Solution Walkthrough
13 Questions with solutions

Design A Code-Deployment System

Design a global and fast code-deployment system.

Many systems design questions are intentionally left very vague and are literally given in the form of `Design Foobar`. It's your job to ask clarifying questions to better understand the system that you have to build.

We've laid out some of these questions below; their answers should give you some guidance on the problem. Before looking at them, we encourage you to take few minutes to think about what questions you'd ask in a real interview.

Solution

Our solution walkthroughs are meant to supplement our video solutions. We recommend starting with the video solution and using the walkthrough either as a point of reference while you watch the video or as a review tool if you need to brush up on this question's solution later on.

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.

From the answers we were given to our clarifying questions (see Prompt Box), we're building a system that involves repeatedly (in the order of thousands of times per day) building and deploying code to ***hundreds of thousands*** *of machines spread out across* ***5-10 regions*** *around the world.*

Building code will involve grabbing snapshots of source code using commit SHA identifiers; beyond that, we can assume that the actual implementation details of the building action are taken care of. In other words, we don't need to worry about how we would build JavaScript code or C++ code; we just need to design the system that enables the repeated building of code.

*Building code will take up to **15 minutes**, it'll result in a binary file of up to **10GB**, and we want to have the entire deployment process (building and deploying code to our target machines) take at most **30 minutes**.*

*Each build will need a clear end-state (**SUCCESS** or **FAILURE**), and though we care about availability (2 to 3 nines), we don't need to optimize too much on this dimension.*

2. Coming Up With A Plan

It's important to organize ourselves and to lay out a clear plan regarding how we're going to tackle our design. What are the major, distinguishable components of our how system?

It seems like this system can actually very simply be divided into two clear subsystems:

- *the Build System that builds code into binaries*
- *the Deployment System that deploys binaries to our machines across the world*

Note that these subsystems will of course have many components themselves, but this is a very straightforward initial way to approach our problem.

3. Build System -- General Overview

*From a high-level perspective, we can call the process of building code into a binary a **job**, and we can design our build system as a queue of jobs. Jobs get added to the queue, and each job has a commit identifier (the commit SHA) for what version of the code it should build and the name of the* *artifact that will be created (the name of the resulting binary). Since we're agnostic to the type of the code being built, we can assume that all languages are handled automatically here.*

We can have a pool of servers (workers) that are going to handle all of these jobs. Each worker will repeatedly take jobs off the queue (in a ***FIFO manner****—no prioritization for now), build the relevant binaries (again, we're assuming that the actual implementation details of building code are given to us), and write the resulting binaries to blob storage (****Google Cloud Storage*** *or* ***S3*** *for instance). Blob storage makes sense here, because binaries are literally blobs of data.*

4. Build System -- Job Queue

A naive design of the job queue would have us implement it in memory (just as we would implement a queue in coding interviews), but this implementation is very problematic; if there's a failure in our servers that hold this queue, we lose the entire state of our jobs: queued jobs and past jobs.

It seems like we would be unnecessarily complicating matters by trying to optimize around this in-memory type of storage, so we're likely better off implementing the queue using a SQL database.

5. Build System -- SQL Job Queue

We can have a ***jobs*** *table in our SQL database where every record in the database represents a job, and we can use record-creation timestamps as the queue's ordering mechanism.*

Our table will be:

- *id: string, the ID of the job, auto-generated*
- *created_at: timestamp*
- *commit_sha: string*
- *name: string, the pointer to the job's eventual binary in blob storage*
- *status: string,* ***QUEUED****,* ***RUNNING****,* ***SUCCEEDED****,* ***FAILED***

We can implement the actual dequeuing mechanism by looking at the oldest creation_timestamp with a QUEUED status. This means that we'll likely want to index our table on both created_at and status.

6. Build System -- Concurrency

***ACID transactions** will make it safe for potentially hundreds of workers to grab jobs off the queue without unintentionally running the same job twice (we'll avoid race conditions). Our actual transaction will look like this:*

```
BEGIN TRANSACTION;

 SELECT * FROM jobs_table WHERE status = 'QUEUED' ORDER BY created_at ASC LIMIT 1;

 // if there's none, we ROLLBACK;

 UPDATE jobs_table SET status = 'RUNNING' WHERE id = id from previous query;

 COMMIT;
```

All of the workers will be running this transaction every so often to dequeue the next job; let's say every 5 seconds. If we arbitrarily assume that we'll have 100 workers sharing the same queue, we'll have 100/5 = 20 reads per second, which is very easy to handle for a SQL database.

7. Build System -- Lost Jobs

Since we're designing a large-scale system, we have to expect and handle edge cases. Here, what if there's a network partition with our workers or one of our workers dies mid-build? Since builds last around 15 minutes on average, this will very likely happen. In this case, we want to avoid having a "lost job" that we were never made aware of, and with our current design, the job will remain RUNNING forever. How do we handle this?

*We could have an extra column on our **jobs** table called last_heartbeat. This will be updated in a heartbeat fashion by the worker running a particular job, where that worker will update the relevant row in the table every 3-5 minutes to just let us know that it's still running the job.*

*We can then have a completely separate service that polls the table every so often (say, every 5 minutes, depending on how responsive we want this build system to be), checks all of the **RUNNING** jobs, and if their last_heartbeat was last modified longer than 2 heartbeats ago (we need some margin of error here), then something's likely wrong, and this service can reset the status of the relevant jobs to **QUEUED**, which would effectively bring them back to the front of the queue.*

The transaction that this auxiliary service will perform will look something like this:

```
 UPDATE jobs_table SET status = 'QUEUED' WHERE

    status = 'RUNNING' AND

    last_heartbeat < NOW() - 10 minutes;
```

8. Build System -- Scale Estimation

We previously arbitrarily assumed that we would have 100 workers, which made our SQL-database queue able to handle the expected load. We should try to estimate if this number of workers is actually realistic.

*With some back-of-the-envelope math, we can see that, since a build can take up to 15 minutes, a single worker can run 4 jobs per hour, or ~100 (96) jobs per day. Given thousands of builds per day (say, 5000-10000), this means that we would need **50-100 workers** (5000 / 100). So our arbitrary figure was accurate.*

Even if the builds aren't uniformly spread out (in other words, they peak during work hours), our system scales horizontally very easily. We can automatically add or remove workers whenever the load warrants it. We can also scale our system vertically by making our workers more powerful, thereby reducing the build time.

9. Build System -- Storage

*We previously mentioned that we would store binaries in blob storage (**GCS**). Where does this storage fit into our queueing system exactly?*

*When a worker completes a build, it can store the binary in GCS before updating the relevant row in the **jobs** table. This will ensure that a binary has been persisted before its relevant job is marked as **SUCCEEDED**.*

Since we're going to be deploying our binaries to machines spread across the world, it'll likely make sense to have regional storage rather than just a single global blob store.

We can design our system based on regional clusters around the world (in our 5-10 global regions). Each region can have a blob store (a regional GCS bucket). Once a worker successfully stores a binary in our main blob store, the worker is released and can run another job, while the main blob store performs some asynchronous replication to store the binary in all of the regional GCS buckets. Given 5-10 regions and 10GB files, this step should take no more than 5-10 minutes, bringing our total build-and-deploy duration so far to roughly 20-25 minutes (15 minutes for a build and 5-10 minutes for global replication of the binary).

10. Deployment System -- General Overview

From a high-level perspective, our actual deployment system will need to allow for the very fast distribution of 10GB binaries to hundreds of thousands of machines across all of our global regions. We're likely going to want some service that tells us when a binary has been replicated in all regions, *another service that can serve as the source of truth for what binary should currently be run on all machines, and finally a peer-to-peer-network design for our actual machines across the world.*

11. Deployment System -- Replication-Status Service

We can have a global service that continuously checks all regional GCS buckets and aggregates the replication status for successful builds (in other words, checks that a given binary in the main blob store has been replicated across all regions). Once a binary has been replicated across all regions, this service updates a separate SQL database with rows containing the name of a binary and a ***replication_status****. Once a binary has a "complete"* ***replication_status****, it's officially deployable.*

12. Deployment System -- Blob Distribution

Since we're going to deploy 10 GBs to hundreds of thousands of machines, even with our regional clusters, having each machine download a 10GB file one after the other from a regional blob store is going to be extremely slow. A peer-to-peer-network approach will be much faster and will allow us to hit our 30-minute time frame for deployments. All of our regional clusters will behave as peer-to-peer networks.

13. Deployment System -- Trigger

Let's describe what happens when an engineer presses a button on some internal UI that says "Deploy build/binary B1 to every machine globally". This is the action that triggers the binary downloads on all the regional peer-to-peer networks.

To simplify this process and to support having multiple builds getting deployed concurrently, we can design this in a goal-state oriented manner.

The goal-state will be the desired build version at any point in time and will look something like: "current_build: ***B1****", and this can be stored in some dynamic configuration service (a* ***key-value store*** *like* ***Etcd*** *or* ***ZooKeeper****). We'll have a global goal-state as well as regional goal-states.*

Each regional cluster will have a K-V store that holds configuration for that cluster about what builds should be running on that cluster, and we'll also have a global K-V store.

When an engineer clicks the "Deploy build/binary B1" button, our global K-V store's build_version will get updated. Regional K-V stores will be continuously polling the global K-V store (say, every 10 seconds) for updates to the build_version and will update themselves accordingly.

Machines in the clusters/regions will be polling the relevant regional K-V store, and when the build_version changes, they'll try to fetch that build from the P2P network and run the binary.

Design AlgoExpert

Many systems design questions are intentionally left very vague and are literally given in the form of `Design Foobar`. It's your job to ask clarifying questions to better understand the system that you have to build.

We've laid out some of these questions below; their answers should give you some guidance on the problem. Before looking at them, we encourage you to take few minutes to think about what questions you'd ask in a real interview.

Solution

Our solution walkthroughs are meant to supplement our video solutions. We recommend starting with the video solution and using the walkthrough either as a point of reference while you watch the video or as a review tool if you need to brush up on this question's solution later on.

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.

From the answers we were given to our clarifying questions (see Prompt Box), we're building the core AlgoExpert user flow, which includes users landing on the website, accessing questions, marking them as complete, writing code, running code, and having their code saved.

We don't need to worry about payments or authentication, and we don't need to go too deep into the code-execution engine.

We're building this platform for a global audience, with an emphasis on U.S. and India users, and we don't need to overly optimize our system's availability. We probably don't need more than two or three nines, because we're not building a health or security system, and this gets us somewhere between ***8 hours and 3 days*** *of downtime per year, which is reasonable. All in all, this means that we don't need to worry too much about availability.*

We care about latency and throughput within reason, but apart from the code-execution engine, this doesn't seem like a particularly difficult aspect of our system.

2. Coming Up With A Plan

It's important to organize ourselves and to lay out a clear plan regarding how we're going to tackle our design. What are the major, distinguishable components of our how system?

On the one hand, AlgoExpert has a lot of static content; the entire home page, for instance, is static, and it has a lot of images. On the other hand, AlgoExpert isn't just a static website; it clearly has a lot of dynamic content that users themselves can generate (code that they can write, for example). So we'll need to have a robust API backing our UI, and given that user content gets saved on the website, we'll also need a database backing our API.

We can divide our system into 3 core components:

- *Static UI content*

- *Accessing and interacting with questions (question completion status, saving solutions, etc.)*
- *Ability to run code*

Note that the second bullet point will likely get further divided.

3. Static UI Content

For the UI static content, we can put public assets like images and JavaScript bundles in a blob store: ***S3 or Google Cloud Storage****. Since we're catering to a global audience and we care about having a responsive website (especially the home page of the website), we might want to use a* ***Content Delivery Network*** *(CDN) to serve that content. This is especially important for a better mobile experience because of the slow connections that phones use.*

4. Main Clusters And Load Balancing

For our main backend servers, we can have ***2 primary clusters*** *in the 2 important regions: U.S. and India.*

We can have some DNS load balancing to route API requests to the cluster closest to the user issuing the requests, and within a region, we can have some ***path-based load balancing*** *to separate our services (payments, authentication, code execution, etc.), especially since the code execution platform will probably need to run on different kinds of servers compared to those of the rest of the API. Each service can probably have a set of servers, and we can do some round-robin load balancing at that level (this is probably handled directly at the path-based load balancing layer).*

5. Static API Content

There's a lot of static API content on AlgoExpert: namely, the list of questions and all of their solutions. We can store all of this data in a blob store for simplicity.

6. Caching

We can implement 2 layers of caching for this static API content.

We can have client-side caching; this will improve the user experience on the platform (users will only need to load questions once per session), and this will reduce the load on our backend servers (this will probably save 2-3 network calls per session).

We can also have some in-memory caching on our servers. If we approximate 100 questions with 10 languages and 5KB per solution, this should be less than ***100 * 10 * 5000 bytes = 5MB*** *of total data to keep in memory, which is perfectly fine.*

Since we were told that we want to make changes to static API content every couple of days and that we want those changes to be reflected in production as soon as possible, we can invalidate, evict and replace the data in our server-side caches every 30 minutes or so.

7. Access Control

Whenever you're designing a system, it's important to think about any potential access control that needs to be implemented. In the case of AlgoExpert, there's straightforward access control with regards to question content: users who haven't purchased AlgoExpert can't access individual questions. We can implement this fairly easily by just making some internal API call whenever a user requests our static API content to figure out if the user owns the product before returning the full content for questions.

8. User Data Storage

For user data, we have to design the storage of question completion status and of user solutions to questions. Since this data will have to be queried a lot, a SQL database like ***Postgres or MySQL*** *seems like a good choice.*

We can have 2 tables. The first table might be ***question_completion_status****, which would probably have the following columns:*

- *id: integer, primary key (an auto-incremented integer for instance)*
- *user_id: string, references the id of the user (can be obtained from auth)*
- *question_id: string, references the id of the question*
- *completion_status: string, enum to represent the completion status of the question*

We could have a uniqueness constraint on (user_id, question_id) and an index on user_id for fast querying.

*The second table might be **user_solutions**:*

- *id: integer, primary key (an auto-incremented integer for instance)*
- *user_id: string, references the id of the user (can be obtained from auth)*
- *question_id: string,references the id of the question*
- *language: string, references the language of the solution*
- *solution: string, contains the user's solution*

*We could have a uniqueness constraint on (**user_id, question_id, language**) and an index on **user_id** as well as one on **question_id**. If the number of languages goes up significantly, we might also want to index on language to allow for fast per-language querying so that the UI doesn't fetch all of a user's solutions at the same time (this might be a lot of data for slow connections).*

9. Storage Performance

*Marking questions as complete and typing code in the coding workspace (with a 1-3 second **debounce** for performance reasons) will issue API calls that write to the database. We likely won't get more than **1000 writes per second** given our user numbers (assuming roughly 10,000 users on the platform at any given point in time), which SQL databases can definitely handle.*

We can have 2 major database servers, each serving our 2 main regions: 1 in North America and 1 in India (perhaps serving Southeast Asia). If need be, we can add a 3rd cluster serving Europe exclusively (or other parts of the world, as our platform grows).

10. Inter-Region Replication

Since we'll have 2 primary database servers, we'll need to keep them up to date with each other. Fortunately, users on AlgoExpert don't share user-generated content; this means that we don't need data that's written to 1 database server to immediately be written to the other database server (this would likely have eliminated the latency improvements we got by having regional databases).

That being said, we do need to keep our databases up to date with each other, since users might travel around the world and hit a different database server than their typical one.

For this, we can have some async replication between our database servers. The replication can occur every 12 hours, and we can adjust this according to behavior in the system and amount of data that gets replicated across continents.

11. Code Execution

First of all, we should implement some rate limiting. A service like code execution lends itself perfectly to rate limiting, and we can implement some tier-based rate limiting using a K-V Store like ***Redis*** *to easily prevent DoS attacks. We can limit the number of code runs to once every second, 3 times per 5 seconds, and 5 times per minute. This will prevent DoS attacks through the code-execution API, but it'll still allow for a good user experience when running code.*

Since we want 1-3 seconds of latency for running code, we need to keep a set of special servers--our "workers"-- ready to run code at all times. They can each clean up after running user code (remove extra generated files as a result of compilation, for example) so that they don’t need to be killed at any point. Our backend servers can contact a free worker and get the response from that worker when it's done running code (or if the code timed out), and our servers can return that to the UI in the same request.

Given that certain languages need to be compiled, we can estimate that it would take on average 1 second to compile and run the code for each language. People don't run code that often, so we can expect 10 run-codes per second in total given roughly 10,000 users on the website at once, so we'll probably need 10-100 machines to satisfy our original latency requirement of 1-3 seconds per run-code (10 machines if 10 run-codes per second is accurate, more if we experience higher load).

This design scales horizontally with our number of users, and it can scale vertically to make running code even faster (more CPU == faster runs).

Lastly, we can have some logging and monitoring in our system, especially for running code (tracking run-code events per language, per user, per question, average response time, etc.). This will help us automatically scale our clusters when user demand goes up or down. This can also be useful to know if any malicious behavior is happening with the code-execution engine.

Design A Stockbroker

Design a stockbroker: a platform that acts as the intermediary between end-customers and some central stock exchange.

Many systems design questions are intentionally left very vague and are literally given in the form of `Design Foobar`. It's your job to ask clarifying questions to better understand the system that you have to build.

We've laid out some of these questions below; their answers should give you some guidance on the problem. Before looking at them, we encourage you to take few minutes to think about what questions you'd ask in a real interview.

Solution

Our solution walkthroughs are meant to supplement our video solutions. We recommend starting with the video solution and using the walkthrough either as a point of reference while you watch the video or as a review tool if you need to brush up on this question's solution later on.

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 building a stock-brokerage platform like Robinhood that functions as the intermediary between end-customers and some central stock exchange. The idea is that the central stock exchange is the platform that actually executes stock trades, whereas the stockbroker is just the platform that customers talk to when they want to place a trade--the stock brokerage is "simpler" and more "human-readable", so to speak.We only care about supporting market trades--trades that are executed at the current stock price--and we can assume that our system stores customer balances (i.e., funds that customers may have previously deposited) in a SQL table.We need to design a PlaceTrade API call, and we know that the central exchange's equivalent API method will take in a callback that's guaranteed to be executed upon completion of a call to that API method.We're designing this system to support millions of trades per day coming from millions of customers in a single region (the U.S., for example). We want the system to be highly available.

2. Coming Up With A Plan

It's important to organize ourselves and to lay out a clear plan regarding how we're going to tackle our design. What are the major, distinguishable components of our how system?

We'll approach the design front to back:

- *the PlaceTrade API call that clients will make*
- *the API server(s) handling client API calls*
- *the system in charge of executing orders for each customer*

We'll need to make sure that the following hold:

- *trades can never be stuck forever without either succeeding or failing to be executed*
- *a single customer's trades have to be executed in the order in which they were placed*
- *balances can never go in the negatives*

3. API Call

The core API call that we have to implement is PlaceTrade.

We'll define its signature as:

```
PlaceTrade(
  customerId: string,
  stockTicker: string,
  type: string (BUY/SELL),
  quantity: integer,
) => (
  tradeId: string,
  stockTicker: string,
  type: string (BUY/SELL),
  quantity: integer,
  createdAt: timestamp,
  status: string (PLACED),
```

reason: string,

)

The customer ID can be derived from an authentication token that's only known to the user and that's passed into the API call.

The status can be one of:

- ***PLACED***
- ***IN PROGRESS***
- ***FILLED***
- ***REJECTED***

That being said, ***PLACED*** *will actually be the defacto status here, because the other statuses will be asynchronously set once the exchange executes our callback. In other words, the trade status will always be* ***PLACED*** *when the PlaceTrade API call returns, but we can imagine that a GetTrade API call could return statuses other than* ***PLACED****.*

Potential reasons for a ***REJECTED*** *trade might be:*

- *insufficient funds*
- *random error*
- *past market hours*

4. API Server(s)

*We'll need multiple API servers to handle all of the incoming requests. Since we don't need any caching when making trades, we don't need any server stickiness, and we can just use some **round-robin load balancing** to distribute incoming requests between our API servers.*

Once API servers receive a PlaceTrade call, they'll store the trade in a SQL table. This table needs to be in the same SQL database as the one that the balances table is in, because we'll need to use ACID transactions to alter both tables in an atomic way.

The SQL table for trades will look like this:

- *id: string, a random, auto-generated string*
- *customer_id: string, the id of the customer making the trade*
- *stockTicker: string, the ticker symbol of the stock being traded*
- *type: string, either **BUY** or **SELL***
- *quantity: integer (no fractional shares), the number of shares to trade*
- *status: string, the status of the trade; starts as **PLACED***
- *created_at: timestamp, the time when the trade was created*
- *reason: string, the human-readable justification of the trade's status*

The SQL table for balances will look like this:

- *id: string, a random, auto-generated string*
- *customer_id: string, the id of the customer related to the balance*
- *amount: float, the amount of money that the customer has in USD*
- *last_modified: timestamp, the time when the balance was last modified*

5. Trade-Execution Queue

With hundreds of orders placed every second, the trades table will be pretty massive. We'll need to figure out a robust way to actually execute our trades and to update our table, all the while making sure of a couple of things:

- *We want to make sure that for a single customer, we only process a single **BUY** trade at any time, because we need to prevent the customer's balance from ever reaching negative values.*
- *Given the nature of market orders, we never know the exact dollar value that a trade will get executed at in the exchange until we get a response from the exchange, so we have to speak to the exchange in order to know whether the trade can go through.*

We can design this part of our system with a Publish/Subscribe pattern. The idea is to use a message queue like Apache Kafka or Google Cloud Pub/Sub and to have a set of topics that customer ids map to. This gives us at-least-once delivery semantics to make sure that we don't miss new trades. When a customer makes a trade, the API server writes a row to the database and also creates a message that gets routed to a topic for that customer (using hashing), notifying the topic's subscriber that there's a new trade.

This gives us a guarantee that for a single customer, we only have a single thread trying to execute their trades at any time.

Subscribers of topics can be rings of 3 workers (clusters of servers, essentially) that use leader election to have 1 master worker do the work for the cluster (this is for our system's high availability)--the leader grabs messages as they get pushed to the topic and executes the trades for the customers contained in the messages by calling the exchange. As mentioned above, a single customer's trades are only ever handled by the same cluster of workers, which makes our logic and our SQL queries cleaner.

As far as how many topics and clusters of workers we'll need, we can do some rough estimation. If we plan to execute millions of trades per day, that comes down to about 10-100 trades per second given open trading hours during a third of a day and non-uniform trading patterns. If we assume that the core execution logic lasts about a second, then we should have roughly 10-100 topics and clusters of workers to process trades in parallel.

```
~100,000 seconds per day (3600 * 24)

~1,000,000 trades per day

trades bunched in 1/3rd of the day

--> (1,000,000 / 100,000) * 3 = ~30 trades per second
```

6. Trade-Execution Logic

The subscribers (our workers) are streaming / waiting for messages. Imagine the following message were to arrive in the topic queue:

```
{"customerId": "c1"}
```

The following would be pseudo-code for the worker logic:

```
// We get the oldest trade that isn't in a terminal state.

trade = SELECT * FROM trades WHERE

    customer_id = 'c1' AND

    (status = 'PLACED' OR status = 'IN PROGRESS')

    ORDER BY created_at ASC LIMIT 1;



// If the trade is PLACED, we know that it's effectively

// ready to be executed. We set it as IN PROGRESS.
```

```
if trade.status == "PLACED" {

    UPDATE trades SET status = 'IN PROGRESS' WHERE id = trade.id;

}



// In the event that the trade somehow already exists in the

// exchange, the callback will do the work for us.

if exchange.TradeExists(trade.id) {

    return;

}



// We get the balance for the customer.

balance = SELECT amount FROM balances WHERE

    customer_id = 'c1';



// This is the callback that the exchange will execute once

// the trade actually completes. We'll define it further down

// in the walkthrough.

callback = ...
```

```
exchange.Execute(
    trade.stockTicker,
    trade.type,
    trade.quantity,
    max_price = balance,
    callback,
)
```

7. Exchange Callback

Below is some pseudo code for the exchange callback:

```
function exchange_callback(exchange_trade) {
    if exchange_trade.status == 'FILLED' {
        BEGIN TRANSACTION;
        trade = SELECT * FROM trades WHERE id = database_trade.id;
        if trade.status <> 'IN PROGRESS' {
            ROLLBACK;
            pubsub.send({customer_id: database_trade.customer_id});
            return;

        }

        UPDATE balances SET amount -= exchange_trade.amount WHERE customer_id =
database_trade.customer_id;

        UPDATE trades SET status = 'FILLED' WHERE id = database_trade.id;

        COMMIT;

    } else if exchange_trade.status == 'REJECTED' {

        BEGIN TRANSACTION;

        UPDATE trades SET status = 'REJECTED' WHERE id = database_trade.id;

        UPDATE trades SET reason = exchange_trade.reason WHERE id = database_trade.id;

        COMMIT;

    }

    pubsub.send({customer_id: database_trade.customer_id});

    return http.status(200);

}
```

Design Amazon

Design the system that powers Amazon's e-commerce business.

Many systems design questions are intentionally left very vague and are literally given in the form of `Design Foobar`. It's your job to ask clarifying questions to better understand the system that you have to build.

We've laid out some of these questions below; their answers should give you some guidance on the problem. Before looking at them, we encourage you to take few minutes to think about what questions you'd ask in a real interview.

Solution

Our solution walkthroughs are meant to supplement our video solutions. We recommend starting with the video solution and using the walkthrough either as a point of reference while you watch the video or as a review tool if you need to brush up on this question's solution later on.

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 e-commerce side of the Amazon website, and more specifically, the system that supports users searching for items on the Amazon home page, adding items to cart, submitting orders, and those orders being assigned to relevant Amazon warehouses for shipment.

We need to handle items going out of stock, and we've been given some guidelines for a simple "stock-reservation" system when users begin the checkout process.

We have access to two smart services: one that handles user search queries and one that handles warehouse order assignment. It's our job to figure out how these services fit into our larger design.

We'll specifically be designing the system that supports amazon.com (i.e., Amazon's U.S. operations), and we'll assume that this system can be replicated for other regional Amazon stores. For the rest of this walkthrough, whenever we refer to "Amazon," we'll be referring specifically to Amazon's U.S. store.

While the system should have low latency when searching for items and high availability in general, serving roughly 10 orders per second in the U.S., we've been told to focus mostly on core functionality.

2. Coming Up With A Plan

We'll tackle this system by first looking at a high-level overview of how it'll be set up, then diving into its storage components, and finally looking at how the core functionality comes to life. We can divide the core functionality into two main sections:

- *The user side.*
- *The warehouse side.*

We can further divide the user side as follows:

- *Browsing items given a search term.*
- *Modifying the cart.*
- *Beginning the checkout process.*
- *Submitting and canceliing orders.*

3. High-Level System Overview

Within a region, user and warehouse requests will get round-robin-load-balanced to respective sets of API servers, and data will be written to and read from a SQL database for that region.

We'll go with a SQL database because all of the data that we'll be dealing with (items, carts, orders, etc.) is, by nature, structured and lends itself well to a relational model.

4. SQL Tables

We'll have six SQL tables to support our entire system's storage needs.

Items

This table will store all of the items on Amazon, with each row representing an item.

itemId: uuid	***name: string***	***description: string***	***price: integer***	***currency: enum***	***other. ..***
...	...	...	...	...	...

Carts

This table will store all of the carts on Amazon, with each row representing a cart. We've been told that each user can only have a single cart at once.

cartId: uuid	***customerId: uuid***	***items: []{itemId, quantity}***
...	...	...

Orders

This table will store all of the orders on Amazon, with each row representing an order.

orderId: uuid	***customerId: uuid***	***orderStatus: enum***	***items: []{itemId, quantity}***	***price: integer***	***paymentInfo: PaymentInfo***	***shippingAddress: string***	***timestamp: datetime***	***other...***
...	...	...	...	...	...	...	...	...

Aggregated Stock

*This table will store all of the item stocks on Amazon that are relevant to users, with each row representing an item. See the **Core User Functionality** section for more details.*

itemId: uuid	***stock: integer***
...	...

Warehouse Orders

This table will store all of the orders that Amazon warehouses get, with each row representing a warehouse order. Warehouse orders are either entire normal Amazon orders or subsets of normal Amazon orders.

warehouseOrderId: uuid	***parentOrderId: uuid***	***warehouseId: uuid***	***orderStatus: enum***	***items: []{itemId, quantity}***	***shippingAddress: string***
...	...	...	...	...	...

Warehouse Stock

This table will store all of the item stocks in Amazon warehouses, with each row representing an {item, warehouse} pairing. The physicalStock field represents an item's actual physical stock in the warehouse in question, serving as a source of truth, while the availableStock field represents an item's effective available stock in the relevant warehouse; this stock gets decreased when orders are assigned to warehouses. See the ***Core Warehouse Functionality*** *section for more details.*

itemId: uuid	***warehouseId: uuid***	***physicalStock: integer***	***availableStock: integer***
...	...	...	...

5. Core User Functionality

GetItemCatalog(search)

*This is the endpoint that users call when they're searching for items. The request is routed by API servers to the smart search-results service, which interacts directly with the **items** table, caches popular item searches, and returns the results.*

*The API servers also fetch the relevant item stocks from the **aggregated_stock** table.*

UpdateCartItemQuantity(itemId, quantity)

*This is the endpoint that users call when they're adding or removing items from their cart. The request writes directly to the **carts** table, and users can only call this endpoint when an item has enough stock in the **aggregated_stock** table.*

BeginCheckout() & CancelCheckout()

*These are the endpoints that users call when they're beginning the checkout process and cancelling it. The BeginCheckout request triggers another read of the **aggregated_stock** table for the relevant items. If some of the items in the cart don't have enough stock anymore, the UI alerts the users accordingly. For items that do have enough stock, the API servers write to the **aggregated_stock** table and decrease the relevant stocks accordingly, effectively "reserving" the items during the duration of the checkout. The CancelCheckout request, which also gets automatically called after 10 minutes of being in the checkout process, writes to the **aggregated_stock** table and increases the relevant stocks accordingly, thereby "unreserving" the items. Note that all of the writes to the **aggregated_stock** are ACID transactions, which allows us to comfortably rely on this SQL table as far as stock correctness is concerned.*

SubmitOrder(), CancelOrder(), & GetMyOrders()

These are the endpoints that users call when they're submitting and cancelling orders. Both the SubmitOrder and CancelOrder requests write to the ***orders*** *table, and CancelOrder also writes to the* ***aggregated_stock*** *table, increasing the relevant stocks accordingly (SubmitOrder doesn't need to because the checkout process already has). GetMyOrders simply reads from the* ***orders*** *table. Note that an order can only be cancelled if it hasn't yet been shipped, which is knowable from the orderStatus field.*

6. Core Warehouse Functionality

On the warehouse side of things, we'll have the smart order-assignment service read from the ***orders*** *table, figure out the best way to split orders up and assign them to warehouses based on shipping addresses, item stocks, and other data points, and write the final warehouse orders to the* ***warehouse_orders*** *table.*

In order to know which warehouses have what items and how many, the order-assignment service will rely on the availableStock of relevant items in the ***warehouse_stock*** *table. When the service assigns an order to a warehouse, it decreases the availableStock of the relevant items for the warehouse in question in the* ***warehouse_stock*** *table. These availableStock values are re-increased by the relevant warehouse if its order ends up being cancelled.*

When warehouses get new item stock, lose item stock for whatever reason, or physically ship their assigned orders, they'll update the relevant physicalStock values in the ***warehouse_stock*** *table. If they get new item stock or lose item stock, they'll also write to the* ***aggregated_stock*** *table (they don't need to do this when shipping assigned orders, since the* ***aggregated_stock*** *table already gets updated by the checkout process on the user side of things).*

Design The Reddit API

Design an API for Reddit subreddits given the following information.

The API includes these 2 entities:

- ***User*** **|** *userId: string, ...*
- ***SubReddit*** **|** *subredditId: string, ...*

Both of these entities likely have other fields, but for the purpose of this question, those other fields aren't needed.

Your API should support the basic functionality of a subreddit on Reddit.

Many systems design questions are intentionally left very vague and are literally given in the form of `Design Foobar`. It's your job to ask clarifying questions to better understand the system that you have to build.

We've laid out some of these questions below; their answers should give you some guidance on the problem. Before looking at them, we encourage you to take few minutes to think about what questions you'd ask in a real interview.

Solution

Our solution walkthroughs are meant to supplement our video solutions. We recommend starting with the video solution and using the walkthrough either as a point of reference while you watch the video or as a review tool if you need to brush up on this question's solution later on.

1. Gathering Requirements

As with any API design interview question, the first thing that we want to do is to gather API requirements; we need to figure out what API we're building exactly.

*We're designing the core user flow of the **subreddit** functionality on Reddit. Users can write posts on subreddits, they can comment on posts, and they can upvote / downvote posts and comments.*

We're going to be defining three primary entities: Posts, Comments, and Votes, as well as their respective CRUD operations.

We're also going to be designing an API for buying and giving awards on Reddit.

2. Coming Up With A Plan

It's important to organize ourselves and to lay out a clear plan regarding how we're going to tackle our design. What are the major, potentially contentious parts of our API? Why are we making certain design decisions?

The first major point of contention is whether to store votes only on Comments and Posts, and to cast votes by calling the EditComment and EditPost methods, or whether to store them as entirely separate entities--siblings of Posts and Comments, so to speak. Storing them as separate entities makes it much more straightforward to edit or remove a particular user's votes (by just calling EditVote, for instance), so we'll go with this approach.

We can then plan to tackle Posts, Comments, and Votes in this order, since they'll likely share some common structure.

3. Posts

Posts will have an id, the id of their creator (i.e., the user who writes them), the id of the subreddit that they're on, a description and a title, and a timestamp of when they're created.

Posts will also have a count of their votes, comments, and awards, to be displayed on the UI. We can imagine that some backend service will be calculating or updating these numbers when some of the Comment, Vote, and Award CRUD operations are performed.

Lastly, Posts will have optional ***deletedAt*** *and* ***currentVote*** *fields. Subreddits display posts that have been removed with a special message; we can use the* ***deletedAt*** *field to accomplish this. The* ***currentVote*** *field will be used to display to a user whether or not they've cast a vote on a post. This field will likely be populated by the backend upon fetching Posts or when casting Votes.*

- *postId: string*
- *creatorId: string*
- *subredditId: string*
- *title: string*
- *description: string*
- *createdAt: timestamp*
- *votesCount: int*
- *commentsCount: int*
- *awardsCount: int*
- *deletedAt?: timestamp*
- *currentVote?: enum UP/DOWN*

Our CreatePost, EditPost, GetPost, and DeletePost methods will be very straightforward. One thing to note, however, is that all of these operations will take in the ***userId*** *of the user performing them; this id, which will likely contain authentication information, will be used for ACL checks to see if the user performing the operations has the necessary permission(s) to do so.*

```
CreatePost(userId: string, subredditId: string, title: string, description: string)

  => Post

EditPost(userId: string, postId: string, title: string, description: string)

  => Post

GetPost(userId: string, postId: string)

  => Post

DeletePost(userId: string, postId: string)

  => Post
```

Since we can expect to have hundreds, if not thousands, of posts on a given subreddit, our ListPosts method will have to be paginated. The method will take in optional ***pageSize*** *and* ***pageToken*** *parameters and will return a list of posts of at most length* ***pageSize*** *as well as a* ***nextPageToken****--the token to be fed to the method to retrieve the next page of posts.*

```
ListPosts(userId: string, subredditId: string, pageSize?: int, pageToken?: string)

  => (Post[], nextPageToken?)
```

4. Comments

Comments will be similar to Posts. They'll have an id, the id of their creator (i.e., the user who writes them), the id of the post that they're on, a content string, and the same other fields as Posts have. The only difference is that Comments will also have an optional ***parentId*** *pointing to the parent post or parent comment of the comment. This id will allow the Reddit UI to reconstruct Comment trees to properly display (indent) replies. The UI can also sort comments within a reply thread by their* ***createdAt*** *timestamps or by their* ***votesCount****.*

- *commentId: string*
- *creatorId: string*
- *postId: string*
- *createdAt: timestamp*
- *content: string*
- *votesCount: int*
- *awardsCount: int*
- *parentId?: string*
- *deletedAt?: timestamp*
- *currentVote?: enum UP/DOWN*

Our CRUD operations for Comments will be very similar to those for Posts, except that the CreateComment method will also take in an optional ***parentId*** *pointing to the comment that it's replying to, if relevant.*

```
CreateComment(userId: string, postId: string, content: string, parentId?: string)

  => Comment



EditComment(userId: string, commentId: string, content: string)

  => Comment

GetComment(userId: string, commentId: string)

  => Comment



DeleteComment(userId: string, commentId: string)

  => Comment

ListComments(userId: string, postId: string, pageSize?: int, pageToken?: string)

  => (Comment[], nextPageToken?)
```

5. Votes

Votes will have an id, the id of their creator (i.e., user who casts them), the id of their target (i.e., the post or comment that they're on), and a type, which will be a simple UP/DOWN enum. They could also have a ***createdAt*** *timestamp for good measure.*

- *voteId: string*
- *creatorId: string*
- *targetId: string*
- *type: enum UP/DOWN*
- *createdAt: timestamp*

Since it doesn't seem like getting a single vote or listing votes would be very useful for our feature, we'll skip designing those endpoints (though they would be straightforward).

Our CreateVote, EditVote, and DeleteVote methods will be simple and useful. The CreateVote method will be used when a user casts a new vote on a post or comment; the EditVote method will be used when a user has already cast a vote on a post or comment and casts the opposite vote on that same post or comment; and the DeleteVote method will be used when a user has already cast a vote on a post or comment and just removes that same vote.

```
CreateVote(userId: string, targetId: string, type: enum UP/DOWN)

  => Vote
```

```
EditVote(userId: string, voteId: string, type: enum UP/DOWN)

  => Vote

DeleteVote(userId: string, voteId: string)

  => Vote
```

6. Awards

We can define two simple endpoints to handle buying and giving awards. The endpoint to buy awards will take in a ***paymentToken****, which will be a string that contains all of the necessary* *information to process a payment. The endpoint to give an award will take in a **targetId**, which will be the id of the post or comment that the award is being given to.*

BuyAwards(userId: string, paymentToken: string, quantity: int)

GiveAward(userId: string, targetId: string)

Design Facebook News Feed

Many systems design questions are intentionally left very vague and are literally given in the form of `Design Foobar`. It's your job to ask clarifying questions to better understand the system that you have to build.

We've laid out some of these questions below; their answers should give you some guidance on the problem. Before looking at them, we encourage you to take few minutes to think about what questions you'd ask in a real interview.

Solution

Our solution walkthroughs are meant to supplement our video solutions. We recommend starting with the video solution and using the walkthrough either as a point of reference while you watch the video or as a review tool if you need to brush up on this question's solution later on.

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 user flow of the **Facebook News Feed**. This consists of loading a user's news feed, scrolling through the list of posts that are relevant to them, posting status updates, and having their friends' news feeds get updated in real time. We're Specifically designing the pipeline that generates and serves news feeds and the system that handles what happens when a user posts and news feeds have to be updated.*

We're dealing with about 1 billion users, each with 500 friends on average.

Getting a news feed should feel fairly instant, and creating a post should update all of a user's friends' news feeds within a minute. We can have some variance with regards to feed updates depending on user locations.

Additionally, we can't be satisfied with a single cluster serving everyone on earth because of large ***latencies*** *that would occur between that cluster and the user in some parts of the world, so we need a mechanism to make sure the feed gets updated within a minute in the regions other than the one the post was created in.*

We can assume that the ranking algorithms used to generate news feeds with the most relevant posts is taken care of for us by some other system that we have access to.

2. Coming Up With A Plan

We'll start with the extremities of our system and work inward, first talking about the two API calls, CreatePost and GetNewsFeed, then, getting into the feed creation and storage strategy, our cross-region design, and finally tying everything together in a fast and scalable way.

3. CreatePost API

For the purpose of this design, the CreatePost API call will be very simple and look something like this:

```
  CreatePost(

      user_id: string,

      post: data

   )
```

When a user creates a post, the API call goes through some load balancing before landing on one of many API servers (which are stateless). Those API servers then create a message on a Pub/Sub topic, notifying its subscribers of the new post that was just created. Those subscribers will do a few things, so let's call them S1 for future reference. Each of the subscribers S1 reads from the topic and is responsible for creating the facebook post inside a relational database.

4. Post Storage

We can have one main relational database to store most of our system's data, including posts and users. This database will have very large tables.

5. GetNewsFeed API

The GetNewsFeed API call will most likely look like this:

```
  GetNewsFeed(
      user_id: string,
      pageSize: integer,
      nextPageToken: integer,
  ) => (
      posts: []{
         user_id: string,
         post_id: string,
         post: data,
      },

      nextPageToken: string,

  )
```

The pageSize and nextPageToken fields are used to ***paginate*** *the newsfeed; pagination is necessary when dealing with large amounts of listed data, and since we'll likely want each news feed to have up to 1000 posts, pagination is very appropriate here.*

6. Feed Creation And Storage

Since our databases tables are going to be so large, with billions of millions of users and tens of millions of posts every week, fetching news feeds from our main database every time a GetNewsFeed call is made isn't going to be ideal. We can't expect low latencies when building news feeds from scratch because querying our huge tables takes time, and sharding the main database holding the posts wouldn't be particularly helpful since news feeds would likely need to aggregate posts across shards, which would require us to perform cross-shard joins when generating news feeds; we want to avoid this.

Instead, we can store news feeds separately from our main database across an array of shards. We can have a separate cluster of machines that can act as a proxy to the relational database and be in charge of aggregating posts, ranking them via the ranking algorithm that we're given, generating news feeds, and sending them to our shards every so often (every 5, 10, 60 minutes, depending on how often we want news feeds to be updated).

If we average each post at 10kB, and a newsfeed comprises of the top 1000 posts that are relevant to a user, that's 10MB per user, or ***10 000TB*** *of data total. We assume that it's loaded 10 times per day per user, which averages at* ***10k QPS*** *for the newsfeed fetching.*

Assuming 1 billion news feeds (for 1 billion users) containing 1000 posts of up to 10 KB each, we can estimate that we'll need 10 PB (petabytes) of storage to store all of our users' news feeds. We can use 1000 machines of 10 TB each as our news-feed shards.

~10 KB per post

~1000 posts per news feed

~1 billion news feeds

*~10 KB * 1000 * 1000^3 = 10 PB = 1000 * 10 TB*

To distribute the newsfeeds roughly evenly, we can shard based on the user id.

When a GetNewsFeed request comes in, it gets load balanced to the right news feed machine, which returns it by reading on local disk. If the newsfeed doesn't exist locally, we then go to the source of truth (the main database, but going through the proxy ranking service) to gather the relevant posts. This will lead to increased latency but shouldn't happen frequently.

7. Wiring Updates Into Feed Creation

We now need to have a notification mechanism that lets the feed shards know that a new relevant post was just created and that they should incorporate it into the feeds of impacted users.

We can once again use a Pub/Sub service for this. Each one of the shards will subscribe to its own topic--we'll call these topics the Feed Notification Topics (FNT)--and the original subscribers S1 will be the publishers for the FNT. When S1 gets a new message about a post creation, it searches the main database for all of the users for whom this post is relevant (i.e., it searches for all of the friends of the user who created the post), it filters out users from other regions who will be taken care of asynchronously, and it maps the remaining users to the FNT using the same hashing function that our GetNewsFeed load balancers rely on.

For posts that impact too many people, we can cap the number of FNT topics that get messaged to reduce the amount of internal traffic that gets generated from a single post. For those big users we can rely on the asynchronous feed creation to eventually kick in and let the post appear in feeds of users whom we've skipped when the feeds get refreshed manually.

8. Cross-Region Strategy

When CreatePost gets called and reaches our Pub/Sub subscribers, they'll send a message to another Pub/Sub topic that some forwarder service in between regions will subscribe to. The forwarder's job will be, as its name implies, to forward messages to other regions so as to replicate all of the CreatePost logic in other regions. Once the forwarder receives the message, it'll essentially mimic what would happen if that same CreatePost were called in another region, which will start the entire feed-update logic in those other regions. We can have some additional logic passed to the forwarder to prevent other regions being replicated to from notifying other regions about the CreatePost call in question, which would lead to an infinite chain of replications; in other words, we can make it such that only the region where the post originated from is in charge of notifying other regions.

Several open-source technologies from big companies like Uber and Confluent are designed in part for this kind of operation.

Design Google Drive

Many systems design questions are intentionally left very vague and are literally given in the form of `Design Foobar`. It's your job to ask clarifying questions to better understand the system that you have to build.

We've laid out some of these questions below; their answers should give you some guidance on the problem. Before looking at them, we encourage you to take few minutes to think about what questions you'd ask in a real interview.

Solution

Our solution walkthroughs are meant to supplement our video solutions. We recommend starting with the video solution and using the walkthrough either as a point of reference while you watch the video or as a review tool if you need to brush up on this question's solution later on.

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 user flow of the ***Google Drive*** *web application. This consists of storing two main entities: folders and files. More specifically, the system should allow users to create folders, upload and download files, and rename and move entities once they're stored. We don't have to worry about ACLs, sharing entities, or any other auxiliary Google Drive features.*

We're going to be building this system at a very large scale, assuming 1 billion users, each with ***15GB*** *of data stored in Google Drive on average. This adds up to approximately* ***15,000 PB*** *of data in total, without counting any metadata that we might store for each entity, like its name or its type.*

We need this service to be ***Highly Available*** *and also very redundant. No data that's successfully stored in Google Drive can ever be lost, even through catastrophic failures in an entire region of the world.*

2. Coming Up With A Plan

It's important to organize ourselves and to lay out a clear plan regarding how we're going to tackle our design. What are the major, distinguishable components of our how system?

First of all, we'll need to support the following operations:

- *For* ***Files***
  - *UploadFile*

    - *DownloadFile*
    - *DeleteFile*
    - *RenameFile*
    - *MoveFile*
- *For **Folders***
    - *CreateFolder*
    - *GetFolder*
    - *DeleteFolder*
    - *RenameFolder*
    - *MoveFolder*

Secondly, we'll have to come up with a proper storage solution for two types of data:

- *File Contents: The contents of the files uploaded to Google Drive. These are opaque bytes with no particular structure or format.*
- *Entity Info: The metadata for each entity. This might include fields like **entityID, ownerID, lastModified, entityName, entityType**. This list is non-exhaustive, and we'll most likely add to it later on.*

Let's start by going over the storage solutions that we want to use, and then we'll go through what happens when each of the operations outlined above is performed.

3. Storing Entity Info

To store entity information, we can use key-value stores. Since we need high availability and data replication, we need to use something like Etcd, Zookeeper, or Google Cloud Spanner (as a K-V store) that gives us both of those guarantees as well as consistency (as opposed to DynamoDB, for instance, which would give us only eventual consistency).

Since we're going to be dealing with many gigabytes of entity information (given that we're serving a billion users), we'll need to shard this data across multiple clusters of these K-V stores. Sharding on entityID means that we'll lose the ability to perform batch operations, which these key-value stores give us out of the box and which we'll need when we move entities around (for instance, moving a file from one folder to another would involve editing the metadata of 3 entities; if they were located in 3 different shards that wouldn't be great). Instead, we can shard based on the ***ownerID*** *of the entity, which means that we can edit the metadata of multiple entities atomically with a transaction, so long as the entities belong to the same user.*

Given the traffic that this website needs to serve, we can have a layer of proxies for entity information, load balanced on a hash of the ***ownerID****. The proxies could have some caching, as well as perform* ***ACL*** *checks when we eventually decide to support them. The proxies would live at the regional level, whereas the source-of-truth key-value stores would be accessed globally.*

4. Storing File Data

When dealing with potentially very large uploads and data storage, it's often advantageous to split up data into blobs that can be pieced back together to form the original data. When uploading a file, the request will be load balanced across multiple servers that we'll call "blob splitters", and these blob splitters will have the job of splitting files into blobs and storing these blobs in some global blob-storage solution like ***GCS*** *or* ***S3*** *(since we're designing* ***Google*** *Drive, it might not be a great idea to pick S3 over GCS :P).*

One thing to keep in mind is that we need a lot of redundancy for the data that we're uploading in order to prevent data loss. So we'll probably want to adopt a strategy like: try pushing to 3 different GCS ***buckets*** *and consider a write successful only if it went through in at least 2 buckets. This way we always have redundancy without necessarily sacrificing availability. In the background, we can have an extra service in charge of further replicating the data to other buckets in an async manner. For our main 3 buckets, we'll want to pick buckets in 3 different availability zones to avoid having all of our redundant storage get wiped out by potential catastrophic failures in the event of a natural disaster or huge power outage.*

*In order to avoid having multiple identical blobs stored in our blob stores, we'll name the blobs after a hash of their content. This technique is called **Content-Addressable Storage**, and by using it, we essentially make all blobs immutable in storage. When a file changes, we simply upload the entire new resulting blobs under their new names computed by hashing their new contents.*

This immutability is very powerful, in part because it means that we can very easily introduce a caching layer between the blob splitters and the buckets, without worrying about keeping caches in sync with the main source of truth when edits are made--an edit just means that we're dealing with a completely different blob.

5. Entity Info Structure

*Since folders and files will both have common bits of metadata, we can have them share the same structure. The difference will be that folders will have an **is_folder** flag set to true and a list of **children_ids**, which will point to the entity information for the folders and files within the folder in question. Files will have an **is_folder** flag set to false and a **blobs** field, which will have the IDs of all of the blobs that make up the data within the relevant file. Both entities can also have a **parent_id** field, which will point to the entity information of the entity's parent folder. This will help us quickly find parents when moving files and folders.*

- *File Info*

```
{

  blobs: ['blob_content_hash_0', 'blob_content_hash_1'],

  id: 'some_unique_entity_id'

  is_folder: false,

  name: 'some_file_name',
  owner_id: 'id_of_owner',

  parent_id: 'id_of_parent',
```

- *}Folder Info*

```
{

  children_ids: ['id_of_child_0', 'id_of_child_1'],

  id: 'some_unique_entity_id'

  is_folder: true,

  name: 'some_folder_name',

  owner_id: 'id_of_owner',

  parent_id: 'id_of_parent',

}
```

6. Garbage Collection

Any change to an existing file will create a whole new blob and de-reference the old one. Furthermore, any deleted file will also de-reference the file's blobs. This means that we'll eventually end up with a lot of ***orphaned*** *blobs that are basically unused and taking up storage for no reason. We'll need a way to get rid of these blobs to free some space.*

We can have a ***Garbage Collection*** *service that watches the entity-info K-V stores and keeps counts of the number of times every blob is referenced by files; these counts can be stored in a SQL table.*

Reference counts will get updated whenever files are uploaded and deleted. When the reference count for a particular blob reaches 0, the Garbage Collector can mark the blob in question as orphaned in the relevant blob stores, and the blob will be safely deleted after some time if it hasn't been accessed.

7. End To End API Flow

Now that we've designed the entire system, we can walk through what happens when a user performs any of the operations we listed above.

CreateFolder is simple; since folders don't have a blob-storage component, creating a folder just involves storing some metadata in our key-value stores.

UploadFile works in two steps. The first is to store the blobs that make up the file in the blob storage. Once the blobs are persisted, we can create the file-info object, store the blob-content hashes inside its ***blobs*** *field, and write this metadata to our key-value stores.*

DownloadFile fetches the file's metadata from our key-value stores given the file's ID. The metadata contains the hashes of all of the blobs that make up the content of the file, which we can use to fetch all of the blobs from blob storage. We can then assemble them into the file and save it onto local disk.

All of the Get, Rename, Move, and Delete operations atomically change the metadata of one or several entities within our key-value stores using the ***transaction*** *guarantees that they give us.*

Design Netflix

Many systems design questions are intentionally left very vague and are literally given in the form of `Design Foobar`. It's your job to ask clarifying questions to better understand the system that you have to build.

We've laid out some of these questions below; their answers should give you some guidance on the problem. Before looking at them, we encourage you to take few minutes to think about what questions you'd ask in a real interview.

Solution

Our solution walkthroughs are meant to supplement our video solutions. We recommend starting with the video solution and using the walkthrough either as a point of reference while you watch the video or as a review tool if you need to brush up on this question's solution later on.

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 Netflix service, which allows users to stream movies and shows from the Netflix website.

Specifically, we'll want to focus on:

- ***Delivering large amounts of high-definition video content to hundreds of millions of users around the globe without too much buffering.***
- ***Processing large amounts of user-activity data to support Netflix's recommendation engine.***

2. Coming Up With A Plan

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

- ***Storage (Video Content, Static Content, and User Metadata)***
- ***General Client-Server Interaction (i.e., the life of a query)***
- ***Video Content Delivery***
- ***User-Activity Data Processing***

3. Video-Content Storage

Since Netflix's service, which caters to millions of customers, is centered around video content, we might need a lot of storage space and a complex storage solution. Let's start by estimating how much space we'll need.

We were told that Netflix has about 200 million users; we can make a few assumptions about other Netflix metrics (alternatively, we can ask our interviewer for guidance here):

- ***Netflix offers roughly 10 thousand movies and shows at any given time***
- ***Since movies can be up to 2+ hours in length and shows tend to be between 20 and 40 minutes per episode, we can assume an average video length of 1 hour***
- ***Each movie / show will have a Standard Definition version and a High Definition version. Per hour, SD will take up about 10GB of space, while HD will take about 20GB.***

~10K videos (stored in SD & HD)

~1 hour average video length

~10 GB/h for SD + ~20 GB/h for HD = 30 GB/h per video

~30 GB/h * 10K videos = 300,000 GB = 300 TB

This number highlights the importance of estimations. Naively, one might think that Netflix stores many petabytes of video, since its core product revolves around video content; but a simple back-of-the-napkin estimation shows us that it actually stores a very modest amount of video.

This is because Netflix, unlike other servies like YouTube, Google Drive, and Facebook, has a bounded amount of video content: the movies and shows that it offers; those other services allow users to upload unlimited amounts of video.

Since we're only dealing with a few hundred terabytes of data, we can use a simple blob storage solution like S3 or GCS to reliably handle the storage and replication of Netflix's video content; we don't need a more complex data-storage solution.

4. Static-Content Storage

Apart from video content, we'll want to store various pieces of static content for Netflix's movies and shows, including video titles, descriptions, and cast lists.

This content will be bounded in size by the size of the video content, since it'll be tied to the number of movies and shows, just like the video content, and since it'll naturally take up less space than the video data.

We can easily store all of this static content in a relational database or even in a document store, and we can cache most of it in our API servers.

5. User Metadata Storage

We can expect to store some user metadata for each video on the Netflix platform. For instance, we might want to store the timestamp that a user left a video at, a user's rating on a video, etc..

Just like the static content mentioned above, this user metadata will be tied to the number of videos on Netflix. However, unlike the static content, this user metadata will grow with the Netflix userbase, since each user will have user metadata.

We can quickly estimate how much space we'll need for this user metadata:

~200M users

~1K videos watched per user per lifetime (~10% of total content)

~100 bytes/video/user

~100 bytes * 1K videos * 200M users = 100 KB * 200M = 1 GB * 20K = 20 TB

Perhaps surprisingly, we'll be storing an amount of user metadata in the same ballpark as the amount of video content that we'll be storing. Once again, this is because of the bounded nature of Netflix's video content, which is in stark contrast with the unbounded nature of its userbase.

We'll likely need to query this metadata, so storing it in a classic relational database like Postgres makes sense.

Since Netflix users are effectively isolated from one another (they aren't connected like they would be on a social-media platform, for example), we can expect all of our latency-sensitive database operations to only relate to individual users. In other words, potential operations like GetUserInfo and GetUserWatchedVideos, which would require fast latencies, are specific to a particular users; on the other hand, complicated database operations involving multiple users' metadata will likely be part of background data-engineering jobs that don't care about latency.

Given this, we can split our user-metadata database into a handful of shards, each managing anywhere between 1 and 10 TB of indexed data. This will maintain very quick reads and writes for a given user.

6. General Client-Server Interaction

The part of the system that handles serving user metadata and static content to users shouldn't be too complicated.

We can use some simple round-robin load balancing to distribute end-user network requests across our API servers, which can then load-balance database requests according to userId (since our database will be sharded based on userId).

As mentioned above, we can cache our static content in our API servers, periodically updating it when new movies and shows are released, and we can even cache user metadata there, using a write-through caching mechanism.

7. Video Content Delivery

We need to figure out how we'll be delivering Netflix's video content across the globe with little latency. To start, we'll estimate the maximum amount of bandwidth consumption that we could expect at any point in time. We'll assume that, at peak traffic, like when a popular movie comes out, a fairly large number of Netflix users might be streaming video content concurrently.

~200M total users

~5% of total users streaming concurrently during peak hours

~20 GB/h of HD video ~= 5 MB/s of HD video

~5% of 200M * 5 MB/s = 10M * 5 MB/s = 50 TB/s

This level of bandwidth consumption means we can't just naively serve the video content out of a single data center or even dozens of data centers. We need many thousands of locations around the world to be distributing this content for us. Thankfully, CDNs solve this precise problem, since they have many thousands of Points of Presence around the world. We can thus use a CDN like Cloudflare and serve our video content out of the CDN's PoPs.

Since the PoPs can't keep the entirety of Netflix's video content in cache, we can have an external service that periodically repopulates CDN PoPs with the most important content (the movies and shows most likely to be watched).

8. User-Activity Data Processing

We need to figure out how we'll process vast amounts of user-activity data to feed into Netflix's recommendation engine. We can imagine that this user-activity data will be gathered in the form of logs that are generated by all sorts of user actions; we can expect terabytes of these logs to be generated every day.

MapReduce can help us here. We can store the logs in a distributed file system like HDFS and run MapReduce jobs to process massive amounts of data in parallel. The results of these jobs can then be fed into some machine learning pipelines or simply stored in a database.

Map Inputs

Our Map inputs can be our raw logs, which might look like:

```
{"userId": "userId1", "videoId": "videoId1", "event": "CLICK"}

{"userId": "userId2", "videoId": "videoId2", "event": "PAUSE"}

{"userId": "userId3", "videoId": "videoId3", "event": "MOUSE_MOVE"}
```

Map Outputs / Reduce Inputs

Our Map function will aggregate logs based on userId and return intermediary key-value pairs indexed on each userId, pointing to lists of tuples with videoIds and relevant events.

These intermediary k/v pairs will be shuffled appropriately and fed into our Reduce functions.

{"userId1": [("CLICK", "videoId1"), ("CLICK", "videoId1"), ..., ("PAUSE", "videoId2")]}

{"userId2": [("PLAY", "videoId1"), ("MOUSE_MOVE", "videoId2"), ..., ("MINIMIZE", "videoId3")]}

Reduce Outputs

Our Reduce functions could return many different outputs. They could return k/v pairs for each userId|videoId combination, pointing to a computed score for that user/video pair; they could return k/v pairs indexed at each userId, pointing to lists of (videoId, score) tuples; or they could return k/v pairs also indexed at eacher userId but pointing to stack-rankings of videoIds, based on their computed score.

("userId1|videoId1", score)

("userId1|videoId2", score)

OR

{"userId1": [("videoId1", score), ("videoId2", score), ..., ("videoId3", score)]}

{"userId2": [("videoId1", score), ("videoId2", score), ..., ("videoId3", score)]}

OR

```
("userId1", ["videoId1", "videoId2", ..., "videoId3"])

("userId2", ["videoId1", "videoId2", ..., "videoId3"])
```

Design The Uber API

Many systems design questions are intentionally left very vague and are literally given in the form of `Design Foobar`. It's your job to ask clarifying questions to better understand the system that you have to build.

We've laid out some of these questions below; their answers should give you some guidance on the problem. Before looking at them, we encourage you to take few minutes to think about what questions you'd ask in a real interview.

Our solution walkthroughs are meant to supplement our video solutions. We recommend starting with the video solution and using the walkthrough either as a point of reference while you watch the video or as a review tool if you need to brush up on this question's solution later on.

Solution

1. Gathering Requirements

As with any API design interview question, the first thing that we want to do is to gather API requirements; we need to figure out what API we're building exactly.

We're designing the core ride-hailing service that Uber offers. Passengers can book a ride from their phone, at which point they're matched with a driver; they can track their driver's location throughout the ride, up until the ride is finished or canceled; and they can also see the price of the ride as well as the estimated time to destination throughout the trip, amongst other things.

The core taxiing service that Uber offers has a passenger-facing side and a driver-facing side; we're going to be designing the API for both sides.

2. Coming Up With A Plan

It's important to organize ourselves and to lay out a clear plan regarding how we're going to tackle our design. What are the major, potentially contentious parts of our API? Why are we making certain design decisions?

We're going to center our API around a Ride entity; every Uber ride will have an associated Ride containing information about the ride, including information about its passenger and driver.

Since a normal Uber ride can only have one passenger (one passenger account--the one that hails the ride) and one driver, we're going to cleverly handle all permissioning related to ride operations through passenger and driver IDs. In other words, operations like GetRide and EditRide will purely rely on a passed userId, the userId of the passenger or driver calling them, to return the appropriate ride tied to that passenger or driver.

We'll start by defining the Ride entity before designing the passenger-facing API and then the driver-facing API.

3. Entities

Ride

The Ride entity will have a unique id, info about its passenger and its driver, a status, and other details about the ride.

- ***rideId: string***
- ***passengerInfo: PassengerInfo***
- ***driverInfo?: DriverInfo***

- *rideStatus: RideStatus - enum CREATED/MATCHED/STARTED/FINISHED/CANCELED*
- *start: GeoLocation*
- *destination: GeoLocation*
- *createdAt: timestamp*
- *startTime: timestamp*
- *estimatedPrice: int*
- *timeToDestination: int*

We'll explain why the driverInfo is optional when we get to the API endpoints.

PassengerInfo

- *id: string*
- *name: string*
- *rating: int*

DriverInfo

- *id: string*
- *name: string*
- *rating: int*
- *ridesCount: int*
- *vehicleInfo: VehicleInfo*

VehicleInfo

- *licensePlate: string*

- ***description: string***

4. Passenger API

The passenger-facing API will be fairly straightforward. It'll consist of simple CRUD operations around the Ride entity, as well as an endpoint to stream a driver's location throughout a ride.

CreateRide(userId: string, pickup: Geolocation, destination: Geolocation)

=> Ride

Usage: called when a passenger books a ride; a Ride is created with no DriverInfo and with a CREATED RideStatus; the Uber backend calls an internal FindDriver API that uses an algorithm to find the most appropriate driver; once a driver is found and accepts the ride, the backend calls EditRide with the driver's info and with a MATCHED RideStatus.

GetRide(userId: string)

=> Ride

Usage: polled every couple of seconds after a ride has been created and until the ride has a status of MATCHED; afterwards, polled every 20-90 seconds throughout the trip to update the ride's estimated price, its time to destination, its RideStatus if it's been canceled by the driver, etc..

EditRide(userId: string, [...params?: all properties on the Ride object that need to be edited])

=> Ride

CancelRide(userId: string)

=> void

Wrapper around EditRide -- effectively calls EditRide(userId: string, rideStatus: CANCELLED).

StreamDriverLocation(userId: string)

=> Geolocation

Usage: continuously streams the location of a driver over a long-lived websocket connection; the driver whose location is streamed is the one associated with the Ride tied to the passed userId.

5. Driver API

The driver-facing API will rely on some of the same CRUD operations around the Ride entity, and it'll also have a SetDriverStatus endpoint as well as an endpoint to push the driver's location to passengers who are streaming it.

SetDriverStatus(userId: string, driverStatus: DriverStatus)

=> void

DriverStatus: enum UNAVAILABLE/IN RIDE/STANDBY

Usage: called when a driver wants to look for a ride, is starting a ride, or is done for the day; when called with STANDBY, the Uber backend calls an internal FindRide API that uses an algorithm to enqueue the driver in a queue of drivers waiting for rides and to find the most appropriate ride; once a ride is found, the ride is internally locked to the driver for 30 seconds, during which the driver can accept or reject the ride; once the driver accepts the ride, the internal backend calls EditRide with the driver's info and with a MATCHED RideStatus.

```
GetRide(userId: string)
  => Ride
```

Usage: polled every 20-90 seconds throughout the trip to update the ride's estimated price, its time to destination, whether it's been canceled, etc..

```
EditRide(userId: string, [...params?: all properties on the Ride object that need to be edited])
  => Ride
```

```
AcceptRide(userId: string)
  => void
```

Calls EditRide(userId, MATCHED) and SetDriverStatus(userId, IN_RIDE).

CancelRide(userId: string)

=> void

Wrapper around EditRide -- effectively calls EditRide(userId, CANCELLED).

PushLocation(userId: string, location: Geolocation)

=> void

Usage: continuously called by a driver's phone throughout a ride; pushes the driver's location to the relevant passenger who's streaming the location; the passenger is the one associated with the Ride tied to the passed userId.

6. UberPool

As a stretch goal, your interviewer might ask you to think about how you'd expand your design to handle UberPool rides.

UberPool rides allow multiple passengers (different Uber accounts) to share an Uber ride for a cheaper price.

One way to handle UberPool rides would be to allow Ride objects to have multiple passengerInfos. In this case, Rides would also have to maintain a list of all destinations that the ride will stop at, as well as the relevant final destinations for individual passengers.

Perhaps a cleaner way to handle UberPool rides would be to introduce an entirely new entity, a PoolRide entity, which would have a list of Rides attached to it. Passengers would still call the CreateRide endpoint when booking an UberPool ride, and so they would still have their own, normal Ride entity, but this entity would be attached to a PoolRide entity with the rest of the UberPool ride information.

Drivers would likely have an extra DriverStatus value to indicate if they were in a ride but still accepting new UberPool passengers.

Most of the other functionality would remain the same; passengers and drivers would still continuously poll the GetRide endpoint for updated information about the ride, passengers would still stream their driver's location, passengers would still be able to cancel their individual rides, etc..

Design Tinder

Many systems design questions are intentionally left very vague and are literally given in the form of `Design Foobar`. It's your job to ask clarifying questions to better understand the system that you have to build.

We've laid out some of these questions below; their answers should give you some guidance on the problem. Before looking at them, we encourage you to take few minutes to think about what questions you'd ask in a real interview.

Solution

Our solution walkthroughs are meant to supplement our video solutions. We recommend starting with the video solution and using the walkthrough either as a point of reference while you watch the video or as a review tool if you need to brush up on this question's solution later on.

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 Tinder, which allows users to create a profile and swipe through a seemingly endless deck of potential matches. Users can also super-like potential matches, putting themselves at the top of the other users' decks, and they can undo

their most recent swipe if it was a left swipe. Users don't have any limitations on the number of swipes, Super Likes, and Undos that they can do per day.

We're explicitly not designing any functionality that's available after two users match, including any kind of notification system to alert users that they've gotten a match, unless the match occurs directly when they swipe right on a potential match.

Our system should serve a global userbase of about 50 million users who are evenly distributed across the world, and we'd like to have mostly instant swipes, allowing for some latency when the Tinder app first loads up and after a user has swiped through a good number of potential matches.

We're told not to focus on the availability of our system, which should help us narrow down our design a little bit.

2. Coming Up With A Plan

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

- ***Storage Overview***
- ***Profile Creation***
- ***Deck Generation***
- ***Swiping***

We'll cover super-liking and undoing at the end, which will likely involve making tweaks to our design for swiping.

3. Storage Overview

Most of the data that we expect to store (profiles, decks, swipes, and matches), makes sense to be structured, so we'll use a SQL storage solution for it, and it'll be served directly from relevant SQL tables.

All of this data will be stored in regional databases, located based on user hot spots (e.g., a database on the east coast of the U.S., one in central U.S., one in western Europe, one in India, etc.), and users fetching Tinder data will be automatically routed to the closest regional database after being routed to intermediary API servers via some round-robin load balancing.

The only exception is users' profile pictures, which we'll store in a global blob store and which will be served via CDN.

We'll have some asynchronous replication between the regional databases, which should take anywhere from a few minutes to a few hours to occur. The asynchronicity of the replication should be fine, because the people that users interact with will usually, by the nature of the app, be close to them and therefore be using the same regional database as them.

4. Profile Creation

We'll store Tinder profiles in an individual SQL table, where each row will represent a profile:

- ***userId: string, the unique id of the user***
- ***geolocation: point***
- ***name: string***
- ***age: int***
- ***gender: enum***
- ***sexualPreference: enum***
- ***job: string***
- ***bio: string***
- ***pictures: string[], a list of blob-store addresses***

The userId field will be automatically assigned to the user, while most of the other fields will be set by the user when creating or editing their profile. The user's geolocation can be updated any time that the user opens the Tinder app and is in a different location than the one stored in their profile.

With 50 million users and an estimated upper bound of ~2KB per profile (pictures excluded), we'll need 2KB * 50e6 = 100GB of storage per region, or 1-5TB in total, assuming 10-50 regional databases. This is very little storage space.

As far as pictures are concerned, we can assume that users will have an average of five pictures each, with an upper bound of ~2MB per picture (high-quality, 1920x1080p). We'll almost certainly want to reduce the dimensions of pictures, since they'll only be viewable on small mobile screens, and we'll perform some lossy compression on them, because we can afford to lose a bit of quality. We can assume that this will bring pictures down to roughly ~50KB per picture (~200-500KB after dimension reduction and ~50KB after lossy compression).

~50KB * 5 = 250KB of pictures per user

~250KB * 50e6 users = 12.5TB (not a lot)

Clearly, the pictures account for most of our storage needs.

5. Deck Generation

For deck generation, we're going to have our smart deck-generation algorithm continuously generate decks of 200 potential matches for each user every day. This will ensure that user decks are as relevant as possible when users interact with them. For example, if someone is traveling out of a location and therefore no longer relevant to a particular user, they'll be

removed from the user's deck within a day, minimizing the chance for a user to see irrelevant profiles.

The deck-generation algorithm can be smart enough not to re-generate decks for users who are inactive for more than a day, and it can also be told to re-generate decks for users who've just changed location (i.e., when a user opens the Tinder app and is in a different location than the one stored in their profile row, the app tells the deck-generation algorithm to re-generate a deck for the user).

We'll store each user's deck of potential matches in an individual SQL table, where each row will represent a deck:

- *userId: string, the id of the user that this deck belongs to*
- *potentialMatches: string[], a list of userIds*

On app load, the Tinder app will request the 40 profiles at the top of their deck, remove them from the top of their deck (i.e., by updating their deck's row in the decks table), and locally store them. It's worth noting that, had we not compressed the profile images at the time of profile creation, each user would be requesting and attempting to store 400MB of data, which would be way too much data. With our compression, where each picture is ~50KB, 40 profiles becomes just 10MB of data, which is acceptable.

It's also worth noting that, if the user shuts their phone down or completely closes the Tinder app process, any locally stored profiles that the user hadn't swiped on will simply be readded to their deck at a later time by the deck-generation algorithm, since they were presumably relevant profiles and haven't yet been swiped on by the user.

The Tinder app will ensure that the number of locally cached profiles never goes below 20, such that the user almost never feels like they've run out of profiles to swipe on, even for a

few seconds. To accomplish this, the user's phone will eagerly fetch 20 additional profiles from the top of their deck when the user has 20 locally stored profiles left.

When the user runs out of potential matches (i.e., their deck has gone from 200 to 0 potential matches), the request for 20 more profiles triggers a new deck to be generated on demand. This is the only time that we might expect some potential loading time in the middle of using the app, but this happens infrequently, since the user would have to swipe on 200 potential matches within a day and would have to be swiping right extremely fast to go through their final 20 profiles before a new deck is generated.

6. Swiping

For swiping, we'll have two more SQL tables: one for swipes and one for matches. The SQL table for swipes will look like this:

- ***swiperId: string, the id of the user that performed the swipe***
- ***swipeeId: string, the id of the user that was swiped on***
- ***swipeType: enum (LIKE, PASS)***
- ***timestamp: datetime***

This table will be indexed on swipeeId and timestamp in order to allow for fast lookups of a user's recent swipes (all of the recent swipes that were performed on the user).

The SQL table for matches will look like this:

- ***userOneId: string, the id of the first user in the match***
- ***userTwoId: string, the id of the second user in the match***
- ***timestamp: datetime***

This matches table will mainly be used for the part of the system that is beyond the scope of this question.

On app load, the Tinder app will fetch all of the rows in the swipes table where swipeeId matches the user's userId. Then, every 30 seconds, it'll fetch the same rows, except only those with a timestamp after the most recent previously-fetched swipe's timestamp.

The Tinder app will keep all of the swipes in memory, in a hashtable-like structure, meaning that for any potential match, the app can know right away if they've already swiped on the user. This data can easily fit in memory on a phone (~20 bytes per swipe * maximum of 100k swipes = 2MB).

When a user swipes, the app will write the swipe to the swipes table. If the swipe is a LIKE, the backend will check for a matching swipe, and if there is one, it'll write a match to the matches table.

On the app's side, if there's a match (instantly knowable because of the local cache of swipes), the app will display a notification to the user; this is instant because we don't rely on the backend's response.

7. Super-Liking

The Super Like feature can be implemented with the following tweaks to our existing system:

1. ***A new SUPER-LIKE value is added to the swipeType in the swipes table.***
2. ***When a user (Foo) super-likes a potential match (Bar), the recorded swipe gets set to a SUPER-LIKE. If the backend notices a match, nothing else happens, except for writing the match to the matches table. Otherwise, the backend writes Foo's userId to Bar's deck row in the decks table, putting Foo at the top of Bar's deck, behind other super-likes, because older super-likes have precedence. If Foo's userId was already in Bar's deck, it simply gets moved.***

Our deck-generation algorithm is smart enough to keep super-likes at the top of decks, ordered by timestamp, such that older super-likes appear first.

As far as the Tinder UI is concerned, when the potential match at the top of a user's deck has super-liked the user (instantly knowable because of the local cache of swipes), a visual indicator is displayed. If a user gets super-liked while on the app by a user whose profile hasn't yet been fetched (i.e., a user who isn't in the 20-40 locally stored profiles), as soon as the app fetches the next 20 profiles from their deck, they'll see the Super Like at the top.

8. Undoing

The Undo feature can be implemented by simply delaying the API calls that occur on a left swipe until the next swipe or until the Tinder app is closed. This avoids doing multiple writes to the swipes table, which would otherwise be required in order to undo a left swipe.

Design Slack

Many systems design questions are intentionally left very vague and are literally given in the form of `Design Foobar`. It's your job to ask clarifying questions to better understand the system that you have to build.

We've laid out some of these questions below; their answers should give you some guidance on the problem. Before looking at them, we encourage you to take few minutes to think about what questions you'd ask in a real interview.

Solution

Our solution walkthroughs are meant to supplement our video solutions. We recommend starting with the video solution and using the walkthrough either as a point of reference while you watch the video or as a review tool if you need to brush up on this question's solution later on.

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 communication system behind Slack, which allows users to send instant messages in Slack channels.

Specifically, we'll want to support:

- ***Loading the most recent messages in a Slack channel when a user clicks on the channel.***
- ***Immediately seeing which channels have unread messages for a particular user when that user loads Slack.***
- ***Immediately seeing which channels have unread mentions of a particular user, for that particular user, when that user loads Slack, and more specifically, the number of these unread mentions in each relevant channel.***
- ***Sending and receiving Slack messages instantly, in real time.***
- ***Cross-device synchronization: if a user has both the Slack desktop app and the Slack mobile app open, with an unread channel in both, and if they read this channel on one device, the second device should immediately be updated and no longer display the channel as unread.***

The system should have low latencies and high availability, catering to a single region of roughly 20 million users. The largest Slack organizations will have as many as 50,000 users, with channels of the same size within them.

That being said, for the purpose of this design, we should primarily focus on latency and core functionality; availability and regionality can be disregarded, within reason.

2. Coming Up With A Plan

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

- ***Handling what happens when a Slack app loads.***
- ***Handling real-time messaging as well as cross-device synchronization.***

We can further divide the first section as follows:

- ***Seeing all of the channels that a user is a part of.***
- ***Seeing messages in a particular channel.***
- ***Seeing which channels have unread messages.***
- ***Seeing which channels have unread mentions and how many they have.***

3. Persistent Storage Solution & App Load

While a large component of our design involves real-time communication, another large part of it involves retrieving data (channels, messages, etc.) at any given time when the Slack app loads. To support this, we'll need a persistent storage solution.

Specifically, we'll opt for a SQL database since we can expect this data to be structured and to be queried frequently.

We can start with a simple table that'll store every Slack channel.

Channels

id (channelId): uuid	orgId: uuid	name: string	description: string
...	...	...	...

Then, we can have another simple table representing channel-member pairs: each row in this table will correspond to a particular user who is in a particular channel. We'll use this table, along with the one above, to fetch a user's relevant when the app loads.

Channel Members

id: uuid	orgId: uuid	channelId: uuid	userId: uuid
...	...	...	...

We'll naturally need a table to store all historical messages sent on Slack. This will be our largest table, and it'll be queried every time a user fetches messages in a particular channel. The API endpoint that'll interact with this table will return a paginated response, since we'll typically only want the 50 or 100 most recent messages per channel.

Also, this table will only be queried when a user clicks on a channel; we don't want to fetch messages for all of a user's channels on app load, since users will likely never look at most of their channels.

Historical Messages

id: uuid	orgId: uuid	channelId: uuid	senderId: uuid	sentAt: timestamp	body: string	mentions: List<uuid>
...	...	...	...	...	...	...

In order not to fetch recent messages for every channel on app load, all the while supporting the feature of showing which channels have unread messages, we'll need to store two extra tables: one for the latest activity in each channel (this table will be updated whenever a user sends a message in a channel), and one for the last time a particular user has read a channel (this table will be updated whenever a user opens a channel).

Latest Channel Timestamps

id: uuid	orgId: uuid	channelId: uuid	lastActive: timestamp
...	...	...	...

Channel Read Receipts

id: uuid	orgId: uuid	channelId: uuid	userId: uuid	lastSeen: timestamp
...	...	...	...	...

For the number of unread user mentions that we want to display next to channel names, we'll have another table similar to the read-receipts one, except this one will have a count of unread user mentions instead of a timestamp. This count will be updated (incremented) whenever a user tags another user in a channel message, and it'll also be updated (reset to 0) whenever a user opens a channel with unread mentions of themself.

Unread Channel-User-Mention Counts

id: uuid	orgId: uuid	channelId: uuid	userId: uuid	count: int
...	...	...	...	...

4. Load Balancing

For all of the API calls that clients will issue on app load, including writes to our database (when sending a message or marking a channel as read), we're going to want to load balance.

We can have a simple round-robin load balancer, forwarding requests to a set of server clusters that will then handle passing requests to our database.

5. "Smart" Sharding

Since our tables will be very large, especially the messages table, we'll need to have some sharding in place.

The natural approach is to shard based on organization size: we can have the biggest organizations (with the biggest channels) in their individual shards, and we can have smaller organizations grouped together in other shards.

An important point to note here is that, over time, organization sizes and Slack activity within organizations will change. Some organizations might double in size overnight, others might experience seemingly random surges of activity, etc.. This means that, despite our relatively sound sharding strategy, we might still run into hot spots, which is very bad considering the fact that we care about latency so much.

To handle this, we can add a "smart" sharding solution: a subsystem of our system that'll asynchronously measure organization activity and "rebalance" shards accordingly. This service can be a strongly consistent key-value store like Etcd or ZooKeeper, mapping orgIds to shards. Our API servers will communicate with this service to know which shard to route requests to.

6. Pub/Sub System for Real-Time Behavior

There are two types of real-time behavior that we want to support:

- ***Sending and receiving messages in real time.***
- ***Cross-device synchronization (instantly marking a channel as read if you have Slack open on two devices and read the channel on one of them).***

For both of these functionalities, we can rely on a Pub/Sub messaging system, which itself will rely on our previously described "smart" sharding strategy.

Every Slack organization or group of organizations will be assigned to a Kafka topic, and whenever a user sends a message in a channel or marks a channel as read, our previously mentioned API servers, which handle speaking to our database, will also send a Pub/Sub message to the appropriate Kafka topic.

The Pub/Sub messages will look like:

```
{
  "type": "chat",
  "orgId": "AAA",
  "channelId": "BBB",
  "userId": "CCC",
  "messageId": "DDD",
  "timestamp": "2020-08-31T01:17:02",
  "body": "this is a message",
  "mentions": ["CCC", "EEE"]
},
{
  "type": "read-receipt",
  "orgId": "AAA",
  "channelId": "BBB",
  "userId": "CCC",
  "timestamp": "2020-08-31T01:17:02"
}
```

We'll then have a different set of API servers who subscribe to the various Kakfa topics (probably one API server cluster per topic), and our clients (Slack users) will establish long-lived TCP connections with these API server clusters to receive Pub/Sub messages in real time.

We'll want a load balancer in between the clients and these API servers, which will also use the "smart" sharding strategy to match clients with the appropriate API servers, which will be listening to the appropriate Kafka topics.

When clients receive Pub/Sub messages, they'll handle them accordingly (mark a channel as unread, for example), and if the clients refresh their browser or their mobile app, they'll go through the entire "on app load" system that we described earlier.

Since each Pub/Sub message comes with a timestamp, and since reading a channel and sending Slack messages involve writing to our persistent storage, the Pub/Sub messages will effectively be idempotent operations.

Design Airbnb

Many systems design questions are intentionally left very vague and are literally given in the form of `Design Foobar`. It's your job to ask clarifying questions to better understand the system that you have to build.

We've laid out some of these questions below; their answers should give you some guidance on the problem. Before looking at them, we encourage you to take few minutes to think about what questions you'd ask in a real interview.

Solution

Our solution walkthroughs are meant to supplement our video solutions. We recommend starting with the video solution and using the walkthrough either as a point of reference while *you watch the video or as a review tool if you need to brush up on this question's solution later on.*

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 shouldn'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, 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.

Design The Twitch API

Design every API endpoint that's interacted with when a user is on an individual Twitch streamer's channel page, watching their livestream.

Many systems design questions are intentionally left very vague and are literally given in the form of `Design Foobar`. It's your job to ask clarifying questions to better understand the system that you have to build.

We've laid out some of these questions below; their answers should give you some guidance on the problem. Before looking at them, we encourage you to take few minutes to think about what questions you'd ask in a real interview.

Solution

Our solution walkthroughs are meant to supplement our video solutions. We recommend starting with the video solution and using the walkthrough either as a point of reference while you watch the video or as a review tool if you need to brush up on this question's solution later on.

1. Gathering Requirements

As with any API design interview question, the first thing that we want to do is to gather API requirements; we need to figure out what API we're building exactly.

We're designing every API endpoint that's interacted with when a user is on an individual Twitch streamer's channel page, watching their livestream. Specifically, we need to handle:

- *displaying the streamer's channel info (description text, follower count, etc.)*
- *following and unfollowing the streamer*

- ***subscribing to and unsubscribing from the streamer***
- ***seeing the live chat and sending messages; sending messages should only be allowable if the user isn't banned***
- ***seeing the livestream and being able to pause / unpause it***
- ***seeing the number of concurrent viewers of the stream, which should automatically be updated every 30 seconds or so***

2. Coming Up With A Plan

It's important to organize ourselves and to lay out a clear plan regarding how we're going to tackle our design. What are the major, potentially contentious parts of our API? Why are we making certain design decisions?

Fortunately for us, the various functionalities that we have to support effectively lay out a step-by-step plan for us, so we'll simply follow that.

Of note, all of the API endpoints that we'll define will take in, by default, the caller's user-specific authentication token as an authorization header. This token will be used by the backend to identify which user is calling each API endpoint.

We'll also be passing a channelId as a parameter to all of the endpoints, which will be the unique username of the streamer in question.

3. Channel Info

This is the most straightforward piece of functionality on the page, since it only consists of displaying static data about the streamer.

The user will call the GetChannelInfo endpoint, which will return the relevant entity, ChannelInfo, to be displayed on the page.

ChannelInfo

- ***name: string***
- ***description: string***
- ***currentStreamTitle: string***
- ***followerCount: int***

This entity might have more fields, but these are the most important ones.

```
GetChannelInfo(channelId: string)

  => ChannelInfo
```

4. Following

The follow status is binary: either the user is following the streamer, or they aren't. Thus, we can support the follow functionality with a single endpoint that uses a toggle mechanism, whereby the backend sets the follow status to the opposite of what's currently stored in the database.

```
ToggleFollow(channelId: string)

  => FollowState (FOLLOWING or NOT_FOLLOWING)
```

Naturally, this endpoint will be called when the user presses the "Follow" / "Unfollow" button.

Note that we haven't yet handled how to know what the user's follow state is. In other words, how do we know whether to show "Follow" or "Unfollow" to the user? See the Relationship To Channel section for details.

5. Subscribing

Subscribing is similar to following. However, unlike following, since there are more details to be provided when a user subscribes to a channel (subscription tier and payment information), we'll separate the acts of subscribing and unsubscribing into two endpoints.

```
CreateSubscription(channelId: string, subscriptionInfo: SubscriptionInfo, paymentInfo: PaymentInfo)

  => Subscription
```

```
CancelSubscription(channelId: string)

  => Subscription
```

Naturally, these endpoints will be called when the user presses the "Subscribe" / "Unsubscribe" button.

Note that we haven't yet handled how to know what the user's subscription state is. In other words, how do we know whether to show "Subscribe" or "Unsubscribe" to the user? See the Relationship To Channel section for details.

6. Chat

To handle the chat's functionality, we'll need two endpoints and a Message entity.

Message

- *sender: string, the username of the sender*
- *text: string*
- *timestamp: string, in ISO format*

StreamChat(channelId: string)

=> Message

SendMessage(channelId: string, message: string)

=> string | Error if user is banned

The StreamChat endpoint streams the stream's chat messages over a long-lived websocket connection and will be called once on page load.

The SendMessage endpoint will naturally be called whenever the user sends a message, and we can have the backend take care of timestamping messages and providing both the sender and the timestamp on the Message entity.

We can handle Twitch emotes by representing them with a special string format, like wrapping unique emote IDs in colons, as follows: :emote-id:. A Twitch a message will therefore look like this in string format:

"This stream is so fun to watch :kappa:" *The UI knows to detect this special string format and to display emotes appropriately. The UI also knows not to display messages sent*

by the user in question and received via StreamChat, since those messages will be displayed as soon as the user sends them via SendMessage.

While SendMessage returns an error if the user is banned from the chat, we won't actually allow the user to hit this endpoint if they're banned. That being said, we haven't yet handled how to know whether a user is banned. See the Relationship To Channel section for details.

7. Video

To display the actual video of the livestream, we'll open another long-lived websocket connection on page load, which will stream the video.

```
StreamVideo(channelId: string, videoQuality: VideoQuality)

  => VideoInfo
```

When this endpoint is called, we can imagine that the backend increases the concurrent-viewer count of the relevant stream in some database, which will be used in the next section to display the number of concurrent viewers to the user. When the long-lived connection is terminated (on tab close or page leave), the backend will decrease the relevant concurrent-viewer count in the database.

Lastly, when the user pauses the video, the UI still streams the video, but it simply doesn't display it.

8. Concurrent Viewers

Displaying the number of concurrent viewers watching a stream at any given time can easily be accomplished by polling an endpoint every 30 seconds or so, which reads from the database that stores every stream's concurrent-viewer count.

```
GetConcurrentViewers(channelId: string)
```

=> int

9. Relationship To Channel

There are a few pieces of functionality on the page that have to do with the relationship between the user and the streamer. Namely, whether the user is following the streamer, whether they're subscribed to the streamer, and whether they're banned from the streamer's chat.

One way to handle the follow and subscription states would be to fetch the user's profile info, which could contain all of their followed and subscribed streamers. This would be used with the streamer's name from GetChannelInfo to display the correct states (buttons) on the UI. The only problem is that a user could theoretically be following / subscribed to thousands of streamers, so we would maybe want to paginate the lists of followed and subscribed streamers, which would complicate things.

To make things simpler, and since we also have to handle the banned state, we can rely on a GetRelationshipToChannel endpoint, which will return the relevant entity, RelationshipToChannel, to be used to display the correct states on the page.

RelationshipToChannel

- ***isBanned: boolean***
- ***isFollowing: boolean***
- ***subscription: Subscription | null***

```
GetRelationshipToChannel(channelId: string)

  => RelationshipToChannel
```

If the user is banned, we'll prevent them from sending chat messages (and calling the SendMessage endpoint) altogether.