### **Problem Understanding:**

1. Scale of the system -> hotel chain of 5000 hotels and 1 million rooms in total.

2. Do customers pay when they make reservations or when they arrive at a hotel?

-> They pay in full at reservation

3. Do customers book hotel rooms through the hotel's website only? Or do we need to support phone call reservations as well?

-> through a website or app.

4. Can customers cancel their reservations?

-> yes

5. Are there any other things to consider?

-> We allow 10% overbooking.

6. Since we have limited time, I assume the hotel room search is not in scope.

-> We focus on following features:

i. Show hotel-related page

ii. Show hotel room-related detail page

iii. Reserve a room

iv. Admin panel to add/remove/update hotel or room info.

v. Support the overbooking feature.

### **Non-functional Requirements:**

1. high concurrency: During peak season or big events, some popular hotels may have a lot of customers trying the same room.

2. Moderate latency: Ideal to have fast response while reservations. But it's acceptable to take few seconds to process reservation request

### **Back-of-the-envelope estimation:**

1. Assuming 70% rooms are occupied and average duration is 3 days

2. Estimated daily reservations: 10^6 \* 0.7 / 3 = 233,333 (240 thousand approx)

3. Reservation per second= 2.40 \*10^5 / 10^5 seconds in a day = 3 approx. So average TPS is not high

View hotel/room details QPS: 300

Room booking page QPS: 30

Room reservation QPS: 3

**High Level Design and Get Buy-in:**

Hotel related APIs:

GET /v1/hotels/ID : Get detailed information about a hotel

For hotel staff usage:

POST /v1/hotels

PUT /v1/hotels/ID

DELETE /v1/hotels/ID

Room-related APIs:

GET /v1/hotels/{hotel\_id}/rooms/{rooms\_id} : Get a detailed information about a room

For hotel staff usage:

POST /v1/hotels/{hotel\_id}/rooms/{rooms\_id}

PUT /v1/hotels/{hotel\_id}/rooms/{rooms\_id}

DELETE /v1/hotels/{hotel\_id}/rooms/{rooms\_id}

Reservation related APIs:

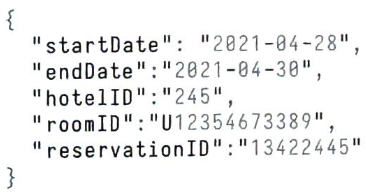
GET /v1/reservations?custID={custID}&startCount={startCount}&endCount={endCount} : get all reservation history of loggedIn user in a pagination way

GET /v1/reservations/ID : Get detailed information about a reservation

POST /v1/reservations : Makes a new reservation

DELETE /v1/reservations/ID : Cancel Reservation

POST /v1/reservations



ReservationID is used as the idempotency key to prevent double booking and handling concurrency.

### **High Level Design and Get Buy-in:**

Query 1: View detailed info about a hotel.

Query 2: Find available types of rooms given a date range.

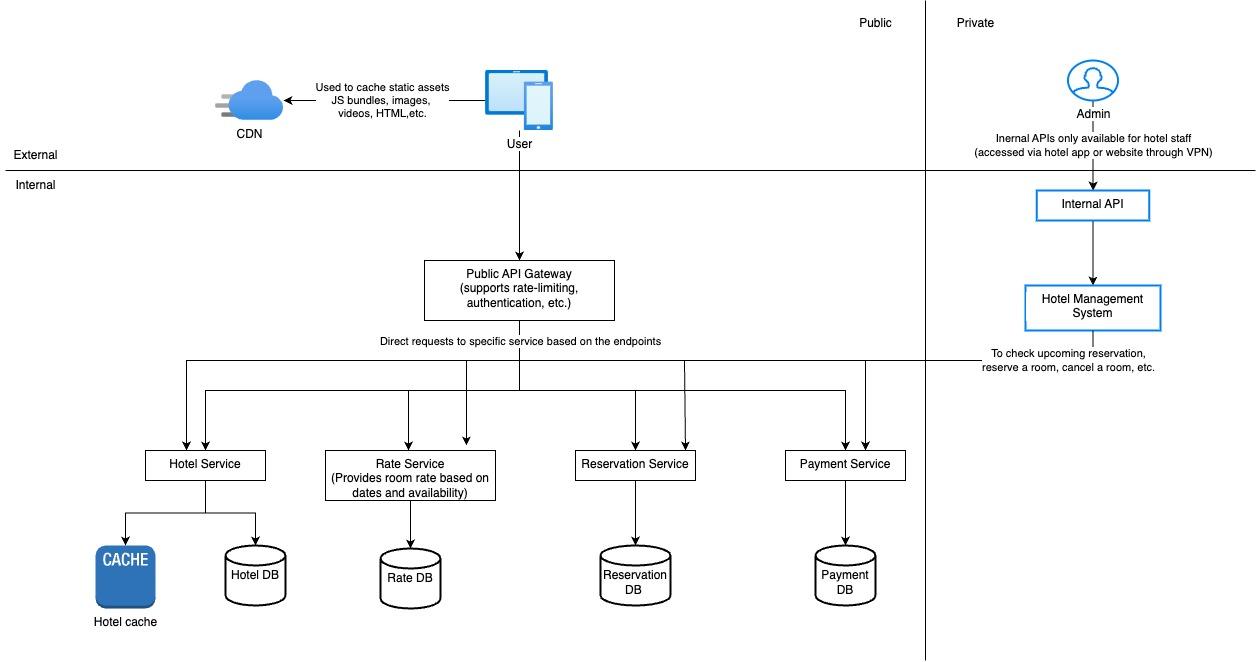
Query 3: Record a reservation.

Query 4: Lookup a reservation or past history of reservations.

Scale of the system is not large but we need to be prepared for traffic surges during bid events. With these requirements in mind, we choose a relational database because:

1. RDMS(SQL) works well with a read-heavy workflow. NoSQL is generally optimized for writing heavy workflows.
2. RDMS(SQL) follows ACID.
3. RDMS(SQL) can easily model the data. The structure of business data is very clear and the relationship between different entities(hotel, room, room\_type, etc) is stable.

The case of hotel reservation is a bit different compared to movie ticket reservation, travel ticket reservation and other reservations. In hotel reservation a user can actually reserve a type(standard room, king-size room, queen size room, etc.) of room in a given hotel instead of specific room. Room numbers are given when the guest checks in and not at the time of the reservation.



For the reservation API, roomID is replaced by roomTypeID in the request parameter. The API to make a reservation looks like this:  
POST /v1/reservations

{

“startDate”: “2021-04-28”,

“endDate”:”2021-04-30”,

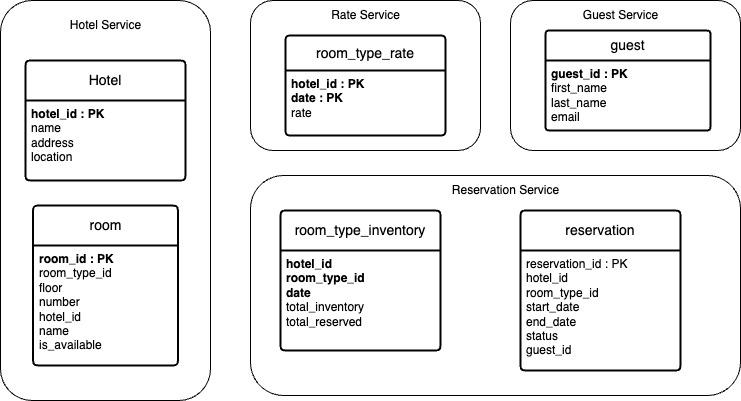
“hotelID”:”245”;

“roomTypeId”:”123456789”,

“reservationId”:”13422445”  
}

### **Design Deep Dive:**

#### **Improved data Model**



Room\_type\_inventory: stores inventory data about hotel rooms. This table is very important for the reservation system.

(hotel\_id, room\_type\_id, date) is composite primary key. The rows of the table are pre-populated by querying the inventory data across all future dates within 2 years. We have scheduled a daily job that pre populates inventory data when the dates advance further.

As mentioned in the back-of-the-envelope estimation, we have 5,000 hotels. Assume each hotel has 20 types of rooms. That's (5,000 hotels ×20 types of rooms x2 years x365 days) = 73 million rows. 73 million is not a lot of data and a single database is enough to store the data. However, a single server means a single point of failure. to achieve high availability, we could set up database replications across multiple regions or availability zones.

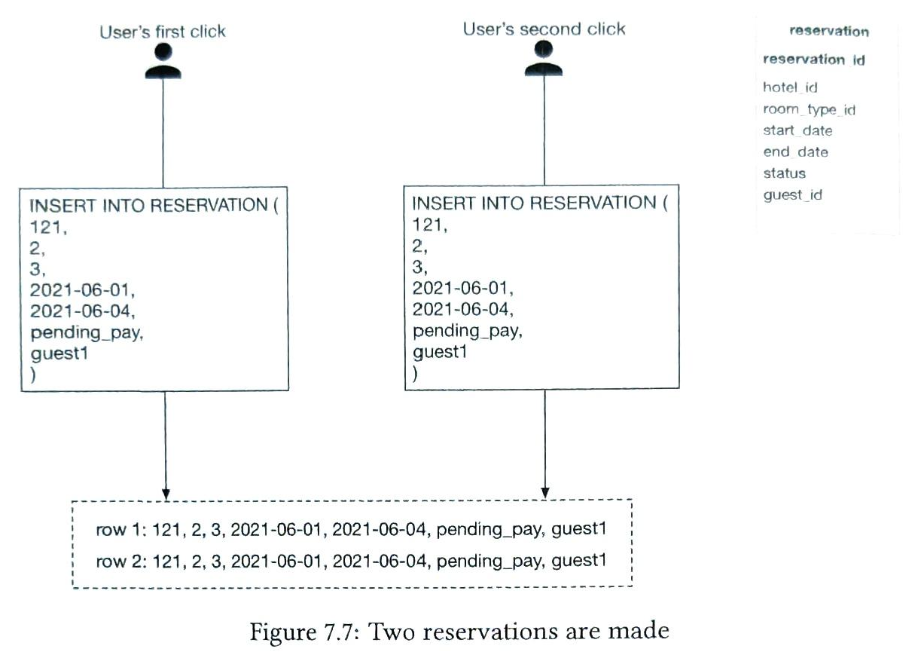
The room\_type\_inventory table is utilized to check if a customer can reserve a specific type of room or not.

**If the reservation data is too large for a single database, what would you do?**

* Store only current and future reservation data. Reservation history is not frequently accessed. So they can be achieved and some can even be moved to cold storage.
* Data sharding: The most frequent queries include making a reservation or looking up a reservation by name. In both queries we need to choose the hotel first, meaning hotel\_id is a good sharding key. The data can be shared using consistent hashing.

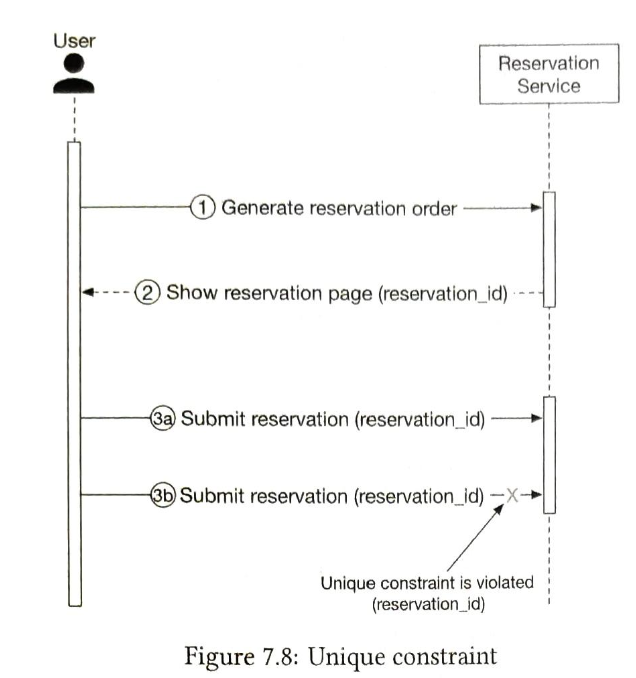
#### **Concurrency issue**

We need to solve two problems:

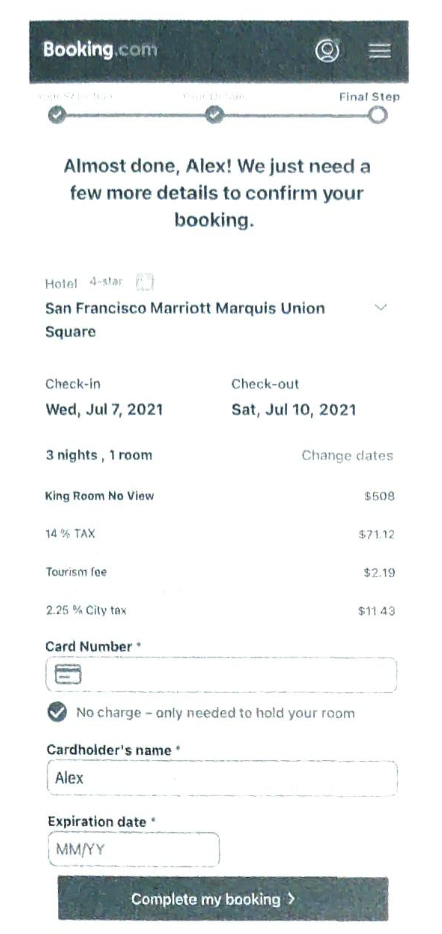
1. The same user clicks on the book button multiple times.
2. Multiple users try to book the same room at the same time.

There are two common approaches to solve this problem:

* Client-side implementation. Aclient can gray out, hide or disable the "submit" button once a request is sent. This should prevent the double-clicking issue most of the time. However, this approach is not very reliable. For example, users can disable javascript, thereby bypassing the client check.
* Idempotent APIs: Add an idempotency 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.



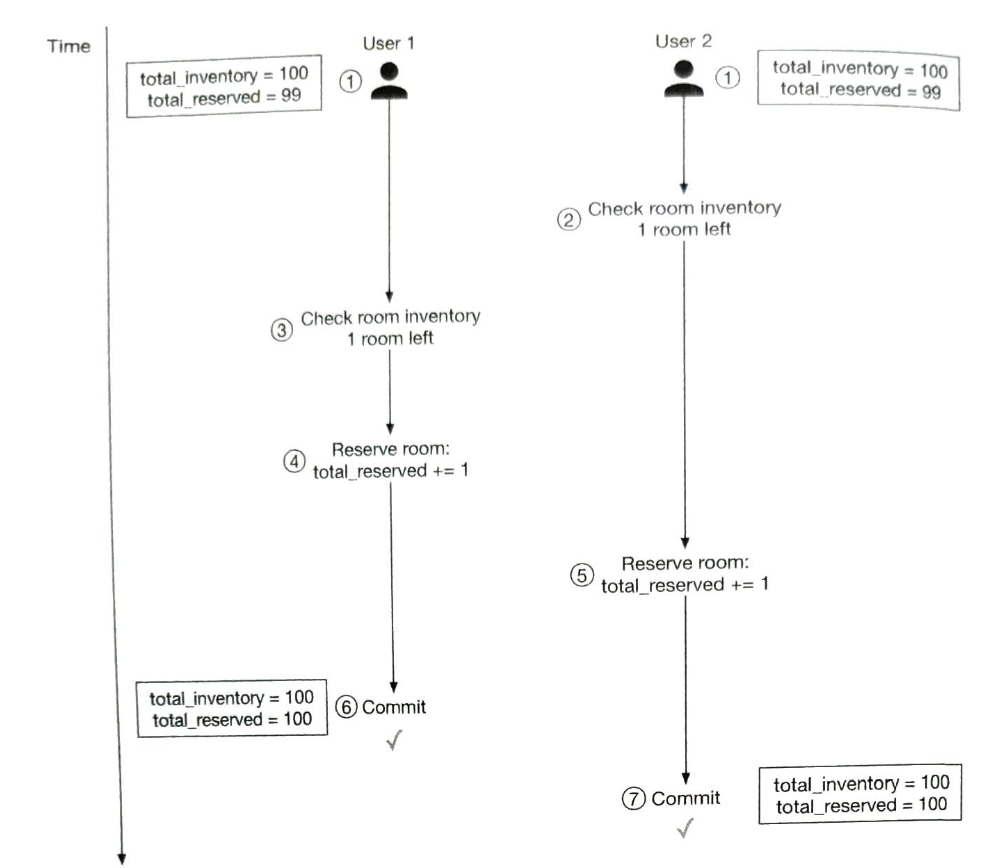
1. Generate a reservation order. After a customer enters detailed information about the reservation (room type, check-in date, check-out date, etc) and clicks the "continue" button, the reservation order is generated by the reservation service.
2. The system generates a reservation order for the customer to review. The unique reservation\_id is generated by a globally unique ID generator and returned as part of the API response. The UI of this step might look like this:



3a. Submit reservation 1. The reservation\_id is included as part of the request. It is the primary key of the reservation table. Please note that the idempotency key doesn't have to be the reservation\_id. We choose reservation\_id because it already exists and works well for our design.

3b. If a user clicks the "Complete my booking" button a second time, reservation 2 is submitted. Because reservation id is the primary key of the reservation table, we can rely on the unique constraint of the key to ensure no double reservation happens.

Scenario 2: what happens if multiple users book the same type of room at the same time when there is only one room left?



The solution to this problem requires some form of locking mechanism. We explore the following techniques:

* Pessimistic locking
* Optimistic locking
* Database Constraint

Before jumping into a fix, let's take a look at the SQL pseudo-code utilized to reserve a room. The SQL has two parts:

• Check room inventory

• Reserve a room

**Option 1: Pessimistic locking**

Let's assume a transaction is started by "transaction 1". Other transactions have to wait for transaction 1 to finish before beginning another transaction.

Pros:

* Prevents applications from updating data that is changing or changed.
* It is easy to implement and it avoids conflict by serializing updates. Pessimistic locking is useful when data contention is heavy.

Cons:

* Deadlocks may occur when multiple resources are locked. Writing deadlock free code could be challenging.
* 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.

**Option 2: Optimistic locking**

There are two common ways to implement optimistic locking: version number and timestamp. Version number is generally considered to be a better option because the server clock can be inaccurate over time.

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. 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.

Optimistic locking is usually faster than pessimistic locking because we do not lock DB. However, the performance of optimistic locking drops drastically when concurrency is high.

To understand why, consider the case when many clients try to reserve a hotel room at the same time. Because there is no limit on how many clients can read the available room count, all of them read back the same available room count and the current version number. When different clients make reservations and write back the results to the database, only one of them will succeed, and the rest of the clients receive a version check failure message. These clients have to retry. In the subsequent round of retries, there is only one successful client again, and the rest have to retry. Although the end result is correct,repeated retries cause a very unpleasant user experience.

Pros:

* Prevents applications from updating stale data.
* We don’t need to lock the DB resource. It’s entirely up to application to handle logic with the version number.
* Generally used when data contention is low. I.e. when conflicts are rare, transactions can complete without the expense of managing locks.

Cons:

* Performance is poor when data contention is heavy.

**Option 3: Database Constraints**

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

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.

Pros:

* Easy to implement
* Works well when data retention is minimal.

Cons:

* Similar to optimistic locking: when data contention is heavy, it can result in a high volume of failures. Users could see there are rooms available, but when they try to book one, they get the "no rooms available" response. The experience can be frustrating to users.
* The database constraints cannot be version-controlled easily like the application code.
* Not all databases support constraints. It might cause problems when we migrate from one DB to another

Since this approach is easy to implement the data contention for a hotel reservation is usually not high(low QPS),it is another good option for the hotel reservation system.

#### **Scalability**

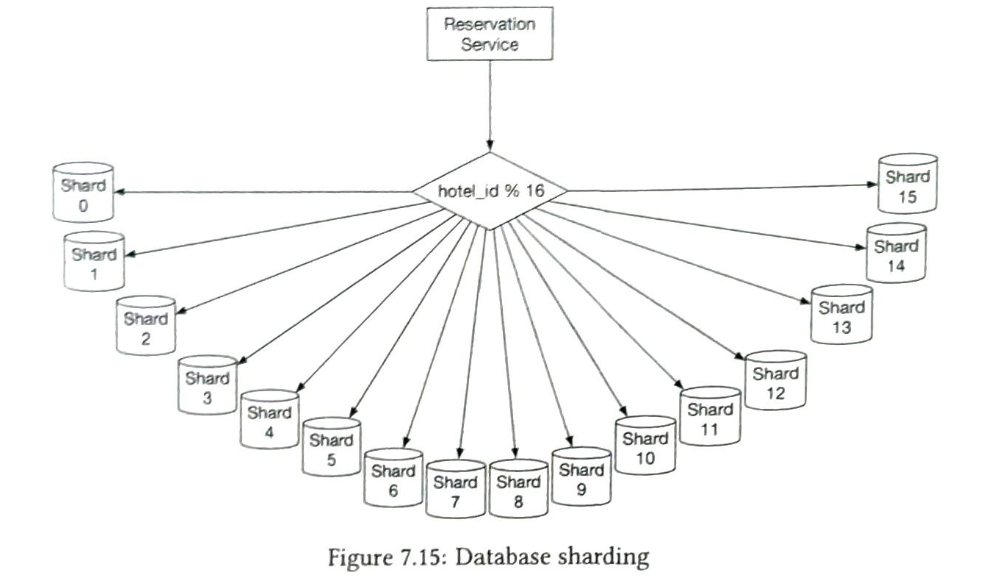
What if the hotel reservation is used not just for a hotel chain but for a popular travel site such as booking.com or expedia.com?In this case QPS is 1000 times higher.

All our services are stateless, so they can be easily expanded by adding more servers. The database however contains all the states and cannot be scaled up by simply adding more database.

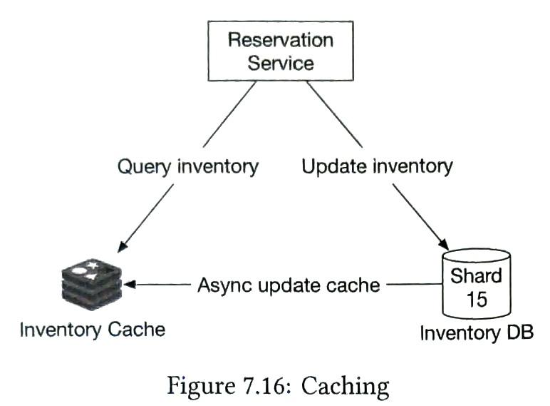
**Database sharing**

One way to scale DB is to apply DB sharding. The idea is to split data into multiple databases so that each of them only contains a portion of data.

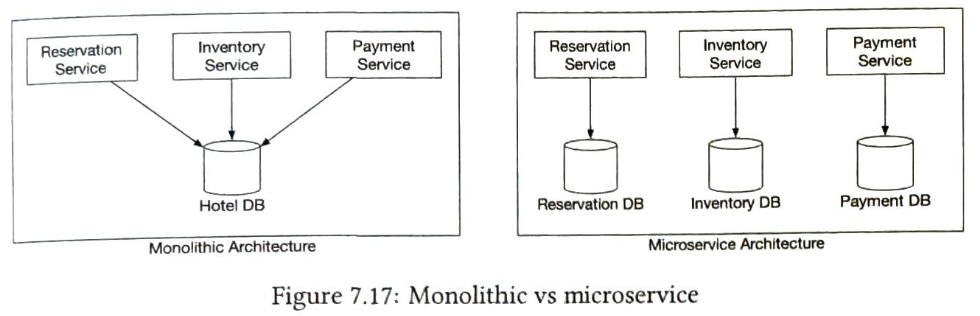
As we can see from the data model section, most queries need to filter by hotel\_id. So the natural conclusion is we shard data by hotel\_id. Assume net QPS is 30,000. After database sharding, each shard handles

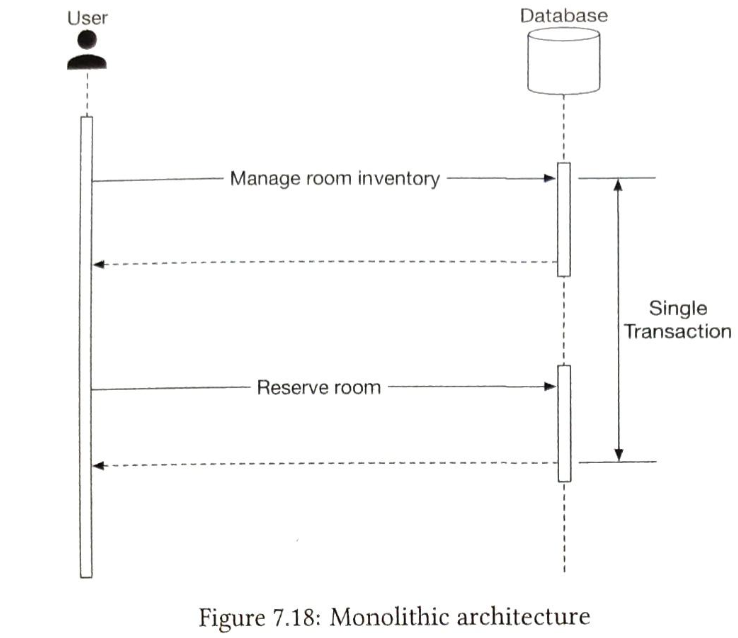
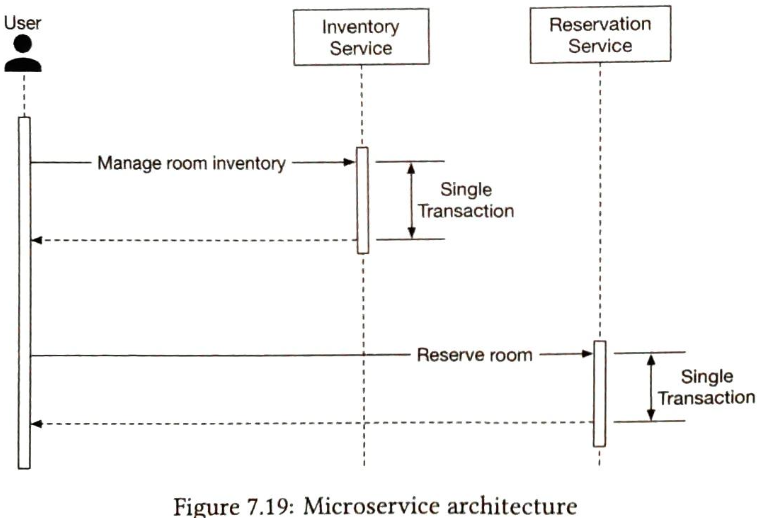
30,000/16=1,875 QPS, which is within a single MySQL server's load capacity.

**Caching**



#### **Data inconsistency among services**





To address the data inconsistency, here is a high-level summary of industry-proven technique:

• Two-phase commit (2PC) [12]. 2PC is a database protocol used to guarantee atomic transaction commit across multiple nodes, i.e., either all nodes succeeded or all nodes failed. Because 2PC is a blocking protocol, a single node failure blocks the progress until the node has recovered. It's not performant.

• Saga. A Saga is a sequence of local transactions. Each transaction updates and pub- lishes a message to trigger the next transaction step. If a step fails, the saga executes compensating transactions to undo the changes that were made by preceding trans- actions [13). 2PC works as a single commit to perform ACID transactions while Saga consists of multiple steps and relies on eventual consistency.