Internship Report – Frontend Dev

Week 5: JavaScript Advanced Topics

Name: Zainab
Father Name: Assad Qayyum
Date: 24th July, 2025
Internship Domain: Front-end Intern
Task: JS - Asynchronous JS: setTimeout, setInterval, Promises

Task Overview: (Day4)

Today's task was to explore and understand advanced JavaScript concepts related to asynchronous programming. The primary focus was on the behavior and usage of **setTimeout, setInterval,** and **Promises,** which help execute time-based or delayed actions in JavaScript without blocking the main thread.

Content Covered:

Asynchronous JavaScript:

- setTimeout
- setInterval
- Promises

Asynchronous JavaScript:

Asynchronous JavaScript allows the code to run without waiting for long tasks (like fetching data or waiting for a timer) to finish. It lets JavaScript continue executing other code while those tasks complete in the background.

- JavaScript is **single-threaded;** it can run only one line of code at a time
- But sometimes, tasks take a **long time** — like waiting for data from a server, reading a file etc.
- To **avoid blocking** the rest of the code while waiting, JS uses **asynchronous programming.**

1. setTimeout

The setTimeout() function allows you to execute a block of code **once after a specified delay** (in milliseconds).

Syntax:

setTimeout(callbackFunction, delayInMilliseconds);

Key Properties:

- **callbackFunction:** The function to execute after the delay.
- **delayInMilliseconds:** Time to wait before executing (1000 ms = 1 sec).

Example:

```
console.log("Start");

setTimeout(() => {
  console.log("Runs after 3 seconds");
}, 3000);

console.log("End");
```

- "Start" prints.
- "End" prints immediately.
- After 3 seconds, "Runs after 3 seconds" prints.

Use case:

Displaying messages after form submission

Introducing timed transitions or animations

2. setInterval

The setInterval() function is used to **repeatedly run** a block of code at a fixed interval.

Syntax:

setInterval(callbackFunction, intervalInMilliseconds);

Key Properties:

- **callbackFunction:** The function to repeat
- **intervalInMilliseconds:** Delay between each repeat

Example:

```
setInterval(() => {
  console.log("This runs every 2 seconds");
}, 2000);
```

This will keep printing the message every 2 seconds, forever, unless you stop it.

To stop it:

```
let id = setInterval(() => {
  console.log("Repeating...");
}, 1000);

// stop after 5 seconds
setTimeout(() => {
  clearInterval(id);
  console.log("Stopped the interval");
}, 5000);
```

Use Case:

Building clocks or countdowns

Auto-refreshing content like notifications

3. Promises

A **Promise** is a way to handle asynchronous operations in a more readable and manageable way. It represents a value that may be available **now, later, or never.**

States of a Promise:

- **Pending:** Initial state, waiting.
- **Resolved (Fulfilled):** Task completed successfully.
- **Rejected:** Task failed.

Syntax:

```
const myPromise = new Promise((resolve, reject) => {
  // async operation
  if (success) {
    resolve("Done");
  } else {
    reject("Error");
  }
});
myPromise
  .then(result => console.log(result))
  .catch(error => console.error(error));
```

Example with delay:

```
function delayPromise() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      resolve("Task completed after 2 seconds");
    }, 2000);
  });
}
delayPromise()
  .then(result => console.log(result))
  .catch(error => console.log(error));
```

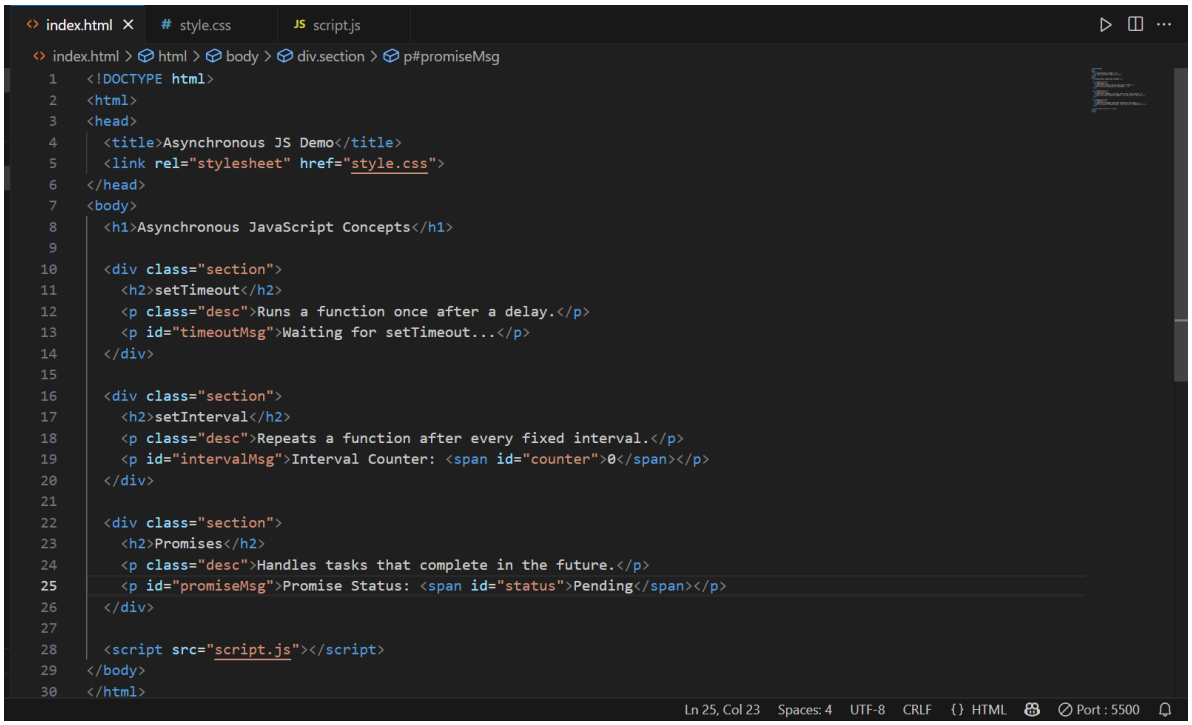
This Promise waits 2 seconds, then resolves with a message.

Use Case:

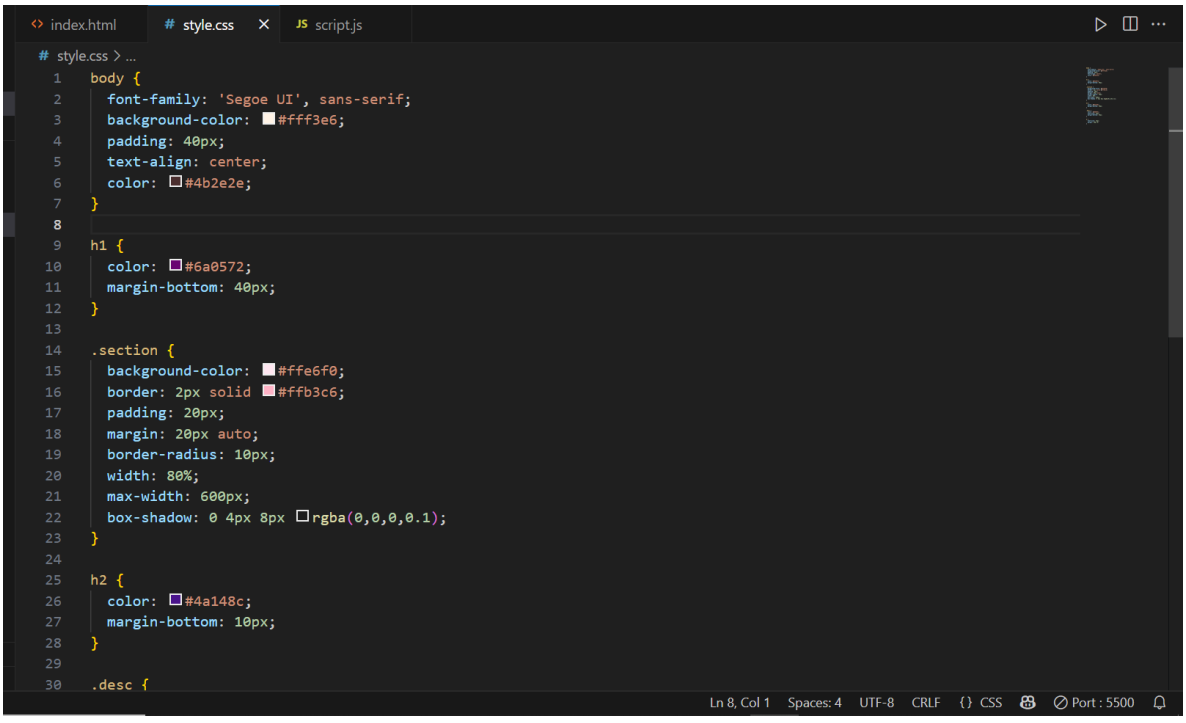
- Fetching API data
- Waiting for user actions
- Simulating success or failure of network tasks

Practice Code:

Html:



CSS:



```
index.html  style.css  script.js
style.css > ...
14  .section {
23  }
24
25  h2 {
26    color: #4a148c;
27    margin-bottom: 10px;
28  }
29
30  .desc {
31    color: #00695c;
32    font-size: 16px;
33    margin-bottom: 10px;
34  }
35
36  p {
37    font-size: 18px;
38    margin: 10px 0;
39  }
40
```

JavaScript:

```
index.html  style.css  script.js
script.js > intervalId > setInterval() callback
1   setTimeout(() => {
2     document.getElementById("timeoutMsg").textContent = "setTimeout: This message appeared after 3 seconds!";
3   }, 3000);  //setTimeout
4
5   let count = 0;
6   const intervalId = setInterval(() => {
7     count++;
8     document.getElementById("counter").textContent = count;
9     if (count >= 7) {
10      clearInterval(intervalId); // stop after 5 count
11    }
12  }, 1000);
13  // Promise Example
14  const asyncTask = new Promise((resolve, reject) => {
15    setTimeout(() => {
16      let success = true; // Simulate result
17      if (success) {
18        resolve("Resolved: Task completed!");
19      } else {
20        reject("Rejected: Task failed!");
21      }
22    }, 4000);
23  });
24
25  asyncTask
26    .then(result => {
27      document.getElementById("status").textContent = result; })
28    .catch(error => {
29      document.getElementById("status").textContent = error;
30    });
Ln 10, Col 53  Spaces: 4  UTF-8  CRLF  {} JavaScript  Port : 5500
```

Before timeout and original values:

After timeout values will be:

Conclusion:

In today's task, I explored key **Asynchronous techniques in JavaScript** including **setTimeout, setInterval,** and **Promises.** These tools help JavaScript handle time-based actions and background tasks without freezing the interface. Understanding these concepts is crucial for building dynamic, responsive web applications that interact smoothly with users and external data.