Function Composition

(Building Systems from Small Functions)

Subtitle: Learn how real software is built by combining simple functions into powerful, scalable logic chains.

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

🧠 Introduction

So far, you've learned:

- functions as blocks
- higher-order functions
- callbacks

Now comes the real engineering idea:

Instead of writing one big function, we build many small functions and combine them.

This is called **Function Composition**.

Most beginners write:

- large messy functions
- mixed logic
- hard-to-debug code

Real developers do the opposite:

- small functions
- single responsibility
- composed systems

⚙️ Step 1: What is Function Composition?

Function composition means:

Combining multiple functions so the output of one becomes input of another.

Simple Example:

function add2(x) {

```
  return x + 2;
}

function multiply3(x) {
  return x * 3;
}

const result = multiply3(add2(5));
console.log(result);
```

Execution Flow:

5 → add2 → 7 → multiply3 → 21

🧠 Step 2: Why Composition Exists

Without composition:

- logic becomes large and messy
- reuse becomes difficult
- debugging becomes slow

With composition:

- each function does one job
- logic becomes predictable
- systems become scalable

Mental Model:

Input → Function A → Function B → Function C → Output

⚙️ Step 3: Breaking Big Problems into Small Functions

Example: User Processing System

Instead of one big function:

```
function processUser(user) {
  // validation + formatting + saving + logging
}
```

We split:

```
function validate(user) {
  return user;
}

function format(user) {
  return { ...user, formatted: true };
}

function save(user) {
  return user;
}
```

Now compose:

```
const result = save(format(validate({ name: "Ali" })));
```

🔄 Step 4: Manual Composition Pattern

```
function compose(a, b) {
  return function (x) {
    return a(b(x));
  };
}
```

Usage:

```
const add2 = x => x + 2;
```

```
const multiply3 = x => x * 3;


const composed = compose(multiply3, add2);


console.log(composed(5));
```

🌍 Step 5: Real-World Use Cases

1. Data Transformation Pipelines

```
const clean = x => x.trim();

const lower = x => x.toLowerCase();

const removeSpaces = x => x.replace(" ", "");


const result = removeSpaces(lower(clean("  Hello World  ")));
```

2. Backend Middleware Chains

Request → Auth → Validation → Processing → Response

3. Frontend UI Logic

- format data
- filter data
- render UI

Each step = separate function

🧠 Step 6: Composition vs Nested Logic

Approach	Problem
Nested logic	messy, hard to debug
Composition	clean, modular

Bad Style:

```
function process(x) {
  return ((x + 2) * 3) - 1;
}
```

Good Style:

```
const add2 = x => x + 2;
const mul3 = x => x * 3;
const sub1 = x => x - 1;


const result = sub1(mul3(add2(5)));
```

⚠️ Step 7: Common Mistakes

❌ 1. Over-complicating simple logic

Not everything needs composition

❌ 2. Too many tiny functions

Leads to unreadable structure

❌ 3. Ignoring execution order

Composition depends on order

❌ 4. Mixing side effects

Pure functions are best for composition

🧪 Step 8: Mini Exercises

Exercise 1

Create 3 functions and compose them manually.

Exercise 2

Build a text processing pipeline.

Exercise 3

Simulate user data transformation system.

🧠 Step 9: Mini Quiz

1. What is function composition?
2. Why is it better than nested logic?
3. What does compose(a, b) mean?
4. Where is composition used in real systems?

🚀 Step 10: Thinking Upgrade

If you understand composition:

- you stop writing monolithic functions
- you start thinking in pipelines
- you design scalable systems

👉 This is real engineering mindset shift.

📌 Step 11: Summary

- Function composition = combining functions
- Output of one becomes input of another
- Improves modularity and reuse
- Used in data pipelines and backend systems
- Core concept in functional programming