5 Das (neue) Reactivity-System

Mit *Reactivity* wird ein Kernkonzept zusammengefasst, das den Unterschied von modernen Web-Frameworks zu reinem JavaScript oder Bibliotheken wie jQuery verdeutlicht. Es ist der Unterschied zwischen deklarativem und imperativem Rendering. In Vue sind die Template-Syntax und das virtuelle DOM zwei Kernaspekte davon. In jQuery, um bei dem Beispiel zu bleiben, müssen Änderungen an der Oberfläche ganz explizit gesetzt werden. Eine Funktion ändert direkt einen Wert im DOM, fügt eine CSS-Klasse hinzu oder löscht ein Kindelement. Das nachfolgende Code-Snippet zeigt das an einem Beispiel, bei dem eine CSS-Klasse zu DOM-Elementen hinzugefügt wird, wenn ein Button-Klick erfolgt.

```
$('button').click(function() {
  $('h1, p').addClass('important');
});
```

Das sind alles direkte, imperative Anweisungen in Form von Kommandos, die das *Wie* darstellen. Bei der deklarativen Sichtweise moderner Web-Frameworks werden Daten verändert, die einen Zustand wiedergeben. Durch diese Zustandsänderungen verändern sich beispielsweise Resultate von Berechnungen, die Inhalte oder die Anzahl von Listenelementen oder sonstige Eigenschaften von View-Elementen. Auf Basis dieser Datenänderungen wird anschließend der View Layer angepasst. Wir Entwicklerinnen und Entwickler haben damit deklarativ bestimmt, *was* geschehen soll. Das *Wie* interessiert uns an dieser Stelle zunächst nicht.

Zusammengefasst bedeutet Reactivity, dass sich das Resultat einer Berechnung, wie auch immer diese ausgestaltet ist, ändert, wenn sich die zugrunde liegenden Daten ändern. Die veränderten Daten lösen somit einen gewollten Seiteneffekt aus. Das ist immer dann sinnvoll, wenn Daten, Informationen oder eben Resultate von Berechnungen in der Benutzeroberfläche angezeigt werden: zum Beispiel bei einer HTML-Seite in einer Webanwendung. Anders formuliert bietet das Reactivity-System einen Mechanismus, die Daten im Modell automatisch synchron mit der Datenrepräsentation, dem View-Layer, zu halten. Änderungen an den Daten werden automatisch in der View wiedergegeben, weil entsprechender Code von Vue ausgeführt wird. Das *Was* (semantisch) ist der bestimmende Faktor, nicht das *Wie* (technisch). Die Abbildung 5–1 visualisiert diesen Unterschied schematisch.

Die Reaktivität von Daten ist ein fundamentales Feature moderner Web-Frameworks. JavaScript als Programmiersprache ist im Standard nicht reaktiv, wie das nachfolgende Beispiel verdeutlicht:

```
let x = 3;
let y = 5;
let result = x + y;

console.log(result); // 8

y = 3;

console.log(result); // Weiterhin 8
```

Die Ausgabe auf der Konsole ist hier vergleichbar mit einem View-Layer, also zum Beispiel der Anzeige in einer HTML-Seite. In modernen Webanwendungen muss sich das Resultat der Berechnung verändern, wenn sich die zugrunde liegenden Daten ändern, was aber hier nicht passiert. Eine einmal im DOM gerenderte Information, zum Beispiel durch eine Variable, ändert sich nicht auf magische Weise, wenn sich der Inhalt der Variablen ändert. Genau das ist aber eine primäre Anforderung moderner, datengetriebener Webanwendungen. Dieses *Zustandsmanagement* (*State Management*) manuell zu implementieren und die Daten mit der View synchron zu halten, ist zwar möglich, bedeutet aber einen erheblichen Aufwand, was Zeit, Nerven und Geld kostet sowie zusätzlich fehleranfällig ist.

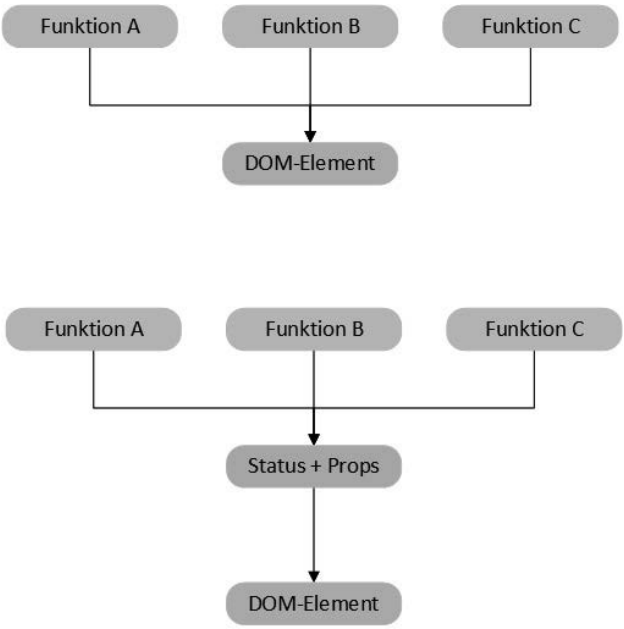

Abb. 5–1 *Schematische Darstellung eines imperativen (oben) und deklarativen (unten) Änderungsprozesses von Funktionen an einem DOM-Element. Im deklarativen Fall wird das Element indirekt über Änderungen an den Daten verändert.*

5.1 Der Ansatz von Vue

Das Reactivity-System von Vue ist eines die zentralen Merkmale des Web-Frameworks. Ein Reactivity-System muss primär drei Aufgaben erfüllen, damit es einen Nutzen entfaltet:

1. Es muss erfassen (tracken), wenn ein Wert gelesen wird. Damit wird ermittelt, welcher Code welche Abhängigkeiten auf Variablen und damit auf Daten hat. Beim Ändern von Daten müssen diese Codeteile erneut ausgeführt werden, damit Änderungen an den Daten über die reine Änderung der Variablen hinweg wirksam werden.
2. Das System muss erfassen, wann Werte verändert werden. Denn das ist der Ausgangspunkt für reaktive Veränderungen in anderen Bereichen der Anwendung.
3. Es ist notwendig, dass der Code neu ausgeführt wird, der Daten einer Variablen gelesen hat, die anschließend verändert wurde. Das ist der Punkt, in dem sich andere Codeteile einer Anwendung, zum Beispiel Elemente im View-Layer, anpassen und das Reactivity-System auch für uns Entwicklerinnen und Entwickler sowie für die Nutzerinnen und Nutzer sichtbar wird.

Das Reactivity-Modell von Vue wird gerne als unaufdringlich oder unauffällig bezeichnet. Das bedeutet, es fällt bei der täglichen Entwicklung nicht auf, dass es überhaupt da ist. Es funktioniert einfach. Das Modell einer Vue-Instanz, der Datenbereich, besteht aus einfachen JavaScript-Objekten. Die Eigenschaften dieser Objekte enthalten die Daten, und Änderungen daran werden automatisch erfasst, damit Vue das View anpassen kann. Das bedeutet aber auch, dass alle Daten, die reaktiv sein sollen, im Datenbereich einer Vue-Instanz oder einer Komponente definiert werden müssen. Fehlt dort zum Beispiel eine Variable, die dann später mit der Textinterpolation gerendert werden soll, warnt Vue, dass die Variable unbekannt ist. Ohne die vorherige Definition weiß Vue nichts von der Variablen und konnte diese nicht ins Reactivity-System einbinden und mit den notwendigen Anpassungen ausstatten. Wie die dafür nötigen Anpassungen aussehen, beschreibt der nachfolgende Abschnitt 5.2 im Detail auch mit zusätzlichen Informationen für alle, die das Reactivity-System aus Vue 2 kennen. Denn zwischen beiden Versionen bestehen erhebliche Unterschiede. Dieses automatische Reactivity-System hat einige Vorteile. Es spart Zeit bei der Entwicklung, der Code wird einfacher und es reduziert unseren kognitiven Aufwand.

Um das Ganze an einem Beispiel festzumachen: In Vue reicht es aus, einen Datenbereich zu erstellen, dort Eigenschaften zu definieren und diese im HTML-Template zu nutzen. Ändern sich die Daten, ändert sich auch die Anzeige im HTML-Template. Das folgende Beispiel zeigt einen Ausschnitt aus der Definition eines Datenbereichs und einer Methode sowie das zugehörige Template.

```
data() {
  return {
    message: 'Ich bin ein Test! ',
  };
},
methods: {
  onChangeMessage() {
    this.message = 'Ich bin die neue Nachricht! ';
  },
},

<div>
  <span>Reaktive Daten, die sich verändern: {{ message }}</span>
  <br />
  <button @click="onChangeMessage">Text verändern</button>
</div>
```

In diesem Fall registriert Vue, dass es eine Eigenschaft `message` gibt, die in den Daten definiert ist. Die Methode `onChangeMessage` greift darauf zu, um sie bei einem Methodenaufruf einmalig auf einen neuen Wert zu verändern. Gebunden werden diese Informationen über das Reactivity-System im Template in einem `span`-Element, um den Text möglichst einfach anzuzeigen. Abbildung 5–2 zeigt die Veränderung bei einem einfachen Button-Klick. An diesem einfachen Beispiel wird deutlich, wie mächtig das Reactivity-System ist, bei gleichzeitig niedrigschwelligem Einsatz. Daten definieren, nutzen und verändern, reicht völlig aus. Dabei ist noch der Hinweis relevant, dass der Mausklick, der die Methode aufruft, nur ein Beispiel von vielen ist. Durch welchen Mechanismus die Daten verändert werden, kümmert Vue nicht. Der relevante Aspekt ist, dass sie verändert werden und dass alle davon abhängigen Teile der Anwendung auf diese Anpassung reagieren.

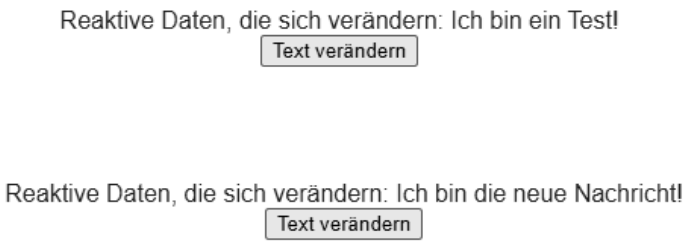

Abb. 5–2 *Reaktive Änderungen durch einen Button-Klick*

Vue bietet drei Möglichkeiten, um Daten in HTML-Templates zu rendern: Propertys, Computed Propertys und Methoden. Näheres dazu verrät das Kapitel 6 zu den Komponenten sowie das Kapitel 7, wenn es um die Verarbeitung von Eingabedaten geht.

Die Updates des DOM, die aus dem Reactivity-System erfolgen, werden von Vue *asynchron* durchgeführt. Intern wird eine Queue verwaltet, um alle Ände-

rungen von Daten in einem Event-Loop zwischenzuspeichern. Das verhindert unnötige Updates des DOM. Im nächsten Event-Loop-Tick werden die Änderungen dann am DOM durchgeführt. Da die Updates des DOM asynchron erfolgen und von Vue bis zum nächsten Tick zwischengespeichert werden, können wir über `nextTick` auf den nächsten Flush der DOM-Updates warten:

```
this.$nextTick(() => {
  // ...
});

await this.$nextTick();
```

Wir können entweder einen Callback als Argument angeben oder mit `await` auf den Promise warten.

5.2 Die Implementierung als Proxy in Vue 3

Damit das Reactivity-System in Vue funktioniert und um der Tatsache gerecht zu werden, dass es so wenig wie möglich im Weg steht, laufen im Hintergrund von Vue einige Prozesse ab, um aus normalen JavaScript-Objekten reaktive Vue-Objekte zu machen. Dieser Ablauf und die dazu notwendige Implementierung wurden erheblich zwischen den Versionen Vue 3 und 2 angepasst. Für Vue 3 wurde das Reactivity-System als Teil der Modularisierungsstrategie aus dem Vue Core entfernt und als eigenständiges Paket veröffentlicht. Der Code befindet sich wie gewohnt auf GitHub (Team V., @vue/reactivity GitHub Repository, 2022).

Vue 3 setzt in der neuen Umsetzung auf *Proxy-Objekte*. Diese sind notwendig, denn JavaScript besitzt im Standard keinen Mechanismus, um mitzubekommen, ob, geschweige denn wann, eine lokale Variable durch einen neuen Wert überschrieben wurde.

Die Zuweisung einer Variablen mit neuen Daten muss Vue aber abfangen, um anschließend darauf reagieren zu können. Daher werden diese Informationen zu Objekten, bei Vue 3 die angesprochenen Proxys, umgewandelt, weil eine Änderung an Eigenschaften eines Objekts erfasst werden kann.

So läuft es in Vue 2

In Vue 2 wurden dazu noch die Daten und Objekte im Data-Bereich einer Komponente durchgegangen, um Getter- und Setter-Methoden für die Daten zu erstellen. Dadurch lassen sich die Zugriffe nachverfolgen.

Die Proxy-Implementierung nutzt die neuen Möglichkeiten von ECMAScript 6 (Mozilla, Proxy-Implementierung, 2022). Das dort eingeführte Proxy-Objekt erlaubt es, ein Proxy für ein anderen Objekt zu erzeugen, um dann die darauf ausgeführten Operationen abzufangen. Genau das, was Vue braucht, um einige der ärgerlichen Probleme mit dem Reactivity-System aus Vue 2 auszumerzen – mehr dazu in Abschnitt 5.3. Das Proxy-Objekt ist dafür zuständig, die lesenden und schreibenden Zugriffe auf das originale Objekt abzufangen und zu tracken. Das folgende Snippet zeigt vereinfacht, wie diese Proxys in Vue zum Einsatz kommen:

```
const person = {
  firstname: 'Fabian',
};

const handler = {
  get(target, property, receiver) {
    console.log('Lesender Zugriff... ');
    track(target, property);
    return Reflect.get(...arguments);
  },

  set(target, property, value, receiver) {
    console.log('Schreibender Zugriff... ');
    trigger(target, property);
    return Reflect.set(...arguments);
  },
};

const proxy = new Proxy(person, handler);
console.log(proxy.firstname);
```

Das originale Objekt heißt in diesem Fall `person`. Zusätzlich wird ein Handler erstellt, der dann beim Erzeugen des Proxys zusammen mit dem originalen Objekt übergeben wird. Da der Handler eine `get`-Methode besitzt, wird diese nun aufgerufen, wenn die Eigenschaft gelesen wird. Es erfolgt somit die Ausgabe

```
Lesender Zugriff...
Fabian
```

auf der Konsole. Vue ist somit in der Lage, festzustellen, wenn die Daten gelesen werden. Durch die `set`-Methode wird auch das Schreiben neuer Daten über den Proxy erfasst. Vue nutzt das, um die lesenden Zugriffe über einen Aufruf der `track`-Methode zu tracken und die schreibenden Zugriffe über die `trigger`-Methode mitzubekommen. Über die `track`-Methode wird erfasst, dass die Eigenschaft eine Abhängigkeit zu einem bestimmten Codefragment ist. In Vue wird das ein Effekt genannt. Die `trigger`-Methode sorgt dafür, dass alle Codeteile erneut ausgeführt werden, wenn sich die Daten verändern. Diese Codeteile sind eben jene, die als abhängiger Code zu einer Variablen erkannt wurden.

`Reflect` (Mozilla, Reflect, 2022) sorgt hier dafür, dass sich das Binding von `this` korrekt verhält. Gewünscht ist, dass alle Methoden an den Proxy gebunden werden und nicht an das originale Zielobjekt. Dadurch ist sichergestellt, dass alle Zugriffe korrekt über den Proxy abgefangen werden können.

Abbildung 5–3 zeigt als schematische Darstellung, wie Änderungen nachverfolgt werden, um Änderungen zu triggern. Die Render-Funktion einer Komponente ist vereinfacht gesagt für den Aufbau des virtuellen DOM zuständig. Wenn die Komponente gerendert wird, werden alle Zugriffe auf Getter-Methoden erfasst, als sogenannter *Touch* bezeichnet. Diese Zugriffe werden im Watcher, pro Komponente wird ein Watcher erstellt, als Abhängigkeit erfasst. Denn diese per Getter angefragten Daten werden ja für das Rendern der Komponenten benötigt. Durch einen Zugriff auf eine Setter-Methode, wenn Daten verändert werden, lässt sich ein Notify an den Watcher absetzen. Da Daten geändert sind, ist ein neues Rendering erforderlich. Der Watcher weiß, welche Abhängigkeiten es gibt, und kann die betreffenden Codebereiche neu anstoßen und dann das Rendering durchführen. Dieser Vorgang passiert bei jedem Rendering beziehungsweise bei jeder Änderung an den Daten.

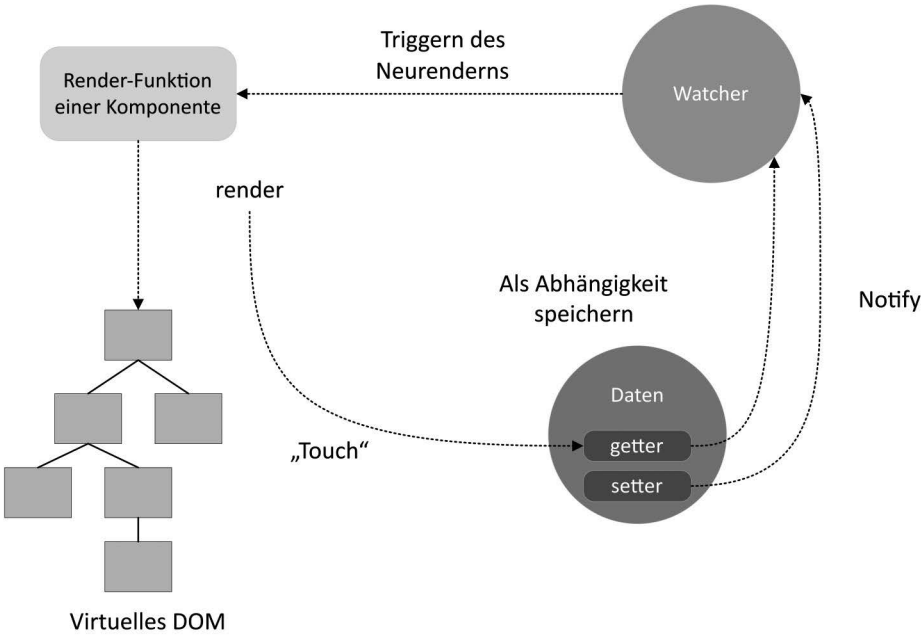

Abb. 5–3 *Schematische Darstellung, wie in Vue Änderungen nachverfolgt werden, um Änderungen zu erfassen und Komponenten neu zu rendern. Das Tracking von Änderungen ist ein Kern des Reactivity-Systems.*

Damit erfüllt die Proxy-Implementierung die drei Kriterien an Reactivity-Systeme aus Abschnitt 5.1. Die Track-Funktion in einer Get-Methode erfasst die aktuelle Eigenschaft und den laufenden Effekt. Der Set-Handler bekommt mit, wenn ein Wert verändert wird. Die Trigger-Funktion findet heraus, welche Effekte auf der veränderten Eigenschaft basieren und kann diese erneut ausführen.

Reactivity-System in Vue 2

In Version 2 von Vue ist das Reactivity-System komplett anders implementiert. Anstatt Proxys nutzt Vue 2 automatisch generierte Eigenschaften für Getter und Setter. Wenn ein JavaScript-Objekt in einer Vue-Instanz im Data-Bereich genutzt wird, geht Vue das Objekt durch und konvertiert alle Eigenschaften zu Getter und Setter (mit `Object.defineProperty`). Diese neuen Getter und Setter sind zwar für den Entwickler zunächst nicht sichtbar, werden aber unter der Haube für alle Änderungen an den Eigenschaften genutzt. Dadurch lassen sich beispielsweise Events auslösen, die über Änderungen informieren. Diese Implementierung ist der Grund, warum Vue 2 den IE8 und niedriger von Anfang an nicht unterstützt hat, denn dieses Feature kann nicht durch einen Patch (shim) nachgerüstet werden.

5.3 Probleme bei der Reaktivität

In Vue 3 kann die Performance beim Umwandeln normaler JavaScript-Objekte zu Proxy-Objekten zu einem Sorgenkind werden. Weil diese Umwandlungen rekursiv durchgeführt werden. Es betrifft somit auch alle verschachtelten Objekte. Das kann bei einer umfassenden Objektstruktur stark auf die Laufzeit schlagen. Insbesondere, wenn sich Objekte von Drittanbieter-Komponenten in dieser Objektstruktur befinden, da diese Objekte wiederum an sich bereits umfangreich sein können. Daher ist es wichtig, darauf zu achten, ob die Umwandlung mit Proxy-Objekten wirklich über die gesamte Objektstruktur hinweg notwendig ist. Über die Mechanismen von `markRaw` und den `shallow`-Funktionen besteht die Möglichkeit, selektiv aus dieser tiefen Umwandlung zu reaktiven oder Read-only-Objekten auszusteigen, also diese nicht durchzuführen. Im Zustandsgraphen einer Vue-Instanz lassen sich damit normale JavaScript-Objekte einbinden.

Vue 2 und reaktive Objekte

In Vue 2 gab es zudem Probleme mit Objekten, wenn dort dynamisch Eigenschaften hinzugefügt oder entfernt wurden. Da die Kapselung mit Gettern und Settern nur bei der Definition des Objekts geschieht, werden nachträgliche Änderungen nicht erfasst. Bei einem Objekt, bei dem vorher keine Eigenschaft mit dem Namen `b` existierte, lässt sich daher über die folgende Zeile keine Eigenschaft hinzufügen, die Vue über das Reactivity-System überwacht.

```
const einObject.b = 12;
```

Bei Arrays gibt es ebenfalls Probleme, wenn Elemente direkt über den Index gesetzt werden oder die Länge des Arrays angepasst wird:

```
this.elements[10] = 99;
this.elements.length = 1;
```

In beiden Fällen bekommt Vue 2 diese Änderungen nicht mit. In allen Fällen lässt sich die Vue.set-Methode nutzen, um Vue direkt mitzuteilen, dass Änderungen an den Objekten beziehungsweise Arrays vorliegen. Beispielsweise mit der folgenden Zeile Code:

```
Vue.set(vm.elements, indexOfItem, newValue);
```

Das funktioniert für das Beispiel mit dem Objekt und dem Array. Alternativ steht der Alias `vm.$set` zur Verfügung. Mit Vue 3 gehören diese Probleme aber erfreulicherweise alle der Vergangenheit an.

Ein weiteres Problem kann bei umfangreichen Objekten entstehen, die durch Vue 3 automatisch in Proxys umgewandelt werden. Dabei werden die Objekte *deep reactive*. Das bedeutet, dass die verschachtelten Daten ebenfalls reaktiv sind. Dies kann auch aus Versehen passieren, wenn wir ein Objekt in ein anderes einfügen und das eingefügte Objekt dadurch reaktiv machen. Über die Funktion `markRaw` markieren wir ein Objekt so, dass es niemals reaktiv wird.