P O L I T E C H N I K A R Z E S Z O W S K A
im. Ignacego Łukasiewicza
WYDZIAŁ MATEMATYKI I FIZYKI STOSOWANEJ

Usługi sieciowe w biznesie

Aplikacja do czatowania w czasie rzeczywistym z wykorzystaniem RabbitMQ

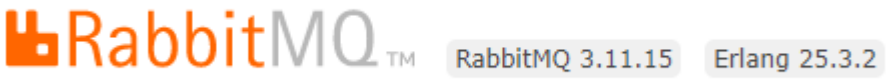
Maciej Żak FS0-DI

Rzeszów , 05.06.2023

Spis treści

Narzędzia wymagane do uruchomienia programu

- RabbitMQ RabbitMQ 3.11.15
- Erlang 25.3.2



- Postgres SQL 13.10

```
C:\Program Files\PostgreSQL\13\bin>psql -V
psql (PostgreSQL) 13.10
```

- Node.js v18.16.0

```
Node.js
Welcome to Node.js v18.16.0.
```

- Python 3.9.13

```
C:\Users\macie>python --version
Python 3.9.13
```

- Framework js Vue 3.3.4

```
C:\Users\macie\Downloads\Chatire-master>npm v vue

vue@3.3.4 | MIT | deps: 5 | versions: 445
The progressive JavaScript framework for building modern web UI
```

- Framework Pythona Django 2.2

```
C:\Users\macie>python -m django --version
2.2
```

Uruchomienie programu

Vue

1. Ustawiamy ścieżkę w konsoli na chatire-frontend ~ *cd …/ Chat-master /chatire-frontend*
2. Następnie instalujemy zależności ~ *npm install*
3. Teraz możemy odpalić serwer deweloperski Webpack ~ *npm run dev*

Django

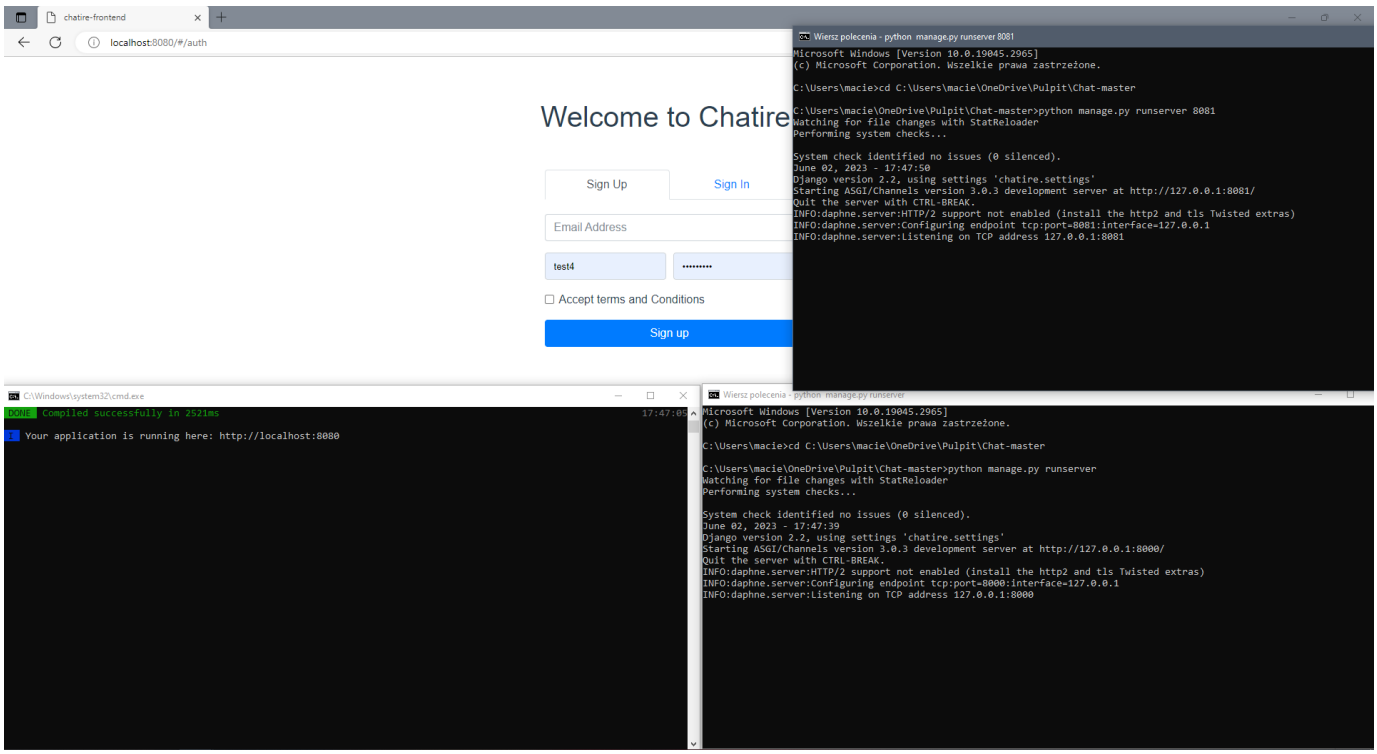
1. Ustawiamy ścieżkę w konsoli na Chat-master ~ *cd…/ Chat-master*
2. Instalujemy wymagane pakiety na podstawie pliku ~ *pip install -r requirements.txt*
3. Instalowanie warstwy kanałowej Django oraz warstwy kanałowej Postgres ~ *pip install channels channels-postgres*
4. Migracja bazy danych ~ *python manage.py migrate*
5. Teraz możemy odpalić serwer deweloperski Django ~ *python manage.py runserver*

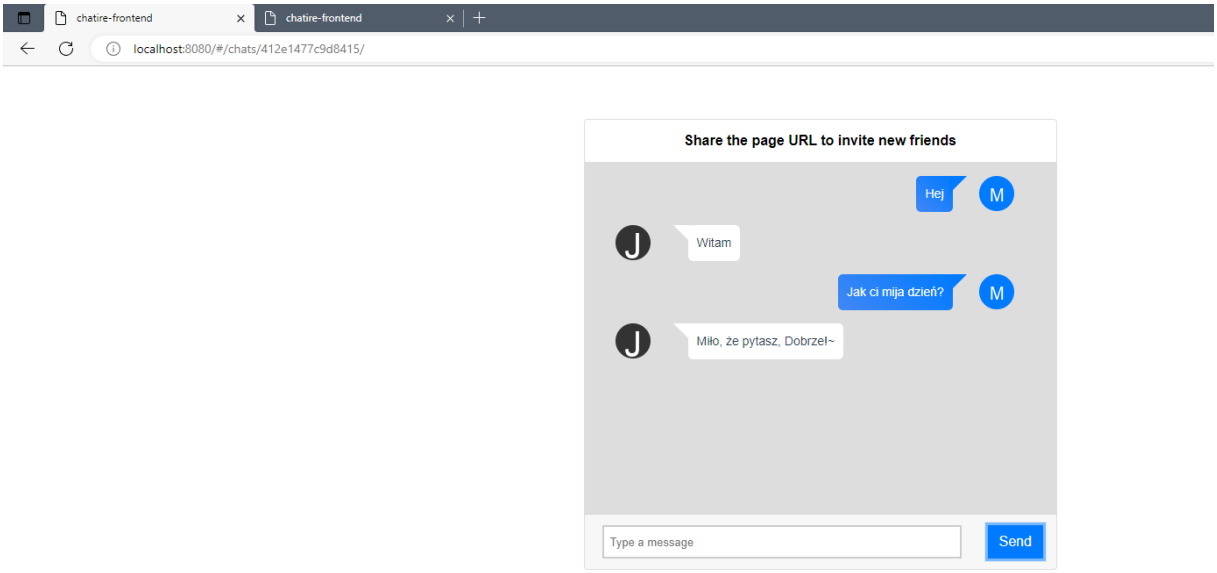
Krok drugi, trzeci i czwarty wystarczy wykonać raz przed pierwszym uruchomieniem

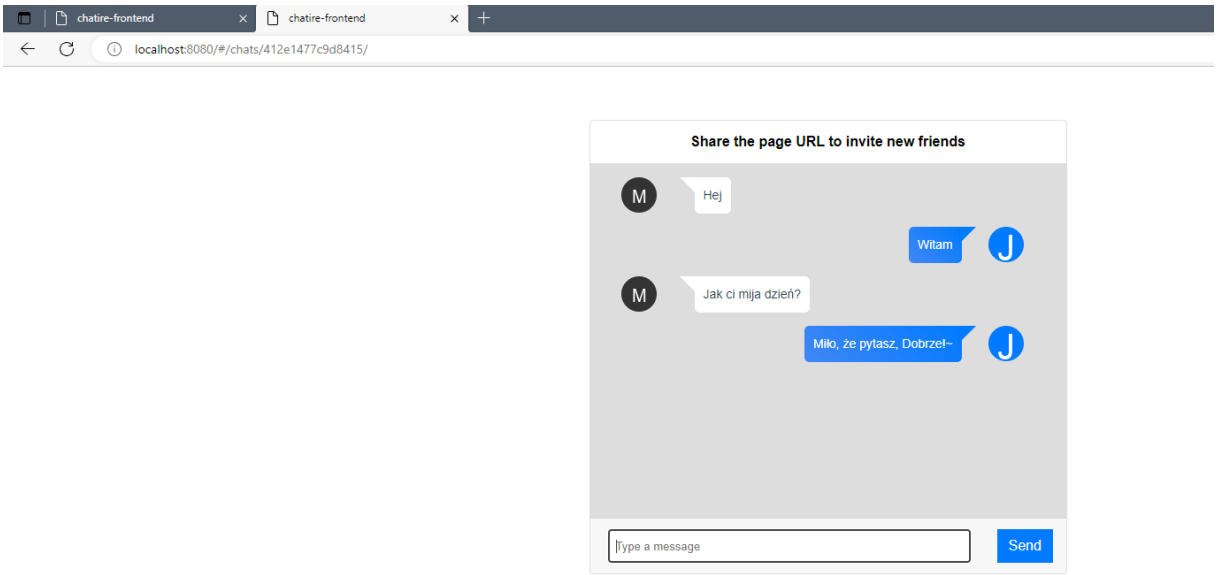
RabbitMQ

1. Ustawiamy ścieżkę w konsoli na Chat-master ~ *cd…/ Chat-master*
2. Odpalamy połączenie z serwerem RabbitMQ ~ *python manage.py runserver 8081*

Teraz po uruchomieniu w przeglądarce http://localhost:8080 możemy się zarejestrować oraz rozpocząć czatowanie.




chatire-frontend
chatire-frontend
localhost:8080/#/chats/412e1477c9d8415/
Share the page URL to invite new friends
Hej
M
J
Witam
Jak ci mija dzień?
M
J
Miło, że pytasz, Dobrze!~
Type a message
Send


chatire-frontend
chatire-frontend
localhost:8080/#/chats/412e1477c9d8415/
Share the page URL to invite new friends
M
Hej
Witam
J
M
Jak ci mija dzień?
Miło, że pytasz, Dobrze!~
J
Type a message
Send

Opis użytych narzędzi

RabbitMQ

RabbitMQ jest jednym z najpopularniejszych i zaawansowanych systemów przetwarzania wiadomości dostępnych na rynku. Jego architektura opiera się na modelu wydawca-subskrybent, który umożliwia elastyczne i efektywne przesyłanie danych w czasie rzeczywistym.

Głównym elementem architektury RabbitMQ są kolejki, które pełnią rolę buforów do przechowywania wiadomości. Kiedy nadawca wysyła wiadomość do RabbitMQ, jest ona umieszczana w odpowiedniej kolejce. Następnie, jedno lub wiele programów odbiorczych (subskrybentów) mogą odebrać te wiadomości z kolejki i przetworzyć je. Kolejki RabbitMQ są elastyczne i umożliwiają skalowanie systemu w celu obsługi dużej ilości wiadomości.

W celu skierowania wiadomości do odpowiednich subskrybentów, RabbitMQ wykorzystuje mechanizm wymian (exchanges) i kluczy wiązania (binding keys). Wymiany pełnią rolę punktów dostępowych do kolejek i mogą kierować wiadomości na podstawie reguł zdefiniowanych przez klucze wiązania. Istnieje kilka typów wymian w RabbitMQ, takich jak wymiana typu "direct", "fanout", "topic" i "headers", które oferują różne sposoby kierowania wiadomości do subskrybentów.

RabbitMQ zapewnia również niezawodność dostarczenia wiadomości poprzez mechanizm potwierdzeń dostarczenia (message acknowledgments). Po odebraniu wiadomości przez subskrybenta, może on wysłać potwierdzenie (acknowledgment) do RabbitMQ, informujące o poprawnym odbiorze i przetworzeniu wiadomości. Jeśli subskrybent nie potwierdzi odbioru, RabbitMQ będzie próbował ponownie dostarczyć wiadomość, zapewniając niezawodność dostarczenia danych.

Dodatkowo, RabbitMQ umożliwia trwałe przechowywanie wiadomości nawet w przypadku awarii systemu. Wiadomości mogą być zapisywane na dysku, dzięki czemu po ponownym uruchomieniu systemu będą dostępne do odczytu i przetworzenia. Ta funkcja zapewnia integralność danych i minimalizuje ryzyko utraty informacji.

Erlang

Erlang to język programowania oraz środowisko wykonawcze, które zostały stworzone w celu budowy niezawodnych i skalowalnych systemów rozproszonych. Erlang ma bogatą historię, sięgającą lat 80. XX wieku, gdy został opracowany przez firmę Ericsson jako narzędzie do budowy telekomunikacyjnych systemów czasu rzeczywistego.

Język Erlang został zaprojektowany z myślą o obsłudze równoległych i rozproszonych procesów. Wykorzystuje model aktorów, w którym poszczególne procesy komunikują się ze sobą za pomocą wiadomości. Dzięki temu modelowi Erlang umożliwia tworzenie systemów, w których setki, tysiące lub nawet miliony procesów mogą współistnieć i komunikować się w sposób niezawodny.

Jedną z głównych cech języka Erlang jest jego zdolność do obsługi błędów i zapewniania wysokiej dostępności systemów. Mechanizmy takie jak nadzorca procesów (supervisor) i restartowanie procesów w przypadku awarii pozwalają na budowę systemów, które są odporne na błędy i mają minimalny czas przestoju.

Erlang posiada również wbudowane narzędzia do obsługi rozproszonych systemów, takich jak mechanizmy komunikacji międzywęzłowej i replikacji danych. Dzięki temu możliwe jest budowanie skalowalnych systemów, które mogą działać na wielu węzłach sieciowych i obsługiwać duże obciążenia.

Język Erlang jest również znany ze swojej wydajności. Dzięki zoptymalizowanemu środowisku wykonawczemu i wbudowanemu systemowi zarządzania pamięcią, Erlang oferuje szybkie i efektywne przetwarzanie danych.

Erlang jest wykorzystywany w różnych dziedzinach, w tym telekomunikacji, finansach, systemach komunikacji w czasie rzeczywistym, a także w budowie rozproszonych systemów przetwarzania danych i chmur obliczeniowych.

Postgres SQL

PostgreSQL jest zaawansowanym, otwartoźródłowym systemem zarządzania bazami danych (DBMS), który obsługuje zarówno SQL (Strukturalne Języki Zapytań) jak i JSON (JavaScript Object Notation) do zarządzania danymi. Opracowany i utrzymany przez społeczność PostgreSQL Global Development Group, PostgreSQL jest dostępny na wielu platformach, w tym Linux, Windows, FreeBSD, Solaris i MacOS.

Pierwsze wydanie PostgreSQL nastąpiło w 1996 roku i od tego czasu zyskało popularność wśród deweloperów na całym świecie, zdobywając uznanie za swoją wydajność, skalowalność i elastyczność. Obecna wersja w momencie mojego ostatniego treningu (wrzesień 2021) to PostgreSQL 13.

PostgreSQL obsługuje szeroki zakres typów danych, w tym pierwotne, jak tekst, liczby całkowite, zmiennoprzecinkowe, binarne i logiczne, jak również zaawansowane typy, takie jak tablice, hstore (dla pary klucz-wartość) i typy geolokalizacyjne. PostgreSQL jest również wyjątkowo rozszerzalny, umożliwiając tworzenie i dodawanie nowych typów danych, operatorów i funkcji.

Wśród innych zaawansowanych funkcji PostgreSQL, warto zwrócić uwagę na:

- Transakcje ACID: PostgreSQL zapewnia atomowość, spójność, izolację i trwałość (ACID), zapewniając bezpieczeństwo danych.
- Wsparcie dla Języków Programowania: PostgreSQL obsługuje wiele języków programowania, w tym Python, Java, C/C++, PHP, Ruby, Perl, .NET i wielu innych.
- Wsparcie dla Procedur i Funkcji Przechowywanych: PostgreSQL pozwala na tworzenie procedur i funkcji przechowywanych za pomocą PL/pgSQL (podobnego do PL/SQL Oracle) lub dowolnego innego obsługiwanego języka programowania.
- Indeksowanie: PostgreSQL oferuje wiele typów indeksów, w tym B-tree, hash, GIN (Generalized Inverted Index) i GiST (Generalized Search Tree), co pozwala na szybkie wyszukiwanie danych.
- Replikacja i Partycjonowanie Danych: PostgreSQL oferuje natywne wsparcie dla replikacji (zarówno master-slave, jak i multi-master) i partycjonowania danych, co ułatwia skalowanie horyzontalne.

PostgreSQL jest odpowiedni dla szerokiego zakresu zastosowań, od małych, lokalnych baz danych po duże systemy gromadzące dane na skalę globalną. Ze względu na jego otwartoźródłowy charakter i wszechstronność, PostgreSQL jest często wybierany przez organizacje różnej wielkości, w tym korporacje, instytucje edukacyjne, rządy i startupy.

Node.js i Framework Vue

Node.js to platforma, której głównym przeznaczeniem jest tworzenie skalowalnych aplikacji sieciowych, zwłaszcza serwerów. Została stworzona na silniku JavaScript V8 od Google, który jest używany w przeglądarce Chrome. Pozwala ona na używanie JavaScriptu po stronie serwera, co oznacza, że deweloperzy mogą pisać zarówno kod klienta, jak i serwera w tym samym języku.

Jednym z kluczowych elementów Node.js jest model obsługi zdarzeń I/O bez blokowania. W praktyce oznacza to, że operacje wejścia/wyjścia, takie jak odczyt i zapis do systemu plików, nie blokują działania całej aplikacji, co pozwala na obsługę wielu zapytań równocześnie.

Node.js oferuje także bogaty ekosystem modułów dostępnych poprzez NPM (Node Package Manager), co pozwala deweloperom na skorzystanie z gotowych rozwiązań do wielu typowych problemów, bez potrzeby tworzenia wszystkiego od zera, jest często używany do tworzenia aplikacji czasu rzeczywistego, takich jak czaty czy gry online, a także do budowania szybkich i skalowalnych API, zwłaszcza w architekturze REST.

Kiedy już mówimy o JavaScript po stronie serwera, warto również wspomnieć o Vue.js - jednym z najpopularniejszych frameworków JavaScript używanych do budowy interfejsów użytkownika.

Vue.js, zaprojektowany przez Evana You w 2014 roku, jest często porównywany do innych popularnych frameworków jak React i Angular, ale Vue wyróżnia się swoją prostotą i elastycznością. Vue.js jest często opisywany jako framework progresywny, co oznacza, że deweloperzy mogą go stopniowo adoptować, zaczynając od małych części istniejącej aplikacji, a kończąc na pełnej jednostronicowej aplikacji.

Podobnie jak inne współczesne frameworki JavaScript, Vue.js korzysta z Virtual DOM, co pozwala na efektywne aktualizowanie i renderowanie komponentów. Vue.js oferuje również system reaktywnych danych, co oznacza, że kiedy dane używane przez komponent się zmieniają, Vue automatycznie aktualizuje DOM. Vue.js obsługuje komponenty, co umożliwia tworzenie wielokrotnego użytku i łatwe do testowania fragmentów interfejsu. W Vue.js, HTML, JavaScript i CSS są zazwyczaj zawarte w jednym pliku komponentu, co sprawia, że kod jest łatwy do zrozumienia i utrzymania.

Python i Framework Django

Python, stworzony przez Guido van Rossuma w 1991 roku, to potężny i wszechstronny język programowania. Jego charakterystyka, obejmująca dynamiczne typowanie, silne typowanie i automatyczne zarządzanie pamięcią, czyni go wyjątkowo efektywnym w wielu zastosowaniach.

Python jest językiem interpretowanym, co oznacza, że kod jest wykonywany linia po linii, co ułatwia debugowanie. Jego składnia jest zdecydowanie prostsza i bardziej zrozumiała w porównaniu do wielu innych języków programowania, co czyni go doskonałym narzędziem dla początkujących programistów, jednocześnie nie ograniczając możliwości doświadczonych deweloperów.

Python obsługuje różne paradygmaty programowania, w tym proceduralne, obiektowe i funkcjonalne. Zawiera też wiele wbudowanych typów danych wysokiego poziomu, takich jak słowniki, listy, krotki i zestawy. Jego standardowa biblioteka jest bardzo obszerna i obejmuje moduły dla wielu zastosowań, od generowania plików XML, przez tworzenie interfejsów graficznych, po manipulację danymi naukowymi. Jednym z najważniejszych elementów ekosystemu Pythona jest Django - potężny framework do tworzenia aplikacji internetowych. Django, wydany po raz pierwszy w 2005 roku, opiera się na filozofii "baterii dołączonych", co oznacza, że zawiera wiele gotowych do użycia narzędzi i funkcji, które mogą przyspieszyć proces tworzenia aplikacji.

Framework Django opiera się na architekturze model-widok-szablon (MVT). "Model" reprezentuje dane, z którymi aplikacja pracuje, na przykład rekordy w bazie danych. "Widok" to miejsce, gdzie decyzje dotyczące tego, co ma być prezentowane, są podejmowane, a "Szablon" to miejsce, gdzie decyduje się, jak te dane mają być prezentowane.

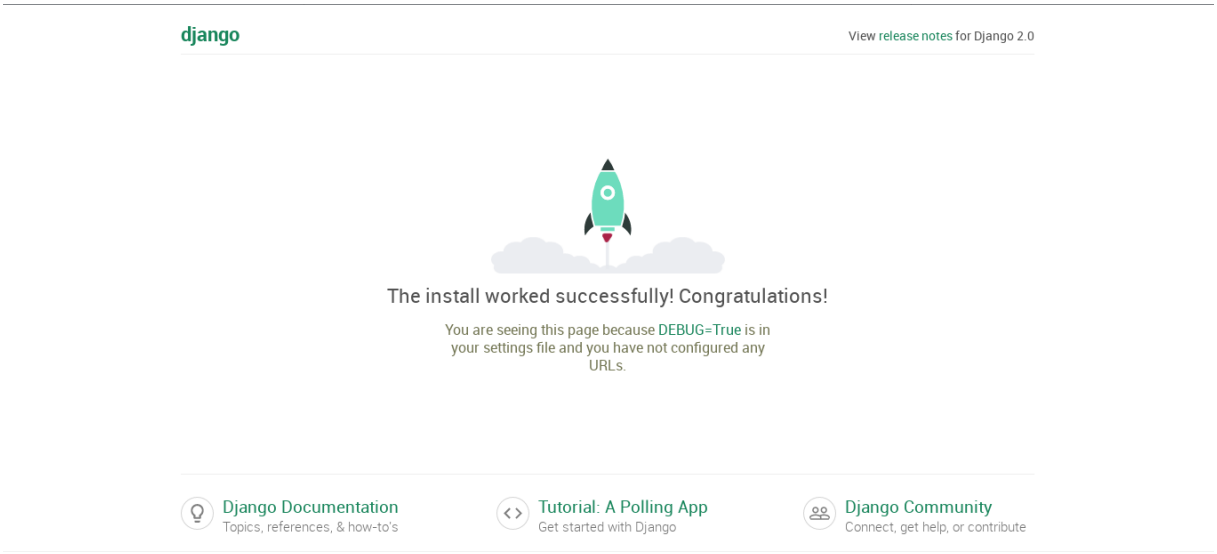
Django jest cenione za swoje narzędzia do obsługi uwierzytelniania użytkowników, mapowania obiektowo-relacyjnego, obsługi formularzy, a także za swoje silne zabezpieczenia. Django posiada również wbudowane narzędzia do tworzenia stron administracyjnych, co pozwala na szybkie i łatwe zarządzanie danymi aplikacji.

Realizacja

Ustawienie Django

Pierwszą rzeczą, którą zrobimy, będzie instalacja Django, stworzenie nowego projektu oraz uruchomienie go.

```
$ pip install django
$ django-admin startproject chatire
$ python manage.py migrate
$ python manage.py runserver
```



Następnie instalujemy resztę potrzebnych bibliotek

```
$pip install djangorestframework
$pip install djoser
```

Oraz edytujemy plik wygenerowany przez Django „settings.py” tak aby djoser był dodany.

```
INSTALLED_APPS = (
    'django.contrib.auth',
    ...,
    'rest_framework',
    'rest_framework.authtoken',
    'djoser',
))
```

Oraz dodajemy adres do djosera w pliku urls.py

```
from django.contrib import admin
from django.urls import path, include


urlpatterns = [
    ...,
    path('auth/', include('djoser.urls')),
    path('auth/', include('djoser.urls.authtoken')),
]
```

Oraz dodajemy klasę autentyfikacji do frameworka django rest.

```
REST_FRAMEWORK = {
    'DEFAULT_AUTHENTICATION_CLASSES': (
        'rest_framework_jwt.authentication.JSONWebTokenAuthentication',
    )
}
```

Ustawienie Vue.js

Instalujemy klienta vue-cli

```
npm install -g vue-cli
```

Oraz tworzymy nowy projekt

```
vue init webpack chatire-frontend
```

Po uruchomieniu polecenia *npm run dev* . Możemy na hoście lokalnym *localhost:8080* odpalić wygenerowany plik.

Welcome to Your Vue.js App

Essential Links

Core Docs Forum Community Chat Twitter
Docs for This Template

Ecosystem

vue-router vuex vue-loader awesome-vue

Konfigurowanie routera Vue

Idealnym rozwiązaniem jest warunkowe wyświetlanie komponentów na podstawie statusu logowania użytkownika. Jeśli użytkownik jest uwierzytelniony, chcemy wyświetlić komponent Chat, w przeciwnym razie chcemy, aby zarejestrował się lub zalogował, co oznacza, że wyświetlimy komponent UserAuth.

Możemy to zrobić, tworząc globalną nawigację. Edytuj plik index.js routera, aby zawierał następujące elementy:

```
Vue.use(Router)

const router = new Router({
  routes: [
    {
      path: '/chats/:uri?',
      name: 'Chat',
      component: Chat
    },

    {
      path: '/auth',
      name: 'UserAuth',
      component: UserAuth
    }
  ]
})

router.beforeEach((to, from, next) => {
  if (sessionStorage.getItem('authToken') !== null || to.path === '/auth') {
    next()
  } else {
    next('/auth')
  }
})

export default router
```

Strażnik beforeEach jest wywoływany przed przejściem do dowolnej trasy w naszej aplikacji. Jeśli token jest przechowywany w sessionStorage, zezwalamy na kontynuowanie nawigacji poprzez wywołanie next(), w przeciwnym razie przekierowujemy do komponentu auth. Bez względu na trasę, do której użytkownik nawiguje w naszej aplikacji, funkcja ta sprawdzi, czy użytkownik ma token auth i odpowiednio go przekieruje.

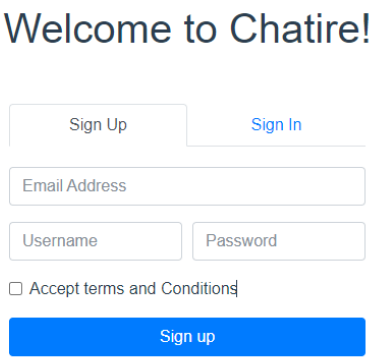
Tworzenie Strony Logowania/Rejestracji

Zbudowałem prostą stronę logowania / rejestracji z zakładkami Bootstrap 4

```
<template>
  <div class="container">
    <h1 class="text-center">Welcome to Chatire!</h1>
    <div id="auth-container" class="row">
      <div class="col-sm-4 offset-sm-4">
        <ul class="nav nav-tabs nav-justified" id="myTab" role="tablist">
          <li class="nav-item">
               <a class="nav-link active" id="signup-tab" data-toggle="tab" href="#signup" role="tab"
aria-controls="signup" aria-selected="true">Sign Up</a>
          </li>
          <li class="nav-item">
                    <a  class="nav-link"  id="signin-tab"  data-toggle="tab"  href="#signin"  role="tab"
aria-controls="signin" aria-selected="false">Sign In</a>
          </li>
        </ul>

        <div class="tab-content" id="myTabContent">

                          <div   class="tab-pane   fade   show   active"   id="signup"   role="tabpanel"
aria-labelledby="signin-tab">
            <form @submit.prevent="signUp">
              <div class="form-group">
                              <input  v-model="email"  type="email"  class="form-control"  id="email"
placeholder="Email Address" required>
              </div>
              <div class="form-row">
                <div class="form-group col-md-6">
                             <input  v-model="username"  type="text"  class="form-control"  id="username"
placeholder="Username" required>
                </div>
                <div class="form-group col-md-6">
                          <input  v-model="password"  type="password"  class="form-control"  id="password"
placeholder="Password" required>
                </div>
              </div>
```

```
              <div class="form-group">
                <div class="form-check">
                  <input class="form-check-input" type="checkbox" id="toc" required>
                  <label class="form-check-label" for="gridCheck">
                    Accept terms and Conditions
                  </label>
                </div>
              </div>
              <button type="submit" class="btn btn-block btn-primary">Sign up</button>
            </form>
          </div>

          <div class="tab-pane fade" id="signin" role="tabpanel" aria-labelledby="signin-tab">
            <form @submit.prevent="signIn">
              <div class="form-group">
                          <input  v-model="username"  type="text"  class="form-control"  id="username"
placeholder="Username" required>
              </div>
              <div class="form-group">
                        <input  v-model="password"  type="password"  class="form-control"  id="password"
placeholder="Password" required>
              </div>
              <button type="submit" class="btn btn-block btn-primary">Sign in</button>
            </form>
          </div>

        </div>
      </div>
    </div>
  </div>
</template>
```

Strona prezentuje się następująco, ważne żeby pamiętać o dołączeniu tego pliku do głównego pliku index.html



Ustawienie tokenu autoryzacji

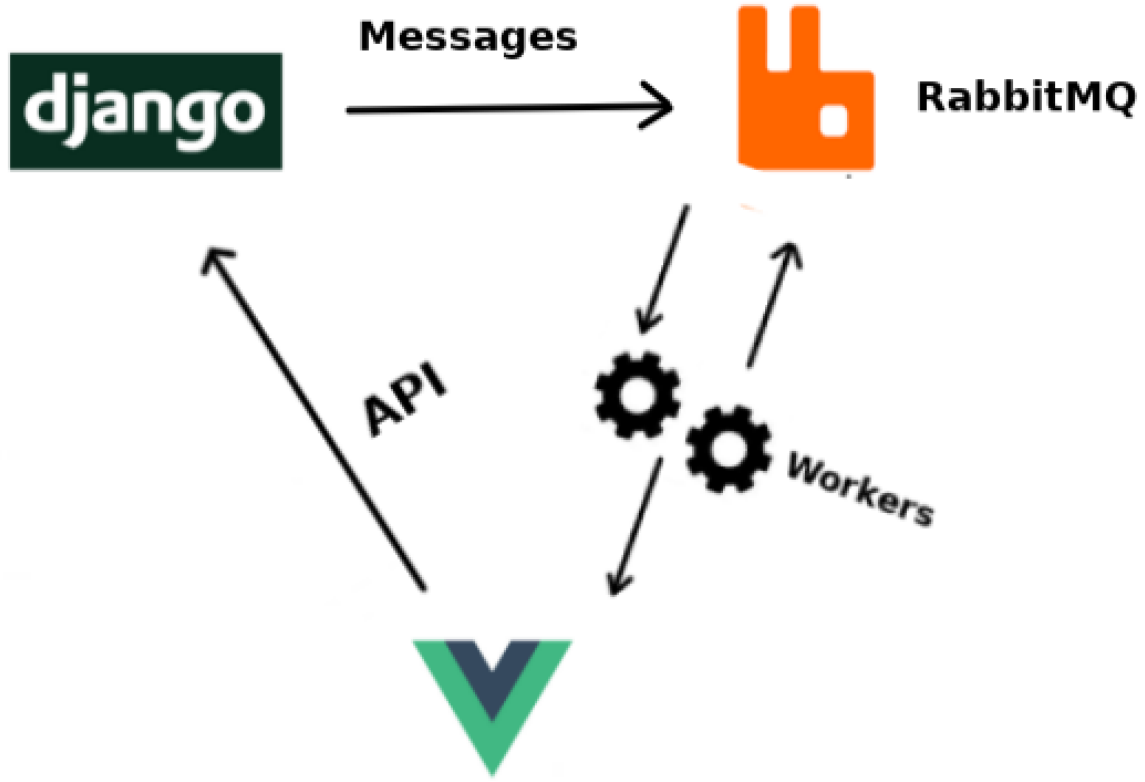
Chcemy zarejestrować użytkownika, a następnie przekierować go do trasy czatu.

Aby to osiągnąć, musimy zaimplementować metody signUp i signIn, które określiliśmy wcześniej:

```
methods: {
  signUp () {
    $.post('http://localhost:8000/auth/users/create/', this.$data, (data) => {
      alert("Your account has been created. You will be signed in automatically")
      this.signIn()
    })
    .fail((response) => {
      alert(response.responseText)
    })
  },

  signIn () {
    const credentials = {username: this.username, password: this.password}

    $.post('http://localhost:8000/auth/token/create/', credentials, (data) => {
      sessionStorage.setItem('authToken', data.auth_token)
      sessionStorage.setItem('username', this.username)
      this.$router.push('/chats')
    })
    .fail((response) => {
      alert(response.responseText)
    })
  }
}
```

Architektura projektu

Zanim przejdziemy do dalszej części, omówmy, jak wszystko działa.



- Gdy użytkownik wysyła wiadomość, ta wiadomość jest przekazywana do django przez API.
- Gdy django otrzyma wiadomość, zostanie ona również przekazana do RabbitMQ.
- RabbitMQ używa wymiany, by rozgłaszać wiadomości do wielu kolejek. Kolejki są kanałami komunikacyjnymi, które ostatecznie dostarczają wiadomości do klientów.

Tworzenie Czatu

Stwórzmy nową aplikację django o nazwie chat i dodajemy do **INSTALLED_APPS**

$ python manage.py startapp chat

Następnie utworzymy modele, które będą przechowywać dane wiadomości, sesji czatu i powiązanych użytkowników. Utwórzmy kilka nowych modeli w models.py.

```python
def deserialize_user(user):
    """Deserialize user instance to JSON."""
    return {
        'id': user.id, 'username': user.username, 'email': user.email,
        'first_name': user.first_name, 'last_name': user.last_name
    }


class TrackableDateModel(models.Model):
    """Abstract model to Track the creation/updated date for a model."""

    create_date = models.DateTimeField(auto_now_add=True)
    update_date = models.DateTimeField(auto_now=True)

    class Meta:
        abstract = True


def _generate_unique_uri():
    """Generates a unique uri for the chat session."""
    return str(uuid4()).replace('-', '')[:15]


class ChatSession(TrackableDateModel):
    """
    A Chat Session.

    The uri's are generated by taking the first 15 characters from a UUID
    """

    owner = models.ForeignKey(User, on_delete=models.PROTECT)
    uri = models.URLField(default=_generate_unique_uri)

    def __str__(self):
        return self.uri


class ChatSessionMessage(TrackableDateModel):
    """Store messages for a session."""

    user = models.ForeignKey(User, on_delete=models.PROTECT)
    chat_session = models.ForeignKey(
        ChatSession, related_name='messages', on_delete=models.PROTECT
    )
    message = models.TextField(max_length=2000)

    def to_json(self):
        """deserialize message to JSON."""
        return {'user': deserialize_user(self.user), 'message': self.message}


class ChatSessionMember(TrackableDateModel):
    """Store all users in a chat session."""

    chat_session = models.ForeignKey(
        ChatSession, related_name='members', on_delete=models.PROTECT
    )
    user = models.ForeignKey(User, on_delete=models.PROTECT)
```

Przed kolejnym krokiem należy przeprowadzić migracje, aby tabele bazy danych mogły zostać utworzone.

Następnym krokiem jest stworzenie widoków (punktów końcowych API), które będą używane przez naszą aplikację Vue do manipulowania danymi na serwerze. Skorzystamy z frameworka django rest, aby je utworzyć (nie będziemy korzystać z serializerów, ponieważ nasze modele są dość proste). Zróbmy to teraz w views.py

```python
class ChatSessionView(APIView):
    """Manage Chat sessions."""

    permission_classes = (permissions.IsAuthenticated,)

    def post(self, request, *args, **kwargs):
        """create a new chat session."""
        user = request.user

        chat_session = ChatSession.objects.create(owner=user)

        return Response({
            'status': 'SUCCESS', 'uri': chat_session.uri,
            'message': 'New chat session created'
        })

    def patch(self, request, *args, **kwargs):
        """Add a user to a chat session."""
        User = get_user_model()

        uri = kwargs['uri']
        username = request.data['username']
        user = User.objects.get(username=username)

        chat_session = ChatSession.objects.get(uri=uri)
        owner = chat_session.owner

        if owner != user:  # Only allow non owners join the room
            chat_session.members.get_or_create(
                user=user, chat_session=chat_session
            )

        owner = deserialize_user(owner)
        members = [
            deserialize_user(chat_session.user)
            for chat_session in chat_session.members.all()
        ]
        members.insert(0, owner)  # Make the owner the first member

        return Response({
            'status': 'SUCCESS', 'members': members,
            'message': '%s joined the chat' % user.username,
            'user': deserialize_user(user)
        })
```

```python
class ChatSessionMessageView(APIView):
    """Create/Get Chat session messages."""

    permission_classes = (permissions.IsAuthenticated,)

    def get(self, request, *args, **kwargs):
        """return all messages in a chat session."""
        uri = kwargs['uri']

        chat_session = ChatSession.objects.get(uri=uri)
        messages = [chat_session_message.to_json()
                    for chat_session_message in chat_session.messages.all()]

        return Response({
            'id': chat_session.id, 'uri': chat_session.uri,
            'messages': messages
        })

    def post(self, request, *args, **kwargs):
        """create a new message in a chat session."""
        uri = kwargs['uri']
        message = request.data['message']

        user = request.user
        chat_session = ChatSession.objects.get(uri=uri)

        chat_session_message = ChatSessionMessage.objects.create(
            user=user, chat_session=chat_session, message=message
        )

        notif_args = {
            'source': user,
            'source_display_name': user.get_full_name(),
            'category': 'chat', 'action': 'Sent',
            'obj': chat_session_message.id,
            'short_description': 'You a new message', 'silent': True,
            'extra_data': {
                notifs_settings.NOTIFICATIONS_WEBSOCKET_URL_PARAM:
                chat_session.uri,
                'message': chat_session_message.to_json()
            }
        }
        notify(**notif_args, channels=['websocket'])

        return Response({
            'status': 'SUCCESS', 'uri': chat_session.uri, 'message': message,
            'user': deserialize_user(user)
```

Metoda patch dla ChatSessionView jest idempotentna, ponieważ wielokrotne wykonanie żądania daje nam ten sam wynik. Oznacza to, że użytkownik może dołączyć do pokoju czatu kilka razy, ale w odpowiedzi będzie tylko jedna instancja tego użytkownika (a także w naszej tabeli bazy danych).

Inną rzeczą, na którą należy zwrócić uwagę w metodzie patch, jest to, że zwraca ona właściciela pokoju czatu jako członka, ale w naszej bazie danych nigdy nie dodajemy właściciela jako członka pokoju, po prostu pobieramy jego informacje i wstawiamy je do listy, która jest zwracana do klienta. Nie ma sensu powielać informacji, mając właściciela jako członka swojego pokoju rozmów w bazie danych.

Mogliśmy łatwo uzyskać użytkownika w metodzie patch, wywołując request.user, zamiast tego uzyskaliśmy nazwę użytkownika z przesłanych danych i użyliśmy jej do uzyskania użytkownika. Powoduje to dodatkowy SELECT do bazy danych, ale dlaczego to zrobiliśmy?

Podam prosty scenariusz, co się stanie, jeśli zdecydujemy się zaprosić naszych znajomych według nazwy użytkownika do sesji czatu. Z request.user nie bylibyśmy w stanie tego zrobić, ponieważ request.user odnosiłby się do bieżącego uwierzytelnionego użytkownika wykonującego żądanie. Z drugiej strony, z nazwą użytkownika to bułka z masłem, musimy tylko wysłać nazwę użytkownika do serwera, a on użyje jej do pobrania użytkownika i dodania go do pokoju rozmów. Używanie nazw użytkowników sprawia, że nasz kod jest bardziej elastyczny i otwarty na ulepszenia.

Teraz dodajmy adresy URL widoków.

```python
"""URL's for the chat app."""

from django.contrib import admin
from django.urls import path

from . import views

urlpatterns = [
    path('chats/', views.ChatSessionView.as_view()),
    path('chats/<uri>/', views.ChatSessionView.as_view()),
    path('chats/<uri>/messages/', views.ChatSessionMessageView.as_view()),
]
```

Tworzenie HTML, CSS i JS do czatu

Tutaj tworzymy układ strony, jej wygląd i funkcjonalności, jako że kod HTML i CSS jest dość długi, więc skupię się to głównie na opisie funkcjonalności javascript.

```
< script >

  const $ = window.jQuery

export default {
  data() {
    return {
      loading: true,
      messages: [],
      message: '',
      notification: new Audio('../../static/plucky.ogg'),
      sessionStarted: false
    }
  },

  created() {
    this.username = sessionStorage.getItem('username')

    // Setup headers for all requests
    $.ajaxSetup({
      beforeSend: function(xhr) {
        xhr.setRequestHeader('Authorization', `JWT ${sessionStorage.getItem('authToken')}`)
      }
    })

    if (this.$route.params.uri) {
      this.joinChatSession()
      this.connectToWebSocket()
    }

    setTimeout(() => {
      this.loading = false
    }, 2000)

    // Refresh the JWT every 240 Seconds (4 minutes)
    setInterval(this.refreshToken, 240000)

    setInterval(this.fetchChatSessionHistory, 4000)
  },
```

this.username = sessionStorage.getItem('username'): Przypisuje nazwę użytkownika pobraną z sessionStorage do zmiennej this.username.

$.ajaxSetup(): Konfiguruje nagłówki żądań AJAX dla wszystkich żądań wysyłanych przez jQuery. Dodaje nagłówek 'Authorization', który zawiera token uwierzytelniania JWT pobrany z sessionStorage.

if (this.$route.params.uri) { ... }: Sprawdza, czy istnieje parametr uri w trasie (adresie URL). Jeśli istnieje, wywołuje funkcje joinChatSession() i connectToWebSocket().

```
updated() {
  // Scroll to bottom of Chat window
  const chatBody = this.$refs.chatBody
  if (chatBody) {
    chatBody.scrollTop = chatBody.scrollHeight
  }
},

methods: {
  startChatSession() {
    $.post('http://localhost:8000/api/chats/', (data) => {
        alert("A new session has been created you'll be redirected automatically")
        this.sessionStarted = true
        this.$router.push(`/chats/${data.uri}/`)
        this.connectToWebSocket()
      })
      .fail((response) => {
        alert(response.responseText)
      })
  },

  postMessage(event) {
    const data = {
      message: this.message
    }

    $.post(`http://localhost:8000/api/chats/${this.$route.params.uri}/messages/`, data, (data) => {
        this.message = '' // clear the message after sending
      })
      .fail((response) => {
        alert(response.responseText)
      })
  },
```

startChatSession(): Ta funkcja rozpoczyna nową sesję czatu. Wysyła żądanie POST na adres 'http://localhost:8000/api/chats/', aby utworzyć nową sesję czatu na serwerze. Jeśli żądanie zakończy się sukcesem, użytkownik zostanie poinformowany o utworzeniu nowej sesji, ustawiona zostanie flaga sessionStarted na true, a użytkownik zostanie automatycznie przekierowany na stronę czatu z odpowiednim URI. Następnie funkcja nawiązuje połączenie z WebSocket, wywołując funkcję connectToWebSocket().

postMessage(event): Ta funkcja wysyła wiadomość czatu. Tworzy obiekt data zawierający wiadomość pobraną z this.message. Następnie wysyła żądanie POST na adres http://localhost:8000/api/chats/${this.$route.params.uri}/messages/, aby wysłać wiadomość do aktualnej sesji czatu na serwerze. Jeśli żądanie zakończy się sukcesem, pole this.message zostanie wyczyszczone.

```
joinChatSession() {
  const uri = this.$route.params.uri

  $.ajax({
    url: `http://localhost:8000/api/chats/${uri}/`,
    data: {
      username: this.username
    },
    type: 'PATCH',
    success: (data) => {
      const user = data.members.find((member) => member.username === this.username)

      if (user) {
        // The user belongs/has joined the session
        this.sessionStarted = true
        this.fetchChatSessionHistory()
      }
    }
  })
},

fetchChatSessionHistory() {
  $.get(`http://127.0.0.1:8000/api/chats/${this.$route.params.uri}/messages/`, (data) => {
    this.messages = data.messages
    setTimeout(() => {
      this.loading = false
    }, 2000)
    this.$forceUpdate() // Force Vue.js to re-render the component
  })
},

connectToWebSocket() {
  const websocket = new WebSocket(`ws://localhost:8081/${this.$route.params.uri}`)
  websocket.onopen = this.onOpen
  websocket.onclose = this.onClose
  websocket.onmessage = this.onMessage
  websocket.onerror = this.onError
},
```

joinChatSession(): Ta funkcja dołącza do istniejącej sesji czatu. Pobiera URI z parametrów trasy (this.$route.params.uri) i wysyła żądanie PATCH na adres http://localhost:8000/api/chats/${uri}/, aby dołączyć do sesji czatu na serwerze. Przesyła również nazwę użytkownika w parametrze username. Jeśli żądanie zakończy się sukcesem i użytkownik zostanie znaleziony w liście członków sesji, ustawiana jest flaga sessionStarted na true, a następnie wywoływana jest funkcja fetchChatSessionHistory().

fetchChatSessionHistory(): Ta funkcja pobiera historię wiadomości sesji czatu. Wysyła żądanie GET na adres

connectToWebSocket(): Ta funkcja nawiązuje połączenie z WebSocket. Tworzy nowy obiekt WebSocket, korzystając z adresu ws://localhost:8081/${this.$route.params.uri}, gdzie ${this.$route.params.uri} to dynamicznie pobrany URI sesji czatu. Następnie przypisuje odpowiednie funkcje do zdarzeń WebSocket, takie jak onopen, onclose, onmessage i onerror, aby obsługiwać odpowiednie zdarzenia podczas połączenia WebSocket.

```
  onOpen(event) {
    console.log('Connection opened.', event.data)
  },

  onClose(event) {
    console.log('Connection closed.', event.data)

    // Try and Reconnect after five seconds
    setTimeout(this.connectToWebSocket, 5000)
  },

  onMessage(event) {
    const message = JSON.parse(event.data)
    this.$set(this.messages, this.messages.length, message)

    this.$nextTick(() => {
      const chatBody = this.$refs.chatBody
      if (chatBody) {
        chatBody.scrollTop = chatBody.scrollHeight
      }
    })

    if (!document.hasFocus()) {
      this.notification.play()
    }
  },

  onError(event) {
    alert('An error occured:', event.data)
  },

  refreshToken() {
    const data = {
      token: sessionStorage.getItem('authToken')
    }

    $.post('http://127.0.0.1:8000/this/is/hard/to/find/', data, (response) => {
      sessionStorage.setItem('authToken', response.token)
    })
  }
}
```

onOpen(event): Ta funkcja jest wywoływana, gdy połączenie WebSocket zostanie otwarte. Wypisuje informację o otwartym połączeniu w konsoli.

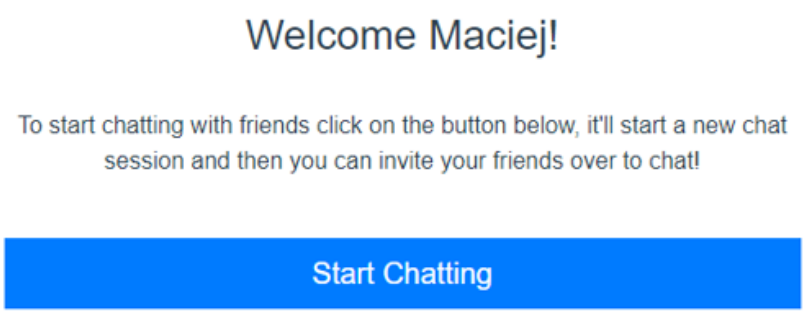
onClose(event): Ta funkcja jest wywoływana, gdy połączenie WebSocket zostanie zamknięte. Wypisuje informację o zamkniętym połączeniu w konsoli i próbuje ponownie połączyć się z WebSocket po pięciu sekundach, wywołując funkcję connectToWebSocket().

onMessage(event): Ta funkcja jest wywoływana, gdy otrzymano wiadomość przez WebSocket. Parsuje otrzymaną wiadomość JSON i dodaje ją do tablicy this.messages przy użyciu this.$set(). Następnie przewija okno czatu na dół, aby wyświetlić najnowsze wiadomości. Jeśli okno przeglądarki nie jest aktywne, odtwarza dźwięk powiadomienia.

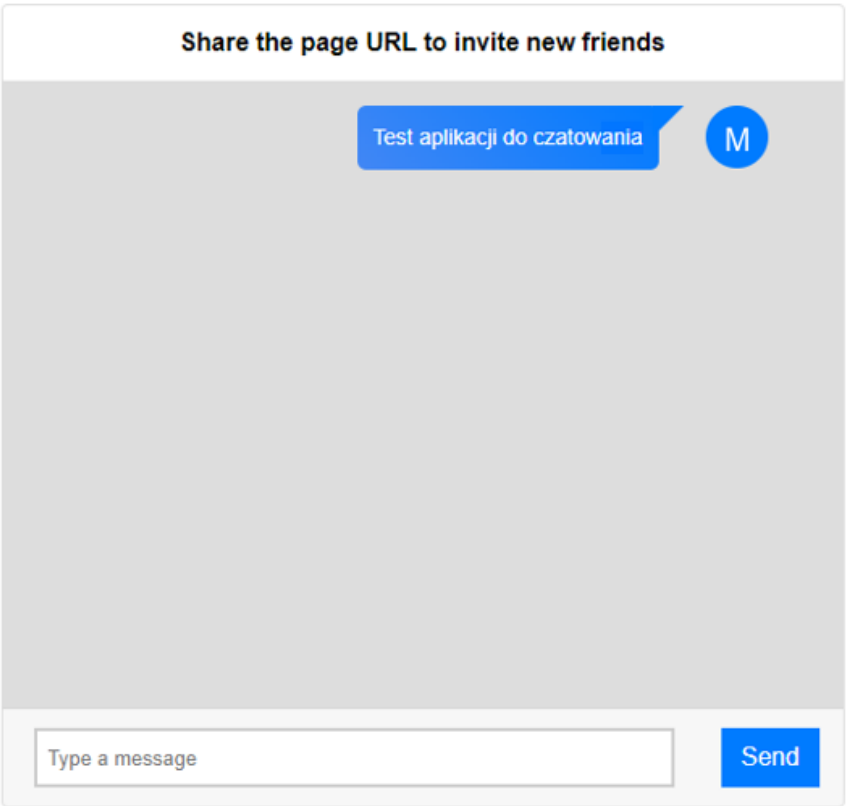
onError(event): Ta funkcja jest wywoływana, gdy wystąpi błąd WebSocket. Wyświetla alert z informacją o błędzie.

refreshToken(): Ta funkcja odświeża token uwierzytelniania JWT. Tworzy obiekt data zawierający token pobrany z sessionStorage. Następnie wysyła żądanie POST na adres 'http://127.0.0.1:8000/this/is/hard/to/find/', aby odświeżyć token na serwerze. Po otrzymaniu odpowiedzi od serwera, zaktualizowany token jest przechowywany w sessionStorage.

Po zaimplementowaniu tych funkcjonalności, jesteśmy w stanie się zalogować i utworzyć nowy czat a także, zaprosić do niego innego użytkownika



Aplikacja jeszcze nie przesyła wiadomości w czasie rzeczywistym, aby zobaczyć kolejne wiadomości trzeba za każdym razem odświeżać stronę. Zajmiemy się tym w kolejnym kroku.



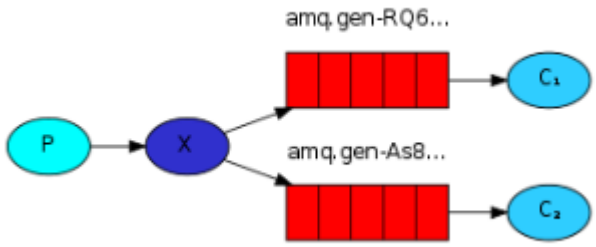
RabbitMQ

Django-notifs wykorzystuje kanały do dostarczania wiadomości. Oznacza to, że możesz napisać własny kanał do dostarczania wiadomości za pośrednictwem e-maili, SMS-ów, Slacka i wszystkiego, co tylko przyjdzie do głowy. Ma nawet wbudowany kanał websocket, ale to nie wystarczy w naszym przypadku, ponieważ jest to kanał użytkownik-użytkownik.

Chcemy rozgłaszać wiadomości do wielu klientów jednocześnie. Ten wzorzec komunikacji nazywa się Pub/Sub (Publish Subscribe), a RabbitMQ obsługuje go jako wymianę.

Wymiana to kanał, który odbiera wiadomości od producenta (naszej aplikacji), a następnie rozsyła je do wielu kolejek. Istnieją cztery różne typy wymiany, a mianowicie bezpośrednia, tematyczna, nagłówkowa i fanout. Skorzystamy z wymiany fanout, która jest najprostsza do zrozumienia i idealnie pasuje do naszego przypadku użycia.

Jest to ilustracja z dokumentacji RabbitMQ na temat działania wymiany fanout:



Zanim kolejka będzie mogła odebrać wiadomość, musi zostać powiązana z centralą. Aby zaimplementować wzorzec Pub/Sub, musimy napisać własny kanał dostarczania. Zrobimy to tworząc nowy plik channels.py

```python
"""Notification channels for django-notifs."""

from json import dumps

import pika

from notifications.channels import BaseNotificationChannel


class BroadCastWebSocketChannel(BaseNotificationChannel):
    """Fanout notification channel with RabbitMQ."""

    def _connect(self):
        """Connect to the RabbitMQ server."""
        connection = pika.BlockingConnection(
            pika.ConnectionParameters(host='localhost')
        )
        channel = connection.channel()

        return connection, channel

    def construct_message(self):
        """Construct the message to be sent."""
        extra_data = self.notification_kwargs['extra_data']

        return dumps(extra_data['message'])

    def notify(self, message):
        """put the message of the RabbitMQ queue."""
        connection, channel = self._connect()

        uri = self.notification_kwargs['extra_data']['uri']

        channel.exchange_declare(exchange=uri, exchange_type='fanout')
        channel.basic_publish(exchange=uri, routing_key='', body=message)

        connection.close()
```

Musimy poinformować django-notifs o nowym kanale, który właśnie utworzyliśmy. Django-notifs używa Celery do asynchronicznego przetwarzania powiadomień, więc długo działające zadania powiadomień (jak wysyłanie e-maili) nie blokują żądania użytkownika.

```python
"""Celery init."""

from __future__ import absolute_import, unicode_literals
import os

from celery import Celery


# set the default Django settings module for the 'celery' program.
os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'chatire.settings')

app = Celery('chatire')
app.config_from_object('django.conf:settings', namespace='CELERY')

app.autodiscover_tasks()
```

Linia app.autodiscover_tasks() jest bardzo ważna. Automatycznie lokalizuje i importuje zadania celery zdefiniowane w django-notifs, bez niej musielibyśmy importować zadania ręcznie.

Tworzymy teraz nowy plik websocket.py

```python
import sys
import asyncio
import aio_pika
import websockets

async def keepalive(websocket):
    try:
        await websocket.recv()
    except websockets.ConnectionClosedError as error:
        print(error)
        sys.exit(1)  # Force the script to exit

async def consume(channel, queue_name, websocket):
    async with channel.consume(queue_name) as queue:
        async for message in queue:
            try:
                await websocket.send(message.body.decode())
                message.ack()
            except websockets.ConnectionClosedError as error:
                print(error)
                sys.exit(1)  # Force the script to exit

async def application(websocket, path):
    connection = await aio_pika.connect_robust("amqp://guest:guest@localhost/")
    async with connection:
        channel = await connection.channel()

        exchange = path.replace('/', '')
        await channel.set_qos(prefetch_count=1)
        await channel.declare_exchange(exchange, aio_pika.ExchangeType.FANOUT)

        queue = await channel.declare_queue(exclusive=True)
        await queue.bind(exchange)

        # Start the keepalive task
        asyncio.create_task(keepalive(websocket))

        # Consume messages
        await consume(channel, queue.name, websocket)

start_server = websockets.serve(application, 'localhost', 8000)

asyncio.get_event_loop().run_until_complete(start_server)
asyncio.get_event_loop().run_forever()
```

Ten kod używa bibliotek aio_pika i websockets do utworzenia serwera WebSocket, który konsumuje wiadomości z kolejki AMQP (RabbitMQ) i wysyła je do klienta WebSocket.

async def keepalive(websocket): Ta funkcja obsługuje zadanie keepalive, które polega na oczekiwaniu na wiadomość od klienta WebSocket. Jeśli połączenie WebSocket zostanie zamknięte, wypisuje błąd i zamyka skrypt.

async def consume(channel, queue_name, websocket): Ta funkcja konsumuje wiadomości z kolejki AMQP przy użyciu podanego kanału. Przesyła otrzymane wiadomości do klienta WebSocket po zdekodowaniu ich treści. Następnie potwierdza odebranie wiadomości (message.ack()). Jeśli połączenie WebSocket zostanie zamknięte, wypisuje błąd i zamyka skrypt.

async def application(websocket, path): Ta funkcja obsługuje aplikację WebSocket. Tworzy połączenie AMQP, kanał i deklaruje wymianę AMQP na podstawie ścieżki. Ustawia limit pobierania wiadomości na 1 (await channel.set_qos(prefetch_count=1)) i deklaruje kolejkę AMQP. Następnie wiąże kolejkę z wymianą i rozpoczyna zadanie keepalive, a następnie rozpoczyna konsumowanie wiadomości z kolejki AMQP przy użyciu funkcji consume().

start_server = websockets.serve(application, 'localhost', 8000): Tworzy serwer WebSocket, który obsługuje funkcję application() i nasłuchuje na porcie 8000 na lokalnym hoście.

Konfiguracja połączenia z bazą oraz migracje

Musimy upewnić, się, że połączenie z bazą jest ustawione w odpowiedni sposób.

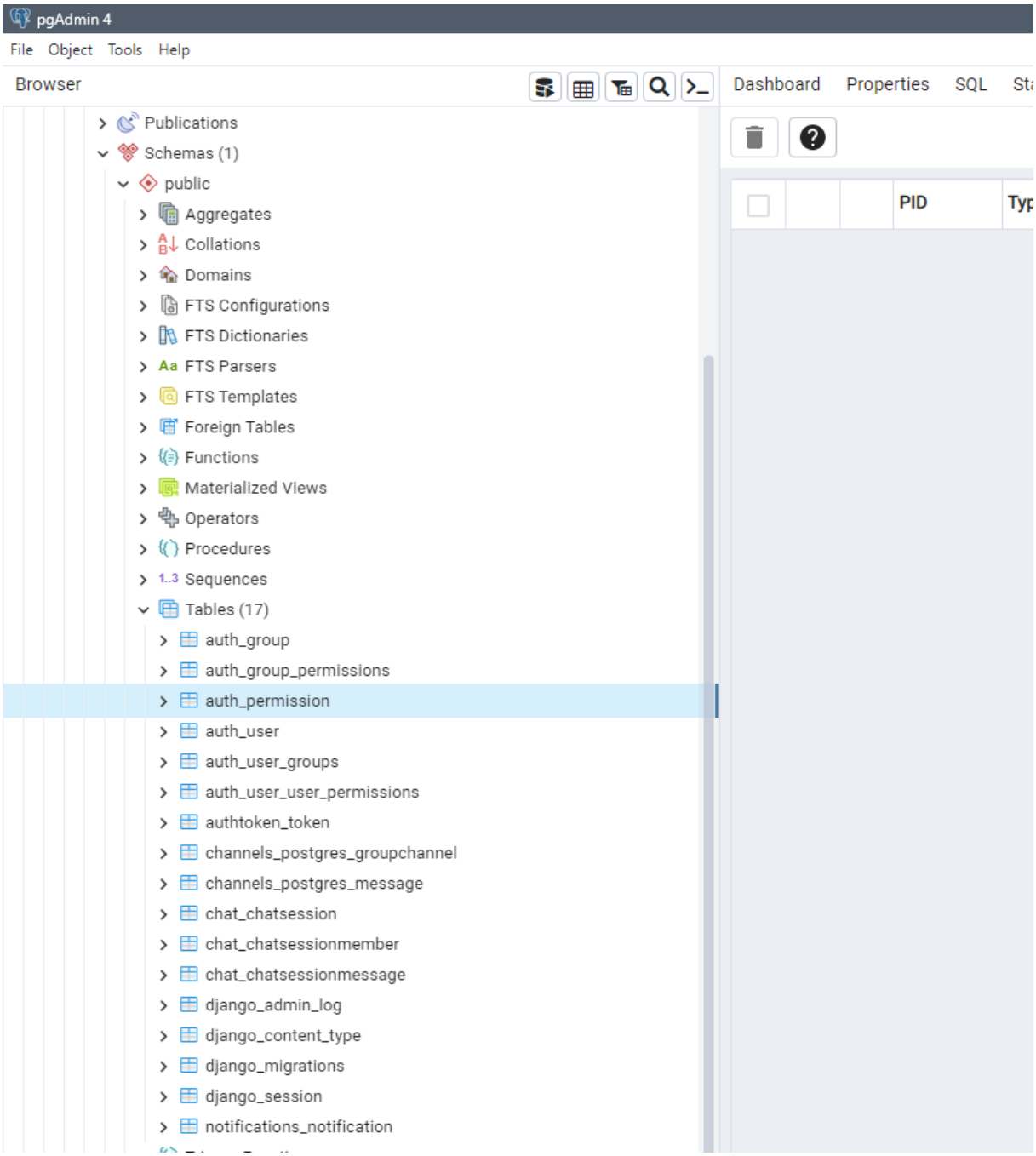
```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql_psycopg2',
        'NAME': 'postgres',
        'USER': 'postgres',
        'PASSWORD': 'admin',
        'HOST': '127.0.0.1',
        'PORT': '5432',
        },

        'channels_postgres': {
        'ENGINE': 'django.db.backends.postgresql_psycopg2',
        'NAME': 'postgres',
        'USER': 'postgres',
        'PASSWORD': 'admin',
        'HOST': '127.0.0.1',
        'PORT': '5432',
        },


}
```

```
# django-channels
ASGI_APPLICATION = 'chatire.asgi.application'
CHANNEL_LAYERS = {
    'default': {
        'BACKEND': 'channels_postgres.core.PostgresChannelLayer',
        'CONFIG': {
            'NAME': 'postgres',
            'USER': 'postgres',
            'PASSWORD': 'admin',
            'HOST': '127.0.0.1',
            'PORT': '5432',
        }
    }
}
```

Inicjalizacja konfiguracji bazy danych o nazwie 'channels_postgres' jest konieczna, ponieważ aplikacja Django korzysta z modułu Channels do obsługi komunikacji w czasie rzeczywistym (real-time communication). Moduł Channels oferuje możliwość obsługi WebSockets, protokołu asynchronicznego, protokołów obsługi kolejek wiadomości takich jak RabbitMQ.

Następnie przeprowadzamy migrację

python manage.py migrate



Utworzone zostało 17 tabel.

Podsumowanie

Projekt, w którym tworzyłem aplikację do czatowania z wykorzystaniem Django, Vue.js, PostgreSQL i RabbitMQ, był ekscytującym i ambitnym przedsięwzięciem, które pozwoliło mi zgłębić różne technologie i integrować je w celu stworzenia kompletnego rozwiązania do komunikacji w czasie rzeczywistym.

Celem projektu było zaprojektowanie i rozwinięcie aplikacji czatu, która umożliwiałaby użytkownikom prowadzenie rozmów tekstowych w czasie rzeczywistym. Do tego celu zdecydowano się na użycie Django jako backendu, Vue.js jako frameworku do budowania interfejsu użytkownika, PostgreSQL jako bazy danych i RabbitMQ jako systemu obsługi kolejek wiadomości.

W części backendowej projektu, Django było doskonałym wyborem. Zapewniało ono solidne i skalowalne fundamenty dla naszej aplikacji. Django zapewniało również wygodne mechanizmy uwierzytelniania i autoryzacji, co było niezwykle istotne dla bezpiecznego dostępu do czatu. PostgreSQL, jako system zarządzania bazą danych, zapewnił niezawodne i skalowalne rozwiązanie przechowywania danych czatu.

W części frontendowej projektu, Vue.js pozwoliło na tworzenie dynamicznego i interaktywnego interfejsu użytkownika. Z wykorzystaniem komponentów Vue.js, mogłem tworzyć modułowe elementy interfejsu, co znacznie ułatwiało rozwój i utrzymanie kodu. Dzięki mechanizmowi dwukierunkowego wiązania danych, użytkownicy mogli natychmiastowo widzieć i reagować na nowe wiadomości czatu.

Do obsługi komunikacji w czasie rzeczywistym między klientami, wybraliśmy RabbitMQ jako system obsługi kolejek wiadomości. Dzięki temu rozwiązaniu, mogliśmy zaimplementować mechanizm publikowania i subskrybowania wiadomości, umożliwiając klientom otrzymywanie aktualizacji czatu w czasie rzeczywistym.

Podsumowując, projekt aplikacji czatu opartej na Django, Vue.js, PostgreSQL i RabbitMQ był udanym przedsięwzięciem, które połączyło różne technologie w celu stworzenia zaawansowanego systemu komunikacji w czasie rzeczywistym. Pozwolił mi na rozwinięcie umiejętności w zakresie tworzenia aplikacji webowych, integracji różnych technologii i pracy w zespole. Dzięki temu projektowi zdobyłem cenne doświadczenie i umiejętności w obszarze tworzenia zaawansowanych aplikacji internetowych.