PPA6: Arrays, DELETE, and Persistent JSON Storage

1 Purpose of This Assignment

In PPA5 you will extend your existing Node REST server and client interface to manage a collection of data using arrays. You will also add basic persistence by saving and loading the appointment list from a JSON file on disk. This assignment builds on your work with statements and control flow. You will use conditionals and loops to manage the array and update the user interface.

PPA5 builds directly on PPA4's calendar view. In PPA4 you displayed appointments on the client. In PPA5 you will store appointments in a server-side array, retrieve them through `GET`, and re-render the calendar view after `POST` and `DELETE`.

2 Key Topics for This Week

- Arrays as ordered collections (indexing, length, iteration).
- REST routes for `GET`, `POST`, and `DELETE`.
- Mutating arrays safely (`push`, `splice`) and understanding index shifts.
- Persisting server state by reading and writing a JSON file.
- Client side rendering of an array into the DOM.

3 Required Folder Structure

Your project must follow this exact structure. Create the JSON file and initialize it with an empty array.

```
ppa5/
│
├── server.js
├── package.json
├── appointments.json
├── public/
│   ├── index.html
│   ├── style.css
│   └── script.js
│
└── README.txt
```

4 Server-Side Requirements

Your server must maintain an in memory array named appointments. When the server starts, it must load appointments from appointments.json. After `POST` or `DELETE`, it must write the updated array back to appointments.json.

- `GET /appointments` returns the full appointments array as JSON.
- `POST /appointments` adds a new appointment to the array, then saves to file.
- `DELETE /appointments/:index` removes one appointment by index, then saves to file.
- If the JSON file cannot be read, the server should start with an empty array.
- The server stores appointments as an array. Each appointment must be stored as an object, not a string.

Note that when you delete an item from an array, the array length changes and indexes shift.

5 Starter Server Code

This starter code includes TODO lists where you must make design decisions. Replace each TODO list with your own choices and explanations. Keep the server behavior synchronous for file operations (`readFileSync` and `writeFileSync`).

```
"use strict";

const http = require("http");
const url  = require("url");
const fs   = require("fs");

const DATA_FILE    = "appointments.json";
let   appointments = [];

function loadAppointments() {
    // TODO list: decide what should happen if the file does not exist.
    // TODO list: decide what should happen if the JSON is invalid.
    // TODO list: decide whether to log errors to the console or stay silent.
    try {
        const text = fs.readFileSync(DATA_FILE, "utf8");
        appointments = JSON.parse(text);
        if (!Array.isArray(appointments)) {
            appointments = [];
        }
    } catch (error) {
        appointments = [];
    }
}

function saveAppointments() {
    // TODO list: decide how you want the JSON formatted (pretty vs compact).
    // TODO list: decide what to do if writing fails.
    const text = JSON.stringify(appointments, null, 2);
    fs.writeFileSync(DATA_FILE, text, "utf8");
}

function sendJson(response, statusCode, data) {
    response.writeHead(statusCode, { "Content-Type": "application/json" });
    response.end(JSON.stringify(data));
}

function sendText(response, statusCode, message) {
    response.writeHead(statusCode, { "Content-Type": "text/plain" });
    response.end(message);
}

loadAppointments();
```

```
const server = http.createServer(function(request, response) {
    const parsedUrl = url.parse(request.url, true);

    if (request.method === "GET" && parsedUrl.pathname === "/appointments") {
        // TODO list: decide whether you want to return raw appointments or a
wrapper object.
        sendJson(response, 200, appointments);
    }

    else if (request.method === "POST" && parsedUrl.pathname ===
"/appointments") {
        // NOTE: reading the request body is event driven, but your file
operations are synchronous.
        let body = "";

        request.on("data", function(chunk) {
            body += chunk;
        });

        request.on("end", function() {
            // TODO list: validate the incoming appointment fields before
pushing into the array.
            const newAppointment = JSON.parse(body);
            appointments.push(newAppointment);
            saveAppointments();
            sendText(response, 200, "TODO");
        });
    }

    else if (request.method === "DELETE" &&
parsedUrl.pathname.startsWith("/appointments/")) {
        const parts = parsedUrl.pathname.split("/");
        const index = Number(parts[2]);

        // TODO list: decide what error message to send for an invalid index.
        if (!Number.isNaN(index) && index >= 0 && index < appointments.length)
{
            appointments.splice(index, 1);
            saveAppointments();
            sendText(response, 200, "TODO");
        } else {
            sendText(response, 400, "TODO");
        }
    }

    else {
        sendText(response, 404, "TODO");
    }
```

```
});

server.listen(3000);
console.log("TODO");
```

6 Client-Side Requirements

Your interface must display the full appointment list and allow the user to add and delete appointments. Your client must re render the list whenever data changes.

- The client must maintain the calendar view from PPA4
- The calendar view must be generated by iterating over the appointments array
- The appointment form must use an HTML datetime picker (`<input type="datetime-local.."`) instead of free-text date/time fields.
- After each POST or DELETE, the client must `GET /appointments` again and then re-render the calendar
- Use document.`getElementById` to read inputs and update output
- Render the full array using a loop (for or while)
- Create a delete button for each rendered appointment

7 Starter Client Rendering Code

This snippet focuses on arrays, loops, and DOM updates. Complete TODO sections as part of the assignment.

```
function renderAppointments(appointmentsArray) {
    const calendar = document.getElementById("calendar");
    calendar.innerHTML = "";

    for (let i = 0; i < appointmentsArray.length; i++) {
        const appt = appointmentsArray[i];

        const card = document.createElement("div");
        card.className = "appointmentCard";

        // TODO list: decide which appointment fields to show.
        // TODO list: decide how to format the datetime value.
        card.innerText = "TODO";

        const del = document.createElement("button");
        del.innerText = "TODO";
        del.onclick = function() {
            // TODO list: call DELETE /appointments/i
            // TODO list: then GET /appointments and re render
        };

        card.appendChild(del);
        calendar.appendChild(card);
    }
}
```

8 Tips: Reading and Storing Data in a File

Your server is now loading state at startup and saving state after changes. This is a simplified version of what a database does.

- `fs.readFileSync(path, "utf8")` reads text from a file before the server starts listening.
- `JSON.parse(text)` converts JSON text into an array or object.
- `JSON.stringify(data, null, 2)` converts an array or object into JSON text.
- `fs.writeFileSync(path, text, "utf8")` saves the updated JSON after `POST` or `DELETE`.
- Wrap file reads in try catch so the server can recover by starting with an empty array.

9 Tips: Useful Array Functions for Developers

Arrays are more than indexing and loops. JavaScript arrays include many methods that help developers write clearer code.

`push(item)`	adds to the end of an array.
`pop()`	removes the last element.
`splice(index, count)`	removes or replaces elements at a specific index.
`slice(start, end)`	returns a copy of part of an array without changing the original.
`indexOf(value)`	finds the first index of a value (or returns -1).
`includes(value)`	returns true or false.
`forEach(callback)`	loops through items (no return value).
`map(callback)`	transforms an array into a new array.
`filter(callback)`	keeps only items that match a condition.
`find(callback)`	returns the first item that matches a condition.

You may use these methods if you want, but you must still demonstrate at least one explicit loop (`for` or `while`) in your client-side rendering code.

10 Submission Checklist

- Project runs with node `server.js`
- `GET` returns the full appointments array
- `POST` adds an appointment and saves `to appointments.json`
- `DELETE` removes an appointment and saves to `appointments.json`
- Client renders the appointment array and supports deleting items
- All code uses vanilla JavaScript