МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

«САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИМЕНИ Н. Г. ЧЕРНЫШЕВСКОГО»

УТВЕРЖДАЮ
Зав.кафедрой,
доцент, к. ф.-м. н.
_____ С. В. Миронов

ОТЧЕТ О ПРАКТИКЕ

студента 3 курса 351 группы факультета КНиИТ
Устюшина Богдана Антоновича

вид практики: производственная
кафедра: математической кибернетики и компьютерных наук
курс: 3
семестр: 6
продолжительность: 4 нед., с 01.08.2024 г. по 28.08.2024 г.

Руководитель практики от университета,

ст. преп., к. ф.-м. н. _____ М. И. Сафрончик

Руководитель практики от организации (учреждения, предприятия),

доцент, к. ф.-м. н. _____ С. В. Миронов

Тема практики:«Курс разработки Backend Web-приложения с DevOps принципами»

ВВЕДЕНИЕ

Практика проходила на базе предприятия ООО «ПрофСофт» и заключалась в прохождении курса по Backend- и DevOps-разработке.

Основным направлением деятельности компании является разработка и поддержка цифровых продуктов, мобильных, web-приложений, сервисов и готовых решений для бизнеса и людей. Также компания ежегодно проводит школу, в рамках которой на каждом из направлений на практике объясняются основы и лучшие практики разработки IT-продуктов.

В рамках курса велась над учебным проектом «Анекдоты», в рамках которого показывались лучшие практики в разработке серверной части приложений, а также техники по поддержке и настройке CI/CD на проекте.

В процессе разработки использовались язык программирования PHP и объектно-реляционная система управления базами данных PostgreSQL. Также в рамках работы над DevOps задачами активно использовалась YAML-нотация для написания настроечных файлов для pipeline на GitLab и написания выполняемых в рамках автоматизации работ.

Целью практики была разработка backend-части приложения.

В рамках производственной практики должны были быть решены следующие задачи:

1. Построение схемы базы данных
2. Создание базовых CRUD для backend-приложения
3. Создание авторизации на основе концепции access- и refresh-токенов
4. Создание docker-compose файлов для настройки контейнеризации и автоматизации запуска приложения

В процессе выполнения данных задач будет рассмотрено использование лучших практик разработки веб-приложений, а также современные подходы к организации рабочего процесса с акцентом на безопасность и масштабируемость приложения.

1 Описание проекта

Современная веб-разработка является одной из самых динамично развивающихся областей информационных технологий. С каждым годом всё большее количество компаний и организаций переносит свои сервисы и системы в онлайн-среду, что требует создания надежных, масштабируемых и безопасных веб-приложений. Особое внимание уделяется разработке серверной части (Backend), которая обеспечивает связь с базами данных, обработку запросов пользователей, а также взаимодействие с различными внешними системами.

Рассмотрим технологии, которые активно использовались в процессе прохождения курсов озвученной выше школы.

Веб-приложения, использующие язык программирования PHP и фреймворк Symfony, предоставляют широкие возможности для создания мощных и гибких систем, особенно в сочетании с такими современными технологиями, как контейнеризация с Docker и использование веб-сервера Nginx. Это делает тему разработки серверной части веб-приложений крайне актуальной в контексте повышения производительности, гибкости и безопасности веб-систем.

В данной работе будет рассматриваться процесс разработки серверной части веб-приложения для обмена анекдотами на базе PHP и фреймворка Symfony.

В процессе работы будет рассмотрено, как с помощью Docker создать контейнеризированное окружение для разработки и развертывания приложения, как настроить веб-сервер Nginx для обработки запросов, а также как тестировать API с помощью Postman и документировать его с использованием Swagger. Также будут освещены основные принципы работы с архитектурой MVC на Symfony и взаимодействие с базами данных через ORM Doctrine. Особое внимание будет уделено выбору и настройке средств разработки, таких как PHPStorm, а также сравнению PHP с другими популярными языками для создания Backend-приложений.

Важной частью работы станет анализ производительности и безопасности разрабатываемого приложения в контейнеризированной среде Docker, а также его автоматизация с помощью Gitlab pipelines.

Целью производственной практики является приобретение навыков разработки серверной части веб-приложений с использованием современных

инструментов и технологий, таких как PHP, Symfony, Docker, Nginx, Postman и Swagger [1, 2].

Практика направлена на углубленное изучение архитектуры веб-приложений, принципов работы серверной части и интеграции различных инструментов для разработки, тестирования и развертывания приложений [3–5].

Кроме того, значительное внимание будет уделено изучению особенностей контейнеризации и управления окружением разработки с помощью Docker, что позволит улучшить навыки работы с современными DevOps-технологиями.

2 Выполненные в рамках проекта задачи

2.1 Построение схемы базы данных

2.1.1 Постановка задачи

Построить схему базы данных для данной предметной области (анекдоты) и представить её в виде ER-диаграммы.

2.1.2 Особенности СУБД PostgreSQL

PostgreSQL (или просто Postgres) — это объектно-реляционная система управления базами данных (СУБД), которая используется для хранения, управления и обработки структурированных данных. Она обладает открытым исходным кодом и распространяется под лицензией PostgreSQL License, что делает её доступной для использования в коммерческих и некоммерческих проектах.

Основные характеристики PostgreSQL:

1. Объектно-реляционная модель. PostgreSQL сочетает в себе возможности реляционных баз данных с объектно-ориентированными функциями, такими как пользовательские типы данных, наследование таблиц и поддержка сложных структур данных.
2. Расширяемость. PostgreSQL предоставляет гибкие возможности для расширения функционала. Пользователи могут добавлять свои типы данных, функции, агрегаты, операторы, индексы и даже языки для написания хранимых процедур.
3. Мощный SQL-движок. PostgreSQL поддерживает стандарт SQL в сочетании с расширениями, такими как оконные функции, CTE (общие табличные выражения) и JSON, что делает её подходящей для сложных аналитических запросов.
4. Масштабируемость и производительность. PostgreSQL поддерживает горизонтальное масштабирование с использованием репликации и шардирования, а также вертикальное масштабирование за счёт оптимизированного использования ресурсов процессора и памяти.
5. Надёжность и отказоустойчивость. PostgreSQL обеспечивает надёжное хранение данных с помощью транзакций, поддерживающих свойства ACID (атомарность, согласованность, изоляция, долговечность). Также поддерживается механизм WAL (журнал записи), который гарантирует

восстановление данных в случае сбоя.

6. Поддержка различных типов данных: PostgreSQL поддерживает не только стандартные типы данных (числа, строки, даты), но и географические данные через расширение PostGIS, JSON/JSONB для работы с неструктурированными данными и массивы.

Архитектура PostgreSQL построена на многоуровневой модели. Основными компонентами являются:

1. Сервер базы данных: Отвечает за управление соединениями, выполнение запросов и управление данными.
2. Ядро SQL: Обеспечивает обработку SQL-запросов, включая синтаксический анализ, оптимизацию и выполнение.
3. Хранилище данных: Реализует физическое хранение данных в виде файлов на диске.
4. Механизм транзакций: Гарантирует согласованность данных и поддерживает механизм блокировок для параллельной работы.

2.1.3 Решение

Для начала опишем структуру таблиц, которые использовались для построения изображения, а затем предоставим SQL-запросы для их создания [6].

Структура таблиц:

— Анекдот — сущность использовалась для хранения информации об анекдоте. Сущность включала следующие поля:
    1. id (Primary Key): идентификатор.
    2. title: заголовок.
    3. text: текст анекдота.
    4. category: категория.
    5. author_id (Foreign Key): ссылка на пользователя.
— User — сущность, использующаяся для хранения информации о пользователе. Сущность включала следующие поля:
    1. id (Primary Key): идентификатор пользователя.
    2. surname: фамилия.
    3. name: имя.
    4. patronym: отчество.
    5. email: электронная почта.

— Mark — сущность, использующаяся для хранения информации об оценке, которую пользователь мог поставить анекдоту. Сущность включала следующие поля:
    1. user_id (Primary Key, Foreign Key): ссылка на пользователя.
    2. anec_id (Primary Key, Foreign Key): Ссылка на анекдот.
    3. value: Оценка.
— Code — сущность, использующаяся для хранения информации о коде подтверждения email пользователя при регистрации.
    1. id (Primary Key): идентификатор.
    2. code: код.
    3. user_id (Foreign Key): ссылка на пользователя.
    4. expired_at: время истечения срока действия.

Для создания схемы базы данных использовался как язык SQL, так и ORM-модели.

ORM (Object-Relational Mapping) — это технология, которая позволяет разработчикам работать с базами данных, используя объектно-ориентированный подход. ORM автоматически преобразует данные из базы данных (реляционные таблицы) в объекты языка программирования и наоборот.

ORM-модель — это объектно-ориентированное представление таблицы базы данных. Каждая таблица в базе данных соответствует классу в коде, а строки таблицы представлены как экземпляры этого класса.

ORM-сущность, которая используется при работе с фреймворком Symfony является Doctrine.

Doctrine — это мощный набор библиотек на языке PHP, предназначенных для работы с базами данных. Наиболее известный компонент Doctrine, как уже было упомянуто выше, это Doctrine ORM (Object-Relational Mapping), который предоставляет возможность связывать объекты PHP с реляционными таблицами баз данных.

Основные компоненты Doctrine:

1. Doctrine ORM:
    — Инструмент для объектно-реляционного отображения.
    — Позволяет работать с базой данных через объекты, избегая необходимости написания SQL-запросов.
    — Сопоставляет классы и их свойства с таблицами и столбцами в базе

данных.

2. Doctrine DBAL (Database Abstraction Layer):
   - Упрощает выполнение низкоуровневых операций с базой данных.
   - Поддерживает работу с различными СУБД, используя унифицированный интерфейс.
3. Doctrine Migrations — обеспечивает управление версионированием схемы базы данных через миграции.
4. Doctrine Common — Содержит базовые функции и утилиты, используемые в других компонентах Doctrine.

В нашем случае Doctrine сгенерировала следующую миграцию, которую впоследствии применили к БД:

```
1  public function up(Schema $schema): void
2  {
3      $this->addSql('CREATE SEQUENCE anecdote_id_seq INCREMENT BY 1
   ↪  MINVALUE 1 START 1');
4      $this->addSql('CREATE SEQUENCE "user_id_seq" INCREMENT BY 1
   ↪  MINVALUE 1 START 1');
5      $this->addSql('CREATE TABLE anecdote (id INT NOT NULL,
   ↪  author_id_id INT NOT NULL, title VARCHAR(255) NOT NULL, text
   ↪  VARCHAR(255) NOT NULL, category VARCHAR(127) NOT NULL, PRIMARY
   ↪  KEY(id))');
6      $this->addSql('CREATE INDEX IDX_A5051EEC69CCBE9A ON anecdote
   ↪  (author_id_id)');
7      $this->addSql('CREATE TABLE code (id INT NOT NULL, code
   ↪  VARCHAR(255) NOT NULL, user_id_id INT NOT NULL, expired_at
   ↪  TIMESTAMP(0) WITHOUT TIME ZONE NOT NULL, PRIMARY KEY(id,
   ↪  code))');
8      $this->addSql('CREATE INDEX IDX_771530989D86650F ON code
   ↪  (user_id_id)');
9      $this->addSql('COMMENT ON COLUMN code.expired_at IS
   ↪  \'(DC2Type:datetime_immutable)\'');
10     $this->addSql('CREATE TABLE mark (user_id_id INT NOT NULL,
   ↪  anecdote_id_id INT NOT NULL, value INT NOT NULL, PRIMARY
   ↪  KEY(user_id_id, anecdote_id_id))');
```

```
11      $this->addSql('CREATE INDEX IDX_6674F2719D86650F ON mark
    ↪   (user_id_id)');
12      $this->addSql('CREATE INDEX IDX_6674F271A347EF68 ON mark
    ↪   (anecdote_id_id)');
13      $this->addSql('CREATE TABLE "user" (id INT NOT NULL, surname
    ↪   VARCHAR(255) NOT NULL, name VARCHAR(255) NOT NULL, patronymic
    ↪   VARCHAR(255) DEFAULT NULL, email VARCHAR(255) NOT NULL,
    ↪   PRIMARY KEY(id))');
14      $this->addSql('ALTER TABLE anecdote ADD CONSTRAINT
    ↪   FK_A5051EEC69CCBE9A FOREIGN KEY (author_id_id) REFERENCES
    ↪   "user" (id) NOT DEFERRABLE INITIALLY IMMEDIATE');
15      $this->addSql('ALTER TABLE code ADD CONSTRAINT
    ↪   FK_771530989D86650F FOREIGN KEY (user_id_id) REFERENCES "user"
    ↪   (id) NOT DEFERRABLE INITIALLY IMMEDIATE');
16      $this->addSql('ALTER TABLE mark ADD CONSTRAINT
    ↪   FK_6674F2719D86650F FOREIGN KEY (user_id_id) REFERENCES "user"
    ↪   (id) NOT DEFERRABLE INITIALLY IMMEDIATE');
17      $this->addSql('ALTER TABLE mark ADD CONSTRAINT
    ↪   FK_6674F271A347EF68 FOREIGN KEY (anecdote_id_id) REFERENCES
    ↪   anecdote (id) NOT DEFERRABLE INITIALLY IMMEDIATE');
18  }
```

Таким образом, была создана схема базы данных.

2.2 Создание базовых CRUD для Backend-приложения

2.2.1 Постановка задачи

Построить CRUD (API) PHP-приложения на фреймворке Symfony для функционирования Backend-приложения.

2.2.2 Необходимые понятия

DTO (Data Transfer Object) — это объект, который используется для передачи данных между различными слоями приложения или между разными сервисами. DTO не содержит бизнес-логики, а лишь представляет структуру данных, которая упрощает и стандартизирует процесс передачи информации.

Основные задачи DTO:

1. Стандартизация данных: DTO помогает определить строгую структуру данных, передаваемых между слоями, что улучшает читаемость и поддержку кода.
2. Изоляция внутренних структур: DTO защищает внутренние модели данных от прямого взаимодействия с внешними системами, предотвращая возможные утечки или модификации данных.
3. Оптимизация передачи данных: DTO может содержать только необходимые для передачи данные, что снижает объем передаваемой информации, особенно важно при работе с REST API.
4. Упрощение сериализации: DTO легко преобразовать в форматы передачи данных, такие как JSON или XML, что удобно при обмене данными между клиентом и сервером.

Преимущества использования Symfony для REST API

1. Автоматизация: Генерация кода упрощает создание API.
2. Мощный ORM: Doctrine облегчает управление базой данных.
3. API Platform: Symfony интегрируется с API Platform для создания сложных API.
4. Валидация: Простая интеграция с системой валидации данных.
5. Гибкость: Легко добавлять middleware, авторизацию и другие механизмы.

Symfony — это мощный PHP-фреймворк, который упрощает создание REST API.

REST (Representational State Transfer) — это архитектурный стиль для разработки распределённых систем, особенно веб-сервисов. Этот подход базируется на наборе принципов и ограничений, которые определяют, как системы взаимодействуют через интернет, используя стандартный протокол HTTP.

1. Клиент-серверная архитектура. Клиент и сервер чётко разделены: клиент отвечает за интерфейс пользователя, а сервер управляет данными и их обработкой. Это разделение упрощает разработку и масштабирование системы.
2. Отсутствие состояния (Stateless). Каждый запрос от клиента к серверу должен содержать всю необходимую информацию для обработки. Сервер не сохраняет состояние сессии между запросами, что упрощает его масштабирование.

3. Кэширование. Ответы сервера могут быть помечены как кэшируемые или некэшируемые. Это уменьшает нагрузку на сервер и повышает производительность, если данные изменяются редко.
4. Единообразие интерфейса (Uniform Interface). REST придерживается стандартизированного набора методов HTTP: GET, POST, PUT, DELETE и др. Ресурсы идентифицируются через URI (Uniform Resource Identifier), и операции над ними описываются этими методами.
5. Многоуровневая система (Layered System). Компоненты могут быть распределены на нескольких уровнях (например, прокси, серверы кэширования), что увеличивает надёжность и масштабируемость системы.
6. Представления ресурсов (Representations). Ресурсы передаются в формате представления: JSON, XML, HTML или других. Это позволяет клиентам работать с данными независимо от внутреннего устройства сервера.
7. Код по требованию (Code-on-Demand) (опционально). Сервер может передавать исполняемый код (например, JavaScript) клиенту для выполнения, что повышает гибкость клиентской части.

Далее рассмотрим особенности модели MVC и для чего в ней требуется контроллер

MVC (Model-View-Controller) — это архитектурный шаблон проектирования, который используется для разделения логики приложения на три взаимосвязанных компонента: Model (модель), View (представление) и Controller (контроллер). Этот подход позволяет повысить модульность, читаемость и тестируемость кода, а также упростить его поддержку и масштабирование.

Компоненты MVC:

1. Model (Модель)
   а) Отвечает за управление данными приложения.
   б) Включает бизнес-логику, правила валидации данных и взаимодействие с базой данных.
   в) Обеспечивает независимость от пользовательского интерфейса.
2. View (Представление)
   а) Отвечает за отображение данных пользователю.
   б) Содержит пользовательский интерфейс (HTML, CSS, JavaScript в веб-приложениях).
   в) Не содержит логики работы с данными, а только принимает их из

модели через контроллер.

3. Controller (Контроллер)
   а) Является посредником между моделью и представлением.
   б) Обрабатывает пользовательские запросы, вызывает соответствующие методы модели и подготавливает данные для представления.

На что следует обратить внимание при внедрении MVC:

1. Сложность для небольших проектов: для маленьких приложений использование MVC может быть избыточным.
2. Явное разделение кода: код каждой части должен строго соответствовать своей ответственности, что требует внимательного проектирования.
3. Обратная связь между компонентами: контроллер соединяет модель и представление, что позволяет избежать тесной зависимости между ними.

MVC используется во многих современных фреймворках, в том числе и в используемом в нашем случае фреймворке PHP — Symfony.

2.2.3 Решение

Для решения данной задачи требовалось:

1. Создать DTO с помощью инструментов PHP-фреймворка Symfony
2. Создать класс-контроллер для сущности Anecdote
3. Создать бизнес-логику для указанных endpoint из вышеописанного контроллера

Рассмотрим пример на одной из сущностей (анекдоте): остальные реализовались по аналогичному сценарию.

Создание контроллера для сущности Anecdote с относительным URI `anecdote`, а также DTO для работы этого контроллера (обработки запросов и ответов) указано в приложении А

В этом коде представлены основные операции с сущностью Anecdote: создание, обновление, чтение, удаление. Это реализуется с помощью вызова у переменной `$anecdoteService` методов create, edit и прочих.

Сама же переменная $anecdoteService является объектом класса Anecdote-Service, в котором хранится вся бизнес-логика методов, связанных с сущностью Anecdote.

Рассмотрим один из методов: их логика достаточно тривиальна:

```
1  public function editAnecdote(Anecdote $anecdote,
   ↪  AnecdoteBaseRequestDTO $DTO): AnecdoteBaseResponseDTO
2  {
3      if ($title = $DTO->title) {
4          $anecdote->setTitle($title);
5      }
6      if ($text = $DTO->text) {
7          $anecdote->setText($text);
8      }
9      if ($category = $DTO->category) {
10         $anecdote->setCategory($category);
11     }
12     $this->entityManager->flush();
13     return new AnecdoteBaseResponseDTO($anecdote);
14 }
```

Таким образом, при последующем использовании кода были выявлены следующие преимущества данного подхода к решению задачи:

1. DTO позволяет отделить внутренние модели приложения от структуры данных, передаваемых клиенту, что упрощает форматирование ответов и защиту чувствительных данных.
2. Контроллеры обеспечивают чистую организацию кода, выступая посредниками между бизнес-логикой и клиентами API, обрабатывая запросы, вызовы сервисов и формирование ответов [7–9].

2.3 Создание авторизации на основе концепции access- и refresh-токенов

2.3.1 Постановка задачи

Создать механизм авторизации на языке PHP с помощью access- и refresh-токенов.

2.3.2 access- и refresh-токены

Использование access и refresh токенов — это подход для безопасной и удобной аутентификации пользователей, часто применяемый в системах с использованием JWT (JSON Web Tokens). Этот метод позволяет

минимизировать риски, связанные с компрометацией токенов, и улучшить пользовательский опыт за счёт автоматического обновления сессии.

Access-токен — краткоживущий токен, содержащий информацию об аутентификации пользователя (например, ID, роли и права доступа). Используется для выполнения запросов к защищённым ресурсам API. Имеет короткий срок действия (например, 15 минут), что снижает последствия его утечки.

Refresh-токен — долгоживущий токен, используемый только для получения нового access-токена. Не используется напрямую для доступа к API. Хранится в более защищённом месте (например, в HTTP-only cookies), чтобы минимизировать риск его утечки.

Алгоритм работы системы:

1. Пользователь аутентифицируется (например, с помощью логина и пароля).
2. Сервер выдаёт:
   а) Access-токен для доступа к ресурсам API.
   б) Refresh-токен для продления действия сессии.
3. Клиент отправляет access-токен в каждом запросе к API (обычно в заголовке Authorization: Bearer <token>).
4. Если access-токен истёк, клиент использует refresh-токен, чтобы получить новый access-токен через специальный API-эндпоинт.
5. Если refresh-токен истёк, пользователь должен пройти повторную аутентификацию [10, 11].

2.3.3 Решение

Для выполнения задачи требуется выполнить следующее:

1. Создать сущность Device и добавить её в базу данных. Указать в ней два токена - access и refresh, у каждого своё время жизни.
2. Реализовать проверку на время жизни токена в ApiAuthenticator.
3. Реализовать метод обновления времени жизни access токена.
4. Добавить функционал отправки письма на почту при регистрации с помощью MailerService

Все эти шаги описаны в классе SecurityService, код которого содержится в приложении Б.

2.4 Создание docker-compose файлов для настройки контейнеризации и автоматизации запуска приложения

2.4.1 Постановка задачи

Создать автоматизацию контейнеризации с помощью docker и docker-compose в .yaml файле.

2.4.2 Docker и docker compose

Docker — это технология контейнеризации, которая упрощает создание, развертывание и управление приложениями. Она позволяет упаковать приложение и его зависимости в изолированные контейнеры, которые могут быть запущены на любой платформе, поддерживающей Docker.

Контейнеризация с использованием Docker отличается рядом особенностей, делающих её удобным инструментом для разработки, тестирования и развертывания приложений.

Преимущества Docker:

1. Изоляция среды. Docker контейнеры изолируют приложения и их зависимости, что позволяет избежать конфликтов с локальной средой разработчика или сервером. Каждое приложение запускается в своей среде, не влияя на другие.
2. Универсальность и совместимость. Контейнеры гарантируют, что приложение будет работать одинаково на любом хосте, где установлен Docker (локальная машина, сервер, облако).
3. Быстрое развертывание. Запуск контейнера занимает секунды, так как он использует легковесные образы, а не тяжелые виртуальные машины.
4. Повторяемость. С использованием Dockerfile можно точно задокументировать и воспроизвести среду разработки и продакшн.
5. Легкость тестирования. Тестирование в изолированной среде Docker помогает обнаружить проблемы, которые могут проявиться в реальном продакшне.
6. Упрощенная миграция и масштабирование. Легко переносить приложение между средами (локальная разработка, staging, production). Контейнеры масштабируются горизонтально с помощью оркестрации (например, Kubernetes или Docker Swarm).

2.4.3 Решение

Контейнеризация с помощью Docker Compose позволяет легко управлять многокомпонентными приложениями, описывая их инфраструктуру (контейнеры, сети, тома) в одном файле docker-compose.yml. Это упрощает настройку, запуск и масштабирование, позволяя разработчикам и командам быстро развертывать целые приложения с зависимостями (например, серверы, базы данных) с помощью одной команды (docker-compose up).

Для решения напишем .yaml файл, который как раз будет контролировать запуск сервисов, описанных в docker compose.

```
1  version: '3.9'
2
3  services:
4      nginx:
5      build:
6          context: ./build/nginx
7          dockerfile: Dockerfile
8      container_name: nginx-task3
9      volumes:
10         - ./app/public:/var/www/app/public/:ro
11         - ./build/nginx/config:/etc/nginx/conf.d/:ro
12     ports:
13         - 8080:80
14     networks:
15         - default
16
17     db:
18     image: postgres:15.2-alpine3.17
19     container_name: db-task3
20     environment:
21         POSTGRES_PASSWORD: postgres
22         POSTGRES_DB: db
23         POSTGRES_USER": krab1o
24         POSTGRES_HOST: krab1ocomp
25     volumes:
```

```
26          - db_volume:/var/lib/postgresql/data
27      networks:
28          - default
29
30      php:
31      build:
32          context: ./build/php
33          dockerfile: Dockerfile
34      container_name: php-task3
35      volumes:
36          - ./app:/var/www/app
37      networks:
38          - default
39  networks:
40      default:
41  volumes:
42      db_volume:
```

При этом docker-compose использует два кастомных Docker-образа, которые описаны ниже.

Здесь представлен Docker-файл для сервиса nginx:

```
1  FROM php:fpm-alpine3.20 as php_upstream
2  FROM composer/composer:2-bin
3
4  FROM php_upstream as php_base
5
6  WORKDIR /var/www/app
7
8  RUN apk update && apk upgrade && apk add php php-fpm
9
10 COPY --from=composer /usr/bin/composer /usr/bin/composer
11
12 EXPOSE 9000
13
14 CMD ["php-fpm"]
```

А здесь — для сервиса backend PHP:

```
1 FROM nginx:stable-alpine
2
3 WORKDIR /var/www
4
5 EXPOSE 80
6
7 CMD ["nginx", "-g", "daemon off;"]
```

Таким образом, приложение на PHP с сервисом базы данных и сервером nginx, который выполняет функцию reverse-proxy, поднимается с помощью docker compose, а кастомные команды, которые требуется выполнить в образах сервера и backend-приложения, описаны в Dockerfile [12, 13].

ЗАКЛЮЧЕНИЕ

В ходе выполнения данной работы были изучены современные инструменты и технологии для разработки серверной части веб-приложений. В частности, были рассмотрены особенности работы с языком PHP и фреймворком Symfony, а также разработана архитектура приложения на основе MVC.

Настроено контейнеризированное окружение с использованием Docker и веб-сервера Nginx для обеспечения эффективной и стабильной работы приложения. Особое внимание уделялось тестированию RESTful API с помощью Postman и документированию его с использованием Swagger. Также проведен анализ преимуществ и недостатков применяемых технологий в контексте разработки масштабируемых и безопасных веб-приложений.

Перспективы применения результатов данной работы достаточно широки.

Во-первых, разработанное приложение может быть использовано как основа для дальнейшего развития и расширения функциональности, например, для добавления новых типов контента или внедрения системы рекомендаций на основе предпочтений пользователей.

Во-вторых, контейнеризация с использованием Docker позволяет легко масштабировать приложение и развертывать его в различных окружениях, что важно для гибкости и мобильности современных веб-сервисов.

Применение Nginx как веб-сервера повышает производительность и надежность работы приложения при обработке большого количества запросов, что открывает возможности для использования данного решения в высоконагруженных системах. Кроме того, навыки тестирования и документирования API с Postman и Swagger могут быть применены в разработке других проектов, требующих четкой и структурированной документации интерфейсов.

Таким образом, созданный проект достигнул поставленных задач школы ПрофСофт, цель производственной практики была достигнута, а все поставленные в ходе практики задачи решены.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1 *Symfony,*. Symfony framework - Официальная документация. — https://symfony.com/doc/current/index.html. — 2024. — (Дата обращения 14.08.2024) Загл. с экр. Яз. англ.

2 *Robots.net,*. What is php symfony? — https://robots.net/programming/symfony-php-framework/. — 2024. — (Дата обращения 14.08.2024) Загл. с экр. Яз. англ.

3 *Postman,*. Postman - Описание концепции api тестирования. — https://www.postman.com/. — 2024. — (Дата обращения 16.08.2024) Загл. с экр. Яз. англ.

4 *Postman,*. Postman - Официальная документация. — https://learning.postman.com/docs/. — 2024. — (Дата обращения 16.08.2024) Загл. с экр. Яз. англ.

5 *Nginx,*. Nginx - Официальная документация. — https://nginx.org/en/docs/. — 2024. — (Дата обращения 16.08.2024) Загл. с экр. Яз. англ.

6 *Group, P. G. D.* Postgresql documentation [Электронный ресурс]. — https://www.postgresql.org/docs/. — (Дата обращения 05.08.2024) Загл. с экр. Яз. англ.

7 *RESTfulAPI.net,*. Rest api tutorial. — 2024. — (Дата обращения 11.08.2024) Загл. с экр. Яз. англ. https://restfulapi.net/.

8 *Microsoft,*. Learn rest api on microsoft docs. — 2024. — (Дата обращения 11.08.2024) Загл. с экр. Яз. англ. https://learn.microsoft.com/en-us/azure/architecture/best-practices/api-design.

9 *Amazon Web Services,*. What is restful api? — 2024. — (Дата обращения 11.08.2024) Загл. с экр. Яз. англ. https://aws.amazon.com/what-is/restful-api/.

10 *Baeldung,*. What are access and refresh tokens? — 2024. — (Дата обращения 09.08.2024) Загл. с экр. Яз. англ. https://www.baeldung.com/cs/access-refresh-tokens.

11 *Auth0,*. What are refresh tokens and how to use them securely. — 2024. — (Дата обращения 09.08.2024) Загл. с экр. Яз. англ. https://auth0.com/docs/secure/tokens/refresh-tokens.

12 *Frazelle, J.* Docker: Up and Running / J. Frazelle. — O'Reilly Media, 2015.

13 *Docker Inc.,*. Docker documentation. — 2024. — (Дата обращения 03.08.2024) Загл. с экр. Яз. англ. https://docs.docker.com/.

ПРИЛОЖЕНИЕ А

Создание контроллера и DTO для него

```
#[Route(path: '/anecdote')]
class AnecdoteController extends AbstractController
{
    public function __construct(
        private readonly ValidatorService    $validator,
        private readonly SerializerInterface $serializer,
    ) { }
    #[Route(path: '', name: 'apiGetAnecdoteList', methods:
↪ Request::METHOD_GET)]
    public function getAnecdoteList(AnecdoteService
↪ $anecdoteService): JsonResponse
    {
        return $this->json(
            data: $anecdoteService->getAnecdoteList(),
            status: Response::HTTP_OK,
        );
    }
    #[Route(path: '', name: 'apiCreateAnecdote', methods:
↪ Request::METHOD_POST)]
    public function createUser(Request $request, AnecdoteService
↪ $anecdoteService): JsonResponse
    {
        $data =
↪ $this->serializer->deserialize($request->getContent(),
↪ AnecdoteBaseRequestDTO::class, 'json');
        $this->validator->validate(body: $data, groupsBody:
↪ ['register']);
        return $this->json(
            data: $anecdoteService->createAnecdote($data),
            status: Response::HTTP_CREATED,
        );
    }
```

```
26      #[Route(path: '/{id<\d+>}', name: 'apiEditAnecdote', methods:
    ↪   Request::METHOD_PATCH)]
27      public function editAnecdote(
28          Anecdote        $id,
29          Request         $request,
30          AnecdoteService $anecdoteService,
31      ): JsonResponse
32      {
33          $data =
    ↪   $this->serializer->deserialize($request->getContent(),
    ↪   AnecdoteBaseRequestDTO::class, 'json');
34          $this->validator->validate(body: $data, groupsBody:
    ↪   ['edit']);
35
36          return $this->json(
37              data: $anecdoteService->editAnecdote($id, $data),
38              status: Response::HTTP_CREATED,
39          );
40      }
41      #[Route(path: '/{id<\d+>}', name: 'apiDeleteAnecdote',
    ↪   methods: Request::METHOD_DELETE)]
42      public function deleteUser(Anecdote $id, AnecdoteService
    ↪   $anecdoteService): JsonResponse
43      {
44          $anecdoteService->deleteAnecdote($id);
45
46          return $this->json(
47              data: [],
48              status: Response::HTTP_NO_CONTENT,
49          );
50      }
51  }
```

```
1  class AnecdoteBaseRequestDTO
2  {
```

```
3       public function __construct(
4           #[Assert\NotNull(groups: ['register'])]
5           #[Assert\Type(type: 'string', groups: ['register',
    ↪  'edit'])]
6           #[Assert\Length(max: 127, groups: ['register', 'edit'])]
7           public ?string $title = null,
8
9           #[Assert\NotNull(groups: ['register'])]
10          #[Assert\Type(type: 'string', groups: ['register',
    ↪  'edit'])]
11          public ?string $text = null,
12
13          #[Assert\NotNull(groups: ['register'])]
14          #[Assert\Type(type: 'string', groups: ['register',
    ↪  'edit'])]
15          public ?bool $category = null,
16      ) { }
17  }
```

```
1   class AnecdoteBaseResponseDTO
2   {
3       public int $id;
4       public string $title;
5       public string $text;
6       public string $category;
7
8       public function __construct(Anecdote $anecdote)
9       {
10          $this->id = $anecdote->getId();
11          $this->title = $anecdote->getTitle();
12          $this->text = $anecdote->getText();
13          $this->category = $anecdote->getCategory();
14      }
15  }
```

ПРИЛОЖЕНИЕ Б

Создание авторизации

```
class SecurityService
{
    private const string SUBJECT = 'Код авторизации';

    private const string ACCESS_TOKEN_LIFETIME = '+10 minutes';
    private const string REFRESH_TOKEN_LIFETIME = '+90 days';

    public function __construct(
        #[Autowire(service: YandexMailerService::class)]
        private MailerServiceInterface   $mailer,
        protected EntityManagerInterface $entityManager,
    ) { }
    public function sendCode(LoginDTO $DTO): void
    {
        $code = new Code();
        if (!$this->entityManager->getRepository(User::class)->
        findOneByEmail($DTO->email) instanceof User) {
            throw new ApiException(
                message: "Пользователь по указанному email не
    ↪ найден",
                status: Response::HTTP_NOT_FOUND,
            );
        }
        $code
            ->setEmail($DTO->email);

        $this->entityManager->persist($code);
        $this->entityManager->flush();
        $this->mailer->send(self::SUBJECT, $code->getCode(),
    ↪ (array)$DTO->email);
    }

```

```
31      public function verifyCode(LoginDTO $DTO): array
32      {
33          $code = $this->entityManager->getRepository(Code::class)->
34          findOneBy([
35              'code' => $DTO->code,
36              'email' => $DTO->email,
37              'status' => CodeStatus::ACTIVE->value,
38          ]);
39
40          if (!$code instanceof Code) {
41              throw new ApiException(
42                  'Неверный код авторизации',
43                  status: Response::HTTP_UNAUTHORIZED,
44              );
45          }
46          if ($code->getExpiredAt() < new \DateTime()) {
47              $code
48                  ->setStatus(CodeStatus::EXPIRED->value);
49              $this->entityManager->flush();
50              throw new ApiException(
51                  'Код авторизации истек',
52                  status: Response::HTTP_UNAUTHORIZED,
53              );
54          }
55          $owner =
    ↪ $this->entityManager->getRepository(User::class)->
56          findOneBy([
57              'email' => $DTO->email,
58          ]);
59
60          $device = (new Device())
61              ->setOwner($owner)
62              ->setTokenExpiresAt((new
    ↪ \DateTime())->modify(self::ACCESS_TOKEN_LIFETIME))
```

```
63              ->setRefreshTokenExpiresAt((new
    ↪ \DateTime())->modify(self::REFRESH_TOKEN_LIFETIME));
64
65          $code
66              ->setStatus(CodeStatus::INACTIVE->value);
67
68          $this->entityManager->persist($device);
69          $this->entityManager->flush();
70
71          return [
72              'token' => $device->getToken(),
73              'refreshToken' => $device->getRefreshToken(),
74          ];
75      }
76      public function logout(string $apikey): void
77      {
78          $device =
    ↪ $this->entityManager->getRepository(Device::class)->
79          findOneBy([
80              'apikey' => $apikey,
81          ]);
82          $device->setStatus(DeviceStatus::INACTIVE->value);
83          $this->entityManager->flush();
84      }
85      public function refresh(?string $refreshToken): array {
86          $device =
    ↪ $this->entityManager->getRepository(Device::class)->
87          findOneBy([
88              'refreshToken' => $refreshToken,
89              'status' => DeviceStatus::ACTIVE->value,
90          ]);
91
92          if (!$device instanceof Device) {
93              throw new ApiException(
```

```
94                  message: 'Некорректный refresh токен',
95                  status: Response::HTTP_UNAUTHORIZED,
96              );
97          }
98
99          if ($device->getRefreshTokenExpiresAt() < new \DateTime())
    ↪ {
100             $device->setStatus(DeviceStatus::EXPIRED->value);
101             $this->entityManager->flush();
102
103             throw new ApiException(
104                 message: 'Refresh токен истёк',
105                 status: Response::HTTP_UNAUTHORIZED,
106             );
107         }
108
109         $device
110             ->setToken($this->generateToken())
111             ->setTokenExpiresAt((new
    ↪ \DateTime())->modify(self::ACCESS_TOKEN_LIFETIME))
112             ->setRefreshToken($this->generateToken());
113
114         $this->entityManager->flush();
115
116         return [
117             'token' => $device->getToken(),
118             'refreshToken' => $device->getRefreshToken(),
119         ];
120     }
121     public function generateToken(): string {
122         return md5(random_int(100000, 999999) . microtime());
123     }
124 }
```