O'REILLY®

React Native

NATIVE APPS PARALLEL FÜR ANDROID UND IOS ENTWICKELN

Erik Behrends

Inhalt

KAPITEL 4

Plattformübergreifende UI-Komponenten verwenden

Wir beginnen in diesem Kapitel mit der Entwicklung einer App namens *MyJournal*, mit der Sie auf dem Smartphone ein Tagebuch führen können. Mit MyJournal können Sie Textnotizen mit und ohne Bild erstellen, die in der App gespeichert werden. Alle erstellten Einträge können Sie in einer Liste in kompakter Ansicht durchblättern und sich die Details eines Eintrags anschauen und bearbeiten, um z.B. die Wetterdaten des Tages für einen bestimmten Ort hinzuzufügen. In Abbildung 4-1 ist dargestellt, wie MyJournal am Ende aussehen wird.

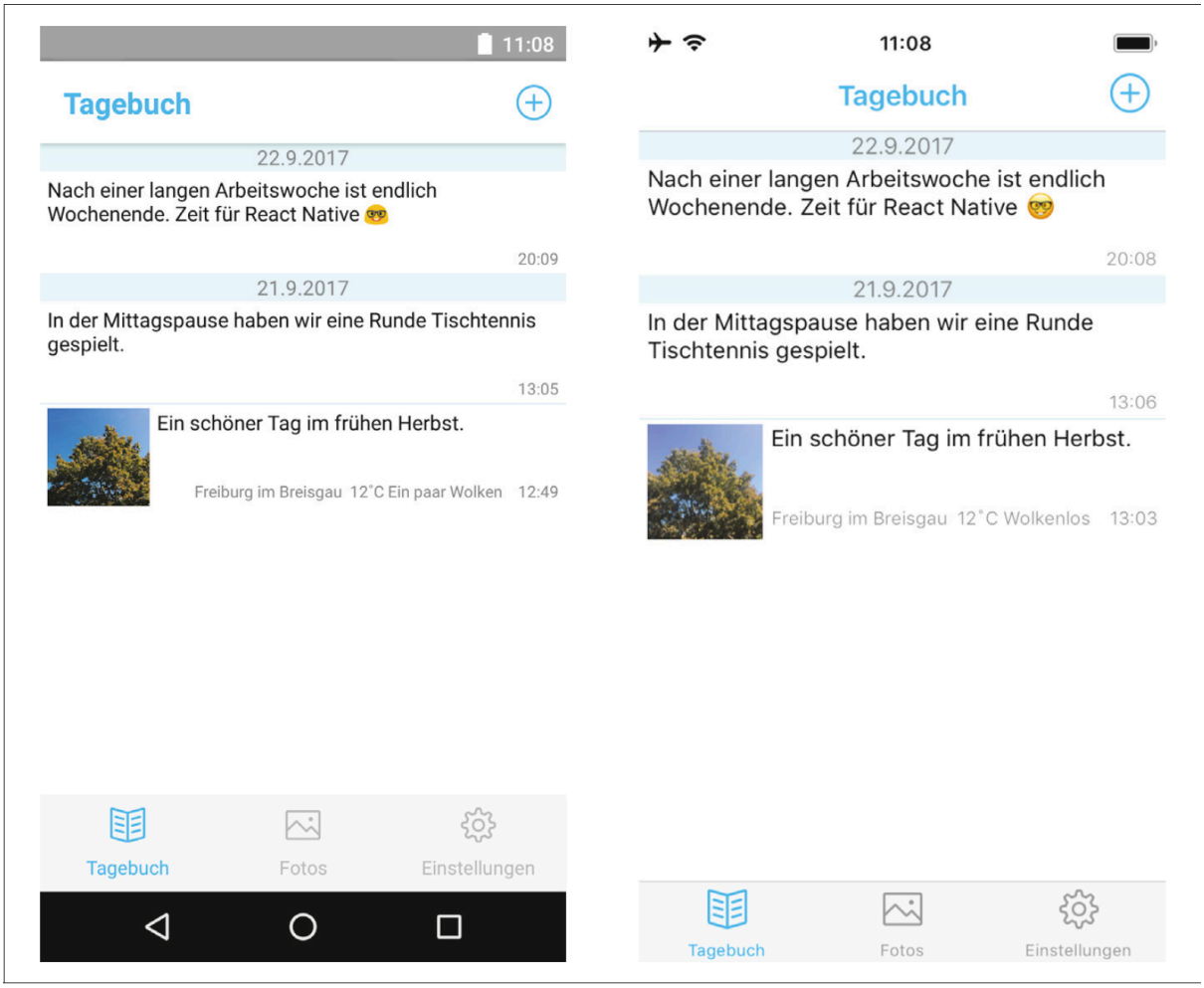

Abbildung 4-1: Ein Tagebuch mit MyJournal führen

Wir werden in diesem Kapitel einen einfachen Prototyp der App MyJournal erstellen, den wir in den nachfolgenden Kapiteln weiterentwickeln werden.

Für die Programmierung von Apps mit React Native stehen uns einige UI-Komponenten zur Verfügung, die auf der jeweiligen Plattform als native Elemente dargestellt werden. Die Programmierung von plattformübergreifenden Apps mit React Native zu erlernen, beinhaltet daher, passende UI-Komponenten einsetzen zu können. So haben wir bereits im zweiten Kapitel von den Komponenten `View`, `Text` und `Button` für die App StepCounter Gebrauch gemacht. In diesem Kapitel gehe ich detaillierter auf verschiedene häufig verwendete UI-Komponenten ein, die in React Native enthalten sind.

MyJournal in Expo laden

Wenn Sie möchten, können Sie die finale Version von MyJournal in der Expo-App laden und ausprobieren. Den QR-Code zum Laden von MyJournal in Expo finden Sie unter *expo.io/@behrends/myjournal*.

View und Text

Bei der Entwicklung der App MyJournal werden wir nun schrittweise vorgehen. Öffnen Sie die Konsole und erstellen Sie zunächst ein neues Projekt für MyJournal:

```
create-react-native-app MyJournal
```

Wechseln Sie dann in den Projektordner und starten Sie dort den React Native Packager:

```
cd MyJournal
```

```
npm start
```

Nun können Sie die App mithilfe des in der Konsole angezeigten QR-Codes mit Expo auf dem Smartphone laden, so wie in Abschnitt »Die App auf dem Smartphone mit Expo testen« auf Seite 15 beschrieben.

Versionskontrolle für MyJournal

Die Entwicklung der App MyJournal wird uns bis zum Ende des Buchs begleiten. Ich möchte Ihnen für die Versionskontrolle zur Verwendung einer Software raten, etwa *git*, damit Sie gegebenenfalls ungewollte Änderungen leicht rückgängig machen können. Dies ist nicht zwingend erforderlich, aber falls Sie mit *git* oder einer ähnlichen Software arbeiten wollen, sollten Sie den Projektordner jetzt unter Versionskontrolle stellen und nach jeder größeren Änderung am Code den aktuellen Stand als neue Version einpflegen.

Starten Sie Ihren Editor und öffnen Sie dort den Projektordner der App. Zuerst betrachten wir die Datei *App.js*, in der eine Komponente `App` als Klasse definiert wird. `App` ist aktuell die einzige Komponente von MyJournal und wird automatisch als Hauptkomponente geladen, wenn MyJournal auf dem Smartphone ausgeführt wird. In der Methode `render` können wir sehen, aus welchen Bestandteilen die Benutzeroberfläche der Komponente `App` aufgebaut ist. Drei `Text`-Komponenten mit verschiedenen Textinhalten werden in einem `View`-Element zusammengefasst:

```
  render() {

    return (

      <View style={styles.container}>

        <Text>Open up App.js to start working on your app!
        </Text>

        <Text>Changes you make will automatically reload.
        </Text>

        <Text>Shake your phone to open the developer menu.
        </Text>

      </View>

    );

  }
```

Texte darstellen mit Text

Offensichtlich wird `Text` verwendet, um Textinhalte darzustellen. Wenn in einer App textbasierte Inhalte angezeigt werden sollen, müssen diese in Elementen vom Typ `Text` eingebettet werden. Umgekehrt dürfen nur `Text`-Komponenten solche Textinhalte enthalten, das heißt, in anderen Komponenten ist bloßer Text als Inhalt des Elements nicht erlaubt. So würde das Einbetten eines Textinhalts in ein `View`-Element einen Fehler hervorrufen:

```
  // Textinhalte dürfen nur in Text-Elementen enthalten sein

  <View>Dies ergibt einen Fehler!</View>
```

Also muss Textinhalt immer in `Text`-Elemente eingebettet werden:

```
// Textinhalt korrekt von Text-Element umgeben

<View>

  <Text>Textinhalte bitte nur in Text-Elementen</Text>

</View>
```

Ein `Text`-Element kann weitere Elemente vom Typ `Text` enthalten, was z.B. dann nützlich ist, wenn verschiedene Textstücke innerhalb eines Textblocks unterschiedlich dargestellt werden sollen. Die von React Native zur Verfügung gestellten UI-Komponenten geben verschiedene Eigenschaften bzw. Props vor, mit denen eine Verwendung der Komponenten angepasst werden kann. So kann z.B. für eine `Text`-Komponente mit dem Prop `numberOfLines` die maximale Anzahl von Zeilen festgelegt werden:

```
<Text numberOfLines={2}>

  Text mit mehr als zwei Zeilen wird nun abgeschnitten…

</Text>
```

Bei der Beschreibung der UI-Komponenten werde ich mich auf die relevanten oder häufig verwendeten Props beschränken. Die offizielle Dokumentation von React Native listet für alle vordefinierten UI-Komponenten die verfügbaren Props inklusive Beschreibung auf.

Hinweise zur Dokumentation von React Native

Die offizielle Dokumentation von React Native ist unter der URL *facebook.github.io/react-native/docs* erreichbar. Neben einer kurzen Einführung in React Native (*The Basics*) bietet die Dokumentation in der Kategorie *Guides* einige Artikel mit nützlichen Hintergrundinformationen zu verschiedenen Themen, wie z.B. Animationen. Die eigentliche Referenz besteht aus getrennten Kategorien für Komponenten (*Components*) und

APIs. Im Bereich *Components* werden alle in React Native enthaltenen UI-Komponenten wie `Text`, `View`, `Image` usw. inklusive Props beschrieben. Zusätzlich werden alle APIs wie z.B. `StyleSheet` in der gleichnamigen Kategorie aufgeführt (*APIs*). Zu beachten ist, dass es einige plattformspezifische Komponenten und APIs gibt, deren Name mit `Android` oder `IOS` endet (z.B. `ToolbarAndroid` und `DatePickerIOS`). Bei der Programmierung mit React Native ist die offizielle Dokumentation stets ein hilfreiches Nachschlagewerk.

Komponenten mit View zusammenfassen

Der Hauptzweck von `View`-Komponenten ist, andere Komponenten zusammenzufassen. Somit ist `View` im Prinzip eine allgemeine Containerkomponente. Diese Rolle spielt `View` häufig in der Methode `render`, deren Rückgabewert aus genau einem JSX-Element (mit möglichen Kindelementen) bestehen muss. Oft werden dort die gewünschten Komponenten mit einem äußeren `View`-Element zusammengefasst, wie im Fall der momentanen Version von *App.js* von MyJournal zu sehen ist. `View` hat im Allgemeinen kein besonderes Erscheinungsbild in der Benutzeroberfläche. In der Webentwicklung entspricht `View` sozusagen einem `div`-Element, in Android einem Widget vom Typ `android.view.View` und in iOS einem `UIView`–Objekt.

Durch das Zusammenfassen verschiedener Komponenten in einem `View`-Element ergibt sich die Möglichkeit, diese Komponenten einheitlich zu gestalten. Wie Sie im Editor in der Datei *App.js* sehen können, wird das bereits für das `View`-Element in der Methode `render` angewendet. Mit dem Attribut `style` werden verschiedene Styling-Anweisungen referenziert, die dazu führen, dass die Inhalte horizontal und vertikal zentriert werden. Da erst Kapitel 5 im Detail auf das Styling von Komponenten eingeht, werde ich in diesem Kapitel nur solche Styling-Anweisungen verwenden, die für eine grundlegende Bedienung der App benötigt werden. Entfernen Sie daher in der Zuweisung der Konstanten `styles` am Ende von *App.js* die beiden Anweisungen für `backgroundColor` und `alignItems`, sodass die Deklaration für `styles` diese Form hat:

```
const styles = StyleSheet.create({

  container: {
```

```
    flex: 1,
    justifyContent: 'center'
  }
});
```

Hierdurch wird lediglich sichergestellt, dass die Inhalte vertikal zentriert werden. Weitere Anpassungen in Bezug auf Gestaltung und Layout werden wir wie bereits erwähnt in Kapitel 5 mit ausführlicheren Erläuterungen vornehmen.

Zu Beginn beschränken wir uns in MyJournal auf das Erzeugen und die Anzeige eines einzigen Tagebucheintrags, der lediglich aus einem Textinhalt besteht. Mit der nächsten Änderung soll umgesetzt werden, dass die App uns zu Beginn mitteilt, dass es noch keine Tagebucheinträge gibt. Entfernen Sie dazu zwei der drei `Text`-Komponenten im Rückgabewert der Methode `render` und ändern Sie den Text für eine passende Anzeige wie folgt:

```
render() {
  return (
    <View style={styles.container}>
      <Text>Keine Einträge im Tagebuch</Text>
    </View>
  );
}
```

Als Nächstes werden wir die Eingabe eines Tagebucheintrags umsetzen.

Benutzereingaben mit TextInput

Für die Eingabe von Text steht in React Native die UI-Komponente `TextInput` bereit. Ändern Sie die Datei *App.js*, wie in Beispiel 4-1 aufgeführt, um eine `TextInput`-Komponente hinzuzufügen.

Um einen `TextInput` in unserer App zu verwenden, müssen wir `TextInput` zunächst importieren. Dann können wir eine konkrete `TextInput`-Komponente dem JSX-Markup in der Methode `render` hinzufügen, sodass unterhalb des bereits vorhandenen `Text`-Elements ein Texteingabefeld erscheint. Damit `TextInput` auf Android- und iOS-Geräten bedienbar ist und relativ ähnlich dargestellt wird, sollten explizite Werte für die Höhe des Eingabefelds angegeben werden. Dies haben wir mit `height: 40` im `styles`-Objekt erreicht, das wir mit einem Attribut `style={styles.input}` im `TextInput`-Element referenzieren. Weitere Anpassungen an der Darstellung dieser Komponente nehmen wir später vor. Mit dem `placeholder`-Prop deuten wir den Zweck dieser `TextInput`-Komponente durch einen leicht ausgegrauten Platzhaltertext in der App an.

Code von der Webseite zum Buch herunterladen

Wir werden die benötigten Änderungen am Code in der Regel in mehreren Teilschritten durchführen. Nach Abschluss bestimmter Entwicklungsschritte stehen Ihnen die betroffenen Dateien auf der Webseite zum Buch als Download zur Verfügung (*www.behrends.io/react-native-buch*).

Beispiel 4-1: TextInput wird importiert, und render wird um ein TextInput-Element mit zugehörigen Styles ergänzt.

```
import React from 'react';

import { StyleSheet, Text, TextInput, View } from 'react-
native';

export default class App extends React.Component {

  render() {

    return (
```

```
      <View style={styles.container}>
        <Text>Keine Einträge im Tagebuch</Text>
        <TextInput
          style={styles.input}
          placeholder="Tagebucheintrag erstellen"
        />
      </View>
    );
  }
}
const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center'
  },
  input: {
    height: 40
```

```
  },

});
```

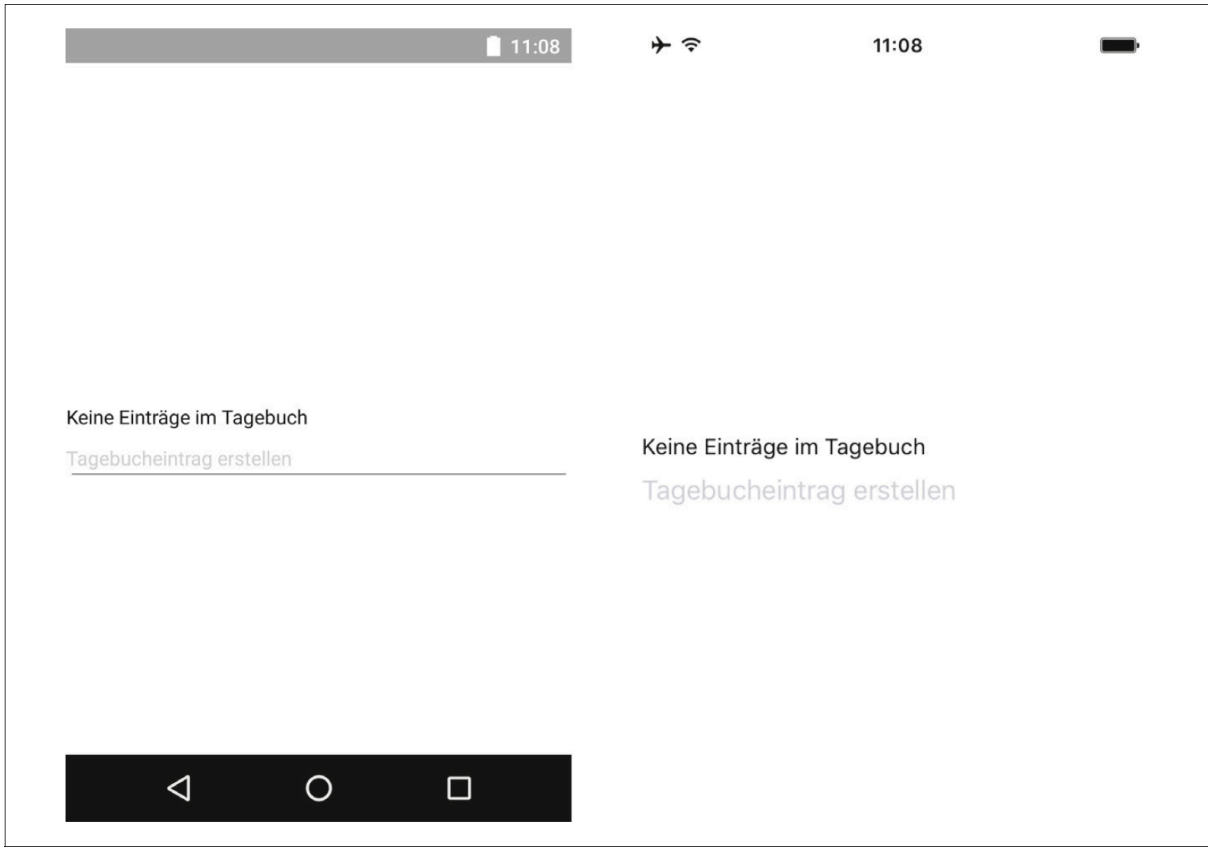

Abbildung 4-2: TextInput mit Placeholder

`TextInput`-Komponenten können auf vielfältige Weise konfiguriert werden. So werden beispielsweise bei der Eingabe automatische Korrekturen am Text durchgeführt, was sich durch die Angabe des Attributs `autoCorrect={false}` ausschalten lässt. Hier sei wieder auf die offizielle Dokumentation von React Native verwiesen, in der alle verfügbaren Props für `TextInput` beschrieben werden. Zu beachten ist dabei, dass ein Teil der Optionen plattformübergreifend anwendbar ist, einige Einstellungen sind jedoch entweder nur für Android oder nur für iOS relevant. Wir nehmen am `TextInput`-Element in `render` noch eine weitere Anpassung vor, indem wir `returnKeyType` auf `"done"` setzen:

```
  <TextInput
```

```
      style={styles.input}
      placeholder="Tagebucheintrag erstellen"
      returnKeyType="done"
    />
```

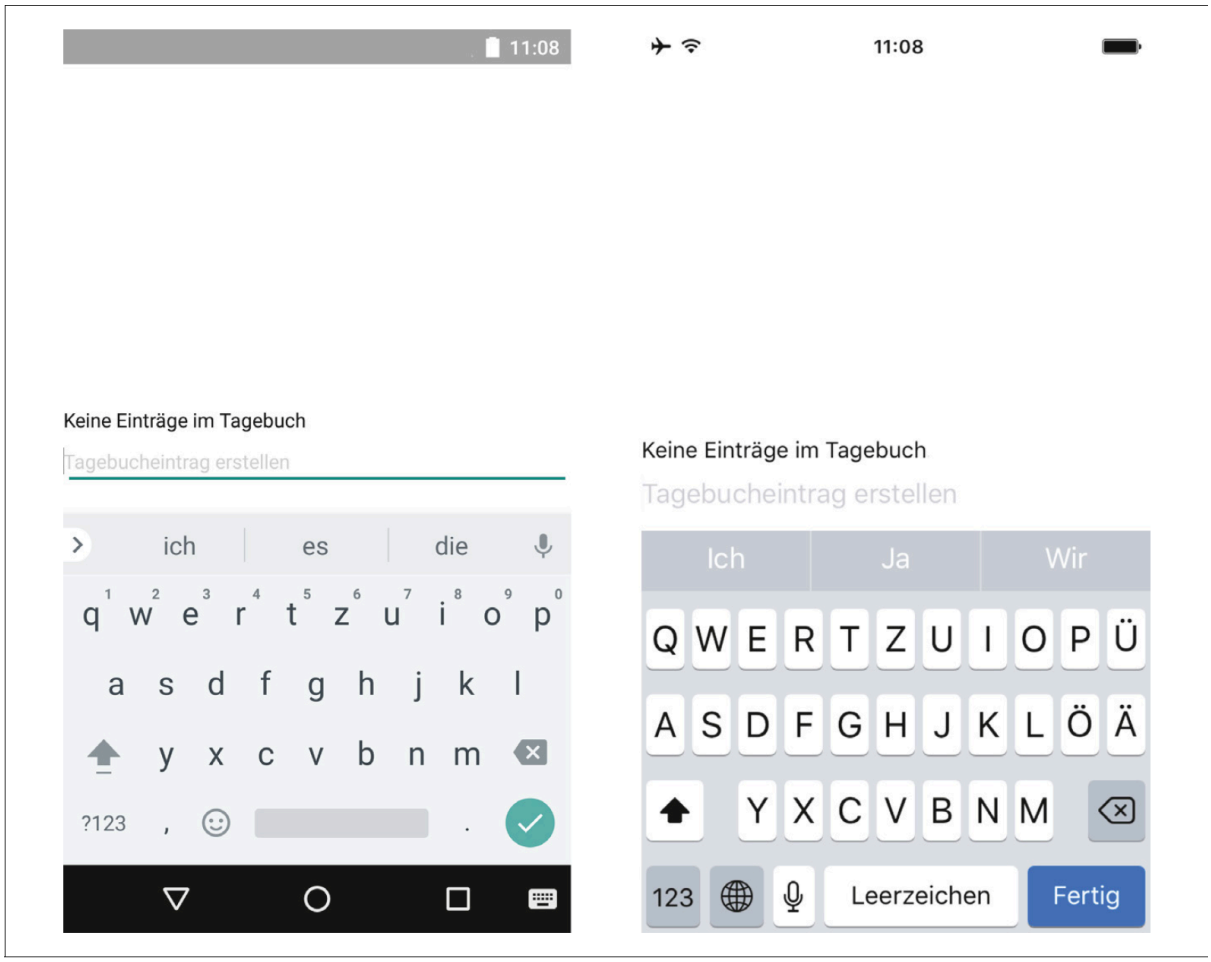

Abbildung 4-3: TextInput mit angepasster Return-Taste

Diese Einstellung wirkt sich auf die Tastatur aus, die bei der Eingabe angezeigt wird. So erscheint die Return-Taste in iOS in blauer Farbe und in der deutschen Spracheinstellung mit dem Text *Fertig*, wie in Abbildung 4-3 zu sehen ist.

Nun soll aus dem eingegebenen Text ein Tagebucheintrag erstellt werden, wenn die Eingabe mit der Return-Taste bestätigt wird. MyJournal wird jedoch zunächst nur höchstens einen Eintrag anzeigen. `TextInput` stellt einige Props zur Verfügung, mit denen unterschiedliche Callback-Methoden definiert werden

können. So können wir z.B. mit `onFocus` ein bestimmtes Verhalten implementieren, wenn die `TextInput`-Komponente aktiv wird, oder mit `onChangeText` auf Änderungen in der Eingabe reagieren. Wir möchten jetzt bei Bestätigung der Eingabe aus dem eingegebenen Text einen Tagebucheintrag erzeugen. Dazu können wir im Prop `onSubmitEditing` einen passenden Callback definieren. Erweitern Sie das `TextInput`-Element in *App.js* folgendermaßen:

```
<TextInput

  style={styles.input}

  placeholder="Tagebucheintrag erstellen"

  returnKeyType="done"

  onSubmitEditing={event =>
  console.log(event.nativeEvent.text)}

/>
```

Dem Attribut `onSubmitEditing` weisen wir eine Pfeilfunktion zu, mit der wir den Textinhalt von `TextInput`-Komponenten über ein `event`-Objekt auslesen und mit `console.log` in der Konsole ausgeben. Wenn Sie nun einen Text eingeben und die Eingabe durch Drücken der Bestätigungstaste abschließen, wird der Textinhalt in der Konsole angezeigt, in der Sie den React Native Packager gestartet haben. Durch `console.log` können wir also die korrekte Funktionsweise des Callbacks überprüfen, was gelegentlich nützlich sein wird.

Nun soll der eingegebene Text in der App als Tagebucheintrag angezeigt werden. Dazu wollen wir zunächst das `Text`-Element verwenden, das momentan statisch die Information *Keine Einträge im Tagebuch* anzeigt. Wir können für die Komponente `App` zwei mögliche Zustände unterscheiden: Entweder gibt es einen Tagebucheintrag, oder es gibt ihn nicht. Zustände einer Komponente werden in React Native (wie auch in React) durch ein Objekt namens `state` verwaltet. Diesem Objekt soll nun der vorerst einzige mögliche Tagebucheintrag als Eigenschaft zugewiesen werden. Dabei wird solch ein Eintrag lediglich

angezeigt, aber noch nicht abgespeichert, sodass beim Start von MyJournal kein Tagebucheintrag (`item`) vorhanden ist. Wir benötigen jetzt einen initialen Zustand in der Komponente `App`, was wir durch eine Initialisierung der Objekteigenschaft (bzw. *Instance Property*) `state` erreichen. Zusätzlich weisen wir im `onSubmitEditing`-Callback den Textinhalt dem `state`-Objekt mit `this.setState` zu. Der Code für alle bisherigen Änderungen ist in Beispiel 4-2 zusammengefasst und steht Ihnen zusätzlich als Download auf der Webseite zum Buch zur Verfügung (*www.behrends.io/react-native-buch*).

Beispiel 4-2: Aktueller Zwischenstand von App.js mit Zustandsinitialisierung und Verwendung von state zur Anzeige. Der Zustand wird durch onSubmitEditing in TextInput verändert.

```
// … die import-Anweisungen bleiben unverändert …

export default class App extends React.Component {

  state = { item: null };

  render() {

    return (

      <View style={styles.container}>

        <Text>{this.state.item || 'Keine Einträge im
        Tagebuch'}</Text>

        <TextInput

          style={styles.input}

          placeholder="Tagebucheintrag erstellen"

          returnKeyType="done"
```

```
          onSubmitEditing={event =>
            this.setState({ item: event.nativeEvent.text })}
        />
      </View>
    );
  }
}

// … die Styles bleiben unverändert …
```

Die Initialisierung der Objekteigenschaft `state` findet bei Erzeugung der Komponente vor Ausführung von `render` statt. Wir setzen den initialen Zustand auf den Wert `{ item: null }`, denn zu Beginn ist noch kein Tagebucheintrag vorhanden.

Der Callback `onSubmitEditing` wird zur Laufzeit nach Bestätigung der Eingabe durch die Tastatur mit einem `event`-Objekt aufgerufen. Dieses Objekt enthält in `event.nativeEvent.text` den eingegebenen Text. Mit `this.setState` wird der Wert für `item` im `state`-Objekt auf den eingegebenen Text gesetzt.

Im JSX-Code wird der Inhalt von `this.state.item` in der `Text`-Komponente verwendet. Falls `this.state.item` nicht definiert ist (z.B. null ist) und somit in diesem logischen Ausdruck `false` ergibt, erscheint wie zu Beginn der Text *Keine Einträge* im Tagebuch.

Testen Sie diese Änderung und geben Sie verschiedene Texte ein. Bei jeder durch die Tastatur bestätigten Änderung wird das `state`-Objekt im Callback von `onSubmitEditing` geändert, was eine erneute Ausführung von `render` und die aktualisierte Darstellung bewirkt. Somit können Sie den Tagebucheintrag ändern. Wenn Sie den Inhalt der `TextInput`-Komponente entfernen und die Eingabe bestätigen, wird der Tagebucheintrag gelöscht, und *Keine Einträge im Tagebuch* wird angezeigt.

Übung

Passen Sie das Verhalten der App so an, dass sich der Tagebucheintrag direkt bei der Eingabe des Texts ändert und nicht erst dann, wenn die Eingabe bestätigt wird. Dadurch lernen Sie mit `onChangeText` einen weiteren häufig gebrauchten Prop mit Callback der Komponente `TextInput` kennen. Diese Änderung sollten Sie rückgängig machen, wenn Sie mit dem nächsten Abschnitt beginnen.

Einfache Listen mit FlatList

Bisher haben wir den einzigen möglichen Tagebucheintrag mit einer `Text`-Komponente dargestellt. Für eine Liste mit beliebig vielen Einträgen benötigen wir nun eine andere Komponente. Für den häufig vorkommenden Anwendungsfall, Listen von Daten in Smartphone-Apps darzustellen, stellt React Native verschiedene Komponenten zur Verfügung: `FlatList`, `SectionList` und `VirtualizedList`.

`FlatList`

: `FlatList` ist eine Komponente zur Darstellung vertikaler oder horizontaler Listen bestehend aus einzelnen Elementen ohne Unterteilung. Daher ist solch eine Liste sozusagen »flach« (*Flat*), woran sich der Name anlehnt.

`SectionList`

: Falls eine Liste aus Abschnitten (*Sections*) bestehen soll, in denen Einträge gruppiert werden, bietet sich der Einsatz einer `SectionList` an.

`VirtualizedList`

: Mit `VirtualizedList` steht eine allgemeine Komponente zur Darstellung von Listen zur Verfügung, auf deren Basis `FlatList` und `SectionList` implementiert sind. `VirtualizedList` ist vielseitig konfigurierbar, und ihre Verwendung kommt in Betracht, wenn die anderen beiden Listenkomponenten den Anforderungen nicht genügen.

Bemerkung zu ListView

Seit Version 0.43 sind `FlatList`, `SectionList` und `VirtualizedList` die bevorzugten Komponenten für Listen in React Native. In früheren Versionen und älteren Beispielen wurde die Komponente `ListView` eingesetzt, die inzwischen nicht mehr weiterentwickelt wird und *deprecated* ist.

Wir werden in unserer App eine Liste der Tagebucheinträge zunächst mit einer `FlatList` und später mit einer `SectionList` implementieren, damit Sie diese beiden Listenkomponenten kennenlernen.

Um eine Liste mit `FlatList` in einer App anzuzeigen, werden in der einfachsten Variante nur die Daten der Liste benötigt, und es muss eine Funktion definiert werden, die die Darstellung eines Listenelements implementiert:

```
<FlatList

  data={[{ key: 1, text: 'one' }, { key: 2, text: 'two' }]}

  renderItem={({ item }) => <Text>{item.text}</Text>}

/>
```

Mit dem Attribut `data` legen wir die Daten der Liste als Array fest. Hier enthält das Array zur Veranschaulichung nur zwei Objekte. Jedes Objekt in dem Array sollte durch eine Eigenschaft identifizierbar sein, die somit als eindeutiger Schlüssel dient. Ist in jedem Objekt eine Eigenschaft `key` enthalten, fasst React Native diese Eigenschaft als Schlüssel der Objekte auf. React Native benötigt diese, um die Funktionsweise der Liste zu gewährleisten, wenn sich z.B. die Reihenfolge der Listenelemente ändert. Alternativ kann bei der Deklaration von Listenkomponenten über das Attribut `keyExtractor` eine Eigenschaft explizit als Schlüssel festgelegt werden. Es führt zu einer Warnung, wenn Objekte keinen Schlüssel haben.

Mit `renderItem` steht ein Prop zur Verfügung, dem eine Funktion zugewiesen wird, die die Darstellung eines Objekts in der Liste übernimmt. Zur Laufzeit repräsentiert das Argument dieser Funktion das darzustellende Objekt (im Beispiel `item` genannt), und die Funktion liefert als Rückgabewert eine Komponente zurück, die in der Liste als Eintrag für das Objekt erscheint. Hier definieren wir eine `Text`-Komponente mit `item.text` als Inhalt.

`FlatList` funktioniert als performante Listenkomponente plattformübergreifend für Android und iOS. Es bestehen viele Möglichkeiten zur Konfiguration einer Liste – beispielsweise können die Deklaration von Kopf- und Fußzeilen und auch komplexeres Verhalten wie »Pull to Refresh« mit relativ wenig Aufwand implementiert werden.

Nun werden wir MyJournal um eine Liste für die textbasierten Tagebucheinträge erweitern. Um `FlatList` zu verwenden, müssen wir die entsprechende Komponente importieren. Fügen Sie also `FlatList` der `import`-Anweisung am Anfang der Datei *App.js* hinzu:

```
import { FlatList, StyleSheet, Text, TextInput, View } from
'react-native';
```

Da wir eine Liste der Tagebucheinträge verwalten möchten, ändern wir die Initialisierung des `state`-Objekts so, dass ein leeres Array zugewiesen wird. Hier sollen alle Einträge verwaltet werden. Wir benennen also die Eigenschaft `item` des `state`-Objekts in `items` um:

```
export default class App extends React.Component {

  state= {items:[]};

  // … der Rest bleibt unverändert …
```

Erweitern Sie nun die Funktion `render` mit dem Einsatz von `FlatList`, wie in Beispiel 4-3 beschrieben. Beachten Sie, dass sich diese Änderung noch nicht auf die Darstellung der App auswirkt.

Beispiel 4-3: Fügen Sie in render den Codeabschnitt ab »let content« vor der return-Anweisung ein.

```
render() {

  let content = <Text>Keine Einträge im Tagebuch</Text>;

  if (this.state.items.length > 0) {

    content = (

      <FlatList
```

```
        style={styles.list}
        data={this.state.items}
        renderItem={({ item }) => <Text>{item.text}</Text>}
        keyExtractor={item => item.date}
      />
    );
  }
  return (
    <View style={styles.container}>
    // … der Rest von render bleibt unverändert
```

Mit der Variablen `content` und der `if`-Verzweigung werden nun die zwei möglichen Zustände der Komponente behandelt (Tagebucheinträge sind vorhanden oder nicht). Die Variable `content` wird zunächst mit einer `Text`-Komponente initialisiert, die dargestellt werden soll, falls die Liste der Tagebucheinträge leer ist. Gibt es jedoch Listeneinträge, was in der Bedingung der `if`-Verzweigung durch `this.state.items.length > 0` überprüft wird, dann wird `content` eine `FlatList`-Komponente zugewiesen, die für die Darstellung der Objekte in `this.state.items` als Liste zuständig ist. Die darzustellenden Objekte werden durch den Prop `data` übergeben, und mit `renderItem` wird festgelegt, dass ein Listeneintrag aus einer `Text`-Komponente besteht. Mit der Funktion in `keyExtractor` wird die Eigenschaft `date` eines Eintrags als Schlüssel deklariert. Wie wir gleich sehen werden, wird bei der Erstellung eines Eintrags der aktuelle Zeitpunkt in `date` festgehalten.

Außerdem haben wir für die `FlatList`-Komponente durch das Attribut `style={styles.list}` einen Style definiert, damit diese Komponente durch einen kleinen Abstand besser sichtbar ist. Dazu wird das `styles`-Objekt um einen Eintrag ergänzt (z.B. vor der Eigenschaft `input`):

```
const styles = StyleSheet.create({

  // … die restlichen Styles bleiben unverändert …

  list: {

    marginTop: 24

  },

  input: {

    height: 40

  }

});
```

Ersetzen Sie jetzt den Rückgabewert bzw. die `return`-Anweisung der Funktion `render` mit folgendem Ausdruck:

```
return (

  <View style={styles.container}>

    {content}

    <TextInput
```

```
        style={styles.input}

        placeholder="Tagebucheintrag erstellen"

        returnKeyType="done"

        onSubmitEditing={event =>
        this._addItem(event.nativeEvent.text)}

    />

  </View>
```

Innerhalb des äußeren `View`-Elements wird zuerst durch die Anweisung `{content}` der Inhalt der Variablen `content` dargestellt. Es wird also entweder die `FlatList` mit den Tagebucheinträgen oder die `Text`-Komponente mit dem Hinweis auf nicht vorhandene Einträge erscheinen. Die Callback-Funktion in `onSubmitEditing` ruft nun eine Hilfsfunktion `_addItem` auf, die für das Hinzufügen eines Eintrags in die Liste der Tagebucheinträge zuständig ist. Definieren Sie diese Funktion in der Klasse `App` z.B. direkt vor der Methode `render`:

```
  _addItem(text) {

    this.setState({

      items: […this.state.items, { text, date: Date.now() }]

    });

  }
```

Der Funktionsname beginnt mit einem Unterstrich, um anzudeuten, dass dies eine Hilfsfunktion ist, die insbesondere keine Implementierung einer Funktion des Frameworks React Native ist (wie z. B. `render`). Durch den Aufruf von

`this.setState` aktualisieren wir den Zustand der Komponente so, dass das Array in `items` mit einem neuen Eintrag erweitert wird. Mit der Anwendung des Spread-Operators (…) auf `this.state.items` fügen wir die bestehenden Einträge ein. Für den neuen Eintrag wird ein Objekt erstellt, das den eingegebenen Text in `text` und den aktuellen Zeitpunkt in `date` enthält. Für die Deklaration der Objekteigenschaft `text` verwenden wir hier die Kurzschreibweise, die eigentlich für `text: text` steht. Mit `Date.now()` erhalten wir in JavaScript den aktuellen Zeitpunkt als Unix-Zeit, also die Anzahl der Millisekunden, die seit dem 1. Januar 1970 um 00:00 Uhr vergangen sind. Hierbei handelt es sich um einen numerischen Wert, der leicht in verschiedene Datumsformate umgewandelt werden kann.

Für jeden erstellten Eintrag wird somit der aktuelle Zeitpunkt festgehalten. Da dieser Wert für alle Objekte eindeutig ist, haben wir bereits bei der Deklaration der `FlatList` mit `keyExtractor` die Eigenschaft `date` als Schlüssel der Listeneinträge festgelegt (siehe Beispiel 4-3). Zusätzlich werden wir später für Einträge die Uhrzeit ihrer Erstellung anzeigen. Die Verwendung von `this.setState` führt zu einer Neudarstellung der Komponente, weil durch die Zustandsänderung implizit `render` aufgerufen wird. Der aktuelle Zwischenstand von *App.js* ist in Beispiel 4-4 aufgelistet und kann auch von der Webseite zum Buch heruntergeladen werden (*www.behrends.io/react-native-buch*).

Beispiel 4-4: Zwischenstand von App.js mit Verwendung einer FlatList

```
import React from 'react';

import { FlatList, StyleSheet, Text, TextInput, View } from
'react-native';

export default class App extends React.Component {

  state= {items:[]};

  _addItem(text) {

    this.setState({
```

```
      items: […this.state.items, { text, date: Date.now() }]
    });
  }
  render() {
    let content = <Text>Keine Einträge im Tagebuch</Text>;
    if (this.state.items.length > 0) {
      content = (
        <FlatList
          style={styles.list}
          data={this.state.items}
          renderItem={({ item }) => <Text>{item.text}</Text>}
          keyExtractor={item => item.date}
        />
      );
    }
    return (
      <View style={styles.container}>
```

```
        {content}

        <TextInput

          style={styles.input}

          placeholder="Tagebucheintrag erstellen"

          returnKeyType="done"

          onSubmitEditing={event =>
          this._addItem(event.nativeEvent.text)}

        />

      </View>

    );

  }

}

const styles = StyleSheet.create({

  container: {

    flex: 1,

    justifyContent: 'center'

  },

  list: {
```

```
    marginTop: 24
  },
  input: {
    height: 40
  }
});
```

Wenn Sie diese Änderungen testen, können Sie bereits mehrere Einträge für das Tagebuch erzeugen, die in einer Liste oberhalb des Texteingabefelds erscheinen. Die Gestaltung der Benutzeroberfläche haben wir bewusst einfach gehalten, da der Schwerpunkt bisher auf der grundlegenden Funktionsweise der App liegt.

Auffallend ist, dass die `TextInput`-Komponente nach der ersten Eingabe nicht mehr in der Mitte des Screens erscheint, sondern am unteren Rand. Das liegt an der `FlatList`-Komponente, die sich in ihrer Grundkonfiguration maximal in horizontale und vertikale Richtung ausdehnt und somit den `TextInput` nach unten drängt. Diese gestalterische Inkonsistenz nehmen wir vorerst in Kauf und werden sie später bereinigen.

Wenn Sie nun mehrere Einträge nacheinander erstellen, werden Ihnen allerdings zwei Probleme auffallen. Einerseits überdeckt die Tastatur das Eingabefeld, sodass Sie nicht sehen können, was Sie eintippen. Außerdem ist das Feld noch mit dem vorigen Eintrag befüllt, der erst manuell entfernt werden muss. Daher müssen wir uns nochmals mit der `TextInput`-Komponente befassen.

Bedienbarkeit von TextInput verbessern

Wir werden zunächst mit der Komponente `KeyboardAvoidingView` gewährleisten, dass das Texteingabefeld bei der Eingabe von Tagebucheinträgen sichtbar bleibt und nicht von der Tastatur überdeckt wird. Anschließend werden wir den Inhalt von `TextInput` nach erfolgter Eingabe leeren, um nachfolgende

Eingaben zu erleichtern. Dabei lernen wir das Konzept von Referenzen in React Native kennen.

Sichtbarkeit mit KeyboardAvoidingView gewährleisten

Wenn in MyJournal Tagebucheinträge in der `FlatList` angezeigt werden, erscheint das Texteingabefeld am unteren Bildschirmrand. Wird dieses Feld ausgewählt, um einen weiteren Eintrag zu erstellen, überdeckt die Tastatur die `TextInput`-Komponente, und wir können nicht sehen, welchen Text wir über die Tastatur eintippen. Es gibt eine spezielle Komponente, deren einzige Aufgabe darin besteht, dafür zu sorgen, dass andere Komponenten auch nach Erscheinen der Tastatur sichtbar bleiben. Diese Komponente hat den bezeichnenden Namen `KeyboardAvoidingView`. Wir erweitern die `import`-Anweisung in *App.js* mit `KeyboardAvoidingView`, um diese Komponente in MyJournal einsetzen zu können:

Beispiel 4-5: KeyboardAvoidingView importieren

```
import {
  FlatList,
  KeyboardAvoidingView,
  StyleSheet,
  Text,
  TextInput,
  View
} from 'react-native';
```

Die `import`-Anweisung enthält nun einige zu importierende Komponenten. Für eine bessere Übersicht geben wir hier nur noch eine Komponente pro Zeile an.

JavaScript-Code automatisch formatieren

Um eine einheitliche Codebasis zu erhalten, ist es praktisch und zeitsparend, wenn beim Speichern einer Datei im Editor der syntaktische Aufbau des Codes automatisch formatiert wird. Einige Editoren bieten diese Funktionalität für verschiedene Programmiersprachen an. Beispielsweise kann festgelegt werden, dass Codeausdrücke ab einer bestimmten Zeilenlänge auf mehrere Zeilen umbrochen werden, wie in Beispiel 4-5 für die `import`-Anweisung zu sehen. Im JavaScript-Umfeld hat sich in letzter Zeit der Formatierer *Prettier* (*www.prettier.io*) zu einer beliebten Lösung entwickelt. Prettier ist sehr performant, vielseitig konfigurierbar, kann in viele Editoren integriert werden und funktioniert gut mit den Besonderheiten von React Native wie etwa JSX. Auch die Codebeispiele in diesem Buch wurden mit Prettier formatiert.

Der Einsatz von `KeyboardAvoidingView` erfordert nicht viel Aufwand. Die Komponenten, die nicht von der Tastatur überdeckt werden sollen, werden im JSX-Ausdruck in ein äußeres `KeyboardAvoidingView` eingebettet, das heißt, wir erweitern den Rückgabewert der Methode `render` entsprechend. Dies wird durch die Änderungen in Beispiel 4-6 ermöglicht.

Beispiel 4-6: KeyboardAvoidingView im JSX-Code der Methode render

```
// … der Rest in render bleibt unverändert …

return (

  <View style={styles.container}>

    {content}

    <KeyboardAvoidingView behavior="padding">

      <TextInput

        style={styles.input}
```

```
        placeholder="Tagebucheintrag erstellen"
        returnKeyType="done"
        onSubmitEditing={event =>
          this._addItem(event.nativeEvent.text)}
      />
    </KeyboardAvoidingView>
  </View>
);
```

Durch `behavior="padding"` legen wir fest, dass die Sichtbarkeit von `TextInput` durch einen automatisch berechneten Abstand zum unteren Rand erreicht wird. Wenn die Tastatur erscheint, wird nun `TextInput` oberhalb der Tastatur erscheinen. Alternative Werte für den Prop `behavior` sind `height` und `position`, womit durch die Höhe bzw. die Position der eingebetteten Elemente die Sichtbarkeit oberhalb der Tastatur gewährleistet wird. Je nach Anwendungsfall und betroffenen Komponenten kann ein passender Wert für `behavior` gewählt werden.

Referenzen auf Komponenten mit ref setzen

Wollen wir mehrere verschiedene Texte nacheinander eingeben, müssen wir jedes Mal zuvor den Inhalt des Eingabefelds entfernen, der von der vorherigen Eingabe stammt. Die Bedienbarkeit der App würde sich verbessern, wenn der Inhalt von `TextInput` automatisch gelöscht würde, nachdem ein neuer Eintrag hinzugefügt wurde. Dazu bietet die Komponente `TextInput` eine Methode `clear` an, die wir z.B. am Ende von `_addItem` aufrufen könnten, denn diese Methode dient als Callback für die erfolgte Eingabe. Unten wird dies durch einen Kommentar angedeutet:

```
  _addItem(text) {

    this.setState({

      items: […this.state.items, { text, date: Date.now() }]

    });

    // Aufruf von TextInput.clear() - aber wie?

  }
```

Das Programmiermodell von React (Native) ist im Wesentlichen deklarativ und basiert auf Komponenten. Das wird vor allem in der Methode `render` deutlich, in der lediglich die Darstellung der Komponente im UI durch JSX-Code beschrieben wird. In der Regel arbeiten wir also nicht mit konkreten Instanzen von Komponenten. Für den eben beschriebenen Anwendungsfall steht uns in `_addItem` keine Instanz der `TextInput`-Komponente zur Verfügung, auf der wir die Methode `clear` aufrufen könnten. React bietet jedoch durch das Konzept der Referenzen (Refs) einen Ausweg für solche Situationen, der uns auch in React Native bereitsteht.[1]

Zunächst deklarieren wir eine Referenz auf die konkrete `TextInput`-Instanz im UI, was durch ein Attribut namens `ref` erreicht wird. Beispiel 4-7 zeigt Ihnen, wie Sie den JSX-Code des `TextInput`-Elements im Rückgabewert der Methode `render` mit einem Attribut `ref` erweitern.

Beispiel 4-7: Das TextInput-Element erhält ein Attribut namens ref, womit diese Komponente im Code explizit referenziert werden kann.

```
// … der Rest in render bleibt unverändert …

return (

  <View style={styles.container}>
```

```
    {content}
    <KeyboardAvoidingView behavior="padding">
      <TextInput
        style={styles.input}
        ref={input => (this.textInput = input)}
        placeholder="Tagebucheintrag erstellen"
        returnKeyType="done"
        onSubmitEditing={event =>
          this._addItem(event.nativeEvent.text)}
      />
    </KeyboardAvoidingView>
  </View>
);
```

Die Referenz wird im Attribut `ref` durch eine Callback-Funktion gesetzt, die nach Darstellung der `TextInput`-Instanz im UI ausgeführt wird und als Argument die konkrete Komponente erhält (hier `TextInput`).[2] Dadurch steht der Komponente `App` zur Laufzeit eine Referenz namens `textInput` zur Verfügung, mit der wir für die konkrete `TextInput`-Komponente Methoden aufrufen können. Mit `this.textInput.clear()` in der Funktion `_addItem` können wir nun nach dem Hinzufügen eines neuen Eintrags zur Liste den Inhalt des Eingabefelds leeren. Ändern Sie `_addItem` folgendermaßen:

```
  _addItem(text) {

    this.setState({

      items: […this.state.items, { text, date: Date.now() }]

    });

    this.textInput.clear();

  }
```

Nun sollte die Bedienbarkeit des Eingabefelds deutlich besser sein. Zu beachten ist, dass Referenzen auf Komponenten mithilfe des `ref`-Attributs nur in seltenen Fällen verwendet werden sollten. Der Datenfluss zu Komponenten sollte in der Regel über Props erfolgen.

Reservierte Namen für Props

In React (und somit auch in React Native) gibt es zwei besondere Props, die für die interne Funktionsweise von React-Komponenten benötigt werden und die nicht für eigene Zwecke verwendet werden sollten. Dies sind `key` und `ref`. Wenn also an eine eigene Komponente z.B. mit `ref="myvalue"` eine Information übermittelt werden soll, kann auf diese nicht innerhalb der Komponente mit `this.props.ref` zugegriffen werden (Gleiches gilt für `this.props.key`). Es wird eine Warnung erscheinen, und daher sollten andere Bezeichner gewählt werden.

SectionList für Listen mit Abschnitten

Wie wir eben gesehen haben, lassen sich Listen ohne viel Aufwand mit `FlatList` umsetzen. Somit werden in MyJournal die Tagebucheinträge in einer Liste untereinander dargestellt. Im Laufe der Zeit kann diese Liste bei vielen Einträgen lang und unübersichtlich werden. Vor allem wollen wir sicher schnell erkennen können, an welchem Tag ein bestimmter Eintrag erstellt wurde. Da wir mehrere Einträge an einem Tag schreiben können, würde eine bessere Übersichtlichkeit dadurch erreicht werden, wenn alle Einträge in deutlich sichtbaren Abschnitten angezeigt werden.

Solch eine gruppierte Darstellung von Elementen in einer Liste ist ein Anwendungsfall, der häufiger in mobilen Apps auftritt. Ein Beispiel hierfür sind die Apps für Kontakte in Android und iOS, bei denen Kontakte mit gleichem Anfangsbuchstaben im selben Abschnitt zusammengefasst werden. Für dieses Szenario bietet React Native eine Komponente namens `SectionList` an. Wir werden in MyJournal eine `SectionList` verwenden, um alle Einträge eines Tages in einem Abschnitt einzuordnen.

Im Prinzip ist `SectionList` eine Erweiterung von `FlatList` um Abschnitte (*Sections*) zur Gruppierung. Wie in `FlatList` benötigen wir eine Funktion zur Darstellung eines Eintrags im Prop `renderItem`, und zusätzlich wird mit einer Funktion in `renderSectionHeader` definiert, wie die Überschrift eines Abschnitts erscheinen soll. Schließlich wird anstelle von `data` ein Prop `sections` für die Daten der einzelnen Abschnitte angegeben. Dies wird durch das Codefragment in Beispiel 4-8 angedeutet, das zu einer Darstellung wie der in Abbildung 4-4 führen würde.

Beispiel 4-8: Die Daten einer SectionList werden in sections durch ein Array definiert. Jeder Abschnitt (section) wird durch ein Objekt repräsentiert, wobei die einzelnen Zeilen pro section in data aufgeführt werden.

```
<SectionList

  sections={[

    {

      data: [{ key: 1, text: 'one' }, { key: 2, text: 'two'
      }],

      title: 'first'

    },

    {

      data: [{ key: 3, text: 'three' }],
```

```
      title: 'second'

    },

    {

      data: [{ key: 4, text: 'four' }, { key: 5, text: 'five'
      }],

      title: 'third'

    }

  ]}

  renderItem={({ item }) => <Text>{item.text}</Text>}

  renderSectionHeader={({ section }) => <Text>{section.title}
  </Text>}

/>
```

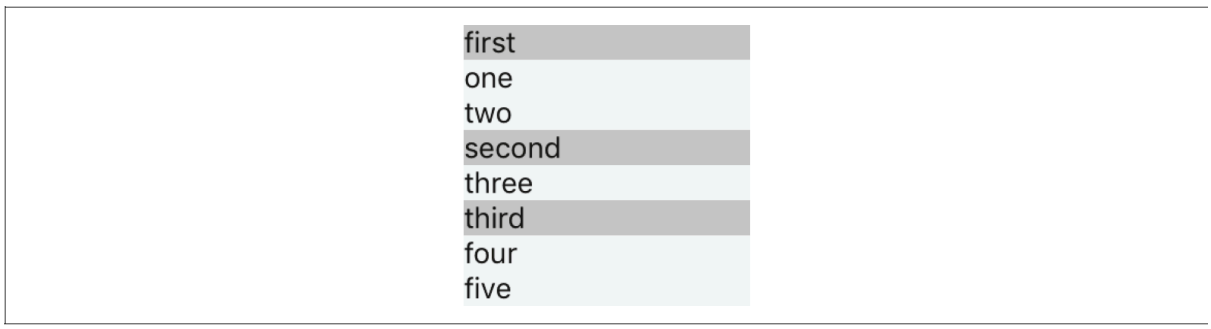

Abbildung 4-4: Darstellung der SectionList aus Beispiel 4-8

Die Datenstruktur für `sections` ist ein Array, das aus Objekten besteht, die jeweils einen Abschnitt repräsentieren. Jedes dieser »Abschnittsobjekte« sollte eine Überschrift für den Abschnitt bereitstellen (hier z.B. `title`). Jeder Abschnitt enthält in `data` ein Array mit den Einträgen für diesen Abschnitt. Es handelt sich

bei `sections` sozusagen um eine geschachtelte oder zweidimensionale Datenstruktur. Jedes Objekt in den `data`-Arrays muss über alle Einträge in `sections` eindeutig identifizierbar sein. Wie bei `FlatList` kann das durch `key`-Eigenschaften oder durch Deklaration einer Schlüsseleigenschaft in `keyExtractor` erreicht werden. Die Funktion in `renderSectionHeader` ist dafür zuständig, eine Komponente für die Überschrift eines Abschnitts zu liefern. Dabei steht ihr ein Objekt als Argument zur Verfügung, das einen Abschnitt repräsentiert. Hier verwenden wir den Wert in `title` für die Überschrift des Abschnitts.

In MyJournal sollen die Einträge pro Tag gruppiert werden. Damit wir bereits beim Start von MyJournal einige Daten sehen, deklarieren wir eine Konstante `journalItems` mit beispielhaften Einträgen für zwei vergangene Tage im Juli 2017 und weisen diese als vorübergehende Lösung dem initialen Zustand im `state`-Objekt zu. Außerdem müssen wir in der `import`-Anweisung `FlatList` durch `SectionList` ersetzen. Der Code in Beispiel 4-9 zeigt Ihnen die relevanten Änderungen.

Beispiel 4-9: SectionList wird anstelle von FlatList importiert. Einer Konstanten journalItems werden Beispieldaten zugewiesen, die vorübergehend im initialen Zustand verwendet werden.

```
import React from 'react';

import {

  KeyboardAvoidingView,

  SectionList,

  StyleSheet,

  Text,

  TextInput,

  View
```

```
} from 'react-native';

const journalItems = [

  {

    data: [

      {

        text: 'Umgang mit SectionList in React Native
        gelernt',

        date: 1 // eindeutiger, willkürlicher Wert für date

      }

    ],

    title: '29.7.2017'

  },

  {

    data: [

      { text: 'Einkauf im Supermarkt', date: 2 },

      { text: 'Wochenendausflug geplant', date: 3 }

    ],
```

```
    title: '28.7.2017'
  }
];
export default class App extends React.Component {
  state = { items: journalItems };
  // … der Rest bleibt zunächst unverändert …
```

In `render` wird nun eine `SectionList` benötigt, die ihre Daten via `sections` anstatt über `data` erhält und die einen zusätzlichen Prop `renderSectionHeader` erfordert. Die Deklaration des `SectionList`-Elements im JSX-Code ist in Beispiel 4-10 dargestellt.

Beispiel 4-10: In render verwenden wir nun SectionList anstatt FlatList.

```
render() {
  let content = <Text>Keine Einträge im Tagebuch</Text>;
  if (this.state.items.length > 0)
    content = (
      <SectionList
        style={styles.list}
        sections={this.state.items}
        renderItem={({ item }) => <Text>{item.text}</Text>}
```

```
        renderSectionHeader={({ section }) => (

          <Text style={styles.listHeader}>{section.title}
          </Text>

        )}

        keyExtractor={item => item.date}

      />

    );

// … der Rest bleibt zunächst unverändert …
```

Der `return`-Ausdruck von `render` ändert sich nicht. Damit die Abschnittsüberschriften visuell hervorgehoben werden, verwenden wir in der `Text`-Komponente in `renderSectionHeader` einen einfachen Style, für den wir eine graue Hintergrundfarbe festlegen. Ergänzen Sie dazu das `styles`-Objekt am Ende von *App.js* mit der folgenden Deklaration:

```
  const styles = StyleSheet.create({

    // … die restlichen Styles bleiben unverändert …

    listHeader: {

      backgroundColor: 'darkgray'

    }

  };
```

Die deklarative Nutzung einer `SectionList`-Komponente erfordert nur wenige Änderungen im Vergleich zu `FlatList`. Lediglich die Anwendungslogik zur Verwaltung der geschachtelten Datenstruktur, die den Abschnitten einer `SectionList` zugrunde liegt, bringt einen erhöhten Aufwand mit sich. Dies zeigt sich im angepassten Code für die Methode `_addItem`, der in Beispiel 4-11 aufgelistet ist. Ändern Sie diese Methode in *App.js*, damit ein neuer Tagebucheintrag am Anfang des Abschnitts für den aktuellen Tag erscheint, wie in Abbildung 4-5 dargestellt. Auf der Webseite zum Buch können Sie die Datei *App.js* für den momentanen Zwischenstand herunterladen.

Beispiel 4-11: Die Methode _addItem fügt einen Tagebucheintrag in die Datenstruktur ein, die in der SectionList verwendet wird.

```
_addItem(text) {

  let { items } = this.state;

  let [head, …tail] = items; // head enthält ersten Abschnitt

  // Datum für heute schrittweise im Format 22.6.2017 aufbauen

  const now = new Date();

  const day = now.getDate();

  const month = now.getMonth() + 1;

  const year = now.getFullYear();

  const today = `${day}.${month}.${year}`; // heutiges Datum

  if (head === undefined || head.title !== today) {

    // ggf. neuen Abschnitt für heutiges Datum erstellen
```

```
      head = { data: [], title: today };

      tail = items;

    }

    // neuen Eintrag (newItem) an vorderster Stelle einfügen

    const newItem = { text: text, date: now.getTime() };

    head.data = [newItem, …head.data];

    items = [head, …tail];

    this.setState({ items });

    this.textInput.clear();

  }
```

Am Anfang werden zwei destrukturierende Zuweisungen ausgeführt: Zuerst initialisieren wir die Variable `items` mit den aktuellen Einträgen aus `state`, dann lesen wir das erste Element der Abschnitte aus `items` in eine Variable `head` ein und weisen der Variablen `tail` mithilfe des Spread-Operators (…) die restlichen Abschnitte zu.

Die Überschrift eines Abschnitts stellt den Tag in einem schlichten Format dar (z.B. `22.6.2017` für den 22. Juni 2017). Dafür wird der String in der Konstanten `today` verwendet, der sich aus den Bestandteilen für den Tag, den Monat und das Jahr des aktuellen Datums in der Konstanten `now` zusammensetzt. Hierbei ist zu beachten, dass die zwölf Monate eines Jahres in JavaScript von der Methode `getDate` mit `0` für Januar bis `11` für Dezember repräsentiert werden. Daher wird die Konstante `month` mit `now.getMonth() + 1` berechnet.

Falls es noch keinen Abschnitt für den heutigen Tag gibt (siehe `if`-Verzweigung), wird dieser mit einem leeren Array in `data` erstellt, und `tail` werden die gesamten Abschnitte aus `items` zugewiesen. Nach dem `if`-Block erstellen wir

den neuen Eintrag (`newItem`) bestehend aus dem eingegebenen Text und dem aktuellen Zeitpunkt (numerischer Wert der Unix-Zeit). Danach wird `head.data` mit `newItem` erweitert. Der Variablen `items` wird die Liste der Abschnitte bestehend aus `head` und `tail` zugewiesen, und schließlich werden die geänderten Daten für `items` im `state`-Objekt mit `setState` gesetzt.

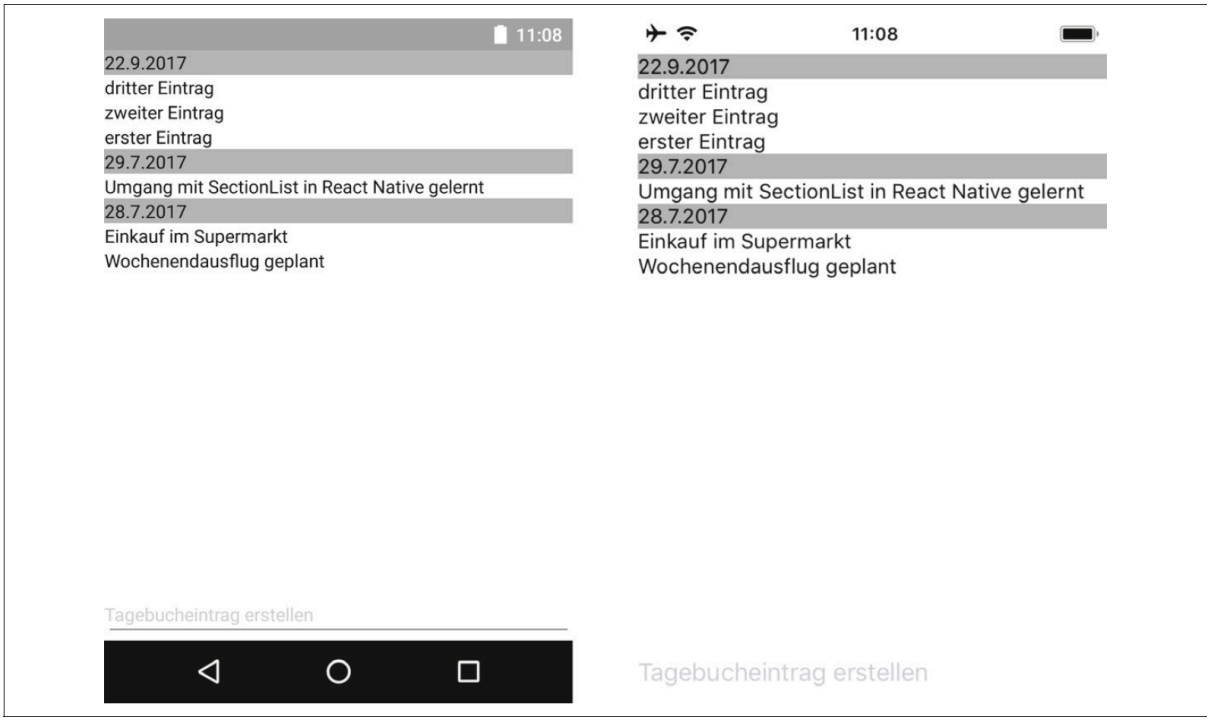

Abbildung 4-5: Tagebucheinträge werden mit einer SectionList in Abschnitten für jeden Tag dargestellt. Die Reihenfolge der Einträge ist absteigend nach Datum sortiert.

Hilfsbibliotheken für Date

Da sich der Umgang mit `Date` in JavaScript als recht umständlich erweist, kann es in Projekten mittelfristig sinnvoll sein, eine Hilfsbibliothek für `Date` zu verwenden. Insbesondere wenn bei Datumsobjekten Zeitzonen berücksichtigt und Darstellungen in verschiedenen Formaten oder Sprachen benötigt werden, kann eine Hilfsbibliothek eine große Erleichterung sein. Im JavaScript-Umfeld gibt es mehrere Alternativen, z.B. die Bibliotheken *date-fns* (*date-fns.org*) und *Moment.js* (*momentjs.com*).

Wenn die App im Alltag verwendet wird, könnte sich herausstellen, dass meistens nicht mehr als ein Eintrag pro Tag erstellt wird. In dem Fall könnte es passender sein, die `SectionList` auf Basis von Monaten anstelle von Tagen zu unterteilen. Da wir jedoch im Rahmen dieses Buchs einen Prototyp der App mit

häufigen Änderungen entwickeln, ist die Verwendung von Tagen zur Unterteilung naheliegender. Sonst müssten wir stets bis zum nächsten Monat warten, damit mehr als eine Unterteilung in der `SectionList` sichtbar wird.

Eine Besonderheit der `SectionList` ist, dass sich das Verhalten dieser Komponente bei Android und iOS unterscheidet. Dies zeigt sich beim Scrollen der Liste. In Android werden die Abschnittsüberschriften mit der Liste nach oben bzw. unten bewegt, während bei iOS die Überschriften so lange fest am oberen Bildschirmrand stehen bleiben, bis für diesen Abschnitt keine Einträge mehr sichtbar sind. Erst dann bewegt sich auch die Überschrift aus dem sichtbaren Bereich. Dieses Verhalten lässt sich bei Bedarf für eine `SectionList`-Komponente mit dem Prop `sticky SectionHeadersEnabled` anpassen.

Button und die Touchable-Komponenten

Benutzer von Smartphone-Apps nutzen das touchsensitive Display, um mit der App zu interagieren. Dazu werden oft klar erkennbare Bereiche dargestellt, auf die Benutzer tippen oder drücken können, um bestimmte Aktionen auszuführen. Mögliche Bereiche sind z.B. Bilder, Icons oder farblich unterlegter Text (bzw. ein *Button*). Mit der Komponente `Button` bietet React Native in einer App eine einfache Möglichkeit für die Interaktion durch Drücken (*Touch*). Wir haben bereits für die App StepCounter eine `Button`-Komponente verwendet. Mit den Props `title` und `onPress` werden die Beschriftung von `Button` und sein Verhalten beim Drücken definiert:

```
<Button title="Knopf" onPress={() => console.log('Knopf
gedrückt')}/>
```

Solch ein Button wird durch React Native auf Android- und iOS-Geräten unterschiedlich dargestellt, wie Sie in Abbildung 4-6 sehen können.

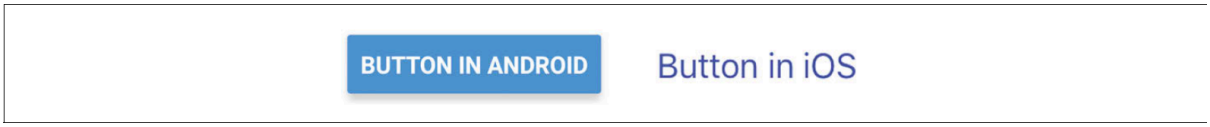

Abbildung 4-6: Darstellung einer Button-Komponente in Android und in iOS

Ein Button erscheint auf Android-Geräten als klar definierte Fläche, während die gleiche Komponente in iOS wie gewöhnlicher Text in normaler Schreibweise aussieht. Außerdem beschriftet Android diese Art von Knöpfen mit Großbuchstaben, und unterhalb der Komponente ist etwas Schatten zu sehen,

so als würde der Knopf »schweben«. Anhand dieser Komponente werden die Unterschiede in den UI-Richtlinien der beiden Plattformen Android und iOS deutlich. Android verwendet das sogenannte *Material Design*,[3] das sich in vielerlei Hinsicht von den Richtlinien unterscheidet, die von Apple für iOS vorgegeben werden (siehe Human Interface Guidelines, HIG).[4]

Viele Aspekte einer mobilen App können wir mit React Native häufig plattformübergreifend mit identischen Komponenten und Code für Android und iOS entwickeln, wie das Beispiel der Komponente `Button` zeigt. Die Komponenten werden durch das jeweilige Betriebssystem entsprechend unterschiedlich dargestellt. Manchmal ist es jedoch wünschenswert, ein bestimmtes Aussehen oder ein Verhalten für jede Plattform im Detail anzupassen. React Native bietet verschiedene Ansätze, um plattformspezifische Unterschiede im Code zu implementieren, und eine dieser Vorgehensweisen werden Sie gleich kennenlernen.

In mobilen Apps können im Prinzip beliebige UI-Elemente auf Touchgesten und insbesondere auf das Antippen reagieren. Im Beispiel von MyJournal könnte die Auswahl eines Tagebucheintrags in der `SectionList` zu einer Detailansicht dieses Eintrags führen. Diese wird später nützlich, wenn ein Tagebucheintrag z.B. längeren Text enthält, der nicht vollständig in der Liste erscheint. Momentan besteht ein Listeneintrag nur aus Text, sodass sich die Frage stellt, wie eine `Text`-Komponente »touchfähig« wird. Dazu gibt es in React Native verschiedene Komponenten: `TouchableWithoutFeedback`, `TouchableNativeFeedback`, `TouchableOpacity` und `TouchableHighlight`. Soll bei einer bestimmten Komponente auf das Antippen reagiert werden, wird diese Komponente in die zu verwendende Touchkomponente eingebettet. Diese Komponenten werden also als sogenannte »Wrapper«-Elemente eingesetzt, wie das folgende Beispiel mit `TouchableOpacity` zeigt:

```
<TouchableOpacity onPress={() => console.log('component
pressed')}>

  <View>

    <Text>Diese beiden Text-Komponenten …</Text>

    <Text>… werden hiermit touchfähig.</Text>
```

```
      </View>

    </TouchableOpacity>
```

Hier werden zwei zusammenhängende `Text`-Komponenten mit einem `View`-Elternelement von einem `TouchableOpacity`-Element umgeben, um auf das Antippen des Texts zu reagieren. Die anderen Touchkomponenten werden analog verwendet. Anstelle der `Text`-Komponente könnten auch beliebige andere Komponenten eingebettet werden. Mit der Funktion in `onPress` wird das gewünschte Verhalten beim Drücken als Callback definiert. Über weitere verfügbare Props, die in der offiziellen Dokumentation dieser Komponente beschrieben werden, können unter anderem visuelle Einstellungen vorgenommen werden. In der Regel dürfen diese Touchkomponenten nur ein Kindelement enthalten, weshalb die beiden `Text`-Komponenten von einem `View`-Element umgeben sind. Es folgt eine Beschreibung der verschiedenen Komponenten für Touchgesten.

`TouchableWithoutFeedback`

: Wie der Name schon andeutet, wird beim Antippen dieser Komponente kein visuelles Feedback dargestellt. Denkbar wäre der Einsatz dieser Komponente im Zusammenhang mit einem Eingabefeld, das per Touch für eine Eingabe aktiviert wird und keinen visuellen Effekt erzeugt. Im Allgemeinen sorgt jedoch ein visueller Effekt bei Touchkomponenten für eine bessere Bedienbarkeit. Daher wird diese Komponente nur in seltenen Fällen verwendet.

`TouchableNativeFeedback`

: Diese Komponente kann nur auf Android-Geräten eingesetzt werden. Dort erscheint beim Antippen solch einer Komponente der sogenannte »Ripple«-Effekt, der seit Einführung des *Material Design* in Android standardmäßig für Touchkomponenten dargestellt wird. Ausgehend von der Stelle, die angetippt wird, dehnt sich für kurze Zeit ein visueller Effekt konzentrisch über die ganze Komponente aus. Dies erinnert an sich kräuselndes Wasser, woran der Name »Ripple«-Effekt angelehnt ist. Für Android ist die Verwendung dieser Komponente in den meisten Fällen empfehlenswert.[5] Bei dieser Komponente muss das Kindelement ein `View`-Element sein,

selbst wenn die für Touchgesten zu erweiternde Komponente nur aus einem einzigen `Text`-Element besteht.

Mit `TouchableNativeFeedback` steht uns also eine Komponente ausschließlich für Android zur Verfügung. Die verbleibenden zwei Komponenten `TouchableOpacity` und `TouchableHighlight` kommen für iOS infrage.

`TouchableOpacity`

Beim Antippen wird die in einem `TouchableOpacity`-Element eingebettete Komponente für kurze Zeit leicht durchsichtig dargestellt. Diese Komponente wird daher in iOS meistens für Text oder Icons verwendet.

`TouchableHighlight`

Auch mit dieser Komponente wird das enthaltene Element beim Antippen kurzzeitig transparent, allerdings wird hierbei eine Hintergrundfarbe eingeblendet, die durch den Prop `underlayColor` definiert wird. In iOS eignet sich `TouchableHighlight` somit für Komponenten mit klar umrissenen Formen oder Farben, z.B. für selbst definierte Buttons, die in iOS nicht nur aus bloßem Text bestehen sollen.

Nun wollen wir die Tagebucheinträge in der Liste mit Touchkomponenten erweitern. Zunächst soll noch kein spezielles Verhalten mit `onPress` definiert werden. Wir möchten lediglich ein visuelles Feedback erhalten, wenn wir einen Listeneintrag auswählen. Für die Android-Version der App wollen wir `TouchableNativeFeedback` einsetzen, und für iOS wählen wir `TouchableOpacity`. Wir benötigen also eine Möglichkeit, um im Code programmatisch zwischen Android und iOS unterscheiden zu können. Dafür stellt React Native eine API namens `Platform` zur Verfügung, die wir nun in *App.js* zusätzlich zu `TouchableNativeFeedback` und `TouchableOpacity` importieren:

```
import {
  KeyboardAvoidingView,
  Platform,
  SectionList,
```

```
    StyleSheet,
    Text,
    TextInput,
    TouchableNativeFeedback,
    TouchableOpacity,
    View
  } from 'react-native';
```

Zur Laufzeit können wir dann mithilfe von `Platform.OS` herausfinden, ob unsere App auf einem Android-Gerät (`'android'`) oder in iOS ausgeführt wird (`'ios'`). In der Methode `render` der Datei *App.js* weisen wir unter Verwendung dieser API einer Konstanten `TouchableItem` die zur Plattform passende Touchkomponente zu:

```
  render() {
    const TouchableItem =
      Platform.OS === 'ios' ? TouchableOpacity :
      TouchableNativeFeedback;
    // … der Rest von render bleibt unverändert …
```

Danach passen wir die Callback-Funktion in `renderItem` der `SectionList`-Deklaration an, um einen Listeneintrag in `TouchableItem` einzubetten:

```
  // diese Änderung betrifft den JSX-Code der SectionList
```

```
<SectionList
  style={styles.list}
  sections={this.state.items}
  renderItem={({ item }) => (
    <TouchableItem>
      <View>
        <Text>{item.text}</Text>
      </View>
    </TouchableItem>
  )}
  // … der Rest der SectionList bleibt unverändert …
```

Plattformspezifische Komponenten in Dateien

Falls eine komplexe Komponente signifikante Unterschiede zwischen Android und iOS aufweist, dann kann der Code durch viele Fallunterscheidungen mit `Platform.OS` unübersichtlich werden. In solchen Situationen kann es sinnvoll sein, den Code in zwei verschiedene Dateien mit plattformspezifischen Zusätzen im Dateinamen aufzuteilen. Heißt die Komponente z.B. `SpecialComponent`, dann sollte die Android-Version in einer Datei namens `SpecialComponent.android.js` definiert werden und für die iOS-Variante entsprechend in der Datei `SpecialComponent.ios.js`. Um die plattformspezifische Komponente zu verwenden, genügt eine gewöhnliche `import`-Anweisung ohne Zusatz im Dateinamen und React Native wird zur Laufzeit die zur jeweiligen Plattform passende Version der Komponente laden:

```
import SpecialComponent from './SpecialComponent';
```

Wenn wir nun einen Eintrag in der Liste antippen, wird visuelles Feedback angezeigt, das zur jeweiligen Plattform passt. In der Android-Version ist der »Ripple«-Effekt zu sehen, während auf dem iPhone der Text des Eintrags für kurze Zeit durchsichtig dargestellt wird. Diese Änderung diente hier hauptsächlich der Veranschaulichung dieser häufig verwendeten Touchable-Komponenten und wird erst in einem späteren Kapitel mit einer konkreten Auswirkung in der App ausgestattet.

Sie haben einige Komponenten kennengelernt und viele Anpassungen am Code durchgeführt. In Beispiel 4-12 habe ich den vollständige Code in *App.js* aufgelistet, damit Sie Ihre Version mit diesem aktuellen Zwischenstand vergleichen können.

Beispiel 4-12: Den vollständigen Code in App.js finden Sie auch online auf der Webseite zum Buch: www.behrends.io/react-native-buch.

```
import React from 'react';

import {

  KeyboardAvoidingView,

  Platform,
```

```
  SectionList,
  StyleSheet,
  Text,
  TextInput,
  TouchableNativeFeedback,
  TouchableOpacity,
  View
} from 'react-native';

const journalItems = [
  {
    data: [
      {
        text: 'Umgang mit SectionList in React Native
        gelernt',
        date: 1
      }
    ],
```

```
    title: '29.7.2017'
  },
  (
    data: [
      { text: 'Einkauf im Supermarkt', date: 2 },
      { text: 'Wochenendausflug geplant', date: 3 }
    ],
    title: '28.7.2017'
  }
];
export default class App extends React.Component {
  state = { items: journalItems };
  _addItem(text) {
    let { items } = this.state;
    let [head, …tail] = items;
    const now = new Date();
    const day = now.getDate();
```

```
    const month = now.getMonth() + 1;

    const year = now.getFullYear();

    const today = `${day}.${month}.${year}`;

    if (head === undefined || head.title !== today) {

      // ggf. neuer Abschnitt für heutiges Datum

      head = { data: [], title: today };

      tail = items;

    }

    const newItem = { text: text, date: now.getTime() };

    head.data = [newItem, …head.data];

    items = [head, …tail];

    this.setState({ items });

    this.textInput.clear();

  }

  render() {

    const TouchableItem =

      Platform.OS === 'ios' ? TouchableOpacity :
      TouchableNativeFeedback;
```

```
let content = <Text>Keine Einträge im Tagebuch</Text>;

if (this.state.items.length > 0) {

  content = (

    <SectionList

      style={styles.list}

      sections={this.state.items}

      renderItem={({ item }) => (

        <TouchableItem>

          <View>

            <Text>{item.text}</Text>

          </View>

        </TouchableItem>

      )}

      renderSectionHeader={({ section }) => (

        <Text style={styles.listHeader}>{section.title}
        </Text>

      )}
```

```
      keyExtractor={item => item.date}
    />
  );
}
return (
  <View style={styles.container}>
    {content}
    <KeyboardAvoidingView behavior="padding">
      <TextInput
        style={styles.input}
        ref={input => (this.textInput = input)}
        placeholder="Tagebucheintrag erstellen"
        returnKeyType="done"
        onSubmitEditing={event =>
          this._addItem(event.nativeEvent.text)}
      />
    </KeyboardAvoidingView>
```

```
      </View>
    );
  }
}
const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center'
  },
  list: {
    marginTop: 24
  },
  input: {
    height: 40
  },
  listHeader: {
    backgroundColor: 'darkgray'
```

```
  }

});
```

Übung

Verwenden Sie in der Liste `TouchableHighlight` mit dem Prop `underlayColor`, um beim Antippen eines Eintrags kurzzeitig einen farbigen Hintergrund erscheinen zu lassen.

Inzwischen hat die Klasse `App`, die die Hauptkomponente in MyJournal ist, durch die Hilfsmethode `_addItem` und die Verwendung verschiedener UI-Komponenten deutlich an Umfang zugenommen und besteht aus mehr als 100 Zeilen Code. Da wir alle Änderungen bisher nur in der Klasse `App` vorgenommen haben, ist diese inzwischen relativ unübersichtlich. Verschiedene Teile des Codes könnten durch Auslagerung in eigenständige Komponenten vereinfacht werden. Deswegen werden wir nun eine Umstrukturierung des Codes durchführen. Die Codestruktur erleichert uns zusätzlich die Änderungen aus den folgenden Kapiteln.

Code durch Komponenten strukturieren

Als Abschluss dieses Kapitels wollen wir die Lesbarkeit des Codes in MyJournal verbessern. Bisher gibt es nur eine Komponente namens `App`, die die Hauptkomponente in MyJournal ist. Als Hauptkomponente definiert `App` die äußerste Komponente der App, die alle anderen Komponenten enthält. In der Methode `render` wird also ein hierarchischer Komponentenbaum erzeugt und als Rückgabewert geliefert. Diese Methode enthält neben dem `return`-Ausdruck eine Fallunterscheidung, um zu bestimmen, ob die Liste der Tagebucheinträge oder nur ein Text angezeigt werden soll. Somit hat diese Methode mehrere Zuständigkeiten (*Concerns*) und ist insgesamt nicht einfach nachzuvollziehen.

Weiterhin ist die App aus Sicht des UI aus drei Komponenten zusammengesetzt: Die Hauptkomponente als äußerer Rahmen ist `App`, die eine Darstellung der Tagebucheinträge und ein Eingabefeld enthält (siehe Abbildung 4-7).

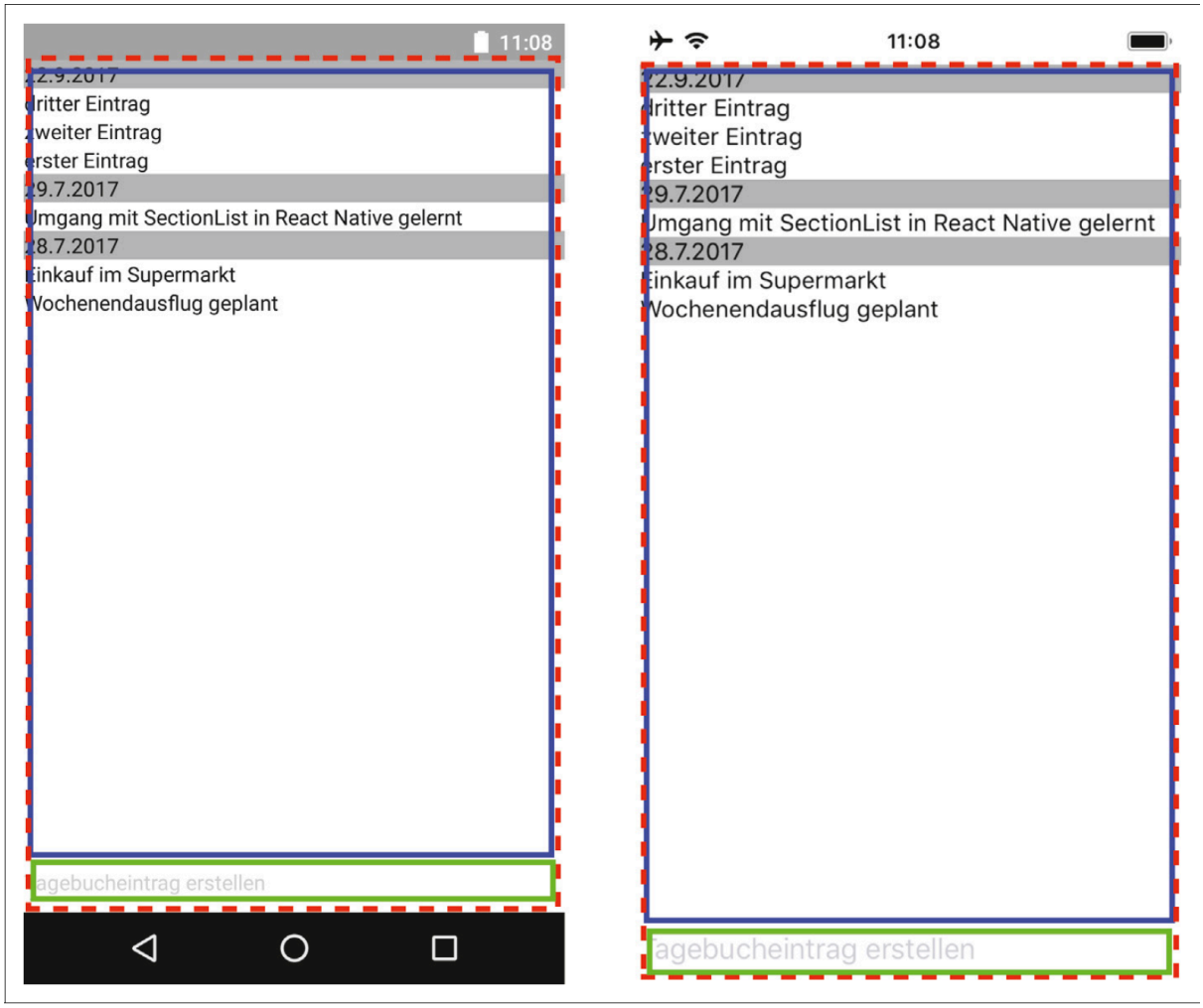

Abbildung 4-7: Die App ist aus drei Komponenten zusammengesetzt. Die Hauptkomponente (gestrichelte Linie) enthält zwei weitere Komponenten.

Diese Sichtweise mit Betonung der Komponenten liegt dem Programmiermodell von React (Native) zugrunde und sollte im Code befolgt werden. Insbesondere sollten so wenige Komponenten wie möglich mit `state` arbeiten. In MyJournal genügt es, wenn die äußere Komponente `App` den Zustand verwaltet und Daten als `props` an die anderen Komponenten weiterreicht.

Aus diesen Gründen wollen wir nun eine Komponente für die Darstellung der Tagebucheinträge extrahieren. Diese werden wir `JournalItems` nennen. Sobald diese Komponente erstellt ist, können wir sie in `App` verwenden, indem wir sie zunächst importieren, im JSX-Code der Methode `render` als Komponente im UI einbinden und ihr mit dem Attribut `items` die Liste der Tagebucheinträge als Prop übergeben:

```
// Komponente importieren:
```

```
import JournalItems from './JournalItems';

// angedeutetes JSX-Fragment mit Prop für items in der
Methode render:

<JournalItems items={this.state.items} />
```

Innerhalb der Komponente `JournalItems` steht die Liste der Tagebucheinträge dann unter `this.props.items` zur Verfügung, wie wir gleich in der Implementierung der Komponente `JournalItems` sehen werden.

Öffnen Sie eine neue Datei im Editor, die Sie mit dem Namen *JournalItems.js* speichern. In dieser Datei werden wir eine eigene Komponente namens `JournalItems` als Klasse implementieren. Die Klasse wird lediglich eine Methode `render` haben, die im Fall eines leeren Arrays in `this.props.items` entweder einen Text oder die Liste der Einträge zurückliefert. Fügen Sie den Code aus Beispiel 4-13 in diese Datei ein (wie gewohnt, ist die Datei *JournalItems.js* auch auf der Webseite zum Buch zu finden).

Beispiel 4-13: Die Liste als eigene Komponente in JournalItems.js

```
import React, { Component } from 'react';

import {

  Platform,

  SectionList,

  StyleSheet,

  Text,

  TouchableNativeFeedback,
```

```
  TouchableOpacity,
  View
} from 'react-native';
const TouchableItem = Platform.OS === 'ios'
  ? TouchableOpacity
  : TouchableNativeFeedback;
export default class JournalItems extends Component {
  render() {
    if (this.props.items.length === 0)
      return <Text>Keine Einträge im Tagebuch</Text>;
    return (
      <SectionList
        style={styles.list}
        sections={this.props.items}
        renderItem={({ item }) =>
          <TouchableItem>
            <View>
```

```
              <Text>{item.text}</Text>

            </View>

          </TouchableItem>}

        renderSectionHeader={({ section }) =>

          <Text style={styles.listHeader}>{section.title}
          </Text>}

        keyExtractor={item => item.date}

      />

    );

  }

}

const styles = StyleSheet.create({

  list: {

    marginTop: 24

  },

  listHeader: {

    backgroundColor: 'darkgray'

  }
```

```
});
```

Im Prinzip haben wir die Teile des Codes, die für die Darstellung der Tagebucheinträge benötigt werden, aus *App.js* übernommen. Zwei Änderungen sind allerdings zu beachten. Erstens wird die Konstante `TouchableItem` außerhalb der Klasse `JournalItems` initialisiert, wodurch die Methode `render` noch etwas kompakter wird. Zweitens wird `JournalItems` ohne Angabe des Moduls `React` als Subklasse von `Component` deklariert. Dazu mussten wir vorher `Component` explizit importieren:

```
// wenn Component aus dem React-Modul importiert wird …

import { React }, Component from 'react';

// … dann kann die Klasse direkt von Component abgeleitet
werden

export default class JournalItems extends Component {
```

Die Klassendeklaration wird hierdurch etwas vereinfacht. Diese Änderung werden wir neben einigen anderen Anpassungen auch für die Komponente `App` vornehmen.

Öffnen Sie jetzt die Datei *App.js* im Editor, die Sie wie in Beispiel 4-14 aufgelistet anpassen. Dort können wir einige Komponenten aus der `import`-Anweisung entfernen. Zusätzlich wird die eben erstellte Komponente `JournalItems` importiert, und wir initialisieren die Konstante `journalItems` mit einem leeren Array, da wir die Beispieldaten nur für das Testen der `SectionList` benötigt haben.

Die Methode `render` ist deutlich übersichtlicher geworden. Wir geben lediglich den passenden JSX-Code zurück und deklarieren dort, wie die neue Komponente `JournalItems` anstelle von `{content}` eingebettet wird. Dieser Komponente übergeben wir durch den Prop `items={this.state.items}` das Array bestehend aus den Tagebucheinträgen. Das `styles`-Objekt am Ende der Datei besteht nun aus weniger Deklarationen.

Beispiel 4-14: App.js verwendet jetzt die neue Komponente JournalItems und erbt direkt von Component.

```
import React, { Component } from 'react';

import {

  KeyboardAvoidingView,

  StyleSheet,

  Text,

  TextInput,

  View

} from 'react-native';

import JournalItems from './JournalItems';

const journalItems = [];

export default class App extends Component {

  state = { items: journalItems };

  _addItem(text) {

    let { items } = this.state;

    let [head, …tail] = items;

    const now = new Date();
```

```
    const day = now.getDate();
    const month = now.getMonth() + 1;
    const year = now.getFullYear();
    const today = `${day}.${month}.${year}`;
    if (head === undefined || head.title !== today) {
      // ggf. neuer Abschnitt für heutiges Datum
      head = { data: [], title: today };
      tail = items;
    }
    const newItem = { text: text, date: now.getTime() };
    head.data = [newItem, …head.data];
    items = [head, …tail];
    this.setState({ items });
    this.textInput.clear();
  }
  render() {
    return (
```

```
      <View style={styles.container}>
        <JournalItems items={this.state.items} />
        <KeyboardAvoidingView behavior="padding">
          <TextInput
            style={styles.input}
            ref={input => (this.textInput = input)}
            placeholder="Tagebucheintrag erstellen"
            returnKeyType="done"
            onSubmitEditing={event =>
              this._addItem(event.nativeEvent.text)}
          />
        </KeyboardAvoidingView>
      </View>
    );
  }
}
const styles = StyleSheet.create({
```

```
  container: {
    flex: 1,
    justifyContent: 'center'
  },
  input: {
    height: 40
  }
});
```

Die Eingabe mit `KeyboardAvoidingView` und `TextInput` könnte auch als eigenständige Komponente extrahiert werden, was ich Ihnen als Übung zur Erstellung eigener Komponenten nahelegen möchte.

Zusammenfassung

In diesem Kapitel haben wir durch die Entwicklung eines Prototyps für eine Tagebuch-App die Verwendung verschiedener UI-Komponenten von React Native kennengelernt. Die Darstellung und die Eingabe von Texten mit `Text` und `TextInput` wurden ausgiebig behandelt, und wir haben gesehen, wie Komponenten mit `View` zusammengefasst werden. Für Listen haben wir aufeinander aufbauend `FlatList` und `SectionList` eingesetzt. Im Zusammenhang mit den verschiedenen `Touchable`-Komponenten haben wir eine plattformspezifische Fallunterscheidung mit der `Platform`-API implementiert. Schließlich haben wir mit `JournalItems` eine eigene Komponente für die Darstellung der Tagebucheinträge erstellt. Somit sind wir grundlegend in der Lage, UI-Komponenten von React Native in eigenen Komponenten zu verwenden, was bereits ein wesentlicher Aspekt der plattformübergreifenden App-Entwicklung mit React Native ist.

Es gibt zwei wichtige UI-Komponenten in React Native, die wir in diesem Kapitel noch nicht verwendet haben: `Image` und `ScrollView`. Die Anzeige von Bildern bzw. Fotos mit `Image` werden wir in Kapitel 6 im Zusammenhang mit dem Einsatz der Kamera des Smartphones betrachten. Mit `ScrollView` steht uns eine häufig eingesetzte Komponente zur Verfügung, mit der Inhalte, die nicht vollständig auf dem Bildschirm dargestellt werden können, in eine scrollbare Ansicht eingebettet werden können (z.B. lange Texte). `ScrollView` wird uns erst in Kapitel 8 begegnen.

React Native umfasst weitere UI-Komponenten, die wir nicht in diesem Buch behandeln können. Alle verfügbaren Komponenten sind in der offiziellen Dokumentation aufgelistet (*facebook.github.io/react-native/docs*). Dennoch möchte ich einige Komponenten erwähnen, die für die Erstellung von UIs in mobilen Apps nützlich sein können:

`Modal`

: Ein `Modal` ermöglicht es, Inhalte über der aktuellen Ansicht einzublenden.

`Picker`

: `Picker` stellt passend zur Plattform eine Auswahlliste dar.

`Slider`

: Dies ist eine Art »Schieberegler« zur Wahl eines Werts aus einem Intervall.

`Switch`

: Hierdurch wird der für mobile Apps typische Schalter mit zwei Zuständen angezeigt (an/aus).

Manchmal werden neue UI-Komponenten in React Native eingeführt, die möglicherweise bestehende Komponenten ersetzen. Das ist z.B. mit `FlatList` und `SectionList` geschehen, wodurch die bis dahin häufig verwendete Komponente `ListView` obsolet wurde. Solche Änderungen und andere Neuigkeiten werden in der Regel im offiziellen *React Native Blog* angekündigt (*facebook.github.io/react-native/blog*).

Übungen

- Erstellen Sie für die Eingabe (`KeyboardAvoidingView` und `TextInput`) eine eigenständige Komponente, die Sie in `App` verwenden.

- Validierung der Eingabe: Verhindern Sie, dass leere Einträge oder solche, die nur aus Leerzeichen bestehen, hinzugefügt werden können. Ein Dialog mit einer Warnung könnte auf fehlgeschlagene Validierungen hinweisen.
- Leerzeichen am Anfang und am Ende einer Eingabe könnten automatisch entfernt werden. In `String` steht dazu eine Methode `trim` bereit.
- Wird die App MyJournal regelmäßig über einen längeren Zeitraum genutzt, wird die Liste übersichtlicher, wenn die Abschnitte alle Einträge eines Monats zusammenfassen (anstatt eines Tages). Ändern Sie die App entsprechend.

Auf der Webseite zum Buch finden Sie Lösungsansätze zu einzelnen Übungen (*www.behrends.io/react-native-buch*).

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

P

R

S

T

V

W