FreemanZhang / **system-design**

Preparing for system design interview questions

#system-design-project #interview-questions

137 commits | **1** branch | **0** releases | **0** contributors

Branch: master | New pull request | Find file | Clone or download

freemanzhang restructure readme outline — Latest commit 39203c4 on Mar 9, 2017

linkedin	test intellij environment	2 years ago
src	add more threadsafe implementations	a year ago
README.md	restructure readme outline	a year ago
crawler.md	add scaling and init impl code	a year ago
databaseDesign.md	summarize lookup service	2 years ago
fileSystemDesign.md	summarize gfs lessons	2 years ago
locationBasedService.md	chunk knowledge up	2 years ago
messenger.md	add scaling messenger	2 years ago
monitorSystem.md	summarize monitoring system	2 years ago
multiThreading.md	summarize runnable interface vs thread class	a year ago
newsFeed.md	add scale newsfeed system part	2 years ago
paymentSystem.md	summarize payment system	a year ago
recommendationSystem.md	system design	2 years ago
system-design.iml	reword	2 years ago
tinyURL.md	add linkedin onsite design problem discussions	2 years ago
topk.md	reword	2 years ago
typeahead.md	scale typeahead.md	a year ago

README.md

Fight for 200 commits

System design

- The process of designing the architecture, components, modules, interfaces, and data for a system to satisfy specified requirements. Flexible, maintainable and scalable.

Typical system design workflow

What are the problem constraints

What's the amount of traffic the system should handle

- ***What's the number of users?***. Usually assume 200 million monthly active users / 100 million daily active user.
    - Monthly active user
    - Daily active user
- ***Let's suppose a user will do XXX operations and YYY operations per day***
    - Read read ratio
        - Read operations: 10 per day
        - Write operations: 3 per day
- ***The average QPS should be***

> Num of operation per day * Number of daily active users / 86400 (~100,000)

- ***Since the traffic load is not evenly distributed across the day. Let's assume that the peak QPS is 3 times the average QPS***.

> Peak QPS = Average QPS * 3

What's the amount of data the system should handle

- ***What's the amount of data we need to store in 5 years?***
    - ***A user will use XXX feature (write feature) YYY times per day***
    - ***The amount of DAU is XXX.***
    - ***In five years, the total amount of new data is***

> New data written per year: DAU * 365 (~400) * 5

What features the system needs to support

List features

- (Interviewee) ***First, let me list down all the features I could think of.***
- (Interviewee) ***Among all these use cases, these are the core features. I would like to focus on these core features first. If we have extra time, then we consider XXX features.***

Common features

- User system
  - Register / Login
  - Profile display / Edit
  - History view
- Friendship system
- User interface (Or only API is needed)
- Payment
- Search
- Notification (Email/SMS)
- Mobile / Desktop / Third party support

Abstract design

- Diagram of components of your designed system and its connections.
  - ***Let's draw a high-level module diagram for the system***.
    - Use rectangles for components
    - Use lines to connect them as communication traffic

Front-end layer

Application service layer

Data cache

- Redis / Memcached
- Cache key / Cache algorithm

Data storage

- Single DB / Master-slave, sharding

Message queue + notification center

- kafka

Logs + storage + analytics

- Kibana

Search

- ElasticSearch

Scale

- Replica
- Sharding
- Denormalization

System design evaluation standards

- Work solution 25%
- Special case 20%
- Analysis 25%
- Tradeoff 15%
- Knowledge base 15%

OO design principles

SRP: The Single Responsibility Principle

- Your classes should have one single responsibility and no more.
    - Take validation of an e-mail address as an example. If you place your validation logic directly in the code that creates user accounts, you will not be able to reuse it in a different context. Having validation logic separated into a distinct class would let you reuse it in multiple places and have only a single implementation.

OCP: The Open-Closed Principle

- Create code that does not have to be modified when requirements change or when new use cases arise. "Open for extension but closed for modification"
    - Requires you to break the problem into a set of smaller problems. Each of these tasks can then vary independently without affecting the reusability of remaining components.
    - MVC frameworks. You have the ability to extend the MVC components by adding new routes, intercepting requests, returning different responses, and overriding default behaviors.

LSP: The Liskov Substitution Principle

DIP: The Dependency-Inversion Principle

- Dependency injection provides references to objects that the class depends on instead of allowing the class to gather the dependencies itself. In practice, dependency injection can be summarized as not using the "new" keyword in your classes and demanding instances of your dependencies to be provided to your class by its clients.
- Dependency injection is an important principle and a subclass of a broader principle called inversion of control. Dependency injection is limited to object creation and assembly of its dependencies. Inversion of control, on the other hand, is a more generic idea and can be applied to different problems on different levels of abstraction.
    - IOC is heavily used by several frameworks such as Spring, Rails and even Java EE containers. Instead of you being in control of creating instances of your objects and invoking methods, you become the creator of plugins or extensions to the framework. The IOC framework will look at the web request and figure out which classes should be instantiated and which components should be delegated to. This means your classes do not have to know when their instances are created, who is using them, or how their dependencies are put together.

ISP: The Interface-Segregation Principle

DRY: Don't repeat yourself

- There are a number of reasons developers repeated waste time:
    - Following an inefficient process
    - Lack of automation
    - Reinventing the wheel

  - Copy/Paste programming

Distributed system concepts

CAP theorem

- If you get a network partition, you have to trade off consistency versus availability.
  - Consistency: Every read would get the most recent write.
  - Availability: Every request received by the nonfailing node in the system must result in a response.
  - Partition tolerance: The cluster can survive communication breakages in the cluster that separate the cluster into multiple partitions unable to communicate with each other.

Consistency

Update consistency

- Def: Write-write conflicts occur when two clients try to write the same data at the same time. Result is a lost update.
- Solutions:
  - Pessimistic approach: Preventing conflicts from occuring.
    - The most common way: Write locks. In order to change a value you need to acquire a lock, and the system ensures that only once client can get a lock at a time.
  - Optimistic approach: Let conflicts occur, but detects them and take actions to sort them out.
    - The most common way: Conditional update. Any client that does an update tests the value just before updating it to see if it is changed since his last read.
    - Save both updates and record that they are in conflict. This approach usually used in version control systems.
- Problems of the solution: Both pessimistic and optimistic approach rely on a consistent serialization of the updates. Within a single server, this is obvious. But if it is more than one server, such as with peer-to-peer replication, then two nodes might apply the update in a different order.
- Often, when people first encounter these issues, their reaction is to prefer pessimistic concurrency because they are determined to avoid conflicts. Concurrent programming involves a fundamental tradeoff between safety (avoiding errors such as update conflicts) and liveness (responding quickly to clients). Pessimistic approaches often severly degrade the responsiveness of a system to the degree that it becomes unfit for its purpose. This problem is made worse by the danger of errors such as deadlocks.

Read consistency

- Def:
  - Read-write conflicts occur when one client reads inconsistent data in the middle of another client's write.
- Types:
  - Logical consistency: Ensuring that different data items make sense together.
    - Example:
      - Martin begins update by modifying a line item
      - Pramod reads both records
      - Martin completes update by modifying shipping charge
  - Replication consistency: Ensuring that the same data item has the same value when read from different replicas.
    - Example:
      - There is one last hotel room for a desirable event. The reservation system runs onmany nodes.
      - Martin and Cindy are a couple considering this room, but they are discussing this on the phone because Martin is in London and Cindy is in Boston.
      - Meanwhile Pramod, who is in Mumbai, goes and books that last room.
      - That updates the replicated room availability, but the update gets to Boston quicker than it gets to

London.
    - When Martin and Cindy fire up their browsers to see if the room is available, Cindy sees it booked and Martin sees it free.
  - Read-your-write consistency (Session consistency): Once you have made an update, you're guaranteed to continue seeing that update. This can be difficult if the read and write happen on different nodes.
    - Solution1: A sticky session. a session that's tied to one node. A sticky session allows you to ensure that as long as you keep read-your-writes consistency on a node, you'll get it for sessions too. The downsides is that sticky sessions reduce the ability of the load balancer to do its job.
    - Solution2: Version stamps and ensure every interaction with the data store includes the latest version stamp seen by a session.

Replication Consistency

- Def: Slaves could return stale data.
- Reason:
  - Replication is usually asynchronous, and any change made on the master needs some time to replicate to its slaves. Depending on the replication lag, the delay between requests, and the speed of each server, you may get the freshest data or you may get stale data.
- Solution:
  - Send critical read requests to the master so that they would always return the most up-to-date data.
  - Cache the data that has been written on the client side so that you would not need to read the data you have just written.
  - Minize the replication lag to reduce the chance of stale data being read from stale slaves.

Message queue

Benefits

- **Enabling asynchronous processing**:
  - Defer processing of time-consuming tasks without blocking our clients. Anything that is slow or unpredictable is a candidate for asynchronous processing. Example include
    - Interact with remote servers
    - Low-value processing in the critical path
    - Resource intensive work
    - Independent processing of high- and low- priority jobs
  - Message queues enable your application to operate in an asynchronous way, but it only adds value if your application is not built in an asynchronous way to begin with. If you developed in an environment like Node.js, which is built with asynchronous processing at its core, you will not benefit from a message broker that much. What is good about message brokers is that they allow you to easily introduce asynchronous processing to other platforms, like those that are synchronous by nature (C, Java, Ruby)
- **Easier scalability**:
  - Producers and consumers can be scaled separately. We can add more producers at any time without overloading the system. Messages that cannot be consumed fast enough will just begin to line up in the message queue. We can also scale consumers separately, as now they can be hosted on separate machines and the number of consumers can grow independently of producers.
- **Decoupling**:
  - All that publishers need to know is the format of the message and where to publish it. Consumers can become oblivious as to who publishes messages and why. Consumers can focus solely on processing messages from the queue. Such a high level decoupling enables consumers and producers to be developed indepdently. They can even be developed by different teams using different technologies.
- **Evening out traffic spikes**:
  - You should be able to keep accepting requests at high rates even at times of icnreased traffic. Even if your publishing generates messages much faster than consumers can keep up with, you can keep enqueueing

messages, and publishers do not have to be affected by a temporary capacity problem on the consumer side.
- **Isolating failures and self-healing**:
    - The fact that consumers' availability does not affect producers allows us to stop message processing at any time. This means that we can perform maintainance and deployments on back-end servers at any time. We can simply restart, remove, or add servers without affecting producer's availability, which simplifies deployments and server management. Instead of breaking the entire application whenever a back-end server goes offline, all that we experience is reduced throughput, but there is no reduction of availability. Reduced throughput of asynchronous tasks is usually invisible to the user, so there is no consumer impact.

Components

- Message producer
    - Locate the message queue and send a valid message to it
- Message broker - where messages are sent and buffered for consumers.
    - Be available at all times for producers and to accept their messages.
    - Buffering messages and allowing consumers to consume related messages.
- Message consumer
    - Receive and process message from the message queue.
    - The two most common ways of implement consumers are a "cron-like" and a "daemon-like" approach.
        - Connects periodically to the queue and checks the status of the queue. If there are messages, it consumes them and stops when the queue is empty or after consuming a certain amount of messages. This model is common in scripting languages where you do not have a persistenly running application container, such as PHP, Ruby, or Perl. Cron-like is also referred to as a pull model because the consumers pulls messages from the queue. It can also be used if messages are added to the queue rarely or if network connectivity is unreliable. For example, a mobile application may try to pull the queue from time to time, assuming that connection may be lost at any point in time.
        - A daemon-like consumer runs constantly in an infinite loop, and it usually has a permanent connection to the message broker. Instead of checking the status of the queue periodically, it simply blocks on the socket read operation. This means that the consumer is waiting idly until messages are pushed by the message broker in the connection. This model is more common in languages with persistent application containers, such as Java, C#, and Node.js. This is also referred to as a push model because messages are pushed by the message broker onto the consumer as fast as the consumer can keep processing them.

Routing methods

- Direct worker queue method
    - Consumers and producers only have to know the name of the queue.
    - Well suited for the distribution of time-consuming tasks such as sending out e-mails, processing videos, resizing images, or uploading content to third-party web services.
- Publish/Subscribe method
    - Producers publish message to a topic, not a queue. Messages arriving to a topic are then cloned for each consumer that has a declared subscription to that topic.
- Custom routing rules
    - A consumer can decide in a more flexible way what messages should be routed to its queue.
    - Logging and alerting are good examples of custom routing based on pattern matching.

Protocols

- AMQP: A standardized protocol accepted by OASIS. Aims at enterprise integration and interoperability.
- STOMP: A minimalist protocol.
    - Simplicity is one of its main advantages. It supports fewer than a dozen operations, so implementation and debugging of libraries are much easier. It also means that the protocol layer does not add much performance overhead.
    - But interoperability can be limited because there is no standard way of doing certain things. A good example of

impaired is message prefetch count. Prefetch is a great way of increasing throughput because messages are received in batches instead of one message at a time. Although both RabbitMQ and ActiveMQ support this feature, they both implement it using different custom STOMP headers.

- JMS
    - A good feature set and is popular
    - Your ability to integrate with non-JVM-based languages will be very limited.

Metrics to decide which message broker to use

- Number of messages published per second
- Average message size
- Number of messages consumed per second (this can be much higher than publishing rate, as multiple consumers may be subscribed to receive copies of the same message)
- Number of concurrent publishers
- Number of concurrent consumers
- If message persistence is needed (no message loss during message broker crash)
- If message acknowledgement is need (no message loss during consumer crash)

Challenges

- No message ordering: Messages are processed in parallel and there is no synchronization between consumers. Each consumer works on a single message at a time and has no knowledge of other consumers running in parallel to it. Since your consumers are running in parallel and any of them can become slow or even crash at any point in time, it is difficult to prevent messages from being occasionally delivered out of order.
    - Solutions:
        - Limit the number of consumers to a single thread per queue
        - Build the system to assume that messages can arrive in random order
        - Use a messaging broker that supports partial message ordering guarantee.
    - It is best to depend on the message broker to deliver messages in the right order by using partial message guarantee (ActiveMQ) or topic partitioning (Kafka). If your broker does not support such functionality, you will need to ensure that your application can handle messages being processed in an unpredictable order.
        - Partial message ordering is a clever mechanism provided by ActiveMQ called message groups. Messages can be published with a special label called a message group ID. The group ID is defined by the application developer. Then all messages belonging to the same group are guaranteed to be consumed in the same order they were produced. Whenever a message with a new group ID gets published, the message broker maps the new group Id to one of the existing consumers. From then on, all the messages belonging to the same group are delivered to the same consumer. This may cause other consumers to wait idly without messages as the message broker routes messages based on the mapping rather than random distribution.
    - Message ordering is a serious issue to consider when architecting a message-based application, and RabbitMQ, ActiveMQ and Amazon SQS messaging platform cannot guarantee global message ordering with parallel workers. In fact, Amazon SQS is known for unpredictable ordering messages because their infrastructure is heavily distributed and ordering of messages is not supported.
- Message requeueing
    - By allowing messages to be delivered to your consumers more than once, you make your system more robust and reduce constraints put on the message queue and its workers. For this approach to work, you need to make all of your consumers idempotent.
        - But it is not an easy thing to do. Sending emails is, by nature, not an idempotent operation. Adding an extra layer of tracking and persistence could help, but it would add a lot of complexity and may not be able to handle all of the faiulres.
        - Idempotent consumers may be more sensitive to messages being processed out of order. If we have two messages, one to set the product's price to $55 and another one to set the price of the same product to $60, we could end up with different results based on their processing order.
- Race conditions become more likely

- Risk of increased complexity
    - When integrating applications using a message broker, you must be very diligent in documenting dependencies and the overarching message flow. Without good documentation of the message routes and visibility of how the message flow through the system, you may increase the complexity and make it much harder for developers to understand how the system works.

Networking

TCP vs UDP

TCP	UDP
Reliable: TCP is connection-oriented protocol. When a file or message send it will get delivered unless connections fails. If connection lost, the server will request the lost part. There is no corruption while transferring a message.	Not Reliable: UDP is connectionless protocol. When you a send a data or message, you don't know if it'll get there, it could get lost on the way. There may be corruption while transferring a message.
Ordered: If you send two messages along a connection, one after the other, you know the first message will get there first. You don't have to worry about data arriving in the wrong order.	Not Ordered: If you send two messages out, you don't know what order they'll arrive in i.e. no ordered
Heavyweight: – when the low level parts of the TCP "stream" arrive in the wrong order, resend requests have to be sent, and all the out of sequence parts have to be put back together, so requires a bit of work to piece together.	Lightweight: No ordering of messages, no tracking connections, etc. It's just fire and forget! This means it's a lot quicker, and the network card / OS have to do very little work to translate the data back from the packets.
Streaming: Data is read as a "stream," with nothing distinguishing where one packet ends and another begins. There may be multiple packets per read call.	Datagrams: Packets are sent individually and are guaranteed to be whole if they arrive. One packet per one read call.
Examples: World Wide Web (Apache TCP port 80), e-mail (SMTP TCP port 25 Postfix MTA), File Transfer Protocol (FTP port 21) and Secure Shell (OpenSSH port 22) etc.	Examples: Domain Name System (DNS UDP port 53), streaming media applications such as IPTV or movies, Voice over IP (VoIP), Trivial File Transfer Protocol (TFTP) and online multiplayer games

HTTP

Status code

Groups

Status code	Meaning	Examples
5XX	Server error	500 Server Error
4XX	Client error	401 Authentication failure; 403 Authorization failure; 404 Resource not found
3XX	Redirect	301 Resource moved permanently; 302 Resource moved temporarily
2XX	Success	200 OK; 201 Created; 203 Object marked for deletion

HTTP 4XX status codes

Status code	Meaning	Examples

400	Malformed request	Frequently a problem with parameter formatting or missing headers
401	Authentication error	The system doesn't know who the request if from. Authentication signature errors or invalid credentials can cause this
403	Authorization error	The system knows who you are but you don't have permission for the action you're requesting
404	Page not found	The resource doesn't exist
405	Method not allowed	Frequently a PUT when it needs a POST, or vice versa. Check the documentation carefully for the correct HTTP method

Verbs

CRUD example with Starbucks

Action	System call	HTTP verb address	Request body	Successful response code + Response body
Order iced team	Add order to system	Post /orders/	{"name" : "iced tea", "size" : "trenta"}	201 Created Location: /orders/1
Update order	Update existing order item	PUT /orders/1	{"name" : "iced tea", "size" : "trenta", "options" : ["extra ice", "unsweetened"]}	204 No Content or 200 Success
Check order	Read order from system	GET /orders/1		200 Success { "name" : "iced tea", "size" : "trenta", "options" : ["extra ice", "unsweetened"]}
Cancel order	Delete order from system	DELETE /orders/1		202 Item Marked for Deletion or 204 No Content

- What about actions that don't fit into the world of CRUD operations?
  - Restructure the action to appear like a field of a resource. This works if the action doesn't take parameters. For example an activate action could be mapped to a boolean activated field and updated via a PATCH to the resource.
  - Treat it like a sub-resource with RESTful principles. For example, GitHub's API lets you star a gist with PUT /gists/:id/star and unstar with DELETE /gists/:id/star.
  - Sometimes you really have no way to map the action to a sensible RESTful structure. For example, a multi-resource search doesn't really make sense to be applied to a specific resource's endpoint. In this case, /search would make the most sense even though it isn't a resource. This is OK - just do what's right from the perspective of the API consumer and make sure it's documented clearly to avoid confusion.

Others

- Put is a full update on the item. Patch is a delta update.
- Head: A lightweight version of GET
- Options: Discovery mechanism for HTTP
  - Link/Unlink: Removes the link between a story and its author

Headers

Request

Header	Example value	Meaning
Accept	Text/html, application/json	The client's preferred format for the response body. Browsers tend to prefer text/html, which is a human-friendly format. Applications using an API are likely to request JSON, which is structured in a machine-parseable way. This can be a list, and if so, the list is parsed in priority order: the first entry is the most desired format, all the way down to the last one.
Accept-language	en-US	The preferred written language for the response. This is most often used by browsers indicating the language the user has specified as a preference
User-agent	Mozilla/5.0	This header tells the server what kind of client is making the request. This is an important header because sometimes responses or JavaScript actions are performed differently for different browsers. This is used less frequently for this purpose by API clients, but it's a friendly practice to send a consistent user-agent for the server to use when determining how to send the information back.
Content-length	size of the content body	When sending a PUT or POST, this can be sent so the server can verify that the request body wasn't truncated on the way to the server.
Content-type	application/json	When a content body is sent, the client can indicate to the server what the format is for that content in order to help the server respond to the request correctly.

Response

Header	Example value	Meaning
Content-Type	application/json	As with the request, when the content body is sent back to the client, the Content-Type is generally set to help the client know how best to process the request. Note that this is tied somewhat indirectly to the Accept header sent by the client. The server will generally do its best to send the first type of content from the list sent by the client but may not always provide the first choice.
Access-Control-Allow-Headers	Content-Type, Authorization, Accept	This restricts the headers that a client can use for the request to a particular resource
Access-Control-Allow-Methods	GET, PUT, POST, DELETE, OPTIONS	What HTTP methods are allowed for this resource
Access-Control-Allow-Origin	* or http://www.example.com	This restricts the locations that can refer requests to the resource

Compression

- Accept-Encoding/Content-Encoding:
  - Condition: Content compression occurs only when a client advertises, wants to use it and a server indicates its willingness to enable it.
    - Clients indicate they want to use it by sending the Accept-Encoding header when making requests. The value of this header is a comma-separated list of compression methods that the client will accept. For example, Accept-Encoding: gzip, deflate.
    - If the server supports any of the compression methods that the client has advertised, it may deliver a

compressed version of the resource. It indicates that the content has been compressed with the Content-Encoding header in the response. For example, Content-Encoding: gzip. Content-Length header in the response indicates the size of the compressed content.

- Methods
    - identity: no compression.
    - compress: UNIX compress method, which is based on the Lempel-Ziv Welch (LZW) aglorithm
    - gzip: the most popular format.
    - deflate: just gzip without the checksum header.
- What to compress:
    - Usually applied to text-based content such as HTML, XML, CSS, and Javascript.
    - Not applied to binary data
        - Many of binary formats such as GIF, PNG, and JPEG already use compression.
- Disadvantages:
    - There is additional CPU usage at both the server side and client side.
    - There will always be a small percentage of clients that simply can't accept compressed content.

Parameters

- Parameters are frequently used in HTTP requests to filter responses or give additional information about the request. They're used most frequently with GET(read) operations to specify exactly what's wanted from the server. Parameters are added to the address. They're separated from the address with a question mark (?), and each key-value pair is separated by an equals sign (=); pairs are separated from each other using the ampersand.

Action	System call	HTTP verb address	Successful response code / Response body
Get order list, only Trenta iced teas	Retrieve list with a filter	Get /orders?name=iced%20tea&size=trenta	[{ "id" : 1, "name" : "iced tea", "size" : "trenta", "options" : ["extra ice", "unsweetened"] }]
Get options and size for the order	Retrieve order with a filter specifying which pieces to return	Get /orders/1?fields=options,size	{ "size" : "trenta", "options" : ["extra ice", "unsweetened"]}

HTTP session

Stateless applications

- Web application servers are generally "stateless":
    - Each HTTP request is independent; server can't tell if 2 requests came from the same browser or user.
    - Web server applications maintain no information in memory from request to request (only information on disk survives from one request to another).
- Statelessness not always convenient for application developers: need to tie together a series of requests from the same user. Since the HTTP protocol is stateless itself, web applications developed techniques to create a concept of a session on top of HTTP so that servers could recognize multiple requests from the same user as parts of a more complex and longer lasting sequence.

Structure of a session

- The session is a key-value pair data structure. Think of it as a hashtable where each user gets a hashkey to put their data in. This hashkey would be the “session id”.

Server-side session vs client-side cookie

Category	Session	Cookie

Location	User ID on server	User ID on web browser
Safeness	Safer because data cannot be viewed or edited by the client	A hacker could manipulate cookie data and attack
Amount of data	Big	Limited
Efficiency	Save bandwidth by passing only a reference to the session (sessionID) each pageload.	Must pass all data to the webserver each pageload
Scalability	Need efforts to scale because requests depend on server state	Easier to implement

Store session state in client-side cookies

Cookie Def

- Cookies are key/value pairs used by websites to store state informations on the browser. Say you have a website (example.com), when the browser requests a webpage the website can send cookies to store informations on the browser.

Cookie typical workflow

```
// Browser request example:

GET /index.html HTTP/1.1
Host: www.example.com

// Example answer from the server:


HTTP/1.1 200 OK
Content-type: text/html
Set-Cookie: foo=10
Set-Cookie: bar=20; Expires=Fri, 30 Sep 2011 11:48:00 GMT
... rest  of the response

// Here two cookies foo=10 and bar=20 are stored on the browser. The second one will expire on 30
September. In each subsequent request the browser will send the cookies back to the server.


GET /spec.html HTTP/1.1
Host: www.example.com
Cookie: foo=10; bar=20
Accept: */*
```

Cookie Pros and cons

- Advantage: You do not have to store the sesion state anywhere in your data center. The entire session state is being handed to your web server with every web request, thus making your application stateless in the context of the HTTP session.
- Disadvantage: Session storage can becomes expensive. Cookies are sent by the browser with every single request, regardless of the type of resource being requested. As a result, all requests within the same cookie domain will have session storage appended as part of the request.
- Use case: When you can keep your data minimal. If all you need to keep in session scope is userID or some security token, you will benefit from the simplicity and speed of this solution. Unfortunately, if you are not careful, adding more data to the session scope can quickly grow into kilobytes, making web requests much slower, especially on mobile devices. The coxt of cookie-based session storage is also amplified by the fact that encrypting serialized data and then Based64 encoding increases the overall byte count by one third, so that 1KB of session scope data becomes 1.3KB of additional data transferred with each web request and web response.

Store session state in server-side

- Approaches:
  - Keep state in main memory
  - Store session state in files on disk
  - Store session state in a database
    - Delegate the session storage to an external data store: Your web application would take the session identifier from the web request and then load session data from an external data store. At the end of the web request life cycle, just before a response is sent back to the user, the application would serialize the session data and save it back in the data store. In this model, the web server does not hold any of the session data between web requests, which makes it stateless in the context of an HTTP session.
    - Many data stores are suitable for this use case, for example, Memcached, Redis, DynamoDB, or Cassandra. The only requirement here is to have very low latency on get-by-key and put-by-key operations. It is best if your data store provides automatic scalability, but even if you had to do data partitioning yourself in the application layer, it is not a problem, as sessions can be partitioned by the session ID itself.

Typical server-side session workflow

1. Every time an internet user visits a specific website, a new session ID (a unique number that a web site's server assigns a specific user for the duration of that user's visit) is generated. And an entry is created inside server's session table

Columns	Type	Meaning
sessionID	string	a global unique hash value
userId	Foreign key	pointing to user table
expireAt	timestamp	when does the session expires

2. Server returns the sessionID as a cookie header to client
3. Browser sets its cookie with the sessionID
4. Each time the user sends a request to the server. The cookie for that domain will be automatically attached.
5. The server validates the sessionID inside the request. If it is valid, then the user has logged in before.

Use a load balancer that supports sticky sessions:

- The load balancer needs to be able to inspect the headers of the request to make sure that requests with the same session cookie always go to the server that initially the cookie.
- But sticky sessions break the fundamental principle of statelessness, and I recommend avoiding them. Once you allow your web servers to be unique, by storing any local state, you lose flexibility. You will not be able to restart, decommission, or safely auto-scale web servers without braking user's session because their session data will be bound to a single physical machine.

DNS

- Resolve domain name to IP address

Design

Initial design

- A simple design for DNS would have one DNS server that contains all the mappings. But the problems with a centralized design include:
  - **A single point of failure**: If the DNS server crashes, so does the entire Internet.
  - **Traffic volume**: A single DNS server would have to handle all DNS queries.
  - **Distant centralized database**: A single DNS server cannot be close to all the querying clients.

- **Maintenance**: The single DNS server would have to keep records for all Internet hosts. It needed to be updated frequently

A distributed, hierarchical database

- **Root DNS servers**:
- **Top-level domain servers**: Responsible for top level domains such as com, org, net, edu, and gov, and all of the country top-level domains such as uk, fr, ca, and jp.
- **Authoritative DNS servers**:
- **Local DNS server**: Each ISP - such as a university, an academic department, an employee's company, or a residential ISP - has a local DNS server. When a host connects to an ISP, the ISP provides the host with the IP addresses of one of its local DNS servers.

Internals

DNS records

- The DNS servers store source records (RRs). A resource record is a four-tuple that contains the following fields: (Name, Value, Type, TTL)
- There are the following four types of records
  - If Type=A, then Name is a hostname and Value is the IP address for the hostname. Thus, a Type A record provides the standard hostname-to-IP address mapping. For example, (relay1.bar.foo.com, 145.37.93.126, A) is a Type A record
  - If Type=NS, then Name is a domain and Value is the hostname of an authoritative DNS server that knows how to obtain the IP addresses for hosts in the domain. This record is used to route DNS queries further along in the query chain. As an example, (foo.com, dns.foo.com, NS) is a Type NS record.
  - If Type=CNAME, then Value is a canonical hostname for the alias hostname Name.
  - If Type=MX, then Value is the canonical name of a mail server that has an alias hostname Name.

Insert records into DNS DB

- Take domain name networkutopia.com as an example.
- First you need to register the domain name network. A registrar is a commercial entity that verifies the uniqueness of the domain name, enters the domain name into the DNS database and collects a small fee for its services.
  - When you register, you need to provide the registrar with the names and IP addresses of your primary and secondary authoritative DNS servers. For each of these two authoritative DNS servers, the registrar would then make sure that a Type NS and a Type A record are entered into the TLD com servers.

DNS query parsing

- When a user enters a URL into the browser's address bar, the first step is for the browser to resolve the hostname (http://www.amazon.com/index.html) to an IP address. The browser extracts the host name www.amazon.com from the URL and delegates the resolving task to the operating system. At this stage, the operating system has a couple of choices.
- It can either resolve the address using a static hosts file (such as /etc/hosts on Linux)
- It then query a local DNS server.
  - The local DNS server forwards to a root DNS server. The root DNS server takes not of the com suffix and returns a list of IP addresss for TLD servers responsible for com domain
  - The local DNS server then resends the query to one of the TLD servers. The TLD server takes note of www.amazon. suffix and respond with the IP address of the authoritative DNS server for amazon.
  - Finally, the local DNS server resends the query message directly to authoritative DNS which responds with the IP address of www.amazon.com.
- Once the browser receives the IP addresses from DNS, it can initiate a TCP connection to the HTTP server process located at port 80 at that IP address.

Types

Round-robin DNS

- A DNS server feature that allowing you to resolve a single domain name to one of many IP addresses.

GeoDNS

- A DNS service that allows domain names to be resolved to IP addresses based on the location of the customer. A client connecting from Europe may get a different IP address than the client connecting from Australia. The goal is to direct the customer to the closest data center to minimize network latency.

Functionality

DNS Caching

- Types:
  - Whenever the client issues a request to an ISP's resolver, the resolver caches the response for a short period (TTL, set by the authoritative name server), and subsequent queries for this hostname can be answered directly from the cache.
  - All major browsers also implement their own DNS cache, which removes the need for the browser to ask the operating system to resolve. Because this isn't particularly faster than quuerying the operating system's cache, the primary motivation here is better control over what is cached and for how long.
- Performance:
  - DNS look-up times can vary dramatically - anything from a few milliseconds to perhaps one-half a second if a remote name server must be queried. This manifests itself mostly as a slight delay when the user first loads the site. On subsequent views, the DNS query is answered from a cache.

Load balancing

- DNS can be used to perform load distribution among replicated servers, such as replicated web servers. For replicated web servers, a set of IP addresses is thus associated with one canonical hostname. The DNS database contains this set of IP addresses. When clients make a DNS query for a name mapped to a set of addresses, the server responds with the entire set of IP addresses, but rotates the ordering of the addresses within each reply. Because a client typically sends its HTTP request to the IP address that is the first in the set, DNS rotation distributes the traffic among the replicated servers.

Host alias

- A host with a complicated hostname can have one or more alias names. For example, a hostname such as relay1.west-coast.enterprise.com could have two aliases such as enterprise.com and www.enterprise.

DNS prefetching

Def

- Performing DNS lookups on URLs linked to in the HTML document, in anticipation that the user may eventually click one of these links. Typically, a single UDP packet can carry the question, and a second UDP packet can carry the answer.

Control prefetching

- Most browsers support a link tag with the nonstandard rel="dns-prefetch" attribute. This causes teh browser to prefetch the given hostname and can be used to precache such redirect linnks. For example

- In addition, site owners can disable or enable prefetching through the use of a special HTTP header like:

> X-DNS-Prefetch-Control: off

Load balancers

Benefits

- Decoupling
    - Hidden server maintenance. You can take a web server out of the load balancer pool, wait for all active connections to drain, and then safely shutdown the web server without affecting even a single client. You can use this method to perform rolling updates and deploy new software across the cluster without any downtime.
    - Seamlessly increase capacity. You can add more web servers at any time without your client ever realizing it. As soon as you add a new server, it can start receiving connections.
    - Automated scaling. If you are on cloud-based hosting with the ability to configure auto-scaling (like Amazon, Open Stack, or Rackspace), you can add and remove web servers throughout the day to best adapt to the traffic.
- Security
    - SSL termination: By making load balancer the termination point, the load balancers can inspect the contents of the HTTPS packets. This allows enhanced firewalling and means that you can balance requests based on teh contents of the packets.
    - Filter out unwanted requests or limit them to authenticated users only because all requests to back-end servers must first go past the balancer.
    - Protect against SYN floods (DoS attacks) because they pass traffic only on to a back-end server after a full TCP connection has been set up with the client.

Round-robin algorithm

- Def: 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.
- Problems:
    - Not all requests have an equal performance cost on the server. But a request for a static resource will be several orders of magnitude less resource-intensive than a requst for a dynamic resource.
    - Not all servers have identical processing power. Need to query back-end server to discover memory and CPU usage, server load, and perhaps even network latency.
    - How to support sticky sessions: Hashing based on network address might help but is not a reliable option. Or the load balancer could maintain a lookup table mapping session ID to server.

Security

SSL

Definition

- Hyper Text Transfer Protocol Secure (HTTPS) is the secure version of HTTP, the protocol over which data is sent between your browser and the website that you are connected to. The 'S' at the end of HTTPS stands for 'Secure'. It means all communications between your browser and the website are encrypted. HTTPS is often used to protect highly confidential online transactions like online banking and online shopping order forms.

How does HTTPS work

- HTTPS pages typically use one of two secure protocols to encrypt communications - SSL (Secure Sockets Layer) or TLS (Transport Layer Security). Both the TLS and SSL protocols use what is known as an 'asymmetric' Public Key Infrastructure (PKI) system. An asymmetric system uses two 'keys' to encrypt communications, a 'public' key and a 'private' key. Anything encrypted with the public key can only be decrypted by the private key and vice-versa.
- As the names suggest, the 'private' key should be kept strictly protected and should only be accessible the owner of the private key. In the case of a website, the private key remains securely ensconced on the web server. Conversely, the public key is intended to be distributed to anybody and everybody that needs to be able to decrypt information that was encrypted with the private key.

How to avoid public key being modified?

- Put public key inside digital certificate.
  - When you request a HTTPS connection to a webpage, the website will initially send its SSL certificate to your browser. This certificate contains the public key needed to begin the secure session. Based on this initial exchange, your browser and the website then initiate the 'SSL handshake'. The SSL handshake involves the generation of shared secrets to establish a uniquely secure connection between yourself and the website.
  - When a trusted SSL Digital Certificate is used during a HTTPS connection, users will see a padlock icon in the browser address bar. When an Extended Validation Certificate is installed on a web site, the address bar will turn green.

How to avoid computation consumption from PKI

- Only use PKI to generate session key and use the session key for further communications.

NoSQL

NoSQL vs SQL

- There is no generally accepted definition. All we can do is discuss some common characteristics of the databases that tend to be called "NoSQL".

Database	SQL	NoSQL
Data uniformness	Uniform data. Best visualized as a set of tables. Each table has rows, with each row representing an entity of interest. Each row is described through columns. One row cannot be nested inside another.	Non-uniform data. NoSQL databases recognize that often, it is common to operate on data in units that have a more complex structure than a set of rows. This is particularly useful in dealing with nonuniform data and custom fields. NoSQL data model can be put into four categories: key-value, document, column-family and graph.
Schema change	Define what table exists, what column exists, what data types are. Although actually relational schemas can be changed at any time with standard SQL commands, it is of hight cost.	Changing schema is casual and of low cost. Essentially, a schemaless database shifts the schema into the application code.
Query flexibility	Low cost on changing query. It allows you to easily look at the data in different ways. Standard SQL supports things like joins and subqueries.	High cost in changing query. It does not allow you to easily look at the data in different ways. NoSQL databases do not have the flexibility of joins or subqueries.
Transactions	SQL has ACID transactions (Atomic, Consistent, Isolated, and Durable). It allows you to manipulate any combination of rows from any tables in a single transaction. This operation either succeeds or fails entirely, and concurrent operations are isolated from each other so they cannot see a partial update.	Graph database supports ACID transactions. Aggregate-oriented databases do not have ACID transactions that span multiple aggregates. Instead, they support atomic manipulation of a single aggregate at a time. If we need to manipulate multiple aggregates in an atomic way, we have to manage that ourselves in application code. An aggregate structure may help with some data interactions but be an obstacle for others.
Consistency	Strong consistency	Trade consistency for availability or partition tolerance. Eventual consistency
	elational database use ACID	Aggregate structure helps greatly with running on a cluster. It we are running on a cluster, we need to minize

Scalability	transactions to handle consistency across the whole database. This inherently clashes with a cluster environment	how many nodes we need to query when we are gathering data. By using aggregates, we give the database important information about which bits of data (an aggregate) will be manipulated together, and thus should live on the same node.
Performance	MySQL/PosgreSQL ~ 1k QPS	MongoDB/Cassandra ~ 10k QPS. Redis/Memcached ~ 100k ~ 1M QPS
Maturity	Over 20 years. Integrate naturally with most web frameworks. For example, Active Record inside Ruby on Rails	Usually less than 10 years. Not great support for serialization and secondary index

NoSQL flavors

Key-value

- Suitable use cases
  - **Storing session information**: Generally, every web session is unique and is assigned a unique sessionid value. Applications that store the sessionid on disk or in a RDBMS will greatly benefit from moving to a key-value store, since everything about the session can be stored by a single PUT request or retrieved using GET. This single-request operation makes it very fast, as everything about the session is stored in a single object. Solutions such as Memcached are used by many web applications, and Riak can be used when availability is important
  - **User profiles, Preferences**: Almost every user has a unique userId, username, or some other attributes, as well as preferences such as language, color, timezone, which products the user has access to, and so on. This can all be put into an object, so getting preferences of a user takes a single GET operation. Similarly, product profiles can be stored.
  - **Shopping Cart Data**: E-commerce websites have shopping carts tied to the user. As we want the shopping carts to be available all the time, across browsers, machines, and sessions, all the shopping information can be put into value where the key is the userid. A riak cluster would be best suited for these kinds of applications.
- When not to use
  - **Relationships among Data**: If you need to have relationships between different sets of data, or correlate teh data between different sets of key, key-value stores are not the best solution to use, even though some key-value stores provide link-walking features.
  - **Multioperation transactions**: If you're saving multiple keys and there is a failure to save any of them, and you want to revert or roll back the rest of the operations, key-value stores are not the best solution to be used.
  - **Query by data**: If you need to search the keys based on something found in the value part of the key-value pairs, then key-value stores are not going to perform well for you. This is no way to inspect the value on the database side, with the exception of some products like Riak Search or indexing engines like Lucene.
  - **Operations by sets**: Since operations are limited to one key at a time, there is no way to operate upon multiple keys at the same time. If you need to operate upon multiple keys, you have to handle this from the client side.

Document

- Suitable use cases
  - **Event logging**: Applications have different event logging needs; within the enterprise, there are many different applications that want to log events. Document databases can store all these different types of events and can act as a central data store for event storage. This is especially true when the type of data being captured by the events keeps changing. Events can be sharded by the name of the application where the event originated or by the type of event such as order_processed or customer_logged.
  - **Content Management Systems, Blogging Platforms**: Since document databases have no predefined schemas and usually uderstand JSON documents, they work well in content management systems or applications for

publishing websites, managing user comments, user registrations, profiles, web-facing documents.
  - **Web Analytics or Real-Time Analytics**: Document databases can store data for real-time analytics; since parts of the document can be updated, it's very easy to store page views or unique visitors, and new metrics can be easily added without schema changes.
  - **E-Commerce Applications**: E-commerce applications often need to have flexible schema for products and orders, as well as the ability to evolve their data models without expensive database refactoring or data migration.

- When not to use

  - **Complex Transactions Spanning Different Operations**: If you need to have atomic cross-document operations, then document databases may not be for you. However, there are some document databases that do support these kinds of operations, such as RavenDB.
  - **Queries against Varying Aggregate Structure**: Flexible schema means that the database does not enforce any restrictions on the schema. Data is saved in the form of application entities. If you need to query these entities ad hoc, your queries will be changing (in RDBMS terms, this would mean that as you join criteria between tables, the tables to join keep changing). Since the data is saved as an aggregate, if the design of the aggregate is constantly changing, you need to save the aggregates at the lowest level of granularity-basically, you need to normalize the data. In this scenario, document databases may not work.

Column-Family

- Suitable use cases
  - **Event Logging**: Column-family databases with their ability to store any data structures are a great choice to store event information, such as application state or errors encountered by the application. Within the enterprise, all applications can write their events to Cassandra with their own columns and the row key of the form appname:timestamp. Since we can scale writes, Cassandra would work ideally for an event logging system.
  - **Content Management Systems, Blogging Platforms**: Using column-families, you can store blog entries with tags, categories, links, and trackbacks in different columns. Comments can be either stored in the same row or moved to a different keyspace; similarly, blog users and the actual blogs can be put into different column families.
  - **Counters**: Often, in web applications you need to count and categorize visitors of a page to calculate analytics, you can use the CounterColumnType during creation of a column family.
  - **Expiring usage**: You may provide demo to users, or may want to show ad banners on a website for a specific time. You can do this by using expiring columns: Cassandra allows you to have columns which, after a given time, are deleted automatically. This time is known as TTL and is defined in seconds. The column is deleted after the TTL has elapsed; when the column does not exist, the access can be revoked or the banner can be removed.

```
CREATE COLUMN FAMILY visit_counter
WITH default_validation_class=CounterColumnType
AND key_validation_class=UTF8Type AND comparator=UTF8Type

// Once a column family is created, you can have arbitrary columns for each page visited within the web
INCR visit_counter['mfowler'][home] BY 1;
INCR visit_counter['mfowler'][products] BY 1;
INCR visit_counter['mfowler'][contactus] BY 1;

// expiring columns
SET Customer['mfowler']['demo_access'] = 'allowed' WITH ttl=2592000;
```

- When not to use
  - **ACID transactions for writes and reads**
  - **Database to aggregate the data using queries (such as SUM or AVG)**: you have to do this on the client side using data retrieved by the client from all the rows.
  - **Early prototypes or initial tech spikes**: During the early stages, we are not sure how the query patterns may change, and as the query patterns change, we have to change the column family design. This causes friction for the product innovation team and slows down developer productivity. RDBMS impose high cost on schema change, which is traded off for a low cost of query change; in Cassandra, the cost may be higher for query

change as compared to schema change.

Graph

* Suitable use cases
    * **Connected data**:
        * Social networks are where graph databases can be deployed and used very effectively. These social graphs don't have to be only of the friend kind; for example, they can represent employees, their knowledge, and where they worked with other employees on different projects. Any link-rich domain is well-suited for graph databases.
        * If you have relationships between domain entities from different domains (such as social, spatial, commerce) in a single database, you can make these relationships more valuable by providing the ability to traverse across domains.
    * **Routing, Dispatch, and Location-Based Services**: Every location or address that has a delivery is node, and all the nodes where the delivery has to be made by the delivery person can be modeled as a graph nodes. Relationships between nodes can have the property of distance, thus allowing you to deliver the goods in an efficient manner. Distance and location properties can also be used in graphs of places of interest, so that your application can provide recommendations of good restaurants or entertainment options nearby. You can also create nodes for your points of sales, such as bookstores or restaurants, and notify the users when they are close to any of the nodes to provide location-based services.
    * **Recommendation Engines**:
        * As nodes and relationships are created in the system, they can be used to make recommendations like "your friends also bought this product" or "when invoicing this item, these other items are usually invoiced." Or, it can be used to make recommendations to travelers mentioning that when other visitors come to Barcelona they usually visit Antonio Gaudi's creations.
        * An interesting side effect of using the graph databases for recommendations is that as the data size grows, the number of nodes and relationships available to make the recommendations quickly increases. The same data can also be used to mine information-for example, which products are always bought together, or which items are always invoiced together; alerts can be raised when these conditions are not met. Like other recommendation engines, graph databases can be used to search for patterns in relationships to detect fraud in transactions.
* When not to use
    * When you want to update all or a subset of entities - for example, in an analytics solution where all entities may need to be updated with a changed property - graph databases may not be optimal since changing a peroperty on all the nodes is not a straight-forward operation. Even if the data model works for the problem domain, some databases may be unable to handle lots of data, especially in global graph operations.

Scaling

Functional partitioning

REST best practices

* Could look at industrial level api design example by Github

Consistency

Endpoint naming conventions

* Use all lowercase, hyphenated endpoints such as /api/verification-tokens. This increases URL "hackability", which is the ability to manually go in and modify the URL by hand. You can pick any naming scheme you like, as long as you're consistent about it.

- Use a noun or two to describe the resource, such as users, products, or verification-tokens.
- Always describe resources in plural: /api/users rather than /api/user. This makes the API more semantic.
    - Collection resource: /users
    - Instance resource: /users/007

HTTP verbs and CRUD consistency

- Use HTTP verbs for CRUD operations (Create/Read/Update/Delete).
    - Updates & creation should return a resource representation
        - A PUT, POST or PATCH call may make modifications to fields of the underlying resource that weren't part of the provided parameters (for example: created_at or updated_at timestamps). To prevent an API consumer from having to hit the API again for an updated representation, have the API return the updated (or created) representation as part of the response.
        - In case of a POST that resulted in a creation, use a HTTP 201 status code and include a Location header that points to the URL of the new resource.

Verb	Endpoint	Description
GET	/products	Gets a list of products
GET	/products/:id	Gets a single product by ID
GET	/products/:id/parts	Gets a list of parts in a single product
PUT	/products/:id/parts	Inserts a new part for a particular product
DELETE	/products/:id	Deletes a single product by ID
PUT	/products	Inserts a new product
HEAD	/products/:id	Returns whether the product exists through a status code of 200 or 404
PATCH	/products/:id	Edits an existing product by ID
POST	/authentication/login	Most other API methods should use POST requests

Versioning

- **What is versioning?** In traditional API scenarios, versioning is useful because it allows you to commit breaking changes to your service without demolishing the interaction with existing consumers.
- **Whether you need versioning?** Unless your team and your application are small enough that both live in the same repository and developers touch on both indistinctly, go for the safe bet and use versions in your API.
    - Is the API public facing as well? In this case, versioning is necessary, baking a bit more predictability into your service's behavior.
    - Is teh API used by several applications?   Are the API and the front end developed by separated teams? Is there a drawn-out process to change an API point? If any of these cases apply, you're probably better off versioning your API.
- **How to implement versioning?** There are two popular ways to do it:
    - The API version should be set in HTTP headers, and that if a version isn't specified in the request, you should get a response from the latest version of the API. But it can lead to breaking changes inadvertently.
    - The API version should be embedded into the URL. This identifies right away which version of the API your application wants by looking at the requested endpoint. An API version should be included in the URL to ensure browser explorability.

Data transfer format

- **Request**: You should decide on a consistent data-transfer strategy to upload the data to the server when making PUT, PATCH, or POST requests that modify a resource in the server. Nowadays, JSON is used almost ubiquitously as the data transport of choice due to its simplicity, the fact that it's native to browsers, and the high availability of JSON parsing libraries across server-side languages.

- **Response**:
  - Responses should conform to a consistent data-transfer format, so you have no surprises when parsing the response. Even when an error occurs on the server side, the response is still expected to be valid according to the chosen transport; For example, if your API is built using JSON, then all the responses produced by our API should be valid JSON.
  - You should figure out the envelope in which you'll wrap your responses. An envelope, or message wrapper, is crucial for providing a consistent experience across all your API endpoints, allowing consumers to make certain assumptions about the responses the API provides. A useful starting point may be an object with a single field, named data, that contains the body of your response.

```
{
        "data" : {}      // actual response
}
```

HTTP status codes and error handling

- Choose the right status codes for the problems your server is encountering so that the client knows what to do, but even more important is to make sure the error messages that are coming back are clear.
  - An authentication error can happen because the wrong keys are used, because the signature is generated incorrectly, or because it's passed to the server in the wrong way. The more information you can give to developers about how and why the command failed, the more likely they'll be able to figure out how to solve the problem.
- When you respond with status codes in the 2XX Success class, the response body should contain all of the relevant data that was requested. Here's an example showing the response to a request on a product that could be found, alongside with the HTTP version and status code:

```
HTTP/1.1 200 OK
{
        "data": {
                "id" : "baeb-b001",
                "name" : "Angry Pirate Plush Toy",
                "description" : "Batteries not included",
                "price" : "$39.99",
                "categories": ["plushies", "kids"]
        }
}
```

- If the request is most likely failed due to an error made by the client side (the user wasn't properly authenticated, for instance), you should use 4XX Client Error codes. If the request is most likely failed due to a server side error, then you should use 5XX error codes. In these cases, you should use the error field to describe why the request was faulty.

```
// if input validation fails on a form while attempting to create a product, you could return a response
HTTP/1.1 400 Bad Request
{
        "error": {
                "code": "bf-400",
                "message": "Some required fields were invalid.",
                "context": {
                        "validation": [
                                "The product name must be 6-20 alphanumeric characters",
                                "The price cann't be negative",
                                "At least one product category should be selected. "
                        ]
                }
        }
}

// server side error
{
        "error": {
                "code": "bf-500",
                "message": "An unexpected error occurred while accessing the database",
                "context": {
                        "id": "baeb-b001"
                }
        }
}
```

Paging

- Suppose a user makes a query to your API for /api/products. How many products should that end point return? You could set a default pagination limit across the API and have the ability to override that default for each individual endpoint. Within a reasonable range, the consumer should have the ability to pass in a query string parameter and choose a different limit.
    - Using Github paging API as an example, requests that return multiple items will be paginated to 30 items by default. You can specify further pages with the ?page parameter. For some resources, you can also set a custom page size up to 100 with the ?per_page parameter. Note that for technical reasons not all endpoints respect the ?per_page parameter, see events for example. Note that page numbering is 1-based and that omitting the ?page parameter will return the first page.

```
curl 'https://api.github.com/user/repos?page=2&per_page=100'
```

- Common parameters
    - page and per_page. Intuitive for many use cases. Links to "page 2" may not always contain the same data.
    - offset and limit. This standard comes from the SQL database world, and is a good option when you need stable permalinks to result sets.
    - since and limit. Get everything "since" some ID or timestamp. Useful when it's a priority to let clients efficiently stay "in sync" with data. Generally requires result set order to be very stable.
- Metadata
    - Include enough metadata so that clients can calculate how much data there is, and how and whether to fetch the next set of results. Examples of how that might be implemented:

```
{
  "results": [ ... actual results ... ],
  "pagination": {
    "count": 2340,
    "page": 4,
    "per_page": 20
  }
}
```

- Link header
    - The pagination info is included in the Link header. It is important to follow these Link header values instead of constructing your own URLs. In some instances, such as in the Commits API, pagination is based on SHA1 and not on page number.

```
Link: <https://api.github.com/user/repos?page=3&per_page=100>; rel="next",
  <https://api.github.com/user/repos?page=50&per_page=100>; rel="last"
```

- Rel attribute
    - describes the relationship between the requested page and the linked page

Name	Description

next	The link relation for the immediate next page of results.
last	The link relation for the last page of results.
first	The link relation for the first page of results.
prev	The link relation for the immediate previous page of results.

- Cases exist where data flows too rapidly for traditional paging methods to behave as expected. For instance, if a few records make their way into the database between requests for the first page and the second one, the second page results in duplicates of items that were on page one but were pushed to the second page as a result of the inserts. This issue has two solutions:
  - The first is to use identifiers instead of page numbers. This allows the API to figure out where you left off, and even if new records get inserted, you'll still get the next page in the context of the last range of identifiers that the API gave you.
  - The second is to give tokens to the consumer that allow the API to track the position they arrived at after the last request and what the next page should look like.

Scaling REST web services

Keeping service machine stateless

Benefits

- You can distribute traffic among your web service machines on a per-request basis. You can deploy a load balancer between your web services and their clients, and each request can be sent to any of the available web service machines. Being able to distribute requests in a round-robin fashion allows for better load distributionn and more flexibility.
- Since each web service request can be served by any of the web service machines, you can take service machines out of the load balancer pool as soon as they crash. Most of the modern load balancers support heartbeat checks to make sure that web services machines serving the traffic are available. As soon as a machine crashes or experiences some other type of failure, the load balancer will remove that host from the load-balancing pool, reducing the capacity of the cluster, but preventing clients from timing out or failing to get responses.
- You can restart and decommission servers at any point in time without worrying about affecting your clients. For example, if you want to shut down a server for maintenance, you need to take that machine out of the load balancer pool. Most load balancers support graceful removal of hosts, so new connections from clients are not sent to that server any more, but existing connections are not terminated to prevent client-side errors. After removing the host from the pool, you need to wait for all of your open connections to be closed by your clients, which can take a minute or two, and then you can safely shut down the machine without affecting even a single web service request.
- You will be able to perform zero-downtime updates of your web services. You can roll out your changes to one server at a time by taking it out of rotation, upgrading, and then putting it back into rotation. If your software does not allow you to run two different versions at the same time, you can deploy to an alternative stack and switch all of the traffic at once on the load balancer level.
- By removing all of the application state from your web services, you will be able to scale your web services layer by simply adding more clones. All you need to do is adding more machines to the load balancer pool to be able to support more concurrent connections, perform more network I/O, and compute more responses.

Common use cases needing share state

- The first use case is related to security, as your web service is likely going to require clients to pass some authentication token with each web service request. The token will have to be validated on the web service side, and client permissions will have to be evaluated in some way to make sure that the user has access to the operation they are attempting to perform. You could cache authentication and authorization details directly on your web service machines, but that could cause problems when changing permissions or blocking accounts, as these objects would need to expire before new permissions could take effect. A better approach is to use a shared in-memory object cache and have each web service machine reach out for the data needed at request time. If not present, data could be fetched from the original data store and placed in the object cache. By having a single central copy of each cached object, you will be able to easily invalidate it when users' permissions change.

- Another common problem when dealing with stateless web services is how to support resource locking. You can use distributed lock systems like Zookeeper or even build your own simple lock service using a data store of your choice. To make sure your web services scale well, you should avoid resource locks for as long as possible and look for alternative ways to synchronize parallel processes.
  - Distributed locking creates an opportunity for your service to stall or fail. This, in turn, increases your latency and reduces the number of parallel clients that your web service can serve. Instead of resource locks, you can sometimes use optimistic concurrency control where you check the state before the final update rather than acquiring locks. You can also consider message queues as a way to decouple components and remove the need for resource locking in the first place.
  - If none of the above techniques work for you and you need to use resource locks, it is important to strike a balance between having to acquire a lot of fine-grained locks and having coarse locks that block access to large sets of data. By having too many fine-grained locks, you increase risk for deadlocks. If you use few coarse locks, you can increase concurrency because multiple web services can be blocked waiting on the same resource lock.
- The last challenge is application-level transactions. A distributed transaction is a set of internal service steps and external web service calls that either complete together or fail entirely. It is very difficult to scale and coordinate without sacrificing high availability. The most common method of implementing distributed transactions is the 2 Phase Commit algorithm. An example of a distributed transaction would be a web service that creates an order within an online shop.
  - The first alternative to distributed transactions is to not support them at all. As long as the core of your system functionality is not compromised, your company may be fine with such a minor inconsistencies in return for the time saved developing it.
  - The second alternative to distributed transactions is to provide a mechanism of compensating transactions. A compensating transactins can be used to revert the result of an operation that was issued as part of a larger logical transaction that has failed. The benefit of this approach is that web services do not need to wait for one another; they do not need to maintain any state or resources for the duration of the overarching transaction either.

Caching service responses

- From a caching perspective, the GET method is the most important one, as GET responses can be cached.
- To be able to scale using cache, you would usually deploy reverse proxies between your clients and your web service. As your web services layer grow, you may end up with a more complex deployment where each of your web services has a reverse proxy dedicated to serve its results. Depending on the reverse proxy used, you may also have load balancers deployed between reverse proxies and web services to distribute the underlying network traffic and provide quick failure recovery.

Cache-Control header

- Setting the Cache-Control header to private bypasses intermediaries (such as nginx, other caching layers like Varnish, and all kinds of hardware in between) and only allows the end client to cache the response.
- Setting it to public allows intermediaries to store a copy of the response in their cache.

Expires

- Tells the browser that a resource should be cached and not requested again until the expiration date has elapsed.
- It's hard to define future Expires headers in API responses because if the data in the server changes, it could mean that the cleint's cache becomes stale, but it doesn't have any way of knowing that until the expiration date. A conservative alternative to Expires header in responses is using a pattern callled "conditional requests"

Last-Modified/If-Modified-Since/Max-age

- Specifying a Last-Modified header in your response. It's best to specify a max-age in the Cache-Control header, to let the browser invalidate the cache after a certain period of time even if the modification date doesn't change

> Cache-Control: private, max-age=86400 Last-Modified: Thu, 3 Jul 2014 18:31:12 GMT

- The next time the browser requests this resource, it will only ask for the contents of the resource if they're unchanged since this date, using the If-Modified-Since request header. If the resource hasn't changed since Thu, 3 Jul 2014

18:31:12 GMT, the server will return with an empty body with the 304 Not Modified status code.

> If-Modified-Since: Thu, 3 Jul 2014 18:31:12 GMT

ETag

* ETag header is usually a hash that represents the source in its current state. This allows the server to identify if the cached contents of the resource are different than the most recent versions:

> Cache-Control: private, max-age=86400 ETag: "d5jiodjiojiojo"

* On subsequent requests, the If-None-Match request header is sent with the ETag value of the last requested version for the same resource. If the current version has the same ETag value, your current version is what the client has cached and a 304 Not Modified response will be returned.

> If-None-Match: "d5jiodjiojiojo"

Vary: Authorization

* You could implement caching of authenticated REST resources by using headers like Vary: Authorization in your web service responses. Responses with such headers instruct HTTP caches to store a separate response for each value of the Authorization header.

Functional partitioning

* By functional partitioning, you group closely related functionality together. The resulting web services are loosely coupled and they can now be scaled independently.

Security

Throttling

* This kind of safeguarding is usually unnecessary when dealing with an internal API, or an API meant only for your front end, but it's a crucial measure to make when exposing the API publicly.
* Suppose you define a rate limit of 2,000 requests per hour for unauthenticated users; the API should include the following headers in its responses, with every request shaving off a point from the remainder. The X-RateLimit-Reset header should contain a UNIX timestamp describing the moment when the limit will be reset

> X-RateLimit-Limit: 2000 X-RateLimit-Remaining: 1999 X-RateLimit-Reset: 1404429213925

* Once the request quota is drained, the API should return a 429 Too Many Request response, with a helpful error message wrapped in the usual error envelope:

```
X-RateLimit-Limit: 2000
X-RateLimit-Remaining: 0
X-RateLimit-Reset: 1404429213925
{
        "error": {
                "code": "bf-429",
                "message": "Request quota exceeded. Wait 3 minutes and try again.",
                "context": {
                        "renewal": 1404429213925
                }
        }
}
```

* However, it can be very useful to notify the consumer of their limits before they actually hit it. This is an area that currently lacks standards but has a number of popular conventions using HTTP response headers.

Use OAuth2 with HTTPS for authorization, authentication and confidentiality.

Documentation

- Good documentation should
  - Explain how the response envelope works
  - Demonstrate how error reporting works
  - Show how authentication, paging, throttling, and caching work on a high level
  - Detail every single endpoint, explain the HTTP verbs used to query those endpoints, and describe each piece of data that should be in the request and the fields that may appear in the response
- Test cases can sometimes help as documentation by providing up-to-date working examples that also indicate best practices in accessing an API. The docs should show examples of complete request/response cycles. Preferably, the requests should be pastable examples - either links that can be pasted into a browser or curl examples that can be pasted into a terminal. GitHub and Stripe do a great job with this.
  - CURL: always illustrating your API call documentation by cURL examples. Readers can simply cut-and-paste them, and they remove any ambiguity regarding call details.
- Another desired component in API documentation is a changelog that briefly details the changes that occur from one version to the next. The documentation must include any deprecation schedules and details surrounding externally visible API updates. Updates should be delivered via a blog (i.e. a changelog) or a mailing list (preferably both!).

Others

- Provide filtering, sorting, field selection and paging for collections
  - Filtering: Use a unique query parameter for all fields or a query language for filtering.
    - GET /cars?color=red Returns a list of red cars
    - GET /cars?seats<=2 Returns a list of cars with a maximum of 2 seats
  - Sorting: Allow ascending and descending sorting over multiple fields
    - GET /cars?sort=-manufactorer,+model. This returns a list of cars sorted by descending manufacturers and ascending models.
  - Field selection: Mobile clients display just a few attributes in a list. They don't need all attributes of a resource. Give the API consumer the ability to choose returned fields. This will also reduce the network traffic and speed up the usage of the API.
    - GET /cars?fields=manufacturer,model,id,color
  - Paging:
    - Use limit and offset. It is flexible for the user and common in leading databases. The default should be limit=20 and offset=0. GET /cars?offset=10&limit=5.
    - To send the total entries back to the user use the custom HTTP header: X-Total-Count.
    - Links to the next or previous page should be provided in the HTTP header link as well. It is important to follow this link header values instead of constructing your own URLs.
- Content negotiation
  - Content-type defines the request format.
  - Accept defines a list of acceptable response formats. If a client requires you to return application/xml and the server could only return application/json, then you'd better return status code 406.
  - We recommend handling several content distribution formats. We can use the HTTP Header dedicated to this purpose: "Accept".
  - By default, the API will share resources in the JSON format, but if the request begins with "Accept: application/xml", resources should be sent in the XML format.
  - It is recommended to manage at least 2 formats: JSON and XML. The order of the formats queried by the header "Accept" must be observed to define the response format.
  - In cases where it is not possible to supply the required format, a 406 HTTP Error Code is sent (cf. Errors — Status Codes).
- Pretty print by default and ensure gzip is supported
  - An API that provides white-space compressed output isn't very fun to look at from a browser. Although some sort of query parameter (like ?pretty=true) could be provided to enable pretty printing, an API that pretty prints by default is much more approachable.
- HATEOAS: Hypertext As The Engine of Application State
  - There should be a single endpoint for the resource, and all of the other actions you'd need to undertake should be

able to be discovered by inspecting that resource.
    - People are not doing this because the tooling just isn't there.

Data partitioning - Sharding

Sharding benefits

- Scale horizontally to any size. Without sharding, sooner or later, your data set size will be too large for a single server to manage or you will get too many concurrent connections for a single server to handle. You are also likely to reach your I/O throughput capacity as you keep reading and writing more data. By using application-level sharing, none of the servers need to have all of the data. This allows you to have multiple MySQL servers, each with a reasonable amount of RAM, hard drives, and CPUs and each of them being responsible for a small subset of the overall data, queries, and read/write throughput.
- Since sharding splits data into disjoint subsets, you end up with a share-nothing architecture. There is no overhead of communication between servers, and there is no cluster-wide synchronization or blocking. Servers are independent from each other because they shared nothing. Each server can make authoritative decisions about data modifications
- You can implement in the application layer and then apply it to any data store, regardless of whether it supports sharding out of the box or not. You can apply sharding to object caches, message queues, nonstructured data stores, or even file systems.

Sharding key

- Determine what tables need to be sharded. A good starting point for deciding that is to look at the number of rows in the tables as well as the dependencies between the tables.
    - Typically you use only a single column as partition key. Using multiple columns can be hard to maintain unless they are hard to maintain.
    - Sharding on a column that is a primary key offers significant advantages. The reason for this is that the column should have a unique index, so that each value in the column uniquely identifies the row.

Sharding function

Static sharding

- Def: The sharding key is mapped to a shard identifier using a fixed assignment that never changes.
    - Static sharding schemes run into problems when the distribution of the queries is not even.
- Types:
    - Range partitioning (used in HBase)
        - Easy to implement but distribution can easy to become uneven. For example, if you are using URIs as keys, "hot" sites will be clustered together when you actually want the opposite, to spread them out. One hard can become overloaded and you have to split it a lot to be able to cope with the increase in load.
    - Hash partitioning
        - Computes a hash of the input in some manner (MD5 or SHA-1) and then uses modulo arithmetic to get a number between 1 and the number of the shards.
            - Evenly distributed but need large amount of data migration when the number of server changes and rehashing
        - Consistent hashing: Gauranteed to move rows from just one old shard to the new shard. The entire hash range is shown as a ring. On the hash ring, the shards are assigned to points on the ring using the hash function. In a similar manner, the rows are distributed over the ring using the same hash function. Each shard is now responsible for the region of the ring that starts at the shard's point on the ring and continues to the next shard point. Because a region may start at the end of the hash range and wrap around to the beginning of the hash range, a ring is used here instead of a flat line.
            - Pick a hash function which must have a big range, hence a lot of "points" on the hash ring where rows can be assigned. The most commonly used functions are MD5, SHA and Murmur hash (murmur3 -2^128, 2^128)

        - Less data migration but hard to balance node
            - Unbalanced scenario 1: Machines with different processing power/speed.
            - Unbalanced scenario 2: Ring is not evenly partitioned.
            - Unbalanced scenario 3: Same range length has different amount of data.
    - Virtual nodes (Used in Dynamo and Cassandra)
        - Solution: Each physical node associated with a different number of virtual nodes.
        - Problems: Data should not be replicated in different virtual nodes but the same physical nodes.

Dynamic sharding

- The sharding key is looked up in a dictionary that indicates which shard contains the data.
    - More flexible. You are allowed to change the location of shards and it is also easy to move data between shards if you have to. You do not need to migrate all of the data in one shot, but you can do it incrementally, one account at a time. To migrate a user, you need to lock its account, migrate the data, and then unlock it. You could usually do these migrations at night to reduce the impact on the system, and you could also migrate multiple accounts at the same time. There is an additional level of flexibility, as you can cherry-pick users and migrate them to the shards of your choice. Depending on the application requirements, you could migrate your largest or busiest clients to separate dedicated database instances to give them more capacity.
    - Requires a centralized store called the sharding database and extra queries to find the correct shard to retrieve the data from.

Challenges

Cross-shard joins

- Tricky to execute queries spanning multiple shards. The most common reason for using cross-shard joins is to create reports. This usually requires collecting information from the entire database. There are basically two approaches to solve this problem
    - Execute the query in a map-reduce fashion (i.e., send the query to all shards and collect the result into a single result set). It is pretty common that running the same query on each of your servers and picking the highest of the values will not guarantee a correct result.
    - Replicate all the shards to a separate reporting server and run the query there. This approach is easier. It is usually feasible, as well, because most reporting is done at specific times, is long-running, and does not depend on the current state of the database.

Using AUTO_INCREMENT

- It is quite common to use AUTO_INCREMENT to create a unique identifier for a column. However, this fails in a sharded environment because the the shards do not syncrhonize their AUTO_INCREMENT identifiers. This means if you insert a row in one shard, it might well happen that the same identifier is used on another shard. If you truly want to generate a unique identifer, there are basically three approaches.
    - Generate a unique UUID. The drawback is that the identifier takes 128 bits (16 bytes).
    - Use a composite identifier. Where the first part is the shard identifier and the second part is a locally generated identifier. Note that the shard identifier is used when generating the key, so if a row with this identifier is moved, the original shard identifier has to move with it. You can solve this by maintaining, in addition to the column with the AUTO_INCREMENT, an extra column containing the shard identifier for the shard where the row was created.
    - Use atomic counters provided by some data stores. For example, if you already use Redis, you could create a counter for each unique identifier. You would then use Redis' INCR command to increase the value of a selected counter and return it with a different value.

Distributed transactions

- Lose the ACID properties of your database as a whole. Maintaining ACID properties across shards requires you to use distributed transactions, which are complex and expensive to execute (most open-source database engines like MySQL do not even support distributed transactions).

Clones - Replication

Replication purpose

High availability by creating redundancy

- Duplicate components
    - Def: Keep duplicates around for each component - ready to take over immediately if the original component fails.
    - Characteristics: Do not lose performance when switching and switching to the standby is usually faster than restructuring the system. But expensive.
    - For example: Hot standby
        - A dedicated server that just duplicates the main master. The hot standby is connected to the master as a slave, so that it reads and applies all changes. This setup is often called primary-backup configuration.
- Create spare capacity
    - Def: Have extra capacity in the system so that if a component fails, you can still handle the load.
    - Characteristics: Should one of the component fail, the system will still be responding, but the capacity of the system will be reduced.

Planning for failures

- Slave failures
    - Because the slaves are used only for read quires, it is sufficient to inform the load balancer that the slave is missing. Then we can take the failing slave out of rotation. rebuild it and put it back.
- Master failures
    - Problems:
        - All the slaves have stale data.
        - Some queries may block if they are waiting for changes to arrive at the slave. Some queries may make it into the relay log of the slave and therefore will eventually be executed by the slave. No special consideration has to be taken on the behalf of these queries.
        - For queries that are waiting for events that did not leave the master before it crashed, they are usually reported as failures so users should reissue the query.
    - Solutions:
        - If simply restart does not work
        - First find out which of your slaves is most up to date.
        - Then reconfigure it to become a master.
        - Finally reconfigure all remaining slaves to replicate from the new master.
- Relay failures
    - For servers acting as relay servers, the situation has to be handled specially. If they fail, the remaining slaves have to be redirected to use some other relay or the master itself.
- Disaster recovery
    - Disaster does not have to mean earthquakes or floods; it just means that something went very bad for the computer and it is not local to the machine that failed. Typical examples are lost power in the data center (not necessarily because the power was lost in the city; just losing power in the building is sufficient.)
    - The nature of a disaster is that many things fail at once, making it impossible to handle redundancy by duplicating servers at a single data center. Instead, it is necessary to ensure data is kept safe at another geographic location, and it is quite common for companies to ensure high availability by having different components at different offices.

Replication for scaling read

When to use

- Scale reads: Instead of a single server having to respond to all the queries, you can have many clones sharing the load. You can keep scaling read capacity by simply adding more slaves. And if you ever hit the limit of how many slaves your master can handle, you can use multilevel replication to further distribute the load and keep adding even more slaves. By adding multiple levels of replication, your replication lag increases, as changes need to propogate through more servers, but you can increase read capacity.
- Scale the number of concurrently reading clients and the number of queries per second: If you want to scale your database to support 5,000 concurrent read connections, then adding more slaves or caching more aggressively can be a great way to go.

When not to use

- Scale writes: No matter what topology you use, all of your writes need to go through a single machine.
    - Although a dual master architecture appears to double the capacity for handling writes (because there are two masters), it actually doesn't. Writes are just as expensive as before because each statement has to be executed twice: once when it is received from the client and once when it is received from the other master. All the writes done by the A clients, as well as B clients, are replicated and get executed twice, which leaves you in no better position than before.
- Not a good way to scale the overall data set size: If you want to scale your active data set to 5TB, replication would not help you get there. The reason why replication does not help in scaling the data set size is that all of the data must be present on each of the machines. The master and each of its slave need to have all of the data.
    - Def of active data set: All of the data that must be accessed frequently by your application. (all of the data your database needs to read from or write to disk within a time window, like an hour, a day, or a week.)
    - Size of active data set: When the active data set is small, the database can buffer most of it in memory. As your active data set grows, your database needs to load more disk blocks because in-memory buffers are not large enough to contain enough of the active disk blocks.
    - Access pattern of data set
        - Like a time-window: In an e-commerce website, you use tables to store information about each purchase. This type of data is usually accessed right after the purchase and then it becomes less and less relevant as time goes by. Sometimes you may still access older transactions after a few days or weeks to update shipping details or to perform a refund, but after that, the data is pretty much dead except for an occasional report query accessing it.
        - Unlimited data set growth: A website that allowed users to listen to music online, your users would likely come back every day or every week to listen to their music. In such case, no matter how old an account is, the user is still likely to log in and request her playlists on a weekly or daily basis.

Replication Topology

Master-slave vs peer-to-peer

Types	Strengths	Weakness
Master-slave	• Helpful for scaling when you have a read-intensive dataset. Can scale horizontally to handle more read requests by adding more slave nodes and ensuring that all read requests are routed to the slaves. • Helpful for read resilience. Should the master fail, the slaves can still handle read requests. • Increase availability by reducing the time needed to replace the broken database. Having slaves as replicas of the master	• Not a good scheme for datasets with heavy write traffic, although offloading the read traffic will help a little bit with handling the write load. All of your writes need to go through a single machine • The failure of the master does eliminate the ability to handle writes until either the master is restored or a new master is appointed. • Inconsistency. Different clients reading different slaves will see different values because the changes haven't all propagated to the slaves. In the worst

	does speed up recovery after a failure of the master since a slave can be appointed a new master very quickly.	case, that can mean that a client cannot read a write it just made.
p2p: Master-master	• Faster master failover. In case of master A failure, or anytime you need to perform long-lasting maintainence, your application can be quickly reconfigured to direct all writes to master B. • More transparent maintainance. Switch between groups with minimal downtime.	Not a viable scalability technique. • Need to use auto-increment and UUID() in a specific way to make sure you never end up with the same sequence number being generated on both masters at the same time. • Data inconsistency. For example, updating the same row on both masters at the same time is a classic race condition leading to data becoming inconsistent between masters. • Both masters have to perform all the writes. Each of the master needs to execute every single write statement either coming from your application or via the replication. To make it worse, each master will need to perform additional I/O to write replicated statements into the relay log. • Both masters have the same data set size. Since both masters have the exact same data set, both of them will need more memory to hold ever-growing indexes and to keep enough of the data set in cache.
p2p: Ring-based	Chain three or more masters together to create a ring.	• All masters need to execute all the write statements. Does not help scale writes. • Reduced availability and more difficult failure recovery: Ring topology makes it more difficult to replace servers and recover from failures correctly. • Increase the replication lag because each write needs to jump from master to master until it makes a full circle.

Master-slave replication

- Responsibility:
  - Master is reponsible for all data-modifying commands like updates, inserts, deletes or create table statements. The master server records all of these statements in a log file called a binlog, together with a timestamp, and a sequence number to each statement. Once a statement is written to a binlog, it can then be sent to slave servers.
  - Slave is responsible for all read statements.
- Replication process: The master server writes commands to its own binlog, regardless if any slave servers are connected or not. The slave server knows where it left off and makes sure to get the right updates. This asynchronous process decouples the master from its slaves - you can always connect a new slave or disconnect slaves at any point in time without affecting the master.
  i. First the client connects to the master server and executes a data modification statement. The statement is executed and written to a binlog file. At this stage the master server returns a response to the client and continues processing other transactions.
  ii. At any point in time the slave server can connect to the master server and ask for an incremental update of the master' binlog file. In its request, the slave server provides the sequence number of the last command that it saw.
  iii. Since all of the commands stored in the binlog file are sorted by sequence number, the master server can quickly locate the right place and begin streaming the binlog file back to the slave server.
  iv. The slave server then writes all of these statements to its own copy of the master's binlog file, called a relay log.

v. Once a statement is written to the relay log, it is executed on the slave data set, and the offset of the most recently seen command is increased.

Number of slaves

- It is a common practice to have two or more slaves for each master server. Having more than one slave machine have the following benefits:
    - Distribute read-only statements among more servers, thus sharding the load among more servers
    - Use different slaves for different types of queries. E.g. Use one slave for regular application queries and another slave for slow, long-running reports.
    - Losing a slave is a nonevent, as slaves do not have any information that would not be available via the master or other slaves.

Peer-to-peer replication

- Dual masters
    - Two masters replicate each other to keep both current. This setup is very simple to use because it is symmetric. Failing over to the standby master does not require any reconfiguration of the main master, and failing back to the main master again when the standby master fails in turn is very easy.
        - Active-active: Writes go to both servers, which then transfer changes to the other master.
        - Active-passive: One of the masters handles writes while the other server, just keeps current with the active master
    - The most common use of active-active dumal masters setup is to have the servers geographically close to different sets of users - for example, in branch offices at different places in the world. The users can then work with local server, and the changes will be replicated over to the other master so that both masters are kept in sync.
- Circular replication

Replication mode

Synchronous and Asynchronous

- Asynchronous: The master does not wait for the slaves to apply the changes, but instead just dispatches each change request to the slaves and assume they will catch up eventually and replicate all the changes.
- Synchronous: The master and slaves are always in sync and a transaction is not allowed to be committed on the master unless the slaves agrees to commit it as well (i.e. synchronous replication makes the master wait for all the slaves to keep up with the writes.)

Synchronous vs Asynchronous

- Asynchronous replication is a lot faster than synchronous replication. Compared with asynchronous replication, synchronous replication requires extra synchronization to guarantee consistency. It is usually implemented through a protocol called two-phase commit, which guarantees consistency between the master and slaves. What makes this protocol slow is that it requires a total of four messages, including messages with the transaction and the prepare request. The major problem is not the amount of network traffic required to handle the synchronization, but the latency introduced by the network and by processing the commit on the slave, together with the fact that the commit is blocked on the master until all the slaves have acknowledged the transaction. In contrast, the master does not have to wait for the slave, but can report the transaction as committed immediately, which improves performance significantly.
- The performance of asynchronous replication comes at the price of consistency. In asynchronous replication the transaction is reported as committed immediately, without waiting for any acknowledgement from the slave.

Cache

Why does cache work

- Long tail
- Locality of reference

Cache hit ratio

- Size of cache key space
    - The more unique cache keys your application generates, the less chance you have to reuse any one of them. Always consider ways to reduce the number of possible cache keys.
- The number of items you can store in cache
    - The more objects you can physically fit into your cache, the better your cache hit ratio.
- Longevity
    - How long each object can be stored in cache before expiring or being invalidated.

How much will cache benefit

- short answer
    - How many times a cached piece of data can and is reused by the application
    - the proportion of response time that is alleviated by caching
- In applications that are I/O bound, most of the response time is getting data from a database.

Access pattern

Write through cache

- def: write go through the cache and write is confirmed as success only if writes to DB and the cache both succeed.
- use-case: applications which write and re-read the information quickly. But the write latency might be much higher because of two write phase

Write around cache

- def: write directly goes to the DB. The cache reads the info from DB in case of a miss
- use-case: lower write load to cache and faster writes, but can lead to higher read latency in case of applications which write and re-read the information quickly

Write back cache

- def: write is directly done to the caching layer and write is confirmed as soon as the write to the cache completes.The cache then asynchronously syncs this write to the DB.
- use-case: quick write latency and high write throughput. But might lose data in case the cache layer dies

How to handle cache failure

- Facebook Lease Get

Typical caching scenarios

- The first and best scenario is allowing your clients to cache a response forever. This is a very important technique and you want to apply it for all of your static content (like image, CSS, or Javascript files). Static content files should be considered immutable, and whenever you need to make a change to the contents of such a file, you should publish it under a new URL. Want you want to deploy a new version of your web application, you can bundle and minify all of your CSS files and include a timestamp or a hash of the contents of the file in the URL. Even though you could cache static files forever, you should not set the Expires header more than one year into the future.
- The second most common scenario is the worst case - when you want to make sure that the HTTP response is never stored, cached, or reused for any users.
- A last use case is for situations where you want the same user to reuse a piece of content, but at the same time you

do not want other users to share the cached response.

HTTP Cache

- All of the caching technologies working in the HTTP layer work as read-through caches
  - Procedures
    - First Client 1 connects to the cache and request a particular web resource.
    - Then the cache has a change to intercept the request and respond to it using a cached object.
    - Only if the cache does not have a valid cached response, will it connect to the origin server itself and forward the client's request.
  - Advantages: Read-through caches are especially attractive because they are transparent to the client. This pluggable architecture gives a lot of flexibility, allowing you to add layers of caching to the HTTP stack without needing to modify any of the clients.

Headers

- Conditional gets: If-Modified-Since header in the get request
  - If the server determines that the resource has been modified since the date given in this header, the resource is returned as normal. Otherwise, a 304 Not Modified status is returned.
  - Use case: Rather than spending time downloading the resource again, the browser can use its locally cached copy. When downloading of the resource only forms only a small fraction of the request time, it doesn't have much benefit.
- max-age inside Expires and Cache-Contrl: The resource expires on such-and-such a date. Until then, you can just use your locally cached copy.
  - The main difference is that Expires was defined in HTTP 1.0, whereas the Cache-Control family is new to HTTP 1.1. So, in theory, Expires is safer because you occasionally still encounter clients that support only HTTP 1.0. Although if both are presents, preferences are given to Cache-Control: max-age.
  - Choosing expiration policies:
    - Images, CSS, Javascript, HTML, Flash movies are primary candidates. The only type of resources you don't usually want to cache is dynamically generated content created by server-side scripting languages such as PHP, Perl and Ruby. Usually one or two months seem like a good figure.
  - Coping with stale content: There are a few tricks to make the client re-request the resource, all of which revolved around changing the URL to trick the browser into thinking the resource is not cached.
    - Use a version/revision number or date in the filename
    - Use a version/revision number or date in the path
    - Append a dummy query string
- Other headers inside Cache-Control:
  - private: The result is specific to the user who requested it and the response cannot be served to any other user. This means that only browsers will be able to cache this response because intermediate caches would not have the knowledge of what identifies a user.
  - public: Indicates the response can be shared between users as long as it has not expired. Note that you cannot specify private and public options together; the response is either public or private.
  - no-store: Indicates the response should not be stored on disks by any of the intermediate caches. In other words, the response can be cached in memory, but it will not be persisted to disk.
  - no-cache: The response should not be cache. To be accurate, it states that the cache needs to ask the server whether this response is still valid every time users request the same resource.
  - max-age: Indicates how many seconds this response can be served from the cache before becoming stale. (TTL of the response)
  - s-maxage: Indicates how many seconds this response can be served from the cache before becoming stale on shared caches.
  - no-transformation: Indicates the response should be served without any modifications. For example, a CDN provider might transcode images to reduce their size, lowering the quality or changing the compression algorithm.

-
    - must-revalidate: Once the response becomes stale, it cannot be returned to clients without revalidation. Although it may seem odd, caches may return stale objects under certain conditions. For example, if the client explicitly allows it or if the cache loses connection to the original server.
- Expires:
    - Allows you to specify an absolute point in time when the object becomes stale.
    - Some of the functionality controlled by the Cache-Control header overlaps that of other HTTP headers. Expiration time of the web response can be defined either by Cache-Control: max-age=600 or by setting an absolute expiration time using the Expires header. Including both of these headers in the response is redundant and leads to confusion and potentially inconsistent behavior.
- Vary:
    - Tell caches that you may need to generate multiple variations of the response based on some HTTP request headers. For example: Vary:Accept-Encoding is the most common Vary header indicating that you may return responses encoded in different ways depending on the Accept-Encoding header that the client sends to your web server. Some clients who accept gzip encoding will get a compressed response, where others who cannot support gzip will get an uncompressed response.
- How not to cache:
    - It's common to see meta tags used in the HTML of pages to control caching. This is a poor man's cache control technique, which isn't terribly effective. Although most browsers honor these meta tags when caching locally, most intermediate proxies do not.

Types

Browser cache

- Browsers have built-in caching capabilities to reduce the number of request sent out. These usually uses a combination of memory and local files.
- There are several problems with browser cache
    - The size of the cache tends to be quite small by default. Usually around 1GB. Given that web pages have become increasingly heavy, browsers would probably be more effective if they defaulted to much larger caches.
    - When the cache becomes full, the algorithm to decide what to remove is crude. Commonly, the LRU algorithm is used to purge old items. It fails to take into account the relative "cost" to request different types of resources. For example, the loading of Javascript resources typically blocks loading of the rest of the page. It makes more sense for these to be given preference in the cache over, say, images.
    - Many browsers offer an easy way for the user to remove temporary data for the sake of privacy. Users often feel that cleaning the browser cache is an important step in somehow stopping their PC from running slow.

Caching proxies

- A caching proxy is a server, usually installed in a local corporate network or by the Internet service provider (ISP). It is a read-through cache used to reduce the amount of traffic generated by the users of the network by reusing responses between users of the network. The larger the network, the larger the potential savings - that is why it was quite common among ISPs to install transparent caching proxies and route all of the HTTP traffic through them to cache as many requests as possible.
- In recent years, the practice of installing local proxy servers has become less popular as bandwidth has become cheaper and as it becomes more popular for websiste to serve their resources soley over the Secure Socket Layer.

Reverse proxy

- A reverse proxy works in the exactly same way as a regular caching proxy, but the intent is to place a reverse proxy in your own data center to reduce the load put on your web servers.
- Purpose:
    - For caching, they can be used to lighten load on the back-end server by serving up cached versions of dynamically generated pages (thus cuttping CPU usage). Using reverse proxies can also give you more flexibility because you can override HTTP headers and better control which requests are being cached and for how long.
    - For load balancing, they can be used for load-balancing multiple back-end web servers.

Content delivery networks

- A CDN is a distributed network of cache servers that work in similar way as caching proxies. They depend on the same HTTP headers, but they are controlled by the CDN service provider.
- Advantage:
    - Reduce the load put on your servers
    - Save network bandwidth
    - Improve the user experience because by pushing content closer to your users.
- Procedures: Web applications would typically use CDN to cache their static files like images, CSS, JavaScript, videos or PDF.
    - You can imlement it easily by creating a static subdomain and generate URLs for all of your static files using this domain
    - Then you configure the CDN provider to accept these requests on your behalf and point DNS for s.example.org to the CDN provider.
    - Any time CDN fails to serve a piece of content from its cache, it forwards the request to your web servers and caches the response for subsequent users.

Scaling

- Do not worry about the scalability of browser caches or third-party proxy servers.
- This usually leaves you to manage reverse proxy servers. For most young startups, a single reverse proxy should be able to handle the incoming traffic, as both hardware reverse proxies and leading open-source ones can handle more than 10,000 requests per second from a single machine.
    - First step: To be able to scale the reverse proxy layer efficiently, you need to first focus on your cache hit ratio first.
        - Cache key space: Describe how many distinct URLs your reverse proxies will observe in a period of time. The more distinct URLs are served, the more memory or storage you need on each reverse proxy to be able to serve a significant portion of traffic from cache. Avoid caching responses that depend on the user (for example, that contain the user ID in the URL). These types of response can easily pollute your cache with objects that cannot be reused.
        - Average response TTL: Describe how long each response can be cached. The longer you cache objects, the more chance you have to reuse them. Always try to cache objects permanently. If you cannot cache objects forever, try to negotiate the longest acceptable cache TTL with your business stakeholders.
        - Average size of cached object: Affects how much memory or storage your reverse proxies will need to store the most commonly accessed objects. Average size of cached object is the most difficult to control, but you should still keep in mind because there are some techniques that help you "shrink" your objects.
    - Second step: Deploying multiple reverse proxies in parallel and distributing traffic among them. You can also scale reverse proxies vertically by giving them more memory or switching their persistent storage to solid-state drive.

Application objects cache

- Application object caches are mostly cache-aside caches. The application needs to be aware of the existence of the object cache, and it actively uses it to store and retrieve objects rather than the cache being transparently positioned between the application and its data sources.
- All of the object cache types discussed in this section can be imagined as key-value stores with support of object expiration.

Types

Client-side web storage

- Web storage allows a web application to use a limited amount (usually up to 5MB to 25MB of data).
- Web storage works as a key-value store.

Caches co-located with code: One located directly on your web servers.

- Objects are cached directly in the application's memory
- Objects are stored in shared memory segments so that multiple processes running on the same machine could access them.
- A caching server is deployed on each web server as a separate application.

Distributed cache store

- Interacting with a distributed object cache usually requires a network round trip to the cache server. On the plus side, distributed object caches usually work as simple key-value stores, allowing clients to store data in the cache. You can scale simply by adding more servers to the cache cluster. By adding servers, you can scale both the throughput and overall memory pool of your cache.

Scaling

- Client-side caches like web browser storage cannot be scaled.
- The web server local caches are usually scaled by falling back to the file system.
- Distributed caches are usually scaled by data partitioning. Adding read-only slaves to sharded node.

Caching rules of thumb

Cache priority

- The higher up the call stack you can cache, the more resources you can save.
- Aggregated time spent = time spent per request * number of requests

Cache reuse

- Always try to reuse the same cached object for as many requests/users as you can.

Cache invalidation

- LRU
- TTL

Pains

Pain of large data sets - When cache memory is full

- Evict policies
  - FIFO (first-in, first out)
  - LRU (least recently used)
  - LFU (least frequently used)
  - See reference section for more discussions
- What to do with evicted one
  - Overflow to disk
  - Delete it

Pain of stale data

- Expiration policy
  - TTI: time to idle, a counter count down if not reset
  - TTL: time to leave, maximum tolerance for staleness

Pain of loading

- Persistent disk store
- Bootstrap cache loader

- def: on startup, create background thread to pull the existing cache data from another peer
- automatically bootstrap key on startup
- cache value on demand

Pain of duplication

- Get failover capability but avoid excessive duplication of data
- Each node hods data it has seen
- Use load balancer to get app-level partitioning
- Use fine grained locking to get concurrency
- Use memory flush/fault to handle memory overflow and availability
- Use casual ordering to guarantee coherency

Thundering herd problem

Def

- Many readers read an empty value from the cache and subseqeuntly try to load it from the database. The result is unnecessary database load as all readers simultaneously execute the same query against the database.
- Let's say you have [lots] of webservers all hitting a single memcache key that caches the result of a slow database query, say some sort of stat for the homepage of your site. When the memcache key expires, all the webservers may think "ah, no key, I will calculate the result and save it back to memcache". Now you have [lots] of servers all doing the same expensive DB query.

```
/* read some data, check cache first, otherwise read from SoR */
public V readSomeData(K key) {
  Element element;
  if ((element = cache.get(key)) != null) {
    return element.getValue();
  }

  // note here you should decide whether your cache
  // will cache 'nulls' or not
  if (value = readDataFromDataStore(key)) != null) {
    cache.put(new Element(key, value));
  }

  return value;
}
```

Solutions

- Stale date solution: The first client to request data past the stale date is asked to refresh the data, while subsequent requests are given the stale but not-yet-expired data as if it were fresh, with the understanding that it will get refreshed in a 'reasonable' amount of time by that initial request
    - When a cache entry is known to be getting close to expiry, continue to server the cache entry while reloading it before it expires.
    - When a cache entry is based on an underlying data store and the underlying data store changes in such a way that the cache entry should be updated, either trigger an (a) update or (b) invalidation of that entry from the data store.
- Add entropy back into your system: If your system doesn’t jitter then you get thundering herds.
    - For example, cache expirations. For a popular video they cache things as best they can. The most popular video they might cache for 24 hours. If everything expires at one time then every machine will calculate the expiration at the same time. This creates a thundering herd.

  - By jittering you are saying randomly expire between 18-30 hours. That prevents things from stacking up. They use this all over the place. Systems have a tendency to self synchronize as operations line up and try to destroy themselves. Fascinating to watch. You get slow disk system on one machine and everybody is waiting on a request so all of a sudden all these other requests on all these other machines are completely synchronized. This happens when you have many machines and you have many events. Each one actually removes entropy from the system so you have to add some back in.

- No expire solution: If cache items never expire then there can never be a recalculation storm. Then how do you update the data? Use cron to periodically run the calculation and populate the cache. Take the responsibility for cache maintenance out of the application space. This approach can also be used to pre-warm the the cache so a newly brought up system doesn't peg the database.

  - The problem is the solution doesn't always work. Memcached can still evict your cache item when it starts running out of memory. It uses a LRU (least recently used) policy so your cache item may not be around when a program needs it which means it will have to go without, use a local cache, or recalculate. And if we recalculate we still have the same piling on issues.
  - This approach also doesn't work well for item specific caching. It works for globally calculated items like top N posts, but it doesn't really make sense to periodically cache items for user data when the user isn't even active. I suppose you could keep an active list to get around this limitation though.

Scaling Memcached at Facebook

- In a cluster:
  - Reduce latency
    - Problem: Items are distributed across the memcached servers through consistent hashing. Thus web servers have to rountinely communicate with many memcached servers to satisfy a user request. As a result, all web servers communicate with every memcached server in a short period of time. This all-to-all communication pattern can cause incast congestion or allow a single server to become the bottleneck for many web servers.
    - Solution: Focus on the memcache client.
  - Reduce load
    - Problem: Use memcache to reduce the frequency of fetching data among more expensive paths such as database queries. Web servers fall back to these paths when the desired data is not cached.
    - Solution: Leases; Stale values;
  - Handling failures
    - Problem:
      - A small number of hosts are inaccessible due to a network or server failure.
      - A widespread outage that affects a significant percentage of the servers within the cluster.
    - Solution:
      - Small outages: Automated remediation system.
      - Gutter pool
  - In a region: Replication
  - Across regions: Consistency

Architecture

Lambda architecture

Building blocks

Load balancer

Hardware vs software

Category	Software	Hardware
Def	Run on standard PC hardware, using applications like Nginx and HAProxy	Run on special hardware and contain any software pre-installed and configured by the vendor.
Model	Operate on Application Layer	Operate on network and transport layer and work with TCP/IP packets. Route traffic to backend servers and possibly handling network address translation
Strength/Weakness	More intelligent because can talk HTTP (can perform the compression of resources passing through and routing-based on the presence of cookies) and more flexible for hacking in new features or changes	Higher throughput and lower latency. High purchase cost. Hardware load balancer prices start from a few thousand dollars and go as high as over 100,000 dollars per device. Specialized training and harder to find people with the work experience necessary to operate them.

HAProxy vs Nginx

Category	Nginx	HAProxy
Strengths	Can cache HTTP responses from your servers.	A little faster than Nginx and a wealth of extra features. It can be configured as either a layer 4 or layer 7 load balancer.

- Extra functionalities of HAProxy. It can be configured as either a layer 4 or layer 7 load balancer.
  - When HAProxy is set up to be a layer 4 proxy, it does not inspect higher-level protocols and it depends solely on TCP/IP headers to distribute the traffic. This, in turn, allows HAProxy to be a load balancer for any protocol, not just HTTP/HTTPS. You can use HAProxy to distribute traffic for services like cache servers, message queues, or databases.
  - HAProxy can also be configured as a layer 7 proxy, in which case it supports sticky sessions and SSL termination, but needs more resources to be able to inspect and track HTTP-specific information. The fact that HAProxy is simpler in design makes it perform sligthly better than Nginx, especially when configured as a layer 4 load balancer. Finally, HAProxy has built-in high-availability support.

Web server

Apache and Nginx

- Apache and Nginx could always be used together.
  - NGINX provides all of the core features of a web server, without sacrificing the lightweight and high-performance qualities that have made it successful, and can also serve as a proxy that forwards HTTP requests to upstream web servers (such as an Apache backend) and FastCGI, memcached, SCGI, and uWSGI servers. NGINX does not seek to implement the huge range of functionality necessary to run an application, instead relying on specialized third-party servers such as PHP-FPM, Node.js, and even Apache.
  - A very common use pattern is to deploy NGINX software as a proxy in front of an Apache-based web application. Can use Nginx's proxying abilities to forward requests for dynamic resources to Apache backend server. NGINX serves static resources and Apache serves dynamic content such as PHP or Perl CGI scripts.

Apache vs Nginx

Category	Apache	Nginx
	Invented around 1990s when web traffic is low and web pages are really simple. Apache's heavyweight, monolithic model has its limit.	Heavy traffic and web pages. Designed for high concurrency. Provides 12

History	Tunning Apache to cope with real-world traffic efficiently is a complex art.	features including which make them appropriate for microservices.
Architecture	One process/threads per connection. Each requests to be handled as a separate child/thread.	Asynchronous event-driven model. There is a single master process with one or more worker processes.
Performance	To decrease page-rendering time, web browsers routinely open six or more TCP connections to a web server for each user session so that resources can download in parallel. Browsers hold these connections open for a period of time to reduce delay for future requests the user might make during the session. Each open connection exclusively reserves an httpd process, meaning that at busy times, Apache needs to create a large number of processes. Each additional process consumes an extra 4MB or 5MB of memory. Not to mention the overhead involved in creating and destroying child processes.	Can handle a huge number of concurrent requests
Easier development	Very easy to insert additional code at any point in Apache's web-serving logic. Developers could add code securely in the knowledge that if newly added code is blocked, ran slowly, leaked resources, or even crashed, only the worker process running the code would be affected. Processing of all other connections would continue undisturbed	Developing modules for it isn't as simple and easy as with Apache. Nginx module developers need to be very careful to create efficient and accurate code, without any resource leakage, and to interact appropriately with the complex event-driven kernel to avoid blocking operations.

Cache

In-memory cache - Guava cache

Standalone cache

Memcached

Redis

Database

DynamoDB

Cassandra

Queue

ActiveMQ

RabbitMQ

SQS

Kafka

Data Processing

Hadoop

Spark

EMR

Stream Processing

Samza

Storm

References

- Hired in Tech courses
- Blogs and papers
- Books: "Professional Website Performance" by Peter Smith
- Books: "Web Scalability for Startup Engineers" by Artur Ejsmont
- Books: "MySQL High Performance" and "MySQL High Availability" from O'Reilly
- Jiuzhang/Bittiger system design class
- Gainlo blog