

PeeGeeQ Architecture & API Reference

© Mark Andrew Ray-Smith Cityline Ltd 2025

Technical reference documentation for PeeGeeQ system architecture, API specifications, and integration patterns.

This document serves as a comprehensive technical reference for developers, architects, and system integrators who need detailed information about PeeGeeQ's internal architecture, API contracts, and integration capabilities.

New to PeeGeeQ? Start with the [PeeGeeQ Complete Guide](#) for step-by-step tutorials and progressive learning.

Document Scope

This reference covers:

- **System Architecture:** Internal design, module relationships, and data flow
- **API Specifications:** Complete interface definitions with method signatures
- **Database Schema:** Table structures, indexes, and relationships
- **Performance Characteristics:** Benchmarks, throughput, and latency metrics
- **Integration Patterns:** Technical integration examples for various platforms
- **Design Patterns:** Architectural patterns and implementation details

Table of Contents

1. [System Architecture](#)
2. [Module Structure](#)
3. [Core API Reference](#)
4. [Database Schema](#)
5. [Design Patterns](#)
6. [REST API Reference](#)
7. [Management Console Architecture](#)
8. [Performance Characteristics](#)
9. [Integration Patterns](#)

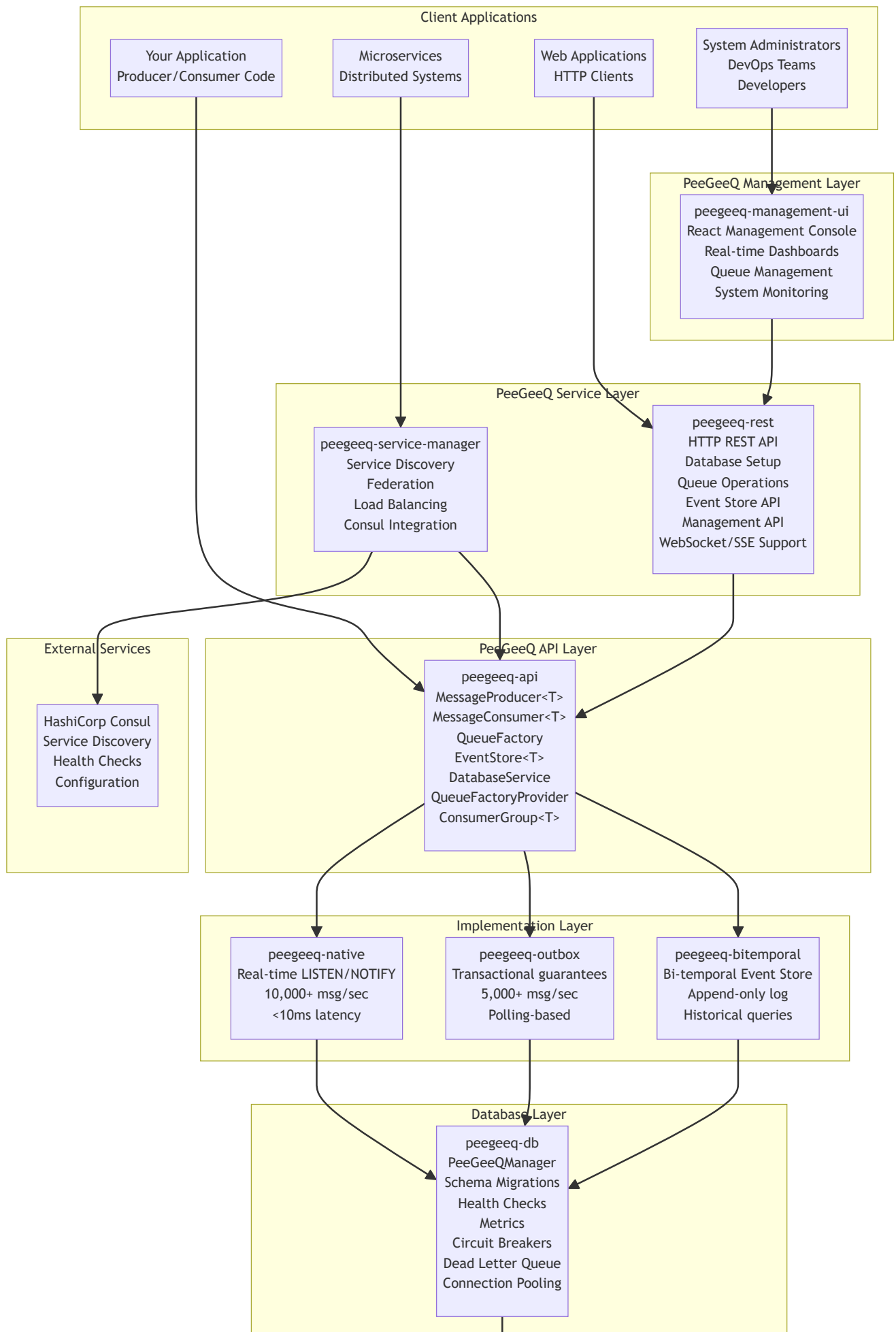
Related Documentation

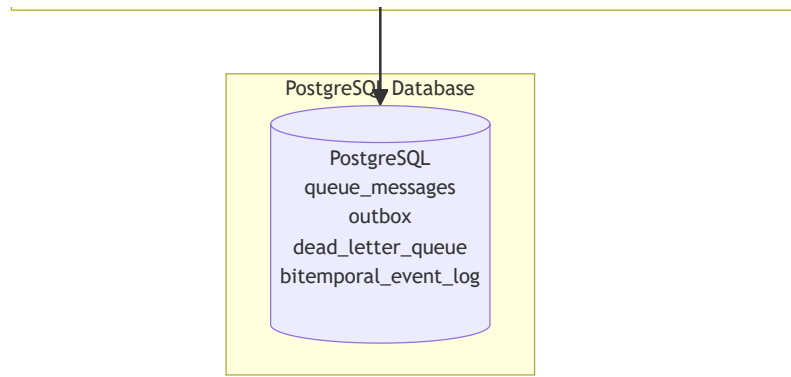
- [PeeGeeQ Complete Guide](#) - Progressive learning guide with tutorials and examples
- [Getting Started Tutorial](#) - Your first PeeGeeQ application
- [Configuration Guide](#) - Production configuration and tuning
- [Troubleshooting Guide](#) - Common issues and solutions

System Architecture

High-Level Architecture

PeeGeeQ is built as a layered architecture that leverages PostgreSQL's advanced features for enterprise-grade messaging:





Core Design Principles

1. **PostgreSQL-Native:** Leverages PostgreSQL's LISTEN/NOTIFY, advisory locks, and ACID transactions
2. **Type Safety:** Strongly typed APIs with generic support
3. **Pluggable Architecture:** Multiple queue implementations via factory pattern
4. **Production Ready:** Built-in health checks, metrics, circuit breakers, and monitoring
5. **Zero Dependencies:** No external message brokers required
6. **Transactional Consistency:** Full ACID compliance with business data

Module Structure

PeeGeeQ consists of 9 core modules organized in a layered architecture:

1. peegeeq-api (Core Interfaces)

Purpose: Defines core contracts and interfaces **Key Components:**

- `MessageProducer<T>` - Message publishing interface
- `MessageConsumer<T>` - Message consumption interface
- `Message<T>` - Message abstraction
- `EventStore<T>` - Bi-temporal event store interface
- `BiTemporalEvent<T>` - Bi-temporal event abstraction
- `DatabaseService` - Database operations interface
- `QueueFactoryProvider` - Factory provider interface

2. peegeeq-db (Database Management)

Purpose: Database infrastructure and management **Key Components:**

- `PeeGeeQManager` - Main entry point and lifecycle management
- `DatabaseService` - Database operations and connection management
- `SchemaMigrationManager` - Versioned schema migrations
- `HealthCheckManager` - Multi-component health monitoring
- `PeeGeeQMetrics` - Metrics collection and reporting
- `CircuitBreakerManager` - Resilience patterns
- `DeadLetterQueueManager` - Failed message handling

3. peegeeq-native (High-Performance Implementation)

Purpose: Real-time LISTEN/NOTIFY based messaging **Key Components:**

- PgNativeQueueFactory - Factory for native queues
- PgNativeProducer<T> - High-performance message producer
- PgNativeConsumer<T> - Real-time message consumer
- PgConnectionProvider - Optimized connection management

Performance: 10,000+ msg/sec, <10ms latency

4. peegeequeue-outbox (Transactional Implementation)

Purpose: Transactional outbox pattern implementation **Key Components:**

- OutboxQueueFactory - Factory for outbox queues
- OutboxProducer<T> - Transactional message producer
- OutboxConsumer<T> - Polling-based message consumer
- OutboxPollingService - Background polling service

Performance: 5,000+ msg/sec, ACID compliance

5. peegeequeue-bitemporal (Event Store)

Purpose: Bi-temporal event sourcing capabilities **Key Components:**

- BiTemporalEventStore<T> - Main event store interface
- PgBiTemporalEventStore<T> - PostgreSQL implementation
- BiTemporalEvent<T> - Event with temporal metadata
- EventQuery - Query builder for temporal queries

6. peegeequeue-rest (HTTP API)

Purpose: HTTP REST API server **Key Components:**

- PeeGeeQRestServer - Vert.x based HTTP server
- DatabaseSetupService - Database setup via REST
- QueueOperationsHandler - Queue operations via HTTP
- EventStoreHandler - Event store operations via HTTP

7. peegeequeue-service-manager (Service Discovery)

Purpose: Service discovery and federation **Key Components:**

- PeeGeeQServiceManager - Main service manager
- ConsulServiceDiscovery - Consul integration
- FederationHandler - Multi-instance coordination
- LoadBalancingStrategy - Request routing

8. peegeequeue-management-ui (Management Console)

Purpose: Web-based administration interface for PeeGeeQ system management **Key Components:**

- React Management Console - Modern web interface inspired by RabbitMQ's admin console


- System Overview Dashboard - Real-time metrics and system health monitoring
- Queue Management Interface - Complete CRUD operations for queues
- Consumer Group Management - Visual consumer group coordination
- Event Store Explorer - Advanced event querying interface
- Message Browser - Visual message inspection and debugging
- Real-time Monitoring - Live dashboards with WebSocket updates
- Developer Portal - Interactive API documentation and testing

Technology Stack: React 18 + TypeScript + Ant Design + Vite **Integration:** Served by PeeGeeQ REST server with management API endpoints

9. peegeeq-examples (Demonstrations)

Purpose: Comprehensive example applications and demonstrations covering all PeeGeeQ features

Core Examples:

- PeeGeeQSelfContainedDemo - Complete self-contained demonstration
-  PeeGeeQExample - Basic producer/consumer patterns
- BiTemporalEventStoreExample - Event sourcing with temporal queries
- ConsumerGroupExample - Load balancing and consumer groups
- RestApiExample - HTTP interface usage
- ServiceDiscoveryExample - Multi-instance deployment

Advanced Examples (Enhanced):

- MessagePriorityExample - Priority-based message processing with real-world scenarios
- EnhancedErrorHandlingExample - Retry strategies, circuit breakers, poison message handling
- SecurityConfigurationExample - SSL/TLS, certificate management, compliance features
- PerformanceTuningExample - Connection pooling, throughput optimization, memory tuning
- IntegrationPatternsExample - Request-reply, pub-sub, message routing, distributed patterns

Specialized Examples:

- TransactionalBiTemporalExample - Combining transactions with event sourcing
- RestApiStreamingExample - WebSocket and Server-Sent Events
- NativeVsOutboxComparisonExample - Performance comparison and use case guidance
- AdvancedConfigurationExample - Production configuration patterns
- MultiConfigurationExample - Multi-environment setup
- SimpleConsumerGroupTest - Basic consumer group testing

Coverage: 95-98% of PeeGeeQ functionality with production-ready patterns

Core API Reference

Message Interfaces

MessageProducer

```
public interface MessageProducer<T> extends AutoCloseable {
    /**
```

```

    * Send a message with the given payload
    */
    CompletableFuture<Void> send(T payload);

    /**
     * Send a message with the given payload and headers
     */
    CompletableFuture<Void> send(T payload, Map<String, String> headers);

    /**
     * Send a message with the given payload, headers, and correlation ID
     */
    CompletableFuture<Void> send(T payload, Map<String, String> headers, String correlationId);

    /**
     * Send a message with the given payload, headers, correlation ID, and message group
     */
    CompletableFuture<Void> send(T payload, Map<String, String> headers, String correlationId, String messageGroup);

    /**
     * Close the producer and release resources
     */
    @Override
    void close();
}

```

MessageConsumer

```

public interface MessageConsumer<T> extends AutoCloseable {
    /**
     * Subscribe to messages with the given handler
     */
    void subscribe(MessageHandler<T> handler);

    /**
     * Unsubscribe from message processing
     */
    void unsubscribe();

    /**
     * Close the consumer and release resources
     */
    @Override
    void close();
}

```

Message

```

public interface Message<T> {
    /**
     * Unique message identifier
     */
    String getId();

    /**
     * Message payload
     */
    T getPayload();

    /**

```



```

    * Message headers
    */
    Map<String, String> getHeaders();

    /**
     * Message priority (0-9, higher = more priority)
     */
    int getPriority();

    /**
     * Message creation timestamp
     */
    Instant getCreatedAt();

    /**
     * Correlation ID for message tracking
     */
    String getCorrelationId();
}

```

Queue Factory Pattern

QueueFactoryProvider

```

public interface QueueFactoryProvider {
    /**
     * Get the singleton instance
     */
    static QueueFactoryProvider getInstance();

    /**
     * Create a queue factory of the specified type with configuration
     */
    QueueFactory createFactory(String implementationType,
                               DatabaseService databaseService,
                               Map<String, Object> configuration);

    /**
     * Create a queue factory of the specified type with default configuration
     */
    QueueFactory createFactory(String implementationType, DatabaseService databaseService);

    /**
     * Get the set of supported implementation types
     */
    Set<String> getSupportedTypes();

    /**
     * Create a queue factory using a named configuration template
     */
    default QueueFactory createNamedFactory(String implementationType,
                                             String configurationName,
                                             DatabaseService databaseService,
                                             Map<String, Object> additionalConfig);
}

```

QueueFactory

```

public interface QueueFactory extends AutoCloseable {
    /**

```

```

    * Create a message producer for the specified topic
    */
    <T> MessageProducer<T> createProducer(String topic, Class<T> payloadType);

    /**
     * Create a message consumer for the specified topic
     */
    <T> MessageConsumer<T> createConsumer(String topic, Class<T> payloadType);

    /**
     * Create a consumer group for the specified topic
     */
    <T> ConsumerGroup<T> createConsumerGroup(String groupName, String topic, Class<T> payloadType);

    /**
     * Get the implementation type of this factory
     */
    String getImplementationType();

    /**
     * Check if the factory is healthy and ready to create queues
     */
    boolean isHealthy();

    /**
     * Close factory and release resources
     */
    @Override
    void close() throws Exception;
}

```

Database Service

DatabaseService

```

public interface DatabaseService {
    /**
     * Get a database connection
     */
    Connection getConnection() throws SQLException;

    /**
     * Execute a query with parameters
     */
    <T> List<T> query(String sql, RowMapper<T> mapper, Object... params);

    /**
     * Execute an update statement
     */
    int update(String sql, Object... params);

    /**
     * Execute within a transaction
     */
    <T> T executeInTransaction(TransactionCallback<T> callback);

    /**
     * Get connection pool statistics
     */
    ConnectionPoolStats getPoolStats();

    /**
     * Check if the database is healthy

```

```
    */
    boolean isHealthy();
}
```

Event Store API

EventStore

```
public interface EventStore<T> {
    /**
     * Append an event to the store
     */
    Future<BiTemporalEvent<T>> appendEvent(String aggregateId, T event);

    /**
     * Append an event with metadata
     */
    Future<BiTemporalEvent<T>> appendEvent(String aggregateId, T event,
                                           Map<String, String> metadata);

    /**
     * Query events by aggregate ID
     */
    Future<List<BiTemporalEvent<T>>> queryByAggregateId(String aggregateId);

    /**
     * Query events by time range
     */
    Future<List<BiTemporalEvent<T>>> queryByTimeRange(Instant from, Instant to);

    /**
     * Query events as of a specific transaction time
     */
    Future<List<BiTemporalEvent<T>>> queryAsOfTransactionTime(Instant asOf);

    /**
     * Correct an existing event
     */
    Future<BiTemporalEvent<T>> correctEvent(String eventId, T correctedEvent,
                                           String reason);
}
```

BiTemporalEvent

```
public interface BiTemporalEvent<T> {
    /**
     * Unique event identifier
     */
    String getEventId();

    /**
     * Aggregate identifier
     */
    String getAggregateId();

    /**
     * Event payload
     */
    T getPayload();
}
```

```

/**
 * Event type
 */
String getEventType();

/**
 * Valid time (business time)
 */
Instant getValidFrom();
Instant getValidTo();

/**
 * Transaction time (system time)
 */
Instant getTransactionTime();

/**
 * Event version (for corrections)
 */
int getVersion();

/**
 * Correlation ID
 */
String getCorrelationId();

/**
 * Event metadata
 */
Map<String, String> getMetadata();
}

```

Configuration Classes

PeeGeeQConfiguration

```

public class PeeGeeQConfiguration {
    // Database settings
    private String host = "localhost";
    private int port = 5432;
    private String database;
    private String username;
    private String password;

    // Connection pool settings
    private int maxPoolSize = 20;
    private int minPoolSize = 5;
    private Duration connectionTimeout = Duration.ofSeconds(30);

    // Queue settings
    private Duration visibilityTimeout = Duration.ofSeconds(30);
    private int maxRetries = 3;
    private boolean deadLetterEnabled = true;

    // Health check settings
    private boolean healthEnabled = true;
    private Duration healthInterval = Duration.ofSeconds(30);

    // Metrics settings
    private boolean metricsEnabled = true;
    private boolean jvmMetricsEnabled = true;

    // Builder pattern and factory methods
}

```

```

    public static Builder builder() { return new Builder(); }
    public static PeeGeeQConfiguration fromProperties(String filename);
    public static PeeGeeQConfiguration fromProperties(Properties properties);
}

```

ConsumerConfig

```

public class ConsumerConfig {
    private int batchSize = 10;
    private Duration pollInterval = Duration.ofSeconds(1);
    private Duration visibilityTimeout = Duration.ofSeconds(30);
    private int maxRetries = 3;
    private boolean autoAcknowledge = true;
    private MessageFilter filter;
    private String consumerGroup;

    // Builder pattern
    public static Builder builder() { return new Builder(); }
}

```

Database Schema

Core Tables

queue_messages

```

CREATE TABLE queue_messages (
    id BIGSERIAL PRIMARY KEY,
    topic VARCHAR(255) NOT NULL,
    payload JSONB NOT NULL,
    visible_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
    created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
    lock_id BIGINT,
    lock_until TIMESTAMP WITH TIME ZONE,
    retry_count INT DEFAULT 0,
    max_retries INT DEFAULT 3,
    status VARCHAR(50) DEFAULT 'AVAILABLE' CHECK (status IN ('AVAILABLE', 'LOCKED', 'PROCESSED', 'FAILED', 'DEAD_LETTER')),
    headers JSONB DEFAULT '{}',
    error_message TEXT,
    correlation_id VARCHAR(255),
    message_group VARCHAR(255),
    priority INT DEFAULT 5 CHECK (priority BETWEEN 1 AND 10)
);

-- Indexes
CREATE INDEX idx_queue_messages_topic_visible ON queue_messages(topic, visible_at, status);
CREATE INDEX idx_queue_messages_lock ON queue_messages(lock_id) WHERE lock_id IS NOT NULL;
CREATE INDEX idx_queue_messages_status ON queue_messages(status, created_at);
CREATE INDEX idx_queue_messages_correlation_id ON queue_messages(correlation_id) WHERE correlation_id IS NOT NULL;
CREATE INDEX idx_queue_messages_priority ON queue_messages(priority, created_at);

```

outbox

```

CREATE TABLE outbox (
    id BIGSERIAL PRIMARY KEY,

```

```

topic VARCHAR(255) NOT NULL,
payload JSONB NOT NULL,
created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
processed_at TIMESTAMP WITH TIME ZONE,
processing_started_at TIMESTAMP WITH TIME ZONE,
status VARCHAR(50) DEFAULT 'PENDING' CHECK (status IN ('PENDING', 'PROCESSING', 'COMPLETED', 'FAILED', 'DEAD_LETTER'))
retry_count INT DEFAULT 0,
max_retries INT DEFAULT 3,
next_retry_at TIMESTAMP WITH TIME ZONE,
version INT DEFAULT 0,
headers JSONB DEFAULT '{}',
error_message TEXT,
correlation_id VARCHAR(255),
message_group VARCHAR(255),
priority INT DEFAULT 5 CHECK (priority BETWEEN 1 AND 10)
);

-- Indexes
CREATE INDEX idx_outbox_status_created ON outbox(status, created_at);
CREATE INDEX idx_outbox_next_retry ON outbox(status, next_retry_at) WHERE status = 'FAILED';
CREATE INDEX idx_outbox_topic ON outbox(topic);
CREATE INDEX idx_outbox_correlation_id ON outbox(correlation_id) WHERE correlation_id IS NOT NULL;
CREATE INDEX idx_outbox_message_group ON outbox(message_group) WHERE message_group IS NOT NULL;
CREATE INDEX idx_outbox_priority ON outbox(priority, created_at);

```

bitemporal_event_log

```

CREATE TABLE bitemporal_event_log (
  -- Primary key and identity
  id BIGSERIAL PRIMARY KEY,
  event_id VARCHAR(255) NOT NULL,
  event_type VARCHAR(255) NOT NULL,

  -- Bi-temporal dimensions
  valid_time TIMESTAMP WITH TIME ZONE NOT NULL,
  transaction_time TIMESTAMP WITH TIME ZONE DEFAULT NOW() NOT NULL,

  -- Event data
  payload JSONB NOT NULL,
  headers JSONB DEFAULT '{}',

  -- Versioning and corrections
  version BIGINT DEFAULT 1 NOT NULL,
  previous_version_id VARCHAR(255),
  is_correction BOOLEAN DEFAULT FALSE NOT NULL,
  correction_reason TEXT,

  -- Grouping and correlation
  correlation_id VARCHAR(255),
  aggregate_id VARCHAR(255),

  -- Metadata
  created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW() NOT NULL
);

-- Comprehensive indexing strategy
CREATE INDEX idx_bitemporal_valid_time ON bitemporal_event_log(valid_time);
CREATE INDEX idx_bitemporal_transaction_time ON bitemporal_event_log(transaction_time);
CREATE INDEX idx_bitemporal_valid_transaction ON bitemporal_event_log(valid_time, transaction_time);
CREATE INDEX idx_bitemporal_event_id ON bitemporal_event_log(event_id);
CREATE INDEX idx_bitemporal_event_type ON bitemporal_event_log(event_type);
CREATE INDEX idx_bitemporal_aggregate_id ON bitemporal_event_log(aggregate_id) WHERE aggregate_id IS NOT NULL;
CREATE INDEX idx_bitemporal_correlation_id ON bitemporal_event_log(correlation_id) WHERE correlation_id IS NOT NULL;

```

```

CREATE INDEX idx_bitemporal_version_chain ON bitemporal_event_log(event_id, version);
CREATE INDEX idx_bitemporal_corrections ON bitemporal_event_log(is_correction, transaction_time) WHERE is_correction = TR
CREATE INDEX idx_bitemporal_latest_events ON bitemporal_event_log(event_type, transaction_time DESC) WHERE is_correction

-- GIN indexes for JSONB queries
CREATE INDEX idx_bitemporal_payload_gin ON bitemporal_event_log USING GIN(payload);
CREATE INDEX idx_bitemporal_headers_gin ON bitemporal_event_log USING GIN(headers);

```

dead_letter_queue

```

CREATE TABLE dead_letter_queue (
    id BIGSERIAL PRIMARY KEY,
    original_table VARCHAR(50) NOT NULL,
    original_id BIGINT NOT NULL,
    topic VARCHAR(255) NOT NULL,
    payload JSONB NOT NULL,
    original_created_at TIMESTAMP WITH TIME ZONE NOT NULL,
    failed_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
    failure_reason TEXT NOT NULL,
    retry_count INT NOT NULL,
    headers JSONB DEFAULT '{}',
    correlation_id VARCHAR(255),
    message_group VARCHAR(255)
);

-- Indexes
CREATE INDEX idx_dlq_original ON dead_letter_queue(original_table, original_id);
CREATE INDEX idx_dlq_topic ON dead_letter_queue(topic);
CREATE INDEX idx_dlq_failed_at ON dead_letter_queue(failed_at);

```

Additional Tables

outbox_consumer_groups

```

CREATE TABLE outbox_consumer_groups (
    id BIGSERIAL PRIMARY KEY,
    outbox_message_id BIGINT NOT NULL REFERENCES outbox(id) ON DELETE CASCADE,
    consumer_group_name VARCHAR(255) NOT NULL,
    status VARCHAR(50) DEFAULT 'PENDING' CHECK (status IN ('PENDING', 'PROCESSING', 'COMPLETED', 'FAILED')),
    processed_at TIMESTAMP WITH TIME ZONE,
    processing_started_at TIMESTAMP WITH TIME ZONE,
    retry_count INT DEFAULT 0,
    error_message TEXT,
    created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),

    UNIQUE(outbox_message_id, consumer_group_name)
);

```

queue_metrics & connection_pool_metrics

```

CREATE TABLE queue_metrics (
    id BIGSERIAL PRIMARY KEY,
    metric_name VARCHAR(100) NOT NULL,
    metric_value DOUBLE PRECISION NOT NULL,
    tags JSONB DEFAULT '{}',
    timestamp TIMESTAMP WITH TIME ZONE DEFAULT NOW()
);

```

```
CREATE TABLE connection_pool_metrics (
    id BIGSERIAL PRIMARY KEY,
    pool_name VARCHAR(100) NOT NULL,
    active_connections INT NOT NULL,
    idle_connections INT NOT NULL,
    total_connections INT NOT NULL,
    pending_threads INT NOT NULL,
    timestamp TIMESTAMP WITH TIME ZONE DEFAULT NOW()
);
```

Design Patterns

Factory Pattern

PeeGeeQ uses the Factory pattern to provide pluggable queue implementations:

```
// Factory Provider (Singleton)
QueueFactoryProvider provider = QueueFactoryProvider.getInstance();

// Create specific factory implementations
QueueFactory nativeFactory = provider.createFactory("native", databaseService);
QueueFactory outboxFactory = provider.createFactory("outbox", databaseService);

// Factories create producers and consumers
MessageProducer<String> producer = nativeFactory.createProducer("orders", String.class);
MessageConsumer<String> consumer = nativeFactory.createConsumer("orders", String.class);
```

Observer Pattern

Message consumption uses the Observer pattern with async callbacks:

```
consumer.subscribe(message -> {
    // Process message
    processOrder(message.getPayload());

    // Return completion future
    return CompletableFuture.completedFuture(null);
});
```

Template Method Pattern

Database operations use template methods for consistent transaction handling:

```
public <T> T executeInTransaction(TransactionCallback<T> callback) {
    Connection conn = getConnection();
    try {
        conn.setAutoCommit(false);
        T result = callback.execute(conn);
        conn.commit();
        return result;
    } catch (Exception e) {
        conn.rollback();
        throw new RuntimeException(e);
    } finally {
```



```
        conn.close();
    }
}
```

Circuit Breaker Pattern

Built-in resilience with circuit breakers:

```
@CircuitBreaker(name = "database-operations", fallbackMethod = "fallbackMethod")
public void performDatabaseOperation() {
    // Database operation that might fail
}

public void fallbackMethod(Exception ex) {
    // Fallback logic when circuit is open
}
```

REST API Reference

Database Setup Endpoints

Create Database Setup

POST /api/v1/database-setup/create
Content-Type: application/json

```
{
  "setupId": "my-setup",
  "databaseConfig": {
    "host": "localhost",
    "port": 5432,
    "databaseName": "my_app_db",
    "username": "postgres",
    "password": "password",
    "schema": "public"
  },
  "queues": [
    {
      "queueName": "orders",
      "maxRetries": 3,
      "visibilityTimeoutSeconds": 30
    }
  ],
  "eventStores": [
    {
      "eventStoreName": "order-events",
      "tableName": "order_events",
      "biTemporalEnabled": true
    }
  ]
}
```

Other Database Setup Endpoints

- DELETE /api/v1/database-setup/{setupId} - Destroy a database setup

- GET /api/v1/database-setup/{setupId}/status - Get setup status
- POST /api/v1/database-setup/{setupId}/queues - Add queue to setup
- POST /api/v1/database-setup/{setupId}/eventstores - Add event store to setup

Queue Operations Endpoints

Send Message to Queue

POST /api/v1/queues/{setupId}/{queueName}/messages
Content-Type: application/json

```
{
  "payload": {
    "orderId": "12345",
    "customerId": "67890",
    "amount": 99.99
  },
  "headers": {
    "source": "order-service",
    "version": "1.0"
  },
  "priority": 5,
  "correlationId": "order-12345"
}
```

Other Queue Endpoints

- POST /api/v1/queues/{setupId}/{queueName}/messages/batch - Send multiple messages
- GET /api/v1/queues/{setupId}/{queueName}/stats - Get queue statistics
- GET /api/v1/queues/{setupId}/{queueName}/messages/next - Get next message
- GET /api/v1/queues/{setupId}/{queueName}/messages - Get messages with filtering
- DELETE /api/v1/queues/{setupId}/{queueName}/messages/{messageId} - Acknowledge message

Event Store Endpoints

Store Event

POST /api/v1/eventstores/{setupId}/{eventStoreName}/events
Content-Type: application/json

```
{
  "aggregateId": "order-12345",
  "eventType": "OrderCreated",
  "payload": {
    "orderId": "12345",
    "customerId": "67890",
    "amount": 99.99
  },
  "validTime": "2025-08-23T10:00:00Z",
  "correlationId": "order-12345",
  "headers": {
    "source": "order-service"
  }
}
```

Query Events

- GET /api/v1/eventstores/{setupId}/{eventStoreName}/events - Query events with filters
- GET /api/v1/eventstores/{setupId}/{eventStoreName}/events/{aggregateId} - Get events by aggregate
- GET /api/v1/eventstores/{setupId}/{eventStoreName}/stats - Get event store statistics

Management API Endpoints

System Health and Overview

- GET /api/v1/health - Health check endpoint
- GET /api/v1/management/overview - System overview dashboard data
- GET /api/v1/management/queues - Queue management data
- GET /api/v1/management/metrics - System metrics
- GET /api/v1/management/consumer-groups - Consumer group information
- GET /api/v1/management/event-stores - Event store management data

Real-time Communication

WebSocket Endpoints

- WS /ws/queues/{setupId}/{queueName} - Real-time queue message streaming
- WS /ws/monitoring - System monitoring updates

Server-Sent Events (SSE)


- GET /sse/metrics - Real-time system metrics stream
- GET /sse/queues/{setupId} - Real-time queue updates stream
- GET /api/v1/queues/{setupId}/{queueName}/stream - Queue message stream

Consumer Group Endpoints

Consumer Group Management

- POST /api/v1/consumer-groups/{setupId} - Create consumer group
- GET /api/v1/consumer-groups/{setupId} - List consumer groups
- GET /api/v1/consumer-groups/{setupId}/{groupName} - Get consumer group details
- DELETE /api/v1/consumer-groups/{setupId}/{groupName} - Delete consumer group
- POST /api/v1/consumer-groups/{setupId}/{groupName}/consumers - Add consumer to group

Management Console Architecture

 For detailed Management Console usage and features, see the [Management Console section](#) in the Complete Guide.

Technical Architecture

The Management Console is built as a React 18 single-page application that integrates with PeeGeeQ's REST API and real-time communication endpoints.

Component Architecture

```
peegee-management-ui/
├── src/
│   ├── components/      # Reusable UI components
│   ├── pages/           # Main application pages
│   ├── services/        # API integration services
│   ├── hooks/           # Custom React hooks
│   ├── utils/           # Utility functions
│   └── types/           # TypeScript type definitions
├── public/              # Static assets
└── dist/                # Built application (served by REST server)
```

Integration Points

- **REST API:** `/api/v1/management/*` endpoints for data operations
- **WebSocket:** `/ws/monitoring` for real-time system updates
- **Server-Sent Events:** `/sse/metrics` for live metrics streaming
- **Static Serving:** Built application served at `/ui/` by PeeGeeQ REST server

Technology Stack

- **Frontend:** React 18 + TypeScript + Vite
- **UI Framework:** Ant Design 5.x
- **State Management:** Zustand
- **Charts:** Recharts for data visualization
- **Build Tool:** Vite for development and production builds

Performance Characteristics

Native Queue Performance

- **Throughput:** 10,000+ messages/second
- **Latency:** <10ms end-to-end
- **Mechanism:** PostgreSQL LISTEN/NOTIFY with advisory locks
- **Concurrency:** Multiple consumers with automatic load balancing
- **Scalability:** Horizontal scaling via consumer groups
- **Memory Usage:** Low memory footprint with streaming processing
- **Connection Efficiency:** Connection pooling with optimized pool sizes

Outbox Pattern Performance

- **Throughput:** 5,000+ messages/second
- **Latency:** ~100ms (polling-based with configurable intervals)
- **Mechanism:** Database polling with ACID transactions
- **Consistency:** Full ACID compliance with business data
- **Reliability:** Exactly-once delivery guarantee
- **Durability:** Transactional outbox ensures no message loss
- **Retry Handling:** Configurable retry policies with exponential backoff

Bi-temporal Event Store Performance

- **Write Throughput:** 3,000+ events/second

- **Query Performance:** <50ms for typical temporal queries
- **Storage:** Append-only, optimized for time-series data
- **Indexing:** Multi-dimensional indexes for temporal and aggregate queries
- **Correction Support:** Efficient event correction with version tracking
- **Historical Queries:** Point-in-time queries with transaction time support
- **Aggregate Reconstruction:** Fast aggregate state reconstruction

REST API Performance


- **HTTP Throughput:** 2,000+ requests/second
- **WebSocket Throughput:** 5,000+ messages/second per connection
- **SSE Throughput:** 3,000+ events/second per connection
- **Latency:** <50ms for REST operations, <20ms for WebSocket
- **Concurrent Connections:** 1,000+ simultaneous WebSocket connections
- **Management Operations:** Sub-second response times for admin operations

Management Console Performance

- **UI Responsiveness:** <100ms for dashboard updates
- **Real-time Updates:** <500ms latency for live metrics
- **Data Visualization:** Handles 10,000+ data points in charts
- **Concurrent Users:** 50+ simultaneous admin users
- **Resource Usage:** <50MB memory footprint in browser

Integration Patterns

Integration Architecture Patterns

 For complete integration examples and tutorials, see the [Integration Patterns section](#) in the Complete Guide.

Microservices Integration Pattern

- **Producer Services:** Publish domain events after business operations
- **Consumer Services:** Subscribe to relevant events for cross-service coordination
- **Event-Driven Architecture:** Loose coupling through asynchronous messaging
- **Transactional Consistency:** Outbox pattern ensures message delivery with business data

Spring Boot Integration Pattern

- **Auto-Configuration:** Automatic bean creation and dependency injection
- **Configuration Properties:** External configuration through `application.properties`
- **Lifecycle Management:** Automatic startup/shutdown with Spring context
- **Health Checks:** Integration with Spring Boot Actuator

REST API Integration Pattern

- **HTTP Endpoints:** RESTful API for message operations
- **Async Processing:** Non-blocking message sending with `CompletableFuture`
- **Error Handling:** Structured error responses and exception mapping
- **Content Negotiation:** JSON request/response format

Real-time Communication Protocols

WebSocket Protocol Specification

- **Endpoint Pattern:** `ws://host:port/ws/queues/{setupId}/{queueName}`
- **Message Format:** JSON-based protocol with type-based message routing
- **Connection Lifecycle:** Connect → Configure → Subscribe → Stream → Disconnect
- **Message Types:** `configure` , `subscribe` , `message` , `batch` , `ack` , `error`
- **Batch Support:** Configurable batch size and wait time parameters
- **Error Handling:** Structured error messages with error codes and descriptions

Server-Sent Events (SSE) Specification

- **Endpoint Pattern:** `/sse/{stream-type}` (metrics, queues, monitoring)
- **Event Types:** `message` , `queue-update` , `consumer-group-update` , `system-alert`
- **Data Format:** JSON payload in event data field
- **Connection Management:** Automatic reconnection with exponential backoff
- **Event Filtering:** Query parameters for event type and topic filtering

Consumer Group Architecture

Load Balancing Mechanism

- **Round-Robin Distribution:** Messages distributed evenly across active consumers
- **Automatic Failover:** Failed consumers removed from rotation automatically
- **Dynamic Scaling:** Add/remove consumers without service interruption
- **Message Affinity:** Route messages based on headers or content patterns

Consumer Group Lifecycle

1. **Group Creation:** `QueueFactory.createConsumerGroup(groupName, topic, payloadType)`
2. **Consumer Registration:** `ConsumerGroup.addConsumer(handler, filter)`
3. **Group Activation:** `ConsumerGroup.start()` begins message distribution
4. **Load Balancing:** Messages distributed across registered consumers
5. **Health Monitoring:** Automatic detection and handling of consumer failures
6. **Graceful Shutdown:** `ConsumerGroup.stop()` completes in-flight messages

Message Filtering Architecture

- **Consumer-Level Filters:** Each consumer can specify message criteria
- **Group-Level Filters:** Apply filters to entire consumer group
- **Header-Based Routing:** Route messages based on header values
- **Content-Based Filtering:** Filter messages based on payload content
- **Filter Composition:** Combine multiple filters with AND/OR logic

Event Store Architecture

Bi-temporal Data Model

- **Valid Time:** Business time when event was valid in real world
- **Transaction Time:** System time when event was recorded in database
- **Event Versioning:** Support for event corrections with version tracking
- **Aggregate Grouping:** Events grouped by aggregate ID for entity reconstruction

Query Patterns

- **Aggregate Queries:** `queryByAggregateId(aggregateId)` - All events for entity
- **Temporal Queries:** `queryByTimeRange(from, to)` - Events in time window
- **Point-in-Time Queries:** `queryAsOfTransactionTime(asOf)` - System state at specific time
- **Correction Queries:** `queryCorrections(eventId)` - Event correction history