A creative book by Sweet Codey

System Design Playbook

Shortcut to Interview Success

Beginner's Guide

www.sweetcodey.com

System Design Playbook

A Beginner's Guide

Authors:

Suresh Gandhi

Rohit Jain

Shubham Chandak

www.sweetcodey.com

Copyright

For more information, contact hello@sweetcodey.com

The authors

Names:

Suresh Gandhi	Software Engineer at Microsoft
Rohit Jain	Software Engineer at Amazon
Shubham Chandak	Software Engineer at Bloomberg

About the authors:

Three friends from IIT Kharagpur—Suresh, Rohit, and Shubham—reunited in the USA after forging a strong bond over their shared passion for technology during their undergraduate years. Each has since excelled at major tech companies; Suresh at **Microsoft**, Rohit at **Amazon**, and Shubham at **Bloomberg**.

Together, they bring a wealth of experience in building scalable systems that serve millions. Suresh dives deep to simplify complex topics, Rohit brings ideas to life with his practical approach, and Shubham specializes in optimizing user experience and ensuring top-tier quality. Their combined expertise and unique perspectives make their teachings particularly insightful and accessible.

Preface

We've noticed a significant gap in resources that explain system design simply and visually. This book aims to fill that gap with clear diagrams and easy-to-understand explanations.

Whether you're just starting to learn about system design, preparing for interviews, a product manager wanting to grasp technical concepts, or simply curious about the topic, this book is perfect for you. **We've designed it to be visually engaging and concise, making complex ideas accessible to everyone.**

Each chapter is divided into subsections that explain relevant concepts and buzzwords. By the end of the book, you'll feel comfortable with system design basics and ready to dive into more advanced topics.

We hope this book serves as a valuable resource and inspires your curiosity and passion for technology.

Suresh, Rohit, and Shubham

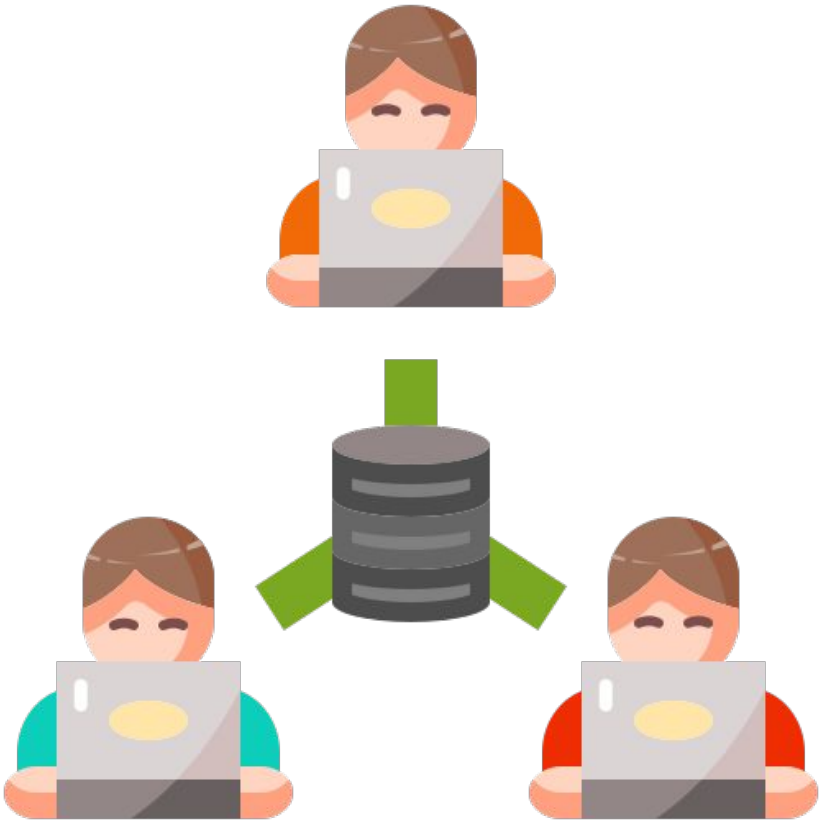
Contents

Chapter 1

The Ultimate System Design Template

The Ultimate System Design Template

- Step 1: Client & Server
- Step 2: Database
- Step 3: Vertical & Horizontal Scaling
- Step 4: Load Balancer
- Step 5: Database Sharding & Replication
- Step 6: Cache
- Step 7: Content Delivery Network
- Step 8: Monolith & Microservice
- Step 9: Message Queue
- Step 10: API Gateway



Step 01

Client & Server

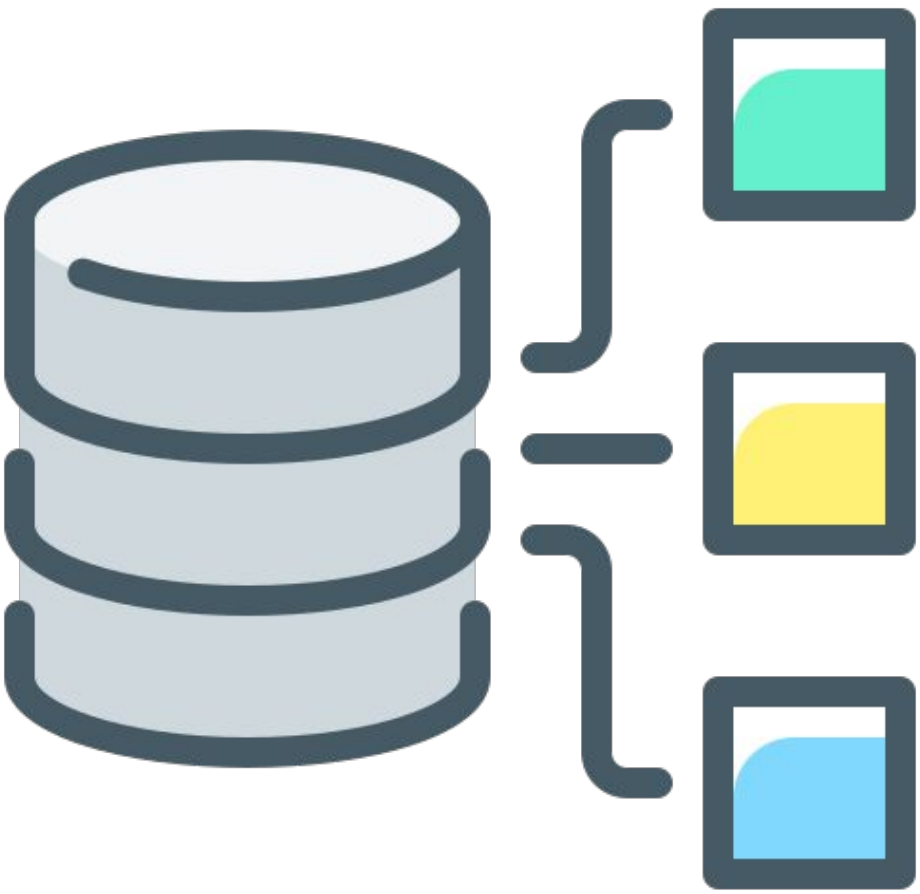
Client & Server

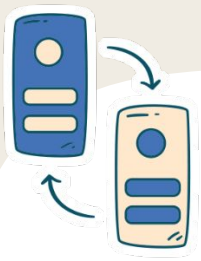
- You open the browser. You type the website address. The website loads.

- There are two main elements involved in the whole process - client and server.
- Your mobile/computer is the **Client** as it requests to view the webpage.
- The computer where the webpage is stored is the **Server**. It takes client's request and returns the webpage.

Step 02

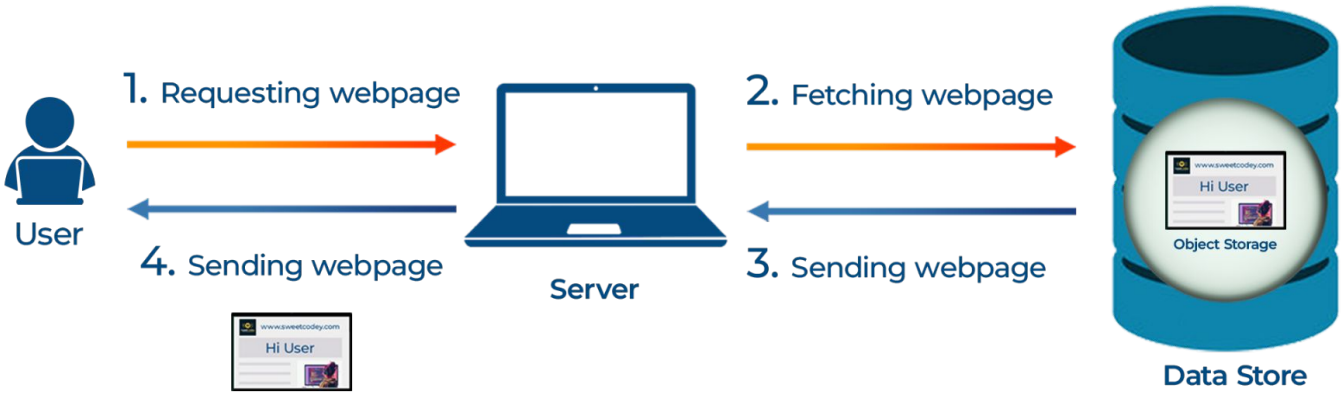
Database

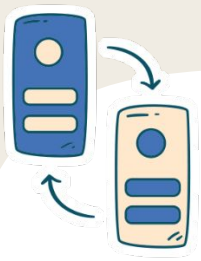




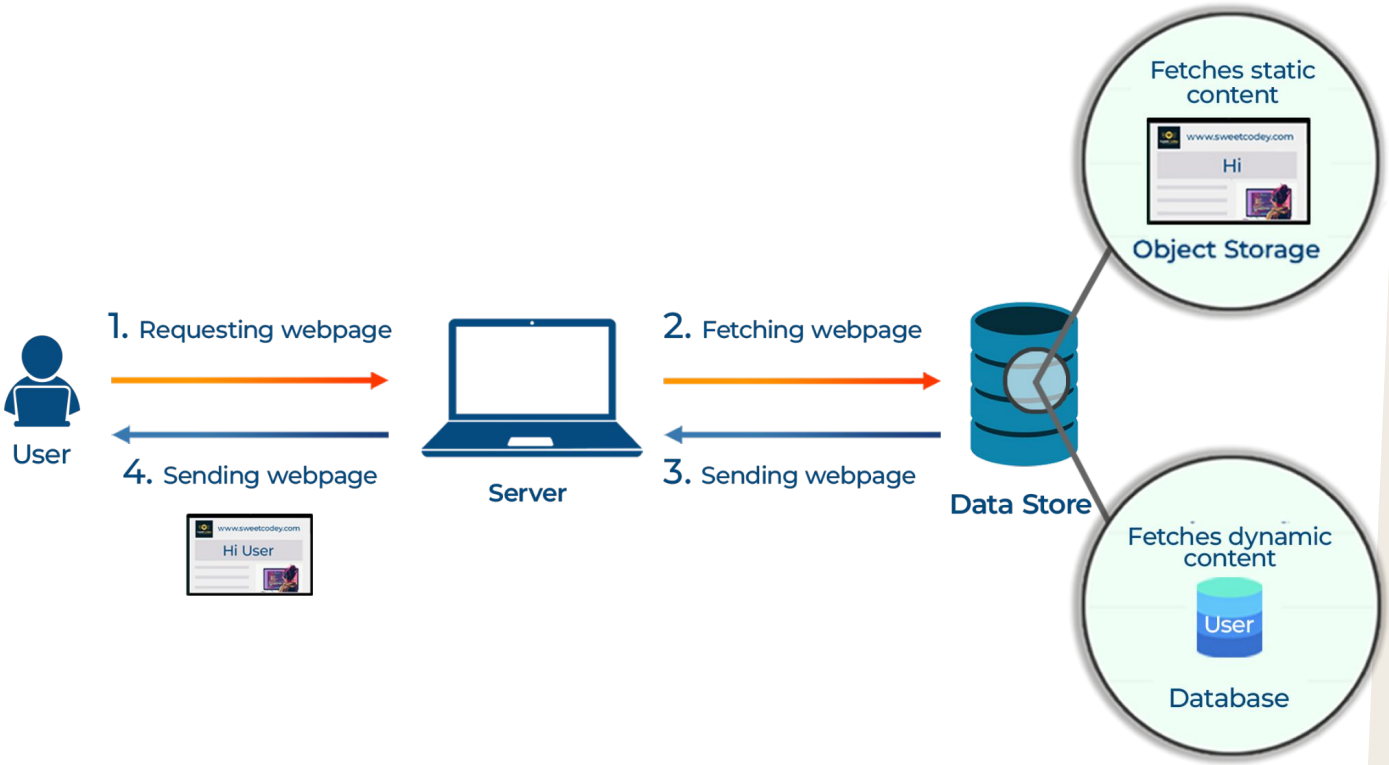
Database

- Lets break down how the server returns the web-page.
- Server first understands what the client is requesting for.
- Based on the request it fetches the data from the Data Store. In this case the web-page data is fetched from the Data Store.
- Note: Think of Data Store as data layer that handles everything related to storing data.





- Data Store includes Database (where dynamic data is stored) and Object storage (where static data like HTML, files, images are stored).

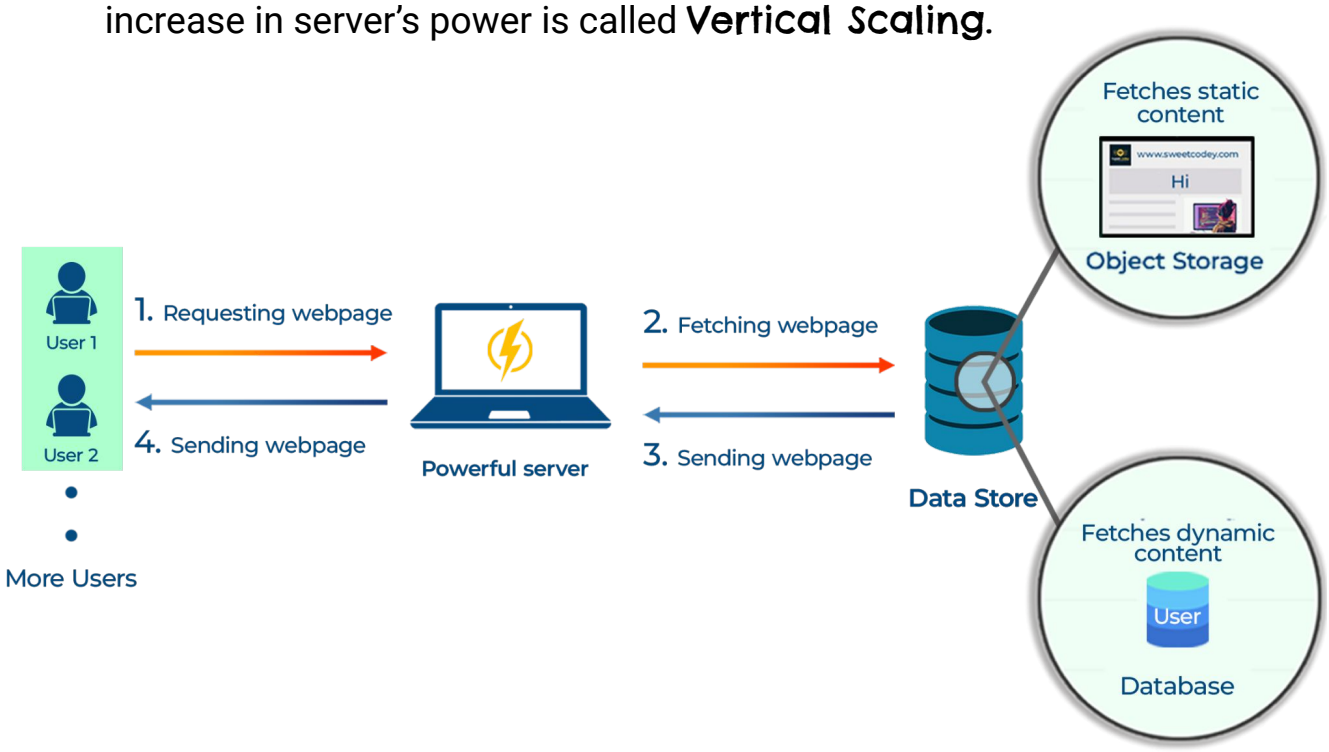


Step 3

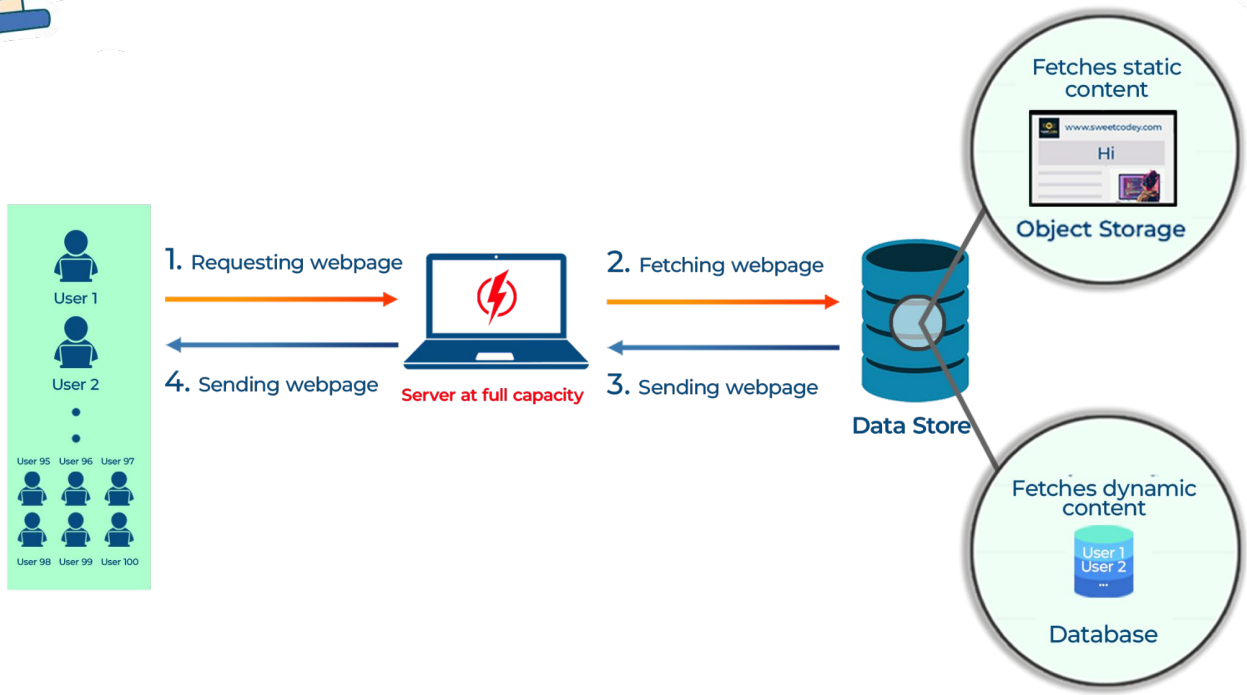
Vertical & Horizontal Scaling

Vertical & Horizontal Scaling

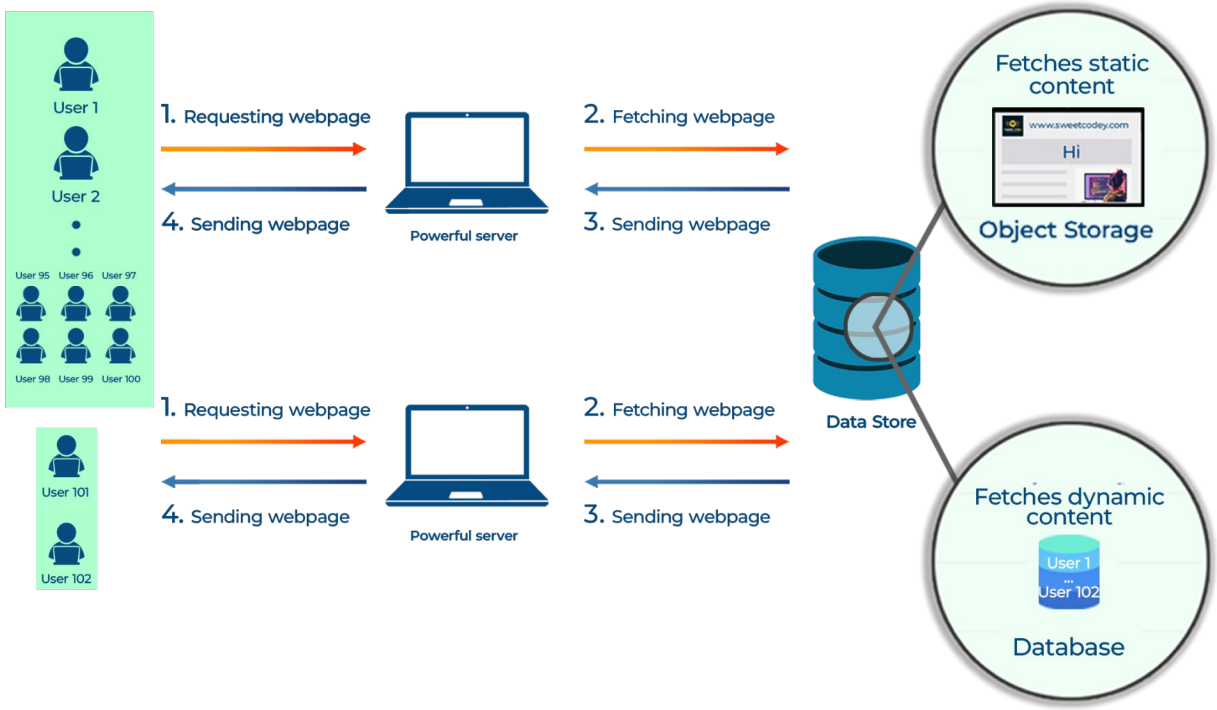
- More and more people are visiting sweetcodey.com.
- Because of this, both the server and database are overwhelmed and are struggling to keep up.
- Lets see how we can solve the 'server getting overwhelmed' problem first.
- To solve this problem, we increase the server's power (CPU, RAM). This increase in server's power is called **Vertical Scaling**.



- Boosting the server's power helped initially. But now, even more people are visiting sweetcodey.com 😊
- Our powerful server has reached its full capacity and couldn't handle any more.

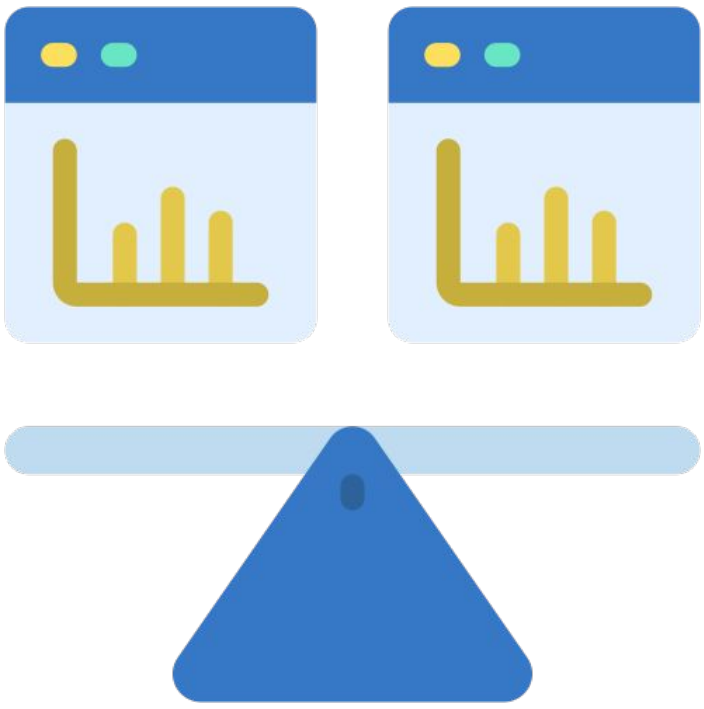


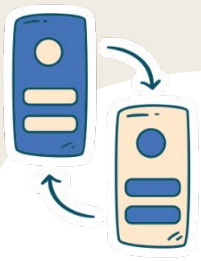
- We need more powerful servers to handle them. One is not enough.
- We therefore add more such powerful servers.
- This increase in the number of servers is called **Horizontal Scaling**.



Step 04

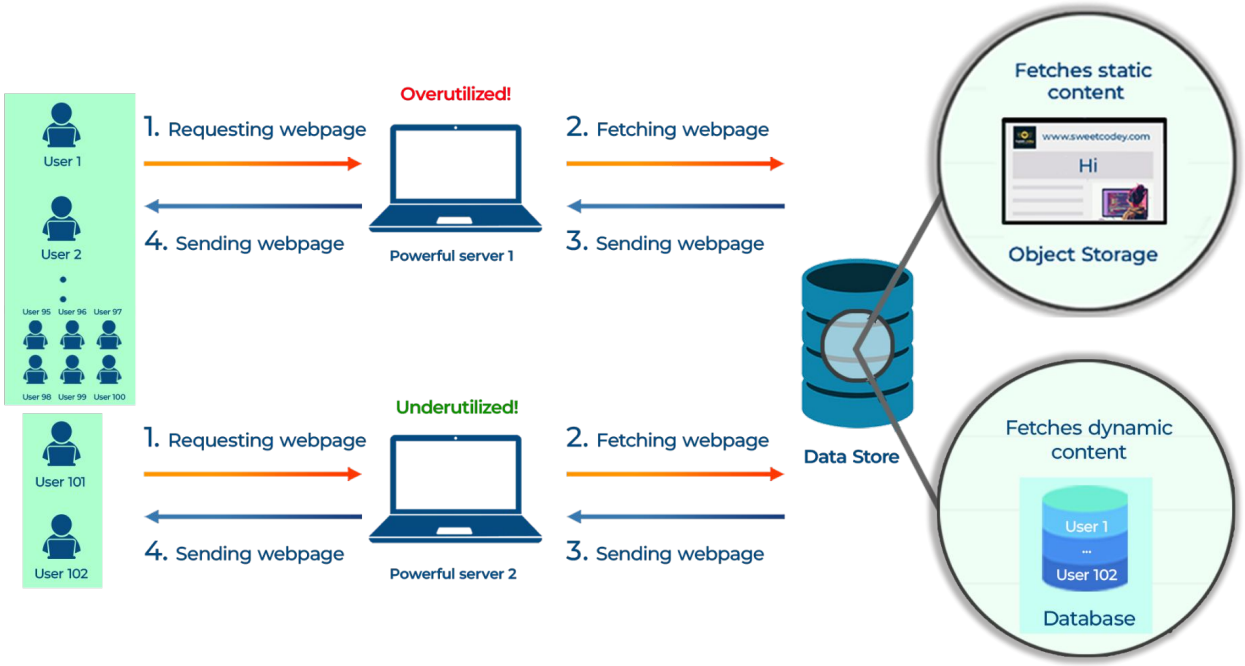
Load Balancer



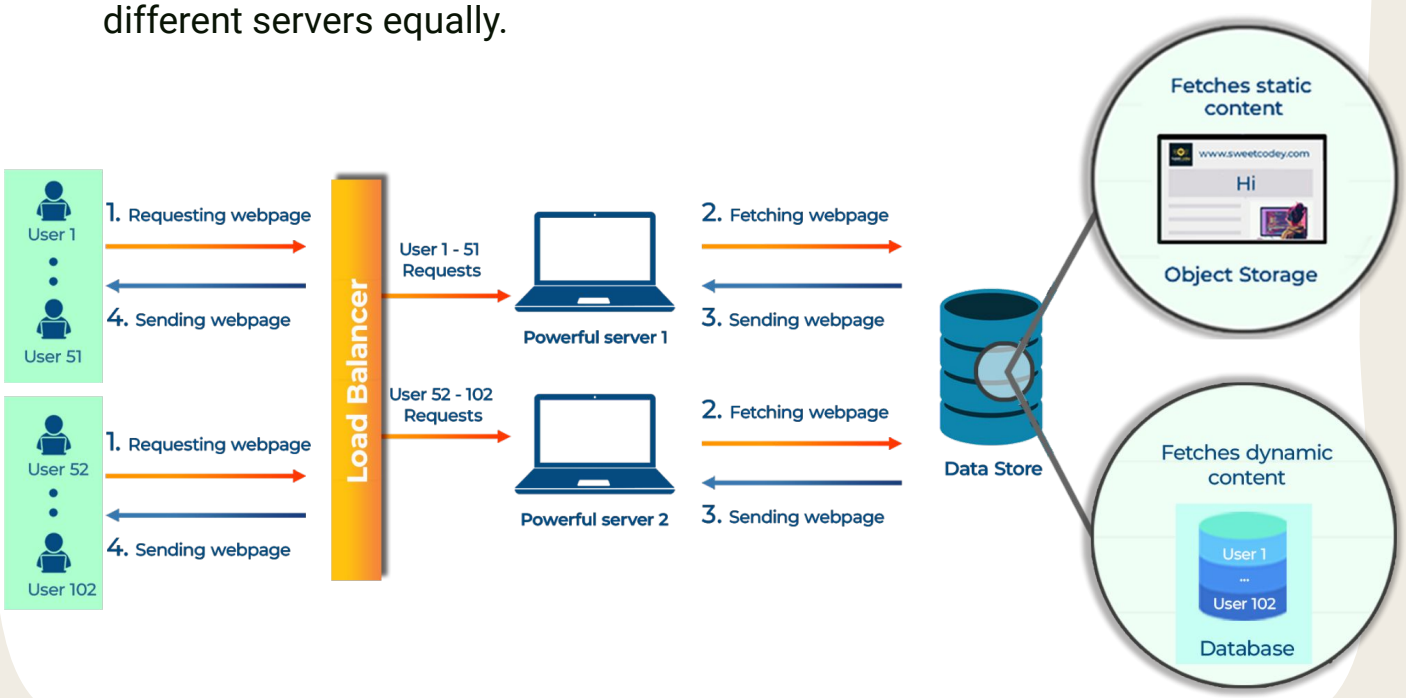


Load Balancer

- Looks like Server 1 is over capacity and Server 2 is under utilized.

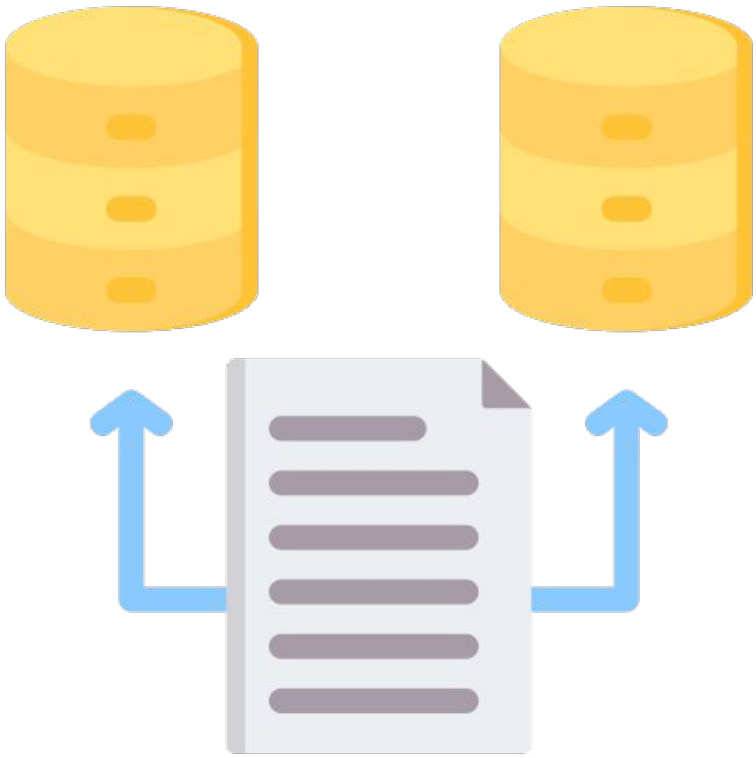


- Let's fix that using a Load Balancer.
- **Load Balancer** balances the load and distributes the requests to different servers equally.



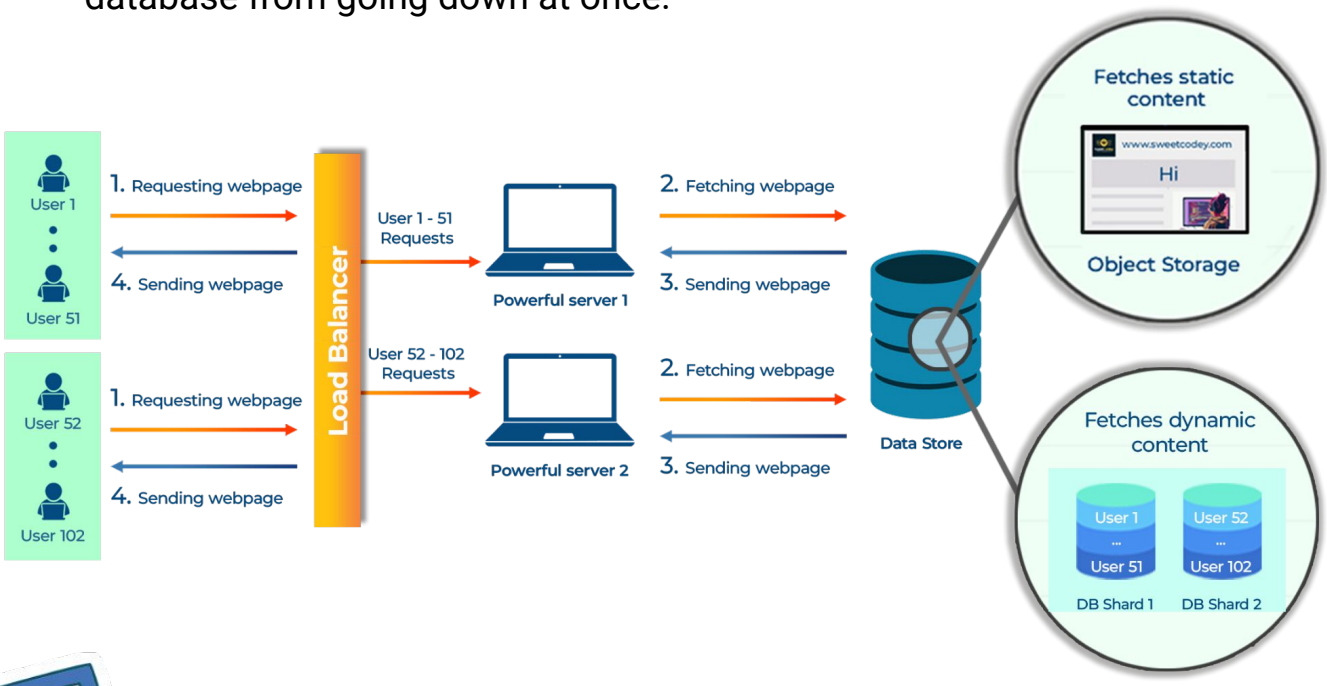
Step 5

Database Sharding & Replication

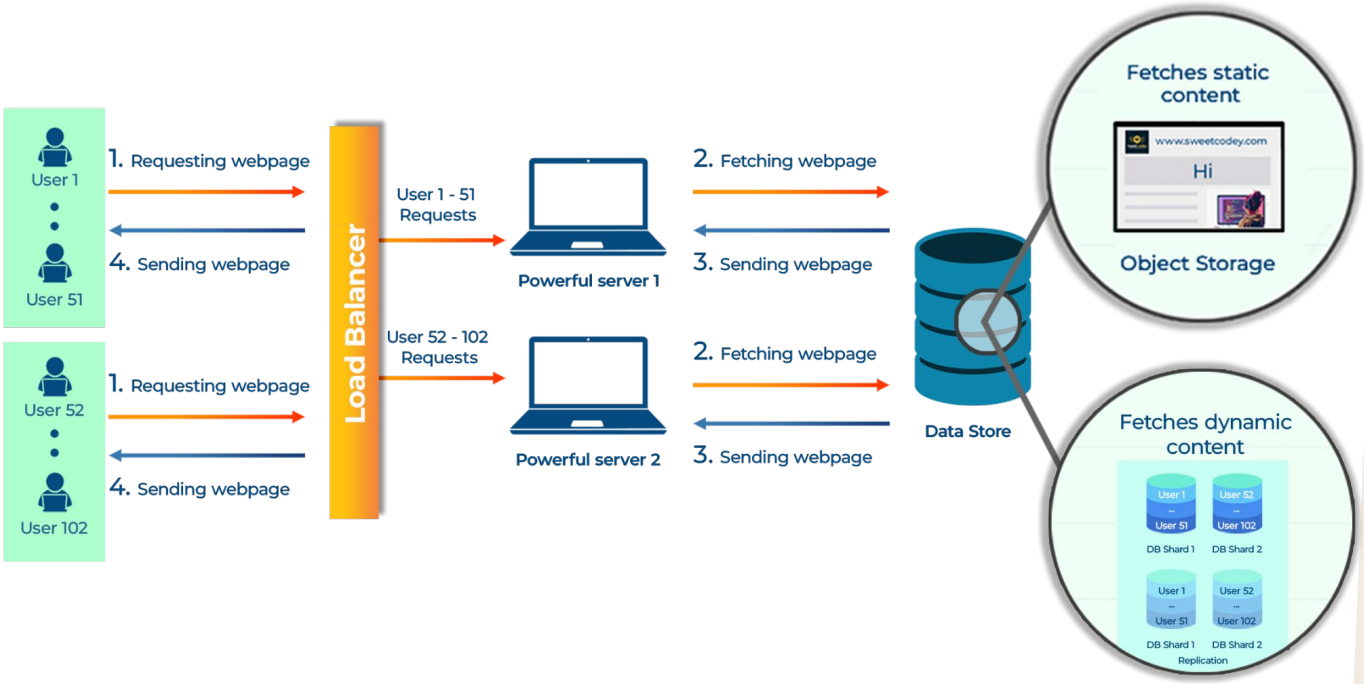


Data Sharding & Replication

- We have solved the ‘server getting overwhelmed’ problem. Now, lets see how we can solve the ‘database getting overwhelmed’ problem.
- The problem is we just have a 'single' database that is handling all the user operations. With more and more users visiting sweetcodey.com, this single database is getting burdened.
- We can solve this problem by splitting our single database into several smaller databases.
- Each one will hold a different part of the user data, called as a shard. This is called as **Database Sharding**.
- Now we don’t have the ‘single’ database overburdened problem. Also, if one shard has issues, the others keep working. This prevents the entire database from going down at once.

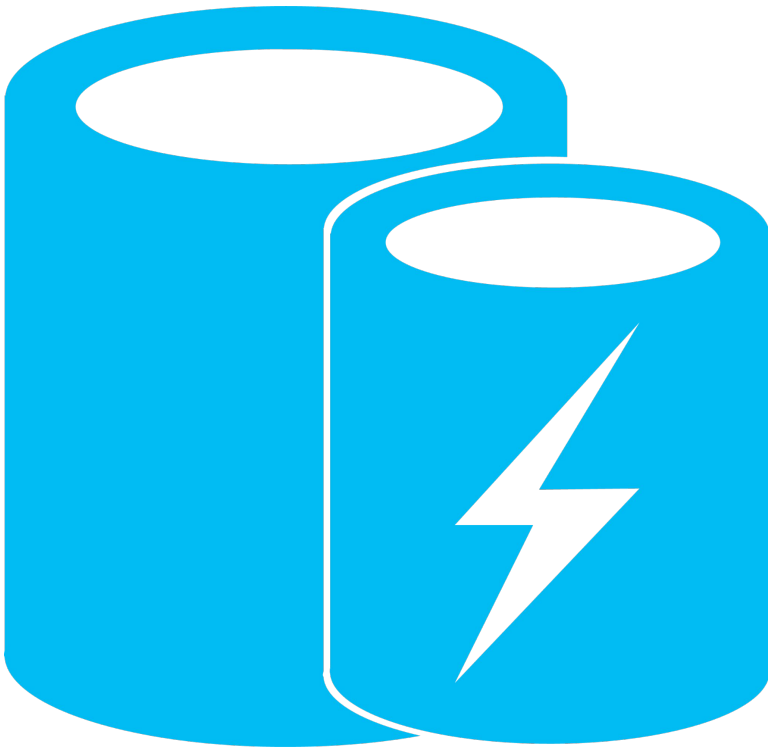


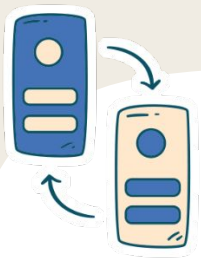
- But what if one of our database shard crashes. We will lose all the data from that shard. How do we deal with it?
- We simply replicate our database shards. This is known as **Database Replication**.
- Now when a database shard crashes, we can replace it with its replica.



Step 06

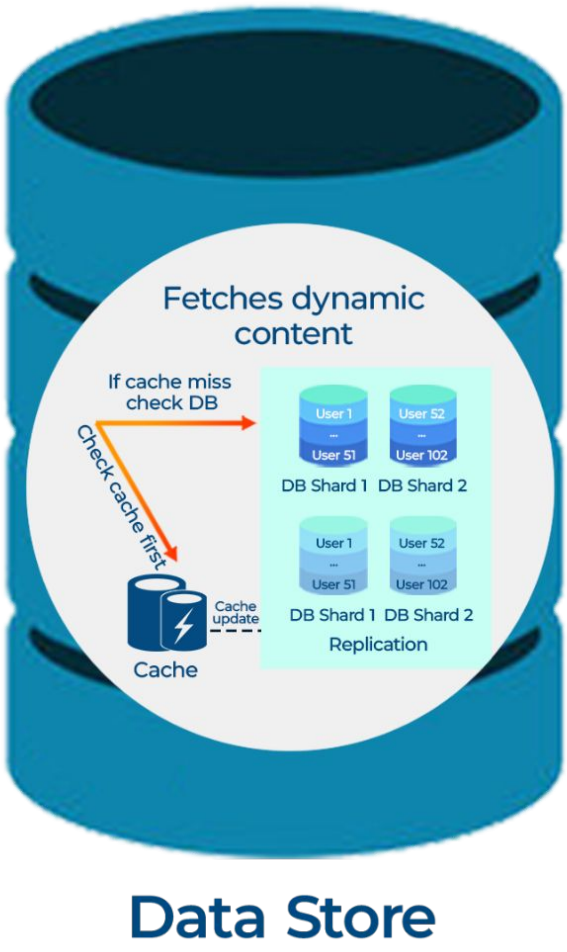
Cache





Cache

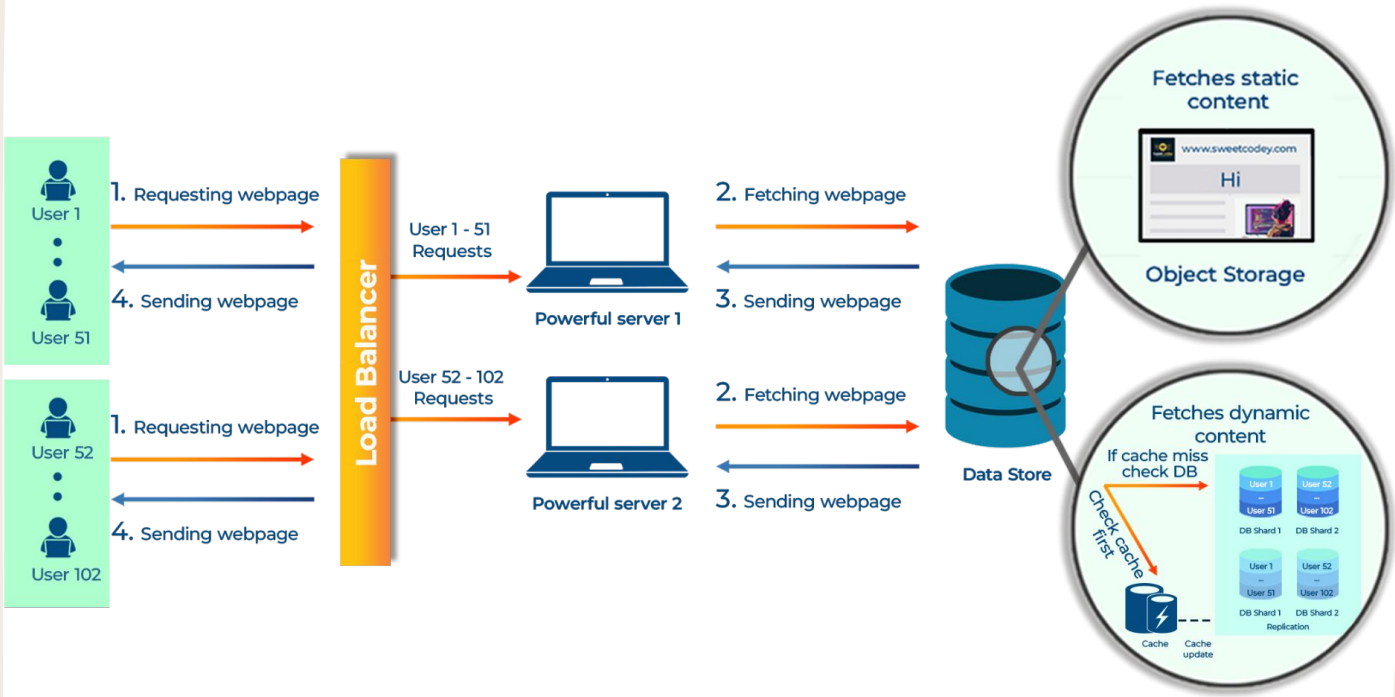
- Now User1 really starts liking sweetcodey.com. He visits the website 100 times in a day.
- This means everytime User1 visits the website, User1's data needs to be fetched from the database. This means we are asking the database for the same data over and over again. Feels repetitive?
- To solve this, we use a **Cache**. The cache is like a 'quick-access' memory that stores information that people ask a lot.
- Fetching from a cache is much faster than a database. One analogy to understand this - picking a book from your bedside table (Fetching from Cache) vs going to the library and borrowing it from there (Fetching from Database).

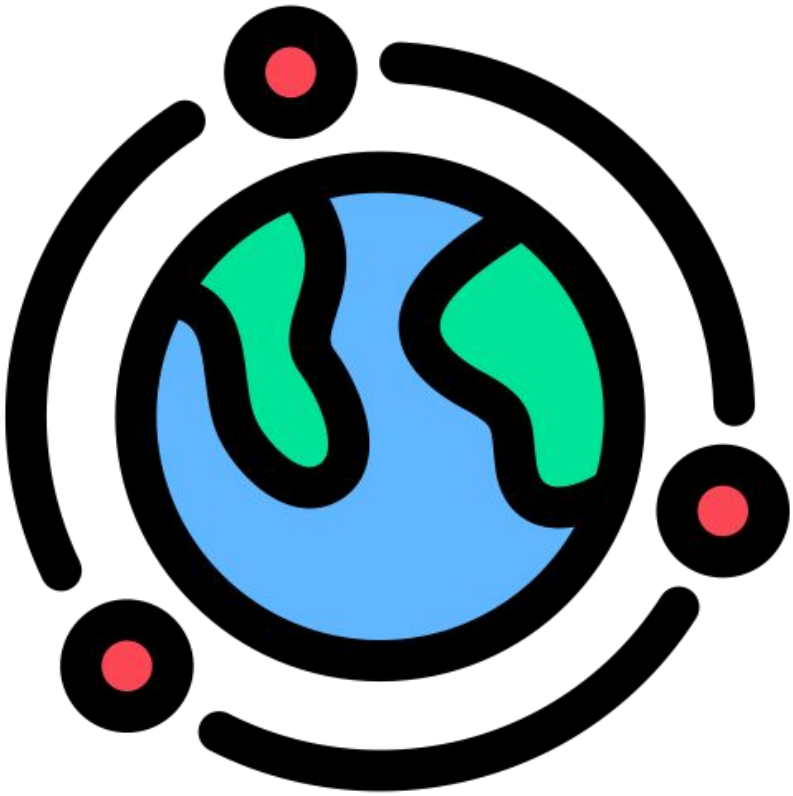


The flow becomes as follows:

1. First we look for the data in the cache.
2. If it's not in the cache, we retrieve it from the database.
3. We then save this data back in the cache, so next time it's needed, it can be accessed much faster.

- The overall flow looks as follows -



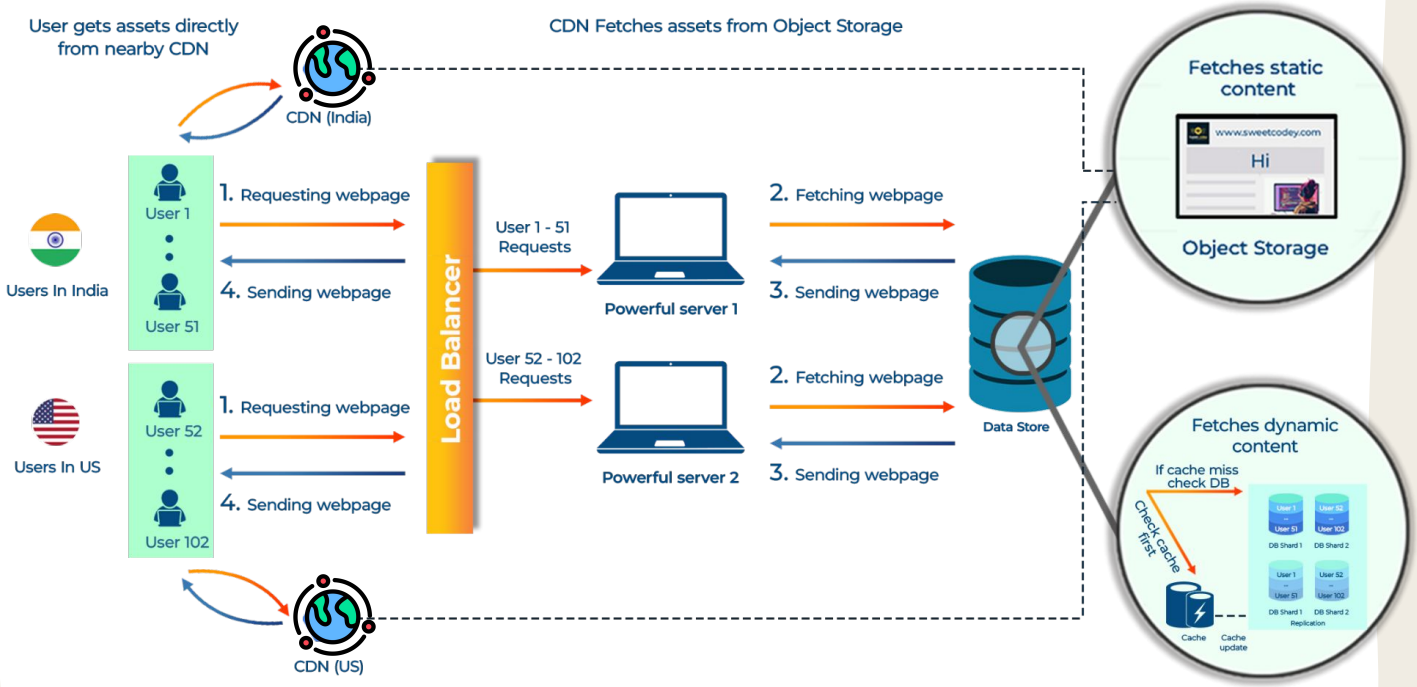


Step 7

Content Delivery Network (CDN)

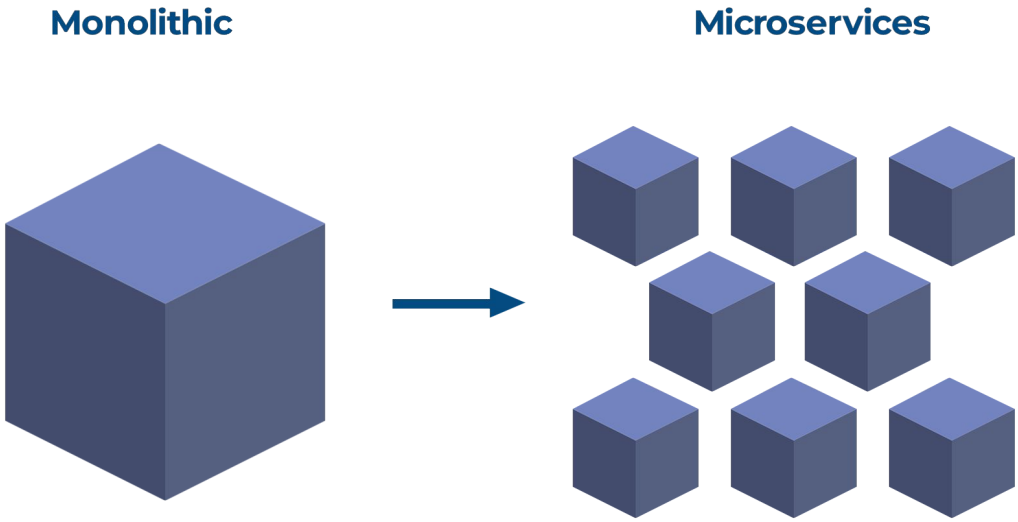
Content Delivery Network

- Now, let's say Sweet Codey has all its servers in the USA. A user from India tries to open sweetcodey.com.
- The website assets (Images, Videos, etc.) are bulky content. This bulky content will have to travel a long distance. This will increase latency a lot.
- A **CDN (Content Delivery Network)** comes handy in this case.
- It stores copies of your website's static content (static content = the data that doesn't change too often) at various locations around the world.
- Now, the user can quickly access static content (images, videos, etc.) directly from a CDN server closer to them.



Step 08

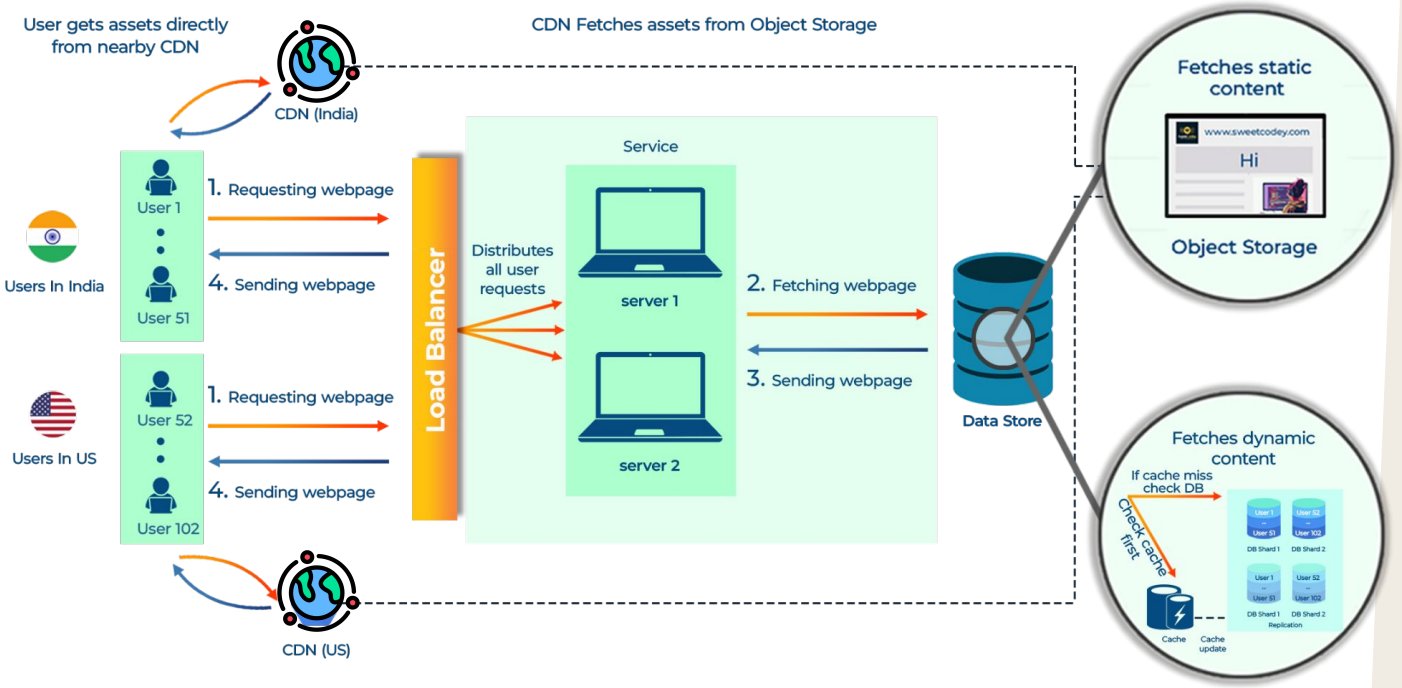
Monolith and Microservices



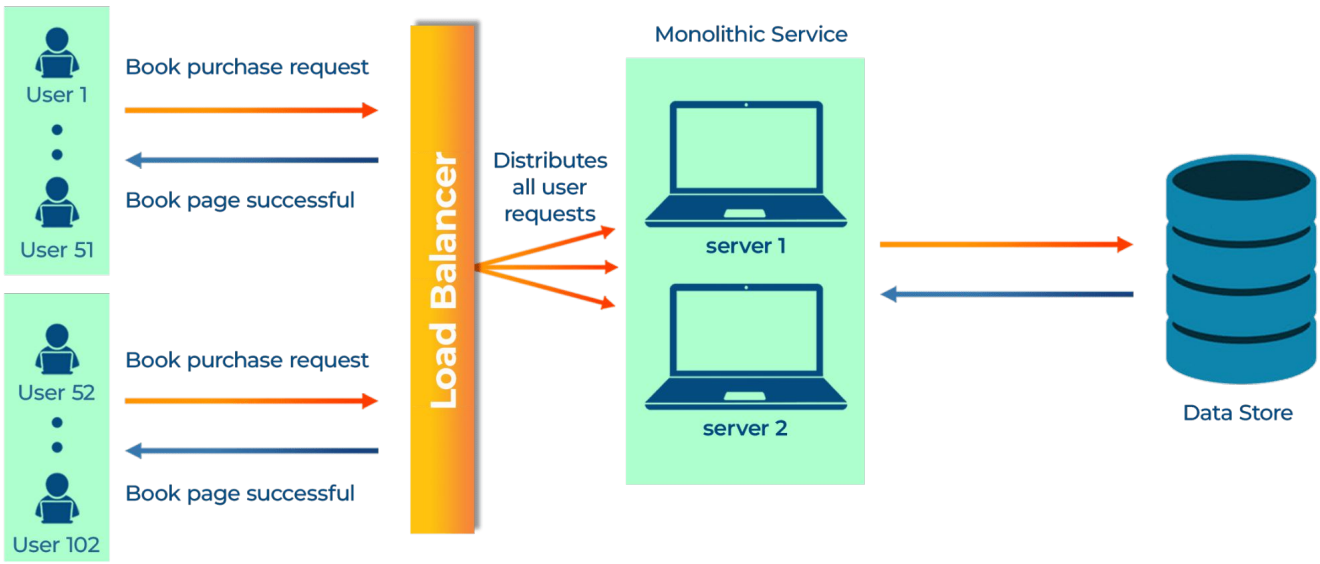
Monolith & Microservices

- Before we proceed further, let's first try to understand what a 'service' is.
- **Service** is a set of servers which specializes in handling a specific task.

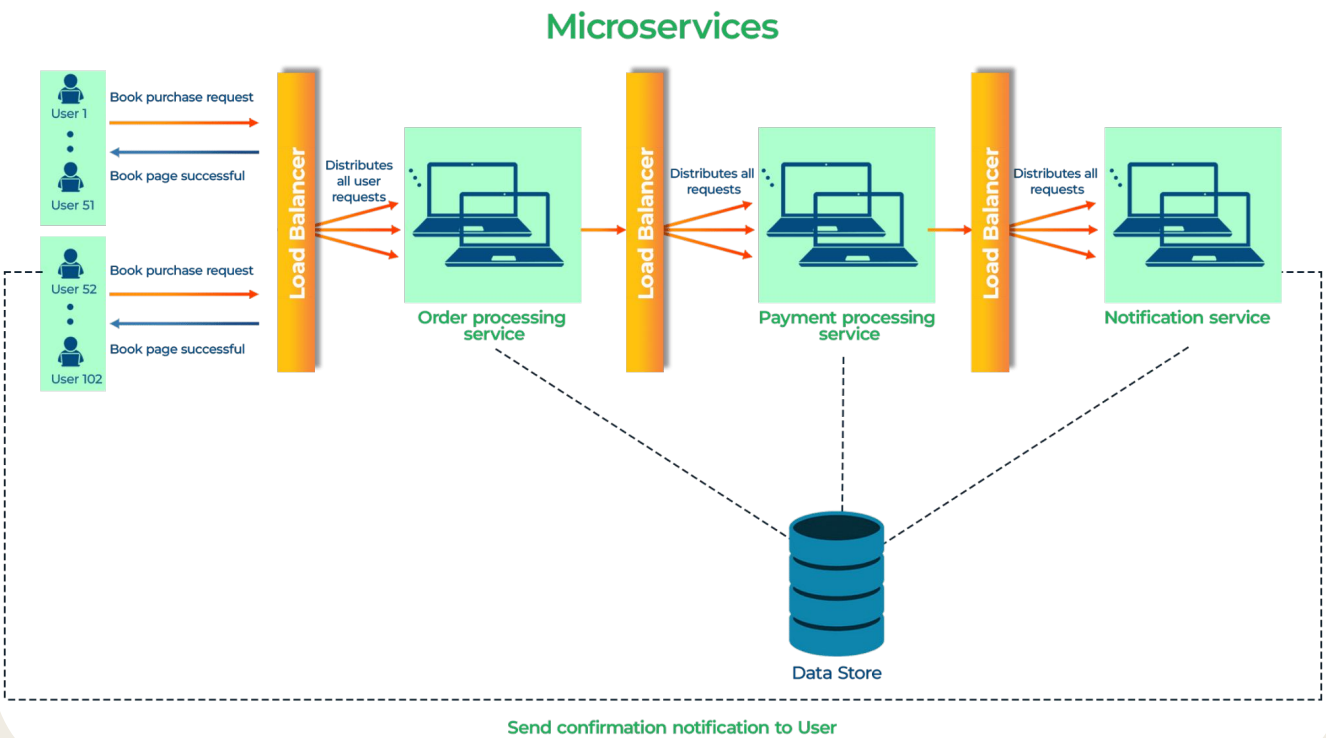
  Example: Set of servers handling user payments.



- Now, lets say you try to buy a book on sweetcodey.com. There are 3 separate tasks that needs to be completed by the servers:
- Take your order
- Process your payment
- Send confirmation notification
- Now, we could have only one service do all these tasks - '**Monolithic Service**'.



- Another approach could be having three different services dedicated to do these tasks individually. Order Processing Service, Payment Processing Service and Notification Service. We call this system '**Microservices** '. Each service has its own load balancer to evenly distribute the load on the servers.

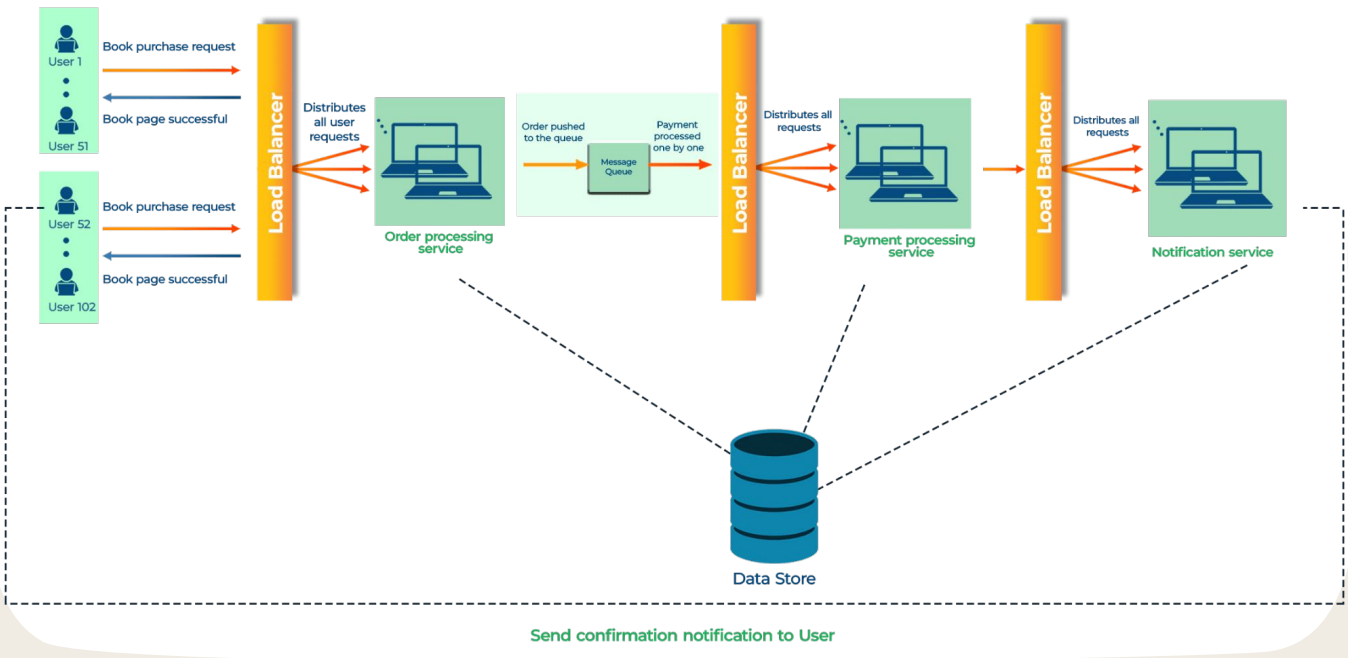


Step 9

Message Queue

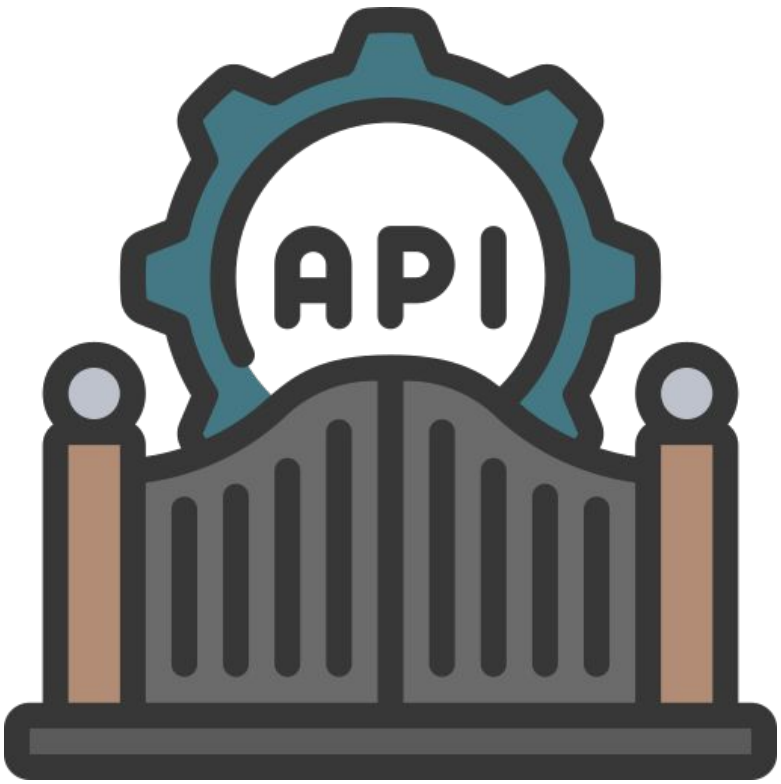
Message Queue

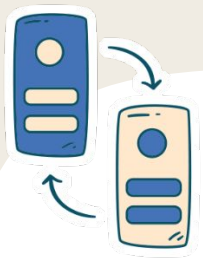
- Now, we run into a problem here - there are a lot of users placing book orders.
- The Order Processing Service is receiving many orders. Processing payment for one order takes time.
- If the Order Processing Service waits for Payment Processing Service to complete payment for that request, it cannot move to the next order.
- This causes delays and spoils the user experience.
- **Message Queue** solves this problem.
- The Order Processing Service pushes the order into the message queue and forgets about it.
- The Payment Processing Service takes messages from the queue and processes payments for the requests one by one.
- Now, the Order Processing Service doesn't need to wait for payment processing before handling new order requests.
- This 'decouples' (makes independent) the two services, making the system more efficient.



Step 10

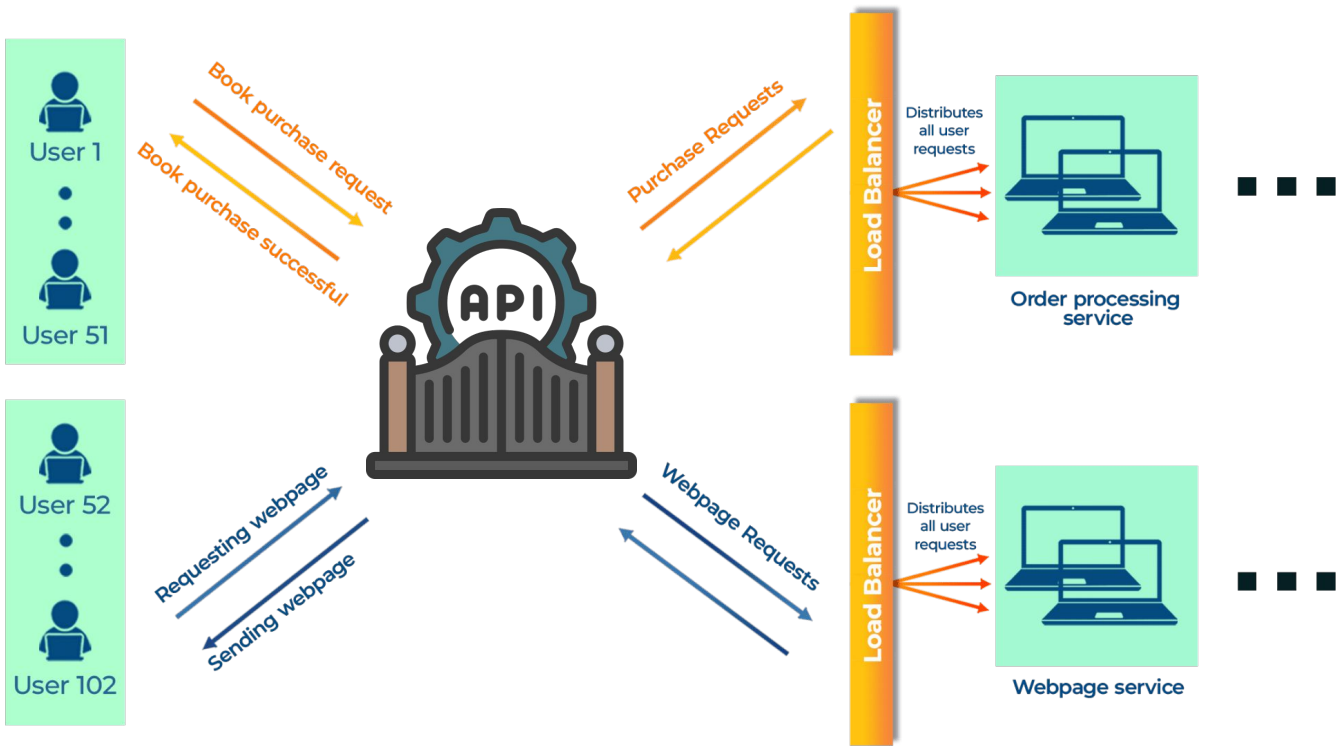
API Gateway





API Gateway

- Now, we run into another problem. Users are making different types of requests.
- Some users are placing book purchase requests, while others are requesting web pages.
- Without a proper system, managing these different types of requests can become chaotic.
- We can use an **API Gateway (APIG)** to handle this problem.
- The API Gateway acts as a single entry point for all user requests.
- All requests go through the API Gateway first.
- The API Gateway then routes the purchase requests to the Order Processing Service and webpage requests to the Webpage Service.
- This helps manage and distribute different types of requests efficiently.



Chapter 2

Design Goals

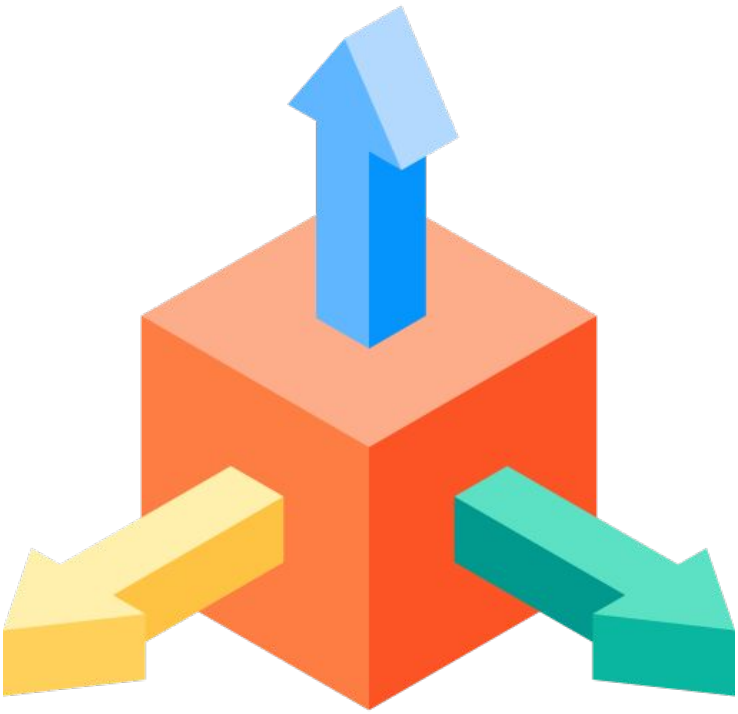
Design Goals

01 Scalability

02 Availability

03 Consistency
(Strong & Eventual)

04 Fault Tolerance &
Single Point of Failure

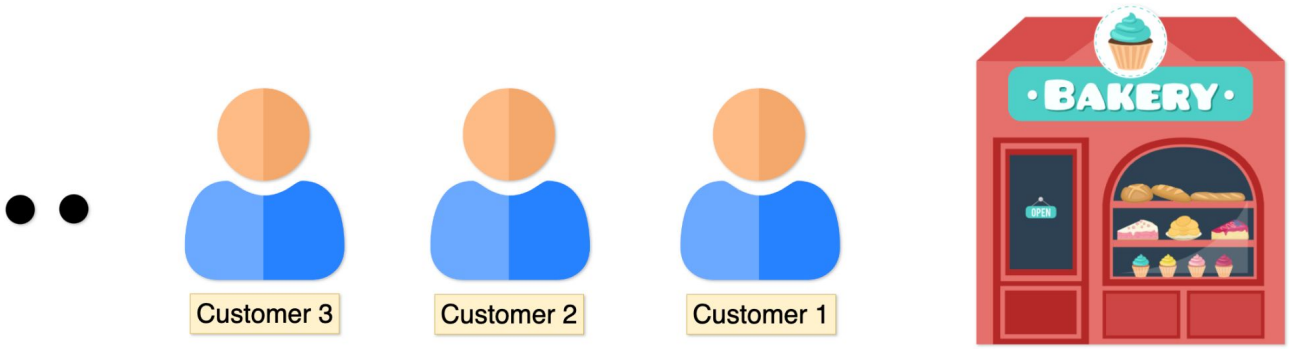


Step 01

Scalability

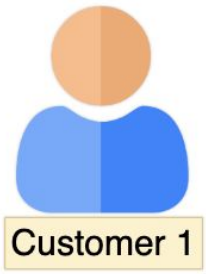
Scalability

- Imagine a local bakery that initially handles its customers with just one cashier.
- Now the bakery becomes more popular.
- Because of that, the line of customer grows longer, and waiting times increase

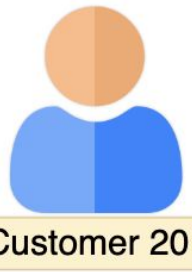


- To serve more customers, the bakery opens additional checkout counters helping them to handle growing crowd efficiently.
- Basically our bakery has ‘scaled up’ to meet increased demand. Just like the bakery, our technical systems also need to ‘scale up’ as more users join.
- A well-designed system can scale up more easily.
- Scalability is the system's ability to handle more work smoothly as demand grows.


BAKERY
OPEN
Counter 1
Counter 2
Counter 3

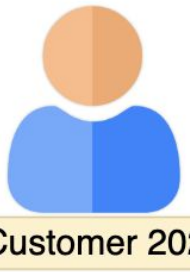

Customer 1

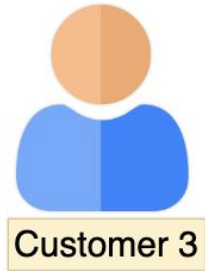

Customer 101


Customer 201


Customer 2


Customer 102


Customer 202

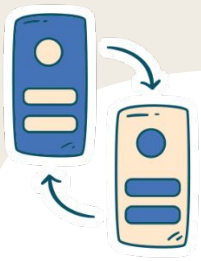

Customer 3


Customer 103


Customer 203

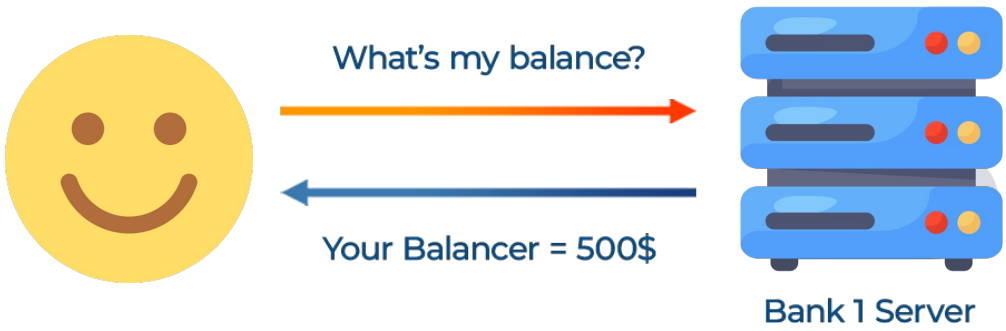
Step 2

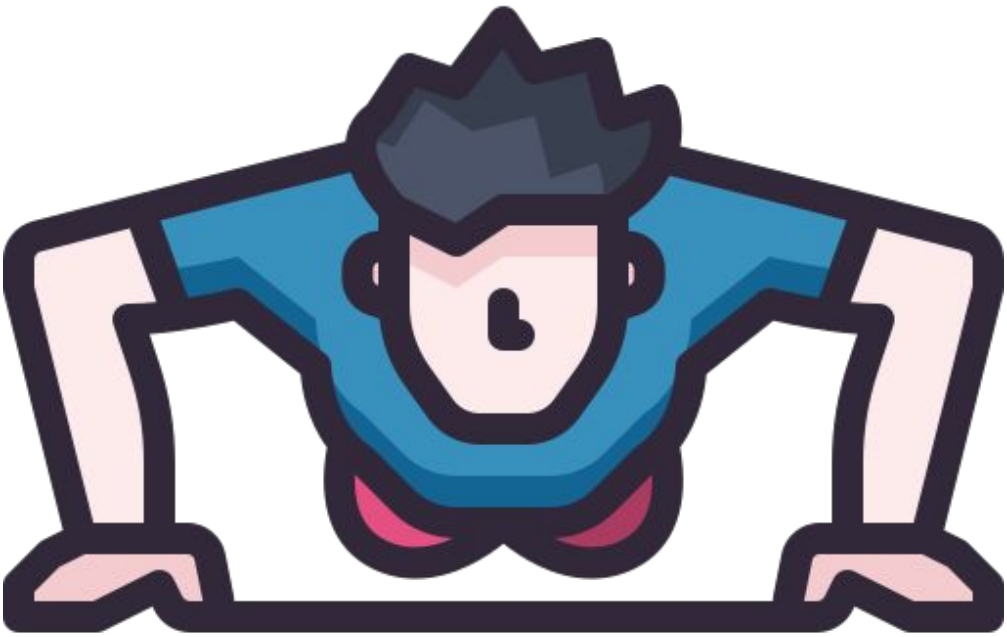
Availability



Availability

- **Availability** means how much time a system is up or operational.
- For example, an online banking website that is available 24/7 ensures that users can access their accounts and perform transactions at any time.
- A system which is available 99.999% of the time also known as "five nines," means it is only allowed 5 minutes (0.001%) of downtime per year.



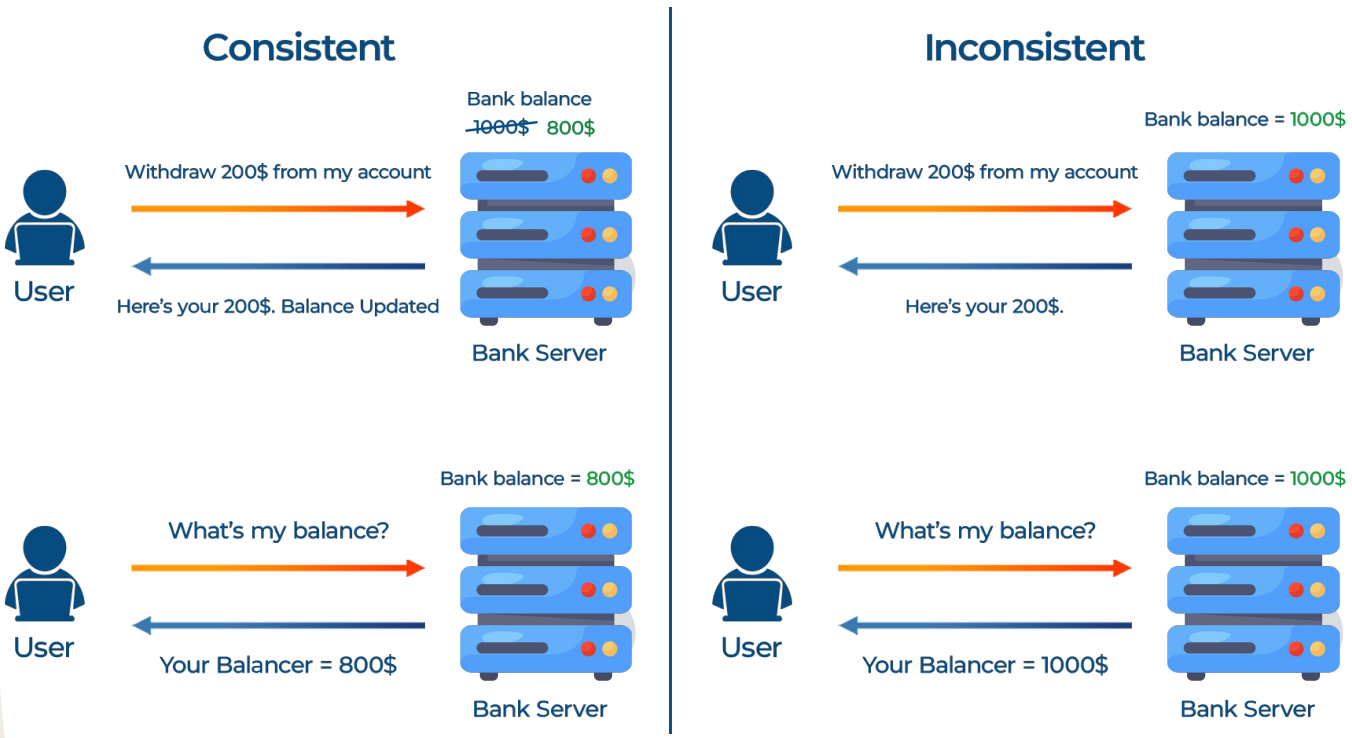


Step 3

Consistency

Consistency Strong/Eventual

- **Consistency** means consistent/same data visible to everyone.
- For example, if you update your profile picture, every user sees the updated picture, nobody sees the old one.
- Ensuring consistency is important for applications where we need up-to-date information. One good example is financial transactions.
- When you withdraw money from the bank, it's essential that the updated balance is immediately reflected, so the same money isn't withdrawn multiple times.

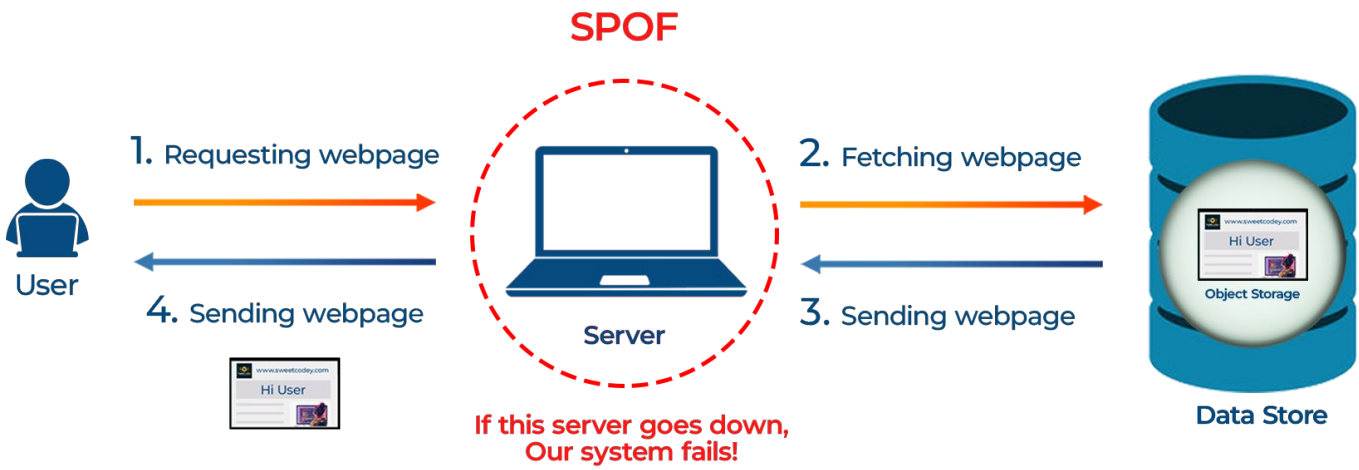


Step 4

Fault Tolerance & Single Point Of Failure

Fault Tolerance & Single Point of Failure

- Let's assume a very simple system where a user is trying to open sweetcodey.com.
- There is a server which handles the client's request and a data store which keeps the site data.
- We can clearly see that if this server goes down the website will be inaccessible. Here, this server is SPOF.
- SPOF (Single Point of Failure) is a component in a system that, if it fails, will stop the entire system from working.



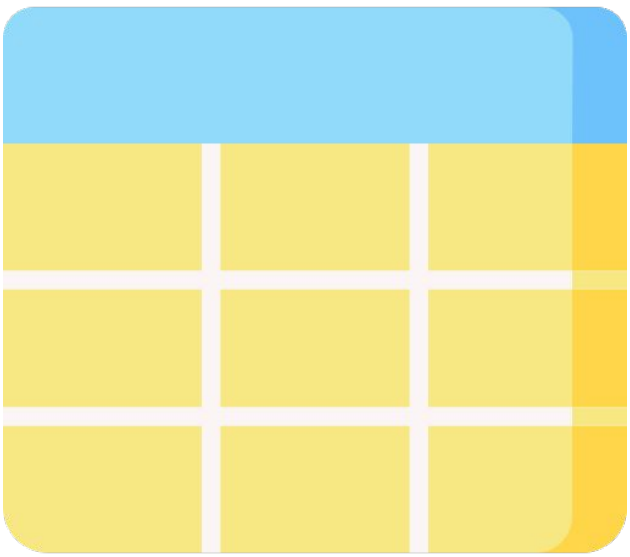
- **Fault Tolerance** means a system's ability to continue operating properly even if some of its parts or components fail.
- For example: If one server in the data center fails other server takes over. If one data center fails, the other data center takes over.

Chapter 3

Buzzwords Database

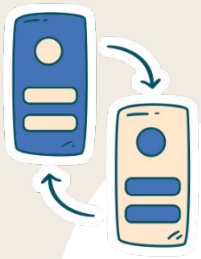
Buzzwords Database

- Step 1: SQL Database
- Step 2: NoSQL Database
- Step 3: SQL vs NoSQL
- Step 4: Object Storage
- Step 5: Database Sharding & Replication
- Step 6: Cache
- Step 7: CDN



Step 01

Relational Database / SQL Database



Relational \ Database / SQL Database

- Stores data in tables, which are like spreadsheets with rows and columns.
- Ideal for the data that has a well structured format like User Data.
- User data is structured because it is organized into predefined fields like name, email, phone number, and address.
- Examples of famous relational / SQL Databases - MySQL, PostgreSQL.

User Table

1	Bob	Male	28	Single
2	Mark	Male	34	Married
3	Alice	Female	50	Married

User data has a defined structure

Step 02

Non-Relational Database / NoSQL

Non-Relational Database / NoSQL

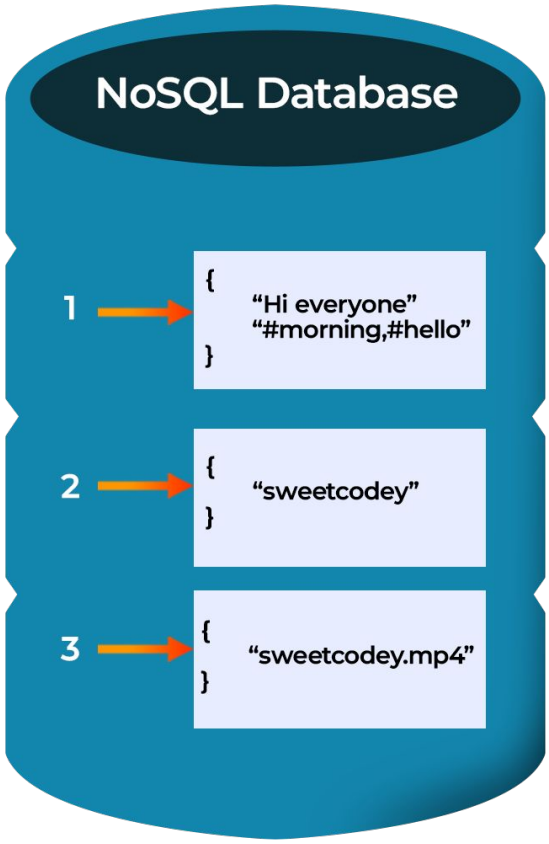
- Imagine saving social media posts in a table with columns for text, images, and videos.
- If a post has only text, the image and video columns remain empty.
- Similarly, a post with only a video leaves the text and image columns empty.
- This leads to many empty spaces in the table, which is inefficient and wastes resources.

Social Media Post Template

1	Hi everyone			#morning
2		Sweetcodey.png		
3			sweetcodey.mp4	

Lot of empty entries == Table space wasted

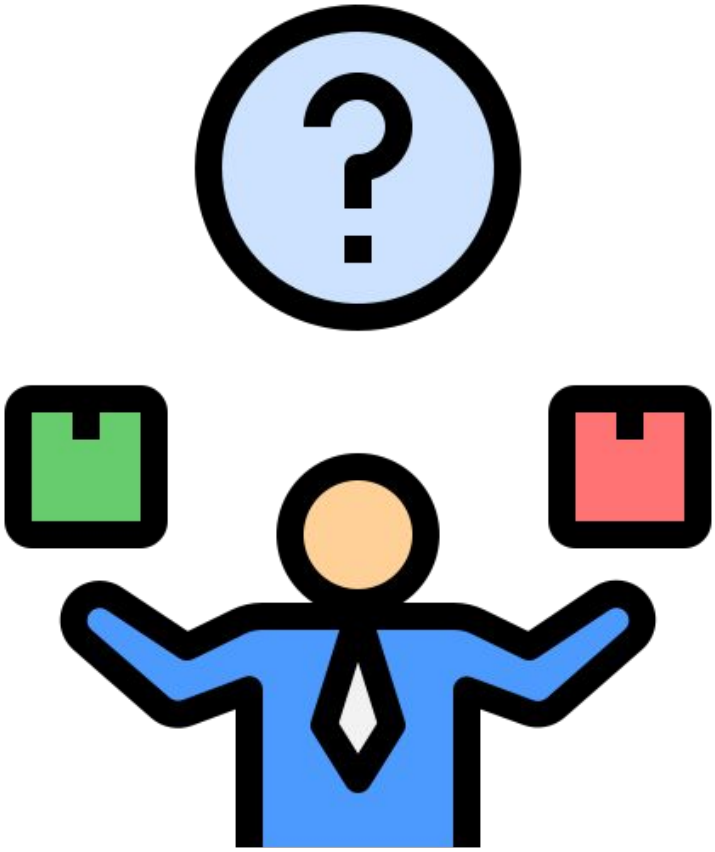
Social media posts doesn't have defined structure



Each post(being unstructured) is srored in a JSON document

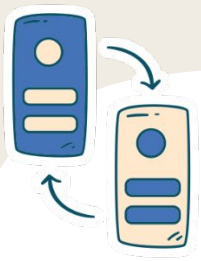
Can be accessed by Key (PostId)

- This is where we use NoSQL Database. It is ideal for this type of data which doesn't have a fixed structure.
- Examples of famous NoSQL Databases: MongoDB, Cassandra, DynamoDB.
- NoSQL databases come in various types, each suited to different needs:
  - Key-Value Stores
  - Document Databases
  - Graph Databases
  - Wide-Column Databases Time-Series Databases.



Step 03

SQL vs NoSQL

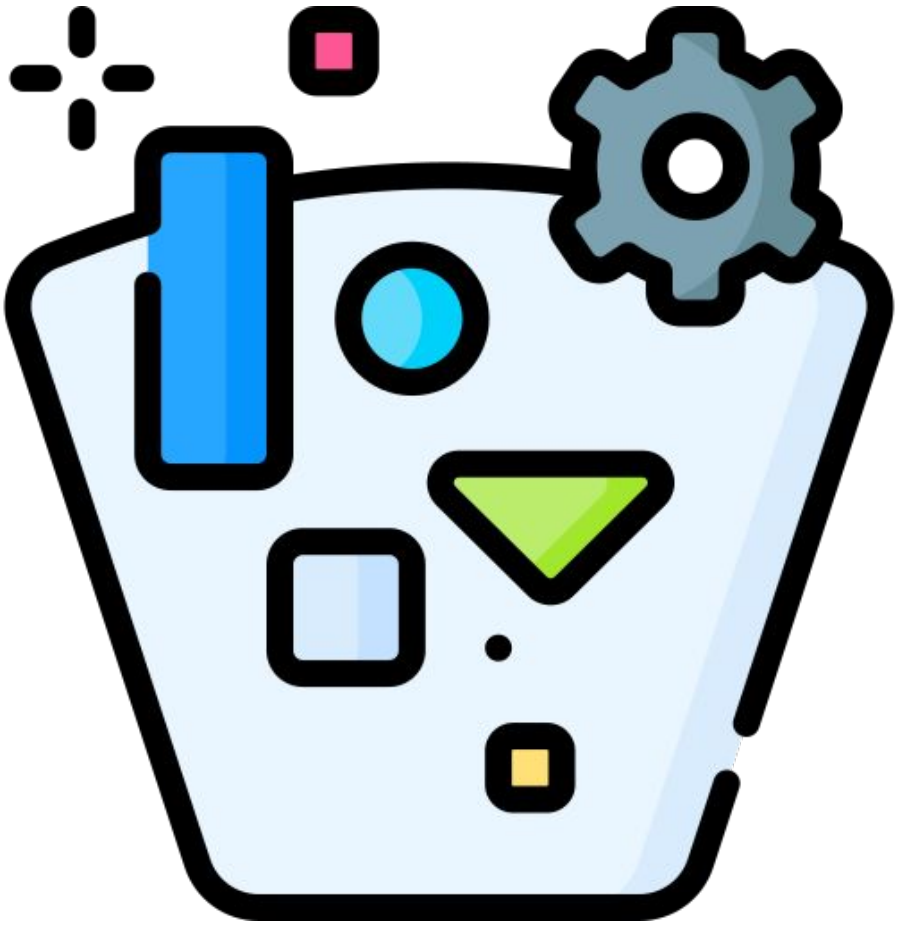


SQL vs NoSQL

- The natural question that arises here is how to choose between SQL vs NoSQL database.
- Here are some general guidelines that you can follow but DO REMEMBER it's not always black and white. A lot of things depends on the project needs.
    a. When you need fast data access, NoSQL is generally preferred over SQL.
    b. When the scale is too large, NoSQL databases tend to perform better than SQL databases.
    c. When the data fits into a fixed structure, SQL is more suited. When the data doesn't fit into a fixed structure, NoSQL should be the choice.
    d. If you have complex queries to execute on your data, SQL should be the choice. If you have simpler queries you can use NoSQL.
    e. If your data changes frequently or will evolve over time go for NoSQL database as it supports flexible structure.

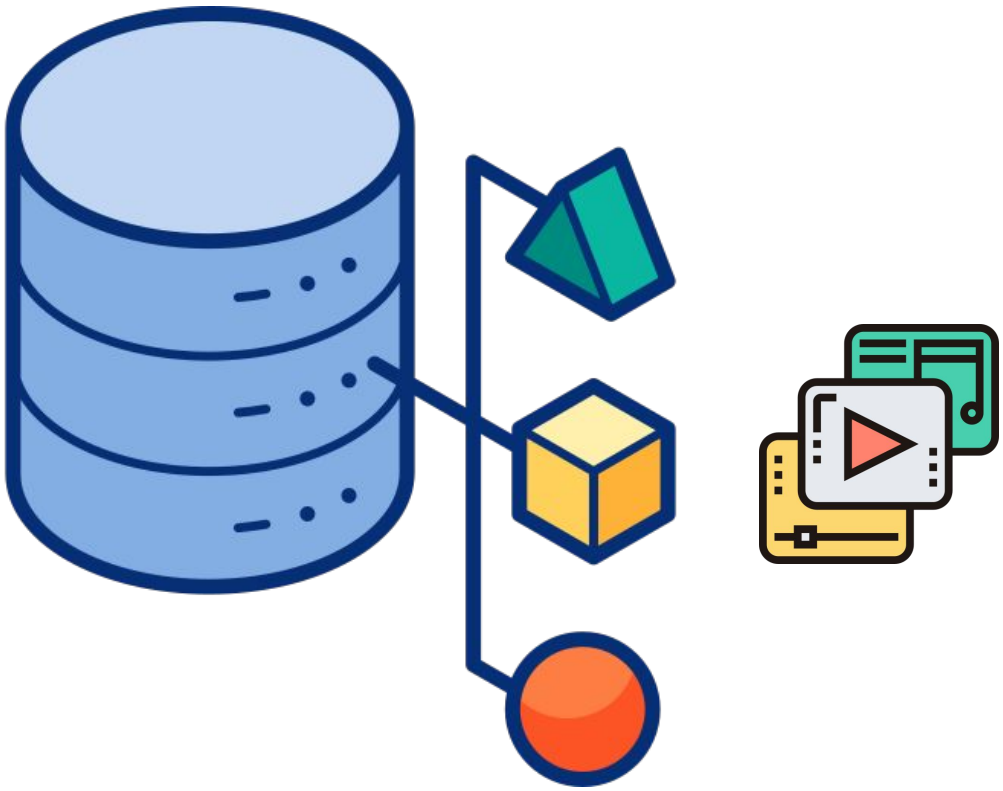
Step 04

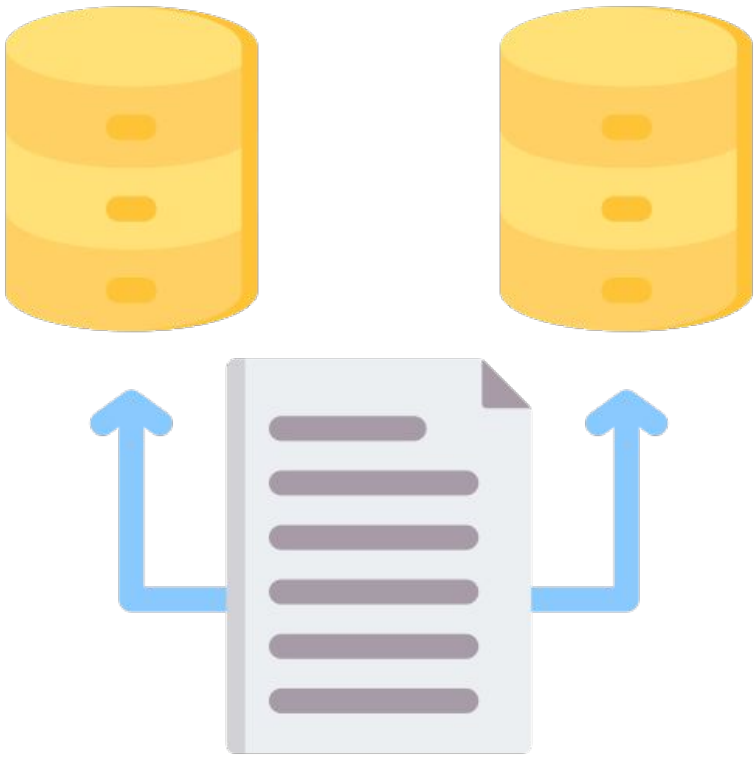
Object Storage



Object Storage

- In **Object Storage** we store objects.
- Each object is either a photo, video, audio, file. Effectively, they are simply units of data composed of bits/bytes.
- This type of storage is perfect for keeping large amounts of data that don't follow a regular structure, like pictures, videos, music, documents, and backups.
- Examples of object storage services include Amazon S3, Google Cloud Storage, and Microsoft Azure Blob Storage.



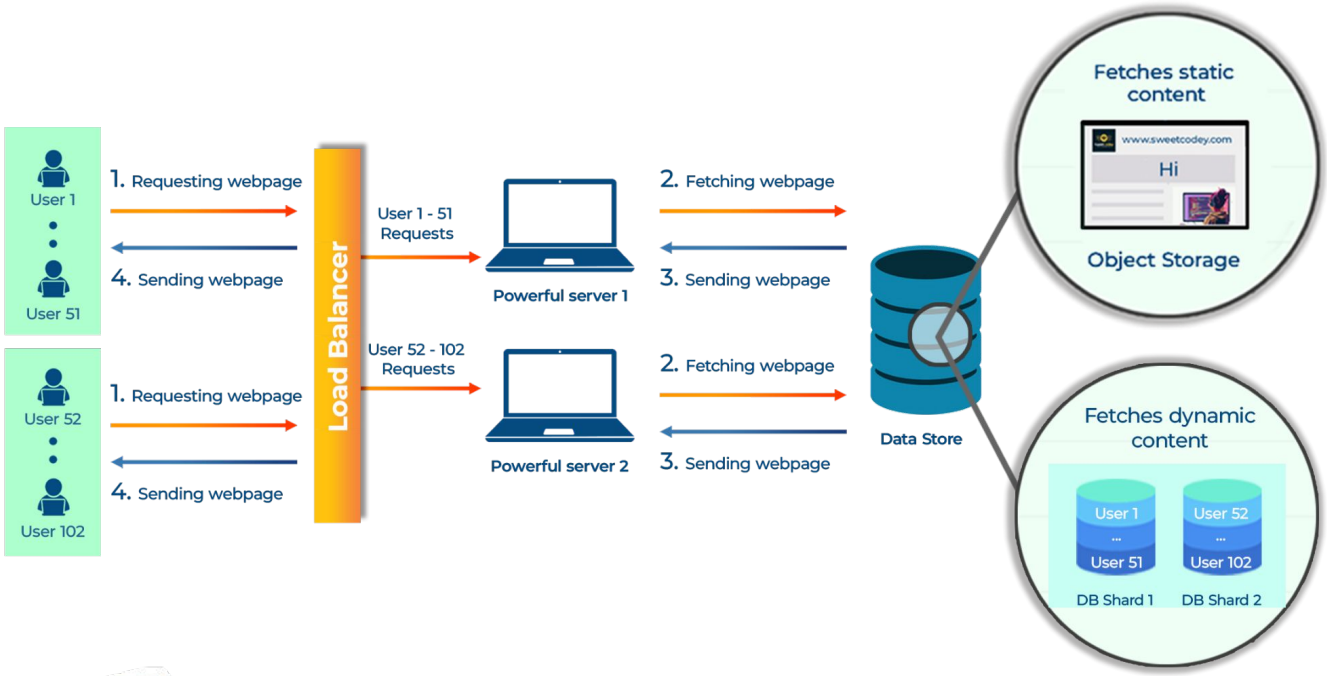


Step 05

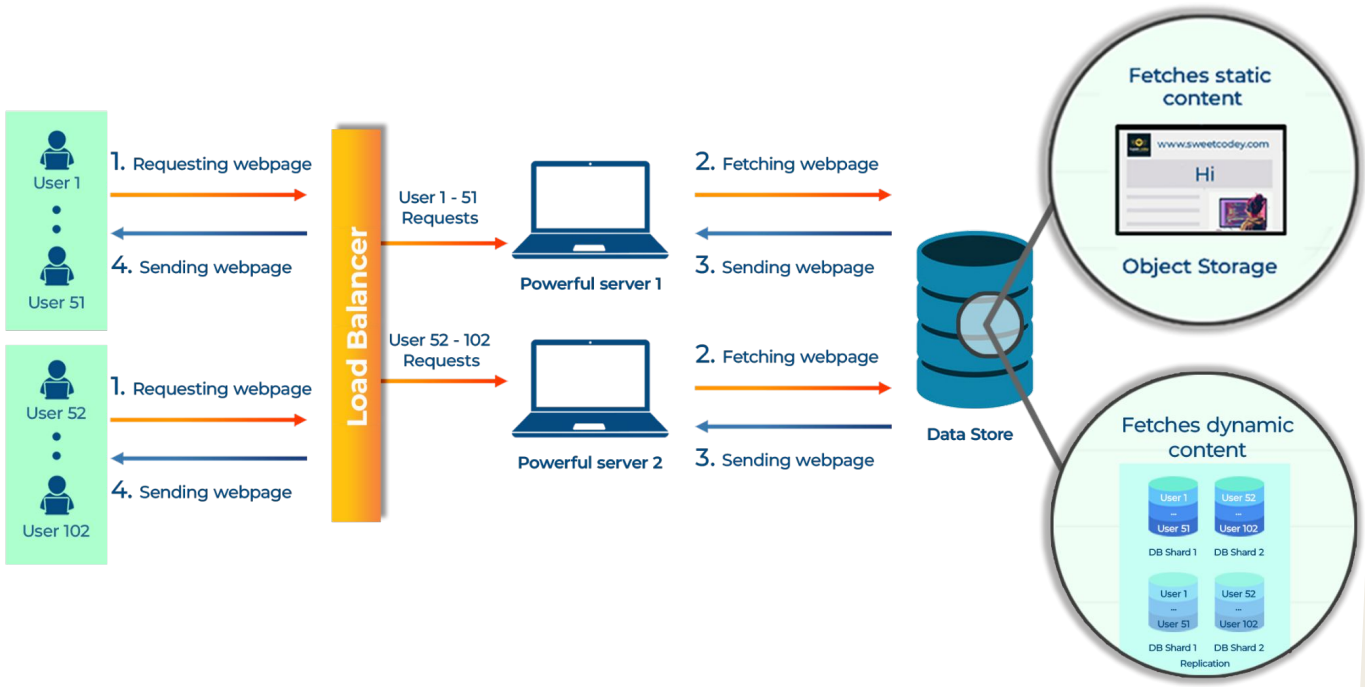
Database Sharding & Replication

Database Sharding & Replication

- **Database sharding** splits a large database into smaller sections called shards.
- Each shard stores a part of the data. This speeds up searches and reduces stress on any single server.
- If one shard has a problem and stops working, the other shards keep functioning. This means the whole system doesn't go down. This makes your database more reliable.

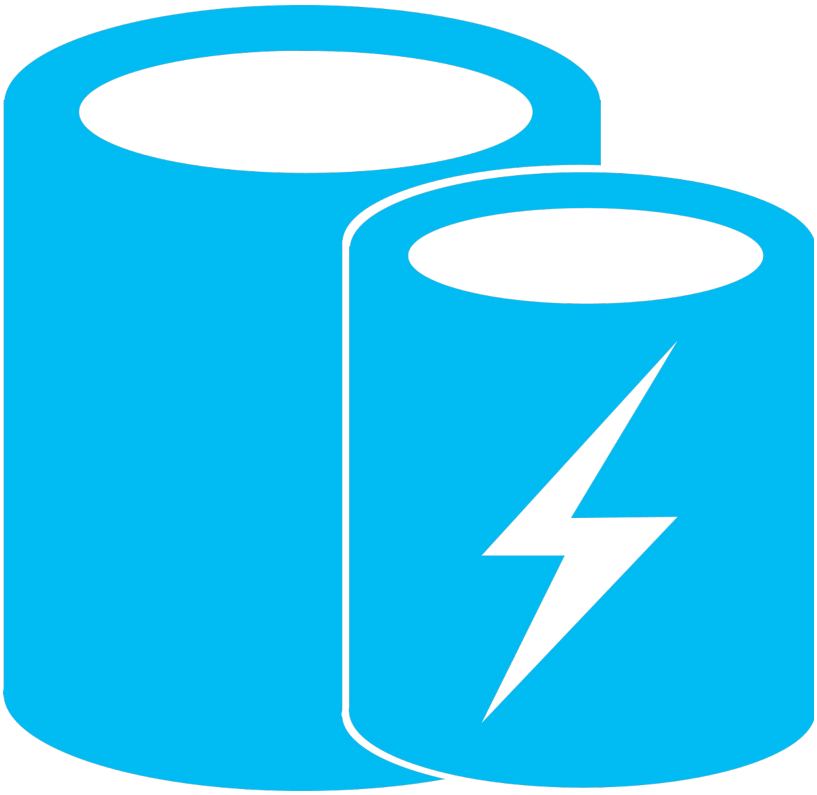


- **Database replication** is simply making copies of your database so that if one fails, others can take over



Step 06

Cache



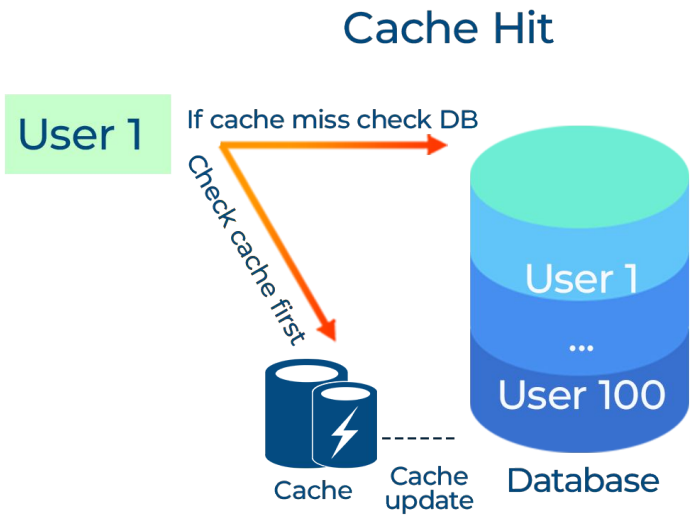
Cache

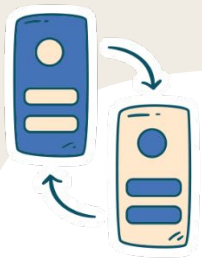
- Accessing data from database takes a long time. But if we want to access it faster, we use cache.
- Accessing from a cache is ~ 50 to 100 times faster than accessing from DB.
- Cache is a type of memory which is super fast but it has limited capacity (very less in comparison to database).
- That is why we use Cache to store frequently accessed data.
- It is like keeping snacks close to you at your desk (cache) while you study. Instead of walking to the kitchen (database) each time you're hungry, you simply grab a snack from your desk.

When the data is found in the cache, it is called a '**Cache Hit**'.
When the data is not found in the cache, it is called a '**Cache Miss**'.

Examples

- User1's data is found in the cache, so it is quickly fetched from the cache without the need for accessing the database.

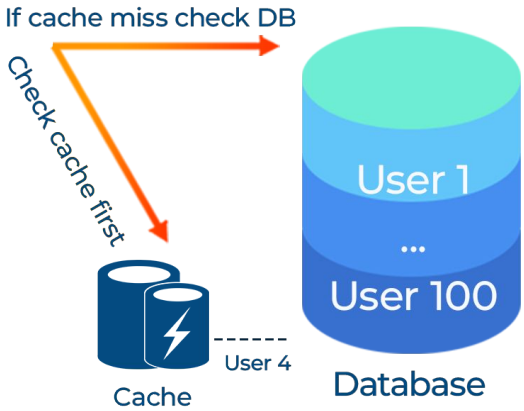




Cache

User4's data isn't in the cache initially. It's fetched from the database (slow) and the cache is updated.

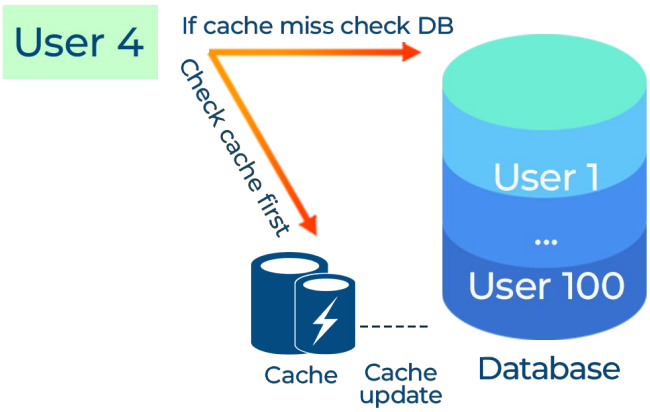
Cache Miss



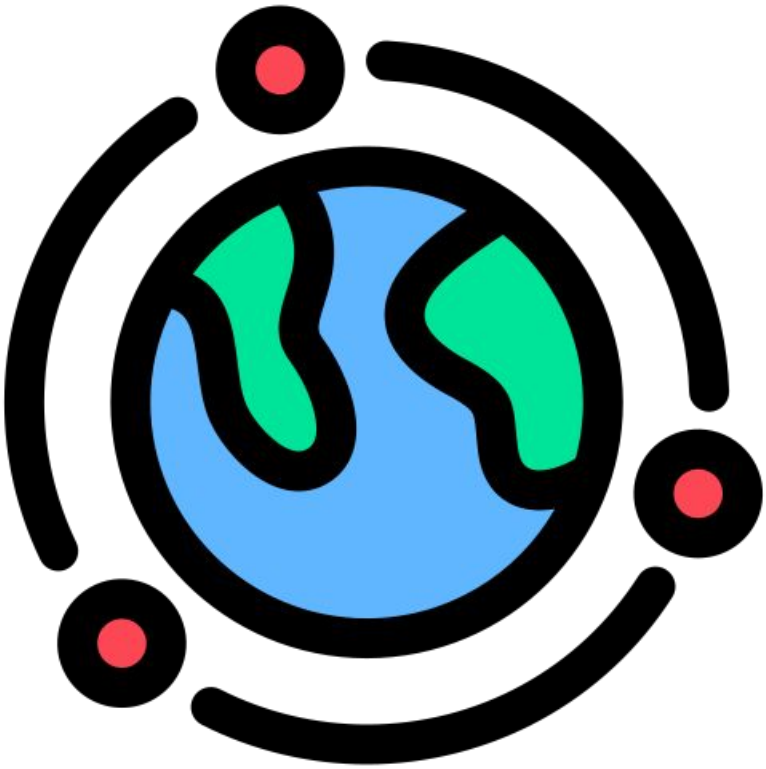
User 4 not in cache

Next request for User4 is quickly served from the cache because User4's data is now in the cache.

Cache Hit



User 4 is in cache

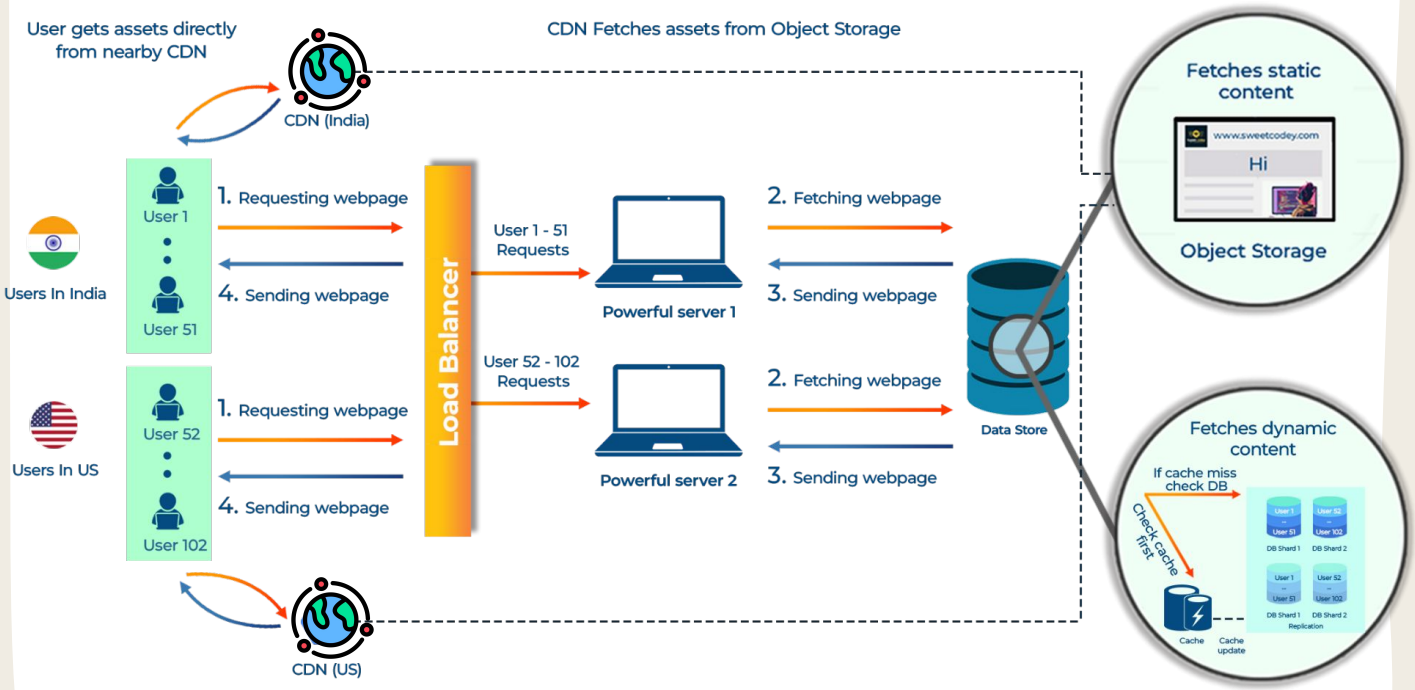


Step 7

Content Delivery Network (CDN)

Content Delivery Network

- Lets say Sweet Codey has all its servers in the US. A user from India tries to open sweetcodey.com.
- The website assets (Images, Videos, etc.) are bulky content. This bulky content will have to travel a long distance. This will increase latency a lot.
- A **CDN (Content Delivery Network)** comes handy in this case.
- It stores copies of your website's static content (static content = the data that doesn't change too often) at various locations around the world.
- Now, the user can quickly access static content (images, videos, etc.) directly from a CDN server closer to them.



Chapter 4

Buzzwords Networking

Buzzwords Networking

01 IP Address

02 DNS (Domain Name Server)

03 Client & Server

04 Protocols
(TCP, UDP, HTTP, Websockets)

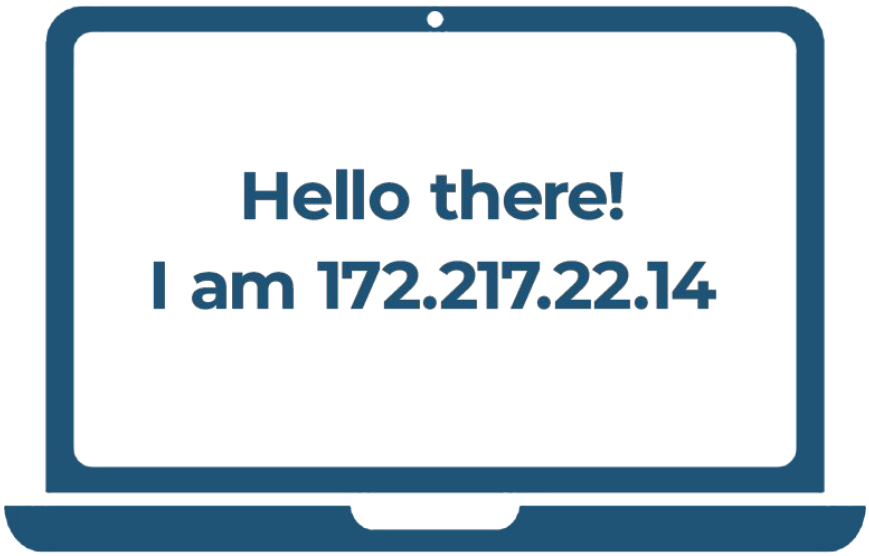
05 Forward & Reverse Proxy

Step 1

IP Address

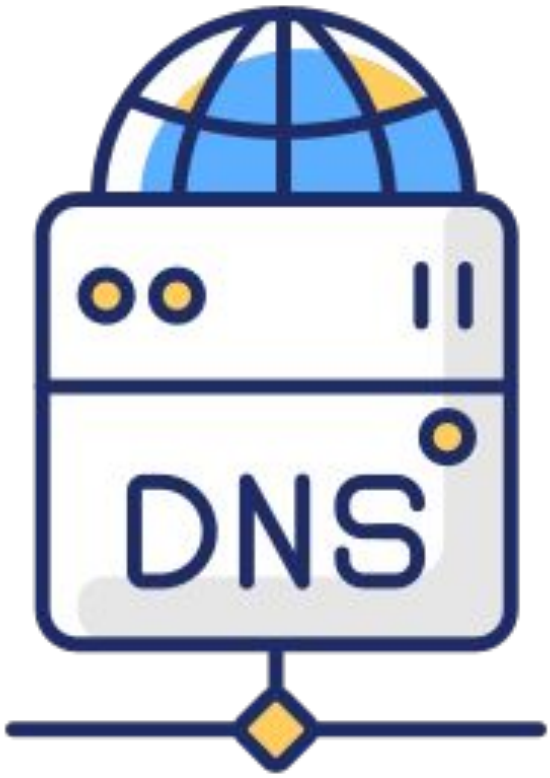
IP Address

- Just as every person in this world is known by their name, each computer on the internet is known by its IP address.
- An IP address looks something like this - 172.217.22.14 OR 2001:db8:3333:4444:5555:6666:7777:8888. Well that's a very horrible name!



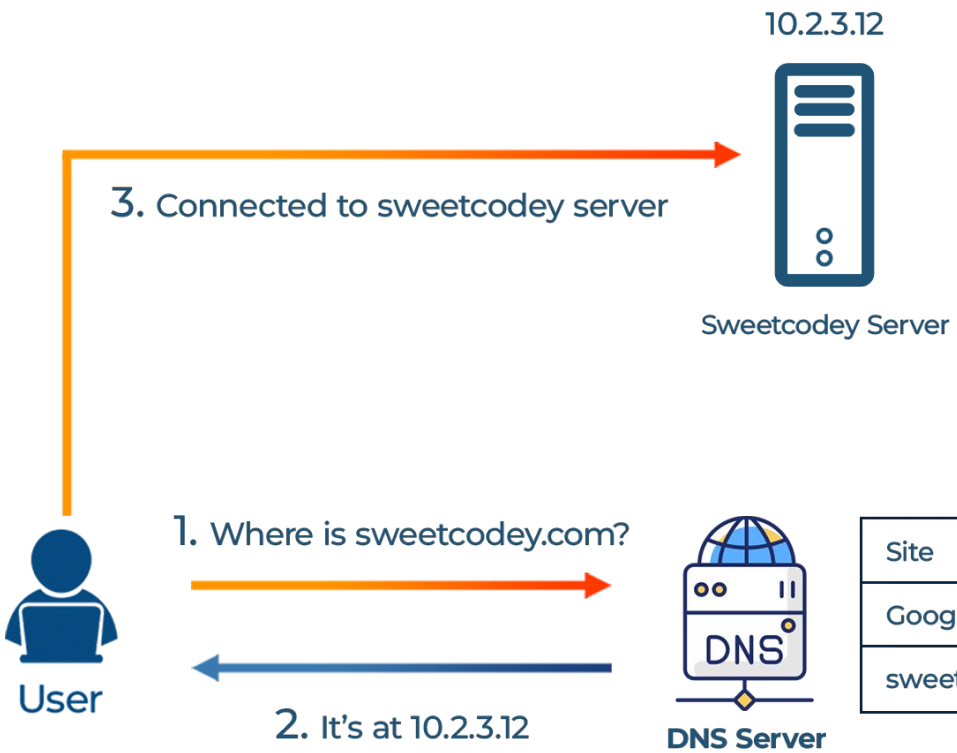
Step 02

DNS (Domain Name Server)

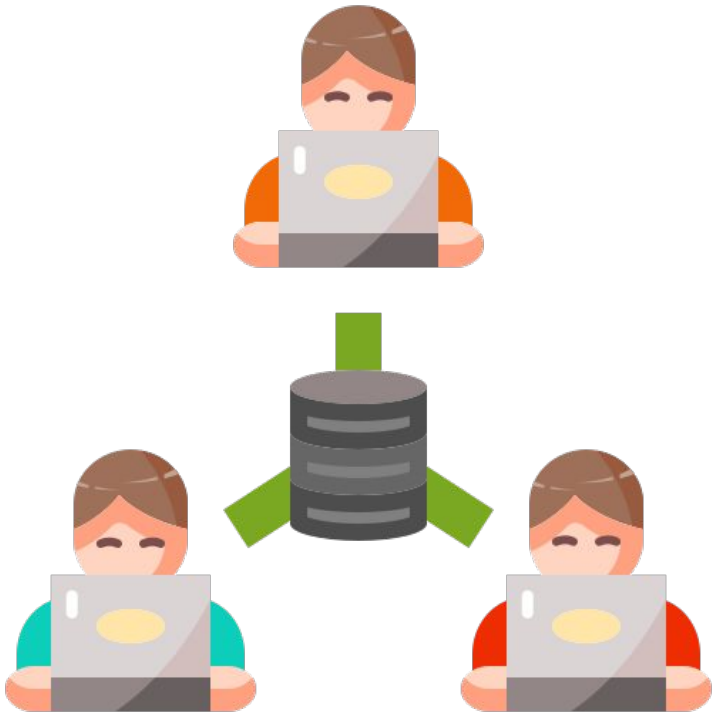


DNS (Domain Name Server)

- When you type sweetcodey.com in your browser, your browser requests sweetcodey's home page from servers.
- But how does your browser know where sweetcodey's servers are?
- That's the purpose of DNS.
- DNS is a service that takes the website name (like sweetcodey.com) and provides the IP address of the server.
- Now the browser knows the sweetcodey's IP address so it gets the homepage from it.



Site	IP
Google.com	10.11.10.191
sweetcodey.com	10.2.3.12



Step 03

Client & Server

Client & Server

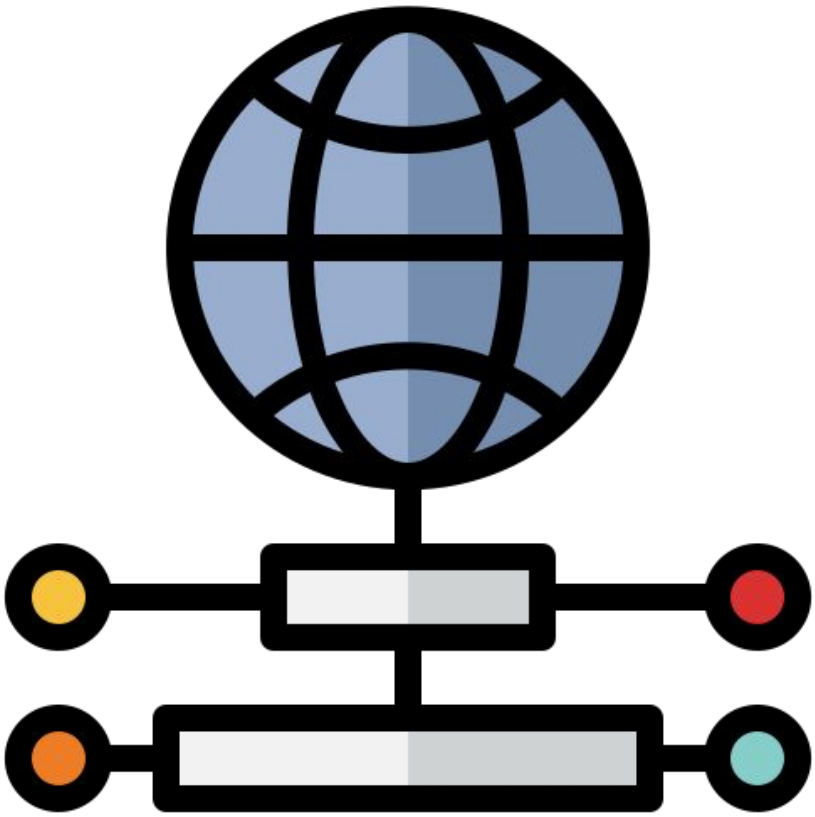
- **Client**: A computer that requests information.
- **Server**: A computer that serves the requested information from the client.

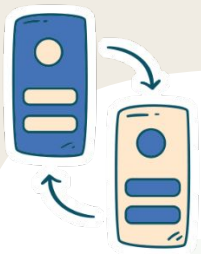
Examples:

- Your Smart TV requests movies (aka streaming) from Netflix - Your Smart TV is the client requesting information from Netflix Server.
- Your phone gets directions from Google Maps - Your phone is the client requesting information from the Google Maps Server.

Step 04

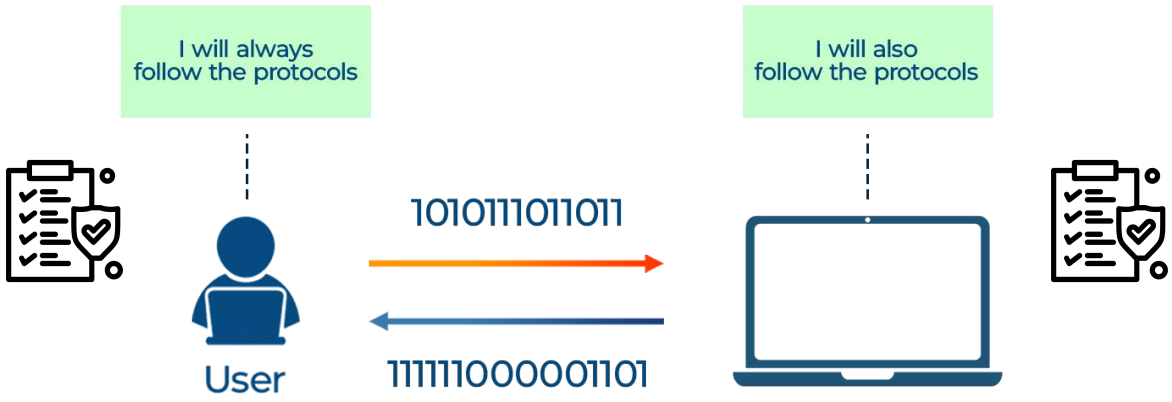
Protocols
TCP, UDP, HTTP, Websocket





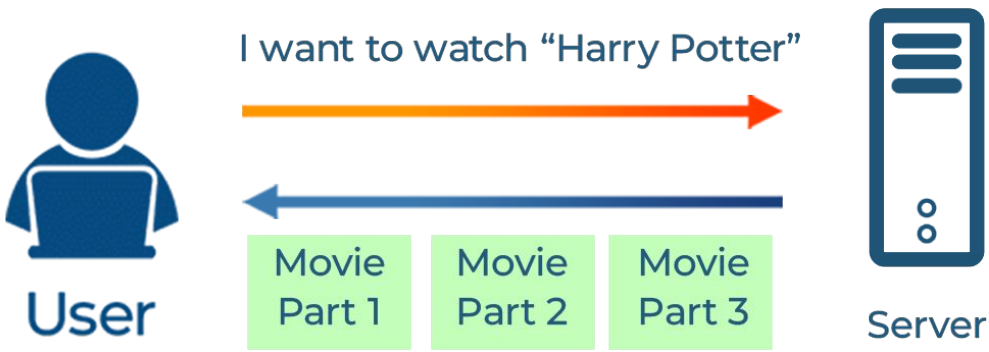
Protocols - TCP, UDP HTTP, Websocket

- Just as people use grammatical rules to communicate, computers also follow certain rules while communicating.
- The rules that computers follow are **Protocols**.
- Based on what the task is, we use different rules / protocols for it.
- Example - If the task to do some common web interactions like sending and receiving web pages, updating content etc. we use HTTP protocol. Similarly, if the task to transfer files we use FTP protocol.



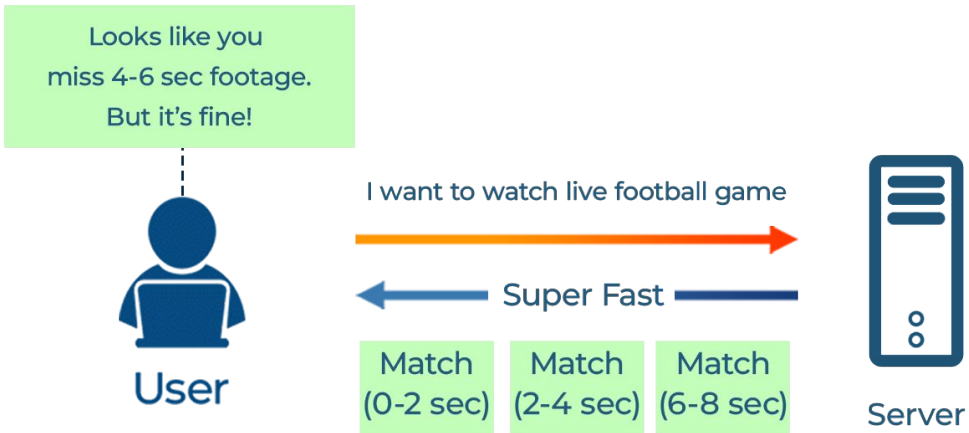
TCP (Transmission Control Protocol)

- Let's say you are streaming a movie and after the first scene, you see the climax directly. That's confusing, right?
- Well, **TCP** prevents this from happening.
- It is a protocol which ensures your data packets are delivered in the correct sequence, so you watch the movie in the proper order.
- So, whenever proper ordering is necessary, like in email, or streaming video, TCP is used.



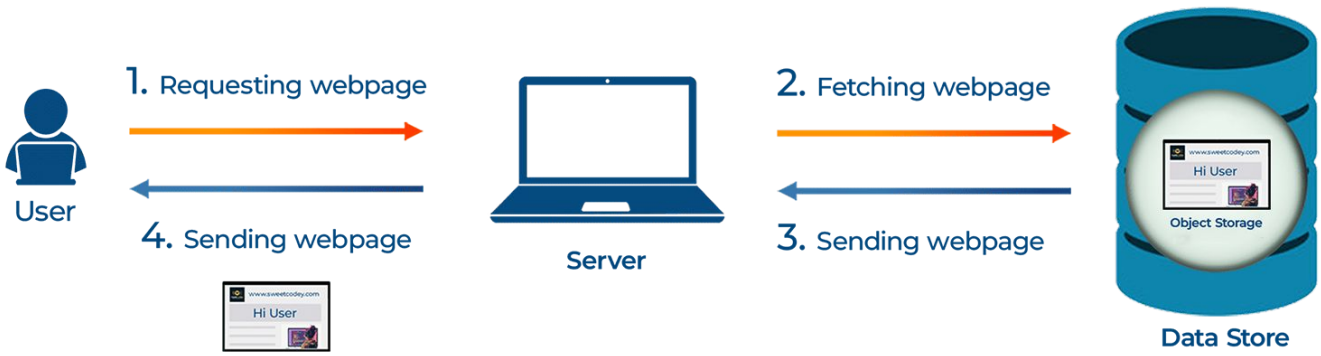
UDP (User Datagram Protocol)

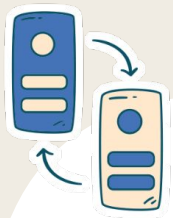
- Imagine you're watching a live football game. You want it to feel live with barely any delay.
- UDP helps with this!
- It sends video fast, though sometimes a few pieces might get lost.
- **UDP** protocol is very fast, but it doesn't guarantee the delivery of all data packets. In summary, UDP is perfect for tasks where speed is more important than reliability.
- Unlike UDP, which prioritizes speed and might skip some data, TCP focuses on reliability. It may be slower, but it ensures everything is complete and in order—making sure you don’t miss any part of your movie.



HTTP (Hypertext Transfer Protocol)

- HTTP is the most standard and commonly used protocol on the internet.
- It operates on a simple principle: you demand something from the server, and the server responds. For instance, you request a webpage, the server sends it back to you.
- Example: Consider shopping online. Each time you click on a product, your browser (client) sends a request to the store's server for product details. The server then fetches this information from its database and sends it back to your browser in the form of a webpage that you view.





Websockets

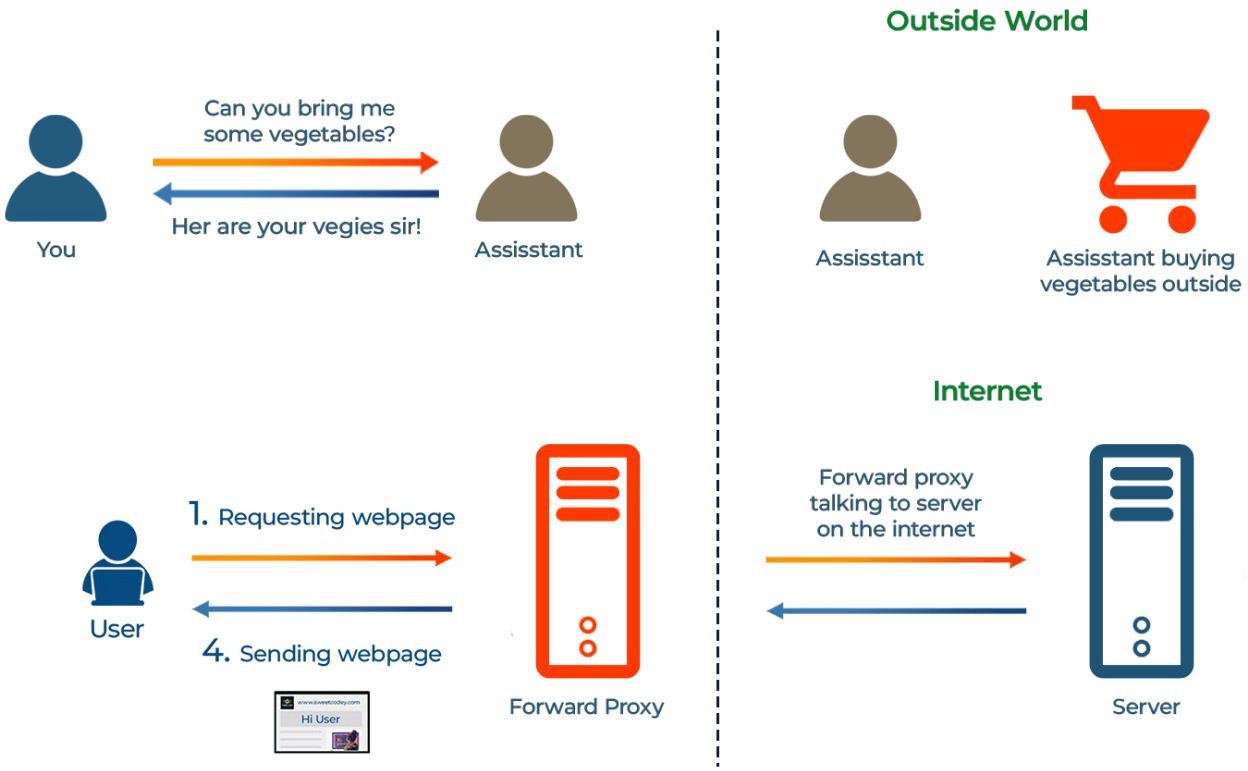
- In standard protocols like HTTP, the interaction is one-directional i.e. the client sends a request and then the server responds to it. Without a request from the client, the server cannot initiate sending data to the client.
- But with **Websockets**, both the client and the server can send data to each other at any time. This makes the communication bi-directional.
- Example: Consider a live chat application. Now your device can't constantly send requests to server every second asking "Do you have a new message?". That would be very inefficient.
- With websockets, whenever a message arrives for a client, server sends it to that client. This makes it very efficient.

Step 05

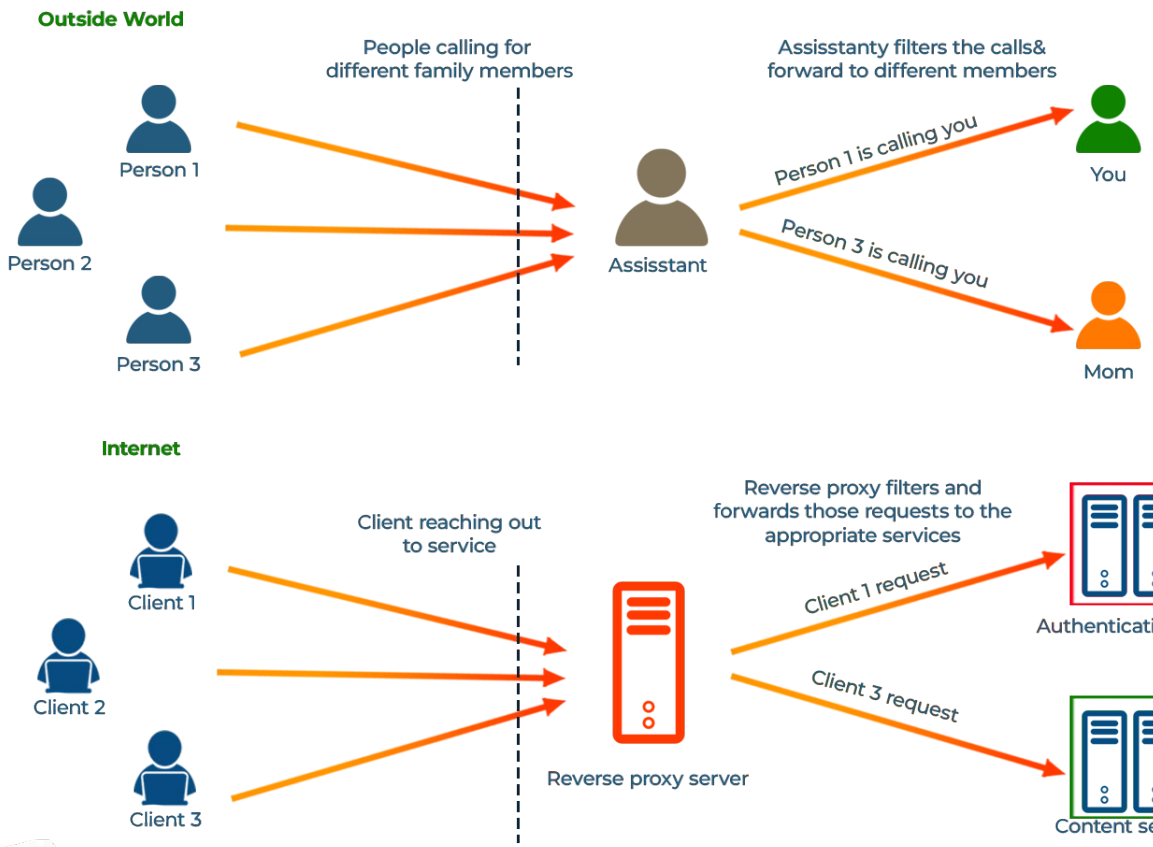
Forward Proxy & Reverse Proxy

Forward Proxy & Reverse Proxy

- **Forward Proxy** is like a personal assistant for your outdoor requests.
- Whenever you need something from outside, you just tell your assistant what you need, and they go get it for you.
- Similarly, a **forward proxy** sits between you (the client) and the internet.
- You send your requests to the proxy, and it fetches the information from the internet on your behalf.



- **Reverse Proxy** is like a personal assistant for your family.
- Now, instead of people contacting your family directly, they go through your assistant. The assistant filters the messages/calls and forwards only the important ones to your family members.
- Similarly, a **reverse proxy** sits between the internet and your services (collection of servers specializing in a task). It receives requests from clients and forwards those requests to the appropriate service.
- For example, if a user sends a login request, the reverse proxy routes it to the Authentication Service. If a user requests content, the reverse proxy routes it to the Content Service.



Chapter 5

Buzzwords Communication

Buzzwords Communication

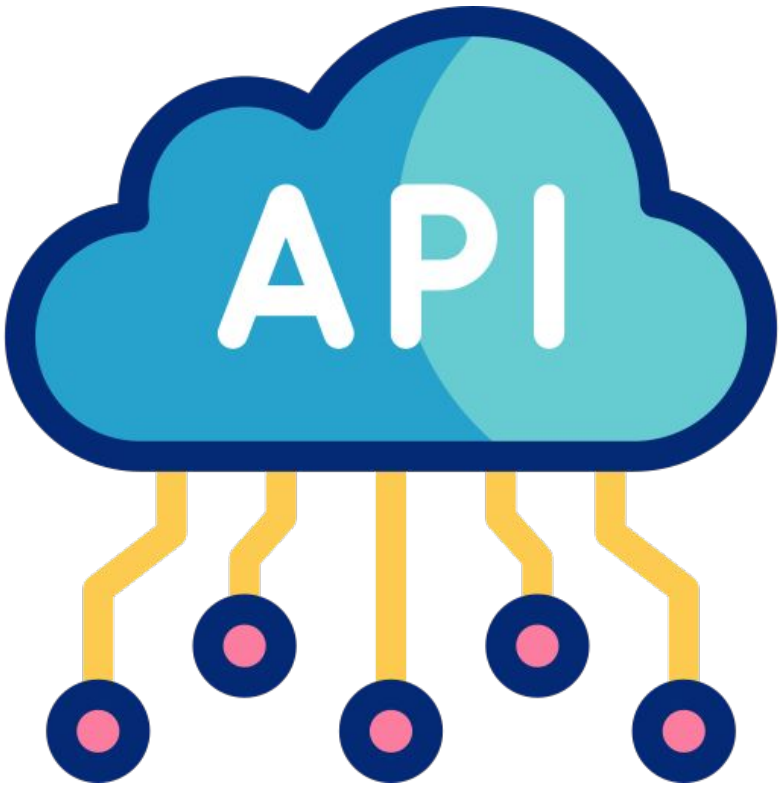
01 API

02 Rest API

03 GraphQL

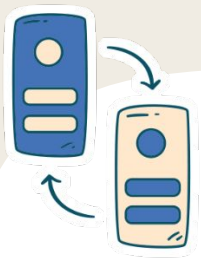
04 gRPC

05 Message Queue



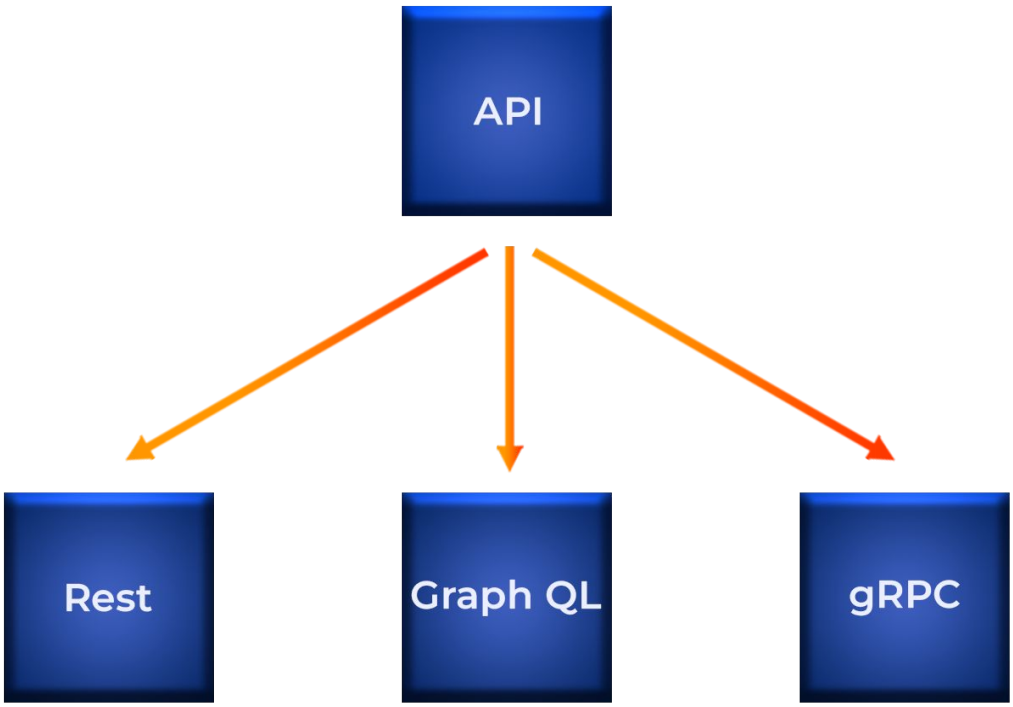
Step 01

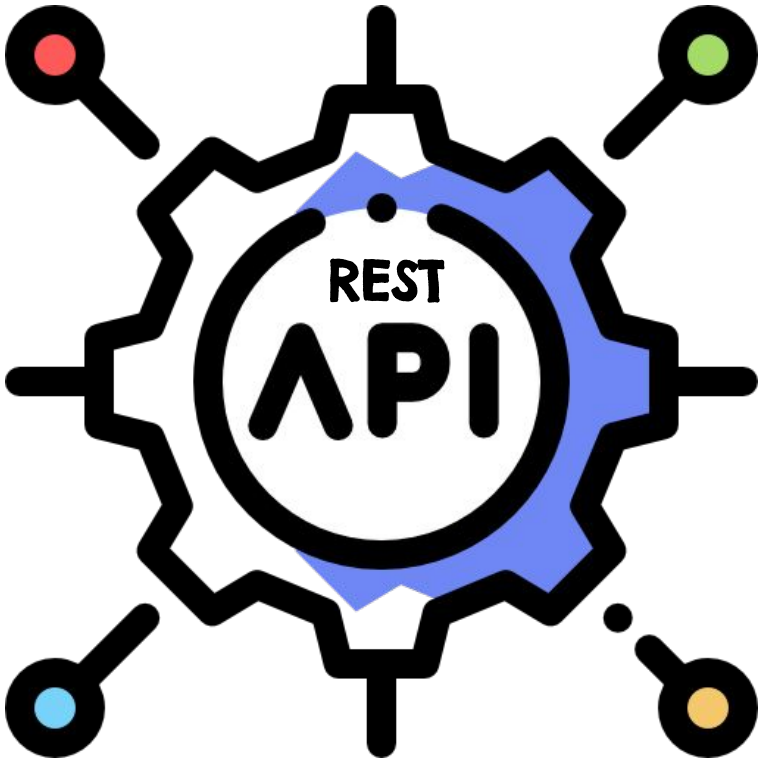
API



API

- Just like people are social, computers are social too. They talk to each other through APIs.
- Based on ‘how’ computers are talking to each other, we can classify the APIs. Here are 3 common types that we can discuss and learn more about.





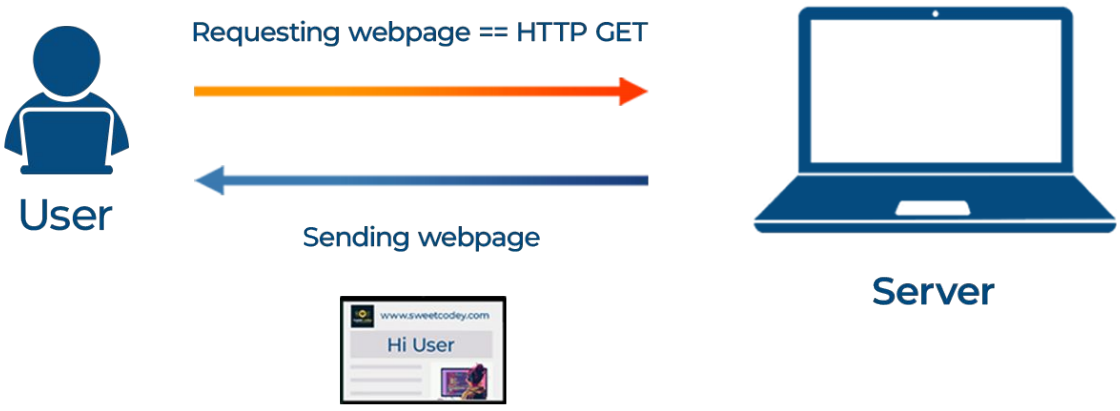
Step 02

Rest API

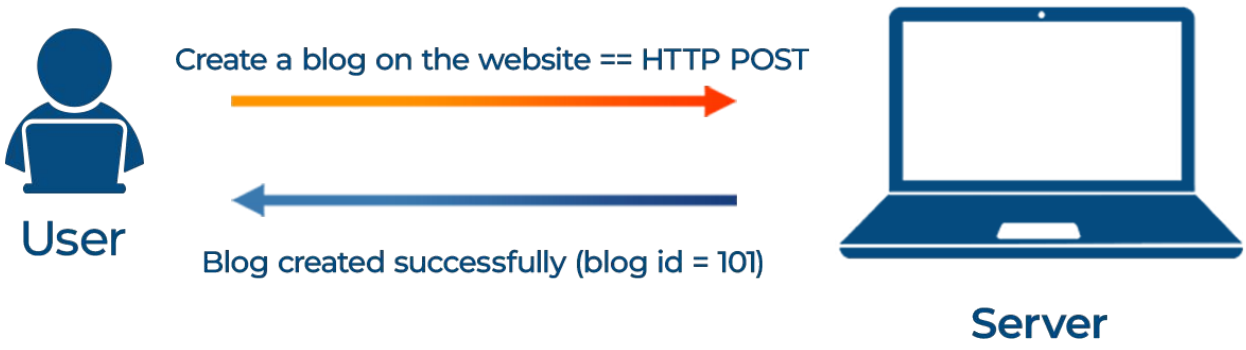
Rest API

- You go to a restaurant → look at the menu → order a couple of choices (eg. burger and fries) to the waiter.
- Waiter acknowledges them → then goes to the kitchen and informs chef → finally comes back with food.
- This is very similar to how REST API operates.
- Just as there are 'standard' ways for you to place an order i.e. only from menu items, computers use REST API to talk to each other, only in certain standard ways, to request and receive data. Shown below are 4 common standard ways.

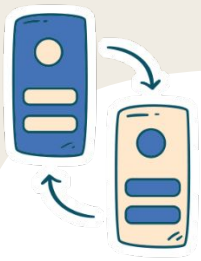
- **HTTP GET:** Client computer gets data from the server computer.



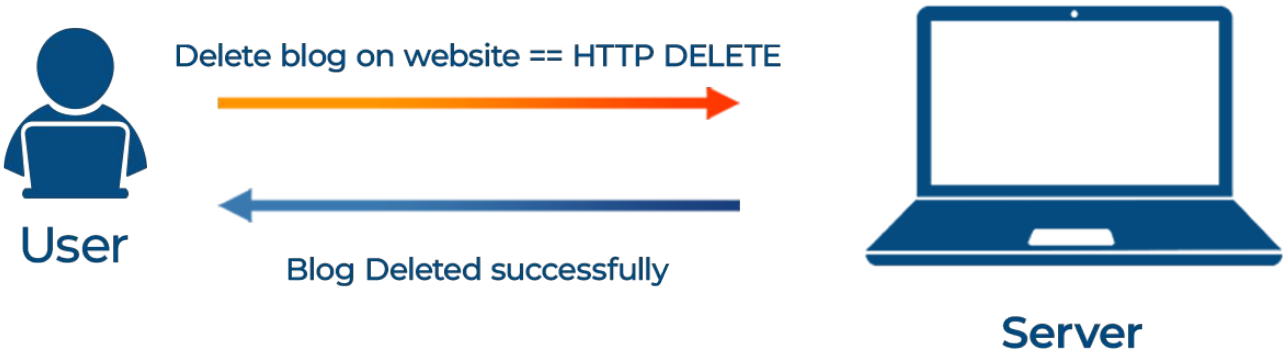
- **HTTP POST:** Client computer creates data in the server computer.



- **HTTP PATCH:** Client computer updates data in the server computer.



- **HTTP DELETE:** Client computer delete data in the server computer.



Step 03

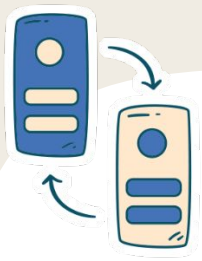
Graph QL

Graph QL

- Imagine at a restaurant, instead of picking directly from the menu, you customize your order. Say a burger with extra pickles, extra cheese, and a gluten-free bun.
- The waiter notes your specifics, tells the chef, who then makes your meal just as you asked. The waiter brings your custom meal exactly to your liking. This is how GraphQL works.
- Unlike REST, where you get the standard menu items, GraphQL lets you request customized menu items.
- If this order were placed using REST, you'd need to order each item separately—burger, extra pickles, extra cheese, gluten-free bun. Once all items are delivered, you would then assemble them into the burger you actually wanted.
- With GraphQL, you describe your complete custom burger in one order. The waiter (akin to the server) understands the detailed request (GraphQL API Request) and brings you your fully customized burger in one go.

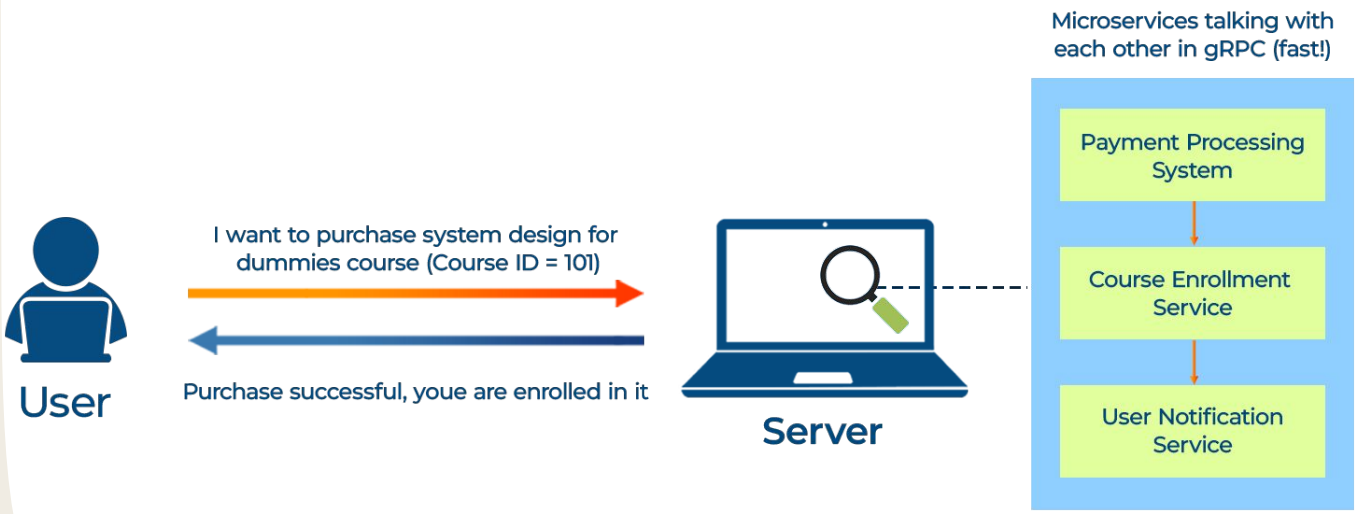
Step 04

gRPC



gRPC

- Imagine you're at a restaurant. You look at the menu and decide to order a burger and fries.
- You tell the waiter your order.
- The waiter quickly goes to the kitchen and says "B+F for table 1" i.e. Burger and Fries for table 1. This special language helps them communicate super fast and efficiently.
- After the kitchen prepares your order using this quick communication, the waiter brings your meal to the table without any delay.
- This quick internal communication at the restaurant is a lot like gRPC in the tech world.
- Just as the kitchen staff use a shorthand to communicate efficiently, gRPC API allows different internal parts of system (like microservices) to communicate efficiently.
- gRPC uses less data to send messages which makes it fast.



Step 5

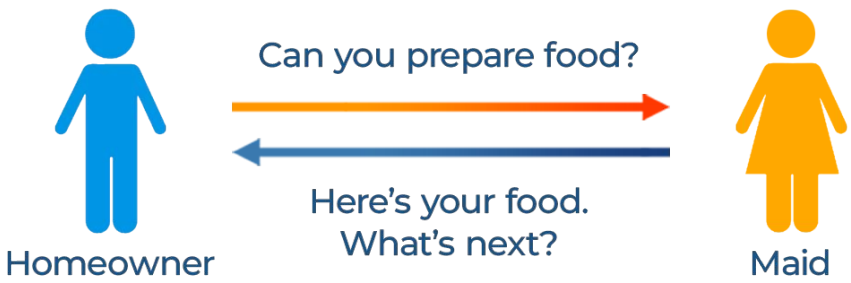
Message Queue

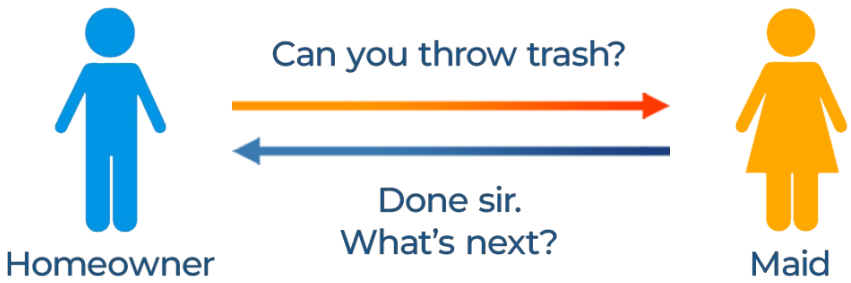
Message Queue

- Imagine a homeowner who has a long list of tasks (prepare food, do trash, clean home, clean utensils etc.).
- These tasks have to be done in the morning before the homeowner leaves for his work.
- He has a helper maid who is gonna help him with these tasks.

Scenario 1:

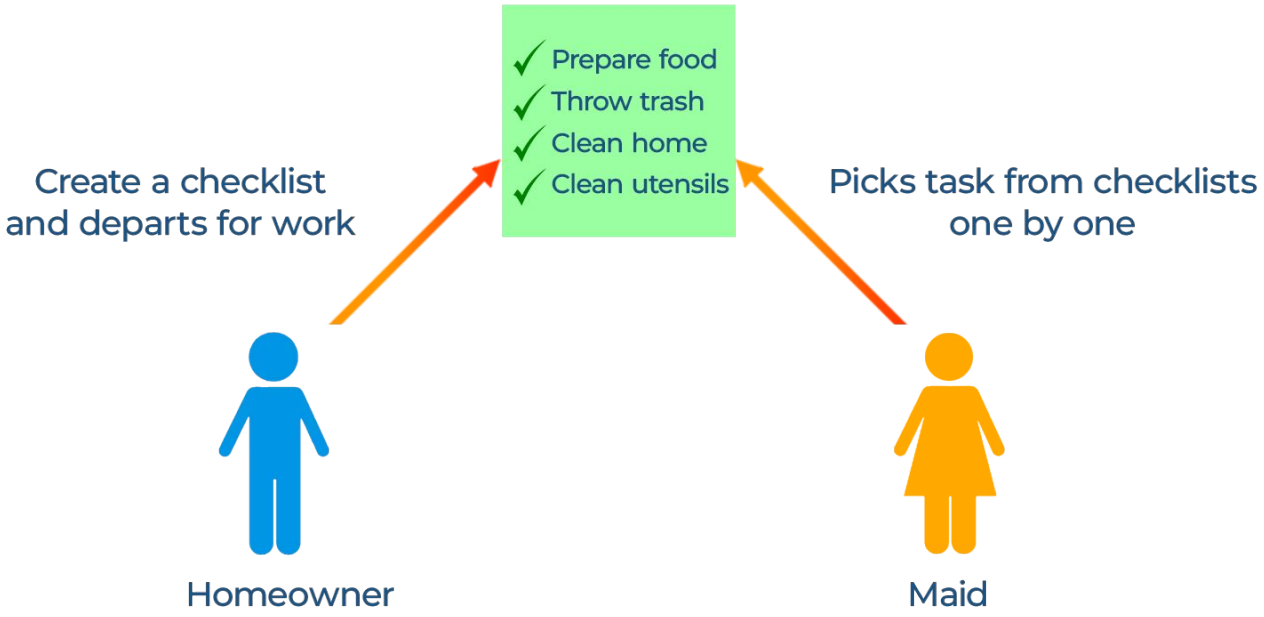
- He starts giving tasks to his maid one by one.
- He waits for each to be completed before assigning the next.
- This is inefficient.
- If he waits for each task to be finished, it will waste his time.
- Also, he has to leave for work, and this approach delays his departure.



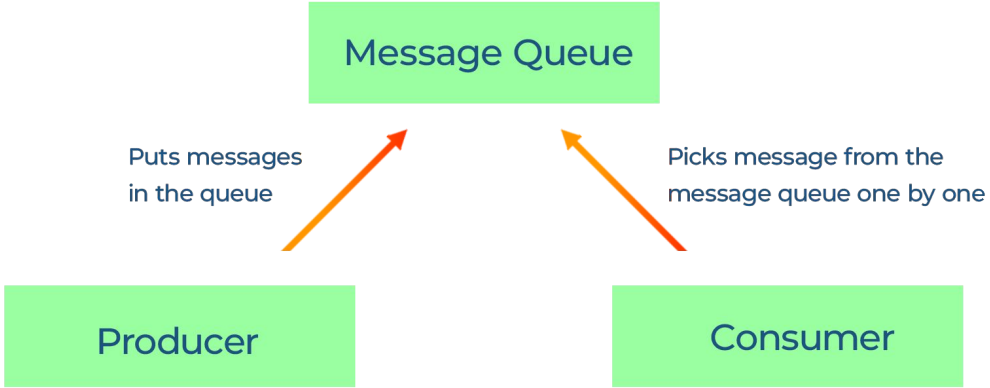


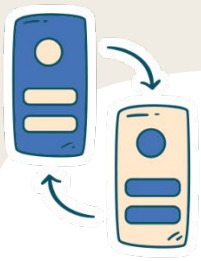
Scenario 2:

- He writes down all the tasks on a checklist and leaves for work.
- The maid picks up tasks from the checklist one by one and completes them independently.
- The homeowner can go to work on time, while the maid completes the tasks at her own pace.
- This is much more efficient.



- A Message Queue works just like this task checklist scenario.
- The homeowner is like the producer who adds tasks (messages) to the queue (checklist), and the maid is like the consumer who processes the tasks one by one.





Benefits:

- It is more efficient because the homeowner saves time and can work on other tasks, without waiting for each one to be processed.
- The task checklist ensures that no tasks are forgotten or lost.
- If the maid needs a break or has to step out, the tasks will remain on the checklist.
- Whenever she returns, she can pick up right where she left off.
- This ensures all the tasks are taken care of and will be completed by the end.

Drawbacks:

- For tasks like turning on the light switch or adding sugar to coffee, writing them in a checklist overcomplicates things. It's faster to handle these tasks directly.
- For urgent tasks like turning off a burning stove, writing them in a checklist causes unnecessary delays. These tasks need immediate attention, and a message queue would be too slow.

Chapter 6

Buzzwords Extras

Buzzwords Extras

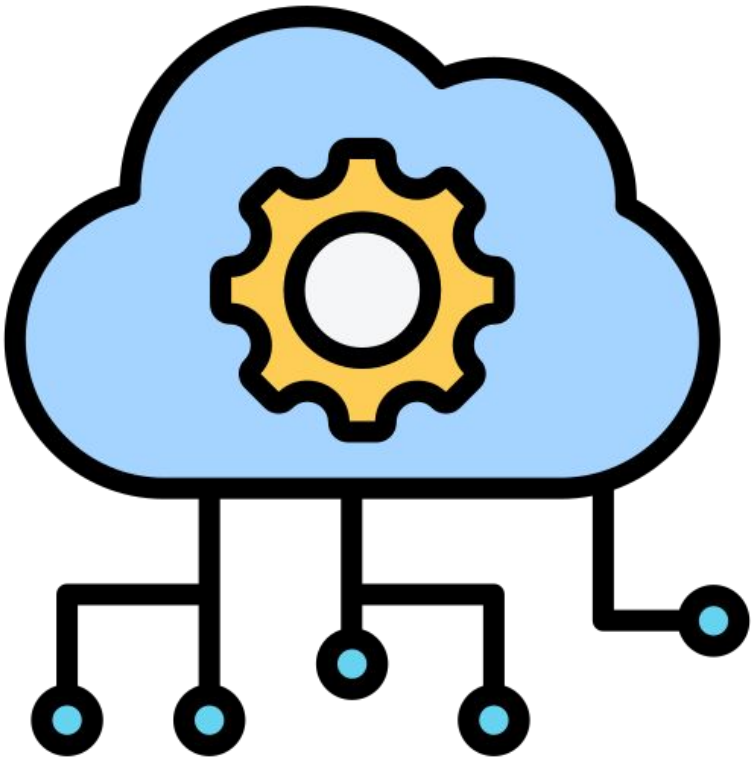
01 Cloud Computing

02 Logging & Monitoring

03 Caching Strategies

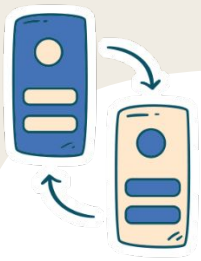
04 Hashing & Consistent Hashing

05 CAP Theorem



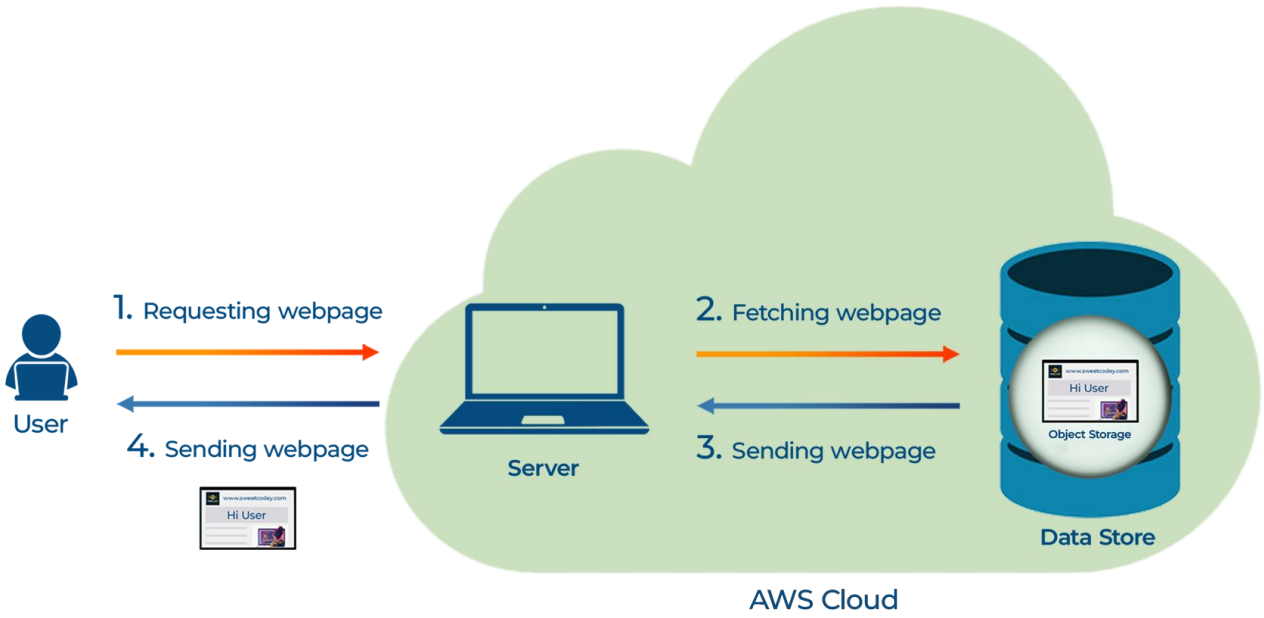
Step 01

Cloud Computing



Cloud Computing

- Myth buster: There is no such thing as cloud. Servers are always kept in a data center.
- If you try to host your website, you will need servers for that. Instead of buying and maintaining physical servers yourself for your website, why don't you rent them?
- This is where **Cloud computing** comes in. With cloud computing, you can rent computing resources from providers like AWS, Google Cloud, or Microsoft Azure, allowing you to focus on your website while they handle the infrastructure.



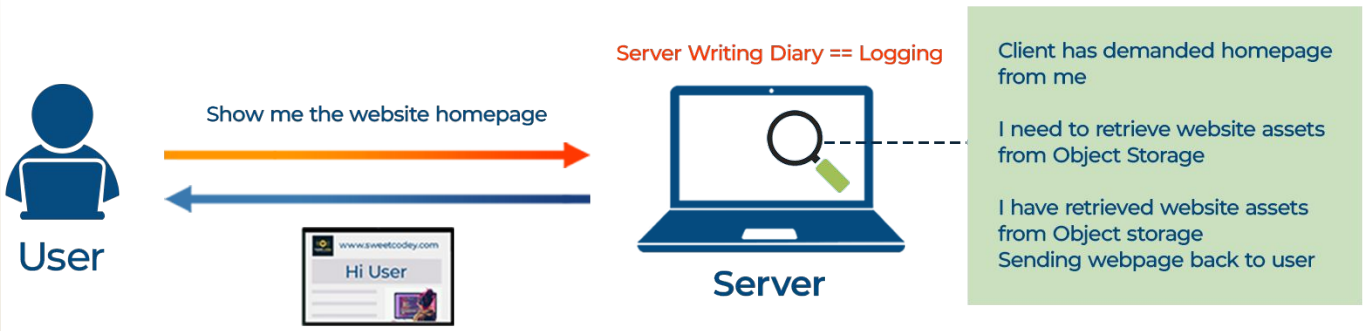
Step 2

Logging and Monitoring

Logging and Monitoring

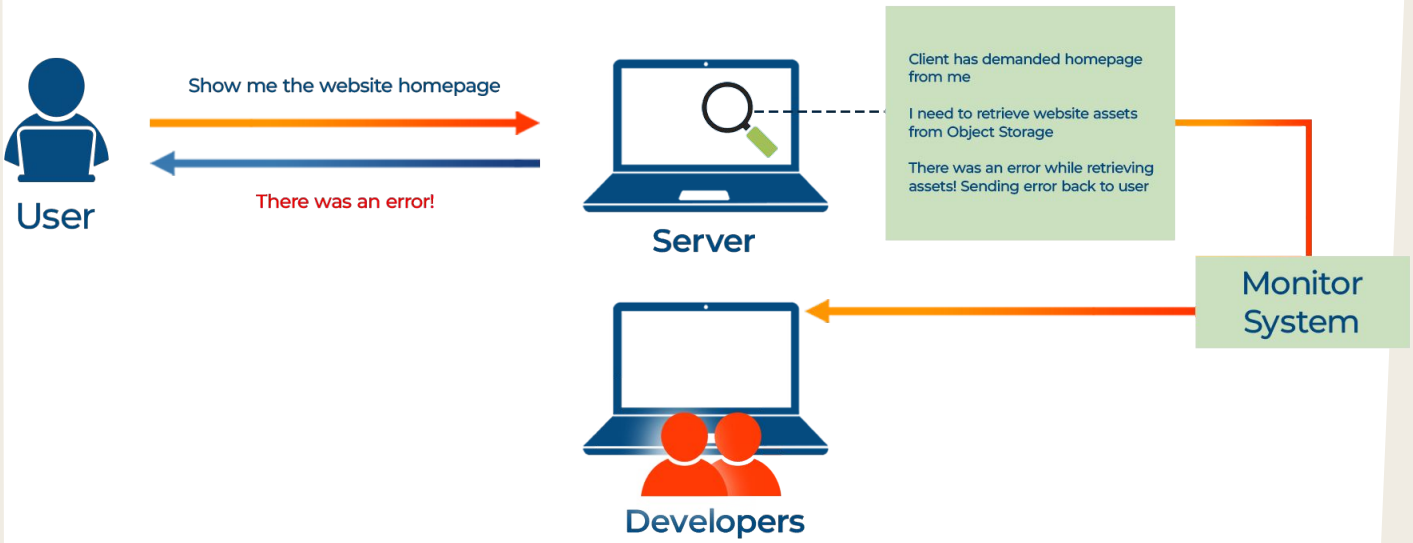
Logging

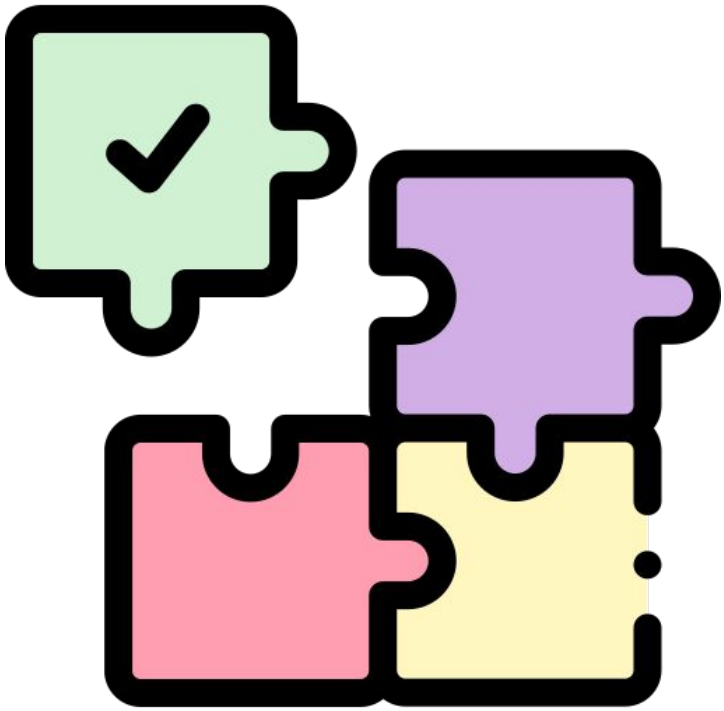
- Logging is like a computer writing in a diary.
- It records everything important like errors and key activities.
- These records are called logs.
- These logs can be used to fix problems (troubleshooting) and also for data analysis.



Monitoring

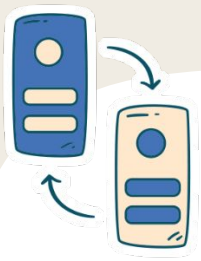
- Monitoring is like keeping a close watch on a computer.
- When we monitor we constantly check that everything is working properly.
- If there are any issues, we catch them and fix them.
- It's like having a security camera on a computer.





Step 03

Caching Strategies

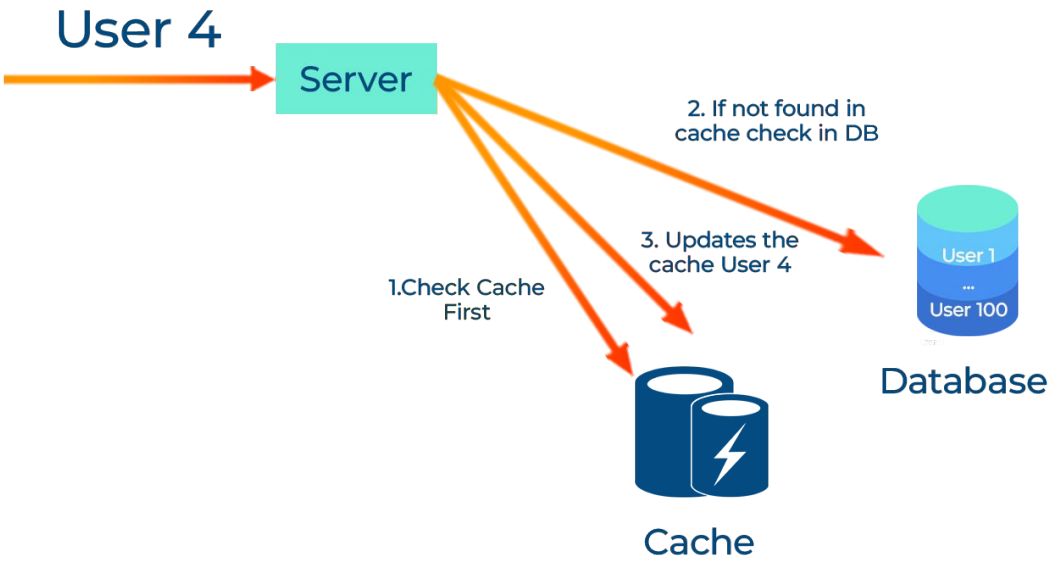


Caching Strategies

Here are two common methods to read data from a cache:

- **Cache Aside Strategy:**

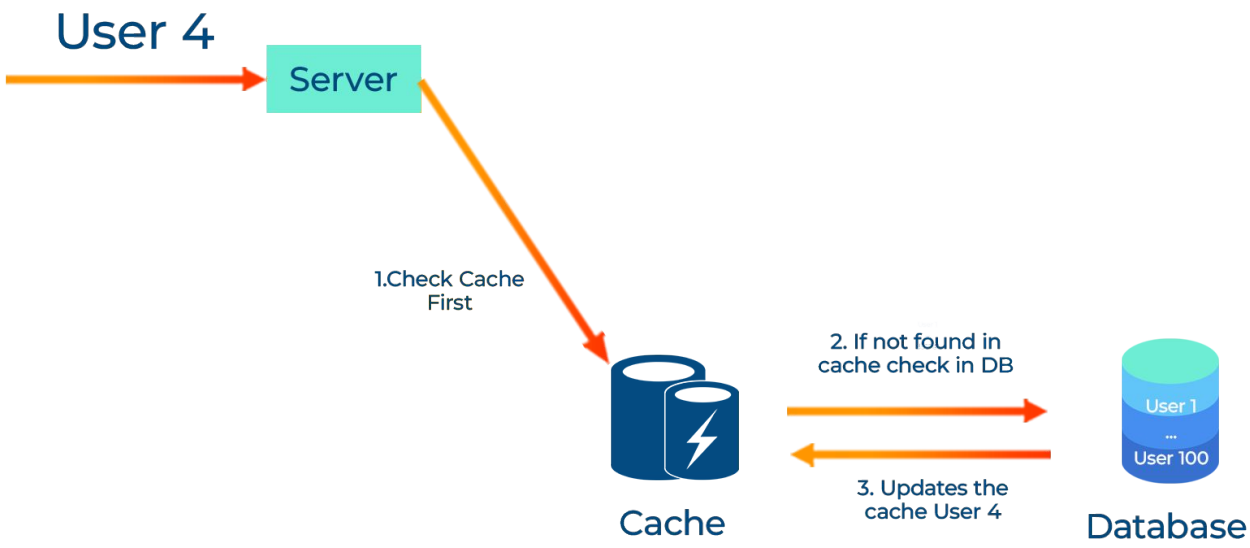
When data is requested, we first look in the cache. If it's not there, we get it from the database and save it in the cache for next time.

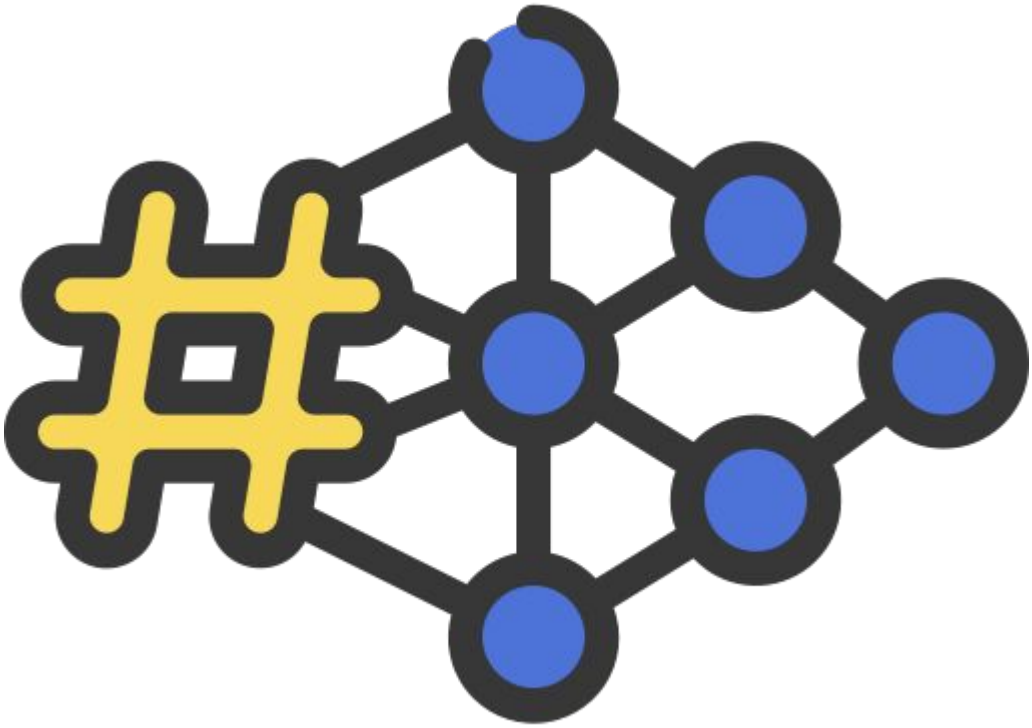


- **Read Through Strategy:**

  When the data is requested, we first look in the cache. If it's not there, the cache itself gets the data from the database and saves in itself for next time.

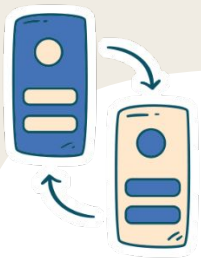
  This is called 'Read Through' because we are reading the data 'through the cache'.





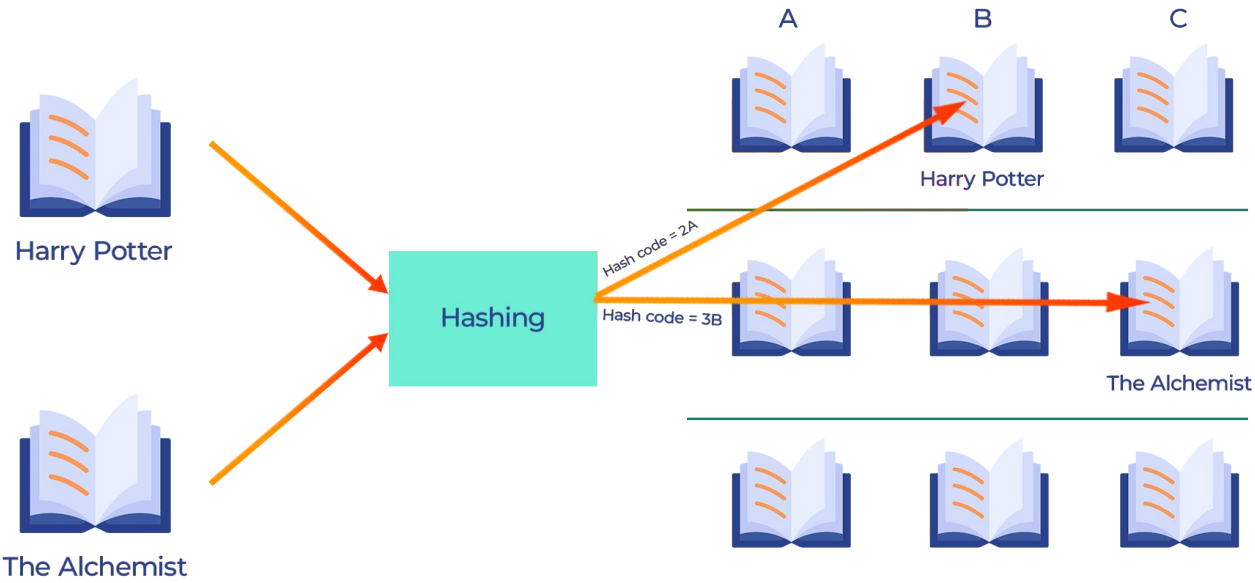
Step 04

Hashing & Consistent Hashing

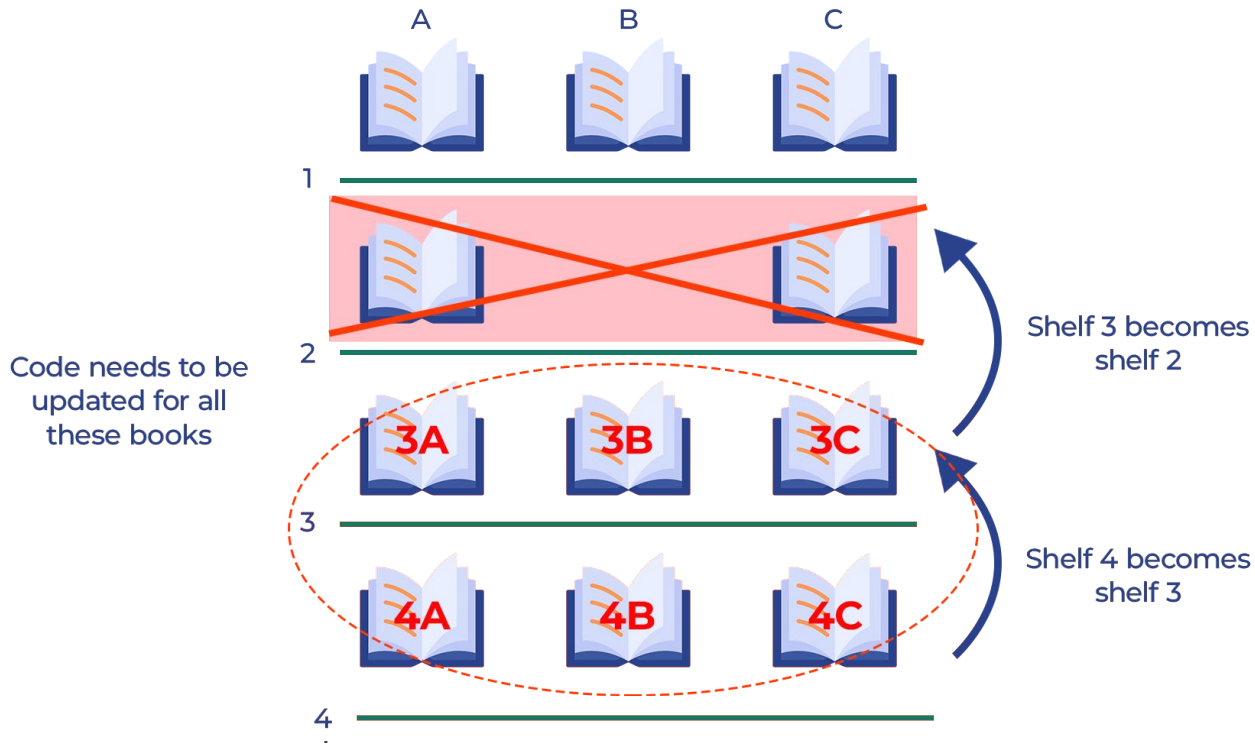


Hashing And Consistent Hashing

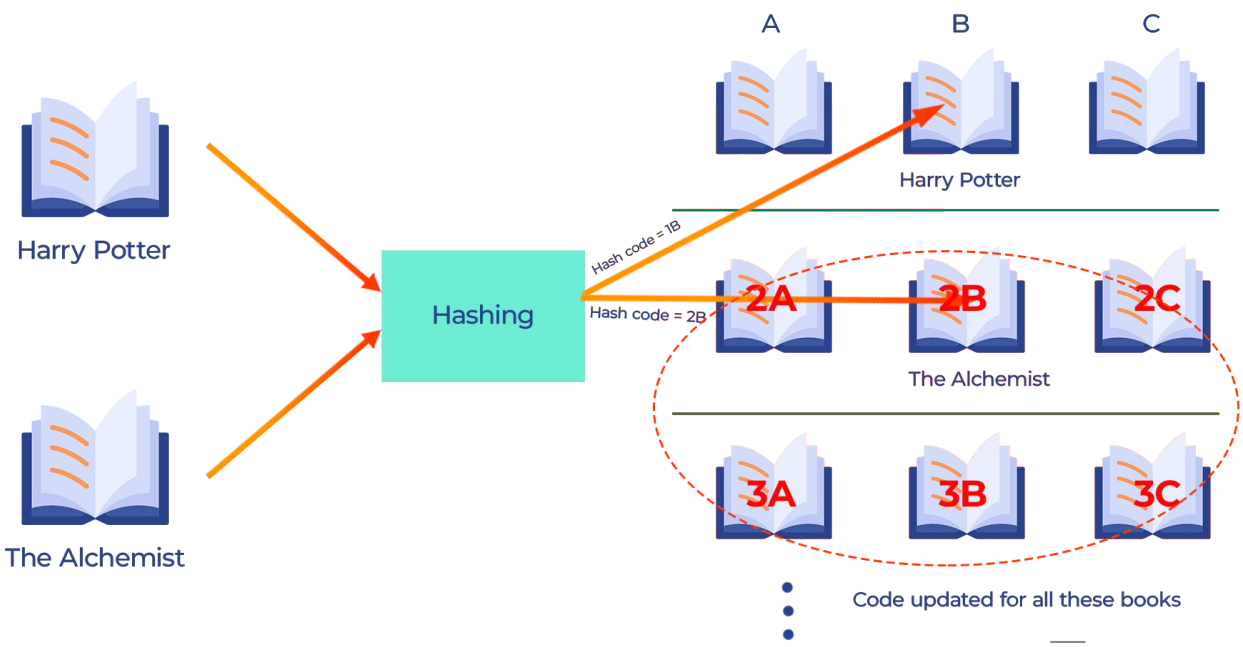
- Think of a library where each new book gets a numeric code. The librarian uses this code to quickly put the book on the right shelf.
- To find a book, the system uses this code, which is much faster than searching by title.
- **Hashing** is very similar. It converts data into a short, random, unique code. This code helps efficiently place and locate data in a system.



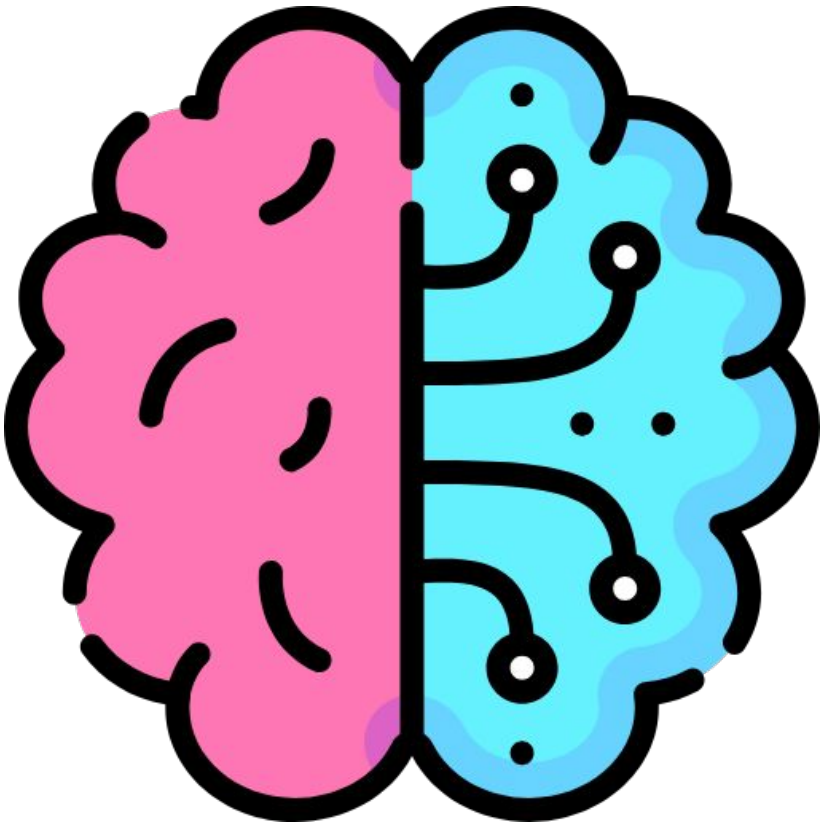
- Let’s say we have a need to reorganize our book shelves. However, if we do so, we would need to change the codes of lot of books.
- Example:
- If we remove shelf number 2, shelf 3 becomes the new shelf 2, and shelf 4 becomes the new shelf 3.
- Suppose we had a book called "The Alchemist" on shelf 3, coded as 3B. Now, since our old shelf 3 is the new shelf 2, "The Alchemist" should have a new code of 2B.



- This means we would need to update our system with the new codes for all these books. If we don’t do it that would be a problem. Why is this a problem?
- Imagine someone wants to borrow "The Alchemist" now. If our system isn't updated. So it will still show the book at code 3B.
- The librarian goes to shelf 3B but cannot find “The Alchemist” there. Instead she finds a different book there.
- Therefore, we would have to update our system with new codes for all these books which is a big hassle.



- **Consistent Hashing**, a special type of hashing, is a smart algorithm that minimizes these reorganizations.
- Even if shelves change, most books will still keep their original codes. Only a few need to be changed, making it much easier to manage.
- We won’t go deep into how it works, as that would be out of the scope of this course. Just imagine it as a magic algorithm that will help us minimize all these re-organizational hassles.



Step 05

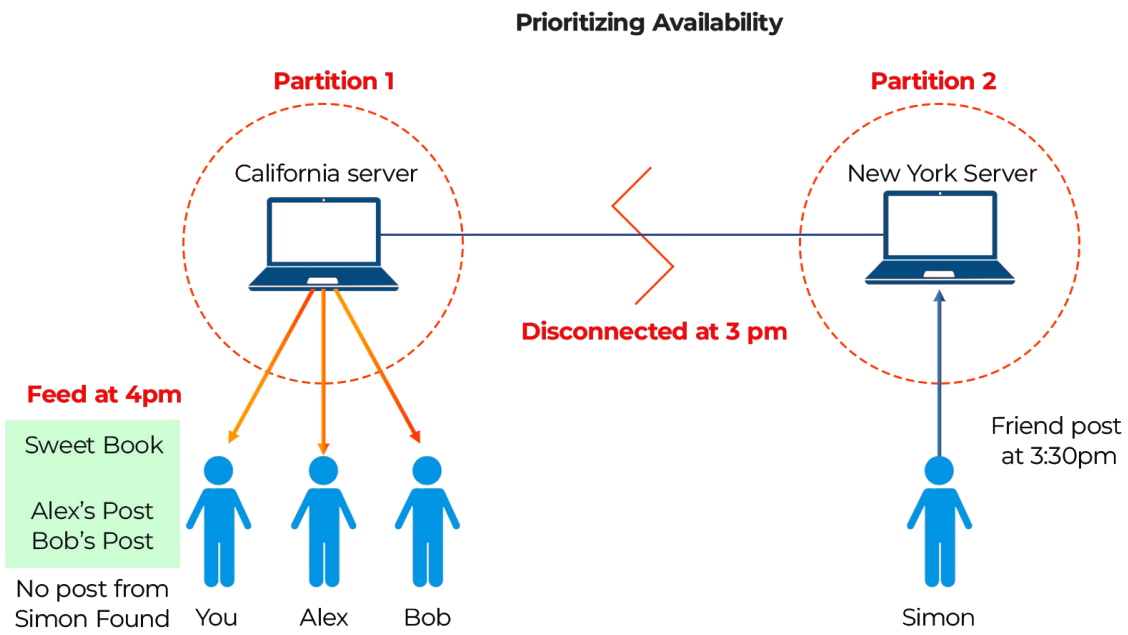
CAP Theorem

CAP Theorem

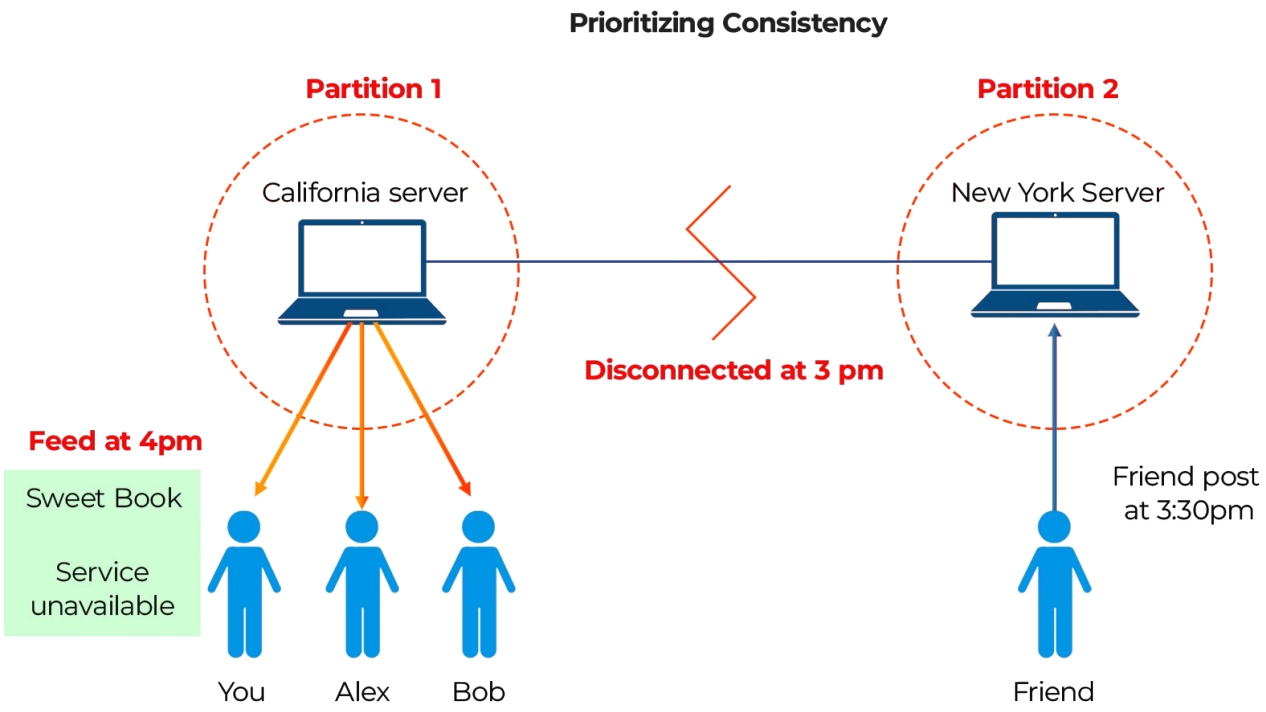
- Ideally you would want that your system to be both consistent and available.
- Lets say an accident happens that creates a partition in our system.
- How will you tolerate the partition?
- The CAP theorem says that either you can make your system available or you can make it consistent. You can't have both at the same time.

Example:

- Consider there is a social media company 'SweetBook'. They have servers all around the world.
- At 3pm, an accident happens and the connection between their New York and San Francisco servers is lost. They have a partition now.
- At 3:30pm, your friend in New York posts something on social media.



- Now there could be two scenarios:
- If SweetBook prioritizes availability, the site remains accessible, but you won't see the new post from your friend in New York—only posts from local San Francisco users.
- If SweetBook prioritizes consistency, you might see a message like "Website not available" when you try to access it from San Francisco.
- You cannot have both at the same time.



That's it folks!

The Learning Continues...

Did you know this?

Free Resources

Discover free advanced System Design resources at sweetcodey.com – dive deep and master the art! 🚀

Your Contribution

If you love this book please give us a rating at: bit.ly/sdplay

Join Our Community

Credits

Anmol Gupta (Graphic Designer)
Icons made by Smashicons from www.flaticon.com
Icons made by Vectors Market from www.flaticon.com
Icons made by Pixel perfect from www.flaticon.com
Icons made by Maxim Basinski Premium from www.flaticon.com
Icons made by Freepik from www.flaticon.com
Icons made by Eucalyp from www.flaticon.com
Icons made by juicy_fish from www.flaticon.com
Icons made by mikan933 from www.flaticon.com
Icons made by Md Tanvirul Haque from www.flaticon.com
Icons made by Frey Wazza from www.flaticon.com
Icons made by smashingstocks from www.flaticon.com
Icons made by Witdhawaty from www.flaticon.com
Icons made by flatart_icons from www.flaticon.com
Icons made by Dreamcreateicons from www.flaticon.com
Icons made by kerismaker from www.flaticon.com
Icons made by Parzival' 1997 from www.flaticon.com
Icons made by logisstudio from www.flaticon.com
Icons made by orvipixel from www.flaticon.com
Icons made by Karyative from www.flaticon.com
Icons made by HAJICON from www.flaticon.com
Icons made by Kalashnyk from www.flaticon.com
Icons made by bsd from www.flaticon.com
Icons made by Indygo from www.flaticon.com
Icons made by Uniconlabs from www.flaticon.com
Icons made by Iconjam from www.flaticon.com
Icons from www.freepik.com
www.iconduck.com/icons/2593/cache

Thanks!

Do you have any questions or suggestions?

hello@sweetcodey.com
www.sweetcodey.com