Who am I?

Humera Tariq

PhD, MS, MCS (Computer Science), B.E (Electrical)

Postdoc (Medical Image Processing, Deep Neural Networks)

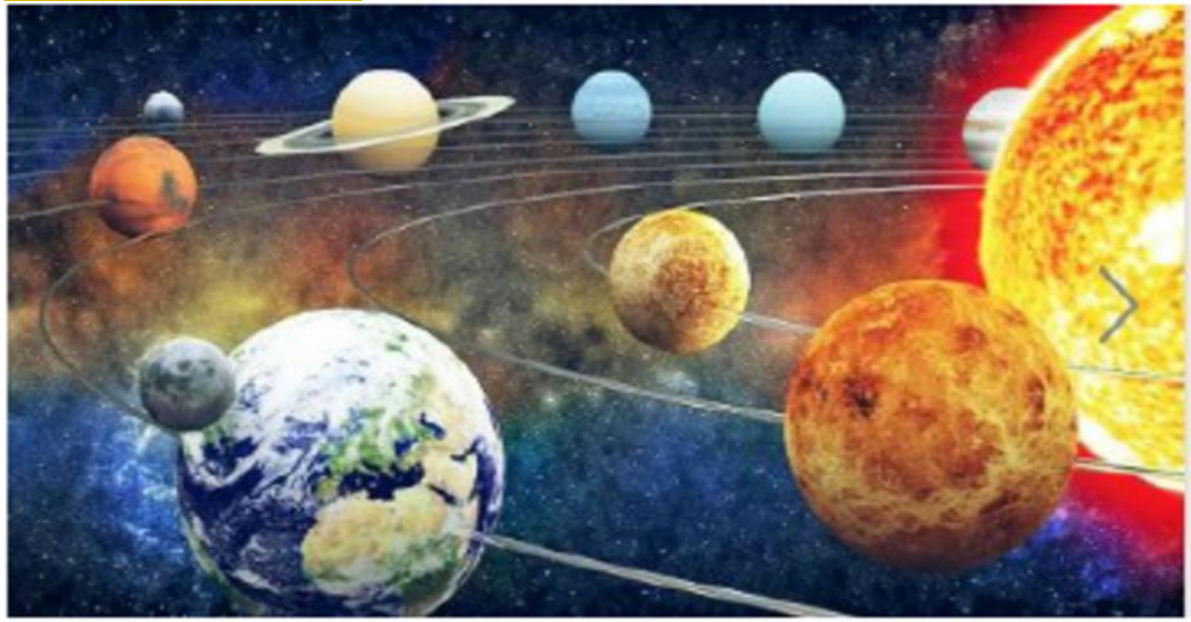
Email: humera@uok.edu.pk

Web: https://humera.pk/

Discord: https://discord.gg/xeJ68vh9

Starting in the name of Allah,

the most beneficial,
the most merciful.



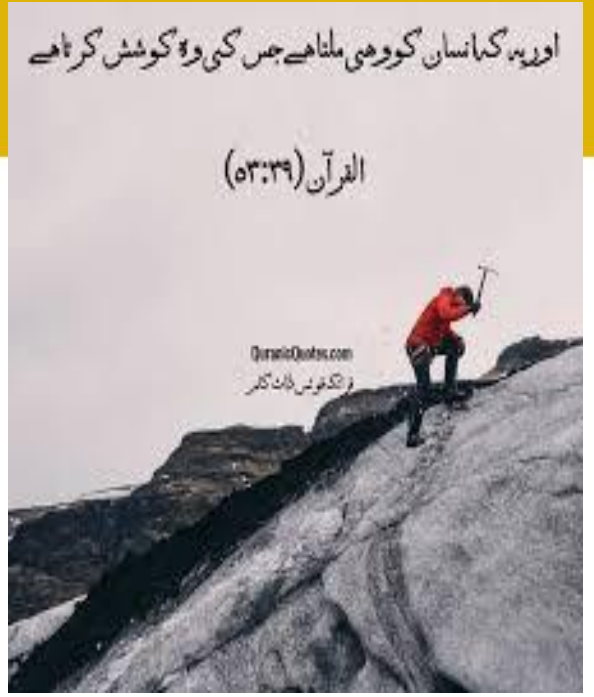
أَمْ لِلْإِنسَانِ مَا تَمَنَّىٰ ﴿٢٤﴾

کیا انسان کو ہر وہ چیز حاصل ہے جس کی اس نے تمنا کی؟

UNIVERSITY OF KARACHI

Surah An-Najm Chapter 53 Verse 39

And there is not for man except that [good] for which he strives.



UNIVERSITY OF KARACHI

Week 11

Internet Application Development

Department of Computer Science (DCS/UBIT)
University of Karachi
January 2025

Week 11

Internet Application Development

Slides	CORS	JS Scope chain, EC, GEC, FEC, Lexical env, Closure ,	React closure, Hoisting revision	front-end + back-end
	5	15-34 ~20 slides	Next time	40-70 ~30 slides

Let’s briefly talk about CORS

What is CORS?

CORS stands for _____ , a security feature built into browsers.

CORS enforce security by blocking the _____(request/response) from being accessible to your _____(front-end/back-end) , unless the _____(client/server) explicitly allows it via _____ .

For example:

- Your frontend is hosted at `frontend.com`.
- Your backend API is hosted at `api.backend.com`.

The browser treats these as different origins and blocks the _____ (requests/response) unless it's explicitly allowed.

UNIVERSITY OF KARACHI

Why Does It Happen?

CORS errors are triggered by the _____ (Same-Origin Policy/Cross-Origin Policy), which prevents malicious websites from making unauthorized API calls using your credentials.

When the _____ (server/client) doesn't include the right CORS headers, the _____ refuses to share the _____ (request/response) and throws this error:

Access to _____ at 'https://api.backend.com' from origin 'https://frontend.com' has been _____ by CORS policy: No ' _____ ' header is present.

In short, the browser isn't blocking the _____ , it's blocking the _____ for-security reasons.

UNIVERSITY OF KARACHI

CORS Example

```
fetch("https://api.backend.com/data")
  .then(res => res.json())
  .then(data => console.log(data));
```

```
fetch("http://localhost:5000/data", {
  method: "POST",
  headers: {
    "Content-Type": "application/json"
  },
  body: JSON.stringify({ name: "Ali" })
});
```

If api.backend.com doesn't send the proper Access-Control-Allow-Origin header:

- ✓ The browser makes the request
- ✓ The server might even respond
- ✓ The browser blocks access to res.json()
- ✓ You see a CORS error in the console.

UNIVERSITY OF KARACHI

Facts Behind CORS

- ✓ It’s not a frontend issue.
- ✓ It’s not a browser bug.
- ✓ It’s a server-side configuration responsibility. The server must send the right headers

How Do You Fix It?

For complex requests (like POST with custom headers), browsers send a preflight request before the actual call. The server must respond to this with “ACAM” , “ACAH”

FIXING CORS: THREE STEPS

UPDATE THE BACKEND

- Access-Control-Allow-Origin: *
- Or specify trusted domains like Access-Control-Allow-Origin: https/frontend.com

HANDLE PREFLIGHT REQUESTS

Browsers send as an OPTIONS request

- Access-Control-Allow-Methods: GET, POST, OPTIONS
- Access-Control-Allow-Headers: Content-Type, Authorization

USE A PROXY FOR LOCAL DEVELOPMENT

Set up a proxy to forward requests — the same origin as your frontend

CAN WE BYPASS CORS?

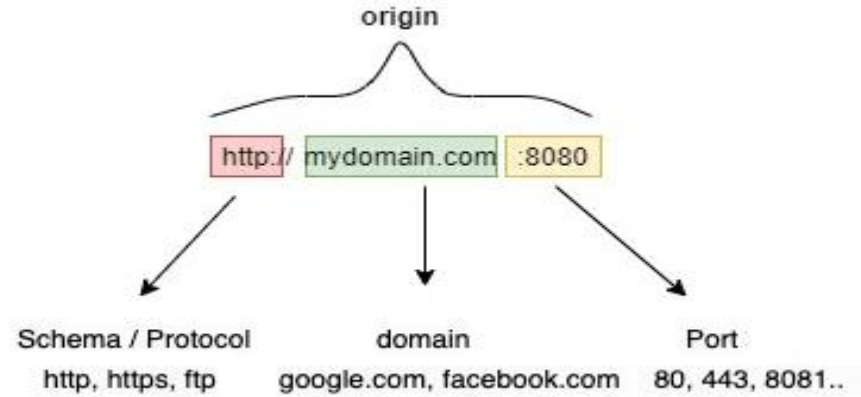
Short answer, No.

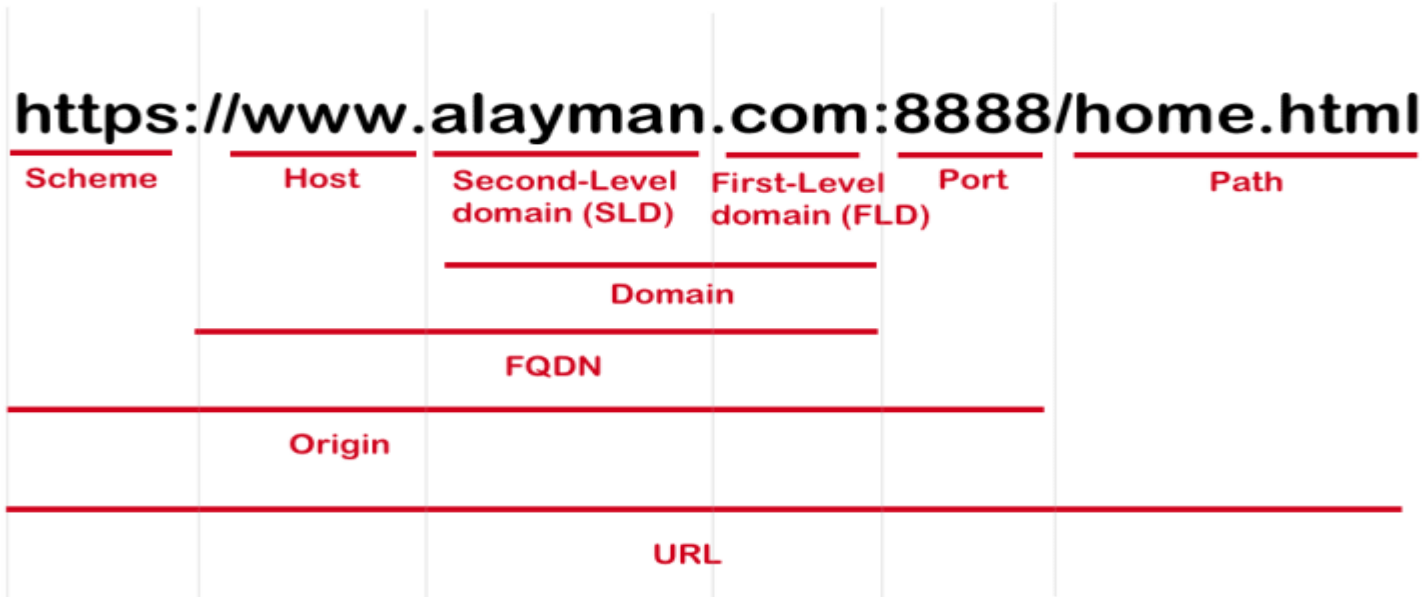
Hacky workarounds or extensions won’t work in production. Fix it properly by configuring the server

WHY IS THIS IMPORTANT IN INTERVIEWS?

- Understand how the web works

UNIVERSITY OF KARACHI





FQDN = Host + Domain

Origin = Scheme + Host + Domain + Port

URL = Scheme + Host + Domain + Port + Path

You're building a frontend app running at http://localhost:3000 that tries to fetch data from your backend API running at http://localhost:5000.

You're using **Express.js** in your backend. How do you update the server to allow requests from your frontend?

Make sure that your server
- ✓ Handles preflight automatically
- ✓ Sends appropriate headers

Front end (React+Vite)

```jsx
import { useEffect, useState } from 'react';

function App() {
  const [data, setData] = useState(null);

  useEffect(() => {
    fetch('http://localhost:5000/data')
      .then(res => res.json())
      .then(setData)
      .catch(err => console.error('CORS issue?', err));
  }, []);

  return (
    <div>
      <h1>Vite + React CORS Demo</h1>
      <pre>{JSON.stringify(data, null, 2)}</pre>
    </div>
  );
}

export default App;
```

Back end

```
// server.js
import express from 'express';
import cors from 'cors';

const app = _____ ;

app.use(cors({
  origin: ' _____ '
}));

app.get('/data', (req, res) => {
  res.json({ message: 'Hello from backend!' });
});

app._____ (5000, () => {

console.log('Server running on http://localhost:5000');

});
```

Cors is basically Middleware

```
import express from 'express';
import cors from 'cors';

const app =  _____ ;

app.use(cors({
  origin: ' _____  ', // frontend origin
  methods: [ _____  ], // Access-Control-Allow-Methods
  allowedHeaders: [ '_____  ', '  _____ '] // Access-Control-Allow-Headers
}));

app.get('/data', (req, res) => {
  res.json({ message: 'Using cors middleware' });
});

app._____ (5000);
```

UNIVERSITY OF KARACHI

IN JAVASCRIPT, WHAT DOES THE "VAR" KEYWORD DO? WHY IS IT BETTER TO USE EITHER "LET" OR "CONST"?

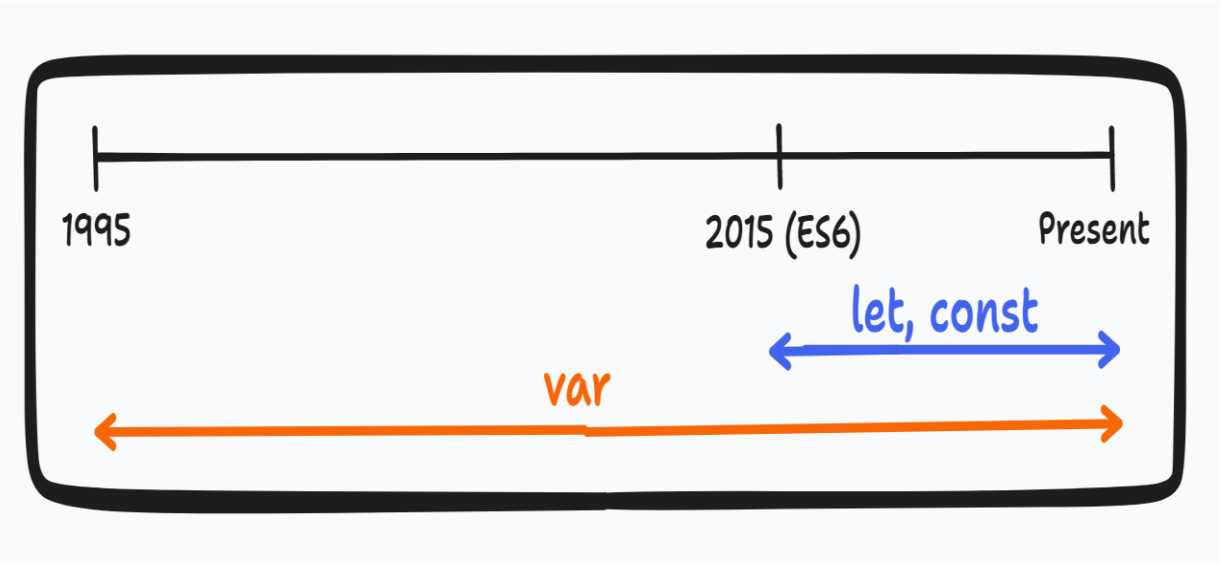
Scope-chain -→ call stack → closure

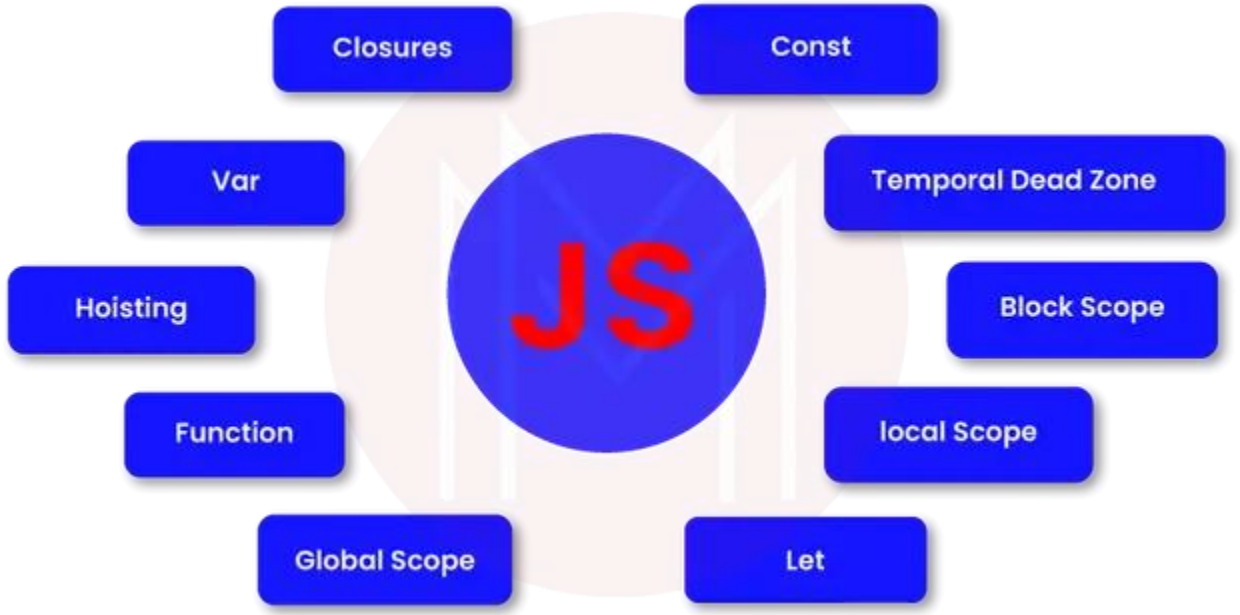
Expectations for proper answer

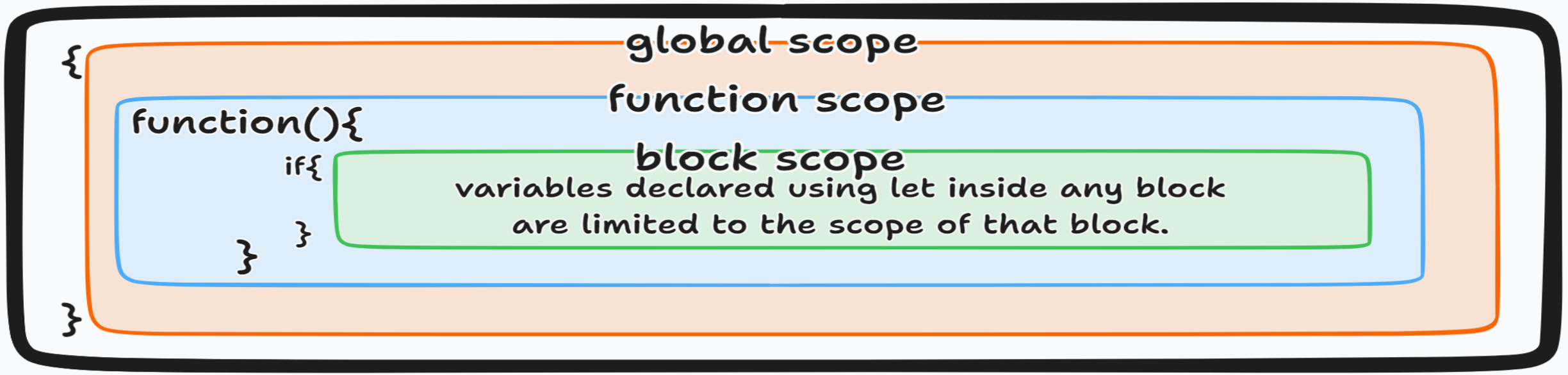
JS closures-and-scoping/

-understanding-execution-context-

and-variable-hoisting





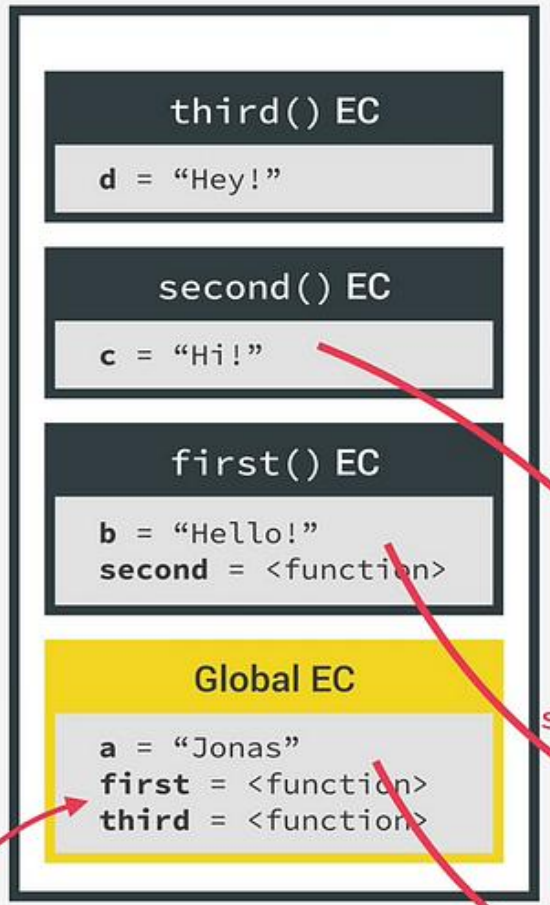


	global scoped	function scoped	block scoped	reassignable	redeclarable	can be hoisted
var	+	+	-	+	+	+
let	-	+	+	+	-	-
const	-	+	+	-	-	-

Scope Chain vs call stack

SCOPE CHAIN VS. CALL STACK

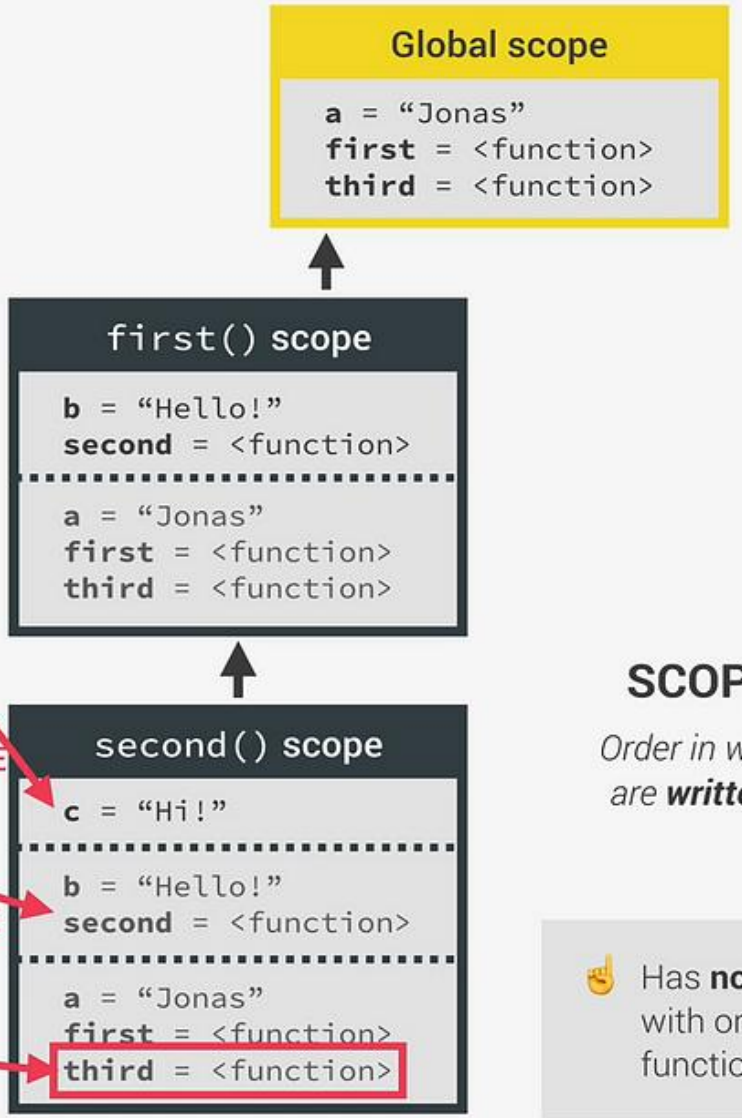
```
const a = 'Jonas';
first();

function first() {
  const b = 'Hello!';
  second();

  function second() {
    const c = 'Hi!';
    third();
  }
}

function third() {
  const d = 'Hey!';
  console.log(d + c + b + a);
  // ReferenceError
}
```



Variable environment (VE)

CALL STACK

*Order in which functions were **called***

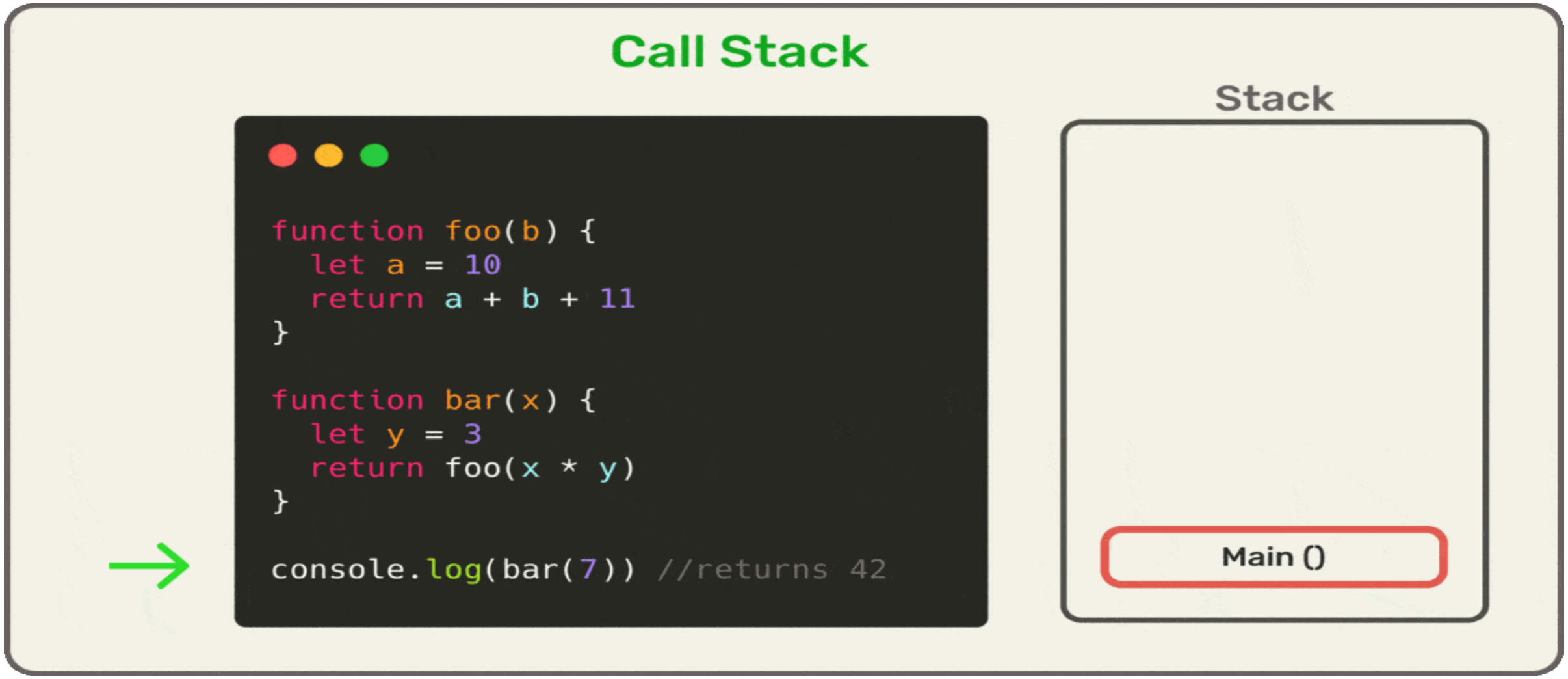


SCOPE CHAIN

*Order in which functions are **written in the code***

☝ Has **nothing** to do with order in which functions were **called**!

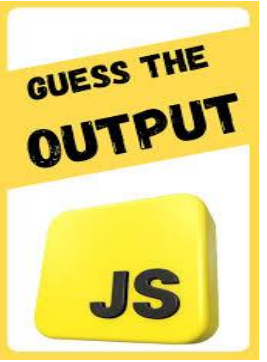
Call Stack which is a part of V8 engine, and it is responsible for the order of executing the code



Scop chain: Uncaught Reference Error !

```
function outer() {
    // Variable 'a' is declared in the local scope of outer()
    var a = 10;

    // inner() is defined inside outer(), so it forms a nested (child) scope
    function inner() {
        // Variable 'b' is declared in the local scope of inner()
        var b = 20;

        // When we use console.log(a), JavaScript looks for 'a' in the current (inner) scope
        // It does NOT find it there, so it goes up one level to the outer() function scope
        // It finds 'a' there and prints its value
        console.log(a); //  Found in outer() → prints: 10

        // Now it looks for 'b' — and this time, it finds it directly in the inner() scope
        console.log(b); //  Found in inner() → prints: 20
    }

    // We call the inner function from within outer()
    inner();

    // Now we try to access variable 'b' here in outer()
    // But 'b' is defined INSIDE inner(), and is not visible here
    // So JavaScript checks outer()'s own scope and global scope but doesn
    // This results in an error if we uncomment the line below
    // console.log(b);  ReferenceError: b is not defined
}

// Start the execution by calling the outer() function
outer();
```

A new Function Execution Context (FEC) is created whenever a function is invoked (or called).
This provides *encapsulation* and enables each function to have its own *private scope*.



Module scope

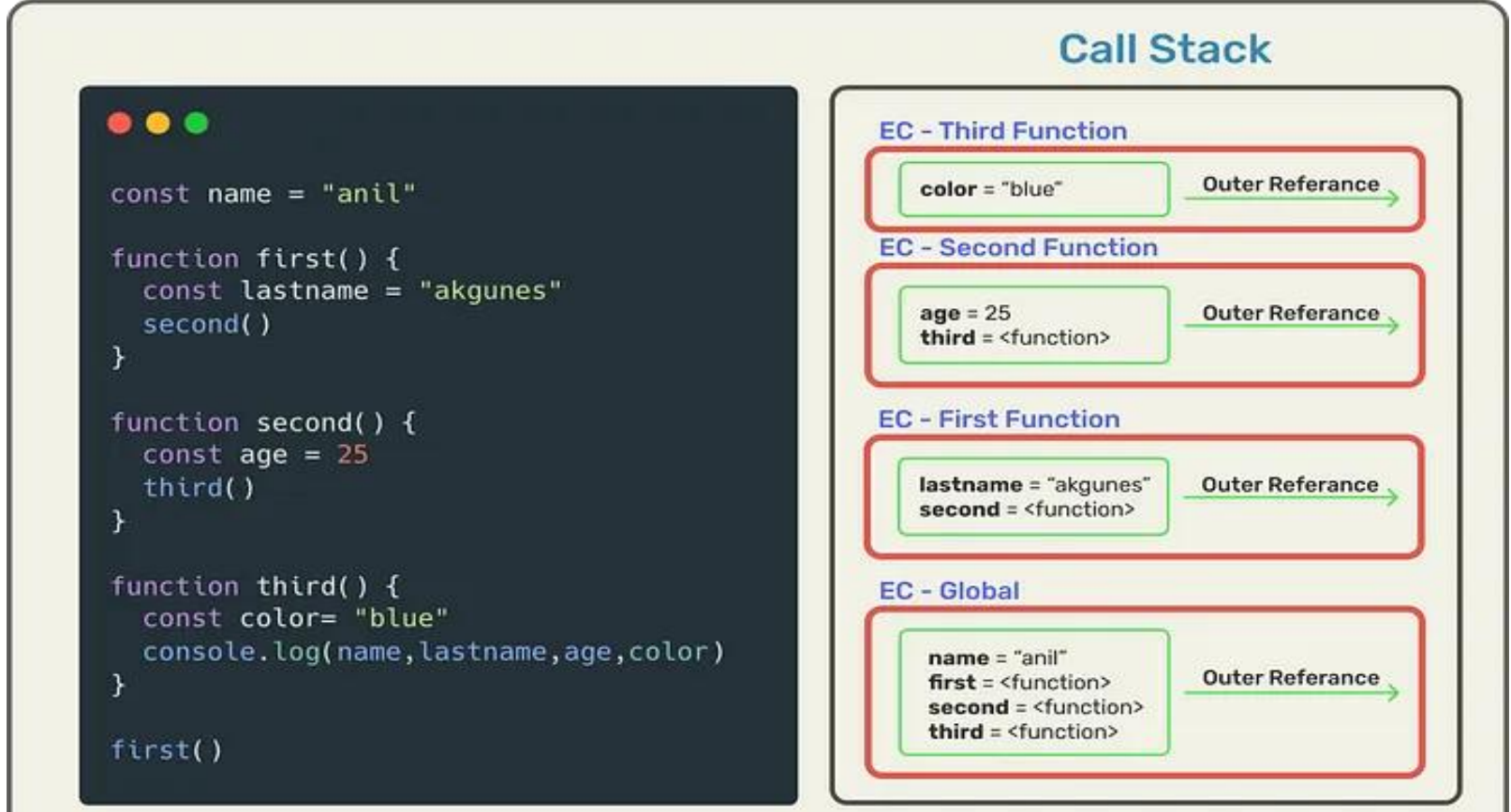
```
<script type="module">
    const foo = "foo";
</script>
<script>
    console.log(foo); // Uncaught ReferenceError: foo is not defined
</script>
```

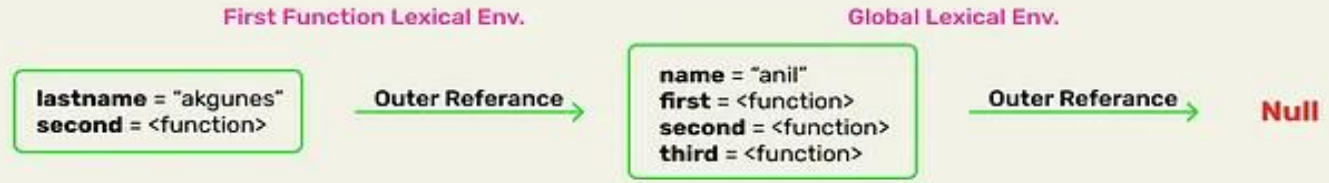
Scope = Lexical environment – outer reference

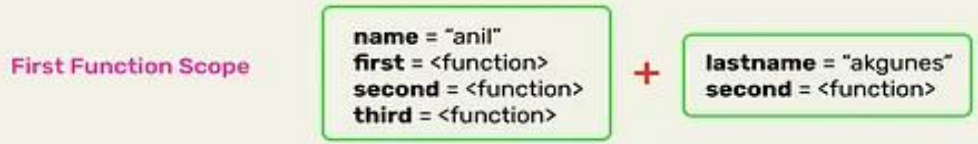
JS Word bank

Parent, reference , data structure, local variable , execution context, lexical environment, environment record, parent's lexical environment

```
const name = "anil"

function first() {
  const lastname = "akgunes"
  second()
}

function second() {
  const age = 25
  third()
}

function third() {
  const color= "blue"
  console.log(name,lastname,age,color)
}

first()
```







JS Word bank

Parent, reference , data structure, local variable , execution context, lexical environment, environment record, parent's lexical environment

Every rectangular block on previous slide is an _____ . Every execution context creates a _____ for variables**.** The lexical environment is basically _____ that keeps the variable and their value reference in memory so that it can easily find it for execution context. Lexical environment consist of two parts : _____ and _____ **to outer lexical environment**. While environmental record keeps the _____ variable data, outer reference keeps a reference for _____lexical environment .

Closure

(inner functions memory to remember _____ and _____on its current scope)

There are many uses of closures, from creating class-like structures that store state and implement
---private methods, Encapsulation ,
---passing callbacks to event handlers and
---Higher order function (HOC),
---recursion.

Scope Chaining in Action

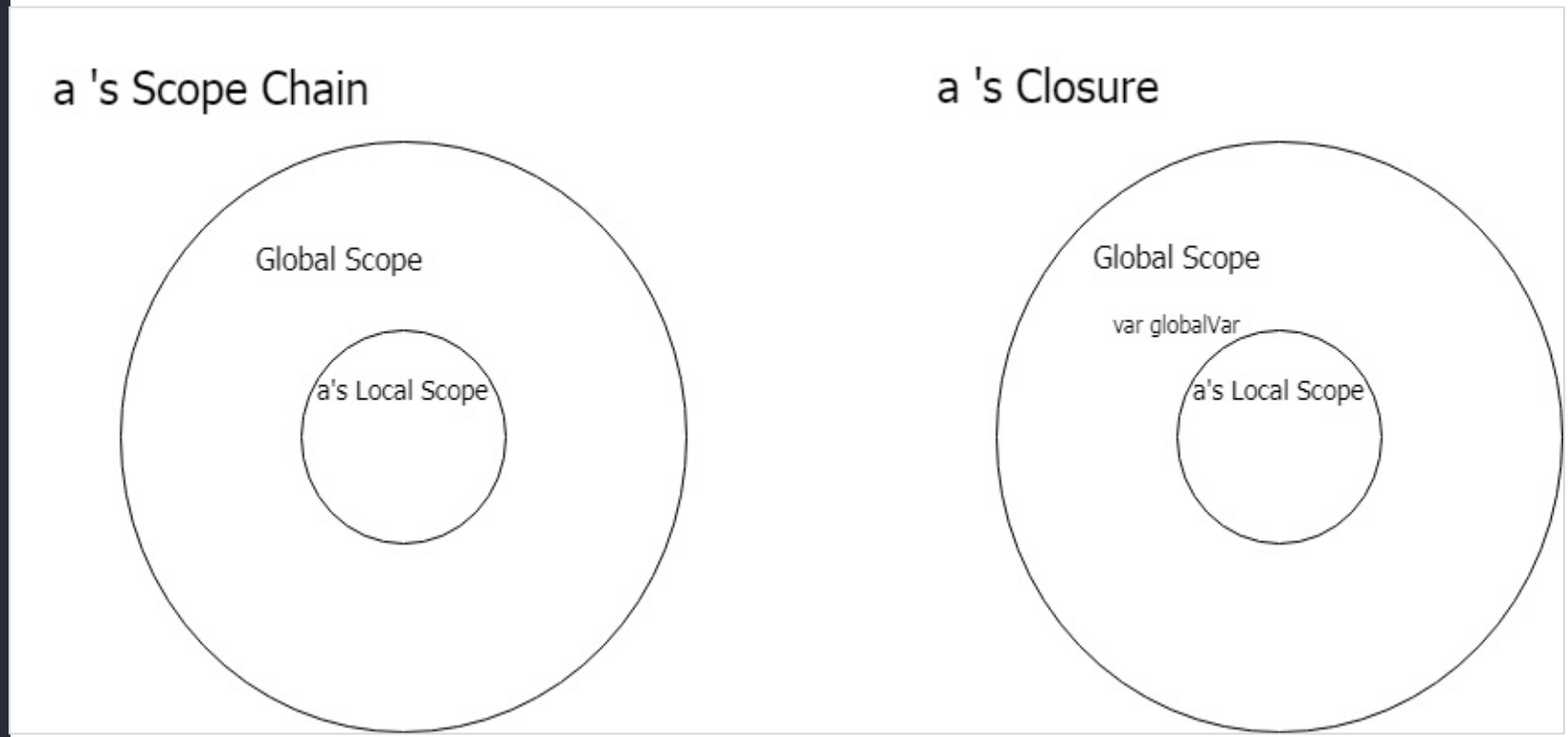
The ability for closures to access variables from outer scopes even after those scopes have exited is possible *due to JavaScript's scope chaining mechanism.*

The Magic: Persisting State

What's magical about closures? is that they remember the environment in which they were created.

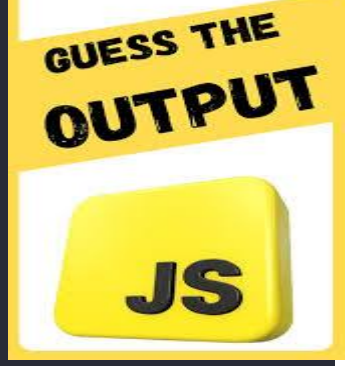
Even if the outer function has finished executing and its variables are out of scope, the inner function still retains access to them.

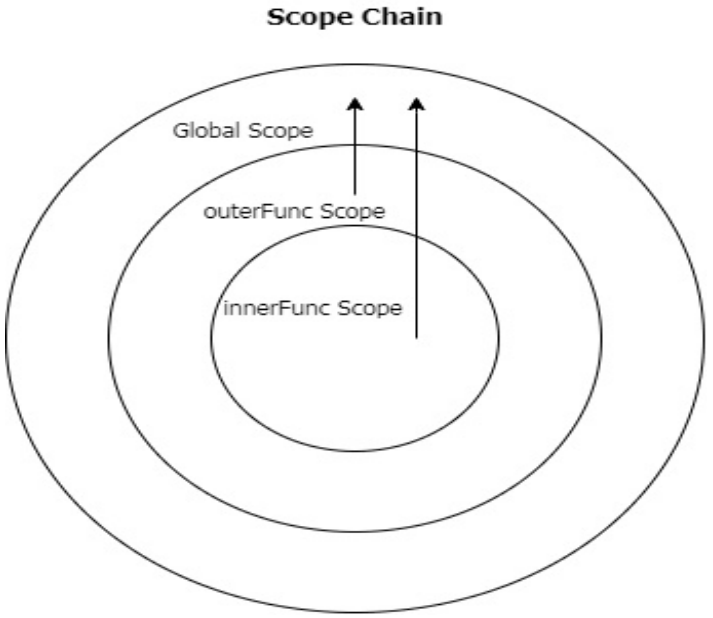
```
//Global's Scope
var globalVar = 'abc';

function a() {
  //testClosures's Scope
  console.log(globalVar);
}

a(); //logs "abc"
/* Scope Chain
   Inside a function perspective
   a's scope -> global's scope
*/
```



*A **child function** remembers and has access to the variables of its **parent function**, even after the parent has executed and returned.*

```
function outer() {
  let secret = "I am a secret!";

  return function inner() {
    console.log(secret); // inner function has access to 'secret' from outer
  };
}

const myClosure = outer();
myClosure(); // I am a secret! // inner remembers outer's variables
```





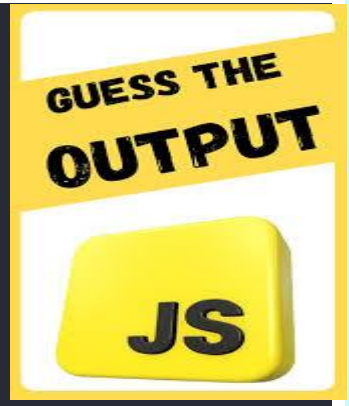

```
var globalVar = 'global';
var outerVar = 'outer';

function outerFunc(outerParam) {
  function innerFunc(innerParam) {
    console.log(globalVar, outerParam, innerParam);
  }
  return innerFunc;
}

const x = outerFunc(outerVar);
outerVar = 'outer-2';
globalVar = 'guess';
x('inner');
```

```
function factory(input) {
  function product(tool) {
    console.log(input, tool);
  }
  return product;
}

const thing = factory('outer'); // thing is now a "product-maker" for 'outer'
thing('inner'); // outputs: outer inner
```

```
const arrFuncs = [];
for(var i = 0; i < 5; i++){
  arrFuncs.push(function (){
    return i;
  });
}
console.log(i);

for (let i = 0; i < arrFuncs.length; i++) {
  console.log(arrFuncs[i]());
}
```

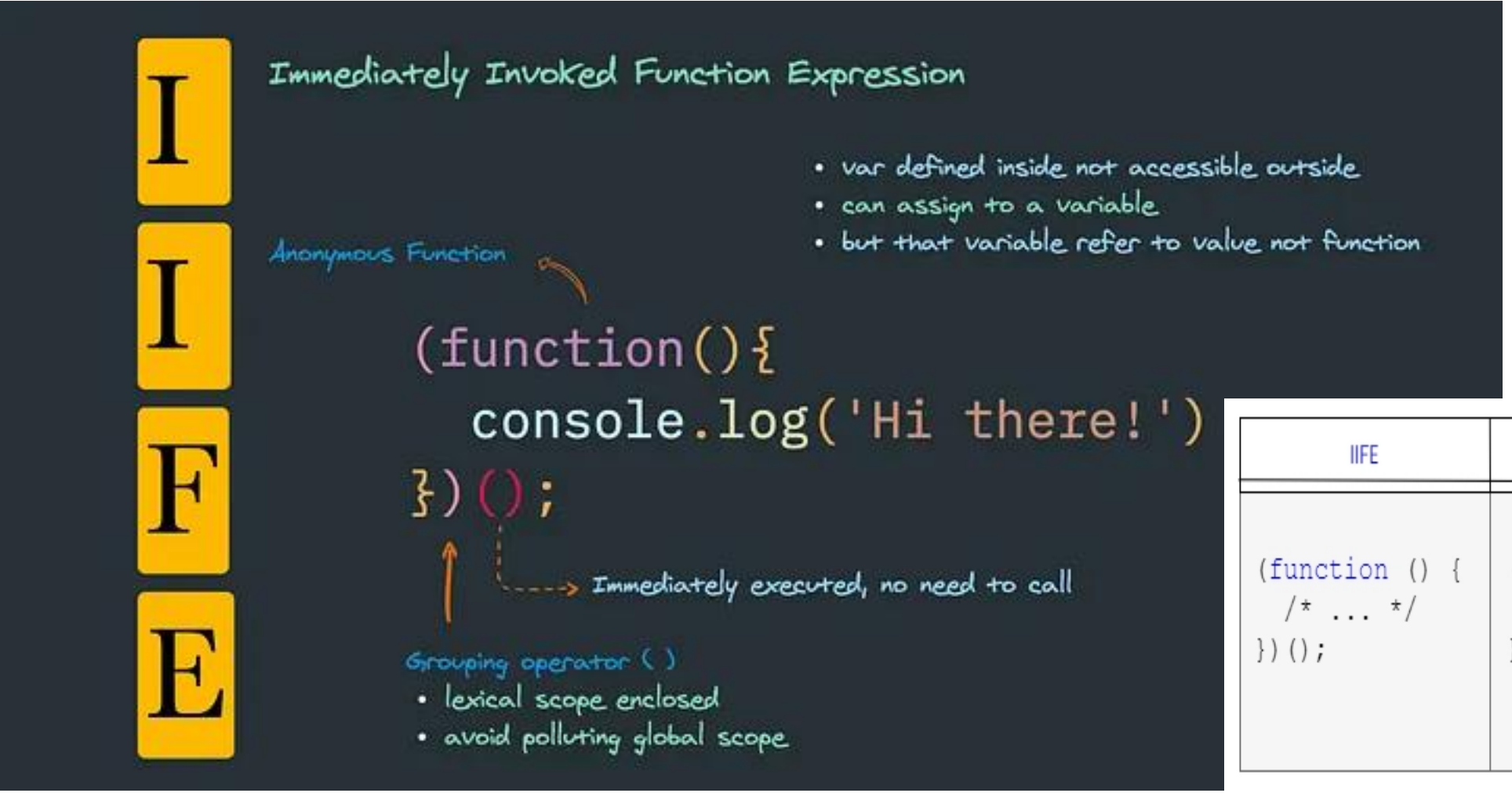
// what will be the value of 'I' and why

// what will be the value



This code is not working as we expected because of _____ . The var keyword makes a _____ variable and when we push a function we return the _____ variable i. So, when we call one of those functions in that array after the loop it logs _____ because we get the current value of i which is _____ and we can access it because it's a _____ variable.

Because **closure** keeps the _____ of that variable not its values at the time of its creation. We can solve this using _____ or changing the var keyword to _____ for block-scoping.




```
(function (param) {
  console.log(param);
})('Hello'); // prints: Hello
```

IIFE	Arrow function IIFE	async IIFE
`(function () { /* ... */ })();`	`(() => { /* ... */ })();`	`(async () => { /* ... */ })();`

Immediately Invoked Function Expression (IIFE) *classic technique of* ***creating a new scope per iteration****, to capture the loop variable* ***by value****, not reference.*

Fix using IIFEC

```
const arrFuncs = [];
for (var i = 0; i < 5; i++) {
  (function (j) {
    arrFuncs.push(function () {
      return j;
    });
  })(i); // pass current i as j
}

console.log(i); // 5

for (let i = 0; i < arrFuncs.length; i++) {
  console.log(arrFuncs[i]());
```

What's happening here?

_____ becomes a new copy of _____ for each loop iteration via the IIFE.

So _____ is captured uniquely for every _____ .

Closure in Callbacks!

It is common for a callback to reference a variable declared outside of itself.

```
function getCarsByMake(make) {
    return cars.filter(x => x.make === make);
}
```

Concept	What It Means
Lexical Scoping	Variables are accessible based on **where** functions are **written** (not called).
Scope Chaining	When looking for a variable, JavaScript checks the **current scope**, then the **outer scope**, and so on — forming a **chain**.
Closure	A function "remembers" variables from its outer scope, even after the outer function has returned.

make is available in the callback because of _____, (lexical scoping/scope chaining/closure) and the value of *make* is persisted when the anonymous function is called by _____ because of a _____ .

Write testing code for given f()

(1) Define a test cars array

(2) Call the function and check the output

```
function getCarsByMake(make) {
    return cars.filter(x => x.make === make);
}
```

UNIVERSITY OF KARACHI

closures return objects from f() that store state

Consider the following makePerson f() which returns an object that can store and change a name:

```
function makePerson(name) {
    let _name = name;

    return {
        setName: (newName) => (_name = newName),
        getName: () => _name,
    };
}

const me = makePerson("Strange");
console.log(me.getName()); // "Strange"

me.setName("Humera Tariq");
console.log(me.getName()); // "Humera Tariq"
```

"Closures do not _____ the values of variables from a function's _____(outer/inner) scope during creation.

Instead, they maintain_____throughout the closure's lifetime."

UNIVERSITY OF KARACHI

closures to implement private methods

JS closures return object state

```
function makePerson(name) {
    let _name = name;

    return {
        setName: (newName) => (_name = newName),
        getName: () => _name,
    };
}

const me = makePerson("Strange");
console.log(me.getName()); // "Strange"

me.setName("Humera Tariq");
console.log(me.getName()); // "Humera Tariq"
```

```
function makePerson(name) {
    let _name = name;

    function privateSetName(newName) {
        _name = newName;
    }

    return {
        setName: (newName) => privateSetName(newName),
        getName: () => _name,
    };
}
```

UNIVERSITY OF KARACHI

Closures + Event Handler + Callback

```html
<!-- index.html -->
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <title>Closure + Event Handler</title>
    <script>
      function setupButton(name) {
        const button = document.createElement('button');
        button.textContent = `Click me: ${name}`;
        button.addEventListener('click', function () {
          alert(`Hello, ${name}`);
        });
        document.body.appendChild(button);
      }

      window.onload = function () {
        setupButton('Humera');
      };
    </script>
  </head>
  <body></body>
</html>
```

```js
function setupButton(name) {
  const button = document.createElement('button');
  button.textContent = `Click me: ${name}`;

  // Event handler with closure
  button.addEventListener('click', function () {
    alert(`Hello, ${name}`);
  });

  document.body.appendChild(button);
}

setupButton('Humera');
```

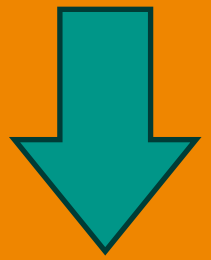
The setupButton function creates a variable (name) and attaches a callback (anonymous function) as an event handler.

When the button is clicked later, the callback remembers the value of name via a closure.

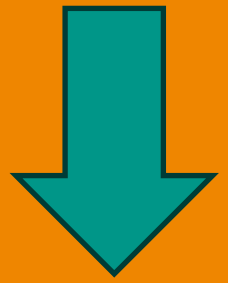
Even though **setupButton** has finished executing, the click event still has access to name.

UNIVERSITY OF KARACHI

Google Services



Firebase + Firestore



MongoDB

Week 10 Agenda/ Flow

Firebase

Top React Design Patterns

- ✓ **Example 1 :** Review Mini List Project (addProject, deleteProject, Array of Objects, importance of Backticks)
- ✓ **Example 2 :** Building and hosting a small React app with Fire base authentication and Fire store DB.
- ✓ **Example 3 :** Container-Presenter Pattern with clean and optimized fetch logic discussion. (fetch .then vs async await fetch debate).

UNIVERSITY OF KARACHI

Rimsha Masood (Sec A)

articleService.js

src > services > articleService.js > ...

```
1
2   // Rimsha Masood
3
4   const API_URL = "http://localhost:3001/articles"
5
6   export async function fetchArticles() {
7     const res = await fetch(API_URL)
8     const data = await res.json()
9     return data
10  }
11
12  export async function createArticle({ title, body }) {
13    const res = await fetch(API_URL, {
14      method: "POST",
15      headers: {
16        "Content-Type": "application/json",
17      },
18      body: JSON.stringify({ title, body }),
19    })
20    const data = await res.json()
21    return data
22  }
```

backend > server.js > app.post("/articles") callback > article

```
1   const express = require("express")
2   const mongoose = require("mongoose")
3   const cors = require("cors")
4
5   const app = express()
6   app.use(cors())
7   app.use(express.json())
8
9   mongoose.connect("mongodb://localhost:27017/blogApp")
10
11  const Article = mongoose.model("Article", {
12    title: String,
13    body: String,
14    date: { type: Date, default: Date.now },
15  })
16
17  app.get("/articles", async (req, res) => {
18    const articles = await Article.find().sort({ date: -1 })
19    res.json(articles)
20  })
21
22  app.post("/articles", async (req, res) => {
23    const { title, body } = req.body
24    const article = new Article({ title, body })
25    await article.save()
26    res.json(article)
27  })
28
29  app.listen(3001, () => {
30    console.log("Server is running on http://localhost:3001")
31  })
```

Urooj Shahab (Sec B)

Urooj shahab B21110006161 4/9/2025 4:32 PM

@Dr. Humera Tariq @Nabeel Khan

drivelink: https://drive.google.com/drive/folders/1RlDZSrcXlgf7N8_dWisUivuYvOgkbmjO

code drivelink: https://drive.google.com/drive/folders/1-IHjdIpr2bMqLpzaXrBvpywZ0KzFMdyq

I have successfully completed my assignment, which focused on implementing CRUD operations in my React application, supported by an Express.js backend and MongoDB for data storage.

The POST /articles endpoint facilitates the creation of new articles by accepting a title and body, which are then stored in the MongoDB database.

The PUT /articles/:id endpoint allows for updating existing articles by modifying their title and body fields.

The DELETE /articles/:id endpoint handles the deletion of articles from the database.

The fetchArticles() function retrieves and displays the list of articles, while the addArticle() function sends new article data to the backend for creation. The updateArticle() function enables editing of existing articles, and the deleteArticle() function manages article removal.

This task helped me better understand how to handle data with RESTful APIs and integrate the frontend with a backend server using MongoDB for data persistence. Thank you for your guidance throughout the class. (edited)

Google Drive

Google Drive

✅ 1 ✴️ 1

Urooj Shahab (Sec B)

🔁 Backend Integration Created RESTful API using Express.js:

Built endpoints:

GET /articles – fetches all articles

POST /articles – creates a new article

PUT /articles/:id – updates an existing article

DELETE /articles/:id – deletes an article

Connected to MongoDB to store articles persistently.

⚛️ Frontend Changes (React) Replaced Static Data with Fetch Requests:

Used useEffect() to load all articles from the backend using GET /articles.

Added Article Creation Logic:

Implemented addArticle() function to handle the form submission and call POST /articles.

Enabled Article Updates:

Added updateArticle() function to update article content using PUT /articles/:id.

Enabled Article Deletion:

Added deleteArticle() function to remove articles via DELETE /articles/:id.

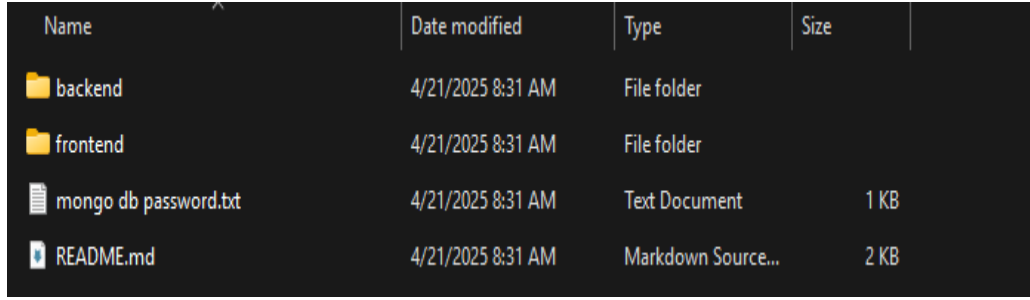
Controlled View Switching:

Used state like writing and editingArticle to toggle between article form and detail view.

✅ Result The final app allows users to:

- ✓ View all blog posts
- ✓ Create new articles
- ✓ Edit existing ones
- ✓ Delete articles

Urooj Shahab Front end (ArticleService.js)

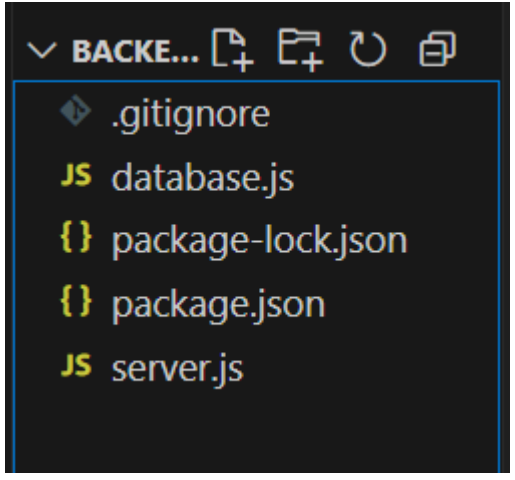


Name	Date modified	Type	Size
backend	4/21/2025 8:31 AM	File folder	
frontend	4/21/2025 8:31 AM	File folder	
mongo db password.txt	4/21/2025 8:31 AM	Text Document	1 KB
README.md	4/21/2025 8:31 AM	Markdown Source...	2 KB

```
const BASE_URL = "http://localhost:5000/articles"

export async function fetchArticles() {
  const res = await fetch(BASE_URL)
  return await res.json()
}

export async function createArticle(article) {
  const res = await fetch(BASE_URL, {
    method: "POST",
    headers: { "Content-Type": "application/json" },
    body: JSON.stringify(article),
  })
  return await res.json()
}
```

```
export async function updateArticle(id, article) {
  const res = await fetch(`${BASE_URL}/${id}`, {
    method: "PUT",
    headers: { "Content-Type": "application/json" },
    body: JSON.stringify(article),
  })
  return await res.json()
}

export async function deleteArticle(id) {
  await fetch(`${BASE_URL}/${id}`, { method: "DELETE" })
}
```

Urooj Shahab Back end (database.js)



database.js > ...

```
1   // database.js
2   import mongoose from "mongoose";
3   import dotenv from "dotenv";
4
5   dotenv.config();
6
7   const connectDB = async () => {
8     try {
9       await mongoose.connect(process.env.MONGODB_URI, {
10        useNewUrlParser: true,
11        useUnifiedTopology: true,
12      });
13      console.log("MongoDB connected");
14    } catch (err) {
15      console.error("MongoDB connection error:", err);
16    }
17  };
18
19  export default connectDB;
20
```

Urooj Shahab Back end (server.js)

```
connectDB();

// CREATE
app.post("/articles", async (req, res) => {
  const article = { ...req.body, date: new Date() };
  const result = await collection.insertOne(article);
  res.json({ _id: result.insertedId, ...article });
});
```

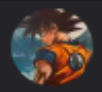
```
// READ
app.get("/articles", async (req, res) => {
  const articles = await collection.find().sort({ date: -1 }).toArray();
  res.json(articles);
});
```

Urooj Shahab Back end (server.js)

```
// UPDATE
app.put("/articles/:id", async (req, res) => {
  const { id } = req.params;
  const updatedData = req.body;
  await collection.updateOne(
    { _id: new ObjectId(id) },
    { $set: updatedData }
  );
  const updatedArticle = await collection.findOne({ _id: new ObjectId(id) });
  res.json(updatedArticle);
});
```

```
// DELETE
app.delete("/articles/:id", async (req, res) => {
  const { id } = req.params;
  await collection.deleteOne({ _id: new ObjectId(id) });
  res.json({ success: true });
});
```

```
const PORT = 5000;
app.listen(PORT, () => console.log(`Server running on port ${PORT}`));
```



Hassan Ahmed Khan#B21110006041 4/9/2025 12:36 AM

I have successfully completed today's class task, which involved replacing Firebase Firestore with MongoDB. For this, I implemented a backend using Express.js and created RESTful APIs for handling article data.

BSCS 633 Samurai 2025!

"A true samurai is not defined by his sword, but by his spirit."

Today, I honor the first Samurai among you — the one who:

1. Accepted the challenge without hesitation.
2. Honored the words of the teacher with action — not just agreement.
3. Demonstrated real web development skill – building his own backend, replacing Google services with his own MongoDB-powered solution.

Well done, Hassan Ahmed Khan
B21110006041

For Server Side

```
1) Open "server" folder in the terminal
2) add your own MongoDb URI in .env folder or use
this (MongoDB_URI=
"mongodb+srv://hassanahmedkhan:EvBoc76BonnLlddG@clus
ter0.kcspxm6.mongodb.net/")
3) Run "npm install" command
4) Run "npm run dev"
```

For Client Side

1) Open "blog-starter-bscs-633-main" folder in the terminal
2) Run "npm install" command
3) Run "npm run dev"

Week06_BSCS-633_RestAPI_MVC.pdf

Week07_BSCS-633_http_communication.pdf

https://cloud.mongodb.com/v2/67c2c6cd20af433eb65f19bb#/connect/HT-Cluster0

1. Select driver

Node.js

2. Install your driver

Run the following on the command line

```
npm install mongodb
```

View MongoDB Node.js Driver installation instructions.

3. Add connection string into your application code.

Get legacy (standard) connection string

String | Sample Code

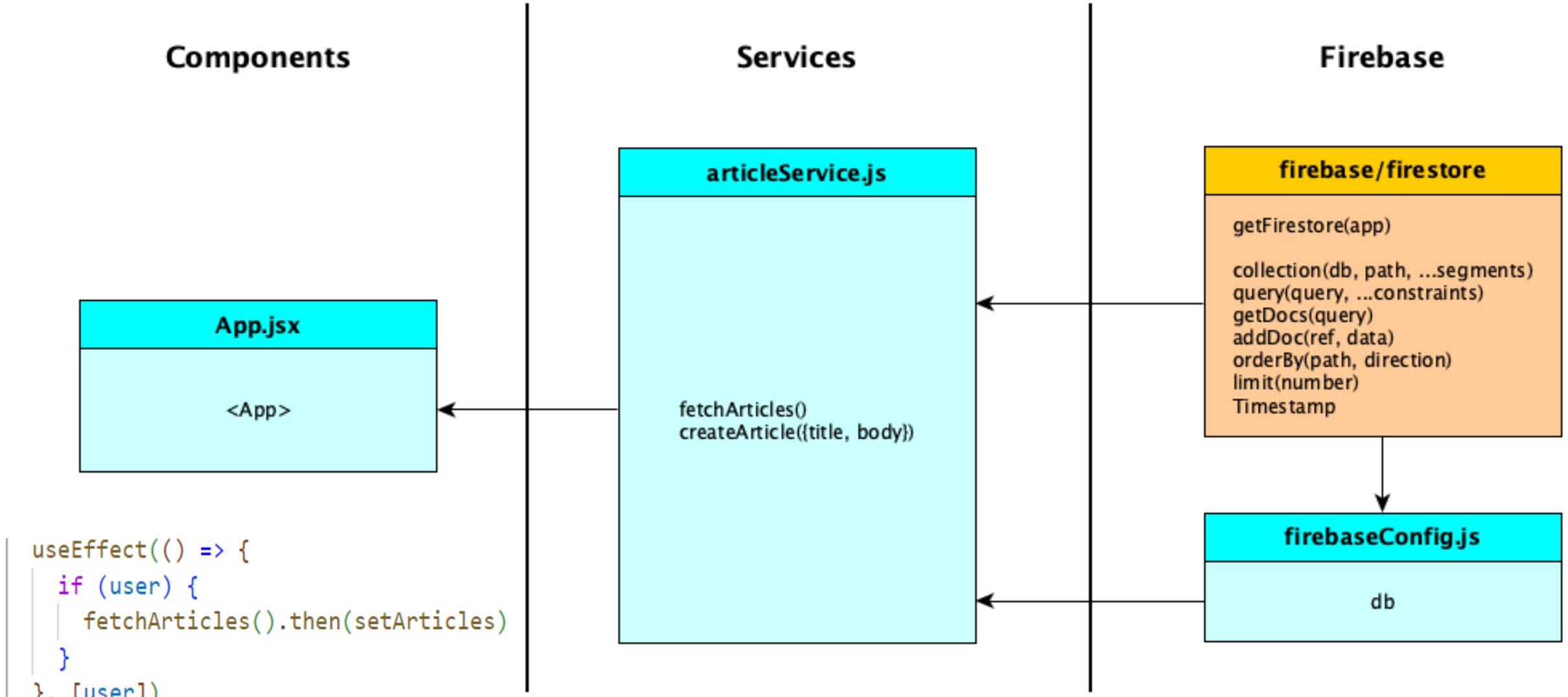
```
mongodb+srv://humera:<db_password>@ht-cluster0.fveay.mongodb.net/?retryWrites=true&w=majority&appName=HT-Cluster0
```

Replace **<db_password>** with the password for the **humera** user. Ensure any option params are URL encoded.

We created cluster in Week 06-07

```
import { fetchArticles, createArticle } from "../services/articleService"
```

Original Code base




```
useEffect(() => {
  if (user) {
    fetchArticles().then(setArticles)
  }
}, [user])
```

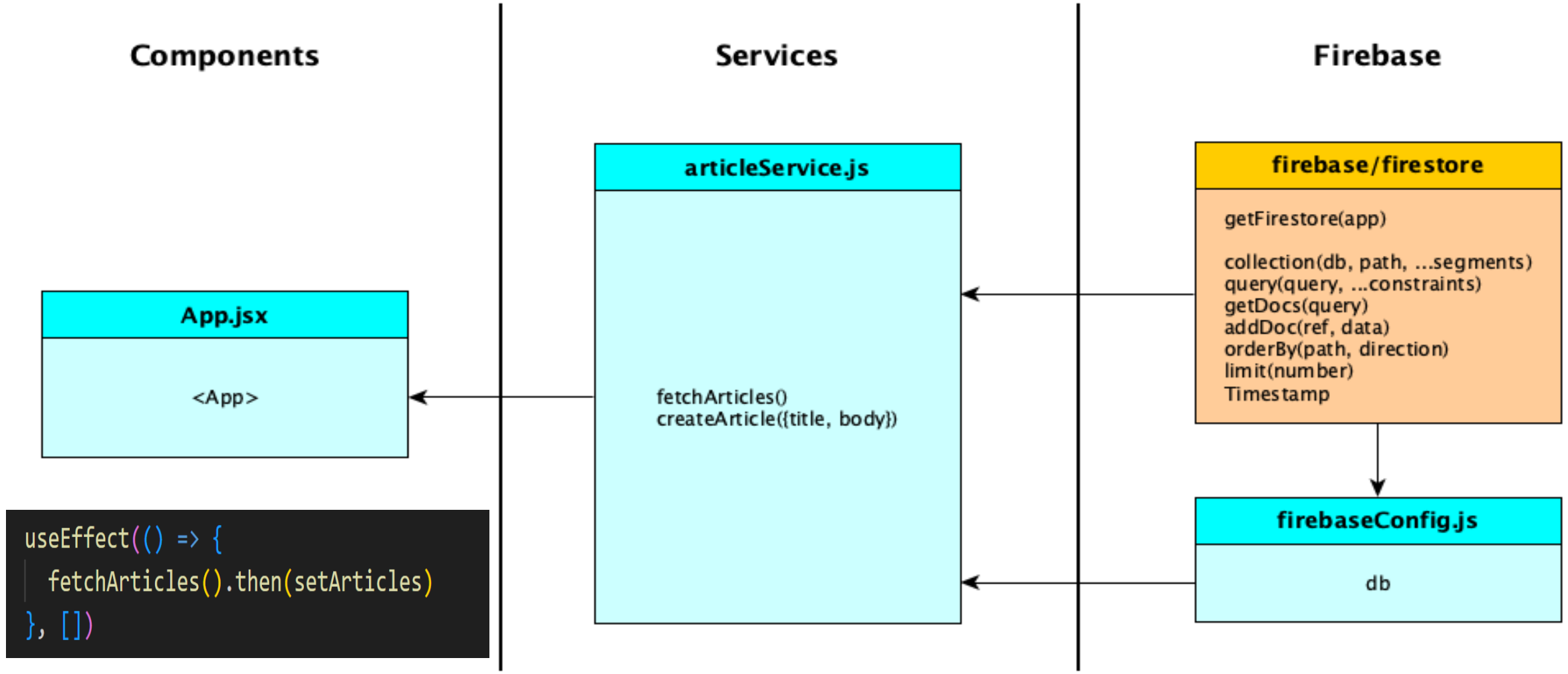
Original Code base provide 2 services

```
export async function fetchArticles()
{…….}

export async function createArticle({ title, body })
{……..}
```

MongoDB updates

Let's Analyze `fetchArticles()` first :

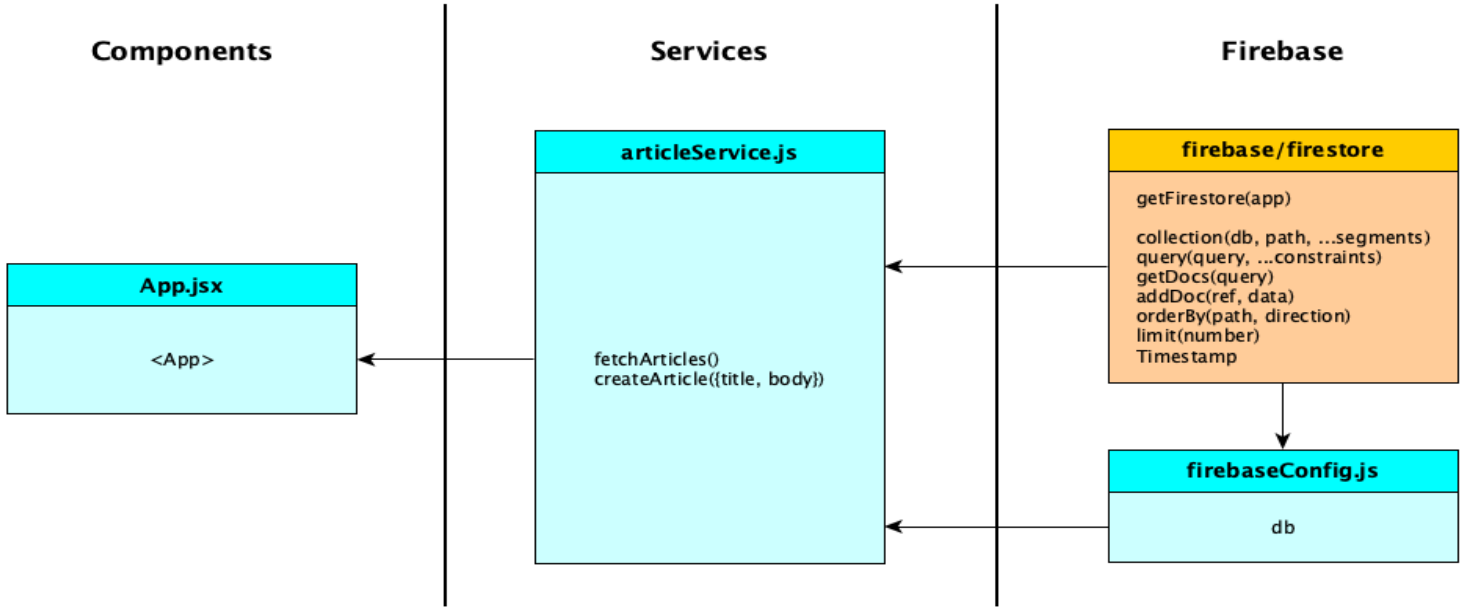
fetchArticles()	Constitutes HTTP Request
GET (Safe & Idempotent)	Retrieve a resource from the server. fetch("http://localhost:3000/articles")




```
useEffect(() => {
  fetchArticles().then(setArticles)
}, [])
```

```
export async function fetchArticles() {

  const response = await fetch("http://localhost:3000/articles");
  const articles = await response.json();
  // In storage the ids are separated from the data, but in this function
  // we are going to combine the id and the data together.
  return ??
}
```

```
useEffect(() => {
  ??
}, [])
```



ArticleEntry.jsx

Title

Body

Create

Frond end request URL

Express server

Router

Controller

Model

MongoDB

Query

```
try{
    const article = await Article.find();
    res.status(200).json(article);
} catch(err){
    res.status(404).json({errors: err.message});
}
```

MongoDB updates

Now Let's Analyze `createArticles({ title, body })`

`{.....}`

Starter code

App.jsx | articleService.js

```
export async function createArticle({ title, body }) {
  // As this is just fake data for messing around, we'll throw in a quick
  // and unreliable database id. In a real app, the id should be generated
  // by the database itself (or you can use UUIDs).
  return { id: Math.random(), title, body, date: new Date() }
}
```

Firebase

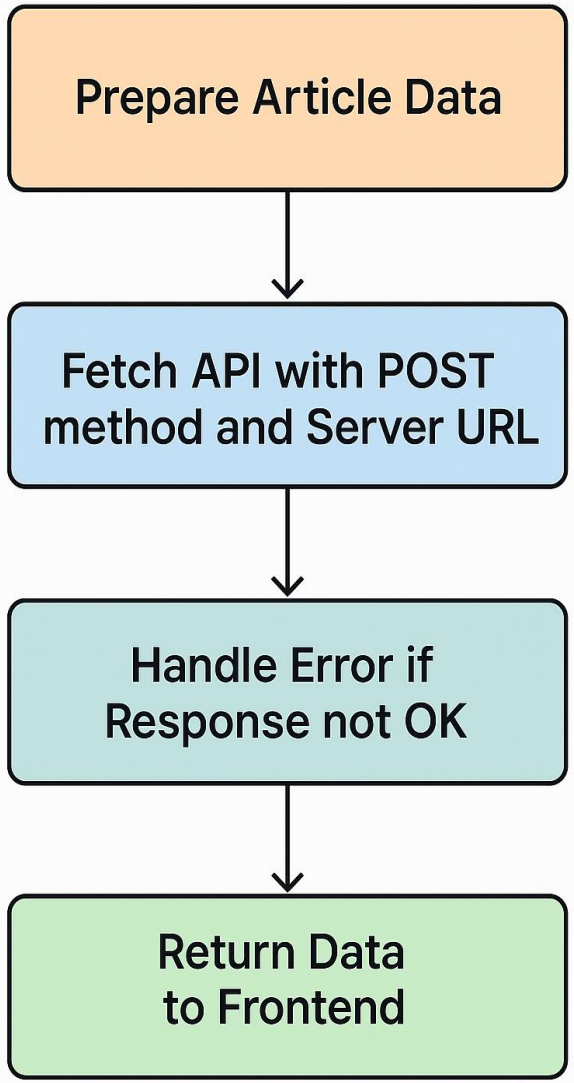
```
export async function createArticle({ title, body }) {
  const data = { title, body, date: Timestamp.now() }
  const docRef = await addDoc(collection(db, "articles"), data)
  return { id: docRef.id, ...data }
}
```

When a user submits a new article, the frontend sends a POST request to a RESTful API endpoint handled by the backend in server.js

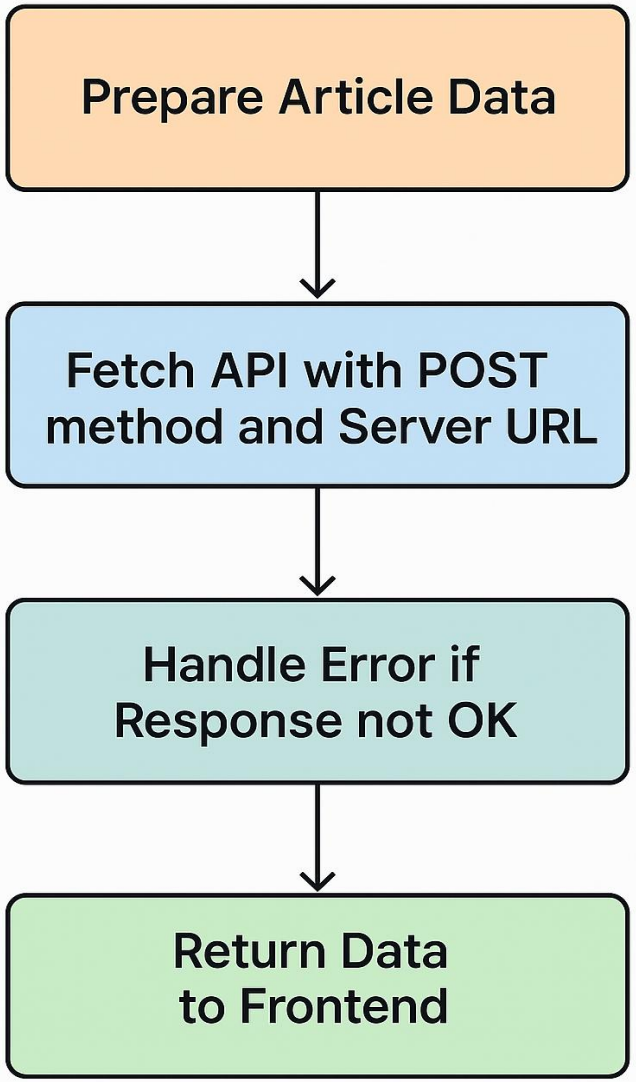
```
export async function createArticle({ title, body }){
                    // 4-piece breakdown
}
```

See next slide for create details

```
export async function createArticle({ title, body }) {
  const article = {
    _id: Math.random(),
    date: new Date(),
    title,
    body,
  }

  const response = await fetch("http://localhost:3000/articles/create-article", {
    method: "POST",
    headers: {
      "Content-Type": "application/json",
    },
    body: JSON.stringify(article),
  })

  if (!response.ok) {
    const error = await response.json()
    throw new Error(error.message || "Failed to create article")
  }

  const data = await response.json()
  return data
}
```



4-piece breakdown




```
export async function createArticle({ title, body }){
                    // 4-piece breakdown
}
```

```
const response = await fetch
("http://localhost:3000/articles/create-article",
    {
        method: _____.
        headers: {
                _____,
        },
        body: _____,
    }
)
```

Revision slide

props passing and lift state up

ArticleEntry.jsx

Render

```
return (
    <div>
      <form onSubmit={submit}>
          ….
     </div> )
```

Frond end request URL

```
export default function ArticleEntry({ addArticle })
{
  const [title, setTitle] = useState("")
  const [body, setBody] = useState("")
  const [error, setError] = useState(null)

  function submit(e) {……..}
```

Props Passing

```
function submit(e) {
    setError(null)
    e.preventDefault()

    if (!title.trim() || !body.trim()) {
      setError("Both the title and body must be supplied")
    } else {
      addArticle({title,body})
    }
  }
```

```
function addArticle({ title, body }) {
  createArticle({ title, body }).then((article) => {
    setArticle(article)
    setArticles([article, ...articles])
    setWriting(false)
  })
}
```

App.jsx

Lift State Up

```
export async function createArticle({ title, body})

{……const response = await

    fetch("http://localhost:3000/articles/create-article", { ……})
……}
```

articleService.jsx

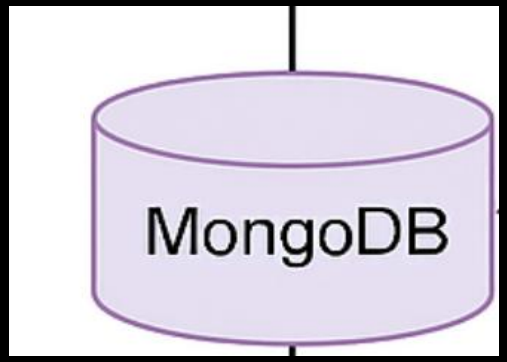
Express server

Router

Controller

Model

Server-side Schema



ArticleEntry.jsx

Title

Body

Create

```
Server Listen on 3000
 Connected to Cloud MongoDB Atlas
OPTIONS /articles/create-article 204 2.125 ms - 0
{
  _id: 0.549532692307346,
  date: '2025-04-13T23:41:39.295Z',
  title: '14042025',
  body: 'Hot day'
}
POST /articles/create-article 201 134.321 ms - 105
```

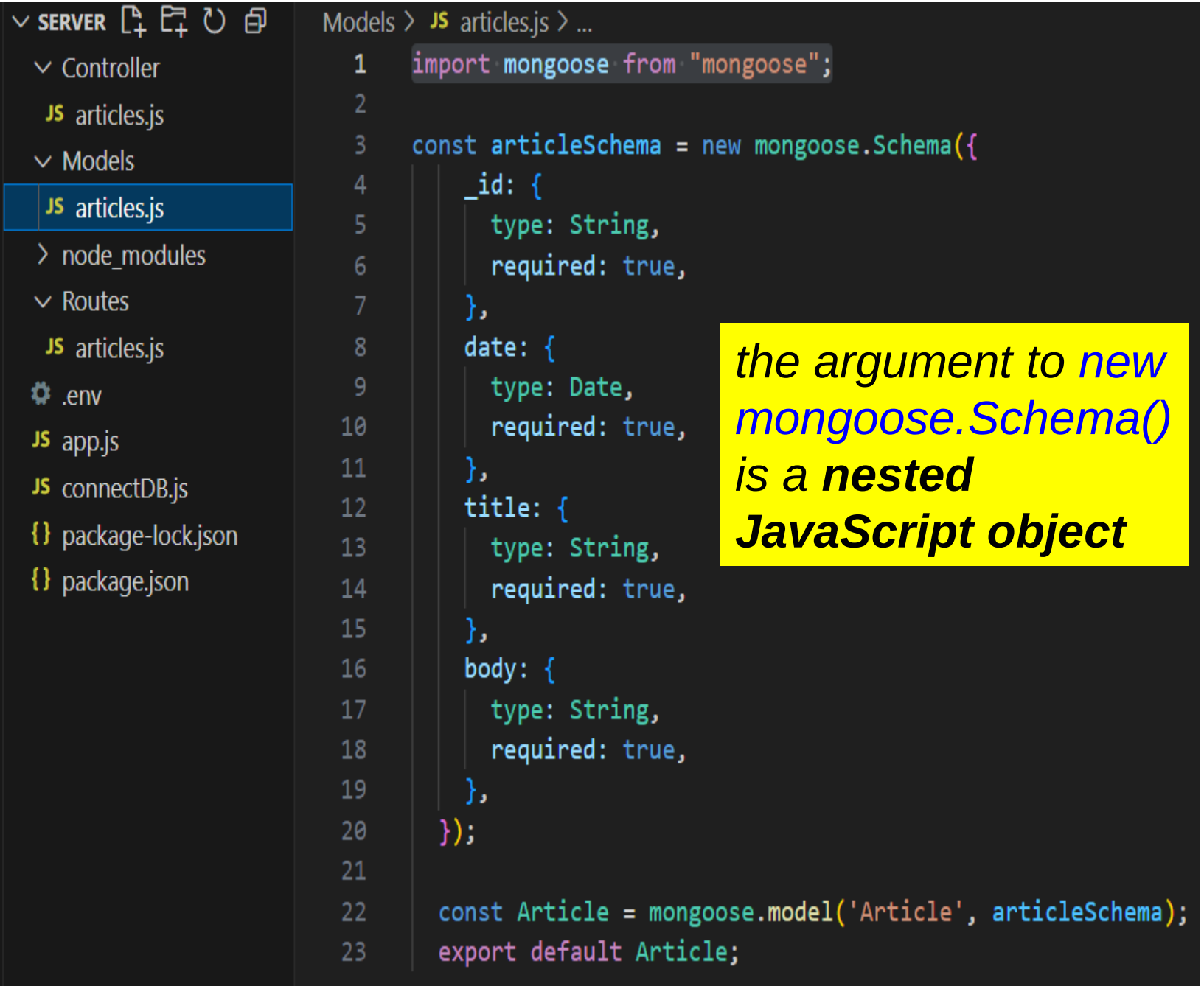
SERVER

- Controller
  - articles.js
- Models
  - articles.js
- node_modules
- Routes
  - articles.js
- .env
- app.js
- connectDB.js
- package-lock.json
- package.json

app.js

```
1  import express from "express";
2  import dotenv from "dotenv";
3  import mongoose from "mongoose";
4  import cors from "cors";
5  import morgan from "morgan";
6  import helmet from "helmet";
7  import path from "path";
8  import { fileURLToPath } from "url";
9  import Article from "./Models/articles.js";
10 import articleRoute from "./Routes/articles.js";
11
12 import connectDB from "./connectDB.js";
13
14
```

Schema



*the argument to new mongoose.Schema() is a **nested JavaScript object***

mongoose.Schema({...}): Creates a blueprint for what a MongoDB document in this collection should look like.

Each field (_id, date, title, body) is defined with:

type: The JavaScript type that field must be (String, Date, etc.)

required: true: A validation rule that enforces this field must be provided when creating a document.

app.js (app.use vs. app.get)

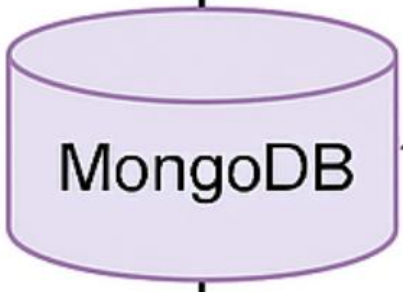
```
app.use("/articles",articleRoute);
app.get("/",(req,res) =>{
    res.status(200).json("Hassan Ahmed Khan B21110006041")
})
```

```
connectDB();
```

```
const app = express();
const PORT = process.env.PORT || 6001;
```

.env

MongoDB_URI
PORT=3000



connectDB.js > connectDB

```
1   import mongoose from "mongoose";
2
3   const connectDB = async () => {
4       try{
5           await mongoose.connect(process.env.MongoDB_URI,{
6               useNewUrlParser:true,
7               useUnifiedTopology:true,
8           });
9
10          // Determine Cloud or Local DB
11      if (process.env.MongoDB_URI.includes("mongodb+srv://")) {
12          console.log(" Connected to Cloud MongoDB Atlas");
13      } else {
14          console.log(" Connected to Local MongoDB");
15      }
16
17
18      } catch(err){
19          console.log(`Data Base Connection Error: ${err.message}`)
20          process.exit(1);
21      }
22  }
```

Server description with emphasize on Routing Logic

```
const response = await fetch("http://localhost:3000/articles");
```

```
const response = await fetch("http://localhost:3000/articles/create-article",
```

Front end Request Listen by server

```
const PORT = process.env.PORT || 6001;
connectDB();
app.listen(PORT,()=>{
    console.log(`Server Listen on ${PORT}`)
    // Article.insertMany(dummyArticles)
});
```

Delegate to router

```
app.use("/articles",articleRoute);
```

```
const router = express.Router();
router.get("/",getArticle);
router.post("/create-article", createArticle);
```

Frond end request URL

```
fetch("http://localhost:3000/articles") // Defaults to GET method
```

Express server

```
app.use("/articles",articleRoute);
```

- Strips the mounted path (/article)

- Passes the remaining portion to the router

- "/create-article"

Router

```
import articleRoute from "./Routes/articles.js";
```

```
router.get("/",getArticle);
router.post("/create-article", createArticle);
```

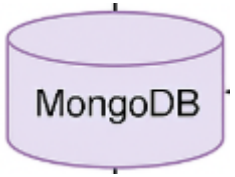
Handles data and sends a response

Model → Controller

```
import { getArticle, createArticle } from "../Controller/articles.js";
```

```
const Article = mongoose.model('Article', articleSchema);
export default Article;
```

```
const articleSchema = new mongoose.Schema({
    _id: {
      type: String,
      required: true,
    },
    date: {
      type: Date,
      required: true,
    },
    title: {
      type: String,
      required: true,
    },
    body: {
      type: String,
      required: true,
    },
  });
```




```
import Article from "../Models/articles.js";
export const getArticle = async (req,res) => {
    try{
        const article = await Article.find();
        res.status(200).json(article);
    } catch(err){
        res.status(404).json({errors: err.message});
    }
};
```

Frond end request URL

```
const response = await fetch("http://localhost:3000/articles/create-article",
```

Express server

```
app.use("/articles",articleRoute);
```

- Strips the mounted path (/article)

- Passes the remaining portion to the router

- Empty " " path string normalize to "/"

Router

```
import articleRoute from "./Routes/articles.js";
const router = express.Router();
router.get("/",getArticle);
```

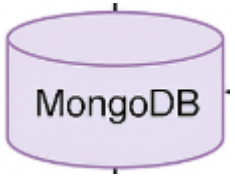
Handles data and sends a response

Model → *Controller*

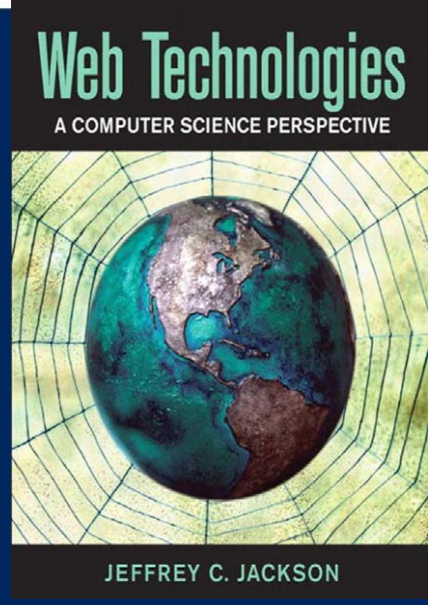
```
import { getArticle, createArticle } from "../Controller/articles.js";
```

```
const Article = mongoose.model('Article', articleSchema);
export default Article;
```

```
const articleSchema = new mongoose.Schema({
    _id: {
      type: String,
      required: true,
    },
    date: {
      type: Date,
      required: true,
    },
    title: {
      type: String,
      required: true,
    },
    body: {
      type: String,
      required: true,
    },
  });
```




```
export const createArticle = async (req, res) => {
    console.log(req.body);
  const { _id, date, title, body } = req.body;

  try {
    const newArticle = new Article({ _id, date, title, body });
    await newArticle.save();
    res.status(201).json(newArticle);
  } catch (err) {
    res.status(400).json({ error: err.message });
  }
};
```



Connected myself back to JJ Chapter 1

Request-Response

TABLE 1.1 Some Non-`http` URL Schemes

Scheme Name	Example URL	Type of Resource
`ftp`	`ftp://ftp.example.org/pub/afile.txt`	File located on FTP server
`telnet`	`telnet://host.example.org/`	Telnet server
`mailto`	`mailto:someone@example.org`	Mailbox
`https`	`https://secure.example.org/sec.txt`	Resource on web server supporting encrypted communication
`file`	`file:///C:/temp/localFile.txt`	File accessible from machine processing this URL

TABLE 1.2 Standard HTTP/1.1 Methods

Method	Requests server to . . .
GET	return the resource specified by the Request-URI as the body of a response message.
POST	pass the body of this request message on as data to be processed by the resource specified by the Request-URI.
HEAD	return the same HTTP header fields that would be returned if a GET method were used, but not return the message body that would be returned to a GET (this provides information about a resource without the communication overhead of transmitting the body of the response, which may be quite large).
OPTIONS	return (in Allow header field) a list of HTTP methods that may be used to access the resource specified by the Request-URI.
PUT	store the body of this message on the server and assign the specified Request-URI to the data stored so that future GET request messages containing this Request-URI will receive this data in their response messages.
DELETE	respond to future HTTP request messages that contain the specified Request-URI with a response indicating that there is no resource associated with this Request-URI.
TRACE	return a copy of the complete HTTP request message, including start line, header fields, and body, received by the server. Used primarily for test purposes.

1.4.5 Header Fields and MIME Types

The Host header field is required in every HTTP/1.1 request message.

Each header field begins with a field name, such as Host, followed by a colon and then a field value. White space is allowed to precede or follow the field value, but such white space is not considered part of the value itself.

```
const response = await fetch("http://localhost:3000/articles/create-article", {
  method: "POST",
  headers: {
    "Content-Type": "application/json",
  },
  body: JSON.stringify(article),
})
```

Why ? Alternatives ?

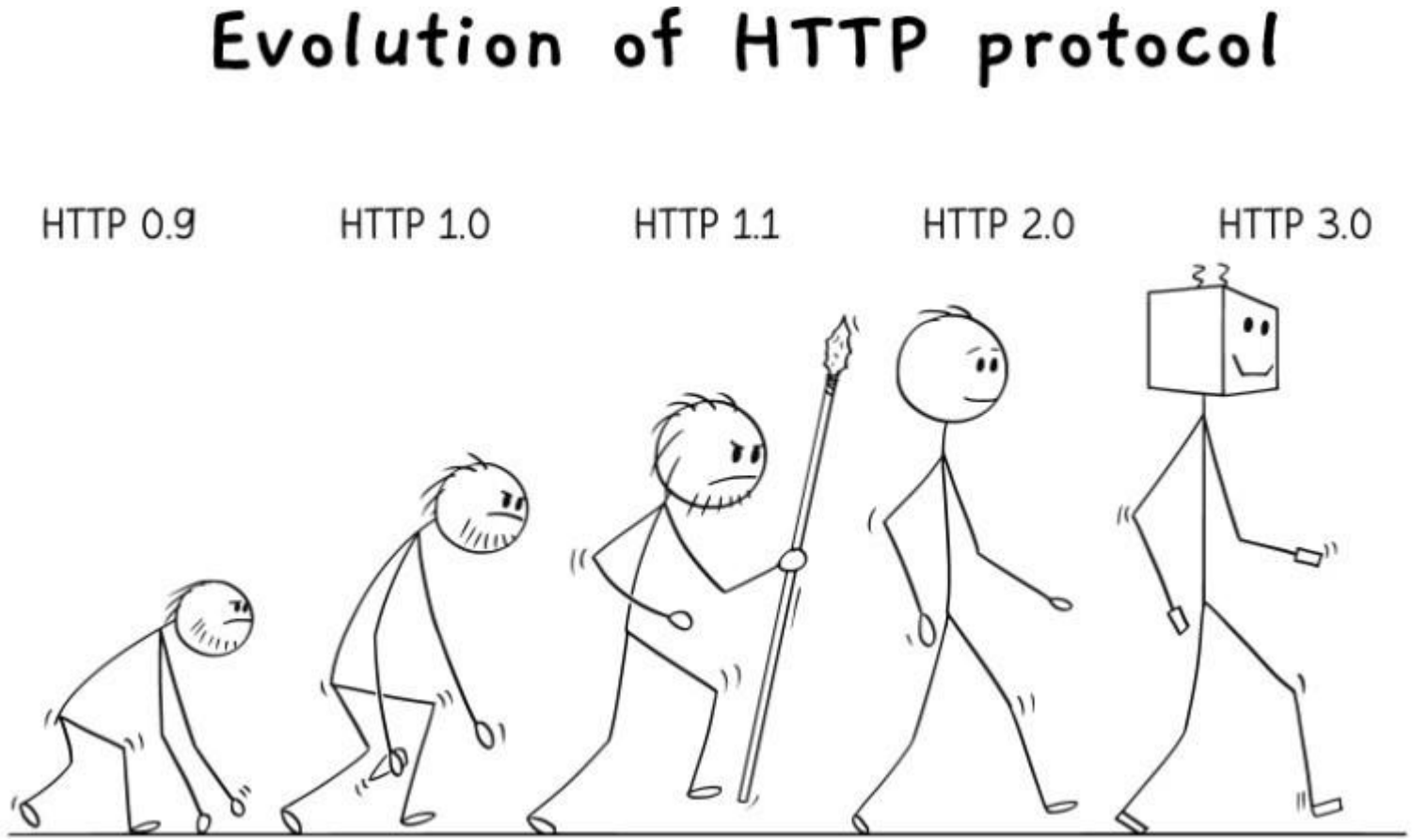
TABLE 1.3 Standard Top-level MIME Content Types

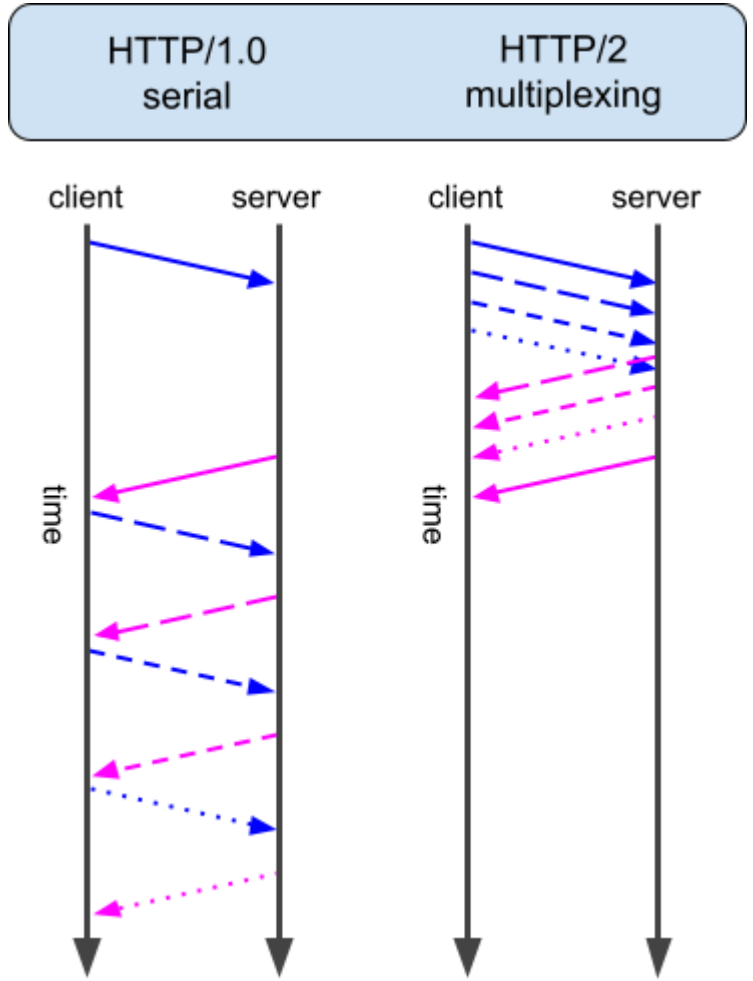
Top-level Content Type	Document Content
application	Data that does not fit within another content type and that is intended to be processed by application software, or that is itself an executable binary.
audio	Audio data. Subtype defines audio format.
image	Image data, typically static. Subtype defines image format. Requires appropriate software and hardware in order to be displayed.
message	Another document that represents a MIME-style message. For example, following an HTTP TRACE request message to a server, the server sends a response with a body that is a copy of the HTTP request. The value of the Content-Type header field in the response is message/http.
model	Structured data, generally numeric, representing physical or behavioral models.
multipart	Multiple entities, each with its own header and body.
text	Displayable as text. That is, a human can read this document without the need for special software, although it may be easier to read with the assistance of other software.
video	Animated images, possibly with synchronized sound.

TABLE 1.4 Some Common MIME Content Types

MIME Type	Description
text/html	HTML document
image/gif	Image represented using Graphics Interchange Format (GIF)
image/jpeg	Image represented using Joint Picture Expert Group (JPEG) format
text/plain	Human-readable text with no embedded formatting information
application/octet-stream	Arbitrary binary data (may be executable)
application/x-www-form-urlencoded	Data sent from a web form to a web server for processing

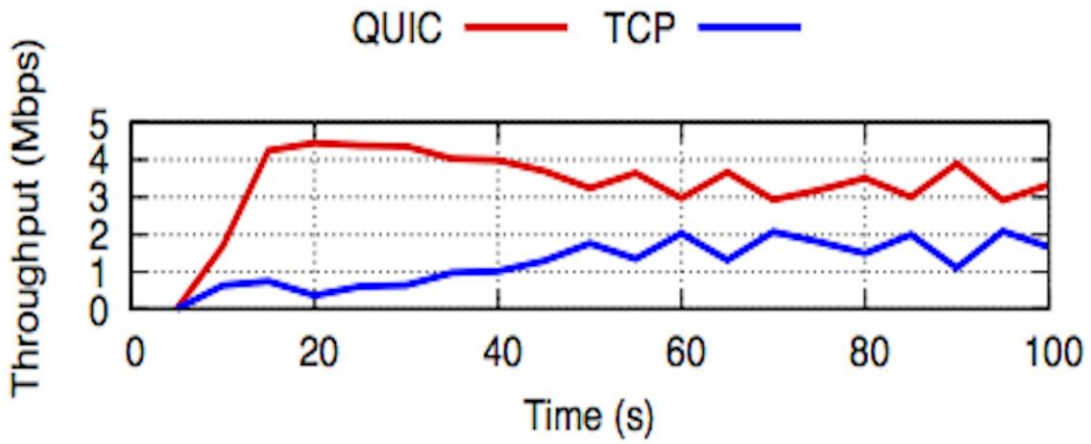
Is modern web request still HTTP/1.1?
Yes, **HTTP/1.1 is still widely used**, but there are newer versions too:
So even if your frontend is using HTTP/1.1, **modern browsers and Node.js may auto-upgrade** to HTTP/2/3 if both client and server support it.

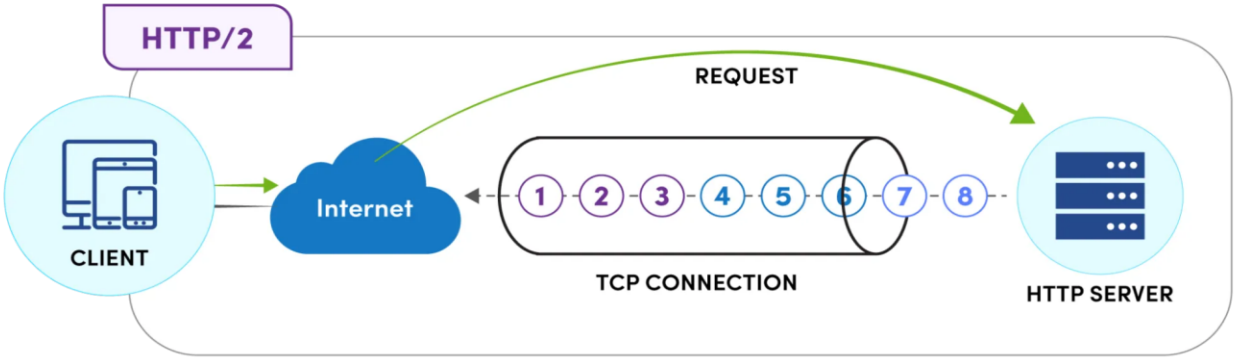


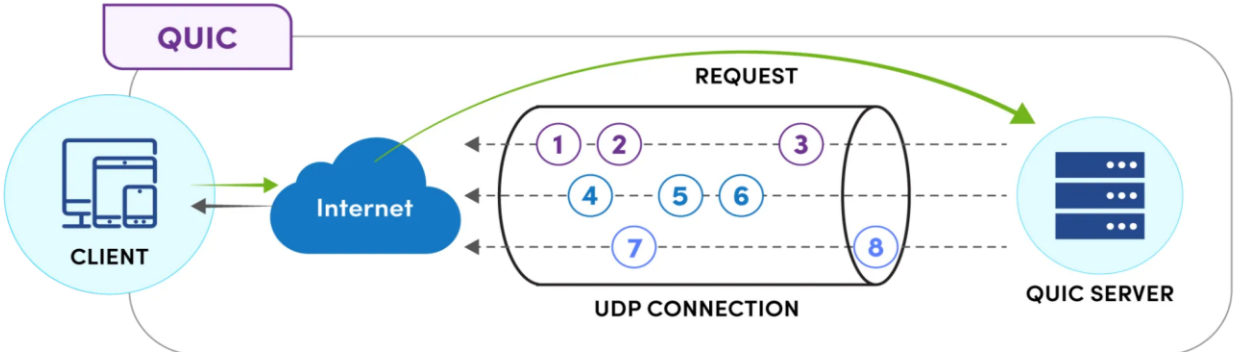


Learn the differences between HTTP/1.1 and HTTP/2 - Neomind

GET /about.html Host: www.example.com	10010 01101 1100 00001 001 001 110 101	10010 01101 1100 00001 001 001 110 101
HTTP/1.1	HTTP/2	HTTP/3
TLS (optional)	TLS	QUIC (+TLS)
TCP	TCP	UPD
IP	IP	IP
Web browser makes several parallel requests for page contents: HTML, images, style, JS	Web browser makes one TCP connection with requests for all page contents in HTTP/2 streams (binary)	Web browser makes one QUIC connection with requests for all page contents in QUIC streams (binary)

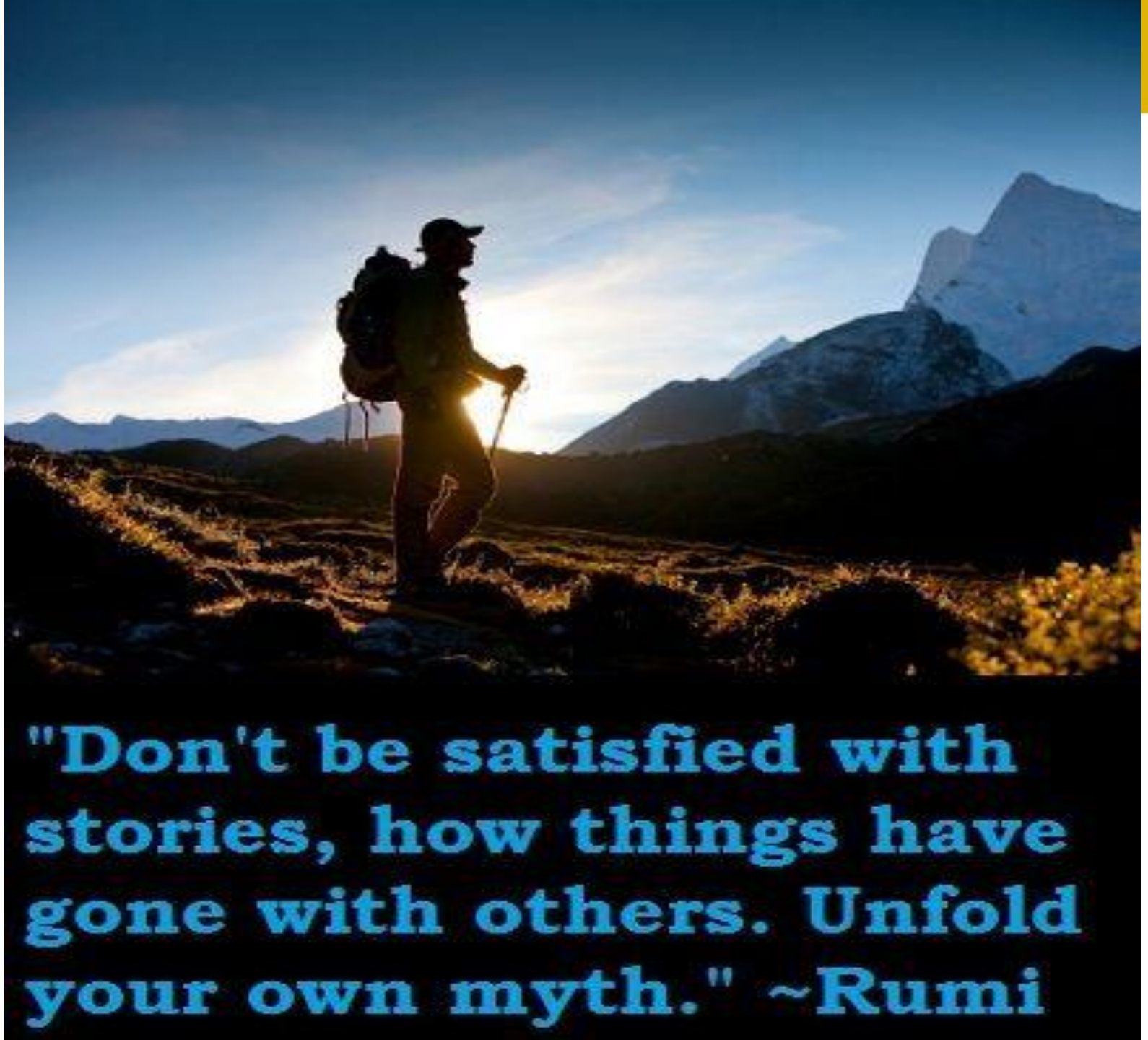






- ✓ **Single connection and multiplexing**
- ✓ **Binary framing layer**
- ✓ **Request Prioritization** (the browser will prioritize CSS files first, even if its request in DOM order comes after the JS.
- ✓ **Server Push** (multiple responses sent)
- ✓ **Automatic compression** (HPACK algorithm)
- ✓ **Data security and encryption** (SSL certificate)

UNIVERSITY OF
KARACHI



UNIVERSITY OF KARACHI

Department of Compute Science (UBIT Building), Karachi, Pakistan.

1200 Acres (5.2 Km sq.)
53 Departments
19 Institutes
25000 Students

UNIVERSITY OF KARACHI

My Homeland Pakistan