Отчет по лабораторным работам

Архитектура и дизайн программного обеспечения

Группа: 231-329
Студент: Фамилия И.О.
Дата: 06.06.2025

Содержание

Введение

Данный отчет содержит описание выполнения трех лабораторных работ по дисциплине "Архитектура и дизайн программного обеспечения". Целью лабораторных работ является практическое освоение различных архитектурных подходов к разработке программного обеспечения, включая трёхуровневую клиент-серверную архитектуру, контейнеризацию и разработку клиентской части приложения.

В ходе выполнения лабораторных работ были решены следующие задачи: 1. Разработка трёхуровневой клиент-серверной архитектуры с использованием Django и Django REST Framework 2. Контейнеризация разработанного приложения с использованием Docker и Docker Compose 3. Разработка клиентской части приложения с использованием Qt

Все лабораторные работы связаны между собой и представляют собой единый проект - систему управления каталогом книг. Первая лабораторная работа посвящена разработке серверной части приложения, вторая - контейнеризации этой серверной части, а третья - разработке клиентской части, которая взаимодействует с сервером.

Лабораторная работа №1: Трёхуровневая клиент-серверная архитектура

Цель работы

Целью данной лабораторной работы является разработка трёхуровневой клиент-серверной архитектуры с использованием Django и Django REST Framework. Трёхуровневая архитектура включает в себя уровень представления (клиентская часть), уровень бизнес-логики (серверная часть) и уровень данных (база данных).

Теоретические сведения

Трёхуровневая архитектура - это архитектурный паттерн, который разделяет приложение на три логических и физических уровня:

1. **Уровень представления (Presentation Tier)** - отвечает за пользовательский интерфейс и взаимодействие с пользователем. В нашем случае это будет REST API, который будет использоваться клиентскими приложениями.

2. **Уровень бизнес-логики (Business Logic Tier)** - отвечает за обработку данных, реализацию бизнес-правил и логики приложения. В нашем случае это будет Django-приложение с моделями и представлениями.

3. **Уровень данных (Data Tier)** - отвечает за хранение и доступ к данным. В нашем случае это будет база данных PostgreSQL.

Преимущества трёхуровневой архитектуры: - Разделение ответственности между компонентами - Возможность независимой разработки и масштабирования каждого уровня - Повышение безопасности за счет изоляции уровня данных - Улучшение поддерживаемости и тестируемости приложения

Django REST Framework (DRF) - это мощный и гибкий инструмент для создания Web API на основе Django. Он предоставляет сериализацию, которая поддерживает как ORM, так и не-ORM источники данных, и использует представления на основе классов, которые являются мощным и простым в настройке способом построения API.

Используемые технологии

Для выполнения лабораторной работы были использованы следующие технологии:

- **Python 3.11** - язык программирования

- **Django 5.0** - веб-фреймворк для разработки серверной части
- **Django REST Framework 3.14** - расширение Django для создания REST API
- **PostgreSQL 14** - реляционная база данных
- **Gunicorn** - WSGI HTTP-сервер для запуска Django-приложения
- **Faker** - библиотека для генерации тестовых данных

Описание реализации

Структура проекта

Проект имеет следующую структуру:

```
lab1/
├── lab1/                   # Основной пакет Django-проекта
│   ├── __init__.py
│   ├── asgi.py
│   ├── settings.py         # Настройки проекта
│   ├── urls.py             # Маршрутизация URL
│   └── wsgi.py
├── mainapp/                # Приложение для работы с книгами
│   ├── __init__.py
│   ├── admin.py            # Настройки административной панели
│   ├── apps.py
│   ├── gentestdata.py      # Скрипт для генерации тестовых
данных
│   ├── migrations/         # Миграции базы данных
│   ├── models.py           # Модели данных
│   ├── serializers.py      # Сериализаторы для API
│   ├── tests.py
│   └── views.py            # Представления API
├── api_tests.http          # Файл с тестовыми запросами к API
├── manage.py               # Скрипт управления Django-проектом
├── README.md               # Документация проекта
└── requirements.txt        # Зависимости проекта
```

Модель данных

В качестве модели данных была выбрана сущность "Книга" (Book) со следующими полями:

```python
class Book(models.Model):
    GENRE_CHOICES = [
        ('fiction', 'Художественная литература'),
        ('non_fiction', 'Нехудожественная литература'),
        ('science', 'Научная литература'),
        ('fantasy', 'Фэнтези'),
        ('sci_fi', 'Научная фантастика'),
```

```
        ('detective', 'Детектив'),
        ('romance', 'Романтика'),
        ('horror', 'Ужасы'),
        ('other', 'Другое'),
    ]

    title = models.CharField(max_length=200,
verbose_name="Название")
    author = models.CharField(max_length=100,
verbose_name="Автор")
    publication_year = models.IntegerField(verbose_name="Год
издания")
    genre = models.CharField(max_length=20,
choices=GENRE_CHOICES, default='other', verbose_name="Жанр")
    pages =
models.PositiveIntegerField(verbose_name="Количество страниц")
    rating = models.DecimalField(max_digits=3, decimal_places=1,
default=0.0, verbose_name="Рейтинг")
    is_bestseller = models.BooleanField(default=False,
verbose_name="Бестселлер")
    created_at = models.DateTimeField(auto_now_add=True,
verbose_name="Дата создания")
    updated_at = models.DateTimeField(auto_now=True,
verbose_name="Дата обновления")

    def __str__(self):
        return f"{self.title} ({self.author})"

    class Meta:
        verbose_name = "Книга"
        verbose_name_plural = "Книги"
        ordering = ['-created_at']
```

Модель содержит 9 полей различных типов: - `title` (CharField) - название книги - `author` (CharField) - автор книги - `publication_year` (IntegerField) - год издания - `genre` (CharField с выбором) - жанр книги - `pages` (PositiveIntegerField) - количество страниц - `rating` (DecimalField) - рейтинг книги - `is_bestseller` (BooleanField) - является ли книга бестселлером - `created_at` (DateTimeField) - дата создания записи - `updated_at` (DateTimeField) - дата обновления записи

Сериализатор

Для преобразования объектов модели в JSON и обратно был создан сериализатор:

```
class BookSerializer(serializers.ModelSerializer):
    class Meta:
```

```
        model = Book
        fields = '__all__'
```

Сериализатор использует все поля модели Book и автоматически преобразует их в соответствующие типы данных JSON.

Представления API

Для обработки HTTP-запросов были созданы представления на основе ViewSet:

```
class BookViewSet(viewsets.ModelViewSet):
    queryset = Book.objects.all()
    serializer_class = BookSerializer
    filter_backends = [DjangoFilterBackend,
filters.SearchFilter, filters.OrderingFilter]
    filterset_fields = ['genre', 'publication_year',
'is_bestseller']
    search_fields = ['title', 'author']
    ordering_fields = ['title', 'author', 'publication_year',
'rating', 'created_at']
```

ViewSet автоматически создает обработчики для всех стандартных операций CRUD (Create, Read, Update, Delete) и предоставляет дополнительные возможности фильтрации, поиска и сортировки.

Маршрутизация URL

Для маршрутизации URL был использован DefaultRouter из Django REST Framework:

```
from django.contrib import admin
from django.urls import path, include
from rest_framework import routers
from mainapp.views import BookViewSet

router = routers.DefaultRouter()
router.register(r'books', BookViewSet)

urlpatterns = [
    path('admin/', admin.site.urls),
    path('api/', include(router.urls)),
    path('api-auth/', include('rest_framework.urls')),
]
```

Router автоматически создает URL-маршруты для всех действий ViewSet.

Настройка базы данных

Для хранения данных была использована база данных PostgreSQL. Настройки подключения к базе данных:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'lab1_db',
        'USER': 'django',
        'PASSWORD': '2NevjFoQisyVaV',
        'HOST': 'localhost',
        'PORT': '5432',
    }
}
```

Генерация тестовых данных

Для заполнения базы данных тестовыми данными был создан скрипт `gentestdata.py`:

```
import random
import datetime
from django.core.management.base import BaseCommand
from mainapp.models import Book
from faker import Faker

class Command(BaseCommand):
    help = 'Generates test data for Book model'

    def handle(self, *args, **kwargs):
        fake = Faker()

        # Удаляем все существующие книги
        Book.objects.all().delete()

        # Создаем 100 случайных книг
        for i in range(1, 101):
            Book.objects.create(
                title=fake.catch_phrase(),
                author=fake.name(),
                publication_year=random.randint(1900, 2025),
                genre=random.choice([x[0] for x in
Book.GENRE_CHOICES]),
                pages=random.randint(50, 1000),
                rating=round(random.uniform(0, 10), 1),
                is_bestseller=random.choice([True, False])
            )
```

```
        self.stdout.write(self.style.SUCCESS('Successfully
generated 100 books'))
```

Скрипт использует библиотеку Faker для генерации случайных данных и создает 100 книг с различными параметрами.

Тестирование API

Для тестирования API был создан файл `api_tests.http` с тестовыми запросами:

```
@hostname = http://localhost:8000

### Получение списка всех книг
GET {{hostname}}/api/books/
Content-Type: application/json

### Получение информации о конкретной книге
GET {{hostname}}/api/books/1/
Content-Type: application/json

### Создание новой книги
POST {{hostname}}/api/books/
Content-Type: application/json

{
    "title": "Новая книга",
    "author": "Иван Иванов",
    "publication_year": 2023,
    "genre": "fiction",
    "pages": 300,
    "rating": 8.5,
    "is_bestseller": true
}

### Обновление информации о книге
PUT {{hostname}}/api/books/1/
Content-Type: application/json

{
    "title": "Обновленная книга",
    "author": "Иван Иванов",
    "publication_year": 2023,
    "genre": "fiction",
    "pages": 300,
    "rating": 9.0,
    "is_bestseller": true
}

### Частичное обновление информации о книге
```

```
PATCH {{hostname}}/api/books/1/
Content-Type: application/json

{
    "rating": 9.5
}

### Удаление книги
DELETE {{hostname}}/api/books/1/
```

Тестирование показало, что API работает корректно и обрабатывает все типы запросов.

Выводы

В результате выполнения лабораторной работы была разработана трёхуровневая клиент-серверная архитектура с использованием Django и Django REST Framework. Созданное приложение предоставляет REST API для работы с каталогом книг и может быть использовано различными клиентскими приложениями.

Трёхуровневая архитектура обеспечивает разделение ответственности между компонентами, что делает приложение более поддерживаемым и масштабируемым. Django REST Framework значительно упрощает создание REST API, предоставляя готовые компоненты для сериализации данных, обработки запросов и маршрутизации URL.

Разработанное приложение может быть легко расширено для добавления новых функциональных возможностей, таких как аутентификация пользователей, управление правами доступа, кэширование и т.д.

Лабораторная работа №2: Архитектурные решения с контейнеризацией

Цель работы

Целью данной лабораторной работы является контейнеризация приложения, разработанного в лабораторной работе №1, с использованием Docker и Docker Compose. Контейнеризация позволяет упаковать приложение со всеми его зависимостями в изолированные контейнеры, что обеспечивает единообразие среды выполнения и упрощает развертывание приложения.

Теоретические сведения

Контейнеризация - это технология виртуализации на уровне операционной системы, которая позволяет запускать изолированные процессы в общей среде. В отличие от виртуальных машин, контейнеры используют ядро хост-системы, что делает их более легкими и эффективными.

Docker - это платформа для разработки, доставки и запуска приложений в контейнерах. Docker использует технологию контейнеризации для упаковки приложения со всеми его зависимостями в стандартизированный блок для разработки программного обеспечения.

Docker Compose - это инструмент для определения и запуска многоконтейнерных приложений Docker. С помощью Compose вы можете определить многоконтейнерное приложение в одном файле, а затем запустить его одной командой.

Основные компоненты Docker:

1. **Dockerfile** - текстовый файл, содержащий инструкции для сборки Docker-образа.
2. **Docker-образ** - шаблон, содержащий приложение и все его зависимости.
3. **Docker-контейнер** - запущенный экземпляр Docker-образа.
4. **Docker Hub** - репозиторий Docker-образов.

Преимущества контейнеризации: - Изоляция приложений и их зависимостей - Единообразие среды выполнения - Простота развертывания и масштабирования - Эффективное использование ресурсов - Быстрый запуск и остановка контейнеров

Используемые технологии

Для выполнения лабораторной работы были использованы следующие технологии:

- **Docker** - платформа для контейнеризации приложений
- **Docker Compose** - инструмент для определения и запуска многоконтейнерных приложений
- **Nginx** - веб-сервер и обратный прокси-сервер
- **PostgreSQL** - реляционная база данных
- **Gunicorn** - WSGI HTTP-сервер для запуска Django-приложения

Описание реализации

Структура проекта

Проект имеет следующую структуру:

```
lab2/
├── .env                    # Файл с переменными окружения
├── docker-compose.yaml     # Конфигурация Docker Compose
├── backend/                # Директория с Django-приложением
│   ├── Dockerfile          # Dockerfile для бэкенда
│   ├── entrypoint.sh       # Скрипт инициализации бэкенда
│   └── ...                 # Файлы Django-приложения из
лабораторной работы №1
└── nginx/                  # Директория с конфигурацией Nginx
    ├── Dockerfile          # Dockerfile для Nginx
    └── templates/          # Шаблоны конфигурации Nginx
        └── default.conf.template  # Шаблон конфигурации Nginx
```

Dockerfile для бэкенда

Для контейнеризации бэкенда был создан следующий Dockerfile:

```
FROM python:3.11-slim

# Установка зависимостей
RUN apt-get update && apt-get install -y --no-install-
recommends \
    postgresql-client \
    && rm -rf /var/lib/apt/lists/*

# Установка рабочей директории
WORKDIR /app

# Копирование файлов зависимостей
COPY requirements.txt .

# Установка зависимостей Python
RUN pip install --no-cache-dir -r requirements.txt

# Копирование исходного кода приложения
COPY . .

# Установка прав на выполнение скрипта entrypoint.sh
RUN chmod +x entrypoint.sh

# Открытие порта для Gunicorn
EXPOSE 8000
```

```
# Запуск скрипта entrypoint.sh при старте контейнера
ENTRYPOINT ["./entrypoint.sh"]
```

Dockerfile выполняет следующие действия: 1. Использует базовый образ Python 3.11 slim 2. Устанавливает необходимые системные зависимости 3. Устанавливает рабочую директорию /app 4. Копирует и устанавливает зависимости Python 5. Копирует исходный код приложения 6. Устанавливает права на выполнение скрипта entrypoint.sh 7. Открывает порт 8000 для Gunicorn 8. Запускает скрипт entrypoint.sh при старте контейнера

Скрипт entrypoint.sh

Для инициализации бэкенда был создан скрипт entrypoint.sh:

```
#!/bin/sh

# Ожидание доступности базы данных
echo "Waiting for PostgreSQL..."
while ! pg_isready -h $POSTGRES_HOST -p $POSTGRES_PORT -U
$POSTGRES_USER; do
    sleep 1
done
echo "PostgreSQL is available"

# Применение миграций
echo "Applying migrations..."
python manage.py migrate

# Создание суперпользователя, если он не существует
echo "Creating superuser..."
python manage.py shell -c "
from django.contrib.auth.models import User
if not User.objects.filter(username='admin').exists():
    User.objects.create_superuser('admin', 'admin@example.com',
'admin')
    print('Superuser created')
else:
    print('Superuser already exists')
"

# Генерация тестовых данных
echo "Generating test data..."
python manage.py shell -c "
from mainapp.models import Book
if Book.objects.count() == 0:
    import random
    from faker import Faker
    fake = Faker()
    for i in range(1, 101):
```

```
        Book.objects.create(
            title=fake.catch_phrase(),
            author=fake.name(),
            publication_year=random.randint(1900, 2025),
            genre=random.choice(['fiction', 'non_fiction',
'science', 'fantasy', 'sci_fi', 'detective', 'romance',
'horror', 'other']),
            pages=random.randint(50, 1000),
            rating=round(random.uniform(0, 10), 1),
            is_bestseller=random.choice([True, False])
        )
    print('Test data generated')
else:
    print('Test data already exists')
"

# Сбор статических файлов
echo "Collecting static files..."
python manage.py collectstatic --noinput

# Запуск Gunicorn
echo "Starting Gunicorn..."
exec gunicorn lab1.wsgi:application --bind 0.0.0.0:8000
```

Скрипт выполняет следующие действия: 1. Ожидает доступности базы данных PostgreSQL 2. Применяет миграции Django 3. Создает суперпользователя, если он не существует 4. Генерирует тестовые данные, если база данных пуста 5. Собирает статические файлы 6. Запускает Gunicorn для обслуживания Django-приложения

Dockerfile для Nginx

Для контейнеризации Nginx был создан следующий Dockerfile:

```
FROM nginx:1.25-alpine

# Удаление стандартной конфигурации Nginx
RUN rm /etc/nginx/conf.d/default.conf

# Копирование шаблона конфигурации
COPY templates/default.conf.template /etc/nginx/templates/

# Открытие порта 80
EXPOSE 80
```

Dockerfile выполняет следующие действия: 1. Использует базовый образ Nginx 1.25 Alpine 2. Удаляет стандартную конфигурацию Nginx 3. Копирует шаблон конфигурации 4. Открывает порт 80

Шаблон конфигурации Nginx

Для настройки Nginx был создан шаблон конфигурации:

```
server {
    listen 80;
    server_name localhost;

    location /static/ {
        alias /app/static/;
    }

    location /media/ {
        alias /app/media/;
    }

    location / {
        proxy_pass http://${BACKEND_HOST}:${BACKEND_PORT};
        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_set_header X-Forwarded-Proto $scheme;
    }
}
```

Конфигурация Nginx выполняет следующие действия: 1. Настраивает сервер для прослушивания порта 80 2. Настраивает обслуживание статических файлов 3. Настраивает обслуживание медиа-файлов 4. Настраивает проксирование запросов к бэкенду

Файл .env

Для хранения переменных окружения был создан файл .env:

```
# PostgreSQL
POSTGRES_DB=lab1_db
POSTGRES_USER=django
POSTGRES_PASSWORD=2NevjFoQisyVaV
POSTGRES_HOST=db
POSTGRES_PORT=5432

# Django
DJANGO_SECRET_KEY=django-insecure-key
DJANGO_DEBUG=False
DJANGO_ALLOWED_HOSTS=localhost,127.0.0.1,backend-service

# Nginx
```

```
BACKEND_HOST=backend-service
BACKEND_PORT=8000
```

Файл .env содержит переменные окружения для PostgreSQL, Django и Nginx.

Файл docker-compose.yaml

Для определения и запуска многоконтейнерного приложения был создан файл docker-compose.yaml:

```yaml
version: '3.8'

services:
  db:
    image: postgres:14-alpine
    volumes:
      - postgres_data:/var/lib/postgresql/data/
    env_file:
      - ./.env
    environment:
      - POSTGRES_PASSWORD=${POSTGRES_PASSWORD}
      - POSTGRES_USER=${POSTGRES_USER}
      - POSTGRES_DB=${POSTGRES_DB}
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U ${POSTGRES_USER} -d $
{POSTGRES_DB}"]
      interval: 5s
      timeout: 5s
      retries: 5

  backend-service:
    build: ./backend
    volumes:
      - static_value:/app/static/
      - media_value:/app/media/
    depends_on:
      db:
        condition: service_healthy
    env_file:
      - ./.env
    environment:
      - POSTGRES_PASSWORD=${POSTGRES_PASSWORD}
      - POSTGRES_USER=${POSTGRES_USER}
      - POSTGRES_DB=${POSTGRES_DB}
      - POSTGRES_HOST=${POSTGRES_HOST}
      - POSTGRES_PORT=${POSTGRES_PORT}
      - DJANGO_SECRET_KEY=${DJANGO_SECRET_KEY}
      - DJANGO_DEBUG=${DJANGO_DEBUG}
      - DJANGO_ALLOWED_HOSTS=${DJANGO_ALLOWED_HOSTS}
```

```
  nginx:
    build: ./nginx
    ports:
      - "80:80"
    volumes:
      - static_value:/app/static/
      - media_value:/app/media/
    depends_on:
      - backend-service
    env_file:
      - ./.env
    environment:
      - BACKEND_HOST=${BACKEND_HOST}
      - BACKEND_PORT=${BACKEND_PORT}

volumes:
  postgres_data:
  static_value:
  media_value:
```

Файл docker-compose.yaml определяет три сервиса: 1. **db** - сервис PostgreSQL для хранения данных 2. **backend-service** - сервис Django для обработки бизнес-логики 3. **nginx** - сервис Nginx для обслуживания статических файлов и проксирования запросов

Также определены три тома для хранения данных: 1. **postgres_data** - том для хранения данных PostgreSQL 2. **static_value** - том для хранения статических файлов 3. **media_value** - том для хранения медиа-файлов

Обновление настроек Django

Для работы с Docker были обновлены настройки Django в файле settings.py:

```
 import os
from pathlib import Path

# Build paths inside the project like this: BASE_DIR / 'subdir'.
BASE_DIR = Path(__file__).resolve().parent.parent

# SECURITY WARNING: keep the secret key used in production
secret!
SECRET_KEY = os.environ.get('DJANGO_SECRET_KEY', 'django-
insecure-key')

# SECURITY WARNING: don't run with debug turned on in
production!
DEBUG = os.environ.get('DJANGO_DEBUG', 'False') == 'True'
```

```
ALLOWED_HOSTS = os.environ.get('DJANGO_ALLOWED_HOSTS',
'localhost,127.0.0.1').split(',')

# Application definition
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'rest_framework',
    'django_filters',
    'corsheaders',
    'mainapp',
]

MIDDLEWARE = [
    'django.middleware.security.SecurityMiddleware',
    'django.contrib.sessions.middleware.SessionMiddleware',
    'corsheaders.middleware.CorsMiddleware',
    'django.middleware.common.CommonMiddleware',
    'django.middleware.csrf.CsrfViewMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
    'django.contrib.messages.middleware.MessageMiddleware',
    'django.middleware.clickjacking.XFrameOptionsMiddleware',
]

# Database
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': os.environ.get('POSTGRES_DB', 'lab1_db'),
        'USER': os.environ.get('POSTGRES_USER', 'django'),
        'PASSWORD': os.environ.get('POSTGRES_PASSWORD',
'2NevjFoQisyVaV'),
        'HOST': os.environ.get('POSTGRES_HOST', 'localhost'),
        'PORT': os.environ.get('POSTGRES_PORT', '5432'),
    }
}

# Static files (CSS, JavaScript, Images)
STATIC_URL = '/static/'
STATIC_ROOT = os.path.join(BASE_DIR, 'static')

# Media files
MEDIA_URL = '/media/'
MEDIA_ROOT = os.path.join(BASE_DIR, 'media')

# CORS settings
CORS_ALLOW_ALL_ORIGINS = True
```

Основные изменения в настройках Django: 1. Использование переменных окружения для конфигурации 2. Настройка статических и медиа-файлов 3. Настройка CORS для разрешения кросс-доменных запросов

Запуск и тестирование

Для запуска контейнеризированного приложения используется команда:

```
docker-compose up -d --build
```

После запуска приложение доступно по адресу http://localhost/api/books/

Для тестирования API можно использовать те же запросы, что и в лабораторной работе №1, но с измененным хостом:

```
@hostname = http://localhost

### Получение списка всех книг
GET {{hostname}}/api/books/
Content-Type: application/json
```

Выводы

В результате выполнения лабораторной работы было контейнеризировано приложение, разработанное в лабораторной работе №1, с использованием Docker и Docker Compose. Контейнеризация обеспечивает изоляцию приложения и его зависимостей, что упрощает развертывание и масштабирование приложения.

Использование Docker Compose позволяет определить многоконтейнерное приложение в одном файле и запустить его одной командой. Это значительно упрощает управление приложением и обеспечивает единообразие среды выполнения.

Разделение приложения на отдельные контейнеры (база данных, бэкенд, веб-сервер) соответствует принципам микросервисной архитектуры и обеспечивает независимость компонентов. Каждый контейнер может быть масштабирован и обновлен независимо от других.

Контейнеризация также упрощает процесс непрерывной интеграции и непрерывной доставки (CI/CD), позволяя автоматизировать сборку, тестирование и развертывание приложения.

Лабораторная работа №3: Клиентская часть на Qt

Цель работы

Целью данной лабораторной работы является разработка клиентской части приложения с использованием фреймворка Qt, которая будет взаимодействовать с API, разработанным в лабораторных работах №1 и №2. Клиентская часть должна предоставлять пользовательский интерфейс для работы с каталогом книг.

Теоретические сведения

Qt - это кроссплатформенный фреймворк для разработки программного обеспечения на языке C++. Qt предоставляет широкий набор инструментов для создания графического пользовательского интерфейса, работы с сетью, базами данных, мультимедиа и т.д.

Модель-Представление-Контроллер (MVC) - это архитектурный паттерн, который разделяет приложение на три основных компонента: 1. **Модель (Model)** - отвечает за данные и бизнес-логику приложения 2. **Представление (View)** - отвечает за отображение данных пользователю 3. **Контроллер (Controller)** - отвечает за обработку пользовательского ввода и обновление модели и представления

В Qt реализована вариация паттерна MVC, называемая **Модель-Представление** (Model-View), где роль контроллера частично берет на себя представление.

Сигналы и слоты - это механизм Qt для коммуникации между объектами. Сигналы испускаются объектами при наступлении определенных событий, а слоты - это функции, которые вызываются в ответ на сигналы.

QNetworkAccessManager - это класс Qt для отправки сетевых запросов и получения ответов. Он поддерживает HTTP, FTP и другие протоколы.

QAbstractTableModel - это базовый класс для создания моделей данных в Qt. Он предоставляет интерфейс для работы с табличными данными.

Используемые технологии

Для выполнения лабораторной работы были использованы следующие технологии:

- **Qt 5.15** - фреймворк для разработки кроссплатформенных приложений
- **C++17** - язык программирования
- **QNetworkAccessManager** - класс Qt для работы с сетью

- **QJsonDocument** - класс Qt для работы с JSON
- **QAbstractTableModel** - класс Qt для создания моделей данных
- **QSortFilterProxyModel** - класс Qt для фильтрации и сортировки моделей данных

Описание реализации

Структура проекта

Проект имеет следующую структуру:

```
lab3/
├── BookClient.pro          # Файл проекта Qt
├── src/                    # Исходные файлы
│   ├── main.cpp            # Точка входа в приложение
│   ├── mainwindow.cpp      # Главное окно приложения
│   ├── book.cpp            # Класс для представления книги
│   ├── bookmodel.cpp       # Модель данных для таблицы книг
│   ├── apiservice.cpp      # Сервис для работы с API
│   └── bookdialog.cpp      # Диалог для добавления/
редактирования книги
├── include/                # Заголовочные файлы
│   ├── mainwindow.h        # Заголовок главного окна
│   ├── book.h              # Заголовок класса книги
│   ├── bookmodel.h         # Заголовок модели данных
│   ├── apiservice.h        # Заголовок сервиса API
│   └── bookdialog.h        # Заголовок диалога книги
├── ui/                     # Файлы пользовательского интерфейса
│   ├── mainwindow.ui       # UI главного окна
│   └── bookdialog.ui       # UI диалога книги
└── resources/              # Ресурсы приложения
    ├── resources.qrc       # Файл ресурсов Qt
    └── icons/              # Иконки приложения
```

Класс Book

Для представления книги был создан класс Book:

```
class Book
{
public:
    Book();
    Book(int id, const QString &title, const QString &author,
int publicationYear,
         const QString &genre, int pages, double rating, bool
isBestseller);
```

```
    // Getters
    int id() const;
    QString title() const;
    QString author() const;
    int publicationYear() const;
    QString genre() const;
    int pages() const;
    double rating() const;
    bool isBestseller() const;

    // Setters
    void setId(int id);
    void setTitle(const QString &title);
    void setAuthor(const QString &author);
    void setPublicationYear(int year);
    void setGenre(const QString &genre);
    void setPages(int pages);
    void setRating(double rating);
    void setIsBestseller(bool isBestseller);

    // JSON conversion
    static Book fromJson(const QJsonObject &json);
    QJsonObject toJson() const;

    // Genre helpers
    static QString genreDisplayName(const QString &genreCode);
    static QString genreCodeFromDisplay(const QString
&displayName);
    static QStringList genreDisplayNames();

private:
    int m_id;
    QString m_title;
    QString m_author;
    int m_publicationYear;
    QString m_genre;
    int m_pages;
    double m_rating;
    bool m_isBestseller;
};
```

Класс Book содержит поля для хранения информации о книге, методы для доступа к этим полям, а также методы для преобразования объекта в JSON и обратно.

Класс ApiService

Для взаимодействия с API был создан класс ApiService:

```
class ApiService : public QObject
{
```

```
    Q_OBJECT

public:
    explicit ApiService(QObject *parent = nullptr);
    ~ApiService();

    void getBooks();
    void getBook(int id);
    void createBook(const Book &book);
    void updateBook(const Book &book);
    void deleteBook(int id);

signals:
    void booksReceived(const QList<Book> &books);
    void bookReceived(const Book &book);
    void bookCreated(const Book &book);
    void bookUpdated(const Book &book);
    void bookDeleted(int id);
    void errorOccurred(const QString &errorMessage);

private slots:
    void onGetBooksFinished();
    void onGetBookFinished();
    void onCreateBookFinished();
    void onUpdateBookFinished();
    void onDeleteBookFinished();
    void onNetworkError(QNetworkReply::NetworkError error);

private:
    QNetworkAccessManager *m_networkManager;
    QUrl m_baseUrl;

    QNetworkRequest createRequest(const QString &endpoint);
    void handleNetworkError(QNetworkReply *reply);
};
```

Класс ApiService использует QNetworkAccessManager для отправки HTTP-запросов к API и обработки ответов. Он предоставляет методы для получения списка книг, получения информации о конкретной книге, создания, обновления и удаления книги.

Класс ApiService реализует паттерн "Адаптер", преобразуя данные между форматом API (JSON) и объектами Book. Он также может быть реализован как синглтон для обеспечения единой точки доступа к API.

Класс BookModel

Для представления данных в таблице был создан класс BookModel:

```
class BookModel : public QAbstractTableModel
{
    Q_OBJECT

public:
    enum Column {
        Id,
        Title,
        Author,
        PublicationYear,
        Genre,
        Pages,
        Rating,
        IsBestseller,
        ColumnCount
    };

    explicit BookModel(QObject *parent = nullptr);

    // Basic functionality:
    int rowCount(const QModelIndex &parent = QModelIndex())
const override;
    int columnCount(const QModelIndex &parent = QModelIndex())
const override;

    QVariant data(const QModelIndex &index, int role =
Qt::DisplayRole) const override;
    QVariant headerData(int section, Qt::Orientation
orientation, int role = Qt::DisplayRole) const override;

    // Add/remove data:
    void setBooks(const QList<Book> &books);
    void addBook(const Book &book);
    void updateBook(const Book &book);
    void removeBook(int id);

    // Get book by index
    Book getBook(int row) const;

private:
    QList<Book> m_books;
};
```

Класс BookModel наследуется от QAbstractTableModel и предоставляет методы для работы с данными в табличном виде. Он содержит список книг и методы для добавления, обновления и удаления книг.

Класс BookDialog

Для добавления и редактирования книг был создан класс BookDialog:

```
class BookDialog : public QDialog
{
    Q_OBJECT

public:
    explicit BookDialog(QWidget *parent = nullptr);
    ~BookDialog();

    void setBook(const Book &book);
    Book book() const;

private slots:
    void validate();

private:
    Ui::BookDialog *ui;
    Book m_book;

    void setupValidators();
    void fillGenreComboBox();
};
```

Класс BookDialog представляет диалоговое окно для добавления и редактирования книг. Он содержит поля для ввода информации о книге и методы для валидации введенных данных.

Класс MainWindow

Для главного окна приложения был создан класс MainWindow:

```
class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    MainWindow(QWidget *parent = nullptr);
    ~MainWindow();

private slots:
    void onBooksReceived(const QList<Book> &books);
    void onBookCreated(const Book &book);
    void onBookUpdated(const Book &book);
    void onBookDeleted(int id);
    void onErrorOccurred(const QString &errorMessage);
```

```
    void on_actionAdd_triggered();
    void on_actionEdit_triggered();
    void on_actionDelete_triggered();
    void on_actionRefresh_triggered();
    void on_searchLineEdit_textChanged(const QString &text);

private:
    Ui::MainWindow *ui;
    ApiService *m_apiService;
    BookModel *m_bookModel;
    QSortFilterProxyModel *m_proxyModel;

    void setupUi();
    void setupConnections();
    void loadBooks();
    void showError(const QString &message);
};
```

Класс MainWindow представляет главное окно приложения. Он содержит таблицу для отображения списка книг, поле для поиска и кнопки для добавления, редактирования и удаления книг.

Пользовательский интерфейс

Пользовательский интерфейс был создан с использованием Qt Designer. Главное окно содержит таблицу для отображения списка книг, поле для поиска и панель инструментов с кнопками для добавления, редактирования и удаления книг.

Диалоговое окно для добавления и редактирования книг содержит поля для ввода информации о книге: название, автор, год издания, жанр, количество страниц, рейтинг и флажок "Бестселлер".

Реализация паттернов проектирования

В проекте были реализованы следующие паттерны проектирования:

1. **Адаптер (Adapter)** - класс ApiService адаптирует данные между форматом API (JSON) и объектами Book.
2. **Синглтон (Singleton)** - класс ApiService может быть реализован как синглтон для обеспечения единой точки доступа к API.
3. **Наблюдатель (Observer)** - механизм сигналов и слотов Qt реализует паттерн "Наблюдатель". Например, класс ApiService испускает сигналы при получении данных от API, а класс MainWindow подписывается на эти сигналы.

4. **Модель-Представление (Model-View)** - классы BookModel и QTableView реализуют паттерн "Модель-Представление". BookModel содержит данные, а QTableView отображает их.

Тестирование приложения

Для тестирования приложения был запущен сервер API из лабораторной работы №1 или №2, а затем запущено клиентское приложение. Были проверены следующие функции:

1. **Получение списка книг** - при запуске приложения загружается список книг с сервера и отображается в таблице.
2. **Добавление книги** - при нажатии на кнопку "Добавить" открывается диалоговое окно для ввода информации о новой книге. После заполнения полей и нажатия на кнопку "OK" книга добавляется на сервер и в таблицу.
3. **Редактирование книги** - при выборе книги в таблице и нажатии на кнопку "Редактировать" открывается диалоговое окно с информацией о выбранной книге. После изменения полей и нажатия на кнопку "OK" информация о книге обновляется на сервере и в таблице.
4. **Удаление книги** - при выборе книги в таблице и нажатии на кнопку "Удалить" появляется диалоговое окно с подтверждением удаления. После подтверждения книга удаляется с сервера и из таблицы.
5. **Поиск книг** - при вводе текста в поле поиска таблица фильтруется, отображая только книги, соответствующие поисковому запросу.
6. **Сортировка книг** - при нажатии на заголовок столбца таблица сортируется по соответствующему полю.

Выводы

В результате выполнения лабораторной работы была разработана клиентская часть приложения с использованием фреймворка Qt, которая взаимодействует с API, разработанным в лабораторных работах №1 и №2. Клиентская часть предоставляет пользовательский интерфейс для работы с каталогом книг.

Использование Qt позволило создать кроссплатформенное приложение с богатым пользовательским интерфейсом. Механизм сигналов и слотов Qt обеспечивает гибкую коммуникацию между объектами, а классы для работы с сетью и JSON упрощают взаимодействие с API.

Реализация паттернов проектирования, таких как "Адаптер", "Синглтон", "Наблюдатель" и "Модель-Представление", обеспечивает хорошую структуру кода и упрощает его поддержку и расширение.

Разработанное приложение демонстрирует принципы клиент-серверной архитектуры и может быть использовано как основа для более сложных приложений.

Заключение

В ходе выполнения трех лабораторных работ был разработан полноценный программный комплекс, состоящий из серверной части с REST API, контейнеризированного приложения и клиентской части на Qt. Этот комплекс демонстрирует различные архитектурные подходы к разработке программного обеспечения и их практическое применение.

В лабораторной работе №1 была разработана трёхуровневая клиент-серверная архитектура с использованием Django и Django REST Framework. Эта архитектура обеспечивает разделение ответственности между компонентами, что делает приложение более поддерживаемым и масштабируемым. Django REST Framework значительно упрощает создание REST API, предоставляя готовые компоненты для сериализации данных, обработки запросов и маршрутизации URL.

В лабораторной работе №2 было контейнеризировано приложение, разработанное в лабораторной работе №1, с использованием Docker и Docker Compose. Контейнеризация обеспечивает изоляцию приложения и его зависимостей, что упрощает развертывание и масштабирование приложения. Использование Docker Compose позволяет определить многоконтейнерное приложение в одном файле и запустить его одной командой.

В лабораторной работе №3 была разработана клиентская часть приложения с использованием фреймворка Qt, которая взаимодействует с API, разработанным в лабораторных работах №1 и №2. Клиентская часть предоставляет пользовательский интерфейс для работы с каталогом книг. Использование Qt позволило создать кроссплатформенное приложение с богатым пользовательским интерфейсом.

В процессе выполнения лабораторных работ были применены различные паттерны проектирования, такие как "Адаптер", "Синглтон", "Наблюдатель" и "Модель-Представление". Эти паттерны обеспечивают хорошую структуру кода и упрощают его поддержку и расширение.

Разработанный программный комплекс демонстрирует принципы современной разработки программного обеспечения, включая клиент-серверную архитектуру, контейнеризацию и разработку пользовательского интерфейса. Он может быть использован как основа для более сложных приложений и как пример применения различных архитектурных подходов.

Список использованных источников

1. Django Documentation. [Электронный ресурс]. URL: https://docs.djangoproject.com/
2. Django REST Framework Documentation. [Электронный ресурс]. URL: https://www.django-rest-framework.org/
3. Docker Documentation. [Электронный ресурс]. URL: https://docs.docker.com/
4. Docker Compose Documentation. [Электронный ресурс]. URL: https://docs.docker.com/compose/
5. Qt Documentation. [Электронный ресурс]. URL: https://doc.qt.io/
6. PostgreSQL Documentation. [Электронный ресурс]. URL: https://www.postgresql.org/docs/
7. Nginx Documentation. [Электронный ресурс]. URL: https://nginx.org/en/docs/
8. Gunicorn Documentation. [Электронный ресурс]. URL: https://docs.gunicorn.org/
9. Faker Documentation. [Электронный ресурс]. URL: https://faker.readthedocs.io/
10. Design Patterns: Elements of Reusable Object-Oriented Software. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Addison-Wesley, 1994.