Министерство образования Республики Беларусь
Учреждение образования «Белорусский государственный университет информатики и радиоэлектроники»

Факультет компьютерных систем и сетей

кафедра Информатики

Дисциплина: Объектно-ориентированное программирование

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
к курсовой работе
на тему

Создание fullstack-приложения: интернет-сервиса для поиска фильмов и кинотеатров

Студент: гр. 053506 Слуцкий Н.С.

Руководитель: Летохо А.С.

Минск 2022

СОДЕРЖАНИЕ

Введение

На сегодняшний день, например, в нашей стране существует сразу несколько веб-сервисов, которые могут предложить потенциальному пользователю онлайн-каталог фильмов, помочь определиться с выбором, а также выбрать кинотеатр для просмотра. Это относится не только к фильмам, но и к другим видам услуг, товаров и т.д.

Целью данной работы является изучение принципа построения настоящего full-stack restful приложения с использованием выбранных технологий и языков программирования. И основываясь на всём этом создать подобный вышеописанный онлайн-каталог с фильмами, кинотеатрами, возможностью регистрации, авторизации и добавления, например, фильмов в список избранных.

В целом данная курсовая работа является отличной возможностью более детально изучить веб-программирование со многих сторон, а также создать рабочий проект для будущего портфолио. Автор не может сказать, что на данном этапе это приложение может быть реально полезным для настоящих пользователей в жизни, однако оно являет собой очень большой фундамент, который можно дополнять всякими микро-сервисами без особых усилий. Это значит, что функционал, например по созданию кастомной панели администратора или по возможности писать отзывы на фильм могут добавиться достаточно легко.

Анализ предметной области

REST API

REST API — это способ взаимодействия сайтов и веб-приложений с сервером. Его также называют RESTful.

У RESTful есть 7 принципов написания кода интерфейсов.

Отделения клиента от сервера (Client-Server). Клиент — это пользовательский интерфейс сайта или приложения, например, поисковая строка видеохостинга. В REST API код запросов остается на стороне клиента, а код для доступа к данным — на стороне сервера. Это упрощает организацию API, позволяет легко переносить пользовательский интерфейс на другую платформу и дает возможность лучше масштабировать серверное хранение данных.

Отсутствие записи состояния клиента (Stateless). Сервер не должен хранить информацию о состоянии (проведенных операций) клиента. Каждый запрос от клиента должен содержать только ту информацию, которая нужна для получения данных от сервера.

Кэшируемость (Casheable). В данных запроса должно быть указано, нужно ли кэшировать данные (сохранять в специальном буфере для частых запросов). Если такое указание есть, клиент получит право обращаться к этому буферу при необходимости.

Единство интерфейса (Uniform Interface). Все данные должны запрашиваться через один URL-адрес стандартными протоколами, например, HTTP. Это упрощает архитектуру сайта или приложения и делает взаимодействие с сервером понятнее.

Многоуровневость системы (Layered System). В RESTful сервера могут располагаться на разных уровнях, при этом каждый сервер взаимодействует только с ближайшими уровнями и не связан запросами с другими.

Предоставление кода по запросу (Code on Demand). Серверы могут отправлять клиенту код (например, скрипт для запуска видео). Так общий код приложения или сайта становится сложнее только при необходимости.

Начало от нуля (Starting with the Null Style). Клиент знает только одну точку входа на сервер. Дальнейшие возможности по взаимодействию обеспечиваются сервером.

Анализ существующих аналогов

Основной аналог, откуда бралась идея, - это сайт bycard.by, самый популярный в нашей стране сайт для таких целей. Он представляет из себя огромную платформу, частичку которой автор и попытался реализовать в данном курсовом проекте за IV семестр. Рисунок 1.

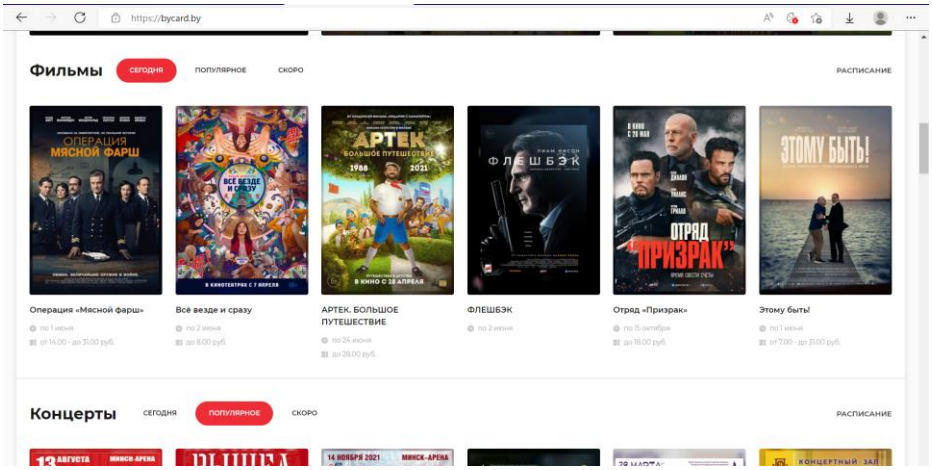


Рисунок 1. Скриншот с сайта ByCard

Постановка задачи

Основной задачей ставится проведение настоящей параллельной разработки серверной и клиентской части fullstack-приложения с учётом всех нюансов разработки как сервера, так и клиента, так и используемых технологий.
Необходимо уделить достаточное внимание следующим каким-то базовым критериям:

- Разработка приятного для пользования пользовательского интерфейса
- Прорабатывание всей логики клиентской части
    - Навигация
    - Верификация форм
    - Кэширование уже полученных данных с сервера
    - Максимальная декомпозиция компонентов
- Разработка нужных моделей в базе данных
- Создание грамотного набора путей для запросов на сервер
- Вопросы авторизации и аутентификации

- Логирование
- Тестирование серверной части
- Др.

Разработка приложения

Среда разработки (используемое ПО)

В качестве сред разработки были выбраны продукты от JetBrains: JetBrains WebStorm и JetBrains PyCharm с бесплатной студенческой лицензией. Также была использована вспомогательная программа Postman для тестирования запросов к серверу и программа PgAdmin для начальной настройки базы данных PostgreSQL. Ну и, конечно же, веб-браузер. В моём случае – Microsoft Edge.

Требования к программе

Программа должна иметь интуитивно понятный интерфейс, корректно уведомлять пользователя о всех успехах/неудачах. Например, при регистрации, входе в систему или попытке найти несуществующий фильм или кинотеатр.
Программный продукт должен обеспечивать должную безопасность сохранности данных пользователя при авторизации, аутентификации.

Используемые технологии, языки программирования и обоснование того или иного выбора

Для начала клиентская часть.
Используемый стек – TypeScript, ReactJS, MobX.
TypeScript был выбран вместо обычного JavaScript ввиду наличия более-менее строгой типизации и удобства разработки. Уже на этапе написания кода можно отловить 90% всех потенциальных ошибок с типами, которые могут возникнуть во время выполнения программы. Это очень удобно и при наличии навыков разработки на TypeScript очень ускоряет процесс разработки.
ReactJS был выбран из-за наличия у автора опыта разработки с использованием этой библиотеки.
MobX был выбран из-за крайней простоты в использовании. Помогает не писать дополнительный и часто в дальнейшем непонятный код и, следовательно, сэкономить время. Во многих настоящих проектах в последние годы в качестве state-менеджера используется именно MobX, а не Redux. Также сыграл свою роль факт, что MobX написан с нуля на

TypeScript и, следовательно, там “встроенная” типизация, а ещё интегрируется с React более легко. В Redux же иногда необходимо дополнительно прописывать типы и т.д.
Теперь о серверной части.
Используемый стек: Python, Django, Django Rest Framework, PostgreSQL. Изначально использовался язык программирования C++ и микрофреймворк Crow. Однако в данном случае оказалась важна непосредственно скорость разработки, поэтому спустя некоторое время функционал был переписан на Python + Django. Опыта работы с этим фреймворком Django и его “расширением” Django Rest Framework у автора до этого проекта не было — поэтому пришлось с нуля освоить основы работы с ними. Однако суммарное время всё равно по итогу, по мнению автора, меньше, чем если бы он писал на C++, Crow и каких-нибудь дополнительных библиотеках, которые пришлось бы непременно устанавливать. А касательно базы данных – была выбрана база данных PostgreSQL. У автора до написания этой курсовой не было опыта работы с БД или с какой-либо ORM, поэтому выбор был сделан исключительно благодаря советам более опытных людей в этой сфере.

Клиент и всё о нём

Клиентская часть приложения, как было описано ранее, разрабатывалась на TypeScript, React, MobX. Это одностраничное многокомпонентное приложение. Суть в том, что фактическая перезагрузка страниц практически никогда и не происходит. Все страницы сайта представляют из себя какие-то компоненты, которые “прячутся” или “отображаются” в зависимости от URL. В этом и есть суть SPA – Single Page Application. Потому что по факту все страницы (то есть компоненты, так как страница – это тоже компонент с комбинацией других компонентов внутри себя) уже загружены при стартовой загрузке приложения, просто не все из них являются видимыми.
При разработке данного курсового проекта, как и при разработке других подобных приложений, всегда следует максимально декомпозировать и переиспользовать компоненты. Например, создать универсальный компонент “Button”, который можно стилизовать каким-то набором предустановленных разработчиком стилей и в котором можно настраивать поведение при клике. И делать это из разных частей программы.

Примеры таких универсальных компонентов, которые, немного кастомизируясь, могут быть использованы в данном курсовом проекте:

- Button
- Grid
- Row
- Container
- Input
- MovieGrid
- Loading

Каждый компонент стилистически независим. Это значит, что можно очень легко переиспользовать компонент в других проектах, просто скопировав папку с ним в другой проект, потому что, в общем случае, компонент индивидуален и “не тянет” каких-то внешних стилистических зависимостей. По крайней мере, речь о компонентах из листьев дерева компонентов данного приложения. Каждый компонент располагается в TSX-файле и сопровождается CSS-модулем. Например, на рисунке 2 представлен пример компонента Input.

Input.tsx

```tsx
interface InputProps {
    placeholder: string,
    type?: string,
    onEnter?: InputCallback,
    onChange?: InputCallback,
    styles?: React.CSSProperties
}

export const Input = (props: InputProps): JSX.Element => {
    const [state, setState] = React.useState<string>( initialState: '')

    const inputChanged = (event: React.ChangeEvent<HTMLInputElement>): void => {

    const onEnterPressed = (event: React.KeyboardEvent<HTMLInputElement>): void

    return (
        <input type={props.type || 'text'}
               className={Style.input}
               style={props.styles}
               placeholder={props.placeholder}
               onChange={event => inputChanged(event)}
               onKeyPress={event => onEnterPressed(event)}
        />
    )
}
```

Input.module.css

```css
.input {
    padding: 7px 10px;
    color: #333;
    border: 1px solid var(--inactive-font-color);
    border-radius: 3px;
    font-size: var(--default-font-size);
    background-color: transparent;
    outline: none;
}

.input::placeholder {
    color: var(--inactive-font-color);
}
```

Рисунок 2. Пример компонента

Как видно, слово компонент более не значит просто HTML-блок со стилями. Теперь это функциональная единица, зачастую с немаленькой внутренней логикой, состоянием и т.д.

На рисунке 3 представлен примерный список компонентов из данного приложения. Это всё либо неделимые компоненты, либо какие-то

объединяющие компоненты (по типу сетки для карточек фильмов или блока информации про кинотеатр), но пока что не страницы.

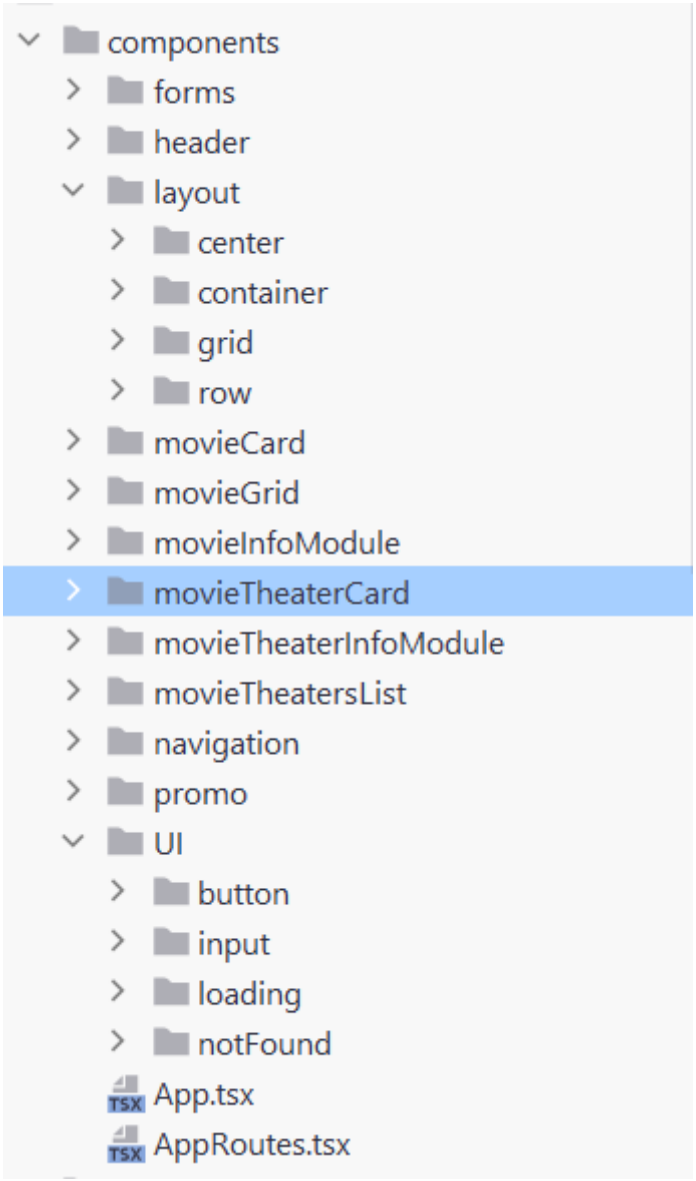


Рисунок 3. Компоненты

Если говорить про страницы, то их список представлен на рисунке 4.
То есть в плане компонентов, это мультистраничное приложение. И, как видно, в нём существует около десятка типов страниц. На рисунке 5 представлен для примера компонент страницы с детальным отображением фильма. Как видно, это просто сложный компонент из других компонентов, который, как мы выясним позже, просто связан с URL страницы, но об этом будет излагаться ниже.

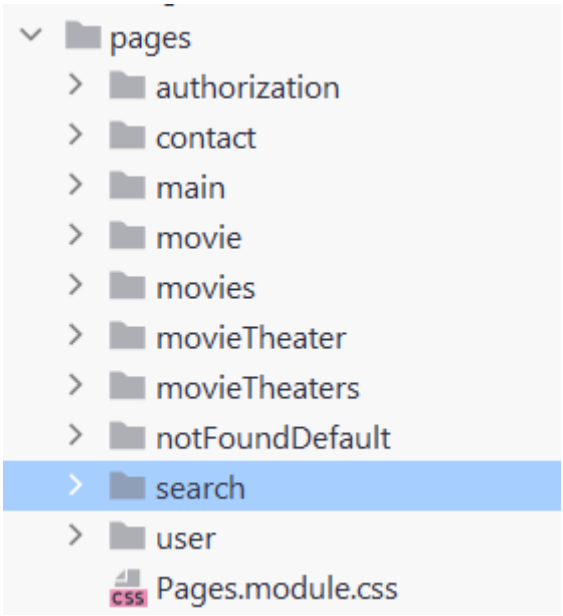


Рисунок 4. Страницы

```
export const MoviePage = observer( baseComponent: (props: { state: MainState }): JSX.Element => {
    React.useEffect( effect: (): void => {
        window.scrollTo( options: {top: 0, left: 0, behavior: 'smooth'})
        const id: string = new URLSearchParams(window.location.search).get('id') || ''
        props.state.loadMovie(id)
    }, deps: [])


    return (
        <div className={StylePages.smoothLoading}>
            <main className={StylePages.main}>
                {
                    props.state.moviePageState.loading === LoadingState.LOADING
                        ?
                        <Loading/>
                        :
                        props.state.moviePageState.movie === null
                            ?
                            <NotFound/>
                            :
                            <MovieInfoModule state={props.state}/>
                }
            </main>
        </div>
    )
})
```

Рисунок 5. Пример страницы

Как видно, на странице с фильмом избирательно отображаются три блока в зависимости от состояния. Это компонент загрузки, который отображается во время запроса на сервер за фильмом. Это компонент “Не найдено” в случае, если сервер сообщил, что такого фильма не существует. Это компонент информации о фильме, который отображается

после того, как запрос на сервер состоялся и сервер ответил положительно, то есть конкретными данными о найденном кино.
Хорошо, мы выяснили кое-что о компонентах данного проекта. Как же страницы отображаются в зависимости от URL ? Это происходит благодаря библиотеке React-Router-Dom и настройке обёртки для всех страниц. Демонстрация того, как это выглядит, на рисунке 6.

```
export const AppRoutes = observer( baseComponent: ({mainState}: {mainState: MainState}): JSX.Element => {
    const navigate: NavigateFunction = useNavigate()

    const callbackForSearch = (value: string): void => {...}

    return (
        <>
            <Header callbackForSearch={callbackForSearch} state={mainState}/>
            <Routes>
                <Route path={'/'} element={<MainPage state={mainState}/>}/>
                <Route path={'/contact'} element={<ContactPage/>}/>
                <Route path={'/movies'} element={<MoviesPage state={mainState}/>}/>
                <Route path={'/movie'} element={<MoviePage state={mainState}/>}/>
                <Route path={'/movie-theaters'} element={<MovieTheatersPage state={mainState}/>}/>
                <Route path={'/movie-theater'} element={<MovieTheaterPage state={mainState}/>}/>
                <Route path={'/search-movie'} element={<SearchPage state={mainState}/>}/>
                <Route path={'/signin'} element={<SignInPage state={mainState}/>}/>
                <Route path={'/login'} element={<LogInPage state={mainState}/>}/>
                <Route path={'/account'} element={<UserPage state={mainState}/>}/>
                <Route path={'*'} element={<NotFoundDefaultPage/>}/>
            </Routes>
        </>
    )
})
```

Рисунок 6. Роутинг

На этом и других скриншотах, вероятно, можно обратить внимание на обёртку для каждого компонента “observer”. Это функция из библиотеки для менеджинга состояния MobX.
Как было описано выше, каждый компонент может внутри себя хранить состояние. Но что насчёт каких-то глобальных состояний ? Например, состояние зарегистрированности пользователя ? Для этого автор использует отдельный класс состояний. Более того – почти абсолютно все состояния вынесены в этот класс. Например, массив уже загруженных карточек фильмов, чтобы не загружать их заново при повторном обращении к странице ‘/movies/’. Или состояние загрузки для той или иной страницы. И всякие другие состояния и полезные функции. Для этого всего в этом проекте используется класс MainState. Он хранит в себе данные, а также является посредником между компонентами и ещё одним

классом, который занимается запросами на сервер. Переменные для состояний в этом классе показаны на рисунке 7.

```
export class MainState {
    private static readonly HOW_MANY_TO_LOAD: number = 4
    private static readonly LOCAL_STORAGE_JWT_KEY: string = 'authTokens'
    private static readonly REFRESH_TOKEN_INTERVAL: number = 1000 * 60 * 4

    public readonly controller: MoviesServiceCore

    public readonly moviesPageState: MoviesPageState
    public readonly mainPageState: MainPageState
    public readonly moviePageState: MoviePageState
    public readonly movieTheatersPageState: MovieTheatersPageState
    public readonly movieTheaterPageState: MovieTheaterPageState
    public readonly searchPageState: SearchPageState
    public readonly userPageState: UserPageState

    public user: User | null

    private readonly refreshTokenIntervalID: number
```

Рисунок 7. Класс состояния всего приложения

Также тут находятся обработчики, которые связывают события в компонентах и последующие необходимые запросы на сервер.
Раз уж был упомянут класс взаимодействия с сервером, то автор предлагает сразу же и описать его. Это класс MoviesServiceCore. Непосредственно он уже посылает запросы на сервер. Он знает все URL и типы запросов, которые нужно послать. Callback для запросов формируется в MainState и посылается уже в Core. Core лишь собирает все параметры (тело, колбек) в один запрос и отсылает его. Callback всегда представляет из себя какую-то функцию, которая при получении ответа от сервера изменяет состояние какой-то переменной в классе. А уже из-за паттерна observer в MobX и React-Mobx это вызывает нужные перерисовки в компонентах страницы. Таким образом, если упрощённо, работает клиент.
А теперь несколько конкретных примеров.
Допустим, есть страничка с фильмом. Изначально она пустая. Как понять, какой фильм запросить из сервера и отобразить на странице ? При формировании списка с превью фильмов каждое превью имеет ссылку вида ‘/movie/?id={…}’. То есть конкретный фильм можно получить из

GET-параметра клиентского URL. Таким образом при заходе на страницу ‘/movie/?id={…}’ изначально вытягивается значение этого GET-параметра, даётся сигнал классу MainState, чтобы он по такому-то movie-id загрузил фильм в состояние страницы ‘/movie/’. MainState формирует данные для запроса, “связывается” с MoviesServiceCore — и уже тот класс делает непосредственно запрос на серверу. Рисунок 8.

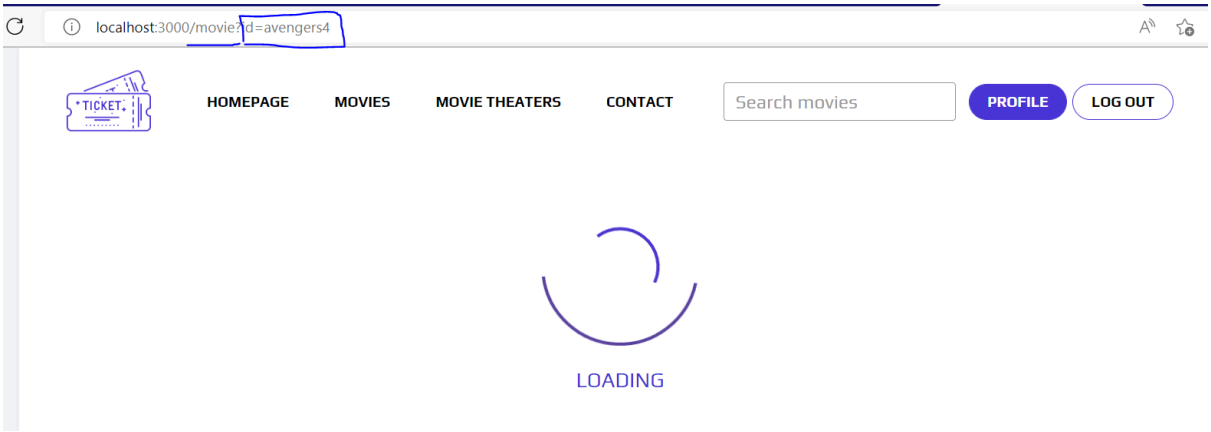


Рисунок 8.

На этом касательно работы фронт-энда, кажется, всё. Только дополнительно автор хотел бы обратить внимание на дизайн приложения. Цветовая палитра содержится в CSS-переменных в файле ‘index.css’, поэтому можно относительно быстро поменять цветовую гамму приложения, изменив несколько строчек в этом файле. Таким образом возможности, например, для внедрения тёмной темы достаточно легко реализуемы.
Приложение наполнено анимациями. Это обеспечивают такие возможности CSS, как CSS Animation Keyframes и CSS Transitions. Анимированы нажатия на кнопки, наведения на кнопки, колесо загрузки, плавное возникновение страниц и др.

Сервер и всё о нём

Как было упомянуто выше, серверная часть писалась на Python, Django Django-Rest-Framework и PostgreSQL. Важно понимать, что Django-Rest-Framework — это не отдельный фреймворк, а лишь своего рода надстройка над большим “гигантом” Django, которая добавляет сериализаторы и некоторые другие вещи, позволяющие строить уже не просто веб-приложение, а бэк-енд fullstack-приложения. То есть все модели, представления, ORM — всё это остаётся. Итак, немного про

концепцию такого типа приложений, как это видит и понимает автор данной курсовой работы.
В fullstack-приложении клиент и сервер “крутятся” отдельно. Отдельно на своём хостинге (подхостинге) “крутится” задеплоенное React-приложение, а отдельно на своём – Django-сервер. Сервер в данном случае также представляет из себя что-то вроде сайта. На него по каким-то установленным URL прилетают запросы (речь об HTTP-запросах с параметрами или без) — а он, в свою очередь, “отдаёт” обратным ответом какие-то данные в виде, например, объекта JSON.
То есть с одним понятием уже определились – URL — это просто своего интерфейс сервера, способ общения с сервером для получения каких-то данных. Далее вводится понятие “представление”. Это (в случае конкретно данного проекта, по крайней мере) просто тот класс или та функция, которая определяет, что вообще делать серверу после того, как на тот URL, к которому привязано это представление, “постучались”. То есть к URL привязывается какое-то представление, которое определяет поведение при доступе к этому URL. Пример на рисунке 8.
Имеется какой-то URL ‘/api/movies/’ и соответствующее представление MoviesAPIView. Оно диктует такие правила, что при поступлении GET-запроса по данному URL сервер должен вернуть первые 4 фильма из базы данных. Или вернуть какой-то интервал фильмов, если запрос поступил по URL ‘/api/movies/0/5/’. Последний вернёт все фильмы с нулевого по пятый. В случае отсутствия какой-то части (из конца) в базе сервер вернёт только доступную часть. Такие ситуации обрабатываются. Просто если что-то не так с корректностью запроса, то сервер всё равно отработает и просто вернёт нормальный ответ с пояснением ошибки. Демонстрация подобного сопоставления URL и View на рисунке 9, как уже было написано выше.

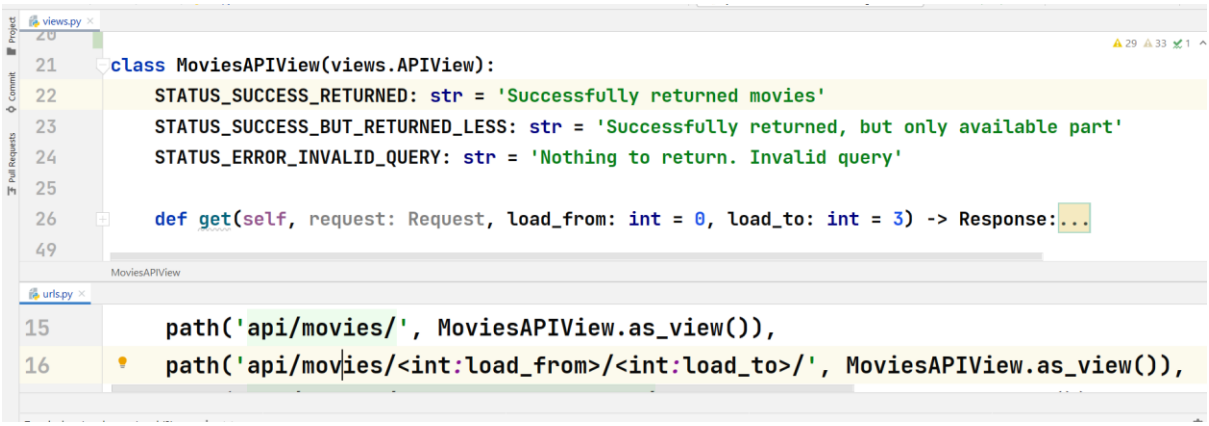


Рисунок 9. Связанное представление и путь

Хорошо, с представлениями и URL более-менее определение введено. Что насчёт модели ? Дело в том, что в Django есть встроенная ORM. Это инструмент, чтобы работать с базой данных, минуя язык запросов (например, SQL) к БД, а работать с ней с помощью классов и определённых для них методов прямо внутри кода Python. Например, чтобы просто определить модель кинотеатра, нужно создать вот такую структуру, как на рисунке 10. Затем только необходимо произвести миграции — и в базе данных появится такая вот модель, то есть таблица. Миграция – это в данном случае перенос “виртуальной” модели из Python в настоящую якобы “физическую” модель в самой базе данных.

```
29  class MovieTheater(models.Model):
30      title = models.CharField(max_length=200)
31      address = models.CharField(max_length=300)
32      location = models.CharField(max_length=600)
33      photo = models.CharField(max_length=600, default='photo')
34      telephone = models.CharField(max_length=80, default='+375')
35      visits_count = models.IntegerField(default=0)
36      movies = models.ManyToManyField(Movie)
37
38      def __str__(self):
39          return f'Movie theater - {self.title}'
```

Рисунок 10. Модель

Наполнять таблицу можно, например, из какого-нибудь админского приложения по типу PgAdmin или из админской панели, которую фреймворк Django предоставляет из коробки. Важно только не забыть добавить отображение созданной модели в файле ‘admin.py’. Это и непосредственно внешний вид встроенной админки Django предоставлены на рисунках 11 и 12 соответственно.

admin.py

```
1  from django.contrib import admin
2
3  from .models import Movie, MovieTheater, CinematicUniverse, UsersFavourites
4
5  admin.site.register(Movie)
6  admin.site.register(MovieTheater)
7  admin.site.register(CinematicUniverse)
8  admin.site.register(UsersFavourites)
```

Рисунок 11. Регистрация моделей для отображения в панели администратора

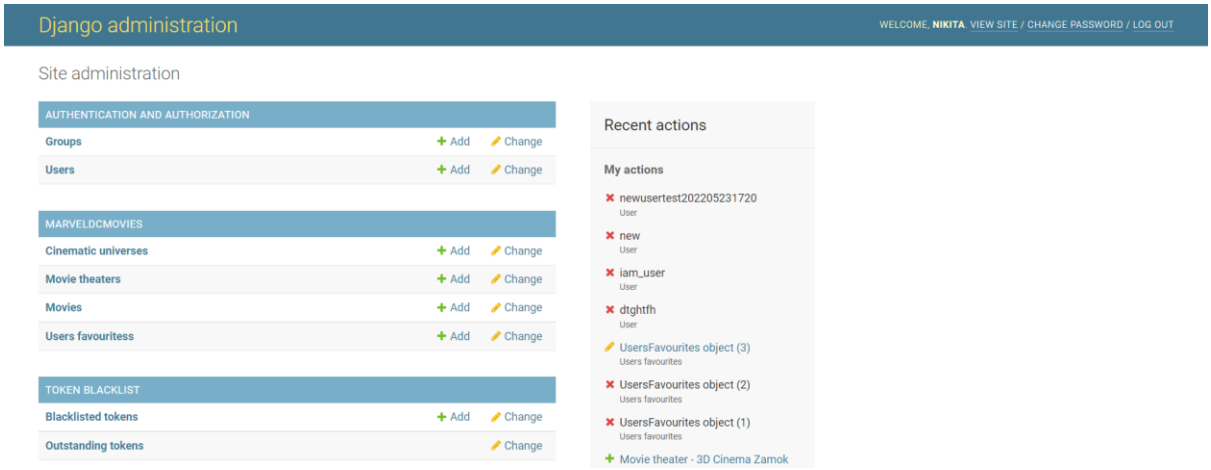


Рисунок 12. Панель администратора

Из панели администратора добавлялись фильмы и кинотеатры в базу данных. А также категории фильмов. И новые пользователи, пока не была добавлена регистрация.
Выше было введено понятие сериализатора. В случае с Django-Rest-Framework сериализатор – это всего-навсего класс, который определяет, какие поля “запаковывать” при отправке на клиент объекта из базы данных. Например, в вышеописанной модели ‘MovieTheater’ есть поле ‘visits_count’. При отображении деталей кинотеатра это поле не нужно. Да и нет необходимости клиенту знать о такой “более личной информации”. Она используется только на сервере. В том числе и про формировании популярной подборки представлением с рисунка 13.

```
103 class MostPopularMovieTheaterAPIView(views.APIView):
104     def get(self, request: Request) -> Response:
105         return Response({
106             'data': MovieTheaterSerializer(MovieTheater.objects.all().order_by('-visits_count')[:2], many=True).data
107         })
```

Рисунок 13. Представление для популярных фильмов

Сам же сериализатор, например, для модели кинотеатра имеет вид с рисунка 14.

```
25 class MovieTheaterSerializer(serializers.ModelSerializer):
26     class Meta:
27         model = MovieTheater
28         exclude = ['visits_count', 'id']
```

Рисунок 14. Сериализатор

Теперь немного подробнее об авторизации, так как это важный момент. Авторизация и аутентификация построена с помощью встроенных возможностей Django, а также с помощью доп. Библиотеки для работы с JWT токенами. Суть такова. На сервер в первый раз поступает запрос, в случае конкретно этого проекта, на URL ‘/api/token/’ с телом, содержащим имя пользователя и пароль. Генерируются два токена. Это access-токен и refresh-токен. Первый нужен для доступа к ресурсам. По нему определяется пользователь. Это можно увидеть на сайте jwt.io. Рисунок 15.

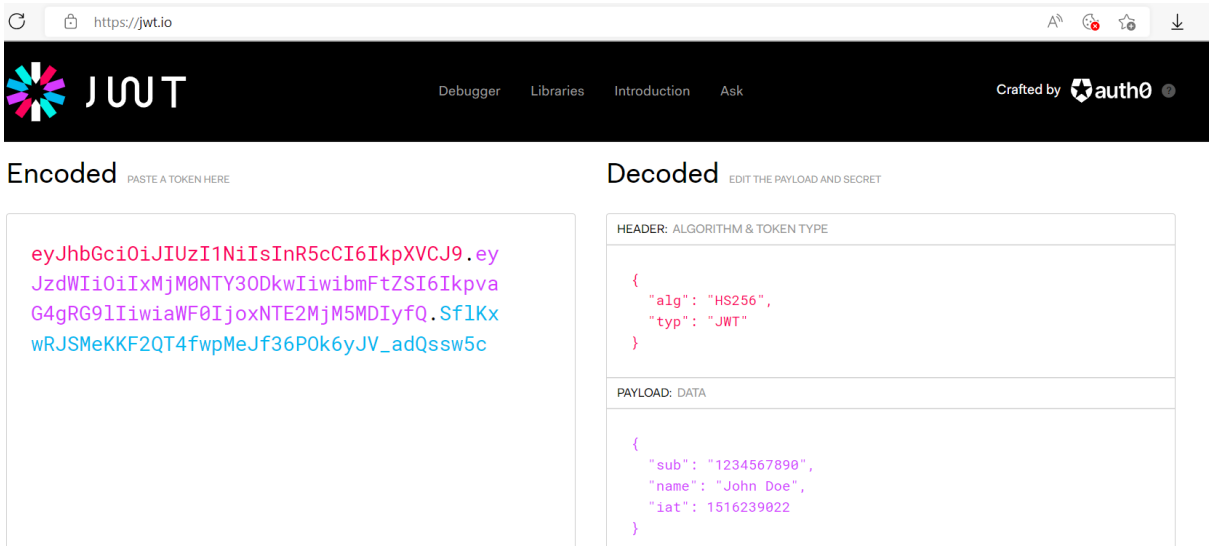


Рисунок 15. Скриншот с сайта JWT

Когда пользователь логинится, сервер высылает пару токенов access-refresh. Далее при каждом запросе, который требует залогиненного состояния, вместе с запросом со стороны клиента необходимо также в заголовке отправлять этот access-токен. Если он действителен, то представление вернёт то, что нужно. Иначе придёт сообщение, что данные авторизации не предоставлены. Но access-токен “живёт” ограниченное количество времени. Его разработчик прописывает в файле ‘settings.py’. Поэтому с помощью refresh-токена надо время от времени обновлять статус авторизации. При отправке на URL ‘/api/token/refresh/’ текущего refresh-токена сервер отдаст новую пару refresh-access. Таким образом уже refresh-токен обеспечивает продлевание авторизованности пользователя. Клиент обновляет токены каждые 5 минут. Потому что на сервере установлена продолжительность жизни access-токена в 5 минут. Refresh-токен может “жить” дольше. В этом проекте это 10+ дней. То есть если не

входить и не выходить, а просто использовать сайт каждый день в течение 10 дней (или не использовать), то через 10 дней сайт разлогинится.

Конкретный пример взаимодействия сервера и клиента

Рассмотрю конкретнее пример формирования списков кинотеатров и последующую возможность перехода на страницу для конкретного кинотеатра.
Пользователь переходит на страницу ‘/movie-theaters’. Рисунок 16.

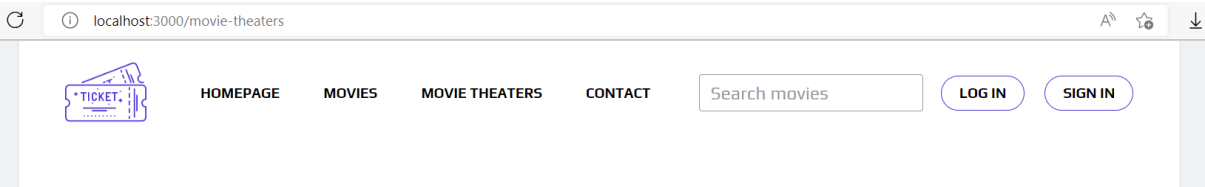


Рисунок 16.

При загрузке компонента MovieTheatersPage, который и являет собой, собственно, эту страницу, отдаётся команда в класс MainState. Этот класс начинает работать с переменной для состояния страницы MoviePage. “Навешивает” статус загрузки (и, следовательно, отображается компонент Loading), после чего формирует запрос и посылает команду классу MoviesServiceCore сделать запрос на сервер. Класс Core отправляет запрос на сервер по URL ‘/api/movietheaters/’. Можно видеть в логировании Django, что запрос действительно прилетел (рисунок 17).

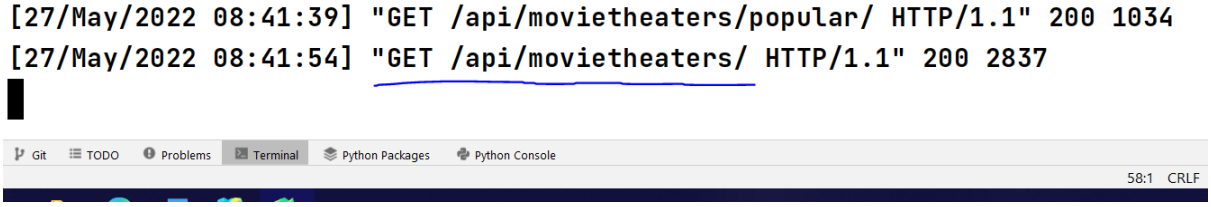
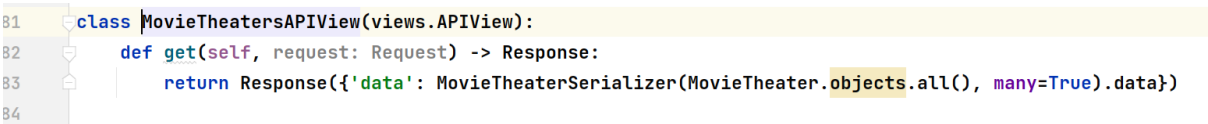


Рисунок 17

Запрос начинает обрабатываться привязанным View. Это класс MovieTheatersAPIView. Рисунок 19.



```
class MovieTheatersAPIView(views.APIView):
    def get(self, request: Request) -> Response:
        return Response({'data': MovieTheaterSerializer(MovieTheater.objects.all(), many=True).data})
```

Рисунок 19

Он формирует выборку (посредством ORM Django) всех кинотеатров из базы данных, сериализует его с помощью MovieTheaterSerializer, описанного выше, и отправляет массив таких данных обратно на клиент. В это время на клиенте по-прежнему висит состояние загрузки и отображается красивая загрузочная анимация. Как только данные получены, срабатывает функция обратного вызова, которая передаётся в запрос. Она принимает данные, снимает состояние загрузки со страницы кинотеатров и записывает данные о кинотеатрах в это же состояние (другое поле). Таким образом со страницы пропадает компонент загрузки, а вместо него уже отрисовывается список кинотеатров. Это всё происходит динамически, за счёт использования вышеописанного стейт-менеджера MobX и паттерна Observer, который и использует этот менеджер. Формируются карточки с кинотеатрами. В каждом содержится ссылка вида '/movie-theater?cinema-name={…}', где вместо многоточия подставляется id, который "прилетел" из базы.
При переходе по такой ссылке React-Router-Dom отрисовывает уже другую страницу (то есть компонент-страницу). Эта страница уже предназначена для конкретного кинотеатра. Что там происходит ? Страница (компонент) при первом открытии "забирает" GET-параметр cinema-name. И всё также посылает запрос через MainState и Core на сервер. Уже на URL '/api/movietheaters/{…}/', где вместо многоточия идёт вытянутый GET-параметр. Всё то же самое. На страницу с кинотеатром навешивается состояние загрузки. После того, как данные прилетели, это состояние снимается, а в данные записывается объект с кинотеатром. И полный блок информации о кинотеатре отображается на странице.

Дополнительно могут также посылаться запросы на какие-то дополнительные данные. Например, в странице с фильмом дополнительно посылается второй запрос на список кинотеатров, в которых можно посмотреть этот фильм.

Методика работы с полученной программой

При заходе на главную страницу сайта пользователь видит меню навигации, промо-блок и далее список четырёх самых популярных

фильмов и двух самых популярных кинотеатров. Эта подборка формируется благодаря счётчику посещений. Рисунок 20.

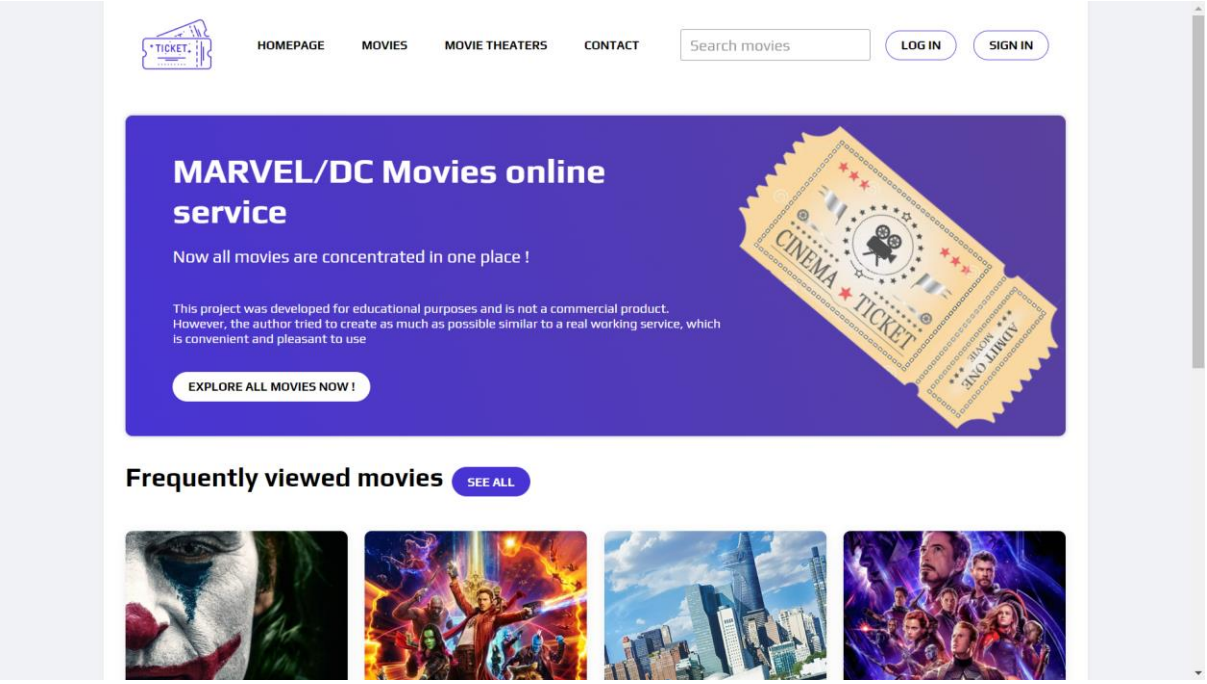


Рисунок 20

Другие страницы (movies, movie-theaters, contact) имеют вид на рисунках 21-23.

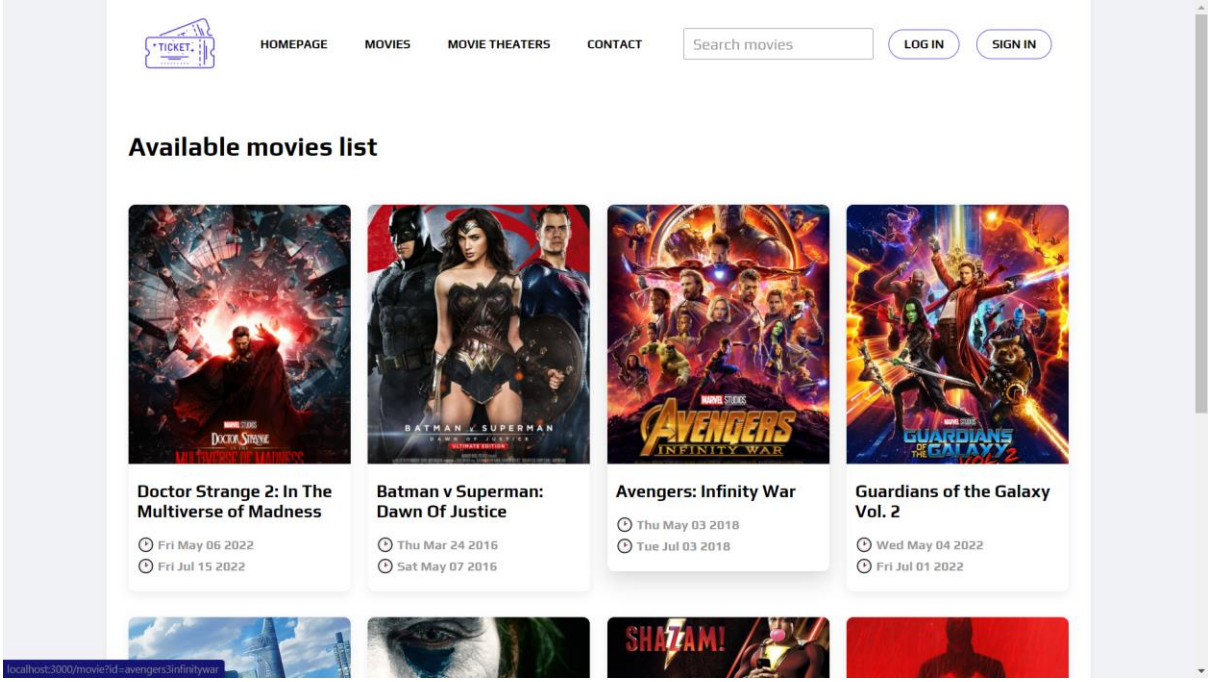


Рисунок 21

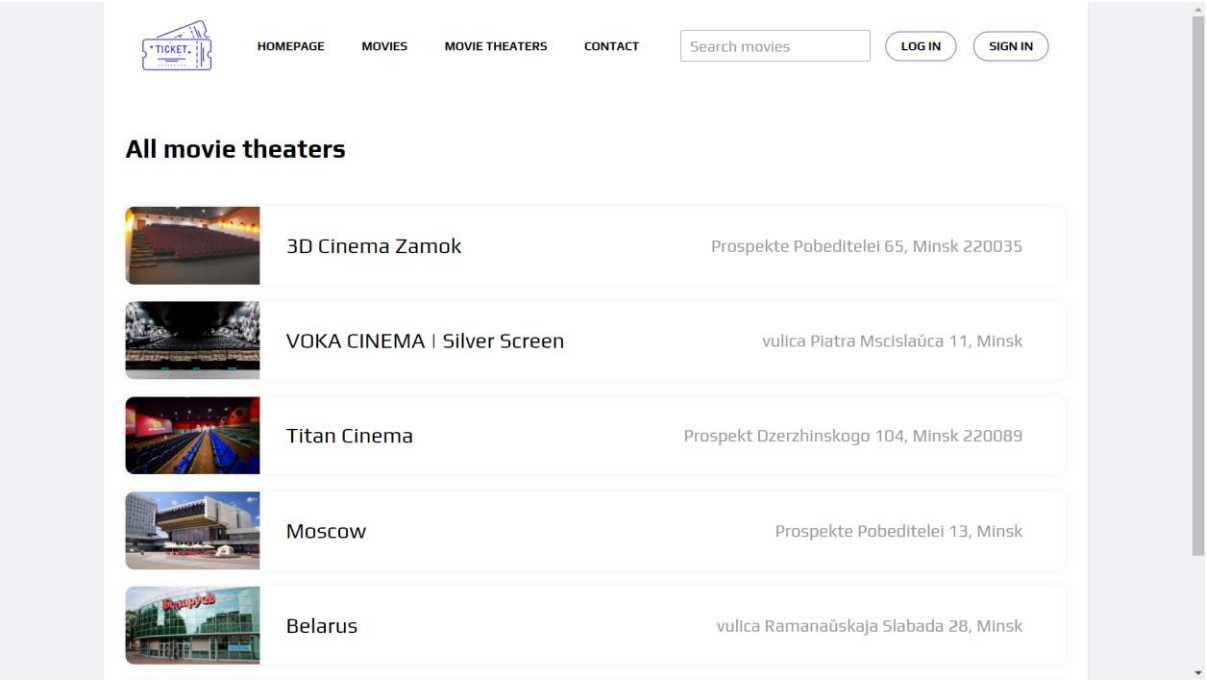


Рисунок 22

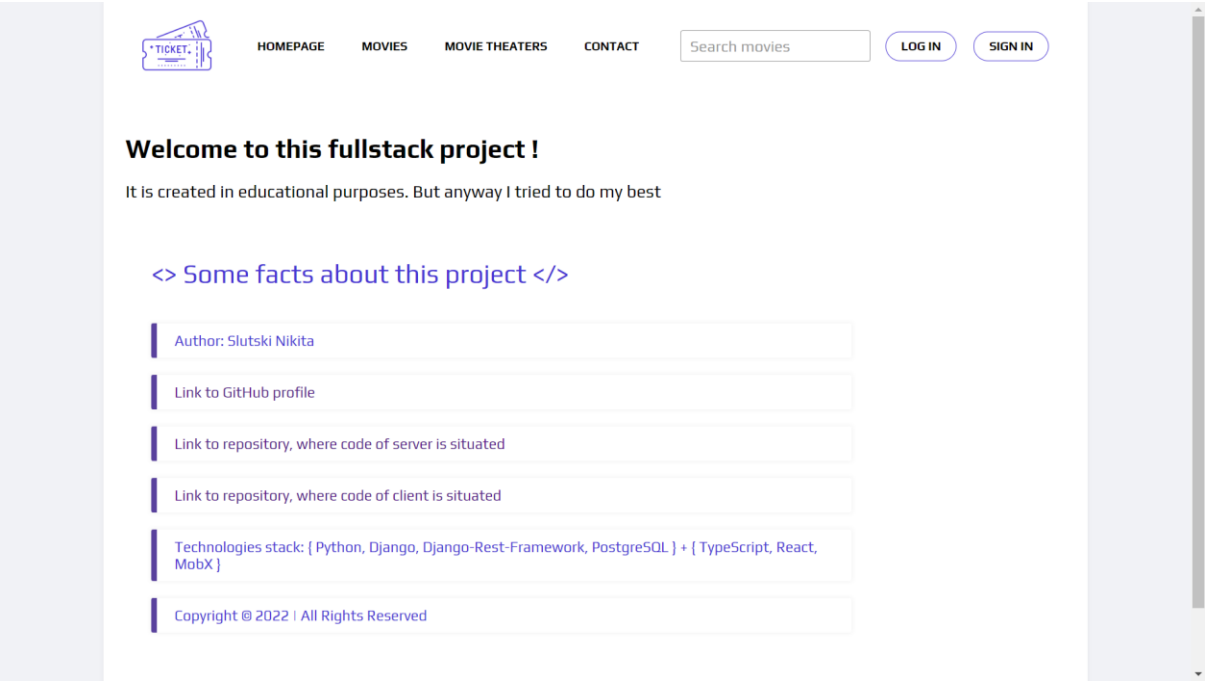


Рисунок 23

Страница авторизации имеет вид с рисунка 24.

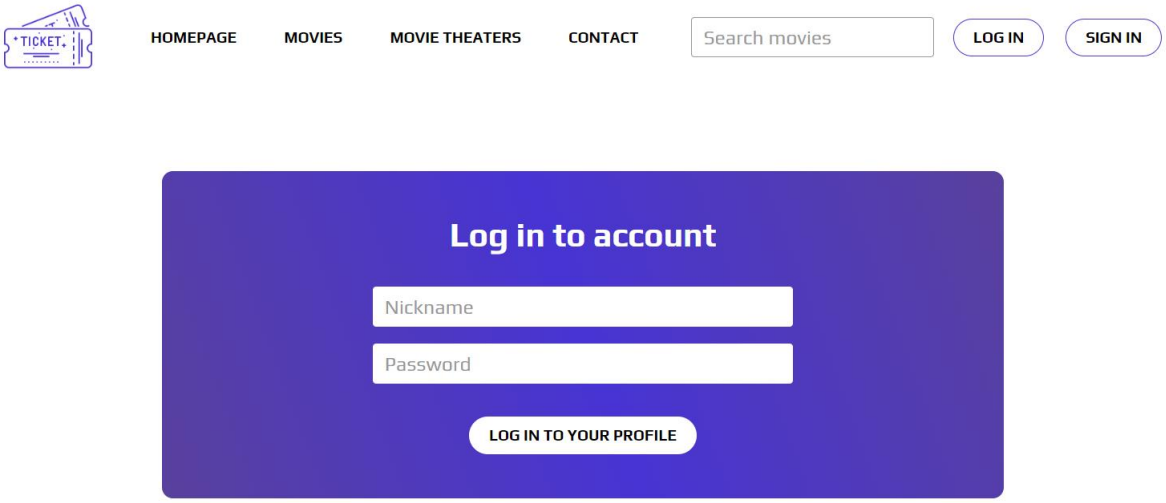


Рисунок 24

Страница конкретного фильма и конкретного кинотеатра имеют вид, представленный на рисунке 25-26.

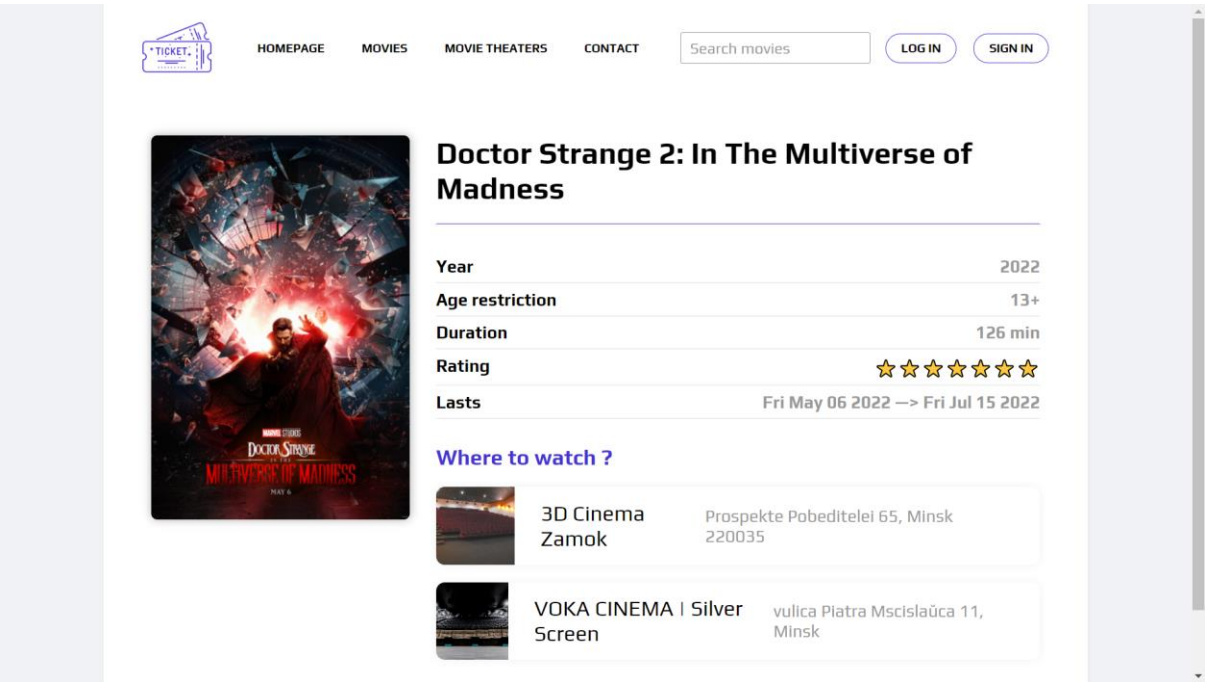


Рисунок 25

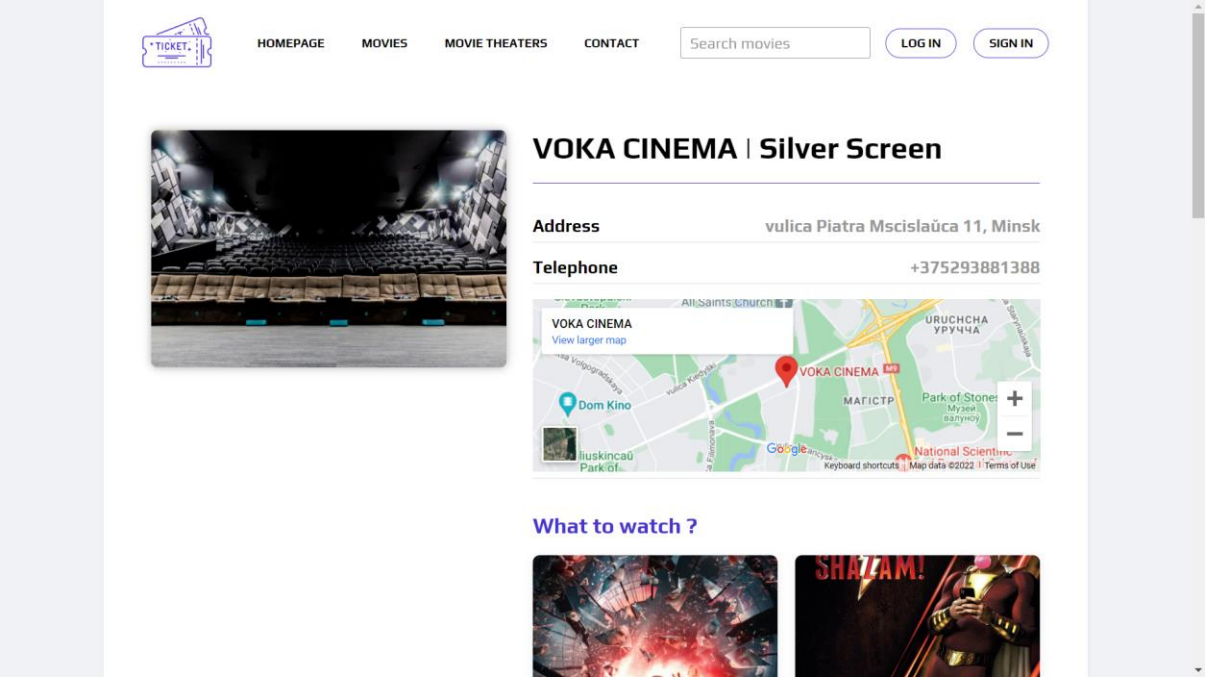


Рисунок 26

Гости сайта имеют возможность просматривать фильмы, кинотеатры. Производить поиск по фильмам.
Авторизированный пользователь может добавлять и удалять фильмы из избранного. Это может быть полезно для последующего возврата к заинтересовавшим фильмам. Рисунок 27

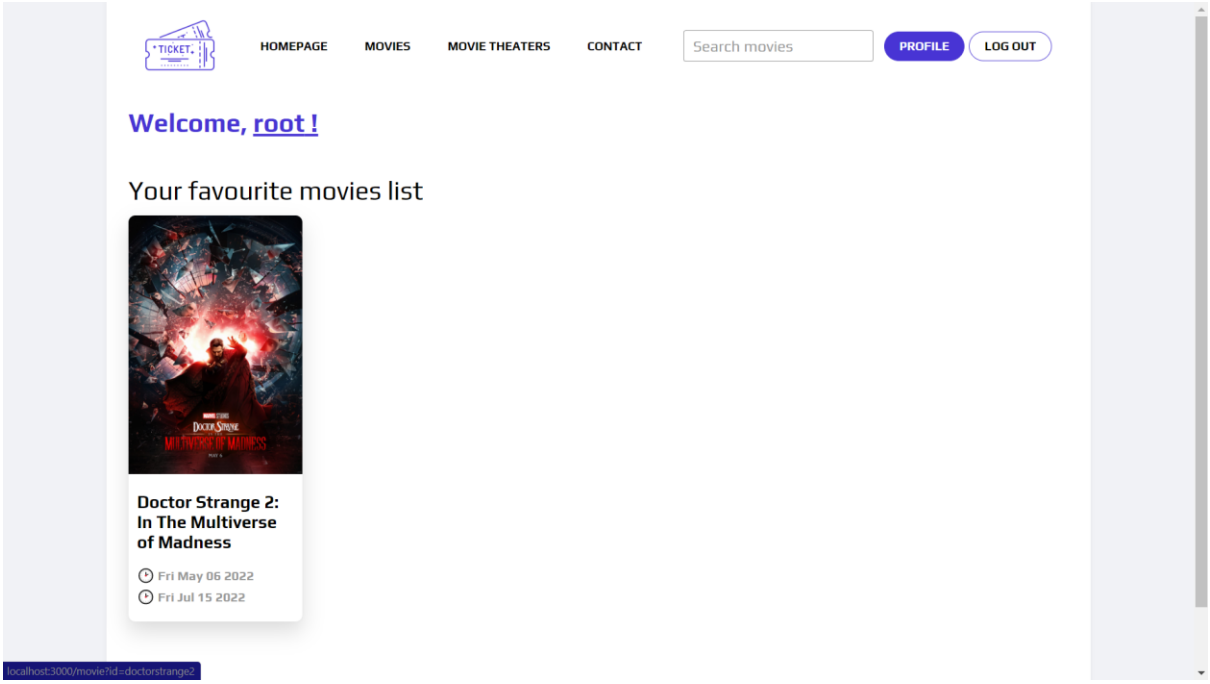


Рисунок 27

Заключение

В результате проделанной работы был изучен принцип построения несложных, но полноценных fullstack-приложений. То есть полностью с нуля была разработана как интерфейсная часть, так и логика клиента, так и логика сервера.
Цели данной курсовой работы можно считать достигнутыми. В теории и на практике было проработано RESTful приложение. Это даёт хороший старт для понимания того, как функционирует настоящий глобальный веб-мир в жизни.

Литература

[1] Django-documentation [Электронный ресурс]:– Электронные данные. – Режим доступа: https://djangoproject.com

[2] Django-rest-framework documentation [Электронный ресурс]:– Электронные данные. – Режим доступа: https://django-rest-framework.org

[2] React documentation [Электронный ресурс]:– Электронные данные. – Режим доступа: https://reactjs.org

Приложение 1. Исходный код

Примечание. Предоставлен только код непосредственно логики приложения. То есть классов MainState и MoviesServiceCore из клиента и написанной автором логики из сервера. Код компонентов опускается.

Клиент:
MainState

```
import {makeAutoObservable} from 'mobx'
import {MoviesServiceCore} from '../core/MoviesServiceCore'
import {MoviesPageState} from './MoviesPageState'
import {LoadingState} from '../LoadingState'
import {RequestCallback} from '../RequestCallback'
import {
    ServerResponseForFullMovie,
    ServerResponseForMoviesForTheater,
    ServerResponseForMoviesList,
    ServerResponseForMoviesSearch,
    ServerResponseForMovieTheater,
    ServerResponseForTheatersList
} from '../ServerResponse'
import {MainPageState} from './MainPageState'
import {MoviePageState} from './MoviePageState'
import {MovieTheatersPageState} from './MovieTheatersPageState'
import {MovieTheaterPageState} from './MovieTheaterPageState'
import {SearchPageState} from './SearchPageState'
import jwtDecode from 'jwt-decode'
import {User} from '../User'
import {UserPageState} from './UserPageState'


export class MainState {
    private static readonly HOW_MANY_TO_LOAD: number = 4
    private static readonly LOCAL_STORAGE_JWT_KEY: string = 'authTokens'
    private static readonly REFRESH_TOKEN_INTERVAL: number = 1000 * 60 * 4

    public readonly controller: MoviesServiceCore

    public readonly moviesPageState: MoviesPageState
    public readonly mainPageState: MainPageState
    public readonly moviePageState: MoviePageState
    public readonly movieTheatersPageState: MovieTheatersPageState
    public readonly movieTheaterPageState: MovieTheaterPageState
    public readonly searchPageState: SearchPageState
    public readonly userPageState: UserPageState

    public user: User | null

    private readonly refreshTokenIntervalID: number


    public constructor() {
        this.controller = new MoviesServiceCore()

        this.moviesPageState = {moviesCardsLoaded: [], loading: LoadingState.LOADING, showLoadMoreButton: true}
        this.mainPageState = {
            popularMovies: {loadedPopularMovies: [], loading: LoadingState.LOADING},
            popularMovieTheaters: {loadedPopularMovieTheaters: [], loading: LoadingState.LOADING}
        }
        this.moviePageState = {loading: LoadingState.LOADING, movie: null, theatersList: [], isFavourite: false}
        this.movieTheatersPageState = {loading: LoadingState.LOADING, movieTheatersLoaded: []}
        this.movieTheaterPageState = {loading: LoadingState.LOADING, theater: null}
        this.searchPageState = {loading: LoadingState.LOADING, foundMovies: []}
        this.userPageState = {favourites: []}

        this.user = null

        this.checkLogIn()

        this.refreshTokenIntervalID = window.setInterval((): void => {
            this.user !== null && this.updateToken()
        }, MainState.REFRESH_TOKEN_INTERVAL)

        makeAutoObservable(this, {}, {deep: true})
    }
```

```
public loadMoreMovies(howMany: number = MainState.HOW_MANY_TO_LOAD): void {
    this.moviesPageState.loading = LoadingState.LOADING

    const currentLoadedTo: number = this.moviesPageState.moviesCardsLoaded.length
    const newLoadedTo: number = currentLoadedTo + howMany - 1

    const onMoviesLoad: RequestCallback = (data, error) => {
        this.moviesPageState.loading = LoadingState.LOADED

        if (error) throw new Error(error.message)

        const parsedResponse: ServerResponseForMoviesList = JSON.parse(data!)

        if (parsedResponse.success) {
            this.moviesPageState.moviesCardsLoaded.push(...Array.from(parsedResponse.data || []))
            this.moviesPageState.showLoadMoreButton = parsedResponse.howManyLeft !== 0
        } else {
            throw new Error(`Something went wrong. Error from server: ${parsedResponse.status}`)
        }
    }

    this.controller.getMovies(onMoviesLoad, currentLoadedTo, newLoadedTo)
}

public loadPopularMovies(): void {
    const onPopularMoviesLoad: RequestCallback = (data, error) => {
        this.mainPageState.popularMovies.loading = LoadingState.LOADED

        if (error) throw new Error(error.message)

        const parsedResponse: ServerResponseForMoviesList = JSON.parse(data!)

        this.mainPageState.popularMovies.loadedPopularMovies = Array.from(parsedResponse.data as [])
    }

    this.controller.getPopularMovies(onPopularMoviesLoad)
}

public loadPopularTheaters(): void {
    const onPopularTheaterLoad: RequestCallback = (data, error) => {
        this.mainPageState.popularMovieTheaters.loading = LoadingState.LOADED

        if (error) throw new Error(error.message)

        const parsedResponse: ServerResponseForTheatersList = JSON.parse(data!)

        this.mainPageState.popularMovieTheaters.loadedPopularMovieTheaters = parsedResponse.data
    }

    this.controller.getPopularTheaters(onPopularTheaterLoad)
}

public loadMovie(id: string): void {
    this.moviePageState.loading = LoadingState.LOADING

    const onMovieLoad: RequestCallback = (data, error) => {
        if (error) throw new Error(error.message)

        const parsedResponse: ServerResponseForFullMovie = JSON.parse(data!)

        if (parsedResponse.success) {
            this.moviePageState.movie = parsedResponse.data
            this.loadTheatersForMovie(id)
            this.checkIfInFavourite(id)
        } else {
            this.moviePageState.loading = LoadingState.LOADING
            this.moviePageState.movie = null
        }
    }

    this.controller.getMovieById(id, onMovieLoad)
}

public loadMovieTheatersList(): void {
    const onTheatersLoad: RequestCallback = (data: string | null, error: Error | undefined) => {
        this.movieTheatersPageState.loading = LoadingState.LOADED

        if (error) throw new Error(error.message)

        const parsedResponse: ServerResponseForMoviesList = JSON.parse(data!)

        console.log(parsedResponse)

        this.movieTheatersPageState.movieTheatersLoaded = Array.from(parsedResponse.data as [])
    }

    this.controller.getMovieTheaters(onTheatersLoad)
}
```

```
public loadMovieTheater(title: string): void {
    this.movieTheaterPageState.loading = LoadingState.LOADING

    const onTheaterLoad: RequestCallback = (data, error) => {
        this.movieTheaterPageState.loading = LoadingState.LOADED

        if (error) throw new Error(error.message)

        const parsedResponse: ServerResponseForMovieTheater = JSON.parse(data!)

        if (parsedResponse.success) {
            this.movieTheaterPageState.theater = parsedResponse.data
            this.loadMoviesForTheater(parsedResponse.data.movies as [])
        } else
            this.movieTheaterPageState.theater = null
    }

    this.controller.getMovieTheaterByTitle(title, onTheaterLoad)
}

public loadMoviesForTheater(movies: Array<string>): void {
    const onTheaterLoad: RequestCallback = (data, error) => {
        this.movieTheaterPageState.loading = LoadingState.LOADED

        if (error) throw new Error(error.message)

        const parsedResponse: ServerResponseForMoviesForTheater = JSON.parse(data!)

        this.movieTheaterPageState.theater!.movies = parsedResponse.data
    }

    this.controller.getMoviesForTheater(movies, onTheaterLoad)
}

public loadTheatersForMovie(movieId: string): void {
    const onTheatersLoad: RequestCallback = (data, error) => {
        this.moviePageState.loading = LoadingState.LOADED

        if (error) throw new Error(error.message)

        const parsedResponse: ServerResponseForTheatersList = JSON.parse(data!)

        this.moviePageState.theatersList = parsedResponse.data as []
    }

    this.controller.getTheatersForMovie(movieId, onTheatersLoad)
}

public searchByQuery(query: string): void {
    this.searchPageState.loading = LoadingState.LOADING

    const onMoviesSearchLoad: RequestCallback = (data, error) => {
        this.searchPageState.loading = LoadingState.LOADED

        if (error) throw new Error(error.message)

        const parsedResponse: ServerResponseForMoviesSearch = JSON.parse(data!)

        this.searchPageState.foundMovies = parsedResponse.data as []
    }

    this.controller.searchMovies(query, onMoviesSearchLoad)
}

public authorize(username: string, password: string): void {
    const onAuthorizationCheckPassed = (response: Response, data: any, error: Error | null) => {
        if (error) throw new Error(error.message)

        if (response.status === 200)
            this.setUpLoggedInUser(data)
        else
            window.alert(response.statusText)
    }
    this.controller.authorize(username, password, onAuthorizationCheckPassed)
}

private setUpLoggedInUser(data: any): void {
    const decoded = jwtDecode(data.access)

    this.user = {serverData: decoded, access: data.access, refresh: data.refresh}

    localStorage.setItem(MainState.LOCAL_STORAGE_JWT_KEY, JSON.stringify(data))
}

public logOut(): void {
    this.user = null
    localStorage.removeItem(MainState.LOCAL_STORAGE_JWT_KEY)
```

```
}

private checkLogIn(): void {
    const storedLocally: any = JSON.parse(localStorage.getItem(MainState.LOCAL_STORAGE_JWT_KEY) || 'null')
    if (storedLocally !== null)
        this.user = {
            serverData: jwtDecode(storedLocally.access),
            access: storedLocally.access,
            refresh: storedLocally.access
        }
}

private updateToken(): void {
    const onUpdatePassed = (response: Response, data: any, error: Error | null) => {
        if (error) throw new Error(error.message)

        if (response.status === 200) {
            this.setUpLoggedInUser(data)
        } else {
            window.alert('Something went wrong')
            this.logOut()
        }
    }

    this.user !== null && this.controller.refreshToken(this.user.refresh, onUpdatePassed)
}

public register(username: string, password: string, email: string): void {
    const onRegistrationTryPassed = (response: Response, data: any, error: Error | null) => {
        if (error) throw new Error(error.message)

        if (response.status === 200)
            window.alert('Registered successfully !')
        else
            window.alert(`Django didn't like something ...${response.statusText}`)
    }

    const formData: FormData = new FormData()
    formData.append('username', username)
    formData.append('password', password)
    formData.append('email', email)

    this.controller.register(formData, onRegistrationTryPassed)
}

public addOrRemoveMovieToFavourite(movieId: string, add: boolean = true): void {
    if (add) {
        const onAddingTryPassed = (response: Response, data: any, error: Error | null) => {
            if (error) throw new Error(error.message)
            window.alert(data.status)
            window.location.reload()
        }

        if (this.user !== null)
            this.controller.addToFavourites(movieId, this.user.access, onAddingTryPassed)
        else
            window.alert('Unable. Unauthorized !')
    } else { // => to remove
        const onRemovingTryPassed = (response: Response, data: any, error: Error | null) => {
            if (error) throw new Error(error.message)
            window.alert(data.status)
            window.location.reload()
        }

        if (this.user !== null)
            this.controller.removeFromFavourites(movieId, this.user.access, onRemovingTryPassed)
        else
            window.alert('Unable. Unauthorized !')
    }
}

public getUsersFavourites(): void {
    const onGettingFavourites = (response: Response, data: any, error: Error | null) => {
        if (error) throw new Error(error.message)
        this.userPageState.favourites = data.data
    }

    if (this.user !== null)
        this.controller.getFavourites(this.user.access, onGettingFavourites)
    else
        window.alert('Unable. Unauthorized !')
}

public checkIfInFavourite(movieId: string): void {
    const onCheckPass = (response: Response, data: any, error: Error | null) => {
        if (error) throw new Error(error.message)
        this.moviePageState.isFavourite = data.result === true
    }
```

```
        if (this.user !== null)
            this.controller.checkIfFavourite(movieId, this.user.access, onCheckPass)
    }
}
```

MoviesServiceCore

```
import {requestToServer, requestToServer2} from '../../utils/requestToServer'


export class MoviesServiceCore {
    private static readonly DEFAULT_SERVER_DOMAIN: string = 'http://127.0.0.1:8000/api/'

    private static readonly SINGLE_MOVIE_ROUTE: string = 'movie'
    private static readonly MOVIES_LIST_ROUTE: string = 'movies'
    private static readonly POPULAR_MOVIES_LIST_ROUTE: string = 'movies/popular'
    private static readonly MOVIE_THEATERS_LIST_ROUTE: string = 'movietheaters'
    private static readonly POPULAR_MOVIE_THEATERS_ROUTE: string = 'movietheaters/popular'
    private static readonly LIST_OF_MOVIES_BY_IDS_ROUTE: string = 'movies/getmovieslistbyids/?ids='
    private static readonly THEATERS_FOR_EXACT_MOVIE: string = 'theatersformovie/'
    private static readonly SEARCH_MOVIES_ROUTE: string = 'searchmovie/'
    private static readonly TOKEN_AUTHORIZATION_ROUTE: string = 'token/'
    private static readonly TOKEN_REFRESH_ROUTE: string = 'token/refresh/'
    private static readonly REGISTER_ROUTE: string = 'register/'
    private static readonly ADD_TO_FAVOURITES_ROUTE: string = 'addfavourite/'
    private static readonly GET_FAVOURITES_ROUTE: string = 'getfavourites/'
    private static readonly CHECK_IF_FAVOURITE_ROUTE: string = 'checkiffavourite/'
    private static readonly REMOVE_FROM_FAVOURITE_ROUTE: string = 'removefromfavourite/'

    private readonly serverDomain: string


    public constructor(serverDomain: string = MoviesServiceCore.DEFAULT_SERVER_DOMAIN) {
        this.serverDomain = serverDomain
    }

    public getMovieById(id: string, callback: any): void {
        const fullUrl: string = `${this.serverDomain}${MoviesServiceCore.SINGLE_MOVIE_ROUTE}/${id}/`
        requestToServer({url: fullUrl, method: 'GET', callback: callback})
    }

    public getPopularMovies(callback: any): void {
        const fullUrl: string = `${this.serverDomain}${MoviesServiceCore.POPULAR_MOVIES_LIST_ROUTE}/`
        requestToServer({url: fullUrl, method: 'GET', callback: callback})
    }

    public getPopularTheaters(callback: any): void {
        const fullUrl: string = `${this.serverDomain}${MoviesServiceCore.POPULAR_MOVIE_THEATERS_ROUTE}/`
        requestToServer({url: fullUrl, method: 'GET', callback: callback})
    }

    public getMovies(callback: any, from: number, to: number): void {
        const fullUrl: string = `${this.serverDomain}${MoviesServiceCore.MOVIES_LIST_ROUTE}/${from}/${to}/`
        requestToServer({url: fullUrl, method: 'GET', callback: callback})
    }

    public getMovieTheaters(callback: any): void {
        const fullUrl: string = `${this.serverDomain}${MoviesServiceCore.MOVIE_THEATERS_LIST_ROUTE}/`
        requestToServer({url: fullUrl, method: 'GET', callback: callback})
    }

    public getMovieTheaterByTitle(title: string, callback: any): void {
        const fullUrl: string = `${this.serverDomain}${MoviesServiceCore.MOVIE_THEATERS_LIST_ROUTE}/${title}/`
        requestToServer({url: fullUrl, method: 'GET', callback: callback})
    }

    public getMoviesForTheater(ids: Array<string>, callback: any): void {
        const ids_string: string = ids.join(',')
        const fullUrl: string = `${this.serverDomain}${MoviesServiceCore.LIST_OF_MOVIES_BY_IDS_ROUTE}${ids_string}`
        requestToServer({url: fullUrl, method: 'GET', callback: callback, body: ids})
    }

    public getTheatersForMovie(movieId: string, callback: any): void {
        const fullUrl: string = `${this.serverDomain}${MoviesServiceCore.THEATERS_FOR_EXACT_MOVIE}${movieId}/`
        requestToServer({url: fullUrl, method: 'GET', callback: callback})
    }

    public searchMovies(query: string, callback: any): void {
        const fullUrl: string = `${this.serverDomain}${MoviesServiceCore.SEARCH_MOVIES_ROUTE}${query}/`
        requestToServer({url: fullUrl, method: 'GET', callback: callback})
    }

    public authorize(username: string, password: string, callback: any): void {
```

```
        const fullUrl: string = `${this.serverDomain}${MoviesServiceCore.TOKEN_AUTHORIZATION_ROUTE}`
        requestToServer2({
            url: fullUrl,
            method: 'POST',
            callback: callback,
            headers: {'Content-Type': 'application/json'},
            body: JSON.stringify({username: username, password: password})
        })
    }

    public refreshToken(token: string, callback: any): void {
        const fullUrl: string = `${this.serverDomain}${MoviesServiceCore.TOKEN_REFRESH_ROUTE}`
        requestToServer2({
            url: fullUrl,
            method: 'POST',
            callback: callback,
            headers: {'Content-Type': 'application/json'},
            body: JSON.stringify({refresh: token})
        })
    }

    public register(formData: FormData, callback: any): void {
        const fullUrl: string = `${this.serverDomain}${MoviesServiceCore.REGISTER_ROUTE}`
        requestToServer2({
            url: fullUrl,
            method: 'POST',
            callback: callback,
            headers: {},
            body: formData
        })
    }

    public addToFavourites(movieId: string, userToken: string, callback: any): void {
        const fullUrl: string = `${this.serverDomain}${MoviesServiceCore.ADD_TO_FAVOURITES_ROUTE}${movieId}/`
        requestToServer2({
            url: fullUrl,
            method: 'PUT',
            callback: callback,
            headers: {
                'Authorization': `Bearer ${userToken}`
            },
            body: null
        })
    }

    public getFavourites(userToken: string, callback: any): void {
        const fullUrl: string = `${this.serverDomain}${MoviesServiceCore.GET_FAVOURITES_ROUTE}`
        requestToServer2({
            url: fullUrl,
            method: 'GET',
            callback: callback,
            headers: {'Authorization': `Bearer ${userToken}`},
            body: null
        })
    }

    public checkIfFavourite(movieId: string, userToken: string, callback: any): void {
        const fullUrl: string = `${this.serverDomain}${MoviesServiceCore.CHECK_IF_FAVOURITE_ROUTE}${movieId}/`
        requestToServer2({
            url: fullUrl,
            method: 'GET',
            callback: callback,
            headers: {'Authorization': `Bearer ${userToken}`},
            body: null
        })
    }

    public removeFromFavourites(movieId: string, userToken: string, callback: any): void {
        const fullUrl: string = `${this.serverDomain}${MoviesServiceCore.REMOVE_FROM_FAVOURITE_ROUTE}${movieId}/`
        requestToServer2({
            url: fullUrl,
            method: 'PUT',
            callback: callback,
            headers: {'Authorization': `Bearer ${userToken}`},
            body: null
        })
    }
}
```

Сервер

Модели:

```
from django.db import models
from django.contrib.auth.models import User
```

```python
class CinematicUniverse(models.Model):
    name = models.CharField(max_length=100)

    def __str__(self):
        return f'Cinematic Universe - {self.name}'


class Movie(models.Model):
    title = models.CharField(max_length=200)
    year = models.IntegerField()
    movie_id = models.CharField(max_length=50)
    date_from = models.DateField()
    date_to = models.DateField()
    rating = models.SmallIntegerField()
    poster_image_link = models.CharField(max_length=500)
    duration = models.SmallIntegerField()
    age_restriction = models.SmallIntegerField()
    visits_count = models.IntegerField(default=0)
    category = models.ForeignKey('CinematicUniverse', on_delete=models.PROTECT, null=True)

    def __str__(self):
        return f'Movie - {self.title}'


class MovieTheater(models.Model):
    title = models.CharField(max_length=200)
    address = models.CharField(max_length=300)
    location = models.CharField(max_length=600)
    photo = models.CharField(max_length=600, default='photo')
    telephone = models.CharField(max_length=80, default='+375')
    visits_count = models.IntegerField(default=0)
    movies = models.ManyToManyField(Movie)

    def __str__(self):
        return f'Movie theater - {self.title}'


class UsersFavourites(models.Model):
    user = models.ForeignKey(User, on_delete=models.CASCADE)
    favourites = models.ManyToManyField(Movie)
```

Представления:

```python
from django.contrib.auth.models import User
from rest_framework import views, generics, permissions
from rest_framework.response import Response
from rest_framework.request import Request

from rest_framework_simplejwt.views import TokenObtainPairView
from rest_framework.permissions import IsAuthenticated, AllowAny

from django.db.models import F

from .models import Movie, MovieTheater, UsersFavourites
from .serializers import MovieShortenSerializer, MovieFullSerializer, MovieTheaterSerializer, \
    MyTokenObtainPairSerializer, RegisterSerializer


class EmptyRouteAPIView(views.APIView):
    def get(self, request: Request) -> Response:
        return Response({'info': 'Welcome to root page'})


class MoviesAPIView(views.APIView):
    STATUS_SUCCESS_RETURNED: str = 'Successfully returned movies'
    STATUS_SUCCESS_BUT_RETURNED_LESS: str = 'Successfully returned, but only available part'
    STATUS_ERROR_INVALID_QUERY: str = 'Nothing to return. Invalid query'

    def get(self, request: Request, load_from: int = 0, load_to: int = 3) -> Response:
        response: dict = dict()

        all_movies = Movie.objects.all()

        real_load_to: int = min(len(all_movies), max(0, load_to + 1))

        if 0 <= load_from <= real_load_to <= len(all_movies):
            response['success'] = True
            response['data'] = MovieShortenSerializer(all_movies[load_from:real_load_to], many=True).data
            response['howManyLeft'] = len(all_movies) - real_load_to

            if real_load_to - 1 != load_to:
```

```
                response['status'] = MoviesAPIView.STATUS_SUCCESS_BUT_RETURNED_LESS
            else:
                response['status'] = MoviesAPIView.STATUS_SUCCESS_RETURNED
        else:
            response['success'] = False
            response['data'] = list()
            response['status'] = MoviesAPIView.STATUS_ERROR_INVALID_QUERY
            response['howManyLeft'] = 0

        return Response(response)


class MovieAPIView(views.APIView):
    def get(self, request: Request, searched_id: str) -> Response:
        response: dict = dict()

        found_in_database = Movie.objects.filter(movie_id=searched_id)

        if len(found_in_database) == 0:
            response['success'] = False
            response['data'] = None
        elif len(found_in_database) == 1:
            found_in_database.update(visits_count=F('visits_count') + 1)
            response['success'] = True
            response['data'] = MovieFullSerializer(found_in_database[0]).data

        return Response(response)


class PopularMoviesAPIView(views.APIView):
    DEFAULT_POPULAR_MOVIES_COUNT: int = 4

    def get(self, request: Request) -> Response:
        all_movies = Movie.objects.all()

        return_count: int = min(PopularMoviesAPIView.DEFAULT_POPULAR_MOVIES_COUNT, len(all_movies))

        popular_movies = all_movies.order_by('-visits_count')[:return_count]

        return Response({'data': MovieShortenSerializer(popular_movies, many=True).data})


class MovieTheatersAPIView(views.APIView):
    def get(self, request: Request) -> Response:
        return Response({'data': MovieTheaterSerializer(MovieTheater.objects.all(), many=True).data})


class MovieTheaterAPIView(views.APIView):
    def get(self, request: Request, searched_title: str) -> Response:
        response: dict = dict()

        found_in_database = MovieTheater.objects.filter(title=searched_title)

        if len(found_in_database) == 0:
            response['success'] = False
            response['data'] = None
        elif len(found_in_database) == 1:
            found_in_database.update(visits_count=F('visits_count') + 1)
            response['success'] = True
            response['data'] = MovieTheaterSerializer(found_in_database[0]).data

        return Response(response)


class MostPopularMovieTheaterAPIView(views.APIView):
    def get(self, request: Request) -> Response:
        return Response({
            'data': MovieTheaterSerializer(MovieTheater.objects.all().order_by('-visits_count')[:2], many=True).data
        })


class MoviesByIdsAPIView(views.APIView):
    def get(self, request: Request) -> Response:
        ids_list: list = list(request.query_params.get('ids').split(','))

        all_movies = Movie.objects

        response: list = list()

        for movie_db_id in ids_list:
            response.append(MovieShortenSerializer(all_movies.filter(id=movie_db_id)[0]).data)

        return Response({'data': response})


class TheatersForMovieAPIView(views.APIView):
    def get(self, request: Request, movie_to_search_theaters: str) -> Response:
        found_in_database = Movie.objects.filter(movie_id=movie_to_search_theaters)
```

```
        if len(found_in_database) == 0:
            return Response({'success': False, 'status': 'No such movie exists', 'data': []})

        field_id_for_movie = Movie._meta.get_field('id')
        field_movie_id_for_movie = Movie._meta.get_field('movie_id')

        according_theaters: list = list()

        field_movies_in_theater = MovieTheater._meta.get_field('movies')

        for theater in MovieTheater.objects.all():
            movies_in_this_theater = field_movies_in_theater.value_from_object(theater)

            for movie in movies_in_this_theater:
                if field_movie_id_for_movie.value_from_object(movie) == movie_to_search_theaters:
                    according_theaters.append(MovieTheaterSerializer(theater).data)
                    break

        return Response({'data': according_theaters})


class SearchMovieAPIView(views.APIView):
    def get(self, request: Request, query: str) -> Response:
        all_movies = Movie.objects.all()

        field_title_for_movie = Movie._meta.get_field('title')
        field_movieid_for_movie = Movie._meta.get_field('movie_id')
        field_year_for_movie = Movie._meta.get_field('year')

        found_data: list = list()

        for movie in all_movies:
            title: str = field_title_for_movie.value_from_object(movie).lower()
            movie_id: str = field_movieid_for_movie.value_from_object(movie).lower()
            year: str = str(field_year_for_movie.value_from_object(movie)).lower()

            if (query in title) or (query in movie_id) or (query in year):
                found_data.append(MovieShortenSerializer(movie).data)

        return Response({'data': found_data})


class MyTokenObtainPairView(TokenObtainPairView):
    serializer_class = MyTokenObtainPairSerializer


class RegisterView(generics.CreateAPIView):
    queryset = User.objects.all()
    permission_classes = (AllowAny,)
    serializer_class = RegisterSerializer


class AddFavouritesAPIView(views.APIView):
    permission_classes = (IsAuthenticated, permissions.IsAuthenticatedOrReadOnly)
    queryset = User.objects.all()

    def put(self, request: Request, what_to_add: str = '') -> Response:
        found_movies = Movie.objects.filter(movie_id=what_to_add)

        if len(found_movies) == 0:
            return Response({
                'success': False,
                'status': 'Unable to add not existing movie to favourites'
            })
        elif len(found_movies) == 1:
            this_user = User.objects.filter(username=request.user.username)[0]

            if len(UsersFavourites.objects.filter(user=request.user)) == 0:
                UsersFavourites.objects.create(user=this_user)

            searched_row = UsersFavourites.objects.filter(user=request.user)[0]

            if len(searched_row.favourites.filter(movie_id=what_to_add)) != 0:
                return Response({'success': False, 'status': 'Already in your favourites !'})

            searched_row.favourites.add(found_movies[0])

            return Response({'success': True, 'status': 'seemed to be added !'})

        return Response({
            'success': False,
            'status': 'Unable to add, because (I do not know, why, but) there are several movies with such id'
        })


class GetFavouritesAPIView(views.APIView):
    permission_classes = (IsAuthenticated, permissions.IsAuthenticatedOrReadOnly)
```

```
    queryset = User.objects.all()

    def get(self, request: Request) -> Response:
        this_user = User.objects.filter(username=request.user.username)[0]

        if len(UsersFavourites.objects.filter(user=request.user)) == 0:
            UsersFavourites.objects.create(user=this_user)

        searched_row = UsersFavourites.objects.filter(user=request.user)[0]

        return Response({'data': MovieShortenSerializer(searched_row.favourites, many=True).data})


class CheckIfInFavourites(views.APIView):
    permission_classes = (IsAuthenticated, permissions.IsAuthenticatedOrReadOnly)
    queryset = User.objects.all()

    def get(self, request: Request, what_to_check: str) -> Response:
        this_user = User.objects.filter(username=request.user.username)[0]

        if len(UsersFavourites.objects.filter(user=request.user)) == 0:
            UsersFavourites.objects.create(user=this_user)

        searched_row = UsersFavourites.objects.filter(user=request.user)[0]

        return Response({'result': len(searched_row.favourites.filter(movie_id=what_to_check)) != 0})


class RemoveFromFavourites(views.APIView):
    permission_classes = (IsAuthenticated, permissions.IsAuthenticatedOrReadOnly)
    queryset = User.objects.all()

    def put(self, request: Request, what_to_remove: str) -> Response:
        if len(UsersFavourites.objects.filter(user=request.user)) == 0:
            return Response({'status': 'Nothing to remove'})

        searched_row = UsersFavourites.objects.filter(user=request.user)[0]

        found_movies = Movie.objects.filter(movie_id=what_to_remove)

        current_favourites = searched_row.favourites.filter(movie_id=what_to_remove)

        if len(current_favourites) != 0:
            searched_row.favourites.remove(found_movies[0])
            return Response({'status': 'successfully removed from favourites'})
        else:
            return Response({'status': 'This movie is not in favourites'})
```

URL:

```
from django.contrib import admin
from django.urls import path
from marveldcmovies.views import MoviesAPIView, MovieAPIView, SearchMovieAPIView, MyTokenObtainPairView, RegisterView
from marveldcmovies.views import PopularMoviesAPIView, MoviesByIdsAPIView, TheatersForMovieAPIView
from marveldcmovies.views import MovieTheatersAPIView, MovieTheaterAPIView, MostPopularMovieTheaterAPIView
from marveldcmovies.views import AddFavouritesAPIView, GetFavouritesAPIView, CheckIfInFavourites, RemoveFromFavourites
from rest_framework_simplejwt.views import TokenRefreshView
from marveldcmovies.views import EmptyRouteAPIView

urlpatterns = [
    path('', EmptyRouteAPIView.as_view()),

    path('admin/', admin.site.urls),

    path('api/movies/', MoviesAPIView.as_view()),
    path('api/movies/<int:load_from>/<int:load_to>/', MoviesAPIView.as_view()),
    path('api/movies/getmovieslistbyids/', MoviesByIdsAPIView.as_view()),
    path('api/popularmovies/', MoviesAPIView.as_view()),
    path('api/movie/<str:searched_id>/', MovieAPIView.as_view()),
    path('api/movies/popular/', PopularMoviesAPIView.as_view()),
    path('api/movietheaters/', MovieTheatersAPIView.as_view()),
    path('api/movietheaters/popular/', MostPopularMovieTheaterAPIView.as_view()),
    path('api/movietheaters/<str:searched_title>/', MovieTheaterAPIView.as_view()),
    path('api/theatersformovie/<str:movie_to_search_theaters>/', TheatersForMovieAPIView.as_view()),
    path('api/searchmovie/<str:query>/', SearchMovieAPIView.as_view()),

    path('api/token/', MyTokenObtainPairView.as_view(), name='token_obtain_pair'),
    path('api/token/refresh/', TokenRefreshView.as_view(), name='token_refresh'),
    path('api/register/', RegisterView.as_view(), name='auth_register'),

    path('api/addfavourite/<str:what_to_add>/', AddFavouritesAPIView.as_view()),
    path('api/getfavourites/', GetFavouritesAPIView.as_view()),
    path('api/checkiffavourite/<str:what_to_check>/', CheckIfInFavourites.as_view()),
    path('api/removefromfavourite/<str:what_to_remove>/', RemoveFromFavourites.as_view()),

]
```

Сериализаторы:

```python
import re

from django.contrib.auth.models import User
from django.contrib.auth.password_validation import validate_password
from rest_framework import serializers
# from .models import CustomUser as User
from rest_framework.validators import UniqueValidator
from rest_framework_simplejwt.serializers import TokenObtainPairSerializer

from marveldcmovies.models import MovieTheater, Movie


class MovieShortenSerializer(serializers.ModelSerializer):
    class Meta:
        model = Movie
        fields = ['title', 'movie_id', 'date_from', 'date_to', 'poster_image_link']


class MovieFullSerializer(serializers.ModelSerializer):
    class Meta:
        model = Movie
        exclude = ['id', 'visits_count']


class MovieTheaterSerializer(serializers.ModelSerializer):
    class Meta:
        model = MovieTheater
        exclude = ['visits_count', 'id']

# Useful post about authorization
# https://habr.com/ru/post/512746/


class MyTokenObtainPairSerializer(TokenObtainPairSerializer):
    @classmethod
    def get_token(cls, user):
        token = super().get_token(user)

        token['username'] = user.username

        return token


class RegisterSerializer(serializers.ModelSerializer):
    PASSWORD_REGEX: str = r'[A-Za-z0-9]{4,}'
    EMAIL_REGEX: str = r'^\w+([\.-]?\w+)*@\w+([\.-]?\w+)*(\.\w{2,4})+$'

    email = serializers.EmailField(
        required=True,
        validators=[UniqueValidator(queryset=User.objects.all())]
    )

    password = serializers.CharField(write_only=True, required=True, validators=[validate_password])

    class Meta:
        model = User
        fields = ('username', 'password', 'email')

    def validate(self, attrs):
        if not re.compile(RegisterSerializer.PASSWORD_REGEX).match(attrs['password']):
            raise serializers.ValidationError({'password': 'Wrong password (must contain 4+ symbols with letters or digits'})

        if not re.compile(RegisterSerializer.EMAIL_REGEX).match(attrs['email']):
            raise serializers.ValidationError({'email': 'Wrong email'})

        return attrs

    def create(self, validated_data):
        user = User.objects.create(
            username=validated_data['username'],
            email=validated_data['email'],
        )

        user.set_password(validated_data['password'])
        user.save()

        return user
```