NITTE EDUCATION TRUST | NITTE MEENAKSHI INSTITUTE OF TECHNOLOGY
(An Autonomous Institution | Accredited with 'A+' Grade by NAAC | NBA | Approved By UGC | AICTE | Govt. of Karnataka | Affiliated to VTU, Belagavi)
Yelahanka, Bangalore – 560 064
KNOWLEDGE • CHARACTER • UNITY
Department of Computer Science and Engineering

5th Semester , Program Elective

Advanced Web Programming-I

(18CSE536), 3 Credits

Dr. Anil Kumar

Assistant Professor

Dept. of CSE, NMIT

AY 2020-2021

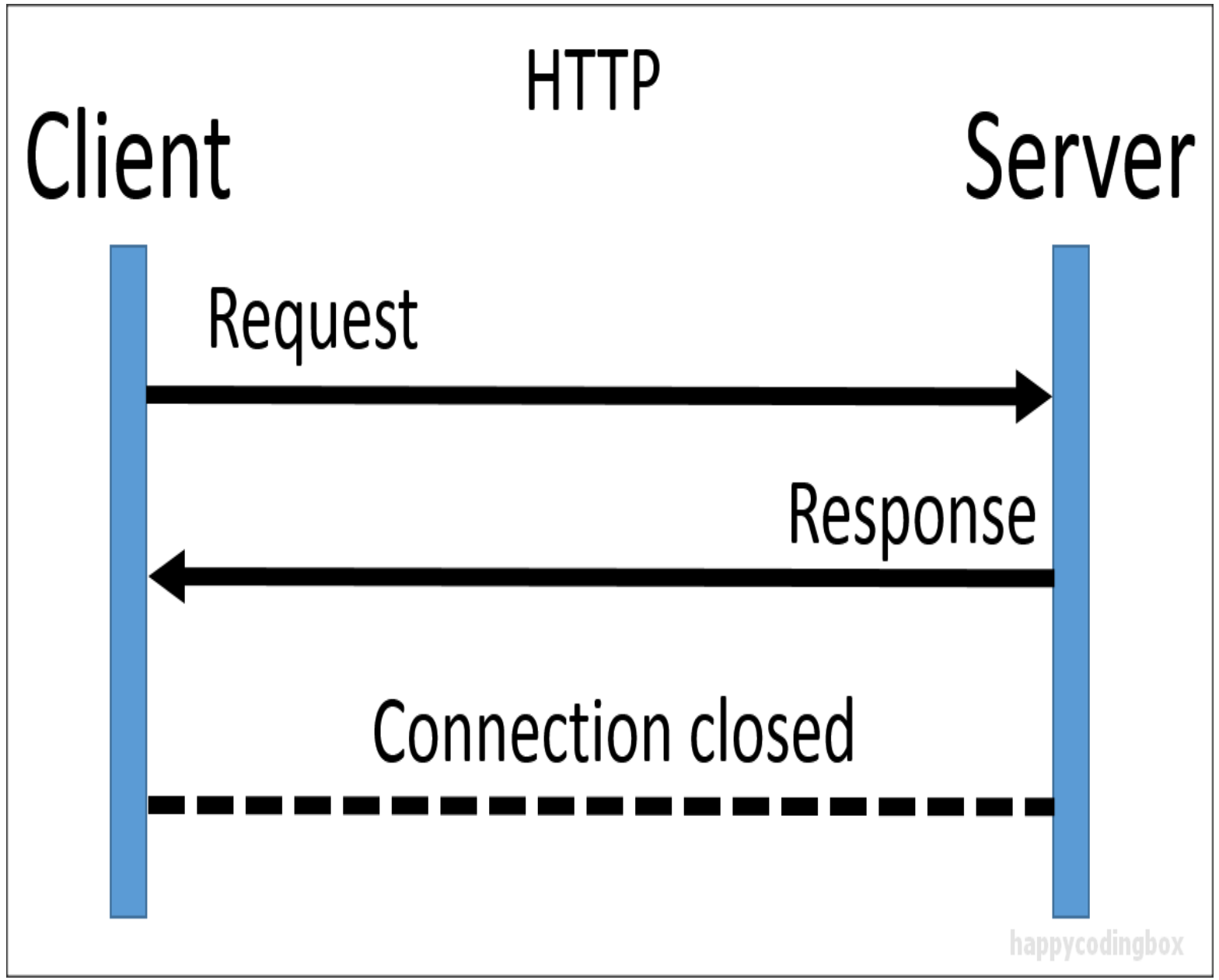
Unit-2

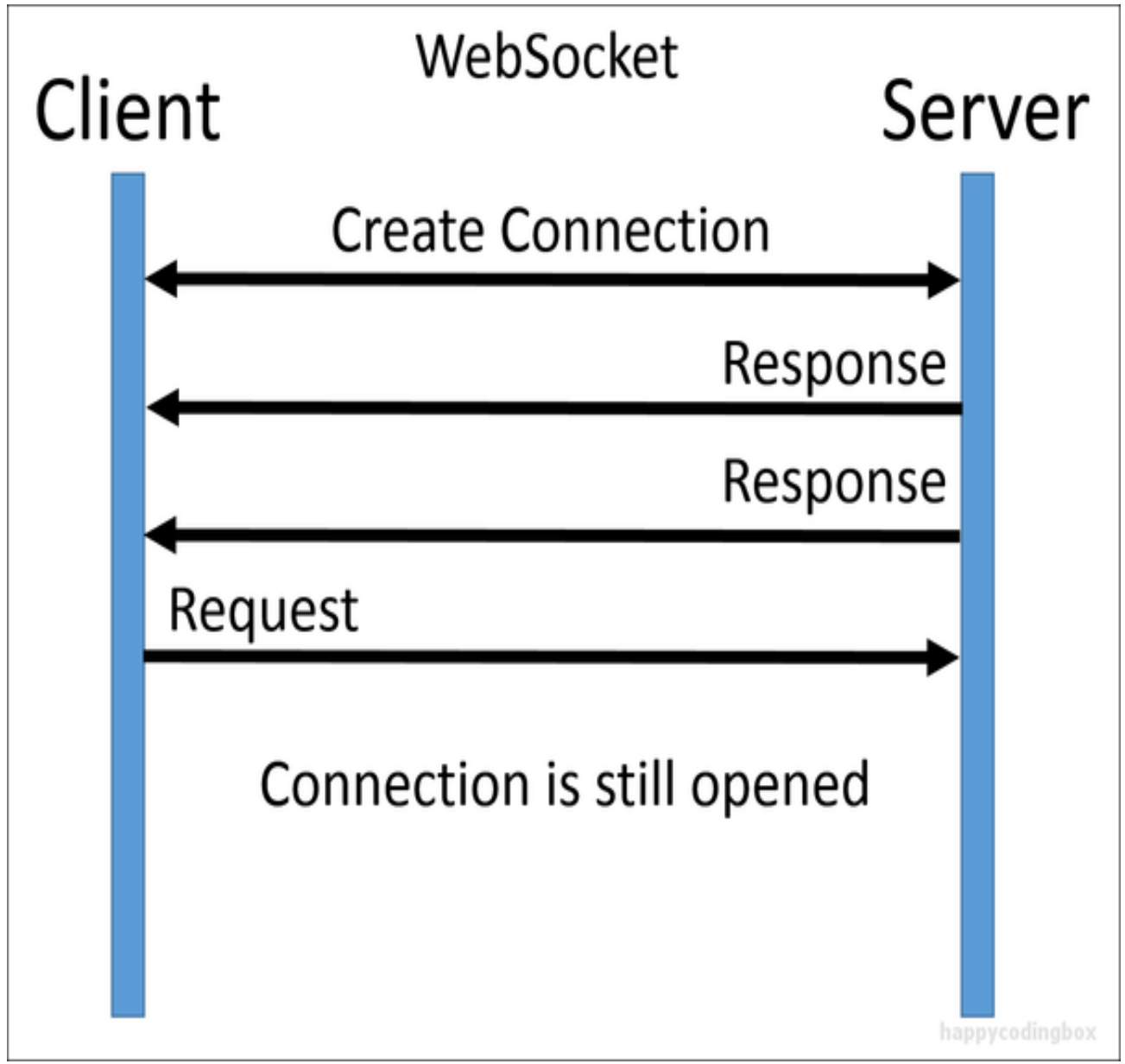
Node.js with MongoDB

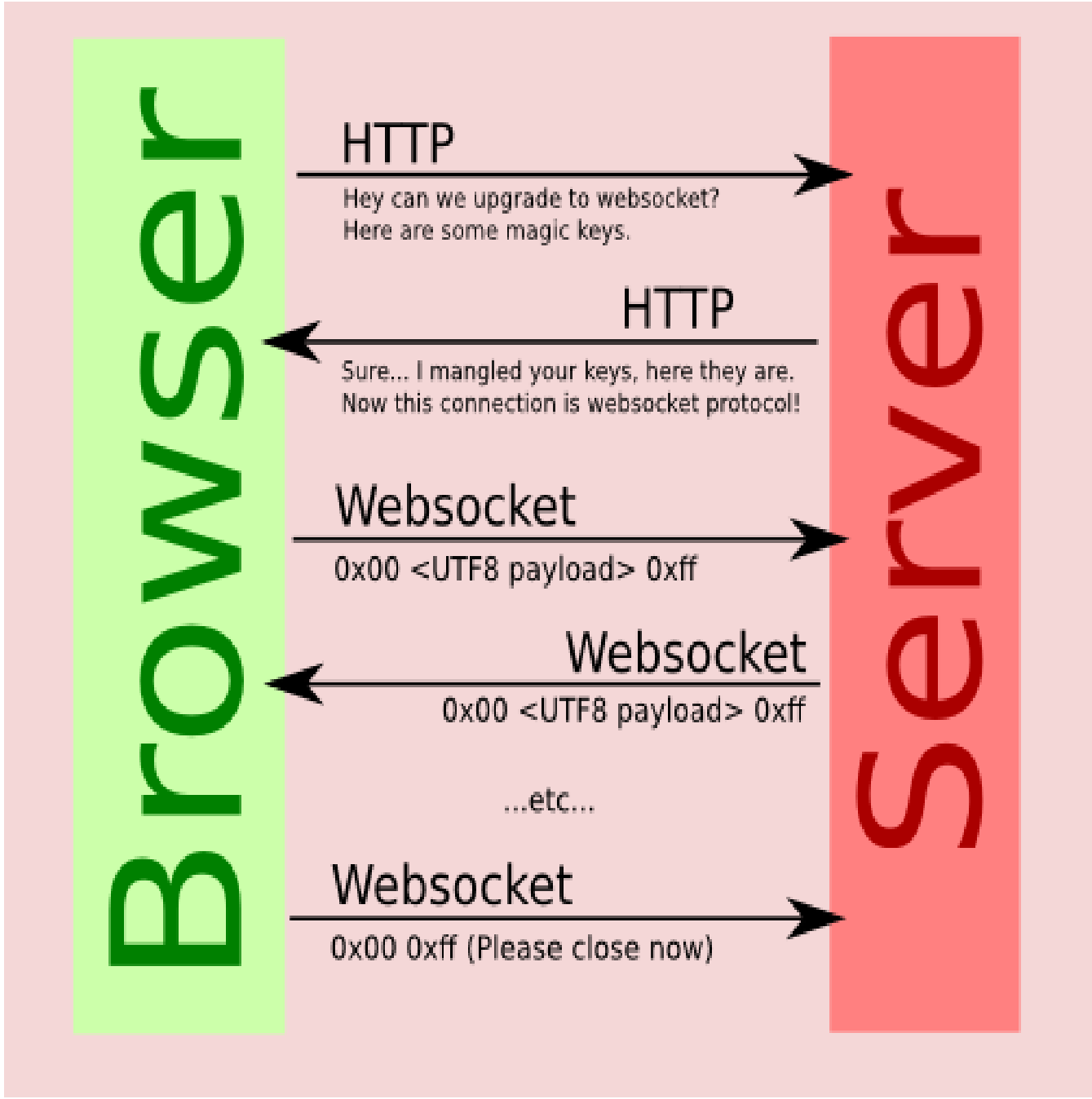
Web Sockets

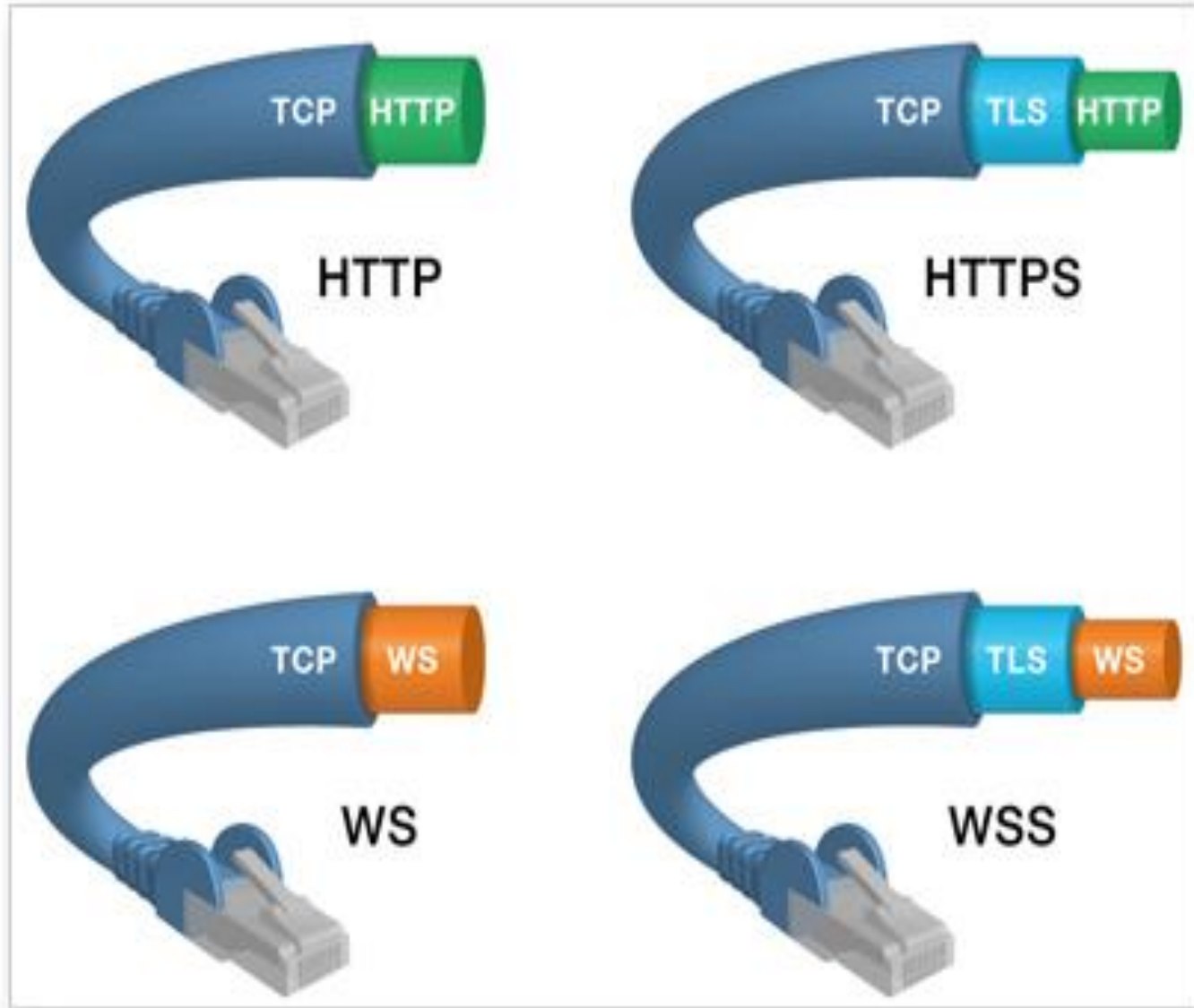
- Protocol providing full-duplex communication channel over single TCP connection.
- Web was built around the idea – 'Client requests, Server fulfills‘
- Other strategies like long-polling had the problem of carry overhead, leading to increase in latency.
- Easy to build real-time applications:
  - Chat Notifications
  - Online games
  - Financial trading
  - Data visualization dashboards
  - Live maps

HTTP	WebSocket
Duplex	
Half	Full
Messaging Pattern	
Request-reponse	Bi-directional
Service Push	
Not natively supported. Client polling or streaming download techniques used.	Core feature
Overhead	
Moderate overhead per request/connection.	Moderate overhead to establish & maintain the connection, then minimal overhead per message.
Intermediary/Edge Caching	
Core feature	Not possible
Supported Clients	
Broad support	Modern languages & clients


HTTP
Client
Server
Request
Response
Connection closed
happycodingbox


WebSocket
Client
Server
Create Connection
Response
Response
Request
Connection is still opened
happycodingbox


Browser
Server
HTTP
Hey can we upgrade to websocket?
Here are some magic keys.
HTTP
Sure... I mangled your keys, here they are.
Now this connection is websocket protocol!
Websocket
0x00 <UTF8 payload> 0xff
Websocket
0x00 <UTF8 payload> 0xff
...etc...
Websocket
0x00 0xff (Please close now)


TCP
HTTP
HTTP
TCP
TLS
HTTP
HTTPS
TCP
WS
WS
TCP
TLS
WS
WSS

Socket.io

- 2 JavaScript libraries
  - socket.io-client (front-end)
  - socket.io (back-end using NodeJS)

- Cross-browser compatibility by automatically using the best protocol for the user’s browser
  - WebSockets
  - Comet
  - Flash
- Socket.IO is quite popular, it is used by **Microsoft Office, Yammer, Zendesk, Trello**, and numerous other organizations to build robust real-time systems.

- To get started with developing using the **Socket.IO**, you need to have **Node** and **npm (node package manager)** installed.
- Open your terminal
  - $ mkdir test-project
  - $ cd test-project
  - $ npm init
- This will create a **‘package.json node.js’** configuration file.
- Now we need to install **Express** and **Socket.IO**
  - $ npm install –save express socket.io
  - $ npm install –g nodemon

- Whenever you need to start the server, instead of using the **node app.js** use, **nodemon app.js**. This will ensure that you do not need to restart the server whenever you change a file. It speeds up the development process.

Socket.io Services

- Socket.io Event Handling
- Socket.io Broadcasting
- Socket.io Namespaces
- Socket.io Rooms
- Socket.io Error Handling
- Socket.io logging and debugging

Events in server side

- There are some reserved events, which can be accessed using the socket object on the server side.
  - Connect
  - Message
  - Disconnect
  - Reconnect
  - Ping
  - Join
  - Leave

Events on Client side

- Connect
- Connect_error
- Connect_timeout
- Reconnect, etc

```
// sending to sender-client only
socket.emit('message', "this is a test");

// sending to all clients, include sender
io.emit('message', "this is a test");

// sending to all clients except sender
socket.broadcast.emit('message', "this is a test");

// sending to all clients in 'game' room(channel) except sender
socket.broadcast.to('game').emit('message', 'nice game');

// sending to all clients in 'game' room(channel), include sender
io.in('game').emit('message', 'cool game');

// sending to sender client, only if they are in 'game' room(channel)
socket.to('game').emit('message', 'enjoy the game');

// sending to all clients in namespace 'myNamespace', include
senderio.of('myNamespace').emit('message', 'gg');
```

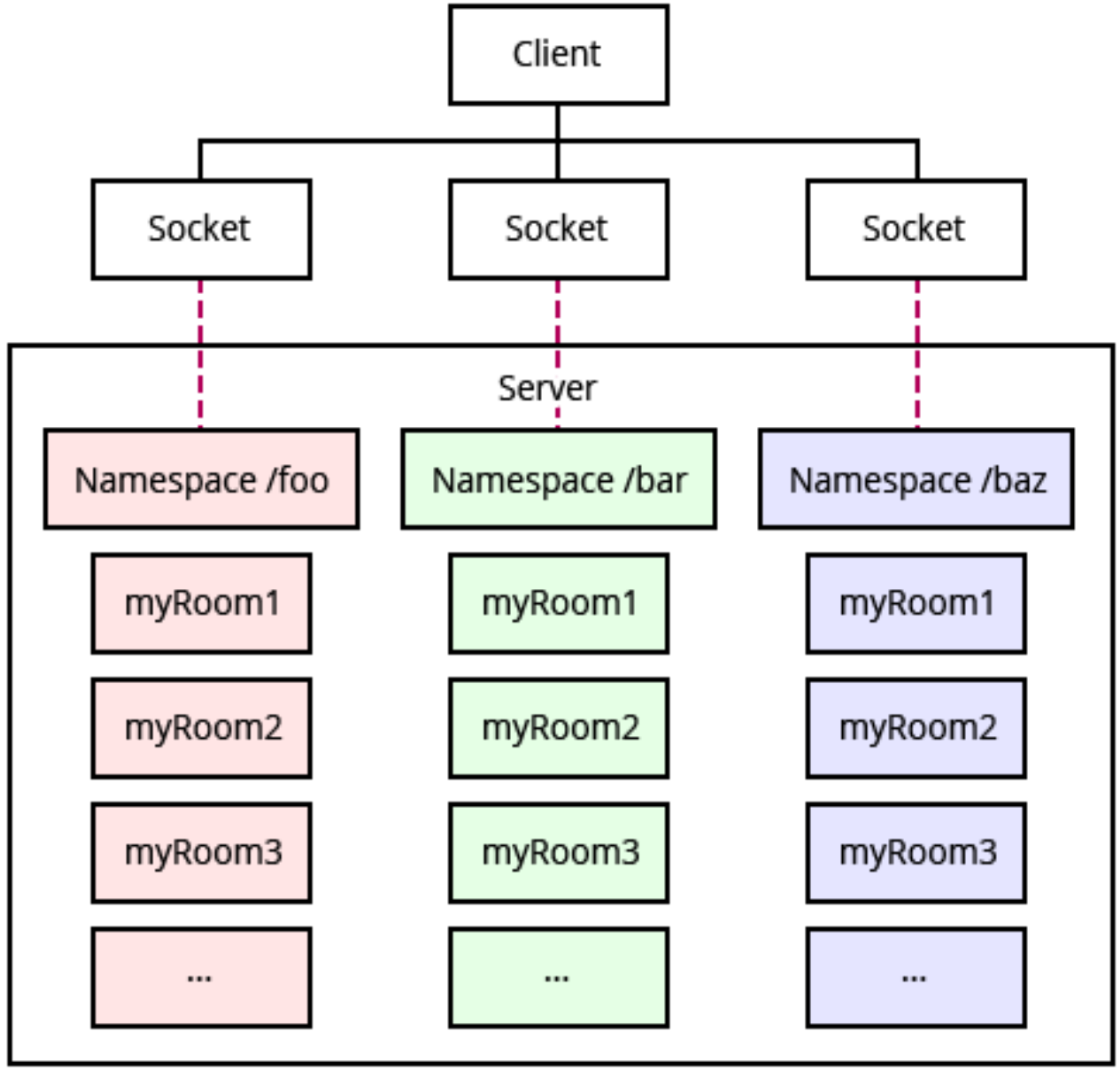
```
// sending to individual socketid (server-side)
socket.broadcast.to(socketid).emit('message', 'for your eyes
only');

// join to subscribe the socket to a given channel (server-side):
socket.join('some room');

// then simply use to or in (they are the same) when
broadcasting or emitting (server-side)
io.to('some room').emit('some event');

// leave to unsubscribe the socket to a given channel (server-
side)
socket.leave('some room');
```

Socket.IO - Broadcasting

- Broadcasting means sending a message to all connected clients.
- Broadcasting can be done at multiple levels. We can send the message to all the connected clients, to clients on a namespace and clients in a particular room.
- To broadcast an event to all the clients, we can use the **io.sockets.emit** method.
- To broadcast a message to a new client and update the other clients about him/her joining, that can be done by using **socket.broadcast.emit**.

Socket.io Namespaces

- Socket.IO allows you to “namespace” your sockets, which essentially means assigning different endpoints or paths.
- This is a useful feature to minimize the number of resources (TCP connections) and at the same time separate concerns within your application by introducing separation between communication channels.
- Multiple namespaces actually share the same WebSockets connection thus saving us socket ports on the server.
- Namespaces are created on the server side. However, they are joined by clients by sending a request to the server.
- The root namespace '/' is the default namespace, which is joined by clients if a namespace is not specified by the client while connecting to the server.
- Namespaces are a great way to make sure that our Socket.IO events are not emitted globally to all the sockets that are connected to the server.


Client
Socket
Socket
Socket
Server
Namespace /foo
Namespace /bar
Namespace /baz
myRoom1
myRoom2
myRoom3
...
myRoom1
myRoom2
myRoom3
...
myRoom1
myRoom2
myRoom3
...

Socket.io Rooms

- Rooms are part of a namespace where you can create as many rooms as your application needs to do pretty much everything concerning sockets and exchanging data (for ex: Private Chat).

- In order to allow a socket (client) to join a room it first has to be requested then the Server endpoint handler handles the join request and either allow or deny depending on your application whether the client needs an authentication or a specific password or whatever you rely on your code base.

Socket.IO - Error Handling

- Till Now we have worked on local servers until now, which will almost never give us errors related to connections, timeouts, etc.
- However, in real life production environments, handling such errors are of most important.
- We see, how we can handle connection errors on the client side.

  - **Connect** − When the client successfully connects.
  - **Connecting** − When the client is in the process of connecting.
  - **Disconnect** − When the client is disconnected.
  - **Connect_failed** − When the connection to the server fails.
  - **Error** − An error event is sent from the server.
  - **Message** − When the server sends a message using the **send** function.
  - **Reconnect** − When reconnection to the server is successful.
  - **Reconnecting** − When the client is in the process of connecting.
  - **Reconnect_failed** − When the reconnection attempt fails.

For example − If we have a connection that fails, we can use the following code to connect to the server again

```
socket.on('connect_failed', function() {
document.write("Sorry, there seems to be an issue with the connection!");
})
```

Socket.IO - Logging and Debugging

- Socket.IO uses a very famous debugging module developed by ExpresJS's main author, called **debug**.

- Server-side

```
DEBUG=* node app.js
```

- Client-side

```
localStorage.debug = '*';
```

Scaling in Node.js

- The main focus of the programmer will be about efficiency and performance, in order to obtain the best result with less resources.
- One way to improve the throughput of a web application is to scale it, instantiate it multiple times balancing the incoming connection between the multiple instances.
- When you scale up, you have to be careful about different aspects of your application, from the state to the authentication.

Horizontally scaling a Node.js application

- Horizontal scaling is about duplicating your application instance to manage a larger number of incoming connections.
- This action can be performed on a single multi-core machine or across different machines.

- Vertical scaling is about increasing the single machine performances, and it do not involve particular work on the code side.

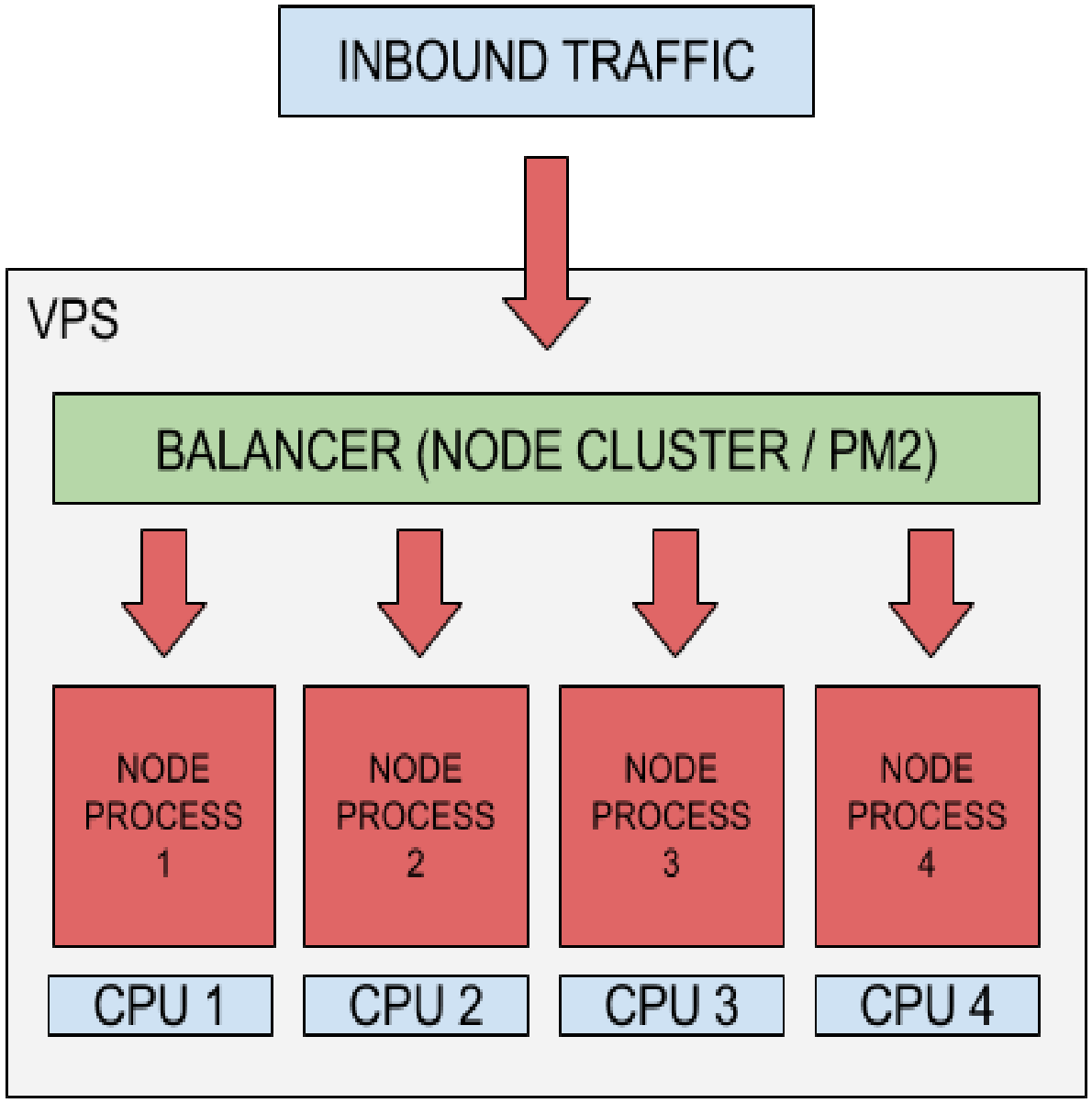
Multiple processes on same machine

- One common way to increase the throughput of your application is to spawn one process for each core of your machine.
- By this way the already efficient “concurrency” management of requests in Node.js (see “event driven, non-blocking I/O”) can be multiplied and parallelized.

Cluster mode in Node.js

- There are different strategies for scaling on a single machine, But the common concept is to have multiple processes running on the same port, With some sort of internal load balancing used to distribute the incoming connections across all the processes/cores.

- The strategies which we will be discussing are the standard Node.js **cluster mode** and the automatic, higher-level **PM2 cluster** functionality.


INBOUND TRAFFIC
VPS
BALANCER (NODE CLUSTER / PM2)
NODE PROCESS 1
NODE PROCESS 2
NODE PROCESS 3
NODE PROCESS 4
CPU 1
CPU 2
CPU 3
CPU 4

Native cluster mode

- The native Node.js cluster module is the basic way to scale a Node app on a single machine (https://Node.js.org/api/cluster.html).
- One instance of your process (called “master”) is the one responsible to spawn the other child processes (called “workers”), one for each core, that are the ones that runs your application.
- The incoming connections are distributed following a round-robin strategy across all the workers, that exposes the service on the same port.
- The main drawback of this approach is the necessity to manage inside the code the difference between master and worker processes manually, typically with a classic if-else block, without the ability to easily modify the number of processes on-the-fly.

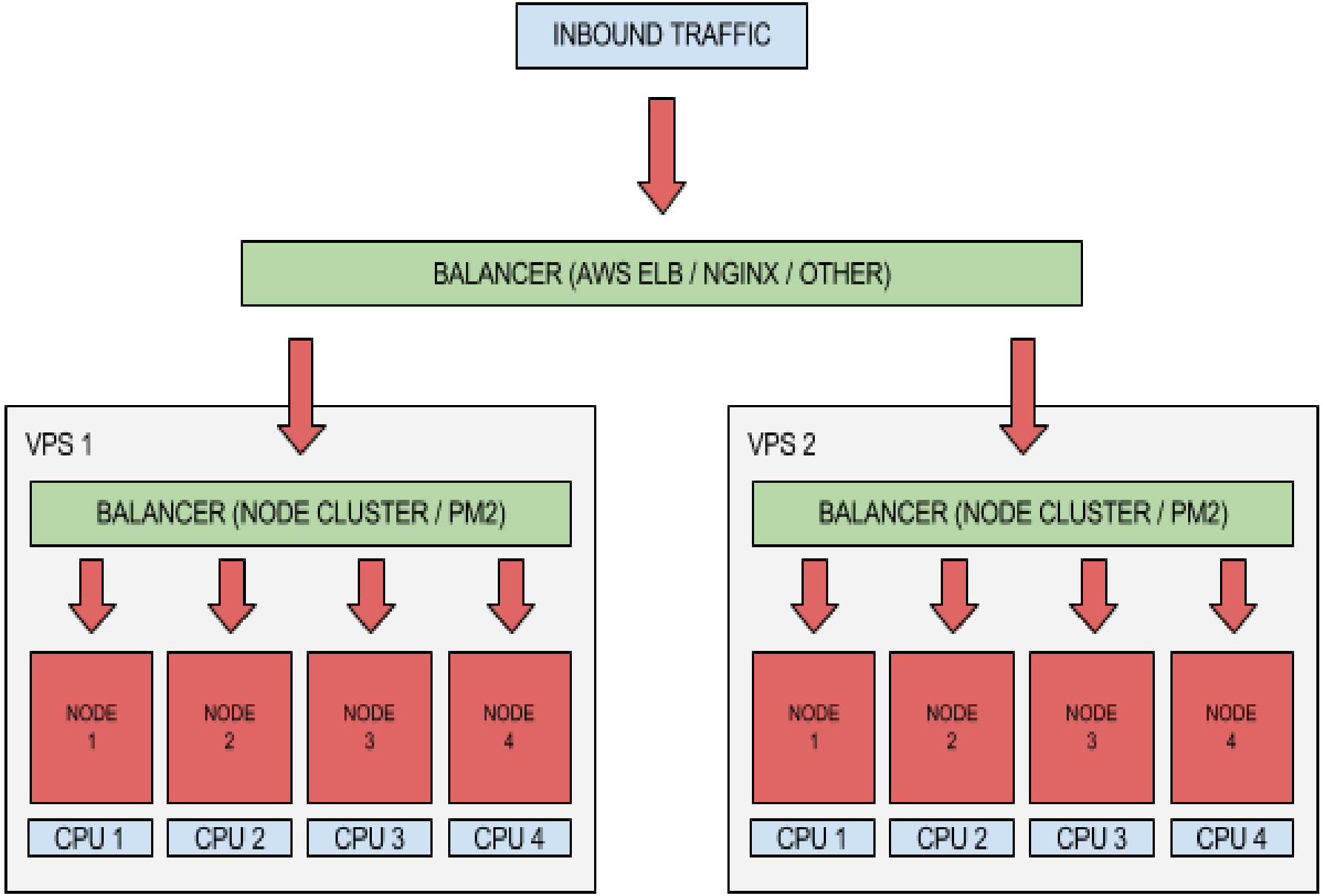
PM2 Cluster mode

- If we use PM2 as our process manager, there is a magic cluster feature that let you scale your process across all the cores without worrying about the cluster module.
- The PM2 daemon will cover the role of the “master” process, and it will spawn N processes of your application as workers, with round-robin balancing.
- Once your application is started in cluster mode, you can adjust the number of instances on-the-fly using “pm2 scale”, and perform “0-second-downtime” reloads, where the processes are restarted in series in order to have always at least one process online.
- As a process manager, PM2 will also take care of restarting your processes if they crash like many other useful things you should consider when running node in production.
- If you need to scale even further, you’ll probably need to deploy more machines.

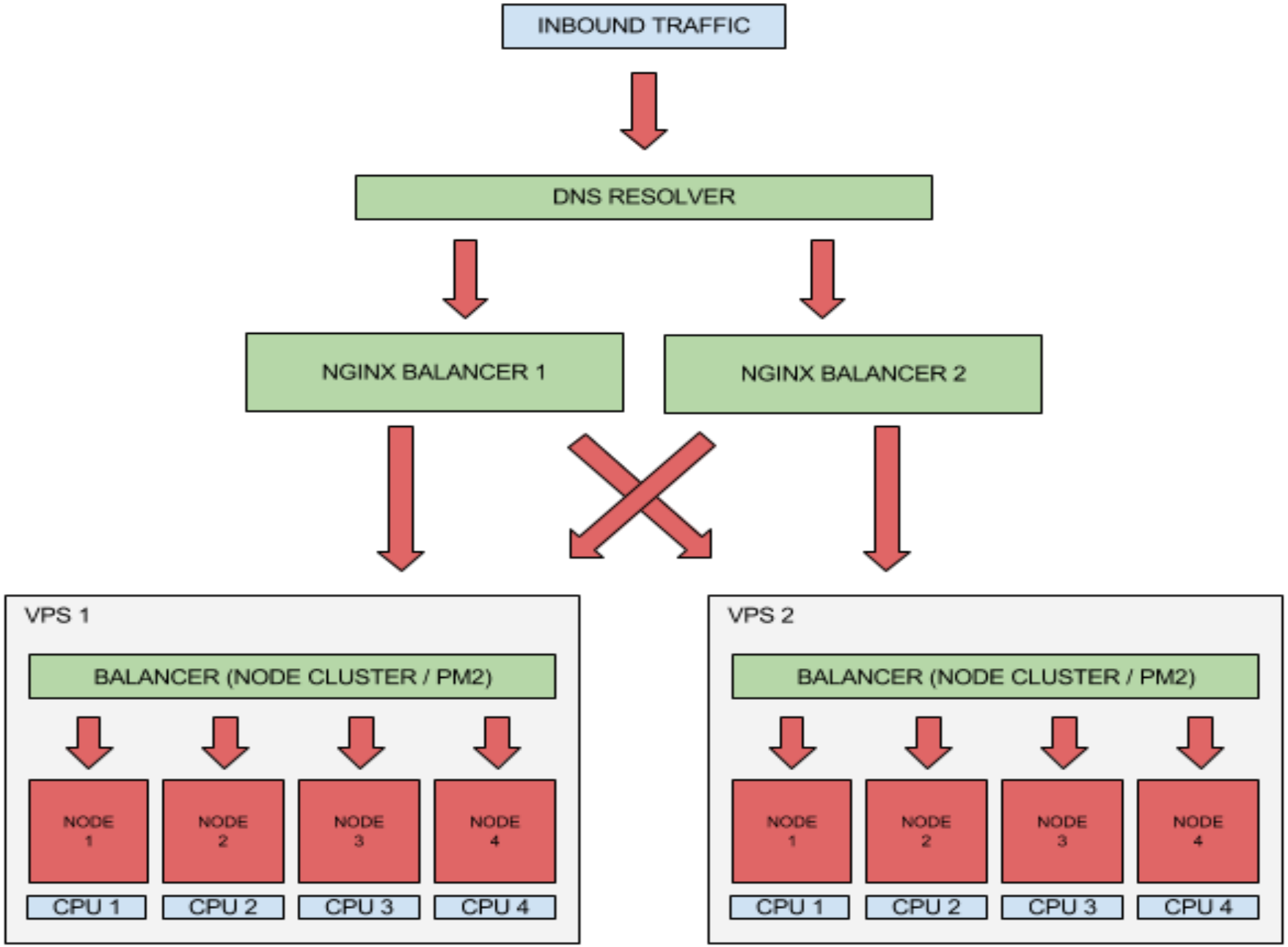
Multiple machines with network load balancing

- The main concept in scaling across multiple machines is similar to scaling on multiple cores, there are multiple machines, each one running one or more processes, and a balancer to redirect traffic to each machine.
- Once the request is sent to a particular node, the internal balancer described in the previous concept sends the traffic to a particular process.

- A network balancer can be deployed in different ways. If you use AWS to provision your infrastructure, a good choice is to use a managed load balancer like ELB (Elastic Load Balancer), because it supports useful features like auto-scaling, and it is easy to set up.


INBOUND TRAFFIC
BALANCER (AWS ELB / NGINX / OTHER)
VPS 1
BALANCER (NODE CLUSTER / PM2)
NODE 1
NODE 2
NODE 3
NODE 4
CPU 1
CPU 2
CPU 3
CPU 4
VPS 2
BALANCER (NODE CLUSTER / PM2)
NODE 1
NODE 2
NODE 3
NODE 4
CPU 1
CPU 2
CPU 3
CPU 4

- You can deploy a machine and setup a balancer with NGINX by yourself.
- By this way the load balancer will be the only entrypoint of your application exposed to the outer world.
- If you worry about it being the single point of failure of your infrastructure, you can deploy multiple load balancers that points to the same servers.
- In order to distribute the traffic between the balancers (each one with its own ip address), you can add multiple DNS “A” records to your main domain, so the DNS resolver will distribute the traffic between your balancers, resolving to a different IP address each time.
- The below figure how you can achieve redundancy also on the load balancers.


INBOUND TRAFFIC
DNS RESOLVER
NGINX BALANCER 1
NGINX BALANCER 2
VPS 1
BALANCER (NODE CLUSTER / PM2)
NODE 1
NODE 2
NODE 3
NODE 4
CPU 1
CPU 2
CPU 3
CPU 4
VPS 2
BALANCER (NODE CLUSTER / PM2)
NODE 1
NODE 2
NODE 3
NODE 4
CPU 1
CPU 2
CPU 3
CPU 4

What is MongoDB?

- **MongoDB** is an open-source database that uses a document-oriented NoSQL database used for high volume data storage and a non-structured query language.
- Instead of using tables and rows as in the traditional relational databases, MongoDB makes use of collections and documents.
- Documents consist of key-value pairs which are the basic unit of data in MongoDB.
- Collections contain sets of documents and function which is the equivalent of relational database tables.
- It is one of the most powerful NoSQL systems and databases around, today.

NOSQL

- NoSQL (Not only SQL) came as a solution for the problem of relational database management systems and allowed the companies to store massive amounts of structured, semi-structured and unstructured data in real-time.
- It does not certainly imply that it restricts the usage of SQL for these databases.
- Some of the popular NoSQL databases are HBase, Cassandra, IBm Informix, MongoDB, Amazon SimpleDB, Cloudata, etc.
- Today most of the world famous firms like Google, Facebook, Amazon, etc., are using NoSQL to provide cloud-based services in real-time.

Node.js MongoDB

- Node.js can be used in database applications.
- One of the most popular NoSQL database is MongoDB.
- To be able to experiment with the code examples, you will need access to a MongoDB database.
- You can download a free MongoDB database at https://www.mongodb.com.
- Or get started right away with a MongoDB cloud service at https://www.mongodb.com/cloud/atlas.

Install MongoDB Driver

- Let us try to access a MongoDB database with Node.js.
- To download and install the official MongoDB driver, open the Command Terminal and execute the following:
    - C:\Users*Your Name*>npm install mongodb
- Node.js can use this module to manipulate MongoDB databases:
    - var mongo = require('mongodb');

MongoDB Advantages

- **MongoDB is schema less**. It is a document database in which one collection holds different documents.
- There may be **difference between number of fields, content and size of the document** from one to other.
- **Structure of a single object is clear** in MongoDB.
- There are **no complex joins** in MongoDB.
- MongoDB provides the **facility of deep query** because it supports a powerful dynamic query on documents.
- It is very **easy to scale**.
- It **uses internal memory for storing working sets** and this is the reason of its fast access.

Distinctive features of MongoDB

- Easy to use
- Light Weight
- Extremely faster than RDBMS

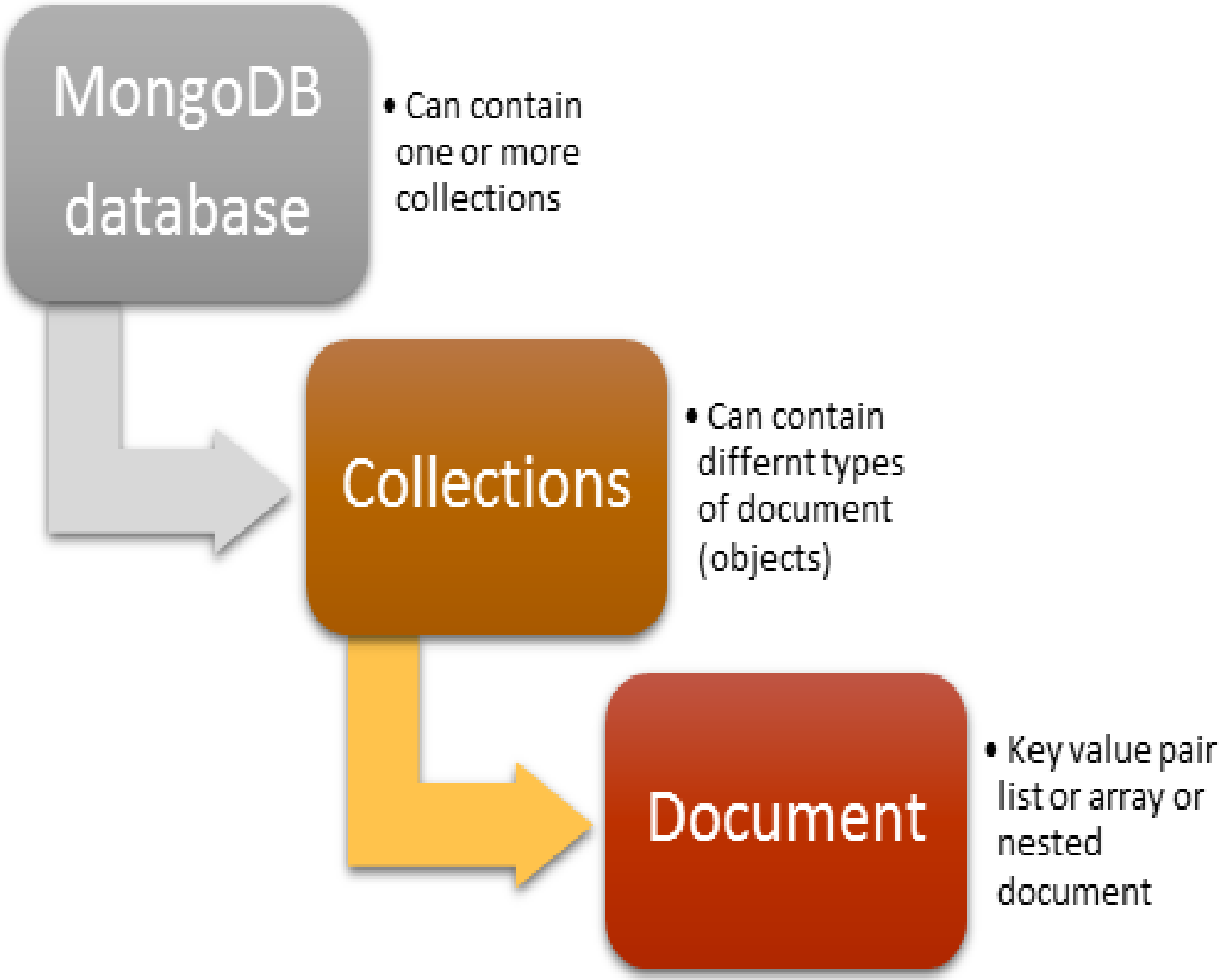
Where MongoDB should be used

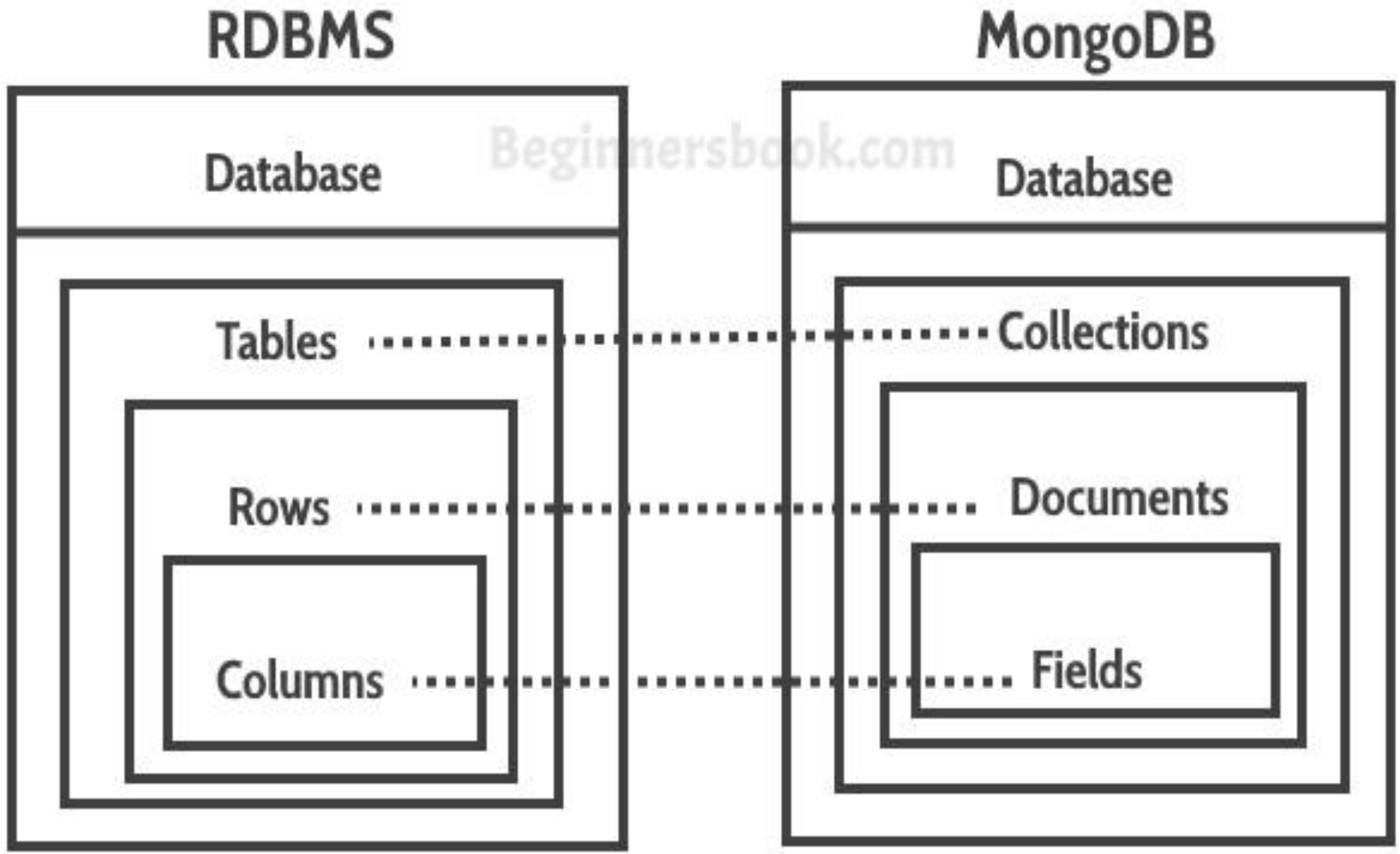
- Big and complex data
- Mobile and social infrastructure
- Content management and delivery
- User data management
- Data hub

Performance analysis of MongoDB and RDBMS

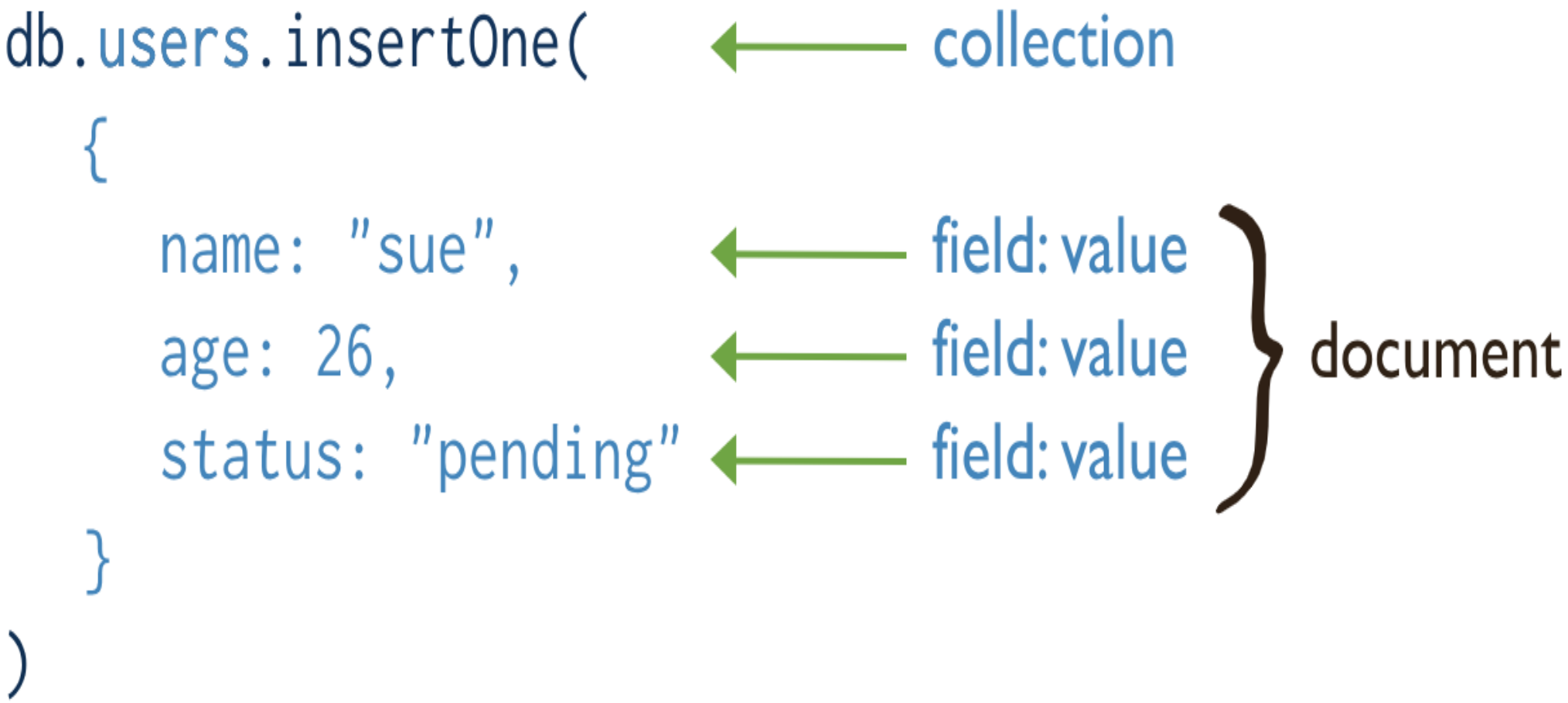
- In relational database (RDBMS) tables are using as storing elements, while in MongoDB collection is used.
- In the RDBMS, we have multiple schema and in each schema we create tables to store data while, MongoDB is a document oriented database in which data is written in BSON format which is a JSON like format.
- MongoDB is almost 100 times faster than traditional database systems.

- **Database:** Database is a physical container for collections. Each database gets its own set of files on the file system. A single MongoDB server typically has multiple databases.
- **Collection:** Collection is a group of MongoDB documents. It is the equivalent of an RDBMS table. A collection exists within a single database. Collections do not enforce a schema. Documents within a collection can have different fields. Typically, all documents in a collection are of similar or related purpose.
- **Document:** A document is a set of key-value pairs. Documents have dynamic schema. Dynamic schema means that documents in the same collection do not need to have the same set of fields or structure, and common fields in a collection's documents may hold different types of data.

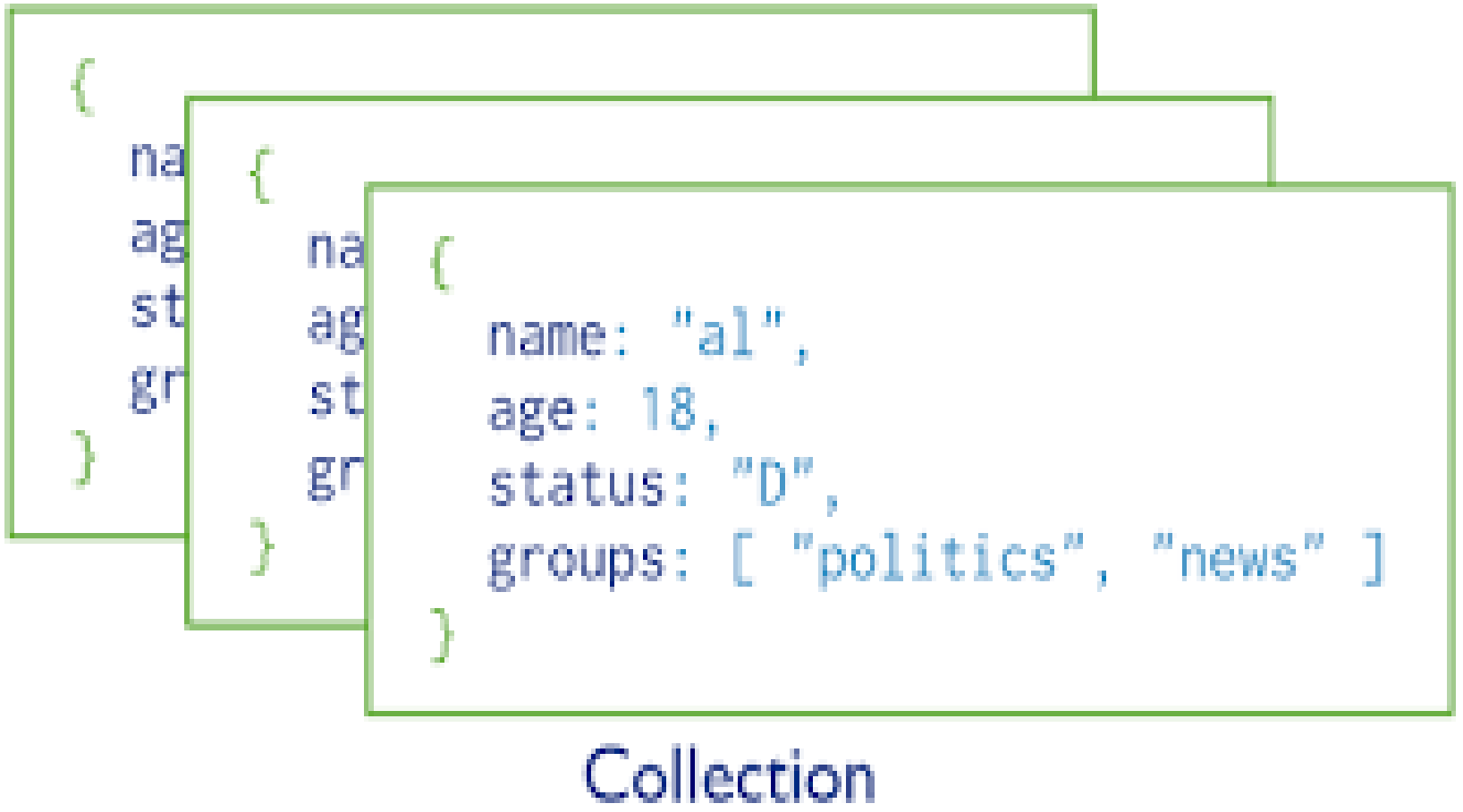

MongoDB database
• Can contain one or more collections
Collections
• Can contain differnt types of document (objects)
Document
• Key value pair list or array or nested document

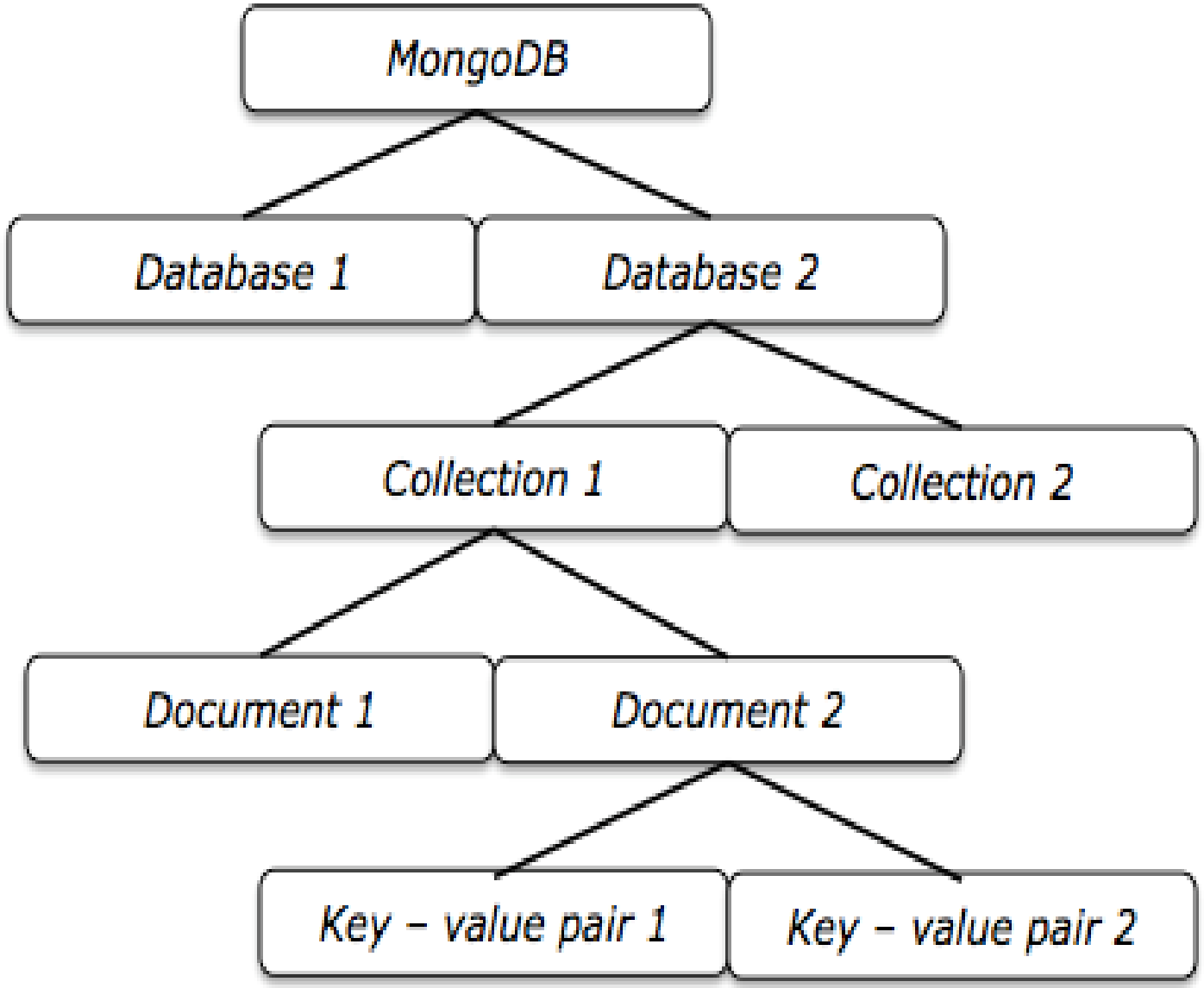

RDBMS
MongoDB
Database
Database
Tables
Collections
Rows
Documents
Columns
Fields

Sample Document



Collection




MongoDB
Database 1
Database 2
Collection 1
Collection 2
Document 1
Document 2
Key – value pair 1
Key – value pair 2

MongoDB Datatypes

String	String is the most commonly used datatype. It is used to store data. A string must be UTF 8 valid in mongodb.
Integer	Integer is used to store the numeric value. It can be 32 bit or 64 bit depending on the server you are using.
Boolean	This datatype is used to store boolean values. It just shows YES/NO values.
Double	Double datatype stores floating point values.
Min/Max Keys	This datatype compare a value against the lowest and highest bson elements.
Arrays	This datatype is used to store a list or multiple values into a single key.
Object	Object datatype is used for embedded documents.
Null	It is used to store null values.
Symbol	It is generally used for languages that use a specific type.
Date	This datatype stores the current date or time in unix time format. It makes you possible to specify your own date time by creating object of date and pass the value of date, month, year into it.

Data Model Design

- MongoDB provides two types of data models: Embedded data model and Normalized data model. Based on the requirement, you can use either of the models while preparing your document.
- Embedded Data Model
  - In this model, you can have (embed) all the related data in a single document, it is also known as de-normalized data model.
- Normalized Data Model
  - In this model, you can refer the sub documents in the original document, using references.

Example of Embedded Data Model

- Assume we are getting the details of employees in three different documents namely, Personal_details, Contact and, Address, you can embed all the three documents in a single one.

```
{
_id: ,
Emp_ID: "10025AE336"
Personal_details:{
First_Name: "Radhika",
Last_Name: "Sharma",
Date_Of_Birth: "1995-09-26"
},
Contact:
{
e-mail: "radhika_sharma.123@gmail.com",
phone: "9848022338"
},
Address:
{
city: "Hyderabad",
Area: "Madapur",
State: "Telangana"
}
}
```

Example of Normalized Data Model

- you can re-write the above document in the normalized model as:

Employee:

```
{
_id: <ObjectId101>,
Emp_ID: "10025AE336"
}
```

Personal_details:

```
{
_id: <ObjectId102>,
 empDocID: " ObjectId101",
First_Name: "Radhika",
 Last_Name: "Sharma",
Date_Of_Birth: "1995-09-26"
}
```

Contact:

```
{
_id: <ObjectId103>,
empDocID: " ObjectId101",
e-mail: "radhika_sharma.123@gmail.com",
phone: "9848022338"
}
```

Address:

```
{
_id: <ObjectId104>,
empDocID: " ObjectId101",
 city: "Hyderabad", Area:
"Madapur", State: "Telangana"
}
```

Considerations while designing the schema in Mongodb

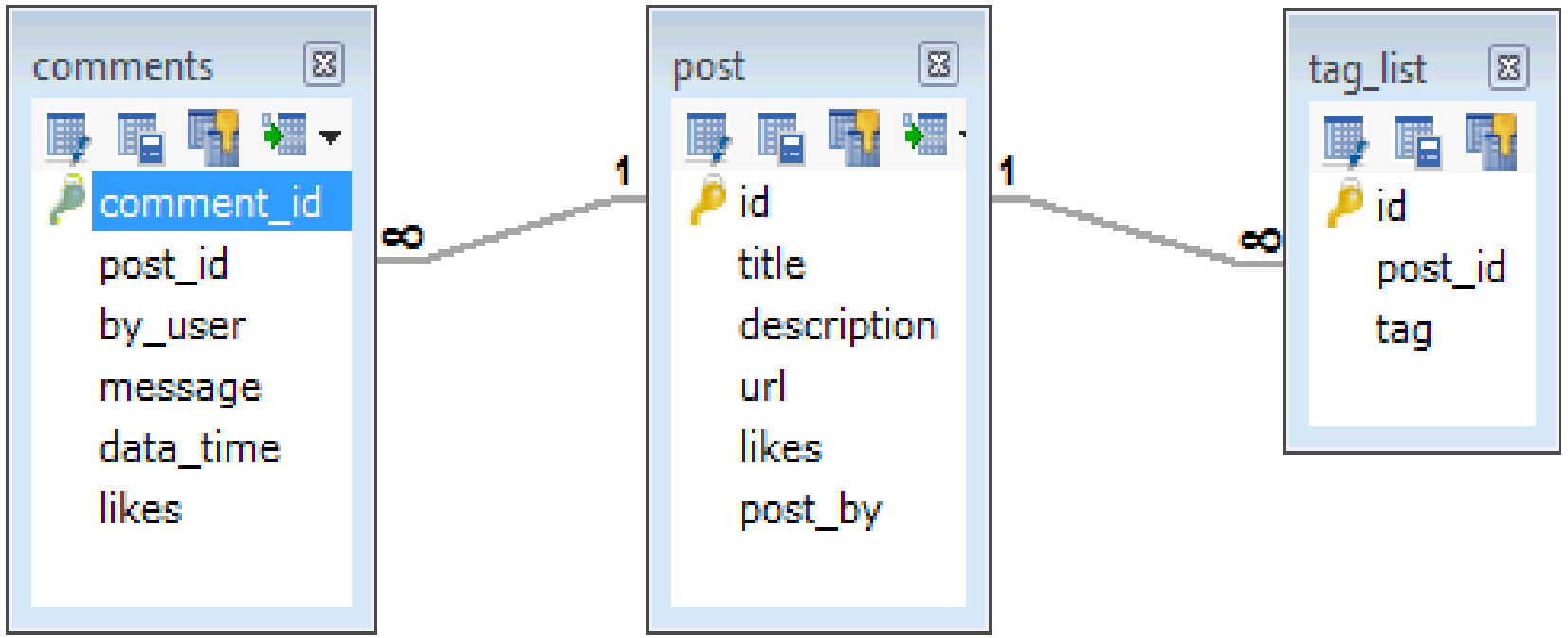
- Design your schema according to user requirements.
- Combine objects into one document if you will use them together. Otherwise separate them (but make sure there should not be need of joins).
- Duplicate the data (but limited) because disk space is cheap as compare to compute time.
- Do joins while write, not on read.
- Optimize your schema for most frequent use cases.

Example schema

- Suppose a client needs a database design for his blog/website and see the differences between RDBMS and MongoDB schema design. Website has the following requirements.

  - Every post has the unique title, description and url.
  - Every post can have one or more tags.
  - Every post has the name of its publisher and total number of likes.
  - Every post has comments given by users along with their name, message, data-time and likes.
  - On each post, there can be zero or more comments.

- In RDBMS schema, design for above requirements will have minimum three tables.



- While in MongoDB schema, design will have one collection post and the following structure

```
{
_id: POST_ID
title: TITLE_OF_POST,
description: POST_DESCRIPTION,
by: POST_BY,
url: URL_OF_POST,
tags: [TAG1, TAG2, TAG3],
likes: TOTAL_LIKES,
comments: [
{ user:'COMMENT_BY',
message: TEXT,
dateCreated: DATE_TIME,
 like: LIKES },
{
user:'COMMENT_BY',
message: TEXT,
dateCreated: DATE_TIME,
like: LIKES
}
]
}
```

- So while showing the data, in RDBMS you need to join three tables and in MongoDB, data will be shown from one collection only.

CRUD operations

- CRUD operations imply to the fundamental operations supported by MongoDB, which are Create, Read, Update, and Delete.
- **Create operation –** Create operation or Insert operation are used to add new documents to the collection and if the collection does not exist, it creates one.
    - Following command can insert a document on the collection – **db.collection.insert()**
- **Read operation –** This operation reads the documents from the collection. This process is taken place by executing a query.
    - The command to read the document is – **db.collection.find()**
- **Update operation –** Update operation is used to modify an existing document.
    - The command that updates a document is – **db.collection.update()**
- **Delete operation –** Delete operation erases the document from the collection.
    - Following command performs the operations – **db.collection.remove()**