Die große Vue-Serie - Teil 8

Dem Fehler auf der Spur

von Marc Teufel

27 Feb. 2024

Auch in dieser Folge haben wir wieder ein spannendes Thema für Sie vorbereitet: Wir schauen uns an, welche Möglichkeiten wir bei der Fehlersuche und beim Debugging von Vue-Anwendungen haben. Im Vue-Umfeld gibt es da eine ganze Menge: Vom einfachen *console*-Objekt über die Vue Devtools hin zum Debugging in WebStorm oder Visual Studio Code streifen wir alle Möglichkeiten.

Während der Entwicklung Ihrer Vue-Applikation werden Sie viel Zeit mit Testen und der Fehlersuche verbringen. Gerade Letzteres kann in bestimmten Situationen zu einer richtigen Herausforderung werden, aber da erzähle ich Ihnen sicher nichts Neues. Im Umfeld von Webanwendungen wird die ganze Angelegenheit nochmal komplizierter, denn es gibt eine Vielzahl an Möglichkeiten, wie Sie Ihren Programmfluss überprüfen und auf Fehlersuche gehen können. Lassen Sie uns das der Reihe nach aufarbeiten.

Fehlersuche im Browser

Erster Anlaufpunkt ist der Webbrowser. Diesen verwenden Sie nicht nur zur Anzeige Ihrer Anwendung, sondern Sie binden diesen auch aktiv in den Entwicklungsprozess mit ein. Moderne Browser verfügen neben der Möglichkeit, Webseiten anzuzeigen, auch über eine Vielzahl von Werkzeugen zur Analyse und zur Fehlersuche. Vor allem in den beiden Platzhirschen Chrome und Firefox sind vollständige Entwicklungsumgebungen integriert, mit denen Sie Webseiten auseinandernehmen und untersuchen können.

console

Bei Entwickler:innen sehr beliebt ist sicherlich die Konsole in den Entwicklungswerkzeugen und die Möglichkeit, von JavaScript heraus mit dem *console*-Objekt Ausgaben zu schreiben. Der wahrscheinlich beliebteste Befehl dürfte in diesem Zusammenhang *console.log* sein. Doch die Möglichkeiten, die Ihnen das *console*-Objekt bietet, gehen weit über einfaches Logging hinaus. Machen Sie ein Experiment: Öffnen Sie einen Browser und darin die Konsole in den Entwicklungstools. Geben Sie den Befehl *console.log(console)* ein. **Abbildung 1** zeigt die Ausgabe. Sie schreiben hiermit das Objekt *console* auf die Konsole und lernen damit gleich zwei Dinge: Sie können mit *console.log* nicht nur Strings loggen, sondern ganze Objekte, um diese dann zu analysieren. Außerdem haben Sie einen Trick kennengelernt, um sich einen Überblick zu verschaffen, wie leistungsfähig das *console*-Objekt ist und was es alles kann.

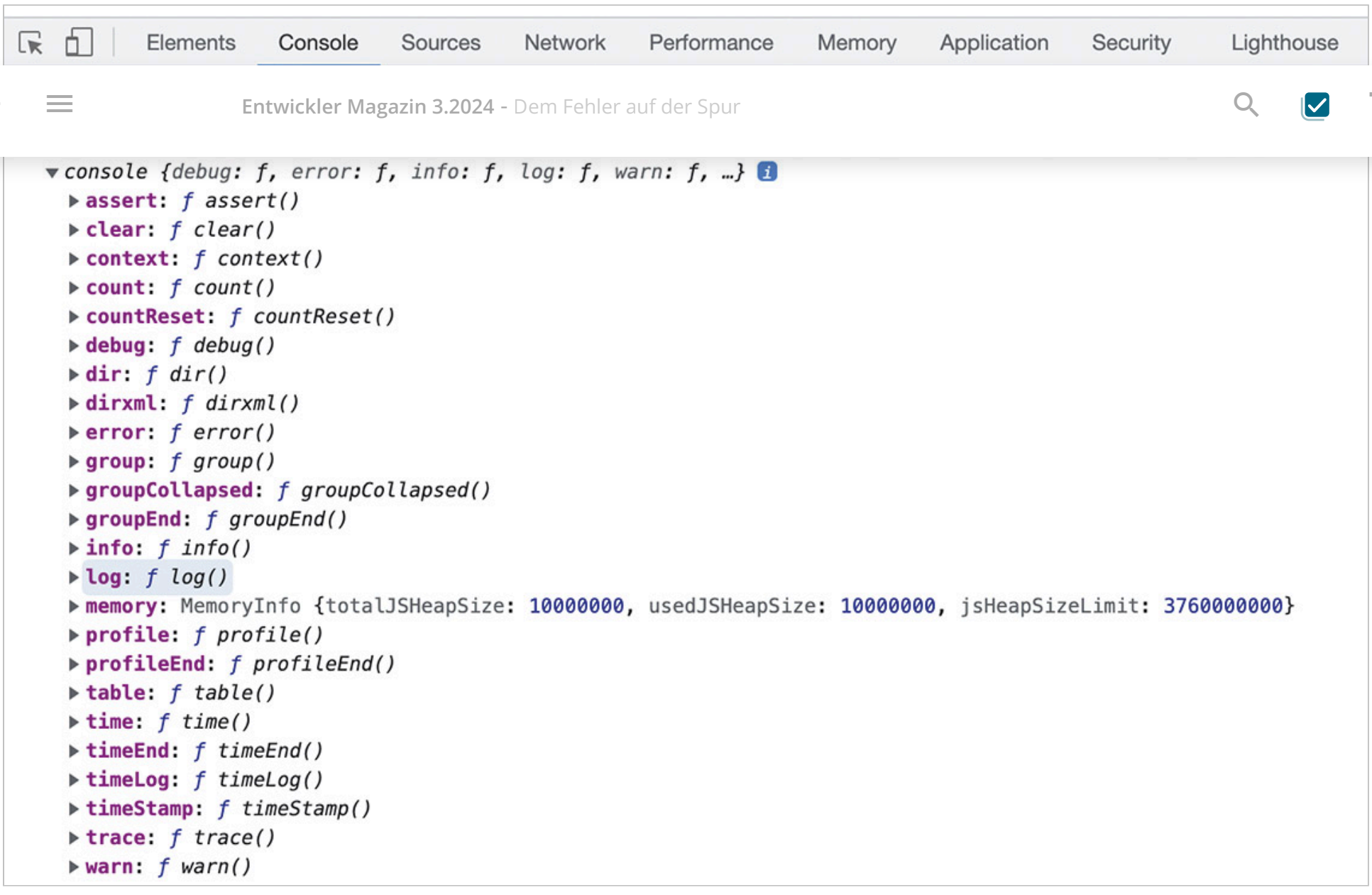

Abb. 1: Das console-Objekt bietet weit mehr als console.log

Leider kann ich an dieser Stelle nicht auf jede der in der Abbildung dargestellten Methoden im Detail eingehen. Ich hoffe aber, dass ich Ihr Interesse geweckt habe, sich das *console*-Objekt bei Gelegenheit nochmal im Detail anzuschauen. Es lohnt sich wirklich. Im Internet und vor allem auf YouTube gibt es unter dem Stichwort „debug more than console.log“ einige interessante und kurzweilige Videos, die jede einzelne dieser Methoden im Detail zeigen.

Debugging im Browser

In den Entwicklungswerkzeugen vor allem der Browser Chrome und Firefox ist außerdem ein sehr leistungsfähiger Debugger enthalten. Das bedeutet, Sie können Haltepunkte setzen, die Anwendung an dieser Stelle stoppen und sich dann schrittweise durch das laufende Programm bewegen. Das spielt alles im Übrigen auch sehr gut mit Vite zusammen. Wir haben in der letzten Episode bereits gelernt, dass der Vite-DevServer die Anwendung unter Berücksichtigung von ESM ausliefert. Das bedeutet, wir können beispielsweise eine SFC als ausführbaren JavaScript-Code in den geöffneten Entwicklungswerkzeugen im Sources-Tab lokalisieren und dann einen Haltepunkt setzen. In **Abbildung 2** sehen Sie die Ansicht des Chrome-Browsers, der an einem Breakpoint in der Applikation angehalten hat.

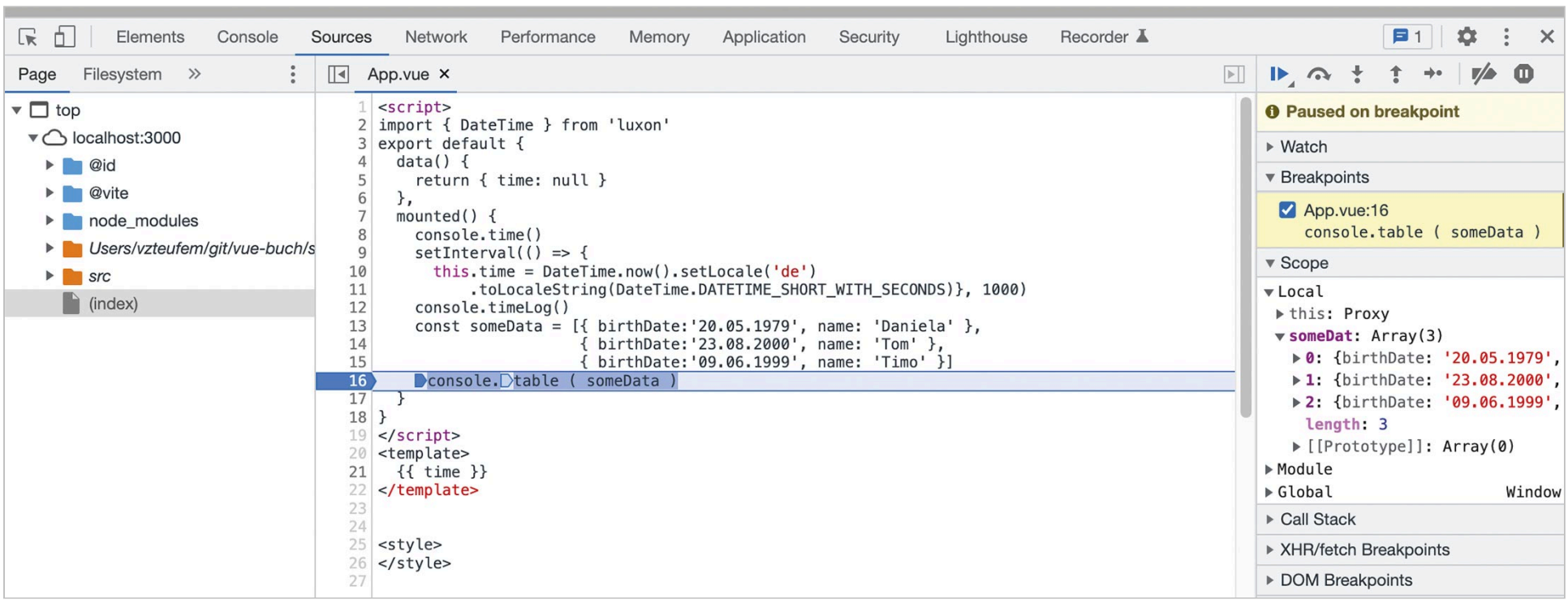

Abb. 2: Debugging im Browser

Ein Rechtsklick auf die Zeilennummer bietet Ihnen die Möglichkeit, einen Breakpoint zu setzen. Sie können Breakpoints auch mit Bedingungen verknüpfen. Alternativ haben Sie die Wahl, den Debugger auch mit dem JavaScript-Befehl *debugger* zu aktivieren. Haben Sie diesen in Ihrer Applikation verbaut und sind die Entwicklungswerkzeuge geöffnet, wird die Ausführung an dieser Stelle angehalten. In **Abbildung 2** sehen Sie im rechten oberen Teil der Entwicklerwerkzeuge eine Symbolleiste. Mit dieser können Sie den Debugger steuern. Dabei funktioniert alles so, wie Sie es von den gängigen Entwicklungsumgebungen her kennen: Sie können sich schrittweise durch den Quellcode bewegen (Step, Step into), aus Funktionen herausspringen (Step out) oder bestimmte Programmteile überspringen (Step over).

Wenn der Debugger an einem Breakpoint angehalten hat, erhalten Sie auch zusätzliche Informationen und Kontext über den Zustand Ihrer Applikation. So können Sie sich die aktuellen Variablen und ihre Werte ansehen und den Call-Stack analysieren. Der Call-Stack ist bei der Fehlersuche besonders interessant, weil er Ihnen genau auflistet, welche Methoden bis zum Haltepunkt seit dem Start der Ausführung nacheinander aufgerufen wurden. Analog zum Call-Stack können Sie in Ihrer Applikation auch den Befehl *console.trace()* einbauen, dann wird der Call-Stack bis zu diesem Zeitpunkt in der Konsole der Entwicklertools ausgegeben. Um bei hartnäckigen und schwer greifbaren Fehlern zu verstehen, wie der Programmfluss war, und um Rückschlüsse zu ziehen, ist die Analyse des Call-Stacks ein unentbehrliches und wichtiges Mittel in Ihrem Werkzeugkasten.

Vue-Devtools

Der Debugger ist aber bei weitem nicht das einzige Hilfsmittel, auf das Sie bei der Entwicklung zurückgreifen können. Sie werden es kaum glauben: Der Wunsch nach den Vue Devtools, die ich Ihnen nun vorstellen möchte, kommt direkt aus der Mitte der Vue-Entwickler:innen selbst. Und tatsächlich, mit den Vue Devtools wird die Entwicklung von Applikationen nochmal spannender. Hierbei handelt es sich um einen Satz von Werkzeugen, mit denen Sie eine Vue-Applikation zur Laufzeit untersuchen können.

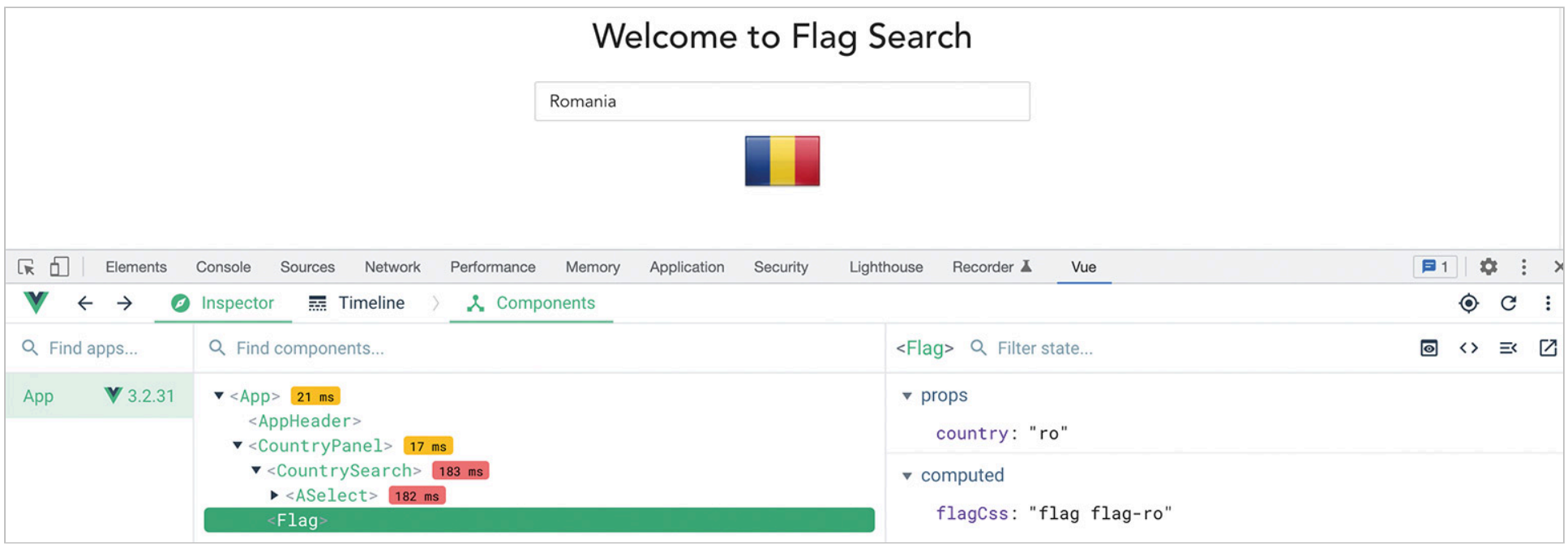

Abb. 3: Die Vue-Devtools in Chrome

In den Vue-Devtools wird, wie **Abbildung 3** zeigt, die Komponentenstruktur einer Applikation hierarchisch dargestellt. Man kann sich durch die Struktur klicken und sieht auch hier den tatsächlichen (vom Vue-Compiler erzeugten) ausgeführten Code. Zusätzlich zum Zustand der einzelnen Komponenten wird auch die Zeit angezeigt, die für das Rendern gebraucht wurde. Die Komponenten können überwacht und ihr Zustand sogar teilweise zur Laufzeit modifiziert werden. Auf diese Weise lässt sich bestimmtes Laufzeitverhalten besser untersuchen. Die Timeline, eine Art Zeitmaschine, vervollständigt den Werkzeugkasten.

Dieser Teil der Devtools ist wirklich beeindruckend. Was hier im Kern nämlich passiert, ist nichts anderes als eine Komplettaufzeichnung sämtlicher Aktionen, Zustände und Ereignisse der laufenden Applikation. Auf dem Graphen kann man sich anschließend am Zeitstrahl entlang durch die Applikation bewegen und schauen, wie der Zustand zu einem bestimmen Zeitpunkt war. Die Informationen werden dabei in die vier Gruppen Mouse, Keyboard, Component events und Performance unterteilt. Neben dem Graphen werden die Informationen auch in Listenform anzeigt, immer chronologisch seit Start der Applikation sortiert. Alle Aktivitäten in der Anwendung inklusive des Zustands zum jeweiligen Zeitpunkt können auf diese Weise einfach nachvollzogen werden.

Vue Devtools im Browser

Sie können die Devtools entweder als Browsererweiterung in Chrome oder Firefox installieren oder Sie betreiben sie einfach stand-alone. Die Installation der Erweiterung für den jeweiligen Browser geht einfach von der Hand. Unter [1] wird Ihnen erklärt, wie Sie die Erweiterung für Ihren Browser beziehen und installieren. Wichtige Zusatzinformationen, die bei der Installation gegebenenfalls zu beachten sind, finden sich ebenfalls hier. Nach der Installation erscheint in der Toolbar ein kleines Icon. Dieses zeigt Ihnen visuell an, sobald ein Vue-Framework entdeckt wird. Wird Vue erkannt, finden Sie in den Entwicklungswerkzeugen einen zusätzlichen Tab mit der Bezeichnung Vue vor.

Vue Devtools als Stand-alone-Lösung betreiben

Sie können die Devtools auch unabhängig vom Browser in einem eigenen Anwendungsfenster betreiben. Das kann sinnvoll sein, wenn Sie mit einem „exotischen“ Browser oder in einer Umgebung arbeiten, in der Sie keinen Zugriff auf die Entwicklerwerkzeuge von Chrome oder Firefox haben. Wenn Ihnen ein eigenes Fenster lieber ist als ein zusätzlicher Reiter, in den ohnehin schon überladenen Entwicklerwerkzeugen des Browsers, sind Sie mit der Stand-alone-Lösung ebenfalls gut beraten. Die Installation und der Start gestalten sich einfach. Führen Sie (vorausgesetzt, Node ist auf Ihrem System installiert) einfach das Kommando *npx @vue/devtools* aus. Die Anwendung startet sofort und teilt mit, dass es auf eine Verbindung mit einer Vue-Applikation wartet. Die Vue Devtools kommunizieren in der Standardeinstellung über den Port *8098*. Um die Verbindung zwischen Ihrer Applikation und den Devtools nun herzustellen, muss lediglich ein kleines Stückchen Code aus den Devtools in Ihre Applikation eingeschleust werden. Das gelingt, in dem Sie in die *index.html* folgendes Fragment zusätzlich aufnehmen:

```
<script src="http://localhost:8098"></script>
```

Vorausgesetzt, Ihr DevServer läuft noch, führt das dazu, dass von der angegebenen Seite das besagte Stückchen Code geladen und eingebunden wird. Danach steht die Verbindung zwischen Ihrer Anwendung und den Devtools.

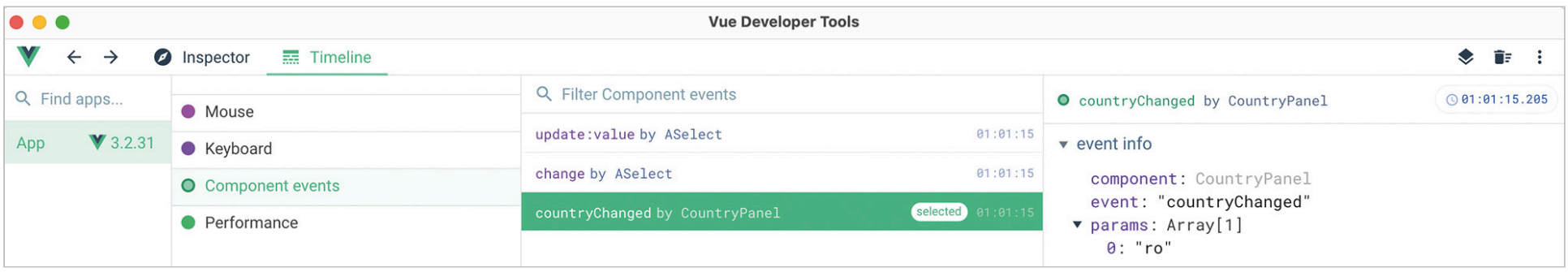

Abb. 4: Die Time der Vue-Devtools in der Stand-alone-Version

Schauen Sie sich **Abbildung 4** in diesem Zusammenhang bitte auch nochmal etwas genauer an. In der hier überwachten Vue-Applikation sind seit dem Start insgesamt drei Ereignisse (*Events*) von den Komponenten ausgegangen. Zuerst wurde nach Ländern gesucht (*update:value*). Dann wurde auch ein Land gefunden (*change*) und damit ein benutzerdefiniertes Ereignis für das gefundene Land ausgelöst (*countryChanged*). Darauf folgten weitere Aktionen.

Ich glaube, bis hierher sind schon viele interessante Informationen für eine spannende Fehlersuche geliefert worden. Die Königsdisziplin, das Debugging aus einer Entwicklungsumgebung heraus, fehlt aber noch. Gehen wir es also an.

Debugging mit WebStorm

Das Debugging direkt aus der Entwicklungsumgebung heraus hat den großen Vorteil, dass Sie im Quellcode arbeiten und dort auch Breakpoints setzen können. WebStorm bietet Ihnen verschiedene Möglichkeiten, wie Sie sich zum Debuggen mit Ihrer Applikation verbinden können. Ich zeige Ihnen hier, wie Sie sich mit WebStorm in eine laufende Applikation im Browser einklinken können.

Öffnen Sie hierzu zunächst Ihr Projekt in WebStorm und starten Sie den Vite DevServer mit *npm run dev* zum Beispiel aus dem WebStorm-Terminalfenster heraus. Das Terminal können Sie direkt innerhalb von WebStorm öffnen, indem Sie die SHIFT-Taste zweimal hintereinander drücken und dann als Suchbegriff einfach „Terminal“ eingeben. Mit dieser einfachen Suche kommen Sie sehr bequem an nahezu jede Funktion von WebStorm heran. Um den Debugger in Gang zu setzen, benötigen wir eine Debug-Konfiguration. Den zugehörigen Konfigurationsdialog finden wir auch hier sehr leicht über die gerade angesprochene globale Suche. WebStorm bietet eine Fülle verschiedener Möglichkeiten, wie eine Debug-Konfiguration angelegt werden kann. Sie wählen als Typ JavaScript Debug aus und erhalten damit einen Dialog (**Abb. 5**).

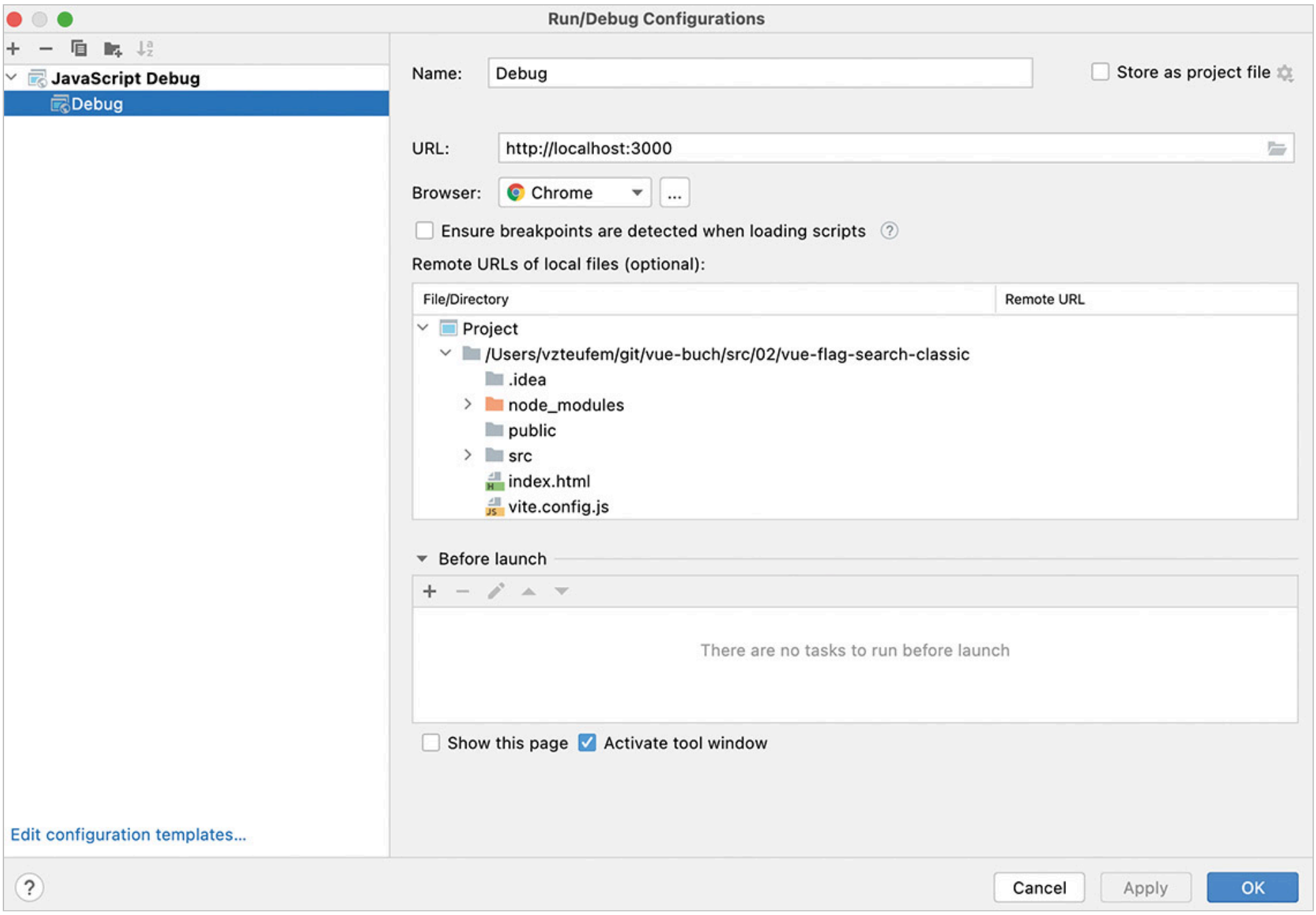

Abb. 5: Eine Debug-Konfiguration in WebStorm einstellen

Hier legen Sie einen Browser fest, den WebStorm beim Öffnen einer neuen Debug-Sitzung starten und einbinden soll. Sie stellen außerdem den Einstiegs-URL Ihrer Applikation ein. Damit das ganze Debugging auch funktioniert, müssen Sie sicherstellen, dass der Vite DevServer läuft. Wenn Sie den Debugger über die Debug-Konfiguration starten (Käfersymbol), werden der Browser und Ihre Applikation darin geöffnet und Sie können sie ganz normal bedienen. Sollte WebStorm im Programmfluss nun auf einen Breakpoint stoßen, hält die Entwicklungsumgebung erwartungsgemäß an.

Debugging mit Visual Studio Code und Volar

Wenn Sie Visual Studio Code installieren, bekommen Sie einen leistungsfähigen JavaScript-Debugger, der sich mit Chrome oder Firefox verbinden kann, frei Haus mitgeliefert. Neben dem Volar-Plug-in brauchen Sie prinzipiell keine weitere zusätzliche Software, um erfolgreich Vue-Applikationen zu entwickeln und zu debuggen. Um den Debugger aus VS Code zu starten, müssen Sie zunächst ebenfalls sicher gehen, dass der Vite DevServer (in diesem Beispiel auf Port *3000*) läuft.

Als Nächstes legen Sie sich eine Launch-Konfiguration an. Diese wird von VS Code automatisch im Projekt im Ordner *.vscode/launch.json* angelegt, wenn Sie in der Toolbar links auf das Debug-Icon klicken. Folgen Sie den Anweisungen von VS Code, wird die *launch.json* entsprechend angelegt und im Editor geöffnet. Sie müssen jetzt nur noch den Einstiegspunkt in Ihre Applikation in der Einstellung URL eintragen.

Listing 1: Launch-Konfiguration aus Visual Studio Code, um eine Vue-Applikation zu debuggen

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "type": "pwa-chrome",
      "request": "launch",
      "name": "Launch Chrome against localhost",
      "url": "http://localhost:3000",
      "webRoot": "${workspaceFolder}"
    }
  ]
}
```

Listing 1 zeigt, wie eine *launch.json* für eine Vue-Applikation aussehen kann, die vom Vite DevServer auf Port *3000* bereitgestellt wird. Jetzt kann der Debugger mit dem grünen Play-Knopf oben rechts (LAUNCH CHROME) gestartet werden. In **Abbildung 6** sehen Sie den Debugger in Aktion. Er steht dem Debugger in WebStorm in nichts nach. Alle benötigten Informationen, Variablen, Zustände und sogar die Konsolenausgabe (*console.log*) werden übersichtlich dargestellt. Mit den Navigationsknöpfen, die Sie oben in der Mitte erkennen können, bewegen Sie sich Schritt für Schritt oder Funktion für Funktion durch das Programm.

Ich möchte Sie an dieser Stelle ermutigen, sowohl WebStorm als auch Visual Studio Code auszuprobieren. Beide liefern nahezu die gleiche Funktionalität. WebStorm hat durch seine übersichtliche Oberfläche, die durchdachte Bedienführung und die gute Integration der einzelnen Werkzeuge die Nase vielleicht ein Stück weit vorn. Am Ende ist es jedoch immer eine individuelle Entscheidung und persönliche Vorliebe, welches Werkzeug man bevorzugt.

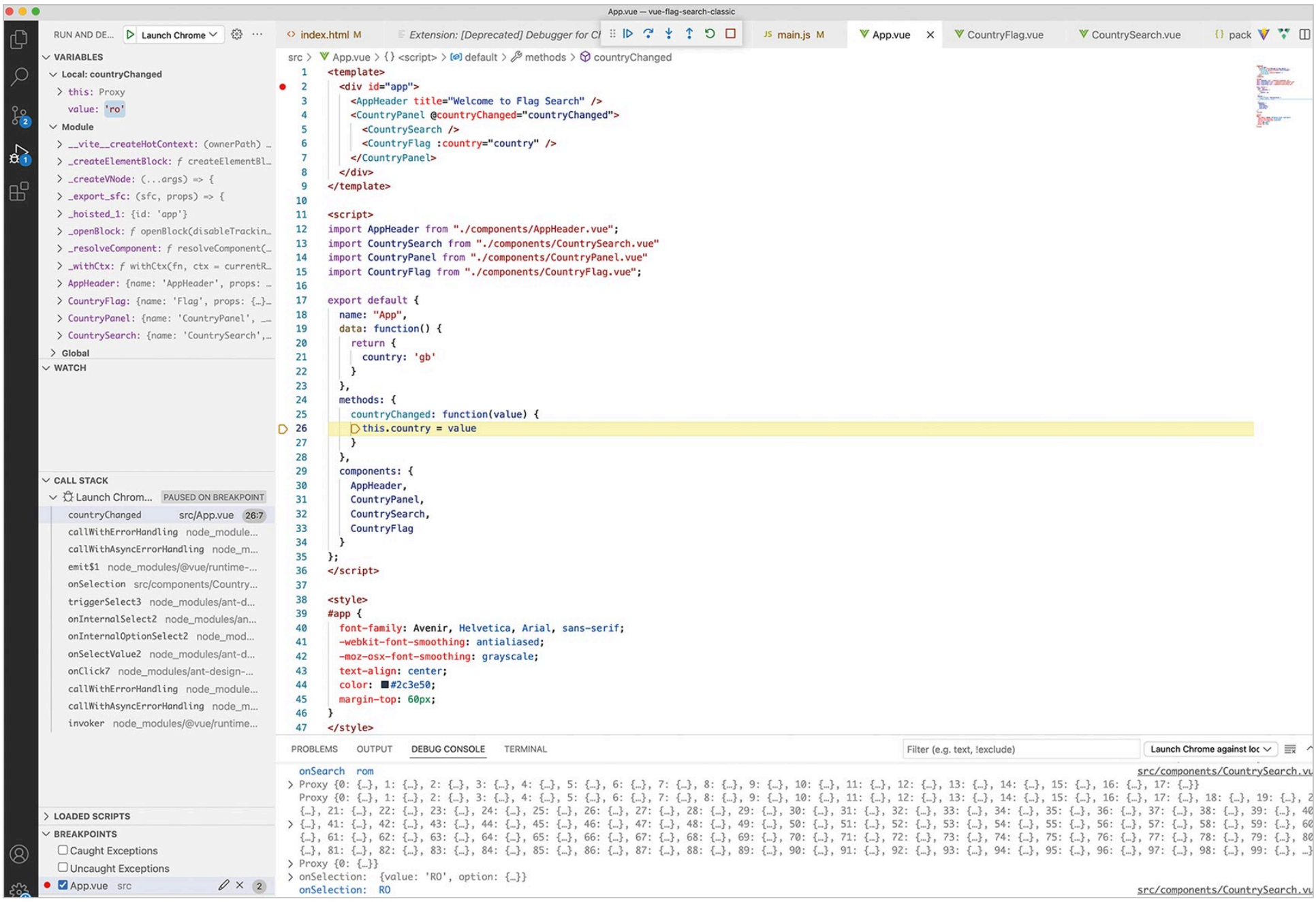

Abb. 6: Debugging-Session in Visual Studio Code

Applikation bauen

Nachdem wir jetzt alle Möglichkeiten gestreift haben, wie wir während der Entwicklung auf Fehlersuche gehen können, möchte ich diese Folge mit einer kleinen Zugabe abschließen. Ich möchte Ihnen zeigen, wie man eine Applikation letztendlich baut und auch ausliefert. Wir halten nochmal fest: Während der Entwicklung liefert der Vite DevServer Module auf Basis von ESM aus und beglückt uns auf diese Weise mit einem performanten und angenehmen Entwicklungserlebnis. Der Code, der dabei entsteht, ist aber noch lange nicht der Weisheit letzter Schluss. Er ist weder für ältere Browser optimiert noch spielen Obfuscation und Minification eine Rolle zur Entwicklungszeit. Um Ihre Applikation für den Produktivbetrieb vorzubereiten, führen Sie den Befehl *npm run build* aus.

In diesem Fall bricht Vite, wie bereits erwähnt, mit der „ESM über alles“-Regel, und benutzt letzten Endes den klassischen Bundler Rollup, um die Produktionsversion Ihrer Applikation zu bauen. Das ist auch nicht weiter schlimm, vielmehr ist es ein guter Kompromiss: Während der Entwicklung können Sie von den Vorteilen von ESM profitieren, und hinten raus, wenn es in die Produktion geht, bleiben Sie weiterhin in der Lage, hochoptimierte Bundles zu erhalten.

Nachdem der Build gelaufen ist, finden Sie Ihre Applikation im Ordner *dist* Ihres Projektverzeichnisses wieder. Um zu testen, ob die Anwendung auch funktioniert, kopieren Sie alle Dateien aus diesem Ordner auf einen Webserver. Alternativ starten Sie aus dem *dist*-Verzeichnis heraus einen Webserverprozess mit npm. In früheren Episoden dieser Artikelserie habe ich Ihnen bereits gezeigt, wie das zum Beispiel mit Servor gelingen kann.

Um den Build zu steuern, stehen Ihnen unterschiedliche Konfigurationsoptionen in der *vite.config.js* zur Verfügung. Alle diese Parameter beginnen mit dem Prefix *build*. So haben Sie mit dem Parameter *build.lib* Einfluss darauf, ob Ihre Applikation

alternativ als Bibliothek ausgeliefert werden soll. Das kann dann interessant sein, wenn Sie zum Beispiel nur eine Komponente oder einen Satz von Komponenten veröffentlichen wollen und keine vollständige Anwendung. Sie können hier zwischen den beiden wichtigen Parametern *es* oder *umd* wählen. Damit bestimmen Sie, für welches Modulsystem die Bibliothek optimiert werden soll. Die verschiedenen Modulsysteme haben wir ebenfalls bereits ausführlich besprochen.

Beachten Sie bitte, dass mit dem Parameter *es* das JavaScript-Modulsystem, also ESM, gemeint ist. Eine Bibliothek in diesem Format kann demnach über den JavaScript-Befehl *import* eingebunden werden, während das UMD-Format als Fallback gedacht ist. Sie können UMD überall dort einsetzen, wo ESM nicht verfügbar ist. Ich empfehle Ihnen, wenn möglich immer mit ESM zu arbeiten.

Minification und Obfuscation

Minification bedeutet, die Größe des Quellcodes so weit wie möglich zu reduzieren. Ziel dabei ist, die Applikation schneller zu machen. Eine kleinere Codebasis bedeutet auch einen kleineren Download. Außerdem kann der JavaScript-Parser die Applikation viel schneller durcharbeiten. Zu guter Letzt ist in dieser Form bearbeiteter Code auch schon gut vor neugierigen Blicken in den Quellcode geschützt, da dieser bereits schwer zu lesen und zu verstehen ist.

Obfuscation geht einen Schritt weiter und macht den Code noch kryptischer und noch schwerer verständlich. Das wird vor allem erreicht, indem die Bezeichnungen von Variablen, Objekten und Funktionen durch sehr kryptische Namen ausgetauscht werden.

Minification und Obfuscation sind bereits enthalten, wenn Sie einen Build aus Vite heraus starten. Diese Funktionalität implementiert Vite nicht selbst, sondern delegiert die Arbeit an die externen Werkzeuge *esbuild* oder *terser*. Wenn Sie alles in der Standardeinstellung belassen, wird *esbuild* verwendet. Um das zu beeinflussen oder die Minification sogar ganz abzuschalten, benutzen Sie die Option *build.minify*. Listing 2 zeigt ein Beispiel.

Listing 2: Minification deaktivieren

```
export default defineConfig({
  plugins: [vue()],
  // hier eventuell noch andere Konfiguration
  build: {
    minify: false
  }
})
```

Wenn Sie auf Terser umschalten, stehen Ihnen deutlich mehr Einstellungsmöglichkeiten für die Minimierung und für den Verschleierungsprozess (in Terser wird dieser als „Mangling“ bezeichnet) zur Verfügung. Listing 3 zeigt, wie es geht. Beachten Sie bitte, dass die meisten sinnvollen Terser-Optionen oft schon im Standard aktiviert sind. Ein Blick in die umfangreiche Terser-Anleitung lohnt sich in diesem Zusammenhang.

Listing 3: Terser für Minification und Obfuscation einsetzen

```
export default defineConfig({
  plugins: [vue()],
  // hier eventuell noch andere Konfiguration
  build: {
    minify: 'terser',
    terserOptions: {
      compress: {
        dead_code: true,     // default true
        drop_debugger: true, // default true, entfernt Debugger Statements
        drop_console: true   // default false, entfernt Console statements
      },
      mangle: {
        keep_fnames: true    // default false
      }
    }
  }
})
```

Ausblick

Mit diesem achten Teil unserer Vue-Artikelserie schließen wir den Überblick über die Entwicklung von Vue-Anwendungen ab. In der nächsten Ausgabe gehen wir wieder ganz zurück an den Anfang. Wir beschäftigen uns mit dem Komponentenbau und der Frage, was Vue unter einer Komponente versteht, wie diese im Detail aufgebaut sind und welche Möglichkeiten wir haben, Komponenten zu entwickeln.

Links & Literatur

[1] https://devtools.vuejs.org/guide/installation.html