TBD CHATAPP

Members

1.1820222024 - 麦默德 - Aimal Khan

2.1820222043 - 巴戈 - Ahmad Wakil Babagana

3.1820222022 - 胡天 Mohammed Alhuteily

4.1820222033 - ☒卡 - Tilan

Our Decision:

We are working with a Django project that uses `django-tailwind` (a tool to integrate Tailwind CSS with Django), the concurrency strategy primarily relates to how Django handles incoming web requests and manages multiple tasks or users concurrently.

To determine the concurrency strategy used by a Django application, we had to consider how Django is deployed and focused on the code made for the Django-Tailwind integration, since `django-tailwind` is primarily concerned with styling (CSS) and front-end assets.

Our Concurrency Strategy:

We're using Django, which is a single-threaded and synchronous by default when running with its built-in development server (`python manage.py runserver`). For production, Django is usually deployed using a WSGI or ASGI server that determines the concurrency strategy.

Web Server Gateway Interface:

We're using ASGI and WSGI. WSGI servers handle concurrency by spawning multiple worker processes or threads, which handle requests concurrently but synchronously. ASGI (Asynchronous Server Gateway Interface) supports asynchronous request handling, allowing Django to handle multiple requests concurrently in an event-driven manner, which is beneficial for our application (**WebSockets**).

Some examples of an **asynchronous view**:

```
python

from django.http import JsonResponse
```

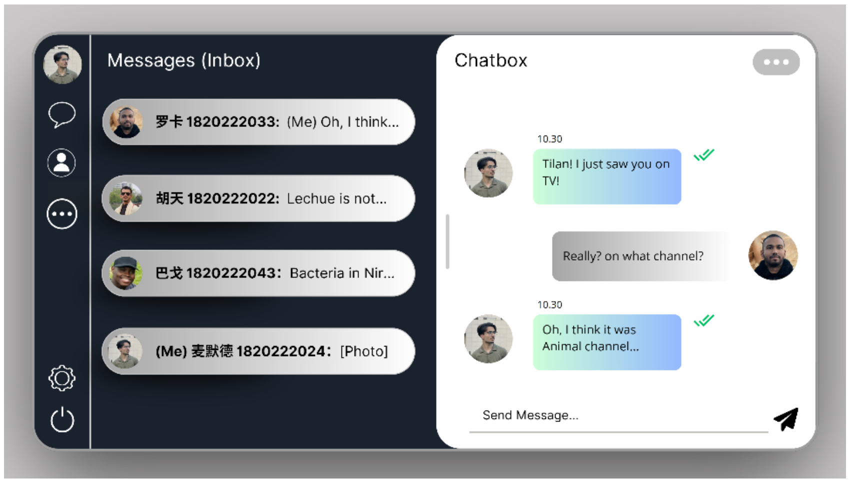
```
import asyncio


async def async_view(request):
    await asyncio.sleep(1)  # Simulate an I/O-bound operation
    return JsonResponse({"message": "This is an async view!"})
```

How to identify our Concurrency strategy: (dig deeper in our project!)

You can also find our concurrency strategy by looking for settings related to ASGI configuration or specific to your WSGI/ASGI server (e.g., `ASGI_APPLICATION` or `WSGI_APPLICATION`). Check the `settings.py` file for middleware that might affect concurrency, such as middleware handling sessions, caching, or real-time updates.

User Interface Sketches:

Our final product should look like this in the future. We're still vigorously working on the UI and trying to troubleshoot the Django-tailwind visibility errors with CSS:



罗卡 1820222033: (Me) Oh, I think...

胡天 1820222022: Lechue is not...

巴戈 1820222043: Bacteria in Nir...

(Me) 麦默德 1820222024: [Photo]

10.30 Tilan! I just saw you on TV!

Really? on what channel?

10.30 Oh, I think it was Animal channel...

Testing Strategy for Django-Tailwind Chat App:

1. Unit Testing for Views and Models:

- We used Django's built-in test framework to check that views and models work correctly.

Test Views:

python

```
  from django.test import TestCase, Client
  from django.urls import reverse
  class ChatViewTest(TestCase):
      def setUp(self):
          self.client = Client()
          self.user = User.objects.create_user(username='testuser', password='testpassword')
          self.client.login(username='testuser', password='testpassword')

      def test_chat_view_renders(self):
          response = self.client.get(reverse('chat_view'))
          self.assertEqual(response.status_code, 200)
```

Test Models:

```
python
  class MessageModelTest(TestCase):
      def test_message_creation(self):
          user = User.objects.create(username='user1')
          message = Message.objects.create(sender=user, content='Hello!')
          self.assertEqual(message.content, 'Hello!')
```

2. Integration Testing for Messaging: Test sending and receiving messages between two different users.

```
  python
  class MessagingTest(TestCase):
      def setUp(self):
          self.client1 = Client()
          self.client2 = Client()
          self.user1 = User.objects.create_user(username='user1', password='pass1')
          self.user2 = User.objects.create_user(username='user2', password='pass2')
          self.client1.login(username='user1', password='pass1')
```

```
        self.client2.login(username='user2', password='pass2')

    def test_send_message(self):
        response = self.client1.post(reverse('send_message'), {'content': 'Hello, user2!', 'receiver':
self.user2.id})
        self.assertEqual(response.status_code, 200)
        self.assertTrue(Message.objects.filter(content='Hello, user2!', sender=self.user1).exists())

    def test_receive_message(self):
        Message.objects.create(sender=self.user1, receiver=self.user2, content='Hello, user2!')
        response = self.client2.get(reverse('chat_view'))
        self.assertContains(response, 'Hello, user2!')
```

3. Static File Testing for Tailwind CSS: Make sure the Tailwind CSS is loaded properly:

python

```
class StaticFilesTest(TestCase):
    def test_static_css_loaded(self):
        response = self.client.get(reverse('chat_view'))
        self.assertContains(response, 'css/styles.css')
```

We will test messages by sending messages between two accounts, this way:

- We'll create two test accounts example: (`user1` and `user2`).
- Test sending a message from `user1` to `user2` and ensure it appears correctly in `user2`'s chat view.

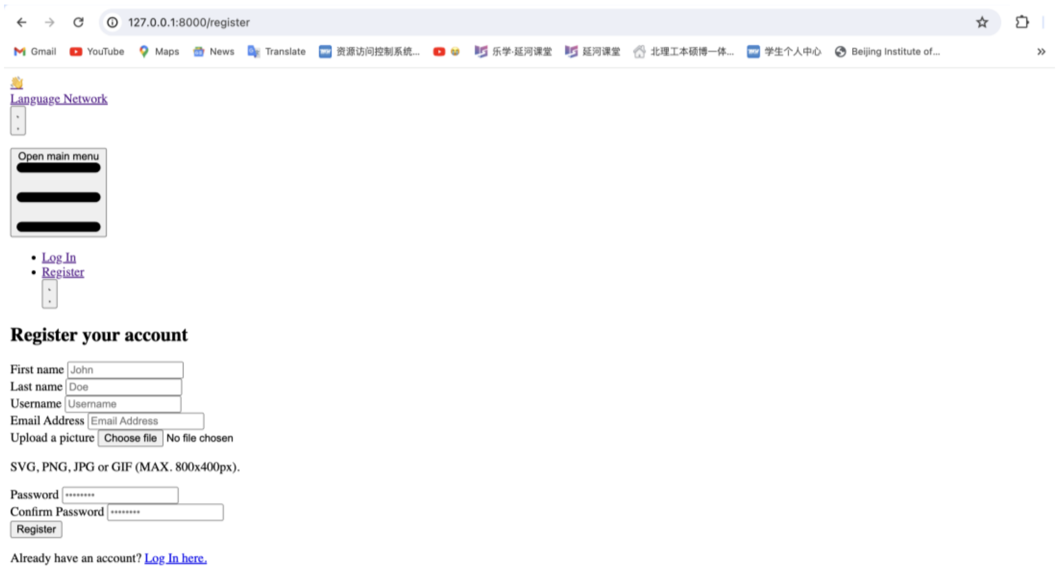
This approach focuses on the pure functionality of our APP, making sure messages are sent and received properly, and that the **Tailwind CSS** is correctly applied.

Critical or high-risk areas for a Django-Tailwind chat app: (Design-wise)

The critical risks for a Django-Tailwind chat app include issues with real-time messaging, such as delays, message loss, or failures in communication that could degrade user experience. Data integrity is another concern, where duplicate messages or incorrect ordering can disrupt the chat flow. Authentication and authorization are high-risk areas due to the potential for unauthorized access or data breaches. The security of user data is also at risk from vulnerabilities like cross-site scripting (XSS), cross-site request forgery (CSRF), and data breaches. Scalability and performance issues could arise when handling an increasing number of users or messages, leading to potential bottlenecks. Static file management is critical because improper loading of Tailwind CSS files could result in broken or inconsistent UI. Managing WebSocket connections poses a risk of connection drops or network failures that could interrupt real-time messaging. High traffic or multiple simultaneous users may overload the server, causing performance issues. Poor error logging and monitoring can result in slow detection and resolution of problems. Dependency management poses risks if third-party libraries introduce vulnerabilities or incompatibilities.

Demo

Login and Registration



127.0.0.1:8000/register

Gmail YouTube Maps News Translate 资源访问控制系统... 乐学-延河课堂 延河课堂 北理工本硕博一体... 学生个人中心 Beijing Institute of...

Language Network

Open main menu

- Log In
- Register

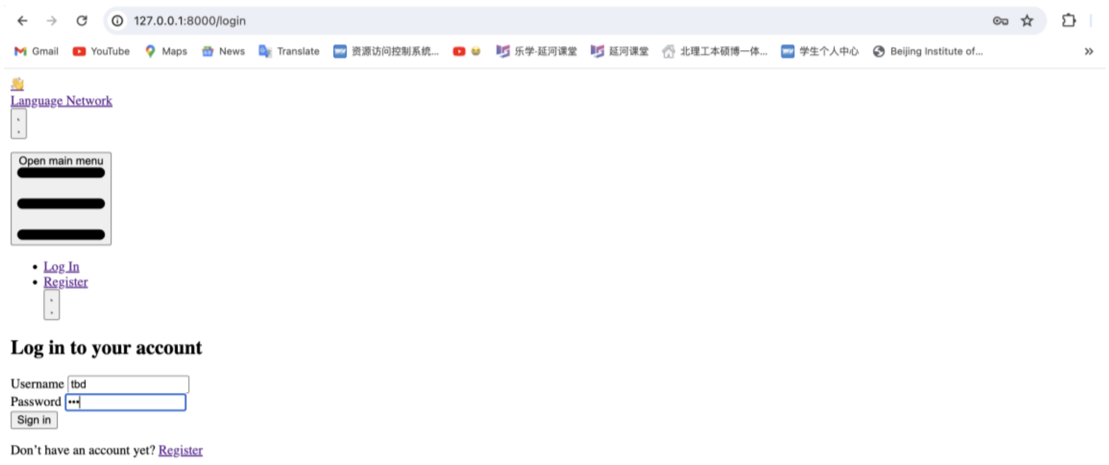
Register your account

First name John
Last name Doe
Username Username
Email Address Email Address
Upload a picture Choose file No file chosen

SVG, PNG, JPG or GIF (MAX. 800x400px).

Password ••••••••
Confirm Password ••••••••
Register

Already have an account? Log In here.



127.0.0.1:8000/login

Gmail YouTube Maps News Translate 资源访问控制系统... 乐学-延河课堂 延河课堂 北理工本硕博一体... 学生个人中心 Beijing Institute of...

Language Network

Open main menu

- Log In
- Register

Log in to your account

Username tbd
Password •••
Sign in

Don't have an account yet? Register