Fsd

1……>

```
Const fs = require('fs');

Function isKaprekarNumber(number) {
  If (number === 1) return true;
  Const square = number * number;
  Const squareStr = square.toString();
  For (let I = 1; I < squareStr.length; i++) {
    Const leftPart = parseInt(squareStr.slice(0, i));
    Const rightPart = parseInt(squareStr.slice(i));
    If (leftPart + rightPart === number && leftPart !== 0 && rightPart !== 0) {
      Return true;
    }
  }
  Return false;
}

Function findKaprekarNumbers(start, end) {
  Const kaprekarNumbers = [];
  For (let I = start; I <= end; i++) {
    If (isKaprekarNumber(i)) {
      kaprekarNumbers.push(i);
    }
  }
  Return kaprekarNumbers;
}

Const start = 1;
Const end = 1000;


Const kaprekarNumbers = findKaprekarNumbers(start, end);


Fs.writeFileSync(‘kaprekar_numbers.txt’, kaprekarNumbers.join(‘\n’), ‘utf-8’);


Console.log(`Kaprekar numbers between ${start} and ${end} have been written to
kaprekar_numbers.txt`);
```

```
Node kaprekar.js
```

2…..>

```
Const fs = require(‘fs’);


Const sourceFile = ‘source.txt’;

Const destinationFile = ‘destination.txt’;


// Read the content from source.txt

Fs.readFile(sourceFile, ‘utf-8’, (err, data) => {

  If (err) {

    Console.error(`Error reading ${sourceFile}: ${err}`);

    Return;

  }


  // Write the content to destination.txt

  Fs.writeFile(destinationFile, data, ‘utf-8’, (err) => {
    If (err) {
      Console.error(`Error writing to ${destinationFile}: ${err}`);
    } else {
      Console.log(`Content from ${sourceFile} copied to ${destinationFile} successfully.`);
    }
  });
});
```

```
Node copyfile.js
```

3…..?

```
Const express = require('express');
Const cookieParser = require('cookie-parser');

Const app = express();
Const port = 3000;

App.use(express.urlencoded({ extended: true }));
App.use(cookieParser());

// Serve HTML and CSS files from a public directory
App.use(express.static('public'));

// Render the signup form
App.get('/', (req, res) => {
  Res.sendFile(__dirname + '/public/signup.html');
```

```
});


// Handle form submission
App.post('/submit', (req, res) => {
  Const { name, contactNumber, email, address, gender, dob } = req.body;


  // Store user information in a cookie with a 15-second expiration time
  Res.cookie('registered', JSON.stringify({ name, contactNumber, email, address, gender, dob }), {
maxAge: 15000 });


  // Render a confirmation message
  Res.send('Thank you for registering! <a href="/details">View Details</a>');
});


// Display user details from the cookie
App.get('/details', (req, res) => {
  Const userData = req.cookies.registered;


  If (!userData) {
    Return res.send('No user data found. <a href="/">Go Back</a>');
  }


  Const user = JSON.parse(userData);


  // Render user details and a logout link
  Res.send(`
    <h2>User Details:</h2>
    <p>Name: ${user.name}</p>
    <p>Contact Number: ${user.contactNumber}</p>
    <p>Email: ${user.email}</p>
    <p>Address: ${user.address}</p>
    <p>Gender: ${user.gender}</p>
    <p>DOB: ${user.dob}</p>
    <a href="/logout">Logout</a>
  `);
});

// Logout by clearing the registered cookie
App.get('/logout', (req, res) => {
  Res.clearCookie('registered');
  Res.redirect('/');
});

App.listen(port, () => {
  Console.log(`Server is running on port ${port}`);
});
```

```
Npm install express cookie-parser
Node app.js
```

4….>

```
Mkdir express-student-form
Cd express-student-form
Npm init -y
Npm install express pug

Const express = require('express');
Const app = express();
Const port = 3000;

// Set up Pug as the view engine
App.set('view engine', 'pug');
App.set('views', __dirname + '/views');

App.use(express.urlencoded({ extended: true }));

// Serve static files (e.g., stylesheets)
App.use(express.static('public'));

// Define a route to display the student form
App.get('/', (req, res) => {
  Res.render('student-form');
});

// Handle form submission and display submitted data
App.post('/data', (req, res) => {
  Const { rollNo, name, division, email, subject } = req.body;
  Res.render('display-data', { rollNo, name, division, email, subject });
});

App.listen(port, () => {
  Console.log(`Server is running on port ${port}`);
});
```

```
Doctype html
Html
 Head
  Title Student Form
 Body
  H1 Student Form
  Form(action="/data", method="POST")
   Label(for="rollNo") Roll No:
   Input(type="number", name="rollNo", required)
   Br
   Label(for="name") Name:
   Input(type="text", name="name", required)
   Br
   Label(for="division") Division:
   Input(type="text", name="division", required)
   Br
   Label(for="email") Email:
   Input(type="email", name="email", required)
   Br
   Label Subject:
   Input(type="radio", name="subject", value="FSD-2", required) FSD-2
   Input(type="radio", name="subject", value="COA", required) COA
   Input(type="radio", name="subject", value="PYTHON-2", required) PYTHON-2
   Input(type="radio", name="subject", value="DM", required) DM
   Input(type="radio", name="subject", value="TOC", required) TOC
   Br
   Input(type="submit", value="Submit")
```

```
Doctype html
Html
  Head
    Title Student Data
  Body
    H1 Student Data
    Ul
      Li Roll No: #{rollNo}
      Li Name: #{name}
      Li Division: #{division}
      Li Email: #{email}
      Li Subject: #{subject}
    A(href=”/”) Back to Form
```

```
Node app.js
```

5….>

```
Const express = require(‘express’);
Const multer = require(‘multer’);
Const path = require(‘path’);
Const app = express();
Const port = 3000;

// Set up multer for handling file uploads
Const storage = multer.memoryStorage();
Const upload = multer({
  Storage: storage,
  Limits: { fileSize: 1024 * 1024 }, // 1MB limit
  fileFilter: (req, file, cb) => {
    // Allow only text/plain MIME type (text files)
    If (file.mimetype === 'text/plain') {
      Cb(null, true);
    } else {
      Cb(new Error('Only text files are allowed'));
    }
  },
});

App.use(express.static('public'));

// Serve the HTML form
App.get('/', (req, res) => {
  Res.sendFile(path.join(__dirname, 'public', 'upload.html'));
});

// Handle file upload
App.post('/upload', upload.single('file'), (req, res) => {
  If (!req.file) {
    Return res.status(400).send('No file uploaded.');
  }

  // Access the file contents from req.file.buffer
  Const fileContents = req.file.buffer.toString('utf-8');

  // Process the file contents here (e.g., save to a database, manipulate, etc.)
  Res.send(`File uploaded successfully. Contents: <pre>${fileContents}</pre>`);
});

App.listen(port, () => {
  Console.log(`Server is running on port ${port}`);
});
```

```
<!DOCTYPE html>
<html>
<head>
  <title>File Upload</title>
</head>
<body>
  <h1>Upload a Text File (Max 1MB)</h1>
  <form action="/upload" method="post" enctype="multipart/form-data">
    <input type="file" name="file" accept=".txt" required>
    <input type="submit" value="Upload">
  </form>
</body>
</html>
```

```
Npm install express multer
Node app.js
```

6…>

```
Npm init -y

Npm install express



Const express = require('express');

Const session = require('express-session');

Const app = express();

Const port = 3000;


// Set up session middleware

App.use(

  Session({

    Secret: 'mysecret', // Change this to a more secure secret key

    Resave: false,

    saveUninitialized: true,

  })

);


// Serve static files (HTML, CSS, etc.)

App.use(express.static('public'));


// Handle form submission and save the username in session

App.post('/savesession', (req, res) => {

  Const { username } = req.body;

  Req.session.username = username;

  Res.redirect('/fetchsession');

});


// Display session value and logout link
```

```
App.get('/fetchsession', (req, res) => {
  Const username = req.session.username;
  If (!username) {
    Res.redirect('/');
  } else {
    Res.send(`
      <h1>Welcome, ${username}!</h1>
      <a href="/deletesession">Logout</a>
    `);
  }
});

// Handle session deletion and redirect to the index page
App.get('/deletesession', (req, res) => {
  Req.session.destroy((err) => {
    If (err) {
      Console.error(err);
    }
    Res.redirect('/');
  });
});

App.listen(port, () => {
  Console.log(`Server is running on port ${port}`);
});
```

```
Node app.js
```

7….>

```
<!DOCTYPE html>
<html>
<head>
  <title>Simple HTML Page</title>
</head>
<body>
  <h1>Hello, World!</h1>
  <p>This is a simple HTML page.</p>
</body>
</html>
```

```
Const http = require('http');
Const fs = require('fs');
Const path = require('path');

Const server = http.createServer((req, res) => {
  // Define the path to the HTML file
  Const filePath = path.join(__dirname, 'simple.html');

  // Check if the request URL is for the HTML file
  If (req.url === '/simple.html') {
    // Read the HTML file
    Fs.readFile(filePath, 'utf-8', (err, data) => {
      If (err) {
        Res.writeHead(500, { 'Content-Type': 'text/plain' });
        Res.end('Internal Server Error');
        Return;
```

```
            }

            // Set the response headers and send the HTML content
            Res.writeHead(200, { 'Content-Type': 'text/html' });
            Res.end(data);
        });
    } else {
        // Handle other requests (e.g., 404 Not Found)
        Res.writeHead(404, { 'Content-Type': 'text/plain' });
        Res.end('Not Found');
    }
});

Const port = 3000;
Server.listen(port, () => {
    Console.log(`Server is running on http://localhost:${port}`);
});
```

```
Node server.js
```

8…..>

Pug file

```
Doctype html
Html
  Head
    Title Online Store
    Link(rel='stylesheet', href='/styles.css')
  Body
```

```
    H1 Welcome to Our Online Store
    H2 Products
    Ul
      Each product in products
        Li
          A(href=`/products/${product.id}`)= product.name
    A(href='/') Go Back Home
```

Pug file

```
Doctype html
Html
  Head
    Title Error
    Link(rel='stylesheet', href='/styles.css')
  Body
    H1 Error 404 – Page Not Found
    P The page you are looking for does not exist.
    A(href='/') Go Back Home
```

```
Const express = require('express');
Const app = express();
Const port = 3000;

// Define an array of product objects
Const products = [
  { id: 1, name: 'Product 1', description: 'Description 1', price: '$10' },
  { id: 2, name: 'Product 2', description: 'Description 2', price: '$20' },
  { id: 3, name: 'Product 3', description: 'Description 3', price: '$30' },
];
```

```
// Set the view engine and views directory

App.set('view engine', 'pug');

App.set('views', __dirname + '/views');


// Serve static files from the public directory

App.use(express.static('public'));


// Define a route to display a welcome message on the homepage

App.get('/', (req, res) => {

  Res.send('Welcome to our online store!');

});


// Define a route to display a list of products

App.get('/products', (req, res) => {

  Res.render('products', { products });

});


// Define a dynamic route for product details

App.get('/products/:id', (req, res) => {

  Const productId = parseInt(req.params.id);

  Const product = products.find((p) => p.id === productId);


  If (!product) {

    Res.status(404).render('error');

    Return;

  }


  Res.send(`Product Details: ${product.name}, ${product.description}, Price: ${product.price}`);
});

// Handle 404 errors
App.use((req, res) => {
  Res.status(404).render('error');
});

App.listen(port, () => {
  Console.log(`Server is running on port ${port}`);
});

Node app.js
```

11….>

```
Let variable1 = 0;
Let variable2 = 0;

Function incrementAndDisplay() {
  Variable1++;
  Variable2++;
  Const sum = variable1 + variable2;
  Console.log(`Variable 1: ${variable1}, Variable 2: ${variable2}, Sum: ${sum}`);
}

// Call the incrementAndDisplay function every 1 second (1000 milliseconds)
setInterval(incrementAndDisplay, 1000);
```

12…>

```
Mkdir express-student-form

Cd express-student-form

Npm init -y

Npm install express pug
```

Pug file

```
Doctype html
Html
  Head
    Title Student Form
  Body
    H1 Student Form
    Form(action=”/student”, method=”POST”)
      Label(for=”name”) Name:
      Input(type=”text”, name=”name”, required)
      Br
      Label(for=”email”) Email:
      Input(type=”email”, name=”email”, required)
      Br
      Label Course:
      Input(type=”radio”, name=”course”, value=”CE”, required) CE
      Input(type=”radio”, name=”course”, value=”IT”, required) IT
      Input(type=”radio”, name=”course”, value=”CSE”, required) CSE
      Br
      Input(type=”submit”, value=”Submit”)
```

Create a Pug file to display the submitted data (views/student.pug):

Pug file

```
Doctype html
Html
  Head
    Title Student Data
  Body
    H1 Student Data
    P Name: #{name}
    P Email: #{email}
    P Course: #{course}
    A(href=”/”) Back to Form
```

Create an Express.js application in app.js:

Javascript:

```
Const express = require(‘express’);
Const app = express();
Const port = 3000;

App.set(‘view engine’, ‘pug’);
App.set(‘views’, __dirname + ‘/views’);

App.use(express.urlencoded({ extended: true }));

App.use(express.static(‘public’));

App.get(‘/’, (req, res) => {
  Res.render(‘student-form’);
});
```

```
App.post('/student', (req, res) => {
  Const { name, email, course } = req.body;
  Res.render('student', { name, email, course });
});

App.listen(port, () => {
  Console.log(`Server is running on port ${port}`);
});

Node app.js
```

13….>

```
Const readline = require('readline');

Const rl = readline.createInterface({
  Input: process.stdin,
  Output: process.stdout
});

// Function to calculate the area of a circle
Function calculateCircleArea(radius) {
  Return Math.PI * Math.pow(radius, 2);
}

// Function to calculate the perimeter of a square
Function calculateSquarePerimeter(side) {
  Return 4 * side;
}
```

```
// Prompt user for the radius of the circle
Rl.question('Enter the radius of the circle: ', (radiusInput) => {
  Const radius = parseFloat(radiusInput);

  If (isNaN(radius) || radius < 0) {
    Console.log('Radius must be positive.');
  } else {
    Const circleArea = calculateCircleArea(radius);
    Console.log(`The area of the circle is: ${circleArea.toFixed(2)}`);
  }

  // Prompt user for the side of the square
  Rl.question('Enter the side of the square: ', (sideInput) => {
    Const side = parseFloat(sideInput);

    If (isNaN(side) || side < 0) {
      Console.log('Side must be positive.');
    } else {
      Const squarePerimeter = calculateSquarePerimeter(side);
      Console.log(`The perimeter of the square is: ${squarePerimeter}`);
    }

    // Close the readline interface
    Rl.close();
  });
});
```

```
Node geometry.js
```

14…>

```
Const EventEmitter = require('events');

// Create a custom event emitter
Const myEmitter = new EventEmitter();

// Listener 1
Function listener1() {
  Console.log('Listener 1 called');
}

// Listener 2
Function listener2() {
  Console.log('Listener 2 called');
}

// Add the listeners to the common event
myEmitter.on('commonEvent', listener1);
myEmitter.on('commonEvent', listener2);

// Print the number of listeners associated with the emitter
Console.log(`Number of listeners: ${myEmitter.listenerCount('commonEvent')}`);

// Emit the common event, which will trigger both listeners
myEmitter.emit('commonEvent');

// Remove one of the listeners (listener2)
```

```
myEmitter.removeListener('commonEvent', listener2);

// Print the number of remaining listeners
Console.log(`Number of remaining listeners: ${myEmitter.listenerCount('commonEvent')}`);

// Emit the common event again, which will trigger only listener1
myEmitter.emit('commonEvent');
```

15…>

```
Const express = require('express');
Const multer = require('multer');
Const app = express();

// Define storage for uploaded files
Const storage = multer.diskStorage({
  Destination: 'uploads/', // Specify the destination directory
  Filename: (req, file, cb) => {
    // Customize the filename as needed (e.g., keep the original filename)
    Const uniqueSuffix = Date.now() + '-' + Math.round(Math.random() * 1E9);
    Cb(null, file.fieldname + '-' + uniqueSuffix + '.' + file.originalname.split('.').pop());
  },
});

// Initialize multer with the defined storage configuration
Const upload = multer({ storage });

// Create a route to handle file uploads
App.post('/upload', upload.single('file'), (req, res) => {
  // Handle the uploaded file here (e.g., save it to a database, perform further processing)
  Res.send('File uploaded successfully');
});

// Start the Express.js server
Const port = 3000;
App.listen(port, () => {
  Console.log(`Server is running on port ${port}`);
})
```

9….>

```
Mkdir weather-forecast-app
Cd weather-forecast-app
Npm init -y
Npm install express pug
```

Pug file

```
Doctype html
Html
  Head
    Title Weather Forecast
  Body
    H1 Welcome to the Weather Forecast Service
    P Please enter a location to check the weather.
    Form(action="/weather", method="get")
      Input(type="text", name="location", placeholder="Enter location", required)
      Button(type="submit") Get Weather
```

Pug file

```
Doctype html
Html
  Head
    Title Weather Forecast
  Body
    H1 Weather Forecast for #{location}
    P Temperature: #{temperature}Â°C
    P Description: #{description}
    A(href=”/”) Go Back
```

Avascript

Copy code

```
Const express = require(‘express’);
Const app = express();
Const port = 3000;

// Set the view engine and views directory
App.set(‘view engine’, ‘pug’);
App.set(‘views’, __dirname + ‘/views’);

// Serve static files from the public directory (e.g., CSS, images)
App.use(express.static(‘public’));

// Define a route for the root URL (“/”) to display the welcome message
App.get(‘/’, (req, res) => {
  Res.render(‘index’);
});
```

```
// Define a route for weather forecast ("/weather")
App.get('/weather', (req, res) => {
  Const location = req.query.location;
  Const weatherData = {
    Location,
    Temperature: '25', // Replace with actual temperature data
    Description: 'Sunny', // Replace with actual weather description data
  };

  If (!location) {
    Res.status(400).send('Please provide a location.');
  } else {
    Res.render('weather', weatherData);
  }
});

Node app.js
```

16…>

```
Mkdir form-processing-app
Cd form-processing-app
Npm init -y
Npm install express body-parser
```

```
Const express = require('express');
Const bodyParser = require('body-parser');
Const app = express();
Const port = 3000;

// Use body-parser middleware to parse form data
App.use(bodyParser.urlencoded({ extended: false }));

// Serve static files (CSS, images, etc.) from the “public” directory
App.use(express.static(‘public’));

// Define a route to serve the HTML form
App.get(‘/’, (req, res) => {
  Res.send(`
    <!DOCTYPE html>
    <html>
    <head>
      <title>Form Processing</title>
    </head>
    <body>
      <h1>Form Processing</h1>
      <form method=”post” action=”/process”>
        <label for=”username”>Username:</label>
        <input type=”text” id=”username” name=”username” required><br><br>

        <label for=”password”>Password:</label>
        <input type=”password” id=”password” name=”password” required><br><br>

        <label for=”confirmPassword”>Confirm Password:</label>
        <input type=”password” id=”confirmPassword” name=”confirmPassword” required><br><br>
        <label for=”gender”>Gender:</label>
        <select id=”gender” name=”gender”>
          <option value=”male”>Male</option>
          <option value=”female”>Female</option>
          <option value=”other”>Other</option>
        </select><br><br>

        <input type=”submit” value=”Submit”>
        <input type=”reset” value=”Reset”>
      </form>
    </body>
    </html>
  `);
});

// Define a route to process the form data
App.post(‘/process’, (req, res) => {
  Const { username, password, confirmPassword, gender } = req.body;

  If (password === confirmPassword) {
    Res.send(`
      <h1>Form Processed Successfully</h1>
      <p>Username: ${username}</p>
      <p>Password: ${password}</p>
      <p>Gender: ${gender}</p>
    `);
  } else {
    Res.send(‘<p style=”color: red;”>Password and Confirm Password do not match. Please try
again.</p>’);
```

```
  }
});

// Start the Express.js server
App.listen(port, () => {
  Console.log(`Server is running on port ${port}`);
});
```

```
Node app.js
```

17

```
Import React, { Component } from 'react';
Import './App.css';

Class App extends Component {
  Constructor() {
    Super();
    This.state = {
      Tasks: [],
      newTask: '',
    };
  }

  handleInputChange = (event) => {
    this.setState({ newTask: event.target.value });
  };
```

```
addTask = () => {
  const { newTask, tasks } = this.state;
  if (newTask.trim() !== ‘’) {
    this.setState({
      tasks: […tasks, newTask],
      newTask: ‘’,
    });
  }
};

Render() {
  Const { tasks, newTask } = this.state;

  Return (
    <div className=”App”>
      <h1>To-Do List</h1>
      <div className=”task-input”>
        <input
          Type=”text”
          Placeholder=”Enter a task”
          Value={newTask}
          onChange={this.handleInputChange}
        />
        <button onClick={this.addTask}>Add</button>
      </div>
      <ul>
        {tasks.map((task, index) => (
          <li key={index}>{task}</li>
        ))}
        </ul>
      </div>
    );
  }
}

Export default App;
```

18

```
Import React, { Component } from ‘react’;
Import ‘./App.css’;

Class App extends Component {
  Constructor() {
    Super();
    This.state = {
      Time: new Date().toLocaleTimeString(),
    };
  }

  componentDidMount() {
    // Update the time every second
    This.intervalID = setInterval(() => {
      This.setState({
        Time: new Date().toLocaleTimeString(),
      });
    }, 1000);
  }
```

```
 componentWillUnmount() {
   // Clear the interval when the component unmounts
   clearInterval(this.intervalID);
 }

 Render() {
   Const { time } = this.state;

   Return (
     <div className="App">
       <h1>Digital Clock</h1>
       <p>{time}</p>
     </div>
   );
 }
}

Export default App;
```

19

```
Import React, { Component } from 'react';
Import './StockDetail.css';

Class StockDetail extends Component {
 Constructor() {
   Super();
   This.state = {
     Name: '',
     purchasePrice: '',
     purchaseQuantity: '',
     sellingPrice: '',
     sellingQuantity: '',
     stocks: [],
   };
 }

 handleInputChange = (event) => {
   const { name, value } = event.target;
   this.setState({ [name]: value });
 };

 addStock = () => {
   const {
     name,
     purchasePrice,
     purchaseQuantity,
     sellingPrice,
     sellingQuantity,
     stocks,
   } = this.state;

   If (parseInt(sellingQuantity) > parseInt(purchaseQuantity)) {
     Alert('Selling quantity cannot be more than purchase quantity.');
     Return;
   }
   Const profitLoss =
     parseInt(sellingQuantity) < parseInt(purchaseQuantity)
       ? ‘Invested’
       : (sellingPrice – purchasePrice) * sellingQuantity;

   Const newStock = {
     Name,
     purchasePrice,
     purchaseQuantity,
     sellingPrice,
     sellingQuantity,
     profitLoss,
   };

   This.setState({
     Stocks: […stocks, newStock],
     Name: ‘’,
     purchasePrice: ‘’,
     purchaseQuantity: ‘’,
     sellingPrice: ‘’,
     sellingQuantity: ‘’,
   });
 };

 Render() {
   Const { name, purchasePrice, purchaseQuantity, sellingPrice, sellingQuantity, stocks } = this.state;

   Return (
     <div className=”StockDetail”>
```

```
<h2>Stock Detail</h2>
<div className=”input-fields”>
 <input
  Type=”text”
  Name=”name”
  Placeholder=”Name”
  Value={name}
  onChange={this.handleInputChange}
 />
 <input
  Type=”number”
  Name=”purchasePrice”
  Placeholder=”Purchase Price”
  Value={purchasePrice}
  onChange={this.handleInputChange}
 />
 <input
  Type=”number”
  Name=”purchaseQuantity”
  Placeholder=”Purchase Quantity”
  Value={purchaseQuantity}
  onChange={this.handleInputChange}
 />
 <input
  Type=”number”
  Name=”sellingPrice”
  Placeholder=”Selling Price”
  Value={sellingPrice}
  onChange={this.handleInputChange}
 />
 <input
  Type="number"
  Name="sellingQuantity"
  Placeholder="Selling Quantity"
  Value={sellingQuantity}
  onChange={this.handleInputChange}
 />
 <button onClick={this.addStock}>Add Stock</button>
</div>

<table>
 <thead>
  <tr>
   <th>Name</th>
   <th>Purchase Price</th>
   <th>Purchase Quantity</th>
   <th>Selling Price</th>
   <th>Selling Quantity</th>
   <th>Profit/Loss</th>
  </tr>
 </thead>
 <tbody>
  {stocks.map((stock, index) => (
   <tr key={index}>
    <td>{stock.name}</td>
    <td>{stock.purchasePrice}</td>
    <td>{stock.purchaseQuantity}</td>
    <td>{stock.sellingPrice}</td>
```

```
        <td>{stock.sellingQuantity}</td>
        <td className={stock.profitLoss === 'Invested' ? 'invested' : stock.profitLoss >= 0 ? 'profit' : 'loss'}>
         {stock.profitLoss === 'Invested' ? 'Invested' : `$${stock.profitLoss}`}
        </td>
       </tr>
      ))}
     </tbody>
    </table>
   </div>
  );
 }
}


Export default StockDetail;
```

20

To enhance the `VegetableCategories` React component as per your requirements, follow these steps:

1. Default Selection:
   - Initialize the default selected category state in the component's constructor.
   - Populate the `defaultCategory` state with the initial category ('fruits') and three random fruits.

```jsx
Constructor() {
 Super();
```

```
  This.state = {
    selectedCategory: ‘fruits’,
    categories: Object.keys(vegetableData),
    defaultCategory: {
      name: ‘Fruits’,
      vegetables: vegetableData[‘fruits’].slice(0, 3),
    },
    Error: null,
  };
}
```

2. Add a New Category:

   - Expand the `vegetableData` object to include the “Legumes” category with its vegetables.

```jsx
Const vegetableData = {
  Fruits: [‘Tomato’, ‘Cucumber’, ‘Bell Pepper’],
  leafyGreens: [‘Spinach’, ‘Kale’, ‘Lettuce’],
  rootVegetables: [‘Carrot’, ‘Potato’, ‘Beetroot’],
  legumes: [‘Lentils’, ‘Chickpeas’, ‘Black Beans’],
};
```

3. Display New Category:

   - Create a function to update the selected category and handle its associated vegetables.

```jsx
handleCategoryChange = (event) => {
  const selectedCategory = event.target.value;
  if (selectedCategory === ‘Select Category’) {
    this.setState({ selectedCategory, error: ‘Please select a category.’, defaultCategory: null });
  } else if (vegetableData[selectedCategory]) {
    This.setState({ selectedCategory, error: null, defaultCategory: null });
  } else {
    This.setState({ selectedCategory, error: ‘Category not found.’, defaultCategory: null });
  }
};
```

4. Styling:

   - Apply CSS or a CSS-in-JS approach to style the component. Below is a basic example using CSS:

```css
/* VegetableCategories.css */

.VegetableCategories {
  Max-width: 400px;
  Margin: 0 auto;
  Padding: 20px;
  Border: 1px solid #ccc;
  Box-shadow: 0 2px 4px rgba(0, 0, 0, 0.1);
}

.error {
  Color: red;
  Font-weight: bold;
}
```

```css
Select {
  Width: 100%;
  Padding: 10px;
  Margin-bottom: 10px;
}

Ul {
  List-style-type: none;
  Padding: 0;
}

Li {
  Margin: 5px 0;
}
```

Remember to import the CSS file into your component:

```jsx
Import './VegetableCategories.css';
```

Now, your `VegetableCategories` component should be enhanced with the requested features, error handling, and styling. It defaults to the "Fruits" category, displays the "Legumes" category when selected, and handles errors appropriately. The styling can be customized further to match your desired visual appeal.

To create a replica set and perform these actions, you typically work with MongoDB. Below are step-by-step instructions on how to achieve your goal:

1. **Set Up Replica Set**:

First, make sure you have MongoDB installed and running.

Start three separate MongoDB instances on ports 27017, 27018, and 27019, each in its own terminal window.

```bash
Mongod –port 27017 –replSet rs1
Mongod –port 27018 –replSet rs1
Mongod –port 27019 –replSet rs1
```

2. **Initialize Replica Set**:

Connect to the primary instance (port 27017) and initialize the replica set:

```bash
Mongo –port 27017
➢ Rs.initiate()
```

Then, add the other two instances to the replica set:

```bash
> rs.add(“localhost:27018”)
> rs.add(“localhost:27019”)
```

3. **Create a Collection and Insert Documents**:

On the primary port (27017), create a database (e.g., “mydb”) and a collection (e.g., “student”). Insert some sample documents into the “student” collection:

```bash
> use mydb
> db.createCollection(“student”)
> db.student.insertMany([
  { name: “Alice”, age: 18, date: new Date() },
  { name: “Bob”, age: 20, date: new Date() },
  { name: “Charlie”, age: 15, date: new Date() },
  { name: “David”, age: 25, date: new Date() }
])
```

4. **Read Documents on Replica Port**:

Connect to one of the secondary instances (port 27018 or 27019) to read documents with age greater than 15:

```bash
Mongo –port 27018
```

```
  > rs.slaveOk() // Allow reading from secondary

  > use mydb

  > db.student.find({ age: { $gt: 15 } })

  ```

This setup replicates data from the primary (port 27017) to the secondary (port 27018 or 27019) in the "rs1" replica set. You can read documents on the secondary with the `rs.slaveOk()` command. Make sure to adjust the port numbers and database/collection names to match your specific configuration if necessary.

22--------

To create an HTML form with validations and insert data into a MongoDB collection, you'll need a combination of HTML for the form, JavaScript for validation, and a backend (e.g., Node.js with Express) to handle the MongoDB insertion. Here's a simplified example:

HTML (form.html):

```html
<!DOCTYPE html>

<html>

<head>

  <title>Form</title>

</head>

<body>

  <h2>Employee Details</h2>

  <form id=”employeeForm” action=”/submit” method=”POST”>

    <label for=”name”>Name (3-12 characters, capital letters only):</label>

    <input type=”text” id=”name” name=”name” required pattern=”[A-Z]{3,12}” /><br /><br />
```

```
    <label for=”email”>Email (Valid Email Address):</label>
    <input type=”email” id=”email” name=”email” required /><br /><br />

    <label for=”doj”>Date of Joining (1-1-2010 to 31-12-2022):</label>
    <input type=”date” id=”doj” name=”doj” required
      Min=”2010-01-01” max=”2022-12-31” /><br /><br />

    <input type=”submit” value=”Submit” />
  </form>
</body>
</html>
```

JavaScript (script.js):

This JavaScript file is responsible for client-side validation. It ensures that the form data meets the specified requirements before submission.

```javascript
Document.getElementById(‘employeeForm’).addEventListener(‘submit’, function € {
  Const nameField = document.getElementById(‘name’);
  Const emailField = document.getElementById(‘email’);
  Const dojField = document.getElementById(‘doj’);

  If (!nameField.checkValidity()) {
    Alert(‘Invalid name format. Use 3-12 capital letters only.’);
    e.preventDefault();
    return;
  }

  If (!emailField.checkValidity()) {
    Alert('Invalid email format.');
    e.preventDefault();
    return;
  }

  If (!dojField.checkValidity()) {
    Alert('Invalid date. Use a date between 1-1-2010 and 31-12-2022.');
    e.preventDefault();
    return;
  }
});
```

Node.js (app.js):

Here's a basic Node.js and Express app to handle form submission and database insertion.

```javascript
Const express = require('express');
Const mongoose = require('mongoose');
Const bodyParser = require('body-parser');

Const app = express();
App.use(bodyParser.urlencoded({ extended: true }));
App.use(express.static('public')); // Serve HTML and JS files from the 'public' folder.
```

```
Mongoose.connect(‘mongodb://localhost/mydb’, { useNewUrlParser: true, useUnifiedTopology: true });
Const detailsSchema = new mongoose.Schema({
  Name: String,
  Email: String,
  Doj: Date
});
Const Detail = mongoose.model(‘Detail’, detailsSchema);

App.post(‘/submit’, (req, res) => {
  Const name = req.body.name.toUpperCase().trim();
  Const email = req.body.email;
  Const doj = req.body.doj;

  Const newDetail = new Detail({ name, email, doj });

  newDetail.save((err) => {
    if (err) {
      console.error(err);
      res.redirect(‘/error.html’); // Redirect to an error page if there’s an issue.
    } else {
      Res.redirect(‘/success.html’); // Redirect to a success page after successful insertion.
    }
  });
});

App.listen(3000, () => {
  Console.log(‘Server started on port 3000’);
});
```

Please note that this is a simplified example for demonstration purposes. In a production environment, you should handle errors more gracefully and securely. Additionally, you would typically use environment variables for database connections and employ best practices for security.

23------

```
Const MongoClient = require('mongodb').MongoClient;

// Connection URL and database name
Const url = 'mongodb://localhost:27017';
Const dbName = 'maindata';

// Create a new MongoClient
Const client = new MongoClient(url, { useNewUrlParser: true, useUnifiedTopology: true });

// Connect to the MongoDB server
Client.connect(async (err) => {
  If (err) {
    Console.error('Error connecting to the database:', err);
    Return;
  }

  Const db = client.db(dbName);
  Const userdata = db.collection('userdata');

  // Task 1: Insert a category field with value "SeniorCitizen" for age > 60
  Await userdata.updateMany({ age: { $gt: 60 } }, { $set: { category: 'SeniorCitizen' } });
  // Task 2: Sort the collection by age in descending order and display the youngest person's name only
  Const youngest = await userdata.find().sort({ age: -1 }).limit(1).toArray();
  Console.log('Youngest Person Name:', youngest[0].name);

  // Task 3: Display total number of documents having age between 30 and 60 only
  Const ageBetween30And60Count = await userdata.countDocuments({ age: { $gte: 30, $lte: 60 } });
  Console.log('Total documents with age between 30 and 60:', ageBetween30And60Count);

  // Task 4: Display only the surname field in ascending order of age
  Const surnamesAscending = await userdata.find({}, { projection: { _id: 0, surname: 1 } }).sort({ age: 1
}).toArray();
  Console.log('Surnames in ascending order of age:', surnamesAscending);

  // Task 5: Delete the record having age greater than 60
  Await userdata.deleteMany({ age: { $gt: 60 } });

  // Close the connection
  Client.close();
});
```

24-------

```
Import React, { Component } from 'react';
Import './App.css';
```

```
Class App extends Component {
 Constructor() {
  Super();
  This.state = {
   Text: ‘Hello’,
   Color: ‘red’,
   isHidden: false,
  };
 }

 handleToggleText = () => {
  this.setState((prevState) => ({
   text: prevState.text === ‘Hello’ ? ‘Welcome’ : ‘Hello’,
  }));
 };

 handleToggleColor = () => {
  this.setState((prevState) => ({
   color: prevState.color === ‘red’ ? ‘blue’ : ‘red’,
  }));
 };

 handleToggleVisibility = () => {
  this.setState((prevState) => ({
   isHidden: !prevState.isHidden,
  }));
 };

 Render() {
  Const { text, color, isHidden } = this.state;

  Return (
   <div className="App">
    <button onClick={this.handleToggleText}>Change Text</button>
    <button onDoubleClick={this.handleToggleColor}>Change Color</button>
    <button onClick={this.handleToggleVisibility}>
     {isHidden ? 'Show' : 'Hide'}
    </button>
    <h1 style={{ color }}>{text}</h1>
    {!isHidden && <h2>Good Morning</h2>}
   </div>
  );
 }
}

Export default App;
```

25--------

Creating a basic To-Do list React component with MongoDB integration involves setting up the React app, creating the component, and implementing the necessary functionality. Here's a step-by-step guide:

1. **Set Up a React App**:

Create a new React app if you haven't already using Create React App or your preferred method.

```bash
Npx create-react-app todo-list-mongodb
```

```
Cd todo-list-mongodb
```

2. **Install Required Dependencies**:

Install the MongoDB Node.js driver to interact with MongoDB from your Node.js server:

```bash
Npm install mongodb
```

3. **Create a TaskList Component**:

Create a new file called `TaskList.js` in the `src` directory and define your `TaskList` component with the required functionality.

```jsx
Import React, { Component } from 'react';
Import './TaskList.css'; // Create this CSS file for styling

Class TaskList extends Component {
  Constructor(props) {
    Super(props);
    This.state = {
      Tasks: [],
      newTask: '',
    };
  }
  // Implement methods for adding, toggling completion, and filtering tasks

  Render() {
    // Render the task list, input field, filter buttons, and implement event handlers

    Return (
      <div className="TaskList">
        {/* Render the UI elements */}
      </div>
    );
  }
}

Export default TaskList;
```

4. **Implement MongoDB Integration**:

In the same component, add code to connect to MongoDB, insert and retrieve tasks from the "tasklist" collection, and handle state updates accordingly.

5. **Implement Task Management Logic**:

Inside your `TaskList` component, create methods to add new tasks, toggle task completion, and filter tasks based on their completion status.

6. **Styling**:

Create a CSS file (e.g., `TaskList.css`) to style your task list and related elements.

7. **Use the Component**:

Import and use the `TaskList` component in your `App.js` or any other parent component.

8. **Set Up MongoDB**:

Make sure you have MongoDB running locally or on a remote server. Create a database named "Task" and a collection named "tasklist" to store the tasks.

Note: In a production environment, you should set up a server to interact with MongoDB and not directly access the database from the React app for security reasons.

This is a high-level overview of how to create a basic To-Do list React component with MongoDB integration. The actual implementation involves coding the details of adding, toggling, filtering tasks, and handling the database interactions.

26-------

```
Import React, { Component } from 'react';
Import './ExpenseTracker.css'; // Create this CSS file for styling

Class ExpenseTracker extends Component {
  Constructor() {
    Super();
    This.state = {
      Expenses: [],
      newExpense: {
        name: '',
        amount: '',
        category: 'Food',
        date: '',
      },
      filterCategory: 'All',
      filterDate: 'All',
      error: null,
    };
  }

  // Implement methods for adding expenses, filtering, and handling errors

  Render() {
    Const { expenses, newExpense, filterCategory, filterDate, error } = this.state;

    Return (
      <div className="ExpenseTracker">
        {/* Render the list of expenses, input fields, filter controls, and error messages */}
      </div>
    );
  }
}

Export default ExpenseTracker
```

27-----

Creating a React component for a weather application with the specified features involves setting up the component, handling user input, fetching weather data, and displaying it. Here’s a simplified example:

```jsx
Import React, { Component } from ‘react’;
Import ‘./WeatherApp.css’; // Create this CSS file for styling

Class WeatherApp extends Component {
  Constructor() {
    Super();
    This.state = {
      cityName: ‘’,
      weatherData: null,
      error: null,
    };
  }

  handleCityChange = (event) => {
    this.setState({ cityName: event.target.value });
  };

  fetchWeatherData = () => {
    const apiKey = ‘YOUR_API_KEY’; // Replace with your actual API key
    const { cityName } = this.state;
    const apiUrl =
`https://api.openweathermap.org/data/2.5/weather?q=${cityName}&appid=${apiKey}&units=metric`;
```

```
  fetch(apiUrl)
    .then((response) => response.json())
    .then((data) => {
      If (data.cod === 200) {
        This.setState({ weatherData: data, error: null });
      } else {
        This.setState({ weatherData: null, error: data.message });
      }
    })
    .catch((error) => {
      This.setState({ weatherData: null, error: 'Network issue. Please try again.' });
    });
};

Render() {
  Const { cityName, weatherData, error } = this.state;

  Return (
    <div className="WeatherApp">
      <h2>Weather App</h2>
      <input
        Type="text"
        Placeholder="Enter city name"
        Value={cityName}
        onChange={this.handleCityChange}
      />
      <button onClick={this.fetchWeatherData}>Get Weather</button>

      {error && <p className="error">{error}</p>}
    {weatherData && (
     <div>
      <h3>{weatherData.name}</h3>
      <p>Temperature: {weatherData.main.temp}°C</p>
      <p>Weather: {weatherData.weather[0].description}</p>
     </div>
    )}
   </div>
  );
 }
}

Export default WeatherApp;
```

In this example:

1. We have an input field for entering the city name, a button to fetch weather data, and a display area for the weather information.

2. We use the OpenWeatherMap API (replace 'YOUR_API_KEY' with your actual API key) to fetch weather data based on the user's input.

3. We handle errors, such as a city not found or network issues, by displaying error messages.

4. When successful, we display the city name, temperature, and weather description.

Remember to replace 'YOUR_API_KEY' with a valid API key from OpenWeatherMap or any other weather data provider you prefer. Additionally, you can style the component by creating a CSS file (`WeatherApp.css`) to make it visually appealing.

28---

Creating a ReactJS script for a form with the specified fields and inserting the submitted values into a MongoDB database requires a combination of frontend and backend development. Below is an example of how you can create the frontend component in React and outline the backend setup.

Frontend (ReactJS):

1. Create a React component that represents the form with the specified fields and handles user input:

~~~
```jsx
Import React, { Component } from ‘react’;

Class UserForm extends Component {
 Constructor() {
  Super();
  This.state = {
   City: ‘Ahmedabad’,
   bloodGroup: ‘O+’,
  };
 }

 handleCityChange = (event) => {
  this.setState({ city: event.target.value });
 };

 handleBloodGroupChange = (event) => {
  this.setState({ bloodGroup: event.target.value });
 };

 handleSubmit = () => {
~~~

```
  const { city, bloodGroup } = this.state;

  // Send the data to the backend for database insertion
  Fetch('/api/addUser', {
    Method: 'POST',
    Headers: {
      'Content-Type': 'application/json',
    },
    Body: JSON.stringify({ city, bloodGroup }),
  })
    .then((response) => response.json())
    .then((data) => {
      Console.log(data);
      // Reset the form or show a success message as needed
    })
    .catch((error) => {
      Console.error('Error:', error);
    });
};

Render() {
  Return (
    <div>
      <h2>User Registration Form</h2>
      <div>
        <label>
          City:
          <select value={this.state.city} onChange={this.handleCityChange}>
            <option value="Ahmedabad">Ahmedabad</option>
```

```
       <option value=”Rajkot”>Rajkot</option>
       <option value=”Surat”>Surat</option>
       <option value=”Vadodara”>Vadodara</option>
      </select>
     </label>
    </div>
    <div>
     <label>
      Blood Group:
      <input
       Type=”radio”
       Name=”bloodGroup”
       Value=”O+”
       Checked={this.state.bloodGroup === ‘O+’}
       onChange={this.handleBloodGroupChange}
      />
      O+
      {/* Repeat similar input elements for other blood group options */}
     </label>
    </div>
    <button onClick={this.handleSubmit}>Submit</button>
   </div>
  );
 }
}

Export default UserForm;
```

Backend (Node.js with Express and MongoDB):

Set up a backend server using Node.js and Express to handle the POST request for adding a user to the MongoDB database. Here’s a basic outline:

1. Install the required dependencies (Express, Mongoose for MongoDB, etc.):

```bash
Npm install express mongoose body-parser
```

2. Create a server file (e.g., `server.js`) for your Node.js backend:

```javascript
Const express = require(‘express’);
Const mongoose = require(‘mongoose’);
Const bodyParser = require(‘body-parser’);

Const app = express();
Const port = process.env.PORT || 5000;

// Connect to MongoDB (replace ‘mongodb://localhost/mydb’ with your MongoDB connection URL)
Mongoose.connect(‘mongodb://localhost/LJU’, { useNewUrlParser: true, useUnifiedTopology: true });
Const db = mongoose.connection;

Db.on(‘error’, console.error.bind(console, ‘MongoDB connection error:’));
Db.once(‘open’, () => {
  Console.log(‘Connected to MongoDB’);
});
```

```
});



// Define a User schema and model (adjust as needed)
Const userSchema = new mongoose.Schema({
  City: String,
  bloodGroup: String,
});

Const User = mongoose.model('User', userSchema);

// Middleware for parsing JSON requests
App.use(bodyParser.json());

// Define a route to add a user to the database
App.post('/api/addUser', (req, res) => {
  Const { city, bloodGroup } = req.body;

  Const newUser = new User({ city, bloodGroup });

  newUser.save((err) => {
    if (err) {
      console.error('Error inserting user:', err);
      res.status(500).json({ error: 'Error inserting user' });
    } else {
      Console.log('User inserted successfully');
      Res.json({ message: 'User inserted successfully' });
    }
  });
});
```

// Start the server

App.listen(port, () => {

  Console.log(`Server is running on port ${port}`);

});

```

3. Replace the MongoDB connection URL with your actual MongoDB connection URL.

4. Run your Node.js server using `node server.js`.

This setup provides a basic structure for handling the frontend form submission and inserting the submitted data into a MongoDB database. You'll need to replace the connection URL and adapt the code as needed for your specific application

**29----**

**Import React from 'react';**

**Import { BrowserRouter as Router, Route, Link, Switch, Redirect } from 'react-router-dom';**

**Const SubjectIndex = () => (**

**  <div>**

**    <h2>Subject Index</h2>**

**    <ul>**

**      <li>**

**        <Link to="/fsd2/json">JSON</Link>**

**      </li>**

**      <li>**

**        <Link to="/fsd2/nodejs">NodeJS/ExpressJS</Link>**

**      </li>**
```

```
      <li>
        <Link to=”/fsd2/reactjs”>React JS</Link>
      </li>
    </ul>
  </div>
);

Const Content = () => (
  <div>
    <h2>Content</h2>
    <table>
      <tr>
        <th>Topic</th>
        <th>Details</th>
      </tr>
      <tr>
        <td>JSON</td>
        <td>JSON content goes here.</td>
      </tr>
      <tr>
        <td>NodeJS/ExpressJS</td>
        <td>NodeJS/ExpressJS content goes here.</td>
      </tr>
      <tr>
        <td>React JS</td>
        <td>React JS content goes here.</td>
      </tr>
    </table>
  </div>
```

```
);

Const NoPage = () => (
  <div>
    <h2>No page found</h2>
    <p>Sorry, the page you requested does not exist.</p>
  </div>
);

Const App = () => (
  <Router>
    <div>
      <Switch>
        <Route path=”/fsd2” exact component={SubjectIndex} />
        <Route path=”/fsd2/json” component={Content} />
        <Route path=”/fsd2/nodejs” component={Content} />
        <Route path=”/fsd2/reactjs” component={Content} />
        <Route path=”/no-page” component={NoPage} />
        <Redirect from=”/” to=”/fsd2” />
        <Redirect to=”/no-page” />
      </Switch>
    </div>
  </Router>
);

Export default App;
```

30----

Creating a ReactJS script for a form with two fields (Subject Name and Marks) and inserting the entered values into a MongoDB database requires a combination of frontend and backend development. Below is an example of how you can create the frontend component in React and outline the backend setup.

Frontend (ReactJS):

1. Create a React component that represents the form with the specified fields and handles user input:

```jsx
Import React, { Component } from 'react';

Class StudentForm extends Component {
  Constructor() {
    Super();
    This.state = {
      subjectName: 'FSD2',
      marks: '',
    };
  }

  handleSubjectNameChange = (event) => {
    this.setState({ subjectName: event.target.value });
  };

  handleMarksChange = (event) => {
    this.setState({ marks: event.target.value });
  };
```

```
handleSubmit = () => {
 const { subjectName, marks } = this.state;

 // Send the data to the backend for database insertion
 Fetch('/api/addStudentData', {
  Method: 'POST',
  Headers: {
   'Content-Type': 'application/json',
  },
  Body: JSON.stringify({ subjectName, marks }),
 })
  .then((response) => response.json())
  .then((data) => {
   Console.log(data);
   // Reset the form or show a success message as needed
  })
  .catch((error) => {
   Console.error('Error:', error);
  });
};

Render() {
 Return (
  <div>
   <h2>Student Data Form</h2>
   <div>
    <label>
     Subject Name:
      <select value={this.state.subjectName} onChange={this.handleSubjectNameChange}>
       <option value=”FSD2”>FSD2</option>
       <option value=”FCSP2”>FCSP2</option>
       <option value=”DS”>DS</option>
       <option value=”TOC”>TOC</option>
       <option value=”COA”>COA</option>
      </select>
     </label>
    </div>
    <div>
     <label>
      Marks:
      <input
       Type=”text”
       Value={this.state.marks}
       onChange={this.handleMarksChange}
      />
     </label>
    </div>
    <button onClick={this.handleSubmit}>Submit</button>
   </div>
  );
 }
}

Export default StudentForm;
```

Backend (Node.js with Express and MongoDB):

});

Set up a backend server using Node.js and Express to handle the POST request for adding student data to the MongoDB database. Here’s a basic outline:

1. Install the required dependencies (Express, Mongoose for MongoDB, etc.):

```bash
Npm install express mongoose body-parser
```

2. Create a server file (e.g., `server.js`) for your Node.js backend:

```javascript
Const express = require(‘express’);
Const mongoose = require(‘mongoose’);
Const bodyParser = require(‘body-parser’);

Const app = express();
Const port = process.env.PORT || 5000;

// Connect to MongoDB (replace ‘mongodb://localhost/mydb’ with your MongoDB connection URL)
Mongoose.connect(‘mongodb://localhost/StudentData’, { useNewUrlParser: true, useUnifiedTopology:
true });
Const db = mongoose.connection;

Db.on(‘error’, console.error.bind(console, ‘MongoDB connection error:’));
Db.once(‘open’, () => {
  Console.log(‘Connected to MongoDB’);
});
```

```
// Define a StudentData schema and model (adjust as needed)
Const studentDataSchema = new mongoose.Schema({
  subjectName: String,
  marks: Number,
});

Const StudentData = mongoose.model('Information', studentDataSchema);

// Middleware for parsing JSON requests
App.use(bodyParser.json());

// Define a route to add student data to the database
App.post('/api/addStudentData', (req, res) => {
  Const { subjectName, marks } = req.body;

  Const newStudentData = new StudentData({ subjectName, marks });

  newStudentData.save((err) => {
    if (err) {
      console.error('Error inserting student data:', err);
      res.status(500).json({ error: 'Error inserting student data' });
    } else {
      Console.log('Student data inserted successfully');
      Res.json({ message: 'Student data inserted successfully' });
    }
  });
});
```

```
// Start the server
App.listen(port, () => {
  Console.log(`Server is running on port ${port}`);
});
```

3. Replace the MongoDB connection URL with your actual MongoDB connection URL.

4. Run your Node.js server using `node server.js`.

This setup provides a basic structure for handling the frontend form submission and inserting the submitted data into a MongoDB database. You'll need to adapt the code as needed for your specific application and replace the connection URL with your actual MongoDB connection URL.

31----

To create an HTML form for collecting data and apply validations, you can use HTML, JavaScript, and MongoDB. Here's an example of an HTML form with validation rules and the corresponding Node.js backend to insert data into a MongoDB collection:

****HTML Form (form.html):****

```html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Exam Registration</title>
```

```
</head>

<body>

   <h1>Exam Registration</h1>

   <form id=”examForm”>

       <label for=”name”>Name (Max 12 characters, uppercase only):</label>

       <input type=”text” id=”name” name=”name” maxlength=”12” pattern=”[A-Z]+”
required><br><br>


       <label for=”email”>Email:</label>

       <input type=”email” id=”email” name=”email” required><br><br>


       <label for=”city”>City (Ahmedabad, Gandhinagar, Vadodara):</label>

       <input type=”text” id=”city” name=”city” pattern=”^(Ahmedabad|Gandhinagar|Vadodara)$”
required><br><br>


       <label for=”examDate”>Date of Exam (Between 1-10-2023 and 12-10-2023):</label>

       <input type=”date” id=”examDate” name=”examDate” min=”2023-10-01” max=”2023-10-12”
required><br><br>


       <input type=”submit” value=”Submit”>

   </form>


   <script src=”form.js”></script>

</body>

</html>
```

JavaScript for Form Validation (form.js):

```javascript
```

```
Document.addEventListener("DOMContentLoaded", function () {
 Const form = document.getElementById("examForm");

 Form.addEventListener("submit", function (event) {
  Event.preventDefault();

  // Get form values
  Const name = document.getElementById("name").value.trim().toUpperCase();
  Const email = document.getElementById("email").value;
  Const city = document.getElementById("city").value.toLowerCase();
  Const examDate = document.getElementById("examDate").value;

  // Perform additional validation here if needed
  // …

  // Create an object with the form data
  Const formData = {
   Name,
   Email,
   City,
   examDate,
  };

  // Send the data to the server for insertion into MongoDB
  Fetch("/api/addExamData", {
   Method: "POST",
   Headers: {
    "Content-Type": "application/json",
   },
   Body: JSON.stringify(formData),
  })
   .then((response) => response.json())
   .then((data) => {
    Console.log(data);
    // Handle success or display error messages to the user
   })
   .catch((error) => {
    Console.error("Error:", error);
    // Handle errors or display error messages to the user
   });
 });
});
```

Node.js Backend for MongoDB Insertion (server.js):

Assuming you have MongoDB set up and running, you can create a Node.js server to handle the form submission and insert data into the MongoDB collection. You'll need to use a MongoDB driver like Mongoose. Here's a simplified example:

```javascript
Const express = require("express");
Const mongoose = require("mongoose");
Const bodyParser = require("body-parser");

Const app = express();
Const port = process.env.PORT || 3000;
```

```
Mongoose.connect("mongodb://localhost/Exam", {
  useNewUrlParser: true,
  useUnifiedTopology: true,
});
Const db = mongoose.connection;

Db.on("error", console.error.bind(console, "MongoDB connection error:"));
Db.once("open", () => {
  Console.log("Connected to MongoDB");
});

Const examSchema = new mongoose.Schema({
  Name: String,
  Email: String,
  City: String,
  examDate: Date,
});

Const Exam = mongoose.model("Exam", examSchema);

App.use(bodyParser.json());

App.post("/api/addExamData", (req, res) => {
  Const { name, email, city, examDate } = req.body;

  Const newExam = new Exam({ name, email, city, examDate });

  newExam.save((err) => {
    if (err) {
```

```
   console.error("Error inserting exam data:", err);
   res.status(500).json({ error: "Error inserting exam data" });
  } else {
   Console.log("Exam data inserted successfully");
   Res.json({ message: "Exam data inserted successfully" });
  }
 });
});

App.listen(port, () => {
 Console.log(`Server is running on port ${port}`);
});
```

This example sets up an HTML form with validation rules, a frontend JavaScript file (`form.js`) to handle form submission and send data to the server, and a Node.js backend (`server.js`) to receive the data and insert it into a MongoDB collection named "Exam." Make sure to adjust the MongoDB connection URL and schema as needed for your environment.

32----

To perform the mentioned tasks using Node.js and Mongoose with the given employee collection, you can follow these steps:

1. Set up Mongoose and connect to your MongoDB database.

```javascript
```

```
Const mongoose = require(‘mongoose’);
Mongoose.connect(‘mongodb://localhost/main’, { useNewUrlParser: true, useUnifiedTopology: true });

Const db = mongoose.connection;
Db.on(‘error’, console.error.bind(console, ‘MongoDB connection error:’));
Db.once(‘open’, () => {
  Console.log(‘Connected to MongoDB’);
});

// Define the Employee schema
Const employeeSchema = new mongoose.Schema({
  Name: String,
  Age: Number,
  Position: String,
  Salary: Number,
});

Const Employee = mongoose.model(‘Employee’, employeeSchema);
```

2. Insert the provided data into the “main” collection:

```javascript
Const initialData = [
  // … The provided employee data …
];

Employee.insertMany(initialData, (err) => {
  If (err) {
    Console.error(‘Error inserting data:’, err);
  } else {
    Console.log(‘Data inserted successfully’);
  }
});
```

3. Perform the requested queries:

```javascript
// (1) Update or insert a document with age 43 and position “Senior Manager”
Employee.updateOne(
  { age: 43, position: ‘Senior Manager’ },
  { $set: { experience: 17 } },
  { upsert: true },
  (err, result) => {
    If (err) {
      Console.error(‘Error updating/inserting document:’, err);
    } else {
      Console.log(‘Document updated/inserted successfully’);
    }
  }
);

// (2) Find the employee with the highest salary
Employee.findOne().sort({ salary: -1 }).exec((err, employee) => {
  If (err) {
    Console.error(‘Error finding employee:’, err);
  } else {
    Console.log(`Employee with highest salary: ${employee.name}, Position: ${employee.position}`);
  }
});

// (3) Count documents where name contains “ric”
Employee.countDocuments({ name: /ric/ }, (err, count) => {
  If (err) {
    Console.error(‘Error counting documents:’, err);
  } else {
    Console.log(`Number of documents with “ric” in name: ${count}`);
  }
});

// (4) Increase the salary of employees with salary less than 45000 by 10%
Employee.updateMany({ salary: { $lt: 45000 } }, { $mul: { salary: 1.1 } }, (err, result) => {
  If (err) {
    Console.error(‘Error updating salaries:’, err);
  } else {
    Console.log(`Salaries updated for ${result.nModified} employees`);
  }
});

// (5) Find positions where name has 4 or 5 letters
Employee.distinct(‘position’, { name: /^[a-zA-Z]{4,5}$/ }, (err, positions) => {
  If (err) {
    Console.error(‘Error finding positions:’, err);
  } else {
    Console.log(`Positions with names containing 4 or 5 letters: ${positions}`);
  }
```

```
});
```

Make sure to adjust the MongoDB connection URL and schema as needed for your environment. This code sets up the Mongoose schema, inserts initial data, and performs the requested queries on the “main” collection.