Advanced Function Patterns

(Memoization, Debouncing & Throttling Basics)

Subtitle: Learn how real-world applications optimize performance by controlling function execution, caching results, and reducing unnecessary work.

Website Name: haas.dev
Website Link: https://dev-roast-app.vercel.app

🧠 Introduction

At this stage, you already know how functions work internally:

- execution context
- closures
- callbacks
- higher-order functions

Now the focus shifts from **“how functions work”** to the following:

how real systems optimize function behavior under load

Modern applications don’t just run functions — they **control when, how often, and whether they should run at all**.

This is where

- memoization
- debouncing
- throttling

become essential.

⚙️ Step 1: Memoization (Caching Function Results)

Memoization means:

Storing previously calculated results to avoid repeating expensive work.

Problem Without Memoization:

```
function slowSquare(n) {

  console.log("Computing...");

  return n * n;
```

```
}


slowSquare(5);

slowSquare(5);
```

Issue:

- same calculation repeated
- unnecessary performance cost

Solution: Memoization

```
function memoizedSquare() {

  let cache = {};


  return function (n) {

    if (cache[n]) {

      return cache[n];

    }


    console.log("Computing...");

    cache[n] = n * n;

    return cache[n];

  };

}


const square = memoizedSquare();


square(5);

square(5);
```

Key Idea:

- first call → compute
- next calls → reuse cached value

🧠 Step 2: Why Memoization Works

It relies on:

- closures
- persistent memory
- lookup optimization

👉 This is why closures are critical in real systems.

⚙️ Step 3: Debouncing (Delay Execution Until User Stops)

Debouncing means:

Execute function only after a pause in repeated actions.

Problem Scenario:

User types in search box:

h

he

hel

hell

hello

Each keypress triggers API call → inefficient.

Solution: Debounce

```
function debounce(fn, delay) {

  let timer;


  return function (...args) {
```

```
    clearTimeout(timer);


    timer = setTimeout(() => {
      fn.apply(this, args);
    }, delay);
  };
}
```

Usage:

```
const search = debounce(function (text) {
  console.log("Searching:", text);
}, 500);
```

Behavior:

- waits until user stops typing
- then executes once

🌍 Step 4: Real Use Cases of Debouncing

- search bars
- form validation
- window resize events
- auto-save systems

⚙️ Step 5: Throttling (Limit Execution Frequency)

Throttling means:

Execute function at fixed intervals, no matter how many times it is triggered.

Problem Scenario:

User scrolls page → scroll event fires 100s of times per second.

Solution: Throttle

```
function throttle(fn, limit) {
```

```
  let lastCall = 0;

  return function (...args) {
    let now = Date.now();

    if (now - lastCall >= limit) {
      lastCall = now;
      fn.apply(this, args);
    }
  };
}
```

Usage:

```
const handleScroll = throttle(() => {
  console.log("Scroll event");
}, 1000);
```

Behavior:

- runs at fixed intervals
- ignores extra calls

🧠 Step 6: Debounce vs Throttle

Feature	Debounce	Throttle
Execution	after delay	at intervals
Use case	search input	scroll/resize
Behavior	waits for pause	limits frequency

⚠️ Step 7: Common Mistakes

❌ 1. Confusing debounce and throttle

They solve different problems.

❌ 2. Overusing memoization

Caching everything can increase memory usage.

❌ 3. Not clearing timers in debounce

Leads to unexpected execution.

❌ 4. Misunderstanding closures dependency

All 3 patterns rely heavily on closure memory.

🧪 Step 8: Mini Exercises

Exercise 1

Implement memoized addition function.

Exercise 2

Create debounce for button click.

Exercise 3

Create throttle for scroll event.

🧠 Step 9: Mini Quiz

1. What is memoization?
2. Difference between debounce and throttle?
3. Why do these patterns use closures?
4. Where is throttling used in real systems?

🚀 Step 10: Thinking Upgrade

If you understand these patterns:

- you are no longer writing basic functions
- you are designing performance systems

- you understand real-world frontend/backend optimization

👉 This is production-level JavaScript thinking.

📌 Step 11: Summary

- Memoization = caching results
- Debounce = delay until action stops
- Throttle = limit execution frequency
- All rely on closures + timers
- Used in real UI and system optimization
- Core performance engineering patterns