KAPITEL 2

Das Leben einer Komponente

Nachdem Sie nun wissen, wie Sie mit den fertigen DOM-Komponenten arbeiten können, sollen Sie im Folgenden erfahren, wie Sie selbst welche bauen können.

Minimalversion

Die API zum Erstellen einer neuen Komponente sieht so aus:

```
var MyComponent = React.createClass({
  /* Spezifikation */
});
```

Bei der »Spezifikation« handelt es sich um ein JavaScript-Objekt, das eine Methode `render()` besitzen muss und eine Reihe optionaler Methoden und Eigenschaften haben kann. Ein minimales Beispiel kann so aussehen:

```
var Component = React.createClass({
  render: function() {
    return React.DOM.span(null, "Ich bin so neugierig");
  }
});
```

Wie Sie sehen, muss zwingend nur die Methode `render()` implementiert werden. Diese muss eine React-Komponente zurückgeben. Daher enthält das Beispiel ein `span` – reiner Text kann nicht zurückgegeben werden.

Ihre Komponente können Sie genau so wie die DOM-Komponenten verwenden:

```
ReactDOM.render(
  React.createElement(Component),
  document.getElementById("app")
);
```

Das Ergebnis Ihrer eigenen Komponente sehen Sie in Abbildung 2-1.

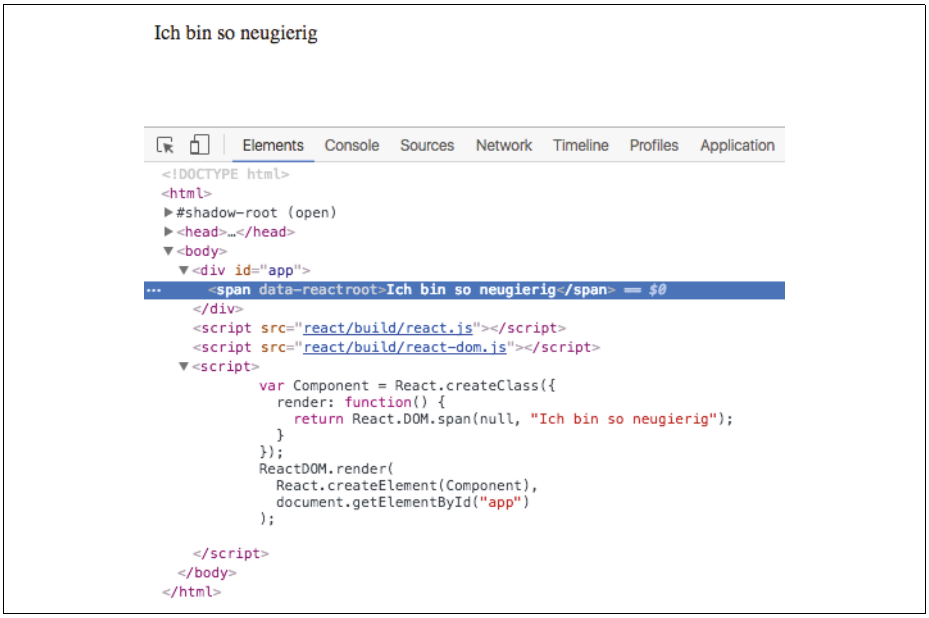

Abbildung 2-1: Ihre erste eigene Komponente

`React.createElement()` ist eine Möglichkeit, eine »Instanz« Ihrer Komponente zu erzeugen. Eine andere – sofern Sie viele Instanzen anlegen wollen – ist der Einsatz einer Fabrik:

```
var ComponentFactory = React.createFactory(Component);

ReactDOM.render(
  ComponentFactory(),
  document.getElementById("app")
);
```

Bei den Methoden aus `React.DOM.*`, die Ihnen schon vertraut sind, handelt es sich tatsächlich nur um Wrapper um `React.createElement()`, die Ihnen das Leben leichter machen sollen. Mit anderen Worten – dieser Code funktioniert auch mit DOM-Komponenten:

```
ReactDOM.render(
  React.createElement("span", null, "Hallo"),
  document.getElementById("app")
);
```

Wie Sie sehen, werden die DOM-Elemente als Strings und nicht wie bei eigenen Komponenten als JavaScript-Funktionen definiert.

Eigenschaften

Ihre Komponenten können Eigenschaften übernehmen und dann abhängig davon gerendert werden oder sich unterschiedlich verhalten. Alle Eigenschaften stehen über das Objekt `this.props` zur Verfügung. Schauen wir uns ein Beispiel an:

```
var Component = React.createClass({
  render: function() {
    return React.DOM.span(null, "Ich heiße " + this.props.name);
  }
});
```

Nun können wir die Eigenschaft beim Rendern der Komponente übergeben:

```
ReactDOM.render(
  React.createElement(Component, {
    Name: "Bob",
  }),
  document.getElementById("app")
);
```

Das Ergebnis sehen Sie in Abbildung 2-2.

`this.props` ist schreibgeschützt zu behandeln. Eigenschaften lassen sich dazu verwenden, Konfigurationsdaten von Eltern-Komponenten an Kind-Komponenten weiterzugeben (und von Kind-Komponenten an Eltern-Komponenten, wie wir später noch sehen werden). Möchten Sie gern eine Eigenschaft von `this.props` setzen, nutzen Sie stattdessen zusätzliche Variablen oder Eigenschaften des Spezifikationsobjekts Ihrer Komponente (also `this.thing` statt `this.props.thing`). In Browsern, die ECMAScript5 unterstützen, ist es außerdem gar nicht möglich, `this.props` zu verändern, denn:

```
> Object.isFrozen(this.props) === true; // true
```

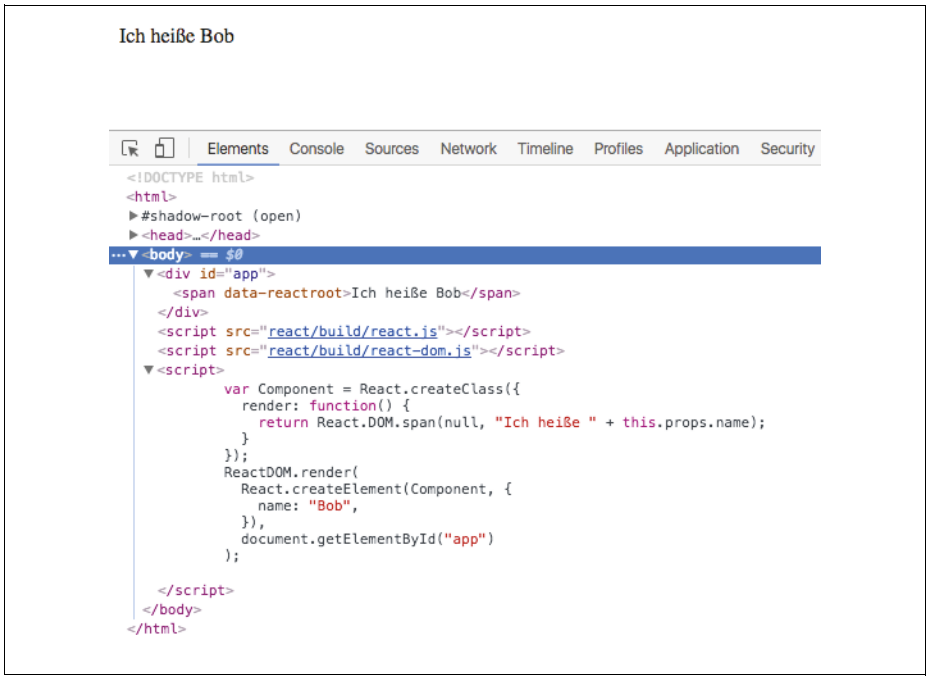

Abbildung 2-2: Der Einsatz von Komponenteneigenschaften

propTypes

In Ihren Komponenten können Sie eine Eigenschaft namens `propTypes` hinzufügen, um die Liste der von der Komponente akzeptierten Eigenschaften und deren Typen zu deklarieren. Hier ein Beispiel:

```
var Component = React.createClass({
  propTypes: {
    name: React.PropTypes.string.isRequired,
  },
  render: function() {
    return React.DOM.span(null, "Ich heiße " + this.props.name);
  }
});
```

Der Einsatz von `propTypes` ist optional, aber er bietet verschiedene Vorteile:

- Sie deklarieren gleich zu Beginn, welche Eigenschaften Ihre Komponente erwartet. Anwender Ihrer Komponente müssen dann nicht im (potenziell länglichen) Quellcode der Funktion `render()` herumsuchen, um herauszufinden, welche Eigenschaften sie zum Konfigurieren der Komponente verwenden können.
- React prüft die Werte der Eigenschaften zur Laufzeit, sodass Sie Ihre Funktion `render()` schreiben können, ohne mit den empfangenen Daten defensiv (oder gar paranoid) umgehen zu müssen.

Schauen wir uns die Überprüfung in Aktion an. `name: React.PropTypes.string.isRequired` fordert ganz klar einen Stringwert für die Eigenschaft `name`. Vergessen Sie, den Wert mitzugeben, erhalten Sie eine Warnung in der Konsole (Abbildung 2-3):

```
ReactDOM.render(
  React.createElement(Component, {
    // name: "Bob",
  }),
  document.getElementById("app")
);
```

Sie erhalten auch eine Warnung, wenn Sie eine Eigenschaft mit einem falschen Typ versehen, zum Beispiel einer Zahl (Abbildung 2-4):

```
React.createElement(Component, {
  name: 123,
})
```

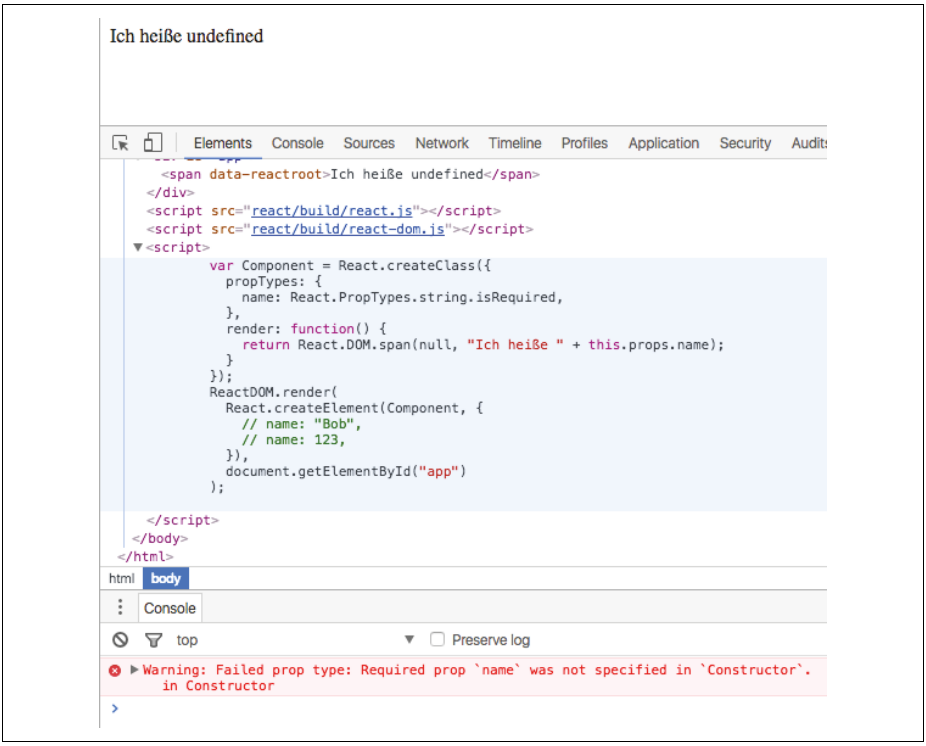

Abbildung 2-3: Warnung, wenn eine Pflichteigenschaft vergessen wird

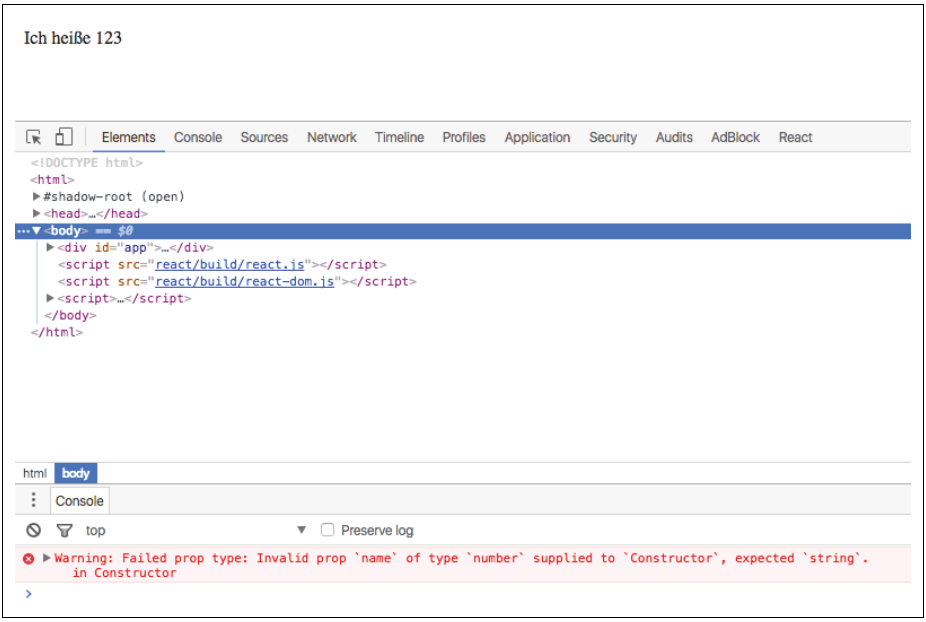

Abbildung 2-4: Warnung bei einem falschen Typ

In Abbildung 2-5 sehen Sie, was für Typen in `PropTypes` zur Verfügung stehen, damit Sie Ihre Erwartungen deklarieren können.

Das Deklarieren von `propTypes` ist in Ihren Komponenten optional, daher ist es auch möglich, ein paar, aber nicht alle Eigenschaften dort aufzuführen. Das klingt zwar nicht gerade nach einer guten Idee, aber Sie sollten sich der Möglichkeit bewusst sein, wenn Sie den Code anderer debuggen.

Abbildung 2-5: Alle React.PropTypes

Standardeigenschaftswerte

Können Ihrer Komponente optionale Eigenschaften mitgegeben werden, müssen Sie darauf achten, dass auch dann alles funktioniert, wenn die optionalen Eigenschaften fehlen. Das führt regelmäßig zu defensiven Standardcodezeilen wie:

```
var text = 'text' in this.props ? this.props.text : '';
```

Sie können es (um sich auf wichtigere Dinge konzentrieren zu können) vermeiden, solchen Code schreiben zu müssen, indem Sie die Methode `getDefaultProps()` implementieren:

```
var Component = React.createClass({
  propTypes: {
    firstName: React.PropTypes.string.isRequired,
    middleName: React.PropTypes.string,
    familyName: React.PropTypes.string.isRequired,
    address: React.PropTypes.string,
  },
```

```
  getDefaultProps: function() {
    return {
      middleName: '',
      address: 'n/a',
    };
  },

  render: function() {/* ... */}
});
```

Wie Sie sehen, gibt `getDefaultProps()` ein Objekt zurück, in dem vernünftige Werte für jede optionale Eigenschaft stehen (also die ohne `.isRequired`).

Status

Die bisherigen Beispiele waren ziemlich statisch (*stateless*, zustandslos). Das Ziel war, Ihnen eine Idee davon zu vermitteln, wie Sie Ihr UI aus Bausteinen aufbauen können. Aber React zeigt seine Stärken vor allem, wenn sich Daten in Ihrer Anwendung ändern (und gerade dort wird es auch mit der klassischen DOM-Anpassung im Browser kompliziert). React nutzt hier das Konzept des *Status* (`state`). Hierbei handelt es sich um die Daten, die Ihre Komponente zum Rendern verwendet. Ändert sich der Status, baut React das UI neu, ohne dass Sie etwas dafür tun müssen. Nachdem Sie also Ihr UI einmalig aufgebaut haben (in Ihrer Methode `render()`), müssen Sie sich nur noch darum kümmern, dass die Daten aktualisiert werden – das UI kann Ihnen egal sein. Denn Ihre `render()`-Methode enthält schließlich schon die Vorlage dafür, wie die Komponente aussehen soll.

Die UI-Aktualisierungen nach einem Aufruf von `setState()` werden mit einem Queuing-Mechanismus vorgenommen, sodass mehrere Änderungen auf einmal vorgenommen werden können. Passen Sie `this.state` direkt an, kann das zu unerwartetem Verhalten führen, daher sollten Sie darauf verzichten. Behandeln Sie `this.state` wie `this.props` als schreibgeschützt – nicht nur, weil ein anderes Vorgehen semantisch gesehen keine gute Idee wäre, sondern auch, weil Sie sonst seltsame Verhaltensweisen beobachten könnten. Genauso sollten Sie `this.render()` nie selbst aufrufen – überlassen Sie das React, das Änderungen zusammenfasst, herausfindet, was minimal angepasst werden muss, und `render()` nur bei Bedarf aufruft.

So, wie Sie Eigenschaften über `this.props` ansprechen können, greifen Sie auf den Status über das Objekt `this.state` zu. Um ihn zu aktualisieren, verwenden Sie `this.setState()`. Wird diese Methode aufgerufen, startet React Ihre `render()`-Methode und aktualisiert das UI.

React aktualisiert das UI, wenn `setState()` aufgerufen wird. Das ist das normale Vorgehen, aber wie Sie später noch lernen werden, gibt es eine Ausstiegsmöglichkeit. Sie können verhindern, dass das UI angepasst wird, indem Sie in einer speziellen *Lifecycle*-Methode namens `shouldComponentUpdate()` den Wert `false` zurückgeben.

Eine zustandsbehaftete Textarea-Komponente

Lassen Sie uns eine neue Komponente bauen – eine Textarea, die sich die Anzahl der eingegebenen Zeichen merkt (Abbildung 2-6).

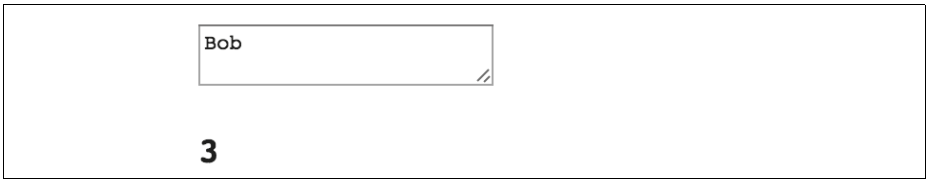

Abbildung 2-6: Das Ergebnis der eigenen Textarea-Komponente

Sie (und alle anderen Anwender dieser wiederverwendbaren Komponente) können die neue Komponente wie folgt einsetzen:

```
ReactDOM.render(
  React.createElement(TextAreaCounter, {
    text: "Bob",
  }),
  document.getElementById("app")
);
```

Implementieren wir nun die Komponente. Sie beginnen damit, eine »zustandslose« Version zu bauen, die sich nicht um Aktualisierungen kümmert – denn diese unterscheidet sich nicht sehr von den bisherigen Beispielen:

```
var TextAreaCounter = React.createClass({
  propTypes: {
    text: React.PropTypes.string,
  },

  getDefaultProps: function() {
    return {
      text: '',
    };
  },

  render: function() {
    return React.DOM.div(null,
      React.DOM.textarea({
        defaultValue: this.props.text,
      }),
      React.DOM.h3(null, this.props.text.length)
    );
  }
});
```

Vielleicht ist Ihnen aufgefallen, dass die Textarea in diesem Beispiel eine Eigenschaft `defaultValue` besitzt und kein `text`-Kind, wie Sie das aus normalem HTML kennen. Das liegt daran, dass es bei Formularen zwischen React und klassischem HTML ein paar Unterschiede gibt. Diese werden in Kapitel 4 besprochen – aber keine Bange, es sind wirklich nur Kleinigkeiten. Zudem werden Sie feststellen, dass diese Unterschiede sinnvoll sind und Ihnen als Entwickler das Leben leichter machen.

Wie Sie sehen, besitzt die Komponente eine optionale Stringeigenschaft `text`, mit der sie eine Textarea rendert – und dazu ein `<h3>`-Element, das einfach die `length` des Strings ausgibt (Abbildung 2-7).

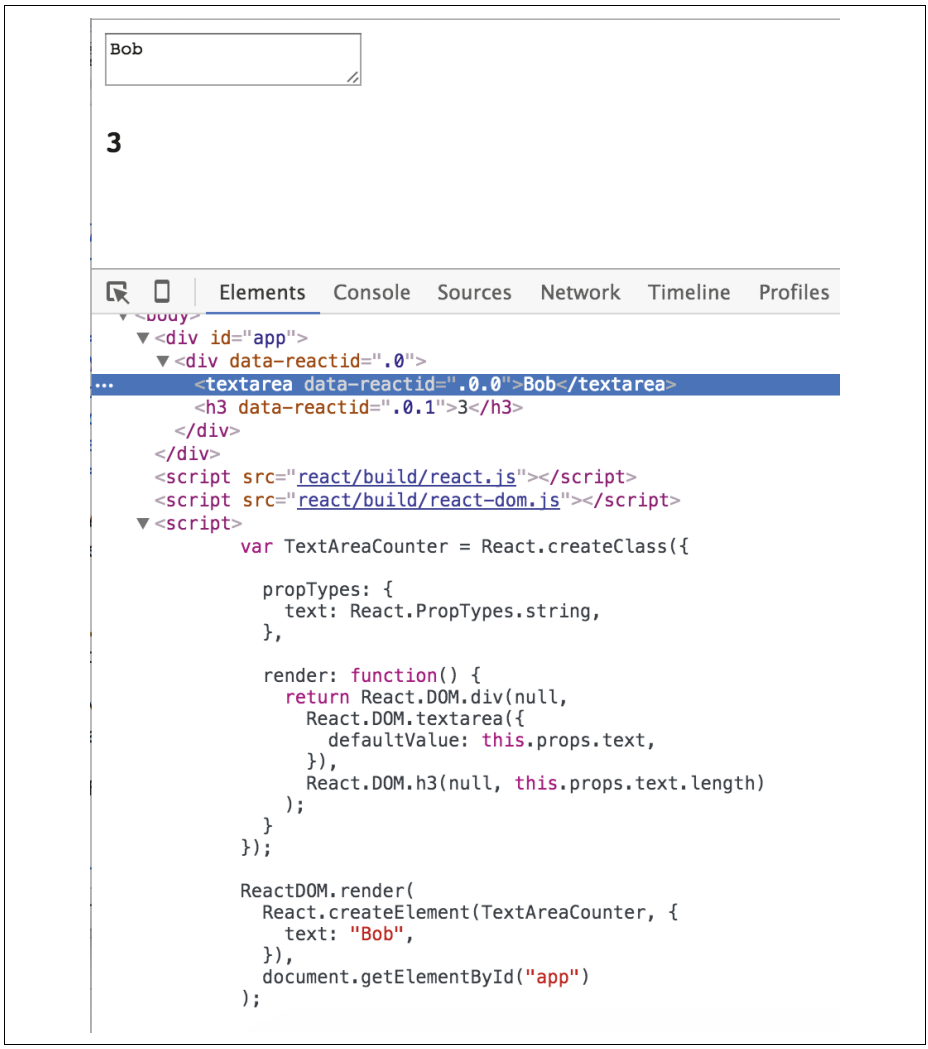

Abbildung 2-7: Die Komponente TextAreaCounter im Einsatz

Im nächsten Schritt wandeln wir diese *zustandslose* in eine *zustandsbehaftete* (stateful) Komponente um. Mit anderen Worten, die Komponente soll Daten verwalten (den Status) und diese nutzen, um sich selbst zunächst initial und später beim Ändern von Daten neu zu rendern.

Implementieren Sie in Ihrer Komponente eine Methode namens `getInitialState()`, damit Sie immer sicher mit sauberen Daten arbeiten:

```
getInitialState: function() {
  return {
    text: this.props.text,
  };
},
```

Bei den Daten, die diese Komponente verwaltet, handelt es sich einfach nur um den Text der Textarea, daher besitzt der Status lediglich eine Eigenschaft `text`, auf die Sie über `this.state.text` zugreifen können. Initial kopieren Sie (in `getInitialState()`) nur die Eigenschaft `text`. Wenn sich die Daten später ändern (der Benutzer tippt etwas in die Textarea), aktualisiert die Komponente ihren Status mit einer Hilfsmethode:

```
_textChange: function(ev) {
  this.setState({
    text: ev.target.value,
  });
},
```

Sie aktualisieren den Status immer mit `this.setState()`, das ein Objekt übernimmt und dieses mit den bestehenden Daten in `this.state` verschmilzt. Wie Sie sicherlich schon vermuten, ist `_textChange()` ein Event-Listener, der ein Eventobjekt `ev` übernimmt und sich daraus den Text der Textarea holt.

Nun müssen wir schließlich noch die Methode `render()` so anpassen, dass sie `this.state` statt `this.props` verwendet, und den Event-Listener einrichten:

```
render: function() {
  return React.DOM.div(null,
    React.DOM.textarea({
      value: this.state.text,
      onChange: this._textChange,
    }),
    React.DOM.h3(null, this.state.text.length)
  );
}
```

Wann immer nun der Anwender etwas in die Textarea eingibt, wird der Wert des Zählers aktualisiert (Abbildung 2-8).

Abbildung 2-8: Tippen in der Textarea

Ein Hinweis zu DOM-Events

Um Verwirrung zu vermeiden, möchte ich zu dieser Zeile:

```
onChange: this._textChange
```

ein paar Dinge klarstellen. React nutzt aus Performancegründen sein eigenes, *synthetisches* Eventsystem. Bequemlichkeit für die Entwickler und ein saubereres Arbeiten sind weitere Gründe. Um zu verstehen, warum das so ist, müssen Sie wissen, wie das Ganze in der reinen DOM-Welt abläuft.

Event-Handling in den alten Tagen

Es ist sehr bequem, *Inline*-Event-Handler wie den folgenden einzusetzen:

```
<button onclick="doStuff">
```

Das mag zwar praktisch und leicht zu lesen sein (der Event-Listener ist direkt im UI definiert), aber es ist sehr ineffizient, viele solcher Listener verteilt im Code zu haben. Zudem ist es so schwierig, mehr als einen Listener für den gleichen Button zu nutzen, besonders wenn sich der Button in einer »Komponente« oder Bibliothek von jemand anderem befindet und Sie darin nicht herumwerkeln wollen. Darum wird in der DOM-Welt `element.addEventListener` genutzt, um Listener einzurichten (was dazu führt, dass sich der Code an zwei oder mehr Stellen befindet) und *Event-Delegation* zu nutzen (um die Performanceprobleme anzugehen). Bei der Event-Delegation lauschen Sie an übergeordneten Knoten auf Events – zum Beispiel an einem `<div>`, das viele Buttons enthält – und richten dort einen Listener für alle diese Buttons ein.

Mit der Event-Delegation gehen Sie zum Beispiel so vor:

```
<div id="parent">
  <button id="ok">OK</button>
  <button id="cancel">Abbruch</button>
</div>

<script>
  document.getElementById('parent').addEventListener(
    'click', function(event) {
    var button = event.target;

    // je nach geklicktem Button etwas anderes tun
    switch (button.id) {
      case 'ok':
        console.log('OK!');
        break;
      case 'cancel':
        console.log('Cancel');
        break;
      default:
        new Error('Falsche Button-ID');
    };
  });
</script>
```

Das funktioniert gut und schnell, hat aber auch Nachteile:

- Der Listener wird weiter entfernt von der UI-Komponente deklariert, wodurch der Code schlechter zu finden und zu debuggen ist.
- Durch Delegation und Switching wird unnötiger, sich permanent wiederholender Code gebraucht, bevor Sie zu den eigentlich relevanten Codezeilen kommen (in diesem Fall auf einen Button-Klick reagieren).

- Durch Browserinkonsistenzen (die wir hier ignoriert haben) wird sogar noch mehr Code benötigt.

Geht es um Code für echte Anwender, ist sogar noch mehr notwendig, um alle Browser zu unterstützen:

- Neben `addEventListener` brauchen Sie auch noch `attachEvent`.
- Am Anfang des Listeners muss `var event = event || window.event` stehen.
- Sie benötigen die Zeile `var button = event.target || event.srcElement`.

All das ist nervig, aber so wichtig, dass Sie früher oder später zu einer Eventbibliothek greifen werden. Warum jedoch sollten Sie noch eine Bibliothek einbinden (und sich mit deren API vertraut machen müssen), wenn React doch schon eine Lösung mitbringt?

Event-Handling in React

React nutzt *synthetische* Events, um die Browserevents zu normalisieren und zu verpacken, sodass Sie sich nicht mehr mit Inkonsistenzen zwischen den Browsern herumschlagen müssen. Sie können sich immer darauf verlassen, dass `event.target` in allen Browsern zur Verfügung steht. Darum benötigen Sie im Snippet mit `TextAreaCounter` auch nur `ev.target.value`. Zudem ist die API zum Abbrechen von Events in allen Browsern gleich – `event.stopPropagation()` und `event.preventDefault()` funktionieren damit sogar in alten IEs.

Die Syntax macht es einfach, UI und Event-Listener zusammenzuhalten. Es scheint sich dabei um klassische Inline-Event-Handler zu handeln, aber so ist es nicht. Tatsächlich nutzt React aus Performancegründen Event-Delegation.

Brauchen Sie – warum auch immer – das eigentliche Browserevent, steht Ihnen dieses unter `event.nativeEvent` zur Verfügung, aber es ist sehr unwahrscheinlich, dass Sie je darauf zugreifen müssen.

Und noch etwas: Das Event `onChange` (zu sehen im Textarea-Beispiel) verhält sich wie erwartet – es feuert, wenn ein Benutzer etwas tippt, und nicht erst, wenn er mit der Eingabe fertig ist und das Feld verlässt. Letzteres Verhalten ist leider das des normalen DOM.

Props versus State

Jetzt wissen Sie, dass Sie Zugriff auf die Eigenschaften in `this.props` und auf den Status in `this.state` haben, wenn Sie Ihre Komponente in Ihrer `render()`-Methode darstellen wollen. Vielleicht fragen Sie sich, wann Sie das eine und wann das andere verwenden sollten.

Eigenschaften sind ein Mechanismus für die Außenwelt (die Anwender der Komponente), um Ihre Komponente zu konfigurieren. Der Status ist Ihre interne Daten-

sammlung. Wenn Sie eine Analogie zur objektorientierten Programmierung ziehen wollen, sind `this.props` die *an einen Klassenkonstruktor übergebenen Argumente*, während es sich bei `this.state` um die Menge Ihrer *privaten Eigenschaften* handelt.

Props im initialen Status: ein Anti-Pattern

Weiter oben haben Sie ein Beispiel für den Einsatz von `this.props` in `getInitialState()` gesehen:

```
getInitialState: function() {
  return {
    text: this.props.text,
  };
},
```

Das ist eigentlich ein Anti-Pattern. Idealerweise nutzen Sie eine Kombination aus `this.state` und `this.props`, um Ihr UI in Ihrer `render()`-Methode aufzubauen. Aber manchmal wollen Sie einen an Ihre Komponente übergebenen Wert zum Aufsetzen des initialen Status verwenden. Das ist in Ordnung, nur erwarten die Aufrufer Ihrer Komponente vermutlich, dass die Eigenschaften (im vorigen Beispiel `text`) immer den aktuellen Wert enthalten – und das Beispiel kann diese Erwartung nicht erfüllen. Besser ist eine kleine Namensänderung – nennen Sie die Eigenschaft beispielsweise `defaultText` oder `initialValue` und nicht einfach nur `text`:

```
propTypes: {
  defaultValue: React.PropTypes.string
},

getInitialState: function() {
  return {
    text: this.props.defaultValue,
  };
},
```

In Kapitel 4 erfahren Sie, wie React das für seine eigene Implementierung von Eingabefeldern und Textareas umsetzt, bei denen die Anwender schon Erwartungen aus früheren HTML-Einsätzen mitbringen.

Von außen auf die Komponente zugreifen

Sie genießen nicht immer den Luxus, eine brandneue React-App erstellen zu können. Manchmal müssen Sie sich mit einer bestehenden Anwendung oder Website arrangieren und können erst nach und nach zu React wechseln. Zum Glück wurde React so entworfen, dass es mit einer beliebigen bestehenden Codebasis zusammenarbeiten kann. Denn schließlich konnten die ursprünglichen Entwickler von React auch nicht einfach die Welt anhalten und eine riesige Anwendung (Facebook) erst einmal komplett neu schreiben.

Sie können Ihre React-App zum Beispiel mit dem Rest der Welt kommunizieren lassen, indem Sie sich von `ReactDOM.render()` eine Referenz auf eine Komponente liefern lassen, auf die Sie dann von außen zugreifen:

```
var myTextAreaCounter = ReactDOM.render(
  React.createElement(TextAreaCounter, {
    defaultValue: "Bob",
  }),
  document.getElementById("app")
);
```

Jetzt nutzen Sie `myTextAreaCounter`, um auf die gleichen Methoden und Eigenschaften zuzugreifen, die Sie normalerweise innerhalb der Komponente über `this` erreichen. Sie können sogar mit der Komponente an Ihrer JavaScript-Konsole herumspielen (Abbildung 2-9).

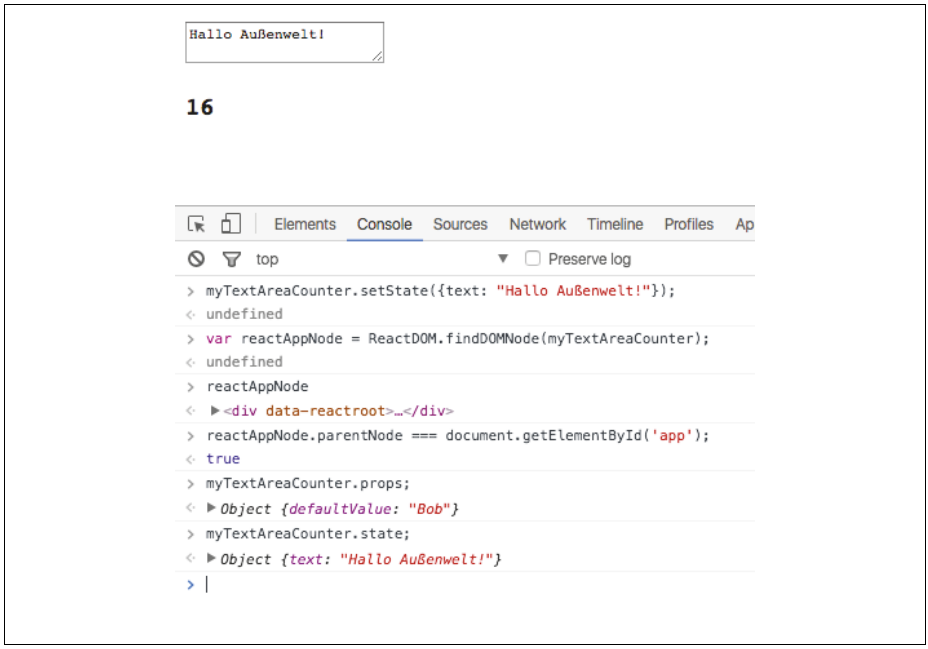

Abbildung 2-9: Zugriff auf die gerenderte Komponente über eine Referenz

Diese Zeile setzt einen neuen Status:

```
myTextAreaCounter.setState({text: "Hallo Außenwelt!"});
```

Damit erhalten Sie eine Referenz auf den Haupt-Eltern-DOM-Knoten, den React erstellt hat:

```
var reactAppNode = ReactDOM.findDOMNode(myTextAreaCounter);
```

Hierbei handelt es sich um das erste Kind-Element von `<div id="app">`, bei dem Sie React seine Arbeit machen ließen:

```
reactAppNode.parentNode === document.getElementById('app'); //true
```

So greifen Sie auf die Eigenschaften und den Status zu:

```
myTextAreaCounter.props; // Object { defaultValue: "Bob"}
myTextAreaCounter.state; // Object { text: "Hallo Außenwelt!"}
```

Sie haben auch von außen Zugriff auf die gesamte Komponenten-API. Nutzen Sie diese Möglichkeit aber – wenn überhaupt – nur selten. Setzen Sie eventuell `ReactDOM.findDOMNode()` ein, wenn Sie die Dimensionen des Knotens herausfinden müssen, um sicherzustellen, dass er auf Ihrer Seite Platz findet. Das reicht dann aber wirklich auch. Es mag verlockend sein, am Status von Komponenten herumzudrehen, die Ihnen nicht gehören, und sie zu »korrigieren«, aber Sie würden Erwartungen verletzen und für Folgefehler sorgen, weil die Komponente nicht davon ausgeht, so von außen manipuliert zu werden. Das Folgende funktioniert zum Beispiel, ist aber nicht zu empfehlen:

```
// schlechtes Beispiel
myTextAreaCounter.setState({text: 'Neiiiiiin'});
```

Eigenschaften später ändern

Wie Sie schon wissen, sind Eigenschaften eine Möglichkeit, eine Komponente zu konfigurieren. Es ist dabei durchaus sinnvoll, sie auch dann noch ändern zu wollen, wenn die Komponente schon erstellt wurde. Diese sollte allerdings darauf vorbereitet sein.

Schauen Sie sich die Methode `render()` aus dem vorigen Beispiel an, wird dort nur `this.state` verwendet:

```
render: function() {
  return React.DOM.div(null,
    React.DOM.textarea({
      value: this.state.text,
      onChange: this._textChange,
    }),
    React.DOM.h3(null, this.state.text.length)
  );
}
```

Ändern Sie die Eigenschaften außerhalb der Komponente, wird das keinen Einfluss auf das Rendern haben. Mit anderen Worten: Der Inhalt der Textarea wird auch bei folgendem Code gleich bleiben:

```
myTextAreaCounter = ReactDOM.render(
  React.createElement(TextAreaCounter, {
    defaultValue: "Hallo", // vorher als "Bob" bekannt
  }),
  document.getElementById("app")
);
```

Obwohl `myTextAreaCounter` durch einen neuen Aufruf von `React DOM.render()` »überschrieben« wird, bleibt der Status der Anwendung erhalten. React betreibt einen Abgleich (*Reconciliation*) der App vor und nach Änderungen. Statt einfach alles zu löschen, wird nur so wenig wie nötig geändert.

Der Inhalt von `this.props` ist nun anders (aber nicht das UI):

```
myTextAreaCounter.props; // Object { defaultValue="Hallo"}
```

Durch das Setzen des Status wird das UI nicht aktualisiert:

```
// schlechtes Beispiel
myTextAreaCounter.setState({text: 'Hallo'});
```

Das ist keine gute Idee, weil es in komplexeren Komponenten zu einem inkonsistenten Status führen kann – interne Zähler, Boolean-Flags, Event-Listener und Ähnliches könnten nun nicht mehr passen.

Wollen Sie Eingriffe von außen (das Ändern der Eigenschaften) korrekt behandeln, können Sie sich mit der Implementierung einer Methode namens `componentWillReceiveProps()` darauf vorbereiten:

```
componentWillReceiveProps: function(newProps) {
  this.setState({
    text: newProps.defaultValue,
  });
},
```

Wie Sie sehen, empfängt diese Methode das Objekt mit den neuen Eigenschaften, und Sie können den `state` entsprechend setzen, aber auch andere Aktivitäten durchführen, um die Komponente in einem sauberen Zustand zu halten.

Lifecycle-Methoden

Die Methode `componentWillReceiveProps()` aus dem vorigen Beispiel ist eine der sogenannten *Lifecycle*-Methoden, die React anbietet. Sie können sie verwenden, um auf Änderungen in Ihrer Komponente zu reagieren. Weitere Lifecycle-Methoden sind:

`componentWillUpdate()`
: Wird ausgeführt, bevor die `render()`-Methode Ihrer Komponente erneut aufgerufen wird (wenn sich etwas an den Eigenschaften oder dem Status geändert hat).

`componentDidUpdate()`
: Wird ausgeführt, wenn `render()` fertig ist und die neuen Änderungen auf das zugrunde liegende DOM angewendet wurden.

`componentWillMount()`
: Wird ausgeführt, bevor der Knoten in das DOM eingefügt wird.

`componentDidMount()`
: Wird ausgeführt, nachdem der Knoten in das DOM eingefügt wurde.

`componentWillUnmount()`
: Wird ausgeführt, kurz bevor die Komponente aus dem DOM entfernt wird.

`shouldComponentUpdate(newProps, newState)`
: Diese Methode wird vor `componentWillUpdate()` ausgeführt, und mit ihr haben Sie die Möglichkeit, `false` zurückzugeben und damit das Update abzubrechen – so wird `render()` nicht aufgerufen. Das ist in performancekritischen Anwendungen nützlich, wenn Sie der Meinung sind, dass sich nichts Interessantes geändert hat und ein Rerendering nicht nötig ist. Diese Entscheidung treffen Sie, indem Sie das Argument `newState` mit dem bestehenden `this.state` und `newProps` mit `this.props` vergleichen oder weil Sie einfach wissen, dass die Komponente statisch ist und sich nicht ändert. (Sie werden gleich ein Beispiel dafür kennenlernen.)

Lifecycle-Beispiel: Alles loggen

Um das Leben einer Komponente besser zu verstehen, wollen wir die Komponente `TextAreaCounter` mit etwas Logging versehen. Wir implementieren einfach alle Lifecycle-Methoden so, dass sie an der Konsole Log-Meldungen ausgeben, wenn sie aufgerufen werden:

```
var TextAreaCounter = React.createClass({
  _log: function(methodName, args) {
    console.log(methodName, args);
  },
  componentWillUpdate: function() {
    this._log('componentWillUpdate', arguments);
  },
  componentDidUpdate: function(){
    this._log('componentDidUpdate', arguments);
  },
  componentWillMount: function(){
    this._log('componentWillMount', arguments);
  },
  componentDidMount: function() {
    this._log('componentDidMount', arguments);
  },
  componentWillUnmount: function() {
    this._log('componentWillUnmount', arguments);
  },

  // ...
  // weitere Implementierungen, render() usw.
};
```

In Abbildung 2-10 sehen Sie, was passiert, wenn Sie die Seite laden.

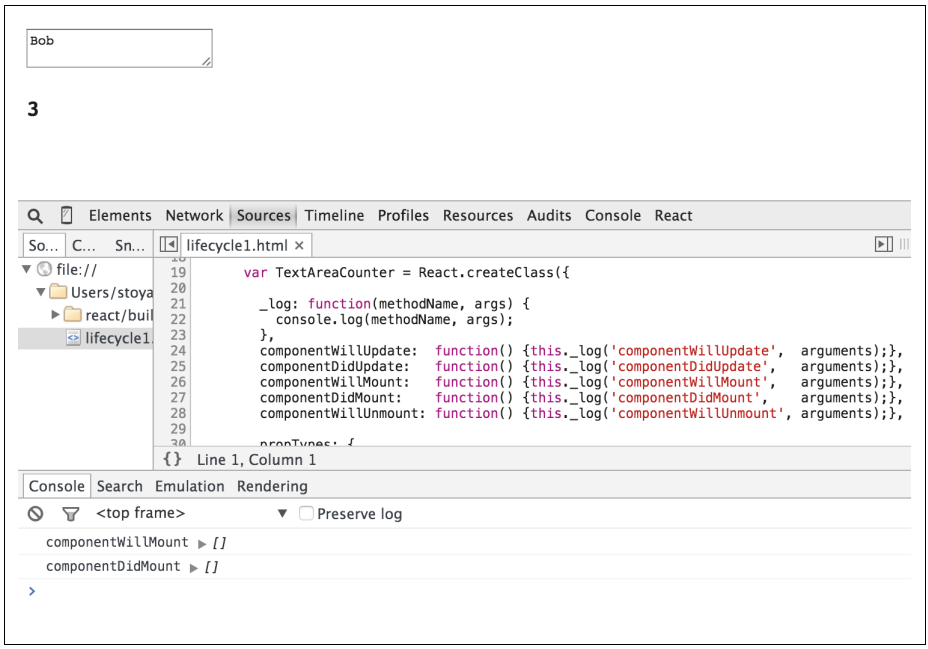

Abbildung 2-10: Die Komponente wird gemountet

Es werden zwei Methoden ohne Argumente aufgerufen, `componentDidMount()` ist im Allgemeinen die interessantere der beiden. Sie erhalten zum Beispiel bei Bedarf per `ReactDOM.findDOMNode(this)` Zugriff auf den frisch gemounteten DOM-Knoten, um die Dimensionen der Komponente auslesen zu können. Auch lässt sich hier Initialisierungskram erledigen, da Ihre Komponente ja nun lebt.

Was passiert, wenn Sie jetzt ein »s« eingeben, um den Text in »Bobs« zu verändern (Abbildung 2-11)?

Die Methode `componentWillUpdate(nextProps, nextState)` wird mit den neuen Daten aufgerufen, aus denen die Komponente gerendert werden wird. Das erste Argument ist der zukünftige Wert von `this.props` (der sich in diesem Beispiel nicht ändert), das zweite der zukünftige Wert von `this.state`. Das dritte Argument ist `context`, das hier nicht von Interesse sein soll. Sie können zum Beispiel das Argument `newProps` mit dem aktuellen `this.props` vergleichen und abhängig davon entscheiden, ob etwas zu tun ist.

Nach `componentWillUpdate()` wird nun `componentDidUpdate(oldProps, oldState)` aufgerufen, wobei die Werte von `props` und `state` in ihrem Zustand vor der Änderung übergeben werden. Das ist eine Gelegenheit, etwas nach dem Update durchzuführen. Sie können hier `this.setState()` verwenden, was in `componentWillUpdate()` nicht möglich ist.

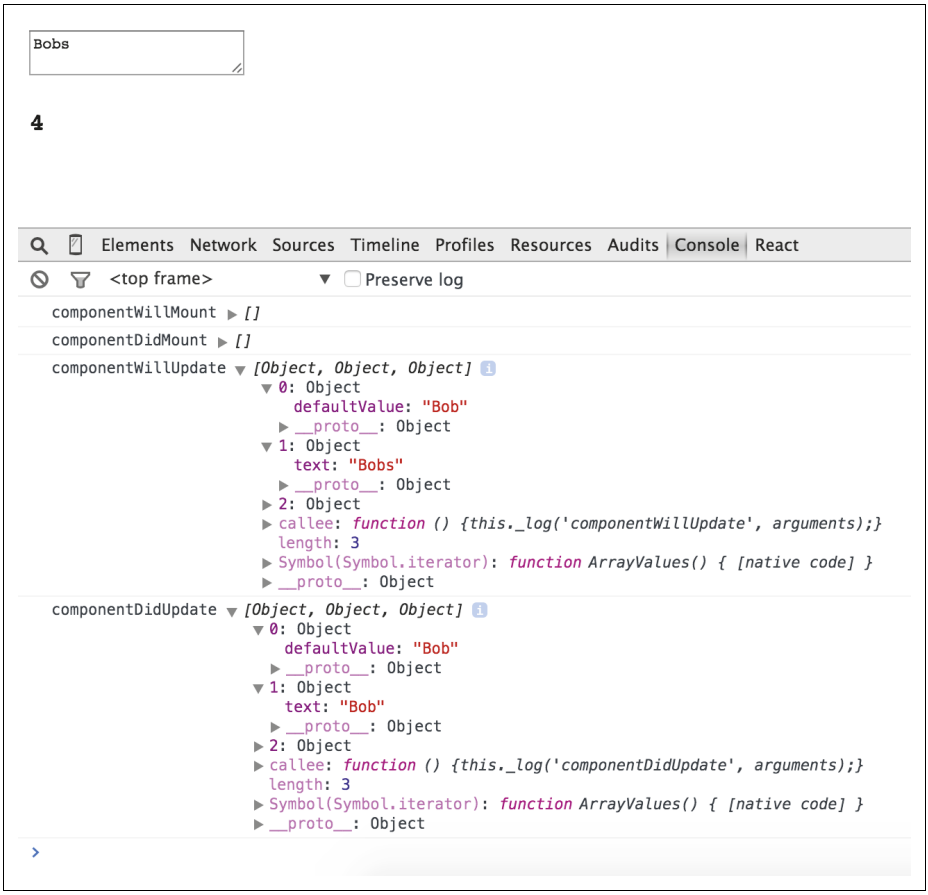

Abbildung 2-11: Die Komponente wird aktualisiert

Stellen Sie sich zum Beispiel vor, Sie wollen die Anzahl der Zeichen begrenzen, die in der Textarea eingegeben werden können. Sie sollten das im Event-Handler `_textChange()` umsetzen, der bei der Benutzereingabe aufgerufen wird. Aber was, wenn jemand (vielleicht eine jüngere, naivere Version Ihrer selbst?) `setState()` außerhalb der Komponente aufruft (was, wie erwähnt, keine gute Idee ist)? Können Sie die Konsistenz und das Wohlbefinden Ihrer Komponente dann trotzdem noch sicherstellen? Natürlich. Nehmen Sie die Validierung in `componentDidUpdate()` vor und setzen Sie den Status auf den alten Wert zurück, wenn die Zeichenzahl größer ist als erlaubt – in etwa so:

```
componentDidUpdate: function(oldProps, oldState) {
  if (this.state.text.length > 3) {
    this.replaceState(oldState);
  }
},
```

Das mag sehr paranoid aussehen, aber es ist möglich.

Beachten Sie, dass wir hier `replaceState()` statt `setState()` verwendet haben. Während `setState(obj)` die Eigenschaften von `obj` mit denen von `this.state` verschmilzt, wird bei `replaceState()` alles überschrieben.

Lifecycle-Beispiel: Ein Mixin verwenden

Im vorigen Beispiel haben wir die Aufrufe von vier unserer fünf Lifecycle-Methoden protokollieren lassen. Die fünfte – `componentWillUnmount()` – lässt sich am besten kennenlernen, wenn Sie Kind-Komponenten haben, die von einer Eltern-Komponente entfernt werden. In diesem Beispiel sollen alle Änderungen sowohl im Kind-Element als auch im Eltern-Element protokolliert werden. Führen wir dazu ein neues Konzept ein, um Code wiederzuverwenden: ein Mixin.

Bei einem Mixin handelt es sich um ein JavaScript-Objekt, das eine Sammlung von Methoden und Eigenschaften besitzt. Das Mixin ist nicht dazu gedacht, allein verwendet zu werden, stattdessen wird es in die Eigenschaften anderer Objekte eingebunden (»eingemixt«). Im Logging-Beispiel kann ein Mixin so aussehen:

```
var logMixin = {
  _log: function(methodName, args) {
    console.log(this.name + '::' + methodName, args);
  },
  componentWillUpdate: function() {
    this._log('componentWillUpdate', arguments);
  },
  componentDidUpdate: function(){
    this._log('componentDidUpdate', arguments);
  },
  componentWillMount: function(){
    this._log('componentWillMount', arguments);
  },
  componentDidMount: function() {
    this._log('componentDidMount', arguments);
  },
  componentWillUnmount: function() {
    this._log('componentWillUnmount', arguments);
  },
};
```

In einer Welt ohne React können Sie mit `for-in` die Eigenschaften durchlaufen und sie in ein neues Objekt kopieren, um die Funktionalität des Mixins dort zu erhalten. Bei React gibt es eine Vereinfachung: die Eigenschaft `mixins`. Sie sieht so aus:

```
var MyComponent = React.createClass({

  mixins: [obj1, obj2, obj3],

  // die restlichen Methoden ...

};
```

Sie weisen der Eigenschaft `mixins` ein Array mit JavaScript-Objekten zu, und React kümmert sich um den Rest. Nehmen Sie das `logMixin` also wie folgt in Ihre Komponente auf:

```
var TextAreaCounter = React.createClass({
  name: 'TextAreaCounter',
  mixins: [logMixin],
  // der Rest ...
};
```

Wie Sie sehen, wird durch den Schnipsel auch noch eine praktische Eigenschaft `name` hinzugefügt, mit der der Aufrufer identifiziert werden kann.

Lassen Sie das Beispiel mit dem Mixin laufen, können Sie das Logging in Aktion verfolgen (Abbildung 2-12).

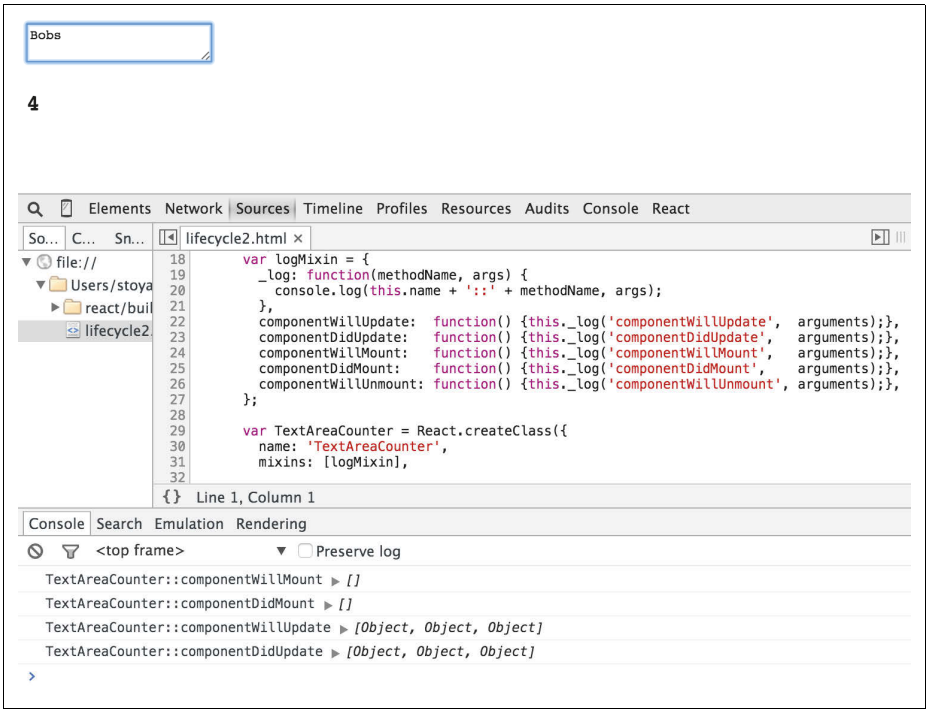

Abbildung 2-12: Ein Mixin verwenden und die Komponente identifizieren

Lifecycle-Beispiel: Der Einsatz einer Kind-Komponente

Sie wissen jetzt, wie Sie React-Komponenten einmischen und einbetten können. Bisher haben wir in den `render()`-Methoden nur Komponenten aus `React.DOM` genutzt (und keine eigenen). Schauen wir uns eine einfache eigene Komponente an, die wir als Kind einsetzen wollen.

Sie können den Zähler-Teil abtrennen und in eine eigene Komponente packen:

```
var Counter = React.createClass({
  name: 'Counter',
  mixins: [logMixin],
  propTypes: {
    count: React.PropTypes.number.isRequired,
  },
  render: function() {
    return React.DOM.span(null, this.props.count);
  }
});
```

Diese Komponente enthält lediglich den Zähler-Teil – sie rendert ein `<span>` und verwaltet keinen `state`, sondern zeigt nur die vom Eltern-Element übergebene Eigenschaft `count` an. Zudem wird `logMixin` eingemischt, um auszugeben, wann die Lifecycle-Methoden aufgerufen werden.

Aktualisieren wir jetzt die `render()`-Methode der Eltern-Komponente `TextArea Counter`. Sie sollte die `Counter`-Komponente nur bei Bedarf einsetzen – hat der Zähler den Wert 0, soll keine Nummer angezeigt werden:

```
render: function() {
  var counter = null;
  if (this.state.text.length > 0) {
    counter = React.DOM.h3(null,
      React.createElement(Counter, {
        count: this.state.text.length,
      })
    );
  }
  return React.DOM.div(null,
    React.DOM.textarea({
      value: this.state.text,
      onChange: this._textChange,
    }),
    counter
  );
}
```

Die Variable `counter` hat den Wert `null`, wenn die Textarea leer ist. Gibt es Text, enthält die Variable den Teil des UI, der dafür zuständig ist, die Anzahl der Zeichen auszugeben. Es ist nicht notwendig, dass das gesamte UI als Argumente der Haupt-Komponente `React.DOM.div` eingebettet wird – Sie können auch UI-Teile in Variablen ablegen und diese nach Bedarf einsetzen.

Schauen Sie sich jetzt die Lifecycle-Methoden an, die für beide Komponenten protokolliert werden. In Abbildung 2-13 sehen Sie, was passiert, wenn Sie die Seite laden und dann den Inhalt der Textarea ändern.

Sie sehen, wie die Kind-Komponente vor der Eltern-Komponente gemountet und aktualisiert wird.

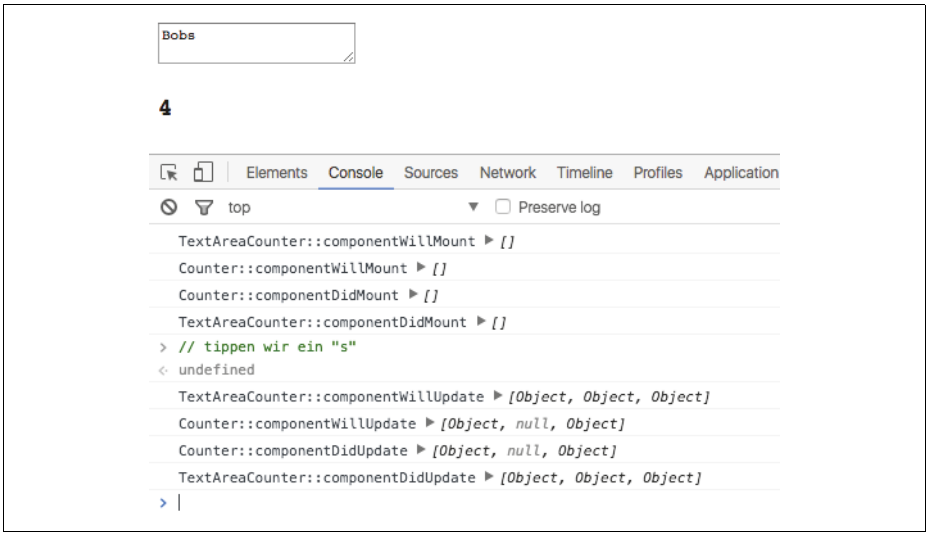

Abbildung 2-13: Zwei Komponenten mounten und aktualisieren

In Abbildung 2-14 sehen Sie die Abfolge, wenn Sie den Text in der Textarea löschen und der Zähler 0 wird. In diesem Fall wird die Kind-Komponente `counter` zu `null` und ihr DOM-Knoten aus dem DOM-Baum entfernt, nachdem Sie per `componentWillUnmount` informiert wurden.

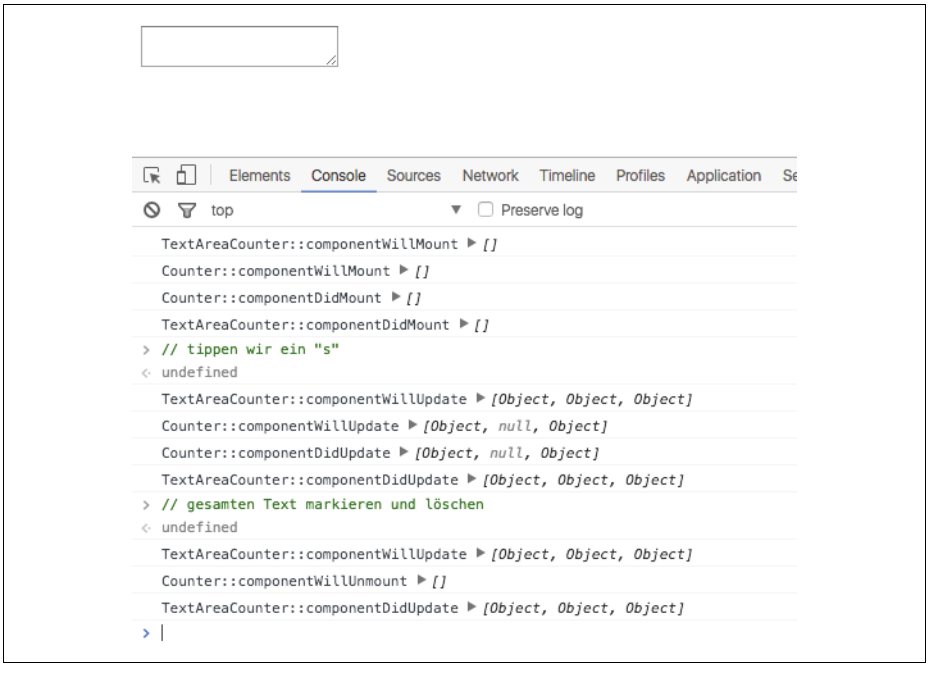

Abbildung 2-14: Die Counter-Komponente unmounten

Performanceverbesserung: Aktualisieren von Komponenten verhindern

Die letzte Lifecycle-Methode, die Sie kennen sollten, – insbesondere wenn Sie an den performancekritischen Teilen Ihrer App bauen –, ist die Methode `shouldComponentUpdate(nextProps, nextState)`. Sie wird vor `componentWillUpdate()` aufgerufen und gibt Ihnen die Chance, die Aktualisierung abzubrechen, wenn Sie der Meinung sind, sie sei nicht nötig.

Es gibt eine ganze Reihe von Komponenten, die in ihrer `render()`-Methode nur `this.props` und `this.state` einsetzen, aber keine anderen Funktionsaufrufe. Diese Komponenten werden als »pur« bezeichnet. Sie können `shouldComponentUpdate()` implementieren und Eigenschaften und Status vor und nach den Änderungen vergleichen. Gibt es keine (wichtigen) Änderungen, kann die Implementierung `false` zurückgeben und damit etwas Rechenleistung sparen. Es kann zudem pure statische Komponenten geben, die weder `props` noch `state` verwenden – solche können immer direkt `false` liefern.

Schauen wir uns an, was beim Aufruf der `render()`-Methoden passiert. Dann implementieren wir `shouldComponentUpdate()`, um einen Performancegewinn zu erzielen.

Als Erstes nehmen wir die neue `Counter`-Komponente. Entfernen Sie das Logging-Mixin und loggen Sie stattdessen jedes Mal an der Konsole, wenn die `render()`-Methode aufgerufen wird:

```
var Counter = React.createClass({
  name: 'Counter',
  // mixins: [logMixin],
  propTypes: {
    count: React.PropTypes.number.isRequired,
  },
  render() {
    console.log(this.name + '::render()');
    return React.DOM.span(null, this.props.count);
  }
});
```

Machen Sie das Gleiche in `TextAreaCounter`:

```
var TextAreaCounter = React.createClass({
  name: 'TextAreaCounter',
  // mixins: [logMixin],

  // alle anderen Methoden ...
  render: function() {
    console.log(this.name + '::render()');
    // ... und der Rest des Renderns
  }
});
```

Wenn Sie jetzt die Seite laden und in der Textarea den alten String »Bob« per Copy-and-paste durch den neuen String »LOL« ersetzen, sehen Sie das Ergebnis aus Abbildung 2-15.

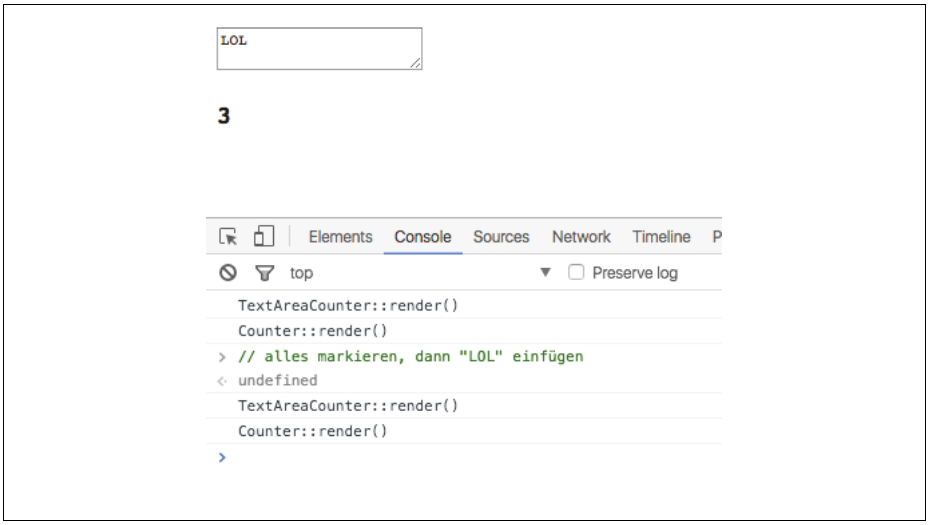

Abbildung 2-15: Beide Komponenten werden gerendert

Verändern Sie den Text, führt das zu einem Aufruf der `render()`-Methode von `TextAreaCounter`, die wiederum die `render()`-Methode von `Counter` aufruft. Wird »Bob« durch »LOL« ersetzt, ist die Anzahl der Zeichen vor und nach dem Update gleich, und das UI des Counters muss nicht geändert werden – der Aufruf von dessen `render()`-Methode ist überflüssig. Sie können React bei der Optimierung dieses Falls helfen, indem Sie `shouldComponentUpdate()` implementieren und `false` zurückgeben, wenn kein weiteres Rendern notwendig ist. Der Methode werden die zukünftigen Werte von `props` und `state` übergeben (wobei `state` in dieser Komponente nicht gebraucht wird), die Sie dann mit den aktuellen Werten vergleichen können:

```
shouldComponentUpdate(nextProps, nextState_ignore) {
  return nextProps.count !== this.props.count;
}
```

Damit wird bei einem Wechsel von »Bob« zu »LOL« dafür gesorgt, dass der `Counter` nicht neu gerendert werden muss (Abbildung 2-16).

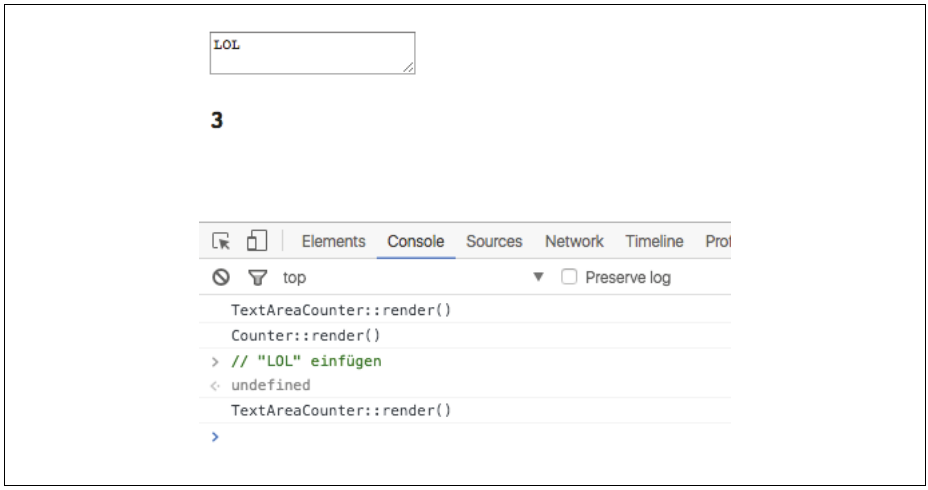

Abbildung 2-16: Performanceverbesserung – ein Rendering-Zyklus wird eingespart

PureRenderMixin

Die Implementierung von `shouldComponentUpdate()` ist ziemlich einfach. Und es ist kein großer Aufwand, sie auch noch generisch zu machen, da Sie immer `this.props` mit `nextProps` und `this.state` mit `nextState` vergleichen. React stellt eine solche generische Implementierung in Form eines Mixins bereit, das Sie in eine beliebige Komponente einbinden können.

So sieht das aus:

```
<script src="react/build/react-with-addons.js"></script>
<script src="react/build/react-dom.js"></script>
<script>

  var Counter = React.createClass({
    name: 'Counter',
    mixins: [React.addons.PureRenderMixin],
    propTypes: {
      count: React.PropTypes.number.isRequired,
    },
    render: function() {
      console.log(this.name + '::render()');
      return React.DOM.span(null, this.props.count);
    }
  });

  // ...
</script>
```

Das Ergebnis (Abbildung 2-17) ist das Gleiche – die `render()`-Methode von `Counter` wird nicht aufgerufen, wenn sich an der Zahl der Zeichen nichts geändert hat.

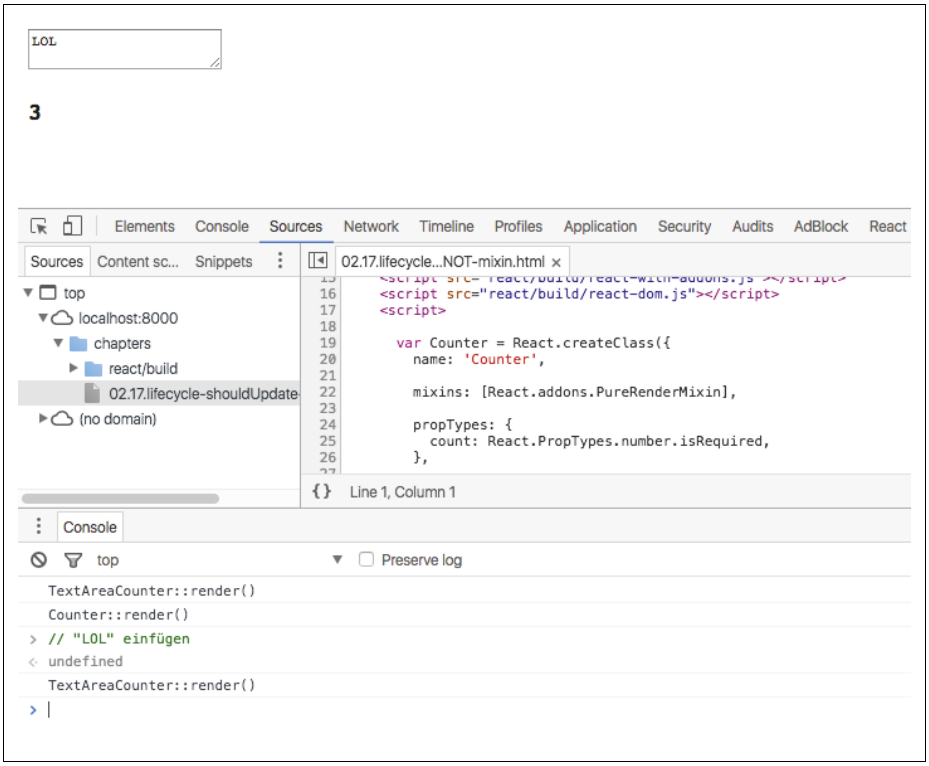

Abbildung 2-17: Einfacher Performancegewinn: Einbinden von PureRenderMixin

Beachten Sie, dass `PureRenderMixin` kein Teil des React-Cores ist, sondern zu einer erweiterten Version von React mit Add-ons gehört. Um dieses Mixin nutzen zu können, müssen Sie statt *react/build/react.js* die Datei *react/build/react-with-addons.js* einbinden. Damit erhalten Sie einen neuen Namensraum `React.addons`, in dem Sie dann sowohl `PureRenderMixin` als auch andere nette Goodies finden können.

Wollen Sie nicht alle Add-ons einbinden oder Ihre eigene Version des Mixins implementieren, sollten Sie einfach einen Blick in die React-Implementierung werfen. Sie ist ziemlich einfach und geradlinig umgesetzt, eine flache (nicht rekursive) Prüfung auf Gleichheit – in etwa so:

```
var ReactComponentWithPureRenderMixin = {
  shouldComponentUpdate: function(nextProps, nextState) {
    return !shallowEqual(this.props, nextProps) ||
           !shallowEqual(this.state, nextState);
  }
};
```