Multi-Screen SRT Streaming System - Developer Documentation

Full-Stack Development for

OpenVideoWalls

Student Name
Sira Kongsiri

Project Overview

The Multi-Screen SRT Streaming System is a comprehensive real-time video streaming solution designed for multi-display installations. It supports synchronized playback across multiple screens with flexible layout configurations using the SRT (Secure Reliable Transport) protocol.

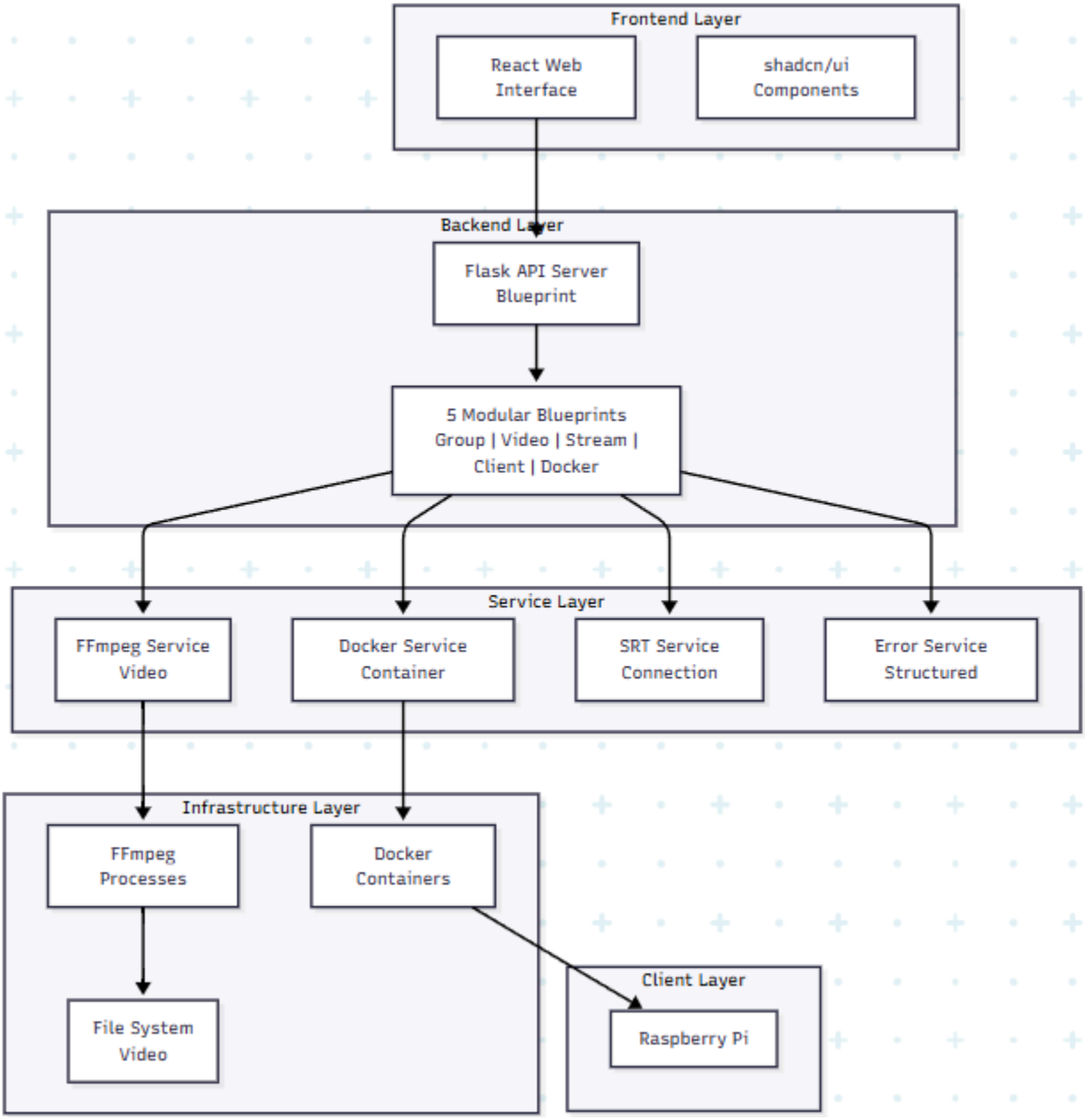
Key Features

- **Multi-Screen Support**: Control 2-16+ screens in various configurations
- **Flexible Layouts**: Horizontal, vertical, and grid arrangements
- **Real-Time Synchronization**: Low-latency SRT protocol with SEI timestamp embedding
- **Multiple Streaming Modes**: Multi-video and single video split modes
- **Professional Video Processing**: FFmpeg integration with format support
- **Automatic Client Management**: Discovery, registration, and smart assignment

Technology Stack

- **Backend**: Python 3.8+ with Flask
- **Frontend**: React 18 with TypeScript
- **Containerization**: Docker and Docker Compose
- **Video Processing**: FFmpeg with H.264 support
- **Streaming**: Simple Realtime Server (SRS) v5
- **UI Framework**: shadcn/ui with Tailwind CSS

System Architecture



This flowchart illustrates the comprehensive architecture of a professional video streaming solution designed for multi-display installations. The system is built using a layered approach that separates concerns and enables scalable, maintainable operations.

Architecture Layers Explained

Frontend Layer - User Interface & Control

The topmost layer provides the user-facing interface for system management:

- **React Web Interface:** A modern TypeScript-based single-page application built with Vite for fast development and optimized production builds
- **shadcn/ui Components:** Professional UI component library providing consistent, accessible design elements styled with Tailwind CSS for responsive layouts

This layer serves as the control center where administrators can manage streaming groups, monitor client devices, upload videos, and control streaming operations in real-time.

Backend Layer - API & Business Logic

The core application layer that processes requests and coordinates system operations:

- **Flask API Server:** Python-based REST API server that handles all client requests and coordinates system operations
- **5 Modular Blueprints:** Organized into specialized modules for clean separation of concerns:
    - **Group Management:** Creates and manages streaming groups
    - **Video Management:** Handles file uploads and video processing
    - **Stream Management:** Manages streaming operations and FFmpeg processes
    - **Client Management:** Handles device registration and assignment
    - **Docker Management:** Controls container lifecycle and health

This blueprint architecture ensures maintainable code with clear responsibilities and enables independent development of different system features.

Service Layer - Core Business Services

Specialized services that handle specific technical operations:

- **Docker Service:** Manages container lifecycle, port allocation, and health monitoring for streaming servers
- **FFmpeg Service:** Handles video processing, encoding, and stream generation with optimized command generation
- **SRT Service:** Manages SRT protocol connections, testing, and monitoring for reliable low-latency streaming
- **Error Service:** Provides structured error handling with categorized error codes and actionable solutions
-

This layer abstracts complex technical operations into reusable services, enabling consistent behavior across the application.

Infrastructure Layer - Core Technology Stack

The foundational technology components that power the streaming operations:

- **Docker Containers:** Isolated SRS (Simple Realtime Server) v5 instances that provide the actual streaming infrastructure
- **FFmpeg Processes:** Video encoding and processing engines that handle video splitting, format conversion, and stream generation
- **File System:** Persistent storage for uploaded video files with organized directory structures

This layer provides the raw computing power and storage needed for professional video streaming operations.

Client Layer - Display Devices

The endpoint devices that receive and display the streamed content:

- Client Player: Lightweight, optimized for playing the stream reliably and synchronously
- Raspberry Pi 4B Hardware: Cost-effective, reliable hardware platform for display endpoints

Data Flow & System Interactions

Request Flow (Top to Bottom)

1. **User Interaction:** Users interact with the React web interface to create groups, upload videos, or start streams
2. **API Processing:** Flask server receives requests and routes them to appropriate blueprints
3. **Service Coordination**: Blueprints call specialized services to perform technical operations
4. **Infrastructure Execution:** Services interact with Docker containers, FFmpeg processes, and file systems
5. **Client Communication:** Streaming servers deliver content to connected Raspberry Pi clients

Key System Benefits

Scalability: Each layer can be scaled independently based on demand

- Frontend can handle multiple concurrent administrators
- Backend can process numerous API requests simultaneously
- Services can manage multiple streaming operations
- Infrastructure can support dozens of concurrent streams

Maintainability: Clear separation of concerns makes the system easy to maintain and extend

- UI changes don't affect backend logic
- Service improvements don't require blueprint modifications
- Infrastructure upgrades don't impact application code

Reliability: Multi-layer error handling and recovery mechanisms

- Frontend validates user input before submission
- Backend provides comprehensive error responses
- Services implement automatic recovery for common failures
- Infrastructure includes health monitoring and automatic restart

Professional Quality: Enterprise-grade architecture suitable for production deployments

- Type-safe TypeScript frontend prevents runtime errors
- Structured Python backend with proper error handling
- Container orchestration for reliable service delivery
- Professional video processing with optimized settings

Real-World Operation Example

When an administrator wants to start streaming a video across 4 horizontal displays:

- Frontend: User uploads video file and creates a 4-screen horizontal group through the React interface
- Backend: Flask API validates the request and coordinates the operation through specialized blueprints
- Services: Docker service creates a streaming container, FFmpeg service generates video splitting commands
- Infrastructure: Docker container starts SRS server, FFmpeg process begins encoding and streaming
- Clients: 4 Raspberry Pi devices automatically connect and begin displaying their assigned video segments

This architecture enables complex multi-screen video walls to be managed through an intuitive web interface while maintaining professional reliability and performance standards suitable for commercial installations.

Project Structure

```
UB_Intern/
├── frontend/                    # React TypeScript application
│   ├── src/
│   │   ├── components/          # React components
│   │   │   ├── ui/              # shadcn/ui component library
│   │   │   ├── StreamsTab/      # Streaming management interface
│   │   │   ├── ClientsTab.tsx # Client management interface
│   │   │   └── VideoFilesTab.tsx # Video file management
│   │   ├── hooks/               # Custom React hooks
│   │   ├── types/               # TypeScript type definitions
│   │   └── App.tsx              # Main application component
│   ├── config/build/            # Build configuration
│   │   └── vite.config.ts       # Vite bundler configuration
│   └── package.json             # Frontend dependencies
├── backend/                     # Flask backend server
│   └── endpoints/
│       ├── flask_app.py         # Main Flask application entry point
│       ├── app_config.py        # Application configuration
management
│       ├── blueprints/          # Modular route handlers
│       │   ├── group_management.py   # Group lifecycle operations
│       │   ├── video_management.py   # Video file operations
│       │   ├── streaming/            # Stream control logic
│       │   ├── client_management/    # Client registration &
polling
│       │   └── docker_management.py  # Container orchestration
│       ├── services/            # Business logic services
│       │   ├── ffmpeg_service.py     # Video processing utilities
│       │   ├── docker_service.py     # Container management
│       │   ├── srt_service.py        # SRT protocol operations
│       │   └── error_service.py      # Structured error handling
│       └── uploads/             # Video file storage directory
├── client/                      # C++ client player application
│   └── multi-screen/
│       └── player/              # Player source code
└── Errors.md                    # Error code documentation
```

Backend Architecture

Blueprint Design Pattern

The backend uses Flask's blueprint pattern for modular organization:

1. Group Management Blueprint `(group_management.py)`

Purpose: Manages streaming groups and Docker container lifecycle

Key Functions:

- `create_group()`: Creates new streaming group with Docker container
- `get_groups()`: Lists all available groups from Docker discovery
- `delete_group()`: Safely removes groups and associated resources

Core Processes:

- **Port Allocation**: Each group gets 10 unique ports (RTMP, HTTP, API, SRT)
- **Container Labeling**: Groups tagged with metadata for discovery
- **Stream ID Generation**: Pre-generates unique identifiers

2. Video Management Blueprint `(video_management.py)`

Purpose: Handles video file upload, validation, and metadata extraction

Key Functions:

- `upload_video()`:Handles chunked file uploads with progress tracking
- `list_videos()`: Returns available video files with metadata
- `delete_video()`: Removes video files with cleanup

Features:

- Format validation (MP4, AVI, MOV, MKV)
- FFmpeg metadata extraction
- Automatic thumbnail generation
- File size and duration validation

3. Client Management Blueprint `(client_management/)`

Purpose: Handles client registration, discovery, and assignment

Core Components:

- `client_endpoints.py`: Registration and polling endpoints
- `info_endpoints.py`: Client information retrieval
- `client_state.py`: Thread-safe in-memory client state
- `client_utils.py`: Utility functions for client operations

Key Processes:

- **Registration**: Automatic client discovery with hostname extraction
- **Assignment**: Smart group assignment with screen positioning
- **Polling**: Real-time status updates with last-seen tracking
- **State Management**: Thread-safe operations with proper locking

4. Streaming Management Blueprint `(streaming/)`

Purpose: Controls FFmpeg processes and stream lifecycle

Key Components:

- `stream_endpoints.py`: Stream control API endpoints
- `stream_utils.py`: FFmpeg command generation utilities
- `process_manager.py`: FFmpeg process lifecycle management

Stream Modes:

- **Single Video Split**: One video divided across multiple screens
- **Multi-Video**: Different videos assigned to each screen
- **Layout Support**: Horizontal, vertical, and grid configurations

5. Docker Management Blueprint `(docker_management.py)`

Purpose: Direct Docker container operations and monitoring

Key Functions:

- `create_container()`: Creates SRS streaming containers

- `get_container_status()`: Real-time container health monitoring
- `cleanup_containers()`: Resource cleanup and management

Service Layer Architecture

1. Docker Service `(docker_service.py)`

Purpose: The Docker Service manages containerized Simple Realtime Server (SRS) instances for each streaming group, providing isolated streaming environments with automatic port allocation and lifecycle management. This service abstracts Docker complexity by handling container creation, health monitoring, and cleanup operations while ensuring complete isolation between different video wall installations. It implements intelligent port management to prevent conflicts in multi-group deployments and provides robust error handling with automatic recovery mechanisms for production reliability.

Key Features:

- Container lifecycle management
- Port allocation and conflict resolution
- Health monitoring and automatic recovery
- Resource limit enforcement

Container Configuration:

```
container_config = {
    "image": "registry.cn-hangzhou.aliyuncs.com/ossrs/srs:v5.0.0",
    "ports": {
        f"{rtmp_port}/tcp": rtmp_port,
        f"{http_port}/tcp": http_port,
        f"{api_port}/tcp": api_port,
        f"{srt_port}/udp": srt_port
    },
    "environment": {
        "SRS_RTMP_ENABLED": "on",
        "SRS_HTTP_ENABLED": "on",
        "SRS_SRT_ENABLED": "on"
    },
    "labels": {
        "multiscreen.group_id": group_id,
        "multiscreen.group_name": group_name,
        "multiscreen.created_at": str(time.time())
```

```
    }
}
```

2. FFmpeg Service (`ffmpeg_service.py`)

Purpose: The FFmpeg Service provides comprehensive video processing capabilities by managing FFmpeg executable detection, command generation, and process lifecycle for multi-screen streaming operations. This service automatically locates FFmpeg across different platforms, validates version compatibility, and generates optimized commands for video splitting, encoding, and streaming. It handles complex video processing scenarios including horizontal, vertical, and grid layouts while managing FFmpeg processes with proper monitoring and cleanup to ensure reliable streaming performance.

Key Features:

- Automatic FFmpeg detection across platforms
- Version compatibility checking
- Optimized command generation
- Process monitoring and cleanup

Command Generation Example:

```
def generate_split_command(input_file, layout, screens, srt_url):
    """Generate FFmpeg command for video splitting"""
    filters = []

    if layout == "horizontal":
        for i in range(screens):
            crop_filter = f"crop=iw/{screens}:ih:{i}*iw/{screens}:0"
            filters.append(f"[0:v]{crop_filter}[out{i}]")

    return [
        "ffmpeg", "-re", "-i", input_file,
        "-filter_complex", ";".join(filters),
        "-c:v", "libx264", "-preset", "veryfast",
        "-f", "mpegts", srt_url
    ]
```

3. SRT Service (`srt_service.py`)

Purpose: The SRT Service ensures reliable low-latency streaming by providing comprehensive SRT protocol testing, connection validation, and server monitoring capabilities. This service verifies SRT server accessibility through multi-layered testing approaches, monitors server readiness with fast polling mechanisms, and provides blocking wait functionality for synchronized startup operations. It implements robust connection testing that validates both basic network connectivity and actual streaming capability to ensure streams can be established before client connections are attempted.

Key Methods:

- `test_connection()`: Validates SRT server accessibility
- `monitor_srt_server()`: Fast polling for server readiness
- `wait_for_server()`: Blocks until server becomes available

Connection Testing Process:

1. Socket Test: Basic UDP connectivity check
2. FFmpeg Test: Validates actual SRT streaming capability
3. Response Timing: Measures connection latency

4. Error Service (`error_service.py`)

Purpose: The Error Service provides a comprehensive error handling framework with structured error codes, user-friendly messages, and actionable troubleshooting solutions for all system components. This service categorizes errors into logical groups, delivers consistent error responses across the application, and provides detailed diagnostic information with step-by-step resolution guides. It enables developers and administrators to quickly identify, understand, and resolve issues while maintaining system reliability through standardized error reporting and recovery mechanisms.

Error Categories:

- **1xx**: Stream Management (FFmpeg, SRT, Configuration)
- **2xx**: Docker Management (Containers, Ports, Services)
- **3xx**: Video Management (Files, Processing, Storage)
- **4xx**: Client Management (Registration, Assignment)
- **5xx**: System-Wide (Resources, Network, General)

Frontend Architecture

Component Architecture The frontend follows a modern React component architecture with TypeScript for type safety, maintainability, and professional user experience:

1. Main Application `(App.tsx)`

Purpose: Root component that orchestrates the entire application with centralized state management and navigation control. This component serves as the foundation for the entire user interface, coordinating global application state, managing real-time data updates through polling mechanisms, and providing a unified error handling strategy. It establishes the main navigation structure and ensures consistent communication between different functional areas of the application.

Key Features:

- Tab-based navigation (Streams, Clients, Videos)
- Global error boundary
- API client initialization
- Real-time polling coordination

2. Streams Management `(StreamsTab/)`

Purpose: Comprehensive interface for managing streaming groups and controlling multi-screen operations. This module provides administrators with complete control over video wall configurations, enabling them to create, monitor, and manage multiple streaming groups simultaneously. It offers intuitive visual controls for complex streaming operations while providing real-time feedback on system status and performance metrics.

Components:

- StreamsTab.tsx: Main streaming interface
- GroupCard/: Individual group management cards
- CreateGroupDialog/: Group creation modal
- StreamControls/: Stream start/stop controls

Key Features:

- Real-time group status monitoring
- Drag-and-drop client assignment
- Visual stream status indicators

- One-click stream controls

3. Client Management `(ClientsTab.tsx)`

Purpose: Centralized interface for device registration, assignment, and monitoring across all streaming groups. This component handles the complex task of managing multiple display devices, providing administrators with visibility into device health, connectivity status, and assignment configurations. It streamlines the process of organizing physical displays into logical groups while maintaining real-time awareness of device availability and performance.

Features:

- Auto-discovery of new clients
- Manual group assignment
- Real-time activity status
- Screen position mapping
- Bulk operations support

4. Video Management `(VideoFilesTab.tsx)`

Purpose: Complete video file lifecycle management with upload, organization, and metadata handling capabilities. This interface provides a professional-grade file management system specifically designed for video content, offering administrators the tools to efficiently organize, preview, and manage large video libraries. It handles the technical complexities of video file processing while presenting a user-friendly interface for content management operations.

Features:

- Drag-and-drop file upload
- Progress tracking with resumable uploads
- Video metadata display
- Thumbnail preview generation
- Batch delete operations

Custom Hooks Architecture

1. API Hooks `(hooks/)`

Purpose: Encapsulate API interactions with proper error handling, loading states, and real-time data synchronization. These hooks abstract complex API communication patterns into reusable components that provide consistent data fetching, caching, and error management across the application. They implement automatic polling mechanisms for real-time updates while handling network failures gracefully and providing developers with clean, predictable interfaces for backend communication.

Key Hooks:

- `useGroups()`: Group management operations
- `useClients()`: Client state and operations
- `useVideos()`: Video file management
- `useStreaming()`: Stream control operations

Example Hook Implementation:

```
export function useGroups() {
  const [groups, setGroups] = useState<Group[]>([]);
  const [loading, setLoading] = useState(false);
  const [error, setError] = useState<string | null>(null);

  const fetchGroups = useCallback(async () => {
    try {
      setLoading(true);
      const response = await api.get('/groups');
      setGroups(response.data.groups);
      setError(null);
    } catch (err) {
      setError(err.message);
    } finally {
      setLoading(false);
    }
  }, []);

  useEffect(() => {
```

```
    fetchGroups();
    const interval = setInterval(fetchGroups, 3000);
    return () => clearInterval(interval);
  }, [fetchGroups]);

  return { groups, loading, error, refetch: fetchGroups };
}
```

2. State Management Hooks

Purpose: Manage complex component state with proper TypeScript typing, optimistic updates, and coordinated loading states. These hooks provide sophisticated state management patterns that enhance user experience through predictive UI updates, comprehensive error handling, and seamless real-time data synchronization. They encapsulate common state patterns to ensure consistent behavior and reduce code duplication across components.

Key Patterns:

- Optimistic updates for better UX
- Error state management
- Loading state coordination
- Real-time data synchronization

UI Component System

1. shadcn/ui Integration

Purpose: Professional, accessible component library that provides a consistent design system with modern aesthetics and comprehensive accessibility features. This integration ensures visual consistency across the application while leveraging battle-tested components that handle complex interaction patterns, keyboard navigation, and screen reader compatibility out of the box.

Core Components Used:

- Card: Group and client information display
- Button: Action triggers with loading states
- Dialog: Modal interactions
- Table: Data presentation with sorting
- Form: Input validation and submission

- Toast: User feedback notifications

2. Custom Components

Purpose: Application-specific UI components tailored to the unique requirements of multi-screen streaming management. These components encapsulate complex domain-specific functionality while maintaining the design consistency established by the shadcn/ui system. They provide specialized interfaces for streaming operations that aren't available in standard component libraries.

Key Components:

- GroupCard: Collapsible group management interface
- ClientStatusBadge: Real-time client status indicators
- VideoUploadZone: Drag-and-drop file upload
- StreamingControls: Stream management buttons

Data Types and Interfaces

Backend Data Structures

1. Group Data Structure

```
class Group:
    id: str                    # Unique group identifier
    name: str                  # Human-readable group name
    layout: str                # "horizontal", "vertical", "grid"
    screen_count: int          # Number of screens in group
    docker_container_id: str   # Associated Docker container
    docker_running: bool       # Container status
    ports: Dict[str, int]      # Allocated port mappings
    stream_urls: Dict[str, str] # Generated stream URLs
    created_at: float          # Creation timestamp
    last_modified: float       # Last modification timestamp
```

2. Client Data Structure

```
class Client:
    client_id: str             # Unique client identifier
(IP_hostname_timestamp)
    hostname: str              # Client hostname
    ip_address: str            # Client IP address
    group_id: Optional[str]    # Assigned group ID
    screen_number: Optional[int] # Assigned screen position
    assignment_status: str     # "unassigned", "group_assigned",
"stream_assigned"
    stream_url: Optional[str]  # Assigned stream URL
    last_seen: float           # Last activity timestamp
    registration_time: float   # Initial registration time
    capabilities: Dict[str, Any] # Client capabilities and metadata
```

3. Video File Data Structure

```
class VideoFile:
    filename: str              # Original filename
    filepath: str              # Server storage path
    filesize: int              # File size in bytes
    duration: float            # Video duration in seconds
    resolution: str            # Video resolution (e.g., "1920x1080")
    codec: str                 # Video codec information
    framerate: float           # Frames per second
    upload_time: float         # Upload timestamp
    metadata: Dict[str, Any]   # FFprobe metadata
```

4. Stream Configuration

```
class StreamConfig:
    group_id: str              # Target group
    mode: str                  # "single_split" or "multi_video"
    video_files: List[str]     # Video file paths
    layout: str                # Screen layout type
    screen_assignments: Dict[int, str] # Screen to video mapping
    streaming_params: Dict[str, Any]   # FFmpeg parameters
```

Frontend TypeScript Interfaces

1. Group Interface

```typescript
interface Group {
  id: string;
  name: string;
  layout: 'horizontal' | 'vertical' | 'grid';
  screenCount: number;
  dockerRunning: boolean;
  streamingStatus: 'stopped' | 'starting' | 'running' | 'error';
  assignedClients: number;
  createdAt: string;
  ports: {
    rtmp: number;
    http: number;
    api: number;
    srt: number;
  };
  streamUrls: Record<string, string>;
}
```

2. Client Interface

```typescript
interface Client {
  clientId: string;
  hostname: string;
  ipAddress: string;
  groupId?: string;
  screenNumber?: number;
  assignmentStatus: 'unassigned' | 'group_assigned' |
'stream_assigned';
  streamUrl?: string;
  lastSeen: number;
  isActive: boolean;
  registrationTime: number;
  displayName: string;
}
```

3. Video File Interface

```
interface VideoFile {
  filename: string;
  filepath: string;
  filesize: number;
  duration: number;
  resolution: string;
  codec: string;
  framerate: number;
  uploadTime: number;
  thumbnailPath?: string;
  metadata: Record<string, any>;
}
```

4. API Response Types

```
interface ApiResponse<T> {
  success: boolean;
  data?: T;
  error?: {
    code: number;
    message: string;
    category: string;
    solutions: string[];
  };
  timestamp: number;
}

interface PaginatedResponse<T> extends ApiResponse<T[]> {
  pagination: {
    page: number;
    pageSize: number;
    total: number;
    totalPages: number;
  };
}
```

Error Response Structure

```
interface ErrorResponse {
  error_code: number;
  error_category: string;
  message: string;
  meaning: string;
  common_causes: string[];
  primary_solution: string;
  detailed_solutions: string[];
  timestamp: number;
  context?: Record<string, any>;
}
```

API Reference

Group Management Endpoints

Create Group

Description: Creates a new streaming group with specified layout configuration and automatically provisions a Docker container with SRS streaming server. This endpoint allocates unique ports, generates stream URLs, and initializes the infrastructure needed for multi-screen streaming operations.

```
POST /api/groups/create
Content-Type: application/json

{
  "name": "Conference Room Display",
  "layout": "horizontal",
  "screen_count": 4
}
```

Response:

```
{
  "success": true,
  "data": {
    "group_id": "group_1648392847123",
    "name": "Conference Room Display",
    "layout": "horizontal",
    "screen_count": 4,
    "docker_container_id": "multiscreen_group_1648392847123",
    "ports": {
      "rtmp": 5001,
      "http": 5011,
      "api": 5021,
      "srt": 10081
    }
  }
}
```

List Groups

Description: Retrieves all existing streaming groups with their current status, including Docker container health, assigned client counts, and streaming state. This endpoint provides a real-time overview of all configured video wall installations.

```
GET /api/groups
```

Response:

```
{
  "success": true,
  "data": {
    "groups": [
      {
        "id": "group_1648392847123",
        "name": "Conference Room Display",
        "layout": "horizontal",
        "screen_count": 4,
        "docker_running": true,
        "assigned_clients": 3,
        "streaming_status": "running"
      }
    ]
  }
}
```

Delete Group

Description: Permanently removes a streaming group, stops any active streams, terminates the associated Docker container, and releases allocated network ports. This operation also unassigns all connected clients from the group.

```
DELETE /api/groups/{group_id}
```

Client Management Endpoints

Register Client

Description: Registers a new display device (client) with the system, creating a unique client identifier and capturing device capabilities. This endpoint is typically called automatically when a Raspberry Pi player application starts up and discovers the server.

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

{
  "hostname": "display-001",
  "capabilities": {
    "resolution": "1920x1080",
    "refresh_rate": 60
  }
}
```

Response:

```
{
  "success": true,
  "data": {
    "client_id": "192.168.1.101_display-001_1648392847123",
    "registration_status": "registered",
    "assigned_group": null,
    "next_poll_interval": 5000
  }
}
```

Client Polling

Description: Allows clients to check for assignment updates, stream URLs, and configuration changes. Clients call this endpoint every 5 seconds to receive real-time updates about their group assignment and streaming instructions. This is the primary communication mechanism between display devices and the server.

```
POST /api/clients/poll
Content-Type: application/json

{
  "client_id": "192.168.1.101_display-001_1648392847123"
}
```

Response:

```
{
  "success": true,
  "data": {
    "status": "assigned",
    "group_id": "group_1648392847123",
    "screen_number": 1,
    "stream_url": "srt://192.168.1.100:10081?streamid=screen_1",
    "next_poll_interval": 5000
  }
}
```

List Clients

Description: Provides comprehensive overview of all registered client devices, including their assignment status, activity state, and connection health. This endpoint includes summary statistics for monitoring the overall health of the display device fleet.

http

```
GET /api/clients
```

Response:

```
{
  "success": true,
  "data": {
    "clients": [
      {
        "client_id": "192.168.1.101_display-001_1648392847123",
        "hostname": "display-001",
        "ip_address": "192.168.1.101",
        "group_id": "group_1648392847123",
        "screen_number": 1,
        "assignment_status": "stream_assigned",
        "last_seen": 1648392900,
        "is_active": true
      }
    ],
    "statistics": {
      "total_clients": 5,
      "active_clients": 4,
      "assigned_clients": 3
    }
  }
}
```

Video Management Endpoints

Upload Video

Description: Handles video file uploads with automatic metadata extraction using FFprobe. This endpoint validates file formats, extracts technical specifications (resolution, duration, codec), and stores files in the server's upload directory for use in streaming operations.

```
POST /api/videos/upload
Content-Type: multipart/form-data

Form Data:
- file: (video file)
- metadata: {"description": "Sample video"}
```

Response:

```
{
  "success": true,
  "data": {
    "filename": "sample_video.mp4",
    "filesize": 15728640,
    "duration": 120.5,
    "resolution": "1920x1080",
    "upload_id": "upload_1648392847123"
  }
}
```

List Videos

Description: Returns all available video files with comprehensive metadata including technical specifications, file sizes, and upload timestamps. This endpoint provides the information needed for video selection in streaming operations.

http

```
GET /api/videos
```

Response:

```
{
  "success": true,
  "data": {
    "videos": [
      {
        "filename": "sample_video.mp4",
        "filesize": 15728640,
        "duration": 120.5,
        "resolution": "1920x1080",
        "codec": "h264",
        "framerate": 30.0,
        "upload_time": 1648392847
      }
    ]
  }
}
```

Streaming Management Endpoints

Start Stream

Description: Initiates video streaming for a specified group using either single-video split mode (one video divided across screens) or multi-video mode (different videos per screen). This endpoint starts FFmpeg processes, generates appropriate filter graphs, and begins SRT streaming to assigned clients.

```
POST /api/streaming/start
Content-Type: application/json

{
  "group_id": "group_1648392847123",
  "mode": "single_split",
  "video_files": ["sample_video.mp4"],
  "layout": "horizontal"
}
```

Response:

```
{
  "success": true,
  "data": {
    "stream_id": "stream_1648392847123",
    "status": "starting",
    "ffmpeg_process_id": 12345,
    "stream_urls": {
      "screen_1": "srt://192.168.1.100:10081?streamid=screen_1",
      "screen_2": "srt://192.168.1.100:10081?streamid=screen_2"
    }
  }
}
```

Stop Stream

Description: Terminates active streaming for a group by stopping FFmpeg processes and cleaning up associated resources. This endpoint gracefully shuts down video processing and notifies connected clients that streaming has ended.

```
POST /api/streaming/stop
Content-Type: application/json

{
  "group_id": "group_1648392847123"
}
```

Stream Status

Description: Provides detailed real-time information about active streaming operations, including process health, performance metrics, and resource utilization. This endpoint is used for monitoring stream quality and diagnosing performance issues.

```
GET /api/streaming/status/{group_id}
```

Response:

```
json
{
  "success": true,
  "data": {
    "group_id": "group_1648392847123",
    "streaming_status": "running",
    "ffmpeg_process_id": 12345,
    "start_time": 1648392847,
    "uptime": 3600,
    "stream_health": {
      "cpu_usage": 25.5,
      "memory_usage": 512,
      "network_throughput": "5.2 Mbps"
    }
  }
}
```

Services Documentation

1. Docker Service `(docker_service.py)`

Purpose

Manages Docker container lifecycle for SRS streaming servers with automatic port allocation and health monitoring.

Key Functions

```
# create_streaming_container(group_id, group_name, screen_count)
def create_streaming_container(group_id: str, group_name: str,
screen_count: int) -> Dict[str, Any]:
    """
    Creates a new SRS streaming container for a group.

    Args:
        group_id: Unique identifier for the group
        group_name: Human-readable name for the group
        screen_count: Number of screens in the group

    Returns:
        Dict containing container info and allocated ports

    Raises:
        DockerException: If container creation fails
        PortAllocationError: If no ports available
    """
```

```
# get_container_health(container_id)
def get_container_health(container_id: str) -> Dict[str, Any]:
    """
    Retrieves detailed health information for a container.

    Returns:
        {
            "status": "running" | "stopped" | "error",
            "cpu_usage": float,
            "memory_usage": int,
```

```
            "network_io": Dict[str, int],
            "uptime": int
        }
    """
```

Port Allocation Strategy

- Each group receives a block of 10 consecutive ports
- Port ranges: 5000-5099 (RTMP), 5100-5199 (HTTP), 5200-5299 (API), 10080-10179 (SRT)
- Automatic conflict detection and resolution
- Port recycling when groups are deleted

2. FFmpeg Service (`ffmpeg_service.py`)

Purpose

Provides FFmpeg executable management, command generation, and process monitoring for video processing operations.

Key Functions

```
# find_ffmpeg_executable()
def find_ffmpeg_executable() -> str:
    """
    Locates FFmpeg executable across different platforms.

    Search order:
    1. Current working directory
    2. System PATH
    3. Common installation directories
    4. Package manager locations

    Returns:
        Path to FFmpeg executable

    Raises:
        FFmpegNotFoundError: If FFmpeg is not available
    """
```

```python
# generate_split_command(input_file, layout, screens, base_srt_url)
def generate_split_command(
    input_file: str,
    layout: str,
    screens: int,
    base_srt_url: str
) -> List[str]:
    """
    Generates FFmpeg command for video splitting across screens.

    Supports:
    - Horizontal splitting (side-by-side screens)
    - Vertical splitting (stacked screens)
    - Grid splitting (rows and columns)

    Args:
        input_file: Path to source video file
        layout: "horizontal", "vertical", or "grid"
        screens: Number of output screens
        base_srt_url: Base SRT URL for streaming

    Returns:
        Complete FFmpeg command as list of arguments
    """
```

Command Generation Examples

```
Horizontal Split (4 screens):
ffmpeg -re -i input.mp4 \
  -filter_complex
"[0:v]crop=iw/4:ih:0*iw/4:0[out1];[0:v]crop=iw/4:ih:1*iw/4:0[out2];[0
:v]crop=iw/4:ih:2*iw/4:0[out3];[0:v]crop=iw/4:ih:3*iw/4:0[out4]" \
  -map "[out1]" -c:v libx264 -preset veryfast -f mpegts
srt://192.168.1.100:10080?streamid=screen_1 \
  -map "[out2]" -c:v libx264 -preset veryfast -f mpegts
srt://192.168.1.100:10080?streamid=screen_2 \
  -map "[out3]" -c:v libx264 -preset veryfast -f mpegts
srt://192.168.1.100:10080?streamid=screen_3 \
  -map "[out4]" -c:v libx264 -preset veryfast -f mpegts
srt://192.168.1.100:10080?streamid=screen_4
```

Grid Split (2x2):

```
ffmpeg -re -i input.mp4 \
  -filter_complex
"[0:v]crop=iw/2:ih/2:0:0[out1];[0:v]crop=iw/2:ih/2:iw/2:0[out2];[0:v]
crop=iw/2:ih/2:0:ih/2[out3];[0:v]crop=iw/2:ih/2:iw/2:ih/2[out4]" \
  ...
```

3. SRT Service (`srt_service.py`)

Purpose

Handles SRT protocol testing, connection validation, and server monitoring for reliable streaming.

Key Functions

```python
# test_srt_connection(host, port, timeout=5)
def test_srt_connection(host: str, port: int, timeout: int = 5) ->
Dict[str, Any]:
    """
    Tests SRT server connectivity using multiple methods.

    Test Methods:
    1. UDP socket connectivity test
    2. FFmpeg SRT probe test
    3. Latency measurement

    Args:
        host: SRT server hostname/IP
        port: SRT server port
        timeout: Connection timeout in seconds

    Returns:
        {
            "success": bool,
            "latency_ms": float,
            "connection_method": str,
            "error_details": Optional[str]
        }
    """
```

```python
# monitor_srt_server(host, port, check_interval=1)
def monitor_srt_server(host: str, port: int, check_interval: int = 1)
-> Iterator[Dict[str, Any]]:
    """
    Continuously monitors SRT server availability.

    Yields real-time server status updates for health monitoring
    and automatic recovery operations.
    """
```

```python
# wait_for_srt_server(host, port, max_wait=30)
def wait_for_srt_server(host: str, port: int, max_wait: int = 30) ->
bool:
    """
    Blocks until SRT server becomes available or timeout occurs.

    Used during container startup to ensure server readiness
    before starting FFmpeg processes.
    """
```

4. Error Service`(error_service.py)`

Purpose

Provides comprehensive error handling with structured error codes, user-friendly messages, and actionable solutions.

Error Code Categories

1xx - Stream Management Errors

- 100-119: FFmpeg Process Errors
- 120-139: SRT Connection Errors
- 140-159: Stream Configuration Errors
- 160-179: Stream Monitoring Errors
- 180-199: Video Processing Errors

2xx - Docker Management Errors

- 200-219: Container Lifecycle Errors
- 220-239: Port Management Errors
- 240-259: Container Health Errors
- 260-279: Docker Service Errors

3xx - Video Management Errors

- 300-319: File Operations Errors
- 320-339: Upload/Download Errors
- 340-359: Validation Errors
- 360-379: Metadata Errors

4xx - Client Management Errors

- 400-419: Registration Errors
- 420-439: Assignment Errors
- 440-459: Communication Errors
- 460-479: State Management Errors

5xx - System-Wide Errors

- 500-519: Resource Errors
- 520-539: Network Errors
- 540-559: Configuration Errors
- 560-579: Authentication/Authorization Errors

Key Functions

```
# get_error_info(error_code)
def get_error_info(error_code: int) -> Dict[str, Any]:
    """
    Retrieves comprehensive error information.

    Returns:
        {
            "error_code": int,
            "error_category": str,
            "message": str,
            "meaning": str,
            "common_causes": List[str],
```

```
            "primary_solution": str,
            "detailed_solutions": List[str],
            "documentation_link": str
        }
    """
```

Example Error Response:

```
{
  "error_code": 143,
  "error_category": "1xx",
  "message": "Group not found in Docker",
  "meaning": "The specified group has no corresponding Docker
container",
  "common_causes": [
    "Group was deleted",
    "Docker container failed to start",
    "Container was manually removed"
  ],
  "primary_solution": "Create Docker container for the group",
  "detailed_solutions": [
    "Check if group exists: docker ps -a --filter
label=multiscreen.group_id=<group_id>",
    "Recreate group: POST /api/groups/create",
    "Check Docker daemon status: systemctl status docker",
    "Verify Docker permissions: docker info"
  ]
}
```

Error Handling System

Backend Error Handling

1. Exception Hierarchy

Description: Establishes a structured exception hierarchy with a base `MultiscreenError` class that standardizes error information across the entire system. Each specialized exception type includes error codes, contextual information, and categorization to enable consistent error handling and automated recovery mechanisms throughout the application.

```
class MultiscreenError(Exception):
    """Base exception for all multiscreen errors"""
    def __init__(self, message: str, error_code: int, context:
Dict[str, Any] = None):
        self.message = message
        self.error_code = error_code
        self.context = context or {}
        super().__init__(self.message)

class StreamingError(MultiscreenError):
    """Errors related to streaming operations"""
    pass

class DockerError(MultiscreenError):
    """Errors related to Docker operations"""
    pass

class ClientError(MultiscreenError):
    """Errors related to client operations"""
    pass
```

2. Error Response Format

Description: Creates standardized error responses that include structured error codes, human-readable messages, detailed explanations, and actionable troubleshooting steps. This function ensures consistent error communication across all API endpoints while providing developers and administrators with comprehensive diagnostic information.

```
def create_error_response(error_code: int, context: Dict[str, Any] =
None) -> Dict[str, Any]:
    """
    Creates standardized error response with troubleshooting
information.
    """
    error_info = get_error_info(error_code)

    return {
        "success": False,
        "error": {
            "code": error_code,
            "category": error_info["error_category"],
            "message": error_info["message"],
            "meaning": error_info["meaning"],
            "solutions": error_info["detailed_solutions"],
            "context": context,
            "timestamp": time.time()
        }
    }
```

3. Error Recovery Mechanisms

Description: Implements intelligent automatic recovery strategies for common system failures. These static methods attempt to resolve issues without user intervention by restarting failed services, reallocating conflicting resources, or re-establishing broken connections. This proactive approach minimizes system downtime and reduces the need for manual troubleshooting.

```
class ErrorRecovery:
    """Automatic error recovery for common issues"""

    @staticmethod
    def recover_from_docker_error(error_code: int, context: Dict[str,
Any]) -> bool:
        """Attempts automatic recovery from Docker-related errors"""
        if error_code == 210:  # Container not running
            container_id = context.get("container_id")
            return restart_container(container_id)
        elif error_code == 225:  # Port conflict
            return reallocate_ports(context.get("group_id"))
        return False

    @staticmethod
    def recover_from_stream_error(error_code: int, context: Dict[str,
Any]) -> bool:
        """Attempts automatic recovery from streaming errors"""
        if error_code == 120:  # SRT connection refused
            return wait_for_srt_server(context.get("host"),
context.get("port"))
        elif error_code == 101:  # FFmpeg process terminated
            return restart_ffmpeg_process(context.get("group_id"))
        return False
```

Frontend Error Handling

1. Error Boundary Component

Description: Provides application-wide error catching and graceful failure handling for React components. This boundary component prevents entire application crashes by catching JavaScript errors anywhere in the component tree, logging detailed error information for debugging, and displaying user-friendly fallback interfaces when components fail unexpectedly.

```
interface ErrorBoundaryState {
  hasError: boolean;
  error: Error | null;
  errorInfo: ErrorInfo | null;
}

class ErrorBoundary extends Component<PropsWithChildren<{}>,
ErrorBoundaryState> {
  constructor(props: PropsWithChildren<{}>) {
    super(props);
    this.state = { hasError: false, error: null, errorInfo: null };
  }

  static getDerivedStateFromError(error: Error): ErrorBoundaryState {
    return { hasError: true, error, errorInfo: null };
  }

  componentDidCatch(error: Error, errorInfo: ErrorInfo) {
    this.setState({ errorInfo });
    // Log error to monitoring service
    console.error('Error Boundary caught an error:', error,
errorInfo);
  }

  render() {
    if (this.state.hasError) {
      return <ErrorFallback error={this.state.error}
errorInfo={this.state.errorInfo} />;
    }
    return this.props.children;
```

```
  }
}
```

2. API Error Handling Hook

Description: Provides a centralized hook for managing API error states across React components. This hook standardizes error handling patterns, extracts structured error information from API responses, and provides fallback error messages for unexpected failures. It enables consistent error state management and user feedback throughout the application.

```
interface ApiError {
  code: number;                  // Structured error code for programmatic handling
  category: string;              // Error category (1xx, 2xx, 3xx, 4xx, 5xx)
  message: string;               // User-friendly error message
  solutions: string[];           // Array of suggested resolution steps
  context?: Record<string, any>; // Additional error context for debugging
}

export function useApiError() {
  const [error, setError] = useState<ApiError | null>(null);

  const handleApiError = useCallback((error: any) => {
    if (error.response?.data?.error) {
      setError(error.response.data.error);
    } else {
      setError({
        code: 500,
        category: '5xx',
        message: 'An unexpected error occurred',
        solutions: ['Please try again later', 'Contact support if the problem persists']
      });
    }
  }, []);
```

```
  const clearError = useCallback(() => setError(null), []);

  return { error, handleApiError, clearError };
}
```

3. Error Display Components

Description: Renders user-friendly error alerts with structured information and actionable guidance. This component transforms technical error responses into accessible user interfaces that display error messages, suggested solutions, and dismissal controls. It provides consistent visual error presentation while helping users understand and resolve issues independently.

```
interface ErrorAlertProps {
  error: ApiError;              // Error object containing code, message, and solutions
  onDismiss: () => void;        // Callback function to clear the error state
  showSolutions?: boolean;      // Whether to display suggested solution steps
}

export function ErrorAlert({ error, onDismiss, showSolutions = true }: ErrorAlertProps) {
  return (
    <Alert variant="destructive">
      <AlertTriangle className="h-4 w-4" />
      <AlertTitle>Error {error.code}</AlertTitle>
      <AlertDescription>
        <p>{error.message}</p>
        {showSolutions && error.solutions.length > 0 && (
          <div className="mt-2">
            <p className="font-medium">Suggested solutions:</p>
            <ul className="list-disc list-inside text-sm">
              {error.solutions.map((solution, index) => (
                <li key={index}>{solution}</li>
              ))}
            </ul>
          </div>
        )}
```

```
      </AlertDescription>
      <Button variant="outline" size="sm" onClick={onDismiss}
className="mt-2">
        Dismiss
      </Button>
    </Alert>
  );
}
```

Development Guide

Setting Up Development Environment

1. Prerequisites

```
# System requirements
- Ubuntu 22.04+ or macOS 10.15+
- Python 3.8+
- Node.js 16+
- Docker 20.10+
- FFmpeg with H.264 support

# Install system dependencies (Ubuntu)
sudo apt update
sudo apt install python3-pip nodejs npm docker.io ffmpeg

# Install system dependencies (macOS)
brew install python@3.9 node docker ffmpeg
```

2. Repository Setup

```
# Clone repository
git clone https://github.com/your-org/multiscreen.git
cd multiscreen

# Backend setup
cd backend/endpoints
python3 -m venv venv
source venv/bin/activate
```

```
pip install -r requirements.txt

# Frontend setup
cd ../../frontend
npm install
```

3. Configuration

```
# Backend configuration
cp backend/endpoints/app_config.example.json
backend/endpoints/app_config.json
# Edit configuration as needed

# Environment variables
export FLASK_ENV=development
export FLASK_DEBUG=1
export MULTISCREEN_UPLOADS_DIR=/path/to/uploads
```

Development Workflow

1. Running Development Servers

```
# Terminal 1: Backend
cd backend/endpoints
source venv/bin/activate
python flask_app.py

# Terminal 2: Frontend
cd frontend
npm run dev

# Terminal 3: Docker containers (if needed)
docker compose up -d
```

2. Code Quality Tools

Backend (Python):

```
# Linting
flake8 backend/endpoints/
black backend/endpoints/
mypy backend/endpoints/

# Testing
pytest backend/endpoints/tests/
pytest --cov=backend/endpoints backend/endpoints/tests/
```

Frontend (TypeScript):

```
# Linting and formatting
npm run lint
npm run format
npm run type-check

# Testing
npm test
npm run test:coverage
```

3. Development Best Practices

Backend Development:

- Use type hints for all function signatures
- Follow PEP 8 style guidelines
- Write comprehensive docstrings
- Implement proper error handling with structured error codes
- Use dependency injection for testability
- Write unit tests for all business logic

Frontend Development:

- Use TypeScript strict mode
- Implement proper error boundaries
- Use custom hooks for API interactions
- Follow React best practices (memo, callback, effect dependencies)
- Write tests for critical user flows
- Use semantic HTML and ARIA attributes

4. Testing Strategy

Backend Testing:

```
# Unit tests
def test_create_group():
    """Test group creation with valid parameters"""
    response = client.post('/api/groups/create', json={
        'name': 'Test Group',
        'layout': 'horizontal',
        'screen_count': 4
    })
    assert response.status_code == 200
    assert response.json['success'] is True

# Integration tests
def test_streaming_workflow():
    """Test complete streaming workflow"""
    # Create group
    group_response = create_test_group()
    group_id = group_response.json['data']['group_id']

    # Upload video
    video_response = upload_test_video()

    # Start streaming
    stream_response = start_test_stream(group_id)
    assert stream_response.json['success'] is True
```

Frontend Testing:

```
// Component tests
describe('GroupCard', () => {
  it('displays group information correctly', () => {
    const mockGroup = createMockGroup();
    render(<GroupCard group={mockGroup} />);

    expect(screen.getByText(mockGroup.name)).toBeInTheDocument();
    expect(screen.getByText(`${mockGroup.screenCount}
screens`)).toBeInTheDocument();
  });

  it('handles stream start action', async () => {
    const mockOnStreamStart = jest.fn();
    const mockGroup = createMockGroup();

    render(<GroupCard group={mockGroup}
onStreamStart={mockOnStreamStart} />);

    const startButton = screen.getByRole('button', { name: /start
stream/i });
    fireEvent.click(startButton);

    await waitFor(() => {
      expect(mockOnStreamStart).toHaveBeenCalledWith(mockGroup.id);
    });
  });
});

// API hook tests
describe('useGroups', () => {
  it('fetches groups on mount', async () => {
    const mockGroups = [createMockGroup()];
    jest.spyOn(api, 'get').mockResolvedValue({ data: { groups:
mockGroups } });

    const { result } = renderHook(() => useGroups());
```

```
    await waitFor(() => {
      expect(result.current.groups).toEqual(mockGroups);
      expect(result.current.loading).toBe(false);
    });
  });
});
```

Build and Deployment

1. Production Build

```
# Frontend build
cd frontend
npm run build

# Backend preparation
cd backend/endpoints
pip install -r requirements-prod.txt
```

2. Docker Deployment

```
# Multi-stage Dockerfile
FROM node:18-alpine AS frontend-build
WORKDIR /app/frontend
COPY frontend/package*.json ./
RUN npm ci
COPY frontend/ ./
RUN npm run build

FROM python:3.9-slim AS backend
WORKDIR /app
COPY backend/endpoints/requirements.txt ./
RUN pip install -r requirements.txt
COPY backend/endpoints/ ./
COPY --from=frontend-build /app/frontend/dist ./static

EXPOSE 5000
CMD ["python", "flask_app.py"]
```

3. Environment Configuration

```
# docker-compose.prod.yml
version: '3.8'
services:
  multiscreen-app:
    build: .
    ports:
      - "5000:5000"
    environment:
      - FLASK_ENV=production
      - MULTISCREEN_UPLOADS_DIR=/app/uploads
    volumes:
      - uploads:/app/uploads
      - /var/run/docker.sock:/var/run/docker.sock
    depends_on:
      - redis

  redis:
    image: redis:7-alpine
    volumes:
      - redis_data:/data

volumes:
  uploads:
  redis_data:
```

Deployment Guide

Production Deployment

1. Server Requirements

```
# Minimum hardware specifications
- CPU: 4 cores (8+ recommended for multiple groups)
- RAM: 8GB (16GB+ recommended)
- Storage: 100GB+ SSD for video files
- Network: Gigabit Ethernet
```

```
# Software requirements
- Ubuntu 22.04 LTS or CentOS 8+
- Docker 20.10+
- Docker Compose 2.0+
- FFmpeg with hardware acceleration support
```

2. System Preparation

```
# Update system
sudo apt update && sudo apt upgrade -y

# Install Docker
curl -fsSL https://get.docker.com -o get-docker.sh
sudo sh get-docker.sh
sudo usermod -aG docker $USER

# Install Docker Compose
sudo curl -L
"https://github.com/docker/compose/releases/download/v2.12.2/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose
sudo chmod +x /usr/local/bin/docker-compose

# Configure firewall
sudo ufw allow 5000/tcp
sudo ufw allow 5000-5100/tcp
sudo ufw allow 10080-10180/udp
sudo ufw enable
```

3. Application Deployment

```
# Clone and configure
git clone https://github.com/your-org/multiscreen.git
cd multiscreen

# Configure production environment
cp docker-compose.prod.yml docker-compose.yml
cp .env.example .env
# Edit .env with production values

# Deploy
docker-compose up -d

# Verify deployment
docker-compose ps
curl http://localhost:5000/api/health
```

4. SSL/TLS Configuration (Optional)

```
# /etc/nginx/sites-available/multiscreen
server {
    listen 80;
    server_name your-domain.com;
    return 301 https://$server_name$request_uri;
}

server {
    listen 443 ssl http2;
    server_name your-domain.com;

    ssl_certificate /path/to/certificate.crt;
    ssl_certificate_key /path/to/private.key;

    location / {
        proxy_pass http://localhost:5000;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;
    }
```

```
}
```

Client Deployment

1. Raspberry Pi Setup

```
# Flash Raspberry Pi OS (64-bit recommended)
# Enable SSH and configure network

# Update system
sudo apt update && sudo apt upgrade -y

# Install dependencies
sudo apt install -y cmake build-essential git
sudo apt install -y libavformat-dev libavcodec-dev libavutil-dev
sudo apt install -y libssl-dev libx11-dev
```

2. Client Application Build

```
# Clone repository on Raspberry Pi
git clone https://github.com/your-org/multiscreen.git
cd multiscreen/client/multi-screen

# Build application
mkdir build && cd build
cmake .. -DCMAKE_BUILD_TYPE=Release
make -j$(nproc)

# Install
sudo make install
```

3. Auto-start Configuration

```
# Create systemd service
sudo tee /etc/systemd/system/multiscreen-client.service > /dev/null
<<EOF
[Unit]
Description=Multiscreen Client Player
After=network.target

[Service]
Type=simple
User=pi
WorkingDirectory=/home/pi/multiscreen/client/multi-screen/build
ExecStart=/home/pi/multiscreen/client/multi-screen/build/player
--server-ip=192.168.1.100
Restart=always
RestartSec=10

[Install]
WantedBy=multi-user.target
EOF

# Enable and start service
sudo systemctl enable multiscreen-client
sudo systemctl start multiscreen-client
```

Troubleshooting

Common Issues and Solutions

1. Docker Container Issues

Problem: Container fails to start

```
# Check container logs
docker logs multiscreen_group_<id>

# Check port conflicts
netstat -tulpn | grep -E "5000|10080"

# Restart Docker service
sudo systemctl restart docker
```

Problem: Port allocation conflicts

```
# List allocated ports
docker port $(docker ps -q --filter label=multiscreen.group_id)

# Check system port usage
sudo lsof -i :5000-5100
sudo lsof -i :10080-10180

# Manual port cleanup
docker stop $(docker ps -q --filter label=multiscreen)
```

2. FFmpeg Process Issues

Problem: FFmpeg process fails to start

```
# Check FFmpeg installation
ffmpeg -version
which ffmpeg

# Test video file accessibility
ffprobe /path/to/video/file.mp4

# Check SRT connectivity
ffmpeg -f lavfi -i testsrc=duration=10:size=320x240:rate=1 -f mpegts
srt://192.168.1.100:10080?streamid=test
```

Problem: Poor streaming performance

```
# Monitor system resources
top -p $(pgrep ffmpeg)
iostat -x 1

# Check network bandwidth
iftop -i eth0

# Optimize FFmpeg settings
# Reduce preset from 'veryfast' to 'ultrafast'
# Lower bitrate or resolution
# Enable hardware acceleration if available
```

3. Client Connection Issues

Problem: Clients not registering

Check network connectivity
ping <server-ip>
telnet <server-ip> 5000

Verify client application
./player --debug --server-ip=<server-ip>

Check server logs
docker logs <container-id> | grep client

Problem: Stream playback issues

```
# Test SRT stream manually
ffplay srt://<server-ip>:10080?streamid=screen_1

# Check client logs
journalctl -u multiscreen-client -f

# Verify stream URL
curl http://<server-ip>:5000/api/clients/<client-id>
```

4. Performance Optimization

System Resource Monitoring:

```
# Monitor CPU usage
htop

# Monitor memory usage
free -h
vmstat 1

# Monitor disk I/O
iotop

# Monitor network usage
nethogs
```

Optimization Strategies:

- Use hardware-accelerated encoding when available
- Optimize video resolution and bitrate for network capacity
- Implement stream quality adaptation
- Use SSD storage for video files
- Configure appropriate buffer sizes
- Monitor and limit concurrent streams

5. Debug Mode Operations

Enable Debug Logging:

```
# Backend debug mode
import logging
logging.basicConfig(level=logging.DEBUG)

# Add to flask_app.py
app.config['DEBUG'] = True
```

Frontend Debug Mode:

```
// Enable React DevTools
// Add to development environment
if (process.env.NODE_ENV === 'development') {
  console.log('Debug mode enabled');
  // Additional debug logging
}
```

Network Debugging:

```
# Monitor SRT traffic
sudo tcpdump -i any port 10080

# Check HTTP API calls
curl -v http://localhost:5000/api/groups

# Monitor Docker network
docker network ls
docker network inspect multiscreen_default
```

Logging and Monitoring

1. Application Logs

```
# Backend logs
tail -f backend/endpoints/app.log

# Docker container logs
docker logs -f multiscreen_group_<id>

# System logs
journalctl -u docker -f
```

2. Performance Metrics

```
# Resource usage
docker stats

# Network statistics
ss -tuln | grep :5000
netstat -i

# Storage usage
df -h
du -sh uploads/
```

3. Health Checks

```
# Application health
curl http://localhost:5000/api/health

# Docker health
docker ps --filter health=unhealthy

# Service status
systemctl status multiscreen-*
```

Conclusion

This documentation provides a comprehensive guide for developers working on the Multi-Screen SRT Streaming System. The modular architecture, comprehensive error handling, and detailed API documentation enable efficient development and maintenance of this complex streaming solution.

For additional support or questions, refer to:

- API endpoint documentation in the codebase
- Error code reference in `Errors.md`
- Individual service documentation in `backend/services/`
- Component documentation in `frontend/src/components/`

The system is designed for scalability, maintainability, and extensibility, supporting future enhancements while maintaining robust operation in production environments.