Lecture 8 Scalability & Security

1. Introduction

Web applications are stored on servers – which is just a dedicate piece of hardware to hold that web app – and those servers respond to user requests.

2 main types of servers

* Cloud
* On Premise

Capacity of 1 server to handle requests depends on how many factors such as (i) type of server – how much RAM, processor etc (ii) how expensive those requests are -> IO bound, poor O(n) timing of algorithm etc

1. Benchmarking

To figure out how many users a server can service we use benchmarking tools. Such as Apache AB

1. Scaling

How to improve the number of users we can service?

1. Vertical Scaling

This simply means swapping out the server for a more powerful one. I.E. increase RAM size or processor speed or cores.

However, limited by hardware cause there is a limit to how much RAM is just available

1. Horizontal Scaling

Increase the number of servers.

However, increases complexity in the system as we need some way to dispatch users to either server A or B.

1. Load Balancing

A load balancing is another piece of hardware that dictates whether requests are sent to server A or B. It sits in front of the servers. This means requests by users hit load balancer before being dispatched to server A or B.

1. How does Load Balancing Dispatch Requests

There are 2 main types (i) Algorithmically (ii) Manually

Algorithmically -> Some examples are (i) Random Generator (ii) Round Robin (iii) Fewest Connections

Manually -> Just have some kind of map in Load Balancer to dictate which requests goes where

However, problem appears with both solutions is how to make sure sessions are kept intact. A certain requirement for web apps is the ability to remember something that a user did in a previous session. This is called sessions in Django. If user directed to server A in first request but server B in second request, then sessions would be lost.

1. Session-Aware Load Balancing

* Stick Sessions -> Just remember which server user was directed to and send them there again. Problem is hard to balance because what if many users tagged to 1 server logs in.
* Sessions in Database -> Just store session data in database then it doesn’t matter which server is directed to
* Client-Side Sessions -> Store sessions on the client side. This means storing session information in cookies. However, exposes security concern. Also sending cookies to and fro client and server can be expensive too.

1. Autoscaling

Certain web applications have a peak and trough pattern to user traffic. I.E. At certain times of the day or when an event happens, many users will seek to use that web app and traffic will spike. For example when breaking news occurs on a newspaper website.

If a company allocate server resource based on projected max number of users, then it means many servers would sit underutilized at most times and this is uneconomical.

Autoscalers are a function of cloud providers that allows an app to dynamically add servers based on traffic flow and have a load balancer automatically distribute requests to new servers.

However, cons exists as well such as it might take some time for new servers to come online and users might experience outage during this time. This problem would not be an issue if you just have extra capacity.

1. Other Pros and Cons of Multiple Servers
2. Increased reliability

If you only have 1 server, you have a single point of failure. I.E. If that server fail you entire application fails. If you have multiple servers, the load balancer can just send request to other servers.

The load balancer figures out whether the server is up or not by implementing a heartbeat process. This is where it pings each server every few seconds and the server will respond with predetermined message if it is active.

However, the load balancer might also be a single point of failure. A common redundancy is to have a second load balancer act as a “hot spare” loaded with all necessary software implement a heartbeat process with the initial load balancer such that it can take over if necessary.

[How would request know to send to new load balancer tho?]

1. Scaling Databases

Databases also need to be hosted on servers and just like web applications, they are limited to how many requests they can handle at any 1 time.

1. Database Partitioning

Split big dataset into multiple parts of that dataset.

**Vertical Partitioning**

This practice already implemented where instead of having 1 big table, you have multiple tables in SQL. Think of flights and airports. This method is called vertical partitioning where you store different types of data into different datasets.

**Horizontal Partitioning**

Horizontal partitioning is when you store the same type of data in different tables based on some other characteristics. For example, we can store flights in different tables depending on if they are international or domestic flights.

1. Database Replication

Databases are still single points of failure. If database server crashes, all data could be lost. Therefore a common practice to create copies of database to make sure there is no single point of failure. There would be replication of data + a backup could take over if the primary database failed.

There are different methods are replication. They are:

**Single-Primary Replication**

One database is considered primary while the other databases are backup. This means we can read and write to the primary DB but can only read from the other DBs. When primary DB updated, the other databases replicate the primary DB.

The cons of this method is that there is still a single point of failure when it comes to writing to the DB.

If your bottleneck on DB performance is on writes, then this won’t solve your problem because all users still are writing to that primary DB.

**Multi-Primary Replication**

All databases can be read and written to. Solves the problem of single point of failure. However, creates a tradeoff as it is now much more difficult to keep all DBs up to date with one another because each DB must now be aware of changes to all DBs.

Also creates the new problem of conflicts:

* Update conflict -> Occurs when 1 user attempt to edit a row in 1 DB whgile another use attempts to edit that same row in another DB creating a problem when the DBs sync
* Uniqueness Conflict -> Every row in a SQL DB must have a unique identifier and we may run into the problem that we assign the same ID to 2 rows in different DBs.
* Delete Conflict -> When one user may delete a row while another user attempts to update it

1. Caching

Databases queries are expensive performance wise. If every time a user require some information from the backend they had to query the DB, it would quickly cause performance issues. Therefore, if we anticipate that the user would require some information multiple times in the near future AND that the information does not change often, we can use caching to fix this problem

Caching refers to the idea of storing data in some accessible location such that if we need it in the near future we can query that location instead of calling an expensive database query.

Types of Caching

1. Storing in user’s web browser [Client Side Caching]

When user loads certain pages, no request to server needs to be sent. Simply, take information from browser. A common method to do so is to include this line in the header of a HTTP response.

Cache-Control: max-age=86400

Tells browser that as long as the user has visited this page within the last 86400 milliseconds, no request has to be made to the server.

Example would be when you refresh a page and that page contains an image. This command would tell the browser to load the image from cache instead of querying the server.

Problem is of course that if the resource changes, then the user would not be able to see it for duration of max-age.

A way to update this approach is with an E-Tag. An E-tag is basically an ID attached to some resource such as JS file or CSS file. The server would tell the browser what E-tag for that resource it intends to server the user when queried and if the E-tag does not match, the browser would query the DB.

1. Server Side Caching

This is when the servers that hold the application communicates with a cache instead of the database server for certain queries. Django has a cache framework specifically for this.

Types of Caching

**Per-View Caching**

Means for a specific view, if a user had requested that view in the past x time, then don’t run through the whole Python code, just serve the previous result.

**Template Fragment Caching**

This caching can apply for specific parts of a template as well instead of the whole template if the end result of the view is to serve a template.

**Low-Level Cache API**

Django gives a low level cache API that if there is any information you want to cache you can save in that API. Then you can access that within your Django app.

**Security**

1. Introduction

Security Concerns are everywhere

1. Git

Git allows anyone who has access to a repo to see all versions of that code. So don’t commit credentials

1. Phishing Attacks

Basically when HTML anchor tag that takes user to another link and not the one that is actually displayed.

1. HTTP & HTTPs