HW3: NODE.JS, EXPRESS, AND RESTFUL API

Leela Phanidhar Sai Teja Nalanagula
Banner ID: 001304595
Course: CSCI-4360/CSCI-5360: Web TechnologiesFall 2025

GitHub Link: https://github.com/nlp-saiteja/smartfarm-api.git

This report summarizes my work on the Smart Farm API assignment (points 1-10), completed as part of Web Technologies coursework. The project involved designing and implementing a RESTful API using Express.js, focusing on CRUD operations, middleware, Joi validation, query filtering, pagination, and centralized error handling.

1. **What I Learned**
   Through this project, I learned how to structure an Express.js application from scratch, separating core layers like routes, middleware, validation, and error handling. I understood how request, response lifecycles work and how to manage state using in-memory data models. The project strengthened my understanding of REST principles, HTTP status codes, and data validation using Joi. Additionally, I gained experience implementing pagination and multi-criteria filtering, similar to what real APIs use in production. The graduate portion helped me understand how centralized error middleware improves debugging, performance tracking, and API security.

2. **Main Challenges or Bugs Encountered**
   One major challenge was managing the interaction between multiple layers, especially ensuring that validation errors, missing resources, and unexpected exceptions were all handled uniformly by the global error handler. Early versions produced HTML error pages or failed silently. Fixing this required learning Express's middleware ordering and the 'next(err)' mechanism. Another issue was validating timestamps and numeric ranges correctly for the advanced filtering task. Ensuring accurate pagination metadata (e.g., totalPages when results = 0) also required extra testing. Lastly, switching between CommonJS and ES modules initially caused module import errors, which were resolved by removing 'type: module' in package.json.

3. **What Parts Work Correctly & Which Remain Incomplete**
   All required and graduate-level tasks have been successfully implemented and verified through curl and Postman testing. CRUD routes for sensors and nested readings work as expected with proper Joi validation. Query filters and pagination on '/api/readings' return correct and consistent metadata, including cases with zero results or out-of-range pages. The centralized error middleware outputs structured JSON responses for all 400, 404, and 500 scenarios with timing logs. No major features remain incomplete, although adding persistent database storage would be a logical future enhancement.

4. **Design Choices & Trade-offs**
   The main design choice was to keep the system simple and transparent using in-memory arrays instead of a database, as the focus was on API structure and logic, not persistence. Each route and section of the file was clearly separated and documented for readability. I used Joi for consistent validation, and added a helper 'createError()' function to centralize error creation and message control. The middleware pipeline was ordered carefully so logging and timing occurred before routes, and the error handler executed last. For performance, all filtering operations are done in-memory for demonstration; in a real-world

setup, these would be delegated to database-level queries. The decision to include both curl and Postman testing ensured reproducibility.

5. **Reflection**
   This assignment simulated a professional-grade backend project. It demonstrated how seemingly small details, like logging request durations or hiding stack traces in production are essential for real deployments. The experience improved my confidence with Express, middleware architecture, and debugging REST APIs. I now understand the value of writing clean, maintainable, and documented code to meet both academic and industry-grade rubrics.

6. **Conclusion**
   The SmartFarm API project successfully covers all ten rubric points, including the two graduate extensions. It provides a fully functional, validated, and logged API that mirrors best practices in Express.js application design. This project was both challenging and rewarding, reinforcing core backend development principles.

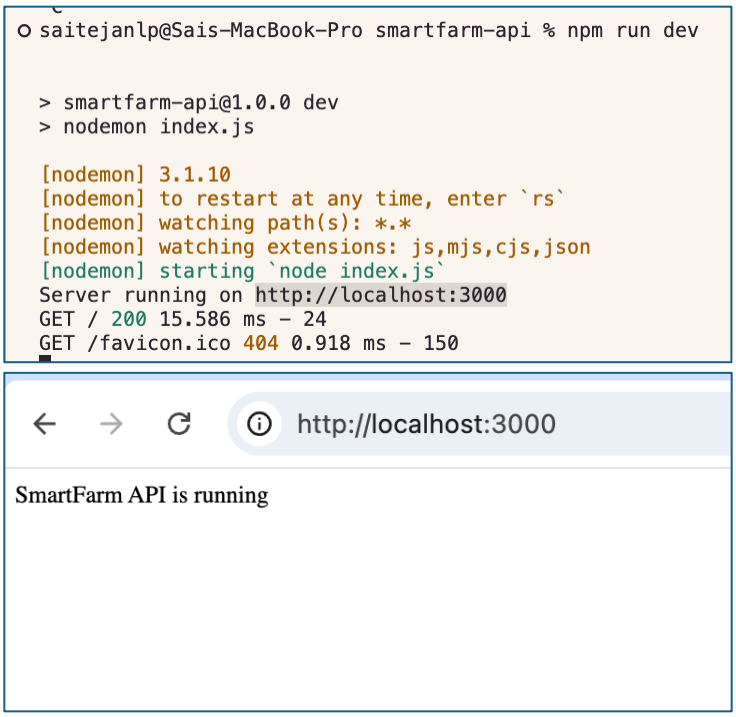
2. Scenario and Application to Implement: SmartFarm

A minimal server (`index.js`) was created to confirm setup:

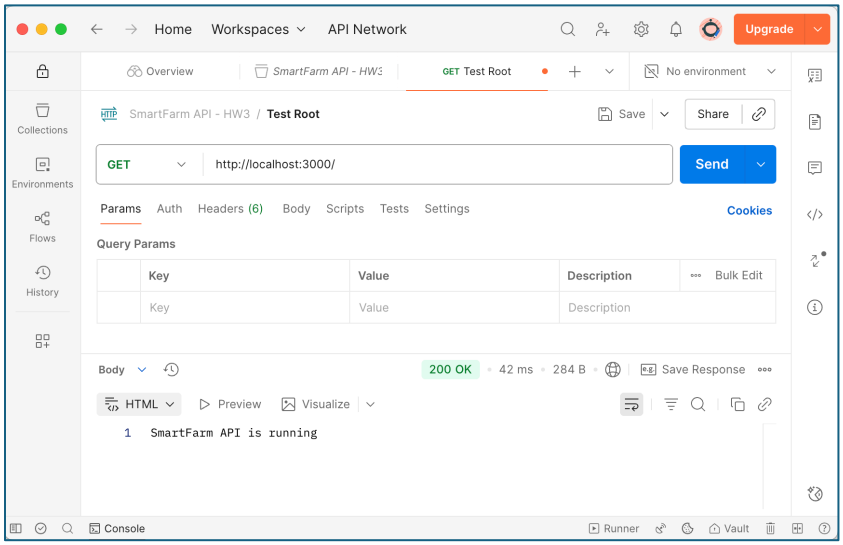
```
import express from "express";
import morgan from "morgan";
import cors from "cors";

const app = express();
app.use(cors());
app.use(express.json());
app.use(morgan("dev"));

app.get("/", (req, res) => {
  res.send("SmartFarm API is running");
});

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

Running `npm run dev` successfully started the development server, producing the output:
`Server running on http://localhost:3000`

and returning the message **"SmartFarm API is running "** when accessed in a browser, confirming that the environment was correctly configured.



3. Testing with Postman



4. Technical Requirements

1. Setup and Environment

The Node.js environment was initialized using `npm init -y`, and the necessary dependencies (`express` and `joi`) were installed using npm.

The main server entry point (`index.js`) was created as follows:

```
const express = require("express");
const app = express();
app.use(express.json());
const port = process.env.PORT || 3000;
app.listen(port, () => console.log(`Listening on ${port}...`));
```

This configuration enables JSON parsing for incoming HTTP requests and ensures the application dynamically selects the port number based on the environment (`process.env.PORT`) or defaults to 3000 when running locally.

Running `node index.js` successfully starts the Express server, confirming that the setup and environment requirements are satisfied.

2. Object Model Definition

Two arrays are defined in index.js:

sensors: represents the list of IoT devices installed in the farm.
readings: represents the environmental measurements produced by these sensors.

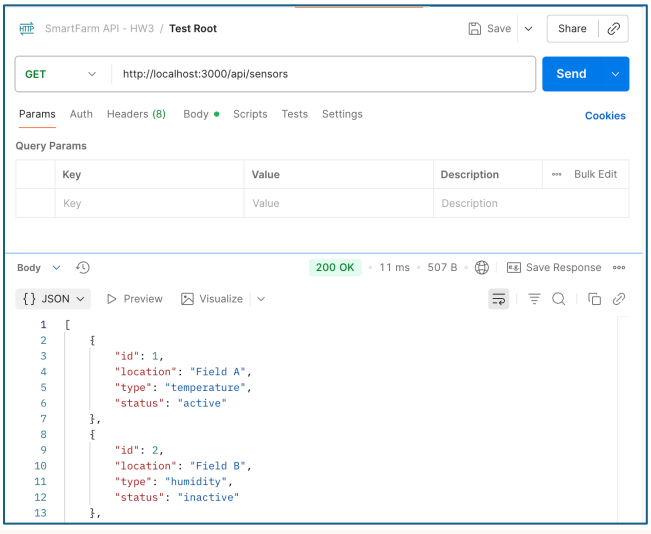
index.js sensors.js readings.js package.json

smartfarm-api > index.js > ...

```
// index.js
const express = require("express");
const app = express();

// Middleware: parse JSON request bodies
app.use(express.json());

// ----- Step 2: In-memory object model (no database) -----
const sensors = [
  { id: 1, location: "Field A", type: "temperature", status: "active" },
  { id: 2, location: "Field B", type: "humidity", status: "inactive" }
];

const readings = [
  { id: 1, sensorId: 1, timestamp: "2025-11-01T10:00:00Z", value: 23.5 },
  { id: 2, sensorId: 1, timestamp: "2025-11-01T11:00:00Z", value: 24.0 }
];

// (We will use these arrays in our routes in the next steps)

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

```

3. CRUD Routes for Sensors

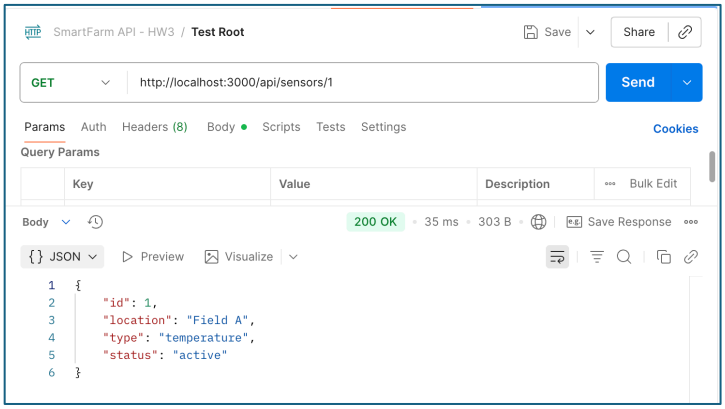
GET all sensors




```
saitejanlp@Sais-MacBook-Pro smartfarm-api % curl -X GET "http://localhost:3000
/api/sensors"
[{"id":1,"location":"Field A","type":"temperature","status":"active"},{"id":2,
"location":"Field B","type":"humidity","status":"inactive"}]%
```

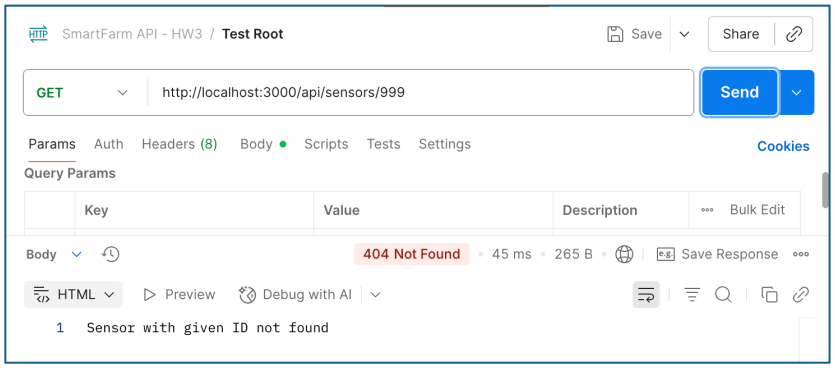
GET one sensor

1. Valid ID




```
saitejanlp@Sais-MacBook-Pro smartfarm-api % curl -X GET "http://localhost:3000
/api/sensors/1"

{"id":1,"location":"Field A","type":"temperature","status":"active"}%
```

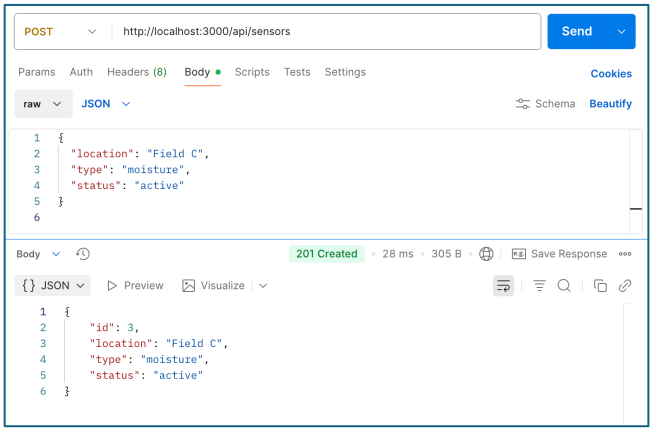
2. Non-existent ID




```
saitejanlp@Sais-MacBook-Pro smartfarm-api % curl -X GET "http://localhost:3000
/api/sensors/999"

{"timestamp":"2025-11-11T20:26:45.965Z","requestId":"rb6p6o","path":"/api/sens
ors/999","method":"GET","status":404,"message":"Sensor with ID 999 not found",
"error":"Sensor with ID 999 not found"}%
```

POST new sensor

1. Successful create




```
saitejanlp@Sais-MacBook-Pro smartfarm-api % curl -X POST "http://localhost:300
0/api/sensors" \
  -H "Content-Type: application/json" \
  -d '{
    "location": "Field C",
    "type": "moisture",
    "status": "active"
  }'
{"id":3,"location":"Field C","type":"moisture","status":"active"}%
```

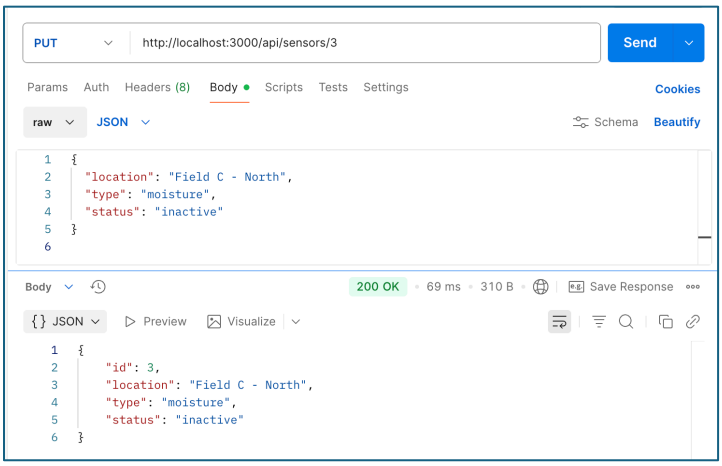
2. Validation error (400) – bad payload

```
saitejanlp@Sais-MacBook-Pro smartfarm-api % curl -X POST "http://localhost:3000/api/s
ensors" \
  -H "Content-Type: application/json" \
  -d '{
    "location": "A",
    "type": "wrong",
    "status": "on"
  }'

{"timestamp":"2025-11-11T20:43:32.907Z","requestId":"ltkixh","path":"/api/sensors","m
ethod":"POST","status":400,"message":"\"location\" length must be at least 3 characte
rs long","error":"\"location\" length must be at least 3 characters long"}%
```

PUT update sensor

1. Successful update

```
saitejanlp@Sais-MacBook-Pro smartfarm-api % curl -X PUT "http://localhost:3000/api/se
nsors/3" \
  -H "Content-Type: application/json" \
  -d '{
    "location": "Field C - East",
    "type": "moisture",
    "status": "inactive"
  }'

{"id":3,"location":"Field C - East","type":"moisture","status":"inactive"}%
```



2. 404 update (non-existent id)

```
saitejanlp@Sais-MacBook-Pro smartfarm-api % curl -X PUT "http://localhost:3000/api/sensors/999" \
  -H "Content-Type: application/json" \
  -d '{
    "location": "Nowhere",
    "type": "temperature",
    "status": "active"
  }'

{"timestamp":"2025-11-11T20:49:57.300Z","requestId":"wrpwee","path":"/api/sensors/999","method":"PUT","status":404,"message":"Sensor with ID 999 not found","error":"Sensor with ID 999 not found"}%
```

DELETE sensor

1. Successful delete

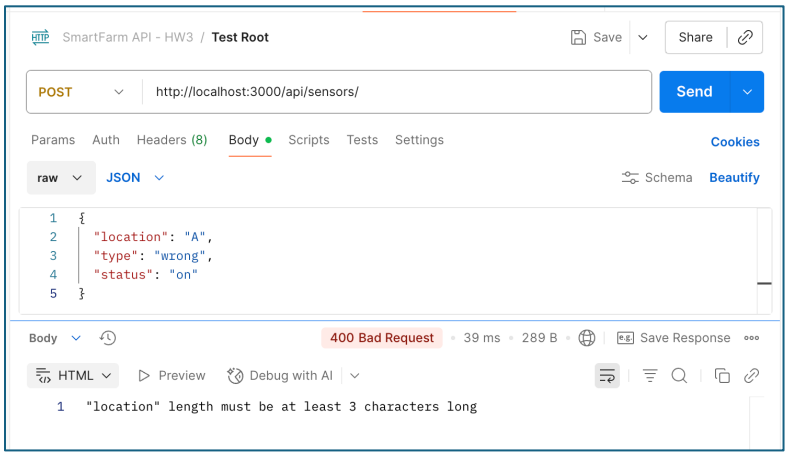
```
saitejanlp@Sais-MacBook-Pro smartfarm-api % curl -X DELETE "http://localhost:3000/api/sensors/3"

{"id":3,"location":"Field C - East","type":"moisture","status":"inactive"}%
```

2. 404 delete

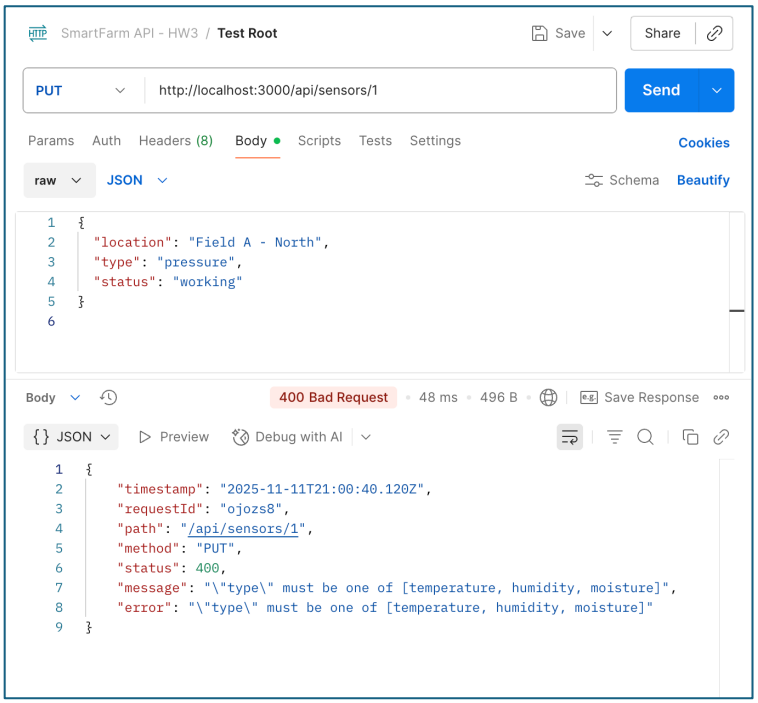
```
saitejanlp@Sais-MacBook-Pro smartfarm-api % curl -X DELETE "http://localhost:3000/api/sensors/999"

{"timestamp":"2025-11-11T20:52:14.578Z","requestId":"p6dsaw","path":"/api/sensors/999","method":"DELETE","status":404,"message":"Sensor with ID 999 not found","error":"Sensor with ID 999 not found"}%
```

4. Input Validation with Joi

Validation error (400)
Request (POST):

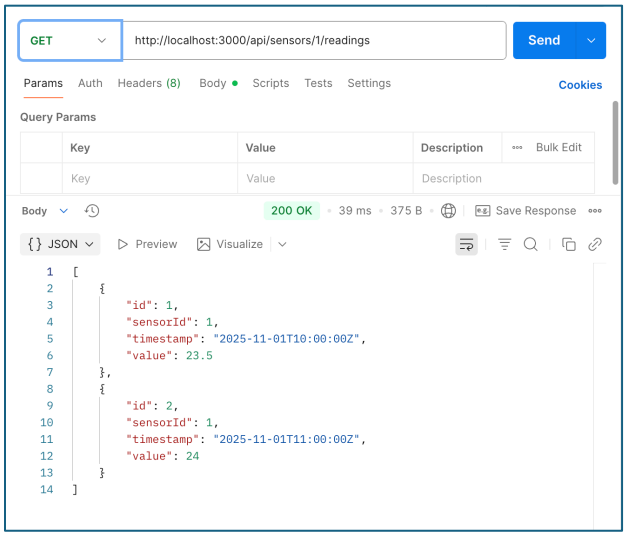


Request (PUT):

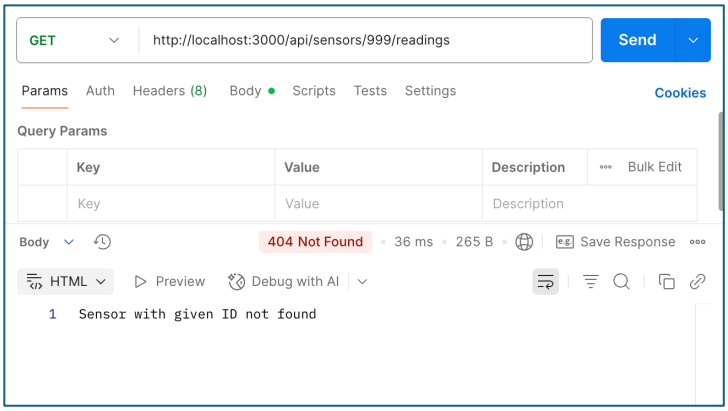


5. Nested Routes for Readings

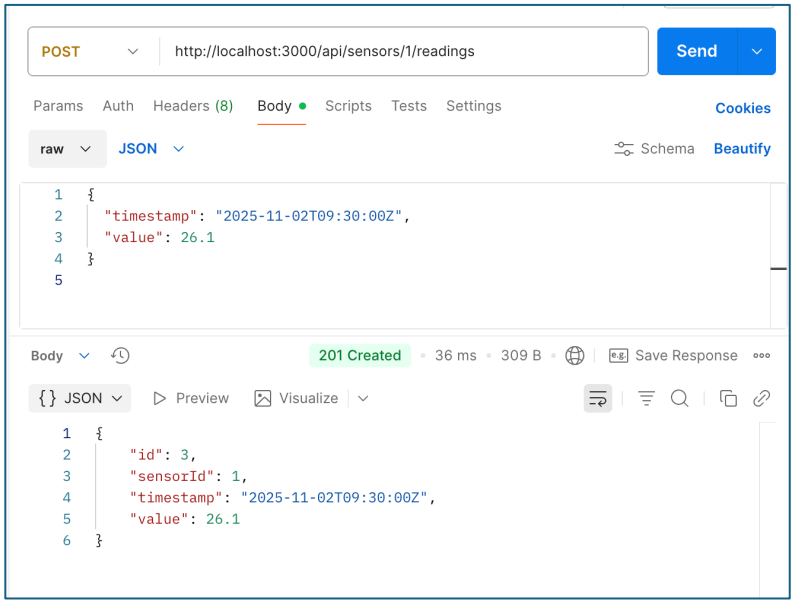
1 - GET readings for a valid sensor



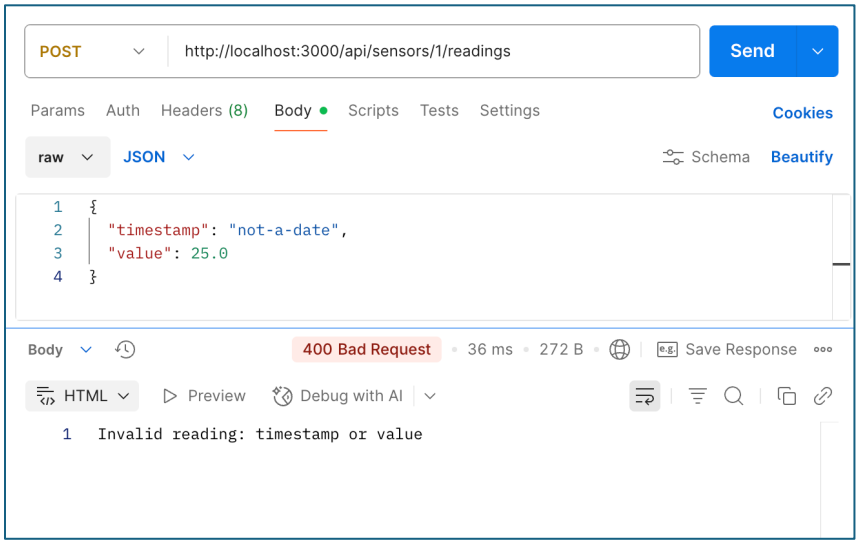
2 - GET readings for non-existent sensor



3 - POST a valid reading

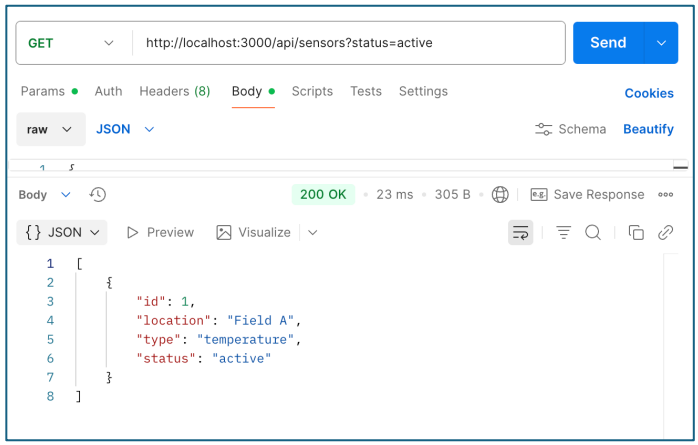


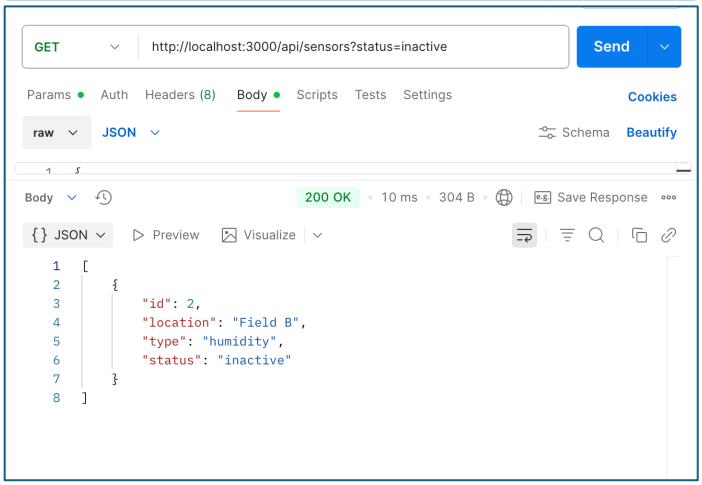
4 - POST invalid reading (bad timestamp or value)



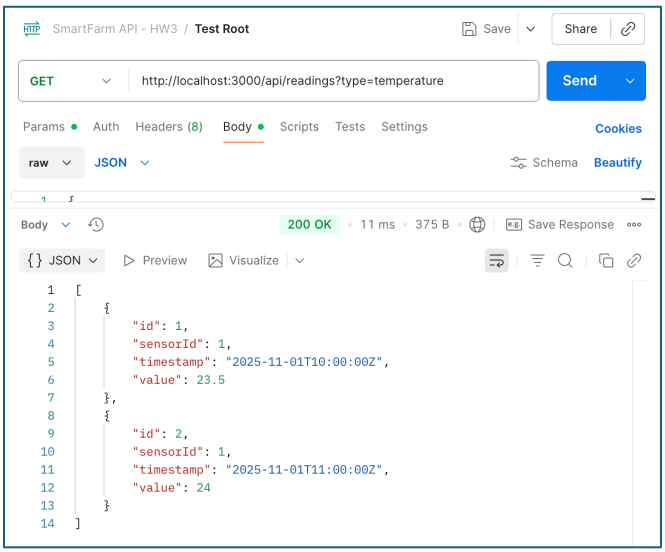
6. Query Parameters and Filtering

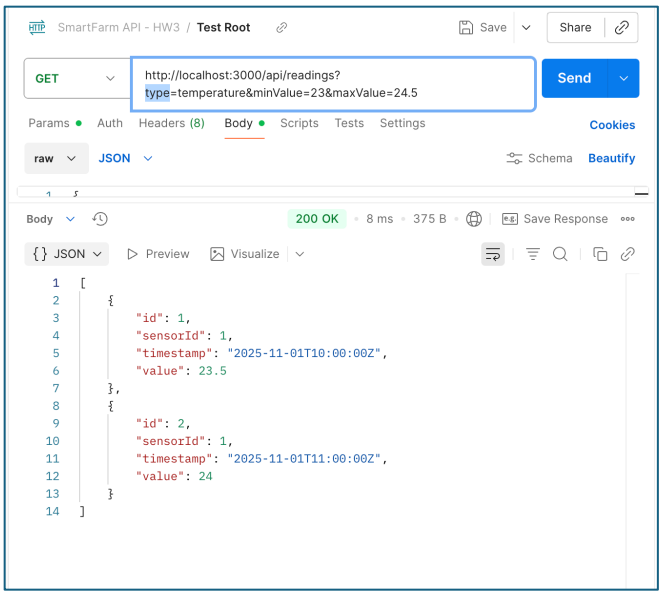
1 - Filter sensors by status





2 - Filter readings by type and values





These query features make the API flexible and useful for analytical queries.

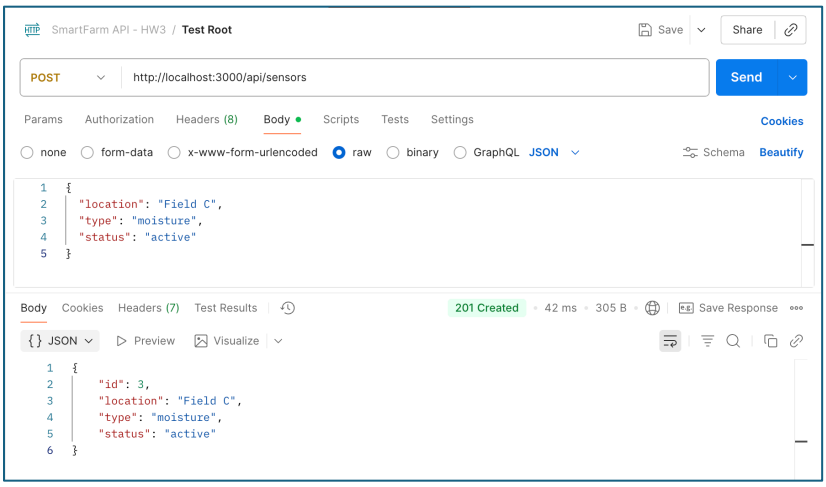
7. Middleware and Logging

The logging middleware was tested using Postman by sending multiple requests of different types (GET, POST, PUT, and nested routes). After each request, the terminal printed a corresponding log entry such as:

```
saitejanlp@Sais-MacBook-Pro smartfarm-api % node index.js

Listening on 3000...
GET /api/sensors - 21ms
GET /api/sensors/1/readings - 14ms
POST /api/sensors - 16ms
```

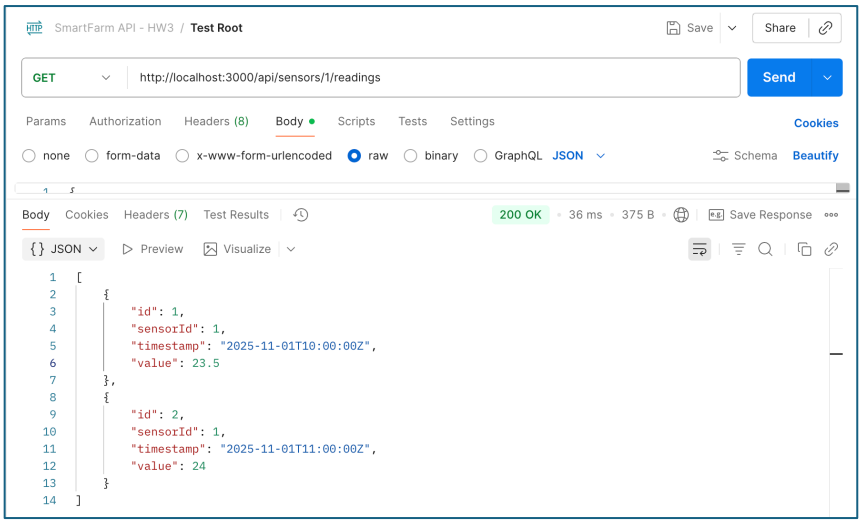
8. Testing Evidence

Successful Sensor Creation (POST)




```
saitejanlp@Sais-MacBook-Pro smartfarm-api % node index.js

  Listening on 3000...
  POST /api/sensors - 12ms
```

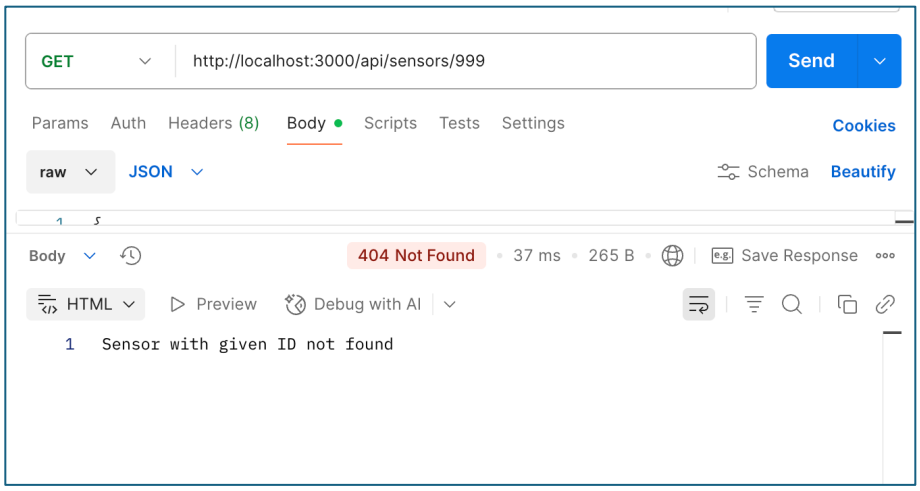
Reading Retrieval (GET)




```
saitejanlp@Sais-MacBook-Pro smartfarm-api % node index.js

  Listening on 3000...
  POST /api/sensors - 12ms
  GET /api/sensors/1/readings - 13ms
```

Invalid Request (400 or 404)

Invalid ID (404):




```
saitejanlp@Sais-MacBook-Pro smartfarm-api % node index.js

  Listening on 3000...
  POST /api/sensors - 12ms
  GET /api/sensors/1/readings - 13ms
  GET /api/sensors/999 - 13ms
```

Invalid POST Body (400):


```
saitejanlp@Sais-MacBook-Pro smartfarm-api % node index.js

  Listening on 3000...
  POST /api/sensors - 12ms
  GET /api/sensors/1/readings - 13ms
  GET /api/sensors/999 - 13ms
  POST /api/sensors - 12ms
```

10. Graduate-Students Only Tasks

(a) Pagination and Advanced Filtering

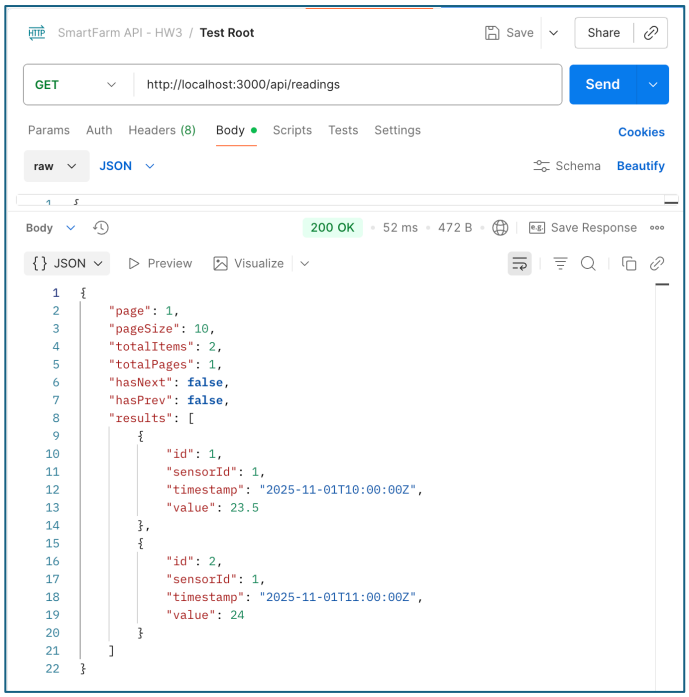
C1. Pagination defaults and bounds are correctly handled.

The `GET /api/readings` endpoint supports pagination with two query parameters: `page` and `limit`. When no query parameters are provided, the API defaults to `page = 1` and `limit = 10`, and returns the first page of readings along with pagination metadata (`page`, `pageSize`, `totalItems`, `totalPages`, `hasNext`, `hasPrev`). Invalid or out-of-range pagination values are rejected with clear error messages.
For example, a request such as: `GET /api/readings?limit=-5`, returns HTTP `400 Bad Request` with:`{ "error": "limit must be between 1 and 100" }`

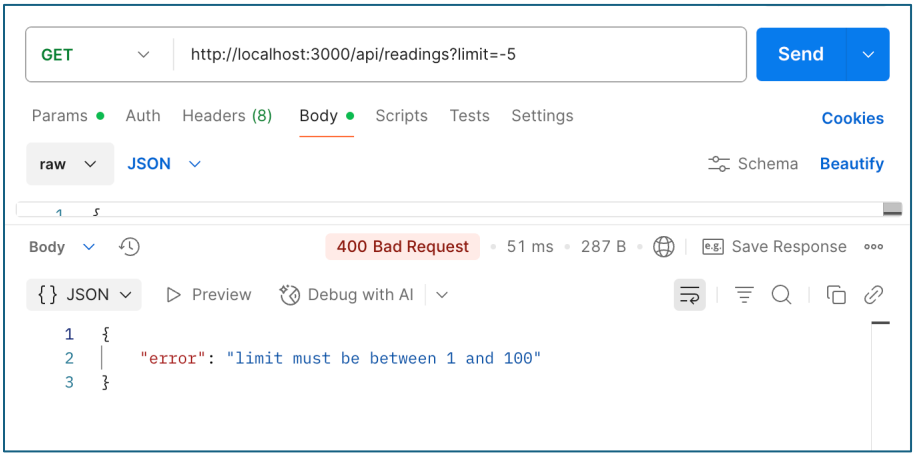
Similarly, invalid page values (e.g., `page=0`) are rejected with:
`{ "error": "page must be an integer greater than or equal to 1" }`
If the client requests an out-of-range page (e.g., `page=999` with `limit=10`), the endpoint returns an empty `results` array but still includes correct metadata, such as:

```
{
  "page": 999,
  "pageSize": 10,
  "totalItems": 2,
  "totalPages": 1,
  "hasNext": false,
  "hasPrev": true,
  "results": []
}
```

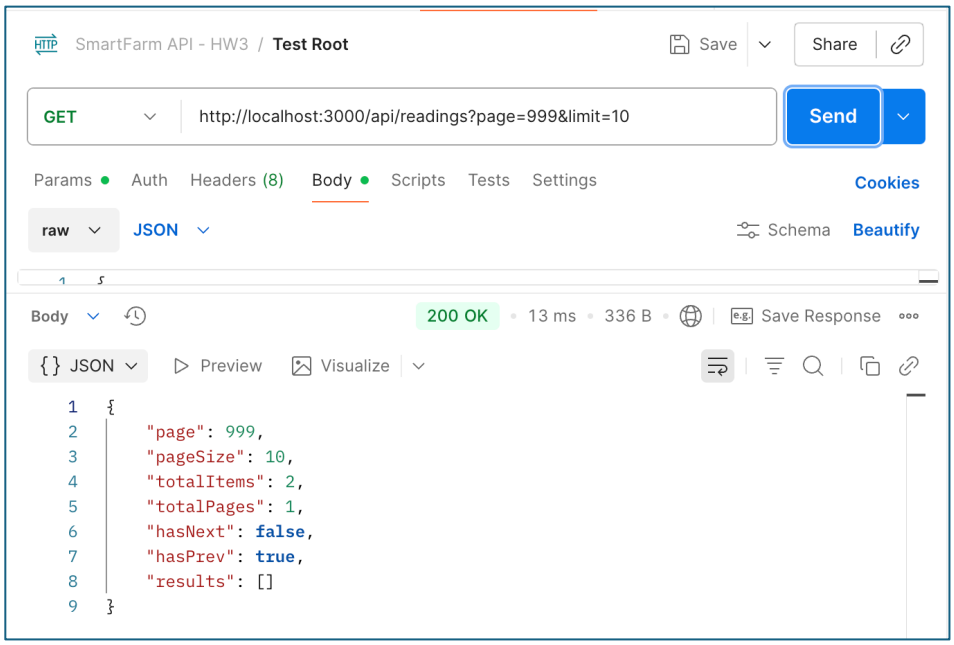
A - Defaults when no query parameters



B - Invalid limit (out of range)



C - Out-of-range page (empty results but valid metadata)



C2. Implements at least five filtering criteria:

The `GET /api/readings` endpoint supports multiple optional filters that can be used individually or combined. The following query parameters are implemented:

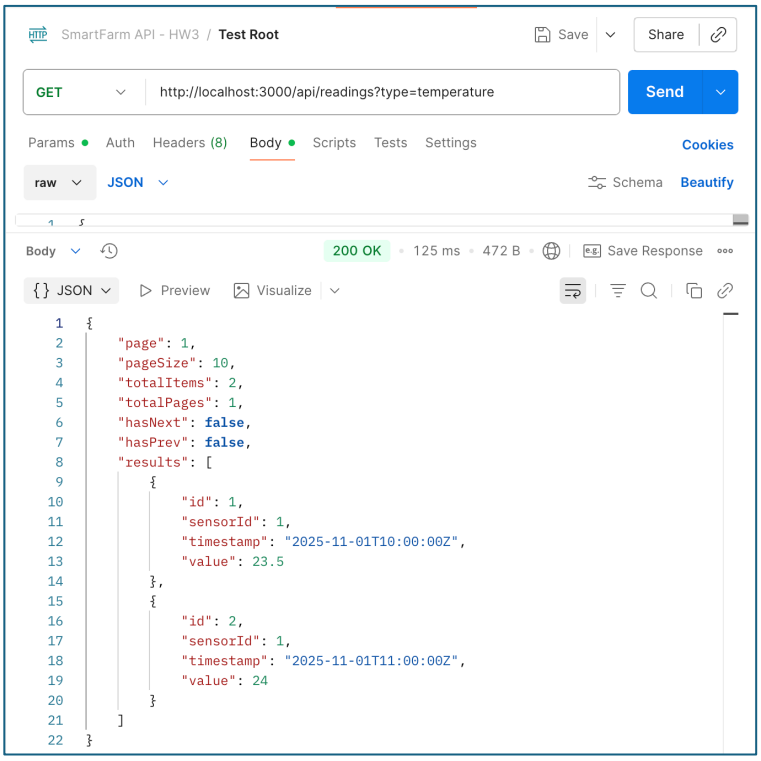
- `type` – filters readings to those whose parent sensor has the specified type (`temperature`, `humidity`, or `moisture`).
- `minValue` – returns readings whose `value` is greater than or equal to this number.
- `maxValue` – returns readings whose `value` is less than or equal to this number.
- `from` – returns readings whose `timestamp` is on or after the given ISO 8601 date/time.
- `to` – returns readings whose `timestamp` is on or before the given ISO 8601 date/time.

These filters can be combined. For example:

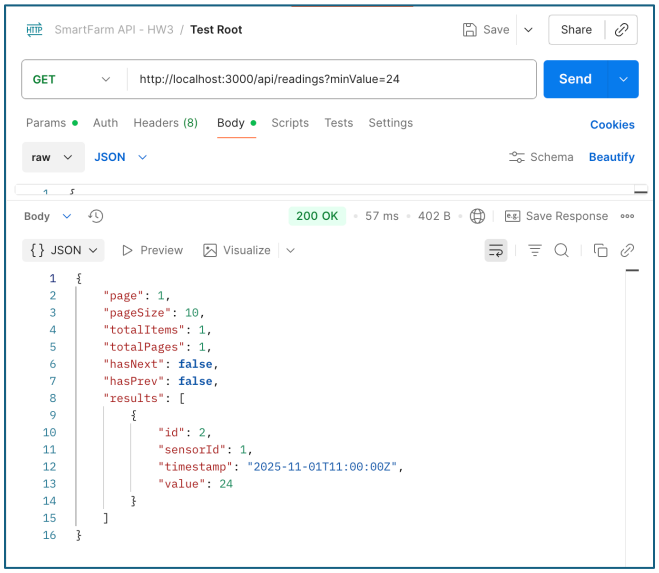
`GET/api/readings?type=temperature&minValue=23&maxValue=25&from=2025-11-01T00:00:00Z&to=2025-11-02T00:00:00Z`

returns only readings produced by temperature sensors, with values between 23 and 25, and timestamps within the specified time window.Invalid filter combinations are explicitly rejected with HTTP `400 Bad Request`. For example: `GET /api/readings?minValue=50&maxValue=10` returns: { "error": "minValue cannot exceed maxValue" } and `GET /api/readings?from=not-a-date` returns: { "error": "Invalid date format for 'from' parameter" } ensuring that the API does not silently accept inconsistent or malformed query parameters.

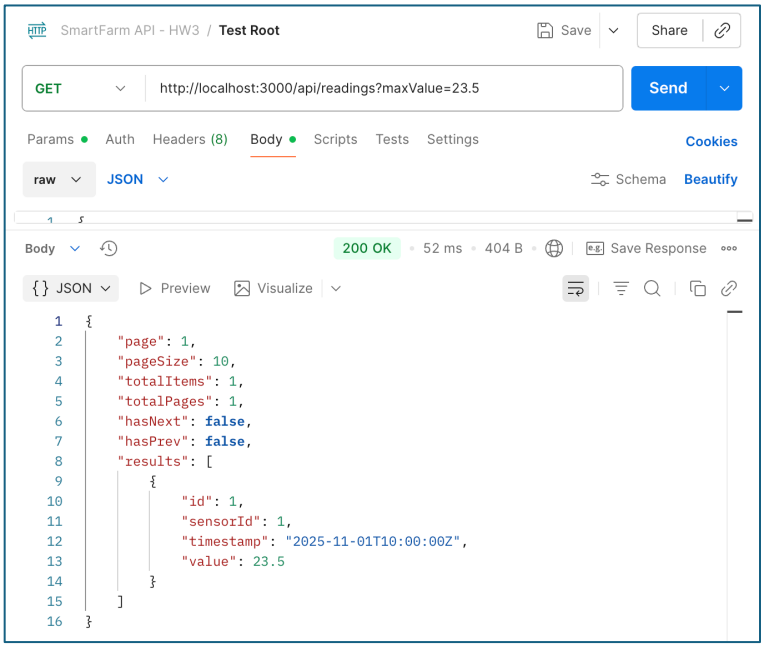
A. Filter by type

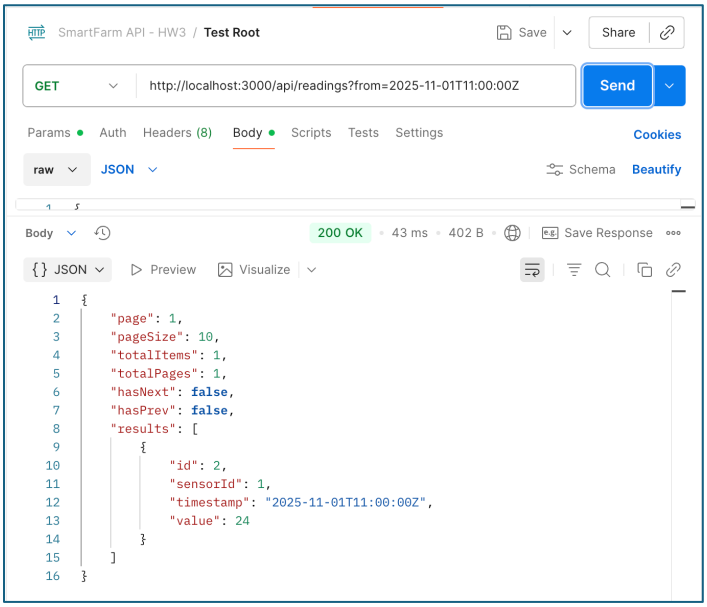


B. Filter by minValue

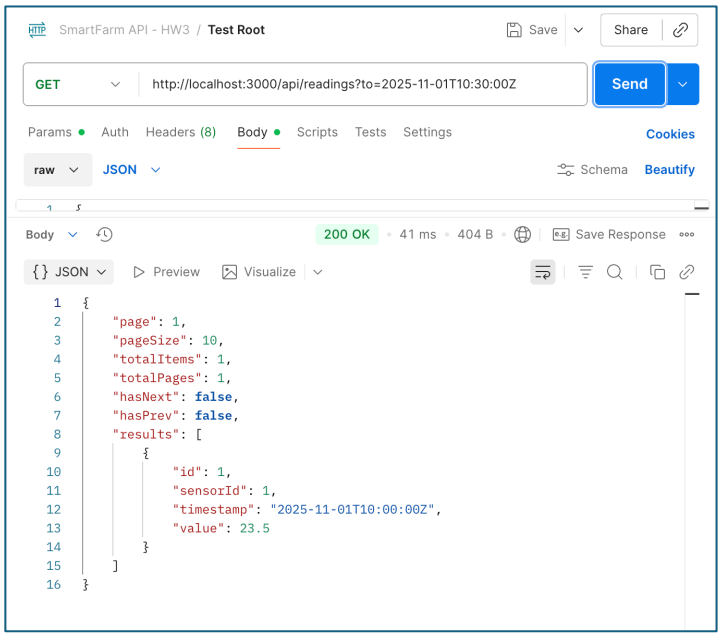


C. Filter by maxValue

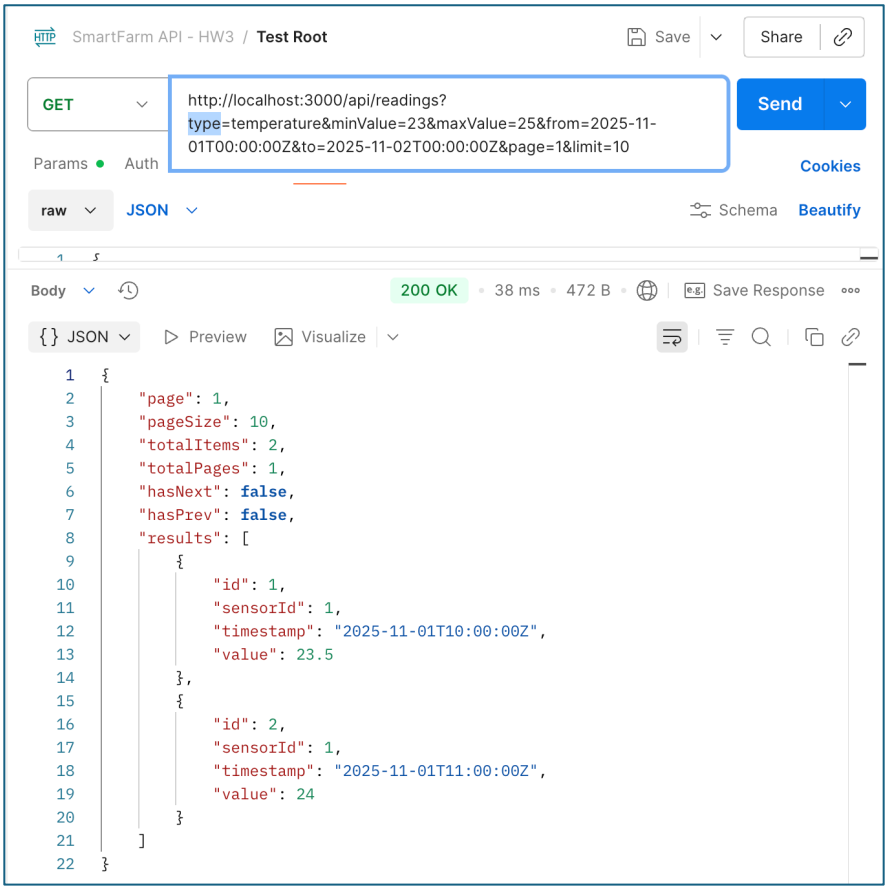




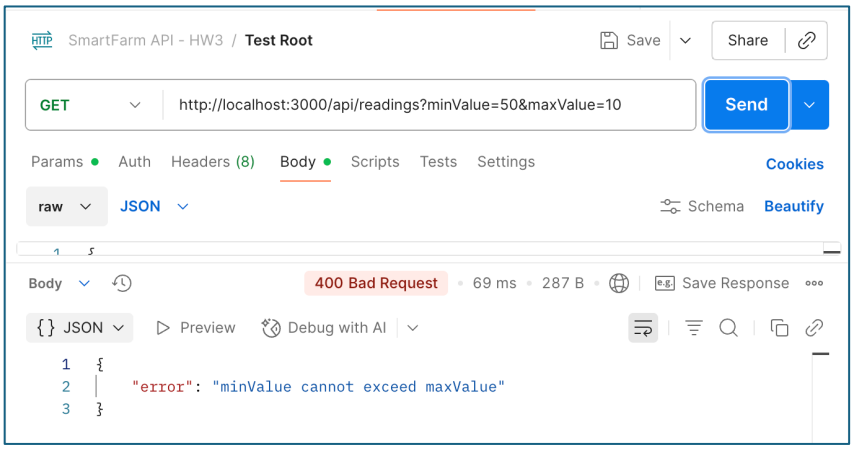
D. Filter by to (time <= to)



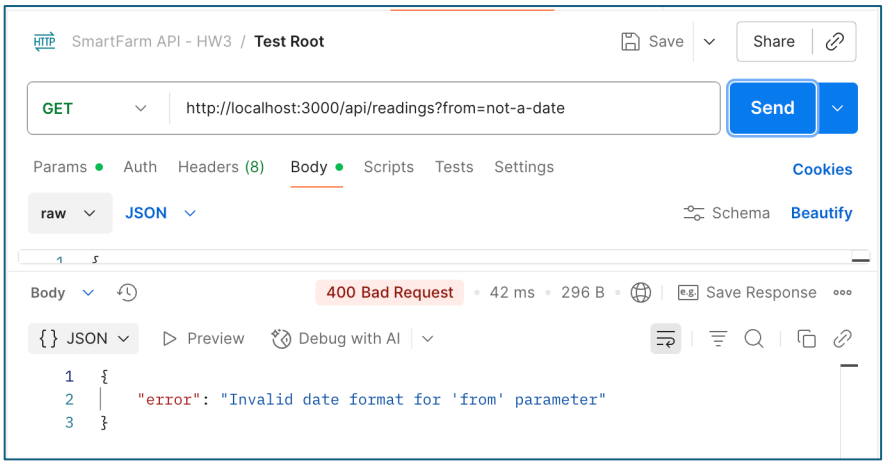
E. Combined filters



F. Invalid combination - minValue > maxValue



G. Invalid timestamps (from / to)



C3. Response includes correct pagination metadata.

The `GET /api/readings` endpoint always returns a consistent JSON structure that includes pagination metadata along with the data itself. Every successful response (HTTP 200) contains the following fields:

- `page` – the current page number.
- `pageSize` – the number of items per page (based on the `limit` parameter).
- `totalItems` – the total number of readings that match the current filters.
- `totalPages` – the total number of pages available.
- `hasNext` – `true` if there is a page after the current one.
- `hasPrev` – `true` if there is a page before the current one.
- `results` – an array of reading objects for the current page (which may be empty).

For example, a basic request such as:
`GET /api/readings?page=1&limit=10`
produces a response of the form:

```
{
  "page": 1,
  "pageSize": 10,
```

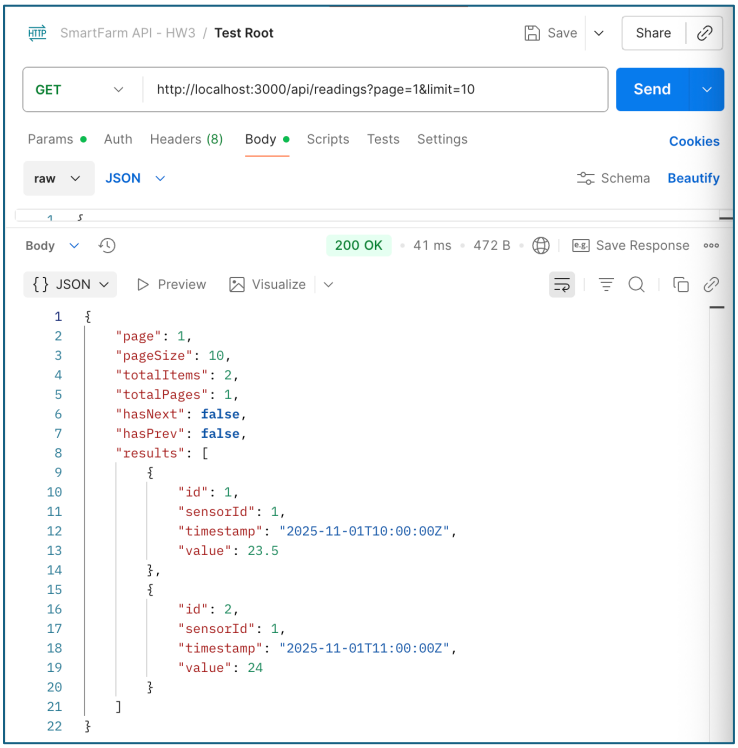
```
  "totalItems": 2,
  "totalPages": 1,
  "hasNext": false,
  "hasPrev": false,
  "results": [ /* reading objects */ ]
}
```

Even when no results are returned (for example, requesting an out-of-range page or a filter that matches zero readings), the API still includes all metadata fields with an empty `results` array, such as:

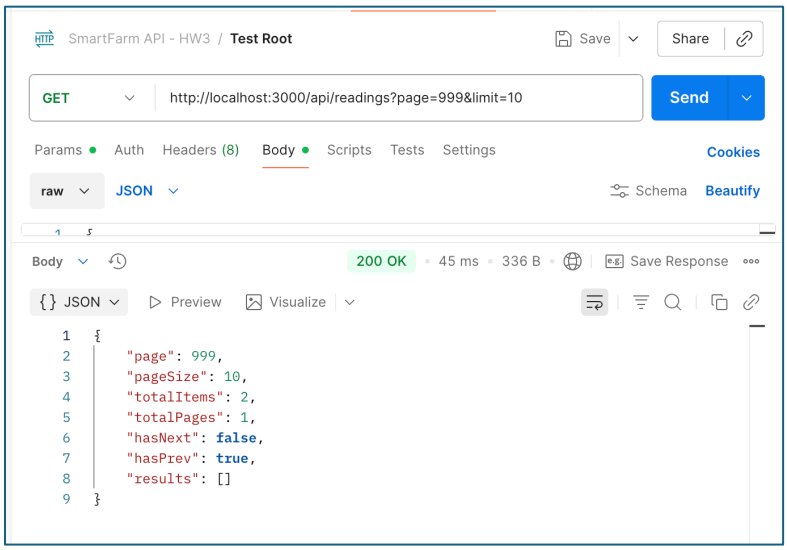
```
{
  "page": 999,
  "pageSize": 10,
  "totalItems": 2,
  "totalPages": 1,
  "hasNext": false,
  "hasPrev": true,
  "results": []
}
```

This consistent response shape satisfies the requirement that all pagination metadata be present in every successful response, regardless of whether any readings are returned.

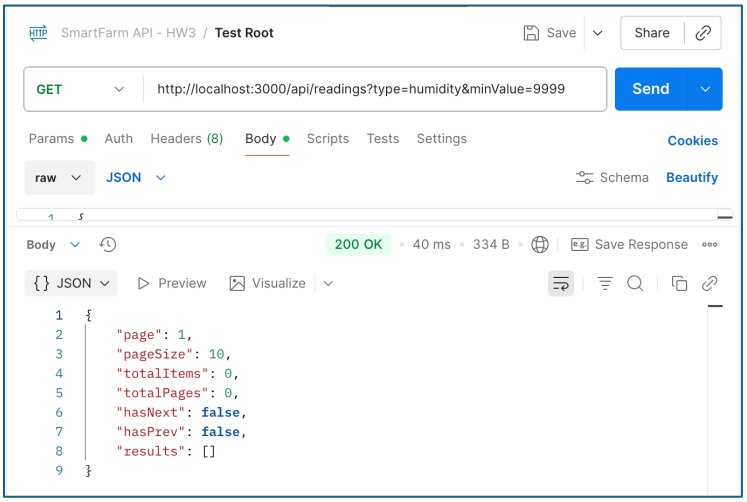
1 - Normal page with data



2 - Valid request but **no results** (metadata still present) An out-of-range page:



3 - a filter that matches nothing



C4. Invalid parameters trigger meaningful error messages.

The `GET /api/readings` endpoint validates all query parameters and rejects invalid inputs with HTTP `400 Bad Request` and a JSON response that always includes an `"error"` field. This prevents the API from crashing or returning HTML error pages.

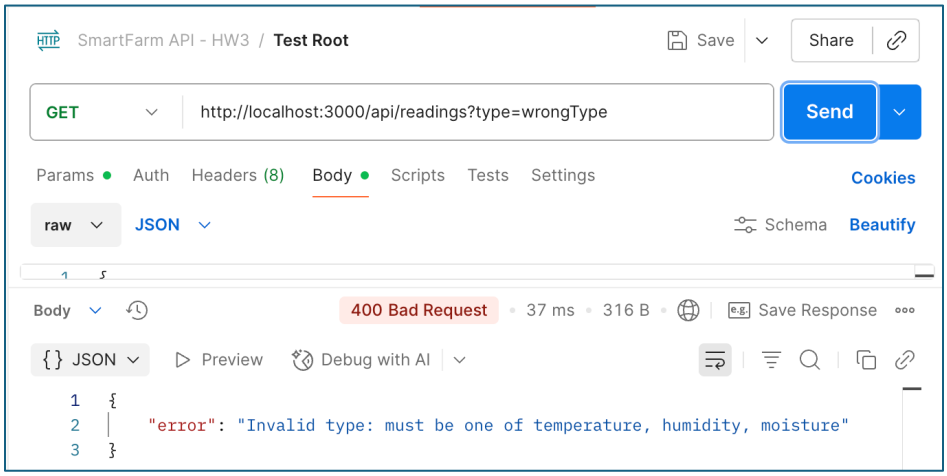
Examples of invalid inputs and their responses:

- Invalid sensor type:
- `GET /api/readings?type=wrongType`
  returns:
  `{ "error": "Invalid type: must be one of temperature, humidity, moisture" }`
- Inconsistent numeric range:
- `GET /api/readings?minValue=50&maxValue=10`
  returns:
  `{ "error": "minValue cannot exceed maxValue" }`

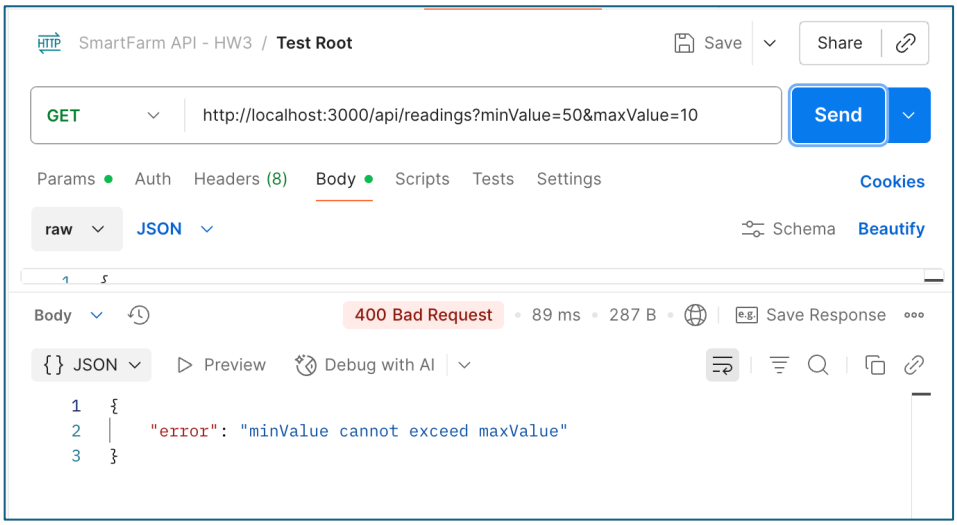
- Malformed timestamp:
- `GET /api/readings?from=not-a-date`
  returns:
  `{ "error": "Invalid date format for 'from' parameter" }`

Additional checks ensure that `minValue` and `maxValue`, when present, are numeric, and that `from` is not later than `to`. All such errors result in a `400` status with a clear JSON error message, satisfying the requirement that invalid parameters trigger meaningful error feedback.

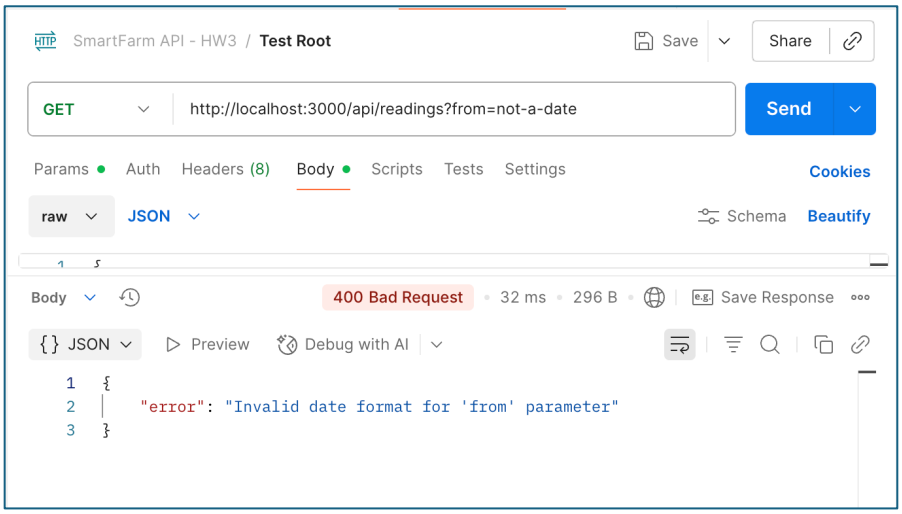
A- Invalid type



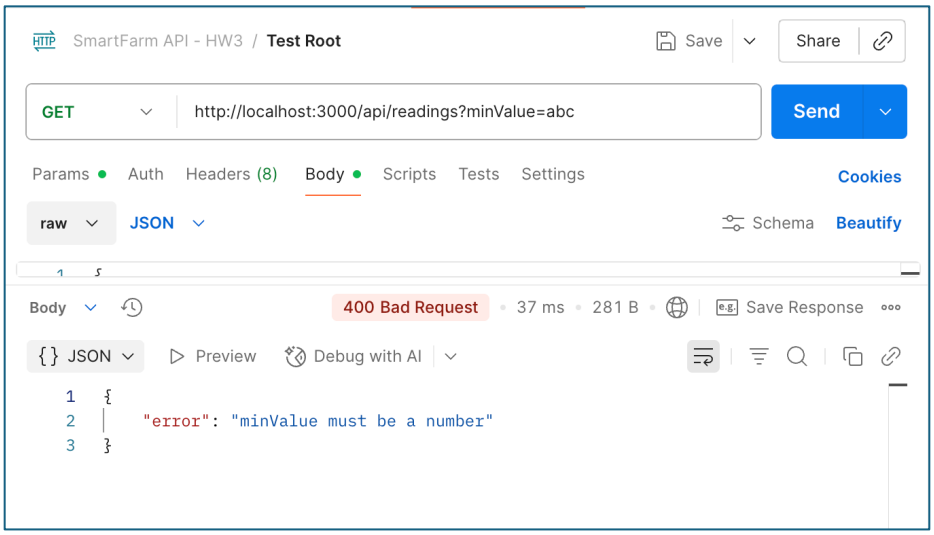
B - minValue > maxValue



C - Invalid from date

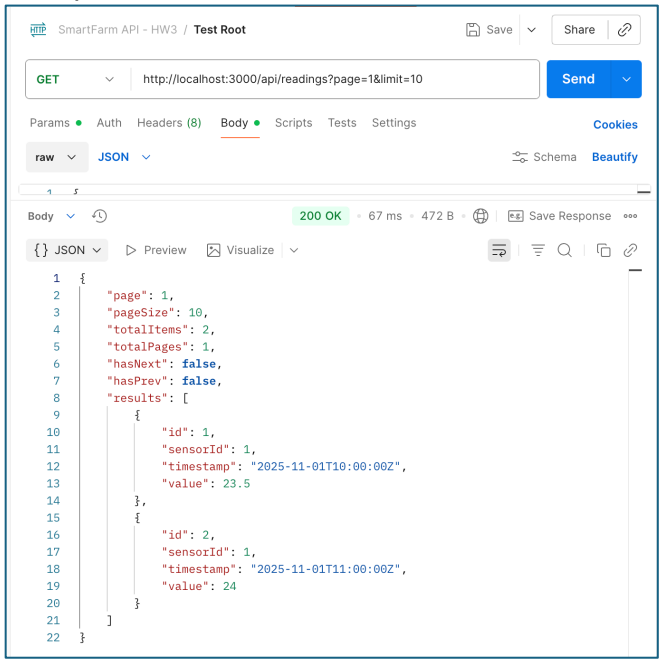


D - Non-numeric minValue

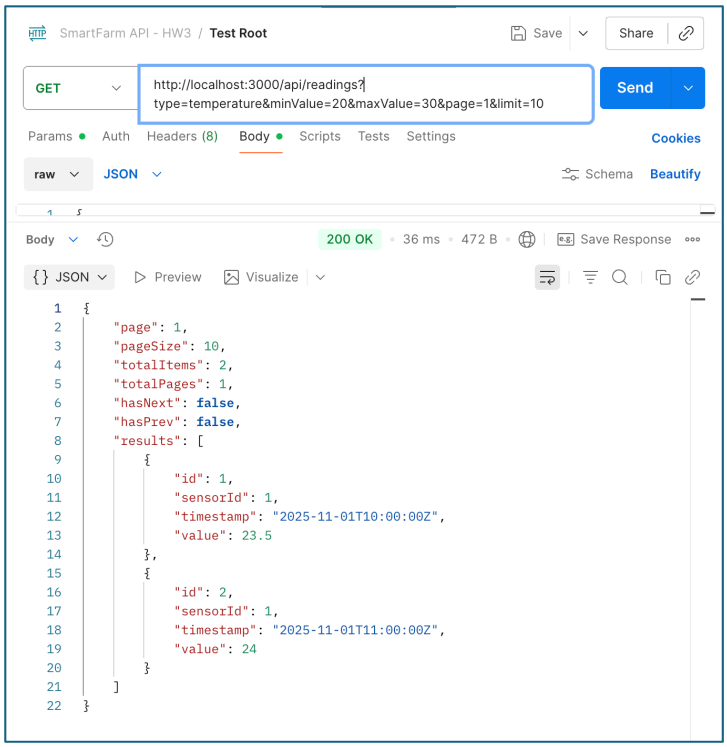


C5. Evidence of functionality provided in Postman or curl screenshots.

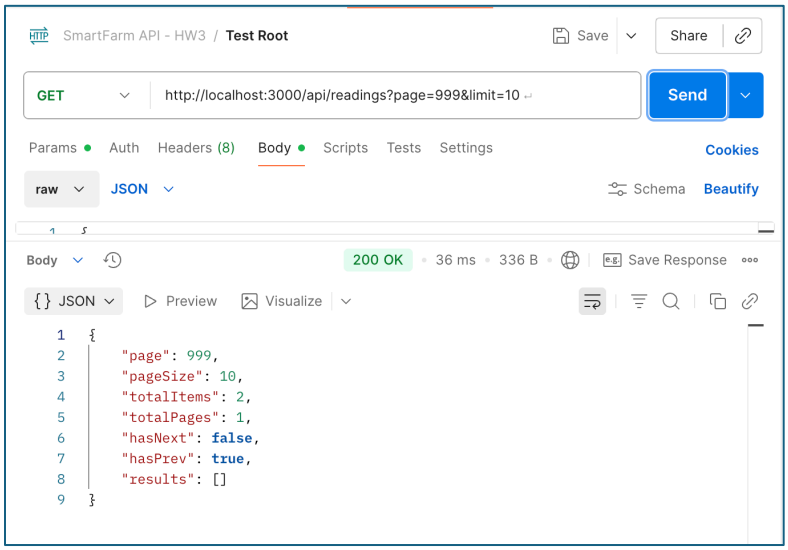
- Basic paginated response:



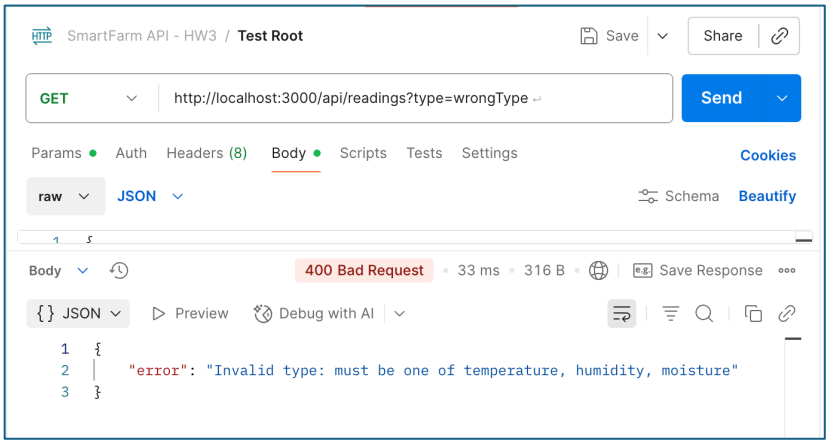
- Combined filters working together:



- Out-of-range page → empty results but valid metadata:



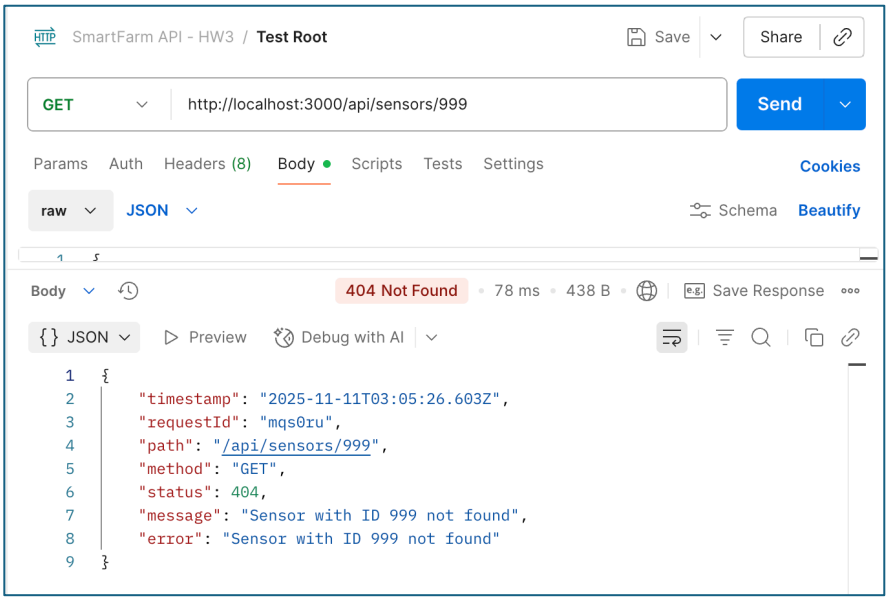
- Invalid query → 400 with JSON error:



(b) Centralized Error Middleware and Timing Report

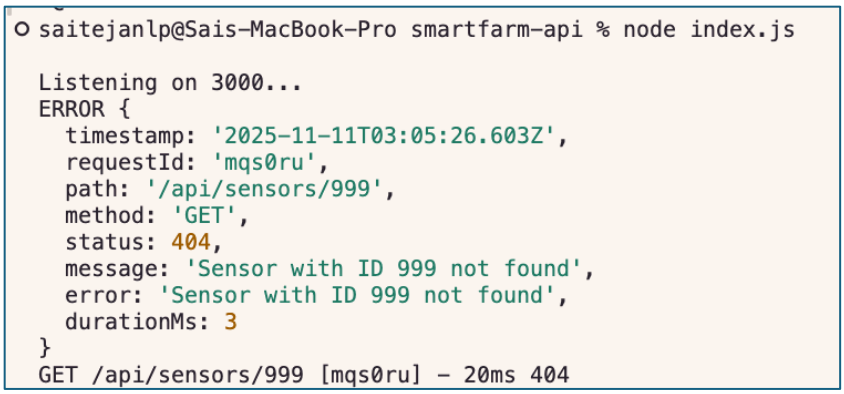
C1. Every error response must include key diagnostic fields

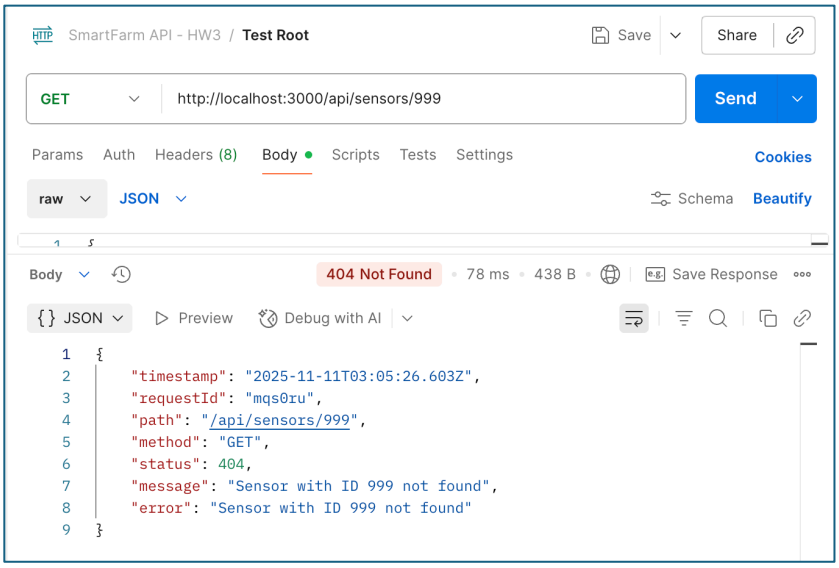
Test: non-existent sensor → 404



C2. Console or log file output includes duration and contextual details.

This log includes the same fields returned in the JSON error payload (timestamp, requestId, path, method, status, message), plus durationMs, which records how long the request took from start to error handling. In development mode, the middleware can also attach the stack trace to the response payload for debugging, while in production mode only the standardized fields are exposed. These logs provide a concise but complete view of when and where an error occurred and how long it took, satisfying the requirements for centralized error monitoring.





C3. Routes must correctly forward errors using next(err).

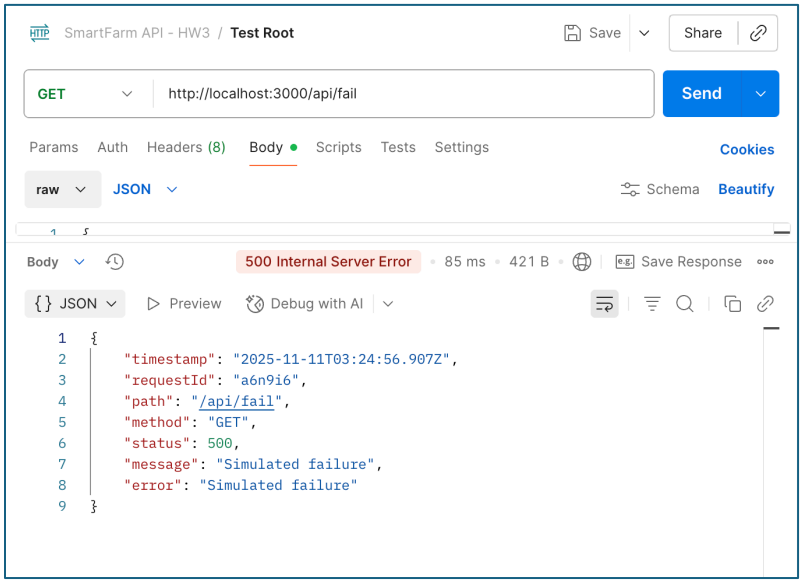
To prove that error handling is centralized, I added a dedicated route `/api/fail` that intentionally throws an error and forwards it using Express's `next(err)` mechanism:

```
app.get("/api/fail", (req, res, next) => {
  const err = new Error("Simulated failure");
  err.statusCode = 500;
  err.publicMessage = "Simulated failure";
  next(err);
});
```

When a client calls `GET /api/fail`, the route does not send a response directly. Instead, it passes the error object to the global error-handling middleware. The middleware then computes the request duration, logs a structured `ERROR` entry to the console

Test: 500 – simulated internal server error

```
ERROR {
  timestamp: '2025-11-11T03:24:56.907Z',
  requestId: 'a6n9i6',
  path: '/api/fail',
  method: 'GET',
  status: 500,
  message: 'Simulated failure',
  error: 'Simulated failure',
  durationMs: 7
}
GET /api/fail [a6n9i6] - 17ms 500
```

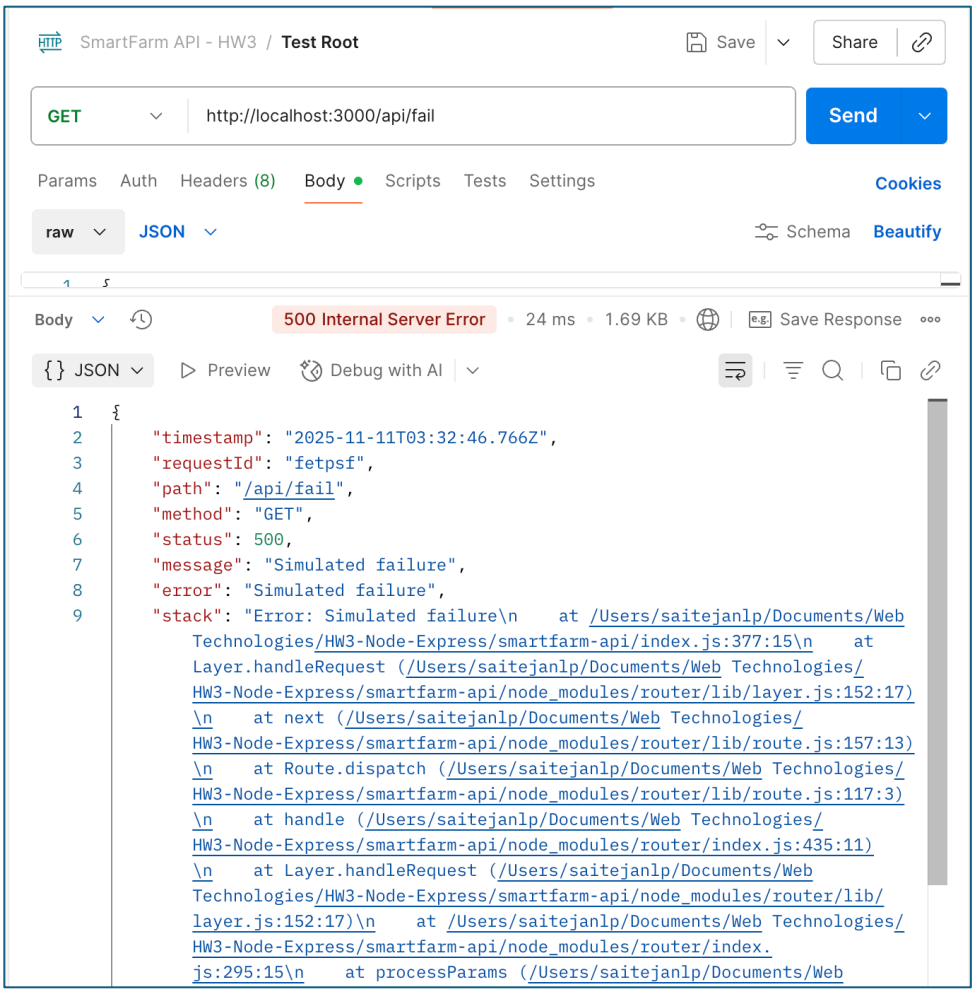


The same pattern is used for validation errors and 404 lookups (e.g., missing sensors), which also call next(err) with appropriate status codes. This confirms that route-level errors are not handled ad hoc, but are consistently forwarded to and processed by a single, centralized error middleware.

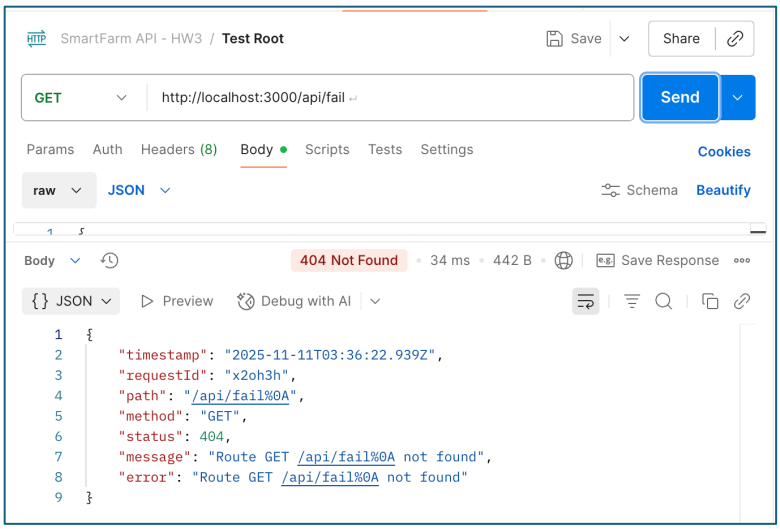
C4. No stack traces or sensitive details are leaked in production mode.

A. Development mode (stack included)
NODE_ENV=development node index.js

```
saitejanlp@Sais-MacBook-Pro smartfarm-api % NODE_ENV=development node index.js
Listening on 3000...
ERROR {
  timestamp: '2025-11-11T03:32:46.766Z',
  requestId: 'fetpsf',
  path: '/api/fail',
  method: 'GET',
  status: 500,
  message: 'Simulated failure',
  error: 'Simulated failure',
  durationMs: 0
}
GET /api/fail [fetpsf] - 9ms 500
```



B. Production mode (stack **hidden**)
NODE_ENV=production node index.js



The global error handler was implemented to prevent stack traces or internal details from being exposed in production. The behavior depends on the `NODE_ENV` variable:

- When `NODE_ENV=development`, the middleware attaches the `err.stack` trace to the JSON payload to assist with debugging.
- When `NODE_ENV=production`, the `stack` field is omitted entirely, ensuring that the client only sees standardized, non-sensitive fields (`timestamp`, `requestId`, `path`, `method`, `status`, `message`, `error`).

When the same request is executed in production mode (NODE_ENV=production), the stack field is omitted

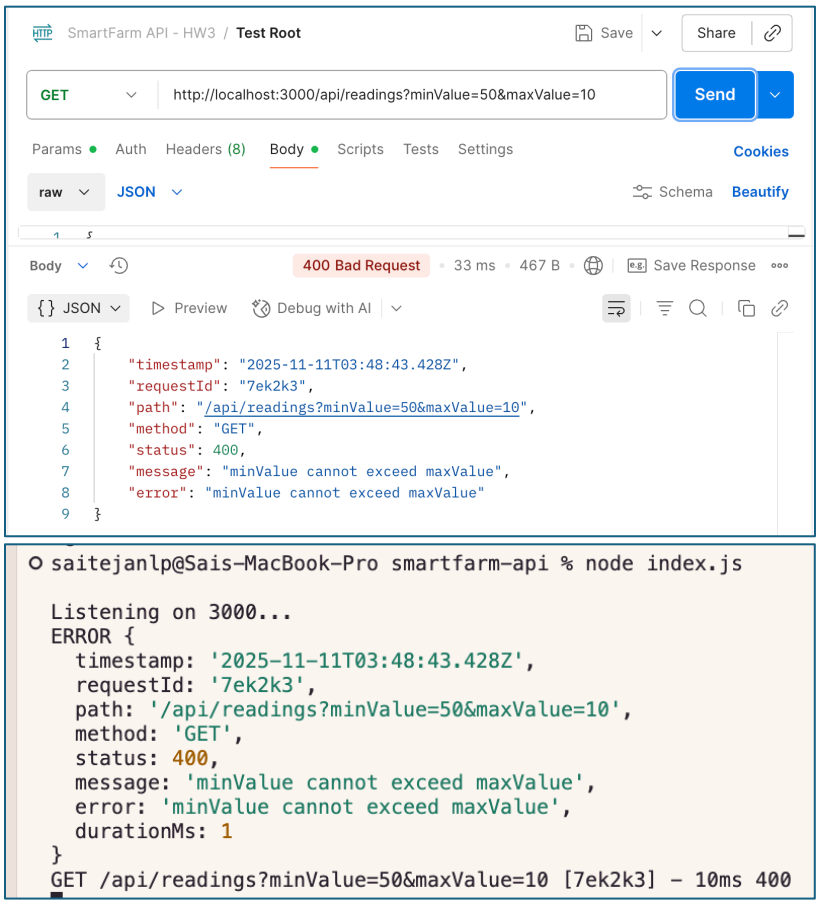
C5. Evidence in Postman/curl screenshots for multiple error types.

- 400 - Validation error (Bad Request)

Option A (clean): invalid query on `/api/readings`

This hits the check:

```
if (minVal !== undefined && maxVal !== undefined && minVal > maxVal) {
  return next(createError(400, "minValue cannot exceed maxValue"));
}
```

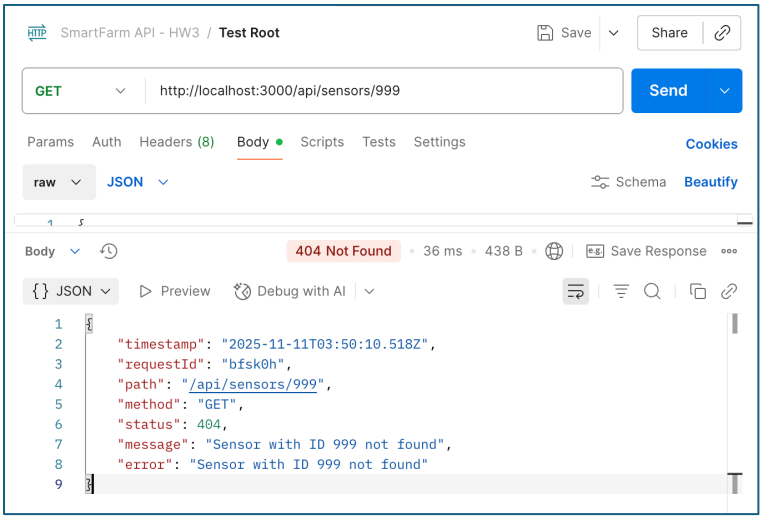


- 404 - Not Found (missing sensor or reading)

Test: non-existent sensor
This hits:

```
if (!sensor) {
  return next(createError(404, `Sensor with ID ${id} not found`));
}
```

```
ERROR {
  timestamp: '2025-11-11T03:50:10.518Z',
  requestId: 'bfsk0h',
  path: '/api/sensors/999',
  method: 'GET',
  status: 404,
  message: 'Sensor with ID 999 not found',
  error: 'Sensor with ID 999 not found',
  durationMs: 7
}
GET /api/sensors/999 [bfsk0h] - 15ms 404
```

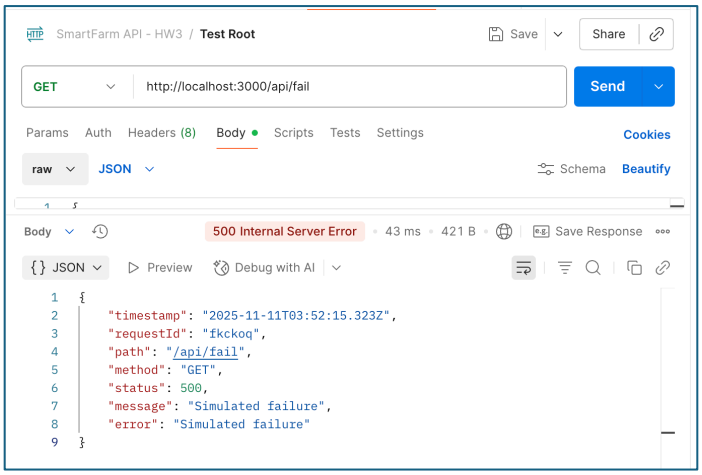


- 500 – Internal Server Error (/api/fail)

Test: simulated failure

This hits:

```
app.get("/api/fail", (req, res, next) => {
  const err = new Error("Simulated failure");
  err.statusCode = 500; // Internal Server Error
  err.publicMessage = "Simulated failure";
  next(err);
});
```



```
ERROR {
  timestamp: '2025-11-11T03:52:15.323Z',
  requestId: 'fkckoq',
  path: '/api/fail',
  method: 'GET',
  status: 500,
  message: 'Simulated failure',
  error: 'Simulated failure',
  durationMs: 6
}
GET /api/fail [fkckoq] - 18ms 500
```