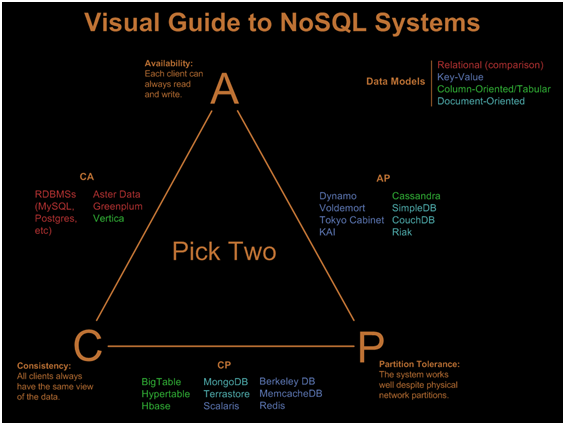
* If you have a **performance** problem, your system is slow for a single user.
* If you have a **scalability** problem, your system is fast for a single user but slow under heavy load.
* **Latency** is the time to perform some action or to produce some result.
* **Throughput** is the number of such actions or results per unit of time.



* **Consistency** - Every read receives the most recent write or an error
* **Availability** - Every request receives a response, without guarantee that it contains the most recent version of the information
* **Partition Tolerance** - The system continues to operate despite arbitrary partitioning due to network failures

**CP - consistency and partition tolerance**

Waiting for a response from the partitioned node might result in a timeout error. CP is a good choice if your business needs require atomic reads and writes.

**AP - availability and partition tolerance**

Responses return the most recent version of the data available on a node, which might not be the latest. Writes might take some time to propagate when the partition is resolved.

AP is a good choice if the business needs allow for [eventual consistency](https://github.com/donnemartin/system-design-primer#eventual-consistency) or when the system needs to continue working despite external errors.

### Weak consistency

After a write, reads may or may not see it. A best effort approach is taken.

### Eventual consistency

After a write, reads will eventually see it (typically within milliseconds). Data is replicated asynchronously.

### Strong consistency

After a write, reads will see it. Data is replicated synchronously.

## Availability patterns

There are two main patterns to support high availability: **fail-over** and **replication**.

### Fail-over

#### Active-passive

With active-passive fail-over, heartbeats are sent between the active and the passive server on standby. If the heartbeat is interrupted, the passive server takes over the active's IP address and resumes service.

#### Active-active

In active-active, both servers are managing traffic, spreading the load between them.

### Disadvantage(s): failover

* Fail-over adds more hardware and additional complexity.
* There is a potential for loss of data if the active system fails before any newly written data can be replicated to the passive.

#### Availability in parallel vs in sequence

###### In sequence

Overall availability decreases when two components with availability < 100% are in sequence:

Availability (Total) = Availability (Foo) \* Availability (Bar)

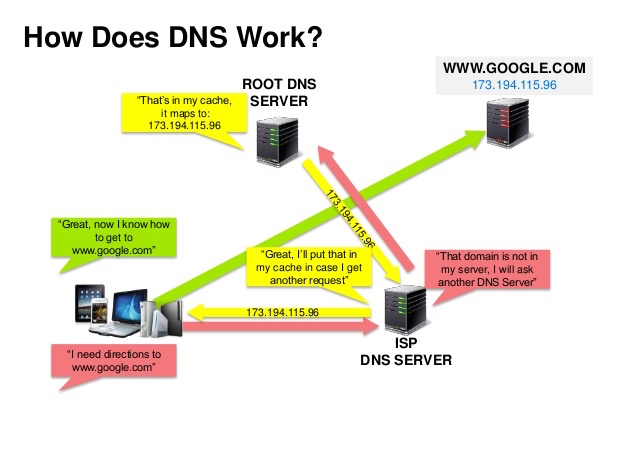
###### In parallel

Overall availability increases when two components with availability < 100% are in parallel:

Availability (Total) = 1 - (1 - Availability (Foo)) \* (1 - Availability (Bar))

## Domain name system

A Domain Name System (DNS) translates a domain name such as [www.example.com](http://www.example.com) to an IP address.



### Disadvantage(s): DNS

* Accessing a DNS server introduces a slight delay, although mitigated by caching described above.
* DNS server management could be complex and is generally managed by [governments, ISPs, and large companies](http://superuser.com/questions/472695/who-controls-the-dns-servers/472729).
* DNS services have recently come under [DDoS attack](http://dyn.com/blog/dyn-analysis-summary-of-friday-october-21-attack/), preventing users from accessing websites such as Twitter without knowing Twitter's IP address(es).

## Content delivery network

A content delivery network (CDN) is a globally distributed network of proxy servers, serving content from locations closer to the user. Generally, static files such as HTML/CSS/JS, photos, and videos are served from CDN, although some CDNs such as Amazon's CloudFront support dynamic content. The site's DNS resolution will tell clients which server to contact.

### Push CDNs

Push CDNs receive new content whenever changes occur on your server.

### Pull CDNs

Pull CDNs grab new content from your server when the first user requests the content.



### Disadvantage(s): CDN

* CDN costs could be significant depending on traffic, although this should be weighed with additional costs you would incur not using a CDN.
* Content might be stale if it is updated before the TTL expires it.
* CDNs require changing URLs for static content to point to the CDN.

## Load balancer

Load balancers distribute incoming client requests to computing resources such as application servers and databases.

Load balancers can be implemented with hardware (expensive) or with software such as HAProxy.

To protect against failures, it's common to set up multiple load balancers, either in [active-passive](https://github.com/donnemartin/system-design-primer#active-passive) or [active-active](https://github.com/donnemartin/system-design-primer#active-active) mode.

### Layer 4 load balancing

Layer 4 load balancers look at info at the [transport layer](https://github.com/donnemartin/system-design-primer#communication) to decide how to distribute requests.

### Layer 7 load balancing

Layer 7 load balancers look at the [application layer](https://github.com/donnemartin/system-design-primer#communication) to decide how to distribute requests.

### Horizontal scaling

Load balancers can also help with horizontal scaling, improving performance and availability. Scaling out using commodity machines is more cost efficient and results in higher availability than scaling up a single server on more expensive hardware, called **Vertical Scaling**.

#### Disadvantage(s): horizontal scaling

* Scaling horizontally introduces complexity and involves cloning servers
  + Servers should be stateless: they should not contain any user-related data like sessions or profile pictures
  + Sessions can be stored in a centralized data store such as a [database](https://github.com/donnemartin/system-design-primer#database) (SQL, NoSQL) or a persistent [cache](https://github.com/donnemartin/system-design-primer#cache) (Redis, Memcached)
* Downstream servers such as caches and databases need to handle more simultaneous connections as upstream servers scale out

### Disadvantage(s): load balancer

* The load balancer can become a performance bottleneck if it does not have enough resources or if it is not configured properly.
* Introducing a load balancer to help eliminate single points of failure results in increased complexity.
* A single load balancer is a single point of failure, configuring multiple load balancers further increases complexity.

## Reverse proxy (web server)

A reverse proxy is a web server that centralizes internal services and provides unified interfaces to the public.

Additional benefits include:

* **Increased security** - Hide information about backend servers, blacklist IPs, limit number of connections per client
* **Increased scalability and flexibility** - Clients only see the reverse proxy's IP, allowing you to scale servers or change their configuration
* **SSL termination** - Decrypt incoming requests and encrypt server responses so backend servers do not have to perform these potentially expensive operations
  + Removes the need to install [X.509 certificates](https://en.wikipedia.org/wiki/X.509) on each server
* **Compression** - Compress server responses
* **Caching** - Return the response for cached requests
* **Static content** - Serve static content directly

### Load balancer vs reverse proxy

* Deploying a load balancer is useful when you have multiple servers. Often, load balancers route traffic to a set of servers serving the same function.
* Reverse proxies can be useful even with just one web server or application server, opening up the benefits described in the previous section.
* Solutions such as NGINX and HAProxy can support both layer 7 reverse proxying and load balancing.

### Disadvantage(s): reverse proxy

* Introducing a reverse proxy results in increased complexity.
* A single reverse proxy is a single point of failure, configuring multiple reverse proxies (ie a [failover](https://en.wikipedia.org/wiki/Failover)) further increases complexity.

## Application layer

The **single responsibility principle** advocates for small and autonomous services that work together. Small teams with small services can plan more aggressively for rapid growth.

### Microservices

Related to this discussion are [microservices](https://en.wikipedia.org/wiki/Microservices), which can be described as a suite of independently deployable, small, modular services.

### Service Discovery

Systems such as [Consul](https://www.consul.io/docs/index.html), [Etcd](https://coreos.com/etcd/docs/latest), and [Zookeeper](http://www.slideshare.net/sauravhaloi/introduction-to-apache-zookeeper) can help services find each other by keeping track of registered names, addresses, and ports.

### Disadvantage(s): application layer

* Adding an application layer with loosely coupled services requires a different approach from an architectural, operations, and process viewpoint (vs a monolithic system).
* Microservices can add complexity in terms of deployments and operations.

## Database

### Relational database management system (RDBMS)

A relational database like SQL is a collection of data items organized in tables.

**ACID** is a set of properties of relational database [transactions](https://en.wikipedia.org/wiki/Database_transaction).

* **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

There are many techniques to scale a relational database: **master-slave replication**, **master-master replication**, **federation**, **sharding**, **denormalization**, and **SQL tuning**.

#### Master-slave replication

The master serves reads and writes, replicating writes to one or more slaves, which serve only reads. Slaves can also replicate to 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.

##### Disadvantage(s): master-slave replication

* Additional logic is needed to promote a slave to a master.
* See [Disadvantage(s): replication](https://github.com/donnemartin/system-design-primer#disadvantages-replication) for points related to **both** master-slave and master-master.

#### Master-master replication

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

##### isadvantage(s): master-master replication

* You'll need a load balancer or you'll need to make changes to your application logic to determine where to write.
* Most master-master systems are either loosely consistent (violating ACID) or have increased write latency due to synchronization.
* Conflict resolution comes more into play as more write nodes are added and as latency increases.
* See [Disadvantage(s): replication](https://github.com/donnemartin/system-design-primer#disadvantages-replication) for points related to **both** master-slave and master-master.

##### Disadvantage(s): replication

* There is a potential for loss of data if the master fails before any newly written data can be replicated to other nodes.
* Writes are replayed to the read replicas. If there are a lot of writes, the read replicas can get bogged down with replaying writes and can't do as many reads.
* The more read slaves, the more you have to replicate, which leads to greater replication lag.
* On some systems, writing to the master can spawn multiple threads to write in parallel, whereas read replicas only support writing sequentially with a single thread.
* Replication adds more hardware and additional complexity.

#### Federation

Federation (or functional partitioning) splits up databases by function. For example, instead of a single, monolithic database, you could have three databases.

##### Disadvantage(s): federation

* Federation is not effective if your schema requires huge functions or tables.
* You'll need to update your application logic to determine which database to read and write.
* Joining data from two databases is more complex with a [server link](http://stackoverflow.com/questions/5145637/querying-data-by-joining-two-tables-in-two-database-on-different-servers).
* Federation adds more hardware and additional complexity.

#### Sharding

Sharding distributes data across different databases such that each database can only manage a subset of the data.

##### Disadvantage(s): sharding

* You'll need to update your application logic to work with shards, which could result in complex SQL queries.
* Data distribution can become lopsided in a shard. For example, a set of power users on a shard could result in increased load to that shard compared to others.
  + Rebalancing adds additional complexity. A sharding function based on [consistent hashing](http://www.paperplanes.de/2011/12/9/the-magic-of-consistent-hashing.html) can reduce the amount of transferred data.
* Joining data from multiple shards is more complex.
* Sharding adds more hardware and additional complexity.

#### Denormalization

Denormalization attempts to improve read performance at the expense of some write performance. Redundant copies of the data are written in multiple tables to avoid expensive joins. Some RDBMS such as [PostgreSQL](https://en.wikipedia.org/wiki/PostgreSQL) and Oracle support [materialized views](https://en.wikipedia.org/wiki/Materialized_view) which handle the work of storing redundant information and keeping redundant copies consistent.

##### Disadvantage(s): denormalization

* Data is duplicated.
* Constraints can help redundant copies of information stay in sync, which increases complexity of the database design.
* A denormalized database under heavy write load might perform worse than its normalized counterpart.

#### SQL tuning

##### Tighten up the schema

* MySQL dumps to disk in contiguous blocks for fast access.
* Use CHAR instead of VARCHAR for fixed-length fields.
  + CHAR effectively allows for fast, random access, whereas with VARCHAR, you must find the end of a string before moving onto the next one.
* Use TEXT for large blocks of text such as blog posts. TEXT also allows for boolean searches. Using a TEXT field results in storing a pointer on disk that is used to locate the text block.
* Use INT for larger numbers up to 2^32 or 4 billion.
* Use DECIMAL for currency to avoid floating point representation errors.
* Avoid storing large BLOBS, store the location of where to get the object instead.
* VARCHAR(255) is the largest number of characters that can be counted in an 8 bit number, often maximizing the use of a byte in some RDBMS.
* Set the NOT NULL constraint where applicable to [improve search performance](http://stackoverflow.com/questions/1017239/how-do-null-values-affect-performance-in-a-database-search).

##### Use good indices

* Columns that you are querying (SELECT, GROUP BY, ORDER BY, JOIN) could be faster with indices.
* Indices are usually represented as self-balancing [B-tree](https://en.wikipedia.org/wiki/B-tree) that keeps data sorted and allows searches, sequential access, insertions, and deletions in logarithmic time.
* Placing an index can keep the data in memory, requiring more space.
* Writes could also be slower since the index also needs to be updated.
* When loading large amounts of data, it might be faster to disable indices, load the data, then rebuild the indices.

##### Avoid expensive joins

* [Denormalize](https://github.com/donnemartin/system-design-primer#denormalization) where performance demands it.

##### Partition tables

* Break up a table by putting hot spots in a separate table to help keep it in memory.

##### Tune the query cache

* In some cases, the [query cache](https://dev.mysql.com/doc/refman/5.7/en/query-cache.html) could lead to [performance issues](https://www.percona.com/blog/2016/10/12/mysql-5-7-performance-tuning-immediately-after-installation/).

### NoSQL

NoSQL is a collection of data items represented in a **key-value store**, **document store**, **wide column store**, or a **graph database**. Data is denormalized, and joins are generally done in the application code. Most NoSQL stores lack true ACID transactions and favor [eventual consistency](https://github.com/donnemartin/system-design-primer#eventual-consistency).

**BASE** is often used to describe the properties of NoSQL databases. In comparison with the [CAP Theorem](https://github.com/donnemartin/system-design-primer#cap-theorem), BASE chooses availability over consistency.

* **Basically available** - the system guarantees availability.
* **Soft state** - the state of the system may change over time, even without input.
* **Eventual consistency** - the system will become consistent over a period of time, given that the system doesn't receive input during that period.

#### Key-value store

A key-value store generally allows for O(1) reads and writes and is often backed by memory or SSD.

#### Document store

A document store is centered around documents (XML, JSON, binary, etc), where a document stores all information for a given object

Some document stores like [MongoDB](https://www.mongodb.com/mongodb-architecture) and [CouchDB](https://blog.couchdb.org/2016/08/01/couchdb-2-0-architecture/) also provide a SQL-like language to perform complex queries. [DynamoDB](http://www.read.seas.harvard.edu/~kohler/class/cs239-w08/decandia07dynamo.pdf) supports both key-values and documents.

#### Wide column store

A wide column store's basic unit of data is a column (name/value pair). A column can be grouped in column families (analogous to a SQL table)

Google introduced [Bigtable](http://www.read.seas.harvard.edu/~kohler/class/cs239-w08/chang06bigtable.pdf) as the first wide column store, which influenced the open-source [HBase](https://www.mapr.com/blog/in-depth-look-hbase-architecture) often-used in the Hadoop ecosystem, and [Cassandra](http://docs.datastax.com/en/cassandra/3.0/cassandra/architecture/archIntro.html) from Facebook.

**Wide column stores offer high availability and high scalability. They are often used for very large data sets.**

#### Graph database

In a graph database, each node is a record and each arc is a relationship between two nodes. Graph databases are optimized to represent complex relationships with many foreign keys or many-to-many relationships.

### SQL or NoSQL

Reasons for **SQL**:

* Structured data
* Strict schema
* Relational data
* Need for complex joins
* Transactions
* Clear patterns for scaling
* More established: developers, community, code, tools, etc
* Lookups by index are very fast

Reasons for **NoSQL**:

* Semi-structured data
* Dynamic or flexible schema
* Non-relational data
* No need for complex joins
* Store many TB (or PB) of data
* Very data intensive workload
* Very high throughput for IOPS

## Cache

Caching improves page load times and can reduce the load on your servers and databases.

### When to update the cache

#### Cache-aside

The application is responsible for reading and writing from storage. The cache does not interact with storage directly. The application does the following:

* Look for entry in cache, resulting in a cache miss
* Load entry from the database
* Add entry to cache
* Return entry

##### Disadvantage(s): cache-aside

* Each cache miss results in three trips, which can cause a noticeable delay.
* Data can become stale if it is updated in the database. This issue is mitigated by setting a time-to-live (TTL) which forces an update of the cache entry, or by using write-through.
* When a node fails, it is replaced by a new, empty node, increasing latency.

#### Write-through

The application uses the cache as the main data store, reading and writing data to it, while the cache is responsible for reading and writing to the database:

* Application adds/updates entry in cache
* Cache synchronously writes entry to data store
* Return

Write-through is a slow overall operation due to the write operation, but subsequent reads of just written data are fast.

##### Disadvantage(s): write through

* When a new node is created due to failure or scaling, the new node will not cache entries until the entry is updated in the database. Cache-aside in conjunction with write through can mitigate this issue.
* Most data written might never be read, which can be minimized with a TTL.

#### Write-behind (write-back)

In write-behind, the application does the following:

* Add/update entry in cache
* Asynchronously write entry to the data store, improving write performance

##### Disadvantage(s): write-behind

* There could be data loss if the cache goes down prior to its contents hitting the data store.
* It is more complex to implement write-behind than it is to implement cache-aside or write-through.

#### Refresh-ahead

You can configure the cache to automatically refresh any recently accessed cache entry prior to its expiration.

##### Disadvantage(s): refresh-ahead

* Not accurately predicting which items are likely to be needed in the future can result in reduced performance than without refresh-ahead.

### Disadvantage(s): cache

* Need to maintain consistency between caches and the source of truth such as the database through [cache invalidation](https://en.wikipedia.org/wiki/Cache_algorithms).
* Cache invalidation is a difficult problem, there is additional complexity associated with when to update the cache.
* Need to make application changes such as adding Redis or memcached.

## Asynchronism

### Message queues

Message queues receive, hold, and deliver messages.

* An application publishes a job to the queue, then notifies the user of job status
* A worker picks up the job from the queue, processes it, then signals the job is complete

### Disadvantage(s): asynchronism

* Use cases such as inexpensive calculations and realtime workflows might be better suited for synchronous operations, as introducing queues can add delays and complexity.

### Hypertext transfer protocol (HTTP)

HTTP is a method for encoding and transporting data between a client and a server.

A basic HTTP request consists of a verb (method) and a resource (endpoint). Below are common HTTP verbs:

| **Verb** | **Description** | **Idempotent\*** | **Safe** | **Cacheable** |
| --- | --- | --- | --- | --- |
| GET | Reads a resource | Yes | Yes | Yes |
| POST | Creates a resource or trigger a process that handles data | No | No | Yes if response contains freshness info |
| PUT | Creates or replace a resource | Yes | No | No |
| PATCH | Partially updates a resource | No | No | Yes if response contains freshness info |
| DELETE | Deletes a resource | Yes | No | No |

\*Can be called many times without different outcomes.

### Transmission control protocol (TCP)

TCP is a connection-oriented protocol over an [IP network](https://en.wikipedia.org/wiki/Internet_Protocol). Connection is established and terminated using a [handshake](https://en.wikipedia.org/wiki/Handshaking). All packets sent are guaranteed to reach the destination in the original order and without corruption.

If the sender does not receive a correct response, it will resend the packets. If there are multiple timeouts, the connection is dropped.

TCP is useful for applications that require high reliability but are less time critical. Some examples include web servers, database info, SMTP, FTP, and SSH.

Use TCP over UDP when:

* You need all of the data to arrive intact
* You want to automatically make a best estimate use of the network throughput

### User datagram protocol (UDP)

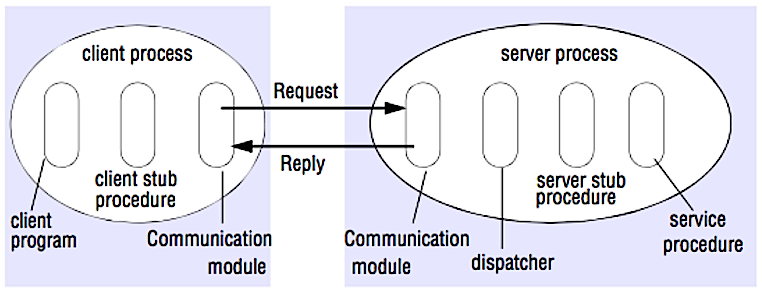
UDP is connectionless. Datagrams (analogous to packets) are guaranteed only at the datagram level. Datagrams might reach their destination out of order or not at all. Without the guarantees that TCP support, UDP is generally more efficient.

UDP can broadcast, sending datagrams to all devices on the subnet. This is useful with [DHCP](https://en.wikipedia.org/wiki/Dynamic_Host_Configuration_Protocol) because the client has not yet received an IP address, thus preventing a way for TCP to stream without the IP address.

Use UDP over TCP when:

* You need the lowest latency
* Late data is worse than loss of data
* You want to implement your own error correction

### Remote procedure call (RPC)



In an RPC, a client causes a procedure to execute on a different address space, usually a remote server. Popular RPC frameworks include [Protobuf](https://developers.google.com/protocol-buffers/), [Thrift](https://thrift.apache.org/), and [Avro](https://avro.apache.org/docs/current/).

Protocol buffers are Google's language-neutral, platform-neutral, extensible mechanism for serializing structured data – think XML, but smaller, faster, and simpler.

RPC is focused on exposing behaviors. RPCs are often used for performance reasons with internal communications, as you can hand-craft native calls to better fit your use cases.

HTTP APIs following **REST** tend to be used more often for public APIs.

#### Disadvantage(s): RPC

* RPC clients become tightly coupled to the service implementation.
* A new API must be defined for every new operation or use case.
* It can be difficult to debug RPC.
* You might not be able to leverage existing technologies out of the box. For example, it might require additional effort to ensure [RPC calls are properly cached](http://etherealbits.com/2012/12/debunking-the-myths-of-rpc-rest/) on caching servers such as [Squid](http://www.squid-cache.org/).

### Representational state transfer (REST)

REST is an architectural style enforcing a client/server model where the client acts on a set of resources managed by the server.

REST is focused on exposing data. It minimizes the coupling between client/server and is often used for public HTTP APIs.

#### Disadvantage(s): REST

* With REST being focused on exposing data, it might not be a good fit if resources are not naturally organized or accessed in a simple hierarchy. For example, returning all updated records from the past hour matching a particular set of events is not easily expressed as a path. With REST, it is likely to be implemented with a combination of URI path, query parameters, and possibly the request body.
* REST typically relies on a few verbs (GET, POST, PUT, DELETE, and PATCH) which sometimes doesn't fit your use case. For example, moving expired documents to the archive folder might not cleanly fit within these verbs.
* Fetching complicated resources with nested hierarchies requires multiple round trips between the client and server to render single views, e.g. fetching content of a blog entry and the comments on that entry. For mobile applications operating in variable network conditions, these multiple roundtrips are highly undesirable.
* Over time, more fields might be added to an API response and older clients will receive all new data fields, even those that they do not need, as a result, it bloats the payload size and leads to larger latencies.

#### Rest API Versioning

* **URI Versioning** – version the URI space using version indicators
* **Media Type Versioning** – version the Representation of the Resource

### URI Versioning

Basic approach to versioning is to create a completely different URI for the new service.

### Request Parameter versioning

Next approach to versioning is to use the request parameter to differentiate versions.

### (Custom) Headers versioning

The third approach to versioning is to use a Request Header to differentiate the versions.

Examples

* http://localhost:8080/person/header
  + headers=[X-API-VERSION=1]

@GetMapping(value = "/student/header", headers = "X-API-VERSION=1")

public StudentV1 headerV1() {

return new StudentV1("Bob Charlie");

}

### Media type versioning (a.k.a “content negotiation” or “accept header”)

The last versioning approach is to use the Accept Header in the request.

Examples

* http://localhost:8080/person/produces
  + headers[Accept=application/vnd.company.app-v1+json]

@GetMapping(value = "/student/produces", produces = "application/vnd.company.app-v1+json")

public StudentV1 producesV1() {

return new StudentV1("Bob Charlie");

}

## Factors affecting Versioning Choice

Following factors affect the choice of versioning:

* URI Pollution - URL versions and Request Param versioning pollute the URI space.
* Misuse of HTTP Headers - Accept Header is not designed to be used for versioning.
* Caching - If you use Header based versioning, we cannot cache just based on the URL. You would need take the specific header into consideration.
* Can we execute the request on the browser? - If you have non technical consumers, then the URL based version would be easier to use as they can be executed directly on the browser.
* API Documentation - How do you get your documentation generation to understand that two different urls are versions of the same service?

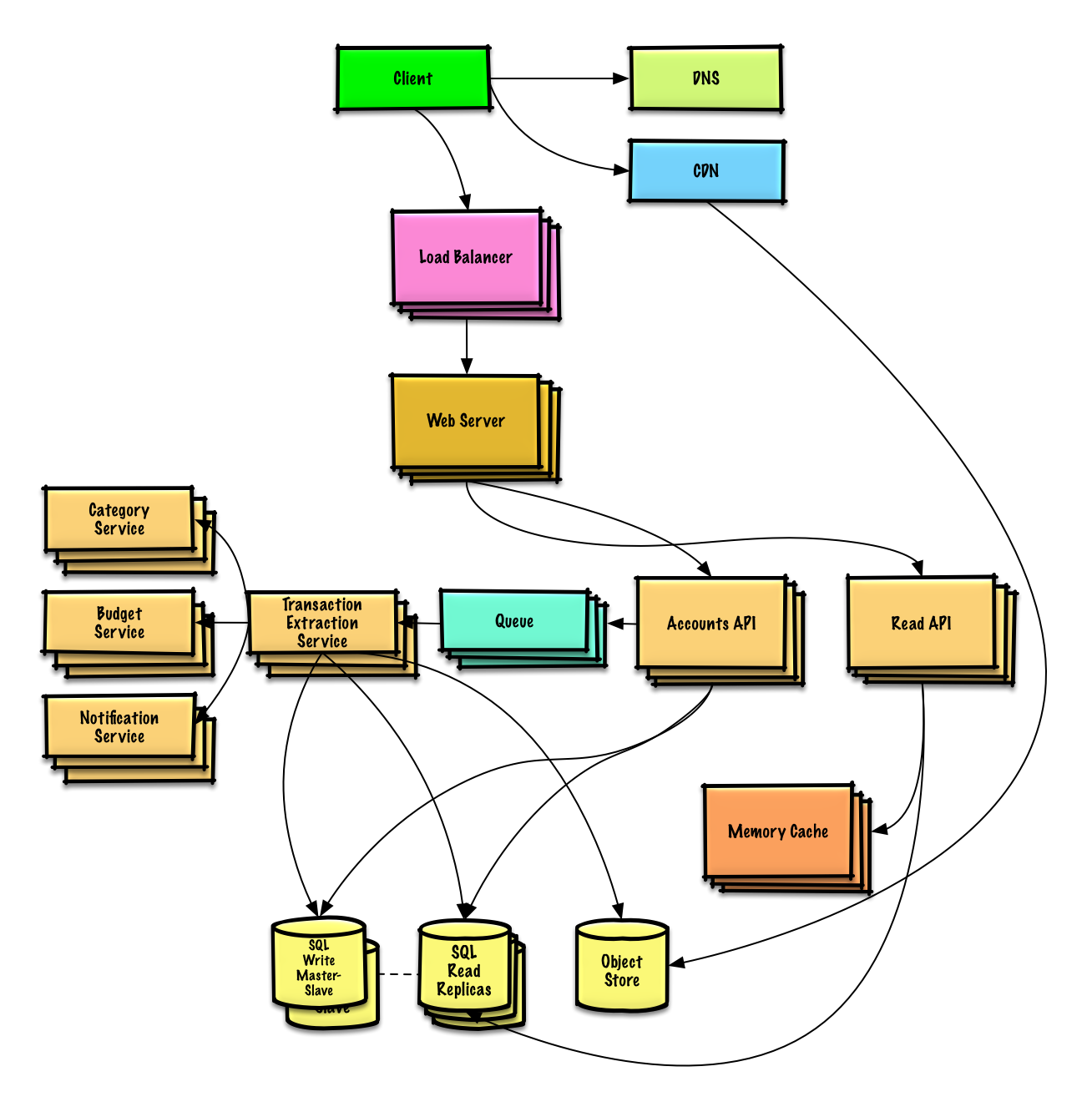
# OATH2



# Design Mint.com

# Imgur

# Design a key-value store for a search engine



# Design a key-value cache to save the results of the most recent web server queries

