Celeste7 Webhook Security & Data Structure Specification

Critical Security Requirements

1. User Authentication Headers

Every webhook request MUST include these headers:

```
headers: {
  'X-User-Token': localStorage.getItem('userToken'), // JWT token from auth
  'X-Session-ID': sessionStorage.getItem('sessionId'), // Current session
  'X-Request-ID': generateUUID(), // Unique request ID for tracking
  'X-Timestamp': new Date().toISOString(), // Request timestamp
  'Content-Type': 'application/json'
}
```

2. User Isolation

- NEVER send one user's data in another user's request
- UserID must be extracted from JWT token server-side, not sent in body
- All queries must be scoped to authenticated user
- Implement request rate limiting (max 60 requests/minute per user)

Webhook Specifications

1. Voice Interaction Webhook

```
POST /api/voice-interaction
{
  // Required fields
  audio: Blob, // Audio file (WAV format)
  metadata: {
    sessionID: string, // Current session ID
    timestamp: string, // ISO timestamp
    duration: number, // Recording duration in seconds
    deviceInfo: {
      platform: string, // 'ios' | 'web'
      userAgent: string,
      timezone: string,
      language: string
    }
  },

  // Optional context for better responses
  context: {
    currentScreen: string, // Which app screen user is on
    recentCategory: string, // Last interaction category
    isInterruption: boolean, // If interrupting AI response
    conversationCount: number // Messages in current session
  }
}
```

```
// Expected Response
{
  success: boolean,
  data: {
    transcription: string,
    aiResponse: string,
    category: string,
    responseId: string,
    patterns: Array<{type: string, confidence: number}>,
    shouldTriggerIntervention: boolean
  }
}
```

2. User Patterns Webhook

```
GET /api/user-patterns
// No body - user identified by token
Query params: {
  timeframe: '7d' | '30d' | 'all',
  limit: number, // Max patterns to return (default 10)
  minConfidence: number // Min confidence threshold (0-1)
}

// Expected Response
{
  patterns: Array<{
    id: string,
    type: string,
    description: string,
    confidence: number,
    firstDetected: string,
    lastObserved: string,
    occurrenceCount: number,
    actionableInsight: string
  }>,
  summary: {
    totalPatterns: number,
    strongestPattern: string,
    needsAttention: boolean
  }
}
```

3. Goals Management Webhooks

```
GET /api/user-goals
Query params: {
  status: 'active' | 'completed' | 'all',
  limit: number
}

POST /api/goal-update
{
  action: 'create' | 'update' | 'complete' | 'archive',
  goalId: string, // Required for update/complete/archive
  data: {
    goalText: string,
    targetDate: string, // ISO date
    category: string,
    priority: 'high' | 'medium' | 'low'
  }
}

// Expected Response
{
  success: boolean,
  goal: {
    id: string,
    text: string,
    status: string,
    progress: number,
    createdAt: string,
    updatedAt: string
  }
}
```

4. Performance Metrics Webhook

```
GET /api/performance-metrics
Query params: {
  period: '7d' | '30d' | '90d',
  metrics: ['engagement', 'satisfaction', 'goals', 'patterns'] // Specific
metrics
}

// Expected Response
{
  metrics: {
    engagement: {
      totalInteractions: number,
      activeDays: number,
      averageDaily: number,
      streak: number
    },
    satisfaction: {
      rate: number, // 0-100
      positive: number,
      negative: number,
      trend: 'improving' | 'stable' | 'declining'
    },
    goals: {
      active: number,
      completed: number,
      completionRate: number
    },
    patterns: {
      total: number,
      addressed: number,
      recurring: Array<string>
    }
  },
  insights: Array<string> // AI-generated insights
}
```

5. Intervention Response Webhook

```
POST /api/intervention-response
{
  interventionId: string,
  response: 'helpful' | 'not_now' | 'dismiss',
  feedback: string, // Optional user comment
  timestamp: string,
  context: {
    currentActivity: string,
    timeSinceLastInteraction: number
  }
}
```

6. Conversation History Webhook

```
GET /api/conversation-history
Query params: {
  limit: number, // Max records (default 50)
  offset: number, // For pagination
  category: string, // Filter by category
  startDate: string, // ISO date
  endDate: string, // ISO date
  search: string // Search in conversations
}

// Expected Response
{
  conversations: Array<{
    id: string,
    timestamp: string,
    userInput: string,
    aiResponse: string,
    category: string,
    feedback: boolean | null,
    duration: number
  }>,
  pagination: {
    total: number,
    limit: number,
    offset: number,
    hasMore: boolean
  }
}
```

7. User Feedback Webhook

```
POST /api/user-feedback
{
  responseId: string, // ID of AI response
  feedback: boolean, // true = 👍, false = 👎
  category: string,
  reason: string, // Optional: why unhelpful
  timestamp: string
}
```

8. Real-time Interventions Webhook (WebSocket)

```
// WebSocket connection for real-time updates
ws://your-domain/ws/interventions

// Subscribe after auth
```

```
{
  type: 'subscribe',
  token: string // User JWT token
}

// Receive interventions
{
  type: 'intervention',
  data: {
    id: string,
    message: string,
    type: string,
    priority: 'high' | 'medium' | 'low',
    actionRequired: boolean
  }
}
```

Security Best Practices

1. Data Minimization

- Only send necessary data
- Never include sensitive info in URLs
- Sanitize all user inputs before sending

2. Request Validation

```
// Before sending any webhook
function validateRequest(data) {
  // Check required fields
  if (!data.sessionID || !data.timestamp) {
    throw new Error('Missing required fields');
  }

  // Validate timestamp is recent (prevent replay attacks)
  const requestTime = new Date(data.timestamp);
  const now = new Date();
  if (now - requestTime > 5 * 60 * 1000) { // 5 minutes
    throw new Error('Request timestamp too old');
  }

  // Sanitize strings
  Object.keys(data).forEach(key => {
    if (typeof data[key] === 'string') {
      data[key] = data[key].trim().slice(0, 1000); // Max 1000 chars
    }
  });

  return data;
}
```

3. Error Handling

```
// Standardized error handling
async function callWebhook(url, data) {
  try {
    const response = await fetch(url, {
      method: 'POST',
      headers: getSecureHeaders(),
      body: JSON.stringify(validateRequest(data))
    });

    if (!response.ok) {
      // Don't expose server errors to user
      throw new Error('Request failed');
    }

    return await response.json();
  } catch (error) {
    console.error('Webhook error:', error);
    // Return safe error message
    return {
      success: false,
      error: 'Unable to process request'
    };
  }
}
```

4. Performance Optimization

Request Debouncing

```
// Debounce pattern detection requests
const debouncedPatternCheck = debounce(checkPatterns, 1000);

// Batch feedback submissions
const feedbackQueue = [];
const flushFeedback = throttle(() => {
  if (feedbackQueue.length > 0) {
    sendBatchFeedback(feedbackQueue);
    feedbackQueue.length = 0;
  }
}, 5000);
```

Response Caching

```
// Cache patterns and metrics for 5 minutes
const cache = new Map();
const CACHE_DURATION = 5 * 60 * 1000;

async function getCachedData(key, fetchFn) {
  const cached = cache.get(key);
  if (cached && Date.now() - cached.timestamp < CACHE_DURATION) {
    return cached.data;
  }

  const data = await fetchFn();
  cache.set(key, { data, timestamp: Date.now() });
  return data;
}
```

5. Offline Queue

```
// Queue requests when offline
const offlineQueue = [];
```

```
function queueRequest(webhook, data) {
  offlineQueue.push({ webhook, data, timestamp: Date.now() });
  localStorage.setItem('offlineQueue', JSON.stringify(offlineQueue));
}

// Process queue when online
window.addEventListener('online', () => {
  const queue = JSON.parse(localStorage.getItem('offlineQueue') || '[]');
  queue.forEach(({ webhook, data }) => {
    callWebhook(webhook, data);
  });
  localStorage.removeItem('offlineQueue');
});
```

Critical Implementation Notes

1. **Never trust client-side data** - Always validate server-side
2. **Use HTTPS only** - Never send data over unencrypted connections
3. **Implement request signing** - Add HMAC signature for critical operations
4. **Rate limit by user** - Prevent abuse and ensure fair usage
5. **Log all requests** - For security auditing and debugging
6. **Fail gracefully** - Never expose internal errors to users
7. **Monitor webhook performance** - Set up alerts for slow responses

This specification ensures secure, efficient, and properly isolated user interactions while maintaining the speed required for a real-time voice interface.