

**МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ
ФЕДЕРАЦИИ**
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
«БЕЛГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНОЛОГИЧЕСКИЙ
УНИВЕРСИТЕТ им. В. Г. ШУХОВА»
(БГТУ им. В.Г. Шухова)



ИНСТИТУТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ И УПРАВЛЯЮЩИХ СИСТЕМ

КУРСОВОЙ ПРОЕКТ

по дисциплине: Объектно-ориентированное программирование
тема: «Библиотека»

Автор работы _____ Пахомов Владислав Андреевич ПВ-223
(подпись)

Автор работы _____ Романов Максим Николаевич ПВ-223
(подпись)

Автор работы _____ Тухтаров Александр Романович ПВ-223
(подпись)

Руководитель проекта _____ Черников Сергей Викторович
(подпись)

Оценка _____

Белгород 2024 г.

Оглавление

Название, цель, постановка задачи	3
1 Анализ предметной области	3
1.1 Объекты и связи между ними	3
1.2 Роли	4
1.3 Дополнительная логика	4
2 Разработка базы данных	5
2.1 ER-модель	5
3 Разработать приложение	5
3.1 Сервер	5
3.1.1 Интерфейс	5
3.1.2 Авторизация и аутентификация	6
3.1.3 Модель	8
3.1.4 Сериализация	8
3.1.5 Проверка прав	9
3.1.6 Фильтрация, пагинация, упорядочивание	9
3.1.7 Собираем всё вместе	11
3.2 Пользовательский интерфейс	12
3.2.1 Взаимодействие с API	12
3.2.2 Дополнительный функционал	15
3.2.3 Страницы	18
3.2.4 Экспорт отчётов	20
4 Автоматизация	20
4.1 Автоматизация процессов доставки	20
4.2 Автоматизация бекапов	21
5 Работа приложения	22
6 Вывод о проделанной работе	22
7 Список источников и литературы	23

Название, цель, постановка задачи

Название: Библиотека

Цель: Разработка приложения для учёта и хранения базы данных библиотеки, поддерживающего распределение по ролям, выгрузку отчётов в файл, автоматическое создание бекапов.

Задачи

- Анализ предметной области
- Разработать базу данных
 - Разработать ER-модель
- Разработать приложение
 - Выбрать стек технологий
 - Разработать API, позволяющее взаимодействовать с базой данных
 - Разработать пользовательский интерфейс для взаимодействия с API
- Настройка автоматических процессов
 - Разработка процессов автоматической доставки актуальной версии приложения пользователю
 - Разработка автоматических бекапов

1 Анализ предметной области

1.1 Объекты и связи между ними

Библиотека содержит **книги**, у книг есть их описание, название, авторы, жанры, издательский дом. Однако в библиотеке может быть множество книг с одинаковым описанием, поэтому можно выделить также сущность **описание книги**, которое в свою очередь будет содержать авторов, жанры, издательский дом, описание, ISBN, изображение. А у книги можем оставить

только её инвентарный номер и состояние. **Издательский дом**, **жанр** - словари, которые содержат только название и описание. **Автор** тоже своего рода словарь, однако он также содержит фамилию, отчество и изображение. У книги должна быть **запись в журнале**, чтобы вести учёт передачи книги. Она может содержать дату выдачи, ожидаемую дату возврата и настоящую дату возврата. Нам нужно знать, кому выдали книгу, поэтому у записи в журнале должна быть связь с **пользователем** - читателем библиотеки. Пользователь должен содержать ФИО, номер телефона, почту и паспортные данные. Объект **изображение** также можем выделить.

1.2 Роли

Обычный пользователь неаутентифицированный пользователь. Может просматривать книги, описания книги, издательский дом, жанр, авторов, изображения.

Читатель может всё, что умеет обычный пользователь а также может просматривать только свои записи в журнале, только информацию о себе, создавать изображения.

Библиотекарь умеет делать всё то же, что и читатель, но ещё дополнительно может создавать, изменять, удалять просматривать все записи в журнале; создавать, изменять, удалять книги; просматривать всех пользователей; создавать, изменять, удалять описание книги.

Администратор умеет делать всё то же, что и библиотекарь, но ещё и может создавать, изменять, удалять авторов, жанры, издательства.

Суперпользователь умеет абсолютно всё, не имеет ограничений.

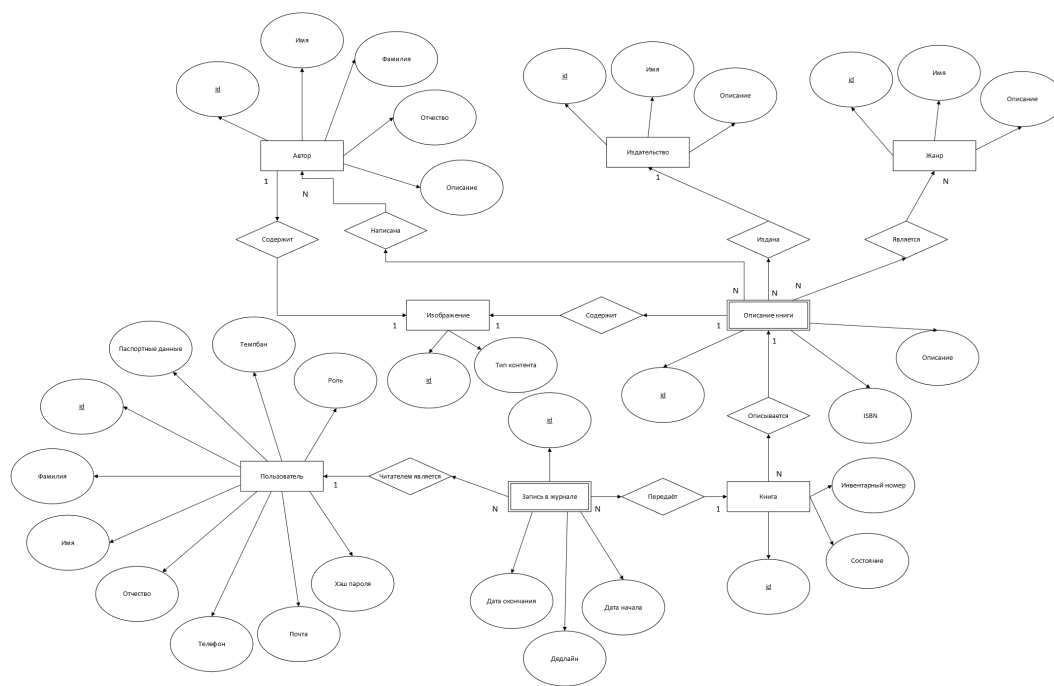
1.3 Дополнительная логика

Необходимо уметь поддерживать корректное состояние журнала, дата выдачи не должна противоречить датам возврата и выдачи прошлым записям; дата возврата также не должна противоречить датам получения следующих объектов. Дата возврата должна быть больше даты выдачи.

2 Разработка базы данных

В качестве базы данных была выбрана база данных PostgreSQL.

2.1 ER-модель



3 Разработать приложение

Для манипуляции базы данных было принято использовать клиент-серверное приложение.

3.1 Сервер

В качестве фреймворка был использован Django.

3.1.1 Интерфейс

Для взаимодействия с приложениями используется API (класс `ModelViewSet`). Плюс этого подхода заключается в том, что для приложения можно будет в дальнейшем использовать множество пользовательских интерфейсов, например можно будет иметь сайт, мобильное приложение, телеграм бот, которые будут взаимодействовать через один API. В качестве протокола был использован REST с базовыми операциями CRUD, которые

предоставляются `ModelViewSet`. Плюсом этого протокола является широкая поддержка, документация и большое количество примеров. Можно было бы использовать, например, `gRPC`. Он бы позволил установить строгую типизацию ответов и запросов и высокую производительность, однако протокол всё ещё слабо поддерживается и библиотек для Django нет.

3.1.2 Авторизация и аутентификация

При регистрации пользователя будем принимать и валидировать его данные при помощи формы, если данные корректны - будем создавать нативного пользователя Django и привязывать к нему профиль. Важно также отметить, что новому пользователю по умолчанию добавляется группа `reader`, которая представляет роль читатель. При логине проверим, что логин пользователя (его почта или телефон) существуют и пароль подходит. Если они подходят, значит аутентифицируем пользователя.

```
@require_http_methods(["POST"])
def user_signup(request):
    form = SignUpForm(request.POST)
    if not form.is_valid():
        return JsonResponse(json.loads(form.errors.as_json()), status=400)

    email = form.cleaned_data.get('email')
    raw_password = form.cleaned_data.get('password1')

    user = User.objects.create_user(username=email,
                                    password=raw_password)

    authenticate(username=email, password=raw_password)
    login(request, user)

    user_profile = Profile(
        user=user,
        phone_number=form.cleaned_data.get('phone_number'),
        surname=form.cleaned_data.get('surname'),
        name=form.cleaned_data.get('name'),
        patronymics=form.cleaned_data.get('patronymics'),
        passport_data=form.cleaned_data.get('passport_data')
    )
    user_profile.save()

    # Дефолтная группа читателя, нужна чтобы вручную не проставлять всем юзерам права вручную
    reader, created = Group.objects.get_or_create(name='reader')
```

```

reader.user_set.add(user)

return JsonResponse({}, status=200)

@require_http_methods(["POST"])
def user_login(request):
    form = LoginForm(request.POST)
    if not form.is_valid():
        return JsonResponse(json.loads(form.errors.as_json()), status=400)

    raw_login = form.cleaned_data.get('login')
    raw_password = form.cleaned_data.get('password')

    single_profile = Profile.objects.filter(phone_number=raw_login).first()

    if not single_profile:
        single_user = User.objects.filter(username=raw_login).first()

        if not single_user:
            return JsonResponse({"message": "User doesn't exists"}, status=400)

        raw_email = single_user.username
    else:
        raw_email = single_profile.user.username

    user = authenticate(request=request, username=raw_email, password=raw_password)

    if user:
        login(request, user)
        return JsonResponse({}, status=200)

    return JsonResponse({"message": "User doesn't exists"}, status=400)

```

В Django по умолчанию для авторизации пользователя используются сессии (SessionAuthentication). Сессия задаётся при помощи Cookie session, при помощи которого в дальнейшем пользователь и аутентифицируется в запросах. Также для опасных действий в Django предусмотрен ещё один механизм - CSRF-токены. Дело в том, что украсть Cookie сессии довольно легко, злоумышленник сможет выполнять от имени пользователя опасные действия, связанные с изменением базы данных. Для этого был придуман CSRF токен, при каждом опасном действии он должен прикрепляться к запросу в форме и в заголовке запроса. При отправке опасного запроса CSRF-токена просрачивается, и пользователю нужно будет запросить новый токен.

Добавим эндпоинт для генерации CSRF.

```
def csrf_ensure(request):
    token = get_token(request)
    return JsonResponse({"csrf": token}, status=200)
```

3.1.3 Модель

Django содержит довольно мощную ORM, задать модель можно при помощи Model, там же и укажем валидаторы для полей:

```
from django.db import models
from django_backend.models import FileModel
from django_backend.validators.author import (
    validate_name,
    validate_surname,
    validate_patronymics,
)

class Author(models.Model):
    id = models.AutoField(primary_key=True)
    surname = models.TextField(blank=False, null=False, validators=[validate_surname])
    name = models.TextField(blank=False, null=False, validators=[validate_name])
    patronymics = models.TextField(blank=True, null=False, validators=[validate_patronymics])
    description = models.TextField(blank=False, null=False)
    icon = models.ForeignKey(FileModel, null=True, on_delete=models.SET_NULL)
```

3.1.4 Сериализация

Для сериализации и десериализации объекта используется ModelSerializer. Он указывает, какие поля модели используются для сериализации и десериализации:

```
from rest_framework import serializers
from django_backend.models import Author

class AuthorSerializer(serializers.ModelSerializer):
    class Meta:
        model = Author
```



```
fields = ('name', 'surname', 'patronymics', 'description', 'icon', 'id')
```

3.1.5 Проверка прав

Для проверки прав используется `BasePermission`. В нём можно задать права для индивидуального объекта и для вызываемого метода:

```
from rest_framework.permissions import BasePermission

class AuthorPermission(BasePermission):
    def has_permission(self, request, view):
        return (
            request.user.is_superuser or
            request.method == "GET" or
            request.method == "POST" and request.user.has_perm("django_backend.add_author") or
            request.method in ["PUT", "PATCH"] and request.user.has_perm("django_backend.change_author")
            ↪ or
            request.method == "DELETE" and request.user.has_perm("django_backend.delete_author")
        )
```

3.1.6 Фильтрация, пагинация, упорядочивание

Фильтрация, пагинация и упорядочивание очень важны для производительности проекта. Дело в том, что запрашивать абсолютно все ресурсы - очень ресурсоёмкая операция. Поэтому для оптимизации используется фильтрация, пагинация и упорядочивание. Пагинация выдаёт ограниченное небольшое число ресурсов. А для удобной навигации по небольшому окну выбранных ресурсов будем использовать фильтры и упорядочивание.

Пользователю чаще всего будет удобней искать по строке, а не по отдельным атрибутам модели. Поэтому в фильтрах будем добавлять ещё один параметр `q`, который и будет содержать поиск по строке по нескольким атрибутам.

Упорядочивание задаётся в `ModelViewSet`.

Ранее было обозначено, что для

API был использован класс `ModelViewSet`. Этот класс содержится в пакете `Django Rest Framework`, он в свою очередь содержит готовый инструментарий для указанного функционала.

Пагинация:

```
from rest_framework import pagination

class CustomPagination(pagination.PageNumberPagination):
    page_size = 2
    page_size_query_param = "size"
    max_page_size = 50
    page_query_param = "page"
```

Фильтрация (пример):

```
from rest_framework import generics
from django_filters import rest_framework as filters
from django_backend.models import Author
from django_backend.filters.base import NumberInFilter
from django.db.models import Q

class AuthorFilter(filters.FilterSet):
    id = NumberInFilter(field_name="id", lookup_expr="in")
    q = filters.CharFilter(method='q_author_custom_filter')

    def q_author_custom_filter(self, queryset, name, value):
        return queryset.filter(
            Q(surname__icontains=value) |
            Q(name__icontains=value) |
            Q(patronymics__icontains=value)
        )

class Meta:
    model = Author
    fields = []
```

3.1.7 Собираем всё вместе

Теперь, кажется, всё готово для объявления API. Сделать это можно следующим образом:

```
from django_backend.models import Author
from django_backend.serializers import AuthorSerializer, AuthorShortSerializer
from rest_framework import viewsets
from django_backend.pagintaion import CustomPagination
from django_filters import rest_framework as filters
from django_backend.filters import AuthorFilter
from rest_framework.filters import OrderingFilter
from django_backend.permissions import AuthorPermission

class AuthorModelViewSet(viewsets.ModelViewSet):
    queryset = Author.objects.all()
    serializer_class = AuthorSerializer
    pagination_class = CustomPagination
    filterset_class = AuthorFilter
    filter_backends = (filters.DjangoFilterBackend, OrderingFilter,)
    permission_classes = (AuthorPermission,)
    ordering_fields = ('id', 'surname')
```

Здесь задаются все ранее объявленные объекты, откуда выбираются данные (queryset). После чего можно подключить API к бекенду

```
from django.contrib import admin
from django.urls import path
from django.urls import re_path as url, include
from django.conf import settings
from django.conf.urls.static import static
from django.views.static import serve
from django.conf.urls.static import static
from django_backend.views.user import urlpatterns as user_urlpatterns
from django_backend.views.file import urlpatterns as file_urlpatterns
from rest_framework.routers import DefaultRouter, SimpleRouter
router = DefaultRouter()

...

from django_backend.views.author import AuthorModelViewSet

...

router.register(r"authors", AuthorModelViewSet, 'authors')
```

```
...

urlpatterns = [
    path('admin/', admin.site.urls),
    *user_urlpatterns,
    *file_urlpatterns,
    path('api/', include(router.urls))
] + static(settings.STATIC_URL, document_root=settings.STATIC_ROOT)
```

3.2 Пользовательский интерфейс

В качестве пользовательского интерфейса было принято решение использовать Single Page Application. Плюс этого подхода в том, что такой сайт получается и работает быстрее, чем при Server Side Rendering, кроме того нагрузка на сервер сильно снизится. Однако такой сайт не получится индексировать поисковыми движками. Нам это вряд-ли понадобится, это приложения для локальной библиотеки, а не глобальной.

Для поддержки типизации используется TypeScript. Для отображения используется React и MUI Material с большим количеством готовых компонентов. Для взаимодействия с API и кеширования ответов с сервера используется RTK Query.

3.2.1 Взаимодействие с API

Объявим типы запросов, ответов при помощи TypeScript:

```
import { PageableListQuery, PageableListResponse, SearchableListQuery, SortableListQuery } from "../base"
import { CSRFMiddlewareTokenQueryFormMixin } from "../csrf"
import { FileObj } from "../file"

// Базовые модели
export interface AuthorShort {
    id: number | string
    name: string
    surname: string
    patronymics: string
    icon: FileObj["id"]
}

export interface Author extends AuthorShort {
    description: string
```

```

}

// Запросы
export interface AuthorListQuery extends PageableListQuery, SortableListQuery, SearchableListQuery {
  id?: Author["id"][]
  short?: boolean
}

export interface AuthorQuery {
  id: Author["id"]
  short?: boolean
}

export interface AuthorCreateQuery extends Exclude<Author, "id">, CSRFMiddlewareTokenQueryFormMixin { }

export interface AuthorUpdateQuery extends Author, CSRFMiddlewareTokenQueryFormMixin { }

export interface AuthorDeleteQuery extends CSRFMiddlewareTokenQueryFormMixin {
  id: AuthorShort["id"]
}

// Ответы
export interface AuthorListResponse extends PageableListResponse<AuthorShort | Author> { }

export type AuthorResponse = AuthorShort | Author;

export interface AuthorCreateResponse extends Author { }

export interface AuthorUpdateResponse extends Author { }

export type AuthorDeleteResponse = {}

```

И API взаимодействия с ресурсом:

```

import { BaseApiBuilder } from "../api/baseApi";
import { shortBase } from "../types/base";
import { constructFormData, withQueryParams } from "../utils";
import { QUERY_TAGS } from "../types/tags";
import {
  AuthorCreateQuery,
  AuthorCreateResponse,
  AuthorDeleteQuery,
  AuthorDeleteResponse,
  AuthorListQuery,
  AuthorListResponse,
  AuthorQuery,
  AuthorResponse,
  AuthorUpdateQuery,
  AuthorUpdateResponse
}

```

```

} from "../types/author";

export const authorsBase = "authors";
export const authorsShortBase = shortBase + authorsBase;

export const endpoints = (builder: BaseApiBuilder) => ({
  getAuthorList: builder.query<AuthorListResponse, AuthorListQuery>({
    query: ({ short = true, ...data }) => withQueryParams(`/${short ? authorsShortBase : authorsBase}
↪  }/`, data),
    providesTags: (result, error) => error ? [] : [
      QUERY_TAGS.Author,
      ...(result.results.map((item) => ({ type: QUERY_TAGS.Author, id: item.id })))
    ]
  }),
  getAuthor: builder.query<AuthorResponse, AuthorQuery>({
    query: ({ short = true, id }) => withQueryParams(`/${short ? authorsShortBase :
↪  authorsBase}/${id}/`, {}),
    providesTags: (result, error) => error ? [] : [
      { type: QUERY_TAGS.Author, id: result.id }
    ]
  }),
  createAuthor: builder.mutation<AuthorCreateResponse, AuthorCreateQuery>({
    query: ({ ...data }) => ({
      url: withQueryParams(`/${authorsBase}/`, {}),
      method: "POST",
      body: constructFormData(data)
    }),
    invalidatesTags: (result, error) => error ? [] : [
      QUERY_TAGS.Author,
      { type: QUERY_TAGS.Author, id: result.id }
    ]
  }),
  updateAuthor: builder.mutation<AuthorUpdateResponse, AuthorUpdateQuery>({
    query: ({ id, ...data }) => ({
      url: withQueryParams(`/${authorsBase}/${id}/`, {}),
      method: "PUT",
      body: constructFormData(data)
    }),
    invalidatesTags: (_result, _error, arg) => [
      { type: QUERY_TAGS.Author, id: arg.id }
    ]
  }),
  deleteAuthor: builder.mutation<AuthorDeleteResponse, AuthorDeleteQuery>({
    query: ({ id }) => ({
      url: withQueryParams(`/${authorsBase}/${id}/`, {}),
      method: "DELETE"
    }),
    invalidatesTags: (_result, error, arg) => error ? [] : [
      QUERY_TAGS.Author,
      { type: QUERY_TAGS.Author, id: arg.id }
    ]
  })
});

```

```
    ]  
  })  
})
```

RTK Query понимает, когда ему нужно выполнить перезапрос на ресурс при помощи тегов. Так при получении авторов мы указываем индивидуальные теги авторов и общий тег авторов. Теперь, например, при обновлении автора, в котором указан конкретный тег пользователя, будет обновлён запрос на авторов, который включал обновлённого пользователя. В этом ещё один плюс пагинации, нам не придётся запрашивать всех пользователей сразу.

3.2.2 Дополнительный функционал

Если ошибка содержит сообщение, было бы удобно отображать его при помощи снейкбара. Если мы используем формы, было бы удобно подсветить некорректные поля. Для использования форм используем библиотеку React Hook Form. Напишем для этого хук:

```
import { enqueueSnackbar, useSnackbar } from "notistack";  
import { useEffect, useRef } from "react";  
import { FieldError, ServerError, ShowErrorProps } from "./types";  
import { FetchBaseQueryError } from "@reduxjs/toolkit/query";  
  
export function useShowError({  
  isError,  
  error,  
  formMethods,  
  resourceMapping = {},  
  onNotFound = () => { },  
  onResourceForbidden = () => { }  
}: ShowErrorProps) {  
  const { enqueueSnackbar } = useSnackbar();  
  
  useEffect(() => {  
    if (!isError) return;  
    if (!(error as FetchBaseQueryError)?.status) return;  
    const fetchError = error as FetchBaseQueryError;  
  
    const serverError = fetchError?.data as ServerError;  
    let messageShown = false;  
  
    if (serverError?.message) {
```

```

        enqueueSnackbar({
            message: serverError?.message,
            variant: "error"
        });
        messageShown = true;
    }

    if (formMethods) {
        for (const [resourceKey, resourceValues] of Object.entries(serverError)) {
            if (Array.isArray(resourceValues)) {
                if (!resourceMapping[resourceKey]) {
                    formMethods.setError(resourceKey, { type: "validate", message: typeof
↪ resourceValues[0] === 'string' ? resourceValues[0] : (resourceValues[0] as FieldError).message })
                } else {
                    formMethods.setError(resourceMapping[resourceKey], { type: "validate", message:
↪ typeof resourceValues[0] === 'string' ? resourceValues[0] : (resourceValues[0] as FieldError).message
↪ })
                }
            }
        }
    } else if (!messageShown) {
        enqueueSnackbar({
            message: "Отправленные данные содержат ошибки",
            variant: "error",
        })
    }

    }, [isError]);
}

```

У нас есть поиск по строке, и было бы довольно ресурсозатратно, если бы мы отправляли каждый раз, когда пользователь бы печатал символ. Для решения этой проблемы есть debounce:

Для сохранения текущего состояния там где используется лист было бы удобно кешировать параметры в параметрах запроса и в локальной памяти. Также для этого используется свой хук:

```

import { useEffect, useLayoutEffect, useState } from "react";
import { useSearchParams } from "react-router-dom";

export const SP_ROOT = "page_state"

function isEmpty(obj) {
    return Object.keys(obj).length === 0;
}

```



```

export function useSearchParamsFilter<T>(key, useParams = true) {
  const [decodedParams, setDecodedParams] = useState<T>();
  const [searchParams, setSearchParams] = useSearchParams();

  // При загрузке страницы
  useEffect(() => {
    const encoded = searchParams.get(SP_ROOT) || localStorage.getItem(key);
    const decoded = JSON.parse(encoded || "{}");

    // Обновить программные параметры поиска
    setDecodedParams(decoded);
    // Обновить параметры поиска страницы
    !isEmpty(decoded) && useParams && setSearchParams({...searchParams, [SP_ROOT]: encoded},
    ↪ {replace: true});
  }, [useParams, key]);

  // При обновлении расположения
  useEffect(() => {
    if (!useParams) return;
    if (!searchParams.has(SP_ROOT)) return;
    // Получаем объект из параметров запроса и декодируем его
    const encoded = searchParams.get(SP_ROOT) || "{}";
    const decoded = JSON.parse(encoded || "{}");

    // Обновить программные параметры поиска
    setDecodedParams(decoded)
    // Обновляем в памяти браузера объект
    localStorage.setItem(key, encoded);
  }, [searchParams, useParams, key]);

  // Перезаписать параметры поиска
  const setParams = (setObject: T) => {
    const encoded = JSON.stringify(setObject);
    if (useParams)
      setSearchParams({...searchParams, [SP_ROOT]: encoded})
    else {
      setDecodedParams(setObject);
      localStorage.setItem(key, encoded);
    }
  }

  // Обновить параметры поиска
  const patchParams = (patchObject: Partial<T>) => {
    let copy = {...decodedParams};
    for (const key in patchObject) {
      if (patchObject[key] === null && copy.hasOwnProperty(key)) delete copy[key];
      else if (patchObject[key] !== null) copy[key] = patchObject[key];
    }

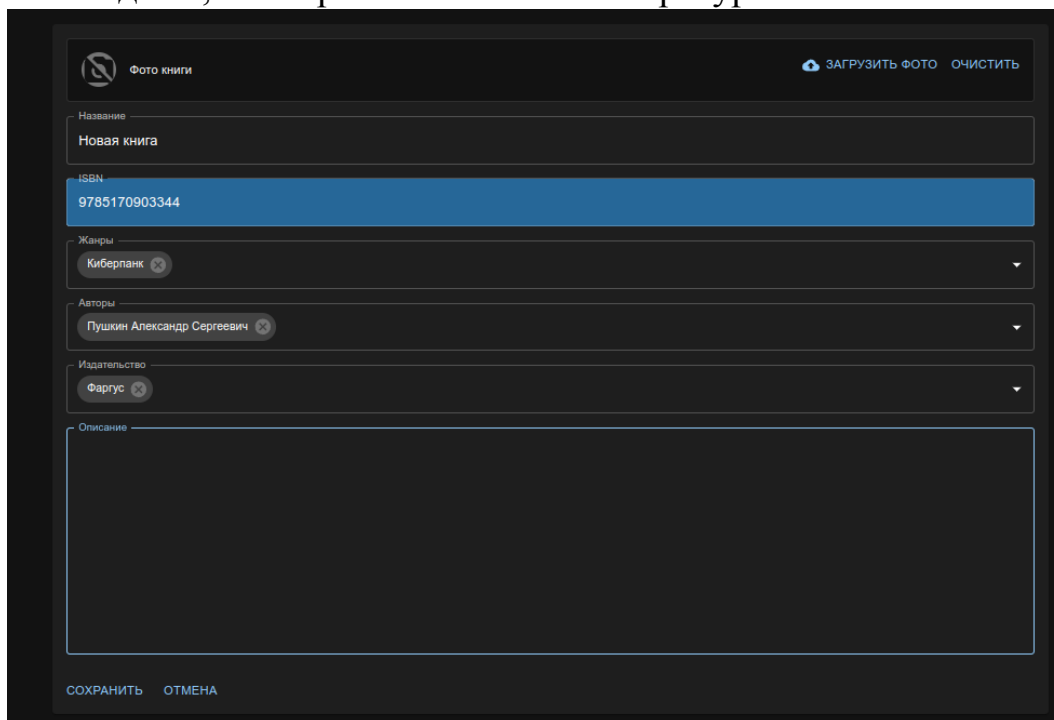
    setParams(copy);
  }

```

```
}  
  
return {params: decodedParams, setParams, patchParams}  
}
```

3.2.3 Страницы

Страница создания отрисовывает форму и отправляет её на бекенд, если необходимо, то запрашивает связанные ресурсы.

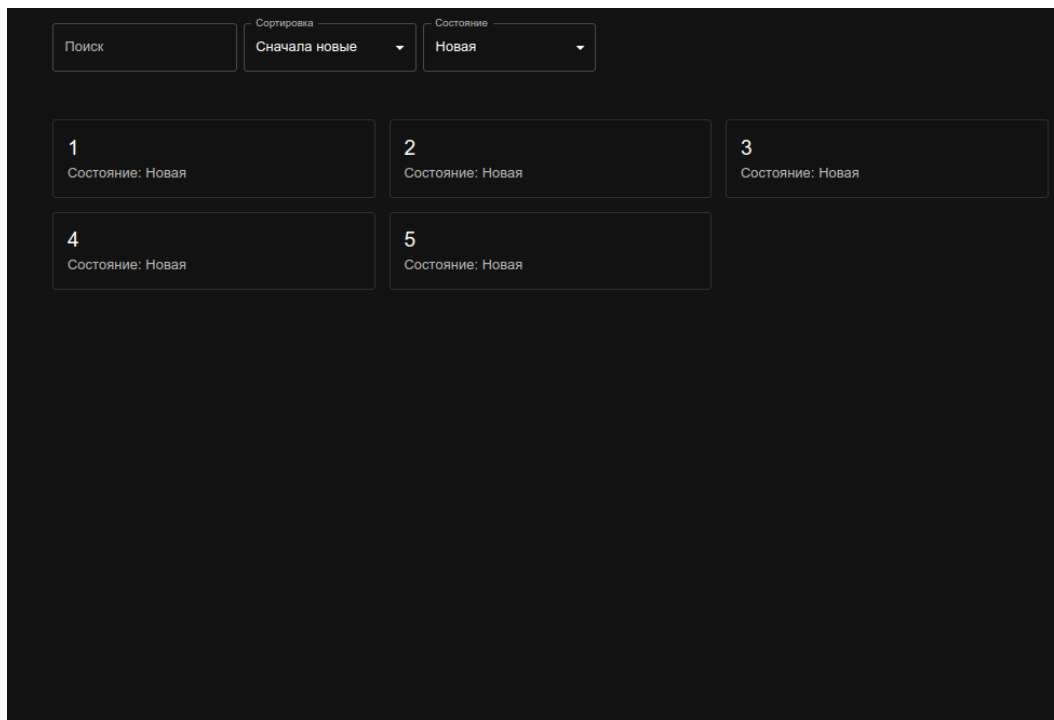


The screenshot shows a web form for creating a new book. At the top, there's a header with a logo and the text "Фото книги" (Book photo), along with links "ЗАГРУЗИТЬ ФОТО" (Upload photo) and "ОЧИСТИТЬ" (Clear). The form fields are as follows:

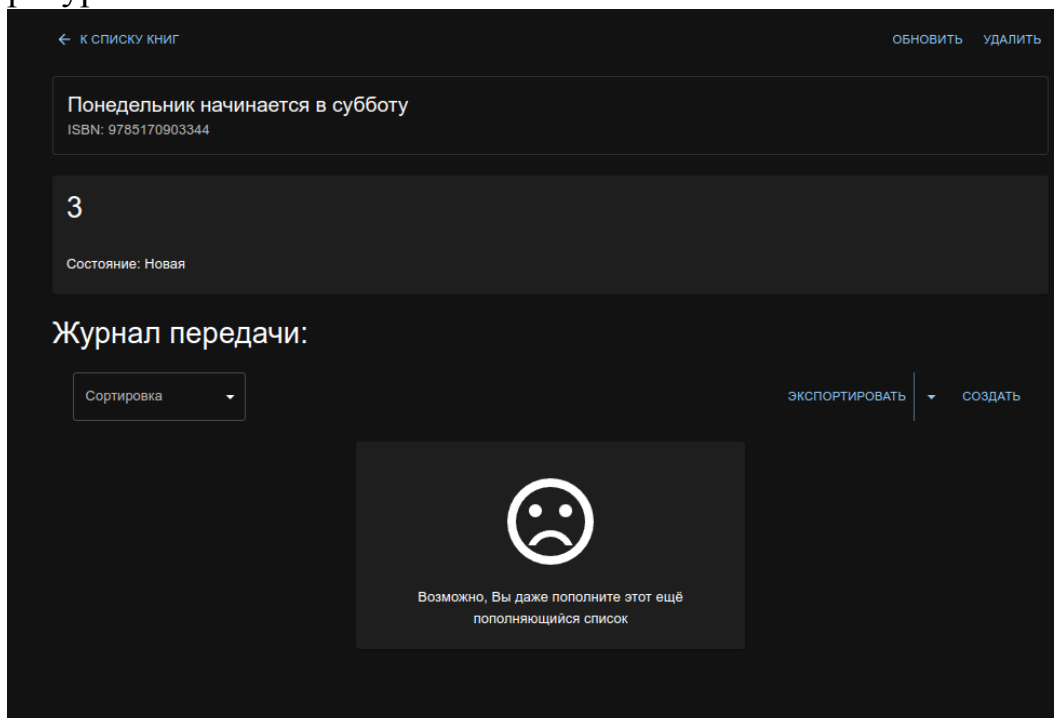
- Название** (Name): A text input field containing "Новая книга" (New book).
- ISBN**: A text input field containing "9785170903344".
- Жанры** (Genres): A dropdown menu with "Киберпанк" (Cyberpunk) selected.
- Авторы** (Authors): A dropdown menu with "Пушкин Александр Сергеевич" (Pushkin Alexander Sergeyevich) selected.
- Издательство** (Publisher): A dropdown menu with "Фаргус" (Fargus) selected.
- Описание** (Description): A large text area for the book description.

At the bottom of the form, there are two buttons: "СОХРАНИТЬ" (Save) and "ОТМЕНА" (Cancel).

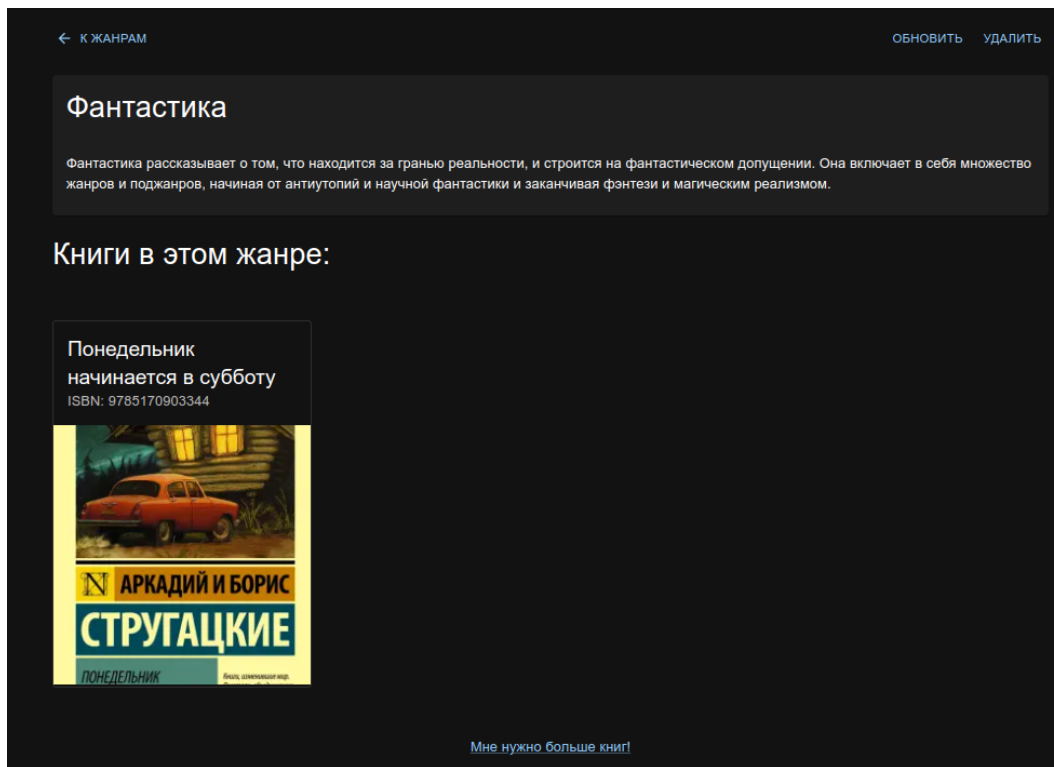
Страница обновления запрашивает конкретный объект и также как и создание отображает форму. Страница листа отображает ресурсы с пагинацией, фильтрацией и сортировкой.



Страница отображения конкретного ресурса отображает конкретный ресурс.

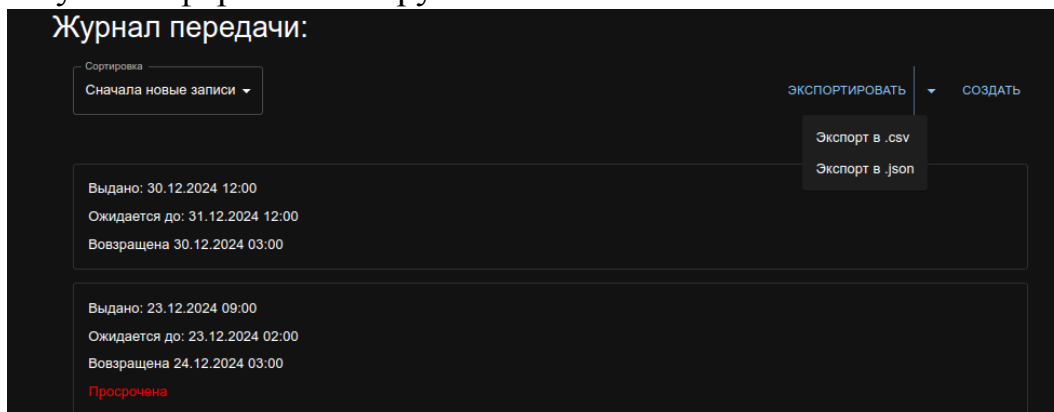


Каждая из страниц может выполнять дополнительные запросы, так например страница конкретного жанра может запрашивать книги с текущим жанром. Гибкое API и роутинг с использованием выше описанного хука позволяет даже перейти на страницу с жанром.



3.2.4 Экспорт отчётов

Страница журнала позволяет экспортировать отчёт без пагинации. Для этого приложение обращается по специальному запросу на бекенд, получает отчёт в нужном формате и загружает его.



4 Автоматизация

4.1 Автоматизация процессов доставки

Стек используемых технологий: ansible, docker, docker-compose, github-workflow(CICD), У проекта есть 2 сборки: локальная и удаленная.

Локальная сборка собирается с помощью `docker-compose.yaml` используя образы, которые получаются в результате написанных `Dockerfiles`. Изначально, в шаблоне монорепозитория был нерабочий (неправильно собирал фронтенд) `docker-compose`, что было исправлено. локальная сборка проекта теперь работает корректно.

Удаленная сборка происходит иначе. Был настроен сервер. Описание пайплайна: триггерится при пуше на ветку `main` логин в `github container registry`

Далее билдятся и пушатся в `ghcr` образы бэкенда, фронтенда и `nginx` (`postgresql` не пушится, т. к. используется стандартный образ `postgres:13-alpine`) На этом заканчивается подготовительная часть (`build`)

Далее начинается часть `deploy` `github-runner` подключается по `ssh` к серверу Запускает `ansible-playbook deploy-playbook.yaml` Шаги плейбука:

1. Проверка прав
2. Рестарт докера
3. Создание директории приложения
4. Копирование переменных среды
5. Копирует из шаблона `docker-compose` и проверяет его существование
6. Логин в `ghcr.io` и пулл всех образов
7. Запуск приложения с помощью `docker-compose`

4.2 Автоматизация бекапов

У планировщика задач `crontab` на бекенде запускается задачка по автоматическому бекапу по контейнеру бд при помощи при помощи команды `pg_dump`.

```
0 2 * * * sudo docker exec -e PGPASSWORD='SuperDockerPassword5432!!!'
↪ cec_i_aguilera_react_django_nginx_postgresql-database-1 pg_dump -U docker docker >
↪ "/path/to/backup/db_backup_$(date +%Y%m%d_%H%M%S).sql" 2>> /path/to/backup/backup_error.log
```

В дальнейшем можно будет восстановить базу данных при помощи `pg_restore`.

5 Работа приложения

Протестировать портал можно на адресе <http://82.97.246.215/>

Ссылка на репозиторий: <https://github.com/IAmProgrammist/BookSTU>

6 Вывод о проделанной работе

В ходе работы разработали клиент-серверное приложение библиотеки, позволяющее вести учёт передачи книг а также базу книг с оптимизированными методами передачи данных. Познакомились с принципами REST и способом организации API, современными методами разработки SPA, работой с CI/CD для автоматической сборки и деплоя приложения из репозитория.

7 Список источников и литературы

1. Django Documentation [Электронный ресурс]
Режим доступа: <https://docs.djangoproject.com/en/5.1/>
2. Django REST framework [Электронный ресурс]
Режим доступа: <https://www.django-rest-framework.org/>
3. React [Электронный ресурс]
Режим доступа: <https://react.dev/>
4. Redux Toolkit [Электронный ресурс]
Режим доступа: <https://redux-toolkit.js.org/rtk-query/overview>
5. React Hook Form [Электронный ресурс]
Режим доступа: <https://react-hook-form.com/docs/useform>