

MStore

Szymon Stępień
Bartłomiej Żamojtel
Kamil Zubko

0. Uruchomienie

a) serwer

- `pip install -r requirements.txt`
- `python manage.py runserver`

b) Połączenie z bazą danych zdalną, kopia lokalna znajduje się w folderze z projektem:

`mongodb+srv://dev_usr:vmFG9idG2kCKnt87@cluster0.ppmeu.mongodb.net/projdb2021?retryWrites=true&w=majority`

zrzut w folderze bazy2021/baza_zrzut

1. Model danych

a) Movie

Kod znajduje się w folderze `api/models`

Sprzedawanym produktem będą filmy, każdy film może mieć wiele kategorii, kategorie muszą być wcześniej osobno zdefiniowane przez pracowników, w każdym filmie może grać wielu aktorów, do których referencje przechowujemy w zagnieżdżonej tablicy dokumentu `Movie`. Założono indexy na najczęściej filtrowane pola.

```
class Movie(models.Model):
    _id = models.ObjectIdField()
    title = models.CharField(max_length=200, unique=True, db_index=True)
    price = models.FloatField(db_index=True)
    categories = ArrayField(
        model_container=_Category,
        db_index=True
    )
    units_available = IntegerField(default=0)
    description = models.CharField(max_length=512)
    length = models.IntegerField() # in seconds
    main_actors = models.ArrayReferenceField(
        to=Actor
    )
    production_date = models.DateField(db_index=True)
    director = models.EmbeddedField(
        model_container=Director
    )
    user_ratings_count = models.IntegerField(default=0)
    total_rates = models.IntegerField(default=0)
    image_src = models.CharField(max_length=256)
```

w efekcie dostajemy dokument MongoDB:

```
Movie: {
  _id: ObjectId
  title: string,
  main_actors: [ObjectId],
  photos: [string],
  productionDate: Date,
  director: {
    firstName: string,
    lastName: string,
  },
  units_available: int,
  price: float,
  description: string,
  user_ratings_count: int,
  total_rates: float,
  length: int
}
```

```
}
```

b) Actor

Przechowywujemy informacje o aktorach:

```
class Actor(models.Model):
    _id = models.ObjectIdField()
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=30)
    photo = models.URLField()
    birthdate = models.DateField()
```

```
Actor: {
  _id: ObjectId,
  firstName: string,
  lastName: string,
  photo: string, //url
  birthDate: Date
}
```

c) Category

Przechowujemy kolekcję zawierającą wszystkie dostępne kategorie

```
class Category(models.Model):
    _id = models.ObjectIdField()
    name = models.CharField(max_length=100)
```

```
Category {
  _id: ObjectId,
  name: string
}
```

d) User

Użytkownik posiada unikatową nazwę oraz email, logowanie odbywa się z wykorzystaniem samego maila.

Zagnieżdżamy obiekt Roles, poza zwykłym domyślnym użytkownikiem dopuszczamy rolę admina oraz pracownika.

Użytkownik ma opcję dodawania produktów do koszyka, koszyk jest modelowany jako tablica, której każdy element zawiera liczbę dodanych sztuk produktu oraz referencję do właściwego produktu, szczegółowe dane nie są zagnieżdżane z uwagi na koszt operacji zmiany danych produktu.

```
class User(AbstractBaseUser, PermissionsMixin):
    _id = models.ObjectIdField()
    username = models.CharField(max_length=255, unique=True, db_index=True)
    email = models.EmailField(max_length=255, unique=True, db_index=True)
    roles = models.EmbeddedField(
        model_container=Roles)
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)
    cart = models.ArrayField(
        model_container=Cart
    )
```

W efekcie dostajemy dokument:

```
User: {
  email: string,
  username: string,
  password: string, //hashowane
  roles: {
    admin: boolean,
    employee: boolean
  },
  createdAt: Date,
  updatedAt: Date,
  cart: [{
    movie_id: ObjectID,
    amount: int
  }]
}
```

e) History

```
class OrderDetails(models.Model):
    price = models.FloatField()
    units_ordered = models.IntegerField()
    # duplicate title info so join wont have to be performed each time
    title = models.CharField(max_length=255)
    movie_id = models.ObjectIdField()

    class Meta:
        abstract = True

class History(models.Model):
    _id = models.ObjectIdField()
    order_date = models.DateField(auto_now_add=True, blank=True)
    user_id = models.CharField(max_length=24, db_index=True)
    details = models.ArrayField(
        model_container=OrderDetails
    )
```

Użytkownik ma opcję kupna produktu, dane takiego zakupu przechowujemy w dokumencie History, zakup może mieć wiele pozycji, które przechowujemy w tablicy obiektów OrderDetails, dane wspólne takie jak identyfikator użytkownika w celu uniknięcia redundancji przechowujemy na poziomie History.

Dokument OrderDetails zawiera referencję do kupionego produktu, aby można było znaleźć jego szczegóły oraz powielenie takich informacji jak cena (ponieważ może się zmienić a chcemy pamiętać tę w dniu zakupu) oraz tytuł i liczbę zamówionych sztuk, gdyż są to główne informacje, o które będziemy wysyłać zapytania z poziomu klienta, dzięki powieleniu wykonanie JOIN'a nie będzie konieczne.

Dostajemy dokument:

```
History: {
  orderDate: Date,
  userId: ObjectId,
  details: [{
    unitsOrdered: int,
    price: float,
    title: string,
    movieId: ObjectId
  }]
}
```

2. Routing

Użytkownik łączący się z serwerem wykorzystuje nazwę strony. Serwer renderuje szablon strony wygenerowany przez Angulara i dalszy routing z poziomu klienta zostaje przejęty przez router Angulara.

Wszelkie zapytania dotyczące danych wysyłane są pod adres /api. Adresy zawierające /api/manage nie są dostępne dla zwykłych użytkowników. Dostępne endpointy:

- **POST auth/register** - rejestruje nowego użytkownika
- **POST auth/login** - loguje użytkownika

- **GET movies** - zwraca listę dostępnych filmów wykorzystuje query parametry:
 - a) offset - od którego filmu w posortowanej kolekcji zwracać wyniki
 - b) amount - ile filmów zwrócić
 - c) price_from - cena minimalna
 - d) price_to - cena maksymalna
 - e) category - kategoria filmu

- **GET movies/{movie_id}** - zwraca szczegóły danego filmu

- **POST manage/movies** - endpoint dostępny dla pracownika/admina POST dodaje nowy film
- **PUT manage/movies/{movie_id}** -dostępne dla pracownika/admina modyfikuje dane filmu
- **DELETE manage/movies/{movie_id}** -dostępne dla pracownika/admina usuwa film

- **GET cart** - zwraca koszyk aktualnie zalogowanego użytkownika
- **POST cart** - dokonuje zakupu całej zawartości koszyka użytkownika
- **POST cart/{movie_id}** - dodaje nowy produkt do koszyka, lub zwiększa liczbę jego sztuk w koszyku

- **GET history** - zwraca całą historię zakupów zalogowanego użytkownika

- **GET news** - zwraca listę wiadomości o filmach

- **GET categories** - zwraca listę wszystkich dostępnych kategorii filmów
- **POST manage/categories** - dodaje nową kategorię
- **DELETE manage/categories?name={category_name}** - usuwa kategorię

- **GET actors/{actor_id}** - zwraca dane aktora

3. Autentykacja

Autentykacja została oparta na JWT. Zalogowany użytkownik wysyła w nagłówku zapytania:

Authorization: token <JWT>

Token jest przechowywany w local storage przeglądarki i z wykorzystaniem mechanizmu interceptora angulara automatycznie dodawany do zapytania.

Odpowienie endpointy są chronione oznaczeniem:

```
permission_classes = (IsAuthenticated,)
```

Jeśli autentykacja nie przejdzie serwer zwraca w odpowiedzi kod 403 Forbidden.

Ponadto endpointy wymagające posiadania odpowiedniej roli w systemie (np. admin) są również chronione z poziomu serwera i klienta.

4. Implementacja endpointów (kod znajduje się w folderze api/views i jwtauth/views)

- GET /api/news

```
def get(self, request):  
    news = News.objects.all().order_by('-timestamp')  
    news_list = []  
    counter = 0  
    for news_item in news:  
        if counter < 5:  
            news_list.append(news_item)  
            counter += 1  
  
    serializer = NewsSerializer(news_list, many=True)  
    return Response(serializer.data, status=status.HTTP_200_OK)
```

Zwraca 5 najnowszych wiadomości.

- GET /api/movies

```
def get(self, request):  
    return filter_movie_data(request, False)  
  
def filter_movie_data(request, authorized):  
    units = request.query_params.get('units_from')  
    if authorized and units is not None:  
        movies = Movie.objects.filter(units_available__gte=int(units))  
    else:
```

```

movies = Movie.objects.filter(units_available__gt=0)

try:
    offset = int(request.query_params.get('offset'))
    amount = int(request.query_params.get('amount'))
except TypeError:
    offset, amount = 0, 10

filter_options = ['category', 'price_from', 'price_to']
transforms = [lambda x: x.split(','), float, float]
filter_vals = [request.query_params.get(key) for key in filter_options]

try:
    filter_params = {key: tf(val) for key, tf, val in zip(
        filter_options, transforms, filter_vals) if val is not None}
except TypeError as e:
    print(e)
    return Response({'error': f'incorrectly formatted query params'},
        status=status.HTTP_400_BAD_REQUEST)

if 'price_from' in filter_params:
    movies = movies.filter(price__gte=filter_params['price_from'])
if 'price_to' in filter_params:
    movies = movies.filter(price__lte=filter_params['price_to'])
if 'category' in filter_params:
    movies = movies.filter(
        categories__in=[{'name': cat} for cat in filter_params['category']])

movies = movies.order_by('_id')[offset: offset + amount]
serializer = MovieSerializer(movies, many=True)
return Response(serializer.data, status=status.HTTP_200_OK)

```

Zwraca dostępne filmy uwzględniając filtrowanie.

- GET api/manage/movies

```

def get(self, request):
    if not (request.user.roles['employee'] or request.user.roles['admin']):
        return Response('Employee or admin permissions are required.',
            status=status.HTTP_403_FORBIDDEN)

    return filter_movie_data(request, False)

```

Zwraca dostępne filmy uwzględniając filtrowanie z danymi dostępnymi dla pracownika.

- POST api/manage/movies

```

def post(self, request):
    if not (request.user.roles['employee'] or request.user.roles['admin']):
        return Response('Employee or admin permissions are required.',
            status=status.HTTP_403_FORBIDDEN)

    data = request.data
    categories = [{"name": name}
        for name in request.data['categories']]

    legal_categories = Category.objects.filter(
        name__in=data['categories'])

    if legal_categories.count() < len(categories):
        for cat in categories:
            if legal_categories.filter(name=cat['name']).first() is None:
                return Response({'error': f'Category {cat["name"]} doesn\'t exist'},
                    status=status.HTTP_400_BAD_REQUEST)

    movie = Movie.objects.filter(title=data["title"]).first()
    if movie is not None:
        return Response({'error': f'movie {data["title"]} already exists'},
            status=status.HTTP_400_BAD_REQUEST)

    units = data.get('units_available')
    movie = Movie(title=data['title'], price=data['price'], categories=categories,
        units_available=units if units is not None else 0,
        description=data['description'], length=data['length'],
        image_src=data['image_src'])
    if movie.price < 0 or movie.units_available < 0 or movie.length < 0:
        return Response({'error': 'Invalid movie params'},
            status=status.HTTP_400_BAD_REQUEST)

    try:
        movie.save()
        return Response(**MovieSerializer(movie).data, "categories":
            movie.categories, "director": movie.director},
            status=status.HTTP_200_OK)
    except:
        return
        Response(status=status.HTTP_500_INTERNAL_SERVER_ERROR)

```

Tworzy nowy film w bazie danych.

- GET api/movies/{movie_id}

```

def get(self, request, id):

```

```
movie = get_movie(id)
```

```
if movie is None:
```

```
    return Response(status=status.HTTP_404_NOT_FOUND)
```

```
    return Response(**MovieSerializer(movie).data, "categories":  
movie.categories, "director": movie.director,  
                    status=status.HTTP_200_OK)
```

Zwraca szczegółowe dane filmu o danym id.

- PUT api/manage/movies/{movie_id}

```
def put(self, request, id):
```

```
    if not (request.user.roles['employee'] or request.user.roles['admin']):  
        return Response('Employee or admin permissions are required.',
```

```
        status=status.HTTP_403_FORBIDDEN)
```

```
    movie = get_movie(id)
```

```
    if movie is None:
```

```
        return Response(status=status.HTTP_404_NOT_FOUND)
```

```
    data = request.data
```

```
    movie.title = data["title"]
```

```
    movie.price = data["price"]
```

```
    movie.categories = data['categories']
```

```
    movie.units_available = data['units_available']
```

```
    movie.description = data['description']
```

```
    movie.length = data['length']
```

```
    movie.main_actors = data['main_actors']
```

```
    movie.production_date = data['production_date']
```

```
    movie.director = data['director']
```

```
    movie.user_ratings_count = data['user_ratings_count']
```

```
    movie.total_rates = data['total_rates']
```

```
    movie.image_src = data['image_src']
```

```
    try:
```

```
        movie.save()
```

```
        return Response(**MovieSerializer(movie).data, "categories":
```

```
movie.categories, "director": movie.director),
```

```
        status=status.HTTP_200_OK)
```

```
    except:
```

```
        return
```

```
Response(status=status.HTTP_500_INTERNAL_SERVER_ERROR)
```

Aktualizuje dane filmu o podanym id. Endpoint dostępny dla pracownika lub admina.

- DELETE api/manage/movies/{movie_id}

```
def delete(self, request, id):  
    if not (request.user.roles['employee'] or request.user.roles['admin']):  
        return Response('Employee or admin permissions are required.',  
            status=status.HTTP_403_FORBIDDEN)  
  
    movie = get_movie(id)  
    if movie is None:  
        return Response({'error': 'movie does not exist'},  
            status=status.HTTP_404_NOT_FOUND)  
  
    movie.delete()  
    return Response({'message': 'Movie was deleted successfully.'},  
        status=status.HTTP_204_NO_CONTENT)
```

Usuwa film o podanym id. Endpoint dostępny dla pracownika lub admina.

- GET api/cart

```
def get(self, request):  
    user = request.user  
    resp = [*user.cart]  
    for res in resp:  
        movie = Movie.objects.filter(_id=res['movie_id']).first()  
        res['movie_id'] = str(res['movie_id'])  
        res['title'] = movie.title  
        res['price'] = movie.price  
  
    return Response(resp, status=status.HTTP_200_OK)
```

Zwracamy całą zawartość koszyka zalogowanego użytkownika, endpoint dostępny tylko dla zalogowanych użytkowników.

- POST api/cart

```
def post(self, request):  
    user = request.user  
    try:  
        with transaction.atomic():  
            cart = user.cart  
            if len(cart) == 0:
```

```

        return Response({'msg': 'Your cart is empty'},
                        status=status.HTTP_400_BAD_REQUEST)

    order_details = []
    movies = []
    for item in cart:
        movie = Movie.objects.filter(_id=item['movie_id']).first()
        movies.append(movie)
        amount = item['amount']

        # not sure if this will be safe given multiple requests at single time
        if amount > movie.units_available:
            return Response({'limit_exceeded': True,
                            'movie_id': str(movie._id),
                            'units_available': movie.units_available},
                            status=status.HTTP_400_BAD_REQUEST)

        detail = {
            'price': movie.price,
            'units_ordered': amount,
            'movie_id': str(movie._id),
            'title': movie.title
        }

        order_details.append(detail)

    for movie, item in zip(movies, cart):
        movie.units_available -= item['amount']
        movie.save()

    user.cart = []
    history = History(user_id=str(user._id), details=order_details)

    user.save()
    history.save()
except:
    return Response(status=status.HTTP_500_INTERNAL_SERVER_ERROR)

return Response({'order': 'send summary here'}, status=status.HTTP_200_OK)

```

Wykorzystując transakcje, sprawdzamy czy koszyk użytkownika nie jest pusty, znajdujemy dane każdego filmu z koszyka, sprawdzamy czy liczba kupionych sztuk nie jest większa od liczby dostępnych.

Tworzymy obiekty zawierające szczegóły zakupu każdego filmu, oraz aktualizujemy liczbę dostępnych sztuk. Czyścimy koszyk użytkownika i tworzymy nowy dokument History zawierający wszystkie dane zakupu.

Endpoint dostępny tylko dla zalogowanych użytkowników.

- GET api/cart/{movie_id}

```
def post(self, request, movie_id):  
    data = request.data  
    amount = data['amount']  
    movie = get_movie(movie_id)  
    if movie is None:  
        return Response({"error": f"Movie ${movie_id} doesn't exist"},  
            status=status.HTTP_400_BAD_REQUEST)  
  
    user = request.user  
    found = False  
  
    for item in user.cart:  
        if item['movie_id'] == movie._id:  
            item['amount'] += amount  
            found = True  
            break  
  
    if not found:  
        user.cart.append({  
            "movie_id": movie._id,  
            "amount": amount  
        })  
  
    try:  
        user.save()  
        return Response(status=status.HTTP_200_OK)  
    except:  
        return Response({"error": "Failed to update users cart"},  
            status=status.HTTP_500_INTERNAL_SERVER_ERROR)
```

Sprawdzamy, czy dany film istnieje oraz czy liczba dostępnych sztuk jest większa od liczby zamówionych.

Jeśli użytkownik posiadał dany film w koszyku, aktualizowana jest liczba sztuk, w przeciwnym wypadku tworzona jest nowa pozycja w koszyku.

Endpoint dostępny tylko dla zalogowanych użytkowników.

- GET api/history

```
def get(self, request):  
    user = request.user  
    history = History.objects.filter(user_id=str(user._id))
```

```

resp = []

for item in history:
    for detail in item.details:
        detail['movie_id'] = str(detail['movie_id'])
        detail['order_date'] = item.order_date
        resp.append(detail)

return Response(resp, status=status.HTTP_200_OK)

```

Zwracamy całą historię zakupów użytkownika, dla każdego zakupu zwracamy ogólne informacje takie jak id kupionego filmu czy data zamówienia.

Endpoint dostępny tylko dla zalogowanych użytkowników.

- GET api/actors/{actor_id}

```

def get(self, request, id):
    actor = get_actor(id)

    if actor is None:
        return Response(status=status.HTTP_404_NOT_FOUND)

    return Response(ActorSerializer(actor).data, status=status.HTTP_200_OK)

```

Zwraca aktora o podanym id.

- PUT api/actors/{actor_id}

```

def put(self, request, id):
    actor = get_actor(id)

    if actor is None:
        return Response(status=status.HTTP_404_NOT_FOUND)

    data = request.data
    actor.first_name = data["first_name"]
    actor.last_name = data["last_name"]
    actor.photo = data["photo"]
    actor.birthdate = data["birthdate"]
    try:
        actor.save()
        return Response(ActorSerializer(actor).data,
status=status.HTTP_200_OK)
    except:
        return
Response(status=status.HTTP_500_INTERNAL_SERVER_ERROR)

```

Aktualizuje dane aktora o podanym id.

- DELETE api/actors/{actor_id}

```
def delete(self, request, id):  
    if not (request.user.roles['employee'] or request.user.roles['admin']):  
        return Response('Employee or admin permissions are required.',  
            status=status.HTTP_403_FORBIDDEN)  
  
    actor = get_actor(id)  
    if actor is None:  
        return Response({'error': 'actor does not exist'},  
            status=status.HTTP_404_NOT_FOUND)  
  
    actor.delete()  
    return Response({'message': 'Actor was deleted successfully.'},  
        status=status.HTTP_204_NO_CONTENT)
```

Usuwa aktora o podanym id.

- GET api/categories

```
@api_view(['GET'])  
def get_categories(request):  
    categories = Category.objects.all()  
    return Response({'categories': [c.name for c in categories]},  
        status=status.HTTP_200_OK)
```

Zwraca wszystkie kategorie filmów.

- POST / DELETE api/manage/categories

```
@api_view(['POST', 'DELETE'])  
@permission_classes((IsAuthenticated,))  
def update_category(request):  
    user = request.user  
    if not (request.user.roles['employee'] or request.user.roles['admin']):  
        return Response('Employee or admin permissions are required.',  
            status=status.HTTP_403_FORBIDDEN)  
  
    if request.method == 'POST':  
        name = request.data['name']  
        if Category.objects.filter(name=name).first() is not None:  
            return Response({'error': f'Category {name} already exists'},  
                status.HTTP_400_BAD_REQUEST)
```

```

category = Category(name=name)
try:
    category.save()
    return Response({'name': name}, status=status.HTTP_201_CREATED)
except:
    return Response({'error': f'Failed to create category {name}'},
                    status=status.HTTP_500_INTERNAL_SERVER_ERROR)
elif request.method == 'DELETE':
    name = request.query_params.get('name')
    if name is None:
        return Response(status=status.HTTP_400_BAD_REQUEST)

    category = Category.objects.filter(name=name).first()
    if category is None:
        return Response({'error': f'Category {name} doesn\'t exist'},
                        status.HTTP_400_BAD_REQUEST)

    if Movie.objects.filter(categories={'name': name}).first() is not None:
        return Response({'error': 'Can\'t delete category if movie with it exists'},
                        status=status.HTTP_400_BAD_REQUEST)

    try:
        category.delete()
        return Response({'name': category.name}, status=status.HTTP_200_OK)
    except Exception as e:
        print(e)
        return Response({'error': f'Failed to delete category {name}'},
                        status=status.HTTP_500_INTERNAL_SERVER_ERROR)

```

POST tworzy nową kategorię filmu. DELETE usuwa istniejącą.
Endpoint dostępny tylko dla pracownika lub admina.

5. Frontend

Filtrowanie:

Min price:

Max price:

Wybierz kategorie

sad

animated

funny

Latest movies

Min price:40

Max price:50


Wybierz kategorie

sad

animated

funny

PODZIEMNY KRĄG




Dwóch mężczyzn znudzonych rutyną zakłada klub, w którym co tydzień odbywają się walki na gołe pięści.

Price: \$40.00

Remaining tickets: 4

Read more

SNATCH



description

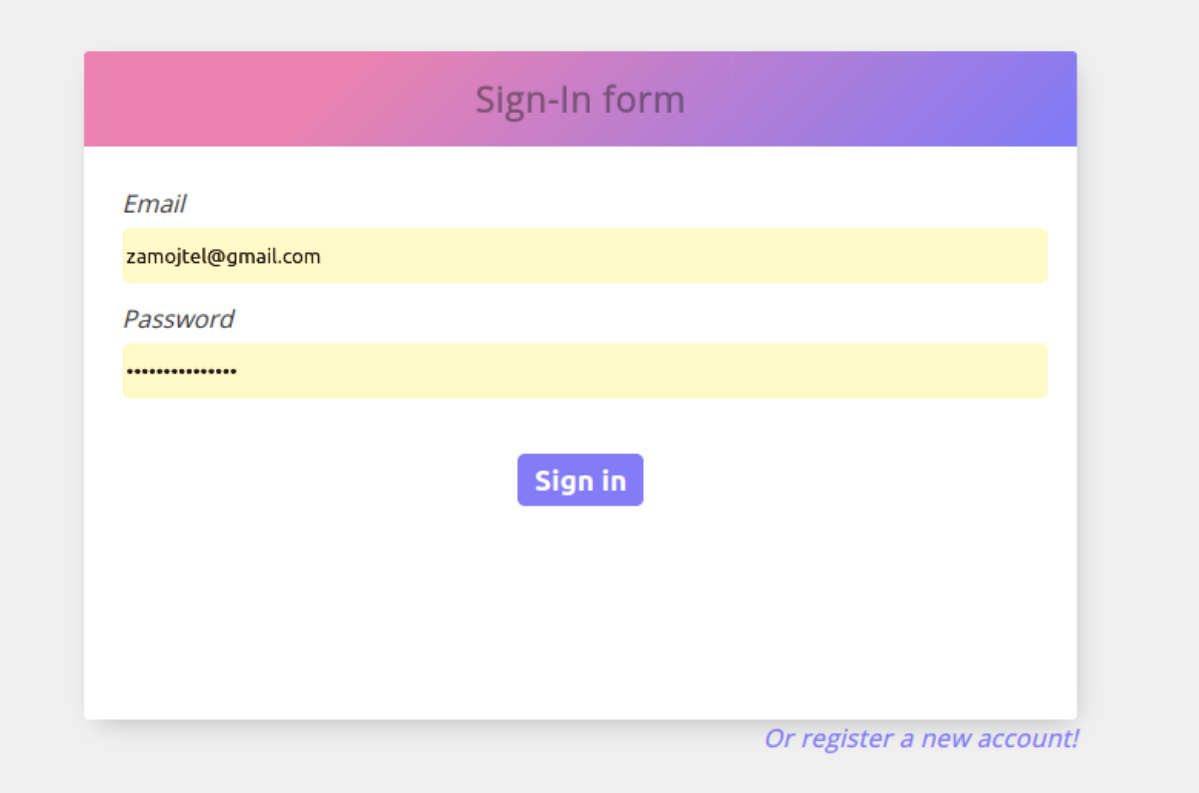
Price: \$50.00

Remaining tickets: 1

Read more

Load more

Formularz Logowania:



The image shows a modern, clean sign-in form. It features a white card with a subtle drop shadow on a light gray background. The card has a header bar with a pink-to-purple gradient containing the text "Sign-In form". Below the header, there are two input fields: "Email" with the value "zamojtel@gmail.com" and "Password" with masked characters ".....". A blue "Sign in" button is centered below the fields. At the bottom right of the card, there is a link that says "Or register a new account!".

Sign-In form

Email

zamojtel@gmail.com

Password

.....

Sign in

Or register a new account!

Formularz Rejestrowania:

Register form

Email

zamojtel@gmail.com

Username

Password

.....

Register

Pusty Koszyk:

CART					
Film name			Price	Amount	
			Total cost: \$ 0		
			Complete order		

Koszyk z Zawartością:

CART

Film name	Price	Amount
Incepcja	\$ 30	1
Mroczny Rycerz	\$ 25	1

Total cost: \$ 55

Complete order

Historia:

HISTORY

Film name	Ordered time	Amount	Price
DJANGO	2021-06-09	2	\$ 55
Mroczny Rycerz	2021-06-09	1	\$ 25
Podziemny Krag	2021-06-09	1	\$ 40
Incepcja	2021-06-09	1	\$ 30
Podziemny Krag	2021-06-10	1	\$ 40

HISTORY

Film name	Ordered time	Amount	Price
DJANGO	2021-06-09	2	\$ 55
Mroczny Rycerz	2021-06-09	1	\$ 25
Podziemny Krag	2021-06-09	1	\$ 40
Incepcja	2021-06-09	1	\$ 30
Podziemny Krag	2021-06-10	1	\$ 40
Incepcja	2021-06-10	1	\$ 30
Mroczny Rycerz	2021-06-10	1	\$ 25

Szczegóły Filmu:



Podziemny Krag *thriller, drama*

0 - 10
9
13 ocen

\$ 40
Add to cart
4 tickets left

Dwóch mężczyzn znudzonych rutyną zakłada klub, w którym co tydzień odbywają się walki na gołe pięści.

LENGTH	2:07:00
RATE	9
NUMBER OF RATINGS	13
DIRECTOR	David Fincher
CATEGORIES	
Thriller	
Drama	
ACTORS	