

CM4025: Enterprise Web Systems


Multi-tiered architectures

This lecture covers:

- Last week's lab
- Presentation, Application, Data tiers
- 1-tier, 2-tier, 3-tier, n-tier architectures
- Caches, Proxies, Load balancers and Queues

Last week's lab (one tier)

Front
end



Just Some User Input

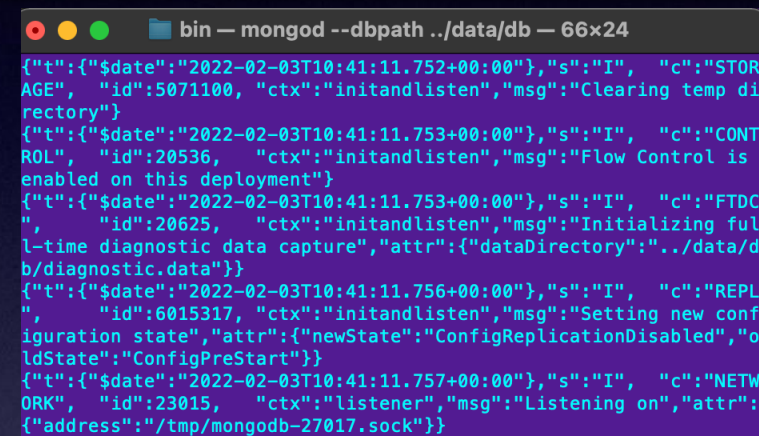
127.0.0.1:8080

Input 1:

Input 2:

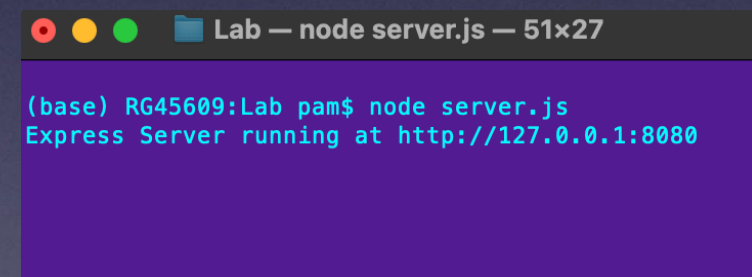
Response

Server response will go here



```
bin — mongod --dbpath ../data/db — 66x24
{"t":{"$date":"2022-02-03T10:41:11.752+00:00"},"s":"I", "c":"STOR
AGE", "id":5071100, "ctx":"initandlisten","msg":"Clearing temp di
rectory"}
{"t":{"$date":"2022-02-03T10:41:11.753+00:00"},"s":"I", "c":"CONT
ROL", "id":20536, "ctx":"initandlisten","msg":"Flow Control is
enabled on this deployment"}
{"t":{"$date":"2022-02-03T10:41:11.753+00:00"},"s":"I", "c":"FTDC
", "id":20625, "ctx":"initandlisten","msg":"Initializing ful
l-time diagnostic data capture","attr":{"dataDirectory":"../data/d
b/diagnostic.data"}}
{"t":{"$date":"2022-02-03T10:41:11.756+00:00"},"s":"I", "c":"REPL
", "id":6015317, "ctx":"initandlisten","msg":"Setting new conf
figuration state","attr":{"newState":"ConfigReplicationDisabled","o
ldState":"ConfigPreStart"}}
{"t":{"$date":"2022-02-03T10:41:11.757+00:00"},"s":"I", "c":"NETW
ORK", "id":23015, "ctx":"listener","msg":"Listening on","attr":
{"address":"/tmp/mongodb-27017.sock"}}
```

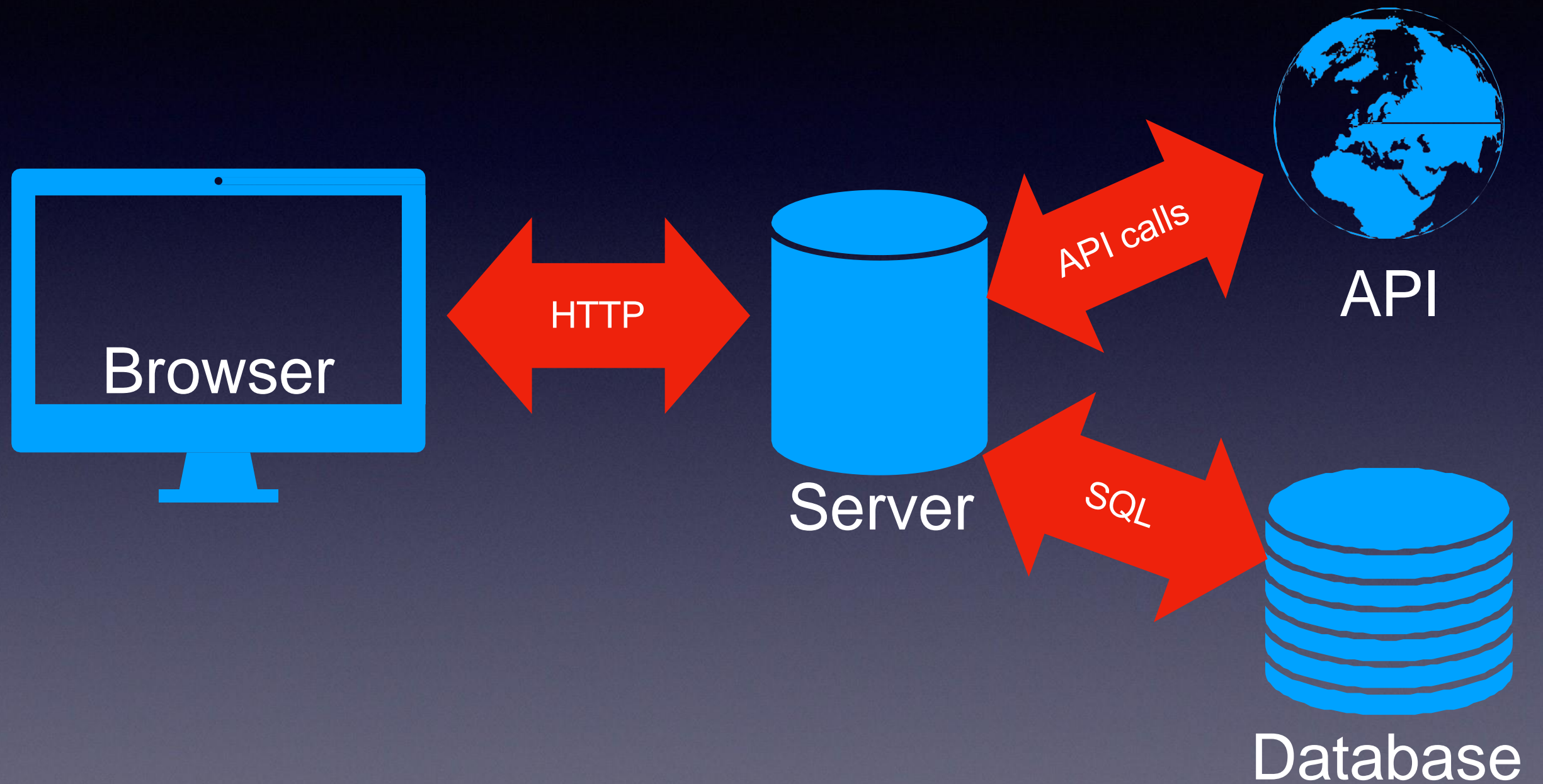
data
base



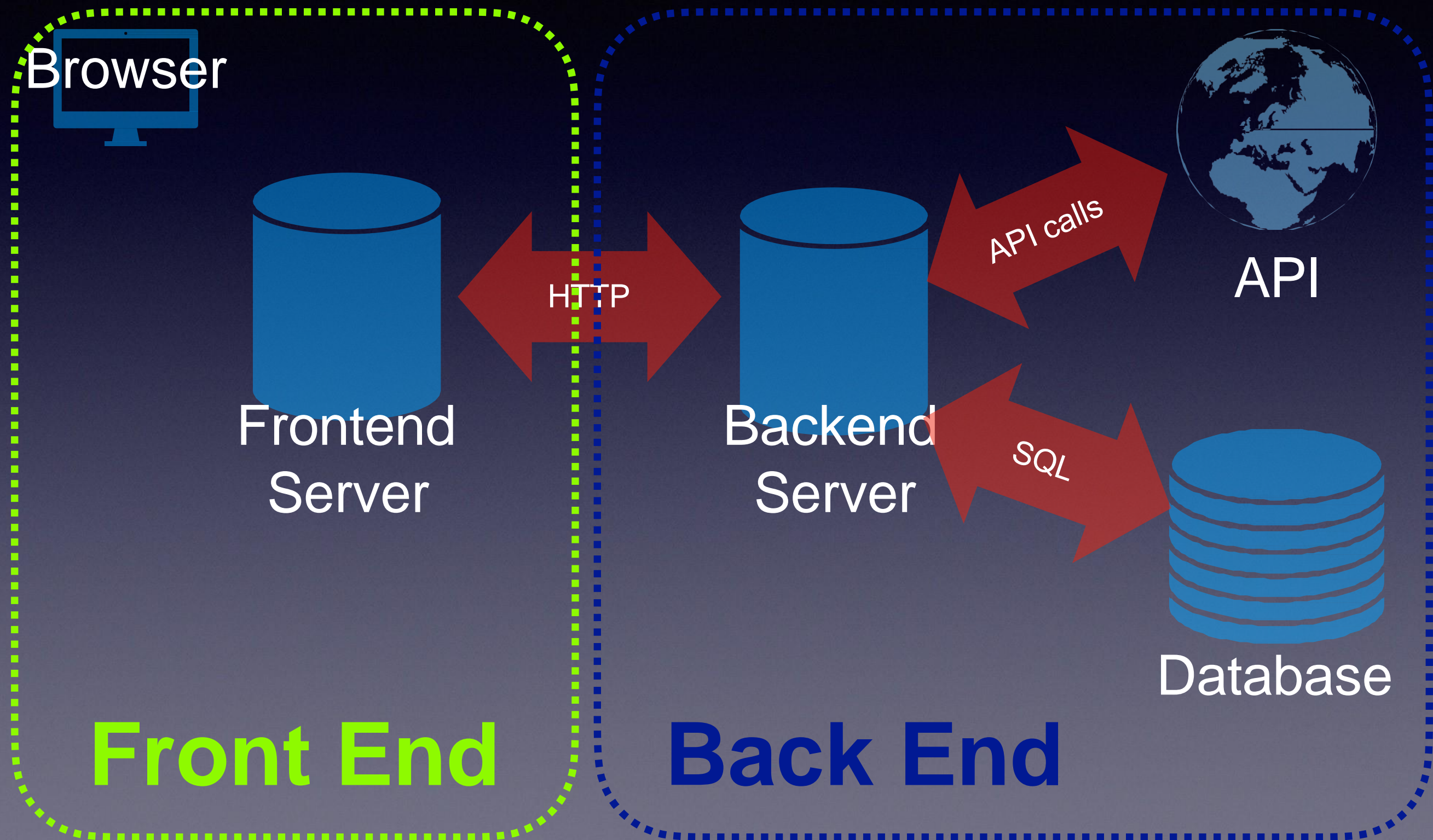
```
Lab — node server.js — 51x27
(base) RG45609:Lab pam$ node server.js
Express Server running at http://127.0.0.1:8080
```

Back
end

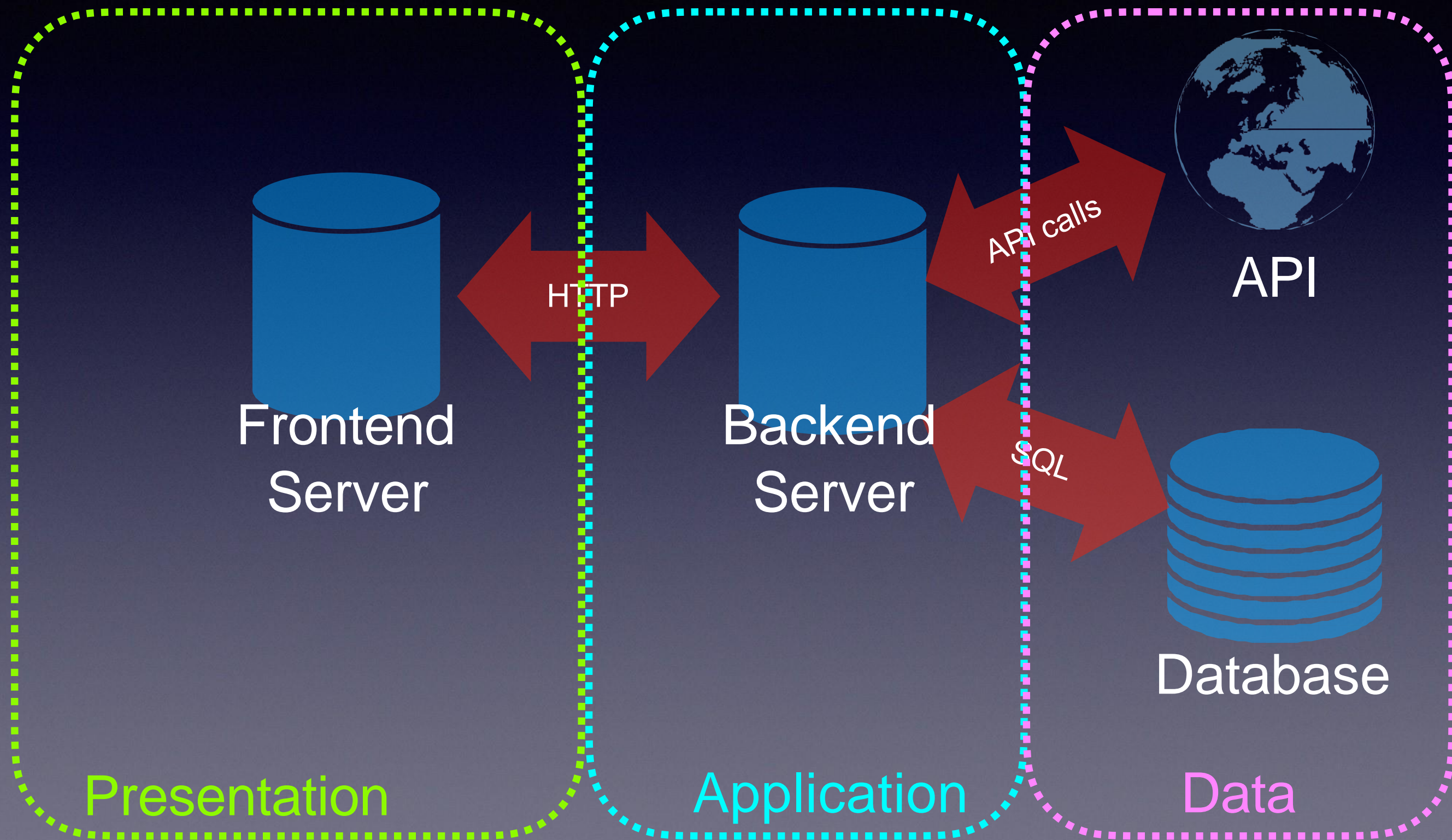
A Web System



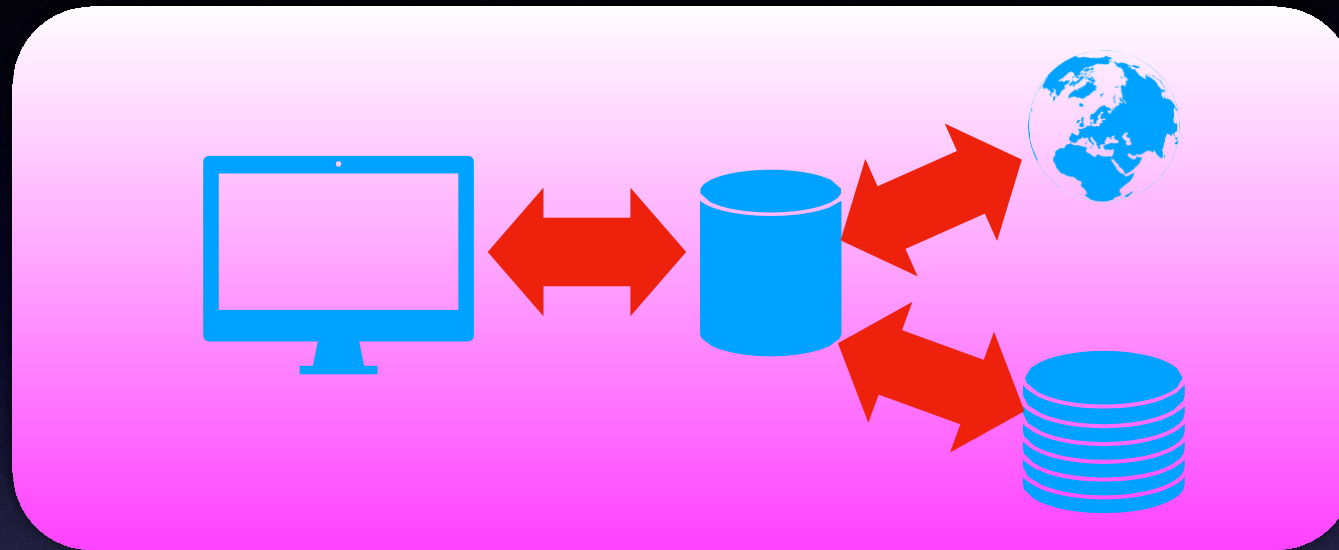
Front End and Back End



Presentation, Application, Data

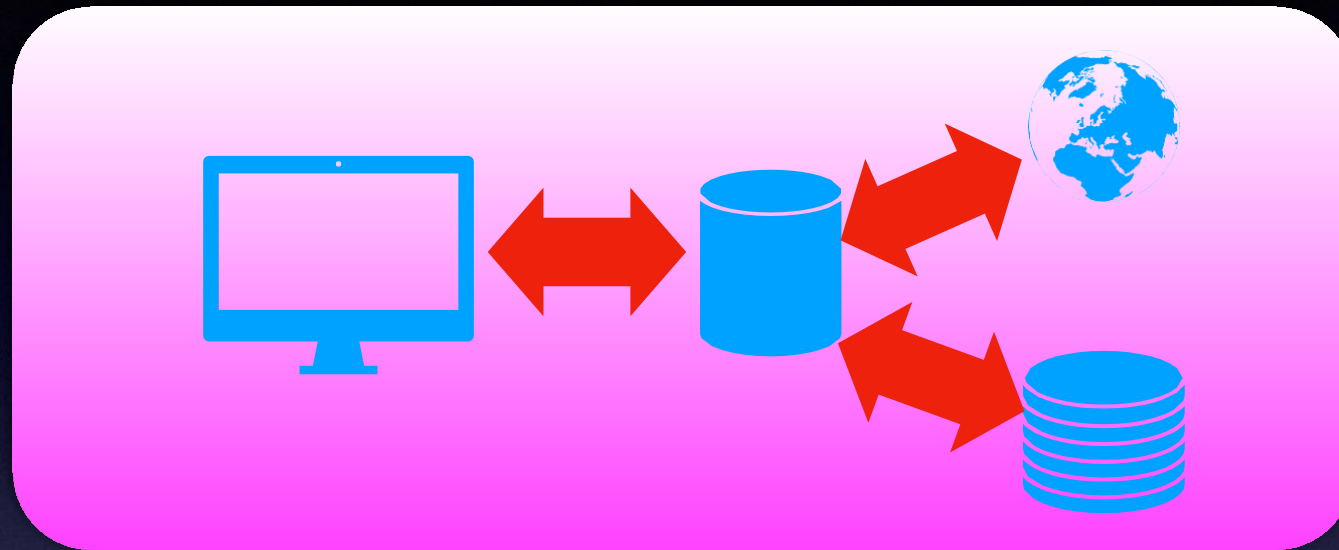


1-tier architecture



- Everything on one computer
- What you tend to develop on
- Good for simplicity
- Everything highly connected

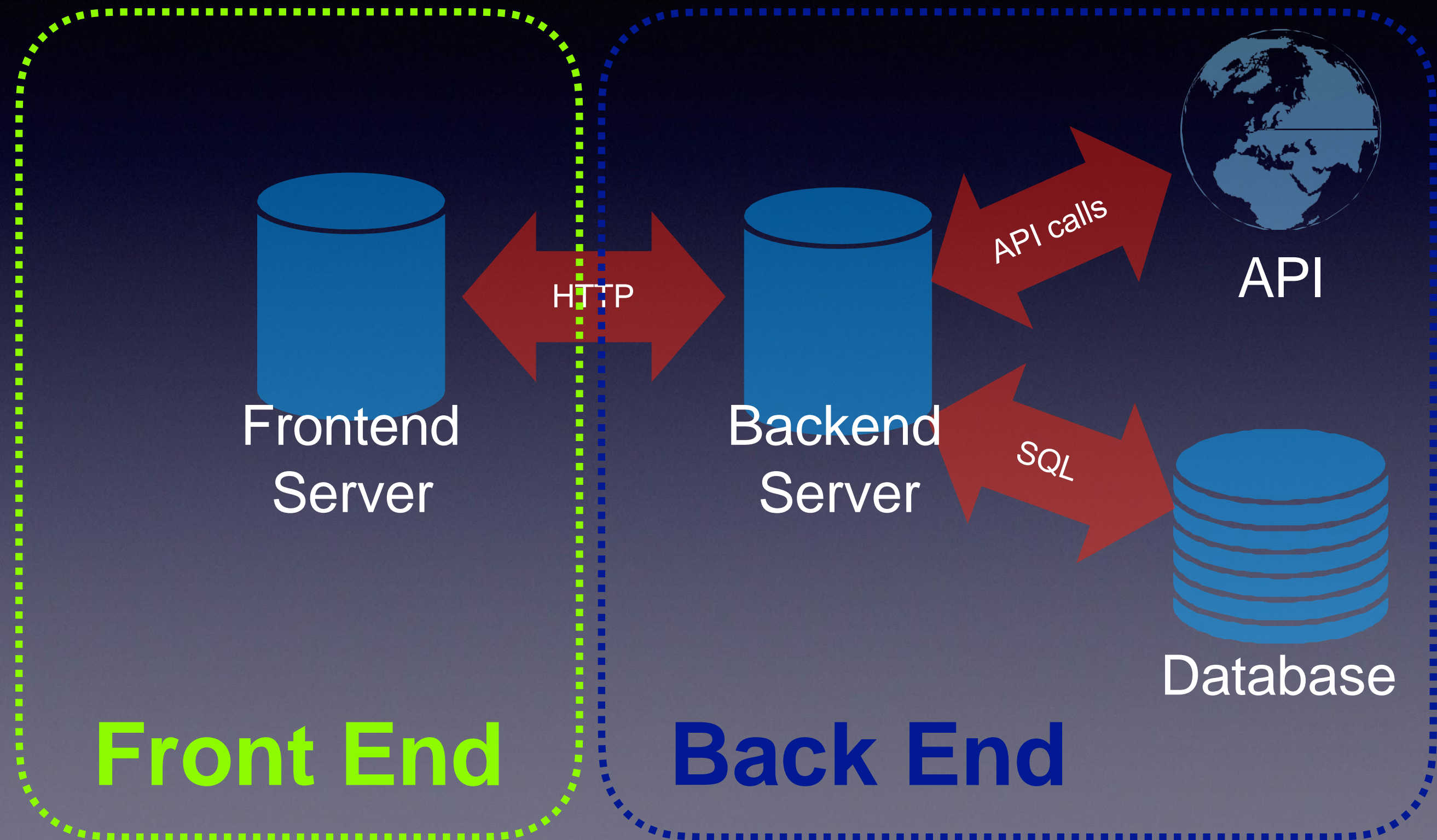
1-tier architecture



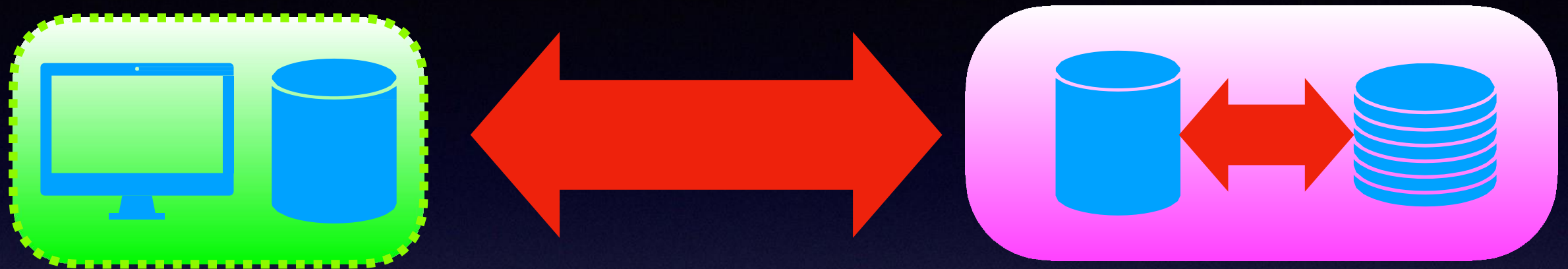
Bad for:

- Scalability
- Portability
- Maintenance

2-tier architecture

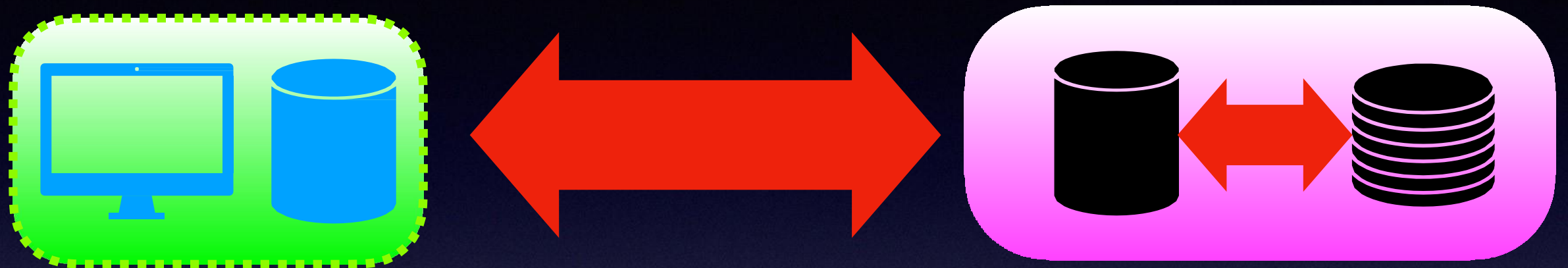


2-tier architecture



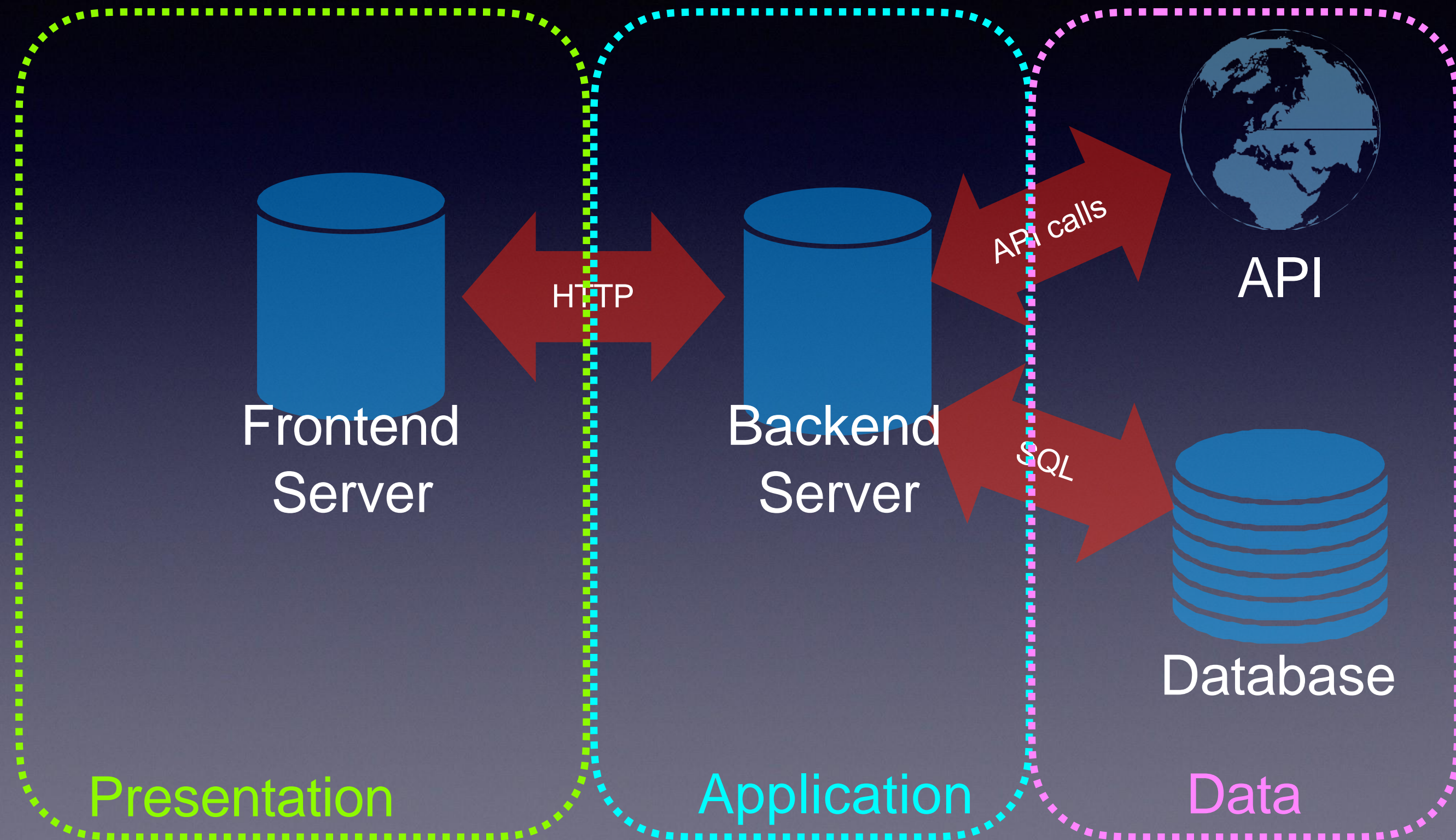
- Most of the business logic is on the front end
- Presentation and Application are tightly connected
- Back end is just Data (or database coupled with database access logic).

2-tier architecture



- Bad for server load
- Bad if you want to make changes to the business layer
- But you can swap out the database...

3-tier architecture



3-tier architecture



- Every layer can be on a different machine
- Presentation, Application and Data are loosely coupled
- All can be maintained independently

Client



- Presentation logic
- Provides User Interface
- Should not contain any “business” logic

Server



- Application Layer
- Contains rules for processing information
- Should not contain any presentation information
- Can handle multiple users efficiently

Database Server

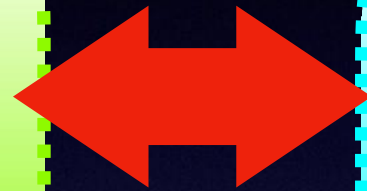


- Data storage layer
- Manages access to information

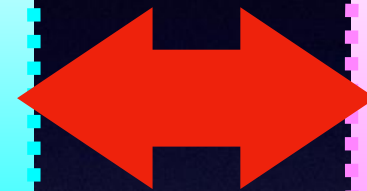
Languages



HTML
CSS
JavaScript
(plus
anything
that can
“transpile”
into these)



PHP
C#
Java
JavaScript
Ruby
Python
etc...



SQL
Database specific

Advantages

- Easier to maintain
- Easier to distribute work between the tiers
- Easier to specialise
- Components are reusable and replacable

Disadvantages

- Some functionality duplication
- Need to think before you edit
- More complicated debugging scenario
- Components are reusable...

What does a web system need?

- Availability
- Performance
- Reliability
- Scalability
- Manageability
- Cost

What does a web system need?

- Availability

- Performance

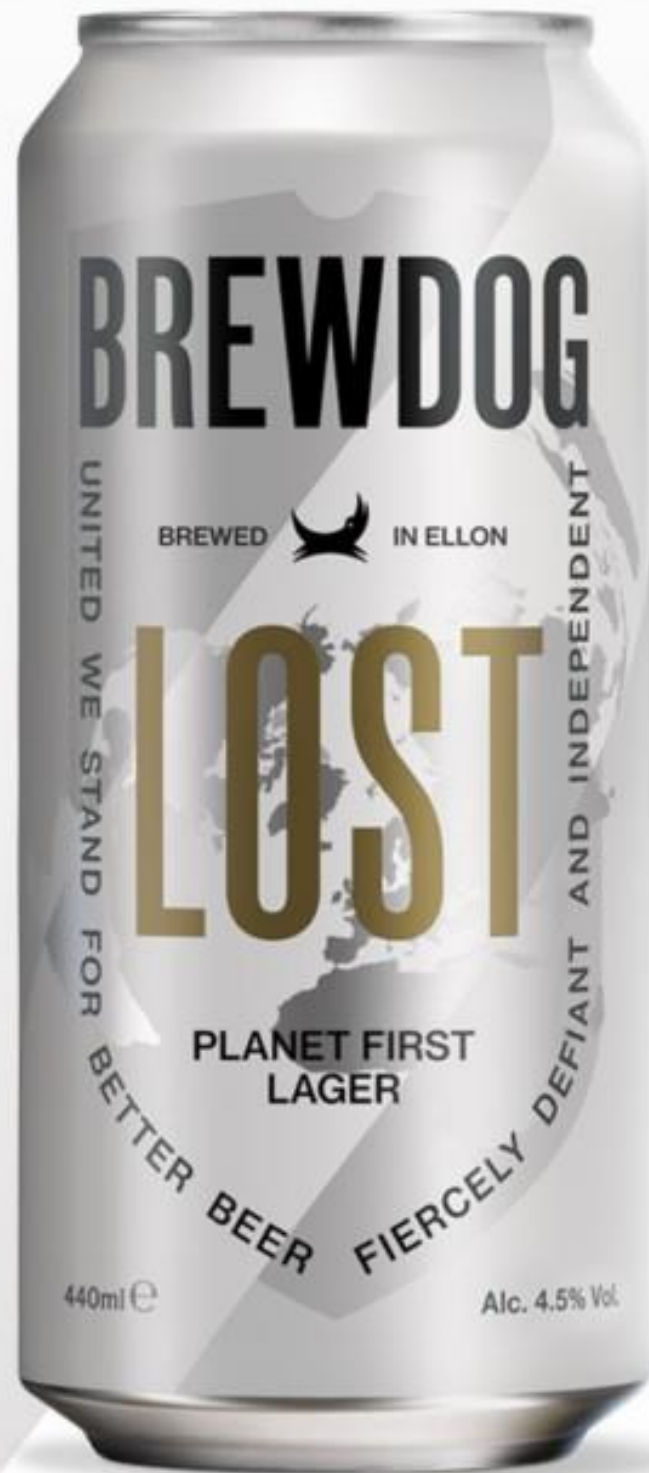
- Reliability

- Scalability

- Manageability

- Cost

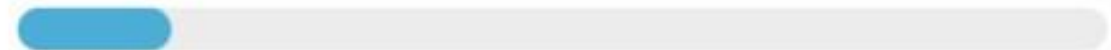
It must be available at all times



BREWDOG

YOU ARE NOW WAITING IN A QUEUE

You are now waiting in a queue for access to BrewDog.com because our website is currently over capacity. When it is your turn you will have 10 minutes to enter the website.



Number of users in queue ahead of you: 46660

Expected arrival time on the website: more than an hour

Your estimated wait time is: more than an hour

☐ Status last updated: 11:34:15

Leave the queue (You will lose your place)

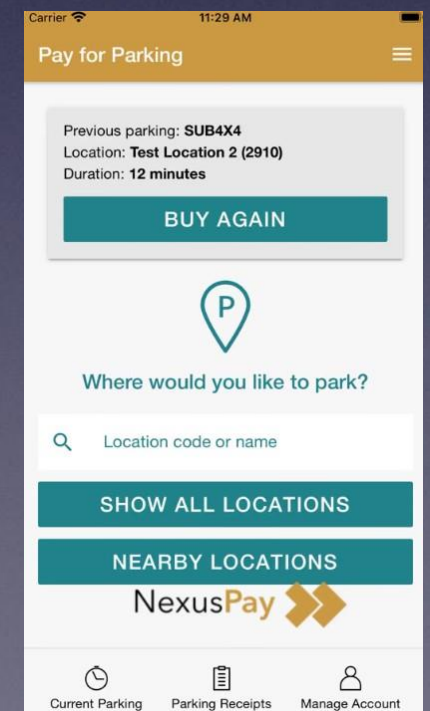
QUEUE.IT

And if you can't be available...

What does a web system need?

- Availability
- Performance
- Reliability
- Scalability
- Manageability
- Cost

Services must be fast.
No lag.
No waiting.



What does a web system need?

- Availability

- Performance

- Reliability

Services must be reliable.
Same inputs -> same outputs.
Same inputs -> same display.

- Scalability

- Manageability

- Cost

What does a web system need?

- Availability
- Performance
- Reliability
- Scalability
- Manageability
- Cost

Services must be easily upgradable to make space for new functionality

“More users” and/or “more data” should not be a problem.

What does a web system need?

- Availability
- Performance
- Reliability
- Scalability
- Manageability
- Cost

Services must be easily managed.

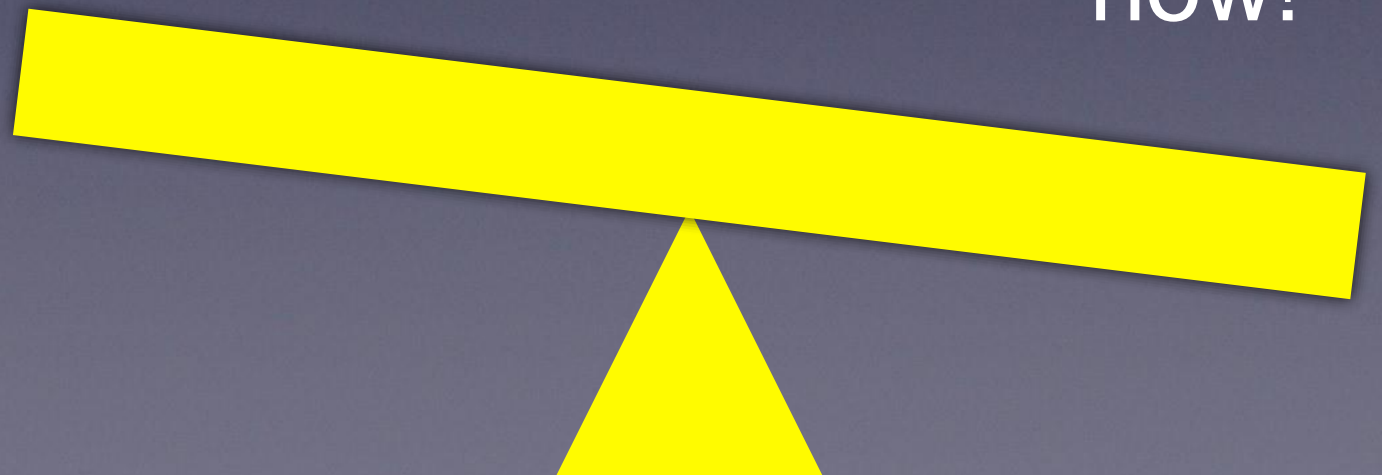
What does a web system need?

- Availability
- Performance
- Reliability
- Scalability
- Manageability
- Cost

Cost efficiency:
Balance run-time and
development costs.

Make it work
quickly, please

Make it work
now!



How do we actually specify a web system?

- What functionality will the browser be responsible for?
- What functionality will the back-end be responsible for?
- What data do we need to store?

More questions?

- What is our front end likely to run on? Desktop? Mobile App?
- What stack will be used? What specifications (RAM, CPU) will be needed for the server? Can we split server functionality over more than one machine?
- How big will our database get? Does our database need to be as fast as our server?

Side exercise...

When you have a minute...

- Compare AWS Elastic Compute Cloud (EC2) with their block storage options (<https://calculator.aws>)
- Or look at DigitalOcean's droplets (<https://www.digitalocean.com/products/droplets/>)
- There are different cloud products for different things.

Attendance

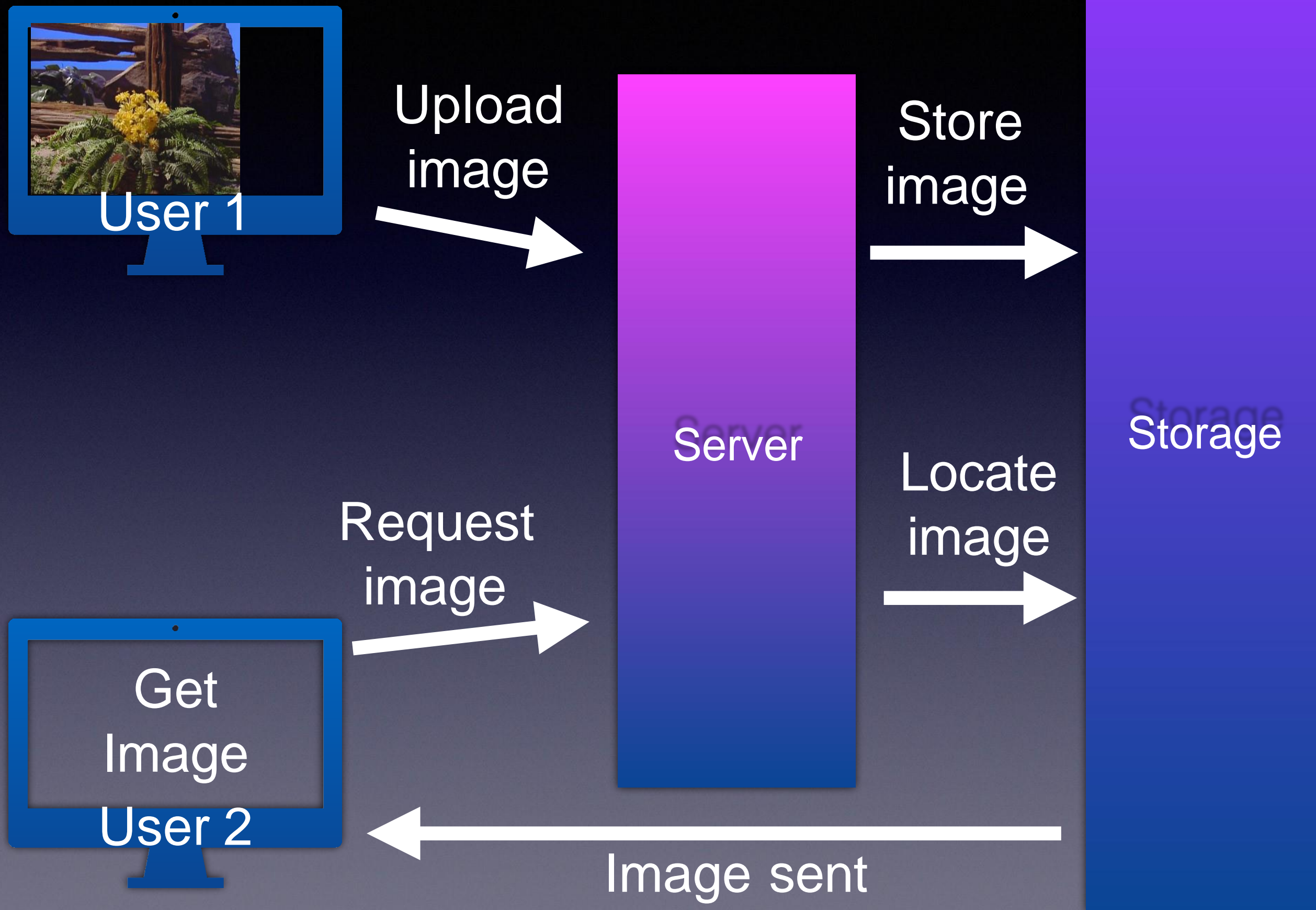
Yay! Attendr!

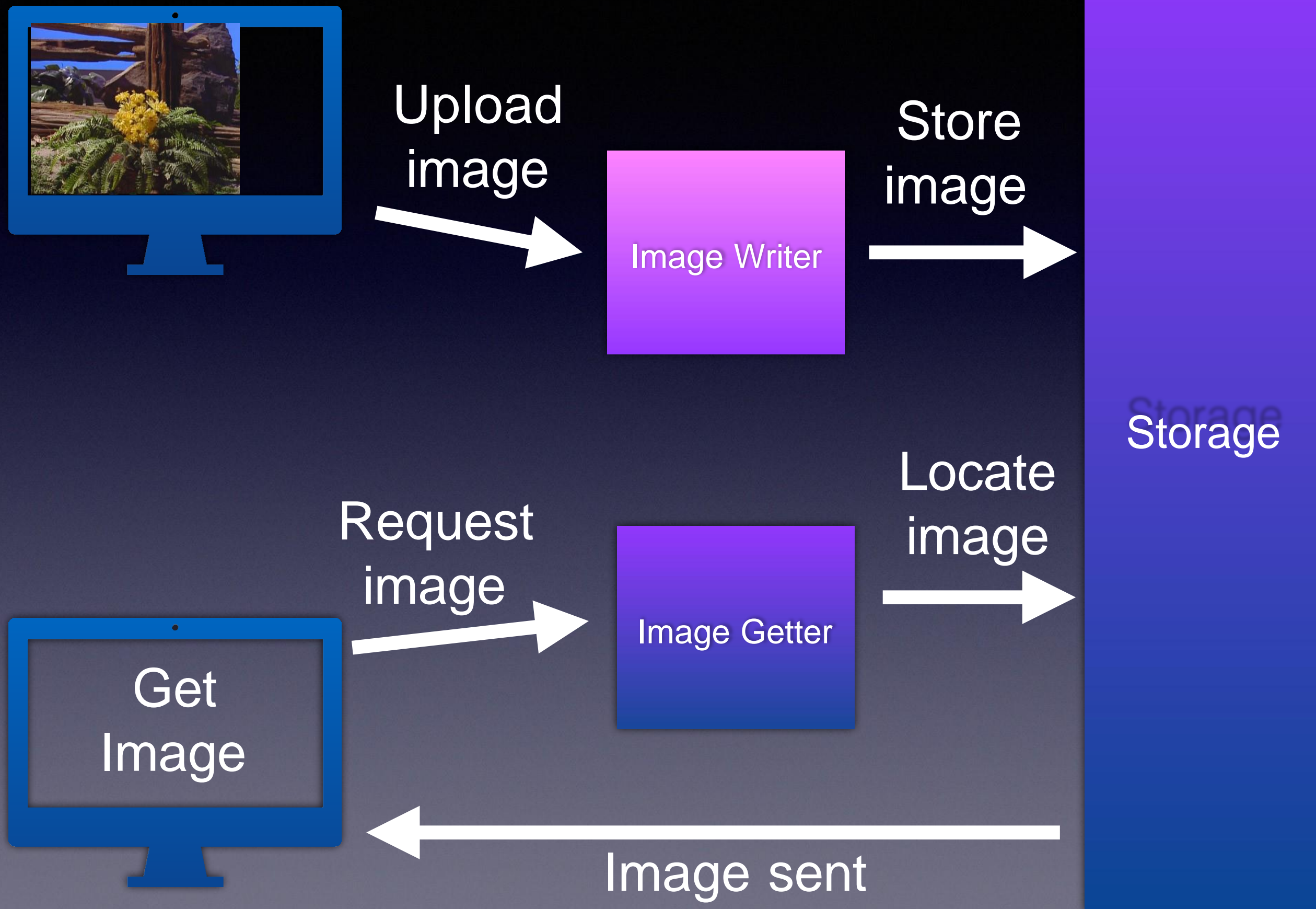


Reminder for Pam - it's probably running already...

Worked Example

- Imagine a web service that handles data
- Users can upload data
- Users can download data
- What does this involve?







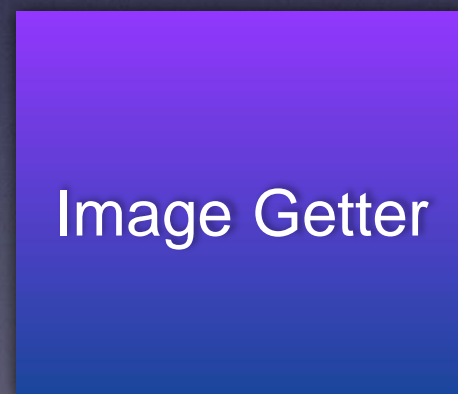
Upload
image



Store
image



Request
image



Locate
image

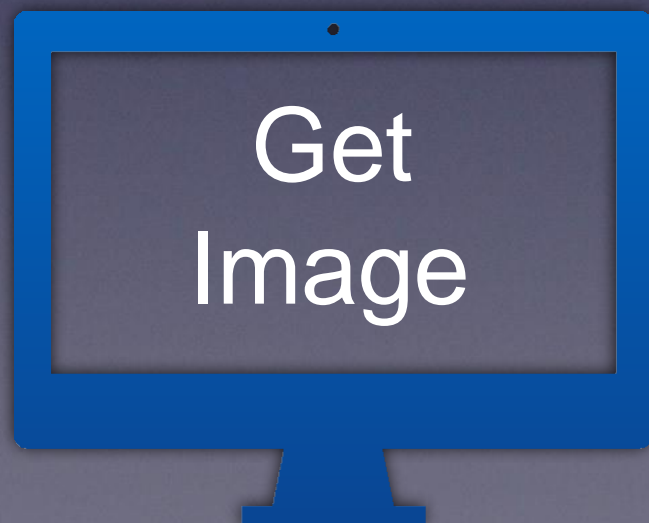


Image sent



Upload
image

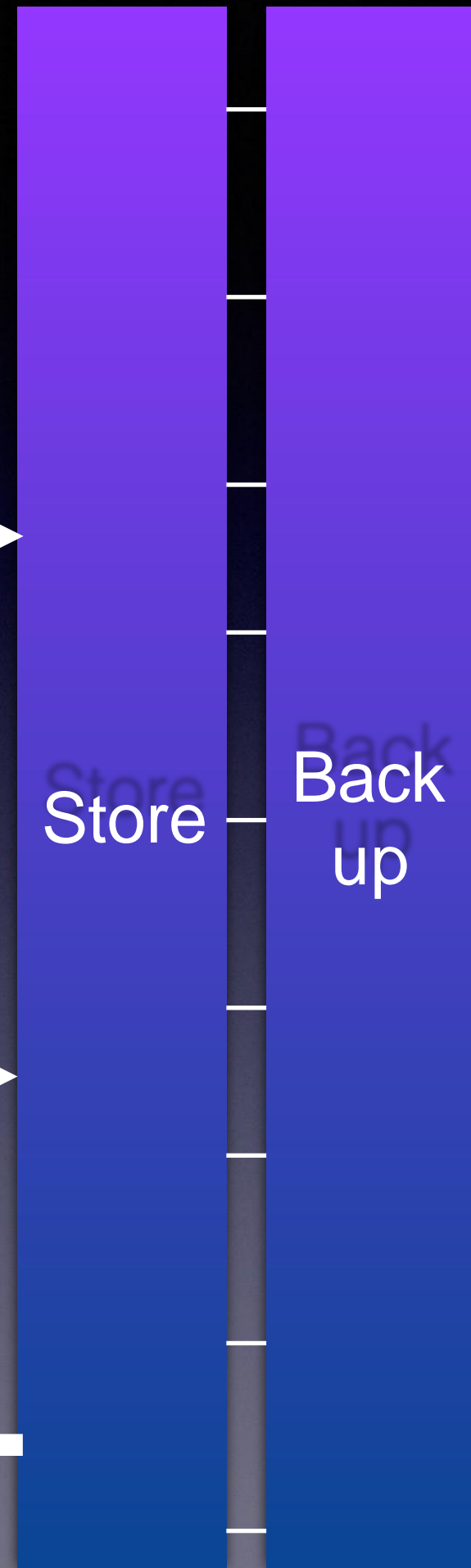


Store
image

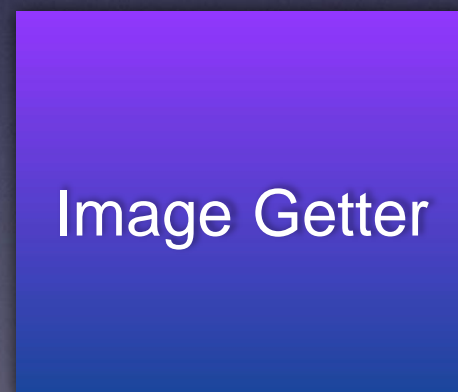


Store

Back
up



Locate
image



Request
image

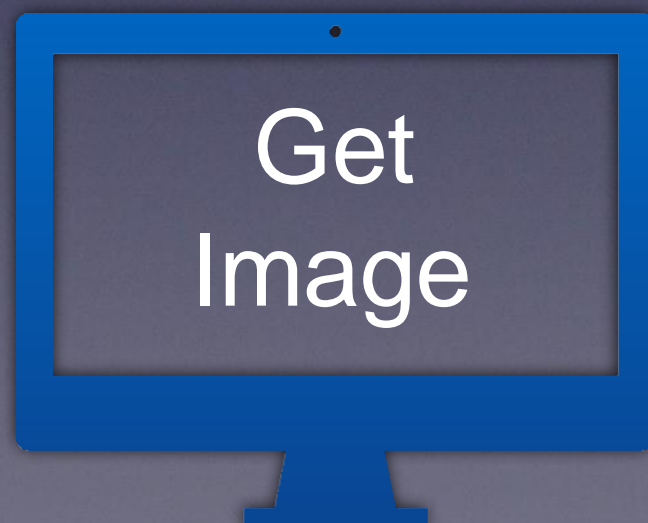


Image sent

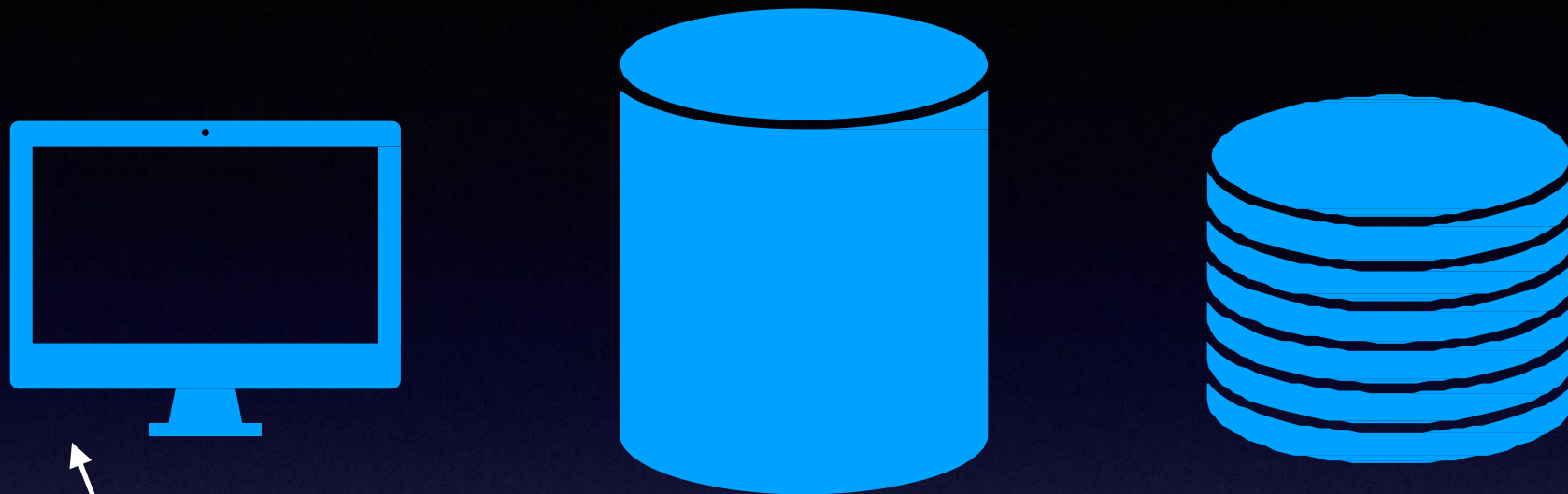


Scaling

Vertical scaling adds more storage/processing power to the same server.

Horizontal scaling adds more nodes or shards.

Vertical Scaling



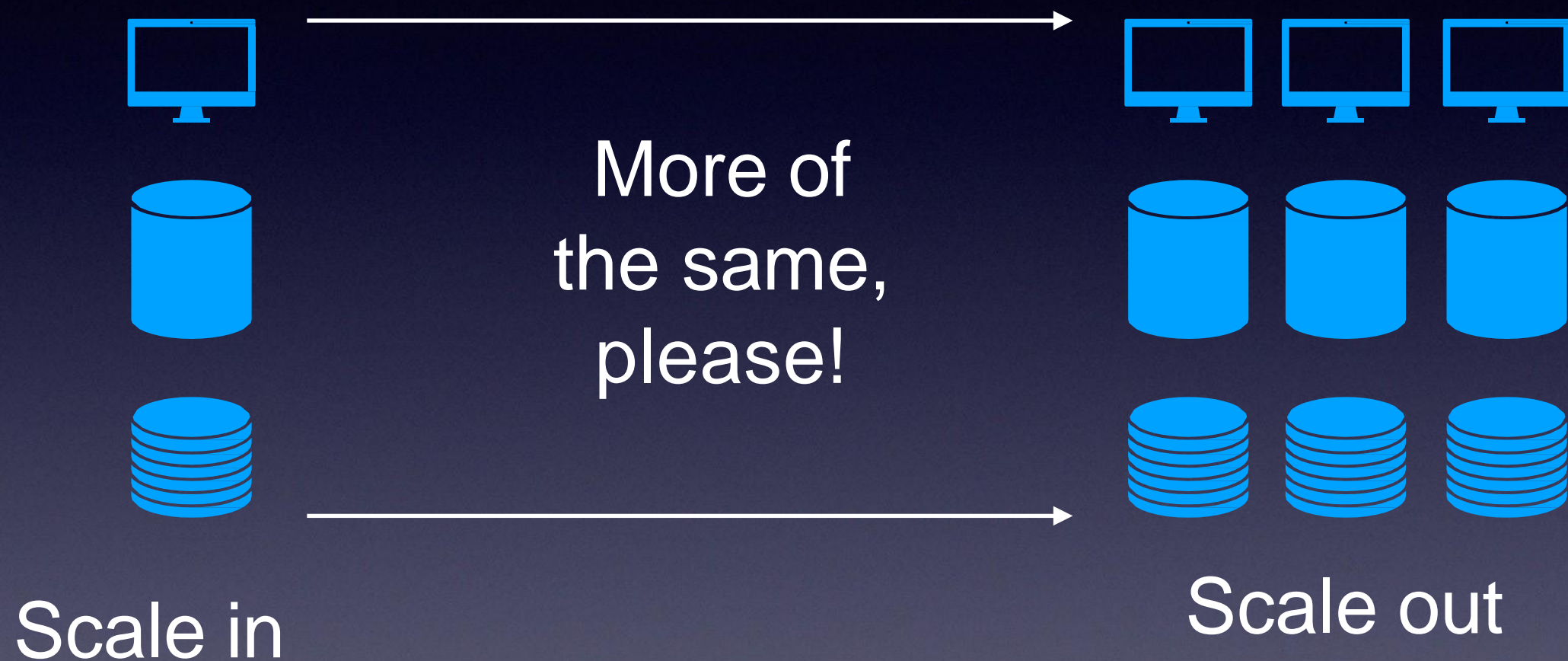
Scale up

More power!
More CPU!
More RAM!
More disk space!
Faster network!



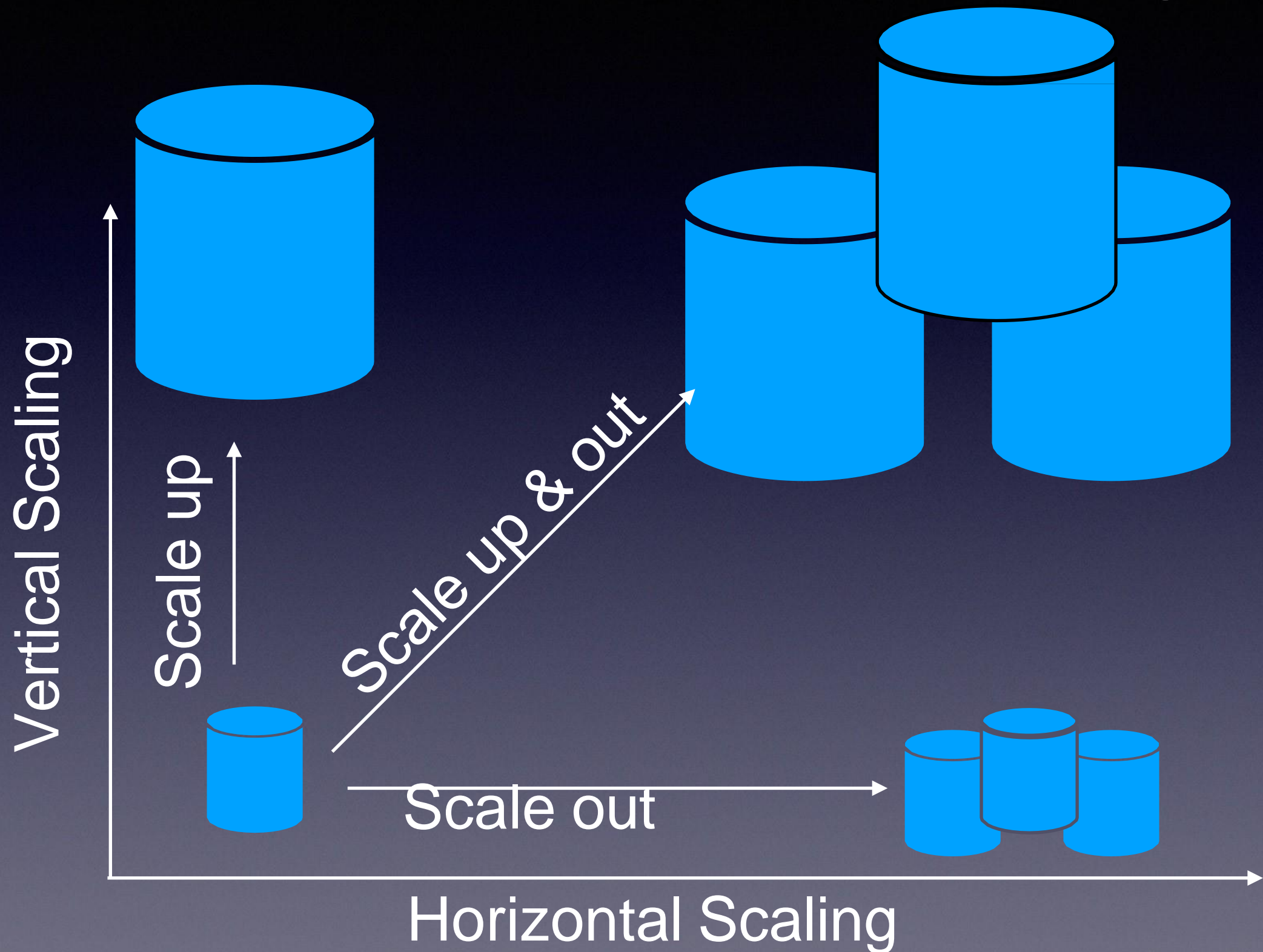
Scale down

Horizontal Scaling



Scale smarter, not harder!

Combination Scaling



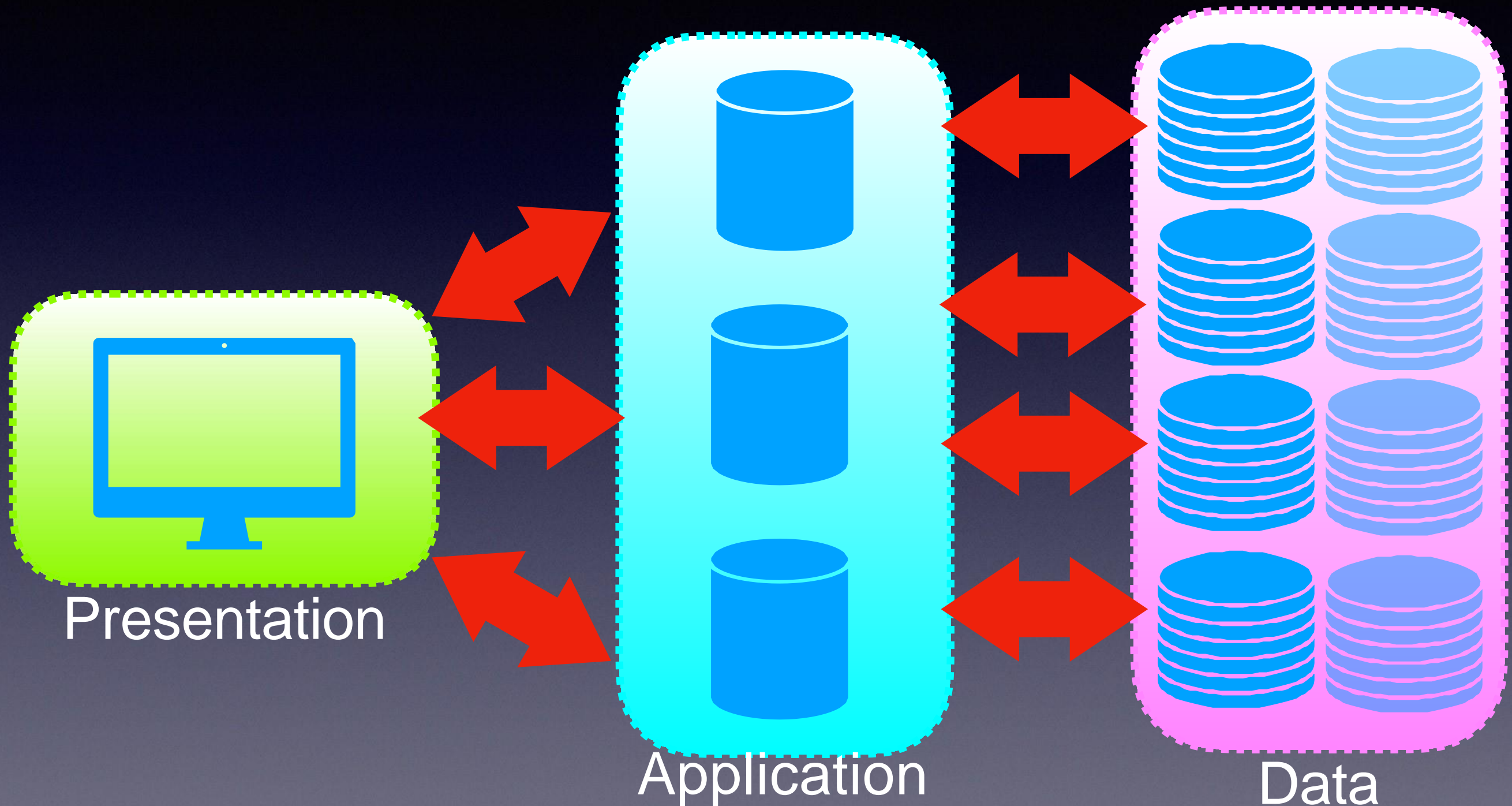
- You must separate key web services into different “logic nodes”
- Make sure that there is backup of all data for full **redundancy**
- Split everything into different **partitions**/shards to deal with increasing data usage
- Accept that you probably **want** to be slightly over capacity at all times

Go from this:



3-tier architecture

To this



n-tier architecture

How do we scale?

- Caching
- Proxies
- Load balancing
- Queuing

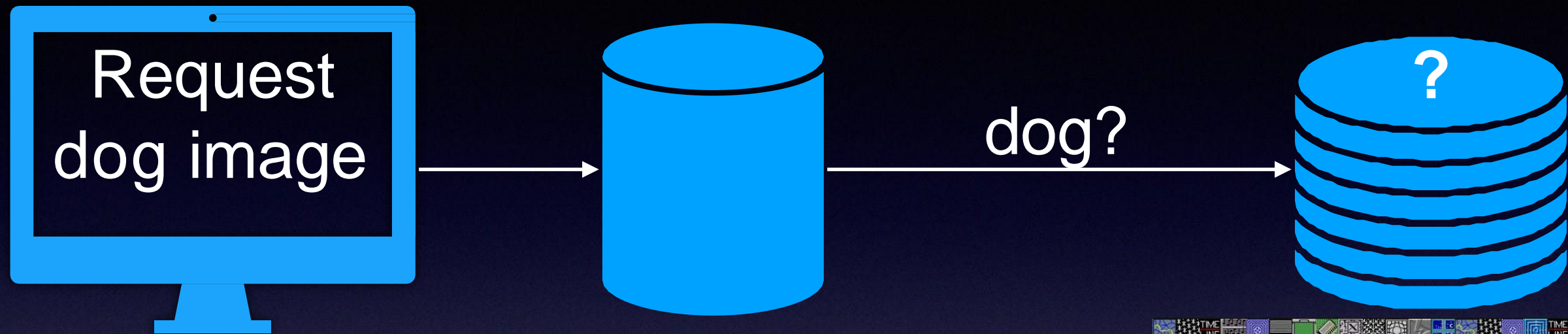
Before we go on...

- Think back to Labs 1 and 2
- You got the same website up and running on 2 different computers
- You could have run different services on different computers and had one machine selecting which computer to send things to
- That's a **reverse proxy**

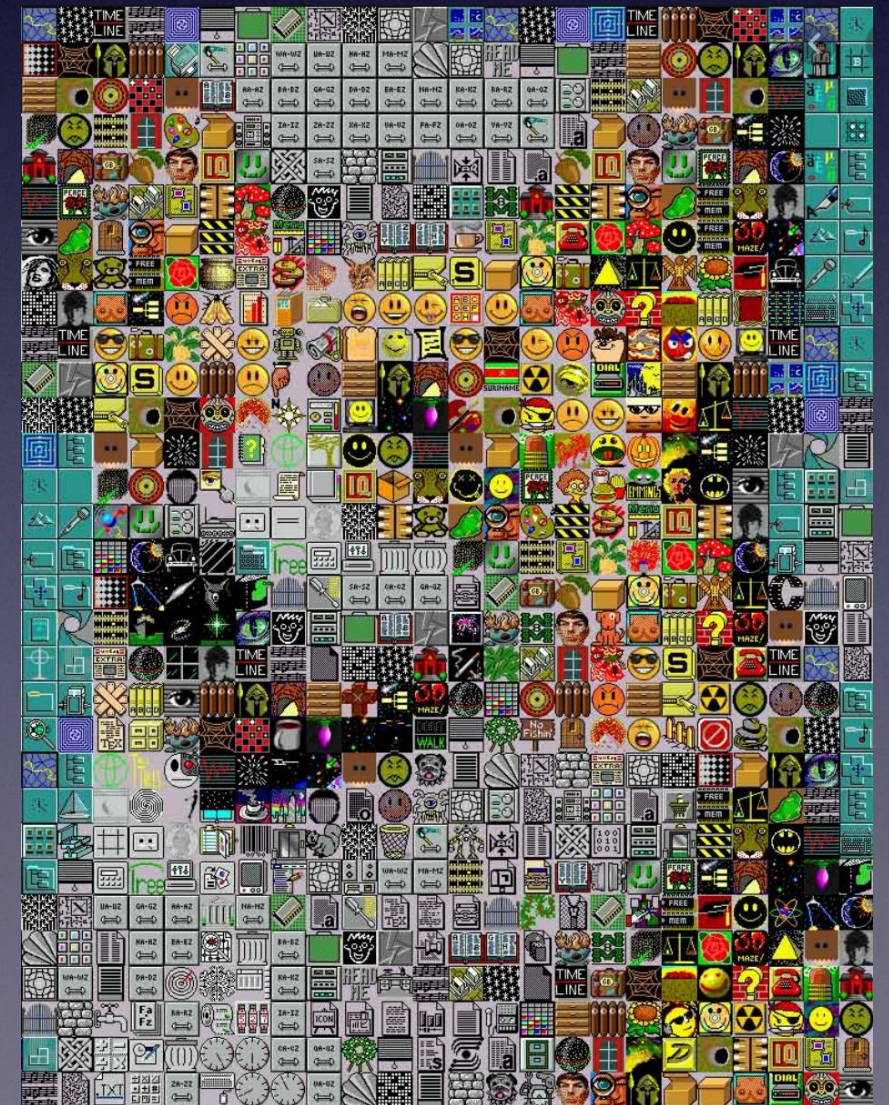
Before we go on...

- Think back to Lab 1 and how you used express for routing
- You simply responded to different urls in different ways
- It is possible to configure express to act as a basic load balancer (but express documentation implies you should use nginx instead)

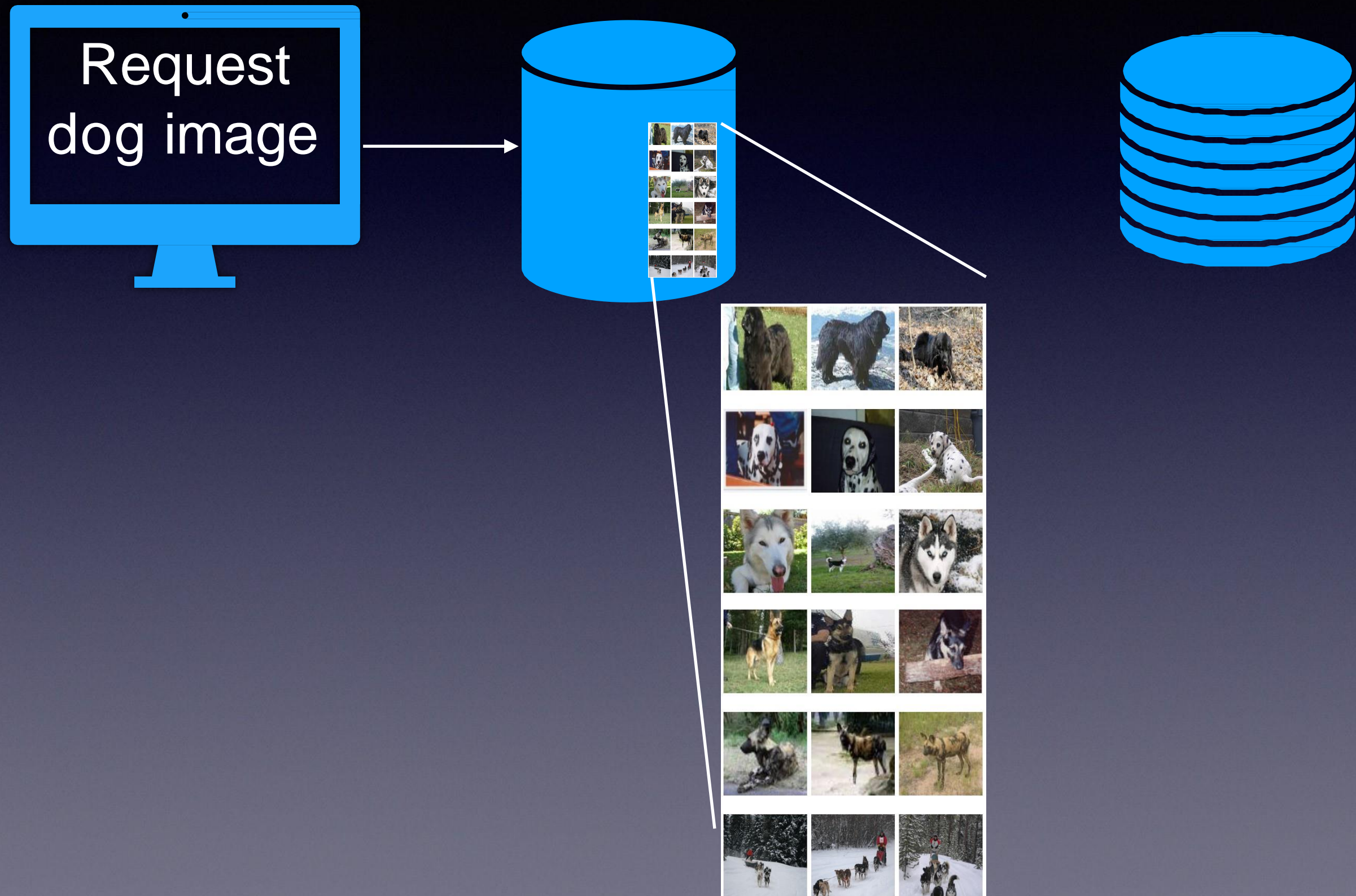
Caches



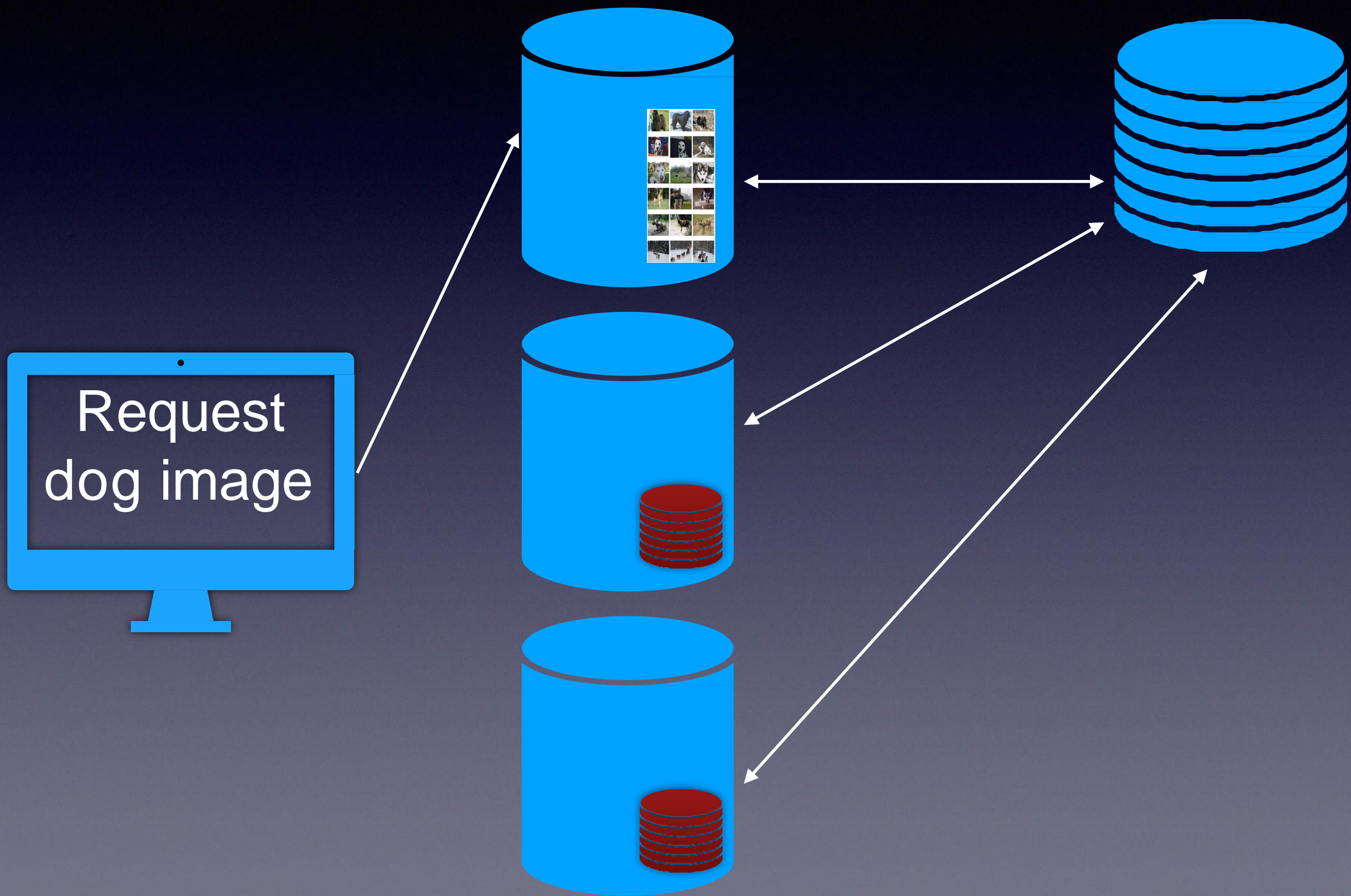
Too many images to
search through
efficiently!



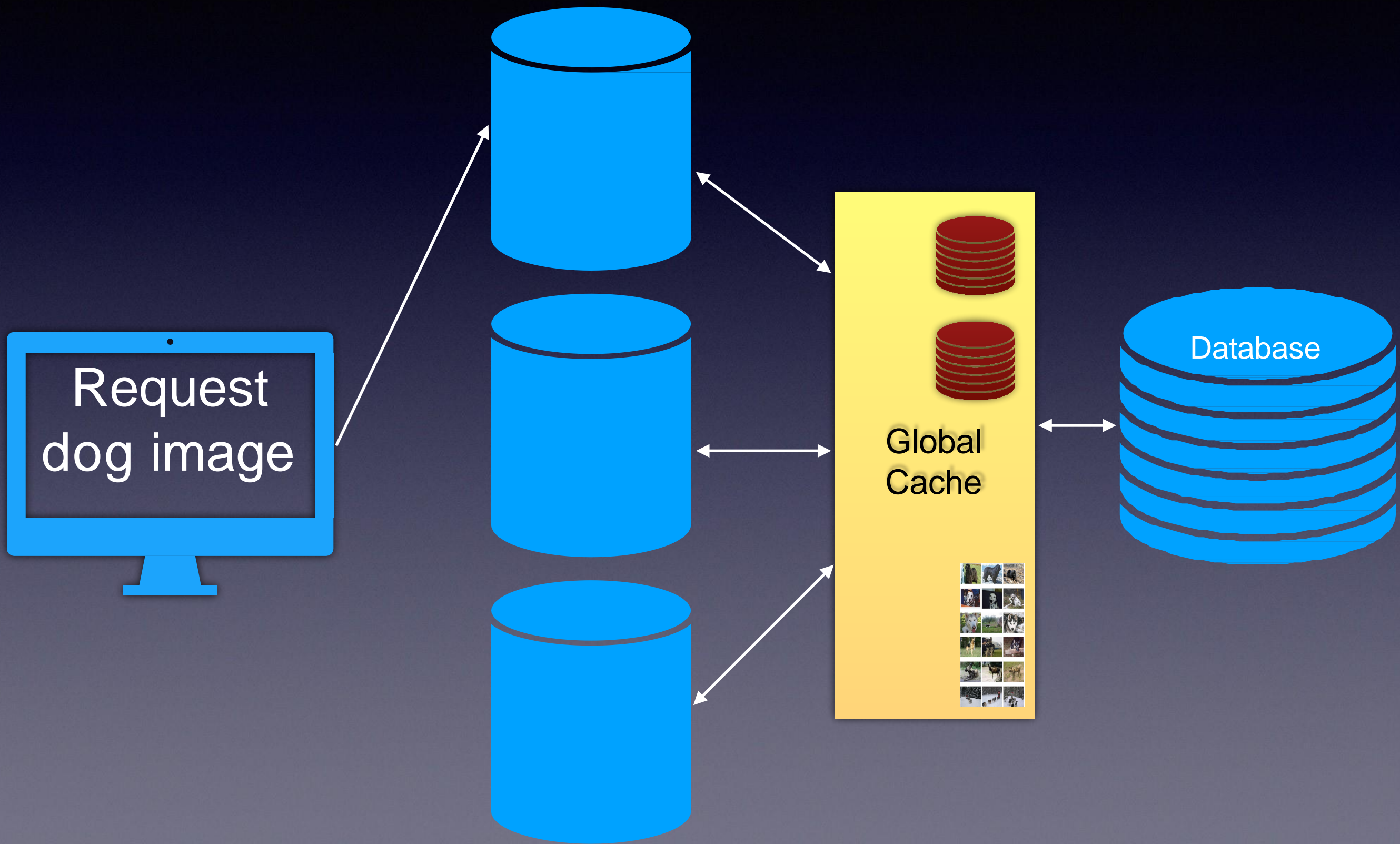
Caches



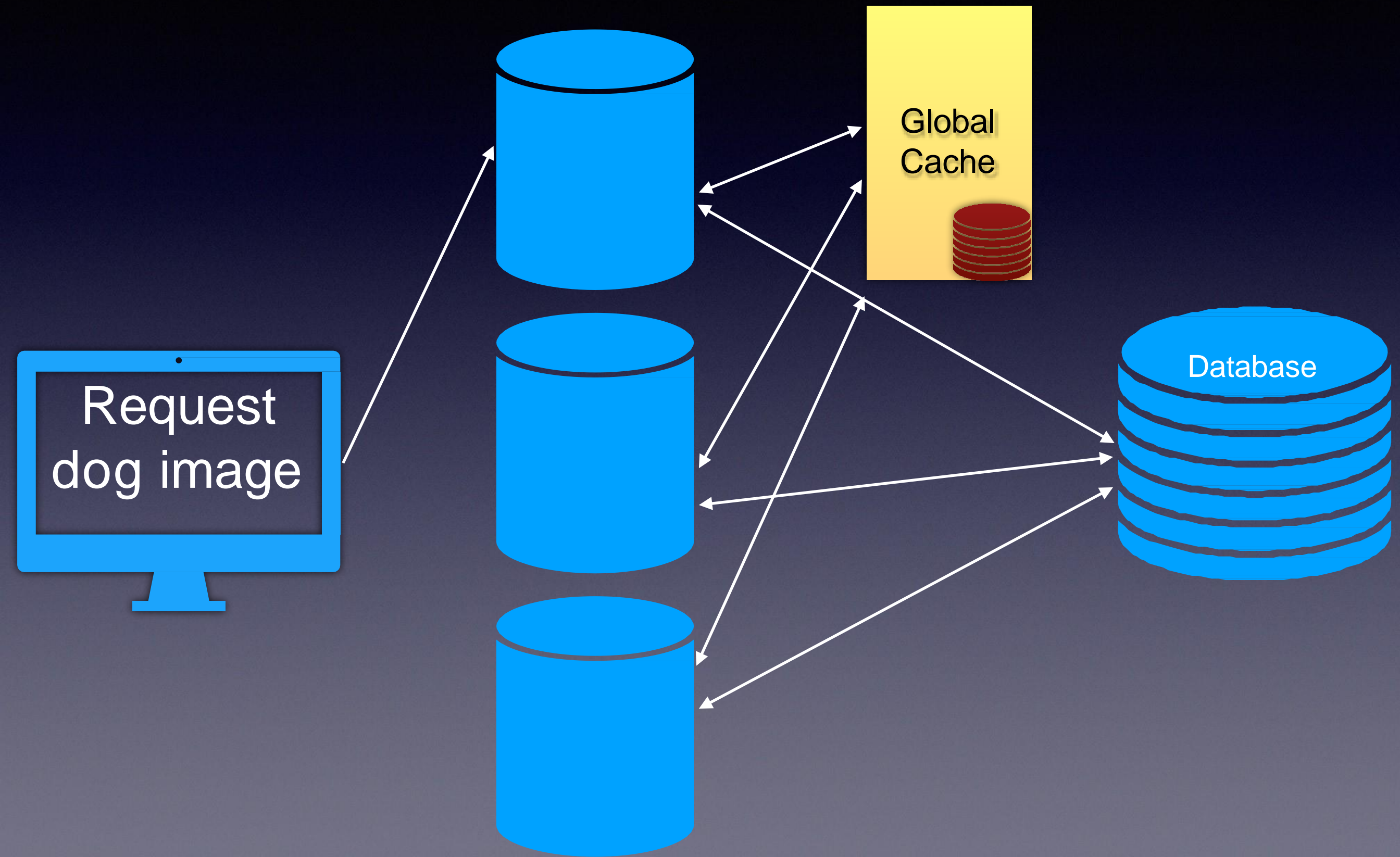
Distributed Cache System



Global Cache System (1)



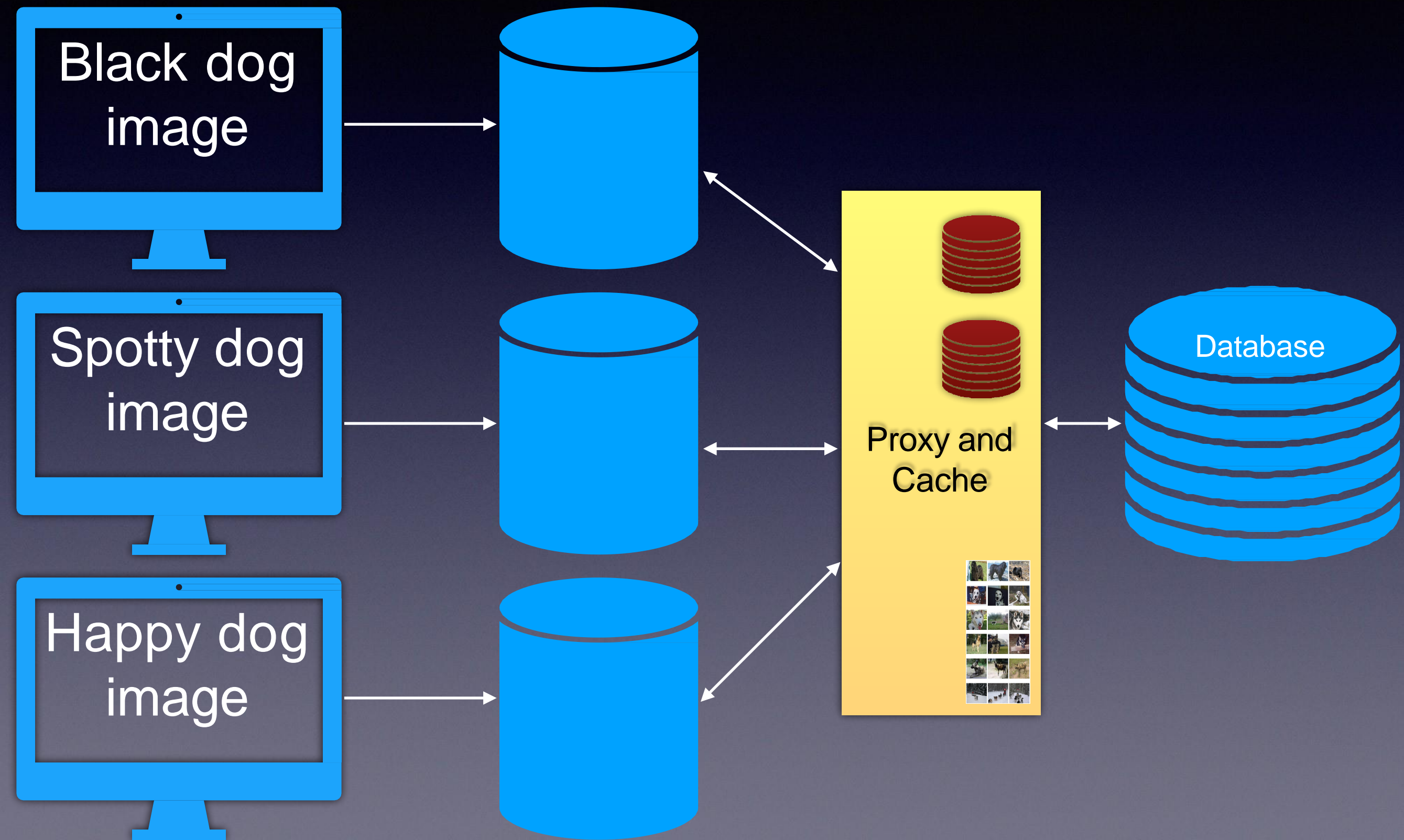
Global Cache System (2)



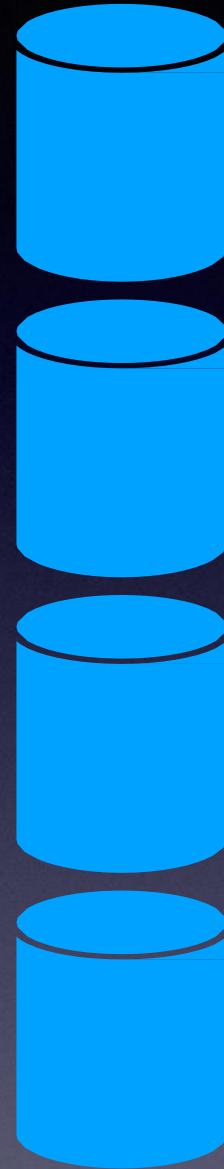
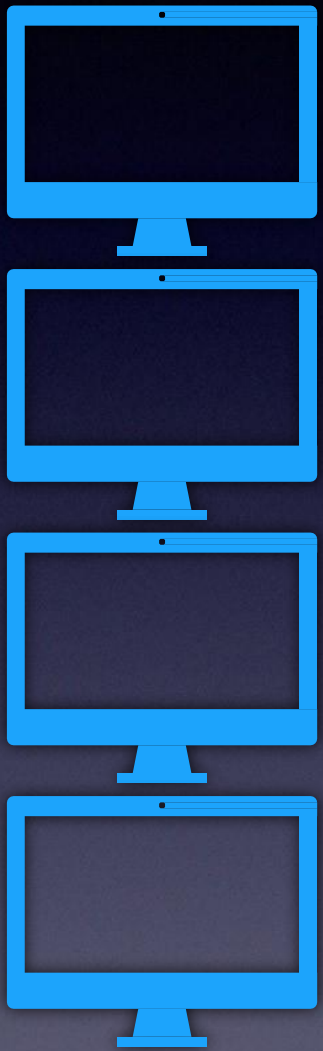
Proxies

- Proxies are good when people are requesting same thing.
- Also good when people are requesting *close* to the same thing.
- They collapse many application server requests into a single database request.
- Some proxies come with a cache built in to them.

Global Cache System (1)

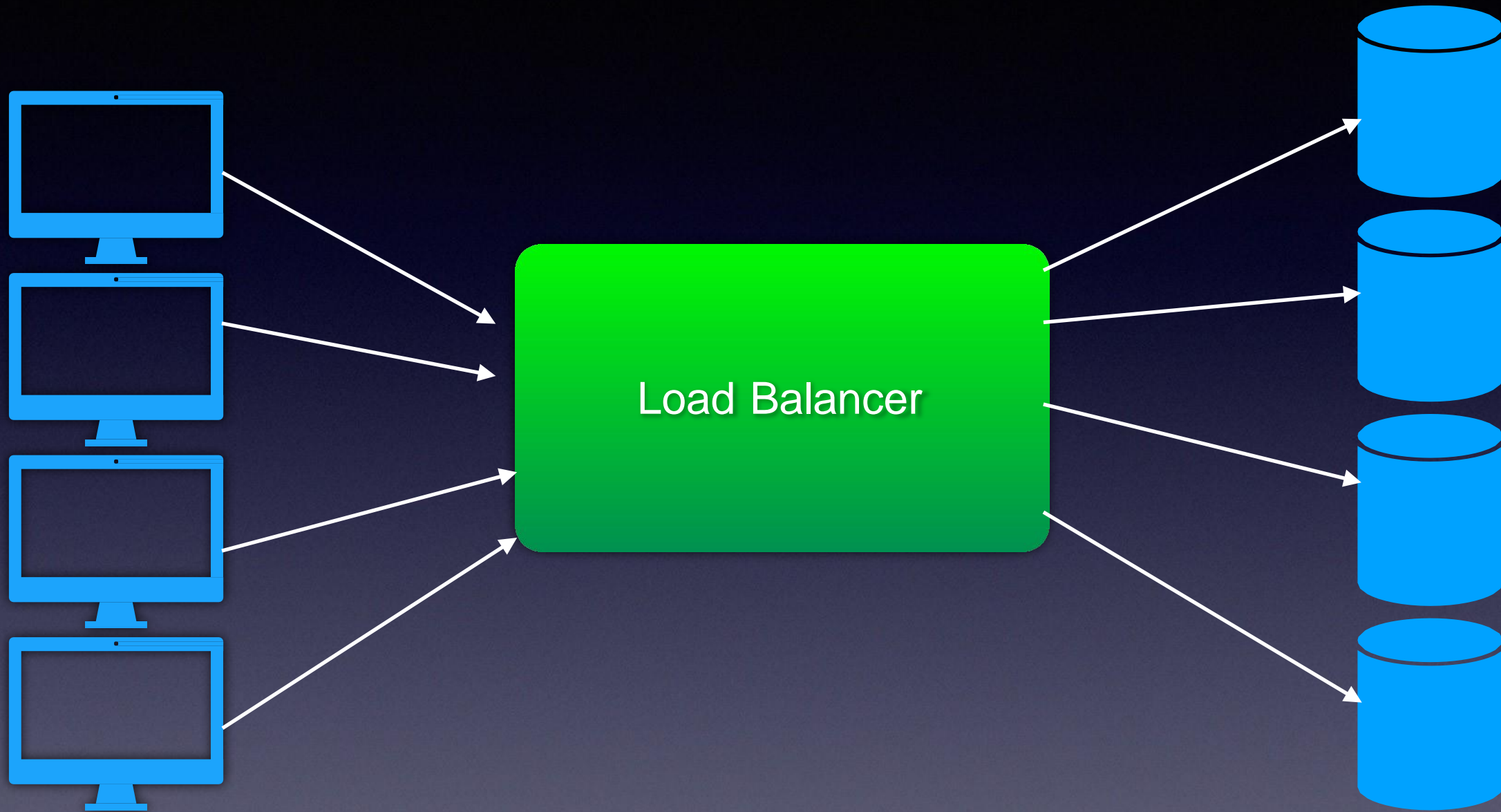


Load Balancing



Lots of clients, lots of application servers...

Load Balancing



The load balancer!

Load balancing methods

- Random node (just random)
- Round robin (just the next in line)
- Criteria based (the least busy server, specific access, IP Hash)
- Location based (the node in Ireland, the node in US East)

You can find implementation details with [nginx docs](#)

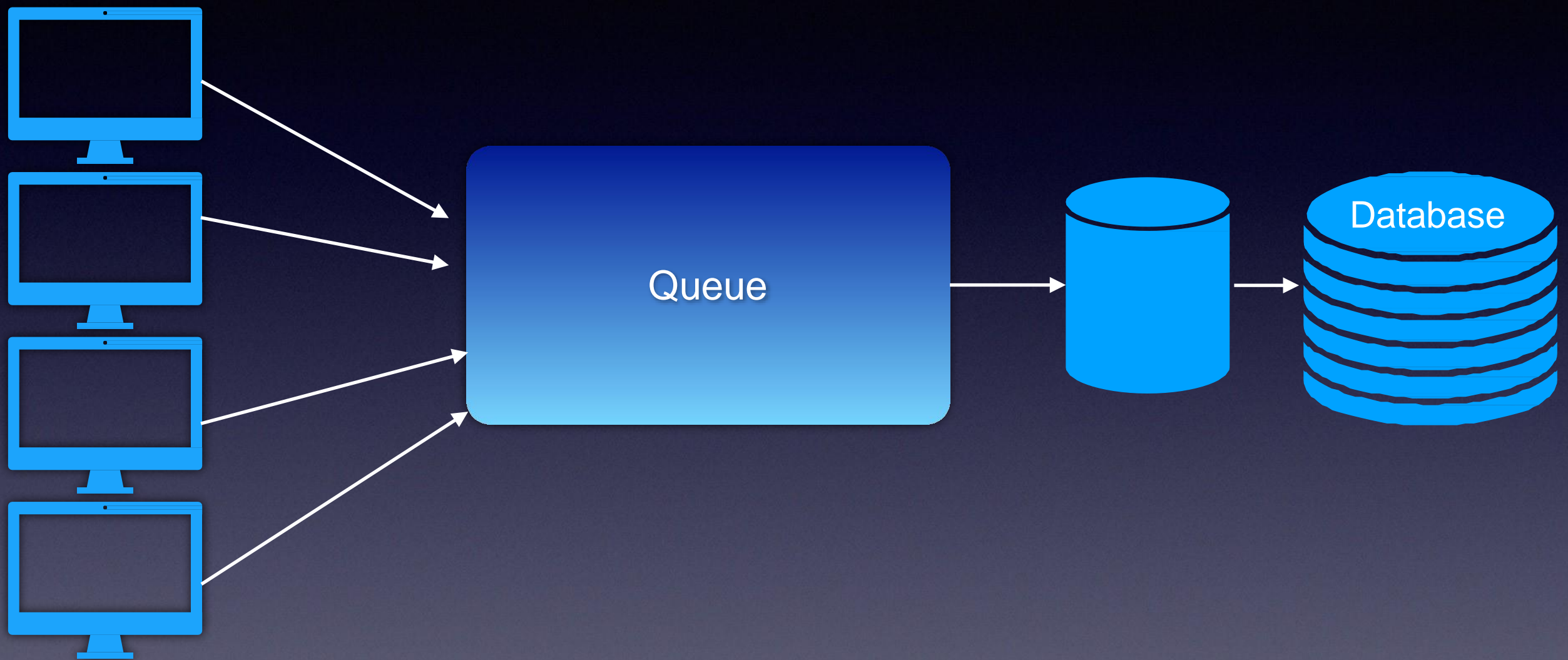
Problems with load balancing

- Connectionless sessions - you (probably) don't want one user to use multiple different servers during the same session (but IP hashing or other criteria might overcome this)
- If you change server, some server-based data (e.g. shopping baskets) might be deleted

Queues

- Caches, proxies and load balancers are good for reading data
- Queues are good for writing data
- Writing to a database can take a while. The user shouldn't be waiting for an acknowledgement.

Queues



Everyone who writes to the queue gets instant acks.

The database writing itself comes later when there isn't a user waiting on a response.

This Week's Lab

- It's a React front end (using next.js for backend)
- Hooks all the way
- localhost vs 127.0.0.1 and CORS
- There are interim solutions on Moodle