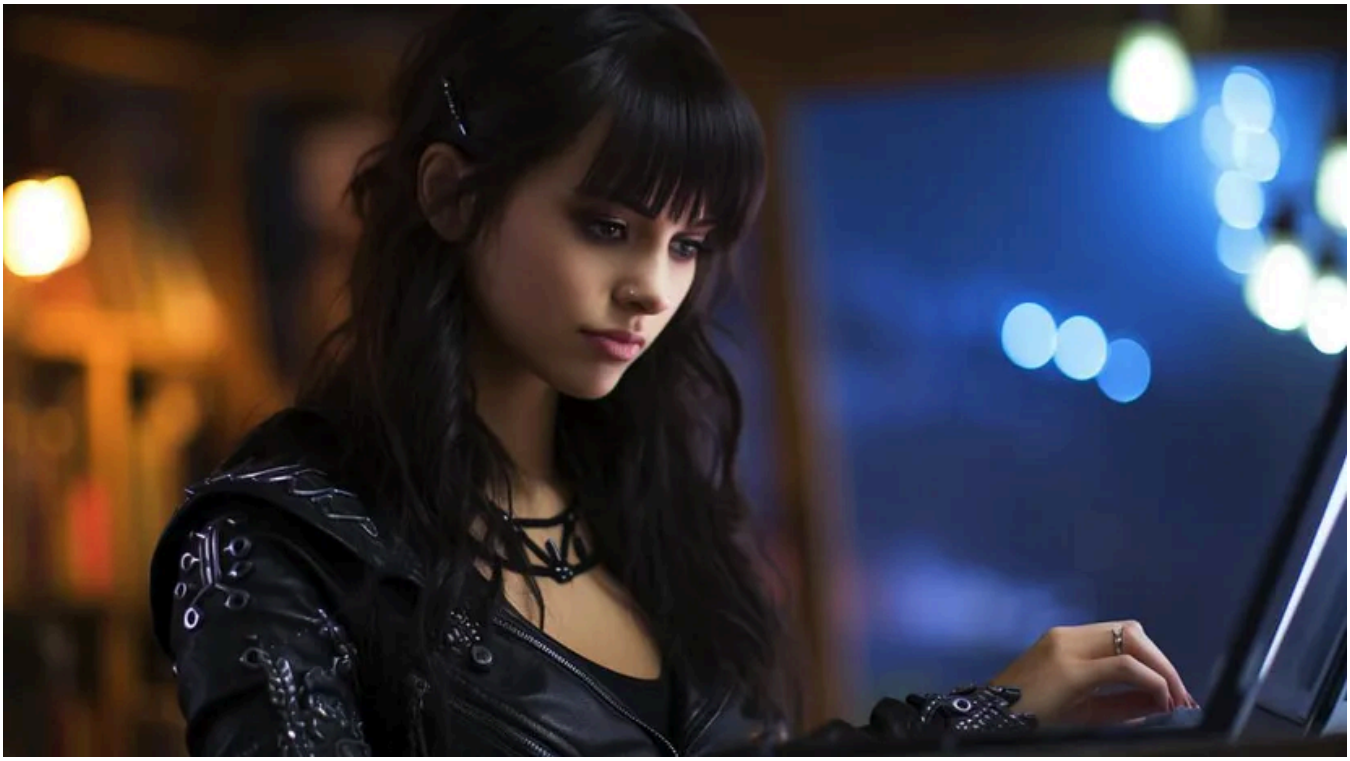
10 Interview Questions Every JavaScript Developer Should Know in 2024

Eric Elliott · Follow

Published in JavaScript Scene
12 min read · Dec 31, 2023

Listen Share



The world of JavaScript has evolved significantly, and interview trends have changed a lot over the years. This guide features 10 essential questions that every JavaScript developer should know the answers to in 2024. It covers a range of topics from closures to TDD, equipping you with the knowledge and confidence to tackle modern JavaScript challenges.

As a hiring manager, I use all of these questions in real technical interviews on a regular basis.

When engineers don't know the answers, I don't automatically reject them. Instead, I teach them the concepts and get a sense of how well they listen, and learn, and deal with the stressful situation of not knowing the answer to an interview question.

A good interviewer is looking for people who are eager to learn and advance their understanding and their career. If I'm hiring for a less experienced role, and the candidate fails all of these questions, but demonstrates a good aptitude for learning, they may still land the job!

1. What is a Closure?

A closure gives you access to an outer function's scope from an inner function. When functions are nested, the inner functions have access to the variables declared in the outer function scope, even after the outer function has returned:

```
const createSecret = (secret) => {
  return {
    getSecret: () => secret,
    setSecret: (newSecret) => {
      secret = newSecret;
    },
  };
};

const mySecret = createSecret("My secret");
console.log(mySecret.getSecret()); // My secret

mySecret.setSecret("My new secret");
console.log(mySecret.getSecret()); // My new secret
```

Closure variables are live references to the outer-scoped variable, not a copy. This means that if you change the outer-scoped variable, the change will be reflected in the closure variable, and vice versa, which means that other functions declared in the same outer function will have access to the changes.

Common use cases for closures include:

- Data privacy
- Currying and partial applications (frequently used to improve function composition, e.g. to parameterize Express middleware or React higher order components)

- Sharing data with event handlers and callbacks

Data Privacy

Encapsulation is a vital feature of object oriented programming. It allows you to hide the implementation details of a class from the outside world. Closures in JavaScript allow you to declare private variables for objects:

```
// Data privacy
const createCounter = () => {
  let count = 0;
  return {
    increment: () => ++count,
    decrement: () => --count,
    getCount: () => count,
  };
};
```

Curried functions and partial applications:

```
// A curried function takes multiple arguments one at a time.
const add = (a) => (b) => a + b;

// A partial application is a function that has been applied to some,
// but not yet all of its arguments.
const increment = add(1); // partial application

increment(2); // 3
```

2. What is a Pure Function?

Pure functions are important in functional programming. Pure functions are predictable, which makes them easier to understand, debug, and test than impure functions. Pure functions follow two rules:

1. **Deterministic** — given the same input, a pure function will always return the same output.

2. **No side-effects** — A side effect is any application state change that is observable outside the called function other than its return value.

Examples of Non-deterministic Functions

Non-deterministic functions include functions that rely on:

- A random number generator.
- A global variable that can change state.
- A parameter that can change state.
- The current system time.

Examples of Side Effects

- Modifying any external variable or object property (e.g., a global variable, or a variable in the parent function scope chain).
- Logging to the console.
- Writing to the screen, file, or network.
- Throwing an error. Instead, the function should return a result indicating the error.
- Triggering any external process.

In Redux, all reducers must be pure functions. If they are not, the state of the application will be unpredictable, and features like time-travel debugging will not work. Impurity in reducer functions may also cause bugs that are difficult to track down, including stale React component state.

3. What is Function Composition?

Function composition is the process of combining two or more functions to produce a new function or perform some computation: `(f ∘ g)(x) = f(g(x))` (`f` composed with `g` of `x` equals `f` of `g` of `x`).

```
const compose = (f, g) => (x) => f(g(x));
```

```
const g = (num) => num + 1;
const f = (num) => num * 2;

const h = compose(f, g);

h(20); // 42
```

React developers can clean up large component trees with function composition. Instead of nesting components, you can compose them together to create a new higher-order component that can enhance any component you pass to it with additional functionality.

4. What is Functional Programming?

Functional programming is a programming paradigm that uses pure functions as the primary units of composition. Composition is so important in software development that virtually all programming paradigms are named after the units of composition they use:

- Object-oriented programming uses objects as the unit of composition.
- Procedural programming uses procedures as the unit of composition.
- Functional programming uses functions as the unit of composition.

Functional programming is a declarative programming paradigm, which means that programs are written in terms of what they do, rather than how they do it. This makes functional programs easier to understand, debug, and test than imperative programs. They also tend to be a lot more concise, which reduces code complexity and makes it easier to maintain.

Other key aspects of functional programming include:

- **Immutability** — immutable data structures are easier to reason about than mutable data structures.
- **Higher-order functions** — functions that take other functions as arguments or return functions as their result.
- **Avoiding shared mutable state** — shared mutable state makes programs difficult to understand, debug, and test. It also makes it difficult to reason about the correctness of a program.

Since pure functions are easy to test, functional programming also tends to lead to better test coverage and fewer bugs.

5. What is a Promise?

A Promise in JavaScript is an object representing the eventual completion or failure of an asynchronous operation. It acts as a placeholder for a value that is initially unknown, typically because the computation of its value is not yet complete.

Key Characteristics of Promises:

Stateful: A Promise is in one of three states:

- **Pending:** Initial state, neither fulfilled nor rejected.
- **Fulfilled:** The operation completed successfully.
- **Rejected:** The operation failed.

Immutable: Once a Promise is fulfilled or rejected, its state cannot change. It becomes immutable, permanently holding its result. This makes Promises reliable in asynchronous flow control.

Chaining: Promises can be chained, meaning the output of one Promise can be used as input for another. This is done using `.then()` for success or `.catch()` for handling failures, allowing for elegant and readable sequential asynchronous operations. Chaining is the async equivalent of function composition.

```
const promise = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve("Success!");
    // You could also reject with a new error on failure.
  }, 1000);
});

promise
  .then((value) => {
    console.log(value); // Success!
  })
  .catch((error) => {
    console.log(error);
  });
```

In JavaScript, you can treat promises and promise returning functions as if they are synchronous, using the async/await syntax. This makes asynchronous code much easier to read and reason about.

```
const processData = async () => {
  try {
    const data = await fetchData(); // Waits until the Promise is resolved
    console.log("Processed:", data); // Process and display the data
  } catch (error) {
    console.error("Error:", error); // Handle any errors
  }
};
```

6. What is TypeScript?

TypeScript is a superset of JavaScript, developed and maintained by Microsoft. It has grown significantly in popularity in recent years, and chances are good that if you are a JavaScript engineer, you will eventually need to use TypeScript. It adds static typing to JavaScript, which is a dynamically typed language. Static typing helps developers catch errors early in the development process, improving code quality and maintainability.

Key Features of TypeScript:

Static Typing: Define types for your variables and function parameters to ensure consistency throughout your code.

Enhanced IDE Support: Integrated Development Environments (IDEs) can provide better autocompletion, navigation, and refactoring, making the development process more efficient.

Compilation: TypeScript code is transpiled into JavaScript, making it compatible with any browser or JavaScript environment. During this process, type errors are caught, making the code more robust.

Interfaces: Interfaces allow you to specify abstract contracts that objects and functions must satisfy.

Compatibility with JavaScript: TypeScript is highly compatible with existing JavaScript code. JavaScript code can be gradually migrated to TypeScript, making

the transition smooth for existing projects.

```
interface User {
  id: number;
  name: string;
}

type GetUser = (userId: number) => User;

const getUser: GetUser = (userId) => {
  // Fetch user data from a database or API
  return {
    id: userId,
    name: "John Doe",
  };
};
```

The best defenses against bugs are code review, TDD, and lint tools such as ESLint. TypeScript is not a substitute for these practices, because type correctness does not guarantee program correctness. TypeScript does occasionally catch bugs even after all your other quality measures have been applied. But its main benefit is the improved developer experience it provides via IDE support.

7. What is a Web Component?

Web Components are a set of web platform APIs that allow you to create new custom, reusable, encapsulated HTML tags to use in web pages and web apps. They are built using open web technologies such as HTML, CSS, and JavaScript. They are part of the browser, and do not require external libraries or frameworks.

Web Components are particularly useful on large teams with many engineers who may be using different frameworks. They allow you to create reusable components that can be used in any framework, or no framework at all. For example, Adobe's Spectrum design system is built using Web Components, and integrates smoothly with popular frameworks like React.

Web Components have existed for a long time, but have grown in popularity recently, especially in large organizations. They are supported by all major browsers, and are a W3C standard.

```
<!-- Defining a simple Web Component -->
<script>
  // Define a class that extends HTMLElement
  class SimpleGreeting extends HTMLElement {
    // Define a constructor that attaches a shadow root
    constructor() {
      super();
      const shadowRoot = this.attachShadow({ mode: "open" });
      // Use a template literal for the shadow root's innerHTML
      shadowRoot.innerHTML = `
        <style>
          /* Style the web component using a style tag */
          p {
            font-family: Arial, sans-serif;
            color: var(--color, black); /* Use a CSS variable for the color */
          }
        </style>
        <!-- The <slot> element is a placeholder for user-provided content. -->
        <!-- If no content is provided, it displays its own default content. --
        <p><slot>Hello, Web Components!</slot></p>
      `;
    }

    // Define a static getter for the observed attributes
    static get observedAttributes() {
      return ["color"]; // Observe the color attribute
    }

    // Define a callback for when an attribute changes
    attributeChangedCallback(name, oldValue, newValue) {
      // Update the CSS variable when the color attribute changes
      if (name === "color") {
        this.style.setProperty("--color", newValue);
      }
    }
  }

  // Register the custom element with a tag name
  customElements.define("simple-greeting", SimpleGreeting);
</script>

<!-- Using the Web Component -->
<!-- Pass a custom greeting message using the slot -->
<simple-greeting>Hello, reader!</simple-greeting>
<!-- Pass a custom color using the attribute -->
<simple-greeting color="blue">Hello, World!</simple-greeting>
```

8. What is a React Hook?

Hooks are functions that let you use state and other React features without writing a class. Hooks allow you to use state, context, refs, and component lifecycle events by calling functions instead of writing class methods. The additional flexibility of functions allows you to organize your code better, grouping related functionality together in a single hook call, and separating unrelated functionality by implementing it in separate function calls. Hooks offer a powerful and expressive way to compose logic inside a component.

Important React Hooks

- `useState` - allows you to add state to functional components. State variables are preserved between re-renders.
- `useEffect` - lets you perform side effects in functional components. It combines the capabilities of `componentDidMount`, `componentDidUpdate`, and `componentWillUnmount` into a single function call, reducing the required code and creating better code organization than class components.
- `useContext` - allows you to consume context in function components.
- `useRef` - allows you to create a mutable reference that persists for the lifetime of the component.
- **Custom Hooks** — to encapsulate reusable logic. This makes it easy to share logic across different components.

Rules of Hooks: Hooks must be used at the top level of React functions (not inside loops, conditions, or nested functions) and only in React function components or custom Hooks.

Hooks solved some common pain points with class components, such as the need to bind methods in the constructor, and the need to split functionality into multiple lifecycle methods. They also make it easier to share logic between components, and to reuse stateful logic without changing your component hierarchy.

9. How Do you Create a Click Counter in React?

You can create a click counter in React by using the `useState` hook as follows:

```
import React, { useState } from "react";
```

```
const ClickCounter = () => {
  const [count, setCount] = useState(0); // Initialize count to 0

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount((count) => count + 1)}>Click me</button>
    </div>
  );
};

export default ClickCounter;
```

Note that passing a function to `setCount` is best practice when you are deriving the new value from existing state, to ensure that you're always working with the latest state.

10. What is Test Driven Development (TDD)?

Test Driven Development (TDD) is a software development approach where tests are written before the actual code. It revolves around a short, repetitive development cycle designed to ensure that the code meets specified requirements and is free of bugs. TDD can play a vital role in improving code quality, reducing bugs, and increasing developer productivity.

One of the most important measures of development team productivity is deployment frequency. One of the primary obstacles to continuous delivery is the fear of change. TDD helps to reduce this fear by ensuring that the code is always in a deployable state. This makes it easier to deploy new features and bug fixes, which in turn increases deployment frequency.

Testing first has many benefits over testing after:

- **Better Code Coverage:** Tests are more likely to cover all edge cases when they are written first.
- **Improved API Design:** Tests force you to think about the API design before you write the code, which helps avoid leaking implementation details into the API.
- **Fewer Bugs:** Testing first helps you catch bugs earlier in the development process, when they are easier to fix.

- **Better Code Quality:** Testing first forces you to write modular, loosely coupled code, which is easier to maintain and reuse.

The final point is my favorite feature of TDD, and it taught me most of what I know about writing modular, cleanly architected code.

Key Steps in TDD:

1. **Write a Test:** This test will fail initially, as the corresponding functionality does not yet exist.
2. **Write the Implementation:** Just enough to make the test pass.
3. **Refactor with Confidence:** Once the test passes, the code can be refactored with confidence. Refactoring is the process of restructuring existing code without changing its external behavior. Its purpose is to clean up the code, improve readability, and reduce complexity. With the test in place, if you make a mistake, you will be alerted to it immediately by the test failure.

Repeat: The cycle repeats for each functional requirement, gradually building up the software while ensuring that all tests continue to pass.

Challenges

- **Learning Curve:** TDD is a skill and discipline that can take considerable time to develop. After 6 months of TDD, you may still feel like TDD is difficult and gets in the way of productivity. However, after 2 years with TDD, you will likely find that it has become second nature, and that you are more productive than ever before.
- **Time-Consuming:** Writing tests for every small functionality can feel time-consuming initially, though it usually pays off in the long term with reduced bugs and easier maintenance. I often tell people, “if you think you don’t have time for TDD, you *really* don’t have time to skip TDD.”

Conclusion

Preparing yourself to answer these questions in an interview setting will certainly help you stand out from the crowd. It will help you become a better JavaScript developer, and that will help you thrive in your new role.

Next Steps

The fastest way to level up your career is 1:1 mentorship. With that in mind, I cofounded a platform that pairs engineers and engineering leaders with senior mentors who will meet with you via video every week. Topics include *JavaScript, TypeScript, React, TDD, AI Driven Development, and Engineering Leadership*. Join today at DevAnywhere.io.

Prefer to learn about topics like functional programming and JavaScript on your own? Check out EricElliottJS.com or purchase my book, Composing Software in ebook or print.

Eric Elliott *is an Engineering Manager for Adobe Firefly, a tech product and platform advisor, author of "Composing Software", creator of SudoLang (the AI programming language), creator of EricElliottJS.com and cofounder of DevAnywhere.io. He has contributed to software experiences for* ***Adobe Systems, Zumba Fitness, The Wall Street Journal, ESPN, BBC,*** *and top recording artists including* ***Usher, Frank Ocean, Metallica,*** *and many more.*

Technology JavaScript Typescript React Tdd

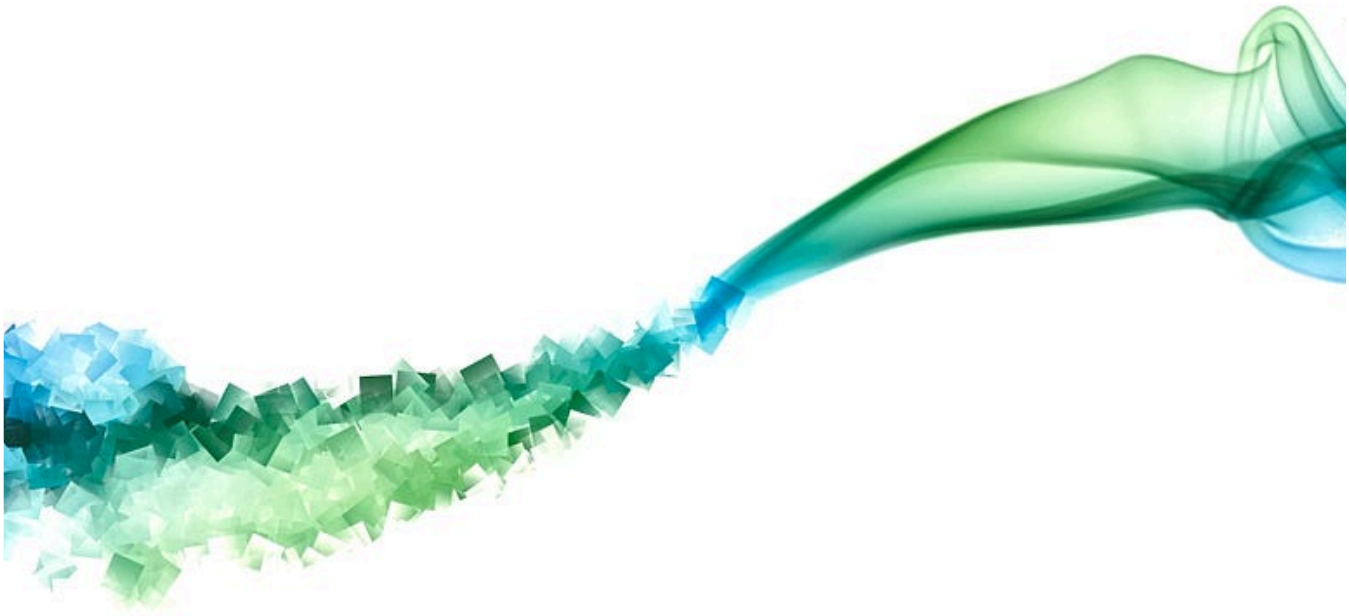
Follow

Written by Eric Elliott

115K Followers · Editor for JavaScript Scene

Make some magic. #JavaScript

More from Eric Elliott and JavaScript Scene



 Eric Elliott in JavaScript Scene

Mocking is a Code Smell

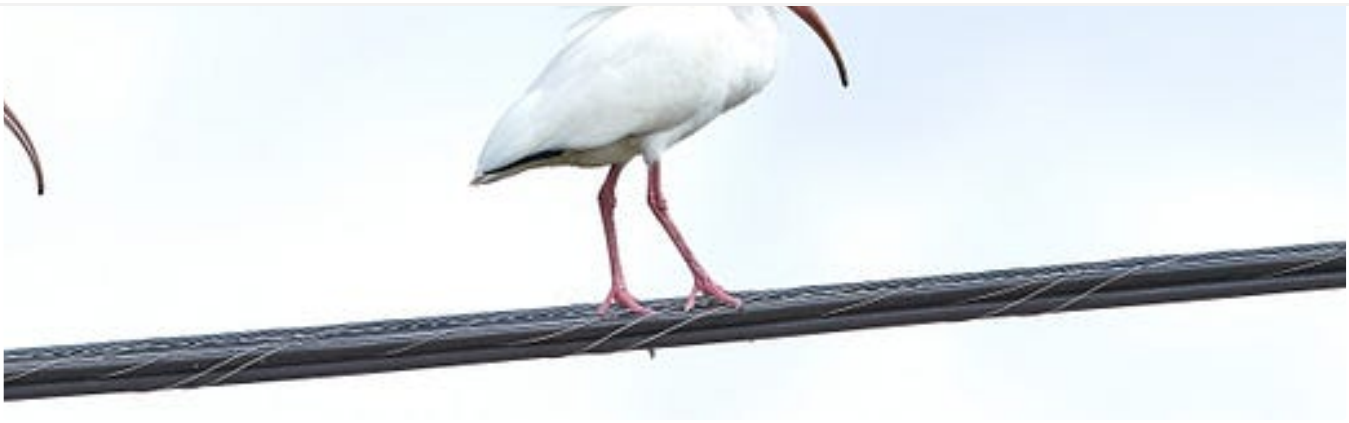
Note: This is part of the “Composing Software” series (now a book!) on learning functional programming and compositional software...

23 min read · Oct 21, 2017

 19.4K  96 

Open in app ↗ Sign up Sign in

Medium Search



Eric Elliott in JavaScript Scene

Software Roles and Titles

I've noticed a lot of confusion in the industry about various software roles and titles, even among founders, hiring managers, and team...

15 min read · Mar 11, 2019

7.3K 34

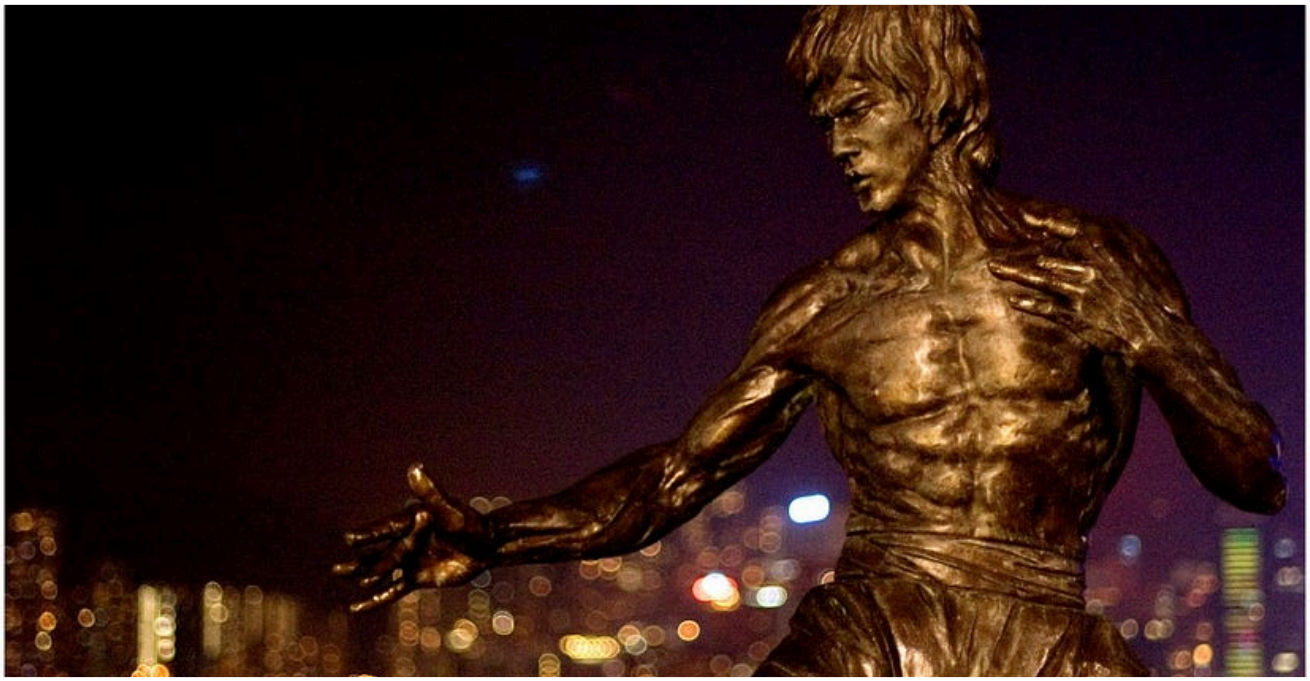
Eric Elliott in JavaScript Scene

Master the JavaScript Interview: What is a Pure Function?

Pure functions are essential for a variety of purposes, including functional programming, reliable concurrency, and React+Redux apps. But...

9 min read · Mar 26, 2016

10.1K 34



Eric Elliott in JavaScript Scene

10 Interview Questions Every JavaScript Developer Should Know

AKA: The Keys to JavaScript Mastery

13 min read · Oct 3, 2015

50K 182

See all from Eric Elliott

See all from JavaScript Scene

Recommended from Medium

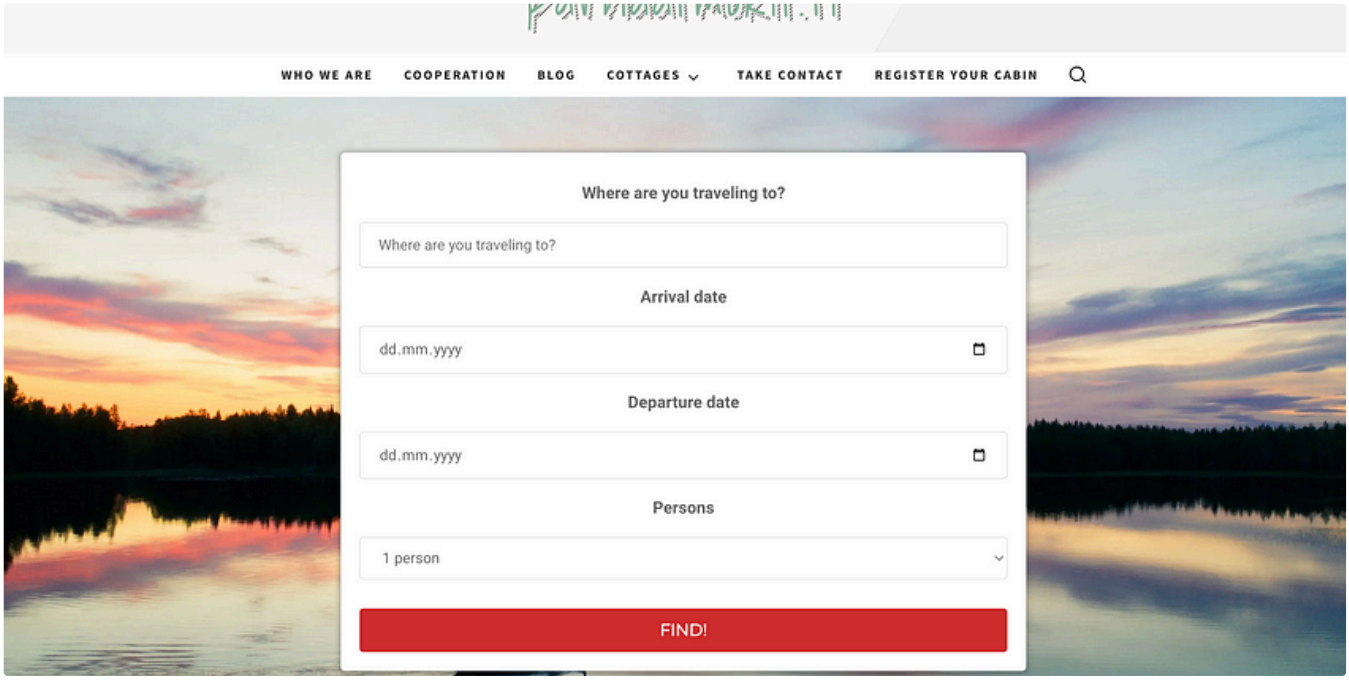
Choco

The React Ecosystem in 2024

Unveiling the Evolved Landscape: Navigating the React Ecosystem in 2024

9 min read · Jan 17, 2024

2K 23



Artturi Jalli

I Built an App in 6 Hours that Makes $1,500/Mo

Copy my strategy!

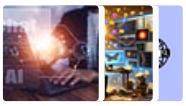
· 3 min read · Jan 24, 2024

12K 143

Lists

AI Regulation

6 stories · 349 saves

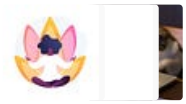


ChatGPT prompts

44 stories · 1206 saves

Generative AI Recommended Reading

52 stories · 786 saves



Stories to Help You Grow as a Software Developer

19 stories · 871 saves

Xiuer Old in Stackademic

14 JavaScript Interview Difficult Questions And Code Implementation

🍦This article will give you an in-depth look at 14 common JavaScript senior interview questions. These questions cover JavaScript's...

✨ · 11 min read · Jan 30, 2024

341 3

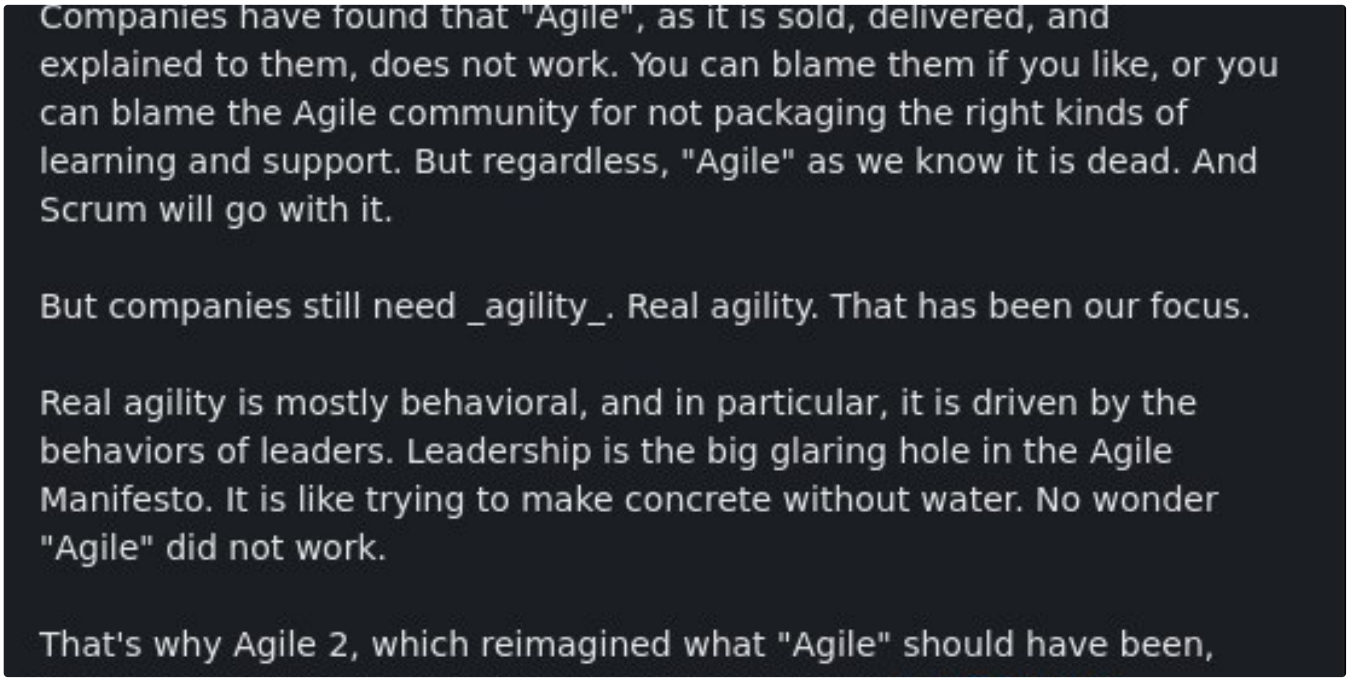
fatfish in JavaScript in Plain English

Interview: Can You Stop "forEach" in JavaScript?

There are 3 ways to stop forEach in JavaScript

✨ · 5 min read · Feb 15, 2024

790 18



Tamás Polgár in Developer rants

Agile has failed. Officially.

Either I'm a gifted oracle, and all of my friends are, or Agile really was just a stupid idea to begin with. After many years of agony...

2 min read · Dec 3, 2023

15.2K 478

Attila Vágó in Level Up Coding

The Inevitable Decline Of TypeScript Has Begun

And I, for one, saw it coming...

· 8 min read · Sep 14, 2023

2.3K 137

See more recommendations