12. Design Pinterest

Step 1. Clarify requirements and make assumptions

All systems exist for a purpose, so with software ones. Meanwhile, software engineers are not artists - we build stuff to fulfill customers' needs. Thus, we should always start with the customer. Meanwhile, to fit the design into a 45-minute session, we must set constraints and scope the work by making assumptions.

Pinterest is a highly scalable photo-sharing service with hundreds of millions of monthly active users. Here are the requirements:

Most important features

- news feed: Customers will see a feed of images after login.
- one customer follows others to subscribe to their feeds.
- upload photos: They can upload their images, which will appear in the followers' feeds.

Scaling out

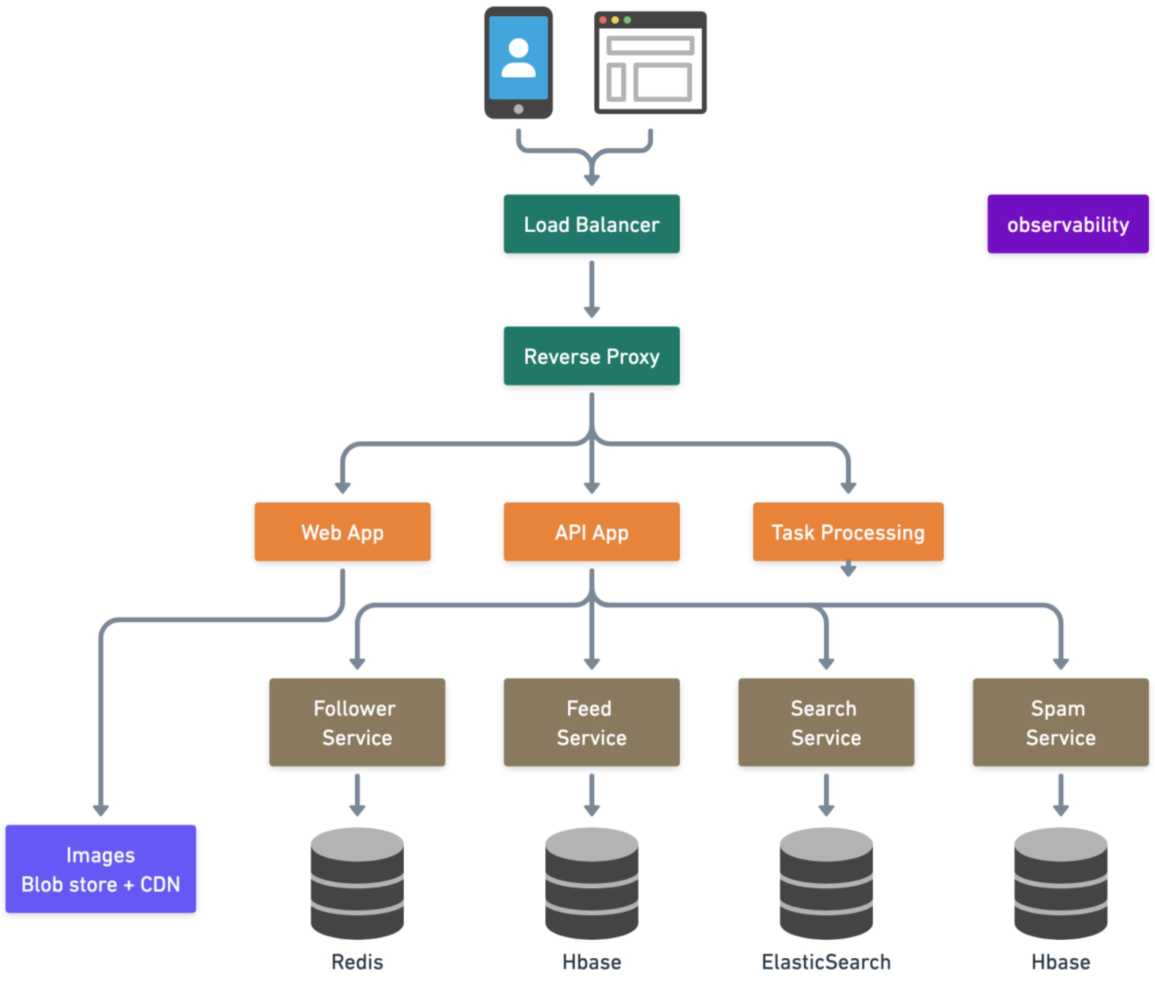
There are too many features and teams developing the product, so the product is decoupled into microservices.

Most of the services should be horizontally scalable and stateless.

Step 2. Sketch out the high-level design

Do not dive into details before outlining the big picture. Otherwise, going off too far in the wrong direction would waste time and prevent you from finishing the task.

Here is the high-level architecture, in which arrows indicate dependencies. (Sometimes, people would use arrows to describe the direction of data flow.)



Step 3. Dive into individual components and how they interact with each other

Once the archiecture is there, we could confirm with the interviewer if they want to go through each component with you. Sometimes, the interviewer may want to zoom into an unexpected domain problem like designing a photo store (that's why I am always saying there is no one-size-fits-all system design solution. Keep learning...). However, here, let's still assume that we are building the core abstraction: upload a photo and then publish to followers.

Again, I will explain as much as possible in a top-down order because this is our first design example. In the real world, you don't have to go through each component in such a level of detail literally; instead, you should focus on the core abstraction first.

Mobile and browser clients connect to the Pinterest data center via edge servers. An edge server is an edge device that provides an entry point into a network. Here we see two kinds of edge servers in the diagram - load balancers and reverse proxy.

Load Balancer (LB)

Load balancers distribute incoming network traffic to a group of backend servers. They fall into three categories:

DNS Round Robin (rarely used): clients get a randomly-ordered list of IP addresses.

- pros: easy to implement and usually free.
- cons: hard to control and not quite responsive because DNS cache takes time to expire.

L3/L4 Network-layer Load Balancer: traffic is routed by IP address and port. L3 is the network layer (IP). L4 is the transport layer (TCP).

- pros: better granularity, simple, responsive. e.g. forward traffic based on the ports.
- cons: content-agnostic: cannot route traffic by the content of the data.

L7 Application-layer Load Balancer: traffic is routed by what is inside the HTTP protocol. L7 is the application layer (HTTP). In case the interviewer wants more, we can suggest exact algorithms like round robin, weighted round robin, least loaded, least loaded with slow start, utilization limit, latency, cascade, etc. Check design L7 load balancer to learn more.

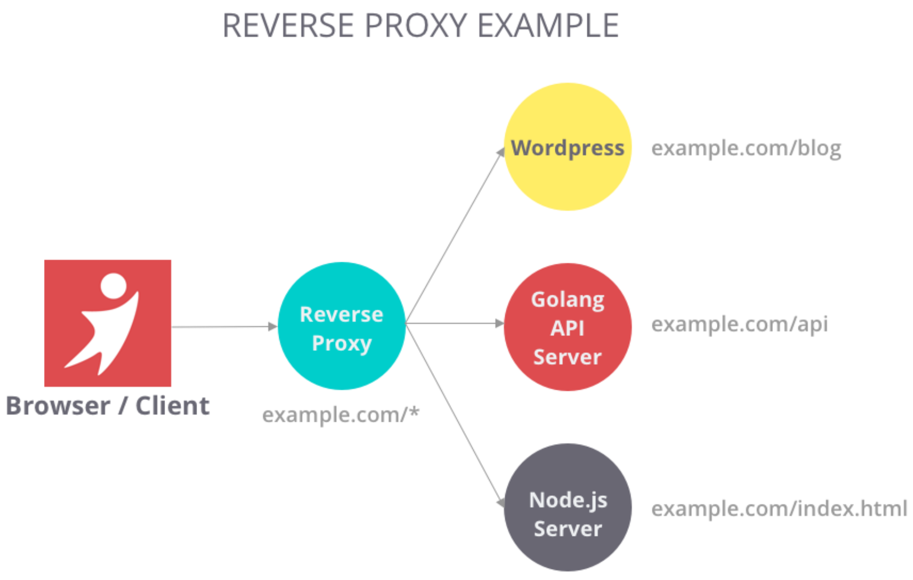
A load balancer could exist in many other places as long as there is a need for balancing traffic.

Reverse Proxy

Unlike a "forward" proxy in front of clients that route traffic to an external network, a reverse proxy is a kind of proxy sitting in front of servers, so it's called "reverse". By this definition, a load balancer is also a reverse proxy.

Reverse proxy brings a lot of benefits according to how you use it, and here are some typical ones:

1. **Routing**: centralize traffic to internal services and provides unified interfaces to the public. For example, www.example.com/index and www.example.com/sports appear to come from the same domain, but those pages are from different servers behind the reverse proxy.
2. **Filtering**: filter out requests without valid credentials for authentication or authorization.
3. **Caching**: Some resources are so popular for HTTP requests that you may want to configure some cache for the route to save some server resources.



Nginx, Varnish, HAProxy, and AWS Elastic Load balancing are popular products on the market. I find it handy but powerful to write a **lightweight reverse proxy** in

Golang. In the context of Kubernetes, it's basically what Ingress and Ingress Controllers are doing.

Web App

This is where we serve web pages. In the early days, web service usually combines the backend with page rendering, as Django and Ruby on Rails frameworks do. Later, growing with the project size, they are often decoupled to dedicated fronend and backend projects. Frontend focuses on App rendering while the backend serves the APIs for the frontend to consume.

Mobile App

Most backend engineers are not familiar with mobile design patterns, go to iOS Architecture Patterns for more.

A dedicated frontend web project is very similar to a standalone mobile app - they are both clients of the servers. Some people would call them "holistic frontend", when engineers can build user experiences on both platforms simultaneously, like react for web and react-native for mobile.

API App

Clients talk to the servers via public APIs. Nowadays, people often serve RESTful or GraphQL APIs. Learn more in public API choices.

Stateless web and API tier

There are two major bottlenecks of the whole system -- load (requests per second) and bandwidth. We could improve the situation by using more efficient software, e.g. using frameworks with async and non-blocking reactor pattern, or by using more hardware, like scaling up, aka vertical scaling: using more powerful machines like supercomputers or mainframes, or scaling out, aka horizontal scaling: using a more significant number of less-expensive machines.

Internet companies prefer scaling out, since

1. It is more cost-efficient with a vast number of commodity machines.
2. This is also good for recruiting - everyone could learn programming with a PC.

To scale out, we'd better keep services stateless, meaning they don't hold states in local memory or storage, so we could kill them unexpectedly or restart them anytime for any reason.

Learn more about scaling in how to scale a web service.

Service Tier

The single responsibility principle advocates small and autonomous services that work together so that Each service can "do one thing and do it well", and grow independently. Small teams owning small services can plan much more aggressively for hyper-growth. Learn more about Micro Services vs. Monolithic Services in Designing Uber

Service Discovery

How do those services find each other?

Zookeeper is a popular and centralized choice. Instances with name, address, port, etc. are registered into the path in ZooKeeper for each service. If one service does not know where to find another service, it can query Zookeeper for the location and memorize it until that location is unavailable.

Zookeeper is a CP system in terms of CAP theorem (See Section 2.3 for more discussion), which means it stays consistent in the case of failures, but the leader of the centralized consensus will be unavailable for registering new services.

In contrast to Zookeeper, Uber did some interesting work in a decentralized way, named hyperbahn, based on Ringpop consistent hash ring, though it turned out to be a big failure. In the context of Kubernetes, I would like to use service objects and Kube-proxy, so it would be easy for programmers to specify the address of the target service with internal DNS.

Follower Service

The follower-and-followee relationship is all around these two straightforward data structures:

1. Map<Followee, List of Followers>
2. Map<Follower, List of Followees>

A key-value store, like Redis, is very suitable here because the data structure is pretty simple, and this service should be mission-critical with high performance and low latency.

The follower service serves functionalities for followers and followees. For an image to appear in the feed, there are two models to make it happen.

- Push. Once the image is uploaded, we push the image metadata into all the followers' feeds. The follower will see its prepared feed directly.
- If the Map <Followee, List of Followers> fan-out is too large, then the push model will cost a lot of time and data duplicates.
- Pull. We don't prepare the feed in advance; instead, when the follower checks its feed, it fetches the list of followees and gets their images.
- If the Map<Follower, List of Followees> fan-out is too large, then the pull model will spend a lot of time iterating the huge followee list.

Feed Service

The feed service stores the image post metadata like URL, name, description, location, etc, in a database, while images themselves are usually saved in a Blob Storage like AWS S3 and Azure Blob store. Take S3 for example, a possible solution is like the following when the customer creates a post with the web or mobile client:

1. The server generates an S3 pre-signed URL which grants write permission.
2. The client uploads the image binary to S3 with the generated pre-signed URL.

3. The client submits the post and image metadata to the server and then triggers the data pipeline to push the post to followers' feeds if there is a push model.

Customers post to feeds as time passes, so HBase / Cassandra's timestamp index is an excellent fit for this use case.

Images Blob store and CDN

Transmitting blobs consumes a lot of brandwiths. Once we uploaded the blob, we read them a lot but seldemly update or delete it. Thus, developers often cache them with CDNs which will distribute those blobs to a closer place to the customer.

AWS CloudFront CDN + S3 might be the most popular combination on the market. I personally use BunnyCDN for my online content. Web3 developers like to use a decentralized store like IPFS and Arware.

Search Service

The search service connects to all the possible data sources and index them so that people could easily search feeds. We usually use ElasticSearch or Algolia to do the work.

Spam Service

The spam service uses machine learning techniques like supervised and unsupervised learning to mark and delete profanity content and fake accounts. Learn more in Fraud Detection with Semi-supervised Learning.

Step 4. Wrap up with blindspots or bottlenecks.

What are the blindspots or bottlenecks of the design above?

- As of 2022, people find it less favorable to use the follower-followee way of organizing feeds, because it would be hard 1) for new customers to bootstrap and 2) for existing customers to find more intriguing content. TikTok and Toutiao lead the new wave of innovations to organize feeds with recommendation algorithms. This design, however, does not cover the recommendation system part.
- For a popular photo-based SNS, scaling is the system's biggest challenge. So to make sure the design can survive the load, we need a capacity-planning.

Capacity planning with a spreadsheet and back-of-the-envelope calculation

There are two directions that we could approach the estimation problem: top-down and bottom-up.

For bottom-up, you do load tests with the existing system and plan the future on the company's current performance and future growth rate.

For top-down, you start with the customers in theory and make the back-of-the-envelope calculation. I highly recommend you do it with a digital spreadsheet, where you can easily list the formula and the assumed/calcuated numbers.

When we rely on external blob storage and CDN, bandwidth is unlikely to be a problem. So I will estimate the capacity for the follower service as an example:

Row	Description ("/" means per)	Estimated Number	Calculated
A	daily active users	33,000,000	
B	requests / user / day	60	
C	rps / machine	10,000 (c10k problem)	
D	scale factor (redundancy for user growth in 1 yr)	3 times	
E	Number of service instances	= A * B / (24 * 3600) / C * D	~= 7

We can see that Row E is a calculated result of the formula. After applying this estimation method to each one of those microservices and storage, we will better understand the entire system.

Real-world capacity planning is not a one-time deal. Provisioning too many machines will waste money, and preparing too few ones will cause outages. We usually do it with a few cycles of estimation and experimentation to find the right answer; or use auto scaling if the system supports this and budgets are not a problem.

Big corp engineers are often indulged with abundant computing and storage resources. However, great engineers will think about costs and benefits. I would sometimes experiment with different tiers of machines and add rows for their monthly expenses for estimation.