Министерство образования Республики Беларусь

Учреждение образования  
 БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ  
 ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерного проектирования

Кафедра проектирования информационно-компьютерных систем

Рефакторинг и оптимизация программного кода

Отчет

по результатам выполнения лабораторных работ  
и заданий к практическим занятиям

Проверил \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ А.В. Шелест

(подпись)

зачтено \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(дата защиты)

Выполнил  Львов Е.Е.

(подпись) гр214371

Минск, 2025

**СОДЕРЖАНИЕ**

[Ссылки на репозитории 3](#_Toc195716211)

[1 Архитектура и система дизайна программного средства 4](#_Toc195716212)

[2 Проектирование пользовательского интерфейса ПС 9](#_Toc195716213)

[3 Реализация клиентской части ПС 10](#_Toc195716214)

[4 Проектирование базы данных 14](#_Toc195716215)

[5 Реализация ПС через UML-диаграммы 16](#_Toc195716216)

[6 Разработка документации к ПС с Open API и оценка качества   
 программного кода 20](#_Toc195716217)

[7 Аутентификация и авторизация пользователей 23](#_Toc195716218)

[8 Тестирование программного средства 27](#_Toc195716219)

[9 Процесс развертывания программного средства 35](#_Toc195716220)

[10 Разработка руководства пользователя 36](#_Toc195716221)

[Выводы 38](#_Toc195716222)

**Ссылки на репозитории**

1 https://github.com/eelvov/LvovEE\_214371\_RIOPK\_Spec

2 https://github.com/eelvov/LvovEE\_214371\_RIOPK\_Server

3 https://github.com/eelvov/LvovEE\_214371\_RIOPK\_Client

**1 Архитектура и система дизайна программного средства**

Цель: спроектировать архитектуру программного средства, следуя требованиям архитектурного шаблона *Clean Architecture*, и разработать систему дизайна (*UI-kit*) пользовательского интерфейса.

Диаграмма вариантов использования программного средства автоматизации рабочего процесса администратора информационной безопасности демонстрирует взаимодействие двух основных актеров – Сотрудника и Администратора.

Сотруднику доступен следующий функционал:

* просмотр назначенных инцидентов;
* поиск инцидентов по названию;
* сортировка инцидентов;
* отправка отчета и решения по инциденту администратору;
* комментирование инцидента;
* просмотр состояния инцидентов информационной безопасности;
* просмотр полезных материалов и инструментов для работы с инцидентами.

Администратор обладает расширенным функционалом:

* просмотр инцидентов системы;
* редактирование инцидентов системы;
* назначение инцидента сотруднику;
* комментирование инцидента;
* проверить отчет сотрудника по инциденту;
* просмотр мониторинг безопасности системы;
* просмотр состояния инцидентов информационной безопасности.

Диаграмма вариантов использования отображена на рисунке 1.1.

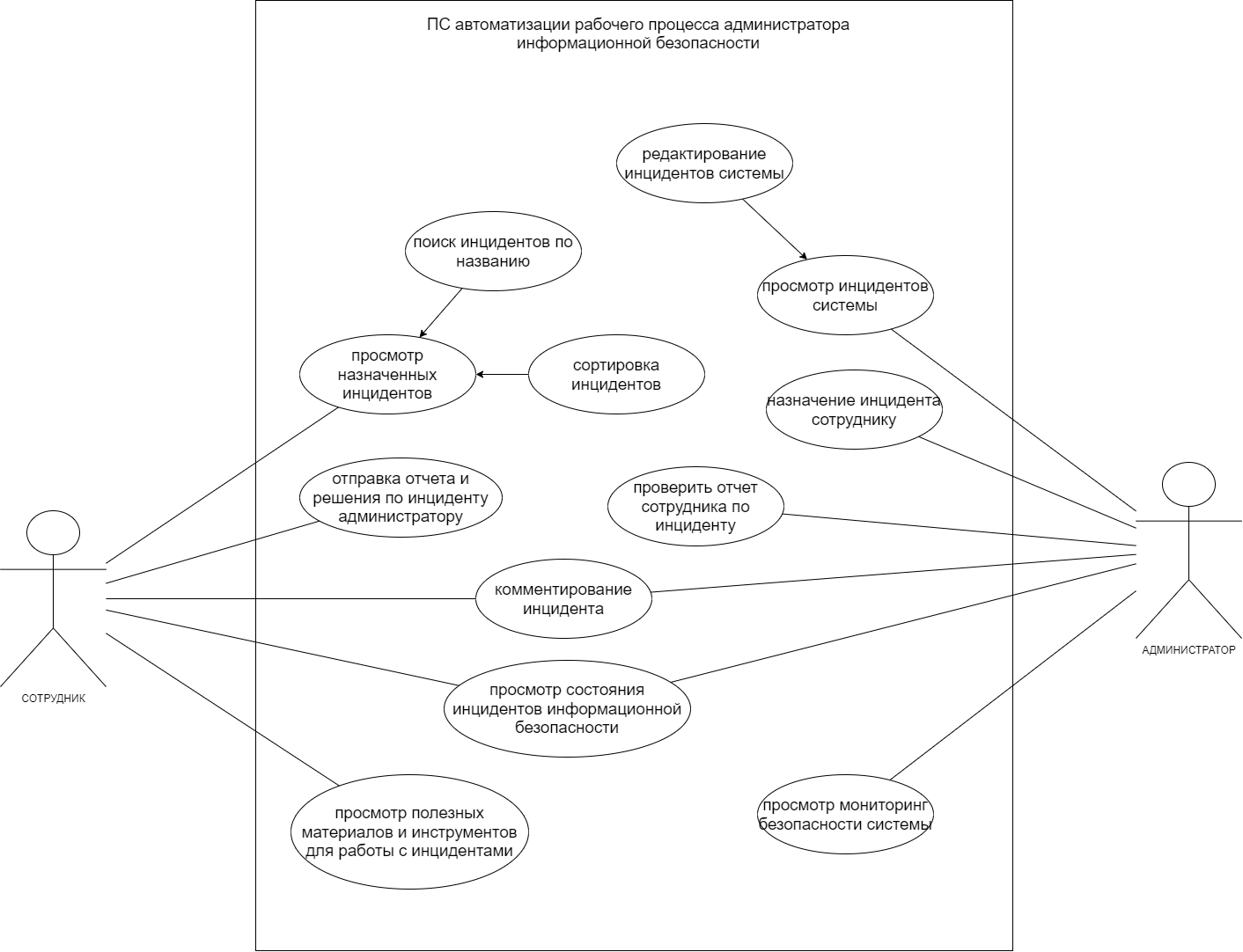


Рисунок 1.1 – Диаграмма вариантов использования

Программное средство автоматизации рабочего процесса администратора информационной безопасности – это веб-приложение, которое позволяет в режиме реального времени отслеживать статус безопасности информационной системы, где сотрудники работают с инцидентами, решают их, а администраторы этот процесс контролируют и просматривают аналитику.

Внешние участники:

* сотрудник – решает проблему с назначенным инцидентом, готовит отчет по инциденту и направляет на рассмотрение администратору;
* администратор – управляет инцидентами, назначает их сотрудникам, просматривает статистику по инцидентам и мониторит статус безопасности информационной системы.
* информационная система – выдает подозрительные действия в системе на рассмотрение администратору;

Взаимодействия:

* сотрудник → рассматривает инцидент, направляет готовый отчет администратору;
* администратор → управляет инцидентами, отслеживает безопасность системы;
* система направляет инцидент на рассмотрение.

Модель C4 по уровням 1, 2, 3 и 4 отображена на рисунке 1.2, 1.3 и 1.4 соответственно.

Изображение выглядит как текст, снимок экрана, визитная карточка, Шрифт

Автоматически созданное описание

Рисунок 1.2 – *System Context Diagram*

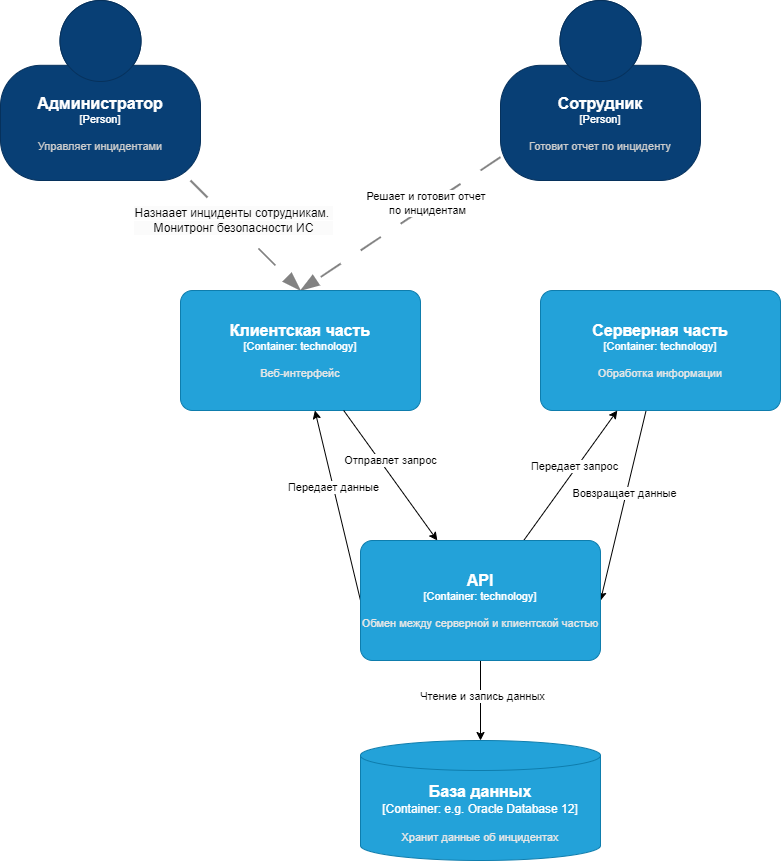


Рисунок 1.3 *– Container diagram*

Изображение выглядит как текст, снимок экрана, Шрифт, диаграмма

Автоматически созданное описание

Рисунок 1.4 – *Component diagram*

Интерфейс выполнен в лаконичном современном дизайне с акцентом на удобство использования и быструю навигацию. Все функциональные блоки чётко разделены визуально.

В цветовой палитре использованы синие, белые, зелёные и красные оттенки, обеспечивающие хорошую контрастность и читаемость. Основной акцентный цвет — синий (применён в кнопках и меню). Белый служит фоном, серый — для разделителей и подложек, а зелёный и красный помогают отображать статусы элементов.

Шрифт: *Roboto / Helvetica / Arial*, размер от 14 до 20 *px* в зависимости от заголовка или текста.

Интерфейс реализован в стиле *Material* *Design*, что обеспечивает читаемость и понятность элементов, логичное размещение блоков, быструю ориентацию пользователя в системе.

В результате проделанной работы была спроектирована архитектура программного средства автоматизации рабочего процесса администратора информационной безопасности в соответствии с принципами архитектурного шаблона *Clean Architecture*. Обеспечивает чёткое разделение ответственности между слоями системы, повышая её масштабируемость, сопровождаемость и устойчивость к изменениям.

**2 Проектирование пользовательского интерфейса программного средства**

Цель: проектирование пользовательского интерфейса программного средства.

Для двух ключевых ролей – Сотрудник и Администратор – разработаны *user-flow* диаграммы. Отображены на рисунке 2.1 и 2.2 соответственно.

Изображение выглядит как диаграмма, текст, План, снимок экрана

Автоматически созданное описание

Рисунок 2.1 – *User-flow* диаграмма для Администратора

Изображение выглядит как диаграмма, текст, План, Технический чертеж

Автоматически созданное описание

Рисунок 2.2 – *User-flow* диаграмма для Сотрудника

**3 Реализация клиентской части программного средства**

Цель: реализация клиентской части программного средства

В качестве инструментов разработки пользовательского интерфейса программного средства выбран *JavaScript React*.

Ниже представлен дизайн программного средства.

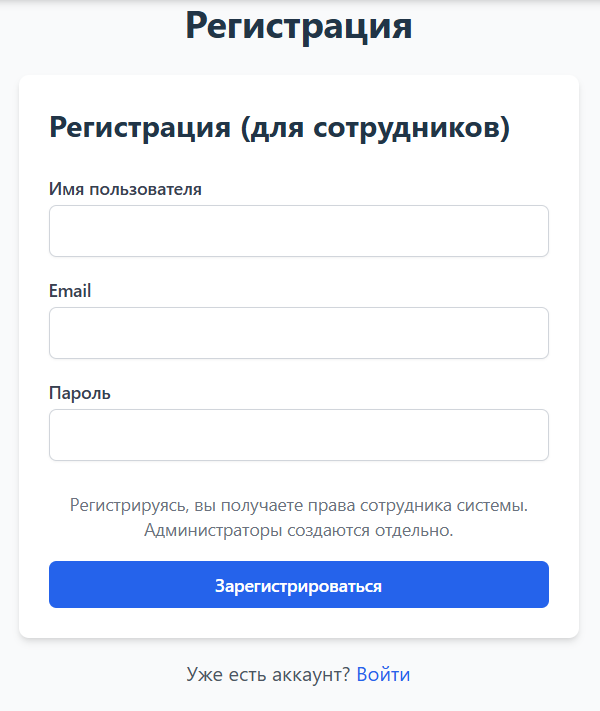


Рисунок 3.1 – Регистрация

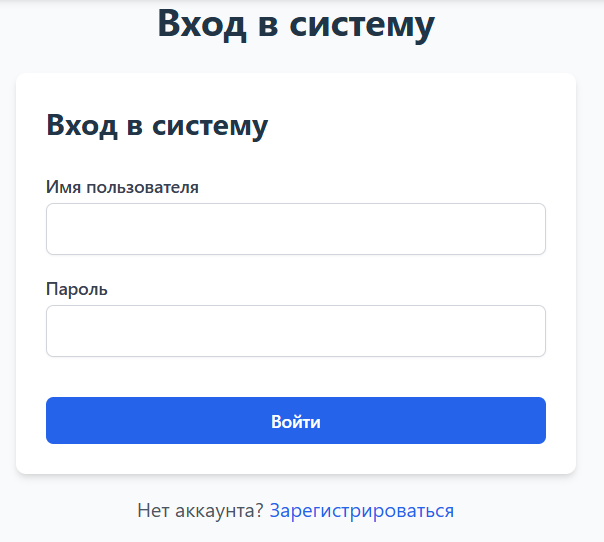


Рисунок 3.2 – Вход в программу

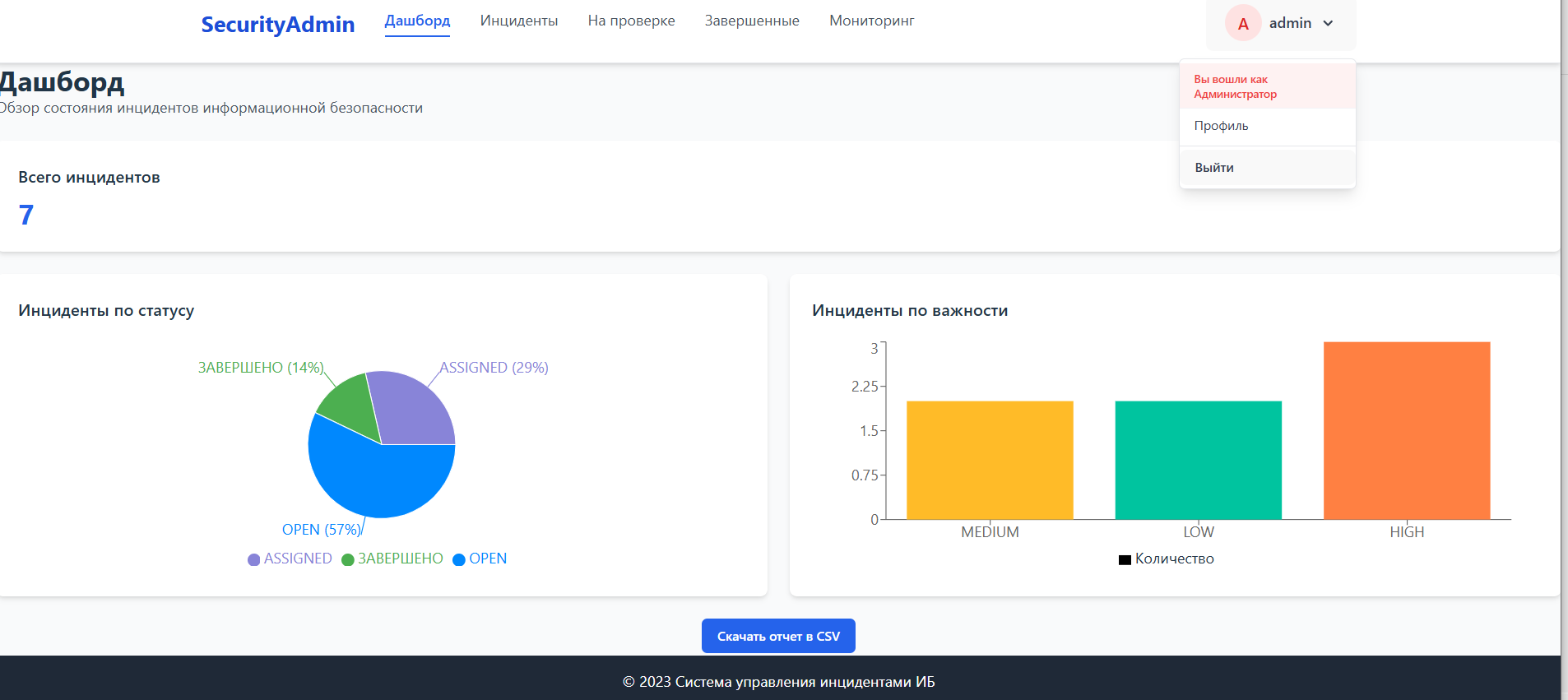


Рисунок 3.3. – Страница статистики инцидентов

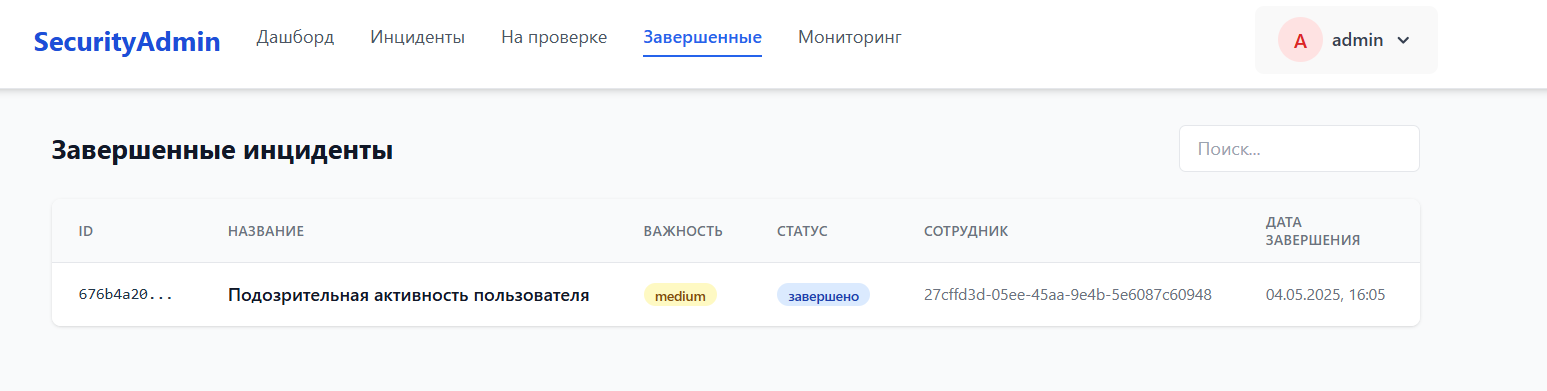


Рисунок 3.4 – Страница завершенных инцидентов

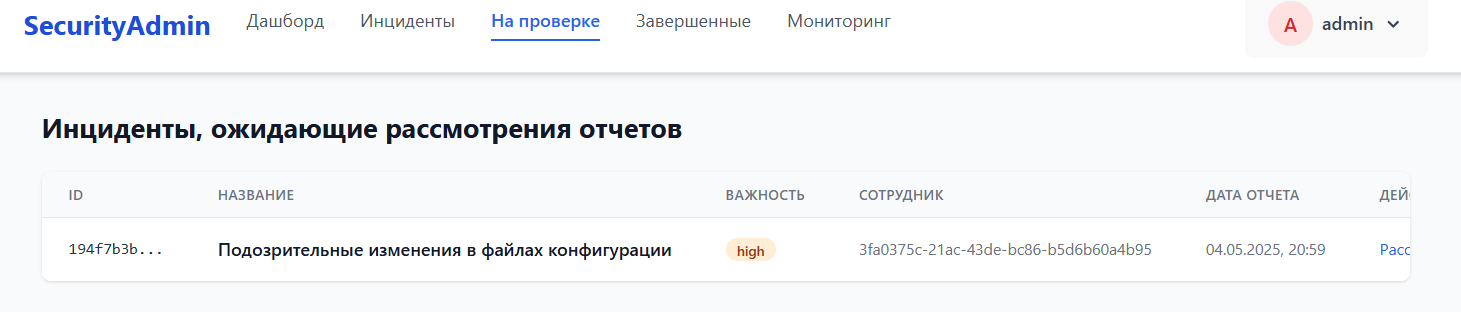


Рисунок 3.5 – Страница отчетов для проверки

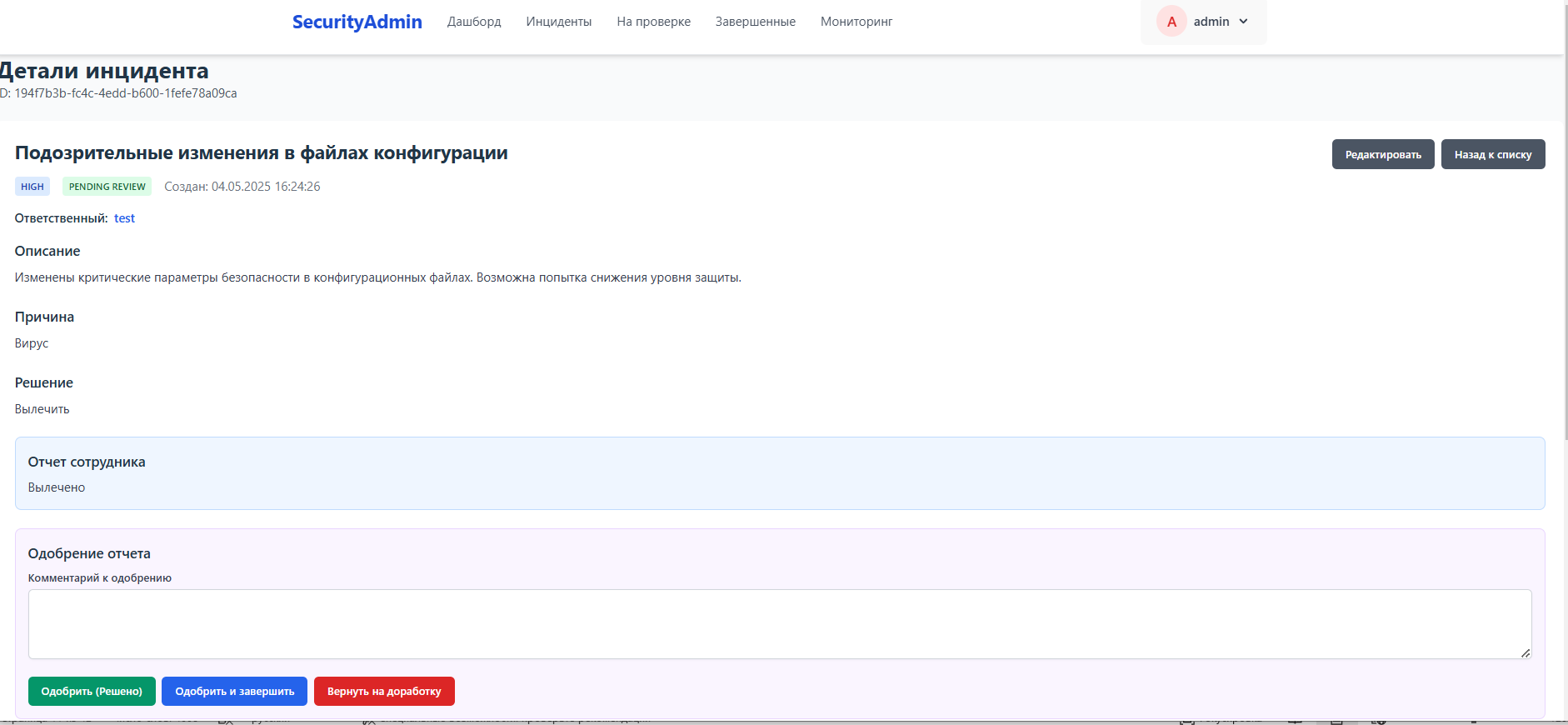


Рисунок 3.6 – Страница деталей инцидента

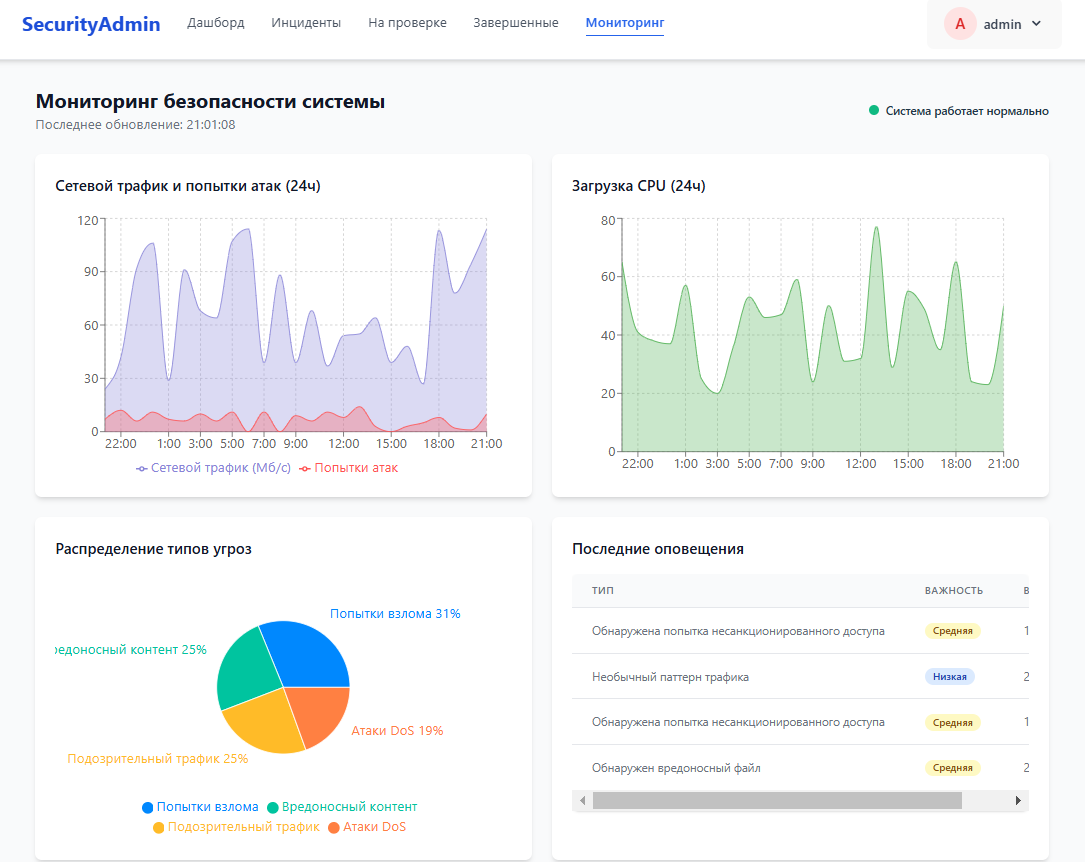


Рисунок 3.6 – Страница мониторинга системы безопасности

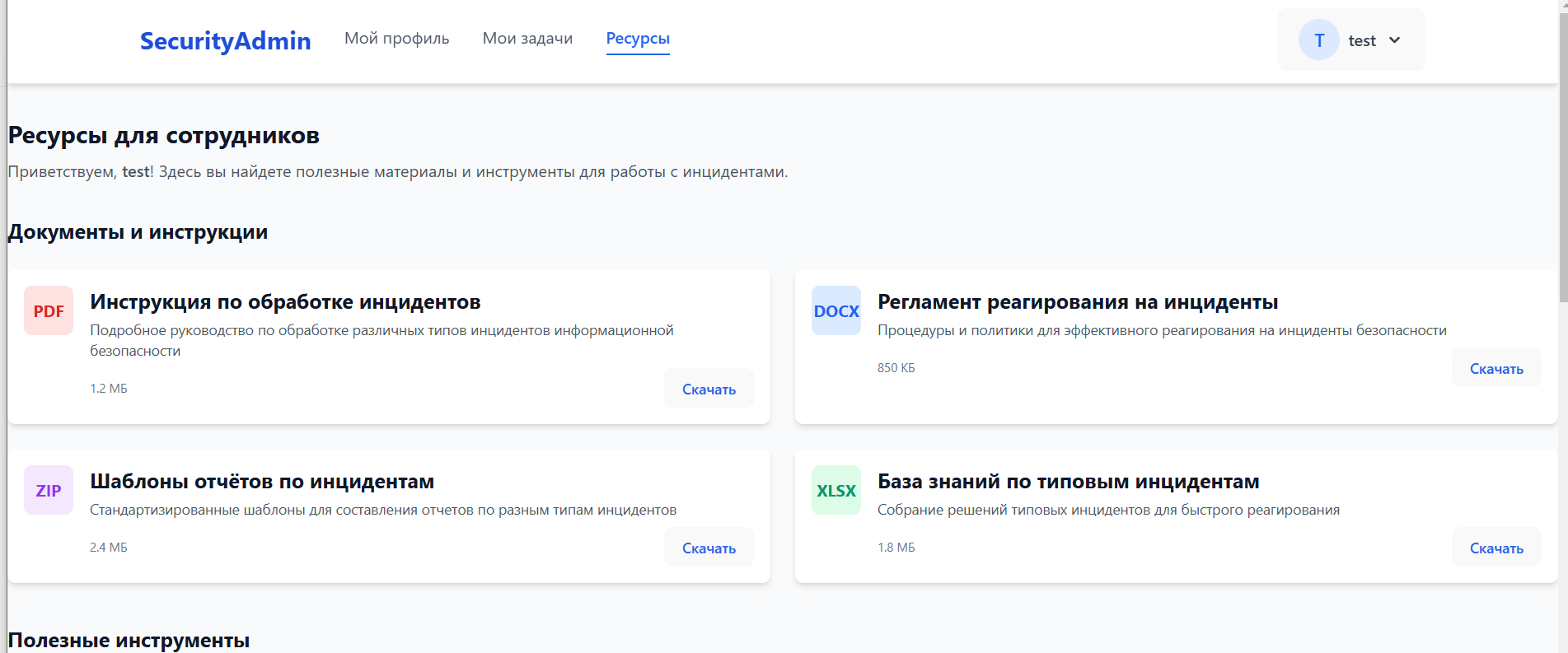


Рисунок 3.7 – Страница полезных ресурсов для сотрудника

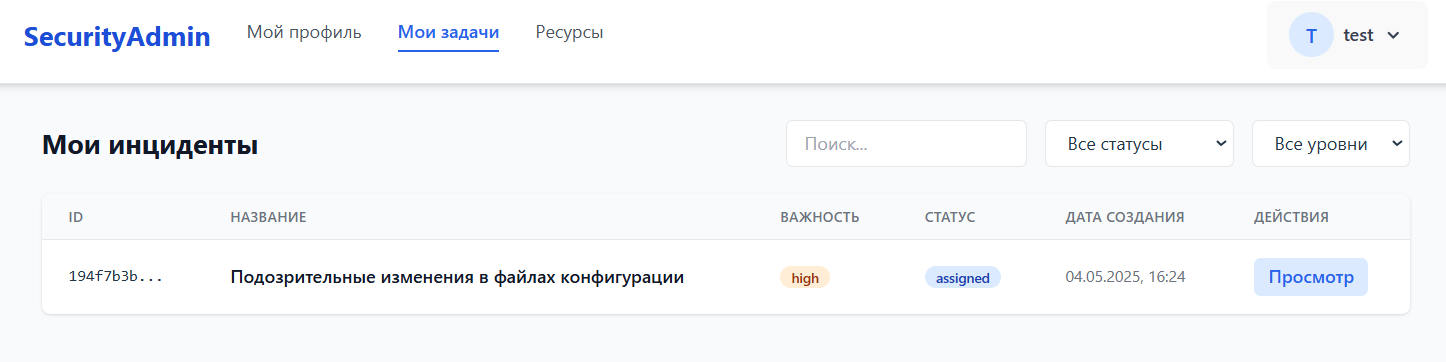


Рисунок 3.8 – Страница задач, назначенных на сотрудника

Интерфейс построен с применением современных подходов к проектированию – была реализована навигация по основным страницам системы в соответствии с ролями (Сотрудник, Администратор), обеспечено логичное взаимодействие пользователя с системой, а также визуальное разделение функциональных блоков.

Все компоненты реализованы в соответствии с выбранным стеком технологий. Интерфейс выдержан в едином стиле: использована цветовая схема, подобранная для повышения читаемости и удобства восприятия информации, применён шрифт *Roboto/Helvetica/Arial*. Интерфейс адаптирован под основные пользовательские сценарии и обеспечивает комфортную работу с системой.

**4 Проектирование базы данных**

Цель: спроектировать схему БД и представить описание ее сущностей и их атрибутов

В соответствии с требованиями предметной области Программное средство автоматизации рабочего процесса администратора информационной безопасности, была спроектирована реляционная база данных, обеспечивающая хранение информации об инцидентах, сотрудниках. Схема приведена к третьей нормальной форме (3НФ). В таблице 4.1 расписаны сущности и их атрибуты.

Таблица 4.1 – Описание сущностей и их атрибутов

|  |  |  |
| --- | --- | --- |
| Поле | Тип данных | Описание |
| Users | | |
| id | UUID / INT | Первичный ключ |
| userName | VARCHAR | Имя пользователя |
| email | VARCHAR (unique) | Email |
| password | VARCHAR | Пароль |
| role | ENUM | Роль пользователя |
| createdAt | DATETIME | Дата создания записи |
| updatedAt | DATETIME | Дата обновления записи |
| Comments | | |
| id | UUID / INT | Первичный ключ |
| userId | UUID / INT | Внешний ключ → Users(id) |
| incidentId | UUID / INT | Внешний ключ → incidents(id) |
| content | VARCHAR | Содержимое комментария |
| createdAt | DATETIME | Создан |
| Incidents | | |
| id | UUID / INT | Первичный ключ |
| title | VARCHAR | Название |
| description | TEXT | Описание |
| severity | VARCHAR | Строгость/Уровень |
| status | ENUM | Статус |
| source | VARCHAR | Источник инцидента |
| adminComment | VARCHAR | Комментарий администратора |
| employeeReport | TEXT | Отчет сотрудника |
| cause | VARCHAR | Причина |
| resolution | VARCHAR | Решение |
| assignedToUserId | UUID / INT | Внешний ключ → Users(id) |
| reportedByUserId | UUID / INT | Внешний ключ → Users(id) |
| isReportSubmitted | BOOLEAN | Отчет отправлен |
| isReportApproved | BOOLEAN | Отчет утвержден |
| createdAt | DATETIME | Дата создания записи |
| updatedAt | DATETIME | Дата обновления записи |
| Migratons | | |
| id | UUID / INT | Первичный ключ |
| timestamp | BIGINT | Время |
| name | VARCHAR | Название |

На рисунке 4.1 отображена схема базы данных.

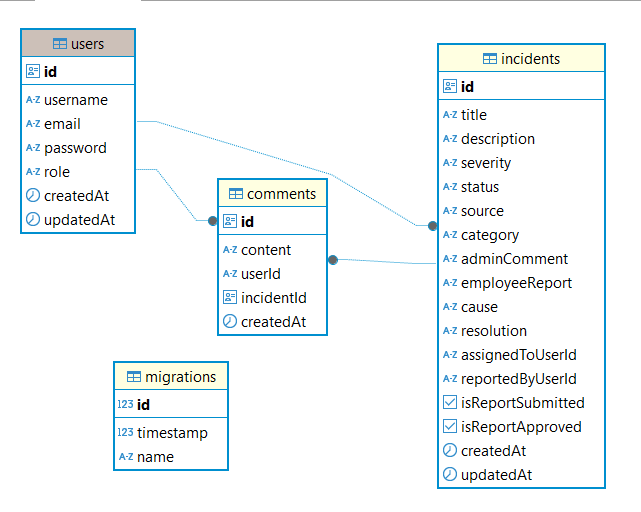


Рисунок 4.1 – Схема базы данных

Схема приведена к третьей нормальной форме следующим образом:

* в таблице соблюдена атомарность значений (1НФ);
* в таблице нет частичных зависимостей от составного ключа, так как используются простые первичные ключи (2НФ);
* все атрибуты зависят только от первичного ключа и не зависят друг от друга (3НФ), что исключает логические аномалии при обновлении, вставке или удалении данных.

**5 Реализация ПС через UML-диаграммы**

Цель: представить детали реализации программного средства с помощью UML-диаграмм.

Диаграмма классов представлена на рисунке 5.1.

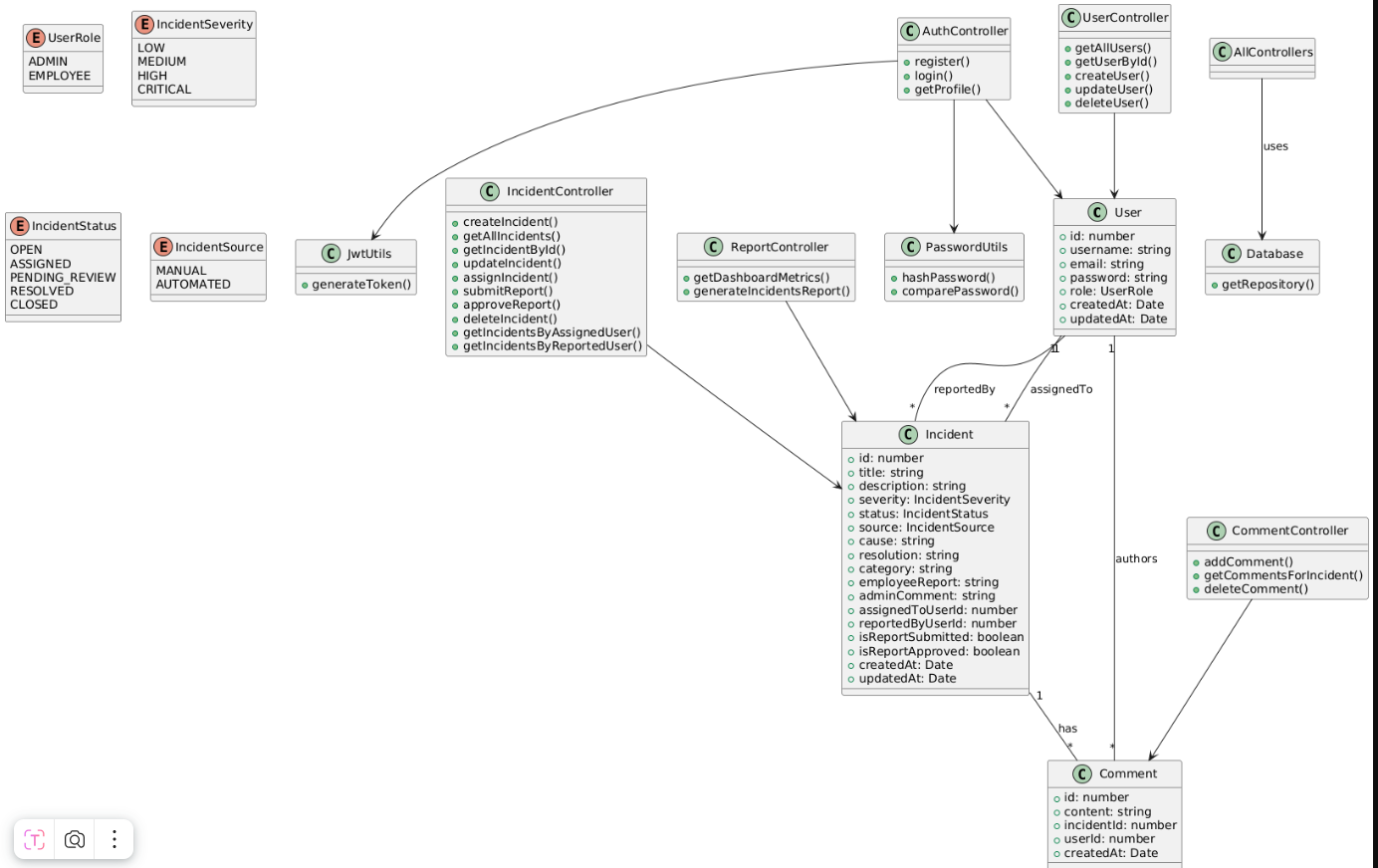
****

Рисунок 5.1 – Диаграмма классов

Диаграмма последовательности представлена на рисунке 5.2.

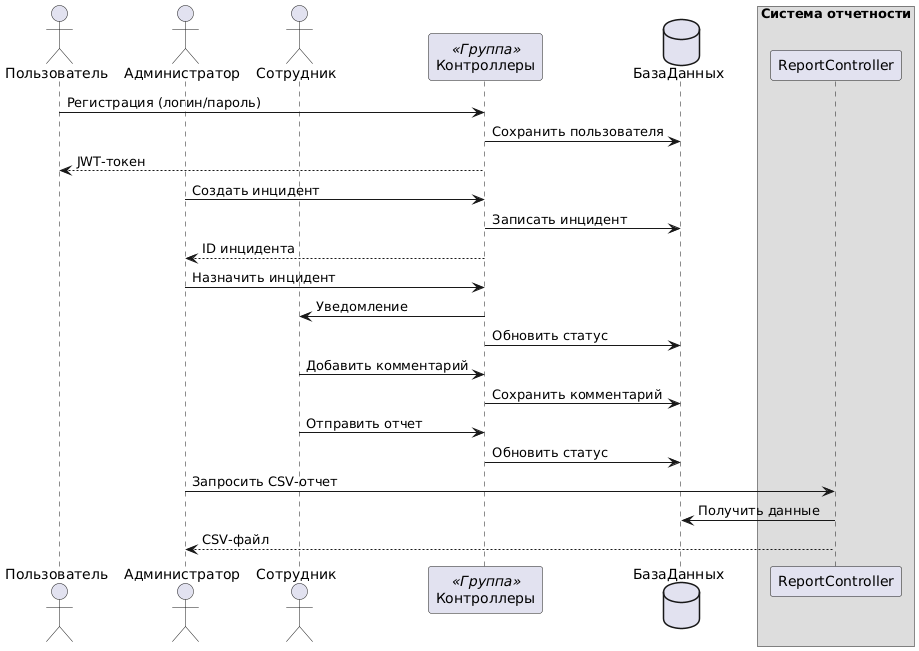


Рисунок 5.2 – Диаграмма последовательности

Диаграмма состояний представлена на рисунке 5.3.

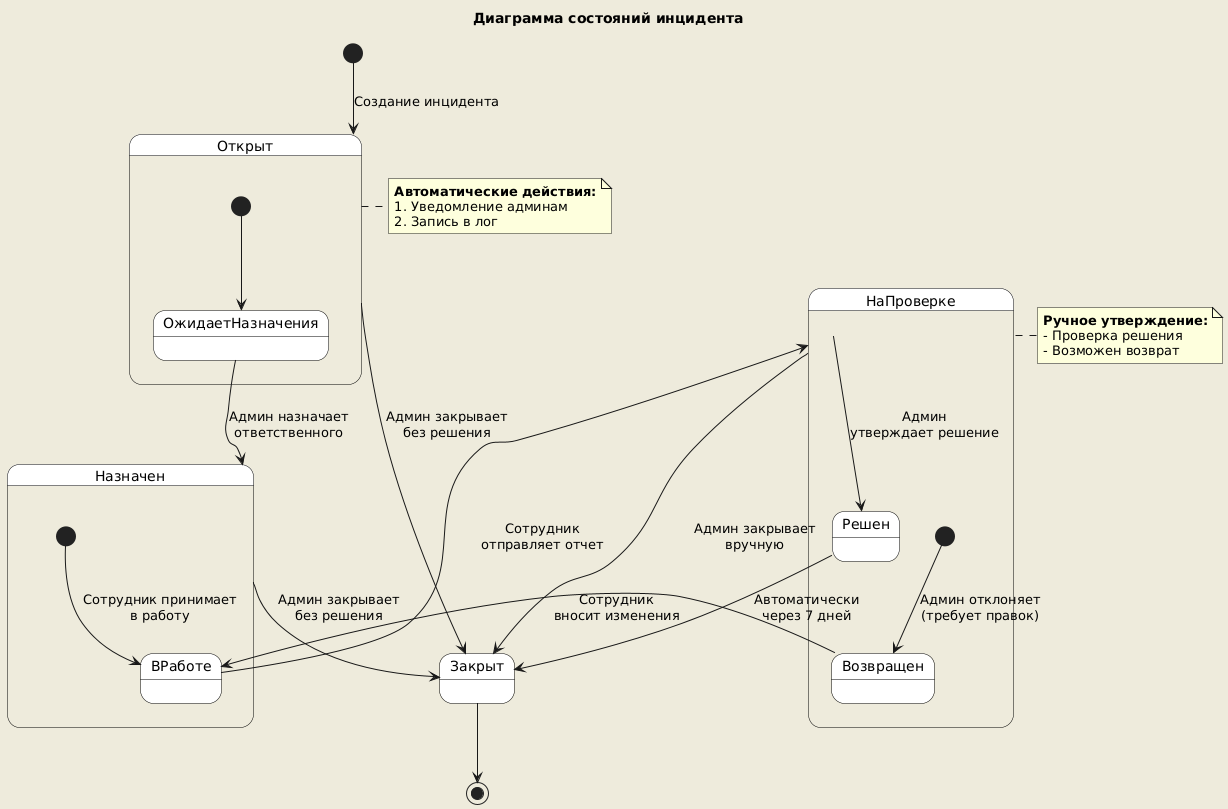


Рисунок 5.3 – Диаграмма состояний

Диаграмма компонентов представлена на рисунке 5.4.

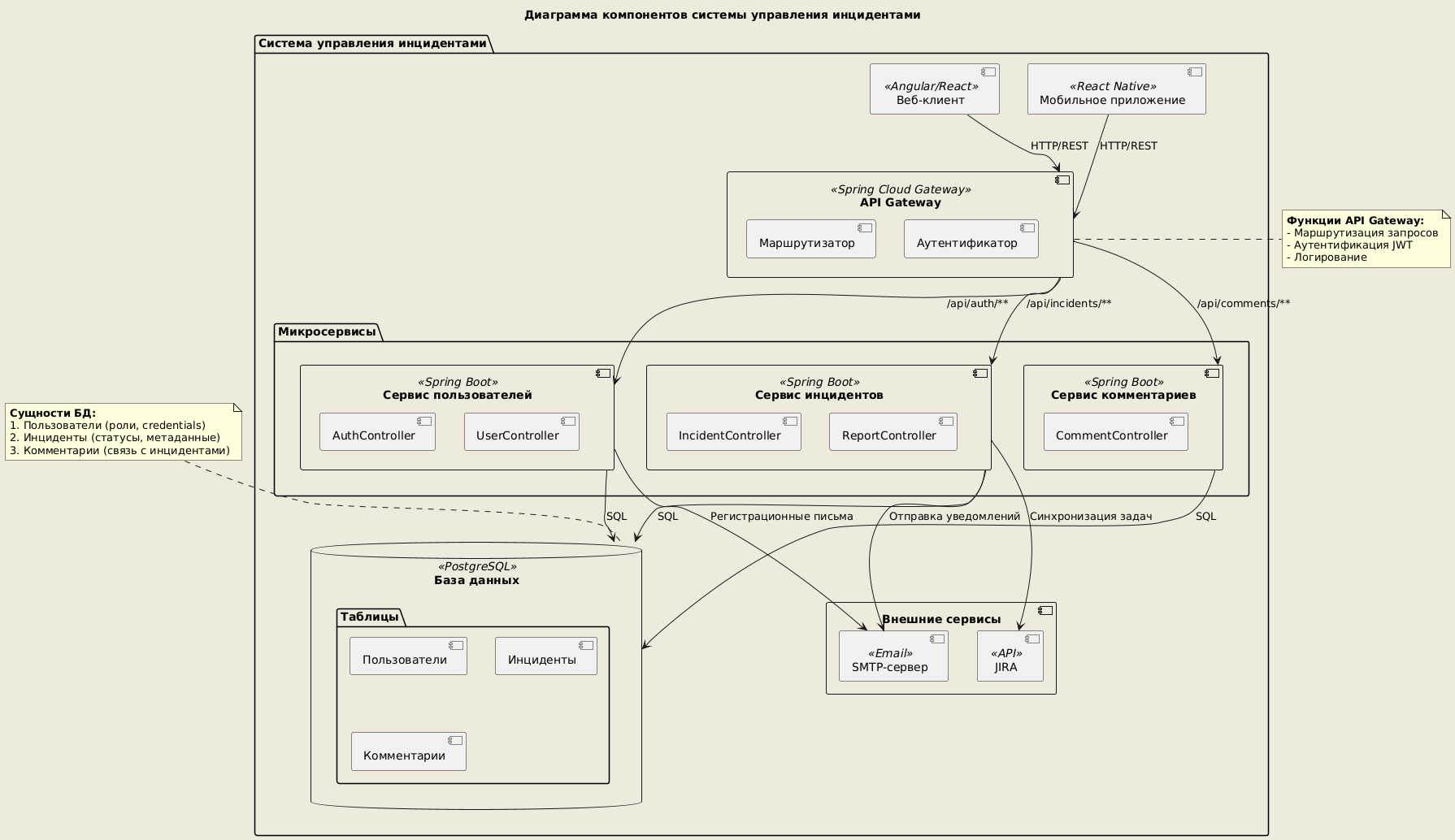


Рисунок 5.4 – Диаграмма компонентов

Диаграмма развертывания представлена на рисунке 5.5.

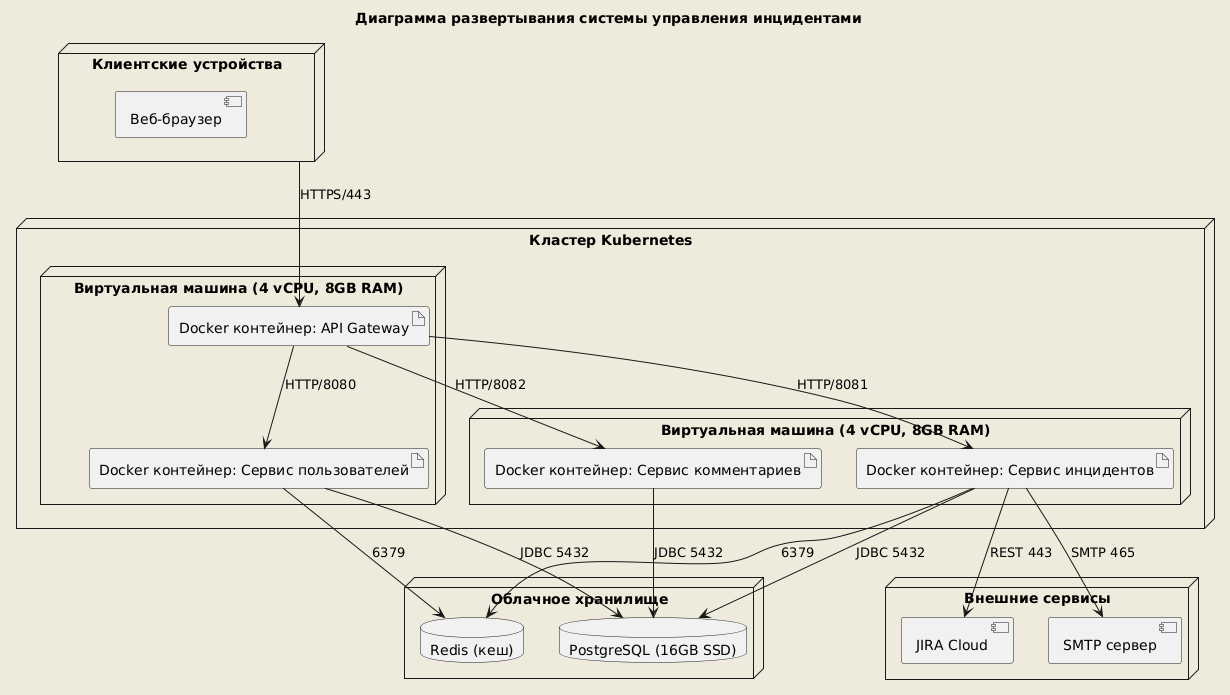


Рисунок 5.5 – Диаграмма развертывания

Диаграмма классов – отражает основные сущности системы: сущности БД *User + UserRole, Incident + IncidentStatus/Severity/Source, Comment,* контроллеры*, AuthController* (регистрация/авторизация*),UserController* (CRUD пользователей*),IncidentController* (управление инцидентами), *CommentController* (комментарии),*ReportController* (аналитика и отчеты), их свойства и взаимосвязи. Представлена структура данных и логика взаимодействия между объектами.

Диаграмма последовательности – описывает порядок взаимодействия объектов при выполнении ключевых операций.

Диаграмма состояний – отображает жизненный цикл инцидента: от рассмотрения до закрытия.

Диаграмма компонентов – показывает, как клиентская часть (*frontend*), сервер (*backend*), БД, система авторизации между собой.

Диаграмма развертывания – демонстрирует развертывание компонентов системы в реальной среде: браузер пользователя, веб-сервер, база данных, сторонние *API*.

**6 Разработка документации к ПС с Open API и оценка качества  
 программного кода**

Цель: разработать документацию к ПС с *Open API* и оценить качество программного кода.

Для обеспечения прозрачности, масштабируемости и простоты интеграции сторонних сервисов с разработанным программным средством была реализована документация к *REST* *API* с использованием спецификации *OpenAPI* *3.0*. Документация позволяет подробно описать каждую конечную точку *API*, её параметры, формат запроса и ответа, а также возможные коды ошибок.

Документация автоматически генерируется при помощи библиотеки *Swagger* и доступна по маршруту */api-docs*. В рамках документации реализовано описание следующих ключевых операций:

* регистрация нового пользователя;
* управление инцидентами;
* комментирование;
* просмотр и фильтрация инцидентов;
* получение статистических данных.

Пример описания эндпоинта для управления аутентификацией пользователей:

*/\*\**

*\* @swagger*

*\* tags:*

*\* name: Authentication*

*\* description: Управление аутентификацией пользователей*

*\*/*

*/\*\**

*\* @swagger*

*\* /api/auth/register:*

*\* post:*

*\* summary: Регистрация нового пользователя*

*\* tags: [Authentication]*

*\* requestBody:*

*\* required: true*

*\* content:*

*\* application/json:*

*\* schema:*

*\* type: object*

*\* required:*

*\* - username*

*\* - email*

*\* - password*

*\* properties:*

*\* username:*

*\* type: string*

*\* minLength: 3*

*\* maxLength: 30*

*\* example: johndoe*

*\* email:*

*\* type: string*

*\* format: email*

*\* example: john@example.com*

*\* password:*

*\* type: string*

*\* minLength: 6*

*\* example: securePassword123*

*\* responses:*

*\* 201:*

*\* description: Пользователь успешно зарегистрирован*

*\* content:*

*\* application/json:*

*\* schema:*

*\* $ref: '#/components/schemas/AuthResponse'*

*\* 400:*

*\* description: Ошибка валидации*

*\* content:*

*\* application/json:*

*\* schema:*

*\* $ref: '#/components/schemas/ErrorResponse'*

*\* 409:*

*\* description: Пользователь с таким email или username уже существует*

*\*/*

Для оценки качества программного кода использовались следующие критерии:

1 Структурированность и читаемость: код разделён по логическим модулям – настройки конфигурации, контроллеры, сервисы и вспомогательные утилиты. Используется единый стиль именования переменных и функций, соблюдаются отступы и правила оформления.

2 Логирование: для упрощения отладки и мониторинга внедрено логирование через отдельный модуль *console.error*, который регистрирует ошибки и ключевые события.

3 Защита конфиденциальных данных: данные конфигурации (пароли, токены, адреса сервисов) вынесены в .*env*-переменные, что соответствует лучшим практикам безопасности.

4 Повторное использование кода: логика работы с базой данных написана в *AppDataSource*

Таким образом, структура и реализация программного кода соответствуют современным требованиям к разработке веб-приложений. Программа демонстрирует высокую степень модульности, расширяемости и устойчивости к ошибкам, что обеспечивает её пригодность к дальнейшему масштабированию и сопровождению.

**7 Аутентификация и авторизация пользователей**

Цель: реализация системы аутентификации и авторизации пользователей ПС и механизмов обеспечения безопасности данных.

В разработанном программном средстве для реализации системы аутентификации и авторизации используется механизм *JSON Web Token (JWT)*. Это современный и безопасный способ управления сессиями без использования серверного хранения токенов.

В процессе реализации использованы следующие основные компоненты:

* express.js – серверный фреймворк;
* jsonwebtoken – библиотека для создания и проверки JWT;
* bcrypt – библиотека для хэширования паролей;
* dotenv – для безопасного хранения секретных ключей и конфигураций.

Вот как это работает шаг за шагом:

1 Установка необходимых пакетов.

2 Создание сервера Express:

const express = require('express');

const cors = require('cors');

const config = require('./config');

const logger = require('./utils/logger');

const { initDatabase } = require('./models');

const errorHandler = require('./middleware/error.middleware');

const installmentRoutes = require('./routes/installment.routes');

const paymentRoutes = require('./routes/payment.routes');

const calculationRoutes = require('./routes/calculation.routes');

const templateRoutes = require('./routes/installmentTemplate.routes');

const transactionRoutes = require('./routes/transaction.routes');

const cardRoutes = require('./routes/card.routes');

const app = express();

app.use(cors(config.cors));

app.use(express.json());

app.get('/api/health', (req, res) => {

res.json({ status: 'ok' });

});

app.use('/api/installments', installmentRoutes);

app.use('/api/payments', paymentRoutes);

app.use('/api/calculations', calculationRoutes);

app.use('/api/templates', templateRoutes);

app.use('/api/transactions', transactionRoutes);

app.use('/api/cards', cardRoutes);

app.use(errorHandler);

const startServer = async () => {

try {

await initDatabase(false); // false - не пересоздавать таблицы при каждом запуске

app.listen(config.server.port, () => {

logger.info(`Сервер запущен на порту ${config.server.port}`);

});

} catch (error) {

logger.error('Ошибка при запуске сервера:', error);

process.exit(1);

}

};

startServer();

3 Регистрация пользователя.

Route регистрации реализован на маршруте */register*. Пользовательские данные валидируются, пароль хэшируется с использованием *bcrypt*, после чего пользователь сохраняется в базе данных:

const bcrypt = require('bcrypt');

const jwt = require('jsonwebtoken');

exports.register = async (req, res) => {

const { email, password } = req.body;

const hashedPassword = await bcrypt.hash(password, 10);

const newUser = new User({ email, password: hashedPassword });

await newUser.save();

res.status(201).json({ message: 'Регистрация прошла успешно' });

};

4 Аутентификация пользователя.

Аутентификация осуществляется по маршруту */login*. После успешной проверки логина и пароля формируется *JWT*-токен:

exports.login = async (req, res) => {

const { email, password } = req.body;

const user = await User.findOne({ email });

const isValidPassword = await bcrypt.compare(password, user.password);

if (!isValidPassword) {

return res.status(401).json({ message: 'Неверные учетные данные' });

}

const token = jwt.sign({ userId: user.\_id }, process.env.JWT\_SECRET, { expiresIn: '24h' });

res.json({ token });

};

5 Защита маршрутов.

Для защиты маршрутов используется *middleware*-функция *authenticate*, которая проверяет наличие и валидность *JWT*:

const jwt = require('jsonwebtoken');

exports.authenticate = (req, res, next) => {

const token = req.headers.authorization?.split(' ')[1];

if (!token) return res.status(401).json({ message: 'Токен не предоставлен' });

try {

const decoded = jwt.verify(token, process.env.JWT\_SECRET);

req.user = decoded;

next();

} catch (error) {

return res.status(401).json({ message: 'Недействительный токен' });

}

};

6 Создание защищённого маршрута.

Пример защищённого маршрута:

router.get('/check', authenticate, authController.check);

7 Запуск сервера.

В проекте реализованы следующие меры обеспечения безопасности:

1 Хэширование паролей: все пароли хранятся в зашифрованном виде с использованием *bcrypt*, что предотвращает их утечку в случае компрометации базы данных.

2 *JWT* для сессий: вместо хранения сессий на сервере используются *JWT*, что упрощает масштабирование и снижает нагрузку.

3 Валидация данных: используется предварительная валидация запросов через middleware, предотвращающая *XSS* и *SQL*-инъекции.

4 Разграничение доступа: реализована система *middleware*-функций, позволяющая ограничивать доступ к определённым маршрутам (например, только для авторизованных пользователей).

Изменения в архитектуре ПС

Вследствие реализации аутентификации и авторизации была добавлена новая архитектурная подсистема:

1 Модуль *authController* для обработки регистрации, логина и проверки токена.

2 *Middleware authenticate* для защиты маршрутов.

3 *JWT*-конфигурация добавлена в *config.js*:

jwt: {

secret: process.env.JWT\_SECRET || 'your-secret-key',

expiresIn: '24h',

}

Конечная архитектура ПС включает следующие логические слои:

* контроллеры (логика обработки запросов);
* сервисы (бизнес-логика);
* middleware (валидация, защита маршрутов);
* конфигурационные файлы (настройки, ключи);
* модель пользователей (взаимодействие с БД).

В результате выполнения задания была успешно реализована система аутентификации и авторизации пользователей с использованием *JWT*. Обеспечена защита маршрутов и разграничение прав доступа. Все пользовательские пароли шифруются, а данные валидируются, что повышает уровень безопасности. В архитектуру программного средства были добавлены модули для обработки авторизации и *middleware* для проверки токенов, что сделало систему более защищённой и масштабируемой.

**8 Тестирование программного средства**

Цель: разработать *unit*- и интеграционные тесты для оценки работоспособности реализованной функциональности ПС.

В тестировании, для проверки, корректности работы программного средства делаются определенные действия и сверяют полученный результат

с ожидаемым. Другими словами – моделируется ситуация работы программного средства. Чтобы описать шаги, создают тест-кейсы.

Тест-кейсы для проверки уровня базовых пользовательских требований приведены в таблице 8.1.

Таблица 8.1 – Тест-кейсы для проверки уровня базовых пользовательских   
 требований

|  |  |  |  |
| --- | --- | --- | --- |
| *ID* | Заглавие тест-кейса | Шаги тест-кейса | Ожидаемый результат |
| *UC-1* | Вход сотрудника в систему | 1. Перейти на страницу авторизации.  2. Ввести логин и пароль.  3. Нажать кнопку «Войти». | Сотрудник успешно авторизуется и переходит в личный кабинет. |
| *UC-2* | Вход администратора в систему | 1. Перейти на главную страницу. 2. Ввести логин и пароль администратора. 3. Нажать кнопку «Войти». | Администратор успешно авторизуется и попадает на свою главную страницу. |
| *UC-3* | Назанчение инцидента сотруднику | 1. Войти как администратор.  2. Перейти на страницу «Инциденты».  3. Назначить инцидент сотруднику. | Инцидент успешно назначен и добавлен в задачи сотрудника |
| *UC-4* | Просмотр задач | 1. Войти в систему как сотрудник. 2. Перейти на страницу «Мои задачи». 3. Выбрать назначенный инцидент. | Инцидент успешно открыт для решения |
| *UC-5* | Отправка отчета по инциденту | 1. Войти как сотрудник.  2. Перейти на «Мои задачи».  3. Выбрать инцидент.  4.Написать отчет.  5. Отправить отчет. | Отчет отправлен администратору на утверждение |
| *UC-6* | Комментировать инцидент | 1. Войти как сотрудник/администратор.  2. Открыть инцидент.  3. Оставить комментарий | Комментарий к инциденту успешно добавлен |

Продолжение таблицы 8.1

|  |  |  |  |
| --- | --- | --- | --- |
| *ID* | Заглавие тест-кейса | Шаги тест-кейса | Ожидаемый результат |
| *UC-7* | Отправка отчета по инциденту на доработку | 1. Войти как администратор.  2. Перейти в «На проверке».  3. Открыть инцидент.  4. Отправить отчет на доработку. | Отчет успешно отправлен на доработку |
| *UC-8* | Утвердить отчет по инциденту | 1. Войти как администратор.  2. Перейти в «На проверке».  3. Открыть инцидент.  4. Утвердить отчет | Инцидент в статусе «Завершен» |
| *UC-9* | Просмотр статистики по инцидентам | 1. Войти как администратор/сотрудник.  2. Перейти на Главную страницу. | Отображается статиски инцидентов |
| *UC-10* | Просмотр монитроинга системы безопаности | 1. Войти как администратор.  2. Перейти на «Мониторинг». | Отображаются показатели системы безопасности |
| *UC-11* | Выход пользователя из системы | 1. Войти в систему.  2. Нажать «Выход». | Пользователь выходит из системы и попадает на страницу входа. |

Результаты тестирования подтвердили, что разработанное программное обеспечение полностью соответствует всем базовым требованиям пользователей и готово к дальнейшему внедрению и эксплуатации.

*UNIT*-тесты использовались для проверки отдельных компонентов системы с целью убедиться, что каждый компонент работает в соответствии с заранее установленными требованиями. Ниже приведены примеры кода *UNIT*-тестов для проверки позитивных и негативных сценариев.

***comment.controller.test.js***

const { addComment, getCommentsForIncident, deleteComment } = require('../controllers/comment.controller');

const { AppDataSource } = require('../utils/database');

jest.mock('../utils/database', () => ({

  AppDataSource: {

    getRepository: jest.fn(),

  },

}));

const mockResponse = () => {

  const res = {};

  res.status = jest.fn(() => res);

  res.json = jest.fn(() => res);

  return res;

};

describe('Comment Controller', () => {

  const mockRepo = {

    create: jest.fn(),

    save: jest.fn(),

    find: jest.fn(),

    findOne: jest.fn(),

    remove: jest.fn(),

  };

  beforeEach(() => {

    AppDataSource.getRepository.mockReturnValue(mockRepo);

    jest.clearAllMocks();

  });

  describe('addComment', () => {

    it('should return 401 if no user', async () => {

      const req = { body: {}, user: null };

      const res = mockResponse();

      await addComment(req, res);

      expect(res.status).toHaveBeenCalledWith(401);

    });

    it('should save comment and return 201', async () => {

      const req = { body: { incidentId: 1, content: 'test' }, user: { userId: 42 } };

      const res = mockResponse();

      const savedComment = { id: 1, content: 'test' };

      mockRepo.create.mockReturnValue(savedComment);

      mockRepo.save.mockResolvedValue(savedComment);

      await addComment(req, res);

      expect(mockRepo.save).toHaveBeenCalledWith(savedComment);

      expect(res.status).toHaveBeenCalledWith(201);

    });

  });

  describe('getCommentsForIncident', () => {

    it('should fetch comments and return 200', async () => {

      const req = { params: { incidentId: 1 } };

      const res = mockResponse();

      mockRepo.find.mockResolvedValue([{ id: 1, content: 'comment' }]);

      await getCommentsForIncident(req, res);

      expect(res.status).toHaveBeenCalledWith(200);

    });

  });

  describe('deleteComment', () => {

    it('should return 401 if no user', async () => {

      const req = { params: { id: 1 }, user: null };

      const res = mockResponse();

      await deleteComment(req, res);

      expect(res.status).toHaveBeenCalledWith(401);

    });

    it('should return 404 if comment not found', async () => {

      const req = { params: { id: 1 }, user: { userId: 42 } };

      const res = mockResponse();

      mockRepo.findOne.mockResolvedValue(null);

      await deleteComment(req, res);

      expect(res.status).toHaveBeenCalledWith(404);

    });

    it('should return 403 if user has no rights', async () => {

      const req = { params: { id: 1 }, user: { userId: 99, role: 'user' } };

      const res = mockResponse();

      mockRepo.findOne.mockResolvedValue({ id: 1, userId: 42 });

      await deleteComment(req, res);

      expect(res.status).toHaveBeenCalledWith(403);

    });

    it('should delete comment for admin', async () => {

      const req = { params: { id: 1 }, user: { userId: 99, role: 'admin' } };

      const res = mockResponse();

      const comment = { id: 1, userId: 42 };

      mockRepo.findOne.mockResolvedValue(comment);

      await deleteComment(req, res);

      expect(mockRepo.remove).toHaveBeenCalledWith(comment);

      expect(res.status).toHaveBeenCalledWith(200);

    });

  });

});

***dashboard.controller.test.js***

const { getDashboardMetrics, generateIncidentsReport } = require('../controllers/dashboard.controller');

const { AppDataSource } = require('../utils/database');

jest.mock('../utils/database', () => ({

  AppDataSource: {

    getRepository: jest.fn(),

  },

}));

describe('Dashboard Controller', () => {

  const mockRepo = {

    count: jest.fn(),

    createQueryBuilder: jest.fn(),

    find: jest.fn(),

  };

  const mockQueryBuilder = {

    select: jest.fn().mockReturnThis(),

    addSelect: jest.fn().mockReturnThis(),

    groupBy: jest.fn().mockReturnThis(),

    getRawMany: jest.fn(),

  };

  const mockResponse = () => {

    const res = {};

    res.status = jest.fn(() => res);

    res.json = jest.fn(() => res);

    res.setHeader = jest.fn();

    res.send = jest.fn();

    return res;

  };

  beforeEach(() => {

    AppDataSource.getRepository.mockReturnValue(mockRepo);

    mockRepo.createQueryBuilder.mockReturnValue(mockQueryBuilder);

    jest.clearAllMocks();

  });

  describe('getDashboardMetrics', () => {

    it('should return dashboard metrics', async () => {

      const req = {};

      const res = mockResponse();

      mockQueryBuilder.getRawMany

        .mockResolvedValueOnce([{ status: 'open', count: '5' }])

        .mockResolvedValueOnce([{ severity: 'high', count: '2' }]);

      mockRepo.count.mockResolvedValueOnce(10).mockResolvedValueOnce(3);

      const resolved = [{

        createdAt: new Date(Date.now() - 3 \* 86400000),

        updatedAt: new Date(),

      }];

      mockRepo.find.mockResolvedValue(resolved);

      await getDashboardMetrics(req, res);

      expect(res.status).toHaveBeenCalledWith(200);

    });

  });

  describe('generateIncidentsReport', () => {

    it('should generate CSV report', async () => {

      const req = {};

      const res = mockResponse();

      const incident = {

        id: 1,

        title: 'Test',

        description: 'Test',

        severity: 'high',

        status: 'open',

        cause: 'unknown',

        resolution: 'none',

        assignedToUserId: 1,

        reportedByUserId: 2,

        createdAt: new Date(),

        updatedAt: new Date(),

      };

      mockRepo.find.mockResolvedValue([incident]);

      await generateIncidentsReport(req, res);

      expect(res.setHeader).toHaveBeenCalledWith(expect.stringContaining('Content-Disposition'), expect.any(String));

      expect(res.status).toHaveBeenCalledWith(200);

    });

  });

});

***authController.test.ts***

// tests/authController.test.ts

import request from 'supertest';

import app from '../app'; // Предположим, что Express-приложение экспортировано из app.ts

import { AppDataSource } from '../utils/database';

import { User } from '../models/User';

beforeAll(async () => {

await AppDataSource.initialize();

await AppDataSource.getRepository(User).clear(); // Очистка базы для тестов

});

afterAll(async () => {

await AppDataSource.destroy();

});

describe('AuthController', () => {

const testUser = {

username: 'testuser',

email: 'test@example.com',

password: 'TestPass123',

role: 'employee'

};

let token = '';

test('register - should create a new user', async () => {

const res = await request(app)

.post('/api/auth/register')

.send(testUser);

expect(res.status).toBe(201);

expect(res.body).toHaveProperty('token');

expect(res.body.user).toHaveProperty('id');

});

test('register - should fail if user already exists', async () => {

const res = await request(app)

.post('/api/auth/register')

.send(testUser);

expect(res.status).toBe(400);

expect(res.body.message).toBe('User with this username or email already exists');

});

test('login - should authenticate with correct credentials', async () => {

const res = await request(app)

.post('/api/auth/login')

.send({ username: testUser.username, password: testUser.password });

expect(res.status).toBe(200);

expect(res.body).toHaveProperty('token');

token = res.body.token;

});

test('login - should fail with incorrect password', async () => {

const res = await request(app)

.post('/api/auth/login')

.send({ username: testUser.username, password: 'wrongpassword' });

expect(res.status).toBe(401);

expect(res.body.message).toBe('Invalid username or password');

});

test('getProfile - should return profile with valid token', async () => {

const res = await request(app)

.get('/api/auth/profile')

.set('Authorization', `Bearer ${token}`);

expect(res.status).toBe(200);

expect(res.body.user).toHaveProperty('email', testUser.email);

});

test('getProfile - should return 404 for invalid user', async () => {

// Мокинг токена с несуществующим ID

const fakeToken = require('../utils/jwt').generateToken({ id: 999999, role: 'employee' });

const res = await request(app)

.get('/api/auth/profile')

.set('Authorization', `Bearer ${fakeToken}`);

expect(res.status).toBe(404);

expect(res.body.message).toBe('User not found');

});

});

В ходе тестирования программного средства были проверены ключевые функциональные возможности, включая авторизацию пользователей, управление инцидентами, комментирование, а также просмотр статистики.

Для описания шагов тестирования и проверки ожидаемых результатов использовались тест-кейсы, приведенные в таблице 8.1, охватывающие основные пользовательские сценарии.

*UNIT*-тесты обеспечили проверку отдельных компонентов системы, включая регистрацию, создание и оплату рассрочек, подтверждение заявок администраторами и работу с шаблонами.

**9 Процесс развертывания программного средства**

Цель: описание процесса развертывания ПС.

Необходимые программы и компоненты. Для успешной установки и запуска веб-приложения необходимо наличие следующих компонентов:

– операционная система семейства *MacOS* или *Windows*;

– веб-сервер *Express*;

– зависимости (библиотеки) для серверной части приложения;

– клиентская часть *React.js*;

– зависимости (библиотеки);

– сервер базы данных *PostgreSQL*.

Последовательность установки. Для установки разработанного программного средства необходимо выполнить следующие шаги:

1. Назначить права для пользователя, под которым будет проводиться установка и настройка.
2. Установить и настроить *Node.js.*
3. Установить зависимости для серверной части.
4. Установить зависимости для клиентской части.
5. Установить и настроить сервер баз данных *PostgreSQL*.
6. Установить и настроить разработанное ПС.

Состав дистрибутива. В поставляемый конечному пользователю дистрибутив входят следующие элементы:

– скрипт генерации пустой базы данных;

– ярлык, содержащий ссылку на сайт.

Проверка работоспособности. Для проверки работоспособности веб-приложения необходимо:

– убедиться в том, что сервис *PostgreSQL* запущен;

– убедиться в том, что сервер веб-приложения запущен;

– убедиться в том, что клиентская часть веб-приложения запущена;

– используя любой браузер перейти на адрес *http://localhost:5173.*

В случае успеха в окне браузера откроется главная страница веб-приложения.

**10 Разработка руководства пользователя**

Цель: разработка руководства пользователя.

В таблице 10.1 описаны функции и задачи веб-приложения.

Таблица 10.1 – Описание функций и задач

|  |  |  |
| --- | --- | --- |
| Функции | Задачи | Описание |
| Просмотр статистики инцидентов | Пользователи могут просматривать статистику инцидентов | Сотрудники и администраторы могут видеть информацию по всем инцидентам системы безопасности |
| Просмотр мониторинга системы безопасности | Пользователи просматривать систему безопасности | Администраторы, перейдя на страницу «Мониторинг» видят все показатели системы безопасности |
| Управление инцидентами | Пользователи могут редактировать, назначать инциденты | Администраторы могу редактировать инциденты системы безопасности и назначать их сотрудникам |
| Отправка отчета по инциденту | Пользователи могут отправить отчет по инциденту | Сотрудник отправляет отчет по инциденту |
| Получение отчетов и статистики | Администраторы и сотрудники могут формировать отчеты и просматривать статистику по инцидентам | Отчёты могут содержать сводные данные по количеству активных и завершённых инцидентам, |

В таблице 10.2 описаны операции обработки данных для задач.

Таблица 10.2 – Описание реализуемых операций

|  |  |
| --- | --- |
| Операция | Описание |
| Регистрация пользователя | Пользователь создает аккаунт, указывая имя пользователя, электронную почту и пароль. Пользователь создается с ролью «Сотрудник» |
| Вход в систему | После регистрации сотрудник входит в личный кабинет с помощью логина и пароля |
| Просмотр задач | Сотрудник видит список всех своих задач, с подробной информацией по каждой |
| Отправление отчета | Заполнение отчета: проблема, решение, как решить, комментарии |
| Сортировка и фильтрация | Сотрудник сортирует свои задачи по статусу и другим критериям |
| Утверждение отчета | Администратор просматривает отчет сотрудника и утверждает его или отправляет на доработку |
| Генерация отчетов | Администратор или сотрудник формирует отчеты по инцидентам |
| Просмотр статистики | Администратор или сотрудник может видеть статистику по инцидентам |

В рамках проектирования и разработки программного средства автоматизации рабочего процесса администратора информационной безопасности реализован набор ключевых функций, направленных на повышение доступности и автоматизацию процессов быстрого реагирования на инциденты системы безопасности.

Пользователи системы могут регистрироваться, авторизоваться и управлять инцидентами, комментировать их, просматривать отчетсность.

Администратор может просматривать систему безопасности в режиме реального времени, назначать инциденты для их решения, редактировать инциденты.

Разработанное программное средство делает рабочий процесс администратора информационной безопасности более прозрачным, удобным и быстрым как для сотрудников, так и для администраторов, повышая общую эффективность работы системы.

**Выводы**

В рамках проекта было создано программное решение для автоматизации рабочих процессов администратора информационной безопасности. Разработка велась поэтапно — от проектирования архитектуры до финального тестирования и развёртывания системы, с соблюдением принципов последовательности и качества.

Проект основан на модульной архитектуре с чётким разделением на:

Клиентскую часть, реализованную с учётом современных принципов UX/UI-дизайна;

Серверную часть, разработанную с использованием Express.js и базы данных PostgreSQL;

Внешние сервисы, включая систему аутентификации и авторизации.

Особое внимание уделено интуитивной навигации в пользовательском интерфейсе, адаптированном под разные роли и категории пользователей.

Реализована функциональная клиентская часть, обеспечивающая эффективное взаимодействие с сервером и внешними модулями.

Разработана и оптимизирована реляционная база данных, отражающая ключевые бизнес-процессы.

Построены UML-диаграммы, включая диаграммы компонентов, состояний и развёртывания, что позволило визуализировать логику и структуру системы.

Внедрена система аутентификации и авторизации с использованием JWT, обеспечивающая безопасный доступ к защищённым маршрутам.

Проведено unit- и функциональное тестирование всех основных модулей, подтвердившее надёжность и стабильность работы приложения.

Разработаны механизмы валидации, разграничения прав доступа и промежуточной обработки (middleware) для проверки данных и предотвращения потенциальных угроз.

Успешно реализован и протестирован модуль управления инцидентами, повышающий уровень реагирования на события информационной безопасности.

Составлена документация API в формате OpenAPI (Swagger).

Подготовлено руководство пользователя с пошаговыми инструкциями по взаимодействию с системой.

Осуществлён процесс развёртывания, позволяющий надёжно запускать и масштабировать систему в продуктивной среде.

Созданное программное средство представляет собой стабильное, масштабируемое и функциональное решение, соответствующее современным требованиям к безопасности и удобству использования. В дальнейшем возможна его адаптация под новые бизнес-задачи, а также расширение за счёт внедрения дополнительных модулей и аналитических инструментов.