Question 1

Q: Like a lot of other sharing-economy products out there, Airbnb has two sides: a host-facing side and a renter-facing side. Are we designing both of these sides or just one of them? A: Let's design both of these sides of the product.

Question 2

Q: Okay. So we're probably designing the system for hosts to create and maybe delete listings, and the system for renters to browse through properties, book them, and manage their bookings afterwards. Is that correct? A: Yes for hosts; but let's actually just focus on browsing through listings and booking them for renters. We can ignore everything that happens after booking on the renter-facing side.

Question 3

Q: Okay, but for booking, is the idea that, when a user is browsing a property for a specific date range, the property gets temporarily reserved for them if they start the booking process?

A: Yes. More specifically, multiple users should be allowed to look at the same property, for the same date range, concurrently without issues. But once a user starts the booking process for a property, it should be reflected that this property is no longer available for the dates in question if another user tries to book it.

Question 4

Q: I see. But so, let's say two users are looking at the exact same property for an overlapping date range, and one user presses "Book Now", at which point they have to enter credit card information. Should we immediately lock the property for the other user for some predetermined period of time, like maybe 15 minutes, and if the first person actually goes through with booking the property, then this "lock" becomes permanent? A: Yes, that makes sense. In real life, there might be slight differences, but for the sake of this design, let's go with that.

Question 5

Q: Okay. And do we want to design any auxiliary features like being able to contact hosts, authentication and payment services, etc., or are we really just focusing on browsing and reserving? A: Let's really just focus on browsing and booking. We can ignore the rest.

Question 6

Q: I see. So, since it sounds like we're designing a pretty targeted part of the entire Airbnb service, I want to make sure that I know exactly every functionality that we want to support. My understanding is that users can go on the main Airbnb website or app, they can look up properties based on certain criteria, like location, available date range, pricing, property details, etc., and then they can decide to book a location. As for hosts, they can basically just create a listing and delete it like I said earlier. Is that correct?

A: Yes. But actually, for this design, let's purely filter based on location and available date range as far as listing characteristics are concerned; let's not worry about other criteria like pricing and property details.

Question 7

Q: What is the scale that we're designing this for? Specifically, roughly how many listings and renters do we expect to cater to? A: Let's only consider Airbnb's U.S. operations. So let's say 50 million users and 1 million listings.

Question 8

Q: Regarding systems characteristics like availability and latency, I'm assuming that, even if a renter looks up properties in a densely-populated area like NYC, where there might be a lot of listings, we care about serving these listings fast, accurately, and reliably. Is that correct?

A: Yes, that's correct. Ideally, we don't want any downtime for renters browsing listings.

1. GATHERING SYSTEM REQUIREMENTS

As with any systems design interview question, the first thing that we want to do is to gather system requirements; we need to figure out what system we're building exactly. We're designing the core system behind Airbnb, which allows hosts to create property listings and renters to browse through these listings and book them.

Specifically, we'll want to support:

- On the host side, creating and deleting listings.
- On the renter side, browsing through listings, getting individual listings, and "reserving" listings.

"Reserving" listings should happen when a renter presses some "Book Now" button and should effectively lock (or reserve) the listing for some predetermined period of time (say, 15 minutes), during which any other renter shouldn't be able to reserve the listing or to even browse it (unless they were already browsing it).

We don't need to support anything that happens after a reservation is made, except for freeing up the reservation after 15 minutes if the renter doesn't follow through with the booking and making the reservation permanent if the renter does actually book the listing in question.

Regarding listings, we should focus on browsing and reserving them based on location and available date range; we can ignore any other property characteristics like price, number of bedrooms, etc.. Browsing listings should be as quick as possible, and it should reflect newly created listings as fast as possible. Lastly, reserved and booked listings shoudln't be browsable by renters.

Our system should serve a U.S.-based audience with approximately 50 million users and 1 million listings.

2. COMING UP WITH A PLAN

We'll tackle this system by dividing it into two main sections:

- The host side.
- The renter side.

We can further divide the renter side as follows:

- Browsing (**listing**) listings.
- **Getting** a single listing.
- **Reserving** a listing.

3. LISTINGS STORAGE & QUADTREE

First and foremost, we can expect to store all of our listings in a **SQL table**. This will be our primary source of truth for listings on Airbnb, and whenever a host creates or deletes a listing, this SQL table will be written to.

Then, since we care about the latency of browsing listings on Airbnb, and since this browsing will require querying listings based on their location, we can store our listings in a region **quadtree** (the db index based on latitude and longitude) , to be traversed for all browsing functionality.

Since we're optimizing for speed, it'll make sense to store this quadtree in memory on some auxiliary machine, which we can call a "geo index," but we need to make sure that we can actually fit this quadtree **in memory**.

In this quadtree, we'll need to store all the information about listings that needs to be displayed on the UI when a renter is browsing through listings: a title, a description, a link pointing to a property image, a unique listing ID, etc..

Assuming a single listing takes up roughly 10 KB of space (as an upper bound), some simple math confirms that we can store everything we need about listings in memory.

~10 KB per listing
~1 million listings
~10 KB * 1000^2 = 10 GB

Since we'll be storing our **quadtree in memory**, we'll want to make sure that a single machine failure doesn't bring down the entire browsing functionality. To ensure this, we can set up a cluster of machines, each holding an instance of our quadtree in memory, and these machines can use **leader election** to safeguard us from machine failures.

Our quadtree solution works as follows : when our system boots up, the geo-index machines create the quadtree by querying our SQL table of listings. When listings are created or deleted, hosts first write to the SQL table, and then they synchronously update the geo-index leader's quadtree. Then, on an interval of say, 10 minutes, the geo-index leader and followers all recreate the quadtree from the SQL table, which allows them to stay up to date with new listings.

If the leader dies at any point, one of the followers takes its place, and data in the new leader's quadtree will be stale for at most a few minutes until the interval forces the quadtree to be recreated.

4. LISTING LISTINGS

When renters browse through listings, they'll have to hit some *ListListings* **API** endpoint. This API call will search through the geo-index leader's quadtree for relevant listings based on the location that the renter passes.

Finding relevant locations should be fairly straightforward and very fast, especially since we can estimate that our quadtree will have a depth of approximately 10, since 4^10 is greater than 1 million.

That being said, we'll have to make sure that we don't return listings that are unavailable during the date range specified by the renter. In order to handle this, each listing in the quad tree will contain a list of unavailable date ranges, and we can perform a simple binary search on this list for each listing, in order to determine if the listing in question is available and therefore browsable by the renter.

We can also make sure that our quadtree returns only a subset of relevant listings for pagination purposes, and we can determine this subset by using an offset: the first page of relevant listings would have an offset of 0, the second page would have an offset of 50 (if we wanted pages to have a size of 50), the third page would have an offset of 100, and so on and so forth.

5. GETTING INDIVIDUAL LISTINGS

This API call should be extremely simple; we can expect to have listing IDs from the list of listings that a renter is browsing through, and we can simply query our SQL table of listings for the given ID.

6. RESERVING LISTINGS

Reserved listings will need to be reflected both in our quadtree and in our persistent storage solution. In our quadtree, because they'll have to be excluded from the list of browsable listings; in our persistent storage solution, because if our quadtree needs to have them, then the main source of truth also needs to have them.

We can have a second SQL table for reservations, holding listing IDs as well as date ranges and timestamps for when their reservations expire. When a renter tries to start the booking process of a listing, the reservation table will first be checked to see if there's currently a reservation for the given listing during the specified date range; if there is, an error is returned to the renter; if there isn't, a reservation is made with an expiration timestamp 15 minutes into the future.

Following the write to the reservation table, we synchronously update the geo-index leader's quadtree with the new reservation. This new reservation will simply be an unavailability interval in the list of unavailabilities on the relevant listing, but we'll also specify an expiration for this unavailability, since it's a reservation.

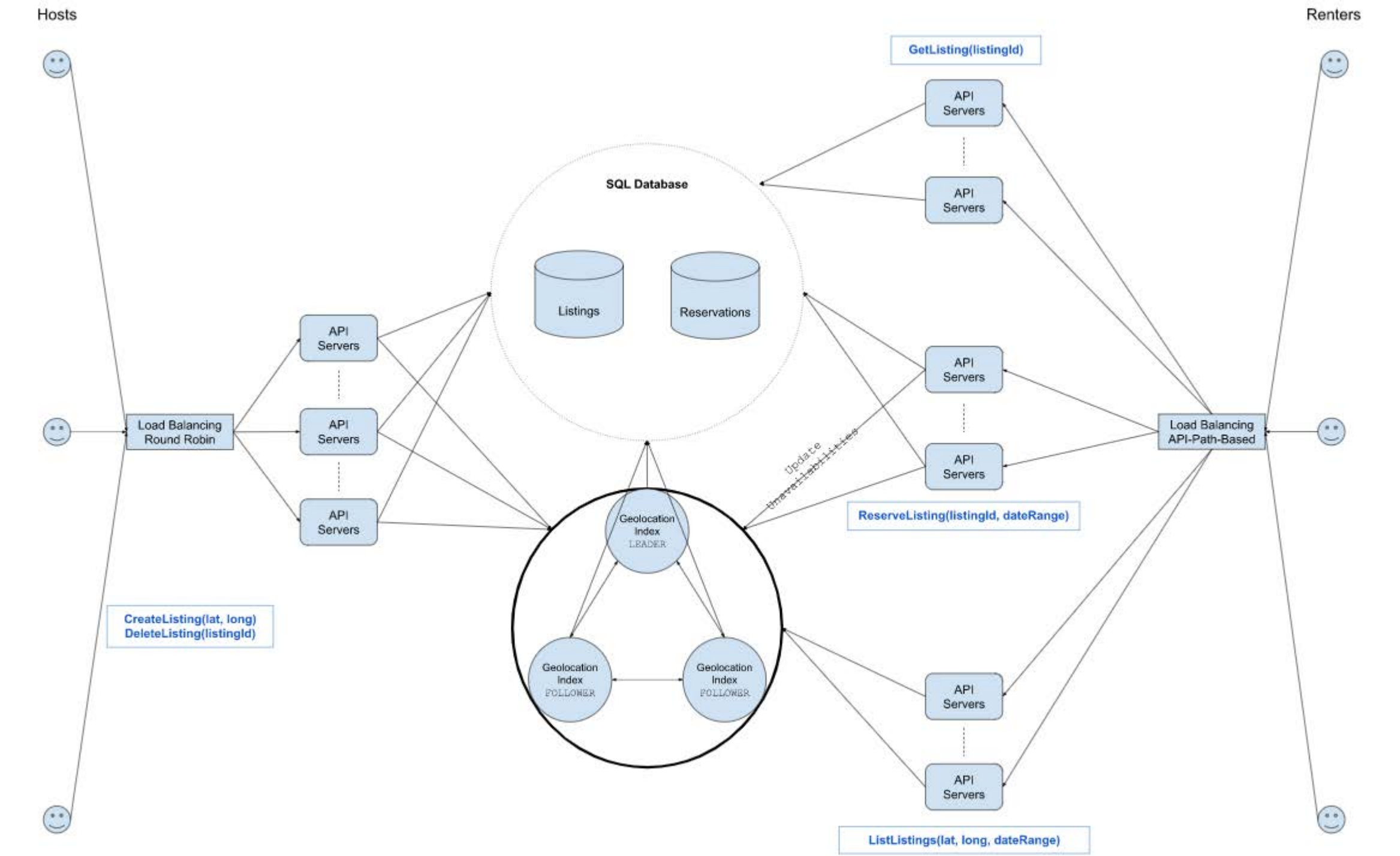
A listing in our quadtree might look something like this:

```
{
 "unavailabilities": [
  {
   "range": ["2020-09-22T12:00:00-05:00", "2020-09-28T12:00:00-05:00"],
   "expiration": "2020-09-16T12:00:00-04:00"
  }
  {
   "range": ["2020-10-02T12:00:00-05:00", "2020-10-10T12:00:00-05:00"],
   "expiration": null
  },
 ],
 "title": "Listing Title",
 "description": "Listing Description",
 "thumbnailUrl": "Listing Thumbnail URL",
 "id": "Listing ID"
}
```

7. LOAD BALANCING

On the host side, we can load balance requests to create and delete listings across a set of API servers using a simple **round-robin** approach. The API servers will then be in charge of writing to the SQL database and of communicating with the geo-index leader.

On the renter side, we can load balance requests to list, get, and reserve listings across a set of API servers using an API-path-based server-selection strategy. Since workloads for these three API calls will be considerably different from one another, it makes sense to separate these calls across different sets of API servers. Of note is that we don't want any caching done at our API servers, because otherwise we'll naturally run into stale data as reservations, bookings, and new listings appear.


Hosts
Renters
GetListing(listingId)
API Servers
API Servers
SQL Database
Listings
Reservations
API Servers
API Servers
API Servers
Load Balancing Round Robin
API Servers
API Servers
API Servers
Load Balancing API-Path-Based
update unavailabilities
Geolocation Index LEADER
ReserveListing(listingId, dateRange)
CreateListing(lat, long)
DeleteListing(listingId)
Geolocation Index FOLLOWER
Geolocation Index FOLLOWER
API Servers
API Servers
ListListings(lat, long, dateRange)