Bazy Danych

Miniprojekt

Błażej Nowicki, Mateusz Furga

10 maj, 2022

Opis projektu

Tematem projektu jest aplikacja webowa pozwalająca na umawianie się na wspólne uczenie oraz odrabianie zadań. Użytkownicy mogą dołączać do grup poprzez wybieranie przedmiotów, na które uczęszczają. Aplikacja umożliwia dodawanie wydarzeń (np. kolokwium Bazy Danych) oraz proponowanych terminów spotkań. Użytkownicy mogą oddać głos na terminy (propozycje) aby oznaczać że dany termin im pasuje.

Technologie

W miniprojekcie wykorzystaliśmy:

- Django 4.0.4
- PostgreSQL 14
- Bootstrap 5.1
- Docker

Jak uruchomić

Aplikację można uruchomić:

```
git clone https://github.com/BlazejNowicki/study-meetup
cd study-meetup
docker-compose run web python3 manage.py makemigrations catalog event
docker-compose run web python3 manage.py migrate
docker-compose run web python manage.py createsuperuser
docker-compose run web python ./add_mock_data.py
docker-compose up
```

Dane do logowania:

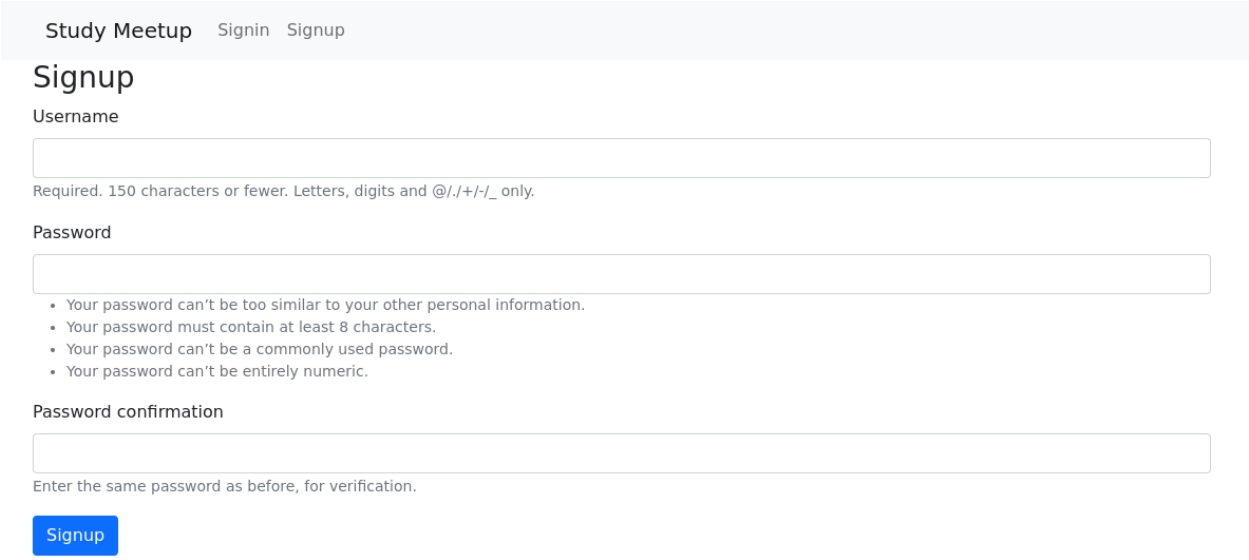
Użytkownik: jankowalski Hasło: jankowalski

Użytkownik: annanowak Hasło: annanowak

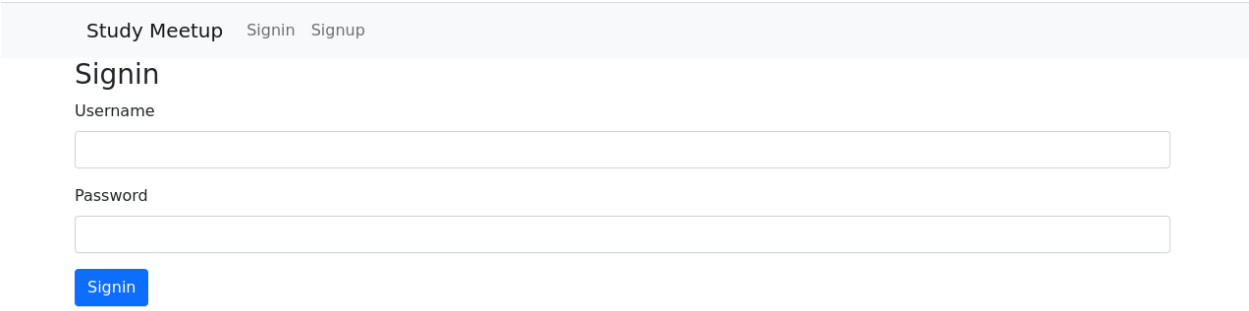
Funkcjonalności

Aby zacząć korzystać z aplikacji należy założyć konto oraz się zalogować.

Widok rejestracji:

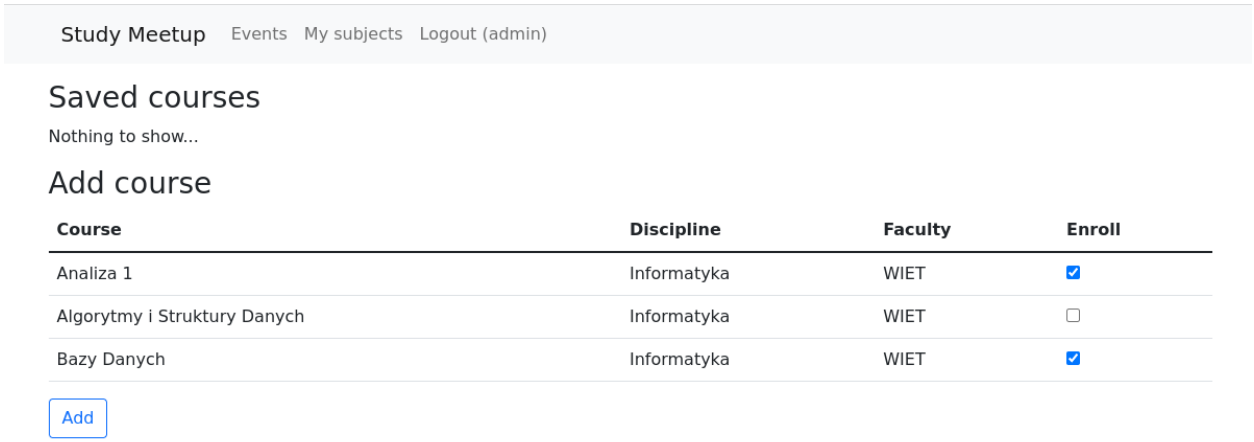


Widok logowania:

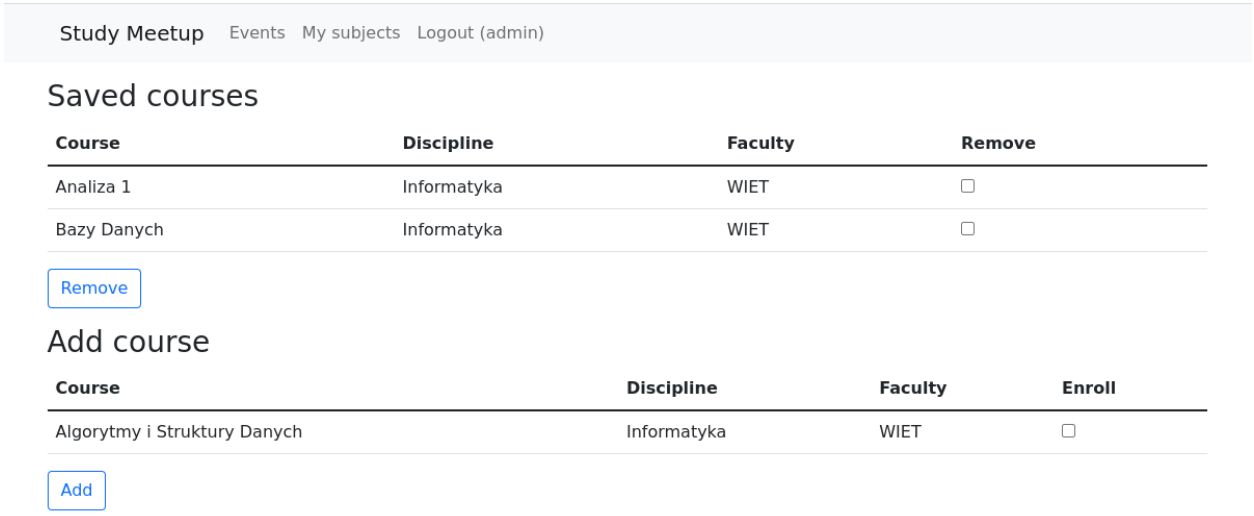


Po zalogowaniu możemy wybrać interesujące nas przedmioty oraz dodać je do listy zapisanych.

Lista przedmiotów do wybrania:



Dodane przedmioty:



Następnie przechodząc do zakładki Events możemy znaleźć wszystkie wydarzenia, które dotyczą zapisanych się przez nas przedmiotów. Możemy filtrować wydarzenia po nazwie oraz statusie.

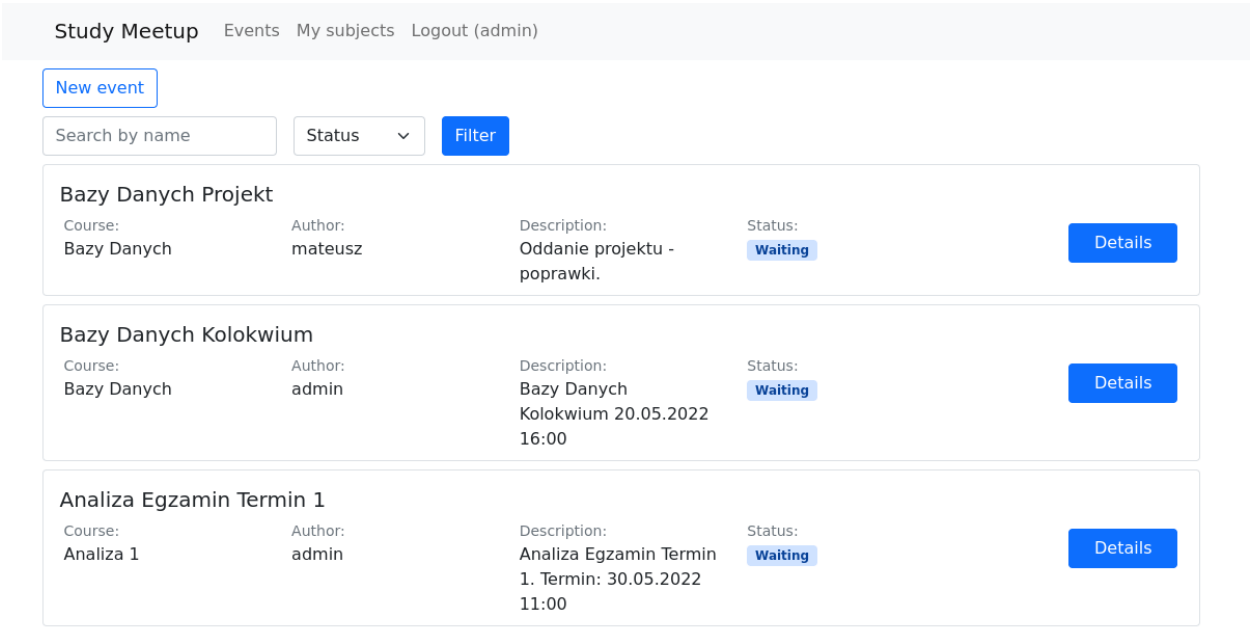
Każde wydarzenie posiada następujące atrybuty:

- name - unikalna nazwa wydarzenia
- description - opis wydarzenia
- author - autor (użytkownik), który dodał wydarzenie
- status - status wydarzenia (opisany niżej)

Wyróżniamy 3 statusy:

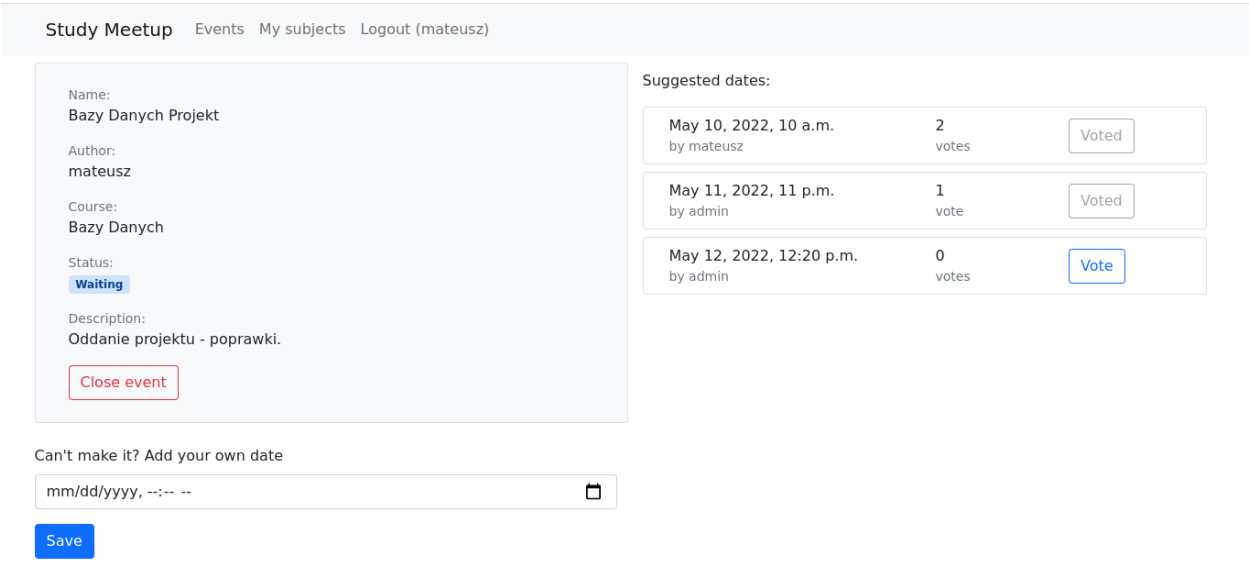
- waiting - wydarzenie nie posiada jeszcze wybranego terminu, możliwe jest głosowanie na propozycje terminów oraz dodawanie nowych
- approved - termin wydarzenia został już wybrany, dodawania oraz głosowanie na propozycje jest niemożliwe
- archived - wydarzenie zostało zarchiwizowane

Widok listy wydarzeń w zależności od wybranych przedmiotów:

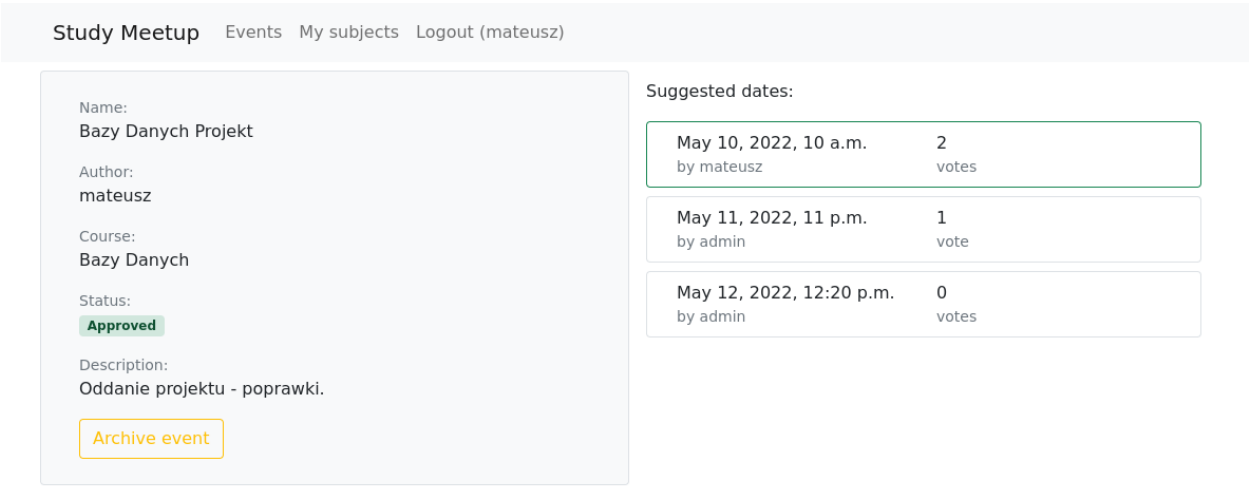


Wybierając interesujące nas wydarzenie oraz klikając w “Details” przechodzimy do widoku szczegółowego, w którym możemy znaleźć wszystkie informacje m.in. propozycje terminów oraz liczbę oddanych na nie głosów. Możemy również dodawać nowe jeśli tylko wydarzenie ma status waiting.

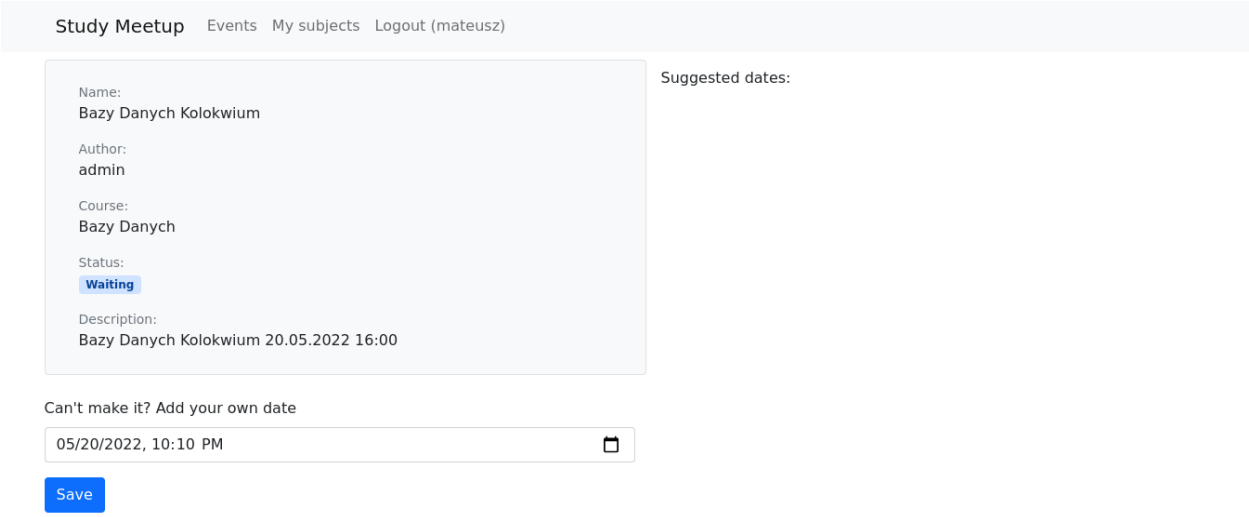
Widok szczegółowy wydarzenia:



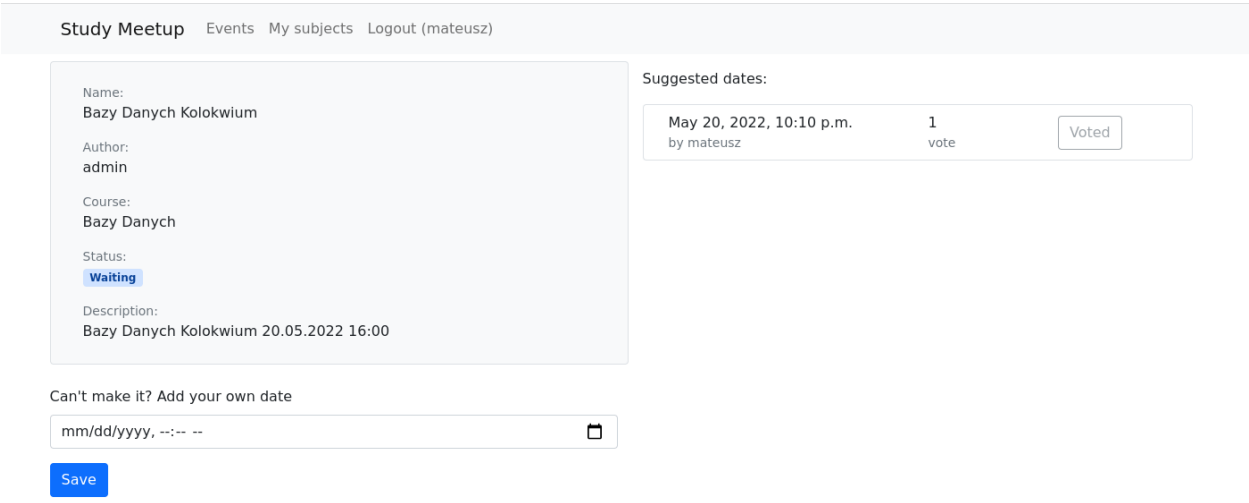
Jeśli jesteśmy autorami wydarzenia, tak jak w przykładzie powyżej, możemy je również zamknąć, tym samym zmienić status na approved klikając w przycisk “Close event”. Propozycja terminu z największą ilością oddanych głosów zostanie automatycznie wybrana jako ostateczny termin wydarzenia i jest ona oznacza zielonym obramowaniem.



Na wydarzenia ze statusem waiting możemy zgłaszać propozycje własnych terminów poprzez wybranie daty i godziny z kalendarza i kliknięcie przycisku “Save”.

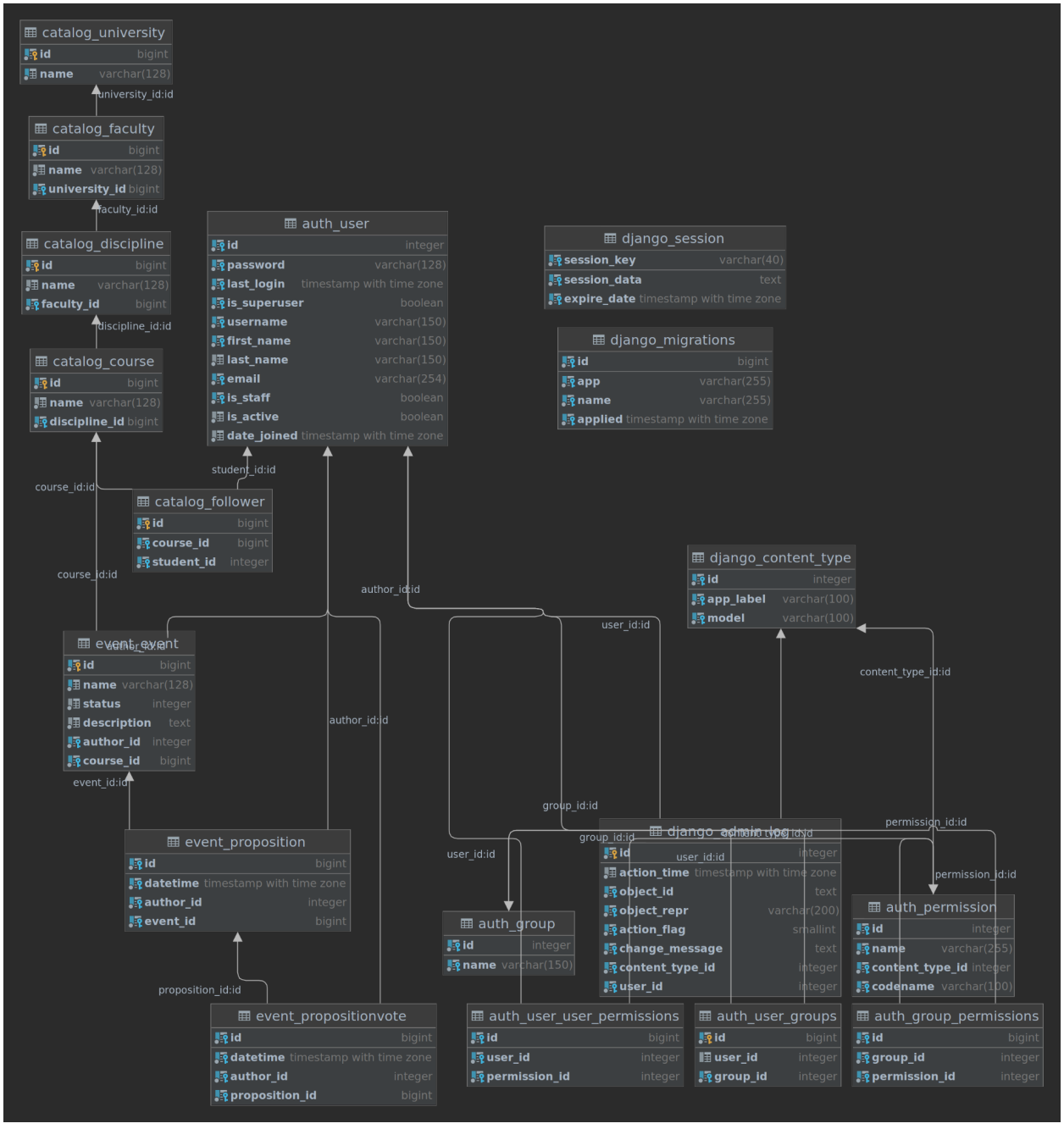


Po zapisaniu nowa propozycja pojawi się na liście dostępnych. Możemy również oddawać głosy na terminy, które nam pasują poprzez kliknięcie przycisku “Vote” przy odpowiedniej propozycji.



Aplikacja umożliwia oddanie tylko jednego głosu przez użytkownika na daną propozycję.

Schemat


catalog_university
id bigint
name varchar(128)
university_id:id
catalog_faculty
id bigint
name varchar(128)
university_id bigint
faculty_id:id
catalog_discipline
id bigint
name varchar(128)
faculty_id bigint
discipline_id:id
catalog_course
id bigint
name varchar(128)
discipline_id bigint
course_id:id
course_id:id
auth_user
id integer
password varchar(128)
last_login timestamp with time zone
is_superuser boolean
username varchar(150)
first_name varchar(150)
last_name varchar(150)
email varchar(254)
is_staff boolean
is_active boolean
date_joined timestamp with time zone
student_id:id
catalog_follower
id bigint
course_id bigint
student_id integer
django_session
session_key varchar(40)
session_data text
expire_date timestamp with time zone
django_migrations
id bigint
app varchar(255)
name varchar(255)
applied timestamp with time zone
author_id:id
user_id:id
django_content_type
id integer
app_label varchar(100)
model varchar(100)
content_type_id:id
event_event
id bigint
name varchar(128)
status integer
description text
author_id integer
course_id bigint
event_id:id
author_id:id
group_id:id
permission_id:id
event_proposition
id bigint
datetime timestamp with time zone
author_id integer
event_id bigint
user_id:id
group_id:id
django_admin_log
id integer
action_time timestamp with time zone
object_id text
object_repr varchar(200)
action_flag smallint
change_message text
content_type_id integer
user_id integer
permission_id:id
auth_group
id integer
name varchar(150)
auth_permission
id integer
name varchar(255)
content_type_id integer
codename varchar(100)
proposition_id:id
event_propositionvote
id bigint
datetime timestamp with time zone
author_id integer
proposition_id bigint
auth_user_user_permissions
id bigint
user_id integer
permission_id integer
auth_user_groups
id bigint
user_id integer
group_id integer
auth_group_permissions
id bigint
group_id integer
permission_id integer

Szczegóły

Projekt podzieliliśmy na 3 moduły:

- catalog - informacje na temat uniwersytetach, wydziałach, kierunkach, przedmiotach oraz zapisanych studentach na dany przedmiot
- event - zarządza listą eventów, propozycjami oraz liczbą oddanych na nie głosów
- user - logowanie i rejestracja

W skład każdego z modułów wchodzi model (ORM, definicja klas które są odpowiednio mapowane na tabele w bazie danych), widok (odpowiedzialny za przetwarzanie requestów) oraz szablon (odpowiedzialny za front aplikacji).

Struktura URLi w głównym module:

```
urlpatterns = [
    path('', home_view),
    path('admin/', admin.site.urls),
    path('catalog/', include('catalog.urls')),
    path('event/', include('event.urls')),
    path('user/', include('user.urls')),
]
```

Przechodząc w przeglądarce np. pod /event/ aplikacja kieruje request do odpowiedniego modułu (w tym przypadku event), w którym są szczegóły dotyczące struktury URL.

Struktura URLi w module event:

```
app_name = 'event'
urlpatterns = [
    path('', views.event_list, name='list'),
    path('<int:pk>/', views.event_detail, name='detail'),
    path('create/', views.event_create, name='create'),
]
```

Przechodząc np. pod /event/create/ aplikacja kieruję zapytanie w tym wypadku bezpośrednio do funkcji z widoku event_create.

Funkcja event_create w module event:

```
def event_create(request):
    if request.method == 'POST':
        form = EventForm(request, request.POST)
        if form.is_valid():
            name = form.cleaned_data.get('name')
            course = form.cleaned_data.get('course')
            description = form.cleaned_data.get('description')

            course = Course.objects.get(pk=int(course))
            event = Event(name=name, author=request.user, course=course, description=description)
            event.save()

            messages.success(request, _('Event successfully created.'))
            return redirect('event:list')
    else:
        form = EventForm(request)
    return render(request, 'event/event_create.html', {'form': form})
```

Widok przetwarza zapytanie oraz jako odpowiedź renderuję szablon.

Szablon event_create.html w module event:

```
{% extends 'base.html' %}

{% block title %}Event create | {{ block.super }}{% endblock %}

{% block content %}
    <section id="event-create">
        <h3>Event create</h3>
        <form method="post">
            {% csrf_token %}
            {% for field in form %}
                <div class="row mb-3">
                    <label for="{{  field.id_for_label }}" class="col-sm-2 col-form-label">{{ field.label}}</label>
                    <div class="col-sm-10">
                        {{ field }}
                        {% if field.errors %}
                            <div class="text-danger">
                                {{ field.errors.as_text }}
                            </div>
                        {% endif %}
                    </div>
                </div>
            {% endfor %}
            <button type="submit" class="btn btn-primary">Create</button>
        </form>
    </section>
{% endblock %}
```

W module event znajduję się również definicja modeli:

```python
class Event(models.Model):
    name = models.CharField(max_length=128, unique=True)
    author = models.ForeignKey(User, on_delete=models.CASCADE)
    course = models.ForeignKey(Course, on_delete=models.PROTECT)
    status = models.IntegerField(choices=EVENT_STATUS, default=1)
    description = models.TextField()

    class Meta:
        verbose_name_plural = 'events'

    def __str__(self):
        return self.name

    def status_to_str(self):
        return [s[1] for s in EVENT_STATUS if s[0] == self.status][0]

    def set_status_approved(self):
        self.status = 2

    def set_status_archived(self):
        self.status = 3

    def is_waiting(self):
        return self.status == 1

    def is_approved(self):
        return self.status == 2

    def is_archived(self):
        return self.status == 3
```

```python
class Proposition(models.Model):
    author = models.ForeignKey(User, on_delete=models.CASCADE)
    event = models.ForeignKey(Event, on_delete=models.CASCADE)
    datetime = models.DateTimeField()

    class Meta:
        verbose_name_plural = 'propositions'

    def __str__(self):
        return f'{self.event}'
```

```python
class PropositionVote(models.Model):
    proposition = models.ForeignKey(Proposition, on_delete=models.CASCADE)
    author = models.ForeignKey(User, on_delete=models.CASCADE)
    datetime = models.DateTimeField(auto_now_add=True)

    class Meta:
        unique_together = ('proposition', 'author')
        verbose_name_plural = 'proposition votes'

    def __str__(self):
        return f'{self.proposition} from {self.author}'
```