Design Pattern and Microservices

Assignment-1

Name- **Paras Gupta**

System id- **2023356731**

Section-**I (G2)**

Semester-**6th**

Course-**Design Pattern and Microservices(CSCR3216)**

PART 1 — The Core Engine (ES6+ Fundamentals):

Keyword	Scope	Reassign?	Redeclare?	Hoisted?
var	Function-scoped	Yes	Yes	Yes (undefined)
let	Block-scoped	Yes	No	Yes (TDZ)
const	Block-scoped	No	No	Yes (TDZ)

Bad Way (Old JS with var):

```
function example() {
   if (true) {
      var x = 10;
   }
   console.log(x); // 10 (leaks outside block)
}
example();
```

Problem:

1. var ignores block scope.
2. Can accidentally overwrite variables.
3. Causes bugs in loops & closures.

Best Way (Modern ES6+):

```
function example() {
   if (true) {
      let x = 10;
```

```
    }
    console.log(x); // ❌ ReferenceError
}
```

What is Hoisting?

Hoisting = JavaScript moves declarations to the top during compilation.

With var

```
console.log(a); // undefined var a = 5;
```

Equivalent to:

```
var a;console.log(a);
a = 5;
```

With let and const:

```
console.log(a); // Reference Error
let a = 10;
```

They are hoisted but cannot be accessed before declaration.

2. Modern Functions — Traditional vs Arrow Functions:

Bad Way (Traditional Function)

```
function add(a, b) {
    return a + b;
}
```

Pro Way (Arrow Function)

```
const add = (a, b) => a + b;
```

Benefits:
Shorter syntax

Implicit return

Cleaner and easier to read

Important difference: Arrow functions do not have their own this, which is useful in modern frameworks.

3. Array Mastery — map(), filter(), reduce()

Bad Way (for loop)

```
let arr = [1,2,3,4];
let result = [];

for (let i = 0; i < arr.length; i++) {
   result.push(arr[i] * 2);
}
```

Pro Way — map() (Transform data)

```
let result = arr.map(num => num * 2);
```

Pro Way — filter() (Select data)

```
let even = arr.filter(num => num % 2 === 0);
```

Pro Way — reduce() (Combine data)

```
let sum = arr.reduce((total, num) => total + num, 0);
```

These methods are cleaner and more readable than loops.

4. Objects & Destructuring + Spread/Rest Operator

Bad Way

```
let user = {
   name: "Shirin",
   age: 20,
   city: "Delhi"
};

let name = user.name;
let age = user.age;
```

Pro Way — Destructuring

```
const { name, age } = user;
```

Spread Operator (...)

Used to copy or merge arrays/objects.

```
let arr2 = [...arr1, 3, 4];
let user2 = { ...user, country: "India" };
```

Rest Operator

```
const [first, ...rest] = [10,20,30,40];
```

Part 2: The Interface (DOM Manipulation & Storage)

JavaScript becomes powerful when it starts interacting with HTML. This is called DOM Manipulation. DOM (Document Object Model) allows JavaScript to change content, style, and behavior of a web page dynamically.

1. Selection & Modification — querySelector vs getElementById
To change anything on a webpage, we first need to select the element.
Using getElementById

```
let title = document.getElementById("heading");
title.textContent = "Welcome to My Website";
```

This works only with IDs.
Using querySelector (Pro Way)

```
let title = document.querySelector("#heading");
title.style.color = "blue";
```

Why querySelector is better:
Can select by id, class, tag

More flexible and modern

Used in most projects

2. Event Handling — Making the page interactive
Events allow JavaScript to respond to user actions like clicks and form submissions.
Button Click Example

```
let btn = document.querySelector("#btn");

btn.addEventListener("click", () => {
   alert("Button Clicked!");
});
```

Form Submit Example

```
let form = document.querySelector("#form");

form.addEventListener("submit", (e) => {
   e.preventDefault();
   alert("Form Submitted");
});
```

e.preventDefault() stops the page from refreshing.

3. Event Bubbling & Event Delegation (Important Concept)
Event Bubbling
When you click a child element, the event moves upward to parent elements.
Real-life analogy:
If a child shouts, the parent and grandparent can also hear it.

Event Delegation (Pro Technique)
Instead of adding events to many child elements, we add one event to the parent.

```
document.querySelector("#list").addEventListener("click", (e) => {
   if (e.target.tagName === "LI") {
      console.log("Item clicked:", e.target.textContent);
   }
});
```

Benefits:
Better performance

Less code

Works for dynamically added elements

4. Persistence using Local Storage:

Normally, data is lost when the page refreshes. LocalStorage allows us to save data in the browser.
Saving Data

```
localStorage.setItem("theme", "dark");
```

Getting Data

```
let theme = localStorage.getItem("theme");
```

5. Why JSON.stringify() and JSON.parse()?

LocalStorage only stores strings. So when we want to store objects or arrays, we convert them.
Store Object

```
let user = { name: "Shirin", age: 20 };
localStorage.setItem("user", JSON.stringify(user));
```

Retrieve Object

```
let userData = JSON.parse(localStorage.getItem("user"));
console.log(userData.name);
```

Part 3: The Data Flow (Asynchronous JavaScript)

Understanding Asynchronous JavaScript
When your browser runs JavaScript, it executes line by line. But what happens when you call an API or wait for a timer? We don't want the entire page to freeze while waiting for a response. This is where asynchronous JavaScript comes in.
The Event Loop: A Simple Analogy
Think of JavaScript as a single chef in a kitchen. The chef can only cook one dish at a time.
If a dish takes a long time (like boiling pasta), the chef hands it to an assistant (async) and moves on to the next dish.

Once the dish is ready, the assistant calls the chef back.

In JavaScript:
Call Stack: Chef's active tasks.

Web APIs: Assistant handles async tasks.

Callback Queue: Completed tasks waiting for the chef.

Event Loop: Makes sure the chef picks the next ready task.

Callback Hell

```
// Bad Way: Nested callbacks
getUser(1, function(user){
  getOrders(user.id, function(orders){
    getShipping(orders[0].id, function(shipping){
      console.log(shipping);
    });
  });
});
```

Problem: Hard to read, hard to debug.

Promises

```
// Pro Way: Promises
getUser(1)
  .then(user => getOrders(user.id))
  .then(orders => getShipping(orders[0].id))
  .then(shipping => console.log(shipping))
  .catch(err => console.error('Error:', err));
```

Flattened structure

Easy error handling

Async/Await

Async/Await is syntactic sugar for Promises. It lets you write asynchronous code like synchronous code.

```
// Pro Way: Async/Await
async function fetchShippingInfo(userId) {
  try {
    const user = await getUser(userId);
    const orders = await getOrders(user.id);
    const shipping = await getShipping(orders[0].id);
    console.log(shipping);
  } catch (err) {
    console.error('Error fetching data:', err);
  }
}
```

Fetching Data from an API

Example: Fetching a list of posts from JSONPlaceholder.

```
async function fetchPosts() {
  try {
    const response = await fetch('https://jsonplaceholder.typicode.com/posts');
    if (!response.ok) throw new Error('Network response was not OK');
    const data = await response.json();
    console.log(data);
  } catch (error) {
    console.error('Fetch failed:', error);
    alert('Unable to load posts. Please try again later.');
  }
}

fetchPosts();
```

- fetch() → Makes the network request.
- await → Waits for the response before moving on.
- try/catch → Handles errors gracefully.

Part 4: The Capstone Mini-Project

Project Chosen: Pokemon Finder Web App

API Used: PokeApi (free public API)

MicroService_Assignment-1 is a full-stack microservice-based application built using **Spring Boot (Backend)** and **JavaScript (Frontend)**.

The backend acts as a microservice that consumes live Pokemon data from the public PokeAPI, while the frontend dynamically renders the data with search, history, and theme persistence features.

This project demonstrates:

- REST API consumption using Spring Boot
- DTO mapping for external APIs
- Microservice architecture principles
- Dynamic DOM manipulation
- LocalStorage persistence
- Error handling and clean UI design

Backend (Spring Boot Microservice):

Architecture :controller → service → external API (PokeAPI)

Live Data (Public API Integration)

The backend consumes live data from:

https://pokeapi.co/api/v2/pokemon/{name}
https://pokeapi.co/api/v2/pokemon/

Using:

```
RestTemplate restTemplate = new RestTemplate();
```

The microservice fetches Pokémon data and maps it into:

PokemonDTO

PokemonDTOList

This ensures the application always displays **real-time data**, not hardcoded content.

REST Endpoints Exposed

Endpoint	Method	Description
/pokemon/search?name=pikachu	GET	Fetch specific Pokémon details
/pokemon/getAll	GET	Fetch list of Pokémon

Example:

http://localhost:8080/pokemon/search?name=pikachu

DTO Mapping

Custom DTOs map only required fields:

PokemonDTO

- name
- height
- weight
- sprites
- types
- abilities

PokemonDTOList

- count
- next
- previous
- results (PokemonSummary)

This keeps the microservice clean and structured.

Error Handling

The controller wraps API calls in try/catch blocks:

```
return new ResponseEntity<>(e.getMessage(), HttpStatus.BAD_REQUEST);
```

If:

Pokémon not found → returns 400

API error → handled gracefully

Frontend (JavaScript)

The frontend is built using:

- HTML
- CSS
- Pure JavaScript (No frameworks)

It communicates with the Spring Boot microservice:

```
const API_BASE = "http://localhost:8080/pokemon";
```

Features Implemented

1 Live Search

User enters a Pokémon name → clicks Search → triggers:

```
fetch(`${API_BASE}/search?name=${name}`)
```

The result is dynamically rendered using:

- innerHTML
- Template literals
- Array .map() for types & abilities

No hardcoded data exists in HTML.

2 View All Pokémon (Mini Cards Grid)

The **View All** button calls:

/pokemon/getAll

Frontend:

- Extracts Pokémon ID from URL

Generates image dynamically

Creates clickable mini cards

Clicking a card auto-searches that Pokémon

3 Dynamic DOM Rendering

All cards are created dynamically:

document.createElement()

The HTML file contains no predefined Pokémon cards.

Each card displays:

Pokémon name

Sprite image

Height & weight

Abilities

Type badges

4 Persistence Using LocalStorage

The app stores:

Stored Data	Purpose
Search History	Remembers last 6 searches
Theme Preference	Saves dark/light mode

Stored using:

JSON.stringify()JSON.parse()

On page load:

History is restored

Theme preference is applied automatically

5 Dark Mode Feature

Theme toggle:

```
document.body.classList.toggle('dark-mode');
```

Theme state saved in:

localStorage → pokeTheme

6 Robust Error Handling

Handled cases:

- Invalid Pokémon name → Alert shown
- Backend not running → Friendly message displayed
- Network error → Prevents blank screen

Improves overall user experience.

How to Run the Project

Backend

```
cd Backend
mvn spring-boot:run
```

Runs on:

http://localhost:8080

Frontend

Open index.html directly in browser
OR use Live Server extension.

Microservice Concepts Demonstrated

✔ Service isolation
✔ External API consumption
✔ DTO abstraction
✔ RESTful endpoint design
✔ Frontend-backend communication
✔ Stateless backend architecture

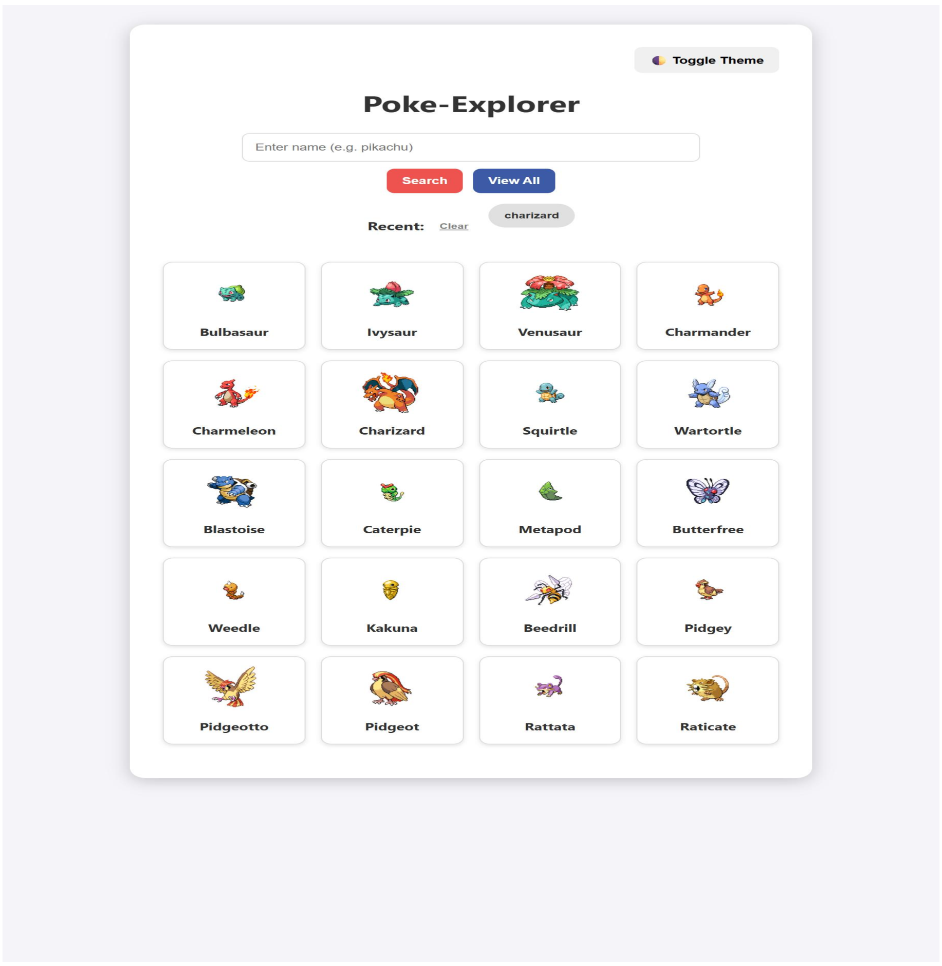
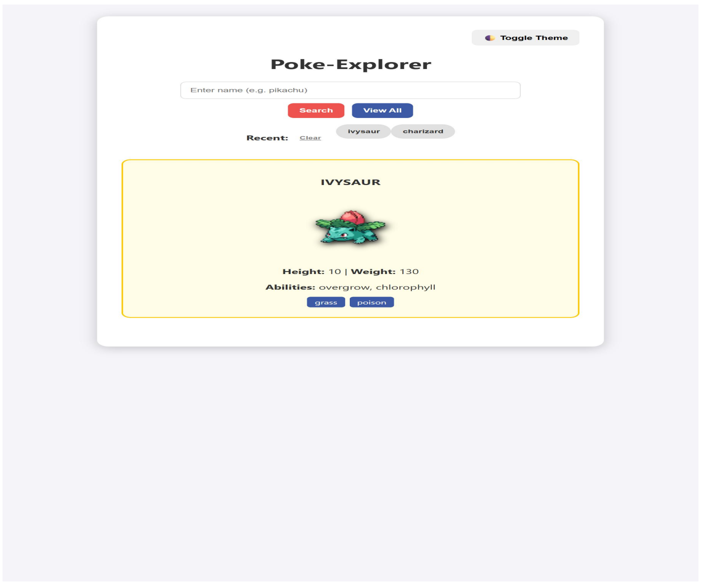


Fig.1: Showing the interface of the Application



Height: 10 | **Weight:** 130
Abilities: overgrow, chlorophyll

Fig.2. Showing the Search Feature in actions