиц |
| CREATE TABLE "users" (  "id" BIGINT NOT NULL,  "name" TEXT NOT NULL,  "login" TEXT NOT NULL,  "password" TEXT NOT NULL,  "role\_id" INTEGER NOT NULL  );  ALTER TABLE "users" ADD PRIMARY KEY("id");  CREATE TABLE "role"(  "id" BIGINT NOT NULL,  "roleName" TEXT NOT NULL,  "description" BIGINT NOT NULL  );  ALTER TABLE "role" ADD PRIMARY KEY("id");  CREATE TABLE "sensors"(  "id" BIGINT NOT NULL,  "sensor\_name" TEXT NOT NULL,  "location" BIGINT NOT NULL,  "active" BOOLEAN NOT NULL  );  ALTER TABLE "sensors" ADD PRIMARY KEY("id");  CREATE TABLE "currentValues"(  "id" BIGINT NOT NULL,  "time" TIMESTAMP(0) WITHOUT TIME ZONE NOT NULL,  "sensors\_id" BIGINT NOT NULL,  "value" FLOAT(53) NOT NULL  );  ALTER TABLE "currentValues" ADD PRIMARY KEY("id");  CREATE TABLE "productionLine"(  "id" BIGINT NOT NULL,  "name" TEXT NOT NULL  );  ALTER TABLE "productionLine" ADD PRIMARY KEY("id");  CREATE TABLE "events"(  "id" BIGINT NOT NULL,  "time" TIMESTAMP(0) WITHOUT TIME ZONE NOT NULL,  "description" TEXT NOT NULL,  "sensors\_id" BIGINT NOT NULL,  "user\_id" BIGINT NOT NULL  );  ALTER TABLE "events" ADD PRIMARY KEY("id"); |

## 3.3 Разработка системы мониторинга параметров

Разработка системы мониторинга производства кабельной продукции состоит из нескольких ключевых элементов, реализующих основной функционал — от сбора данных с датчиков до их обработки и отображения в веб-интерфейсе.

Сбор данных с датчиков осуществляется через протокол MQTT (Приложение А), который обеспечивает надёжную передачу сообщений даже в условиях нестабильного сетевого соединения. Клиент обеспечивает подключение к MQTT брокеру, подписку на необходимые топики, обработку входящих сообщений и отказоустойчивость при разрыве соединения.

API системы реализовано с использованием FastAPI и предоставляет доступ к данным о текущем состоянии производства, истории показаний датчиков и оповещениях. API поддерживает JWT-аутентификацию и разграничение доступа по ролям.

Обработка данных в режиме реального времени является ключевым аспектом разработанной системы. В этом разделе рассматривается архитектура и реализация компонентов, обеспечивающих сбор, обработку, анализ и визуализацию данных в реальном времени.

Система обработки данных в режиме реального времени построена на принципах событийно-ориентированной архитектуры (Event-Driven Architecture), где каждое изменение показаний датчика рассматривается как событие, требующее обработки.

Такая архитектура обеспечивает:

1. Низкую задержку между получением данных и их отображением;
2. Масштабируемость системы при увеличении количества датчиков;
3. Устойчивость к сбоям отдельных компонентов;
4. Возможность параллельной обработки больших объёмов данных.

MQTT (Message Queuing Telemetry Transport) был выбран как основной протокол для сбора данных с датчиков благодаря его легковесности, надёжности и поддержке модели публикации/подписки. Система использует иерархическую структуру топиков для организации потоков данных от различных участков производства.

Для обеспечения обновления данных в веб-интерфейсе без перезагрузки страницы используется технология WebSocket. Код класса, отвечающего за работу по web socket, приведен в Приложении Б.

Реализация WebSocket сервера поддерживает следующие функции:

* подключение клиентов к различным каналам данных;
* широковещательная рассылка обновлений;
* целевая отправка оповещений конкретным клиентам.

## 3.4 Разработка интерфейса

Разработка интерфейсов для системы была ориентирована на предоставление удобного, информативного и эффективного инструмента для работы операторов и руководителей производства. Основной целью являлось создание интуитивно понятного веб-интерфейса, обеспечивающего оперативный мониторинг и управление процессами производства в режиме реального времени.

Для разработки фронтенд-части системы были использованы следующие технологии:

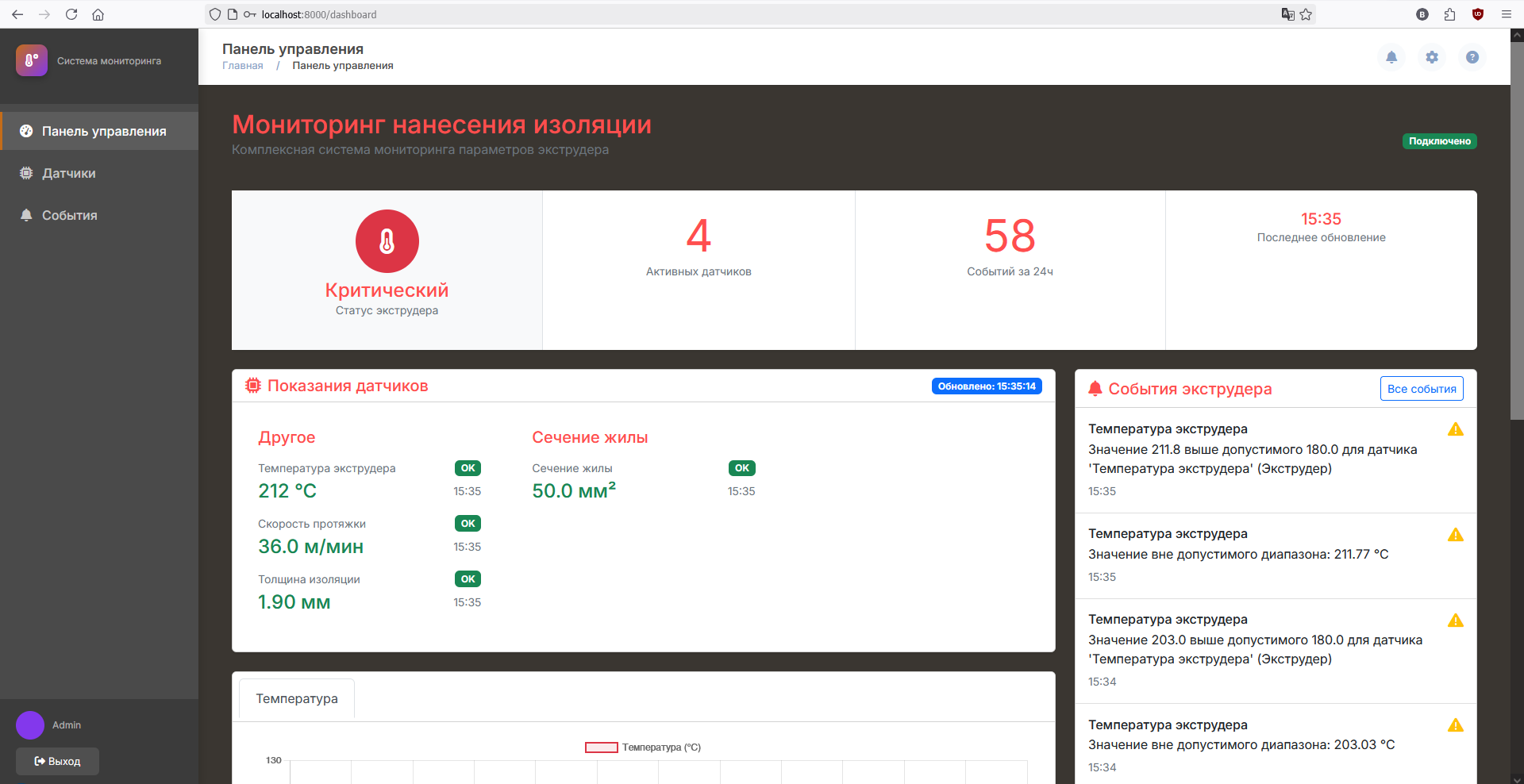
1. HTML5/CSS3/JavaScript — основа веб-интерфейса.
2. Bootstrap 5 — фреймворк для создания адаптивного и современного дизайна.
3. Chart.js — библиотека для визуализации данных.
4. WebSocket API — для обеспечения обновления данных в реальном времени.
5. Jinja2 — шаблонизатор для генерации HTML на стороне сервера.
6. FontAwesome — набор иконок для улучшения визуального восприятия интерфейса.
7. Fetch API — для асинхронного взаимодействия с REST API сервера.

Структура пользовательского интерфейса

Интерфейс системы организован в виде панели управления с различными функциональными разделами (Рисунок 3.1).

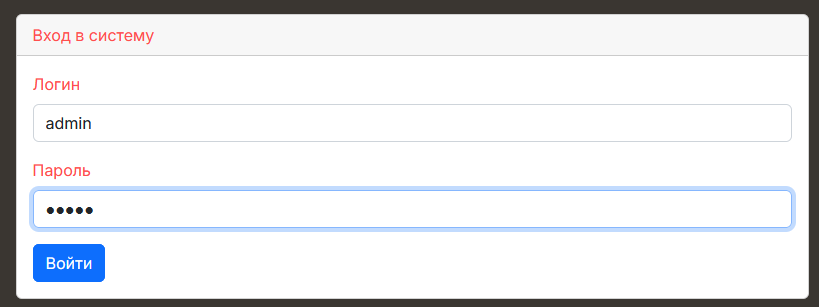
Содержимое интерфейса:

1. Навигационное меню — обеспечивает быстрый доступ ко всем разделам системы
2. Информационная панель (дашборд) — отображает общее состояние производства.
3. Детальные страницы датчиков — для просмотра подробной информации по каждому датчику.
4. Раздел оповещений и событий — для отслеживания нештатных ситуаций.
5. Аналитические отчёты и графики — для анализа производственных показателей.
6. Панель настроек — для конфигурирования параметров системы.
7. Страница авторизации — для аутентификации пользователей с разным уровнем доступа.



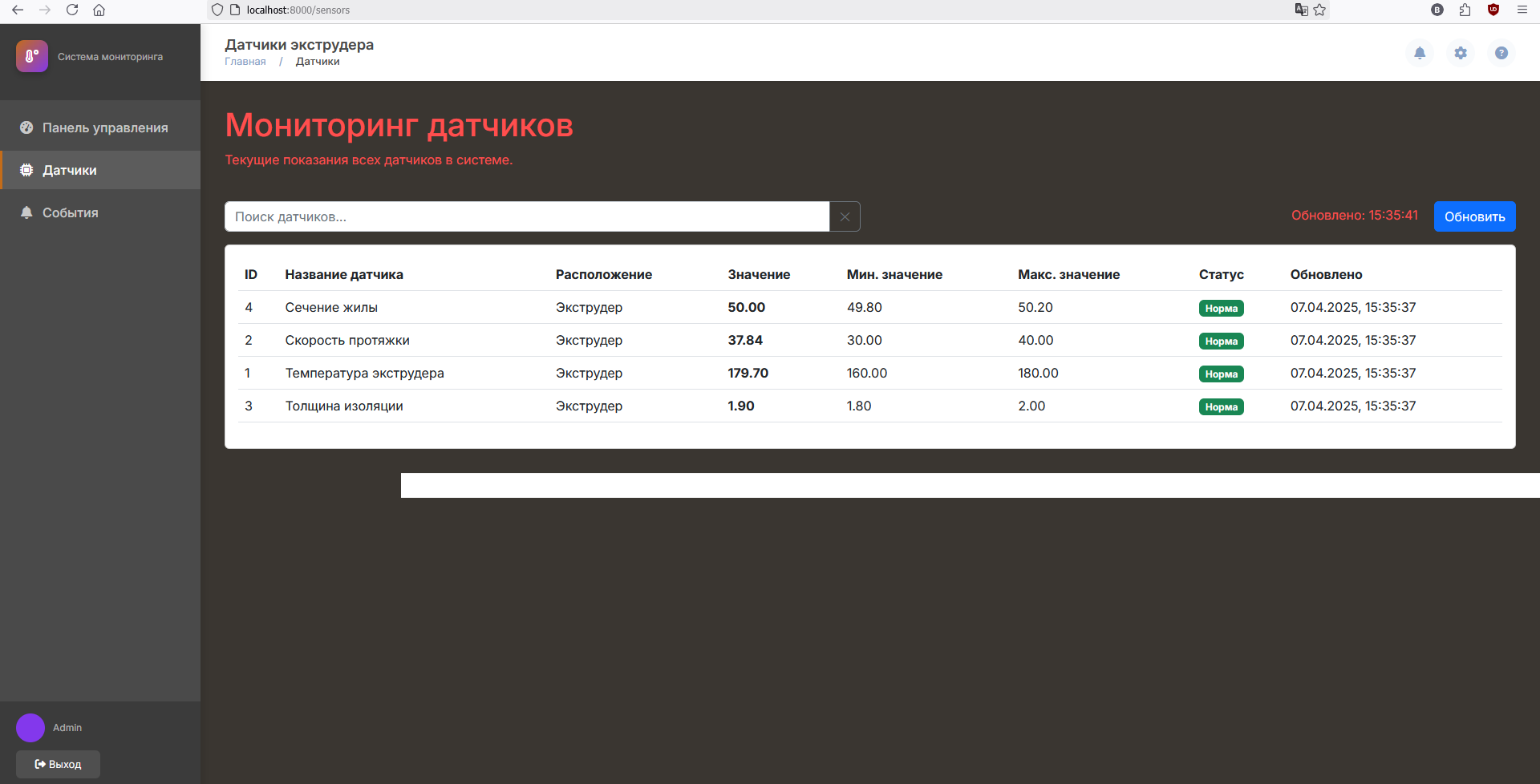
**Рисунок 3.1 — Интерфейс системы мониторинга**

При входе в систему появляется окно аутентификации (Рисунок 3.2).



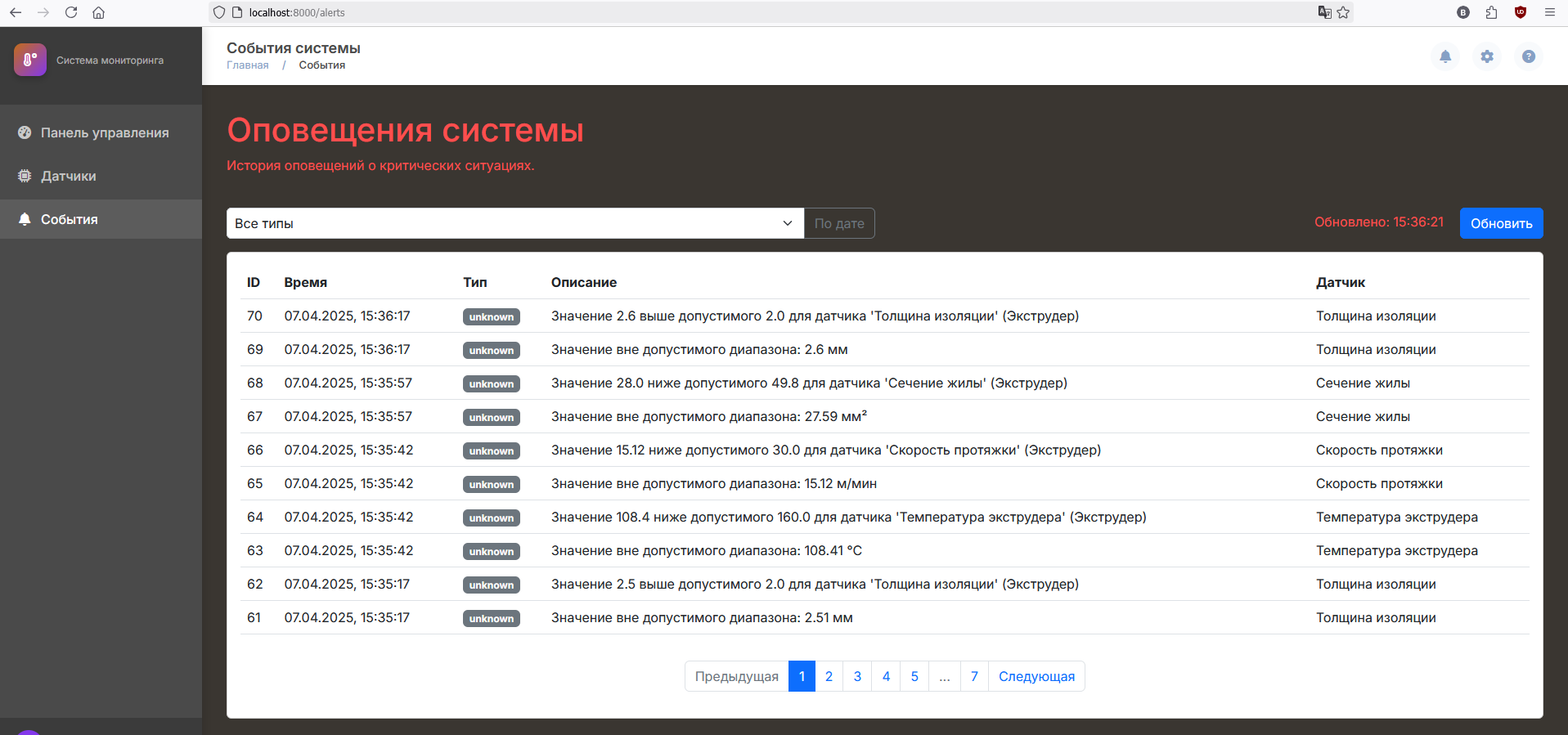
**Рисунок 3.2 — Аутентификация**

На Рисунке 3.3 представлен раздел мониторинга значений. Здесь отображаются все значения с датчиков в реальном времени.



**Рисунок 3.3 — Мониторинг значений датчиков**

Раздел «Оповещения системы» отображает все записанные ошибки, которые были обработаны системой (Рисунок 3.4).



**Рисунок 3.4 — Оповещения системы**

# ВЫВОД

В ходе выполнения работы создана комплексная система автоматизации мониторинга за процессом стерилизации свиных консервов, обеспечивающая мониторинг, анализ и управление производственными процессами в режиме реального времени. Система реализована на основе событийно-ориентированной архитектуры с использованием современных технологий: Python, FastAPI, PostgreSQL, MQTT, Apache Kafka и WebSocket. Результат работы представляет собой готовое к промышленному внедрению решение, объединяющее современные технологии программной инженерии с глубоким пониманием технологического процесса стерилизации консервов.

# ПРИЛОЖЕНИЯ

Приложение А

Приложение Б

### Приложение А

Функция обработки сообщений с Mqqt брокера

|  |
| --- |
| Листинг А — mqtt\_client\_thread |
| def mqtt\_client\_thread():  global mqtt\_client\_instance, mqtt\_connected, stop\_flag  # Создаем и настраиваем клиент  client = mqtt.Client()  mqtt\_client\_instance = client  # Устанавливаем обработчики  client.on\_connect = on\_connect  client.on\_message = on\_message  client.on\_disconnect = on\_disconnect  # Устанавливаем учетные данные, если указаны  if config.MQTT\_USERNAME and config.MQTT\_PASSWORD:  client.username\_pw\_set(config.MQTT\_USERNAME, config.MQTT\_PASSWORD)  while not stop\_flag:  try:  if not mqtt\_connected:  logger.info(f"Попытка подключения к MQTT брокеру {config.MQTT\_BROKER}:{int(config.MQTT\_PORT)}...")  client.connect(config.MQTT\_BROKER, int(config.MQTT\_PORT), 60)  # Запускаем цикл обработки сообщений  client.loop\_start()  while not stop\_flag:  time.sleep(1)  if not mqtt\_connected:  break  client.loop\_stop()  if stop\_flag:  break  logger.info("Переподключение к MQTT брокеру...")  time.sleep(5)  except Exception as e:  logger.error(f"Ошибка MQTT клиента: {e}")  time.sleep(5)  try:  if mqtt\_connected:  client.disconnect()  logger.info("MQTT клиент отключен")  except Exception as e:  logger.error(f"Ошибка при отключении MQTT клиента: {e}") |

### Приложение Б

Код класса, отвечающего за работу по Websocket

|  |
| --- |
| Листинг Б — ConnectionManager |
| class ConnectionManager:  def \_\_init\_\_(self):  # Словарь подключений по группам  self.active\_connections: Dict[str, List[WebSocket]] = {}  # Задачи для отправки данных  self.tasks: Dict[str, asyncio.Task] = {}    async def connect(self, websocket: WebSocket, group: str):  """Подключение нового клиента"""  await websocket.accept()    if group not in self.active\_connections:  self.active\_connections[group] = []    self.active\_connections[group].append(websocket)  logger.info(f"Клиент подключен к группе {group}, всего подключений: {len(self.active\_connections[group])}")    def disconnect(self, websocket: WebSocket, group: str):  """Отключение клиента"""  if group in self.active\_connections:  if websocket in self.active\_connections[group]:  self.active\_connections[group].remove(websocket)  logger.info(f"Клиент отключен от группы {group}, осталось подключений: {len(self.active\_connections[group])}")    async def send\_personal\_message(self, message: dict, websocket: WebSocket):  """Отправка сообщения конкретному клиенту"""  try:  await websocket.send\_json(message)  except Exception as e:  logger.error(f"Ошибка отправки сообщения: {e}")    async def broadcast(self, message: dict, group: str):  """Отправка сообщения всем подключенным клиентам группы"""  if group not in self.active\_connections:  return    disconnected = []  for connection in self.active\_connections[group]:  try:  await connection.send\_json(message)  except Exception as e:  logger.error(f"Ошибка широковещательной отправки: {e}")  disconnected.append(connection)    # Удаление отключенных соединений  for connection in disconnected:  self.disconnect(connection, group) |

|  |
| --- |
| Продолжение Листинга 3.1 |
| async def start\_broadcast\_task(self, group: str, interval: float, data\_generator):  """Запуск задачи для периодической отправки данных"""  if group in self.tasks and not self.tasks[group].done():  self.tasks[group].cancel()    task = asyncio.create\_task(self.\_broadcast\_task(group, interval, data\_generator))  self.tasks[group] = task  return task    async def \_broadcast\_task(self, group: str, interval: float, data\_generator):  """Периодическая отправка данных всем клиентам группы"""  while True:  try:  # Если нет подключений, просто ждем  if group not in self.active\_connections or not self.active\_connections[group]:  await asyncio.sleep(interval)  continue    # Получаем данные от генератора  data = await data\_generator()    # Отправляем данные всем клиентам группы  await self.broadcast(data, group)    # Ждем указанный интервал  await asyncio.sleep(interval)    except asyncio.CancelledError:  logger.info(f"Задача трансляции для группы {group} отменена")  break  except Exception as e:  logger.error(f"Ошибка в задаче трансляции для группы {group}: {e}")  await asyncio.sleep(interval)    def stop\_all\_tasks(self):  """Остановка всех запущенных задач"""  for group, task in self.tasks.items():  if not task.done():  task.cancel()  self.tasks.clear() |