**What is Idempotency** (অদক্ষতা) **in Distributed Systems?**

Imagine you're making a **purchase** from an online store.

You hit **"pay"** but the screen freezes, and you're unsure if the payment went through.

So, you **refresh** the page and **try again**.

Behind the scenes, how does the system ensure you aren’t accidentally **charged twice**?

[[A diagram of a service

AI-generated content may be incorrect.](https://substackcdn.com/image/fetch/$s_!yjks!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2F538d3bda-046f-48f0-b62d-730327bf3bd7_954x696.png)](https://substackcdn.com/image/fetch/$s_!yjks!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2F538d3bda-046f-48f0-b62d-730327bf3bd7_954x696.png" \t "_blank)

This scenario highlights a common problem in distributed systems: **handling repeated operations gracefully** (সুন্দরভাবে)**.**

The solution to this problem lies in the concept of **idempotency.**

In this blog, we'll explore what idempotency is, why it matters, how to implement it, challenges, considerations and best practices to ensure robust and reliable systems.

**What is Idempotency?**

In mathematics, an operation is idempotent if applying it multiple times produces the same result as applying it once.

For example, the absolute value function is idempotent: ||-5|| = |-5| = 5.

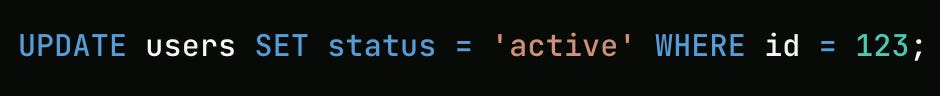
**Idempotency** is a property of certain operations whereby executing the same operation multiple times produces the same result as executing it once.

For example: If a request to delete an item is idempotent—all requests after the first will have no impact.

In programming, setting a value is idempotent, while incrementing a value is not.

Idempotent: user.status = 'active' Not Idempotent: user.login\_count += 1

Some operations are naturally idempotent.

[[](https://substackcdn.com/image/fetch/$s_!S0ki!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2Ff6358570-6fec-4c8c-bc26-32071450a939_940x96.png)](https://substackcdn.com/image/fetch/$s_!S0ki!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2Ff6358570-6fec-4c8c-bc26-32071450a939_940x96.png" \t "_blank)

No matter how many times you run this, the result remains the same.

**Why Idempotency Matters**

Distributed systems often require **fault tolerance** to ensure high availability. When a network issue causes a **timeout** or an **error**, the client might **retry** the request.

[[A computer server diagram with words

AI-generated content may be incorrect.](https://substackcdn.com/image/fetch/$s_!czYH!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2F637607fe-3d2c-49db-be80-3d247bea6ab8_1212x394.png)](https://substackcdn.com/image/fetch/$s_!czYH!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2F637607fe-3d2c-49db-be80-3d247bea6ab8_1212x394.png" \t "_blank)

If the system handles retries without idempotency, every retry could change the system’s state unpredictably.

By designing operations to be idempotent, engineers create a buffer against unexpected behaviors caused by retries.

This “safety net” prevents repeated attempts from distorting the outcome, ensuring stability and reliability.

**Strategies to Implement Idempotency**

**1. Unique Request Identifiers**

One of the simplest techniques to achieve idempotency is by attaching a **unique identifier**, often called an **idempotency key** to each request.

When a client makes a request, it generates a **unique ID** that the server uses to track the request. If the server receives a request with the same ID later, it knows it’s a duplicate and discards it.

**Example**: A payment service could require every transaction request to include a unique ID. If the client retries with the same ID, the server will skip the charge, preventing duplicate transactions.

**Code Example:**

[[A computer screen with many colorful text

AI-generated content may be incorrect.](https://substackcdn.com/image/fetch/$s_!wVHX!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2Fa2db9023-fa9a-4d77-9941-a9eaa0b19e92_1698x1500.png)](https://substackcdn.com/image/fetch/$s_!wVHX!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2Fa2db9023-fa9a-4d77-9941-a9eaa0b19e92_1698x1500.png" \t "_blank)

In this example, each request includes a unique request\_id stored in the database to track processed requests and prevent duplicates.

**2. Database Design Adjustments (Upsert Operation)**

Some database operations, such as inserting the same record multiple times, can lead to unintended duplicate entries.

Achieving idempotency in these cases often requires redesigning the database operations to be inherently idempotent.

This can involve using upsert operations (which updates a record if it exists or inserts it otherwise) or applying **unique constraints** that prevent duplicates from being added in the first place.

In this example, we use SQL INSERT ... ON CONFLICT to achieve an upsert operation, ensuring that duplicate entries don’t affect the database state.

[[A screen shot of a computer program

AI-generated content may be incorrect.](https://substackcdn.com/image/fetch/$s_!2O8p!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2F2283e98b-4bb9-4785-a247-60b571ffe43c_876x228.png)](https://substackcdn.com/image/fetch/$s_!2O8p!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2F2283e98b-4bb9-4785-a247-60b571ffe43c_876x228.png" \t "_blank)

This SQL statement inserts a new item if it doesn't exist. If it does exist (conflict on item\_id), it updates the stock by adding the new stock quantity, ensuring the operation remains idempotent.

**3. Idempotency in Messaging Systems**

In a messaging system, we can enforce idempotency by storing a log of processed message IDs and checking against it for every incoming message.

[[A computer screen with text

AI-generated content may be incorrect.](https://substackcdn.com/image/fetch/$s_!tJRt!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2F3a1e1402-6368-424e-b3e7-f8be7247a786_1372x878.png)](https://substackcdn.com/image/fetch/$s_!tJRt!,f_auto,q_auto:good,fl_progressive:steep/https%3A%2F%2Fsubstack-post-media.s3.amazonaws.com%2Fpublic%2Fimages%2F3a1e1402-6368-424e-b3e7-f8be7247a786_1372x878.png" \t "_blank)

Each message has a unique messageId. Before processing, we check if the messageId is already in processedMessages. If it is, the message is ignored; otherwise, it’s processed and added to the set to avoid duplicates.

**4. Idempotency in HTTP Methods**

HTTP defines several methods (verbs) for different types of requests.

These methods can be categorized by whether they are idempotent or non-idempotent, influencing how a system handles retries and preventing unintended side effects.

**Idempotent Methods:**

* **GET:** Retrieves data from a resource. GET requests are inherently idempotent because they only read data and do not alter the server’s state.
  + **Example:** Accessing a blog post by making a GET request to /posts/123 will simply retrieve that post, without modifying any server data. Whether you retrieve it once or a thousand times, the post remains unchanged.
* **PUT:** Update or completely replace an existing resource. PUT requests are idempotent because the final state is the same whether the PUT request is executed once or multiple times.
  + **Example:** Updating user information by making a PUT request to /users/45 with updated user details will overwrite the user’s data with the new information provided. Executing the same PUT request repeatedly results in the same final user data on the server.
* **DELETE:** Removes a resource from the server. DELETE requests are idempotent because deleting a resource that’s already been deleted has no further effect.
  + **Example:**Deleting an item by making a DELETE request to /items/678 will remove the item. If you attempt the DELETE request again, it will have no effect since the item no longer exists.

**Non-Idempotent Methods:**

* **POST:** Creates a new resource on the server. POST requests are non-idempotent because each request usually results in the creation of a new resource.
  + **Example:**Creating a new order by making a POST request to /orders with order details will generate a new order each time the request is made.

**Challenges and Considerations**

While idempotency is powerful, it comes with its own set of challenges:

1. **Performance Overhead**: Storing idempotency keys or checking for duplicate operations can add overhead and increase the overall latency.
2. **State Management**: Idempotency often requires maintaining state, which can be challenging in stateless architectures.
3. **Distributed Systems**: Ensuring idempotency across distributed systems can be challenging and may require **distributed locking** or **consensus algorithms**.
4. **Time Window**: How long should idempotency guarantees be maintained? Forever, or for a limited time?
5. **Database Constraints**: Not all operations are idempotent by default; unique constraints or upsert logic may be necessary to avoid duplication.

**Best Practices**

When implementing idempotency in your system, consider these best practices:

1. **Use Unique Identifiers**: Attach a unique ID (idempotency key) to each request to track and prevent duplicate processing.
2. **Design for Idempotency from the Start**: It's much easier to design for idempotency from the beginning than to add it later.
3. **Implement Retry with Backoff**: When retrying idempotent operations, use an exponential backoff strategy to avoid overwhelming the system.
4. **Employ Idempotent HTTP Methods**: Prefer idempotent methods (GET, PUT, DELETE) for operations that may be retried; design POST with unique identifiers if idempotency is required.
5. **Document Idempotent Operations**: Clearly document which operations are idempotent in your API specifications.
6. **Test Thoroughly**: Implement tests that verify the idempotency of your operations, including edge cases and failure scenarios.
7. **Use Locks or Versioning**: Use locks, optimistic concurrency control, or version numbers to manage simultaneous requests safely.

Idempotency is a powerful concept in distributed systems that can greatly enhance the reliability and fault-tolerance of your systems.

Whether you're designing a distributed database, a payment processing system, or a simple web API, considering idempotency in your design can save you (and your users) from many headaches down the road.

**Basic Questions**

1. **What is idempotency in computing?**  
   **Answer:**  
   Idempotency means an operation can be applied multiple times without changing the result beyond the initial application.  
   Example: Deleting a record (DELETE) is idempotent — deleting it again has no further effect.
2. **Why is idempotency important in APIs?**  
   **Answer:**  
   It ensures that retrying the same request (due to failures, network errors, or client retries) doesn’t cause unintended side effects, such as double charges or duplicate data creation.
3. **Which HTTP methods are considered idempotent according to REST standards?**  
   **Answer:**  
   **GET, PUT, DELETE, HEAD, OPTIONS, TRACE** are idempotent. **POST** is generally not idempotent.
4. **Is a GET request always idempotent?**  
   **Answer:**  
   By convention, yes — GET should only fetch data and not modify server state. However, if implemented incorrectly (e.g., logging visits in DB), it may break idempotency.
5. **What is the difference between idempotent and safe HTTP methods?**  
   **Answer:**
   * **Safe** methods (like GET, HEAD) don’t change server state.
   * **Idempotent** methods may change state but will have the same effect regardless of how many times they’re called.

**Intermediate Questions**

1. **How does idempotency help in payment systems?**  
   **Answer:**  
   It prevents charging a customer multiple times if a payment request is retried due to network errors or client retries. A unique idempotency key is used to identify repeated requests.
2. **Can a POST request be idempotent?**  
   **Answer:**  
   Normally no, but it can be designed to be idempotent by using **idempotency keys** to detect and ignore duplicate requests.
3. **What is an idempotency key?**  
   **Answer:**  
   A unique identifier (e.g., UUID) sent by the client with a request, so the server can detect if the same operation was already processed and return the same result without redoing it.
4. **What is the difference between idempotency and concurrency control?**  
   **Answer:**
   * **Idempotency** ensures multiple identical requests have the same effect.
   * **Concurrency control** manages access to resources so simultaneous different requests don’t cause conflicts.
5. **If a DELETE request returns 404 on the second call, is it still idempotent?**  
   **Answer:**  
   Yes — because the state of the resource remains the same (non-existent) after both calls. The response code may change, but the final state is unchanged.

**Advanced Questions**

1. **How would you implement idempotency in a REST API that creates resources?**  
   **Answer:**
   * Client generates an **idempotency key**.
   * Server stores a mapping of the key to the result.
   * On repeated requests with the same key, return the stored result instead of creating a new resource.
2. **What challenges arise when implementing idempotency in distributed systems?**  
   **Answer:**
   * Synchronizing idempotency keys across nodes.
   * Handling partial failures where the request reaches one node but not others.
   * Expiring stored keys to prevent memory bloat while avoiding premature deletion.
3. **Is database UPSERT an idempotent operation?**  
   **Answer:**  
   Yes — because running the same UPSERT with the same data multiple times will leave the database in the same final state.
4. **How does idempotency differ from eventual consistency?**  
   **Answer:**
   * **Idempotency:** Ensures repeated identical requests produce the same result.
   * **Eventual consistency:** Ensures that given enough time, all replicas converge to the same final state, but repeated requests might still temporarily give different results.
5. **In a retry mechanism, how can you make sure idempotency does not hide genuine errors?**  
   **Answer:**  
   Log retries separately, monitor for unusual retry patterns, and return proper error codes for permanent failures rather than masking them as successful idempotent responses.

**Scenario-Based Idempotency Questions & Answers**

1. **Payment Gateway Retry**  
   A customer clicks "Pay Now" and the request goes through, but the network drops before they get confirmation. They click again.
   * **Question:** How do you prevent double charging?
   * **Answer:** Use an **idempotency key** for each payment attempt. The server checks if a request with the same key is already processed and returns the same confirmation.
2. **Booking System Double Submission**  
   A user books a hotel room, but their browser freezes and they hit refresh.
   * **Question:** How to ensure only one booking is made?
   * **Answer:** Use an idempotent **reservation endpoint** with a client-generated booking ID so retries don’t create duplicates.
3. **DELETE Request Twice**  
   An API receives DELETE /users/123. It’s processed, but the request is resent accidentally.
   * **Question:** Is this still idempotent?
   * **Answer:** Yes — because the user remains deleted after both calls, even if the second returns 404.
4. **Inventory Update Race Condition**  
   An API receives PUT /inventory/sku123 { quantity: 50 } twice due to retries.
   * **Question:** Will this cause issues?
   * **Answer:** No — PUT is idempotent, so setting the quantity to 50 multiple times leaves the inventory in the same state.
5. **Password Reset Link**  
   A user clicks the password reset link multiple times.
   * **Question:** How to make this idempotent?
   * **Answer:** Ensure the link updates the password once and returns the same success response if clicked again before expiration.
6. **Distributed Microservices Call**  
   A payment service calls a ledger service to record a transaction, but retries happen because of timeout.
   * **Question:** How to prevent duplicate ledger entries?
   * **Answer:** Pass an **idempotency key** through service calls so each downstream service processes it only once.
7. **Order Creation with POST**  
   Normally, POST is not idempotent. But your business requires order creation to be retry-safe.
   * **Question:** How to make it idempotent?
   * **Answer:** Accept an Order-ID from the client and store results keyed by it. If the same ID is sent again, return the original order.
8. **Refund Processing**  
   A refund is triggered twice due to a system bug.
   * **Question:** How to avoid refunding twice?
   * **Answer:** Implement a refund ID check — once processed, further requests with the same refund ID return the stored result.
9. **Email Sending via API**  
   Your API sends an email when called, but the client retries if no response is received.
   * **Question:** Is this idempotent?
   * **Answer:** No — unless the API tracks email IDs sent and suppresses duplicates.
10. **IoT Device Command**  
    A command turn\_off\_light is sent twice to a smart device.
    * **Question:** Is this idempotent?
    * **Answer:** Yes — both calls leave the light turned off.
11. **File Upload with Retry**  
    A file upload API receives the same file twice due to retry logic.
    * **Question:** How to make uploads idempotent?
    * **Answer:** Use file hash as the idempotency key so duplicates are ignored.
12. **Account Activation Link**  
    A user clicks the activation link multiple times.
    * **Question:** How to ensure consistent behavior?
    * **Answer:** The link marks the account active on first click, and subsequent clicks return the same confirmation.
13. **Database UPSERT in API**  
    The API updates a user’s profile. If the user does not exist, it creates one.
    * **Question:** Is this idempotent?
    * **Answer:** Yes — if the same data is sent again, the result is unchanged.
14. **Logging API**  
    A logging endpoint records each call in the database.
    * **Question:** Is this idempotent?
    * **Answer:** No — each call appends a new record. It can be made idempotent by deduplicating logs based on event ID.
15. **SMS Sending API**  
    A service sends SMS notifications but retries if acknowledgment is delayed.
    * **Question:** How to prevent sending multiple SMS for the same event?
    * **Answer:** Track SMS event IDs and only send once per unique event.