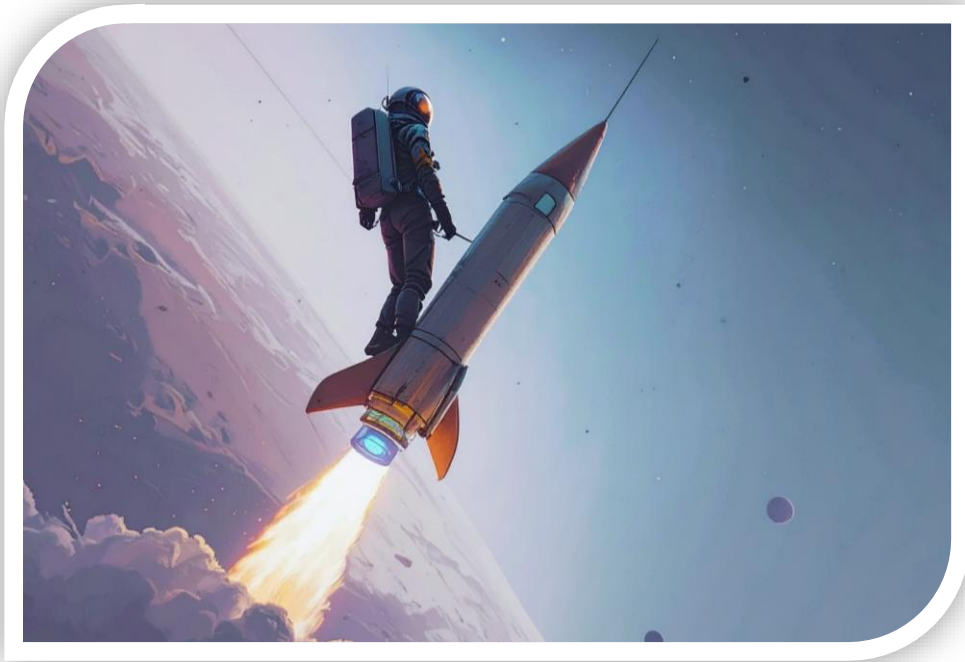


Best Practice API Design



David Wright Sr –
Enterprise Architect

Contents

Introduction	3
API Standards and Best Practices	4
Key API principles.....	4
API Design and Documentation	4
Security.....	5
Synchronous versus Asynchronous Mechanisms	6
Appendix – Asynchronous API Pattern and Design Strategies	7
Designing Asynchronous APIs for Atomic Transactions	7
1. Implement idempotency	7
2. Employ messaging and task queues	7
3. Implement compensating transactions.....	7
4. Ensure proper error handling and recovery	7
5. Be mindful of eventual consistency	8
6. Additional best practices	8
Examples of Saga Pattern Implementation with Asynchronous APIs	8
1. Choreography-based Saga (event-driven).....	8
2. Orchestration-based Saga (centralized control)	9



Introduction

API standards generally adhere to common industry practices for web services, primarily utilizing RESTful principles and JSON payloads for data exchange. While specific implementations may vary slightly across different products and services, the overarching approach emphasizes simplicity, scalability, and ease of integration.

Key aspects of API standards include:

- **RESTful Architecture:** Many APIs are built on REST principles, enabling interaction through standard HTTP methods (GET, POST, PUT, DELETE) for resource manipulation.
- **JSON Payloads:** Data is typically exchanged using JSON (JavaScript Object Notation) format for requests and responses, providing a lightweight and human-readable structure.
- **Authentication:** Best practices strongly suggest OAuth Server-to-Server Authentication for secure access to APIs.
- **API-First Philosophy:** Many SaaS products are developed with an "API first" approach, ensuring that core functionalities are exposed through well-defined APIs for programmatic access and integration. New products and services, where appropriate, can be developed with the same approach.



API Standards and Best Practices

API best practices encompass a range of considerations for designing, developing, and maintaining APIs to ensure they are robust, secure, scalable, and user-friendly.

With some products, the emphasis is API first development, meaning these products are designed with APIs as a core component. Instead of building a user-facing application first and adding an API later, these products are designed around their data structures and integration contracts before anything else, making their backend logic available as reusable building blocks for end users. In the end user context this allows End User services to be easily integrated with other API First capabilities from many sources.

The End User's API ecosystem should adhere to established industry standards and best practices to ensure interoperability and ease of use. Here's a summary:

Key API principles

- **RESTful architecture:** Most APIs should be designed following the Representational State Transfer (REST) architectural style, which leverages standard HTTP methods like GET, POST, PUT, and DELETE for CRUD (Create, Read, Update, Delete) operations. REST APIs promote statelessness, meaning each request contains all the information needed for processing, and the server doesn't store client-specific context between requests.
- **Uniform interface:** APIs should emphasize a consistent and predictable way of interacting with their various services. This includes standardized URL patterns, consistent use of HTTP methods, and predictable response formats.
- **Resource-based architecture:** Data and functionality are represented as resources, each uniquely identified by a URI (Uniform Resource Identifier).
- **Statelessness:** This design principle enhances scalability and independence between client and server components by eliminating the need for the server to maintain session information between requests.
- **Caching Ability:** [REST APIs](#) can leverage HTTP caching mechanisms, improving performance and reducing server load by allowing clients to store responses for a specified duration.
- **Emphasis should be placed upon client user/developer experience:** Developers should aim to provide positive user/developer experiences with APIs. This should include clear documentation, easy-to-use SDKs, and helpful tools.

API Design and Documentation

- **Standardized documentation:** Openthinka places importance on comprehensive and well-structured API documentation. Good documentation includes:
 - Clear and concise language.



- Examples of API calls, parameters, and expected responses.
 - Code samples in popular languages like Java, JavaScript, PHP, and Python.
 - Explanation of error messages and status codes.
 - Quickstart guides and tutorials.
- **Use of tools like Swagger/OpenAPI:** Openthinka promotes the use of tools like Swagger or OpenAPI for defining, documenting, and generating code for REST APIs.
- **Clear and consistent naming conventions:** Using nouns for resources (e.g., /users instead of /getUsers) and following hierarchical structures in URLs improves API readability and usability.
- **Versioning strategies:** Best practices use versioning methods, such as URI, header-based, and query parameter versioning, allowing users to introduce changes to business capabilities without breaking existing integrations and ensure backward compatibility.
- **Error Handling:** Provide clear, informative error messages and use standard HTTP status codes.
- **Documentation:** Create comprehensive and up-to-date documentation using tools like OpenAPI (Swagger) to guide developers.

Security

- **Robust authentication:** End User deliverables must employ strong authentication protocols like OAuth 2.0 and JWT (JSON Web Tokens) to secure API access.
- **Secure data handling:** Data in transit and at rest must be protected through encryption (HTTPS/TLS v1.2). Sensitive information is to be avoided in URLs and logs.
- **Access control and authorization:** A consistent Identity Management Services (IMS) should be used to manage access, including authentication and authorization, to APIs.
- **Protection against vulnerabilities:** Best practices must be implemented to prevent common vulnerabilities like SQL injection, cross-site scripting (XSS), and DDoS attacks. This includes input validation and using API gateways.

By adhering to such standards, the End User API strategy can promote a robust, secure, and user/developer-friendly environment for building integrations and leveraging products and services into other capabilities.



Synchronous versus Asynchronous Mechanisms

Asynchronous API

- **Non-blocking behavior:** When a client makes an asynchronous API call, it sends the request and then continues with other tasks without waiting for an immediate response. The API call is processed in the background.
- **Callback or polling mechanisms:** Asynchronous APIs typically use mechanisms like callbacks (where the server notifies the client when the task is complete) or polling (where the client periodically checks the status of the request) to deliver the result. Webhooks are a common way to implement asynchronous communication.

Synchronous API

When a client makes a synchronous API call, it sends the request and then waits for the server to process the request and send back a response before it can proceed with any other tasks. The client's execution is "blocked" until the API call completes.

- **Real-time interaction:** Synchronous APIs are suitable for scenarios where an immediate response is required and subsequent operations depend directly on the API's return value. Examples include user login authentication, immediate data retrieval for UI rendering, or real-time transaction processing.
- **Potential for latency:** If the API's processing takes a long time, the client might experience delays or a frozen interface while waiting for the response.



Appendix – Asynchronous API Pattern and Design Strategies

Designing Asynchronous APIs for Atomic Transactions

Ensuring atomicity with asynchronous APIs requires careful design to prevent data inconsistencies and maintain the reliability of any system. Here are key considerations:

1. Implement idempotency

- **Principle:** Idempotency ensures that performing the same operation multiple times has the same effect as performing it once.
- **Techniques:**
 - **Unique Identifiers:** Unique message IDs should be used and stored in a persistent store. If a message with the same ID is processed again, processing should be skipped.
 - **State Management:** The state of operations should be tracked, ensuring an operation is not executed twice.
 - **Idempotent APIs:** APIs should be designed such that repeated requests (e.g., using PUT for updates) produce the same outcome.

2. Employ messaging and task queues

- **Message Queues:** The sender should be decoupled from the consumer, and messages stored until they are processed, thus enhancing availability and resilience.
- **Task Queues:** Asynchronous tasks should be managed by receiving them with data, running them, and delivering results. This can guarantee completion and support various communication patterns such as request/reply and notifications.

3. Implement compensating transactions

- **Saga Pattern:** For complex distributed transactions, especially in microservices, the Saga pattern should be used. This involves a series of local transactions across services. If one fails, compensating transactions are triggered to undo, or reverse/roll back, prior changes.
- **Order of Compensation:** Compensating actions may not necessarily be the exact reverse of the original steps. Thus, it is necessary to prioritize the reversing of changes in data stores sensitive to inconsistencies.

4. Ensure proper error handling and recovery

- **Try-Catch:** Handle errors within async functions using try-catch blocks to prevent crashes.
- **.catch():** Handle errors by chaining a .catch() method directly to the function call.



- **Retry Mechanisms:** Implement retries for temporary failures, like network issues.
- **Timeouts:** An AbortController or similar mechanisms should be used to cancel requests that take too long to prevent an application from hanging or blocking.

5. Be mindful of eventual consistency

- **Asynchronous Data Replication:** In distributed systems, data replication can occur asynchronously, leading to eventual consistency. While this provides higher availability and scalability, it is crucial to acknowledge potential delays in data synchronization and design systems accordingly.

6. Additional best practices

- **Atomic Operations:** The appropriate atomic operation should be used for the specific use case.
- **Lock Scope:** The scope of synchronization should be minimized to avoid deadlocks.
- **Testing:** Thorough test planning should be created for race conditions and deadlocks.
- **Serialization:** A serializable isolation level for database transactions should be used to emulate serial execution, where each transaction runs one after another.

These strategies allow reliable and robust systems to be built that leverage the power of asynchronous APIs while maintaining the atomicity of transactions.

Examples of Saga Pattern Implementation with Asynchronous APIs

The EA End-to-End Customer Journey Based Architecture requires atomicity of transactions across many distributed systems and services. The Saga pattern is a method of managing distributed transactions in distributed systems and microservices, ensuring data consistency even when a single atomic transaction spanning multiple services is not possible.

This is achieved by breaking the transaction into a sequence of local transactions within each service.

Thus, if a local transaction fails, compensating transactions are triggered to reverse the completed steps. This approach is particularly well-suited for asynchronous APIs, as it aligns with the event-driven nature of distributed systems.

Examples of implementing the Saga pattern with asynchronous APIs are shown here focusing on the two main coordination approaches: choreography and orchestration:

1. Choreography-based Saga (event-driven)

- **Scenario:** Order processing in an e-commerce platform.
- **Approach:** Each service involved in the transaction publishes events to a message broker (e.g., Kafka, RabbitMQ). Other services subscribe to these events and react accordingly, triggering subsequent steps or compensating actions.



- **Example Steps:**
 1. **Order Service:** Receives a request to create an order. Creates the order in a PENDING state and publishes an OrderCreated event to the message broker.
 2. **Payment Service:** Subscribes to the OrderCreated event. Processes the payment (e.g., deducting from the customer's account). If successful, publishes a PaymentCompleted event. If it fails, publishes a PaymentFailed event.
 3. **Inventory Service:** Subscribes to the PaymentCompleted event. Reserves the items in the inventory. If successful, publishes a StockReserved event. If it fails due to insufficient stock, publishes an InventoryFailed event.
 4. **Shipping Service:** Subscribes to the StockReserved event. Schedules the shipment and publishes a ShippingScheduled event.
 5. **Notification Service:** Subscribes to various events (e.g., OrderApproved, OrderRejected, PaymentRefunded, ShippingScheduled) and sends notifications to the customer based on the outcome.
- **Compensating Actions (Asynchronous):**
 - If the Payment Service publishes a PaymentFailed event, the Inventory Service, listening to this event, might release the reserved items, and the Order Service might change the order status to CANCELED.
 - If the Inventory Service publishes an InventoryFailed event, the Payment Service, listening to this event, might issue a refund, and the Order Service might update the order status to CANCELED.

2. Orchestration-based Saga (centralized control)

- **Scenario:** Managing a credit card request process involving multiple services.
- **Approach:** A central orchestrator service (also known as a Saga Execution Coordinator or SEC) defines and manages the workflow, sending commands to participant services and reacting to their responses (events).
- **Example Steps:**
 - **Card Service:** Receives a new credit card request. Triggers the Orchestrator Service to initiate the saga.
 - **Orchestrator Service:**
 - i. Initiates parallel calls to the Verification Service and Identity Service (e.g., sending commands via message queues) to perform necessary checks.
 - ii. Waits for responses (events) from both services.
 - iii. If both services respond successfully, triggers the Calculation Service to compute the credit limit.



- iv. If any step fails, triggers compensating actions by sending commands to the services involved.
- **Verification Service:** Performs verification checks and sends a response (event) back to the orchestrator.
 - **Identity Service:** Verifies the user's identity and sends a response (event) back to the orchestrator.
 - **Calculation Service:** Calculates the credit limit and sends a response (event) back to the orchestrator.
- **Compensating Actions (Orchestrated):**
 - If the Verification Service fails. The Orchestrator cancels the Credit Card request and informs the Card Service.
 - The orchestrator uses its central knowledge of the workflow and the state of each local transaction to trigger appropriate rollback actions in the correct sequence.

Key considerations for asynchronous saga implementation include:

- **Reliable messaging:** Use a robust message broker to ensure message delivery and handle retries.
- **Idempotency:** Design services and compensating transactions to be idempotent to handle duplicate messages.
- **Error Handling and rollback:** Implement clear error handling strategies and define compensating transactions for each local transaction.
- **Observability:** Implement monitoring and logging to track the progress of sagas and debug failures.
- **Correlation IDs:** Use correlation IDs to link related events across services within a saga.

