CS50's Web Programming with Python and JavaScript

OpenCourseWare

Brian Yu (https://brianyu.me)
brian@cs.harvard.edu

David J. Malan (https://cs.harvard.edu/malan/)
malan@harvard.edu
(https://www.facebook.com/dmalan) (https://github.com/dmalan) (https://www.instagram.com/davidjmalan/)
(https://www.linkedin.com/in/malan/) (https://www.quora.com/profile/David-J-Malan) (https://twitter.com/davidjmalan)

Scalability

- After a web application has been developed, there are still issues that need to be considered when deploying to the internet. A big concern is scalability. An application can work well with only a few users, but it also needs to be able to support many more people accessing data and using the application simultaneously. The vast majority of the time, there is no one best way to scale, but rather a number ways with different trade-offs that should be taken into account.

Server Scaling

- A server can only perform a finite number of tasks per second, measured in hertz. Modern servers and processors often have speeds in gigahertz. The operations being measured are low-level: adding two numbers together, for example. Due to limited server speed, a server can only respond to a limited number of users in a given second.
- Determining the how much a server can handle is a process called benchmarking. It is not a good idea to figure out a server's capacity by waiting for it to hit capacity.
- One response to limited server capacity is 'vertical scaling': give the server more resources (memory, computing power) so it can handle additional load. Vertical scaling has its limits, however. A server can only be built up so much.
- Another approach is 'horizontal scaling': add more servers to handle additional load.
- With horizontal scaling, when a user tries to connect to the application, another piece of hardware, a load balancer, determines which server to send the user to. The load balancer can decide where to send a user based on a variety of different algorithms:
  - Random Choice: Every time a user connects, a random server is chosen.
  - Round Robin: The first connection is directed to server 1, the second to server 2, and so on, in a circular fashion.
  - Fewest Connections: Whichever server is currently serving the fewest users will receive any new users.
- Random choice and round robin might end up loading certain servers more heavily, and fewest connections might not always be a perfect measure of actual computational load. Putting a lot of effort into load balancing, however, creates an entirely new problem of creating another chokepoint that could get bogged down by a large number of users.
- A general problem with the load balancing model is that when a user is accessing multiple different pages repeatedly and making multiple requests, the load balancer might end up sending the user to a different server. The different servers a user has been directed to might not have the user's session synchronized, for example. Therefore, there has to be some sort of 'session-aware' load balancing.
  - Sticky Session: After the initial connection, send the user to the same server repeatedly.
  - Sessions in Database: Store all session information in a universally accessible database. This does result in more communication time, however.
  - Client-Side Sessions: Store session information client-side, through a 'cookie' for example, which is a collection of user-related session info which is sent with all requests. This might result in security issues, since cookies could be fabricated, etc.
- The amount of traffic that a web application receives is often varied, which makes it difficult to set a fixed, permanent number of servers that should be used. Underestimating traffic could result in servers being overloaded, whearas overestimating could result in wasted resources. 'Autoscaling' is a tool offered by many cloud-computing servers which scales resources based on how much traffic is coming in. Often, a minimum and maximum number of servers can be set, and then a load balancer will take care of the rest.
- 'Autoscaling' and other features are one of hthe benefits of 'cloud computing' which allows web applications to be hosted on remote servers, as opposed to having to set up a server in the actual office of the company who owns the application, for example. Other benefits of cloud computing include not having to worry about IT services, etc. to actually maintain the server.
- If one server goes offline, session data could be lost (depending on implementation), or a user could be continually redirected to an offline server. In order for a load balancer to be aware of the state of the servers, the servers should send a 'heartbeat' back to the balancer at a

known interval. Faster heartbeats allow for a more current idea of server status, but slower heartbeats allowed for save energy and resources.

Cookies

- A simple way to ensure that a user is sent to the same server repeatedly is to give the user a cookie to send b ack which simply tells the load balancer what server the user came from.
- Cookies can also store session information directly. Flask, for example, uses 'signed cookies' which stores all the users session information. These sessions can be something a simple as a dictionary that contains a user ID and any other pertinent info.
- One issue with cookies is increasing size as data becomes more complex. In terms of security, cookies can be stolen to gain access to a user's account. Even without access to a cookie, storing explicitly formatted data (integers for user ids) is easily modified and faked. By including a private key in a web application, cookies can be generated with a signature based on the data and the private key. This signature should be difficult enough to generate that it can be received with reasonable confidence that it's genuine.

Scaling Databases

- Databases can get overloaded in much the same way as servers, especially if a single database is supporting multiple servers. If there is only one database, then that's a 'single point of failure'. If the database drops, the whole system comes down. A load balancer is another example of single point of failure.

Database Partitioning

- Querying large database tables can become complicated and time-consuming. Those large tables can often be split up into multiple, more managable and more efficient tables.
- Vertical database partitioning consists of separating a table by decreasing the number of columns. This has already been seen in the recurring airline example when foreign keys were used to partition a table into a locations table and a flights table.
- Horizontal database partitioning consists of splitting up the rows into logical groups. For example, an entire flights table could be split up into a table for domestic flights and another for international flights. Both of these tables have the same columns, but fewer rows. This partitioning results in faster and more specific queries. This separation does require more code to manage, however. Any column changes now need to be updated on many tables. A query might actually be slower if data from two tables needs to be accessed.
- Database 'sharding' consists of dividing a database amongst separate servers. This can help eliminate the slowdown from having to query two tables at once, assuming those two tables are on separate servers. Joining tables, however, becomes slower.

Database Replication

- Creating multiple copies of the same database allows for the distribution of load.
- A single-primary replication model has a single database which can be both read from and written to. Other, secondary databases, can only be read from. Any writes to the primary database are automatically passed to secondary databases. Data is both replicated and synchronized.
  - Potential issues include race conditions. Single-primary replication is less favorable for applications which expect a lot of writes, since there is still a single point of failure with the primary database.
- A multi-primary replication model allows for any number of databases which can be read from and written to. Any writes are copied to other databases.
  - Similar issues to race conditions can occur. If two users try to register themselves on two different databases, they might end up with the same primary key user ID, which will be an issue when the databases try to update each other. Two databases might try to update the same row at the same time. Whatever they might be, multi-primary replication systems need rules to resolve issues with simultaneous updates.

Caching

- Caching seeks to avoid wasting time performing operations that have already been done before, namely, by taking data and storing it locally temporarily. For a relatively static homepage, for example, it doesn't make much sense to regenerate a page every time a user requests it repeatedly.
- Client-side caching, performed by the web browser, stores files that are likely to be static (`.css` or `.js` files, for example) to be re-used. This saves time and computational energy. Inside an HTTP response, the server adds HTTP headers, one of which might be `Cache-Control: max-age=86400` . This sets the maximum time the page should be cached for (1 day, in this case). After that, the server should be queried again.
  - Issues can occur on both sides of the timeframe. If a page changes sooner than expected, a user might not seethose changes. If a page changes later then expected, resources are waseted querying for the same old page. To circumvent this, an identifier can be associated with the webpage or resource which is modified after any update. In HTML, this is an `ETag` , a long hexadecimal sequence that is

uniquely associated with a version of a resource. If the server sees that this identifier hasn't changed it can send back a '304 Not Modified' response code to indicate the cache is not 'stale'.

- If a cache is serving an entire network, private pages, such as social media pages, shouldn't be cached and accessed by different users. To help with this, a cache can be set as either public or private in the HTTP header.

- Server-side caching adds a cache to the server-side web, such that each server has access to the same cache. Instead of querying a database repeatedly, servers can query the cache and receive a much faster response if the query has been made recently.
  - Any time the database is updated, there is potential for the cache to become stale or invalid. The cache could be updated with any write, but often times it is wiser to employ some logic to only invalidate the cache at certain points. One simple workaround is to simply set an expiration for the cache if the temporary inaccuracy is tolerable.