Enhanced ODK MCP System: Complete API Documentation

Version: 2.0.0
Date: June 2025
Author: Manus AI
Document Type: Comprehensive API Reference

Table of Contents

1. Introduction
2. Authentication and Authorization
3. API Architecture
4. Form Management API
5. Data Collection API
6. Data Aggregation API
7. Analytics API
8. AI Services API
9. User Management API
10. Webhook API
11. Error Handling
12. Rate Limiting
13. SDK and Libraries
14. Examples and Tutorials

Introduction

The Enhanced ODK MCP System provides a comprehensive RESTful API that enables developers to integrate data collection and analysis capabilities into their applications. This API documentation serves as a complete reference for all available endpoints, request/response formats, authentication methods, and integration patterns.

API Overview and Philosophy

The Enhanced ODK MCP System API is designed following REST architectural principles, providing a consistent and intuitive interface for all system operations. The API

emphasizes simplicity, reliability, and performance while maintaining the flexibility needed to support diverse integration scenarios.

The API architecture follows the microservices pattern, with each major system component exposing its own set of endpoints. This design provides several advantages including independent scaling of services, isolated failure domains, and the ability to evolve different parts of the system independently. All services communicate through well-defined interfaces and maintain backward compatibility to ensure stable integrations.

Resource-oriented design principles guide the API structure, with each endpoint representing a specific resource or collection of resources. Standard HTTP methods (GET, POST, PUT, DELETE) are used consistently across all endpoints, and HTTP status codes provide clear indication of operation results. This approach makes the API intuitive for developers familiar with REST conventions.

The API supports both synchronous and asynchronous operations depending on the nature of the request. Simple operations like retrieving form definitions or submitting data are handled synchronously, while complex operations like data analysis or report generation are handled asynchronously with status polling or webhook notifications.

Versioning and Compatibility

API versioning ensures that existing integrations continue to function as the system evolves. The Enhanced ODK MCP System uses semantic versioning for the API, with version numbers following the format major.minor.patch. Major version changes indicate breaking changes that may require integration updates, minor version changes add new functionality while maintaining backward compatibility, and patch versions include bug fixes and minor improvements.

Version information is included in the API URL path (e.g., /api/v2/forms) and in response headers. Clients should always specify the API version they are designed to work with to ensure consistent behavior. The system maintains support for previous major versions for a defined period to allow for gradual migration.

Deprecation notices are provided well in advance of any breaking changes, with clear migration guidance and timelines. The API documentation includes version-specific information and migration guides to help developers update their integrations when necessary.

Base URL and Endpoints

All API endpoints are accessed through a base URL that varies depending on your deployment configuration. For cloud-hosted instances, the base URL typically follows the pattern https://your-organization.odk-mcp.com/api/v2/. For self-hosted deployments, the base URL will be determined by your organization's domain and configuration.

The API is organized into logical groups based on functionality, with each group having its own base path. Form management operations are accessed through /api/v2/forms/, data collection operations through /api/v2/submissions/, analytics operations through /api/v2/analytics/, and so on. This organization makes it easy to understand the API structure and locate relevant endpoints.

All endpoints support HTTPS encryption, and HTTP requests are automatically redirected to HTTPS for security. The API also supports HTTP/2 for improved performance and efficiency, particularly for applications that make multiple concurrent requests.

Authentication and Authorization

The Enhanced ODK MCP System implements a comprehensive authentication and authorization framework that supports multiple authentication methods and fine-grained access control. This section provides detailed information about authentication mechanisms, token management, and authorization patterns.

Authentication Methods

The API supports several authentication methods to accommodate different integration scenarios and security requirements. The primary authentication method is OAuth 2.0 with JWT tokens, which provides secure and scalable authentication for both user-based and service-based integrations.

OAuth 2.0 authentication follows the standard authorization code flow for user-based applications and the client credentials flow for service-to-service integrations. User-based authentication requires users to log in through the web interface and grant permission for the application to access their data. Service-based authentication uses client credentials (client ID and secret) to obtain access tokens for automated operations.

API key authentication is available for simple integrations and legacy systems. API keys are long-lived credentials that can be used for direct API access without the OAuth flow.

However, API keys have limited scope and are recommended only for specific use cases where OAuth is not feasible.

Multi-factor authentication (MFA) can be enforced for API access when enabled at the organizational level. MFA requirements apply to both OAuth and API key authentication and may require additional verification steps during the authentication process.

Token Management

JWT (JSON Web Token) access tokens are used for API authentication and contain encoded information about the user or service, their permissions, and token expiration. Access tokens have a limited lifetime (typically 1 hour) and must be refreshed using refresh tokens for continued access.

Token refresh is handled automatically by most OAuth libraries, but applications should implement proper token refresh logic to handle expired tokens gracefully. The API returns specific error codes for expired tokens, allowing applications to trigger the refresh process automatically.

Token revocation is supported for both access tokens and refresh tokens, allowing users and administrators to immediately revoke access when necessary. Revoked tokens are maintained in a blacklist to prevent their reuse, and the revocation status is checked on each API request.

Token introspection endpoints allow applications to validate tokens and retrieve information about their scope and expiration. This feature is useful for applications that need to make authorization decisions based on token contents or for debugging authentication issues.

Authorization and Permissions

The API implements role-based access control (RBAC) with fine-grained permissions that determine what operations users can perform on specific resources. Permissions are organized hierarchically, with higher-level permissions implying lower-level permissions.

Resource-level permissions control access to specific forms, projects, or datasets. Users can have different permission levels for different resources, allowing for flexible access control that matches organizational structures and workflows. Common permission levels include read-only access, data collection access, data analysis access, and administrative access.

Operation-level permissions control what actions users can perform on resources they have access to. For example, a user might have permission to view form data but not to

modify or delete it. Operation permissions are checked on each API request to ensure that users can only perform authorized actions.

Dynamic permissions can be configured based on data attributes such as geographic location, time periods, or data sensitivity levels. This feature allows organizations to implement complex access control policies that adapt to changing requirements and contexts.

Permission inheritance allows permissions to be granted at higher levels (such as organizations or projects) and automatically applied to lower levels (such as forms or submissions). This approach simplifies permission management while maintaining security and control.

Security Best Practices

API security is implemented through multiple layers of protection including encryption, authentication, authorization, and monitoring. All API communications are encrypted using TLS 1.3 or higher, and sensitive data is encrypted at rest using industry-standard encryption algorithms.

Rate limiting protects the API from abuse and ensures fair resource allocation among users. Rate limits are applied per user, per API key, and per IP address, with different limits for different types of operations. Exceeded rate limits result in HTTP 429 responses with information about when requests can be resumed.

Request validation ensures that all API requests conform to expected formats and contain valid data. Invalid requests are rejected with detailed error messages that help developers identify and fix issues without exposing sensitive system information.

Audit logging records all API access and operations for security monitoring and compliance purposes. Audit logs include information about the user, operation, timestamp, and result, allowing administrators to track system usage and identify potential security issues.

IP whitelisting can be configured to restrict API access to specific IP addresses or ranges. This feature is particularly useful for service-to-service integrations where the client IP address is known and stable.

API Architecture

The Enhanced ODK MCP System API is built on a modern microservices architecture that provides scalability, reliability, and maintainability. Understanding the architectural

principles and design patterns will help developers create more effective integrations and troubleshoot issues when they arise.

Microservices Design

The API is composed of several independent microservices, each responsible for a specific domain of functionality. This design allows each service to be developed, deployed, and scaled independently while maintaining clear boundaries and responsibilities.

The Form Management Service handles all operations related to form creation, modification, versioning, and deployment. This service exposes endpoints for form CRUD operations, template management, and form distribution. The service maintains its own database and business logic, ensuring that form-related operations are isolated from other system components.

The Data Collection Service manages the data collection process including form rendering, data validation, submission handling, and synchronization. This service is optimized for high-volume operations and includes sophisticated caching and queuing mechanisms to handle peak loads during data collection campaigns.

The Data Aggregation Service processes and analyzes collected data, providing endpoints for data retrieval, aggregation, and export. This service includes advanced query capabilities and supports real-time and batch processing modes depending on the operation requirements.

The Analytics Service provides statistical analysis and reporting capabilities through a set of specialized endpoints. This service integrates with machine learning models and external analytics tools to provide advanced insights and predictive capabilities.

The User Management Service handles authentication, authorization, and user profile management. This service implements OAuth 2.0 and other authentication protocols and maintains user permissions and access control policies.

Service Communication

Inter-service communication follows established patterns that ensure reliability and performance. Services communicate through well-defined APIs using HTTP/REST for synchronous operations and message queues for asynchronous operations.

Synchronous communication is used for operations that require immediate responses, such as form retrieval or data validation. These operations use standard HTTP requests with appropriate timeout and retry policies to handle temporary failures.

Asynchronous communication is used for long-running operations such as data processing or report generation. These operations use message queues to decouple services and provide reliable delivery guarantees. Status updates and completion notifications are provided through webhooks or polling endpoints.

Service discovery mechanisms allow services to locate and communicate with each other dynamically. This approach provides flexibility for deployment and scaling while maintaining loose coupling between services.

Circuit breaker patterns protect services from cascading failures by automatically detecting and isolating failing dependencies. When a service becomes unavailable, circuit breakers prevent additional requests from being sent and provide fallback responses when possible.

Data Consistency and Transactions

Data consistency across microservices is managed through a combination of techniques including eventual consistency, saga patterns, and distributed transactions where necessary. The system is designed to handle temporary inconsistencies gracefully while ensuring that critical business rules are maintained.

Eventual consistency is used for operations where immediate consistency is not required, such as analytics data or reporting metrics. These operations can tolerate slight delays in data propagation in exchange for better performance and availability.

Saga patterns are used for complex operations that span multiple services, such as form deployment or user provisioning. Sagas ensure that either all operations complete successfully or compensating actions are taken to maintain system consistency.

Distributed transactions are used sparingly for critical operations that require strong consistency guarantees, such as financial transactions or audit logging. These operations use two-phase commit protocols to ensure atomicity across multiple services.

Conflict resolution mechanisms handle situations where concurrent operations might create data conflicts. The system uses optimistic locking, version vectors, and last-writer-wins strategies depending on the specific use case and consistency requirements.

Caching and Performance

The API implements multiple layers of caching to optimize performance and reduce load on backend services. Caching strategies are tailored to the specific characteristics and usage patterns of different types of data.

Application-level caching stores frequently accessed data such as form definitions, user permissions, and configuration settings in memory for fast retrieval. This cache is automatically invalidated when underlying data changes to ensure consistency.

Database query caching reduces the load on database servers by caching the results of expensive queries. Query cache invalidation is managed through database triggers and application logic to ensure that cached results remain accurate.

Content delivery network (CDN) caching is used for static assets such as form templates, images, and documentation. CDN caching provides global distribution and reduces latency for users in different geographic regions.

HTTP caching headers are used to enable client-side caching of API responses where appropriate. Cache headers include expiration times and validation tokens that allow clients to cache responses while ensuring data freshness.

Performance monitoring and optimization tools continuously analyze API performance and identify opportunities for improvement. These tools track response times, error rates, and resource utilization to ensure optimal system performance.

Form Management API

The Form Management API provides comprehensive functionality for creating, managing, and deploying data collection forms. This API is designed to support both simple form operations and complex form management workflows required by large organizations.

Form CRUD Operations

Form creation through the API allows developers to programmatically create new forms using JSON form definitions. The API accepts form definitions in multiple formats including XLSForm JSON, ODK XForm XML, and the system's native JSON format. Form validation is performed automatically to ensure that form definitions are syntactically correct and semantically valid.

The POST /api/v2/forms endpoint creates new forms and returns a unique form identifier that can be used for subsequent operations. The request body should contain the form definition along with metadata such as form title, description, and deployment settings. The API validates the form definition and returns detailed error messages if validation fails.

Form retrieval operations allow developers to access existing form definitions and metadata. The GET /api/v2/forms/{formId} endpoint returns the complete form

definition along with version information, deployment status, and usage statistics. Query parameters can be used to specify the desired format and include or exclude specific sections of the form definition.

Form listing operations provide access to collections of forms with filtering and pagination support. The GET /api/v2/forms endpoint returns a paginated list of forms that the authenticated user has access to. Filters can be applied based on form status, creation date, project assignment, and other criteria.

Form updates are handled through the PUT /api/v2/forms/{formId} endpoint, which replaces the entire form definition with a new version. The API automatically creates a new version number and maintains the previous version for backward compatibility. Partial updates can be performed using the PATCH endpoint for specific form properties.

Form deletion is supported through the DELETE /api/v2/forms/{formId} endpoint, which marks forms as deleted rather than physically removing them from the database. Deleted forms are no longer available for data collection but remain accessible for historical data analysis and audit purposes.

Form Versioning and History

Form versioning is automatically managed by the API to ensure that changes to forms are properly tracked and that existing data collection activities are not disrupted. Each form modification creates a new version with a unique version identifier and timestamp.

Version creation occurs automatically when forms are updated through the API. The system compares the new form definition with the current version and creates a new version only if significant changes are detected. Minor changes such as help text updates may not trigger version creation depending on system configuration.

Version retrieval allows developers to access specific versions of forms for analysis or rollback purposes. The GET /api/v2/forms/{formId}/versions/{versionId} endpoint returns the form definition for a specific version along with metadata about when the version was created and what changes were made.

Version comparison functionality helps developers understand the differences between form versions. The GET /api/v2/forms/{formId}/versions/{versionId}/compare/{otherVersionId} endpoint returns a detailed comparison showing added, modified, and removed form elements.

Version rollback allows administrators to revert forms to previous versions when necessary. The POST /api/v2/forms/{formId}/versions/{versionId}/rollback endpoint

creates a new version based on the specified previous version, effectively undoing recent changes while maintaining version history.

Version deployment controls which version of a form is active for data collection. The POST /api/v2/forms/{formId}/versions/{versionId}/deploy endpoint activates a specific version and updates all deployed instances of the form. Deployment can be scheduled for future execution or performed immediately.

Form Templates and Libraries

Form templates provide a foundation for rapid form development and promote consistency across projects. The API provides comprehensive access to template libraries and supports both system-provided and custom templates.

Template retrieval allows developers to access available form templates through the GET /api/v2/templates endpoint. Templates are organized by category and include metadata such as description, target use case, and complexity level. The response includes template previews and usage statistics to help developers select appropriate templates.

Template instantiation creates new forms based on existing templates through the POST /api/v2/templates/{templateId}/instantiate endpoint. The instantiation process allows for customization of template parameters such as organization branding, language settings, and field modifications. The resulting form can be further customized using standard form editing operations.

Custom template creation allows organizations to create their own templates from existing forms. The POST /api/v2/forms/{formId}/create-template endpoint converts a form into a reusable template that can be shared within the organization or contributed to the community library.

Template sharing and collaboration features enable organizations to share successful form designs with other users. The API supports template publishing, rating, and commenting to facilitate knowledge sharing and continuous improvement of template quality.

Template versioning ensures that improvements to templates are properly managed and distributed. Template updates can be automatically propagated to forms based on those templates, or users can choose to manually update their forms when new template versions become available.