Mahak Accounting System – Sync API Documentation (English)

1. Introduction

The Mahak Accounting System's Sync API is a robust RESTful interface designed to facilitate seamless data integration between the Mahak Accounting System and external client applications. This API enables a variety of operations, including secure authentication, efficient retrieval of data from Mahak, and the submission of bulk data updates back into the Mahak system. The primary objective of the Sync API is to maintain data integrity and consistency across all integrated systems.

The architecture is built to support multiple concurrent clients accessing a single Mahak instance. To optimize data transfer and minimize bandwidth usage, the Sync API employs two key mechanisms:

- RowVersion-based Delta Synchronization: Each record within Mahak is associated with a `rowVersion` number. This is an integer that increments with every modification to a record. By providing the `rowVersion` from the last successful synchronization for a specific entity, clients can request only those records that have changed since that point. This significantly reduces the amount of data transferred compared to fetching all records every time.
- Logical Deletion (**`deleted`** flag): Instead of physically removing records from the database when they are no longer needed, the Sync API utilizes a `deleted` boolean flag. When a record is marked as deleted, it is effectively hidden from most regular operations but remains in the system for audit trails and historical reference. This approach is crucial for maintaining referential integrity, especially when dealing with linked entities.

This document will detail the endpoints, authentication mechanisms, data models, synchronization strategies, error handling, security considerations, and best practices for utilizing the Mahak Sync API.

---

2. Authentication

Secure access to the Mahak Sync API is managed through a token-based authentication system, specifically using JSON Web Tokens (JWT). Clients must first authenticate to obtain a valid JWT, which is then included in subsequent requests to prove their identity and authorization.

2.1 Endpoints for Authentication

- **`POST /Sync/Login`** : An older version of the login endpoint. It is recommended to use `LoginV2` for enhanced features and potentially better security.
- **`POST /Sync/LoginV2`** : The recommended endpoint for obtaining a JWT bearer token.

2.2 Purpose of Authentication

The primary purpose of these endpoints is to securely issue a JWT bearer token to authenticated clients. This token acts as an access credential for all subsequent API calls made to the Sync API. The token is typically valid for a specific duration and must be refreshed once it expires.

2.3 Parameters for Authentication

The `LoginV2` endpoint accepts a `LoginModel` as its request body. The `LoginModel` comprises the following essential parameters:

- **`DatabaseId`** (Required, Integer): This identifier uniquely specifies the Mahak database instance to which the client is attempting to connect. This ID is typically obtained from the Mahak's Bazaar plugin or provided by Mahak Customer Service during setup.

- **`PackageNumber`** (Required, Integer): This number is usually displayed prominently on the main screen of the Mahak software. It helps identify the specific version or deployment of the Mahak application.
- **`Username`** (Required, String): The username provided by Mahak Customer Service for accessing the system.
- **`Password`** (Required, String): The password associated with the provided username.

2.4 Example Request (LoginV2)

```
 POST /API/v3/Sync/LoginV2
{
  "username": "user1",
  "password": "secure_password_here",
  "databaseId": 1001,
  "packageNumber": 2002
}
```

2.5 Example Response

Upon successful authentication, the API will return a JSON object containing the JWT and its expiration date.

```
 {
  "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3OD
  "expireDate": "2025-01-15T12:00:00Z"
}
```

Notes on Token Usage:

- Authorization Header: The obtained JWT must be included in the `Authorization` header of all subsequent API requests. The format should be `Authorization: Bearer <jwt-token>`.
- Token Expiration: JWTs have an expiration date. Clients are responsible for monitoring this date and refreshing their tokens before they expire to

maintain uninterrupted access. This typically involves calling the `LoginV2` endpoint again with valid credentials.

---

3. Core Synchronization Endpoints

These are the foundational endpoints for retrieving and submitting data to the Mahak Accounting System.

3.1 **`GetAllData / GetAllDataV2`**

This endpoint is the cornerstone of the delta synchronization mechanism. It allows clients to efficiently retrieve changes made to data within Mahak.

- Purpose: To retrieve all entities from Mahak or, more importantly, only those entities that have been modified since the last synchronization point. This is achieved by providing the `rowVersion` of the data previously fetched.
- Endpoint: `POST /Sync/GetAllDataV2`
- Body Model: `RequestAllDataModel`
    - This model is used to specify which entities the client is interested in and what the last known `rowVersion` is for each of those entities.
    - **`rowVersion`** per entity: For each entity type (e.g., `banks`, `orders`, `products`), a `rowVersion` number is provided.
        - If a `rowVersion` of `0` (or a value lower than any existing record) is sent for an entity, it signifies that the client needs to retrieve all records for that entity type, including historical and currently active ones. This is typically used on the initial sync or after a significant data reset.
        - If a positive `rowVersion` is provided, the API will return only those records whose `rowVersion` is greater than the provided value, effectively delivering only the changes.
- Sample Request:

`json { "banksRowVersion": 0, "ordersRowVersion": 16500, "productsRowVersion": 22000, "customersRowVersion": 18000, "invoicesRowVersion": 25000 }` In this example, the client requests all banks, orders modified after `rowVersion` 16500, products modified after `rowVersion` 22000, and so on.

- Sample Response: The response is structured to group entities by type. Each entity type will contain a collection of records that have changed, along with the updated `rowVersion` for that entity type.

`json { "success": true, "code": 0, "message": "Data retrieved successfully", "data": { "banks": { "rowVersion": 2200, "records": [{ "bankId": 1, "bankClientId": 101, "bankCode": 300, "name": "Saman Bank", "deleted": false, "rowVersion": 2200 }, { "bankId": 2, "bankClientId": 102, "bankCode": 301, "name": "Melli Bank", "deleted": false, "rowVersion": 2190 }] }, "orders": { "rowVersion": 16505, "records": [{ "orderId": 500, "orderClientId": 100, "customerId": 20, "totalAmount": 95000, "deleted": false, "rowVersion": 16501 }, { "orderId": 501, "orderClientId": 102, "customerId": 25, "totalAmount": 120000, "deleted": false, "rowVersion": 16505 }] }, // ... other entities } }`

- Best Practices:
  - Request Only What You Need: To minimize network traffic and processing load, request only those entity types that are relevant to the client's functionality.
  - Persistent **`rowVersion`** Storage: Always store the highest `rowVersion` received for each entity type. This stored value will be used as the input for the next `GetAllDataV2` request for that specific entity, ensuring that only new or modified data is fetched.

3.2 `SaveAllData / SaveAllDataV2`

This endpoint is used to submit changes made on the client-side back to the Mahak Accounting System. It allows for the efficient, bulk submission of data.

- Purpose: To commit multiple entity changes, including creations, updates, and logical deletions, to the Mahak system in a single request. This is crucial for maintaining data consistency and transactional integrity.
- Endpoint: `POST /Sync/SaveAllDataV2`
- Body Model: `CommitDataModel`
    - This model is designed to carry collections of various Mahak entities. It can accommodate multiple related entities within a single payload, allowing for the submission of complex transactions, such as an order and its associated order details.
    - Support for Related Entities: The `CommitDataModel` can contain arrays of different entity types (e.g., `orders`, `orderDetails`, `payments`, `products`). Clients must ensure that dependent entities are committed together where necessary. For example, an `OrderModel` should be submitted along with its corresponding `OrderDetailModel` entries to maintain referential integrity.
    - **`deleted`** Flag Usage: When submitting data, clients should set the `deleted` flag to `true` for any records that are to be logically deleted in Mahak.
- Sample Request:

```
 json { "orders": [ { "orderId": null, // Server-generated ID for new orders "orderClientId": 100, // Client-defined unique identifier for this order "customerId": 20, "orderDate": "2025-08-09T10:00:00Z", "totalAmount": 95000, "deleted": false, "rowVersion": null // Not provided when creating/updating by client }, { "orderId": 501, // Existing server ID "orderClientId": 102, "customerId": 25, "orderDate": "2025-08-10T11:30:00Z", "totalAmount": 120000,
```

```
"deleted": false, "rowVersion": 16505 // Client may send its last known rowVersion } ], "orderDetails": [ { "orderDetailId": null, // Server-generated ID for new details "orderClientId": 100, // Links to the order with orderClientId 100 "productId": 50, "quantity": 2, "unitPrice": 47500, "deleted": false }, { "orderDetailId": null, "orderClientId": 102, "productId": 65, "quantity": 1, "unitPrice": 120000, "deleted": false } ], "products": [ { "productId": 50, "productClientId": 501, "productCode": "PROD-A", "name": "Advanced Gadget", "price": 47500, "deleted": false, "rowVersion": 22000 } ] }
```

- Notes on Submission:
  - Referential Integrity: It is paramount to commit dependent entities together. For instance, if you create a new `OrderModel`, you must also submit its associated `OrderDetailModel` records in the same or a preceding batch to avoid orphaned data. The API server is designed to handle these dependencies.
  - Batching for High Volumes: For applications that generate a very large number of changes, it is advisable to batch the data into smaller, manageable chunks. This prevents API request timeouts and improves the robustness of the synchronization process. The optimal batch size may need to be determined through testing and monitoring.

3.3 Other Endpoints

Beyond the core retrieval and submission of data, the Sync API offers specialized endpoints for handling specific business logic and operations.

- **`/Sync/SolveOrderDispute`**: This endpoint is dedicated to resolving specific disputes related to `OrderModel` records. It might be used to mark an order as disputed, provide a reason, or confirm a resolution that affects the order's status or financial implications.

- **`/Sync/SolveDispute`** : A more general endpoint for resolving various types of disputes within the accounting system, not necessarily tied to a specific order. This could apply to invoice disputes, payment discrepancies, or other financial disagreements.
- **`/Sync/Upload`** : This endpoint is provided for uploading files or attachments that are related to Mahak entities. For example, a client might upload a scanned invoice image, a proof of payment, or a signed contract to be associated with a specific customer or transaction. The request body would likely include the entity ID it relates to and the file content (e.g., base64 encoded or as multipart/form-data).

---

4. Entities & Models

The Mahak Sync API operates on various data entities, each representing a core concept within the accounting system. These entities share common attributes that are critical for synchronization and data management.

4.1 Common Entity Attributes

Every Mahak entity synchronized via the API typically includes the following key attributes:

- ID Fields:
  - Server-managed ID: This is the primary key managed by the Mahak database itself (e.g., `bankId` , `orderId` , `productId`). When creating a new record on the client and submitting it to Mahak, this ID is usually `null` or omitted, allowing Mahak to assign a new, unique ID. Mahak will return this assigned ID in the response.
  - Client-managed ID: A unique identifier generated by the client application (e.g., `bankClientId` , `orderClientId` , `productClientId`). This is crucial for establishing relationships and tracking records between the client and Mahak, especially before a server-generated ID is available. It helps in preventing duplicate submissions and mapping records.

- Code Fields: These are business-oriented codes that are often human-readable and used for identification within Mahak (e.g., `bankCode`, `productCode`).
- **`deleted`** (Boolean): A flag indicating the logical deletion status of the record. `false` means the record is active, while `true` signifies it has been logically deleted.
- **`rowVersion`** (Integer): An auto-incrementing integer that tracks changes to a record. Each time a record is updated in Mahak, its `rowVersion` is incremented. This is the backbone of the delta synchronization.

4.2 Example Entity: **`BankModel`**

This model exemplifies the common structure for a Mahak entity.

```
 {
  "bankId": 1,           // Server-managed ID
  "bankClientId": 101,   // Client-managed ID
  "bankCode": 300,       // Mahak-specific code
  "name": "Saman Bank",  // Entity specific data
  "deleted": false,      // Logical deletion status
  "rowVersion": 2200     // Synchronization version
}
```

4.3 Common Entity Types

The Sync API supports a wide range of Mahak entities. Here are some of the frequently used ones:

- Financial & Bank Entities:
  - `BankModel`: Information about bank accounts.
  - `BankGroupModel`: Grouping for bank accounts.
  - `CashModel`: Details of cash accounts or transactions.
  - `ChequeModel`: Information related to cheques.
- Checklist & Workflow Entities:
  - `ChecklistModel`: Data related to checklists or task management.

- Order & Sales Entities:
  - `OrderModel`: Represents sales orders, purchase orders, etc.
  - `OrderDetailModel`: Line items associated with an `OrderModel`.
- Customer & Supplier Entities:
  - `PersonModel`: General customer or contact information.
  - `PersonGroupModel`: Grouping for customers/suppliers.
- Product & Inventory Entities:
  - `ProductModel`: Information about items or services sold.
  - `ProductCategoryModel`: Categorization of products.
  - `ProductDetailModel`: Additional details or specifications for products.
- Invoice & Payment Entities: (Often included though not explicitly listed in the initial prompt, these are critical for accounting)
  - `InvoiceModel`: Represents customer invoices.
  - `InvoiceDetailModel`: Line items for invoices.
  - `PaymentModel`: Records of payments received or made.
  - `ReceiptModel`: Receipts issued for payments.
- Journal Entry Entities:
  - `JournalEntryModel`: Records of financial transactions.
  - `JournalEntryLineModel`: Individual lines within a journal entry.

Clients should refer to the specific API documentation or schema for the complete list of attributes and types for each entity.

---

5. Synchronization Logic

Effective data synchronization relies on understanding and implementing specific strategies for handling data changes, deletions, and potential conflicts. The Mahak Sync API provides mechanisms for all of these.

5.1 RowVersion Tracking

This is the core mechanism for delta synchronization.

- How it works: Each record in Mahak has a `rowVersion` attribute, which is an integer that increments every time the record is modified.
- Client Responsibility:
    1. On the initial sync, or after a full data refresh, the client should request all data for an entity by sending `rowVersion: 0` for that entity in the `GetAllDataV2` request.
    2. The client receives a set of records, each with its current `rowVersion`. The client must store the highest `rowVersion` value received for each entity type.
    3. For subsequent synchronizations, the client sends this stored highest `rowVersion` for each entity.
    4. Mahak's API then returns only those records whose `rowVersion` is greater than the `rowVersion` provided by the client.
    5. Upon receiving the new data, the client updates its stored highest `rowVersion` with the latest one returned by the API for that entity.

5.2 Logical Deletion

Mahak uses logical deletion to maintain data integrity and audit trails.

- How it works: Instead of physically removing a record from the database when it's no longer needed (e.g., a deleted customer), the record is marked with a `deleted` flag set to `true`.
- Client Responsibility:
    - When retrieving data using `GetAllDataV2`, clients should look for records where `deleted` is `true`. These records should be handled appropriately on the client-side, typically by marking them as deleted in the client's local database or by removing them from active display.
    - When sending data changes to Mahak via `SaveAllDataV2`, if a client wants to delete a record, it should include that record in the

payload with its `deleted` flag set to `true`. This signals to Mahak that the record should be marked as deleted.

5.3 Conflict Resolution

Conflicts can arise when both the client and Mahak modify the same record between synchronizations. The Sync API provides a strategy for handling these.

- General Principle: Mahak generally favors its own data as the authoritative source, especially for financial and critical records.
- Server Version Preference: For financial records (like invoices, payments, journal entries), it is often recommended to prefer the server version in case of a conflict. This means if both the client and Mahak have updated a record, the version in Mahak will be considered the correct one. The client should then update its local copy to match the Mahak version.
- Merging Non-Conflicting Fields: For less critical entities or when fields are independent, the API may support or expect clients to merge changes. For example, if a `PersonModel` has a `notes` field that was updated on the client and a `phone` number updated on the server, and these are considered independent, it might be possible to merge both changes. However, the `rowVersion` mechanism itself is the primary driver: if the server `rowVersion` is higher, the server's entire record usually prevails unless specific merging logic is implemented by the API.
- Client-Side Conflict Detection: Clients can detect a conflict if they attempt to `SaveAllDataV2` an updated record, and Mahak responds with an error indicating a version mismatch or that the record was updated by another process. In such scenarios, the client might need to re-fetch the latest version from Mahak, re-apply its client-specific changes to the new server version, and then resubmit.

Understanding these principles is crucial for building a robust synchronization engine that maintains data accuracy and consistency.

---

6. Error Handling

Effective error handling is critical for any integration. The Mahak Sync API follows standard RESTful practices for indicating success or failure of operations.

6.1 Standard Response Structure

Most API responses from the Mahak Sync API will adhere to a consistent JSON structure, providing details about the operation's outcome.

```
{
  "success": true,   // Boolean indicating overall success (true) or fa
  "code": 0,         // Integer status code. 0 usually means success. N
  "message": "",     // Human-readable message describing the status.
  "data": { }        // Contains the actual response data if successful
}
```

When an error occurs, `success` will be `false`, `code` will be a non-zero value representing the error type, and `message` will provide a description of the error. The `data` field might contain more specific error details, such as validation errors for specific fields.

6.2 Common Error Codes

The following are some common HTTP status codes and API-specific codes that clients should be prepared to handle:

- **`401 Unauthorized`**: This typically occurs when the provided JWT token is missing, invalid, or expired. The client must re-authenticate to obtain a new token.
- **`400 Bad Request`**: Indicates that the request payload is malformed, missing required fields, or contains data that violates validation rules (e.g., incorrect data types, invalid formats). The `message` and `data` fields in the response should provide details on what went wrong.

- **`429 Too Many Requests`** : This is a rate limiting error. The client has exceeded the allowed number of requests within a given time period. The client should implement a backoff strategy and retry the request later. The response might include headers like `Retry-After` to indicate when it's safe to retry.
- **`500 Internal Server Error`** : A generic server-side error. This could be due to an unexpected issue within the Mahak system. Clients should log this error and retry the request after a short delay. If the issue persists, it's advisable to contact Mahak support.
- API-Specific Codes: Beyond HTTP status codes, Mahak might use its own internal codes within the `code` field for finer-grained error reporting. For example:
  - `1001` : Invalid `DatabaseId` or `PackageNumber` during login.
  - `1002` : Authentication failed (incorrect username/password).
  - `2001` : Referential integrity violation (e.g., trying to link an order to a non-existent customer).
  - `2002` : Concurrency conflict (e.g., trying to update a record with an outdated `rowVersion`).
  - `3001` : Validation error on a specific entity field.

6.3 Retry Strategies

When encountering transient errors (like rate limiting or internal server errors), clients should implement robust retry mechanisms.

- Exponential Backoff: This is a standard and effective retry strategy. Instead of retrying immediately, the client waits for an increasing amount of time between retries. For example, retry after 1 second, then 2 seconds, then 4 seconds, and so on, up to a maximum number of retries or a maximum wait time. This prevents overwhelming the server and allows it time to recover.
- Jitter: To avoid all clients retrying at the exact same moment, it's good practice to add a small random delay (jitter) to the backoff period.
- Dead Letter Queue (DLQ): For integrations that rely on message queues or asynchronous processing, if a message consistently fails to be processed after multiple retries, it should be moved to a Dead Letter Queue. This

queue can then be monitored and analyzed by developers to diagnose and fix the root cause of the persistent failures.

---

7. Security Considerations

Securing the data synchronization process is paramount. The Mahak Sync API provides mechanisms and guidelines to ensure a secure integration.

- HTTPS Only: All communication with the Mahak Sync API must be conducted exclusively over HTTPS. This encrypts the data in transit, protecting it from eavesdropping and man-in-the-middle attacks. Ensure your client applications are configured to only connect to the API endpoints using the `https://` protocol.
- Secure JWT Storage:
  - Browser-based Clients: Avoid storing JWTs in `localStorage`. `localStorage` is vulnerable to Cross-Site Scripting (XSS) attacks, which could expose the token to malicious actors. Instead, consider using secure, HTTP-only cookies or storing tokens in memory within JavaScript applications, ensuring they are not transmitted with every request unless via the `Authorization` header.
  - Server-based Clients / Mobile Apps: Store JWTs in secure, encrypted storage mechanisms provided by the operating system or framework. Access to these storage locations should be restricted and protected.
- Server-Side Token Expiry Checks: While clients are responsible for refreshing tokens, the Mahak API server must also perform checks on the validity and expiration of the provided JWT for every incoming request. This ensures that even if a client fails to refresh its token, unauthorized access is prevented.
- Least Privilege Principle: Configure API access credentials (username/password for login) for users who have only the necessary permissions required for synchronization. Avoid using administrator credentials for routine sync operations.
- Input Validation: Always validate all data received from the API and any data being sent to the API. This includes checking data types, formats, lengths, and ranges to prevent injection attacks or unexpected behavior.

- Logging and Monitoring: Implement comprehensive logging on the client-side to track API calls, responses, and any errors. Monitor these logs regularly to detect suspicious activity or recurring issues.

By adhering to these security best practices, clients can significantly reduce the risk of unauthorized access and data breaches, ensuring the integrity and confidentiality of the synchronized data.

---

8. Example Integration Flow

This section outlines a typical step-by-step process for a client application integrating with the Mahak Sync API.

1. Client Initiates Synchronization: The client application determines it needs to synchronize data (e.g., on a scheduled interval, or when new client-side data is ready to be pushed).

2. Authentication:

    - The client sends a `POST` request to `/Sync/LoginV2` with its `username`, `password`, `DatabaseId`, and `PackageNumber`.
    - Upon successful authentication, the client receives a JWT bearer token and its `expireDate`.

3. Data Retrieval (Incremental Sync):

    - The client consults its stored `rowVersion` values for each entity type (e.g., `banksRowVersion`, `ordersRowVersion`).
    - The client sends a `POST` request to `/Sync/GetAllDataV2`, including these `rowVersion` values in the `RequestAllDataModel`.
    - The Mahak API returns data containing only records that have changed since the provided `rowVersion`s, along with updated `rowVersion`s for each entity.
    - The client processes this data:
        - Updates its local records with changed data.
        - Applies logical deletions (`deleted: true`) to corresponding local records.

        - Stores the new, highest `rowVersion` s for each entity for the next sync cycle.

4. Prepare Client-Side Changes for Submission:

    - The client identifies any new or modified records created or changed in its own system since the last sync.
    - These changes are mapped to the Mahak `CommitDataModel` format. New records will have their server-managed IDs (e.g., `orderId` , `bankId`) set to `null` . Records intended for deletion will have their `deleted` flag set to `true` .
    - The client groups related entities together (e.g., `orders` and `orderDetails`).

5. Data Submission:

    - The client sends a `POST` request to `/Sync/SaveAllDataV2` with the prepared `CommitDataModel` in the request body, including the JWT in the `Authorization` header.
    - Mahak processes the submitted data, creating, updating, or logically deleting records.
    - Mahak returns a response indicating success or failure of the commit operation.

6. Handle Submission Response:

    - Success: If successful, the client acknowledges the successful submission of its changes.
    - Failure: If there are errors (e.g., validation errors, concurrency conflicts), the client must parse the error response to understand the cause.
        - For validation errors, the client may need to correct the data and resubmit.
        - For concurrency conflicts (outdated `rowVersion`), the client might need to re-fetch the latest version of the affected record from Mahak, re-apply its changes to the new server version, and then resubmit.
        - For transient errors, the client should employ retry logic (e.g., exponential backoff).

7. Token Refresh:
    - The client continuously monitors the `expireDate` of its JWT.
    - Before the token expires, or if an `Unauthorized` error is received, the client repeats step 2 to obtain a new JWT.

This cyclical flow ensures that data remains synchronized and consistent between the client application and the Mahak Accounting System.

---

9. Best Practices

To maximize the efficiency, reliability, and scalability of your integration with the Mahak Sync API, consider the following best practices:

- Synchronize Low-Churn Entities Less Frequently: Not all entities change at the same rate. For example, `BankModel` or `PersonGroupModel` might update infrequently compared to `OrderModel` or `PaymentModel`. Optimize your synchronization schedule by polling less volatile data less often, while polling high-churn entities more frequently. This reduces unnecessary API calls and server load.

- Handle Large Datasets with Paging: While `GetAllDataV2` is designed for bulk retrieval, if an entity type contains an extremely large number of records that have changed, the response payload could become very large. It is good practice to check if the API supports paging for such scenarios (though the `rowVersion` approach generally mitigates the need for entity-level paging in delta sync). If possible, implement client-side pagination or be prepared to handle large responses efficiently.

- Always Record Both **`rowVersion`** and **`deleted`** State: These two attributes are fundamental for correct synchronization.
    - **`rowVersion`** : Essential for delta synchronization; without it, you'd have to re-download all data every time.
    - **`deleted`** : Crucial for maintaining data integrity and replicating deletions accurately. If you ignore the `deleted` flag, your client's data will diverge from Mahak's, leading to inconsistencies.

- Use DLQ for Unprocessed Messages in Queue-Based Integration: If your integration architecture uses message queues (e.g., for background processing of `SaveAllDataV2` operations or for handling webhooks from Mahak), ensure you implement a Dead Letter Queue (DLQ) mechanism. If a message (or a batch of data) consistently fails to be processed after multiple retries, it should be moved to the DLQ. This prevents critical data from being lost and allows for manual inspection and reprocessing by support personnel or developers.
- Error Handling and Retry Logic: Implement robust error handling as described in Section 6. Use exponential backoff for retries, and be prepared to handle specific error codes like rate limiting (`429`) and unauthorized access (`401`). Test your retry logic thoroughly.
- Client-Side Unique Identifiers: Leverage client-managed IDs (`clientId`) effectively. They are invaluable for mapping records between your system and Mahak, especially before a server-generated ID is available or if server IDs are not directly exposed in all contexts.
- Transaction Management: When submitting related data (e.g., an invoice and its payment), aim to include them in the same `SaveAllDataV2` request to ensure transactional consistency. If the submission fails for the entire batch, you'll need a strategy to roll back or retry the whole transaction.

By incorporating these practices, you will build a more resilient, efficient, and accurate integration between your application and the Mahak Accounting System.

Version: API V2 – Generated 2025-08-09