- Idempotency
  - Scenario
    - Network problems or repeated operations
    - Third party callbacks
  - Http idempotency
    - Safe vs Idempotent Methods
    - Why PUT Idempotent and PATCH not
    - Why DELETE is Idempotent
  - Implementation
    - Within standalone applications
    - Within DB layer of distributed applications
      - CRUD operations
        - Create/Insert
        - Read/Select
        - Update
        - Delete
      - Avoid replica databases
      - Unique constraint within DB

Idempotency

Scenario

Network problems or repeated operations

- For example, the point praise function, a user can only point praise once for the same piece of article, repeated point praise prompt has already point praise.

Third party callbacks

- Our system often needs to deal with third party systems, such as WeChat recharge and Alipay recharge, WeChat and Alipay will often notify you to pay the result by callback your interface. In order to ensure that you receive callbacks, it is often possible to make multiple callbacks.

Http idempotency

Safe vs Idempotent Methods

- Safe methods: HTTP methods that do not modify resources.
- Idempotent methods: HTTP methods that can be called many times

without different outcomes.

HTTP METHOD	USE CASE	IDEMPOTENCE	SAFETY
GET	Get resources, no side effect	YES	YES
HEAD	Same as GET except no response body	YES	YES
OPTIONS	Used to get the HTTP Methods supported by the URL	YES	YES
TRACE	performs a message loop-back test to the target resource	YES	YES
POST	Create new resources. e.g. POST http://www.forum.com/articles	NO	NO
PATCH	Partially update a resource	NO	NO
PUT	Completely update a resource	YES	NO
DELETE	Delete a resource	YES	NO

Why PUT Idempotent and PATCH not

- It's because it matters how you apply your changes. If you'd like to change the name property of a resource, you might send something like {"name": "foo"} as a payload and that would indeed be idempotent since executing this request any number of times would yield the same result: The resources name attribute is now "foo".
- But PATCH is much more general in how you can change a resource (check this definition on how to apply a JSON patch). It could also, for example, mean to move the resource and would look something like this: { "op": "move", "from": "/a/b/c", "path": "/a/b/d" }. This operation is obviously not idempotent since calling at a second time would result in an error.

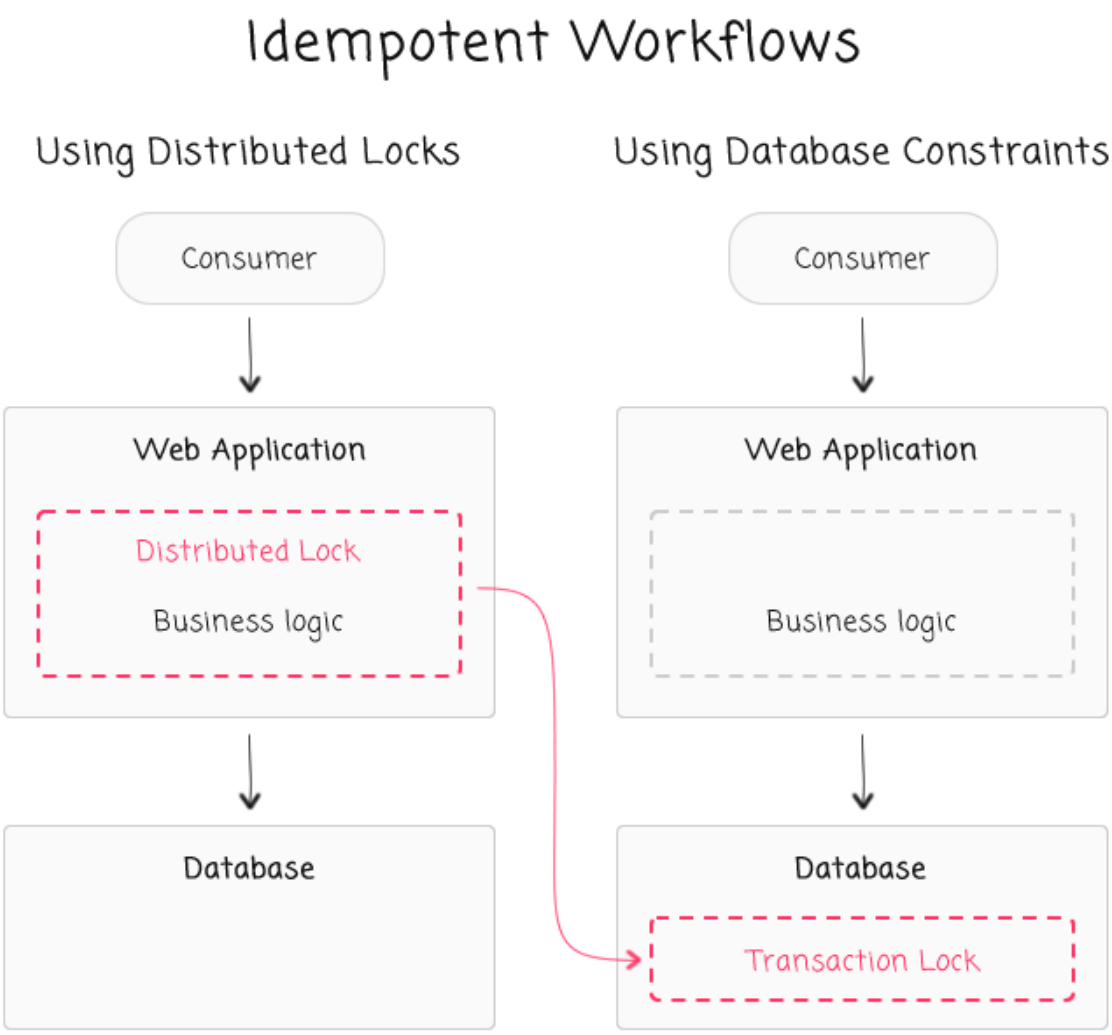
- So while most PATCH operations might be idempotent, there are some that aren't.

Why DELETE is Idempotent

- "Methods can also have the property of "idempotence" in that (aside from error or expiration issues) the side-effects of N > 0 identical requests is the same as for a single request. The methods GET, HEAD, PUT and DELETE share this property. Also, the methods OPTIONS and TRACE SHOULD NOT have side effects, and so are inherently idempotent. "
- The key bit there is the side-effects of N > 0 identical requests is the same as for a single request.
- You would be correct to expect that the status code would be different but this does not affect the core concept of idempotency - you can send the request more than once without additional changes to the state of the server.

Implementation

- Idempotency could be implemented in different layers of the service architecture.



Within standalone applications

Within DB layer of distributed applications

- Scenario: Operation retry

CRUD operations

Create/Insert

- Example: Insert user values (uid, name, age, sex, ts) where uid is the primary key
- Needs a little work to guarantee idempotence: If the primary key is generated using DB auto-increment id, then it is not idempotent. The primary key should rely on id which is related with business logic.

Read/Select

- Idempotent

Update

- Example (Update to absolute value): Update user set age = 18 where uid = 58.
  - Suffers from ABA problem in multi-thread environment
    a. current age = 17
    b. operation A: set age = 18
    c. operation B: set age = 19
    d. operation A: set age = 18
  - Needs optimistic concurrency control (version number) to guarantee idempotence
    a. current age = 17
    b. operation A: set age = 19, v++ where v = 1;
    c. Operation B: set age = 18, v++ where v = 1;
- Example (Update to relative value): Update user set age++ where uid = 58
  - Convert to absolute example

Delete

- Idempotent

Avoid replica databases

- See the section within reference: https://medium.com/airbnb-engineering/avoiding-double-payments-in-a-distributed-payments-system-2981f6b070bb

Unique constraint within DB

- Reference: https://www.bennadel.com/blog/3390-considering-strategies-for-idempotency-without-distributed-locking-with-ben-darfler.htm

Within business layer of distributed applications

Distributed lock

- Scenario: Request only once within a short time window. e.g. User click accidently twice on the order button.
- Please see Distributed lock

Generate the idempotency key

Where

- Layers of architecture: App => Nginx => Network gateway => Business logic => Data access layer => DB / Cache
- Idempotency considerations should reside within data access layer, where CRUD operations happen.
  - The idempotency key could not be generated within app layer due to security reasons
  - The process is similar to OAuth
    a. Step1: App layer generates a code
    b. Step2: App talks to business layer with the generated code to get an idempotency key
    c. Step3: The generated idempotency keys are all stored within an external data store. Busines layer check the external data store with mapping from code => idempotency key
      - If exist, directly return the idempotency key
      - Otherwise, generate a new idempotency key, store it within external store and return the generated idempotency key.
  - An optimization on the process above: Don't always need to check the external data store because repeated requests are minorities. Could rely on DB optimistic concurrency control for it instead of checking every time. For example, below queries will only be executed when there is no conflicts.
    a. insert into ... values ... on DUPLICATE KEY UPDATE ...
    b. update table set status = “paid” where id = xxx and status = “unpaid”;

How

- Request level idempotency: A random and unique key should be chosen

from the client in order to ensure idempotency for the entire entity collection level. For example, if we wanted to allow multiple, different payments for a reservation booking (such as Pay Less Upfront), we just need to make sure the idempotency keys are different. UUID is a good example format to use for this.

- Entity level idempotency: Say we want to ensure that a given $10 payment with ID 1234 would only be refunded $5 once, since we can technically make $5 refund requests twice. We would then want to use a deterministic idempotency key based on the entity model to ensure entity-level idempotency. An example format would be “payment-1234-refund”. Every refund request for a unique payment would consequently be idempotent at the entity-level (Payment 1234).

Snowflake

- https://github.com/twitter-archive/snowflake/tree/snowflake-2010
- Upside:
  i. 64-bit unique IDs
- Downside:
  i. Introduce another component in our infrastructure that needs to be maintained.
  ii. Limitation: when time is reset/rolled back, duplicated id will be generated.

MongoDB's Object Id

Database ticket servers

Leaf

Wechat seqsvr

- https://www.infoq.cn/article/wechat-serial-number-generator-architecture/

Design by yourself

- The IDs generated by this sequence generator are composed of -
  - Epoch timestamp in milliseconds precision - 42 bits. The maximum

timestamp that can be represented using 42 bits is 242 - 1, or 4398046511103, which comes out to be Wednesday, May 15, 2109 7:35:11.103 AM. That gives us 139 years with respect to a custom epoch.
    - Node ID - 10 bits. This gives us 1024 nodes/machines.
    - Local counter per machine - 12 bits. The counter's max value would be 4095.

- References: https://www.callicoder.com/distributed-unique-id-sequence-number-generator/