~~~~
/**

- Complete Task Schema Version 3
- Designed for multi-device sync with comprehensive conflict resolution
  */

// TASK OBJECT SCHEMA v3
{
// 🆔 Primary Identification
id: “2a9c85b3-b9a5-4a2f-a9f8-1a1a9c5b3e9f”,  // UUID v4 for global uniqueness

// 📝 Core Task Properties
text: “Check air compressor”,
completed: false,
dueDate: “2025-08-09T15:30:00.000Z”,          // Full ISO 8601 with timezone
highPriority: true,

// 🔔 Individual Task Reminder Settings
remindersEnabled: true,
reminderSettings: {
enabled: true,                              // Explicit reminder toggle
startTime: “2025-08-08T14:00:00.000Z”,      // When reminders begin (ISO 8601)
frequencyValue: 60,                         // Every 60 units
frequencyUnit: “minutes”,                   // “minutes” | “hours” | “days”
lastFired: “2025-08-08T15:00:00.000Z”,      // Last reminder sent (ISO 8601)
maxReminders: null,                         // Limit reminders (null = unlimited)
timezone: “America/New_York”                // User’s timezone for reminders
},

// 🔁 Recurring Task Configuration
recurring: true,
recurringSettings: {
// Core recurrence settings
frequency: “daily”,                         // “hourly” | “daily” | “weekly” | “biweekly” | “monthly” | “yearly”
interval: 1,                                // Every X units (e.g., every 2 days)
timeOfDay: “08:00:00”,                      // HH:MM:SS format (24-hour)
timezone: “America/New_York”,               // Timezone for recurring execution

```
// Date boundaries
startDate: "2025-08-08T00:00:00.000Z",      // When recurrence begins (ISO 8601)
defaultRecurTime: "2025-08-08T14:00:00.000Z", // Fallback execution time

// End conditions
~~~~

```
endCondition: {
  type: "never",                    // "never" | "count" | "date"
  count: null,                      // Number of occurrences (if type = "count")
  endDate: null                     // Stop date (if type = "date") - ISO 8601
},

// Specific dates override (takes precedence over frequency)
specificDates: {
  enabled: false,
  dates: [                          // Array of specific ISO dates
    "2025-08-15T09:00:00.000Z",
    "2025-08-22T09:00:00.000Z"
  ]
},

// Frequency-specific settings
hourly: {
  useSpecificMinute: true,
  minute: 30                        // Run at :30 minutes past each hour
},

weekly: {
  useSpecificDays: true,
  days: ["Monday", "Wednesday", "Friday"]  // Days of week
},

biweekly: {
  useSpecificDays: true,
  days: ["Monday", "Thursday"]       // Days within biweekly cycle
},

monthly: {
  useSpecificDays: true,
  days: [1, 15, 30]                 // Days of month (1-31)
},

yearly: {
  useSpecificMonths: true,
  months: [1, 7, 12],               // Months (1-12)
  useSpecificDays: true,
  daysByMonth: {                    // Days per month
    "1": [1, 15],                   // January: 1st and 15th
    "7": [4],                       // July: 4th
    "12": [25]                      // December: 25th
```

~~~
  },
  applyDaysToAll: false                   // Apply same days to all months
},

// Time configuration
useSpecificTime: true,                   // Whether timeOfDay is enforced
lastTriggered: "2025-08-08T08:00:00.000Z" // Last execution timestamp
```

},

// 🔐 Sync & Conflict Resolution Metadata
metadata: {
// Timestamps (all ISO 8601)
createdAt: “2025-08-08T14:00:00.000Z”,
modifiedAt: “2025-08-08T17:30:00.000Z”,
lastSyncedAt: “2025-08-08T17:25:00.000Z”,
completedAt: null,                        // When task was completed

```
// Device tracking
createdByDevice: "device-MJPhone2025",
lastModifiedByDevice: "device-MJMacbook",

// Version control for conflict resolution
version: 3,                               // Incremental version number
syncStatus: "synced",                    // "synced" | "pending" | "conflict" | "error"

// Conflict resolution
conflictResolution: "manual",             // "auto" | "manual" | "newest-wins" | "device-priority"
conflictData: null,                       // Stores conflicting versions if needed

// Soft delete & change tracking
isDeleted: false,                         // Soft delete flag
isDirty: false,                           // Has unsaved changes

// Sync relationships
syncedWith: [                             // Devices this task is synced with
  "device-MJMacbook",
  "device-MJTablet"
]
```

},
~~~

```
// 🔢 Schema Version
schemaVersion: 3
}

/**

- Complete Mini Cycle Schema Version 3
- Container for task collections with sync capabilities
  */

// MINI CYCLE OBJECT SCHEMA v3
{
// 🆔 Primary Identification
id: “cycle-a1b2c3d4-e5f6-7890-abcd-ef1234567890”,  // UUID v4 for cycle identification
name: “morning-checks”,                             // Unique name identifier (URL-safe)
title: “Morning Safety Checks”,                     // Display name

// 📋 Task Management
tasks: [
// Array of Task objects (schema defined separately)
// Each task follows the Task Schema v3 specification
],

// 🔁 Recurring Task Templates
recurringTemplates: {
// Map of task IDs to their recurring templates
“2a9c85b3-b9a5-4a2f-a9f8-1a1a9c5b3e9f”: {
id: “2a9c85b3-b9a5-4a2f-a9f8-1a1a9c5b3e9f”,
text: “Check air compressor”,
recurring: true,
recurringSettings: { /* Full recurring settings object */ },
highPriority: true,
dueDate: null,
remindersEnabled: true,
reminderSettings: { /* Full reminder settings */ },
lastTriggeredTimestamp: “2025-08-08T08:00:00.000Z”,
suppressUntil: null,                         // Optional suppression date
schemaVersion: 3
}
},

// ⚙️ Cycle Behavior Settings
autoReset: true,                              // Auto-cycle when all tasks complete
```

```
deleteCheckedTasks: false,                        // Delete vs reset completed tasks

// 📊 Cycle Statistics
cycleCount: 11,                                   // Number of completed cycles
totalTasksCompleted: 145,                         // Lifetime task completions
averageCompletionTime: 1440,                      // Average minutes to complete cycle

// 🎯 Cycle Goals & Targets
goals: {
dailyTarget: 1,                                   // Target cycles per day
weeklyTarget: 7,                                  // Target cycles per week
streakCurrent: 5,                                 // Current completion streak
streakBest: 12                                    // Best completion streak
},

// 🔔 Cycle-Level Reminder Settings
reminderSettings: {
enabled: true,
cycleReminders: true,                             // Remind about incomplete cycles
overdueTaskReminders: true,                       // Remind about overdue tasks
completionCelebration: true,                      // Celebrate cycle completion
reminderFrequency: {
value: 2,
unit: “hours”                                     // Remind every 2 hours
}
},

// 🎨 UI/UX Preferences
preferences: {
theme: “default”,                                 // “default” | “dark-ocean” | “golden-glow”
showMoveArrows: false,                            // Show task reorder arrows
showThreeDots: true,                              // Show three-dot menu
alwaysShowRecurring: false,                       // Always show recurring buttons
taskDisplayMode: “list”,                          // “list” | “grid” | “compact”
sortOrder: “manual”                               // “manual” | “priority” | “dueDate” | “alphabetical”
},

// 🏷️ Organization & Categories
tags: [“work”, “safety”, “morning”],              // Category tags
category: “safety-checks”,                        // Primary category
color: “#4A90E2”,                                 // Cycle color (hex)
icon: “🔧”,                                       // Cycle emoji/icon

// 📅 Schedule & Timing
```

~~~
schedule: {
enabled: false,                     // Scheduled cycle execution
frequency: “daily”,                 // “daily” | “weekly” | “monthly”
time: “08:00:00”,                   // Scheduled time (HH:MM:SS)
timezone: “America/New_York”,               // Schedule timezone
daysOfWeek: [“Monday”, “Tuesday”, “Wednesday”, “Thursday”, “Friday”], // For weekly
daysOfMonth: [1, 15],                  // For monthly
autoStart: true,                   // Auto-start at scheduled time
notifications: true                // Send schedule notifications
},

// 🔗 Sharing & Collaboration
sharing: {
isShared: false,                    // Is cycle shared with others
shareMode: “read-only”,                 // “read-only” | “collaborative” | “template”
sharedWith: [],                    // Array of user/device IDs
shareCode: null,                    // Public sharing code
permissions: {
canEdit: false,
canAddTasks: false,
canComplete: false,
canShare: false
}
},

// 🔐 Sync & Conflict Resolution Metadata
metadata: {
// Timestamps (all ISO 8601)
createdAt: “2025-08-01T10:00:00.000Z”,
modifiedAt: “2025-08-08T17:30:00.000Z”,
lastSyncedAt: “2025-08-08T17:25:00.000Z”,
lastCompletedAt: “2025-08-08T09:15:00.000Z”,   // Last cycle completion
lastAccessedAt: “2025-08-08T17:30:00.000Z”,    // Last time cycle was opened

```
// Device tracking
createdByDevice: "device-MJPhone2025",
lastModifiedByDevice: "device-MJMacbook",
primaryDevice: "device-MJPhone2025",          // Main editing device

// Version control
version: 15,                       // Incremental version number
syncStatus: "synced",                  // "synced" | "pending" | "conflict" | "error"
~~~

```
// Conflict resolution
conflictResolution: "newest-wins",            // "auto" | "manual" | "newest-wins" | "device-priority"
conflictData: null,                          // Stores conflicting versions
mergeStrategy: "task-level",                 // "cycle-level" | "task-level" | "field-level"

// Change tracking
isDeleted: false,                            // Soft delete flag
isDirty: false,                              // Has unsaved changes
isArchived: false,                           // Archived status
isFavorite: true,                            // Favorite/pinned status

// Sync relationships
syncedWith: [                                // Devices this cycle is synced with
  "device-MJMacbook",
  "device-MJTablet"
],

// Backup & Export
lastBackupAt: "2025-08-07T00:00:00.000Z",    // Last backup timestamp
exportCount: 3,                              // Number of times exported
importedFrom: null                           // Source if imported from another system
```

},

// 🔢 Schema Version
schemaVersion: 3
}

/**

- Complete Device Schema Version 3
- Device registration and sync management
  */

// DEVICE OBJECT SCHEMA v3
{
// 🆔 Device Identification
deviceID: “device-2f8a9b3c-d4e5-6789-abc1-23456789def0”, // UUID v4 for device
deviceName: “MJ’s iPhone”,   // User-friendly device name
deviceType: “mobile”,   // “mobile” | “desktop” | “tablet” | “web”
platform: “ios”,   // “ios” | “android” | “windows” | “macos” | “linux” | “web”

// 📱 Device Specifications

```
deviceInfo: {
manufacturer: “Apple”,                              // Device manufacturer
model: “iPhone 15 Pro”,                            // Device model
osVersion: “iOS 17.4.1”,                           // Operating system version
screenSize: “6.1 inch”,                            // Screen dimensions
resolution: “2556x1179”,                           // Screen resolution
userAgent: “Mozilla/5.0 (iPhone; CPU iPhone OS 17_4_1 like Mac OS X)…” // Browser user
agent (if web)
},

// 🔧 App Configuration
appInfo: {
version: “1.2.3”,                                  // App version number
buildNumber: “142”,                                // Internal build number
installationID: “inst-4c7d9e2f-8a1b-5678-def0-123456789abc”, // Unique installation
firstInstallDate: “2025-07-15T10:30:00.000Z”,         // First app installation
lastUpdateDate: “2025-08-01T14:20:00.000Z”            // Last app update
},

// 🌍 Localization & Timezone
localization: {
timezone: “America/New_York”,                          // Device timezone
locale: “en-US”,                                   // Language/region code
dateFormat: “MM/dd/yyyy”,                             // Preferred date format
timeFormat: “12”,                                  // “12” | “24” hour format
firstDayOfWeek: “Sunday”,                             // “Sunday” | “Monday”
currency: “USD”                                    // Preferred currency
},

// 🔗 Registration & Pairing
registration: {
registeredAt: “2025-08-01T07:55:00.000Z”,             // Device registration timestamp
registrationMethod: “manual”,                        // “manual” | “qr-code” | “link” | “auto”
activationCode: “ABC123”,                            // Temporary activation code (if used)
verificationStatus: “verified”,                     // “pending” | “verified” | “expired”
pairedWith: [                                     // Connected devices
“device-MJMacbook-2025”,
“device-MJTablet-Air”
]
},

// 📡 Sync Configuration
syncSettings: {
enabled: true,                                     // Master sync toggle
```

~~~~
syncMethod: “realtime”,                    // “realtime” | “interval” | “manual”
syncInterval: 300,                         // Seconds between syncs (if interval)
wifiOnly: false,                           // Only sync on WiFi
backgroundSync: true,                      // Allow background syncing
conflictResolution: “manual”,              // “auto” | “manual” | “newest-wins”
maxRetries: 3,                             // Max sync retry attempts
retryDelay: 30,                            // Seconds between retries

```
// Data type sync preferences
syncPreferences: {
  tasks: true,                             // Sync task data
  cycles: true,                            // Sync cycle data
  settings: true,                          // Sync app settings
  themes: true,                            // Sync theme preferences
  statistics: false,                       // Sync usage statistics
  backups: true                            // Sync backup data
}
```

},

// 🔐 Security & Authentication
security: {
encryptionEnabled: true,                   // Local encryption status
encryptionLevel: “AES-256”,                // Encryption algorithm
biometricEnabled: true,                    // Biometric authentication
biometricType: “TouchID”,                  // “TouchID” | “FaceID” | “Fingerprint”
pinEnabled: false,                         // PIN protection
autoLockEnabled: true,                     // Auto-lock after inactivity
autoLockDelay: 300,                        // Seconds until auto-lock
remoteWipeEnabled: true,                   // Allow remote data wipe
trustedDevice: true                        // Is device trusted for sync
},

// 📊 Usage Statistics
usage: {
totalSessions: 247,                        // Total app sessions
totalUsageTime: 18420,                     // Total usage time (seconds)
averageSessionTime: 74.6,                  // Average session length (seconds)
lastActiveAt: “2025-08-08T17:30:00.000Z”,  // Last activity timestamp
featuresUsed: {                            // Feature usage tracking
tasksCreated: 156,
tasksCompleted: 142,
~~~~

```
cyclesCompleted: 23,
themesChanged: 3,
exportsCreated: 2
},
crashReports: 0,                                    // Number of app crashes
errorReports: 1                                     // Number of error reports
},

// 🔄 Sync History & Status
syncHistory: {
lastSyncedAt: “2025-08-08T17:15:00.000Z”,           // Last successful sync
lastSyncDuration: 2.3,                              // Last sync duration (seconds)
totalSyncs: 89,                                     // Total successful syncs
failedSyncs: 2,                                     // Total failed syncs
lastSyncError: null,                                // Last sync error message
syncQueueSize: 0,                                   // Pending changes to sync

```
// Sync performance metrics
averageSyncTime: 1.8,                               // Average sync duration
largestSyncSize: 245760,                            // Largest sync payload (bytes)
totalDataSynced: 5242880,                           // Total data synced (bytes)

// Conflict resolution history
conflictsResolved: 3,                               // Total conflicts resolved
lastConflictAt: "2025-08-05T14:22:00.000Z",         // Last conflict timestamp
autoResolvedConflicts: 1,                           // Auto-resolved conflicts
manualResolvedConflicts: 2                          // Manually resolved conflicts
```


},

// 🎯 Device Preferences
preferences: {
notifications: {
enabled: true,   // Master notification toggle
sound: true,   // Notification sounds
vibration: true,   // Vibration (mobile)
badge: true,   // App badge numbers
types: {   // Notification type preferences
taskReminders: true,
cycleCompletions: true,
syncUpdates: false,
systemMessages: true
```

~~~
}
},

```
performance: {
  animationsEnabled: true,                    // UI animations
  hapticFeedback: true,                       // Haptic feedback (mobile)
  reducedMotion: false,                       // Accessibility: reduced motion
  powerSaving: false,                         // Power saving mode
  backgroundRefresh: true                     // Background app refresh
}
```

},

// 🏷️ Device Management
management: {
isActive: true,   // Device active status
isPrimary: false,   // Is primary device for account
trustLevel: "high",   // "low" | "medium" | "high"
managedBy: null,   // Managing device ID (if any)
restrictions: [],   // Device restrictions array
lastHeartbeat: "2025-08-08T17:30:00.000Z",   // Last connectivity check

```
// Remote management capabilities
remoteCommands: {
  wipeSupported: true,                        // Supports remote wipe
  lockSupported: true,                        // Supports remote lock
  locateSupported: false,                     // Supports device location
  logSupported: true                          // Supports remote logging
}
```

},

// 🔐 Metadata
metadata: {
createdAt: "2025-08-01T07:55:00.000Z",   // Device registration timestamp
modifiedAt: "2025-08-08T17:30:00.000Z",   // Last metadata update
version: 8,   // Device record version
syncStatus: "active",   // "active" | "inactive" | "suspended"
isDeleted: false,   // Soft delete flag
tags: ["personal", "primary"]   // Device organization tags
~~~

```
},

// 🔢 Schema Version
schemaVersion: 3
}

/**

- Migration Functions: Schema v2 → v3
- Handles data transformation and UUID generation
  */

// Utility function to generate UUID v4
function generateUUID() {
if (typeof crypto !== ‘undefined’ && crypto.randomUUID) {
return crypto.randomUUID();
}
// Fallback UUID generation
return ‘xxxxxxxx-xxxx-4xxx-yxxx-xxxxxxxxxxxx’.replace(/[xy]/g, function(c) {
const r = Math.random() * 16 | 0;
const v = c === ‘x’ ? r : (r & 0x3 | 0x8);
return v.toString(16);
});
}

// Get current device ID (generate if not exists)
function getCurrentDeviceId() {
let deviceId = localStorage.getItem(‘deviceId’);
if (!deviceId) {
deviceId = `device-${generateUUID()}`;
localStorage.setItem(‘deviceId’, deviceId);
}
return deviceId;
}

// Convert date string to ISO 8601 format
function toISOString(dateInput) {
if (!dateInput) return null;

try {
// If already ISO format, return as is
if (typeof dateInput === ‘string’ && dateInput.includes(‘T’)) {
return new Date(dateInput).toISOString();
}
```

~~~~
```
// If date-only format (YYYY-MM-DD), append time
if (typeof dateInput === 'string' && /^\d{4}-\d{2}-\d{2}$/.test(dateInput)) {
  return new Date(dateInput + 'T00:00:00.000Z').toISOString();
}

// Convert any other format
return new Date(dateInput).toISOString();
```

} catch (error) {
console.warn(‘Invalid date format:’, dateInput);
return null;
}
}

/**

- Migrate Task from v2 to v3
  */
  function migrateTaskToV3(taskV2, globalReminderSettings = {}) {
  const now = new Date().toISOString();
  const deviceId = getCurrentDeviceId();

// Generate new UUID if old ID format
const newId = taskV2.id && !taskV2.id.startsWith(‘task-’) ?
taskV2.id : generateUUID();

const taskV3 = {
// Core identification
id: newId,
text: taskV2.text || “Untitled Task”,
completed: Boolean(taskV2.completed),
dueDate: toISOString(taskV2.dueDate),
highPriority: Boolean(taskV2.highPriority),

```
// Migrate reminder settings
remindersEnabled: Boolean(taskV2.remindersEnabled),
reminderSettings: {
  enabled: Boolean(taskV2.remindersEnabled),
  startTime: globalReminderSettings.reminderStartTime ?
    toISOString(globalReminderSettings.reminderStartTime) : null,
~~~~

~~~
  frequencyValue: globalReminderSettings.frequencyValue || 60,
  frequencyUnit: globalReminderSettings.frequencyUnit || "minutes",
  lastFired: null,
  maxReminders: globalReminderSettings.indefinite ? null :
globalReminderSettings.repeatCount,
  timezone: Intl.DateTimeFormat().resolvedOptions().timeZone
},

// Migrate recurring settings
recurring: Boolean(taskV2.recurring),
recurringSettings: migrateRecurringSettingsToV3(taskV2.recurringSettings || {}),

// Add comprehensive metadata
metadata: {
  createdAt: now,
  modifiedAt: now,
  lastSyncedAt: null,
  completedAt: taskV2.completed ? now : null,
  createdByDevice: deviceId,
  lastModifiedByDevice: deviceId,
  version: 1,
  syncStatus: "pending",
  conflictResolution: "manual",
  conflictData: null,
  isDeleted: false,
  isDirty: true,
  syncedWith: []
},

schemaVersion: 3
```

};

return taskV3;
}

/**

- Migrate Recurring Settings from v2 to v3
  */
  function migrateRecurringSettingsToV3(recurringV2) {
  if (!recurringV2 || Object.keys(recurringV2).length === 0) {
  return {
~~~

~~~~
  frequency: “daily”,
  interval: 1,
  timeOfDay: “08:00:00”,
  timezone: Intl.DateTimeFormat().resolvedOptions().timeZone,
  startDate: new Date().toISOString(),
  defaultRecurTime: new Date().toISOString(),
  endCondition: { type: “never”, count: null, endDate: null },
  specificDates: { enabled: false, dates: [] },
  hourly: { useSpecificMinute: false, minute: 0 },
  weekly: { useSpecificDays: false, days: [] },
  biweekly: { useSpecificDays: false, days: [] },
  monthly: { useSpecificDays: false, days: [] },
  yearly: {
  useSpecificMonths: false,
  months: [],
  useSpecificDays: false,
  daysByMonth: {},
  applyDaysToAll: false
  },
  useSpecificTime: false,
  lastTriggered: null
  };
  }

const timezone = Intl.DateTimeFormat().resolvedOptions().timeZone;

return {
frequency: recurringV2.frequency || “daily”,
interval: 1,
timeOfDay: formatTimeToV3(recurringV2.time),
timezone: timezone,
startDate: toISOString(recurringV2.startDate) || new Date().toISOString(),
defaultRecurTime: toISOString(recurringV2.defaultRecurTime) || new Date().toISOString(),

```
endCondition: {
  type: recurringV2.recurIndefinitely !== false ? "never" : "count",
  count: recurringV2.recurCount || null,
  endDate: null
},

specificDates: {
  enabled: Boolean(recurringV2.specificDates?.enabled),
  dates: (recurringV2.specificDates?.dates || []).map(toISOString).filter(Boolean)
~~~~

~~~~
},

// Migrate frequency-specific settings
hourly: {
  useSpecificMinute: Boolean(recurringV2.hourly?.useSpecificMinute),
  minute: recurringV2.hourly?.minute || 0
},

weekly: {
  useSpecificDays: Boolean(recurringV2.weekly?.useSpecificDays),
  days: recurringV2.weekly?.days || []
},

biweekly: {
  useSpecificDays: Boolean(recurringV2.biweekly?.useSpecificDays),
  days: recurringV2.biweekly?.days || []
},

monthly: {
  useSpecificDays: Boolean(recurringV2.monthly?.useSpecificDays),
  days: recurringV2.monthly?.days || []
},

yearly: {
  useSpecificMonths: Boolean(recurringV2.yearly?.useSpecificMonths),
  months: recurringV2.yearly?.months || [],
  useSpecificDays: Boolean(recurringV2.yearly?.useSpecificDays),
  daysByMonth: recurringV2.yearly?.daysByMonth || {},
  applyDaysToAll: Boolean(recurringV2.yearly?.applyDaysToAll)
},

useSpecificTime: Boolean(recurringV2.useSpecificTime),
lastTriggered: toISOString(recurringV2.lastTriggeredTimestamp)
```

};
}

/**

- Format time from v2 to v3 (ensure HH:MM:SS format)
  */
  function formatTimeToV3(timeV2) {
  if (!timeV2) return “08:00:00”;
~~~~

~~~
if (typeof timeV2 === ‘object’ && timeV2.hour !== undefined) {
// Convert from time object
const hour = String(timeV2.hour || 8).padStart(2, ‘0’);
const minute = String(timeV2.minute || 0).padStart(2, ‘0’);
return `${hour}:${minute}:00`;
}

if (typeof timeV2 === ‘string’) {
// Ensure HH:MM:SS format
if (timeV2.match(/^\d{1,2}:\d{2}$/)) {
return timeV2 + “:00”;
}
if (timeV2.match(/^\d{1,2}:\d{2}:\d{2}$/)) {
return timeV2;
}
}

return “08:00:00”; // Default fallback
}

/**

- Migrate Mini Cycle from v2 to v3
  */
  function migrateMiniCycleToV3(cycleV2, cycleName) {
  const now = new Date().toISOString();
  const deviceId = getCurrentDeviceId();

// Load global reminder settings for task migration
const globalReminders = JSON.parse(localStorage.getItem(“miniCycleReminders”) || “{}”);

const cycleV3 = {
// Core identification
id: generateUUID(),
name: cycleName || “untitled-cycle”,
title: cycleV2.title || cycleName || “Untitled Cycle”,

```
// Migrate tasks
tasks: (cycleV2.tasks || []).map(task => migrateTaskToV3(task, globalReminders)),

// Migrate recurring templates
recurringTemplates: migrateRecurringTemplatesToV3(cycleV2.recurringTemplates || {}),
~~~

```
// Core settings
autoReset: Boolean(cycleV2.autoReset),
deleteCheckedTasks: Boolean(cycleV2.deleteCheckedTasks),

// Statistics
cycleCount: cycleV2.cycleCount || 0,
totalTasksCompleted: calculateTotalTasksCompleted(cycleV2),
averageCompletionTime: 1440, // Default to 24 hours

// Goals (new in v3)
goals: {
  dailyTarget: 1,
  weeklyTarget: 7,
  streakCurrent: 0,
  streakBest: 0
},

// Migrate reminder settings
reminderSettings: {
  enabled: globalReminders.enabled || false,
  cycleReminders: true,
  overdueTaskReminders: globalReminders.dueDatesReminders || false,
  completionCelebration: true,
  reminderFrequency: {
    value: globalReminders.frequencyValue || 2,
    unit: mapFrequencyUnit(globalReminders.frequencyUnit)
  }
},

// Default preferences
preferences: {
  theme: localStorage.getItem('currentTheme') || "default",
  showMoveArrows: localStorage.getItem("miniCycleMoveArrows") === "true",
  showThreeDots: localStorage.getItem("miniCycleThreeDots") === "true",
  alwaysShowRecurring: JSON.parse(localStorage.getItem("miniCycleAlwaysShowRecurring") ||
"false"),
  taskDisplayMode: "list",
  sortOrder: "manual"
},

// Organization (new in v3)
tags: [],
category: null,
color: "#4A90E2",
icon: "📋",

// Schedule (new in v3)
schedule: {
  enabled: false,
  frequency: "daily",
  time: "08:00:00",
  timezone: Intl.DateTimeFormat().resolvedOptions().timeZone,
  daysOfWeek: ["Monday", "Tuesday", "Wednesday", "Thursday", "Friday"],
  daysOfMonth: [1, 15],
  autoStart: false,
  notifications: true
},

// Sharing (new in v3)
sharing: {
  isShared: false,
  shareMode: "read-only",
  sharedWith: [],
  shareCode: null,
  permissions: {
    canEdit: false,
    canAddTasks: false,
    canComplete: false,
    canShare: false
  }
},

// Comprehensive metadata
metadata: {
  createdAt: now,
  modifiedAt: now,
  lastSyncedAt: null,
  lastCompletedAt: null,
  lastAccessedAt: now,
  createdByDevice: deviceId,
  lastModifiedByDevice: deviceId,
  primaryDevice: deviceId,
  version: 1,
  syncStatus: "pending",
  conflictResolution: "newest-wins",
  conflictData: null,
  mergeStrategy: "task-level",
```

~~~
  isDeleted: false,
  isDirty: true,
  isArchived: false,
  isFavorite: false,
  syncedWith: [],
  lastBackupAt: null,
  exportCount: 0,
  importedFrom: null
},

schemaVersion: 3
```

};

return cycleV3;
}

/**

- Migrate recurring templates from v2 to v3
  */
  function migrateRecurringTemplatesToV3(templatesV2) {
  const migratedTemplates = {};

Object.entries(templatesV2).forEach(([taskId, template]) => {
const newId = taskId.startsWith('task-') ? generateUUID() : taskId;

```
migratedTemplates[newId] = {
  id: newId,
  text: template.text || "Untitled Task",
  recurring: true,
  recurringSettings: migrateRecurringSettingsToV3(template.recurringSettings || {}),
  highPriority: Boolean(template.highPriority),
  dueDate: toISOString(template.dueDate),
  remindersEnabled: Boolean(template.remindersEnabled),
  reminderSettings: {
    enabled: Boolean(template.remindersEnabled),
    startTime: null,
    frequencyValue: 60,
    frequencyUnit: "minutes",
    lastFired: null,
    maxReminders: null,
~~~

~~~~
    timezone: Intl.DateTimeFormat().resolvedOptions().timeZone
  },
  lastTriggeredTimestamp: toISOString(template.lastTriggeredTimestamp),
  suppressUntil: null,
  schemaVersion: 3
};
```

});

return migratedTemplates;
}

/**

- Create new Device record for v3
  */
  function createDeviceRecordV3() {
  const deviceId = getCurrentDeviceId();
  const now = new Date().toISOString();

// Detect device information
const deviceInfo = detectDeviceInfo();

const deviceV3 = {
deviceID: deviceId,
deviceName: deviceInfo.deviceName,
deviceType: deviceInfo.deviceType,
platform: deviceInfo.platform,

```
deviceInfo: {
  manufacturer: deviceInfo.manufacturer,
  model: deviceInfo.model,
  osVersion: deviceInfo.osVersion,
  screenSize: deviceInfo.screenSize,
  resolution: deviceInfo.resolution,
  userAgent: navigator.userAgent
},

appInfo: {
  version: "1.0.0", // Default version
  buildNumber: "1",
  installationID: generateUUID(),
~~~~

```
  firstInstallDate: now,
  lastUpdateDate: now
},

localization: {
  timezone: Intl.DateTimeFormat().resolvedOptions().timeZone,
  locale: navigator.language || "en-US",
  dateFormat: "MM/dd/yyyy",
  timeFormat: "12",
  firstDayOfWeek: "Sunday",
  currency: "USD"
},

registration: {
  registeredAt: now,
  registrationMethod: "auto",
  activationCode: null,
  verificationStatus: "verified",
  pairedWith: []
},

syncSettings: {
  enabled: true,
  syncMethod: "realtime",
  syncInterval: 300,
  wifiOnly: false,
  backgroundSync: true,
  conflictResolution: "manual",
  maxRetries: 3,
  retryDelay: 30,
  syncPreferences: {
    tasks: true,
    cycles: true,
    settings: true,
    themes: true,
    statistics: false,
    backups: true
  }
},

security: {
  encryptionEnabled: false,
  encryptionLevel: "AES-256",
  biometricEnabled: false,
  biometricType: null,
  pinEnabled: false,
  autoLockEnabled: false,
  autoLockDelay: 300,
  remoteWipeEnabled: false,
  trustedDevice: true
},

usage: {
  totalSessions: 1,
  totalUsageTime: 0,
  averageSessionTime: 0,
  lastActiveAt: now,
  featuresUsed: {
    tasksCreated: 0,
    tasksCompleted: 0,
    cyclesCompleted: 0,
    themesChanged: 0,
    exportsCreated: 0
  },
  crashReports: 0,
  errorReports: 0
},

syncHistory: {
  lastSyncedAt: null,
  lastSyncDuration: 0,
  totalSyncs: 0,
  failedSyncs: 0,
  lastSyncError: null,
  syncQueueSize: 0,
  averageSyncTime: 0,
  largestSyncSize: 0,
  totalDataSynced: 0,
  conflictsResolved: 0,
  lastConflictAt: null,
  autoResolvedConflicts: 0,
  manualResolvedConflicts: 0
},

preferences: {
  notifications: {
    enabled: true,
    sound: true,
    vibration: true,
    badge: true,
    types: {
      taskReminders: true,
      cycleCompletions: true,
      syncUpdates: false,
      systemMessages: true
    }
  },
  performance: {
    animationsEnabled: true,
    hapticFeedback: true,
    reducedMotion: false,
    powerSaving: false,
    backgroundRefresh: true
  }
},

management: {
  isActive: true,
  isPrimary: true,
  trustLevel: "high",
  managedBy: null,
  restrictions: [],
  lastHeartbeat: now,
  remoteCommands: {
    wipeSupported: false,
    lockSupported: false,
    locateSupported: false,
    logSupported: false
  }
},

metadata: {
  createdAt: now,
  modifiedAt: now,
  version: 1,
  syncStatus: "active",
  isDeleted: false,
  tags: ["personal"]
},

schemaVersion: 3
```

```
};

return deviceV3;
}

/**

- Utility functions
  */

function calculateTotalTasksCompleted(cycleV2) {
if (!cycleV2.tasks) return 0;
return cycleV2.tasks.filter(task => task.completed).length;
}

function mapFrequencyUnit(oldUnit) {
const unitMap = {
“minutes”: “minutes”,
“hours”: “hours”,
“days”: “days”
};
return unitMap[oldUnit] || “hours”;
}

function detectDeviceInfo() {
const ua = navigator.userAgent;
let deviceInfo = {
deviceName: “Unknown Device”,
deviceType: “web”,
platform: “web”,
manufacturer: “Unknown”,
model: “Unknown”,
osVersion: “Unknown”,
screenSize: `${screen.width}x${screen.height}`,
resolution: `${screen.width}x${screen.height}`
};

// Mobile detection
if (/iPhone/.test(ua)) {
deviceInfo.deviceType = “mobile”;
deviceInfo.platform = “ios”;
deviceInfo.manufacturer = “Apple”;
deviceInfo.deviceName = “iPhone”;
```

~~~
deviceInfo.model = “iPhone”;
} else if (/iPad/.test(ua)) {
deviceInfo.deviceType = “tablet”;
deviceInfo.platform = “ios”;
deviceInfo.manufacturer = “Apple”;
deviceInfo.deviceName = “iPad”;
deviceInfo.model = “iPad”;
} else if (/Android/.test(ua)) {
deviceInfo.deviceType = /Mobile/.test(ua) ? “mobile” : “tablet”;
deviceInfo.platform = “android”;
deviceInfo.manufacturer = “Android”;
deviceInfo.deviceName = “Android Device”;
deviceInfo.model = “Android”;
} else if (/Windows/.test(ua)) {
deviceInfo.deviceType = “desktop”;
deviceInfo.platform = “windows”;
deviceInfo.manufacturer = “PC”;
deviceInfo.deviceName = “Windows PC”;
deviceInfo.model = “PC”;
} else if (/Macintosh/.test(ua)) {
deviceInfo.deviceType = “desktop”;
deviceInfo.platform = “macos”;
deviceInfo.manufacturer = “Apple”;
deviceInfo.deviceName = “Mac”;
deviceInfo.model = “Mac”;
}

return deviceInfo;
}

/**

- Main migration function
  */
  function migrateAllDataToV3() {
  console.log(“🔄 Starting migration to Schema v3…”);

try {
// 1. Create device record
const deviceRecord = createDeviceRecordV3();
localStorage.setItem(‘deviceRecord’, JSON.stringify(deviceRecord));

```
// 2. Migrate mini cycles
~~~

~~~~
const savedMiniCycles = JSON.parse(localStorage.getItem("miniCycleStorage") || "{}");
const migratedCycles = {};

Object.entries(savedMiniCycles).forEach(([cycleName, cycleData]) => {
  if (cycleData.schemaVersion === 3) {
    migratedCycles[cycleName] = cycleData; // Already v3
  } else {
    migratedCycles[cycleName] = migrateMiniCycleToV3(cycleData, cycleName);
  }
});

// 3. Save migrated data
localStorage.setItem("miniCycleStorage", JSON.stringify(migratedCycles));

// 4. Update schema version marker
localStorage.setItem("miniCycleSchemaVersion", "3");

console.log("✅ Migration to Schema v3 completed successfully!");

return {
  success: true,
  migratedCycles: Object.keys(migratedCycles).length,
  deviceRecord: deviceRecord.deviceID
};
```

} catch (error) {
console.error("❌ Migration failed:", error);
return {
success: false,
error: error.message
};
}
}

// Export functions for use
if (typeof module !== 'undefined' && module.exports) {
module.exports = {
migrateTaskToV3,
migrateMiniCycleToV3,
createDeviceRecordV3,
migrateAllDataToV3,
generateUUID,
toISOString
~~~~

```
};
}

/**

- Conflict Resolution System for Schema v3
- Handles data conflicts during sync operations
  */

// Conflict resolution strategies
const CONFLICT_STRATEGIES = {
AUTO: ‘auto’,
MANUAL: ‘manual’,
NEWEST_WINS: ‘newest-wins’,
DEVICE_PRIORITY: ‘device-priority’,
FIELD_LEVEL: ‘field-level’
};

// Conflict types
const CONFLICT_TYPES = {
VERSION: ‘version’,        // Version number mismatch
TIMESTAMP: ‘timestamp’,    // Modified timestamp conflict
CONTENT: ‘content’,        // Data content differs
DELETION: ‘deletion’,      // One side deleted, other modified
CREATION: ‘creation’       // Same ID created on multiple devices
};

/**

- Main conflict resolution function
  */
  function resolveConflict(localData, remoteData, strategy =
CONFLICT_STRATEGIES.MANUAL) {
  // Pre-validation
  if (!localData || !remoteData) {
  throw new Error(‘Both local and remote data are required for conflict resolution’);
  }

// Detect conflict type
const conflictType = detectConflictType(localData, remoteData);

// Log conflict detection
console.log(`🔍 Conflict detected: ${conflictType}`, {
local: {
```

~~~~
id: localData.id,
version: localData.metadata?.version,
modifiedAt: localData.metadata?.modifiedAt
},
remote: {
id: remoteData.id,
version: remoteData.metadata?.version,
modifiedAt: remoteData.metadata?.modifiedAt
}
});

// Apply resolution strategy
switch (strategy) {
case CONFLICT_STRATEGIES.AUTO:
return autoResolveConflict(localData, remoteData, conflictType);

```
case CONFLICT_STRATEGIES.NEWEST_WINS:
  return newestWinsResolution(localData, remoteData);

case CONFLICT_STRATEGIES.DEVICE_PRIORITY:
  return devicePriorityResolution(localData, remoteData);

case CONFLICT_STRATEGIES.FIELD_LEVEL:
  return fieldLevelResolution(localData, remoteData);

case CONFLICT_STRATEGIES.MANUAL:
default:
  return prepareManualResolution(localData, remoteData, conflictType);
```

}
}

/**

- Detect the type of conflict
  */
  function detectConflictType(localData, remoteData) {
  const localMeta = localData.metadata || {};
  const remoteMeta = remoteData.metadata || {};

// Check for deletion conflicts
if (localMeta.isDeleted && !remoteMeta.isDeleted) {
~~~~

```
return CONFLICT_TYPES.DELETION;
}
if (!localMeta.isDeleted && remoteMeta.isDeleted) {
return CONFLICT_TYPES.DELETION;
}

// Check version conflicts
if (localMeta.version !== remoteMeta.version) {
return CONFLICT_TYPES.VERSION;
}

// Check timestamp conflicts
if (localMeta.modifiedAt !== remoteMeta.modifiedAt) {
return CONFLICT_TYPES.TIMESTAMP;
}

// Check content conflicts
if (JSON.stringify(excludeMetadata(localData)) !==
JSON.stringify(excludeMetadata(remoteData))) {
return CONFLICT_TYPES.CONTENT;
}

return CONFLICT_TYPES.CONTENT; // Default fallback
}

/**

- Auto-resolve conflict using intelligent heuristics
  */
  function autoResolveConflict(localData, remoteData, conflictType) {
  switch (conflictType) {
  case CONFLICT_TYPES.DELETION:
  // If one side is deleted, prefer deletion
  return localData.metadata?.isDeleted || remoteData.metadata?.isDeleted ?
  (localData.metadata?.isDeleted ? localData : remoteData) :
  newestWinsResolution(localData, remoteData);

  case CONFLICT_TYPES.VERSION:
  // Higher version wins
  const localVersion = localData.metadata?.version || 0;
  const remoteVersion = remoteData.metadata?.version || 0;
  return localVersion > remoteVersion ? localData : remoteData;

  case CONFLICT_TYPES.TIMESTAMP:
```

```
  case CONFLICT_TYPES.CONTENT:
  default:
  // Fall back to newest wins
  return newestWinsResolution(localData, remoteData);
  }
  }

/**

- Newest timestamp wins resolution
  */
  function newestWinsResolution(localData, remoteData) {
  const localTime = new Date(localData.metadata?.modifiedAt || 0);
  const remoteTime = new Date(remoteData.metadata?.modifiedAt || 0);

const winner = localTime > remoteTime ? localData : remoteData;
const loser = winner === localData ? remoteData : localData;

return {
resolved: true,
strategy: CONFLICT_STRATEGIES.NEWEST_WINS,
result: mergeMetadata(winner, loser),
conflictData: {
winner: winner === localData ? ‘local’ : ‘remote’,
reason: ‘newer_timestamp’,
localTime: localTime.toISOString(),
remoteTime: remoteTime.toISOString()
}
};
}

/**

- Device priority resolution (trusted device wins)
  */
  function devicePriorityResolution(localData, remoteData) {
  const deviceRecord = JSON.parse(localStorage.getItem(‘deviceRecord’) || ‘{}’);
  const trustedDevices = deviceRecord.registration?.pairedWith || [];

const localDevice = localData.metadata?.lastModifiedByDevice;
const remoteDevice = remoteData.metadata?.lastModifiedByDevice;

// Current device always has highest priority
if (localDevice === deviceRecord.deviceID) {
```

```
return {
resolved: true,
strategy: CONFLICT_STRATEGIES.DEVICE_PRIORITY,
result: mergeMetadata(localData, remoteData),
conflictData: {
winner: ‘local’,
reason: ‘current_device_priority’
}
};
}

// Check trusted device priority
const localTrusted = trustedDevices.includes(localDevice);
const remoteTrusted = trustedDevices.includes(remoteDevice);

if (localTrusted && !remoteTrusted) {
return {
resolved: true,
strategy: CONFLICT_STRATEGIES.DEVICE_PRIORITY,
result: mergeMetadata(localData, remoteData),
conflictData: { winner: ‘local’, reason: ‘trusted_device’ }
};
}

if (!localTrusted && remoteTrusted) {
return {
resolved: true,
strategy: CONFLICT_STRATEGIES.DEVICE_PRIORITY,
result: mergeMetadata(remoteData, localData),
conflictData: { winner: ‘remote’, reason: ‘trusted_device’ }
};
}

// Fall back to newest wins if both or neither are trusted
return newestWinsResolution(localData, remoteData);
}

/**

- Field-level resolution (merge compatible fields)
  */
  function fieldLevelResolution(localData, remoteData) {
  const merged = JSON.parse(JSON.stringify(localData)); // Deep clone
  const changes = [];
```

~~~
// Compare each top-level field
Object.keys(remoteData).forEach(key => {
if (key === ‘metadata’) return; // Handle metadata separately

```
const localValue = localData[key];
const remoteValue = remoteData[key];

if (JSON.stringify(localValue) !== JSON.stringify(remoteValue)) {
  // Use newer timestamp for this field if available
  const localTime = new Date(localData.metadata?.modifiedAt || 0);
  const remoteTime = new Date(remoteData.metadata?.modifiedAt || 0);

  if (remoteTime > localTime) {
    merged[key] = remoteValue;
    changes.push({
      field: key,
      action: 'updated',
      from: localValue,
      to: remoteValue,
      reason: 'remote_newer'
    });
  }
}
```

});

// Merge metadata
merged.metadata = mergeMetadata(localData, remoteData).metadata;

return {
resolved: true,
strategy: CONFLICT_STRATEGIES.FIELD_LEVEL,
result: merged,
conflictData: {
changes: changes,
fieldsChanged: changes.length
}
};
}

/**
~~~

```
- Prepare data for manual resolution
  */
  function prepareManualResolution(localData, remoteData, conflictType) {
  return {
  resolved: false,
  strategy: CONFLICT_STRATEGIES.MANUAL,
  conflictType: conflictType,
  options: {
  local: {
  data: localData,
  label: ‘Keep Local Version’,
  device: localData.metadata?.lastModifiedByDevice,
  modifiedAt: localData.metadata?.modifiedAt,
  version: localData.metadata?.version
  },
  remote: {
  data: remoteData,
  label: ‘Use Remote Version’,
  device: remoteData.metadata?.lastModifiedByDevice,
  modifiedAt: remoteData.metadata?.modifiedAt,
  version: remoteData.metadata?.version
  },
  merge: {
  data: fieldLevelResolution(localData, remoteData).result,
  label: ‘Merge Both Versions’
  }
  },
  differences: generateDifferenceReport(localData, remoteData)
  };
  }

/**

- Generate detailed difference report
  */
  function generateDifferenceReport(localData, remoteData) {
  const differences = [];
  const allKeys = new Set([…Object.keys(localData), …Object.keys(remoteData)]);

allKeys.forEach(key => {
if (key === ‘metadata’) return; // Skip metadata for now

```

~~~~
const localValue = localData[key];
const remoteValue = remoteData[key];

if (JSON.stringify(localValue) !== JSON.stringify(remoteValue)) {
  differences.push({
    field: key,
    local: localValue,
    remote: remoteValue,
    type: getFieldType(localValue, remoteValue)
  });
}
```

});

return differences;
}

/**

- Determine field difference type
  */
  function getFieldType(localValue, remoteValue) {
  if (localValue === undefined) return ‘added’;
  if (remoteValue === undefined) return ‘removed’;
  if (typeof localValue !== typeof remoteValue) return ‘type_changed’;
  if (Array.isArray(localValue) && Array.isArray(remoteValue)) return ‘array_modified’;
  if (typeof localValue === ‘object’ && typeof remoteValue === ‘object’) return ‘object_modified’;
  return ‘value_changed’;
  }

/**

- Merge metadata from winner and loser
  */
  function mergeMetadata(winner, loser) {
  const merged = JSON.parse(JSON.stringify(winner)); // Deep clone
  const now = new Date().toISOString();
  const deviceId = getCurrentDeviceId();

// Update metadata for the merge
merged.metadata = {
…winner.metadata,
modifiedAt: now,
~~~~

~~~~
lastSyncedAt: now,
lastModifiedByDevice: deviceId,
version: Math.max(
winner.metadata?.version || 0,
loser.metadata?.version || 0
) + 1,
syncStatus: ‘synced’,
isDirty: false,
conflictResolution: winner.metadata?.conflictResolution || ‘auto’,

```
// Merge sync relationships
syncedWith: [
  ...new Set([
    ...(winner.metadata?.syncedWith || []),
    ...(loser.metadata?.syncedWith || []),
    loser.metadata?.lastModifiedByDevice
  ].filter(Boolean))
]
```

};

return merged;
}

/**

- Exclude metadata for content comparison
  */
  function excludeMetadata(data) {
  const { metadata, …content } = data;
  return content;
  }

/**

- Apply manual resolution choice
  */
  function applyManualResolution(conflictData, choice) {
  if (!conflictData.options[choice]) {
  throw new Error(`Invalid resolution choice: ${choice}`);
  }
~~~~

```
const resolved = conflictData.options[choice].data;
const now = new Date().toISOString();
const deviceId = getCurrentDeviceId();

// Update metadata to reflect manual resolution
resolved.metadata = {
…resolved.metadata,
modifiedAt: now,
lastSyncedAt: now,
lastModifiedByDevice: deviceId,
version: (resolved.metadata?.version || 0) + 1,
syncStatus: ‘synced’,
isDirty: false,
conflictResolution: ‘manual’
};

return {
resolved: true,
strategy: CONFLICT_STRATEGIES.MANUAL,
result: resolved,
conflictData: {
choice: choice,
resolvedAt: now,
resolvedBy: deviceId
}
};
}

/**

- Conflict resolution UI helper
  */
  function showConflictResolutionDialog(conflictData) {
  return new Promise((resolve) => {
  // This would typically show a UI dialog
  // For now, return a mock resolution
  console.log(‘🔀 Manual conflict resolution required:’, conflictData);

  // Auto-resolve for demo (in real app, this would be user choice)
  const choice = ‘local’; // User would select: ‘local’, ‘remote’, or ‘merge’
  resolve(applyManualResolution(conflictData, choice));
  });
  }
```

```
// Utility function to get device ID (shared with migration)
function getCurrentDeviceId() {
let deviceId = localStorage.getItem('deviceId');
if (!deviceId) {
deviceId = `device-${crypto.randomUUID?.() || Math.random().toString(36)}`;
localStorage.setItem('deviceId', deviceId);
}
return deviceId;
}

// Export functions
if (typeof module !== 'undefined' && module.exports) {
module.exports = {
resolveConflict,
detectConflictType,
autoResolveConflict,
newestWinsResolution,
devicePriorityResolution,
fieldLevelResolution,
prepareManualResolution,
applyManualResolution,
showConflictResolutionDialog,
generateDifferenceReport,
mergeMetadata,
CONFLICT_STRATEGIES,
CONFLICT_TYPES
};
}

/**

- Example Usage:
-
- // Basic conflict resolution
- const result = resolveConflict(localTask, remoteTask,
CONFLICT_STRATEGIES.NEWEST_WINS);
-
- if (result.resolved) {
- // Conflict automatically resolved
- updateLocalData(result.result);
- } else {
- // Manual resolution required
- const manualResult = await showConflictResolutionDialog(result);
- updateLocalData(manualResult.result);
```

- }
-
- // Task-specific conflict resolution
- function resolveTaskConflict(localTask, remoteTask) {
- // Custom logic for task conflicts
- if (localTask.completed !== remoteTask.completed) {

-
  ```
  // Completed tasks take precedence
  ```

-
  ```
  return localTask.completed ? localTask : remoteTask;
  ```
- }
-
- // Fall back to standard resolution
- return resolveConflict(localTask, remoteTask, CONFLICT_STRATEGIES.FIELD_LEVEL);
- }
-
- // Cycle-specific conflict resolution
- function resolveCycleConflict(localCycle, remoteCycle) {
- // Merge tasks at individual level
- const mergedTasks = mergeTasks(localCycle.tasks, remoteCycle.tasks);
-
- // Use newest wins for cycle-level properties
- const cycleResult = resolveConflict(localCycle, remoteCycle,
CONFLICT_STRATEGIES.NEWEST_WINS);
-
- if (cycleResult.resolved) {

-
  ```
  cycleResult.result.tasks = mergedTasks;
  ```

-
  ```
  return cycleResult;
  ```
- }
-
- return cycleResult;
- }
-
- function mergeTasks(localTasks, remoteTasks) {
- const mergedTasks = [];
- const taskMap = new Map();
-
- // Add all local tasks

~~~
- localTasks.forEach(task => taskMap.set(task.id, { local: task }));
-
- // Process remote tasks
- remoteTasks.forEach(remoteTask => {
- ```
  const existing = taskMap.get(remoteTask.id);
  ```
- ```
  if (existing) {
  ```
- ```
    // Conflict - resolve at task level
  ```
- ```
    const resolution = resolveConflict(existing.local, remoteTask);
  ```
- ```
    existing.resolved = resolution.resolved ? resolution.result : existing.local;
  ```
- ```
  } else {
  ```
- ```
    // New remote task
  ```
- ```
    taskMap.set(remoteTask.id, { resolved: remoteTask });
  ```
- ```
  }
  ```
- });
-
- // Build final task array
- taskMap.forEach(entry => {
- ```
  if (entry.resolved) {
  ```
- ```
    mergedTasks.push(entry.resolved);
  ```
- ```
  }
  ```
~~~

```
- });
-
- return mergedTasks;
- }
  */

/**

- Validation & Utility Functions for Schema v3
- Comprehensive validation and helper functions
  */

// Validation error types
const VALIDATION_ERRORS = {
REQUIRED_FIELD: ‘required_field’,
INVALID_TYPE: ‘invalid_type’,
INVALID_FORMAT: ‘invalid_format’,
INVALID_VALUE: ‘invalid_value’,
INVALID_LENGTH: ‘invalid_length’,
INVALID_RANGE: ‘invalid_range’,
INVALID_ENUM: ‘invalid_enum’,
INVALID_UUID: ‘invalid_uuid’,
INVALID_DATE: ‘invalid_date’,
INVALID_SCHEMA: ‘invalid_schema’
};

// Valid enum values
const VALID_ENUMS = {
deviceTypes: [‘mobile’, ‘desktop’, ‘tablet’, ‘web’],
platforms: [‘ios’, ‘android’, ‘windows’, ‘macos’, ‘linux’, ‘web’],
frequencies: [‘hourly’, ‘daily’, ‘weekly’, ‘biweekly’, ‘monthly’, ‘yearly’],
frequencyUnits: [‘minutes’, ‘hours’, ‘days’],
endConditionTypes: [‘never’, ‘count’, ‘date’],
syncStatuses: [‘synced’, ‘pending’, ‘conflict’, ‘error’],
conflictResolutions: [‘auto’, ‘manual’, ‘newest-wins’, ‘device-priority’],
weekdays: [‘Sunday’, ‘Monday’, ‘Tuesday’, ‘Wednesday’, ‘Thursday’, ‘Friday’, ‘Saturday’],
timeFormats: [‘12’, ‘24’],
themes: [‘default’, ‘dark-ocean’, ‘golden-glow’],
taskDisplayModes: [‘list’, ‘grid’, ‘compact’],
sortOrders: [‘manual’, ‘priority’, ‘dueDate’, ‘alphabetical’]
};

/**
```

```
- Validate Task Schema v3
  */
  function validateTaskV3(task, context = {}) {
  const errors = [];

// Required fields validation
const requiredFields = [‘id’, ‘text’, ‘completed’, ‘metadata’, ‘schemaVersion’];
requiredFields.forEach(field => {
if (!(field in task)) {
errors.push({
type: VALIDATION_ERRORS.REQUIRED_FIELD,
field: field,
message: `Required field '${field}' is missing`
});
}
});

// Schema version validation
if (task.schemaVersion !== 3) {
errors.push({
type: VALIDATION_ERRORS.INVALID_VALUE,
field: ‘schemaVersion’,
message: `Schema version must be 3, got ${task.schemaVersion}`
});
}

// ID validation (must be UUID v4)
if (task.id && !isValidUUID(task.id)) {
errors.push({
type: VALIDATION_ERRORS.INVALID_UUID,
field: ‘id’,
message: ‘Task ID must be a valid UUID v4’
});
}

// Text validation
if (task.text !== undefined) {
if (typeof task.text !== ‘string’) {
errors.push({
type: VALIDATION_ERRORS.INVALID_TYPE,
field: ‘text’,
message: ‘Task text must be a string’
});
} else if (task.text.length === 0) {
```

```
errors.push({
type: VALIDATION_ERRORS.INVALID_LENGTH,
field: ‘text’,
message: ‘Task text cannot be empty’
});
} else if (task.text.length > 500) {
errors.push({
type: VALIDATION_ERRORS.INVALID_LENGTH,
field: ‘text’,
message: ‘Task text cannot exceed 500 characters’
});
}
}

// Boolean field validations
[‘completed’, ‘highPriority’, ‘remindersEnabled’, ‘recurring’].forEach(field => {
if (task[field] !== undefined && typeof task[field] !== ‘boolean’) {
errors.push({
type: VALIDATION_ERRORS.INVALID_TYPE,
field: field,
message: `${field} must be a boolean`
});
}
});

// Date validations
if (task.dueDate !== undefined && task.dueDate !== null) {
if (!isValidISODate(task.dueDate)) {
errors.push({
type: VALIDATION_ERRORS.INVALID_DATE,
field: ‘dueDate’,
message: ‘Due date must be a valid ISO 8601 date string’
});
}
}

// Reminder settings validation
if (task.reminderSettings) {
errors.push(…validateReminderSettings(task.reminderSettings, ‘reminderSettings’));
}

// Recurring settings validation
if (task.recurring && task.recurringSettings) {
errors.push(…validateRecurringSettings(task.recurringSettings, ‘recurringSettings’));
```

```
}

// Metadata validation
if (task.metadata) {
errors.push(…validateMetadata(task.metadata, ‘metadata’));
}

return {
valid: errors.length === 0,
errors: errors
};
}

/**

- Validate Mini Cycle Schema v3
  */
  function validateMiniCycleV3(cycle, context = {}) {
  const errors = [];

// Required fields validation
const requiredFields = [‘id’, ‘name’, ‘title’, ‘tasks’, ‘metadata’, ‘schemaVersion’];
requiredFields.forEach(field => {
if (!(field in cycle)) {
errors.push({
type: VALIDATION_ERRORS.REQUIRED_FIELD,
field: field,
message: `Required field '${field}' is missing`
});
}
});

// Schema version validation
if (cycle.schemaVersion !== 3) {
errors.push({
type: VALIDATION_ERRORS.INVALID_VALUE,
field: ‘schemaVersion’,
message: `Schema version must be 3, got ${cycle.schemaVersion}`
});
}

// ID validation
if (cycle.id && !isValidUUID(cycle.id)) {
errors.push({
```

```
type: VALIDATION_ERRORS.INVALID_UUID,
field: ‘id’,
message: ‘Cycle ID must be a valid UUID v4’
});
}

// Name validation (URL-safe)
if (cycle.name !== undefined) {
if (typeof cycle.name !== ‘string’) {
errors.push({
type: VALIDATION_ERRORS.INVALID_TYPE,
field: ‘name’,
message: ‘Cycle name must be a string’
});
} else if (!/^[a-zA-Z0-9-_]+$/.test(cycle.name)) {
errors.push({
type: VALIDATION_ERRORS.INVALID_FORMAT,
field: ‘name’,
message: ‘Cycle name must be URL-safe (alphanumeric, hyphens, underscores only)’
});
}
}

// Title validation
if (cycle.title !== undefined) {
if (typeof cycle.title !== ‘string’) {
errors.push({
type: VALIDATION_ERRORS.INVALID_TYPE,
field: ‘title’,
message: ‘Cycle title must be a string’
});
} else if (cycle.title.length > 200) {
errors.push({
type: VALIDATION_ERRORS.INVALID_LENGTH,
field: ‘title’,
message: ‘Cycle title cannot exceed 200 characters’
});
}
}

// Tasks validation
if (cycle.tasks !== undefined) {
if (!Array.isArray(cycle.tasks)) {
errors.push({
```

```
type: VALIDATION_ERRORS.INVALID_TYPE,
field: ‘tasks’,
message: ‘Tasks must be an array’
});
} else {
cycle.tasks.forEach((task, index) => {
const taskValidation = validateTaskV3(task, { parentCycle: cycle });
if (!taskValidation.valid) {
taskValidation.errors.forEach(error => {
errors.push({
…error,
field: `tasks[${index}].${error.field}`,
message: `Task ${index}: ${error.message}`
});
});
}
});
}
}

// Numeric validations
[‘cycleCount’, ‘totalTasksCompleted’, ‘averageCompletionTime’].forEach(field => {
if (cycle[field] !== undefined) {
if (typeof cycle[field] !== ‘number’ || cycle[field] < 0) {
errors.push({
type: VALIDATION_ERRORS.INVALID_VALUE,
field: field,
message: `${field} must be a non-negative number`
});
}
}
});

// Boolean validations
[‘autoReset’, ‘deleteCheckedTasks’].forEach(field => {
if (cycle[field] !== undefined && typeof cycle[field] !== ‘boolean’) {
errors.push({
type: VALIDATION_ERRORS.INVALID_TYPE,
field: field,
message: `${field} must be a boolean`
});
}
});
```

```
// Color validation (hex format)
if (cycle.color !== undefined) {
if (!/^#[0-9A-Fa-f]{6}$/.test(cycle.color)) {
errors.push({
type: VALIDATION_ERRORS.INVALID_FORMAT,
field: ‘color’,
message: ‘Color must be a valid hex color code (e.g., #4A90E2)’
});
}
}

// Preferences validation
if (cycle.preferences) {
errors.push(…validateCyclePreferences(cycle.preferences, ‘preferences’));
}

// Schedule validation
if (cycle.schedule) {
errors.push(…validateSchedule(cycle.schedule, ‘schedule’));
}

// Metadata validation
if (cycle.metadata) {
errors.push(…validateMetadata(cycle.metadata, ‘metadata’));
}

return {
valid: errors.length === 0,
errors: errors
};
}

/**

- Validate Device Schema v3
  */
  function validateDeviceV3(device, context = {}) {
  const errors = [];

// Required fields validation
const requiredFields = [‘deviceID’, ‘deviceName’, ‘deviceType’, ‘platform’, ‘metadata’,
‘schemaVersion’];
requiredFields.forEach(field => {
if (!(field in device)) {
```

```
errors.push({
type: VALIDATION_ERRORS.REQUIRED_FIELD,
field: field,
message: `Required field '${field}' is missing`
});
}
});

// Schema version validation
if (device.schemaVersion !== 3) {
errors.push({
type: VALIDATION_ERRORS.INVALID_VALUE,
field: ‘schemaVersion’,
message: `Schema version must be 3, got ${device.schemaVersion}`
});
}

// Device ID validation
if (device.deviceID && !device.deviceID.startsWith(‘device-’)) {
errors.push({
type: VALIDATION_ERRORS.INVALID_FORMAT,
field: ‘deviceID’,
message: ‘Device ID must start with “device-”’
});
}

// Enum validations
if (device.deviceType && !VALID_ENUMS.deviceTypes.includes(device.deviceType)) {
errors.push({
type: VALIDATION_ERRORS.INVALID_ENUM,
field: ‘deviceType’,
message: `Device type must be one of: ${VALID_ENUMS.deviceTypes.join(', ')}`
});
}

if (device.platform && !VALID_ENUMS.platforms.includes(device.platform)) {
errors.push({
type: VALIDATION_ERRORS.INVALID_ENUM,
field: ‘platform’,
message: `Platform must be one of: ${VALID_ENUMS.platforms.join(', ')}`
});
}

// Metadata validation
```

```
if (device.metadata) {
errors.push(…validateMetadata(device.metadata, ‘metadata’));
}

return {
valid: errors.length === 0,
errors: errors
};
}

/**

- Validate Reminder Settings
  */
  function validateReminderSettings(settings, prefix = ‘’) {
  const errors = [];

if (typeof settings !== ‘object’ || settings === null) {
errors.push({
type: VALIDATION_ERRORS.INVALID_TYPE,
field: prefix,
message: ‘Reminder settings must be an object’
});
return errors;
}

// Boolean validations
if (settings.enabled !== undefined && typeof settings.enabled !== ‘boolean’) {
errors.push({
type: VALIDATION_ERRORS.INVALID_TYPE,
field: `${prefix}.enabled`,
message: ‘Enabled must be a boolean’
});
}

// Date validations
[‘startTime’, ‘lastFired’].forEach(field => {
if (settings[field] !== undefined && settings[field] !== null && !isValidISODate(settings[field])) {
errors.push({
type: VALIDATION_ERRORS.INVALID_DATE,
field: `${prefix}.${field}`,
message: `${field} must be a valid ISO 8601 date string`
});
}
```

```
});

// Frequency validation
if (settings.frequencyValue !== undefined) {
if (typeof settings.frequencyValue !== ‘number’ || settings.frequencyValue <= 0) {
errors.push({
type: VALIDATION_ERRORS.INVALID_VALUE,
field: `${prefix}.frequencyValue`,
message: ‘Frequency value must be a positive number’
});
}
}

if (settings.frequencyUnit && !VALID_ENUMS.frequencyUnits.includes(settings.frequencyUnit)) {
errors.push({
type: VALIDATION_ERRORS.INVALID_ENUM,
field: `${prefix}.frequencyUnit`,
message: `Frequency unit must be one of: ${VALID_ENUMS.frequencyUnits.join(', ')}`
});
}

return errors;
}

/**

- Validate Recurring Settings
  */
  function validateRecurringSettings(settings, prefix = ‘’) {
  const errors = [];

if (typeof settings !== ‘object’ || settings === null) {
errors.push({
type: VALIDATION_ERRORS.INVALID_TYPE,
field: prefix,
message: ‘Recurring settings must be an object’
});
return errors;
}

// Frequency validation
if (settings.frequency && !VALID_ENUMS.frequencies.includes(settings.frequency)) {
errors.push({
type: VALIDATION_ERRORS.INVALID_ENUM,
field: `${prefix}.frequency`,
message: `Frequency must be one of: ${VALID_ENUMS.frequencies.join(', ')}`
});
}

// Interval validation
if (settings.interval !== undefined) {
if (typeof settings.interval !== ‘number’ || settings.interval < 1) {
errors.push({
type: VALIDATION_ERRORS.INVALID_VALUE,
field: `${prefix}.interval`,
message: ‘Interval must be a positive integer’
});
}
}

// Time format validation
if (settings.timeOfDay && !/^\d{2}:\d{2}:\d{2}$/.test(settings.timeOfDay)) {
errors.push({
type: VALIDATION_ERRORS.INVALID_FORMAT,
field: `${prefix}.timeOfDay`,
message: ‘Time of day must be in HH:MM:SS format’
});
}

// Date validations
[‘startDate’, ‘defaultRecurTime’, ‘lastTriggered’].forEach(field => {
if (settings[field] !== undefined && settings[field] !== null && !isValidISODate(settings[field])) {
errors.push({
type: VALIDATION_ERRORS.INVALID_DATE,
field: `${prefix}.${field}`,
message: `${field} must be a valid ISO 8601 date string`
});
}
});

// End condition validation
if (settings.endCondition) {
if (settings.endCondition.type &&
!VALID_ENUMS.endConditionTypes.includes(settings.endCondition.type)) {
errors.push({
type: VALIDATION_ERRORS.INVALID_ENUM,
field: `${prefix}.endCondition.type`,
message: `End condition type must be one of: ${VALID_ENUMS.endConditionTypes.join(', ')}`
```

```
});
}
}

// Weekly days validation
if (settings.weekly && settings.weekly.days) {
settings.weekly.days.forEach((day, index) => {
if (!VALID_ENUMS.weekdays.includes(day)) {
errors.push({
type: VALIDATION_ERRORS.INVALID_ENUM,
field: `${prefix}.weekly.days[${index}]`,
message: `Day must be one of: ${VALID_ENUMS.weekdays.join(', ')}`
});
}
});
}

return errors;
}

/**

- Validate Metadata
  */
  function validateMetadata(metadata, prefix = ‘’) {
  const errors = [];

if (typeof metadata !== ‘object’ || metadata === null) {
errors.push({
type: VALIDATION_ERRORS.INVALID_TYPE,
field: prefix,
message: ‘Metadata must be an object’
});
return errors;
}

// Date validations
[‘createdAt’, ‘modifiedAt’, ‘lastSyncedAt’, ‘completedAt’, ‘lastAccessedAt’].forEach(field => {
if (metadata[field] !== undefined && metadata[field] !== null && !isValidISODate(metadata[field]))
{
errors.push({
type: VALIDATION_ERRORS.INVALID_DATE,
field: `${prefix}.${field}`,
message: `${field} must be a valid ISO 8601 date string`
});
}
});

// Version validation
if (metadata.version !== undefined) {
if (typeof metadata.version !== ‘number’ || metadata.version < 0) {
errors.push({
type: VALIDATION_ERRORS.INVALID_VALUE,
field: `${prefix}.version`,
message: ‘Version must be a non-negative number’
});
}
}

// Sync status validation
if (metadata.syncStatus && !VALID_ENUMS.syncStatuses.includes(metadata.syncStatus)) {
errors.push({
type: VALIDATION_ERRORS.INVALID_ENUM,
field: `${prefix}.syncStatus`,
message: `Sync status must be one of: ${VALID_ENUMS.syncStatuses.join(', ')}`
});
}

// Conflict resolution validation
if (metadata.conflictResolution &&
!VALID_ENUMS.conflictResolutions.includes(metadata.conflictResolution)) {
errors.push({
type: VALIDATION_ERRORS.INVALID_ENUM,
field: `${prefix}.conflictResolution`,
message: `Conflict resolution must be one of: ${VALID_ENUMS.conflictResolutions.join(', ')}`
});
}

return errors;
}

/**

- Validate Cycle Preferences
  */
  function validateCyclePreferences(preferences, prefix = ‘’) {
  const errors = [];
```

```
if (preferences.theme && !VALID_ENUMS.themes.includes(preferences.theme)) {
errors.push({
type: VALIDATION_ERRORS.INVALID_ENUM,
field: `${prefix}.theme`,
message: `Theme must be one of: ${VALID_ENUMS.themes.join(', ')}`
});
}

if (preferences.taskDisplayMode &&
!VALID_ENUMS.taskDisplayModes.includes(preferences.taskDisplayMode)) {
errors.push({
type: VALIDATION_ERRORS.INVALID_ENUM,
field: `${prefix}.taskDisplayMode`,
message: `Task display mode must be one of: ${VALID_ENUMS.taskDisplayModes.join(', ')}`
});
}

if (preferences.sortOrder && !VALID_ENUMS.sortOrders.includes(preferences.sortOrder)) {
errors.push({
type: VALIDATION_ERRORS.INVALID_ENUM,
field: `${prefix}.sortOrder`,
message: `Sort order must be one of: ${VALID_ENUMS.sortOrders.join(', ')}`
});
}

return errors;
}

/**

- Validate Schedule
  */
  function validateSchedule(schedule, prefix = '') {
  const errors = [];

if (schedule.time && !/^\d{2}:\d{2}:\d{2}$/.test(schedule.time)) {
errors.push({
type: VALIDATION_ERRORS.INVALID_FORMAT,
field: `${prefix}.time`,
message: 'Schedule time must be in HH:MM:SS format'
});
}

if (schedule.daysOfWeek && Array.isArray(schedule.daysOfWeek)) {
```

```
schedule.daysOfWeek.forEach((day, index) => {
if (!VALID_ENUMS.weekdays.includes(day)) {
errors.push({
type: VALIDATION_ERRORS.INVALID_ENUM,
field: `${prefix}.daysOfWeek[${index}]`,
message: `Day must be one of: ${VALID_ENUMS.weekdays.join(', ')}`
});
}
});
}

return errors;
}

/**

- Utility Functions
  */

function isValidUUID(uuid) {
const uuidRegex = /^[0-9a-f]{8}-[0-9a-f]{4}-4[0-9a-f]{3}-[89ab][0-9a-f]{3}-[0-9a-f]{12}$/i;
return uuidRegex.test(uuid);
}

function isValidISODate(dateString) {
if (typeof dateString !== ‘string’) return false;

try {
const date = new Date(dateString);
return date.toISOString() === dateString;
} catch {
return false;
}
}

function sanitizeTaskText(text) {
if (typeof text !== ‘string’) return ‘’;

// Remove potentially dangerous characters
const sanitized = text
.replace(/<script\b[^<]*(?:(?!<\/script>)<[^<]*)*<\/script>/gi, ‘’) // Remove script tags
.replace(/[<>]/g, ‘’) // Remove angle brackets
.trim();
```

```
return sanitized.substring(0, 500); // Limit length
}

function generateValidationReport(validationResult) {
if (validationResult.valid) {
return {
status: ‘VALID’,
summary: ‘All validations passed’,
errors: []
};
}

const groupedErrors = validationResult.errors.reduce((groups, error) => {
const category = error.type;
if (!groups[category]) groups[category] = [];
groups[category].push(error);
return groups;
}, {});

return {
status: ‘INVALID’,
summary: `${validationResult.errors.length} validation error(s) found`,
errorCount: validationResult.errors.length,
errorsByType: groupedErrors,
errors: validationResult.errors
};
}

// Export functions
if (typeof module !== ‘undefined’ && module.exports) {
module.exports = {
validateTaskV3,
validateMiniCycleV3,
validateDeviceV3,
validateReminderSettings,
validateRecurringSettings,
validateMetadata,
isValidUUID,
isValidISODate,
sanitizeTaskText,
generateValidationReport,
VALIDATION_ERRORS,
VALID_ENUMS
};
```

```
}
```