**СОДЕРЖАНИЕ**

[ВВЕДЕНИЕ 6](#_Toc194508225)

[1.ПРОЕКТИРОВАНИЕ АРХИТЕКТУРЫ ПРИЛОЖЕНИЯ «СОЗДАНИЕ, ХРАНЕНИЕ И ОБРАБОТКА ДАННЫХ О ПРОГРАММНОМ ОБЕСПЕЧЕНИИ, ИЗУЧЕННОМ СТУДЕНТОМ ЗА ТЕКУЩИЙ СЕМЕСТР» 7](#_Toc194508226)

[2.РЕАЛИЗАЦИЯ ФУНКЦИОНАЛЬНОЙ ЧАСТИ ПРИЛОЖЕНИЯ «СОЗДАНИЕ, ХРАНЕНИЕ И ОБРАБОТКА ДАННЫХ О ПРОГРАММНОМ ОБЕСПЕЧЕНИИ, ИЗУЧЕННОМ СТУДЕНТОМ ЗА ТЕКУЩИЙ СЕМЕСТР» 9](#_Toc194508227)

[3.РЕАЛИЗАЦИЯ СЕРВЕРНОЙ ЧАСТИ ПРИЛОЖЕНИЯ «СОЗДАНИЕ, ХРАНЕНИЕ И ОБРАБОТКА ДАННЫХ О ПРОГРАММНОМ ОБЕСПЕЧЕНИИ, ИЗУЧЕННОМ СТУДЕНТОМ ЗА ТЕКУЩИЙ СЕМЕСТР» 10](#_Toc194508228)

[3.1 Проектирование внешнего API 10](#_Toc194508229)

[3.2 Спецификация реализованного внешнего API 11](#_Toc194508230)

[3.2.1 Технологический стек 15](#_Toc194508231)

[3.2.2 Артефакты процесса разработки 16](#_Toc194508232)

[3.3 Тестирование серверной части приложения 19](#_Toc194508233)

[4.РАЗВЕРТЫВАНИЕ СЕРВЕРНОЙ ЧАСТИ ПРИЛОЖЕНИЯ «СОЗДАНИЕ, ХРАНЕНИЕ И ОБРАБОТКА ДАННЫХ О ПРОГРАММНОМ ОБЕСПЕЧЕНИИ, ИЗУЧЕННОМ СТУДЕНТОМ ЗА ТЕКУЩИЙ СЕМЕСТР» 27](#_Toc194508234)

[4.1 Описание процесса развертывания 27](#_Toc194508235)

[4.2 Тестирование развертывания 28](#_Toc194508236)

[5.РЕАЛИЗАЦИЯ КЛИЕНТСКОЙ ЧАСТИ ПРИЛОЖЕНИЯ «СОЗДАНИЕ, ХРАНЕНИЕ И ОБРАБОТКА ДАННЫХ О ПРОГРАММНОМ ОБЕСПЕЧЕНИИ, ИЗУЧЕННОМ СТУДЕНТОМ ЗА ТЕКУЩИЙ СЕМЕСТР» 30](#_Toc194508237)

[5.1 Технологический стек 30](#_Toc194508238)

[5.2 Описание реализации 31](#_Toc194508239)

[5.3 Описание процесса взаимодействия конечного пользователя с клиентом 31](#_Toc194508240)

[6.РАЗВЕРТЫВАНИЕ КЛИЕНТСКОЙ ЧАСТИ ПРИЛОЖЕНИЯ «СОЗДАНИЕ, ХРАНЕНИЕ И ОБРАБОТКА ДАННЫХ О ПРОГРАММНОМ ОБЕСПЕЧЕНИИ, ИЗУЧЕННОМ СТУДЕНТОМ ЗА ТЕКУЩИЙ СЕМЕСТР» 34](#_Toc194508241)

[6.1 Описание процесса развертывания 34](#_Toc194508242)

[6.2 Тестирование развертывания 34](#_Toc194508243)

[7.НАСТРОЙКА КОНФИГУРАЦИИ МНОГОКОНТЕЙНЕРНОГО РАЗВЕРТЫВАНИЯ ПРИЛОЖЕНИЯ «СОЗДАНИЕ, ХРАНЕНИЕ И ОБРАБОТКА ДАННЫХ О ПРОГРАММНОМ ОБЕСПЕЧЕНИИ, ИЗУЧЕННОМ СТУДЕНТОМ ЗА ТЕКУЩИЙ СЕМЕСТР» 36](#_Toc194508244)

[ЗАКЛЮЧЕНИЕ 39](#_Toc194508245)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 40](#_Toc194508246)

# ВВЕДЕНИЕ

Актуальность темы данного исследования обусловлена необходимостью разработки эффективной системы для создания, хранения и обработки данных о программном обеспечении, изученном студентами за текущий семестр. В условиях интенсивного обучения и быстрого обновления информации об образовательных курсах, разработка системы для учета таких данных становится важной задачей. Точные и актуальные сведения о программных продуктах, освоенных студентами, необходимы преподавателям, студентам учебных заведений для оценки успехов и планирования образовательных программ.

Объект исследования – это система для управления данными о программном обеспечении, изученном студентами. Предмет исследования заключается в особенностях проектирования, разработки и реализации архитектуры данной системы, включая процессы создания базы данных, обработки информации и обеспечения интеграции с другими компонентами образовательного процесса.

Цель работы – разработать и развернуть функциональную систему для создания, хранения и обработки данных о программном обеспечении, изученном студентами в течение семестра.

В ходе работы будет разработана структура базы данных, проектирована архитектура системы, реализованы функции для обработки данных. Важной частью работы будет использование современных технологий для обеспечения надежности, удобства использования решения, что сделает систему доступной и эффективной в рамках образовательного процесса.

# 1. ПРОЕКТИРОВАНИЕ АРХИТЕКТУРЫ ПРИЛОЖЕНИЯ «СОЗДАНИЕ, ХРАНЕНИЕ И ОБРАБОТКА ДАННЫХ О ПРОГРАММНОМ ОБЕСПЕЧЕНИИ, ИЗУЧЕННОМ СТУДЕНТОМ ЗА ТЕКУЩИЙ СЕМЕСТР»

На Рисунке 1.1 представлена структурная схема приложения в виде диаграммы развертывания.

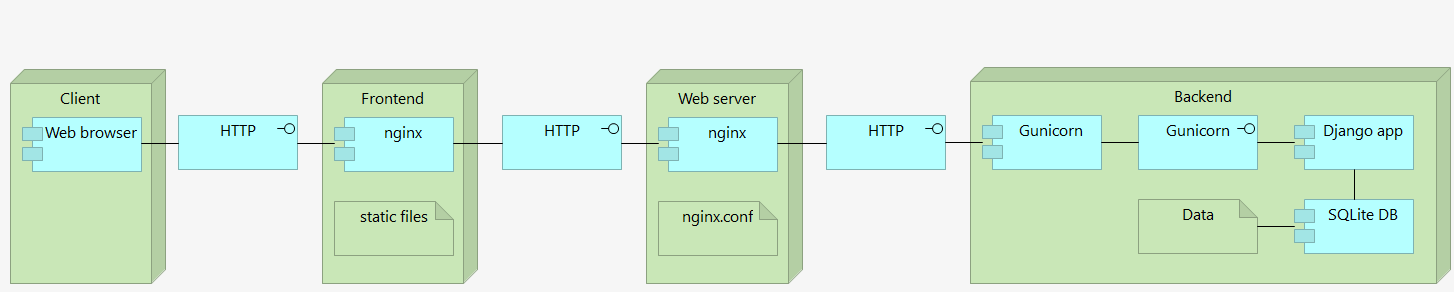


Рисунок . — Структурная схема приложения

Система построена на Python-фреймворке, который реализует ключевую бизнес-логику, обеспечивает безопасность, обработку запросов и работу с данными. В качестве основного хранилища используется встроенная реляционная СУБД SQLite3. Для создания интерактивного и динамичного пользовательского интерфейса применяется JavaScript, что позволяет разрабатывать удобные веб-приложения с высокой отзывчивостью. Обмен данными между клиентом и сервером осуществляется через RESTful API, реализованный в Django, обеспечивающий взаимодействие по стандартным HTTP-методам с передачей данных в формате JSON. В качестве прокси-сервера используется Nginx, который выполняет балансировку нагрузки, кэширование статических файлов, маршрутизацию запросов и защиту системы. Для удобства развертывания и согласованной работы всех компонентов применяется Docker. Каждая часть системы – фронтенд, сервер на Django и Nginx – контейнеризуется отдельно, что гарантирует их изолированность, удобство масштабирования и воспроизводимость среды в различных окружениях. Управление инфраструктурой осуществляется с помощью Docker Compose, что упрощает процесс развертывания, тестирования и эксплуатации системы.

# 2. РЕАЛИЗАЦИЯ ФУНКЦИОНАЛЬНОЙ ЧАСТИ ПРИЛОЖЕНИЯ «СОЗДАНИЕ, ХРАНЕНИЕ И ОБРАБОТКА ДАННЫХ О ПРОГРАММНОМ ОБЕСПЕЧЕНИИ, ИЗУЧЕННОМ СТУДЕНТОМ ЗА ТЕКУЩИЙ СЕМЕСТР»

Приложение реализует управление данными о программном обеспечении, освоенном студентами, включая их создание, обновление и представление. Для работы с базой данных SQLite используется Django ORM, что обеспечивает удобное взаимодействие с хранилищем. В процессе разработки применялись основные методы моделей, такие как save() и delete() для сохранения и удаления записей, а также методы запросов, включая get\_object\_or\_404() и all(), позволяющие получать необходимые данные.

Архитектура системы разделена на две части: клиентскую и серверную. Фронтенд представлен статическими HTML-страницами и предоставляет удобный интерфейс для взаимодействия пользователей с системой. Серверная часть, разработанная на основе Django, отвечает за обработку пользовательских запросов, управление логикой приложения и работу с базой данных через Django ORM.

Взаимодействие между фронтендом и бекендом осуществляется с использованием RESTful API, что позволяет передавать данные в формате JSON, обеспечивая гибкость и расширяемость системы. Для обеспечения стабильности и корректности работы приложения проведено тестирование базовой функциональности, включающее проверку процессов создания, обновления, просмотра и удаления данных о студентах и их изученном ПО. Все основные сценарии использования API были протестированы вручную, что подтвердило соответствие системы требованиям и корректность ее функционирования.

# 3. РЕАЛИЗАЦИЯ СЕРВЕРНОЙ ЧАСТИ ПРИЛОЖЕНИЯ «СОЗДАНИЕ, ХРАНЕНИЕ И ОБРАБОТКА ДАННЫХ О ПРОГРАММНОМ ОБЕСПЕЧЕНИИ, ИЗУЧЕННОМ СТУДЕНТОМ ЗА ТЕКУЩИЙ СЕМЕСТР»

## 3.1 Проектирование внешнего API

Для разработки серверной части приложения была построена таблица целей API (Таблица 3.1). Для роли «Преподаватель» предусмотрен сценарий «Управление студентами», а для роли «Студент» — «Просмотр информации». В результате было выделено пять целей.

Таблица . - Цели API

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Кто** | **Что** | **Как** | **Входные данные (источник)** | **Выходные данные (использование)** | **Цели** |
| Преподаватель | Управление студентами | Создать нового студента | Данные студента (Форма) | Созданный студент (БД) | Регистрация студента |
| Обновить информацию о студенте | Обновленные данные (Форма) | Обновленный студент (БД) | Редактирование данных |
| Удалить студента | ID студента | Подтверждение (БД) | Удаление студента |
| Студент | Просмотр информации | Просмотреть список студентов | Запрос | Список студентов (UI) | Поиск студентов |
| Просмотреть ПО студента | ID студента | Список ПО (UI) | Проверка освоенных ПО |

В рамках проектирования был определен набор схем входных и выходных данных, представленных в Таблицах 3.2-3.4.

Таблица . — Схема данных « StudentDetailed»

|  |  |  |  |
| --- | --- | --- | --- |
| **Имя** | **Тип** | **Обязательно** | **Описание** |
| id | Целое число | Да | Уникальный ID студента |
| name | Строка | Да | ФИО студента |
| software | Массив[Объект] | Нет | Освоенное ПО (вложенные данные) |

Таблица . — Схема данных «StudentBrief»

|  |  |  |  |
| --- | --- | --- | --- |
| **Имя** | **Тип** | **Обязательно** | **Описание** |
| id | Целое число | Да | Уникальный ID студента |
| name | Строка | Да | ФИО студента |

Таблица . — Схема данных «ApiException»

|  |  |  |  |
| --- | --- | --- | --- |
| **Имя** | **Тип** | **Обязательно** | **Описание** |
| detail | Строка | Да | Текст ошибки |

## 3.2 Спецификация реализованного внешнего API

Спецификация реализованного внешнего API представлена в Листинге 3.1.

Листинг . — Спецификация реализованного API

|  |
| --- |
| openapi: 3.0.3  info:  version: 1.0.0  title: API управления студентами и ПО  description: API для создания, хранения и обработки данных о студентах и освоенном ими ПО  tags:  - name: Students  description: Управление студентами  - name: Student-Software  description: Управление связями студентов и ПО  paths:  /api/student/:  get:  tags: [Students]  summary: Получить список всех студентов  responses:  '200':  description: Список студентов  content:  application/json:  schema:  type: array  items:  $ref: '#/components/schemas/StudentBrief'  post:  tags: [Students]  summary: Создать нового студента  requestBody:  required: true  content:  application/json:  schema:  $ref: '#/components/schemas/StudentRequest'  responses:  '201':  description: Студент создан  content:  application/json:  schema:  $ref: '#/components/schemas/StudentResponse'  /api/student/{student\_id}/:  get:  tags: [Students]  summary: Получить информацию о студенте  parameters:  - name: student\_id  in: path  required: true  schema:  type: integer  responses:  '200':  description: Информация о студенте  content:  application/json:  schema:  $ref: '#/components/schemas/StudentResponse'  put:  tags: [Students]  summary: Обновить информацию о студенте  parameters:  - name: student\_id  in: path  required: true  schema:  type: integer  requestBody:  required: true  content:  application/json:  schema:  $ref: '#/components/schemas/StudentRequest'  responses:  '200':  description: Информация обновлена  delete:  tags: [Students]  summary: Удалить студента  parameters:  - name: student\_id  in: path  required: true  schema:  type: integer  responses:  '204':  description: Студент удален  /api/student/{student\_id}/software/:  get:  tags: [Student-Software]  summary: Получить список ПО студента  parameters:  - name: student\_id  in: path  required: true  schema:  type: integer  responses:  '200':  description: Список ПО студента  content:  application/json:  schema:  type: array  items:  $ref: '#/components/schemas/SoftwareResponse'  post:  tags: [Student-Software]  summary: Добавить ПО студенту  parameters:  - name: student\_id  in: path  required: true  schema:  type: integer  requestBody:  required: true  content:  application/json:  schema:  $ref: '#/components/schemas/SoftwareRequest'  responses:  '201':  description: ПО добавлено студенту  delete:  tags: [Student-Software]  summary: Удалить ПО у студента  parameters:  - name: student\_id  in: path  required: true  schema:  type: integer  requestBody:  required: true  content:  application/json:  schema:  type: object  properties:  id:  type: integer  description: ID программного обеспечения  responses:  '204':  description: ПО удалено у студента  components:  schemas:  StudentRequest:  type: object  required: [name]  properties:  name:  type: string  description: ФИО студента  example: "Иванов Иван Иванович"  StudentBrief:  type: object  properties:  id:  type: integer  name:  type: string  StudentResponse:  allOf:  - $ref: '#/components/schemas/StudentBrief'  - type: object  properties:  software\_list:  type: array  items:  $ref: '#/components/schemas/SoftwareBrief'  SoftwareRequest:  type: object  required: [name, study\_level]  properties:  name:  type: string  example: "Python"  study\_level:  type: string  example: "80%"  SoftwareBrief:  type: object  properties:  id:  type: integer  name:  type: string  SoftwareResponse:  allOf:  - $ref: '#/components/schemas/SoftwareBrief'  - type: object  properties:  study\_level:  type: string |

Основные эндпоинты:

Управление студентами:

* POST /api/student/ – Создать нового студента;
* GET /api/student/ – Получить список всех студентов;
* GET /api/student/{student\_id}/ – Получить информацию о студенте;
* PUT /api/student/{student\_id}/ – Обновить информацию о студенте;
* DELETE /api/student/{student\_id}/ – Удалить студента

Управление ПО студентов:

* GET /api/student/{student\_id}/software/ – Получить список ПО студента;
* POST /api/student/{student\_id}/software/ – Добавить ПО студенту;
* DELETE /api/student/{student\_id}/software/ – Удалить ПО у студента.

Ключевыми аспектами данной спецификации являются определения моделей запросов и ответов, включая формат данных и возможные коды HTTP-статусов. Основные схемы, на которые следует обратить внимание, включают StudentRequest (структура данных для создания или обновления информации о студенте), StudentResponse (ответ с детальной информацией о студенте, включая список освоенного ПО), SoftwareRequest (структура данных для добавления программного обеспечения студенту) и SoftwareResponse (ответ с деталями о программном обеспечении, включая уровень изучения). В целом, эта спецификация представляет собой полный контракт API, описывающий взаимодействие клиентов с сервисом управления студентами и их освоенным программным обеспечением.

### 3.2.1 Технологический стек

Для разработки серверной части приложения использовались следующие библиотеки:

1. Django — это высокоуровневый фреймворк для разработки веб-приложений, написанный на Python.
2. Django Rest Framework — это дополнительный фреймворк для Django, который облегчает создание API. Он обеспечивает функциональность для сериализации данных (преобразования сложных типов данных в JSON или XML и обратно), создания маршрутов и обработчиков запросов.
3. Django CORS Headers — это приложение Django, которое добавляет необходимые заголовки CORS (Cross-Origin Resource Sharing) к ответам сервера.

### 3.2.2 Артефакты процесса разработки

В Листинге 3.2 представлено содержимое файла с описанием моделей базы данных с использованием Django ORM. Определена модель Student с единственным полем name (ФИО студента) и модель Software с полями name (название ПО) и study\_level (уровень освоения). Между студентами и ПО установлена связь ManyToManyField, позволяющая каждому студенту иметь несколько связанных программ, а программам – быть привязанными к нескольким студентам.

Листинг . — Содержимое файла models.py

|  |
| --- |
| from django.db import models  class Student(models.Model):  name = models.CharField('ФИО', max\_length=255)  def \_\_str\_\_(self):  return self.name  class Software(models.Model):  name = models.CharField('ПО',max\_length=255)  study\_level = models.CharField('Уровень', max\_length=50)  students = models.ManyToManyField(Student, related\_name='software\_list')  def \_\_str\_\_(self):  return f'{self.name} ({self.study\_level})' |

В Листинге 3.3 представлено содержимое файла, который реализует сериализаторы, преобразующие данные между объектами Python и форматом JSON. Определен SoftwareSerializer, включающий поля id, name и study\_level, а также StudentSerializer, который помимо id и name содержит software\_list – список изученного ПО, представленный в виде вложенного сериализатора.

Листинг . — Содержимое файла serializers.py

|  |
| --- |
| from rest\_framework import serializers  from .models import Student, Software  class SoftwareSerializer(serializers.ModelSerializer):  class Meta:  model = Software  fields = ['id', 'name', 'study\_level']  class StudentSerializer(serializers.ModelSerializer):  software\_list = SoftwareSerializer(many=True, read\_only=True)  class Meta:  model = Student  fields = ['id', 'name', 'software\_list'] |

В Листинге 3.4 представлено содержимое файла, который содержит контроллеры API для управления студентами и их списком изученного ПО. StudentViewSet реализует CRUD-операции над студентами, а метод destroy дополнительно очищает связи перед удалением. StudentSoftwareView позволяет получать список ПО студента (GET), добавлять новое ПО (POST) и удалять привязанное ПО (DELETE) по его id, проверяя корректность данных и существование объектов.

Листинг .4 — Содержимое файла views.py

|  |
| --- |
| from django.shortcuts import get\_object\_or\_404  from rest\_framework import viewsets, status  from rest\_framework.views import APIView  from rest\_framework.response import Response  from .models import Student, Software  from .serializers import StudentSerializer, SoftwareSerializer  class StudentViewSet(viewsets.ModelViewSet):  queryset = Student.objects.all()  serializer\_class = StudentSerializer  def destroy(self, request, \*args, \*\*kwargs):  student = self.get\_object()  student.software\_list.clear()  student.delete()  return Response(status=status.HTTP\_204\_NO\_CONTENT)  class StudentSoftwareView(APIView):  def get(self, request, student\_id):  student = get\_object\_or\_404(Student, pk=student\_id)  serializer = SoftwareSerializer(student.software\_list.all(), many=True)  return Response(serializer.data)  def post(self, request, student\_id):  student = get\_object\_or\_404(Student, pk=student\_id)  # Создаем новое ПО и связываем со студентом  serializer = SoftwareSerializer(data=request.data)  if not serializer.is\_valid():  return Response(serializer.errors, status=status.HTTP\_400\_BAD\_REQUEST)  software = serializer.save()  student.software\_list.add(software)  return Response(  {'status': 'ПО успешно создано и добавлено', 'data': serializer.data},  status=status.HTTP\_201\_CREATED  )  def delete(self, request, student\_id):  student = get\_object\_or\_404(Student, pk=student\_id)  if 'id' not in request.data:  return Response(  {'error': 'Необходимо указать id ПО для удаления'},  status=status.HTTP\_400\_BAD\_REQUEST  )  software = get\_object\_or\_404(Software, pk=request.data['id'])  student.software\_list.remove(software)  return Response(  {'status': 'ПО успешно удалено у студента'},  status=status.HTTP\_204\_NO\_CONTENT  ) |

В Листинге 3.5 представлено содержимое файла, который определяет маршрутизацию запросов для работы с API. Используется DefaultRouter для автоматического формирования URL-адресов для StudentViewSet, а также отдельно объявлен маршрут student/<int:student\_id>/software/, который привязывается к StudentSoftwareView и отвечает за управление ПО студента.

Листинг .5 — Содержимое файла myapp/urls.py

|  |
| --- |
| from django.urls import path  from rest\_framework.routers import DefaultRouter  from .views import StudentViewSet, StudentSoftwareView  router = DefaultRouter()  router.register(r'student', StudentViewSet, basename='student')  urlpatterns = [  path('student/<int:student\_id>/software/',  StudentSoftwareView.as\_view(),  name='student-software'),  \*router.urls  ] |

В Листинге 3.6 представлено содержимое файла, который определяет маршруты верхнего уровня для всего проекта. В нем настраивается доступ к административной панели Django через путь /admin/, а также включается маршрутизация API, объявленная в myapp/urls.py, по префиксу /api/. Это обеспечивает структурированное разделение маршрутов и удобное подключение новых модулей в будущем.

Листинг 3.6 — Содержимое файла kursovaya/urls.py

|  |
| --- |
| from django.contrib import admin  from django.urls import path, include  urlpatterns = [  path('admin/', admin.site.urls),  path('api/', include('myapp.urls')),  ] |

## 3.3 Тестирование серверной части приложения

Серверная часть тестировалась с помощью встроенного веб-сервера фреймворка и программы Postman. Тест-кейсы и ожидаемые результаты представлены в таблицах 3.5-3.11, а результаты тестирования - на Рисунках 3.1-3.7.

В рамках первого тест-кейса проверяется корректное добавление нового студента.

Таблица .5 — Тест-кейс успешного создания студента

|  |  |  |
| --- | --- | --- |
| Номер | 1 | |
| Название | Создание нового студента | |
| Входные данные | Ожидаемый результат | Результат |
| В теле запроса: данные нового студента (например, имя) | HTTP 201 статус код. Студент успешно создан | Успешно пройден |

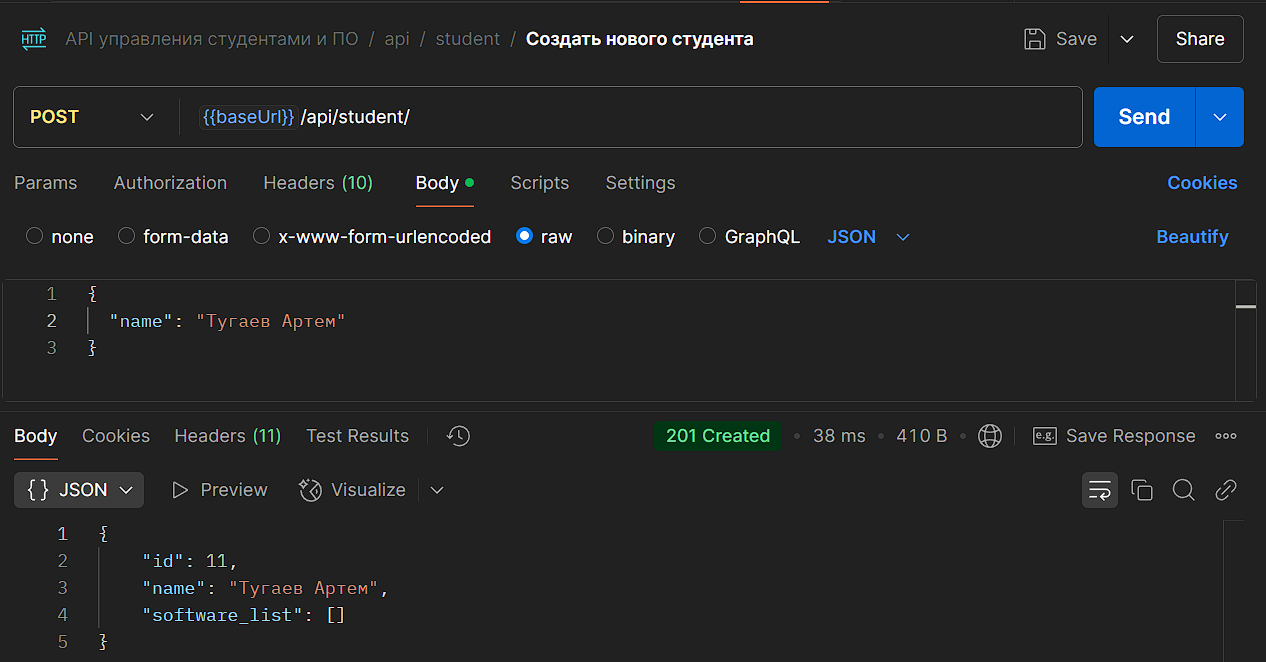


Рисунок .1 — Результат тест-кейса №1

В рамках второго тест-кейса проверяется корректное получение списка студентов.

Таблица .6 — Тест-кейс успешного получения списка студентов

|  |  |  |
| --- | --- | --- |
| Номер | 2 | |
| Название | Получение списка всех студентов | |
| Входные данные | Ожидаемый результат | Результат |
| Отсутствуют | HTTP 200 статус код. Возвращается список студентов в формате JSON | Успешно пройден |

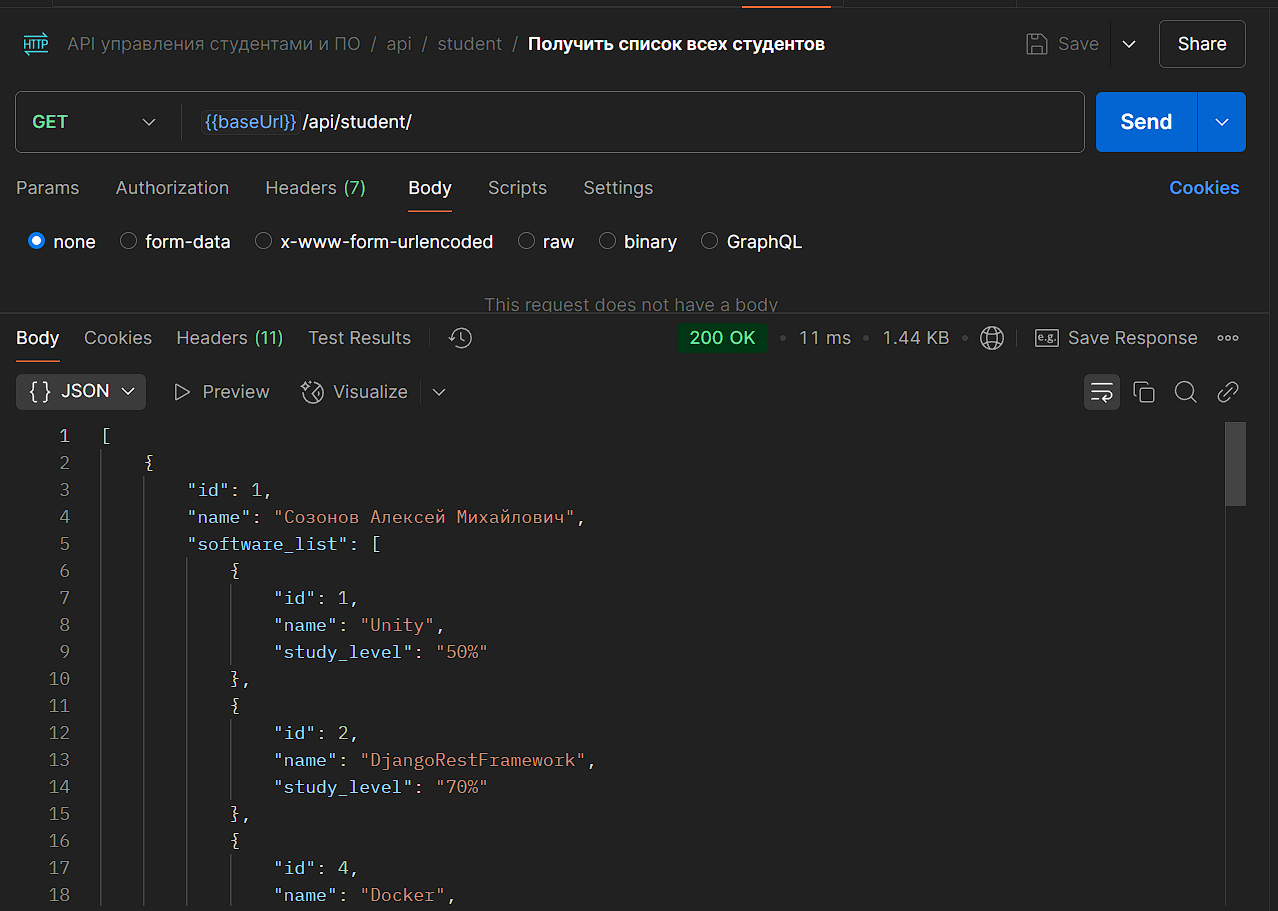


Рисунок .2 — Результат тест-кейса №2

В рамках третьего тест-кейса проверяется корректное получение информации о студенте.

Таблица .7 — Тест-кейс успешного получения информации о студенте

|  |  |  |
| --- | --- | --- |
| Номер | 3 | |
| Название | Получение информации о студенте | |
| Входные данные | Ожидаемый результат | Результат |
| Параметр student\_id в URL (например, 1) | HTTP 200 статус код. Возвращается информация о студенте в формате JSON | Успешно пройден |

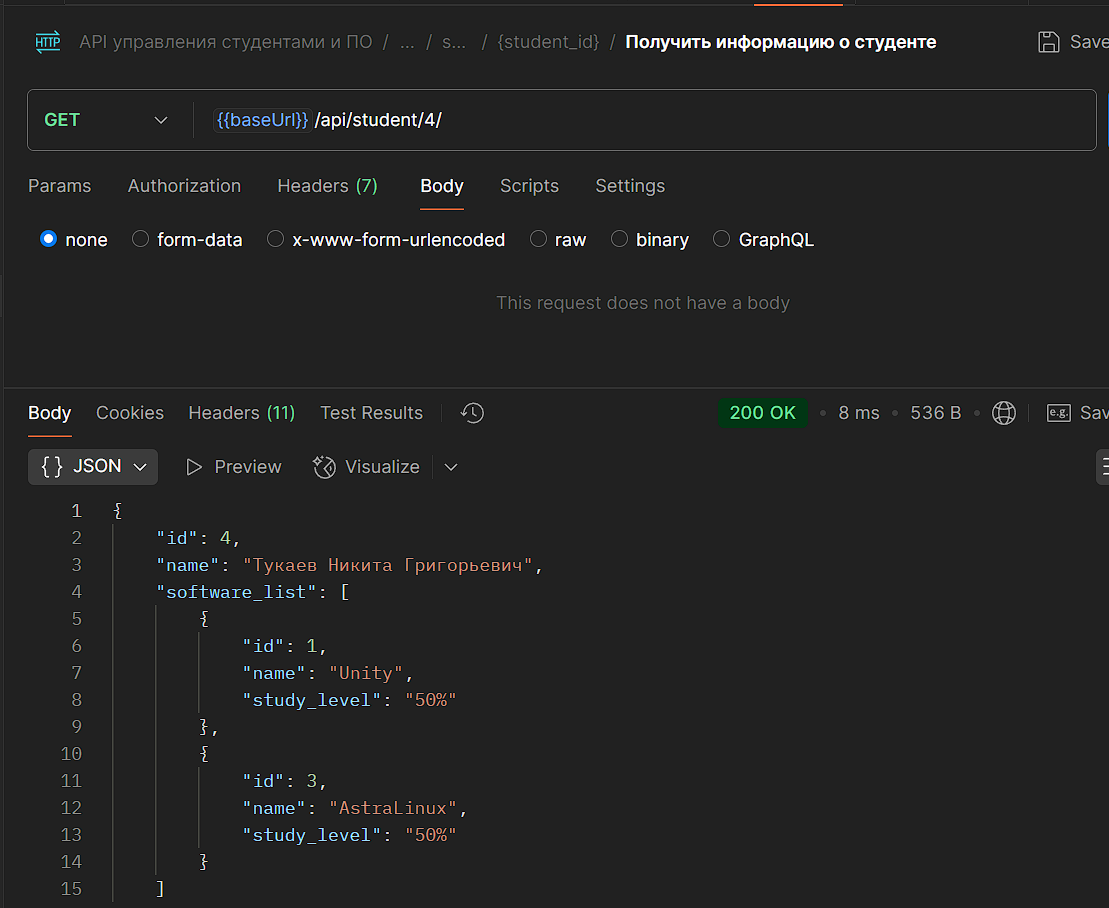


Рисунок . — Результат тест-кейса №3

В рамках четвертого тест-кейса проверяется корректное обновление информации о студенте.

Таблица .8 — Тест-кейс успешного обновления информации о студенте

|  |  |  |
| --- | --- | --- |
| Номер | 4 | |
| Название | Обновление информации о студенте | |
| Входные данные | Ожидаемый результат | Результат |
| Параметр student\_id в URL (например, 1), новые данные студента в теле запроса | HTTP 200 статус код. Студент успешно обновлен | Успешно пройден |

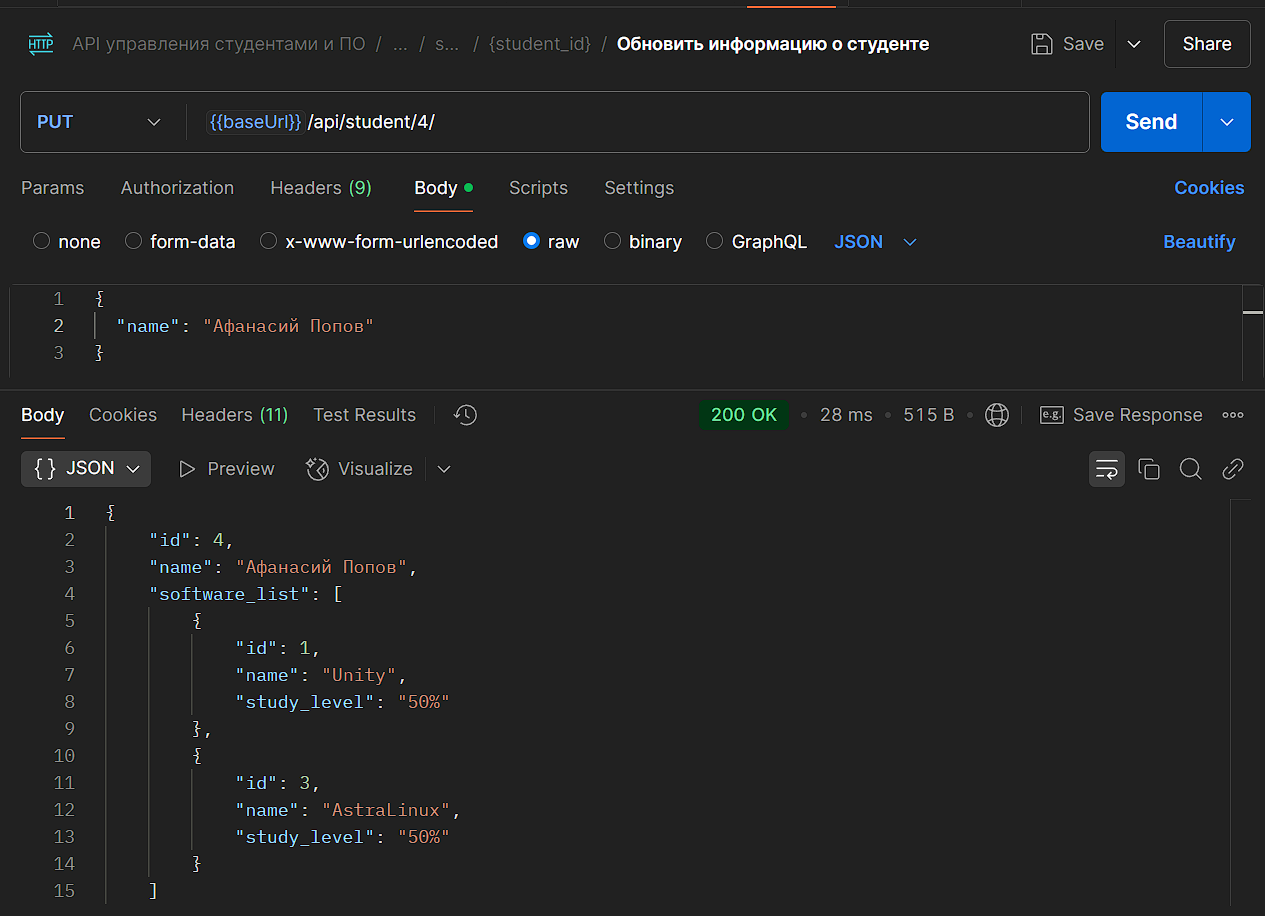


Рисунок .4 — Результат тест-кейса №4

В рамках пятого тест-кейса проверяется корректное удаление студента.

Таблица .9 — Тест-кейс успешного удаления студента

|  |  |  |
| --- | --- | --- |
| Номер | 5 | |
| Название | Удаление студента | |
| Входные данные | Ожидаемый результат | Результат |
| Параметр student\_id в URL (например, 1) | HTTP 204 статус код. Студент успешно удален | Успешно пройден |

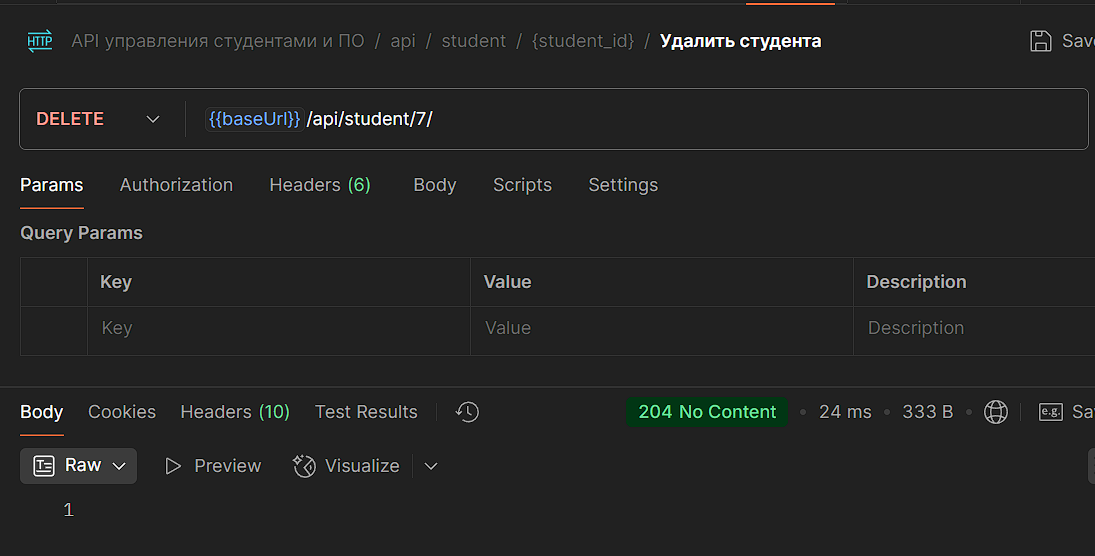


Рисунок .5 — Результат тест-кейса №5

В рамках шестого тест-кейса проверяется корректное добавление нового ПО для студента.

Таблица .10 — Тест-кейс успешного добавления ПО студенту

|  |  |  |
| --- | --- | --- |
| Номер | 6 | |
| Название | Добавление ПО студенту | |
| Входные данные | Ожидаемый результат | Результат |
| Параметр student\_id в URL (например, 1), данные ПО в теле запроса (например, имя ПО и уровень освоения) | HTTP 201 статус код. ПО успешно добавлено студенту | Успешно пройден |

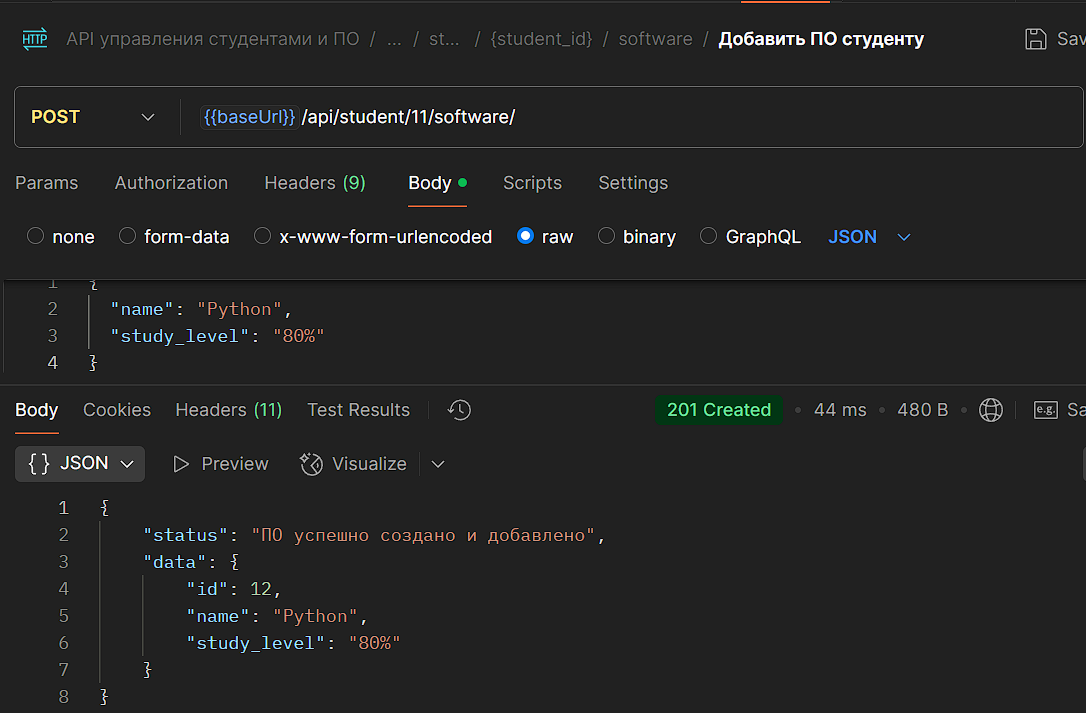


Рисунок .6 — Результат тест-кейса №6

В рамках седьмого тест-кейса проверяется корректное получение списка ПО студента.

Таблица .11 — Тест-кейс успешного получения списка ПО студента

|  |  |  |
| --- | --- | --- |
| Номер | 7 | |
| Название | Получение списка ПО студента | |
| Входные данные | Ожидаемый результат | Результат |
| Параметр student\_id в URL (например, 1) | HTTP 200 статус код. Возвращается список ПО студента в формате JSON | Успешно пройден |

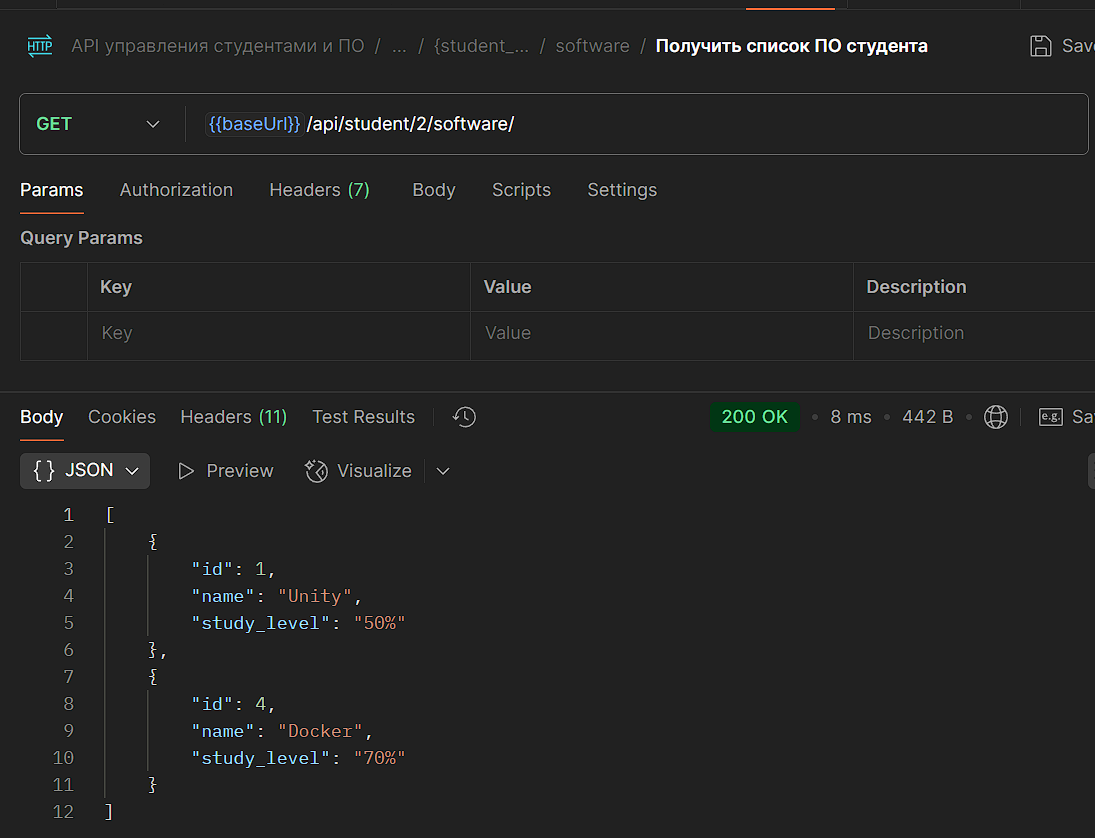


Рисунок .7 — Результат тест-кейса №7

Таким образом, была реализована и протестирована серверная часть приложения.

# 4. РАЗВЕРТЫВАНИЕ СЕРВЕРНОЙ ЧАСТИ ПРИЛОЖЕНИЯ «СОЗДАНИЕ, ХРАНЕНИЕ И ОБРАБОТКА ДАННЫХ О ПРОГРАММНОМ ОБЕСПЕЧЕНИИ, ИЗУЧЕННОМ СТУДЕНТОМ ЗА ТЕКУЩИЙ СЕМЕСТР»

## 4.1 Описание процесса развертывания

Для развертывания серверной части приложения необходимо два Docker-контейнера. Первый будет отвечать за прокси-сервер nginx, через который будет передавать все запросы на сервер, а второй за саму серверную часть приложения.

Для работы прокси-сервера была написана конфигурация (Листинг 4.1).

Листинг . — Конфигурация прокси-сервера nginx

|  |
| --- |
| server {  listen 80;  server\_name localhost;  # Корневая директория  root /usr/share/nginx/html;  index index.html;  # API  location /api/ {  proxy\_pass http://django:8000;  proxy\_set\_header Host $host;  proxy\_set\_header X-Real-IP $remote\_addr;  proxy\_set\_header X-Forwarded-For $proxy\_add\_x\_forwarded\_for;  proxy\_connect\_timeout 300s;  proxy\_read\_timeout 300s;  }  # Статика Django  location /static/ {  alias /app/staticfiles/;  expires 30d;  access\_log off;  }  } |

Далее была описана спецификация запуска прокси-сервера в виде Docker-контейнера (Листинг 4.2).

Листинг . — Спецификация запуска прокси-сервера в виде Docker-контейнера

|  |
| --- |
| FROM nginx:1.25-alpine  COPY ./nginx.conf /etc/nginx/conf.d/default.conf |

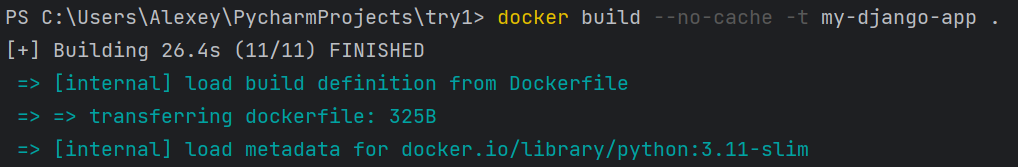
Для запуска Django приложения используется gunicorn. Также была описана спецификация сборки серверной части приложения в виде Docker-контейнера (Листинг 4.3).

Листинг .3 — Спецификация сборки серверной части приложения в виде Docker-контейнера

|  |
| --- |
| FROM python:3.11-slim  WORKDIR /app  COPY requirements.txt .  RUN pip install --no-cache-dir -r requirements.txt  COPY . .  RUN mkdir -p /app/staticfiles  CMD ["sh", "-c", "python manage.py collectstatic --no-input && gunicorn --bind 0.0.0.0:8000 kursovayaapp.wsgi:application"] |

## 4.2 Тестирование развертывания

Сборка образа из написанного Dockerfile. Результат представлен на Рисунке 4.1.

 Рисунок 4. - Сборка образа для серверной части приложения

После успешной сборки образа запустим контейнер с серверной частью приложения (Рисунок 4.2).



Рисунок 4. - Запуск контейнера с серверной частью приложения

Теперь нужно собрать образ для прокси-сервера nginx. Успешная сборка представлена на Рисунке 4.3.



Рисунок 4. - Сборка образа nginx

После успешной сборки образа происходит запуск контейнера. Запуск представлен на Рисунке 4.4.

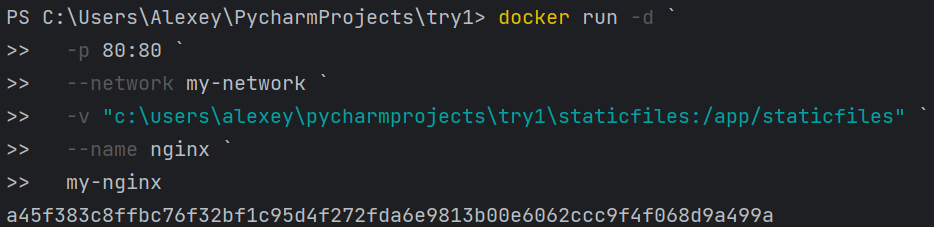


Рисунок 4.4 – Запуск контейнера с nginx

Таким образом, были развернуты и запущены два Docker-контейнера с серверной частью приложения и сервером nginx.

# 5. РЕАЛИЗАЦИЯ КЛИЕНТСКОЙ ЧАСТИ ПРИЛОЖЕНИЯ «СОЗДАНИЕ, ХРАНЕНИЕ И ОБРАБОТКА ДАННЫХ О ПРОГРАММНОМ ОБЕСПЕЧЕНИИ, ИЗУЧЕННОМ СТУДЕНТОМ ЗА ТЕКУЩИЙ СЕМЕСТР»

## 5.1 Технологический стек

Для реализации клиентской части приложения использовался стандартный набор веб-технологий, включающий HTML, CSS и JavaScript.

HTML предоставляет основу для структуры веб-страницы, включая создание основных элементов, таких как заголовки, формы, кнопки и текстовые поля, которые организуют интерфейс и обеспечивают взаимодействие с пользователем.

CSS применяется для стилизации этих элементов, настраивая их внешний вид, включая цвета, шрифты, размеры и расположение на странице, что делает интерфейс более привлекательным и удобным для пользователей.

JavaScript играет ключевую роль в добавлении интерактивности и динамического поведения на странице. Он используется для обработки событий, таких как клики по кнопкам или отправка форм, а также для асинхронного взаимодействия с сервером. Взаимодействие с сервером происходит с использованием fetch, что позволяет отправлять и получать данные через API, выполняя операции добавления, удаления и редактирования данных студентов и их программного обеспечения без необходимости перезагружать страницу.

## 5.2 Описание реализации

В реализации клиентской части основное внимание уделено созданию удобного и интуитивно понятного интерфейса для работы с данными студентов и их программного обеспечения. HTML-разметка определяет ключевые элементы страницы, такие как форма для добавления нового студента, список студентов с кнопками для взаимодействия (удаление, редактирование, просмотр), а также раздел для работы с программным обеспечением студентов. JavaScript используется для управления поведением этих элементов и для обработки взаимодействий с пользователем. Например, кнопка "Добавить студента" открывает форму для ввода данных, а при отправке формы данные передаются на сервер для добавления нового студента в базу данных. Для обновления данных на странице без перезагрузки используется fetch, который выполняет запросы к серверу для получения списка студентов или их программного обеспечения, а также для выполнения CRUD-операций, таких как добавление, удаление и редактирование. В результате, пользователь получает динамичный интерфейс, который всегда отображает актуальную информацию, обновляя данные в реальном времени.

## 5.3 Описание процесса взаимодействия конечного пользователя с клиентом

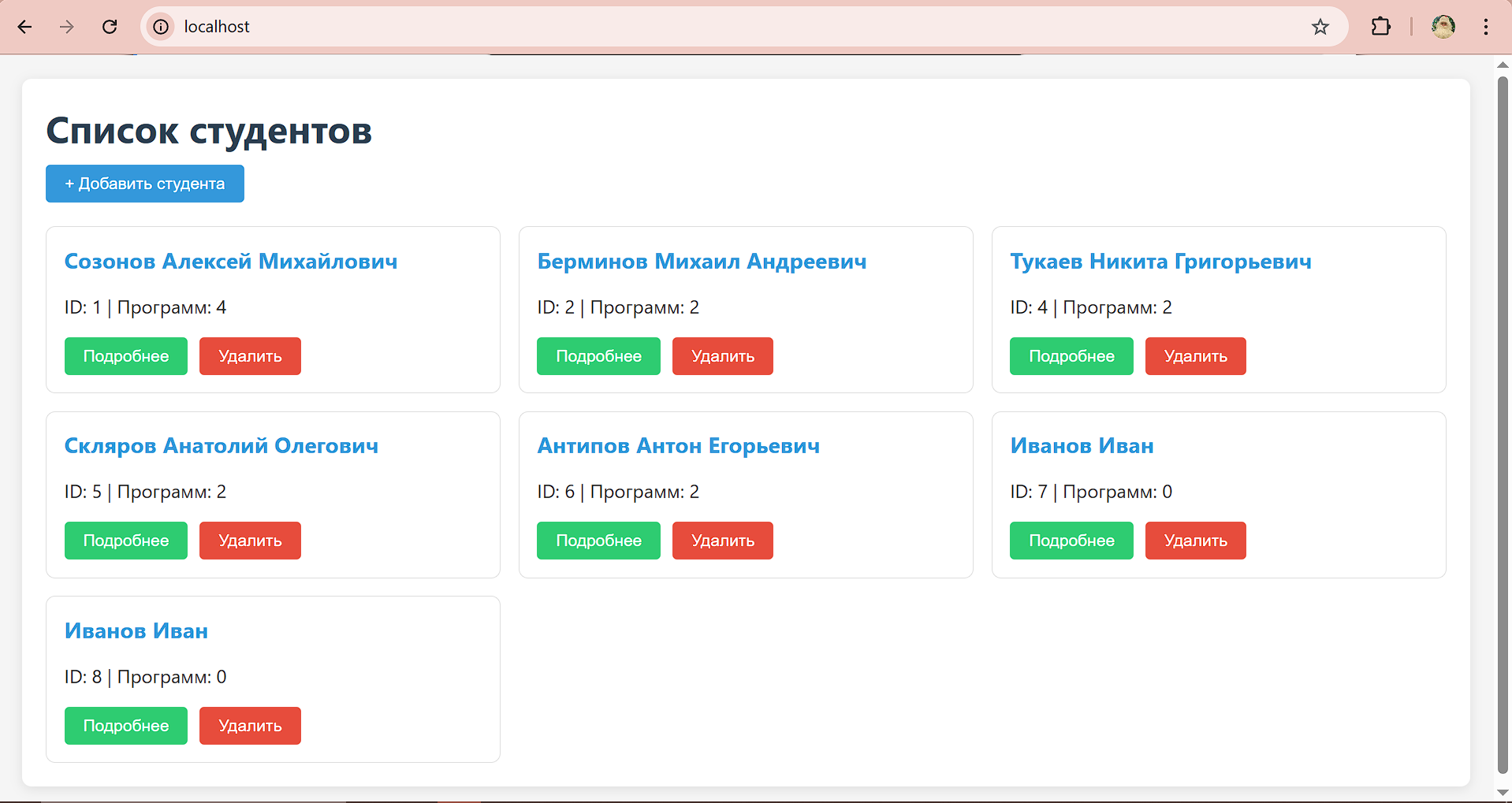
На Рисунках 5.1-5.2 представлен интерфейс клиентской части приложения. Пользователь может просматривать список студентов, добавлять новых, удалять существующих, а также переходить к детальной информации о каждом студенте. На странице подробностей отображаются данные о студенте, включая его имя и идентификатор, а также перечень используемого программного обеспечения с указанием уровня владения. Доступны функции добавления нового программного обеспечения, а также удаления уже существующего.

Рисунок . — Интерфейс страницы списка студентов

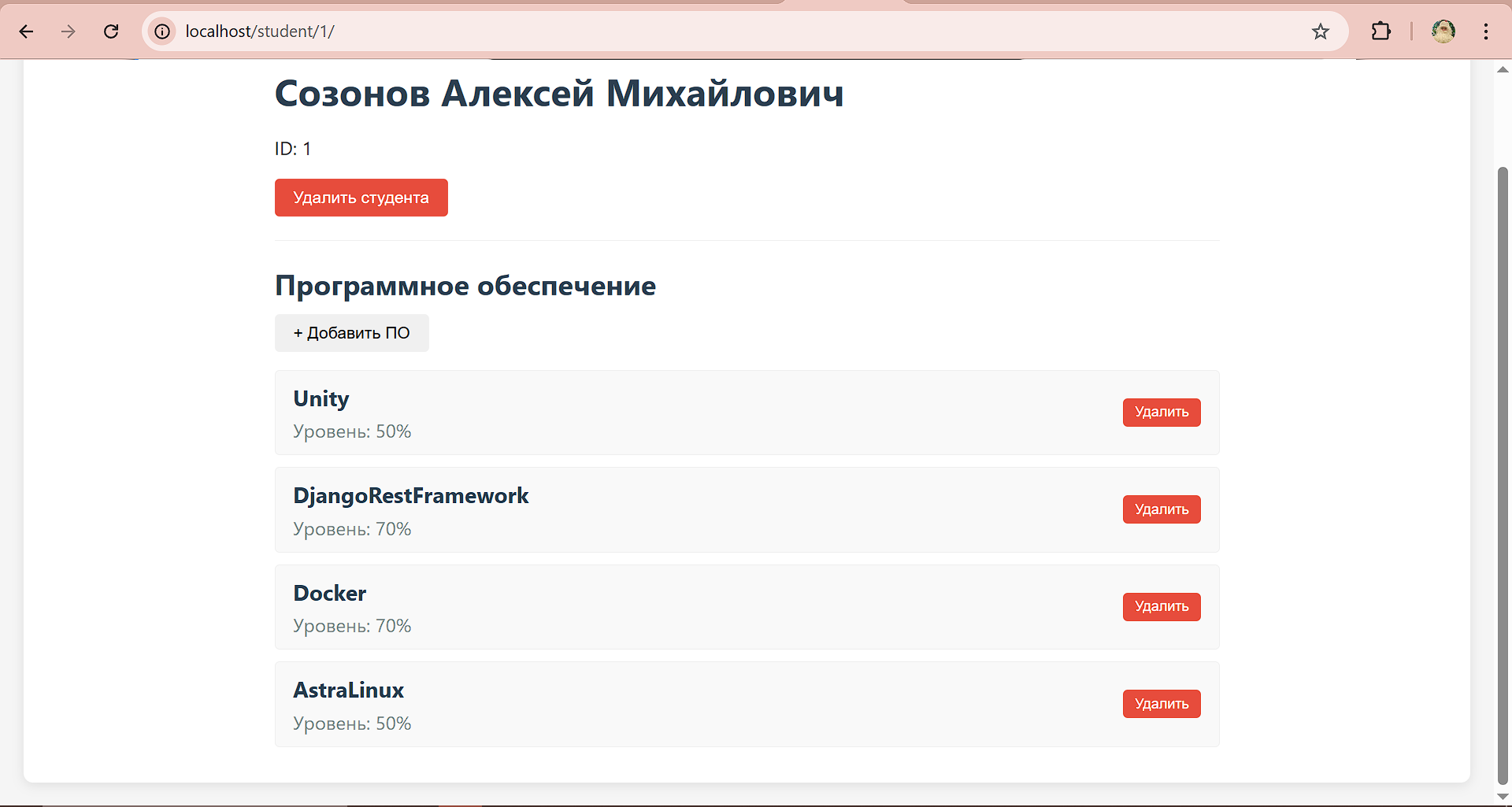


Рисунок . — Интерфейс страницы списка ПО студента

Ниже приведены пользовательские истории, позволяющие взглянуть на функционал продукта с точки зрения конечного пользователя.

Как пользователь я хочу видеть список всех студентов с их основными данными, чтобы быстро получать информацию о каждом студенте.

Как пользователь я хочу добавить нового студента в систему, чтобы учитывать его в базе данных и назначать ему программное обеспечение.

Как пользователь я хочу удалить студента из системы, чтобы исключить его данные из базы, если он больше не нужен.

Как пользователь я хочу открыть страницу студента, чтобы видеть его данные и список освоенного программного обеспечения.

Как пользователь я хочу добавлять студенту программное обеспечение, чтобы фиксировать его навыки в системе.

Таким образом, разработанная клиентская часть приложения обеспечивает удобное взаимодействие пользователя с системой, а также позволяет эффективно управлять студентами и их навыками в области программного обеспечения.

# 6. РАЗВЕРТЫВАНИЕ КЛИЕНТСКОЙ ЧАСТИ ПРИЛОЖЕНИЯ «СОЗДАНИЕ, ХРАНЕНИЕ И ОБРАБОТКА ДАННЫХ О ПРОГРАММНОМ ОБЕСПЕЧЕНИИ, ИЗУЧЕННОМ СТУДЕНТОМ ЗА ТЕКУЩИЙ СЕМЕСТР»

## 6.1 Описание процесса развертывания

Для развертывания клиентской части приложения была описана спецификация сборки в виде Docker-контейнера (Листинг 6.1).

Листинг . — Спецификация сборки клиентской части приложения в виде Docker-контейнера

|  |
| --- |
| FROM nginx:1.25-alpine  RUN apk add --no-cache curl  RUN rm /etc/nginx/conf.d/default.conf  COPY nginx.conf /etc/nginx/conf.d/  RUN mkdir -p /app/staticfiles /usr/share/nginx/html  EXPOSE 80  CMD ["nginx", "-g", "daemon off;"] |

На этапе сборки образа статические файлы перебрасываются в соответствующую директорию nginx сервера. Сам контейнер отвечает за то, чтобы отдавать эти страницы клиенту.

## 6.2 Тестирование развертывания

Проверим сборку образа для клиентской части приложения. Результат представлен на Рисунке 6.1.

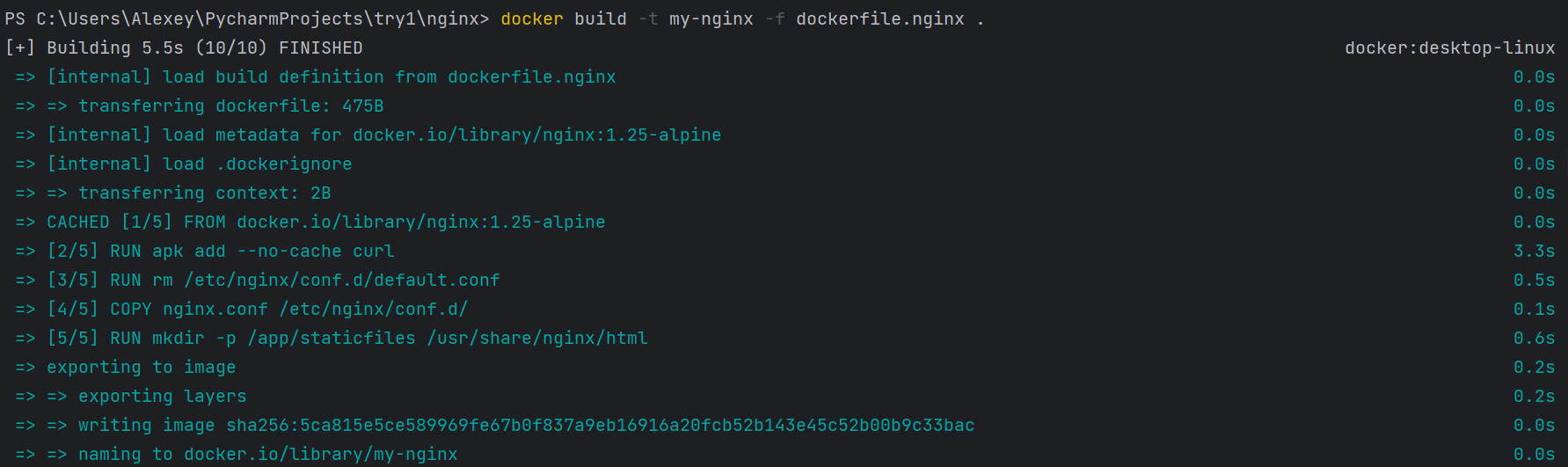


Рисунок 6.1 - Сборка образа для клиентской части приложения

Теперь запустим контейнер с успешно собранным образом клиентской части (Рисунок 6.2).



Рисунок 6.2 - Запуск контейнера с клиентской частью приложения

В ходе развертывания клиентской части приложения была создана спецификация сборки в виде Docker-контейнера на базе Nginx, обеспечивающего раздачу статических файлов. В процессе тестирования успешно выполнена сборка образа, после чего контейнер был запущен и проверен на корректность работы.

# 7. НАСТРОЙКА КОНФИГУРАЦИИ МНОГОКОНТЕЙНЕРНОГО РАЗВЕРТЫВАНИЯ ПРИЛОЖЕНИЯ «СОЗДАНИЕ, ХРАНЕНИЕ И ОБРАБОТКА ДАННЫХ О ПРОГРАММНОМ ОБЕСПЕЧЕНИИ, ИЗУЧЕННОМ СТУДЕНТОМ ЗА ТЕКУЩИЙ СЕМЕСТР»

Запуск приложения целиком производится с использованием технологии Docker Compose. Разработанная спецификация представлена в Листинге 7.1.

Листинг . — Спецификация запуска многоконтейнерного приложения

|  |
| --- |
| services:  backend:  build:  context: .  dockerfile: Dockerfile  container\_name: backend  volumes:  - ./staticfiles:/app/staticfiles  - ./media:/app/media  environment:  - DEBUG=1  - ALLOWED\_HOSTS=localhost backend nginx  - CORS\_ALLOWED\_ORIGINS=http://localhost  networks:  - app\_network  command: >  sh -c "python manage.py collectstatic --no-input &&  gunicorn --bind 0.0.0.0:8000 kursovayaapp.wsgi:application"  nginx:  build:  context: ./nginx  dockerfile: dockerfile.nginx  container\_name: nginx  ports:  - "80:80"  volumes:  - ./staticfiles:/app/staticfiles  - ./nginx/nginx.conf:/etc/nginx/conf.d/default.conf  - ./front:/usr/share/nginx/html  depends\_on:  - backend  networks:  - app\_network  frontend:  build:  context: ./front  dockerfile: Dockerfile  container\_name: frontend  volumes:  - ./front:/usr/share/nginx/html  networks:  - app\_network  networks:  app\_network:  driver: bridge |

Запуск всего приложения осуществляется с использованием Docker Compose, что позволяет управлять многоконтейнерной архитектурой. В составе приложения работают три основных сервиса: backend, nginx и frontend, каждый из которых выполняет свою роль. Сервис backend отвечает за обработку серверной логики, использует Django и запускается с Gunicorn, предоставляя API на порту 8000. Он монтирует локальные директории для хранения статических и медиафайлов, а также настраивается с переменными окружения, определяющими параметры работы. Сервис nginx выполняет роль прокси-сервера, перенаправляя запросы к серверу Django и раздавая статические файлы. Он загружается из отдельного Docker-образа и использует конфигурационный файл для правильной маршрутизации запросов. Сервис frontend обрабатывает клиентскую часть приложения, предоставляя пользователям доступ к веб-интерфейсу. Все сервисы соединены в единую сеть app\_network, что обеспечивает их взаимодействие внутри контейнерной среды. Такая структура позволяет запускать и управлять приложением в изолированной среде, упрощая развертывание и поддержку

На Рисунках 7.1, 7.2 представлено создание и запуск контейнеров с помощью Docker Compose.

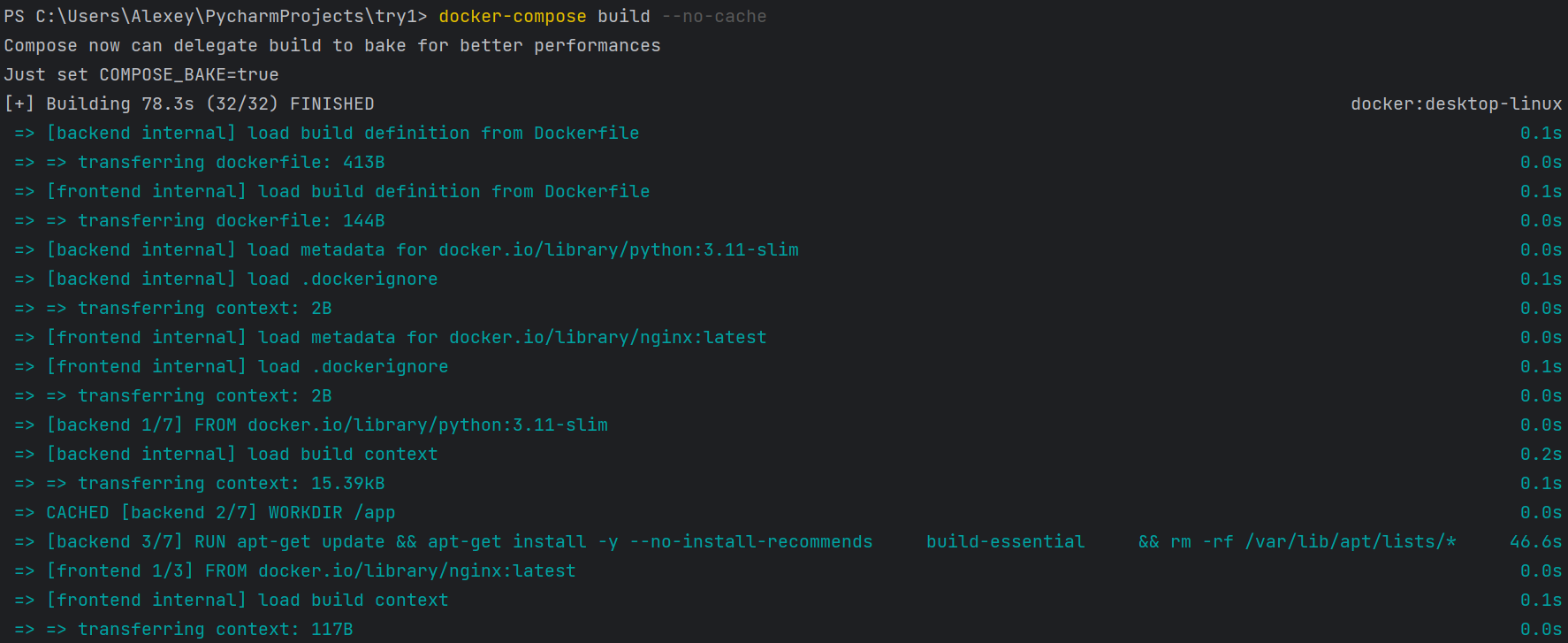


Рисунок 7.1 – Создание и запуск контейнеров ч.1

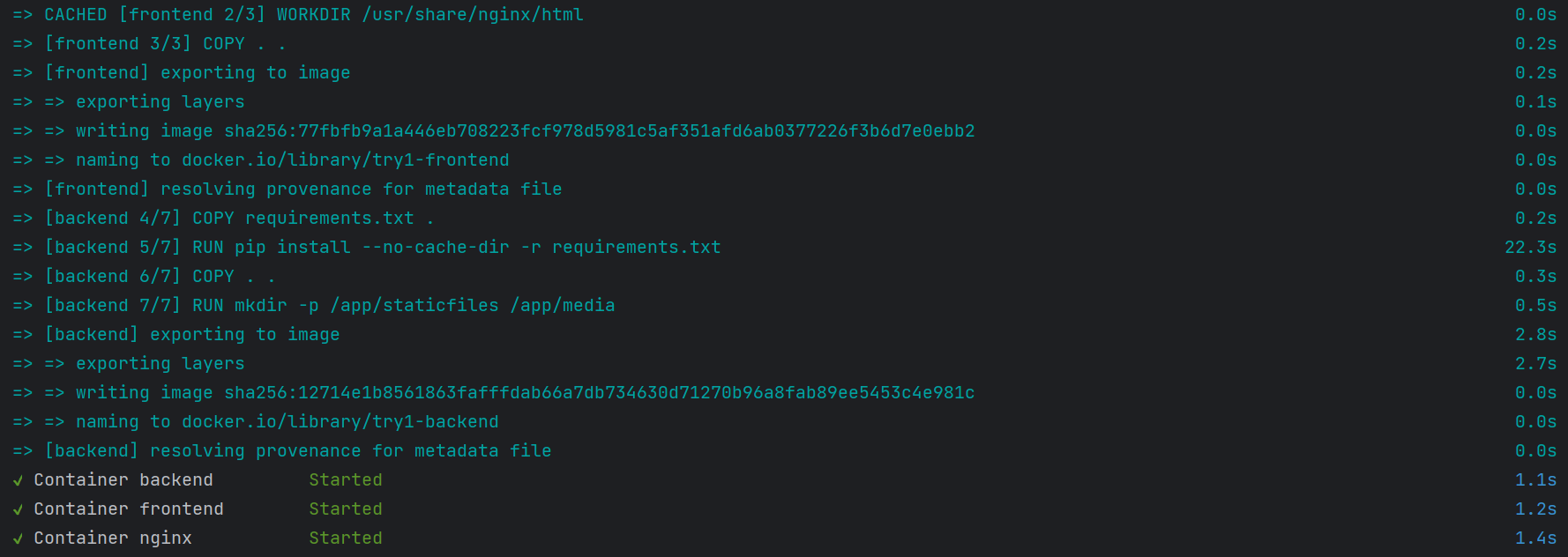


Рисунок 7.2 – Создание и запуск контейнеров ч.2

Использование Docker Compose для многоконтейнерного развертывания приложения обеспечивает эффективное управление компонентами системы, такими как backend, nginx и frontend. Это упрощает развертывание, изоляцию и масштабирование приложения, обеспечивая стабильную работу и удобство в поддержке.

# ЗАКЛЮЧЕНИЕ

В ходе выполнения курсовой работы было разработано приложение для создания, хранения и обработки данных о программном обеспечении, освоенного студентами в течение семестра, реализованное с использованием современных веб-технологий.

На серверной стороне применен фреймворк Django REST Framework, обеспечивающий надежное API с полным набором CRUD-операций, что позволяет управлять информацией о студентах и их навыках работы с различным ПО.

Nginx выступает в роли высокопроизводительного обратного прокси, оптимизирующего обработку запросов и раздачу статических файлов.

Клиентская часть, построенная на HTML, CSS и JavaScript с использованием Fetch API, предоставляет интуитивно понятный интерфейс с возможностями динамического обновления контента без перезагрузки страницы.

Особое внимание уделено архитектуре развертывания - система контейнеризирована с помощью Docker, где каждый компонент (веб-сервер Nginx, Django-приложение и фронтенд) работает в изолированной среде, что обеспечивает стабильность работы, легкую масштабируемость и простоту развертывания на различных платформах.

Реализованное решение соответствует поставленным задачам и обладает потенциалом для дальнейшего развития.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Методические указания по выполнению КР\_Архитектура интеграции и развертывания — URL: <https://online-edu.mirea.ru/pluginfile.php?file=%2F1233502%2Fmod_resource%2Fcontent%2F1%2FАрхитектура_интеграции_и_развертывания_Курсовой_проект_Методические%20указания.pdf> (дата обращения 12.03.2024)
2. Django documentation [Электронный ресурс]. — URL: <https://docs.djangoproject.com/en/5.0/> (дата обращения 23.03.2024)
3. Getting started with Django [Электронный ресурс]. — URL: <https://www.djangoproject.com/start/> (дата обращения 23.03.2024)
4. Nginx: документация [Электронный ресурс]. — URL: <https://nginx.org/ru/docs/> (дата обращения 25.03.2024)