INTERNATIONAL UNIVERSITY

School of Computer Science & Engineering

WEB APPLICATION DEVELOPMENT

LABORATORY

Lab 04

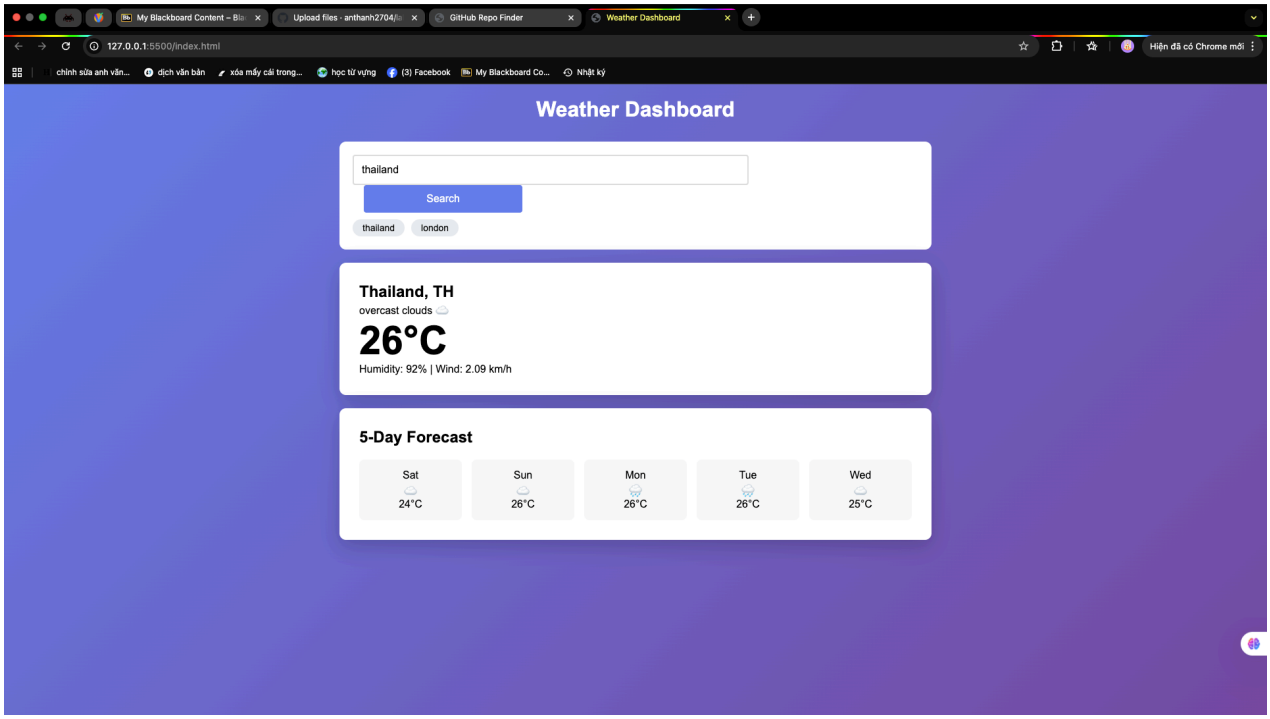
JavaScript Fundamentals

Submitted by:

Nguyễn Hà An Thạnh – ITITWE22051

Date submits:	7/11/2025
Date performs:	7/11/2025
Lab section:	Lab 4
Course instructor:	Msc. N. T. Nghĩa

Task 2.1 - Weather Dashboard



+Overview of How the App Works

When you open the page:

1. The browser loads your HTML, CSS, and JavaScript.
2. The JavaScript initializes — it **loads saved searches** from `localStorage` and waits.
3. When you type a city (like "London") and click **Search**, it triggers:
    - `searchWeather()`
    - which then calls:
        - `fetchWeather(city)` → gets **current weather**
        - `fetchForecast(city)` → gets **5-day forecast**
    - and finally updates your screen with the results.

So the app is a cycle of:

Input → Fetch Data from API → Display Data → Save Search

1. async fetchWeather(city) - Fetch current weather

```
async function fetchWeather(city) {
    try {
        const response = await fetch(
```

```
`https://api.openweathermap.org/data/2.5/weather?q=${city}&appid=${API_KEY}&units=metric`
        );
        if (!response.ok) throw new Error("City not found");
        const data = await response.json();
        displayWeather(data);
    } catch (error) {
        showError(error.message);
    }
}
```

How it works:

- When you enter a city, say `London`, the function creates a **URL** like:
- `https://api.openweathermap.org/data/2.5/weather?q=London&appid=YOUR_KEY&units=metric`

  This URL asks OpenWeather for the current temperature, humidity, and other data for London.

- `fetch()` sends a request to that URL and waits for a response.
- Because the function is `async`, it can use `await` — meaning the code "pauses" until the response returns.
- Once the data arrives, it's converted from raw text to **JSON** (a JavaScript-friendly object).
- That object contains keys like:
- `{`
- `  "name": "London",`
- `  "main": { "temp": 15, "humidity": 63 },`
- `  "weather": [{ "main": "Clouds", "description": "broken clouds" }],`
- `  "wind": { "speed": 4.5 }`
- `}`
- The function then passes this object to `displayWeather(data)` so you can visually show it on the page.
- If anything goes wrong (city not found, wrong key, no internet), the `catch` block catches the error and shows it in red using `showError()`.

2. async fetchForecast(city) - Fetch 5-day forecast

```
async function fetchForecast(city) {
    try {
        const response = await fetch(
```

```
`https://api.openweathermap.org/data/2.5/forecast?q=${city}&appid=${
API_KEY}&units=metric`
        );
        if (!response.ok) throw new Error("Forecast not found");
        const data = await response.json();
        displayForecast(data);
    } catch (error) {
        showError(error.message);
    }
}
```

Purpose: Get the **next 5 days** of weather for the chosen city.

How it works:

- Very similar to `fetchWeather()`, but it calls the **forecast** API endpoint instead:
- `https://api.openweathermap.org/data/2.5/forecast?q=London&appid=YOUR_KEY&units=metric`
- The forecast data is huge — OpenWeather sends around **40 data points**, one every 3 hours.
- Example:
- `{`
- `  "list": [`
- `    { "dt_txt": "2025-11-07 09:00:00", "main": { "temp": 15 }, "weather": [{ "main": "Clouds" }] },`
- `    { "dt_txt": "2025-11-07 12:00:00", "main": { "temp": 17 }, "weather": [{ "main": "Clear" }] },`
- `    ...`
- `  ]`
- `}`
- We don't want all 40 entries (too much).
  The next function, `displayForecast(data)`, filters out only **one time per day** (usually around noon) — so you see 5 boxes, one per day.
- If the forecast isn't found, an error appears.

3. displayWeather(data) - Display current weather

```
function displayWeather(data) {
    const weatherDisplay =
document.getElementById("weatherDisplay");
    const icon = getWeatherIcon(data.weather[0].main);
    const html = `
        <div class="weather-card">
            <div class="current-weather">
```

```
                <div>
                    <h2>${data.name}, ${data.sys.country}</h2>
                    <p>${data.weather[0].description} ${icon}</p>
                    <p class="temp-
display">${Math.round(data.main.temp)}°C</p>
                    <p>Humidity: ${data.main.humidity}% | Wind:
${data.wind.speed} km/h</p>
                </div>
            </div>
        </div>
    `;
    weatherDisplay.innerHTML = html;
}
```

How it works:

- It extracts pieces of information from the `data` object:
  - City name (`data.name`)
  - Country code (`data.sys.country`)
  - Temperature (`data.main.temp`)
  - Description (`data.weather[0].description`)
  - Humidity and wind speed
- It then builds an HTML layout (a card) with those details, e.g.:
- `London, GB`
- `broken clouds` ☁️
- `15°C`
- `Humidity: 63% | Wind: 4.5 km/h`
- It uses the helper `getWeatherIcon()` to convert the weather type (like `"Rain"`) into an emoji (🌧️).
- Finally, it injects that HTML into your `<div id="weatherDisplay">`.

4. displayForecast(data) - Display forecast

```
function displayForecast(data) {
    const forecastDisplay =
document.getElementById("forecastDisplay");
    const forecastList = data.list.filter(item =>
item.dt_txt.includes("12:00:00"));

    let html = `
        <div class="weather-card">
            <h2>5-Day Forecast</h2>
            <div class="forecast-grid">
```

```
    `;

    forecastList.forEach(item => {
        const date = new Date(item.dt_txt);
        const day = date.toLocaleDateString("en-US", { weekday:
"short" });
        const icon = getWeatherIcon(item.weather[0].main);
        html += `
            <div class="forecast-item">
                <p>${day}</p>
                <p>${icon}</p>
                <p>${Math.round(item.main.temp)}°C</p>
            </div>
        `;
    });

    html += `</div></div>`;
    forecastDisplay.innerHTML = html;
}
```

How it works:

- The forecast data from OpenWeather contains many time slots per day (every 3 hours).
  This function picks **just one per day** — usually the one at **12:00:00** (noon), so that each day's weather is represented once.
- It loops through those daily entries and extracts:
  - Date (to display the day name like "Mon", "Tue")
  - Temperature
  - Weather icon (☀️, 🌧️, ☁️)
- Then, it creates a row of 5 small "cards", like this:
- Mon ☀️ 25°C
- Tue ☁️ 22°C
- Wed 🌧️ 19°C
- Thu 🌧️ 18°C
- Fri ☀️ 27°C
- Finally, it inserts that grid into `<div id="forecastDisplay">`.

5. saveRecentSearch(city) - Save to localStorage

```
function saveRecentSearch(city) {
```

```
    let cities = JSON.parse(localStorage.getItem("recentCities")) ||
[];
    if (!cities.includes(city)) {
        cities.unshift(city);
        if (cities.length > 5) cities.pop();
        localStorage.setItem("recentCities",
JSON.stringify(cities));
    }
}
```

How it works:

- Uses **localStorage**, a small built-in browser database that can save data as key/value pairs.
- It reads any saved cities:
- `localStorage.getItem("recentCities")`

  → returns something like `["London", "Tokyo", "Paris"]`

- Then it:
  - Adds the new city to the front of the list (`unshift`).
  - Removes duplicates (so you don't get London twice).
  - Keeps only the 5 most recent.
- Finally, it saves the updated list back:
- `localStorage.setItem("recentCities", JSON.stringify(cities))`
- This means when you close and reopen the browser, your last searches are still there!

6. loadRecentSearches() - Load from localStorage

```
function loadRecentSearches() {
    const recentDiv = document.getElementById("recentSearches");
    const cities = JSON.parse(localStorage.getItem("recentCities"))
|| [];
    recentDiv.innerHTML = cities.map(city =>
        `<div class="recent-city"
onclick="searchCity('${city}')">${city}</div>`
    ).join("");
}
```

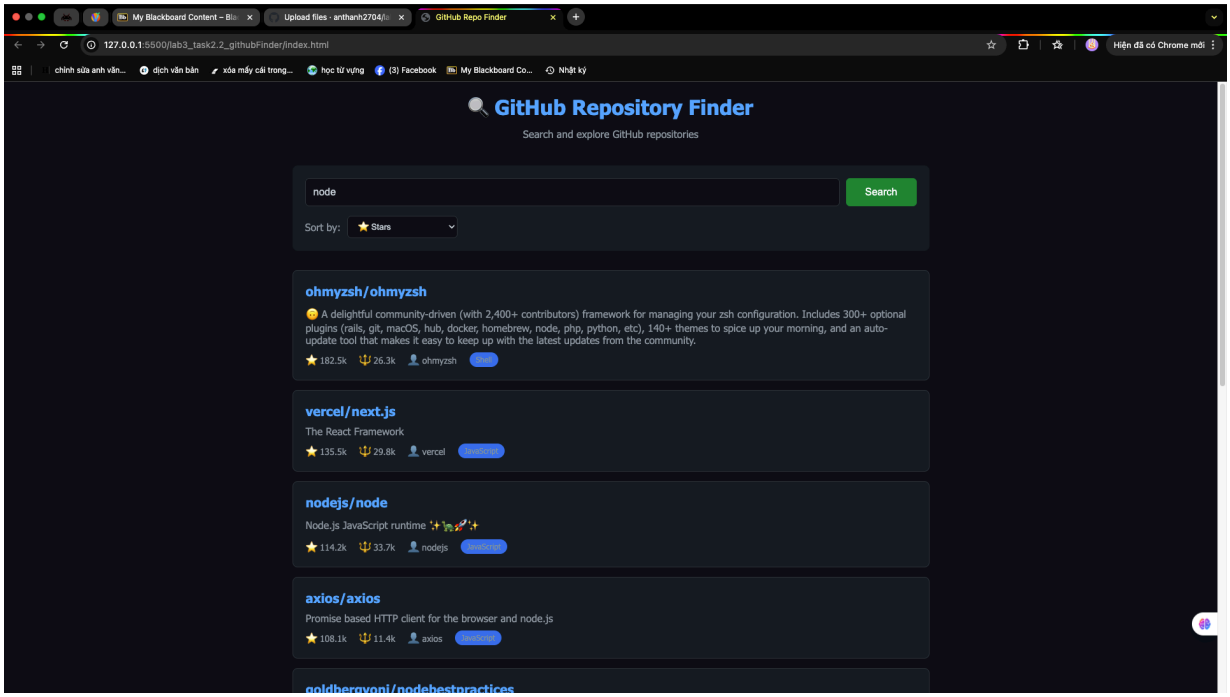
How it works:

- When the app first loads (or after a search), it reads from `localStorage`.
- It creates small buttons or “chips” for each recent city:
- `[London] [Tokyo] [Paris]`
- Each one has an `onclick` event that calls:
- `searchCity("London");`

  Which in turn sets the input box and re-runs the search.

- This gives you quick access to your most common searches.

Task 2.2 - GitHub Finder



7. async searchRepositories(query, sort, page) - Search repos

```
async function searchRepositories(query, sort = 'stars', page = 1) {
  clearError();

  const q = encodeURIComponent((query || '').trim());
  const url =
`https://api.github.com/search/repositories?q=${q}&sort=${sort}&page
=${page}&per_page=${PER_PAGE}`;

  try {
    const response = await fetch(url);

    // Rate limit (unauthenticated ~60 req/hour)
```

```
    if (response.status === 403) {
      showError("⚠️ GitHub rate limit exceeded! Please try again
later.");
      return null;
    }

    if (!response.ok) {
      showError("❌ Failed to fetch data from GitHub.");
      return null;
    }

    const data = await response.json();
    totalResults = data.total_count || 0; // used for pagination
    return Array.isArray(data.items) ? data.items : [];
  } catch (error) {
    showError("❌ Network error. Please check your internet
connection.");
    return null;
  }
}
```

How it works

- **Receives** the user's keyword (`query`), chosen sort (`stars`/`forks`/`updated`), and which page to fetch.
- **Cleans & encodes** the query so it's safe in a URL.
- **Builds** the GitHub Search API URL with `q`, `sort`, `page`, and `per_page=10`.
- **Fetches** data from GitHub.
- **Handles errors**:
  - If HTTP **403** → rate limit hit → shows a friendly message and returns `null`.
  - If any other non-OK status or network failure → shows a generic error and returns `null`.
- **Parses** the JSON when successful.
- **Updates global state**: saves `total_count` (used to compute total pages).
- **Returns** the `items` array (can be empty) so the UI can render results.

Side effects: sets `totalResults` globally.
Returns: an array of repos (or `null` on error).

8. displayRepositories(repos) - Display results

```
function displayRepositories(repos) {
```

```
  const repoList = document.getElementById('repoList');
  const loadMoreContainer =
document.getElementById('loadMoreContainer');

  // Clear previous results
  repoList.innerHTML = '';

  // If null, an error was already shown
  if (repos === null) {
    loadMoreContainer.innerHTML = '';
    return;
  }

  // Empty-state
  if (!repos || repos.length === 0) {
    repoList.innerHTML = '<p class="loading">No repositories
found.</p>';
    loadMoreContainer.innerHTML = '';
    return;
  }

  // Render each card
  repos.forEach(repo => {
    repoList.appendChild(createRepoCard(repo));
  });

  // Decide whether to show the Load More button
  const maxTotal = Math.min(totalResults, 1000); // Search API
visible cap
  const totalPages = Math.ceil(maxTotal / PER_PAGE);

  if (currentPage < totalPages && repos.length === PER_PAGE) {
    loadMoreContainer.innerHTML = `
      <div class="load-more">
        <button onclick="loadMore()">Load More Results</button>
      </div>
    `;
  } else {
    loadMoreContainer.innerHTML = '';
  }
}
```

How it works

- **Clears** any old results from the page.
- If `repos` is **`null`** (meaning fetch failed), it **stops**—the error banner is already visible.
- If `repos` is an **empty array**, shows **"No repositories found."**
- Otherwise, **loops** through repos and **renders** a card for each repo.
- **Computes pagination** using `totalResults` (capped at 1,000 by GitHub's Search API) and `PER_PAGE`.
- **Shows/Hides** the **Load More** button:
  - Shows it if there are more pages and this page returned a full batch (10).
  - Hides it otherwise.

Side effects: updates the DOM; toggles the Load More button.
Input contract: expects an array or `null`.

9. appendRepositories(repos) - Add more results

```
function appendRepositories(repos) {
  const repoList = document.getElementById('repoList');
  const loadMoreContainer =
document.getElementById('loadMoreContainer');

  if (repos === null) {
    // Error already shown
    return;
  }

  if (!repos || repos.length === 0) {
    // No more results to append
    loadMoreContainer.innerHTML = '';
    return;
  }

  // Append cards
  repos.forEach(repo => {
    repoList.appendChild(createRepoCard(repo));
  });

  // Re-evaluate Load More visibility
  const maxTotal = Math.min(totalResults, 1000);
  const totalPages = Math.ceil(maxTotal / PER_PAGE);
```

```
  if (currentPage < totalPages && repos.length === PER_PAGE) {
    loadMoreContainer.innerHTML = `
      <div class="load-more">
        <button onclick="loadMore()">Load More Results</button>
      </div>
    `;
  } else {
    loadMoreContainer.innerHTML = '';
  }
}
```

How it works

- Used when loading the **next page**; it **does not** clear existing results.
- If `repos` is **`null`**, **does nothing** (error already shown).
- If `repos` is an **empty array**, **hides** the Load More button (no more data).
- Otherwise, **adds** each repo card to the end of the current list.
- **Re-evaluates pagination**:
  - If more pages remain and this batch size is 10, keep the button.
  - Otherwise, hide the button (last page reached).

Side effects: appends to the DOM; may hide/show the Load More button.
Input contract: expects an array or `null`.

10. getSortValue() - Get selected sort option

```
function getSortValue() {
  const el = document.getElementById('sortSelect');
  const value = el && el.value ? el.value : 'stars';
  return ['stars', 'forks', 'updated'].includes(value) ? value :
'stars';
}
```

How it works

- **Reads** the current value from the sort `<select>` in the DOM.
- **Validates** it against allowed options (`stars`, `forks`, `updated`).
- **Fallbacks** to `stars` if anything's missing or invalid.

Side effects: none.
Returns: a safe string (`stars`/`forks`/`updated`).

11. loadMore() - Load next page of results

```
async function loadMore() {
```

```
  const sort = getSortValue();
  const nextPage = currentPage + 1;

  const repos = await searchRepositories(currentQuery, sort,
nextPage);
  if (repos === null) {
    // Keep currentPage unchanged on failure
    return;
  }

  currentPage = nextPage;
  appendRepositories(repos);
}
```

How it works

- **Calculates** the next page number (`currentPage + 1`).
- **Reads** the current sort via `getSortValue()`.
- **Calls** `searchRepositories(currentQuery, sort, nextPage)` to fetch the next batch.
- If the fetch **fails** (`null`), it **does not** advance the page (state stays consistent).
- If successful:
    - **Updates** `currentPage` to the next page.
    - **Calls** `appendRepositories(repos)` to add the new items below the existing ones.
- **Relies on** `appendRepositories` to decide whether the Load More button remains visible.

Side effects: advances `currentPage` only on success; updates the DOM via `appendRepositories`.
Dependencies: needs `currentQuery` already set by the initial search.