→ type BMW struct {
}

→ func (t BMW) start () {
fmt. Println ("starting your BMW")
}

→ func main () {
mondayCar := Tesla { }
tuesdayCar := BMW{ }

universalRemote := &button { // taking button as
car : mondayCar, because receiver
} pointer

universalRemote. press () // Starting your Tesla

// Next day

universalRemote. Car = tuesdayCar
universalRemote. press () // Starting your BM
}

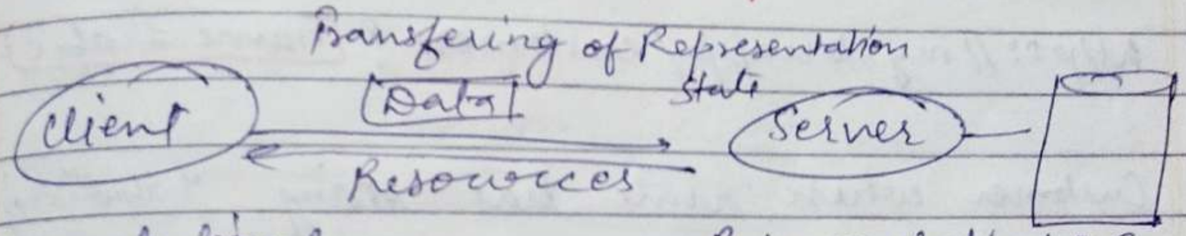
⟹ How REST API's are implemented over HTTPS ?

→ Representation
→ Representational State
→ Resource
→ Headers

- Request & Response
- Request Payload
- Status Codes

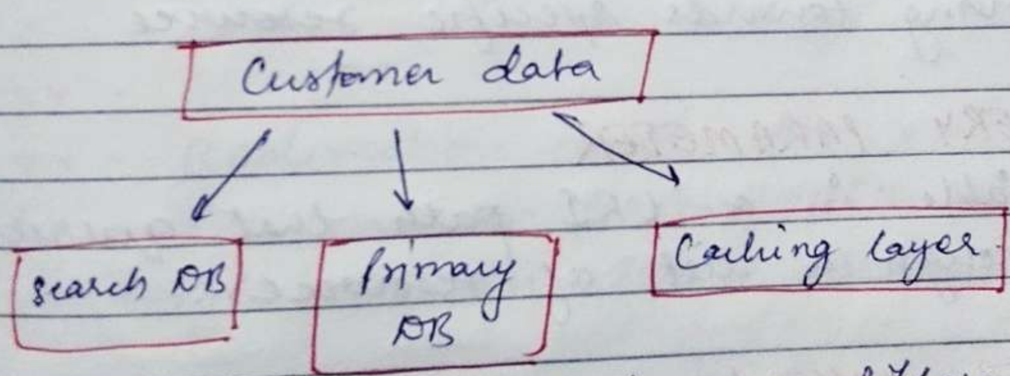
REST

Representation — Representational State



Representational State 1 of Resource

Representational DB State 2 of resource



Each layer is going to have a different REPSENTATION of this data/ resource

DNS ↑ resource name

https://mystoreapp/ customer / 1 → PATH PARAMETER

API endpoint / PATH

- GET
- POST
- PUT
- PATCH
- DELETE

https://mystoreapp/customer/1/orders

All the orders of customer 1

Customer has orders

Query Parameter

https://mystoreapp/customers? name = abc

Customer whose name has string "abc" in it

PATH PARAMETER

Variable in a URI path that ~~hef~~ helps in pointing towards specific resource

QUERY PARAMETER

Variable in a URI path that queries/filters through a list of resources.

HTTP method

PUT/POST/UPDATE

API path

https://mystoreapp/customer/1

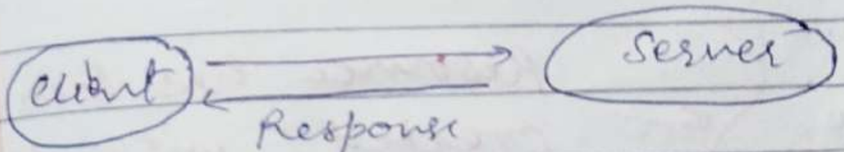
Request Payload / Request Body

```
{
  "city" : "Dubai"
  "country": "UAE"
}
```

Response body

```
{
  "id": "1",
  "updated": "true",
  ...
}
```

RESPONSE CODES



HTTP STATUS Codes

2xx - SUCCESS
3xx - Redirection and others
4xx - Problem on client side
5xx - Problem on server side

HTTP HEADERS
set of attributes that corresponds to any meta-data associated with the API request

```
{
  "accept-ranges": "bytes",
  "content-type": "application/json",

  ...
}
```

https :// localhost : 8000/ storeapp / customer

```
{
    "name" : "Sunil",
    "city" : "Faridabad"
    ---
}
```

} Request.

```
{
    er" : " ",
    "code" : 200,
    "id" : 1,
    "name" : "Sunil",
    ---
}
```

← Resource has been created (with details of resource in the body)

201 Resource has been created (resource id returned)

200 Resource has been created (details of resource returned)

204 Resource processed successfully with no content returned.

```
{
    ---
    ---
}
```

~~Request (what happens if you send something wrong in the request data?)~~

```
{
    ---
    ---
}
```

Response (Error messages and codes so that the client can understand better)

Making multiple identical requests has the same effect as making a single request

POST

IDEMPOTENT

✗

GET

IDEMPOTENT

✓

(If you send same GET request multiple times, it will give you same result.)

POST URIs

↳ can have a PATH PARAMETER

↳ should never have a Query Parameter

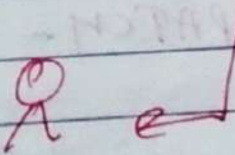
↳ Preferably have a REQUEST PAYLOAD

GET HTTP Method

↳ URI has to have a <u>path parameter</u>, incase a single resource is fetched

http://mystoreapp/customer/$\boxed{213}$

fetching data from server



User # 213

http://mystoreapp/customer?limit=10

How to use different <u>query parameter</u> we are to limit the body of response we are fetching from the server

FILTERING and PAGINATION

If the number of these orders is huge, we can use limit query parameter.

Implementation of limits, filtering and pagination is done on the backend.

STATUS CODE 200

http://mystoreapp/customer
.../customer/1
.../customer?limit=10
.../customer?limit=10&&offset=
.../customer/1/orders

STATUS CODE 200 (Resource fetched successfully)

STATUS CODE 404 (Resource not found)

PUT/PATCH - Update Resources
or
Create Resource

Resource does not exist

UPDATE

UPSERT

CREATE

Resource exists

PUT → require complete BODY of the Resource to be UPDATED

Request Method - PUT/PATCH

request body

```
{
    "name": "--"
    "age": "--"
    "city": "--"
      ---
}
```

UPSERT REQUIRES ID FROM Client

Case: 1

user

Data + ID → { Resource customer with ID exists so resource is updated and new resource is not created.

Case: 2 UPSERT uses ID from client only to update but not Create because server does not trusts.

DATA + ID →

Server can use the sent ID update after checking if ID is valid.

→ CREATING Resource using <u>PUT</u>

↳ server trusts on ID from client

↳ ID needs to be in the request body

↳ If ID is in PATH PARAMETER there will be 404

Error of PUT Request

404 - Wrong Path parameter
400 - Wrong Payload
409 - Not possible because An existing ID needed to refer the resources that has to be updated.

PUT vs PATCH

PUT → Update all attributes, replaces the whole resource

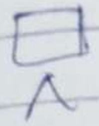
PATCH → Just changes few attributes

→ Why PATCH when PUT can update all attributes?
Updating all 100 attributes when only changes to 1 is needed, wastes time and is not efficient as we are transferring a lot of data.

→ PUT / PATCH are Idempotent.
↳ Change the state of the resources only once and not again and again.

DELETE
↳ Status code 404 is possible if the ID in path param is already deleted before.

FILTER and PAGINATION



Should all the data be transferred to the client? NO

http://localhost:8000/myappstore/customer/1/order?item=microwave

filter

FILTERING

- get only the items that are useful to you.
- Load on the server is also reduced.

http://localhost:8000/myappstore/customer/1/order?item=microwave&quantity=1

PAGINATION

http://localhost:8080/myappstore/customers!limit=2000&offset=0

Request

API

Server side

Apply max limit = 20

Return only 20 orders

Response

Pages $\boxed{1}$ 2 3 4 5 ....

2000 per paage

Request	RESPONSE	SERVER SIDE
...?limit=20& offset=0	page $\boxed{1}$ 2 3 ... 20 per page	max limit = 2
" " =20	page 1 $\boxed{2}$ 3 ... 20 per page	
" " =40	page 1 2 $\boxed{3}$ ... 20 per page	

http://localhost:8080/myappstore/customer/1/orders?
item=$\boxed{\text{microwave}}$ & limit=$\boxed{20}$ & offset=$\boxed{0}$

PAGINATION (always) and
Filtering (optional) } both

System Design

Components of System Design
- Logical Entities
- Tangible Entities (Technology)

Logical Entities	Tangible Entities
→ Data	→ Text, images, videos
→ Database	→ Mongodb, MySQL etc
→ Applications	→ Java, Golango, PHP etc
→ Cache	
→ Message Queue	→ Redis, Memecache
→ Infra	→ Kafka, RabbitMQ,
→ Communication	→ AWS, GCP, Azure
	→ API's, RPCS, Messages

Client-Server Architecture
- **Thin-Client** E-commerce sites, streaming app.
- **Thick-Client** Gaming apps, video editing apps
- **2-tier** light weight website for small business.
- **3-tier** Basic library management for school.
- **N-tier** Large scale systems (gmail, facebook)

Proxies

forward

Client → $\boxed{\text{Proxy}}$ → Server

← backend ←

Server does not know the IP Address of client

Forward proxy: Disguises a client's IP address

Block malicious traffic from reaching an origin web server.
Improve user's experience by caching external site content.

Reverse proxies
Scrubs all incoming traffic before it's sent to our backend servers.
Provides a single configuration point to manage SSL/ TSL

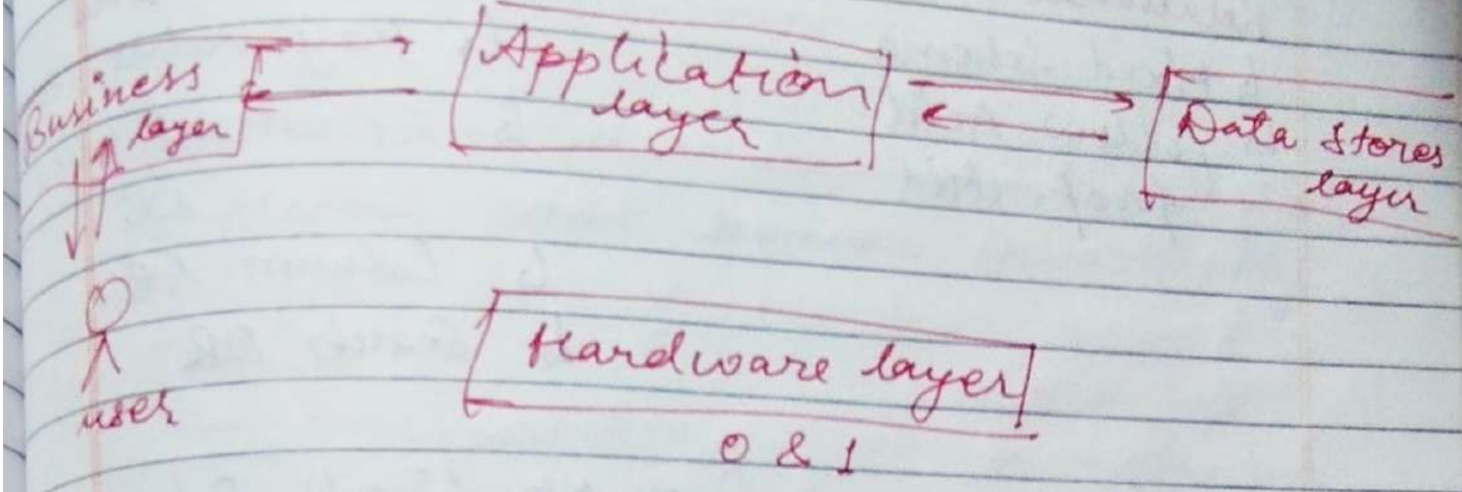
→ Date & Data flow
→ Data
→ Date format / representation
→ Mechanisms for data flow
→ Factors type, volume, scale, purpose

Bussiness layer → texts/videos/images
Application layer → JSON/XML
Data stores → index, lists, trees
Network layer → Packets
Hardware layer → Os & 1's

→ Data stores
· Databases
· Queues
· Caches
· Indexes

Data flow Methods
APIs
Messages
Events

Network layer



→ DataStores Examples

Databases → [Username, City, Address]
Cache → [Request : Response]
Queue → [send sms request, send email request]
Indexes → [most searched items, items searched in last 1 hour]

→ Type of System :-

Authorization :- login, Identity Management
Streaming :- Netflix, Hotstar, Prime Video
Transactional :- E-Commerce sites, ride booking apps, grocery ordering apps.
Heavy Compute System :- Image recognition, video processing using ML models.

→ Databases

Relational
- ↳ Fixed Schema
- ↳ follow Acid properties

Non-Relational
- ↳ Key-value store
- ↳ Document based DB
- ↳ Column DB
- ↳ Search DB.

- → No fixed Schema
- → Does not follow ACID properties

→ ACID (Atomicity, Consistency, Isolation, Durability)

→ Bank App Example.

Atomicity : Debited from one credit to another.

Consistency :

what is the account balance of id 141

Request2.

400Rs $\boxed{\text{balance}}$

Request 1

It should be same amount for both requests

1000 - 500 = 500

PAGE No
DATE: / /201

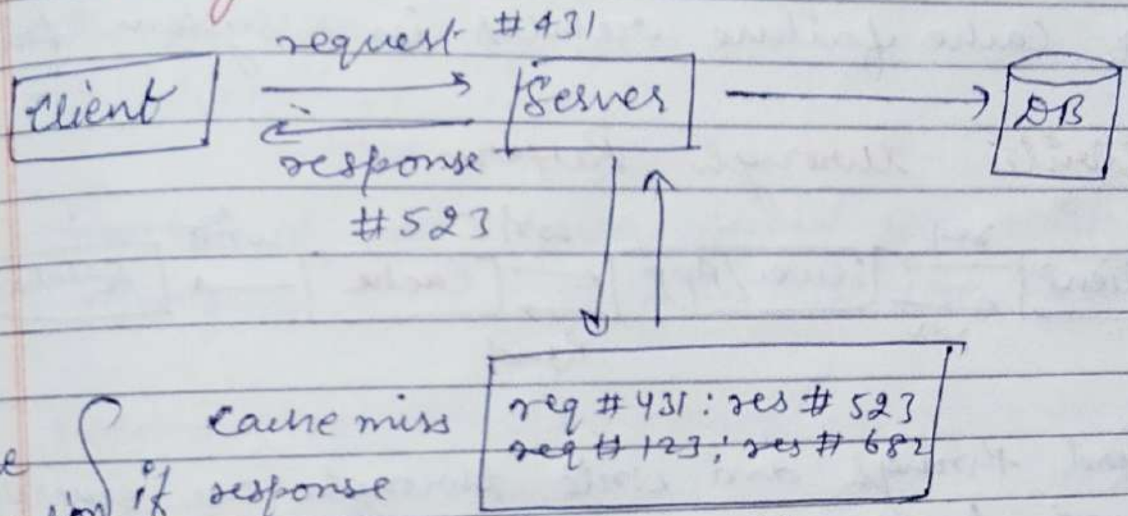
Isolation :-

balance (updated balance 500 after deduction)

Read does not know about write and vice-versa

Durability :- Guarantee that transactions completed will survive permanently

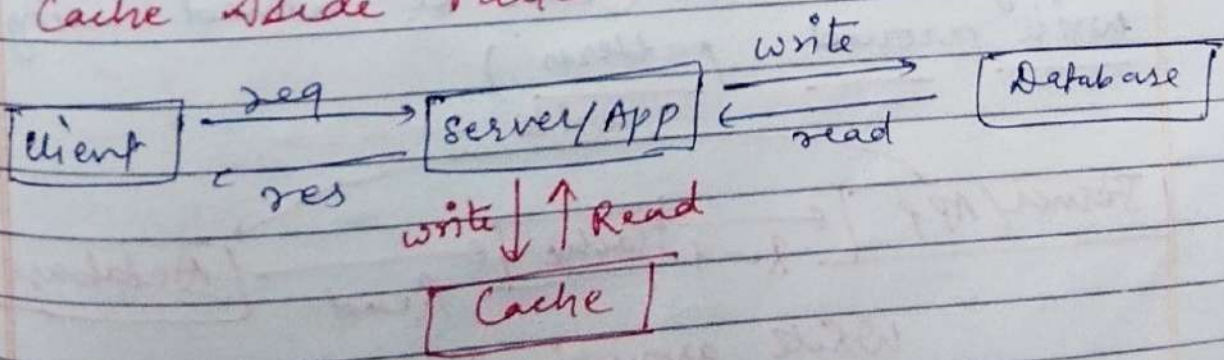
→ **Caching**



Cache invalidation

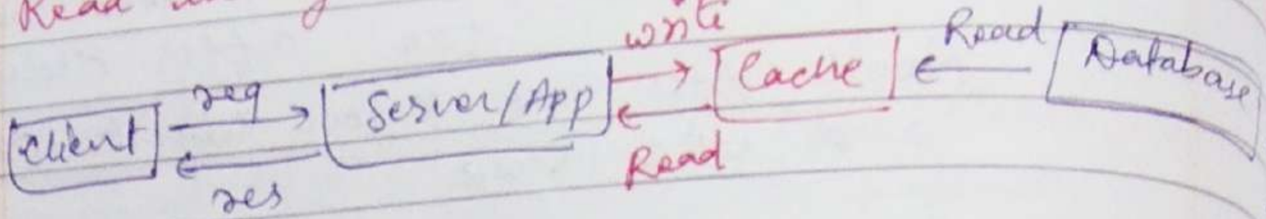
Cache miss if response not found in cache memory

→ **Cache Aside Pattern**



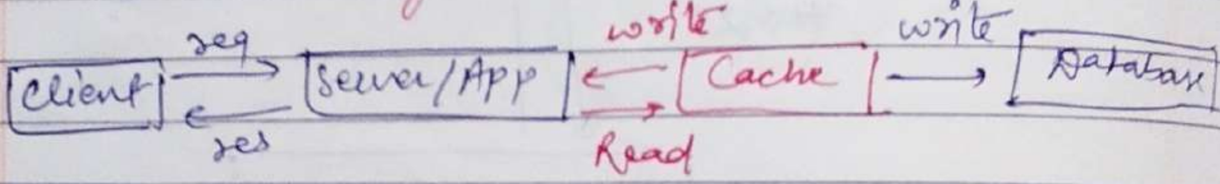
- Support heavy reads
- works even if cache goes down
- TTL/application code have to be used to keep DB and cache consistent.

→ Read through Pattern



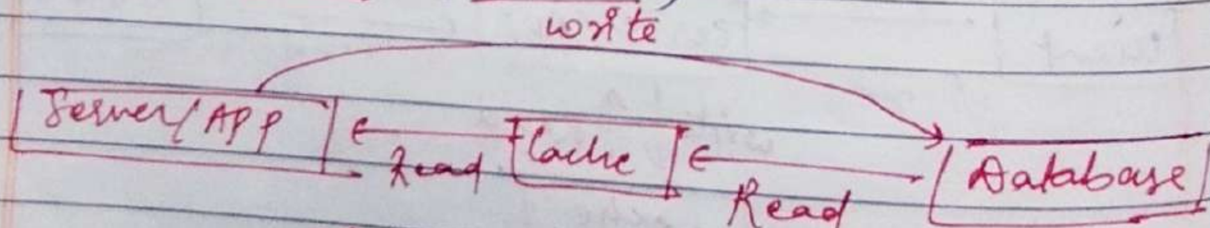
→ Great ~~alternative~~ alternate for read heavy workloads Example newsfeed.
→ Data modelling of cache and DB have to be similar.
→ Cache failure results in system failure

→ Write through Pattern



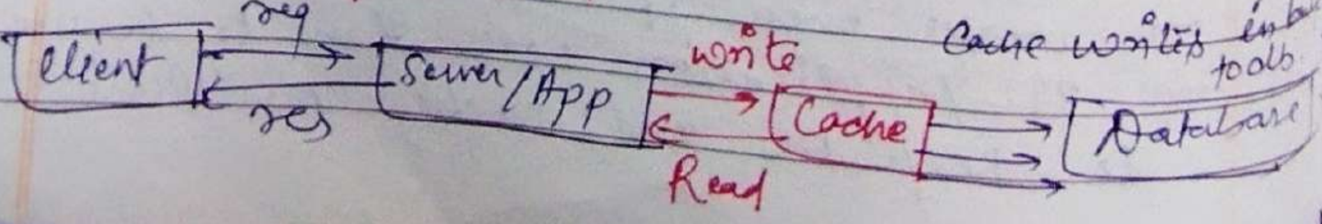
Read through and write through are generally used together

→ Caching layer adds extra layer of latency while writing to the DB (can be solved using write around pattern)



Write around Pattern

→ Write back Pattern



- useful for write heavy workloads
- Database failure can be sustained for the time cache keeps data in bulk
- used by various DBs internal implementation.
- Cache failure results in system failure.

- Performance Metrices.
  - Throughput = load / Time taken
  - Bandwidth
  - Latency
  - Response Time

number of API calls served per unit time
throughput in 30 mins is 1000 orders.

- Performance of Components
  - Applications
  - DBs
  - Caches
  - Workers
  - Message Queues.
  - Memory
  - CPU.