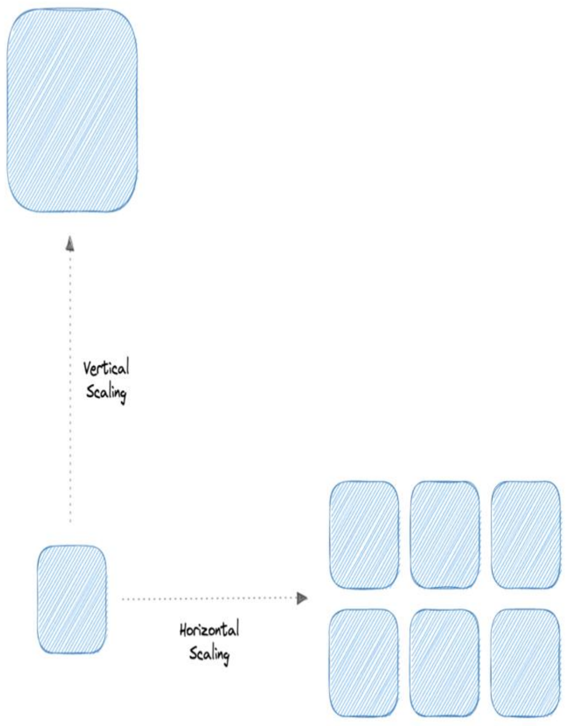
SCALING SYSTEMS AND SECURITY IN SYSTEM DESIGN

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



Let's discuss different types of scaling:

1. 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

2. 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

- Increased complexity
- Data inconsistency
- Increased load on downstream services

Storage

Storage is a mechanism that enables a system to retain data, either temporarily or permanently. This topic is mostly skipped over in the context of system design, however, it is important to have a basic understanding of some common types of storage techniques that can help us fine-tune our storage components. Let's discuss some important storage concepts:

RAID

RAID (Redundant Array of Independent Disks) is a way of storing the same data on multiple hard disks or solid-state drives (SSDs) to protect data in the case of a drive failure.

There are different RAID levels, however, and not all have the goal of providing redundancy. Let's discuss some commonly used RAID levels:

- **RAID 0**: Also known as striping, data is split evenly across all the drives in the array.
- **RAID 1**: Also known as mirroring, at least two drives contains the exact copy of a set of data. If a drive fails, others will still work.
- **RAID 5**: Striping with parity. Requires the use of at least 3 drives, striping the data across multiple drives like RAID 0, but also has a parity distributed across the drives.
- **RAID 6**: Striping with double parity. RAID 6 is like RAID 5, but the parity data are written to two drives.
- **RAID 10**: Combines striping plus mirroring from RAID 0 and RAID 1. It provides security by mirroring all data on secondary drives while using striping across each set of drives to speed up data transfers.

Comparison

Let's compare all the features of different RAID levels:

Features	RAID 0	RAID 1	RAID 5	RAID 6	RAID 10
Description	Striping	Mirroring	Striping with Parity	Striping with double parity	Striping and Mirroring
Minimum Disks	2	2	3	4	4
Read Performance	High	High	High	High	High
Write Performance	High	Medium	High	High	Medium
Cost	Low	High	Low	Low	High
Fault Tolerance	None	Single-drive failure	Single-drive failure	Two-drive failure	Up to one disk failure in each sub-array
Capacity Utilization	100%	50%	67%-94%	50%-80%	50%

Volumes

Volume is a fixed amount of storage on a disk or tape. The term volume is often used as a synonym for the storage itself, but it is possible for a single disk to contain more than one volume or a volume to span more than one disk.

File storage

File storage is a solution to store data as files and present it to its final users as a hierarchical directories structure. The main advantage is to provide a user-friendly solution to store and retrieve files. To locate a file in file storage, the complete path of the file is required. It is economical and easily structured and is usually found on hard drives, which means that they appear exactly the same for the user and on the hard drive.

Example: Amazon EFS, Azure files, Google Cloud Filestore, etc.

Block storage

Block storage divides data into blocks (chunks) and stores them as separate pieces. Each block of data is given a unique identifier, which allows a storage system to place the smaller pieces of data wherever it is most convenient.

Block storage also decouples data from user environments, allowing that data to be spread across multiple environments. This creates multiple paths to the data and allows the user to retrieve it quickly. When a user or application requests data from a block storage system, the underlying storage system reassembles the data blocks and presents the data to the user or application

Example: Amazon EBS.

Object Storage

Object storage, which is also known as object-based storage, breaks 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.

Example: Amazon S3, Azure Blob Storage, Google Cloud Storage, etc.

NAS

A NAS (Network Attached Storage) is a storage device connected to a network that allows storage and retrieval of data from a central location for authorized network users. NAS devices are flexible, meaning that as we need additional storage, we can add to what we have. It's faster, less expensive, and provides all the benefits of a public cloud on-site, giving us complete control.

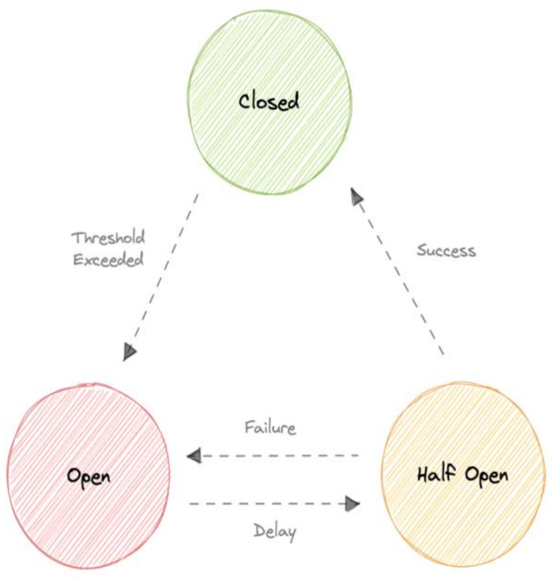
HDFS

The Hadoop Distributed File System (HDFS) is a distributed file system designed to run on commodity hardware. HDFS is highly fault-tolerant and is designed to be deployed on low-cost hardware. HDFS provides high throughput access to application data and is suitable for applications that have large data sets. It has many similarities with existing distributed file systems.

HDFS is designed to reliably store very large files across machines in a large cluster. It stores each file as a sequence of blocks, all blocks in a file except the last block are the same size. The blocks of a file are replicated for fault tolerance.

Circuit breaker

The circuit breaker is a design pattern used to detect failures and encapsulates the logic of preventing a failure from constantly recurring during maintenance, temporary external system failure, or unexpected system difficulties.



The basic idea behind the circuit breaker is very simple. We wrap a protected function call in a circuit breaker object, which monitors for failures. Once the failures reach a certain threshold, the circuit breaker trips, and all further calls to the circuit breaker return with an error, without the protected call being made at all. Usually, we'll also want some kind of monitor alert if the circuit breaker trips.

Why do we need circuit breaking?

It's common for software systems to make remote calls to software running in different processes, probably on different machines across a network. One of the big differences between in-memory calls and remote calls is that remote calls can fail, or hang without a response until some timeout limit is reached. What's worse is if we have many callers on an unresponsive supplier, then we can run out of critical resources leading to cascading failures across multiple systems.

States of circuit breaker:

Let's discuss circuit breaker states:

1. Closed

When everything is normal, the circuit breakers remain closed, and all the request passes through to the services as normal. If the number of failures increases beyond the threshold, the circuit breaker trips and goes into an open state.

2. Open

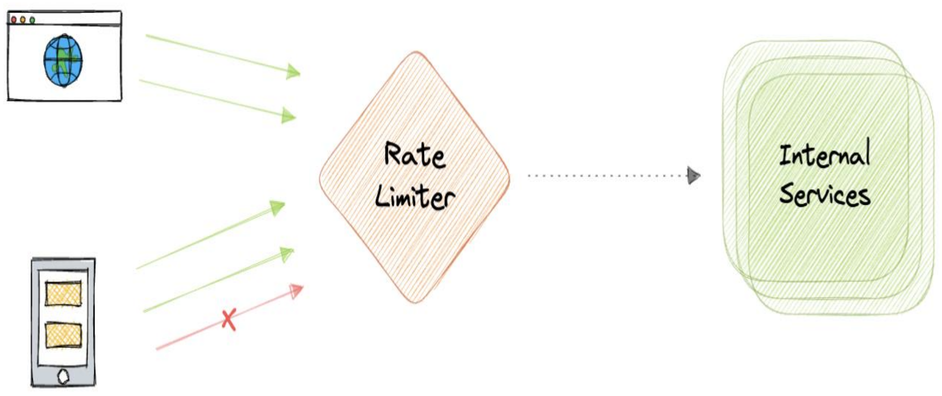
In this state circuit breaker returns an error immediately without even invoking the services. The Circuit breakers move into the half-open state after a certain timeout period elapses. Usually, it will have a monitoring system where the timeout will be specified.

3. Half-open

In this state, the circuit breaker allows a limited number of requests from the service to pass through and invoke the operation. If the requests are successful, then the circuit breaker will go to the closed state. However, if the requests continue to fail, then it goes back to the open state.

Rate Limiting

Rate limiting refers to preventing the frequency of an operation from exceeding a defined limit. In large-scale systems, rate limiting is commonly used to protect underlying services and resources. Rate limiting is generally used as a defensive mechanism in distributed systems, so that shared resources can maintain availability. It also protects our APIs from unintended or malicious overuse by limiting the number of requests that can reach our API in a given period of time.



Why do we need Rate Limiting?

Rate limiting is a very important part of any large-scale system and it can be used to accomplish the following:

- Avoid resource starvation as a result of Denial of Service (DoS) attacks.
- Rate Limiting helps in controlling operational costs by putting a virtual cap on the auto-scaling of resources which if not monitored might lead to exponential bills.
- Rate limiting can be used as defense or mitigation against some common attacks.
- For APIs that process massive amounts of data, rate limiting can be used to control the flow of that data.

Algorithms

There are various algorithms for API rate limiting, each with its advantages and disadvantages. Let's briefly discuss some of these algorithms:

Leaky Bucket

Leaky Bucket is an algorithm that provides a simple, intuitive approach to rate limiting via a queue. When registering a request, the system appends it to the end of the queue. Processing for the first item on the queue occurs at a regular interval or first-in, first-out (FIFO). If the queue is full, then additional requests are discarded (or leaked).

Token Bucket

Here we use a concept of a bucket. When a request comes in, a token from the bucket must be taken and processed. The request will be refused if no token is available in the bucket, and the requester will have to try again later. As a result, the token bucket gets refreshed after a certain time period.

Fixed Window

The system uses a window size of n seconds to track the fixed window algorithm rate. Each incoming request increments the counter for the window. It discards the request if the counter exceeds a threshold.

Sliding Log

Sliding Log rate-limiting involves tracking a time-stamped log for each request. The system stores these logs in a time-sorted hash set or table. It also discards logs with timestamps beyond a threshold. When a new request comes in, we calculate the sum of logs to determine the request rate. If the request would exceed the threshold rate, then it is held.

Sliding Window

Sliding Window is a hybrid approach that combines the fixed window algorithm's low processing cost and the sliding log's improved boundary conditions. Like the fixed window algorithm, we track a counter for each fixed window. Next, we account for a weighted value of the previous window's request rate based on the current timestamp to smooth out bursts of traffic.

Rate Limiting in Distributed Systems

Rate Limiting becomes complicated when distributed systems are involved. The two broad problems that come with rate limiting in distributed systems are:

Inconsistencies

When using a cluster of multiple nodes, we might need to enforce a global rate limit policy. Because if each node were to track its rate limit, a consumer could exceed a global rate limit when sending requests to different nodes. The greater the number of nodes, the more likely the user will exceed the global limit.

The simplest way to solve this problem is to use sticky sessions in our load balancers so that each consumer gets sent to exactly one node but this causes a lack of fault tolerance and scaling problems. Another approach might be to use a centralized data store like Redis but this will increase latency and cause race conditions.

Race Conditions

This issue happens when we use a naive "get-then-set" approach, in which we retrieve the current rate limit counter, increment it, and then push it back to the datastore. This model's problem is that additional requests can come through in the time it takes to perform a full cycle of read-increment-store, each attempting to store the increment counter with an invalid (lower) counter value. This allows a consumer to send a very large number of requests to bypass the rate limiting controls.

One way to avoid this problem is to use some sort of distributed locking mechanism around the key, preventing any other processes from accessing or writing to the counter. Though the lock will become a significant bottleneck and will not scale well. A better approach might be to use a "set-then-get" approach, allowing us to quickly increment and check counter values without letting the atomic operations get in the way.

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.

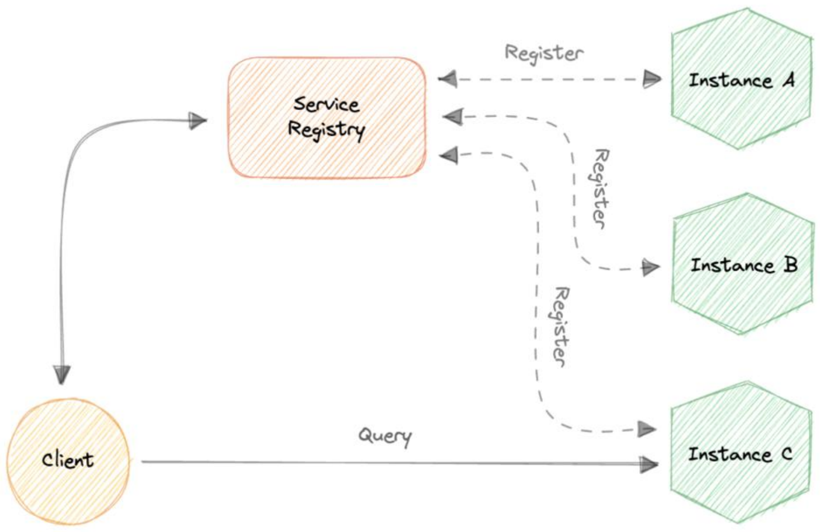
Why do we need Service Discovery?

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

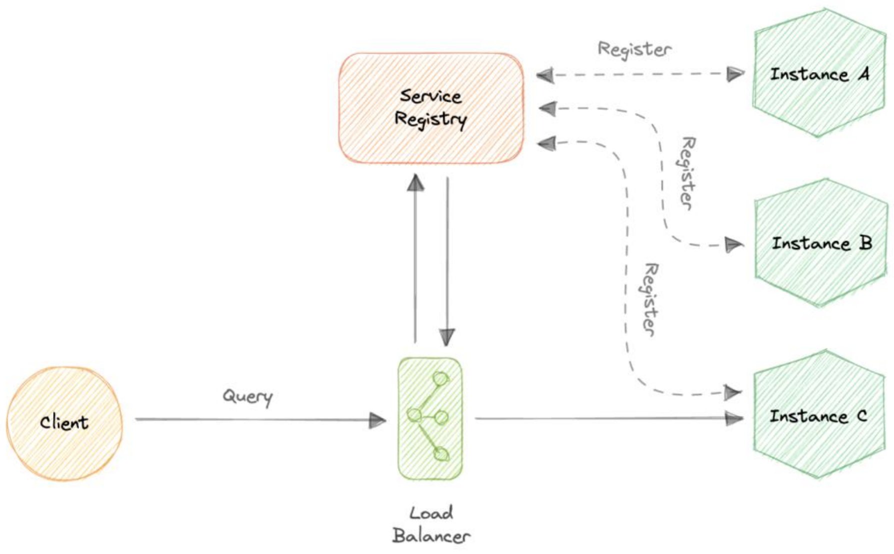
There are two main service discovery patterns:

1. Client-side discovery



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.

2. Server-side discovery



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 and envoy are some of the most commonly used service mesh technologies.

Examples

Here are some commonly used service discovery infrastructure tools:

- etcd
- Consul
- Apache Thrift
- Apache Zookeeper

SLA, SLO, SLI

Let's briefly discuss SLA, SLO, and SLI. These are mostly related to the business and site reliability side of things but good to know nonetheless.

Why are they important?

SLAs, SLOs, and SLIs allow companies to define, track and monitor the promises made for a service to its users. Together, SLAs, SLOs, and SLIs should help teams generate more user trust in their services with an added emphasis on continuous improvement to incident management and response processes.

1. SLA

- An SLA, or Service Level Agreement, is an agreement made between a company and its users of a given service. The SLA defines the different promises that the company makes to users regarding specific metrics, such as service availability.
- SLAs are often written by a company's business or legal team.

2. SLO

- An SLO, or Service Level Objective, is the promise that a company makes to users regarding a specific metric such as incident response or uptime. SLOs exist within an SLA as individual promises contained within the full user agreement. The SLO is the specific goal that the service must meet in order to comply with the SLA. SLOs should always be simple, clearly defined, and easily measured to determine whether or not the objective is being fulfilled.

3. SLI

- An SLI, or Service Level Indicator, is a key metric used to determine whether or not the SLO is being met. It is the measured value of the metric described within the SLO. In order to remain in compliance with the SLA, the SLI's value must always meet or exceed the value determined by the SLO.

Disaster recovery

Disaster recovery (DR) is a process of regaining access and functionality of the infrastructure after events like a natural disaster, cyber attack, or even business disruptions.

Disaster recovery relies upon the replication of data and computer processing in an off-premises location not affected by the disaster. When servers go down because of a disaster, a business needs to recover lost data from a second location where the data is backed up. Ideally, an organization can transfer its computer processing to that remote location as well in order to continue operations.

Disaster Recovery is often not actively discussed during system design interviews but it's important to have some basic understanding of this topic. You can learn more about disaster recovery from AWS Well-Architected Framework.

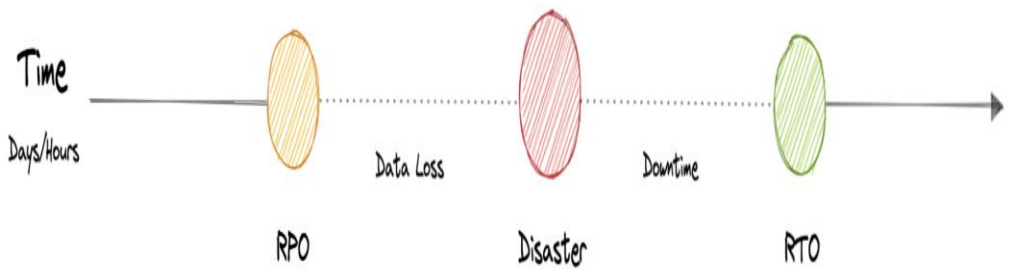
Why is disaster recovery important?

Disaster recovery can have the following benefits:

- Minimize interruption and downtime
- Limit damages
- Fast restoration
- Better customer retention

Terms

Let's discuss some important terms relevantly for disaster recovery:



1. RTO

Recovery Time Objective (RTO) is the maximum acceptable delay between the interruption of service and restoration of service. This determines what is considered an acceptable time window when service is unavailable.

2. RPO

Recovery Point Objective (RPO) is the maximum acceptable amount of time since the last data recovery point. This determines what is considered an acceptable loss of data between the last recovery point and the interruption of service.

3. Strategies

A variety of disaster recovery (DR) strategies can be part of a disaster recovery plan.

4. Back-up

This is the simplest type of disaster recovery and involves storing data off-site or on a removable drive.

5. Cold Site

In this type of disaster recovery, an organization sets up basic infrastructure in a second site.

6. Hot site

A hot site maintains up-to-date copies of data at all times. Hot sites are time-consuming to set up and more expensive than cold sites, but they dramatically reduce downtime.

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:

1. 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.

2. Workload portability

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

3. 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.

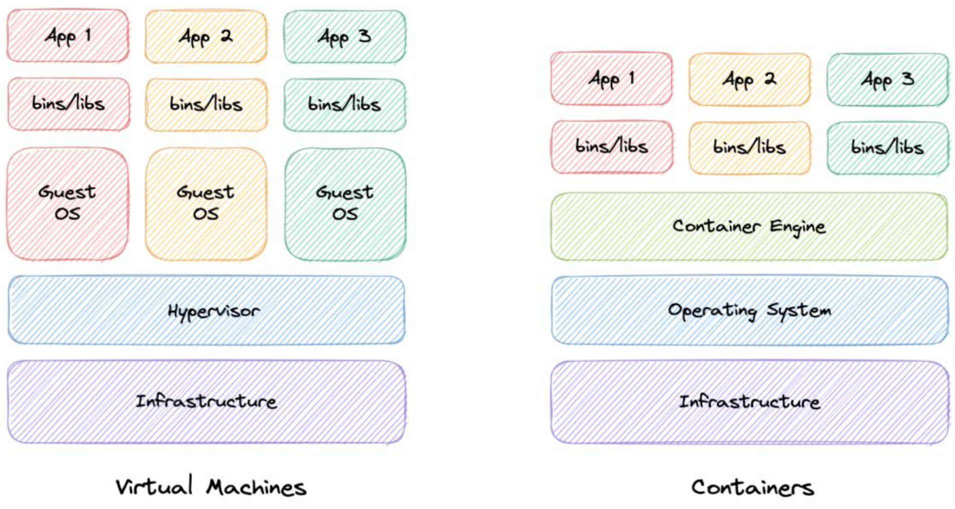
4. Agile development

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

5. Efficient operations

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

Virtualization vs. Containerization



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.

OAuth 2.0 and OpenID Connect (OIDC)

1. OAuth 2.0

OAuth 2.0, which stands for Open Authorization, is a standard designed to provide consented access to resources on behalf of the user, without ever sharing the user's credentials. OAuth 2.0 is an authorization protocol and not an authentication protocol, it is designed primarily as a means of granting access to a set of resources, for example, remote APIs or user's data.

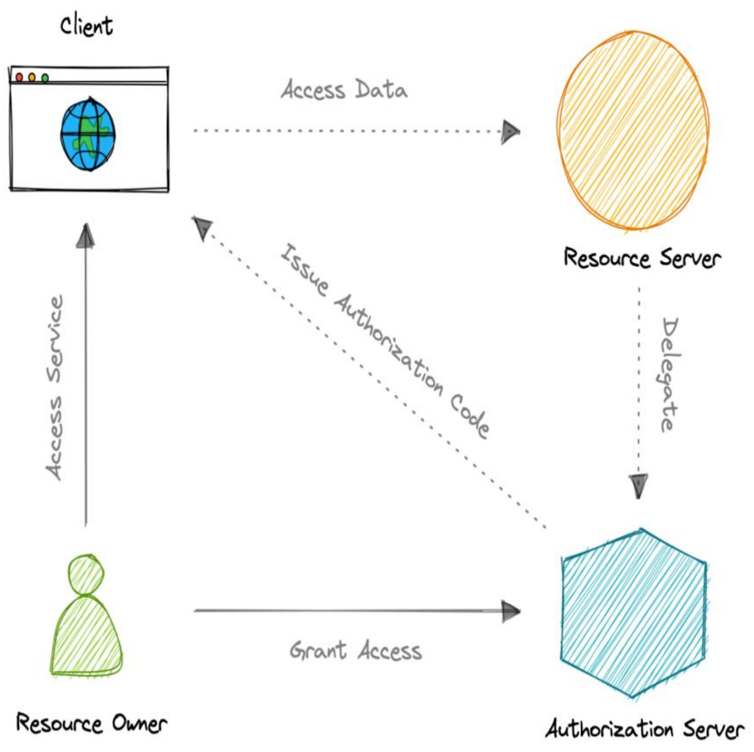
Concepts

The OAuth 2.0 protocol defines the following entities:

- **Resource Owner**: The user or system that owns the protected resources and can grant access to them.
- **Client**: The client is the system that requires access to the protected resources.
- **Authorization Server**: This server receives requests from the Client for Access Tokens and issues them upon successful authentication and consent by the Resource Owner.
- **Resource Server**: A server that protects the user's resources and receives access requests from the Client. It accepts and validates an Access Token from the Client and returns the appropriate resources.
- **Scopes**: They are used to specify exactly the reason for which access to resources may be granted. Acceptable scope values, and which resources they relate to, are dependent on the Resource Server.
- **Access Token**: A piece of data that represents the authorization to access resources on behalf of the end-user.

How does OAuth 2.0 work?

Let's learn how OAuth 2.0 works:



1. The client requests authorization from the Authorization Server, supplying the client id and secret as identification. It also provides the scopes and an endpoint URI to send the Access Token or the Authorization Code.
2. The Authorization Server authenticates the client and verifies that the requested scopes are permitted.
3. The resource owner interacts with the authorization server to grant access.
4. The Authorization Server redirects back to the client with either an Authorization Code or Access Token, depending on the grant type. A Refresh Token may also be returned.
5. With the Access Token, the client can request access to the resource from the Resource Server.

Disadvantages

Here are the most common disadvantages of OAuth 2.0:

- Lacks built-in security features.
- No standard implementation.
- No common set of scopes.

2. OpenID Connect

OAuth 2.0 is designed only for authorization, for granting access to data and features from one application to another. OpenID Connect (OIDC) is a thin layer that sits on top of OAuth 2.0 that adds login and profile information about the person who is logged in.

When an Authorization Server supports OIDC, it is sometimes called an Identity Provider (IdP), since it provides information about the Resource Owner back to the Client. OpenID Connect is relatively new, resulting in lower adoption and industry implementation of best practices compared to OAuth.

Concepts

The OpenID Connect (OIDC) protocol defines the following entities:

- **Relying Party**: The current application.
- **OpenID Provider**: This is essentially an intermediate service that provides a one-time code to the Relying Party.
- **Token Endpoint**: A web server that accepts the One-Time Code (OTC) and provides an access code that's valid for an hour. The main difference between OIDC and OAuth 2.0 is that the token is provided using JSON Web Token (JWT).
- **UserInfo Endpoint**: The Relying Party communicates with this endpoint, providing a secure token and receiving information about the end-user

Both OAuth 2.0 and OIDC are easy to implement and are JSON based, which is supported by most web and mobile applications. However, the OpenID Connect (OIDC) specification is more strict than that of basic OAuth.

Single Sign-On (SSO)

Single Sign-On (SSO) is an authentication process in which a user is provided access to multiple applications or websites by using only a single set of login credentials. This prevents the need for the user to log separately into the different applications.

The user credentials and other identifying information are stored and managed by a centralized system called Identity Provider (IdP). The Identity Provider is a trusted system that provides access to other websites and applications.

Single Sign-On (SSO) based authentication systems are commonly used in enterprise environments where employees require access to multiple applications of their organizations.

Components

Let's discuss some key components of Single Sign-On (SSO).

1. Identity Provider (IdP)

1. User Identity information is stored and managed by a centralized system called Identity Provider (IdP). The Identity Provider authenticates the user and provides access to the service provider.
2. The identity provider can directly authenticate the user by validating a username and password or by validating an assertion about the user's identity as presented by a separate identity provider. The identity provider handles the management of user identities in order to free the service provider from this responsibility.

2. Service Provider

- A service provider provides services to the end-user. They rely on identity providers to assert the identity of a user, and typically certain attributes about the user are managed by the identity provider. Service providers may also maintain a local account for the user along with attributes that are unique to their service.

3. Identity Broker

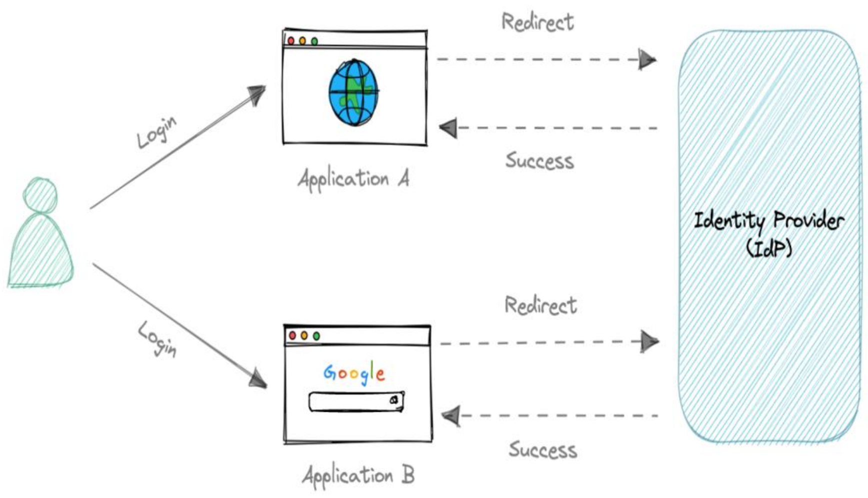
- An identity broker acts as an intermediary that connects multiple service providers with various different identity providers. Using Identity Broker, we can perform single sign-on over any application without the hassle of the protocol it follows.

4. SAML

- Security Assertion Markup Language is an open standard that allows clients to share security information about identity, authentication, and permission across different systems. SAML is implemented with the Extensible Markup Language (XML) standard for sharing data.
- SAML specifically enables identity federation, making it possible for identity providers (IdPs) to seamlessly and securely pass authenticated identities and their attributes to service providers.

How does SSO work?

Now, let's discuss how Single Sign-On works:



1. The user requests a resource from their desired application.
2. The application redirects the user to the Identity Provider (IdP) for authentication.
3. The user signs in with their credentials (usually, username and password).
4. Identity Provider (IdP) sends a Single Sign-On response back to the client application.
5. The application grants access to the user.

SAML vs OAuth 2.0 and OpenID Connect (OIDC)

There are many differences between SAML, OAuth, and OIDC. SAML uses XML to pass messages, while OAuth and OIDC use JSON. OAuth provides a simpler experience, while SAML is geared towards enterprise security.

OAuth and OIDC use RESTful communication extensively, which is why mobile, and modern web applications find OAuth and OIDC a better experience for the user. SAML, on the other hand, drops a session cookie in a browser that allows a user to access certain web pages. This is great for short-lived workloads.

OIDC is developer-friendly and simpler to implement, which broadens the use cases for which it might be implemented. It can be implemented from scratch pretty fast, via freely available libraries in all common programming languages. SAML can be complex to install and maintain, which only enterprise-size companies can handle well.

OpenID Connect is essentially a layer on top of the OAuth framework. Therefore, it can offer a built-in layer of permission that asks a user to agree to what the service provider might access. Although SAML is also capable of allowing consent flow, it achieves this by hard-coding carried out by a developer and not as part of its protocol.

Both of these authentication protocols are good at what they do. As always, a lot depends on our specific use cases and target audience.

Advantages

Following are the benefits of using Single Sign-On:

- Ease of use as users only need to remember one set of credentials.
- Ease of access without having to go through a lengthy authorization process.
- Enforced security and compliance to protect sensitive data.
- Simplifying the management with reduced IT support cost and admin time.

Disadvantages

Here are some disadvantages of Single Sign-On:

- Single Password Vulnerability, if the main SSO password gets compromised, all the supported applications get compromised.
- The authentication process using Single Sign-On is slower than traditional authentication as every application has to request the SSO provider for verification.

Examples

These are some commonly used Identity Providers (IdP):

- Okta
- Google
- Auth0
- OneLogin

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.

1. 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.

2. 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.

3. 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 to verify each other.