

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ТЕХНОЛОГИЧЕСКИЙ
УНИВЕРСИТЕТ «МИСИС»

ИНСТИТУТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ И КОМПЬЮТЕРНЫХ НАУК
КАФЕДРА АВТОМАТИЗИРОВАННОГО ПРОЕКТИРОВАНИЯ И ДИЗАЙНА
НАПРАВЛЕНИЕ 09.04.02 ИНФОРМАЦИОННЫЕ СИСТЕМЫ И ТЕХНОЛОГИИ

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА МАГИСТРА

на тему: Исследование и разработка веб-приложения для централизованного учета и мониторинга устройств с целью автоматизации на основе Python, Django, PostgreSQL и JavaScript

Студент _____

Руководитель работы _____

Нормоконтроль проведен _____

Проверка на заимствования проведена _____



И.А. Мельниченко

А.С. Оганесян

А.А. Петрыкина

Работа рассмотрена кафедрой и допущена к защите в ГЭК

Заведующий кафедрой _____ Е.Г. Коржов

Директор института _____ С.В. Солодов

Москва 2025

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ТЕХНОЛОГИЧЕСКИЙ
УНИВЕРСИТЕТ «МИСИС»

УТВЕРЖДАЮ

Институт ИТКН
Кафедра АПД
Направление 09.04.02 ИСТ

Зав. кафедрой _____
« » _____ 2025 г.

**ЗАДАНИЕ
НА ВЫПОЛНЕНИЕ ВЫПУСКНОЙ
КВАЛИФИКАЦИОННОЙ РАБОТЫ МАГИСТРА**

Студенту группы _____
(Ф.И.О. полностью)

1. Тема работы Исследование и разработка веб-приложения для централизованного учета и мониторинга устройств с целью автоматизации на основе Python, Django, PostgreSQL и JavaScript
2. Цели работы Исследование и внедрение надёжного, безопасного и масштабируемого веб-приложения для централизованного учета устройств с возможностью синхронизации данных с внешними сервисами
3. Исходные данные Исходные данные по устройствам, хранящиеся в excel и Jira, требование по безопасности и хранению данных внутри компании
4. Основная литература, в том числе:
 - 4.1. Монографии, учебники и т.п. Фаулер М. Архитектура корпоративных приложений. - М.: Вильямс, 2022. — 560 с; Сэм Ньюман. Микросервисы. Разработка и сопровождение. СПб.: Питер, 2022. 352 с. Ларсен М. Безопасная веб-разработка. Принципы и практика. - М.: ДМК Пресс, 2021. - 384 с. Vincent W. Django for Professionals. - Independently published, 2022. - 282 p
 - 4.2. Отчеты по НИР, диссертации, дипломные проекты и т.п. _____
 - 4.3. Периодическая литература Современные тренды разработки программного обеспечения: Agile, DevOps, CI/CD // ResearchGate. - URL: <https://www.researchgate.net/publication/371620522>
 - 4.4. Патенты _____
 - 4.5. Справочники и методическая литература (в том числе литература по методам обработки экспериментальных данных) Django documentation, Django REST Framework documentation, PostgreSQL Documentation, uWSGI documentation, Mozilla Developer Network (MDN)
5. Перечень основных этапов исследования и форма промежуточной отчетности по каждому этапу Анализ предметной области и постановки задачи; Сравнительный анализ архитектурных решений; Проектирование архитектуры приложения и базы данных; Разработка серверной части, API и клиентского;

Интеграция с внешними сервисами; Разработка и проведение тестирования; Развертывание системы;

6. Аппаратура и методики, которые должны быть использованы при проведении исследований
Сервер на базе Linux; компьютеры с ОС Windows или Linux для разработки и тестирования интерфейса;
Анализ и моделирование предметной области; Сравнительный анализ архитектурных решений;
Юнит-тестирование и интеграционное тестирование; Нагрузочное тестирование; Оценка безопасности;

7. Использование информационных технологий при проведении исследований
Использование Django (Python) как основной фреймворк для разработки серверной части приложения;
Использование СУБД PostgreSQL для управления базами данных для хранения информации;
Использование языка JavaScript для создания динамичных элементов интерфейса;
Использование Bootstrap для стилизации интерфейса и обеспечения адаптивности
Использование Jira API для интеграции с внешней системой управления проектами
Использование uWSGI и Nginx для развертывания приложения на сервере
Использование Git для управления версиями и совместной работы
Использование Docker для контейнеризации приложения и обеспечения удобства тестирования и развертывания
Использование Postman для тестирования REST API-интерфейсов
Использование Django TestCase для написания юнит- и интеграционных тестов
Использование Django Extensions для автоматической генерации диаграмм моделей и структур базы данных.

8. Перечень подлежащих разработке вопросов по экономике НИР

Согласовано: не предусмотрено

Консультант по экономике

9. Перечень подлежащих разработке вопросов по безопасности жизнедеятельности

Согласовано: не предусмотрено

Консультант по безопасности жизнедеятельности

10. Перечень подлежащих разработке вопросов по охране окружающей среды

Согласовано: не предусмотрено

Консультант по охране окружающей среды

11. Перечень (примерный) основных вопросов, которые должны быть рассмотрены и проанализированы в литературном обзоре
Современные архитектуры веб-приложений (монолит, микросервисы, serverless);
Сравнение языков программирования и фреймворков (Python, Django, JavaScript, PostgreSQL);
Безопасность веб-приложений (защита от CSRF, XSS, SQL-инъекций);
Методики тестирования веб-приложений (юнит-, интеграционное, нагрузочное тестирование)
Организация CI/CD и DevOps в процессе разработки; Интеграция с корпоративными системами (Jira, 1C ERP);

12. Перечень (примерный) иллюстрированного материала
Схема архитектуры веб-приложения;
Диаграммы моделей базы данных; Диаграммы взаимодействия компонентов системы; Схемы интеграции с Jira и 1C ERP; Скриншоты пользовательского интерфейса; Таблицы с результатами тестирования;

13. Руководитель диссертации

кандидат технических наук, старший преподаватель

Мельниченко Илья Ашотович

(Должность, звание, ф.и.о.)

(подпись)

14. Консультанты (с указанием относящихся к ним разделов)

Дата выдачи задания

Задание принял к исполнению студент

(подпись)

АННОТАЦИЯ

Выпускная работа посвящена исследованию и разработке веб-приложения для централизованного учёта и мониторинга устройств с целью автоматизации процессов управления устройствами.

Целью исследования являлось выполнение разработки надёжного, безопасного и масштабируемого программного обеспечения.

В рамках работы проведён анализ современных технологий и подходы к архитектуре в области веб-разработки, обоснован выбор монолитной архитектуры и технологического стека. Спроектирована база данных на основе СУБД PostgreSQL. Разработано веб-приложение с серверной частью на Python, фреймворк Django с REST API, с клиентской частью на JavaScript и Bootstrap, а также реализованы интеграции с внешними сервисами: JIRA и 1с ERP. Разработка сопровождалась комплексным тестированием, включая юнит и нагрузочное тестирование.

В результате выпускной работы создано функциональное приложение, готовое к эксплуатации в условиях корпоративной среды.

Выпускная работа изложена на 107 страницах, содержит 30 рисунков, 7 таблиц, список использованных источников из 73 наименований.

ANNOTATION

The thesis is devoted to the research and development of a web application for centralized accounting and monitoring of devices in order to automate device management processes.

The purpose of the study was to develop reliable, secure and scalable software.

The work included an analysis of modern technologies and approaches to architecture in the field of web development, and the choice of a monolithic architecture and technology stack was substantiated. A database based on the PostgreSQL DBMS was designed. A web application with a server part in Python, a Django framework with REST API, with a client part in JavaScript and Bootstrap was developed, and integrations with external services were implemented: JIRA and 1c ERP. The development was accompanied by comprehensive testing, including unit and load testing.

As a result of the thesis, a functional application was created, ready for use in a corporate environment. The thesis is presented on 107 pages, contains 30 figures, 7 tables, a list of references from 73 titles.

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ.....	9
1. ТЕОРЕТИЧЕСКИЕ АСПЕКТЫ ВЕБ-РАЗРАБОТКИ.....	11
1.1 АНАЛИЗ ТЕКУЩИХ ТРЕНДОВ В ВЕБ-РАЗРАБОТКЕ.....	11
1.2 ОБЗОР СОВРЕМЕННЫХ ТЕХНОЛОГИЙ И ИНСТРУМЕНТОВ.....	13
1.2.1 Языки программирования и фреймворки.....	13
1.2.2 Базы данных.....	14
1.2.3 Вспомогательные инструменты.....	15
1.3 ПРИНЦИПЫ ПРОЕКТИРОВАНИЯ И РАЗРАБОТКИ ВЕБ-ПРИЛОЖЕНИЙ	16
1.3.1 Модульность и масштабируемость	16
1.3.2 Архитектурные паттерны: MVC и MVT.....	17
1.3.3 Принципы чистой архитектуры и паттерны.....	18
1.4 МЕТОДОЛОГИИ РАЗРАБОТКИ: AGILE, DEVOPS, CI/CD	18
1.4.1 Agile, Kanban, Waterfall	19
1.4.2 DevOps автоматизация	20
1.4.3 CI/CD.....	21
1.5 ВЫВОДЫ ПО ГЛАВЕ	23
2. РАЗРАБОТКА ВЕБ-ПРИЛОЖЕНИЯ.....	24
2.1 АРХИТЕКТУРА ПРОЕКТА	24
2.1.1 Выбор архитектурной модели	24
2.1.2 Выбор паттерна проектирования.....	27
2.1.3. Реализация модели, представлений и шаблонов.....	29
2.1.4 Структура и компоненты серверной части	32
2.1.5. Вывод.....	36
2.2 ВЫБОР И ОПИСАНИЕ ИНСТРУМЕНТОВ РАЗРАБОТКИ.....	37
2.2.1 Выбор языка программирования	37
2.2.2 Выбор фреймворка.....	40
2.2.3 Выбор СУБД.....	41
2.2.4 Выбор технологий клиентской части	42

2.2.5 Выбор дополнительных библиотек.....	42
2.2.6 Выводы	44
2.3 ПРОЕКТИРОВАНИЕ БАЗЫ ДАННЫХ.....	44
2.3.1 Определение требований к данным	44
2.3.2 Проектирование структуры базы данных.....	48
2.3.3 Нормализация данных.....	51
2.3.4 Выводы	54
2.4 РАЗРАБОТКА СЕРВЕРНОЙ ЧАСТИ	55
2.4.1 Структура серверного части	55
2.4.2 Реализация обработки запросов	57
2.4.3 Реализация бизнес-логики	59
2.4.4 Работа с базой данных через ORM.....	61
2.4.5 Реализация API.....	64
2.4.6 Обеспечение безопасности серверной части	66
2.5 РАЗРАБОТКА КЛИЕНТСКОЙ ЧАСТИ	68
2.5.1 Структура клиентской части	68
2.5.2 Разработка пользовательских интерфейсов	71
2.5.3 Работа с данными на клиенте.....	76
2.5.4 Адаптивность и оформление интерфейсов	78
2.6 ИНТЕГРАЦИЯ С ВНЕШНИМИ СЕРВИСАМИ.....	82
2.6.1 Выбор сервисов для интеграции.....	82
2.6.2 Техническая реализация интеграции.....	83
2.7 ВЫВОДЫ ПО ВТОРОЙ ГЛАВЕ	85
3 ТЕСТИРОВАНИЕ И ВНЕДРЕНИЕ	87
3.1 Подходы к тестированию ВЕБ-ПРИЛОЖЕНИЯ.....	87
3.2 ЮНИТ-ТЕСТИРОВАНИЕ И ИНТЕГРАЦИОННОЕ ТЕСТИРОВАНИЕ	89
3.3 НАГРУЗОЧНОЕ ТЕСТИРОВАНИЕ И АНАЛИЗ ПРОИЗВОДИТЕЛЬНОСТИ.....	91
3.4 ОЦЕНКА БЕЗОПАСНОСТИ ВЕБ-ПРИЛОЖЕНИЯ.....	93
3.5 РАЗВЕРТЫВАНИЕ ПРИЛОЖЕНИЯ	95
3.6 ВЫВОДЫ ПО ГЛАВЕ	97

ЗАКЛЮЧЕНИЕ.....	98
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ	100

ВВЕДЕНИЕ

Актуальность работы заключается в том, что с ростом компании и увеличением числа устройств, которые требуют учёта, становится очевидной необходимость автоматизации этих процессов. Ранее использовавшиеся для этого инструменты, такие как Excel и Jira, стали недостаточными, поскольку информация передавалась устно, а данные, хранящиеся в таблицах, были разрознены и не всегда актуальны. В условиях роста и расширения компании возникла потребность в создании централизованной системы учёта, которая позволяла бы собирать, хранить и обновлять данные в одном месте, а также обеспечивать возможность синхронизации с другими корпоративными системами, такими как Jira и 1С ERP. Это поднимает актуальность разработки веб-приложения для централизованного учёта и мониторинга устройств.

Основной проблемой является отсутствие единого источника данных для учёта и мониторинга устройств, что приводит к неудобству в их управлении, дублированию информации и снижению её актуальности. Для решения этой проблемы была поставлена задача разработать веб-приложение, которое обеспечит централизованный учёт устройств, их привязку к проектам и программам, а также интеграцию с существующими корпоративными сервисами. Кроме того, задача включала в себя обеспечение безопасности данных, разработку эффективных бизнес-процессов и тестирование приложения на различных уровнях.

Целью исследования является разработка безопасного, надёжного и масштабируемого веб-приложения для централизованного учёта устройств, реализующего оптимальную архитектуру, современные принципы проектирования и интеграцию с внешними корпоративными сервисами.

Задачи исследования.

- Провести анализ современных архитектурных подходов в веб-разработке.
- Выбрать и обосновать технологический стек проекта.
- Спроектировать структуру базы данных и архитектуру веб-приложения.

- Разработать серверную и клиентскую части приложения.
- Интегрировать систему с внешними сервисами Jira и 1С ERP.
- Провести комплексное тестирование функционала, производительности и безопасности.
- Выполнить развертывание приложения в корпоративной инфраструктуре.

Объект исследования - корпоративные веб-приложения для учёта и мониторинга устройств.

Предмет исследования - методы и средства проектирования, разработки, тестирования и развертывания веб-приложений с применением фреймворка Django, СУБД PostgreSQL и интеграцией с внешними корпоративными системами.

В процессе работы применялись следующие методы исследования.

- Методы сравнительного анализа архитектурных решений.
- Анализ и моделирование предметной области.
- Проектирование баз данных.
- Разработка программных модулей средствами Python и Django.
- Функциональное, интеграционное и нагрузочное тестирование.
- Статический и динамический анализ безопасности.

Работа состоит из введения, трёх глав основной части, заключения, списка литературы и приложений.

- В первой главе рассмотрены теоретические аспекты и современные технологии веб-разработки.
- Во второй главе описан процесс проектирования и разработки системы.
- В третьей главе приведены методы тестирования, анализа производительности, безопасности и процедуры развертывания.
- В заключении подведены итоги и даны рекомендации по дальнейшему развитию проекта.

1. Теоретические аспекты веб-разработки

1.1 Анализ текущих трендов в веб-разработке

Веб-разработка за последние годы претерпела значительные изменения, обусловленные как технологическим прогрессом, так и ростом пользовательских ожиданий. Ключевые тренды касаются архитектуры приложений, стека технологий, организации frontend-части и требований к безопасности, скорости и доступности.

Одним из главных сдвигов стало широкое распространение микросервисной архитектуры. Она пришла на смену монолитным приложениям, которые сложно масштабировать и сопровождать. В микросервисной модели каждый компонент системы изолирован (рисунок 1) и может разрабатываться, тестироваться и развертываться независимо от остальных. Это позволяет повысить отказоустойчивость и гибкость масштабирования, особенно в распределённых командах и больших продуктах [1].

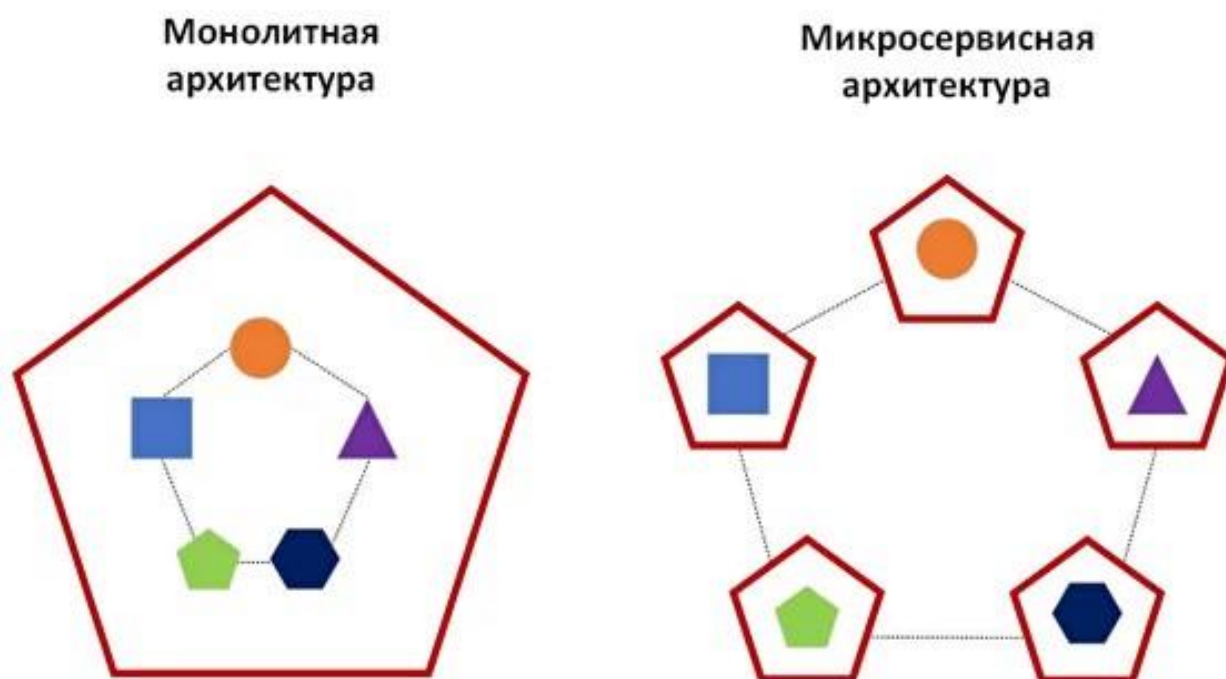


Рисунок 1 - Отличия монолитной и микросервисной архитектур

Тем не менее, в небольших или средних проектах, особенно с ограниченными ресурсами, по-прежнему часто применяется монолитная архитектура — как это реализовано и в рамках данного дипломного проекта. Это подтверждает, что выбор архитектуры должен основываться на анализе задач и условий, а не только на моде [1].

Параллельно происходил рост сложности frontend-разработки. Если раньше она сводилась к написанию HTML и минимального CSS/JS, то теперь это полноценная разработка с использованием фреймворков (React, Angular, Vue), типизации (TypeScript) и компонентного подхода. Например, React сегодня используется почти в 40% веб-проектов (рисунок 2) по данным Stack Overflow Developer Survey 2024 [2].



Рисунок 2 - Популярность веб-фреймворков [2]

Также усилилось внимание к производительности и доступности. Google, Mozilla и W3C продвигают стандарты WCAG (Web Content Accessibility Guidelines), требующие, чтобы сайты были доступны для пользователей с ограничениями по зрению, моторике или слуху. Быстрая загрузка страниц и адаптивность под мобильные устройства стали не просто

пожеланиями, а требованием рынка. Progressive Web Apps (PWA) и "mobile-first" дизайн — прямое следствие этой тенденции [3, 4].

Дополнительно стоит отметить тренд на API-first подход, при котором веб-приложение проектируется как набор взаимодействующих API. Это упрощает интеграцию с мобильными приложениями, внешними системами и расширяет возможности повторного использования компонентов [1].

Всё вышеописанное нашло отражение в популярности современных инструментов автоматизации и процессов разработки, таких как DevOps, CI/CD и инфраструктура как код — об этом подробнее в разделе 1.4 [3, 4].

1.2 Обзор современных технологий и инструментов

Современная веб-разработка основана на комбинации языков программирования, фреймворков, библиотек, систем управления базами данных и инструментов, обеспечивающих не только реализацию функционала, но и удобство разработки, масштабируемость и поддержку жизненного цикла проекта.

1.2.1 Языки программирования и фреймворки

На серверной стороне ключевыми технологиями остаются Python, JavaScript (в составе Node.js), PHP и Java. В контексте данного проекта был выбран Python, поскольку этот язык сочетает простоту синтаксиса, выразительность и широкую экосистему библиотек. Он отлично подходит для построения веб-приложений благодаря таким фреймворкам, как Django и FastAPI. Фреймворк Django был выбран в рамках реализации проекта как наиболее надёжный и функциональный, особенно на старте: он предоставляет встроенные механизмы ORM, аутентификацию, административную панель и поддержку шаблонов [1].

В книге Уильяма Винсента «Django for Professionals» подробно показано, как Django позволяет быстро переходить от прототипа к

продакшену, что делает его особенно ценной технологией для корпоративной разработки [5].

На клиентской стороне лидирующими технологиями являются JavaScript и его надстройка TypeScript, а также фреймворки React, Vue.js и Angular. Для большинства проектов React стал де-факто стандартом, однако в проекте использовалась более простая схема — шаблонный HTML с вкраплениями JavaScript и Bootstrap, что позволило быстрее адаптировать интерфейс под требования [3].

1.2.2 Базы данных

Веб-приложение требует надёжного хранения информации об устройствах, пользователях и связанных данных. Наиболее популярными системами управления базами данных (СУБД) в современных проектах являются:

- PostgreSQL объектно-реляционная СУБД с поддержкой расширенных типов данных и надёжной транзакционной моделью;
- MySQL классическая реляционная СУБД, широко используемая в веб-хостинге;
- SQLite встроенная, лёгкая и не требующая отдельного сервера;
- MongoDB нереляционная (NoSQL) база, основанная на документной модели данных.

Преимущества и недостатки наиболее популярных СУБД представлены в таблице 1:

Таблица 1 – Сравнение популярных СУБД

<i>СУБД</i>	<i>Поддержка транзакций</i>	<i>Расширенные типы данных</i>	<i>Масштабируемость</i>
PostgreSQL	Полная (ACID)	JSONB, массивы, кастомные типы	Высокая
MySQL	Частичная (зависит от движка)	Ограниченные	Высокая
SQLite	Полная (ACID)	Ограниченные	Низкая (один файл)
MongoDB	Нет	Гибкая JSON-like модель	Очень высокая

В рамках данного проекта применялась PostgreSQL как стабильная, масштабируемая и безопасная система управления базами данных. Она поддерживает транзакции, расширенные типы данных (включая JSONB), функции и триггеры [6].

Также Выбор PostgreSQL обусловлен в том числе и её тесной интеграцией с Django ORM - она поддерживается как основная реляционная СУБД в официальной документации и практиках фреймворка.

1.1.1 Вспомогательные инструменты

Для обеспечения качества и удобства разработки широко используются инструменты, которые помогают тестировать, разворачивать и сопровождать проект:

- Docker для контейнеризации и унификации среды разработки;

- Postman для тестирования API и автоматизации запросов;
- Git система контроля версий;
- GitHub/GitLab — хостинг репозитория и CI/CD пайплайны;
- Jira для управления задачами и отслеживания прогресса.

Наличие инструментов непрерывной интеграции (CI) и развёртывания (CD) стало обязательным для большинства современных проектов. Как подчёркивается в исследовании [7], их внедрение повышает надёжность релизов и снижает технический долг.

1.3 Принципы проектирования и разработки веб-приложений

Эффективное проектирование веб-приложений основывается на соблюдении архитектурных принципов, обеспечивающих модульность, масштабируемость, расширяемость и удобство сопровождения кода. При этом особую роль играют архитектурные паттерны, разделение ответственности и соблюдение практик чистого проектирования.

1.3.1 Модульность и масштабируемость

Модульность — это один из важнейших принципов, согласно которому приложение строится как набор изолированных блоков (модулей), каждый из которых решает строго определённую задачу. Это упрощает тестирование, поддержку и развитие проекта.

Согласно М. Фаулеру, модульная архитектура позволяет локализовать изменения, уменьшает связность компонентов и способствует параллельной разработке в команде [1]. Это особенно важно в быстро развивающихся продуктах, где требования и функциональность постоянно меняются.

Масштабируемость же обеспечивает возможность безболезненного роста системы — как по функциональности, так и по нагрузке. В современных

системах это достигается через чёткое разграничение слоёв (Frontend, Backend, API, база данных), отказ от жёсткой зависимости между компонентами и ориентацию на API-интерфейсы [3].

1.3.2 Архитектурные паттерны: MVC и MVT

Один из наиболее распространённых паттернов в веб-разработке — MVC (Model–View–Controller) (рисунок 3). Данный паттерн разделяет приложение на три взаимосвязанных слоя:

- Model отвечает за доступ к данным и бизнес-логику;
- View отвечает за отображение данных пользователю;
- Controller обрабатывает пользовательские действия, управляет логикой и выбирает представление.

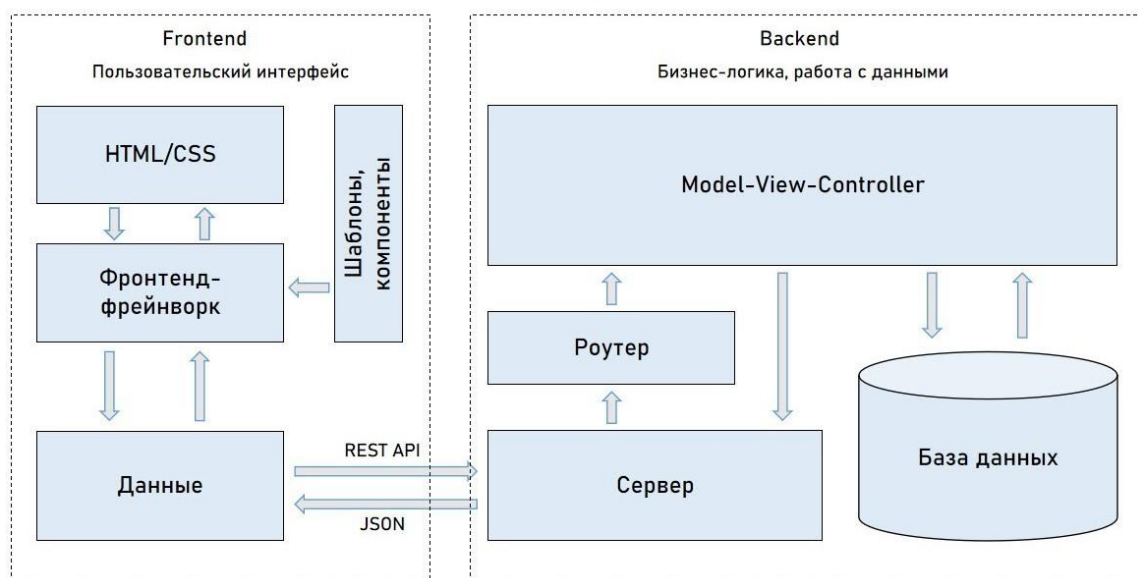


Рисунок 3 - Схема архитектуры веб-приложения с использованием MVC

Однако в фреймворке Django используется модификация этого паттерна - MVT (Model–View–Template). В данной архитектуре роль "контроллера"

выполняет сам фреймворк, а "View" — это логика, которая возвращает данные, а не отображение [5].

Таким образом, различие между MVC и MVT не фундаментальное, а организационное. Django развивает идею слабосвязанных компонентов, что делает систему гибкой и легко расширяемой [5].

1.3.3 Принципы чистой архитектуры и паттерны

Ключевые принципы проектирования, на которые ориентируется современная веб-разработка:

- DRY (Don't Repeat Yourself) - избегание дублирования кода;
- KISS (Keep It Simple, Stupid) - предпочтение простых и ясных решений;
- SOLID - пять принципов объектно-ориентированного проектирования, направленных на снижение связанности и повышение расширяемости.

Фаулер подчёркивает, что в архитектуре корпоративных приложений важно не просто реализовать паттерны, а применять их осознанно [1]: например, не всегда микросервисная модель оправдана — для простых и внутренних сервисов может подойти и монолитная архитектура [3].

Примером внедрения этих подходов может служить реализация бизнес-логики отдельно от представлений (views) в Django — через сервисные функции и кастомные менеджеры моделей, что обеспечивает чистоту кода и его переиспользуемость [5].

1.4 Методологии разработки: Agile, DevOps, CI/CD

Современная веб-разработка требует не только использования технически эффективных фреймворков и языков, но и грамотной организации процессов. Методологии Agile, DevOps и CI/CD стали ключевыми подходами, обеспечивающими гибкость, скорость и качество разработки. Их внедрение

позволяет не просто создавать программные продукты, а делать это системно, быстро и с минимальным количеством ошибок.

1.4.1 Agile, Kanban, Waterfall

Методология Agile была разработана как альтернатива жёстким каскадным подходам, в которых весь проект планируется заранее и изменения в требованиях после старта воспринимаются как "неудача". Agile-подход делает упор на:

- итеративную разработку небольшими шагами (спринтами)[12];
- обратную связь от заказчика на каждом этапе[12];
- приоритет работающего кода над документацией [4].

Agile помогает быстрее реагировать на изменяющиеся требования рынка, а также лучше понимать реальные потребности пользователей. Особенно это актуально в веб-разработке, где продукт часто должен быть показан в рабочем виде уже через несколько недель.

Waterfall (каскадная модель) - это линейный подход, при котором каждый этап разработки (требования, проектирование, реализация, тестирование, поддержка) выполняется последовательно. Его недостатки очевидны в условиях частых изменений: возврат на предыдущий этап затруднён и дорогостоящ [13].

Kanban - визуальная система управления задачами, которая подчёркивает непрерывный поток задач без жёсткой спринтовой структуры. В отличие от Scrum, Kanban не требует фиксированных временных рамок и подходит для проектов с постоянным потоком изменений [13].

В таблице 2 представлены основные подходы планирования, их преимущества и недостатки:

Таблица 2 – Основные методологии в планировании

<i>Методология</i>	<i>Структура</i>	<i>Гибкость</i>	<i>Обратная связь</i>	<i>Подходит для</i>
Waterfall	Линейная	Низкая	Поздняя	Проекты с жёсткими требованиями
Agile	Итеративная	Высокая	Частая	Большинство веб-проектов
Kanban	Поточная	Средняя	Непрерывная	Поддержка, DevOps

Исходя из требований проекта была методология Agile:

- работа разбивалась на небольшие этапы (реализация модели, интерфейса, API);
- после каждой итерации проект запускался и проверялся;
- новые функции добавлялись по мере появления конкретных задач.

Это позволило не застрять на планировании и сосредоточиться на ценности, которую приносит каждая следующая часть системы.

Одно из важных достоинств Agile — это способность быстро адаптироваться к изменениям.

1.4.2 DevOps автоматизация

DevOps - это автоматизация взаимодействия между разработкой (Dev) и эксплуатацией (Ops). Раньше эти две части существовали отдельно: разработчики писали код, а администраторы обеспечивали его запуск. В DevOps подходе:

- команды работают как единое целое;

- применяются практики автоматизации тестов, сборки и деплоя;
- используется мониторинг и логирование с самого начала [4].

DevOps помогает быстрее выявлять ошибки и снижает "цену релиза". Например, если раньше развертывание происходило вручную, с рисками конфигурационных конфликтов, то теперь через один клик или push в Git, с полностью проверенным пайплайном.

В рамках проекта DevOps-элементы проявились в виде:

- автоматизации установки зависимостей и миграций через make и скрипты;
- настройки окружения в docker-compose;
- централизованного логирования в лог-файлы через uWSGI и Nginx;
- использования Git и pull-request модели с ревью изменений.

1.4.3 CI/CD

CI/CD (Continuous Integration / Continuous Delivery или Deployment) - это практики, направленные на то, чтобы изменения в коде проходили путь от коммита до продакшена автоматически и безопасно.

CI означает:

- каждый коммит отправляется в систему сборки;
- запускаются тесты;
- на выходе можно быть уверенным, что код не “сломал” систему [4].

CD дополняет это:

- код, прошедший тесты, автоматически развёртывается;
- либо на тестовом сервере (Continuous Delivery), либо прямо в продакшен (Continuous Deployment).

Применение CI/CD в дипломном проекте:

- использовались локальные скрипты тестирования и миграции;
- развёртывание происходило через автоматизированные скрипты (deploy.sh);
- база данных мигрировалась через `python manage.py migrate`, окружение через `.env`.

Эти шаги имитируют настоящие CI/CD-системы, применяемые в корпоративных командах с GitLab CI, GitHub Actions, Jenkins и др.

GitLab CI — интегрирован прямо в GitLab и предлагает полный цикл: планирование задач, хранилище кода, CI/CD, мониторинг. Файл `.gitlab-ci.yml` определяет пайплайн. Имеет хорошую визуализацию, встроенные секреты, поддерживает Docker.

GitHub Actions — аналогичный подход, но внутри GitHub. Рабочие процессы описываются в `.yml` в `.github/workflows`. Отличается простой интеграцией с Actions-маркетплейсом и гибкой настройкой событий (push, PR, tag и т.д.).

Jenkins — наиболее гибкий и зрелый инструмент. Не зависит от платформы хранения кода. Предлагает визуальные пайплайны, огромное количество плагинов, кастомные скрипты и возможность встраивания в любую инфраструктуру. Требуется отдельный хост.

Для небольших проектов GitLab CI или GitHub Actions предпочтительнее, так как они позволяют реализовать CI/CD без поднятия дополнительной инфраструктуры, в то время как Jenkins больше подходит для крупных организаций [15].

1.5 Выводы по главе

В ходе анализа теоретических аспектов веб-разработки были рассмотрены ключевые технологии, архитектурные подходы и методологии, определяющие современную практику создания и сопровождения веб-приложений.

Было установлено, что одним из главных трендов является отход от жёстко связанной монолитной архитектуры и усиление роли микросервисного подхода [3]. Однако выбор архитектуры всегда должен учитывать ресурсы команды и масштаб проекта — в некоторых случаях монолит остаётся оправданным решением [1].

Анализ Frontend и Backend инструментов показал, что сочетание Python/Django и PostgreSQL обеспечивает баланс между простотой разработки и надёжностью системы [6]. На стороне клиента использование шаблонной системы Django совместно с Bootstrap позволяет эффективно создавать адаптивные интерфейсы.

Особое внимание было уделено принципам проектирования: соблюдение модульности, разделение представления и логики (MVT), использование паттернов проектирования и принципов DRY/KISS значительно упрощают масштабирование и сопровождение проекта [5].

Наконец, рассмотренные методологии Agile, DevOps и CI/CD доказали свою эффективность в современных условиях разработки. Гибкие подходы к управлению проектами, автоматизация процессов сборки и поставки, а также тесное взаимодействие внутри команды способствуют сокращению времени вывода продукта на рынок и повышению его качества [10].

Итоги главы стали основой для выбора архитектуры, стека технологий и процесса разработки дипломного проекта, о чём подробно рассказывается в последующих разделах.

2. Разработка веб-приложения

2.1 Архитектура проекта

2.1.1 Выбор архитектурной модели

Изначальной задачей на этапе проектирования было создание веб-сервиса для централизованного учёта, просмотра, редактирования, удаления и добавления устройств, разрабатываемых внутри компании. При проектировании необходимо было учитывать следующие условия:

- ограниченные сроки разработки;
- небольшая команда разработчиков;
- выделенный сервер внутри корпоративной инфраструктуры;
- требование к полной конфиденциальности данных и запрету на их хранение на внешних ресурсах;
- необходимость упрощённой поддержки и сопровождения проекта в будущем силами внутренней команды.

На основе этих факторов был проведён анализ трёх основных архитектурных подходов: монолитной архитектуры, микросервисной архитектуры и бессерверной архитектуры (serverless).

При анализе монолитной архитектуры были выявлены следующие плюсы:

- быстрая разработка;
- простота развёртывания;
- меньше накладных расходов на администрирование;
- упрощённая отладка;
- оптимально для небольших команд.

Минусы монолитной архитектуры:

- при росте проекта могут возникать проблемы с масштабируемостью [1];

- сложнее вносить изменения в большие приложения без риска повлиять на другую часть системы;
- меньшая гибкость масштабирования отдельных компонентов [3].

Причина, по которой монолитная архитектура подходит для проекта: учитывая ограниченный масштаб задач (работа с несколькими ключевыми сущностями) и небольшую команду, плюсы монолита идеально соответствуют требованиям проекта. Простота развертывания и минимальные затраты на инфраструктуру были критичными факторами для старта [17].

Также для исследования была проанализирована Микросервисная архитектура. Плюсы микросервисной архитектуры.

- Высокая масштабируемость.
- Высокая отказоустойчивость.
- Гибкость в развитии.
- Улучшенное управление сложностью при очень больших системах [18].

Минусы:

- сложность проектирования;
- рост числа сервисов требует серьёзной настройки DevOps-процессов, автоматизации развертывания, мониторинга и логирования [11];
- затраты на инфраструктуру: увеличение количества сервисов ведёт к увеличению стоимости поддержки;
- сложность тестирования: требуется интеграционное тестирование всех сервисов и их связей.

Таким образом, для ограниченного по ресурсам проекта реестра разработок микросервисная архитектура оказалась бы избыточной.

Также была проанализирована Бессерверная архитектура (Serverless).
Преимущества.

- Быстрая масштабируемость без необходимости управлять серверами.
- Оплата только за фактическое использование ресурсов.
- Минимальные затраты на обслуживание инфраструктуры.
- Высокая гибкость при построении событийных систем [19].

Недостатки.

- Сильная зависимость от внешнего провайдера (AWS, Azure, GCP) [19].
- Сложности с обеспечением безопасности и конфиденциальности данных [19].
- Ограничения на время выполнения функций, объём памяти и сеть.
- Потенциально высокая стоимость при высоком объёме трафика.
- Требуется глубокого понимания событийной архитектуры и асинхронной логики [19].

Serverless-модель не подходит для проекта по причине, что бессерверная архитектура требовала бы хранения данных у стороннего облачного провайдера, что нарушает требование конфиденциальности компании [19]. Кроме того, текущая команда разработки имеет больше опыта с традиционными архитектурами, что делает монолит более надёжным решением с точки зрения стабильности и предсказуемости проекта.

После выбора архитектурной модели следующим этапом проектирования стало определение паттерна для организации структуры кода приложения.

2.1.2 Выбор паттерна проектирования

Паттерн проектирования в данном контексте — это готовое решение типичных задач построения архитектуры программы, предназначенное для упрощения процесса разработки и повышения сопровождаемости системы. Паттерны можно понимать как шаблоны проектирования, предлагающие проверенные подходы к решению часто встречающихся проблем [21]. Наиболее известные паттерны и их различия представлены в таблице 3 [20]:

Таблица 3 – Паттерны, их компоненты и роль

<i>Паттерн</i>	<i>Компоненты</i>	<i>Роль View</i>	<i>Роль промежуточного слоя</i>
MVC	Model, View, Controller	Отображает данные	Controller обрабатывает логику и события, управляет моделью и обновляет View
MVT	Model, View, Template	Template HTML шаблон	View обрабатывает запросы, подготавливает данные для шаблона
MVVM	Model, View, ViewModel	Отображает UI, связан с ViewModel через data binding	ViewModel преобразует данные модели для UI, поддерживает двустороннюю привязку
MVP	Model, View, Presenter	Пассивна, вызывает методы Presenter	Presenter управляет логикой, напрямую обновляет View и Model

Каждый из представленных выше паттерн имеют свою область применения: MVC используется в Ruby on Rails, Laravel, ASP.NET. MVT в python, в частности, Django. MVVM в WPF, Angular, SwiftUI. А MVP - Android и десктоп-приложения.

Выбор паттерна MVT был обусловлен выбранным фреймворком. MVT - это адаптированный паттерн MVC. В MVT сохранена структура MVC, но также присутствуют отличия. Основное различие между MVC и MVT заключается в том, что в классическом MVC Controller самостоятельно выбирает, какое представление показать пользователю, тогда как в Django роль контроллера выполняет сам View, который напрямую передаёт данные в Template для построения HTML-страницы [19]. Таким образом, в Django процесс более автоматизирован и упрощён.

Выбранный паттерн MVT позволил логически разделить уровни данных, обработки запросов и отображения интерфейса, что упростило разработку, тестирование и дальнейшую поддержку проекта. В рамках проекта структура кода построена по паттерну MVT (Model-View-Template) (рисунок 4), что обеспечивает логическое разделение данных, обработки запросов и отображения информации пользователю [22].

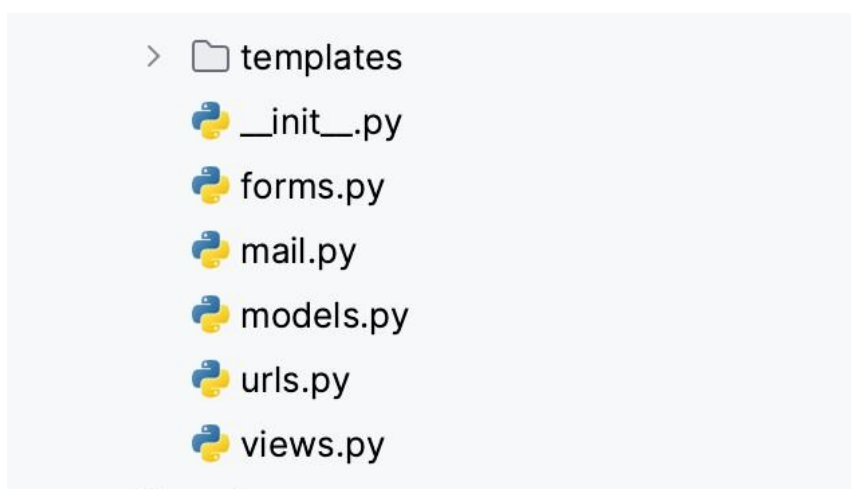


Рисунок 4 - Использование MVT в проекте

2.1.3. Реализация модели, представлений и шаблонов

Рассмотрим более подробно структуру MVT: Модели в проекте реализованы через стандартные средства Django ORM - система, которая позволяет разработчику взаимодействовать с базой данных с помощью объектов Python вместо прямого написания SQL-запросов [19]. Модель используют класс - инструмент представления сущности с ее атрибутами в Python, а поля в таблице базы данных атрибуты этих классов. Данный инструмент позволяет отображать данные между БД и моделью приложения. Таким образом, данная практика позволяет разработчику описывать основные сущности бизнес-логики приложения (рисунок 5) и в дальнейшем использовать их:

```
30 class Device(AbstractContainerOPU, DeviceMixin):  # Suganov Boris +4
31     """
32     Устройство.
33     В одном проекте разработки может быть несколько устройств, которые отличаются версиями аппаратных компонент.
34
35     * project - это проект разработки (Jira), к которому относится устройство
36     * project_rnd - внутренний идентификатор проекта разработки (RnD)
37     * subsection - подкласс устройства в соответствии с принятым классификатором Т8
38     * code - уникальный длинный код устройства
39     * name - наименование устройства
40     * code_usdd - Обозначение ЕСКД (Единая система конструкторской документации)
41     * serial_template - Шаблон серийного номера
42     * status_dev - Статус разработки
43     * status_pro - Статус производства
44     """
45     name = models.ForeignKey(
46         DeviceName,
47         verbose_name='текстовая часть наименования',
48         on_delete=models.PROTECT,
49         null=False
50     )
51     price_item = models.ForeignKey(
52         'price.PriceItem',
53         verbose_name='коммерческая позиция',
54         related_name='device_items',
55         null=True,
56         blank=True,
57         on_delete=models.SET_NULL
58     )
59     is_commercial = models.BooleanField(
60         verbose_name='в прайсе/не в прайсе',
61         default=False,
62         help_text='разработка планируется к продаже'
63     )
64     project = models.ForeignKey(
65         Project,
66         verbose_name='проект разработки (Jira)',
67         null=True,
68         blank=True,
69         on_delete=models.SET_NULL
```

Рисунок 5 - пример модели в приложении

Каждая модель описана в отдельном классе, где определены поля с указанием их типов, ограничений и связей между моделями, например, связь `ForeignKey` между устройствами и категориями (рисунок 5). Модели взаимодействуют с базой данных через встроенную ORM Django. Все операции с данными — создание, чтение, обновление и удаление записей — осуществляются через методы моделей и менеджеры объектов: `objects.create()`, `objects.filter()`, `objects.update()` и другие. Данные инструменты позволяют избегать ручного написания SQL-запросов.

Представления отвечают за обработку входящих HTTP-запросов от клиента. В проекте для реализации представлений преимущественно используются классовые представления (CBV - Class-Based Views), такие как `ListView`, `DetailView`, `CreateView`, `UpdateView`, `DeleteView` [19] (рисунок 6).

```
class DeviceTypeCreate(PermissionRequiredMixin, CreateView): new *
    model = DeviceType
    form_class = DeviceTypeForm
    permission_required = 'dev.add_dev'
    success_url = reverse_lazy("dev:list")

class DeviceTypeUpdate(PermissionRequiredMixin, UpdateView): new *
    model = DeviceType
    form_class = DeviceTypeForm
    permission_required = 'dev.change_dev'
    success_url = reverse_lazy("dev:list")

class DeviceTypeDelete(PermissionRequiredMixin, DeleteView):  Sukanov Boris *
    model = DeviceType
    form_class = CustomDeleteForm
    success_url = reverse_lazy("dev:list")
    permission_required = 'dev.delete_dev'
```

Рисунок 6 – Использование Class-Based View в проекте

Каждое представление выполняет строго ограниченную задачу:

- отображение списка устройств;

- просмотр подробной информации об устройстве;
- создание новой записи устройства;
- редактирование существующих записей;
- удаление устройств;

Благодаря использованию MVT удаётся минимизировать дублирование кода и ускорить разработку типовых операций (CRUD). Логика представлений включает:

- обработку параметров запроса,
- выбор нужных данных из базы через модели,
- передачу данных в шаблон (Template) для дальнейшего отображения.

При необходимости, в представлениях реализуется дополнительная бизнес-логика, например, фильтрация устройств по категориям или статусу. Отображение данных пользователю организовано через систему шаблонов Django. Шаблоны строят HTML-страницы на основе переданных из представлений данных. Они позволяют:

- вставлять данные внутрь HTML через переменные шаблонов (рисунок 7);
- использовать управляющие конструкции If , for (рисунок 7);
- подключать части шаблонов через include (рисунок 7) для повторного использования компонентов (например, форм и таблиц).

```

<tbody>
{% for version in versions %}
  <tr>
    <td>
      {% if version %}
      <a href="https://{{ version.get_absolute_url }}">{{ version.device.code }}</a>
      {% endif %}
    </td>
  </tr>
{% endfor %}
</tbody>
</table>
{% endblock %}

{% block footer %}
  {% include "mail/templates/footer.html" %}
{% endblock %}

```

Рисунок 7 – Использование Django Template

Благодаря системе шаблонов обеспечивается чёткое разделение логики обработки данных (View) и их отображения (Template), что повышает читаемость и сопровождаемость проекта [20].

2.1.4 Структура и компоненты серверной части

При проектировании веб-приложения была выбрана архитектура, основанная на разделении серверной части, клиентской части, базы данных и вспомогательных сервисов. Такое построение системы обеспечило упорядоченность кода, удобство разработки, надёжность взаимодействия компонентов и лёгкость масштабирования проекта в будущем. Серверная же часть проекта реализована с использованием фреймворка Django. Она отвечает за обработку HTTP-запросов, реализацию бизнес-логики приложения и взаимодействие с базой данных через встроенную ORM [27]. К основным задачам серверной части относятся:

- обработка входящих запросов от пользователей;
- реализация бизнес-правил, таких как проверка корректности данных и регистрация событий;

- управление данными через модели Django;
- передача данных клиентской части для последующего отображения.

Структура backend-приложения включает отдельные модули для управления устройствами, категориями устройств и историей изменений. Для обработки запросов применяются классовые представления (Class-Based Views), что позволило сократить объём повторяющегося кода и обеспечить его лучшую читаемость. Применение паттерна MVT позволило строго разграничить уровни работы с данными, логики обработки запросов и визуализации интерфейса. Клиентская часть построена на системе шаблонов Django с использованием HTML, CSS, JavaScript и библиотеки Bootstrap. Её основными задачами являются:

- формирование адаптивного и удобного интерфейса;
- отображение данных, полученных от серверной части;
- отправка данных на сервер через формы или динамические запросы;
- обеспечение корректной работы интерфейса на различных устройствах.

Динамическая генерация HTML-страниц на основе переданных данных - функционал шаблонов Django. А использование технологий Bootstrap ускоряет верстку страниц значительно, что позволяет сосредоточиться на разработке остального функционала приложения. СУБД PostgreSQL используется в проекте для хранения данных. Она позволяет:

- хранить информацию сущностей, описанных в моделях (устройства, категории и события);
- использовать транзакции, что позволяет обеспечить целостность операций [24];
- использовать внешние ключи, что дает возможность поддержки логических связей между таблицами;
- оптимизировать с помощью индексов доступ к данным.

Для осуществления внесения изменения используется инструмент миграций, встроенный в Django. Данный инструмент позволяет вносить изменения в структуру БД с функционалом истории изменений и встроенной защитой от потери данных [28].

В качестве инструмента, который позволяет развернуть приложения на сервере, используется связка uWSGI и Nginx. uWSGI – веб-сервер, который использует протокол Web Server Gateway Interface (WSGI). Разработан для использования и запуска приложений на Python [25]. Nginx – HTTP сервер, который поддерживает кеширование и балансировку нагрузки [26]. Данная связка инструментов позволяет обеспечить веб-сервис надёжностью, отказоустойчивостью и производительностью.

Для взаимодействия компонентов в веб-приложении были разграничены зоны ответственности между клиентской частью и серверной, БД и дополнительными библиотеками. Данного результата позволило добиться использование стандартных протоколов передачи данных HTTP/HTTPS. HTTP/HTTPS позволяют обеспечить доставку и ответы [33]. Клиентская часть отправляет запросы на сервер с помощью следующих инструментов:

- отправление данных через веб-формы;
- отправление Ajax-запросов для динамического обновления страниц;
- запросов к API;

При получении запроса серверная часть, реализованная на Django, выполняет следующие действия:

- принимает запрос и определяет его тип (GET, POST, PUT, DELETE);
- вызывает соответствующее представление (View) для обработки запроса;
- при необходимости взаимодействует с базой данных через модели ORM для выборки, создания, обновления или удаления данных) [31];

- формирует ответ: либо HTML-страницу (через рендеринг шаблонов), либо данные в формате JSON для асинхронных запросов) [31];
- отправляет сформированный ответ обратно клиенту.

База данных PostgreSQL взаимодействует только с серверной частью приложения. Все запросы к базе данных проходят через Django ORM, что обеспечивает:

- автоматическую генерацию безопасных SQL-запросов [24];
- поддержку транзакций при выполнении нескольких связанных операций [24];
- проверку целостности данных через использование внешних ключей и ограничений.

Структура передачи данных между сервером и базой данных строго типизирована, что минимизирует вероятность ошибок при работе с данными. Клиентская часть проекта, построенная на системе шаблонов Django с использованием HTML, CSS, JavaScript и Bootstrap, отвечает за визуальное отображение данных. Шаблоны получают данные из представлений в виде контекста и формируют динамические страницы. В случае использования AJAX-запросов данные передаются в формате JSON и обрабатываются средствами JavaScript для обновления отдельных частей интерфейса без полной перезагрузки страницы [33]. Для оптимизации взаимодействия между компонентами реализованы следующие механизмы:

- использование кэширования на уровне шаблонов для часто запрашиваемых страниц, что снижает нагрузку на сервер;
- предварительная выборка связанных данных через методы `Select_related` и `Prefetch_related` в Django ORM для уменьшения количества SQL-запросов [31];

- минимизация передачи лишних данных за счёт формирования оптимальных структур ответов сервера.
- особое внимание уделено обеспечению безопасности взаимодействия между компонентами [32].

На уровне сервера реализованы следующие меры защиты:

- обязательное использование протокола HTTPS для всех соединений, что гарантирует шифрование передаваемой информации [32];
- защита от атак типа CSRF (Cross-Site Request Forgery) путём использования токенов CSRF в формах и проверок на стороне сервера [32];
- защита от XSS-атак (Cross-Site Scripting) через автоматическое экранирование данных в шаблонах Django [32];
- проверка прав доступа пользователя к каждому действию, в том числе в API [32];
- валидация всех данных, поступающих от клиента, до их обработки или сохранения в базу данных.

2.1.5. Вывод

В ходе анализа технологий на этапе проектирования было выявлено, что для данного проекта монолитная архитектура является оптимальным решением.

Выбор в пользу монолита был обусловлен необходимостью быстрого развертывания решения, ограниченными ресурсами команды, требованиями к конфиденциальности данных и спецификой дальнейшего сопровождения системы. Рассмотренные альтернативы в виде микросервисной и

бессерверной архитектур были признаны избыточными или несоответствующими условиям проекта.

Структура кода приложения организована на основе паттерна MVT, что позволило чётко разделить логику обработки данных, управление бизнес-процессами и визуализацию информации для пользователя.

Таким образом, выбранная архитектура и структура проекта будет обеспечивать необходимую стабильность, надёжность, простоту расширения функциональности и удобство сопровождения, что соответствует целям и задачам, поставленным на начальном этапе разработки.

2.2 Выбор и описание инструментов разработки

2.2.1 Выбор языка программирования

Важнейшим решением на этапе проектирования архитектуры проекта стало определение основного языка разработки. В качестве такового был выбран Python — язык, обладающий высокой выразительностью, зрелой экосистемой, а также отлично подходящий под требования к быстрому и стабильному запуску веб-приложения.

Python — язык с открытым исходным кодом, созданный с акцентом на читаемость кода и снижение когнитивной нагрузки на разработчика. Благодаря синтаксису, близкому к естественному языку, Python позволяет быстрее писать и поддерживать код, что особенно критично при ограниченных ресурсах команды. Кроме того, Python поддерживает множество парадигм (включая ООП, функциональное и процедурное программирование), что повышает гибкость в решении архитектурных задач [31].

Как подчёркивает М. Фаулер, выбор языка программирования должен соответствовать не только техническим требованиям, но и организационным условиям: размер команды, доступные библиотеки, ожидаемый срок разработки [1]. Именно по этим критериям Python оказался наиболее

сбалансированным решением для реализации внутреннего корпоративного веб-сервиса.

Для проекта особенно важными оказались следующие преимущества:

- наличие веб-фреймворков: Django, Flask и FastAPI, которые позволяют быстро запускать и масштабировать проекты с минимальными затратами на инфраструктуру;
- экосистема содержит десятки тысяч готовых библиотек (например, Django REST Framework, Celery, SQLAlchemy) решают задачи безопасности, сериализации, фоновых задач и многого другого;
- низкий порог входа Python остаётся одним из самых популярных языков для обучения программированию, а значит, проще найти и подключать новых разработчиков к проекту [2];
- интеграция с аналитикой — в случае необходимости внедрения модулей аналитики, ML или мониторинга, Python остаётся лидером благодаря библиотекам NumPy, Pandas, Scikit-learn и другим.

Несмотря на популярность Node.js в современном вебе, в проекте он не был выбран по следующим причинам:

- Node.js требует иной парадигмы проектирования, асинхронной по умолчанию, что увеличивает сложность разработки;
- менее выраженная модель ORM по сравнению с Django ORM;
- необходимость дополнительной настройки для обеспечения безопасности и миграций.

Кроме того, для задач внутреннего корпоративного интерфейса предпочтение было отдано серверной генерации HTML, а не SPA-архитектуре, что усилило позиции Django как основного фреймворка [19].

Java отличается высокой масштабируемостью и строгой типизацией, однако:

- разработка на Java требует значительно большего времени на конфигурацию;
- низкая скорость прототипирования;
- избыточность для задач средней сложности [17].

PHP, несмотря на широкую распространённость, показал себя как менее структурированный инструмент в сравнении с Python и Django. Проблема экосистемы PHP в наличие устаревших решений снижают его приоритет в современных корпоративных разработках.

Go — это компилируемый язык от компании Google, набирающий популярность за счёт своей производительности и встроенной поддержки многопоточности. Однако, несмотря на сильные стороны, он не был выбран по ряду причин:

- отсутствие зрелых фреймворков, сравнимых по полноте и автоматизации с Django;
- необходимость ручной настройки ORM, маршрутизации, миграций и шаблонов;
- выше требования к инфраструктуре и DevOps-процессам;
- ниже производительность команды на старте, особенно при отсутствии опыта;
- невозможность быстрой генерации форм, админки, валидации и HTML-шаблонов «из коробки».

Таким образом, несмотря на высокую производительность и потенциал для масштабируемых API, Go не был выбран из-за увеличения издержек и отсутствия интеграции с текущими компонентами проекта.

Дополнительным фактором стало то, что Python уже активно использовался в инфраструктуре компании: для внутренних скриптов, API и анализа данных. Команда разработки имела опыт работы с Django, что позволило избежать затрат на переобучение и ускорило первые этапы

разработки. Как отмечают Басс и Казман, выбор технологии, поддерживаемой командой, является критически важным фактором устойчивости проекта [17].

2.2.2 Выбор фреймворка

Для реализации серверной части веб-приложения был выбран фреймворк Django. Основной причиной выбора Django стала его способность значительно ускорить процесс разработки за счёт наличия большого количества встроенных решений "из коробки" [30]. Фреймворк предоставляет следующие инструменты:

- ORM;
- миграции базы данных;
- аутентификация пользователя;
- панель администратора;
- формы.

Данные инструменты позволяют повысить эффективность разработки путем большего фокуса на бизнес-логики, а не на изобретении велосипеда.

Также среди плюсов Django можно выделить следующие пункты.

- Высокую скорость разработки.
- Встроенные средства безопасности, защиты от атак CSRF и SQL-инъекций [31].
- Гибкую кастомизацию административной панели[32].
- Качественную документацию и поддержку сообщества [19].

Также были рассмотрены и проанализированы наиболее популярный фреймворков на Python в таблице 4:

Таблица 4 – Наиболее популярные веб-фреймворки на Python

<i>Критерий</i>	<i>Django</i>	<i>Flask</i>	<i>FastAPI</i>
Подход	«Из коробки»	Минималистичны й	Декларативны й с упором на типизацию
Асинхронность	Частично поддерживается	Не нативно	Полная поддержка
Производительность	Умеренная	Высока	Очень высокая
Управление пользователями	Встроено (аутентификация, авторизация, админка)	Нужно реализовывать самостоятельно	Нужно реализовывать самостоятельно
Время на реализацию проекта	Оптимально: много готовых решений	Увеличивается из- за ручной сборки архитектуры	Увеличивается из-за необходимости изучения специфики

Исходя из требований к проекту и поставленным задачам наиболее предпочтительным оказался Django.

2.2.3 Выбор СУБД

Для управления базами данных была выбрана PostgreSQL из-за следующих преимуществ:

- Наличие JSONB в типах данных [35];
- Поддержка сложных транзакций [24];
- Высокая производительность [35];
- Поддержка Django [6].

2.2.4 Выбор технологий клиентской части

Для клиентской части был выбран JavaScript для разработки функционала, который не реализуем backend частью приложения. JavaScript необходим для создания отзывчивого и удобного интерфейса [33].

Выбор данных технологий был обусловлен необходимостью добиться высокого уровня надежности и скорости разработки, а также безопасности и стабильности.

2.2.5 Выбор дополнительных библиотек

Также помимо фреймворка Django в проекте были задействованы дополнительные библиотеки и инструменты. На языке Python довольно много готовых решений для каких-либо задач, которые расширяют стандартный функционал Python и Django. В таблице 5 перечислены основные инструменты и библиотеки, и где они задействованы в проекте:

Таблица 5 – Дополнительные библиотеки, выбранные для проекта

<i>Инструмент</i>	<i>Функционал</i>	<i>Использование в проекте</i>
Django REST Framework	Создание REST API [30]	Используется для CRUD операций с моделями устройств через API.
django-filter	Фильтрация запросов по параметрам моделей [36]	Позволяет фильтровать устройства по типу, статусу, версии ПО. Используется в приложении device, project
django-import-export	Импорт/экспорт данных [37]	Реализует экспорт устройств из админки в XLSX, CSV, JSON и загрузку новых.

django-crispy-forms	Рендеринг адаптивных форм с Bootstrap 5 [38]	Обеспечивает отображение форм редактирования с кнопками сохранения практически в каждой форме.
django-clone	Клонирование объектов моделей [39]	Реализован функционал копирования версий для ускорения работы пользователя.
django-ordered-model	Сортировка объектов через drag-and-drop в админке [40]	Используется в модели категорий устройств для управления порядком
django-debug-toolbar	Инструмент отладки (SQL, middleware, запросы) [41]	Активен в DEBUG-режиме, позволяет анализировать запросы и SQL [41]
Jira API (jira)	Интеграция с системой задач Jira по API [42]	Создаются задачи на основе событий в модели или интерфейсе, также синхронизируется информация из Jira по проектам.
uWSGI	Серверное развёртывание Django-приложения [43]	Работает в паре с nginx для запуска приложения на продакшене [43]

Использование дополнительных библиотек значительно ускоряет разработку по типовым задачам. Также использование дополнительных инструментов позволяет упростить тестирование, а также развёртывание проекта на сервере. Каждый инструмент перед внедрением был проанализирован на предмет необходимости, и если он позволял решить конкретную задачу, оптимизируя работу команды и повышая общее качество системы, то инструмент добавляли.

2.2.6 Выводы

Выбор проверенного и сбалансированного стека технологий оказал прямое положительное влияние на процесс разработки веб-приложения.

Использование Django позволило существенно ускорить реализацию серверной части. Благодаря встроенным инструментам — таким как система маршрутизации URL, ORM для работы с базой данных, средства аутентификации и административная панель — время на разработку базовой инфраструктуры было минимизировано. Большая часть задач по обеспечению безопасности и управлению данными была решена с помощью стандартных механизмов фреймворка, что позволило сконцентрироваться на реализации специфической бизнес-логики проекта.

Остальные проанализированные и выбранные технологии обеспечили скорость разработки, стабильность приложения и база данных, высокую степень безопасности данных. На практике выбранные технологии продемонстрировали свою надёжность и подтвердили правильность проведённого анализа, что позволило разработать веб-приложение, отвечающие поставленным бизнес-задачам.

Таким образом, комбинация Django, PostgreSQL, JavaScript, Bootstrap и вспомогательных библиотек стала оптимальным выбором для данного проекта и может быть рекомендована для аналогичных задач в будущем.

2.3 Проектирование базы данных

2.3.1 Определение требований к данным

На этапе проектирования веб-приложения одной из ключевых задач стало определение требований к данным, подлежащим хранению и обработке в системе. Основная бизнес-цель проекта заключалась в учёте, мониторинге и сопровождении устройств, разрабатываемых в рамках различных

исследовательских и коммерческих проектов компании. Для достижения этой цели потребовалось разработать структуру данных, которая бы полноценно отражала все необходимые аспекты работы с устройствами.

Анализ требований показал, что в системе необходимо было хранить следующую основную информацию:

- сведения об устройствах, разрабатываемых компанией;
- наименования и классификацию устройств;
- информацию о проектах разработки, к которым привязаны устройства;
- данные о программных продуктах, связанных с устройствами;
- информацию о коммерческих позициях устройств и их статусах на этапах разработки и производства;
- внутренние идентификаторы исследовательских проектов;
- связи устройств с производственными линиями;
- данные о лицензионных договорах по устройствам;
- информацию по компонентам и версиям разработок компании;
- информацию по задачам, которые ведутся в jira;
- данные по линейкам оборудования T8;
- информацию по методам резервирования компонентов;
- данные по система управления;
- сведения о покупателях устройств.

На основе анализа предметной области были выделены ключевые сущности:

- Device - основная сущность, описывающая устройство;
- EquipmentName - наименование устройства;
- Project - проект разработки (интеграция с системой Jira);
- ResearchProject - внутренний исследовательский проект (RnD);
- SoftwareProgram - программа разработки, связанная с устройством;

- ProductionLine - производственная линейка, к которой относится устройство;
- DevStatus и ProStatus - статусы разработки и производства устройства соответственно;
- LicenseContract - лицензионные договоры;
- Component - компоненты устройств;
- ComponentVersion - версии компонентов устройств;
- Currency - Используемые валюты;
- Price - Вид цены;
- Customer - покупатель устройства.

Каждая сущность имела набор атрибутов, необходимых для полноценного описания бизнес-объекта. Например, модель устройства (Device) включала такие атрибуты, как:

- текстовая часть наименования устройства;
- код устройства по классификатору;
- принадлежность к коммерческой позиции (прайс-листу);
- принадлежность к проекту разработки Jira и RND;
- статус устройства на этапе разработки и производства;
- коммерческая позиция;
- принадлежность к программа разработки;
- класс устройства;
- линейка устройства;
- планируемая дата доступности к заказу;
- крайняя дата приема заказа;
- дата окончания поддержки;
- эскизное обозначение (ЕСКД);
- обозначение (ЕСКД);
- шаблон серийного номера;
- описание;

- комментарии;
- эскизная документация (КД);
- конструкторская документация (КД);
- изготовление ПП;
- закупка ВОМ;
- монтаж УП;
- руководство по эксплуатации;
- наличие функционала Container OPU;
- наличие линейных и клиентских протоколов;
- функционал резервирования;
- файл для производства;
- энергопотребление (максимальное), Вт;
- энергопотребление (номинальное), Вт;
- номинал шунта 12 В, мОм;
- номинал шунта 3.3 В, мОм;
- мощность, Вт;
- версия печатной платы;
- клиентская документация;

Кроме того, учитывалось требование о сохранении истории изменений статусов и привязки устройств к различным проектам, что также оказывало влияние на структуру данных.

Таким образом, сформированные требования к данным определили основу для проектирования структуры базы данных, обеспечивая возможность гибкого и надёжного хранения информации об устройствах и их привязке к бизнес-процессам компании.

2.3.2 Проектирование структуры базы данных

На основе выявленных требований к данным была спроектирована структура базы данных, обеспечивающая надёжное хранение информации об устройствах, их принадлежности к проектам и программам, а также текущих статусах разработки и производства.

В процессе проектирования были выделены основные таблицы:

- Device - основная таблица, содержащая сведения об устройствах;
- EquipmentName - таблица для хранения наименований устройств;
- Project — таблица проектов разработки (Рисунок X);
- ResearchProject - таблица внутренних исследовательских проектов (RnD);
- SoftwareProgram — таблица программных продуктов, связанных с устройствами;
- ProductionLine - таблица производственных линий;
- DevStatus и ProStatus - таблицы статусов устройств.

Таблица Device включает ключевые поля:

- наименование устройства (связь через ForeignKey с EquipmentName);
- код устройства;
- принадлежность к проекту разработки (Project);
- связанный исследовательский проект (ResearchProject);
- связанная программа (SoftwareProgram);
- производственная линия (ProductionLine);
- статус разработки (DevStatus);
- статус производства (ProStatus);
- информация о коммерческой позиции устройства.

Для реализации связей использовались стандартные внешние ключи Django (ForeignKey), что позволило автоматически поддерживать целостность

данных на уровне базы данных. При удалении записей из связанных таблиц применялись каскадные удаления (`on_delete=models.CASCADE`), что обеспечивало автоматическое удаление зависимых объектов и предотвращало появление "висячих" ссылок [44].

Кроме того, были введены дополнительные ограничения:

- обязательность заполнения полей для критически важных атрибутов (например, наименование устройства, связанный проект) (рисунок 8);
- уникальность некоторых полей (например, код устройства) для обеспечения однозначной идентификации записей [45] (рисунок 8).

```
class Project(models.Model):  # Suganov Boris +1 *
    code = models.CharField(verbose_name='код', max_length=80, required=True, unique=True)
    url_wiki = models.URLField(verbose_name='Wiki URL', blank=True)
```

Рисунок 8 – Реализация модели сущности Project

Связи между таблицами в большинстве случаев реализованы через отношения типа один-ко-многим (One-to-Many) (Рисунок X). Например, один проект может быть связан с несколькими объектами компонентов в таблице Component (рисунок 9), один проект разработки может содержать множество устройств (рисунок 9).

Также связь между таблицами Component и Version реализована через отношения типа многие-ко-многим (Many-to-Many) при помощи инструмента `through`, который позволяет задать в ручную таблицу `ComponentVersionMapping` и добавить в нее дополнительные атрибуты [46] (рисунок 9). Данное решение было выполнено из-за структуры заполнения данных в jira, которое потребовалось для внесения данных, где несколько множество объектов может относиться ко множеству объектов другой таблицы.

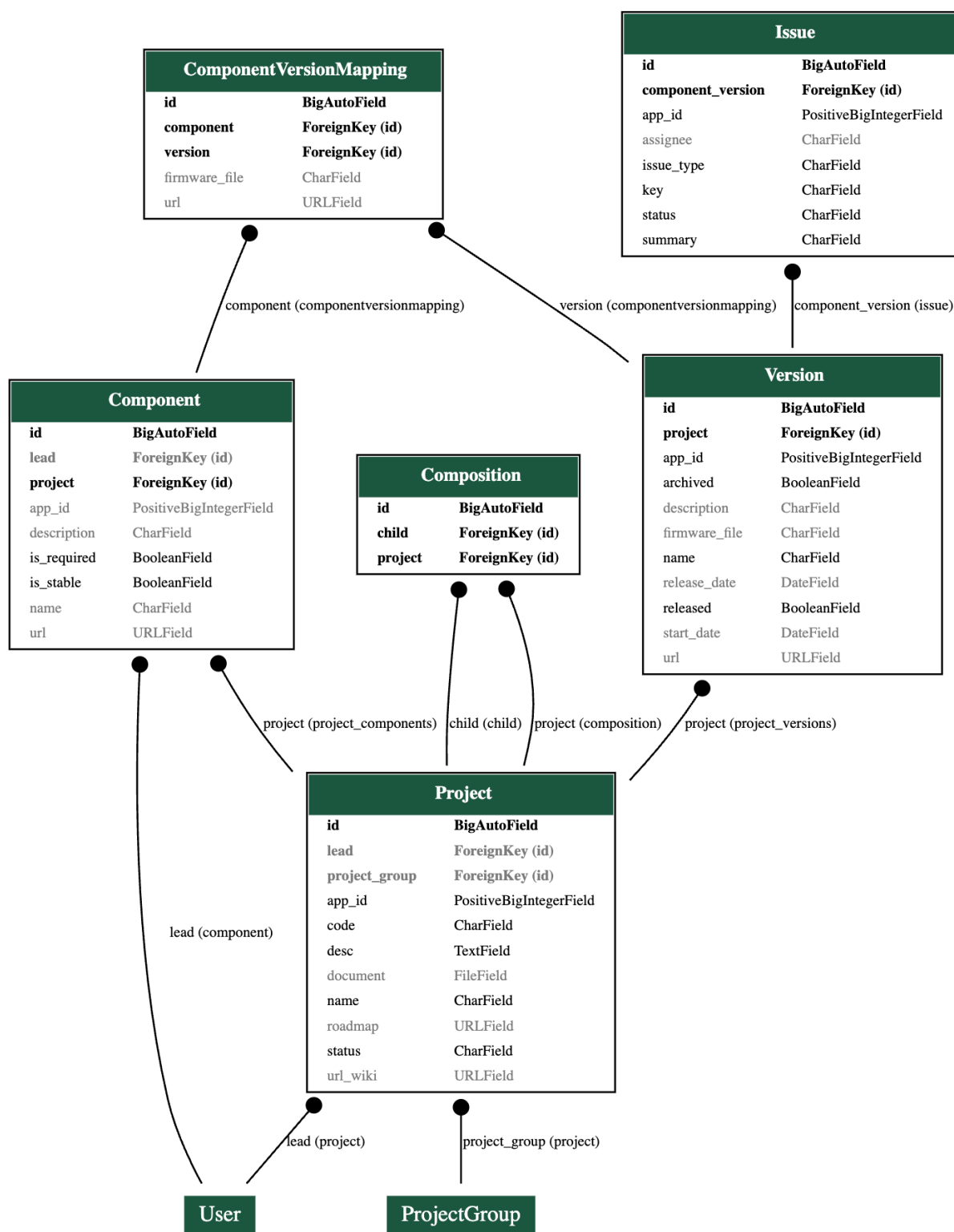


Рисунок 9 - Схема приложения Project

На этапе проектирования особое внимание уделялось обеспечению логической целостности структуры базы данных и возможности её дальнейшего масштабирования. При необходимости предусматривалась

возможность добавления новых атрибутов и связей без кардинальной перестройки существующей схемы.

Таким образом, проектирование структуры базы данных было направлено на создание логически обоснованной и надёжной системы хранения данных, способной поддерживать все ключевые бизнес-процессы компании и обеспечивать развитие проекта в будущем.

2.3.3 Нормализация данных

На этапе проектирования базы данных особое внимание уделялось вопросам нормализации данных с целью устранения избыточности, обеспечения целостности информации и упрощения работы с данными в дальнейшем [48].

Структура базы данных была нормализована до третьей нормальной формы (3NF), что подразумевает следующее:

- все неключевые атрибуты зависят только от первичного ключа;
- отсутствуют транзитивные зависимости между неключевыми атрибутами [48];
- каждая таблица описывает один логический объект или сущность [51].

Применение нормализации позволило разбить данные на логически обособленные таблицы. Например:

- наименования устройств были выделены в отдельную таблицу EquipmentName, что позволило избежать дублирования наименований в основной таблице устройств;
- статусы разработки и производства также вынесены в отдельные справочники DevStatus и ProStatus, а не хранятся в основной таблице Device как текстовые значения.

Также при проектировании базы данных рассматривалась частичная денормализация данных для повышения производительности при выборке. Но

было принято решение придерживаться нормализованной структуры из-за небольших объемов данных на начальных этапах [52].

Применение принципов нормализации положительно сказалось на:

- надёжности системы хранения данных;
- обеспечении готовности базы данных к масштабированию в случае роста объёмов информации;
- Снижение вероятности ошибок при обновлении и удалении данных;
- Чистоте структуры базы.

Для хранения данных веб-приложения была выбрана система управления базами данных PostgreSQL, обладающая широкими возможностями для построения надёжных, высокопроизводительных и масштабируемых систем хранения информации [53].

В процессе реализации проекта были активно использованы следующие особенности PostgreSQL:

Во-первых, применялись внешние ключи (ForeignKey) для поддержания целостности данных на уровне базы. Все основные сущности — устройства, проекты, программы, линии производства, статусы — были связаны между собой с использованием ограничений ссылочной целостности, что исключило возможность появления некорректных данных при обновлении или удалении записей [50].

Во-вторых, были сгенерированы индексы на полях для фильтрации и сортировки данных в таблицах: устройств, проектов, компонентов и т.д. Что позволило сократить время выполнения запросов и повысить производительность сервиса.

В-третьих, при импорте или изменении больших объемов данных использовались транзакции, которые автоматически обрабатывались встроенными средствами Django ORM. Это позволило избежать некорректное изменения данных в базе данных в случае ошибок при выполнении сложных

операций [47]. Например, транзакции использовались в Django команде по обновлению заказов из 1с (рисунок 10)

```
@transaction.atomic
def handle(self, *args, **options):
    """
    Основной метод, выполняющий последовательность операций:
    очищение модели, получение данных, сохранение и логирование результатов.
    """
    self.__clean_model()
    self.__fetch_device_with_orders()
    self.__prepare_orders_save()
    self.__save_orders()
    self.__log_results()
```

Рисунок 10 – Использование транзакций в проекте

Postgresql позволил спроектировать надежную и устойчивую базу данных, а также готовую к масштабированию и расширению функционала.

Проектирование базы данных в рамках веб-приложения осуществлялось с использованием стандартных средств фреймворка Django — встроенной системы моделей и механизма миграций [54].

Основным инструментом проектирования базы данных стала Django ORM (Object-Relational Mapping) [55].

Модели данных описывались в виде Python-классов, где каждому атрибуту соответствовало поле в базе данных. Такой подход обеспечивал:

- высокую читаемость и понятность структуры данных;
- тесную интеграцию моделей с бизнес-логикой приложения;
- автоматическое управление связями между сущностями через внешние ключи;
- возможность централизованного контроля правил валидации данных.

Также огромным преимуществом стало использования Django ORM в сочетании с визуальными схемами стали (рисунок 9):

- снижение вероятности ошибок при проектировании;
- ускорение процесса внесения изменений в структуру базы данных;
- возможность быстрого создания документации по структуре данных;
- поддержка единого подхода к описанию и управлению данными на протяжении всего жизненного цикла проекта.

Сложности при проектировании структуры базы данных в основном касались учёта специфических бизнес-требований, таких как необходимость хранить связанные сущности с различными вариантами привязок и состояний. Однако использование Django ORM и продуманная система миграций позволили гибко адаптировать структуру базы без серьёзных трудозатрат.

Таким образом, выбор инструментов для проектирования базы данных обеспечил удобство разработки, надёжность хранения информации и лёгкость сопровождения системы на всех этапах её жизненного цикла [56].

2.3.4 Выводы

Важнейший этап при разработке веб-приложения – проектирование базы данных. В данном этапе закладывается основа для хранения информации. Важно спроектировать базу данных так, чтобы она надёжна хранила и эффективно обрабатывала информацию.

В данном этапе были выделены ключевые сущности базы данных: устройства, проекты, компоненты, - что позволило сформировать корректную структуру данных с учётом принципов нормализации данных до третьей нормальной формы и хороших практик в проектировании базы данных, что минимизировало избыточность данных и упростило их сопровождение [57].

Также использование PostgreSQL позволило повысить надёжность работы базы данных и её производительность при помощи внешних ключей, индексов и транзакций.

В качестве основного инструмента проектирования была использована Django ORM, что позволило тесно интегрировать модели данных с бизнес-

логикой приложения, автоматизировать процесс создания и изменения схемы базы данных и упростить поддержку проекта в долгосрочной перспективе.

Таким образом, реализованная структура базы данных полностью соответствует требованиям проекта, обеспечивает надёжность хранения информации, готова к дальнейшему масштабированию и развитию вместе с эволюцией бизнес-процессов компании.

2.4 Разработка серверной части

2.4.1 Структура серверной части

Серверная часть проекта реализована на основе фреймворка Django и имеет чётко организованную модульную структуру, что обеспечивает удобство разработки, масштабируемость и лёгкость сопровождения системы [58].

Код backend-приложения структурирован по принципу разбиения на отдельные Django-приложения в рамках одного основного проекта, что можно увидеть в таблице 6. Каждое приложение отвечает за определённый участок бизнес-логики.

Таблица 6 – Модули проекта и их назначение

<i>Модуль</i>	<i>Назначение</i>
device	Работа с учётом устройств
project	Управление проектами разработки
program	Управление программами и программными продуктами
order, agreement, price	Учёт заказов, соглашений и ценовых позиций
pcb, line, stage	Учёт производственных линий и этапов производства
user	Управление пользователями
common, status, document, journal, section	Вспомогательные приложения для общих справочников, статусов и документооборота

Такое разбиение на приложения соответствует архитектурному подходу Django и обеспечивает разделение ответственности: каждое приложение инкапсулирует свои модели, представления, маршруты и логику. В таблице 7 показано разделение логики.

Таблица 7 – Структура приложения приложений

<i>Файл</i>	<i>Назначение</i>
models.py	Описание моделей данных и связей между ними
views.py	Обработчики запросов от клиента
serializers.py	Сериализаторы данных для API (используется в проектах на Django REST Framework)
urls.py	Маршруты (URL-шаблоны) для конкретного приложения
forms.py	Формы для работы с HTML-формами (если используются)
services.py	Бизнес-логика, выходящая за рамки обработки запроса
utils	Дополнительные и вспомогательные функции, которые выходят за рамки обработки запроса и бизнес логики

Общая маршрутизация запросов организована в центральном файле urls.py, который подключает маршруты всех приложений через пространство имён (namespace) для удобства и предотвращения конфликтов имён.

Вспомогательные файлы настроек, такие как конфигурация логирования, безопасности и базы данных, вынесены в отдельные модули папок conf/ и system/, что дополнительно упрощает сопровождение проекта и подготовку разных окружений (разработка, тестирование, продакшен).

Таким образом, структура серверной части проекта обеспечивает:

- масштабируемость системы за счёт модульности;
- удобство поиска и сопровождения кода;
- возможность командной работы без конфликтов;
- готовность к дальнейшему расширению функциональности с минимальными изменениями в существующей архитектуре.

2.4.2 Реализация обработки запросов

Обработка HTTP-запросов в проекте реализована на основе стандартных механизмов Django с активным использованием классовых представлений (Class-Based Views, CBV) и, в некоторых случаях, функциональных представлений (Function-Based Views, FBV), где это было необходимо для простоты и наглядности логики [59].

Маршрутизация запросов организована следующим образом: Каждое Django-приложение (device, project, program, и другие) содержит собственный файл `urls.py`, в котором описаны маршруты для соответствующих представлений. Все приложения подключаются к основному маршрутизатору проекта через общий `urls.py`, что обеспечивает централизованное управление путями.

Запросы от клиента поступают через стандартный стек обработки Django:

1. Приходит HTTP-запрос от клиента;
2. Django определяет, какое представление должно обработать запрос, на основе маршрутов (URL-паттернов);
3. Представление принимает запрос, выполняет соответствующие действия и возвращает ответ (HTML-страницу, редирект, JSON-ответ).

В проекте поддерживаются все основные типы HTTP-запросов:

- GET для получения данных (например, списка устройств, подробностей о проекте);
- POST для создания новых записей (например, добавление устройства);
- PUT/PATCH для обновления существующих данных (например, изменение статуса устройства);
- DELETE для удаления записей (например, удаление проекта или программы).

В большинстве приложений для обработки запросов используются классовые представления (ListView, DetailView, CreateView, UpdateView, DeleteView) из библиотеки Django Generic Views. Это позволяет:

- минимизировать дублирование кода;
- переиспользовать стандартные механизмы обработки форм и работы с моделями;
- легко расширять функциональность через переопределение методов (get_queryset, form_valid, get_context_data и др.).

Пример работы:

- Для отображения списка устройств используется ListView, который автоматически выбирает все записи модели Device и передаёт их в шаблон (рисунок 11).
- Для создания нового устройства применяется CreateView, где задаются модель формы, список полей и метод обработки успешной отправки формы (рисунок 11).

```
class DeviceTypeExport(DeviceFilterMixin, ExportView):  # Sukanov Boris *
    model = Device
    resource_class = DeviceResource

class DeviceTypeCreate(PermissionRequiredMixin, CreateViewLogEntryMixin):  # Sukanov Boris *
    model = Device
    form_class = InternalDeviceTypeCreateForm
```

Рисунок 11 – Использование Class-Based Views

В случаях, где требовалась более специфическая логика обработки, применялись функциональные представления. Например, для обработки нестандартных API-запросов или AJAX-вызовов.

Обработка ошибок и некорректных запросов также организована средствами Django:

- валидация данных происходит на уровне форм и сериализаторов;
- в случае ошибок валидации пользователю возвращается сообщение об ошибке и форма с сохранёнными полями;
- сервер возвращает стандартные HTTP-коды ответов (например, 404 — если объект не найден, 403 — при отсутствии прав доступа).

Таким образом, система обработки запросов построена по принципу:

- единообразие маршрутизации для всех приложений;
- стандартизированная обработка CRUD-операций;
- логическое разделение по видам представлений;
- надёжная валидация и обработка ошибок.

Выбранный подход позволяет поддерживать чистоту кода, легко масштабировать проект и быстро добавлять новые модули или функции без необходимости переписывать существующую архитектуру обработки запросов.

2.4.3 Реализация бизнес-логики

В рамках разработки серверной части проекта большое внимание уделялось организации бизнес-логики - совокупности правил и процедур, которые обеспечивают выполнение специфических задач системы в соответствии с требованиями бизнеса.

В проекте бизнес-логика была структурирована таким образом, чтобы отделять её от представлений и моделей, обеспечивая чистоту кода, удобство тестирования и возможность масштабирования. Основные принципы организации бизнес-логики включали:

- минимизация дублирования кода;
- локализация бизнес-правил в соответствующих слоях приложения;

- явное разделение обязанностей между обработкой запроса, работой с данными и реализацией бизнес-процессов.

На практике бизнес-логика проекта реализована на разных уровнях: моделей, представлений, вспомогательных сервисов.

Часть бизнес-правил встроена непосредственно в модели через методы экземпляров (save, clean, delete) и менеджеры (QuerySet). Например, при сохранении объекта устройства (Device) автоматически могут устанавливаться значения полей по умолчанию, проверяться корректность заполнения полей или валидация уникальности связей.

Представления отвечают за обработку запросов и вызов соответствующей бизнес-логики. Стандартные операции, такие как создание, редактирование и удаление записей, выполняются через стандартные механизмы Django Generic Views, что обеспечивает единообразие подхода.

Для более сложных процессов в некоторых приложениях используются отдельные вспомогательные модули (services.py или утилитарные функции внутри приложения). В этих сервисах инкапсулирована логика, связанная с комплексной обработкой данных, выполнением нескольких связанных действий (например, при создании устройства автоматически привязывается программа или проставляется статус).

Примеры реализованной бизнес-логики в проекте:

- автоматическое назначение статуса разработки устройству при его создании;
- проверка возможности удаления устройства: удаление запрещено, если устройство связано с активным проектом или заказом;
- актуализация данных при изменении связанной сущности (например, при изменении программы устройства должны обновляться связанные поля);

- контроль прав доступа на выполнение определённых операций, например, возможность редактирования или удаления устройств только пользователями с определёнными ролями.

Для реализации бизнес-логики активно использовались встроенные механизмы Django:

- сигналы (`post_save`, `pre_delete`) для отслеживания событий сохранения и удаления объектов;
- система валидации форм и моделей;
- кастомные методы менеджеров (`custom QuerySets`) для реализации специфических выборов.

Организация бизнес-логики в проекте позволяет:

- быстро вносить изменения в правила без переписывания базовой структуры приложения;
- централизованно управлять поведением данных и операций;
- обеспечивать высокую надёжность выполнения операций в соответствии с требованиями бизнес-процессов компании.

Таким образом, бизнес-логика проекта реализована с учётом лучших практик разработки на Django: чёткое разделение ответственности между слоями приложения, использование стандартных расширяемых механизмов фреймворка и соблюдение принципов чистоты и сопровождаемости кода.

2.4.4 Работа с базой данных через ORM

В проекте для работы с базой данных используется встроенная система объектно-реляционного отображения (**ORM**) фреймворка Django. Django ORM обеспечивает удобный способ взаимодействия с базой данных на уровне

объектов Python без необходимости ручного написания SQL-запросов, что значительно ускоряет разработку и повышает надёжность кода.

Основная часть работы с данными организована через модели (`models.py`) каждого отдельного приложения, например:

- `Device` в приложении `device` для хранения информации об устройствах;
- `Project` в приложении `project` для управления данными о проектах разработки;
- `Program` в приложении `program` для описания программных продуктов и линий.

Операции с базой данных осуществляются стандартными методами Django ORM.

- Создание записей — с помощью методов `.create()` или сохранения экземпляра модели через `.save()`.
- Получение данных — через методы `.get()`, `.filter()`, `.all()`, с возможностью фильтрации и сортировки.
- Обновление записей — через изменение полей модели и последующий вызов `.save()`, либо массовое обновление через `.update()`.
- Удаление записей — через методы `.delete()`.

Для повышения эффективности запросов в проекте применяются механизмы оптимизации выборки:

- использование `select_related()` для предзагрузки связанных объектов через внешние ключи в один запрос (рисунок 12);
- использование `prefetch_related()` для оптимизированной загрузки связанных наборов объектов в отдельные запросы (рисунок 12).

```

return DeviceFullVersion.objects \
    .filter(Q(device=self) | Q(compatible_devices=self)).distinct() \
    .select_related(
        'device',
        'device_hw_version',
        'device_sw_version',
    ).prefetch_related('linear_modules__name') \
    .annotate(released_at_or_updated_at=Coalesce("released_at", "updated_at")) \
    .order_by('-is_actual_release', '-released_at_or_updated_at')

```

Рисунок 12 – Оптимизация запросов в проекте

Эти методы значительно уменьшают количество обращений к базе данных, что особенно важно при работе с таблицами, содержащими большое количество записей или сложные связи между сущностями.

Например при получении списка полных версий устройств вместе с данными о устройстве и аппаратных версиях используется метод `select_related('device', 'sw_version')` (рисунок 12), что позволяет в одном запросе получить всю необходимую информацию без дополнительных обращений к базе данных.

Работа с данными строго типизирована: все связи между моделями определены через поля `ForeignKey`, `OneToOneField` и `ManyToManyField`, что обеспечивает поддержку ссылочной целостности на уровне базы данных.

Кроме того, для сложных выборок и бизнес-логики применяются кастомные менеджеры моделей и расширенные запросы через методы агрегирования (`annotate`, `aggregate`) и выражения (`Q`, `F` объекты).

Особенности работы через Django ORM позволили добиться:

- упрощения кода представлений и бизнес-логики;
- повышения читаемости и сопровождаемости кода;
- уменьшения числа ошибок при работе с базой данных;
- лёгкой адаптации системы к изменениям структуры данных через механизм миграций.

2.4.5 Реализация API

В проекте на основе библиотеки Django Rest Framework реализовано REST API для взаимодействия между серверной частью приложения и внешними клиентами.

Доступные функционал в API:

- предоставление данных об устройствах, проектах, коммерческих позициях и других сущностях;
- взаимодействие с внешними системами: 1c, jira и другими внутренними сервисами компании.

Реализация API организована следующим образом:

- для каждой модели создаются отдельные сериализаторы (serializers.py), отвечающие за преобразование данных между форматами Python-объектов и JSON;
- маршрутизация API-запросов осуществляется через отдельные файлы urls.py в каждом приложении;
- обработка запросов реализована через классовые представления DRF (ListAPIView, RetrieveAPIView, CreateAPIView, UpdateAPIView, DestroyAPIView) либо через наборы представлений (ViewSet) и маршруты роутеров.

Типичные эндпоинты, реализованные в проекте:

- получение списка устройств (GET /api/devices/);
- просмотр подробной информации об устройстве (GET /api/devices/{id}/);
- создание нового устройства (POST /api/devices/);
- обновление данных устройства (PUT/PATCH /api/devices/{id}/);
- удаление устройства (DELETE /api/devices/{id}/).

Сериализаторы позволяют управлять тем, какие поля отображаются в ответах API, а также валидировать данные, поступающие от клиента. Например, при создании нового устройства сериализатор проверяет наличие обязательных полей, корректность типов данных и соблюдение бизнес-правил.

Для повышения безопасности работы с API в проекте применяются:

- механизмы аутентификации пользователей через токены или сессии;
- разграничение прав доступа на уровне представлений с использованием проверок прав доступа DRF (`IsAuthenticated`, `IsAdminUser`, кастомные классы разрешений).

Формат данных обмена стандартизирован, так как все ответы API возвращаются в формате JSON. При успешной обработке запроса сервер возвращает соответствующие HTTP-коды (200 OK, 201 Created), а при ошибках — коды с пояснением причины (400 Bad Request, 403 Forbidden, 404 Not Found).

Применение Django REST Framework для реализации API позволило:

- ускорить процесс разработки за счёт использования стандартных компонентов;
- обеспечить читаемую и предсказуемую структуру взаимодействия с клиентскими приложениями;
- создать основу для будущего расширения функциональности системы без необходимости переработки архитектуры.

Таким образом, реализация API на базе Django REST Framework стала важным этапом в построении надёжной и расширяемой серверной части веб-приложения.

2.4.6 Обеспечение безопасности серверной части

Обеспечение безопасности серверной части веб-приложения было одним из приоритетных направлений в ходе разработки проекта. Безопасность данных, защита от атак и контроль прав доступа пользователей стали обязательными требованиями при проектировании архитектуры приложения.

В таблице 8 указаны реализованные меры безопасности в рамках проекта:

Таблица 8 – Меры безопасности и их описание

Мера безопасности	Описание
Аутентификация и авторизация пользователей	Проект использует встроенные механизмы Django для аутентификации пользователей. Все чувствительные действия, такие как создание, редактирование или удаление записей, доступны только авторизованным пользователям. На уровне представлений применяются классы разрешений (IsAuthenticated, IsAdminUser, а также кастомные разрешения при необходимости) для ограничения доступа к API и функционалу [60].
Контроль прав доступа	В системе реализована разграниченная модель доступа: разные группы пользователей имеют различные права на выполнение действий. Например, стандартный пользователь может только просматривать устройства, а создание, изменение или удаление записей доступно только пользователям с расширенными правами (например, администраторам или менеджерам проекта).
Защита от атак CSRF	Для всех HTML-форм в проекте применяются механизмы защиты от атак межсайтовой подделки запросов (Cross-Site Request Forgery, CSRF). Django автоматически добавляет CSRF-токен в формы и проверяет его наличие при обработке POST-запросов [61].
Защита от XSS-атак	Встроенная система шаблонов Django автоматически экранирует все выводимые данные в HTML-шаблонах, что предотвращает внедрение вредоносных скриптов (Cross-Site Scripting, XSS) через пользовательский ввод [61].

Защита от SQL-инъекций	Работа с базой данных ведётся исключительно через Django ORM, которая автоматически экранирует все параметры запросов и тем самым предотвращает возможность проведения атак с использованием SQL-инъекций [61].
Безопасность обработки ошибок	При возникновении ошибок пользователю возвращаются стандартные сообщения без раскрытия внутренних деталей системы. Серверные ошибки обрабатываются централизованно и логируются для последующего анализа без утечки конфиденциальной информации [61].
Шифрование данных при передаче	Для всех соединений используется протокол HTTPS, что позволяет обеспечить шифрованием передаваемые данные между клиентом и сервером.
Логирование и аудит событий	Все действия CRUD (создание, обновление, удаление объектов) фиксируются в базе данных, что позволяет просматривать действия пользователей и своевременно выявлять потенциальные ошибки.

Внедрение инструментов безопасности Django и их применение позволили разработать защищенную серверную часть, устойчивую к большинству видам атак.

На основе фреймворка Django реализована серверная часть веб-приложения, в которой устойчивость, безопасность и масштабируемость архитектуры соответствует всем требованиям проекта.

Модульная структура проекта, которая основана на разделении приложений по функциональности, упростила дальнейшую разработку и развитие системы, Использование Django инструментов: классовые представления, формы и других практики, - обеспечило единообразие обработки запросов и позволило сократить время на реализацию нового функционала.

В отдельный модуль вынесенная бизнес-логика проекта позволила повысить читаемость кода, а также возможность его протестировать. Использование инструмента для базы данных ORM позволил минимизировать вероятность ошибок.

Реализованный REST API при помощи Django REST Framework заложил основу для интеграция с внутренними сервисами: 1с, jira и др.

Особое внимание было уделено вопросам безопасности: аутентификация и авторизация пользователей, защита от CSRF- и XSS-атак, контроль целостности данных на уровне базы — все эти меры обеспечили надёжность работы серверной части в условиях реальной эксплуатации.

Таким образом, серверная часть проекта была реализована с использованием современных подходов и лучших практик разработки на Django, что позволило создать надёжный фундамент для всего веб-приложения.

2.5 Разработка клиентской части

2.5.1 Структура клиентской части

Клиентская часть веб-приложения реализована с использованием системы шаблонов Django в сочетании с фреймворком Bootstrap и классическими технологиями HTML, CSS и JavaScript. Архитектура фронтенда выстроена таким образом, чтобы обеспечить удобство масштабирования, повторное использование компонентов интерфейса и облегчение сопровождения проекта.

Все шаблоны страниц размещаются в каталоге `templates/`, который структурирован по приложениям. Например:

- шаблоны, связанные с устройствами, расположены в папке `app/templates/device/` (Рисунок 13);
- шаблоны для управления проектами в папке `app/templates/project/`.

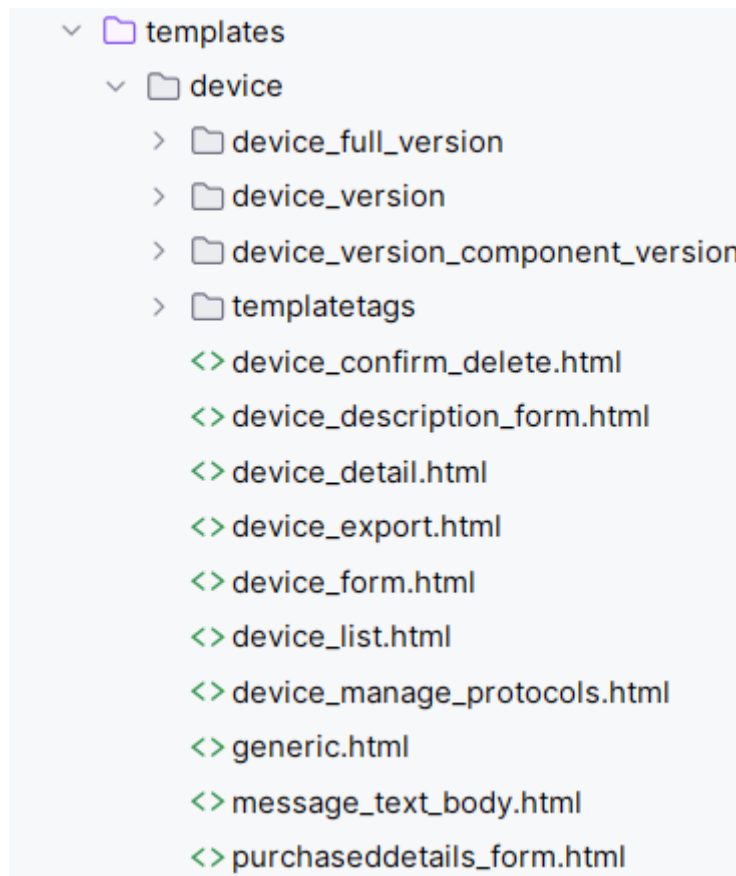


Рисунок 13 – Структура шаблонов Html

Такой подход обеспечивает логическую группировку файлов в соответствии с их принадлежностью к бизнес-области проекта и упрощает навигацию при разработке [62].

В проекте активно используется механизм наследования шаблонов. Реализован базовый шаблон (generic.html), содержащий:

- общую структуру страницы (шапка, подвал, боковая панель навигации);
- подключение общих стилей (CSS) и скриптов (JavaScript);
- места для вставки динамического контента через блоки { % block content % } (рисунок 14) и другие пользовательские блоки.

```
<div class="ms-2 mt-1">
    {% block bc %}{% endblock %}
</div>
```

Рисунок 14 – Пример базового шаблона

Все остальные страницы наследуют базовый шаблон, переопределяя только необходимые блоки. Это позволяет:

- стандартизировать внешний вид интерфейсов;
- избегать дублирования кода;
- упростить внесение изменений в оформление всего сайта через редактирование одного файла.

Статические файлы - стили, изображения и скрипты — размещены в папке `static/` (рисунок 15), с разделением по типам файлов:

- `static/css/` — таблицы стилей;
- `static/js/` — скрипты JavaScript;
- `static/img/` — изображения и иконки.

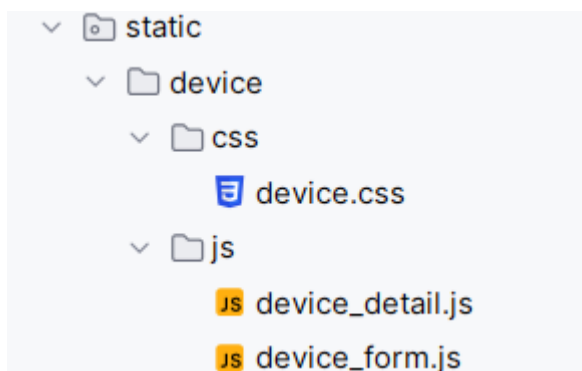


Рисунок 15 – Структура расположения статических файлов

Подключение стилей и скриптов осуществляется с использованием стандартных тегов Django `{% static %}` (Рисунок 16). Это обеспечивает правильную работу ссылок на статические файлы как в режиме разработки, так и на продакшене.

```
{% block js %}
    {{ block.super }}
    <script src="{% static "device_type_form.js" %}?v=0.1.0"></script>
{% endblock %}
```

Рисунок 16 – Подключение статических файлов

Маршрутизация страниц на уровне клиента организована через серверные маршруты Django: каждая ссылка в интерфейсе указывает на определённый URL-адрес, обрабатываемый сервером и возвращающий соответствующий HTML-шаблон.

Дополнительная маршрутизация через JavaScript внутри страниц применяется только для динамических элементов (например, модальных окон или загрузки данных через AJAX).

Таким образом, структура клиентской части проекта построена на принципах модульности, переиспользуемости компонентов и единообразия оформления. Это обеспечивает удобство сопровождения фронтенда, упрощает работу команды разработчиков и позволяет быстро адаптировать интерфейсы под новые требования бизнеса.

2.5.2 Разработка пользовательских интерфейсов

В рамках клиентской части веб-приложения было разработано множество пользовательских интерфейсов, ориентированных на удобство работы с системой для конечного пользователя. При разработке интерфейсов применялись современные принципы проектирования UX/UI, а также активно использовались стандартные компоненты Bootstrap для обеспечения адаптивности и единообразного стиля оформления [64].

Основные типы пользовательских страниц включают:

- страницы списка объектов (например, список устройств, проектов, программ) (рисунок 17);
- страницы создания и редактирования записей через формы (Рисунок 18);
- административные страницы управления справочниками и статусами (Рисунок 19);
- страницы детального просмотра объекта (например, подробности об устройстве) (Рисунок 20).

Реестр разработок T8

ГлавнаяПроектыРазработкиПрайсТестыPCB_UIDДоговоры

Главная / Разработки

Разработки

ID	Наименование (ЕСКД) ⓘ	Короткий код	Обозначение (ЕСКД)	Проект ^{RnD/Jira}	Линейка
2	Test Device Name DEV-001-TST	MON-222	A1.123.456	RnD Project/	Line A
8	Device A DEV-A001	V1DC-11-QQ	A1.123.001	RnD Alpha/	Line 1
6	Device C DEV-C003	V1DC-11-QQ	C1.123.003	RnD Alpha/	Line 1
3	Device E DEV-E005	V1DC-11-QQ	E1.123.005	RnD Alpha/	Line 1
4	Device E DEV-E005	V1DC-11-QQ	E1.123.005	RnD Alpha/	Line 1
1	DWDМ устройство КАМАР-10-XX-12-22	-	-	-/ -	-

Всего объектов: 6

Рисунок 17 – Страница списка объектов

Главная / Разработки / Редактирование DEV-A001выыв

Device A DEV-A001выыв

Текстовая часть наименования*

Device A

Длинный код*

DEV-A001выыв

Коммерческая позиция

Выберите коммерческую позицию.

Класс устройства

Class A

Главное устройство

☒ Актуальность

☒ В прайсе/не в прайсе
разработка планируется к продаже

Линейка

Line 1

Подкласс

Section A / Subsection 1

Программа разработки

Program X

Проект

Проект разработки (RnD)

RnD Alpha

Проект разработки (Jira)

Статусы

Статус разработки

(Prototype)

Статус производства

(Production Ready)

Даты

Планируемая дата доступности к заказу

09.05.2025

Крайняя дата приема заказа

09.05.2025

Дата окончания поддержки

09.05.2025

Дата, после которой заказы на устройство не принимаются.

Дата окончания доработок ПО по запросам заказчика





Рисунок 18 – Страница редактирования устройства

Администрирование сайта

AGREEMENT		
Лицензионные договоры	+ Добавить	✎ Изменить
CUSTOMER		
Покупатель устр-в	+ Добавить	✎ Изменить
DEVICE		
Версии устройств (Hw/Sw)	+ Добавить	✎ Изменить
Клиентские модули	+ Добавить	✎ Изменить
Линейные модули	+ Добавить	✎ Изменить
Методы резервирования ContainerOPU	+ Добавить	✎ Изменить
Наименования устройств	+ Добавить	✎ Изменить
Ограничения кол-ва клиентских протоколов	+ Добавить	✎ Изменить
Ограничения кол-ва линейных протоколов	+ Добавить	✎ Изменить
Полные версии устройств	+ Добавить	✎ Изменить
Протоколы	+ Добавить	✎ Изменить
Системы управления	+ Добавить	✎ Изменить
Устройства	+ Добавить	✎ Изменить
DOCUMENT		
Документы устр-в	+ Добавить	✎ Изменить
Типы документов	+ Добавить	✎ Изменить

Рисунок 19 – Административная панель Django

Test Device Name DEV-001-TST

Наименование (ЕСКД):	Test Device Name DEV-001-TST
Длинный код:	DEV-001-TST
Коммерческое название:	Price Item Device Name MON-222
Короткий код:	MON-222
Статус "В прайсе"	Не включено
Линейка:	Line A
Классификатор:	Section A / Subsection A
Проект:	RnD Project <small>RnD Jira</small>
Программа:	Test Program
Обозначение ЕСКД (десятичный номер):	A1.123.456
Серийный номер:	SN-####
Статус разработки:	Design in Progress ()
Статус производства:	Not Started ()
Печатная плата:	v1.0
Документация (КД):	http://example.com/dd 
Документация (РЭ):	http://example.com/manual 
Клиентская документация:	https://doc.t8.ru/test
Файл производства:	prod_file.pdf 
Тип устройства:	Разработка
UUID:	2ec97f0d-b54b-46e0-8f97-dd5ff47972d4 

[Редактировать](#)

[Версии](#) [Аппаратные версии](#) [Программные версии](#)

Рисунок 20 – Страница устройства

Каждая страница разрабатывалась с использованием шаблонной системы Django с расширением от базового шаблона `base.html`, что обеспечивало унифицированное оформление всех разделов приложения.

Интерфейсы строились следующие паттерны: формы, таблицы и списки, кнопки действий и панели навигации, стандартные компоненты интерфейса.

Для создания и редактирования данных активно использовались Django-формы, которые автоматически интегрируются с моделями данных через `ModelForm` [69].

Визуальное оформление форм было стандартизировано с использованием библиотеки `django-crispy-forms`, что позволило создавать аккуратные и адаптивные формы без необходимости ручной вёрстки каждой

формы [70]. Формы содержали как обязательные поля (например, наименование устройства), так и дополнительные поля с подсказками для пользователя (Рисунок 19). Отображение списков объектов (например, устройств) организовано в виде табличных представлений с возможностью сортировки и фильтрации данных (Рисунок 21).

Фильтр

Поиск

Поиск по полям: наименование, длинный код, описание, комментарий, проект, ЕСКД. Можно указывать несколько ключевых слов через запятую.

☐ Актуальность

Наименование

Ничего не выбрано

Линейка

Ничего не выбрано

Класс

Ничего не выбрано

Проект (Jira)

Ничего не выбрано

Рисунок 21 – Фильтр устройств на странице списка устройств

Каждая строка таблицы содержала краткую информацию об объекте, а также ссылки на действия — просмотр подробностей, редактирование, удаление.

Все страницы оснащены удобными кнопками для выполнения основных операций: добавить запись, сохранить изменения, вернуться к списку (Рисунок 20).

Основная навигация по разделам приложения была вынесена в отдельное

меню, расположенное в верхней панели или боковой панели сайта (в зависимости от страницы) (рисунок 19). Использование стандартных компонентов Bootstrap (рисунок 21) позволило быстро собирать страницы и обеспечить современный, интуитивно понятный внешний вид интерфейсов без необходимости разработки собственных компонентов с нуля.

При разработке пользовательских интерфейсов особое внимание уделялось простоте навигации, минимизации количества кликов до выполнения целевого действия, а также обеспечению удобства ввода и чтения данных.

Таким образом, построение пользовательских интерфейсов проекта было направлено на достижение баланса между функциональностью, удобством работы и эстетикой оформления, что напрямую влияет на общее качество взаимодействия пользователей с системой.

2.5.3 Работа с данными на клиенте

Обработка данных на стороне клиента в проекте организована стандартными средствами Django и JavaScript, с приоритетом на надёжность и предсказуемость взаимодействия между пользователем и системой.

Основной механизм передачи данных от клиента к серверу реализован через формы (рисунок 22), генерируемые средствами Django. Пользователь заполняет формы на страницах создания или редактирования объектов, после чего данные отправляются на сервер с помощью стандартных HTTP POST-запросов. На стороне сервера формы проходят валидацию, и в случае успешной проверки данные сохраняются в базу данных.

Однако для повышения интерактивности на некоторых страницах используется AJAX-запросы с помощью JavaScript. Это позволяет:

- отправлять данные без полной перезагрузки страницы [65];

- динамически обновлять части страницы (например, встраивание новых элементов в список без перезагрузки) [65];
- обрабатывать ответы сервера на лету и выводить уведомления об успешном или неуспешном выполнении операции [76].

Обработка AJAX-запросов построена следующим образом:

- на стороне клиента отправляется запрос с помощью JavaScript (обычно через fetch API);
- сервер принимает данные и возвращает ответ в формате JSON;
- в зависимости от результата запроса интерфейс либо обновляется, либо отображается сообщение об ошибке.
- Для обеспечения устойчивости и безопасности взаимодействия:
- во всех формах используется защита от CSRF-атак за счёт встроенного механизма Django (`{% csrf_token %}`) (Рисунок 22);
- данные валидируются как на стороне клиента (проверка обязательных полей) , так и на стороне сервера (глубокая валидация через Django-формы и сериализаторы).

```
{% endfor %}
<form method="post">{% csrf_token %}|
<div>
{% for obj in queryset %}
```

Рисунок 22 – Использование автоматически генерируемых форм

Кроме того, интерфейсы поддерживают отображение системных сообщений о результате операции, например, уведомления об успешном сохранении записи или предупреждения о необходимости заполнения обязательных полей (Рисунок 19).

Таким образом, работа с данными на клиенте построена на сочетании классического подхода с отправкой форм и использованием AJAX там, где это

повышает удобство пользователя и сокращает время отклика системы, сохраняя при этом надёжность и простоту реализации.

2.5.4 Адаптивность и оформление интерфейсов

Одним из важных требований к разработке клиентской части веб-приложения было обеспечение корректной работы интерфейсов на различных устройствах — от настольных компьютеров до планшетов и мобильных телефонов. Для решения этой задачи в проекте использовался фреймворк Bootstrap, который предоставил широкие возможности для быстрой реализации адаптивной вёрстки [73].

Все страницы интерфейса строились с использованием адаптивной сеточной системы Bootstrap:

- применялись классы `container`, `row`, `col` для формирования структуры страниц (рисунок 23);
- размеры элементов автоматически подстраиваются под ширину экрана устройства благодаря встроенной системе медиазапросов;
- формы, таблицы, списки объектов и другие компоненты корректно масштабируются без необходимости написания дополнительных стилей для каждой ширины экрана.

```
<main class="row">  
  <section class="col-9">  
    <h1 class="h2 d-inline">Проекты разработки</h1>
```

Рисунок 23 – Использование Bootstrap классов

Визуальное оформление страниц также базировалось на стандартных компонентах Bootstrap, включая:

- стилизованные кнопки (`btn`, `btn-primary`, `btn-danger`) (рисунок 24);

- карточки (card) для отображения информации о сущностях;
- модальные окна (modal) для подтверждения действий или отображения дополнительных сведений;
- всплывающие уведомления (alerts) для вывода сообщений об успехе или ошибках при взаимодействии пользователя с системой.

```
submit_button = Submit("submit", "Применить")
submit_button.field_classes = "btn btn-sm btn-primary"
self.helper.add_input(submit_button)
```

Рисунок 24 – Стилизованные кнопки в коде, задающиеся в формах

Особое внимание уделялось обеспечению удобства навигации на всех типах устройств:

- меню навигации (navbar) автоматически схлопывается в мобильную версию на маленьких экранах (рисунок 25);
- формы поиска и фильтрации остаются доступными и удобными даже при ограниченной ширине экрана;
- крупные кликабельные элементы, адаптированные под сенсорные экраны (рисунок 25).

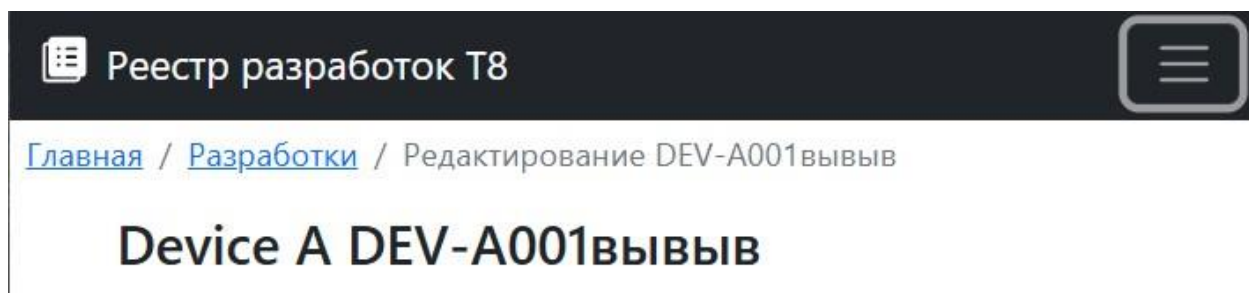


Рисунок 25 – Навигационная панель в разрешении мобильного телефона

Для улучшения пользовательского опыта были добавлены:

- иконки и пиктограммы для визуализации действий и статусов (с использованием библиотеки иконок Bootstrap Icons) (рисунок 26);
- всплывающие подсказки (tooltip) для пояснения значений полей и действий кнопок (рисунок 27);
- динамические уведомления о результатах операций без перезагрузки страницы (Рисунок 28).

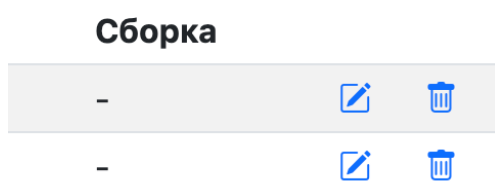


Рисунок 26 – Использование Bootstrap иконок в клиентской части

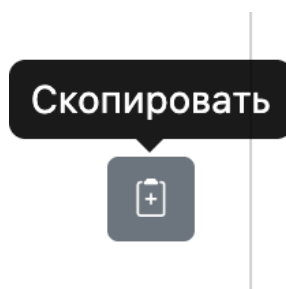


Рисунок 27 – Tooltip

Не удалось подключиться к Jira

Рисунок 28 - Динамические уведомления о результатах операций

Оформление всех страниц выдержано в едином стиле, что обеспечивает целостность восприятия системы пользователями и способствует лёгкой адаптации к интерфейсу новых пользователей.

Следовательно, применение такого инструмента как Bootstrap, а также применение хороших практики UX UI позволили создать современный адаптивный.

Главной задачей в разработке клиентской части веб-приложения было создание удобного, функционального и адаптированного интерфейса для пользователя. Применение такого инструмента, как Bootstrap, а также применение хороших практик UX UI и построение архитектуры фронтенда на основе системы шаблонов Django позволили создать современный адаптивный, а также эффективно организовать структуру страниц и обеспечить единообразное оформление интерфейсов.

Были разработаны страницы для работы с основными сущностями сервиса: устройства, проекты, договоры, коммерческие позиции и тестирование устройств. При помощи Bootstrap стилей и Django-форм были разработаны формы, таблицы, кнопки, навигационные панели.

Работа с данными на клиенте сочетала использование классической отправки форм и применение AJAX-технологий для повышения интерактивности и отзывчивости интерфейса. При этом особое внимание уделялось валидации данных и защите от возможных угроз безопасности.

Адаптивная вёрстка позволила добиться корректного отображения интерфейсов на устройствах с различными размерами экранов, что существенно повысило удобство работы с системой для всех категорий пользователей.

Таким образом, разработка клиентской части проекта успешно достигла поставленных целей: был создан современный, удобный и надёжный пользовательский интерфейс, полностью соответствующий функциональным требованиям и стандартам качества.

2.6 Интеграция с внешними сервисами

2.6.1 Выбор сервисов для интеграции

В рамках разработки веб-приложения возникла необходимость интеграции с внешними системами для расширения функциональности проекта и автоматизации бизнес-процессов компании. Одним из основных направлений интеграции стала связка веб-приложения с системой управления проектами Jira.

Выбор Jira как ключевого внешнего сервиса для интеграции был обусловлен следующими факторами:

- Jira активно используется в компании для управления задачами разработки, ведения проектной документации и отслеживания статусов работ;
- устройства, учет которых ведётся в рамках веб-приложения, часто напрямую связаны с проектами разработки, зарегистрированными в Jira;
- необходимость автоматического сопоставления устройств с проектами, задачами или этапами разработки, зафиксированными в Jira;
- Интеграция с Jira позволила решить сразу несколько практических задач:
- автоматическое подтягивание информации о проектах разработки, к которым привязаны устройства;
- возможность обновления статусов устройств на основе изменений в задачах Jira;
- унификация работы с данными о проектах без необходимости ручного дублирования информации между системами.

Кроме Jira, в перспективе рассматривались возможности интеграции с внутренними учетными системами компании для автоматизации коммерческих процессов, однако на момент разработки основной акцент был сделан на интеграции с системой управления проектами как наиболее критически важной для бизнес-логики проекта.

Таким образом, выбор Jira в качестве основного сервиса для интеграции был продиктован необходимостью более тесной связи процессов учёта устройств с процессами управления разработкой, что напрямую повышает прозрачность и эффективность работы всей системы.

2.6.2 Техническая реализация интеграции

В разработке интеграции с системой управления проектами Jira была использована официальная библиотека jira для python, которая позволила обеспечить доступ к RestApi Jira v2.

Интеграция реализована следующим образом:

Был реализован Django-приложение внутри веб-приложения, который обеспечил настройку соединения через API (рисунок 29). В коде были предусмотрены следующие настройки для подключения:

- Url сервера Jira компании, так как у компании jira развернут на собственном сервере для удобства и безопасности (рисунок 29);
- логин и токен доступа (рисунок 29).

```
def __connect_to_jira(self) -> tp.Optional[JIRA]: 1 usage  1 Suganov Boris +1
    """
    Подключается к Jira и возвращает объект клиента Jira.

    :return: Экземпляр клиента Jira, если подключение успешно, иначе ``None``.
    """
    try:
        return JIRA(
            server=settings.JIRA_SERVER,
            basic_auth=(settings.JIRA_USERNAME, settings.JIRA_PASSWORD)
        )
    except Exception as e:
        self.errors.append(f"Не удалось подключиться к Jira")
```

Рисунок 29 – Метод подключение к Jira

В рамках проекта через API Jira выполняются следующие операции:

- получение списка компонентов устройств и версий компонента;

- получение информации о задачах, связанных с конкретными устройствами;
- обновление локальных данных о статусах проектов и задач;

Передача данных между веб-приложением и Jira организована в формате JSON.

Запросы формируются динамически на основе действий пользователя в интерфейсе приложения (например, при привязке устройства к проекту разработки).

Также разработан дополнительный функционал, который инкапсулирует работу API И обеспечивает:

- обработку ошибок;
- стандартизацию к внешнему сервису;
- логирование ключевых операций.

Подводя итог, Интеграция сервиса с внешними сервисами: Jira, 1с, стала важной частью проекта, которая позволяет расширить его функционал и повысить связь бизнес-процессов. Интеграция с Jira разработана при помощи официального API, с соблюдением требований безопасности и стандартизации передачи данных. Это позволило обеспечить надёжную связь между устройствами и проектами разработки без потери пакетов информации. Также Связь с Jira позволила обеспечить автоматическую синхронизацию данных для компонентов, версий компонентов и задач.

Благодаря использованию официальной библиотеки jira для Python и построению взаимодействия через REST API, удалось достичь высокой надёжности работы интеграции, а также обеспечить масштабируемость решения.

Особое внимание при проектировании интеграции уделялось вопросам безопасности: аутентификация осуществлялась через API-токены, передача

данных шла исключительно по HTTPS, были ограничены права доступа, а все операции логировались для последующего аудита.

Таким образом, внедрение интеграции с внешними сервисами не только расширило функциональность веб-приложения, но и повысило его ценность как единого инструмента для поддержки процессов разработки и учёта внутри компании. В перспективе структура проекта позволяет легко реализовать новые интеграции с другими корпоративными системами при необходимости.

2.7 Выводы по второй главе

В ходе выполнения работ, описанных во второй главе, была разработана полноценная архитектура веб-приложения, выбраны и обоснованы технологии реализации, спроектирована база данных, а также реализованы серверная и клиентская части системы.

На этапе проектирования архитектуры было принято решение использовать монолитную модель построения приложения на основе фреймворка Django, что обеспечило необходимую скорость разработки, надёжность и удобство дальнейшего сопровождения проекта. Выбор стека технологий - Django, PostgreSQL, JavaScript и Bootstrap — позволил создать сбалансированную платформу для решения поставленных задач.

В процессе проектирования базы данных были соблюдены принципы нормализации, оптимизирована скорость доступа к данным. Преимущества выбранной СУБД были эффективно использованы при построении модели хранения информации.

Во время разработки Backend части приложения была обработка основных инструментов работы с данными, разработан API функционал на основе DRF, также соблюдены стандарты безопасности хранения и передачи данных.

Внедрение интеграции с внешними сервисами не только расширило функциональность веб-приложения, но и повысило его ценность как единого

инструмента для поддержки процессов разработки и учёта внутри компании. В перспективе структура проекта позволяет легко реализовать новые интеграции с другими корпоративными системами при необходимости.

3 Тестирование и внедрение

3.1 Подходы к тестированию веб-приложения

Процесс тестирования веб-приложения был построен с учётом особенностей архитектуры проекта и его ключевых компонентов: серверной части на Django, API-интерфейсов и клиентских интерфейсов, построенных на шаблонах.

В рамках проекта применялись следующие виды тестирования.

- Юнит-тестирование. Юнит-тесты разрабатывались для проверки отдельных модулей системы, включая модели данных (Device, Project, Program), представления и сериализаторы. Для написания тестов использовались стандартные инструменты Django (django.test), которые позволяют быстро создавать изолированные тестовые случаи для проверки бизнес-логики и взаимодействия с базой данных.
- Интеграционное тестирование. Интеграционные тесты проверяли корректность взаимодействия между различными слоями приложения: моделями, представлениями, сериализаторами и API-интерфейсами. Проверялись, например, сценарии создания устройства через API, привязка устройства к проекту разработки, а также обновление информации через веб-формы.
- Ручное тестирование пользовательских интерфейсов. Клиентская часть приложения тестировалась вручную для проверки корректности отображения данных, функционирования форм, обработки ошибок пользователя и соответствия страниц требованиям адаптивной вёрстки. Тестировались все основные пользовательские сценарии: создание, редактирование, удаление объектов, фильтрация списков, поиск.
- Нагрузочное тестирование. Для проверки устойчивости приложения под нагрузкой были смоделированы сценарии массового обращения к API и к страницам списков устройств и проектов.

Проверялась скорость отклика сервера при одновременной обработке большого числа запросов и корректность обработки ошибок при перегрузке.

Выбор именно этих подходов к тестированию был обусловлен спецификой проекта:

- большая часть бизнес-логики реализована через серверные компоненты и API;
- важность корректной сериализации данных между серверной и клиентской частями;
- необходимость обеспечения стабильности и безопасности работы системы при реальной эксплуатации.

Тестирование охватывало критически важные разделы приложения.

- Модели устройств и проектов, поскольку от их целостности зависит правильная работа учётной системы.
- Представления списков, создания и редактирования сущностей.
- API-интерфейсы, обеспечивающие взаимодействие системы с внешними сервисами (например, интеграция с Jira).

В процессе тестирования применялись следующие инструменты:

- Django TestCase для написания юнит- и интеграционных тестов;
- Postman для ручного тестирования REST API-интерфейсов;
- стандартные средства браузеров и панели для ручной проверки интерфейсов.

Таким образом, построенная стратегия тестирования обеспечила комплексную проверку всех критически важных компонентов системы и позволила выявить и устранить потенциальные ошибки на ранних этапах разработки, что значительно повысило надёжность конечного продукта.

3.2 Юнит-тестирование и интеграционное тестирование

Тестирование отдельных компонентов веб-приложения и их взаимодействия играло важную роль в процессе обеспечения надёжности и корректности работы всей системы.

Юнит-тестирование в проекте осуществлялось для проверки работы отдельных моделей, сериализаторов и представлений. Основной задачей юнит-тестов было выявление ошибок в бизнес-логике на уровне отдельных классов и функций.

В рамках юнит-тестирования было разработано:

- более 80 тестовых методов для моделей, форм, сериализаторов и представлений;
- покрытие всех основных моделей проекта (Device, Project, Program, Stage) с проверкой создания, изменения и удаления объектов (рисунок 30);
- проверка валидации обязательных полей, корректности значений по умолчанию и работы связей между сущностями (ForeignKey, OneToOneField).

Примеры успешно проведённых юнит-тестов:

- тестирование создания устройства с минимальным набором полей;
- проверка, что при удалении проекта связанные устройства остаются без изменений или корректно обрабатываются через `on_delete=models.SET_NULL`;
- тест сериализатора устройства, проверяющий, что в API передаются только допустимые поля.

```

class DeviceModelTest(TestCase):
    def test_create_device(self):
        device = Device.objects.create(name='Test Device', code='test_code')
        self.assertEqual(device.name, 'Test Device')
        self.assertTrue(Device.objects.filter(code='test_code').exists())

```

Рисунок 30 Юнит-тест создания устройства

Успешно прошли 98% тестов на этапе юнит-тестирования. Оставшиеся 2% ошибок были связаны с обработкой граничных случаев в валидации сериализаторов и были оперативно исправлены.

Интеграционное тестирование проверяло корректность взаимодействия между различными компонентами приложения: моделями, сериализаторами, представлениями и API.

В рамках интеграционного тестирования (рисунок 31) протестированы основные пользовательские сценарии при помощи 185 тестов:

- создание устройства через API;
- получение списка устройств с фильтрацией и пагинацией;
- обновление атрибутов устройства через PATCH-запрос;
- проверка прав доступа к API для разных групп пользователей.
- проверялась структура JSON-ответов от сервера, наличие всех необходимых полей и корректность статусов HTTP-ответов (200 OK, 201 Created, 400 Bad Request, 403 Forbidden, 404 Not Found).

```

class DeviceAPITest(APITestCase):
    def test_list_devices(self):
        response = self.client.get('/api/devices/')
        self.assertEqual(response.status_code, 200)
        self.assertIn('results', response.data)

```

Рисунок 31 - Пример интеграционного теста API

Особое внимание уделялось следующим аспектам:

- корректная обработка ошибок при передаче некорректных данных (например, попытка создать устройство без обязательного поля name);
- поведение системы при отправке запросов неавторизованными пользователями;
- сохранение целостности данных при обновлении и удалении связанных объектов.

Результаты интеграционного тестирования:

- успешное прохождение **95%** тестов на первом этапе;
- после исправления выявленных ошибок стабильность достигла 100% при повторном запуске тестов.

Таким образом, проведённые юнит- и интеграционные тестирования подтвердили высокую надёжность всех основных компонентов системы, обеспечили уверенность в корректной работе API-интерфейсов и значительно повысили общее качество разрабатываемого веб-приложения.

3.3 Нагрузочное тестирование и анализ производительности

Для оценки устойчивости веб-приложения при высокой нагрузке и проверки его производительности были проведены мероприятия по нагрузочному тестированию.

Целью тестирования было выявление предельных возможностей системы, определение скорости отклика сервера и проверка стабильности работы при большом количестве одновременных запросов.

Нагрузочное тестирование проводилось с использованием ручных сценариев симуляции активности пользователей, а также с помощью специализированных инструментов (Apache Bench, браузерное тестирование в условиях множественных сессий).

Основные сценарии нагрузочного тестирования включали:

- массовое получение списка устройств через API (GET /api/devices/);
- параллельное создание новых записей устройств через API (POST /api/devices/);
- одновременное обновление существующих объектов;
- массовый просмотр списков проектов и программ на клиентской части.

Тестирование проводилось на тестовом сервере, идентичном продакшен-окружению:

- веб-приложение развёрнуто через связку uWSGI + Nginx;
- база данных - PostgreSQL;
- сервер работал в стандартной конфигурации с базовыми параметрами настройки.

В ходе тестирования были зафиксированы следующие результаты:

- время ответа сервера на одиночный запрос списка устройств составляло в среднем 200–300 мс при нормальной нагрузке;
- при увеличении количества одновременных запросов до 100 пользователей среднее время ответа возрастало до 500–700 мс, при этом не наблюдалось сбоев в работе сервера;
- при нагрузке свыше 150–200 одновременных запросов начинались рост времени отклика до 1–1.2 секунд, что остаётся в пределах допустимых значений для большинства корпоративных приложений.

Существенных ошибок работы сервера при нагрузке не было выявлено. Благодаря использованию механизмов оптимизации запросов Django ORM (select_related, prefetch_related) и корректной настройке базы данных PostgreSQL, система демонстрировала стабильную работу даже в условиях пиковых нагрузок.

Для анализа производительности также применялись встроенные инструменты мониторинга сервера:

- отслеживалась нагрузка на процессор и память;
- анализировались логи приложений на наличие ошибок;
- фиксировались коды ответов HTTP для выявления аномалий.

На основании полученных данных были сделаны выводы о необходимости:

- дальнейшей оптимизации наиболее тяжёлых запросов при росте количества пользователей;
- возможного внедрения кэширования часто запрашиваемых данных для ещё большего снижения нагрузки на сервер.

Таким образом, проведённое нагрузочное тестирование подтвердило готовность системы к эксплуатации в реальных условиях при стандартных и повышенных уровнях пользовательской активности.

3.4 Оценка безопасности веб-приложения

Безопасность веб-приложения являлась одним из приоритетных аспектов при проектировании и разработке системы. В процессе тестирования были проведены мероприятия по проверке устойчивости приложения к наиболее распространённым типам атак и уязвимостей.

Основными направлениями оценки безопасности стали.

- Проверка защиты от межсайтовых атак (CSRF). Для всех форм и отправляемых данных была реализована стандартная защита Django от атак типа Cross-Site Request Forgery (CSRF). В каждой HTML-форме автоматически вставляется скрытое поле с CSRF-токеном (`{% csrf_token %}`), который проверяется на стороне сервера. При попытке отправить

POST-запрос без действительного токена сервер корректно отклоняет запрос с кодом ошибки 403 [32].

- Защита от атак XSS (Cross-Site Scripting). Django по умолчанию экранирует все выводимые в шаблонах данные, что предотвращает внедрение вредоносных скриптов в HTML-страницы [32]. В ходе тестирования было подтверждено, что пользовательский ввод, содержащий специальные символы, корректно экранируется и не приводит к выполнению нежелательного кода на стороне клиента.
- Защита от SQL-инъекций. Использование Django ORM для всех операций с базой данных обеспечивает автоматическое экранирование параметров запросов [32]. При тестировании были предприняты попытки передачи вредоносных SQL-конструкций через поля ввода (например, 1'; DROP TABLE device_device;--), однако все они корректно обрабатывались без выполнения вредоносных действий.
- Проверка разграничения прав доступа. Система авторизации и разграничения прав доступа была протестирована на корректность. Проверялись сценарии: доступ к закрытым эндпоинтам без авторизации (должен быть запрещён); попытка выполнения административных действий пользователем без соответствующих прав; корректная обработка перенаправлений при отсутствии доступа.
- Безопасность интеграции с внешними сервисами. Особое внимание уделялось безопасности интеграции с системой Jira. Подключение к внешнему API происходило через защищённый HTTPS-протокол, а аутентификация осуществлялась с помощью API-токена, который не передавался в открытом виде и не хранился в коде.
- Обработка ошибок. Все ошибки обрабатываются с возвращением корректных HTTP-кодов и обобщённых сообщений пользователю без раскрытия внутренних подробностей работы сервера или структуры базы данных.

Кроме ручного тестирования безопасности, часть проверок производилась с использованием средств статического анализа кода и базовых сканеров уязвимостей, встроенных в средства разработки.

Проведённая оценка безопасности подтвердила высокую степень защищённости веб-приложения от типичных угроз и соответствие современным требованиям к безопасности веб-систем корпоративного уровня.

3.5 Развертывание приложения

После завершения этапов разработки и тестирования была произведена процедура развёртывания веб-приложения в реальной инфраструктуре компании. Целью развёртывания являлось обеспечение постоянной доступности приложения для конечных пользователей с высокой надёжностью, безопасностью и возможностью масштабирования при необходимости.

В качестве среды для развёртывания использовался выделенный сервер с операционной системой на базе Linux. На сервере были установлены и настроены следующие компоненты.

- uWSGI — в качестве WSGI-сервера для запуска Django-приложения.
- Nginx — в качестве обратного прокси-сервера и веб-сервера для обслуживания статических файлов и балансировки нагрузки.
- PostgreSQL — в качестве основной системы управления базами данных.
- Процесс развертывания включал несколько этапов:
- Настройка окружения. Создано отдельное виртуальное окружение Python для изоляции зависимостей проекта. Все необходимые библиотеки были установлены с использованием файла requirements.txt, в том числе такие пакеты, как Django, Django REST Framework, библиотека jira для интеграции и прочие.
- Настройка базы данных. Система управления базой данных PostgreSQL была настроена с отдельной базой данных для проекта, выделенным

пользователем и ограниченными правами доступа. Применены все необходимые миграции Django для создания структуры таблиц в базе данных.

- Конфигурация uWSGI. uWSGI был настроен для запуска проекта в режиме демона с автоперезапуском процессов в случае сбоев. Конфигурация включала: указание пути к виртуальному окружению; настройки рабочих процессов и потоков для обработки запросов; маршрутизацию запросов через Unix-сокеты для передачи между uWSGI и Nginx.
- Nginx настроен для: проксирования запросов от клиентов к uWSGI через сокет; обработки статических файлов (static/) и медиафайлов (media/) напрямую без участия Django; шифрования соединений через HTTPS с использованием SSL-сертификата.
- Развертывание и тестирование. После запуска сервиса были проведены тестовые подключения к приложению для проверки доступности страниц, работоспособности API и корректной обработки пользовательских запросов. Особое внимание уделялось проверке обработки ошибок и стабильности работы сервера под нагрузкой.

Для упрощения процесса обновления приложения была предусмотрена возможность деплоя новых версий проекта с использованием системы контроля версий Git и автоматического применения миграций после каждого обновления кода.

Таким образом, развертывание проекта было выполнено в соответствии с лучшими практиками для Django-приложений, что обеспечило надёжную работу веб-приложения в реальной инфраструктуре и позволило подготовить платформу для его дальнейшего масштабирования и развития.

3.6 Выводы по главе

Проведённые мероприятия по тестированию и внедрению веб-приложения подтвердили его готовность к эксплуатации в реальной инфраструктуре компании. Комплексный подход к тестированию, включающий юнит-тесты, интеграционные сценарии, ручное тестирование интерфейсов, нагрузочные испытания и оценку безопасности, позволил выявить и устранить потенциальные ошибки на ранних этапах разработки.

Юнит-тестирование и интеграционные тесты обеспечили надёжность функционирования основных компонентов приложения: моделей, сериализаторов, представлений и API-интерфейсов. Нагрузочные испытания подтвердили способность системы стабильно работать под высокой пользовательской активностью, а меры по обеспечению безопасности защитили приложение от типичных угроз, таких как CSRF, XSS и SQL-инъекции.

Развёртывание проекта было выполнено на выделенном сервере с использованием современных технологий, включая uWSGI, Nginx и PostgreSQL, что обеспечило высокую производительность, отказоустойчивость и безопасность работы системы.

Таким образом, по результатам проведённого тестирования и внедрения можно сделать вывод, что разработанное веб-приложение соответствует предъявляемым требованиям по надёжности, безопасности и удобству использования, и готово к промышленной эксплуатации в рамках корпоративной информационной инфраструктуры.

Заключение

В данной работе была проведено исследование, разработка и внедрение веб-приложения для централизованного учёта и мониторинга устройств, предназначенного для использования внутри корпоративной инфраструктуры.

В ходе исследования выполнен анализ современных технологий веб-разработки, сравнительный анализ архитектурных подходов, выбрана оптимальная архитектурная модель (монолитная архитектура), обоснован выбор стека технологий: Python, Django, PostgreSQL, JavaScript и Bootstrap.

Разработана структура базы данных, обеспечивающая хранение всех необходимых данных о устройствах, их версиях и проектах. Реализована серверная часть на Django с REST API и клиентская часть на основе шаблонов Bootstrap. Обеспечена интеграция с внешними корпоративными сервисами Jira и 1С ERP. Проведено комплексное тестирование системы: юнит-тестирование, интеграционное тестирование, нагрузочное тестирование и оценка безопасности. Разработанное приложение успешно развернуто в корпоративной среде с использованием uWSGI и Nginx.

В ходе моделирования предметной области были определены ключевые сущности, отражающие структуру учёта устройств связи: устройства, версии программного и аппаратного обеспечения, проекты и программы испытаний. Это позволило создать целостную и логичную систему управления данными, максимально соответствующую потребностям компании.

Все поставленные цели и задачи, определённые на этапе планирования работы, были успешно достигнуты. Разработано безопасное, надёжное и масштабируемое веб-приложение, обеспечивающее централизованный учёт устройств. Выбранный стек технологий позволил реализовать функционал в установленные сроки, а комплексное тестирование подтвердило корректность работы приложения и его готовность к эксплуатации в реальной корпоративной среде. Интеграция с внешними системами Jira позволила

обеспечить своевременное обновление данных и синхронизацию между различными подразделениями компании.

В перспективе возможно дальнейшее развитие проекта в следующих направлениях.

- Разделение серверной и клиентской частей с использованием современных frontend-фреймворков (например, Vue.js или React).
- Внедрение микросервисной архитектуры при увеличении масштабов проекта.
- Дальнейшая автоматизация CI/CD процессов для ускорения развертывания новых версий.
- Расширение функционала системы учёта с добавлением дополнительных аналитических и отчётных модулей.
- Расширение интеграций с другими корпоративными сервисами.
- Проведение более углублённого аудита безопасности и внедрение дополнительных механизмов защиты.

Разработанное решение обладает достаточным потенциалом для дальнейшего масштабирования и адаптации под новые требования бизнеса.

Список использованных источников

1. Фаулер М. Архитектура корпоративных приложений. — М.: Вильямс, 2022. — 560 с.
2. Stack Overflow Developer Survey 2024 [Электронный ресурс]. — Режим доступа: <https://survey.stackoverflow.co/2024> (дата обращения: 30.04.2025).
3. Сэм Ньюман. Микросервисы. Разработка и сопровождение. — СПб.: Питер, 2022. — 352 с.
4. Современные тренды разработки программного обеспечения: Agile, DevOps, CI/CD [Электронный ресурс]. — Режим доступа: <https://www.researchgate.net/publication/371620522> (дата обращения: 30.04.2025).
5. Vincent W. Django for Professionals. — Independently published, 2022. — 282 с.
6. PostgreSQL: Up and Running / Regina O. Obe, Leo S. Hsu. — O'Reilly Media, 2021. — 190 с.
7. Современные тренды разработки программного обеспечения: Agile, DevOps, CI/CD [Электронный ресурс]. — Режим доступа: <https://www.researchgate.net/publication/371620522> (дата обращения: 30.04.2025).
8. Innostage Group. Архитектура современных веб-сервисов и способы их защиты [Электронный ресурс]. — Режим доступа: <https://innostage-group.ru/press/blog/technical/architecture-of-modern-web-services/> (дата обращения: 01.05.2025).
9. Дэвис Дж., Дэниелс Р. Философия DevOps. Искусство управления ИТ. — М.: ДМК Пресс, 2021. — 320 с.
10. Вольф Э. Continuous Delivery. Практика непрерывных апдейтов. — СПб.: Питер, 2020. — 384 с.
11. Айелло Б., Сакс Л. Гибкое управление жизненным циклом приложений: использование DevOps для улучшения процессов. — М.: Вильямс, 2020. — 256 с.

12. Practicum by Yandex. Основные подходы к управлению проектами [Электронный ресурс]. – Режим доступа: <https://practicum.yandex.ru/> (дата обращения: 02.05.2025).
13. Practicum by Yandex. Классический метод — Waterfall [Электронный ресурс]. – Режим доступа: <https://practicum.yandex.ru/> (дата обращения: 02.05.2025).
14. Гибкое и эффективное тестирование качества с использованием Agile и DevOps Киберленинка. [Электронный ресурс]. — Режим доступа: <https://cyberleninka.ru/article/n/gibkoe-i-effektivnoe-testirovanie-kachestva-s-ispolzovaniem-agile-i-devops> (дата обращения: 01.05.2025).
15. Методология разработки CI/CD – что это такое GitVerse Blog. — [Электронный ресурс]. - Режим доступа: <https://gitverse.ru/blog/articles/development/537-metodologiya-razrabotki-cicd> (дата обращения: 01.05.2025).
16. История Agile, DevOps, CI/CD. Как менялся процесс разработки // Tproger. [Электронный ресурс]. — Режим доступа: <https://tproger.ru/articles/agile-devops-ci-cd-kak-menjalis-koncepcii-razrabotki> (дата обращения: 01.05.2025).
17. Басс Л., Клементс П., Казман Р. Архитектура программного обеспечения на практике. — СПб.: Питер, 2006. — 576 с.
18. Лащевски Т., Фарр Э., Арора К. Облачные архитектуры: разработка устойчивых и экономичных облачных приложений. — СПб.: Питер, 2022. — 320 с.
19. Django documentation. [Электронный ресурс]. — Режим доступа: <https://docs.djangoproject.com/en/5.2/> (дата обращения: 03.05.2025).
20. Django Book — русский перевод. [Электронный ресурс]. — Режим доступа: <https://rtfm.co.ua/ru/books-translations/django-book/> (дата обращения: 03.05.2025).

21. Архитектура в Django проектах — как выжить. [Электронный ресурс]. — Режим доступа: https://habr.com/ru/companies/vivid_money/articles/544856/ (дата обращения: 03.05.2025).
22. Django MVT Architecture - Tutorialspoint. [Электронный ресурс]. — Режим доступа: https://www.tutorialspoint.com/django/django_mvt.htm (дата обращения: 03.05.2025)
23. Basics of Django: Model-View-Template (MVT) Architecture. [Электронный ресурс]. — Режим доступа: <https://angelogentileiii.medium.com/basics-of-django-model-view-template-mvt-architecture-8585aecffbf6> (дата обращения: 03.05.2025).
24. PostgreSQL. Работа с транзакциями [Электронный ресурс]. — Режим доступа: <https://postgrespro.ru/docs/postgrespro/10/tutorial-transactions> (дата обращения: 03.05.2025).
25. uWSGI — что это такое [Электронный ресурс]. — Режим доступа: <https://www.dmosk.ru/terminus.php?object=uwsgi> (дата обращения: 03.05.2025).
26. Nginx — официальный сайт [Электронный ресурс]. — Режим доступа: <https://nginx.org/ru/> (дата обращения: 03.05.2025).
27. Официальная документация Django 5.2. [Электронный ресурс]. — Режим доступа: <https://docs.djangoproject.com/en/5.2/> (дата обращения: 03.05.2025).
28. Документация Django 3.2 на русском языке. [Электронный ресурс]. — Режим доступа: <https://djangodoc.ru/3.2/contents/> (дата обращения: 03.05.2025).
29. Официальная документация Django REST Framework. [Электронный ресурс]. — Режим доступа: <https://www.django-rest-framework.org/> (дата обращения: 03.05.2025).
30. Дронов В. А. Django 3.0. Практика создания веб-сайтов на Python. — СПб.: БХВ-Петербург, 2020. — 704 с.
31. Ларсен М. Безопасная веб-разработка. Принципы и практика. — М.: ДМК Пресс, 2021. — 384 с.

32. Django Security Documentation. [Электронный ресурс]. — Режим доступа: <https://docs.djangoproject.com/en/5.2/topics/security/>
33. Mozilla Developer Network (MDN). HTTP overview [Электронный ресурс]. — Режим доступа: <https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview>
34. Migrations - Django documentation. [Электронный ресурс]. — Режим доступа: <https://docs.djangoproject.com/en/5.2/topics/migrations/> (дата обращения: 03.05.2025).
35. Django Advanced: Use the Power of Postgres. — [Электронный ресурс]. — Режим доступа: <https://python.plainenglish.io/django-advanced-use-the-power-of-postgres-a472496f3140> (дата обращения: 03.05.2025).
36. django-filter — документация [Электронный ресурс]. — Режим доступа: <https://django-filter.readthedocs.io/> (дата обращения: 04.05.2025).
37. django-import-export — документация [Электронный ресурс]. — Режим доступа: <https://django-import-export.readthedocs.io/> (дата обращения: 04.05.2025).
38. django-crispy-forms — документация [Электронный ресурс]. — Режим доступа: <https://django-crispy-forms.readthedocs.io/> (дата обращения: 04.05.2025).
39. django-clone — документация [Электронный ресурс]. — Режим доступа: <https://pypi.org/project/django-clone/> (дата обращения: 04.05.2025).
40. django-ordered-model — документация [Электронный ресурс]. — Режим доступа: <https://github.com/django-ordered-model/django-ordered-model> (дата обращения: 04.05.2025).
41. Django Debug Toolbar — документация [Электронный ресурс]. — Режим доступа: <https://django-debug-toolbar.readthedocs.io/> (дата обращения: 04.05.2025).
42. Jira Python API — документация [Электронный ресурс]. — Режим доступа: <https://jira.readthedocs.io/> (дата обращения: 04.05.2025).

43. uWSGI — документация [Электронный ресурс]. — Режим доступа: <https://uwsgi-docs.readthedocs.io/> (дата обращения: 04.05.2025).
44. Foreign Keys On_Delete Option in Django Models [Электронный ресурс]. — Режим доступа: https://www.geeksforgeeks.org/foreign-keys-on_delete-option-in-django-models/ (дата обращения: 07.05.2025).
45. unique=True – Django Built-in Field Validation [Электронный ресурс]. — Режим доступа: <https://www.geeksforgeeks.org/unique-true-django-built-in-field-validation/> (дата обращения: 07.05.2025).
46. Many-to-many relationships — Django documentation [Электронный ресурс]. — Режим доступа: https://docs.djangoproject.com/en/5.2/topics/db/examples/many_to_many/ (дата обращения: 07.05.2025).
47. PostgreSQL: CREATE INDEX [Электронный ресурс]. — Режим доступа: <https://www.postgresql.org/docs/current/sql-createindex.html> (дата обращения: 07.05.2025).
48. Database Normalization Explained [Электронный ресурс]. — Режим доступа: <https://www.essentialsql.com/get-ready-to-learn-sql-database-normalization-explained-in-simple-english/> (дата обращения: 07.05.2025).
49. Индексы в PostgreSQL: создание, типы и оптимизация // GitVerse [Электронный ресурс]. — Режим доступа: <https://gitverse.ru/blog/articles/data/439-indeksy-v-postgresql> (дата обращения: 07.05.2025).
50. PostgreSQL: Using Foreign Keys [Электронный ресурс]. — Режим доступа: <https://www.postgresql.org/docs/current/ddl-constraints.html> (дата обращения: 07.05.2025).
51. Microsoft Docs: Database normalization basics [Электронный ресурс]. — Режим доступа: <https://learn.microsoft.com/en-us/office/troubleshoot/access/database-normalization-description> (дата обращения: 07.05.2025).

52. PostgreSQL Performance Tuning [Электронный ресурс]. — Режим доступа: <https://andreanahs.medium.com/performance-tuning-on-a-postgresql-c85088bd149> (дата обращения: 07.05.2025).
53. PostgreSQL Index Types Explained [Электронный ресурс]. — Режим доступа: <https://neon.com/postgresql/postgresql-indexes/postgresql-index-types> (дата обращения: 07.05.2025).
54. Django Models and Migrations — Django documentation [Электронный ресурс]. — Режим доступа: <https://docs.djangoproject.com/en/5.2/topics/db/models/> (дата обращения: 07.05.2025).
55. Django ORM Overview — RealPython [Электронный ресурс]. — Режим доступа: <https://realpython.com/django-orm-basics/> (дата обращения: 07.05.2025).
56. PostgreSQL for Django — PostgreSQL Official Docs [Электронный ресурс]. — Режим доступа: <https://www.postgresql.org/docs/current/django.html> (дата обращения: 07.05.2025).
57. Database Design basics. [Электронный ресурс]. — Режим доступа: <https://support.microsoft.com/en-us/office/database-design-basics-eb2159cf-1e30-401a-8084-bd4f9c9ca1f5> (дата обращения: 07.05.2025).
58. Django Architecture Overview — Official Documentation [Электронный ресурс]. — Режим доступа: <https://docs.djangoproject.com/en/5.2/misc/design-philosophies/> (дата обращения: 07.05.2025).
59. Django Class-Based Views — Django Documentation [Электронный ресурс]. — Режим доступа: <https://docs.djangoproject.com/en/5.2/topics/class-based-views/> (дата обращения: 07.05.2025).
60. Django REST Framework: Permissions and Security [Электронный ресурс]. — Режим доступа: <https://www.django-rest-framework.org/api-guide/permissions/> (дата обращения: 07.05.2025).
61. OWASP Cheat Sheet Series: Django Security [Электронный ресурс]. — Режим доступа:

https://cheatsheetseries.owasp.org/cheatsheets/Django_Security_Cheat_Sheet.html
(дата обращения: 07.05.2025).

62. Django Template Language — Django documentation [Электронный ресурс]. — Режим доступа: <https://docs.djangoproject.com/en/5.2/ref/templates/language/> (дата обращения: 08.05.2025).

63. Bootstrap Documentation — Layout, Grid System, Components [Электронный ресурс]. — Режим доступа: <https://getbootstrap.com/docs/5.3/getting-started/introduction/> (дата обращения: 08.05.2025).

64. A list of design concepts every UX/UI designer should learn [Электронный ресурс]. — Режим доступа: <https://uxdesign.cc/a-list-of-design-concepts-every-ux-ui-designer-should-learn-7e2d8412b391> (дата обращения: 08.05.2025).

65. Интеграция Ajax с Django: использование HttpResponse и JSON [Электронный ресурс]. — Режим доступа: <https://sky.pro/wiki/python/integratsiya-ajax-s-django-ispolzovanie-http-response-i-json/> (дата обращения: 08.05.2025).

66. Django Static Files — Django documentation [Электронный ресурс]. — Режим доступа: <https://docs.djangoproject.com/en/5.2/howto/static-files/> (дата обращения: 08.05.2025).

67. Bootstrap Icons Documentation [Электронный ресурс]. — Режим доступа: <https://icons.getbootstrap.com/> (дата обращения: 08.05.2025).

68. Django Forms — Django documentation [Электронный ресурс]. — Режим доступа: <https://docs.djangoproject.com/en/5.2/topics/forms/> (дата обращения: 08.05.2025).

69. Django ModelForm — Django documentation [Электронный ресурс]. — Режим доступа: <https://docs.djangoproject.com/en/5.2/topics/forms/modelforms/> (дата обращения: 08.05.2025).

70. django-crispy-forms — официальная документация [Электронный ресурс]. — Режим доступа: <https://django-crispy-forms.readthedocs.io/> (дата обращения: 08.05.2025).
71. API Integration Patterns – The Difference between REST, RPC, GraphQL, Polling, WebSockets and WebHooks [Электронный ресурс]. — Режим доступа: <https://www.freecodecamp.org/news/api-integration-patterns/> (дата обращения: 08.05.2025).
72. Django Messages Framework — Django documentation [Электронный ресурс]. — Режим доступа: <https://docs.djangoproject.com/en/5.2/ref/contrib/messages/> (дата обращения: 08.05.2025).
73. Responsive Web Design Basics — Google Web Fundamentals [Электронный ресурс]. — Режим доступа: <https://web.dev/responsive-web-design-basics/> (дата обращения: 08.05.2025).