⚡ Real-time Event Handling

> **Master advanced webhook patterns and event processing**

Build robust, scalable systems that respond intelligently to every avatar interaction.

🎯 Event Types Deep Dive

🔌 room_join

When: User connects to an avatar session
Frequency: Once per user connection
Use for: Session tracking, user onboarding, capacity monitoring


```
{
  "agent_id": "agent_customer_support",
  "event_type": "room.join",
  "data": {
    "room_name": "customer-support-room",
    "participant_count": 1,
    "session_id": "session_xyz789"
  },
  "timestamp": 1705312200.0
}
```


💬 chat_push

When: Any message sent in the conversation
Frequency: Per message (both user and agent)
Use for: Chat logging, sentiment analysis, keyword triggers


```
{
  "agent_id": "agent_customer_support",
  "event_type": "chat.push",
  "data": {
    "role": "user",
    "message": "I need help with my order #12345",
    "session_id": "session_xyz789",
    "timestamp": 1705312285.0
  },
  "timestamp": 1705312285.0
}
```

🏗️ Advanced Processing Patterns

🔄 Async Processing

Handle webhooks efficiently without blocking responses:

```
from celery import Celery
from flask import Flask, request, jsonify

app = Flask(__name__)
celery = Celery('webhook_processor')

@app.route('/webhook', methods=['POST'])
def webhook_handler():
    # Return 200 immediately
    data = request.json

    # Queue for background processing
    process_webhook_async.delay(data)

    return jsonify({'status': 'accepted'}), 200

@celery.task
def process_webhook_async(data):
    """Process webhook in background"""
    event_type = data.get('event_type')

    try:
        if event_type == 'chat.push':
            analyze_sentiment(data)
            update_conversation_log(data)
            check_keyword_triggers(data)
        elif event_type == 'room.join':
            log_user_session(data)
            update_capacity_metrics(data)
            trigger_welcome_message(data)

    except Exception as e:
        logger.error(f"Webhook processing failed: {e}")
        # Send to dead letter queue for retry
        handle_processing_error(data, e)
```

🎛️ Event Routing

Route different events to specialized handlers:

```
class WebhookRouter:
    def __init__(self):
        self.handlers = {
            'room.join': [
                self.log_user_session,
                self.update_capacity_metrics,
                self.trigger_welcome_message
            ],
            'chat.push': [
                self.analyze_message_sentiment,
                self.detect_urgent_keywords,
                self.update_conversation_state
            ]
```

```
        }

    def route_event(self, webhook_data):
        event_type = webhook_data.get('event_type')
        handlers = self.handlers.get(event_type, [])

        for handler in handlers:
            try:
                handler(webhook_data)
            except Exception as e:
                logger.error(f"Handler {handler.__name__} failed: {e}")
```

📊 Event Aggregation

Combine multiple events for insights:

```
class EventAggregator:
    def __init__(self):
        self.session_buffer = {}  # sessionId -> events

    def process_event(self, event_data):
        session_id = event_data.get('sessionId')
        event_type = event_data.get('event')

        if session_id not in self.session_buffer:
            self.session_buffer[session_id] = {
                'events': [],
                'start_time': None,
                'user_id': None
            }

        session = self.session_buffer[session_id]
        session['events'].append(event_data)

        # Update session metadata
        if event_type == 'room.join':
            session['start_time'] = event_data['timestamp']
            session['session_id'] = event_data['data']['session_id']

        elif event_type == 'chat.push':
            # Update latest activity timestamp
            session['last_activity'] = event_data['timestamp']

    def cleanup_inactive_sessions(self):
        """Clean up sessions that have been inactive for too long"""
        from datetime import datetime
        current_time = datetime.now()
        inactive_sessions = []

        for session_id, session_data in self.session_buffer.items():
            last_activity = session_data.get('last_activity', session_data.get('
            if last_activity:
                inactive_duration = (current_time - datetime.fromisoformat(last_a
                if inactive_duration > 1800:  # 30 minutes of inactivity
                    inactive_sessions.append(session_id)

        # Process and remove inactive sessions
        for session_id in inactive_sessions:
```

```
            session_data = self.session_buffer[session_id]
            self.analyze_session_activity(session_data)
            del self.session_buffer[session_id]

    def analyze_session_activity(self, session_data):
        """Analyze session activity based on available events"""
        events = session_data['events']

        # Calculate engagement metrics from available events
        message_events = [e for e in events if e['event_type'] == 'chat.push']
        user_messages = [e for e in message_events if e['data']['role'] == 'user

        engagement_score = self.calculate_engagement(user_messages)
        session_quality = self.assess_session_quality(events)

        # Store analytics based on current activity
        self.store_session_analytics({
            'session_id': session_data['session_id'],
            'message_count': len(message_events),
            'user_message_count': len(user_messages),
            'engagement_score': engagement_score,
            'quality_score': session_quality,
            'last_activity': session_data.get('last_activity')
        })
```

🎨 Integration Patterns

📱 Real-time Dashboard

Stream events to live dashboard:

```
from flask_socketio import SocketIO, emit

socketio = SocketIO(app, cors_allowed_origins="*")

@app.route('/webhook', methods=['POST'])
def webhook_handler():
    data = request.json

    # Process event
    processed_data = process_event(data)

    # Broadcast to connected dashboards
    socketio.emit('avatar_event', processed_data, namespace='/dashboard')

    return jsonify({'status': 'success'})

# Dashboard receives real-time updates
@socketio.on('connect', namespace='/dashboard')
def dashboard_connected():
    emit('status', {'message': 'Connected to avatar events'})
```

🤖 Smart Alerts

Intelligent notifications based on patterns:

```
class SmartAlertSystem:
    def __init__(self):
        self.user_sessions = {}
        self.error_patterns = {}

    def analyze_chat_event(self, data):
        message = data['message']['content'].lower()
        user_id = data['user']['id']

        # Detect frustration keywords
        frustration_words = ['frustrated', 'angry', 'terrible', 'awful']
        if any(word in message for word in frustration_words):
            self.escalate_to_human(data, reason='user_frustration')

        # Detect repeated questions
        if self.is_repeat_question(user_id, message):
            self.suggest_help_resources(data)

    def analyze_error_patterns(self, error_data):
        agent_id = error_data['agentId']
        error_code = error_data['error']['code']

        # Track error frequency
        key = f"{agent_id}:{error_code}"
        self.error_patterns[key] = self.error_patterns.get(key, 0) + 1

        # Alert if error rate exceeds threshold
        if self.error_patterns[key] > 5:  # 5 errors in window
            self.alert_operations_team({
                'agent_id': agent_id,
                'error_code': error_code,
                'frequency': self.error_patterns[key],
                'severity': 'high'
            })
```

📈 Analytics Pipeline

Feed events into analytics systems:

```
import boto3
from datetime import datetime

class AnalyticsPipeline:
    def __init__(self):
        self.kinesis = boto3.client('kinesis')
        self.s3 = boto3.client('s3')

    def process_webhook(self, event_data):
        # Stream to real-time analytics
        self.stream_to_kinesis(event_data)

        # Archive for batch processing
        self.archive_to_s3(event_data)
```

```
        # Update metrics in database
        self.update_metrics_db(event_data)

    def stream_to_kinesis(self, data):
        """Stream to AWS Kinesis for real-time analytics"""
        record = {
            'Data': json.dumps(data),
            'PartitionKey': data.get('agentId', 'default')
        }

        self.kinesis.put_record(
            StreamName='avatar-events',
            **record
        )

    def archive_to_s3(self, data):
        """Archive events for batch processing"""
        date = datetime.now().strftime('%Y-%m-%d')
        hour = datetime.now().strftime('%H')

        key = f"avatar-events/{date}/{hour}/{data['sessionId']}.json"

        self.s3.put_object(
            Bucket='avatar-analytics',
            Key=key,
            Body=json.dumps(data),
            ContentType='application/json'
        )
```

🛡️ Error Handling & Resilience

🔄 Retry Logic

Handle transient failures gracefully:

```
import time
import random
from functools import wraps

def retry_with_backoff(max_retries=3, base_delay=1):
    def decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            for attempt in range(max_retries):
                try:
                    return func(*args, **kwargs)
                except Exception as e:
                    if attempt == max_retries - 1:
                        raise e

                    # Exponential backoff with jitter
                    delay = base_delay * (2 ** attempt) + random.uniform(0, 1)
                    time.sleep(delay)
```

```
            return None
        return wrapper
    return decorator

@retry_with_backoff(max_retries=3)
def process_webhook_event(data):
    # Your processing logic here
    analytics_service.track_event(data)
    crm_service.update_contact(data)
```

💀 Dead Letter Queues

Handle failed events for later processing:

```
import redis
import json

class DeadLetterQueue:
    def __init__(self):
        self.redis_client = redis.Redis(host='localhost', port=6379)

    def add_failed_event(self, event_data, error_info):
        """Add failed event to DLQ for later processing"""
        dlq_item = {
            'event': event_data,
            'error': str(error_info),
            'failed_at': datetime.now().isoformat(),
            'retry_count': 0
        }

        self.redis_client.lpush('webhook_dlq', json.dumps(dlq_item))

    def process_dlq(self):
        """Process failed events from DLQ"""
        while True:
            item = self.redis_client.brpop('webhook_dlq', timeout=5)
            if not item:
                continue

            dlq_data = json.loads(item[1])

            try:
                # Retry processing
                process_webhook_event(dlq_data['event'])
                logger.info(f"DLQ item processed successfully")

            except Exception as e:
                dlq_data['retry_count'] += 1

                if dlq_data['retry_count'] < 3:
                    # Re-queue for retry
                    self.redis_client.lpush('webhook_dlq', json.dumps(dlq_data))
                else:
                    # Move to permanent failure queue
                    self.redis_client.lpush('webhook_failed', json.dumps(dlq_data
                    logger.error(f"DLQ item failed permanently: {e}")
```

🔍 Monitoring & Alerting

Track webhook health and performance:

```
import prometheus_client
from prometheus_client import Counter, Histogram, Gauge

# Metrics
webhook_requests = Counter('webhook_requests_total', 'Total webhook requests', [
processing_time = Histogram('webhook_processing_seconds', 'Time spent processing
active_sessions = Gauge('active_avatar_sessions', 'Current active avatar session

def track_webhook_metrics(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        start_time = time.time()
        event_type = request.json.get('event', 'unknown')

        try:
            result = func(*args, **kwargs)
            webhook_requests.labels(event_type=event_type, status='success').inc
            return result

        except Exception as e:
            webhook_requests.labels(event_type=event_type, status='error').inc()
            raise e

        finally:
            processing_time.observe(time.time() - start_time)

    return wrapper

@app.route('/webhook', methods=['POST'])
@track_webhook_metrics
def webhook_handler():
    # Your webhook logic here
    pass
```

🎉 Production Checklist

Before going live with your webhook integration:

✅ Security

- [ ] HTTPS endpoint with valid SSL certificate
- [ ] Webhook signature verification implemented
- [ ] IP whitelisting configured (optional)
- [ ] Authentication headers secured
- [ ] Input validation and sanitization

✅ Reliability

- [ ] Async processing for heavy operations
- [ ] Retry logic for transient failures

- [ ] Dead letter queue for failed events
- [ ] Monitoring and alerting configured
- [ ] Response time under 5 seconds

✅ Scalability

- [ ] Horizontal scaling capability
- [ ] Database connection pooling
- [ ] Message queue for high volume
- [ ] Rate limiting and throttling
- [ ] Circuit breaker pattern

✅ Monitoring

- [ ] Application metrics and logging
- [ ] Error tracking and alerting
- [ ] Performance monitoring
- [ ] Health check endpoint
- [ ] Dashboard for real-time visibility

🚀 Next Steps

1. 🔧 **Start simple** - Begin with basic event logging
2. 📊 **Add analytics** - Track user engagement and patterns
3. 🤖 **Build intelligence** - Add sentiment analysis and smart alerts
4. 🎯 **Optimize performance** - Implement async processing and caching
5. 📈 **Scale up** - Add queuing and horizontal scaling

Resources

- 📚 **Integration examples:** Webhook Integration Guide
- 💬 **Community support:** Discord
- 🛠️ **API reference:** bitHuman API Docs

Build powerful, real-time avatar integrations! 🎯

Previous
Webhook Integration
🔔 INTEGRATIONS