Design a reservation system

Functional requirements

1. The system should support multiple hotels and offer features like viewing, updating, adding, and deleting hotel information.
2. Room assigned durin gcheck in here

**Non-Functional Requirements**

1. A cache can store available room counts to prevent double booking, with updates made through an event queue mechanism.
2. Images can be stored in S3, and a geographically located CDN can retrieve them, saving space and bandwidth.
3. Concurrency issues can be addressed by optimizing ways to control additions to the reservation table.

Non-funcational

1. Scalability
2. Latency : view hotel can be 150 ms very fast
3. Security: IAM / Ddos /Rate limiter here

A screenshot of a computer

Description automatically generated

Estimatino write this down

What does the API look like for this?

**API design**

We explore the API design for the hotel reservation system. The most important APIs are listed below using the RESTful conventions.

Note that this chapter focuses on the design of a hotel reservation system. For a complete hotel website, the design needs to provide intuitive features for customers to search for rooms based on a large array of criteria. The APIs for these search features, while important, are not technically challenging. They are out of scope for this chapter.

**Hotel-related APIs**

| **API** | **Detail** |
| --- | --- |
| GET /v1/hotels/ID | Get detailed information about a hotel. |
| POST /v1/hotels | Add a new hotel. This API is only available to hotel staff. |
| PUT /v1/hotels/ID | Update hotel information. This API is only available to hotel staff. |
| DELETE /v1/hotels/ID | Delete a hotel. This API is only available to hotel staff. |

Table 1 Hotel-related APIs

**Room-related APIs**

| **API** | **Detail** |
| --- | --- |
| GET /v1/hotels/ID/rooms/ID | Get detailed information about a room. |
| POST /v1/hotels/ID/rooms | Add a room. This API is only available to hotel staff. |
| PUT /v1/hotels/ID/rooms/ID | Update room information. This API is only available to hotel staff. |
| DELETE /v1/hotels/ID/rooms/ID | Delete a room. This API is only available to hotel staff. |

Table 2 Room-related APIs

**Reservation related APIs**

| **API** | **Detail** |
| --- | --- |
| GET /v1/reservations | Get the reservation history of the logged-in user. |
| GET /v1/reservations/ID | Get detailed information about a reservation. |
| POST /v1/reservations | Make a new reservation. |
| DELETE /v1/reservations/ID | Cancel a reservation. |

Table 3 Reservation-related APIs

The endpoint

Making a new reservation is a very important feature. The request parameters of making a new reservation (POST /v1/reservations) could look like this.

{

"startDate":"2021-04-28",

"endDate":"2021-04-30",

"hotelID":"245",

"roomTypeID":"12354673389",

"roomCount":"3",

"reservationID":"13422445"

}

Please note *reservationID* is used as the idempotency key to prevent double booking. Double booking means multiple reservations are made for the same room on the same day. The details are explained in “Concurrency issue” in the ‘Deep Dive’ section.

**What about database deisgn here?**

What are the questions here?

Should reversation and inventory have the same database?

Companies have different database for each service here

**How to make payment side scalable? (This is very important here)**

1. Event q listening to payment events from payment processor for each topic
2. Each topic can be payment\_failed, payment\_succeeded and all that

A diagram of a server

Description automatically generated

A group of black and white squares

Description automatically generated with medium confidence

What table do we look at to figure out inventory issue like how many rooms are left?

Use room\_type\_inventory table

**What happens when user books a room? What sql query takes places**

1. We select rows within a date range

SELECT date, total\_inventory, total\_reserved

FROM room\_type\_inventory

WHERE room\_type\_id = ${roomTypeId} AND hotel\_id = ${hotelId}

AND date between ${startDate} and ${endDate}

And the query returns sth like this:

| **date** | **total\_inventory** | **total\_reserved** |
| --- | --- | --- |
| 2021-07-01 | 100 | 97 |
| 2021-07-02 | 100 | 96 |
| 2021-07-03 | 100 | 95 |

And then next

if (total\_reserved + ${numberOfRoomsToReserve}) <= total\_inventory

update r*oom\_type\_inventory* total\_reserviced = total\_reserved + ${numberOfRoomsToReserve})

else:

fail the transaction

**How to prevent the same user from double booking here?**

1. We can pass an idempotent key in the reservation API request. An API call is idempotent if it produces the same result no matter how many times it is called. F
2. We can use the reservation id or we can use sth else as well that’s ok too.
3. And then we can use the unique constraint in database to make sure no double bookig happens here and this will be important here
4. Reservation id here is the number 1 thing here and number 2 thing is what we think of this stragtegy over time

Pessmisitic locking here

1. prevents simultaneous updates by placing a lock on a record as soon as one user starts to update it, and this is very useful here
2. This approach is not scalable. If a transaction is locked for too long, other transactions cannot access the resource. This has a significant impact on database performance, especially when transactions are long-lived or involve a lot of entities.

Optimistic locking:

1. What are the 2 methods that’s needed here (version number and time stamp)?

Using versino number and time stamp?

Version number is generally considered to be a better option because the server clock can be inaccurate over time. We explain how optimistic locking works with version number.

What happens when using version?

1. A new column called “version” is added to the database table.
2. Before a user modifies a database row, the application reads the version number of the row.
3. When the user updates the row, the application increases the version number by 1 and writes it back to the database.
4. A database validation check is put in place; the next version number should exceed the current version number by 1. The transaction aborts if the validation fails and the user tries again from step 2.

Pros and cons here?

Pros:

* It prevents applications from editing stale data.
* We don’t need to lock the database resource. There's actually no locking from the database point of view. It's entirely up to the application to handle the logic with the version number.
* Optimistic locking is generally used when the data contention is low. When conflicts are rare, transactions can complete without the expense of managing locks.

Cons:

* Performance is poor when data contention is heavy.

Option 3:

Using database constraints:

CONSTRAINT `check\_room\_count` CHECK((`total\_inventory - total\_reserved` >= 0))

Using the same example as shown in Figure 14, when user 2 tries to reserve a room, *total\_reserved* becomes 101, which violates the *total\_inventory (100) - total\_reserved (101) >= 0* constraint. The transaction is then rolled back.

Cons: can result in high number of failures when the data contention is heavy here

How to improve scalability here

We can just add more servers and scale horizontally. How would we shard this?

Shard this by hotel id?

A diagram of a diagram

Description automatically generated

**How would we implement caching here?**

Can use redis (use TTL or LRU) for caching things here

A diagram of a server

Description automatically generated

What are key and values used in cache?

Inventory cache: all inventory management query operations are moved to the inventory cache (Redis) and we need to pre-populate inventory data to the cache. The cache is a key-value store with the following structure:

key: hotelID\_roomTypeID\_{date}

value: the number of available rooms for the given hotel ID, room type ID and date.

What’s flow for updating the inventory table?

* 1. It will update the inventory db first, and then async update the cache table here
  2. And then when using this this is quite interesting here

**How do we make sure both inventory and reservation tables are in sync in microservices?**

However, in a microservice architecture, each service has its own database. One logically atomic operation can span multiple services. This means we cannot use a single transaction to ensure data consistency. As shown in Figure 19, if the update operation fails in the reservation database, we need to roll back the reserved room count in the inventory database. Generally, there is only one happy path, but many failure cases that could cause data inconsistency.

1. Using saga pattern or 2-phase commit here

**How are images stored?**

Using CDN and S3

A screenshot of a computer

Description automatically generated