System Design

I'm going to kick ass in system design. All this studying will give me superpowers.

Terms

Scalability is the capability of a system, process, or a network to grow and manage increased demand.

Reliability is the probability a system will fail in a given period. In simple terms, a distributed system is considered **reliable** if it keeps delivering its services even when one or several of its software or hardware components fail.

Availability is the time a system remains operational to perform its required function in a specific period. It is a simple measure of the percentage of time that a system, service, or a machine remains operational under normal conditions.

If a system is **reliable**, it is **available**. However, if it is **available**, it is not necessarily **reliable**. In other words, high reliability contributes to high availability, but it is possible to achieve a high availability even with an unreliable product by minimizing repair time and ensuring that spares are always available when they are needed.

Blob: Binary Large Object. Blog storage. Example would be storing pictures as a blog.

(Instagram) The system should be highly **reliable**; any uploaded photo or video should never be lost.

S.O.L.I.D:

- **S - Single-Responsibility Principle**: A class or module should only have a single reason to change.
- **O - Open-Closed Principle**: In its extremist form, you should never have to rewrite any of your code. You should only write new code (add new code to your code base). If we were to do that, it would be easier to avoid bugs in our code. (If we have good dependency injection, we can always delete an old module, replace it with a new one).
- **L - Liskov Substitution Principle**: If you have a base-class and a subclass, the subclass should be substitutable for the base class at any point in the program. In other words, whenever you are using the base type, you should be able to substitute the derived type without causing any unwanted behavior in program. This mean the derived class should handle more inputs (Pre-Conditions), but should output (Post-Condition) the set or a subset from the base class has.

- **I - Interface Segregation Principle**: It's better to have many small interfaces than few large ones. Example of an animal class when designing a game. We could create IAnimal { Eat, Sleep, Move } but it's much better to actually create IEat, ISleep, IMove interfaces. What if later on, the game gets an animal that can't sleep.
- **D** - Dependency Inversion Principle: This has a lot of do with dependency injection/constructor injection. In my own term, your class should not use the new keyword to create another class. That can be injected (via constructor or method). Again this seems like an art-form to me. We can go crazy with injections.

- High-level modules should not depend on low-level modules. Both should depend on abstractions.
- Abstractions should not depend on details. Details should depend on abstractions.

A system that is difficult to test is an indicator that it is poorly architected. It is highly coupled.

How to avoid single point of failure, throw more money at the problem. Only one database, have another one that replicates in real-time. Only one load balance? Get another one and have your DNS point to them (facebook.com would map to IP Addresses of two load balancers). Only one located in one region? Host in two regions. Netflix has a great way to test single point of failure (called chaos monkey). Randomly go in production and take a service down.

Monolith (Pomo) vs Microservices (Kronos)

Monolith: One machine/multiple running a huge system. You can horizontally scale these systems	
Advantages	**Disadvantages**
Good for small teams that doesn't have a lot of time. It takes time to create these micro-services.	New members in team must have a lot of context on what they are developing. Too much code to go

	through.
Lesser moving parts. Don't have to worry about multiple services. Deployment is simple.	Code is deployed very frequently. Because any change means deploying the whole thing.
Less code duplication (Don't have different solutions for multiple little services).	Too much responsibility on each server. If server crashes, we can have system-wide failure.
Faster - you're not making lots of calls across network. These are making method calls here.	Parallel development is hard. Merging changes will be challenging. Tight coupling in developer time.
	Hard to say what part is being used more and what part is being used less.

Microservices: Tiny Services. Single business unit.	
Advantages	**Disadvantages**
Partial failures - the likelihood of whole system crashing is low.	Need a smarter architect. Lot more complexity on how they communicate.
Super easy to scale.	Need more resources. Each instance might need their own container. Need more memory.
Easy for new members to come and be productive. You can assign them a task related to just one service. They don't have to know the whole codebase.	Increases network communication. We have to use some kind of messaging system.
Parallel development is easy.	Due to lots of moving parts, these applications are more prone to

	security vulnerabilities.
Lesser parts that are hidden. If you see a huge load on one server, you can easily scale that out.	Testing the whole system might become a little difficult. Each microsoft is running on different environments.

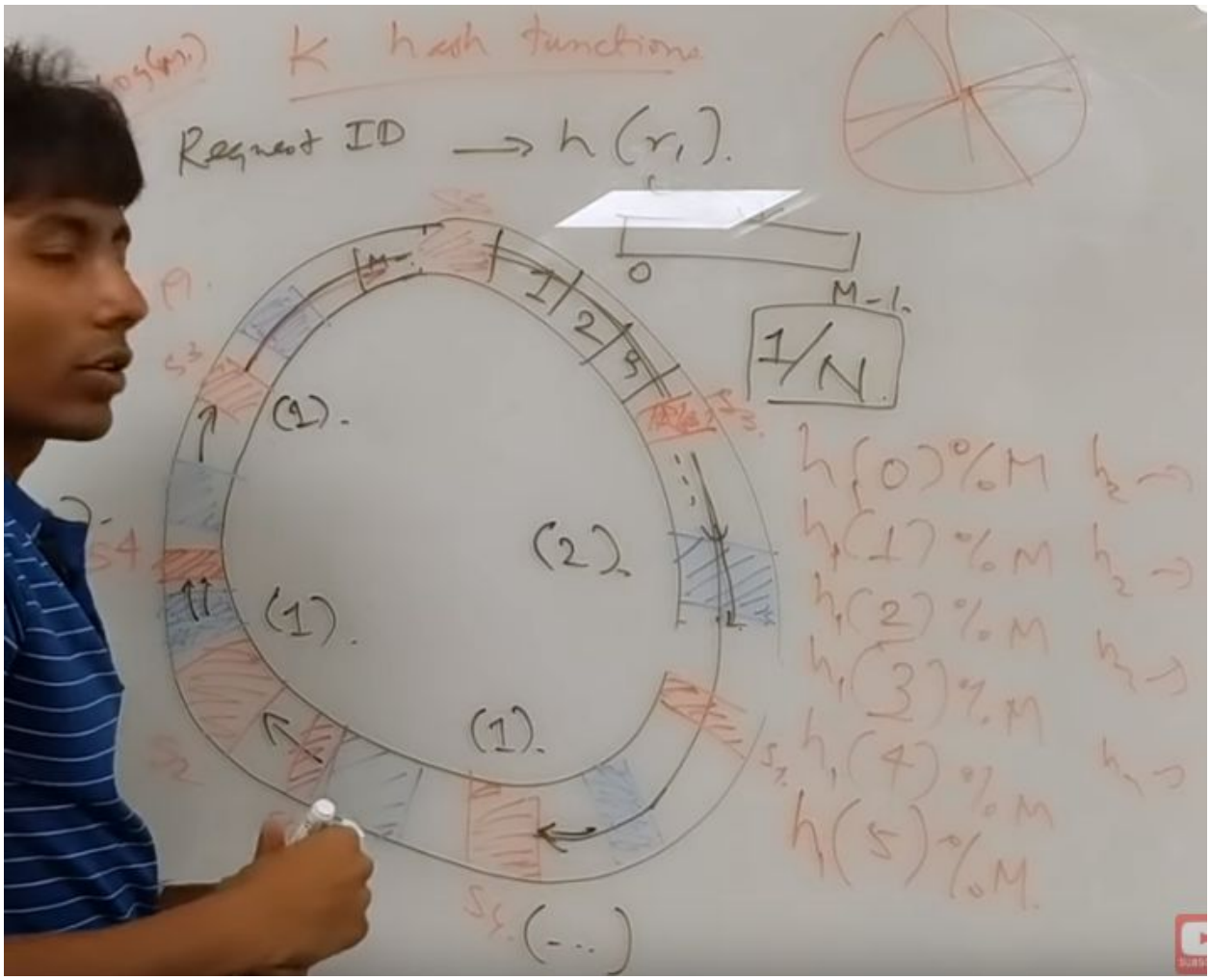
Horizontal vs Vertical Scaling

Horizontal	**Vertical**
Load balancing required: ones you have two more how servers, how do we route people?	If single machine, no load to balance.
Resilient. When one server goes down, route traffic to another.	Single point of failure
Talk to each other by making network calls. Lots of network traffic.	Inter process communication. Make function calls.
Data Inconsistency. If you have a transaction, we might have to lock down all service databases. Data is complicated maintain.	Consistent
Scales well, matter of adding servers	Hardware limit

Vertical: Add more memory, CPU ect for existing host. Expensive. Limitations on how much hardware. Does not have distributed systems problems (if one machine)
Horizontal: Keep adding hosts. Infinite number of hosts. Distributed system challenges.

Consistent Hashing

Load balances are using consistent hashing behind the scene. In the image, load balancers are in orange, clients are in red. We have many hashing functions that maps to many different areas of the circle. When a new client requests, we can route them to the next load balancer. If the hashing algorithm is great, clients will be evenly distributed to load balancer. If one machine fails (orange spot in circle goes away), all it's clients can be routed to the next orange spot.



Sharding:

Sharding is a type of database partitioning that separates very large databases the into smaller, faster, more easily managed parts called data shards. The word shard means a small part of a whole.

Horizontal partitioning: In this scheme, we put different rows into different tables. Example would be using zip code to shard data in different tables. Disadvantage here is that if we might end up with very unbalanced tables.

Vertical Partitioning: Example would be when you're building a social app, storing user data in one database, photos they upload in a different database and people they follow in a different database. It is almost like a separate database for each of the features. Vertical partitioning is straightforward to implement and has a low impact on the application. The main problem with this approach is that if our application experiences additional growth, then it may be necessary to further partition a feature specific DB across various servers.

Directory Based Partitioning: A loosely coupled approach to work around issues mentioned in the above schemes is to create a lookup service which knows your current partitioning scheme and abstracts it away from the DB access code. So, to find out where a particular data entity resides, we query the directory server that holds the mapping between each tuple key to its DB server. This loosely coupled approach means we can perform tasks like adding servers to the DB pool or changing our partitioning scheme without having an impact on the application.

Load Balancing

Load balancing is the process of spreading requests across multiple resources according to some metric (random, round-robin, random with weighting for machine capacity, etc) and their current status (available for requests, not responding, elevated error rate, etc).

Load needs to be balanced between user requests and your web servers, but must also be balanced at every stage to achieve full scalability and redundancy for your system. A moderately large system may balance load at three layers:

- user to your web servers,
- web servers to an internal platform layer,
- internal platform layer to your database.

There are a number of ways to implement load balancing.

Smart Clients: developer writes some shitty connection logic to servers. This is a horrible idea.
Hardware Load Balancers: expensive as fuck, typically used as first point of contact from internet to your server.
Software Load Balancers: self explanatory.

There is a variety of load balancing methods, which use different algorithms for different needs.

- **Least Connection Method** — This method directs traffic to the server with the fewest active connections. This approach is quite useful when there are a large number of persistent client connections which are unevenly distributed between the servers.
- **Least Response Time Method** — This algorithm directs traffic to the server with the fewest active connections and the lowest average response time.
- **Least Bandwidth Method** - This method selects the server that is currently serving the least amount of traffic measured in megabits per second (Mbps).
- **Round Robin Method** — This method cycles through a list of servers and sends each new request to the next server. When it reaches the end of the list, it starts over at the beginning. It is most useful when the servers are of equal specification and there are not many persistent connections.
- **Weighted Round Robin Method** — The weighted round-robin scheduling is designed to better handle servers with different processing capacities.

Each server is assigned a weight (an integer value that indicates the processing capacity). Servers with higher weights receive new connections before those with less weights and servers with higher weights get more connections than those with less weights.

- **IP Hash** — Under this method, a hash of the IP address of the client is calculated to redirect the request to a server.

L4 vs L7:

Layer 7: Application Load Balancers

Application load balancing is the distribution of requests based on multiple variables, from the network layer to the application layer. It is context-aware and can direct requests based on any single variable as easily as it can a combination of variables. Applications are load balanced based on their peculiar behavior and not solely on server (operating system or virtualization layer) information.

Layer 4: Network Load Balancers.

Network load balancing is the distribution of traffic based on network variables, such as IP address and destination ports. It is layer 4 (TCP) and below and is not designed to take into consideration anything at the application layer such as content type, cookie data, custom headers, user location, or the application behavior. It is context-less, caring only about the network-layer information contained within the packets it is directing this way and that.

The difference between the two is important because network load balancing cannot assure availability of the application. This is because it bases its decisions solely on network and TCP-layer variables and has no awareness of the application at all. Generally a network load balancer will determine “availability” based on the ability of a server to respond to ICMP ping, or to correctly complete the three-way TCP handshake. An application load balancer goes much deeper, and is capable of determining availability based on not only a successful HTTP GET of a particular page but also the verification that the content is as was expected based on the input parameters.

File vs. Block vs. Object Storage

File: File-based storage is the old school approach to storage. And like most older things, it's about as simple as it gets. You give files a name, tag them with metadata, then organize them in folders under directories and sub-directories. The standard naming convention makes them easy enough to organize while storage technologies such as **NAS** allow for convenient sharing at the local level. Many companies demand a centralized, easily accessible way to store files and folders. File level storage can deliver these perks at a cost that is typically affordable on a small business budget.

With file level storage, you have a hierarchical system that excels at handling relatively small amounts of data. However, any IT admin with experience managing this sort of architecture can probably attest to its shortcomings. Sure, you can technically create and store an unlimited number of files. But simply finding those files is a real chore over time. The more files you accumulate, the bigger the headache. Flipping through dozens of folders. Scrolling through hundreds of files all to find that one you need. It's not fun.

Block: Block storage services are relatively simple and familiar. They provide a traditional block storage device — like a hard drive — over the network. Cloud providers often have products that can provision a block storage device of any size and attach it to your virtual machine.

From there, you would treat it like a normal disk. You could format it with a filesystem and store files on it, combine multiple devices into a RAID array, or configure a database to write directly to the block device, avoiding filesystem overhead entirely. Additionally, network-attached block storage devices often have some unique advantages over normal hard drives:

You can easily take live snapshots of the entire device for backup purposes
Block storage devices can be resized to accommodate growing needs
You can easily detach and move block storage devices between machines
This is a very flexible setup that can be useful for most any kind of application.
Let's summarize some advantages and disadvantages of the technology.

Some advantages of block storage are:

Block storage is a familiar paradigm. People and software understand and support files and file systems almost universally.

Block devices are well supported. Every programming language can easily read and write files.

Filesystem permissions and access controls are familiar and well-understood Block storage devices provide low latency IO, so they are suitable for use by databases.

The disadvantages of block storage are:

Storage is tied to one server at a time.

Blocks and filesystems have limited metadata about the blobs of information they're storing (creation date, owner, size). Any additional information about what you're storing will have to be handled at the application and database level, which is additional complexity for a developer to worry about.

You need to pay for all the block storage space you've allocated, even if you're not using it.

You can only access block storage through a running server.

Block storage needs more hands-on work and setup vs object storage (filesystem choices, permissions, versioning, backups, etc.)

Block Storage Use Cases

Databases: Block storage is common in databases and other mission-critical applications that demand consistently high performance.

Email servers: Block storage is the defacto standard for Microsoft’s popular email server Exchange, which doesn’t support file or network-based storage systems.

RAID: Block storage can create an ideal foundation for RAID arrays designed to bolster data protection and performance by combining multiple disks as independent volumes.

Virtual machines: Virtualization software vendors such as VMware use block storage as file systems for the guest operating systems packaged inside virtual machine disk images?

Object Store: Object storage is the storage and retrieval of unstructured blobs of data and metadata using an HTTP API. Instead of breaking files down into blocks to store it on disk using a filesystem, we deal with whole objects stored over the network. These objects could be an image file, logs, HTML files, or any self-contained blob of bytes. They are unstructured because there is no specific schema or format they need to follow.

Object storage took off because it greatly simplified the developer experience. Because the API consists of standard HTTP requests, libraries were quickly developed for most programming languages. Saving a blob of data became as easy as an HTTP PUT request to the object store. Retrieving the file and metadata is a normal GET request. Further, most object storage services can also serve the files publicly to your users, removing the need to maintain a web server to host static assets.

On top of that, object storage services charge only for the storage space you use (some also charge per HTTP request, and for transfer bandwidth). This is a boon for small developers, who can get world-class storage and hosting of assets at costs that scale with use.

Some advantages of object storage are:

A simple HTTP API, with clients available for all major operating systems and programming languages.

A cost structure that means you only pay for what you use.

Built-in public serving of static assets means one less server for you to run yourself

Some object stores offer built-in CDN integration, which cache your assets around the globe to make downloads and page loads faster for your users.

Optional versioning means you can retrieve old versions of objects to recover from accidental overwrites of data.

Object storage services can easily scale from modest needs to really intense use-cases without the developer having to launch more resources or rearchitect to handle the load.

Using an object storage service means you don't have to maintain hard drives and RAID arrays, as that's handled by the service provider.

Being able to store chunks of metadata alongside your data blob can further simplify your application architecture.

Some disadvantages of object storage are:

You can't use object storage services to back a traditional database, due to the high latency of such services.

Object storage doesn't allow you to alter just a piece of a data blob, you must read and write an entire object at once. This has some performance implications. For instance, on a filesystem, you can easily append a single line to the end of a log file. On an object storage system, you'd need to retrieve the object, add the new line, and write the entire object back. This makes object storage less ideal for data that changes very frequently.

Operating systems can't easily mount an object store like a normal disk. There are some clients and adapters to help with this, but in general, using and

browsing an object store is not as simple as flipping through directories in a file browser.

Because of these properties, object storage is useful for hosting static assets, saving user-generated content such as images and movies, storing backup files, and storing logs, for example.

Object Storage Use Cases

Big data: Object storage has the ability to accommodate unstructured data with relative ease. This makes it a perfect fit for the big data needs of organizations in finance, healthcare, and beyond.

Web apps: You can normally access object storage through an API. This is why it's naturally suited for API-driven web applications with high-volume storage needs.

Backup archives: Object storage has native support for large data sets and near infinite scaling capabilities. This is why it is primed for the massive amounts of data that typically accompany archived backups.

Caching

Caching consists of: precalculating results, pre-generating expensive indexes, and storing copies of frequently accessed data in a faster backend (e.g. Memcache instead of PostgreSQL.

Memcached and Redis are both examples of in-memory caches (caveat: Redis can be configured to store some data to disk).

Caches take advantage of the locality of reference principle: recently requested data is likely to be requested again. They are used in almost every layer of computing: hardware, operating systems, web browsers, web applications, and more. A cache is like short-term memory: it has a limited amount of space, but is typically faster than the original data source and contains the most recently accessed items. Caches can exist at all levels in architecture, but are often found

at the level nearest to the front end where they are implemented to return data quickly without taxing downstream levels.

Cache Invalidation Strategies:

LRU: Least Frequently Used
TTL: Time to live

Cache Writing Strategies:

Write-through cache: Under this scheme, data is written into the cache and the corresponding database at the same time. The cached data allows for fast retrieval and, since the same data gets written in the permanent storage, we will have complete data consistency between the cache and the storage. Also, this scheme ensures that nothing will get lost in case of a crash, power failure, or other system disruptions.

Although, write through minimizes the risk of data loss, since every write operation must be done twice before returning success to the client, this scheme has the disadvantage of higher latency for write operations.

Write-around cache: This technique is similar to write through cache, but data is written directly to permanent storage, bypassing the cache. This can reduce the cache being flooded with write operations that will not subsequently be re-read, but has the disadvantage that a read request for recently written data will create a “cache miss” and must be read from slower back-end storage and experience higher latency.

Write-back cache: Under this scheme, data is written to cache alone and completion is immediately confirmed to the client. The write to the permanent storage is done after specified intervals or under certain conditions. This results in low latency and high throughput for write-intensive applications, however, this speed comes with the risk of data loss in case of a crash or other adverse event because the only copy of the written data is in the cache.

Content Distribution Networks & Edge

CDN

CDNs take the burden of serving static media off of your application servers (which are typically optimized for serving dynamic pages rather than static media), and provide geographic distribution. Overall, your static assets will load more quickly and with less strain on your servers (but a new strain of business expense).

In a typical CDN setup, a request will first ask your CDN for a piece of static media, the CDN will serve that content if it has it locally available (HTTP headers are used for configuring how the CDN caches a given piece of content). If it isn't available, the CDN will query your servers for the file and then cache it locally and serve it to the requesting user (in this configuration they are acting as a read-through cache).

CDN and authentication:

Token authentication is a mechanism that allows you to prevent the Azure Content Delivery Network (CDN) from serving assets to unauthorized clients.

Token authentication verifies that requests are generated by a trusted site by requiring requests to contain a token value that holds encoded information about the requester. Content is served to a requester only if the encoded information meets the requirements; otherwise, requests are denied. You can set up the requirements by using one or more of the following parameters:

- Country: Allow or deny requests that originate from the countries/regions specified by their country code.
- URL: Allow only requests that match the specified asset or path.
- Host: Allow or deny requests that use the specified hosts in the request header.
- Referrer: Allow or deny request from the specified referrer.
- IP address: Allow only requests that originated from specific IP address or IP subnet.
- Protocol: Allow or deny requests based on the protocol used to request the content.
- Expiration time: Assign a date and time period to ensure that a link remains valid only for a limited time

Edge
sdf

Message Queues

For processing you'd like to perform inline with a request but is too slow, the easiest solution is to create a message queue.

Message queues have another benefit, which is that they allow you to create a separate machine pool for performing off-line processing rather than burdening your web application servers. This allows you to target increases in resources to your current performance or throughput bottleneck rather than uniformly increasing resources across the bottleneck and non-bottleneck systems.

Map-Reduce

Adding a map-reduce layer makes it possible to perform data and/or processing intensive operations in a reasonable amount of time. You might use it for calculating suggested users in a social graph, or for generating analytics reports.

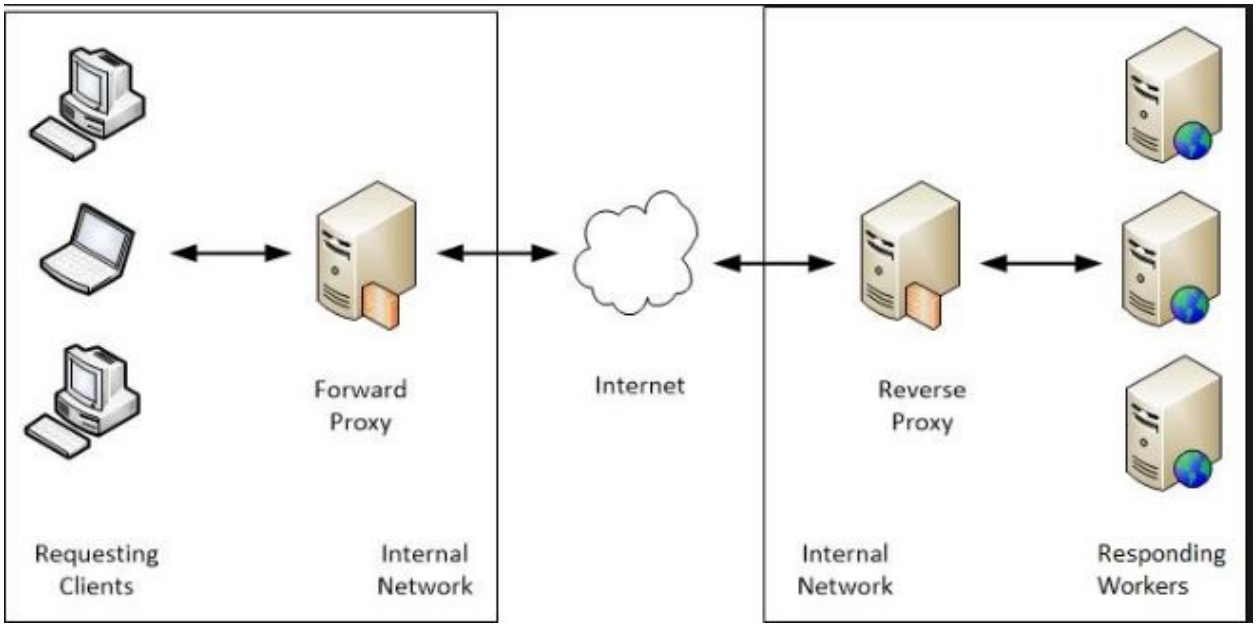
For sufficiently small systems you can often get away with adhoc queries on a SQL database, but that approach may not scale up trivially once the quantity of data stored or write-load requires sharding your database, and will usually require dedicated slaves for the purpose of performing these queries (at which point, maybe you'd rather use a system designed for analyzing large quantities of data, rather than fighting your database)

Proxy Server

A proxy server is an intermediate server between the client and the back-end server.

A reverse proxy accepts a request from a client, forwards it to a server that can fulfill it, and returns the server's response to the client.

Picture of Proxy and Reverse Proxy. Proxies are usually installed in companies to prevent users from using unauthorized websites such as facebook.com. Reverse proxies can be used to hide your web servers or even return cached data. Because clients see only the reverse proxy's IP address, you are free to change the configuration of your backend infrastructure, making reverse proxies a great tool in scalability.



NOSQL

NOSQL is a hot topic but not everyone uses NOSQL (unless for analytics):

- YouTube does not use NOSQL
- StackOverflow does not use NOSQL
- Instagram does not use NOSQL

Imagine you get a person object. In SQL, you might break it down to an address table and a person table. In NOSQL, you save the whole object (person JSON) into the database.

- This means NOSQL does not have to check for referential integrity.
- In SQL, you have to create an address or make sure an address object exists. That is extra overhead.

In NOSQL, relevant data is stored in a block. In the case of the person object, we save the whole object. So NOSQL is more flexible. NOSQL only cares about the JSON object, so if we have a new field, we don't care. Whereas in SQL, you will have to change the schema (make a new column)

In SQL, we need locks to maintain consistency. NOSQL does not need those locks.

Advantages of NOSQL:

- Insertions and retrievals requires the whole blob (json object). This means insertions are lightning fast. "SELECT *"s are lighting fast.
- Flexible Schema - new/deleting columns have no effect. We only care about the blog (json document).
- Horizontal partitioning - sharding
- Aggregations are easy.

Disadvantages of NOSQL:

- Update = DELETE + INSERT. We can't update a single column. Have to delete and reinsert the whole blog. We might get 2 separate updates for the same id (update 1 changes property a, update 2 changes property b). Will update 1 change be lost because update 2 does not have that change? SQL does not have this problem. If an update changes a column, next one another column, both changes are there. Thank you ACID. ACID is not guaranteed in NOSQL.
- Not read optimized: can't read a single JSON element. In SQL we can read a single column.
- No implicit information about relations. Referential integrity is missing in NOSQL.
- JOINS are hard to do.

The BASE acronym is used to describe the properties of certain databases, usually NoSQL databases. It's often referred to as the opposite of ACID.

- **Basically Available**: indicates that the system does guarantee availability, in terms of the CAP theorem.
- **Soft State**: indicates that the state of the system may change over time, even without input. This is because of the eventual consistency model.
- **Eventual Consistency**: indicates that the system will become consistent over time, given that the system doesn't receive input during that time.

Example: Let's say we have 4 nodes in our system [1-4]. Node 1 and 3 are exact replicas. When something updates in 1, it updates in 3 but there is a gap of time t (15 minutes) for updates to happen. If we write something to node1, the system is not consistent but after 15 min, it will be consistent (eventually consistent), granted nothing else is written to node 1.

In this example, our soft state is 15 minutes.

We also provide basic availability, because if the system was strong consistent, then node 1 and 3 has to match before returning something to the user (if another user queried for what we updated in node 1). If it takes 15 min to sync the two nodes, we have two options (let the user wait 15 min or return an error message). Both options are sacrificing availability.

Youtube: https://www.youtube.com/watch?v=l8SoqbLNN7Q

SQL

ACID is a set of properties of relational database transactions.

- **Atomicity** - Each transaction is all or nothing
- **Consistency** - Any transaction will bring the database from one valid state to another
- **Isolation** - Executing transactions concurrently has the same results as if the transactions were executed serially
- **Durability** - Once a transaction has been committed, it will remain so

Letters/Bits and Bytes

- A bit is an on or off switch
- 8 bits in a byte
- 32 bits = 4 bytes
- Boolean type occupies 1 byte in memory. A bit pattern of all zeros denotes a value of false. A bit pattern with any one or more bits set (analogous to a non-zero integer) denotes a value of true.
- DateTime object occupies 8 bytes in .NET
- Characters in ASCII are 8 bits or 1 byte. Though we can fit all characters in 7 bits, we use 8.
- UNIX Epoch time is a system for describing a point in time. It is the number of seconds that have elapsed since 00:00:00 Thursday, 1 January 1970. End of days will be Jan 19, 2038.

ASCII: American Standard for Information Interchange.

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

C# Primitive Data Types

Chart can be found here:
http://condor.depaul.edu/sjost/nwdp/notes/cs1/CSDatatypes.htm

C# Datatype	Bytes	Range
byte	1	0 to 255
sbyte	1	-128 to 127
short	2	-32,768 to 32,767
ushort	2	0 to 65,535
int	4	-2 billion to 2 billion
uint	4	0 to 4 billion
long	8	-9 quintillion to 9 quintillion
ulong	8	0 to 18 quintillion
float	4	7 significant digits[1]
double	8	15 significant digits[2]
object	4 byte address	All C# Objects
char	2	Unicode characters
string	4 byte address	Length up to 2 billion bytes[3]
decimal	24	28 to 29 significant digits[4]
bool	1	true, false[5]
DateTime	8	0:00:00am 1/1/01 to 11:59:59pm 12/31/9999
DateSpan		-10675199.02:48:05.4775808 to 10675199.02:48:05.4775807[6]

The Byte Table

Here it all is together, which helps to illustrate just how big some of those big numbers get!

Byte Comparison Table		
Metric	**Value**	**Bytes**
Byte (B)	1	1
Kilobyte (KB)	$1{,}024^1$	1,024
Megabyte (MB)	$1{,}024^2$	1,048,576
Gigabyte (GB)	$1{,}024^3$	1,073,741,824
Terabyte (TB)	$1{,}024^4$	1,099,511,627,776
Petabyte (PB)	$1{,}024^5$	1,125,899,906,842,624
Exabyte (EB)	$1{,}024^6$	1,152,921,504,606,846,976
Zettabyte (ZB)	$1{,}024^7$	1,180,591,620,717,411,303,424
Yottabyte (YB)	$1{,}024^8$	1,208,925,819,614,629,174,706,176

SOAP vs Rest

REST: **RE**presentational **S**tate **T**ransfer
SOAP: **S**imple **O**bject **A**ccess **P**rotocol

SOAP only allows XML.
SOAP is a protocol and REST is not.

REST has more flexible architecture than SOAP (loose guidelines, different messaging formats such as HTML, JSON, XML, Plain Text).

REST is more lightweight, therefore super popular in the mobile era.

REST can only be used with HTTP and HTTPS.

The general consensus among experts these days is that REST is the typically preferred protocol unless there's a compelling reason to use SOAP (and there are some cases in which SOAP is preferred).

Why REST?

- Since REST uses standard HTTP it is much simpler in just about every way.
- REST is easier to implement, requires less bandwidth and resources.
- REST permits many different data formats where as SOAP only permits XML.
- REST allows better support for browser clients due to its support for JSON.
- REST has better performance and scalability. REST reads can be cached, SOAP based reads cannot be cached.
- If security is not a major concern and we have limited resources. Or we want to create an API that will be easily used by other developers publicly then we should go with REST.
- If we need Stateless CRUD operations then go with REST.
- REST is commonly used in social media, web chat, mobile services and Public APIs like Google Maps.
- RESTful service return various MediaTypes for the same resource, depending on the request header parameter "Accept" as application/xml or application/json for POST and /user/1234.json or GET /user/1234.xml for GET.
- REST services are meant to be called by the client-side application and not the end user directly.

- ST in REST comes from State Transfer. You transfer the state around instead of having the server store it, this makes REST services scalable.

Why SOAP?

- SOAP is not very easy to implement and requires more bandwidth and resources.
- SOAP message request is processed slower as compared to REST and it does not use web caching mechanism.
- WS-Security: While SOAP supports SSL (just like REST) it also supports WS-Security which adds some enterprise security features.
- WS-AtomicTransaction: Need ACID Transactions over a service, you're going to need SOAP.
- WS-ReliableMessaging: If your application needs Asynchronous processing and a guaranteed level of reliability and security. Rest doesn't have a standard messaging system and expects clients to deal with communication failures by retrying.
- If the security is a major concern and the resources are not limited then we should use SOAP web services. Like if we are creating a web service for payment gateways, financial and telecommunication related work then we should go with SOAP as here high security is needed.

Internet Protocol

- **HTTP:** Request/Response. Web runs on HTTP
- **HTTP2:** Improvements over HTTP. HTTP/2 supports multiple requests, headers compression, priority and more intelligent packet streaming management. This results in reduced latency and accelerates content download on modern web pages.
- **Web Sockets:** Bi-directional communication between client and server.
- **SMTP**
- **TCP/IP:**
    - **These are the layers:**

- **IPV4 vs IPV6**
    - IPV4: 32 bit addresses
    - IPV6: 128 bit addresses
- **How do we route web traffic?**
- **TCP vs UDP**
- **DNS Lookup**
- **HTTPS vs TLS**

Back of the Envelope Calculations

Handy conversion guide:

- 2.5 million seconds per month
- 1 request per second = 2.5 million requests per month
- 40 requests per second = 100 million requests per month
- 400 requests per second = 1 billion requests per month

HTTP Codes

400:

Difference between a Thread and Process

To Be Reliable

To be truly reliable, a distributed system must have the following characteristics:

- **Fault-Tolerant**: It can recover from component failures without performing incorrect actions.
- **Highly Available**: It can restore operations, permitting it to resume providing services even when some components have failed.
- **Recoverable**: Failed components can restart themselves and rejoin the system, after the cause of failure has been repaired.

Consistent: The system can coordinate actions by multiple components often in the presence of concurrency and failure. This underlies the ability of a distributed system to act like a non-distributed system.

Scalable: It can operate correctly even as some aspect of the system is scaled to a larger size. For example, we might increase the size of the network on which the system is running. This increases the frequency of network outages and could degrade a "non-scalable" system. Similarly, we might increase the number of users or servers, or overall load on the system. In a scalable system, this should not have a significant effect.

- Predictable Performance: The ability to provide desired responsiveness in a timely manner.

Secure: The system authenticates access to data and services

CAP Theorem

CAP stands for consistency, availability, partition tolerance.
Consistency: Consistency means that data is the same across the cluster, so you can read or write from/to any node and get the same data.
Availability: Availability means the ability to access the cluster even if a node in the cluster goes down.
Partition Tolerance: Partition tolerance means that the cluster continues to function even if there is a "partition" (communication break) between two nodes (both nodes are up, but can't communicate).

Consistency means that data is the same across the cluster, so you can read or write from/to any node and get the same data.

Availability means the ability to access the cluster even if a node in the cluster goes down.

Partition tolerance means that the cluster continues to function even if there is a "partition" (communication break) between two nodes (both nodes are up, but can't communicate).

In order to get both availability and partition tolerance, you have to give up consistency. Consider if you have two nodes, X and Y, in a master-master setup. Now, there is a break between network communication between X and Y, so they can't sync updates. At this point you can either:

A) Allow the nodes to get out of sync (giving up consistency), or

B) Consider the cluster to be "down" (giving up availability)

All the combinations available are:

- **CA** - data is consistent between all nodes - as long as all nodes are online - and you can read/write from any node and be sure that the data is the same, but if you ever develop a partition between nodes, the data will be out of sync (and won't re-sync once the partition is resolved).
- **CP** - data is consistent between all nodes, and maintains partition tolerance (preventing data desync) by becoming unavailable when a node goes down.
- **AP** - nodes remain online even if they can't communicate with each other and will resync data once the partition is resolved, but you aren't guaranteed that all nodes will have the same data (either during or after the partition)

You should note that CA systems don't practically exist (even if some systems claim to be so).

On Consistency:
Most people seem to understand this, but it bears repetition: a system is consistent if an update is applied to all relevant nodes at the same logical time. Among other things, this means that standard database replication is not strongly consistent. As anyone whose read replicas have drifted from the master knows, special logic must be introduced to handle replication lag.

That said, consistency which is both instantaneous and global is impossible. The universe simply does not permit it. So the goal here is to push the time resolutions at which the consistency breaks down to a point where we no longer notice it. Just don't try to act outside your own light cone...

On Availability:

Despite the notion of "100% uptime as much as possible," there are limits to availability. If you have a single piece of data on five nodes and all five nodes die, that data is gone and any request which required it in order to be processed cannot be handled.

On Partition Tolerance

For a distributed (i.e., multi-node) system to not require partition-tolerance it would have to run on a network which is guaranteed to never drop messages(or even deliver them late) and whose nodes are guaranteed to never die. You and I do not work with these types of systems because they don't exist.

Optimistic vs Pessimistic Locking

Transactional isolation is usually implemented by locking whatever is accessed in a transaction. There are two different approaches to transactional locking: Pessimistic locking and optimistic locking.
The disadvantage of pessimistic locking is that a resource is locked from the time it is first accessed in a transaction until the transaction is finished, making it inaccessible to other transactions during that time. If most transactions simply look at the resource and never change it, an exclusive lock may be overkill as it may cause lock contention, and optimistic locking may be a better approach. With pessimistic locking, locks are applied in a fail-safe way. In the banking application example, an account is locked as

soon as it is accessed in a transaction. Attempts to use the account in other transactions while it is locked will either result in the other process being delayed until the account lock is released, or that the process transaction will be rolled back. The lock exists until the transaction has either been committed or rolled back.

With optimistic locking, a resource is not actually locked when it is first is accessed by a transaction. Instead, the state of the resource at the time when it would have been locked with the pessimistic locking approach is saved. Other transactions are able to concurrently access to the resource and the possibility of conflicting changes is possible. At commit time, when the resource is about to be updated in persistent storage, the state of the resource is read from storage again and compared to the state that was saved when the resource was first accessed in the transaction. If the two states differ, a conflicting update was made, and the transaction will be rolled back.

In the banking application example, the amount of an account is saved when the account is first accessed in a transaction. If the transaction changes the account amount, the amount is read from the store again just before the amount is about to be updated. If the amount has changed since the transaction began, the transaction will fail itself, otherwise the new amount is written to persistent storage.

Types of No SQL

- **KeyValue:**
- **Wide Column**
- **Document Based**
- **Graph Based**

Data Center/Racks/Hosts

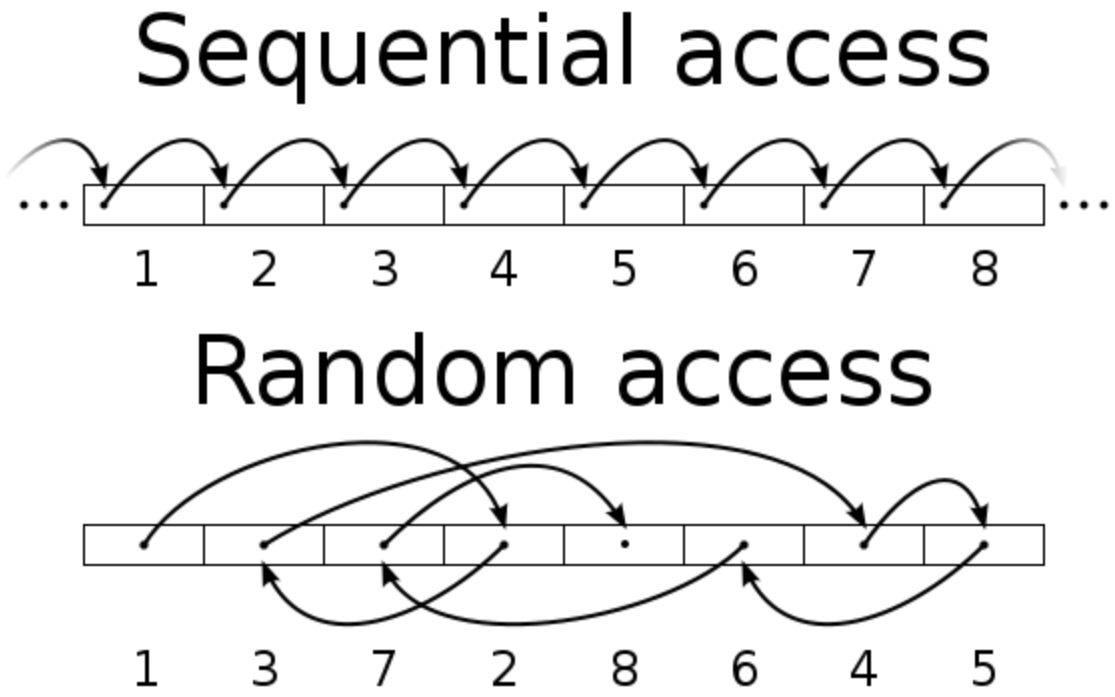
How are data centers architected/arranged?
What is the latency talking across racks/hosts?
What happens if a rack goes down or data center goes down?

Random vs Sequential Read/Write on Disk

When people talk about sequential vs random writes to a file, they're generally drawing a distinction between writing without intermediate seeks ("sequential"), vs. a pattern of seek-write-seek-write-seek-write, etc. ("random").

The distinction is very important in traditional disk-based systems, where each disk seek will take around 10ms. Sequentially writing data to that same disk takes about 30ms per MB. So if you sequentially write 100MB of data to a disk, it will take around 3 seconds. But if you do 100 random writes of 1MB each, that will take a total of 4 seconds (3 seconds for the actual writing, and 10ms*100 == 1 second for all the seeking).

As each random write gets smaller, you pay more and more of a penalty for the disk seeks. In the extreme case where you perform 100 million random 1-byte writes, you'll still net 3 seconds for all the actual writes, but you'd now have 11.57 days worth of seeking to do! So clearly the degree to which your writes are sequential vs. random can really affect the time it takes to accomplish your task.



Public Key Infrastructure & Certificate Authority

Symmetric vs Asymmetric Key

Bloom Filters & Count-Min Sketch

Paxos - Consensus over distributed hosts. Leader election

Virtual Machines vs Containers

Operating system in a shared hardware environment.
Containers are isolated environments where you can run your software.

Publisher-Subscriber vs Queue

Software:

Cassandra:

- Wide column, highly scalable database.
- Key value store
- Time series data
- Rows with many columns
- Supports both eventual and strong consistency
- Consistent hashing under hood to shard data
- Use gossiping to keep all the nodes informed about the cluster

MongoDB/Couchbase:

- ACID properties at a document level
- Great for JSON structures
- Scales well

MySQL:

- Strong ACID
- Many tables and relationships
- Supports master/slave architecture

Memcached/Redis:

- Distributed cache
- Hold data in memory
- Memcached is simple, fast KeyValue. Redis can support KeyValue and much more.
- Redis can be set up as a cluster so it can provide more availability and data applications.
- Redis can also flush data to the hard drive.

ZooKeeper:

- Centralized configuration management tool
- Used for leader election and distributed locking
- ZooKeeper scales really well for reads but not well for writes.
- ZooKeeper keeps all data in memory.

Kaufka:

- Fault tolerant highly available queue. Used by publisher/subscriber or steaming applications.
- Can deliver messages exactly.
- Keeps all the messages ordered inside a topic.

NGINX/HAProxy:

- Load balancers
- NGINX can manage 10s of thousands of connections from a single instance

Solar/Elastic Search:

- Search platforms
- Full text search

Blobstore like Amazon S3

- Picture or File storage

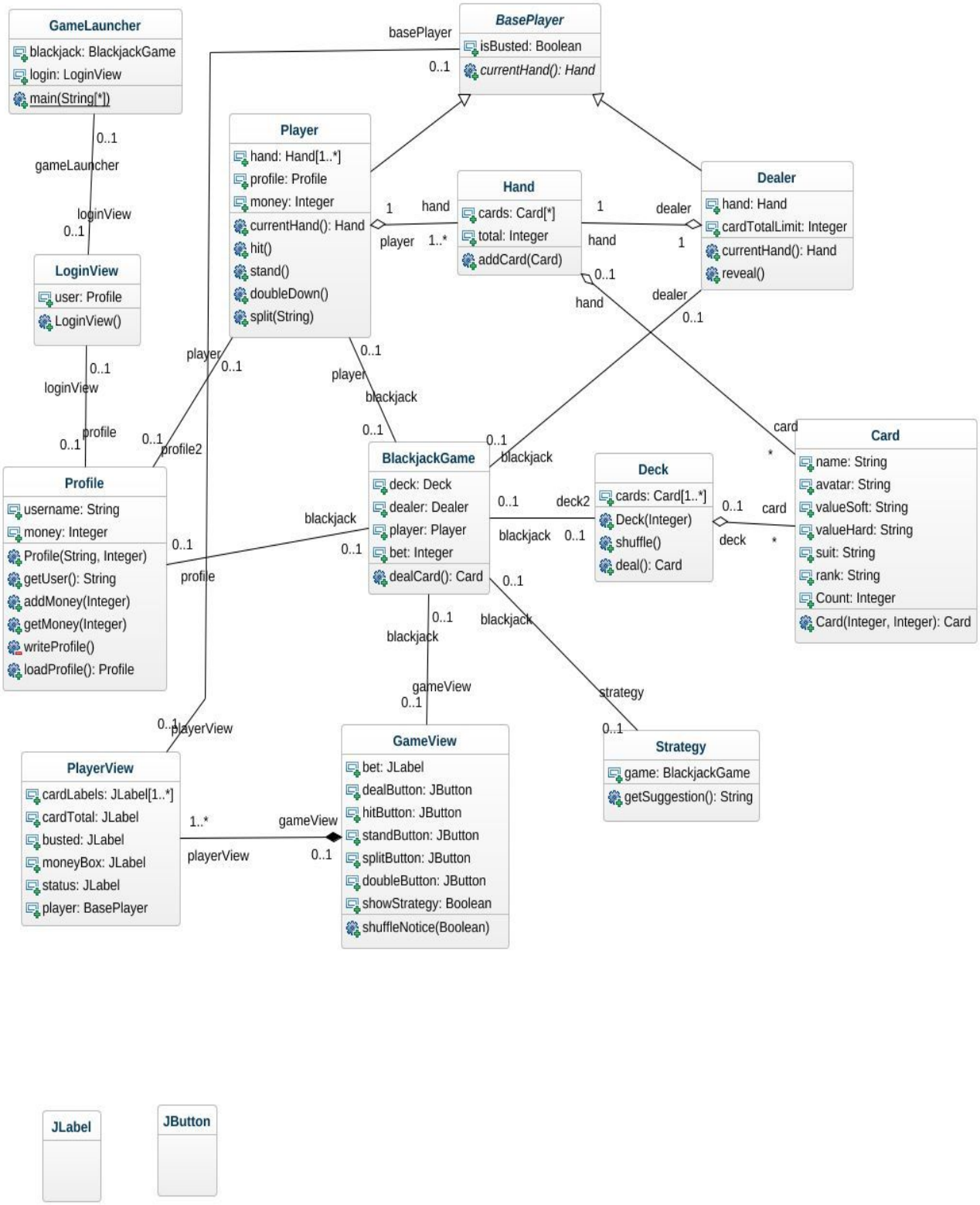
Docker: Containers for distributed applications. Containers can run on laptop, data centers or cloud.

- **Kubernetes/Mesos**
    - Tools to manage and coordinate containers

Hadoop/Spark: Map Reduce. Spark is faster than Hadoop because data processing is done in memory.

- **HDFS:** Java based file system which is distributed and fault tolerant. Hadoop relies on HDFS for processing.

Blackjack Game UML


GameLauncher
blackjack: BlackjackGame
login: LoginView
main(String[*])
BasePlayer
isBusted: Boolean
currentHand(): Hand
basePlayer
0..1
Player
hand: Hand[1..*]
profile: Profile
money: Integer
currentHand(): Hand
hit()
stand()
doubleDown()
split(String)
Hand
cards: Card[*]
total: Integer
addCard(Card)
Dealer
hand: Hand
cardTotalLimit: Integer
currentHand(): Hand
reveal()
gameLauncher
loginView
LoginView
user: Profile
LoginView()
player
blackjack
profile
profile2
hand
dealer
BlackjackGame
deck: Deck
dealer: Dealer
player: Player
bet: Integer
dealCard(): Card
Deck
cards: Card[1..*]
Deck(Integer)
shuffle()
deal(): Card
deck2
card
Card
name: String
avatar: String
valueSoft: String
valueHard: String
suit: String
rank: String
Count: Integer
Card(Integer, Integer): Card
Profile
username: String
money: Integer
Profile(String, Integer)
getUser(): String
addMoney(Integer)
getMoney(Integer)
writeProfile()
loadProfile(): Profile
PlayerView
cardLabels: JLabel[1..*]
cardTotal: JLabel
busted: JLabel
moneyBox: JLabel
status: JLabel
player: BasePlayer
playerView
gameView
GameView
bet: JLabel
dealButton: JButton
hitButton: JButton
standButton: JButton
splitButton: JButton
doubleButton: JButton
showStrategy: Boolean
shuffleNotice(Boolean)
strategy
Strategy
game: BlackjackGame
getSuggestion(): String
JLabel
JButton

One Liners

- Relational databases pick consistency over availability. NoSQL databases pick availability over consistency, granted you configure them that way. This all depends on the configuration. But having eventual consistency means more availability.
- CPU, memory, hard-drive, network bandwidth are expensive are limited resources. When designing a system, how do we work around those limitations. How can we scale/provide low latency at a budget.