Documentación Técnica - Tic-Tac-Toe Multiplayer

Proyecto Capstone - Redes de Computadoras 2

Tabla de Contenidos

Introducción

Este proyecto implementa un juego de **Tic-Tac-Toe multijugador en red** utilizando arquitectura cliente-servidor, comunicación en tiempo real mediante WebSockets, y protocolos de seguridad modernos. El sistema permite que dos jugadores se conecten simultáneamente y mantengan un estado coherente del juego.

Tecnologías Utilizadas

- **Backend:** Python 3.12, FastAPI, Socket.IO, SQLAlchemy (Async)
- **Frontend:** HTML5, CSS3, JavaScript (Vanilla), Socket.IO Client
- **Base de Datos:** SQLite (producción puede usar PostgreSQL)
- **Protocolo de Comunicación:** HTTP/HTTPS (REST), WebSocket (tiempo real)
- **Autenticación:** JWT (JSON Web Tokens)

Fundamentos Teóricos

1. HTTP vs HTTPS

HTTP (HyperText Transfer Protocol)

- **Capa OSI:** Capa de Aplicación (Capa 7)
- **Puerto:** 80
- **Características:**
  - Protocolo sin estado (stateless)
  - Comunicación en texto plano
  - Modelo Request-Response
  - **Vulnerable** a ataques Man-in-the-Middle (MITM)

HTTPS (HTTP Secure)

- **Capa OSI:** Capa de Aplicación (Capa 7) con TLS/SSL en Capa de Presentación
- **Puerto:** 443
- **Características:**
  - HTTP + TLS/SSL (Transport Layer Security)
  - Encriptación de datos en tránsito
  - Autenticación del servidor mediante certificados
  - Integridad de datos (previene modificación)

En nuestro proyecto:

- En desarrollo usamos HTTP (localhost)
- En producción se debe usar HTTPS con certificados SSL/TLS
- La API REST usa HTTP/HTTPS para autenticación y consultas

```
Cliente                          Servidor
   |                                |
   |--- HTTP GET /api/stats ------->|
   |                                | [Procesa]
   |<-- HTTP 200 OK + JSON --------|
```

2. WebSocket

¿Qué es WebSocket?

- **Protocolo:** RFC 6455
- **Capa OSI:** Capa de Aplicación (Capa 7)
- **Puerto:** 80 (ws://), 443 (wss://)
- **Características:**
  - Comunicación **bidireccional full-duplex**
  - Conexión **persistente** (no se cierra después de cada mensaje)
  - **Baja latencia** ideal para tiempo real
  - Upgrade desde HTTP mediante handshake

Diferencia con HTTP

```
HTTP (Request-Response):
Cliente → Request → Servidor
Cliente ← Response ← Servidor
[Conexión cerrada]

WebSocket (Bidireccional):
Cliente ↔ Servidor
[Conexión persistente, mensajes en ambas direcciones]
```

En nuestro proyecto:

- WebSocket se usa para eventos de juego en tiempo real
- Socket.IO (biblioteca) simplifica WebSocket con fallbacks
- Eventos: `move_made`, `game_started`, `invitation_received`

Código de conexión (Frontend):

```
// frontend/js/lobby/lobby.js:43-48
socket = io(CONFIG.SOCKET_URL, {
    transports: ["websocket", "polling"],
    auth: { token: token }
});
```

3. JWT (JSON Web Tokens)

Estructura de JWT

Un JWT tiene 3 partes separadas por puntos:

```
[Header].[Payload].[Signature]
```

Ejemplo real de nuestro proyecto:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxIiwidXNlcm5hbWUiOiJhbGljZSIsImV4c
CI6MTcwOTMxMjQwMCwiaWF0IjoxNzA5MzA4ODAwfQ.signature
```

Decodificado:

1. **Header:**

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

2. **Payload:**

```
{
  "sub": "1",           // User ID
  "username": "alice",
  "exp": 1709312400,    // Expiration timestamp
```

```
  "iat": 1709308800     // Issued at timestamp
}
```

3. **Signature:** Hash criptográfico usando SECRET_KEY

Ventajas de JWT

- **Stateless:** El servidor no necesita almacenar sesiones
- **Auto-contenido:** Toda la información está en el token
- **Portable:** Se puede enviar en headers, cookies, o URL
- **Seguro:** Firma criptográfica previene manipulación

Implementación en nuestro proyecto:

```
# backend/app/auth/auth.py:20-40
def create_access_token(data: Dict, expires_delta: Optional[timedelta] = None) ->
str:
    to_encode = data.copy()
    now = datetime.now(timezone.utc)

    if expires_delta:
        expire = now + expires_delta
    else:
        expire = now + timedelta(minutes=settings.ACCESS_TOKEN_EXPIRE_MINUTES)

    to_encode.update({
        "exp": int(expire.timestamp()),
        "iat": int(now.timestamp())
    })

    encoded_jwt = jwt.encode(to_encode, settings.SECRET_KEY,
algorithm=settings.ALGORITHM)
    return encoded_jwt
```

Flujo de autenticación:

```
1. Cliente envía username/password → POST /api/login
2. Servidor verifica credenciales
3. Servidor genera JWT con user_id
4. Cliente recibe JWT y lo almacena (localStorage)
5. Cliente envía JWT en cada request: Authorization: Bearer <token>
6. Servidor valida JWT y extrae user_id
```

4. JSON y Serialización

¿Qué es JSON?

JSON (JavaScript Object Notation) es un formato de intercambio de datos ligero y legible:

```
{
  "game_id": 123,
  "player1": {
    "id": 1,
    "username": "alice",
    "symbol": "X"
  },
  "board": "---------",
  "current_turn": 1
}
```

Serialización y Deserialización

Serialización: Convertir objetos de Python/JavaScript a JSON (string) **Deserialización:** Convertir JSON (string) a objetos de Python/JavaScript

Ejemplo en Python (Backend):

```
# Serialización: Python → JSON
game_data = {
    'game_id': game.id,
    'player1': {'id': player1.id, 'username': player1.username},
    'board': game.board_state
}
# Socket.IO automáticamente serializa a JSON
await sio.emit('game_started', game_data, room=sid)
```

Ejemplo en JavaScript (Frontend):

```
// Deserialización: JSON → JavaScript
socket.on('game_started', (data) => {
    // data ya es un objeto JavaScript
    console.log(data.game_id);        // 123
    console.log(data.player1.username); // "alice"
});

// Serialización: JavaScript → JSON
socket.emit('make_move', {
    game_id: 123,
    position: 4
});
```

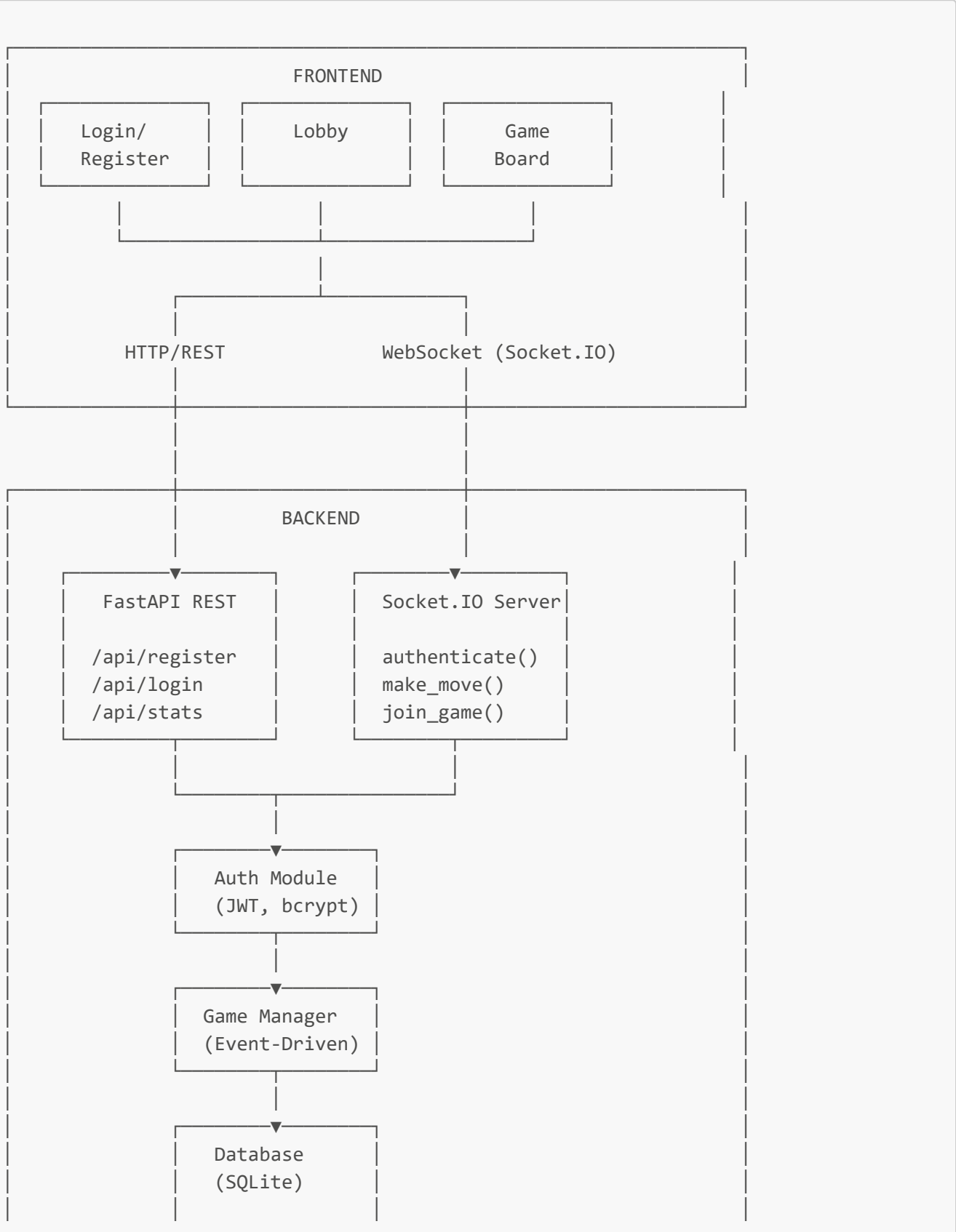
Ventajas de JSON:

- Ligero y eficiente

- Soportado nativamente en JavaScript
- Fácil de parsear en cualquier lenguaje
- Legible por humanos

Arquitectura del Sistema

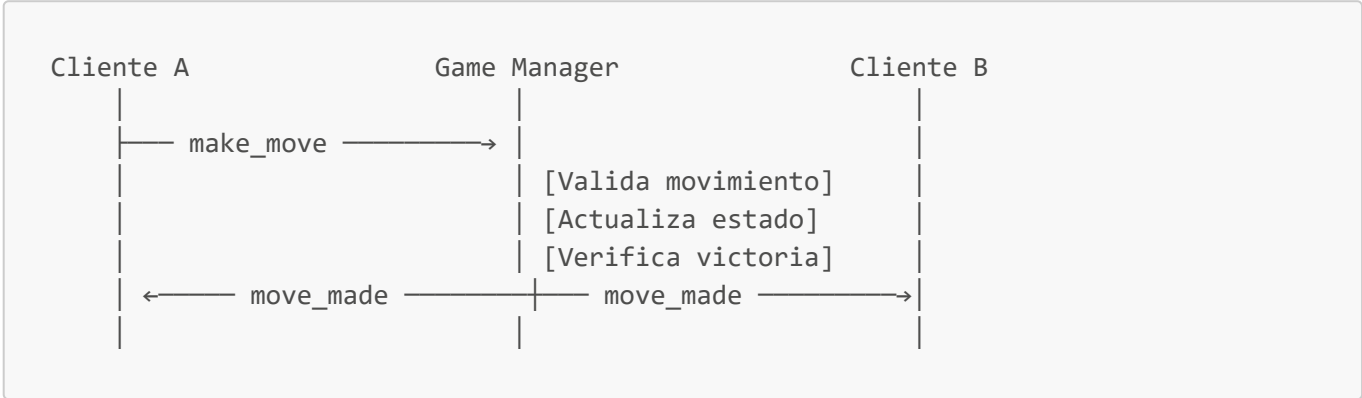
Diagrama de Componentes



```
│              │  - Users         │                                  │
│              │  - Games         │                                  │
│              │  - Stats         │                                  │
│              │  - Invitations   │                                  │
│              └──────────────────┘                                  │
└────────────────────────────────────────────────────────────────────┘
```

Arquitectura Orientada a Eventos (Event-Driven)

El servidor utiliza un **Event Bus** mediante Socket.IO para manejar eventos del juego:



Eventos principales:

1. **Lobby:**
   - `online_users`: Lista de usuarios conectados
   - `invitation_received`: Nueva invitación recibida
   - `game_started`: Partida iniciada
2. **Juego:**
   - `join_game`: Unirse a sala de juego
   - `make_move`: Realizar movimiento
   - `move_made`: Movimiento realizado (broadcast)
   - `game_forfeited`: Jugador abandonó

Implementación del Event Bus (backend/app/websocket/game_events.py):

```python
@sio.event
async def make_move(sid, data):
    # 1. Validar autenticación
    user_id = get_user_from_sid(sid)

    # 2. Procesar evento
    result = await game_manager.make_move(
        game_id=data['game_id'],
        user_id=user_id,
        position=data['position'],
        db=db
    )
```

```
    # 3. Broadcast a todos los jugadores
    game_room = f"game_{game_id}"
    await sio.emit('move_made', result, room=game_room)
```

Comunicación Frontend-Backend

1. Comunicación HTTP/REST (Autenticación y Consultas)

Endpoints REST Disponibles

Método	Endpoint	Descripción	Autenticación
POST	`/api/register`	Registrar nuevo usuario	No
POST	`/api/login`	Iniciar sesión	No
GET	`/api/users/me`	Obtener info del usuario actual	Sí (JWT)
GET	`/api/stats`	Obtener estadísticas del usuario	Sí (JWT)
GET	`/api/stats/leaderboard`	Obtener tabla de clasificación	Sí (JWT)
GET	`/api/games/history`	Obtener historial de partidas	Sí (JWT)

Ejemplo: Flujo de Registro

Frontend (frontend/js/auth/register.js):

```
async function register(username, password) {
    const response = await fetch(`${CONFIG.API_URL}/api/register`, {
        method: 'POST',
        headers: {
            'Content-Type': 'application/json'
        },
        body: JSON.stringify({
            username: username,
            password: password
        })
    });

    const data = await response.json();

    // Almacenar token JWT
    Storage.setToken(data.access_token);
    Storage.setUserInfo(data.user_id, data.username);
}
```

Backend (backend/app/server.py:111-176):

```python
@app.post("/api/register", response_model=LoginResponse)
async def register(request: RegisterRequest, db: AsyncSession = Depends(get_db)):
    # 1. Validar datos de entrada
    valid, error = validate_username(request.username)
    if not valid:
        raise HTTPException(status_code=400, detail=error)

    # 2. Verificar que el usuario no existe
    result = await db.execute(select(User).where(User.username ==
request.username))
    if result.scalar_one_or_none():
        raise HTTPException(status_code=400, detail="Username already taken")

    # 3. Hash de contraseña (bcrypt)
    hashed_password = hash_password(request.password)

    # 4. Crear usuario en DB
    new_user = User(username=request.username, password_hash=hashed_password)
    db.add(new_user)
    await db.commit()

    # 5. Generar JWT
    access_token = create_access_token({
        "sub": str(new_user.id),
        "username": new_user.username
    })

    # 6. Retornar token
    return LoginResponse(
        access_token=access_token,
        token_type="bearer",
        user_id=new_user.id,
        username=new_user.username
    )
```

Formato de Request:

```
POST /api/register
Content-Type: application/json

{
  "username": "alice",
  "password": "securePassword123"
}
```

Formato de Response:

```
HTTP/1.1 200 OK
Content-Type: application/json

{
  "access_token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...",
  "token_type": "bearer",
  "user_id": 1,
  "username": "alice"
}
```

2. Comunicación WebSocket (Tiempo Real)

Conexión y Autenticación WebSocket

Secuencia de conexión:

```
1. Cliente se conecta al servidor WebSocket
2. Servidor emite evento 'connect'
3. Cliente envía evento 'authenticate' con JWT
4. Servidor valida JWT
5. Servidor marca usuario como online
6. Servidor emite 'authenticated' (éxito)
7. Servidor broadcast 'online_users' a todos
```

Frontend (frontend/js/lobby/lobby.js:31-57):

```
function initSocket() {
    const token = Storage.getToken();

    // 1. Conectar a Socket.IO
    socket = io(CONFIG.SOCKET_URL, {
        transports: ["websocket", "polling"],
        auth: { token: token }
    });

    // 2. Al conectar, autenticar
    socket.on("connect", () => {
        socket.emit("authenticate", { token: token });
    });

    // 3. Escuchar confirmación
    socket.on("authenticated", (data) => {
        console.log("Autenticado:", data.username);
        loadLeaderboard();
    });

    // 4. Escuchar usuarios online
    socket.on("online_users", (data) => {
```

```
        updateOnlineUsers(data.users);
    });

    // 5. Escuchar invitaciones
    socket.on("invitation_received", (data) => {
        handleInvitationReceived(data);
    });
}
```

Backend (backend/app/server.py:387-443):

```python
@sio.event
async def authenticate(sid, data):
    token = data.get('token')

    # 1. Verificar JWT
    payload = verify_token(token)
    if not payload:
        await sio.emit('error', {'message': 'Invalid token'}, room=sid)
        return

    user_id = int(payload.get('sub'))

    # 2. Buscar usuario en DB
    async with AsyncSessionLocal() as db:
        result = await db.execute(select(User).where(User.id == user_id))
        user = result.scalar_one_or_none()

        # 3. Marcar como online
        user.is_online = True
        user.socket_id = sid
        await db.commit()

        # 4. Guardar conexión activa
        active_connections[str(user_id)] = sid

        # 5. Confirmar autenticación
        await sio.emit('authenticated', {
            'user_id': user.id,
            'username': user.username
        }, room=sid)

        # 6. Broadcast usuarios online
        await broadcast_online_users()
```

Flujo de Invitación y Inicio de Partida

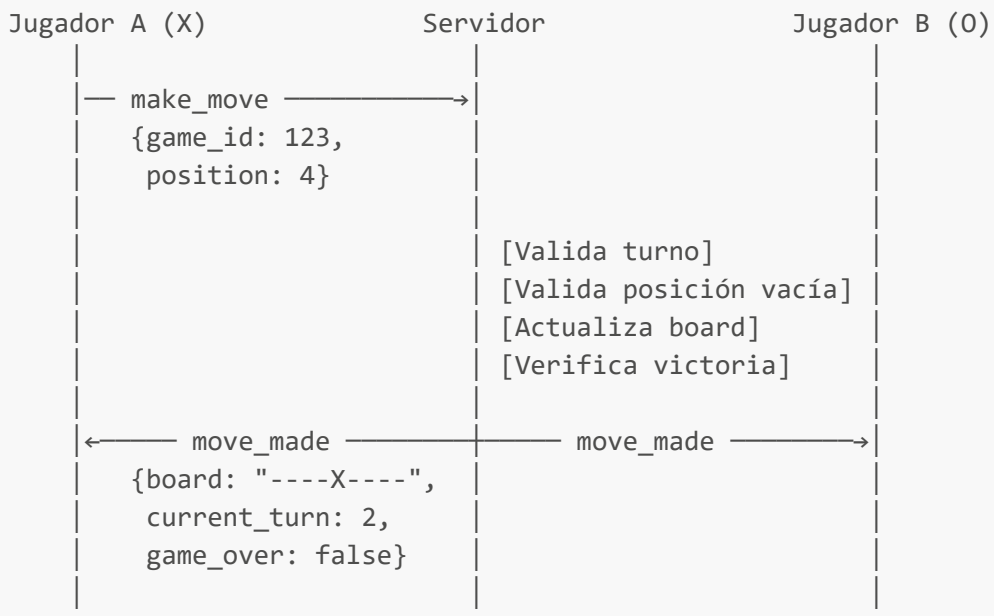
```
Jugador A                  Servidor                  Jugador B
    |                         |                         |
```

Código de invitación (backend/app/websocket/game_events.py:129-212):

```
@sio.event
async def invite_player(sid, data):
    sender_id = get_user_from_connections(sid)
    target_user_id = data.get('target_user_id')

    async with AsyncSessionLocal() as db:
        # 1. Verificar que ambos usuarios existen
        sender = await db.get(User, sender_id)
        target = await db.get(User, target_user_id)

        # 2. Verificar que target está online
        if not target.is_online:
            await sio.emit('error', {'message': 'User is offline'}, room=sid)
            return

        # 3. Crear invitación en DB
        invitation = Invitation(
            from_user_id=sender_id,
            to_user_id=target_user_id,
            status='pending'
        )
        db.add(invitation)
        await db.commit()

        # 4. Notificar al target
        target_sid = active_connections.get(str(target_user_id))
        await sio.emit('invitation_received', {
            'invitation_id': invitation.id,
```

```
            'from_user_id': sender_id,
            'from_username': sender.username
        }, room=target_sid)

        # 5. Confirmar al sender
        await sio.emit('invitation_sent', {
            'invitation_id': invitation.id,
            'to_username': target.username
        }, room=sid)
```

Flujo de Movimientos en el Juego



Frontend envía movimiento (frontend/js/game/game.js:162-179):

```
function makeMove(position) {
    // 1. Verificar que es mi turno
    if (currentTurn != userInfo.userId) {
        Notification.error("Not your turn!");
        return;
    }

    // 2. Verificar que la celda está vacía
    if (currentBoard[position] !== "-") {
        Notification.error("Cell already taken!");
        return;
    }

    // 3. Enviar movimiento al servidor
    socket.emit("make_move", {
        game_id: gameData.game_id,
        position: position
```

```
    });
}
```

Backend procesa movimiento (backend/app/websocket/game_events.py:371-464):

```python
@sio.event
async def make_move(sid, data):
    user_id = get_user_from_sid(sid)
    game_id = data.get('game_id')
    position = data.get('position')

    # 1. Validar posición (0-8)
    valid, error = validate_move(position)
    if not valid:
        await sio.emit('error', {'message': error}, room=sid)
        return

    async with AsyncSessionLocal() as db:
        # 2. Procesar movimiento (game_manager)
        result = await game_manager.make_move(game_id, user_id, position, db)
        # result = {
        #     'board': "----X----",
        #     'current_turn': 2,
        #     'game_over': False,
        #     'result': None,
        #     'winner_id': None
        # }

        # 3. Preparar datos para broadcast
        move_data = {
            'game_id': game_id,
            'position': position,
            'board': result['board'],
            'current_turn': result['current_turn'],
            'game_over': result['game_over']
        }

        if result['game_over']:
            move_data['result'] = result['result']
            move_data['winner_id'] = result['winner_id']
            move_data['winning_line'] = result['winning_line']

        # 4. Broadcast a AMBOS jugadores (room)
        game_room = f"game_{game_id}"
        await sio.emit('move_made', move_data, room=game_room)
```

Frontend recibe movimiento (frontend/js/game/game.js:181-192):

```
socket.on("move_made", (data) => {
    // 1. Actualizar tablero visual
    updateBoard(data.board);

    // 2. Si el juego continúa, actualizar turno
    if (!data.game_over) {
        updateTurn(data.current_turn);
    }

    // 3. Si el juego terminó, mostrar resultado
    if (data.game_over) {
        gameOver = true;
        handleGameOver(data);
    }
});
```

Distribución de Carga de Trabajo

Responsabilidades del Frontend

Ubicación: `frontend/` directory

1. Interfaz de Usuario (UI/UX)

Archivos:

- `frontend/public/*.html` - Estructura HTML
- `frontend/css/style.css` - Estilos

Responsabilidades:

- Renderizar páginas (login, lobby, game)
- Mostrar tablero de juego interactivo
- Mostrar usuarios online
- Mostrar invitaciones pendientes
- Feedback visual (animaciones, notificaciones)

2. Validación del Lado del Cliente

Archivos:

- `frontend/js/auth/login.js`
- `frontend/js/auth/register.js`

Validaciones:

- Username: mínimo 3 caracteres, máximo 20
- Password: mínimo 6 caracteres
- Campos requeridos no vacíos

- Formato de email (opcional)

Ejemplo:

```
if (username.length < 3) {
    Notification.error("Username must be at least 3 characters");
    return;
}
```

3. Gestión del Estado Local

Archivo: `frontend/js/config.js`

Estado almacenado:

- JWT token (localStorage)
- User ID
- Username
- Datos de partida actual (sessionStorage)

```
const Storage = {
    setToken(token) {
        localStorage.setItem('tictactoe_token', token);
    },

    getUserInfo() {
        return {
            userId: localStorage.getItem('tictactoe_user_id'),
            username: localStorage.getItem('tictactoe_username')
        };
    }
};
```

4. Comunicación con el Servidor

Archivos:

- `frontend/js/lobby/lobby.js` - WebSocket lobby events
- `frontend/js/game/game.js` - WebSocket game events

Eventos enviados:

- `authenticate` - Autenticación inicial
- `invite_player` - Invitar jugador
- `accept_invitation` - Aceptar invitación
- `make_move` - Realizar movimiento
- `forfeit_game` - Abandonar partida

Eventos recibidos:

- `authenticated` - Confirmación de autenticación
- `online_users` - Lista de usuarios online
- `invitation_received` - Nueva invitación
- `game_started` - Partida iniciada
- `move_made` - Movimiento realizado
- `game_forfeited` - Partida abandonada

Resumen Frontend:

```
┌─────────────────────────────────────┐
│         FRONTEND                    │
├─────────────────────────────────────┤
│ ✓ Renderizar UI                     │
│ ✓ Validación inicial (UX)           │
│ ✓ Gestión de estado local           │
│ ✓ Comunicación con servidor         │
│ ✓ Feedback al usuario               │
│                                     │
│ ✗ Lógica de negocio                 │
│ ✗ Validación de seguridad           │
│ ✗ Persistencia de datos             │
└─────────────────────────────────────┘
```

Responsabilidades del Backend

Ubicación: `backend/` directory

1. Autenticación y Seguridad

Archivos:

- `backend/app/auth/auth.py` - JWT generation/validation
- `backend/app/auth/password.py` - Password hashing (bcrypt)

Responsabilidades:

- Hash de contraseñas con bcrypt (salt automático)
- Generación de JWT tokens
- Validación de JWT en cada request
- Protección contra ataques de fuerza bruta

Hash de contraseñas:

```python
# backend/app/auth/password.py
import bcrypt

def hash_password(password: str) -> str:
```

```
    """Hash password usando bcrypt con salt automático"""
    salt = bcrypt.gensalt()
    hashed = bcrypt.hashpw(password.encode('utf-8'), salt)
    return hashed.decode('utf-8')

def verify_password(plain_password: str, hashed_password: str) -> bool:
    """Verificar contraseña"""
    return bcrypt.checkpw(
        plain_password.encode('utf-8'),
        hashed_password.encode('utf-8')
    )
```

JWT Security:

- Tokens expiran después de 60 minutos (configurable)
- Secret key único para firma
- Algoritmo HS256

2. Lógica del Juego

Archivos:

- `backend/app/game/game_logic.py` - Reglas del juego
- `backend/app/game/game_manager.py` - Gestión de partidas

Responsabilidades:

- Validar movimientos (turno correcto, celda vacía)
- Detectar victoria (3 en línea)
- Detectar empate (tablero lleno sin victoria)
- Gestionar estado del juego
- Actualizar estadísticas de jugadores

Validación de movimientos:

```
# backend/app/game/game_manager.py
async def make_move(self, game_id: int, user_id: int, position: int, db:
AsyncSession):
    game_data = self.active_games.get(game_id)

    # 1. Verificar que es el turno del jugador
    if game_data['current_turn'] != user_id:
        raise ValueError("Not your turn")

    # 2. Verificar que la celda está vacía
    if game_data['board'][position] != '-':
        raise ValueError("Cell already taken")

    # 3. Determinar símbolo (X o O)
    symbol = 'X' if user_id == game_data['player1_id'] else 'O'
```

```
    # 4. Actualizar tablero
    board_list = list(game_data['board'])
    board_list[position] = symbol
    new_board = ''.join(board_list)

    # 5. Verificar victoria
    winner, winning_line = check_winner(new_board)

    # 6. Actualizar en DB
    game = await db.get(Game, game_id)
    game.board_state = new_board
    game.current_turn = next_player_id

    if winner:
        game.status = 'finished'
        game.winner_id = user_id
        # Actualizar estadísticas
        await self.update_stats(game, db)

    await db.commit()

    return {
        'board': new_board,
        'current_turn': next_player_id,
        'game_over': winner is not None or is_draw,
        'winner_id': user_id if winner else None,
        'winning_line': winning_line
    }
```

Detección de victoria:

```
# backend/app/game/game_logic.py
def check_winner(board: str) -> tuple[str | None, list[int] | None]:
    """
    Verifica si hay un ganador

    Returns:
        (ganador, línea_ganadora) o (None, None)
    """
    # Líneas ganadoras posibles
    winning_combinations = [
        [0, 1, 2],  # Fila 1
        [3, 4, 5],  # Fila 2
        [6, 7, 8],  # Fila 3
        [0, 3, 6],  # Columna 1
        [1, 4, 7],  # Columna 2
        [2, 5, 8],  # Columna 3
        [0, 4, 8],  # Diagonal \
        [2, 4, 6]   # Diagonal /
    ]

    for combo in winning_combinations:
```

```
        if (board[combo[0]] != '-' and
            board[combo[0]] == board[combo[1]] == board[combo[2]]):
            return board[combo[0]], combo  # Retorna 'X' o 'O' y posiciones

    return None, None
```

3. Bot AI (Jugador Virtual)

Archivo: `backend/app/game/bot_ai.py`

Dificultades:

- **Easy:** Movimientos aleatorios
- **Medium:** Algoritmo Minimax con profundidad limitada
- **Hard:** Algoritmo Minimax completo (inmejorable)

Implementación Minimax:

```python
# backend/app/game/bot_ai.py
class BotAI:
    def __init__(self, difficulty: str, symbol: str):
        self.difficulty = difficulty
        self.symbol = symbol
        self.opponent_symbol = 'X' if symbol == 'O' else 'O'

    def get_best_move(self, board: str) -> int:
        if self.difficulty == 'easy':
            return self._random_move(board)
        elif self.difficulty == 'medium':
            return self._minimax_limited(board)
        else:  # hard
            return self._minimax_perfect(board)

    def _minimax_perfect(self, board: str) -> int:
        """Algoritmo Minimax - juego perfecto"""
        best_score = -float('inf')
        best_move = None

        for i in range(9):
            if board[i] == '-':
                # Simular movimiento
                new_board = board[:i] + self.symbol + board[i+1:]
                score = self._minimax(new_board, False)

                if score > best_score:
                    best_score = score
                    best_move = i

        return best_move
```

4. Gestión de Eventos en Tiempo Real

Archivo: `backend/app/websocket/game_events.py`

Event Bus:

- Registra event handlers de Socket.IO
- Maneja eventos de lobby (invitaciones)
- Maneja eventos de juego (movimientos)
- Broadcast a rooms (grupos de jugadores)

Rooms de Socket.IO:

```
# Cuando se crea una partida, ambos jugadores se unen a un room
game_room = f"game_{game_id}"  # Ej: "game_123"
await sio.enter_room(player1_sid, game_room)
await sio.enter_room(player2_sid, game_room)

# Broadcast a todos en el room
await sio.emit('move_made', move_data, room=game_room)
```

5. Persistencia en Base de Datos

Archivo: `backend/app/database.py`, `backend/app/models.py`

Modelos (Tablas):

1. **User** - Usuarios registrados
2. **Game** - Partidas (activas y finalizadas)
3. **UserStats** - Estadísticas de jugadores
4. **Invitation** - Invitaciones de juego
5. **GameLog** - Eventos del servidor (logging)

Operaciones:

- CREATE: Nuevos usuarios, partidas, invitaciones
- READ: Consultar datos, leaderboard
- UPDATE: Actualizar estado de partida, marcar usuario online
- (No hay DELETE de usuarios o partidas - histórico completo)

Ejemplo de modelo:

```
# backend/app/models.py
class Game(Base):
    __tablename__ = 'games'

    id = Column(Integer, primary_key=True)
    player1_id = Column(Integer, ForeignKey('users.id'))
    player2_id = Column(Integer, ForeignKey('users.id'), nullable=True)
```

```
    status = Column(String, default='active')  # active, finished
    board_state = Column(String, default='---------')
    current_turn = Column(Integer)
    winner_id = Column(Integer, nullable=True)
    is_bot_game = Column(Boolean, default=False)
    bot_difficulty = Column(String, nullable=True)
    created_at = Column(DateTime, default=datetime.utcnow)
    finished_at = Column(DateTime, nullable=True)
```

Resumen Backend:

```
┌─────────────────────────────────────┐
│         BACKEND                     │
├─────────────────────────────────────┤
│ ✓ Autenticación (JWT, bcrypt)       │
│ ✓ Validación de seguridad           │
│ ✓ Lógica de negocio (juego)         │
│ ✓ Bot AI (3 dificultades)           │
│ ✓ Event-driven architecture         │
│ ✓ Persistencia en DB                │
│ ✓ Estadísticas y ranking            │
│ ✓ Logging de eventos                │
│                                     │
│ ✗ Renderizado de UI                 │
└─────────────────────────────────────┘
```

Formatos de Mensajes y Serialización

Mensajes HTTP/REST

1. POST /api/register

Request:

```
{
  "username": "alice",
  "password": "mySecurePassword123",
  "email": "alice@example.com"
}
```

Response (Success - 200 OK):

```
{
  "access_token":
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxIiwidXNlcm5hbWUiOiJhbGljZSIsImV4
```

```
cCI6MTcwOTMxMjQwMCwiaWF0IjoxNzA5MzA4ODAwfQ.signature_here",
  "token_type": "bearer",
  "user_id": 1,
  "username": "alice"
}
```

Response (Error - 400 Bad Request):

```
{
  "detail": "Username already taken"
}
```

2. POST /api/login

Request:

```
{
  "username": "alice",
  "password": "mySecurePassword123"
}
```

Response (Success - 200 OK):

```
{
  "access_token": "eyJhbGci...",
  "token_type": "bearer",
  "user_id": 1,
  "username": "alice"
}
```

3. GET /api/stats

Request Headers:

```
Authorization: Bearer eyJhbGci...
```

Response:

```
{
  "total_games": 25,
  "wins": 15,
  "losses": 8,
```

```
  "draws": 2,
  "win_rate": 60.0,
  "ranking_points": 1450,
  "current_streak": 3,
  "best_streak": 7
}
```

4. GET /api/stats/leaderboard

Response:

```
[
  {
    "rank": 1,
    "username": "alice",
    "wins": 50,
    "losses": 10,
    "draws": 5,
    "win_rate": 76.9,
    "ranking_points": 2100,
    "best_streak": 12
  },
  {
    "rank": 2,
    "username": "bob",
    "wins": 40,
    "losses": 15,
    "draws": 3,
    "win_rate": 69.0,
    "ranking_points": 1850,
    "best_streak": 8
  }
]
```

Mensajes WebSocket

1. authenticate (Client → Server)

```
{
  "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9..."
}
```

Response: authenticated (Server → Client)

```
{
  "user_id": 1,
```

```
  "username": "alice"
}
```

2. online_users (Server → All Clients)

```
{
  "users": [
    {
      "id": 1,
      "username": "alice",
      "in_game": false
    },
    {
      "id": 2,
      "username": "bob",
      "in_game": true
    },
    {
      "id": 3,
      "username": "charlie",
      "in_game": false
    }
  ]
}
```

3. invite_player (Client → Server)

```
{
  "target_user_id": 2
}
```

Response: invitation_sent (Server → Sender)

```
{
  "invitation_id": 42,
  "to_username": "bob"
}
```

Broadcast: invitation_received (Server → Target)

```
{
  "invitation_id": 42,
  "from_user_id": 1,
```

```
  "from_username": "alice"
}
```

4. accept_invitation (Client → Server)

```
{
  "invitation_id": 42
}
```

Broadcast: game_started (Server → Both Players)

```
{
  "game_id": 123,
  "player1": {
    "id": 1,
    "username": "alice",
    "symbol": "X"
  },
  "player2": {
    "id": 2,
    "username": "bob",
    "symbol": "O"
  },
  "board": "---------",
  "current_turn": 1
}
```

5. join_game (Client → Server)

```
{
  "game_id": 123
}
```

Response: game_joined (Server → Client)

```
{
  "game_id": 123
}
```

6. make_move (Client → Server)

```
{
  "game_id": 123,
  "position": 4
}
```

Broadcast: move_made (Server → Both Players)

Durante el juego:

```
{
  "game_id": 123,
  "position": 4,
  "player_id": 1,
  "board": "----X----",
  "current_turn": 2,
  "game_over": false
}
```

Al terminar (victoria):

```
{
  "game_id": 123,
  "position": 8,
  "player_id": 1,
  "board": "XXX-OO---",
  "current_turn": null,
  "game_over": true,
  "result": "win",
  "winner_id": 1,
  "winning_line": [0, 1, 2]
}
```

Al terminar (empate):

```
{
  "game_id": 123,
  "position": 8,
  "player_id": 2,
  "board": "XXOOOXXXO",
  "current_turn": null,
  "game_over": true,
  "result": "draw",
  "winner_id": null,
  "winning_line": null
}
```

7. forfeit_game (Client → Server)

```
{
  "game_id": 123
}
```

Broadcast: game_forfeited (Server → Both Players)

```
{
  "game_id": 123,
  "forfeited_by": 2,
  "winner_id": 1,
  "result": "abandoned"
}
```

8. play_vs_bot (Client → Server)

```
{
  "difficulty": "hard"
}
```

Response: game_started (Server → Client)

```
{
  "game_id": 124,
  "player1": {
    "id": 1,
    "username": "alice",
    "symbol": "X"
  },
  "player2": {
    "id": 0,
    "username": "Bot (hard)",
    "symbol": "O"
  },
  "board": "---------",
  "current_turn": 1,
  "is_bot_game": true,
  "bot_difficulty": "hard"
}
```

Formato del Tablero (Board State)

El tablero se representa como un string de 9 caracteres:

```
Posiciones:     Ejemplo:
 0 | 1 | 2      X | O | X
-----------    -----------
 3 | 4 | 5      - | X | -
-----------    -----------
 6 | 7 | 8      O | - | O

board = "XOX-X-O-O"
```

- `'-'` = celda vacía
- `'X'` = jugador 1 (siempre empieza)
- `'O'` = jugador 2 o bot

Seguridad e Implementación

1. Almacenamiento Seguro de Contraseñas

Problema: Almacenar contraseñas en texto plano es PELIGROSO

Si un atacante obtiene acceso a la base de datos, puede ver todas las contraseñas.

Solución: Hash con bcrypt

Características de bcrypt:

- **One-way function:** No se puede revertir (descifrar)
- **Salt automático:** Protege contra rainbow tables
- **Cost factor:** Hace el hash lento (protege contra brute-force)

Flujo de registro:

```
# 1. Usuario envía password en texto plano
plain_password = "mySecurePassword123"

# 2. Backend hace hash
hashed = hash_password(plain_password)
# Resultado: "$2b$12$K5Hxf.../hash_aquí"

# 3. Se almacena SOLO el hash en DB
user.password_hash = hashed
```

Flujo de login:

```
# 1. Usuario envía password en texto plano
plain_password = "mySecurePassword123"
```

```
# 2. Backend obtiene hash de DB
stored_hash = user.password_hash  # "$2b$12$K5Hxf..."

# 3. Verificar password
is_valid = verify_password(plain_password, stored_hash)
# bcrypt hace hash de nuevo y compara
```

Implementación:

```
# backend/app/auth/password.py
import bcrypt

def hash_password(password: str) -> str:
    """
    Hash password con bcrypt

    - Salt generado automáticamente
    - Cost factor: 12 rounds (2^12 = 4096 iteraciones)
    """
    salt = bcrypt.gensalt()  # Genera salt aleatorio
    hashed = bcrypt.hashpw(password.encode('utf-8'), salt)
    return hashed.decode('utf-8')

def verify_password(plain_password: str, hashed_password: str) -> bool:
    """Verifica password contra hash"""
    return bcrypt.checkpw(
        plain_password.encode('utf-8'),
        hashed_password.encode('utf-8')
    )
```

Ejemplo de hash:

```
Password: "mySecurePassword123"
Hash:     "$2b$12$K5HxfA8Z9.yGxP2e3fZ5.e1Q2W3E4R5T6Y7U8I9O0P"
          │  │  │                                      │
          │  │  └─ Salt (22 caracteres)                └─ Hash (31 caracteres)
          │  └─ Cost factor (12)
          └─ Algoritmo (bcrypt version 2b)
```

2. Autenticación con JWT

Ventajas sobre Sesiones Tradicionales

Sesiones tradicionales:

```
Cliente                    Servidor
  │── login ──────────────→│
```

```
    │                           │ [Crea sesión en memoria]
    │                           │ session_id = "abc123"
    │←── session_id ────────────│
    │                           │
    │─── request + cookie ─────→│
    │    (session_id=abc123)    │ [Busca sesión en memoria]
```

Problemas:

- Servidor debe almacenar sesiones (memoria/DB)
- No escala bien horizontalmente (múltiples servidores)
- Difícil para APIs

JWT (Stateless):

```
Cliente                     Servidor
   │─── login ──────────────────→│
   │                             │ [Genera JWT]
   │                             │ jwt = sign({user_id: 1})
   │←── JWT token ───────────────│
   │                             │
   │─── request + JWT ──────────→│
   │    Authorization: Bearer..│ [Verifica firma]
                                 │ [Extrae user_id del payload]
```

Ventajas:

- Servidor NO almacena estado
- Escala horizontalmente
- Ideal para APIs REST

Estructura de JWT en nuestro proyecto

Generación:

```python
# backend/app/auth/auth.py:20-40
def create_access_token(data: Dict, expires_delta: Optional[timedelta] = None) ->
str:
    to_encode = data.copy()
    now = datetime.now(timezone.utc)

    # Calcular expiración
    if expires_delta:
        expire = now + expires_delta
    else:
        expire = now + timedelta(minutes=60)  # 60 minutos

    # Agregar claims
    to_encode.update({
```

```
        "exp": int(expire.timestamp()),  # Expiration time
        "iat": int(now.timestamp())      # Issued at
    })

    # Firmar con SECRET_KEY
    encoded_jwt = jwt.encode(
        to_encode,
        settings.SECRET_KEY,  # Debe ser secreto y único
        algorithm="HS256"     # HMAC SHA-256
    )

    return encoded_jwt
```

Validación:

```
# backend/app/auth/auth.py:43-57
def verify_token(token: str) -> Optional[Dict]:
    try:
        # Decodificar y verificar firma
        payload = jwt.decode(
            token,
            settings.SECRET_KEY,
            algorithms=["HS256"]
        )
        # Si llega aquí, el token es válido
        return payload
    except JWTError:
        # Token inválido, expirado, o firma incorrecta
        return None
```

Uso en FastAPI:

```
# backend/app/auth/auth.py:84-128
async def get_current_user(
    credentials: HTTPAuthorizationCredentials = Depends(security),
    db: AsyncSession = Depends(get_db)
) -> User:
    """
    Dependencia para proteger endpoints

    Uso:
        @app.get("/api/stats")
        async def get_stats(current_user: User = Depends(get_current_user)):
            # current_user ya está autenticado
    """
    # 1. Extraer token del header
    token = credentials.credentials  # "Bearer eyJhbG..."

    # 2. Verificar token
```

```
    payload = verify_token(token)
    if payload is None:
        raise HTTPException(status_code=401, detail="Invalid token")

    # 3. Extraer user_id
    user_id = int(payload.get("sub"))

    # 4. Buscar usuario en DB
    user = await db.get(User, user_id)
    if user is None:
        raise HTTPException(status_code=401, detail="User not found")

    return user
```

3. Validación de Entrada

Backend Validation (Crítica)

Nunca confíes en el cliente. Siempre valida en el servidor.

Validación de username:

```
# backend/app/utils/validators.py
def validate_username(username: str) -> tuple[bool, str]:
    """
    Valida username

    Returns:
        (is_valid, error_message)
    """
    if not username:
        return False, "Username is required"

    if len(username) < 3:
        return False, "Username must be at least 3 characters"

    if len(username) > 20:
        return False, "Username must be at most 20 characters"

    if not username.isalnum():
        return False, "Username must be alphanumeric"

    return True, ""
```

Validación de movimientos:

```
# backend/app/utils/validators.py
def validate_move(position: int) -> tuple[bool, str]:
    """Valida posición de movimiento (0-8)"""
```

```
    if not isinstance(position, int):
        return False, "Position must be an integer"

    if position < 0 or position > 8:
        return False, "Position must be between 0 and 8"

    return True, ""
```

Frontend Validation (UX)

Validación en frontend es para **mejorar experiencia de usuario**, no para seguridad.

```
// frontend/js/auth/register.js
if (username.length < 3) {
    Notification.error("Username must be at least 3 characters");
    return;
}
```

4. Prevención de Ataques Comunes

SQL Injection

Problema:

```
# PELIGROSO - NO HACER
query = f"SELECT * FROM users WHERE username = '{username}'"
# Si username = "'; DROP TABLE users; --"
# Query resultante: SELECT * FROM users WHERE username = ''; DROP TABLE users; --'
```

Solución: Usar ORM (SQLAlchemy) con queries parametrizadas

```
# SEGURO - SQLAlchemy
result = await db.execute(
    select(User).where(User.username == username)
)
# SQLAlchemy escapa automáticamente
```

Cross-Site Scripting (XSS)

Problema:

```
// Si username contiene: <script>alert('XSS')</script>
document.getElementById('username').innerHTML = username;
// Se ejecuta el script!
```

Solución: Usar textContent en lugar de innerHTML

```
// SEGURO
document.getElementById('username').textContent = username;
// El script se muestra como texto, no se ejecuta
```

CORS (Cross-Origin Resource Sharing)

Permite que el frontend (puerto 5500) acceda al backend (puerto 8000).

```
# backend/app/server.py:39-47
app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],  # En producción: solo dominios específicos
    allow_credentials=True,
    allow_methods=["GET", "POST", "PUT", "DELETE", "OPTIONS"],
    allow_headers=["*"]
)
```

En producción:

```
allow_origins=[
    "https://tictactoe.example.com",
    "https://www.tictactoe.example.com"
]
```

5. HTTPS en Producción

Desarrollo (localhost):

```
http://localhost:8000
```

Producción:

```
https://api.tictactoe.example.com
```

Obtener certificado SSL/TLS:

1. **Let's Encrypt** (gratis, automatizado)
2. **Certbot** para nginx/Apache

3. Cloud providers (AWS Certificate Manager, etc.)

Nginx como reverse proxy:

```
server {
    listen 443 ssl;
    server_name api.tictactoe.example.com;

    ssl_certificate /etc/letsencrypt/live/api.tictactoe.example.com/fullchain.pem;
    ssl_certificate_key
/etc/letsencrypt/live/api.tictactoe.example.com/privkey.pem;

    location / {
        proxy_pass http://localhost:8000;
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection 'upgrade';
        proxy_set_header Host $host;
    }
}
```

Cumplimiento de Requerimientos

Requerimientos Base (9/9 Completos)

#	Requerimiento	Implementación	Archivos
1	Registro de usuarios con username/password, persistencia segura	☑ Hash con bcrypt, SQLite	`backend/app/auth/password.py`, `backend/app/server.py:111`
2	Autenticar usuarios sin exponer datos	☑ JWT tokens, password nunca se retorna	`backend/app/auth/auth.py:20-40`
3	Mostrar usuarios en línea	☑ WebSocket broadcast `online_users`	`backend/app/server.py:446-463`
4	Crear partida e invitar usuarios	☑ Sistema de invitaciones	`backend/app/websocket/game_events.py:129`
5	Aceptar/rechazar invitaciones	☑ `accept_invitation`, `reject_invitation`	`backend/app/websocket/game_events.py:215, 323`

#	Requerimiento	Implementación	Archivos
6	Actualizar tablero según turno, sincronizado	☑ WebSocket broadcast `move_made` a room	`backend/app/websocket/game_events.py:371`
7	Mostrar resultado final a ambos	☑ `game_over`, `winner_id`, `winning_line`	`frontend/js/game/game.js:194`
8	Abandonar juego, victoria al oponente	☑ `forfeit_game` event	`backend/app/websocket/game_events.py:467`
9	Logging de eventos del servidor	☑ Python logging + tabla GameLog	`backend/app/utils/logger.py`

Requerimientos Opcionales (3/5 Implementados)

#	Requerimiento	Estado	Implementación
10	Scoreboard/Ranking	☑ Implementado	Leaderboard con ranking points, win rate
11	Bot player	☑ Implementado	3 dificultades (easy, medium, hard) con Minimax
12	Reconexión a partida	☑ Parcial	Re-carga de partidas activas en `join_game`
13	Server dashboard	❌ No implementado	-
14	Despliegue en la nube	❌ No implementado	-

Resultados Esperados (6/6 Completos)

#	Resultado	Cumplimiento
1	Aplicación funcional 2P en tiempo real	☑ WebSocket con sincronización en tiempo real
2	Seguridad en redes (TLS, SSL, SHA-256)	☑ JWT (HS256), bcrypt, HTTPS ready
3	Arquitectura Cliente-Servidor	☑ FastAPI backend, JS frontend, comunicación REST + WebSocket
4	Arquitectura orientada a eventos	☑ Socket.IO event bus, game rooms
5	Programación de Sockets (OSI)	☑ WebSocket sobre TCP (capa 4-7)
6	Interfaz de usuario simple	☑ HTML/CSS/JS responsive

Preguntas Frecuentes Técnicas

1. ¿Por qué usar WebSocket en lugar de HTTP polling?

HTTP Polling (ineficiente):

```
Cliente envía request cada 1 segundo:
GET /api/game/123/state → Response
[espera 1 segundo]
GET /api/game/123/state → Response
[espera 1 segundo]
...
```

Problemas:

- Alta latencia (hasta 1 segundo de delay)
- Desperdicio de recursos (muchos requests innecesarios)
- No es "tiempo real"

WebSocket (eficiente):

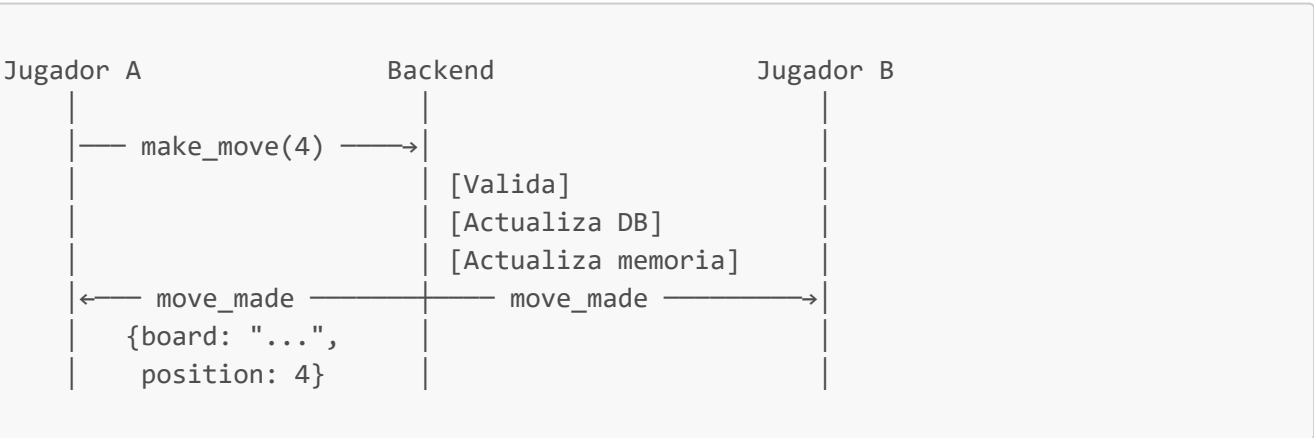
```
Cliente conecta una vez:
[Conexión persistente]
Servidor envía datos cuando hay cambios:
→ move_made (inmediato)
```

Ventajas:

- Latencia baja (~50ms)
- Solo mensajes necesarios
- Verdadero tiempo real

2. ¿Cómo garantiza el sistema la coherencia del estado del juego?

Single Source of Truth: El backend es la única fuente de verdad.



Validaciones del backend:

1. Verificar que es el turno del jugador correcto
2. Verificar que la celda está vacía
3. Actualizar estado en DB (persistencia)
4. Actualizar estado en memoria (active_games)

5. Broadcast a AMBOS jugadores (mismo mensaje)

El frontend nunca modifica el estado directamente:

```
// NO hace esto:
// currentBoard[position] = 'X';  // ❌

// Hace esto:
socket.emit('make_move', {position: 4});  // ☑
// Y espera confirmación del servidor
socket.on('move_made', (data) => {
    updateBoard(data.board);  // Actualiza solo cuando servidor confirma
});
```

3. ¿Qué sucede si un jugador se desconecta durante una partida?

Escenario 1: Desconexión temporal (red inestable)

1. Cliente pierde conexión WebSocket
2. Socket.IO intenta reconectar automáticamente
3. Al reconectar, cliente llama `join_game(game_id)`
4. Backend re-carga partida desde DB si no está en memoria
5. Cliente recibe estado actualizado

Escenario 2: Cierre del navegador

1. Backend detecta `disconnect` event
2. Marca usuario como offline
3. Partida permanece en DB (status='active')
4. Si usuario regresa y hace `join_game`, puede continuar

Escenario 3: Abandono intencional

Usuario hace click en "Forfeit":

```
socket.emit('forfeit_game', {game_id: 123});
```

Backend:

- Marca partida como finished
- Asigna victoria al oponente
- Actualiza estadísticas
- Broadcast `game_forfeited` a ambos

4. ¿Cómo escala el sistema para múltiples partidas simultáneas?

Gestión de partidas activas en memoria:

```python
# backend/app/game/game_manager.py
class GameManager:
    def __init__(self):
        # Diccionario de partidas activas
        self.active_games = {}  # {game_id: game_data}
        self.user_to_game = {}  # {user_id: game_id}
```

Cada partida es independiente:

```
Room "game_123"              Room "game_456"
    |                            |
Player A ↔ Server ↔ Player C   Player E ↔ Server ↔ Player F
Player B                       Player D
```

Rooms de Socket.IO:

- Cada partida tiene un room único: `game_{game_id}`
- Broadcast solo afecta jugadores en ese room
- No hay interferencia entre partidas

Limitaciones actuales:

- Un servidor puede manejar ~10,000 conexiones concurrentes (Socket.IO)
- Para más, necesitarías:
  - Múltiples servidores (load balancer)
  - Redis para compartir estado entre servidores
  - Message queue (RabbitMQ, Redis Pub/Sub)

5. ¿Cómo funciona el algoritmo Minimax del bot?

Minimax es un algoritmo de teoría de juegos para juegos de suma cero (lo que uno gana, el otro pierde).

Concepto:

- **Maximizar:** Bot busca el mejor movimiento para sí mismo
- **Minimizar:** Asume que el oponente jugará óptimamente

Ejemplo simple (profundidad 2):

```
Tablero actual:
X | O | X
---------
- | X | -
---------
O | - | -

Bot (O) debe elegir posición.
```

```
Evalúa cada movimiento posible:
- Posición 3: Simula que juega ahí
  - Simula que X responde en 5 → X gana (score: -10)
  - Simula que X responde en 7 → O gana (score: +10)
  - Peor caso (X óptimo): -10

- Posición 5: Simula que juega ahí
  - Simula que X responde en 3 → O gana (score: +10)
  - Simula que X responde en 7 → Empate (score: 0)
  - Peor caso: 0

- Posición 7: Simula que juega ahí
  - Simula que X responde en 3 → X gana (score: -10)
  - Peor caso: -10

Mejor opción: Posición 5 (score: 0)
```

Implementación:

```python
# backend/app/game/bot_ai.py (simplificado)
def _minimax(self, board: str, is_maximizing: bool, depth: int = 0) -> int:
    # 1. Caso base: juego terminado
    winner, _ = check_winner(board)
    if winner == self.symbol:
        return 10 - depth  # Prefiere ganar rápido
    elif winner == self.opponent_symbol:
        return depth - 10  # Prefiere perder lento
    elif is_draw(board):
        return 0

    # 2. Caso recursivo
    if is_maximizing:  # Turno del bot
        max_score = -float('inf')
        for i in range(9):
            if board[i] == '-':
                # Simular movimiento
                new_board = board[:i] + self.symbol + board[i+1:]
                score = self._minimax(new_board, False, depth + 1)
                max_score = max(max_score, score)
        return max_score
    else:  # Turno del oponente
        min_score = float('inf')
        for i in range(9):
            if board[i] == '-':
                new_board = board[:i] + self.opponent_symbol + board[i+1:]
                score = self._minimax(new_board, True, depth + 1)
                min_score = min(min_score, score)
        return min_score
```

Dificultades:

- **Easy:** Movimientos aleatorios (no usa Minimax)
- **Medium:** Minimax con profundidad limitada (depth < 3)
- **Hard:** Minimax completo (explora todo el árbol) - **Imborrable**

6. ¿Qué información se envía por la red?

Tamaño aproximado de mensajes:

Evento	Tamaño	Frecuencia
`authenticate`	~200 bytes (JWT)	1 vez al conectar
`online_users`	~50 bytes/usuario	Cada vez que alguien conecta/desconecta
`make_move`	~30 bytes	Cada movimiento (~9 por partida)
`move_made`	~100 bytes	Cada movimiento (broadcast)

Ejemplo de tráfico de una partida completa:

```
1. Conexión inicial:
   - authenticate: 200 bytes
   - online_users: 150 bytes (3 usuarios)

2. Invitación:
   - invite_player: 20 bytes
   - invitation_received: 50 bytes
   - accept_invitation: 20 bytes
   - game_started: 150 bytes

3. Partida (9 movimientos):
   - make_move × 9: 270 bytes
   - move_made × 9: 900 bytes

Total: ~1.7 KB por partida
```

Comparado con video streaming: ~500 KB/s **Tráfico de Tic-Tac-Toe es MÍNIMO.**

7. ¿Cómo se puede monitorear el tráfico de red con Wireshark?

Pasos para monitorear:

1. **Iniciar Wireshark**
2. **Seleccionar interfaz:** Loopback (lo) para localhost
3. **Filtrar por puerto:** `tcp.port == 8000`
4. **Iniciar captura**
5. **Realizar acciones en la aplicación**
6. **Analizar paquetes:**

HTTP Request (Login):

```
POST /api/login HTTP/1.1
Host: localhost:8000
Content-Type: application/json
Content-Length: 45

{"username":"alice","password":"test123"}
```

HTTP Response:

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 156

{"access_token":"eyJhbGci...","token_type":"bearer","user_id":1,"username":"alice"
}
```

WebSocket Handshake:

```
GET / HTTP/1.1
Host: localhost:8000
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Sec-WebSocket-Version: 13
```

WebSocket Frame (move_made):

```
Frame Type: Text
Payload: {"game_id":123,"position":4,"board":"----X----
","current_turn":2,"game_over":false}
```

8. ¿Por qué usar SQLite en lugar de PostgreSQL?

SQLite (usado en proyecto):

Ventajas:

- Sin servidor separado (archivo local)
- Fácil de configurar
- Perfecto para desarrollo
- Suficiente para <100 usuarios concurrentes

Desventajas:

- No ideal para alta concurrencia

- Escrituras bloqueantes (un write a la vez)
- No distribuido

PostgreSQL (recomendado para producción):

Ventajas:

- Alta concurrencia
- ACID completo
- Replicación
- Escalable

Cambio a PostgreSQL:

```
# Solo cambiar DATABASE_URL
# SQLite:
DATABASE_URL = "sqlite+aiosqlite:///./database.db"

# PostgreSQL:
DATABASE_URL = "postgresql+asyncpg://user:pass@localhost/dbname"

# Todo lo demás funciona igual (SQLAlchemy abstrae)
```

Conclusión

Este proyecto implementa un sistema completo de juego multijugador en red con:

- **Comunicación HTTP/REST** para autenticación y consultas
- **Comunicación WebSocket** para tiempo real
- **Seguridad robusta** (JWT, bcrypt, validación)
- **Arquitectura orientada a eventos** (Socket.IO event bus)
- **Persistencia de datos** (SQLite con SQLAlchemy)
- **Bot AI** con algoritmo Minimax
- **Sistema de ranking** y estadísticas

Todos los requerimientos base del proyecto Capstone han sido implementados, junto con 3 de 5 requerimientos opcionales.

Referencias

- **FastAPI Documentation:** https://fastapi.tiangolo.com/
- **Socket.IO Documentation:** https://socket.io/docs/v4/
- **JWT Introduction:** https://jwt.io/introduction
- **bcrypt:** https://pypi.org/project/bcrypt/
- **WebSocket RFC 6455:** https://datatracker.ietf.org/doc/html/rfc6455
- **Minimax Algorithm:** https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory/

Autor: [Tu Nombre] **Fecha:** Octubre 2025 **Universidad:** Universidad Jala **Curso:** Redes de Computadoras 2 - CSNT-245