



**CHAITANYA BHARATHI
INSTITUTE OF TECHNOLOGY (A)**
Kokapet(Village), Gandipet, Hyderabad, Telangana-500075. www.cbti.ac.in

COMMITTED TO
RESEARCH,
INNOVATION AND
EDUCATION

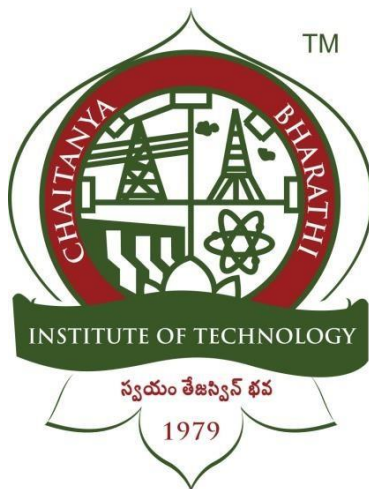
44
years

Laboratory Manual
For
ENTERPRISE APPLICATION DEVELOPMENT LAB
(Professional Elective – II)

Course Code: 20CSE17

Faculty: Mr.Venkata Siva Rao.Alapati

VI Semester
Undergraduate Course
In
COMPUTER SCIENCE AND ENGINEERING



Department of Computer Science and Engineering
Chaitanya Bharathi Institute of Technology (A),
Gandipet, Hyderabad – 500075

20CSE17
ENTERPRISE APPLICATION DEVELOPMENT LAB
(Professional Elective – II)

| | |
|--------------------------------|------------------|
| Instruction | 2 Hours per week |
| Duration of End Examination | 3 Hours |
| Semester End Examination | 50 Marks |
| Continuous Internal Evaluation | 50 Marks |
| Credits | 1 |

Pre-requisites: Internet and web technologies, OOPs, Database management systems.

Course Objectives: The objectives of this course are,

1. To acquire knowledge on MongoDB, ReactJS, Express, Node.js and Angular2 to develop webapplications.
2. Ability to develop dynamic web content using web frameworks.
3. To understand the design and development process of a complete web application.

Course Outcomes: On successful completion of the course, students will be able to,

1. Prepare database connections with application servers.
2. Design user interfaces using ReactJS.
3. Construct strong expertise on Express framework to develop responsive web applications.
4. Create server side applications using Node.js
5. Develop SPA using Angular 2.
6. Invent next culture-shifting web applications.

List of Programs:

1. Installation, configuration and connection establishment of MongoDB.
2. CRUD operations on MongoDB.
3. Building and Deploying React App.
4. Demonstration of component intercommunication using ReactJS
5. Create Express application,
6. Demonstration of authentication and authorization using Express.
7. Data access using Node.js
8. Create a form to edit the data using Angular2
9. A case study on a single platform for all financial data for NSE India.

Textbook:

1. Amos Q. Haviv, MEAN Web Development, Second Edition, Packt Publications, November 2016
2. Vasan Subramanian, "Pro MERN Stack, Full Stack Web App Development with Mongo, Express, React, and Node", 2nd Edition, APRESS.

Suggested Reading:

1. Shelly Powers, "Learning Node: Moving to the Server-Side", 2nd Edition, O'REILLY, 2016.
2. Simon D. Holmes and Clive Harber, "Getting MEAN with Mongo, Express, Angular, and Node", Second Edition, Manning Publications, 2019.
3. Brad Dayley, "Node.js, MongoDB and Angular Web Development", 2nd Edition, Addison-WesleyProfessional, 2017.

Online Resources:

1. <https://www.mongodbtutorial.org/mongodb-crud/>
2. <https://reactjs.org/tutorial/tutorial.html>
3. <https://www.javatpoint.com/expressjs-tutorial>
4. <https://www.javatpoint.com/nodejs-tutorial>
5. <https://angular-training-guide.rangle.io/>

Index

| S.No | Name of the Experiment | Page Number |
|------|----------------------------------------------------------------------|-------------|
| 1. | Installation, configuration and connection establishment of MongoDB. | 1 |
| 2. | CRUD operations on MongoDB. | 3 |
| 3. | Building and Deploying React App. | 9 |
| 4. | Demonstration of component intercommunication using ReactJS | 11 |
| 5. | Create Express application | 14 |
| 6. | Demonstration of authentication and authorization using Express. | 16 |
| 7. | Data access using Node.js | 19 |
| 8. | Create a form to edit the data using Angular2 | 21 |

EXPERIMENT -1

Aim: Installation, configuration and connection establishment of MongoDB.

Description:

MongoDB is a popular NoSQL database that stores data in JSON-like documents with dynamic schemas, making it highly flexible and scalable. It is widely used for its high performance, scalability, and ease of use.

Some of the key features of MongoDB include:

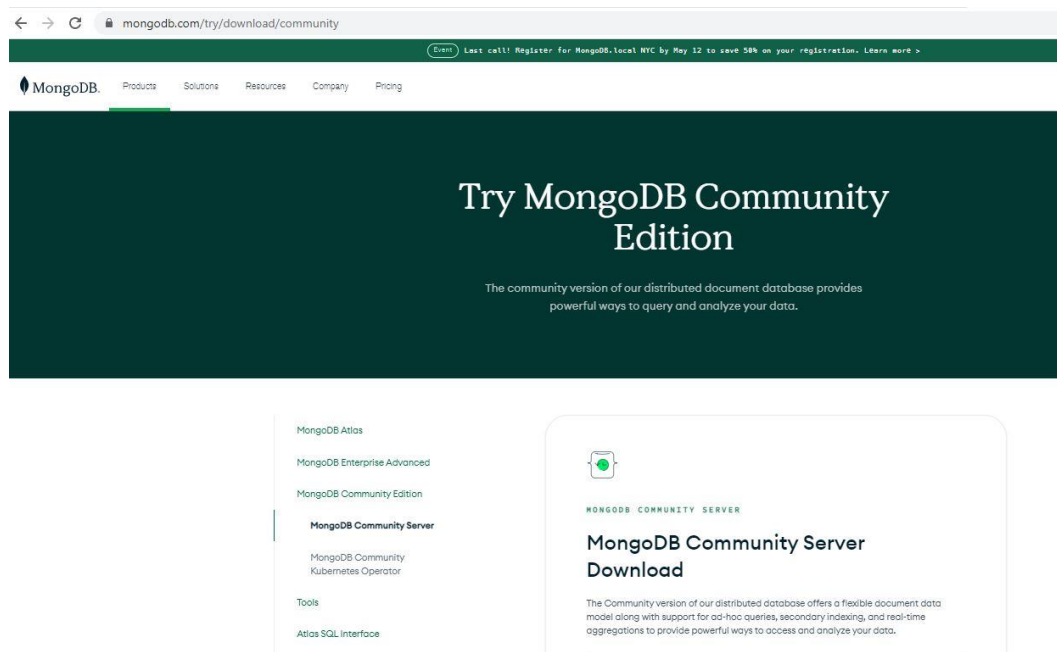
- Document-oriented: MongoDB stores data in JSON-like documents, which makes it easy to work with and highly flexible. Documents can have different fields and structures, allowing for dynamic schemas.
- Scalable: MongoDB can scale horizontally across multiple servers, allowing it to handle large amounts of data and high traffic loads. It also has built-in sharing and replication features for automatic distribution and redundancy of data.
- High performance: MongoDB uses a variety of optimization techniques, including indexing, caching, and native aggregation support, to provide fast read and write performance.
- Querying and aggregation: MongoDB supports a powerful query language and aggregation framework, making it easy to search, filter, and aggregate data in real-time.
- Schema validation: MongoDB allows you to define validation rules for your documents, ensuring that data is always consistent and valid.
- Flexible data model: MongoDB allows you to model complex data structures with ease, including hierarchical data, arrays, and nested documents.

Demonstration:

Here are the steps to install, configure and establish a connection with MongoDB:

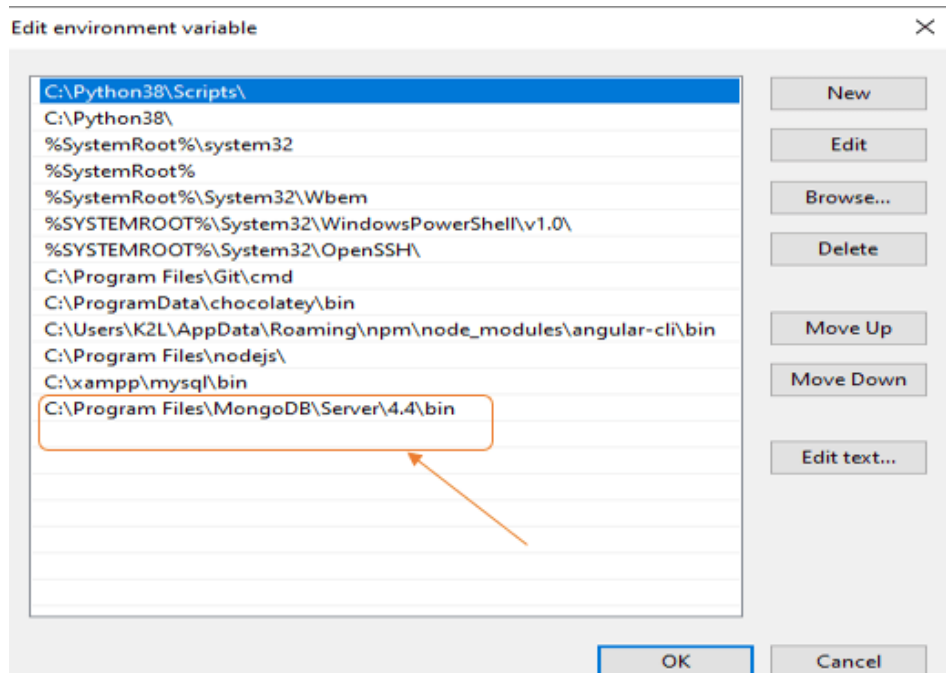
Step 1: Download and Install MongoDB

- Visit the MongoDB download page(<https://www.mongodb.com/try/download/community>)
- Download the Community server edition of MongoDB for your operating system.
- Follow the installation instructions for your operating system.



Step 2: Configure MongoDB

- Now we go to location where MongoDB installed in step-5 in our system and copy the binpath:
- Now, to create an environment variable open system properties << Environment Variable << System variable << path << Edit Environment variable and paste the copied link to your environment system and click Ok:



Step 3: Connection Establishment

- Open a terminal or command prompt and start the MongoDB server with the following command:

mongosh

```
C:\Users\CBIT>mongosh
Current Mongosh Log ID: 63e9c1371d6927c5720d55d3
Connecting to:      mongodb://127.0.0.1:27017/?directConnection=true&serverSelectionTimeoutMS=2000&appName=mongosh+1
.6.2
Using MongoDB:      6.0.3
Using Mongosh:      1.6.2

For mongosh info see: https://docs.mongodb.com/mongodb-shell/

-----
  The server generated these startup warnings when booting
  2023-01-25T15:03:02.375+05:30: Access control is not enabled for the database. Read and write access to data and conf
  igation is unrestricted
  -----
  Enable MongoDB's free cloud-based monitoring service, which will then receive and display
  metrics about your deployment (disk utilization, CPU, operation statistics, etc).

  The monitoring data will be available on a MongoDB website with a unique URL accessible to you
  and anyone you share the URL with. MongoDB may use this information to make product
  improvements and to suggest MongoDB products and deployment options to you.

  To enable free monitoring, run the following command: db.enableFreeMonitoring()
  To permanently disable this reminder, run the following command: db.disableFreeMonitoring()
  -----
```

EXPERIMENT -2

Aim: CRUD operations on MongoDB.

Description:

CRUD operations describe the conventions of a user-interface that let users view, search, and modify parts of the database.

MongoDB documents are modified by connecting to a server, querying the proper documents, and then changing the setting properties before sending the data back to the database to be updated. CRUD is data-oriented, and it's standardized according to HTTP action verbs.

When it comes to the individual CRUD operations:

- The Create operation is used to insert new documents in the MongoDB database.
- The Read operation is used to query a document in the database.
- The Update operation is used to modify existing documents in the database.
- The Delete operation is used to remove documents in the database.

Demonstration:

➤ **Create Operations**

For MongoDB CRUD, if the specified collection doesn't exist, the create operation will create the collection when it's executed. Create operations in MongoDB target a single collection, not multiple collections. Insert operations in MongoDB are atomic on a single document level.

- MongoDB provides two different create operations that you can use to insert documents into a collection:
 - **db.collection.insertOne()**
 - **db.collection.insertMany()**
- **insertOne()**
 - It allows you to insert one document into the collection.
 - For this example, we're going to work with a collection called RecordsDB. We can insert a single entry into our collection by calling the insertOne() method on RecordsDB.
 - We then provide the information we want to insert in the form of key-value pairs, establishing the schema.

```
db.RecordsDB.insertOne({
  name: "Marsh",
  age: "6 years",
  species: "Dog",
  ownerAddress: "380 W. Fir Ave",
  chipped: true
})
```

- If the create operation is successful, a new document is created. The function will return an object where "acknowledged" is "true" and "insertID" is the newly created "ObjectId."

Output:

```
db.RecordsDB.insertOne({
... name: "Marsh",
... age: "6 years",
... species: "Dog",
... ownerAddress: "380 W. Fir Ave",
... chipped: true
... })
{
  "acknowledged" : true,
  "insertedId" : ObjectId("5fd989674e6b9ceb8665c57d")
}
```

• insertMany()

- It's possible to insert multiple items at one time by calling the insertMany() method on the desired collection.
- In this case, we pass multiple items into our chosen collection (RecordsDB) and separate them by commas. Within the parentheses, we use brackets to indicate that we are passing in a list of multiple entries.
- This is commonly referred to as a nested method.

```
db.RecordsDB.insertMany([
  {
    name: "Marsh",
    age: "6 years",
    species: "Dog",
    ownerAddress: "380 W. Fir Ave",
    chipped: true
  },
  {
    name: "Kitana",
    age: "4 years",
    species: "Cat",
    ownerAddress: "521 E. Cortland",
    chipped: true
  }
])
```

Output:

```
db.RecordsDB.insertMany([
  { name: "Marsh", age: "6 years", species: "Dog",
    ownerAddress: "380 W. Fir Ave", chipped: true },
  { name: "Kitana", age: "4 years",
    species: "Cat", ownerAddress: "521 E. Cortland", chipped: true }
])
{
  "acknowledged" : true,
  "insertedIds" : [
    ObjectId("5fd98ea9ce6e8850d88270b4"),
    ObjectId("5fd98ea9ce6e8850d88270b5")
  ]
}
```

➤ Read Operations

The read operations allow you to supply special query filters and criteria that let you specify which documents you want. The MongoDB documentation contains more information on the available query filters. Query modifiers may also be used to change how many results are returned.

- MongoDB has two methods of reading documents from a collection:

➤ **db.collection.find()**

➤ **db.collection.findOne()**

find()

- In order to get all the documents from a collection, we can simply use the find() method on our chosen collection.
- Executing just the find() method with no arguments will return all records currently in the collection.

```
db.RecordsDB.find()
{ "_id" : ObjectId("5fd98ea9ce6e8850d88270b5"), "name" : "Kitana", "age" : "4 years", "species" : "Cat",
  "ownerAddress" : "521 E. Cortland", "chipped" : true }
{ "_id" : ObjectId("5fd993a2ce6e8850d88270b7"), "name" : "Marsh", "age" : "6 years", "species" : "Dog",
  "ownerAddress" : "380 W. Fir Ave", "chipped" : true }
{ "_id" : ObjectId("5fd993f3ce6e8850d88270b8"), "name" : "Loo", "age" : "3 years", "species" : "Dog",
  "ownerAddress" : "380 W. Fir Ave", "chipped" : true }
{ "_id" : ObjectId("5fd994efce6e8850d88270ba"), "name" : "Kevin", "age" : "8 years", "species" : "Dog",
  "ownerAddress" : "900 W. Wood Way", "chipped" : true }
```

Here we can see that every record has an assigned “ObjectId” mapped to the “_id” key.

- If you want to get more specific with a read operation and find a desired subsection of the records, you can use the previously mentioned filtering criteria to choose what results should be returned.
- One of the most common ways of filtering the results is to search by value.

Output:

```
db.RecordsDB.find({"species":"Cat"})
{ "_id" : ObjectId("5fd98ea9ce6e8850d88270b5"), "name" : "Kitana", "age" : "4 years", "species" : "Cat",
  "ownerAddress" : "521 E. Cortland", "chipped" : true }
```

findOne()

- In order to get one document that satisfies the search criteria, we can simply use the findOne() method on our chosen collection.
- If multiple documents satisfy the query, this method returns the first document according to the natural order which reflects the order of documents on the disk.
- If no documents satisfy the search criteria, the function returns null. The function takes the following form of syntax.

```
db.{collection}.findOne({query}, {projection})
```

Let's take the following collection—say, RecordsDB, as an example.

```
{ "_id" : ObjectId("5fd98ea9ce6e8850d88270b5"), "name" : "Kitana", "age" : "8 years", "species" : "Cat",
  "ownerAddress" : "521 E. Cortland", "chipped" : true }
{ "_id" : ObjectId("5fd993a2ce6e8850d88270b7"), "name" : "Marsh", "age" : "6 years", "species" : "Dog",
  "ownerAddress" : "380 W. Fir Ave", "chipped" : true }
{ "_id" : ObjectId("5fd993f3ce6e8850d88270b8"), "name" : "Loo", "age" : "3 years", "species" : "Dog",
  "ownerAddress" : "380 W. Fir Ave", "chipped" : true }
{ "_id" : ObjectId("5fd994efce6e8850d88270ba"), "name" : "Kevin", "age" : "8 years", "species" : "Dog",
  "ownerAddress" : "900 W. Wood Way", "chipped" : true }
```


And, we run the following line of code:

```
db.RecordsDB.find({"age":"8 years"})
```

➤ We would get the following result:

```
{ "_id" : ObjectId("5fd98ea9ce6e8850d88270b5"), "name" : "Kitana", "age" : "8 years", "species" : "Cat",  
  "ownerAddress" : "521 E. Cortland", "chipped" : true }
```

Notice that even though two documents meet the search criteria, only the first document that matches the search condition is returned.

➤ Update Operations

Like create operations, update operations operate on a single collection, and they are atomic at a single document level.

An update operation takes filters and criteria to select the documents you want to update.

You should be careful when updating documents, as updates are permanent and can't be rolled back. This applies to delete operations as well.

For MongoDB CRUD, there are three different methods of updating documents:

- **db.collection.updateOne()**
- **db.collection.updateMany()**
- **db.collection.replaceOne()**

updateOne()

➤ We can update a currently existing record and change a single document with an update operation. To do this, we use the updateOne() method on a chosen collection, which here is "RecordsDB."

➤ To update a document, we provide the method with two arguments: an update filter and an update action.

➤ The update filter defines which items we want to update, and the update action defines how to update those items. We first pass in the update filter.

➤ Then, we use the "\$set" key and provide the fields we want to update as a value. This method will update the first record that matches the provided filter.

```
db.RecordsDB.updateOne({ name: "Marsh" }, { $set: { ownerAddress: "451 W. Coffee St. A204" } })  
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }  
{ "_id" : ObjectId("5fd993a2ce6e8850d88270b7"), "name" : "Marsh", "age" : "6 years", "species" : "Dog",  
  "ownerAddress" : "451 W. Coffee St. A204", "chipped" : true }
```

updateMany()

➤ updateMany() allows us to update multiple items by passing in a list of items, just as we did when inserting multiple items. This update operation uses the same syntax for updating a single document.

```
db.RecordsDB.updateMany({ species: "Dog" }, { $set: { age: "5" } })  
{ "acknowledged" : true, "matchedCount" : 3, "modifiedCount" : 3 }
```

Output:

```
db.RecordsDB.find()
{ "_id" : ObjectId("5fd98ea9ce6e8850d88270b5"), "name" : "Kitana", "age" : "4 years", "species" : "Cat",
  "ownerAddress" : "521 E. Cortland", "chipped" : true }
{ "_id" : ObjectId("5fd993a2ce6e8850d88270b7"), "name" : "Marsh", "age" : "5", "species" : "Dog",
  "ownerAddress" : "451 W. Coffee St. A204", "chipped" : true }
{ "_id" : ObjectId("5fd993f3ce6e8850d88270b8"), "name" : "Loo", "age" : "5", "species" : "Dog",
  "ownerAddress" : "380 W. Fir Ave", "chipped" : true }
{ "_id" : ObjectId("5fd994efce6e8850d88270ba"), "name" : "Kevin", "age" : "5", "species" : "Dog",
  "ownerAddress" : "900 W. Wood Way", "chipped" : true }
```

replaceOne()

- The replaceOne() method is used to replace a single document in the specified collection.
- replaceOne() replaces the entire document, meaning fields in the old document not contained in the new will be lost.

```
db.RecordsDB.replaceOne({ name: "Kevin"}, { name: "Maki"})
{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
```

Output:

```
db.RecordsDB.find()
{ "_id" : ObjectId("5fd98ea9ce6e8850d88270b5"), "name" : "Kitana", "age" : "4 years", "species" : "Cat",
  "ownerAddress" : "521 E. Cortland", "chipped" : true }
{ "_id" : ObjectId("5fd993a2ce6e8850d88270b7"), "name" : "Marsh", "age" : "5", "species" : "Dog",
  "ownerAddress" : "451 W. Coffee St. A204", "chipped" : true }
{ "_id" : ObjectId("5fd993f3ce6e8850d88270b8"), "name" : "Loo", "age" : "5", "species" : "Dog",
  "ownerAddress" : "380 W. Fir Ave", "chipped" : true }
{ "_id" : ObjectId("5fd994efce6e8850d88270ba"), "name" : "Maki" }
```

➤ Delete Operations

Delete operations operate on a single collection, like update and create operations. Delete operations are also atomic for a single document. You can provide delete operations with filters and criteria in order to specify which documents you would like to delete from a collection. The filter options rely on the same syntax that read operations utilize.

MongoDB has two different methods of deleting records from a collection:

- db.collection.deleteOne()
- db.collection.deleteMany()

deleteOne()

- deleteOne() is used to remove a document from a specified collection on the MongoDB server.
- A filter criteria is used to specify the item to delete. It deletes the first record that matches the provided filter.

```
db.RecordsDB.deleteOne({ name: "Maki"})
{ "acknowledged" : true, "deletedCount" : 1 }
```

Output:

```
db.RecordsDB.find()
{ "_id" : ObjectId("5fd98ea9ce6e8850d88270b5"), "name" : "Kitana", "age" : "4 years", "species" : "Cat",
  "ownerAddress" : "521 E. Cortland", "chipped" : true }
{ "_id" : ObjectId("5fd993a2ce6e8850d88270b7"), "name" : "Marsh", "age" : "5", "species" : "Dog",
  "ownerAddress" : "451 W. Coffee St. A204", "chipped" : true }
{ "_id" : ObjectId("5fd993f3ce6e8850d88270b8"), "name" : "Loo", "age" : "5", "species" : "Dog",
  "ownerAddress" : "380 W. Fir Ave", "chipped" : true }
```

deleteMany()

- deleteMany() is a method used to delete multiple documents from a desired collection with a single delete operation.
- A list is passed into the method and the individual items are defined with filter criteria as in deleteOne().

```
db.RecordsDB.deleteMany({species:"Dog"})
{ "acknowledged" : true, "deletedCount" : 2 }
```

Output:

```
db.RecordsDB.find()
{ "_id" : ObjectId("5fd98ea9ce6e8850d88270b5"), "name" : "Kitana", "age" : "4 years", "species" : "Cat",
  "ownerAddress" : "521 E. Cortland", "chipped" : true }
```

EXPERIMENT -3

Aim: Building and Deploying React App.

Description:

A React app is a web application built using the React JavaScript library. React, developed by Facebook, is a popular open-source library for building user interfaces, particularly for single-page applications (SPAs).

React follows a component-based architecture, where the UI is divided into reusable components that manage their own state and can be composed to build complex user interfaces. These components are written using JavaScript (JSX syntax), which combines HTML-like syntax with JavaScript code.

Key features of React that make it a powerful framework for building web applications include:

Virtual DOM: React uses a virtual representation of the DOM (Document Object Model) called the Virtual DOM. This allows React to efficiently update and render only the necessary parts of the UI, resulting in improved performance.

Component Reusability: React promotes reusability by breaking the UI into independent and reusable components. These components can be composed and combined to build complex interfaces, making the code more modular and maintainable.

One-Way Data Flow: React follows a one-way data flow, where data is passed from parent components to child components through props. This helps to maintain a predictable state and makes it easier to debug and reason about the application.

Declarative Syntax: React uses a declarative approach, where you define how the UI should look based on the current state, and React takes care of updating the actual DOM to reflect the desired state. This simplifies the development process and reduces the amount of manual DOM manipulation.

React Router: React Router is a popular routing library for React applications. It allows developers to implement client-side routing, enabling navigation and rendering of different components based on the URL without requiring a full page reload.

React Context: React Context provides a mechanism for sharing state between components without passing props through multiple levels. It allows data to be accessed by any component within a defined context, making it useful for managing global state or sharing data that is used across different components.

React Hooks: Introduced in React 16.8, Hooks allow developers to use state and other React features in functional components, removing the need to write class components. Hooks provide a simpler and more intuitive way to manage state and lifecycle in React applications.

React can be combined with other libraries and tools, such as Redux for centralized state management, Axios for handling API requests, and many UI libraries and frameworks for styling and component design.

Demonstration:

Building and deploying a React app involves several steps, including setting up the development environment, creating a production build, and deploying the application to a hosting provider.

- Set up the Development Environment:

- Install Node.js: Visit the official Node.js website (<https://nodejs.org>) and download the latest LTS version. Follow the installation instructions for your operating system.
- Create a new React app: Open a terminal or command prompt and run the following command to create a new React app using Create React App: **npx create-react-app my-app**
- Navigate to the app directory: **cd my-app**
- Start the development server: **npm start**

This will launch the app in development mode, and you can view it in your browser at <http://localhost:3000>.

- Develop the React App:

Open your preferred code editor and modify the code in the `src` directory to develop your React app. Add components, styles, and functionality according to your project requirements.

- Test the React App:

Write unit tests for your components using testing frameworks like Jest or React Testing Library. Run the tests to ensure the app functions as expected: **npm test**

- Create a Production Build:

When you're ready to deploy your app, generate a production build by running the following command: **npm run build**

- This command creates an optimized and minified version of your app in the `build` directory.

- Choose a Hosting Provider:

Select a hosting provider to deploy your React app. Some popular options include Netlify, Vercel, Firebase, AWS Amplify, and GitHub Pages. Compare their features, pricing, and deployment options to choose the one that suits your needs.

- Deploy the React App:

- Follow the hosting provider's documentation to deploy your React app. Generally, you need to sign up for an account, create a new project, and configure the deployment settings. Most hosting providers allow you to deploy directly from a Git repository or by uploading the build files.
- Once deployed, the hosting provider will provide you with a URL where your React app is accessible to the public.

EXPERIMENT -4

Aim: Demonstration of component intercommunication using ReactJS

Description:

Component intercommunication in ReactJS refers to the ability of different components to communicate and share data with each other. It involves passing data, invoking functions, and managing state between components to enable coordination and interaction within the application.

There are several approaches to achieve component intercommunication in ReactJS:

- **Props:**
 - Props (short for properties) are used to pass data from a parent component to its child components.
 - The parent component can pass data or functions as props, which can then be accessed and used by the child components. This allows for one-way communication from parent to child.
- **Callbacks:**
 - Callbacks are functions passed from a parent component to its child components as props.
 - Child components can invoke these callbacks, passing data back to the parent component. This enables two-way communication between parent and child components.
- **Context API:**
 - React's Context API allows for the creation of a global state that can be accessed by multiple components throughout the component tree.
 - It eliminates the need to pass props through intermediate components.
 - Context provides a way to share data between components without explicitly passing it down as props.
- **State Management Libraries:**
 - State management libraries like Redux or MobX can be used to manage the application state and facilitate intercommunication between components.
 - These libraries provide a central store to store and update data, which can be accessed by any component within the application.
- **Event Emitter/Subscriber Pattern:**
 - An event emitter/subscriber pattern can be implemented using libraries like EventEmitter3 or PubSubJS.
 - Components can emit events or subscribe to events, allowing for communication between unrelated components.

These approaches can be combined or used independently based on the complexity and requirements of the application. The choice depends on factors such as the component hierarchy, the amount of shared data, and the desired level of decoupling between components.

Demonstration:

Let's walk through a demonstration of component intercommunication using ReactJS.

Suppose we have two components: ParentComponent and ChildComponent. The ParentComponent will pass data and a callback function to the ChildComponent, which will then use that data and invoke the callback function to communicate back to the parent.

Here's an example implementation:

ParentComponent.js:

```
import React, { useState } from 'react';
import ChildComponent from './ChildComponent';

const ParentComponent = () => {
  const [message, setMessage] = useState("");

  const handleMessageChange = (newMessage) => {
    setMessage(newMessage);
  };

  return (
    <div>
      <h2>Parent Component</h2>
      <ChildComponent message={message} onMessageChange={handleMessageChange} />
      <p>Message from ChildComponent: {message}</p>
    </div>
  );
};

export default ParentComponent;
```

ChildComponent.js:

```
import React from 'react';

const ChildComponent = ({ message, onMessageChange }) => {
  const handleChange = (event) => {
    const newMessage = event.target.value;
    onMessageChange(newMessage);
  };

  return (
    <div>
      <h3>Child Component</h3>
      <input type="text" value={message} onChange={handleChange} />
    </div>
  );
};

export default ChildComponent;
```

- In the ParentComponent, we define the message state using the useState hook. We also define the handleMessageChange callback function, which updates the message state when invoked.
- The ParentComponent renders the ChildComponent and passes the message state and the handleMessageChange callback as props.
- In the ChildComponent, we destructure the message and onMessageChange props. When the input field value changes, we invoke the handleChange function, which extracts the new message from the event and calls the onMessageChange callback with the new message as an argument.
- As a result, when the user types in the input field of the ChildComponent, the handleChange function is triggered, and the updated message is passed back to the ParentComponent through the onMessageChange callback. The ParentComponent updates its message state accordingly and renders the new message value.

This demonstrates how data and intercommunication can flow between parent and child components in ReactJS.

EXPERIMENT -5

Aim: Create Express application

Description:

Express.js is a popular, open-source, minimalist web framework for Node.js. It provides a robust set of features for developing web and mobile applications. Express is designed to be lightweight, flexible, and easy to use, making it a popular choice among developers who want to build scalable, high-performance web applications.

Some of the key features of Express include:

- **Routing:** Express provides a simple and intuitive way to define routes for handling HTTP requests. It allows developers to create custom middleware functions to handle specific requests, and to chain multiple middleware functions together to handle more complex use cases.
- **Middleware:** Express allows developers to use middleware functions to perform various tasks during the request-response cycle, such as logging, authentication, and data validation. It also supports third-party middleware modules that can be easily integrated into an Express application.
- **Templating:** Express supports a variety of templating engines, including EJS, Pug, and Handlebars. These engines make it easy to generate dynamic HTML pages by merging data with pre-defined templates.
- **Error handling:** Express provides robust error handling capabilities, allowing developers to catch and handle errors in a consistent and reliable manner. It also includes built-in error handling middleware that can be used to catch and handle common types of errors.
- **Static file serving:** Express makes it easy to serve static files, such as images, CSS, and JavaScript, from a directory on the server. This allows developers to build front-end applications that can be easily deployed to a web server.

Express has a large and active community of developers who contribute to its ongoing development and support. It is widely used by companies of all sizes, from small startups to large enterprises, to build high-performance web and mobile applications.

Demonstration:

Step 1: Install Node.js and npm:

- You'll need Node.js and npm (Node Package Manager) installed on your machine to use Express.
- You can download them from the official website:
<https://nodejs.org/en/download/>

Step 2: Create a new directory for your Express application:

- Open up your terminal and create new directory for your application, e.g. `mkdir my-express-app`.
- Navigate to the directory using `cd my-express-app`.

```
C:\Users\CBIT\Documents>node -v
v18.14.2

C:\Users\CBIT\Documents>npm -v
9.5.0

C:\Users\CBIT\Documents>mkdir my_express_app

C:\Users\CBIT\Documents>cd my_express_app
```

Step 3: Initialize your project:

- Run npm init in your terminal and follow the prompts to initialize your project. This will create a package.json file in your project directory.

```
C:\Users\CBIT\Documents\my_express_app>npm init -y
Wrote to C:\Users\CBIT\Documents\my_express_app\package.json:

{
  "name": "my_express_app",
  "version": "1.0.0",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC",
  "keywords": [],
  "description": ""
}

C:\Users\CBIT\Documents\my_express_app>
```

Step 4: Install Express:

- Run npm install express in your terminal to install Express and add it to your project's dependencies.

```
C:\Users\CBIT\Documents\my_express_app>npm install express
added 57 packages, and audited 58 packages in 12s

7 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities

C:\Users\CBIT\Documents\my_express_app>
```

Step 5: Create a new file for your server:

- Create a new file in your project directory, e.g. server.js. This will be the entry point for your application.

Step 6: Set up your Express application:

- In server.js, require Express and create an instance of it:

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

```
const app = express();
```

Step 7: Define routes:

- You can define routes using the app.get(), app.post(), app.put(), app.delete(), and other methods.

- For example, here's how you can define a basic route that sends a message to the client:

```
app.get('/', (req, res) => { res.send('Hello, World!'); });
```

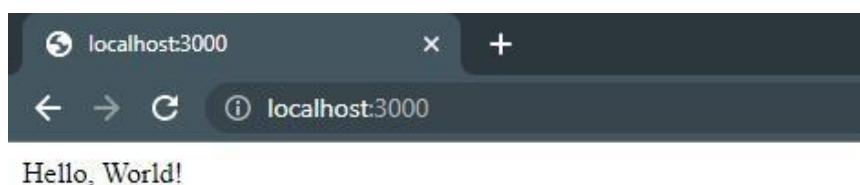
Step 8: Start your server:

- Finally, start your server by listening on a port.

- For example, you can listen on port 3000 like this:

```
app.listen(3000, () => { console.log('Server is running on port 3000'); });
```

```
C:\Users\CBIT\Documents\my_express_app>node server.js
Server is running on port 3000
```



EXPERIMENT -6

Aim: Demonstration of authentication and authorization using Express.

Description:

- **Authentication:**

Authentication is the process of verifying the identity of a user or client. It ensures that the user is who they claim to be before granting access to protected resources. Common authentication methods include username and password, social login (OAuth), and token-based authentication.

In a typical authentication flow:

- The user provides their credentials (e.g., username and password) to the server.
- The server verifies the credentials. This may involve comparing the provided password against a stored and hashed password.
- If the credentials are valid, the server generates a token or session identifier and sends it back to the client.
- The client includes the token or session identifier in subsequent requests to authenticate itself to the server.

- **Authorization:**

Authorization determines the access rights and permissions granted to a user after they have been authenticated. It involves defining and enforcing rules that restrict or grant access to specific resources or actions within the application.

Authorization can be based on various factors, such as user roles, user groups, or specific permissions associated with the user. It ensures that authenticated users only have access to the resources they are allowed to use.

In a typical authorization flow:

- Once a user is authenticated, the server receives the authentication token or session identifier in subsequent requests.
 - The server validates the token or session identifier to ensure its authenticity and integrity.
 - Based on the authenticated user's identity and associated permissions, the server determines whether the user is authorized to access the requested resource or perform the requested action.
 - If authorized, the server processes the request and returns the requested data or performs the requested action. Otherwise, it returns an error or denies access.
- By combining authentication and authorization, an Express application can ensure secure access to protected resources, control user permissions, and protect sensitive data from unauthorized access.

Demonstration:

Here's an example of implementing authentication and authorization using Express:

Step 1: Set up the project:

- Create a new directory for your project and navigate to it in the terminal.
- Run `npm init` to initialize a new Node.js project.
- Install Express and other required dependencies: **`npm install express bcrypt jsonwebtoken`**
- Create an Express server file (`server.js`) and add the following code:

```
const express = require('express');
const bcrypt = require('bcrypt');
const jwt = require('jsonwebtoken');

const app = express();
const PORT = 3000;

// Middleware to parse JSON body
app.use(express.json());

// Simulated user data (replace with your database or storage mechanism)
const users = [
  { id: 1, username: 'user1', password: 'password1', role: 'user' },
  { id: 2, username: 'admin', password: 'password2', role: 'admin' }
];

// Authenticate user
app.post('/login', (req, res) => {
  const { username, password } = req.body;

  // Find user by username
  const user = users.find((user) => user.username === username);

  if (!user) {
    return res.status(401).json({ message: 'Invalid username or password' });
  }

  // Check password
  if (!bcrypt.compareSync(password, user.password)) {
    return res.status(401).json({ message: 'Invalid username or password' });
  }

  // Generate JWT token
  const token = jwt.sign({ id: user.id, username: user.username, role: user.role }, 'secretkey');

  // Send token in response
  res.json({ token });
});

// Protected route that requires authentication and authorization
app.get('/protected', authenticateToken, (req, res) => {
  res.json({ message: 'Protected route accessed successfully' });
});

// Middleware to authenticate JWT token
function authenticateToken(req, res, next) {
```

```

const authHeader = req.headers['authorization'];
const token = authHeader && authHeader.split(' ')[1];

if (!token) {
  return res.sendStatus(401);
}

jwt.verify(token, 'secretkey', (err, user) => {
  if (err) {
    return res.sendStatus(403);
  }

  req.user = user;
  next();
});

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

```

Step 2: Run the server: Run **node server.js** in the terminal to start the server.

Step 3: Testing the endpoints:

- Use a tool like cURL, Postman, or any HTTP client to test the endpoints.
- Send a POST request to **http://localhost:3000/login** with a JSON body containing the username and password. You should receive a JWT token in the response.
- Copy the token and send a GET request to **http://localhost:3000/protected** with an **Authorization** header in the format **Bearer <token>**. If the token is valid, you should receive a response with the message "Protected route accessed successfully".

EXPERIMENT -7

Aim: To Access Data Using Nodejs

Description:

Mongoose :

Mongoose is a popular object modeling library for MongoDB. It provides a schema-based solution to model your application data and includes features like validation, query building, and middleware.

Step-1 Create a project folder.

Step-2 install mongoose.

Step-3 Create .js file

Step-4 run node file.js

Demonstration:

Step 1: Connecting mongoose to mongodb and creating blog post

```
const mongoose = require('mongoose');
main().catch(err => console.log(err));
async function main() {
  await mongoose.connect('mongodb://127.0.0.1:27017/mydb');
  const postschema = new mongoose.Schema({
    title: String,
    content: String,
    date: { type: Date, default: Date.now },
  });
  const Post = mongoose.model('Post', postschema);
  const post = new Post({
    title: 'My first blog post',
    content: 'This is the content of my first blog post',
  });
  await post.save();
}
```

```
{
  _id: ObjectId('642a6994a548bdf5317adecb'),
  title: "My first blog post",
  content: "This is the content of my first blog post",
  date: 2023-04-03T05:52:20.122+00:00,
  __v: 0
}
```

Step 2: Find

➤The find() function is used to find particular data from the MongoDB database.

➤Syntax: Model.find()

```
const p = await Post.find();
```

```
console.log(p);
```

```
PS C:\Users\CBIT\Documents\my_mongoose> node moongoose.js
[
  {
    _id: new ObjectId("642a6b142e7f013c05728a8b"),
    title: 'My first blog post',
    content: 'This is the content of my first blog post',
    date: 2023-04-03T05:58:44.272Z,
    __v: 0
  }
]
```

Step 3: Findbyid

➤ In MongoDB, all documents are unique because the `_id` field or path that MongoDB uses to automatically create a new document.

➤ Syntax: `Model.findById(id)`

```
const id = "642a6b142e7f013c05728a8b"
const l = await Post.findById(id)
console.log(l)
```

```
PS C:\Users\CBIT\Documents\my_mongoose> node moongoose.js
{
  _id: new ObjectId("642a6b142e7f013c05728a8b"),
  title: 'My first blog post',
  content: 'This is the content of my first blog post',
  date: 2023-04-03T05:58:44.272Z,
  __v: 0
}
```

Step 4: Findbyidandupdate:

➤ Function is used to find a matching document, updates it according to the update arg, passing any options, and returns the found document (if any) to the callback.

➤ Syntax: `Model.findByIdAndUpdate()`

```
const id = "642a6b142e7f013c05728a8b"
const h= await Post.findByIdAndUpdate(id,
{ title: 'Updated blog post title' },
{ new: true },
)
console.log(h)
```

```
PS C:\Users\CBIT\Documents\my_mongoose> node moongoose.js
{
  _id: new ObjectId("642a6b142e7f013c05728a8b"),
  title: 'Updated blog post title',
  content: 'This is the content of my first blog post',
  date: 2023-04-03T05:58:44.272Z,
  __v: 0
}
```

Step 5: Findbyanddelete:

➤ It is used to find a matching document, removes it, and passing the found document (if any) to the callback.

➤ Syntax: `Model.findByIdAndDelete(id)`

```
const id = "642a6b142e7f013c05728a8b"
const m= await Post.findByIdAndDelete(id)
console.log(m)
```

```
PS C:\Users\CBIT\Documents\my_mongoose> node moongoose.js
null
```

EXPERIMENT -8

Aim: To Create a form to edit the data using Angular2.

Description:

Angular 2 is a popular open-source, front-end web application framework that is designed to help developers build dynamic, responsive, and scalable web applications. It is a complete rewrite of the original AngularJS framework, with a focus on simplicity, performance, and scalability. Angular 2 is based on a component-based architecture, where each component encapsulates the presentation and logic for a specific feature or functionality. Components can be nested within other components, allowing developers to create complex UI elements with ease.

One of the key features of Angular 2 is its use of declarative templates, which allow developers to define the UI structure and behavior using HTML and a set of directives. These directives provide a way to bind data to the UI, listen to events, and create reusable components.

Angular 2 also includes a powerful set of tools and libraries, such as RxJS for reactive programming, TypeScript for type-checking and compile-time error detection, and the Angular CLI for automating project setup and build processes.

In addition, Angular 2 provides support for internationalization and accessibility, making it easier for developers to create applications that can be used by a global audience. It also has built-in support for testing, with a comprehensive suite of testing tools and libraries.

Demonstration:

Step 1:

- Set up a new Angular 2 project using the Angular CLI.
- ```
npm install -g @angular/cli
```
- Create new project by this command, Choose yes for routing option and, CSS ng new myNewApp
  - Go to your project directory cd myNewApp
  - Run server and see your application in action ng serve

**Step 2:**

- Create a new component for your form using the Angular CLI ng generate component edit-form

**Step 3:**

- In your component's HTML file (edit-form.component.html), create a form using the Angular ngForm directive:

```
<form #editForm="ngForm" (ngSubmit)="onSubmit()">
<label for="name">Name:</label>
<input type="text" id="name" name="name" [(ngModel)]="user.name" required>
<label for="email">Email:</label>
<input type="email" id="email" name="email" [(ngModel)]="user.email" required>
<button type="submit" [disabled]="editForm.invalid">Submit</button>
</form>
```

- This form has two input fields for the user's name and email, and a submit button.
- The ngModel directive is used to bind the form fields to a user object in your component's TypeScript file.



#### **Step 4:**

➤ In your component's TypeScript file (edit-form.component.ts), define the user object and write a function to handle form submissions:

```
import { Component } from '@angular/core'; @Component({
selector: 'app-edit-form',
templateUrl: './edit-form.component.html', styleUrls: ['./edit-form.component.css']
})
export class EditFormComponent {
```

```
 user = {
 name: 'John Doe',
 email: 'johndoe@example.com'
 };
 onSubmit() {
 console.log('User data submitted: ', this.user);
 }
}
```

➤ In this example, the user object is pre-populated with some sample data. When the form is submitted, the onSubmit() function logs the user object to the console.

#### **Step 5:**

➤ Add your form component to your application by adding it to your app.module.ts file:

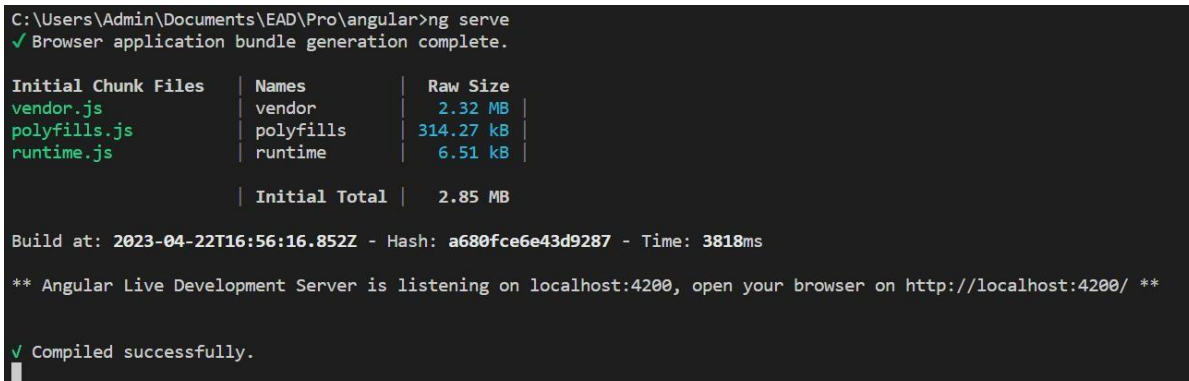
```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { FormsModule } from '@angular/forms';
import { AppComponent } from './app.component';
import { EditFormComponent } from './edit-form/edit-form.component'; @NgModule({
declarations: [
 AppComponent, EditFormComponent
],
imports: [BrowserModule, FormsModule
],
providers: [],
bootstrap: [AppComponent]
})
export class AppModule { }
```

#### **Step 6:**

➤ Run your application using the Angular CLI:

ng serve

➤ This should start a development server and open your application in a browser. You should now be able to see your form and submit data to it.



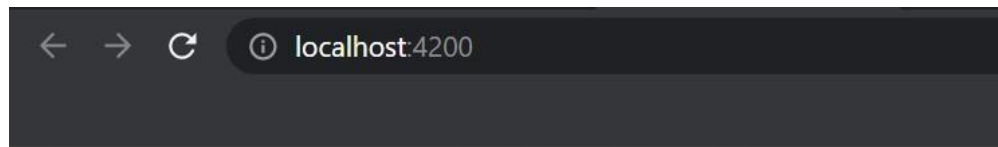
```
C:\Users\Admin\Documents\EAD\Pro\angular>ng serve
✓ Browser application bundle generation complete.

Initial Chunk Files | Names | Raw Size
vendor.js | vendor | 2.32 MB
polyfills.js | polyfills | 314.27 kB
runtime.js | runtime | 6.51 kB
 | Initial Total | 2.85 MB

Build at: 2023-04-22T16:56:16.852Z - Hash: a680fce6e43d9287 - Time: 3818ms

** Angular Live Development Server is listening on localhost:4200, open your browser on http://localhost:4200/ **

✓ Compiled successfully.
```



Name:  Email:

