SUROGA WebRTC Streaming Service Manual

This document aims to describe the SUROGA web streaming service: server configuration strategy, Janus server installation, API description for stream channel setup and reservation, current front-end version overview, and integration with applications.

Table of Contents

---

0. Introduction

The SUROGA streaming service is exposed through a robust API. Initially tested with the SUROGA application, the service is now evolving to be provider-agnostic, enabling smooth integration with platforms like Greenways and other third-party streaming solutions.

1. Server Selection and Configuration

For a cost-sensitive WebRTC service limited to the EU (typically 1-on-1 sessions or a few concurrent streams), virtual machine (VM) selection and network configuration are critical.

1.1 Azure VM Selection

• Dsv5 Series (e.g., Standard_D2s_v5, D4s_v5)

- **Modern CPU performance**: 3rd Gen Intel Xeon (Ice Lake) processors provide superior single-thread performance, essential for signaling operations and software-based codecs.
- **High bandwidth networking**: Supports Accelerated Networking, offering low latency, high throughput, and reduced jitter via SR-IOV.
- **Location**:

    - Recommend deployment in West Europe (Netherlands) or North Europe (Ireland) for low round-trip latency (<50 ms median).
    - Other options like Germany West Central (Frankfurt) may be evaluated, but West/North Europe generally balance cost and latency well.
  - **Disk/storage**:
    - For 1-on-1 streaming, persistent storage is not critical. Local SSD (temporary disk) is sufficient for short-term logs or buffering.

• B Series (e.g., B2s, B4ms)

- **Burstable, low-cost**: Suitable for intermittent workloads with CPU spikes (e.g., Janus instances primarily handling signaling or occasional recording).
- **Optimized for light 1-on-1 use**: If each Janus instance handles a single stream and is idle often, B2s can significantly reduce costs.
- **Baseline considerations**: If CPU usage exceeds baseline often (e.g., for transcoding), performance throttling may occur. Begin testing with D2s_v5, then downgrade to B2s if feasible.

• Spot Instances (Optional)

- **Up to ~90% discount**: Useful for non-critical or test workloads where VM pre-emption is acceptable.
- **Eviction risk**: Not recommended for real-time 1-on-1 sessions unless failover strategies are in place.

• Optimized Networking

- **Accelerated Networking**: Should be enabled where supported to reduce jitter/latency and CPU overhead.
- **Additional settings**:
  - Use Proximity Placement Groups for multi-VM setups (e.g., high availability), though often unnecessary in 1-on-1.
  - Configure NSGs (Network Security Groups) to only allow required ports (see firewall section).

Initial Recommendation

- **Preliminary Testing**:
  1. Deploy a Standard_D2s_v5 VM with Accelerated Networking in West Europe.
  2. Validate CPU/network behavior during simulated WebRTC sessions from EU-based clients.
  3. If average CPU usage is low, consider migrating to B2s for cost savings.
- **Monitoring**:

    1. Use Azure metrics to monitor CPU credits (B Series) and network latency/jitter.
    2. Adjust VM choice accordingly.
- **Estimated Cost**:
    1. Varies by region and reservation. EU regions tend to be ~10–20% cheaper than global averages.

1.2 Automation via API

To optimize costs, VMs should only run when needed.

- **Scheduled provisioning**:
    - With 24h notice, schedule VM startup ~30 minutes before stream, ensuring Janus/TURN readiness.
    - Automatically stop/deallocate VM after session, allowing a grace period for delays.
- **Azure CLI/SDK**:
    - Use `az vm start` and `az vm deallocate` to control VM lifecycle.
    - Example:

      ```
      az vm start --resource-group MyRG --name JanusVM
      az vm deallocate --resource-group MyRG --name JanusVM
      ```

    - Integrate commands into CI/CD scripts, management servers (cron), or Azure Automation Runbooks.
    - Azure SDKs (Python, .NET, etc.) allow programmatic control aligned with business logic.
- **Central Scheduling**:
    - Maintain a reservation database or calendar service.
    - Use a Job Scheduler (cron, Azure Function) to periodically check reservations and start/stop VMs accordingly.
    - Log success/failure and alert if VM state changes do not complete as expected.
- **Manual Override**:
    - Provide an admin interface for manual VM control (e.g., extend session on client request).
    - Ensure strong authentication (MFA) and logging of manual actions.
- **Startup Latency Consideration**:
    - Cold starts may take several minutes. Schedule ~30 minutes before events.
    - Use pre-warmed VMs during high-usage windows to reduce latency.
- **Spot Instance Handling**:
    - Scripts must detect eviction and reassign workload (fallback VM or reschedule).

1.3 Forward Planning

- **Central Calendar & Load Forecasting**:
  - Maintain a weekly/monthly plan with session start/end times and stream counts.
  - Estimate needed VMs. For 1-on-1, typically one VM per session. Overlap may require extra VMs.
- **Smart Allocation**:
  - For closely timed sessions, keep VMs alive and reuse them.
  - For sporadic usage, shut down immediately after sessions.
- **Billing Integration**:
  - Align VM runtime with client billing.
- **Monitoring & Alerts**:
  - Automatically adapt to reservation changes (e.g., cancellations).
  - Notify operations team weekly with schedules and manual check needs.
- **Periodic Testing**:
  - In off-peak times, spin up test VMs to validate start/stop logic, image updates, and Janus setup.

2. Janus Server Firewall Configuration

To ensure proper operation of Janus as a WebRTC server, it is essential to open and forward specific ports and correctly handle NAT behavior.

2.1 Required Ports

HTTP/HTTPS and WebSocket Access

- **HTTP (for Janus REST APIs or web UI)**:
  - TCP port **8088** by default.
  - Alternatively, expose via a reverse proxy (e.g., Nginx) mapping **TCP 80** to localhost:8088.
- **HTTPS and Secure WebSocket (WSS)**:
  - Janus supports HTTPS on **TCP 8089**, but it is recommended to use a reverse proxy (Nginx, Traefik, Apache) on **TCP 443** for TLS termination.
  - Configure reverse proxy to forward to Janus' internal ports: 8088/8089 (REST/HTTPS) and 8188/8989 (WS/WSS).
  - WSS is typically served via **TCP 8989**, configured in `janus.transport.websockets.jcfg` (ensure `ws_ssl=true`).

RTP/RTCP (UDP Media Traffic)

- Janus uses a configurable UDP port range defined in `rtp_port_range` in `janus.jcfg`.
    - Common ranges: **10000–20000** or **20000–20500**.
    - Open this UDP range for both inbound and outbound traffic in firewall/NSG.

TURN/STUN (Optional, if using your own server)

- **TURN**:
    - Typically on **UDP 3478** and **5349**.
    - For restrictive networks, TURN over TCP/TLS via port **443** may be needed.
- **STUN**:
    - Uses **UDP 3478**.
    - If using an external STUN server, only outbound traffic is required.

Admin API Port (Optional)

- If Janus' Admin API is enabled, restrict access by IP or limit it to private network only.

2.2 NAT Configuration

1:1 NAT Mapping

- If Janus is behind NAT, configure `nat_1_1_mapping = "PUBLIC_IP"` in `janus.jcfg` to advertise the correct external IP in SDP ICE candidates.

Firewall/NAT Forwarding

- At Azure NSG or external firewall level:
    - Forward **UDP 10000–20000** from the public IP to Janus' internal IP.
    - Forward **TCP 80/443** to the proxy server forwarding to Janus.
    - Ensure any health checks (e.g., Azure Load Balancer probes) target signaling ports (HTTP/HTTPS), not RTP.

Hairpin NAT / Loopback NAT

- If clients behind the same NAT access Janus via a public domain, ensure the NAT device supports hairpin NAT (loopback) so the VM can resolve and connect to its own public IP.

2.3 Enhancing Accessibility in Restricted Networks

TCP 443 Multiplexing

- For networks that block UDP:
    - Use reverse proxy (e.g., Nginx) to forward WSS and TURN traffic over **TCP 443**.
    - Configure SNI-based or path-based TLS routing:
        - Example: `/janus` to Janus WSS, `/turn` to TURN server.

TLS and Certificates

- Use a trusted certificate authority (e.g., Let's Encrypt) for HTTPS/WSS.
- Automate certificate renewal on the reverse proxy to maintain availability and avoid browser warnings.

Firewall Hardening

- Only allow trusted IPs for admin interface access.
- For public signaling endpoints, open only **80** and **443**.
- Block unnecessary ports and restrict SSH access to specific IPs (e.g., via jump host or VPN).

Traffic Monitoring

- Monitor NSG/firewall logs to detect suspicious access or scanning on RTP ports.
- Set alerts for unusual traffic spikes, which could indicate a DDoS attempt.

Azure DDoS Protection (Optional)

- Enable Azure DDoS Standard Protection for critical VMs.
- Helps detect and mitigate large-scale network attacks automatically.

3. When to Use a Load Balancer with Janus

Although the typical scenario may involve low concurrency and 1-to-1 sessions, understanding when to introduce a load balancer is crucial for future scalability planning.

3.1 Scenarios Requiring a Load Balancer

- **Horizontal Scalability / Multiple Janus Instances:**
    - If at any point multiple concurrent sessions are needed (even if each session is 1-to-1), deploying several Janus VMs behind a load balancer allows session distribution across available servers.
    - **High Availability:** If one instance fails, the load balancer can redirect new sessions to healthy nodes, ensuring service continuity.
- **Zero Downtime Maintenance:**
    - Instances can be removed for maintenance while others continue handling connections without service disruption.

3.2 Sticky Sessions in WebRTC

- **Session Persistence (Affinity):**
    - WebSockets require that, after the handshake, the connection remains with the same Janus instance since the PeerConnection state is held in-memory.
    - On Azure, Azure Load Balancer supports "source IP affinity". Alternatively, Azure Application Gateway offers WebSocket support with cookie or connection-based affinity. However, WebRTC usually uses WSS without cookies, making IP-based or path-hash affinity (e.g., including `room_id` in the path) more suitable.

- **Alternative: DNS-Based Routing:**
    - This approach involves client-side logic to query instance load and select a specific URL, though it introduces added complexity.
- **Consistent Hashing via Proxy (Nginx):**
    - If not using Azure LB, Nginx or HAProxy can apply consistent hashing based on the path (e.g., `/janus/<session-id>`) to always route requests to the same backend.
    - Health checks must be non-disruptive to active sessions.

3.3 When a Load Balancer May Not Be Needed

- **Low Usage, Single Instance:**
    - If demand is minimal (1-4 simultaneous sessions) and immediate failover is unnecessary, a single Janus VM can operate at lower complexity and cost.
- **Client-Side Fallback or DNS Round-Robin:**
    - For a few instances, clients can attempt connection using a predefined list of addresses. If one fails, the client retries another. This offers basic redundancy without session stickiness.
- **Operational Overhead:**
    - Load balancers introduce extra infrastructure, configuration (health probes, affinity), and potential licensing/proxy costs. Evaluate if HA/scale benefits outweigh complexity in low-usage environments.

---

3. Janus Installation

This section details the preparation of the server, Janus installation from an internal Git repository (DevOps), service configuration, and update/rollback scripting.

3.1 Preparing the Server to Run Janus

3.1.1 OS and Basic Environment

- **Recommended OS:** Ubuntu 20.04 LTS or newer.
    - Widely tested and supported by Janus dependencies.
- **System Updates:**

```
sudo apt-get update && sudo apt-get upgrade -y
```

- **Time Synchronization:**

```
sudo apt-get install -y chrony
sudo systemctl enable chrony && sudo systemctl start chrony
```

- **Dedicated User:**

```
sudo useradd -r -s /usr/sbin/nologin janus
```

- **Directories:**

```
sudo mkdir -p /opt/janus/{bin,etc,var,log}
sudo chown $USER:$USER /opt/janus -R
```

- **System Limits (ulimits):**

```
/etc/security/limits.d/janus.conf
janus soft nofile 65536
janus hard nofile 65536
```

- **sysctl Configuration:**

```
sudo sysctl -w fs.file-max=100000
```

- **Network / Kernel Tuning:**

```
sudo sysctl -w net.ipv4.ip_forward=1
```

- **Install Dependencies:**

```
sudo apt-get install -y \
  build-essential git pkg-config automake autoconf libtool cmake \
  libmicrohttpd-dev libjansson-dev libssl-dev libsrtp2-dev libsofia-sip-ua-dev \
  libglib2.0-dev libopus-dev libogg-dev libcurl4-openssl-dev liblua5.3-dev 
libnice-dev \
  libwebsockets-dev libavformat-dev jq
```

- **Firewall:**

```
sudo ufw allow OpenSSH
sudo ufw enable
```

3.1.2 Source Code Retrieval

- Use an internal Git (Azure DevOps) or mirrored repo from upstream.
- Always pin to stable tags (e.g., `v1.10.6`) for reproducibility.

3.2 Installation via DevOps Pipeline

Refer to the YAML pipeline provided for automated build/install using variables like `$(JanusRepoUrl)` and `$(JanusVersion)`.

Key best practices include:

- Clean checkouts.
- Controlled variables for flexibility.
- Minimal plugin enablement to reduce dependencies.
- Post-install permission changes for the `janus` user.
- systemd service with `ExecStart` pointing to Janus binary and log path.

3.2.1 Server Config Before Janus Startup

- **TLS Certificates:** Configure HTTPS for Janus directly or via a reverse proxy (Nginx, Traefik).

- **Environment Variables:** Set `LD_LIBRARY_PATH=/opt/janus/lib` if needed.
- **Logrotate:** Avoid log growth with daily rotation under `/etc/logrotate.d/janus`.
- **Monitoring:** Install agents (e.g., Azure Monitor) and configure alerts.

3.3 Update Script for Production Deployments

- Includes a build step, backup of current install, validation, and controlled restart.
- Backup naming with timestamps allows rollback.
- Cleanup step to remove older backups.
- Health check post-restart to ensure the new version is running.

3.3.1 Directory-Based Versioning (Alternative)

- Install into `/opt/janus-<version>`.
- Use a symlink `/opt/janus_current` for service reference.
- Allows parallel versioning and easier rollback.

3.3.2 Configuration Files and Plugins

- Main config: `janus.jcfg`
- Customize network interfaces, NAT mappings, and plugin parameters as needed.

This structure ensures a robust, repeatable, and scalable setup for Janus deployments in production environments.

4. Booking API Containerization and Deployment

This section details how the Booking API is packaged, configured, and executed within a Docker container, and how it is expected to integrate with the broader Suroga platform in the future.

4.1 API Containerization

Purpose

The Booking API is containerized using Docker to ensure consistent deployment across different environments (development, staging, production), and to support future orchestration alongside other Suroga services such as authentication, frontend UI, Janus gateway proxies, etc.

Dockerfile Structure

A typical Dockerfile for a Django/DRF-based API includes:

```
# Lightweight base image with Python
FROM python:3.11-slim

# Set working directory
WORKDIR /app

# Copy dependency list
```

```
COPY requirements.txt .

# Install required system packages (e.g., for psycopg2)
RUN apt-get update && apt-get install -y \
    build-essential libpq-dev netcat \
    && rm -rf /var/lib/apt/lists/*

# Install Python dependencies
RUN pip install --no-cache-dir -r requirements.txt

# Copy project files
COPY . .

# Set environment variables
ENV DJANGO_SETTINGS_MODULE=myproject.settings.prod
ENV PYTHONUNBUFFERED=1

# Optional: prepare log directory
RUN mkdir -p /vol/log && chown -R nobody:nogroup /vol/log

# Optional: collect static files or run migrations
# RUN python manage.py collectstatic --noinput

# Launch using a WSGI server
CMD ["gunicorn", "myproject.wsgi:application", "--bind", "0.0.0.0:8000", "--
workers", "4"]
```

Best Practices

- **System Dependencies:** Keep minimal, e.g., `libpq-dev` only if using PostgreSQL.
- **Multi-stage Builds:** Split build and runtime if compiling frontend assets to reduce final image size.
- **Non-root User:** Improve security by running under a non-privileged user:

```
RUN addgroup --system django && adduser --system --ingroup django django
USER django
```

- **Volumes:** Map external volumes if generating temporary files or logs.
- **Environment Configuration:** Use `DATABASE_URL`, `JANUS_URL`, `JANUS_KEY`, `SECRET_KEY`, `DEBUG=False`, `ALLOWED_HOSTS`, etc., as environment variables.

Environment Separation

- Pass environment-specific values through the CI/CD pipeline or orchestrator (Kubernetes, Azure App Service, Docker Compose, etc.).
- Maintain a `.env.example` file (non-sensitive) for developer reference.

Health Check

- Implement a `/health/` endpoint to return an OK status when:
    - Database connectivity is healthy.
    - (Optional) Janus API is reachable (lightweight HTTP check).
- Used by orchestrators to restart containers upon failure.

Logging & Metrics

- Configure Django logging to `stdout`/`stderr` for compatibility with log collectors (e.g., Azure Monitor, ELK).
- Optionally expose Prometheus-compatible metrics.
- Log rotation is typically delegated to the container runtime.

External Dependencies

- **Database:** PostgreSQL (managed or in a separate container).
- **Janus Gateway:** API calls to `JANUS_URL` (e.g., `https://janus.example.com/janus`).
- **Authentication:** Initially uses JWT or bearer tokens; full integration with Suroga API planned.

4.2 Container Deployment via DevOps

CI/CD Pipeline

- Code resides in a Git repository (Azure DevOps Git, GitHub, etc.).
- Build pipeline includes:
    1. Checkout from main/release branch.
    2. Run automated tests (unit, lint, coverage).
    3. Build Docker image:
        - Tag image semantically (e.g., `registry.azurecr.io/suroga-booking-api:v1.2.3`).
    4. (Optional) Scan for vulnerabilities.
    5. Push image to container registry.
    6. (Optional) Publish build artifacts (test reports, logs).

```
trigger:
  branches:
    include:
      - main

pool:
  vmImage: 'ubuntu-latest'

variables:
  imageName: 'suroga-booking-api'
  registry: 'myregistry.azurecr.io'

stages:
- stage: Build
  displayName: 'Build and Push Docker Image'
  jobs:
  - job: BuildAndPush
    displayName: 'Build & Push'
    steps:
```

```
    - task: Checkout@1

    - task: Bash@3
      displayName: 'Run Tests'
      inputs:
        targetType: 'inline'
        script: |
          pip install -r requirements.txt
          pytest --maxfail=1 --disable-warnings -q

    - task: Docker@2
      displayName: 'Build Docker Image'
      inputs:
        containerRegistry: 'MyACRServiceConnection'
        repository: '$(registry)/$(imageName)'
        command: 'buildAndPush'
        Dockerfile: '**/Dockerfile'
        tags: |
          $(Build.BuildId)
          latest

- stage: Deploy
  displayName: 'Deploy to Environment'
  dependsOn: Build
  jobs:
  - deployment: DeployAPI
    displayName: 'Deploy Booking API'
    environment: 'production'
    strategy:
      runOnce:
        deploy:
          steps:
          - task: Bash@3
            displayName: 'Deploy container'
            inputs:
              targetType: 'inline'
              script: |
                az webapp config container set \
                  --name suroga-booking-api \
                  --resource-group MyRG \
                  --docker-custom-image-name $(registry)/$(imageName):$
(Build.BuildId)
                az webapp restart --name suroga-booking-api --resource-group
MyRG
```

Secrets and Environment Variables

- Store sensitive values in Azure DevOps secure variables or Secret Store.
- Orchestrator injects these into containers at runtime.

Deployment Targets

- **Azure App Service for Containers:** Suitable for standalone container apps; configure image, variables, instances, autoscaling.
- **Azure Kubernetes Service (AKS):** Ideal for multi-service systems; define `Deployments`, `Services`, `Ingress`, `Secrets`, etc.
- **Azure Container Instances (ACI) or Docker Compose on VM:** Lightweight scenarios.
- Ensure health probes point to `/health/` for automatic restarts.

- Enable autoscaling based on CPU, memory, or request load.

4.3 Future Integration with Suroga API

- The Booking API can run as a standalone container or as part of a multi-container setup (e.g., Docker Compose or sidecar model).
- Use a common orchestrator (Kubernetes, Docker Compose, Azure Container Apps) to coordinate:
    - Booking API, Frontend, Auth API, reverse proxies (e.g., NGINX), Janus proxy, auxiliary services (Redis, Celery broker).
- Document internal networking:
    - Booking API must connect to Janus, the database, and (eventually) authentication services using cluster DNS.

4.4 Versioning and API Paths

- Adopt Semantic Versioning (`v1.2.3`) for Docker image tags and Git releases.
- Include versioning in API routes (e.g., `/api/v1/bookings/`) to allow backward-compatible evolution.

6. API Documentation (OpenAPI / Swagger)

The Booking API provides a full OpenAPI 3.0.3 specification to enable easy integration, self-service discovery, and client generation. This section outlines its structure and usage patterns.

Basic Information

```
openapi: 3.0.3
info:
  title: Suroga Booking API
  version: "1.0.0"
  description: API for configuring and scheduling WebRTC rooms via the Janus
gateway.
servers:
  - url: https://api.suroga.com/api/v1
components:
  securitySchemes:
    BearerAuth:
      type: http
      scheme: bearer
      bearerFormat: JWT
security:
  - BearerAuth: []
```

API Endpoints

`/rooms/`

- `GET`: List all rooms. Supports optional filters:

    - `available` (boolean): only available rooms.
    - `start_time`, `end_time` (date-time): required if filtering by availability.
- `POST`: Create a new room.

`/rooms/{room_id}/`

- `GET`: Retrieve room details by ID.
- `DELETE`: Delete a room (admin only). Will fail if future bookings exist or permission is lacking.

`/rooms/{room_id}/participants/`

- `GET`: Retrieve real-time participant list and count from Janus.

`/rooms/{room_id}/password/reset/`

- `POST`: Admin-only operation to reset a room's password.

`/rooms/availability/`

- `GET`: Check room availability for a given time window.
    - Requires `start_time` and `end_time` query parameters.

`/bookings/`

- `GET`: Retrieve all bookings for the authenticated user.
- `POST`: Create a new booking. Supports idempotent behavior (find-or-create).

`/bookings/{booking_id}/`

- `GET`: Retrieve detailed information for a specific booking.
- `DELETE`: Cancel a booking. May be restricted based on time or permission.

Schema Definitions (`components/schemas`)

- `Room`:

```
room_id: integer
max_participants: integer
created_at: string (date-time)
updated_at: string (date-time)
```

- `RoomCreate`:

```
max_participants: integer (minimum 1)
optional_name: string (optional)
metadata: object (optional)
```

- `RoomDetail`: Same as `Room` + `has_future_booking: boolean`
- `RoomParticipants`: `room_id`, `participant_count`, optional `participants` array

- `RoomAvailability`: subset of Room with minimal metadata for available rooms
- `BookingCreate`: `start_time`, `end_time`, optional `room_id`
- `BookingResponse`: `booking_id`, `room_id`, `password`, `start_time`, `end_time`
- `BookingDetail`: includes status and optional password

Automated Documentation Generation

To ensure synchronization between API implementation and its documentation, tools such as `drf-spectacular` or `drf-yasg` should be used. These libraries generate Swagger/OpenAPI documentation directly from Django REST Framework views and serializers.

---

7. Final Notes

- **Keep Documentation Updated**: Always update the OpenAPI spec and Swagger examples when changing endpoints or payloads.
- **Developer Onboarding**: Include local setup instructions via Docker Compose. Example:

```
version: '3.8'
services:
  db:
    image: postgres:13
    environment:
      POSTGRES_USER: suroga
      POSTGRES_PASSWORD: <password>
      POSTGRES_DB: suroga_db
    volumes:
      - pgdata:/var/lib/postgresql/data
  booking-api:
    build:
      context: .
      dockerfile: Dockerfile
    environment:
      DATABASE_URL: postgres://suroga:<password>@db:5432/suroga_db
      JANUS_URL: http://janus:8088/janus
      JANUS_KEY: <janus_api_key>
      DJANGO_SETTINGS_MODULE: myproject.settings.dev
    ports:
      - "8000:8000"
    depends_on:
      - db
  redis:
    image: redis:6
volumes:
  pgdata:
```

- **Future Integration**:
    - When merging the Booking API into the Suroga monolith, consolidate Docker layers and unify shared environment variables, routes, and authentication.

    - Evaluate using a **multi-stage pipeline** if Suroga evolves into a multi-service platform (e.g., auth, analytics, bookings), with separate Docker images and deployments per module.
- **Security in Production**:
    - Secrets like `SECRET_KEY`, `JANUS_KEY`, and DB credentials should be managed via Azure Key Vault (or equivalent) and injected securely at runtime.
- **Monitoring and Alerts**:
    - Use Azure Application Insights (or equivalent) to track endpoint usage, latency, error rates, and container health.
    - Set up alerting for anomalies (e.g., spike in 5xx errors or failed health checks).

6. API Documentation (OpenAPI / Swagger)

The Booking API is fully documented using the OpenAPI 3.0.3 specification. This ensures clarity, consistency, and ease of integration for clients and services consuming the API. Below is a detailed breakdown of the structure, endpoints, authentication mechanisms, and schemas involved.

Specification Metadata

```
openapi: 3.0.3
info:
  title: Suroga Booking API
  version: "1.0.0"
  description: API for configuration and booking of WebRTC rooms via the Janus
Gateway.
servers:
  - url: https://api.suroga.com/api/v1
components:
  securitySchemes:
    BearerAuth:
      type: http
      scheme: bearer
      bearerFormat: JWT
security:
  - BearerAuth: []
```

Key Endpoints

`/rooms/`

- **GET**: Retrieve a list of rooms.
    - Optional query parameters:
        - `available` (boolean): Filter to only available rooms (requires `start_time` and `end_time`).
        - `start_time`, `end_time` (ISO 8601 datetime): Time range for availability.
- **POST**: Create a new room.

        - Request body: `RoomCreate` schema.
        - Response: `RoomDetail` schema.

`/rooms/{room_id}/`

- **GET**: Get detailed information about a room.
- **DELETE**: Delete a room (admin-only).
    - May return `403` if the room has future bookings or user lacks permissions.

`/rooms/{room_id}/participants/`

- **GET**: Get real-time participant count and list for a room.
    - Returns `502` if Janus is unreachable.

`/rooms/{room_id}/password/reset/`

- **POST**: Reset the room password (admin-only).
    - Returns the `new_password` in response.
    - Returns `503` if Janus fails.

`/rooms/availability/`

- **GET**: Check availability of rooms within a time range.
    - Required query parameters: `start_time`, `end_time`.
    - Response includes `available_rooms` array.

`/bookings/`

- **GET**: List bookings for the authenticated user.
- **POST**: Create a new booking (find-or-create behavior).
    - Request body: `BookingCreate` schema.
    - Returns `BookingResponse` or errors (`409`, `503`).

`/bookings/{booking_id}/`

- **GET**: Retrieve booking details.
- **DELETE**: Cancel an existing booking.
    - Returns `403` if cancellation isn't allowed.

Schema Definitions

```
components:
  schemas:
    Room:
      type: object
      properties:
        room_id:
```

```
        type: integer
      max_participants:
        type: integer
      created_at:
        type: string
        format: date-time
      updated_at:
        type: string
        format: date-time

  RoomCreate:
    type: object
    properties:
      max_participants:
        type: integer
        minimum: 1
      optional_name:
        type: string
      metadata:
        type: object

  RoomDetail:
    allOf:
      - $ref: '#/components/schemas/Room'
      - type: object
        properties:
          has_future_booking:
            type: boolean

  RoomParticipants:
    type: object
    properties:
      room_id:
        type: integer
      participant_count:
        type: integer
      participants:
        type: array
        items:
          type: object

  RoomAvailability:
    type: object
    properties:
      room_id:
        type: integer
      max_participants:
        type: integer

  BookingCreate:
    type: object
    properties:
      start_time:
        type: string
        format: date-time
      end_time:
        type: string
        format: date-time
      room_id:
        type: integer

  BookingResponse:
    type: object
    properties:
      booking_id:
        type: integer
      room_id:
        type: integer
      password:
        type: string
      start_time:
        type: string
      end_time:
        type: string

  BookingDetail:
    type: object
    properties:
      booking_id:
        type: integer
      room_id:
        type: integer
      start_time:
        type: string
      end_time:
        type: string
      status:
        type: string
      password:
        type: string
```

Automatic Documentation

We recommend using tools such as **drf-spectacular** or **drf-yasg** to auto-generate OpenAPI specs from Django REST Framework views and serializers. This ensures the documentation remains in sync with the codebase.

These tools can also generate Swagger UI or ReDoc interfaces for easier API exploration and onboarding.

6. Front-End of the Service

This section outlines how the client (web app or mobile WebView) consumes the WebRTC service, interfaces with the Janus Gateway, and provides a user interface for initiating or participating in 1-on-1 streaming sessions.

6.1 Goals and Scope

- **Objective**: Provide clear guidance to front-end developers on how to integrate WebRTC streaming using Janus, ensuring usability, performance, and security.
- **Target Audience**: Front-end teams (web, mobile via WebView or hybrid), UI/UX teams, and support staff.
- **Scope**:
    - **Core flows**: Initiating and joining 1-on-1 streams (host starts, guest joins), audio/video sharing, optional screen sharing.

    - **UI controls**: Start/stop buttons for camera, microphone, and screen sharing; connection indicators; participant status.
    - **Error and state handling**: Display clear messages for media or connection failures.
    - **Browser compatibility**: Document minimum supported versions (Chrome, Firefox, Safari) and provide fallbacks or alerts.
    - **Responsiveness**: Ensure UI adapts to desktop and mobile (portrait and landscape).
    - **Auth integration**: Use access tokens for secure stream access.
    - **Quality of Experience**: Handle latency, network indicators, and automatic bitrate/resolution adjustments.

6.2 Technologies and Dependencies

- **Janus JS Client Library**:
    - Include the exact version used (via CDN or bundler), with commit/tag for traceability.
    - Initialization: `Janus.init({debug: [...]})`, create session, attach the "videoroom" plugin.
- **Adapter.js**:
    - Include to ensure cross-browser WebRTC compatibility.
- **UI Frameworks**:
    - For React/Vue/Angular, encapsulate Janus logic in hooks or services. For example, in React, use `useJanusSession()` to manage connection, state events, and expose methods (`joinRoom`, `leaveRoom`, `publishStream`, `subscribeStream`).
    - Component-based UI: reusable local preview, remote video, and control buttons.
    - Styling: Responsive layout with CSS flex/grid, status indicators (icons), alert messages (modals/banners).
- **Backend Communication**:
    - Call backend API to retrieve room credentials (e.g., `room_id`, password, token) before initializing Janus.
    - Configurable endpoints: `JANUS_SERVER_URL`, `API_BASE_URL`, authentication keys.
- **Signaling Mechanisms**:
    - Janus supports WebSocket or HTTP signaling. Front-end constructs WebSocket URL (e.g., `wss://janus.example.com/janus`), including optional params (token, room ID).
    - Do not expose `apisecret` on the front-end; use the backend to generate temporary credentials or JWTs validated by Janus.

- **Media Capture**:
  - Access media using `navigator.mediaDevices.getUserMedia({ audio: true, video: true })`.
  - For screensharing: use `navigator.mediaDevices.getDisplayMedia(...)` and document fallbacks for unsupported browsers.
- **Multiple Stream Management**:
  - For 1-on-1, manage one publishing stream and one subscribing stream. Prepare for future multi-party support with a flexible stream array.
- **Adaptive Quality**:
  - Enable dynamic constraints (e.g., resolution, frame rate) based on bandwidth.
  - Use WebRTC `getStats` to monitor metrics (jitter, packet loss) and adjust stream settings accordingly.
- **Front-end Metrics and Logging**:
  - Track events: connection time, media capture failures, ICE errors, disconnections.
  - Send logs to a backend observability service for field diagnostics.
- **Internationalization (i18n)**:
  - Externalize button texts and error messages to support multiple languages.
- **Accessibility (a11y)**:
  - Ensure controls are properly labeled, provide alt text/icons, and support screen readers and keyboard navigation.

6.3 Typical Usage Flow

1. **Start Conference Page/Component**:
   - Authenticated user clicks "Start Call".
   - Call API (POST `/bookings/`) to create a room and receive credentials.
   - Initialize Janus:

     ```
     Janus.init({ debug: [...] });
     const janus = new Janus({
       server: JANUS_SERVER_URL,
       success: () => {},
       error: (err) => {},
       destroyed: () => {}
     });
     ```

   - Attach videoroom plugin:

     ```
     janus.attach({
       plugin: "janus.plugin.videoroom",
     ```

```
  success: (handle) => {},
  error: (err) => {},
  consentDialog: (on) => {},
  webrtcState: (on) => {},
  mediaState: (medium, on) => {},
  onmessage: (msg, jsep) => {},
  onlocalstream: (stream) => {},
  onremotestream: (stream) => {},
  oncleanup: () => {}
});
```

- Join as publisher:

  ```
  pluginHandle.send({
    message: {
      request: "join",
      room: room_id,
      ptype: "publisher",
      display: username,
      secret: password
    }
  });
  ```

- Create offer and publish:

  ```
  pluginHandle.createOffer({
    success: (jsep) => {
      pluginHandle.send({
        message: { request: "configure", audio: true, video: true },
        jsep: jsep
      });
    },
    error: (err) => {}
  });
  ```

2. **Join as Subscriber**:
   - User receives invite (room ID + password).
   - Initialize Janus and attach plugin.
   - Join as subscriber:

     ```
     pluginHandle.send({
       message: {
         request: "join",
         room: room_id,
         ptype: "subscriber",
         feed: publisherId
       }
     });
     ```

   - Create answer and display remote stream.
3. **Real-time Controls**:
   - Toggle camera/mic/screenshare using plugin `configure` messages.
   - Handle disconnections with `webrtcState` events.
4. **End Session**:

    - Leave room: `pluginHandle.send({ message: { request: "leave" } })`
    - Destroy session: `janus.destroy()`, stop tracks.
    - Notify backend if necessary.
5. **Screensharing**:
    - On click, call `getDisplayMedia`, then renegotiate offer with new video track.
6. **Error Handling and Fallbacks**:
    - Handle `getUserMedia` denial with user-friendly modals.
    - Display messages for ICE failure, timeouts, or attach/join errors.
7. **UI/UX Enhancements**:
    - Visual indicators for mic/cam status, connection state.
    - Invite link with room ID and token; copy-to-clipboard button.
    - Responsive layout, front/rear camera handling on mobile.
    - Use native APIs for notifications if applicable.

6.4 Front-End Security

- **HTTPS Required**: WebRTC must run in secure contexts.
- **Authentication/Tokens**:
    - Never expose `JANUS_APISECRET` in front-end.
    - Use JWTs or temporary tokens generated by backend.
    - Document token validation by Janus (JWT or plugin).
- **CORS**: Configure Janus and API to allow client origin.
- **Abuse Protection**: Only allow room access via backend-issued `room_id`.
- **Media Protection**: Use WebRTC DTLS-SRTP. Consider end-to-end encryption if needed.
- **Security Headers**: Configure `Content-Security-Policy`, `X-Frame-Options`, etc.

6.5 Testing and Quality Assurance

- **Manual & Automated Testing**:
    - Test across browsers and devices.
    - Simulate poor network to test adaptive streaming.
    - Validate permission-denied cases for media and screenshare.
    - Reconnect on network drop.

- **Error Monitoring**:
    - Capture JS errors via tools like Sentry.
    - Log SDK load and WebRTC connection times.
- **Documentation for Developers**:
    - Provide README/wiki with setup instructions, Janus integration, and examples.
    - Sample components (e.g., `<VideoChat roomId=... password=... onEnd={...} />`).
    - Style and code guidelines.

6.6 Client Observability

- **Event Logs**:
    - Log key events: `requestCreateSession`, `attachedPlugin`, `joinedRoom`, `onlocalstream`, etc.
    - Send to backend logging or analytics system.
- **Usage Metrics**:
    - Count sessions started, average duration, connection failures, screen share usage.
- **User Feedback**:
    - Show quality indicators (e.g., "Good Connection", "High Latency").

7. Integration

This section documents how the WebRTC streaming service integrates with the broader Suroga ecosystem, including the booking/backend API, authentication, container orchestration, monitoring, mobile applications, CI/CD, and deployment processes.

7.1 Integration with Booking API and Backend

- **Booking to Streaming Flow:**
    1. The user initiates a call via the app or web interface: the front-end calls the Booking API (`/bookings/`) to obtain a `room_id` and password.
    2. The backend registers the booking, optionally schedules the time, and returns the credentials.
    3. The front-end uses the received credentials to initiate a Janus session and join the room.
- **Permission Validation:**

    1. The backend verifies whether the user has the right to join the room (i.e., if the user booked it or was invited).
    2. Upon joining, the front-end sends an auth token and `room_id/password`; the backend may act as a proxy for Janus to validate the session before returning session parameters.
- **State Webhooks/Notifications:**
    1. When a session starts or ends, either the front-end or Janus can notify the backend via webhook or API, e.g., "call started at X", "call ended at Y". Useful for logging, billing, and audit purposes.
- **JWT Token for Janus:**
    1. Instead of exposing the `JANUS_KEY`, the backend generates a JWT for each session that Janus validates (using its JWT plugin):
        - The JWT contains the room, user, expiration, and is signed with a shared secret.
        - The front-end receives the JWT and passes it to Janus during session creation or room join.
        - Janus validates the token, allowing the user to join without exposing the API secret.
- **Active Session Management:**
    1. The backend maintains records of active sessions (Janus `sessionId` and associated user) to enable administrative actions (force disconnect, monitor usage).
- **Fallback and Error Handling:**
    1. If the backend or Janus is unavailable, the front-end should show a user-friendly error message and the backend should trigger alerts.
- **Analytics API:**
    1. At the end of a call, the front-end or Janus sends quality metrics (jitter, packet loss, bitrate) via the backend for user experience analysis.

7.2 Integration with Mobile / Native Apps

- **WebView or Native SDK:**
    - For hybrid apps, use a WebView to load the web front-end; ensure camera/microphone permissions are set in the manifest (Android) or Info.plist (iOS).
    - For native apps (React Native, Flutter, Swift/Java), use native WebRTC libraries or Janus Native SDK:
        - For example, in React Native, use `react-native-webrtc` and implement Janus signaling via WebSocket/HTTP.
        - Document native integration examples: PeerConnection creation, SDP exchange via backend signaling or directly with Janus.

- **Runtime Permissions:**
    - Document how to request camera/mic permissions on Android/iOS, and how to handle denial cases.
- **Mobile Screen Sharing:**
    - Screen sharing is more complex on mobile: iOS and Android require specific APIs (ReplayKit, MediaProjection). Document the integration approach if needed, likely using native code to send a track to Janus.
- **Push Notifications:**
    - For scheduled calls, the backend can send a push notification to the invited user. Document the flow: backend schedules, mobile app receives, opens call screen, starts front-end streaming.
- **Quality & Adaptability:**
    - Handle unstable mobile networks with reconnection strategies and automatic bitrate/resolution adjustments.
- **Real Device Testing:**
    - Provide instructions for testing across device models and OS versions to ensure WebRTC compatibility.

7.3 Authentication and Authorization

- **Single Sign-On (SSO):**
    - If Suroga uses centralized authentication, the front-end logs in and the backend issues tokens for the booking API and Janus sessions (via JWT).
- **Short-lived Tokens:**
    - Use short-expiration tokens for Janus, scoped to specific rooms and users, minimizing misuse risks.
- **Access Control:**
    - The backend checks user permissions before providing room credentials. The front-end must deny access if the token is invalid.
- **Session Revocation:**
    - If a user is removed or a call is canceled, the backend can notify the front-end (via control WebSocket) or instruct Janus to terminate the session.
- **Endpoint Protection:**
    - Document required headers, token scopes, and how the front-end should handle 401/403 responses.

7.4 CI/CD and Deployment

- **Front-end Build Pipeline:**

    - If the UI is a SPA (React/Vue), implement a pipeline that:
        - Runs linting/tests,
        - Produces an optimized bundle (minification, tree-shaking),
        - Publishes to a CDN or web container (e.g., nginx).
    - Integrate with the overall Suroga CI/CD, possibly multi-stage: build front-end, booking API backend, proxy/Janus containers.
- **Janus Gateway Pipeline:**
    - Although Janus is not a front-end component, its configuration (jcfg) and deployment scripts can be versioned and automated: a pipeline applies them to VM/Kubernetes instances.
- **Container Orchestration:**
    - Define Kubernetes manifests (Deployments, Services, Ingress) for:
        - Front-end (SPA or static files),
        - Booking backend,
        - Reverse proxy (NGINX) for Janus (WebSocket/HTTPS),
        - Janus Gateway instances (Deployment/StatefulSet),
        - TURN server (if needed),
        - Monitoring (Prometheus exporters, logging),
        - Redis/Celery for async tasks,
        - Database.
    - Document how to version and apply changes using GitOps tools (FluxCD/ArgoCD) or Azure DevOps pipelines.
- **Environment Configuration:**
    - Use environment variables and secrets (Azure Key Vault or Kubernetes Secrets) for URLs, keys, tokens, DB credentials.
- **Health Checks and Probes:**
    - Front-end: probe `/health/`.
    - Janus: check responsiveness via HTTP API or WebSocket pings.
- **Monitoring and Alerts:**
    - Dashboards (Grafana) showing pod/VM CPU/memory, WebRTC latency, active session count, connection errors.
    - Alerts for instance crashes, high latency, backend 5xx errors, Janus failures.
- **Rollback and Versioning:**

    - Use versioned container images and deploy strategies that allow rollback if regressions are found.
- **End-to-End Integration Testing:**
    - Automate tests that start real WebRTC sessions with a test Janus instance to validate streaming functionality after each deploy.
- **Upgrade Procedures:**
    - Document how to update Janus (compilation), rotate instances with no downtime, and maintain a failover-ready pool.
    - Describe how to upgrade the streaming front-end without affecting active users: use feature flags or blue/green deployments.

7.5 Monitoring and Observability

- **Aggregated Logs:**
    - Front-end: send JS error logs to systems like Sentry.
    - Backend: log Booking API calls and Janus integration events.
    - Janus: collect gateway logs, ICE errors, plugin messages; send to centralized logging (ELK, Azure Monitor).
- **WebRTC Metrics:**
    - Collect stats via Janus API/plugins: bitrate, jitter, packet loss, server CPU.
    - Expose dashboards for operational and product teams to monitor QoS.
- **Alerts:**
    - Set thresholds for key metrics (session count, CPU usage, repeated errors).
    - Alert SRE teams on QoS degradation (e.g., high packet loss, connection failures).
- **Periodic Reports:**
    - Weekly/monthly usage reports: session counts, average duration, usage peaks, top regions.
    - Useful for capacity and cost planning.

7.6 Documentation and Onboarding

- **Developer Docs:**
    - Wiki or repo with full instructions: local setup (Docker Compose), env variables, running tests, using Janus locally.
    - Front-end examples: video component, Janus hooks/services, backend API calls.
    - Coding standards and folder structures.
- **Operator Docs:**

    - Provisioning infra: VMs/Kubernetes for Janus/TURN, firewall rules, Azure NSGs, load balancer setup.
    - SSL/TLS configuration for secure WebSocket/HTTPS.
    - How to back up configs, update Janus, rotate certificates.
- **QA Documentation:**
    - Manual and automated test cases: happy path and edge cases (unstable network, permission denial).
    - Instructions for WebRTC debugging tools (e.g., `chrome://webrtc-internals`).
- **Troubleshooting Guides:**
    - Common scenarios (ICE failure, codec mismatch, blocked UDP/firewall) and how to debug (Janus logs, browser logs).
    - CLI tools/scripts to verify UDP ports, NAT, TURN setup.

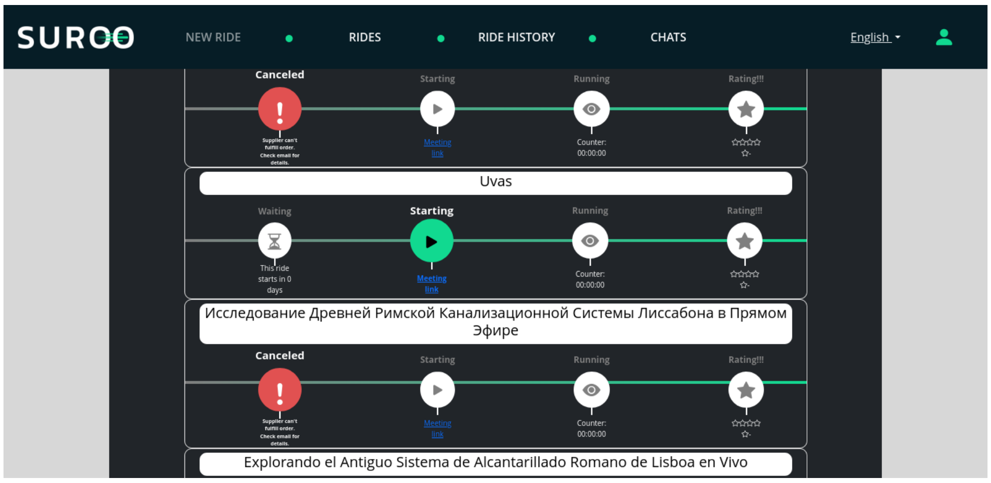
7.7 Security and Compliance

- **Network Security:**
    - Ensure NSG/Azure Firewall allows only required ports (TCP 80/443, UDP 10000-20000 for RTP).
    - Enable DDoS protection if facing public traffic.
- **Data Protection:**
    - Avoid storing recordings without user consent or regulatory compliance.
    - If recordings are allowed, encrypt at rest and in transit.
- **Privacy:**
    - Inform users about media capture and request camera/mic access consent.
- **Auditing:**
    - Log who created/joined calls, timestamps, and IPs (GDPR compliant).
- **Library Updates:**
    - Monitor vulnerabilities in front-end (Janus JS, adapter.js) and backend dependencies.
- **CORS and CSP:**
    - Use strict Content-Security-Policy, allowing only trusted origins for scripts and frames.
- **Penetration Testing:**
    - Test WebRTC flows and endpoints against common attacks (DoS, injection, CSRF, XSS).

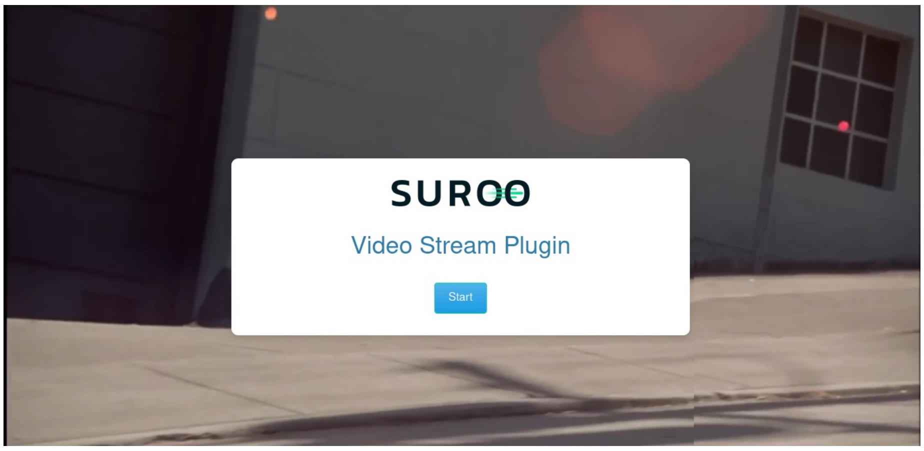
7.8 Future Enhancements

- **Multi-party & Scalability:**
  - Though currently 1:1, consider supporting multi-user rooms using Janus SFU. Document how to scale.
- **Transcoding / Broadcast:**
  - Plan for 1-to-N streaming (e.g., live broadcasts), using Janus plugins or CDN integration.
- **Streaming Platform Integration:**
  - Suroga is streaming-agnostic: document how to integrate RTMP or HLS output from Janus.
- **PWA and Native Evolution:**
  - Consider evolving front-end into a PWA: support notifications, background sync, offline features (e.g., message buffering).
- **Advanced Analytics:**
  - Use ML to detect quality issues and auto-suggest optimizations.
- **Admin APIs:**
  - Build admin dashboards to monitor active sessions, force disconnects, broadcast messages, etc.

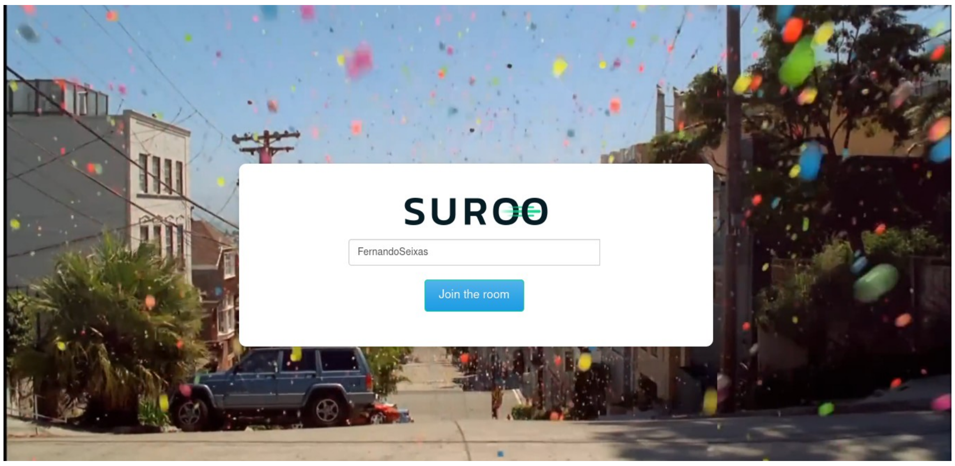
A. Integração no front-end do Suroo


SUROO
NEW RIDE
RIDES
RIDE HISTORY
CHATS
English
Canceled
Starting
Running
Rating!!!
Supplier can't fulfill order. Check email for details.
Meeting link
Counter: 00:00:00
Uvas
Waiting
Starting
Running
Rating!!!
This ride starts in 0 days
Meeting link
Counter: 00:00:00
Исследование Древней Римской Канализационной Системы Лиссабона в Прямом Эфире
Canceled
Starting
Running
Rating!!!
Supplier can't fulfill order. Check email for details.
Meeting link
Counter: 00:00:00
Explorando el Antiguo Sistema de Alcantarillado Romano de Lisboa en Vivo

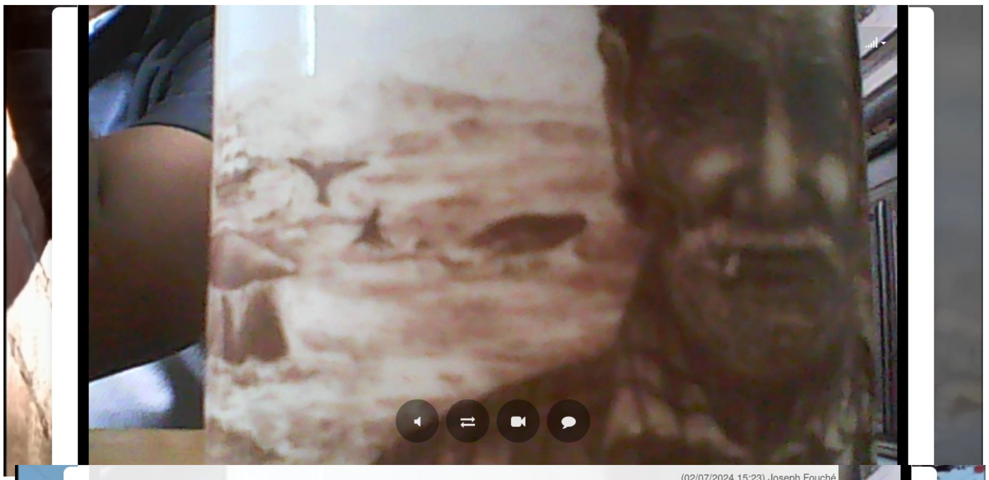
B. Authentication



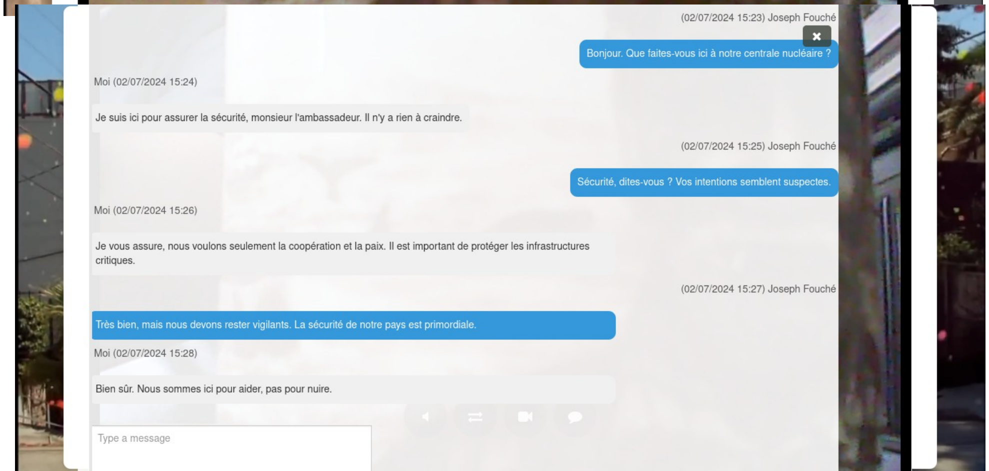
C. Set client visible name



D. Client Board with options: Sound, Video, Chat



C. Ride message feed:



(02/07/2024 15:23) Joseph Fouché

Bonjour. Que faites-vous ici à notre centrale nucléaire ?

Moi (02/07/2024 15:24)

Je suis ici pour assurer la sécurité, monsieur l'ambassadeur. Il n'y a rien à craindre.

(02/07/2024 15:25) Joseph Fouché

Sécurité, dites-vous ? Vos intentions semblent suspectes.

Moi (02/07/2024 15:26)

Je vous assure, nous voulons seulement la coopération et la paix. Il est important de protéger les infrastructures critiques.

(02/07/2024 15:27) Joseph Fouché

Très bien, mais nous devons rester vigilants. La sécurité de notre pays est primordiale.

Moi (02/07/2024 15:28)

Bien sûr. Nous sommes ici pour aider, pas pour nuire.

Type a message