TicTacToe game system manual

1. Introduction

This project aims to create an engaging **Tic-Tac-Toe** game with enhanced functionality, including real-time gameplay and communication capabilities. Designed for multiplayer interaction, the platform enables users to play the classic game while simultaneously enjoying a video call and chat features for a more immersive experience.

Purpose of this project is to enhance the traditional Tic-Tac-Toe game by integrating modern technologies like WebRTC for video calls, WebSockets for real-time updates, and an intuitive user interface. The game serves as an interactive platform for players to connect, compete, and communicate, blending entertainment with social interaction.

In the realm of casual online gaming, many multiplayer games lack direct and immersive interaction between players. While traditional platforms focus on gameplay mechanics, they often neglect the social and communication aspects that make in-person games enjoyable. Players are left with disjointed experiences, switching between separate tools for chatting, video calls, or sharing gameplay updates. This creates a fragmented interaction that diminishes the sense of connection and engagement.

For simple games like Tic-Tac-Toe, which thrive on friendly competition and social interaction, this gap is even more pronounced. Most existing implementations only provide basic turn-based gameplay with minimal or no communication features, making them feel sterile and impersonal. There is a significant lack of platforms that seamlessly combine real-time gaming with face-to-face or conversational interactions.

This product allows you to initiate video chats using WebRTC, engage in text chat, and share the game state through WebSockets. The game state is stored in Redis after each move.

2. System architecture

You begin on the login page, where you enter your name. After submitting, you are redirected to tictactoe.html, and your name is passed via URL parameters.

When the DOM content is loaded, an API call is made to the backend's session controller to either create a new game session or join an existing one if another player is waiting. The drawBoard() method is called to render an empty game board, and the joinGame() method, with the gameSessionId passed as a parameter, subscribes to the WebSocket message broker at /websockets. Then subscriptions are established via the stompClient, which connects to the message broker at /topic/game/ + sessionId for game state updates and /topic/chatMessage/ + sessionId for chat messages.

There are three buttons: Make a Move, Send Message, and Start Call:

The Make a Move buttons are located on the game board and are triggered when clicked, marking the spot with an "X" or "O". On click, a gameSession object is sent to the backend message broker, which forwards it to /topic/game/{sessionId}. The backend first checks if the game has started using isGameStarted(). If it hasn't, a gameSession object with an empty board is returned via WebSockets. If the game has started, it validates the move using isMoveLegal(). If the move is not legal, the backend retrieves the last valid gameSession object from Redis and sends it back via WebSockets. If the move is legal, the

current gameSession object is stored in Redis and returned via WebSockets. The frontend then updates the game board using the drawBoard() function.

The Send Message button, when clicked, sends the user's typed message to /topic/chatMessage/{sessionId}. The message is then broadcast to all users subscribed to that session.

The Start Call button toggles between starting and disconnecting a video call. When clicked, it first checks if the user is already connected. If not connected, it updates the button label to "Start Call" and disconnects all video streams. If connected, the button label changes to "Disconnect." The process for starting a call includes creating a local media stream, displaying it in an HTML element, adding tracks to the localStream, and sending it to the Coturn server. It then generates an offer, which is sent to /signal/{sessionId}. Upon receiving the offer, the system sends an answer. Once the offer and answer are exchanged, a remote stream is created and added through Coturn, displaying it in another HTML element.

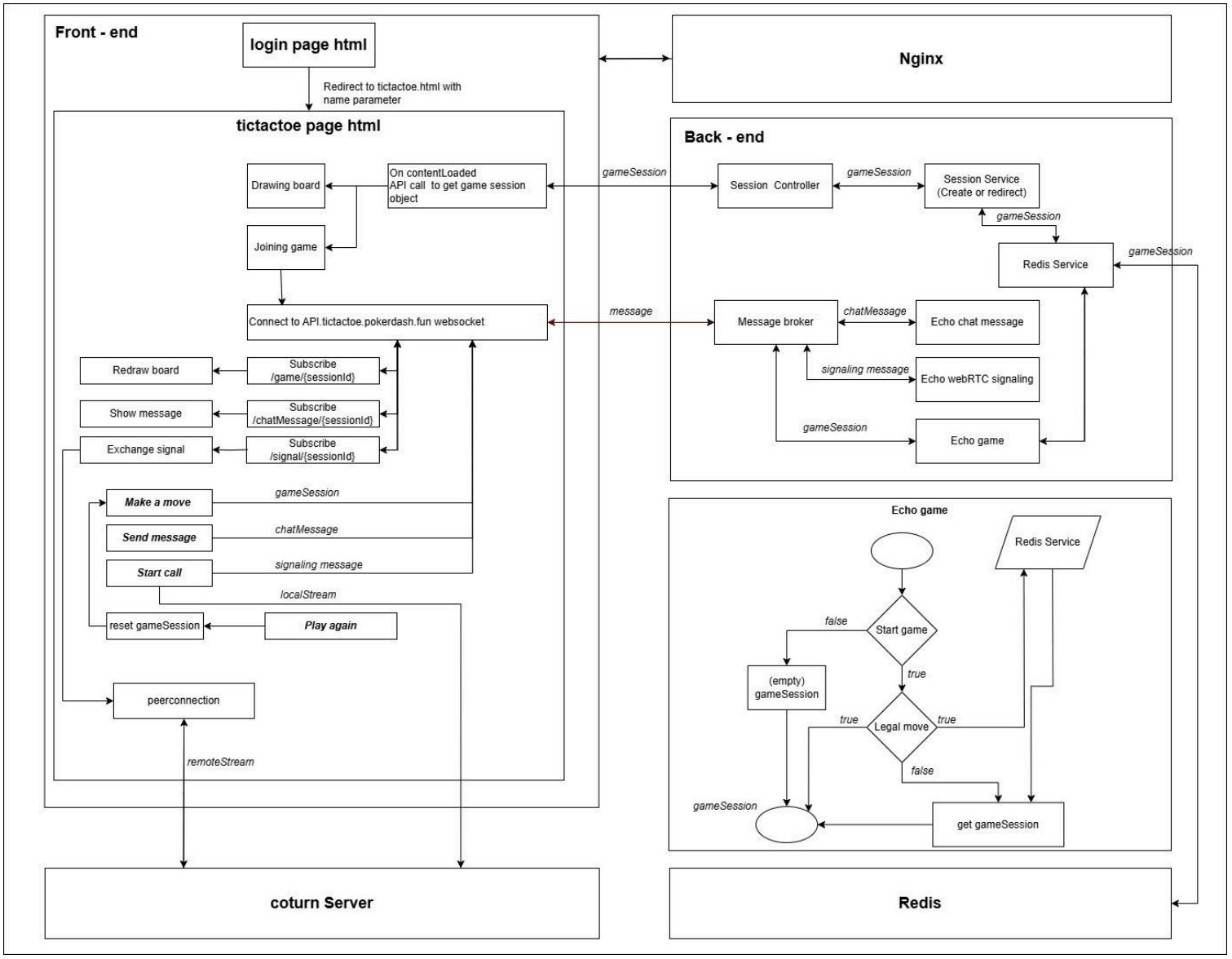

Figure 1. System structure diagram.

3. Installation instructions

Software and System Requirements.

Hosting Platform: Ubuntu Server.

Web Server: A web server Nginx to host the game application.

Application Server: Spring Boot. The backend framework responsible for handling WebSocket connections, game logic, and communication between clients.

Frontend Dependencies: HTML5, CSS3, and JavaScript for responsive UI design.

Real-time Communication and external libraries: SockJS for WebSocket fallback; STOMP.js for message protocol management; WebRTC for real-time video streaming. The server should support STOMP protocol over WebSocket for messaging and data updates.

In-memory Database: Redis: serves as the primary database for managing game session-based data, such as game state (e.g. player names, game session IDs, board status, player turns); temporary chat history.

SSL Certificates: Required for secure WebRTC and WebSocket communication. Certbot provided free SSL certificates.

TURN Server: WebRTC connectivity for establishing peer-to-peer video connections.

Installation.

Install Nginx web server:

```
sudo apt update
sudo apt install nginx -y
sudo systemctl enable nginx
sudo systemctl start nginx
```

make a new configuration file:

```
sudo nano /etc/nginx/sites-available/api.tictactoe.pokerdash.fun
```

Paste in the following configuration block and save:

```
server {
    server_name api.tictactoe.pokerdash.fun;
    location / {
        proxy_pass http://127.0.0.1:8080;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;
        proxy_redirect off;
        # WebSocket headers
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection 'upgrade';
    }
    # WebSocket handling (if you use a specific endpoint)
    location /ws/ {
        proxy_pass http://127.0.0.1:8080;  # Assuming WebSocket is served on
the same port
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;
        # WebSocket headers
```

```
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection "Upgrade";
    }
    listen 443 ssl; # managed by Certbot
    ssl_certificate
/etc/letsencrypt/live/api.tictactoe.pokerdash.fun/fullchain.pem; # managed by
Certbot
    ssl_certificate_key
/etc/letsencrypt/live/api.tictactoe.pokerdash.fun/privkey.pem; # managed by
Certbot
    include /etc/letsencrypt/options-ssl-nginx.conf; # managed by Certbot
    ssl_dhparam /etc/letsencrypt/ssl-dhparams.pem; # managed by Certbot

}
server {
    if ($host = api.tictactoe.pokerdash.fun) {
        return 301 https://$host$request_uri;
    } # managed by Certbot


    listen 80;
    server_name api.tictactoe.pokerdash.fun;
    return 404; # managed by Certbot
}
```

enable the file by creating a link from it to the sites-enabled directory:

```
sudo ln -s /etc/nginx/sites-available/api.tictactoe.pokerdash.fun
/etc/nginx/sites-enabled/
sudo systemctl reload nginx
```

Install Cerbot and set SSL certificates:

```
sudo apt install python3 python3-venv libaugeas0
sudo python3 -m venv /opt/certbot/
sudo /opt/certbot/bin/pip install --upgrade pip
sudo /opt/certbot/bin/pip install certbot certbot-nginx
sudo ln -s /opt/certbot/bin/certbot /usr/bin/certbot
sudo certbot --nginx
(select domain)
```

Install a Java Development Kit (JDK):

```
sudo apt update
sudo apt install openjdk-21-jdk -y
```

Install Supervisor to start programs in backgorund:

```
sudo apt update && sudo apt install supervisor
sudo nano /etc/supervisor/conf.d/tictactoe.conf
```

Paste in the following configuration block and save:

```
[program:tictactoe]
command=java -jar /home/ticTacToe-0.0.1-SNAPSHOT.jar
autostart=true
autorestart=true
```

```
stderr_logfile=/var/log/tictactoe.err.log
stdout_logfile=/var/log/tictactoe.out.log
```

Run the following commands:

```
sudo systemctl enable supervisor
sudo systemctl start tictactoe
```

Install Redis in-memory data store:

```
sudo apt update
sudo apt install redis -y
sudo systemctl enable redis
sudo systemctl start redis
```

Coturn install Turn server:

```
sudo apt update
sudo apt install coturn -y
sudo systemctl enable coturn
sudo systemctl start coturn
sudo turnserver -n -c /etc/turnserver.conf
```

Paste in the following configuration block and save:

```
# Basic settings
realm=tictactoe.pokerdash.fun
listening-ip=161.97.151.187
relay-ip=161.97.151.187
external-ip=161.97.151.187
userdb=/var/lib/turn/turndb
server-name=tictactoe.pokerdash.fun
fingerprint

# Security and Authentication
dh-file=/etc/letsencrypt/live/tictactoe.pokerdash.fun/dhparam.pem
cli-password=JavaVakariniai123
lt-cred-mech
user=user:JavaVakariniai123

# Session Lifetime
default-lifetime=3600
max-lifetime=86400

# Ports
listening-port=3478
tls-listening-port=5349

# Certificates for TLS
cert=/etc/letsencrypt/live/tictactoe.pokerdash.fun/cert.pem
pkey=/etc/letsencrypt/live/tictactoe.pokerdash.fun/privkey.pem

# Enable STUN and TURN over UDP and TCP
# Comment out `no-udp` to allow UDP, as its typically needed for WebRTC
#no-udp
```

```
# Logging
verbose

# Relay Ports
min-port=49152
max-port=65535

default-lifetime=3600
max-lifetime=86400
```

Run the following commands:

```
sudo systemctl restart coturn
sudo ufw allow 3478/tcp
sudo ufw allow 3478/udp
sudo ufw allow 5349/tcp
sudo ufw allow 49152:65535/udp
```

Firewall Configuration. Allow essential ports through the firewall:

```
sudo ufw allow 80     # Nginx HTTP
sudo ufw allow 443    # Nginx HTTPS
sudo ufw allow 3478   # Coturn
sudo ufw allow 22     # SSH
sudo ufw allow 8080   # Spring Boot
sudo ufw allow 6379   # Redis
sudo ufw enable
```

If your Coturn server is not functioning as expected, a few steps can help identify and resolve the problem. First, consider reinstalling Coturn to ensure a clean setup and remove any corrupted or incomplete files from the previous installation. During reinstallation, ensure no residual configuration files from the previous installation interfere with the new setup. Next, check your firewall settings to confirm that all necessary ports are open. This includes ports 3478 for STUN/TURN traffic, 5349 for TLS-secured communication, and the relay port range, typically 49152–65535, required for TURN services. Proper firewall configuration is crucial for uninterrupted service.

Additionally, review the configuration file located in /etc/turnserver.conf. Ensure that all settings match your requirements, such as IP addresses, authentication methods, and certificate paths. One critical setting to check is the no-udp option; if this line is uncommented, UDP traffic will be blocked, which is essential for many WebRTC applications. Comment out or remove this line to enable UDP traffic. Once you have confirmed the configuration is correct, restart the Coturn service to apply the changes. Regularly monitor system logs to identify any errors or warnings that can provide insights into issues requiring attention. These steps will help ensure your Coturn server operates smoothly and reliably.

4. User guide

To start a game, visit https://tictactoe.pokerdash.fun/. On the main page, enter your name in the provided field and press the Login button. This action takes you to the game lobby, where you wait for a second player to join. The interface is simple and intuitive, making it easy for players to navigate and start playing quickly.

Once the second player connects, the game board becomes active. The board features buttons for placing X or O symbols, which are only enabled when it is your turn to play and both players are connected. This ensures that gameplay proceeds in an orderly and interactive manner. The platform also includes a Play Again button that allows players to reset the board after the game concludes, making it convenient to start a new round without leaving the session.

Additionally, there is a built-in chat feature that lets players communicate throughout the game. The chat area includes a text box where players can type messages and a Send Message button to share them in real time. This enhances the social aspect of the game, creating a more engaging and personal experience.

For an even more immersive interaction, the platform includes a Video Call button. If both players click this button, a video call is initiated, enabling face-to-face communication during the game. This feature takes the casual gameplay of Tic-Tac-Toe to another level, making it a fun and interactive way to connect with friends or other players.

Finally, the Connect button for video calls transforms into a Disconnect button once clicked, providing an easy way to end the video conversation when desired. This thoughtful design ensures seamless transitions between connected and disconnected states, keeping the experience user-friendly and enjoyable.

API and WebSocket Functionality

The platform includes several API calls and WebSocket-based interactions to facilitate real-time gameplay, chat communication, and video call functionality. Here's an overview of the available API and WebSocket features:

1. API Call:

Endpoint: GET /getGameSession/{name}

Functionality: This API call retrieves a *gameSession* object, which is essential to start the game. The *gameSession* object contains details such as the session ID, player information, and the current state of the game.

Example gameSession Object:


```
{
  "sessionId": "dfecf88f-a012-43af-89e1-161e19339d2d",
  "player1": "Sergejus",
  "player2": null,
  "boardStatus": [
    " ",
    " ",
    " ",
    " ",
    " ",
    " ",
    " ",
    " ",
    " ",
    " "
  ],
  "playerOneMove": false
}
```


Players use this endpoint when logging in or joining a session to initialize the game environment and synchronize their game state with the server. The API returns a

gameSession object containing details about the game session, including session ID, player information, board status, and the current player's turn.

2. WebSocket channel: */topic/chatMessage/' + sessionId*

The WebSocket enables real-time chat functionality, allowing players to communicate seamlessly during the game. It listens for and echoes a *chatMessage* object, broadcasting the message to all connected players in the session. This ensures that communication happens instantly and enhances the interactive experience between players.

Example of *chatMessage* Object:

```
{
    "name": "name",
    "message": "I will win!",
    "localDateTime": "2024-11-21 14:30:00"
}
```

This feature creates a dynamic environment where players can exchange friendly banter or comment on the game's progress in real-time. By integrating communication directly into the game interface, it bridges the gap between gameplay and social interaction, making the experience more immersive and enjoyable.

3. WebSocket channel: *topic/game/' + sessionId*

The WebSocket channel facilitates real-time game state updates, allowing players in a specific session to interact seamlessly during gameplay. It listens for and echoes a gameSession object, ensuring that any changes in the game state, such as player moves or the game outcome, are broadcasted instantly to connected players in the session.

Example gameSession Object:

```
{
  "sessionId": "dfecf88f-a012-43af-89e1-161e19339d2d",
  "player1": "Sergejus",
  "player2": "Jonas",
  "boardStatus": [
    " ",
    "O",
    "X",
    " ",
    " ",
    "X",
    "O",
    " ",
    " "
  ],
  "playerOneMove": false
}
```

This feature ensures that players in the session remain synchronized in real time, with immediate updates to the game board, current turn, and game outcome. By integrating this functionality directly into the game platform, it creates a highly responsive and engaging gameplay experience, enhancing the sense of competition and interaction between players.

4. WebSocket channel: */topic/signal/" + sessionId*

The WebSocket channel enables real-time WebRTC signaling, facilitating the setup and management of video call connections between players in a specific session. It listens for and echoes signaling messages, ensuring that data required for peer-to-peer communication, such as session descriptions and ICE candidates, is transmitted instantly to all connected participants.

```
{
  "type": "offer",
  "sdp": "v=0\r\no=- 4611734056136356383 2 IN IP4 127.0.0.1...",
}
```

Message Types:

- offer/answer: Contain session descriptions needed to establish the video call.
- candidate: Include ICE candidates to assist with NAT traversal and peer connection.

This feature ensures that WebRTC connections are established and maintained smoothly, allowing players to initiate and sustain video calls during gameplay. By handling signaling directly through the WebSocket channel, the platform provides a seamless and real-time mechanism for creating video connections, enhancing interaction and making the gaming experience more immersive and social.

5. CoTURN Server Integration

The platform integrates a CoTURN server to ensure reliable peer-to-peer connectivity for video calls, even in challenging network environments. CoTURN serves as a relay for WebRTC connections when direct communication between players is not possible due to NAT (Network Address Translation) or firewall restrictions. By facilitating the exchange of media data through a relay, the CoTURN server ensures stable video and audio communication, enhancing the platform's usability and ensuring a seamless video calling experience regardless of the players' network configurations. This integration is crucial for providing a consistent and uninterrupted interaction between players during gameplay.

5. Techninės detalės

Java Back-end

Models:

ChatMessage:

- private String name;
- private String message;
- private LocalDateTime localDateTime;
- private MessageType type;

GameSession:

- private String sessionId;
- private String player1;
- private String player2;
- private Character[] boardStatus;
- private boolean playerOneMove;

MessageType (enum):

- REGULAR

- SYSTEM

Controllers:

SessionController:

- public ResponseEntity<GameSession> getGameSession(@PathVariable String name)

WebsocketsController:

- public String handleSignalMessage(String message)
- public ChatMessage echoChatMessage(ChatMessage chatMessage)
- public GameSession chatMessage(GameSession gameSession)
- public boolean isMoveLegal(GameSession gameSession)
- public boolean startGame(GameSession gameSession)
- public void setTurn(GameSession gameSession)
- public GameSession createEmptyBoard(GameSession gameSession)

Services:

RedisService:

- public RedisService(@Value("${redis.host}") String host, @Value("${redis.port}") int port)
- public void put(String key, GameSession gameSession)
- public GameSession getGameSession(String key)
- public Set<GameSession> getAllGameSessions()
- public boolean deleteGameSession(String key)
- public void close()

SessionService:

- public synchronized GameSession getGameSession(String name)
- private String getKey()

Security:

WebConfig:

- public WebMvcConfigurer corsConfigurer()

WebSocketController:

- public void configureMessageBroker(MessageBrokerRegistry config)
- public void registerStompEndpoints(StompEndpointRegistry registry)

Technologies used:

- Spring Boot Starter Web
- Spring Boot Devtools
- H2 Database
- Websocket
- Jedis
- Jackson 2.15.2
- Maven 21

JavaScript, CSS, HTML Front-end

JS:

login.js:

- async function createLoginContainer(containerID)
- function checkIfNameValid(username)
- function getUserData()
- function clearUserData()

chatController.js:

- function sendMessage()

- async function writeAComment(containerID, comment)
- async function addCommenbt(comment, shifted)
- function scrollToBottom()

index.js:

- createLoginContainer("main-container")

tic-tac-toe.js:

- function makeMove(index)
- function checkWin()
- function resetBoard()
- function drawBoard()

webrtc.js:

- async function joinWebCRT()
- function handleSignalingMessage(message)
- function sendSignalingMessage(message)
- async function startCall()
- peerConnection.onicecandidate = event =>
- peerConnection.oniceconnectionstatechange = () =>
- peerConnection.onsignalingstatechange = () =>
- peerConnection.ontrack = event =>
- function addIceCandidate(candidate)
- function processBufferedCandidates()
- function setRemoteAnswer(answer)
- function disconnectVideo()
- function connectVideo()
- function disconnectMediaStream(stream)

websocket.js:

- async function getSessionId(name)
- async function joinGame(sessionId)
- function sendGameStatus(gameSession)
- async function sendMessageToAPI(message)

HTML:

index.html
tic-tac-toe.html

CSS:

Login.css
chatController.css
index.css
Tic-tac-toe.css

Technologies used:

- SockJS client Latest
- Stomp 2.3.3
- WebRTC

Other technologies:

- Ubuntu
- Nginx
- Redis

- Coturn

Third-party integrations:

- Google Ads

6. Maintenance and Support

The platform is designed to be scalable and maintainable, enabling seamless updates and extensions with minimal disruption to ongoing sessions. Its modular design allows for the smooth integration of new features or enhancements without impacting core gameplay.

However, the absence of a logging mechanism currently affects the platform's scalability, as it limits the ability to monitor system performance and identify bottlenecks during increased usage. To address this, implementing a structured logging system is essential. Logs should capture details such as timestamps, session IDs, player actions, and error messages. Integrating tools like ELK Stack (Elasticsearch, Logstash, and Kibana) or Grafana would enable real-time monitoring, helping to manage performance under high loads and ensuring the system scales effectively while maintaining reliability.

For future expansion, the backend is structured to support modifications to the *gameSession* object. This design enables the platform to incorporate additional games beyond Tic-Tac-Toe. For instance, fields and properties can be added to the *gameSession* object to accommodate the mechanics of other games, such as Snake, while maintaining compatibility with the existing infrastructure. This flexible backend setup minimizes the need for extensive rework, ensuring quick and efficient implementation of new features.

On the frontend, the interface is designed to support a variety of game types. Its modular layout allows different game interfaces to be embedded within dedicated div containers. For example, while Tic-Tac-Toe occupies one section of the screen, another game like Snake can be rendered in a separate, configurable div. This approach simplifies integration and ensures that new games are visually distinct, maintaining a consistent and intuitive user experience.

By ensuring scalability at both backend and frontend levels, along with robust logging and monitoring capabilities, the platform is well-prepared to grow and adapt, providing users with a dynamic and evolving gaming environment.

Potential Issues and Solutions

One issue with the platform is that a single player can open too many game sessions, which leads to excessive resource usage and unnecessary clutter in the system. This can impact overall efficiency and performance. To resolve this, the system can periodically clear unused or inactive sessions from Redis, ensuring that only active sessions occupy resources.

Another problem is the lack of robust security measures. Without authentication or session validation, the platform is vulnerable to unauthorized access and misuse. This could be addressed by adding secure user authentication, validating sessions, and using secure WebSocket connections. Enforcing limits on the number of sessions a user can create would also prevent abuse and enhance the platform’s security.

A further issue is that there’s no immediate indication when the second player connects to a game session. The first player only notices their presence when the second player makes a move, which can cause confusion. Adding a real-time notification system to inform the first player as soon as the second player connects would improve the user experience and reduce uncertainty.

Additionally, if the first player presses the Connect to Video button before the second player joins, the video call doesn't initialize automatically when the second player connects. This forces the first player to refresh or reconnect to enable the video feature. To solve this, the system could queue video connection requests and trigger the WebRTC setup as soon as both players are present. Temporarily disabling the Connect to Video button until both players are connected would also help prevent this issue. Addressing these problems would enhance the platform's functionality, reliability, and user experience.

If you encounter any bugs or problems please contact us on tictactoe@weneverreply.com

7. Log Format and Monitoring

Logging and monitoring are not currently implemented in the system.

1. Įžanga
Aprašykite projekto tikslą.
Paaiškinkite, kokią problemą projektas sprendžia.
Trumpai apibūdinkite produkto ar sistemos funkcionalumą.
2. Sistemos architektūra
Įtraukite sistemos struktūrą: diagramos, blokų schemos ar modeliai (pvz., UML).
Paaiškinkite pagrindinius komponentus ir jų funkcijas.
Apibūdinkite, kaip komponentai sąveikauja tarpusavyje.
3. Diegimo instrukcijos
Programinės įrangos ar sistemos reikalavimai (pvz., serveriai, duomenų bazės, priklausomybės).
Diegimo žingsniai:
Programinės įrangos atsisiuntimas ir diegimas.
Konfigūracijos nustatymai (pvz., aplinkos kintamieji, API raktai).
Galimos problemos ir jų sprendimo būdai.
4. Vartotojo vadovas
Kaip naudoti produktą ar sistemą:
Naudotojo sąsajos (jei yra) aprašymas.
API galimybės ir naudojimo pavyzdžiai.
Tipiniai naudojimo scenarijai (angl. use cases).
5. Techninės detalės
Kodo struktūra:
Klasės, moduliai, funkcijos, failų išdėstymas.
Naudojamos technologijos, bibliotekos ir jų versijos.
Integracijos su trečiųjų šalių paslaugo**mis.**
6. Priežiūra ir palaikymas
Sistemos atnaujinimo ar plėtimo procesai.
Probleminių situacijų sprendimo būdai.
Kontaktai, jei reikalinga techninė pagalba.
7. Žurnalo formatas ir stebėjimas
Logų struktūra (jei sistema generuoja žurnalus).
Stebėjimo įrankiai ir rekomendacijos.