WeatherSense Testing report

Prepared For

Assoc.Prof. Worasit Choochaiwattana, Ph.D.
Withawat Tangtrongpairoj, Ph.D.

Prepared By

6510545748	Sukprachoke	Leelapisuth
6510545721	Wissarut	Kanasub

This Project is collected and evaluated in 01219343 Software Testing

Software and knowledge Engineers

Overview

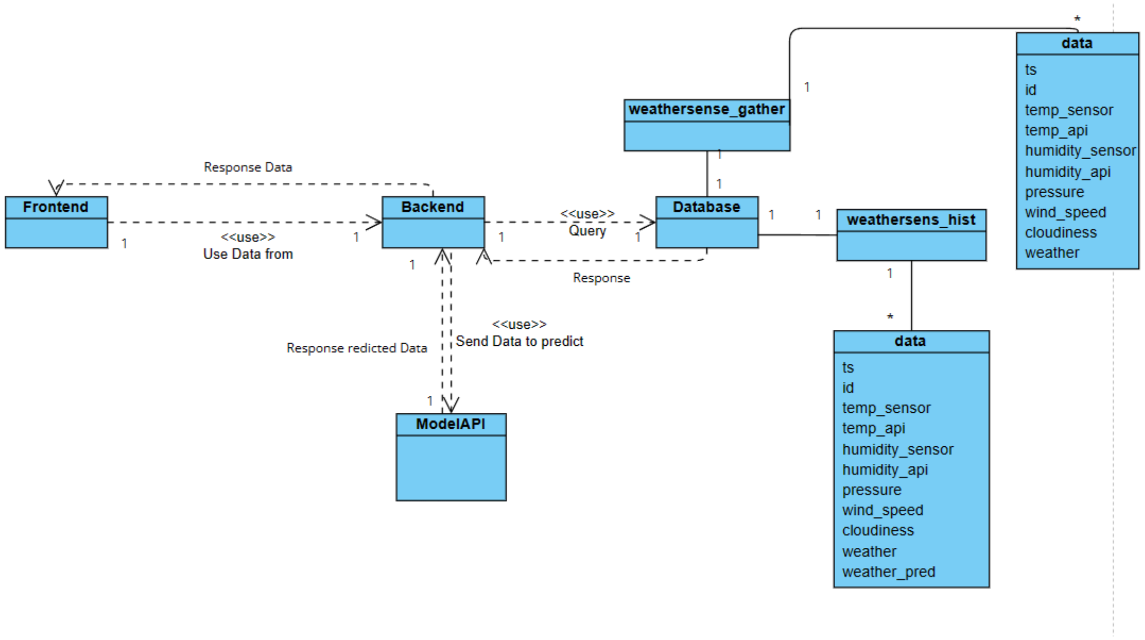
WeatherSense is a weather tracking website that updates weather data and classification every 10 minutes. Leveraging our trained random forest classification model, WeatherSense provides accurate classification and historical weather information. The system corrects data using the Ky-015 sensor and integrates with the OpenWeatherMap API to ensure reliability. Node-RED assists in data correction.

Our project comprises three major components:

- Frontend: Developed using the SvelteKit framework.
- Backend: Developed with Node.js using Express.js, Node-RED, and KU MySQL server.
- ModelAPI-Server: Developed using Flask.

ISP Part

Domain Model



Use Cases

Use Case Number	1
Use Case Name	Show the latest data.
Actor(s)	User
Description	This use case describes the process of accessing the latest data available in the system.
Precondition(s)	The system is operational and up-to-date with the latest data.
Main Flow	The user opens the website or application. The system retrieves and displays the most recent data. The user views the latest data presented on the homepage/dashboard.
Postcondition(s)	The user has viewed the latest data.

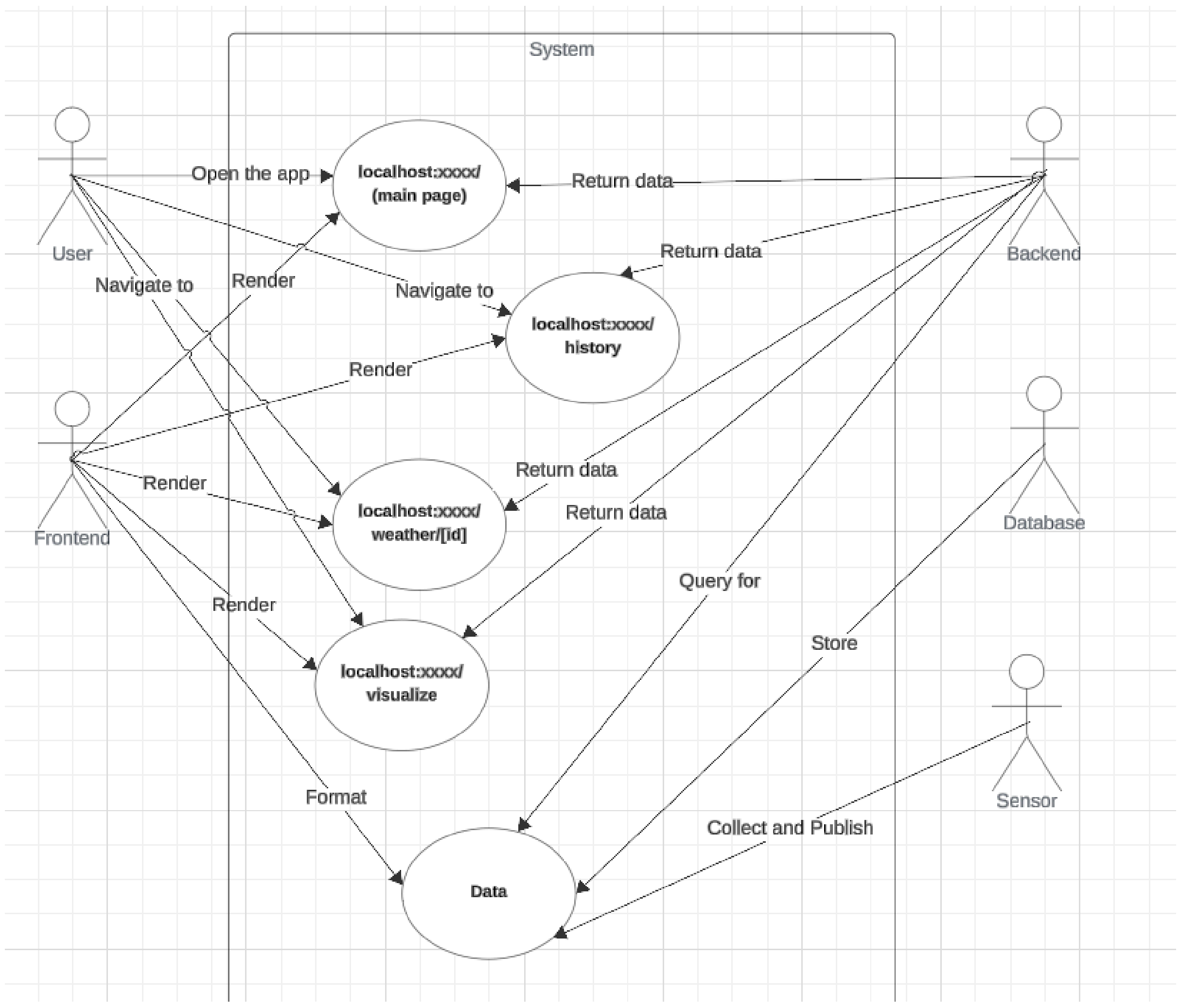
Use Case Number	2
Use Case Name	Show the list of previous data.
Actor(s)	User
Description	This use case describes the process of accessing a chronological list of historical data entries.
Precondition(s)	Historical data entries are available in the system.
Main Flow	The user navigates to the "History" section of the website or application. The system retrieves and displays a list of previous data entries, each with its associated timestamp. The user scans the list to find the desired historical data entry.
Postcondition(s)	The user has accessed the list of previous data entries with timestamps.

Use Case Number	3
Use Case Name	Show specific data in history.
Actor(s)	User
Description	This use case describes the process of viewing detailed information about a specific data entry at a particular point in time.
Precondition(s)	The user has accessed the "History" section of the website or application. Historical data entries are available in the system.
Main Flow	The user selects a specific data entry from the list of historical data. The system retrieves and displays detailed information about the selected data entry. The user reviews the specific data at the chosen point in time.
Postcondition(s)	The user has viewed detailed information about a specific data entry from the historical records.

Use Case Number	4
Use Case Name	Show visualized data.
Actor(s)	User
Description	This use case involves visualizing weather data retrieved from the backend system.
Precondition(s)	Historical data entries are available in the system.
Main Flow	The user navigates to the "Data Visualize" section of the website or application. The system retrieves weather data from the backend using the provided API endpoint.

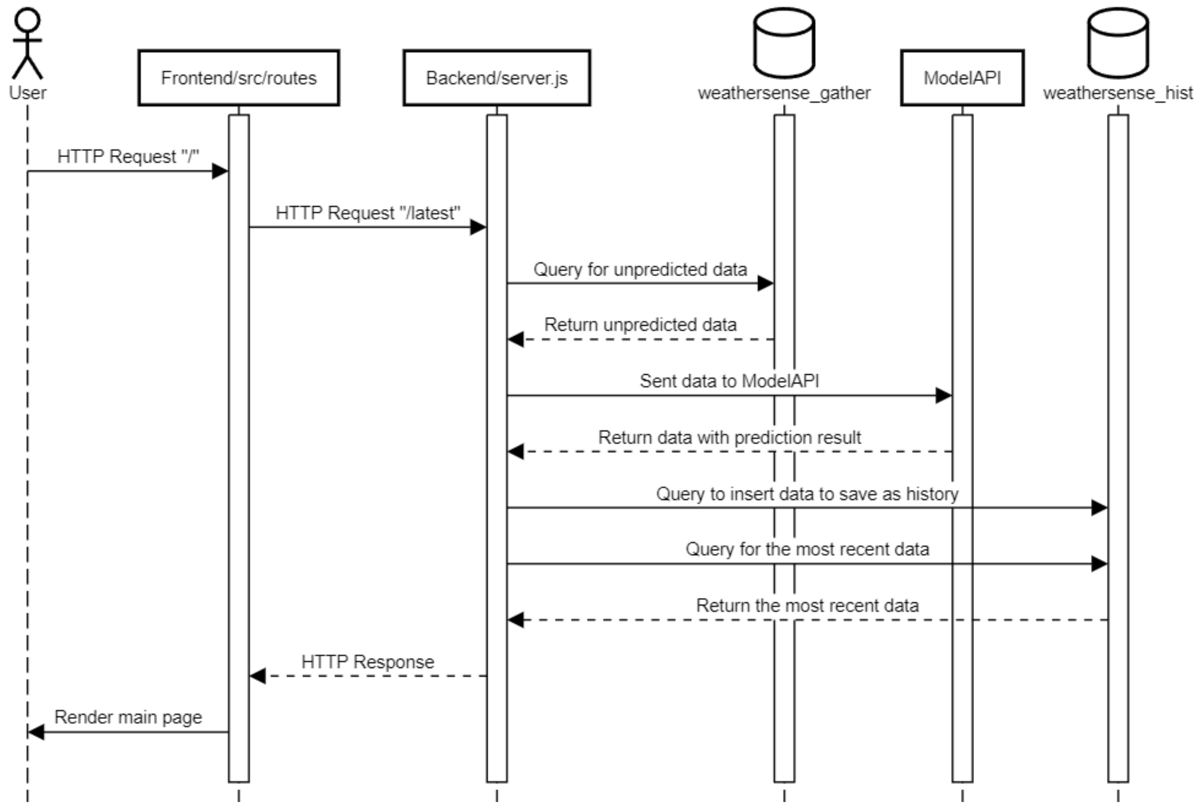
	The system processes the retrieved data to generate visualizations such as charts, graphs, or maps. The user views the visualized data in the form of charts, graphs or maps.
Postcondition(s)	The user has viewed the detailed visualization of data in the form of charts, graphs or maps.

Use Case Diagram

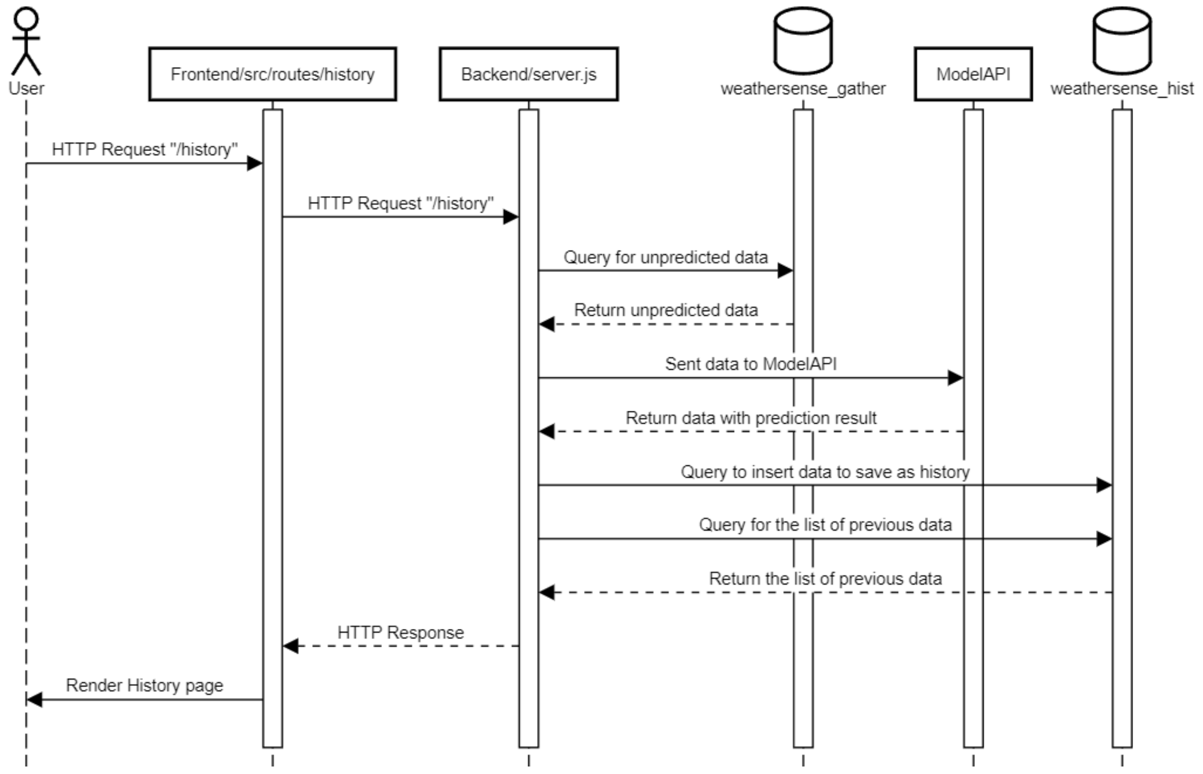


Sequence Diagram

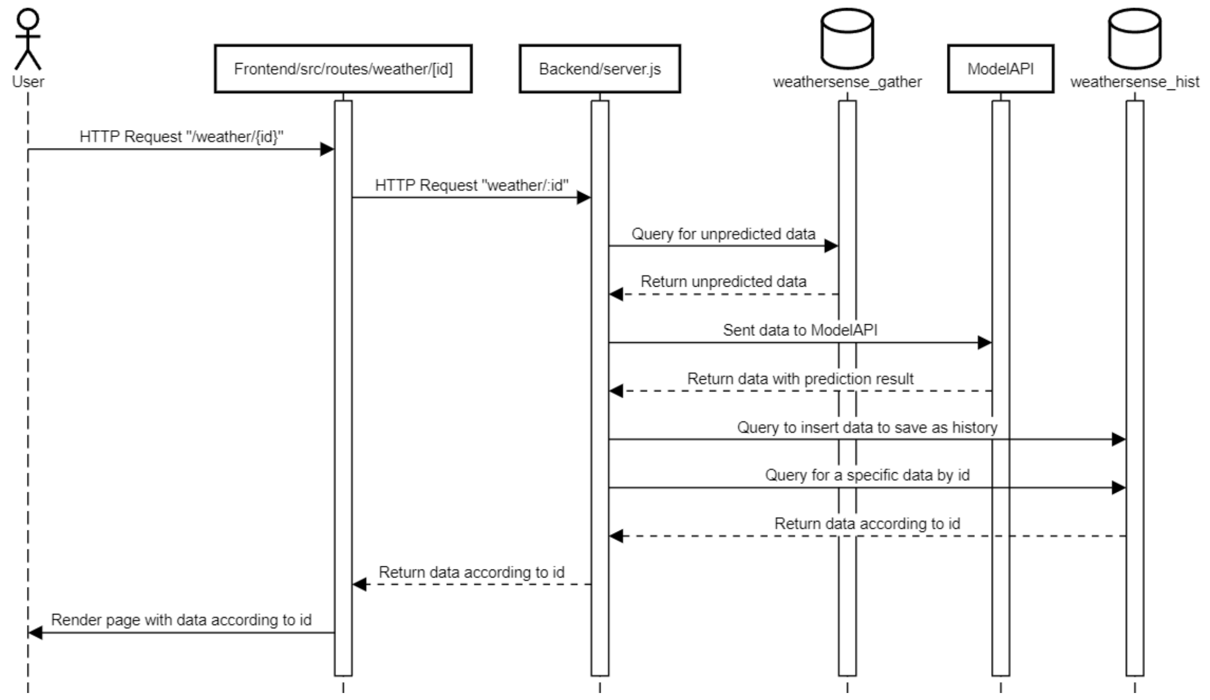
User go to the main page of the website



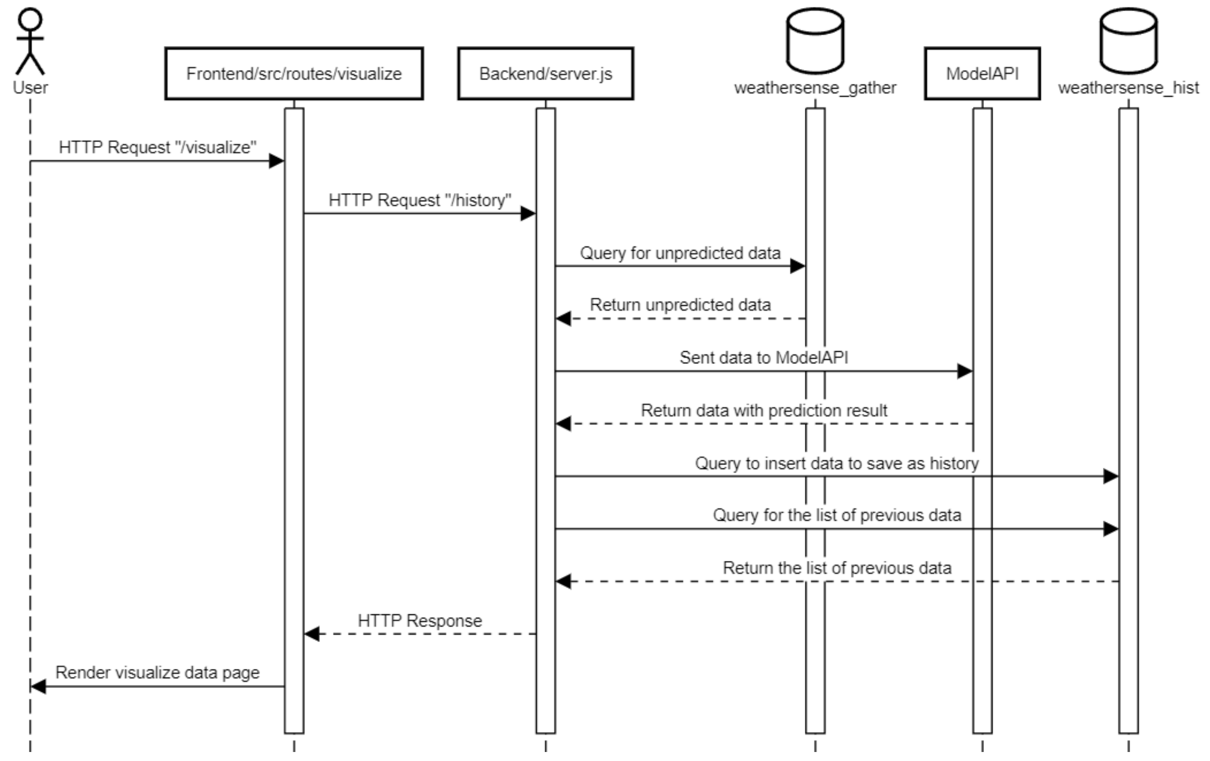
User go to History page "/history"


User
Frontend/src/routes/history
Backend/server.js
weathersense_gather
ModelAPI
weathersense_hist
HTTP Request "/history"
HTTP Request "/history"
Query for unpredicted data
Return unpredicted data
Sent data to ModelAPI
Return data with prediction result
Query to insert data to save as history
Query for the list of previous data
Return the list of previous data
HTTP Response
Render History page

User go to specific weater id page


User
Frontend/src/routes/weather/[id]
Backend/server.js
weathersense_gather
ModelAPI
weathersense_hist
HTTP Request "/weather/{id}"
HTTP Request "weather/:id"
Query for unpredicted data
Return unpredicted data
Sent data to ModelAPI
Return data with prediction result
Query to insert data to save as history
Query for a specific data by id
Return data according to id
Return data according to id
Render page with data according to id

User go to visualize data page


User
Frontend/src/routes/visualize
Backend/server.js
weathersense_gather
ModelAPI
weathersense_hist
HTTP Request "/visualize"
HTTP Request "/history"
Query for unpredicted data
Return unpredicted data
Sent data to ModelAPI
Return data with prediction result
Query to insert data to save as history
Query for the list of previous data
Return the list of previous data
HTTP Response
Render visualize data page

Test Plan

Objective:

As our project comprises three major components, we want to make sure they function independently and integrated together.

Strategy:

We will use both manual and automated tools to test our project.

Activities:

1. Choose one of the three major components.
2. Choose the part that we want to test on its functionality.
3. Determine the objective of our test on the part.
4. Set a clear starting and ending point for each part that we want to test.

Frontend

Our frontend site is developed using Svelte on the SvelteKit framework, which is based on JavaScript.

Frontend Unit Testing:

- Unit testing in our project focuses on testing web components.
- Unit testing on functional aspects is not applicable because our website does not feature complex functionalities.
- The tool used for unit testing is **Vitest.**

Test Scenario

Test Scenario: Rendering Svelte Components with Props.
Description: This test scenario verifies that all Svelte components render correctly with their respective props.
Precondition: 1. The Svelte component is available. 2. Props are correctly defined and accessible within the component.
Step: 1. Instantiate the Svelte component. 2. Verify that each prop is correctly rendered within the component.
Expected Result: 1. The Svelte component renders without error. 2. Each prop is displayed or utilized as expected within the component.

Card component.

Test Case title:
Should render with title and value

Preconditions:
1. The Card.svelte component is available.
2. Props are correctly defined and accessible within the component.

Test step:
1. Render Card with defined props.
2. Verify Card component.

Expected result:
1. The Card component renders correctly without errors.
2. Both title and value props are displayed correctly.

Test environment: JestDOM

Actual result:
1. The Card component renders correctly without errors.
2. Props are displayed correctly.

Test Scenario: Rendering Svelte Components with Props

Status: Pass

Code:
```
describe("Card component", () => {

   beforeEach(() => {
       render(Card, { props: { title: "Test Title", value: "Test
Value" } })
   })

   it('should render with title and value', () => {
       expect(screen.getByText("Test Title")).toBeInTheDocument();
       expect(screen.getByText("Test Value")).toBeInTheDocument();
   });
});
```

Footer component.

Test Case title: Should render footer text
Preconditions: The Footer.svelte component is available.
Test step: 1. Render the Footer component. 2. Verify that the footer text is rendered correctly.
Expected result: The footer text "© 2024 Your Website. All rights reserved." is displayed within the Footer component.
Test environment: JestDOM
Actual result: Footer text rendered as expected.
Test Scenario: Rendering Svelte Components with Props
Status: **Pass**
Code:

```
describe("Footer component", () => {
   it("should render footer text", () => {
       render(Footer);
       expect(screen.getByText("© 2024 Your Website. All rights
reserved.")).toBeInTheDocument();
   });
});
```

Navbar component

Test Case title:
Should render navigation links

Preconditions:
1. The Navbar.svelte component is available.
2. Props are correctly defined and accessible within the component.

Test step:
1. Render the Navigation component with defined props.
2. Verify that the navigation links are rendered correctly.

Expected result:
The navigation links for "Home", "History", "Github", and "Contact" are displayed within the Navigation component, along with the title "WeatherSense".

Test environment: JestDOM

Actual result: The Navbar component renders correctly without errors.

Test Scenario: Rendering Svelte Components with Props

Status: **Pass**

Code:

```
describe("Navigation component", () => {
   beforeEach(() => {
       render(Navbar, {
           props: {
               link: {
                   home: "/home",
                   history: "/history",
                   github: "/github",
                   contact: "/contact"
               }
           }
       });
   });

   it('should render navigation links', () => {

expect(screen.getByText("WeatherSense")).toBeInTheDocument();
       expect(screen.getByText("Home")).toBeInTheDocument();
       expect(screen.getByText("History")).toBeInTheDocument();
       expect(screen.getByText("Github")).toBeInTheDocument();
       expect(screen.getByText("Contact")).toBeInTheDocument();
       expect(screen.getByText("Data
Visualize")).toBeInTheDocument();

   });
```

Test Case title: Should have correct href attributes
Preconditions: The Navbar.svelte component has been rendered correctly.
Test step: Verify Navbar href.
Expected result: The Navbar component has the correct href attributes for each navigation link.
Test environment: JestDOM
Actual result: The Navbar component has the correct href attributes for each navigation link.
Test Scenario: Rendering Svelte Components with Props
Status: Pass

Code:

```
it('should have correct href attributes', () => {

expect(screen.getByText("Home").closest("a")).toHaveAttribute("hr
ef", "/home");

expect(screen.getByText("History").closest("a")).toHaveAttribute(
"href", "/history");

expect(screen.getByText("Github").closest("a")).toHaveAttribute("
href", "/github");

expect(screen.getByText("Contact").closest("a")).toHaveAttribute(
"href", "/contact");

expect(screen.getByText("Data
Visualize").closest("a")).toHaveAttribute("href", "/visualize");

   });
});
```

Timestamp component

Test Case title: Should render formatted timestamp with prefix
Preconditions: 1. The Timestamp.svelte component is available. 2. Props are correctly defined and accessible within the component.
Test step: 1. Render Timestamp with defined props. 2. Verify Timestamp component.
Expected result: 1. The Timestamp component renders correctly without errors. 2. Both timestamp and prefix props are displayed correctly.
Test environment: JestDOM
Actual result: 1. The Timestamp component renders correctly without errors. 2. Props are displayed correctly.
Test Scenario: Rendering Svelte Components with Props
Status: Pass
Code:

```
describe("Timestamp component", () => {

   beforeEach(() => {
       render(Timestamp, { props: {
           timestamp: "2024-04-21T11:35:00.000Z",
               prefix: "Latest update on"

           }});
   });

   it("should render formatted timestamp with prefix", () => {
       expect(screen.getByText("Latest update on April 21, 2024
at 6:35 PM")).toBeInTheDocument();
   });
});
```

TimestampHistory Component

Test Case title:
Should render formatted timestamp with link

Preconditions:
1. The TimestampHistory.svelte component is available.
2. Props are correctly defined and accessible within the component.

Test step:
1. Render TimestampHistory with defined props.
2. Verify TimestampHistory component.

Expected result:
1. The TimestampHistory component renders correctly without errors.
2. Both timestamp props are displayed correctly.
3. The href attribute contains a correct link.

Test environment: JestDOM

Actual result:
1. The TimestampHistory component renders correctly without errors.
2. Both timestamp props are displayed correctly.
3. The href attribute contains a correct link.

Test Scenario: Rendering Svelte Components with Props

Status: Pass

Code:
```
describe("TimestampHistory component", () => {
   it("should render formatted timestamp with link", () => {
       const timestamp = "2024-04-21T11:35:00.000Z";
       const link = "/details";

       render(TimestampHistory, { props: { timestamp, link } });
       const renderedText = screen.getByText("Data on April 21,
2024 at 6:35 PM");
       const renderedLink = screen.getByRole("link", { name:
"View" });

       expect(renderedText).toBeInTheDocument();
       expect(renderedLink).toBeInTheDocument();
       expect(renderedLink).toHaveAttribute("href", "/details");
   });
});
```

Frontend Integral testing

This integral testing is created by Vitest to verify that all frontend web pages or routes can be accessed from the user.

Test Scenario

Test Scenario: HTTP Requests to Every Web Page Route.
Description: This test scenario verifies that all frontend routes can correctly request and receive responses as expected.
Precondition: The frontend server is running.
Step: 1. Send a request to each webpage route. 2. Verify the HTTP response.
Expected Result: The response status should be 200.

Test Case

Test Case title: GET http request to"/"
Preconditions: The front-end server must be running.
Test step: 1. Go to route "localhost:xxxx/" 2. Verify response status code
Expected result: status 200
Test Scenario: Request to every web page route
Test environment: HTTP
Actual result: status 200
Status: Pass

Code:
`test("should allow users go to main page", async () => {`
`  const response = await axios.get(backendRoutes.getLatest);`
`  expect(response.status).toBe(200);`
`});`

Test Case title: GET request to "/history"

Preconditions:
1. The front-end server must be running.

Test step:
Go to route “localhost:xxxx/history”
Verify response status code

Expected result: status 200

Test Scenario: Request to every web page route

Test environment: HTTP

Actual result: status 200

Status: Pass

Code:
```
test("should allow users go history page", async () => {
   const response = await axios.get(backendRoutes.getHistory);
   expect(response.status).toBe(200);
});
```

Test Case title: GET request to “/weather/id” with valid id

Preconditions:
1.The front-end server is running.
2.Back-end server is running.
3.Define valid id to test

Test step:
Go to route “localhost:xxxx/weather/id”
Verify response status code

Expected result: status 200

Test Scenario: Request to every web page route

Test environment: HTTP

Actual result: status 200

Status: Pass

Code:
```
test("should allow users go weather page (valid ID)", async () => {
   const id = 1;
   const response = await
axios.get(`${backendRoutes.getWeatherById}${id}`);
   expect(response.status).toBe(200);
});
```

Test Case title: GET request to “/weather/id” with invalid id

Preconditions:

1.The front-end server is running.
2.Back-end server is running.
3.Define valid id to test

Test step:

Go to route “localhost:xxxx/weather/id”
Verify response status code

Expected result: status 404

Test Scenario: Request to every web page route

Test environment: HTTP

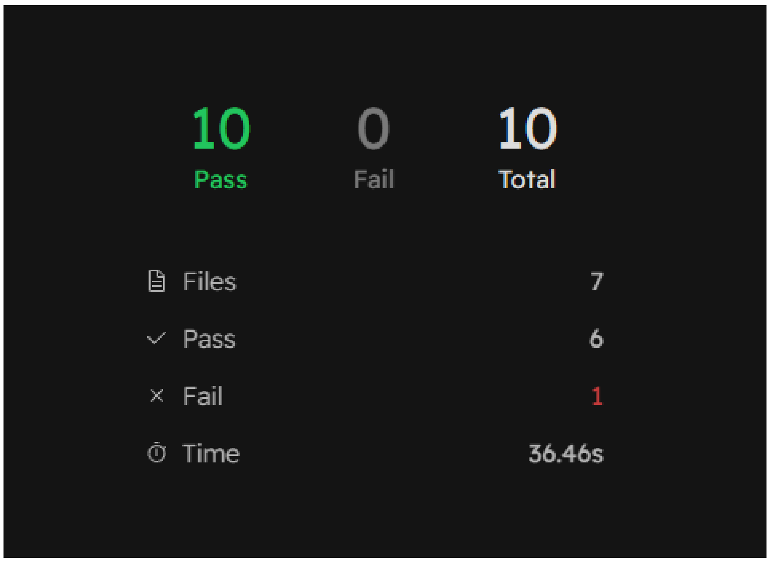
Actual result: status 404

Status: Pass

Code:

```
test("should not allow users to go to history page with invalid ID",
async () => {
   const id = -99;
   try {

       const response = await
axios.get(`${backendRoutes.getWeatherById}${id}`);
       expect(response.status).not.toBe(200);

   } catch (error) {
       expect(error.response.status).toBe(404);
   }
});
```

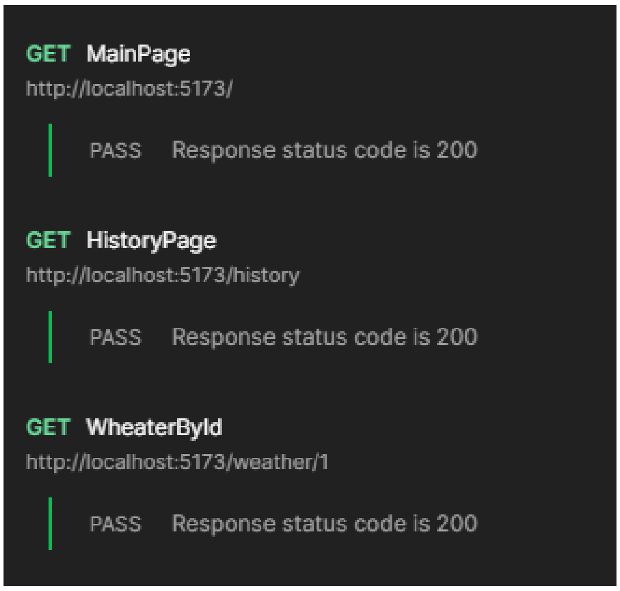
Vitest result



```
10      0      10
Pass    Fail   Total

Files              7
Pass               6
Fail               1
Time          36.46s
```

!Note that one fails because a conflict of Vitest tries to test the playwright test file.

Postman result



```
GET  MainPage
http://localhost:5173/
    PASS   Response status code is 200

GET  HistoryPage
http://localhost:5173/history
    PASS   Response status code is 200

GET  WheaterById
http://localhost:5173/weather/1
    PASS   Response status code is 200
```

Frontend E2E testing:

- **Using playwright framework**

Test Scenario

Test Scenario: Verifies that the website functions correctly and as expected from the user's perspective.
Description: This test scenario validates the expected functionality and user experience of the website.
Precondition: 1. The frontend server is running. 2. The backend server is running. 3. The ModelAPI server is running.
Step: 1. Open the website in a web browser. 2. Navigate through various pages and functionalities. 3. Verify that all interactions and functionalities perform as expected. 4. Test the website across different devices and browsers (if applicable).
Expected Result: All pages and features of the website load without errors, ensuring a seamless user experience.

Test Case title: Users can go to the main page and can click all the button navigation bars.

Preconditions:

1. All website services are running.

Test step:

1. Go to route "localhost:xxxx/"
2. Click all Navigation bar buttons.

Expected result:

1. All buttons on navigation bars can click and redirect.

Test Scenario: Verifies that the website functions correctly and as expected from the user's perspective.

Test environment: Chromium

Actual result:

1. Card component rendered correctly without errors.
2. Props displayed correctly.

Status: Pass

Code:

```
test("User can go to main page and can click all the button navigation
bar.", async ({ page }) => {
 await page.goto('http://localhost:5175/');
 await page.getByRole('link', { name: 'Home' }).click();
 await page.getByRole('link', { name: 'History' }).click();
 await page.getByRole('link', { name: 'Github' }).click();
 await page.getByRole('link', { name: 'Contact' }).click();
});
```

Test Case title: Users can go to the History page and View old weather data and go back to home.
Preconditions: 1. All website services are running.
Test step: 1. Go to route “localhost:xxxx/” 2. Click the button History on the Navigation bar. 3. Click view on some data. 4. Click the Home button on the Navigation bar.
Expected result: Users can go to the history page and view old data and can go back to home.
Test Scenario: Verifies that the website functions correctly and as expected from the user's perspective.
Test environment: Chromium
Actual result: Users can go to the history page and view old data and can go back to home.
Status: Pass
Code:

```
test("User can go to History page and View old weather data and go back
to home.", async ({ page }) => {
 await page.goto('http://localhost:5175/');
 await page.getByRole('link', { name: 'History' }).click();
 await page.locator('.inline-block').first().click();
 await page.getByRole('link', { name: 'Home' }).click();
 await page.getByRole('link', { name: 'History' }).click();
});
```

Test Case title: In the home page all data Card components are rendered.

Preconditions:

1. All website services are running.

Test step:

1. Go to route “localhost:xxxx/”.
2. Verify the page.

Expected result: All data displayed

Test Scenario: Verifies that the website functions correctly and as expected from the user's perspective.

Test environment: Chromium

Actual result: All data displayed

Status: Pass

Code:

```
test("In home page all data Card component are render.", async ({ page })
=> {
 await page.goto('http://localhost:5175/');
 await page.getByRole('navigation').click();
 await page.getByRole('heading', { name: 'WeatherSens', exact: true
}).click();
 await page.getByRole('heading', { name: 'OpenWeatherMap', exact: true
}).click();
 await page.getByRole('heading', { name: 'Temperature WeatherSense'
}).click();
 await page.getByRole('heading', { name: 'Temperature OpenWeatherMap'
}).click();
 await page.getByRole('heading', { name: 'Temperature Percentage Error'
}).click();
 await page.getByRole('heading', { name: 'Humidity WeatherSense'
}).click();
 await page.getByRole('heading', { name: 'Humidity OpenWeatherMap'
}).click();
 await page.getByRole('heading', { name: 'Humidity Percentage Error'
}).click();
 await page.getByRole('heading', { name: 'Cloudiness' }).click();
 await page.getByRole('heading', { name: 'Pressure' }).click();
 await page.getByRole('heading', { name: 'Wind Speed' }).click();
});
```

Test Case title: In the history page the View button must be loaded.

Preconditions:

1. All website services are running.

Test step:

1. Go to route “localhost:xxxx/”.
2. Click the History button.
3. Verify that the view button appears.

Expected result: In each old data view button must be rendered

Test Scenario: Verifies that the website functions correctly and as expected from the user's perspective.

Test environment: Chromium

Actual result: In each old data view button must be rendered

Status: Pass

Code:

```
test("In history page View button must loaded.", async ({ page }) => {
 await page.goto('http://localhost:5175/');
 await page.getByRole('link', { name: 'History' }).click();
 await page.getByRole('navigation').click();
 await page.locator('.inline-block').first().click();
});
```

Test Case title: In another weather data board (history) Data card must be loaded.

Preconditions:
All website services are running.

Test step:

1. Go to route “localhost:xxxx/”.
2. Click the History button.
3. Click the View button on any data.
4. Verify that all data cards must be rendered all.

Expected result: In the old weather data dashboard all data cards must be loaded.

Test Scenario: Verifies that the website functions correctly and as expected from the user's perspective.

Test environment: Chromium

Actual result: In the old weather data dashboard all data cards must be loaded.

Status: Pass

Code:

```
test("In another weather data board Data card must loaded.", async ({
page }) => {
 await page.goto('http://localhost:5175/');
 await page.getByRole('link', { name: 'History' }).click();
 await page.locator('.inline-block').first().click();
 await page.getByRole('navigation').click();
 await page.getByRole('heading', { name: 'WeatherSens', exact: true
}).click();
 await page.getByRole('heading', { name: 'OpenWeatherMap', exact: true
}).click();
 await page.getByRole('heading', { name: 'Temperature WeatherSense'
}).click();
 await page.getByRole('heading', { name: 'Temperature OpenWeatherMap'
}).click();
 await page.getByRole('heading', { name: 'Temperature Percentage Error'
}).click();
 await page.getByRole('heading', { name: 'Humidity WeatherSense'
}).click();
 await page.getByRole('heading', { name: 'Humidity OpenWeatherMap'
}).click();
 await page.getByRole('heading', { name: 'Humidity Percentage Error'
}).click();
 await page.getByRole('heading', { name: 'Cloudiness' }).click();
 await page.getByRole('heading', { name: 'Pressure' }).click();
 await page.getByRole('heading', { name: 'Wind Speed' }).click();
});
```

Test Case title: Visualize page must render all component

Preconditions:
All website services are running.

Test step:

1. Go to route “localhost:xxxx/”.
2. Click the History button.
3. Click the View button on any data.

4. Verify that all data cards must be rendered all.

Expected result: User can see all visualized data components.

Test Scenario: Verifies that the website functions correctly and as expected from the user's perspective.

Test environment: Chromium

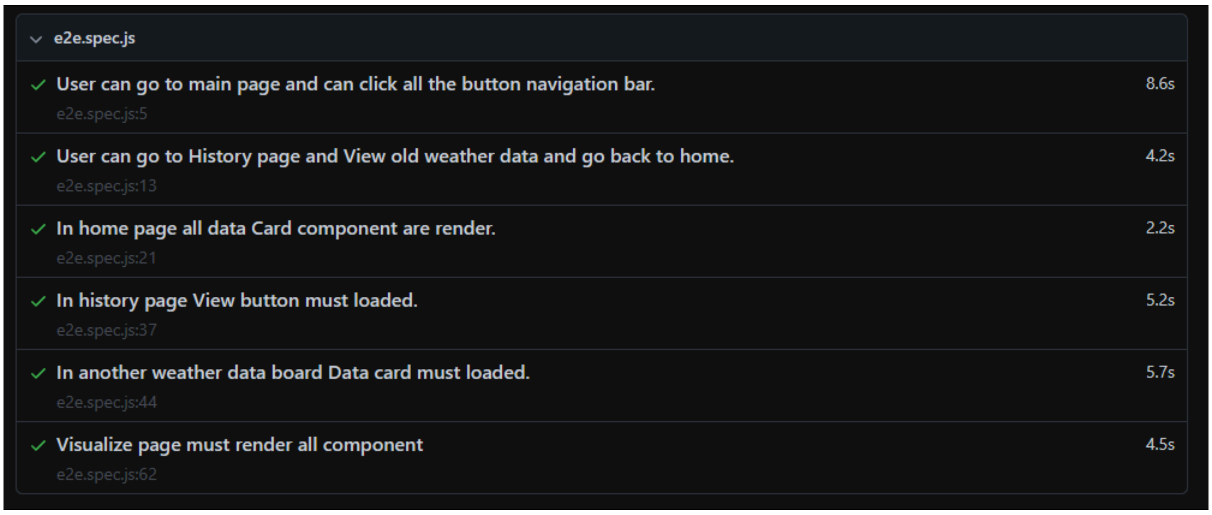
Actual result: User can see all visualized data components.

Status: Pass

Code:

```
test('Visualize page must render all component', async ({ page }) => {
 await page.goto('url');
 await page.getByRole('link', { name: 'Data Visualize' }).click();
 await page.getByRole('heading', { name: 'WeatherSense', exact: true
}).click();
 await page.getByRole('heading', { name: 'OpenWeatherAPI' }).click();
 await page.getByRole('heading', { name: 'Temperature Sensor °C'
}).click();
 await page.getByRole('heading', { name: 'Temperature API °C Histogram'
}).click();
 await page.getByRole('heading', { name: 'Cloudiness Percent Histogram'
}).click();
 await page.getByRole('heading', { name: 'Humidity Sensor Percent'
}).click();
 await page.getByRole('heading', { name: 'Humidity API Percent Histogram'
}).click();
 await page.getByRole('heading', { name: 'Wind Speed m/s Histogram'
}).click();
 await page.getByRole('heading', { name: 'Pressure HectoPascal Histogram'
}).click();
 await page.getByRole('heading', { name: 'Temperature&Humidity Scatter'
}).click();
 await page.getByRole('heading', { name: 'Day of Week Average Temperature
Sensor' }).click();
 await page.getByRole('heading', { name: 'WeatherSense Vs OpenWeather
Temperature' }).click();
 await page.getByRole('heading', { name: 'Day of Week Average Temperature
API' }).click();
 await page.getByRole('heading', { name: 'Day of Week Average Humidity
Sensor' }).click();
 await page.getByRole('heading', { name: 'WeatherSense Vs OpenWeather
Humidity' }).click();
 await page.getByRole('heading', { name: 'Day of Week Average Humidity
API' }).click();
});
```

Playwright test on chromium result:



```
e2e.spec.js
✓ User can go to main page and can click all the button navigation bar.    8.6s
  e2e.spec.js:5
✓ User can go to History page and View old weather data and go back to home.    4.2s
  e2e.spec.js:13
✓ In home page all data Card component are render.    2.2s
  e2e.spec.js:21
✓ In history page View button must loaded.    5.2s
  e2e.spec.js:37
✓ In another weather data board Data card must loaded.    5.7s
  e2e.spec.js:44
✓ Visualize page must render all component    4.5s
  e2e.spec.js:62
```

Backend API Testing

In our backend, we have two major servers:

- Node.js server (Express): This is the main backend server for our website.
- Model API server (Flask): This server runs to predict when someone sends a POST request.

Both servers are running on localhost:

- Node.js server runs on port localhost:3000.
- Model API server runs on port localhost:5000.

Node.js:

Test scenario

Test Scenario: Backend API Endpoint Validation
Description: This test scenario verifies the functionality of all backend API endpoints by ensuring they return JSON responses correctly.
Precondition: **1.** The backend server is running.
Step: **1.** Send an HTTP request to the expected endpoint. **2.** Verify the response.
Expected Result: The response should return as expected without errors.

- **/host/latest: (GET)** This endpoint receives API requests from those who want the latest weather data from our database. In our project, the first page of the website requests
this to display the latest weather data.

get host/latest output (json)

```
[
    {
        "id": 270,
        "ts": "2024-04-21T17:45:10.000Z",
        "temp_sensor": 33,
        "humidity_sensor": 70,
        "temp_api": 31.19,
        "humidity_api": 74,
        "pressure": 1006,
        "wind_speed": 5.14,
        "cloudiness": 20,
        "weather": "Clouds",
        "weather_pred": "Clouds"
    }
]
```

Test Case title: GET HTTP Request to “/latest”

Preconditions:
1. Back-end Server is running

Test step:
1. HTTP GET request to “localhost:3000/”.
2. Verify the response.

Expected result: Response status and json are the latest weather data.

Test Scenario: HTTP Request to Back-end Server

Test environment: HTTP

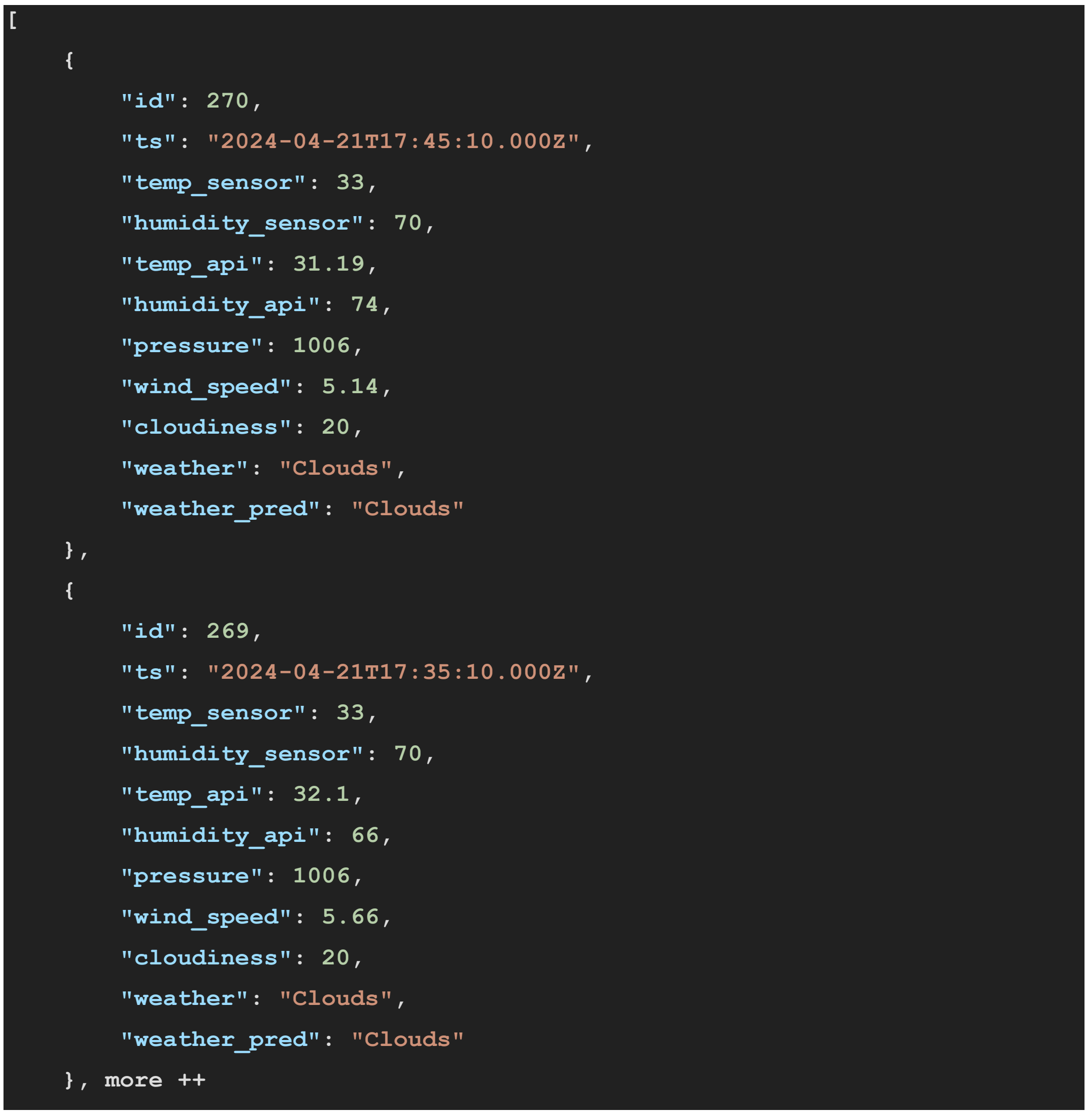
Actual result: Response status and json are the latest weather data.

Status: Pass

Code (Postman):
```
pm.test("Response status code is 200", function () {
    pm.expect(pm.response.code).to.equal(200);
});

pm.test("Verify that the 'id' is a non-negative integer", function () {
    const responseData = pm.response.json();

    pm.expect(responseData).to.be.an('array');
    pm.expect(responseData).to.have.lengthOf.at.least(1);

    responseData.forEach(function (data) {
        pm.expect(data.id).to.be.a('number');
        pm.expect(data.id).to.be.at.least(0);
    });
});

pm.test("Verify that 'ts' is a non-empty string", function () {
    const responseData = pm.response.json();

    pm.expect(responseData).to.be.an('array');

    responseData.forEach(function (data) {

pm.expect(data.ts).to.be.a('string').and.to.have.lengthOf.at.least(1,
"'ts' should be a non-empty string");
    });
});

pm.test("Verify that the 'temp_sensor' is a non-negative number",
function () {
    const responseData = pm.response.json();

    pm.expect(responseData).to.be.an('array');
    responseData.forEach(function(data) {
        pm.expect(data.temp_sensor).to.be.a('number');
        pm.expect(data.temp_sensor).to.be.at.least(0);
    });
```

- **/host/history: (GET)** This endpoint receives API requests from those who want all weather data from our database, ordered from latest to oldest. In our project, the history page of the website requests this to display all historical weather data.

get host/history output (json)



```
[
    {
        "id": 270,
        "ts": "2024-04-21T17:45:10.000Z",
        "temp_sensor": 33,
        "humidity_sensor": 70,
        "temp_api": 31.19,
        "humidity_api": 74,
        "pressure": 1006,
        "wind_speed": 5.14,
        "cloudiness": 20,
        "weather": "Clouds",
        "weather_pred": "Clouds"
    },
    {
        "id": 269,
        "ts": "2024-04-21T17:35:10.000Z",
        "temp_sensor": 33,
        "humidity_sensor": 70,
        "temp_api": 32.1,
        "humidity_api": 66,
        "pressure": 1006,
        "wind_speed": 5.66,
        "cloudiness": 20,
        "weather": "Clouds",
        "weather_pred": "Clouds"
    }, more ++
```

Test Case title: GET HTTP Request to “/history”
Preconditions: 1. Back-end Server is running
Test step: 1. HTTP GET request to “localhost:3000/history”. 2. Verify the response.
Expected result: Response status and json are all the weather data ordered by DESC.
Test Scenario: HTTP Request to Back-end Server
Test environment: HTTP
Actual result: Response status and json are all the weather data ordered by DESC.
Status: Pass
Code (Postman):

```
pm.test("Response status code is 200", function () {
    pm.expect(pm.response.code).to.equal(200);
});

pm.test("Response is an array with at least one element", function () {
  const responseData = pm.response.json();

  pm.expect(responseData).to.be.an('array');
  pm.expect(responseData).to.have.lengthOf.at.least(1);
});



pm.test("Verify that the 'id' field is a non-negative integer", function
() {
    const responseData = pm.response.json();

    pm.expect(responseData).to.be.an('array');
    responseData.forEach(function (data) {

pm.expect(data.id).to.exist.and.to.be.a('number').and.to.satisfy(function
(id) {
            return id >= 0;
        });
    });
});
```

- **/host/weather/:id: (GET)** This endpoint receives API requests from those who want the weather data by its ID from our database. In our project, the /weather/id endpoint requests this to show specific data.

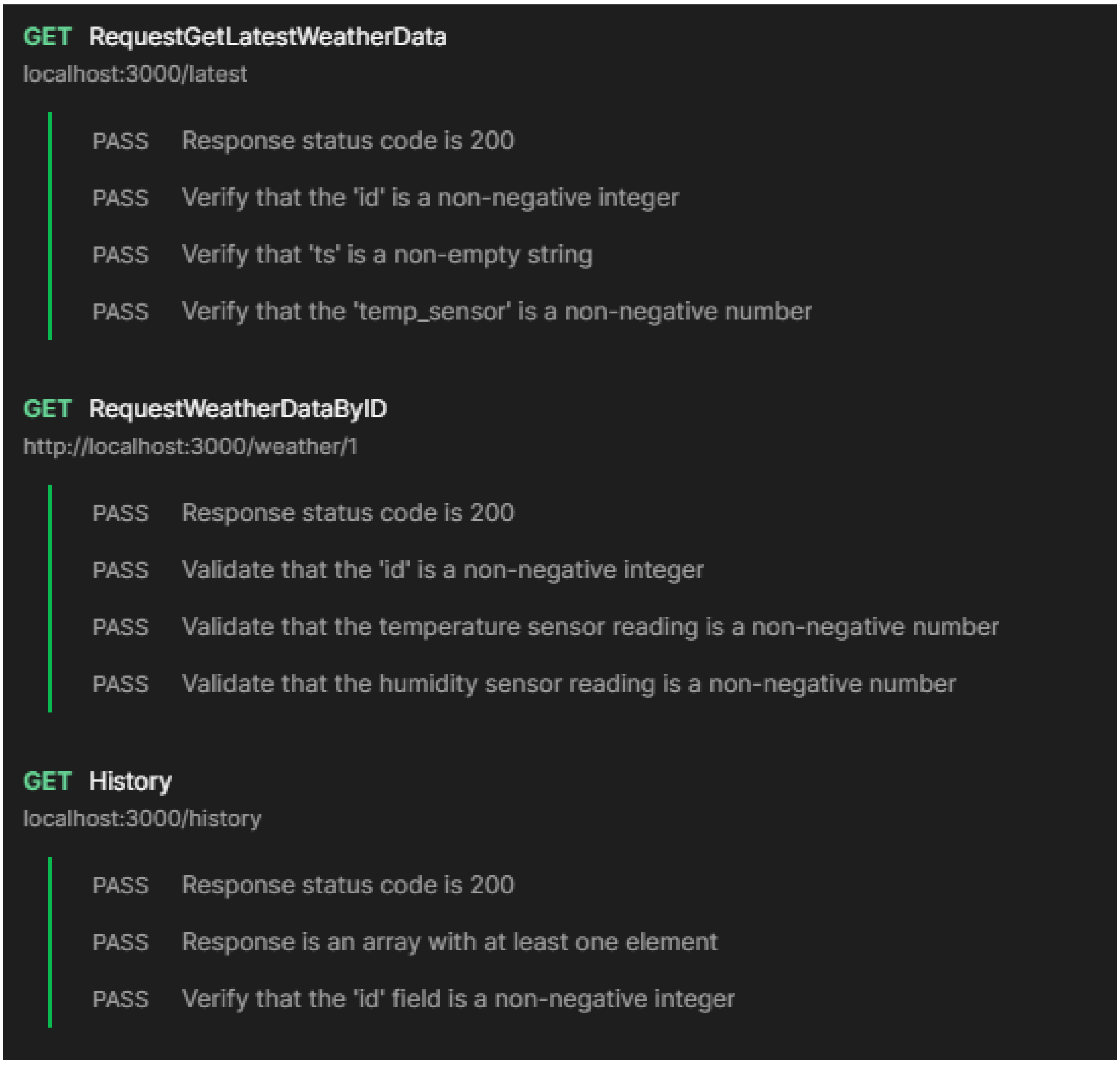
get host/weather/1 output (json)

```
{
    "id": 1,
    "ts": "2024-04-18T10:04:21.000Z",
    "temp_sensor": 20,
    "humidity_sensor": 50,
    "temp_api": 36.88,
    "humidity_api": 57,
    "pressure": 1004,
    "wind_speed": 5.66,
    "cloudiness": 20,
    "weather": "Clouds",
    "weather_pred": "Clouds"
}
```

Test Case title: HTTP GET Request to "/weather/id"
Preconditions: 1. Back-end Server is running 2. Defined id
Test step: 1. HTTP GET request to "localhost:3000/weather/id". 2. Verify the response.
Expected result: Response status and json are the weather data specific id.
Test Scenario: HTTP Request to Back-end Server
Test environment: HTTP
Actual result: Response status and json are the weather data specific id.
Status: **Pass**
Code (Postman):

```
pm.test("Response status code is 200", function () {
  pm.response.to.have.status(200);
});
pm.test("Validate that the 'id' is a non-negative integer", function () {
    const responseData = pm.response.json();
    pm.expect(responseData.id).to.exist;
    pm.expect(responseData.id).to.be.a('number');
    pm.expect(responseData.id).to.be.greaterThan(-1);
});

pm.test("Validate that the temperature sensor reading is a non-negative
number", function () {
  const responseData = pm.response.json();


pm.expect(responseData.temp_sensor).to.be.a('number').and.to.be.at.least(
0);
});

pm.test("Validate that the humidity sensor reading is a non-negative
number", function () {
  const responseData = pm.response.json();


pm.expect(responseData.humidity_sensor).to.be.a('number').and.to.be.at.le
ast(0);
});
```

We will test with Postman, the test result is



```
GET  RequestGetLatestWeatherData
localhost:3000/latest

     PASS    Response status code is 200
     PASS    Verify that the 'id' is a non-negative integer
     PASS    Verify that 'ts' is a non-empty string
     PASS    Verify that the 'temp_sensor' is a non-negative number

GET  RequestWeatherDataByID
http://localhost:3000/weather/1

     PASS    Response status code is 200
     PASS    Validate that the 'id' is a non-negative integer
     PASS    Validate that the temperature sensor reading is a non-negative number
     PASS    Validate that the humidity sensor reading is a non-negative number

GET  History
localhost:3000/history

     PASS    Response status code is 200
     PASS    Response is an array with at least one element
     PASS    Verify that the 'id' field is a non-negative integer
```

Model API server (Flask)

Test scenario

Test Scenario: ModelAPI-Server Endpoint Validation
Description: This test scenario verifies that the ModelAPI-Server endpoint works correctly and returns JSON data as expected.
Precondition: **1.** The ModelAPI-Server is running. **2.** Data for model prediction is defined.
Step: 1. Send an HTTP request to the expected endpoint. 2. Verify the response.
Expected Result: The response should return as expected without errors.

- **host/predict: (POST)** This endpoint receives API requests from those who want to predict data from our model by sending data in json and receiving responses.In our project, NodeJs server sent a data to Model

Body

```
[
    {
        "id": 1,
        "ts": "2024-04-14 21:51:13",
        "temp_sensor": 25,
        "humidity_sensor": 50,
        "temp_api": 31.03,
        "humidity_api": 73,
        "pressure": 1010,
        "wind_speed": 5.66,
        "cloundiness": 20,
        "weather": "few clouds"
    }
]
```

Response

```
[
    {
        "id": 1,
        "ts": "2024-04-14 21:51:13",
        "temp_sensor": 25,
        "humidity_sensor": 50,
        "temp_api": 31.03,
        "humidity_api": 73,
        "pressure": 1010,
        "wind_speed": 5.66,
        "cloundiness": 20,
        "weather": "few clouds",
        "weather_pred": "Clouds"
    }
]
```

Test Case title: HTTP POST Request to “/predict”

Preconditions:
1. ModelAPI-Server is running
2. Defined data to send to predict a model.

Test step:
1. HTTP GET request to “localhost:5000/predict”.
2. Verify the response.

Expected result: Response status and json are the same data but have new attribute name “weather_pred” and its value.

Test Scenario: HTTP Request to ModelAPI-Server

Test environment: HTTP

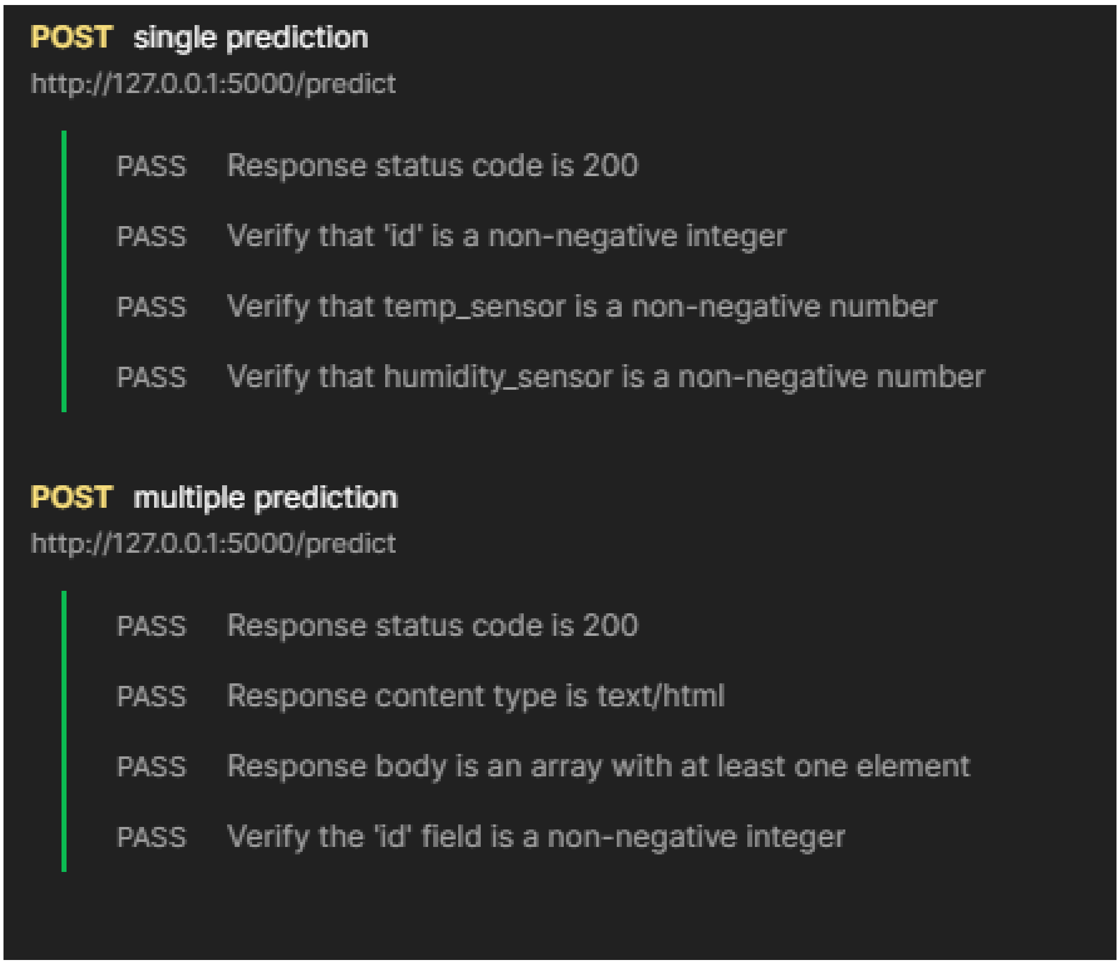
Actual result: Response status and json are the same data but have new attribute name “weather_pred” and its value.

Status: Pass

Code (Postman):
```
pm.test("Response status code is 200", function () {
  pm.response.to.have.status(200);
});
```

```
pm.test("Response content type is text/html", function () {

pm.expect(pm.response.headers.get("Content-Type")).to.include("text/html"
);
});


pm.test("Response body is an array with at least one element", function
() {
    const responseData = pm.response.json();

    pm.expect(responseData).to.be.an('array').that.is.not.empty;
});


pm.test("Verify the 'id' field is a non-negative integer", function () {
    const responseData = pm.response.json();

    pm.expect(responseData).to.be.an('array');

    responseData.forEach(function (data) {
        pm.expect(data.id).to.be.a('number').and.to.be.at.least(0);
    });
});
```

Test with Postman, the test result is



```
POST  single prediction
http://127.0.0.1:5000/predict

    PASS    Response status code is 200
    PASS    Verify that 'id' is a non-negative integer
    PASS    Verify that temp_sensor is a non-negative number
    PASS    Verify that humidity_sensor is a non-negative number

POST  multiple prediction
http://127.0.0.1:5000/predict

    PASS    Response status code is 200
    PASS    Response content type is text/html
    PASS    Response body is an array with at least one element
    PASS    Verify the 'id' field is a non-negative integer
```