Dashboard Component Technical Documentation

Overview

The Dashboard component is the central piece of the Message Processing Dashboard application. It provides a real-time interface for monitoring file processing operations, displaying system metrics, and managing message data. This documentation covers the complete implementation and functionality of the Dashboard component.

Core State Management

Let's examine the state management implementation:

```
const [messages, setMessages] = useState([]);
const [loading, setLoading] = useState(true);
const [isConnected, setIsConnected] = useState(false);
const ws = useRef();
const [expandedContentIds, setExpandedContentIds] = useState([]);
const [searchTerm, setSearchTerm] = useState('');
const [currentPage, setCurrentPage] = useState(1);
const [itemsPerPage, setItemsPerPage] = useState(10);
const [selectedContentType, setSelectedContentType] = useState('All');
const [showAnalytics, setShowAnalytics] = useState(false);
```

This state management setup establishes several critical aspects of the dashboard:

1. **Message Management**
   - `messages`: Holds all processed messages in the system
   - `loading`: Controls loading state during data fetches
   - `isConnected`: Tracks WebSocket connection status
   - `ws`: Maintains a reference to the WebSocket connection
2. **UI Control**
   - `expandedContentIds`: Tracks which message groups are expanded
   - `searchTerm`: Manages the current search filter
   - `currentPage` and `itemsPerPage`: Handle pagination
   - `selectedContentType`: Controls content type filtering
   - `showAnalytics`: Toggles the analytics panel

Performance Monitoring State

```
const [performanceStats, setPerformanceStats] = useState({
  averageResponseTime: 0,
  cpuUtilization: 0,
  memoryUsage: 0,
```

```
    currentLoad: 0,
    uptime: 0,
    networkStats: {
        bytesSent: 0,
        bytesReceived: 0,
        activeConnections: 0,
        messageRate: 0,
    },
});
```

The performance monitoring system tracks:
- System response times
- Resource utilization
- Network performance
- Message processing rates
- System uptime

Each metric serves a specific purpose:
- `averageResponseTime`: Measures message processing efficiency
- `cpuUtilization` and `memoryUsage`: Monitor system resource usage
- `networkStats`: Tracks communication efficiency and system load

File Statistics Management

```
const [fileStats, setFileStats] = useState({
    totalFilesProcessed: 0,
    fileTypeDistribution: {
        Document: 0,
        Image: 0,
        Audio: 0,
    },
});
```

This state tracks:
- Total number of processed files
- Distribution across different file types
- Processing statistics by category
- Overall system throughput

System Health Monitoring

```
const [systemHealth, setSystemHealth] = useState({
    activeConnections: isConnected ? 1 : 0,
```

```
    queueDepth: 0,
    memoryUsage: 0,
    successRate: 100,
});
```

The system health monitoring:
- Tracks active connections
- Monitors queue depth
- Measures memory usage
- Calculates processing success rates

WebSocket Connection Management

```
useEffect(() => {
    const connectWebSocket = () => {
        ws.current = new WebSocket('ws://localhost:5001');
        ws.current.onopen = () => {
            console.log('WebSocket connected');
            setIsConnected(true);
        };
        ws.current.onmessage = (event) => {
            try {
                const data = JSON.parse(event.data);
                console.log('WebSocket data received:', data);
                if (data.type === 'initialMessages') {
                    setMessages([...data.data]);
                    setLoading(false);
                } else if (data.type === 'newMessage') {
                    setMessages((prevMessages) => [data.data, ...prevMessages]);
                } else if (data.type === 'analytics') {
                    setPerformanceStats((prevStats) => ({
                        ...prevStats,
                        ...data.data.performanceStats,
                    }));
                }
            } catch (err) {
                console.error('Failed to parse WebSocket message:', err);
            }
        };
        ws.current.onerror = (error) => {
            console.error('WebSocket error:', error);
            setIsConnected(false);
        };
        ws.current.onclose = () => {
```

```
            console.log('WebSocket disconnected');
            setIsConnected(false);
            setTimeout(connectWebSocket, 3000);
        };
    };
    connectWebSocket();
    return () => {
        if (ws.current) {
            ws.current.close();
        }
    };
}, []);
```

The WebSocket connection management system handles:
1. **Connection Establishment**
   - Creates a new WebSocket connection
   - Sets up event handlers
   - Manages connection state

2. **Message Processing**
   - Handles initial message load
   - Processes real-time updates
   - Manages analytics data
   - Implements error handling

3. **Connection Recovery**
   - Detects connection losses
   - Implements automatic reconnection
   - Provides error feedback
   - Maintains system stability
4. **Resource Cleanup**

   - Handles component unmounting
   - Closes connections properly
   - Prevents memory leaks

Message Processing Functions

Let's examine the core message processing functions:

```
const toggleExpandContentId = (contentId) => {
    setExpandedContentIds((prevExpanded) =>
        prevExpanded.includes(contentId)
            ? prevExpanded.filter((id) => id !== contentId)
```

```
        : [...prevExpanded, contentId]
    );
};
const truncateId = (id) => {
    if (!id) return '';
    return id.length > 10 ? `${id.slice(0, 5)}...${id.slice(-5)}` : id;
};
const sanitizeClassName = (text) => {
    return text?.toLowerCase().replace(/\s+/g, '-').replace(/[^\w-]/g, '');
};
```

These functions serve specific purposes:
1. **toggleExpandContentId**
   - Manages expandable content sections
   - Maintains state for expanded items
   - Implements toggle functionality
   - Ensures smooth UI transitions

2. **truncateId**
   - Formats long identifiers for display
   - Maintains readability
   - Ensures consistent presentation
   - Handles edge cases

3. **sanitizeClassName**
   - Processes text for CSS classes
   - Ensures valid class names
   - Maintains consistency
   - Handles special characters

Message Filtering and Data Organization

```
const groupedMessages = messages.reduce((acc, msg) => {
    const contentId = msg.content_id;
    if (!acc[contentId]) {
        acc[contentId] = [];
    }
    acc[contentId].push(msg);
    return acc;
}, {});
const sortedGroupedMessages = Object.entries(groupedMessages).sort(
    (a, b) => {
        const getLastTimestamp = (msgs) => {
            return Math.max(
```

```
            ...msgs.map((msg) => new Date(msg.time).getTime() || 0)
        );
    };
    return getLastTimestamp(b[1]) - getLastTimestamp(a[1]);
  }
);
```

This message organization system implements several critical features:
1. **Message Grouping**
   - Groups related messages by content ID
   - Maintains processing order within groups
   - Creates hierarchical message structure
   - Enables efficient message management

2. **Timestamp-based Sorting**
   - Sorts message groups by latest timestamp
   - Ensures chronological message display
   - Handles missing or invalid timestamps
   - Maintains consistent ordering

The filtering system extends this organization:

```
const filteredContent = sortedGroupedMessages.filter(([contentId, msgs]) => {
  const matchesSearch =
    contentId.toLowerCase().includes(searchTerm.toLowerCase()) ||
    msgs.some((msg) =>
      Object.values(msg).some((value) =>
        value?.toString().toLowerCase().includes(searchTerm.toLowerCase())
      )
    );
  const matchesContentType =
    selectedContentType === 'All' ||
    msgs.some((msg) => msg.content_type === selectedContentType);
  return matchesSearch && matchesContentType;
});
```

This filtering implementation provides:
1. **Search Functionality**
   - Case-insensitive search across all fields
   - Handles nested object structures
   - Provides real-time filtering
   - Maintains grouping integrity

2. **Content Type Filtering**
   - Filters by document type
   - Preserves group relationships
   - Handles multiple content types
   - Supports "All" type selection

Pagination Implementation

```
const indexOfLastContent = currentPage * itemsPerPage;
const indexOfFirstContent = indexOfLastContent - itemsPerPage;
const currentContent = filteredContent.slice(
    indexOfFirstContent,
    indexOfLastContent
);
const totalPages = Math.ceil(filteredContent.length / itemsPerPage);
useEffect(() => {
    setCurrentPage(1);
}, [searchTerm, selectedContentType, itemsPerPage]);
```

The pagination system manages:
1. **Page Calculation**
   - Determines content slice for current page
   - Calculates total page count
   - Handles edge cases
   - Maintains consistent page size

2. **Navigation Management**
   - Resets to first page on filter changes
   - Handles page size adjustments
   - Prevents invalid page states
   - Ensures smooth transitions

Analytics Management

```
const requestAnalytics = () => {
    if (ws.current?.readyState === WebSocket.OPEN) {
        ws.current.send(JSON.stringify({ type: 'getAnalytics' }));
    }
};
const handleAnalyticsClick = () => {
    if (ws.current?.readyState === WebSocket.OPEN) {
        ws.current.send(JSON.stringify({ type: 'getAnalytics' }));
```

```
    }
    setShowAnalytics(!showAnalytics);
};
useEffect(() => {
    if (showAnalytics) {
        requestAnalytics();
        const interval = setInterval(requestAnalytics, 5000);
        return () => clearInterval(interval);
    }
}, [showAnalytics]);
```

The analytics system provides:
1. **Data Collection**
   - Regular analytics updates
   - Real-time performance monitoring
   - Automatic refresh mechanism
   - Connection state verification

2. **Display Management**
   - Toggle analytics visibility
   - Handle data updates
   - Manage refresh intervals
   - Clean up on panel close

Statistics Updates

```
useEffect(() => {
    const totalMessages = messages.length;
    const documentMessages = messages.filter((m) => m.content_type === 'Document').length;
    const imageMessages = messages.filter((m) => m.content_type === 'Image' ||
m.content_type === 'Picture').length;
    const audioMessages = messages.filter((m) => m.content_type === 'Audio').length;
    setFileStats({
        totalFilesProcessed: totalMessages,
        fileTypeDistribution: {
            Document: documentMessages,
            Image: imageMessages,
            Audio: audioMessages,
        },
    });
    const queuedMessages = messages.filter((m) => m.status === 'Queued').length;
    const processedMessages = messages.filter((m) => m.status === 'Processed').length;
    const successRate = totalMessages > 0 ? (processedMessages / totalMessages) * 100 : 100;
    setSystemHealth({
```

```
        activeConnections: isConnected ? 1 : 0,
        queueDepth: queuedMessages,
        memoryUsage: (totalMessages * 1024) / (1024 * 1024),
        successRate: successRate,
    });
}, [messages, isConnected]);
```

This statistics update system handles:
1. **Message Counting**
   - Tracks total message count
   - Categorizes by content type
   - Monitors processing status
   - Updates file statistics

2. **System Health Calculations**
   - Calculates success rates
   - Monitors queue depth
   - Tracks memory usage
   - Updates connection status

UI Rendering Implementation

The UI rendering system includes several specialized components:

MetricCard Component

```
const MetricCard = ({ title, value }) => (
    <div className="metric-card">
        <h4>{title}</h4>
        <p>{value}</p>
    </div>
);
```

This reusable component:
- Displays individual metrics
- Maintains consistent styling
- Handles various data types
- Provides clear visual hierarchy

Table Rendering

```
{currentContent.map(([contentId, msgs]) => (
    <React.Fragment key={contentId}>
        <tr
```

```
            className="content-row"
            onClick={() => toggleExpandContentId(contentId)}
        >
            <td className="id-cell" title={contentId}>
                {expandedContentIds.includes(contentId)
                    ? contentId
                    : truncateId(contentId)}
            </td>
            <td colSpan="7" className="expand-cell">
                {expandedContentIds.includes(contentId) ? (
                    <ExpandLess />
                ) : (
                    <ExpandMore />
                )}
            </td>
        </tr>
        {expandedContentIds.includes(contentId) && (
            // Expanded row content implementation
        )}
    </React.Fragment>
))}
```

The table implementation provides:
1. **Dynamic Content Display**
   - Expandable row sections
   - Content grouping
   - Interactive elements
   - Responsive layout

2. **Data Presentation**
   - Formatted content display
   - Status indicators
   - Type-specific styling
   - Truncated ID display

Analytics Panel

```
{showAnalytics && (
    <div className="analytics-panel">
        <div className="analytics-section">
            <h3>System Performance</h3>
            <div className="metrics-grid">
                <MetricCard
```

```
                    title="CPU Usage"
                    value={`${performanceStats.cpuUtilization}%`}
                />
                // Additional metrics
            </div>
        </div>
        // Additional sections
    </div>
)}
```

The analytics panel implements:
1. **Performance Metrics Display**
   - CPU and memory usage
   - Network statistics
   - System load information
   - Response time metrics

2. **File Processing Statistics**
   - File type distribution
   - Processing success rates
   - Queue status
   - Throughout metrics

Error Handling and Edge Cases

The Dashboard component implements comprehensive error handling:
1. **WebSocket Errors**
   - Connection failures
   - Message parsing errors
   - State synchronization issues
   - Recovery mechanisms

2. **Data Validation**
   - Message format verification
   - Type checking
   - Null/undefined handling
   - Edge case management
3. **UI State Management**
   - Invalid page numbers
   - Empty search results
   - Loading states
   - Connection status updates

Performance Optimizations

The component implements several optimization strategies:
1. **Efficient Rendering**
   - Memoized calculations
   - Conditional rendering
   - Batched updates
   - Optimized filtering
2. **Resource Management**
   - Controlled re-renders
   - Cleanup operations
   - Memory usage optimization
   - Connection management

Integration Points

The Dashboard component integrates with several system components:
1. **WebSocket Server**
   - Real-time updates
   - Analytics requests
   - Connection management
   - Error handling
2. **State Management**
   - Message processing
   - Analytics tracking
   - UI state control
   - Filter management

Backend Systems Technical Documentation

WebSocket Server Implementation

The WebSocket server forms the backbone of our real-time communication system. Let's examine its core components and functionality in detail.

Core Server Initialization

The server initializes with critical components that manage its operation:

```
class WebSocketServer:
    def __init__(self):
        self.connected_clients = set()
        self.db_handler = DBHandler()
        self.db_handler.init_db()
        self.message_queue = asyncio.Queue()
        self.start_time = datetime.now()
```

This initialization establishes several key elements. The connected_clients set tracks all active client connections, enabling efficient broadcast operations and connection management. The database handler provides persistent storage capabilities, while the asynchronous message queue manages message flow through the system. The start_time timestamp enables uptime tracking and performance metrics.

Message Format Handling

The server implements sophisticated message format conversion to ensure data consistency:

```
def convert_bson_to_json(self, data):
    if isinstance(data, bytes):
        return data.decode('utf-8', errors='replace')
    elif isinstance(data, ObjectId):
        return str(data)
    elif isinstance(data, dict):
        if 'ID' in data:
            return {
                'time': data.get('time', datetime.now().strftime('%m/%d/%Y, %I:%M:%S %p')),
                'job_id': data.get('ID'),
                'content_id': data.get('content_id'),
                'content_type': self._determine_content_type(data),
                'media_id': data.get('DocumentId') or data.get('PictureID') or data.get('AudioID') or
data.get('VideoID'),
                'file_name': data.get('file_name'),
```

```
            'status': 'Processed',
            'message': self._generate_message(data)
        }
```

This conversion system handles multiple data types and formats. It converts binary data to UTF-8 strings, transforms MongoDB ObjectIds to string format, and processes complex dictionary structures. The system standardizes message formats, ensuring consistent data structure throughout the application.

Content Type Management

The content type system provides intelligent file type detection and message generation:

```
def _determine_content_type(self, data):
    if 'DocumentId' in data:
        return 'Document'
    elif 'PictureID' in data:
        return 'Picture'
    elif 'AudioID' in data:
        return 'Audio'
    return data.get('content_type', 'Unknown Type')
def _generate_message(self, data):
    file_name = data.get('file_name', 'unknown file')
    if 'DocumentId' in data:
        return f"Document file '{file_name}' was successfully processed"
    elif 'PictureID' in data:
        return f"Picture file '{file_name}' was successfully processed"
    elif 'AudioID' in data:
        return f"Audio file '{file_name}' was successfully processed"
    return 'No additional information'
```

These functions work together to identify file types and generate appropriate status messages. The type determination examines specific ID fields to categorize content accurately, while the message generator creates context-aware status updates for each processed file.

Client Connection Management

The connection management system handles the complete lifecycle of client connections:

```
async def handle_client(self, websocket, path):
    try:
        self.connected_clients.add(websocket)
        print(f"Client connected. Total clients: {len(self.connected_clients)}")
        await self.send_initial_messages(websocket)
```

```
    broadcast_task = asyncio.create_task(self.broadcast_messages(websocket))
    async for message in websocket:
        try:
            message_data = json.loads(message)
            if message_data.get('type') == 'getAnalytics':
                analytics_data = await self.get_analytics_data()
                await websocket.send(json.dumps({
                    'type': 'analytics',
                    'data': analytics_data
                }))
                Continue
```

This system manages client connections, handles message routing, and maintains connection state. It implements initial data loading, real-time updates, and analytics data transmission, ensuring smooth communication between server and clients.

RabbitMQ Integration

The RabbitMQ integration provides robust message queue management:

```
async def consume_rabbitmq(self):
  def callback(ch, method, properties, body):
    try:
        message = BSON(body).decode()
        json_message = self.convert_bson_to_json(message)

        self.db_handler.save_message_to_db(json_message)

        asyncio.run_coroutine_threadsafe(
            self.message_queue.put({
                'type': 'newMessage',
                'data': json_message
            }),
            self.loop
        )
```

This implementation handles message consumption from RabbitMQ, processes messages through the conversion pipeline, persists data to the database, and broadcasts updates to connected clients. It manages queue connections, handles message decoding, and ensures reliable message delivery.

Server.js Implementation

The Express server implementation provides the web application framework:

```
const express = require('express');
const http = require('http');
const socketIo = require('socket.io');
const mongoose = require('mongoose');
const messageSchema = new mongoose.Schema({
    time: String,
    job_id: String,
    content_id: String,
    content_type: String,
    file_name: String,
    status: String,
    message: String
}, { timestamps: true });
```

This setup establishes the core server infrastructure, including the Express application, HTTP server, Socket.IO integration, and MongoDB connection. The message schema defines the structure for stored messages, including automatic timestamp management.

Database Integration

The MongoDB integration provides persistent data storage:

```
mongoose.connect('mongodb://localhost:27017/dashboard_db')
    .then(() => console.log('Connected to MongoDB'))
    .catch(err => console.error('MongoDB connection error:', err));
io.on('connection', (socket) => {
    Message.find({})
        .sort({ createdAt: -1 })
        .limit(100)
        .then((messages) => {
            socket.emit('initialMessages', messages);
        })
        .catch((err) => console.error('Error fetching messages:', err));
```

This system manages database connections, implements data operations, and handles error conditions. It provides message storage and retrieval capabilities, with built-in sorting and limiting functionality.

Socket Event Handling

The Socket.IO event system manages real-time communication:

```
socket.on('newMessage', (msg) => {
```

```
    const newMessage = new Message(msg);
    newMessage.save()
        .then(() => {
            io.emit('newMessage', msg);
            console.log('New message saved and broadcasted');
        })
        .catch((err) => console.error('Error saving new message:', err));
});
socket.on('disconnect', () => {
    console.log('Client disconnected');
});
```

This implementation handles message processing, database persistence, and client notifications. It manages new message creation, broadcasts updates to connected clients, and handles client disconnections.

Error Handling and Recovery

Both systems implement comprehensive error handling strategies:

1. The WebSocket server handles connection failures, message parsing errors, database errors, and queue-related issues.
2. The Express server manages MongoDB connection problems, Socket.IO errors, message processing failures, and resource allocation issues.

The error handling system ensures robust operation and graceful recovery from various failure conditions.