Engineer

JavaScript

Programmiersprache, die Webbrowser ausführen

Was es ist

JavaScript ist eine dynamisch typisierte, als ECMAScript standardisierte Programmiersprache, die jeder Webbrowser ausführen kann. Über Laufzeitumgebungen wie Node.js läuft sie auch auf Servern und Build-Maschinen. Die Kurzform lautet JS. Die jährlichen Ausgaben des Standards heißen ES2015, ES2024 und so weiter; auch die ältere Nummerierung ES5, ES6 ist noch gebräuchlich. „Vanilla JavaScript“ bedeutet die Sprache ohne zusätzliches Framework. Außer dem Namen und einem Teil der Syntax mit geschweiften Klammern hat sie mit Java wenig gemeinsam.

Brendan Eich schrieb die erste Version bei Netscape. Sie wurde mit dem Browser Navigator 2.0 ausgeliefert. Ecma International verabschiedete die erste Ausgabe des Standards, ECMA-262, im Juni 1997. Seit ECMAScript 2016 veröffentlicht das zuständige Komitee TC39 jährlich eine neue Ausgabe. Ein Vorschlag geht erst in Stufe 4 des Verfahrens in die Spezifikation ein. Dafür sind bereits ausgelieferte Implementierungen erforderlich, sodass Text und Engines selten lange voneinander abweichen.

Die Sprache sagt sehr wenig über die Außenwelt. ECMA-262 definiert Syntax, Typen, Objekte, Promises und Module und überlässt Ein- und Ausgabe ihrer jeweiligen Hostumgebung. Im Browser ist das die Webplattform: Dokument, Ereignisse, Timer und Netzwerkanfragen, überwiegend im HTML-Standard und seinen benachbarten Standards spezifiziert. Node.js stellt stattdessen Dateien, Sockets und Prozesse bereit. Diese Trennung erklärt eine häufige Überraschung: Code kann völlig gültiges JavaScript sein und beim Umzug von einer Seite auf einen Server trotzdem scheitern, weil die Sprache mitkam, ihre Umgebung aber zurückblieb.

Im Browser teilen sich die Skripte einer Seite einen Hauptthread mit Layout, Zeichnen und Eingaben. Eine Aufgabe läuft bis zum Ende, bevor der Browser auf den nächsten Klick reagieren kann. Mit Promises und `async` -Funktionen blockiert das Warten diesen Thread nicht. Berechnungen bleiben dagegen darauf, sofern sie nicht in einen Web Worker verlagert werden. Zahlen sind IEEE-754-Werte mit doppelter Genauigkeit. Ganze Zahlen sind deshalb nur bis 2^53 – 1 exakt; größere benötigen `BigInt` .

Das Konzept für eine Entscheidung nutzen

Nutzen Sie „Was ist JavaScript?“ für eine konkrete Entscheidung. Wir können helfen, das Konzept mit dem System in Verbindung zu bringen, das Sie entwickeln oder einkaufen. Oberflächen, die schnell laden, auf dem Smartphone funktionieren und sich in allen Browsern Ihrer Kunden gleich verhalten.

Unsere Leistungen entdecken Kontakt aufnehmen →

Die Annahmen hinter dem Begriff prüfen

Nutzen Sie die Erklärung und die verlinkten Arbeiten, um Fragen für Ihr Projekt zu formulieren. Wir können die Annahmen gemeinsam mit Ihnen prüfen, bevor ein Werkzeug ausgewählt wird. Full-Stack-GIS-Kartenentwicklung geleitet — Full-Stack-Kartenentwicklung von der Datenbank bis zum Browser

Full-Stack-GIS-Kartenentwicklung geleitet Kontakt aufnehmen →

Warum es wichtig ist

Die eigentliche Entscheidung lautet, wie viel davon an die Leser Ihrer Website gesendet werden soll. JavaScript lässt eine Seite reagieren, ohne eine Anfrage an einen Server und dessen Antwort abzuwarten. Jedes Kilobyte wird auf einem Gerät heruntergeladen, geparst, kompiliert und ausgeführt, das Sie nicht kontrollieren. Das Bundle, das auf einem Entwicklerlaptop sofort reagiert, kann ein Mittelklasse-Smartphone sekundenlang beschäftigen. Der Leser bezahlt das mit Akkulaufzeit und Geduld.

Skriptcode ist auch die größte Angriffsfläche einer Seite. Wird das Skript eines Angreifers ausgeführt, läuft es mit den Berechtigungen der Seite: Es kann die Eingaben des Lesers mitlesen und sie überallhin senden, wohin die Seite senden darf. Dieses Risiko soll Content Security Policy begrenzen. Wird kein ausführbares JavaScript ausgeliefert, entfällt die Angriffsfläche der eigenen Browserskripte der Website. Nicht vertrauenswürdige Eingaben müssen trotzdem sicher behandelt werden; eine restriktive Richtlinie bleibt ebenfalls erforderlich. Das Paketökosystem schafft eine zweite Angriffsfläche. Ein typischer Frontend-Build zieht Hunderte transitive Abhängigkeiten nach sich. Jede davon ist Code, der mit denselben Rechten wie Ihr eigener läuft.

Dem stehen echte Vorteile gegenüber. Läuft auf beiden Seiten eines Systems dieselbe Sprache, können Browser und Server eine Validierungsregel oder Datenstruktur gemeinsam verwenden. Manche Produkte benötigen viel Skriptcode: Offlinezustand, Drag-and-drop, Karten und Zusammenarbeit in Echtzeit. Die sinnvolle Frage wird je Funktion gestellt: Muss das im Browser des Lesers geschehen, ohne einen Server zu fragen? Wenn nicht, rendern Sie es auf dem Server oder beim Build und liefern Sie HTML aus.

Die damit verbundene Wahl ist JavaScript oder TypeScript. TypeScript ergänzt eine Typprüfung und einen Build-Schritt und wird wieder zu reinem JavaScript kompiliert. Bei einer Codebasis, die mehrere Personen jede Woche ändern, lohnt sich die Prüfung gewöhnlich. Für eine Erweiterung mit hundert Zeilen auf einer statischen Seite reichen JavaScript und ein strenger Linter.

Wie es in der Praxis aussieht

Diese Website liefert ihren Lesern statische Seiten ohne ausführbares JavaScript. Ihr Suchcode bleibt erhalten, ist aber deaktiviert. JavaScript läuft weiterhin in den Build- und Prüfwerkzeugen. Drei Seiten zeigen die frühere Ausnahme im Browser und die Arbeit außerhalb des Browsers.

Eine frühere dreisprachige Version hatte kein Laufzeit-Backend und lieferte ein einziges eigenes Skript aus. Die Dokumentation des Hugo-Builds beschreibt diese begrenzte Ausnahme: einen Portfoliofilter, dessen Bedienelemente bis zur Skriptausführung verborgen blieben. Seine 108 Vorlagendateien, darunter 53 Partials und 18 Shortcodes, erledigten ihre Arbeit beim Bauen der Website. Der Bericht nennt auch den damaligen Preis dieser Wahl: Interaktive Funktionen mussten beim Build gelöst werden, und ein Suchindex wurde kalkuliert und zurückgestellt, statt ausgeliefert zu werden. Diese Zahlen und die Ausnahme beschreiben die frühere Implementierung, nicht den heute an Leser ausgelieferten Code.

Dieselben Inhalte werden über Gemini und Gopher veröffentlicht, zwei Protokolle ohne jeglichen Platz für ein Skript. Ein Build schrieb im dokumentierten Implementierungsstand je 777 Dokumente in beide Formate, aus derselben Quelle wie die Seiten. Ein gesonderter Build-Vergleich bestätigte, dass sich beim Hinzufügen dieser Ausgabeformate null HTML-Bytes änderten. Das Ergebnis erfasst keine späteren redaktionellen oder visuellen Änderungen.

Die Website vertritt den Standpunkt, dass ein Dokument zum Lesen kein Megabyte Skriptcode benötigt. Ein Format, das keinen aufnehmen kann, zeigt das am deutlichsten.

Das meiste JavaScript dieser Website steckt in den Werkzeugen. In dem vom verlinkten Projekt dokumentierten Implementierungsstand umfasste die verbindliche Qualitätsprüfung der Website zweiunddreißig Skripte und 10.242 Zeilen JavaScript, damals die größte Codemenge im Repository. Eine Zeit lang war dies der einzige Codebereich, den nichts formatierte, lintete oder auf Typen prüfte. Die Einführung von Formatierer und Linter begann damit, den Formatierer an den Code anzupassen: Die repositoryweite Voreinstellung von vier Leerzeichen hätte 2.173 Zeilen des größten Prüfers umgeschrieben. Eine auf diese Dateien begrenzte Einstellung mit zwei Leerzeichen ließ 38 Zeilen tatsächlicher Abweichungen übrig. Der Linter meldete anschließend 13 Befunde. Alle 13 wurden behoben, keiner unterdrückt. Ein Fehler in einem Prüfer zeigt sich als Prüfung, die besteht, obwohl sie scheitern müsste. Deshalb wird der Code der verbindlichen Qualitätsprüfung genauso behandelt wie alles, was er selbst prüft.

Wo es schiefgeht

Die folgenden Fehler ergeben sich aus dem Aufbau der Sprache. Für jeden gibt es eine bekannte Abhilfe.

Die Seite erscheint, reagiert dann aber eine Sekunde oder länger nicht auf das Antippen. Eine lange Aufgabe, etwa die Hydration eines großen Bundles, das Parsen einer großen JSON-Antwort oder das Sortieren Tausender Zeilen, blockiert den einzigen für Eingaben zuständigen Thread. Liefern Sie für die erste Interaktion weniger Code aus, indem Sie Bundles je Route aufteilen und den Rest zurückstellen. Zerlegen Sie lange Arbeit in Teile, die die Kontrolle an die Ereignisschleife zurückgeben, und verlagern Sie aufwendige Berechnungen in einen Web Worker. Messen Sie auf einem langsamen Smartphone statt auf dem Laptop, auf dem der Code geschrieben wurde.

Preise und Kennungen sind geringfügig falsch. Jede Zahl ist ein binärer Gleitkommawert. Dezimalbrüche werden deshalb gerundet und sehr große ganze Zahlen werden auf benachbarte Werte abgebildet. Das typische Opfer ist ein als JSON-Zahl gesendeter 64-Bit-Datenbankschlüssel:

```
0.1 + 0.2 === 0.3; // false
Number.MAX_SAFE_INTEGER; // 9007199254740991
9007199254740993 === 9007199254740992; // true
```

Speichern Sie Geldbeträge in Untereinheiten wie Cent als ganze Zahlen, senden Sie große Kennungen als Zeichenketten und verwenden Sie `BigInt`, wo exakt gerechnet werden muss.

Ein Vergleich lässt den falschen Wert durch. Der Operator `==` konvertiert seine Operanden vor dem Vergleich. Deshalb sind `"" == 0` und `"0" == false` beide wahr. Verwenden Sie durchgehend `===` und lassen Sie die Linterregel `eqeqeq` dies durchsetzen, damit der Code jede Konvertierung ausdrücklich anfordert.

Nutzertext wird als Code ausgeführt. Nicht vertrauenswürdigen Text an `innerHTML` zuzuweisen, Markup durch Zeichenkettenverkettung aufzubauen oder Eingaben an `eval` zu übergeben macht Daten zu Skriptcode. Schreiben Sie Text mit `textContent`, senden Sie eine Content Security Policy, die Skripte nur über Nonce oder Hash zulässt, und halten Sie die Abhängigkeitsliste kurz und in einer Lockdatei festgeschrieben. Lesen Sie, was jede Aktualisierung nachzieht.

Im Browser funktionierender Code stürzt auf dem Server ab. Die Sprache wurde verlagert, ihre Umgebung nicht: `window` und `document` existieren nur im Browser. Node.js liest eine Datei je nach Einstellungen, die der Browser nie sieht, als CommonJS oder als ES-Modul. Beschränken Sie laufzeitspezifische Aufrufe auf die Schnittstellen zur jeweiligen Umgebung, setzen Sie `"type": "module"` ausdrücklich in `package.json` und führen Sie jeden Codeteil in der Umgebung aus, in der er tatsächlich laufen wird.

Weiterführende Lektüre

- ECMAScript-Sprachspezifikation — Der fortlaufend aktualisierte Entwurf von ECMA-262, den jede Engine implementiert und der mit Vorschlägen aktualisiert wird, sobald sie Stufe 4 erreichen.
- Das TC39-Verfahren — Die Stufen, die ein Vorschlag vor seiner Aufnahme in die Sprache durchläuft, und die jeweiligen Anforderungen.
- ECMA-262 bei Ecma International — Die veröffentlichten Jahresausgaben für Projekte, die ihre Zielversion benennen müssen.

- HTML-Standard: Ereignisschleifen — Wie ein Browser Aufgaben, Mikroaufgaben und Rendering rund um die Skripte einer Seite plant.
- HTML-Standard: das script-Element — Klassische Skripte und Module sowie der Einfluss von `async` und `defer` auf ihren Ausführungszeitpunkt.

Glossar

Für Ihre tatsächlichen Rahmenbedingungen auswählen

Beschreiben Sie den Arbeitsablauf, die vorhandenen Schnittstellen und die Nutzer, die die Veränderung benötigen. Wir können eine prüfbare Lieferung definieren und die Übergabe so gestalten, dass Ihr Team die gelieferten Bestandteile warten kann.

Preise ansehen Kontakt aufnehmen →

Die Begriffe praktisch nutzbar machen

Fragen Sie, wie „Was ist JavaScript?“ auf Ihre Organisation zutrifft. Ein konkretes Beispiel aus Ihrem bestehenden System reicht aus, um das Gespräch zu beginnen.

Unsere Leistungen entdecken

welcome@engineer.company +45 50 10 61 84 Kopenhagen, Dänemark

Nehmen Sie diese Seite mit

Kontaktkarte

Mit Sorgfalt und Liebe gestaltet von D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Copyright © 2025 – heute · Engineer ApS · Alle Rechte vorbehalten.

Gedruckt von https://engineer.company/de/glossary/javascript/