Министерство науки и высшего образования Российской Федерации
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

Факультет безопасности информационных технологий

Дисциплина:

«Программно-аппаратные средства защиты информации»

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ №4

«Разработка и реализация механизмов защиты fullstack веб-приложения»

Выполнили:

Суханкулиев Мухаммет,

студент группы N3346

_____
(подпись)

Бардышев Артём Антонович,

студент группы N3346

_____
(подпись)

Проверил:

Чешев Никита Игоревич

_____
(отметка о выполнении)

_____
(подпись)

Санкт-Петербург

2025 г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Цель работы – собрать полноценный минимальный продукт (MVP):

Frontend (React) → Backend (Python/FastAPI/Flask) → DB (PostgreSQL).

Регистрация пользователя: логин + пароль → создание записи в БД → хэш пароля, логирование события, возврат статуса на UI. Всё упаковано в Docker и поднимается docker compose up.

Для достижения поставленной цели были решены следующие задачи:

- спроектирована архитектура многокомпонентного веб-приложения;
- разработан Backend на FastAPI, реализующий API для регистрации и аутентификации;
- разработан Frontend на React, предоставляющий пользовательский интерфейс для взаимодействия с API;
- настроена база данных PostgreSQL с использованием миграций Alembic для управления схемой;
- реализованы ключевые механизмы защиты информации, включая безопасное хранение паролей, валидацию данных и защиту от дублирования учетных записей;
- обеспечена полная контейнеризация приложения с помощью Docker и Docker Compose для изоляции и простоты развертывания;
- проведено тестирование реализованного функционала и механизмов защиты.

1 АРХИТЕКТУРА И ТЕХНОЛОГИЧЕСКИЙ СТЕК

Разработанное приложение построено на основе микросервисной архитектуры и состоит из трех независимых, но взаимодействующих компонентов, каждый из которых работает в собственном Docker-контейнере:

1. Frontend разработан на React 18 с использованием Vite. Отвечает за пользовательский интерфейс, клиентскую валидацию и взаимодействие с Backend через REST API. Для продакшн-сборки используется веб-сервер Nginx, который обслуживает статические файлы и проксирует запросы к API;

2. Backend разработан на Python 3.12 с использованием фреймворка FastAPI. Реализует бизнес-логику, API-эндпоинты, серверную валидацию данных (Pydantic), аутентификацию и взаимодействие с базой данных;

3. Используется система управления базами данных PostgreSQL 16. Для взаимодействия с БД на Backend применяется асинхронная библиотека SQLAlchemy, а для управления версиями схемы базы данных – инструмент миграций Alembic.

Все компоненты объединены и управляются с помощью Docker-compose, что обеспечивает их согласованный запуск и взаимодействие в изолированной сети.

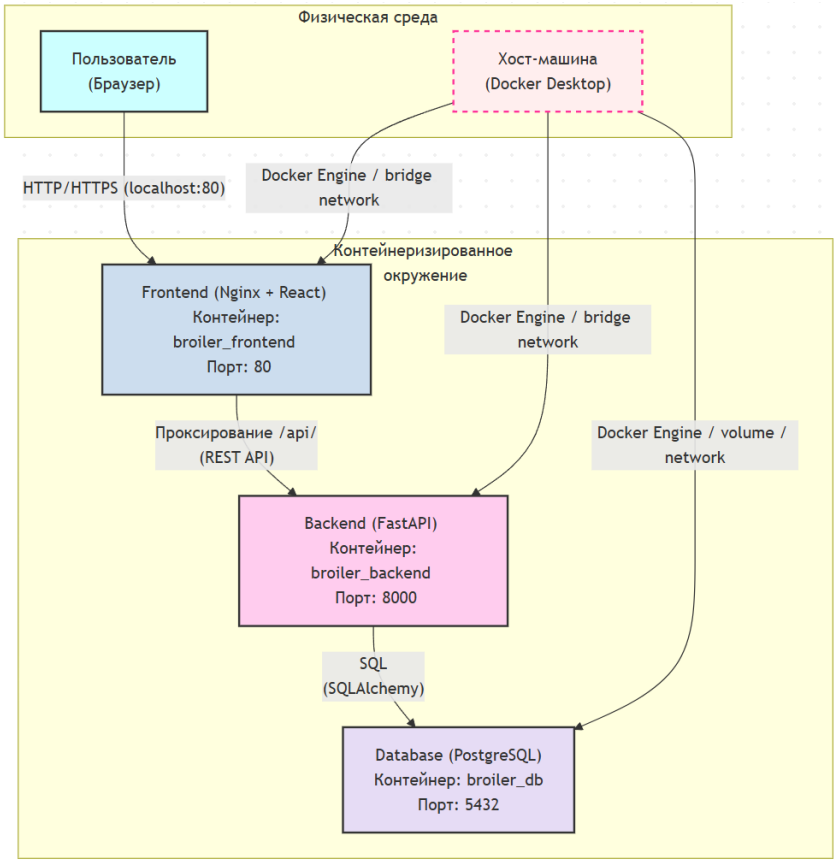

Рисунок 1 – Схема архитектуры приложения

2 МОДЕЛЬ УГРОЗ И ПРИНЯТЫЕ МЕРЫ ЗАЩИТЫ

В рамках лабораторной работы была составлена модель угроз, ориентированная на базовые уязвимости веб-приложений, и реализованы соответствующие контрмеры.

Таблица 1 – Модель угроз и реализованные меры защиты

Категория риска (OWASP Top 10:2021)	**Угроза / Уязвимость в контексте проекта**	**Реализованная мера защиты**
A01:2021 – Нарушение контроля доступа	Возможность создания нескольких учетных записей с одинаковым идентификатором (логином), что приводит к нарушению целостности данных и потенциальным проблемам с авторизацией.	Обеспечение уникальности логина на уровне СУБД. На поле `login` в таблице `users` наложено ограничение `UNIQUE`. Любая попытка вставить дублирующую запись будет отклонена базой данных, что является самым надежным уровнем защиты.
A03:2021 – Инъекции (Injection)	Отправка клиентом вредоносных данных в полях формы регистрации (например, SQL-кода), которые могут быть исполнены на сервере или в базе данных.	Строгая валидация и типизация входных данных. Библиотека `Pydantic` на стороне Backend'a проверяет все входящие данные на соответствие заранее определенной схеме (длина, тип, разрешенные символы), отсекая любые невалидные запросы до их обработки.

A05:2021 – Небезопасная конфигурация	Хранение секретов (ключей для JWT, паролей от БД) в исходном коде, что приводит к их утечке при публикации кода в системе контроля версий.	Использование переменных окружения (`.env`). Вся конфигурация, включая секреты, вынесена в `.env`-файл, который исключен из `Git`. Приложение читает эти переменные при старте, что соответствует принципу разделения кода и конфигурации.
A07:2021 – Идентификация и аутентификация	Компрометация учетных данных пользователей из-за хранения паролей в открытом виде или с использованием слабых, устаревших алгоритмов хэширования (например, MD5).	Использование криптостойкого хэширования `Argon2id`. Пароли хешируются с уникальной солью перед сохранением в БД. Алгоритм устойчив к атакам перебора на GPU, что соответствует современным требованиям безопасности.
A09:2021 – Недостатки журналирования и мониторинга	Отсутствие записей о ключевых событиях безопасности (регистрация, попытки входа) или, наоборот, логирование чувствительной информации (паролей), что создает риски и усложняет анализ инцидентов.	Безопасное структурированное логирование событий. С помощью библиотеки `structlog` фиксируются факты успешной регистрации и попыток аутентификации, но в логи никогда не попадают пароли или другие секреты в открытом виде.

3 ХОД РАБОТЫ И РАСПРЕДЕЛЕНИЕ ЗАДАЧ

Работа над проектом велась совместно с использованием системы контроля версий Git (GitHub) и стратегии ветвления Git Flow. Задачи были распределены поровну, однако для достижения лучшего результата практиковался подход кросс-функционального взаимодействия.

- Бардышев Артём (Frontend & Backend): Основная зона ответственности включала полную разработку клиентского приложения на React, включая пользовательский интерфейс, стилизацию, клиентскую валидацию и интеграцию с API. Также внес вклад в разработку Backend, реализовав эндпоинты API.
- Суханкулиев Мухаммет (Backend & DevOps): Основная зона ответственности включала проектирование и реализацию серверной части на FastAPI, создание архитектуры (сервисы, репозитории), настройку базы данных и миграций Alembic, а также полную контейнеризацию проекта с помощью Docker и Docker-compose. Также отвечал за набор автоматических тестов на Pytest и слияние веток.

Несмотря на разделение, оба участника проводили взаимное ревью кода и вносили правки в смежные части проекта для обеспечения лучшей интеграции компонентов.

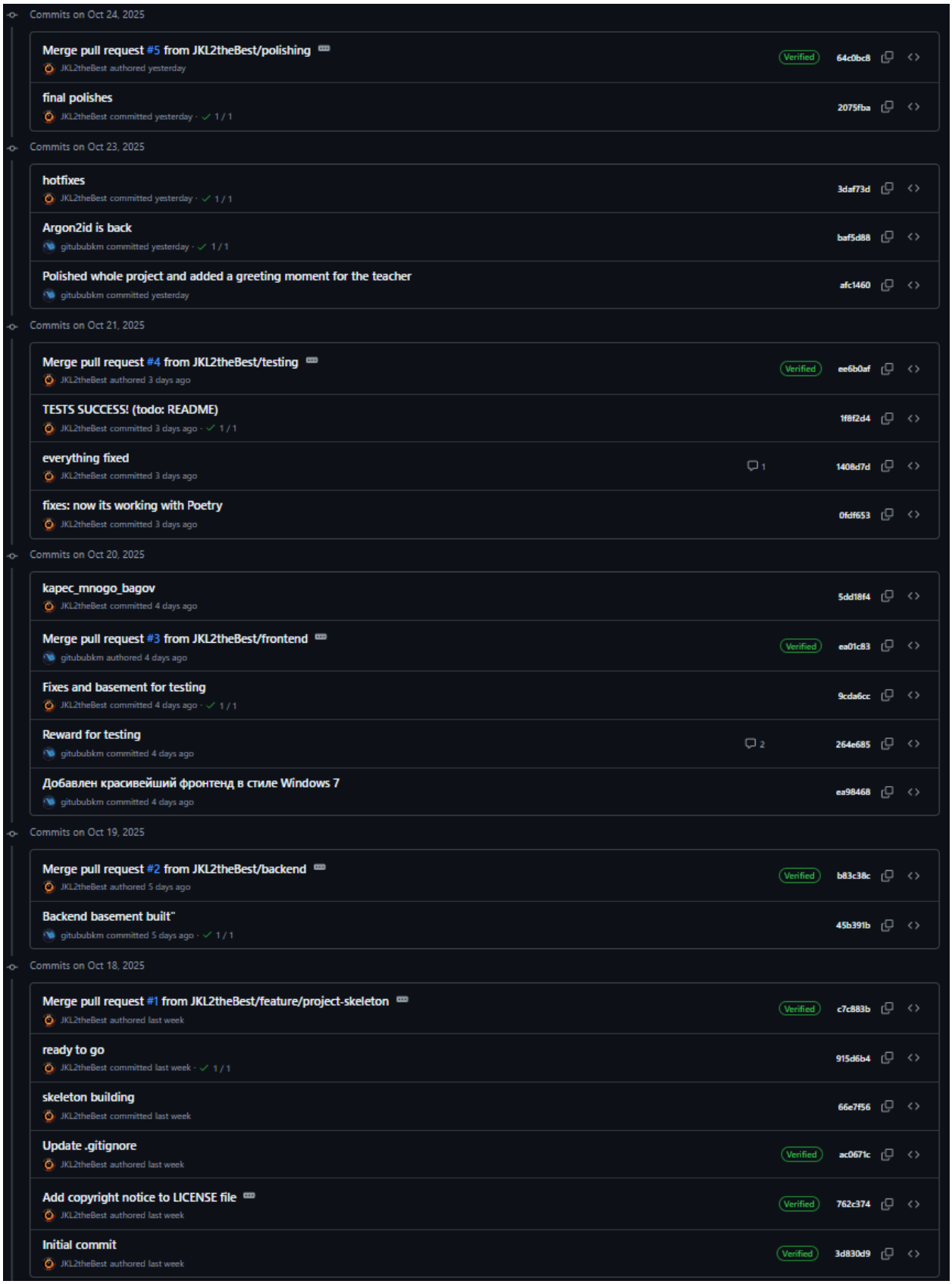

Рисунок 2 – История коммитов и веток в Git-репозитории

4 РЕАЛИЗАЦИЯ МЕХАНИЗМОВ ЗАЩИТЫ

Ниже приведены конкретные примеры реализации описанных в модели угроз мер защиты, которые в совокупности создают многоуровневую систему безопасности приложения.

4.1 Хэширование паролей (Argon2id)

Для предотвращения компрометации учетных данных в случае утечки базы данных пароли никогда не хранятся в открытом виде. Используется криптостойкий алгоритм `Argon2id`, реализованный через библиотеку `passlib`. Параметры криптостойкости (`time_cost`, `memory_cost`, `parallelism`) являются настраиваемыми и загружаются из файла окружения `.env`, что позволяет адаптировать их под мощность сервера.

Листинг 1 – backend/app/core/security.py

```
from datetime import datetime, timedelta, timezone

from jose import jwt
from passlib.context import CryptContext

from app.core.config import settings

# Создаем контекст Passlib для хеширования паролей.
pwd_context = CryptContext(
    schemes=[settings.HASH_SCHEME],
    deprecated="auto",
    # Настройки для Argon2, загруженные из .env файла.
    # time_cost: количество итераций (защита от перебора).
    # memory_cost: объем памяти (в KiB), необходимый для вычисления (защита
      от ASIC/GPU).
    # parallelism: количество параллельных потоков.
    argon2__time_cost=settings.ARGON2_TIME_COST,
    argon2__memory_cost=settings.ARGON2_MEMORY_COST,
    argon2__parallelism=settings.ARGON2_PARALLELISM,
)


def hash_password(password: str) -> str:
    """Хеширует пароль с использованием настроенного алгоритма (Argon2id)."""
    return pwd_context.hash(password)


def verify_password(plain_password: str, hashed_password: str) -> bool:
    """
    Проверяет, соответствует ли открытый пароль хешированному.
    """
    return pwd_context.verify(plain_password, hashed_password)


def create_access_token(data: dict) -> str:
    """
```

```
    Создает JWT access токен.
    """
    to_encode = data.copy()
    # Получаем текущее время один раз
    now = datetime.now(timezone.utc)
    expire = now + timedelta(minutes=settings.ACCESS_TOKEN_EXPIRE_MINUTES)

    # Явно устанавливаем ОБА временных поля: 'iat' и 'exp'
    to_encode.update({"iat": int(now.timestamp()), "exp":
      int(expire.timestamp())})

    return jwt.encode(to_encode, settings.SECRET_KEY,
      algorithm=settings.ALGORITHM)
```

4.2 Валидация данных (Pydantic)

Для защиты от инъекций и приема некорректных данных все запросы к API проходят строгую валидацию на стороне сервера с помощью Pydantic. В схеме UserCreateRequest определены четкие правила для логина и пароля, включая длину, разрешенные символы и сложность.

Листинг 2 – backend/app/schemas/user.py – класс: UserCreateRequest

```
class UserCreateRequest(UserBase):
    """Схема для валидации данных при регистрации пользователя (входные
      данные от клиента)."""

    password: str = Field(min_length=8, description="Пароль пользователя")

    @field_validator("login")
    @classmethod
    def validate_login(cls, value: str) -> str:
        """
        Валидатор логина. Разрешает только безопасные символы.
        """
        if not re.match(r"^[a-zA-Z0-9_.-]+$", value):
            raise ValueError(
                "Login must contain only latin letters, numbers, and symbols:
      . _ -"
            )
        return value

    @field_validator("password")
    @classmethod
    def validate_password(cls, value: str) -> str:
        """
        Валидатор сложности пароля.
        """
        if not re.search(r"[A-Z]", value):
            raise ValueError("Password must contain at least one uppercase
      letter")
        if not re.search(r"[a-z]", value):
            raise ValueError("Password must contain at least one lowercase
      letter")
```

```
        if not re.search(r"\d", value):
            raise ValueError("Password must contain at least one digit")
        if not re.search(r"[!@#$%^&*(),.?:{}|<>]", value):
            raise ValueError("Password must contain at least one special
      character")
        return value
```

4.3 Защита от дублирования учетных записей на уровне СУБД

Для обеспечения целостности данных и предотвращения создания нескольких пользователей с одинаковым логином, на уровне схемы базы данных для поля `login` установлено ограничение `UNIQUE`. Это наиболее надежный способ защиты, так как он не зависит от кода приложения.

Листинг 3 – backend/app/db/models.py – модель: User

```
class User(Base):
    """Модель SQLAlchemy для таблицы 'users'."""

    __tablename__ = "users"

    id: uuid.UUID = Column(UUID(as_uuid=True), primary_key=True,
      default=uuid.uuid4)

    login: str = Column(String(32), unique=True, index=True, nullable=False)

    password_hash: str = Column(String, nullable=False)

    created_at: datetime = Column(
        DateTime(timezone=True), server_default=func.now(), nullable=False
    )

    def __repr__(self):
        """Строковое представление объекта для отладки."""
        return f"<User(login='{self.login}')>"
```

4.4 Безопасное логирование и обработка ошибок

Для мониторинга событий безопасности и затруднения атак на перечисление пользователей реализованы следующие механизмы:

1. Структурированное логирование: Фиксируются важные события, но никогда не логируются пароли или токены.

2. Общие сообщения об ошибках: При неудачной аутентификации пользователю возвращается общее сообщение, не уточняющее, что именно было введено неверно – логин или пароль.

Листинг 4 – backend/app/services/user_service.py – функция: authenticate_user

```
    async def authenticate_user(self, login: str, password: str) -> dict:
        """Аутентифицирует пользователя и возвращает токен."""
        user = await self.repo.get_by_login(login)

        # Безопасность: Проверка пользователя и пароля выполняется в одной
      условной
        # конструкции. Это усложняет атаки по времени (timing attacks).
        if not user or not verify_password(password, user.password_hash):
            log.warn("authentication_failed", reason="invalid_credentials",
      login=login)
            # Безопасность: Сообщение об ошибке должно быть общим, чтобы не
      дать
            # злоумышленнику понять, что именно неверно: логин или пароль.
            raise HTTPException(
                status_code=status.HTTP_401_UNAUTHORIZED,
                detail="Invalid login or password",
                headers={"WWW-Authenticate": "Bearer"},
            )

        log.info("user_authenticated", user_id=str(user.id),
      login=user.login)

        # Создаем JWT токен с информацией о пользователе.
        access_token = create_access_token(
            data={"sub": user.login, "user_id": str(user.id)}
        )
        return {"access_token": access_token, "token_type": "bearer"}
```

4.5 Безопасная конфигурация и управление секретами

Все чувствительные данные (пароли от БД, секретные ключи) вынесены из кода в `.env`-файл. `Docker-compose` передает эти переменные в контейнеры при запуске, что соответствует принципу разделения кода и конфигурации и предотвращает утечку секретов в `Git`.

Листинг 5 – docker-compose.yml – backend

```
  backend:
    build: ./backend
    container_name: broiler_backend
    env_file: .env
    ports: ["${BACKEND_PORT}:${BACKEND_PORT}"]
    volumes:
      - ./backend:/app
    depends_on:
      db: { condition: service_healthy }
    networks: [broiler_network]
    restart: unless-stopped
```

4.6 Усиление защиты на уровне контейнеризации и веб-сервера

Для минимизации поверхности атаки и защиты от распространенных веб-уязвимостей применены следующие практики DevSecOps:

1. frontend/Dockerfile использует два этапа: На первом этапе устанавливаются все зависимости и собирается React-приложение. На втором, финальном этапе в чистый образ Nginx копируются только готовые статические файлы, без исходного кода и инструментов сборки.

2. Веб-сервер Nginx настроен на отправку HTTP-заголовков безопасности, которые защищают пользователя на стороне браузера от атак типа Clickjacking (X-Frame-Options) и некоторых видов XSS (X-XSS-Protection).

3. В конфигурации Nginx отключена отправка версии сервера (server_tokens off), что усложняет злоумышленнику поиск уязвимостей для конкретной версии ПО.

Листинг 6 – frontend/nginx.conf – Заголовки безопасности

```
server {
    ...
    # Скрываем версию nginx
    server_tokens off;

    # Настройки безопасности
    add_header X-Frame-Options "SAMEORIGIN" always;
    add_header X-XSS-Protection "1; mode=block" always;
    add_header X-Content-Type-Options "nosniff" always;
    add_header Referrer-Policy "no-referrer-when-downgrade" always;
    add_header Content-Security-Policy "default-src 'self' http: https: data:
      blob: 'unsafe-inline'" always;
}
```

5 РАЗВЕРТЫВАНИЕ И ТЕСТИРОВАНИЕ

Приложение полностью готово к развертыванию с помощью Docker. Файл `docker-compose.yml` описывает все сервисы и их взаимодействие. Для запуска в production-режиме используется команда `docker compose up --build -d`.

Тестирование Backend проводилось с помощью набора из 13 автоматических тестов на `Pytest`, которые обеспечивают комплексное покрытие основных сценариев регистрации и аутентификации. Для запуска тестов в изолированном окружении используется команда `docker compose run --rm backend python -m pytest`.

Ключевые проверенные сценарии:

- Успешная регистрация нового пользователя (ожидается статус 201).
- Защита от дубликатов: Попытка регистрации с существующим логином (ожидается статус 409).
- Валидация входных данных: Проверено 8 случаев регистрации с невалидными данными, включая слишком короткий/длинный логин, использование недопустимых символов, а также пароли, не соответствующие политике сложности (отсутствие заглавной/строчной буквы, цифры или спецсимвола). Во всех случаях ожидается статус 422.
- Успешная аутентификация: Проверка входа с корректными данными, валидация полученного JWT-токена и его содержимого (ожидается статус 200).
- Неуспешная аутентификация: Проверено 2 случая попытки входа с неверными данными (несуществующий логин и неверный пароль) (ожидается статус 401).

Все 13 тестов успешно пройдены, что подтверждает корректность работы реализованной бизнес-логики и механизмов защиты на серверной стороне.

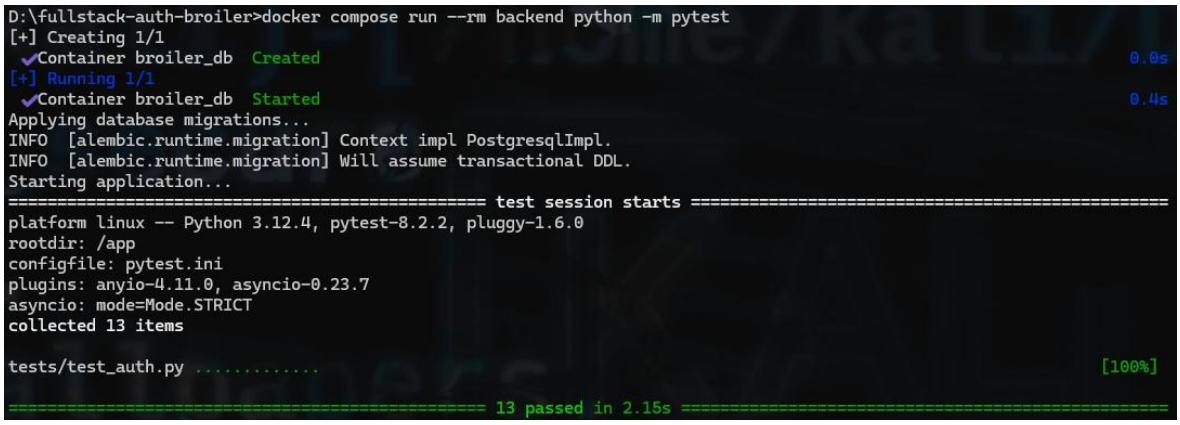

Рисунок 3 – Результаты выполнения автоматических тестов бэкенда

ЗАКЛЮЧЕНИЕ

В ходе выполнения лабораторной работы была успешно достигнута поставленная цель: разработано и развернуто полноценное fullstack MVP-приложение с упором на реализацию базовых программно-аппаратных средств защиты.

Были изучены и применены на практике ключевые принципы защиты веб-приложений: спроектирована безопасная архитектура, реализовано криптостойкое хэширование паролей с помощью `Argon2id`, настроена строгая валидация всех входных данных, обеспечена защита от дублирования учетных записей на уровне СУБД, а также внедрено безопасное логирование.

Использование Docker и Docker-compose позволило создать изолированную, воспроизводимую и готовую к развертыванию среду, что является современным стандартом в разработке. Совместная работа с использованием Git Flow продемонстрировала эффективность командного подхода к разработке. В результате был получен работающий продукт, полностью соответствующий техническому заданию.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. OWASP Top 10:2021 / The Open Web Application Security Project. – 2021. – Текст : электронный. – URL: https://owasp.org/Top10/ (дата обращения: 24.10.2025).
2. FastAPI Documentation / Tiangolo. – 2025. – Текст : электронный. – URL: https://fastapi.tiangolo.com/ (дата обращения: 24.10.2025).
3. Docker Documentation / Docker Inc. – 2025. – Текст : электронный. – URL: https://docs.docker.com/ (дата обращения: 24.10.2025).
4. Лазутин А. И. Курс лекций по дисциплине «Программно-аппаратные средства защиты информации» / А. И. Лазутин, Армавир, 2023. – Текст: непосредственный.