```
// 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;
}
```

```
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.

21

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
```