Este proyecto es una aplicación web para jugar al Pong en línea. Está compuesto por:

* **Backend:** Implementado en Django, con soporte asíncrono mediante Django Channels para manejar websockets. El backend utiliza PostgreSQL para la persistencia y Redis como capa de canal para la comunicación en tiempo real.
* **Frontend:** Desarrollado en React, usando por ejemplo react-router-dom para la navegación. Este se comunica con el backend tanto mediante peticiones HTTP (APIs REST) como por websockets para el juego en tiempo real.
* **Contenedorización:** Se utiliza Docker Compose para orquestar los contenedores (backend, frontend, base de datos y Redis).

**Estructura y Función de los Archivos del Proyecto**

**1. Docker Compose y Configuración del Proyecto**

* **docker-compose.yml**  
  Este archivo define los cuatro servicios principales:
  + **backend:** Construido a partir del directorio ./backend usando un Dockerfile propio. Se configura con variables de entorno para Django (como DJANGO\_SETTINGS\_MODULE, DATABASE\_URL, SECRET\_KEY, etc.), y se asegura de que dependa de la base de datos (PostgreSQL) y Redis.
  + **frontend:** Construido desde ./frontend, expone el puerto 3000 y utiliza la variable REACT\_APP\_BACKEND\_URL para apuntar al backend.
  + **db:** Servicio de PostgreSQL configurado con usuario, contraseña y nombre de base de datos.
  + **redis:** Servicio de Redis que sirve de backend para la capa de canales de Django.
* **package.json y package-lock.json**  
  Estos archivos forman parte del proyecto React y definen las dependencias del frontend, como por ejemplo react-router-dom. Se encargan de gestionar las versiones y dependencias necesarias para la parte del cliente.

**2. Configuración y Puesta en Marcha del Backend (Django)**

* **settings.py**  
  Ubicado en el directorio del proyecto Django, este archivo:
  + Configura las aplicaciones instaladas, incluyendo apps de terceros como channels (para websockets), corsheaders (para permitir peticiones desde el frontend) y rest\_framework (para APIs REST).
  + Define el modelo de usuario personalizado (AUTH\_USER\_MODEL).
  + Configura la base de datos para usar PostgreSQL, leyendo las variables de entorno definidas en Docker Compose.
  + Establece la capa de canales usando Redis y define ASGI\_APPLICATION para que se utilice el protocolo asíncrono.
  + Configura las rutas estáticas, middleware, y orígenes permitidos para CORS y CSRF.
* **asgi.py**  
  Este archivo es el punto de entrada para el servidor asíncrono de Django (ASGI). Sus funciones principales son:
  + Inicializar Django correctamente antes de cargar cualquier componente asíncrono.
  + Importar y combinar los patrones de URL para websockets desde las apps involucradas (por ejemplo, users.routing y pong.routing).
  + Configurar un ProtocolTypeRouter que diferencia entre tráfico HTTP (que se dirige a la aplicación tradicional de Django) y tráfico websocket (gestionado mediante AuthMiddlewareStack y URLRouter).
* **wsgi.py**  
  Es el punto de entrada tradicional para servidores WSGI (sin soporte asíncrono). Aunque en este proyecto se da preferencia a ASGI para websockets, este archivo sigue existiendo para compatibilidad con servidores que usen WSGI.
* **urls.py (nivel de proyecto)**  
  Aquí se definen las rutas HTTP principales de la aplicación:
  + La ruta para el panel de administración (admin/).
  + Endpoints para la API de usuarios (api/users/).
  + Un endpoint de prueba (api/test/) que devuelve un mensaje de confirmación.
  + La inclusión de las rutas para la app de Pong (api/pong/).
* **admin.py**  
  Archivo básico para registrar modelos en la interfaz de administración de Django. Actualmente es un esqueleto, listo para extenderse según se añadan modelos.

**3. Módulo de Pong (Juego y Websockets)**

Esta app se encarga de la lógica del juego, la gestión de salas y la comunicación en tiempo real mediante websockets.

* **apps.py (pong/apps.py)**  
  Configura la aplicación pong y define parámetros como el default\_auto\_field.
* **models.py (pong/models.py)**  
  Define el modelo **PongRoom**, que representa una sala de juego:
  + **Atributos principales:**
    - id: Un campo UUID que actúa como identificador único de la sala.
    - player1 y player2: Relaciones con el modelo de usuario para identificar a los jugadores.
    - created\_at: Fecha de creación de la sala.
  + **Métodos auxiliares:**
    - is\_full(): Comprueba si la sala tiene ambos jugadores.
    - add\_player(user): Añade un jugador a la sala, asignándolo a player1 o player2 según disponibilidad.
    - remove\_player(user): Elimina a un jugador y, si la sala queda vacía, la elimina del sistema.
* **views.py (pong/views.py)**  
  Contiene dos vistas protegidas por login\_required:
  + **create\_pong\_room:** Crea una nueva sala asignando al usuario como player1 y retorna el ID de la sala.
  + **join\_pong\_room:** Permite a un usuario unirse a una sala existente como player2, si la sala aún tiene un hueco disponible.
* **urls.py (pong/urls.py)**  
  Define las rutas HTTP específicas para la app de Pong:
  + create/ para crear una nueva sala.
  + join/<uuid:room\_id>/ para unirse a una sala existente mediante su identificador.
* **consumers.py (pong/consumers.py)**  
  Aquí se implementa el corazón de la comunicación en tiempo real mediante websockets:
  + **PongGameConsumer:**
    - **Conexión (connect):**
      * Extrae el room\_id de la URL del websocket.
      * Verifica que el usuario esté autenticado. Si no lo está, cierra la conexión.
      * Intenta obtener (o crear) la sala de juego (PongRoom) y añade al usuario a la sala usando el método add\_player.
      * Añade la conexión a un grupo (channel group) identificado por el patrón "pong\_{room\_id}", lo que permite enviar mensajes a todos los participantes de la sala.
      * Acepta la conexión y envía una actualización del estado de la sala (por ejemplo, quiénes están conectados).
    - **Desconexión (disconnect):**
      * Al desconectar, elimina al usuario de la sala y del grupo, y envía una actualización para reflejar los cambios.
    - **Recepción de mensajes (receive):**
      * Procesa mensajes entrantes en formato JSON.
      * Si el mensaje es de tipo "move\_paddle", se invoca update\_paddle\_position para actualizar la posición de la paleta del jugador.
      * Si el mensaje es de tipo "game\_update", se retransmite el mensaje a todos los miembros del grupo.
    - **Actualización de la paleta (update\_paddle\_position y update\_paddle):**
      * Calcula la nueva posición de la paleta, asegurando que se mantenga en un rango válido.
      * Utiliza group\_send para notificar a todos los clientes conectados sobre la actualización.
    - **Actualización del estado de la sala (send\_room\_update y room\_update):**
      * Envía información sobre los jugadores conectados en la sala, permitiendo que el frontend sincronice el estado (por ejemplo, mostrando quién es player1 y quién es player2).
* **routing.py (pong/routing.py)**  
  Define la ruta websocket específica para la app de Pong:
  + Usa una expresión regular para capturar el room\_id en la URL (por ejemplo, ws/pong/<room\_id>/) y lo asigna al consumidor PongGameConsumer.
* **tests.py (pong/tests.py)**  
  Archivo destinado a pruebas automatizadas para la app de Pong. Actualmente es un esqueleto, listo para que se añadan tests conforme se desarrolle más funcionalidad.

**4. Módulo de Usuarios**

Aunque en los archivos subidos no se muestra todo el contenido de la app de usuarios, se puede deducir lo siguiente:

* Se define un modelo de usuario personalizado (referenciado en settings.py con AUTH\_USER\_MODEL = 'users.CustomUser'), lo que permite extender la funcionalidad del usuario estándar de Django.
* **Websockets en Usuarios:**  
  En el archivo de rutas de websockets (por ejemplo, en el routing a nivel de proyecto), se importa el OnlineUsersConsumer desde users.consumers. Este consumidor probablemente se encarga de gestionar la presencia en línea de los usuarios (por ejemplo, mostrando qué usuarios están conectados en tiempo real).

**Funcionamiento de los Websockets**

El uso de websockets es fundamental para el juego en tiempo real. A continuación, se describe cómo se implementa y funciona este mecanismo:

1. **Configuración ASGI y Enrutamiento:**
   * En **asgi.py**, se inicializa el entorno de Django y se combinan los patrones de websocket de ambas apps (usuarios y pong).
   * Se utiliza ProtocolTypeRouter para distinguir entre solicitudes HTTP y websocket. Las solicitudes websocket se envuelven con AuthMiddlewareStack, lo que permite acceder a la información del usuario autenticado.
2. **Conexión del Cliente:**
   * Cuando un cliente (desde el frontend React) se conecta a una URL de websocket (por ejemplo, ws://<dominio>/ws/pong/<room\_id>/), se invoca el método connect del consumidor correspondiente.
   * El consumidor verifica la autenticación, gestiona la asignación del usuario a la sala (creando o uniéndose a ella) y añade la conexión a un "grupo" mediante el channel layer. Este grupo representa una sala de juego y permite enviar mensajes a todos los miembros simultáneamente.
3. **Recepción y Procesamiento de Mensajes:**
   * Los mensajes enviados desde el cliente se reciben en el método receive. Estos mensajes, en formato JSON, especifican el tipo de acción:
     + **move\_paddle:** Indica que un jugador ha movido su paleta. Se calcula la nueva posición y se notifica a todos los jugadores del grupo.
     + **game\_update:** Permite enviar actualizaciones generales del estado del juego (por ejemplo, el movimiento de la pelota u otros eventos).
   * El uso de group\_send permite que cualquier mensaje se retransmita a todos los clientes conectados en la misma sala, manteniendo la sincronización en tiempo real.
4. **Desconexión y Actualización del Estado:**
   * Cuando un cliente se desconecta, se elimina del grupo y se actualiza el estado de la sala (por ejemplo, notificando a los otros jugadores que alguien ha abandonado el juego). Si la sala queda vacía, se elimina del sistema.

Este mecanismo, basado en Django Channels y Redis, permite un flujo de datos bidireccional y en tiempo real, ideal para la interactividad necesaria en un juego como Pong.

Los websockets son una tecnología que permite establecer una conexión persistente y bidireccional entre el cliente (por ejemplo, un navegador) y el servidor, lo que resulta ideal para aplicaciones en tiempo real, como un juego de Pong. A continuación, se detalla de forma exhaustiva cómo funcionan:

**1. Concepto Básico y Diferencias con HTTP**

* **HTTP vs. WebSocket:**  
  En una comunicación HTTP tradicional, el cliente realiza una solicitud y el servidor responde de manera aislada; cada petición es independiente y la conexión se cierra una vez completada la respuesta. En cambio, los websockets mantienen una única conexión abierta durante la sesión, permitiendo el envío de datos en cualquier dirección en cualquier momento sin tener que reestablecer la conexión.
* **Conexión Persistente:**  
  Una vez establecida, la conexión websocket se mantiene abierta, permitiendo la comunicación en tiempo real. Esto es fundamental para aplicaciones interactivas y colaborativas donde la latencia debe ser mínima.

**2. El Proceso de Conexión**

* **Handshake Inicial:**  
  La conexión websocket se inicia mediante un proceso de "handshake" en el que el cliente envía una petición HTTP especial (con cabeceras específicas, como Upgrade: websocket y Connection: Upgrade) para solicitar la actualización del protocolo. Si el servidor soporta websockets, responde aceptando la solicitud y la conexión se transforma en una conexión websocket persistente.
* **Establecimiento de la Conexión:**  
  Una vez que el handshake se completa, el canal de comunicación queda abierto y tanto el cliente como el servidor pueden enviar y recibir mensajes sin la sobrecarga de múltiples peticiones HTTP.

**3. Implementación en Django con Channels**

En el proyecto, Django Channels es el framework que permite trabajar con websockets de manera nativa en Django. Veamos cómo se integra:

* **Archivo asgi.py:**  
  Este archivo configura el entorno asíncrono y define un ProtocolTypeRouter, que es el encargado de dirigir las peticiones según su tipo:
  + **HTTP:** Se enrutan a la aplicación tradicional de Django.
  + **WebSocket:** Se gestionan a través de un URLRouter envuelto en AuthMiddlewareStack para poder acceder a la información del usuario autenticado. Esto significa que cada conexión websocket tiene un scope que incluye datos de autenticación, lo que es útil para saber qué usuario está conectado.
* **Enrutamiento de Websockets (routing.py):**  
  Se definen las rutas específicas para websockets. Por ejemplo, para el juego de Pong se utiliza una URL que incluye un identificador de sala (room\_id), lo que permite distinguir entre diferentes partidas y agrupar conexiones en "channel groups".
* **Consumidor (Consumer):**  
  El PongGameConsumer es el componente que gestiona la comunicación en tiempo real. Su funcionamiento se puede dividir en varios métodos clave:
  + **connect():**
    - Se extrae el room\_id de la URL y se forma un identificador de grupo (por ejemplo, "pong\_<room\_id>").
    - Se verifica que el usuario esté autenticado. Si no lo está, se cierra la conexión.
    - Se obtiene (o crea) la sala de juego correspondiente y se añade el usuario a la misma.
    - Se añade el canal (la conexión websocket) a un grupo usando el channel layer, lo que permite enviar mensajes a todos los miembros de esa sala.
    - Se acepta la conexión y se envía una actualización del estado de la sala (como quiénes están conectados).
  + **receive():**
    - Cada mensaje recibido es en formato JSON. Según el tipo de mensaje (por ejemplo, "move\_paddle" o "game\_update"), se invoca la función correspondiente.
    - Para movimientos de paleta, se calcula la nueva posición y se envía una actualización a todo el grupo.
    - Los mensajes se retransmiten a todos los usuarios conectados mediante group\_send, lo que garantiza la sincronización en tiempo real.
  + **disconnect():**
    - Cuando se cierra la conexión, se elimina al usuario del grupo y, en algunos casos, se actualiza el estado de la sala (por ejemplo, removiendo al jugador y, si la sala queda vacía, eliminándola).
* **Uso del Channel Layer con Redis:**  
  Django Channels utiliza una capa de canales (channel layer) para gestionar la mensajería entre procesos. En este caso, se utiliza Redis, que actúa como un intermediario para almacenar y distribuir los mensajes entre los diferentes consumidores conectados. Esto permite que, por ejemplo, si dos jugadores están en la misma sala, un mensaje enviado por uno se retransmita instantáneamente al otro.

**4. Beneficios de Usar Websockets en Aplicaciones en Tiempo Real**

* **Baja Latencia:**  
  Al mantener la conexión abierta, se evitan los retrasos que supone abrir y cerrar conexiones repetidamente. Esto es esencial para juegos donde cada milisegundo cuenta.
* **Comunicación Bidireccional:**  
  Tanto el servidor como el cliente pueden enviar mensajes en cualquier momento, lo que permite actualizaciones en tiempo real de eventos (por ejemplo, el movimiento de las paletas, cambios en el estado del juego, etc.).
* **Escalabilidad:**  
  Con el uso de channel layers y Redis, la aplicación puede escalar y gestionar múltiples conexiones simultáneas sin problemas.

**5. Flujo de Datos en un Escenario de Juego de Pong**

1. **Inicio de la Conexión:**
   * El jugador inicia el juego y se conecta a la URL del websocket (por ejemplo, ws://<dominio>/ws/pong/<room\_id>/).
   * Se realiza el handshake y se establece la conexión, añadiendo al jugador al grupo de la sala.
2. **Interacción Durante el Juego:**
   * Cada vez que un jugador mueve la paleta, el cliente envía un mensaje JSON indicando el movimiento.
   * El consumidor recibe el mensaje, calcula la nueva posición y utiliza group\_send para enviar la actualización a todos los participantes de la sala.
3. **Actualizaciones y Sincronización:**
   * Cada cliente recibe el mensaje y actualiza la posición de la paleta en su interfaz.
   * Si se envían otros tipos de actualizaciones (por ejemplo, estado del juego, puntuaciones, etc.), se manejan de forma similar, asegurando que todos los jugadores vean la misma información en tiempo real.
4. **Cierre de la Conexión:**
   * Cuando un jugador abandona la sala, se elimina del grupo y se actualiza el estado de la sala, lo que puede implicar notificar al otro jugador o eliminar la sala si está vacía.