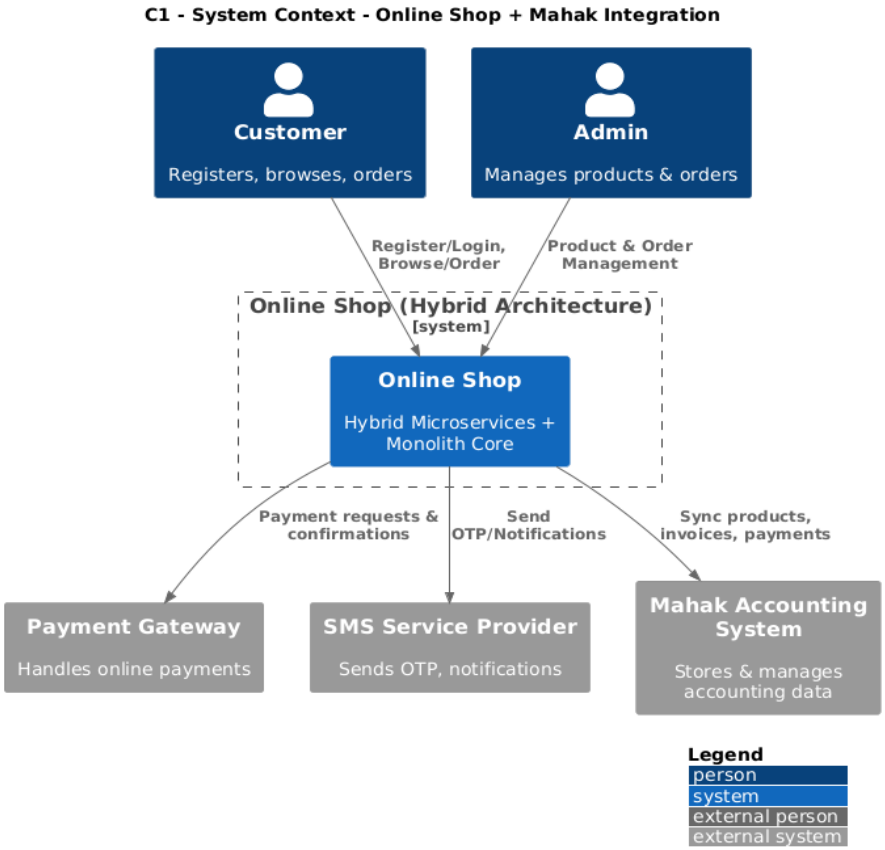
Hybrid Online Store – Mahak Integration Architecture

1. Project Overview

This document outlines the comprehensive architecture and detailed integration strategy for seamlessly connecting the Hybrid Online Store with the Mahak Accounting System. It serves as a foundational guide for developers and stakeholders, encompassing the complete technical blueprint. The document includes detailed explanations of the C4 model diagrams at various levels of abstraction, sequence diagrams illustrating key data flows, justifications for the chosen technology stack, and in-depth guidelines for interacting with the Mahak Accounting System's APIs. This integration is critical for ensuring data consistency, operational efficiency, and a unified view of business operations across both systems.

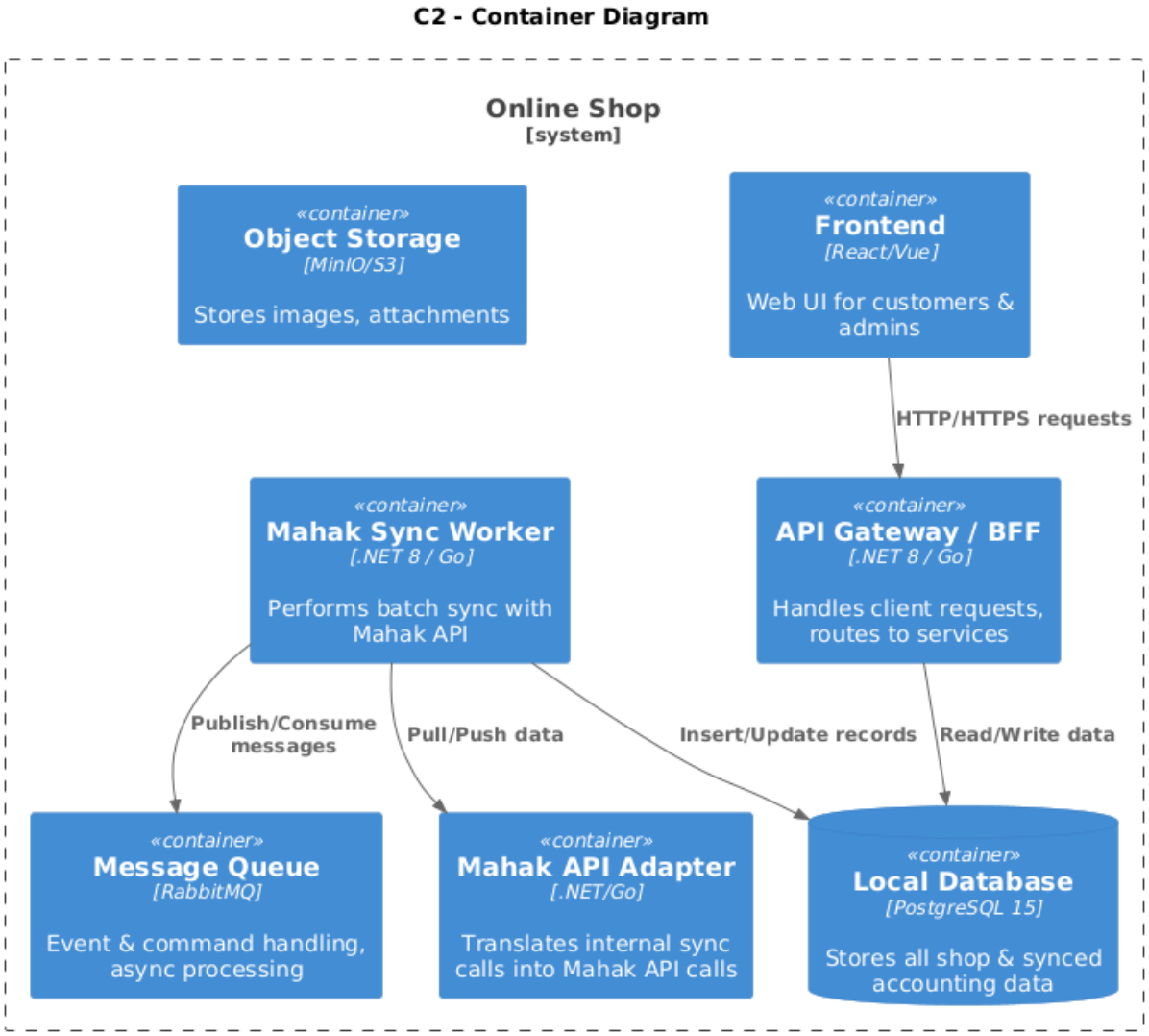
2. System Context (C1)



Description: The System Context diagram (C1) provides a high-level overview of the Hybrid Online Store within its surrounding ecosystem. It depicts the primary interactions between the core system and its external actors and dependencies.

- **Customer:** The end-user interacting with the Hybrid Online Store to browse products, place orders, and manage their accounts.
- **Admin:** Internal users responsible for managing the Hybrid Online Store, including product management, order fulfillment, and user administration.
- **Payment Gateway:** An external service responsible for processing all financial transactions initiated by customers. The Online Store integrates with this for payment authorization and confirmation.
- **SMS Service:** An external service used for sending transactional notifications to customers, such as order confirmations, shipping updates, and delivery alerts.
- **Mahak Accounting System:** The central financial and accounting system. This is the primary external system with which the Hybrid Online Store needs to integrate for data synchronization (e.g., orders, customers, products, payments) to maintain accurate financial records.
- **Online Store (Hybrid):** Represented as the central system in this context. It encompasses all the functionalities of the online store, including product catalog, order management, user management, and payment processing. This diagram sets the stage for understanding where the Mahak integration fits into the broader business landscape.

3. Containers (C2)



Description: The Containers diagram (C2) dives deeper into the architectural structure of the Hybrid Online Store, illustrating the major deployable units or services (containers). This level of detail shows how the system is broken down into independent, scalable, and manageable units, and how they interact.

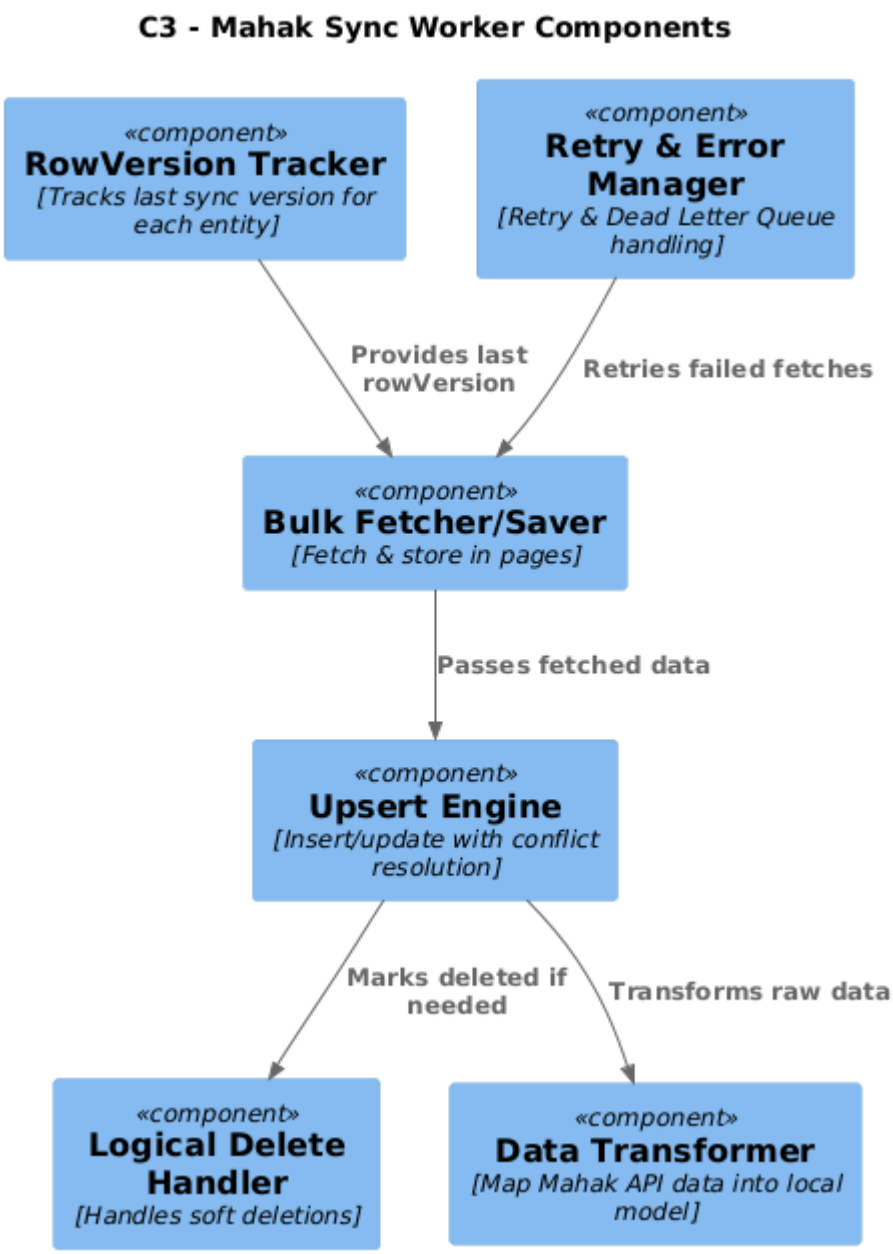
- **API Gateway/BFF (Backend for Frontend):** Acts as a single entry point for all client requests (Frontend, Mobile Apps, etc.). It routes requests to appropriate backend services, handles authentication, rate limiting, and potentially transforms responses. This layer is crucial for managing external access and securing the backend services.
- **Frontend:** The user interface layer, typically a Single Page Application (SPA) or a set of web pages, that customers and admins interact with. It communicates with the backend via the API Gateway.
- **Local Database (PostgreSQL 15):** The primary relational database for the Hybrid Online Store. It stores all operational data related to products, users, orders, inventory, sessions, etc. PostgreSQL is chosen for its robustness, advanced features (like JSONB and Upsert), and strong support in cloud-native environments.
- **Message Queue (RabbitMQ):** A robust and reliable message broker. It facilitates asynchronous communication between different services within

the Hybrid Online Store and with external systems. Key uses include decoupling services, handling background tasks, and enabling event-driven architectures. This is fundamental for resilient integration with Mahak.

- **Mahak Sync Worker:** A dedicated service responsible for all data synchronization operations with the Mahak Accounting System. It consumes messages from the Message Queue, interacts with the Mahak API Adapter, and manages the complexities of data transformation, error handling, and retries.
- **Object Storage (MinIO/S3):** Used for storing large binary objects such as product images, documents, or backups. MinIO is an open-source, S3-compatible object storage solution, providing scalability and cost-effectiveness.
- **Mahak API Adapter:** A service that encapsulates all interactions with the Mahak Accounting System's APIs. It handles authentication, request formatting, response parsing, and error mapping for Mahak-specific operations. This adapter provides an abstraction layer, making the Mahak Sync Worker independent of the direct Mahak API specifics.

These containers are designed to be independently deployable and scalable, contributing to a resilient and maintainable architecture.

4. Components (C3 – Mahak Sync Worker)



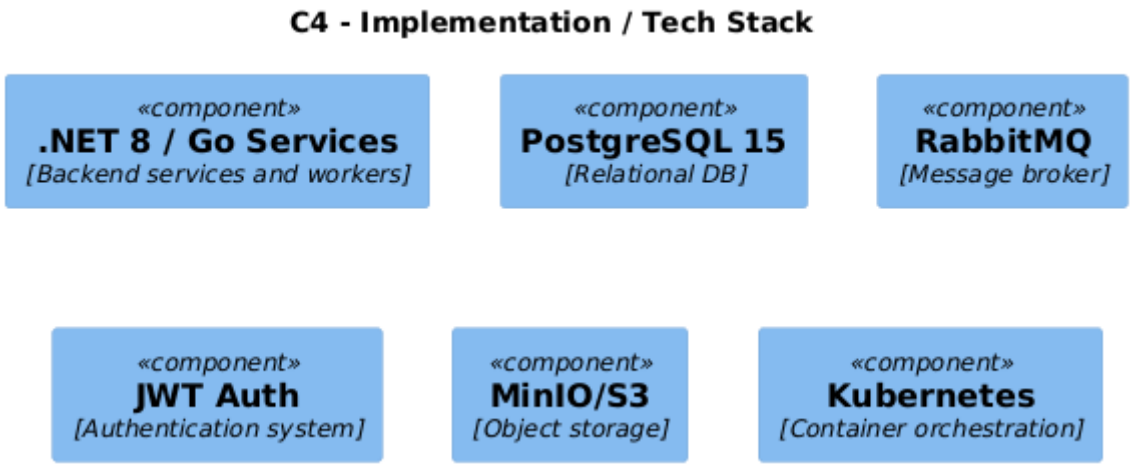
Description: The Components diagram (C3) for the Mahak Sync Worker provides a granular view of its internal structure. It breaks down the worker service into its constituent logical components, each responsible for a specific aspect of the synchronization process. This level of detail is crucial for understanding the implementation of the synchronization logic.

- **RowVersion Tracker:** This component is responsible for managing and tracking the `RowVersion` (or a similar mechanism like timestamp or version number) for entities synchronized with Mahak. It keeps track of the last successfully synchronized `RowVersion` for each entity type, enabling delta synchronization and efficient fetching of only changed data from the Online Store or Mahak.

- **Bulk Fetcher/Saver:** Handles the efficient retrieval of data in bulk from the Hybrid Online Store's database and the efficient saving of data in bulk to Mahak via its APIs. It is optimized to work with batching and pagination to manage large datasets.
- **Upsert Engine:** This component implements the logic for upserting (inserting if not present, updating if present) data into the Mahak system. It leverages Mahak's API capabilities, likely through batch save operations, to efficiently manage data updates.
- **Logical Delete Handler:** Manages the propagation of logical deletions from the Online Store to Mahak. When an entity is logically deleted in the Online Store (marked as inactive rather than physically removed), this component ensures a corresponding "delete" or "inactivate" operation is performed in Mahak to maintain data consistency.
- **Retry & Error Manager:** A critical component responsible for implementing robust error handling and retry mechanisms. It manages transient API errors, network failures, and data validation issues. It can implement strategies like exponential backoff and utilize Dead Letter Queues (DLQs) for persistent failures.
- **Data Transformer:** This component is responsible for converting data from the Hybrid Online Store's domain model to the format expected by Mahak's APIs, and vice-versa. This includes mapping fields, formatting data types, and ensuring adherence to Mahak's data schema.

These components work in concert to ensure reliable and efficient data synchronization between the two systems.

5. Code/Implementation Details (C4)



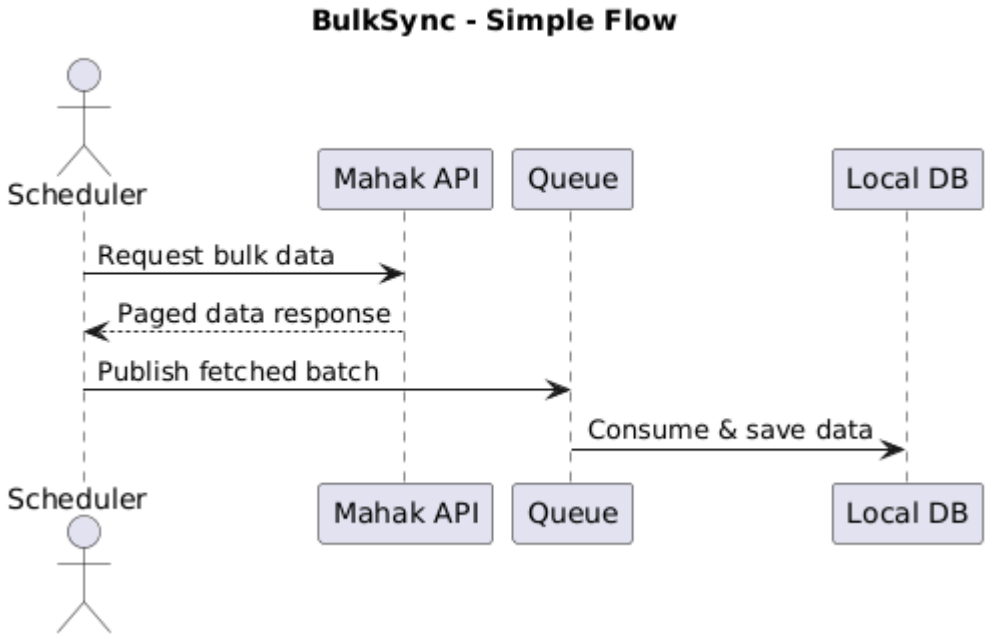
Description: The Code/Implementation Details diagram (C4) focuses on the technologies and practices used within the core services of the Hybrid Online Store, particularly those relevant to the Mahak integration. This level provides concrete implementation choices that guide development.
Tech Stack:

- **Languages/Frameworks:**
  - **.NET 8:** A modern, high-performance, open-source framework for building various applications, including microservices. Its strong typing, extensive libraries, and asynchronous capabilities make it well-suited for data-intensive and API-driven integrations.
  - **Go (Golang):** Known for its concurrency primitives, performance, and simplicity, Go is an excellent choice for building efficient microservices and background workers like the Mahak Sync Worker.
  - *Justification:* Both .NET 8 and Go offer excellent performance, strong ecosystem support, and are well-suited for building scalable microservices. The choice between them can depend on team expertise and specific service requirements.
- **Database:**
  - **PostgreSQL 15:** A powerful, open-source relational database.
  - *Justification:* Chosen for its robustness, ACID compliance, advanced features like JSONB support, sophisticated indexing, and excellent performance. Its strong community support and compatibility with container orchestration platforms make it a reliable choice for the Online Store's core data.
- **Message Broker:**
  - **RabbitMQ:** A widely adopted, feature-rich message broker implementing the Advanced Message Queuing Protocol (AMQP).
  - *Justification:* Provides reliable asynchronous communication, supporting both publish/subscribe and point-to-point messaging

patterns. Its features like durable queues, message acknowledgments, and support for Dead Letter Queues (DLQs) are crucial for ensuring the reliability of data synchronization and handling failures gracefully.

- **Object Storage:**
  - **MinIO / S3 Compatible:** For storing binary assets.
  - *Justification:* MinIO offers an open-source, S3-compatible object storage solution. S3 compatibility ensures interoperability with cloud services and a standardized API for accessing stored objects, making it a scalable and cost-effective choice for managing media and documents.
- **Auth:**
  - **JWT (JSON Web Token) Authentication:** Used for secure authentication and authorization between services and clients.
  - *Justification:* JWT provides a stateless and scalable way to manage user sessions and API access. It's commonly used in microservice architectures for securely transmitting claims between parties.
- **Orchestration:**
  - **Kubernetes:** The de facto standard for container orchestration.
  - *Justification:* Essential for deploying, scaling, and managing the containerized services. Kubernetes provides features like self-healing, automated rollouts and rollbacks, service discovery, and load balancing, ensuring high availability and resilience of the Hybrid Online Store and its integration components.

6. Sequence Diagram – BulkSync (Simple)



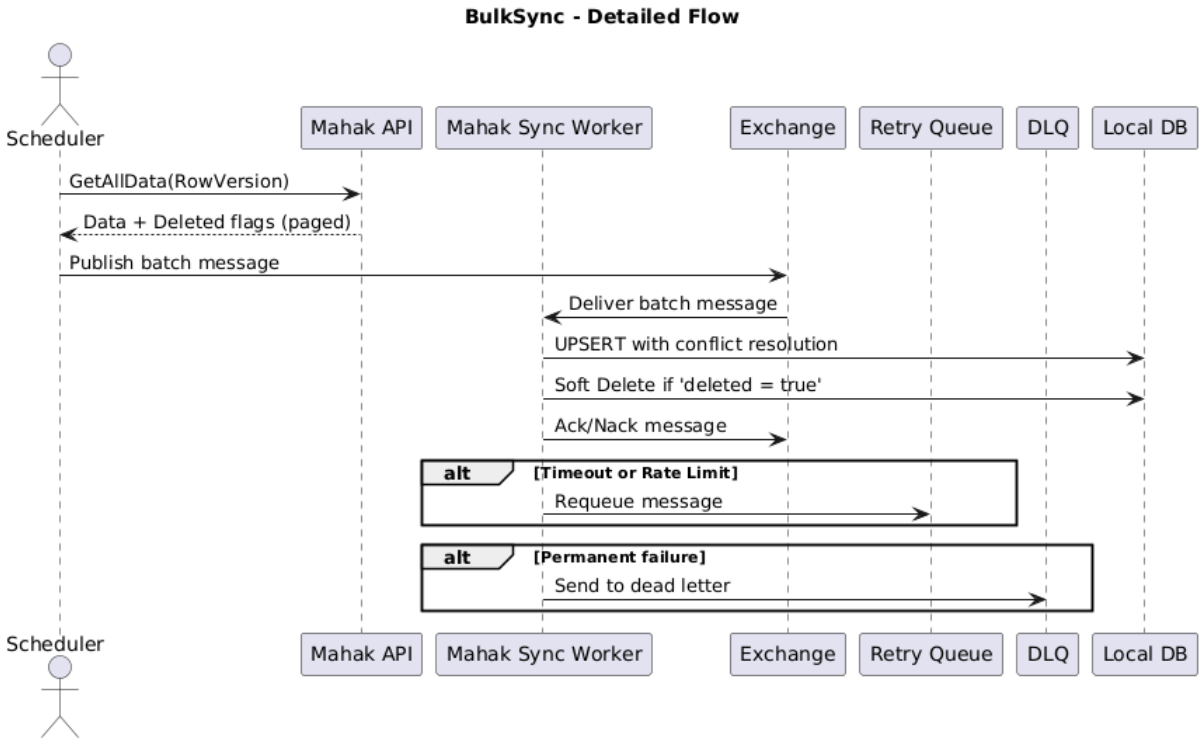
Description: This sequence diagram provides a simplified, high-level view of the bulk synchronization process. It outlines the fundamental steps and actors involved without detailing intricate error handling or specific data payloads. This diagram is useful for understanding the overall flow of information.

1. **Scheduler/Trigger:** Initiates the bulk synchronization process (e.g., on a schedule or an event).
2. **Online Store:** The source system from which data needs to be synchronized to Mahak.
3. **Mahak Sync Worker:** The orchestrator that pulls data from the Online Store and pushes it to Mahak.
4. **Mahak API Adapter:** Handles the communication with the Mahak API.
5. **Mahak Accounting System:** The target system receiving the synchronized data.

Flow: * The process is triggered. * The Mahak Sync Worker fetches a batch of data from the Online Store (this might involve querying the database directly or via an internal API). * The Mahak Sync Worker then sends this batch of data to the Mahak Accounting System via the Mahak API Adapter. * The Mahak Accounting System receives and processes the data. * A confirmation or response is sent back.

This simplified view highlights the "what" of the synchronization without getting bogged down in the "how."

7. Sequence Diagram – BulkSync (Detailed)



Description: This detailed sequence diagram illustrates the robust flow of bulk synchronization, incorporating crucial elements like the message queue, error handling, and specific API interactions. It provides a technical blueprint for implementing the synchronization.

Actors/Components:

- **Scheduler:** Triggers the sync process periodically.
- **Online Store DB:** Source database.
- **Mahak Sync Worker:** The main service.
- **Message Queue (RabbitMQ):** Facilitates asynchronous tasks.• **Mahak API Adapter:** Encapsulates Mahak API calls.
- **Mahak Accounting System:** Target system.
- **Retry Queue/DLQ:** For handling failed messages.

Flow:

1. **Trigger:** The Scheduler initiates a task to sync data (e.g., "Sync Orders").
2. **Publish to Queue:** The Scheduler publishes a message to a specific queue in RabbitMQ (e.g., `order_sync_request`). This message might contain parameters like the sync date range or last `RowVersion`.
3. **Consume from Queue:** The Mahak Sync Worker consumes the message from the `order_sync_request` queue.

4. **Fetch Data from Online Store:** The Mahak Sync Worker (or a sub-component) queries the Online Store DB to retrieve new or updated entities (e.g., orders, customers) based on the criteria in the message (e.g., `RowVersion` greater than last synced version).
5. **Transform Data:** The Mahak Sync Worker's Data Transformer component converts the fetched Online Store entities into the format required by Mahak.
6. **Prepare for Bulk Save:** The transformed data is batched into logical groups for efficient processing by Mahak.
7. **Call Mahak API (Save):** The Mahak Sync Worker invokes the Mahak API Adapter to send the batched data to Mahak's `/Sync/SaveAllDataV2` endpoint.
8. **Mahak Processing:** Mahak Accounting System receives the batch, processes each record, and attempts to save them.
9. **Mahak Response:** Mahak returns a response indicating the success or failure of each record in the batch.
10. **Handle Response:**
    - **Success:** The Mahak Sync Worker updates its internal state (e.g., last synced `RowVersion`) and potentially publishes a success event.
    - **Partial Success/Specific Errors:** Records that failed processing are identified. For transient errors (e.g., network timeout, temporary service unavailability), the Mahak Sync Worker's Retry & Error Manager component might:
        - Place the failed records or the entire batch back into a retry queue with a delay (e.g., exponential backoff).
        - Log the specific error details.
    - **Permanent/Unrecoverable Errors:** For data validation errors or unrecoverable issues, the Mahak Sync Worker might:
        - Place the failed records into a Dead Letter Queue (DLQ) for manual investigation.
        - Log detailed error information, including the problematic data and Mahak's error message.
11. **Retry Mechanism:** If records are placed in a retry queue, the Mahak Sync Worker will consume them again after a delay, repeating steps 4-10.
12. **Update Last Sync State:** Upon successful processing of a batch, the Mahak Sync Worker updates the `RowVersion` tracker to ensure the next sync cycle fetches only subsequent changes.

This detailed flow emphasizes the resilience and robustness required for reliable data synchronization.

8. Technology Stack & Justification

A carefully selected technology stack is crucial for the success and maintainability of the Hybrid Online Store and its integration with Mahak. Each component is chosen based on its specific strengths and suitability for the task.

- **PostgreSQL 15:**
  - **Justification:**
    - **Free License:** Eliminates licensing costs, making it a cost-effective solution.
    - **Strong Linux/Kubernetes Support:** Seamless integration with cloud-native deployment environments.
    - **Advanced JSONB & Upsert:** Enables efficient handling of semi-structured data and optimized update operations, vital for complex data models and synchronization logic.
    - **Active Ecosystem:** A vast community, extensive documentation, and a rich set of tools and extensions contribute to its reliability and ease of development.
    - **ACID Compliance:** Ensures data integrity and reliability, which is paramount for financial and transactional data.
- **RabbitMQ:**
  - **Justification:**
    - **Open-Source Message Broker:** Widely adopted and mature, offering a cost-effective and powerful messaging solution.
    - **Supports Pub/Sub and Point-to-Point:** Provides flexibility in designing communication patterns between services.
    - **High Reliability:** Features like durable queues, message persistence, and acknowledgments guarantee message delivery.
    - **DLQ (Dead Letter Queue) Support:** Essential for robust error handling. Failed messages that cannot be processed are rerouted to a DLQ for investigation, preventing data loss.
    - **Retry Queues:** Enables the implementation of sophisticated retry strategies (e.g., exponential backoff) for transient failures, improving the overall success rate of synchronization tasks.
- **MinIO (S3 Compatible):**
  - **Justification:**
    - **Scalable Open-Source Object Storage:** Provides a flexible and scalable solution for storing binary assets like product images, videos, or documents.
    - **Inexpensive:** Lower operational cost compared to managed cloud storage services in some scenarios.
    - **Compatible with S3 API:** Ensures broad compatibility with existing tools and cloud infrastructure, simplifying integration and management.

- **.NET 8 / Go:**
    - **Justification:**
        - **High-Performance:** Both languages are known for their efficiency, crucial for handling large data volumes and real-time operations.
        - **Robust Ecosystems:** Extensive libraries, frameworks, and community support accelerate development.
        - **Suitable for Microservices:** Designed for building scalable, independent services.
        - **Strong Concurrency Support:** Essential for building resilient and performant background workers and API services that can handle multiple requests simultaneously. .NET's `async/await` and Go's goroutines and channels are key enablers.
- **Kubernetes:**
    - **Justification:**
        - **Automated Deployment, Scaling, and Management:** Simplifies the operational overhead of deploying and managing a microservices-based architecture.
        - **Self-Healing:** Automatically restarts failed containers, ensuring high availability.
        - **Zero-Downtime Deployments:** Enables seamless updates and rollbacks without interrupting service.
        - **Resource Optimization:** Efficiently utilizes underlying compute resources.

9. Mahak API Overview

This section details the essential aspects of integrating with the Mahak Accounting System's APIs for data synchronization. Understanding these guidelines is crucial for the Mahak Sync Worker and Mahak API Adapter components.

Authentication: * **Login to Retrieve Token:** Interaction with Mahak's APIs typically begins with an authentication process. You will need to call a login endpoint (e.g., `/Sync/Login` or `/Sync/LoginV2`) with valid credentials. * **Token in Authorization Header:** The retrieved authentication token must be included in the `Authorization` header of all subsequent API requests. The scheme is usually `Bearer`. * Example: `Authorization: Bearer <your_api_token>`

Core Synchronization Endpoints: These are the primary endpoints used for data exchange between the Online Store and Mahak.

- `/Sync/Login` & `/Sync/LoginV2` : Used to obtain an authentication token. `LoginV2` might offer enhanced security or parameters.
- `/Sync/GetAllData` & `/Sync/GetAllDataV2` : Used to fetch data from Mahak. The `V2` endpoints are likely improved versions, possibly offering more control over data selection (e.g., filtering by `RowVersion`) or returning more comprehensive data. These are typically used for initial data dumps or full reconciliation.
- `/Sync/SaveAllData` & `/Sync/SaveAllDataV2` : Used to send data from the Online Store to Mahak. These endpoints are critical for pushing updates such as new orders, customer information, or product changes. They are designed to handle bulk operations and often support batch processing.
- `/Sync/SolveOrderDispute` , `/Sync/SolveDispute` : These endpoints are used to resolve data conflicts or errors reported by Mahak. When synchronization fails due to data inconsistencies, these endpoints can be used to correct or resolve the issues, possibly by providing additional context or manually overriding certain values.
- `/Sync/Upload` : This endpoint is likely for uploading files, such as documents or reports, which might be related to transactions or specific data entries.

Entities & Models: The Mahak APIs expose various entities relevant to accounting and business operations. Key entities include:

- **Banks, Bank Groups:** For managing financial accounts.
- **Products:** Details about items sold in the online store.
- **Orders:** Information about customer purchases.
- **Customers:** Data about clients and their details.
- **Payments:** Records of financial transactions.
- **And more:** Such as invoices, receipts, inventory adjustments, etc.

Key API Design Principles: * **Logical Deletion:** Entities in Mahak likely support logical deletion (marking as inactive) rather than physical removal. The integration must respect this to maintain historical data integrity. * **RowVersion for Delta Synchronization:** Each record in Mahak (and ideally in the Online Store) has a `RowVersion` (or a similar versioning mechanism like a timestamp). This is crucial for efficient delta synchronization. By tracking the last synchronized `RowVersion`, you can fetch only records that have changed since the last sync, significantly reducing data transfer volume and processing time.
Synchronization Logic:

- **Optimize with RowVersion :** Always utilize `RowVersion` to fetch only changed data from both systems. This is the cornerstone of efficient and scalable synchronization.
- **Bulk Data Operations:** Leverage Mahak's `/Sync/SaveAllDataV2` (or similar) endpoints for batching multiple records into a single API call. This reduces network overhead and improves processing throughput.
- **Group Related Entities:** When sending data, group related entities together. For example, an order might include customer details, payment information, and line items. Sending these as a coherent unit can simplify processing on Mahak's side and reduce the number of API calls.

Error Handling:

- **Implement Retries:** For transient errors (e.g., network issues, temporary service unavailability, rate limiting), implement robust retry mechanisms with exponential backoff.
- **Use DLQ for Message Failures:** Messages that repeatedly fail processing (due to unrecoverable data errors or persistent service issues) should be sent to a Dead Letter Queue (DLQ) for manual inspection and resolution.
- **Conflict Resolution:** Utilize the `/Sync/SolveOrderDispute` and `/Sync/SolveDispute` endpoints when Mahak reports data conflicts or validation errors. These endpoints require careful design to manage how conflicts are detected and resolved.

Security Considerations:

- **JWT for Authentication:** Ensure all requests are authenticated using the JWT obtained from the login endpoint.
- **HTTPS for all API Calls:** All communication with Mahak APIs must be over HTTPS to ensure data is encrypted in transit.
- **Enforce Role-Based Access Control (RBAC):** If Mahak's API supports it, ensure the credentials used have only the necessary permissions for synchronization tasks to adhere to the principle of least privilege.

Example Integration Flow:

1. **Obtain Token:** Call `/Sync/LoginV2` with valid credentials to get an authentication token. Store this token securely and refresh it as needed before it expires.
2. **Request Updated Entities:** Periodically (or based on triggers), call `/Sync/GetAllDataV2` to fetch entities that have changed in Mahak since the last known `RowVersion` from the Online Store.
3. **Fetch Online Store Changes:** Query the Hybrid Online Store's database for entities (e.g., orders, customers) that have a `RowVersion` greater than the last successfully synchronized `RowVersion` .
4. **Transform and Send Data:** Transform the fetched Online Store data into Mahak's expected format and send it using `/Sync/SaveAllDataV2` . This might involve multiple calls for different entity types or batched groups of entities.
5. **Handle Responses and Disputes:** Process the responses from `/Sync/SaveAllDataV2` . If errors occur, implement retry logic or use dispute endpoints (`/Sync/SolveOrderDispute` , `/Sync/SolveDispute`) to resolve conflicts and resubmit data.
6. **Update Last Sync RowVersion :** After successfully synchronizing a batch or a set of data, update the stored `RowVersion` to reflect the latest synchronized state for subsequent cycles.

End of Document