

SUROGA WebRTC Streaming Service

Este documento tem por objectivo descrever o serviço de web streaming do suroga. Estratégia de configuração do servidor, instalação do Janus Server, descrição da API para configuração de canais de streaming e reserva de canais, descrição da corrente versão do front-end e integração do front-end em APP.

0. Introdução

O serviço de streaming oferecido pela SUROGA é disponibilizado através da sua robusta API. Inicialmente testado através da aplicação SUROGA, o serviço está agora sendo expandido para ser agnóstico em relação aos provedores de streaming, permitindo integração fluida com plataformas como Greenways e outras soluções de streaming.

1. Seleção e Configuração de Servidor

Para um serviço WebRTC restrito à UE (1-para-1, poucos streams paralelos) com forte ênfase em contenção de custos, a escolha da VM e sua configuração de rede são cruciais.

1.1 Escolha da VM na Azure

- Série Dsv5 (ex.: Standard_D2s_v5, D4s_v5)
 - **Desempenho de CPU recente:** CPUs Intel Xeon de 3ª geração (Ice Lake) oferecem desempenho single-thread superior às gerações anteriores, importante para operações de sinalização e possíveis codecs em software (cdrdv2-public.intel.com, cloudprice.net).
 - **Rede de alta largura de banda:** As instâncias Dsv5 suportam *Accelerated Networking*, oferecendo baixa latência, alto throughput e menor jitter via SR-IOV, beneficiando diretamente WebRTC (learn.microsoft.com, learn.microsoft.com).
 - **Localização:**
 - Recomenda-se implantar em regiões próximas geograficamente aos utilizadores/Amsterdam, como **West Europe (Países Baixos)** ou **North Europe (Irlanda)**, para reduzir latência de ida e volta. Estudos de latência inter-regiões mostram que essas regiões tendem a manter p50 abaixo de ~50 ms dentro da UE (learn.microsoft.com).
 - Pode-se também avaliar Frankfurt (Germany West Central) ou outras zonas centrais, mas West/North Europe costumam ter bom equilíbrio entre custo e latência.
 - **Uso de disco/armazenamento:** Em streaming 1-1, geralmente não há necessidade de armazenamento persistente intenso; discos temporários (SSD local) podem ser suficientes para logs temporários ou buffering breve.

- **Série B (ex.: B2s, B4ms)**
 - **Burstable, baixo custo:** Ideal para workloads intermitentes com picos curtos de CPU (ex.: instâncias de Janus que só precisam processar picos de sinalização ou plugin de gravação eventualmente). Quando o uso de CPU fica abaixo do baseline, acumulam créditos, permitindo bursts quando necessário (learn.microsoft.com).
 - **Adequado para 1-para-1 leve:** Se cada instância de Janus atender a um único stream por vez e espera-se longos períodos inativos, B2s pode reduzir custos significamente.
 - **Considerações de baseline:** Deve-se estimar o consumo médio de CPU; se frequentemente excede o baseline (p. ex., se há codificação/transcoding intensivo), créditos podem se esgotar e a VM ser limitada. Portanto, iniciar testes em D2s_v5 e, se constatar que uso de CPU é baixo, migrar para B2s.
- **Instâncias Spot (opcional)**
 - **Descontos até ~90%** em cargas tolerantes a interrupções, ideal se o agendamento é bem previsível e existam janelas de tolerância à pré-emissão da VM (azure.microsoft.com).
 - **Risco de eviction:** Se o stream for crítico (mesmo 1-1), cuidado: uma interrupção forçada durante sessão ativa causaria falha no usuário. Poderia ser usado para testes ou duplicar instâncias com failover, mas para 1-1 real-time não é recomendado sem estratégia de fallback robusta.
- **Rede otimizada**
 - **Accelerated Networking:** Deve ser habilitado em VMs que suportam (Dsv5 e algumas B-series suportam), reduzindo latência/jitter e uso de CPU para pacotes de rede (learn.microsoft.com).
 - **Configurações adicionais:**
 - Use Proximity Placement Groups se tiver múltiplas VMs no mesmo serviço de baixa latência (útil em configurações de alta disponibilidade), embora em 1-1 poucas instâncias paralelas, pode não ser necessário.
 - Configure NSGs (Network Security Groups) restritivos permitindo apenas portas necessárias (ver seção de firewall).

Recomendação inicial

- **Teste preliminar:**
 1. Criar uma VM **Standard_D2s_v5** com **Accelerated Networking** em West Europe.
 2. Validar consumo de CPU e comportamento de rede durante sessões WebRTC de teste (simular clientes na UE).
 3. Se uso de CPU médio for baixo (< baseline do B2s) com picos esporádicos, testar migrar para **B2s** para economia adicional.
- **Monitoramento:**

1. Use métricas de CPU/network da Azure para acompanhar créditos de CPU no B-series e latência/jitter de rede. Ajuste a escolha de VM conforme resultados.
- **Custo estimado:** Varia por região e reserva (compromissos de uso, instâncias reservadas, spot). Regiões UE tendem a ser ~10-20% mais baratas que outras áreas globais (cloudpice.net).

1.2 Automação via API

Para minimizar custos mantendo VMs ligadas apenas quando necessárias:

- **Provisionamento agendado baseado em solicitações (24 h de antecedência)**
 - Ao receber reserva de streaming com ao menos 24 h de antecedência, planeja-se ligar a VM ~30 min antes do início da transmissão, garantindo que Janus e dependências (TURN, etc.) estejam prontas.
 - Após término previsto, desligar/dealocar a VM automaticamente após janela de encerramento (considerar margem extra para atrasos eventuais).
- **Ferramentas Azure CLI/SDK**
 - **Azure CLI (az vm start, az vm deallocate):**
Exemplo de comando para iniciar:

```
az vm start --resource-group MeuRG --name JanusVM
```


Para desligar (deallocate):

```
az vm deallocate --resource-group MeuRG --name JanusVM
```


(learn.microsoft.com).
 - Pode-se orquestrar via scripts em CI/CD, cron em servidor de gestão ou Azure Automation Runbooks (Start/Stop VMs during off-hours) com parâmetros dinâmicos extraídos do sistema de reservas (learn.microsoft.com).
 - **Azure SDKs (Python, .NET, etc.):** Permitem integrar diretamente nas aplicações de backend para iniciar/desligar VMs conforme a lógica de negócio.
- **Agendamento centralizado**
 - **Módulo de agenda:** Banco de dados ou serviço de calendário que armazena reservas de streaming.
 - **JobScheduler:** Tarefa periódica (ex.: cron ou Azure Function com Timer Trigger) consulta reservas futuras e emite comandos de start/stop via CLI/SDK no horário certo.
 - **Logs e alertas:** Registrar sucesso/falha de start/stop; enviar alerta (e.g., via email ou Slack) se uma VM não iniciar em tempo útil ou não desligar corretamente.
- **Fallback manual / overrides**
 - Interface administrativa para iniciar/estender manualmente VMs em caso de imprevistos (e.g., cliente pede prorrogação de sessão).

- Autenticação forte (e.g., MFA) e auditoria (logs) para alterações manuais de estado de VM.
- **Considerações de latência de start-up**
 - VMs “cold start” podem levar alguns minutos para ficarem prontas. Planeje start ~30 min antes do evento. Se for essencial reduzir esse tempo, considere VMs pré-aquecidas em janelas de alta probabilidade de uso.
- **Instâncias Spot**
 - Se adotar Spot para testes ou cargas não críticas, scripts devem lidar com possível eviction (e.g., fallback para VM padrão ou reatribuir sessão a outra instância).

1.3 Planeamento Antecipado

- **Agenda centralizada e previsões de carga**
 - **Gerar plano semanal/mensal:** Listagem de todas as reservas, com horários de início/fim, número de canais (neste caso 1-1, mas ainda há múltiplas reservas sequenciais).
 - **Dimensionar quantidade de VMs necessárias:** No modelo 1-1, normalmente 1 VM por sessão. Se houver pequenas sobreposições, garanta capilaridade suficiente (ex.: 2-3 VMs prontas em janelas de pico).
 - **Alocação inteligente:**
 - Se múltiplas sessões previstas próximas, avaliar manter VMs ativas em pool e reutilizá-las (reduzir tempo de start/stop frequente).
 - Se uso esporádico isolado, desligar imediatamente após cada sessão.
 - **Integração com billing/cobrança**
 - Relacionar tempo ativo das VMs com custo do cliente, para cobrança alinhada ao consumo de infraestrutura.
 - **Monitoramento e alertas antecipados**
 - Detectar alterações de reserva (cancelamentos, adições) e reprogramar start/stop automaticamente.
 - Enviar lembretes às equipes responsáveis (ops) sobre programação semanal, caso haja necessidade de verificações manuais.
 - **Testes periódicos**
 - Em janelas de baixa ocupação, disparar VMs de teste para validar pipelines de start/stop, atualizações de imagem e configuração de Janus (garantir que no dia real tudo funcione).
-

2. Configuração da Firewall do Servidor Janus

Para que Janus opere corretamente como servidor WebRTC, é necessário abrir/encaminhar portas específicas e tratar NAT de forma adequada.

2.1 Portas necessárias

- **HTTP/HTTPS e WebSocket**
 - **HTTP** (se usar web UI ou endpoints REST de Janus): TCP 8088 (padrão Janus) ou expor via proxy Nginx em TCP 80 → localhost:8088 (voipnuggets.com).
 - **HTTPS/WSS:**
 - Janus suporta HTTPS em TCP 8089 ou, preferencialmente, usar Nginx/Traefik/Apache como proxy reverso em TCP 443 para TLS (certificado Let's Encrypt) e encaminhar para Janus em 8088/8089/8188/8989 conforme configuração (voipnuggets.com, facsiaginsa.com).
 - **WSS (WebSocket seguro):** TCP 8989 → configurado em `janus.transport.websockets.jcfg` (se `ws_ssl=true`) (voipnuggets.com).
- **Portas RTP/RTCP (UDP)**
 - Range configurável em `rtp_port_range` no `janus.jcfg`, tipicamente **10000–20000** ou **20000–20500**. A firewall/NSG deve permitir UDP inbound/outbound neste intervalo, encaminhando para o IP interno do Janus (facsiaginsa.com).
 - Se houver plugin SIP, pode haver outro range (definido especificamente), mas aqui focamos em WebRTC PeerConnections.
- **TURN/STUN (se usar servidor próprio)**
 - **TURN:** normalmente UDP 3478/5349; se estiver atrás do proxy, encapsular via TCP/TLS em 443 pode ser necessário para redes restritivas.
 - **STUN:** UDP 3478; se usar serviço externo, apenas saída UDP.
- **Porta de administração (opcional)**
 - Se habilitar API administrativa de Janus, restringir acesso a IPs de gestão via firewall ou colocar em rede privada.

2.2 Configuração de NAT

- **nat_1_1_mapping** em `janus.jcfg`:
 - Se Janus está atrás de NAT public IP, configure `nat_1_1_mapping = "IP_Público"` para anunciar o endereço correto nos SDP ICE candidates (facsiaginsa.com).
- **IP forwarding + Port forwarding (firewall/NSG)**

- No nível da Azure NSG/NAT Gateway ou firewall on-premises, mapear UDP 10000–20000 do IP público para o IP privado da VM.
- Encaminhar TCP 80/443 para o serviço proxy que repassa a Janus.
- Assegurar health probes do load balancer (se houver) apontem para as portas de signaling (HTTP/HTTPS) e não para RTP.
- **Hairpin NAT / loopback**
 - Se clientes internos/específicos acessam via domínio público, garantir que o firewall/NAT suporte hairpin (NAT loopback) para que a VM consiga se conectar ao próprio endpoint público sem falhas.

2.3 Melhorar acessibilidade em redes restritas

- **Multiplexagem TCP 443**
 - Em redes que bloqueiam UDP, usar proxy reverso (Nginx) para encaminhar WebSocket (WSS) e TURN sobre TCP 443 (útil para fallback).
 - Configurar Nginx para rotear TLS SNI ou path-based routing (e.g., `/janus` para Janus WSS, `/turn` para servidor TURN) (voipnuggets.com, facsiaginsa.com).
 - **TLS/Certificados**
 - Usar Let's Encrypt ou outro CA confiável para certificados, garantindo compatibilidade ampla de clientes WebRTC no navegador.
 - Renovação automatizada de certificados no proxy.
 - **Firewall mais restrito**
 - Permitir apenas IPs conhecidos na interface administrativa.
 - Para signaling público, abrir apenas portas necessárias (80/443).
 - Bloquear acessos desnecessários (SSH restrito via jump host ou VPN).
 - **Monitoramento de tráfego**
 - Usar logs de NSG/Firewall para detectar tentativas anômalas de conexão em portas RTP.
 - Alertas se picos de tráfego inesperados (possível ataque DDoS).
 - **Proteção DDoS básica da Azure**
 - Em VMs críticas, considerar habilitar proteção DDoS padrão da Azure, que monitora tráfego anômalo.
-

3. Quando usar Load Balancer com Janus

Embora o cenário seja 1-para-1 e baixa concorrência, entender o momento de introduzir balanceamento de carga ajuda a planejar escalabilidade futura.

3.1 Cenários que exigem Load Balancer

- **Escalabilidade horizontal / múltiplas instâncias Janus**
 - Se em algum período for necessário suportar múltiplas sessões simultâneas em paralelo (mesmo que 1-1 cada), pode-se ter várias VMs Janus atrás de um load balancer para distribuir nova sessão aos servidores disponíveis.
 - **Alta disponibilidade:** se uma instância falhar, o LB direciona para outra sem impactar novos clientes.
- **Manutenção sem downtime**
 - Pode retirar instâncias em manutenção enquanto outras atendem as conexões.

3.2 Sticky Sessions em WebRTC

- **Persistência de sessão (affinity):**
 - WebSocket requer que, após handshake, a sessão mantenha conexão com a mesma instância Janus, pois o estado da PeerConnection vive em memória dessa instância.
 - Em Azure, se usar **Azure Load Balancer**, a afinidade pode ser configurada por “source IP affinity” (session persistence) ou usar Azure Application Gateway com suporte a WebSocket com afinidade baseada em cookie ou conexão persistente. Porém, WebRTC costuma usar WSS sem cookies, então normalmente a afinidade por IP ou hash do caminho (ex.: incluir room_id no path) é mais adequada ([janus.discourse.group](#)).
 - Outra abordagem: **DNS-based routing** combinado com lógica de cliente que verifica carga das instâncias e escolhe URL específica, mas mais complexo.
- **Hash consistente no proxy (Nginx)**
 - Se não usar LB da Azure, mas Nginx/HAProxy em VM frontal, pode usar hashing consistente no path (e.g. /janus/<session-id>) para mapear sempre à mesma backend.
 - Deve assegurar que health checks não quebrem a sessão.

3.3 Quando não é necessário

- **Uso leve, instância única:**
 - Se a demanda atual for poucas sessões paralelas ($\leq 1-4$ simultâneas) e não houver necessidade de failover imediato, pode operar com uma única VM Janus: menor complexidade e custo.
- **DNS round-robin ou fallback no cliente:**
 - Para poucas instâncias, cliente pode tentar em lista de endereços configurados; se falhar em uma, tenta outra. Menos “sticky” mas suficiente em cenários de baixa escala.
- **Sobrecarga de gerência:**

- LB traz custos adicionais (infraestrutura, configuração de health probes, afinidade, possíveis licenças/proxies). Avaliar se o benefício em HA/escala compensa em cenários de baixa utilização.

3. Instalação do Janus

Esta secção aborda todo o processo de preparação do servidor, instalação via repositório Git interno (DevOps), configuração de serviço e scripts de atualização/roll-back.

3.1 Configuração do servidor para correr o Janus

Antes de compilar e instalar o Janus, a VM precisa estar preparada para atender aos requisitos de desempenho, segurança e confiabilidade.

3.1.1 Sistema Operativo e Ambiente Básico

- **Distribuição recomendada:** Ubuntu 20.04 LTS (ou posterior).
 - Ubuntu 20.04 LTS é amplamente testado e suportado pelas bibliotecas necessárias ao Janus (byteplus.com, facsiaginsa.com).
- **Atualização do sistema:** garantir kernel e pacotes atualizados:

```
sudo apt-get update && sudo apt-get upgrade -y
# Opcional: reboot se houver nova versão de kernel instalada
```

- **Sincronização de horário:**

- Instalar e ativar `chrony` ou `ntp` para manter o relógio preciso (importante para certificados TLS e logs).

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

- **Usuário dedicado:** criar usuário e grupo “janus” com permissões mínimas:

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

- O Janus, quando executado, deve rodar sob este usuário, evitando privilégios de root (nickb.dev).

- **Diretórios de instalação e configuração:**

```
sudo mkdir -p /opt/janus/{bin,etc,var,log}
sudo chown $USER:$USER /opt/janus -R
# Após instalação, ajustar permissões para o usuário janus:
# sudo chown -R janus:janus /opt/janus
```

- **Limites de sistema (ulimits):**

- Aumentar número de descritores de arquivo se for esperado grande volume de conexões (mesmo em 1-1, logs e handles de plugins podem requerer ajuste):

```
# Em /etc/security/limits.conf (ou em arquivo em
/etc/security/limits.d/):
janus    soft    nofile   65536
janus    hard    nofile   65536
```

- Verificar `fs.file-max` no `sysctl` e ajustar se necessário:

```
sudo sysctl -w fs.file-max=100000
# Para persistir, editar /etc/sysctl.conf
```

- **Configurações de rede / kernel tuning:**

- Se NAT ou firewall na VM, ativar IP forwarding se for gateway de TURN/SFU:

```
sudo sysctl -w net.ipv4.ip_forward=1
# Persistir em /etc/sysctl.conf: net.ipv4.ip_forward=1
```

- Ajustes de buffer de rede podem ser feitos conforme carga, mas para 1-1 leve configurações padrão costumam bastar inicialmente.

- **Instalar dependências essenciais:** As bibliotecas requeridas para compilar e rodar o Janus, incluindo WebRTC, segurança, SIP/SFU e plugins:

```
sudo apt-get install -y \
    build-essential git pkg-config automake autoconf libtool cmake \
    libmicrohttpd-dev libjansson-dev libssl-dev libsrtplib2-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
```

- **Observações:**

- Incluir `libwebsockets-dev` para suporte a WebSocket interno, se usado.
- `libavformat-dev` para plugins que envolvem gravação/transcoding.
- `jq` pode ser útil em scripts de parsing de JSON em pipelines de deploy.
- Verificar documentação oficial do Janus para versões específicas de bibliotecas, pois podem mudar conforme versão do Janus (github.com).

- **Firewalls locais:**

- Enquanto pré-instalação, bloquear tudo exceto SSH para acesso remoto; após instalação, abrir apenas portas necessárias (ver seção de firewall).

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

3.1.2 Obtenção do código-fonte

- **Repositório interno (Azure DevOps Git) vs upstream GitHub:**

- Normalmente, faz-se mirror ou fork privado do repositório oficial `meetecho/janus-gateway`.
- No pipeline, usa-se a variável `$(JanusRepoUrl)` apontando para o repositório interno (pode ser protegido via SSH ou token de acesso).

- **Versão fixa:** sempre referenciar tags estáveis na compilação (`$(JanusVersion)`), garantindo reproduzibilidade.

3.2 Instalação a partir do repositório do DevOps

A seguir, exemplo de pipeline YAML no Azure DevOps para compilar e instalar o Janus em Ubuntu 20.04. Ajuste conforme as diretrizes da organização.

```
trigger:
  branches:
    include:
      - main

pool:
  vmImage: 'ubuntu-20.04'

variables:
  # Defina estas variáveis em Library ou diretamente aqui (preferir Library para
  # segredos/configuração)
  JanusRepoUrl: 'git@ssh.dev.azure.com:v3/ORG/PROJECT/JanusGateway.git'
  JanusVersion: 'v1.XX.YY' # tag ou branch estável

stages:
- stage: BuildAndInstall
  displayName: 'Build and Install Janus'
  jobs:
    - job: CompileJanus
      displayName: 'Compile Janus from source'
      steps:
        - checkout: self
          clean: true

        - task: Bash@3
          name: InstallDependencies
          displayName: 'Instalar dependências no agente'
          inputs:
            targetType: 'inline'
            script: |
              sudo apt-get update
              sudo apt-get install -y \
                build-essential git pkg-config automake autoconf libtool cmake \
                libmicrohttpd-dev libjansson-dev libssl-dev libsrtplib2-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

        - task: Bash@3
          name: CloneAndBuild
          displayName: 'Clonar e compilar Janus'
          inputs:
            targetType: 'inline'
            script: |
              # Clonar repositório
              git clone $(JanusRepoUrl) janus
              cd janus
              git fetch --all --tags
              git checkout $(JanusVersion)
              # Preparar autotools
              ./autogen.sh
              # Configurar prefixo de instalação
              ./configure --prefix=/opt/janus \
                --disable-docs \
                --disable-rabbitmq \
                --disable-mqtt \
```

```

# Ajuste flags conforme plugins necessários (revisar com a equipa:
ex: --enable-recording)
    # Exemplos: --enable-data-channels, --enable-post-processing
make -j$(nproc)
sudo make install
sudo make configs
# Ajustar permissões para usuário janus
sudo chown -R janus:janus /opt/janus

- task: Bash@3
name: SetupService
displayName: 'Configurar Systemd service para Janus'
inputs:
  targetType: 'inline'
  script: |
    # Criar arquivo de serviço systemd
    sudo tee /etc/systemd/system/janus.service > /dev/null << 'EOF'
    [Unit]
    Description=Janus WebRTC Gateway
    After=network.target
    Wants=network-online.target

    [Service]
    Type=simple
    User=janus
    Group=janus
    ExecStart=/opt/janus/bin/janus -F /opt/janus/etc/janus.jcfg
-L /opt/janus/log/janus.log
        # Para rodar em foreground e log em arquivo configurado no jcfg, pode
        usar somente ExecStart=/opt/janus/bin/janus
        Restart=on-failure
        LimitNOFILE=65536
        # Opcional: configurar sandboxing básico via systemd:
        # PrivateTmp=true
        # NoNewPrivileges=true
        # AmbientCapabilities=
        # (Avaliar conforme necessidades. Janus precisa de network e acesso
        a /opt/janus).
        Environment=LD_LIBRARY_PATH=/opt/janus/lib

    [Install]
    WantedBy=multi-user.target
EOF
    sudo systemctl daemon-reload
    sudo systemctl enable janus.service
    # Não iniciar agora; iniciar manualmente ou em fase de deploy.

- task: Bash@3
name: VerifyCompilation
displayName: 'Verificar binário Janus'
inputs:
  targetType: 'inline'
  script: |
    if [ ! -x /opt/janus/bin/janus ]; then
      echo "Janus binário não encontrado ou não executável"; exit 1
    fi
    /opt/janus/bin/janus --version

# Opcional: publicar artefatos (logs de compilação, tar.gz de /opt/janus)
para inspeção
- publish: $(Build.SourcesDirectory)/janus
  artifact: janus-source

```

Explicações e boas práticas:

- **Clean checkout:** assegura que não há artefatos antigos.
- **Variáveis de Pipeline:** como `JanusRepoUrl` e `JanusVersion`, definidas em Library protegida ou variáveis seguras, para controlar a versão sem alterar o YAML.
- **Flags de configuração:**
 - Ajustar `--disable-...` ou `--enable-...` conforme os plugins realmente necessários (recording, data-channels, SIP, etc.). Isso minimiza dependências e tempo de compilação.
- **Permissões:** após `make install`, mudar `/opt/janus` para `janus:janus`.
- **Systemd service:**
 - Usa `ExecStart=/opt/janus/bin/janus -F ... -L ...` para apontar config e log.
 - Ajusta `LimitNOFILE` compatível com ulimits configurados.
 - Pode adicionar diretivas de sandboxing (e.g., `PrivateTmp=true`), mas requer teste porque Janus precisa de acesso a rede e arquivos.
- **Logs:** Janus por padrão escreve logs em `stdout/stderr`; indicamos redirecionar via systemd ou via parâmetro `-L /opt/janus/log/janus.log` ou configurado no `janus.jcfg`.
- **Saúde do build:** checar saída de `janus --version` garante que a compilação ocorreu como esperado.
- **Artefatos:** opcionalmente armazenar o código-fonte compilado ou pacotes `.deb` criados a partir de `/opt/janus` para rollback.

3.2.1 Configurações adicionais no servidor antes de iniciar Janus

- **Certificados TLS:**
 - Se o Janus for exposto diretamente em HTTPS, obter certificados (Let's Encrypt ou gerenciados pela infra).
 - Para proxy reverso (Nginx/Traefik), configurar virtual host apontando para Janus em 8088/8089/8188/8989.
- **Variáveis de ambiente:**
 - `LD_LIBRARY_PATH=/opt/janus/lib` caso libs instaladas em local não padrão.
 - Se usar plugins externos que dependem de variáveis adicionais, configurar em `/etc/default/janus` ou env file systemd (`EnvironmentFile=`).
- **Logrotate:**

- Criar arquivo `/etc/logrotate.d/janus` para evitar crescimento indefinido de logs:

```
/opt/janus/log/janus.log {
    daily
    rotate 14
    compress
    missingok
    notifempty
    copytruncate
}
```

- **Monitoramento:**

- Instalar agentes de monitoramento (Azure Monitor Agent) para coletar métricas de CPU, memória, disco, rede e logs da aplicação (janelas de log em `/opt/janus/log`).
- Configurar alertas: CPU > X%, memória > Y% durante sessão para ajustar VM.

3.3 Script para update do serviço

Ao precisar atualizar a versão do Janus em produção, é importante ter processo repetível, com capacidade de rollback caso a nova versão apresente problemas. A seguir, um exemplo de job em pipeline DevOps para atualização:

```
- stage: DeployJanus
  displayName: 'Deploy/Update Janus em produção'
  dependsOn: BuildAndInstall
  jobs:
    - deployment: UpdateJanus
      displayName: 'Deploy Janus'
      environment: 'production'
      pool:
        vmImage: 'ubuntu-20.04'
      strategy:
        runOnce:
          deploy:
            steps:
              - checkout: self

              - task: Bash@3
                name: BackupCurrentVersion
                displayName: 'Backup versão atual de Janus'
                inputs:
                  targetType: 'inline'
                  script: |
                    # Exemplo: mover /opt/janus para /opt/janus_backup_TIMESTAMP
                    TIMESTAMP=$(date '+%Y%m%d%H%M%S')
                    if [ -d /opt/janus ]; then
                      sudo mv /opt/janus /opt/janus_backup_${TIMESTAMP}
                    fi

              - task: Bash@3
                name: CloneNewVersion
                displayName: 'Clonar e compilar nova versão Janus'
                inputs:
                  targetType: 'inline'
                  script: |
                    # Clonar repo interno
                    git clone $(JanusRepoUrl) janus_new
                    cd janus_new
```

```

git fetch --all --tags
git checkout $(JanusNewVersion)
./autogen.sh
./configure --prefix=/opt/janus \
    --disable-docs \
    # flags ajustadas conforme necessidade
make -j$(nproc)
sudo make install
sudo make configs
sudo chown -R janus:janus /opt/janus

- task: Bash@3
  name: ValidateNewBinary
  displayName: 'Validar instalação'
  inputs:
    targetType: 'inline'
    script: |
      if [ ! -x /opt/janus/bin/janus ]; then
        echo "Falha: binário Janus não encontrado"; exit 1
      fi
      /opt/janus/bin/janus --version

- task: Bash@3
  name: RestartJanus
  displayName: 'Reiniciar serviço Janus'
  inputs:
    targetType: 'inline'
    script: |
      sudo systemctl restart janus
      # Aguardar e verificar status
      sleep 5
      sudo systemctl is-active janus || (echo "Janus não iniciou
corretamente"; exit 1)
      sudo journalctl -u janus --since "2 minutes ago" | tail -n 50

      # Em caso de falha, o pipeline falhará e não avançará.
      # Para rollback manual:
      # - Parar nova instância: sudo systemctl stop janus
      # - Remover /opt/janus atual, restaurar backup: sudo rm -rf
/opt/janus; sudo mv /opt/janus_backup_TIMESTAMP /opt/janus
      # - Ajustar permissões, systemctl daemon-reload e reiniciar: sudo
systemctl restart janus

- task: Bash@3
  name: CleanupOldBackups
  displayName: 'Limpar backups antigos (opcional)'
  inputs:
    targetType: 'inline'
    script: |
      # Manter apenas os X backups mais recentes, remover demais
      cd /opt
      ls -dt janus_backup_* | tail -n +6 | xargs -r sudo rm -rf

```

Aspectos de um processo de update robusto:

- **Backup prévio:** mover a instalação antiga para permitir rollback.
- **Compilação isolada:** clonar em diretório temporário `janus_new` para não sobrescrever até compilação completada.

- **Validação:** checar se o binário foi criado e executa `janus --version`. Poderia inclusive executar testes básicos ou health-check de Janus (por exemplo, probe HTTP/HTTPS se habilitado).
- **Reinício controlado:** reiniciar serviço via `systemd` e verificar status. Capturar logs recentes (`journalctl`) para inspeção rápida.
- **Rollback documentado:** instruções claras de como restaurar a versão anterior via backups. Poderia automatizar rollback em pipeline caso health-check falhe.
- **Limpeza de backups antigos:** evitar acumular muitos backups que consomem disco.
- **Notificações/alertas:** enviar notificações (e-mail, Slack) sobre sucesso/falha do deploy. Integrar com sistema de monitoramento para rastrear regressões após atualização.

3.3.1 Alternativa: versão baseada em diretórios nomeados

Em vez de sempre instalar em `/opt/janus`, pode-se usar instalação versionada:

- Compilar e instalar em `/opt/janus-<versão>`, ex.: `/opt/janus-1.10.6`.
- Manter symlink `/opt/janus_current → /opt/janus-1.10.6`.
- Systemd service usa `ExecStart /opt/janus_current/bin/janus`.
- No update, compilar em novo diretório, alterar symlink e reiniciar.
- Vantagem: múltiplas versões podem coexistir, facilitando rollbacks imediatos.
- Atenção: ajustar `LD_LIBRARY_PATH` e permissões para o usuário janus.

3.3.2 Configurações de plugins e arquivos de configuração

- **Configuração principal (`janus.jcfg`):**
 - Geralmente copiada de `/opt/janus/etc/janus/janus.jcfg`. Ajustar:
 - Interfaces de rede (ip), NAT mapping (`nat_1_1_mapping = "IP_Público"`).
 - `rtp_port_range = "10000-20000"`.
 - SSL paths (certificados privados e públicos) se Janus gerenciar TLS; caso contrário, deixar TLS ao proxy.
 - Ajustes de logging (nível de log, arquivos).
- **Configurações de plugins** (em `/opt/janus/etc/janus/`):
 - Habilitar/desabilitar plugins conforme necessidade (transcoder, recording, streaming, etc.).
 - Verificar dependências de bibliotecas (e.g., GStreamer) e compilações adicionais se usar plugins avançados.
- **Persistência de configuração no pipeline:**

- Armazenar arquivos de configuração em repositório de configuração (ex.: GitOps), aplicar em pipeline de deploy via cópia para `/opt/janus/etc/janus/`.
- Validar sintaxe de JSON antes de reiniciar: `jq . /opt/janus/etc/janus/janus.jcfg` para garantir JSON válido.
- **Migrações e mudanças na configuração:**
 - Documentar alterações de versão para versão: se uma nova versão do Janus introduzir parâmetros novos ou depreciar outros, ajustar `janus.jcfg` no pipeline antes de deploy.

4. Instanciação da API para Booking

Nesta seção, descreve-se como a API de agendamento de salas (booking) está empacotada, configurada e executada em container Docker, e como ela se integrará futuramente ao container do Suroga API.

4.1 Containerização da API

- **Objetivo:**

A API de booking é empacotada num container Docker, facilitando deploy consistente em diferentes ambientes (desenvolvimento, staging, produção) e futura integração em orquestração junto a outros serviços do Suroga (e.g., autenticação, front-end, Janus gateway proxies, etc.).

- **Estrutura do Dockerfile:**

Um Dockerfile típico para a API Django/DRF pode ter:

```

dockerfile
CopyEdit
# Imagem base leve com Python
FROM python:3.11-slim

# Definir diretório de trabalho
WORKDIR /app

# Copiar requirements
COPY requirements.txt .

# Instalar dependências do sistema operacional necessárias para pacotes
# Python (p. ex., psycopg2, libs)
RUN apt-get update && apt-get install -y \
    build-essential libpq-dev netcat \
    && rm -rf /var/lib/apt/lists/*

# Instalar dependências Python
RUN pip install --no-cache-dir -r requirements.txt

# Copiar código da API
COPY . .

# Variáveis de ambiente típicas
ENV DJANGO_SETTINGS_MODULE=myproject.settings.prod
ENV PYTHONUNBUFFERED=1

# Criar diretório para logs, se necessário

```

```

RUN mkdir -p /vol/log && chown -R nobody:nogroup /vol/log

# Estágio de coleta de estáticos, migrações, etc., se necessário
# Por exemplo:
# RUN python manage.py collectstatic --noinput

# Comando de inicialização: gunicorn ou outro WSGI server
CMD ["gunicorn", "myproject.wsgi:application", "--bind", "0.0.0.0:8000",
"--workers", "4"]

```

- **Dependências de sistema:** incluir apenas o mínimo necessário (e.g., libpq-dev se usar PostgreSQL).

- **Multi-stage builds:** se for relevante (e.g., compilar assets frontend), separar build/produção para reduzir imagem final.

- **Usuário não-root:** para aumentar segurança, criar e usar usuário não privilegiado:

```

dockerfile
CopyEdit
RUN addgroup --system django && adduser --system --ingroup django
django
USER django

```

- **Volume para logs ou arquivos temporários:** se a API gerar arquivos temporários (e.g., relatórios, cache local), mapear volumes externos.

- **Configurações de ambiente:** variáveis como DATABASE_URL, JANUS_URL, JANUS_KEY, SECRET_KEY, DEBUG=False, ALLOWED_HOSTS, definidas em variáveis de ambiente no container ou secret store do orquestrador.

- **Separação de Ambientes:**

- Em cada ambiente (desenvolvimento, staging, produção), passar variáveis apropriadas via pipeline/devOps ou orquestrador (Kubernetes, Azure App Service, Docker Compose em VM, etc.).
- Usar arquivo .env.example com chaves nomeadas (sem valores sensíveis), para referenciar em docs de equipe.

- **Health Check:**

- Definir endpoint /health/ ou similar que retorne status OK se a API conseguir conectar ao banco e, opcionalmente, consiga conversar com Janus (pode ser verificação leve, p. ex., testar URL de Janus ou pool de conexões).
- Health check é usado pelo orquestrador para reiniciar container caso fique indisponível ou emloop de erro.

- **Logging e métricas:**

- Configurar logger do Django para enviar logs estruturados para stdout/stderr, permitindo que orquestrador/carregador de logs (e.g., Azure Monitor, ELK) capture e processe.
- Exportar métricas via endpoint ou integrar com bibliotecas como Prometheus client, se necessário.

- Rotacionar logs (mas em container geralmente se redireciona para stdout).
- **Dependências externas:**
 - Banco de dados (ex.: PostgreSQL) em serviço gerenciado ou container separado.
 - Janus gateway: a API faz chamadas HTTP ao endpoint Janus; configurar URL base via variável JANUS_URL (p. ex., <https://janus.example.com/janus>).
 - Autenticação/Autorização: se for integrada com Suroga API, aguardar integração futura. Hoje usar token ou JWT para autenticar requests; definir cabeçalhos esperados (e.g., Authorization: Bearer <token>).

4.2 Instalação do Container Docker API a partir do DevOps

- **Repositório e pipeline de CI/CD:**
 - O código da API está versionado em repositório (Azure DevOps Git, GitHub ou similar).
 - Definir pipeline de build que:
 1. Faz checkout do código na branch principal (e.g., main ou release/*).
 2. Executa testes automatizados (unitários, lint, qual cobertura mínima).
 3. Constrói a imagem Docker:
 - Usa o Dockerfile para criar imagem com tag semântica (e.g., registry.azurecr.io/suroga-booking-api:\$ (Build.BuildId) ou :v1.2.3).
 4. Faz scan de vulnerabilidades (se integrado, opcional).
 5. Publica imagem no Container Registry (Azure Container Registry ou outro registry privado), autenticação via service principal ou token de acesso.
 6. Opcional: publicar artefatos de build (e.g., relatório de teste, logs).
- **Exemplo de YAML de pipeline (Azure Pipelines):**

```

yaml
CopyEdit
trigger:
  branches:
    include:
      - main
pool:
  vmImage: 'ubuntu-latest'

variables:
  imageName: 'suroga-booking-api'
  registry: 'myregistry.azurecr.io'
  # Definir connection ao ACR em service connection no Azure DevOps

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' # Configurado
no Azure DevOps
    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: |
      # Exemplos de deploy:
      # Se usar Azure App Service for Containers:
      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

      # Se usar Azure Kubernetes Service:
      # kubectl set image deployment/booking-api booking-
      api=$(registry)/$(imageName):$(Build.BuildId)
      # kubectl rollout status deployment/booking-api

```

1. Citações:

- Exemplo de pipeline para deploy de containers no Azure Pipelines learn.microsoft.com/learn.microsoft.com.
- Boas práticas de build e push de Docker images para Azure Container Registry learn.microsoft.com.

- **Configuração de segredos e variáveis de ambiente:**
 - No Azure DevOps, definir variáveis seguras para DJANGO_SECRET_KEY, DATABASE_URL, JANUS_URL, JANUS_KEY, ALLOWED_HOSTS, credenciais de ACR, credenciais de acesso ao banco, tokens de autenticação externos.
 - No runtime, orquestrador (App Service, Kubernetes, Docker Compose) injeta variáveis no container.
- **Deployment:**
 - Em produção, pode-se usar:
 1. **Azure App Service for Containers**: fácil deploy de container único; configurar image, variáveis de ambiente, número de instâncias, escalonamento automático.
 2. **Azure Kubernetes Service (AKS)**: para cenários com múltiplos serviços (booking API, autenticação, front-end) orquestrados; definir Deployments, Services, Ingress, ConfigMaps/Secrets.
 3. **Azure Container Instances (ACI)** ou VM com Docker Compose: para cenários simples.
 - Garantir health probe: apontar para endpoint /health/ para reiniciar container em caso de falha.
 - Configurar auto-scale baseado em métricas (CPU, memória, número de requests) se demanda variar.
- **Integração futura com Suroga API:**
 - A API de booking pode rodar como container próprio ou como parte de um container multi-service (usando Docker Compose ou sidecar).
 - Definir um orquestrador (Kubernetes, Docker Compose, Azure Container Apps) que coordene:
 1. Booking API, Front-end (web UI), Autenticação API, possivelmente gateways (NGINX), Janus proxies, serviços auxiliares (Redis, Celery broker).
 - Documentar dependências de rede entre containers: por exemplo, Booking API precisa conectar ao Janus via URL interna (DNS do cluster), banco de dados compartilhado, serviço de autenticação.
- **Versionamento e tags:**
 - Adotar versionamento SemVer para a API (tags Git, labels na imagem Docker).
 - Em rota de API, incluir versionamento de path se necessário (ex.: /api/v1/bookings/), para futura evolução sem quebrar clientes.

5. API Endpoints

Apresentar a lista de endpoints, métodos HTTP, descrições, parâmetros de entrada (query params e corpo JSON), formatos de resposta, códigos de status e exemplos. A seguir, um modelo baseado nas views refatoradas sugeridas anteriormente.

Nota: Aqui consideramos uma estrutura RESTful mais uniforme. Ajuste nomes conforme convenção do projeto.

5.1 Autenticação

- **Header:** Authorization: Bearer <token>
- A API requer autenticação para quase todas rotas.
- Caso use JWT, documentar endpoint de login (POST /auth/login/) e refresh de token, se aplicável. (Se a autenticação for responsabilidade de outro serviço Suroga, apenas referenciar como obter token.)

5.2 Health Check

- **GET /health/**
 - **Descrição:** Verifica se a API está operacional, consegue conectar ao banco e ao Janus (opcional).
 - **Resposta (200 OK):**

```
json
CopyEdit
{
    "status": "ok",
    "database": "ok",
    "janus": "ok"
}
```
 - **Possíveis falhas:**
 - 500 Internal Server Error se a API não conectar ao banco ou Janus.

5.3 Rooms

5.3.1 Listar Salas

- **GET /rooms/**
- **Descrição:** Retorna lista de salas atualmente cadastradas no sistema. Não inclui senha por padrão.
- **Query Params** (opcionais):
 - `available=true` — filtra apenas salas livres num horário: junto com `start_time` e `end_time`. Ex.: `/rooms/?available=true&start_time=2025-06-20T10:00:00Z&end_time=2025-06-20T11:00:00Z`.

- Outros filtros: `max_participants`, `created_before`, `created_after`, conforme necessidade.
- **Resposta (200 OK):**

```
json
CopyEdit
[
  {
    "room_id": 12345,
    "max_participants": 2,
    "created_at": "2025-06-10T12:00:00Z",
    "updated_at": "2025-06-10T12:00:00Z"
  },
  {
    "room_id": 67890,
    "max_participants": 4,
    "created_at": "...",
    "updated_at": "..."
  }
]
```

- **Erros:**
 - 401 Unauthorized se sem token válido.
 - 400 Bad Request para parâmetros inválidos (e.g., data mal formatada em filter de disponibilidade).

5.3.2 Criar Sala

- **POST /rooms/**
- **Descrição:** Cria nova sala no sistema e no Janus. Gera `room_id` único e senha aleatória.
- **Body (JSON):**

```
json
CopyEdit
{
  "max_participants": 2,
  "optional_name": "Sala de reunião X",
  "metadata": {...} // se houver
}
```

- **Resposta (201 Created):**

```
json
CopyEdit
{
  "room_id": 54321,
  "password": "Ab3$kl9pQ",
  "max_participants": 2,
  "created_at": "2025-06-12T09:00:00Z"
}
```

- **Erros:**
 - 400 Bad Request se payload inválido (e.g., `max_participants` fora de faixa).
 - 401 Unauthorized se não autenticado.

- 502 Bad Gateway ou 503 Service Unavailable se falha ao criar sala no Janus: a API deve reverter a criação local e retornar mensagem:

```
json
CopyEdit
{
  "detail": "Erro ao criar sala no serviço de WebRTC. Tente
  novamente mais tarde."
}
```

5.3.3 Detalhar Sala

- **GET /rooms/<room_id>/**
- **Descrição:** Retorna detalhes da sala (sem expor senha).
- **Resposta (200 OK):**

```
json
CopyEdit
{
  "room_id": 54321,
  "max_participants": 2,
  "created_at": "...",
  "updated_at": "...",
  "has_future_booking": true
}
```

- **Erros:**
 - 404 Not Found se sala não existir.
 - 401 Unauthorized.

5.3.4 Deletar Sala (opcional / Admin)

- **DELETE /rooms/<room_id>/**
- **Descrição:** Remove sala do sistema e requisita remoção no Janus. Geralmente restrito a administradores.
- **Resposta (204 No Content)** em sucesso.
- **Erros:**
 - 403 Forbidden se usuário não tiver permissão.
 - 404 Not Found se sala não existir.
 - 409 Conflict se houver bookings futuros; impedir remoção até encerrar bookings ou cancelar manualmente.
 - 502/503 se falha ao remover no Janus: retornar erro e não remover local.

5.4 Disponibilidade de Salas

- **GET /rooms/availability/**
- **Descrição:** Retorna salas livres no intervalo solicitado.

- **Query Params (obrigatórios):**

- `start_time` (ISO 8601 UTC): ex.: `2025-06-20T10:00:00Z`.
- `end_time` (ISO 8601 UTC): ex.: `2025-06-20T11:00:00Z`.

- **Resposta (200 OK):**

```
json
CopyEdit
{
  "available_rooms": [
    {
      "room_id": 12345,
      "max_participants": 2
    },
    {
      "room_id": 67890,
      "max_participants": 4
    }
  ]
}
```

- Lista vazia se nenhuma disponível:

```
json
CopyEdit
{
  "available_rooms": []
}
```

- **Erros:**

- 400 Bad Request se parâmetros ausentes ou inválidos.
- 401 Unauthorized.

- **Observações de implementação:**

- Verificar `start_time < end_time`.
- Converter para UTC e usar transação ou cache conforme a frequência de chamadas.
- Não lançar 404 quando lista vazia; retornar array vazio com 200, pois a API funcionou corretamente.

5.5 Bookings

5.5.1 Listar Bookings do Usuário

- **GET /bookings/**

- **Descrição:** Retorna reservas do usuário autenticado, ordenadas por data (passadas e futuras).
- **Resposta (200 OK):**

```
json
CopyEdit
[
  {
    "booking_id": 101,
```

```

    "room_id": 12345,
    "start_time": "2025-06-20T10:00:00Z",
    "end_time": "2025-06-20T11:00:00Z",
    "status": "upcoming" // ou "past", ou "cancelled"
},
...
]

```

- **Erros:**

- 401 Unauthorized.

- **Filtros opcionais:**

- ?status=upcoming para apenas futuras.
- ?date=2025-06-20 para reservas em data específica.

5.5.2 Criar Booking (find-or-create de sala)

- **POST /bookings/**

- **Descrição:** Reserva automática: se room_id for informado no corpo, tenta reservar essa sala; se não for informado, busca sala livre no intervalo e, se não existir, cria sala nova e reserva.

- **Body (JSON):**

```

json
CopyEdit
{
  "start_time": "2025-06-20T10:00:00Z",
  "end_time": "2025-06-20T11:00:00Z",
  // opcional:
  "room_id": 12345
}

```

- **Resposta (201 Created):**

- Em ambos os casos, retorna:

```

json
CopyEdit
{
  "booking_id": 202,
  "room_id": 12345,
  "password": "Ab3$kl9pQ",
  "start_time": "2025-06-20T10:00:00Z",
  "end_time": "2025-06-20T11:00:00Z"
}

```

- Se sala informada estiver disponível, a API não cria nova sala.
 - Se sala informada não estiver disponível, retorna erro 409 Conflict:
- ```

json
CopyEdit
{"detail": "Sala 12345 já reservada no intervalo solicitado."}

```
- Se nenhuma sala livre e falha ao criar sala no Janus, retorna 503 Service Unavailable:

```
json
CopyEdit
{"detail": "Erro ao criar nova sala no serviço de WebRTC. Tente
novamente mais tarde."}
```

- **Erros:**

- 400 Bad Request se parâmetros ausentes ou `start_time` >= `end_time`.
- 401 Unauthorized.
- 404 Not Found se `room_id` informado não existir.
- 409 Conflict se sala existente mas indisponível.
- 503 Service Unavailable para falha na integração com Janus.

- **Implementação recomendada:**

- Usar `transaction.atomic()` e `select_for_update()` para prevenir race conditions ao verificar e criar reserva.
- Validar e parsear datas via serializer DRF (`DateTimeField`), garantindo timezone-aware em UTC [django-rest-framework.org](https://django-rest-framework.org).
- Chamada a Janus para criar sala (quando necessário) preferencialmente em tarefa assíncrona (Celery), mas podendo ser síncrona com timeout curto e rollback local em caso de falha.
- Uso de `ExclusionConstraint` no Postgres (`DateTimeRangeField`) para prevenir sobreposição, se disponível no projeto [learn.microsoft.com](https://learn.microsoft.com).

### 5.5.3 Detalhar Bookings

- **GET /bookings/<booking\_id>/**

- **Descrição:** Retorna detalhes de uma reserva específica, se pertence ao usuário ou se o usuário tem permissão.

- **Resposta (200 OK):**

```
json
CopyEdit
{
 "booking_id": 202,
 "room_id": 12345,
 "start_time": "2025-06-20T10:00:00Z",
 "end_time": "2025-06-20T11:00:00Z",
 "password": "Ab3$KL9pQ", // Se estiver próximo da sessão ou se for
 permitido ao usuário
 "status": "upcoming"
}
```

- **Erros:**

- 404 Not Found se booking não existir ou não pertencer ao usuário.
- 401 Unauthorized.

- **Observação:** A senha da sala pode ser retornada apenas se a reserva for ativa ou próxima da hora de início; caso contrário, omitir para segurança.

#### 5.5.4 Cancelar Booking

- **DELETE** /bookings/<booking\_id>/ ou **POST** /bookings/<booking\_id>/cancel/
- **Descrição:** Cancela reserva existente antes do início.
- **Resposta (204 No Content)** em sucesso.
- **Erros:**
  - 403 Forbidden se já tiver passado (não pode cancelar reserva concluída).
  - 404 Not Found se booking não existir ou não pertencer ao usuário.
  - 400 Bad Request se tentativa de cancelar fora do prazo permitido (por regra de negócio, e.g., mínimo 1h de antecedência).
- **Efeito em Janus:**
  - Caso a sala tenha sido criada especificamente para essa reserva (e não será reutilizada), pode-se agendar a remoção da sala no Janus após o término normal, ou imediatamente se desejado.
  - Se sala for compartilhada (pool de salas pré-criadas), não remover imediatamente; apenas liberar slot.
  - Implementar lógica em background (Celery) para chamadas à API Janus de remoção ou alteração de estado da sala.

#### 5.6 Reset de Senhas

- **POST** /rooms/<room\_id>/password/reset/
- **Descrição:** Gera nova senha para a sala especificada e atualiza no Janus. Endereço geralmente restrito a administradores ou processos automatizados.
- **Resposta (200 OK):**

```
json
CopyEdit
{
 "room_id": 12345,
 "new_password": "Xy9#Ab7M"
}
```

- **Erros:**
  - 403 Forbidden se usuário não tiver permissão.
  - 404 Not Found se sala não existir.
  - 503 Service Unavailable se falha na atualização no Janus.
- **Implementação:**

- Validar que a sala não tem booking em andamento ou futura (ou se for permitido, forçar reset).
- Usar transação: atualizar `Room.password` localmente, chamar Janus. Se Janus falhar, reverter senha antiga no DB ou marcar inconsistente.
- Usar `requests.post(..., timeout=...)` e logger para capturar erros.

- **Reset em lote:**

- Não expor endpoint público para reset em massa; implementar tarefa periódica (Celery Beat) que executa lógica de reset em salas sem bookings futuras, conforme discutido anteriormente.

## 5.7 Participantes ativos na sala

- **GET /rooms/<room\_id>/participants/**
- **Descrição:** Consulta Janus para obter lista de participantes ou apenas quantidade, em tempo real.
- **Resposta (200 OK):**

```
json
CopyEdit
{
 "room_id": 12345,
 "participant_count": 2,
 "participants": [
 {
 "id": 5678,
 "display": "user@example.com",
 "metadata": {...}
 },
 {
 "id": 6789,
 "display": "other@example.com",
 "metadata": {...}
 }
]
}
```

- **Erros:**

- 404 Not Found se sala local não existir.
- 502 Bad Gateway se falha ao contatar Janus.
- 401 Unauthorized.

- **Observações:**

- Cache curto para reduzir chamadas repetidas (e.g., 5 segundos).
- Timeout configurado; tratamento de exceções de rede.
- Checar autorização: apenas usuário da reserva ou admin pode visualizar participantes.

## 5.8 Documentação Automática

- **OpenAPI / Swagger:**
  - Usar ferramentas como **drf-spectacular** ou **drf-yasg** para gerar documentação interativa.
  - Cada endpoint documentado com:
    - Descrição clara do propósito.
    - Parâmetros (path, query, body) com exemplos.
    - Respostas de sucesso e erro (códigos HTTP e schema JSON).
  - Exibir via UI (e.g., `/docs/` ou `/swagger/`) para que consumidores (frontend, mobile, parceiros) testem e entendam a API. [django-rest-framework.org/rootstrap.com](https://django-rest-framework.org/rootstrap.com).
- **Exemplos em Postman / OpenAPI JSON:**
  - Incluir coleção Postman ou link para arquivo OpenAPI, facilitando importação.

## 5.9 Versionamento de API

- Definir prefixo de versão se houver necessidade de evoluir sem quebrar clientes existentes.  
Exemplo: `/api/v1/rooms/`, `/api/v1/bookings/`.
- Documentar política de versionamento (SemVer, data-based, etc.).

## 5.10 Segurança e Políticas de Rate Limiting

- **CORS:** configurar se frontend estiver em domínio diferente.
- **Throttling:** limitar número de requests por cliente para evitar abusos (e.g., DRF Throttling).
- **Proteção CSRF:** em APIs REST token-based, desativar CSRF ou usar mecanismos adequados.
- **Headers de segurança:** configurar via servidor/proxy (e.g., HSTS, X-Content-Type-Options).
- **Controle de acesso:** permissions classes (IsAuthenticated, IsAdminUser ou customizadas) para endpoints sensíveis (criação/remoção de salas, reset de senha).

## 5.11 Exemplos de Uso (cURL)

- **Listar salas disponíveis:**

```
bash
CopyEdit
curl -X GET "https://api.suroga.com/api/v1/rooms/?available=true&start_time=2025-06-20T10:00:00Z&end_time=2025-06-20T11:00:00Z" \
-H "Authorization: Bearer <token>"
```

- **Criar reserva automática:**

```
bash
```

```
CopyEdit
curl -X POST "https://api.suroga.com/api/v1/bookings/" \
 -H "Authorization: Bearer <token>" \
 -H "Content-Type: application/json" \
 -d '{
 "start_time": "2025-06-20T10:00:00Z",
 "end_time": "2025-06-20T11:00:00Z"
}'
```

- **Reservar sala específica:**

```
bash
CopyEdit
curl -X POST "https://api.suroga.com/api/v1/bookings/" \
 -H "Authorization: Bearer <token>" \
 -H "Content-Type: application/json" \
 -d '{
 "room_id": 12345,
 "start_time": "2025-06-20T10:00:00Z",
 "end_time": "2025-06-20T11:00:00Z"
}'
```

- **Resetar senha de sala (admin):**

```
bash
CopyEdit
curl -X POST "https://api.suroga.com/api/v1/rooms/12345/password/reset/" \
 -H "Authorization: Bearer <admin-token>"
```

- **Listar participantes:**

```
bash
CopyEdit
curl -X GET "https://api.suroga.com/api/v1/rooms/12345/participants/" \
 -H "Authorization: Bearer <token>"
```

## 5.12 Exceções e Códigos de Erro

Para cada endpoint, documentar:

- **400 Bad Request:** parâmetros ausentes ou inválidos (formatos de data, valores fora de faixa).
- **401 Unauthorized:** token faltando ou inválido.
- **403 Forbidden:** falta de permissão para ação (e.g., deletar sala, reset de senha).
- **404 Not Found:** recurso não existe (sala ou booking).
- **409 Conflict:** conflito de booking ou outro estado inconsistente.
- **502 Bad Gateway / 503 Service Unavailable:** falha de integração externa (Janus inacessível).
- **500 Internal Server Error:** falha não prevista; registrar logs para investigação.

## 5.13 Integração com Janus

- Em chamadas que criam, editam ou removem salas, detalhar payloads enviados ao Janus:

- **Create room:**

```
json
CopyEdit
{
 "janus": "message",
 "body": {
 "request": "create",
 "room": <room_id>,
 "secret": "<senha>",
 "publishers": <max_participants>,
 "permanent": false
 },
 "transaction": "<uuid>",
 "apisecret": "<JANUS_KEY>"
}
```

- **Edit room** (e.g., alterar senha):

```
json
CopyEdit
{
 "janus": "message",
 "body": {
 "request": "edit",
 "room": <room_id>,
 "secret": "<senha_atual>",
 "new_secret": "<senha_nova>",
 "permanent": false
 },
 "transaction": "<uuid>",
 "apisecret": "<JANUS_KEY>"
}
```

- **Delete room:**

```
json
CopyEdit
{
 "janus": "message",
 "body": {
 "request": "destroy",
 "room": <room_id>,
 "secret": "<senha>"
 },
 "transaction": "<uuid>",
 "apisecret": "<JANUS_KEY>"
}
```

- **List participants:**

```
json
CopyEdit
{
 "janus": "message",
 "body": {
 "request": "listparticipants",
 "room": <room_id>
 },
 "transaction": "<uuid>",
 "apisecret": "<JANUS_KEY>"
}
```

- Documentar no manual da API quais endpoints internos do Janus são usados, para facilitar debugging e futuras manutenções.
  - Incluir exemplos de resposta do Janus e como a API os processa para retornar ao cliente.
- 

## 6. Exemplo de Documentação (OpenAPI / Swagger)

- **Título e versão:**

```
yaml
CopyEdit
openapi: 3.0.3
info:
 title: Suroga Booking API
 version: "1.0.0"
 description: API para configuração e agendamento de salas WebRTC via
Janus.
servers:
 - url: https://api.suroga.com/api/v1
components:
 securitySchemes:
 BearerAuth:
 type: http
 scheme: bearer
 bearerFormat: JWT
 security:
 - BearerAuth: []
paths:
 /rooms/:
 get:
 summary: Lista salas
 parameters:
 - in: query
 name: available
 schema:
 type: boolean
 description: Filtrar apenas salas livres (requer start_time e
end_time).
 - in: query
 name: start_time
 schema:
 type: string
 format: date-time
 description: Início para filtro de disponibilidade (UTC).
 - in: query
 name: end_time
 schema:
 type: string
 format: date-time
 description: Fim para filtro de disponibilidade.
 responses:
 '200':
 description: Lista de salas.
 content:
 application/json:
 schema:
 type: array
 items:
 $ref: '#/components/schemas/Room'
 post:
```

```

summary: Cria nova sala
requestBody:
 required: true
 content:
 application/json:
 schema:
 $ref: '#/components/schemas/RoomCreate'
responses:
 '201':
 description: Sala criada com sucesso.
 content:
 application/json:
 schema:
 $ref: '#/components/schemas/RoomDetail'
/rooms/{room_id}:
get:
 summary: Detalha sala
 parameters:
 - name: room_id
 in: path
 required: true
 schema:
 type: integer
 responses:
 '200':
 description: Detalhes da sala.
 content:
 application/json:
 schema:
 $ref: '#/components/schemas/RoomDetail'
 '404':
 description: Sala não encontrada.
delete:
 summary: Deleta sala (admin)
 parameters:
 - name: room_id
 in: path
 required: true
 schema:
 type: integer
 responses:
 '204':
 description: Sala removida.
 '403':
 description: Sem permissão ou há bookings futuros.
 '404':
 description: Sala não encontrada.
/rooms/{room_id}/participants:
get:
 summary: Lista participantes em tempo real
 parameters:
 - name: room_id
 in: path
 required: true
 schema:
 type: integer
 responses:
 '200':
 description: Quantidade e lista de participantes.
 content:
 application/json:
 schema:
 $ref: '#/components/schemas/RoomParticipants'
 '502':

```

```

 description: Falha ao contatar Janus.
/rooms/{room_id}/password/reset/:
post:
 summary: Reseta senha da sala (admin)
 parameters:
 - name: room_id
 in: path
 required: true
 schema:
 type: integer
 responses:
 '200':
 description: Senha resetada.
 content:
 application/json:
 schema:
 type: object
 properties:
 room_id:
 type: integer
 new_password:
 type: string
 '403':
 description: Sem permissão.
 '503':
 description: Falha no Janus.
/rooms/availability/:
get:
 summary: Verifica disponibilidade de salas
 parameters:
 - in: query
 name: start_time
 schema:
 type: string
 format: date-time
 required: true
 - in: query
 name: end_time
 schema:
 type: string
 format: date-time
 required: true
 responses:
 '200':
 description: Salas disponíveis.
 content:
 application/json:
 schema:
 type: object
 properties:
 available_rooms:
 type: array
 items:
 $ref: '#/components/schemas/RoomAvailability'
/bookings/:
get:
 summary: Lista bookings do usuário
 responses:
 '200':
 description: Lista de reservas.
 content:
 application/json:
 schema:
 type: array

```

```

 items:
 $ref: '#/components/schemas/BookingDetail'
post:
 summary: Cria booking (find-or-create)
 requestBody:
 required: true
 content:
 application/json:
 schema:
 $ref: '#/components/schemas/BookingCreate'
responses:
 '201':
 description: Reserva criada.
 content:
 application/json:
 schema:
 $ref: '#/components/schemas/BookingResponse'
 '409':
 description: Conflito de reserva.
 '503':
 description: Erro ao criar sala no Janus.
/bookings/{booking_id}:
get:
 summary: Detalha reserva
 parameters:
 - name: booking_id
 in: path
 required: true
 schema:
 type: integer
 responses:
 '200':
 description: Detalhes da reserva.
 content:
 application/json:
 schema:
 $ref: '#/components/schemas/BookingDetail'
 '404':
 description: Reserva não encontrada.
delete:
 summary: Cancela reserva
 parameters:
 - name: booking_id
 in: path
 required: true
 schema:
 type: integer
 responses:
 '204':
 description: Reserva cancelada.
 '403':
 description: Não é possível cancelar.

```

```

yaml
CopyEdit
- **Schemas** (`components/schemas`):
 - `Room`: `{ room_id: integer, max_participants: integer, created_at: string(date-time), updated_at: string(date-time) }`
 - `RoomCreate`: `{ max_participants: integer (min 1), optional_name?: string, metadata?: object }`
 - `RoomDetail`: inclui `Room` + flags como `has_future_booking: boolean`.
 - `RoomParticipants`: `{ room_id: integer, participant_count: integer, participants?: [...] }`

```

```

- `RoomAvailability`: `{ room_id: integer, max_participants: integer }`
- `BookingCreate`: `{ start_time: string(date-time), end_time: string(date-time), room_id?: integer }`
- `BookingResponse`: `{ booking_id: integer, room_id: integer, password: string, start_time: string, end_time: string }`
- `BookingDetail`: `{ booking_id: integer, room_id: integer, start_time: string, end_time: string, status: string, password?: string }`
- **Geração automática**: usar drf-spectacular ou drf-yasg para manter documentação sincronizada com código :contentReference[oaicite:5]{index=5}.

```

## ## 7. Observações Finais

- \*\*Manter documentação atualizada\*\*: sempre que houver alteração nos endpoints ou payloads, atualizar OpenAPI/Swagger e exemplos de cURL/Postman.
- \*\*Onboarding de novos desenvolvedores\*\*: incluir neste documento (ou wiki) instruções de como rodar localmente via Docker Compose:

```

```yaml
version: '3.8'
services:
  db:
    image: postgres:13
    environment:
      POSTGRES_USER: suroga
      POSTGRES_PASSWORD: <senha>
      POSTGRES_DB: suroga_db
    volumes:
      - pgdata:/var/lib/postgresql/data
  booking-api:
    build:
      context: .
      dockerfile: Dockerfile
    environment:
      DATABASE_URL: postgres://suroga:<senha>@db:5432/suroga_db
      JANUS_URL: http://janus:8088/janus
      JANUS_KEY: <chave>
      DJANGO_SETTINGS_MODULE: myproject.settings.dev
    ports:
      - "8000:8000"
    depends_on:
      - db
  # opcional: serviço Janus de teste ou mock Janus
  redis:
    image: redis:6
volumes:
  pgdata:

```

- **Integração futura:** quando incorporar a API de booking dentro do container do Suroga API, ajustar Dockerfile ou base image para incluir o código de booking, unificar variáveis de ambiente, rotas e autenticação compartilhada.
- **Pipeline monolítico ou múltiplos jobs?** Se no futuro Suroga API tiver múltiplos microserviços (booking, auth, analytics), pode-se usar multi-stage pipeline que constrói imagens independentes e as deploya separadamente, ou usar monorepo com jobs específicos para cada serviço.
- **Segurança em produção:** certificar-se de que as variáveis sensíveis (SECRET_KEY, tokens Janus) sejam armazenadas em Azure Key Vault ou equivalente, e injetadas com processo seguro no deployment.

- **Monitoramento e alertas:** configurar Application Insights (ou equivalente) para coletar métricas de uso (requests por endpoint), latência, erros 5xx, saúde do container, e alertar a equipe de operações em caso de degradação.
-

A seguir, uma proposta de como estruturar e redigir as seções finais da documentação do serviço WebRTC streaming do Suroga, focando em:

- **6. Front-end do serviço:** o que deve conter, como apresentar a integração com Janus/WebRTC, exemplos de componentes, UX, fluxo de mídia.
- **7. Integração:** como integrar este serviço com outras partes do ecossistema Suroga (apps web/mobile, API de booking, autenticação, monitoramento, CI/CD, deployment, observabilidade etc.).

Em cada item, indico tópicos a cobrir, exemplos ou considerações e sugestões de conteúdo. Ajuste conforme a stack (frameworks, padrões de UI/UX e requisitos de projeto) adotados pela sua equipe.

6. Front-end do Serviço

Esta seção descreve como o cliente (app web ou webview em mobile) consome o serviço WebRTC, interage com Janus Gateway e oferece interface ao usuário para iniciar/participar de streaming 1-1.

6.1 Objetivos e escopo

- **Objetivo:** apresentar de forma clara aos desenvolvedores front-end como incorporar o streaming WebRTC usando Janus, garantindo usabilidade, performance e segurança.
- **Público-alvo:** times de frontend (web, mobile via WebView ou híbrido), equipes de UI/UX e suporte.
- **Escopo:**
 - Fluxos principais: iniciar streaming 1-1 (host inicia, convidado ingressa), compartilhamento de áudio/vídeo, possivelmente screenshare.
 - UI de controle: botões de iniciar/parar câmera, microfone, screenshare, indicador de conexão, status de participantes.
 - Gestão de erros e estados: exibir mensagens claras quando falha de mídia ou de conexão.
 - Compatibilidade de navegadores: identificar requisitos mínimos (Chrome, Firefox, Safari etc.) e fallback ou alertas quando não disponível.
 - Responsividade: interface adaptada a desktop e mobile (portrait e landscape).
 - Integração com autenticação e autorização (token para acesso ao streaming).
 - Qualidade de experiência (UX): latência, indicadores de rede, ajustes automáticos de bitrate/resolução se necessário.

6.2 Tecnologias e dependências

- **Biblioteca Janus JS Client:**

- Incluir link para a versão utilizada do Janus JS (por exemplo, a partir do CDN ou bundler). Documentar versão exata e commit/tag, para rastreabilidade.
- Explicar como inicializar: `Janus.init({debug: [...]})`, criar sessão, anexar plugin “videoroom” ou plugin adequado.

- **Adapter.js:**

- Garantir que o adapter WebRTC seja incluído para compatibilidade cross-browser.

- **Frameworks de UI:**

- Se for React/Vue/Angular/etc., encapsular a lógica Janus em hooks ou services. Exemplo: em React, criar hook `useJanusSession()`, que gerencia conexão, evento de estado e expõe métodos (`joinRoom`, `leaveRoom`, `publishStream`, `subscribeStream`).
- UI de componentes: usar componentes reutilizáveis para vídeo local (preview), vídeo remoto, controles (botões mutar, encerrar, compartilhar tela).
- Estilos: responsivos (CSS flex/grid), indicadores de status (ícones de microfone, câmera), mensagens de alerta (modals ou banners).

- **Comunicação com backend:**

- Endpoints para obter credenciais de sala: antes de iniciar Janus, front-end faz chamada à API de booking para obter `room_id`, `password` e possivelmente token (se houver assinatura de evento).
- Configurações: variável de ambiente ou configuração em runtime para `JANUS_SERVER_URL`, `API_BASE_URL`, chaves de autenticação.

- **Mecanismos de signaling:**

- Janus usa WebSocket ou HTTP para sinalização. Front-end deve construir a URL do WebSocket (e.g., `wss://janus.suroga.com/janus`) com eventuais parâmetros (token, sala, apisecret? Normalmente apisecret não enviado pelo front-end, mas service backend fornece secret temporário ou token JWT que Janus valida).
- Documentar como gerar e passar credenciais de forma segura: não expor `apisecret` no front-end; usar backend para criar sessões Janus ou tokens temporários, se Janus estiver configurado para token-based auth.

- **Media capture:**

- Chamar `navigator.mediaDevices.getUserMedia({ audio: true, video: [...] })` para câmera/microfone.
- Para screenshare: `navigator.mediaDevices.getDisplayMedia(...)`. Se precisar de fallback legado, documentar que navegadores antigos podem não suportar, e exibir mensagem de “não suportado”.

- **Gestão de múltiplos fluxos:**
 - Em 1-1, geralmente apenas um fluxo de publicação (host) e um de subscrição (guest). Mas planejar lógica genérica: array de streams remotos, caso no futuro suporte multi-party.
- **Qualidade adaptativa:**
 - Se desejado, usar constraints dinâmicos (p.ex., mudar resolução ou frameRate dependendo de largura de banda). Poder expor opção “ajustar qualidade automaticamente”.
 - Medir estatísticas WebRTC (`getStats`) e exibir ao usuário indicadores (jitter, packet loss) ou ajustar parâmetros. Documentar como coletar e usar essas métricas.
- **Métricas e logs do front-end:**
 - Incluir event tracking: tempo de conexão, falha em captura, erros de ICE, desconexões abruptas.
 - Enviar logs para sistema de observabilidade do Suroga (via API de logging ou analytics), para diagnosticar problemas no campo.
- **Internacionalização (i18n):**
 - Se a UI for multilíngue, preparar textos de botões e mensagens de erro via sistema de i18n.
- **Acessibilidade (a11y):**
 - Garantir que controles tenham labels, ícones com alt/texto, foco para teclado e compatibilidade com leitores de tela.

6.3 Fluxo de uso típico

1. Página/Componente de “Iniciar Conferência”:

- Usuário autenticado clica em “Iniciar chamada”.
- Front-end chama API de booking (`POST /bookings/`) para alocar ou criar sala. Recebe `room_id`, `password`, horário permitido etc.
- Front-end inicializa Janus: `Janus.init({debug: [...]})`. Após callback de `init`, cria sessão:

```
js
CopyEdit
const janus = new Janus({
  server: JANUS_SERVER_URL, // array ou string
  success: () => { /* sessão criada */ },
  error: (err) => { /* tratar erro de conexão */ },
  destroyed: () => { /* cleanup */ }
});
```

- Anexa plugin Videoroom:

```
js
CopyEdit
```

```

janus.attach({
  plugin: "janus.plugin.videoroom",
  success: (pluginHandle) => { /* armazenar handle */ },
  error: (err) => { /* erro de attach */ },
  consentDialog: (on) => { /* mostrar overlay se needed */ },
  webrtcState: (on) => { /* indicar se WebRTC conectou */ },
  mediaState: (medium, on) => { /* indicar mute/unmute local */ },
  onmessage: (msg, jsep) => { /* processar mensagens Janus */ },
  onlocalstream: (stream) => { /* exibir preview local */ },
  onremotestream: (stream) => { /* exibir vídeo remoto */ },
  oncleanup: () => { /* limpar vídeo */ }
});

```

- Após attach, enviar mensagem de “join” como publisher:

```

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

```

- Gerar offer via `pluginHandle.createOffer(...)`, enviar via `pluginHandle.send({ message: { request: "configure", audio: true, video: true }, jsep: jsep })`.
- Exibir stream local e aguardar subscriber ingressar.

2. Ingressar como participante (subscriber):

- Front-end do outro usuário lista salas disponíveis, escolhe sala ou usa link compartilhado. Após receber `room_id` e `password`, inicializa Janus e attach plugin Videoroom.
- Envia “join” como subscriber:

```

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

```

- Recebe jsep do publisher e cria `pluginHandle.createAnswer(...)`, envia para Janus e exibe vídeo do publisher.

3. Controles em tempo real:

- Botões “mudar câmera on/off”, “mudo microfone”, “encerrar chamada”.
- Enviar mensagens via Janus: `pluginHandle.send({ message: { request: "configure", audio: false } })` para mutar microfone local, etc.
- Lidar com eventos de desconexão inesperada: detectar `webrtcState` off, tentar reconectar ou notificar usuário.

4. Encerrar sessão:

- Quando o usuário clicar “Encerrar”, chamar `pluginHandle.send({ message: { request: "leave" } })` e depois `janus.destroy()` ou `pluginHandle.hangup()`, e limpar streams (`stopAllTracks`).
- Se for host, possivelmente notificar backend para marcar sala como finalizada.

5. Screenshare (opcional):

- Botão “Compartilhar Tela”: ao clicar, chamar `navigator.mediaDevices.getDisplayMedia(...)`, obter `MediaStream`, e renegociar WebRTC: normalmente usar `pluginHandle.createOffer({ media: { video: stream } })` ou método específico conforme Janus client. Documentar exemplo de código para renegociação de track de vídeo.
- Cuidar de troca de streams: parar track anterior, adicionar track de tela, renegociar, e vice-versa quando parar screenshare.

6. Mensagens de erro e fallback:

- Caso `getUserMedia` falhe (permissão negada ou dispositivo não disponível), exibir modal informando: “Não foi possível acessar câmera/microfone. Verifique permissões.”.
- Se ICE não conectar, exibir “Falha na conexão de rede. Tente em outra rede ou verifique firewall.”.
- Timeouts de Janus: se não conseguir attach ou join em X segundos, mostrar erro e sugerir recarregar.

7. UI/UX:

- Indicadores visuais de “ligado/desligado” para microfone e câmera.
- Indicador de “conectando”, “conectado”, “desconectado”.
- Ao compartilhar link de convite, criar botão “Copiar link” que inclui `room_id` e possivelmente token curto para entrada.
- Layout responsivo: vídeo local em canto, vídeo remoto em destaque.
- Se for mobile, considerar orientações sobre uso de câmera frontal/traseira, permissão de hardware.
- Considerar uso de APIs nativas para notificações (e.g., “chamada recebida” em mobile web) se aplicável.

6.4 Segurança no Front-end

- **HTTPS obrigatório:** WebRTC requer contexto seguro. Garantir que a aplicação esteja servida por HTTPS.

- **Autenticação/Token:**
 - Não expor JANUS_APISecret no front-end. Backend deve gerar credenciais temporárias (por exemplo, um token JWT que Janus valida) ou usar Janus plugin de autenticação/token. Documentar como o gateway Janus está configurado para validar credenciais vindas do front-end (via JWT ou outro mecanismo).
 - Front-end envia token no parâmetro de conexão (por exemplo, na URL do WebSocket ou em mensagem inicial). Documentar esse fluxo.
- **CORS:** configurar no servidor Janus (se houver proxy) e na API para permitir chamadas do domínio da aplicação.
- **Proteção contra abuso:** limitar tentativas de criar/join de salas inválidas. O front-end deve obter room_id via backend autorizado; não permitir que usuário tente arbitrary room_id sem permissão.
- **Proteção de mídia:** se for sensível (conferência privada), considerar criptografia ponta-a-ponta ou DTLS-SRTP padrão do WebRTC. Documentar que Janus usa DTLS.
- **Headers de segurança:** configurar Content-Security-Policy, X-Frame-Options, etc., principalmente se a aplicação for carregada em iframes.

6.5 Testes e Qualidade

- **Testes manuais e automatizados:**
 - Testar em diferentes navegadores e dispositivos (desktop e mobile).
 - Simular condições de rede ruim (throttling) para verificar comportamento de qualidade adaptativa e feedback ao usuário.
 - Testar casos de permissão negada (câmera/mic) e screenshare.
 - Testar reconexão (rede muda, queda temporária).
- **Monitoramento de erros no front-end:**
 - Capturar erros JS (Sentry ou equivalente) para diagnosticar falhas no campo.
 - Capturar métricas de performance de carregamento do SDK Janus e tempo até conexão WebRTC estabelecida.
- **Documentação para desenvolvedores front-end:**
 - Incluir README ou wiki com instruções passo-a-passo para integrar a biblioteca Janus no projeto específico (ex.: instalação via npm/CDN, configuração, inicialização, exemplos de código).
 - Exemplos de componentes prontos: ex.: React component <VideoChat roomId=... password=... onEnd={...} />.
 - Padrões de estilo de código, linting, formatação para manter consistência.

6.6 Observabilidade no Cliente

- **Logs de evento:**
 - Registrar eventos-chave: “requestCreateSession”, “attachedPlugin”, “joinedRoom”, “onlocalstream”, “onremotestream”, “hangup”, “errorXYZ”.
 - Enviar logs para backend ou serviço de logging/analytics para análise de uso e problemas.
 - **Métricas de uso:**
 - Quantas sessões iniciadas, duração média, falhas de conexão, uso de screenshare.
 - **Feedback ao usuário:**
 - Mostrar indicadores de qualidade (e.g., “boa conexão”, “latência alta, qualidade reduzida”).
-

7. Integração

Nesta seção, documentamos como o serviço WebRTC streaming se integra ao restante do ecossistema Suroga: backend booking/API, autenticação, orquestração de containers, monitoramento, mobile apps, CI/CD e deployment.

7.1 Integração com API de Booking e Backend

- **Fluxo de reserva → streaming:**
 1. Usuário solicita via app ou web “Iniciar chamada”: front-end chama API Booking (`/bookings/`) para obter room_id e senha.
 2. Backend registra reserva, possivelmente agenda horário, e devolve credenciais.
 3. Front-end: usa credenciais recebidas para iniciar Janus session e join na sala.
- **Validação de permissão:**
 1. Backend deve verificar se o usuário tem direito de ingressar naquela sala (verificar se reservou ou foi convidado).
 2. Ao ingressar, front-end envia token de autenticação e room_id/password, backend pode em alguns casos atuar como proxy para Janus, validando sessão antes de retornar parâmetros para front-end.
- **Webhook ou sinalização de estado:**
 1. Quando sessão inicia ou termina, front-end ou Janus pode informar backend via webhook ou API: ex.: “chamada iniciada às X”, “chamada encerrada às Y”. Isso ajuda em logs, billing ou auditoria.
- **Token JWT para Janus:**

1. Em vez de expor JANUS_KEY, backend pode gerar token JWT para cada sessão, que o Janus valida (usando plugin JWT). Documentar fluxo de geração e validação no Janus Gateway:
 - Backend cria JWT contendo room, user, expiração, assina com segredo compart.
 - Front-end recebe JWT e passa ao Janus ao criar sessão ou join.
 - Janus valida token e permite join sem revelar apisecret ao cliente.
- **Gerenciamento de sessões ativas:**
 1. Backend pode manter registro de sessões ativas (armazenar sessionId Janus e usuário), para permitir funcionalidades administrativas (forçar encerramento, monitorar uso).
- **Fallback e erro de integração:**
 1. Se backend ou Janus estiver indisponível, front-end exibe mensagem amigável e backend aciona alertas.
- **API de analytics:**
 1. Quando final da chamada, front-end ou Janus envia métricas (via backend) de qualidade (jitter, packet loss, bitrate), para análise de experiência do usuário.

7.2 Integração com Apps Móveis / Nativo

- **WebView ou SDK nativo:**
 - Se app móvel for híbrido, usar WebView carregando front-end web; garantir permissões de câmera/mic no manifest (Android) e Info.plist (iOS).
 - Se for app nativo (React Native, Flutter, Swift/Java), usar bibliotecas WebRTC nativas ou Janus Native SDK:
 - Por exemplo, em React Native, usar react-native-webrtc e implementações customizadas para Janus signaling via WebSocket/HTTP.
 - Documentar exemplos de como integrar com Janus no ambiente nativo: criação de PeerConnection, troca de SDP via signaling do backend ou diretamente para Janus.
- **Permissões de runtime:**
 - Documentar como solicitar permissão de câmera e microfone no Android/iOS, como tratar rejeição de permissão no app.
- **Captura de tela em mobile:**
 - Screenshare em mobile é mais complexo: iOS e Android exigem APIs específicas (ReplayKit no iOS, MediaProjection API no Android). Documentar se desejado e como integrar (provavelmente via código nativo e enviar track para Janus).
- **Notificações push:**

- Se for chamada agendada, backend pode enviar push notification ao usuário convidado para ingressar na hora. Documentar fluxo: backend agenda notificação, app móvel recebe, abre tela de call, inicia front-end streaming.
- **Qualidade e adaptividade:**
 - Mobile pode ter redes instáveis; implementar estratégias de re-conexão, ajustes automáticos de bitrate/resolução.
- **Testes em dispositivos reais:**
 - Incluir instruções para testar em diferentes modelos, versões de OS, garantir compatibilidade WebRTC.

7.3 Autenticação e Autorização

- **Single Sign-On (SSO):**
 - Se Suroga possui sistema central de autenticação, front-end faz login e backend fornece token para APIs de booking e para criar sessões Janus (via JWT).
- **Tokens temporários:**
 - Para Janus, usar tokens de curta duração, com escopo limitado (ex.: só para aquela sala e usuário específico). Expiração curta para reduzir risco de uso indevido.
- **Controle de acesso:**
 - Backend verifica permissão de usuário antes de entregar credenciais de sala. Front-end deve recusar acesso se token inválido.
- **Revogação de sessão:**
 - Se usuário for removido ou call for cancelada, backend pode notificar front-end para sair da sala (via WebSocket de controle) ou forçar Janus a destruir sessão.
- **Proteção de endpoints:**
 - Documentar cabeçalhos necessários, scopes de token e possíveis erros de autorização que front-end deve tratar (401, 403).

7.4 CI/CD e Deployment

- **Pipeline de build para front-end:**
 - Se UI for Single Page App (React/Vue), pipeline que:
 - Executa lint/tests,
 - Gera bundle otimizado (minificação, tree-shaking),
 - Publica em CDN ou container web (ex.: nginx container).
 - Integrar com pipeline geral do Suroga API, possivelmente multi-stage: build front-end, build backend booking API, build containers de proxy/Janus config.
- **Pipeline para Janus Gateway:**

- Embora Janus não seja “front-end”, sua configuração (jcfg) e scripts de deploy podem estar versionados e automatizados: pipeline que aplica configurações em instâncias VM ou containers Kubernetes.
- **Orquestração de containers:**
 - Definir manifestos Kubernetes (Deployments, Services, Ingress) para:
 - Front-end (servir estáticos ou SPA),
 - Backend booking API,
 - Proxy reverso (NGINX) para Janus (WebSocket/HTTPS),
 - Janus Gateway instances (como Deployment/StatefulSet),
 - Servidor TURN (se necessário) em cluster,
 - Monitoramento (Prometheus exporters, logs),
 - Redis/Celery para tarefas assíncronas,
 - Banco de dados.
 - Documentar como versionar e aplicar alterações via GitOps (FluxCD/ArgoCD) ou pipelines Azure DevOps.
- **Configuração de ambientes:**
 - Variáveis de ambiente e secrets (via Azure Key Vault ou Kubernetes Secrets) para URLs, chaves, tokens, credenciais de DB.
- **Health checks e readiness/liveness probes:**
 - Front-end: probe em `/health/`.
 - Janus: sondas que verificam se Janus ainda responde (por exemplo, via API HTTP ou via WebSocket ping).
- **Monitoramento e alertas:**
 - Configurar dashboards (Grafana) com métricas de CPU/memória de pods/VMs, latência de WebRTC, número de sessões ativas, erros de conexões.
 - Alertas para quedas de instâncias, latência alta, erros 5xx do backend, falhas de Janus.
- **Rollback e versionamento:**
 - Estratégias de deploy com versionamento de imagens e possibilidade de rollback se versão nova apresentar regressão no front-end ou na API.
- **Testes de integração end-to-end:**
 - Automatizar testes que iniciam sessão WebRTC real com Janus de teste, para validar que deploy não quebrou fluxos de streaming.
- **Documentar procedimentos de upgrade:**

- Como atualizar versão do Janus (compilação), rotacionar instâncias sem downtime, manter pool de instâncias para failover.
- Como atualizar front-end de streaming sem impactar usuários ativos: feature flags ou deploy paralelo.

7.5 Monitoramento e Observabilidade

- **Logs agregados:**
 - Front-end: logs JS de erros enviados a sistema de logging (Sentry).
 - Backend: logs de booking API, integração com Janus (criação/edição/deleção de salas).
 - Janus logs: coletar logs de gateway, erros de ICE, mensagens de plugin. Enviar para sistema central (ELK, Azure Monitor).
- **Métricas WebRTC:**
 - Coletar stats de chamadas via Janus (via plugin ou API): métricas de bitrate, jitter, packet loss, CPU do servidor.
 - Expor dashboards para ops e produto acompanhar qualidade de serviço.
- **Alertas:**
 - Definir limites para métricas críticas (número de sessões ativas, utilização de CPU, erros repetidos).
 - Alertar time de SRE se degradação de qualidade (e.g., aumento de packet loss ou falhas de conexão).
- **Relatórios periódicos:**
 - Relatório semanal/mensal de uso de streaming: quantas sessões, duração média, picos de uso, regiões mais usadas.
 - Usar para planejamento de capacidade (escalabilidade e custo).

7.6 Documentação e Onboarding

- **Documentação para desenvolvedores:**
 - Repositório ou wiki com instruções completas: como rodar local (Docker Compose), como configurar variáveis de ambiente, como executar testes, como usar Janus localmente para desenvolvimento.
 - Exemplos de front-end: componente de vídeo, hooks/serviços de Janus, exemplos de chamada de API backend.
 - Padrões de codificação e organização de pastas.
- **Documentação para operadores:**
 - Como provisionar infraestrutura: VM ou Kubernetes para Janus Gateway e TURN, configurações de firewall, NSG no Azure, load balancer.

- Como configurar SSL/TLS para WebSocket seguro e HTTPS do front-end.
- Como rodar backups de configurações, atualizar Janus, rotacionar certificados.
- **Documentação para QA:**
 - Casos de teste manuais e automatizados: cenários de fluxo feliz e de falhas (rede instável, permissões negadas).
 - Instruções para usar ferramentas de debug WebRTC (about:webrtc-internals, chrome://webrtc-internals).
- **Guias de troubleshooting:**
 - Cenários comuns (ice fail, mismatch de codec, problema de firewall/portas UDP bloqueadas) e como diagnosticar (logs Janus, logs browser).
 - Comandos ou scripts para verificar portas de UDP abertas, NAT, TURN configurado.

7.7 Segurança e Compliance

- **Segurança de rede:**
 - Garantir NSG/Azure Firewall permita apenas portas necessárias (80/443 TCP, 10000-20000 UDP para RTP).
 - Habilitar DDoS protection se houver tráfego público.
- **Proteção de dados:**
 - Não armazenar gravações (se aplicável) sem consentimento ou compliance.
 - Se gravação for permitida, armazenar em local seguro, criptografar em descanso e em trânsito.
- **Privacidade:**
 - Informar usuários sobre captura de dados de mídia, consentimento para uso de câmera/microfone.
- **Auditoria:**
 - Manter logs de quem criou/participou de chamadas, horários, IPs (respeitando GDPR se aplicável).
- **Atualizações de biblioteca:**
 - Monitorar vulnerabilidades em dependências front-end (Janus JS, adapter.js) e backend.
- **CORS e CSP:**
 - Configurar políticas estritas de Content-Security-Policy, permitindo apenas origens confiáveis para scripts e frames.
- **Penetration tests:**
 - Testar endpoints e fluxo de WebRTC contra ataques comuns (DoS, injeção, CSRF, XSS).

7.8 Evolução Futuras

- **Multi-party e escalabilidade:**
 - Embora hoje seja 1-1, planejar se futuramente deseja suportar múltiplos participantes (videoroom com vários publishers/subscribers). Documentar possibilidade de ampliar Janus SFU para multi-party.
- **Transcoding / livestreaming em broadcast:**
 - Futuro suporte para streaming 1 para N (ex.: transmissão ao vivo), integrando com CDN ou serviços de streaming adaptativo.
- **Integração com outras plataformas de streaming:**
 - Já mencionado que Suroga é agnóstico: documentar como integrar com flows de RTMP ou HLS gerados por Janus (plugins ou pipeline externa).
- **Mobile nativo e PWA:**
 - Evoluir front-end para PWA, notificações, background sync, capacidades offline (talvez salvar mensagens de chat se implementado).
- **Analytics avançado:**
 - Machine learning para detectar problemas de qualidade e sugerir ajustes automáticos ao usuário.
- **APIs de administração:**
 - Dashboard administrativo para monitorar sessões ativas em tempo real, forçar desconexões, enviar mensagens de broadcast aos participantes etc.

Estrutura sugerida para as seções 6 e 7 no documento

Abaixo, um esqueleto em Markdown para incluir na documentação, adaptável conforme estilo do documento todo:

```
markdown
CopyEdit
## 6. Front-end do Serviço
```

Nesta seção detalhamos como integrar o streaming WebRTC (via Janus) no cliente web/mobile, cobrindo arquitetura, fluxos, exemplos de código e boas práticas.

```
### 6.1 Objetivos e escopo
- Descrever responsabilidades do front-end.
- Listar fluxos principais (iniciar/ingressar em chamada, screenshare, encerramento).
- Definir requisitos de compatibilidade e UX.
```

```
### 6.2 Tecnologias e dependências
- Janus JS Client: versão, inicialização e configuração.
- Adapter.js para cross-browser.
- Frameworks UI (React/Vue/Angular/etc.) ou Vanilla JS.
- Comunicação com backend (endpoints de booking, autenticação).
- Configurações (URLs, tokens, variáveis de ambiente).
```

6.3 Arquitetura do front-end

- Diagrama de componentes ou módulos:
 - Serviço Janus: init, session management, plugin handles.
 - Componentes de UI de vídeo local e remoto.
 - Serviço de chamadas à API booking/autenticação.
 - Serviço de logs e métricas no cliente.
- Fluxo de dados:
 1. Obter credenciais de sala (room_id, password) via API.
 2. Inicializar Janus e attach plugin.
 3. join room como publisher/subscriber.
 4. Exibir streams e controles.
 5. Encerrar e cleanup.
- Fluxo de erros e reconexão.

6.4 Exemplos de código

- Trechos de JS (ou TypeScript) ilustrando:

- Inicialização Janus:

```
```js
Janus.init({ debug: ["log","warn","error"], callback: onJanusInit });
```
- Criação de sessão e attach plugin:

```
```js
const janus = new Janus({...});
janus.attach({...});
```
- Fluxo de join/publish:

```
```js
handle.send({ message: { request: "join", room: roomId,ptype: "publisher", secret: password } });
// createOffer, send configure...
```
- Fluxo de subscribe:

```
```js
handle.send({ message: { request: "join", room: roomId, ptype: "subscriber", feed: publisherId } });
```
- Tratamento de onlocalstream/onremotestream:

```
```js
onlocalstream: stream => attachMediaStream(localVideoElem, stream);
onremotestream: stream => attachMediaStream(remoteVideoElem, stream);
```
- Exemplo de renegociação para screenshare:

```
```js
async function startScreenShare() {
  const screenStream = await navigator.mediaDevices.getDisplayMedia({ video: true });
  // trocar track no PeerConnection via Janus API
  handle.createOffer({ media: { video: screenStream.getVideoTracks()[0] }, success: ... });
}
```
- Exemplos de wrappers em frameworks:
 - React hook `useJanus()` ou Vue composable.
 - Serviço Axios/fetch para chamadas à API booking.
- Exemplos de tratamento de permissões (câmera/microfone) e mensagens de UI.

6.5 UX e UI/Design

- Layouts responsivos: vídeo local em picture-in-picture, vídeo remoto em destaque.
- Controles de mídia: ícones para mutar, desligar câmera, finalizar chamada, iniciar screenshare.
- Indicadores de status: “Conectando...”, “Conectado”, “Reconectando...”, “Erro de rede”.

- Mensagens de erro amigáveis: permissão negada, falha de ICE, sala não encontrada, horário fora de janela de reserva.
- Fluxo de convite: gerar link compartilhável que encapsula room_id (e possivelmente token curto), exibição de QR code em mobile.

6.6 Segurança no front-end

- Uso obrigatório de HTTPS.
- Mecanismo de obtenção de tokens JWT ou credenciais temporárias para Janus, sem expor apisecret.
- Configuração de CORS e CSP no servidor.
- Evitar exposição de senhas em listagens públicas.
- Sanitização de inputs (e.g., campos de chat, se houver).
- Tratamento de dados sensíveis no localStorage/sessionStorage (evitar armazenar token indefinidamente).

6.7 Testes e qualidade

- Testes unitários de módulos de lógica Janus (mock de Janus JS).
- Testes de integração (E2E) simulando fluxo de chamada real (usando instância de Janus de teste).
- Testes manuais em múltiplos navegadores/dispositivos (lista de cenários).
- Ferramentas de debug WebRTC (chrome://webrtc-internals).
- Monitoramento de erros (Sentry ou equivalente).

6.8 Observabilidade e métricas

- Coleta de logs de eventos no front-end.
- Envio de dados de performance: tempo para conectar, latência de ICE, qualidade de mídia.
- Dashboard para analisar problemas recorrentes e melhorar UX.
- Alertas automáticos em caso de falhas recorrentes.

7. Integração

Esta seção aborda como o serviço WebRTC se encaixa no ecossistema Suroga, incluindo backend, apps móveis, infraestrutura e processos operacionais.

7.1 Integração com backend de booking/autenticação

- Descrição do fluxo reserva → streaming: endpoints, payloads, credenciais.
- Geração de tokens JWT para Janus: algoritmo, claims (room, usuário, expiração).
- Webhooks ou callbacks: notificar backend quando sessão inicia/finaliza.
- Políticas de autorização: quem pode criar/join sala, regras de expiração.
- Exemplo de diagrama de sequência (sequence diagram) mostrando chamadas entre front-end, backend booking, Janus e possíveis serviços auxiliares (e.g., logging, analytics).

7.2 Integração com apps móveis/nativo

- Estratégia WebView vs SDK nativo: prós e contras.
- Permissões e APIs específicas de mobile para WebRTC e screen capture.
- Notificações push para agendamento de chamadas.
- Exemplo de código ou referências a projetos que usem Janus em React Native ou Flutter.
- Considerações de performance e UX mobile (layout, gestão de rede instável, economia de bateria).

7.3 Orquestração de containers e infraestrutura

- Visão geral da arquitetura em produção:
 - Front-end servindo estáticos (CDN ou container web).
 - Backend booking API.
 - Proxy reverso (NGINX) para Janus (WS/TLS) e para API.
 - Instâncias Janus Gateway (em VMs Azure ou containers Kubernetes).
 - Servidor(s) TURN/STUN se necessário.
 - Banco de dados (PostgreSQL), cache (Redis) e brokers (RabbitMQ/Celery).

- Descrever deployment no Azure (App Service, AKS, Container Instances, VMs).
- Configuração de NSG, balanceador de carga, certificados TLS (Let's Encrypt ou Azure Key Vault).
- Health checks, readiness/liveness probes e auto-scale.
- Pipeline CI/CD: build de imagens, push para registry, deploy automatizado.
- Diagramas de infraestrutura (network diagram) mostrando sub-redes, segurança e componentes.

7.4 Monitoramento, logs e alertas

- Backend: logs de booking, estatísticas de uso.
- Janus: logs de gateway, métricas de WebRTC, uso de CPU/memória.
- Front-end: erros JS, métricas de performance.
- Ferramentas: Azure Monitor, Prometheus/Grafana, ELK, Sentry.
- Alertas configurados para falhas críticas ou degradação de qualidade.

7.5 Segurança e compliance

- Políticas de rede e firewall (portas UDP/TCP).
- Proteção DDoS, WAF se aplicável.
- Proteção de dados (senhas de sala, gravações).
- Logs de auditoria (quem criou, ingressou e terminou chamadas, horários e IPs).
- Políticas de retenção de dados, GDPR e privacidade.
- Atualizações e patching (dependências e SO).
- Revisões de segurança periódicas.

7.6 Testes de integração end-to-end

- Automatizar testes que simulam fluxo completo: reserva → front-end inicia streaming → participante ingressa → media flui → encerramento.
- Ambiente de staging com Janus de teste.
- Testes de carga para verificar limites de sessões simultâneas e dimensionar infraestrutura.
- Ferramentas de teste de WebRTC (SIPp, test suites customizadas).

7.7 Documentação operacional e de desenvolvimento

- Guia de onboarding: como rodar local (Docker Compose) e conectar a Janus local ou de teste.
- Documentação das variáveis de ambiente e configurações necessárias.
- Exemplos de troubleshooting: verificações de firewall, logs de Janus, logs de front-end.
- Procedimentos de upgrade/rollback para Janus e para front-end/backend.
- Padrões de versionamento de API (ex.: v1, v2) e compatibilidade retroativa.
- Checklist de pré-lançamento: testes, revisão de segurança, revisão de performance.

7.8 Roadmap e futuras evoluções

- Suporte a multi-party (SFU para salas com mais de 2 participantes).
- Streaming 1→N via CDN ou HLS, integração com serviços de broadcast.
- Integração com analytics avançado ou AI (detecção de fala, transcrição em tempo real, blur de fundo).
- Suporte a gravação de chamadas e gerenciamento de arquivos gravados.
- Melhoria de qualidade adaptativa: algoritmos automáticos de ajuste de codec/resolução.
- Versão PWA ou aplicativo desktop (Electron) para maior integração.
- Integração com sistemas externos (CRM, LMS, plataformas de e-learning) via APIs.
- Internacionalização e customização de UI para diferentes clientes.

Exemplo de Trecho de Documentação (Markdown) para seção 6 e 7

```
```markdown
6. Front-end do Serviço
```

Nesta seção abordamos como integrar o streaming WebRTC (Janus) no cliente web/mobile.

#### ### 6.1 Objetivos e escopo

- Prover interface para chamadas 1-1 (host e participante).
- Permitir controle de mídia (áudio, vídeo, screen share).
- Oferecer experiência responsiva em desktop e mobile.
- Tratar autenticação, permissões e erros de forma amigável.

#### ### 6.2 Tecnologias e dependências

- \*\*Janus JS Client\*\*: versão X.Y.Z, import via CDN ou bundler (npm).
- \*\*Adapter.js\*\* para compatibilidade WebRTC cross-browser.
- \*\*Framework UI\*\*: React/Vue/Angular ou Vanilla JS.
- \*\*Comunicação com backend\*\*: endpoints REST (booking, auth).
- \*\*Configurações\*\*: variáveis de ambiente para URLs (API e Janus).
- \*\*Ferramentas de build\*\*: Webpack/Vite/etc.

#### ### 6.3 Arquitetura do front-end

- \*\*Módulo JanusService\*\*: encapsula init, session, attach, publish/subscribe.
- \*\*Componentes de UI\*\*:
  - `<VideoLocal />`: preview da câmera local.
  - `<VideoRemoto />`: exibe stream remoto.
  - `<Controls />`: botões de mutar, câmera, share screen, encerrar.
  - `<ConnectionStatus />`: mostra "Conectando...", "Conectado", "Desconectado".
- \*\*Fluxo de dados\*\*:
  1. Chamar API `/bookings/` e obter credenciais.
  2. Inicializar Janus: `Janus.init({ ... })`.
  3. Criar sessão e attach plugin Videoroom.
  4. Enviar "join" como publisher/subscriber.
  5. Gerenciar negociação via SDP/jsep.
  6. Exibir streams e controles.
  7. Encerrar e cleanup.

#### ### 6.4 Exemplos de código

```
```js
// Inicialização
Janus.init({ debug: ["warn", "error"], callback: () => {
    // Cria sessão Janus
    const janus = new Janus({
        server: JANUS_SERVER_URL,
        success: () => attachVideoRoomPlugin(janus),
        error: err => showError("Erro conexão WebRTC", err)
    });
} });

// Attach plugin Videoroom
function attachVideoRoomPlugin(janus) {
    janus.attach({
        plugin: "janus.plugin.videoroom",
        success: handle => {
            window.vrHandle = handle;
            joinAsPublisher(handle);
        },
        error: err => showError("Erro attach plugin", err),
        onmessage: onMessageFromJanus,
        onlocalstream: stream => attachMediaStream(localVideoElem, stream),
        onremotestream: stream => attachMediaStream(remoteVideoElem, stream),
        oncleanup: () => cleanupUI()
    });
}

// Join publisher
function joinAsPublisher(handle) {
    handle.send({
```

```

        message: {
          request: "join",
          room: roomId,
          ptype: "publisher",
          display: username,
          secret: password
        }
      });
      // criar offer e enviar configure...
    }
  
```

6.5 UX e UI/Design

- Layout responsivo: definir grid/flex para vídeo local e remoto.
- Botões claros: ícones de microfone, câmera, share screen, end call.
- Mensagens de status e de erro: permissão negada, falha ICE, sala indisponível.
- Feedback visual: indicadores de rede e qualidade.

6.6 Segurança no front-end

- Uso obrigatório de HTTPS.
- Tokens JWT para Janus em vez de expor apisecret.
- CORS e CSP configurados no servidor.
- Proteção contra XSS/CSRF se usar cookies.
- Sanitização de inputs (chat ou campos de nome).

6.7 Testes e qualidade

- Testes unitários de JanusService com mocks.
- Testes E2E simulando chamada real em instância de teste Janus.
- Testes manuais em navegadores distintos e mobile.
- Ferramentas de debug WebRTC: chrome://webrtc-internals.

6.8 Observabilidade

- Logs de eventos enviadas ao backend (via API de logging).
- Métricas de performance: tempo para conectar, packet loss.
- Dashboard de qualidade de chamadas.

7. Integração

7.1 Integração com Backend de Booking/Autenticação

- Fluxo reserva → streaming: chamadas ao backend, obtenção de room_id e senha.
- Geração de JWT para Janus: claims necessários (room, user, expiração).

- Webhooks/callbacks para notificar início/término de sessão.
- Verificação de permissão e revogação de acesso.

7.2 Integração com Apps Móveis

- Versão WebView vs SDK nativo (ex.: React Native WebRTC).
- Configuração de permissões (câmera, mic) nos manifests.
- Notificações push para convites.
- Adaptividade de rede em mobile, qualidade de mídia.

7.3 Infraestrutura e Deployment

- Arquitetura de containers: Front-end, Backend booking, Proxy Janus, Instâncias Janus, TURN, DB, Redis, Celery.
- Deploy em Azure: App Service, AKS, Container Registry.
- Configuração de NSG, load balancer, SSL/TLS, health checks.
- CI/CD: pipeline build front-end, backend, Janus configs; deploy automatizado.

7.4 Monitoramento e Observabilidade

- Coleta de logs (front, backend, Janus).
- Métricas de WebRTC (via Janus ou client) e dashboards.
- Alertas para falhas de streaming ou uso excessivo de recursos.

7.5 Segurança e Compliance

- Firewall (UDP RTP, TCP signaling), DDoS protection.
- Rotação de segredos, políticas de acesso, auditoria de chamadas.
- Armazenamento seguro de gravações (se houver).
- Conformidade GDPR e privacidade do usuário.

7.6 Testes de Integração End-to-End

- Automação de fluxo completo (reserva → join → publish/sub).
- Testes de carga para dimensionar instâncias Janus.
- Cenários de falhas e recuperação automática.

7.7 Documentação e Onboarding

- Guia de desenvolvimento local (Docker Compose, variáveis de ambiente).
- Como executar testes, debug WebRTC, debugging de Janus.
- Padrões de code style, revisão de PR, versionamento de API.

- Checklists de deploy e rollback.

7.8 Roadmap e Futuras Evoluções

- Suporte a multi-party (SFU), broadcast $1 \rightarrow N$, integração com CDN.
 - Recurso de gravação de chamadas e gerenciamento de arquivos.
 - Qualidade adaptativa avançada, AI/analytics em tempo real.
 - Suporte a PWA, desktop (Electron) ou apps nativos avançados.
 - Integração com sistemas externos (CRM, LMS, etc.).
-