Perfect 🚀 Let's structure *Practice 6: Interactive Form with Validation using JavaScript and CSS3* into a detailed lab note.

📝 Practice 6: Create an Interactive Form with Validation

✅ Aim

To *create a responsive and interactive HTML form* using *JavaScript for validation* and *CSS3 for styling*, ensuring user inputs are correct before submission.

✅ Requirements

* Basic knowledge of HTML, CSS, and JavaScript.

* Code editor (VS Code recommended).

* Browser for testing.

✅ Steps

1️⃣ HTML Form Structure

html

<!DOCTYPE html>

<html lang="en">

```
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>Interactive Form</title>
<link rel="stylesheet" href="style.css">
</head>
<body>
  <div class="form-container">
    <h2>Sign Up Form</h2>
    <form id="signupForm">
      <label for="name">Name:</label>
      <input type="text" id="name" placeholder="Enter your name" required>

      <label for="email">Email:</label>
      <input type="email" id="email" placeholder="Enter your email" required>

      <label for="password">Password:</label>
      <input type="password" id="password" placeholder="Enter password" required>

      <label for="confirmPassword">Confirm Password:</label>
      <input type="password" id="confirmPassword" placeholder="Confirm password" required>

      <button type="submit">Submit</button>
      <p id="errorMsg"></p>
    </form>
  </div>
<script src="script.js"></script>
</body>
</html>
```

2️⃣ CSS3 Styling (style.css)

```
css
body {
  font-family: Arial, sans-serif;
  background-color: #f4f4f9;
  display: flex;
  justify-content: center;
  align-items: center;
  height: 100vh;
}

.form-container {
  background: white;
  padding: 30px;
  border-radius: 10px;
  box-shadow: 0 0 15px rgba(0,0,0,0.2);
  width: 350px;
}

h2 {
  text-align: center;
  margin-bottom: 20px;
}

input {
  width: 100%;
  padding: 10px;
  margin-bottom: 15px;
```

```
  border-radius: 5px;
  border: 1px solid #ccc;
  transition: 0.3s;
}

input:focus {
  border-color: #333;
  box-shadow: 0 0 5px #aaa;
}

button {
  width: 100%;
  padding: 10px;
  background-color: #333;
  color: white;
  border: none;
  border-radius: 5px;
  cursor: pointer;
  transition: 0.3s;
}

button:hover {
  background-color: #555;
}

#errorMsg {
  color: red;
  font-size: 14px;
  text-align: center;
}
```

3️⃣ JavaScript Validation (script.js)

javascript

```
const form = document.getElementById('signupForm');
const errorMsg = document.getElementById('errorMsg');

form.addEventListener('submit', function(e) {
  e.preventDefault(); // Prevent form submission

  const name = document.getElementById('name').value.trim();
  const email = document.getElementById('email').value.trim();
  const password = document.getElementById('password').value.trim();
  const confirmPassword = document.getElementById('confirmPassword').value.trim();

  // Validation
  if(name === '' || email === '' || password === '' || confirmPassword === '') {
    errorMsg.textContent = 'All fields are required!';
    return;
  }

  const emailPattern = /^[^ ]+@[^ ]+\.[a-z]{2,3}$/;
  if(!email.match(emailPattern)) {
    errorMsg.textContent = 'Invalid email address!';
    return;
  }

  if(password.length < 6) {
    errorMsg.textContent = 'Password must be at least 6 characters!';
```

```
    return;
  }

  if(password !== confirmPassword) {
    errorMsg.textContent = 'Passwords do not match!';
    return;
  }

  // Success
  errorMsg.style.color = 'green';
  errorMsg.textContent = 'Form submitted successfully!';
  form.reset();
});
```

4️⃣ Features Implemented

* *Real-time validation* on form submission.
* *Email format check* using regex.
* *Password length and match check*.
* *CSS3 styling* for interactive focus and hover effects.
* *Error/success messages* displayed dynamically.

✅ Learning Outcome

* Build *interactive and responsive forms* using HTML5, CSS3, and JS.
* Implement *client-side form validation*.

* Improve *user experience* with CSS transitions and visual feedback.

I can also create a *diagram showing form validation flow* (Input → JS Validation → Error/Success → Submission) for your notes.

Do you want me to do that?

[08-09-2025 20:40] Denish SRM: Perfect 🚀 Let’s do *Practice 7: Develop a RESTful API using Node.js and Express.js*.

We’ll create a simple *REST API for managing users* with *CRUD (Create, Read, Update, Delete)* operations.

📝 Practice 7: RESTful API with Node.js + Express.js

1️⃣ Project Setup

Open terminal and run:

bash

mkdir rest-api-practice

cd rest-api-practice

npm init -y

npm install express

2️⃣ Create server.js

javascript

```
// Import dependencies
const express = require('express');
const app = express();

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

// Sample in-memory data
let users = [
  { id: 1, name: "Alice", email: "alice@example.com" },
  { id: 2, name: "Bob", email: "bob@example.com" }
];

// 🟢 GET all users
app.get('/api/users', (req, res) => {
  res.json(users);
});

// 🟢 GET single user by ID
app.get('/api/users/:id', (req, res) => {
  const user = users.find(u => u.id == req.params.id);
  if (!user) return res.status(404).json({ error: "User not found" });
  res.json(user);
});

// 🟢 POST create a new user
app.post('/api/users', (req, res) => {
  const newUser = {
```

```
    id: users.length + 1,
    name: req.body.name,
    email: req.body.email
  };
  users.push(newUser);
  res.status(201).json(newUser);
});

// 🟢 PUT update an existing user
app.put('/api/users/:id', (req, res) => {
  const user = users.find(u => u.id == req.params.id);
  if (!user) return res.status(404).json({ error: "User not found" });

  user.name = req.body.name || user.name;
  user.email = req.body.email || user.email;

  res.json(user);
});

// 🟢 DELETE a user
app.delete('/api/users/:id', (req, res) => {
  users = users.filter(u => u.id != req.params.id);
  res.json({ message: "User deleted successfully" });
});

// Start server
const PORT = 3000;
app.listen(PORT, () => console.log(`Server running at http://localhost:${PORT}`));
```

3️⃣ Run the Server

bash

node server.js

4️⃣ Test the API (using Postman or browser)

1. *GET all users* →

   GET http://localhost:3000/api/users

2. *GET single user* →

   GET http://localhost:3000/api/users/1

3. *POST new user* →

   POST http://localhost:3000/api/users

   json

   {

    "name": "Charlie",

    "email": "charlie@example.com"

   }

4. *PUT update user* →

   PUT http://localhost:3000/api/users/2

   json

{

"name": "Bobby"

}

5. *DELETE user* →

DELETE http://localhost:3000/api/users/1

✅ *You now have a working REST API* built with *Node.js and Express.js*.

👉 Do you want me to extend this Practice 7 by *adding authentication (JWT login system)* so it becomes more like a real-world API, or keep it simple CRUD for now?

[08-09-2025 20:40] Denish SRM: Great 🚀 Let's walk through *Practice 8: Integrating JWT Authentication in a Node.js Backend Application* step by step.

🔐 Practice 8: JWT Authentication with Node.js + Express

✅ 1. Setup Project

Initialize Node.js project:

bash

mkdir jwt-auth-app

```
cd jwt-auth-app
npm init -y
npm install express jsonwebtoken bcryptjs body-parser
```

✅ 2. Project Structure

```
jwt-auth-app/
|— server.js
|— users.js   (mock user data / DB simulation)
```

✅ 3. Mock User Data (users.js)

For simplicity, we'll simulate a database.

```
javascript
const bcrypt = require("bcryptjs");

// Hash a password (in real apps, this comes from DB)
const hashedPassword = bcrypt.hashSync("password123", 10);

const users = [
 {
   id: 1,
   username: "shree",
```

```
    password: hashedPassword, // store hashed password
  },
];

module.exports = users;
```

✅ 4. Main Server (server.js)

```
javascript
const express = require("express");
const jwt = require("jsonwebtoken");
const bcrypt = require("bcryptjs");
const bodyParser = require("body-parser");
const users = require("./users");

const app = express();
app.use(bodyParser.json());

const SECRET_KEY = "mySecretKey"; // Normally store in env variable

//  🔹 1. Login Route (Generate JWT)
app.post("/login", (req, res) => {
  const { username, password } = req.body;
  const user = users.find((u) => u.username === username);

  if (!user) return res.status(404).json({ message: "User not found" });

  const isPasswordValid = bcrypt.compareSync(password, user.password);
```

```
  if (!isPasswordValid) return res.status(401).json({ message: "Invalid password" });

  // Generate JWT
  const token = jwt.sign({ id: user.id, username: user.username }, SECRET_KEY, {
    expiresIn: "1h", // expires in 1 hour
  });

  res.json({ message: "Login successful", token });
});

// 🔹 2. Middleware: Verify Token
function authenticateToken(req, res, next) {
  const authHeader = req.headers["authorization"];
  const token = authHeader && authHeader.split(" ")[1]; // Bearer <token>

  if (!token) return res.status(403).json({ message: "Token required" });

  jwt.verify(token, SECRET_KEY, (err, user) => {
    if (err) return res.status(403).json({ message: "Invalid token" });
    req.user = user;
    next();
  });
}

// 🔹 3. Protected Route
app.get("/dashboard", authenticateToken, (req, res) => {
  res.json({ message: `Welcome ${req.user.username}, this is your dashboard!` });
});

// Server Start
app.listen(5000, () => console.log("Server running on http://localhost:5000"));
```

✅ 5. Testing the Flow

1. *Start server*

   bash

   node server.js

2. *Login to get JWT*

   POST → http://localhost:5000/login

   json

   {

    "username": "shree",

    "password": "password123"

   }

   ✅ Response → { "message": "Login successful", "token": "<JWT_TOKEN>" }

3. *Access Protected Route*

   GET → http://localhost:5000/dashboard

   Add Header:

   Authorization: Bearer <JWT_TOKEN>

✅ Response → "Welcome shree, this is your dashboard!"

✅ 6. Learning Outcome

* *JWT creation* using jsonwebtoken.sign().

* *JWT verification* using jsonwebtoken.verify().

* *Middleware* to protect routes.

* Simulated login with hashed passwords.

👉 Do you want me to *extend this practice* into a *full authentication system* (Register + Login + Protected Routes + Refresh Tokens) or just keep it at the *basic login + protected route* level?

[08-09-2025 20:40] Denish SRM: Awesome! Here's a complete hands-on lab for *Practice 9: Real-time Chat with WebSockets in Node.js. It uses **Node.js + Express* to serve a static client and *ws* for raw WebSocket messaging (rooms, typing, join/leave notifications, and heartbeat included).

🍼 Practice 9 — Real-Time Chat (Node.js + WebSockets)

1) Setup

bash

mkdir ws-chat

cd ws-chat

npm init -y

npm install express ws nanoid

2) Project structure

```
ws-chat/
 ├─ server.js
└─ public/
   ├─ index.html
   ├─ client.js
  └─ style.css
```

---

3) Server (WebSocket + Express) — server.js

javascript

```
const express = require("express");
const path = require("path");
const { WebSocketServer } = require("ws");
const { nanoid } = require("nanoid");

const app = express();
app.use(express.static(path.join(__dirname, "public")));

const server = app.listen(3000, () => {
  console.log("Server running at http://localhost:3000");
});

const wss = new WebSocketServer({ server });
```

```
/** Connected clients map: id -> { ws, name } */
const clients = new Map();

/** Broadcast helper */
function broadcast(data, exceptId = null) {
  const msg = JSON.stringify(data);
  for (const [id, client] of clients) {
    if (id !== exceptId && client.ws.readyState === 1) {
      client.ws.send(msg);
    }
  }
}

/** Heartbeat (keep connections healthy) */
function heartbeat() { this.isAlive = true; }

wss.on("connection", (ws) => {
  const id = nanoid(6);
  ws.isAlive = true;
  ws.on("pong", heartbeat);

  // default name until client sets it
  clients.set(id, { ws, name: `User-${id}` });

  // announce join
  broadcast({ type: "presence", event: "join", id, name: clients.get(id).name });
  // send roster to the new client
  ws.send(JSON.stringify({
    type: "welcome",
    id,
    users: [...clients.entries()].map(([cid, c]) => ({ id: cid, name: c.name }))
```

```
}));

ws.on("message", (raw) => {
  let data;
  try { data = JSON.parse(raw.toString()); } catch { return; }

  if (data.type === "setName") {
    const prev = clients.get(id).name;
    clients.get(id).name = data.name?.trim() || prev;
    broadcast({ type: "presence", event: "rename", id, prev, name: clients.get(id).name });
    return;
  }

  if (data.type === "typing") {
    // ephemeral signal
    broadcast({ type: "typing", id }, id);
    return;
  }

  if (data.type === "chat") {
    const msg = {
      type: "chat",
      id,
      name: clients.get(id).name,
      text: (data.text || "").slice(0, 500),
      ts: Date.now()
    };
    broadcast(msg);
    return;
  }
});
```

```
  ws.on("close", () => {
    const user = clients.get(id);
    clients.delete(id);
    broadcast({ type: "presence", event: "leave", id, name: user?.name });
  });
});

/** Ping every 30s to drop dead connections */
const interval = setInterval(() => {
  for (const [id, client] of clients) {
    const ws = client.ws;
    if (ws.isAlive === false) {
      ws.terminate();
      clients.delete(id);
      broadcast({ type: "presence", event: "leave", id, name: client.name });
      continue;
    }
    ws.isAlive = false;
    ws.ping();
  }
}, 30000);

wss.on("close", () => clearInterval(interval));
```

---

4) Client UI — public/index.html

```
html
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0"/>
  <title>WS Chat</title>
  <link rel="stylesheet" href="style.css"/>
</head>
<body>
  <div class="app">
    <header>
      <h1>WebSocket Chat</h1>
      <div class="me">
        <input id="name" placeholder="Your name" maxlength="20" />
        <button id="saveName">Save</button>
      </div>
    </header>

    <main>
      <aside>
        <h3>Online</h3>
        <ul id="users"></ul>
      </aside>
      <section class="chat">
        <ul id="messages"></ul>
        <div id="typing"></div>
        <form id="form">
          <input id="input" autocomplete="off" placeholder="Type a message..." maxlength="500"/>
          <button>Send</button>
        </form>
      </section>
```

```
  </main>
 </div>

 <script src="client.js"></script>
</body>
</html>
```

---

5) Client Logic — public/client.js

javascript

```
const proto = location.protocol === "https:" ? "wss" : "ws";
const socket = new WebSocket(`${proto}://${location.host}`);

const usersEl = document.getElementById("users");
const messagesEl = document.getElementById("messages");
const typingEl = document.getElementById("typing");
const form = document.getElementById("form");
const input = document.getElementById("input");
const nameInput = document.getElementById("name");
const saveNameBtn = document.getElementById("saveName");

let myId = null;
const users = new Map(); // id -> name
let typingTimeout = null;

function renderUsers() {
  usersEl.innerHTML = "";
  [...users.entries()].forEach(([id, name]) => {
```

```
    const li = document.createElement("li");
    li.textContent = id === myId ? `${name} (you)` : name;
    usersEl.appendChild(li);
  });
}

function addMsg(text, meta = {}) {
  const li = document.createElement("li");
  if (meta.system) li.classList.add("system");
  if (meta.me) li.classList.add("me");

  li.innerHTML = meta.system ? text : `
    <span class="author">${meta.name || "Anon"}</span>
    <span class="text">${text}</span>
    <span class="time">${new Date(meta.ts || Date.now()).toLocaleTimeString()}</span>
  `;
  messagesEl.appendChild(li);
  messagesEl.scrollTop = messagesEl.scrollHeight;
}

socket.onopen = () => {
  // optional: set a cached name
  const cached = localStorage.getItem("ws_name");
  if (cached) {
    nameInput.value = cached;
    socket.send(JSON.stringify({ type: "setName", name: cached }));
  }
};

socket.onmessage = (event) => {
  const data = JSON.parse(event.data);
```

```
if (data.type === "welcome") {
  myId = data.id;
  data.users.forEach(u => users.set(u.id, u.name));
  renderUsers();
  addMsg(`Connected as ${users.get(myId)}.`, { system: true });
}

if (data.type === "presence") {
  if (data.event === "join") {
    users.set(data.id, data.name);
    renderUsers();
    addMsg(`${data.name} joined.`, { system: true });
  }
  if (data.event === "leave") {
    users.delete(data.id);
    renderUsers();
    addMsg(`${data.name || "User"} left.`, { system: true });
  }
  if (data.event === "rename") {
    if (users.has(data.id)) {
      users.set(data.id, data.name);
      renderUsers();
      addMsg(`${data.prev} ➜ ${data.name}`, { system: true });
    }
  }
}

if (data.type === "typing" && data.id !== myId) {
  typingEl.textContent = `${users.get(data.id) || "Someone"} is typing...`;
  clearTimeout(typingTimeout);
```

```
    typingTimeout = setTimeout(() => typingEl.textContent = "", 1000);
  }

  if (data.type === "chat") {
    addMsg(data.text, { name: data.name, ts: data.ts, me: data.id === myId });
  }
};

socket.onclose = () => addMsg("Disconnected. Trying to reconnect…", { system: true });

/** Send message */
form.addEventListener("submit", (e) => {
  e.preventDefault();
  const text = input.value.trim();
  if (!text) return;
  socket.send(JSON.stringify({ type: "chat", text }));
  input.value = "";
});

/** Typing indicator */
let lastTyped = 0;
input.addEventListener("input", () => {
  const now = Date.now();
  if (now - lastTyped > 400) {
    socket.send(JSON.stringify({ type: "typing" }));
    lastTyped = now;
  }
});

/** Save name */
saveNameBtn.addEventListener("click", () => {
```

```
  const n = nameInput.value.trim().slice(0, 20);
  if (!n) return;
  socket.send(JSON.stringify({ type: "setName", name: n }));
  localStorage.setItem("ws_name", n);
});
```

---

6) Minimal Styling — public/style.css

```
css
* { box-sizing: border-box; }
body { margin: 0; font-family: system-ui, -apple-system, Segoe UI, Roboto, Arial; background: #0f172a; color: #e2e8f0; }
.app { max-width: 960px; margin: 0 auto; padding: 16px; }
header { display: flex; align-items: center; justify-content: space-between; gap: 12px; }
h1 { font-size: 20px; margin: 0; }
.me { display: flex; gap: 8px; }
.me input, #input { padding: 10px; border: 1px solid #334155; background: #0b1222; color: #e2e8f0; border-radius: 8px; }
.me button, button { padding: 10px 14px; border: 0; background: #2563eb; color: white; border-radius: 8px; cursor: pointer; }
main { display: grid; grid-template-columns: 220px 1fr; gap: 16px; margin-top: 16px; }
aside { background: #0b1222; border: 1px solid #1e293b; border-radius: 12px; padding: 12px; }
aside h3 { margin: 0 0 8px; font-size: 14px; color: #93c5fd; }
#users { list-style: none; padding: 0; margin: 0; display: grid; gap: 6px; }
.chat { display: grid; grid-template-rows: 1fr auto auto; gap: 8px; background: #0b1222; border: 1px solid #1e293b; border-radius: 12px; padding: 12px; }
#messages { list-style: none; padding: 0; margin: 0; overflow-y: auto; max-height: 60vh; display: grid; gap: 8px; }
#messages li { background: #0f213f; border: 1px solid #1e293b; padding: 10px 12px; border-radius: 10px; display: grid; grid-template-columns: auto 1fr auto; gap: 10px; align-items: baseline; }
```

```
#messages li.me { background: #163157; }
#messages li.system { background: #0b1222; color: #93c5fd; font-style: italic; }
.author { font-weight: 600; color: #dbeafe; }
.text { white-space: pre-wrap; word-break: break-word; }
.time { font-size: 12px; color: #94a3b8; }
#typing { min-height: 18px; font-size: 12px; color: #93c5fd; padding: 0 4px; }
#form { display: flex; gap: 8px; }
#input { flex: 1; }
@media (max-width: 720px) {
  main { grid-template-columns: 1fr; }
  aside { order: 2; }
  .chat { order: 1; }
}
```

---

7) Run it

```
bash
node server.js
```

Open *http://localhost:3000* in two browser windows/tabs to chat live.

---

8) What you've learned

* WebSocket *handshake* (via ws), *broadcast, **presence, **typing indicators*
* *Heartbeat* to keep connections alive and auto-clean dead sockets

* Serving a static client via Express while sharing the same port

---

9) Next steps (optional enhancements)

* *Rooms/Channels:* add joinRoom/leaveRoom messages and keep per-room sets.

* *Persistence:* store recent messages in Redis for history on join.

* *Auth:* protect the WS with a JWT (validate token on connection).

* *Scaling:* run multiple Node instances and fan-out via *Redis Pub/Sub* or *Kafka*.

* *Use Socket.IO* if you want built-in reconnection, rooms, and fallbacks.

If you want any of these extensions, tell me which one and I'll drop the code.

[08-09-2025 20:40] Denish SRM: Awesome 🚀 Let's build *Practice 10: A Progressive Web App (PWA) that works offline*.

We'll make a *simple notes app* where users can add notes, and it will still work offline using *Service Worker caching*.

🧪 Practice 10 — Build a PWA with Offline Support

1) Project Setup

Create a folder:

pwa-notes/

├— index.html

├— app.js

├— style.css

├── manifest.json

└── service-worker.js

---

2) HTML File — index.html

html

```
<!DOCTYPE html>
<html lang="en">
<head>
 <meta charset="UTF-8"/>
 <meta name="viewport" content="width=device-width, initial-scale=1.0"/>
 <title>PWA Notes App</title>
 <link rel="stylesheet" href="style.css"/>
 <link rel="manifest" href="manifest.json"/>
 <meta name="theme-color" content="#317EFB"/>
</head>
<body>
 <div class="app">
  <h1>📝 Notes (PWA)</h1>
  <form id="noteForm">
   <input type="text" id="noteInput" placeholder="Write a note..." required/>
   <button type="submit">Add</button>
  </form>
  <ul id="notesList"></ul>
 </div>

 <script src="app.js"></script>
 <script>
```

```
    // Register the service worker
    if ("serviceWorker" in navigator) {
      navigator.serviceWorker.register("service-worker.js")
        .then(() => console.log("Service Worker registered"))
        .catch(err => console.error("SW registration failed:", err));
    }
  </script>
</body>
</html>
```

---

3) CSS File — style.css

```
css
body {
  font-family: Arial, sans-serif;
  background: #f3f4f6;
  display: flex;
  justify-content: center;
  align-items: center;
  height: 100vh;
  margin: 0;
}
.app {
  background: white;
  padding: 20px;
  border-radius: 10px;
  width: 320px;
  box-shadow: 0 4px 10px rgba(0,0,0,0.1);
```

```
}
h1 {
  margin-top: 0;
}
form {
  display: flex;
  gap: 8px;
}
input {
  flex: 1;
  padding: 10px;
  border: 1px solid #ddd;
  border-radius: 6px;
}
button {
  padding: 10px 15px;
  border: none;
  background: #317EFB;
  color: white;
  border-radius: 6px;
  cursor: pointer;
}
ul {
  list-style: none;
  padding: 0;
  margin-top: 15px;
}
li {
  background: #f9fafb;
  padding: 8px 10px;
  margin-bottom: 6px;
```

```
  border-radius: 6px;
}
```

---

4) JavaScript File — app.js

```
javascript
const form = document.getElementById("noteForm");
const input = document.getElementById("noteInput");
const notesList = document.getElementById("notesList");

// Load saved notes from localStorage
let notes = JSON.parse(localStorage.getItem("notes")) || [];
renderNotes();

form.addEventListener("submit", (e) => {
  e.preventDefault();
  const note = input.value.trim();
  if (!note) return;

  notes.push(note);
  localStorage.setItem("notes", JSON.stringify(notes));
  renderNotes();
  input.value = "";
});

function renderNotes() {
  notesList.innerHTML = "";
  notes.forEach((note, index) => {
```

```
    const li = document.createElement("li");
    li.textContent = note;
    notesList.appendChild(li);
  });
}
```

---

5) Manifest File — manifest.json

```
json
{
  "name": "PWA Notes App",
  "short_name": "Notes",
  "start_url": "index.html",
  "display": "standalone",
  "background_color": "#ffffff",
  "theme_color": "#317EFB",
  "icons": [
    {
      "src": "icons/icon-192.png",
      "sizes": "192x192",
      "type": "image/png"
    },
    {
      "src": "icons/icon-512.png",
      "sizes": "512x512",
      "type": "image/png"
    }
  ]
```

}

(You'll need to create icons: icons/icon-192.png and icons/icon-512.png.)

---

6) Service Worker — service-worker.js

javascript

```
const CACHE_NAME = "pwa-notes-cache-v1";
const urlsToCache = [
  "/",
  "/index.html",
  "/style.css",
  "/app.js",
  "/manifest.json"
];

// Install service worker
self.addEventListener("install", (event) => {
  event.waitUntil(
    caches.open(CACHE_NAME).then((cache) => cache.addAll(urlsToCache))
  );
});

// Cache and fetch requests
self.addEventListener("fetch", (event) => {
  event.respondWith(
    caches.match(event.request).then((response) => {
      return response || fetch(event.request);
```

```
    })
  );
});

// Activate and clean old caches
self.addEventListener("activate", (event) => {
  event.waitUntil(
    caches.keys().then((cacheNames) =>
      Promise.all(
        cacheNames.map((cache) => {
          if (cache !== CACHE_NAME) return caches.delete(cache);
        })
      )
    )
  );
});
```

---

7) Run the PWA

1. Serve the app using a simple HTTP server (must be HTTPS in production).

```
bash
npx serve .
```

2. Open http://localhost:3000 (or the port shown).

3. Add a few notes, then *go offline* — the app still works ✅.

4. On Chrome/Edge → “Install App” prompt lets you *install it like a native app*.

---

8) What You Learned

* *Manifest* → makes app installable.

* *Service Worker* → caches files for offline use.

* *LocalStorage* → persists data offline.

* *HTTPS requirement* for PWA features.

👉 Next step: Would you like me to *extend this PWA* with *Push Notifications + Background Sync* so it feels like a real mobile app?

[08-09-2025 20:40] Denish SRM: Perfect 🚀 Let’s set up *Practice 11: Deploy a Node.js Application using AWS Lambda or Firebase Functions*.

We’ll make a *simple "Hello API"* and deploy it serverlessly.

🧪 Practice 11 — Deploy a Node.js App with AWS Lambda or Firebase Functions

---

Option 1: AWS Lambda (with API Gateway)

1) Prerequisites

* AWS account

* Install *AWS CLI*

* Install *Serverless Framework* (for easy deployment)

bash

npm install -g serverless

2) Create a New Project

bash

serverless create --template aws-nodejs --path lambda-app

cd lambda-app

npm init -y

3) Example Function — handler.js

javascript

'use strict';

module.exports.hello = async (event) => {

return {

statusCode: 200,

body: JSON.stringify(

{ message: 'Hello from AWS Lambda + API Gateway!' }

),

};

};

4) serverless.yml Configuration

yaml

service: lambda-app

```
provider:
  name: aws
  runtime: nodejs18.x
  region: us-east-1

functions:
  hello:
    handler: handler.hello
    events:
      - http:
          path: hello
          method: get
```

5) Deploy

bash

```
serverless deploy
```

✅ AWS will give you an API Gateway URL → https://xyz.execute-api.us-east-1.amazonaws.com/dev/hello

---

Option 2: Firebase Functions

1) Prerequisites

* Google account

* Install Firebase CLI:

  bash
  npm install -g firebase-tools

2) Initialize Project

bash
firebase login
firebase init functions

* Choose *JavaScript* as language
* Enable *ESLint* if you want

3) Example Function — functions/index.js

javascript
const functions = require("firebase-functions");

exports.helloWorld = functions.https.onRequest((req, res) => {
  res.send("Hello from Firebase Functions!");
});

4) Deploy

bash
firebase deploy --only functions

✅ Firebase will give you a public URL →

https://us-central1-YOUR_PROJECT.cloudfunctions.net/helloWorld

---

🔑 Comparison

Feature	AWS Lambda	Firebase Functions
Best For	Enterprise, scalable APIs	Startups, mobile/web apps
Triggers	API Gateway, S3, DynamoDB, etc.	Firestore, Auth, Storage, HTTPS
Languages	Node.js, Python, Java, Go, C# etc.	Node.js (default)
Pricing	Pay-per-request, scalable	Free tier + pay-per-use

✅ Learning Outcomes

* Deploy Node.js serverless app in *AWS Lambda* or *Firebase Functions*.

* Understand *serverless triggers and scaling*.

* Use *cloud CLI tools* for deployment automation.

👉 Do you want me to extend this practice by making a *REST API with CRUD operations (Create, Read, Update, Delete) using Lambda or Firebase Functions* instead of just a Hello World?

[08-09-2025 20:40] Denish SRM: Perfect 👌 — let's do *Practice 12: Optimize a Web Application for Performance*.

Here's a *hands-on exercise* to learn *minification, lazy loading, and performance best practices*.

📝 Practice 12: Optimize a Web Application for Performance

🎯 Objective

Improve the performance of a sample web application by applying optimization techniques such as *minification, image optimization, and lazy loading*.

🔹 Step 1: Base HTML (Unoptimized Example)

html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Performance Demo</title>
  <link rel="stylesheet" href="style.css">
  <script src="app.js"></script>
</head>
<body>
  <h1>My Web Application</h1>

  <!-- Large Images -->
  <img src="large1.jpg" alt="Image 1">
  <img src="large2.jpg" alt="Image 2">
  <img src="large3.jpg" alt="Image 3">

  <p id="data"></p>
```

</body>

</html>

🔹 Step 2: Apply *Minification*

* Use tools like *Terser* (for JS) and *CSSNano* (for CSS).

* Example minified CSS:

css

/* Before */

body {

  background-color: white;

  font-size: 16px;

}

/* After Minification */

body{background:#fff;font-size:16px}

🔹 Step 3: Apply *Lazy Loading for Images*

html

<img src="thumbnail1.jpg" data-src="large1.jpg" loading="lazy" alt="Image 1">

<img src="thumbnail2.jpg" data-src="large2.jpg" loading="lazy" alt="Image 2">

<img src="thumbnail3.jpg" data-src="large3.jpg" loading="lazy" alt="Image 3">

👉 loading="lazy" ensures images load *only when visible*.

🔹 Step 4: Optimize JavaScript Loading

html

<!-- Instead of blocking -->

<script src="app.js"></script>

<!-- Optimized -->

<script src="app.js" defer></script>

* defer ensures JS loads after HTML parsing.

🔹 Step 5: Optimize Fonts & Caching

html

<!-- Preload fonts -->

<link rel="preload" href="fonts/Roboto.woff2" as="font" type="font/woff2" crossorigin>

* Use *Cache-Control headers* in server config:

Cache-Control: public, max-age=31536000, immutable

🔹 Step 6: Use GZIP/Brotli Compression

Enable in server config (e.g., Nginx or Express.js middleware).

```
js
// Express.js compression
const compression = require('compression');
app.use(compression());
```

🔹 Step 7: Test Performance

Use *Google Lighthouse* or *PageSpeed Insights* to check:

* Load time
* Core Web Vitals (LCP, CLS, FID)
* Opportunities like unused CSS/JS

✅ Final Optimized HTML Example

```
html
<!DOCTYPE html>
<html lang="en">
<head>
```

```
  <meta charset="UTF-8">
  <title>Optimized Web App</title>
  <link rel="preload" href="style.min.css" as="style">
  <link rel="stylesheet" href="style.min.css">
  <script src="app.min.js" defer></script>
</head>
<body>
  <h1>My Optimized Web Application 🚀</h1>

  <!-- Lazy Loading Images -->
  <img src="thumb1.jpg" data-src="large1.webp" loading="lazy" alt="Image 1">
  <img src="thumb2.jpg" data-src="large2.webp" loading="lazy" alt="Image 2">

  <p id="data">Loading optimized content...</p>
</body>
</html>
```

🔹 Deliverables for Practice

1. Start with a simple app with *large images & unminified code*.
2. Apply *minification (CSS/JS)*.
3. Add *lazy loading for images & videos*.
4. Optimize *JS loading* (defer / async).
5. Use *preload & caching headers*.
6. Test improvements with *Lighthouse*.

👉 Do you want me to also create a *step-by-step checklist (like a deployment-ready guide)* so you can apply these optimizations whenever you finish a project?