Schemas Type

- There are the two main types of schemas used in Mongoose to define the structure of documents in MongoDB collections.
- Regular schemas define the top-level structure, while nested schemas allow for more complex document structures with subdocuments.

Regular Schemas

- Regular schemas are the primary way to define the structure of documents (data) in MongoDB collections using Mongoose.
- They define the shape of documents, including the fields and their types, default values, validation rules, and other options.
- Regular schemas are defined using the new mongoose.Schema() constructor function and then compiled into a model using mongoose.model().

```
const mongoose = require('mongoose');

// Define a regular schema
const userSchema = new mongoose.Schema({
   name: String,
   age: Number,
   email: {
              type: String,
              required: true,
               unique: true }
});

// Compile the schema into a model
const User = mongoose.model('User', userSchema);
```

Nested Schemas

- Nested schemas allow you to define subdocuments within a document's structure.
- They are useful for organizing data hierarchically and encapsulating related fields.
- Nested schemas are defined similarly to regular schemas but are embedded within another schema definition.

```
const mongoose = require('mongoose');

// Define a nested schema for address
const addressSchema = new mongoose.Schema({
  street: String,
  city: String,
  state: String,
  country: String
});

// Define a regular schema containing the nested schema
const userSchema = new mongoose.Schema({
  name: String,
  age: Number,
  email: {
           type: String,
           required: true,
           unique: true },
  address: addressSchema  // Nested schema as a field
});

// Compile the schema into a model
const User = mongoose.model('User', userSchema);
```

MVC Architecture

- In MERN (MongoDB, Express.js, React.js, Node.js) stack development, the MVC (Model-View-Controller) architecture pattern can be applied to organize code and structure the application.
- The MVC architecture helps in separating concerns, improving code organization, and promoting code reusability in MERN stack development.
- It provides a clear structure for building scalable and maintainable applications by dividing the application into three distinct layers: Model, View, and Controller.
- By adopting MVC architecture, developers can build robust, scalable, and maintainable applications that meet the requirements of modern web development.

How MVC architecture can be implemented in the context of a MERN stack application

Model (M)

- The Model represents the data layer of the application. It deals with data manipulation, validation, and database interaction.
- In a MERN stack application, MongoDB is commonly used as the database. Therefore, the Model layer includes interactions with the MongoDB database using Mongoose or other MongoDB libraries.
- Model components define the structure of data, handle data validation, and perform CRUD (Create, Read, Update, Delete) operations.
- Define MongoDB schemas and models using Mongoose. Handle database operations such as CRUD operations, data validation, and business logic.

View (V)

- The View represents the presentation layer of the application. It is responsible for rendering the user interface and displaying data to the user.
- In a MERN stack application, React.js is typically used to build the frontend user interface components.
- React components serve as the View layer in the MVC architecture. They receive data from the Controller and render it to the user interface.
- Create React components to render the user interface. Use JSX to describe the UI components and manage their state and lifecycle.

Controller (C)

- The Controller acts as an intermediary between the Model and the View. It handles user input, processes requests, and updates the Model accordingly.
- In a MERN stack application, Express.js is commonly used to build the backend server and handle HTTP requests.
- Controllers in Express.js define route handlers that receive HTTP requests, interact with the Model layer to perform business logic and database operations, and then send back responses to the client.
- Define Express.js route handlers to handle incoming HTTP requests. Use controllers to interact with the Model layer, perform data processing, and send responses back to the client.

Local and Global npm modules

In the context of Node.js and npm (Node Package Manager), there are two main types of modules: global modules and local modules.

Global Modules

- Global modules are installed globally on your system and can be accessed from any directory.
- These modules are typically installed using the -g flag with the npm install command.
- When you install a package globally, npm will place it in a central location on your system (such as /usr/local/lib/node_modules on Unix-like systems or C:\Users\{username}\AppData\Roaming on Windows).
- Global modules are often command-line tools or utilities that you want to be available system-wide.

  For example, you might install the nodemon package globally to automatically restart your Node.js server when files change:

  ```
  npm install -g nodemon
  ```

  Then you can use nodemon command from any directory to run your Node.js application.

Local Modules

- Local modules are installed locally in your project directory.
- They are typically listed in the dependencies or devDependencies section of your package.json file and are installed without the -g flag.
- Local modules are specific to each project and are not accessible outside of their project directory unless explicitly referenced.

  For example, you might install the express package locally for a Node.js web application:

  ```
  npm install express
  ```

  This command installs express in the node_modules directory within your project. You can then require and use express in your Node.js application files.

In summary, global modules are installed globally and are accessible system-wide, while local modules are installed locally in each project and are specific to that project.

Nodemon

- Nodemon is a utility tool for Node.js applications that helps developers in the development process by automatically restarting the Node application when file changes in the directory are detected.
- This eliminates the need for developers to manually stop and restart the server every time they make changes to their code, thereby improving the development workflow efficiency.
- Nodemon significantly streamlines the development process for Node.js applications by providing a convenient way to monitor file changes and automatically restart the server, thereby saving time and effort for developers.

How Nodemon works

- **Installation:** First, you need to install Nodemon globally or locally within your Node.js project. You can do this via npm:
  ```
  npm install -g nodemon
  ```
  or
  ```
  npm install nodemon --save-dev
  ```
  if you want to save it as a development dependency in your project's package.json file.

- **Usage:** Once installed, you can start your Node.js application with Nodemon instead of the regular node command.
  For example, if you have an app.js file as your entry point:
  ```
  nodemon app.js
  ```
  If running scripts is disabled on the system & you will get an error then use command
  ```
  Set-ExecutionPolicy RemoteSigned
  ```
- **Development:** Now, let's say you're working on your Node.js application, and you make changes to your code in app.js or any other relevant files.
- **Automatic Restart:** As soon as you save the changes, Nodemon detects the file modifications and automatically restarts the Node.js server. This means you can instantly see the effects of your changes without manually stopping and restarting the server.
- **Repeat:** You can continue this cycle of making changes, saving files, and seeing the changes reflected in your application without interruption, thanks to Nodemon's automatic restarting feature.

Example : CRUD Operations using mongoose, node & express

Sample Database

Database Name : empdata, Collection Name : employees

Documents:

```
[
{ id: 4, name: "mahesh", department: "front-end" },
{ id: 12, name: "stuti", department: "ui-ux" },
{ id: 19, name: 'neha', department: "hr" },
{id:8, name:"tushar", department:"back-end"}
]
```

Modules Installation

```
npm install express
npm install mongoose
npm install body-parser
```

App.js

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

const app = express();

app.use(bodyParser.json());
app.use(bodyParser.urlencoded());

mongoose.connect("mongodb://localhost:27017/empdata",
{
   serverSelectionTimeoutMS: 5000
})
.then(() => {
   console.log('Connected to MongoDB');
})
.catch((error) => {
   console.error('Error connecting to MongoDB:', error);
});
```

```
const empSchema = new mongoose.Schema({
    id: Number,
    name: String,
    department: String
});

const Employee = mongoose.model("Employee", empSchema);

app.get("/employee", (req, res) => {
    Employee.find({ id: { $gt: 10 } }).then(docs => {
            console.log(docs); // Log the result
            res.json(docs);
        })
        .catch(err => {
            console.error(err);
            res.status(500).json({ error: 'Internal server error' });
        });
});

// Endpoint to insert data into the MongoDB collection
app.post("/employee", (req, res) => {
    const { id, name, department } = req.body;

    // Create a new employee document
    const newEmployee = new Employee({
        id,        name,        department
    });

    // Save the new employee document to the database
    newEmployee.save().then(savedEmployee => {
            res.status(201).json(savedEmployee);
        })
        .catch(err => {
            console.error(err);
            res.status(500).json({ error: 'Failed to save employee' });
        });
});
```

```
// Endpoint to update data in the MongoDB collection
app.put("/employee/:id", (req, res) => {
   const id = req.params.id;
   const { name, department } = req.body;

   // Find the employee by id and update their details
   Employee.findOneAndUpdate({ id: id }, { name: name, department: department
}, { new: true })
      .then(updatedEmployee => {
         if (!updatedEmployee) {
            return res.status(404).json({ error: 'Employee not found' });
         }
         res.json(updatedEmployee);
      })
      .catch(err => {
         console.error(err);
         res.status(500).json({ error: 'Failed to update employee' });
      });
});

// Endpoint to delete data from the MongoDB collection based on ID
app.delete("/employee/:id", (req, res) => {
   const id = req.params.id;

   // Delete the employee with the specified ID
   Employee.findOneAndDelete({ id: id })
      .then(deletedEmployee => {
         if (!deletedEmployee) {
            return res.status(404).json({ error: 'Employee not found' });
         }
         res.json({ message: `Employee with ID ${id} deleted successfully` });
      })
      .catch(err => {
         console.error(err);
         res.status(500).json({ error: 'Failed to delete employee' });
      });
});
```

```
// Start the server
const PORT = process.env.PORT || 3000;
app.listen(PORT, () => {
   console.log(`Server is running on port ${PORT}`);
});
```

JSON input for create operation

```
{
   "id": emp_Id,
   "name": "Emp_Name",
   "department": "Emp_Department"
}
```

JSON input for update operation

```
{
   "name": "Updated Name",
   "department": "Updated Department"
}
```

Updating Multiple Documents in a collection

If you want to update multiple documents based on the basis of your requirements then use the following code snippet concept:

```
// Endpoint to update multiple documents based on department
app.put("/employee/department/:department", (req, res) => {
   const department = req.params.department;
   const { name } = req.body;
   // Update documents with matching department
   Employee.updateMany({ department: department }, { name: name })
      .then(result => {
         res.json({ message: `${result.modifiedCount} documents updated` });
      })
      .catch(err => {
         console.error(err);
         res.status(500).json({ error: 'Failed to update documents' });
      });
});
```

JSON input to update multiple documents

```
{
    "name": "Updated Name"
}
```

Delete Multiple Documents in a collection

If you want to delete multiple documents based on the basis of your requirements then use the following code snippet concept:

```
// Endpoint to delete data from the MongoDB collection based on department
app.delete("/employee/:department", (req, res) => {
    const department = req.params.department;

    // Delete all employees with the specified department
    Employee.deleteMany({ department: department })
        .then(deletedEmployees => {
            res.json({ message: `${deletedEmployees.deletedCount} employees
deleted with department '${department}'` });
        })
        .catch(err => {
            console.error(err);
            res.status(500).json({ error: 'Failed to delete employees' });
        });
});
```

Example : Simple MERN stack application to perform create, read, update, and delete operations

Required module installation for backend

```
npm install express mongoose body-parser cors
```

server.js (Backend Code)

```
const express = require('express');
const bodyParser = require('body-parser');
const cors = require('cors');
const mongoose = require('mongoose');

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

// MongoDB Connection
mongoose.connect('mongodb://localhost:27017/empdata',
{ serverSelectionTimeoutMS: 5000  });

// Employee Schema
const employeeSchema = new mongoose.Schema({
   id: Number,
   name: String,
   department: String,
   company: String,
   city: String
});

const Employee = mongoose.model('Employee', employeeSchema);

app.use(cors());        // Enables CORS for all routes
app.use(bodyParser.json());

// Routes
app.get('/employees', async (req, res) => {
   try {
      const employees = await Employee.find();
      res.json(employees);
```

```
  } catch (error) {
    res.status(500).json({ message: error.message });
  }
});

// Read single employee
app.get('/employees/:id', async (req, res) => {
  try {
    const employee = await Employee.findById(req.params.id);
    if (!employee) {
      return res.status(404).json({ message: 'Employee not found' });
    }
    res.json(employee);
  } catch (error) {
    res.status(500).json({ message: error.message });
  }
});


//create employee
app.post('/employees', async (req, res) => {
  const employee = new Employee(req.body);
  try {
    const newEmployee = await employee.save();
    res.status(201).json(newEmployee);
  } catch (error) {
    res.status(400).json({ message: error.message });
  }
});

// Update employee
app.put('/employees/:id', async (req, res) => {
  try {
    const employee = await Employee.findByIdAndUpdate(req.params.id,
req.body, { new: true });
    if (!employee) {
      return res.status(404).json({ message: 'Employee not found' });
```

```
        }
        res.json(employee);
    } catch (error) {
        res.status(400).json({ message: error.message });
    }
});

// Delete employee
app.delete('/employees/:id', async (req, res) => {
    try {
        const employee = await Employee.findByIdAndDelete(req.params.id);
        if (!employee) {
            return res.status(404).json({ message: 'Employee not found' });
        }
        res.json({ message: 'Employee deleted successfully' });
    } catch (error) {
        res.status(500).json({ message: error.message });
    }
});

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

Now come to front end, and create a react app using command

```
npx create-react-app employee-app
```

enter inside the react app directory and install the axios using `npm install axios`

App.js (Frontend Code)

```
import React, { useState, useEffect } from 'react';
import axios from 'axios';

function App() {
  const [employees, setEmployees] = useState([]);
  const [newEmployee, setNewEmployee] = useState({
    id: '',
    name: '',
    department: '',
    company: '',
    city: ''
  });
  const [selectedEmployee, setSelectedEmployee] = useState(null);

  useEffect(() => {
    fetchEmployees();
  }, []);

  const fetchEmployees = async () => {
    try {
      const response = await axios.get('http://localhost:5000/employees');
      setEmployees(response.data);
    } catch (error) {
      console.error('Error fetching employees:', error);
    }
  };

  const addEmployee = async () => {
    try {
      await axios.post('http://localhost:5000/employees', newEmployee);
      setNewEmployee({
        id: '',
        name: '',
        department: '',
        company: '',
        city: ''
      });
      fetchEmployees();
    } catch (error) {
      console.error('Error adding employee:', error);
    }
  };
```

```
  const updateEmployee = async () => {
    try {
      await      axios.put(`http://localhost:5000/employees/${selectedEmployee._id}`,
selectedEmployee);
      fetchEmployees();
    } catch (error) {
      console.error('Error updating employee:', error);
    }
  };

  const deleteEmployee = async () => {
    try {
      await axios.delete(`http://localhost:5000/employees/${selectedEmployee._id}`);
      fetchEmployees();
    } catch (error) {
      console.error('Error deleting employee:', error);
    }
  };

  return (
    <div>      <h1>Employee Management</h1>
      <div>       <h2>Add Employee</h2>
        <input      type="number"      placeholder="ID"      value={newEmployee.id}
onChange={e => setNewEmployee({ ...newEmployee, id: e.target.value })} />
        <input     type="text"     placeholder="Name"     value={newEmployee.name}
onChange={e => setNewEmployee({ ...newEmployee, name: e.target.value })} />
        <input                type="text"                placeholder="Department"
value={newEmployee.department}     onChange={e     =>     setNewEmployee({
...newEmployee, department: e.target.value })} />
        <input type="text" placeholder="Company" value={newEmployee.company}
onChange={e => setNewEmployee({ ...newEmployee, company: e.target.value })}
/>
        <input     type="text"     placeholder="City"     value={newEmployee.city}
onChange={e => setNewEmployee({ ...newEmployee, city: e.target.value })} />
        <button onClick={addEmployee}>Add</button>
      </div>
      <div>       <h2>Employee List</h2>
        <ul>        {employees.map(employee => (
            <li key={employee._id}>
              {employee.name} - {employee.department} - {employee.company} -
{employee.city}
              <button onClick={() => setSelectedEmployee(employee)}>Edit</button>
            </li>
          ))}
        </ul>
      </div>      {selectedEmployee && (
        <div>
          <h2>Edit Employee</h2>
          <input type="number" placeholder="ID" value={selectedEmployee.id}
disabled />
          <input type="text" placeholder="Name" value={selectedEmployee.name}
onChange={e => setSelectedEmployee({ ...selectedEmployee, name: e.target.value
})} />
          <input type="text" placeholder="Department"
value={selectedEmployee.department} onChange={e => setSelectedEmployee({
...selectedEmployee, department: e.target.value })} />
          <input type="text" placeholder="Company"
value={selectedEmployee.company} onChange={e => setSelectedEmployee({
...selectedEmployee, company: e.target.value })} />
          <input type="text" placeholder="City" value={selectedEmployee.city}
onChange={e => setSelectedEmployee({ ...selectedEmployee, city: e.target.value })}
/>
          <button onClick={updateEmployee}>Update</button>
          <button onClick={deleteEmployee}>Delete</button>
        </div>
      )}
    </div>
  );}
export default App;
```

Now, use node server.js in one terminal and npm start command in another terminal to run the project.

Assignment

Schema Modification

Modify the employee schema to include a new field called email. Update both the frontend and backend code to accommodate this change. Describe the steps you took and explain any challenges you encountered during the modification process.

Form Validation

Implement form validation on the frontend to ensure that all fields are required when adding a new employee. Additionally, validate the format of the email field to ensure it follows the standard email format. Describe your validation approach and provide examples of error messages displayed to the user for invalid inputs.