Magnate Backend API Documentation

None

08 - Roberta Williams - Equipo de backend

2026

Tabla de Contenidos

1. Welcome to Magnate backend documentation

PDF version of the documentation

2. Integration

2.1 WebSockets

2.1.1 `PublicQueueConsumer`

Bases: `AsyncWebsocketConsumer`

WebSocket consumer for the public matchmaking queue.

Handles automatic matchmaking: players connect, wait until enough players are in the queue, and are then redirected to a newly created game.

How to Connect

Endpoint: `ws://<host>/ws/queue/`

Authentication required: Yes — the user must be authenticated via session or token before connecting. Unauthenticated connections are closed with code `4002`.

Connection Lifecycle

1. Client opens the WebSocket.
2. Server adds the user to the matchmaking queue.
3. If enough players are queued (`NUM_PUBLIC_GAME_PLAYERS`), the server automatically creates a game and sends a `match_found` event to every matched player.
4. Each matched client receives the `game_id` and should navigate to the game screen, then connect to `GameConsumer`.

Messages: Client → Server

Cancel Queue

Voluntarily leave the matchmaking queue.

```
{ "action": "cancel" }
```

Messages: Server → Client

Match Found

Sent when matchmaking succeeds. The connection is closed with code `4001` immediately after this message.

```
{
    "action": "match_found",
    "game_id": 42
}
```

Error

Sent when an invalid message is received.

```
{
    "action": "error",
    "message": "Datos invalidos."
}
```

Close Codes

Code	Meaning
4001	Match found — user was moved to a game. Connect to `GameConsumer`.
4002	Unauthorized — user was not authenticated.
4000	User cancelled the queue.

Example Flow (JavaScript)

```
const socket = new WebSocket("ws://localhost:8000/ws/queue/");

socket.onmessage = (event) => {
    const data = JSON.parse(event.data);
    if (data.action === "match_found") {
        // Navigate to game with data.game_id
        connectToGame(data.game_id);
    }
};

// To leave the queue manually:
socket.send(JSON.stringify({ action: "cancel" }));
```

- **Source code in** `magnate/consumers.py`

```
class PublicQueueConsumer(AsyncWebsocketConsumer):
    """
    WebSocket consumer for the public matchmaking queue.

    Handles automatic matchmaking: players connect, wait until enough players
    are in the queue, and are then redirected to a newly created game.

    ---
    ## How to Connect

    **Endpoint:** ``ws://<host>/ws/queue/``

    **Authentication required:** Yes — the user must be authenticated via session
    or token before connecting. Unauthenticated connections are closed with code ``4002``.

    ---
    ## Connection Lifecycle

    1. Client opens the WebSocket.
    2. Server adds the user to the matchmaking queue.
    3. If enough players are queued (``NUM_PUBLIC_GAME_PLAYERS``), the server
       automatically creates a game and sends a ``match_found`` event to every
```

```
   matched player.
4. Each matched client receives the ``game_id`` and should navigate to the
   game screen, then connect to ``GameConsumer``.

---
## Messages: Client → Server

### Cancel Queue
Voluntarily leave the matchmaking queue.

```json
{ "action": "cancel" }
```

Messages: Server → Client

Match Found
Sent when matchmaking succeeds. The connection is closed with code ``4001``
immediately after this message.

```json
{
    "action": "match_found",
    "game_id": 42
}
```

Error
Sent when an invalid message is received.

```json
{
    "action": "error",
    "message": "Datos invalidos."
}
```

Close Codes

Code	Meaning
4001	Match found — user was moved to a game. Connect to ``GameConsumer``.
4002	Unauthorized — user was not authenticated.
4000	User cancelled the queue.

Example Flow (JavaScript)

```js
const socket = new WebSocket("ws://localhost:8000/ws/queue/");

socket.onmessage = (event) => {
    const data = JSON.parse(event.data);
    if (data.action === "match_found") {
        // Navigate to game with data.game_id
        connectToGame(data.game_id);
    }
```
```

```
};

// To leave the queue manually:
socket.send(JSON.stringify({ action: "cancel" }));
```
"""

QUEUE_GROUP = "public_queue"

async def connect(self):
    scope_user = self.scope.get('user')
    if scope_user is None or getattr(scope_user, 'is_anonymous', True):
        await self.close(code=4002)
        return

    self.user = await database_sync_to_async(CustomUser.objects.get)(pk=scope_user.pk)
    await self.accept()

    # join queue
    await self.channel_layer.group_add(self.QUEUE_GROUP, self.channel_name)

    await self.add_user_to_queue(self.user, self.channel_name)

    # checking
    match_result = await self.matchmaking_logic()

    if match_result:
        game_id, players_channels = match_result
        for channel in players_channels:
            await self.channel_layer.send(
                channel,
                {
                    'type': 'match_found_event',
                    'game_id': game_id
                }
)

async def disconnect(self, close_code):
    await self.channel_layer.group_discard(self.QUEUE_GROUP, self.channel_name)

    if close_code == 4001:
        print(f"User {self.user} found a match and is leaving the queue.")
    elif close_code == 4002:
        return
    else:
        print(f"User {self.user} left the queue.")
        await self.remove_user_from_queue(self.user)

async def receive(self, text_data):
    try:
        data = json.loads(text_data)
        if data.get('action') == 'cancel':
            await self.close(code=4000)
    except json.JSONDecodeError:
        await self.send_error("Datos invalidos.")

async def match_found_event(self, event):
    await self.send(text_data=json.dumps({
        'action': 'match_found',

```
            'game_id': event['game_id']
        }))
        await self.close(code=4001)

    async def send_error(self, message):
        await self.send(text_data=json.dumps({
            'action': 'error',
            'message': message
        }))


# ---------------- DB access methods --------------#
    @database_sync_to_async
    def add_user_to_queue(self, user, channel_name):
        existing_position = PublicQueuePosition.objects.filter(user=user).first()
        if existing_position:
            # User already in queue, don't add again
            return None

        # Create queue position for the user
        PublicQueuePosition.objects.create(
            user=user,
            channel = channel_name,
            date_time=timezone.now()

        )


    @database_sync_to_async
    def remove_user_from_queue(self, user):
        existing_position =  PublicQueuePosition.objects.filter(user=user).first()
        if existing_position:
            existing_position.delete()
        else:
            # User not in queue
            return None


    # TODO: be aware of race conditions -> new users while executing the method
    @database_sync_to_async
    def matchmaking_logic(self):
        with transaction.atomic():

            players = list(PublicQueuePosition.objects.select_for_update().order_by('date_time')[:NUM_PUBLIC_GAME_PLA

            if len(players) < NUM_PUBLIC_GAME_PLAYERS:
                return None

            player_channels = [player.channel for player in players]
            users = [player.user for player in players]

            # create with compulsory sh -> TODO: change to random
            game = Game.objects.create(
                datetime=timezone.now(),
                active_turn_player=users[0],
                active_phase_player=users[0],
                phase=GameManager.ROLL_THE_DICES
            )
```

```
            # initialize money and positions (see then what the optimal money)
            game.money = {str(u.pk): 1500 for u in users}
            game.positions = {str(u.pk): "000" for u in users}


            game.players.set(users)
            game.ordered_players = [u.pk for u in users]
            game.ordered_players = random.sample(game.ordered_players, len(game.ordered_players)) #random order of pl


            task = kick_out_callback.apply_async(args=[game.pk, users[0].pk], countdown=50) #necessary for first turn
            game.kick_out_task_id = task.id
            game.save()


            for user in users:
                user.active_game = game
                user.played_games.add(game)
                user.save()

            PublicQueuePosition.objects.filter(pk__in=[p.pk for p in players]).delete()

            return (game.pk, player_channels)
```

2.1.2 `PrivateRoomConsumer`

Bases: `AsyncWebsocketConsumer`

WebSocket consumer for private game room lobbies.

Manages the pre-game lobby where a host invites friends, players set their ready status, and the host starts the game when all players are ready. Supports bot-filling up to the configured target player count.

How to Connect

Endpoint: `ws://<host>/ws/room/<room_code>/`

Authentication required: Yes. Unauthenticated users are rejected with code `4002`.

The `room_code` must correspond to an existing `PrivateRoom`. Attempting to join a non-existent or full room will result in an error message followed by close code `4003`.

Connection Lifecycle

1. Client opens the WebSocket with a valid `room_code`.
2. Server validates room existence, capacity, and that the user is not already in the room.
3. On success, the server broadcasts a `joined` lobby update to all members.
4. Players toggle ready status. The host can change settings (bot level, target player count).
5. When all players are ready, the host sends `start_game`.
6. Server creates the game (filling remaining slots with bots if needed) and broadcasts a `game_start` event to everyone.
7. Each client receives the `game_id` and connects to `GameConsumer`.

Messages: Client → Server

All messages are JSON objects with a `command` field.

Toggle Ready Status

```
{ "command": "ready_status", "is_ready": true }
```

Start Game *(host only)*

Requires all players to be ready and at least `MIN_PRIVATE_GAME_PLAYERS` in the room. Bots are added to reach `target_players` if needed.

```
{ "command": "start_game" }
```

Send Chat Message

```
{ "command": "chat_message", "message": "Hello!" }
```

Update Room Settings *(host only)*

Change the target number of players and/or the bot difficulty level.

```
{
    "command": "update_settings",
    "bot_level": "easy",
    "target_players": 4
}
```

Messages: Server → Client

Player Joined

Broadcast to all members when someone connects.

```
{
    "action": "joined",
    "user": "alice",
    "owner": "alice",
    "is_owner": true,
    "players": [
        { "username": "alice", "ready_to_play": false }
    ]
}
```

Player Left

Broadcast to remaining members when someone disconnects. `owner` may change if the previous owner left (host migration).

```
{
    "action": "player_left",
    "user_left": "bob",
    "owner": "alice",
    "is_owner": true,
    "players": [
```

```
        { "username": "alice", "ready_to_play": false }
    ]
}
```

Ready Status Update

Broadcast to all members when any player changes their ready status.

```
{
    "action": "ready_status",
    "user": "bob",
    "is_ready": true,
    "owner": "alice",
    "is_owner": false
}
```

Settings Changed

Broadcast to all members when the host updates room settings.

```
{
    "action": "settings_changed",
    "bot_level": "hard",
    "target_players": 4
}
```

Game Start

Broadcast to all members when the game has been created. The connection is closed with code `4001` immediately after.

```
{
    "action": "game_start",
    "game_id": 42
}
```

Chat Message

Broadcast to all members.

```
{
    "action": "chat_message",
    "user": "alice",
    "message": "Good luck!"
}
```

Error

Sent only to the client that triggered the error.

```
{
    "action": "error",
    "message": "Solo el host puede iniciar una partida."
}
```

Close Codes

Code	Meaning
4001	Game started — connect to `GameConsumer` with the received `game_id`.
4002	Unauthorized.
4003	Room not found, full, or user already in room.

Notes

- `is_owner` in lobby events is computed per-recipient: `true` only for the current room host.
- If the host disconnects, ownership is automatically transferred to the next oldest player.
- If the last player leaves, the room is deleted.
- Bots are created at game start to fill slots up to `target_players`; they are not visible in the lobby.

Example Flow (JavaScript)

```
const socket = new WebSocket("ws://localhost:8000/ws/room/ABC123/");

socket.onmessage = (event) => {
    const data = JSON.parse(event.data);

    switch (data.action) {
        case "joined":
        case "player_left":
            updatePlayerList(data.players, data.owner);
            break;
        case "ready_status":
            updateReadyIndicator(data.user, data.is_ready);
            break;
        case "settings_changed":
            updateSettings(data.bot_level, data.target_players);
            break;
        case "game_start":
            connectToGame(data.game_id);
            break;
        case "error":
            showError(data.message);
            break;
    }
};

// Mark yourself as ready:
socket.send(JSON.stringify({ command: "ready_status", is_ready: true }));

// Host starts the game:
socket.send(JSON.stringify({ command: "start_game" }));
```

- **Source code in `magnate/consumers.py`**

```
class PrivateRoomConsumer(AsyncWebsocketConsumer):
    """
    WebSocket consumer for private game room lobbies.

    Manages the pre-game lobby where a host invites friends, players set their
```

````
ready status, and the host starts the game when all players are ready. Supports
bot-filling up to the configured target player count.

---
## How to Connect

**Endpoint:** ``ws://<host>/ws/room/<room_code>/``

**Authentication required:** Yes. Unauthenticated users are rejected with code ``4002``.

The ``room_code`` must correspond to an existing ``PrivateRoom``. Attempting to
join a non-existent or full room will result in an error message followed by
close code ``4003``.

---
## Connection Lifecycle

1. Client opens the WebSocket with a valid ``room_code``.
2. Server validates room existence, capacity, and that the user is not already in
   the room.
3. On success, the server broadcasts a ``joined`` lobby update to all members.
4. Players toggle ready status. The host can change settings (bot level, target
   player count).
5. When all players are ready, the host sends ``start_game``.
6. Server creates the game (filling remaining slots with bots if needed) and
   broadcasts a ``game_start`` event to everyone.
7. Each client receives the ``game_id`` and connects to ``GameConsumer``.

---
## Messages: Client → Server

All messages are JSON objects with a ``command`` field.

### Toggle Ready Status
```json
{ "command": "ready_status", "is_ready": true }
```

### Start Game *(host only)*
Requires all players to be ready and at least ``MIN_PRIVATE_GAME_PLAYERS`` in
the room. Bots are added to reach ``target_players`` if needed.
```json
{ "command": "start_game" }
```

### Send Chat Message
```json
{ "command": "chat_message", "message": "Hello!" }
```

### Update Room Settings *(host only)*
Change the target number of players and/or the bot difficulty level.
```json
{
    "command": "update_settings",
    "bot_level": "easy",
    "target_players": 4
}
```
````

Messages: Server → Client

Player Joined
Broadcast to all members when someone connects.
```json
{
    "action": "joined",
    "user": "alice",
    "owner": "alice",
    "is_owner": true,
    "players": [
        { "username": "alice", "ready_to_play": false }
    ]
}
```

Player Left
Broadcast to remaining members when someone disconnects. ``owner`` may change
if the previous owner left (host migration).
```json
{
    "action": "player_left",
    "user_left": "bob",
    "owner": "alice",
    "is_owner": true,
    "players": [
        { "username": "alice", "ready_to_play": false }
    ]
}
```

Ready Status Update
Broadcast to all members when any player changes their ready status.
```json
{
    "action": "ready_status",
    "user": "bob",
    "is_ready": true,
    "owner": "alice",
    "is_owner": false
}
```

Settings Changed
Broadcast to all members when the host updates room settings.
```json
{
    "action": "settings_changed",
    "bot_level": "hard",
    "target_players": 4
}
```

Game Start
Broadcast to all members when the game has been created. The connection is
closed with code ``4001`` immediately after.
```json
```

```
{
    "action": "game_start",
    "game_id": 42
}
```

Chat Message
Broadcast to all members.
```json
{
    "action": "chat_message",
    "user": "alice",
    "message": "Good luck!"
}
```

Error
Sent only to the client that triggered the error.
```json
{
    "action": "error",
    "message": "Solo el host puede iniciar una partida."
}
```

Close Codes

Code	Meaning
4001	Game started — connect to ``GameConsumer`` with the received ``game_id``.
4002	Unauthorized.
4003	Room not found, full, or user already in room.

Notes

- ``is_owner`` in lobby events is computed per-recipient: ``true`` only for the current room host.
- If the host disconnects, ownership is automatically transferred to the next oldest player.
- If the last player leaves, the room is deleted.
- Bots are created at game start to fill slots up to ``target_players``; they are not visible in the lobby.

Example Flow (JavaScript)

```js
const socket = new WebSocket("ws://localhost:8000/ws/room/ABC123/");

socket.onmessage = (event) => {
    const data = JSON.parse(event.data);

    switch (data.action) {
        case "joined":
        case "player_left":
            updatePlayerList(data.players, data.owner);
            break;
```

```
        case "ready_status":
            updateReadyIndicator(data.user, data.is_ready);
            break;
        case "settings_changed":
            updateSettings(data.bot_level, data.target_players);
            break;
        case "game_start":
            connectToGame(data.game_id);
            break;
        case "error":
            showError(data.message);
            break;
    }
};

// Mark yourself as ready:
socket.send(JSON.stringify({ command: "ready_status", is_ready: true }));

// Host starts the game:
socket.send(JSON.stringify({ command: "start_game" }));
```
```
"""
async def connect(self):
    # Triggered when user opens a new private room or joins an existing one.
    scope_user = self.scope.get('user')
    if scope_user is None or getattr(scope_user, 'is_anonymous', True):
        await self.close(code=4002)
        return

    self.user = await database_sync_to_async(CustomUser.objects.get)(pk=scope_user.pk)

    self.url = self.scope.get('url_route')
    if self.url is None:
        await self.close(code=4002)
        return

    self.room_code = self.url.get('kwargs').get('room_code')

    self.room_group_name = f"lobby_{self.room_code}"

    can_join, message = await self.check_room(self.room_code)

    # Case of invalid code or full lobby -> reject connection
    if not can_join:
        # Acept to send error message and close connection
        await self.accept()
        await self.send(text_data=json.dumps({'error': message}))
        await self.close(code=4003)
        return

    await self.channel_layer.group_add(
        self.room_group_name,
        self.channel_name
    )

    await self.accept()

    players = await self.join_room_group_db(self.room_code, self.user)
    if not players:
```

```
            await self.close(code=4003)
            return

        # Notify lobby members of new player and update player list
        await self.channel_layer.group_send(
            self.room_group_name,
            {
                'type': 'lobby_update',
                'action': 'joined',
                'user': self.user.username,
                'players': players,
                'owner': players[0]['username']
            }
        )

    async def disconnect(self, close_code):
        # Triggered when user leaves the private room lobby.
        # If owner leaves -> change host to the second older player. Else -> just update lobby.
        if close_code == 4002:
            print(f"Unauthorized user attempted to connect and was rejected.")
            return

        await self.channel_layer.group_discard(
            self.room_group_name,
            self.channel_name
        )

        # DB operations -> if needed, rotate host, remove room if empty, etc. Return updated player list and new host
        room_data = await self.leave_room_and_update_host(self.room_code, self.user)

        if not room_data:
            return

        if self.user is None:
            return


        if room_data:
            await self.channel_layer.group_send(
                self.room_group_name,
                {
                    'type': 'lobby_update',
                    'action': 'player_left',
                    'user_left': self.user.username,
                    'owner': room_data["owner"],
                    'players': room_data['players']
                }
            )


    async def receive(self, text_data):
        # Chat messages or 'start_game' command /'ready' status updates.
        if self.user is None:
            return

        try:
            data = json.loads(text_data)
            command = data.get('command')
```

```
if not command:
    await self.send_error("Comando invalido.")
    return


if command == 'start_game':
    is_owner = await self.is_owner(self.user, self.room_code)
    if not is_owner:
            await self.send_error("Solo el host puede iniciar una partida.")
            return
    num_players = await self.get_num_players(self.room_code)

    if num_players < MIN_PRIVATE_GAME_PLAYERS:
        await self.send_error(f"Se necesitan {MIN_PRIVATE_GAME_PLAYERS} jugadores para iniciar la partida
        return

    all_ready = await self.check_all_ready(self.room_code)
    if not all_ready:
        await self.send_error("Todos los jugadores deben estar listos para iniciar la partida.")
        return

    game_pk = await self.create_private_game(self.room_code)

    await self.channel_layer.group_send(
        self.room_group_name,
        {
            'type': 'game_start',
            'game_id': game_pk
        })



elif command == 'ready_status':
    is_ready = data.get('is_ready')

    # Update in db
    await self.update_player_ready_status(self.room_code, self.user, is_ready)

    owner = await self.get_room_owner(self.room_code)


    await self.channel_layer.group_send(
        self.room_group_name,
        {
            'type': 'lobby_update',
            'action': 'ready_status',
            'user': self.user.username,
            'is_ready': is_ready,
            'owner': owner
        }
    )


elif command == 'chat_message':
    message = data.get('message')
    await self.channel_layer.group_send(
                self.room_group_name,
                {
                    'type': 'chat_event',
```

```
                            'user': self.user.username,
                            'message': message
                        }
                    )

            elif command == 'update_settings': # owner changes target users and bot level
                is_owner = await self.is_owner(self.user, self.room_code)
                if not is_owner:
                    await self.send_error("Solo el host puede cambiar la configuración.")
                    return

                bot_level = data.get('bot_level')
                target_players = data.get('target_players')

                await self.update_room_settings(self.room_code, bot_level, target_players)

                # Avisar a todos los de la sala del cambio
                await self.channel_layer.group_send(
                    self.room_group_name,
                    {
                        'type': 'lobby_update',
                        'action': 'settings_changed',
                        'bot_level': bot_level,
                        'target_players': target_players
                    }
                )

        except json.JSONDecodeError:
            await self.send_error("Datos invalidos.")
            return

# -------------------- Handlers  -------------------- #
    async def lobby_update(self, event):
        if self.user is None:
            return

        # Send lobby updates to frontend (new player joined, player left)
        if event['action'] == 'joined':
            await self.send(text_data=json.dumps({
                'action': event['action'],
                'user': event['user'],
                'owner': event['owner'],
                'is_owner': (self.user.username == event['owner']),
                'players': event['players']
            }))
        elif event['action'] == 'player_left':
            await self.send(text_data=json.dumps({
                'action': event['action'],
                'user_left': event['user_left'],
                'owner': event['owner'],
                'is_owner': (self.user.username == event['owner']),
                'players': event['players']
            }))
        elif event['action'] == 'ready_status':
            await self.send(text_data=json.dumps({
                'action': event['action'],
                'user': event['user'],
                'is_ready': event['is_ready'],
                'owner': event['owner'],
```

```
            'is_owner': (self.user.username == event['owner'])
        }))
    elif event['action'] == 'settings_changed':
        await self.send(text_data=json.dumps({
            'action': event['action'],
            'bot_level': event['bot_level'],
            'target_players': event['target_players']
        }))


@database_sync_to_async
def update_room_settings(self, room_code, bot_level, target_players):
    room = PrivateRoom.objects.get(room_code=room_code)
    if bot_level:
        room.bot_level = bot_level
    if target_players:
        room.target_players = target_players
    room.save()

@database_sync_to_async
def create_private_game(self, room_code):
    import random
    from django.utils import timezone
    from .models import PrivateRoom, Game, CustomUser
    from .games import GameManager


    room = PrivateRoom.objects.get(room_code=room_code)
    real_users = list(room.players.all())
    users = real_users.copy()

    # fill with bots
    huecos = room.target_players - len(real_users)
    for i in range(huecos):
        # Generar un nombre único para la partida
        bot_username = f"Bot_{room_code}_{i+1}"
        bot_user, _ = CustomUser.objects.get_or_create(
            username=bot_username,
            defaults={'email': f"{bot_username}@magnate.com", 'is_bot': True } #change level
        )

        bot_user.bot_level = room.bot_level
        bot_user.save()

        users.append(bot_user)

    game = Game.objects.create(
        datetime=timezone.now(),
        active_turn_player=users[0], # Se ajusta abajo
        active_phase_player=users[0],
        phase=GameManager.ROLL_THE_DICES
    )

    game.money = {str(u.pk): 1500 for u in users}
    game.positions = {str(u.pk): "000" for u in users}

    game.players.set(users)
    ordered_pks = [u.pk for u in users]
    random.shuffle(ordered_pks)
```

```
        game.ordered_players = ordered_pks

        first_player = CustomUser.objects.get(pk=ordered_pks[0])
        game.active_turn_player = first_player
        game.active_phase_player = first_player

        for user in users:
            user.active_game = game
            user.played_games.add(game)
            user.current_private_room = None

            user.save()
            PlayerGameStatistic.objects.get_or_create(user=user, game=game)

        room.delete()

        GameManager._set_kick_out_timer(game, first_player)
        if first_player.is_bot:
            from .tasks import bot_play_callback
            bot_play_callback.apply_async(args=[game.pk, first_player.pk], countdown=5) # wait 5 for front to charge

        game.save()



        return game.pk



    async def chat_event(self, event):
        # Send chat messages to frontend
        await self.send(text_data=json.dumps({
            'action': 'chat_message',
            'user': event['user'],
            'message': event['message']
        }))

    async def game_start(self, event):
        await self.send(text_data=json.dumps({
            'action': 'game_start',
            'game_id': event['game_id']
        }))
        await self.close(code=4001)



# ------------------- DB access methos -------------------- #
    @database_sync_to_async
    def check_room(self, room_code):
        if self.user is None:
            return False, None

        # Check if room exists and if user can join (not full, not already in, etc). Create if doesn't exist and user
        room  = PrivateRoom.objects.filter(room_code=room_code).first()

        if not room:
            return False, "Sala no encontrada."
```

```
    #Using relation players in users - private room
    current_number_players =  room.players.count()

    if current_number_players >= MAX_PRIVATE_GAME_PLAYERS:
        return False, "Sala llena."

    user = CustomUser.objects.get(username=self.user.username)
    current_private_room = user.current_private_room

    if current_private_room== room:
        return False, "Ya estás en esta sala."

    return True, None



@database_sync_to_async
def join_room_group_db(self, room_code, user):
    current_user = CustomUser.objects.get(username=user.username)
    room = PrivateRoom.objects.get(room_code=room_code)

    current_user.current_private_room = PrivateRoom.objects.get(room_code=room_code)
    current_user.ready_to_play = False
    current_user.save()

    return list(room.players.values('username', 'ready_to_play'))

@database_sync_to_async
def leave_room_and_update_host(self, room_code, user):
    room = PrivateRoom.objects.filter(room_code=room_code).first()
    if not room:
        return None
    user_from_db = CustomUser.objects.get(username=user.username)

    user_from_db.current_private_room = None
    user_from_db.save()

    new_owner = room.players.exclude(pk=user_from_db.pk).first()

    # if no one left, delete
    if new_owner is None:
        room.delete()
        return None

    room.owner = new_owner
    room.save()

    return {
        'owner': room.owner.username,
        'players': list(room.players.values('username', 'ready_to_play'))
    }

@database_sync_to_async
def update_player_ready_status(self, room_code, user, is_ready):
    user_from_db = CustomUser.objects.get(username=user.username)

    if user_from_db.current_private_room is None:
        return None
```

```
        if user_from_db.current_private_room.room_code != room_code:
            return False

        user_from_db.ready_to_play = is_ready
        user_from_db.save()

    @database_sync_to_async
    def get_num_players(self, room_code):
        # Return the current number of players in the room
        room = PrivateRoom.objects.get(room_code=room_code)
        if not room:
            return 0
        return room.players.count()

    @database_sync_to_async
    def check_all_ready(self, room_code):
        # Check if all players in the room are ready
        room = PrivateRoom.objects.get(room_code=room_code)
        if not room:
            return False

        for player in room.players.all():
            if not player.ready_to_play:
                return False

        return True

    @database_sync_to_async
    def is_owner(self, user, room_code):
        # Check if the user is the host of the room
        room = PrivateRoom.objects.get(room_code=room_code)
        if not room:
            return False

        return room.owner == user

    @database_sync_to_async
    def get_room_owner(self, room_code):
        # Return the username of the room's owner
        room = PrivateRoom.objects.get(room_code=room_code)
        if not room:
            return False

        if room.owner is None:
            return None

        return room.owner.username

    async def send_error(self, message):
        await self.send(text_data=json.dumps({
            'action': 'error',
            'message': message
        }))
```

2.1.3 `GameConsumer`

Bases: `AsyncWebsocketConsumer`

WebSocket consumer for an active game session.

This is the main gameplay socket. Once connected, the client receives the current game state, can send player actions, and receives broadcasts of all actions and their resolved outcomes in real time.

How to Connect

Endpoint: `ws://<host>/ws/game/<game_id>/`

Authentication required: Yes. Unauthenticated connections are closed with code `4002`.

The `game_id` must match an existing `Game` that the authenticated user belongs to. If the user is not a participant, the connection is rejected with code `4003`.

Connection Lifecycle

1. Client opens the WebSocket with the `game_id` received from `PublicQueueConsumer` or `PrivateRoomConsumer`.
2. Server validates that the user is a participant.
3. On success, the server immediately sends a `game_state` event to the connecting client with the full current game state.
4. The client sends `Action` messages as the game progresses.
5. Each valid action is broadcast as a `game_action` event, followed by a `game_response` event with the resolved outcome.
6. All players in the game receive both broadcasts.

Messages: Client → Server

Game Action

Sends a player action for the current game phase. The `type` field must correspond to a valid `Action` type for the active phase. All other fields are action-specific.

```
{
    "type": "SomeActionType",
    "...": "action-specific fields"
}
```

The `game` and `player` fields are injected server-side — do **not** include them manually.

Chat Message

```
{
    "type": "ChatMessage",
    "msg": "Hello everyone!"
}
```

Messages: Server → Client

Game State

Sent immediately on connection. Contains the full serialized game state.

```
{
    "event_type": "game_state",
    "game_state": { "...": "full GameStatusSerializer output" }
}
```

Game Action

Broadcast to all players when a valid action is received. Contains the raw action data as sent by the acting player.

```
{
    "event_type": "game_action",
    "data": { "type": "SomeActionType", "game": 42, "player": 7, "...": "..." }
}
```

Game Response

Broadcast to all players immediately after `game_action`. Contains the resolved outcome of the action as serialized by `GeneralResponseSerializer`.

```
{
    "event_type": "game_response",
    "data": { "...": "GeneralResponseSerializer output" }
}
```

Chat Message

Broadcast to all players in the game.

```
{
    "event_type": "chat_message",
    "game": 42,
    "user": "alice",
    "msg": "Hello everyone!"
}
```

Error

Sent only to the client that triggered the error (invalid action, wrong phase, etc.).

```
{
    "event_type": "error",
    "message": "Acción no válida en la fase actual."
}
```

Close Codes

Code	Meaning
4002	Unauthorized or missing route kwargs.
4003	User is not a participant in this game.

Important Notes

- Every valid action triggers **two** consecutive broadcasts: `game_action` (what was sent) and `game_response` (the outcome). Always handle both.
- Invalid actions (wrong phase, serialization errors, game logic errors) produce an `error` event and are **not** broadcast to other players.
- The game state snapshot sent on connect may be slightly stale by the time the first `game_action` arrives; prefer the response stream for live updates.

Example Flow (JavaScript)

```
const socket = new WebSocket(`ws://localhost:8000/ws/game/${gameId}/`);

socket.onmessage = (event) => {
    const data = JSON.parse(event.data);

    switch (data.event_type) {
        case "game_state":
            initializeBoard(data.game_state);
            break;
        case "game_action":
            highlightAction(data.data);
            break;
        case "game_response":
            applyOutcome(data.data);
            break;
        case "chat_message":
            appendChat(data.user, data.msg);
            break;
        case "error":
            showError(data.message);
            break;
    }
};

// Send an action:
socket.send(JSON.stringify({ type: "RollDice" }));

// Send a chat message:
socket.send(JSON.stringify({ type: "ChatMessage", msg: "Good luck!" }));
```

- **Source code in `magnate/consumers.py`**

```
class GameConsumer(AsyncWebsocketConsumer):
    """
    WebSocket consumer for an active game session.

    This is the main gameplay socket. Once connected, the client receives the
    current game state, can send player actions, and receives broadcasts of
    all actions and their resolved outcomes in real time.

    ---
    ## How to Connect

    **Endpoint:** ``ws://<host>/ws/game/<game_id>/``
```

Authentication required: Yes. Unauthenticated connections are closed with code ``4002``.

The ``game_id`` must match an existing ``Game`` that the authenticated user
belongs to. If the user is not a participant, the connection is rejected with
code ``4003``.

Connection Lifecycle

1. Client opens the WebSocket with the ``game_id`` received from ``PublicQueueConsumer``
   or ``PrivateRoomConsumer``.
2. Server validates that the user is a participant.
3. On success, the server immediately sends a ``game_state`` event to the
   connecting client with the full current game state.
4. The client sends ``Action`` messages as the game progresses.
5. Each valid action is broadcast as a ``game_action`` event, followed by a
   ``game_response`` event with the resolved outcome.
6. All players in the game receive both broadcasts.

Messages: Client → Server

Game Action
Sends a player action for the current game phase. The ``type`` field must
correspond to a valid ``Action`` type for the active phase. All other fields
are action-specific.

```json
{
    "type": "SomeActionType",
    "...": "action-specific fields"
}
```

The ``game`` and ``player`` fields are injected server-side — do **not** include
them manually.

Chat Message
```json
{
    "type": "ChatMessage",
    "msg": "Hello everyone!"
}
```

Messages: Server → Client

Game State
Sent immediately on connection. Contains the full serialized game state.

```json
{
    "event_type": "game_state",
    "game_state": { "...": "full GameStatusSerializer output" }
}
```

Game Action

Broadcast to all players when a valid action is received. Contains the raw action data as sent by the acting player.

```json
{
    "event_type": "game_action",
    "data": { "type": "SomeActionType", "game": 42, "player": 7, "...": "..." }
}
```

Game Response
Broadcast to all players immediately after ``game_action``. Contains the resolved outcome of the action as serialized by ``GeneralResponseSerializer``.

```json
{
    "event_type": "game_response",
    "data": { "...": "GeneralResponseSerializer output" }
}
```

Chat Message
Broadcast to all players in the game.

```json
{
    "event_type": "chat_message",
    "game": 42,
    "user": "alice",
    "msg": "Hello everyone!"
}
```

Error
Sent only to the client that triggered the error (invalid action, wrong phase, etc.).

```json
{
    "event_type": "error",
    "message": "Acción no válida en la fase actual."
}
```

Close Codes

Code	Meaning
4002	Unauthorized or missing route kwargs.
4003	User is not a participant in this game.

Important Notes

- Every valid action triggers **two** consecutive broadcasts: ``game_action`` (what was sent) and ``game_response`` (the outcome). Always handle both.
- Invalid actions (wrong phase, serialization errors, game logic errors) produce an ``error`` event and are **not** broadcast to other players.

```
- The game state snapshot sent on connect may be slightly stale by the time
  the first ``game_action`` arrives; prefer the response stream for live updates.

---
## Example Flow (JavaScript)

```js
const socket = new WebSocket(`ws://localhost:8000/ws/game/${gameId}/`);

socket.onmessage = (event) => {
    const data = JSON.parse(event.data);

    switch (data.event_type) {
        case "game_state":
            initializeBoard(data.game_state);
            break;
        case "game_action":
            highlightAction(data.data);
            break;
        case "game_response":
            applyOutcome(data.data);
            break;
        case "chat_message":
            appendChat(data.user, data.msg);
            break;
        case "error":
            showError(data.message);
            break;
    }
};

// Send an action:
socket.send(JSON.stringify({ type: "RollDice" }));

// Send a chat message:
socket.send(JSON.stringify({ type: "ChatMessage", msg: "Good luck!" }));
```
"""
async def connect(self):
    # Triggered when user joins a specific match ID (game really begins) -> add to Redis room group.
    scope_user = self.scope.get('user')
    if scope_user is None or getattr(scope_user, 'is_anonymous', True):
        await self.close(code=4002)
        return

    # Extraemos la instancia real de CustomUser de la BD
    self.user = await database_sync_to_async(CustomUser.objects.get)(pk=scope_user.pk)

    self.url = self.scope.get('url_route')
    if self.url is None:
        await self.close(code=4002)
        return

    kwargs = self.url.get('kwargs')
    if not kwargs or 'room_id' not in kwargs:
        await self.close(code=4002)
        return

    self.game_id = int(kwargs['room_id'])
```

```
    self.game_group_name = f"game_{self.game_id}"

    player_is_in_game = await self.is_player_in_game(self.user, self.game_id)

    if not player_is_in_game:
        await self.close(code=4003)
        return

    await self.channel_layer.group_add(
        self.game_group_name,
        self.channel_name
    )

    await self.accept()

    game = await self.get_game()
    game_state = await database_sync_to_async(
            lambda: GameStatusSerializer(game).data)()

    await self.channel_layer.send(
        self.channel_name,
        {
            'type': 'game_state',
            'game_state': game_state
        }
    )

async def disconnect(self, close_code):
    # Triggered when user leaves game -> notify opponent.
    await self.channel_layer.group_discard(
        self.game_group_name,
        self.channel_name
    )


async def receive(self, text_data):
    """
    Triggered when user sends a move -> broadcast to room group.
    Also manages game over conditions triggering disconnects.
    Manages DB interactions over purchases, rents etc
    """

    game = await self.get_game()
    if game is None:
        await self.send_error("Game not found")
        return

    data = json.loads(text_data)

    if data.get('type') == 'ChatMessage':
        message = data.get('msg')
        if message:
            await self.channel_layer.group_send(
                self.game_group_name,
                {
                    'type': 'chat_message',
                    'game': self.game_id,
                    'user': self.user.username,
```

```
                        'msg': message
                    }
                )
            return

        data['game'] = self.game_id
        data['player'] = self.user.pk

        action, errors = await validate_and_save_action(data)

        if errors or action is None:
            await self.send_error(f"Invalid data: {errors}")
            return

        action = cast(Action, action)

        try:
            response = await GameManager.process_action(game, self.user, action)

            if response is None:
                await database_sync_to_async(action.delete)() # Limpiamos BD
                await self.send_error("Acción no válida en la fase actual.")
                return

            # Broadcast action
            await self.channel_layer.group_send(
                self.game_group_name,
                {
                    'type': 'game_action_event',
                    'data': data
                }
            )

            response_data = await database_sync_to_async(lambda: GeneralResponseSerializer(response).data)()

            await self.channel_layer.group_send(
                self.game_group_name,
                {
                    'type': 'game_response_event',
                    'data': response_data
                }
            )

        except (MaliciousUserInput, GameLogicError, GameDesignError, Exception) as e:
            await database_sync_to_async(action.delete)()
            await self.send_error(f"{e}")

# --------------------- Handlers ---------------------- #

    async def game_state(self, event):
        await self.send(text_data=json.dumps({
            'event_type': 'game_state',
            'game_state': event['game_state']
        }))

    async def game_action_event(self, event):
        await self.send(text_data=json.dumps({
            'event_type': 'game_action',
            'data': event['data']
```

```
        }))

    async def game_response_event(self, event):
        await self.send(text_data=json.dumps({
            'event_type': 'game_response',
            'data': event['data']
        }))

    async def chat_message(self, event):
        await self.send(text_data=json.dumps({
            'event_type': 'chat_message',
            'game': event['game'],
            'user': event['user'],
            'msg': event['msg']
        }))

    async def send_error(self, message):
        await self.send(text_data=json.dumps({
            'event_type': 'error',
            'message': message
        }))


#---------------------- DB access --------------------#
    @database_sync_to_async
    def is_player_in_game(self, user, game_id):
        try:
            game = Game.objects.get(pk=game_id)
            return game.players.filter(pk=user.pk).exists()
        except Game.DoesNotExist:
            return False

    @database_sync_to_async
    def get_game(self):
        try:
            return Game.objects.get(pk=self.game_id)
        except Game.DoesNotExist:
            return None
```

`receive(text_data)` `async`

Triggered when user sends a move -> broadcast to room group. Also manages game over conditions triggering disconnects. Manages DB interactions over purchases, rents etc

- **Source code in** `magnate/consumers.py`

```
async def receive(self, text_data):
    """
    Triggered when user sends a move -> broadcast to room group.
    Also manages game over conditions triggering disconnects.
    Manages DB interactions over purchases, rents etc
    """

    game = await self.get_game()
    if game is None:
        await self.send_error("Game not found")
        return
```

```
data = json.loads(text_data)

if data.get('type') == 'ChatMessage':
    message = data.get('msg')
    if message:
        await self.channel_layer.group_send(
            self.game_group_name,
            {
                'type': 'chat_message',
                'game': self.game_id,
                'user': self.user.username,
                'msg': message
            }
        )
    return

data['game'] = self.game_id
data['player'] = self.user.pk

action, errors = await validate_and_save_action(data)

if errors or action is None:
    await self.send_error(f"Invalid data: {errors}")
    return

action = cast(Action, action)

try:
    response = await GameManager.process_action(game, self.user, action)

    if response is None:
        await database_sync_to_async(action.delete)() # Limpiamos BD
        await self.send_error("Acción no válida en la fase actual.")
        return

    # Broadcast action
    await self.channel_layer.group_send(
        self.game_group_name,
        {
            'type': 'game_action_event',
            'data': data
        }
    )

    response_data = await database_sync_to_async(lambda: GeneralResponseSerializer(response).data)()

    await self.channel_layer.group_send(
        self.game_group_name,
        {
            'type': 'game_response_event',
            'data': response_data
        }
    )

except (MaliciousUserInput, GameLogicError, GameDesignError, Exception) as e:
    await database_sync_to_async(action.delete)()
    await self.send_error(f"{e}")
```

2.2 Actions

Game class

2.2.1 `Game`

Bases: `Model`

Represents the core state and data of a single "Magnate" game session.

Attributes:

Name	Type	Description
`datetime`	`DateTimeField`	The timestamp of when the game was created or started.
`positions`	`JSONField`	Maps a player's user ID (str/int) to their current square's custom ID (int).
`money`	`JSONField`	Maps a player's user ID (str/int) to their current money balance (int).
`active_phase_player`	`ForeignKey`	The user who must take action in the current micro-phase (e.g., the user who needs to respond to a trade, which might differ from the active_turn_player).
`active_turn_player`	`ForeignKey`	The user whose actual turn it is on the board.
`phase`	`CharField`	The current `GamePhase` of the game state machine.
`players`	`ManyToManyField`	The pool of users actively participating in this game.
`ordered_players`	`JSONField`	A list of player primary keys `[pk1, pk2, pk3, ...]` representing the strict turn order of the game.
`streak`	`IntegerField`	Tracks consecutive identical dice rolls (e.g., rolling doubles). Usually triggers jail time if it hits 3.
`possible_destinations`	`JSONField`	Maps a target `square_id` (str) to the `dice_combination` (int) required to get there. Used when a player has multiple routing options (e.g., taking a tram).
`parking_money`	`PositiveIntegerField`	The accumulated jackpot for landing on the "Free Parking" equivalent.
`jail_remaining_turns`	`JSONField`	Maps a player's user ID (str/int) to the number of turns (int) they have left to serve in jail.
`proposal`	`ForeignKey`	A reference to an active trade proposal (`ActionTradeProposal`) currently blocking the game state awaiting a response.
`fantasy_event`	`ForeignKey`	A reference to an active chance/community chest style event (`FantasyEvent`) currently being resolved.
`current_auction`	`ForeignKey`	A reference to an active property auction (`Auction`) taking place.
`finished`	`BooleanField`	Flag indicating if the game has concluded.
`bonus_response`	`ForeignKey`	Reference to a specific bonus or penalty modifier (`ResponseBonus`) applied to the current state.
`kick_out_task_id`	`CharField`	The ID of the scheduled Celery task responsible for kicking a player if they fail to act within the time limit.
`next_phase_task_id`	`CharField`	The ID of the scheduled Celery task responsible for auto-advancing the game to the next phase if a timeout occurs.
`current_turn`	`PositiveIntegerField`	The global counter for the number of turns that have elapsed.

- **Source code in** `magnate/models.py`

```
class Game(models.Model):
    """
    Represents the core state and data of a single "Magnate" game session.

    Attributes:
        datetime (DateTimeField): The timestamp of when the game was created or started.
        positions (JSONField): Maps a player's user ID (str/int) to their current square's custom ID (int).
        money (JSONField): Maps a player's user ID (str/int) to their current money balance (int).
        active_phase_player (ForeignKey): The user who must take action in the current micro-phase
            (e.g., the user who needs to respond to a trade, which might differ from the active_turn_player).
        active_turn_player (ForeignKey): The user whose actual turn it is on the board.
        phase (CharField): The current `GamePhase` of the game state machine.
        players (ManyToManyField): The pool of users actively participating in this game.
        ordered_players (JSONField): A list of player primary keys `[pk1, pk2, pk3, ...]`
            representing the strict turn order of the game.
        streak (IntegerField): Tracks consecutive identical dice rolls (e.g., rolling doubles).
            Usually triggers jail time if it hits 3.
        possible_destinations (JSONField): Maps a target `square_id` (str) to the `dice_combination` (int)
            required to get there. Used when a player has multiple routing options (e.g., taking a tram).
        parking_money (PositiveIntegerField): The accumulated jackpot for landing on the "Free Parking" equivalent.
        jail_remaining_turns (JSONField): Maps a player's user ID (str/int) to the number of turns (int)
            they have left to serve in jail.
        proposal (ForeignKey): A reference to an active trade proposal (`ActionTradeProposal`) currently
            blocking the game state awaiting a response.
        fantasy_event (ForeignKey): A reference to an active chance/community chest style event
            (`FantasyEvent`) currently being resolved.
        current_auction (ForeignKey): A reference to an active property auction (`Auction`) taking place.
        finished (BooleanField): Flag indicating if the game has concluded.
        bonus_response (ForeignKey): Reference to a specific bonus or penalty modifier (`ResponseBonus`)
            applied to the current state.
        kick_out_task_id (CharField): The ID of the scheduled Celery task responsible for kicking
            a player if they fail to act within the time limit.
        next_phase_task_id (CharField): The ID of the scheduled Celery task responsible for auto-advancing
            the game to the next phase if a timeout occurs.
        current_turn (PositiveIntegerField): The global counter for the number of turns that have elapsed.
    """
    datetime = models.DateTimeField()
    # Maps user_id -> square_custom_id
    positions = models.JSONField(default=dict, blank=True)
    # Maps user_id -> amount
    money = models.JSONField(default=dict, blank=True)
    active_phase_player = models.ForeignKey('CustomUser', on_delete=models.SET_NULL, null=True, related_name='phase_t
    active_turn_player = models.ForeignKey('CustomUser', on_delete=models.SET_NULL, null=True, related_name='turns_to

    class GamePhase(models.TextChoices):
        """
        Enumeration of the possible states (phases) within a game's turn cycle.

        Attributes:
            roll_the_dices: The initial phase of a turn. The `active_turn_player`
                must throw the dice to determine movement.
            choose_square: Triggered when a player's movement path presents a fork
                or routing option (e.g., deciding whether to take a tram/subway line).
                The player must select their specific destination square.
            choose_fantasy: Triggered when a player lands on a dynamic event square
                The player must acknowledge and resolve the active `fantasy_event`.
            management: Triggered when a player lands on an unowned property. The
```

```
            player must choose to either purchase the property at its list price
            or decline the purchase (which typically immediately triggers an `auction`).
        business: A versatile phase usually occurring at the end of a turn
            or before a roll. The player can build/demolish houses, mortgage/unmortgage
            properties, or finalize their board state before passing the turn.
        liquidation: An emergency phase triggered when a player owes a debt
            (to the bank or another player) that exceeds their current liquid cash.
            They are forced to sell assets or mortgage properties to cover the debt,
            or face bankruptcy.
        auction: A competitive, multi-player phase triggered when a property
            is declined in the `management` phase. The standard turn loop pauses, and
            players take turns placing bids until a winner is determined.
        proposal_acceptance: An interruptive phase triggered when one player
            sends a trade request to another. The game loop pauses, the
            `active_phase_player` switches to the recipient, and they must either
            accept or decline the pending `proposal`.
        end_game: A terminal state indicating the match has concluded, usually
            because all other players have gone bankrupt. No further actions can
            be taken.
    """
    roll_the_dices = 'roll_the_dices'
    choose_square = 'choose_square'
    choose_fantasy = 'choose_fantasy'
    management = 'management'
    liquidation = 'liquidation'
    business = 'business'
    auction = 'auction'
    proposal_acceptance = 'proposal_acceptance'
    end_game = 'end_game'


phase = models.CharField(choices=GamePhase, max_length=20, default='roll_the_dices')
players = models.ManyToManyField('CustomUser', related_name='active_playing')
# ordered_player = [pk1, pk2, pk3, ...]
ordered_players = models.JSONField(default=list)
streak = models.IntegerField(default=0)
#dict[string,int], key=square_id, value=dice_combination to get there
possible_destinations = models.JSONField(default=dict, blank=True)
parking_money = models.PositiveIntegerField(default=0)
# Maps user_id -> uint
jail_remaining_turns = models.JSONField(default=dict, blank=True)
proposal = models.ForeignKey('ActionTradeProposal', on_delete=models.SET_NULL, null=True, blank=True, related_nar

fantasy_event = models.ForeignKey('FantasyEvent', on_delete=models.SET_NULL, null=True, blank=True, related_name=

current_auction = models.ForeignKey('Auction', on_delete=models.SET_NULL, null=True, blank=True, related_name='ac

finished = models.BooleanField(default=False)
bonus_response = models.ForeignKey('ResponseBonus', on_delete=models.SET_NULL, null=True, blank=True, related_nar

kick_out_task_id = models.CharField(max_length=255, null=True, blank=True)
next_phase_task_id = models.CharField(max_length=255, null=True, blank=True)

current_turn = models.PositiveIntegerField(default=1)
```

`GamePhase`

Bases: `TextChoices`

Enumeration of the possible states (phases) within a game's turn cycle.

Attributes:

Name	Type	Description
`roll_the_dices`		The initial phase of a turn. The `active_turn_player` must throw the dice to determine movement.
`choose_square`		Triggered when a player's movement path presents a fork or routing option (e.g., deciding whether to take a tram/subway line). The player must select their specific destination square.
`choose_fantasy`		Triggered when a player lands on a dynamic event square The player must acknowledge and resolve the active `fantasy_event`.
`management`		Triggered when a player lands on an unowned property. The player must choose to either purchase the property at its list price or decline the purchase (which typically immediately triggers an `auction`).
`business`		A versatile phase usually occurring at the end of a turn or before a roll. The player can build/demolish houses, mortgage/unmortgage properties, or finalize their board state before passing the turn.
`liquidation`		An emergency phase triggered when a player owes a debt (to the bank or another player) that exceeds their current liquid cash. They are forced to sell assets or mortgage properties to cover the debt, or face bankruptcy.
`auction`		A competitive, multi-player phase triggered when a property is declined in the `management` phase. The standard turn loop pauses, and players take turns placing bids until a winner is determined.
`proposal_acceptance`		An interruptive phase triggered when one player sends a trade request to another. The game loop pauses, the `active_phase_player` switches to the recipient, and they must either accept or decline the pending `proposal`.
`end_game`		A terminal state indicating the match has concluded, usually because all other players have gone bankrupt. No further actions can be taken.

- **Source code in `magnate/models.py`**

```
class GamePhase(models.TextChoices):
    """
    Enumeration of the possible states (phases) within a game's turn cycle.

    Attributes:
        roll_the_dices: The initial phase of a turn. The `active_turn_player`
            must throw the dice to determine movement.
        choose_square: Triggered when a player's movement path presents a fork
            or routing option (e.g., deciding whether to take a tram/subway line).
            The player must select their specific destination square.
        choose_fantasy: Triggered when a player lands on a dynamic event square
            The player must acknowledge and resolve the active `fantasy_event`.
        management: Triggered when a player lands on an unowned property. The
            player must choose to either purchase the property at its list price
            or decline the purchase (which typically immediately triggers an `auction`).
        business: A versatile phase usually occurring at the end of a turn
```

```
            or before a roll. The player can build/demolish houses, mortgage/unmortgage
            properties, or finalize their board state before passing the turn.
        liquidation: An emergency phase triggered when a player owes a debt
            (to the bank or another player) that exceeds their current liquid cash.
            They are forced to sell assets or mortgage properties to cover the debt,
            or face bankruptcy.
        auction: A competitive, multi-player phase triggered when a property
            is declined in the `management` phase. The standard turn loop pauses, and
            players take turns placing bids until a winner is determined.
        proposal_acceptance: An interruptive phase triggered when one player
            sends a trade request to another. The game loop pauses, the
            `active_phase_player` switches to the recipient, and they must either
            accept or decline the pending `proposal`.
        end_game: A terminal state indicating the match has concluded, usually
            because all other players have gone bankrupt. No further actions can
            be taken.
    """
    roll_the_dices = 'roll_the_dices'
    choose_square = 'choose_square'
    choose_fantasy = 'choose_fantasy'
    management = 'management'
    liquidation = 'liquidation'
    business = 'business'
    auction = 'auction'
    proposal_acceptance = 'proposal_acceptance'
    end_game = 'end_game'
```

2.2.2 `GameStatusSerializer`

Bases: `ModelSerializer`

- **Example**
  A standard serialized response during the 'roll_the_dices' phase:

```
{
    "id": 1,
    "datetime": "2026-04-06T18:30:00Z",
    "positions": {"42": 0, "85": 12},
    "money": {"42": 1500, "85": 1350},
    "active_phase_player": 42,
    "active_turn_player": 42,
    "phase": "roll_the_dices",
    "players": [42, 85],
    "ordered_players": [42, 85],
    "streak": 0,
    "possible_destinations": {},
    "parking_money": 200,
    "jail_remaining_turns": {},
    "proposal": null,
    "fantasy_event": null,
    "current_auction": null,
    "finished": false,
    "bonus_response": null,
```

```
    "current_turn": 5
}
```

- **Source code in** `magnate/serializers.py`

```
class GameStatusSerializer(serializers.ModelSerializer):
    """
    Example:
        A standard serialized response during the 'roll_the_dices' phase:
        ```json
        {
            "id": 1,
            "datetime": "2026-04-06T18:30:00Z",
            "positions": {"42": 0, "85": 12},
            "money": {"42": 1500, "85": 1350},
            "active_phase_player": 42,
            "active_turn_player": 42,
            "phase": "roll_the_dices",
            "players": [42, 85],
            "ordered_players": [42, 85],
            "streak": 0,
            "possible_destinations": {},
            "parking_money": 200,
            "jail_remaining_turns": {},
            "proposal": null,
            "fantasy_event": null,
            "current_auction": null,
            "finished": false,
            "bonus_response": null,
            "current_turn": 5
        }
        ```
    """
    class Meta:
        model = Game
        exclude = ['kick_out_task_id', 'next_phase_task_id']
```

2.3 Actions

2.3.1 `Action`

Bases: `Model`

Represents generic action, so every action shares these fields

Frontend Request Payload Example:

```
{
  "type": "Action",
  "game": 1,
  "player": 2,
}
```

- **Source code in** `magnate/models.py`

```
class Action(models.Model):
    """
    Represents generic action, so every action shares these fields

    Frontend Request Payload Example:
    ```json
    {
      "type": "Action",
      "game": 1,
      "player": 2,
    }
    ```
    """
    game = models.ForeignKey('Game', on_delete=models.CASCADE, related_name='in_game')
    player = models.ForeignKey('CustomUser', on_delete=models.CASCADE, related_name='made_by')
```

2.3.2 `ActionThrowDices`

Bases: `Action`

Action to throw the dices. It is basically empty.

Frontend Request Payload Example:

```
{
  "type": "ActionThrowDices",
  "game": 1,
  "player": 2,
}
```

- **Source code in** `magnate/models.py`

```
class ActionThrowDices(Action):
    """
    Action to throw the dices. It is basically empty.

    Frontend Request Payload Example:
```

```
    ```json
    {
      "type": "ActionThrowDices",
      "game": 1,
      "player": 2,
    }
    ```
    """
    pass
```

2.3.3 `ActionMoveTo`

Bases: `Action`

Action to move to a square.

Frontend Request Payload Example:

```
{
  "type": "ActionThrowDices",
  "game": 1,
  "player": 2,
  "square": 101,
}
```

- **Source code in `magnate/models.py`**

```
class ActionMoveTo(Action):
    """
    Action to move to a square.

    Frontend Request Payload Example:
    ```json
    {
      "type": "ActionThrowDices",
      "game": 1,
      "player": 2,
      "square": 101,
    }
    ```
    """
    square = models.ForeignKey('BaseSquare', on_delete=models.CASCADE, related_name='move_to')
```

2.3.4 `ActionTakeTram`

Bases: `Action`

Action to take the tram when possible

Frontend Request Payload Example:

```
{
  "type": "ActionTakeTram",
```

```
  "game": 1,
  "player": 2,
  "square": 200,
}
```

- **Source code in** `magnate/models.py`

```
class ActionTakeTram(Action):
    """
    Action to take the tram when possible

    Frontend Request Payload Example:
    ```json
    {
      "type": "ActionTakeTram",
      "game": 1,
      "player": 2,
      "square": 200,
    }
    ```
    """
    square = models.ForeignKey('BaseSquare', on_delete=models.CASCADE, related_name='tram_move_to')
```

2.3.5 `ActionDropPurchase`

Bases: `Action`

Action to decline purchasing a property.

Frontend Request Payload Example:

```
{
  "type": "ActionDropPurchase",
  "game": 1,
  "player": 2,
  "square": 15
}
```

- **Source code in** `magnate/models.py`

```
class ActionDropPurchase(Action):
    """
    Action to decline purchasing a property.

    Frontend Request Payload Example:
    ```json
    {
      "type": "ActionDropPurchase",
      "game": 1,
      "player": 2,
      "square": 15
    }
    ```
```

```
    """
    square = models.ForeignKey('BaseSquare', on_delete=models.CASCADE, related_name='dropped')
```

2.3.6 `ActionBuySquare`

Bases: `Action`

Action to buy an unowned property.

Frontend Request Payload Example:

```
{
  "type": "ActionBuySquare",
  "game": 1,
  "player": 2,
  "square": 15
}
```

- **Source code in `magnate/models.py`**

```
class ActionBuySquare(Action):
    """
    Action to buy an unowned property.

    Frontend Request Payload Example:
    ```json
    {
      "type": "ActionBuySquare",
      "game": 1,
      "player": 2,
      "square": 15
    }
    ```
    """
    square = models.ForeignKey('BaseSquare', on_delete=models.CASCADE, related_name='bought')
```

2.3.7 `ActionSellSquare`

Bases: `Action`

Action to sell a property.

Frontend Request Payload Example:

```
{
  "type": "ActionSellSquare",
  "game": 1,
  "player": 2,
  "square": 15
}
```

- **Source code in** `magnate/models.py`

```
class ActionSellSquare(Action):
    """
    Action to sell a property.

    Frontend Request Payload Example:
    ```json
    {
      "type": "ActionSellSquare",
      "game": 1,
      "player": 2,
      "square": 15
    }
    ```
    """
    square = models.ForeignKey('BaseSquare', on_delete=models.CASCADE, related_name='sold')
```

2.3.8 `ActionBuild`

Bases: `Action`

Action to build houses/hotels on a property.

Frontend Request Payload Example:

```
{
  "type": "ActionBuild",
  "game": 1,
  "player": 2,
  "houses": 1,
  "square": 12
}
```

- **Source code in** `magnate/models.py`

```
class ActionBuild(Action):
    """
    Action to build houses/hotels on a property.

    Frontend Request Payload Example:
    ```json
    {
      "type": "ActionBuild",
      "game": 1,
      "player": 2,
      "houses": 1,
      "square": 12
    }
    ```
    """
```

```
    houses = models.IntegerField(default=1)
    square = models.ForeignKey('BaseSquare', on_delete=models.CASCADE, related_name='build_square')
```

2.3.9 `ActionDemolish`

Bases: `Action`

Action to demolish houses/hotels on a property.

Frontend Request Payload Example:

```
{
  "type": "ActionDemolish",
  "game": 1,
  "player": 2,
  "houses": 1,
  "square": 12
}
```

- **Source code in `magnate/models.py`**

```
class ActionDemolish(Action):
    """
    Action to demolish houses/hotels on a property.

    Frontend Request Payload Example:
    ```json
    {
      "type": "ActionDemolish",
      "game": 1,
      "player": 2,
      "houses": 1,
      "square": 12
    }
    ```
    """
    houses = models.IntegerField(default=0)
    square = models.ForeignKey('BaseSquare', on_delete=models.CASCADE, related_name='demolish_square')
```

2.3.10 `ActionChooseCard`

Bases: `Action`

Action to interact with or draw a fantasy card.

Frontend Request Payload Example:

```
{
  "type": "ActionChooseCard",
  "game": 1,
  "player": 2,
  "chosen_revealed_card": true
}
```

- **Source code in** `magnate/models.py`

```
class ActionChooseCard(Action):
    """
    Action to interact with or draw a fantasy card.

    Frontend Request Payload Example:
    ```json
    {
      "type": "ActionChooseCard",
      "game": 1,
      "player": 2,
      "chosen_revealed_card": true
    }
    ```
    """
    chosen_revealed_card = models.BooleanField(default=False)
```

2.3.11 `ActionSurrender`

Bases: `Action`

Action to quit/surrender the game. It is empty.

Frontend Request Payload Example:

```
{
  "type": "ActionSurrender",
  "game": 1,
  "player": 2
}
```

- **Source code in** `magnate/models.py`

```
class ActionSurrender(Action):
    """
    Action to quit/surrender the game. It is empty.

    Frontend Request Payload Example:
    ```json
    {
      "type": "ActionSurrender",
      "game": 1,
      "player": 2
    }
    ```
    """
    pass
```

2.3.12 `ActionTradeProposal`

Bases: `Action`

Action to propose a trade to another player.

Frontend Request Payload Example:

```
{
  "type": "ActionTradeProposal",
  "game": 1,
  "player": 2,
  "destination_user": 3,
  "offered_money": 200,
  "asked_money": 0,
  "offered_properties": [5, 6],
  "asked_properties": [12]
}
```

- **Source code in** `magnate/models.py`

```
class ActionTradeProposal(Action):
    """
    Action to propose a trade to another player.

    Frontend Request Payload Example:
    ```json
    {
      "type": "ActionTradeProposal",
      "game": 1,
      "player": 2,
      "destination_user": 3,
      "offered_money": 200,
      "asked_money": 0,
      "offered_properties": [5, 6],
      "asked_properties": [12]
    }
    ```
    """
    destination_user = models.ForeignKey('CustomUser', on_delete=models.CASCADE, related_name='destination_user')
    offered_money = models.PositiveIntegerField(default=0)
    asked_money = models.PositiveIntegerField(default=0)
    offered_properties = models.ManyToManyField('PropertyRelationship', related_name='offered_properties')
    asked_properties = models.ManyToManyField('PropertyRelationship', related_name='asked_properties')
```

2.3.13 `ActionTradeAnswer`

Bases: `Action`

Action to accept or decline a trade proposal.

Frontend Request Payload Example:

```
{
  "type": "ActionTradeAnswer",
  "game": 1,
  "player": 2,
  "choose": true,
```

```
  "proposal": 8
}
```

- **Source code in** `magnate/models.py`

```
class ActionTradeAnswer(Action):
    """
    Action to accept or decline a trade proposal.

    Frontend Request Payload Example:
    ```json
    {
      "type": "ActionTradeAnswer",
      "game": 1,
      "player": 2,
      "choose": true,
      "proposal": 8
    }
    ```
    """
    choose = models.BooleanField(default=False)
    proposal = models.OneToOneField('ActionTradeProposal', on_delete=models.CASCADE, related_name='proposal')
```

2.3.14 `ActionMortgageSet`

Bases: `Action`

Action to mortgage a property.

Frontend Request Payload Example:

```
{
  "type": "ActionMortgageSet",
  "game": 1,
  "player": 2,
  "square": 15
}
```

- **Source code in** `magnate/models.py`

```
class ActionMortgageSet(Action):
    """
    Action to mortgage a property.

    Frontend Request Payload Example:
    ```json
    {
      "type": "ActionMortgageSet",
      "game": 1,
      "player": 2,
      "square": 15
    }
    ```
```

```
    """
    square = models.ForeignKey('BaseSquare', on_delete=models.CASCADE, related_name='mortgage_set_square')
```

2.3.15 `ActionMortgageUnset`

Bases: `Action`

Action to lift a mortgage from a property.

Frontend Request Payload Example:

```
{
  "type": "ActionMortgageUnset",
  "game": 1,
  "player": 2,
  "square": 15
}
```

- **Source code in `magnate/models.py`**

```
class ActionMortgageUnset(Action):
    """
    Action to lift a mortgage from a property.

    Frontend Request Payload Example:
    ```json
    {
      "type": "ActionMortgageUnset",
      "game": 1,
      "player": 2,
      "square": 15
    }
    ```
    """
    square = models.ForeignKey('BaseSquare', on_delete=models.CASCADE, related_name='mortgage_unset_square')
```

2.3.16 `ActionPayBail`

Bases: `Action`

Action to pay the bail fee to get out of jail. It is empty.

Frontend Request Payload Example:

```
{
  "type": "ActionPayBail",
  "game": 1,
  "player": 2
}
```

- **Source code in** `magnate/models.py`

```
class ActionPayBail(Action):
    """
    Action to pay the bail fee to get out of jail. It is empty.

    Frontend Request Payload Example:
    ```json
    {
      "type": "ActionPayBail",
      "game": 1,
      "player": 2
    }
    ```
    """
    pass
```

2.3.17 `ActionNextPhase`

Bases: `Action`

Action to transition to the next game phase or end turn. It is empty.

Frontend Request Payload Example:

```
{
  "type": "ActionNextPhase",
  "game": 1,
  "player": 2
}
```

- **Source code in** `magnate/models.py`

```
class ActionNextPhase(Action):
    """
    Action to transition to the next game phase or end turn. It is empty.

    Frontend Request Payload Example:
    ```json
    {
      "type": "ActionNextPhase",
      "game": 1,
      "player": 2
    }
    ```
    """
    pass
```

2.3.18 `ActionBid`

Bases: `Action`

Action to place a bid on a property auction.

Frontend Request Payload Example:

```
{
  "type": "ActionBid",
  "game": 1,
  "player": 2,
  "auction": 4,
  "amount": 150
}
```

- **Source code in** `magnate/models.py`

```
class ActionBid(Action):
    """
    Action to place a bid on a property auction.

    Frontend Request Payload Example:
    ```json
    {
      "type": "ActionBid",
      "game": 1,
      "player": 2,
      "auction": 4,
      "amount": 150
    }
    ```
    """
    amount = models.PositiveIntegerField(default=0)
```

2.4 Responses

2.4.1 `Response`

Bases: `Model`

Base model for game state updates broadcasted to the frontend.

Frontend Response Payload Example:

```
{
  "type": "Response",
  "money": {"1": 1500, "2": 1200},
  "active_phase_player": 2,
  "active_turn_player": 2,
  "phase": "management"
}
```

- **Source code in** `magnate/models.py`

```
class Response(models.Model):
    """
    Base model for game state updates broadcasted to the frontend.

    Frontend Response Payload Example:
    ```json
    {
      "type": "Response",
      "money": {"1": 1500, "2": 1200},
      "active_phase_player": 2,
      "active_turn_player": 2,
      "phase": "management"
    }
    ```
    """
    money = models.JSONField(default=dict, blank=True)
    active_phase_player = models.ForeignKey('CustomUser', on_delete=models.SET_NULL, null=True, related_name='respons
    active_turn_player = models.ForeignKey('CustomUser', on_delete=models.SET_NULL, null=True, related_name='response
    phase = models.CharField(choices=Game.GamePhase, max_length=20)
    positions = models.JSONField(default=dict, blank=True)
```

2.4.2 `ResponseMovement`

Bases: `Response`

Base response for an event that moves a player across the board.

Frontend Response Payload Example:

```
{
  "type": "ResponseMovement",
  "money": {"1": 1500, "2": 1200},
  "active_phase_player": 2,
  "active_turn_player": 2,
  "phase": "moving",
```

```
  "path": [10, 11, 12, 13, 14, 15],
  "fantasy_event": null
}
```

- **Source code in** `magnate/models.py`

```
class ResponseMovement(Response):
    """
    Base response for an event that moves a player across the board.

    Frontend Response Payload Example:
    ```json
    {
      "type": "ResponseMovement",
      "money": {"1": 1500, "2": 1200},
      "active_phase_player": 2,
      "active_turn_player": 2,
      "phase": "moving",
      "path": [10, 11, 12, 13, 14, 15],
      "fantasy_event": null
    }
    ```
    """
    path = models.JSONField(default=list, blank=True)
    fantasy_event = models.ForeignKey('FantasyEvent', on_delete=models.CASCADE, null=True, blank=True)
```

2.4.3 `ResponseThrowDices`

Bases: `ResponseMovement`

Response detailing the result of a dice roll and valid movement destinations.

Frontend Response Payload Example:

```
{
  "type": "ResponseThrowDices",
  "money": {"1": 1500, "2": 1200},
  "active_phase_player": 2,
  "active_turn_player": 2,
  "phase": "moving",
  "path": [10, 11, 12],
  "fantasy_event": null,
  "dice1": 4,
  "dice2": 3,
  "dice_bus": 1,
  "destinations": [17],
  "triple": false,
  "streak": 0
}
```

- **Source code in** `magnate/models.py`

```
class ResponseThrowDices(ResponseMovement):
    """
    Response detailing the result of a dice roll and valid movement destinations.
```

```
    Frontend Response Payload Example:
    ```json
    {
      "type": "ResponseThrowDices",
      "money": {"1": 1500, "2": 1200},
      "active_phase_player": 2,
      "active_turn_player": 2,
      "phase": "moving",
      "path": [10, 11, 12],
      "fantasy_event": null,
      "dice1": 4,
      "dice2": 3,
      "dice_bus": 1,
      "destinations": [17],
      "triple": false,
      "streak": 0
    }
    ```
    """
    dice1 = models.PositiveIntegerField(default=0)
    dice2 = models.PositiveIntegerField(default=0)
    dice_bus = models.PositiveIntegerField(default=0)
    destinations = models.JSONField(default=list, blank=True)
    triple = models.BooleanField(default=False)
    streak = models.IntegerField(default=0)
```

2.4.4 `ResponseChooseSquare`

Bases: `ResponseMovement`

Response confirming a player's movement to a specifically chosen square.

Frontend Response Payload Example:

```
{
  "type": "ResponseChooseSquare",
  "money": {"1": 1500, "2": 1200},
  "active_phase_player": 2,
  "active_turn_player": 2,
  "phase": "moving",
  "path": [10, 25],
  "fantasy_event": null
}
```

- **Source code in** `magnate/models.py`

```
class ResponseChooseSquare(ResponseMovement):
    """
    Response confirming a player's movement to a specifically chosen square.

    Frontend Response Payload Example:
    ```json
    {
      "type": "ResponseChooseSquare",
      "money": {"1": 1500, "2": 1200},
```
```

```
      "active_phase_player": 2,
      "active_turn_player": 2,
      "phase": "moving",
      "path": [10, 25],
      "fantasy_event": null
    }
    ```
    """
    pass
```

### 2.4.5 `ResponseChooseFantasy`

Bases: `Response`

Response delivering the result of a drawn Chance/Community Chest card.

Frontend Response Payload Example:

```
{
  "type": "ResponseChooseFantasy",
  "money": {"1": 1600, "2": 1200},
  "active_phase_player": 2,
  "active_turn_player": 2,
  "phase": "management",
  "fantasy_event": 5
}
```

- **Source code in** `magnate/models.py`

```
class ResponseChooseFantasy(Response):
    """
    Response delivering the result of a drawn Chance/Community Chest card.

    Frontend Response Payload Example:
    ```json
    {
      "type": "ResponseChooseFantasy",
      "money": {"1": 1600, "2": 1200},
      "active_phase_player": 2,
      "active_turn_player": 2,
      "phase": "management",
      "fantasy_event": 5
    }
    ```
    """
    fantasy_event = models.ForeignKey('FantasyEvent', on_delete=models.CASCADE, null=True, blank=True)
```

### 2.4.6 `ResponseAuction`

Bases: `Response`

Response updating the state of an ongoing or completed auction. Note: The @property decorators will likely be serialized as regular fields.
```

Frontend Response Payload Example:

```
{
  "type": "ResponseAuction",
  "money": {"1": 1150, "2": 1200},
  "active_phase_player": 1,
  "active_turn_player": 2,
  "phase": "management",
  "auction": 12,
  "winner": 1,
  "final_amount": 350,
  "is_tie": false
}
```

- **Source code in** `magnate/models.py`

```
class ResponseAuction(Response):
    """
    Response updating the state of an ongoing or completed auction.
    Note: The @property decorators will likely be serialized as regular fields.

    Frontend Response Payload Example:
    ```json
    {
      "type": "ResponseAuction",
      "money": {"1": 1150, "2": 1200},
      "active_phase_player": 1,
      "active_turn_player": 2,
      "phase": "management",
      "auction": 12,
      "winner": 1,
      "final_amount": 350,
      "is_tie": false
    }
    ```
    """
    auction = models.OneToOneField('Auction', on_delete=models.CASCADE, related_name='response')

    @property
    def winner(self):
        return self.auction.winner

    @property
    def final_amount(self):
        return self.auction.final_amount

    @property
    def is_tie(self):
        return self.auction.is_tie
```

2.5 Fantasy events

2.5.1 `FantasyEvent`

Bases: `Model`

Fantasy events are triggered when a player lands on a fantasy square. One of these effects takes place:

- winPlainMoney: Grants a flat amount of money. Cost: 130 | Possible values: [20, 60, 120, 150, 200]
- winRatioMoney: Increases money by a percentage ratio. Cost: 500 | Possible values: [1, 2, 5, 10]
- losePlainMoney: Deducts a flat amount of money. Cost: 80 | Possible values: [40, 80, 120, 150, 200]
- loseRatioMoney: Deducts money by a percentage ratio. Cost: 30 | Possible values: [1%, 2%, 5%, 10%]
- breakOpponentHouse: Demolishes a house on an opponent's property. Cost: 150
- breakOwnHouse: Demolishes a house on the player's own property. Cost: 30
- shufflePositions: Moves all players to random squares. Cost: 50
- moveAnywhereRandom: Moves the player to a random square. Cost: 50
- moveOpponentAnywhereRandom: Moves a random opponent to a random square. Cost: 60
- shareMoneyAll: Gives a specific amount of the player's money to all opponents. Cost: 5 | Possible values: [20, 30, 50]
- freeHouse: Builds a free house on one of the player's properties. Cost: 80
- goToJail: Sends the player directly to jail. Cost: 25
- sendToJail: Sends a random opponent directly to jail. Cost: 80
- everybodyToJail: Sends all players to jail. Cost: 50
- doubleOrNothing: 50/50 chance to either double current money or lose it all. Cost: 50
- getParkingMoney: Awards the accumulated parking money to the player. Cost: 500
- reviveProperty: Unmortgages a property for free (deshipotecas gratis). Cost: 100
- earthquake: Every developed property on the board loses one house (todas las calles pierden una casa). Cost: 200
- everybodySendsYouMoney: Every opponent pays a specific amount to the player. Cost: 120 | Possible values: [20, 30, 50]
- magnetism: Pulls all other players to the player's current square (todo el mundo va a ti). Cost: 100
- goToStart: Sends the player directly to the Start (Exit) square, collecting the pass-go money. Cost: 90

Frontend Fantasy Payload Example:

```
{
  "type":"win_plain_money",
  "values":[20,60,120,150,200],
  "cost": 130
}
```

- **Source code in** `magnate/models.py`

```
class FantasyEvent(models.Model):
    """
    Fantasy events are triggered when a player lands on a fantasy square. One
    of these effects takes place:

    - winPlainMoney: Grants a flat amount of money.
        Cost: 130 | Possible values: [20, 60, 120, 150, 200]
    - winRatioMoney: Increases money by a percentage ratio.
        Cost: 500 | Possible values: [1, 2, 5, 10]
```

```
- losePlainMoney: Deducts a flat amount of money.
    Cost: 80 | Possible values: [40, 80, 120, 150, 200]
- loseRatioMoney: Deducts money by a percentage ratio.
    Cost: 30 | Possible values: [1%, 2%, 5%, 10%]
- breakOpponentHouse: Demolishes a house on an opponent's property.
    Cost: 150
- breakOwnHouse: Demolishes a house on the player's own property.
    Cost: 30
- shufflePositions: Moves all players to random squares.
    Cost: 50
- moveAnywhereRandom: Moves the player to a random square.
    Cost: 50
- moveOpponentAnywhereRandom: Moves a random opponent to a random square.
    Cost: 60
- shareMoneyAll: Gives a specific amount of the player's money to all opponents.
    Cost: 5 | Possible values: [20, 30, 50]
- freeHouse: Builds a free house on one of the player's properties.
    Cost: 80
- goToJail: Sends the player directly to jail.
    Cost: 25
- sendToJail: Sends a random opponent directly to jail.
    Cost: 80
- everybodyToJail: Sends all players to jail.
    Cost: 50
- doubleOrNothing: 50/50 chance to either double current money or lose it all.
    Cost: 50
- getParkingMoney: Awards the accumulated parking money to the player.
    Cost: 500
- reviveProperty: Unmortgages a property for free (deshipotecas gratis).
    Cost: 100
- earthquake: Every developed property on the board loses one house (todas las calles pierden una casa).
    Cost: 200
- everybodySendsYouMoney: Every opponent pays a specific amount to the player.
    Cost: 120 | Possible values: [20, 30, 50]
- magnetism: Pulls all other players to the player's current square (todo el mundo va a ti).
    Cost: 100
- goToStart: Sends the player directly to the Start (Exit) square, collecting the pass-go money.
    Cost: 90

Frontend Fantasy Payload Example:
```json
{
  "type":"win_plain_money",
  "values":[20,60,120,150,200],
  "cost": 130
}
```
"""
class FantasyType(models.TextChoices):
    winPlainMoney = 'winPlainMoney',
    winRatioMoney = 'winRatioMoney',
    losePlainMoney = 'losePlainMoney',
    loseRatioMoney = 'loseRatioMoney',
    breakOpponentHouse = 'breakOpponentHouse',
    breakOwnHouse = 'breakOwnHouse',
    shufflePositions = 'shufflePositions',
    moveAnywhereRandom = 'moveAnywhereRandom',
    moveOpponentAnywhereRandom = 'moveOpponentAnywhereRandom',
    shareMoneyAll = 'shareMoneyAll',
```

```
        freeHouse = 'freeHouse',
        goToJail = 'goToJail',
        sendToJail = 'sendToJail',
        everybodyToJail = 'everybodyToJail',
        doubleOrNothing = 'doubleOrNothing',
        getParkingMoney = 'getParkingMoney',
        reviveProperty = 'reviveProperty',
        earthquake = 'earthquake',
        everybodySendsYouMoney = 'everybodySendsYouMoney',
        magnetism = 'magnetism',
        goToStart = 'goToStart'


    fantasy_type = models.CharField(choices=FantasyType, max_length=40)
    values = models.JSONField(null=True)
    card_cost = models.IntegerField(default=0)
```

3. Technical reference

3.1 Core Game Logic (games.py)

This module encapsulates the rules, state transitions, and actions of the game. It functions as the authoritative backend engine, ensuring that all moves are valid according to the game state.

3.1.1 🎲 Game State Machine Flow

The game operates as a finite state machine where players transition through specific phases. Any action sent by the frontend must be authorized by the `GameManager` based on the current active phase.

3.1.2 🛠️ Utility & Data Functions

These functions handle the database-level interactions required to resolve game logic.

Core Game Logic Module.

This module handles the rules, state transitions, and actions of the game. It provides helper functions to calculate net worth, rent, building/demolishing rules, and a GameManager class that acts as a state machine for the different phases of a player's turn.

3.1.3 `_get_square_by_custom_id(custom_id)`

Retrieves a BaseSquare instance by its custom_id.

Parameters:

Name	Type	Description	Default
`custom_id`	`int`	The custom identifier of the square.	*required*

Returns:

Name	Type	Description
`BaseSquare`	`BaseSquare`	The square instance.

Raises:

Type	Description
`GameLogicError`	If no square with the given custom_id exists.

- **Source code in** `magnate/game_utils.py`

```
def _get_square_by_custom_id(custom_id: int) -> BaseSquare:
    """
    Retrieves a BaseSquare instance by its custom_id.

    Args:
        custom_id (int): The custom identifier of the square.

    Returns:
        BaseSquare: The square instance.

    Raises:
```

```
        GameLogicError: If no square with the given custom_id exists.
    """
    square = BaseSquare.objects.filter(custom_id=custom_id).first()
    if square is None:
        raise GameLogicError(f"no square with id {custom_id}")

    return square
```

3.1.4 `_get_user_square(game, user)`

Retrieves the square where a specific user is currently located.

Parameters:

Name	Type	Description	Default
`game`	`Game`	The current game instance.	*required*
`user`	`CustomUser`	The user whose position is being queried.	*required*

Returns:

Name	Type	Description
`BaseSquare`	`BaseSquare`	The square where the user is currently standing.

Raises:

Type	Description
`GameLogicError`	If the user is not part of the game.

- **Source code in `magnate/game_utils.py`**

```
def _get_user_square(game: Game, user: CustomUser) -> BaseSquare:
    """
    Retrieves the square where a specific user is currently located.

    Args:
        game (Game): The current game instance.
        user (CustomUser): The user whose position is being queried.

    Returns:
        BaseSquare: The square where the user is currently standing.

    Raises:
        GameLogicError: If the user is not part of the game.
    """
    user_key = str(user.pk)

    if user_key not in game.positions:
        raise GameLogicError(f"user {user} not in the game")

    return _get_square_by_custom_id(game.positions[user_key])
```

3.1.5 `_get_relationship(game, square)`

Gets the ownership relationship between a game and a specific square.

Parameters:

Name	Type	Description	Default
`game`	`Game`	The current game instance.	*required*
`square`	`BaseSquare`	The property square to check.	*required*

Returns:

Type	Description
`Optional[PropertyRelationship]`	Optional[PropertyRelationship]: The relationship object if owned, None otherwise.

Raises:

Type	Description
`GameLogicError`	If more than one owner is found for the same square.

- **Source code in `magnate/game_utils.py`**

```
def _get_relationship(game: Game, square: BaseSquare) -> Optional[PropertyRelationship]:
    """
    Gets the ownership relationship between a game and a specific square.

    Args:
        game (Game): The current game instance.
        square (BaseSquare): The property square to check.

    Returns:
        Optional[PropertyRelationship]: The relationship object if owned, None otherwise.

    Raises:
        GameLogicError: If more than one owner is found for the same square.
    """
    try:
        return PropertyRelationship.objects.get(game=game, square=square)
    except PropertyRelationship.DoesNotExist:
        return None
    except MultipleObjectsReturned:
        raise GameLogicError("more than one owners for the same square")
```

3.1.6 `_get_jail_square()`

Retrieves the designated Jail square for the board.

Returns:

Name	Type	Description
`BaseSquare`	`BaseSquare`	The JailSquare instance.

Raises:

Type	Description
`GameDesignError`	If there are no jail squares or too many jail squares.

- **Source code in** `magnate/game_utils.py`

```
def _get_jail_square() -> BaseSquare:
    """
    Retrieves the designated Jail square for the board.

    Returns:
        BaseSquare: The JailSquare instance.

    Raises:
        GameDesignError: If there are no jail squares or too many jail squares.
    """
    try:
        return JailSquare.objects.get()
    except JailSquare.DoesNotExist:
        raise GameDesignError("there are no jail squares in the game")
    except MultipleObjectsReturned:
        raise GameDesignError("there are too many jail squares")
```

3.1.7 🏠 Property & Economy Rules

Calculates financial impacts, building constraints, and mortgage status.

Core Game Logic Module.

This module handles the rules, state transitions, and actions of the game. It provides helper functions to calculate net worth, rent, building/demolishing rules, and a GameManager class that acts as a state machine for the different phases of a player's turn.

3.1.8 `_calculate_rent_price(game, user, square)`

Calculates the rent price a user must pay when landing on a square.

Handles different logic for PropertySquares (houses), TramSquares, BridgeSquares, and ServerSquares based on ownership and multipliers.

Parameters:

Name	Type	Description	Default
`game`	`Game`	The current game instance.	*required*
`user`	`CustomUser`	The user landing on the square.	*required*
`square`	`BaseSquare`	The square being landed on.	*required*

Returns:

Name	Type	Description
`int`	`int`	The calculated rent amount to pay. Returns 0 if unowned or owned by the user.

Raises:

Type	Description
GameDesignError	If the rent arrays on the square are incorrectly configured.
GameLogicError	If an unexpected condition occurs during calculation.

- **Source code in** `magnate/game_utils.py`

```
def _calculate_rent_price(game: Game, user: CustomUser, square: BaseSquare) -> int:
    """
    Calculates the rent price a user must pay when landing on a square.

    Handles different logic for PropertySquares (houses), TramSquares,
    BridgeSquares, and ServerSquares based on ownership and multipliers.

    Args:
        game (Game): The current game instance.
        user (CustomUser): The user landing on the square.
        square (BaseSquare): The square being landed on.

    Returns:
        int: The calculated rent amount to pay. Returns 0 if unowned or owned by the user.

    Raises:
        GameDesignError: If the rent arrays on the square are incorrectly configured.
        GameLogicError: If an unexpected condition occurs during calculation.
    """
    # If it is not owned or is owned by the same user, no rent is paid
    prop_rel = _get_relationship(game, square)
    if not prop_rel or prop_rel.owner == user or prop_rel.mortgage:
        return 0

    houses = prop_rel.houses

    if isinstance(square, PropertySquare):
        if not square.rent_prices or len(square.rent_prices) < 6:
            raise GameDesignError(f"Incorrect rent prices for square {square.custom_id}")
        if houses == -1:
            return square.rent_prices[0]
        elif houses == 0:
            return square.rent_prices[0] * 2  # grupo completo = doble del alquiler base
        elif houses == 1:
            return square.rent_prices[1]
        elif houses == 2:
            return square.rent_prices[2]
        elif houses == 3:
            return square.rent_prices[3]
        elif houses == 4:
            return square.rent_prices[4]
        elif houses == 5:  # hotel
            return square.rent_prices[5]

        return 0

    elif isinstance(square, TramSquare):
        # TODO
        return 0

    elif isinstance(square, BridgeSquare):
```

```
        property_owner = prop_rel.owner
        bridges_owned = PropertyRelationship.objects.filter(game=game, square__bridgesquare__isnull=False, owner=prop
        if not square.rent_prices or len(square.rent_prices) < bridges_owned:
            raise GameDesignError(f"Incorrect rent prices for bridge {square.custom_id}")
        return square.rent_prices[bridges_owned - 1]

    elif isinstance(square, ServerSquare):
        property_owner = prop_rel.owner

        if not square.rent_prices or len(square.rent_prices) < 2:
            raise GameDesignError(f"Incorrect rent prices for square {square.custom_id}")

        squares = PropertyRelationship.objects.filter(game=game, square__serversquare__isnull=False, owner=property_o

        if squares.count() == 2:
            return square.rent_prices[1]
        elif squares.count() == 1:
            return square.rent_prices[0]
        else:
            # TODO: Write something
            raise GameLogicError()
    else:
        return 0
```

3.1.9 `_build_square(game, user, building_square, number_built, free_build)`

Builds houses or hotels on a property.

Validates that the user owns the complete color group and respects the uniform building rule.

Parameters:

Name	Type	Description	Default
`game`	`Game`	The current game instance.	*required*
`user`	`CustomUser`	The user attempting to build.	*required*
`building_square`	`BaseSquare`	The target property square.	*required*
`number_built`	`int`	Number of buildings to construct.	*required*
`free_build`	`bool`	If True, the user is not charged for the construction.	*required*

Returns:

Name	Type	Description
`PropertyRelationship`	`PropertyRelationship`	The updated ownership relationship.

Raises:

Type	Description
`MaliciousUserInput`	If the user does not own the full group, violates uniform building rules, or tries to build beyond the max limit.
`GameLogicError`	If negative house values are encountered.

- **Source code in** `magnate/game_utils.py`

```
def _build_square(game: Game,
                  user: CustomUser,
                  building_square: BaseSquare,
                  number_built: int,
                  free_build: bool) -> PropertyRelationship:
    """
    Builds houses or hotels on a property.

    Validates that the user owns the complete color group and respects
    the uniform building rule.

    Args:
        game (Game): The current game instance.
        user (CustomUser): The user attempting to build.
        building_square (BaseSquare): The target property square.
        number_built (int): Number of buildings to construct.
        free_build (bool): If True, the user is not charged for the construction.

    Returns:
        PropertyRelationship: The updated ownership relationship.

    Raises:
        MaliciousUserInput: If the user does not own the full group, violates
            uniform building rules, or tries to build beyond the max limit.
        GameLogicError: If negative house values are encountered.
    """
    # Check if it's  a property and take its group
    building_square = building_square.get_real_instance()

    if not isinstance(building_square, PropertySquare):
        raise MaliciousUserInput(user, "tried to build in a non property square")

    relationship = _get_relationship(game, building_square)

    if relationship is None:
        raise MaliciousUserInput(user, "no user owns this square")

    if relationship.owner != user:
        raise MaliciousUserInput(user, "tried to build in an unowned property")

    square_group = building_square.group
    actual_houses = relationship.houses

    # Check if user has every square in the group and if its a property
    total_squares_in_group = PropertySquare.objects.filter(
        board=building_square.board,
        group=square_group
    ).count()

    group_relationships = PropertyRelationship.objects.filter(
        game=game,
        owner=user,
        square__propertysquare__group=square_group
    ).select_related('square')

    if group_relationships.count() != total_squares_in_group:
        raise MaliciousUserInput(user, "does not own the group")
```

```
    for rel in group_relationships:
        if rel.houses < 0:
            raise GameLogicError(f"negative house value")
        elif actual_houses + number_built - 1 > rel.houses:
            raise MaliciousUserInput(user, "already owns more than other")

    if actual_houses == 5:
        raise MaliciousUserInput(user, "nothing more to build")

    relationship.houses += number_built
    relationship.save()

    stats = PlayerGameStatistic.objects.get(user=user,game=game)
    stats.built_houses += number_built

    if not free_build:
        coste = building_square.build_price * number_built

        game.money[str(user.pk)] -= coste
        game.save()
        stats.lost_money += coste

    stats.save()

    return relationship  #ack
```

3.1.10 `_demolish_square(game, user, demolition_square, number_demolished, free_demolish)`

Demolishes houses/hotels on a property and returns money to the user.

Ensures that uniform building rules are respected during demolition.

Parameters:

Name	Type	Description	Default
`game`	`Game`	The current game instance.	*required*
`user`	`CustomUser`	The user attempting to demolish.	*required*
`demolition_square`	`BaseSquare`	The square where buildings will be demolished.	*required*
`number_demolished`	`int`	The amount of houses to demolish.	*required*
`free_demolish`	`bool`	If True, no money is refunded to the user.	*required*

Returns:

Name	Type	Description
`PropertyRelationship`	`PropertyRelationship`	The updated ownership relationship.

Raises:

Type	Description
`MaliciousUserInput`	If the property is unowned, owned by someone else, is not a property square, or violates uniform building rules.

- **Source code in** `magnate/game_utils.py`

```
def _demolish_square(game: Game,
                     user: CustomUser,
                     demolition_square: BaseSquare,
                     number_demolished: int,
                     free_demolish: bool) -> PropertyRelationship:
    """
    Demolishes houses/hotels on a property and returns money to the user.

    Ensures that uniform building rules are respected during demolition.

    Args:
        game (Game): The current game instance.
        user (CustomUser): The user attempting to demolish.
        demolition_square (BaseSquare): The square where buildings will be demolished.
        number_demolished (int): The amount of houses to demolish.
        free_demolish (bool): If True, no money is refunded to the user.

    Returns:
        PropertyRelationship: The updated ownership relationship.

    Raises:
        MaliciousUserInput: If the property is unowned, owned by someone else,
            is not a property square, or violates uniform building rules.
    """
    # Check if it's a property
    demolition_square = demolition_square.get_real_instance()
    if not isinstance(demolition_square, PropertySquare):
        raise MaliciousUserInput(user, "tried to demolish a non property square")

    relationship = _get_relationship(game, demolition_square)

    if relationship is None:
        raise MaliciousUserInput(user, "no user owns this square")

    if relationship.owner != user:
        raise MaliciousUserInput(user, "tried to demolish an unowned property")

    actual_houses = relationship.houses

    if actual_houses < number_demolished:
        raise MaliciousUserInput(user, "tried to demolish more houses than are built")

    square_group = demolition_square.group

    group_relationships = PropertyRelationship.objects.filter(
        game=game,
        owner=user,
        square__propertysquare__group=square_group
    ).select_related('square')

    # Check if we can demolish -> respect rule
    for rel in group_relationships:
        if (actual_houses - number_demolished) < (rel.houses - 1):
            raise MaliciousUserInput(user, "unable to demolish so many houses: violates the uniform building rule")

    # demolish
    relationship.houses -= number_demolished
```

```
    relationship.save()

    stats = PlayerGameStatistic.objects.get(user=user,game=game)
    stats.demolished_houses += number_demolished

    if not free_demolish:
        coste = demolition_square.build_price

        game.money[str(user.pk)] += coste // 2 * number_demolished
        game.save()
        stats.won_money += coste // 2 * number_demolished

    stats.save()

    return relationship
```

3.1.11 `_set_mortgage(game, user, target_square, free_mortgage)`

Mortgages a property to receive immediate funds.

Parameters:

Name	Type	Description	Default
`game`	`Game`	The current game instance.	*required*
`user`	`CustomUser`	The user attempting the mortgage.	*required*
`target_square`	`BaseSquare`	The property to be mortgaged.	*required*
`free_mortgage`	`bool`	If True, no money is added to the user's balance.	*required*

Returns:

Name	Type	Description
`PropertyRelationship`	`PropertyRelationship`	The updated relationship.

Raises:

Type	Description
`MaliciousUserInput`	If not owned, already mortgaged, or wrong type of square.
`GameLogicError`	If trying to mortgage a property that still has houses built on it.

- **Source code in** `magnate/game_utils.py`

```
def _set_mortgage(game: Game, user: CustomUser, target_square: BaseSquare, free_mortgage: bool) -> PropertyRelationsh
    """
    Mortgages a property to receive immediate funds.

    Args:
        game (Game): The current game instance.
        user (CustomUser): The user attempting the mortgage.
        target_square (BaseSquare): The property to be mortgaged.
        free_mortgage (bool): If True, no money is added to the user's balance.

    Returns:
        PropertyRelationship: The updated relationship.

    Raises:
```

```
        MaliciousUserInput: If not owned, already mortgaged, or wrong type of square.
        GameLogicError: If trying to mortgage a property that still has houses built on it.
    """
    target_square = target_square.get_real_instance()
    if not (isinstance(target_square, PropertySquare) or
            isinstance(target_square, BridgeSquare) or
            isinstance(target_square, ServerSquare)):
        raise MaliciousUserInput(user, "tried to mortgage a non property/bridge/server square")

    relationship = _get_relationship(game=game, square=target_square)

    if relationship is None:
        raise MaliciousUserInput(user, "no user owns this square")

    if relationship.owner != user:
        raise MaliciousUserInput(user, "tried to mortgage an unowned property")

    if relationship.mortgage:
        raise MaliciousUserInput(user, "tried to mortgage an already mortgaged property")



    if isinstance(target_square, PropertySquare):
        if relationship.houses > 0:
            raise GameLogicError("tried to mortgage a property with houses")

    relationship.mortgage = True
    relationship.save()
    stats = PlayerGameStatistic.objects.get(user=user,game=game)
    stats.num_mortgages += 1

    if not free_mortgage:
        mortgage_value = target_square.buy_price // 2
        game.money[str(user.pk)] += mortgage_value

        stats.won_money += mortgage_value
        game.save()

    stats.save()

    return relationship
```

3.1.12 `_unset_mortgage(game, user, target_square, free_unset_mortgage)`

Lifts the mortgage from a property by paying the required fee.

Parameters:

Name	Type	Description	Default
`game`	`Game`	The current game instance.	*required*
`user`	`CustomUser`	The user lifting the mortgage.	*required*
`target_square`	`BaseSquare`	The property to unmortgage.	*required*
`free_unset_mortgage`	`bool`	If True, the user is not charged.	*required*

Returns:

Name	Type	Description
`PropertyRelationship`	`PropertyRelationship`	The updated relationship.

Raises:

Type	Description
`MaliciousUserInput`	If not owned, not mortgaged, or wrong type of square.

- **Source code in** `magnate/game_utils.py`

```
def _unset_mortgage(game: Game, user: CustomUser, target_square: BaseSquare, free_unset_mortgage: bool) -> PropertyRe
    """
    Lifts the mortgage from a property by paying the required fee.

    Args:
        game (Game): The current game instance.
        user (CustomUser): The user lifting the mortgage.
        target_square (BaseSquare): The property to unmortgage.
        free_unset_mortgage (bool): If True, the user is not charged.

    Returns:
        PropertyRelationship: The updated relationship.

    Raises:
        MaliciousUserInput: If not owned, not mortgaged, or wrong type of square.
    """
    target_square = target_square.get_real_instance()
    if not (isinstance(target_square, PropertySquare) or
            isinstance(target_square, BridgeSquare) or
            isinstance(target_square, ServerSquare)):
        raise MaliciousUserInput(user, "tried to unset mortgage a non property/bridge/server square")

    relationship = _get_relationship(game=game, square=target_square)

    if relationship is None:
        raise MaliciousUserInput(user, "no user owns this square")

    if relationship.owner != user:
        raise MaliciousUserInput(user, "tried to unset mortgage an unowned property")

    if not relationship.mortgage:
        raise MaliciousUserInput(user, "tried to unset mortgage a not mortgaged property")

    relationship.mortgage = False
    relationship.save()
    target_square = target_square.get_real_instance()

    if not free_unset_mortgage:
        mortgage_value = target_square.buy_price // 2
        game.money[str(user.pk)] -= mortgage_value
        game.save()
        stats = PlayerGameStatistic.objects.get(user=user,game=game)
        stats.lost_money += mortgage_value
```

```
    return relationship
```

3.1.13 ⚙️ GameManager

The primary gateway for all frontend actions. Every action sent via WebSocket or REST must be processed through the `process_action` method.

3.1.14 `magnate.games.GameManager`

State machine and core processor for Game Actions.

The GameManager encapsulates the logic for each specific phase of a player's turn, processing polymorphic Action objects and mutating the database state accordingly.

- **Source code in `magnate/games.py`**

```
class GameManager:
    """
    State machine and core processor for Game Actions.

    The GameManager encapsulates the logic for each specific phase of a
    player's turn, processing polymorphic Action objects and mutating the
    database state accordingly.
    """
    ################################################################################
    # Phase logic
    ################################################################################

    ROLL_THE_DICES = Game.GamePhase.roll_the_dices
    CHOOSE_SQUARE = Game.GamePhase.choose_square
    MANAGEMENT = Game.GamePhase.management
    BUSINESS = Game.GamePhase.business
    ANSWER_TRADE_PROPOSAL = Game.GamePhase.proposal_acceptance
    LIQUIDATION = Game.GamePhase.liquidation
    AUCTION = Game.GamePhase.auction
    PROPOSAL_ACCEPTANCE = Game.GamePhase.proposal_acceptance
    CHOOSE_FANTASY = Game.GamePhase.choose_fantasy
    END_GAME = Game.GamePhase.end_game


    @classmethod
    @database_sync_to_async
    def process_action(cls, game: Game, user: CustomUser, action: Action) -> Response:
        """
        The only public method exposed in the API. It processes each action
        in dedicated functions depending on the current phase and returns
        a Response.

        Args:
            game (Game): The current active game instance.
            user (CustomUser): The user performing the action.
            action (Action): The deserialized action payload.

        Returns:
```

```
    Response: A response object representing the outcome.

Raises:
    MaliciousUserInput: If the user acts out of turn or phase.
    GameLogicError: If an unrecognized phase is encountered.
"""



if isinstance(action, ActionSurrender):
    # TODO
    cls._bankrupt_player(game, user)
    response = Response()
    return _add_basic_response_data(game, response)

if user != game.active_phase_player and not isinstance(action, ActionBid): # if aucction there are no turns
    raise MaliciousUserInput(user, "is not the active player")



if game.phase == cls.ROLL_THE_DICES:
    if isinstance(action, ActionPayBail):
        response = cls._pay_bail_logic(game, user, action)
    elif isinstance(action, ActionThrowDices):
        response = cls._roll_dices_logic(game, user, action)
    else:
        raise MaliciousUserInputAction(game, user, action)
elif game.phase == cls.CHOOSE_SQUARE:
    if not isinstance(action, ActionMoveTo):
        raise MaliciousUserInputAction(game, user, action)
    response = cls._square_chosen_logic(game, user, action)
elif game.phase == cls.CHOOSE_FANTASY:
    if not isinstance(action, ActionChooseCard):
        raise MaliciousUserInputAction(game, user, action)
    response = cls._choose_fantasy_logic(game, user, action)
elif game.phase == cls.MANAGEMENT:
    if not isinstance(action, (ActionBuySquare, ActionDropPurchase, ActionTakeTram, ActionNextPhase)):
        raise MaliciousUserInputAction(game, user, action)
    response = cls._management_logic(game, user, action)
elif game.phase == cls.BUSINESS or game.phase == cls.LIQUIDATION:
    if not isinstance(action, (ActionBuild, ActionDemolish, ActionTradeProposal, ActionMortgageSet, ActionMo
        raise MaliciousUserInputAction(game, user, action)
    response = cls._business_logic(game, user, action)
elif game.phase == cls.ANSWER_TRADE_PROPOSAL:
    if not isinstance(action, ActionTradeAnswer):
        raise MaliciousUserInputAction(game, user, action)
    response = cls._answer_trade_proposal_logic(game, user, action)
elif game.phase == cls.AUCTION:
    if not isinstance(action, ActionBid):
        raise MaliciousUserInputAction(game, user, action)
    response = cls._bid_property_auction_logic(game, user, action)
elif game.phase == cls.END_GAME:
    # TODO: check instance???
    # response = cls._end_game_logic(game,user,action)
    return cls._end_game_logic(game,user,action)
else:
    raise GameLogicError(f"Fase no reconocida o no manejada: {game.phase}")

return _add_basic_response_data(game, response)
```

```
@staticmethod
def _pay_bail_logic(game: Game, user: CustomUser, action: ActionPayBail) -> Response:
    """
    Processes the action to pay the jail bail.

    Args:
        game (Game): The current game instance.
        user (CustomUser): The jailed user.
        action (ActionPayBail): The bail action payload.

    Returns:
        Response: The standard response.

    Raises:
        MaliciousUserInput: If the user is not actually in jail.
        GameLogicError: If the user does not have enough money.
    """

    GameManager._cancel_all_timers(game)

    square = _get_user_square(game, user).get_real_instance()

    if not isinstance(square, JailSquare):
        raise MaliciousUserInput(user, "is not in jail")

    remaining = game.jail_remaining_turns.get(str(user.pk), 0)
    if remaining == 0:
        raise MaliciousUserInput(user, "is not in jail (no turns remaining)")

    bail_price = square.bail_price

    if game.money[str(user.pk)] < bail_price:
        raise GameLogicError("not enough money to pay bail")

    game.money[str(user.pk)] -= bail_price
    game.jail_remaining_turns[str(user.pk)] = 0
    game.save()

    stats = PlayerGameStatistic.objects.get(user=user,game=game)
    stats.lost_money += bail_price
    stats.save()
    # roll the dices -> continue normally
    GameManager._set_next_phase_timer(game, user)

    game.save()
    return Response()

@staticmethod
def _roll_dices_logic(game: Game, user: CustomUser, action: ActionThrowDices) -> Response:
    """
    Handles rolling the dice, checking for doubles/triples, and resolving jail mechanics.

    Args:
        game (Game): The current game instance.
        user (CustomUser): The user throwing the dice.
        action (ActionThrowDices): The dice throw action payload.

    Returns:
        Response: Standard response.
```

```
    """

    GameManager._cancel_all_timers(game)

    response: ResponseThrowDices = ResponseThrowDices()

    d1 = random.randint(1,6)
    d2 = random.randint(1,6)
    d3 = random.randint(1,6) # 4-6 are the bus faces

    response.dice1, response.dice2, response.dice_bus = d1, d2, d3

    triples = d3 <= 3 and (d1 == d2 == d3)
    doubles = (d1 == d2) and not triples

    response.triple = triples

    current_pos_square = _get_user_square(game, user).get_real_instance()
    current_pos_id = current_pos_square.custom_id

    # jail logic
    remaining_jail_turns = game.jail_remaining_turns.get(str(user.pk), 0)
    is_jailed = remaining_jail_turns > 0

    if is_jailed:
        jail_sq = current_pos_square
        if isinstance(jail_sq, JailSquare):
            if remaining_jail_turns == 1: #obligado a salir
                game.money[str(user.pk)] -= jail_sq.bail_price
                game.jail_remaining_turns[str(user.pk)] = 0
                stats = PlayerGameStatistic.objects.get(user=user,game=game)
                stats.turns_in_jail += 1
                stats.lost_money += jail_sq.bail_price
                stats.save()
            elif doubles: #sale gratis
                game.jail_remaining_turns[str(user.pk)] = 0
                game.streak = 0
            else:
                # stays in jail
                game.jail_remaining_turns[str(user.pk)] -= 1
                game.phase = GameManager.BUSINESS
                game.save()
                stats = PlayerGameStatistic.objects.get(user=user,game=game)
                stats.turns_in_jail += 1
                stats.save()
                response.path = [current_pos_id]
                GameManager._set_next_phase_timer(game, user)
                return response
        else:
            raise GameLogicError(f"Cannot be in jail status and not in jail square")

    # Not jailed
    if triples:
        # path current -> decided in chosen
        response.triple = True
        square = current_pos_square
        all_squares = BaseSquare.objects.filter(board=square.board)
        # All squares are suitable destinations
        possible_destinations = [s.custom_id for s in all_squares]
```

```
    possible_destinations.remove(_get_jail_square().custom_id)
    game.possible_destinations = {str(c_id): 0 for c_id in possible_destinations}
    response.destinations = possible_destinations
    game.phase = GameManager.CHOOSE_SQUARE
    response.path = [current_pos_id]
    game.save()
    GameManager._set_next_phase_timer(game, user)
    return response
elif doubles: # doubles streak only if not getting out of jail via doubles
    if game.streak >= 2:
        # path -> current and jail
        jail_square = _get_jail_square()
        response.destinations = [jail_square.custom_id]
        game.streak = 0
        response.streak = game.streak

        game.positions[str(user.pk)] = jail_square.custom_id
        game.jail_remaining_turns[str(user.pk)] = 3
        response.path = [current_pos_id, jail_square.custom_id]

        stats = PlayerGameStatistic.objects.get(user=user,game=game)
        stats.times_in_jail += 1
        stats.save()

        game.phase = GameManager.LIQUIDATION

        game.save()
        GameManager._set_next_phase_timer(game, user)
        return response
    elif not is_jailed:
        game.streak = game.streak + 1
else:
    game.streak = 0

response.streak = game.streak

# Hasn't gone to jail
dice_combinations = _compute_dice_combinations(d1, d2, d3)
game.possible_destinations, passed_go_map = _get_possible_destinations_ids(game, user, dice_combinations)

response.destinations = list(game.possible_destinations.keys())
if len(game.possible_destinations) > 1:
    # path in square chosen logic
    game.phase = GameManager.CHOOSE_SQUARE
    response.path = [current_pos_id]
else:
    stats = PlayerGameStatistic.objects.get(user=user,game=game)
    stats.walked_squares += dice_combinations[0]
    stats.save()
    dest_square_id = next(iter(game.possible_destinations))
    steps = game.possible_destinations[dest_square_id]

    # Use _move_player_logic to get the traversed path and check for "Go to Jail" or "Passing Go".
    move_result = _move_player_logic(current_pos_square, steps)
    response.path = move_result["path"]

    if move_result["jailed"]:
        # landing in go to jail; update state to jail the player.
        jail = JailSquare.objects.first()
```

```
            if jail is None:
                raise GameDesignError('no jail in game')

            game.positions[str(user.pk)] = jail.custom_id
            game.jail_remaining_turns[str(user.pk)] = 3
            game.phase = GameManager.LIQUIDATION
            stats = PlayerGameStatistic.objects.get(user=user, game=game)
            stats.times_in_jail += 1
            stats.save()
        else:
            game.positions[str(user.pk)] = dest_square_id
            square = _get_square_by_custom_id(dest_square_id)
            _apply_square_arrival(game, user, response, square, move_result["passed_go"])


    game.save()
    GameManager._set_next_phase_timer(game, user)
    return response

@staticmethod
def _square_chosen_logic(game: Game, user: CustomUser, action: Action) -> Response:
    """
    Handles the logic when a user selects their final destination (if multiple choices were given).

    Args:
        game (Game): The current game instance.
        user (CustomUser): The current user.
        action (ActionMoveTo): Action indicating the chosen square.

    Returns:
        Response: Standard response.

    Raises:
        MaliciousUserInput: If the chosen square is not in `possible_destinations`.
    """

    GameManager._cancel_all_timers(game)
    response: ResponseChooseSquare = ResponseChooseSquare()

    if not isinstance(action, ActionMoveTo):
        raise MaliciousUserInputAction(game, user, action)

    square = action.square

    if str(square.custom_id) not in game.possible_destinations:
        raise MaliciousUserInput(user, "tried to move to an illegal square")

    current_pos_id = game.positions[str(user.pk)]
    current_pos_square = _get_square_by_custom_id(current_pos_id).get_real_instance()

    steps = game.possible_destinations.get(str(square.custom_id))

    move_result = _move_player_logic(current_pos_square, steps)

    response.path = move_result["path"]

    if move_result["jailed"]:
        # landing in go to jail; update state to jail the player.
        jail = JailSquare.objects.first()
```

```
            if jail is None:
                raise GameDesignError('no jail in game')

            game.positions[str(user.pk)] = jail.custom_id
            game.jail_remaining_turns[str(user.pk)] = 3
            game.phase = GameManager.LIQUIDATION
            stats = PlayerGameStatistic.objects.get(user=user, game=game)
            stats.times_in_jail += 1
            stats.save()
        else:
            game.positions[str(user.pk)] = square.custom_id
            square = _get_square_by_custom_id(square.custom_id)
            _apply_square_arrival(game, user, response, square, move_result["passed_go"])

        stats = PlayerGameStatistic.objects.get(user=user,game=game)
        stats.walked_squares += steps
        stats.save()

        game.possible_destinations = dict()
        game.save()

        GameManager._set_next_phase_timer(game, user)
        return response

    @staticmethod
    def _choose_fantasy_logic(game: Game, user: CustomUser, action: ActionChooseCard) -> Response:
        GameManager._cancel_all_timers(game)

        response = ResponseChooseFantasy()
        fantasy_event = game.fantasy_event
        generate = not action.chosen_revealed_card
        new_fantasy = None

        if not generate:
            apply_fantasy_event(game, user, fantasy_event)
            response.fantasy_event = fantasy_event
            game.fantasy_event = None

        else:
            new_fantasy = FantasyEventFactory.generate()
            apply_fantasy_event(game, user, new_fantasy)
            response.fantasy_event = new_fantasy
            game.fantasy_event = None


        if game.streak == 0:
            game.phase = GameManager.BUSINESS
        else:
            game.phase = GameManager.ROLL_THE_DICES

        if game.phase == GameManager.BUSINESS:
            GameManager._set_next_phase_timer(game, user)
        elif game.phase == GameManager.ROLL_THE_DICES:
            GameManager._set_kick_out_timer(game, user)

        game.save()
        return response

    @staticmethod
```

```
def _management_logic(game: Game, user: CustomUser, action: Action) -> Response:
    """
    Logic of management phase, where the user can buy properties, pay bills etc.

    It checks the user's action against the current square they are on to update
    the game state and transition to the next phase.

    Args:
        game (Game): The current game instance.
        user (CustomUser): The acting user.
        action (Action): The payload of the action taken (e.g., ActionBuySquare).

    Returns:
        Response: Standard response.

    Raises:
        MaliciousUserInputAction: If the action does not fit the phase/context.
    """
    GameManager._cancel_all_timers(game)

    current_square = _get_user_square(game, user).get_real_instance()
    prop_rel = _get_relationship(game, current_square)

    if isinstance(action, ActionBuySquare):
        if isinstance(current_square, PropertySquare):
            # TODO: Check money
            game.money[str(user.pk)] -= current_square.buy_price
            stats = PlayerGameStatistic.objects.get(user=user,game=game)
            stats.lost_money += current_square.buy_price
            stats.save()
            new_property = PropertyRelationship(game=game, square=current_square, owner=user)
            user_properties = PropertyRelationship.objects.filter(game=game, owner=user)
            user_same_group_properties = user_properties.filter(square__propertysquare__group=current_square.grou

            group_squares = PropertySquare.objects.filter(group=current_square.group, board = current_square.boar

            if user_same_group_properties.count() == group_squares.count() - 1:
                new_property.houses = 0
                user_same_group_properties.update(houses=0)
            else:
                new_property.houses = -1
            new_property.save()

        elif isinstance(current_square, ServerSquare):
            game.money[str(user.pk)] -= current_square.buy_price
            stats = PlayerGameStatistic.objects.get(user=user,game=game)
            stats.lost_money += current_square.buy_price
            stats.save()
            new_property = PropertyRelationship(game=game, square=current_square, owner=user)
            new_property.save()

        elif isinstance(current_square, BridgeSquare):
            game.money[str(user.pk)] -= current_square.buy_price
            stats = PlayerGameStatistic.objects.get(user=user, game=game)
            stats.lost_money += current_square.buy_price
            stats.save()
            new_property = PropertyRelationship(game=game, square=current_square, owner=user)
            new_property.save()
```

```
            else:
                raise MaliciousUserInputAction(game, user, action)
        elif isinstance(action, ActionDropPurchase):
            if isinstance(action.square, (PropertySquare, ServerSquare, BridgeSquare)):
                return GameManager._initiate_auction(game, action.square)
            else:
                raise MaliciousUserInputAction(game, user, action)
        elif isinstance(action, ActionTakeTram):
            if isinstance(current_square, TramSquare):
                square = action.square
                tram_squares = TramSquare.objects.filter()
                tram_square_actual_id = game.positions[str(user.pk)]
                tram_squares_extern_ids = [s.custom_id for s in tram_squares]

                if square.custom_id == tram_square_actual_id: # case stay in the same square, free
                    pass
                elif square.custom_id in tram_squares_extern_ids: # Move to another square
                    if game.money[str(user.pk)] < square.buy_price:
                        raise MaliciousUserInput(user, "does not have enough money to take tram")
                    game.money[str(user.pk)] -= square.buy_price
                    stats = PlayerGameStatistic.objects.get(user=user,game=game)
                    stats.lost_money += square.buy_price
                    stats.save()
                    game.positions[str(user.pk)] = square.custom_id
                else:
                    raise MaliciousUserInput(user, "tried to take a tram to a non tram square")
            else:
                raise MaliciousUserInputAction(game, user, action)
        elif isinstance(action, ActionNextPhase):
            # TODO: Remove?
            pass
        else:
            raise MaliciousUserInputAction(game, user, action)

        if game.phase == GameManager.MANAGEMENT:
            if game.streak == 0:
                game.phase = GameManager.BUSINESS
            else:
                game.phase = GameManager.ROLL_THE_DICES

        if game.phase == GameManager.BUSINESS:
            GameManager._set_next_phase_timer(game, user)
        elif game.phase == GameManager.ROLL_THE_DICES:
            GameManager._set_kick_out_timer(game, user)

        game.save()
        return Response()

    @staticmethod
    def _business_logic(game: Game, user: CustomUser, action: Action) -> Response:
        """
        Unifies the business and liquidation phases. Handles building, demolishing,
        trading, mortgaging, and proceeding to the next turn.

        Args:
            game (Game): The current game instance.
            user (CustomUser): The active user.
            action (Action): The payload of the action taken.
```

```
        Returns:
            Response: Standard response.

        Raises:
            MaliciousUserInput: If restrictions (e.g., negative balance on next turn) are unmet.
        """

        GameManager._cancel_all_timers(game)
        # Logic for the business phase where players can build houses, trade, etc.
        if isinstance(action, ActionBuild):
            # Check owner
            building_square = action.square
            relationship = _build_square(game, user, building_square, action.houses, False)

        elif isinstance(action, ActionDemolish):
            # Similiar to build
            demolition_square = action.square
            relationship = _demolish_square(game, user, demolition_square, action.houses, False)

        elif isinstance(action, ActionTradeProposal):
            relationship = GameManager._propose_trade(game, user, action)

        elif isinstance(action, ActionMortgageSet):
            relationship = _set_mortgage(game, user, action.square, False)

        elif isinstance(action, ActionMortgageUnset):
            relationship = _unset_mortgage(game, user, action.square, False)
        elif isinstance(action, ActionNextPhase):
            current_money = game.money[str(user.pk)]


            if game.phase == GameManager.BUSINESS:
                if current_money >= 0:
                    GameManager._next_turn(game, user)
                else:
                    game.phase = GameManager.LIQUIDATION
                    game.save()

            elif game.phase == GameManager.LIQUIDATION:
                if current_money >= 0:
                    GameManager._next_turn(game, user)
                else:
                    raise MaliciousUserInput(user, "Cannot end in NEGATIVE")
            return Response() # timers set in next turn

        GameManager._set_next_phase_timer(game, user)
        return Response()

    @staticmethod
    def _answer_trade_proposal_logic(game: Game, user: CustomUser, action: Action) -> Response:
        """
        Processes a user's answer (accept/reject) to an active trade proposal.

        Args:
            game (Game): The current game instance.
            user (CustomUser): The targeted user responding to the offer.
            action (ActionTradeAnswer): The user's response payload.

        Returns:
```

```
        Response: Standard response.

    Raises:
        MaliciousUserInput: If an unauthorized user attempts to answer,
            or references an invalid proposal.
    """
    if isinstance(action, ActionTradeAnswer):
        offer = action.proposal

        offering = offer.player

        offered_money = offer.offered_money
        asked_money = offer.asked_money
        offered_properties = offer.offered_properties
        asked_properties = offer.asked_properties

        # TODO: Verify no player goes to negative (unless liquidation)

        if user != offer.destination_user:
            raise MaliciousUserInput(user, f"cannot accept proposal {offer}")

        if offer != game.proposal:
            raise MaliciousUserInput(user, f"tried to reference a non-existent proposal")

        if action.choose:
            for relationship in offered_properties.all():
                relationship.owner = user
                relationship.houses = -1 # reset houses
                relationship.save()

            for relationship in asked_properties.all():
                relationship.owner = offering
                relationship.houses = -1
                relationship.save()

            final_money = offered_money - asked_money

            game.money[str(offering.pk)] -= offered_money
            game.money[str(offering.pk)] += asked_money

            stats = PlayerGameStatistic.objects.get(user=offering,game=game)
            if final_money > 0:
                stats.lost_money += abs(final_money)
            else:
                stats.won_money += abs(final_money)
            stats.num_trades += 1
            stats.save()

            game.money[str(user.pk)] -= asked_money
            game.money[str(user.pk)] += offered_money

            stats = PlayerGameStatistic.objects.get(user=user,game=game)
            if final_money > 0:
                stats.won_money += abs(final_money)
            else:
                stats.lost_money += abs(final_money)
            stats.num_trades += 1
            stats.save()
```

```
                game.save()

            game.phase = GameManager.BUSINESS
            game.active_phase_player = offering
            game.save()
            GameManager._set_next_phase_timer(game, offering)

            return Response()
        else:
            raise MaliciousUserInputAction(game, user, action)

    @staticmethod
    def _initiate_auction(game: Game, square: BaseSquare) -> Response:
        """
        Transitions the game into an AUCTION phase for a dropped property.

        Args:
            game (Game): The current game instance.
            square (BaseSquare): The property square going up for auction.

        Returns:
            Response: Standard response.
        """
        GameManager._cancel_all_timers(game)

        game.next_phase_task_id = None
        game.phase = GameManager.AUCTION

        auction = Auction.objects.create(game=game, square=square, is_active=True, bids = {})
        game.current_auction = auction
        game.save()

        # TODO: Remove magic number
        GameManager._set_auction_timer(game)
        # TODO: Remove in production
        game.refresh_from_db()

        return Response()

    @staticmethod
    def _bid_property_auction_logic(game: Game, user: CustomUser, action: Action) -> Response:
        """
        Registers a player's bid during an active auction phase.

        Args:
            game (Game): The current game instance.
            user (CustomUser): The user placing the bid.
            action (ActionBid): The action carrying the bid amount.

        Returns:
            Response: Standard response.

        Raises:
            GameLogicError: If auction state is corrupted.
            MaliciousUserInput: If the user bids twice or exceeds their balance.
        """
        if not isinstance(action, ActionBid):
            raise MaliciousUserInputAction(game, user, action)
```

```
        auction = game.current_auction
        if not auction:
            raise GameLogicError("No active auction")


        bids = auction.bids
        # user has not bid yet
        if bids.get(str(user.pk)):
            raise MaliciousUserInput(user, "User already placed a bid in this auction")

        # user who started the bid cant bid
        dropped = ActionDropPurchase.objects.filter(game=game, player=user, square=auction.square).exists()
        if dropped:
            raise MaliciousUserInput(user, "cannot bid in an auction they triggered")

        # user has enough money -> compulsory for auctions
        amount = action.amount
        if amount > game.money[str(user.pk)]:
            raise MaliciousUserInput(user, "Bid amount exceeds current balance")


        bids[str(user.pk)] = amount
        auction.bids = bids
        auction.save()

        game.save()
        return Response()

    @staticmethod
    def _end_auction(game: Game) -> Response | None:
        """
        Ends an active auction, resolves the winner based on the highest bid,
        handles ties, and transitions the game state back to BUSINESS.

        Args:
            game (Game): The current game instance.

        Returns:
            Auction: The finalized auction object.

        Raises:
            GameLogicError: If not in AUCTION phase or state is missing square ID.
        """
        ## call this with a timer -> end of the auction

        if game.phase == GameManager.END_GAME:
            return None # Inevitable if the auction callback is triggered after the game has ended, just ignore it

        if game.phase != GameManager.AUCTION:
            raise GameLogicError("Tried to end auction but game is not in auction phase")

        auction = game.current_auction
        if not auction:
            raise GameLogicError("Game is in AUCTION phase but no auction object found")

        square = auction.square.get_real_instance()
        bids = auction.bids
```

```
        # no one bid
        if not bids:
            if game.streak == 0:
                game.phase = GameManager.BUSINESS
            else:
                game.phase = GameManager.ROLL_THE_DICES

            auction.is_active = False
            auction.winner = None
            auction.final_amount = 0
            auction.is_tie = False
            auction.save()

            game.current_auction = None
            game.save()

            if game.phase == GameManager.BUSINESS:
                GameManager._set_next_phase_timer(game, game.active_turn_player)
            elif game.phase == GameManager.ROLL_THE_DICES:
                GameManager._set_kick_out_timer(game, game.active_turn_player)

            game.save()
            return ResponseAuction(auction=auction)

        max_bid_amount = max(bids.values())
        winners = [int(uid) for uid, amt in bids.items() if amt == max_bid_amount]

        if len(winners)> 1:
            if game.streak == 0:
                game.phase = GameManager.BUSINESS
            else:
                game.phase = GameManager.ROLL_THE_DICES

            auction.is_active = False
            auction.winner = None
            auction.final_amount = max_bid_amount
            auction.is_tie = True
            auction.save()

            game.current_auction = None
            game.save()

            if game.phase == GameManager.BUSINESS:
                GameManager._set_next_phase_timer(game, game.active_turn_player)
            elif game.phase == GameManager.ROLL_THE_DICES:
                GameManager._set_kick_out_timer(game, game.active_turn_player)

            game.save()
            return ResponseAuction(auction=auction)

        winner_id = winners[0]
        if str(winner_id) not in game.money: # surrends mid auction -> no winner
            game.phase = GameManager.BUSINESS if game.streak == 0 else GameManager.ROLL_THE_DICES
            auction.is_active = False
            auction.save()

            game.current_auction = None
            game.save()
```

```
            if game.phase == GameManager.BUSINESS:
                GameManager._set_next_phase_timer(game, game.active_turn_player)
            elif game.phase == GameManager.ROLL_THE_DICES:
                GameManager._set_kick_out_timer(game, game.active_turn_player)

            game.save()
            return ResponseAuction(auction=auction)

        winner = CustomUser.objects.get(pk=winner_id)
        highest_bid = max_bid_amount

        game.money[str(winner.pk)] -= highest_bid
        stats = PlayerGameStatistic.objects.get(user=winner,game=game)
        stats.lost_money += highest_bid
        stats.save()

        new_property = PropertyRelationship(game=game, square=square, owner=winner)

        # groups n houses
        if isinstance(square, PropertySquare):
            user_properties = PropertyRelationship.objects.filter(game=game, owner=winner)
            user_same_group_properties = user_properties.filter(square__propertysquare__group=square.group)
            group_squares = PropertySquare.objects.filter(group=square.group, board = square.board)

            if user_same_group_properties.count() == group_squares.count() - 1:
                new_property.houses = 0
                user_same_group_properties.update(houses=0)
            else:
                new_property.houses = -1
        else:
            new_property.houses = -1

        new_property.save()

        auction.winner = winner
        auction.final_amount = highest_bid
        auction.is_active = False
        auction.is_tie = False
        auction.save()

        if game.streak == 0:
            game.phase = GameManager.BUSINESS
        else:
            game.phase = GameManager.ROLL_THE_DICES

        game.current_auction = None
        game.save()

        if game.phase == GameManager.BUSINESS:
            GameManager._set_next_phase_timer(game, game.active_turn_player)
        elif game.phase == GameManager.ROLL_THE_DICES:
            GameManager._set_kick_out_timer(game, game.active_turn_player)

        game.save()

        return ResponseAuction(auction=auction)

    @classmethod
    def _next_turn(cls, game: Game, user: CustomUser) -> None:
```

```
        players_list = list(game.players.all())
        num_players = len(players_list)
        current_index = -1
        current_player_id = -1
        for p in players_list:
            if p == game.active_turn_player:
                current_player_id = p.pk
                current_index = game.ordered_players.index(current_player_id)
                break

        if current_index == -1:
            raise GameLogicError('current player not found')

        next_index = (current_index + 1) % num_players
        # The next active user is for both: phase and turn
        next_player = game.players.filter(pk=game.ordered_players[next_index]).first()
        if next_player is None:
            raise GameLogicError('next player is None')

        game.active_phase_player = next_player
        game.active_turn_player = next_player
        game.phase = GameManager.ROLL_THE_DICES
        game.current_turn += 1
        game.save()

        GameManager._cancel_all_timers(game)

        GameManager._set_kick_out_timer(game, next_player)

        game.save()

    @classmethod
    def _propose_trade(cls, game: Game, user: CustomUser, action: ActionTradeProposal) -> None:
        if action.player != user or action.offered_money < 0 or action.asked_money < 0:
            raise MaliciousUserInput(user, "cannot do operation")
        if action.destination_user not in game.players.all():
            # FIXME: Change to internal order so that it handles player change to AI
            raise MaliciousUserInput(user, "referenced a player that is not in game")

        asked_properties_list = action.asked_properties.all()
        asked_count = PropertyRelationship.objects.filter(
            game=game,
            owner=action.destination_user,
            id__in=action.asked_properties.all()
        ).count()

        if asked_count != asked_properties_list.count():
            raise MaliciousUserInput(user, "destination does not have enough properties")

        offered_properties_list = action.offered_properties.all()
        offered_count = PropertyRelationship.objects.filter(
            game=game,
            owner=action.player,
            id__in=offered_properties_list
        ).count()

        if offered_count != offered_properties_list.count():
            raise MaliciousUserInput(user, "offer does not have enough properties")
```

```
    all_trade_properties = list(asked_properties_list) + list(offered_properties_list)
    for rel in all_trade_properties:
        real_sq = rel.square.get_real_instance()
        if isinstance(real_sq, PropertySquare):
            group_has_houses = PropertyRelationship.objects.filter(
                game=game,
                square__propertysquare__group=real_sq.group,
                houses__gt=0
            ).exists()
            if group_has_houses:
                raise MaliciousUserInput(user, "cannot trade properties from a group with constructions")

    game.phase = GameManager.PROPOSAL_ACCEPTANCE
    game.active_phase_player = action.destination_user
    game.proposal = action # type: ignore
    game.save()
    GameManager._set_next_phase_timer(game, action.destination_user)

@classmethod
def _bankrupt_player(cls, game: Game, user: CustomUser):
    try:
        current_idx = game.ordered_players.index(user.pk)
        next_pk = game.ordered_players[(current_idx + 1) % len(game.ordered_players)]
        next_player = CustomUser.objects.get(pk=next_pk)
    except ValueError:
        next_player = None

    if user.pk in game.ordered_players:
        game.ordered_players.remove(user.pk)

    game.players.remove(user)
    game.money.pop(str(user.pk), None)
    game.positions.pop(str(user.pk), None)
    game.jail_remaining_turns.pop(str(user.pk), None)


    PropertyRelationship.objects.filter(game=game, owner=user).delete()
    user.active_game = None
    user.save()


    if game.players.count() == 1:
        game.phase = GameManager.END_GAME
        GameManager._cancel_all_timers(game)

        if game.current_auction:
            auction = game.current_auction
            auction.is_active = False
            auction.save()
            game.current_auction = None

        game.save()
        return #TODO: endgame logic

    if game.active_turn_player == user and next_player:
        game.active_turn_player = next_player
        game.active_phase_player = next_player
        game.phase = GameManager.ROLL_THE_DICES
        game.streak = 0
```

```
            GameManager._cancel_all_timers(game)

            # new task
            GameManager._set_kick_out_timer(game, next_player)

        game.save()

    #TODO: llegar a fase final donde se reparte esto
    @classmethod
    def _apply_end_bonuses(cls, game: Game, num_bonuses: int = 3) -> ResponseBonus:
        all_categories = list(BonusCategory.objects.all())
        chosen = random.sample(all_categories, min(num_bonuses, len(all_categories)))

        response = ResponseBonus()
        bonuses = {}

        for category in chosen:
            field = category.stat_field
            stats = PlayerGameStatistic.objects.filter(game=game)

            max_value = stats.aggregate(Max(field))[f'{field}__max']
            if max_value and max_value > 0:
                winners = list(stats.filter(**{field: max_value}).values_list('user__pk', flat=True))
                for pk in winners:
                    game.money[str(pk)] += category.bonus_amount
            else:
                winners = []

            bonuses[str(category.pk)] = {
                'bonus_amount': category.bonus_amount,
                'winners': winners
            }

        response.bonuses = bonuses
        game.save()
        return response

    @classmethod
    def _end_game_logic(cls, game: Game, user: CustomUser, action: Action) -> Response:
        GameManager._cancel_all_timers(game)

        if not game.finished:
            game.finished = True
            response = cls._apply_end_bonuses(game, num_bonuses=3)
            response.save()
            game.bonus_response = response
            game.save()
            return response
        else:
            raise GameLogicError('game was already ended')

    @staticmethod
    def _set_next_phase_timer(game: Game, user: CustomUser):
        from .tasks import next_phase_callback, bot_play_callback
        from .celery import app

        GameManager._cancel_all_timers(game)
```

```
        task = next_phase_callback.apply_async(args=[game.pk, user.pk], countdown=50)
        game.next_phase_task_id = task.id
        game.save()

        if user.is_bot:
            bot_play_callback.apply_async(args=[game.pk, user.pk], countdown=2)

    @staticmethod
    def _set_kick_out_timer(game: Game, user: CustomUser):
        from .tasks import kick_out_callback, bot_play_callback
        from .celery import app

        if game.kick_out_task_id:
            app.control.revoke(game.kick_out_task_id, terminate=True)

        task = kick_out_callback.apply_async(args=[game.pk, user.pk], countdown=50)
        game.kick_out_task_id = task.id
        game.save()

        if user.is_bot:
            bot_play_callback.apply_async(args=[game.pk, user.pk], countdown=2)

    @staticmethod
    def _cancel_all_timers(game: Game):
        from .celery import app

        if game.next_phase_task_id:
            app.control.revoke(game.next_phase_task_id, terminate=True)
            game.next_phase_task_id = None

        if game.kick_out_task_id:
            app.control.revoke(game.kick_out_task_id, terminate=True)
            game.kick_out_task_id = None

        game.save()

    @staticmethod
    def _set_auction_timer(game: Game):
        import random
        from .tasks import auction_callback, bot_play_callback

        auction_callback.apply_async(args=[game.pk], countdown=10)

        for player in game.players.all():
            if player.is_bot:
                bot_play_callback.apply_async(args=[game.pk, player.pk], countdown=random.randint(2, 6))
```

`process_action(game, user, action)` `classmethod`

The only public method exposed in the API. It processes each action in dedicated functions depending on the current phase and returns a Response.

Parameters:

Name	Type	Description	Default
`game`	`Game`	The current active game instance.	*required*
`user`	`CustomUser`	The user performing the action.	*required*
`action`	`Action`	The deserialized action payload.	*required*

Returns:

Name	Type	Description
`Response`	`Response`	A response object representing the outcome.

Raises:

Type	Description
`MaliciousUserInput`	If the user acts out of turn or phase.
`GameLogicError`	If an unrecognized phase is encountered.

- **Source code in** `magnate/games.py`

```
@classmethod
@database_sync_to_async
def process_action(cls, game: Game, user: CustomUser, action: Action) -> Response:
    """
    The only public method exposed in the API. It processes each action
    in dedicated functions depending on the current phase and returns
    a Response.

    Args:
        game (Game): The current active game instance.
        user (CustomUser): The user performing the action.
        action (Action): The deserialized action payload.

    Returns:
        Response: A response object representing the outcome.

    Raises:
        MaliciousUserInput: If the user acts out of turn or phase.
        GameLogicError: If an unrecognized phase is encountered.
    """


    if isinstance(action, ActionSurrender):
        # TODO
        cls._bankrupt_player(game, user)
        response = Response()
        return _add_basic_response_data(game, response)

    if user != game.active_phase_player and not isinstance(action, ActionBid): # if aucction there are no turns
        raise MaliciousUserInput(user, "is not the active player")



    if game.phase == cls.ROLL_THE_DICES:
        if isinstance(action, ActionPayBail):
            response = cls._pay_bail_logic(game, user, action)
        elif isinstance(action, ActionThrowDices):
            response = cls._roll_dices_logic(game, user, action)
```

```
        else:
            raise MaliciousUserInputAction(game, user, action)
    elif game.phase == cls.CHOOSE_SQUARE:
        if not isinstance(action, ActionMoveTo):
            raise MaliciousUserInputAction(game, user, action)
        response = cls._square_chosen_logic(game, user, action)
    elif game.phase == cls.CHOOSE_FANTASY:
        if not isinstance(action, ActionChooseCard):
            raise MaliciousUserInputAction(game, user, action)
        response = cls._choose_fantasy_logic(game, user, action)
    elif game.phase == cls.MANAGEMENT:
        if not isinstance(action, (ActionBuySquare, ActionDropPurchase, ActionTakeTram, ActionNextPhase)):
            raise MaliciousUserInputAction(game, user, action)
        response = cls._management_logic(game, user, action)
    elif game.phase == cls.BUSINESS or game.phase == cls.LIQUIDATION:
        if not isinstance(action, (ActionBuild, ActionDemolish, ActionTradeProposal, ActionMortgageSet, ActionMortgag
            raise MaliciousUserInputAction(game, user, action)
        response = cls._business_logic(game, user, action)
    elif game.phase == cls.ANSWER_TRADE_PROPOSAL:
        if not isinstance(action, ActionTradeAnswer):
            raise MaliciousUserInputAction(game, user, action)
        response = cls._answer_trade_proposal_logic(game, user, action)
    elif game.phase == cls.AUCTION:
        if not isinstance(action, ActionBid):
            raise MaliciousUserInputAction(game, user, action)
        response = cls._bid_property_auction_logic(game, user, action)
    elif game.phase == cls.END_GAME:
        # TODO: check instance???
        # response = cls._end_game_logic(game,user,action)
        return cls._end_game_logic(game,user,action)
    else:
        raise GameLogicError(f"Fase no reconocida o no manejada: {game.phase}")

    return _add_basic_response_data(game, response)
```

`_pay_bail_logic(game, user, action)` `staticmethod`

Processes the action to pay the jail bail.

Parameters:

Name	Type	Description	Default
`game`	`Game`	The current game instance.	*required*
`user`	`CustomUser`	The jailed user.	*required*
`action`	`ActionPayBail`	The bail action payload.	*required*

Returns:

Name	Type	Description
`Response`	`Response`	The standard response.

Raises:

Type	Description
`MaliciousUserInput`	If the user is not actually in jail.
`GameLogicError`	If the user does not have enough money.

- **Source code in** `magnate/games.py`

```
@staticmethod
def _pay_bail_logic(game: Game, user: CustomUser, action: ActionPayBail) -> Response:
    """
    Processes the action to pay the jail bail.

    Args:
        game (Game): The current game instance.
        user (CustomUser): The jailed user.
        action (ActionPayBail): The bail action payload.

    Returns:
        Response: The standard response.

    Raises:
        MaliciousUserInput: If the user is not actually in jail.
        GameLogicError: If the user does not have enough money.
    """

    GameManager._cancel_all_timers(game)

    square = _get_user_square(game, user).get_real_instance()

    if not isinstance(square, JailSquare):
        raise MaliciousUserInput(user, "is not in jail")

    remaining = game.jail_remaining_turns.get(str(user.pk), 0)
    if remaining == 0:
        raise MaliciousUserInput(user, "is not in jail (no turns remaining)")

    bail_price = square.bail_price

    if game.money[str(user.pk)] < bail_price:
        raise GameLogicError("not enough money to pay bail")

    game.money[str(user.pk)] -= bail_price
    game.jail_remaining_turns[str(user.pk)] = 0
    game.save()

    stats = PlayerGameStatistic.objects.get(user=user,game=game)
    stats.lost_money += bail_price
    stats.save()
    # roll the dices -> continue normally
    GameManager._set_next_phase_timer(game, user)

    game.save()
    return Response()
```

`_roll_dices_logic(game, user, action)` `staticmethod`

Handles rolling the dice, checking for doubles/triples, and resolving jail mechanics.

Parameters:

Name	Type	Description	Default
`game`	`Game`	The current game instance.	*required*
`user`	`CustomUser`	The user throwing the dice.	*required*
`action`	`ActionThrowDices`	The dice throw action payload.	*required*

Returns:

Name	Type	Description
`Response`	`Response`	Standard response.

- **Source code in** `magnate/games.py`

```
@staticmethod
def _roll_dices_logic(game: Game, user: CustomUser, action: ActionThrowDices) -> Response:
    """
    Handles rolling the dice, checking for doubles/triples, and resolving jail mechanics.

    Args:
        game (Game): The current game instance.
        user (CustomUser): The user throwing the dice.
        action (ActionThrowDices): The dice throw action payload.

    Returns:
        Response: Standard response.
    """

    GameManager._cancel_all_timers(game)

    response: ResponseThrowDices = ResponseThrowDices()

    d1 = random.randint(1,6)
    d2 = random.randint(1,6)
    d3 = random.randint(1,6) # 4-6 are the bus faces

    response.dice1, response.dice2, response.dice_bus = d1, d2, d3

    triples = d3 <= 3 and (d1 == d2 == d3)
    doubles = (d1 == d2) and not triples

    response.triple = triples

    current_pos_square = _get_user_square(game, user).get_real_instance()
    current_pos_id = current_pos_square.custom_id

    # jail logic
    remaining_jail_turns = game.jail_remaining_turns.get(str(user.pk), 0)
    is_jailed = remaining_jail_turns > 0

    if is_jailed:
        jail_sq = current_pos_square
        if isinstance(jail_sq, JailSquare):
            if remaining_jail_turns == 1: #obligado a salir
                game.money[str(user.pk)] -= jail_sq.bail_price
```

```
            game.jail_remaining_turns[str(user.pk)] = 0
            stats = PlayerGameStatistic.objects.get(user=user,game=game)
            stats.turns_in_jail += 1
            stats.lost_money += jail_sq.bail_price
            stats.save()
        elif doubles: #sale gratis
            game.jail_remaining_turns[str(user.pk)] = 0
            game.streak = 0
        else:
            # stays in jail
            game.jail_remaining_turns[str(user.pk)] -= 1
            game.phase = GameManager.BUSINESS
            game.save()
            stats = PlayerGameStatistic.objects.get(user=user,game=game)
            stats.turns_in_jail += 1
            stats.save()
            response.path = [current_pos_id]
            GameManager._set_next_phase_timer(game, user)
            return response
    else:
        raise GameLogicError(f"Cannot be in jail status and not in jail square")

# Not jailed
if triples:
    # path current -> decided in chosen
    response.triple = True
    square = current_pos_square
    all_squares = BaseSquare.objects.filter(board=square.board)
    # All squares are suitable destinations
    possible_destinations = [s.custom_id for s in all_squares]
    possible_destinations.remove(_get_jail_square().custom_id)
    game.possible_destinations = {str(c_id): 0 for c_id in possible_destinations}
    response.destinations = possible_destinations
    game.phase = GameManager.CHOOSE_SQUARE
    response.path = [current_pos_id]
    game.save()
    GameManager._set_next_phase_timer(game, user)
    return response
elif doubles: # doubles streak only if not getting out of jail via doubles
    if game.streak >= 2:
        # path -> current and jail
        jail_square = _get_jail_square()
        response.destinations = [jail_square.custom_id]
        game.streak = 0
        response.streak = game.streak

        game.positions[str(user.pk)] = jail_square.custom_id
        game.jail_remaining_turns[str(user.pk)] = 3
        response.path = [current_pos_id, jail_square.custom_id]

        stats = PlayerGameStatistic.objects.get(user=user,game=game)
        stats.times_in_jail += 1
        stats.save()

        game.phase = GameManager.LIQUIDATION

        game.save()
        GameManager._set_next_phase_timer(game, user)
        return response
```

```
        elif not is_jailed:
            game.streak = game.streak + 1
    else:
        game.streak = 0

    response.streak = game.streak

    # Hasn't gone to jail
    dice_combinations = _compute_dice_combinations(d1, d2, d3)
    game.possible_destinations, passed_go_map = _get_possible_destinations_ids(game, user, dice_combinations)

    response.destinations = list(game.possible_destinations.keys())
    if len(game.possible_destinations) > 1:
        # path in square chosen logic
        game.phase = GameManager.CHOOSE_SQUARE
        response.path = [current_pos_id]
    else:
        stats = PlayerGameStatistic.objects.get(user=user,game=game)
        stats.walked_squares += dice_combinations[0]
        stats.save()
        dest_square_id = next(iter(game.possible_destinations))
        steps = game.possible_destinations[dest_square_id]

        # Use _move_player_logic to get the traversed path and check for "Go to Jail" or "Passing Go".
        move_result = _move_player_logic(current_pos_square, steps)
        response.path = move_result["path"]

        if move_result["jailed"]:
            # landing in go to jail; update state to jail the player.
            jail = JailSquare.objects.first()
            if jail is None:
                raise GameDesignError('no jail in game')

            game.positions[str(user.pk)] = jail.custom_id
            game.jail_remaining_turns[str(user.pk)] = 3
            game.phase = GameManager.LIQUIDATION
            stats = PlayerGameStatistic.objects.get(user=user, game=game)
            stats.times_in_jail += 1
            stats.save()
        else:
            game.positions[str(user.pk)] = dest_square_id
            square = _get_square_by_custom_id(dest_square_id)
            _apply_square_arrival(game, user, response, square, move_result["passed_go"])


    game.save()
    GameManager._set_next_phase_timer(game, user)
    return response
```

`_square_chosen_logic(game, user, action)` `staticmethod`

Handles the logic when a user selects their final destination (if multiple choices were given).

Parameters:

Name	Type	Description	Default
`game`	`Game`	The current game instance.	*required*
`user`	`CustomUser`	The current user.	*required*
`action`	`ActionMoveTo`	Action indicating the chosen square.	*required*

Returns:

Name	Type	Description
`Response`	`Response`	Standard response.

Raises:

Type	Description
`MaliciousUserInput`	If the chosen square is not in `possible_destinations`.

- **Source code in `magnate/games.py`**

```
@staticmethod
def _square_chosen_logic(game: Game, user: CustomUser, action: Action) -> Response:
    """
    Handles the logic when a user selects their final destination (if multiple choices were given).

    Args:
        game (Game): The current game instance.
        user (CustomUser): The current user.
        action (ActionMoveTo): Action indicating the chosen square.

    Returns:
        Response: Standard response.

    Raises:
        MaliciousUserInput: If the chosen square is not in `possible_destinations`.
    """

    GameManager._cancel_all_timers(game)
    response: ResponseChooseSquare = ResponseChooseSquare()

    if not isinstance(action, ActionMoveTo):
        raise MaliciousUserInputAction(game, user, action)

    square = action.square

    if str(square.custom_id) not in game.possible_destinations:
        raise MaliciousUserInput(user, "tried to move to an illegal square")

    current_pos_id = game.positions[str(user.pk)]
    current_pos_square = _get_square_by_custom_id(current_pos_id).get_real_instance()

    steps = game.possible_destinations.get(str(square.custom_id))

    move_result = _move_player_logic(current_pos_square, steps)

    response.path = move_result["path"]

    if move_result["jailed"]:
        # landing in go to jail; update state to jail the player.
```

```
        jail = JailSquare.objects.first()
        if jail is None:
            raise GameDesignError('no jail in game')

        game.positions[str(user.pk)] = jail.custom_id
        game.jail_remaining_turns[str(user.pk)] = 3
        game.phase = GameManager.LIQUIDATION
        stats = PlayerGameStatistic.objects.get(user=user, game=game)
        stats.times_in_jail += 1
        stats.save()
    else:
        game.positions[str(user.pk)] = square.custom_id
        square = _get_square_by_custom_id(square.custom_id)
        _apply_square_arrival(game, user, response, square, move_result["passed_go"])

    stats = PlayerGameStatistic.objects.get(user=user,game=game)
    stats.walked_squares += steps
    stats.save()

    game.possible_destinations = dict()
    game.save()

    GameManager._set_next_phase_timer(game, user)
    return response
```

`_management_logic(game, user, action)` `staticmethod`

Logic of management phase, where the user can buy properties, pay bills etc.

It checks the user's action against the current square they are on to update the game state and transition to the next phase.

Parameters:

Name	Type	Description	Default
`game`	`Game`	The current game instance.	*required*
`user`	`CustomUser`	The acting user.	*required*
`action`	`Action`	The payload of the action taken (e.g., ActionBuySquare).	*required*

Returns:

Name	Type	Description
`Response`	`Response`	Standard response.

Raises:

Type	Description
`MaliciousUserInputAction`	If the action does not fit the phase/context.

- **Source code in** `magnate/games.py`

```
@staticmethod
def _management_logic(game: Game, user: CustomUser, action: Action) -> Response:
    """
    Logic of management phase, where the user can buy properties, pay bills etc.
```

```
It checks the user's action against the current square they are on to update
the game state and transition to the next phase.

Args:
    game (Game): The current game instance.
    user (CustomUser): The acting user.
    action (Action): The payload of the action taken (e.g., ActionBuySquare).

Returns:
    Response: Standard response.

Raises:
    MaliciousUserInputAction: If the action does not fit the phase/context.
"""
GameManager._cancel_all_timers(game)

current_square = _get_user_square(game, user).get_real_instance()
prop_rel = _get_relationship(game, current_square)

if isinstance(action, ActionBuySquare):
    if isinstance(current_square, PropertySquare):
        # TODO: Check money
        game.money[str(user.pk)] -= current_square.buy_price
        stats = PlayerGameStatistic.objects.get(user=user,game=game)
        stats.lost_money += current_square.buy_price
        stats.save()
        new_property = PropertyRelationship(game=game, square=current_square, owner=user)
        user_properties = PropertyRelationship.objects.filter(game=game, owner=user)
        user_same_group_properties = user_properties.filter(square__propertysquare__group=current_square.group)

        group_squares = PropertySquare.objects.filter(group=current_square.group, board = current_square.board)

        if user_same_group_properties.count() == group_squares.count() - 1:
            new_property.houses = 0
            user_same_group_properties.update(houses=0)
        else:
            new_property.houses = -1
        new_property.save()

    elif isinstance(current_square, ServerSquare):
        game.money[str(user.pk)] -= current_square.buy_price
        stats = PlayerGameStatistic.objects.get(user=user,game=game)
        stats.lost_money += current_square.buy_price
        stats.save()
        new_property = PropertyRelationship(game=game, square=current_square, owner=user)
        new_property.save()

    elif isinstance(current_square, BridgeSquare):
        game.money[str(user.pk)] -= current_square.buy_price
        stats = PlayerGameStatistic.objects.get(user=user, game=game)
        stats.lost_money += current_square.buy_price
        stats.save()
        new_property = PropertyRelationship(game=game, square=current_square, owner=user)
        new_property.save()

    else:
        raise MaliciousUserInputAction(game, user, action)
elif isinstance(action, ActionDropPurchase):
    if isinstance(action.square, (PropertySquare, ServerSquare, BridgeSquare)):
```

```
            return GameManager._initiate_auction(game, action.square)
        else:
            raise MaliciousUserInputAction(game, user, action)
    elif isinstance(action, ActionTakeTram):
        if isinstance(current_square, TramSquare):
            square = action.square
            tram_squares = TramSquare.objects.filter()
            tram_square_actual_id = game.positions[str(user.pk)]
            tram_squares_extern_ids = [s.custom_id for s in tram_squares]

            if square.custom_id == tram_square_actual_id: # case stay in the same square, free
                pass
            elif square.custom_id in tram_squares_extern_ids: # Move to another square
                if game.money[str(user.pk)] < square.buy_price:
                    raise MaliciousUserInput(user, "does not have enough money to take tram")
                game.money[str(user.pk)] -= square.buy_price
                stats = PlayerGameStatistic.objects.get(user=user,game=game)
                stats.lost_money += square.buy_price
                stats.save()
                game.positions[str(user.pk)] = square.custom_id
            else:
                raise MaliciousUserInput(user, "tried to take a tram to a non tram square")
        else:
            raise MaliciousUserInputAction(game, user, action)
    elif isinstance(action, ActionNextPhase):
        # TODO: Remove?
        pass
    else:
        raise MaliciousUserInputAction(game, user, action)

    if game.phase == GameManager.MANAGEMENT:
        if game.streak == 0:
            game.phase = GameManager.BUSINESS
        else:
            game.phase = GameManager.ROLL_THE_DICES

    if game.phase == GameManager.BUSINESS:
        GameManager._set_next_phase_timer(game, user)
    elif game.phase == GameManager.ROLL_THE_DICES:
        GameManager._set_kick_out_timer(game, user)

    game.save()
    return Response()
```

`_business_logic(game, user, action)` `staticmethod`

Unifies the business and liquidation phases. Handles building, demolishing, trading, mortgaging, and proceeding to the next turn.

Parameters:

Name	Type	Description	Default
`game`	`Game`	The current game instance.	*required*
`user`	`CustomUser`	The active user.	*required*
`action`	`Action`	The payload of the action taken.	*required*

Returns:

Name	Type	Description
Response	Response	Standard response.

Raises:

Type	Description
MaliciousUserInput	If restrictions (e.g., negative balance on next turn) are unmet.

- **Source code in** `magnate/games.py`

```
@staticmethod
def _business_logic(game: Game, user: CustomUser, action: Action) -> Response:
    """
    Unifies the business and liquidation phases. Handles building, demolishing,
    trading, mortgaging, and proceeding to the next turn.

    Args:
        game (Game): The current game instance.
        user (CustomUser): The active user.
        action (Action): The payload of the action taken.

    Returns:
        Response: Standard response.

    Raises:
        MaliciousUserInput: If restrictions (e.g., negative balance on next turn) are unmet.
    """

    GameManager._cancel_all_timers(game)
    # Logic for the business phase where players can build houses, trade, etc.
    if isinstance(action, ActionBuild):
        # Check owner
        building_square = action.square
        relationship = _build_square(game, user, building_square, action.houses, False)

    elif isinstance(action, ActionDemolish):
        # Similiar to build
        demolition_square = action.square
        relationship = _demolish_square(game, user, demolition_square, action.houses, False)

    elif isinstance(action, ActionTradeProposal):
        relationship = GameManager._propose_trade(game, user, action)

    elif isinstance(action, ActionMortgageSet):
        relationship = _set_mortgage(game, user, action.square, False)

    elif isinstance(action, ActionMortgageUnset):
        relationship = _unset_mortgage(game, user, action.square, False)
    elif isinstance(action, ActionNextPhase):
        current_money = game.money[str(user.pk)]


        if game.phase == GameManager.BUSINESS:
            if current_money >= 0:
                GameManager._next_turn(game, user)
            else:
                game.phase = GameManager.LIQUIDATION
```

```
                game.save()

        elif game.phase == GameManager.LIQUIDATION:
            if current_money >= 0:
                GameManager._next_turn(game, user)
            else:
                raise MaliciousUserInput(user, "Cannot end in NEGATIVE")
        return Response() # timers set in next turn

    GameManager._set_next_phase_timer(game, user)
    return Response()
```

3.2 Exceptions

Game Exceptions Module.

This module defines custom exceptions used throughout the game logic to handle internal errors, board configuration issues, and invalid or malicious actions performed by users.

3.2.1 `GameLogicError`

Bases: `Exception`

Exception raised for errors in the internal game logic.

This is typically thrown when the game reaches an impossible state or encounters an unexpected condition during normal execution.

Attributes:

Name	Type	Description
`message`	`str`	Explanation of the internal logic error.

- **Source code in `magnate/exceptions.py`**

```
class GameLogicError(Exception):
    """
    Exception raised for errors in the internal game logic.

    This is typically thrown when the game reaches an impossible state
    or encounters an unexpected condition during normal execution.

    Attributes:
        message (str): Explanation of the internal logic error.
    """
    def __init__(self, message=''):
        """
        Initializes the GameLogicError.

        Args:
            message (str, optional): Specific details about the logic error. Defaults to ''.
        """
        self.message = "Internal logic error: " + message
        super().__init__(self.message)
```

`__init__(message='')`

Initializes the GameLogicError.

Parameters:

Name	Type	Description	Default
`message`	`str`	Specific details about the logic error. Defaults to ''.	`''`

- **Source code in** `magnate/exceptions.py`

```
def __init__(self, message=''):
    """
    Initializes the GameLogicError.

    Args:
        message (str, optional): Specific details about the logic error. Defaults to ''.
    """
    self.message = "Internal logic error: " + message
    super().__init__(self.message)
```

3.2.2 `GameDesignError`

Bases: `Exception`

Exception raised for errors in the game's design or board configuration.

This is used when a square lacks required data (like rent prices) or when the board is fundamentally misconfigured (e.g., missing a Jail square).

Attributes:

Name	Type	Description
`message`	`str`	Explanation of the design error.

- **Source code in** `magnate/exceptions.py`

```
class GameDesignError(Exception):
    """
    Exception raised for errors in the game's design or board configuration.

    This is used when a square lacks required data (like rent prices) or
    when the board is fundamentally misconfigured (e.g., missing a Jail square).

    Attributes:
        message (str): Explanation of the design error.
    """
    def __init__(self, message=''):
        """
        Initializes the GameDesignError.

        Args:
            message (str, optional): Specific details about the configuration or design error. Defaults to ''.
        """
        self.message = "Internal logic error: " + message
        super().__init__(self.message)
```

`__init__(message='')`

Initializes the GameDesignError.

Parameters:

Name	Type	Description	Default
`message`	`str`	Specific details about the configuration or design error. Defaults to ''.	`''`

- **Source code in** `magnate/exceptions.py`

```
def __init__(self, message=''):
    """
    Initializes the GameDesignError.

    Args:
        message (str, optional): Specific details about the configuration or design error. Defaults to ''.
    """
    self.message = "Internal logic error: " + message
    super().__init__(self.message)
```

3.2.3 `MaliciousUserInput`

Bases: `Exception`

Exception raised when a user provides invalid, unauthorized, or potentially malicious input.

This acts as a security and validation layer, thrown when a user attempts to manipulate assets they do not own, or tries to exploit the game mechanics.

Attributes:

Name	Type	Description
`message`	`str`	Explanation of the malicious input, tagged with the user's primary key.

- **Source code in** `magnate/exceptions.py`

```
class MaliciousUserInput(Exception):
    """
    Exception raised when a user provides invalid, unauthorized, or potentially malicious input.

    This acts as a security and validation layer, thrown when a user attempts
    to manipulate assets they do not own, or tries to exploit the game mechanics.

    Attributes:
        message (str): Explanation of the malicious input, tagged with the user's primary key.
    """
    def __init__(self, user: CustomUser, message=''):
        """
        Initializes the MaliciousUserInput exception.

        Args:
            user (CustomUser): The user who triggered the exception.
            message (str, optional): Specific details about what the user attempted to do. Defaults to ''.
        """
        self.message = f"[{user.pk}] Potentially malicious input: " + message
        super().__init__(self.message)
```

`__init__(user, message='')`

Initializes the MaliciousUserInput exception.

Parameters:

Name	Type	Description	Default
`user`	`CustomUser`	The user who triggered the exception.	*required*
`message`	`str`	Specific details about what the user attempted to do. Defaults to ''.	`''`

- **Source code in** `magnate/exceptions.py`

```
def __init__(self, user: CustomUser, message=''):
    """
    Initializes the MaliciousUserInput exception.

    Args:
        user (CustomUser): The user who triggered the exception.
        message (str, optional): Specific details about what the user attempted to do. Defaults to ''.
    """
    self.message = f"[{user.pk}] Potentially malicious input: " + message
    super().__init__(self.message)
```

3.2.4 `MaliciousUserInputAction`

Bases: `MaliciousUserInput`

Exception raised when a user attempts an action that is not allowed in the current game phase.

This is a specialized form of MaliciousUserInput used heavily in the GameManager to enforce strict phase-based state machine transitions.

Attributes:

Name	Type	Description
`message`	`str`	Detailed message including the invalid action and the current game phase.

- **Source code in** `magnate/exceptions.py`

```
class MaliciousUserInputAction(MaliciousUserInput):
    """
    Exception raised when a user attempts an action that is not allowed in the current game phase.

    This is a specialized form of MaliciousUserInput used heavily in the GameManager
    to enforce strict phase-based state machine transitions.

    Attributes:
        message (str): Detailed message including the invalid action and the current game phase.
    """
    def __init__(self, game: Game, user: CustomUser, action: Action):
        """
        Initializes the MaliciousUserInputAction exception.

        Args:
            game (Game): The current game instance where the violation occurred.
```

```
            user (CustomUser): The user attempting the invalid action.
            action (Action): The forbidden action the user attempted to perform.
        """
        self.message = f"cannot perform action {action} in phase {game.phase}"
        super().__init__(user, self.message)
```

`__init__(game, user, action)`

Initializes the MaliciousUserInputAction exception.

Parameters:

Name	Type	Description	Default
`game`	`Game`	The current game instance where the violation occurred.	*required*
`user`	`CustomUser`	The user attempting the invalid action.	*required*
`action`	`Action`	The forbidden action the user attempted to perform.	*required*

- **Source code in** `magnate/exceptions.py`

```
def __init__(self, game: Game, user: CustomUser, action: Action):
    """
    Initializes the MaliciousUserInputAction exception.

    Args:
        game (Game): The current game instance where the violation occurred.
        user (CustomUser): The user attempting the invalid action.
        action (Action): The forbidden action the user attempted to perform.
    """
    self.message = f"cannot perform action {action} in phase {game.phase}"
    super().__init__(user, self.message)
```

3.3 Agent Heuristics: Probabilities and Dynamic Synergies

3.3.1 0. Core Concepts and Constants

Required Constants

- **LANDING_PROB (PROB_CAER):** 1/54 (Probability of landing on a specific square, excluding jail).
- **TOTAL_TURNS:** 500 (Used as a baseline to calculate remaining turns).
- **AUCTION_ROI_CTE:** 0.75 (Safety margin to ensure ROI in maximum bids).
- **FANTASY_CTE:** 0.0 (Expected net value of an unknown fantasy card).

Core Concepts

- **Expected Visits:** Landing Probability * Number of Opponents * Remaining Turns.
- **Dynamic Reserve:** The highest rent currently chargeable by any opponent on the board.
- **Rent Delta:** The gross change in rent income (including monopoly multipliers or station/bridge scaling) when gaining or losing a property.

3.3.2 1. Jail

- **EV_ExitJail:** Sum of EV of unowned buyables - Expected rent paid to others' properties - Bail cost (if applicable).
- **EV_StayInJail:** -EV_ExitJail.

3.3.3 2. Buying and Special Moves

- **Buyables (Properties/Bridges/Servers):** EV = (Rent Delta * Expected Visits) + Block Value - Buy Price+Mortgage Value (Residual Value)
- *Block Value:* Evaluated ONLY if an opponent already owns properties of the same group. Weighted by the number of opponents.
- **Trams:**
- **Take Tram:** EV = Uniform average EV of the next 12 linearly connected squares - Travel cost.
- **Skip Tram:** EV = Uniform average EV of the next 12 linearly connected squares from the current position.

3.3.4 3. Square Selection

The square with the highest expected value is chosen. * **Opponent's Property EV:** -Current Rent owed.

3.3.5 4. Business and Construction

- **Build:** EV = (Projected rent increase * Expected Visits) - Build price.
- **Demolish:** EV = Refund (Build price / 2) - (Rent loss * Expected Visits).
- **Unmortgage:** EV = (Rent Delta gained * Expected Visits) - Cost (Buy price / 2).
- **Mortgage:** EV = Cash gained (Buy price / 2) - (Rent Delta lost * Expected Visits). *(Note: Rent Delta automatically factors in the monopoly multiplier impact).*

3.3.6 5. Auctions

- **Max ROI Bid:** (Square Buying EV + Buy Price) * AUCTION_ROI_CTE.
- **Budget:** Current Money - Dynamic Reserve.
- **Max Bid:** Minimum between Budget and Max ROI Bid.
- **Real Bids:** Distributed across MaxBid or 0 (not to bid).

3.3.7 6. Liquidation and Surrender

- **Liquidate:** Handled inherently by EV sorting. Demolishing/mortgaging monopolies generates massive negative EV due to the Rent Delta penalty, forcing the agent to liquidate low-value single properties first.
- **Surrender:** Executed only if there are absolutely no other actions available to raise funds.

3.3.8 7. Trades

- **EV_Trade:** Own Net Benefit - (Rival Net Benefit / Number of Opponents).
- **Net Benefit:** (Money Gained + EV_properties_gained) - (Money Lost + EV_properties_lost).
- **Propose (Initiative):** Identify target properties and generate random offers prioritizing maximum EV_trade. Offer value is capped at 80% of the target's buying EV to ensure profitability.
- **Decision:** Accept or propose exclusively if EV_Trade > 0.