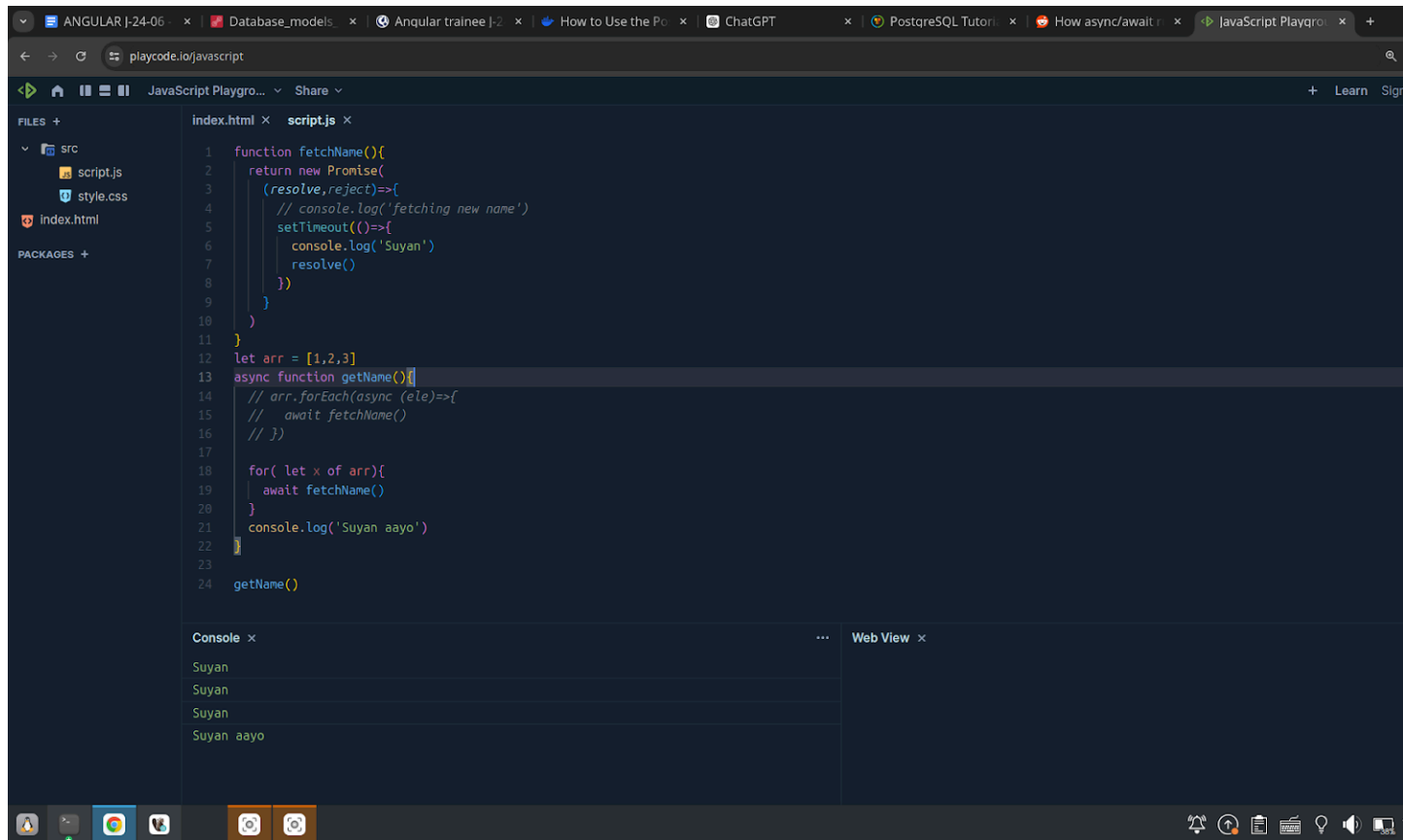


forEach and async



```
1 function fetchName(){
2   return new Promise(
3     (resolve,reject)=>{
4       // console.log('fetching new name')
5       setTimeout(()=>{
6         console.log('Suyan')
7         resolve()
8       })
9     }
10  )
11 }
12 let arr = [1,2,3]
13 async function getName(){
14   // arr.forEach(async (ele)=>{
15   //   await fetchName()
16   // })
17   for( let x of arr){
18     await fetchName()
19   }
20   console.log('Suyan aayo')
21 }
22
23
24 getName()
```

Console x

```
Suyan
Suyan
Suyan
Suyan aayo
```

Web View x

Day18

MVCC

- a. Key aspect of postgresSQL for concurrency control.
- b. Problem solved by MVCC**
 - i. Any database user can read or write the database.
 - ii. Now, while reading data from a database, it won't cause any concurrency issues or so.
 - iii. But, if a database is being updated by a transaction, then the database locks the data which is currently being updated.
 - iv. This will make it challenging for readers to view the data, cause concurrency issues and also can cause deadlocks.

c. How it solved the problem

- i. MVCC creates several versions of a single database record, enabling various transactions to access different versions of one database record without conflicting with one another.
- ii. This will help transactions run simultaneously and also there won't be any need of locking and blocking.

d. How MVCC works

- i. The whole behavior of MVCC is controlled by snapshots which will determine what users can see in the database.
- ii. Each transaction has its own snapshot, which represents the state of the database at the beginning of the transaction.
- iii. This snapshot will have all the info about rows, etc in the database.
- iv. Each transaction will just operate on a snapshot of the database, and won't interfere with any other transactions.

e. Example on working on MVCC

Product ID	Product Description	Amount (USD)	Customer Name
100121	Apple Headphones	500.00	Raymond Ryan
100122	T-shirt	102.00	Raymond Ryan
100122	C-Type USB Cable	68.00	Harry Abubakar
100124	Pair of Spanner	121.00	Steven Snipes

- i. Suppose we have this table, and there are two transactions T1 and T2, which are concurrently trying to update the value of the first item (100121) from 500 to 550 and 580 respectively.
- ii. Both T1 and T2 will create their own snapshots of the database table.
- iii. Now while updating the value, suppose T1 finishes first, and then the database's value for the first record becomes 550 (the new version of the database record with an amount of \$550.00 becomes the current version.).
- iv. Now when T2 tries to update the database accordingly, it sees that snapshot taken (it was taken at start of transaction, and is different from the one after T1 has committed)
- v. Hence T2 won't be able to update the already modified table.
- vi. Then T2's transaction will create a serialization error, indicating that it conflicted with T1's transaction because T2's snapshot is outdated.
- vii. So, T2 will have to refresh its snapshot by starting a new transaction and taking a new snapshot of the database (with price 550)
- viii. Then T2 will change the amount from 550 to 580, and then commit its changes.

Changing pw for Postgres

```
supersuyan@supersuyan:~$ sudo -i -u postgres
postgres@supersuyan:~$ psql
psql (16.3 (Ubuntu 16.3-0ubuntu0.24.04.1))
Type "help" for help.

postgres=# # ALTER USER postgres with password 'admin';
ERROR:  syntax error at or near "#"
LINE 1: # ALTER USER postgres with password 'admin';
        ^
postgres=# ALTER USER postgres with password 'admin';
ALTER ROLE
postgres=#
```

CAP Theorem

The CAP theorem states that it is not possible to guarantee all three of the desirable properties – consistency, availability, and partition tolerance at the same time in a distributed system with data replication.

a. Consistency

- i. guarantee that every node in a distributed cluster returns the same, most recent and a successful write.
- ii. every client having the same view of the data.
- iii. For the write to one node to succeed, it must also instantly be replicated or forwarded to all the other nodes in the system.
- iv. Difference between consistency of ACID vs CAP
 1. In CAP, consistency means having information that is the most up-to-date.
 2. Consistency in ACID refers to the hardness of the database that protects it from corruption

b. Availability

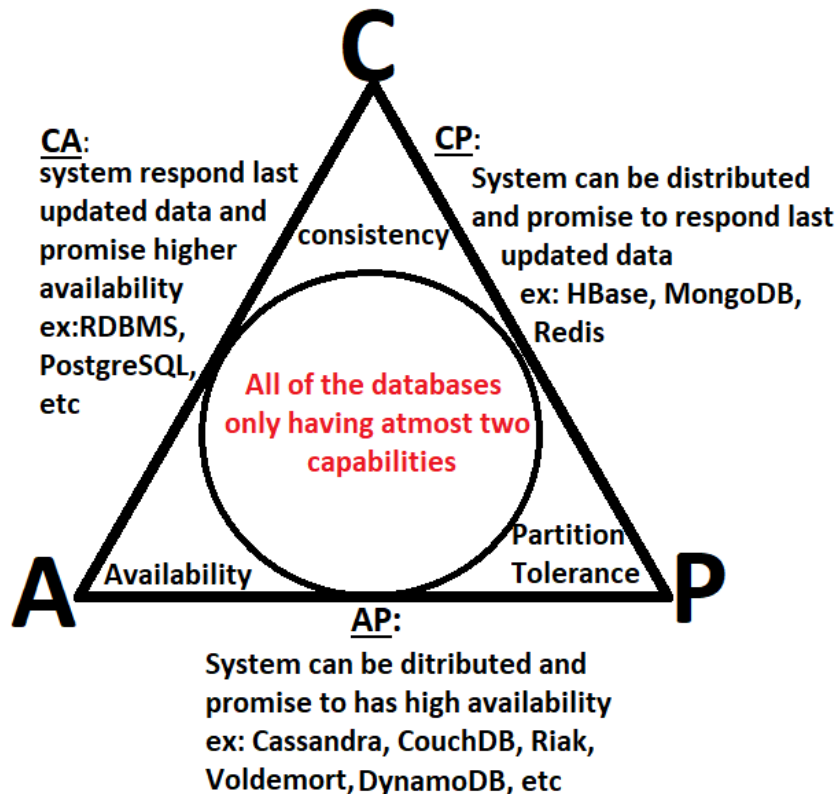
- i. Even if one or more nodes are down, any client making a data request receives a response.
- ii. Every non-failing node returns a response for all the read and write requests in a reasonable amount of time.
- iii. An available system ensures that every request eventually receives a response, though **it doesn't guarantee that the response contains the most recent data.**

c. Partition Tolerance

- i. In a distributed system, a partition is a break in communications—a temporarily delayed or lost connection between nodes.
- ii. Partition tolerance means that in spite of any number of breakdowns in communication between nodes in the system, the cluster will continue to work.

What do CA, AP, CP mean?

- a. CP
 - i. The practical result is that when a partition occurs, the system must make the inconsistent node unavailable until it can resolve the partition.
 - ii. MongoDB and Redis are examples of CP databases.
- b. AP
 - i. All nodes remain available when a partition occurs, but some might return an older version of the data.
 - ii. CouchDB, Cassandra, and ScyllaDB are examples of AP databases.
- c. CA
 - i. A CA database delivers consistency and availability, but it can't deliver fault tolerance



nodes in the system have a partition between them.

nce if any two

BASE

- a. Basically Available, Soft State, and Eventual Consistency.
- b. Basically Available**
 - i. the database system should always be available to respond to user requests, even if it cannot guarantee immediate access to all data.
 - ii. The focus is on ensuring that the system remains operational and able to handle requests, even in the face of network partitions or other failures.
- c. Soft State**
 - i. The focus is on ensuring that the system remains operational and able to handle requests, even in the face of network partitions or other failures.
 - ii. This is because the data is not necessarily synchronized across all nodes at all times.
 - iii. This can happen due to the effects of background processes, updates to data, and other factors.
 - iv. The database should be designed to handle this change gracefully, and ensure that it does not lead to data corruption or loss.
- d. Eventual Consistency**
 - i. the database should eventually converge to a consistent state, even if it takes some time for all updates to propagate and be reflected in the data.
 - ii. The system will become consistent over time, given that no new updates are made.
 - iii. If you wait long enough, all replicas of a piece of data will converge to the same value.
 - iv. This approach accepts temporary inconsistencies in exchange for improved availability and partition tolerance.
- e. Summary**
 - i. Basically Available: Guarantees availability of the system.
 - ii. Soft state: Allows for a temporary state where data may be inconsistent.
 - iii. Eventual consistency: Ensures that the system will eventually become consistent.
- f. Example**

Consider a social media application where user posts and comments are distributed across many servers. In a BASE system, when a user posts a comment, it might not be immediately visible to all other users because the update needs to propagate across multiple servers. However, the system ensures that the comment will eventually become visible to all users. This approach ensures that the application remains available and responsive, even if some servers are temporarily unreachable.

ACID	BASE
ACID (Atomicity, Consistency, Isolation, Durability) is a set of properties that guarantee the integrity and consistency of data in a traditional database.	The BASE properties are a more relaxed version of ACID that trade off some consistency guarantees for greater scalability and availability.
The primary difference between the two is that ACID requires immediate consistency,	while BASE only requires eventual consistency.
ACID is better suited to traditional transactional databases.	The BASE is more suitable for use in large-scale, highly-available systems,

Handling junction tables

- for every many-to-many association, we will need an additional table, known as a junction table.
- A junction table in the database bridges the table together by referencing the primary key of each table.
- Example

```
CREATE TABLE article (
  id SERIAL PRIMARY KEY,
  title TEXT
)

CREATE TABLE tag (
  id SERIAL PRIMARY KEY,
  tag_value TEXT
)
```

```
CREATE TABLE article_tag (
  article_id INT
  tag_id INT
  PRIMARY KEY (article_id, tag_id)
  CONSTRAINT fk_article FOREIGN KEY(article_id) REFERENCES article(id)
  CONSTRAINT fk_tag FOREIGN KEY(tag_id) REFERENCES tag(id)
)
```

- i. In practical use cases, each article can have multiple tags and each tag can be mapped to multiple articles.
- ii. So a bridge table called `article_tag` is created.

d. IMPORTANT

Now, what should be understood is :

- i. When working with a one to one relationship or many to one or one to many tables, **the primary key of side “one” is kept as foreign key in “many” side.**
- ii. But since for many to many relationships, there is no “one” side in any side. So, a junction table is necessary.
- iii. Let's take an example, where one author can have multiple articles.

```
CREATE TABLE author (  
  id SERIAL PRIMARY KEY,  
  name TEXT  
)  
  
CREATE TABLE article (  
  id SERIAL PRIMARY KEY,  
  author_id INT NOT NULL,  
  title TEXT NOT NULL,  
  content TEXT NOT NULL,  
  CONSTRAINT fk_author FOREIGN KEY(author_id) REFERENCES author(id)  
)
```

Another example of many to many

```
CREATE TABLE songs (  
  id integer PRIMARY KEY,  
  name varchar(100)  
);  
  
CREATE TABLE artists (  
  id integer PRIMARY KEY,  
  name varchar(100)  
);  
  
CREATE TABLE songs_artists (  
  artist_id integer REFERENCES artists(id),  
  song_id integer REFERENCES songs(id),  
  PRIMARY KEY (artist_id, song_id)  
);
```

Joins

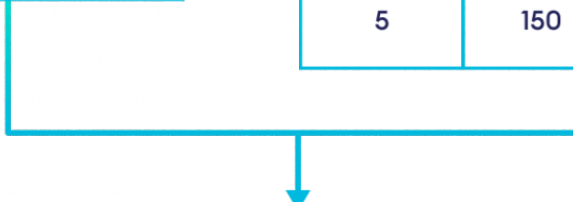
SQL JOIN

Table: Customers

customer_id	first_name
1	John
2	Robert
<u>3</u>	David
4	John
<u>5</u>	Betty

Table: Orders

order_id	amount	customer
1	200	10
2	500	<u>3</u>
3	300	6
4	800	<u>5</u>
5	150	8



customer_id	first_name	amount
3	David	500
5	Betty	800

Suppose we have initial data like this

Customer Table :

	customer_id [PK] bigint	first_name character varying (50)	last_name character varying (50)	address_id bigint
1	1	Mary	Smith	5
2	3	Linda	Williams	7
3	4	Barbara	Jones	8
4	2	Madan	Mohan	6

Payment Table :

	customer_id [PK] bigint	amount bigint	mode character varying (50)	payment_date date
1	1	60	Cash	2020-09-24
2	2	30	Credit Card	2020-04-27
3	8	110	Cash	2021-01-26
4	10	70	mobile Payment	2021-02-28
5	11	80	Cash	2021-03-01









a. INNER JOIN

- i. matching values from both table

Query Query History

```
1 SELECT c.first_name, p.amount, p.mode
2 FROM customer AS c
3 INNER JOIN payment AS p
4 ON c.customer_id = p.customer_id
```

Data output Messages Notifications

	first_name character varying (50)	amount bigint	mode character varying (50)
1	Mary	60	Cash
2	Madan	30	Credit Card

b. LEFT JOIN

- i. returns all records from left table and matched records from right table

Query

Query History

1

SELECT *

2

FROM customer AS c

3

LEFT JOIN payment AS p

4

ON c.customer_id = p.customer_id

5

Data output

Messages

Notifications

≡+

▼

		first_name character varying (50)		last_name character varying (50)		address_id bigint		customer_id bigint		amount bigint	
1	1	Mary		Smith		5		1		60	
2	2	Madan		Mohan		6		2		30	
3	4	Barbara		Jones		8		[null]		[null]	
4	3	Linda		Williams		7		[null]		[null]	

SQL LEFT JOIN

Table: Customers

customer_id	first_name
1	John
2	Robert
<u>3</u>	David
4	John
<u>5</u>	Betty

Table: Orders

order_id	amount	customer
1	200	10
2	500	<u>3</u>
3	300	6
4	800	<u>5</u>
5	150	8



customer_id	first_name	amount
1	John	
2	Robert	
3	David	500
4	John	
5	Betty	800

c. RIGHT JOIN

Query		Query History	
1	SELECT *		
2	FROM customer AS c		
3	RIGHT JOIN payment AS p		
4	ON c.customer_id = p.customer_id		
5			

Data output		Messages		Notifications	
	customer_id bigint	first_name character varying (50)	last_name character varying (50)	address_id bigint	customer_id bigint
1	1	Mary	Smith	5	1
2	2	Madan	Mohan	6	2
3	[null]	[null]	[null]	[null]	8
4	[null]	[null]	[null]	[null]	10
5	[null]	[null]	[null]	[null]	11

SQL RIGHT JOIN

Table: Customers		Table: Orders		
customer_id	first_name	order_id	amount	customer
1	John	1	200	10
2	Robert	2	500	3
3	David	3	300	6
4	John	4	800	5
5	Betty	5	150	8

customer_id	first_name	amount
3	David	500
5	Betty	800
		200
		300
		150