Retro Gaming Catalog with RabbitMQ

A Docker-based Microservices Architecture

Docker Message-Oriented Architecture Project

April 17, 2025

Abstract

This report presents a comprehensive overview of the Retro Gaming Catalog application, which implements a message-oriented architecture using Docker containers and RabbitMQ. The system consists of a frontend, backend API, database, and Python-based microservices that process images and enrich metadata asynchronously. The report details the architecture, Docker configuration, message flow, and resilience features of the application.

Contents

1 Introduction

The Retro Gaming Catalog is a web application that allows users to browse, add, edit, and delete retro video games. The application follows a microservices architecture pattern and implements asynchronous processing using RabbitMQ message queues. This approach demonstrates how to decouple time-intensive operations from the main request-response cycle, improving user experience and system scalability.

1.1 Project Objectives

The main objectives of this project are:

- Implement a producer/consumer pattern using RabbitMQ
- Demonstrate containerization of a multi-service application using Docker
- Showcase asynchronous processing for time-intensive operations
- Create a responsive web application with real-time status updates
- Implement resilience features for robust operation

1.2 System Architecture

The application consists of the following components:

- Frontend: HTML, CSS, and JavaScript served by Nginx
- Backend API: Node.js with Express
- Database: MySQL
- Message Broker: RabbitMQ
- Image Processor: Python service for image processing
- Metadata Enricher: Python service for metadata enrichment

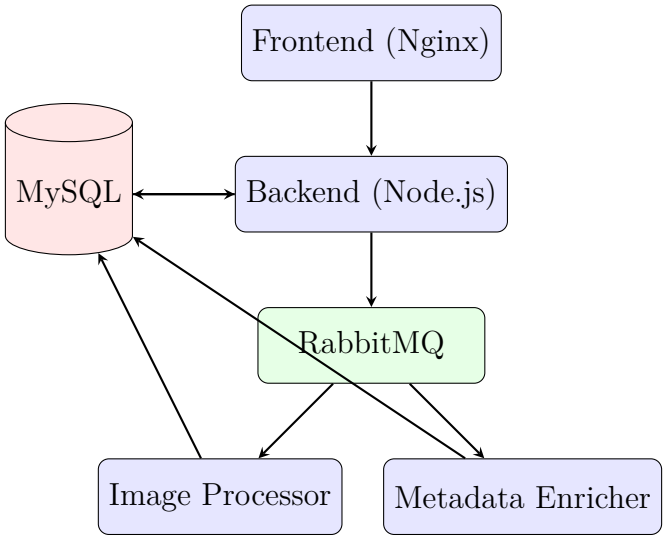

Figure 1: System Architecture Diagram

2 Docker Configuration

The application is containerized using Docker, with Docker Compose orchestrating the multi-container setup. This section details the Docker configuration for each service.

2.1 Docker Compose

The docker-compose.yml file defines the services, networks, and volumes for the application:

```
version: '3.8'

services:
  # Frontend service
  frontend:
    build:
      context: ./frontend
      dockerfile: Dockerfile
    ports:
      - "80:80"
    depends_on:
      backend:
        condition: service_healthy
    networks:
      - app-network
    restart: unless-stopped
```

```
17
18  # Backend service
19  backend:
20    build:
21      context: ./backend
22      dockerfile: Dockerfile
23    ports:
24      - "3000:3000"
25    depends_on:
26      database:
27        condition: service_healthy
28      rabbitmq:
29        condition: service_healthy
30    environment:
31      - DB_HOST=database
32      - DB_USER=retro_user
33      - DB_PASSWORD=retro_password
34      - DB_NAME=retro_games_catalog
35      - PORT=3000
36      - RABBITMQ_URL=amqp://guest:guest@rabbitmq:5672
37    networks:
38      - app-network
39    restart: unless-stopped
40    healthcheck:
41      test: ["CMD", "wget", "--no-verbose", "--tries=1", "--
     spider", "http://localhost:3000/api/platforms"]
42      interval: 10s
43      timeout: 5s
44      retries: 5
45      start_period: 15s
46
47  # Database service
48  database:
49    build:
50      context: ./database
51      dockerfile: Dockerfile
52    ports:
53      - "3307:3306"
54    environment:
55      - MYSQL_ROOT_PASSWORD=retro_password
56      - MYSQL_DATABASE=retro_games_catalog
57      - MYSQL_USER=retro_user
58      - MYSQL_PASSWORD=retro_password
59    volumes:
60      - mysql-data:/var/lib/mysql
61    networks:
62      - app-network
63    restart: unless-stopped
64    healthcheck:
```

```
65      test: ["CMD", "mysqladmin", "ping", "-h", "localhost",
    "-u", "root", "-pretro_password"]
66      interval: 10s
67      timeout: 5s
68      retries: 5
69      start_period: 30s
70
71  # RabbitMQ service
72  rabbitmq:
73    image: rabbitmq:3-management
74    ports:
75      - "5672:5672"   # AMQP protocol port
76      - "15672:15672" # Management UI port
77    environment:
78      - RABBITMQ_DEFAULT_USER=guest
79      - RABBITMQ_DEFAULT_PASS=guest
80    volumes:
81      - rabbitmq-data:/var/lib/rabbitmq
82    networks:
83      - app-network
84    restart: unless-stopped
85    healthcheck:
86      test: ["CMD", "rabbitmq-diagnostics", "
    check_port_connectivity"]
87      interval: 10s
88      timeout: 5s
89      retries: 5
90      start_period: 30s
91
92  # Image processor service (Python consumer)
93  image-processor:
94    build:
95      context: ./image-processor
96      dockerfile: Dockerfile
97    depends_on:
98      rabbitmq:
99        condition: service_healthy
100     backend:
101       condition: service_healthy
102   environment:
103     - RABBITMQ_URL=amqp://guest:guest@rabbitmq:5672
104     - BACKEND_URL=http://backend:3000
105   volumes:
106     - processed-images:/app/processed
107   networks:
108     - app-network
109   restart: unless-stopped
110
111 # Metadata enricher service (Python consumer)
```

```
  metadata-enricher:
    build:
      context: ./metadata-enricher
      dockerfile: Dockerfile
    depends_on:
      rabbitmq:
        condition: service_healthy
      backend:
        condition: service_healthy
    environment:
      - RABBITMQ_URL=amqp://guest:guest@rabbitmq:5672
      - BACKEND_URL=http://backend:3000
    networks:
      - app-network
    restart: unless-stopped

# Define networks
networks:
  app-network:
    driver: bridge

# Define volumes
volumes:
  mysql-data:
    driver: local
  rabbitmq-data:
    driver: local
  processed-images:
    driver: local
```

Listing 1: docker-compose.yml

Key features of the Docker Compose configuration:

- Health checks for critical services
- Dependency management with condition-based startup
- Persistent volumes for data storage
- Shared network for inter-service communication
- Environment variable configuration
- Port mapping for external access

2.2 Dockerfiles

2.2.1 Frontend Dockerfile

```
FROM nginx:alpine

# Copy static files
COPY . /usr/share/nginx/html/

# Copy nginx configuration
COPY nginx.conf /etc/nginx/conf.d/default.conf

# Expose port
EXPOSE 80

# Start nginx
CMD ["nginx", "-g", "daemon off;"]
```

Listing 2: frontend/Dockerfile

The frontend Dockerfile uses the official Nginx Alpine image, copies the static files and Nginx configuration, and exposes port 80 for HTTP traffic.

2.2.2 Backend Dockerfile

```
FROM node:16-alpine

WORKDIR /app

# Copy package.json and package-lock.json
COPY package*.json ./

# Install dependencies
RUN npm install

# Copy application code
COPY . .

# Create directory for processed images
RUN mkdir -p /app/processed

# Expose port
EXPOSE 3000

# Start the application
CMD ["node", "server.js"]
```

Listing 3: backend/Dockerfile

The backend Dockerfile uses the Node.js Alpine image, installs dependencies, copies the application code, creates a directory for processed images, and exposes port 3000 for the API.

2.2.3 Database Dockerfile

```
FROM mysql:8.0

# Copy initialization script
COPY retro_games.sql /docker-entrypoint-initdb.d/

# Set MySQL configuration
COPY my.cnf /etc/mysql/conf.d/

# Expose port
EXPOSE 3306
```

Listing 4: database/Dockerfile

The database Dockerfile uses the official MySQL 8.0 image, copies the initialization script and MySQL configuration, and exposes port 3306 for database connections.

2.2.4 Image Processor Dockerfile

```
FROM python:3.9-slim

WORKDIR /app

# Install system dependencies for Pillow
RUN apt-get update && apt-get install -y \
    gcc \
    libjpeg-dev \
    zlib1g-dev \
    && rm -rf /var/lib/apt/lists/*

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

# Copy application code
COPY . .

# Create directory for processed images
RUN mkdir -p /app/processed

# Run the application
CMD ["python", "app.py"]
```

Listing 5: image-processor/Dockerfile

The image processor Dockerfile uses the Python 3.9 slim image, installs system dependencies for the Pillow library, installs Python dependencies,

copies the application code, creates a directory for processed images, and runs the Python application.

2.2.5 Metadata Enricher Dockerfile

```
FROM python:3.9-slim

WORKDIR /app

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

# Copy application code
COPY . .

# Run the application
CMD ["python", "app.py"]
```

Listing 6: metadata-enricher/Dockerfile

The metadata enricher Dockerfile uses the Python 3.9 slim image, installs Python dependencies, copies the application code, and runs the Python application.

3 Message-Oriented Architecture with RabbitMQ

The application implements a message-oriented architecture using RabbitMQ as the message broker. This section details the implementation of the producer/consumer pattern.

3.1 RabbitMQ Configuration

RabbitMQ is configured with two queues:

- **image_processing**: Queue for image processing tasks
- **metadata_enrichment**: Queue for metadata enrichment tasks

Both queues are configured with durability to ensure messages survive broker restarts.

3.2 Producer Implementation (Node.js Backend)

The backend server acts as the producer, sending messages to RabbitMQ queues when games are added or updated:

```
// RabbitMQ connection with retry
async function connectToRabbitMQ(retries = 5, delay = 5000) {
  let lastError = null;

  for (let attempt = 1; attempt <= retries; attempt++) {
    try {
      console.log(`Attempting RabbitMQ connection (attempt ${attempt}/${retries})...`);

      rabbitConnection = await amqp.connect(RABBITMQ_URL);
      rabbitChannel = await rabbitConnection.createChannel();

      // Declare queues with durability
      await rabbitChannel.assertQueue("image_processing", { durable: true });
      await rabbitChannel.assertQueue("metadata_enrichment", { durable: true });

      console.log("RabbitMQ connection successful");

      // Set up connection error handling
      rabbitConnection.on("error", (err) => {
        console.error("RabbitMQ connection error:", err);
        setTimeout(() => connectToRabbitMQ(), 5000);
      });

      rabbitConnection.on("close", () => {
        console.log("RabbitMQ connection closed, attempting to reconnect...");
        setTimeout(() => connectToRabbitMQ(), 5000);
      });

      return { connection: rabbitConnection, channel: rabbitChannel };
    } catch (error) {
      lastError = error;
      console.error(`RabbitMQ connection failed (attempt ${attempt}/${retries}):`, error);

      if (attempt < retries) {
        console.log(`Retrying in ${delay / 1000} seconds...`);
        await new Promise((resolve) => setTimeout(resolve, delay));
```

```
      }
    }
  }

  console.error(`Failed to connect to RabbitMQ after ${retries} attempts`);
  // Don't throw error, continue without RabbitMQ
  console.log("Continuing without RabbitMQ...");
  return { connection: null, channel: null };
}

// Function to send message to RabbitMQ
async function sendToQueue(queue, message) {
  try {
    if (!rabbitChannel) {
      console.error("RabbitMQ channel not available");
      return false;
    }

    const success = rabbitChannel.sendToQueue(
      queue,
      Buffer.from(JSON.stringify(message)),
      { persistent: true }, // Message will survive broker restarts
    );

    if (success) {
      console.log(`Message sent to queue ${queue}:`, message);
      return true;
    } else {
      console.error(`Failed to send message to queue ${queue}`);
      return false;
    }
  } catch (error) {
    console.error(`Error sending message to queue ${queue}:`, error);
    return false;
  }
}

// Example of sending messages when a game is created
app.post("/api/games", async (req, res) => {
  try {
    // ... (validation and database insertion)

    const newGameId = result.insertId;

```

```
    // Send messages to RabbitMQ for background processing
    if (newGameId && rabbitChannel) {
      // Send image processing message
      sendToQueue("image_processing", {
        gameId: newGameId,
        imageUrl: imageUrl || null,
      });

      // Send metadata enrichment message
      sendToQueue("metadata_enrichment", {
        gameId: newGameId,
        title: title,
        developer: developer,
      });
    }

    // ... (response handling)
  } catch (error) {
    // ... (error handling)
  }
});
```

Listing 7: backend/server.js (Producer Implementation)

3.3 Consumer Implementation (Python Services)

3.3.1 Image Processor Consumer

```
def callback(ch, method, properties, body):
    """
    Process messages from the queue
    """
    try:
        # Parse the message
        message = json.loads(body)
        logger.info(f"Received message: {message}")

        game_id = message.get('gameId')
        image_url = message.get('imageUrl')

        if not game_id:
            logger.error("Message missing gameId, cannot
   process")
            ch.basic_ack(delivery_tag=method.delivery_tag)
            return

        # Simulate processing time (1-5 seconds)
        processing_time = random.uniform(1, 5)
```

```
20          logger.info(f"Processing will take approximately {
    processing_time:.2f} seconds")
21          time.sleep(processing_time)
22
23          # Process the image
24          processed_paths = process_image(image_url, game_id)
25
26          # Update the game with processed image paths
27          update_game_with_processed_images(game_id,
    processed_paths)
28
29          # Acknowledge the message
30          ch.basic_ack(delivery_tag=method.delivery_tag)
31          logger.info(f"Processing complete for game {game_id}"
    )
32
33      except json.JSONDecodeError:
34          logger.error(f"Invalid JSON in message: {body}")
35          # Acknowledge the message to remove it from the queue
36          ch.basic_ack(delivery_tag=method.delivery_tag)
37
38      except Exception as e:
39          logger.error(f"Error processing message: {str(e)}")
40          logger.error(traceback.format_exc())
41
42          # Negative acknowledgment to requeue the message
43          ch.basic_nack(delivery_tag=method.delivery_tag,
    requeue=True)
44
45  def main():
46      """
47      Main function to set up the RabbitMQ consumer
48      """
49      # Connection retry loop
50      connection = None
51      max_retries = 10
52      retry_delay = 5  # seconds
53
54      for attempt in range(1, max_retries + 1):
55          try:
56              logger.info(f"Connecting to RabbitMQ at {
    RABBITMQ_URL} (attempt {attempt}/{max_retries})...")
57              connection = pika.BlockingConnection(pika.
    URLParameters(RABBITMQ_URL))
58              break
59          except pika.exceptions.AMQPConnectionError:
60              logger.warning(f"Failed to connect to RabbitMQ (
    attempt {attempt}/{max_retries})")
61              if attempt < max_retries:
```

```
                logger.info(f"Retrying in {retry_delay} seconds...")
                time.sleep(retry_delay)
            else:
                logger.error(f"Failed to connect to RabbitMQ after {max_retries} attempts")
                return

    if not connection:
        logger.error("Could not establish connection to RabbitMQ")
        return

    channel = connection.channel()

    # Declare the queue with durability
    channel.queue_declare(
        queue=QUEUE_NAME,
        durable=True,  # Queue will survive broker restarts
    )

    # Set prefetch count to 1 to ensure fair dispatch
    channel.basic_qos(prefetch_count=1)

    # Set up the consumer
    channel.basic_consume(
        queue=QUEUE_NAME,
        on_message_callback=callback
    )

    logger.info(f"Image processor started, waiting for messages on queue '{QUEUE_NAME}'")

    try:
        channel.start_consuming()
    except KeyboardInterrupt:
        logger.info("Interrupted by user, shutting down")
        channel.stop_consuming()
    except Exception as e:
        logger.error(f"Unexpected error: {str(e)}")
        logger.error(traceback.format_exc())
    finally:
        if connection and connection.is_open:
            connection.close()
            logger.info("Connection closed")
```

Listing 8: image-processor/app.py (Consumer Implementation)

3.3.2 Metadata Enricher Consumer

The metadata enricher consumer follows a similar pattern to the image processor consumer, with specific logic for enriching game metadata.

3.4 Message Flow

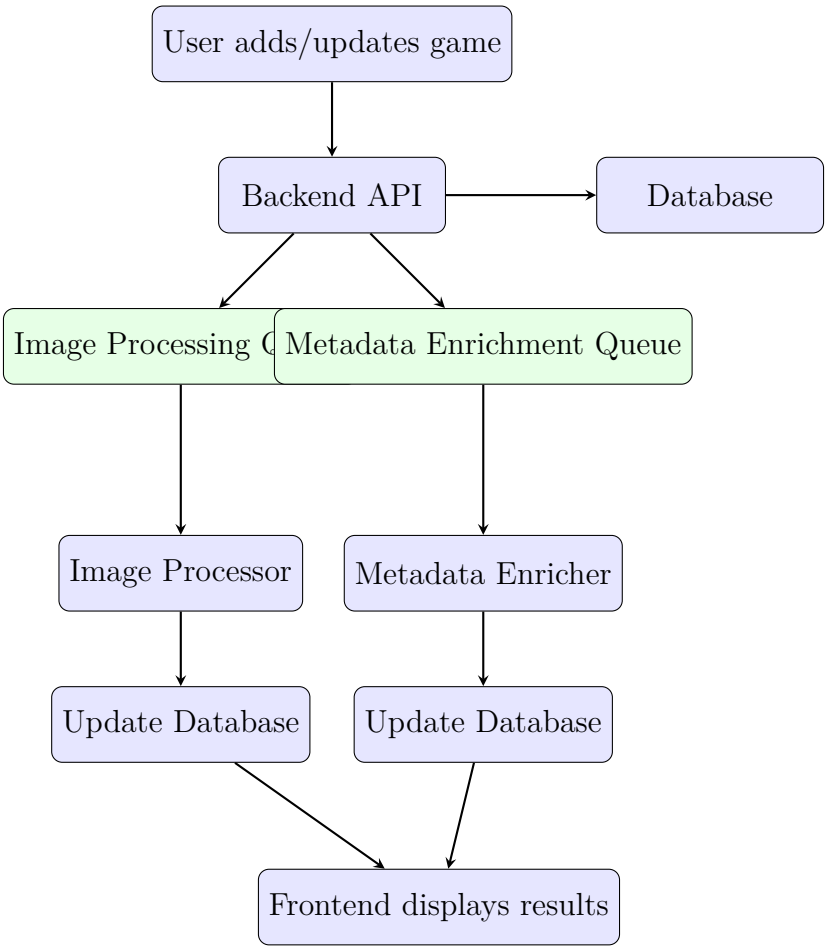

Figure 2: Message Flow Diagram

The message flow in the application follows these steps:

1. User adds or updates a game through the frontend
2. Backend API validates the request and saves the game to the database

3. Backend sends messages to RabbitMQ queues for background processing
4. Python consumers receive messages and process them asynchronously
5. After processing, consumers update the database with the results
6. Frontend displays the processing status and results to the user

4 Database Schema

The database schema consists of three main tables:

- **platforms**: Stores information about gaming platforms
- **genres**: Stores information about game genres
- **games**: Stores information about games, including processing status

```
-- Create database
CREATE DATABASE IF NOT EXISTS retro_games_catalog;
USE retro_games_catalog;

-- Create platforms table
CREATE TABLE IF NOT EXISTS platforms (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(50) NOT NULL UNIQUE,
    manufacturer VARCHAR(50),
    release_year INT,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE
    CURRENT_TIMESTAMP
);

-- Create genres table
CREATE TABLE IF NOT EXISTS genres (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(50) NOT NULL UNIQUE,
    description TEXT,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE
    CURRENT_TIMESTAMP
);

-- Create games table with additional fields for RabbitMQ
    processing
```

```
CREATE TABLE IF NOT EXISTS games (
    id INT AUTO_INCREMENT PRIMARY KEY,
    title VARCHAR(100) NOT NULL,
    platform_id INT NOT NULL,
    genre_id INT NOT NULL,
    developer VARCHAR(100) NOT NULL,
    release_year INT NOT NULL,
    year_category VARCHAR(20) NOT NULL,
    description TEXT NOT NULL,
    image_url VARCHAR(255),
    thumbnail_url VARCHAR(255),
    processing_status ENUM('pending', 'processing', 'completed', 'failed') DEFAULT 'completed',
    metadata_status ENUM('pending', 'processing', 'completed', 'failed') DEFAULT 'completed',
    average_rating DECIMAL(3,1),
    total_reviews INT,
    difficulty_level VARCHAR(50),
    estimated_play_time VARCHAR(50),
    tags JSON,
    fun_fact TEXT,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
    FOREIGN KEY (platform_id) REFERENCES platforms(id) ON DELETE CASCADE,
    FOREIGN KEY (genre_id) REFERENCES genres(id) ON DELETE CASCADE
);
```

Listing 9: database/retro$_g ames.sql(Schema)$

The games table includes fields for tracking the processing status of images and metadata, as well as fields for storing the enriched metadata.

5 Resilience Features

The application implements several resilience features to ensure robust operation:

5.1 Message Persistence

Messages in RabbitMQ queues are marked as persistent, ensuring they survive broker restarts:

```
const success = rabbitChannel.sendToQueue(
  queue,
```

```
  Buffer.from(JSON.stringify(message)),
  { persistent: true }, // Message will survive broker
   restarts
);
```

Listing 10: Message Persistence

5.2 Acknowledgment Mechanism

Consumers use acknowledgment to ensure messages are only removed from the queue after successful processing:

```
# Acknowledge the message
ch.basic_ack(delivery_tag=method.delivery_tag)

# Negative acknowledgment to requeue the message
ch.basic_nack(delivery_tag=method.delivery_tag, requeue=True)
```

Listing 11: Acknowledgment Mechanism

5.3 Connection Retry

Services implement connection retry logic with exponential backoff:

```
# Connection retry loop
connection = None
max_retries = 10
retry_delay = 5  # seconds

for attempt in range(1, max_retries + 1):
    try:
        logger.info(f"Connecting to RabbitMQ at {RABBITMQ_URL
    } (attempt {attempt}/{max_retries})...")
        connection = pika.BlockingConnection(pika.
    URLParameters(RABBITMQ_URL))
        break
    except pika.exceptions.AMQPConnectionError:
        logger.warning(f"Failed to connect to RabbitMQ (
    attempt {attempt}/{max_retries})")
        if attempt < max_retries:
            logger.info(f"Retrying in {retry_delay} seconds
    ...")
            time.sleep(retry_delay)
        else:
            logger.error(f"Failed to connect to RabbitMQ
    after {max_retries} attempts")
            return
```

Listing 12: Connection Retry

5.4 Health Checks

Docker Compose includes health checks for critical services:

```
healthcheck:
  test: ["CMD", "wget", "--no-verbose", "--tries=1", "--
   spider", "http://localhost:3000/api/platforms"]
  interval: 10s
  timeout: 5s
  retries: 5
  start_period: 15s
```

Listing 13: Health Checks

5.5 Graceful Degradation

The backend implements graceful degradation, continuing to operate with limited functionality when RabbitMQ is unavailable:

```
// Send messages to RabbitMQ for background processing
if (newGameId && rabbitChannel) {
  // Send image processing message
  sendToQueue("image_processing", {
    gameId: newGameId,
    imageUrl: imageUrl || null,
  });

  // Send metadata enrichment message
  sendToQueue("metadata_enrichment", {
    gameId: newGameId,
    title: title,
    developer: developer,
  });
} else {
  console.log("Skipping RabbitMQ message sending - channel
   not available");
}
```

Listing 14: Graceful Degradation

6 Frontend Implementation

The frontend is implemented using HTML, CSS, and JavaScript, with a retro-inspired design. It includes features for:

- Browsing games with filtering options

- Adding, editing, and deleting games
- Viewing game details, including processed images and enriched metadata and deleting games
- Viewing game details, including processed images and enriched metadata
- Monitoring background processing tasks

The frontend communicates with the backend API to fetch and update data, and displays real-time processing status to the user.

6.1 Processing Status Monitoring

A dedicated page allows users to monitor the status of background processing tasks:

```
<div class="task-list" id="task-list">
    <div class="loading">Loading tasks...</div>
</div>

<script>
    // API URL
    const API_URL = "http://localhost:3000/api";

    // Auto-refresh interval (in milliseconds)
    const REFRESH_INTERVAL = 5000;
    let refreshTimer = null;

    // Load tasks from API
    async function loadTasks() {
        try {
            const taskList = document.getElementById("task-
list");
            taskList.innerHTML = '<div class="loading">
Loading tasks...</div>';

            // Fetch tasks
            const response = await fetch(`${API_URL}/
processing`);
            if (!response.ok) throw new Error("Failed to load
 tasks");

            const tasks = await response.json();

            // Display tasks
            if (tasks.length === 0) {
```

```
27                taskList.innerHTML = '<div class="no-tasks">
  No tasks are currently being processed.</div>';
28                return;
29            }
30
31            taskList.innerHTML = '';
32
33            tasks.forEach(task => {
34                const taskCard = document.createElement("div"
  );
35                taskCard.className = "task-card";
36
37                // Create task ID element
38                const taskId = document.createElement("div");
39                taskId.className = "task-id";
40                taskId.textContent = `#${task.id}`;
41
42                // Create task title element
43                const taskTitle = document.createElement("div
  ");
44                taskTitle.className = "task-title";
45                taskTitle.textContent = task.title;
46
47                // Create image processing status badge
48                const imageStatus = document.createElement("
  div");
49                imageStatus.className = `status-badge status-
  ${task.processing_status}`;
50                imageStatus.textContent = `Image: ${
  capitalizeFirstLetter(task.processing_status)}`;
51
52                // Create metadata status badge
53                const metadataStatus = document.createElement
  ("div");
54                metadataStatus.className = `status-badge
  status-${task.metadata_status}`;
55                metadataStatus.textContent = `Metadata: ${
  capitalizeFirstLetter(task.metadata_status)}`;
56
57                // Add elements to task card
58                taskCard.appendChild(taskId);
59                taskCard.appendChild(taskTitle);
60                taskCard.appendChild(imageStatus);
61                taskCard.appendChild(metadataStatus);
62
63                // Add task card to list
64                taskList.appendChild(taskCard);
65            });
66
```

```
        } catch (error) {
            console.error("Error loading tasks:", error);
            document.getElementById("task-list").innerHTML =
                '<div class="no-tasks">Failed to load tasks.
    Please try again later.</div>';
            showToast("Failed to load tasks. Please try again
     later.", true);
        }
    }
</script>
```

Listing 15: frontend/processing.html (Excerpt)

7 Conclusion

The Retro Gaming Catalog application demonstrates the implementation of a message-oriented architecture using Docker containers and RabbitMQ. The application showcases several key concepts:

- Containerization of a multi-service application using Docker
- Asynchronous processing using RabbitMQ message queues
- Producer/consumer pattern for decoupling time-intensive operations
- Resilience features for robust operation
- Real-time status updates for background processing

The architecture provides several benefits:

- Improved user experience by offloading time-intensive operations
- Better scalability by distributing workloads across multiple services
- Enhanced reliability through message persistence and acknowledgment
- Easier maintenance and deployment through containerization

This project serves as a practical example of how to implement a message-oriented architecture in a real-world application, demonstrating best practices for containerization, asynchronous processing, and resilience.

8 Future Enhancements

Potential future enhancements for the application include:

- Implementing a circuit breaker pattern for more resilient service communication
- Adding a comprehensive monitoring dashboard for all services
- Implementing graceful degradation for all components
- Adding user authentication and authorization
- Implementing a more sophisticated image processing pipeline
- Adding real-time notifications using WebSockets