
Chapter 6: Building a Full-Stack CRUD Application

Introduction

In modern web applications, almost everything revolves around **data**. Whether it's a social media post, a product in an e-commerce store, or a simple to-do item, applications need to **store, retrieve, update, and delete** data. These four operations are collectively called **CRUD**:

- **C – Create**: Add new data.
- **R – Read**: Retrieve or display existing data.
- **U – Update**: Modify data.
- **D – Delete**: Remove data.

This chapter will guide you in building a **full-stack Task Manager application** from scratch using **HTML, CSS, JavaScript, Node.js, Express, and MongoDB**. Along the way, we will explain:

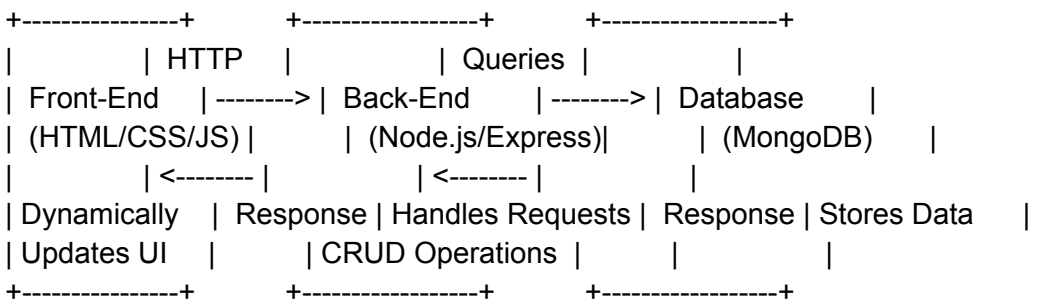
1. How **front-end and back-end communicate**.
2. How **HTTP methods** map to CRUD operations.
3. How to **store, query, update, and delete data** in MongoDB.
4. How **JavaScript updates the UI dynamically**.
5. Best practices for structuring code and projects.

By the end, you will understand **not only how to code** but also **why each step works**, giving you the knowledge to build any CRUD application in the future.

---

1. Full-Stack Architecture Overview

Before coding, let's visualize how the different parts of our application interact:



Roles of Each Layer

- **Front-End (Client)**: Handles **user interaction**. It collects input and displays output. JavaScript allows the UI to **update dynamically** without reloading the page.
- **Back-End (Server)**: Handles **business logic and data operations**. Express simplifies **routing and request handling**.
- **Database**: Stores **persistent data**. MongoDB uses **collections and documents**, making it flexible and scalable.

Analogy

Think of your application like a **restaurant**:

- **Front-End**: The **waiter** who takes your order (input) and serves your food (output).
- **Back-End**: The **kitchen** that prepares the meal according to instructions.
- **Database**: The **pantry/storage** where ingredients (data) are kept.

---

2. Core Concepts Before Coding

2.1 CRUD Operations and HTTP Methods

CRUD Operation	HTTP Method	Description
Create	POST	Add new data to the database
Read	GET	Retrieve existing data from the database

Update	PUT/PATCH	Modify existing data in the database
Delete	DELETE	Remove data from the database

- **POST** is used when creating new records.
- **GET** is used for fetching data without changing it.
- **PUT/PATCH** is for updating existing records.
- **DELETE** removes data permanently.

2.2 MongoDB Fundamentals

- MongoDB is a **NoSQL database**.
- Data is stored as **documents** in **collections**, similar to JSON objects.
- Each document has a unique `_id` for identification.

Example document:

```
{
  "_id": "64f9a0f2b6f1f2d1e1234567",
  "name": "Buy groceries"
}
```

- `_id` ensures each task can be uniquely identified.
- MongoDB allows **flexible schema**, meaning documents in the same collection can have slightly different fields.

2.3 Client-Server Communication

- **Front-End**: Sends **HTTP requests** (GET, POST, PUT, DELETE) using **Fetch API**.
- **Back-End**: Listens on **routes** (e.g., `/tasks`) and executes corresponding operations.

- **Database**: Performs CRUD operations and sends results back.
- **Front-End**: Receives response and **updates the DOM dynamically**.

Data Flow Example: Adding a Task

1. User types task → clicks **Add**.
2. JavaScript sends a **POST request** to `/tasks`.
3. Server receives request → inserts task into MongoDB.
4. Server responds with success.
5. JavaScript fetches updated task list → updates UI.

---

3. Setting Up the Project

3.1 Folder Structure

```
task-manager/
├── public/
│   ├── index.html
│   ├── style.css
│   └── script.js
├── server.js
├── package.json
```

- **public/**: Contains front-end files.
- **server.js**: Back-end entry point.
- **package.json**: Manages project dependencies.

3.2 Initialize Node.js Project

npm init -y

This creates `package.json`.

3.3 Install Dependencies

npm install express mongodb
npm install --save-dev nodemon

- **Express**: Framework for handling HTTP requests.
- **MongoDB driver**: Connects Node.js to MongoDB.
- **Nodemon**: Automatically restarts the server during development.

---

4. Front-End Development

4.1 HTML

```
<!DOCTYPE html>
<html>
<head>
   <title>Task Manager</title>
   <link rel="stylesheet" href="style.css">
</head>
<body>
   <div class="container">
      <h1>Task Manager</h1>
      <form id="task-form">
         <input type="text" id="task-input" placeholder="Enter a new task" required>
         <button type="submit">Add Task</button>
      </form>
      <ul id="task-list"></ul>
   </div>
   <script src="script.js"></script>
</body>
</html>
```

Explanation:

- `<form>`: Allows user to input tasks.
- `<ul>`: Dynamically lists all tasks.

- `script.js`: Handles **API requests** and **DOM updates**.

4.2 CSS

```
body {
    font-family: Arial;
    background: #f4f4f4;
    display: flex;
    justify-content: center;
    align-items: center;
    height: 100vh;
}

.container {
    background: white;
    padding: 20px;
    width: 400px;
    border-radius: 5px;
    box-shadow: 0 0 10px rgba(0,0,0,0.1);
}

h1 { text-align: center; }

form { display: flex; gap: 10px; }
input[type="text"] { flex: 1; padding: 8px; border-radius: 3px; border: 1px solid #ccc; }
button { padding: 8px 12px; border: none; border-radius: 3px; background: #5cb85c; color:
white; cursor: pointer; }
button:hover { background: #4cae4c; }

ul { list-style: none; padding: 0; margin-top: 20px; }
li { display: flex; justify-content: space-between; background: #eee; padding: 8px;
margin-bottom: 5px; border-radius: 3px; }
li button { margin-left: 5px; }
```

Explanation:

- Provides a **clean, readable UI**.
- Makes buttons visually distinct for **Edit** and **Delete** actions.

4.3 JavaScript

```
document.addEventListener('DOMContentLoaded', () => {
    const form = document.getElementById('task-form');
    const input = document.getElementById('task-input');
    const list = document.getElementById('task-list');

    // Load all tasks from server
    function loadTasks() {
        fetch('/tasks')
        .then(res => res.json())
        .then(tasks => {
            list.innerHTML = '';
            tasks.forEach(task => {
                const li = document.createElement('li');
                li.textContent = task.name;

                // Edit button
                const editBtn = document.createElement('button');
                editBtn.textContent = 'Edit';
                editBtn.onclick = () => {
                    const newName = prompt('Edit task', task.name);
                    if(newName) {
                        fetch(`/tasks/${task._id}`, {
                            method: 'PUT',
                            headers: {'Content-Type': 'application/json'},
                            body: JSON.stringify({name: newName})
                        }).then(loadTasks);
                    }
                };

                // Delete button
                const deleteBtn = document.createElement('button');
                deleteBtn.textContent = 'Delete';
                deleteBtn.onclick = () => {
                    fetch(`/tasks/${task._id}`, { method: 'DELETE' }).then(loadTasks);
                };

                li.appendChild(editBtn);
                li.appendChild(deleteBtn);
                list.appendChild(li);
            });
        });
    }

    // Add new task
    form.addEventListener('submit', e => {
        e.preventDefault();
        const name = input.value.trim();
        if(name) {
```

```
        fetch('/tasks', {
            method: 'POST',
            headers: {'Content-Type': 'application/json'},
            body: JSON.stringify({name})
        }).then(() => { input.value=''; loadTasks(); });
    }
  });

  loadTasks();
});
```

Explanation:

- `loadTasks()`: Fetches tasks and updates the DOM.
- `Edit` button: Sends a PUT request to update task.
- `Delete` button: Sends a DELETE request.
- `form` submission: Sends a POST request.

---

5. Back-End Development

```
const express = require('express');
const { MongoClient, ObjectId } = require('mongodb');

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

app.use(express.json());
app.use(express.static('public'));

const url = 'mongodb://localhost:27017';
const dbName = 'taskdb';

MongoClient.connect(url, { useUnifiedTopology:true }, (err, client) => {
  if(err) return console.error(err);
  const db = client.db(dbName);
  const tasks = db.collection('tasks');

  // GET all tasks
  app.get('/tasks', (req, res) => {
    tasks.find({}).toArray((err, result) => {
```

```
            if(err) res.status(500).send(err); else res.json(result);
        });
    });

    // POST new task
    app.post('/tasks', (req, res) => {
        const task = {name: req.body.name};
        tasks.insertOne(task, (err) => {
            if(err) res.status(500).send(err); else res.status(201).send('Task added');
        });
    });

    // PUT update task
    app.put('/tasks/:id', (req, res) => {
        tasks.updateOne({_id: ObjectId(req.params.id)}, {$set: {name: req.body.name}}, (err)
=> {
            if(err) res.status(500).send(err); else res.send('Task updated');
        });
    });

    // DELETE task
    app.delete('/tasks/:id', (req, res) => {
        tasks.deleteOne({_id: ObjectId(req.params.id)}, (err) => {
            if(err) res.status(500).send(err); else res.send('Task deleted');
        });
    });

    app.listen(port, () => console.log(`Server running at http://localhost:${port}`));
});
```

Theory & Explanation:

- Express handles routing.
- Each route corresponds to a CRUD operation.
- `ObjectId` ensures each task can be uniquely updated or deleted.
- `app.use(express.json())` parses JSON from incoming requests.

---

6. Running and Testing the Application

1. Start MongoDB.
2. Run the server:

```
npm start
```

3. Open browser at `http://localhost:3000`.
4. Test adding, editing, deleting tasks.
5. Observe how data persists in MongoDB.

---

7. Key Concepts Recap

- **CRUD Operations** – Core of any data-driven application.
- **HTTP Methods** – POST, GET, PUT, DELETE map to CRUD.
- **Front-End** – HTML, CSS, JS for interaction and dynamic updates.
- **Back-End** – Node.js + Express for request handling.
- **Database** – MongoDB stores tasks as documents.
- **Full-Stack Flow** – Front-End ↔ Server ↔ Database.

---

8. Advanced Tips for Students

- Add input validation to prevent empty tasks.
- Add **task completion** feature.
- Implement **timestamps** and **sorting**.
- Add **user authentication**.
- Enhance UI with **CSS frameworks** or **animations**.