Kafka Integration for Standard Logger JS

This document describes the implementation of Kafka integration in the Standard Logger JS library.

Overview

The Kafka integration allows sending log messages to Apache Kafka, which is particularly useful in distributed systems where centralized log collection and processing is required. This feature enables:

- Sending structured logs to Kafka topics
- Filtering logs by level before sending
- Batching messages for better performance
- Configurable connection and authentication options
- Graceful connection handling

Implementation Details

1. Core Components

The integration consists of the following key components:

- **KafkaTransport**: A class that handles communication with Kafka brokers
- **KafkaTransportOptions**: Interface defining configuration options
- **Logger enhancement**: Methods added to the Logger class to configure and use Kafka

2. File Structure

```
src/
├── interfaces/
│   └── kafka-options.ts       # Defines configuration options
├── transports/
│   └── kafka-transport.ts     # Main implementation for Kafka transport
├── core/
│   ├── logger.ts              # Enhanced with Kafka support
│   └── kafka-pino-transport.ts # Optional Pino integration
```

3. Key Classes and Interfaces

KafkaTransportOptions

```typescript
export interface KafkaTransportOptions {
    // Connection options
    brokers: string[];        // Kafka broker addresses
    clientId: string;         // Client identifier
    topic: string;            // Topic for log messages
    compression?: CompressionTypes; // Message compression

    // Performance options
    batchSize?: number;       // Number of messages to batch
    flushIntervalMs?: number; // Flush interval

    // Filtering options
    minLogLevel?: LogLevel;   // Minimum log level to send

    // Security options
    ssl?: boolean;
    sasl?: {
        mechanism: 'plain' | 'scram-sha-256' | 'scram-sha-512';
        username: string;
        password: string;
    };
}
```

KafkaTransport

The `KafkaTransport` class handles:

1. **Connection management**: Establishing and maintaining connection to Kafka brokers
2. **Message buffering**: Collecting messages in a queue
3. **Batching**: Sending messages in batches for better performance
4. **Level filtering**: Only sending messages of specified levels
5. **Error handling**: Managing connection issues and retries
6. **Graceful shutdown**: Properly closing connections

4. How the Integration Works

Message Flow

1. Logger records a log entry with `logger.info()`, `logger.error()`, etc.
2. The log entry is formatted according to the standard format

3. If Kafka transport is configured, the entry is also sent to the `KafkaTransport`
4. `KafkaTransport` checks if the log level meets the minimum threshold
5. If it does, the message is added to the queue
6. Messages are sent to Kafka when:
    - The batch size is reached
    - The flush interval timer triggers
    - The application is shutting down

Configuration Example

```typescript
logger.addKafkaTransport({
  brokers: ['localhost:9092'],
  clientId: 'my-service-logger',
  topic: 'application-logs',
  batchSize: 100,
  flushIntervalMs: 3000,
  minLogLevel: 'INFO'
});
```

Performance Considerations

- **Batching**: Multiple log messages are sent in a single request to reduce network overhead
- **Compression**: Messages can be compressed to reduce bandwidth usage
- **Asynchronous operation**: Logging doesn't block the main application flow
- **Level filtering**: Reduces the volume of messages sent to Kafka

5. Technical Implementation

Connection Management

The transport connects to Kafka when the first log message is sent, or when explicitly initialized:

typescript

```typescript
async init(): Promise<void> {
    if (this.connected || this.connecting) return;

    this.connecting = true;
    try {
        await this.producer.connect();
        this.connected = true;
        this.startFlushTimer();
    } catch (error) {
        console.error('Failed to connect to Kafka:', error);
    } finally {
        this.connecting = false;
    }
}
```

Message Batching

Messages are queued and sent in batches:

typescript

```typescript
async log(entry: LogEntry): Promise<void> {
    if (!this.shouldLog(entry.level as LogLevel)) return;

    if (!this.connected) {
        await this.init();
    }

    const message: Message = {
        value: JSON.stringify(entry)
    };

    this.messageQueue.push(message);

    if (this.messageQueue.length >= this.batchSize) {
        await this.flush();
    }
}
```

Graceful Shutdown

The transport ensures all messages are sent before shutting down:

typescript

```typescript
async close(): Promise<void> {
    if (this.flushTimer) {
        clearInterval(this.flushTimer);
        this.flushTimer = null;
    }

    if (this.messageQueue.length > 0) {
        await this.flush();
    }

    if (this.connected) {
        await this.producer.disconnect();
        this.connected = false;
    }
}
```

6. Logger Integration

The main `Logger` class was enhanced to support Kafka:

typescript

```typescript
public addKafkaTransport(options: KafkaTransportOptions): void {
    // Create Kafka transport
    this.kafkaTransport = new KafkaTransport(options);

    // Initialize Kafka connection
    this.kafkaTransport.init().catch(err => {
      console.error('Failed to initialize Kafka transport:', err);
    });

    // Add shutdown handlers
    if (!process.listenerCount('SIGTERM')) {
      process.on('SIGTERM', async () => {
        await this.close();
        process.exit(0);
      });
    }
}

public async close(): Promise<void> {
    if (this.kafkaTransport) {
      await this.kafkaTransport.close();
    }
    return Promise.resolve();
}
```

Usage Scenarios

Microservices Architecture

In a microservices environment, Kafka integration allows:

1. **Centralized logging**: All services send logs to a common Kafka topic
2. **Log processing**: Consumer applications can process logs for monitoring, alerting, or analytics
3. **Correlation**: Distributed tracing with request IDs across services

High-Volume Applications

For applications with high log volume:

1. **Buffering**: Logs are buffered and sent in batches to handle spikes
2. **Filtering**: Only important logs (INFO and above) are sent to Kafka

3. **Compression**: Reduces network bandwidth usage

Production Monitoring

For production monitoring:

1. **Real-time alerting**: Critical errors can trigger alerts via Kafka consumers
2. **Dashboard integration**: Logs can be visualized in real-time dashboards
3. **Audit trail**: All application activities are recorded for compliance or debugging

Testing

The Kafka integration can be tested using Docker Compose:

```yaml
version: '3'
services:
  zookeeper:
    image: confluentinc/cp-zookeeper:latest
    environment:
      ZOOKEEPER_CLIENT_PORT: 2181
      ZOOKEEPER_TICK_TIME: 2000
    ports:
      - "2181:2181"

  kafka:
    image: confluentinc/cp-kafka:latest
    depends_on:
      - zookeeper
    ports:
      - "9092:9092"
    environment:
      KAFKA_BROKER_ID: 1
      KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
      KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://localhost:9092
      KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
      KAFKA_AUTO_CREATE_TOPICS_ENABLE: "true"
```

Example test script:

```javascript
const { createLogger } = require('@stepstone/standard-logger');

const logger = createLogger({
  service: {
    name: 'kafka-test',
    version: '1.0.0',
    environment: 'test'
  }
});

logger.addKafkaTransport({
  brokers: ['localhost:9092'],
  clientId: 'test-client',
  topic: 'application-logs',
  minLogLevel: 'INFO'
});

logger.info('Test message');
logger.error('Test error', new Error('Something went wrong'));

setTimeout(async () => {
  await logger.close();
}, 5000);
```

Future Improvements

Potential enhancements for the Kafka integration:

1. **Customizable serialization**: Allow custom serializers for log messages
2. **Schema registry integration**: Support for Avro/Protobuf schemas
3. **Multiple topics**: Send different log levels to different topics
4. **Backpressure handling**: Better handling of Kafka broker capacity issues
5. **Circuit breaker pattern**: Prevent overwhelming Kafka during issues
6. **Metrics**: Track success/failure rates and performance metrics

Dependencies

The integration relies on:

- **kafkajs**: A modern Kafka client for Node.js

- **pino**: The underlying logger used by Standard Logger JS

Conclusion

The Kafka integration enhances Standard Logger JS with distributed logging capabilities, making it suitable for modern cloud-native applications. It maintains the standardized log format while providing efficient, configurable transport to Kafka clusters.