**System Design: Netflix**

Let's design a [Netflix](https://netflix.com/) like video streaming service, similar to services like [Amazon Prime Video](https://www.primevideo.com/), [Disney Plus](https://www.disneyplus.com/), [Hulu](https://www.hulu.com/), [Youtube](https://youtube.com/" \t "_blank), [Vimeo](https://vimeo.com/), etc.

**What is Netflix?**

Netflix is a subscription-based streaming service that allows its members to watch TV shows and movies on an internet-connected device. It is available on platforms such as the Web, iOS, Android, TV, etc.

**Requirements**

Our system should meet the following requirements:

**Functional requirements**

* Users should be able to stream and share videos.
* The content team (or users in YouTube's case) should be able to upload new videos (movies, tv shows episodes, and other content).
* Users should be able to search for videos using titles or tags.
* Users should be able to comment on a video similar to YouTube.

**Non-Functional requirements**

* High availability with minimal latency.
* High reliability, no uploads should be lost.
* The system should be scalable and efficient.

**Extended requirements**

* Certain content should be [geo-blocked](https://en.wikipedia.org/wiki/Geo-blocking).
* Resume video playback from the point user left off.
* Record metrics and analytics of videos.

**Estimation and Constraints**

Let's start with the estimation and constraints.

*Note: Make sure to check any scale or traffic-related assumptions with your interviewer.*

**Traffic**

This will be a read-heavy system, let us assume we have 1 billion total users with 200 million daily active users (DAU), and on average each user watches 5 videos a day. This gives us 1 billion videos watched per day.

200 *million*×5 *videos*=1 *billion*/*day*

Assuming, a 200:1 read/write ratio, about 50 million videos will be uploaded every day.

1200×1 billion=50 million/day2001​×1 *billion*=50 *million*/*day*

**What would be Requests Per Second (RPS) for our system?**

1 billion requests per day translate into 12K requests per second.

1 billion(24 hrs×3600 seconds)=∼12K requests/second(24 *hrs*×3600 *seconds*)1 *billion*​=∼12*K* *requests*/*second*

### Storage

If we assume each video is 100 MB on average, we will require about 5 PB of storage every day.

50 million×100 MB=5 PB/day50 *million*×100 *MB*=5 *PB*/*day*

And for 10 years, we will require an astounding 18,250 PB of storage.

5 PB×365 days×10 years=∼18,250 PB5 *PB*×365 *days*×10 *years*=∼18,250 *PB*

### Bandwidth

As our system is handling 5 PB of ingress every day, we will a require minimum bandwidth of around 58 GB per second.

5 PB(24 hrs×3600 seconds)=∼58 GB/second(24 *hrs*×3600 *seconds*)5 *PB*​=∼58 *GB*/*second*

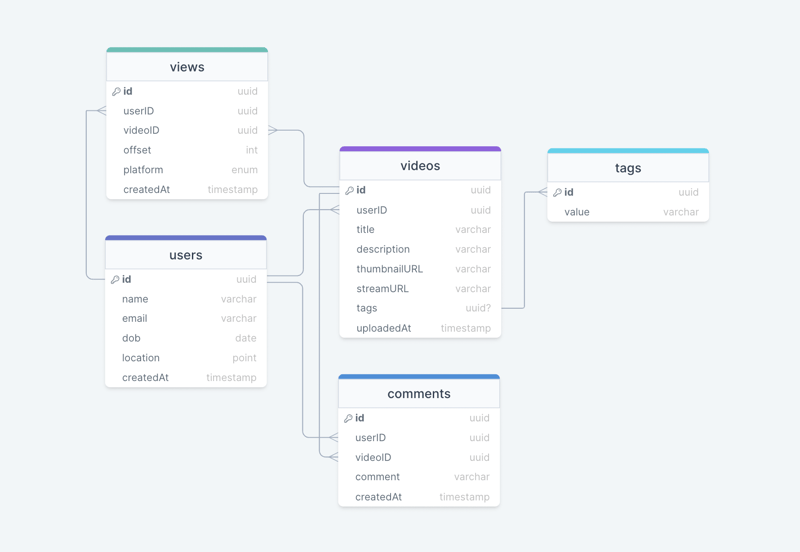
### High-level estimate

Here is our high-level estimate:

| **Type** | **Estimate** |
| --- | --- |
| Daily active users (DAU) | 200 million |
| Requests per second (RPS) | 12K/s |
| Storage (per day) | ~5 PB |
| Storage (10 years) | ~18,250 PB |
| Bandwidth | ~58 GB/s |

**Data model design**

This is the general data model which reflects our requirements.

[](https://media2.dev.to/dynamic/image/width=800%2Cheight=%2Cfit=scale-down%2Cgravity=auto%2Cformat=auto/https%3A%2F%2Fdev-to-uploads.s3.amazonaws.com%2Fuploads%2Farticles%2Fn3395yv46k5mjdp1xa6q.png)

We have the following tables:

**users**

This table will contain a user's information such as name, email, dob, and other details.

**videos**

As the name suggests, this table will store videos and their properties such as title, streamURL, tags, etc. We will also store the corresponding userID.

**tags**

This table will simply store tags associated with a video.

**views**

This table helps us to store all the views received on a video.

**comments**

This table stores all the comments received on a video (like YouTube).

**What kind of database should we use?**

While our data model seems quite relational, we don't necessarily need to store everything in a single database, as this can limit our scalability and quickly become a bottleneck.

We will split the data between different services each having ownership over a particular table. Then we can use a relational database such as [PostgreSQL](https://www.postgresql.org/) or a distributed NoSQL database such as [Apache Cassandra](https://cassandra.apache.org/_/index.html) for our use case.

**API design**

Let us do a basic API design for our services:

**Upload a video**

Given a byte stream, this API enables video to be uploaded to our service.

uploadVideo(title: string, description: string, data: Stream<byte>, tags?: string[]): boolean

**Parameters**

Title (string): Title of the new video.

Description (string): Description of the new video.

Data (Byte[]): Byte stream of the video data.

Tags (string[]): Tags for the video *(optional)*.

**Returns**

Result (boolean): Represents whether the operation was successful or not.

**Streaming a video**

This API allows our users to stream a video with the preferred codec and resolution.

streamVideo(videoID: UUID, codec: Enum<string>, resolution: Tuple<int>, offset?: int): VideoStream

**Parameters**

Video ID (UUID): ID of the video that needs to be streamed.

Codec (Enum<string>): Required [codec](https://en.wikipedia.org/wiki/Video_codec) of the requested video, such as h.265, h.264, VP9, etc.

Resolution (Tuple<int>): [Resolution](https://en.wikipedia.org/wiki/Display_resolution) of the requested video.

Offset (int): Offset of the video stream in seconds to stream data from any point in the video *(optional)*.

**Returns**

Stream (VideoStream): Data stream of the requested video.

**Search for a video**

This API will enable our users to search for a video based on its title or tags.

searchVideo(query: string, nextPage?: string): Video[]

**Parameters**

Query (string): Search query from the user.

Next Page (string): Token for the next page, this can be used for pagination *(optional)*.

**Returns**

Videos (Video[]): All the videos available for a particular search query.

**Add a comment**

This API will allow our users to post a comment on a video (like YouTube).

comment(videoID: UUID, comment: string): boolean

**Parameters**

VideoID (UUID): ID of the video user wants to comment on.

Comment (string): The text content of the comment.

**Returns**

Result (boolean): Represents whether the operation was successful or not.

**High-level design**

Now let us do a high-level design of our system.

**Architecture**

We will be using [microservices architecture](https://karanpratapsingh.com/courses/system-design/monoliths-microservices#microservices) since it will make it easier to horizontally scale and decouple our services. Each service will have ownership of its own data model. Let's try to divide our system into some core services.

**User Service**

This service handles user-related concerns such as authentication and user information.

**Stream Service**

The tweet service will handle video streaming-related functionality.

**Search Service**

The service is responsible for handling search-related functionality. It will be discussed in detail separately.

**Media service**

This service will handle the video uploads and processing. It will be discussed in detail separately.

**Analytics Service**

This service will be used for metrics and analytics use cases.

**What about inter-service communication and service discovery?**

Since our architecture is microservices-based, services will be communicating with each other as well. Generally, REST or HTTP performs well but we can further improve the performance using [gRPC](https://karanpratapsingh.com/courses/system-design/rest-graphql-grpc" \l "grpc" \t "_blank) which is more lightweight and efficient.

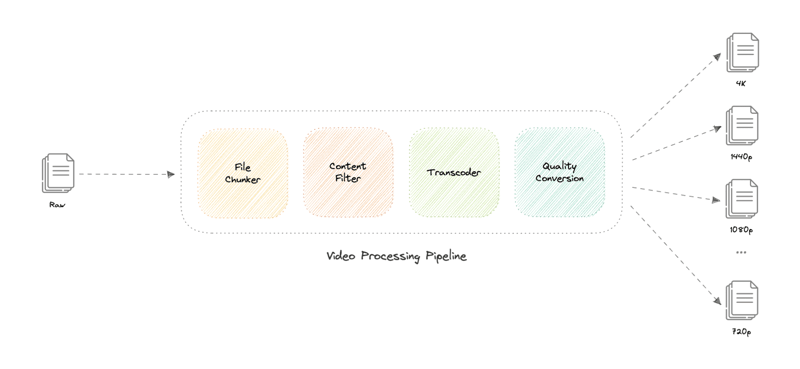
[Service discovery](https://karanpratapsingh.com/courses/system-design/service-discovery) is another thing we will have to take into account. We can also use a service mesh that enables managed, observable, and secure communication between individual services.

*Note: Learn more about*[*REST, GraphQL, gRPC*](https://karanpratapsingh.com/courses/system-design/rest-graphql-grpc)*and how they compare with each other.*

**Video processing**

There are so many variables in play when it comes to processing a video. For example, an average data size of two-hour raw 8K footage from a high-end camera can easily be up to 4 TB, thus we need to have some kind of processing to reduce both storage and delivery costs.

Here's how we can process videos once they're uploaded by the content team (or users in YouTube's case) and are queued for processing in our [message queue](https://karanpratapsingh.com/courses/system-design/message-queues).

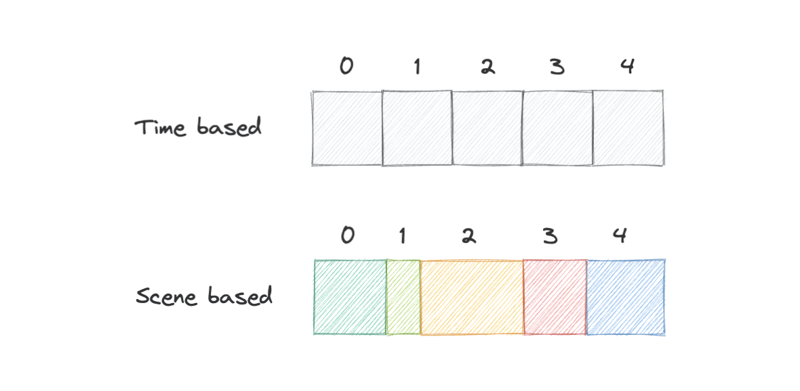
[](https://media2.dev.to/dynamic/image/width=800%2Cheight=%2Cfit=scale-down%2Cgravity=auto%2Cformat=auto/https%3A%2F%2Fdev-to-uploads.s3.amazonaws.com%2Fuploads%2Farticles%2Fgzffdro9wvxf8ffrot2q.png)

Let's discuss how this works:

* **File Chunker**

This is the first step of our processing pipeline. File chunking is the process of splitting a file into smaller pieces called chunks. It can help us eliminate duplicate copies of repeating data on storage, and reduces the amount of data sent over the network by only selecting changed chunks.

Usually, a video file can be split into equal size chunks based on timestamps but Netflix instead splits chunks based on scenes, this slight variation becomes a huge factor for a better user experience as whenever the client requests a chunk from the server, there is a lower chance of interruption as a complete scene will be retrieved.

[](https://media2.dev.to/dynamic/image/width=800%2Cheight=%2Cfit=scale-down%2Cgravity=auto%2Cformat=auto/https%3A%2F%2Fdev-to-uploads.s3.amazonaws.com%2Fuploads%2Farticles%2Fo7t66h8atatrbmui4c7s.png)

* **Content Filter**

This step checks if the video adheres to the content policy of the platform, this can be pre-approved in the case of Netflix as per the [content rating](https://en.wikipedia.org/wiki/Motion_picture_content_rating_system) of the media or can be strictly enforced like YouTube.

This entire step is done by a machine learning model which performs copyright, piracy, and NSFW checks. If issues are found, we can push the task to a [dead-letter queue (DLQ)](https://karanpratapsingh.com/courses/system-design/message-queues#dead-letter-queues) and someone from the moderation team can do further inspection.

* **Transcoder**

[Transcoding](https://en.wikipedia.org/wiki/Transcoding) is a process in which the original data is decoded to an intermediate uncompressed format, which is then encoded into the target format. This process uses different [codecs](https://en.wikipedia.org/wiki/Video_codec) to perform bitrate adjustment, image downsampling, or re-encoding the media.

This results in a smaller size file and a much more optimized format for the target devices. Standalone solutions such as [FFmpeg](https://ffmpeg.org/" \t "_blank) or cloud-based solutions like [AWS Elemental MediaConvert](https://aws.amazon.com/mediaconvert) can be used to implement this step of the pipeline.

* **Quality Conversion**

This is the last step of the processing pipeline and as the name suggests, this step handles the conversion of the transcoded media from the previous step into different resolutions such as 4K, 1440p, 1080p, 720p, etc.

This allows us to fetch the desired quality of the video as per the user's request, and once the media file finishes processing, it will be uploaded to a distributed file storage such as [HDFS](https://karanpratapsingh.com/courses/system-design/storage#hdfs), [GlusterFS](https://www.gluster.org/" \t "_blank), or an [object storage](https://karanpratapsingh.com/courses/system-design/storage#object-storage) such as [Amazon S3](https://aws.amazon.com/s3) for later retrieval during streaming.

*Note: We can add additional steps such as subtitles and thumbnails generation as part of our pipeline.*

**Why are we using a message queue?**

Processing videos as a long-running task makes much more sense, and a [message queue](https://karanpratapsingh.com/courses/system-design/message-queues) also decouples our video processing pipeline from the uploads functionality. We can use something like [Amazon SQS](https://aws.amazon.com/sqs) or [RabbitMQ](https://www.rabbitmq.com/) to support this.

**Video streaming**

Video streaming is a challenging task from both the client and server perspectives. Moreover, internet connection speeds vary quite a lot between different users. To make sure users don't re-fetch the same content, we can use a [Content Delivery Network (CDN)](https://karanpratapsingh.com/courses/system-design/content-delivery-network).

Netflix takes this a step further with its [Open Connect](https://openconnect.netflix.com/) program. In this approach, they partner with thousands of Internet Service Providers (ISPs) to localize their traffic and deliver their content more efficiently.

**What is the difference between Netflix's Open Connect and a traditional Content Delivery Network (CDN)?**

Netflix Open Connect is our purpose-built [Content Delivery Network (CDN)](https://karanpratapsingh.com/courses/system-design/content-delivery-network) responsible for serving Netflix's video traffic. Around 95% of the traffic globally is delivered via direct connections between Open Connect and the ISPs their customers use to access the internet.

Currently, they have Open Connect Appliances (OCAs) in over 1000 separate locations around the world. In case of issues, Open Connect Appliances (OCAs) can failover, and the traffic can be re-routed to Netflix servers.

Additionally, we can use [Adaptive bitrate streaming](https://en.wikipedia.org/wiki/Adaptive_bitrate_streaming) protocols such as [HTTP Live Streaming (HLS)](https://en.wikipedia.org/wiki/HTTP_Live_Streaming) which is designed for reliability and it dynamically adapts to network conditions by optimizing playback for the available speed of the connections.

Lastly, for playing the video from where the user left off (part of our extended requirements), we can simply use the offset property we stored in the views table to retrieve the scene chunk at that particular timestamp and resume the playback for the user.

**Searching**

Sometimes traditional DBMS are not performant enough, we need something which allows us to store, search, and analyze huge volumes of data quickly and in near real-time and give results within milliseconds. [Elasticsearch](https://www.elastic.co/) can help us with this use case.

[Elasticsearch](https://www.elastic.co/) is a distributed, free and open search and analytics engine for all types of data, including textual, numerical, geospatial, structured, and unstructured. It is built on top of [Apache Lucene](https://lucene.apache.org/).

**How do we identify trending content?**

Trending functionality will be based on top of the search functionality. We can cache the most frequently searched queries in the last N seconds and update them every M seconds using some sort of batch job mechanism.

**Sharing**

Sharing content is an important part of any platform, for this, we can have some sort of URL shortener service in place that can generate short URLs for the users to share.

*For more details, refer to the*[*URL Shortener*](https://karanpratapsingh.com/courses/system-design/url-shortener)*system design.*

**Detailed design**

It's time to discuss our design decisions in detail.

**Data Partitioning**

To scale out our databases we will need to partition our data. Horizontal partitioning (aka [Sharding](https://karanpratapsingh.com/courses/system-design/sharding)) can be a good first step. We can use partitions schemes such as:

* Hash-Based Partitioning
* List-Based Partitioning
* Range Based Partitioning
* Composite Partitioning

The above approaches can still cause uneven data and load distribution, we can solve this using [Consistent hashing](https://karanpratapsingh.com/courses/system-design/consistent-hashing).

*For more details, refer to*[*Sharding*](https://karanpratapsingh.com/courses/system-design/sharding)*and*[*Consistent Hashing*](https://karanpratapsingh.com/courses/system-design/consistent-hashing)*.*

**Geo-blocking**

Platforms like Netflix and YouTube use [Geo-blocking](https://en.wikipedia.org/wiki/Geo-blocking) to restrict content in certain geographical areas or countries. This is primarily done due to legal distribution laws that Netflix has to adhere to when they make a deal with the production and distribution companies. In the case of YouTube, this will be controlled by the user during the publishing of the content.

We can determine the user's location either using their [IP](https://karanpratapsingh.com/courses/system-design/ip) or region settings in their profile then use services like [Amazon CloudFront](https://aws.amazon.com/cloudfront) which supports a geographic restrictions feature or a [geolocation routing policy](https://docs.aws.amazon.com/Route53/latest/DeveloperGuide/routing-policy-geo.html) with [Amazon Route53](https://aws.amazon.com/route53) to restrict the content and re-route the user to an error page if the content is not available in that particular region or country.

**Recommendations**

Netflix uses a machine learning model which uses the user's viewing history to predict what the user might like to watch next, an algorithm like [Collaborative Filtering](https://en.wikipedia.org/wiki/Collaborative_filtering) can be used.

However, Netflix (like YouTube) uses its own algorithm called Netflix Recommendation Engine which can track several data points such as:

* User profile information like age, gender, and location.
* Browsing and scrolling behavior of the user.
* Time and date a user watched a title.
* The device which was used to stream the content.
* The number of searches and what terms were searched.

*For more detail, refer to*[*Netflix recommendation research*](https://research.netflix.com/research-area/recommendations)*.*

**Metrics and Analytics**

Recording analytics and metrics is one of our extended requirements. We can capture the data from different services and run analytics on the data using [Apache Spark](https://spark.apache.org/) which is an open-source unified analytics engine for large-scale data processing. Additionally, we can store critical metadata in the views table to increase data points within our data.

**Caching**

In a streaming platform, caching is important. We have to be able to cache as much static media content as possible to improve user experience. We can use solutions like [Redis](https://redis.io/) or [Memcached](https://memcached.org/) but what kind of cache eviction policy would best fit our needs?

**Which cache eviction policy to use?**

[Least Recently Used (LRU)](https://en.wikipedia.org/wiki/Cache_replacement_policies#Least_recently_used_(LRU)) can be a good policy for our system. In this policy, we discard the least recently used key first.

**How to handle cache miss?**

Whenever there is a cache miss, our servers can hit the database directly and update the cache with the new entries.

*For more details, refer to*[*Caching*](https://karanpratapsingh.com/courses/system-design/caching)*.*

**Media streaming and storage**

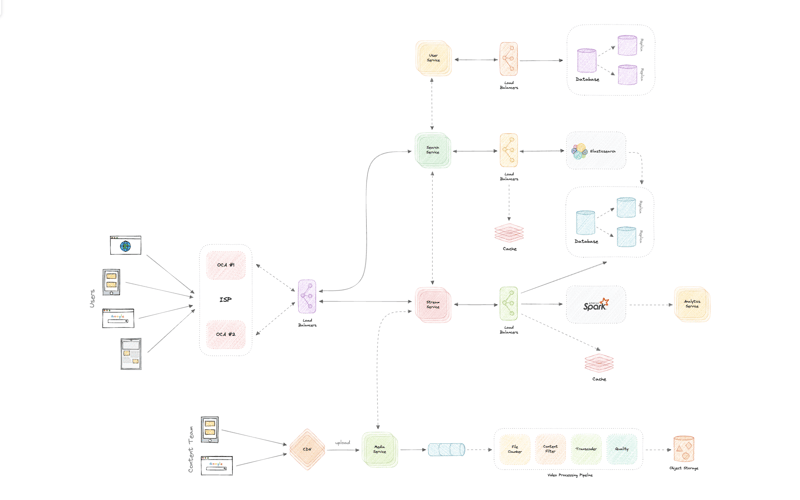
As most of our storage space will be used for storing media files such as thumbnails and videos. Per our discussion earlier, the media service will be handling both the upload and processing of media files.

We will use distributed file storage such as [HDFS](https://karanpratapsingh.com/courses/system-design/storage#hdfs), [GlusterFS](https://www.gluster.org/" \t "_blank), or an [object storage](https://karanpratapsingh.com/courses/system-design/storage#object-storage) such as [Amazon S3](https://aws.amazon.com/s3) for storage and streaming of the content.

**Content Delivery Network (CDN)**

[Content Delivery Network (CDN)](https://karanpratapsingh.com/courses/system-design/content-delivery-network) increases content availability and redundancy while reducing bandwidth costs. Generally, static files such as images, and videos are served from CDN. We can use services like [Amazon CloudFront](https://aws.amazon.com/cloudfront) or [Cloudflare CDN](https://www.cloudflare.com/cdn) for this use case.

**Identify and resolve bottlenecks**

[](https://media2.dev.to/dynamic/image/width=800%2Cheight=%2Cfit=scale-down%2Cgravity=auto%2Cformat=auto/https%3A%2F%2Fdev-to-uploads.s3.amazonaws.com%2Fuploads%2Farticles%2F5gvmskguwhi5ww7389gj.png)

Let us identify and resolve bottlenecks such as single points of failure in our design:

* "What if one of our services crashes?"
* "How will we distribute our traffic between our components?"
* "How can we reduce the load on our database?"
* "How to improve the availability of our cache?"

To make our system more resilient we can do the following:

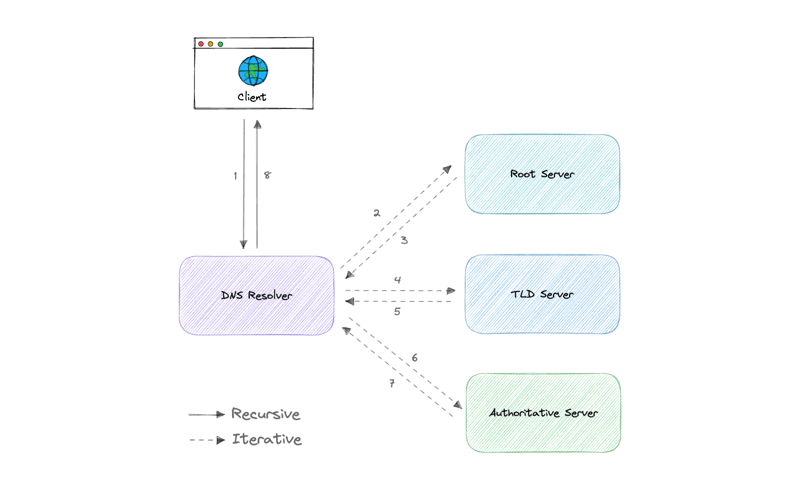
* Running multiple instances of each of our services.
* Introducing [load balancers](https://karanpratapsingh.com/courses/system-design/load-balancing) between clients, servers, databases, and cache servers.
* Using multiple read replicas for our databases.
* Multiple instances and replicas for our distributed cache.

**System Design: Domain Name System (DNS)**

Earlier we learned about IP addresses that enable every machine to connect with other machines. But as we know humans are more comfortable with names than numbers. It's easier to remember a name like google.com than something like 122.250.192.232.

This brings us to Domain Name System (DNS) which is a hierarchical and decentralized naming system used for translating human-readable domain names to IP addresses.

**How DNS works**

**[](https://media2.dev.to/dynamic/image/width=800%2Cheight=%2Cfit=scale-down%2Cgravity=auto%2Cformat=auto/https%3A%2F%2Fraw.githubusercontent.com%2Fkaranpratapsingh%2Fportfolio%2Fmaster%2Fpublic%2Fstatic%2Fcourses%2Fsystem-design%2Fchapter-I%2Fdomain-name-system%2Fhow-dns-works.png)**

DNS lookup involves the following eight steps:

1. A client types [example.com](http://example.com/) into a web browser, the query travels to the internet and is received by a DNS resolver.
2. The resolver then recursively queries a DNS root nameserver.
3. The root server responds to the resolver with the address of a Top Level Domain (TLD).
4. The resolver then makes a request to the .com TLD.
5. The TLD server then responds with the IP address of the domain's nameserver, [example.com](http://example.com/).
6. Lastly, the recursive resolver sends a query to the domain's nameserver.
7. The IP address for [example.com](http://example.com/) is then returned to the resolver from the nameserver.
8. The DNS resolver then responds to the web browser with the IP address of the domain requested initially.

Once the IP address has been resolved, the client should be able to request content from the resolved IP address. For example, the resolved IP may return a webpage to be rendered in the browser

**Server types**

Now, let's look at the four key groups of servers that make up the DNS infrastructure.

**DNS Resolver**

A DNS resolver (also known as a DNS recursive resolver) is the first stop in a DNS query. The recursive resolver acts as a middleman between a client and a DNS nameserver. After receiving a DNS query from a web client, a recursive resolver will either respond with cached data, or send a request to a root nameserver, followed by another request to a TLD nameserver, and then one last request to an authoritative nameserver. After receiving a response from the authoritative nameserver containing the requested IP address, the recursive resolver then sends a response to the client.

**DNS root server**

A root server accepts a recursive resolver's query which includes a domain name, and the root nameserver responds by directing the recursive resolver to a TLD nameserver, based on the extension of that domain (.com, .net, .org, etc.). The root nameservers are overseen by a nonprofit called the [Internet Corporation for Assigned Names and Numbers (ICANN)](https://www.icann.org/).

There are 13 DNS root nameservers known to every recursive resolver. Note that while there are 13 root nameservers, that doesn't mean that there are only 13 machines in the root nameserver system. There are 13 types of root nameservers, but there are multiple copies of each one all over the world, which use [Anycast routing](https://en.wikipedia.org/wiki/Anycast) to provide speedy responses.

**TLD nameserver**

A TLD nameserver maintains information for all the domain names that share a common domain extension, such as .com, .net, or whatever comes after the last dot in a URL.

Management of TLD nameservers is handled by the [Internet Assigned Numbers Authority (IANA)](https://www.iana.org/), which is a branch of [ICANN](https://www.icann.org/). The IANA breaks up the TLD servers into two main groups:

* **Generic top-level domains**: These are domains like .com, .org, .net, .edu, and .gov.
* **Country code top-level domains**: These include any domains that are specific to a country or state. Examples include .uk, .us, .ru, and .jp.

**Authoritative DNS server**

The authoritative nameserver is usually the resolver's last step in the journey for an IP address. The authoritative nameserver contains information specific to the domain name it serves (e.g. [google.com](http://google.com/)) and it can provide a recursive resolver with the IP address of that server found in the DNS A record, or if the domain has a CNAME record (alias) it will provide the recursive resolver with an alias domain, at which point the recursive resolver will have to perform a whole new DNS lookup to procure a record from an authoritative nameserver (often an A record containing an IP address). If it cannot find the domain, returns the NXDOMAIN message.

**Query Types**

There are three types of queries in a DNS system:

**Recursive**

In a recursive query, a DNS client requires that a DNS server (typically a DNS recursive resolver) will respond to the client with either the requested resource record or an error message if the resolver can't find the record.

**Iterative**

In an iterative query, a DNS client provides a hostname, and the DNS Resolver returns the best answer it can. If the DNS resolver has the relevant DNS records in its cache, it returns them. If not, it refers the DNS client to the Root Server or another Authoritative Name Server that is nearest to the required DNS zone. The DNS client must then repeat the query directly against the DNS server it was referred.

**Non-recursive**

A non-recursive query is a query in which the DNS Resolver already knows the answer. It either immediately returns a DNS record because it already stores it in a local cache, or queries a DNS Name Server which is authoritative for the record, meaning it definitely holds the correct IP for that hostname. In both cases, there is no need for additional rounds of queries (like in recursive or iterative queries). Rather, a response is immediately returned to the client.

**Records Types**

DNS records (aka zone files) are instructions that live in authoritative DNS servers and provide information about a domain including what IP address is associated with that domain and how to handle requests for that domain.

These records consist of a series of text files written in what is known as *DNS syntax*. DNS syntax is just a string of characters used as commands that tell the DNS server what to do. All DNS records also have a *"TTL"*, which stands for time-to-live, and indicates how often a DNS server will refresh that record.

There are more record types but for now, let's look at some of the most commonly used ones:

* **A (Address record)**: This is the record that holds the IP address of a domain.
* **AAAA (IP Version 6 Address record)**: The record that contains the IPv6 address for a domain (as opposed to A records, which stores the IPv4 address).
* **CNAME (Canonical Name record)**: Forwards one domain or subdomain to another domain, does NOT provide an IP address.
* **MX (Mail exchanger record)**: Directs mail to an email server.
* **TXT (Text Record)**: This record lets an admin store text notes in the record. These records are often used for email security.
* **NS (Name Server records)**: Stores the name server for a DNS entry.
* **SOA (Start of Authority)**: Stores admin information about a domain.
* **SRV (Service Location record)**: Specifies a port for specific services.
* **PTR (Reverse-lookup Pointer records)**: Provides a domain name in reverse lookups.
* **CERT (Certificate record)**: Stores public key certificates.

**Subdomains**

A subdomain is an additional part of our main domain name. It is commonly used to logically separate a website into sections. We can create multiple subdomains or child domains on the main domain.

For example, blog.example.com where blog is the subdomain, example is the primary domain and .com is the top-level domain (TLD). Similar examples can be support.example.com or careers.example.com.

**DNS Zones**

A DNS zone is a distinct part of the domain namespace which is delegated to a legal entity like a person, organization, or company, who is responsible for maintaining the DNS zone. A DNS zone is also an administrative function, allowing for granular control of DNS components, such as authoritative name servers.

**DNS Caching**

A DNS cache (sometimes called a DNS resolver cache) is a temporary database, maintained by a computer's operating system, that contains records of all the recent visits and attempted visits to websites and other internet domains. In other words, a DNS cache is just a memory of recent DNS lookups that our computer can quickly refer to when it's trying to figure out how to load a website.

The Domain Name System implements a time-to-live (TTL) on every DNS record. TTL specifies the number of seconds the record can be cached by a DNS client or server. When the record is stored in a cache, whatever TTL value came with it gets stored as well. The server continues to update the TTL of the record stored in the cache, counting down every second. When it hits zero, the record is deleted or purged from the cache. At that point, if a query for that record is received, the DNS server has to start the resolution process.

**Reverse DNS**

A reverse DNS lookup is a DNS query for the domain name associated with a given IP address. This accomplishes the opposite of the more commonly used forward DNS lookup, in which the DNS system is queried to return an IP address. The process of reverse resolving an IP address uses PTR records. If the server does not have a PTR record, it cannot resolve a reverse lookup.

Reverse lookups are commonly used by email servers. Email servers check and see if an email message came from a valid server before bringing it onto their network. Many email servers will reject messages from any server that does not support reverse lookups or from a server that is highly unlikely to be legitimate.

*Note: Reverse DNS lookups are not universally adopted as they are not critical to the normal function of the internet.*

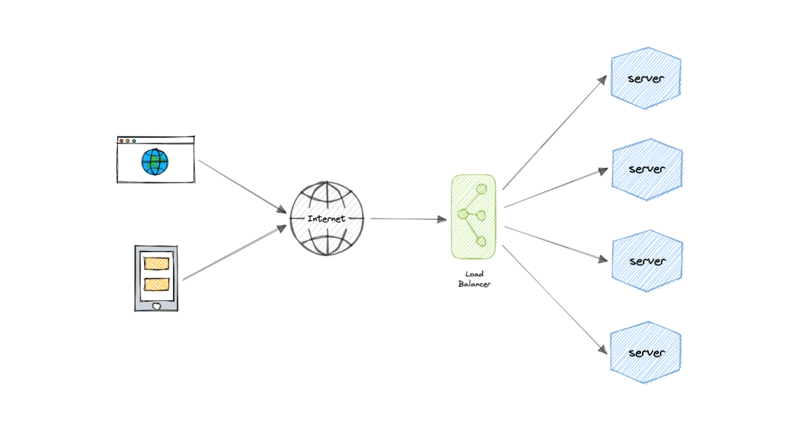
**Examples**

These are some widely used managed DNS solutions:

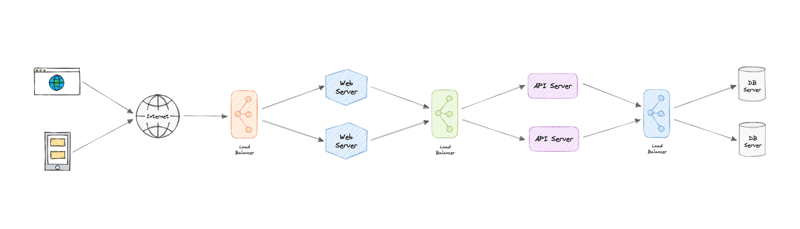
* [Route53](https://aws.amazon.com/route53)
* [Cloudflare DNS](https://www.cloudflare.com/dns)
* [Google Cloud DNS](https://cloud.google.com/dns)
* [Azure DNS](https://azure.microsoft.com/en-in/services/dns)
* [NS1](https://ns1.com/products/managed-dns)

**System Design: Load Balancing**

Load balancing lets us distribute incoming network traffic across multiple resources ensuring high availability and reliability by sending requests only to resources that are online. This provides the flexibility to add or subtract resources as demand dictates.

[](https://media2.dev.to/dynamic/image/width=800%2Cheight=%2Cfit=scale-down%2Cgravity=auto%2Cformat=auto/https%3A%2F%2Fraw.githubusercontent.com%2Fkaranpratapsingh%2Fportfolio%2Fmaster%2Fpublic%2Fstatic%2Fcourses%2Fsystem-design%2Fchapter-I%2Fload-balancing%2Fload-balancer.png)

For additional scalability and redundancy, we can try to load balance at each layer of our system:

[](https://media2.dev.to/dynamic/image/width=800%2Cheight=%2Cfit=scale-down%2Cgravity=auto%2Cformat=auto/https%3A%2F%2Fraw.githubusercontent.com%2Fkaranpratapsingh%2Fportfolio%2Fmaster%2Fpublic%2Fstatic%2Fcourses%2Fsystem-design%2Fchapter-I%2Fload-balancing%2Fload-balancer-layers.png)

**But why?**

Modern high-traffic websites must serve hundreds of thousands, if not millions, of concurrent requests from users or clients. To cost-effectively scale to meet these high volumes, modern computing best practice generally requires adding more servers.

A load balancer can sit in front of the servers and route client requests across all servers capable of fulfilling those requests in a manner that maximizes speed and capacity utilization. This ensures that no single server is overworked, which could degrade performance. If a single server goes down, the load balancer redirects traffic to the remaining online servers. When a new server is added to the server group, the load balancer automatically starts sending requests to it.

**Workload distribution**

This is the core functionality provided by a load balancer and has several common variations:

* **Host-based**: Distributes requests based on the requested hostname.
* **Path-based**: Using the entire URL to distribute requests as opposed to just the hostname.
* **Content-based**: Inspects the message content of a request. This allows distribution based on content such as the value of a parameter.

**Layers**

Generally speaking, load balancers operate at one of the two levels:

**Network layer**

This is the load balancer that works at the network's transport layer, also known as layer 4. This performs routing based on networking information such as IP addresses and is not able to perform content-based routing. These are often dedicated hardware devices that can operate at high speed.

**Application layer**

This is the load balancer that operates at the application layer, also known as layer 7. Load balancers can read requests in their entirety and perform content-based routing. This allows the management of load based on a full understanding of traffic.

**Types**

Let's look at different types of load balancers:

**Software**

Software load balancers usually are easier to deploy than hardware versions. They also tend to be more cost-effective and flexible, and they are used in conjunction with software development environments. The software approach gives us the flexibility of configuring the load balancer to our environment's specific needs. The boost in flexibility may come at the cost of having to do more work to set up the load balancer. Compared to hardware versions, which offer more of a closed-box approach, software balancers give us more freedom to make changes and upgrades.

Software load balancers are widely used and are available either as installable solutions that require configuration and management or as a managed cloud service.

**Hardware**

As the name implies, a hardware load balancer relies on physical, on-premises hardware to distribute application and network traffic. These devices can handle a large volume of traffic but often carry a hefty price tag and are fairly limited in terms of flexibility.

Hardware load balancers include proprietary firmware that requires maintenance and updates as new versions and security patches are released.

**DNS**

DNS load balancing is the practice of configuring a domain in the Domain Name System (DNS) such that client requests to the domain are distributed across a group of server machines.

Unfortunately, DNS load balancing has inherent problems limiting its reliability and efficiency. Most significantly, DNS does not check for server and network outages, or errors. It always returns the same set of IP addresses for a domain even if servers are down or inaccessible.

**Routing Algorithms**

Now, let's discuss commonly used routing algorithms:

* **Round-robin**: Requests are distributed to application servers in rotation.
* **Weighted Round-robin**: Builds on the simple Round-robin technique to account for differing server characteristics such as compute and traffic handling capacity using weights that can be assigned via DNS records by the administrator.
* **Least Connections**: A new request is sent to the server with the fewest current connections to clients. The relative computing capacity of each server is factored into determining which one has the least connections.
* **Least Response Time**: Sends requests to the server selected by a formula that combines the fastest response time and fewest active connections.
* **Least Bandwidth**: This method measures traffic in megabits per second (Mbps), sending client requests to the server with the least Mbps of traffic.
* **Hashing**: Distributes requests based on a key we define, such as the client IP address or the request URL.

**Advantages**

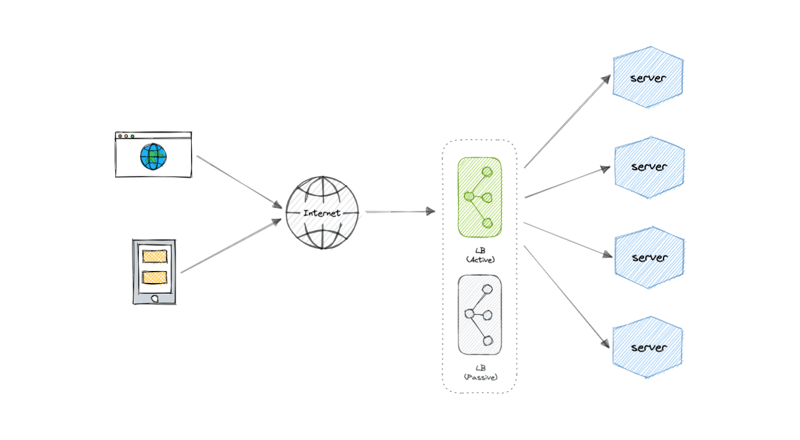
Load balancing also plays a key role in preventing downtime, other advantages of load balancing include the following:

* Scalability
* Redundancy
* Flexibility
* Efficiency

**Redundant load balancers**

As you must've already guessed, the load balancer itself can be a single point of failure. To overcome this, a second or N number of load balancers can be used in a cluster mode.

And, if there's a failure detection and the *active* load balancer fails, another *passive* load balancer can take over which will make our system more fault-tolerant.

[](https://media2.dev.to/dynamic/image/width=800%2Cheight=%2Cfit=scale-down%2Cgravity=auto%2Cformat=auto/https%3A%2F%2Fraw.githubusercontent.com%2Fkaranpratapsingh%2Fportfolio%2Fmaster%2Fpublic%2Fstatic%2Fcourses%2Fsystem-design%2Fchapter-I%2Fload-balancing%2Fredundant-load-balancer.png)

**Features**

Here are some commonly desired features of load balancers:

* **Autoscaling**: Starting up and shutting down resources in response to demand conditions.
* **Sticky sessions**: The ability to assign the same user or device to the same resource in order to maintain the session state on the resource.
* **Healthchecks**: The ability to determine if a resource is down or performing poorly in order to remove the resource from the load balancing pool.
* **Persistence connections**: Allowing a server to open a persistent connection with a client such as a WebSocket.
* **Encryption**: Handling encrypted connections such as TLS and SSL.
* **Certificates**: Presenting certificates to a client and authentication of client certificates.
* **Compression**: Compression of responses.
* **Caching**: An application-layer load balancer may offer the ability to cache responses.
* **Logging**: Logging of request and response metadata can serve as an important audit trail or source for analytics data.
* **Request tracing**: Assigning each request a unique id for the purposes of logging, monitoring, and troubleshooting.
* **Redirects**: The ability to redirect an incoming request based on factors such as the requested path.
* **Fixed response**: Returning a static response for a request such as an error message.

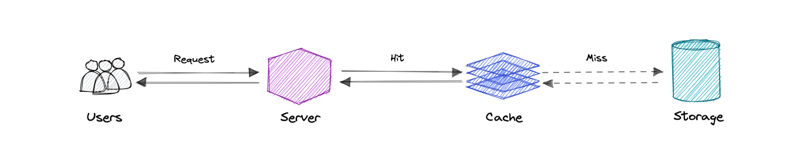
**Examples**

Following are some of the load balancing solutions commonly used in the industry:

* [Amazon Elastic Load Balancing](https://aws.amazon.com/elasticloadbalancing)
* [Azure Load Balancing](https://azure.microsoft.com/en-in/services/load-balancer)
* [GCP Load Balancing](https://cloud.google.com/load-balancing)
* [DigitalOcean Load Balancer](https://www.digitalocean.com/products/load-balancer)
* [Nginx](https://www.nginx.com/)
* [HAProxy](http://www.haproxy.org/)

**System Design: Caching**

*"There are only two hard things in Computer Science: cache invalidation and naming things." - Phil Karlton*

[](https://media2.dev.to/dynamic/image/width=800%2Cheight=%2Cfit=scale-down%2Cgravity=auto%2Cformat=auto/https%3A%2F%2Fraw.githubusercontent.com%2Fkaranpratapsingh%2Fportfolio%2Fmaster%2Fpublic%2Fstatic%2Fcourses%2Fsystem-design%2Fchapter-I%2Fcaching%2Fcaching.png)

A cache's primary purpose is to increase data retrieval performance by reducing the need to access the underlying slower storage layer. Trading off capacity for speed, a cache typically stores a subset of data transiently, in contrast to databases whose data is usually complete and durable.

Caches take advantage of the locality of reference principle *"recently requested data is likely to be requested again".*

**Caching and Memory**

Similar to a computer's memory, a cache is a compact, fast-performing memory that stores data in a hierarchy of levels, starting at level one, and progressing from there sequentially. They are labeled as L1, L2, L3, and so on. A cache also gets written if requested, such as when there has been an update and new content needs to be saved to the cache, replacing the older content that was saved.

No matter whether the cache is read or written, it's done one block at a time. Each block also has a tag that includes the location where the data was stored in the cache. When data is requested from the cache, a search occurs through the tags to find the specific content that's needed in level one (L1) of the memory. If the correct data isn't found, more searches are conducted in L2.

If the data isn't found there, searches are continued in L3, then L4, and so on until it has been found, then, it's read and loaded. If the data isn't found in the cache at all, then it's written into it for quick retrieval the next time.

**Cache hit and Cache miss**

**Cache hit**

A cache hit describes the situation where content is successfully served from the cache. The tags are searched in the memory rapidly, and when the data is found and read, it's considered a cache hit.

**Cold, Warm, and Hot Caches**

A cache hit can also be described as cold, warm, or hot. In each of these, the speed at which the data is read is described.

A hot cache is an instance where data was read from the memory at the *fastest* possible rate. This happens when the data is retrieved from L1.

A cold cache is the *slowest* possible rate for data to be read, though, it's still successful so it's still considered a cache hit. The data is just found lower in the memory hierarchy such as in L3, or lower.

A warm cache is used to describe data that's found in L2 or L3. It's not as fast as a hot cache, but it's still faster than a cold cache. Generally, calling a cache warm is used to express that it's slower and closer to a cold cache than a hot one.

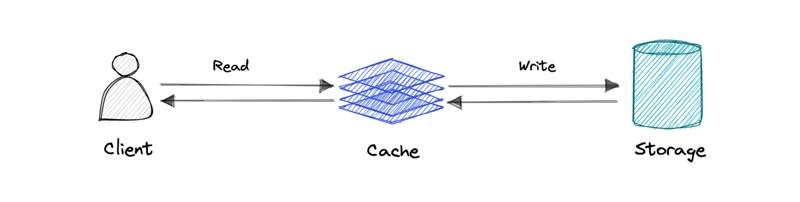
**Cache miss**

A cache miss refers to the instance when the memory is searched and the data isn't found. When this happens, the content is transferred and written into the cache.

**Cache Invalidation**

Cache invalidation is a process where the computer system declares the cache entries as invalid and removes or replaces them. If the data is modified, it should be invalidated in the cache, if not, this can cause inconsistent application behavior. There are three kinds of caching systems:

**Write-through cache**

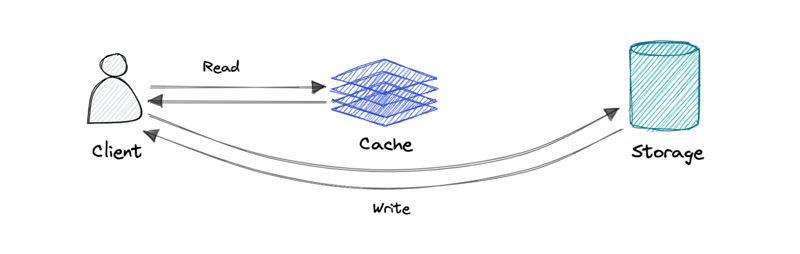
[](https://media2.dev.to/dynamic/image/width=800%2Cheight=%2Cfit=scale-down%2Cgravity=auto%2Cformat=auto/https%3A%2F%2Fraw.githubusercontent.com%2Fkaranpratapsingh%2Fportfolio%2Fmaster%2Fpublic%2Fstatic%2Fcourses%2Fsystem-design%2Fchapter-I%2Fcaching%2Fwrite-through-cache.png)

Data is written into the cache and the corresponding database simultaneously.

**Pro**: Fast retrieval, complete data consistency between cache and storage.

**Con**: Higher latency for write operations.

**Write-around cache**

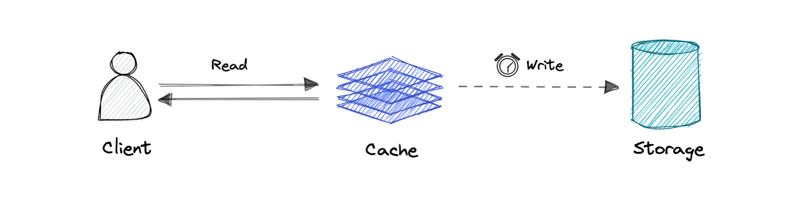
[](https://media2.dev.to/dynamic/image/width=800%2Cheight=%2Cfit=scale-down%2Cgravity=auto%2Cformat=auto/https%3A%2F%2Fraw.githubusercontent.com%2Fkaranpratapsingh%2Fportfolio%2Fmaster%2Fpublic%2Fstatic%2Fcourses%2Fsystem-design%2Fchapter-I%2Fcaching%2Fwrite-around-cache.png)

Where write directly goes to the database or permanent storage, bypassing the cache.

**Pro**: This may reduce latency.

**Con**: It increases cache misses because the cache system has to read the information from the database in case of a cache miss. As a result, this can lead to higher read latency in the case of applications that write and re-read the information quickly. Read happen from slower back-end storage and experiences higher latency.

**Write-back cache**

[](https://media2.dev.to/dynamic/image/width=800%2Cheight=%2Cfit=scale-down%2Cgravity=auto%2Cformat=auto/https%3A%2F%2Fraw.githubusercontent.com%2Fkaranpratapsingh%2Fportfolio%2Fmaster%2Fpublic%2Fstatic%2Fcourses%2Fsystem-design%2Fchapter-I%2Fcaching%2Fwrite-back-cache.png)

Where the write is only done to the caching layer and the write is confirmed as soon as the write to the cache completes. The cache then asynchronously syncs this write to the database.

**Pro**: This would lead to reduced latency and high throughput for write-intensive applications.

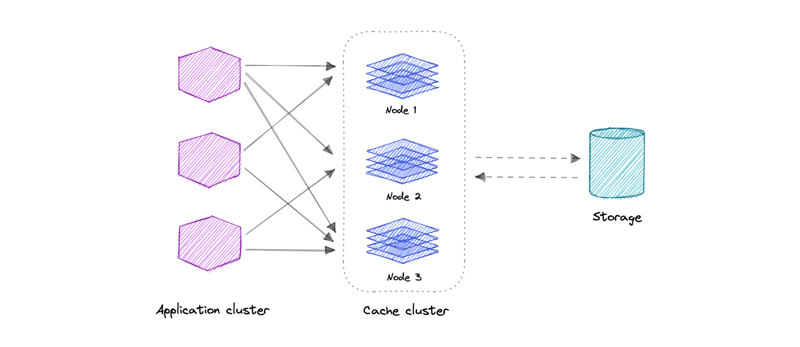
**Con**: There is a risk of data loss in case the caching layer crashes. We can improve this by having more than one replica acknowledging the write in the cache.

**Eviction policies**

Following are some of the most common cache eviction policies:

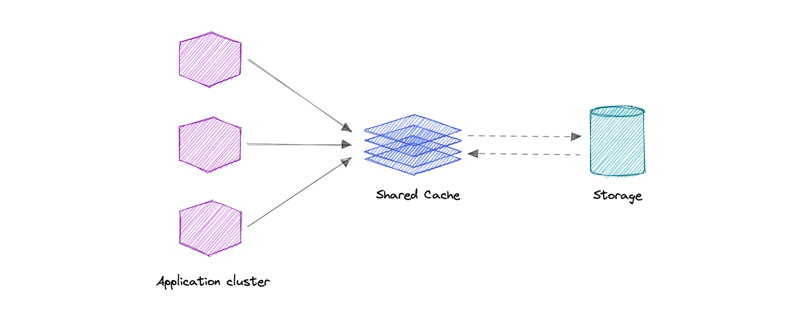
* **First In First Out (FIFO)**: The cache evicts the first block accessed first without any regard to how often or how many times it was accessed before.
* **Last In First Out (LIFO)**: The cache evicts the block accessed most recently first without any regard to how often or how many times it was accessed before.
* **Least Recently Used (LRU)**: Discards the least recently used items first.
* **Most Recently Used (MRU)**: Discards, in contrast to LRU, the most recently used items first.
* **Least Frequently Used (LFU)**: Counts how often an item is needed. Those that are used least often are discarded first.
* **Random Replacement (RR)**: Randomly selects a candidate item and discards it to make space when necessary.

**Distributed Cache**

[](https://media2.dev.to/dynamic/image/width=800%2Cheight=%2Cfit=scale-down%2Cgravity=auto%2Cformat=auto/https%3A%2F%2Fraw.githubusercontent.com%2Fkaranpratapsingh%2Fportfolio%2Fmaster%2Fpublic%2Fstatic%2Fcourses%2Fsystem-design%2Fchapter-I%2Fcaching%2Fdistributed-cache.png)

A distributed cache is a system that pools together the random-access memory (RAM) of multiple networked computers into a single in-memory data store used as a data cache to provide fast access to data. While most caches are traditionally in one physical server or hardware component, a distributed cache can grow beyond the memory limits of a single computer by linking together multiple computers.

**Global Cache**

[](https://media2.dev.to/dynamic/image/width=800%2Cheight=%2Cfit=scale-down%2Cgravity=auto%2Cformat=auto/https%3A%2F%2Fraw.githubusercontent.com%2Fkaranpratapsingh%2Fportfolio%2Fmaster%2Fpublic%2Fstatic%2Fcourses%2Fsystem-design%2Fchapter-I%2Fcaching%2Fglobal-cache.png)

As the name suggests, we will have a single shared cache that all the application nodes will use. When the requested data is not found in the global cache, it's the responsibility of the cache to find out the missing piece of data from the underlying data store.

**Use cases**

Caching can have many real-world use cases such as:

* Database Caching
* Content Delivery Network (CDN)
* Domain Name System (DNS) Caching
* API Caching

**When not to use caching?**

Let's also look at some scenarios where we should not use cache:

* Caching isn't helpful when it takes just as long to access the cache as it does to access the primary data store.
* Caching doesn't work as well when requests have low repetition (higher randomness), because caching performance comes from repeated memory access patterns.
* Caching isn't helpful when the data changes frequently, as the cached version gets out of sync, and the primary data store must be accessed every time.

*It's important to note that a cache should not be used as permanent data storage. They are almost always implemented in volatile memory because it is faster, and thus should be considered transient.*

**Advantages**

Below are some advantages of caching:

* Improves performance
* Reduce latency
* Reduce load on the database
* Reduce network cost
* Increase Read Throughput

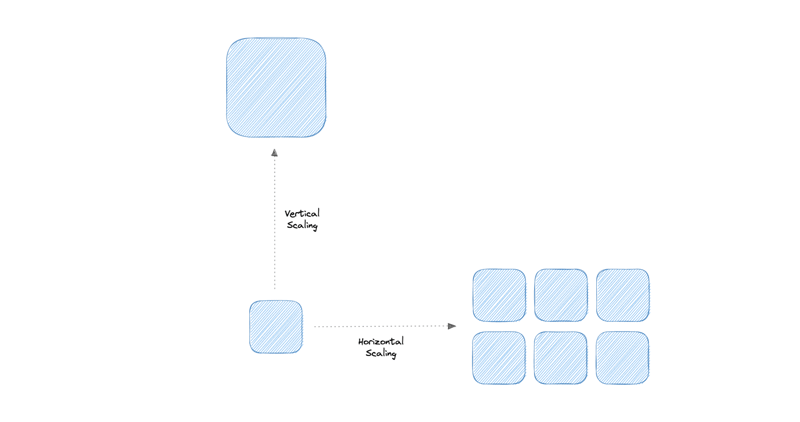
**Examples**

Here are some commonly used technologies for caching:

* [Redis](https://redis.io/)
* [Memcached](https://memcached.org/)
* [Amazon Elasticache](https://aws.amazon.com/elasticache)
* [Aerospike](https://aerospike.com/)

**System Design: Scalability**

Scalability is the measure of how well a system responds to changes by adding or removing resources to meet demands.

[](https://media2.dev.to/dynamic/image/width=800%2Cheight=%2Cfit=scale-down%2Cgravity=auto%2Cformat=auto/https%3A%2F%2Fraw.githubusercontent.com%2Fkaranpratapsingh%2Fportfolio%2Fmaster%2Fpublic%2Fstatic%2Fcourses%2Fsystem-design%2Fchapter-I%2Fscalability%2Fscalability.png)

Let's discuss different types of scaling:

**Vertical scaling**

Vertical scaling (also known as scaling up) expands a system's scalability by adding more power to an existing machine. In other words, vertical scaling refers to improving an application's capability via increasing hardware capacity.

**Advantages**

* Simple to implement
* Easier to manage
* Data consistent

**Disadvantages**

* Risk of high downtime
* Harder to upgrade
* Can be a single point of failure

**Horizontal scaling**

Horizontal scaling (also known as scaling out) expands a system's scale by adding more machines. It improves the performance of the server by adding more instances to the existing pool of servers, allowing the load to be distributed more evenly.

**Advantages**

* Increased redundancy
* Better fault tolerance
* Flexible and efficient
* Easier to upgrade

**Disadvantages**

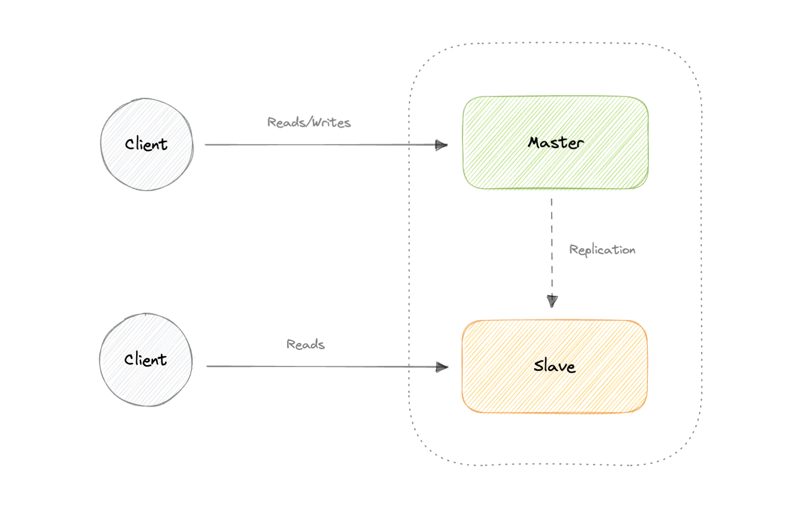
* Increases complexity
* Data inconsistency
* Increased load on downstream services

**System Design: Database Replication**

Replication is a process that involves sharing information to ensure consistency between redundant resources such as multiple databases, to improve reliability, fault-tolerance, or accessibility.

**Master-Slave Replication**

The master serves reads and writes, replicating writes to one or more slaves, which serve only reads. Slaves can also replicate additional slaves in a tree-like fashion. If the master goes offline, the system can continue to operate in read-only mode until a slave is promoted to a master or a new master is provisioned.

[](https://media2.dev.to/dynamic/image/width=800%2Cheight=%2Cfit=scale-down%2Cgravity=auto%2Cformat=auto/https%3A%2F%2Fraw.githubusercontent.com%2Fkaranpratapsingh%2Fportfolio%2Fmaster%2Fpublic%2Fstatic%2Fcourses%2Fsystem-design%2Fchapter-II%2Fdatabase-replication%2Fmaster-slave-replication.png)

**Advantages**

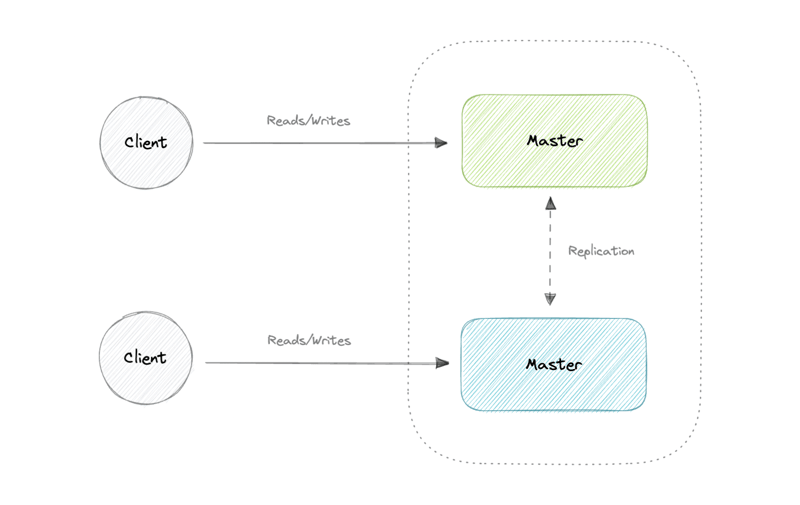
* Backups of the entire database of relatively no impact on the master.
* Applications can read from the slave(s) without impacting the master.
* Slaves can be taken offline and synced back to the master without any downtime.

**Disadvantages**

* Replication adds more hardware and additional complexity.
* Downtime and possibly loss of data when a master fails.
* All writes also have to be made to the master in a master-slave architecture.
* The more read slaves, the more we have to replicate, which will increase replication lag.

**Master-Master Replication**

Both masters serve reads/writes and coordinate with each other. If either master goes down, the system can continue to operate with both reads and writes.

[](https://media2.dev.to/dynamic/image/width=800%2Cheight=%2Cfit=scale-down%2Cgravity=auto%2Cformat=auto/https%3A%2F%2Fraw.githubusercontent.com%2Fkaranpratapsingh%2Fportfolio%2Fmaster%2Fpublic%2Fstatic%2Fcourses%2Fsystem-design%2Fchapter-II%2Fdatabase-replication%2Fmaster-master-replication.png)

**Advantages**

* Applications can read from both masters.
* Distributes write load across both master nodes.
* Simple, automatic, and quick failover.

**Disadvantages**

* Not as simple as master-slave to configure and deploy.
* Either loosely consistent or have increased write latency due to synchronization.
* Conflict resolution comes into play as more write nodes are added and as latency increases.

**Synchronous vs Asynchronous replication**

The primary difference between synchronous and asynchronous replication is how the data is written to the replica. In synchronous replication, data is written to primary storage and the replica simultaneously. As such, the primary copy and the replica should always remain synchronized.

In contrast, asynchronous replication copies the data to the replica after the data is already written to the primary storage. Although the replication process may occur in near-real-time, it is more common for replication to occur on a scheduled basis and it is more cost-effective.

*This article is part of my open source*[*System Design Course*](https://github.com/karanpratapsingh/system-design)*available on Github.*

[karanpratapsingh](https://github.com/karanpratapsingh" \t "_blank)/ [**system-design**](https://github.com/karanpratapsingh/system-design)

Learn how to design systems at scale and prepare for system design interviews

**System Design**

Hey, welcome to the course. I hope this course provides a great learning experience.

*This course is also available on my*[*website*](https://karanpratapsingh.com/courses/system-design)*and as an ebook on [leanpub](https://leanpub.com/systemdesign" \t "_blank). Please leave a ⭐ as motivation if this was helpful!*

**Table of contents**

* **Getting Started**
  + [What is system design?](https://github.com/karanpratapsingh/system-design#what-is-system-design)
* **Chapter I**
  + [IP](https://github.com/karanpratapsingh/system-design#ip)
  + [OSI Model](https://github.com/karanpratapsingh/system-design#osi-model)
  + [TCP and UDP](https://github.com/karanpratapsingh/system-design#tcp-and-udp)
  + [Domain Name System (DNS)](https://github.com/karanpratapsingh/system-design#domain-name-system-dns)
  + [Load Balancing](https://github.com/karanpratapsingh/system-design#load-balancing)
  + [Clustering](https://github.com/karanpratapsingh/system-design#clustering)
  + [Caching](https://github.com/karanpratapsingh/system-design#caching)
  + [Content Delivery Network (CDN)](https://github.com/karanpratapsingh/system-design#content-delivery-network-cdn)
  + [Proxy](https://github.com/karanpratapsingh/system-design#proxy)
  + [Availability](https://github.com/karanpratapsingh/system-design#availability)
  + [Scalability](https://github.com/karanpratapsingh/system-design#scalability)
  + [Storage](https://github.com/karanpratapsingh/system-design#storage)
* **Chapter II**
  + [Databases and DBMS](https://github.com/karanpratapsingh/system-design#databases-and-dbms)
  + [SQL databases](https://github.com/karanpratapsingh/system-design#sql-databases)
  + [NoSQL databases](https://github.com/karanpratapsingh/system-design#nosql-databases)
  + [SQL vs NoSQL databases](https://github.com/karanpratapsingh/system-design#sql-vs-nosql-databases)
  + [Database Replication](https://github.com/karanpratapsingh/system-design#database-replication)
  + [Indexes](https://github.com/karanpratapsingh/system-design#indexes)
  + [Normalization and Denormalization](https://github.com/karanpratapsingh/system-design#normalization-and-denormalization)
  + [ACID and BASE consistency models](https://github.com/karanpratapsingh/system-design#acid-and-base-consistency-models)
  + [CAP theorem](https://github.com/karanpratapsingh/system-design#cap-theorem)
  + [PACELC Theorem](https://github.com/karanpratapsingh/system-design#pacelc-theorem)
  + [Transactions](https://github.com/karanpratapsingh/system-design#transactions)
  + [Distributed Transactions](https://github.com/karanpratapsingh/system-design#distributed-transactions)
  + [Sharding](https://github.com/karanpratapsingh/system-design#sharding)
  + [Consistent Hashing](https://github.com/karanpratapsingh/system-design#consistent-hashing)
  + [Database Federation](https://github.com/karanpratapsingh/system-design#database-federation)
* **Chapter III**
  + [N-tier architecture](https://github.com/karanpratapsingh/system-design#n-tier-architecture)
  + [Message Brokers](https://github.com/karanpratapsingh/system-design#message-brokers)
  + [Message Queues](https://github.com/karanpratapsingh/system-design#message-queues)
  + [Publish-Subscribe](https://github.com/karanpratapsingh/system-design#publish-subscribe)
  + [Enterprise Service Bus (ESB)](https://github.com/karanpratapsingh/system-design#enterprise-service-bus-esb)
  + [Monoliths and Microservices](https://github.com/karanpratapsingh/system-design#monoliths-and-microservices)
  + [Event-Driven Architecture (EDA)](https://github.com/karanpratapsingh/system-design#event-driven-architecture-eda)
  + [Event Sourcing](https://github.com/karanpratapsingh/system-design#event-sourcing)
  + [Command and Query Responsibility Segregation (CQRS)](https://github.com/karanpratapsingh/system-design#command-and-query-responsibility-segregation-cqrs)
  + [API Gateway](https://github.com/karanpratapsingh/system-design#api-gateway)
  + [REST, GraphQL, gRPC](https://github.com/karanpratapsingh/system-design#rest-graphql-grpc)
  + [Long polling, WebSockets, Server-Sent Events (SSE)](https://github.com/karanpratapsingh/system-design#long-polling-websockets-server-sent-events-sse)
* **Chapter IV**
  + …

[View on GitHub](https://github.com/karanpratapsingh/system-design)

**System Design: WhatsApp**

Let's design a [Whatsapp](https://whatsapp.com/" \t "_blank) like instant messaging service, similar to services like [Whatsapp](https://www.whatsapp.com/" \t "_blank), [Facebook Messenger](https://www.messenger.com/), and [WeChat](https://www.wechat.com/).

**What is Whatsapp?**

Whatsapp is a chat application that provides instant messaging services to its users. It is one of the most used mobile applications on the planet connecting over 2 billion users in 180+ countries. Whatsapp is also available on the web.

**Requirements**

Our system should meet the following requirements:

**Functional requirements**

* Should support one-on-one chat.
* Group chats (max 100 people).
* Should support file sharing (image, video, etc.).

**Non-functional requirements**

* High availability with minimal latency.
* The system should be scalable and efficient.

**Extended requirements**

* Sent, Delivered, and Read receipts of the messages.
* Show the last seen time of users.
* Push notifications.

**Estimation and Constraints**

Let's start with the estimation and constraints.

*Note: Make sure to check any scale or traffic-related assumptions with your interviewer.*

**Traffic**

Let us assume we have 50 million daily active users (DAU) and on average each user sends at least 10 messages to 4 different people every day. This gives us 2 billion messages per day.

50 million×20 messages=2 billion/day50 *million*×20 *messages*=2 *billion*/*day*

Messages can also contain media such as images, videos, or other files. We can assume that 5 percent of messages are media files shared by the users, which gives us additional 200 million files we would need to store.

5 percent×2 billion=200 million/day5 *percent*×2 *billion*=200 *million*/*day*

**What would be Requests Per Second (RPS) for our system?**

2 billion requests per day translate into 24K requests per second.

2 billion(24 hrs×3600 seconds)=∼24K requests/second(24 *hrs*×3600 *seconds*)2 *billion*​=∼24*K* *requests*/*second*

### Storage

If we assume each message on average is 100 bytes, we will require about 200 GB of database storage every day.

2 billion×100 bytes=∼200 GB/day2 *billion*×100 *bytes*=∼200 *GB*/*day*

As per our requirements, we also know that around 5 percent of our daily messages (100 million) are media files. If we assume each file is 50 KB on average, we will require 10 TB of storage every day.

100 million×100 KB=10 TB/day100 *million*×100 *KB*=10 *TB*/*day*

And for 10 years, we will require about 38 PB of storage.

(10 TB+0.2 TB)×10 years×365 days=∼38 PB(10 *TB*+0.2 *TB*)×10 *years*×365 *days*=∼38 *PB*

### Bandwidth

As our system is handling 10.2 TB of ingress every day, we will a require minimum bandwidth of around 120 MB per second.

10.2 TB(24 hrs×3600 seconds)=∼120 MB/second(24 *hrs*×3600 *seconds*)10.2 *TB*​=∼120 *MB*/*second*

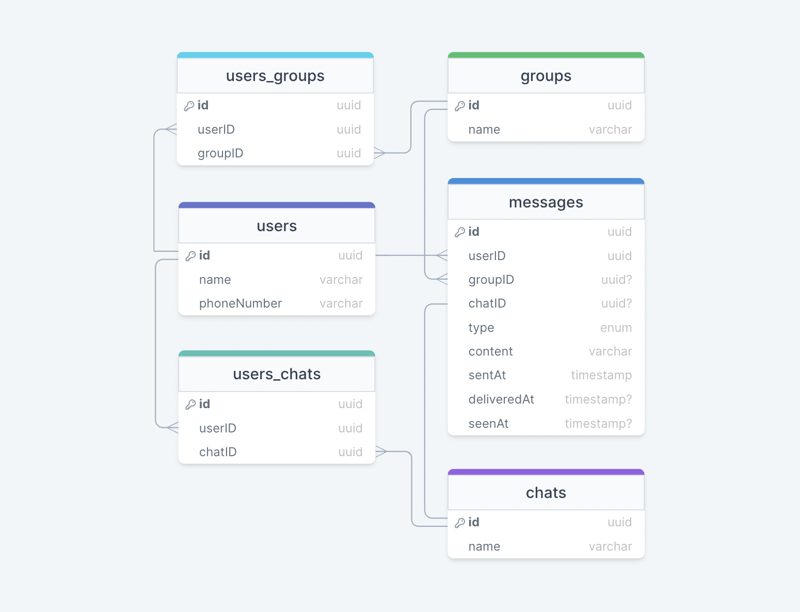
### High-level estimate

Here is our high-level estimate:

| **Type** | **Estimate** |
| --- | --- |
| Daily active users (DAU) | 50 million |
| Requests per second (RPS) | 24K/s |
| Storage (per day) | ~10.2 TB |
| Storage (10 years) | ~38 PB |
| Bandwidth | ~120 MB/s |

**Data model design**

This is the general data model which reflects our requirements.

[](https://media2.dev.to/dynamic/image/width=800%2Cheight=%2Cfit=scale-down%2Cgravity=auto%2Cformat=auto/https%3A%2F%2Fdev-to-uploads.s3.amazonaws.com%2Fuploads%2Farticles%2Fd6ivl6es8tyco7ogxzkr.png)

We have the following tables:

**users**

This table will contain a user's information such as name, phoneNumber, and other details.

**messages**

As the name suggests, this table will store messages with properties such as type (text, image, video, etc.), content, and timestamps for message delivery. The message will also have a corresponding chatID or groupID.

**chats**

This table basically represents a private chat between two users and can contain multiple messages.

**users\_chats**

This table maps users and chats as multiple users can have multiple chats (N:M relationship) and vice versa.

**groups**

This table represents a group between multiple users.

**users\_groups**

This table maps users and groups as multiple users can be a part of multiple groups (N:M relationship) and vice versa.

**What kind of database should we use?**

While our data model seems quite relational, we don't necessarily need to store everything in a single database, as this can limit our scalability and quickly become a bottleneck.

We will split the data between different services each having ownership over a particular table. Then we can use a relational database such as [PostgreSQL](https://www.postgresql.org/) or a distributed NoSQL database such as [Apache Cassandra](https://cassandra.apache.org/_/index.html) for our use case.

**API design**

Let us do a basic API design for our services:

**Get all chats or groups**

This API will get all chats or groups for a given userID.

getAll(userID: UUID): Chat[] | Group[]

**Parameters**

User ID (UUID): ID of the current user.

**Returns**

Result (Chat[] | Group[]): All the chats and groups the user is a part of.

**Get messages**

Get all messages for a user given the channelID (chat or group id).

getMessages(userID: UUID, channelID: UUID): Message[]

**Parameters**

User ID (UUID): ID of the current user.

Channel ID (UUID): ID of the channel (chat or group) from which messages need to be retrieved.

**Returns**

Messages (Message[]): All the messages in a given chat or group.

**Send message**

Send a message from a user to a channel (chat or group).

sendMessage(userID: UUID, channelID: UUID, message: Message): boolean

**Parameters**

User ID (UUID): ID of the current user.

Channel ID (UUID): ID of the channel (chat or group) user wants to send a message to.

Message (Message): The message (text, image, video, etc.) that the user wants to send.

**Returns**

Result (boolean): Represents whether the operation was successful or not.

**Join or leave a group**

Send a message from a user to a channel (chat or group).

joinGroup(userID: UUID, channelID: UUID): boolean

leaveGroup(userID: UUID, channelID: UUID): boolean

**Parameters**

User ID (UUID): ID of the current user.

Channel ID (UUID): ID of the channel (chat or group) the user wants to join or leave.

**Returns**

Result (boolean): Represents whether the operation was successful or not.

**High-level design**

Now let us do a high-level design of our system.

**Architecture**

We will be using [microservices architecture](https://karanpratapsingh.com/courses/system-design/monoliths-microservices#microservices) since it will make it easier to horizontally scale and decouple our services. Each service will have ownership of its own data model. Let's try to divide our system into some core services.

**User Service**

This is an HTTP-based service that handles user-related concerns such as authentication and user information.

**Chat Service**

The chat service will use WebSockets and establish connections with the client to handle chat and group message-related functionality. We can also use cache to keep track of all the active connections sort of like sessions which will help us determine if the user is online or not.

**Notification Service**

This service will simply send push notifications to the users. It will be discussed in detail separately.

**Presence Service**

The presence service will keep track of the last seen status of all users. It will be discussed in detail separately.

**Media service**

This service will handle the media (images, videos, files, etc.) uploads. It will be discussed in detail separately.

**What about inter-service communication and service discovery?**

Since our architecture is microservices-based, services will be communicating with each other as well. Generally, REST or HTTP performs well but we can further improve the performance using [gRPC](https://karanpratapsingh.com/courses/system-design/rest-graphql-grpc" \l "grpc" \t "_blank) which is more lightweight and efficient.

[Service discovery](https://karanpratapsingh.com/courses/system-design/service-discovery) is another thing we will have to take into account. We can also use a service mesh that enables managed, observable, and secure communication between individual services.

*Note: Learn more about*[*REST, GraphQL, gRPC*](https://karanpratapsingh.com/courses/system-design/rest-graphql-grpc)*and how they compare with each other.*

**Real-time messaging**

How do we efficiently send and receive messages? We have two different options:

**Pull model**

The client can periodically send an HTTP request to servers to check if there are any new messages. This can be achieved via something like [Long polling](https://karanpratapsingh.com/courses/system-design/long-polling-websockets-server-sent-events#long-polling).

**Push model**

The client opens a long-lived connection with the server and once new data is available it will be pushed to the client. We can use [WebSockets](https://karanpratapsingh.com/courses/system-design/long-polling-websockets-server-sent-events" \l "websockets" \t "_blank) or [Server-Sent Events (SSE)](https://karanpratapsingh.com/courses/system-design/long-polling-websockets-server-sent-events#server-sent-events-sse) for this.

The pull model approach is not scalable as it will create unnecessary request overhead on our servers and most of the time the response will be empty, thus wasting our resources. To minimize latency, using the push model with [WebSockets](https://karanpratapsingh.com/courses/system-design/long-polling-websockets-server-sent-events" \l "websockets" \t "_blank) is a better choice because then we can push data to the client once it's available without any delay given the connection is open with the client. Also, WebSockets provide full-duplex communication, unlike [Server-Sent Events (SSE)](https://karanpratapsingh.com/courses/system-design/long-polling-websockets-server-sent-events#server-sent-events-sse) which are only unidirectional.

*Note: Learn more about*[*Long polling, WebSockets, Server-Sent Events (SSE)*](https://karanpratapsingh.com/courses/system-design/long-polling-websockets-server-sent-events)*.*

**Last seen**

To implement the last seen functionality, we can use a [heartbeat](https://en.wikipedia.org/wiki/Heartbeat_(computing)) mechanism, where the client can periodically ping the servers indicating its liveness. Since this needs to be as low overhead as possible, we can store the last active timestamp in the cache as follows:

| **Key** | **Value** |
| --- | --- |
| User A | 2022-07-01T14:32:50 |
| User B | 2022-07-05T05:10:35 |
| User C | 2022-07-10T04:33:25 |

This will give us the last time the user was active. This functionality will be handled by the presence service combined with [Redis](https://redis.io/) or [Memcached](https://memcached.org/) as our cache.

Another way to implement this is to track the latest action of the user, once the last activity crosses a certain threshold, such as *"user hasn't performed any action in the last 30 seconds"*, we can show the user as offline and last seen with the last recorded timestamp. This will be more of a lazy update approach and might benefit us over heartbeat in certain cases.

**Notifications**

Once a message is sent in a chat or a group, we will first check if the recipient is active or not, we can get this information by taking the user's active connection and last seen into consideration.

If the recipient is not active, the chat service will add an event to a [message queue](https://karanpratapsingh.com/courses/system-design/message-queues) with additional metadata such as the client's device platform which will be used to route the notification to the correct platform later on.

The notification service will then consume the event from the message queue and forward the request to [Firebase Cloud Messaging (FCM)](https://firebase.google.com/docs/cloud-messaging) or [Apple Push Notification Service (APNS)](https://developer.apple.com/documentation/usernotifications) based on the client's device platform (Android, iOS, web, etc). We can also add support for email and SMS.

**Why are we using a message queue?**

Since most message queues provide best-effort ordering which ensures that messages are generally delivered in the same order as they're sent and that a message is delivered at least once which is an important part of our service functionality.

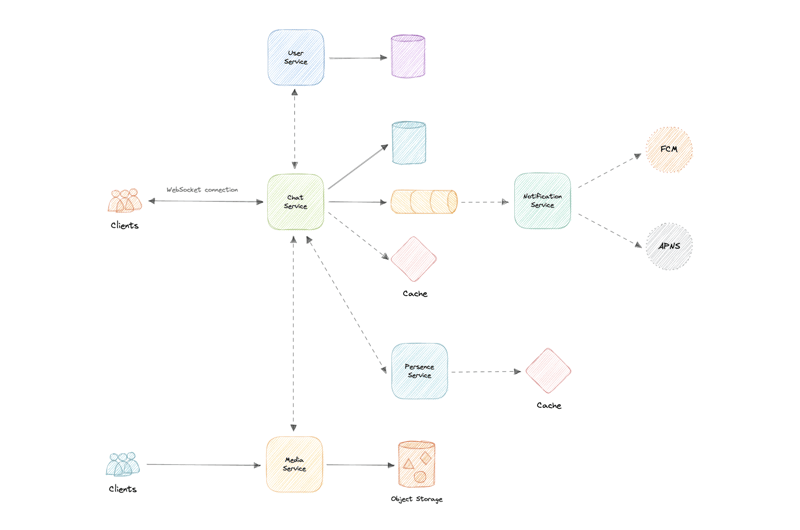
While this seems like a classic [publish-subscribe](https://karanpratapsingh.com/courses/system-design/publish-subscribe) use case, it is actually not as mobile devices and browsers each have their own way of handling push notifications. Usually, notifications are handled externally via Firebase Cloud Messaging (FCM) or Apple Push Notification Service (APNS) unlike message fan-out which we commonly see in backend services. We can use something like [Amazon SQS](https://aws.amazon.com/sqs) or [RabbitMQ](https://www.rabbitmq.com/) to support this functionality.

**Read receipts**

Handling read receipts can be tricky, for this use case we can wait for some sort of [Acknowledgment (ACK)](https://en.wikipedia.org/wiki/Acknowledgement_(data_networks)) from the client to determine if the message was delivered and update the corresponding deliveredAt field. Similarly, we will mark message the message seen once the user opens the chat and update the corresponding seenAt timestamp field.

**Design**

Now that we have identified some core components, let's do the first draft of our system design.

[](https://media2.dev.to/dynamic/image/width=800%2Cheight=%2Cfit=scale-down%2Cgravity=auto%2Cformat=auto/https%3A%2F%2Fdev-to-uploads.s3.amazonaws.com%2Fuploads%2Farticles%2Fibq8zcctgrz760s1smf6.png)

**Detailed design**

It's time to discuss our design decisions in detail.

**Data Partitioning**

To scale out our databases we will need to partition our data. Horizontal partitioning (aka [Sharding](https://karanpratapsingh.com/courses/system-design/sharding)) can be a good first step. We can use partitions schemes such as:

* Hash-Based Partitioning
* List-Based Partitioning
* Range Based Partitioning
* Composite Partitioning

The above approaches can still cause uneven data and load distribution, we can solve this using [Consistent hashing](https://karanpratapsingh.com/courses/system-design/consistent-hashing).

*For more details, refer to*[*Sharding*](https://karanpratapsingh.com/courses/system-design/sharding)*and*[*Consistent Hashing*](https://karanpratapsingh.com/courses/system-design/consistent-hashing)*.*

**Caching**

In a messaging application, we have to be careful about using cache as our users expect the latest data, but many users will be requesting the same messages, especially in a group chat. So, to prevent usage spikes from our resources we can cache older messages.

Some group chats can have thousands of messages and sending that over the network will be really inefficient, to improve efficiency we can add pagination to our system APIs. This decision will be helpful for users with limited network bandwidth as they won't have to retrieve old messages unless requested.

**Which cache eviction policy to use?**

We can use solutions like [Redis](https://redis.io/) or [Memcached](https://memcached.org/) and cache 20% of the daily traffic but what kind of cache eviction policy would best fit our needs?

[Least Recently Used (LRU)](https://en.wikipedia.org/wiki/Cache_replacement_policies#Least_recently_used_(LRU)) can be a good policy for our system. In this policy, we discard the least recently used key first.

**How to handle cache miss?**

Whenever there is a cache miss, our servers can hit the database directly and update the cache with the new entries.

*For more details, refer to*[*Caching*](https://karanpratapsingh.com/courses/system-design/caching)*.*

**Media access and storage**

As we know, most of our storage space will be used for storing media files such as images, videos, or other files. Our media service will be handling both access and storage of the user media files.

But where can we store files at scale? Well, [object storage](https://karanpratapsingh.com/courses/system-design/storage#object-storage) is what we're looking for. Object stores break data files up into pieces called objects. It then stores those objects in a single repository, which can be spread out across multiple networked systems. We can also use distributed file storage such as [HDFS](https://karanpratapsingh.com/courses/system-design/storage#hdfs) or [GlusterFS](https://www.gluster.org/" \t "_blank).

*Fun fact: Whatsapp deletes media on its servers once it has been downloaded by the user.*

We can use object stores like [Amazon S3](https://aws.amazon.com/s3), [Azure Blob Storage](https://azure.microsoft.com/en-in/services/storage/blobs), or [Google Cloud Storage](https://cloud.google.com/storage) for this use case.

**Content Delivery Network (CDN)**

[Content Delivery Network (CDN)](https://karanpratapsingh.com/courses/system-design/content-delivery-network) increases content availability and redundancy while reducing bandwidth costs. Generally, static files such as images, and videos are served from CDN. We can use services like [Amazon CloudFront](https://aws.amazon.com/cloudfront) or [Cloudflare CDN](https://www.cloudflare.com/cdn) for this use case.

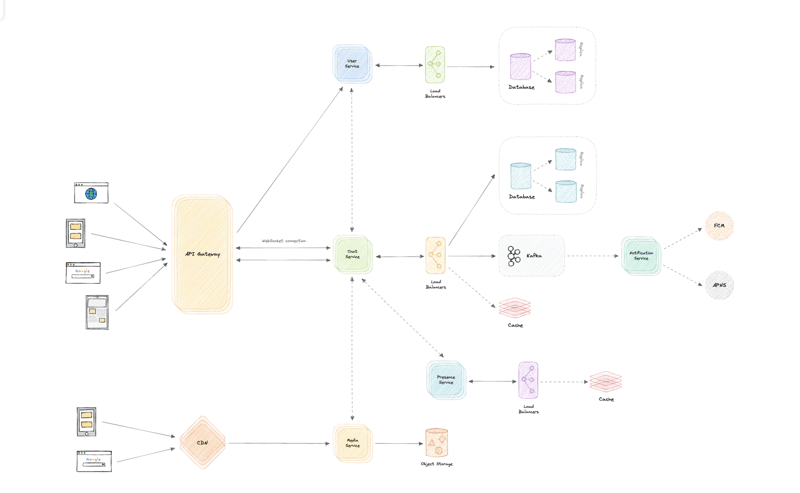
**API gateway**

Since we will be using multiple protocols like HTTP, WebSocket, TCP/IP, deploying multiple L4 (transport layer) or L7 (application layer) type load balancers separately for each protocol will be expensive. Instead, we can use an [API Gateway](https://karanpratapsingh.com/courses/system-design/api-gateway) that supports multiple protocols without any issues.

API Gateway can also offer other features such as authentication, authorization, rate limiting, throttling, and API versioning which will improve the quality of our services.

We can use services like [Amazon API Gateway](https://aws.amazon.com/api-gateway) or [Azure API Gateway](https://azure.microsoft.com/en-in/services/api-management) for this use case.

**Identify and resolve bottlenecks**

[](https://media2.dev.to/dynamic/image/width=800%2Cheight=%2Cfit=scale-down%2Cgravity=auto%2Cformat=auto/https%3A%2F%2Fdev-to-uploads.s3.amazonaws.com%2Fuploads%2Farticles%2Flbzpcdilnf9o1y8f34s4.png)

Let us identify and resolve bottlenecks such as single points of failure in our design:

* "What if one of our services crashes?"
* "How will we distribute our traffic between our components?"
* "How can we reduce the load on our database?"
* "How to improve the availability of our cache?"
* "Wouldn't API Gateway be a single point of failure?"
* "How can we make our notification system more robust?"
* "How can we reduce media storage costs"?
* "Does chat service has too much responsibility?"

To make our system more resilient we can do the following:

* Running multiple instances of each of our services.
* Introducing [load balancers](https://karanpratapsingh.com/courses/system-design/load-balancing) between clients, servers, databases, and cache servers.
* Using multiple read replicas for our databases.
* Multiple instances and replicas for our distributed cache.
* We can have a standby replica of our API Gateway.
* Exactly once delivery and message ordering is challenging in a distributed system, we can use a dedicated [message broker](https://karanpratapsingh.com/courses/system-design/message-brokers) such as [Apache Kafka](https://kafka.apache.org/) or [NATS](https://nats.io/) to make our notification system more robust.
* We can add media processing and compression capabilities to the media service to compress large files similar to Whatsapp which will save a lot of storage space and reduce cost.
* We can create a group service separate from the chat service to further decouple our services.

**System Design: SSL, TLS, mTLS**

Let's briefly discuss some important communication security protocols such as SSL, TLS, and mTLS. I would say that from a *"big picture"* system design perspective, this topic is not very important but still good to know about.

**SSL**

SSL stands for Secure Sockets Layer, and it refers to a protocol for encrypting and securing communications that take place on the internet. It was first developed in 1995 but since has been deprecated in favor of TLS (Transport Layer Security).

**Why is it called an SSL certificate if it is deprecated?**

Most major certificate providers still refer to certificates as SSL certificates, which is why the naming convention persists.

**Why was SSL so important?**

Originally, data on the web was transmitted in plaintext that anyone could read if they intercepted the message. SSL was created to correct this problem and protect user privacy. By encrypting any data that goes between the user and a web server, SSL also stops certain kinds of cyber attacks by preventing attackers from tampering with data in transit.

**TLS**

Transport Layer Security, or TLS, is a widely adopted security protocol designed to facilitate privacy and data security for communications over the internet. TLS evolved from a previous encryption protocol called Secure Sockets Layer (SSL). A primary use case of TLS is encrypting the communication between web applications and servers.

There are three main components to what the TLS protocol accomplishes:

* **Encryption**: hides the data being transferred from third parties.
* **Authentication**: ensures that the parties exchanging information are who they claim to be.
* **Integrity**: verifies that the data has not been forged or tampered with.

**mTLS**

Mutual TLS, or mTLS, is a method for mutual authentication. mTLS ensures that the parties at each end of a network connection are who they claim to be by verifying that they both have the correct private key. The information within their respective TLS certificates provides additional verification.

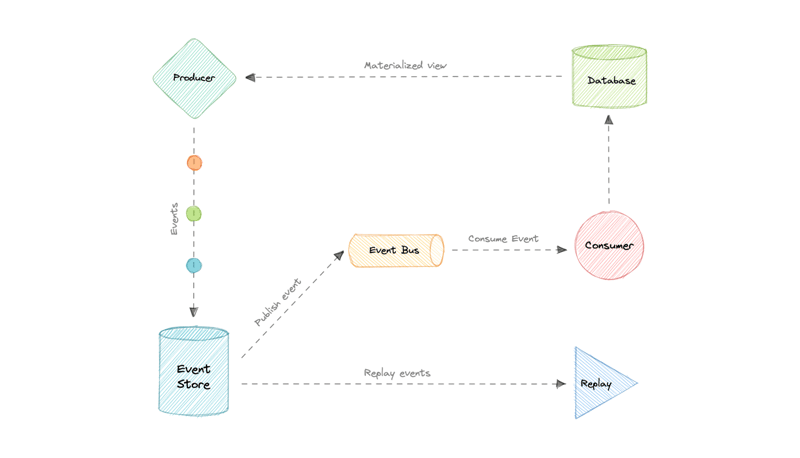
**Why use mTLS?**

mTLS helps ensure that the traffic is secure and trusted in both directions between a client and server. This provides an additional layer of security for users who log in to an organization's network or applications. It also verifies connections with client devices that do not follow a login process, such as Internet of Things (IoT) devices.

Nowadays, mTLS is commonly used by microservices or distributed systems in a [zero trust security model](https://en.wikipedia.org/wiki/Zero_trust_security_model) to verify each other.

**System Design: Event Sourcing**

Instead of storing just the current state of the data in a domain, use an append-only store to record the full series of actions taken on that data. The store acts as the system of record and can be used to materialize the domain objects.

[](https://media2.dev.to/dynamic/image/width=800%2Cheight=%2Cfit=scale-down%2Cgravity=auto%2Cformat=auto/https%3A%2F%2Fraw.githubusercontent.com%2Fkaranpratapsingh%2Fportfolio%2Fmaster%2Fpublic%2Fstatic%2Fcourses%2Fsystem-design%2Fchapter-III%2Fevent-sourcing%2Fevent-sourcing.png)

This can simplify tasks in complex domains, by avoiding the need to synchronize the data model and the business domain, while improving performance, scalability, and responsiveness. It can also provide consistency for transactional data, and maintain full audit trails and history that can enable compensating actions.

**Event sourcing vs Event-Driven Architecture (EDA)**

Event sourcing is seemingly constantly being confused with [Event-driven Architecture (EDA)](https://karanpratapsingh.com/courses/system-design/event-driven-architecture). Event-driven architecture is about using events to communicate between service boundaries. Generally, leveraging a message broker to publish and consume events asynchronously within other boundaries.

Whereas, event sourcing is about using events as a state, which is a different approach to storing data. Rather than storing the current state, we're instead going to be storing events. Also, event sourcing is one of the several patterns to implement an event-driven architecture.

**Advantages**

Let's discuss some advantages of using event sourcing:

* Excellent for real-time data reporting.
* Great for fail-safety, data can be reconstituted from the event store.
* Extremely flexible, any type of message can be stored.
* Preferred way of achieving audit logs functionality for high compliance systems.

**Disadvantages**

Following are the disadvantages of event sourcing:

* Requires an extremely efficient network infrastructure.
* Requires a reliable way to control message formats, such as a schema registry.
* Different events will contain different payloads.

**Event Sourcing in a Stock Trading System**

**Scenario:**

Imagine a simplified stock trading platform. Instead of directly updating the user's account balance and portfolio, we'll use event sourcing.

**Events:**

* **AccountCreated:** Emitted when a new trading account is created. Contains user details.
* **DepositMade:** Emitted when a user deposits funds into their account. Contains the amount deposited.
* **WithdrawalMade:** Emitted when a user withdraws funds from their account. Contains the amount withdrawn.
* **StockPurchased:** Emitted when a user buys shares of a stock. Contains the stock symbol, number of shares, and purchase price.
* **StockSold:** Emitted when a user sells shares of a stock. Contains the stock symbol, number of shares, and sale price.

**Implementation:**

1. **Event Store:**
   * Store all events in a chronological order (e.g., using a database like Apache Cassandra or an event store specifically designed for event sourcing).
   * Each event should have a unique identifier, timestamp, and the type of event.
2. **Event Handlers:**
   * Create event handlers that process each type of event.
   * For example:
     + The DepositMade event handler would increase the user's account balance.
     + The StockPurchased event handler would update the user's portfolio, deduct the purchase amount from the account balance, and potentially calculate and store the profit/loss.
3. **State Reconstruction:**
   * To get the current state of a user's account, the system replays all events for that user from the beginning.
   * By processing the events in order, the system can reconstruct the current account balance, portfolio holdings, and other relevant information.

**Benefits in this Context:**

* **Audit Trail:** Complete history of all account activities, enabling easy auditing, compliance, and fraud detection.
* **Data Recovery:** If data is lost or corrupted, the system can be easily restored by replaying the events from the beginning.
* **Flexibility:** New features and functionalities can be added by creating new event handlers without modifying existing code.
* **Time Travel Debugging:** Easily debug issues by replaying events up to a specific point in time and examining the system's state.
* **Reporting and Analytics:** The event stream can be used to generate reports, analyze trading patterns, and gain valuable insights into user behavior.

**Key Considerations:**

* **Event Store:** Choosing a suitable event store is crucial for performance, scalability, and durability.
* **Event Handling:** Designing and implementing efficient and reliable event handlers is essential.
* **Data Consistency:** Ensuring data consistency across different components and services can be challenging in event-sourced systems.

This is a simplified example. In a real-world stock trading system, event sourcing would involve more complex event types, sophisticated event handling logic, and robust mechanisms for ensuring data consistency and scalability.

This example demonstrates how event sourcing can be applied to a real-world scenario, providing a clear audit trail, improved flexibility, and enhanced data integrity.

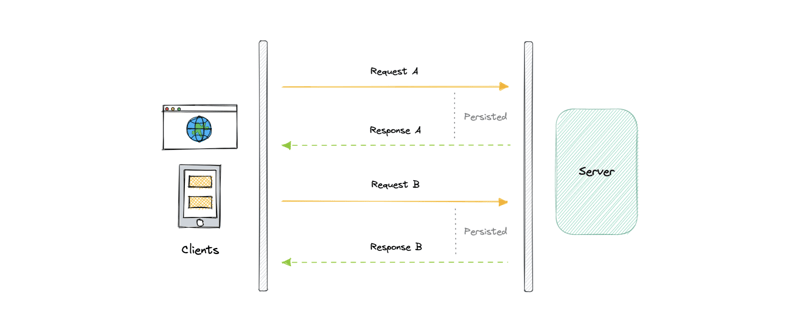
**System Design: Long polling, WebSockets, Server-Sent Events (SSE)**

Web applications were initially developed around a client-server model, where the web client is always the initiator of transactions like requesting data from the server. Thus, there was no mechanism for the server to independently send, or push, data to the client without the client first making a request. Let's discuss some approaches to overcome this problem.

**Long polling**

HTTP Long polling is a technique used to push information to a client as soon as possible from the server. As a result, the server does not have to wait for the client to send a request.

In Long polling, the server does not close the connection once it receives a request from the client. Instead, the server responds only if any new message is available or a timeout threshold is reached.

[](https://media2.dev.to/dynamic/image/width=800%2Cheight=%2Cfit=scale-down%2Cgravity=auto%2Cformat=auto/https%3A%2F%2Fdev-to-uploads.s3.amazonaws.com%2Fuploads%2Farticles%2Frx9e0l33vp315nzz13xn.png)

Once the client receives a response, it immediately sends a new request to the server to have a new pending connection to send data to the client, and the operation is repeated. With this approach, the server emulates a real-time server push feature.

**Working**

Let's understand how long polling works:

1. The client makes an initial request and waits for a response.
2. The server receives the request and delays sending anything until an update is available.
3. Once an update is available, the response is sent to the client.
4. The client receives the response and makes a new request immediately or after some defined interval to establish a connection again.

**Advantages**

Here are some advantages of long polling:

* Easy to implement, good for small-scale projects.
* Nearly universally supported.

**Disadvantages**

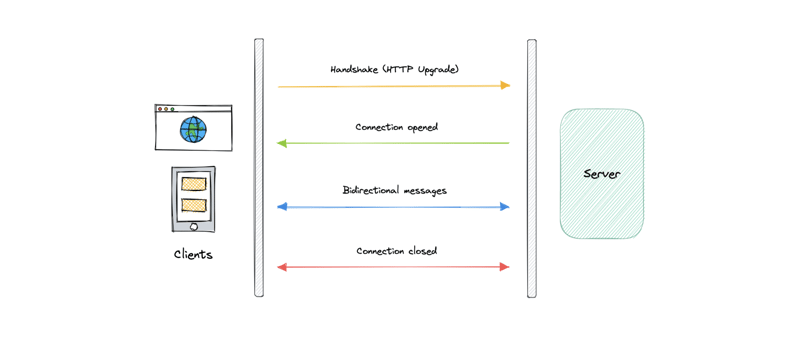
A major downside of long polling is that it is usually not scalable. Below are some of the other reasons:

* Creates a new connection each time, which can be intensive on the server.
* Reliable message ordering can be an issue for multiple requests.
* Increased latency as the server needs to wait for a new request.

**WebSockets**

WebSocket provides full-duplex communication channels over a single TCP connection. It is a persistent connection between a client and a server that both parties can use to start sending data at any time.

The client establishes a WebSocket connection through a process known as the WebSocket handshake. If the process succeeds, then the server and client can exchange data in both directions at any time. The WebSocket protocol enables the communication between a client and a server with lower overheads, facilitating real-time data transfer from and to the server.

[](https://media2.dev.to/dynamic/image/width=800%2Cheight=%2Cfit=scale-down%2Cgravity=auto%2Cformat=auto/https%3A%2F%2Fdev-to-uploads.s3.amazonaws.com%2Fuploads%2Farticles%2Fmxpeej9ax0p2cq7f6uwe.png)

This is made possible by providing a standardized way for the server to send content to the client without being asked and allowing for messages to be passed back and forth while keeping the connection open.

**Working**

Let's understand how WebSockets work:

1. The client initiates a WebSocket handshake process by sending a request.
2. The request also contains an [HTTP Upgrade](https://en.wikipedia.org/wiki/HTTP/1.1_Upgrade_header) header that allows the request to switch to the WebSocket protocol (ws://).
3. The server sends a response to the client, acknowledging the WebSocket handshake request.
4. A WebSocket connection will be opened once the client receives a successful handshake response.
5. Now the client and server can start sending data in both directions allowing real-time communication.
6. The connection is closed once the server or the client decides to close the connection.

**Advantages**

Below are some advantages of WebSockets:

* Full-duplex asynchronous messaging.
* Better origin-based security model.
* Lightweight for both client and server.

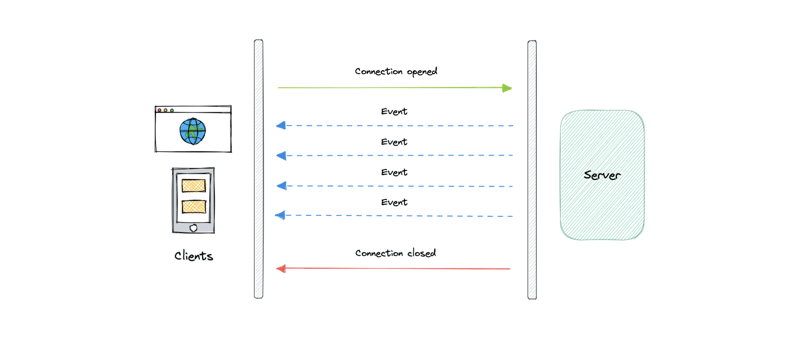
**Disadvantages**

Let's discuss some disadvantages of WebSockets:

* Terminated connections aren't automatically recovered.
* Older browsers don't support WebSockets (becoming less relevant).

**Server-Sent Events (SSE)**

Server-Sent Events (SSE) is a way of establishing long-term communication between client and server that enables the server to proactively push data to the client.

[](https://media2.dev.to/dynamic/image/width=800%2Cheight=%2Cfit=scale-down%2Cgravity=auto%2Cformat=auto/https%3A%2F%2Fdev-to-uploads.s3.amazonaws.com%2Fuploads%2Farticles%2Fjc98ezf8nehh51vg22d5.png)

It is unidirectional, meaning once the client sends the request it can only receive the responses without the ability to send new requests over the same connection.

**Working**

Let's understand how server-sent events work:

1. The client makes a request to the server.
2. The connection between client and server is established and it remains open.
3. The server sends responses or events to the client when new data is available.

**Advantages**

* Simple to implement and use for both client and server.
* Supported by most browsers.
* No trouble with firewalls.

**Disadvantages**

* Unidirectional nature can be limiting.
* Limitation for the maximum number of open connections.
* Does not support binary data.

**System Design: Service Discovery**

Service discovery is the detection of services within a computer network. Service Discovery Protocol (SDP) is a networking standard that accomplishes the detection of networks by identifying resources.

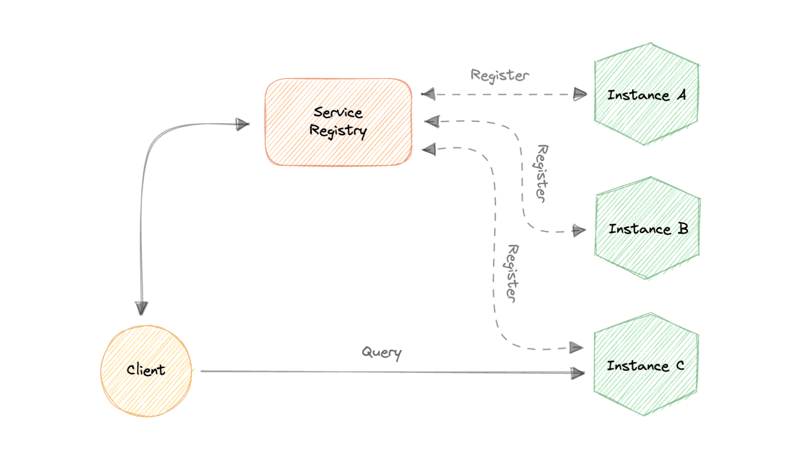
**Why do we need Service Discovery?**

In a monolithic application, services invoke one another through language-level methods or procedure calls. However, modern microservices-based applications typically run in virtualized or containerized environments where the number of instances of a service and their locations change dynamically. Consequently, we need a mechanism that enables the clients of service to make requests to a dynamically changing set of ephemeral service instances.

**Implementations**

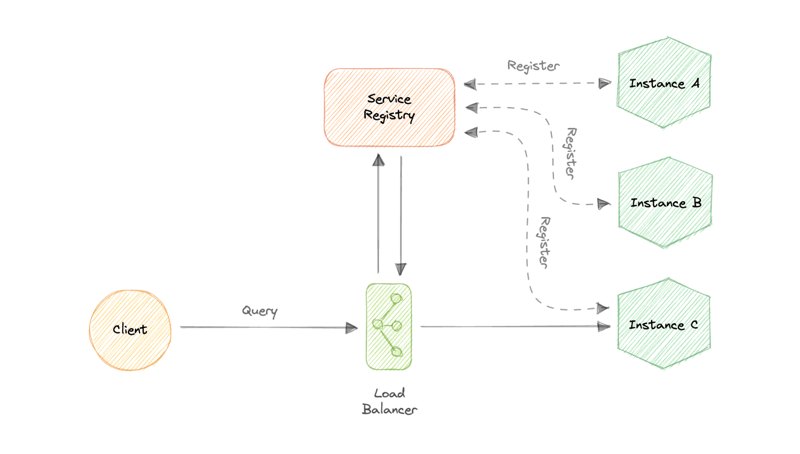
There are two main service discovery patterns:

**Client-side discovery**

[](https://media2.dev.to/dynamic/image/width=800%2Cheight=%2Cfit=scale-down%2Cgravity=auto%2Cformat=auto/https%3A%2F%2Fraw.githubusercontent.com%2Fkaranpratapsingh%2Fportfolio%2Fmaster%2Fpublic%2Fstatic%2Fcourses%2Fsystem-design%2Fchapter-IV%2Fservice-discovery%2Fclient-side-service-discovery.png)

In this approach, the client obtains the location of another service by querying a service registry which is responsible for managing and storing the network locations of all the services.

**Server-side discovery**

[](https://media2.dev.to/dynamic/image/width=800%2Cheight=%2Cfit=scale-down%2Cgravity=auto%2Cformat=auto/https%3A%2F%2Fraw.githubusercontent.com%2Fkaranpratapsingh%2Fportfolio%2Fmaster%2Fpublic%2Fstatic%2Fcourses%2Fsystem-design%2Fchapter-IV%2Fservice-discovery%2Fserver-side-service-discovery.png)

In this approach, we use an intermediate component such as a load balancer. The client makes a request to the service via a load balancer which then forwards the request to an available service instance.

**Service Registry**

A service registry is basically a database containing the network locations of service instances to which the clients can reach out. A Service Registry must be highly available and up-to-date.

**Service Registration**

We also need a way to obtain service information, often known as service registration. Let's look at two possible service registration approaches:

**Self-Registration**

When using the self-registration model, a service instance is responsible for registering and de-registering itself in the Service Registry. In addition, if necessary, a service instance sends heartbeat requests to keep its registration alive.

**Third-party Registration**

The registry keeps track of changes to running instances by polling the deployment environment or subscribing to events. When it detects a newly available service instance, it records it in its database. The Service Registry also de-registers terminated service instances.

**Service mesh**

Service-to-service communication is essential in a distributed application but routing this communication, both within and across application clusters, becomes increasingly complex as the number of services grows. Service mesh enables managed, observable, and secure communication between individual services. It works with a service discovery protocol to detect services. [Istio](https://istio.io/latest/about/service-mesh) and [envoy](https://www.envoyproxy.io/) are some of the most commonly used service mesh technologies.

**Examples**

Here are some commonly used service discovery infrastructure tools:

* [etcd](https://etcd.io/)
* [Consul](https://www.consul.io/)
* [Apache Thrift](https://thrift.apache.org/)
* [Apache Zookeeper](https://zookeeper.apache.org/)

**System Design: Virtual Machines (VMs) and Containers**

Before we discuss virtualization vs containerization, let's learn what are virtual machines (VMs) and Containers.

**Virtual Machines (VM)**

A Virtual Machine (VM) is a virtual environment that functions as a virtual computer system with its own CPU, memory, network interface, and storage, created on a physical hardware system. A software called a hypervisor separates the machine's resources from the hardware and provisions them appropriately so they can be used by the VM.

VMs are isolated from the rest of the system, and multiple VMs can exist on a single piece of hardware, like a server. They can be moved between host servers depending on the demand or to use resources more efficiently.

**What is a Hypervisor?**

A Hypervisor sometimes called a Virtual Machine Monitor (VMM), isolates the operating system and resources from the virtual machines and enables the creation and management of those VMs. The hypervisor treats resources like CPU, memory, and storage as a pool of resources that can be easily reallocated between existing guests or new virtual machines.

**Why use a Virtual Machine?**

Server consolidation is a top reason to use VMs. Most operating system and application deployments only use a small amount of the physical resources available. By virtualizing our servers, we can place many virtual servers onto each physical server to improve hardware utilization. This keeps us from needing to purchase additional physical resources.

A VM provides an environment that is isolated from the rest of a system, so whatever is running inside a VM won't interfere with anything else running on the host hardware. Because VMs are isolated, they are a good option for testing new applications or setting up a production environment. We can also run a single-purpose VM to support a specific use case.

**Containers**

A container is a standard unit of software that packages up code and all its dependencies such as specific versions of runtimes and libraries so that the application runs quickly and reliably from one computing environment to another. Containers offer a logical packaging mechanism in which applications can be abstracted from the environment in which they actually run. This decoupling allows container-based applications to be deployed easily and consistently, regardless of the target environment.

**Why do we need containers?**

Let's discuss some advantages of using containers:

**Separation of responsibility**

Containerization provides a clear separation of responsibility, as developers focus on application logic and dependencies, while operations teams can focus on deployment and management.

**Workload portability**

Containers can run virtually anywhere, greatly easing development and deployment.

**Application isolation**

Containers virtualize CPU, memory, storage, and network resources at the operating system level, providing developers with a view of the OS logically isolated from other applications.

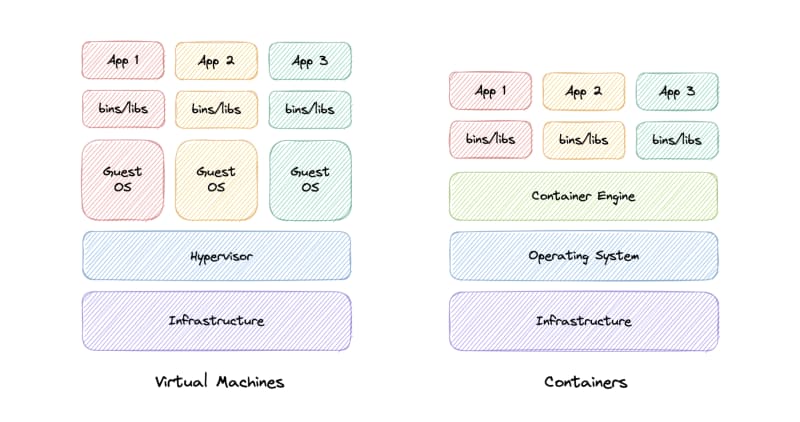
**Agile development**

Containers allow developers to move much more quickly by avoiding concerns about dependencies and environments.

**Efficient operations**

Containers are lightweight and allow us to use just the computing resources we need.

**Virtualization vs Containerization**

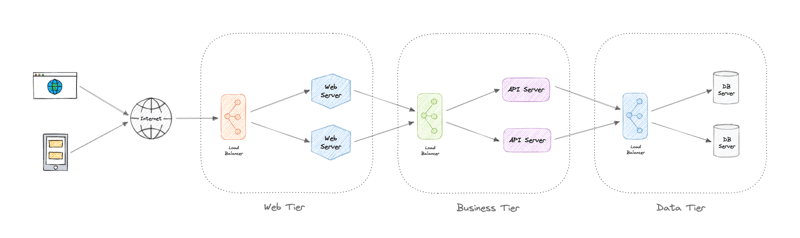
[](https://res.cloudinary.com/practicaldev/image/fetch/s--mn651v4---/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_800/https:/raw.githubusercontent.com/karanpratapsingh/portfolio/master/public/static/courses/system-design/chapter-IV/virtual-machines-and-containers/virtualization-vs-containerization.png)

In traditional virtualization, a hypervisor virtualizes physical hardware. The result is that each virtual machine contains a guest OS, a virtual copy of the hardware that the OS requires to run, and an application and its associated libraries and dependencies.

Instead of virtualizing the underlying hardware, containers virtualize the operating system so each container contains only the application and its dependencies making them much more lightweight than VMs. Containers also share the OS kernel and use a fraction of the memory VMs require.

**System Design: N-tier architecture**

N-tier architecture divides an application into logical layers and physical tiers. Layers are a way to separate responsibilities and manage dependencies. Each layer has a specific responsibility. A higher layer can use services in a lower layer, but not the other way around.

[](https://media2.dev.to/dynamic/image/width=800%2Cheight=%2Cfit=scale-down%2Cgravity=auto%2Cformat=auto/https%3A%2F%2Fraw.githubusercontent.com%2Fkaranpratapsingh%2Fportfolio%2Fmaster%2Fpublic%2Fstatic%2Fcourses%2Fsystem-design%2Fchapter-III%2Fn-tier-architecture%2Fn-tier-architecture.png)

Tiers are physically separated, running on separate machines. A tier can call to another tier directly, or use asynchronous messaging. Although each layer might be hosted in its own tier, that's not required. Several layers might be hosted on the same tier. Physically separating the tiers improves scalability and resiliency and adds latency from the additional network communication.

An N-tier architecture can be of two types:

* In a closed layer architecture, a layer can only call the next layer immediately down.
* In an open layer architecture, a layer can call any of the layers below it.

A closed-layer architecture limits the dependencies between layers. However, it might create unnecessary network traffic, if one layer simply passes requests along to the next layer.

**Types of N-Tier architectures**

Let's look at some examples of N-Tier architecture:

**3-Tier architecture**

3-Tier is widely used and consists of the following different layers:

* **Presentation layer**: Handles user interactions with the application.
* **Business Logic layer**: Accepts the data from the application layer, validates it as per business logic and passes it to the data layer.
* **Data Access layer**: Receives the data from the business layer and performs the necessary operation on the database.

**2-Tier architecture**

In this architecture, the presentation layer runs on the client and communicates with a data store. There is no business logic layer or immediate layer between client and server.

**Single Tier or 1-Tier architecture**

It is the simplest one as it is equivalent to running the application on a personal computer. All of the required components for an application to run are on a single application or server.

**Advantages**

Here are some advantages of using N-tier architecture:

* Can improve availability.
* Better security as layers can behave like a firewall.
* Separate tiers allow us to scale them as needed.
* Improve maintenance as different people can manage different tiers.

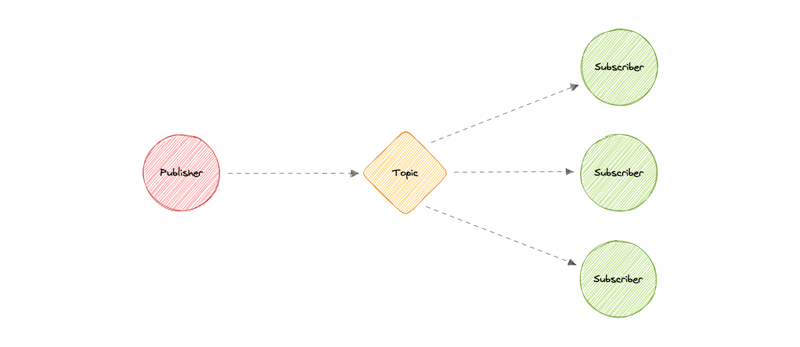
**Disadvantages**

Below are some disadvantages of N-tier architecture:

* Increased complexity of the system as a whole.
* Increased network latency as the number of tiers increases.
* Expensive as every tier will have its own hardware

**System Design: Publish-Subscribe**

Similar to a message queue, publish-subscribe is also a form of service-to-service communication that facilitates asynchronous communication. In a pub/sub model, any message published to a topic is pushed immediately to all the subscribers of the topic.

[](https://media2.dev.to/dynamic/image/width=800%2Cheight=%2Cfit=scale-down%2Cgravity=auto%2Cformat=auto/https%3A%2F%2Fraw.githubusercontent.com%2Fkaranpratapsingh%2Fportfolio%2Fmaster%2Fpublic%2Fstatic%2Fcourses%2Fsystem-design%2Fchapter-III%2Fpublish-subscribe%2Fpublish-subscribe.png)

The subscribers to the message topic often perform different functions, and can each do something different with the message in parallel. The publisher doesn't need to know who is using the information that it is broadcasting, and the subscribers don't need to know where the message comes from. This style of messaging is a bit different than message queues, where the component that sends the message often knows the destination it is sending to.

**Working**

Unlike message queues, which batch messages until they are retrieved, message topics transfer messages with little or no queuing and push them out immediately to all subscribers. Here's how it works:

* A message topic provides a lightweight mechanism to broadcast asynchronous event notifications and endpoints that allow software components to connect to the topic in order to send and receive those messages.
* To broadcast a message, a component called a *publisher* simply pushes a message to the topic.
* All components that subscribe to the topic (known as *subscribers*) will receive every message that was broadcasted.

**Advantages**

Let's discuss some advantages of using publish-subscribe:

* **Eliminate Polling**: Message topics allow instantaneous, push-based delivery, eliminating the need for message consumers to periodically check or *"poll"* for new information and updates. This promotes faster response time and reduces the delivery latency which can be particularly problematic in systems where delays cannot be tolerated.
* **Dynamic Targeting**: Pub/Sub makes the discovery of services easier, more natural, and less error-prone. Instead of maintaining a roster of peers where an application can send messages, a publisher will simply post messages to a topic. Then, any interested party will subscribe its endpoint to the topic, and start receiving these messages. Subscribers can change, upgrade, multiply or disappear and the system dynamically adjusts.
* **Decoupled and Independent Scaling**: Publishers and subscribers are decoupled and work independently from each other, which allows us to develop and scale them independently.
* **Simplify Communication**: The Publish-Subscribe model reduces complexity by removing all the point-to-point connections with a single connection to a message topic, which will manage subscriptions and decide what messages should be delivered to which endpoints.

**Features**

Now, let's discuss some desired features of publish-subscribe:

**Push Delivery**

Pub/Sub messaging instantly pushes asynchronous event notifications when messages are published to the message topic. Subscribers are notified when a message is available.

**Multiple Delivery Protocols**

In the Publish-Subscribe model, topics can typically connect to multiple types of endpoints, such as message queues, serverless functions, HTTP servers, etc.

**Fanout**

This scenario happens when a message is sent to a topic and then replicated and pushed to multiple endpoints. Fanout provides asynchronous event notifications which in turn allows for parallel processing.

**Filtering**

This feature empowers the subscriber to create a message filtering policy so that it will only get the notifications it is interested in, as opposed to receiving every single message posted to the topic.

**Durability**

Pub/Sub messaging services often provide very high durability, and at least once delivery, by storing copies of the same message on multiple servers.

**Security**

Message topics authenticate applications that try to publish content, this allows us to use encrypted endpoints and encrypt messages in transit over the network.

**Examples**

Here are some technologies commonly used for publish-subscribe:

* [Amazon SNS](https://aws.amazon.com/sns)
* [Google Pub/Sub](https://cloud.google.com/pubsub)

**The Simplest Guide To OAuth 2.0**

[[Takahiko Kawasaki](https://darutk.medium.com/?source=post_page---byline--8c71bd9a15bb--------------------------------)](https://darutk.medium.com/?source=post_page---byline--8c71bd9a15bb--------------------------------)

[Takahiko Kawasaki](https://darutk.medium.com/?source=post_page---byline--8c71bd9a15bb--------------------------------)

·

[Follow](https://medium.com/m/signin?actionUrl=https%3A%2F%2Fmedium.com%2F_%2Fsubscribe%2Fuser%2Ff485feca545e&operation=register&redirect=https%3A%2F%2Fdarutk.medium.com%2Fthe-simplest-guide-to-oauth-2-0-8c71bd9a15bb&user=Takahiko+Kawasaki&userId=f485feca545e&source=post_page-f485feca545e--byline--8c71bd9a15bb---------------------post_header-----------)

6 min read

·

Aug 1, 2017

7.2K

38

For the past three years, I've repeated to explain OAuth 2.0 to those who don't have a technical background, mainly to investors as a co-founder of [Authlete, Inc.](https://www.authlete.com/" \t "_blank) (Tech In Asia: *"*[*API security startup Authlete raises $1.2m in seed funding*](https://www.techinasia.com/authlete-gets-seed-funding-500-startups)*"*). As a result, I found a way to explain OAuth 2.0 in an easily understandable manner. This article introduces the steps.

**1. There are data of a user.**

A white background with black and white objects

Description automatically generated with medium confidence

**2. There is a server which manages the user's data. The server is called "Resource Server".**

A white background with black and white objects

Description automatically generated with medium confidence

**3. There is a "Client Application" which wants to use the user's data.**

A white background with black and white objects

Description automatically generated with medium confidence

**4. Let's prepare a gate to pass the user's data through. The gate is called "API".**

A green and white text on a white background

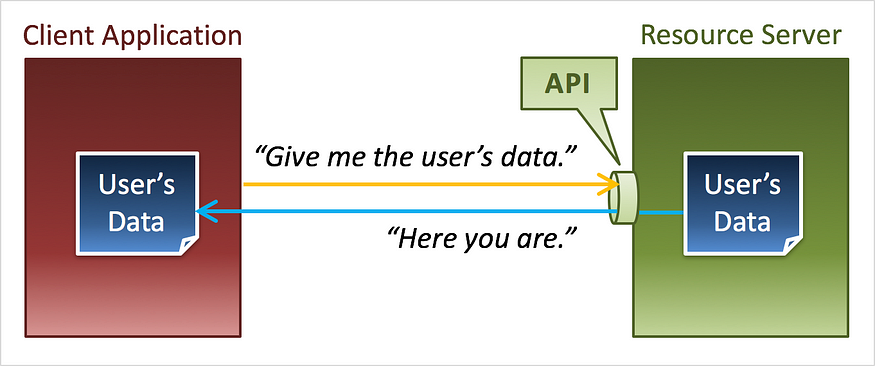
Description automatically generated with medium confidence

**5. The client application requests the user's data.**

A close-up of a computer

Description automatically generated

**6. The resource server returns the user's data.**

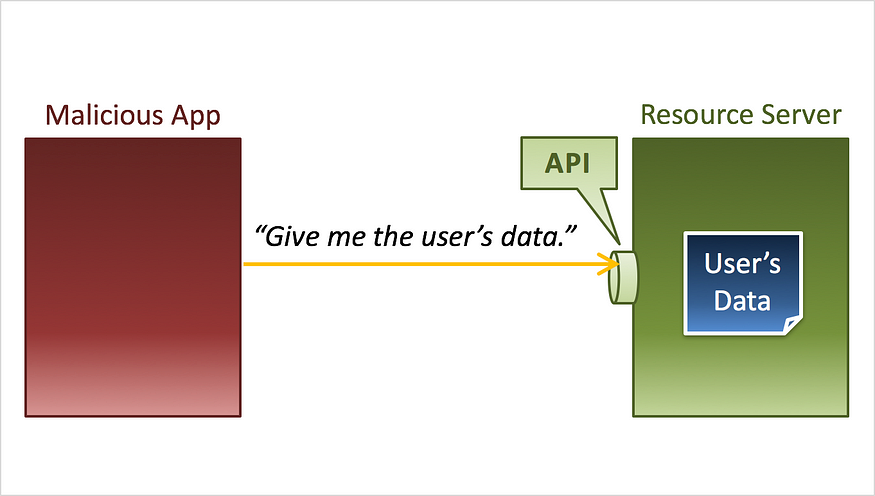


**7. What if there is a malicious client application?**

A close-up of a diagram

Description automatically generated

**8. Even if the client application that requests the user's data is a malicious one, ...**

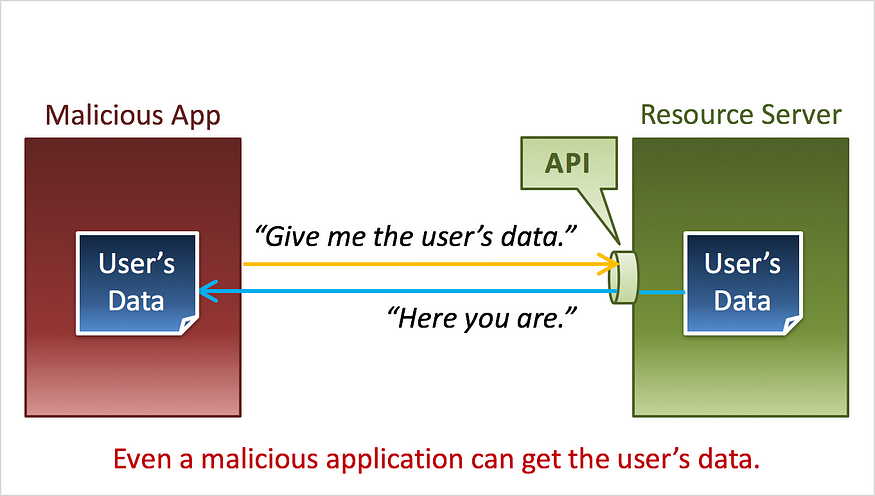


**9. ... the resource server returns the user's data.**

A diagram of a software application

Description automatically generated

**10. Even a malicious application can get the user's data.**

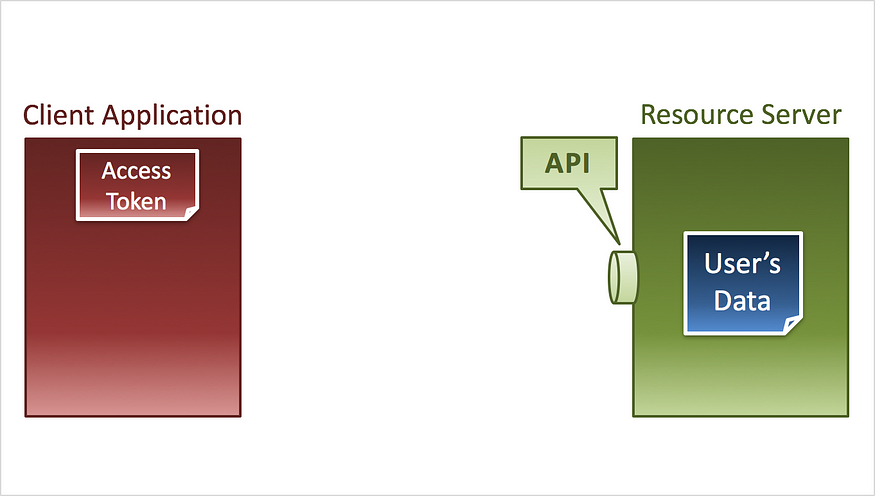


**11. We need a mechanism to protect the user's data.**

A diagram of a computer system

Description automatically generated

**12. In the best practice, an "Access Token" is given to the client application in advance. An access token represents that the said client application has been given permissions to access the user's data.**

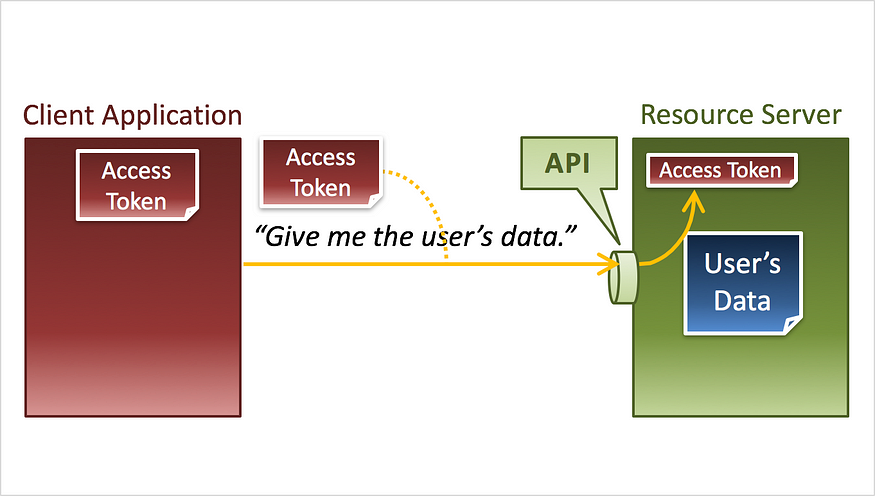


**13. The client application presents the access token when it requests the user's data.**

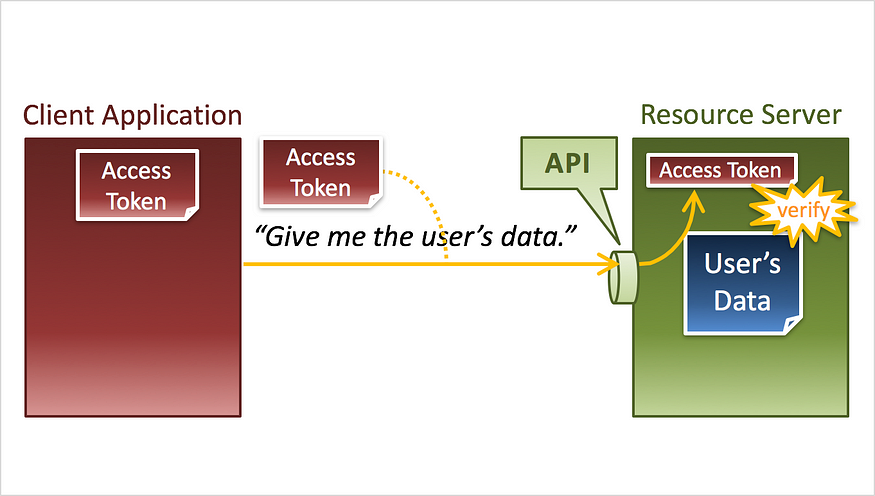
A diagram of a software application

Description automatically generated

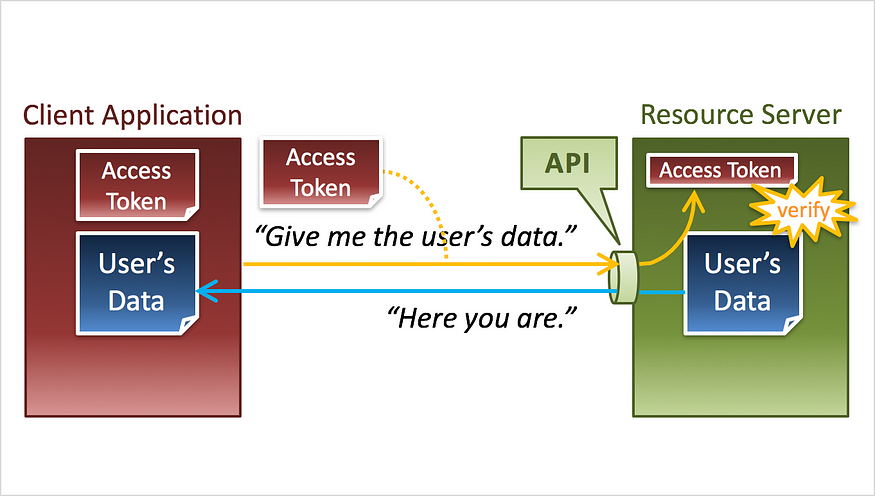
**14. The resource server extracts the access token that is included in the request, ...**



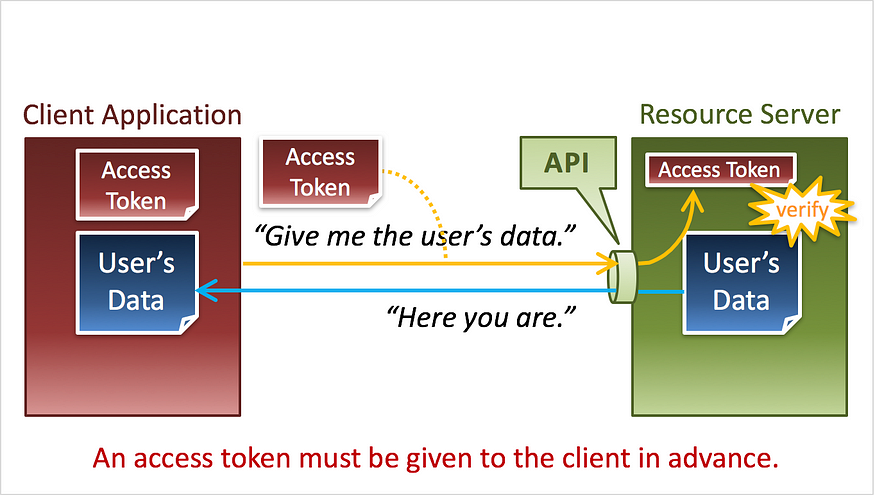
**15. ... and confirms that the access token denotes that the client application has permissions to access the user's data.**



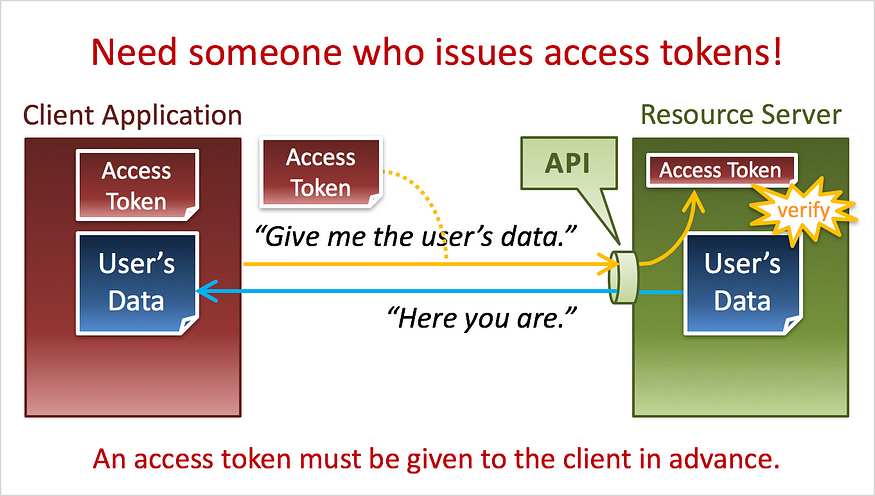
**16. After the confirmation, the resource server returns the user's data.**



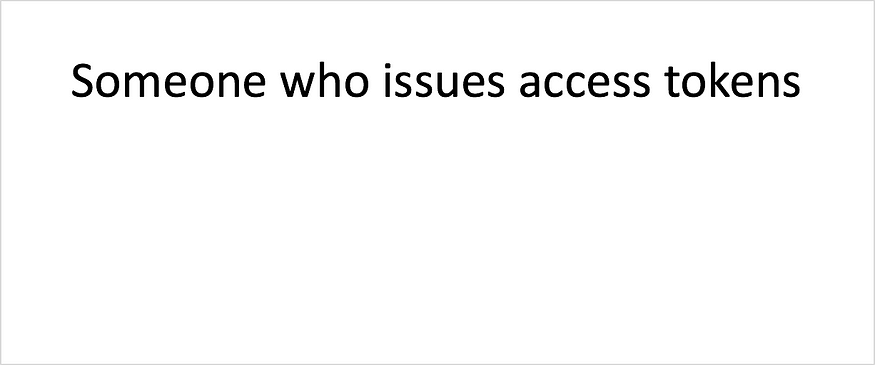
**17. To make this mechanism work, an access token must be given to the client application in advance.**



**18. Consequently, we need someone who issues access tokens.**



**19. Someone who issues access tokens ...**

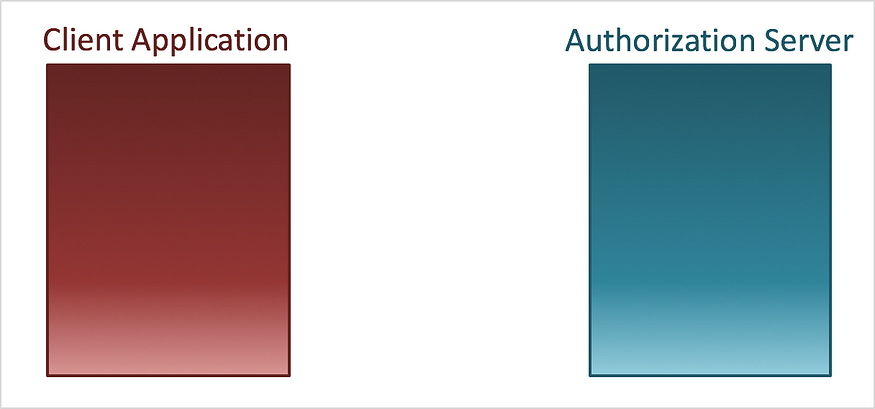


**20. ... is called "Authorization Server".**

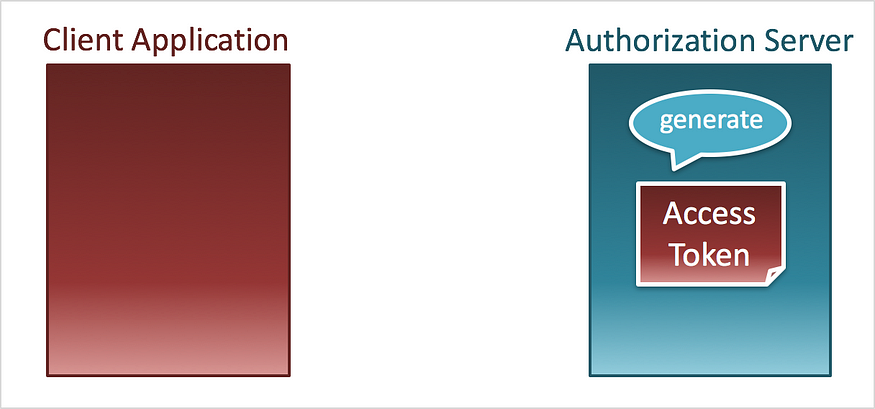
A close-up of a sign

Description automatically generated

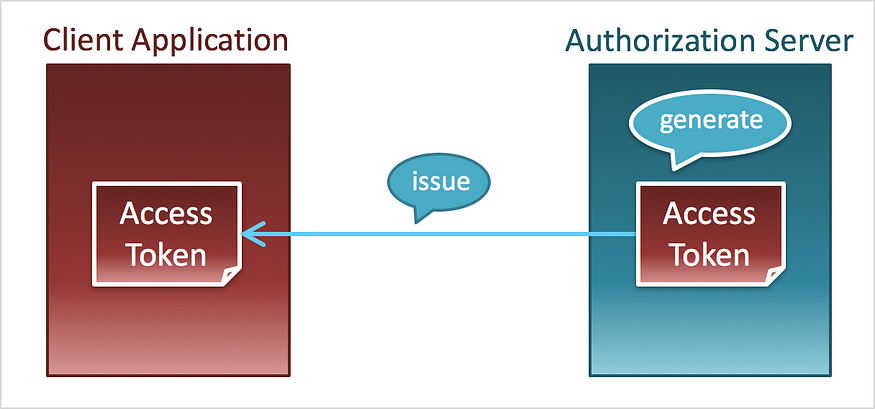
**21. The relationship between a client application and an authorization server is as follows.**



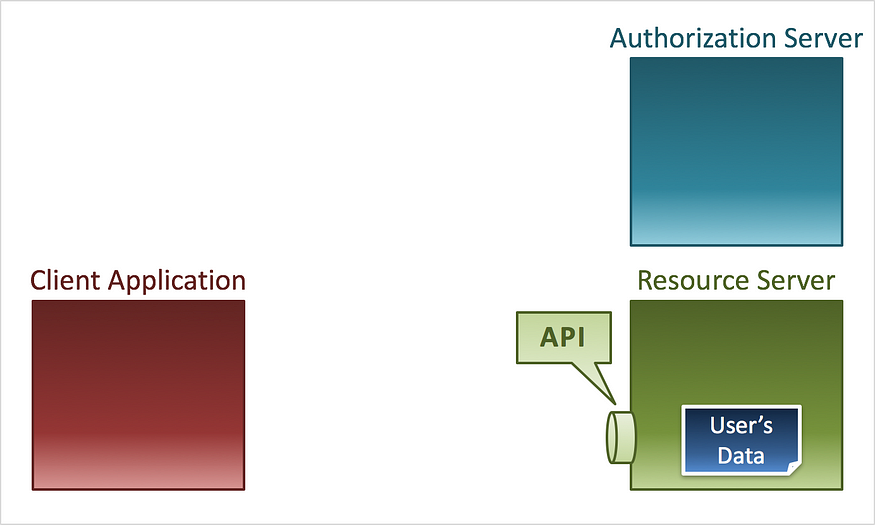
**22. An authorization server generates an access token ...**



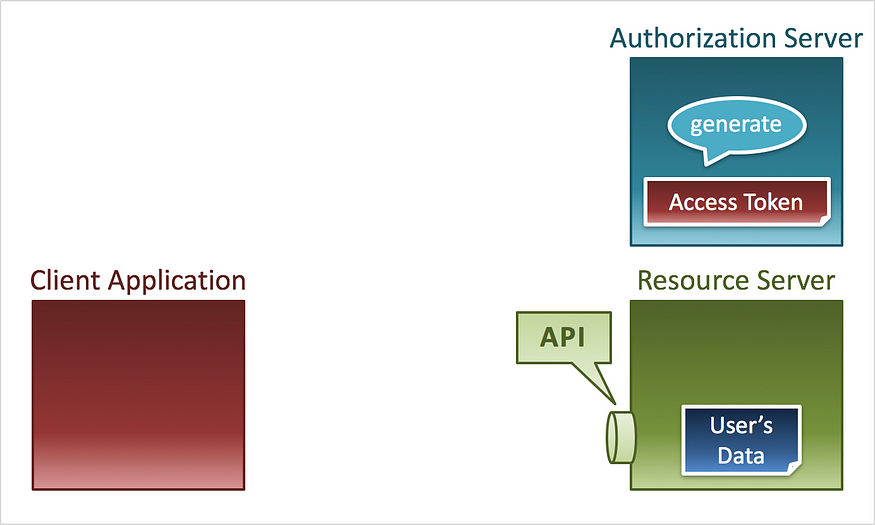
**23. ... and issues the access token to a client application.**



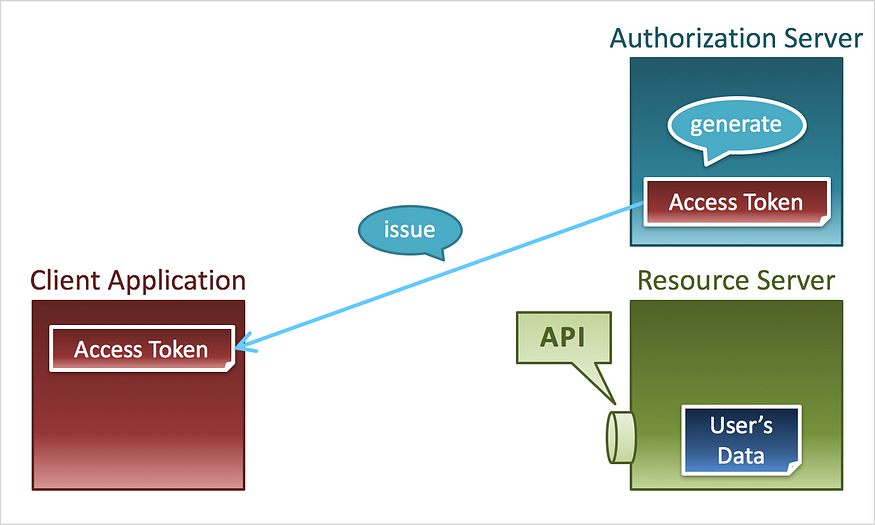
**24. Let's review what we've learned so far. Characters are an "Authorization Server", a "Client Application" and a "Resource Server".**



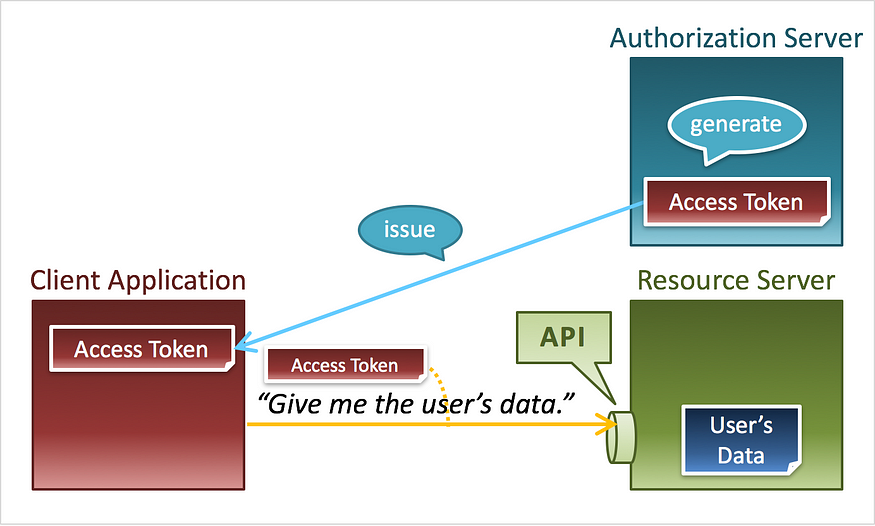
**25. The authorization server generates an access token ...**



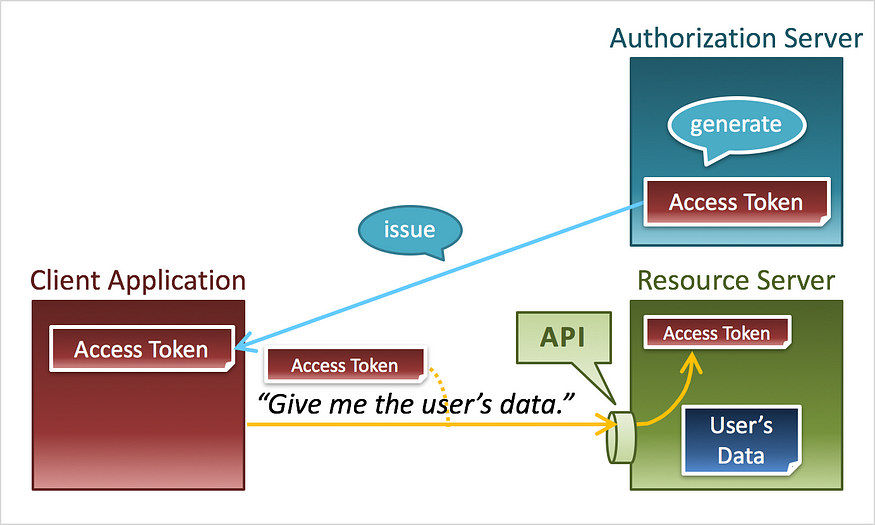
**26. ... and issues the access token to the client application.**



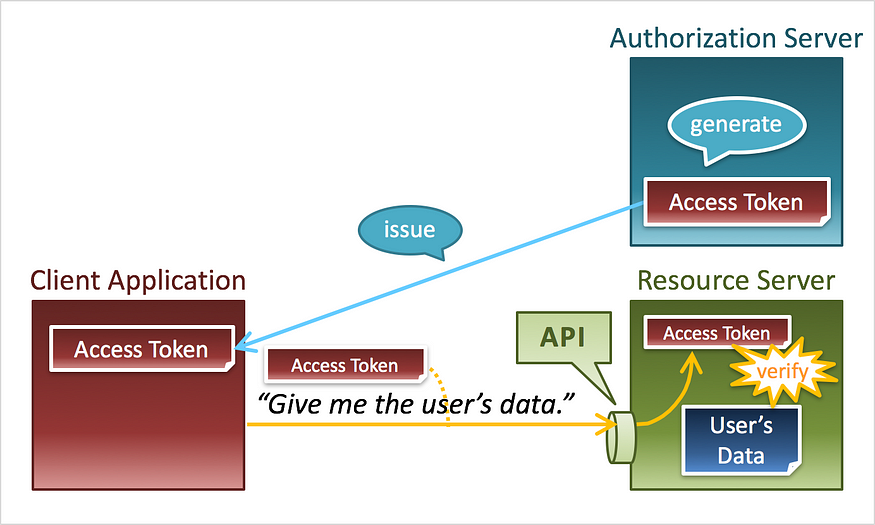
**27. The client application requests the user's data with the access token.**



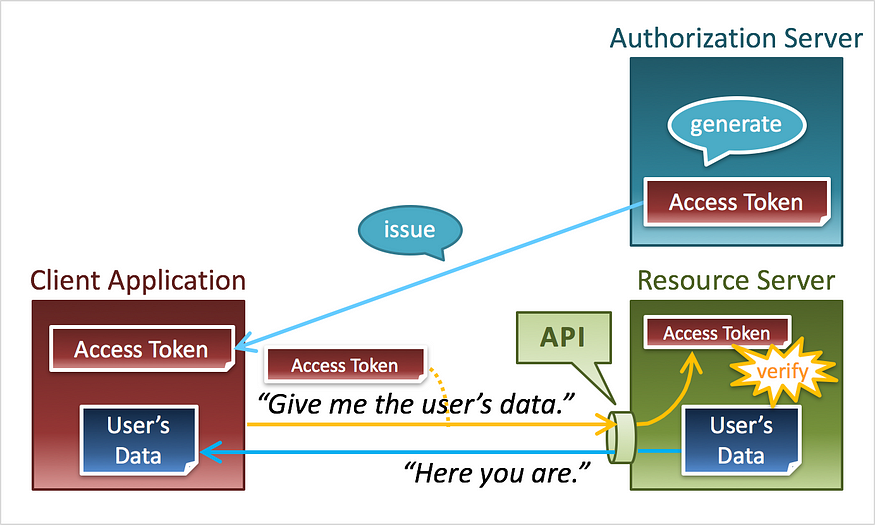
**28. The resource server extracts the access token from the request, ...**



**29. ... confirms that the access token has permissions to access the user's data ...**



**30. ... and returns the user's data to the client application.**



**31. In the flow above, the first step is access token generation by an authorization server. However, in a real flow, the user is asked before an access token is issued.**

A close-up of a blue and red box

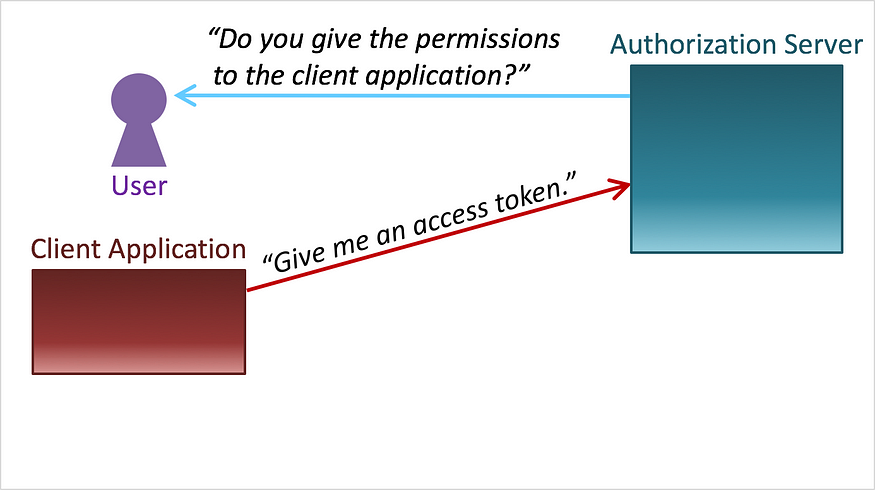
Description automatically generated

**32. First, the client application requests an access token.**

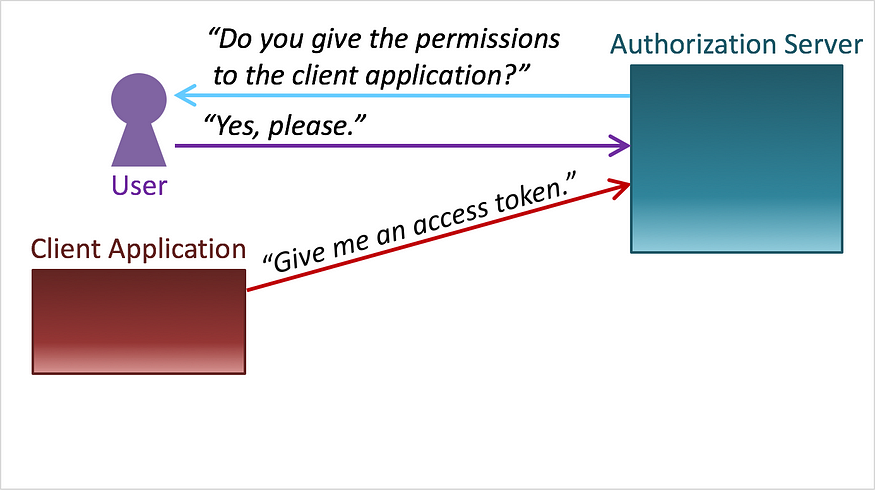
A diagram of a diagram

Description automatically generated with medium confidence

**33. Then, the authorization server asks the user whether to grant the requested permissions to the client application.**



**34. If the user allows the authorization server to issue an access token to the client application, ...**



**35. ... the authorization server generates an access token ...**

A diagram of a process

Description automatically generated

**36. ... and issues the access token to the client application.**

A diagram of a process

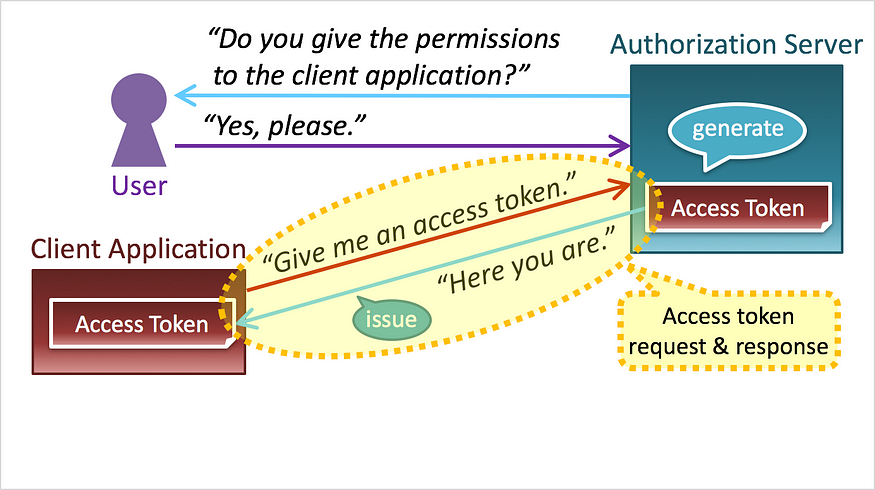
Description automatically generated

**37. By the way, pay attention to the part encircled by the yellow ellipse.**

A diagram of a process

Description automatically generated

**38. The part represents an access token request and a response to the request.**



**39. And, it is "OAuth 2.0" that has standardized the part. Details of OAuth 2.0 are described in the technical document,**[**RFC 6749**](https://tools.ietf.org/html/rfc6749)**(The OAuth 2.0 Authorization Framework).**

A diagram of a flowchart

Description automatically generated

**Next To Read**

Q: JWT vs Session Authentication ?

A:

**JWT vs Session Authentication**

[#jwt](https://dev.to/t/jwt)[#authjs](https://dev.to/t/authjs)[#javascript](https://dev.to/t/javascript)[#security](https://dev.to/t/security)

**Authentication vs Authorization**

What exactly is authentication, and how does it differ from authorization? Authentication is the process of verifying who someone is, whereas authorization is the process of verifying what specific applications, files, and data a user has access to. How do you ensure that the person requesting access to a resource is who they claim to be? And once their identity is confirmed, how do you control what they can do or see?

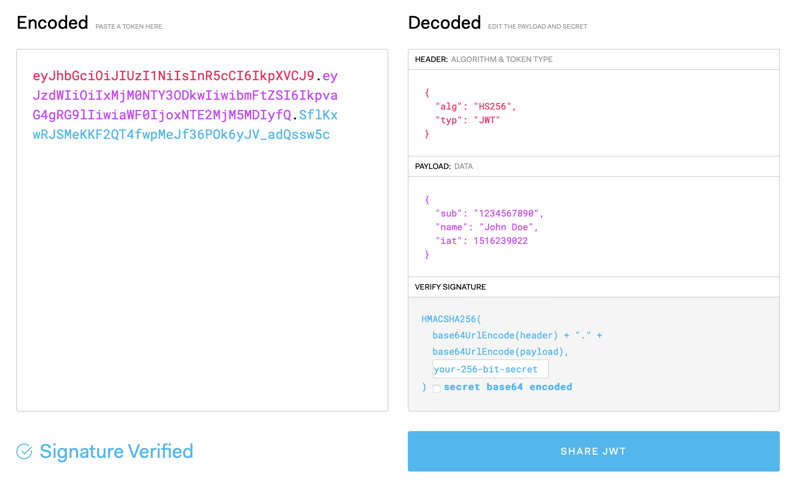
**JWT vs. Session Authentication - The Basic Differences**

The debate between **JWT (JSON Web Token)** and **Session-Based Authentication** is a important point in modern web development.

* **JWT Authentication**: Here, the server generates a token that the client stores and presents with each request. It's a stateless method, meaning the server doesn't need to keep a record of the token.
* **Session-Based Authentication**: Contrarily, it's stateful. The server creates a session for the user and stores session data on the server-side. The client holds only a session identifier, typically in a cookie.

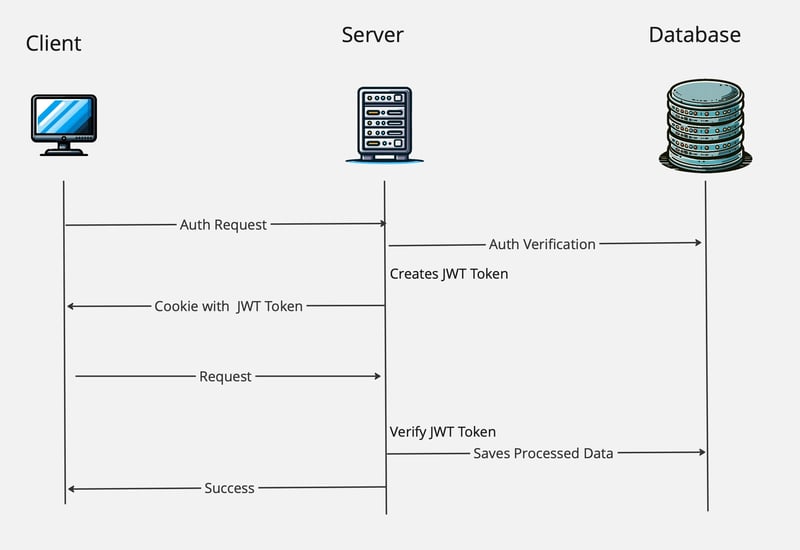
**What is JWT?**

**JSON Web Token (JWT)** serves as a compact and self-contained mechanism for securely transmitting information between parties as a JSON object.

[](https://media2.dev.to/dynamic/image/width=800%2Cheight=%2Cfit=scale-down%2Cgravity=auto%2Cformat=auto/https%3A%2F%2Fdev-to-uploads.s3.amazonaws.com%2Fuploads%2Farticles%2Ffydm4uic65eaiv956hyp.png)

**JWT Structure:**

* **Header:** Specifies the token type (JWT) and the signing algorithm (e.g., HMAC SHA256).
* **Payload:** Contains the claims, which are statements about an entity (user) and additional metadata.
* **Signature:** Created by encoding the header and payload with a secret, ensuring the token’s integrity.

[](https://media2.dev.to/dynamic/image/width=800%2Cheight=%2Cfit=scale-down%2Cgravity=auto%2Cformat=auto/https%3A%2F%2Fdev-to-uploads.s3.amazonaws.com%2Fuploads%2Farticles%2Fxceigflhs68dw1yzm26e.jpg)

**JWT in Action:**

* Upon user authentication, the server generates a JWT.
* This JWT is sent back to the client and stored, often in local storage or an HTTP-only cookie.
* The client includes this token in the HTTP Authorization header for subsequent requests.
* The server validates the token and grants access if valid.

**Advantages:**

* **Scalability:** Due to their stateless nature, JWTs are ideal for distributed systems.
* **Flexibility:** They can be used across different domains and applications.
* **Security:** When properly implemented, they provide a secure way to handle user authentication.

**Security Concerns:**

* **Transmission Security:** It's vital to transmit JWTs over HTTPS.
* **Storage:** Store JWTs securely to prevent XSS attacks and other vulnerabilities.

**Handling Token Expiry:**

* Implement short-lived JWTs and use refresh tokens for renewing access without re-authentication.

**Understanding Session-Based Authentication**

Session-based authentication, often referred to as cookie-based authentication, is a method where the server plays a pivotal role in maintaining user authentication records.

**How it works:**

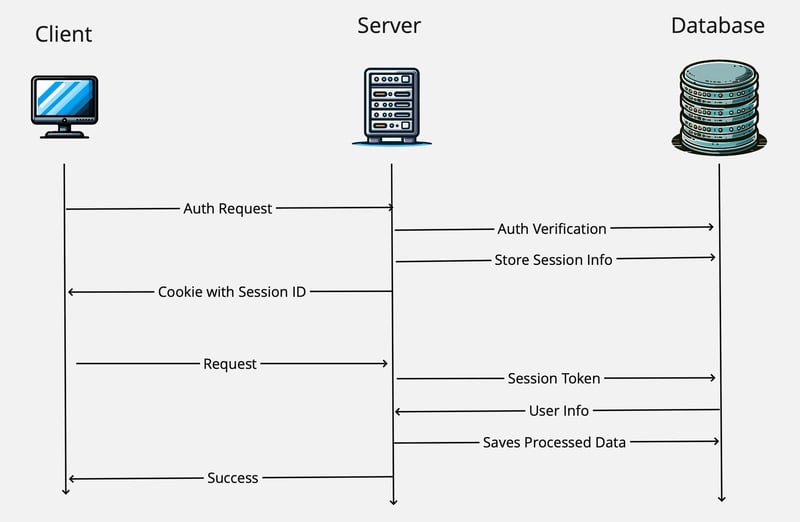
1. **User Authentication**: The user provides credentials, which the server verifies.
2. **Session Creation**: Upon successful authentication, the server creates a session record with a unique identifier, user identifier, session start time, expiry, and possibly additional context like IP address and User Agent. Stores that in Database.
3. **Cookie Storage**: This session identifier is sent back and stored as a cookie in the user’s browser.
4. **Session Validation**: Each request from the user’s browser includes this cookie, then server validates the session by querying to Database. If valid, the request is processed.

**Advantages:**

* **Simplicity and Reliability**: The server’s session record acts as a centralized truth source, making it straightforward to manage user sessions.
* **Revocation Efficiency**: Access can be quickly revoked by deleting or invalidating the session record, ensuring up-to-date session validity.

**Disadvantages:**

* **Performance Issues at Scale**: The dependency on database interactions for every session validation can introduce latency, particularly for high-traffic applications.
* **Latency in Dynamic Environments**: In applications with dynamic clients, this latency can impact user experience, making session-based authentication less ideal in such scenarios.

[](https://media2.dev.to/dynamic/image/width=800%2Cheight=%2Cfit=scale-down%2Cgravity=auto%2Cformat=auto/https%3A%2F%2Fdev-to-uploads.s3.amazonaws.com%2Fuploads%2Farticles%2F52i0zb81mjlfp95b27cw.jpg)

**Conclusion: Making the Right Authentication Choice**

Choosing between JWT and session-based authentication depends on your application's specific needs. If you prioritize statelessness and scalability, JWT might be your go-to. For traditional applications where immediate control over sessions is crucial, session-based authentication holds the upper hand. Understanding these concepts and their implications is key to developing secure and efficient web applications.

Q: explain "Prototype Design Pattern" with an example

**Prototype Design Pattern**

The Prototype design pattern is a creational design pattern that focuses on creating new objects by copying existing objects. This is particularly useful when object creation is expensive or complex. Instead of creating a new object from scratch, you clone an existing object and then modify it as needed.

**Key Components:**

1. **Prototype Interface:** An interface or abstract class that defines a common method for cloning objects, typically called clone().
2. **Concrete Prototypes:** Classes that implement the Prototype interface and provide their specific cloning logic.

interface Shape {

Shape clone();

void draw();

}

class Circle implements Shape {

private int radius;

public Circle(int radius) {

this.radius = radius;

}

@Override

public Circle clone() {

return new Circle(this.radius);

}

@Override

public void draw() {

System.out.println("Drawing a circle with radius: " + radius);

}

}

class Rectangle implements Shape {

private int width;

private int height;

public Rectangle(int width, int height) {

this.width = width;

this.height = height;

}

@Override

public Rectangle clone() {

return new Rectangle(this.width, this.height);

}

@Override

public void draw() {

System.out.println("Drawing a rectangle with width: " + width + " and height: " + height);

}

}