

עברית - REACT Docs

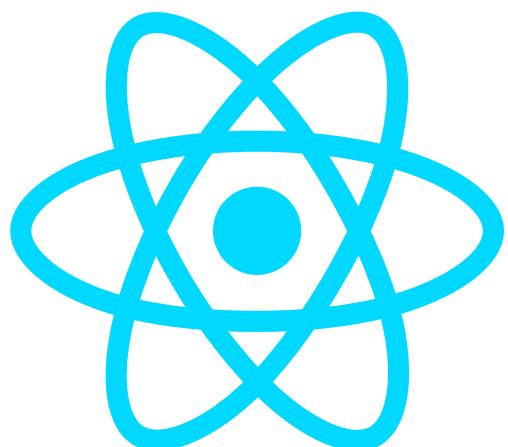


Table of contents

• Accessibility	5
• Add react /to /a /website	16
• Addons animation	21
• Addons create /fragment	28
• Addons perf	30
• Addons pure /render /mixin	34
• Addons shallow /compare	35
• Addons shallow /renderer	36
• Addons test /utils	38
• Addons two /way /binding /helpers	44
• Addons update	47
• Addons	50
• Cdn links	52
• Code splitting	54
• Codebase overview	61
• Components and /props	65
• Composition vs /inheritance	70
• Conditional rendering	73
• Context	78
• Create a /new /react /app	84
• Cross origin /errors	87
• Design principles	89
• Error boundaries	96
• Error decoder	100
• Faq ajax	101
• Faq build	104
• Faq functions	105
• Faq internals	112
• Faq state	113
• Faq structure	116
• Faq styling	118
• Faq versioning	119
• Forms	122
• Forwarding refs	128

• Fragments	130
• Getting started	133
• Handling events	137
• Hello world	140
• Higher order /components	142
• Hooks custom	150
• Hooks effect	155
• Hooks faq	165
• Hooks intro	186
• Hooks overview	191
• Hooks reference	197
• Hooks rules	212
• Hooks state	215
• How to /contribute	221
• Implementation notes	226
• Integrating with /other /libraries	244
• Introducing jsx	253
• Jsx in /depth	257
• Legacy context	265
• Legacy event /pooling	270
• Lifting state /up	271
• Lists and /keys	278
• Optimizing performance	283
• Portals	292
• React without /es6	295
• React without /jsx	299
• Reconciliation	300
• Reference dom /elements	305
• Reference events	310
• Reference glossary	318
• Reference javascript /environment /requirements	323
• Reference profiler	325
• Reference pure /render /mixin	328
• Reference react /component	330
• Reference react /dom /client	345
• Reference react /dom /server	347

• Reference react /dom	356
• Reference react	366
• Reference test /renderer	375
• Refs and /the /dom	381
• Release channels	387
• Render props	390
• Rendering elements	396
• State and /lifecycle	398
• Static type /checking	406
• Strict mode	413
• Testing environments	419
• Testing recipes	421
• Testing	432
• Thinking in /react	434
• Typechecking with /proptypes	439
• Uncontrolled components	444
• Web components	446

נגישות

למה נגישות ?

היא יצירה ועיצוב אתרים שמתאימים לשימוש ע"י כולם. תמייהה ב נגישות נדרשת ע"י ([WCAG 1.1](#) ידועה גם כ) נגישות ברשות טכנולוגית מסוימת כדי לפרש דפי אינטרנט.

סטנדריות HTML מספקת תמייהה מלאה לבניית אתרים נגישים, בדרך כלל ע"י שימוש בטכניות React.

סטנדרטים וקווים מנהיים

WCAG

ניתן למצוא קוים מנהיים ליצירת אתרים נגישים ב [Web Content Accessibility Guidelines](#).

מספקת סקירה כללית WCAG הרשימה הבאה מ:

- [WCAG 1.1 רשיימת Wuchag](#)
- [WCAG 1.1 רשיימת WebAIM](#)
- [WCAG 1.1 הרשימה מפרויקט ה](#)

WAI-ARIA

מכיל טכניות לבניית וידג'טים נגישים [Web Accessibility Initiative - Accessible Rich Internet Applications](#) המסמך מ- JavaScript.

שנקראות ב React-ב DOM-בשונה מרוב תכונות ה `aria-*`. HTML תומך לחלוון בכל תכונות ה JSX שימוש לב ש camelCase, (וכו' case kebab-case, hyphen-cased, lisp-case) בדיקות נקראות אלה נקראות ב JavaScript: רגיל HTML-כמו ב:

```
<input
  type="text"
  aria-label={labelText}
  aria-required="true"
  onChange={onchangeHandler}
  value={inputValue}
  name="name"
/>
```

HTML סמנטי

סמנטי (או דקדוקי) הוא הבסיס לנגישות באפליקציות ואתרי אינטרנט. השימוש באלמנטים שונים על מנת לחזק את HTML המשמעות של המידע שמוצג אתרים, בדרך כלל יוביל לנגישות בפני עצמו.

- [MDN HTML elements reference](#)

לעבוד כמו שצרכיך, בעיקר React כדי לגרום לJS-ל `<div>` לפעמים הסמנטיקה נשברת כשאנו חנו מוסיפים אלמנטים כמו ``, ``, `<dl>`, `<table>` (וכו' במקרים האלה ניתן להשתמש בפרגמנטים בReact) בזמן שימוש ברשימות ועבלאות כדי לאחד מספר אלמנטים, `div` במקום.

לדוגמא,

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

function ListItem({ item }) {
  return (
    <Fragment>
      <dt>{item.term}</dt>
      <dd>{item.description}</dd>
    </Fragment>
  );
}

function Glossary(props) {
  return (
    <dl>
      {props.items.map(item => (
        <ListItem item={item} key={item.id} />
      ))}
    </dl>
  );
}
```

:ניתן כibold למפות רשימת פריטים למערך של פרגמנטים באוטה צורה שמאפיין כל אלמנט אחר

```
function Glossary(props) {
  return (
    <dl>
      {props.items.map(item => (
        // בזמן מיפוי רשימות צריך להוציא את ה `prop` גם לפרגמנטים
        <Fragment key={item.id}>
          <dt>{item.term}</dt>
          <dd>{item.description}</dd>
        </Fragment>
      ))}
    </dl>
  );
}
```

:ניתן להשתמש בסינטקס מקוצר, בהנחה ושאר الكلים תומכים בו props כשיין צריך בהוספה

```
function ListItem({ item }) {
  return (
    <>
      <dt>{item.term}</dt>
      <dd>{item.description}</dd>
    </>
  );
}
```

לעוד מידע, ראו את עמוד [תיעוד הפרויקטים](#).

טפסים נגישים

תיווג

צריכים לקבל תיווג נגיש. علينا לספק תgioת שמתארות, (`<input>` ו `<textarea>` כמו) ו-HTML-ב `form` כל אלמנט או. אותם בשביל קוראי המסק.

:המשאים הנ"ל מראים לנו איך לספק תgioת מתאימות

- [מראה לנו איך לתיווג אלמנטים W3C-ה](#)
- [מראה לנו איך לתיווג אלמנטים WebAIM](#)
- [מסביר על שמות נגישים Paciello Group](#)

למשל `for` שימושו לב שהתקונה, -שירות ב HTML-למרות שבדרך כלל אנחנו יכולים להשתמש בטונדריטים הנוהגים ב `htmlFor` כ `htmlFor` :

```
<label htmlFor="namedInput">Name:</label>
<input id="namedInput" type="text" name="name"/>
```

ידיעו המשמש במקרה של שגיאות

:שגיאות צריכות להיות מובנות לכל משתמש. הLINK הבא מראה לנו איך להעביר את תוכן השגיאה גם לקוראי המסק

- [מדגים הודיעות משתמש W3C-ה](#)
- [מסביר על ולדציה של דפים WebAIM](#)

ביקורת פוקוס

:חשוב לבנות אפליקציות שנייניות לשימוש בעזרת המקלדת בלבד (לא עזרת העכבר)

- [מסביר על נגישות מקלדת WebAIM](#)

פוקוס מקלדת ומסגרת פוקוס

שנבחר ומוכן לקבל נתונים מהמקלדת. ניתן לראות את הפוקוס במסגרת או הדגש, DOM-פוקוס המקלדת מתאר את האלמנט ב: כמו בתמונה הבאה:



(0 `outline: 0` לדוגמא בעזרת הקונפיגורציה) מומלץ להסיר את ההדגש הנ"ל אך ורק במקרה להחליף אותו בצורת הדגש אחרת

דילוג לתוכן מבוקש

על מנת שהשימוש באתר יהיה מהיר ואופטימלי עם המקלדת, ניתן לספק דרך לדג בעזרת אזורי ניוט לתוכן מבוקש

הם LINKS נסתרים שמתגלים בזמן אינטראקציה עם האתר באמצעות המקלדת בלבד. הם (SkipLinks) LINKS לדילוג ניוט: וסגולות עיצוב (anchors) פשוטים לפיתוח עצמאות קישורים עוגנים

- [لينكيم לדילוג ניווט - WebAIM](#)

על מנת לצייןאזורים בדף בשביל להרשות למשתמש , `<main>` ו `<aside>` בנוסח, אפשר להשתמש אלמנטים לציין דרך כמו לנווט אליהם بكلות ובמהירות בעזרת טכנולוגיה מסוימת.

קרא עוד על אלמנטים לציין דרך לשיפור נגישות בlienck הבא:

- [צינוי דרך נגישים](#)

שליטה מבוקרת בפוקוס

בזמן ריצה באופן מתמשך, מה שגורם למקלדת לאבד פוקוס או לפוקוס לעבר DOM-משנות את ה-React אפליקציות למקום בלתי צפוי. על מנת למנוע זאת, ביכולתנו להחזיר את פוקוס המקלדת למקום הנכון באופן תוכני. לדוגמה, ע"י החזרת הפוקוס לכפתור שפתח טופס מודאלי, אחריו סגרתו.

מתאר איך לבנות [ניווט מקלדת בווידג'טים ב-JavaScript](#) Web Docs MDN-התיעד ב-

[DOM](#)-אנחנו יכולים להשתמש בקישור לאלמנטים ב-React, כדי לתת פוקוס ב-

של מחלקת קומפוננטה JSX-נתחיל ביצירת קישור לאלמנט ב-

```
class CustomTextInput extends React.Component {
  constructor(props) {
    super(props);
    // DOM-textInput
    this.textInput = React.createRef();
  }
  render() {
    // `ref` על מנת לאחסן את החפניה
    // (this.textInput) בשדה מופע
    return (
      <input
        type="text"
        ref={this.textInput}
      />
    );
  }
}
```

לאחר מכן נוכל לשנות את הפוקוס למקום אחר בקומפוננטה לפי הצורך:

```
focus() {
  // DOM-textInput
  // DOM-textInput.current
  this.textInput.current.focus();
}
```

אלמנט האב [בעזרת DOM](#)-לפעמים אלמנט אב צריך לשנות פוקוס לקומפוננטת יلد. ניתן לעשות זאת ע"י [חשיפות הפניתה](#) של הילד DOM-מיוחד בקומפוננטת הילד שמעבירה לאלמנט האב את צומת ה prop

```
function CustomTextInput(props) {
  return (
    <div>
      <input ref={props.inputRef} />
    </div>
  );
}
```

```

}

class Parent extends React.Component {
  constructor(props) {
    super(props);
    this.inputElement = React.createRef();
  }
  render() {
    return (
      <CustomTextInput inputRef={this.inputElement} />
    );
  }
}

// עכשוויתו תוכלו להעביר פוקוס לפוי הלקוח
this.inputElement.current.focus();

```

להרחבת קומפוננטות, מומלץ [להעביר את הפונקציה](#) **HOC** בזמן השימוש בforwardRef (צד שלישי) לא ממשת את העברת הפונקציה, השיטה HOC במקורה וקומפוננטות React. לעיל יכולה בכל זאת לעזור.

זאת דוגמא יחסית נדירה של חלון מודאלי נגיש למחררי. לא רק [react-aria-modal](#) דוגמא טובה לשיטה בפוקוס אפשר למצוא בשם פוקוס התחלתי בכפתור הביטול (כדי למנוע משתמש המקלדת להתחל את פעולה אישור בטעות) ושומר את החזרת הפוקוס למקלדת בתוך החלון, הוא גם מחזיר את הפוקוס לאלמנט שפתח את החלון מלכתחילה לאחר סגירתו.

הערה:

למרות שזאת תוכנות נגישות חשובה מאוד, חשוב להשתמש בשיפורו בזמן השימוש בטכניתה. המטרה צריכה להיות החזרת הפוקוס למקלדת, ולא לנסות לחזות את דרישות המשתמש ואופן השימוש שלו/ה באתר.

AIRUI עכבר וסמן

יש לדאוג שכל פונקציונליות שזמינים דרך שימוש בעכבר, נגיש באותו מידה בשימוש במקלדת בלבד. תלות בסמן (דרך עכבר או משטח מגע) מובילה להרבה מקרים לא נגישים למשתמשי מקלדת, שכוטצאה מכך לא יוכל להשתמש באפליקציה.

כדוגמה, נציג מקרה שכיח ביותר של נגישות שבורה כתוצאה מאירועי לחיצה - כשהמשתמש יכול לסגור חלון צ' בפעולות אחדות לחיצה מהוות לחalon.

Select an option

Load the option

Remove the option

שסגור את החלון הנוכחי `window.click` בעצם החלון או מושמת ע"י קישור אירוע הלחיצה:

```
class OuterClickExample extends React.Component {
  constructor(props) {
    super(props);

    this.state = { isOpen: false };
    this.toggleContainer = React.createRef();

    this.onClickHandler = this.onClickHandler.bind(this);
    this.onClickOutsideHandler = this.onClickOutsideHandler.bind(this);
  }

  componentDidMount() {
    window.addEventListener('click', this.onClickOutsideHandler);
  }

  componentWillUnmount() {
    window.removeEventListener('click', this.onClickOutsideHandler);
  }

  onClickHandler() {
    this.setState(currentState => ({
      isOpen: !currentState.isOpen
    }));
  }

  onClickOutsideHandler(event) {
    if (this.state.isOpen && !this.toggleContainer.current.contains(event.target))
    {
      this.setState({ isOpen: false });
    }
  }

  render() {
    return (
      <div ref={this.toggleContainer}>
        <button onClick={this.onClickHandler}>Select an option</button>
        {this.state.isOpen && (
          <ul>
            <li>Option 1</li>
            <li>Option 2</li>
            <li>Option 3</li>
          </ul>
        )}
      </div>
    );
  }
}
```

```

    );
}
}
}

```

למרות שהתנהגות זו לא מהווה בעיה למשתמשים שיכולים לעזור בסמן, היא מובילה להתקנות שבורה למשתמשים שתלויים במקלדת כדי לנוט בין אלמנטים, כי עצם החולן לא מקבל את אירוע הלחיצה. ובסיומו של דבר גם מונעת משתמשים גישה לפונקציונליות באתר.

Select an option

Load the option

Remove the option

אפשר במקומות זאת להגעה להתנהגות זהה בעזרת מטפלי אירועים כמו `onBlur` ו- `onFocus`:

```

class BlurExample extends React.Component {
  constructor(props) {
    super(props);

    this.state = { isOpen: false };
    this.timeOutId = null;

    this.onClickHandler = this.onClickHandler.bind(this);
    this.onBlurHandler = this.onBlurHandler.bind(this);
    this.onFocusHandler = this.onFocusHandler.bind(this);
  }

  onClickHandler() {
    this.setState(currentState => ({
      isOpen: !currentState.isOpen
    }));
  }

  // setTimeout את החולן הץ בטיק הבא בעדרת
  // זה חשוב כי אנחנו צריכים קודם כל לבדוק אם יlid אחר של האלמנט
  // קיבול פוקוס, כיון שאירוע של blur קורה לפניו אירוע הפוקוס
  onblurHandler() {
    this.timeOutId = setTimeout(() => {
      this.setState({
        isOpen: false
      });
    });
  }

  // מסירים את ה Timeout כדי לא לסגור //
  // את החולן הץ כשילך אחר מתקבל פוקוס //
  onFocusHandler() {
    clearTimeout(this.timeOutId);
  }
}

```

```

render() {
  // focus blur לאב להנילאת אירובי React
  return (
    <div onBlur={this.onBlurHandler}
      onFocus={this.onFocusHandler}>
      <button onClick={this.onClickHandler}
        aria-haspopup="true"
        aria-expanded={this.state.isOpen}>
        Select an option
      </button>
      {this.state.isOpen && (
        <ul>
          <li>Option 1</li>
          <li>Option 2</li>
          <li>Option 3</li>
        </ul>
      )}
    </div>
  );
}

```

`aria-`* props הנקוד הזה חושף פונקציונליות למשתמשי המקלדת בלי להפקיר את הסמן. בנוסף, שימושו לב לתוספתה השותמכים במשתמשים שנעזרים בקוראי מסך. על מנת להשאיר את הדוגמא פשוטה, לא כללונו את אירובי המקלדת שמאפשרים עם החלון הציג אינטראקטיבית `arrow key`.

Select an option

Load the option

Remove the option

זאת דוגמא שבה מספר מקרים שתלויים רק בסמן מונעים משתמשים שתלויים במקלדת נגישות לפונקציונליות. תמיד בדקנו את האתר עם המקלדת כדי למצוא את האזוריں הביעתיים ולתקן אותם בעזרת מאפיין אירוביים שאמינים במקלדת.

ויזיג'טים מסוובכים יותר

חוויות משתמש מסוובכת יותר לא צריכה להיות פחות נגישה. הדרך פשוטה ביותר לפתח אתר נגיש היא לפתח קוד קרוב ככל אבל גם ויזיג'טים מסוובכים יותר אפשר לפתח בצורה נגישה, HTML האפשר ל

חוויות מסוובכת יותר לא צריכה להיות פחות נגישה. הדרך פשוטה ביותר לפתח אתר נגיש היא לפתח קוד קרוב ככל אבל גם ויזיג'טים מסוובכים יותר אפשר לפתח בצורה נגישה, HTML האפשר ל

חוויות מסוובכת יותר לא צריכה להיות פחות נגישה. הדרך פשוטה ביותר לפתח אתר נגיש היא לפתח קוד קרוב ככל אבל גם ויזיג'טים מסוובכים יותר אפשר לפתח בצורה נגישה, HTML האפשר ל

חוויות מסוובכת יותר לא צריכה להיות פחות נגישה. הדרך פשוטה ביותר לפתח אתר נגיש היא לפתח קוד קרוב ככל אבל גם ויזיג'טים מסוובכים יותר אפשר לפתח בצורה נגישה, HTML האפשר ל

כל סוג ויזיג'ט ממומש בצורה ובמבנה עיצוב שונה, אבל המשתמש וסוכני המשתמש מצפים ממנו להתנהגות מסוימת.

- [WAI-ARIA Authoring Practices - TABNIOOT UICTOB VVIZIG'TIM](#)

- Heydon Pickering - ARIA דוגמאות
- קומפוננטות אינקלוסיביות

עוד נקודות למחשבה

קביעת שפה

: ציון את השפה שבה כתובים הטקסטים בדף על מנת לאפשר לקרואו המסר לבחור את הגדרות הקול בהתאם:

- [טייעוד שפה](#) - WebAIM

קביעת כוורת הדף

: על מנת לתאר את תוכן ומטרת הדף המוצג באופן מדויק למשתמש `<title>` דאג לציין את כוורת הדף באמצעות האלמנט

- [הבנת דרישות כוורת הדף](#) - WCAG

בעזרת [קומפוננטת כוורת הדף](#) React אנחנו יכולים לציין זאת ב-

ניגוד צבעים

ודאו שלכל הטקסטים באתר יש ניגוד צבעים מספק, על מנת להקל על משתמשים עם יכולות ראייה פחותות לקרוא אותם:

- [הבנת דרישות ניגוד צבעים - WCAG](#)
- [כל מה שתרצו לדעת על ניגוד צבעים וסיבות לחשוב עליהם שוב](#)
- [מה הוא ניגוד צבעים - A11yProject](#)

חישוב הניגוד לכל קומבינציה צבעים באתר יכול לחתך הרבה זמן. דרך יותר פשוטה היא לחשב באופן אוטומטי [פלטת צבעים Colorable](#) מלאה לאתר באמצעות.

: מספקים בדיקות אוטומטיות ומדווחים על שגיאות בניגודי צבעים (WAVE ו eXe) הכלים המצוין להלן

- [בדיקות ניגודי צבעים - WebAIM](#)
- [מודוא ניגודי צבעים - The Paciello Group](#)

כליים לפיתוח ובדיקה

יש מספר כלים שימושיים לעזרה ביצירת אפליקציות נגישות.

המקלדת

הבדיקה הכני פשוטה וגם אחת מהחשיבות, היא שהאתר מתפרק בצורה מוחלטת בשימוש במקלדת בלבד. אפשר לעשות זאת בצורה הזאת:

1. נתק/i את העכבר.
2. כדי לנוט `Tab` או `Shift+Tab` בעזרת שימוש במקש ה-.
3. כדי להפעיל אלמנטים `Enter` בעזרת שימוש במקש ה-.
4. שימוש בחצי המקלדת כדי לתפעל אלמנטים כמו תפריטים ורשימות לפי הצורך.

עזרה בפיתוח

בנוסף לintellisense ARIA roles, states and properties, ניתן להשתמש בכלים IDE'ים (כגון VSCode) כדי לבדוק מספק תכונות נגישות באופן ישיר בקוד:

eslint-plugin-jsx-a11y

רבים מ-IDE'ים מומלץ לintelligeance נגישות בקוד AST ESLint. הוא מוסיף ל-ESLint יכולות יישור בקוד המאפשרות אינטגרציה ישירה עם כלים כמו יישור חלון שבו נכתב הקוד לאחר ששמירה.

כולל את התוסף הזה עם חלק מהחוקים מופעלים בצורה מובנית. כדי להפעיל חוקים אחרים, ניתן לצור [צור אפליקציה](#) בתיקייה הראשית של הפרויקט עם התוכן `.eslintrc`. קובץ

```
{
  "extends": ["react-app", "plugin:jsx-a11y/recommended"],
  "plugins": ["jsx-a11y"]
}
```

בדיקות נגישות בדף

קיימים מספר כלים שבוחנים נגישות על עמודי אינטרנט בדף. כדאי להשתמש בהם בתוספת לכלים אחרים שציינו קודם, כיון שהם בודקים אף ורket את הנגישות הטכנית של הקוד HTML.

aXe-core, aXe ו-react-axe

Deque Systems [aXe-core](#) מבצעChecks selenium. aXe-core

הוא מוסיף לדף שבודק נגישות ומרחיב את, aXe [aXe-core](#).

בזמן פיתוח ובדיקה console כדי ליזור דוחות על ממצאים נגישות יישור ל [@axe-core/react](#) ניתן להשתמש גם ב

WebAIM WAVE

[כל בבדיקה הנגישות ברשת](#) הוא מוסיף לדף נספחים בבדיקה הנגישות.

בודק נגישות ועכבר הנגישות

שמכיל עצמים נגישים כל אלמנט שחוורף לקרוא מסך או טכנולוגיות מסוימות אחרות DOM [עכבר הנגישות](#) הוא חלק מה

בדף מסויימים ניתן לראות מידע נגישות לכל אלמנט בעכבר הנגישות:

- [שימוש בFirefox בבדיקה הנגישות של](#)
- [הפעלת בדיקת הנגישות של Chrome](#)
- [שימוש בבדיקה הנגישות של OS X Safari](#)

קוראי מסך

חשוב למשתמש בקוראי מסך כחלק מבדיקות הנגישות באתר.

שימוש לב שילוב בין סוג הדף וקוראי מסך חשוב, ומומלץ לבדוק את האפליקציה בדף המתאים ביותר לקוראי מסך שבחרתם.

קוראי מסך מוכרים ונפוצים

NVDA ב Firefox

/shareables Windows הוא קורא מסך פתוח ל NVDA או בקישור NVDA.

הקישורים הבאים מסבירים על שימוש נכון בNVDA:

- [WebAIM בNVDA נגישות](#) -NVDA שימוש ב -
- [Deque NVDA קיזורי מקלדת ב](#) -

VoiceOver ב Safari

VoiceOver הוא קורא מסך מובנה במכשירים של Apple.

-הקישורים הבאים מסבירים איך להפעיל ולהשתמש בVoiceOver:

- [WebAIM בNVDA VoiceOver נגישות](#) -NVDA שימוש ב -
- [Deque OS X ב VoiceOver קיזורי מקלדת ל](#) -
- [Deque iOS ב VoiceOver קיזורי מקלדת ל](#) -

JAWS ב Internet Explorer

[Job Access With Speech](#) הוא קורא מסך נפוץ ל JAWS או בקישור Windows.

JAWS הקישורים הבאים מסבירים איך להשתמש ב:

- [WebAIM ב JAWS נגישות האתר עם](#) -
- [Deque JAWS קיזורי מקלדת עם](#) -

קוראי מסך אחרים

ChromeVox ב Google Chrome

ChromeVox שאפשר להוריד כ [תוסף](#) ל Chromebooks הוא קורא מסך מובנה ב Google Chrome.

הקישורים הבאים ChromeVox מסבירים איך להשתמש ב:

- [Google Chromebook Help מסך מובנה](#) -
- [Deque ChromeVox קיזורי מקלדת קלסיים ב](#) -

אתר אינטראקטיבי עם React

לפי צרכיך השתמש בכמה שפות או בכמה שיטות.

שאנו צריכים. יתכן **React** הונדרה מההתחלת לAIMOZ הדרגתית, ואנו יכולים להשתמש בכמה שפות או בכמה שירותים. זה דרך מעולה לעשות זאת React שנרצה להופיע כמו "נצחניים של אינטראקטיביות" לדף קיים. קומפוננטות.

הרוב המוחץ של אתרי האינטראקטיבי הם לא, ולא צריכים להיות, יישומי דף-יחיד. בערך כמה שורות קוד ולא כל-בניה, נסה או לשמר אותה מוגבלת לכמה, חלק קטן של האתר האינטראקטיבי שלו. תוכל להגדיל בהדרגה את הנוכחות של React או יישומונים דינמיים בלבד.

- [הוסף את ריאקט בדקה](#)
- [JSON אופציונלי: נסה את ריאקט עם](#) (no bundler necessary!)

הוסף את ריאקט בדקה

קיים. תוכל לעקוב אחר צעדים אלו עם האתר שלך, או ליצור קובץ HTML לדף React בחלק זה, נראה איך להוסיף קומפוננטת HTML. ריק על מנת לתרגל HTML.

לא יהיו כלים מורכבים או דרישות התקנה – **על מנת להשלים חלק זה, כל מה שתצטרך הוא חיבור אינטראקטיבי ודקאה זמן-**

[\(2KB zipped\) אופציונלי: הורד את הדוגמה המלאה](#)

HTML-לקובץ ה DOM שלב ראשון: הוסף קונטיינר

ריקה על מנת לסמון את המקום בו אתה רוצה להציג `<div>` שטרצה לעורך. הוסף תגית HTML-ראשית, פתח את עמודה של דוגמה מהו עם:

```
<! -- ... HTML
<div id="like_button_container"></div>
-- ... HTML
```

마וחר יותר ולהציג JavaScript-זה אפשר לנו למצוא אותו מקוד ה HTML ייחודי כמאפיין `id` זה `<div>` זה -הבאנו לתוכו קומפוננטת React.

טיפ

יכולים להיות לך מספר בלתי מוגבל של `<body>`-באופן זה **בכל מקום** בתוך תגית ה `<div>` ניתן להציב "קונטיינר DOM" תחליף כל תוכן קיים בתוך קונטיינר React -- בעמוד אחד לפי צרכיך. הם בדרך כלל ריקים DOM קונטיינרים של

שלב שני: הוסף את תגיוט הסקריפט

הסוגרת `</body>` - ממש לפני תגית ה HTML-דף ה `<script>` לאחר מכן, הוסף שלוש תגיוט:

```
<!-- ... אחר HTML -->
<!-- טעינה ריאקט -->
<!-- שים לב: כשמעלים לאווריר, מחליפים את -->
-->
<script src="https://unpkg.com/react@18/umd/react.development.js" crossorigin>
</script>
<script src="https://unpkg.com/react-dom@18/umd/react-dom.development.js" crossorigin></script>
<!-- שלנו -טעינה קומפוננטה -->
<script src="like_button.js"></script>
</body>
```

השלישיתتطען את קוד הקומפוננטה שלך React. שתיה התגיות הראשונות טוענות את

שלב שלישי: יצרת קומפוננטת React

שלך HTML-בצמוד לדף ה `like_button.js` צורקובץ בשם.

פתח את [הקוד ההתחלתי זהה](#) והדבק אותו בקובץ שיצרת.

טיפ

אל תdag אם אתה לא מבין את זה עדין ---Ano נכח את `LikeButton` שנקראת React קוד זה מגדר קומפוננטת מהוחר יותר [במדריך המשעי ובדירק הקונספטים העיקריים](#). בין היתר, בוא רק נdag שזה יוצג על היסודות של המסר!

אחרי [הקוד ההתחלתי](#), הוסף שתי שורות לתחתית הקובץ :

```
// ... the starter code you pasted ...
const domContainer = document.querySelector('#like_button_container');
const root = ReactDOM.createRoot(domContainer);
root.render(e(LikeButton));
```

בשלב הראשון, ואז מציגות את קומפוננטת כפטור ה-HTML-שהוספנו לדף ה `<div>`-שתי שורות אלו מוצאות את תגיית ה `"ליק"` בתוכו.

זהו זה!

הראשונה לאתר אינטרנט שלך React-אין שלב רביעי. הרגע והוסף את קומפוננטת ה

ראה את הפרקים הבאים על מנת למדוד עוד טיפים על שילוב React.

[ראה את קוד הדוגמה המלא](#)

[הורד את הדוגמה המלאה \(2KB zipped\)](#)

טיפ: שימוש חוזר בקומפוננטה

הנה דוגמה שמציגה את כפתרו ה-"לייק" שלוש HTML-בכמה מקומות בדף React לעתים קרובות, נרצה להציג קומפוננטות פשוטים ומעבירה אליהם מידע:

[ראה את קוד הדוגמה המלא](#)

[הורד את הדוגמה המלאה \(2KB zipped\)](#)

הערה

mbhddim אחד מן השני. בתוך קוד React-אסטרטגיה זו היא שימושית בעיקר בשחלקים בעמוד שימושים בו. כל יותר להשתמש בחיבור קומפוננטות במקום

בפודקshan JavaScript טיפ: צמצם

לא מצומצם יכול להאט באופן משמעותי העלת האתגר לפודקshan, צריך לזכור כי קוד שלך.

HTML-אם צמצמת כבר את הסקריפטים של האפליקציה שלך, האתר שלך יהיה מוכן לפודקshan במידה אתה מודע שהוא מסתיימות עם `production.min.js`:

```
<script src="https://unpkg.com/react@18/umd/react.production.min.js" crossorigin>
</script>
<script src="https://unpkg.com/react-dom@18/umd/react-dom.production.min.js"
crossorigin></script>
```

אם אין לך שלב צמצום לסקריפטים שלך, הנה דרך לעשות זאת.

օպציונאלי: נסה את JSX

בדוגמאות הקודמות, השתכננו רק על פיצ'רים שנתמכים באופן טבעי על ידי דפדפניים. זאת הסיבה שהשתמשנו בקריאה מה להציג React-על מנת לומר לJavaScript לפונקציית

```
const e = React.createElement;

// הציג כפתרו "לייק"
return e(
  'button',
  { onClick: () => this.setState({ liked: true }) },
  'Like'
);
```

במקום JSX-לעומת זאת, ריאקט מציע אפשרות להשתמש ב:

```
// הציג כפתרו "לייק"
return (
  <button onClick={() => this.setState({ liked: true })}>
    Like
  </button>
);
```

הוא אופציונלי לגמר!, מספר רב של אנשים מוצאים אותו יעיל עבור כתיבת קוד **JSX**-שתי דוגמאות הקוד זהות. למורoutes שוגם בספריות אחרות React-משק משתמש -- ב-

באמצעות **ממיר אונליין זה** JSX תוכל להתנסות עם

במהירות JSX נסה את

הזה לעמוד שלך <script>-בפרויקט שלך היא להוסיף את תגיית JSX הדרך הכى מהירה לנסות את

```
<script src="https://unpkg.com/babel-standalone@6/babel.min.js"></script>
```

הנה **דוגמה של דף**. על ידי הוספת מאפיין <script> **type="text/babel"** בתוך כל Tagית JSX-עכשו תוכל להשתמש ב **שתוכל להוריד ולהתנסות עמה HTML JSX עם**

גישה זו היא טוביה למידה וליצירת הדוגמות פשוטות. לעומת זאת, היא עשויה את אתוך איטי והיא **אינה מתאימה לפודקשן**. שהוספת. במקום זאת, **type="text/babel"** החדש ואות מאפייני <script>-כשהאת מוכן להתקדם, הסר את Tagית ה- **JSX** שלך באופן אוטומטי <script>-שיםיר את כל Tagיות JSX בפרק הבא נכוון קדם-מעבד

לפרויקט JSX הוספה

הוא **תחילה** דומה JSX לפרויקט אינה דורשת כלים מורכבים כמו אנדרואיד או שורת פיתוח. לעומת זאת, הוספה JSX הוספה **CSS**. יהיה מותקן על מחשבך **Node.js**-הדרישה היחידה היא ש **להוספה קדם-מעבד**.

גש אל תקיות הפרויקט שלך בשורת הפקודה, והדבק את שתי הפקודות הבאות:

(אם הוא נכשל, הנה פיתרון) **npm init -y** **npm** **שלב ראשון:** הרץ את 1.

npm install babel-cli@6 babel-preset-react-app@3 **שלב שני:** הרץ את 2.

טיפ

לא יהיה תצערך אותו לדברים נוספים. גם **JSX**-cano על מנת להתקין את קדם-מעבד **הcano**-אנו משתמשים בReact ללא שינויי <script> וגם קוד האפליקציה שלך יכולם להישאר בתגיונות

מוכן לפודקשן לפרויקט שלך JSX מזל טוב! הוספה הרגע.

JSX הרצת קדם-מעבד

והרץ את הפקודה הבאה בשורת הפקודה **src** צור תקייה שנקראת

```
npx babel --watch src --out-dir . --presets react-app/prod
```

הערה

אינה טעות כתיב -- זה **מרכיב חבילות שבא עם npm 5.2+**.

יתכן שפספסת את הצעד, You have mistakenly installed the `babel` package", הקודם. בצע זאת באותה התיקייה, אז נסה שוב

JSX-אל תחכה שההתקן יסתים -- פקודה זאת מתחילה צופה אוטומטי ל

הבסיסי הזה, הצופה יצור קובץ JSX-עם קוד ה `src/like_button.js` אם עכשו תיצור קובץ שנקרו שינויי הצורה ירוץ JSX שמתאים לדפדף. כשאתה עורך את קוד המקור עם קוד ה `src/like_button.js`-מעובד עם קוד ה JSX-על, שוב באופן אוטומטי.

מודרני כמו מחלקות בלי לדאוג לגבי שבירת גרסאות דפדפן קודמות. הכללי JavaScript כבונוס, זה נותן לך להשתמש בקוד ותוכל ללמוד עד לגבי מהדוקומנטציה שלו Babel, שהשתמשנו בו הרגע נקרא!

אם אתה שם לב שאתה מתחיל להרגיש בונה יותר עם כלים בנייה ואתה רוצה שם יעשו יותר עבורך, **התקה הבא** מסביר על הכלים הפופולריים והנגישים ביותר. אם לא -- תגיוטה הסקריפט הללו יעשו את העבודה מצווין!

[Go to TOC](#)

Animation Add-Ons

Note:

`ReactTransitionGroup` and `ReactCSSTransitionGroup` have been moved to the `react-transition-group` package that is maintained by the community. Its 1.x branch is completely API-compatible with the existing addons. Please file bugs and feature requests in the [new repository](#).

The `ReactTransitionGroup` add-on component is a low-level API for animation, and `ReactCSSTransitionGroup` is an add-on component for easily implementing basic CSS animations and transitions.

High-level API: ReactCSSTransitionGroup

`ReactCSSTransitionGroup` is a high-level API based on `ReactTransitionGroup` and is an easy way to perform CSS transitions and animations when a React component enters or leaves the DOM. It's inspired by the excellent [ng-animate](#) library.

Importing

```
import ReactCSSTransitionGroup from 'react-transition-group'; // ES6
var ReactCSSTransitionGroup = require('react-transition-group'); // ES5 with npm
```

```
class TodoList extends React.Component {
  constructor(props) {
    super(props);
    this.state = {items: ['hello', 'world', 'click', 'me']};
    this.handleAdd = this.handleAdd.bind(this);
  }

  handleAdd() {
    const newItems = this.state.items.concat([
      prompt('Enter some text')
    ]);
    this.setState({items: newItems});
  }

  handleRemove(i) {
    let newItems = this.state.items.slice();
    newItems.splice(i, 1);
    this.setState({items: newItems});
  }

  render() {
    const items = this.state.items.map((item, i) => (
      <div key={i} onClick={() => this.handleRemove(i)}>
        {item}
      </div>
    ));
    return (

```

```

        <div>
          <button onClick={this.handleAdd}>Add Item</button>
          <ReactCSSTransitionGroup
            transitionName="example"
            transitionEnterTimeout={500}
            transitionLeaveTimeout={300}>
            {items}
          </ReactCSSTransitionGroup>
        </div>
      );
    }
  }
}

```

Note:

You must provide the `key` attribute for all children of `ReactCSSTransitionGroup`, even when only rendering a single item. This is how React will determine which children have entered, left, or stayed.

In this component, when a new item is added to `ReactCSSTransitionGroup` it will get the `example-enter` CSS class and the `example-enter-active` CSS class added in the next tick. This is a convention based on the `transitionName` prop.

You can use these classes to trigger a CSS animation or transition. For example, try adding this CSS and adding a new list item:

```

.example-enter {
  opacity: 0.01;
}

.example-enter.example-enter-active {
  opacity: 1;
  transition: opacity 500ms ease-in;
}

.example-leave {
  opacity: 1;
}

.example-leave.example-leave-active {
  opacity: 0.01;
  transition: opacity 300ms ease-in;
}

```

You'll notice that animation durations need to be specified in both the CSS and the render method; this tells React when to remove the animation classes from the element and -- if it's leaving -- when to remove the element from the DOM.

Animate Initial Mounting

`ReactCSSTransitionGroup` provides the optional prop `transitionAppear`, to add an extra transition phase at the initial mount of the component. There is generally no transition phase at the initial mount as the default value of `transitionAppear` is `false`. The following is an example which passes the prop `transitionAppear` with the value `true`.

```

render() {
  return (
    <ReactCSSTransitionGroup
      transitionName="example"
      transitionAppear={true}
      transitionAppearTimeout={500}
      transitionEnter={false}
      transitionLeave={false}>
      <h1>Fading at Initial Mount</h1>
    </ReactCSSTransitionGroup>
  );
}

```

During the initial mount `ReactCSSTransitionGroup` will get the `example-appear` CSS class and the `example-appear-active` CSS class added in the next tick.

```

.example-appear {
  opacity: 0.01;
}

.example-appear.example-appear-active {
  opacity: 1;
  transition: opacity .5s ease-in;
}

```

At the initial mount, all children of the `ReactCSSTransitionGroup` will `appear` but not `enter`. However, all children later added to an existing `ReactCSSTransitionGroup` will `enter` but not `appear`.

Note:

The prop `transitionAppear` was added to `ReactCSSTransitionGroup` in version 0.13. To maintain backwards compatibility, the default value is set to `false`.

However, the default values of `transitionEnter` and `transitionLeave` are `true` so you must specify `transitionEnterTimeout` and `transitionLeaveTimeout` by default. If you don't need either enter or leave animations, pass `transitionEnter={false}` or `transitionLeave={false}`.

Custom Classes

It is also possible to use custom class names for each of the steps in your transitions. Instead of passing a string into `transitionName` you can pass an object containing either the `enter` and `leave` class names, or an object containing the `enter`, `enter-active`, `leave-active`, and `leave` class names. If only the `enter` and `leave` classes are provided, the `enter-active` and `leave-active` classes will be determined by appending '`-active`' to the end of the class name. Here are two examples using custom classes:

```

// ...
<ReactCSSTransitionGroup
  transitionName={{ 
    enter: 'enter',
    enterActive: 'enterActive',
    leave: 'leave',
    leaveActive: 'leaveActive',
  }}

```

```

        appear: 'appear',
        appearActive: 'appearActive'
    } }>
{item}
</ReactCSSTransitionGroup>

<ReactCSSTransitionGroup
    transitionName={ {
        enter: 'enter',
        leave: 'leave',
        appear: 'appear'
    } }>
{item2}
</ReactCSSTransitionGroup>
// ...

```

Animation Group Must Be Mounted To Work

In order for it to apply transitions to its children, the `ReactCSSTransitionGroup` must already be mounted in the DOM or the prop `transitionAppear` must be set to `true`.

The example below would **not** work, because the `ReactCSSTransitionGroup` is being mounted along with the new item, instead of the new item being mounted within it. Compare this to the [Getting Started](#) section above to see the difference.

```

render() {
  const items = this.state.items.map((item, i) =>
    <div key={item} onClick={() => this.handleRemove(i)}>
      <ReactCSSTransitionGroup transitionName="example">
        {item}
      </ReactCSSTransitionGroup>
    </div>
  );
}

return (
  <div>
    <button onClick={this.handleAdd}>Add Item</button>
    {items}
  </div>
);
}

```

Animating One or Zero Items

In the example above, we rendered a list of items into `ReactCSSTransitionGroup`. However, the children of `ReactCSSTransitionGroup` can also be one or zero items. This makes it possible to animate a single element entering or leaving. Similarly, you can animate a new element replacing the current element. For example, we can implement a simple image carousel like this:

```

import ReactCSSTransitionGroup from 'react-transition-group';

function ImageCarousel(props) {
  return (
    <div>
      <ReactCSSTransitionGroup
        transitionName="carousel"
        transitionEnterTimeout={300}

```

```

        transitionLeaveTimeout={300}>
          <img src={props.imageSrc} key={props.imageSrc} />
        </ReactCSSTransitionGroup>
      </div>
    );
}

```

Disabling Animations

You can disable animating `enter` or `leave` animations if you want. For example, sometimes you may want an `enter` animation and no `leave` animation, but `ReactCSSTransitionGroup` waits for an animation to complete before removing your DOM node. You can add `transitionEnter={false}` or `transitionLeave={false}` props to `ReactCSSTransitionGroup` to disable these animations.

Note:

When using `ReactCSSTransitionGroup`, there's no way for your components to be notified when a transition has ended or to perform any more complex logic around animation. If you want more fine-grained control, you can use the lower-level `ReactTransitionGroup` API which provides the hooks you need to do custom transitions.

Low-level API: `ReactTransitionGroup`

Importing

```

import ReactTransitionGroup from 'react-addons-transition-group' // ES6
var ReactTransitionGroup = require('react-addons-transition-group') // ES5 with
npm

```

`ReactTransitionGroup` is the basis for animations. When children are declaratively added or removed from it (as in the [example above](#)), special lifecycle methods are called on them.

- `componentWillAppear()`
- `componentDidAppear()`
- `componentWillEnter()`
- `componentDidEnter()`
- `componentWillLeave()`
- `componentDidLeave()`

Rendering a Different Component

`ReactTransitionGroup` renders as a `span` by default. You can change this behavior by providing a `component` prop. For example, here's how you would render a ``:

```

<ReactTransitionGroup component="ul">
  {/* ... */}
</ReactTransitionGroup>

```

Any additional, user-defined, properties will become properties of the rendered component. For example, here's how you would render a `` with CSS class:

```
<ReactTransitionGroup component="ul" className="animated-list">
  {/* ... */}
</ReactTransitionGroup>
```

Every DOM component that React can render is available for use. However, `component` does not need to be a DOM component. It can be any React component you want; even ones you've written yourself! Just write `component={List}` and your component will receive `this.props.children`.

Rendering a Single Child

People often use `ReactTransitionGroup` to animate mounting and unmounting of a single child such as a collapsible panel. Normally `ReactTransitionGroup` wraps all its children in a `span` (or a custom `component` as described above). This is because any React component has to return a single root element, and `ReactTransitionGroup` is no exception to this rule.

However if you only need to render a single child inside `ReactTransitionGroup`, you can completely avoid wrapping it in a `` or any other DOM component. To do this, create a custom component that renders the first child passed to it directly:

```
function FirstChild(props) {
  const childrenArray = React.Children.toArray(props.children);
  return childrenArray[0] || null;
}
```

Now you can specify `FirstChild` as the `component` prop in `<ReactTransitionGroup>` props and avoid any wrappers in the result DOM:

```
<ReactTransitionGroup component={FirstChild}>
  {someCondition ? <MyComponent /> : null}
</ReactTransitionGroup>
```

This only works when you are animating a single child in and out, such as a collapsible panel. This approach wouldn't work when animating multiple children or replacing the single child with another child, such as an image carousel. For an image carousel, while the current image is animating out, another image will animate in, so `<ReactTransitionGroup>` needs to give them a common DOM parent. You can't avoid the wrapper for multiple children, but you can customize the wrapper with the `component` prop as described above.

Reference

`componentWillAppear()` **#componentwillappear**

```
componentWillAppear(callback)
```

This is called at the same time as `componentDidMount()` for components that are initially mounted in a `TransitionGroup`. It will block other animations from occurring until `callback` is called. It is only called on the initial render of a `TransitionGroup`.

`componentDidAppear()` **{#componentdidappear}**

`componentDidAppear()`

This is called after the `callback` function that was passed to `componentWillAppear()` is called.

`componentWillEnter()` **{#componentwillenter}**

`componentWillEnter(callback)`

This is called at the same time as `componentDidMount()` for components added to an existing `TransitionGroup`. It will block other animations from occurring until `callback` is called. It will not be called on the initial render of a `TransitionGroup`.

`componentDidEnter()` **{#componentdidenter}**

`componentDidEnter()`

This is called after the `callback` function that was passed to `componentWillEnter()` is called.

`componentWillLeave()` **{#componentwillleave}**

`componentWillLeave(callback)`

This is called when the child has been removed from the `ReactTransitionGroup`. Though the child has been removed, `ReactTransitionGroup` will keep it in the DOM until `callback` is called.

`componentDidLeave()` **{#componentdidleave}**

`componentDidLeave()`

This is called when the `willLeave` `callback` is called (at the same time as `componentWillUnmount()`).

[Go to TOC](#)

Keyed Fragments

Note:

`React.addons` entry point is deprecated as of React v15.5. We now have first class support for fragments which you can read about [here](#).

Importing

```
import createFragment from 'react-addons-create-fragment'; // ES6
var createFragment = require('react-addons-create-fragment'); // ES5 with npm
```

Overview

In most cases, you can use the `key` prop to specify keys on the elements you're returning from `render`. However, this breaks down in one situation: if you have two sets of children that you need to reorder, there's no way to put a key on each set without adding a wrapper element.

That is, if you have a component such as:

```
function Swapper(props) {
  let children;
  if (props.swapped) {
    children = [props.rightChildren, props.leftChildren];
  } else {
    children = [props.leftChildren, props.rightChildren];
  }
  return <div>{children}</div>;
}
```

The children will unmount and remount as you change the `swapped` prop because there aren't any keys marked on the two sets of children.

To solve this problem, you can use the `createFragment` add-on to give keys to the sets of children.

```
Array<ReactNode> createFragment(object children) {#arrayreactnode-createfragmentobject-children}
```

Instead of creating arrays, we write:

```
import createFragment from 'react-addons-create-fragment';

function Swapper(props) {
  let children;
  if (props.swapped) {
    children = createFragment({
      right: props.rightChildren,
      left: props.leftChildren
    });
  }
  return <div>{children}</div>;
}
```

```
    } else {
      children = createFragment({
        left: props.leftChildren,
        right: props.rightChildren
      });
    }
    return <div>{children}</div>;
}
```

The keys of the passed object (that is, `left` and `right`) are used as keys for the entire set of children, and the order of the object's keys is used to determine the order of the rendered children. With this change, the two sets of children will be properly reordered in the DOM without unmounting.

The return value of `createFragment` should be treated as an opaque object; you can use the `React.Children` helpers to loop through a fragment but should not access it directly. Note also that we're relying on the JavaScript engine preserving object enumeration order here, which is not guaranteed by the spec but is implemented by all major browsers and VMs for objects with non-numeric keys.

Performance Tools

Note:

As of React 16, `react-addons-perf` is not supported. Please use [your browser's profiling tools](#) to get insight into which components re-render.

Importing

```
import Perf from 'react-addons-perf'; // ES6
var Perf = require('react-addons-perf'); // ES5 with npm
```

Overview

React is usually quite fast out of the box. However, in situations where you need to squeeze every ounce of performance out of your app, it provides a `shouldComponentUpdate()` method where you can add optimization hints to React's diff algorithm.

In addition to giving you an overview of your app's overall performance, `Perf` is a profiling tool that tells you exactly where you need to put these methods.

See these articles for an introduction to React performance tooling:

- ["How to Benchmark React Components"](#)
- ["Performance Engineering with React"](#)
- ["A Deep Dive into React Perf Debugging"](#)

Development vs. Production Builds

If you're benchmarking or seeing performance problems in your React apps, make sure you're testing with the [minified production build](#). The development build includes extra warnings that are helpful when building your apps, but it is slower due to the extra bookkeeping it does.

However, the perf tools described on this page only work when using the development build of React. Therefore, the profiler only serves to indicate the *relatively* expensive parts of your app.

Using Perf

The `Perf` object can be used with React in development mode only. You should not include this bundle when building your app for production.

Getting Measurements

- `start()`
- `stop()`

- `getLastMeasurements()`

Printing Results

The following methods use the measurements returned by `Perf.getLastMeasurements()` to pretty-print the result.

- `printInclusive()`
 - `printExclusive()`
 - `printWasted()`
 - `printOperations()`
 - `printDOM()`
-

Reference

`start() {#start}`

`stop() {#stop}`

```
Perf.start()  
// ...  
Perf.stop()
```

Start/stop the measurement. The React operations in-between are recorded for analyses below. Operations that took an insignificant amount of time are ignored.

After stopping, you will need `Perf.getLastMeasurements()` to get the measurements.

`getLastMeasurements() {#getlastmeasurements}`

```
Perf.getLastMeasurements()
```

Get the opaque data structure describing measurements from the last start-stop session. You can save it and pass it to the other print methods in `Perf` to analyze past measurements.

Note

Don't rely on the exact format of the return value because it may change in minor releases. We will update the documentation if the return value format becomes a supported part of the public API.

`printInclusive() {#printinclusive}`

```
Perf.printInclusive(measurements)
```

Prints the overall time taken. When no arguments are passed, `printInclusive` defaults to all the measurements from the last recording. This prints a nicely formatted table in the console, like so:

(index)	Owner > component	Inclusive time (ms)	Instances
0	"<root> > App"	15.31	37
1	"App > Box"	6.47	37
2	"Box > Box2"	1.96	37

printExclusive() `{#printexclusive}`

`Perf.printExclusive(measurements)`

"Exclusive" times don't include the times taken to mount the components: processing props, calling `componentWillMount` and `componentDidMount`, etc.

(index)	Component c...	Total inclus...	Exclusive m...	Exclusive r...	Mount time ...	Render time...	Instances
0	"App"	15.31	1.76	1.23	0.04	0.03	37

printWasted() `{#printwasted}`

`Perf.printWasted(measurements)`

The most useful part of the profiler.

"Wasted" time is spent on components that didn't actually render anything, e.g. the render stayed the same, so the DOM wasn't touched.

(index)	Owner > component	Wasted time (ms)	Instances
0	"<root> > App"	15.317999947001226	37
1	"App > Box"	6.479999952716753	37
2	"Box > Box2"	1.9630000460892916	37

printOperations() `{#printoperations}`

`Perf.printOperations(measurements)`

Prints the underlying DOM manipulations, e.g. "set innerHTML" and "remove".

(index)	data-reactid	type	args
0	"0"	"set innerHTML"	"<div data-reactid=".0\..."

printDOM() `{#printdom}`

`Perf.printDOM(measurements)`

This method has been renamed to `printOperations()`. Currently `printDOM()` still exists as an alias but it prints a deprecation warning and will eventually be removed.

PureRenderMixin

Note:

`PureRenderMixin` is a legacy add-on. Use `React.PureComponent` instead.

Importing

```
import PureRenderMixin from 'react-addons-pure-render-mixin'; // ES6
var PureRenderMixin = require('react-addons-pure-render-mixin'); // ES5 with npm
```

Overview

If your React component's render function renders the same result given the same props and state, you can use this mixin for a performance boost in some cases.

Example:

```
const createReactClass = require('create-react-class');

createReactClass({
  mixins: [PureRenderMixin],

  render: function() {
    return <div className={this.props.className}>foo</div>;
  }
});
```

Under the hood, the mixin implements `shouldComponentUpdate`, in which it compares the current props and state with the next ones and returns `false` if the equalities pass.

Note:

This only shallowly compares the objects. If these contain complex data structures, it may produce false-negatives for deeper differences. Only mix into components which have simple props and state, or use `forceUpdate()` when you know deep data structures have changed. Or, consider using [immutable objects](#) to facilitate fast comparisons of nested data.

Furthermore, `shouldComponentUpdate` skips updates for the whole component subtree. Make sure all the children components are also "pure".

[Go to TOC](#)

Shallow Compare

Note:

`shallowCompare` is a legacy add-on. Use `React.memo` or `React.PureComponent` instead.

Importing

```
import shallowCompare from 'react-addons-shallow-compare'; // ES6
var shallowCompare = require('react-addons-shallow-compare'); // ES5 with npm
```

Overview

Before `React.PureComponent` was introduced, `shallowCompare` was commonly used to achieve the same functionality as `PureRenderMixin` while using ES6 classes with React.

If your React component's render function is "pure" (in other words, it renders the same result given the same props and state), you can use this helper function for a performance boost in some cases.

Example:

```
export class SampleComponent extends React.Component {
  shouldComponentUpdate(nextProps, nextState) {
    return shallowCompare(this, nextProps, nextState);
  }

  render() {
    return <div className={this.props.className}>foo</div>;
  }
}
```

`shallowCompare` performs a shallow equality check on the current `props` and `nextProps` objects as well as the current `state` and `nextState` objects.

It does this by iterating on the keys of the objects being compared and returning true when the values of a key in each object are not strictly equal.

`shallowCompare` returns `true` if the shallow comparison for props or state fails and therefore the component should update.

`shallowCompare` returns `false` if the shallow comparison for props and state both pass and therefore the component does not need to update.

[Go to TOC](#)

מרנדר רדוד

ייבורא

```
import ShallowRenderer from 'react-test-renderer/shallow'; // ES6
var ShallowRenderer = require('react-test-renderer/shallow'); // npm ערך ES5
```

סקירה כללית

מרנדר רדוד יכול לעזור. מרנדר רדוד נותן לך לレンדר קומפוננטה "רמת עומק אחת", React, בעת כתיבת בדיקות ייחודית עבורו ולוחאת על מה המתודהレンדר מחייבת, מוביל לדאגן על התנהוגותם של קומפוננטות הילדים שלה, אשר לא מייצרים מופיע חדש או מתורנדים. זה לא דרוש DOM.

לדוגמה, אם יש לך את הקומפוננטה הבאה:

```
function MyComponent() {
  return (
    <div>
      <span className="heading">כותרת</span>
      <Subcomponent foo="bar" />
    </div>
  );
}
```

אז אתה יכול לוודא:

```
import ShallowRenderer from 'react-test-renderer/shallow';

// בבדיקה שלא:
const renderer = new ShallowRenderer();
renderer.render(<MyComponent />);
const result = renderer.getRenderOutput();

expect(result.type).toBe('div');
expect(result.props.children).toEqual([
  <span className="heading">כותרת</span>,
  <Subcomponent foo="bar" />
]);
```

-למרנדר רדוד כרגע יש כמה מגבלות, למשל חוסר תמיכה ב refs.

הערה:

מרמה-גבואה יותר על אותה API הוא מספק Enzyme. של המרנדר הרדוד API- אנחנו גם ממליצים לבדוק את ה פונקציונליות.

יעון

`shallowRenderer.render()` {#shallowrendererrender}

כ-“מקום” לרנדור קומפוננטה שאתה בודק, וממנו ניתן להוציא את הפלט של shallowRenderer אתה יכול לחשב על הקומפוננטה.

<<<<< HEAD `shallowRenderer.render()` -זומה ל `ReactDOM.render()` . רמת עומק אחת. זה אומר שאתה יכול לבדוק קומפוננטות מבודדות מבלי לדעת איך הילדים שלהם ממושכים `shallowRenderer.render()` is similar to `root.render()` but it doesn't require DOM and only renders a single level deep. This means you can test components isolated from how their children are implemented.

84ad3308338e2bb819f4f24fa8e9dfeeffaa970b

`shallowRenderer.getRenderOutput()` {#shallowrenderergetrenderoutput}

כדי `shallowRenderer.render()` -נקרא, אתה יכול להשתמש ב `shallowRenderer.getRenderOutput()` -אחרי שתקבל את הפלט הרדו שرونדר.

אחריו זה אתה יכול להתחילה לשאל עבודות על הפלט.

[Go to TOC](#)

כלי בדיקה

ייבוא

```
import ReactTestUtils from 'react-dom/test-utils'; // ES6
var ReactTestUtils = require('react-dom/test-utils'); // ES5 עם npm
```

סקירה כללית

אנו משתמשים Facebook-בכל פרימוֹרְק בבדיקה שתבחר. ב React מקל על תהליך בדיקת קומפוננטות `ReactTestUtils` מילא את התפקיד בדיקת קומפוננטות. ב Jest באתר האינטרנט של [React](#) ניתן לראות דרך [MDN](#) מדריך Jest ב-JavaScript.

הערה:

אחרת לשות מניפולציות על פלטי קומפוננטות בקלות [Enzyme](#) לגרסאות ריאקט מתחתי או בגרסה 16, ספריית.

- `act()`
- `mockComponent()`
- `isElement()`
- `isElementOfType()`
- `isDOMComponent()`
- `isCompositeComponent()`
- `isCompositeComponentWithType()`
- `findAllInRenderedTree()`
- `scryRenderedDOMComponentsWithClass()`
- `findRenderedDOMComponentWithClass()`
- `scryRenderedDOMComponentsWithTag()`
- `findRenderedDOMComponentWithTag()`
- `scryRenderedComponentsWithType()`
- `findRenderedComponentWithType()`
- `renderIntoDocument()`
- `Simulate`

סימוכין

`act()` {#act}

פעולה זו גורמת `.act()` כדי להציג קומפוננטה לויזוא, עטוף את הקוד שמרנדר אותה ומבצע עליה עדכונים בתוך קריית. עובדת בדפדפן React לבדיקה שלך לרווח אופן דומה לצורה שבה.

הערה

ש망תנהגת באוֹתָה צוֹרָה `act` הַסְּפִירִיה גַם מְסֻפֶּקֶת מְתוּדָת , react-test-renderer - אֵם אַתָּה מְשֻׁתָּמֵשׁ ב

הבאָה Counter לדוגמה, נגיד שיש לנו את קומפוננטה:

```
class Counter extends React.Component {
  constructor(props) {
    super(props);
    this.state = {count: 0};
    this.handleClick = this.handleClick.bind(this);
  }
  componentDidMount() {
    document.title = `You clicked ${this.state.count} times`;
  }
  componentDidUpdate() {
    document.title = `You clicked ${this.state.count} times`;
  }
  handleClick() {
    this.setState(state => ({
      count: state.count + 1,
    }));
  }
  render() {
    return (
      <div>
        <p>You clicked {this.state.count} times</p>
        <button onClick={this.handleClick}>
          Click me
        </button>
      </div>
    );
  }
}
```

ככה נוכל לבדוק אותה:

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import { act } from 'react-dom/test-utils';
import Counter from './Counter';

let container;

beforeEach(() => {
  container = document.createElement('div');
  document.body.appendChild(container);
});

afterEach(() => {
  document.body.removeChild(container);
  container = null;
});

it('can render and update a counter', () => {
  // וְאַתָּה בְּדִיקַת רִינְדוֹר רַאשְׁׂוֹן
  act(() => {
    ReactDOM.createRoot(container).render(<Counter />);
  });
  const button = container.querySelector('button');
  const label = container.querySelector('p');
```

```
expect(label.textContent).toBe('You clicked 0 times');
expect(document.title).toBe('You clicked 0 times');

//-בדיקה רינדור שמי ו-componentDidUpdate
act(() => {
  button.dispatchEvent(new MouseEvent('click', {bubbles: true}));
});
expect(label.textContent).toBe('You clicked 1 times');
expect(document.title).toBe('You clicked 1 times');
});
```

ניתן להשתמש בעוזר כמו `document`.-נוסף ל DOM-עובד רק כשקונטיינר ה DOM אסור לשוכח שsigmoid אירובי `react-testing-library` על מנת להפחית קוד תבנית קבועה.

עם דוגמאות ודרכי שימוש, `act()` מכיל עוד מידע על ההתנהגות של המסמך `recipes`.

`mockComponent()` **#mockcomponent**

```
mockComponent(
  componentClass,
  [mockTagName]
)
```

הعبر מודול קומפוננטה "מזויפת" למתחודה זו על מנת להרוחיב אותה עם מתודות שימושיות שנוטנוות את האפשרות להשתמש או תג אחר אם סופק גם פשוט `<div>`-מודומה. במקום לレンדר כרגע, הקומפוננטה תהפוך ל `React` בה כקומפוננטה `mockTagName` שמכיל את הילדים שסופקו () .

הערה:

במקום `(())` ישן. אנו ממליצים להשתמש ב-רינדור רדום או ב API הוא `jest.mock()`.

`isElement()` **#iselement**

```
isElement(element)
```

כלשחו `true` אם `element` הוא אלמנט React. מוחזיר

`isElementOfType()` **#iselementoftype**

```
isElementOfType(
  element,
  componentClass
)
```

מصحה `true` אם `element` מסוג React הוא אלמנט `componentClass` של React. מוחזיר

isDOMComponent() {#isdomcomponent}

```
isDOMComponent(instance)
```

מחזיר `true` אם `instance` הוא קומפוננטה DOM (כמו `<div>` או ``).

isCompositeComponent() {#iscompositecomponent}

```
isCompositeComponent(instance)
```

הוא קומפוננטה שהוגדרה על ידי המפתח, לדוגמה מחלקה או פונקציה `true` אם `instance` מחזיר.

isCompositeComponentWithType() {#iscompositecomponentwithtype}

```
isCompositeComponentWithType(
  instance,
  componentClass
)
```

מחזיר `true` אם `instance` הוא קומפוננטה מסווג `componentClass` של React.

findAllInRenderedTree() {#findallinrenderedtree}

```
findAllInRenderedTree(
  tree,
  test
)
```

זה לא כל כך. הוא `true` אם `test(component)` הוצאה את כל הקומפוננטות ב-`tree` וצובר את כל הקומפוננטות שבו. שימושי לפני עצמו, אבל זה משמש כבסיס לבדיקה אחרים.

scryRenderedDOMComponentsWithClass()
{#scryrendereddomcomponentswithclass}

```
scryRenderedDOMComponentsWithClass(
  tree,
  className
)
```

עם שם המחלקת התואם DOM של קומפוננטות בעץ המרונדר שהן קומפוננטות DOM-מattaר את כל אלמנטי `className`.

findRenderedDOMComponentWithClass()
{#findrendereddomcomponentwithclass}

```
findRenderedDOMComponentWithClass(
  tree,
  className
)
```

אבל מטרת שתהיה תוצאה אחת, ומהזיר את התוצאה האחת הזו, או `(())` כמו `scryRenderedDOMComponentsWithClass()` זורק שגיאה אם יש מספר אחר של תוצאות מעבר לאחת.

`scryRenderedDOMComponentsWithTag()` {#scryrendereddomcomponentswithtag}

```
scryRenderedDOMComponentsWithTag(
  tree,
  tagName
)
```

אם שם `tagName` מתייחס לTAG שקיים בהTREE מטרת שתהיה תוצאה אחת עם שמו.

`findRenderedDOMComponentWithTag()` {#findrendereddomcomponentwithtag}

```
findRenderedDOMComponentWithTag(
  tree,
  tagName
)
```

אבל מטרת שתהיה תוצאה אחת, ומהזיר את התוצאה האחת הזו, או זורק `(())` כמו `scryRenderedDOMComponentsWithTag()` שגיאה אם יש מספר אחר של תוצאות מעבר לאחת.

`scryRenderedComponentsWithType()` {#scryrenderedcomponentswithtype}

```
scryRenderedComponentsWithType(
  tree,
  componentClass
)
```

אם שם `componentClass` מתייחס לCOMPONENT שקיים בהTREE מטרת שתהיה תוצאה אחת.

`findRenderedComponentWithType()` {#findrenderedcomponentwithtype}

```
findRenderedComponentWithType(
  tree,
  componentClass
)
```

אבל מטרת שתהיה תוצאה אחת, ומהזיר את התוצאה האחת הזו, או זורק שגיאה אם יש מספר אחר של תוצאות מעבר לאחת.

`renderIntoDocument()` {#renderintodocument}

```
renderIntoDocument(element)
```

נוכח. זה שווה ערך ל **DOM** מנותקת בדף. **פונקציה זו דורשת DOM** לצומת React מרנדר אלמנט

```
const domContainer = document.createElement('div');
ReactDOM.createRoot(domContainer).render(element);
```

הערה:

זמןניים באופן גלובלי **לפני** `window`, `window.document` או `window.createElement` יש צורך ב-`לא יעדכו` וMETHODS כמו DOM-תחשוב שאין לה גישה ל- `React`. `setState` שמיבאים את

כליים אחרים

Simulate {#simulate}

```
Simulate.{eventName}(
  element,
  [eventData]
)
```

אפשרונאליים עם נתונים אירוע DOM מדמה שיגור אירוע על צומת `eventData`.

מビינה של `Simulate` מתחודה **לכל אירוע**.

לחיצה על אלמנט

```
// <button ref={(node) => this.button = node}>...</button>
const node = this.button;
ReactTestUtils.Simulate.click(node);
```

שינוי ערך של שדה קלט וואז לחיצת **ENTER**.

```
// <input ref={(node) => this.textInput = node} />
const node = this.textInput;
node.value = 'giraffe';
ReactTestUtils.Simulate.change(node);
ReactTestUtils.Simulate.keyDown(node, {key: "Enter", keyCode: 13, which: 13});
```

הערה

לא-React-מקיון ש ('וכו `keyCode`, `which`) תצטרך לספק כל מאפיין אירוע שאתה משתמש בו בקומפוננטה שלך. יוצרת אף אחד מהם עבורך.

[Go to TOC](#)

Two-way Binding Helpers

Note:

`LinkedStateMixin` is deprecated as of React v15. The recommendation is to explicitly set the value and change handler, instead of using `LinkedStateMixin`.

Importing

```
import LinkedStateMixin from 'react-addons-linked-state-mixin'; // ES6
var LinkedStateMixin = require('react-addons-linked-state-mixin'); // ES5 with npm
```

Overview

`LinkedStateMixin` is an easy way to express two-way binding with React.

In React, data flows one way: from owner to child. We think that this makes your app's code easier to understand. You can think of it as "one-way data binding."

However, there are lots of applications that require you to read some data and flow it back into your program. For example, when developing forms, you'll often want to update some React `state` when you receive user input. Or perhaps you want to perform layout in JavaScript and react to changes in some DOM element size.

In React, you would implement this by listening to a "change" event, read from your data source (usually the DOM) and call `setState()` on one of your components. "Closing the data flow loop" explicitly leads to more understandable and easier-to-maintain programs. See [our forms documentation](#) for more information.

Two-way binding -- implicitly enforcing that some value in the DOM is always consistent with some React `state` -- is concise and supports a wide variety of applications. We've provided `LinkedStateMixin`: syntactic sugar for setting up the common data flow loop pattern described above, or "linking" some data source to React `state`.

Note:

`LinkedStateMixin` is just a thin wrapper and convention around the `onChange / setState()` pattern. It doesn't fundamentally change how data flows in your React application.

LinkedStateMixin: Before and After

Here's a simple form example without using `LinkedStateMixin`:

```
var createReactClass = require('create-react-class');

var NoLink = createReactClass({
  getInitialState: function() {
    return {message: 'Hello!'};
  },
  handleChange: function(event) {
    this.setState({message: event.target.value});
  },
  render: function() {
    var message = this.state.message;
    return <input type="text" value={message} onChange={this.handleChange} />;
  }
});
```

This works really well and it's very clear how data is flowing, however, with a lot of form fields it could get a bit verbose. Let's use `LinkedStateMixin` to save us some typing:

```
var createReactClass = require('create-react-class');

var WithLink = createReactClass({
  mixins: [LinkedStateMixin],
  getInitialState: function() {
    return {message: 'Hello!'};
  },
  render: function() {
    return <input type="text" valueLink={this.linkState('message')} />;
  }
});
```

`LinkedStateMixin` adds a method to your React component called `linkState()`. `linkState()` returns a `valueLink` object which contains the current value of the React state and a callback to change it.

`valueLink` objects can be passed up and down the tree as props, so it's easy (and explicit) to set up two-way binding between a component deep in the hierarchy and state that lives higher in the hierarchy.

Note that checkboxes have a special behavior regarding their `value` attribute, which is the value that will be sent on form submit if the checkbox is checked (defaults to `on`). The `value` attribute is not updated when the checkbox is checked or unchecked. For checkboxes, you should use `checkedLink` instead of `valueLink`:

```
<input type="checkbox" checkedLink={this.linkState('booleanValue')} />
```

Under the Hood

There are two sides to `LinkedStateMixin`: the place where you create the `valueLink` instance and the place where you use it. To prove how simple `LinkedStateMixin` is, let's rewrite each side separately to be more explicit.

valueLink Without LinkedStateMixin

```
var createReactClass = require('create-react-class');

var WithoutMixin = createReactClass({
```

```

getInitialState: function() {
  return {message: 'Hello!'};
},
handleChange: function(newValue) {
  this.setState({message: newValue});
},
render: function() {
  var valueLink = {
    value: this.state.message,
    requestChange: this.handleChange
  };
  return <input type="text" valueLink={valueLink} />;
}
);

```

As you can see, `valueLink` objects are very simple objects that just have a `value` and `requestChange` prop. And `LinkedStateMixin` is similarly simple: it just populates those fields with a `value` from `this.state` and a callback that calls `this.setState()`.

LinkedStateMixin Without valueLink

```

var LinkedStateMixin = require('react-addons-linked-state-mixin');
var createReactClass = require('create-react-class');

var WithoutLink = createReactClass({
  mixins: [LinkedStateMixin],
  getInitialState: function() {
    return {message: 'Hello!'};
  },
  render: function() {
    var valueLink = this.linkState('message');
    var handleChange = function(e) {
      valueLink.requestChange(e.target.value);
    };
    return <input type="text" value={valueLink.value} onChange={handleChange} />;
  }
});

```

The `valueLink` prop is also quite simple. It simply handles the `onChange` event and calls `this.props.valueLink.requestChange()` and also uses `this.props.valueLink.value` instead of `this.props.value`. That's it!

Immutability Helpers

Note:

`update` is a legacy add-on. Use `immutability-helper` instead.

Importing

```
import update from 'react-addons-update'; // ES6
var update = require('react-addons-update'); // ES5 with npm
```

Overview

React lets you use whatever style of data management you want, including mutation. However, if you can use immutable data in performance-critical parts of your application it's easy to implement a fast `shouldComponentUpdate()` method to significantly speed up your app.

Dealing with immutable data in JavaScript is more difficult than in languages designed for it, like [Clojure](#). However, we've provided a simple immutability helper, `update()`, that makes dealing with this type of data much easier, *without* fundamentally changing how your data is represented. You can also take a look at Facebook's [Immutable-js](#) and the [Advanced Performance](#) section for more detail on Immutable-js.

The Main Idea

If you mutate data like this:

```
myData.x.y.z = 7;
// or...
myData.a.b.push(9);
```

You have no way of determining which data has changed since the previous copy has been overwritten. Instead, you need to create a new copy of `myData` and change only the parts of it that need to be changed. Then you can compare the old copy of `myData` with the new one in `shouldComponentUpdate()` using triple-equals:

```
const newData = deepCopy(myData);
newData.x.y.z = 7;
newData.a.b.push(9);
```

Unfortunately, deep copies are expensive, and sometimes impossible. You can alleviate this by only copying objects that need to be changed and by reusing the objects that haven't changed. Unfortunately, in today's JavaScript this can be cumbersome:

```
const newData = extend(myData, {
  x: extend(myData.x, {
    y: extend(myData.x.y, {z: 7}),
```

```
  },
  a: extend(myData.a, {b: myData.a.b.concat(9)})
});
```

While this is fairly performant (since it only makes a shallow copy of `log n` objects and reuses the rest), it's a big pain to write. Look at all the repetition! This is not only annoying, but also provides a large surface area for bugs.

update() {#update}

`update()` provides simple syntactic sugar around this pattern to make writing this code easier. This code becomes:

```
import update from 'react-addons-update';

const newData = update(myData, {
  x: {y: {z: {$set: 7}}},
  a: {b: {$push: [9]}}
});
```

While the syntax takes a little getting used to (though it's inspired by MongoDB's query language) there's no redundancy, it's statically analyzable and it's not much more typing than the mutative version.

The `$`-prefixed keys are called *commands*. The data structure they are "mutating" is called the *target*.

Available Commands

- `={$push: array} push()` all the items in `array` on the target.
- `={$unshift: array} unshift()` all the items in `array` on the target.
- `={$splice: array of arrays} for each item in arrays call splice()` on the target with the parameters provided by the item.
- `={$set: any} replace the target entirely.`
- `={$merge: object} merge the keys of object with the target.`
- `={$apply: function} passes in the current value to the function and updates it with the new returned value.`

Examples

Simple push

```
const initialArray = [1, 2, 3];
const newArray = update(initialArray, {$push: [4]}); // => [1, 2, 3, 4]
```

`initialArray` is still `[1, 2, 3]`.

Nested collections

```
const collection = [1, 2, {a: [12, 17, 15]}];
const newCollection = update(collection, {2: {a: {$splice: [[1, 1, 13, 14]]}}});
// => [1, 2, {a: [12, 13, 14, 15]}]
```

This accesses `collection`'s index `2`, key `a`, and does a splice of one item starting from index `1` (to remove `17`) while inserting `13` and `14`.

Updating a value based on its current one

```
const obj = {a: 5, b: 3};
const newObj = update(obj, {b: {$apply: function(x) {return x * 2;}}});
// => {a: 5, b: 6}
// This is equivalent, but gets verbose for deeply nested collections:
const newObj2 = update(obj, {b: {$set: obj.b * 2}});
```

(Shallow) Merge

```
const obj = {a: 5, b: 3};
const newObj = update(obj, {$merge: {b: 6, c: 7}}); // => {a: 5, b: 6, c: 7}
```

Add-Ons

Note:

`React.addons` entry point is deprecated as of React v15.5. The add-ons have moved to separate modules, and some of them have been deprecated.

The React add-ons are a collection of useful utility modules for building React apps. **These should be considered experimental** and tend to change more often than the core.

- `createFragment`, to create a set of externally-keyed children.

The add-ons below are in the development (unminified) version of React only:

- `Perf`, a performance profiling tool for finding optimization opportunities.
- `ReactTestUtils`, simple helpers for writing test cases.

Legacy Add-ons

The add-ons below are considered legacy and their use is discouraged. They will keep working in observable future, but there is no further development.

- `PureRenderMixin`. Use `React.PureComponent` instead.
- `shallowCompare`, a helper function that performs a shallow comparison for props and state in a component to decide if a component should update. We recommend using `React.PureComponent` instead.
- `update`. Use `kolodny/immutability-helper` instead.
- `ReactDOMFactories`, pre-configured DOM factories to make React easier to use without JSX.

Deprecated Add-ons

- `LinkedStateMixin` has been deprecated.
- `TransitionGroup` and `CSSTransitionGroup` have been deprecated in favor of [their drop-in replacements](#).

Using React with Add-ons

You can install the add-ons individually from npm (e.g. `npm install react-addons-create-fragment`) and import them:

```
import createFragment from 'react-addons-create-fragment'; // ES6
var createFragment = require('react-addons-create-fragment'); // ES5 with npm
```

When using React 15 or earlier from a CDN, you can use `react-with-addons.js` instead of `react.js`:

```
<script src="https://unpkg.com/react@15/dist/react-with-addons.js"></script>
```

The add-ons will be available via the `React.addons` global (e.g. `React.addons.TestUtils`).

[Go to TOC](#)

קישורי CDN

React ו-ReactDOM דרך ה-CDN זמינים בדרך זו.

```
<script crossorigin src="https://unpkg.com/react@18/umd/react.development.js">
</script>
<script crossorigin src="https://unpkg.com/react-dom@18/umd/react-
dom.development.js"></script>
```

של production גרסאות מומוחזרות ומואופטומות של production-הגרסאות המצוינות לעיל מתיחסות לפיתוח ולא מתאימות ל-React:

```
<script crossorigin src="https://unpkg.com/react@18/umd/react.production.min.js">
</script>
<script crossorigin src="https://unpkg.com/react-dom@18/umd/react-
dom.production.min.js"></script>
```

החלף את המספר 17 במספר גרסא אחר על מנת לטען גרסא ספציפית של React ו- react-dom .
===== To load a specific version of react and react-dom , replace 18 with the version number.

84ad3308338e2bb819f4f24fa8e9dfeeffaa970b

למה משמשת תכונת crossorigin ?

אנו ממליצים להשתמש בסط מאפייני, דרך ה-CDN או React אם משתמש ב-

```
<script crossorigin src="... "></script>
```

HTTP header - Access-Control-Allow-Origin: * .
הבו משתמשים משתמשים CDN-בנוסף, אנו ממליצים לוודא שה

The screenshot shows a network request details view. At the top, there are tabs for Headers, Preview, Response, and Timing. The Headers tab is selected. Below it, under the General section, there are fields for Request URL (https://unpkg.com/react@16.0.0/umd/react.development.js), Request Method (GET), Status Code (200), Remote Address (124.0.0.1:6811), and Referrer Policy (no-referrer-when-downgrade). Under the Response Headers section, two entries are listed: 'access-control-allow-origin: *' and 'cache-control: public, max-age=31536000'.

Access-Control-Allow-Origin: *

שימוש זה מאפשר **חוויות טיפול בשגיאות** טוביה יותר בגרסאות 16 ומעלה של ריאקט.

[Go to TOC](#)

פיצול-קוד

איגוד

איגוד הוא תהליך שעוקב אחרי קבצים מיו באים ומאחד אותם לקובץ יחיד: "באנדל". את הבאנדל אפשר לצרף לדף אינטרנט כדי לטעון אפליקציה שלמה בביטחון אחת.

דוגמא

:אפליקציה

```
// app.js
import { add } from './math.js';

console.log(add(16, 26)); // 42
```

```
// math.js
export function add(a, b) {
  return a + b;
}
```

:באנדל

```
function add(a, b) {
  return a + b;
}

console.log(add(16, 26)); // 42
```

הערה:

הדוגמה להמחשה בלבד, וקובץ הבאנדל במצבות נראה שונה לאחרי.

וכדומה, תוכלו להשתמש בהתקנה [Create React App](#), [Next.js](#), [Gatsby](#) אם יצרתם את האפליקציה בעוזרת כלים כמו כדי לאגד את האפליקציה Webpack המובנית של Webpack.

אם לא, תצטרכו להתקין כל איגוד בעצמכם. לדוגמה, קראו את המדריכים [להתקנת ומשתמש המתחיל](#) בדוקומנטציה של Webpack.

פיצול קוד

איגוד הוא תהליך נחמד, אבל כשהאפליקציה שלכם גדלה, קובץ הבאנדל גדול אליה. בעיקר אם אתם משתמשים בספריות צד שלישי גדולות. תצטרכו לשים עין על הקוד שכלל בבאנדל כדי שהקובץ לא יהיה גדול שיגרום לאפליקציה שלכם להטען לפחות מדי.

כדי שלא נגיע לנקודת שהバンドל שלנו גדול מדי, כדאי להקדים את הבעה ולהתחליל "לפצל" את הבאנדל. פיצול קוד הוא פיצ'ר שמאפשר ליצור מספר קבציバンドל שיטענו, (Browserify [factor-bundle](#) בעורת) באנדרואיד.

פיצול קוד יכול לעזור לנו למשוך "טעינה עצלה" של הדברים באפליקציה שהמשתמש צריך, וטעינה זאת יכולה לשפר באופן משמעותי את הביצועים של האפליקציה. בצורה זאת לא נאלץ להפחית את כמות הקוד, ורק לא לטען קוד שהמשתמש אולי לא יצטרך לעולם, ונפחית את כמות הקוד שנדרשת בזמן הטעינה הראשונה של האפליקציה.

import() {#import}

הдинמי `import()` -הדרך הטובה ביותר לפצל קוד באפליקציה היא דרך סינטקס זה:

לפני:

```
import { add } from './math';
console.log(add(16, 26));
```

אחריו:

```
import("./math").then(math => {
  console.log(math.add(16, 26));
});
```

- מוצא סינטקס כזה, הוא מתחילה לפצל את הקוד באפליקציה באופן אוטומטי. אם השתמשם ב Create React App - תוכלו [להתחליל להשתמש בפיצול קוד](#) באופן מיידי. זה מתפרק גם באופן מובנה ב, רק על מנת מוגברת ב [Next.js](#).

שלכם אמרו Webpack - בעצמכם, כנראה תרצו לקרוא את [המדריך לפיצול קוד](#). קובץ קונפיגורציית Webpack אם התקנתם [React](#) בערך [כאן](#).

הдинמי בili לתרגם אותו. בשביל זה, import - יכול לפרש את סינטקס [Babel](#) - תצטרכו לוודא ש [@babel/plugin-syntax-dynamic-import](#).

React.lazy {#reactlazy}

<<<<< HEAD

הערה:

לא זמינים בינתיים למימוש בצד השרת. אם תרצו לפצל קוד שמרונדר בצד השרת, מומלץ [Suspense](#) - או להשתמש ב- [קומפוננטות נתענות](#). הנה [מדריך נחמד לשימוש בפיצול קוד בצד השרת](#).

עזרה לרנדר יבוא -`React.lazy` פונקציה ה- דינמי בקומפוננטה רגילה

The `React.lazy` function lets you render a dynamic import as a regular component.

84ad3308338e2bb819f4f24fa8e9dfeeffaa970b

:לפני

```
import OtherComponent from './OtherComponent';

function MyComponent() {
  return (
    <div>
      <OtherComponent />
    </div>
  );
}
```

:אחרי

```
const OtherComponent = React.lazy(() => import('./OtherComponent'));
```

בצורה אוטומטית כשהקומפוננטה מ戎דרת לראשונה `OtherComponent` הקוד ייען את הבאנדל שמכיל אל הקומפוננטה.

ש망פרשת למדו עם `Promise` דינמי. הוא חייב להחזיר `import()`-מקביל פונקציה שחיבת לקרוא ל `default export` שמכיל קומפוננטה `React`.

Suspense

שמאפשרת לנו להציג תוכן בסיס(כגון מחוון טעינה) בזמן שהוא צריך להתרנדר בתוך קומפוננטה `Suspense`.

מרונדרת, צריך `OtherComponent` עדין לא נטען כשהקומפוננטה `MyComponent` מקבל שמקל את הקומפוננטה `OtherComponent` עד שהוא מוכנה - כמו מחוון טעינה. אפשר לעשות זאת בעזרת קומפוננטה `Suspense`.

```
import React, { Suspense } from 'react';
const OtherComponent = React.lazy(() => import('./OtherComponent'));
```

-ה `fallback` prop מקבל אלמנטReact לשים את קומפוננטה `Suspense` בכל מקום מעל לקומפוננטה העצלה. אפשר אפילו לעטוף מספר קומפוננטות עצלות עם קומפוננטה `Suspense` אחת.

```
import React, { Suspense } from 'react';
const OtherComponent = React.lazy(() => import('./OtherComponent'));
const AnotherComponent = React.lazy(() => import('./AnotherComponent'));

function MyComponent() {
  return (
    <div>
      <Suspense fallback={<div>Loading...</div>}>
        <section>
          <OtherComponent />
          <AnotherComponent />
        </section>
      </Suspense>
    </div>
  );
}
```

<<<<< HEAD ### גבולות שגיאה {#error-boundaries} ===== ## Avoiding fallbacks {#avoiding-fallbacks} Any component may suspend as a result of rendering, even components that were already shown to the user. In order for screen content to always be consistent, if an already shown component suspends, React has to hide its tree up to the closest `<Suspense>` boundary. However, from the user's perspective, this can be disorienting.

Consider this tab switcher:

```
import React, { Suspense } from 'react';
import Tabs from './Tabs';
import Glimmer from './Glimmer';

const Comments = React.lazy(() => import('./Comments'));
const Photos = React.lazy(() => import('./Photos'));

function MyComponent() {
  const [tab, setTab] = React.useState('photos');

  function handleTabSelect(tab) {
    setTab(tab);
  };
}
```

```

return (
  <div>
    <Tabs onTabSelect={handleTabSelect} />
    <Suspense fallback={<Glimmer />}>
      {tab === 'photos' ? <Photos /> : <Comments />}
    </Suspense>
  </div>
);
}

```

In this example, if tab gets changed from `'photos'` to `'comments'`, but `Comments` suspends, the user will see a glimmer. This makes sense because the user no longer wants to see `Photos`, the `Comments` component is not ready to render anything, and React needs to keep the user experience consistent, so it has no choice but to show the `Glimmer` above.

However, sometimes this user experience is not desirable. In particular, it is sometimes better to show the "old" UI while the new UI is being prepared. You can use the new `startTransition` API to make React do this:

```

function handleTabSelect(tab) {
  startTransition(() => {
    setTab(tab);
  });
}

```

Here, you tell React that setting tab to `'comments'` is not an urgent update, but is a `transition` that may take some time. React will then keep the old UI in place and interactive, and will switch to showing `<Comments />` when it is ready. See [Transitions](#) for more info.

Error boundaries

84ad3308338e2bb819f4f24fa8e9dfeeffaa970b

במקרה והמודול לא נטען בהצלחה (בגלל תקלת ברשות לדוגמא) תתקבל שגיאה. תוכלו לטפל בשגיאות כאלה באמצעות **גבולה** **שגיאה** כדי לספק חוות משתמש טובה יותר. ניתן להגיד ולהשתמש בגבול שגיאה בכל מקום מעל הקומפוננטה העצלה כדי להציג מצב שגיאה בזמן תקלת ברשות.

```
import React, { Suspense } from 'react';
import MyErrorBoundary from './MyErrorBoundary';

const OtherComponent = React.lazy(() => import('./OtherComponent'));
const AnotherComponent = React.lazy(() => import('./AnotherComponent'));

const MyComponent = () => (
  <div>
    <MyErrorBoundary>
      <Suspense fallback={<div>Loading...</div>}>
        <section>
          <OtherComponent />
          <AnotherComponent />
        </section>
      </Suspense>
    </MyErrorBoundary>
  </div>
);
```

פיזול קוד לפי route

ההחלטה איפה לפצל את הקוד יכולה להיות קצר בעיתית. תרצו לוודא מצד אחד שהబאנדלים יפוצלו באופן אחיד אבל גם שלא תפגע בחווית המשתמש.

רוב האנשים ברשות רגילים שמעבר בין דפים לוקחים זמן טינה מסוים. בנוסף, הדף כולל routes- מקום טוב להתחילה בו הוא המורנדר מחדש בביטחון, וכך המשמשים לא יהיו באמצעות אינטראקציה עם אלמנטים בדף בזמן הטעינה.

<<<<<< HEAD עם **React Router** עם **React.lazy**.
===== Here's an example of how to setup route-based code splitting into your app using libraries like **React Router** with **React.lazy**.

84ad3308338e2bb819f4f24fa8e9dfeeffaa970b

```
import React, { Suspense, lazy } from 'react';
import { BrowserRouter as Router, Routes, Route } from 'react-router-dom';

const Home = lazy(() => import('./routes/Home'));
const About = lazy(() => import('./routes/About'));

const App = () => (
  <Router>
    <Suspense fallback={<div>Loading...</div>}>
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/about" element={<About />} />
      </Routes>
    </Suspense>
  </Router>
);
```

ויצוא בשם

תומך רק בייצוא ברירת מחדל. אם המודול שרציתם להשתמש בו משתמש בייצוא בשם, תוכלו `React.lazy` נכון לעכשו ימшиיך לעבוד וSKUומפוננטות (tree shaking) ליצור מודולBINים שמייצוא את המודול כברירת מחדל. זה מודיא שניסיונות מיובאות רק כשהן בשימוש.

```
// ManyComponents.js
export const MyComponent = /* ... */;
export const MyUnusedComponent = /* ... */;
```

```
// MyComponent.js
export { MyComponent as default } from "./ManyComponents.js";
```

```
// MyApp.js
import React, { lazy } from 'react';
const MyComponent = lazy(() => import("./MyComponent.js"));
```

[Go to TOC](#)

Codebase Overview

This section will give you an overview of the React codebase organization, its conventions, and the implementation.

If you want to [contribute to React](#) we hope that this guide will help you feel more comfortable making changes.

We don't necessarily recommend any of these conventions in React apps. Many of them exist for historical reasons and might change with time.

Top-Level Folders

After cloning the [React repository](#), you will see a few top-level folders in it:

- `packages` contains metadata (such as `package.json`) and the source code (`src` subdirectory) for all packages in the React repository. **If your change is related to the code, the `src` subdirectory of each package is where you'll spend most of your time.**
- `fixtures` contains a few small React test applications for contributors.
- `build` is the build output of React. It is not in the repository but it will appear in your React clone after you [build it](#) for the first time.

The documentation is hosted [in a separate repository from React](#).

There are a few other top-level folders but they are mostly used for the tooling and you likely won't ever encounter them when contributing.

Colocated Tests

We don't have a top-level directory for unit tests. Instead, we put them into a directory called `__tests__` relative to the files that they test.

For example, a test for `setInnerHTML.js` is located in `__tests__/setInnerHTML-test.js` right next to it.

Warnings and Invariants

The React codebase uses `console.error` to display warnings:

```
if (__DEV__) {
  console.error('Something is wrong.');
}
```

Warnings are only enabled in development. In production, they are completely stripped out. If you need to forbid some code path from executing, use `invariant` module instead:

```
var invariant = require('invariant');

invariant(
```

```
2 + 2 === 4,
'You shall not pass!'
);
```

The invariant is thrown when the `invariant` condition is `false`.

"Invariant" is just a way of saying "this condition always holds true". You can think about it as making an assertion.

It is important to keep development and production behavior similar, so `invariant` throws both in development and in production. The error messages are automatically replaced with error codes in production to avoid negatively affecting the byte size.

Development and Production

You can use `__DEV__` pseudo-global variable in the codebase to guard development-only blocks of code.

It is inlined during the compile step, and turns into `process.env.NODE_ENV !== 'production'` checks in the CommonJS builds.

For standalone builds, it becomes `true` in the unminified build, and gets completely stripped out with the `if` blocks it guards in the minified build.

```
if (__DEV__) {
  // This code will only run in development.
}
```

Flow

We recently started introducing [Flow](#) checks to the codebase. Files marked with the `@flow` annotation in the license header comment are being typechecked.

We accept pull requests [adding Flow annotations to existing code](#). Flow annotations look like this:

```
ReactRef.detachRefs = function(
  instance: ReactInstance,
  element: ReactElement | string | number | null | false,
): void {
  // ...
}
```

When possible, new code should use Flow annotations. You can run `yarn flow` locally to check your code with Flow.

Multiple Packages

React is a [monorepo](#). Its repository contains multiple separate packages so that their changes can be coordinated together, and issues live in one place.

React Core

The "core" of React includes all the top-level `React APIs`, for example:

- `React.createElement()`
- `React.Component`
- `React.Children`

React core only includes the APIs necessary to define components. It does not include the [reconciliation](#) algorithm or any platform-specific code. It is used both by React DOM and React Native components.

The code for React core is located in `packages/react` in the source tree. It is available on npm as the `react` package. The corresponding standalone browser build is called `react.js`, and it exports a global called `React`.

Renderers

React was originally created for the DOM but it was later adapted to also support native platforms with [React Native](#). This introduced the concept of "renderers" to React internals.

Renderers manage how a React tree turns into the underlying platform calls.

Renderers are also located in `packages/`:

- [React DOM Renderer](#) renders React components to the DOM. It implements [top-level ReactDOM APIs](#) and is available as `react-dom` npm package. It can also be used as standalone browser bundle called `react-dom.js` that exports a `ReactDOM` global.
- [React Native Renderer](#) renders React components to native views. It is used internally by React Native.
- [React Test Renderer](#) renders React components to JSON trees. It is used by the [Snapshot Testing](#) feature of [Jest](#) and is available as `react-test-renderer` npm package.

The only other officially supported renderer is `react-art`. It used to be in a separate [GitHub repository](#) but we moved it into the main source tree for now.

Note:

Technically the `react-native-renderer` is a very thin layer that teaches React to interact with React Native implementation. The real platform-specific code managing the native views lives in the [React Native repository](#) together with its components.

Reconcilers

Even vastly different renderers like React DOM and React Native need to share a lot of logic. In particular, the [reconciliation](#) algorithm should be as similar as possible so that declarative rendering, custom components, state, lifecycle methods, and refs work consistently across platforms.

To solve this, different renderers share some code between them. We call this part of React a "reconciler". When an update such as `setState()` is scheduled, the reconciler calls `render()` on components in the tree and mounts, updates, or unmounts them.

Reconcilers are not packaged separately because they currently have no public API. Instead, they are exclusively used by renderers such as React DOM and React Native.

Stack Reconciler

The "stack" reconciler is the implementation powering React 15 and earlier. We have since stopped using it, but it is documented in detail in the [next section](#).

Fiber Reconciler

The "fiber" reconciler is a new effort aiming to resolve the problems inherent in the stack reconciler and fix a few long-standing issues. It has been the default reconciler since React 16.

Its main goals are:

- Ability to split interruptible work in chunks.
- Ability to prioritize, rebase and reuse work in progress.
- Ability to yield back and forth between parents and children to support layout in React.
- Ability to return multiple elements from `render()`.
- Better support for error boundaries.

You can read more about React Fiber Architecture [here](#) and [here](#). While it has shipped with React 16, the async features are not enabled by default yet.

Its source code is located in `packages/react-reconciler`.

Event System

React implements a layer over native events to smooth out cross-browser differences. Its source code is located in `packages/react-dom/src/events`.

What Next?

Read the [next section](#) to learn about the pre-React 16 implementation of reconciler in more detail. We haven't documented the internals of the new reconciler yet.

Props-קומפוננטות 1

קומפוננטות מאפשרות לך לשלב את משקל המשמש לחיצות עצמאיות שימוש חזרה, ולהשוו על כל חתיכה של קומפוננטות [כאן API](#)-בנפרד. דף זה מספק מבוא לרעיון של קומפוננטות. תוכלו למצוא [הסבר מפורט של ה](#).

וחזירות אלמנטים ("props" נקרא) הן מקובלות כמעט שירוטי. קומפוננטות הן כמו פונקציות המתארים מה אמרור להופיע על המ██ך של React.

קומפוננטות מסוגי פונקציות ומחלקות

הדרך פשוטה ביותר להגדיר קומפוננטה היא לכתוב פונקציה JavaScript:

```
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}
```

שימוש של המילה "props" חוקית משום שהיא מקבלת ארגומנט אובייקט נתונים ייחיד מסוג React פונקציה זו היא קומפוננטת אנו מכנים קומפוננטות כ אלה "קומפוננטת פונקציה" ממשם שם פשטן ומחזירה אלמנט (באנגלית properties, תכונות) פונקציות JavaScript.

כדי להגדיר קומפוננטה [ES6](#) תוכל גם להשתמש [במחלקה של](#):

```
class Welcome extends React.Component {
  render() {
    return <h1>Hello, {this.props.name}</h1>;
  }
}
```

שתי הקומפוננטות מעלה הן זהות מנקודת המבט של React.

לקומפוננטות פונקציה ומחלקות יש כמה פיצרים נוספים שנדרש עליהם [בקטעים הבאים](#).

רינדור של קומפוננטה

שמייצגים תגי React עד כה, נתקלנו רק ברכיבי DOM:

```
const element = <div />;
```

עם זאת, אלמנטים יכולים גם לייצג קומפוננטות המוגדרות על ידי המשתמש

```
const element = <Welcome name="Sara" />;
```

ואת JSX-רוואה אלמנט המייצג קומפוננטה שהודגשה על ידי המשתמש, היא מעבירה את המאפיינים שהוגדרו ב React כאשר "ילדים" להילדים" קומפוננטה זו כאובייקט ייחיד. אנו קוראים לאובייקט זה "props".

על העמוד "Hello, Sara" לדוגמה, הקוד הזה מрендר

```

function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}

const root = ReactDOM.createRoot(document.getElementById('root'));
const element = <Welcome name="Sara" />;
root.render(element);

```

[העלאה נסoco את זה ב-codepen](#)

באוו נסכם מה קורה בדוגמה זו:

1. `ReactDOM.render()` עם האלמנט `<Welcome name="Sara" />`.
2. React-בBORO ה `Welcome` עם `{name: 'Sara'}` קוראת לkomponenntת props.
3. בBORO התוצאה שלה `<h1>Hello, Sara</h1>` שלנו מוחזרה אלמנט komponenntת.
4. React DOM מעדכן ביעילות את ה DOM `<h1>Hello, Sara</h1>`.

הערה: יש להתחיל שמות komponenntות עם אות גודלה.

של `div` מייצג Tag `</div>`, לדוגמה DOM מתייחסת לkomponenntות המתחילהות באותיות קטנות בתור Taggi HTML `<Welcome>` אך, HTML-scope-להיות ב מייצג komponenntה ומחיב את `</>`.

[העלאה נסoco את זה ב-open معמייק JSX](#) כדי ללמידה עד על הסיבות מאחוריו קונבנצייה זו, אנא קראו את

יצירת komponenntות

komponenntות יכולות להתיחס לkomponenntות אחרות בפלט שלהם. דבר זה מאפשר לנו להשתמש באותה הפשתת komponenntות כל אלה בדרך כלל באים לידי ביטוי, React עברו כל רמה של פירוט. לחצן, טופס, תיבת דו-שיח, מסך: באפליקציות komponenntות.

הרבה פעמים `App` שמורנדרת את `Welcome` למשל, אנו יכולים ליצור komponenntת:

```

function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}

function App() {
  return (
    <div>
      <Welcome name="Sara" />
      <Welcome name="Cahal" />
      <Welcome name="Edite" />
    </div>
  );
}

```

[העלאה נסoco את זה ב-codepen](#)

אחד בראש האפליקציה. עם זאת, אם תשלבו את `App` חדשות כוללות komponenntת React בדרך כלל, אפליקציות בודרגה להגעה לחלק העליון, `Button`, באפליקציה קיימת, תוכלו להתחיל מלמטה למעלה באמצעות komponenntה קטנה כגון `Button`. ביותר של היררכיית התצוגה.

חילוץ קומפוננטות

אל תפחדו לפצל קומפוננטות לקומפוננטות קטנות יותר.

הבא `Comment` למשל, הבינו בקומפוננטת:

```
function Comment(props) {
  return (
    <div className="Comment">
      <div className="UserInfo">
        <img className="Avatar"
          src={props.author.avatarUrl}
          alt={props.author.name}>
      />
      <div className="UserInfo-name">
        {props.author.name}
      </div>
    </div>
    <div className="Comment-text">
      {props.text}
    </div>
    <div className="Comment-date">
      {formatDate(props.date)}
    </div>
  </div>
);
}
```

[-נסו את זה ב-codepen](#)

ומתארת התגובה באתר אינטרנט של `Comment` בתוך (`author`, `text`, `date`) - ו (מחוזת) (`UserInfo`, `Avatar`) היא מקבלת את מדיה חברתית.

קומפוננטה זו יכולה להיות מסובכת לשינוי בגלל כל הקינון שבה, ובנוסף קשה לעשות שימוש חואר בחלקים אינדיידואלים שלה. בואו נחלץ מספר קומפוננטות ממנה.

ראשית, נחלץ את `Avatar`:

```
function Avatar(props) {
  return (
    <img className="Avatar"
      src={props.user.avatarUrl}
      alt={props.user.name}>
  );
}
```

שלו שם גנרי יותר `Avatar`-זו הסיבה שבגללה נתנו לו `Comment`. לא צריך לדעת שהוא מתרנדר בתוך `user` (`author` ולא (`UserInfo`)).

מנקודת המבט של הקומפוננטה עצמה ולא על סמך הקונטקסט שבו הם נמצאים `props`-אנו ממליצים על מתן שמות למשימוש.

מעט `Comment` אנו יכולים כתעת לפשט:

```
function Comment(props) {
  return (
    <div className="Comment">
      <div className="UserInfo">
        <Avatar user={props.author} />
        <div className="UserInfo-name">
          {props.author.name}
        </div>
      </div>
      <div className="Comment-text">
        {props.text}
      </div>
      <div className="Comment-date">
        {formatDate(props.date)}
      </div>
    </div>
  );
}
```

לצד שם המשתמש `UserInfo` בשלב הבא, נחלץ את קומפוננטת `Avatar`:

```
function UserInfo(props) {
  return (
    <div className="UserInfo">
      <Avatar user={props.user} />
      <div className="UserInfo-name">
        {props.user.name}
      </div>
    </div>
  );
}
```

אפילו יותר זה מאפשר לנו לפשט את `Comment`:

```
function Comment(props) {
  return (
    <div className="Comment">
      <UserInfo user={props.author} />
      <div className="Comment-text">
        {props.text}
      </div>
      <div className="Comment-date">
        {formatDate(props.date)}
      </div>
    </div>
  );
}
```

[codepen -נסה את זה ב](#)

חילוץ קומפוננטות أولי נראה כמו עבודה שחויה בהתחלה, אבל בעלות על מגוון קומפוננטות לשימוש חוויר משתלמת (`Button`) באפליקציות גדולות יותר. כלל אכבע טוב הוא שם חלק ממשך המשמש שלכם נמצא בשימוש מספר פעמים הוא מועמד טוב להיות (`App`, `FeedStory`, `Comment`) או שהוא מרכיב מספק בכוחות עצמו, (`Panel`, `Avatar`). מחולץ לקומפוננטה אחרת.

הם לקרוא בלבד Props

שלה. הבינו בפונקציה `props`-בין אם אתם מצהירים על קומפוננטה **בפונקציה או כמחלקה**, אסור לה לעולם לשנות את ה `sum` הבא:

```
function sum(a, b) {
  return a + b;
}
```

פונקציות אלה נקראות **"טהורות"** מכיוון שהן לא מנסות לשנות את הקלטים שלהן, ותמיד מחזירות את אותה התוצאה עבור אותם קלטים.

לעומת זאת, פונקציה זו אינה טהורה משום שהיא משנה את הקלט שלה:

```
function withdraw(account, amount) {
  account.total -= amount;
}
```

היא די גמישה אבל יש לה כלל אחד נוקשה:

שלהן `props`-חיבות לפעול כמו פונקציות טהורות ביחס ל React כל קומפוננטות.

"state" כibold, ממשקי המשתמש של אפליקציות הם דינמיים ומשתנים עם הזמן. **בחילק הבא**, נציג את הקונספט החדש של לשנות את הפלט שלהן לאורך הזמן בתגובה לפעולות משתמש, תשובות מהרשות, React מאפשר לקומפוננטות state-ה. (מצב) וכל דבר אחר, מבלי להפר כלל זה.

הכלה לעומת הורשה

היא בעלת מודל הכלה רב עצמה ואני ממליצים להשתמש בהכלה במקום הורשה על מנת לבצע שימוש חוזר בקוד בין קומפוננטות.

חדשים בוחרים לעיתים קרובות להשתמש בהורשה ונציג איך ניתןReact בפרק זה עוסק במספר בעיות הקשורות כאשר מתכנתים לפתרו בעיות אלו באמצעות הכלה.

Containment

אשר Dialog או Sidebar ישן קומפוננטות אשר אין מכירות את ידיהן מראש. מצב זה נפוץ בעיקר בקומפוננטות כמו מציגות קופסאות גדריות.

על מנת להעביר אלמנטי ילדים ישירות כפלט שלהם children ממליצים שקומפוננטות אלו יעשו שימוש ב-

```
function FancyBorder(props) {
  return (
    <div className={'FancyBorder FancyBorder-' + props.color}>
      {props.children}
    </div>
  );
}
```

JSX-דבר זה מאפשר לkomponenotot אחרות להעביר אלמנטי ילדים שירוטים על ידי קינון ה

```
function WelcomeDialog() {
  return (
    <FancyBorder color="blue">
      <h1 className="Dialog-title">
        Welcome
      </h1>
      <p className="Dialog-message">
        Thank you for visiting our spacecraft!
      </p>
    </FancyBorder>
  );
}
```

[-נסחה זאת ב-CodePen](#)

מכיוון-children בשם prop-children מועבר לkomponenotah FancyBorder של JSX-כל דבר בתוך Tagית הальמנטים המועברים מופיעים בפלט הסופי, -בתוך אלמנט ה {props.children} מרנדר ש

לעתים באופן פחות נפוץ נדרש מספר "חורים" בkomponenotah. במקרים הללו אתה יכול לבחור בكونבנצייה שלך במקום children-להשתמש ב:

```
function SplitPane(props) {
  return (
    <div className="SplitPane">
      <div className="SplitPane-left">
        {props.left}
      </div>
```

```

        <div className="SplitPane-right">
          {props.right}
        </div>
      </div>
    );
}

function App() {
  return (
    <SplitPane
      left={
        <Contacts />
      }
      right={
        <Chat />
      } />
    );
}

```

[הכללה לעומת הורשה צאת ב-CodePen](#)

כמו כל מידע props-ם פשוט אובייקטיבים, אז אתה יכול להעביר אותם כ-`<Contacts />` ו-`<Chat />` כמוReact אלמנטי-ב-props-ב-בסיסות אחרות, אבל אין הגבלות לגבי מה שניתן להעביר כ"sotsaslots". גישה זאת יכולה להזכיר

Specialization

-לעתים אנו חושבים על קומפוננטות כ"מרקמים מיוחדים" של קומפוננטות אחרות. לדוגמה, אנחנו יכולים להגיד ש-`WelcomeDialog` הוא מקרה מיוחד של `Dialog`.

זה מושג גם על ידי הכללה, היכן שקומפוננטה "ספציפית" מרנדרת קומפוננטה "גנרטית" יותר ומקנפת אותה עם props:

```

function Dialog(props) {
  return (
    <FancyBorder color="blue">
      <h1 className="Dialog-title">
        {props.title}
      </h1>
      <p className="Dialog-message">
        {props.message}
      </p>
    </FancyBorder>
  );
}

function WelcomeDialog() {
  return (
    <Dialog
      title="Welcome"
      message="Thank you for visiting our spacecraft!" />
  );
}

```

[הכללה לעומת הורשה צאת ב-CodePen](#)

הכללה עבדת טוב בצורה שווה עבור קומפוננטות המוגדרות כמחלקות:

```

function Dialog(props) {
  return (
    <FancyBorder color="blue">
      <h1 className="Dialog-title">
        {props.title}
      </h1>
      <p className="Dialog-message">
        {props.message}
      </p>
      {props.children}
    </FancyBorder>
  );
}

class SignUpDialog extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.handleSignUp = this.handleSignUp.bind(this);
    this.state = {login: ''};
  }

  render() {
    return (
      <Dialog title="Mars Exploration Program"
        message="How should we refer to you?">
        <input value={this.state.login}
          onChange={this.handleChange} />
        <button onClick={this.handleSignUp}>
          Sign Me Up!
        </button>
      </Dialog>
    );
  }

  handleChange(e) {
    this.setState({login: e.target.value});
  }

  handleSignUp() {
    alert(`Welcome aboard, ${this.state.login}!`);
  }
}

```

[-נסזה זאת ב-CodePen](#)

אז מה לגבי הורשה?

באלפי קומפוננטות, ולא מצאנו שימושים כלשהם בהם אנו ממליצים על יצירת היררכיה React-בפייסבוק אנחנו משתמשים בהורשת קומפוננטות.

והכללה נותנים את כל הגישות שצריך על מנת להתאים נראות והתנהגות של קומפוננטה בצורה מפורשת ובוטחה יותר. Props או פונקציות שרירותיים, כולל ערכים פרימיטיביים, אלמנטי props זכור כי קומפוננטות יכולות לקבל

נפרד. אם ברצונך לבצע שימוש חזר בפונקציונליות, שאינה של ממש משתמש, אנו מציעים להוציא אותה למודול הקומפוננטות יכולות ליבא זאת ולהשתמש בפונקציה, באובייקט או במחלקה מבלי להרחיב אותם.

[Go to TOC](#)

רינדור מותנה

אתם יכולים ליצור קומפוננטות ייחדיות אשר מכמינות את ההתחנחות שאותם מחפשים. לאחר מכן, אתם יכולים, ב-React, של האפליקציה שלכם state-לרנדור רק חלק מהן, על פי תלות ב

כמו JavaScript המשמש באופרטורים של JavaScript. פועל באותו אופן שבו עובדים תנאים ב-React-רינדור מותנה ב **if** לעדכן את משק המשתמש כדי-React-הנוכחי, ותנו לו state-או **האופרטור המותנה** כדי ליצור אלמנטים המייצגים את ה-**שייהיה** תואם אליום.

הבינו בשתי הקומפוננטות הללו:

```
function UserGreeting(props) {
  return <h1>ברוך הבא!</h1>;
}

function GuestGreeting(props) {
  return <h1>אני הירשם.</h1>;
}
```

שמציגה אחת מהקומפוננטות הללו כתלות באם משתמש מחובר **Greeting** ניצור קומפוננטה:

```
function Greeting(props) {
  const isLoggedIn = props.isLoggedIn;
  if (isLoggedIn) {
    return <UserGreeting />;
  }
  return <GuestGreeting />;
}

const root = ReactDOM.createRoot(document.getElementById('root'));
// Try changing to isLoggedIn={true}:
root.render(<Greeting isLoggedIn={false} />);
```

[-נסו זאת ב-CodePen](#)

הבינו prop-דוגמה זו מונדרת ברכה שונה כתלות בערך של ה-**isLoggedIn**.

משתני אלמנט

אתם יכולים להשתמש במשתנים כדי לשמר אלמנטים. זה יכול לעזור לכם לרנדור חלק מהקומפוננטה באופן מותנה בעוד ששאר הפלט אינם משתנה.

הבינו בשתי הקומפוננטות החדשות המייצגות כפתורי התנטקות והתחברות:

```
function LoginButton(props) {
  return (
    <button onClick={props.onClick}>
      התחבר
    </button>
  );
}

function LogoutButton(props) {
```

```

return (
  <button onClick={props.onClick}>
    התחזק
  </button>
);
}

```

שנקראת בדוגמה הבאה, ניצור קומפוננטה התלויה ב `LoginControl`.

הווכחי שלו. בנוסף היא תרנnder state-כטולות ב `<LoginControl />` מהדוגמה הקודמת:

```

class LoginControl extends React.Component {
  constructor(props) {
    super(props);
    this.handleLoginClick = this.handleLoginClick.bind(this);
    this.handleLogoutClick = this.handleLogoutClick.bind(this);
    this.state = {isLoggedIn: false};
  }

  handleLoginClick() {
    this.setState({isLoggedIn: true});
  }

  handleLogoutClick() {
    this.setState({isLoggedIn: false});
  }

  render() {
    const isLoggedIn = this.state.isLoggedIn;
    let button;

    if (isLoggedIn) {
      button = <LogoutButton onClick={this.handleLogoutClick} />;
    } else {
      button = <LoginButton onClick={this.handleLoginClick} />;
    }

    return (
      <div>
        <Greeting isLoggedIn={isLoggedIn} />
        {button}
      </div>
    );
  }
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<LoginControl />);

```

ניצן צאת ב-CodePen

היא דרך מצוינת להתנות רינדר קומפוננטה, לפעמים ייתכן שתרצו להשתמש `if` בעוד שהכרצה על משתנה ושימוש בהוצאה שמוסברות בהמשך JSX-בתחביר קצר יותר. ישנן מספר דרכים להטמע תנאים ב-

עם אופרטור **&&** לוגי If הטמעת תנאי

זה JavaScript על ידי עטיפתם בסוגרים מסולסלים. זה כולל את האופרטור הלוגי `&&` של **JSON**-ביבולטים **להטמעה ביטויים** ב: יכול להיות שימושי עבור הוספה מותנית של אלמנט:

```
function Mailbox(props) {
  const unreadMessages = props.unreadMessages;
  return (
    <div>
      <h1>Hello!</h1>
      {unreadMessages.length > 0 &&
        <h2>
          הודעות שלא נקראו {unreadMessages.length} .
        </h2>
      }
    </div>
  );
}

const messages = ['React', 'Re: React', 'Re:Re: React'];

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Mailbox unreadMessages={messages} />);
```

[-נסו זאת ב-CodePen](#)

ב-JavaScript, `true && expression`-תמיד שווה ערך `expression`, `!false && expression`-תמיד שווה ערך `false`.

תתעלם ממנו ותציג עלייו `false`, `React` האלמנט מימין אחריו `&&` יופיע בפלט. אם הוא `true`, `!true` אם התנאי הוא.

שים לב שהחזרת ביטוי שלילי עדין תגרום לדילוג על האלמנט שאחריו `&&` אבל עדין תחזיר את הביטוי השלילי. בדוגמה להלן `<div>0</div>` יוחזר על ידי מתחמת הרינדרו.

```
render() {
  const count = 0;
  return (
    <div>
      {count && <h1>Messages: {count}</h1>}
    </div>
  );
}
```

עם אופרטור If-Else הטמעת

שיטה נוספת להטמעת רינדור אלמנטים מותנית היא לשמש באופרטור **ההתניה** של `condition ? true : false`.

בדוגמה הבאה, אנו משתמשים בו כדי לרינדר באופן מותנה בлок קטן של טקסט.

```
render() {
  const isLoggedIn = this.state.isLoggedIn;
  return (
    <div>
      מהובך {isLoggedIn ? 'ברוך' : 'לא'} !
    </div>
  );
}
```

```

        </div>
    );
}

```

זה יכול לשמש גם עבור ביטויים גדולים יותר למורדות שכך פחות ברור מה קורה:

```

render() {
  const isLoggedIn = this.state.isLoggedIn;
  return (
    <div>
      {isLoggedIn
        ? <LogoutButton onClick={this.handleLogoutClick} />
        : <LoginButton onClick={this.handleLoginClick} />
      }
    </div>
  );
}

```

זה באחריותכם לבחור סגנון מתאים על סמך מה שאתם והצווות שלכם מחשבים יותר קריין. כמו כן JavaScript-בדיקות כמו ב-Zero-Shot בודק כמו ב-[לחץ קומפוננטה](#).

מניעת רינדור של קומפוננטה

במקרים נדירים ייתכן שתרצו שקומפוננטה תשטייר את עצמה למורות שהיא רונדרה על ידי קומפוננטה אחרת. כדי לעשות זאת במקום את הפלט שלו הוא `null` החזרו:

`<WarningBanner />` שנקרא prop-מرونדר בהתאם לערך של ה `warn`. בדוגמה הבאה, ה-`warn` הוא `false`, אז הרכיב אינו מתרנnder:

```

function WarningBanner(props) {
  if (!props.warn) {
    return null;
  }

  return (
    <div className="warning">
      זה!
    </div>
  );
}

class Page extends React.Component {
  constructor(props) {
    super(props);
    this.state = {showWarning: true};
    this.handleToggleClick = this.handleToggleClick.bind(this);
  }

  handleToggleClick() {
    this.setState(state => ({
      showWarning: !state.showWarning
    }));
  }

  render() {
    return (
      <div>
        <WarningBanner warn={this.state.showWarning} />
      </div>
    );
  }
}

```

```
<button onClick={this.handleToggleClick}>
  {this.state.showWarning ? 'הצתר' : 'חצג'}
</button>
</div>
);
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Page />);
```

ב-CodePen נסוי צאת

של קומפוננטה לא משפיעה על הריצת מethods מחזור החיים של קומפוננטה. למשל `render` ממתהדת `null` החזרת `componentDidUpdate` עדיין יקרה.

קונטקסט

באופן יידי לכל קומפוננטה `props`-קונטקסט מספק דרך להעביר מידע דרך עץ הקומפוננטות בלי להשתמש ב-

אבל עבור מידע שנדרש, `props` טיפוסית, המידע מועבר למיטה (מרקם קומפוננטת אב לkomponenat ילד) דרך React באפליקציה יכול `props`-בהרבה קומפוננטות באפליקציה (כמו לדוגמה העדפות שפה או ערכת נושא של משק המשמש ב- להיות מסובבל). קונטקסט מספק דרך לשחרר מידע זהה בין קומפוננטות בלי להעביר אותו באופן מפורש לכל קומפוננטה

- [מתי להשתמש בקונטקסט](#)
- [לפני השימוש בקונטקסט](#)
- [משק תכנות](#)
 - [React.createContext](#)
 - [Context.Provider](#)
 - [Class.contextType](#)
 - [Context.Consumer](#)
 - [Context.displayName](#)
- [דוגמאות](#)
 - [קונטקסט דינמי](#)
 - [עדכן הקונטקסט מתוך קומפוננטה מקוונת](#)
 - [שימוש ביותר מkonstext אחד](#)
- [הסתיגיות](#)
- [משק תכנות מדור קדם](#)

מתי להשתמש בקונטקסט

הكونטקסט נועד לשחרר מידע שנחשב "גלאבלי" לכל הקומפוננטות בעץ, כמו מידע על המשתמש המאומת, ערכות הנושאות או של "ערכות הנושא" בשבייל לעצב את קומפוננטות הcpfotor `prop`-השפה המועדף. בקוד הנ"ל אנחנו מעבירים את ה-

```
embed:context/motivation-problem.js
```

דרך רכיביBINIMIM `drop`-בעזרת הקונטקסט, אפשר להמנע מההעביר את ה-

```
embed:context/motivation-solution.js
```

לפני השימוש בקונטקסט

השימוש בקונטקסט נועד בעיקר למצב שבו חלק מהמידע צריך להיות נגיש להרבה קומפוננטות בעומקים שונים. עדיף להשתמש בקונטקסט בחסכנות כי הוא יכול להקשות על שימוש חוזר בקומפוננטות.

להרבה קומפוננטות, **הכלת קומפוננטות היא `props` אם המטרה היחידה שלך בשימוש בקונטקסט היא להמנע מההעברה בדרך כלל פתרון פשוט יותר.**

לכמה רמות עומק, כדי שקומפוננטות ילד כמו `Page`-`props`-שמעבירה את ה `user` ו- `avatarSize` לדוגמא, קומפוננטת `Link` ו- `Avatar` יכולו להשתמש בהם:

```
<Page user={user} avatarSize={avatarSize} />
// ... שמאנדרת ...
<PageLayout user={user} avatarSize={avatarSize} />
// ... שמאנדרת ...
<NavigationBar user={user} avatarSize={avatarSize} />
// ... שמאנדרת ...
<Link href={user.permalink}>
  <Avatar user={user} size={avatarSize} />
</Link>
```

דרך כל כך הרבה ורמות עמוק תרגיש מיותרו, בעייר כי בסוף רק `user` ו-`avatarSize`- יכול להיות שהעברת ה-`props`, שמייד מושך אליו `Avatar` - באמצעות משתמש בהם. זה גם מעצבן שככל פעם שkomponenat ה-`Avatar` - komponenat ה-`Avatar` או `user` או `avatarSize` להעביר אותו דרך כל רכיבי הבניינים.

עצמה כדי שkomponenat ה-`Avatar` - דרך אחת לפטור את הבעייה **לא שימוש בקונטקט** היא **להעביר את komponenat ה-`Avatar`** באמצעות `user`-props הבניינים לא יצטרכו לדעת על ה-

```
function Page(props) {
  const user = props.user;
  const userLink = (
    <Link href={user.permalink}>
      <Avatar user={user} size={props.avatarSize} />
    </Link>
  );
  return <PageLayout userLink={userLink} />;
}

// עכשו, יש לנו:
<Page user={user} avatarSize={avatarSize} />
// ... שמאנדרת ...
<PageLayout userLink={...} />
// ... שמאנדרת ...
<NavigationBar userLink={...} />
// ... שמאנדרת ...
{props.userLink}
```

שahn `props`-ועל ה-`Link` `Avatar` - צריכה לדעת על komponenat ההזה, רק הקומפוננטה העליונה. דרושות.

`props` - מבנית העיצוב הזה נקראת היפוך שליטה והוא מאפשר לכתוב קוד נקי יותר במקרים רבים, להפחית את מספר ה-`props` להעביר באפליקציה, ולהחזיר שליטה לkomponenat העליונה. היפוך שליטה זה, עם זאת, הוא לא תמיד הדרכ הנכונה בכל מצב: העברת קוד מסובך למעלה בעץ komponenat יגרום לkomponenat השורש להיות יותר מסובכת ויכריך את komponenat הילך להיות יותר מדי גמישות.

מרובות לילדים, **כמתועד ("slots")** אין הגבלה של ילך ייחיד לכל komponenat. אפשר להעביר ילדים מרובים, ואפילו "משבצות" **כאן**:

```
function Page(props) {
  const user = props.user;
  const content = <Feed user={user} />;
  const topBar = (
    <NavigationBar>
      <Link href={user.permalink}>
        <Avatar user={user} size={props.avatarSize} />
      </Link>
    </NavigationBar>
```

```

);
return (
  <PageLayout
    topBar={topBar}
    content={content}
  />
);
}

```

tabniah היעזר ממספרם ובין כשרוצים להפריד קומפוננטתILD מקומפוננטת האב שלו. אפשר להרחיב את במרקירים שקומפוננטת הילד צריכה לתקשר עם קומפוננטת האב לפני הרינדור `render props` הפתרון עד יותר עם

למרות זאת, לעיתים אותו המידע צריך להיות נגיש ע"י מספר קומפוננטות בעז, ובעמקרים שונים. במרקירים כאלה, הקונטקסט מאפשר "לשדר" את המידע, ושינויים במידע, לכל הקומפוננטות בעז. דוגמאות שכיחות שבן שימוש בקונטקסט פשוט יותר מהאלטרנטיבות הנו כמו תואר קודם - ערכות נושא, העדפות שפה או זכרון מטעמו.

ממשק תכונות

`React.createContext` {#reactcreatecontext}

```
const MyContext = React.createContext(defaultValue);
```

מレンדר את הקומפוננטות שמאזינות לكونטקסט, הוא קורא את ערך הקונטקסט React - הקוד הזה יוצר אובייקט קונטקסט. כשל ברירת המחדל נקרא `value` כשהוא לא מעלייה מעלה בעז ספק תואם. ערך ברירת המחדל זה יכול להיות שימושי בבדיקות

ערך הספק לא יגרום `undefined` אוטומטיות לkomפוננטות מבודדות - בלי צורך לעטוף אותן. הערה: העברת הערך לkomפוננטות תחתיו נדרש את ערך ברירת המחדל.

`Context.Provider` {#contextprovider}

```
<MyContext.Provider value={/* some value */}>
```

שנותנת לkomפוננטות שצורךו אותו להקליב לשינויים בكونטקסט. (Provider) כל עצם קונטקסט מגיש עם komפוננטת ספק שיעבור לkomפוננטותILD שצורךו את הספק בכל רמות העומק של העז. ספק אחד יכול להתחבר `value` prop הספק מקבל לצרכים ובים. אפשר להגדיר ספקים ברמות שונות של אותו העץ כדי לעקוף את הערכים המוגדרים בהם בעומקים שונים של עץ הקומפוננטות.

אשר יועבר לkomפוננטות צאצאיות הדרישה מספק זה. ספק אחד יכול להיות מחובר `value` prop komפוננטת הספק מקבלת להרבה צרכנים. ספקים יכולים להיות מוקנים כדי לדודר ערכים עמוק יותר בתוך העץ.

כולל) ההפעלה מהספק לצאצאיו הצרכנים. `value`-כל הצרכנים שהם צאצאים של ספק, יונדרו בכל שינוי ב כך שהצרכן מתעדכן אפילו `useContext` אינה כפופה למתחה (`shouldComponentUpdate`-ו). בשקומפוננטה מעלייה מדגת על העדכון.

שינויים קבועים ע"י השוואת הערכים החדשניים מול הישנים בעזרתו האלגוריתם כמו `Object.is`.

הערה

הדרך שבה שינויים נקבעים יכולה ליצור בעיות כשמעברים עצמים כערכים: ראה [הסתיגיות](#).

Class.contextType {#classcontexttype}

```
class MyClass extends React.Component {
  componentDidMount() {
    let value = this.context;
    /* MyContext-mount בזמן ה- mount בשימוש בערך ה- */
  }
  componentDidUpdate() {
    let value = this.context;
    /* ... */
  }
  componentWillUnmount() {
    let value = this.context;
    /* ... */
  }
  render() {
    let value = this.context;
    /*MyContext- render משווה שmbos על ערך ה- */
  }
}
MyClass.contextType = MyContext;
```

שימוש `React.createContext()` במחלקה מוקצת בעצם קונטקסט שנוצר על ידי המתודה `contextType` -מאפיין האפשר להשתמש בהפניה זו `this.context`. במאפיין זה ניתן לנו לצורר את ערך הקונטקסט הנוכחי הקרוב ביותר בעזרת כל אחת מתודות מחזור החיים כולל מתודת הרינדור.

הערה:

אפשר לצורך רק קונטקסט אחד בעזרת משיק זה. על מנת לצורך יותר מאשר, ראו [שימוש ביותר מקונטקסט אחד](#).

אם אתם משתמשים בסינטקט הנשיוני של [מאפייני מחלקה ציבורית](#), תוכלם להשתמש במאפיין מחלקה `سطוי` על מנת לאתחל את ה `contextType`.

```
class MyClass extends React.Component {
  static contextType = MyContext;
  render() {
    let value = this.context;
    /* רינדור משווה בהתאם לערך */
  }
}
```

Context.Consumer {#contextconsumer}

```
<MyContext.Consumer>
  {value => /* רינדור משווה בהתאם לערך */}
</MyContext.Consumer>
```

שימוש בקומפוננטה זו מאפשר לצורך קונטקסט מתוך [קומפוננטת פונקציה](#) קומפוננטה.

ערך הארגומנט שמוסבר Dorshet פונציה בתו `ILD`. הפונציה הזאת מקבלת את ערך הקונטקסט הנוכחי ומחזירה צומת של ספק הקונטקסט הקרוב ביותר מעליינו בעץ. אם אין ספק לקונטקסט, הערך יהיה זהה prop `value` - לפונציה יהיה זהה ל`createContext()` לערך בירית המודול שנקבע בזמן ייצרת הקונטקסט).

הערה

למידע נוסף על תבנית "פונציה ילדים" בקרו בעמוד [render props](#).

`Context.displayName {#contextdisplayname}`

משתמש במחוזות זו על מנת לקבוע מה `displayName` מקבל מחרחות `React DevTools`. מסוג מחרחות אובייקט `-context`.

דוגמה-ב-DevTools: komponeneta-hava-tovifus-c

```
const MyContext = React.createContext(/* */);
MyContext.displayName = 'MyDisplayName';

<MyContext.Provider> // "MyDisplayName.Provider" ב-DevTools
<MyContext.Consumer> // "MyDisplayName.Consumer" ב-DevTools
```

דוגמאות

קונטקסט דינامي

דוגמה מורכבת יותר עם ערך דינامي של ערכת הנושא:

theme-context.js `embed:context/theme-detailed-theme-context.js`

themed-button.js `embed:context/theme-detailed-themed-button.js`

app.js `embed:context/theme-detailed-app.js`

עדכן הקונטקסט מתוך Komponeneta makonnet

לפעמים יש צורך לעדכן את הקונטקסט מתוך Komponeneta makonnetה שモגדרת עמוק בתוך עצ הקומפוננטות. במקרה זה, אפשר להזכיר פונציה דרך הקונטקסט כדי לתת לצרכני לעדכן אותו:

theme-context.js `embed:context/updating-nested-context-context.js`

theme-toggler-button.js `embed:context/updating-nested-context-theme-toggler-button.js`

app.js `embed:context/updating-nested-context-app.js`

שימוש ביותר Kontekst Achd

צורך להפוך את כל אחד מצרכני הקונטקסט לצומת נפרדת בעץ React, כדי לדא שרינדור הקונטקסט חדש יהיה מהיר.

`embed:context/multiple-contexts.js`

שתספק `drop` אם שני ערכי קונטקט (או יותר) בדרך כלל משומשים ביחד, יכול להיות שתרצו לשקלן יצירת קומפוננטת רינדורם יחד.

הסתיגויות

בגלל שקונטקט משתמש בזיהוי הפניה כדי להחליט מתי לעורר רינדור חדש, יש כל מיני מקרים קצרים שיכולים לגרום לרינדורם הרצכים בטעות, כשקומפוננטת האב של הספק מרדנרת את עצמה מחדש. לדוגמה, הקוד הנ"ל ירדנרט את כל הרצכים בכל שהוא עצם שנוצר מחדש כל פעם - פעם שהספק מרדנרט את עצמו מחדש, כיון ש

```
embed:context/reference-caveats-problem.js
```

של האב `state`-`child` לעקוב את הבעייה הזאת, אפשר להעביר את הערך ל

```
embed:context/reference-caveats-solution.js
```

ממתק תכנות מדור קודם

הערה

אבל אפליקציות, א.הכילה ממתק תכנות נסיוני לקונטקט. הממתק הישן ייתמך בכל גרסאות ה-16, בעבר עוד מידע React שימושה בו צרכות לעבור לשימוש בגרסת החדש. הממתק הישן יוסר בגרסה הראשית הבאה של על [ממתק הקונטקט הישן](#).

[Go to TOC](#)

חדרה React ייצרת אפליקציה

השתמש בסט של כלים משלבים על מנת לקבל את חווית המשתמש והمفہמת הטובה ביותר.

פופולריים שמקלים על שימושים כמו React עמדו זה מסביר על מספר סטם של כלים:

- גדילה לקבצים וקומפוננטות מורובות.
- סקח-שימוש בספריות צד שלישי מ-
- איתור טוויות נפוצות מוקדם.
- בפיתוח CSS ו-JavaScript עריכה בזמן אמיתי של-
- אופטימיזציה של המוצר פרודקסן.

הסטם של הכלים שאנו ממליצים עליהם בעמוד זה לא דרישים קונפיגורציה על מנת להתחילה.

ויתכן שלא תזדקק לשט כלים

עדין, שcool להוסיף את ריאקט JavaScript אם לא חוות את הביעות המתוארות לעיל או שאינך מרגיש בונה להשתמש בכלים או בלי `<script>` כתיגת `HTML` עם `JS` פשוטה בדף.

لتוך אתר קיים. תמיד קיימת האפשרות להוסיף סט כלים גדול יותר אם **React** זאת גם **הדרך הפשוטה ביותר לשלב את** תמצא זהה יכול לעזור!

סטם של כלים מומלצים

ממליץ על הפתרונות הבאיםReact הוצאות של:

- או יוצר אפליקציית **דף-יחיד** חדשה השתמש ב **React** אם אתה לומד.
- נסה את **server-rendered** עם **Node.js**, **Next.js**.
- אם אתה בונה אתר **STATIC** מבוסס **TOCO**, נסה את **Gatsby**.
- אתה בונה **ספרייה** **קומפוננטות** או **משלב עם קוד קיים**, נסה **כלים גמישים** יותר.

Create React App

يוצר سביבה נוחה למטרת **למידה של ריאקט**, והוא הדרך הטובה ביותר להתחיל לבנות **אפליקציה** **React**-**דף-יחיד** חדשה ב.

מספק חוות מפתח נוחה, הוא מארגן לך את סביבת העבודה כך שתוכל להשתמש בפיצרים החדשניים ביותר של בגרסה 5.6 **סקח-בגרסה** 14.0.0 ומעלה ו-node ועושה אופטימיזציה על האפליקציה שלך עבור פרודקסן. תצטרכ לתקן את: ומעלה. על מנת ליצור פרויקט, הרץ

```
npx create-react-app my-app
cd my-app
npm start
```

הערה

npm בשורה הראשונה זה לא שגיאת כתיב – זה כלי הרצת כבilities שmagיע עם **5.2+** apk

או במודול נתונים; הוא רכז יוצר מערכת לבניית פרויקט-אנדרואיד, שנitin להשתמש backend לא מטל בלוגיקתarrière. אבל אין צורך לידע עליהם [Babel](#)-[webpack](#), מאחריו הקלעים, הוא משתמש backend בה עם כל

ניתן. build -תיצור גרסה ייעילה של האפליקציה שלך בתיקיית ה `npm run build` כשאתה מוכן להעלות לפודקשו, הרצת Create React App-README מהו. שלו ומדריך המשתמש למדוד עוד על

Next.js

הו מכיל **פתרונות** React הוא פרימיום פופולרי וקל משקל לאפליקציות סטטיות ומרונדרות-שרות הנבנות עם [Next.js](#).
כסבירת השירות [Node.js](#) - עיצוב וניתוב ישר מהקובוסא, ומניה שאתה משתמש ב-

מהדריך הרשמי [\[ס. Next\]](#) למד על

Gatsby

אבל פולט React הוא נותן לך להשתמש בקומפוננטות React. הוא הדרך הכי טובה ליצור *אתרים שטויים עם HTML-CSS-1-CSS*. שرونדרו מראש על מנת להבהיר את זמני טעינה המהירים ביותר.

מהמדריך הרשמי ומגלאת ערכות התחלה Gatsby למד על

עוד סטימ של כלים גמישים

טט הכללים הבאים מציע יותר גמישות ובחירה. אנו ממליצים עליהם למשתמשים מתקדמים יותר

- **Neutrino** preset לאפליקציות webpack ומכל preseets הפשוטות של **React** משלב את הכוח של React.
 - ועד Nx, פול-סטאך, עם תמיכה מובנית לפרויקט monorepo הוא ערכת כלים לפיתוח Next.js, Express.
 - **Parcel** שעבוד **React** אפליקציית ווב מהירה, נטולת קונפיגורציות שימושת כ parcel-bundler.
 - **Razzle**-הוא פריאוורק שמשמש לרינדור צד-שרת ולא דרוש שום קונפיגורציה, אך מציע יותר גמישות מ Next.js.

יצירת שט כלים מאפס

:מורכב מ JavaScript בדרכן כלל שט כל

- זה מאפשר לך לנצל אкосיסטם רחב של חבילות צד-שלישי, ולהתקין או לעדכן אותן [npm](#) או [Yarn](#) או [מנהל חבילות](#), כמו בклות.
 - זה מאפשר לך לכתוב קוד מודולרי ולקבץ אותו לחבילות קטנות על מנת לשפר זמן [Parcel](#). או [webpack](#) כמו [Parcel](#). טעינה.
 - מודרני שעדיין עובד בדפדפנים ישנים זה מאפשר לך לכתוב קוד [Babel](#). כמו [Babel](#).

אם אתה מעדיף לתוכנן את סט הכלים שלך מאפס, [קרא את המדריך הזה](#) שיוצר מחדש כמה מהפונקציונליות של Create React App.

אל תשכח לודא שט הכלים שלך מוגדר באופן נכון לפירודקשיון.

[Go to TOC](#)

Cross-origin Errors

Note:

The following section applies only to the development mode of React. Error handling in production mode is done with regular try/catch statements.

In [development mode](#), React uses a global `error` event handler to preserve the "pause on exceptions" behavior of browser DevTools. It also logs errors to the developer console.

If an error is thrown from a [different origin](#) the browser will mask its details and React will not be able to log the original error message. This is a security precaution taken by browsers to avoid leaking sensitive information.

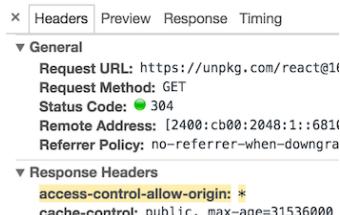
You can simplify the development/debugging process by ensuring that errors are thrown with a same-origin policy. Below are some common causes of cross-origin errors and ways to address them.

CDN

When loading React (or other libraries that might throw errors) from a CDN, add the `crossorigin` attribute to your `<script>` tags:

```
<script crossorigin src="..."></script>
```

Also ensure the CDN responds with the `Access-Control-Allow-Origin: *` HTTP header:



`Access-Control-Allow-Origin: *`

Webpack

Source maps

Some JavaScript bundlers may wrap the application code with `eval` statements in development. (For example Webpack will do this if `devtool` is set to any value containing the word "eval".) This may cause errors to be treated as cross-origin.

If you use Webpack, we recommend using the `cheap-module-source-map` setting in development to avoid this problem.

Code splitting

If your application is split into multiple bundles, these bundles may be loaded using JSONP. This may cause errors thrown in the code of these bundles to be treated as cross-origin.

To resolve this, use the `crossOriginLoading` setting in development to add the `crossorigin` attribute to the `<script>` tags generated for the JSONP requests.

[Go to TOC](#)

Design Principles

We wrote this document so that you have a better idea of how we decide what React does and what React doesn't do, and what our development philosophy is like. While we are excited to see community contributions, we are not likely to choose a path that violates one or more of these principles.

Note:

This document assumes a strong understanding of React. It describes the design principles of *React itself*, not React components or applications.

For an introduction to React, check out [Thinking in React](#) instead.

Composition

The key feature of React is composition of components. Components written by different people should work well together. It is important to us that you can add functionality to a component without causing rippling changes throughout the codebase.

For example, it should be possible to introduce some local state into a component without changing any of the components using it. Similarly, it should be possible to add some initialization and teardown code to any component when necessary.

There is nothing "bad" about using state or lifecycle methods in components. Like any powerful feature, they should be used in moderation, but we have no intention to remove them. On the contrary, we think they are integral parts of what makes React useful. We might enable [more functional patterns](#) in the future, but both local state and lifecycle methods will be a part of that model.

Components are often described as "just functions" but in our view they need to be more than that to be useful. In React, components describe any composable behavior, and this includes rendering, lifecycle, and state. Some external libraries like [Relay](#) augment components with other responsibilities such as describing data dependencies. It is possible that those ideas might make it back into React too in some form.

Common Abstraction

In general we [resist adding features](#) that can be implemented in userland. We don't want to bloat your apps with useless library code. However, there are exceptions to this.

For example, if React didn't provide support for local state or lifecycle methods, people would create custom abstractions for them. When there are multiple abstractions competing, React can't enforce or take advantage of the properties of either of them. It has to work with the lowest common denominator.

This is why sometimes we add features to React itself. If we notice that many components implement a certain feature in incompatible or inefficient ways, we might prefer to bake it into React. We don't do it lightly. When we do it, it's because we are confident that raising the abstraction level benefits the whole ecosystem. State, lifecycle methods, cross-browser event normalization are good examples of this.

We always discuss such improvement proposals with the community. You can find some of those discussions by the "[big picture](#)" label on the React issue tracker.

Escape Hatches

React is pragmatic. It is driven by the needs of the products written at Facebook. While it is influenced by some paradigms that are not yet fully mainstream such as functional programming, staying accessible to a wide range of developers with different skills and experience levels is an explicit goal of the project.

If we want to deprecate a pattern that we don't like, it is our responsibility to consider all existing use cases for it and [educate the community about the alternatives](#) before we deprecate it. If some pattern that is useful for building apps is hard to express in a declarative way, we will [provide an imperative API](#) for it. If we can't figure out a perfect API for something that we found necessary in many apps, we will [provide a temporary subpar working API](#) as long as it is possible to get rid of it later and it leaves the door open for future improvements.

Stability

We value API stability. At Facebook, we have more than 50 thousand components using React. Many other companies, including [Twitter](#) and [Airbnb](#), are also heavy users of React. This is why we are usually reluctant to change public APIs or behavior.

However we think stability in the sense of "nothing changes" is overrated. It quickly turns into stagnation. Instead, we prefer the stability in the sense of "It is heavily used in production, and when something changes, there is a clear (and preferably automated) migration path."

When we deprecate a pattern, we study its internal usage at Facebook and add deprecation warnings. They let us assess the impact of the change. Sometimes we back out if we see that it is too early, and we need to think more strategically about getting the codebases to the point where they are ready for this change.

If we are confident that the change is not too disruptive and the migration strategy is viable for all use cases, we release the deprecation warning to the open source community. We are closely in touch with many users of React outside of Facebook, and we monitor popular open source projects and guide them in fixing those deprecations.

Given the sheer size of the Facebook React codebase, successful internal migration is often a good indicator that other companies won't have problems either. Nevertheless sometimes people point out additional use cases we haven't thought of, and we add escape hatches for them or rethink our approach.

We don't deprecate anything without a good reason. We recognize that sometimes deprecations warnings cause frustration but we add them because deprecations clean up the road for the improvements and new features that we and many people in the community consider valuable.

For example, we added a [warning about unknown DOM props](#) in React 15.2.0. Many projects were affected by this. However fixing this warning is important so that we can introduce the support for [custom attributes](#) to React. There is a reason like this behind every deprecation that we add.

When we add a deprecation warning, we keep it for the rest of the current major version, and [change the behavior in the next major version](#). If there is a lot of repetitive manual work involved, we release a [codemod](#) script that automates most of the change. Codemods enable us to move forward without stagnation in a massive codebase, and we encourage you to use them as well.

You can find the codemods that we released in the [react-codemod](#) repository.

Interoperability

We place high value in interoperability with existing systems and gradual adoption. Facebook has a massive non-React codebase. Its website uses a mix of a server-side component system called XHP, internal UI libraries that came before React, and React itself. It is important to us that any product team can [start using React for a small feature](#) rather than rewrite their code to bet on it.

This is why React provides escape hatches to work with mutable models, and tries to work well together with other UI libraries. You can wrap an existing imperative UI into a declarative component, and vice versa. This is crucial for gradual adoption.

Scheduling

Even when your components are described as functions, when you use React you don't call them directly. Every component returns a [description of what needs to be rendered](#), and that description may include both user-written components like `<LikeButton>` and platform-specific components like `<div>`. It is up to React to "unroll" `<LikeButton>` at some point in the future and actually apply changes to the UI tree according to the render results of the components recursively.

This is a subtle distinction but a powerful one. Since you don't call that component function but let React call it, it means React has the power to delay calling it if necessary. In its current implementation React walks the tree recursively and calls render functions of the whole updated tree during a single tick. However in the future it might start [delaying some updates to avoid dropping frames](#).

This is a common theme in React design. Some popular libraries implement the "push" approach where computations are performed when the new data is available. React, however, sticks to the "pull" approach where computations can be delayed until necessary.

React is not a generic data processing library. It is a library for building user interfaces. We think that it is uniquely positioned in an app to know which computations are relevant right now and which are not.

If something is offscreen, we can delay any logic related to it. If data is arriving faster than the frame rate, we can coalesce and batch updates. We can prioritize work coming from user interactions (such as an animation caused by a button click) over less important background work (such as rendering new content just loaded from the network) to avoid dropping frames.

To be clear, we are not taking advantage of this right now. However the freedom to do something like this is why we prefer to have control over scheduling, and why `setState()` is asynchronous. Conceptually, we think of it as "scheduling an update".

The control over scheduling would be harder for us to gain if we let the user directly compose views with a "push" based paradigm common in some variations of [Functional Reactive Programming](#). We want to own the "glue" code.

It is a key goal for React that the amount of the user code that executes before yielding back into React is minimal. This ensures that React retains the capability to schedule and split work in chunks according to what it knows about the UI.

There is an internal joke in the team that React should have been called "Schedule" because React does not want to be fully "reactive".

Developer Experience

Providing a good developer experience is important to us.

For example, we maintain [React DevTools](#) which let you inspect the React component tree in Chrome and Firefox. We have heard that it brings a big productivity boost both to the Facebook engineers and to the community.

We also try to go an extra mile to provide helpful developer warnings. For example, React warns you in development if you nest tags in a way that the browser doesn't understand, or if you make a common typo in the API. Developer warnings and the related checks are the main reason why the development version of React is slower than the production version.

The usage patterns that we see internally at Facebook help us understand what the common mistakes are, and how to prevent them early. When we add new features, we try to anticipate the common mistakes and warn about them.

We are always looking out for ways to improve the developer experience. We love to hear your suggestions and accept your contributions to make it even better.

Debugging

When something goes wrong, it is important that you have breadcrumbs to trace the mistake to its source in the codebase. In React, props and state are those breadcrumbs.

If you see something wrong on the screen, you can open React DevTools, find the component responsible for rendering, and then see if the props and state are correct. If they are, you know that the problem is in the component's `render()` function, or some function that is called by `render()`. The problem is isolated.

If the state is wrong, you know that the problem is caused by one of the `setState()` calls in this file. This, too, is relatively simple to locate and fix because usually there are only a few `setState()` calls in a single file.

If the props are wrong, you can traverse the tree up in the inspector, looking for the component that first "poisoned the well" by passing bad props down.

This ability to trace any UI to the data that produced it in the form of current props and state is very important to React. It is an explicit design goal that state is not "trapped" in closures and combinators, and is available to React directly.

While the UI is dynamic, we believe that synchronous `render()` functions of props and state turn debugging from guesswork into a boring but finite procedure. We would like to preserve this constraint in React even though it makes some use cases, like complex animations, harder.

Configuration

We find global runtime configuration options to be problematic.

For example, it is occasionally requested that we implement a function like `React.configure(options)` or `React.register(component)`. However this poses multiple problems, and we are not aware of good solutions to them.

What if somebody calls such a function from a third-party component library? What if one React app embeds another React app, and their desired configurations are incompatible? How can a third-party component specify that it requires a particular configuration? We think that global configuration doesn't work well with composition. Since composition is central to React, we don't provide global configuration in code.

We do, however, provide some global configuration on the build level. For example, we provide separate development and production builds. We may also [add a profiling build](#) in the future, and we are open to considering other build flags.

Beyond the DOM

We see the value of React in the way it allows us to write components that have fewer bugs and compose together well. DOM is the original rendering target for React but [React Native](#) is just as important both to Facebook and the community.

Being renderer-agnostic is an important design constraint of React. It adds some overhead in the internal representations. On the other hand, any improvements to the core translate across platforms.

Having a single programming model lets us form engineering teams around products instead of platforms. So far the tradeoff has been worth it for us.

Implementation

We try to provide elegant APIs where possible. We are much less concerned with the implementation being elegant. The real world is far from perfect, and to a reasonable extent we prefer to put the ugly code into the library if it means the user does not have to write it. When we evaluate new code, we are looking for an implementation that is correct, performant and affords a good developer experience. Elegance is secondary.

We prefer boring code to clever code. Code is disposable and often changes. So it is important that it [doesn't introduce new internal abstractions unless absolutely necessary](#). Verbose code that is easy to move around, change and remove is preferred to elegant code that is prematurely abstracted and hard to change.

Optimized for Tooling

Some commonly used APIs have verbose names. For example, we use `componentDidMount()` instead of `didMount()` or `onMount()`. This is [intentional](#). The goal is to make the points of interaction with the library highly visible.

In a massive codebase like Facebook, being able to search for uses of specific APIs is very important. We value distinct verbose names, and especially for the features that should be used sparingly. For example, `dangerouslySetInnerHTML` is hard to miss in a code review.

Optimizing for search is also important because of our reliance on [codemods](#) to make breaking changes. We want it to be easy and safe to apply vast automated changes across the codebase, and unique verbose names help us achieve this. Similarly, distinctive names make it easy to write custom [lint rules](#) about using React without worrying about potential false positives.

[JSX](#) plays a similar role. While it is not required with React, we use it extensively at Facebook both for aesthetic and pragmatic reasons.

In our codebase, JSX provides an unambiguous hint to the tools that they are dealing with a React element tree. This makes it possible to add build-time optimizations such as [hoisting constant elements](#), safely lint and codemod internal component usage, and [include JSX source location](#) into the warnings.

Dogfooding

We try our best to address the problems raised by the community. However we are likely to prioritize the issues that people are *also* experiencing internally at Facebook. Perhaps counter-intuitively, we think this is the main reason why the community can bet on React.

Heavy internal usage gives us the confidence that React won't disappear tomorrow. React was created at Facebook to solve its problems. It brings tangible business value to the company and is used in many of its products. [Dogfooding](#) it means that our vision stays sharp and we have a focused direction going forward.

This doesn't mean that we ignore the issues raised by the community. For example, we added support for [web components](#) and [SVG](#) to React even though we don't rely on either of them internally. We are actively [listening to your pain points](#) and [address them](#) to the best of our ability. The community is what makes React special to us, and we are honored to contribute back.

After releasing many open source projects at Facebook, we have learned that trying to make everyone happy at the same time produced projects with poor focus that didn't grow well. Instead, we found that picking a small audience and focusing on making them happy brings a positive net effect. That's exactly what we did with React, and so far solving the problems encountered by Facebook product teams has translated well to the open source community.

The downside of this approach is that sometimes we fail to give enough focus to the things that Facebook teams don't have to deal with, such as the "getting started" experience. We are acutely aware of this, and we are thinking of how to improve in a way that would benefit everyone in the community without making the same mistakes we did with open source projects before.

גבולות שגיאה

וגרמו ל`פליטות שגיאות אניגמטיות` בוילו להשחתת המצב הפנימי של JavaScript-בעבר, שגיאות בתוך קומפוננטות בלא סיפק דרך לטפל בהם בוחן React ברינדור המסר הבא. מקור הבעיה תמיד נבע משגיאות קוד她们 האפליקציה, אבל בתוך הקומפוננטות, ולא מצא דרך להתחוש ממהן.

גבולות השגיאה

ב חלק ממשך השימוש לא אמורה לשבור את כל האפליקציה. כדי לפטור את הבעיה למשתמשי JavaScript שגיאת גרסה 16 מציגה קונספט חדש של "גבולות שגיאה".

שגורות בכל אחד מkomponenot הילך שלhn, מתעדות JavaScript גבולות שגיאה הם בעצם komponenot שתופסות שגיאות אותן ומציגות משך חולפי. במקום להציג את הקומפוננטה השבורה. הן תופסות שגיאות בזמן רינדור, במתודות מחזור חיים ובמבנה הקומפוננטות עברו כל אחת מkomponenot הילך שלhn.

הערה

:גבולות שגיאה לא תופסים שגיאות ב-

- מטפלים אירוטים (`מידע נוסף`)
- קוד אסינכרוני `setTimeout()` או `requestAnimationFrame()`
- רינדור בצד השירות
- שגיאות הקשורות בגבול השגיאה עצמו

קומפוננטת מחלקה הופכת לגבול שגיאה אם היא מגדרה לפחות אחת ממתודות מחזור החיים `static getDerivedStateFromError()` או `componentDidCatch()`. `static getDerivedStateFromError()` ווארת בティיעד השגיאה `componentDidCatch()` - משמשת לרינדור משך חולפי לאחר שגיאה שנתקפה, ו-

```
class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }

  static getDerivedStateFromError(error) {
    // מעדכנת את ה state כדי שהринדור הבא יציג משך חולפי
    return { hasError: true };
  }

  componentDidCatch(error, errorInfo) {
    // אפשר גם לתרעד את השגיאה לשירות לוגר //
    logErrorToMyService(error, errorInfo);
  }

  render() {
    if (this.state.hasError) {
      // מגדירים משך חולפי מותאם
      return <h1>Something went wrong.</h1>;
    }
  }
}
```

```

        return this.props.children;
    }
}

```

השימוש בגבולות שגיאה זהה לשימוש בכל קומפוננטה וגילו:

```

<ErrorBoundary>
  <MyWidget />
</ErrorBoundary>

```

אבל בתוך הקומפוננטה. רק קומפוננטות מחלקה JavaScript, ב-`catch` { } גבולות שגיאה עובדים בצורה דומה לבlok יכולות להיות גבולות שגיאה. בפועל, מגדירים בדרך כלל גבול שגיאה אחד ונשתמש בו בצורה איחודית בכל האפליקציה.

שימוש לב-**גבולות שגיאה תופסים אך ורק שגיאות בקומפוננטות הילד שלהם**, ולא בתוך עצם. אם מתרחשת שגיאה בקוד של גבולות השגיאה לדוגמא, השגיאות תעלתה לגבול שגיאה הבא מעלה, בדומה להתנהגות של בלוק `catch` { }-ב-JavaScript.

הדגמה חייה

שימוש לב-[דוגמה הבאה של הגדרה ושימוש בגבולות שגיאה עם גרסה 16 של React](#).

איפה למקם את גבולות שגיאה

ברמה העליונה על מנת להציג הודעה `route` ומתרטט של הגדרת גבולות שגיאה היא לשיקולכם. תוכלו לעטוף קומפוננטות "משהו השתבש" למשתמש, כמו הצורה שבה מערכות צד-שרת מטעלות בקריסות לעיתים קרובות. תוכלו גם לעטוף וידיגיטים אינדיבידואליים בגבולות שגיאה כדי להגן עליהם מהקורסית שאר האפליקציה.

התנהגות חדשה לשגיאות שלא נתפסו

מציה שינוי בהתנהגות עם השלכות חשובות. **שגיאות של נתפסות על ידי גבולות שגיאה יביאו לפירוק React** גרסה 16 של מוחלט של עץ קומפוננטות הראשי.

done בהחלטה זו לא מעט, אבל מנסיונו תמיד כדי שלא להשתמש בממשק מושחת, אלא להפרר ממנו לגמרי. לדוגמא, באפליקציה כמו מסנגר, להשאיר ממשק שבור בצורה גלויה לעין יכול להוביל לשליקות הודעה לאדם הלא נכון. באותו מידה, באפליקציה שמנהלת כספים עדיף לא להציג כלום מאשר להציג סכום שגוי.

השינוי הזה אומר שכשմשדרגים גרסה 16, בדרך כלל מגלים שגיאות באפליקציה שעדיין לא שמננו לב אליהם. הוספת גבולות שגיאה מאפשר לנו לספק חוות משתחש טובה יותר בכל מצב.

עוטף תוכן מהסרגל הצדדי, חלונית המידע, ושדה הקלט של ההודעה בגבולות Facebook (דוגמה, המסנגר) של שגיאה נפרדים. במקרה אחד מהם קורס, האחרים נשאים זמינים וrinteractive.

על מנת למצואו, (או לבנות שירותים דומים בעצמכם) אנחנו ממליצים גם לשימוש בשירותי דיווח השגיאות של בעיות בסביבת הייצור ולתקן אותן בקלות ובמהירות.

לקומפוננטות stack trace

כל השגיאות שקרוות בזמן הרינדור בסביבת הפיתוח מודפסות למסך בדף, אפילו אם האפליקציה React בגרסה 16 של הקומפוננטה. JavaScript-מודפס גם ה-`stack trace` של JavaUtil אתם בטועת. בכך נסחף להדעת השגיאה וה-

```
▶ React caught an error thrown by BuggyCounter. You should fix this error in your code. react-dom.development.js:7708
React will try to recreate this component tree from scratch using the error boundary you provided, ErrorBoundary.

Error: I crashed!

The error is located at:
in BuggyCounter (created by App)
in ErrorBoundary (created by App)
in div (created by App)
in App
```

זה עובד בברירת המחדל בפרויקטים - אפשר לראות את שם הקובץ ומספר השורה בקוד הקומפוננטה בעררת ה-[Create React](#):

```
▶ React caught an error thrown by BuggyCounter. You should fix this error in your code. react-dom.development.js:7708
React will try to recreate this component tree from scratch using the error boundary you provided, ErrorBoundary.

Error: I crashed!

The error is located at:
in BuggyCounter (at App.js:26)
in ErrorBoundary (at App.js:21)
in div (at App.js:8)
in App (at index.js:5)
```

Babel-תוכלו להשתמש [בתוספת זהה](#) - הוסיפו אותו לתצורת Create React, אם לא יצרתם את הפרויקט עם אפליקציה בפרויקט. שימו לב שהוא מיועד רק לשימוש בסביבת הפיתוח וחובה לנטרל אותו בסביבת הייצור.

הערה

אם אתם צריכים לתמוך `Function.name` שמות הקומפוננטות שמוצגים במעקב העריםות תלוי בשם שהוגדר במאפיין -תוכלו להוסיף את המאפיין `c`, (למשל IE 11 כמו) בדףנים או מכשירים ישנים יותר שלא תומכים זהה באופן סטנדרטי דרך נוספת היא לספק באופן ישיר את `.name` `function.name-polyfill`, כמו `polyfill` ב-`bundle` ב-`displayName` המאפיין. בכל קומפוננטה `displayName` המאפיין.

מה עם `try/catch`?

זה מצוין אבל עובד רק בקוד אימפרטיבי (קוד שמשנה את מצב האפליקציה) בлок `try/catch`:

```
try {
  showButton();
} catch (error) {
  // ...
}
```

הן דקלרטיביות ורך מצינות מה צריך לרנדר באפליקציה React לעומת זאת, קומפוננטות:

```
<Button />
```

ומספקים התנהגות דומה. לדוגמה, אפיו אם שגיאה צעה במתודה React גבולות שגיאה משמרים את האופן הדקלרטיבי של `componentDidUpdate` שנגרמה איפשהו עמוק בתוך העץ בתוקן `setState`. היא תוצף לגבול השגיאה הקרוב ביותר, `componentDidUpdate`.

מה עם מטפלים אירועים?

גבולות שגיאה לא תופסים שגיאות מתוך מטפל אירועים.

לא צריך גבולות שגיאה כדי להתחושש משלויות במטפל אירועים. בשונה מהמתודה הרינדורי ומתחות מחזור החיים, React עדין ידע מה להציג, שגיאות שצצות במטפל האירועים לא קוראות בזמן הרינדורי. אז אם צעה שגיאה

הilog של JavaScript: `try / catch` -ński צורך לתפוס שגיאה במטפל אירע, השתמשו בביטוי ה

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = { error: null };
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    try {
      // קוד שדורק שגיאה
    } catch (error) {
      this.setState({ error });
    }
  }

  render() {
    if (this.state.error) {
      return <h1>Caught an error.</h1>
    }
    return <button onClick={this.handleClick}>Click Me</button>
  }
}
```

סטנדרטי לטיפול בשגיאות ולא קשרה בשום אופן לגבולות שגיאות JavaScript שימושם לב שהדוגמא הנ"ל מדגימה קוד.

שינוי שם מגרסה 15

כלה תמייה מוגבלת ביותר לגבולות שגיאות תחת מתודה בשם אחר `unstable_handleError`. בקוד שלכם החל מגרסת הביתא הראשונה `componentDidCatch` - המתודה הזאת כבר לא נתמכת, ותאלצו לשנות אותה ל `React 16`.

כדי לעדכן באופן אוטומטי את הקוד הרלוונטי (codemod) בשביל השינוי זהה, כללונו [שינוי קוד](#).

[Go to TOC](#)

Error Decoder

In the minified production build of React, we avoid sending down full error messages in order to reduce the number of bytes sent over the wire.

We highly recommend using the development build locally when debugging your app since it tracks additional debug info and provides helpful warnings about potential problems in your apps, but if you encounter an exception while using the production build, this page will reassemble the original text of the error.

AJAX ו-APIs

איך אני יכול ליצור בקשה AJAX?

והמובנית בדף [Axios](#), [jQuery AJAX](#) כמה פופולריות הן שבחר עם AJAX אתה יכול להשתמש בכל ספרייה `window.fetch`.

איפה במחזור החיים של הקומponentה אני צריך ליצור בקשה AJAX?

-במחזור החיים. זה שביל שתוכל להשתמש ב `componentDidMount` במתודה AJAX אתה צריך לאכלס נתונים ובקשות `setState` כדי לעדכן את הקומponentה שלך מתי שהנתונים חווירים.

מוקמי state כדי להציג AJAX-דוגמה: שימוש בתוצאה מ

מוקמי בקומponentה state לאכלס `componentDidMount` - ב AJAX הקומponentה למיטה מדגימה איך ליצור בקשה

כזה JSON מחרירה אובייקט API-דוגמה:

```
{
  "items": [
    { "id": 1, "name": "Apples", "price": "$2" },
    { "id": 2, "name": "Peaches", "price": "$5" }
  ]
}
```

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      error: null,
      isLoading: false,
      items: []
    };
  }

  componentDidMount() {
    fetch("https://api.example.com/items")
      .then(res => res.json())
      .then(result => {
        this.setState({
          isLoading: true,
          items: result.items
        });
      },
      //שים לב: חשוב לטפל בשגיאות כאן
      //במקרה הבלוק-catch() כדי שלא נפספס
      //על מנת שלא "נבלע" חריגות מבאים בקומponentות שלנו.
      (error) => {
        this.setState({
          isLoading: true,
          error
        });
      }
    )
  }
}
```

```

}

render() {
  const { error, isLoading, items } = this.state;
  if (error) {
    return <div>Error: {error.message}</div>;
  } else if (!isLoading) {
    return <div>Loading...</div>;
  } else {
    return (
      <ul>
        {items.map(item => (
          <li key={item.id}>
            {item.name} {item.price}
          </li>
        ))}
      </ul>
    );
  }
}

```

Here is the equivalent with [Hooks](#):

```

function MyComponent() {
  const [error, setError] = useState(null);
  const [isLoading, setIsLoaded] = useState(false);
  const [items, setItems] = useState([]);

  // Note: the empty deps array [] means
  // this useEffect will run once
  // similar to componentDidMount()
  useEffect(() => {
    fetch("https://api.example.com/items")
      .then(res => res.json())
      .then(
        (result) => {
          setIsLoaded(true);
          setItems(result);
        },
        // Note: it's important to handle errors here
        // instead of a catch() block so that we don't swallow
        // exceptions from actual bugs in components.
        (error) => {
          setIsLoaded(true);
          setError(error);
        }
      )
    }, [])

  if (error) {
    return <div>Error: {error.message}</div>;
  } else if (!isLoading) {
    return <div>Loading...</div>;
  } else {
    return (
      <ul>
        {items.map(item => (
          <li key={item.id}>
            {item.name} {item.price}
          </li>
        ))}
      </ul>
    );
  }
}

```

```
)});  
  </ul>  
};  
}
```

ויצוּדי בְּנוּיה JSX, Babel

ויצוּדי בְּנוּיה JSX עם React?

כדי ללמידה עוד "JSX בלי React" לא! בדוק את

ויצוּדי בְּנוּיה ES6 (+) עם React?

כדי ללמידה עוד "React בלי ES6" לא! בדוק את

ויצוּדי בְּנוּיה JSX-איך אני יכול לכתוב הערוות ב

```
<div>
  {/* הערות כאן
  שולחן, {name}!
</div>
```

```
<div>
  זה עובד גם עבור
  {/* הערות מרובות שוררות.
  שולחן, {name}!
</div>
```

[Go to TOC](#)

העברת פונקציות לkomponenotot

לkomponenotah onClick כmo איך אני מעביר מטפל אירוע?

לkomponenotah ילדים props-העביר מטפל אירוע ופונקציות אחרות כ:

```
<button onClick={this.handleClick}>
```

לפונקציה למופע הקומפוננטה (ראה bind אם אתה צריך גישה לkomponenotah האב מתוך מטפל האירוע, אתה צריך גם לעשות למטה)

למופע של komponenotah bind איך אני עושה לפונקציה?

תלוי באיזה תחביר , יש כמה דרכים לוודא שלפונקציות יש גישה לתוכנות של komponenotah כmo . ושלבי בנייה אתה משתמש

bind (בתוך בניין ES2015)

```
class Foo extends Component {
  constructor(props) {
    super(props);
    this.handleClick = this.handleClick.bind(this);
  }
  handleClick() {
    console.log('הכפתור נלחץ');
  }
  render() {
    return <button onClick={this.handleClick}>לחץ על ליי</button>;
  }
}
```

תכונות מחלוקת (הצעה בשלב 3)

```
class Foo extends Component {
  // הערכה: תחביר זה ניסיוני ואינו działa מהתוך ביבנרים
  handleClick = () => {
    console.log('הכפתור נלחץ');
  }
  render() {
    return <button onClick={this.handleClick}>לחץ על ליי</button>;
  }
}
```

bind ב-render

```
class Foo extends Component {
  handleClick() {
    console.log('הכפתור נלחץ');
  }
  render() {
    return <button onClick={this.handleClick.bind(this)}>לחץ על ליי</button>;
  }
}
```

הערה:

يُוצר فونקציה جديدة بكل فעם שהكومponeנטה מרונדרת، דבר render בתוך `function.prototype.bind` שימוש ב-`this` יכול לגורם להשלכות על ביצועים (ראה למטה).

render-fonkציית ח' ב

```
class Foo extends Component {
  handleClick() {
    console.log('הכפתור נלחץ');
  }
  render() {
    return <button onClick={() => this.handleClick()}>לחץ על לי</button>;
  }
}
```

הערה:

يُוצר فونكציה جديدة بكل فעם שהكومponeנטה מרונדרת، דבר render שבו שימוש בfonkציית ח' בתוך `onClick` אופטימיזציות SMBOSSES על השוואות זהות נוקשיות.

האם זה בסדר להשתמש בfonkציות ח' בתוך מתחות render?

באופן כללי, כן, זה בסדר, ועתים קרובות זו הדרך הכי קלה להעביר פרמטרים לfonkציות callback.

אם אכן יש לך בעיות ביצועים, בהחלטה, בצע אופטימיזציה

בכל binding מדוע נחוץ לעשות?

:שני קטעי הקוד האלה אינן שווים רק JavaScript-

```
obj.method();
```

```
var method = obj.method;
method();
```

למתחות עוזר לוודא שקטע הקוד השני יעבד באותה צורה כמו הראשון binding לעשות.

רק למתחות שאתה מעביג לkomponeנטות אחרות. לדוגמה bind בדרך כלל עלייך לעשות עם `React`, `onClick={this.handleClick}`. לעומת זאת, אין צורך bind. bind כך שתרצה לעשות לה `this.handleClick` מעביר את המתחה או למתחות מעגל החיים: אנחנו מעבירים אותן לkomponeנטות אחרות `render` למתחות bind לעשות.

בצורה מפורטת, Yehuda Katz ואיך פוןקציות עובדות בbinding, מסביר מה זה [הפוסט הזה](#) של

?מדוע הפונקציה שלי מקבלת קרייה בכל פעם שהקומponeנטה מרונדרת?

וודא שאתה לא קורא לפוןקציה כאשר מעביר אותה לkomponeנטה

```
render() {  
  // טעות: קריואה לפונקציה במקום העברת שלה כרפרנס!  
  return <button onClick={this.handleClick()}>Click Me</button>  
}
```

במקום, עבר את הפקנץיה עצמה (ללא סוגרים)

```
render() {
  // בוכנו: הפקונקציה מועברת כרפרנס!
  return <button onClick={this.handleClick}>Click Me</button>
}
```

-איך אני מעביר פרמטר למטפל אירוע או לcallback?

אתה יכול להשתמש בפונקציית `ץ` כדי לעטוף את מטפל האירוע ולהעביר פרמטרים:

```
<button onClick={() => this.handleClick(id)} />
```

bind. זה שווה ערך לקריאה ל:

```
<button onClick={this.handleClick.bind(this, id)} />
```

דוגמא: העברת פרמטרים על ידי שימוש בפונקציות ח'ץ

```
const A = 65 // גַּתְתָּא-ASCII

class Alphabet extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      justClicked: null,
      letters: Array.from({length: 26}, (_, i) => String.fromCharCode(A + i))
    };
  }
  handleClick(letter) {
    this.setState({ justClicked: letter });
  }
  render() {
    return (
      <div>
        Just clicked: {this.state.justClicked}
        <ul>
          {this.state.letters.map(letter =>
            <li key={letter} onClick={() => this.handleClick(letter)}>
              {letter}
            </li>
          )}
        </ul>
      </div>
    )
  }
}
```

(**data-attributes**) דוגמה: העברת פרמטרים על ידי שימוש במאפייני-מידע

בשביל לאכון מידע שnochז למטפל אירופים. שכול גישה זו אם אתה צריך לעשות APIs DOM-לחילופין, אתה יכול להשתמש ב React.PureComponent.

```

const A = 65 // קוד תוו ASCII

class Alphabet extends React.Component {
  constructor(props) {
    super(props);
    this.handleClick = this.handleClick.bind(this);
    this.state = {
      justClicked: null,
      letters: Array.from({length: 26}, (_, i) => String.fromCharCode(A + i))
    };
  }

  handleClick(e) {
    this.setState({
      justClicked: e.target.dataset.letter
    });
  }

  render() {
    return (
      <div>
        Just clicked: {this.state.justClicked}
        <ul>
          {this.state.letters.map(letter =>
            <li key={letter} data-letter={letter} onClick={this.handleClick}>
              {letter}
            </li>
          )}
        </ul>
      </div>
    );
  }
}

```

איך אני יכול למנוע מפונקציה להיקרא יותר מדי מהר או יותר מדי פעמים ברצף?

להיקרא מהר מדי, אתה יכול להגביל callback-callback-ואתה רוצה למנוע מה `onClick` או `onScroll` אם יש לך מטפל אירוע כמו מוצר לפועל. זה יכול לעשות על ידי callback-את הקצב שבו ה-

- **throttling**: (למשל) דגום שינויים על בסיס תדירות: `_.throttle`
- **debouncing**: (למשל) פرسم שינויים לאחר פרק זמן של אי-פעילות: `_.debounce`
- **requestAnimationFrame throttling**: דגום שינויים על בסיס `requestAnimationFrame` (למשל `raf-sched`)

ראה את **היזואלייזציה הזאת** לשם השוואת בין פונקציות `throttle` ו-`debounce`.

הערה:

עליך מספקים מתחדש `callbacks` בצד `cancel`.
או לוודא מתוך הפונקציה המועכבת שהקומפוננטה עדין `componentWillUnmount` לקרוא למתחדשה הזאת מתוקן (`mounted`). מעוגנת.

Throttle

למפל אירוע לחיצה throttle מונע מפונקציה להיקרא יותר מפעם אחת בתוך חלון זמן נתון. הדוגמה למיטה עשויה כדי למנוע מקריאה לפונקציה יותר מפעם אחת בשנית.

```
import throttle from 'lodash.throttle';

class LoadMoreButton extends React.Component {
  constructor(props) {
    super(props);
    this.handleClick = this.handleClick.bind(this);
    this.handleClickThrottled = throttle(this.handleClick, 1000);
  }

  componentWillMount() {
    this.handleClickThrottled.cancel();
  }

  render() {
    return <button onClick={this.handleClickThrottled}>טען עוד</button>;
  }

  handleClick() {
    this.props.loadMore();
  }
}
```

Debounce

מודא שהפונקציה לא תוצאה לפועל עד שפרק זמן מסוים חלף מאז הפעם האחרון בה היא נקראת. זה יכול להיות שימושי כאשר אתה צריך לבצע חישוב יקר כלשהו בתגובה לאירוע שעלול להיות משוגר בתדירות גבוהה (למשל אירועים קלט טקסט עם השהייה של 250 מילישניות או גלילה או הקלה). הדוגמה למיטה עשויה

```
import debounce from 'lodash.debounce';

class Searchbox extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.emitChangeDebounced = debounce(this.emitChange, 250);
  }

  componentWillMount() {
    this.emitChangeDebounced.cancel();
  }

  render() {
    return (
      <input
        type="text"
        onChange={this.handleChange}
        placeholder="חיפוש..."
        defaultValue={this.props.value}
      />
    );
  }

  handleChange(e) {
    this.emitChangeDebounced(e.target.value);
  }
}
```

```

    }

    emitChange(value) {
      this.props.onChange(value);
    }
}

```

requestAnimationFrame throttling

היא דרך לקביעת תור להוצאה לפועל של פונקציה בתוך הדפדפן בזמן האופטימלי מבחינה תבצע בפריים הבא. הדפדפן יבודד קשה בכדי לוודא `requestAnimationFrame` ביצועי רינדור. פונקציה שהונסה לתור עם יחד עם זאת, אם הדפדפן לא מסוגל לעשות זאת, הוא יגביל באופן טבעי את כמות (60 fps) שישנים 60 פרייםים לשניה הפרייםים בשניה. לדוגמה, מכשיר מסוים עלול להיות מסוגל להתמודד רק עם 30 פרייםים לשניה וכך שתקבל רק 30 פרייםיםbial בשניה. היא טכנית שימושית בכך שהיא מונעת מחר throttling בפועל requestAnimationFrame -באותה שניה. שימוש לבצע יותר מ-60 עדכונים בשניה. אם אתה מבצע 100 עדכונים בשניה זה יוצר בעודה נוספת נספחת לדפדפן שהמשתמש בכל מקרה לא ייראה.

הערה:

שימוש בטכנית הזאת ילמד רק את הערך האחרון שפורסם בפריים. אתה יכול לראות דוגמה לכך האופטימיזציה הזאת MDN -עובדת ב

```

import rafSchedule from 'raf-schd';

class ScrollListener extends React.Component {
  constructor(props) {
    super(props);

    this.handleScroll = this.handleScroll.bind(this);

    // צור פונקציה חדשה לקביעת עדכונים.
    this.scheduleUpdate = rafSchedule(
      point => this.props.onScroll(point)
    );
  }

  handleScroll(e) {
    // כאשר אנחנו מקבלים אירוע גלילה, קבע ערך זה.
    // אם אנחנו מקבלים עדכונים רבים באותו זמן רק את הערך האחרון
    // this.scheduleUpdate({ x: e.clientX, y: e.clientY });
  }

  componentWillUnmount() {
    // בטל את כל העדכונים המתינו לפני שאנו נספחים (unmounted).
    this.scheduleUpdate.cancel();
  }

  render() {
    return (
      <div
        style={{ overflow: 'scroll' }}
        onScroll={this.handleScroll}
      >
        
      </div>
    );
  }
}

```

```
    );  
}
```

בדיקות הגבולות קצב

אתה `jest`-כasher אתה בוחק שקוד הגבולות הקצב עובד בצורה נכונה, היכולת להריץ זמן קדימה עוזרת. אם אתה משתמש ב`throttling`-בשביל להריץ קדימה זמן. אם אתה משתמש ב`mock timers`- יכול להשתמש ב`requestAnimationFrame raf-stub`-ייתכן שתמצא ש.

Virtual DOM and Internals

What is the Virtual DOM?

The virtual DOM (VDOM) is a programming concept where an ideal, or "virtual", representation of a UI is kept in memory and synced with the "real" DOM by a library such as ReactDOM. This process is called [reconciliation](#).

This approach enables the declarative API of React: You tell React what state you want the UI to be in, and it makes sure the DOM matches that state. This abstracts out the attribute manipulation, event handling, and manual DOM updating that you would otherwise have to use to build your app.

Since "virtual DOM" is more of a pattern than a specific technology, people sometimes say it to mean different things. In React world, the term "virtual DOM" is usually associated with [React elements](#) since they are the objects representing the user interface. React, however, also uses internal objects called "fibers" to hold additional information about the component tree. They may also be considered a part of "virtual DOM" implementation in React.

Is the Shadow DOM the same as the Virtual DOM?

No, they are different. The Shadow DOM is a browser technology designed primarily for scoping variables and CSS in web components. The virtual DOM is a concept implemented by libraries in JavaScript on top of browser APIs.

What is "React Fiber"?

Fiber is the new reconciliation engine in React 16. Its main goal is to enable incremental rendering of the virtual DOM. [Read more](#).

Component State

What does `setState` do?

`setState()` schedules an update to a component's `state` object. When state changes, the component responds by re-rendering.

What is the difference between `state` and `props`?

`props` (short for "properties") and `state` are both plain JavaScript objects. While both hold information that influences the output of render, they are different in one important way: `props` get passed *to* the component (similar to function parameters) whereas `state` is managed *within* the component (similar to variables declared within a function).

Here are some good resources for further reading on when to use `props` vs `state`:

- [Props vs State](#)
- [ReactJS: Props vs. State](#)

Why is `setState` giving me the wrong value?

In React, both `this.props` and `this.state` represent the *rendered* values, i.e. what's currently on the screen.

Calls to `setState` are asynchronous - don't rely on `this.state` to reflect the new value immediately after calling `setState`. Pass an updater function instead of an object if you need to compute values based on the current state (see below for details).

Example of code that will *not* behave as expected:

```

incrementCount() {
  // Note: this will *not* work as intended.
  this.setState({count: this.state.count + 1});
}

handleSomething() {
  // Let's say `this.state.count` starts at 0.
  this.incrementCount();
  this.incrementCount();
  this.incrementCount();
  // When React re-renders the component, `this.state.count` will be 1, but you
  // expected 3.

  // This is because `incrementCount()` function above reads from
  // `this.state.count`,
  // but React doesn't update `this.state.count` until the component is re-
  // rendered.
  // So `incrementCount()` ends up reading `this.state.count` as 0 every time, and
  // sets it to 1.

  // The fix is described below!
}

```

See below for how to fix this problem.

How do I update state with values that depend on the current state?

Pass a function instead of an object to `setState` to ensure the call always uses the most updated version of state (see below).

What is the difference between passing an object or a function in `setState`?

Passing an update function allows you to access the current state value inside the updater. Since `setState` calls are batched, this lets you chain updates and ensure they build on top of each other instead of conflicting:

```
incrementCount() {
  this.setState((state) => {
    // Important: read `state` instead of `this.state` when updating.
    return {count: state.count + 1}
  });
}

handleSomething() {
  // Let's say `this.state.count` starts at 0.
  this.incrementCount();
  this.incrementCount();
  this.incrementCount();

  // If you read `this.state.count` now, it would still be 0.
  // But when React re-renders the component, it will be 3.
}
```

[Learn more about `setState`](#)

When is `setState` asynchronous?

Currently, `setState` is asynchronous inside event handlers.

This ensures, for example, that if both `Parent` and `Child` call `setState` during a click event, `Child` isn't re-rendered twice. Instead, React "flushes" the state updates at the end of the browser event. This results in significant performance improvements in larger apps.

This is an implementation detail so avoid relying on it directly. In the future versions, React will batch updates by default in more cases.

Why doesn't React update `this.state` synchronously?

As explained in the previous section, React intentionally "waits" until all components call `setState()` in their event handlers before starting to re-render. This boosts performance by avoiding unnecessary re-renders.

However, you might still be wondering why React doesn't just update `this.state` immediately without re-rendering.

There are two main reasons:

- This would break the consistency between `props` and `state`, causing issues that are very hard to debug.
- This would make some of the new features we're working on impossible to implement.

This [GitHub comment](#) dives deep into the specific examples.

Should I use a state management library like Redux or MobX?

Maybe.

It's a good idea to get to know React first, before adding in additional libraries. You can build quite complex applications using only React.

מבנה קבצים

האם ישנה דרך מומלצת לבנות פרויקטי React?

אין דעה לדרכ שבה אתה מסדר קבצים בתיקיות. עם זאת ישנו כמה דרכים נפוצות שאולי תרצה לזכור בחשבון-ל-React.

קיובץ לפי פיצ'רים או routes

ובדיוקות ביחס בלבד ביחס לתיקיות לפי פיצ'רים או JS, CSS דרך נפוצה אחת בבניית פרויקטים היא לקובץ קבצי HEAD <<<<<< ===== One common way to structure projects is to locate CSS, JS, and tests together inside folders grouped by feature or route.

e548bd7e65086a8206fee46bd9e24b18b68bf045

```
common/
  Avatar.js
  Avatar.css
  APIUtils.js
  APIUtils.test.js
feed/
  index.js
  Feed.js
  Feed.css
  FeedStory.js
  FeedStory.test.js
  FeedAPI.js
profile/
  index.js
  Profile.js
  ProfileHeader.js
  ProfileHeader.css
  ProfileAPI.js
```

ההגדרה של "פיצ'ר" היא לא אוניברסלית, ובחרית רמת הפירוט תלויך. אם אתה לא יכול לחשב על רשיימה של תיקיות ברמה עליונה, אתה יכול לשאול את משתמשי המוצר שלך מה הם החלקים העיקריים מהם המוצר שלך מורכב, ולהשתמש בתשובה שלהם כבסיס לפרויקט.

קיבוץ לפי סוג קובץ

דרך פופולרית נוספת לבניית פרויקטים היא קיבוץ קבצים מסווג דומה יחדיו, לדוגמה:

```
api/
  APIUtils.js
  APIUtils.test.js
  ProfileAPI.js
  UserAPI.js
components/
  Avatar.js
  Avatar.css
  Feed.js
  Feed.css
  FeedStory.js
  FeedStory.test.js
  Profile.js
  ProfileHeader.js
  ProfileHeader.css
```

ישנם אנשים שמעדיפים ללכט רוחק יותר, ולהפריד קומפוננטות בתוך תיקיות על פי התפקיד שלהם בישום. לדוגמה [Atomic Design](#) מגדולה יותר שנבנתה על עיקרון זה. זכור שהעיתים יוטר פרודוקטיבי להתייחס למגדולות כלשהם. כדוגמאות שיכולות לעזור מאשר כחוקים שצורך לציתם להם.

הימנע מעדף קינון

בינהם, או `relative imports` זה נהיה קשה יותר לכתוב JavaScript. ישנו כמה בעיות הקשורות עם קינון עמוק בפרויקט הלו כשהקבצים זים. אלא אם יש לך סיבה מוצדקת להשתמש במבנה תיקיות עמוק, עדיף שייהי עמוק `imports` לעדכן את השל מקסימום שלוש או ארבע ומות קינון לתיקיות בפרויקט אחד. עם זאת, זה רק המלצה, והוא יכול להיות שהוא לא רלוונטי לפROYיקט שלך.

אין צורך להשקיע מחשבה מיותרת בנושא

אם אתה רק מתחילה פרויקט, [אל תבזבז יותר מחמש דקות](#) על בחירת מבנה הפרויקט. בחר אחת מהגישות שלמעלה (או שתחשוב על אחת משלך) ותתחיל לכתוב קוד! אתה ננראה תרצה לחשב על זה מחדש אחרי שתכתבו קוד אמיתי.

אם אתה מרגיש תקווע למגררי, תתחיל בהשahir את כל הקבצים בתיקייה אחת. בסופו של דבר היא תגדל מספיק עד שתרגיש את הצורך להפריד כמה קבצים מהתיקייה. בעת ההיא יהיה לך מספיק ידע על אילו קבצים אתה עורך בຕדיירות הכי גבוהה. [באופן כללי](#), זה רעיון טוב להשahir קבצים שימושיים בຕדיירות גבוהה קרובים אחד לשני. עיקרונו זה נקרא "קולוקציה".

כשפROYיקטים גדלים, ווב הזמן הם משלבים את שתי הגישות שהראנו קודם. מכאן נובע שבבחירה הגישה ה"נכונה" בתחילת הפרויקט היא לא חשובה במיוחד.

[Go to TOC](#)

CSS-עיצוב 1

לקומפוננטה שלי CSS classes איך אני מוסיף?

prop-העבר מחוותת `className`:

```
render() {
  return <span className="menu navigation-menu"></span>
}
```

של הקומפוננטה או props state- תלויים נפוץ CSS classes זה:

```
render() {
  let className = 'menu';
  if (this.props.isActive) {
    className += ' menu-active';
  }
  return <span className={className}></span>
}
```

טיפול

יכולת פשוט זאת `classnames` אם לעתים קרובות אתה מוצא את עצמו כותב קוד זהה, ספריות.

עיצוב inline האם אני יכול כתוב?

כו, ראה את התיעוד על עיצוב [כאן](#).

inline הוא רע?

עיצוב inline בדרך כלל טובים יותר לביצועים מאשר CSS classes.

CSS-JS מה זה?

במוקום להיות מוגדר בקובץ חיצוני JavaScript מתייחס לתבנית שבה "JS-ב-CSS".

אין דעה על איך עיצוב React-אך מספקת מספריות צד שלישי. ל-React-שים לב שהfonkczionialiot היא לא חלק מ- CSS-Chicniyi כרגע לפנות אליהם באמצעות `*.css`. מוגדר; אם יש ספק, נקודת התחליה טובה היא להגדיר את העיצוב של קובץ `className`.

האם אני יכול לעשות אнимציות ב-React?

להנפשת אнимציות. ראה [React Transition Group](#), [React Motion](#), [React Spring](#), או [Framer Motion](#) לדוגמא.

[Go to TOC](#)

Versioning Policy

React follows [semantic versioning \(semver\)](#) principles.

That means that with a version number **x.y.z**:

- When releasing **critical bug fixes**, we make a **patch release** by changing the **z** number (ex: 15.6.2 to 15.6.3).
- When releasing **new features** or **non-critical fixes**, we make a **minor release** by changing the **y** number (ex: 15.6.2 to 15.7.0).
- When releasing **breaking changes**, we make a **major release** by changing the **x** number (ex: 15.6.2 to 16.0.0).

Major releases can also contain new features, and any release can include bug fixes.

Minor releases are the most common type of release.

This versioning policy does not apply to prerelease builds in the Next or Experimental channels. [Learn more about prereleases](#).

Breaking Changes

Breaking changes are inconvenient for everyone, so we try to minimize the number of major releases – for example, React 15 was released in April 2016 and React 16 was released in September 2017, and React 17 was released in October 2020.

Instead, we release new features in minor versions. That means that minor releases are often more interesting and compelling than majors, despite their unassuming name.

Commitment to Stability

As we change React over time, we try to minimize the effort required to take advantage of new features. When possible, we'll keep an older API working, even if that means putting it in a separate package. For example, [mixins have been discouraged for years](#) but they're supported to this day [via create-react-class](#) and many codebases continue to use them in stable, legacy code.

Over a million developers use React, collectively maintaining millions of components. The Facebook codebase alone has over 50,000 React components. That means we need to make it as easy as possible to upgrade to new versions of React; if we make large changes without a migration path, people will be stuck on old versions. We test these upgrade paths on Facebook itself – if our team of less than 10 people can update 50,000+ components alone, we hope the upgrade will be manageable for anyone using React. In many cases, we write [automated scripts](#) to upgrade component syntax, which we then include in the open-source release for everyone to use.

Gradual Upgrades via Warnings

Development builds of React include many helpful warnings. Whenever possible, we add warnings in preparation for future breaking changes. That way, if your app has no warnings on the latest release, it will be compatible with the next major release. This allows you to upgrade your apps one component at a time.

Development warnings won't affect the runtime behavior of your app. That way, you can feel confident that your app will behave the same way between the development and production builds -- the only differences are that the production build won't log the warnings and that it is more efficient. (If you ever notice otherwise, please file an issue.)

What Counts as a Breaking Change?

In general, we *don't* bump the major version number for changes to:

- **Development warnings.** Since these don't affect production behavior, we may add new warnings or modify existing warnings in between major versions. In fact, this is what allows us to reliably warn about upcoming breaking changes.
- **APIs starting with `unstable_`.** These are provided as experimental features whose APIs we are not yet confident in. By releasing these with an `unstable_` prefix, we can iterate faster and get to a stable API sooner.
- **Alpha and canary versions of React.** We provide alpha versions of React as a way to test new features early, but we need the flexibility to make changes based on what we learn in the alpha period. If you use these versions, note that APIs may change before the stable release.
- **Undocumented APIs and internal data structures.** If you access internal property names like `__SECRET_INTERNALS_DO_NOT_USE_OR_YOU_WILL_BE_FIRED` or `__reactInternalInstance$uk43rzhitjg`, there is no warranty. You are on your own.

This policy is designed to be pragmatic: certainly, we don't want to cause headaches for you. If we bumped the major version for all of these changes, we would end up releasing more major versions and ultimately causing more versioning pain for the community. It would also mean that we can't make progress in improving React as fast as we'd like.

That said, if we expect that a change on this list will cause broad problems in the community, we will still do our best to provide a gradual migration path.

If a Minor Release Includes No New Features, Why Isn't It a Patch?

It's possible that a minor release will not include new features. [This is allowed by semver](#), which states "[a minor version] MAY be incremented if substantial new functionality or improvements are introduced within the private code. It MAY include patch level changes."

However, it does raise the question of why these releases aren't versioned as patches instead.

The answer is that any change to React (or other software) carries some risk of breaking in unexpected ways. Imagine a scenario where a patch release that fixes one bug accidentally introduces a different bug. This would not only be disruptive to developers, but also harm their confidence in future patch releases. It's especially regrettable if the original fix is for a bug that is rarely encountered in practice.

We have a pretty good track record for keeping React releases free of bugs, but patch releases have an even higher bar for reliability because most developers assume they can be adopted without adverse consequences.

For these reasons, we reserve patch releases only for the most critical bugs and security vulnerabilities.

If a release includes non-essential changes — such as internal refactors, changes to implementation details, performance improvements, or minor bugfixes — we will bump the minor version even when there are no new features.

טפסים

מכיוון שאלמנטים של טפסים, ב-React, מושתמשים כמעט מינימלית מאשר אלמנטים אחרים של HTML, מתקבלים מעט שונה מאחר אלמנטים אחרים של HTML. מתקבל שם אחד-HTML-פנימי. למשל, הטופס הבא ב state באפין טבוי שמורים על:

```
<form>
  <label>
    טבוי:
    <input type="text" name="name" />
  </label>
  <input type="submit" value="Submit" />
</form>
```

של מעבר לדף חדש כאשר המשתמש שולח את הטופס. אם אTEM HTML לטופס זה יש את התנהוגות ברירת המחדל של טפסי המנצל בשילוח JavaScript ככה זה עובד. אבל ברוב המקרים, זה נוח שיש פונקציית-React, רוצחים את התנהוגות זו ב-טופס ויש לה גישה לנוטונים שהמשתמש הכנס לטופס. הדרך הסטנדרטית להשיג זאת היא באמצעות טכניקה הנקראת ".komponenotot-mbokrot".

קומפוננטות מבוקרות

שומרים בדרך כלל על מצבם ומעודכנים אותו על `<input>`, `<textarea>`, ו- `<select>` אלמנטים של טופס כגון, HTML-ב-של קומפוננטות, ומעודכו רק עם state-בר-שינוי נשמר בדרך כלל במאפייני state-סמרק קלט המשתמש. ב-`setState()`.

לאחר (מקור אמת ייחיד) "single source of truth" להיות של React state-אנחנו יכולים לשלב את השניים על ידי הפיכת השמרנדרת טופס גם שולטות בהמה שקרה באותו טופס על קלט המשתמש הבא. קלט מאלמנט טופס מכון קומפוננטה "בדרכו זו נקרא" "קומפוננטה מבוקרת React" שערכו נשלט על ידי.

את השם כאשר הטופס נשלח, אנחנו יכולים לרשום log-לדוגמה, אם אנחנו רוצחים לשנות את הדוגמה הקודמת כך שתרשום את הטופס כקומפוננטה מבוקרת:

```
class NameForm extends React.Component {
  constructor(props) {
    super(props);
    this.state = {value: ''};

    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }

  handleChange(event) {
    this.setState({value: event.target.value});
  }

  handleSubmit(event) {
    alert('שם נשלח: ' + this.state.value);
    event.preventDefault();
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
```

```

        <label>
          פש:
          <input type="text" value={this.state.value} onChange={this.handleChange}>
        />
      </label>
      <input type="submit" value="פיש" />
    </form>
  );
}
}
}

```

ב-CodePen נסwo זאת

למרות שהוא אומר שיש צורך בכתיבה יותר קוד, עכשו ניתן עם קומפוננטה נשלטת, ערך הקלט תמיד מונע על ידי הothers, או לתחילת אותו מ-UI להעביר את הערך לאלמנטי.

ה-**taggit textarea**

מגדיר את הטקסט שלו על ידי הילדים שלו `<textarea>` אלמנט-ב-

```

<textarea>
  זה מעת טקסט ב-הו,
</textarea>

```

יכול להכתב באופן `-במקום`. בדרך זו, טופס המשתמש ב-`value` משתמש במאפיין `value` של שורה אחת דומה לטופס המשתמש ב-

```

class EssayForm extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      value: ''
    };
    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }
  handleChange(event) {
    this.setState({value: event.target.value});
  }
  handleSubmit(event) {
    alert('מאתך פיש: ' + this.state.value);
    event.preventDefault();
  }
  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          פש:
          <textarea value={this.state.value} onChange={this.handleChange}>
        </label>
        <input type="submit" value="פיש" />
      </form>
    );
  }
}

```

```

    );
}
}

```

מאותחל עם קצת טקסט בתוכו area-text במבנה, כך שה `this.state.value` לב ש

תגית select

הבא מיצר רשימה נפתחת של טעמים HTML-מייצר רשימה נפתחת. למשל, ה-

```

<select>
  <option value="grapefruit">אשכולית</option>
  <option value="lime">לימון</option>
  <option selected value="coconut">קוקוס</option>
  <option value="mango">מנゴ</option>
</select>

```

במקום להשתמש במאפיין `selected` שימו לב שהאפשרות קוקוס נבחרה תחילה, בגל השימוש במאפיין `selected` של תגית השורש `value` משתמשת במאפיין `value` של `select`. זה נכון בקומפוננטה מבוקרת מכיוון שאתם צריים. לעדכן אותו רק במקומות אחד. לדוגמה:

```

class FlavorForm extends React.Component {
  constructor(props) {
    super(props);
    this.state = {value: 'coconut'};

    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }

  handleChange(event) {
    this.setState({value: event.target.value});
  }

  handleSubmit(event) {
    alert('הטעם האהוב עליך: ' + this.state.value);
    event.preventDefault();
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          Pick your favorite flavor:
          <select value={this.state.value} onChange={this.handleChange}>
            <option value="grapefruit">אשכולית</option>
            <option value="lime">לימון</option>
            <option value="coconut">קוקוס</option>
            <option value="mango">מנגו</option>
          </select>
        </label>
        <input type="submit" value="שלוח" />
      </form>
    );
  }
}

```

טפסים - נסוחות ב-CodePen

כולם עובדים באופן מאוד דומה - `<input type="text">`, `<textarea>`, `<select>`-באופן כללי, זה גורם לכך ששבו ניתן להשתמש כדי למשוך קומפוננטה מבוקרת `value` כולם מקבלים מאפיין

הערה

מערך, המאפשר לכם לבחור אפשרויות רבות בתגית `value` ניתן להעביר למאפיין :

```
<select multiple={true} value={['B', 'C']}>
```

טפסים - Taggit ה-file input

מאפשרת למשתמש לבחורקובץ אחד או יותר מזכרון המכשיר שלהם להעלאה `<input type="file">` תגית HTML-בדרך JavaScript לשורת או לביצוע מניפולציות על ידי [File API](#).

```
<input type="file" />
```

היא נדונה בהרחבה יחד עם קומפוננטות React-מכיוון שהערך שלו הוא לקריאה בלבד, זהה קומפוננטה **שאינה מבוקרת** בבלטי מבוקרות אחרות **בשלב מאוחר יותר בתיעוד**.

טיפול בקלטאים מרובים

לכל אלמנט ולתת `name` מבוקרים, אתם יכולים להוסיף מאפיין `input` כאשר אתם צריכים להתמודד עם מספר אלמנטים אחדים ולתת `event.target.name` לפונקציה המתפלת לבחור מה לעשות על סמך הערך של:

לדוגמא:

```
class Reservation extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      isGoing: true,
      number0fGuests: 2
    };

    this.handleInputChange = this.handleInputChange.bind(this);
  }

  handleInputChange(event) {
    const target = event.target;
    const value = target.type === 'checkbox' ? target.checked : target.value;
    const name = target.name;

    this.setState({
      [name]: value
    });
  }

  render() {
    return (
      <div>
        <h1>Please indicate whether you're going or not</h1>
        <input checked="" type="checkbox" name="isGoing"/> I'm going
        <br/>
        <input type="checkbox" name="isGoing"/> I'm not going
      </div>
    );
  }
}
```

```

<form>
  <label>
    מטייל:
    <input
      name="isGoing"
      type="checkbox"
      checked={this.state.isGoing}
      onChange={this.handleInputChange} />
  </label>
  <br />
  <label>
    מספר אורחים:
    <input
      name="numberOfGuests"
      type="number"
      value={this.state.numberOfGuests}
      onChange={this.handleInputChange} />
  </label>
</form>
);
}
}

```

התקבָּל - נסזו זאת ב-

בהתאם לשם הקלט state-state- כדי לעדכן את מפתח ה ES6 שימו לב כיצד אנו משתמשים בתחביר שם מאפיין מחושב של שהתקבל:

```
this.setState({
  [name]: value
});
```

זהה ES5-זה שווה ערך לקוד זה:

```
var partialState = {};
partialState[name] = value;
this.setState(partialState);
```

הנוכחי אוטומטיות, אנחנו רק צריכים לקרוא לה עם החלקים state-state- חלקיקי ל state מזוגת () -בנוסף, לאחר ש השתנו.

ערך ריק בקלט מבוקר

על קומפוננטה מבוקרת מונעת מהמשתמש לשנות את הקלט אלא אם כן אתם חפצם בכך. אם ציינתם ערך prop- ציוו ערך ה value עדין ניתן לעריכה, יכול להיות שבטעות הגדרתם את key-value- אבל ה value undefined או null .

(נעול בהתחלה, אך הופך לנitinן לעריכה לאחר עיבוב קצר setTimeout-ה). הקוד הבא מגדים זאת.

```
ReactDOM.render(<input value="היי" />, mountNode);
setTimeout(function() {
  ReactDOM.createRoot(mountNode).render(<input value={null} />);
}, 1000);
```

אלטרנטיבות לקומפוננטות מבוקרות

זה יכול לפעמים להיות מיגע להשתמש בקומפוננטות מבוקרות, כי אתם צריכים לכתוב מטפל אירוע עבור כל דרך בה הנטונים זה יכול להיות מעצבן במיוחד כאשר אתם React שלכם יכולים לשנות ולשגר את כל מצב הלקוח באמצעות קומפוננטת במצבים אלו, ייתכן שתרצו לבדוק על React עם ספרייה שאינה או משלבים אפליקציית-React, ממירם קוד קיים ל [קומפוננטות בלתי מבוקרות](#), טכניקה חלופית למימוש טפסי קלט.

פתרונות כוללים

היא אחת האפשרויות [Formik](#), אם אתם מוחשים פתרון מלא הכלל אימות, מעקב אחר שדות שבוקרו, וטיפול בשילוח טופס אז אל תזנחו לימוד שלהם — state הפופולריות. עם זאת, הוא בנוי על אותם עקרונות של קומפוננטות מבוקרות וניהול

העברת רפנסים

העברת רפנסים היא טכניתה להעברה אוטומטית של **רפנס** דרך קומפוננטה לאחד מkomפוננטות הילד שלה. בדרך כלל, זאת טכניתה שלא נחוצה לרוב הקומפוננטות באפליקציה, אבל יכולה להיות שימושית במצבים מסוימים, בעיקר בספריות קומפוננטות רב פעמיות. התרחישים הנפוצים ביותר מתוארים בה.

DOM-העברה רפנסים לkomפוננטות ב

DOM-העברה רפנסים אלמנט כפטור פשוט בתוך ה `FancyButton` נכח לדוגמא komפוננטה כפטור `embed:forwarding-refs/fancy-button-simple.js`

מחביאות את פרטי השימוש שלהן, כולל את הפלט המרונדר שלהן. komפוננטות אחרות משתמשות ב React komפוננטות `FancyButton` כדי למנוע DOM-בדרך כלל לא יצטרכו להציג רפנס לאלמנט הCPFTOOR של ה `FancyButton` שהוא של השניה יותר מדי DOM-komפוננטות לסמן על מבנה ה.

היא יכולה להפוך לפחות נוחה `FeedStory` או `Comment` למורות שכimos כזה וצואה בkomפוננטות ברמת האפליקציה כמו השימוש בkomפוננטות `FancyButton` או `MyTextInput`. הלאה על עצמו כל חזר על עצמו פעמים רבות באפליקציה, דומה לשימוש בולפעמים אין ברירה אלא, `button` או `input` - שלhn כדי לשנות פוקוס, בחירה או אнимציה DOM-לגשת שירות לאלמנטי ה.

העברת רפנסים היא פיצ'ר שמאפשר komפוננטות **לקחת הפניה (רפנס)** שם קיבלו ולהעביר אותה הלאה **לקומפוננטת יلد**

כדי לקבל את הרפנס שהועבר אליו, ואז מעביר `FancyButton`-משתמש ב `React.forwardRef` בדוגמה הנ"ל, הCPFTOOR ב-DOM-אוטו לאלמנט הCPFTOOR שהוא מרנדר ב:

`embed:forwarding-refs/fancy-button-simple-ref.js`

יכולות לקבל רפנס לצומת הCPFTOOR עצמה ולהשתמש בה בעת הצורך `FancyButton`-בצורה זו, komפוננטות משתמשות בראופן ישר -- כדי להזמין ה CPFTOOR בו הוא השתמש בCPFTOOR ב.

להלן הסבר מפורט של מה שקרה בדוגמה לעיל:

1. ושים אותו במשתנה ה `React.createRef` - ע"י שימוש ב `ref` .
2. JSX-בעזרת השימוש במאפיין ה `FancyButton ref={ref}` - אנחנו מعتبرים את הרפנס למטה ל `React`.
3. `React` כפרמטר השני בתוך `... = > (props, ref) forwardRef` .
4. JSX-אנחנו מعتبرים את ארגומנט הרפנס למטה לכפטור על ידי שימוש במאפיין ה `ref` .
5. יקבע לצומת הCPFTOOR ב `ref.current` , כשהרפנס מחובר DOM.

הערה

פונקציות וגילות או `React.forwardRef` . ארגומנט הרפנס השני קיים ורק כשמדוברים komפוננטה עם קרייה לשלhn `props`-והוא גם לא זמין ב `ref` -komפוננטות מחלוקת לא מקובלות את ארגומנט ה

ניתן להעביר רפנסים גם למופיע של komפוננטות מחלוקת. DOM העברת רפנסים לא מוגבלת רק לkomפוננטות

הערה למתחזקי ספריות קומפוננטות

בספריות קומפוננטות, צריך להתייחס לשינויו שובר ולחזרו גרסה ראשית `forwardRef` -כשותחילים להשתמש בחדשה של הספרייה. הסיבה לכך היא שבעת לספרייה יש התנהגות נcrypt שונה (כמו איזה רפנסים מיושמים ואיזה סוג אוBIKטים מוצאים), זה יכול לשבור אפליקציות וספריות אחרות שהיו תלויות בהתנהגות הקודמת

כשהוא קיים גם כן לא מומלץ מאותה הסיבה: הוא משנה את התנהגות הספרייה `React.forwardRef` -השימוש המותנה בעצמה-ויכול לשבור את האפליקציות שימושת בה כשהן משדרות את גרסת ה-React.

העברת רפנסים בקומפוננטות מסדר גובה יותר

נתחיל עם HOCs-שידועות גם כה הטכניקה הזאת יכולה להיות שימושית ביותר בשימוש עם קומפוננטות מסדר גובה יותר של `props-console`: `embed:forwarding-refs/log-props-before.js`

דרכו לקומפוננטה שהוא עוטף, ולכן הפלט יהיה זהה. לדוגמה, אפשר להשתמש `props` -מעבר את כל ה-`"logProps"`-HOC-ה-props שולנו "fancy button": `embed:forwarding-refs/fancy-button.js`

כמו `prop` הוא לא `ref` -יש הסתייגות אחת לדוגמא שהציגנו: רפנסים לא יעברו הלאה. הסיבה היא שמתיחס אליו בצורה קצת שונה. אם תוסיפו את הרפנס לקומפוננטה כפטור ה-`props`-זהה כדי לתעד את כל ה-HOC-ב-`key` הקומפוננטה החיצוני, ולא לקומפוננטה העוטפה.

שלנו יהיו דוקא מוקשרים לקומפוננטה `FancyButton` -זה אומר שרפנסים שייעדו לקומפוננטה `LogProps`: `embed:forwarding-refs/fancy-button-ref.js`

בעזרת המשק `FancyButton` למולנו, אפשר להעביר את הרפנס בצורה מפורשת לקומפוננטה הפנימית `React.forwardRef`. `React.forwardRef` מקבל פונקציית רינדור עם פרמטרים `props` ו-`ref` ומחזיר צומת `React`: `embed:forwarding-refs/log-props-after.js`

הציג שם מותאים בכלי פיתוח

משתמשים בפונקציה הזאת כדי להציג מה לReact מקבל פונקציית רינדור. כלי הפיתוח של `React.forwardRef` עברו קומפוננטת העברת הרפנסים.

בכלי הפיתוח של "ForwardRef": `ForwardRef(myFunction)`:

`embed:forwarding-refs/wrapped-component.js`

לדוגמא, אם נתן שם לפונקציית הרינדור, כל הפיתוח יציג גם אותו:

`embed:forwarding-refs/wrapped-component-with-function-name.js`

של הפונקציה בצורה שתכלול את הקומפוננטה העוטפה `displayName` -אפשר לציין את ה-

`embed:forwarding-refs/customized-display-name.js`

[Go to TOC](#)

Fragments

נותנים לך לאחד רשימה של ילדים מלבני **Fragments**. עברו קומפוננטה שמחזירה אלמנטים מרובים -תבנית נפוצה ב-DOM-נוספים לnodes להוסיף.

```
render() {
  return (
    <React.Fragment>
      <ChildA />
      <ChildB />
      <ChildC />
    </React.Fragment>
  );
}
```

קיים גם **תחביר קצר** להצחרותם, אבל זה עדין לא נטמן בידי כלים פופולריים.

מוטיבציה

תבנית נפוצה עברו קומפוננטה שמחזירה רשימה של ילדים. קח לדוגמה את הקטע הזה של React:

```
class Table extends React.Component {
  render() {
    return (
      <table>
        <tr>
          <Columns />
        </tr>
      </table>
    );
  }
}
```

ישמש כהורה ב `<div>`-יהיה חוקי. אם ה-HTML-מרובים כדי שהרנדור של ה `<td>` יצטרך להחזיר אלמנטים `<td>` לתא. תהייה אינה חוקית HTML-או התוצאה של ה `<Columns />` של `render()`.

```
class Columns extends React.Component {
  render() {
    return (
      <div>
        <td>Hello</td>
        <td>World</td>
      </div>
    );
  }
}
```

התוצאה בפלט של ה `<Table />`:

```
<table>
  <tr>
    <div>
      <td>Hello</td>
```

```

<td>World</td>
</div>
</tr>
</table>

```

פותרים את הבעיה הזו.

שימוש

```

class Columns extends React.Component {
  render() {
    return (
      <React.Fragment>
        <td>Hello</td>
        <td>World</td>
      </React.Fragment>
    );
  }
}

```

לפלט תקין `<Table />` מה שמביא את

```

<table>
  <tr>
    <td>Hello</td>
    <td>World</td>
  </tr>
</table>

```

תחבר קצר

הוא נראה כמו Tagיות ריקות. קיים תחבר חדש וקצר יותר שתוכל להשתמש בו להצחה.

```

class Columns extends React.Component {
  render() {
    return (
      <>
        <td>Hello</td>
        <td>World</td>
      </>
    );
  }
}

```

או תכונות `keys`-אתה יכול להשתמש `</><>` באותו אופן שהשתמש בכל אלמנט אחר, אך הוא אינו תומך ב

Keyed Fragments

Fragments שימושי הוא כאשר מפותים אוסף `<React.Fragment>` עם התחבר המפורש יכול להזכיר `keys`. מובהר `props.items.map(item => (` לדוגמה, יצירת רשימה של תיאור - `fragments` לערך של

```

function Glossary(props) {
  return (
    <dl>
      {props.items.map(item => (
        // Without the `key`, React will fire a key warning

```

```
<React.Fragment key={item.id}>
  <dt>{item.term}</dt>
  <dd>{item.description}</dd>
</React.Fragment>
);
});
};
```

בעתיד, אנו נוסיף תמיכה לתוכנות נוספות, כגון `event` -הו התוכנה היחידה שאפשר להעביר ל `handlers`.

דוגמה חייה

הזה [עם JSX fragment](#) עם [CodePen](#) החדש של `fragment`.

[Go to TOC](#)

מתחלים

- "download.html"
- "downloads.html"

ומשאים קשורים דף זה הוא סקירה כללית של התיעד של

[בדף הבית שלנו או במדריך React לבניית ממשקי משתמש.](#) למדו הכל אודות JavaScript היא ספרייה React.

- [לנסות את React](#)
- [ללמד React](#)
- [להישאר מעודכנים](#)
- [גרסאות תיעוד](#)
- [משהו חסר?](#)

מגרש משחקים אונליין

מעט או יותר [לפי צרכיכם](#). בין אם אתם רוצים **React**-תוכננה מלכתחילה לaimoz הדרגי, ואתם יכולים להשתמש בReact הקישורים פשוט, או להתחליל יישום מורכב מופעל HTML להוסיף קצת אינטראקטיביות לדף, לקבלת טעימה של בסעיף זה יעזר לכם להתחליל.

מגרש משחקים אונליין

תוכלו להשתמש במגרש משחקים של קוד אונליין. נסו את תבנית שלום עולם ב-React אם אתם מעוניינים לשחק עם [CodePen](#), [CodeSandbox](#), או [Stackblitz](#).

זה, לעורך אותו, ולפתחו אותו ממערכת [HTML](#) אם אתם מעדיפים להשתמש בעורך הטקסט שלכם, תוכלו גם [להוריד קובץ הקבצים המקומיים בדףם](#). זה מאט את הטנספורמציה של הקוד בזמן הריצה, אז הינו ממליצים להשתמש בזה רק עבור הדוגמות פשוטות.

לאתר React הוסףת

[תיק דקהacha](#). לאחר מכן תוכלו להרוחיב בהדרגה את נוכחותה, או לשמר אותה בתחום [HTML](#) לדף [React](#) תוכלו להוסיף את לכמה ווידג'טים דינמיים בוחדים.

חדש React יצירת אפליקציית

עדין עשוי להיות האפשרות הטובה ביותר. לוקח רק דקה [Script פשוט עם Tag HTML](#) דף [React](#) בעת התחלת פרויקט להציג את זה!

שאנו JavaScript בכל שהאפליקציה גדלה, ייתכן שתרצו לשקל התקינה יותר מורכבת. ישנים מספר כלים שעובדה של ממליצים עבור יישומים גדולים יותר. כל אחד מהם יכול לעבוד עם הגדרה קענה או ללא הגדרות בכלל ומאפשר לכם לנצל את [למד איך](#) מלאה היתרונות של האkosיסיטם העשיר של

לلمוד **React**

מרקעים שונים ועם סגנוןות במידה שונות. בין אם אתם מעדיפים גישה תיאורטיבית יותר או גישה **React**-אנשים מגיעים למשית יותר, אנו מקווים שתמצאו את החלק הזה מועיל.

- אם אתם מעדיפים **לلمוד על ידי עשייה**, התחילה עם **המדריך המעשוי**.
- אם אתם מעדיפים **לلمוד קונספטים צעד אחר צעד**, התחילה עם **המדריך שלנו לקונספטים העיקריים**.

יש עוקמת במידה. עם תרגול ומעט סבלנות, אתם תתפסו את העניין **React**-כמו כל טכנולוגיה לא מוכרת, ל

דוגמאות ראשונות

נסו לשנות, עם עורך חי. גם אם איינכם ידעים עדין דבר על React מכיל מספר דוגמאות קטנות של **דף הבית** שלReact. את הקוד שלון ולראות כיצד זה משפייע על התוצאה.

למתחילים

הולך בקצב מהיר יותר ממה שאתם חשים בו בנוח, בדקו את **הסקירה הכללית זו של React** אם אתם מרגשים שהטייעוד של **React** מפורטת, וידידותית למתחילים. React היא מציגה את הקונספטים החשובים ביותר של **Tania Rascia**. לאחר שתסייעו, תנו לティיעוד עד ניסיון!

למעצבים

אם אתם מגיעים מרקע עיצובי, **משאבים אלה** הם מקוםמצוין להתחילה.

JavaScript משאבי

אתם לא צריכים להיות מומחיהם, אבל קשה יותר למדוד גם JavaScript מניה הিירות כלשהו עם תכונות בשפת תיעוד React. באותו זמן JavaScript גם

כדי לבדוק את רמת הדעת שלכם. זה ייקח לכם בין 30 דקות לשעה אבל **JavaScript** אלו ממליצים לעבור על **סקירה כללית זו של TER** תרגישו יותר בטחון בעת מידת React.

טיפ

הם אתרים אינטרנט -**MDN** ו-**javascript.info** -בכל פעם שאתם מרגשים מבולבלים בגלל שהוא שקשורה במצוינים לקבלת מידע. יש גם את **פורום תמיכת הקהילה** שבו אתם יכולים לבקש עזרה.

מדריך מעשי

אם אתם מעדיפים **לلمוד על ידי עשייה**, בדקו את **המדריך המעשוי** שלנו. במדריך זה, אנו בונים משחק איקס-עיגול בReact. אתם עלולים להתਪתות לדלג על זה כי אם לא בונים משחקים -- אבל תנו לו סיורי. הטכניקות שתלמדו במדריך הם היסוד ושליטה בהן תיתן לכם הבנה הרובה יותר عمוקה **React**, לבנית כל אפליקציה.

מדריך צעד-אחר-צעד

אם אתם מעדיפים **לلمוד קונספטים צעד אחר צעד**, **מדריך המושגים העיקריים** שלנו הוא המקום הטוב ביותר ביזור להתחילה. כל פרק בו מסתמך על הידע שהוצע בפרק הקודמים, וכך שלא תחמייצו שום דבר כל עד אטם מתקדמים לפי הסדר.

קורס לחשוב ב-React

-סוף סוף "עשה קлик" עבורם. זה נראה מדריך ה-React המקורי שבו React כרגע שבו **לחשוב ב-React** נוותנים קורדייט הרבה משתמשי React היישן ביותר אבל הוא עדין רלוונטי בבדיקה באותה מידת.

קורסים מומלצים

לפעמים אנשים מוצאים ספרי צד-שלישי וקורסיו ידאו מועילים יותר מהתייעוד הרשמי. אנו מתחזקים **רשימה של משאבים מומלצים בדף כל**, חלקם חינמיים.

קונספטים מתקדמים

אתם עשויים להתעניין בנושאים מתקדמים, ברגע שאתם מרגשים בנות עם **הकונספטים העיקריים** ושיחקתם קצת עם **refs** - או **context** כגון React יותר. חלק זה יציג בפניכם את התכונות החזקות, אך פחות נפוצות לשימוש, של

API Reference

לדוגמה, ספציפי של API חלק זה ב特意 שימושי כאשר ברכונכם לקבל פרטיים נוספים על מנת לאילו שימושים מתחדשות, `setState()` יכול לספק לכם פרטיים על אופן הפעולה של `React.Component API reference`. מחזור חיים שונות שימושות.

מילון מונחים ושאלות נפוצות

יש גם חלק שאלות נפוצות המוקדש **מילון המונחים** מכיל סקירה של המונחים הנפוצים ביותר ביזור שתראו ב特意 של **AJAX, state ביצוע בקש של קומפוננטה ומבנה קובץ**.

ליהיאר מעודכנים

כל דבר חשוב, כולל הערות שחרור גירסאות או הודיעות על React הוא המקור הרשמי לעדכונים מצוות **הבלוג של הפסקת תמכה בפיצ'רים**, יפורסמו שם כמקור ראשון.

אך לא תחמייצו שום דבר חינוי אם תקרואו את הבלוג בלבד, Twitter - ב- `@reactjs` תוכלו גם לעקוב אחר **חשוב**.

ראוייה הבלוג פוסט مثل עצמה, אך תוכלו למצוא קובץ שינויים מפורט עבור כל גירסה **בקובץ CHANGELOG.md** של **React**, וכן בדף **השינויים** בrifactorio של `.

גרסאות תיעוד

ניתן למצוא גרסאות ישנות יותר של התיעוד, React 16, React 16. מאז React 16. מש夸ר תמיד את הגרסה היציבה האחורונה של **בדף נפרד**. שימושו לב כי תיעוד עברו גרסאות קודמות הוא הקפהה ושמירה בזמן השחרור, והוא לא מתחדכן באופן רציף.

משהו חסר?

אם חסר משהו בתיעוד או אם מצאתם חלק כלשהו מבלבול, אני [הגישו בעיה לריפואיטורי של התיעוד](#) עם ההצעות שלכם
אנחנו אוהבים לשמעו מכם [@reactjs](#) ב-Twitter, או ציינו [לחשבון](#)!

[Go to TOC](#)

טיפול באירועים

ישנם כמה הבדלים תחביריים. DOM דומה מאוד לטיפול באירועים אלמנטיים של React לטיפול באירועים עם אלמנטים של

- ולא, (כל תחילת מילה באות גודלה פרט לראשונה)camelCase נכתבים באמצעות תחביר React שמות אירועים של **באותיות קטנות**.
- מעבירים פונקציה כמתפל האירוע, ולא מחרוזת JSX-ב**.

-HTML, למשל, ה

```
<button onclick="activateLasers()">
  הפעל לייזרים
</button>
```

-הוא מעט שונה ב-React:

```
<button onClick={activateLasers}>
  הפעל לייזרים
</button>
```

-אתם חיברים לקרוא ל-React. כדי למנוע התנהגות ברירת מחדל ב **false** הבדל נוסף הוא שאינכם יכולים להחזיר רגיל, כדי למנוע את התנהגות ברירת המחדל עבור קישור של פתיחת דף HTML במפורש. לדוגמה, עם **preventDefault**, אם החדש, אתם יכולים לכתוב:

```
<form onSubmit="console.log('!.לחצת שלפניך'); return false">
  <button type="submit">לחץ</button>
</form>
```

-זה יכול להיות במקום זאת, ב-React:

```
function Form() {
  function handleSubmit(e) {
    e.preventDefault();
    console.log('!.לחצת שלפניך');
  }

  return (
    <form onSubmit={handleSubmit}>
      <button type="submit">לחץ</button>
    </form>
  );
}
```

כך שאתה לא צריכים לדאוג, **W3C** מגדיר אירועים סינטטיים אלה בהתאם ל[מפרט](#). הוא אירוע סינטטי **e**, כאן **events** עיינו בהפנייה למדרך native events. בריאקט לא עובדים בדיקות כמו **events** **SyntheticEvent** כדי למדוד עוד.

לאחר DOM כדי להוסיף מאזינים לאלמנט **addEventListener** -אתם דרך כלל לא צריכים לקרוא ל-React-בעת שימוש ב**shnazar**. במקום זאת, רק שפקו מאזין כאשר האלמנט רונדר בהתחלה.

זהו דפוס נפוץ שמתפל אירוע הוא מתודת במחלקה. למשל, רכיב [ES6](#), כאשר אתם מגדירים קומפוננטה באמצעות **מחלקה Toggle** זה מレンדר כפטור המאפשר לשימוש עבור בין מצבי "ON"-ו-"OFF":

```
class Toggle extends React.Component {
  constructor(props) {
    super(props);
    this.state = {isToggleOn: true};

    // n-binding => יעבד בתוכה `this` הרצוי כדי לגרום לכך
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    this.setState(prevState => ({
      isToggleOn: !prevState.isToggleOn
    }));
  }

  render() {
    return (
      <button onClick={this.handleClick}>
        {this.state.isToggleOn ? 'ON' : 'OFF'}
      </button>
    );
  }
}
```

-נסו זאת ב-CodePen

כברירת מחדל, מתודות מחלקה אינן JavaScript-ב. בקריאה `this` עליכם להיות זהירים לגבי המשמעות של כארה הפונקציה `this` יהיה `undefined` ותעבירו אותה ל-`onClick`, `this` ל-`bind` אם תשבחו לעשות תקרה למעשה.

באופן כללי, אם אתם מתייחסים **מאלך שפונקציות פועלות ב-React**; זו אינה התנהגות ספציפית לאותה מתודה bind עליכם לעשות `onClick={this.handleClick}` למתודה ללא () אחרת, כמו:

מפרעה לכם, יש שתי דרכים לעקוף את זה. אם אתם משתמשים **בתחביר שדות ציבוריים של מחלקה** bind-אם קריאה ל-`bind` הנכונה bind callbacks:

```
class LoggingButton extends React.Component {
  // תחביר זה bound בתוכה `this` הוא bound handleClick.
  // אזהרה: זהו תחביר *ণיסיוני*.
  handleClick = () => {
    console.log('this is:', this);
  }

  render() {
    return (
      <button onClick={this.handleClick}>
        לחץ עליו!
      </button>
    );
  }
}
```

תחביר זה מופעל כברירת מחדל ב-[Create React App](#).

אם איןכם משתמשים בתחביר שדות של מחלוקת, באפשרותם להשתמש בפונקציית `hz` ב-callback:

```
class LoggingButton extends React.Component {
  handleClick() {
    console.log('this is:', this);
  }

  render() {
    // תחביר bound הוי `this`-תחביר זה מבטיח ש
    return (
      <button onClick={() => this.handleClick()}>
        לazarus
      </button>
    );
  }
}
```

מן נדר. ברוב המקרים, זה בסדר. עם זאת, `LoggingButton`-שונה בכל פעם שהבעה עם תחביר זה היא שנוצרת קומפוננטות נוספות יותר, קומפוננטות אלו עשוות לבצע רינדור חדש נוסף. באופן כללי אנוPROP-זה מועבר כcallback אם במבנה או באמצעות תחביר שדות מחלוקת, כדי למנוע בעית ביצועים זו binding ממליצים על ביצוע.

העברת ארגומנטים למטפל אירופים

הוא מזוהה השורה, כל אחת `id` בתוך לולאה זהה דבר נפוץ לארבע פרמטר נוסף למטפל האירופ. לדוגמה, אם מהאפשרויות הבאות תעבורו:

```
<button onClick={(e) => this.deleteRow(id, e)}>מחק שורה</button>
<button onClick={this.deleteRow.bind(this, id)}>מחק שורה</button>
```

בהתאם `Function.prototype.bind`-שתי השורות מעלה שוות, ומשתמשות בפונקציית `hz` ו-

יעבר ארגומנט שני לאחר המזוהה. עם פונקציית `hz`, אנחנו צריכים את אירופ `e` בשני המקרים, הארגומנט כל הארגומנטים הנוספים מועברים באופן אוטומטי `bind` להארבע אותו במפורש, אבל עם

[Go to TOC](#)

שלום עולם

נראית כך React-הדוגמה הקטנה ביותר ב:

```
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<h1>שלום עולם</h1>);
```

היא מראה את הכתובת "שלום עולם" על הדף.

[codepen-נסה זה ב](#)

לחצו על הקישור לעליה כדי לפתוח עורך אונליין. תרגישו בנוח לעשות כמה שינויים כדי שתראו איך הם משנהים את התוצאה.
לרוב העמודים במדריך הזה יהיו דוגמאות שניתנות לעריכה.

ביצד לקרוא מדריך זה

יסודות וקומפוננטות. ברגע שתשלטו בהם תוכלו ליצור אפליקציות React: במדריך אנו נבחן את החלקים של אפליקציות מורכבות בעזרת חלקים קטנים ועל ידי שימוש חזרה בהם.

עצה

מדריך זה נועד לאנשים שמעדיפים **למוד רעונות צעד אחר צעד**. אם הנכם מעדיפים ללמידה על ידי עשייה, בדקו את [המדריך המushi](#) שלנו. ייתכן שתמצאו שהמדריך והדרך המעשית משלימים אחד את השני.

תוכלו למצוא רשימה של כל הפרקים זה הפרק הראשון במדריך שמלמד צעד אחר צעד על הרוונות העיקריים של בסרגל הנויות. אם אתם קוראים את זה מכשיר נייד, אתם יכולים להיכנס לנויוט על ידי לחיצה על הכפתור הנמצא מצד הימני של תחתית המסך.

על ידי קראת **פרק React** כל פרק במדריך זה נבנה על הידע שהוזג בפרקים קודמים. **אתם יכולים ללמידה הרבה יותר** הוא הפרק הבא ["JSON" קונספטים עיקריים לפי סדר הופעתם בסרגל](#). לדוגמא, ["היכרות עם](#)

הערכת רמת הידע

אם איןכם מرجישים בטוחים JavaScript לכן אנו מניחים שיש לכם הבנה בסיסית של שפת, JavaScript היא ספרית על מנת לבדוק רמת הידע שלכם ולאפשר לכם לעקוב אחר המדריך **JavaScript בעצמכם או ממליצים שתעבירו על מדריך** JavaScript וגם React בily ללקת לאיבוד. זה יכול לקחת עד 30 דקות עד שעה, אבל כתוצאה, לא תרגישו שאתם לומדים גם באותו הזמן.

הערה

בשנים האחרונות, JavaScript חדש דוגמאות. אם לא עבדתם עם JavaScript מדי פעם, מדריך זה משתמש בתחביר [שלשות הנקודות האלו](#) אמורות לעדכן אתכם.

בואו נתחל!

המשיכו לגלול ותמצאו את הקישור [פרק הבא במדריך זה](#), מיד לפני התחלתית של האתר.

[Go to TOC](#)

קומפוננטות מסדר גובה יותר

הן לא בדיק HOCs. שעורת kod קומפוננטות React היא טכנית מתקדמת של (HOC) קומפוננטה מסדר גובה יותר אלא תבנית עיצוב שהתפתחה מהטבע הקומפואיציוני של React, חלק מההמשק של React.

מבחן היחסום, קומפוננטה מסדר גובה יותר היא פונקציה שלוקחת קומפוננטה ומיחזירה קומפוננטה אחרת.

```
const EnhancedComponent = higherOrderComponent(WrappedComponent);
```

בשונה מקומפוננטה רגילה שתרגמת מאפיינים למשק משתמש, קומפוננטה מסדר גובה יותר מתרגם קומפוננטה לkomponenntה אחרת.

ה-HOCs כמו למשל ה React שכיחות בספריות צד שלישי של Redux של `connect` ו- `createFragmentContainer` של Relay.

כאן נסביר למה קומפוננטות מסדר גובה יותר שימושית, ואיך ליצור אותה בעצמנו.

פעולות בשימוש נרחב HOC-שימוש ב

הערה

כדי לטפל בפעולות בשימוש נרחב. מאז, הבנו שהן גורמות ליותר בעיות מאשר מביאות `mixins` המלצנו בעבר על `mixins`-תועלת. [קראו כאן](#) למה עזבנו את השימוש ב

למרות זאת, יש לא מעט תבניות עיצוב שקשה למש בעזרת React - קומפוננטות הן ייחדות הקוד וכי ממוחזרות בקומפוננטות מסורתיות.

שמתחברת למקור נתונים חיצוני כדי להציג רשימה של העורות `CommentList` לדוגמה, נניח שיש לנו קומפוננטה:

```
class CommentList extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.state = {
      // הוא מקור נתונים חיצוני גלובלי
      comments: DataSource.getComments()
    };
  }

  componentDidMount() {
    // מקשרים לשינויים במקור הנתונים
    DataSource.addChangeListener(this.handleChange);
  }

  componentWillUnmount() {
    // מפסיקים להקשיב לשינויים
    DataSource.removeChangeListener(this.handleChange);
  }

  handleChange() {
```

```
// מעדכנים את רשימת הערות כשמתקבל שינוי
this.setState({
  comments: DataSource.getComments()
});

render() {
  return (
    <div>
      {this.state.comments.map((comment) => (
        <Comment comment={comment} key={comment.id} />
      ))}
    </div>
  );
}
}
```

לאחר מכן, נכתוב קומפוננטה שתזין לבלוג פוסט ייחיד, ש谟ומשת בצהרה דומה:

```
class BlogPost extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.state = {
      blogPost: DataSource.getBlogPost(props.id)
    };
  }

  componentDidMount() {
    DataSource.addChangeListener(this.handleChange);
  }

  componentWillUnmount() {
    DataSource.removeChangeListener(this.handleChange);
  }

  handleChange() {
    this.setState({
      blogPost: DataSource.getBlogPost(this.props.id)
    });
  }

  render() {
    return <TextBlock text={this.state.blogPost} />;
  }
}
```

לא זהות - הוא משתמשות בMETHODS שונות במקור הנתונים ומציגות מידע שונה. אבל רוב השימוש שלהן דומה:

- מתחילהים להאזין לשינויים במקור המידע -mount-אחרי ה
- - setState- כשמתקבל שינוי, קוראים ל
- מפסיקים להאזין לשינויים, unmount-ב

תחזור שוב ושוב. סביר להניח שנרצה ליצור state-state-באפליקציה גדולה, התבנית הזאת של האזנה למקור נתונים ועדכו ההפשה שתאפשר לנו להגיד את הפעולה הזאת במקום אחד ולהשתמש בה במספר קומפוננטות שונות. קומפוננטות מסדר גובה יותר מצינוות בדיק במצבים כאלה.

שמאזינה למקור הנתונים 1- ב- `CommentList`, `BlogPost`, שיווצרת קומפוננטה כמו נקרא לפונקציה `prop`-הfonקציה תקבל כאחד מהארוגומנטים, קומפוננטתILD שמקבלת את המידע הקשור כ `withSubscription`:

```
const CommentListWithSubscription = withSubscription(
  CommentList,
  (DataSource) => DataSource.getComments()
);

const BlogPostWithSubscription = withSubscription(
  BlogPost,
  (DataSource, props) => DataSource.getBlogPost(props.id)
);
```

הפרמטר הראשון הוא הקומפוננטה העטופה. הפרמטר השני מוחזר את המידע שאנו צריכים, באמצעות מקור הנתונים והprops הנוכחים.

כשהקומפוננטות `CommentListWithSubscription` ו- `BlogPostWithSubscription`, `CommentList` ו- `BlogPost` יתקבלו props עם המידע העדכני ביותר שהתקבל ממקור הנתונים `DataSource`:

```
// הפונקציה מקבלת קומפוננטה...
function withSubscription(WrappedComponent, selectData) {
  // ...ומחזירה קומפוננטה אחות...
  return class extends React.Component {
    constructor(props) {
      super(props);
      this.handleChange = this.handleChange.bind(this);
      this.state = {
        data: selectData(dataSource, props)
      };
    }

    componentDidMount() {
      // ...שלדותה למקור המידע להאזנה...
      dataSource.addChangeListener(this.handleChange);
    }

    componentWillUnmount() {
      dataSource.removeChangeListener(this.handleChange);
    }

    handleChange() {
      this.setState({
        data: selectData(dataSource, this.props)
      });
    }

    render() {
      // ...ומרנדרת את הקומפוננטה העטופה עם המידע החדש
      // ...props לב שאנו מביברים את כל ה-HOC
      // ...בצורה של props ל
      return <WrappedComponent data={this.state.data} {...this.props} />;
    }
  };
}
```

לא משנה את קלט הקומפוננטה, וגם לא מעתיקה את התנהלותה שלה באמצעות הורשה. במקרה זאת, ה-HOC-שים לו שהיא פונקציה טהורה שלא גורמת לשום תופעות לוויי HOC-עותפת את הקומפוננטה המקורית בקומפוננטה מכילה. ה-HOC

שבעזרתו היא `data` חדש בשם `prop` מהקומפוננטה המכילה, וגם `data`-וְהוּא! הקומפוננטה העטופה מקבלת את כל הלא צריכה לדעת מה ולמה עושים עם המידע, והקומפוננטה העטופה לא צריכה לדעת HOC-מןדרת את הפלט שלה. כך המאיפה המידע הגיע.

היא פונקציה וגילה, תוכלו להעביר לה יותר או פחות ארגומנטים לפי הצורך. `withSubscription` כיוון שהfonקציה עד יותר מתוכן הקומפוננטה העטופה. או HOC - כדי להפריד את ה `data` לדוגמה, תוכלו לשנות את השם של האו של מקור הנתונים. כל השינויים האלה `shouldComponentUpdate` שתוכלו לקבל ארגומנט שמשנה את התצורה של יש שליטה מלאה על הגדרת הקומפוננטה העטופה HOC-אפשרים כי ל.

כך ניתן `props` והקומפוננטה העטופה נשלט לגמרי על ידי `withSubscription` כמו קומפוננטות, ה"חוואה" בין הפונקציה לקומפוננטה העטופה. תוכנה שמאוד `props`-אחת לאחרות בклות, כל דעון מספקות את אותם ה HOC להחליף שימוש מועילה כמשנים ספרייה לטעינת מידע, לדוגמה.

השימוש בקומפוזיציה במקום מוציאה המקורי

יש לנו HOC-של הקומפוננטה (וכל מוציאה אחרת) בתוך ה `prototype` - עמדו בפני היפויו לשנות את ה

```
function logProps(Component) {
  Component.prototype.componentDidUpdate = function(prevProps) {
    console.log('Current props: ', this.props);
    console.log('Previous props: ', prevProps);
  };
  // העובדה שאנו מחדירים את הקלט המקורי הוא רמז לכך שהקלט עבר
  // מוציאה ככליה
  return Component;
}

// יתעד כל prop שמתקיים EnhancedComponent
const EnhancedComponent = logProps(Component);
```

יש כמה בעיות עם מוציאה. הבעה הראשונה היא שאפשר להשתמש בקומפוננטה שהועברה כקלט בנפרד. מעבר לזה, אם הראשונה HOC-התפקיד של ה `EnhancedComponent` שוג משנה את `componentDidUpdate`, נוסף ל HOC תישמו גם לא יעבד עם קומפוננטות פונקציית לא מתודות מחזור חיים HOC - יירמס!

HOCs יוצרת הפשטה דולפת - המשתמש צריך לדעת איך מה קורה בתוך הקוד כדי למנוע מעימות עם HOCs - מוציאה באחרות.

שימושים בקומפוזיציה, ע"י עטיפת קומפוננטה הקלט בקומפוננטה מכילה HOCs במקום מוציאה, עדיף לכתוב:

```
function logProps(WrappedComponent) {
  return class extends React.Component {
    componentDidUpdate(prevProps) {
      console.log('Current props: ', this.props);
      console.log('Previous props: ', prevProps);
    }
    render() {
      // מוטפים את קומפוננטה הקלט בקומפוננטה מכילה, בלי מוציאה!
      return <WrappedComponent {...this.props} />;
    }
  }
}
```

בדוגמה מספקת את אותה הפונקציונליות של הגרסה שעברה מוציאה שהציגו קודם, בלי הפוטנציאלי ליצור עימותים HOC-ה עם קומפוננטות אחרות. בנוסף, היא תעבור כמו שצריך גם עם קומפוננטות פונקציה וקומפוננטות מחלקה. כיוון שהיא פונקציה אחרת או אפילו עם עצמה HOC טהורה, אפשר לשלב אותה עם

קומפוננטות (container components). ובנויות עיצוב בשם **קומפוננטות מכילות** HOCs יכול להיות שימושם לב לדמיון בין state-מכילות הן חלק אסטרטגיית פיצול אחריות בין פעולות ברמה גבוהה וברמה נמוכה. הן מנהלות דברים כמו האזנה ומשתמשות בקומפוננטות מכילות חלק HOCs. לקומפוננטות שmailto: שmailto: מושך משמש props ומעבירות מהמיושר שלהן. אפשר לחשב עליהם כקומפוננטות מכילות עם פרמטרים

לא קשרים לקומפוננטות עטופות props מסווגות לגבי העברת

קומפוננטות מסדר גביה יותר מוסיפות פיצרים לקומפוננטה. הן לא אמורים לשנות את התפקיד של הקומפוננטה באופן תספק מושך דומה לקומפוננטה העטופה HOC-משמעותי. ניתן לצפות שקומפוננטה המוחזרת מ

שלא בהכרח קשרים אליה הלאה לקומפוננטה העטופה. בדרך כלל ניתן למצוא בהן מותחת props צריכה להעביר HOC-ה render: שנראית פחות או יותר ככה

```
render() {
  // נפלטר החוצה את ה-props המיותרים הספציפיים ל-HOC זהה שלא אמורים להיות
  // מועברים הלאה
  const { extraProp, ...passThroughProps } = this.props;

  // וכיסוי props לתוך הקומפוננטה העטופה. אלו בדרך כלל ערכיהם מה-state או
  // מתחודות מושך.
  const injectedProp = someStateOrInstanceMethod;

  // ונכשיו נוכל להעביר את ה-props לkomponenntה העטופה
  return (
    <WrappedComponent
      injectedProp={injectedProp}
      {...passThroughProps}
    />
  );
}
```

ನשארות גמישות כדי שנוכל למחזר אותן במקומות רבים באפליקציה HOCs - המוסכמה הזאת עוזרת לוודא שה

מוסכמות למקסום קומפוזיציה

נראות אותו הדבר. לעיתים הן מקבלות ארגומנט אחד בלבד, הקומפוננטה העטופה HOC - לא כל ה

```
const NavbarWithRouter = withRouter(Navbar);
```

מוועבר אובייקט קונפיגורציה שמצוין את המידע, Relay - בדרך כלל הן מקבלות ארגומנטים נוספים. בדוגמה הזאת מ: שהקומפוננטה תלואה בו

```
const CommentWithRelay = Relay.createContainer(Comment, config);
```

נראית כך HOCs - החתימה השכיחה ביותר ל

```
// React Redux פונקציית connect ` ` ש
const ConnectedComment = connect(commentSelector, commentActions)(CommentList);
```

מה?! פשוט יותר להבין מה קורה כאן כשמפצלים את שני החלקים.

```
// היא פונקציה שמחזירה פונקציה שמחזירה פונקציה אחת
const enhance = connect(commentListSelector, commentListActions);
// הפונקציה המוחדרת היא HOC, שמחזירה קומפוננטה שמחזירה Redux store-ל
const ConnectedComment = enhance(CommentList);
```

היא פונקציה מסדר גובה יותר שמחזירה קומפוננטה מסדר גובה יותר `connect`, במילים אחרות

עם ארגומנט אחד כמו זה שמוחזר על ידי HOCs. יכול להיות שהוא מבלב או לא נכון, אבל יש בזה מאפיינים שימושיים כל מאד לשלב פונקציות שיש להן פلت מסווג זהה. `Component => Component` משתמש בחתימה `connect` -פונקציית ה-`פלט` הקולט שלהן.

```
// במקום זה...
const EnhancedComponent = withRouter(connect(commentSelector)(WrappedComponent))

// תומכו להשתמש בפונקציית שירות קומפוזיציה ...
// compose(f, g, h) זהה ל - (...args) => f(g(h(...args)))
const enhance = compose(
  // שתי אלה הן HOCs עם ארגומנט ייחיד
  withRouter,
  connect(commentSelector)
)
const EnhancedComponent = enhance(WrappedComponent)
```

עד הצעה נסיונית של - `connect`-decorators -מסוג דומה כ HOCs -ו- המאפיין הזה מרצה לנו להשתמש ב- JavaScript.)

lodash -כ) lodash מסופקת על ידי ספריות צד שלישי רבות כגון `compose` פונקציית השירות `lodash.flowRight`, Redux Ramda.

מוסכמות לעיטוף השם המוצג בשבייל דיבאגינג

כמו כל קומפוננטה אחרת. בשבייל להקל [React Developer Tools](#)-מופיעת ב-HOCs הקומפוננטות המכילות שמיוצרות על ידי HOC-על דיבאגינג, כדאי לתת לקומפוננטה שם שմסביר שהיא נוצרה כתוצאה שימוש ב-

של הקומפוננטה העוטפה. לדוגמה, אם הקומפוננטה מסדר `(displayName)` הטכניקה הנפוצה ביותר היא לעוטף את השם נתוך את השם המוצג ונחזיר `withSubscription`, והקומפוננטה העוטפה נקראת `CommentList`, גובה יותר נקבע `WithSubscription(CommentList)`:

```
function withSubscription(WrappedComponent) {
  class WithSubscription extends React.Component {/* ... */}
  WithSubscription.displayName =
    `WithSubscription(${getDisplayName(WrappedComponent)})`;
  return WithSubscription;
}

function getDisplayName(WrappedComponent) {
  return WrappedComponent.displayName || WrappedComponent.name || 'Component';
}
```

הסתיגיות

- קומפוננטות מסדר גובה יותר באוט עם מספר הסתיגיות שלא מובנות מאליו, במיוחד לאלה שחדשים ל React.

בתוכה מותודת ה-HOCs -אל תשתמש ב render

משתמש בזהות קומפוננטה כדי להחליט אם לעדכן את עץ ([Reconciliation](#)) אלגוריתם ההבדלה של זהה (`====`) לקומפוננטה `render` - הקומפוננטות או לורוק אותו וליצור חדש במקומו. אם הקומפוננטה המוחזרת מעדכן באופן וקורסיבי את עץ הקומפוננטות על ידי השוואתו עם העץ החדש. אם הם לא זרים, עץ זהים, מחרינדו הקודם הקומפוננטות הנוכחיים יזרק.

HOC בדרך כלל, לא צריך לחשב על זה. אבל זה חשוב בזמן שימוש בקומפוננטות מסדר גובה יותר כי זה אומר שאי אפשר לשימוש בקומפוננטה בעוזרת מותודת `render`:

```
render() {
  // גרסה חדשה של נוצרות בכל רינדור
  // EnhancedComponent1 !== EnhancedComponent2
  const EnhancedComponent = enhance(MyComponent);
  // זה גורם לכל עץ הקומפוננטה להיזרק ולהיווצר מחדש כל פעם!
  return <EnhancedComponent />;
}
```

של הקומפוננטות ושל כל קומפוננטות הילד שלה להעלם `state`-הבעיה היא לא רק ביצועית - היא גם תגרום ל

מחוץ להגדרת הקומפוננטות כך שהקומפוננטה תוצרך רק פעם אחת. לאחר מכן, הזהות HOCs -במקום זאת, הגדרו את הולה תישאר עקבית עם כל רינדור, שזה מה שבדרך כלל נרצה בכל מקרה.

במקרים הנדרים שהם תרצו ליצור קומפוננטה מסדר גובה יותר באופן דינמי, תוכלו לעשות זאת מתוך אחת ממתודות מחזוץ `constructor` -החיים של הקומפוננטה, או ה

חוובה להעתיק מותודות סטטיסטיות

חוופשות מותודה Relay לדוגמה, קומפוננטות מכילות של React. לעיתים יש צורך בהגדרת מותודה סטטיסטית בקומפוננטה `getFragment` כדי לאפשר קומפוזיציה של פרגמנטיות של GraphQL. סטטיסטית בשם

על קומפוננטה, הקומפוננטה המקורית נעתפת על ידי הקומפוננטה המכילה. זאת אומרת HOC -כשמשתמשים בשאהקומפוננטה החדשה לא כוללת את המותודות הסטטיסטיות של הקומפוננטה המקורית.

```
// נגיד רותודה סטטיסטית
WrappedComponent.staticMethod = function() {/*...*/}
// עכשוויו נשתמש ב- HOC על קומפוננטה
const EnhancedComponent = enhance(WrappedComponent);

// הקומפוננטה שנקלע לא מגדרה את המותודה הסטטיסטית
typeof EnhancedComponent.staticMethod === 'undefined' // true
```

כדי לפטור את הבעיה הזאת, ניתן להעתיק את המותודות לתוך הקומפוננטה המכילה לפני שימושיהם אותה:

```
function enhance(WrappedComponent) {
  class Enhance extends React.Component {/*...*/}
  // נאלאן לדעת בדיעוק אידה מותודות להעתיק
```

```
Enhance.staticMethod = WrappedComponent.staticMethod;
return Enhance;
}
```

כדי להעתיק באופן `hoist-non-react-statics`-החישרונו הוא שנאלץ לדעת בדיק איזה מתחודות להעתיק. אפשר להשתמש ב-`hoistNonReactStatic` בלבד אלה שמודדות על יד (אווטומטי את כל המתחודות הסטטיסטיות React):

```
import hoistNonReactStatic from 'hoist-non-react-statics';
function enhance(WrappedComponent) {
  class Enhance extends React.Component {/*...*/}
  hoistNonReactStatic(Enhance, WrappedComponent);
  return Enhance;
}
```

פתרון נוסף הוא ליצא את המתחודות הסטטיסטיות בנפרד מתוך הקומפוננטה עצמה.

```
// במקומו ...
MyComponent.someFunction = someFunction;
export default MyComponent;

// נוציא את המתחודות בנפרד ...
export { someFunction };

// נזקiba את שתיהן כדי לשלב אותן ...
import MyComponent, { someFunction } from './MyComponent.js';
```

הפרנסים לא מועברים

הלאה לקומפוננטה העטופה, זה לא עובד `props` - למרות שהמוסכמתה לקומפוננטות מסדר גובה יותר היא להעביר את כל ה-`props` בו באופן מיוחד. אם `prop` כמו `key`, `ref` או `key` לא בדיק - הסיבה לכך היא שה-`key` (refs) עברו ופרנסים הרפרנס מתיחס למופיע הקומפוננטה המכילה החיצונית ביותר, HOC תוסף ופרנס לאלמנט שהקומפוננטה שלו נוצרה על ידי ולא לקומפוננטה העטופה.

תוכלו [למצוא עוד](#) (שנינו לשימוש החל מגרסת `React.forwardRef` (16.3). [הפתרון](#) לבעה זאת הוא להשתמש במשק [מידע בעמוד העברת רפנסים](#).

משלך Hooks בנית

ambilי לכטוב מחלוקת Hooks ופיצ'רים אחרים של state-הם נתונים לנו להשתמש ב React 16.8. הם תוספת חדשה ב Hooks.

משלך נותן לך לחלץ לוגיקת קומפוננטות לפונקציות שניות לעשות בהן שימוש חוזר Hooks בנית.

ראינו את קומפוננטה זו מיישום צ'אט שמציג הודעה שמצוינת האם חבר מחובר או מנותק, [Effect Hook](#)-כשמדובר על שימוש ב:

```
import React, { useState, useEffect } from 'react';

function FriendStatus(props) {
  const [isOnline, setIsOnline] = useState(null);

  useEffect(() => {
    function handleStatusChange(status) {
      setIsOnline(status.isOnline);
    }

    ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
    return () => {
      ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
    };
  });

  if (isOnline === null) {
    return 'טען...';
  }
  return isOnline ? 'מחובר' : 'מנוטק';
}
```

עבדשו נגד שימוש הcz'אט שלנו מכיל רשימת אנשי קשר, ואנו רוצים לרדר שמות של משתמשים מחוברים עם צבע ירוק. נוכל אבל זה לא יהיה אידיאלי [FriendListItem](#) להעתיק ולהדביק לוגיקה דומה לעיל לתוך הקומפוננטה:

```
import React, { useState, useEffect } from 'react';

function FriendListItem(props) {
  const [isOnline, setIsOnline] = useState(null);

  useEffect(() => {
    function handleStatusChange(status) {
      setIsOnline(status.isOnline);
    }

    ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
    return () => {
      ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
    };
  });

  return (
    <li style={{ color: isOnline ? 'green' : 'black' }}>
      {props.friend.name}
    </li>
  );
}
```

במוקם, נרצה לשתף את לוגיקה זו בין `FriendStatus` ו-`FriendListItem`.

בין קומפוננטות `stateful` היו לנו שתי דרכי פופולריות לשתף לוגיקה שהיא React-Hooks-באופן מסורתי בפותרים הרבה מאותן בעיות מביי להכריח אותנו להוסיף עד Hooks **קומפוננטות מסדר גבוה יותר**. בעת נשתכל על איך קומפוננטות לעז.

モותאם אישית Hook חילוץ

-אנו מחלכים אותה לפונקציה שלישית. שתי הקומפוננטות וה-JavaScript כשאנו רוצים לשתף לוגיקה בין שתי פונקציות ה們 פונקציות, אז זה עובד גם בשbillם Hooks:

אחרים. -והיא יכולה לקרוא ל "use" שמה מתחילה עם JavaScript מותאם אישית הוא פונקציית Hook: המותאם אישית הראשון שלנו Hook-היא ה `useFriendStatus`:

```
import { useState, useEffect } from 'react';

function useFriendStatus(friendID) {
  const [isOnline, setIsOnline] = useState(null);

  useEffect(() => {
    function handleStatusChange(status) {
      setIsOnline(status.isOnline);
    }

    ChatAPI.subscribeToFriendStatus(friendID, handleStatusChange);
    return () => {
      ChatAPI.unsubscribeFromFriendStatus(friendID, handleStatusChange);
    };
  });

  return isOnline;
}
```

Hooks-אין שום דבר חדש בתוכו – הלוגיקה מועתקת מהקומפוננטה, תודא שאתה קורא ל המותאם אישית שלך Hook-אחרים ללא תנאי ברמה העליונה של ה.

モותאם אישית לא צריך שתהיה לו חתימה ספציפית. אנחנו יכולים להחליט מה הוא לוקח React, Hook בשונה מקומפוננטת כארוגמנטים, ומה, אם בכלל, הוא אמרו להחזיר. במקרים אחרים, זה בדוק כמו פונקציה וגילה. השם שלו צריך להתחיל תמיד תקפים לגבי `use` כך שתמיד יוכל לדעת **שהוקו** עם `use`.

לסתוטוס של חבר. זאת הסיבה שהוא מקבל את subscribe היא לעשות לנו `useFriendStatus` המטורה של ה `friendID`: כארוגמנט, ומחייב האם החבר מחובר או לא:

```
function useFriendStatus(friendID) {
  const [isOnline, setIsOnline] = useState(null);

  // ...

  return isOnline;
}
```

הモותאם אישית Hook-עכשו נראה כיצד יוכל להשתמש ב.

モთאם אישית Hook-שימוש ב

בהתחלתה, המטרה המוצהרת שלנו הייתה להסיר את הלוגיקה הכפולה מהkomponenotot. **FriendStatus** ו- **FriendListItem** שניהם רצו לדעת האם חבר מחובר.

אנחנו יכולים פשוט להשתמש בה, **useFriendStatus** עכשו כשהילצנו את הלוגיקה זו ל

```
function FriendStatus(props) {
  const isOnline = useFriendStatus(props.friend.id);

  if (isOnline === null) {
    return 'טוען...';
  }
  return isOnline ? 'מחובר' : 'מנוטק';
}

function FriendListItem(props) {
  const isOnline = useFriendStatus(props.friend.id);

  return (
    <li style={{ color: isOnline ? 'green' : 'black' }}>
      {props.friend.name}
    </li>
  );
}
```

האם קוד זה שווה ערך לדוגמאות המקוריות? כן, הוא עובד בדיק באותה דרך. אם נסתכל קרוב יותר, נראה שלא ביצענו מותאמים אישית Hooks. שינויים להתנהגות. כל מה שעשינו הוא לחץ קוד נפוץ בין שני פונקציות לתוך פונקציה נפרדת מאשר פיצ'ר של Hooks הם מוסכמה שנובעת מהעיצוב של React.

בהתחלתה? בבקשתה עשה זאת. המוסכמה הזו היא **"use"** המותאמים אישית **Hooks**-האם אני צריך להוסיף לשם של ה-**Hooks** בגלל שלא נוכל לדעת אם פונקציה מסוימת **Hooks** חשובה מאוד. בפועל, לא נוכל לבדוק באופן אוטומטי הפרות של חוקי בתוכה Hooks-מכילה קריואות ל.

מותאמים אישית הם מנגנון לשימוש חומר Hooks. לא? **חולקות Hook האם שני komponenotot שמשמשות** באתו מותאם Hook-אבל בכל פעם שאתה משתמש ב, (ולזכור את הערך הנוכחי **subscription** כמו הגדרת) **stateful** בלוגיקה שהיא בתוכן הם מבודדים למגרמי effects-והו א-אישית, כל ה-

-מבודד. בgal שלאנו קוראים ל **state** מותאם אישית **Hook** או? כל קראיה ל **state** מותאם אישית **מקבל Hook** וכמו **useState** ו- **useEffect**. הקומפוננטה שלנו קוראת ל **React** ישירות, מנוקדת המבט של **useState** ו- **useEffect**, והוא עצמאיים למגרמי **useState** ו- **useEffect**, אנו יכולים לקרוא ל. פעמים רבות בקומפוננטה אחת, והם יהיו עצמאיים למגרמי **useState** ו- **useEffect**, אנו יכולים לקרוא ל

טיפ: העבר מידע בין Hooks

הם פונקציות, ניתן להעביר מידע ביןיהם Hooks-מכיוון ש

להדגים זאת, נשתמש בקומפוננטה אחת מדוגמת הצ'אט ההיפותטית שלנו. זה ברור לנו של הדעות צ'אט שמציג האם החבר: הנבחר הוא מחובר:

```
const friendList = [
];

function ChatRecipientPicker() {
```

```

const [recipientID, setRecipientID] = useState(1);
const isRecipientOnline = useFriendStatus(recipientID);

return (
  <>
    <Circle color={isRecipientOnline ? 'green' : 'red'} />
    <select
      value={recipientID}
      onChange={e => setRecipientID(Number(e.target.value))}>
    >
      {friendList.map(friend => (
        <option key={friend.id} value={friend.id}>
          {friend.name}
        </option>
      ))}
    </select>
  </>
);
}

```

ומעדכנים אותו אם המשתמש בוחר חבר אחר ב state recipientID-אנחנו שומרים את ה <select>.

אנחנו יכולים state נוותנת לנו את הערך האחרון של המשתנה recipientID-בגלל שהקריאה ל useState Hook המותאמת אישית שלנו useFriendStatus-להעביר את זה ל כארוגומנט.

```

const [recipientID, setRecipientID] = useState(1);
const isRecipientOnline = useFriendStatus(recipientID);

```

זה נותן לנו לדעת האם החבר שנבחר נבחר הוא מחובר. אם אנחנו בוחרים חבר אחר ומעדכנים את המשתנה recipientID, לסתוטוס של הנבחר subscribe מהחבר שנבחר קודם לכן, ויעשה unsubscribe לנו יבצע.

useYourImagination() {#useyourimagination}

בעבר. אתה יכול React מותאים אישית מציעים את הפעולות של שיתוף לוגיקה שלא היה אפשרי בקומפוננטות Hooks. מותאים אישית שמכסים מגוון רחב של מקרי שימוש כמו טיפול בטפסים, אнимציה לכתוב שהם קלים לשימוש כמו Hooks טיררים, וכגראה הרבה עד שלא שקלנו עדיין. ובנוסף, אתה יכול לבנות פיצרים של React.

נסה לא להוסיף הפשטות מוקדם מדי. עכשו שkomponenntות פונקצייה יכולות לעשות יותר, סביר להניח שkomponenntת הפונקציה אבל אנחנו גם Hooks-המוצעת בקוד שלך תהפוך לארכואה יותר. זה נורמלי – אל תרגיש שאתה חיב ישר לפצל אותה ל מותאם אישית יכול להציג לוגיקה מורכבת מארחורי משק פשוט, או Hook מעודדים אותו להתחיל לאחר מכן מקרים שבהם לעזור לפורים komponenntה בעיתית.

אינו מרכז useState. שמנוהל בדרך של גוף של עניין local state לדוגמה, אולי יש לך komponenntה מרכיבת שמכללה הרבה את לוגיקת העדכון בклות כך שיכל להיות שתעדיך כתוב את זה בתור Redux reducer:

```

function todosReducer(state, action) {
  switch (action.type) {
    case 'add':
      return [...state, {
        text: action.text,
        completed: false
      }];
  }
}

```

```

        }];
        // ... פטולות אחרות ...
    default:
        return state;
    }
}

```

קטנים יותר reducers-הם נוחים לבדיקה בבדיקה, ובקנה מידה כדי לבטא לוגיקת עדכון מורכבת. ניתן לפרק אותם לאו שלא תרצה להתקין ספרייה, ובמידת הצורך local state-ב React, למורות זאת, תוכל גם ליהנות מஹיטוניות של שימוש בNospat.

גרסה? של הקומפוננטה שלנו עם `useReducer` Hook נוהל `local state` שמאפשר לנו לנוהל reducer? איז מה אם נוכל לנכתב מופשטת של זה יכולה להיראות כמו זה:

```

function useReducer(reducer, initialState) {
    const [state, setState] = useState(initialState);

    function dispatch(action) {
        const nextState = reducer(state, action);
        setState(nextState);
    }

    return [state, dispatch];
}

```

לעפל בניהול reducer-עכשו נוכל להשתמש זהה בתוך הקומפוננטה שלנו, ולתת לו state:

```

function Todos() {
    const [todos, dispatch] = useReducer(todosReducer, []);

    function handleAddClick(text) {
        dispatch({ type: 'add', text });
    }

    // ...
}

```

בתוך קומפוננטה מורכבת הוא נפוץ מספיק כך שבנינו את local state עם reducer הדרוש לנוהל React. מובנים אחרים [בתיקות Hooks API](#).

[Go to TOC](#)

ב- שימוש Effect Hook

ופיצ'רים נוספים של ריאקט מביי לכתחזק מחלוקת state - הםאפשרים לנו להשתמש ב Hooks -React 16.8.

נותן לנו לבצע תופעות לוואי בתוך קומפוננטות פונקציונליות -ה Effect Hook:

```
import React, { useState, useEffect } from 'react';

function Example() {
  const [count, setCount] = useState(0);

  // - componentDidMount 1- componentDidUpdate:
  useEffect(() => {
    // עדכון API- כתובת העמוד על ידי שימוש ב
    document.title = `You clicked ${count} times`;
  });

  return (
    <div>
      <p>{count} ללחצת</p>
      <button onClick={() => setCount(count + 1)}>
        לחץ עליי
      </button>
    </div>
  );
}
```

מהעמדת הקודם, אבל הוספנו פיצ'ר חדש זהה:קובעים את כוורתה העמוד להודעה counter - הקטע לעיל מבוסס על דוגמת ה מותאמת שככלות את מספר ההקלקות.

בקומפוננטות ריאקט הם כולם דוגמאות של תופעות לוואי. בין אם DOM -שינוי ידני של ה subscription, אחזור מידע, הגדרת לאו דווקא אתה וגיל לקרווא לפעולות הללו "תופעות לוואי" (או רק "תופעות"), ווב הסיכויים שביצעת אותן בקומפוננטות שלך בעבר.

טיפ

אם יש לך נסיון עם מתודות מהאזור חיים בחלוקת ריאקט, אתה יכול לחשב על useEffect - כ componentDidMount ,(componentDidUpdate ,componentWillUnmount).

ישנם שני סוגי של תופעות לוואי בקומפוננטות ריאקט: אלה שלא דורשים ניקוי, ואלה שכן. נסתכל על הבחנה זו ביתר פירוט.

ללא ניקוי

- DOM -בקשות רשת, שינויים ידניים של ה DOM -לעתים, אנו רוצים להריץ קוד **נוסף** לאחר שריאקט עדכנת את ה שללא דורשים ניקוי. אנו אומרים זאת בגלל שניתן להריץ אותם ומיד לשכוח מהם. הבה נשווה effects המ דוגמאות של logging נתנים לנו לבטא את תופעות הלואי הללו Hooks- בין איך מחלוקת ו-

דוגמאות עם שימוש במחולקות

לא אמורה ליצור תופעות לוואי. זה יהיה מוקדם מדי – בדרך כלל נרצה `render` במחולקות קומפוננטות בריאקט, המתודה `DOM`-שלנו לאחר שריאקט עדכנה את ה `effects` -לבצע את ה

בזהירה. זאת הסיבה שבמחולקות ריאקט, אנו שמים תופעות לוואי בתוך `componentDidMount` ו- `componentDidUpdate`. בקומפוננטה שמעדכנת את כוורתה העמוד מיד לאחר שריאקט משנה את ה `counter` לדוגמה שלנו, הנה מחולקת:

```
class Example extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0
    };
  }

  componentDidMount() {
    document.title = `${this.state.count}` `פעמים`;
  }

  componentDidUpdate() {
    document.title = `${this.state.count}` `פעמים`;
  }

  render() {
    return (
      <div>
        <p> ${this.state.count} פעמים</p>
        <button onClick={() => this.setState({ count: this.state.count + 1 })}>
          לחץ עליי
        </button>
      </div>
    );
  }
}
```

שים לב כיצד אנו **צריכים לשכפל את הקוד בין שתי מתודות מחזור החיים במחולקה**.

או אם, ("זה בגלל שבמקרים רבים אנו רוצים לבצע את אותה תופעת הלואוי ללא קשר לאם הקומפוננטה "הותקנה היא עודכנה. מבחינה מסווגית, אנו רוצים שזה יקרה אחרי כל רינדור – אבל לקומפוננטות מחולקות בריאקט אין מתודה זאת. נוכל להלץ מתודה נפרדת אבל נדרש עדין לקרוא לה בשתי מקומות.

בהה נראה כיצד נוכל לעשות את אותו הדבר עם `useEffect`.

דוגמה של שימוש ב Hooks

ראינו כבר את הדוגמא זו בראש העמוד, אבל בהה נסתכל עליה מקרוב:

```
import React, { useState, useEffect } from 'react';

function Example() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    document.title = `${count}` `פעמים`;
  });
}
```

```

return (
  <div>
    <p>{count}</p>
    <button onClick={() => setCount(count + 1)}>
      לחץ עליו
    </button>
  </div>
);
}

```

זהה, אנו אומרים לריקט שהkomponennta שלך צריכה לעשות משהו לאחר Hook-**עשה**? על ידי שימוש ב **useEffect** מה-**ב**. DOM-ונקרא לה לאחר ביצוע עדכוני ה-**state**, (שלנו effect -נתיחס אל זה כה) הרינדור. ריאקט תזקוף את הפונקציה שהעברת effect הכרחי אחר API -זהה, אנו קובעים את כותרת העמוד, אבל נוכל בנוסף לבצע אחזור מידע או לקרוא ל-

-בתוך הקומפוננטה נתנת לנו גישה למשתנה **useEffect count** -מודע קוראים ל **state** scope -מיוחד על מנת לקרוא אותו -זה כבר בתוך ה API אנו לא צריכים-effect. ישר מה (אחר prop או כל-) closures-מננים מהציג APIs וنمנים מהתמסים ב Hooks. נווטנת פתרון JavaScript-ספקטיבים של ריאקט היכן ש-

רץ אחרי כל רינדור? כן! כבירית מחדל, זה רץ גם אחרי הרינדור הראשון וגם אחרי כל עדכון. (נדבר אחר כך **useEffect האם עדכון**, יכול להיות שהיא לך יותר קל לחשב ש "mounting" על **להתאים אישית את זה**). במקומות חשובים של effects-עדכון כאשר זה מרים את ה DOM-effects. ריאקט מבטיח שה-

הסבר מפורט

השורות הבאות אמורות להיות מוגנות effects, בעצם כשהאנו יודעים עד על:

```

function Example() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    document.title = `${count} פעמים לחץ`;
  });
}

```

-נעביר פונקציה ל effect -ואז אנו אומרים לריקט שאנו צריכים להשתמש ב **state count** אנו מצהירים על המשתנה **useEffect** -שלנו, נקבע את כותרת העמוד על ידי שימוש ב effect -שלנו. בתוך ה effect-הfonktsia שאנו מעבירים היא ה **document.title** -של הדף. נוכל לקרוא את ה API-**count** -שהוא חלק מה **useEffect** -האהרון בתוך ה **useEffect** -של ה Fonktsia. כשייקט מרדדר את הקומפוננטה שלנו, הוא יזכיר את ה **useEffect** -שבו השתמשנו, ואז יירץ את effect -של ה Fonktsia שלנו. כשייקט מרדדר את הקומפוננטה שלנו, הוא יזכיר את ה **useEffect** -של רינדור, כולל הרាជון. לאחר שיעדכו את ה **useEffect** -ההאהרון את ה **useEffect** -של רינדור, כולל הרាជון.

הולכת להיות שונה לכל רינדור. זה נעשה **useEffect** -מנוסים אולי ישימו לב שהfonktsia שמועברת ל **useEffect** מחייב לדאוג שהוא לא יהיה עדכני. כל זמן שאנו(effect count) בכוונה. למעשה, זה מה שנותן לנו לקרוא את הערך להתנהג יותר כמו חלק effects -אחד, שמחlijף את קודמו. במידה מסוימת, זה גורם ל effect מרדדים מחדש, אנחנו מתזמינים לשיק"ר רינדור מסוים. נראה יותר ברור למה זה שימושי **בהמשך העמוד**" effect מותצתת הרינדור - כל

טיפ

לא חסמים `useEffect` שמתואמנים עם `componentDidUpdate`, `effects` או `componentDidMount` -בניגוד ל-

לא צריכים להתקיים `effects` -את הדףן מלעדכן את המסק. זה גורם לישום שלך להרגיש יותר וספונטי. הרוב ה-

במקרים הנדרים שהם כן מתקיימים בסינכרוניות `useLayoutEffect` יש (כמו מדידת ה `layout`) נפרד עם `useEffect` -מקביל ל `useEffect`.

עם ניקוי Effects

למקור מידע חיצוני כלשהו. במקרה זה, **subscription** שכון דורשים. לדוגמא, **אולי נרצה להציגו** effects למורות זאת, ישנו Hook -זה חשוב לנו על מנת שלא תהיה דיליפת זיכרון! נשווה איך ניתן לעשות זאת עם מחלקות ו-

דוגמה עם שימוש במחלקות

ומנקים אותו בתוך `componentDidMount`, `componentWillUnmount` בתוכה `subscription` במחלקה ריאקט, בדרך כלל מגדירים לסתוטוס חיבור של חבר `subscribe` שנตอนן לנו לעשות API `ChatAPI`, נגד שיש לנו מודול. להציגו על ידי שימוש במחלקה אונליין. הנה איך נוכל לעשות:

```
class FriendStatus extends React.Component {
  constructor(props) {
    super(props);
    this.state = { isOnline: null };
    this.handleStatusChange = this.handleStatusChange.bind(this);
  }

  componentDidMount() {
    ChatAPI.subscribeToFriendStatus(
      this.props.friend.id,
      this.handleStatusChange
    );
  }

  componentWillUnmount() {
    ChatAPI.unsubscribeFromFriendStatus(
      this.props.friend.id,
      this.handleStatusChange
    );
  }

  handleStatusChange(status) {
    this.setState({
      isOnline: status.isOnline
    });
  }

  render() {
    if (this.state.isOnline === null) {
      return 'טען...';
    }
    return this.state.isOnline ? 'מחובר' : 'לא מחובר';
  }
}
```

צריכים לשחק אחד את השני. מתחות מחזורי חיים `componentDidMount` ו- `componentWillUnmount` שים לב איך מחרחות אותןנו לפצל את הלוגיקה למורות שבאופן מסווגי הקוד בשניות קשור לאלו.

הערה

כדי להיות נכון לשלוטין. `componentDidUpdate` קוראים חד-עין ישימו לב שדוגמה זאת צריכה בנוסח מתודת `נתעלם מכך אבל נחזור להה` בחלק אחר של העמוד.

דוגמה עם שימוש ב Hooks

הבה נראה איך נוכל לכתוב את קומפוננטה זו בשימוש של Hooks.

subscription נפרד על מנת לבצע את הnickio. אולם קוד שמיועד להוספה ולהסרה של effect יכול להיות שאתה חושב שנוצרת שלק מחזיר פונקציה, ריאקט יירץ אותה כשייגע הזמן-effect-מתוכנן להשאיר אותו ביחד. אם ה `useEffect` -כל כך קשור שלנקות:

```
import React, { useState, useEffect } from 'react';

function FriendStatus(props) {
  const [isOnline, setIsOnline] = useState(null);

  useEffect(() => {
    function handleStatusChange(status) {
      setIsOnline(status.isOnline);
    }

    ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
    // ציינו כאן איך לנתקות אחוריו ה-effect:
    return function cleanup() {
      ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
    };
  });

  if (isOnline === null) {
    return 'לול...';
  }
  return isOnline ? 'מחובר' : 'לא מחובר';
}
```

יכול להחזיר פונקציה שמנקה effect כל -**שלנו**? זה מנגנון ניקוי אופציוניAli ל-**effect** -**מדוע** החזרנו פונקציה מה **subscriptions**?

של קומפוננטה. למרות זאת, כפי Chaitin-Gomberg ריאקט מבצעת את הnickion בזמן? **effect**? מתי בדיקת ריאקט מנקה אחריו מרינדרוים effects ריצים אחריו כל רינדרו ולא רק פעם אחת. זאת הסיבה שריאקט מנקה גם effects, שלמדונו קודם בפעם הבאה. נדבר על **למה** זה עוזר לנו להימנע מABAGs ו**וגם איך להימנע מהתנהגות זו** במקרה זה יוצר effects לפני הרצת בעיות מבחינת ביצועים. בהמשך

הערה

פה כדי להבהיר את מטרתה, אך תוכל `cleanup` זה. קראונו לה `effect` - אלו לא צריכים להחזיר פונקציה עם שם מואיל. לדוגמה לה בשם אחר `arrow function` להחזיר

סיכום

מסויימים **effects** - נתונים לנו להביע סוגים שונים של תופעות לוואי לאחר רינדור קומפוננטה. בגלל ש `useEffect` - למדנו ש דורשים ניקיון הם מחזירים פונקציה:

```
useEffect(() => {
  function handleStatusChange(status) {
    setIsOnline(status.isOnline);
  }

  ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
  return () => {
    ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
  };
});
```

אחרים לא ידרשו שלב ניקיון, מכאן שלא יחזירו כלום

```
useEffect(() => {
  document.title = `${count} פעמים`;
});
```

אחד API - מהאחד בו את שני מקרי השימוש ב Effect Hook.

עובד, או אם אתה מרגיש שזה יותר מדי מידע, תוכל **Effect Hook** אם אתה מרגיש שיש לך תפיסה טובה של איך עכשו [לעבור לעמוד הבא שמדובר על חוקי Hooks](#).

טיפים לשימוש ב Effects

שיכולים לעניין משתמשי ריאקט מנוסים. אין חובה `useEffect` נמשיך את עמוד זה עם מבט עמוק על כמה מההיבטים של העומק בHOOK. תוכל תמיד לחזור לעמוד זה על מנת ללמוד עוד פרטים על:

לביעות נפרדות effects טיפ: השימוש במספר

היא שמחלות מתחות מחזורי חיים מכילות לוגיקה לא קשורה, אך Hooks - אחת מהבעיות שפירענו עליה [במוטיבציה](#) לדוגמת בדיקת הסטטוס חבר אונליין counter-לוגיקה קשורה מפוצלת לכמה מתחות. הנה קומפוננטה שמחברת את לוגיקת הלקוחות הקודומות:

```
class FriendStatusWithCounter extends React.Component {
  constructor(props) {
    super(props);
    this.state = { count: 0, isOnline: null };
    this.handleStatusChange = this.handleStatusChange.bind(this);
  }

  componentDidMount() {
    document.title = `${this.state.count} פעמים`;
    ChatAPI.subscribeToFriendStatus(
      this.props.friend.id,
      this.handleStatusChange
    );
  }
}
```

```

componentDidUpdate() {
  document.title = `לחץ ${this.state.count}`; `פתרונות`;
}

componentWillUnmount() {
  ChatAPI.unsubscribeFromFriendStatus(
    this.props.friend.id,
    this.handleStatusChange
  );
}

handleStatusChange(status) {
  this.setState({
    isOnline: status.isOnline
  });
}
// ...

```

שים לב איך הלוגיקה שגדירה את מפוצלת בין `componentDidMount` ו- `componentDidUpdate`. לוגיקת הסטוס גם היא מפוצלת בין `componentDidMount` ו- `componentWillUnmount`. מילה קוד לשתי המטלות מכילה קוד לשתי המטלות.

יותר מפעם אחת, ניתן להשתמש במספר [State Hook](#) או, איך שונאים effects - זה נותן לנו לפצל לוגיקה לא קשורה ל:

```

function FriendStatusWithCounter(props) {
  const [count, setCount] = useState(0);
  useEffect(() => {
    document.title = `${count}`; `פתרונות`;
  });

  const [isOnline, setIsOnline] = useState(null);
  useEffect(() => {
    function handleStatusChange(status) {
      setIsOnline(status.isOnline);
    }
    ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
    return () => {
      ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
    };
  });
}
// ...

```

Hooks נותנים לנו לפצל קוד בהתאם על מה שהוא עושה ולא בהתבסס על שם מתודת מחזור חיים. ריאקט ייחיל כל שמשומש על ידי הקומפוננטה, בסדר שהן צוינו.

רצים על כל עדכון effects הסביר: למה

קורה לאחר רינדור חדש, ולא רק פעם אחת-effect-אם אתה רגיל למחלקות, יכול להיות שאתה תזהה למה שלב ניקוי הiba נראה דוגמה פרקטית שסבירה למה תכננו זה עוזר לנו ליצור קומפוננטות עם פחות אגמים.

שמציגה אם חבר מחובר או לא. המחלוקת שלנו קוראת את `FriendStatus` מוקדם יותר בעמוד זה, הציגנו את הקומפוננטה של הקומפוננטה, ומتنתקת בזמן mounting-מתחברת לסטטוס החבר לאחר ה- `this.props` - מ unmounting:

```

componentDidMount() {
  ChatAPI.subscribeToFriendStatus(
    this.props.friend.id,
    this.handleStatusChange
  );
}

componentWillUnmount() {
  ChatAPI.unsubscribeFromFriendStatus(
    this.props.friend.id,
    this.handleStatusChange
  );
}

```

משתנה בזמן שהkomponenntה על המסך? הקומפוננטה שלנו תמשיך להציג את הסטטוס **friend prop** - אבל מה קורה אם ה`this.props.friend.id` - בגלל שקריאת התנתקות משתמש ב `unmounting` של חבר אחר. זה באג. נגרום גם לדיליפט זיכרון או קריישן בזמן של חבר.

לטפל במקרה זה `componentDidUpdate` בקומפוננטת מחלוקת, נctrיך גם להווסף:

```

componentDidMount() {
  ChatAPI.subscribeToFriendStatus(
    this.props.friend.id,
    this.handleStatusChange
  );
}

componentDidUpdate(prevProps) {
  // הקודם מה התנתק friend.id
  ChatAPI.unsubscribeFromFriendStatus(
    prevProps.friend.id,
    this.handleStatusChange
  );
  // החביב ל friend.id
  ChatAPI.subscribeToFriendStatus(
    this.props.friend.id,
    this.handleStatusChange
  );
}

componentWillUnmount() {
  ChatAPI.unsubscribeFromFriendStatus(
    this.props.friend.id,
    this.handleStatusChange
  );
}

```

בישומי ריאקט - מקור נפוץ של באגים הוא מפתחים שעוכחים לטפל ב `componentDidUpdate`.

- עכשו נשלול את הגרסה הזו של הקומפוננטה שמשתמשת ב Hooks:

```

function FriendStatus(props) {
  // ...
  useEffect(() => {
    // ...
    ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
    return () => {

```

```
    ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
  };
};
```

היא לא סובלת מבאג זה. (אבל לא שינויו אותה גם).

הקודמים לפניו-effects-מטפלת בהם כבירות מחדל. היא מנקה את ה `useEffect` - אין קוד מיוחד לטיפול בעדכניםים בגלל ש הבאים. על מנת להדגים זאת, הנה רצף של קריאות התחרבות והتانקות שהקומפוננטה יכולה ליצור לארוך-effects-הshipment הזמן:

```
ChatAPI.subscribeToFriendStatus(100, handleStatusChange); // ראשוני הרצת effect
ChatAPI.unsubscribeFromFriendStatus(100, handleStatusChange); // קודם ניקוי effect
ChatAPI.subscribeToFriendStatus(200, handleStatusChange); // הבא-effect הרצת ה
ChatAPI.unsubscribeFromFriendStatus(200, handleStatusChange); // קודם ניקוי effect
ChatAPI.subscribeToFriendStatus(300, handleStatusChange); // הבא-effect הרצת ה
// Unmount
ChatAPI.unsubscribeFromFriendStatus(300, handleStatusChange); // ניקוי ה effect
האזור
```

התנהגות זאת מבטיחה עקבות כבירות מחדל ומונעת באגים נפוצים בקומפוננטות מחלוקת בגלל לוגיקת עדכון חסירה.

טיפ: מיטוב ביצועים על ידי דילוג על effects

לאחר כל רינדור יכול ליצור בעיות ביצועים. בקומפוננטות מחלוקת, נוכל לפחות effect במקדים מסוימים, ניקיון או השמת prevProps או prevState בתוך `componentDidUpdate`:

```
componentDidUpdate(prevProps, prevState) {
  if (prevState.count !== this.state.count) {
    document.title = `${this.state.count}`; `פעמים
  }
}
```

אם ערכים effect אתה יכול לומר לריאקט לדלג על השמת `useEffect`. דרישת זו היא נפוצה מספיק שהיא כבר בנויות לתוך מסויימים לא השתנו בין רינדרומים. על מנת לעשות זאת, העבר מערך ארגומנט שני-opzionali ל- `useEffect`:

```
useEffect(() => {
  document.title = `${count}`; `פעמים
}, [count]); // אם משתנה count אמ' הרץ מחדש את ה
```

שווה ל- 5, אז הקומפוננטה שלנו `count` כארוגמנט שני. מה זה אומר? אם `[count]` בדוגמה לעיל, אנו מعتبرים שעדין שווה ל- 5, ריאקט תשווה את `[5]` מהרינדור הקודם ו- `[5]` מהרינדור הבא. בגלל `count` מתורנחתה מחדש עם זהה האופטימיזציה שלנו. שככל הערכים במערך הם אותו `(5 === 5)`, ריאקט ידלג על ה

שמעודכו ל- 6, ריאקט תשווה את הערכים בתוך מערך ה- `[5]` מהרינדור הקודם לערכים `count` כאשרנו מרדכנים עם בגלל ש- `5 !== 6`. אם ישנים ערכים נוספים-effect-במערך ה- `[6]` בrinndor הבא. במקרה הזה, ריאקט ישים מחדש את ה-

מחדש גם אם אחד מהם שונה-effect-במערך, ריאקט ייריץ את ה-

:בעלי שלב ניקיון effects זה גם עובד על

```

useEffect(() => {
  function handleStatusChange(status) {
    setIsOnline(status.isOnline);
  }

  ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
  return () => {
    ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
  };
}, [props.friend.id]); // משנה props.friend.id התחבר מחדש רק אם

```

בעתיד, יכול להיות שהารוגומנט השני יוסיף אוטומטיות על ידי טרנספורמציה בזמן בניה (build-time).

הערה

-ו props כמו של הקומפוננטה scope - אם אתה משתמש באופטימיזציה זו, ודא שהמערך מכיל את כל הערכים מה אחרת, הקוד שולץ יתיחס לערכים יפנים מרינדרויים **effect** -**משתנים לאורך זמן ומשמעותם על ידי ה state**. קודמים. למד עוד על [איך לטפל בפונקציות ומה לעשות כשהמערך משתנה בתדריות גבוהה מדי](#).

תוכל להעביר מערך ריק ([]) ולנקות אותו רק פעם אחת effect אם אתה רוצה להריץ זאת הוא לא צריך לזרוץ מחדש. או props state או שולץ לא תלוי בשום ערך מ-effect -ארוגומנט שני. זה אומר לריקט שהוא תמיד עובד על ידי מקרה מיוחד – הדבר ממשיק ישירות מאייך מערכ ה-dependencies.

תמיד יכולו את הערכים ההתחלתיים שלהם. בזמן state-effect-בתוך ה state -ו props-אם אתה מעביר מערך ריק ([]), ה componentDidMount -ו componentWillUnmount השערת ([]) הארוגומנט שני היא קרובה יותר למודלים המוכרים מהר מדי. בנוסף, אל תשכח effects שנסמ בדרך כלל [פתרונות טובים יותר](#) על מנת להימנע מהרצאה מחדש שלount. אז ביצוע עבודה נוספת היא לא בעיה, עד לאחר שחדפסן "ציר" את ה useEffect שריקט דוחה את הרצת

זה מזהיר eslint-plugin-react-hooks -[מחלק מחבילת exhaustive-deps](#) אנו ממליצים על שימוש בחוק מצוינים באופן לא נכון ומציעה פתרון dependencies -[כש](#).

השלבים הבאים

ולמדת גם על ה effects-state hook. למדת גם על ה effects hook. מזל טוב! זה היה עמוד ארוך, אבל אנו מקווים שבஸפו רוב השאלה שולץ בקשר ל שיש עוד הרבה שניתן לעשות עם שנייהם ביחד. הם מכיסים את רוב מקרי השימוש במחלקות – והיכן שהם, וגם על [נוספים שימושי Hooks](#) לא, יכול להיות שתמצא את.

מןעו שכפול ב effect פותרים בעיות שהוצגו במוטיבציה. ראיינו איך ניקיון Hooks אנו גם מתחילה לראות איך מביא קוד הקשור קרוב יותר, ועזרה לנו להימנע מbagים. ראיינו גם, componentDidUpdate -ו componentWillUnmount, על פי מטרתם, זהה מהهو שלא יכולנו לעשות במחלקות כלל effects איך ניתן לפצל

מתאים לאייה useState [עובדים](#). איך ריקט יודעת איזה קראית Hooks בנקודה זו יכול להיות שאתה עדין תוהה איך קודמים והבאים על כל עדכון? **בעמוד הבא נלמד עוד על effects** "בין רינדרויים? איך ריקט" מתאימה state משתנה [Rules of Hooks](#) -**הם חיווניים על מנת לגרום ל --**

[Go to TOC](#)

Hooks FAQ

Hooks are a new addition in React 16.8. They let you use state and other React features without writing a class.

This page answers some of the frequently asked questions about [Hooks](#).

- **Adoption Strategy**

- Which versions of React include Hooks?
- Do I need to rewrite all my class components?
- What can I do with Hooks that I couldn't with classes?
- How much of my React knowledge stays relevant?
- Should I use Hooks, classes, or a mix of both?
- Do Hooks cover all use cases for classes?
- Do Hooks replace render props and higher-order components?
- What do Hooks mean for popular APIs like Redux connect() and React Router?
- Do Hooks work with static typing?
- How to test components that use Hooks?
- What exactly do the lint rules enforce?

- **From Classes to Hooks**

- How do lifecycle methods correspond to Hooks?
- How can I do data fetching with Hooks?
- Is there something like instance variables?
- Should I use one or many state variables?
- Can I run an effect only on updates?
- How to get the previous props or state?
- Why am I seeing stale props or state inside my function?
- How do I implement getDerivedStateFromProps?
- Is there something like forceUpdate?
- Can I make a ref to a function component?
- How can I measure a DOM node?
- What does const [thing, setThing] = useState() mean?

- **Performance Optimizations**

- Can I skip an effect on updates?
- Is it safe to omit functions from the list of dependencies?
- What can I do if my effect dependencies change too often?
- How do I implement shouldComponentUpdate?
- How to memoize calculations?
- How to create expensive objects lazily?
- Are Hooks slow because of creating functions in render?
- How to avoid passing callbacks down?
- How to read an often-changing value from useCallback?

- [Under the Hood](#)
 - How does React associate Hook calls with components?
 - What is the prior art for Hooks?

Adoption Strategy

Which versions of React include Hooks?

Starting with 16.8.0, React includes a stable implementation of React Hooks for:

- React DOM
- React Native
- React DOM Server
- React Test Renderer
- React Shallow Renderer

Note that **to enable Hooks, all React packages need to be 16.8.0 or higher**. Hooks won't work if you forget to update, for example, React DOM.

[React Native 0.59](#) and above support Hooks.

Do I need to rewrite all my class components?

No. There are [no plans](#) to remove classes from React -- we all need to keep shipping products and can't afford rewrites. We recommend trying Hooks in new code.

What can I do with Hooks that I couldn't with classes?

Hooks offer a powerful and expressive new way to reuse functionality between components. "[Building Your Own Hooks](#)" provides a glimpse of what's possible. [This article](#) by a React core team member dives deeper into the new capabilities unlocked by Hooks.

How much of my React knowledge stays relevant?

Hooks are a more direct way to use the React features you already know -- such as state, lifecycle, context, and refs. They don't fundamentally change how React works, and your knowledge of components, props, and top-down data flow is just as relevant.

Hooks do have a learning curve of their own. If there's something missing in this documentation, [raise an issue](#) and we'll try to help.

Should I use Hooks, classes, or a mix of both?

When you're ready, we'd encourage you to start trying Hooks in new components you write. Make sure everyone on your team is on board with using them and familiar with this documentation. We don't recommend rewriting your existing classes to Hooks unless you planned to rewrite them anyway (e.g. to fix bugs).

You can't use Hooks *inside* a class component, but you can definitely mix classes and function components with Hooks in a single tree. Whether a component is a class or a function that uses Hooks is an implementation detail of that component. In the longer term, we expect Hooks to be the primary way people write React components.

Do Hooks cover all use cases for classes?

Our goal is for Hooks to cover all use cases for classes as soon as possible. There are no Hook equivalents to the uncommon `getSnapshotBeforeUpdate`, `getDerivedStateFromError` and `componentDidCatch` lifecycles yet, but we plan to add them soon.

Do Hooks replace render props and higher-order components?

Often, render props and higher-order components render only a single child. We think Hooks are a simpler way to serve this use case. There is still a place for both patterns (for example, a virtual scroller component might have a `renderItem` prop, or a visual container component might have its own DOM structure). But in most cases, Hooks will be sufficient and can help reduce nesting in your tree.

What do Hooks mean for popular APIs like Redux `connect()` and React Router?

You can continue to use the exact same APIs as you always have; they'll continue to work.

React Redux since v7.1.0 [supports Hooks API](#) and exposes hooks like `useDispatch` or `useSelector`.

React Router [supports hooks](#) since v5.1.

Other libraries might support hooks in the future too.

Do Hooks work with static typing?

Hooks were designed with static typing in mind. Because they're functions, they are easier to type correctly than patterns like higher-order components. The latest Flow and TypeScript React definitions include support for React Hooks.

Importantly, custom Hooks give you the power to constrain React API if you'd like to type them more strictly in some way. React gives you the primitives, but you can combine them in different ways than what we provide out of the box.

How to test components that use Hooks?

From React's point of view, a component using Hooks is just a regular component. If your testing solution doesn't rely on React internals, testing components with Hooks shouldn't be different from how you normally test components.

Note

[Testing Recipes](#) include many examples that you can copy and paste.

For example, let's say we have this counter component:

```
function Example() {
  const [count, setCount] = useState(0);
  useEffect(() => {
    document.title = `You clicked ${count} times`;
  });
  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

We'll test it using React DOM. To make sure that the behavior matches what happens in the browser, we'll wrap the code rendering and updating it into `ReactTestUtils.act()` calls:

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import { act } from 'react-dom/test-utils';
import Counter from './Counter';

let container;

beforeEach(() => {
  container = document.createElement('div');
  document.body.appendChild(container);
});

afterEach(() => {
  document.body.removeChild(container);
  container = null;
});

it('can render and update a counter', () => {
  // Test first render and effect
  act(() => {
    ReactDOM.createRoot(container).render(<Counter />);
  });
  const button = container.querySelector('button');
  const label = container.querySelector('p');
  expect(label.textContent).toBe('You clicked 0 times');
  expect(document.title).toBe('You clicked 0 times');

  // Test second render and effect
  act(() => {
    button.dispatchEvent(new MouseEvent('click', {bubbles: true}));
  });
  expect(label.textContent).toBe('You clicked 1 times');
  expect(document.title).toBe('You clicked 1 times');
});
```

The calls to `act()` will also flush the effects inside of them.

If you need to test a custom Hook, you can do so by creating a component in your test, and using your Hook from it. Then you can test the component you wrote.

To reduce the boilerplate, we recommend using [React Testing Library](#) which is designed to encourage writing tests that use your components as the end users do.

For more information, check out [Testing Recipes](#).

What exactly do the [lint rules](#) enforce?

We provide an [ESLint plugin](#) that enforces [rules of Hooks](#) to avoid bugs. It assumes that any function starting with "`use`" and a capital letter right after it is a Hook. We recognize this heuristic isn't perfect and there may be some false positives, but without an ecosystem-wide convention there is just no way to make Hooks work well -- and longer names will discourage people from either adopting Hooks or following the convention.

In particular, the rule enforces that:

- Calls to Hooks are either inside a `PascalCase` function (assumed to be a component) or another `useSomething` function (assumed to be a custom Hook).
- Hooks are called in the same order on every render.

There are a few more heuristics, and they might change over time as we fine-tune the rule to balance finding bugs with avoiding false positives.

From Classes to Hooks

How do lifecycle methods correspond to Hooks?

- `constructor`: Function components don't need a constructor. You can initialize the state in the `useState` call. If computing the initial state is expensive, you can pass a function to `useState`.
- `getDerivedStateFromProps`: Schedule an update [while rendering](#) instead.
- `shouldComponentUpdate`: See [React.memo](#) [below](#).
- `render`: This is the function component body itself.
- `componentDidMount`, `componentDidUpdate`, `componentWillUnmount`: The `useEffect` [Hook](#) can express all combinations of these (including [less common](#) cases).
- `getSnapshotBeforeUpdate`, `componentDidCatch` and `getDerivedStateFromError`: There are no Hook equivalents for these methods yet, but they will be added soon.

How can I do data fetching with Hooks?

Here is a [small demo](#) to get you started. To learn more, check out [this article](#) about data fetching with Hooks.

Is there something like instance variables?

Yes! The `useRef()` Hook isn't just for DOM refs. The "ref" object is a generic container whose `current` property is mutable and can hold any value, similar to an instance property on a class.

You can write to it from inside `useEffect`:

```
function Timer() {
  const intervalRef = useRef();

  useEffect(() => {
    const id = setInterval(() => {
      // ...
    });
    intervalRef.current = id;
    return () => {
      clearInterval(intervalRef.current);
    };
  });
  // ...
}
```

If we just wanted to set an interval, we wouldn't need the ref (`id` could be local to the effect), but it's useful if we want to clear the interval from an event handler:

```
// ...
function handleCancelClick() {
  clearInterval(intervalRef.current);
}
// ...
```

Conceptually, you can think of refs as similar to instance variables in a class. Unless you're doing [lazy initialization](#), avoid setting refs during rendering -- this can lead to surprising behavior. Instead, typically you want to modify refs in event handlers and effects.

Should I use one or many state variables?

If you're coming from classes, you might be tempted to always call `useState()` once and put all state into a single object. You can do it if you'd like. Here is an example of a component that follows the mouse movement. We keep its position and size in the local state:

```
function Box() {
  const [state, setState] = useState({ left: 0, top: 0, width: 100, height: 100 });
}
// ...
```

Now let's say we want to write some logic that changes `left` and `top` when the user moves their mouse. Note how we have to merge these fields into the previous state object manually:

```
// ...
useEffect(() => {
  function handleWindowMouseMove(e) {
    // Spreading "...state" ensures we don't "lose" width and height
```

```

        setState(state => ({ ...state, left: e.pageX, top: e.pageY }));
    }
    // Note: this implementation is a bit simplified
    window.addEventListener('mousemove', handleWindowMouseMove);
    return () => window.removeEventListener('mousemove', handleWindowMouseMove);
}, []);
// ...

```

This is because when we update a state variable, we *replace* its value. This is different from `this.setState` in a class, which *merges* the updated fields into the object.

If you miss automatic merging, you could write a custom `useLegacyState` Hook that merges object state updates. However, **we recommend to split state into multiple state variables based on which values tend to change together.**

For example, we could split our component state into `position` and `size` objects, and always replace the `position` with no need for merging:

```

function Box() {
  const [position, setPosition] = useState({ left: 0, top: 0 });
  const [size, setSize] = useState({ width: 100, height: 100 });

  useEffect(() => {
    function handleWindowMouseMove(e) {
      setPosition({ left: e.pageX, top: e.pageY });
    }
    // ...
  })
}

```

Separating independent state variables also has another benefit. It makes it easy to later extract some related logic into a custom Hook, for example:

```

function Box() {
  const position = useWindowPosition();
  const [size, setSize] = useState({ width: 100, height: 100 });
  // ...
}

function useWindowPosition() {
  const [position, setPosition] = useState({ left: 0, top: 0 });
  useEffect(() => {
    // ...
  }, []);
  return position;
}

```

Note how we were able to move the `useState` call for the `position` state variable and the related effect into a custom Hook without changing their code. If all state was in a single object, extracting it would be more difficult.

Both putting all state in a single `useState` call, and having a `useState` call per each field can work. Components tend to be most readable when you find a balance between these two extremes, and group related state into a few independent state variables. If the state logic becomes complex, we recommend [managing it with a reducer](#) or a custom Hook.

Can I run an effect only on updates?

This is a rare use case. If you need it, you can [use a mutable ref](#) to manually store a boolean value corresponding to whether you are on the first or a subsequent render, then check that flag in your effect. (If you find yourself doing this often, you could create a custom Hook for it.)

How to get the previous props or state?

Currently, you can do it manually [with a ref](#):

```
function Counter() {
  const [count, setCount] = useState(0);

  const prevCountRef = useRef();
  useEffect(() => {
    prevCountRef.current = count;
  });
  const prevCount = prevCountRef.current;

  return <h1>Now: {count}, before: {prevCount}</h1>;
}
```

This might be a bit convoluted but you can extract it into a custom Hook:

```
function Counter() {
  const [count, setCount] = useState(0);
  const prevCount = usePrevious(count);
  return <h1>Now: {count}, before: {prevCount}</h1>;
}

function usePrevious(value) {
  const ref = useRef();
  useEffect(() => {
    ref.current = value;
  });
  return ref.current;
}
```

Note how this would work for props, state, or any other calculated value.

```
function Counter() {
  const [count, setCount] = useState(0);

  const calculation = count + 100;
  const prevCalculation = usePrevious(calculation);
  // ...
```

It's possible that in the future React will provide a `usePrevious` Hook out of the box since it's a relatively common use case.

See also [the recommended pattern for derived state](#).

Why am I seeing stale props or state inside my function?

Any function inside a component, including event handlers and effects, "sees" the props and state from the render it was created in. For example, consider code like this:

```
function Example() {
  const [count, setCount] = useState(0);

  function handleAlertClick() {
    setTimeout(() => {
      alert('You clicked on: ' + count);
    }, 3000);
  }

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
      <button onClick={handleAlertClick}>
        Show alert
      </button>
    </div>
  );
}
```

If you first click "Show alert" and then increment the counter, the alert will show the `count` variable **at the time you clicked the "Show alert" button**. This prevents bugs caused by the code assuming props and state don't change.

If you intentionally want to read the *latest* state from some asynchronous callback, you could keep it in a [ref](#), mutate it, and read from it.

Finally, another possible reason you're seeing stale props or state is if you use the "dependency array" optimization but didn't correctly specify all the dependencies. For example, if an effect specifies `[]` as the second argument but reads `someProp` inside, it will keep "seeing" the initial value of `someProp`. The solution is to either remove the dependency array, or to fix it. Here's [how you can deal with functions](#), and here's [other common strategies](#) to run effects less often without incorrectly skipping dependencies.

Note

We provide an `exhaustive-deps` ESLint rule as a part of the `eslint-plugin-react-hooks` package. It warns when dependencies are specified incorrectly and suggests a fix.

How do I implement `getDerivedStateFromProps`?

While you probably [don't need it](#), in rare cases that you do (such as implementing a `<Transition>` component), you can update the state right during rendering. React will re-run the component with updated state immediately after exiting the first render so it wouldn't be expensive.

Here, we store the previous value of the `row` prop in a state variable so that we can compare:

```
function ScrollView({row}) {
  const [isScrollingDown, setIsScrollingDown] = useState(false);
  const [prevRow, setPrevRow] = useState(null);

  if (row !== prevRow) {
    // Row changed since last render. Update isScrollingDown.
    setIsScrollingDown(prevRow !== null && row > prevRow);
    setPrevRow(row);
  }

  return `Scrolling down: ${isScrollingDown}`;
}
```

This might look strange at first, but an update during rendering is exactly what `getDerivedStateFromProps` has always been like conceptually.

Is there something like `forceUpdate`?

Both `useState` and `useReducer` Hooks [bail out of updates](#) if the next value is the same as the previous one. Mutating state in place and calling `setState` will not cause a re-render.

Normally, you shouldn't mutate local state in React. However, as an escape hatch, you can use an incrementing counter to force a re-render even if the state has not changed:

```
const [ignored, forceUpdate] = useReducer(x => x + 1, 0);

function handleClick() {
  forceUpdate();
}
```

Try to avoid this pattern if possible.

Can I make a ref to a function component?

While you shouldn't need this often, you may expose some imperative methods to a parent component with the `useImperativeHandle` Hook.

How can I measure a DOM node?

One rudimentary way to measure the position or size of a DOM node is to use a `callback ref`. React will call that callback whenever the ref gets attached to a different node. Here is a [small demo](#):

```
function MeasureExample() {
  const [height, setHeight] = useState(0);

  const measuredRef = useCallback(node => {
    if (node !== null) {
      setHeight(node.getBoundingClientRect().height);
    }
  }, []);

  return (
    <>
  );
}
```

```

        <h1 ref={measuredRef}>Hello, world</h1>
        <h2>The above header is {Math.round(height)}px tall</h2>
      </>
    );
}

```

We didn't choose `useRef` in this example because an object ref doesn't notify us about *changes* to the current ref value. Using a callback ref ensures that even if a child component displays the measured node later (e.g. in response to a click), we still get notified about it in the parent component and can update the measurements.

Note that we pass `[]` as a dependency array to `useCallback`. This ensures that our ref callback doesn't change between the re-renders, and so React won't call it unnecessarily.

In this example, the callback ref will be called only when the component mounts and unmounts, since the rendered `<h1>` component stays present throughout any rerenders. If you want to be notified any time a component resizes, you may want to use `ResizeObserver` or a third-party Hook built on it.

If you want, you can extract this logic into a reusable Hook:

```

function MeasureExample() {
  const [rect, ref] = useClientRect();
  return (
    <>
      <h1 ref={ref}>Hello, world</h1>
      {rect !== null &&
        <h2>The above header is {Math.round(rect.height)}px tall</h2>
      }
    </>
  );
}

function useClientRect() {
  const [rect, setRect] = useState(null);
  const ref = useCallback(node => {
    if (node !== null) {
      setRect(node.getBoundingClientRect());
    }
  }, []);
  return [rect, ref];
}

```

What does `const [thing, setThing] = useState()` mean?

If you're not familiar with this syntax, check out the [explanation](#) in the State Hook documentation.

Performance Optimizations

Can I skip an effect on updates?

Yes. See [conditionally firing an effect](#). Note that forgetting to handle updates often [introduces bugs](#), which is why this isn't the default behavior.

Is it safe to omit functions from the list of dependencies?

Generally speaking, no.

```
function Example({ someProp }) {
  function doSomething() {
    console.log(someProp);
  }

  useEffect(() => {
    doSomething();
  }, []); // 🚫 This is not safe (it calls `doSomething` which uses `someProp`)
}
```

It's difficult to remember which props or state are used by functions outside of the effect. This is why **usually you'll want to declare functions needed by an effect *inside* of it**. Then it's easy to see what values from the component scope that effect depends on:

```
function Example({ someProp }) {
  useEffect(() => {
    function doSomething() {
      console.log(someProp);
    }

    doSomething();
  }, [someProp]); // ✅ OK (our effect only uses `someProp`)
}
```

If after that we still don't use any values from the component scope, it's safe to specify `[]`:

```
useEffect(() => {
  function doSomething() {
    console.log('hello');
  }

  doSomething();
}, []); // ✅ OK in this example because we don't use *any* values from component
        scope
```

Depending on your use case, there are a few more options described below.

Note

We provide the `exhaustive-deps` ESLint rule as a part of the `eslint-plugin-react-hooks` package. It helps you find components that don't handle updates consistently.

Let's see why this matters.

If you specify a [list of dependencies](#) as the last argument to `useEffect`, `useLayoutEffect`, `useMemo`, `useCallback`, or `useImperativeHandle`, it must include all values that are used inside the callback and participate in the React data flow. That includes props, state, and anything derived from them.

It is **only** safe to omit a function from the dependency list if nothing in it (or the functions called by it) references props, state, or values derived from them. This example has a bug:

```
function ProductPage({ productId }) {
  const [product, setProduct] = useState(null);

  async function fetchProduct() {
    const response = await fetch('http://myapi/product/' + productId); // Uses
    productId prop
    const json = await response.json();
    setProduct(json);
  }

  useEffect(() => {
    fetchProduct();
  }, []); // 🚫 Invalid because `fetchProduct` uses `productId`
  // ...
}
```

The recommended fix is to move that function *inside* of your effect. That makes it easy to see which props or state your effect uses, and to ensure they're all declared:

```
function ProductPage({ productId }) {
  const [product, setProduct] = useState(null);

  useEffect(() => {
    // By moving this function inside the effect, we can clearly see the values it
    uses.
    async function fetchProduct() {
      const response = await fetch('http://myapi/product/' + productId);
      const json = await response.json();
      setProduct(json);
    }

    fetchProduct();
  }, [productId]); // ✅ Valid because our effect only uses productId
  // ...
}
```

This also allows you to handle out-of-order responses with a local variable inside the effect:

```
useEffect(() => {
  let ignore = false;
  async function fetchProduct() {
    const response = await fetch('http://myapi/product/' + productId);
    const json = await response.json();
    if (!ignore) setProduct(json);
  }

  fetchProduct();
  return () => { ignore = true };
}, [productId]);
```

We moved the function inside the effect so it doesn't need to be in its dependency list.

Tip

Check out [this small demo](#) and [this article](#) to learn more about data fetching with Hooks.

If for some reason you can't move a function inside an effect, there are a few more options:

- You can try moving that function outside of your component. In that case, the function is guaranteed to not reference any props or state, and also doesn't need to be in the list of dependencies.
- If the function you're calling is a pure computation and is safe to call while rendering, you may **call it outside of the effect instead**, and make the effect depend on the returned value.
- As a last resort, you can **add a function to effect dependencies but wrap its definition** into the `useCallback` Hook. This ensures it doesn't change on every render unless *its own* dependencies also change:

```
function ProductPage({ productId }) {
  // ✅ Wrap with useCallback to avoid change on every render
  const fetchProduct = useCallback(() => {
    // ... Does something with productId ...
  }, [productId]); // ✅ All useCallback dependencies are specified

  return <ProductDetails fetchProduct={fetchProduct} />;
}

function ProductDetails({ fetchProduct }) {
  useEffect(() => {
    fetchProduct();
  }, [fetchProduct]); // ✅ All useEffect dependencies are specified
  // ...
}
```

Note that in the above example we **need** to keep the function in the dependencies list. This ensures that a change in the `productId` prop of `ProductPage` automatically triggers a refetch in the `ProductDetails` component.

What can I do if my effect dependencies change too often?

Sometimes, your effect may be using state that changes too often. You might be tempted to omit that state from a list of dependencies, but that usually leads to bugs:

```
function Counter() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    const id = setInterval(() => {
      setCount(count + 1); // This effect depends on the `count` state
    }, 1000);
    return () => clearInterval(id);
  }, []); // 🚨 Bug: `count` is not specified as a dependency

  return <h1>{count}</h1>;
}
```

The empty set of dependencies, `[]`, means that the effect will only run once when the component mounts, and not on every re-render. The problem is that inside the `setInterval` callback, the value of `count` does not change, because we've created a closure with the value of `count` set to `0` as it was when the effect callback ran. Every second, this callback then calls `setCount(0 + 1)`, so the count never goes above 1.

Specifying `[count]` as a list of dependencies would fix the bug, but would cause the interval to be reset on every change. Effectively, each `setInterval` would get one chance to execute before being cleared (similar to a `setTimeout`.) That may not be desirable. To fix this, we can use the [functional update form of `useState`](#). It lets us specify *how* the state needs to change without referencing the *current* state:

```
function Counter() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    const id = setInterval(() => {
      setCount(c => c + 1); // ✅ This doesn't depend on `count` variable outside
    }, 1000);
    return () => clearInterval(id);
  }, []); // ✅ Our effect doesn't use any variables in the component scope

  return <h1>{count}</h1>;
}
```

(The identity of the `setCount` function is guaranteed to be stable so it's safe to omit.)

Now, the `setInterval` callback executes once a second, but each time the inner call to `setCount` can use an up-to-date value for `count` (called `c` in the callback here.)

In more complex cases (such as if one state depends on another state), try moving the state update logic outside the effect with the [useReducer Hook](#). [This article](#) offers an example of how you can do this. **The identity of the `dispatch` function from `useReducer` is always stable** — even if the reducer function is declared inside the component and reads its props.

As a last resort, if you want something like `this` in a class, you can [use a ref](#) to hold a mutable variable. Then you can write and read to it. For example:

```
function Example(props) {
  // Keep latest props in a ref.
  const latestProps = useRef(props);
  useEffect(() => {
    latestProps.current = props;
  });

  useEffect(() => {
    function tick() {
      // Read latest props at any time
      console.log(latestProps.current);
    }
  });

  const id = setInterval(tick, 1000);
  return () => clearInterval(id);
}, []); // This effect never re-runs
```

Only do this if you couldn't find a better alternative, as relying on mutation makes components less predictable. If there's a specific pattern that doesn't translate well, [file an issue](#) with a runnable example code and we can try to help.

How do I implement `shouldComponentUpdate`?

You can wrap a function component with `React.memo` to shallowly compare its props:

```
const Button = React.memo((props) => {
  // your component
});
```

It's not a Hook because it doesn't compose like Hooks do. `React.memo` is equivalent to `PureComponent`, but it only compares props. (You can also add a second argument to specify a custom comparison function that takes the old and new props. If it returns true, the update is skipped.)

`React.memo` doesn't compare state because there is no single state object to compare. But you can make children pure too, or even [optimize individual children with `useMemo`](#).

How to memoize calculations?

The `useMemo` Hook lets you cache calculations between multiple renders by "remembering" the previous computation:

```
const memoizedValue = useMemo(() => computeExpensiveValue(a, b), [a, b]);
```

This code calls `computeExpensiveValue(a, b)`. But if the dependencies `[a, b]` haven't changed since the last value, `useMemo` skips calling it a second time and simply reuses the last value it returned.

Remember that the function passed to `useMemo` runs during rendering. Don't do anything there that you wouldn't normally do while rendering. For example, side effects belong in `useEffect`, not `useMemo`.

You may rely on `useMemo` as a performance optimization, not as a semantic guarantee. In the future, React may choose to "forget" some previously memoized values and recalculate them on next render, e.g. to free memory for offscreen components. Write your code so that it still works without `useMemo` — and then add it to optimize performance. (For rare cases when a value must *never* be recomputed, you can [lazily initialize a ref](#).)

Conveniently, `useMemo` also lets you skip an expensive re-render of a child:

```
function Parent({ a, b }) {
  // Only re-rendered if `a` changes:
  const child1 = useMemo(() => <Child1 a={a} />, [a]);
  // Only re-rendered if `b` changes:
  const child2 = useMemo(() => <Child2 b={b} />, [b]);
  return (
    <>
      {child1}
      {child2}
    </>
  )
}
```

Note that this approach won't work in a loop because Hook calls `can't` be placed inside loops. But you can extract a separate component for the list item, and call `useMemo` there.

How to create expensive objects lazily?

`useMemo` lets you [memoize an expensive calculation](#) if the dependencies are the same. However, it only serves as a hint, and doesn't *guarantee* the computation won't re-run. But sometimes you need to be sure an object is only created once.

The first common use case is when creating the initial state is expensive:

```
function Table(props) {
  // ⚠️ createRows() is called on every render
  const [rows, setRows] = useState(createRows(props.count));
  // ...
}
```

To avoid re-creating the ignored initial state, we can pass a **function** to `useState`:

```
function Table(props) {
  // ✅ createRows() is only called once
  const [rows, setRows] = useState(() => createRows(props.count));
  // ...
}
```

React will only call this function during the first render. See the [useState API reference](#).

You might also occasionally want to avoid re-creating the `useRef()` initial value. For example, maybe you want to ensure some imperative class instance only gets created once:

```
function Image(props) {
  // ⚠️ IntersectionObserver is created on every render
  const ref = useRef(new IntersectionObserver(onIntersect));
  // ...
}
```

`useRef` **does not** accept a special function overload like `useState`. Instead, you can write your own function that creates and sets it lazily:

```
function Image(props) {
  const ref = useRef(null);

  // ✅ IntersectionObserver is created lazily once
  function getObserver() {
    if (ref.current === null) {
      ref.current = new IntersectionObserver(onIntersect);
    }
    return ref.current;
  }

  // When you need it, call getObserver()
  // ...
}
```

This avoids creating an expensive object until it's truly needed for the first time. If you use Flow or TypeScript, you can also give `getObserver()` a non-nullable type for convenience.

Are Hooks slow because of creating functions in render?

No. In modern browsers, the raw performance of closures compared to classes doesn't differ significantly except in extreme scenarios.

In addition, consider that the design of Hooks is more efficient in a couple ways:

- Hooks avoid a lot of the overhead that classes require, like the cost of creating class instances and binding event handlers in the constructor.
- **Idiomatic code using Hooks doesn't need the deep component tree nesting** that is prevalent in codebases that use higher-order components, render props, and context. With smaller component trees, React has less work to do.

Traditionally, performance concerns around inline functions in React have been related to how passing new callbacks on each render breaks `shouldComponentUpdate` optimizations in child components. Hooks approach this problem from three sides.

- The `useCallback` Hook lets you keep the same callback reference between re-renders so that `shouldComponentUpdate` continues to work:

```
// Will not change unless `a` or `b` changes
const memoizedCallback = useCallback(() => {
  doSomething(a, b);
}, [a, b]);
```

- The `useMemo` Hook makes it easier to control when individual children update, reducing the need for pure components.
- Finally, the `useReducer` Hook reduces the need to pass callbacks deeply, as explained below.

How to avoid passing callbacks down?

We've found that most people don't enjoy manually passing callbacks through every level of a component tree. Even though it is more explicit, it can feel like a lot of "plumbing".

In large component trees, an alternative we recommend is to pass down a `dispatch` function from `useReducer` via context:

```
const TodosDispatch = React.createContext(null);

function TodosApp() {
  // Note: `dispatch` won't change between re-renders
  const [todos, dispatch] = useReducer(todosReducer);

  return (
    <TodosDispatch.Provider value={dispatch}>
      <DeepTree todos={todos} />
    </TodosDispatch.Provider>
  );
}
```

```
</TodosDispatch.Provider>
});
```

Any child in the tree inside `TodosApp` can use the `dispatch` function to pass actions up to `TodosApp`:

```
function DeepChild(props) {
  // If we want to perform an action, we can get dispatch from context.
  const dispatch = useContext(TodosDispatch);

  function handleClick() {
    dispatch({ type: 'add', text: 'hello' });
  }

  return (
    <button onClick={handleClick}>Add todo</button>
  );
}
```

This is both more convenient from the maintenance perspective (no need to keep forwarding callbacks), and avoids the callback problem altogether. Passing `dispatch` down like this is the recommended pattern for deep updates.

Note that you can still choose whether to pass the application *state* down as props (more explicit) or as context (more convenient for very deep updates). If you use context to pass down the state too, use two different context types -- the `dispatch` context never changes, so components that read it don't need to rerender unless they also need the application state.

How to read an often-changing value from `useCallback`?

Note

We recommend to [pass `dispatch` down in context](#) rather than individual callbacks in props. The approach below is only mentioned here for completeness and as an escape hatch.

In some rare cases you might need to memoize a callback with `useCallback` but the memoization doesn't work very well because the inner function has to be re-created too often. If the function you're memoizing is an event handler and isn't used during rendering, you can use [ref as an instance variable](#), and save the last committed value into it manually:

```
function Form() {
  const [text, updateText] = useState('');
  const textRef = useRef();

  useEffect(() => {
    textRef.current = text; // Write it to the ref
  });

  const handleSubmit = useCallback(() => {
    const currentText = textRef.current; // Read it from the ref
    alert(currentText);
  }, [textRef]); // Don't recreate handleSubmit like [text] would do
```

```

    return (
      <>
        <input value={text} onChange={e => updateText(e.target.value)} />
        <ExpensiveTree onSubmit={handleSubmit} />
      </>
    );
}
}

```

This is a rather convoluted pattern but it shows that you can do this escape hatch optimization if you need it. It's more bearable if you extract it to a custom Hook:

```

function Form() {
  const [text, updateText] = useState('');
  // Will be memoized even if `text` changes:
  const handleSubmit = useEventCallback(() => {
    alert(text);
  }, [text]);

  return (
    <>
      <input value={text} onChange={e => updateText(e.target.value)} />
      <ExpensiveTree onSubmit={handleSubmit} />
    </>
  );
}

function useEventCallback(fn, dependencies) {
  const ref = useRef(() => {
    throw new Error('Cannot call an event handler while rendering.');
  });

  useEffect(() => {
    ref.current = fn;
  }, [fn, ...dependencies]);

  return useCallback(() => {
    const fn = ref.current;
    return fn();
  }, [ref]);
}

```

In either case, we **don't recommend this pattern** and only show it here for completeness. Instead, it is preferable to [avoid passing callbacks deep down](#).

Under the Hood

How does React associate Hook calls with components?

React keeps track of the currently rendering component. Thanks to the [Rules of Hooks](#), we know that Hooks are only called from React components (or custom Hooks -- which are also only called from React components).

There is an internal list of "memory cells" associated with each component. They're just JavaScript objects where we can put some data. When you call a Hook like `useState()`, it reads the current cell (or initializes it during the first render), and then moves the pointer to the next one. This is how multiple `useState()` calls each get independent local state.

What is the prior art for Hooks?

Hooks synthesize ideas from several different sources:

- Our old experiments with functional APIs in the [react-future](#) repository.
- React community's experiments with render prop APIs, including [Ryan Florence's Reactions Component](#).
- [Dominic Gannaway](#)'s `adopt keyword` proposal as a sugar syntax for render props.
- State variables and state cells in [DisplayScript](#).
- [Reducer components](#) in ReasonReact.
- [Subscriptions](#) in Rx.
- [Algebraic effects](#) in Multicore OCaml.

[Sebastian Markbåge](#) came up with the original design for Hooks, later refined by [Andrew Clark](#), [Sophie Alpert](#), [Dominic Gannaway](#), and other members of the React team.

היכרות עם Hooks

הם תוספת חדשה בראקט 16.8. הם נותנים לנו להשתמש בסיטייט ובתכונות אחרות של ריאקט מבלתי כתוב מחלקות *Hooks*.

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

function Example() {
  // הערה משתמש שנקרא "count"
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

היא הוק הראשון שעליו נלמד, והדוגמא הזו היא רק הצעה אליו `useState` הפונקציה החדשה.

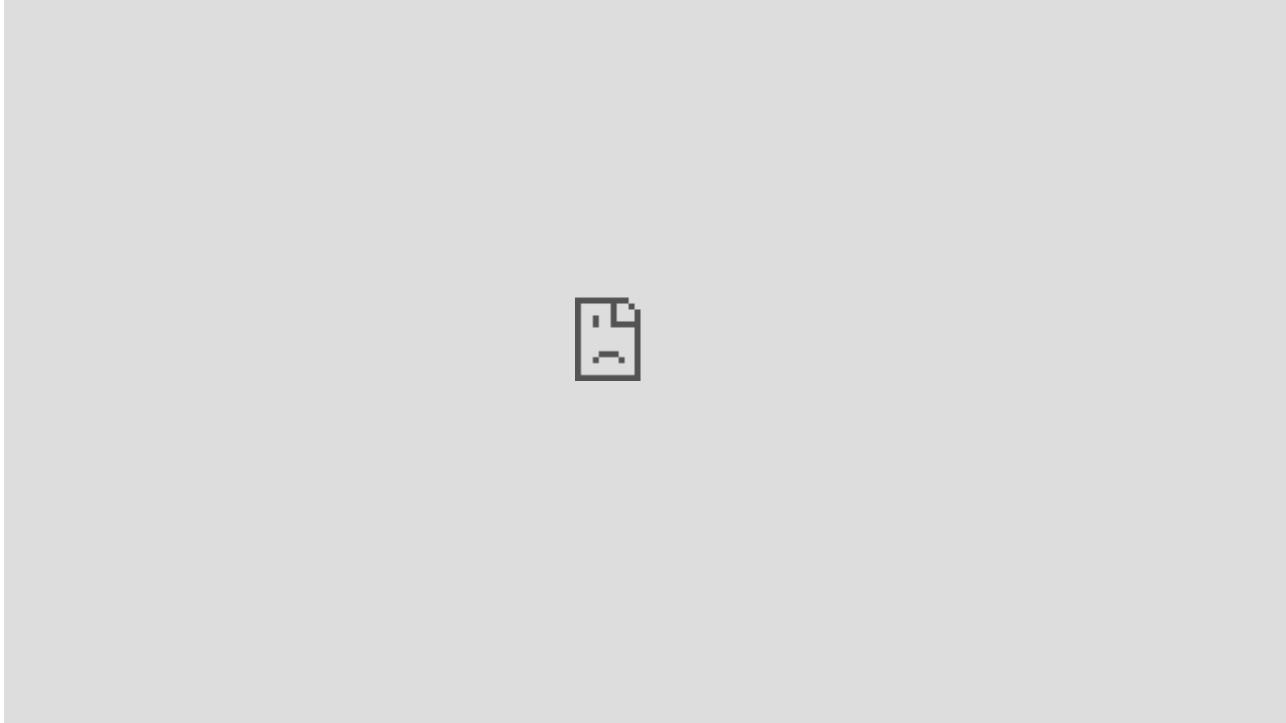
ניתן להתחיל ללמידה על הוקים [בעמוד הבא](#). בעמוד זה נמשך בהסביר למה הוספנו הוקים לראקט, ואיך הם יכולים לעזור לנו לכתוב יישומים מעולים.

הערה

ראקט 16.8.0 היא הגרסה הראשונה שתומכת בהוקים. בעת שדרוג אליה, שים לב לעדכן את כל החבילות, כולל [React Native 0.59](#).

סרטון מבוא

הראה איך Ryan Florence הציגו את הנושא של הוקים, ואחריהם Sophie Alpert ו-Dan Abramov ב React Conf 2018, לשכטב אפליקציית Riot על מנת שתעשה שימוש בהם. צפו בסרטון:



אין שינויים ששוברים תאימות לאחר

לפניהם שמשיך, שימושם לב שהוקים הם

- בגדר בחירה.** ניתן לכתוב הוקים בכמה קומפוננטות מביי לשכטב אף קוד קיים. בנוסף אין חובה ללמוד עליהם או להשתמש בהם.
- breaking changes לאחור ב-100%.** הם לא גורמים ל-**זמןניים עכשוויים**. ניתן לכתוב הוקים לאחר שחרור גרסה 16.8.0.

אין תוכניות להסיר את המחלקות מפרויקט. ניתן לקרוא בהרחבה על האימוץ הדרוגתי של הוקים [בפרק התחתון](#) של עמוד זה.

ישיר יותר לكونספטיים בפרויקט API **הhookים לא משנים שום דבר בكونספטיים העיקריים של ריאקט**. אדרבא, הם מספקים כפי שנראה בהמשך, הוקים אפשריים לנו דרך חדשה (lifecycle) ומעגל החיים refs, state, context, props. ושזהו לשלב בין הקונספטיים הללו.

אם אתה רוצה להתחיל ללמידה על הוקים, קפוץ ישירות [לעמוד הבא!](#). תוכל גם להמשיך לקרוא עמוד זה וללמוד על הסיבות להוספת הhookים, ואיך נוכל להתחל לשתף בהם מביי לכתוב מחדש את האפליקציות שלנו.

המניע

הוקים פותרים מגוון רחב של בעיות שלכורה לא הקשורות אחת לשניה בפרויקט, אשר נתקלנו בהן במשך יותר מחמש שנים של כתיבה ותחזוקת שורות אלפי קומפוננטות. בין אם אתה לומד ריאקט, כותב על בסיס יומיומי, או אפילו מעדיך ספריה אחרת עם מודול קומפוננטות דומה, ייתכן שתזהה חלק מבעיות אלו.

קשה לעשות שימוש חזר בלוגיקה בין קומפוננטות

אם עבדת עם ריאקט זמן מה, (store-לדוגמה, לחברו אותה ל) ריאקט אינה מציעה דרך "קשר" פועלה החורט לקומפוננטה שמנסים לפטור את זה. הבעיה היא שהם **render props** **higher-order components** נראה שאתה מכיר פיצ'רים כמו דורשים ארגון מחדש של הקומפוננטות בעת השימוש בהן, מה שיכל להיות מסובך ומסבוך על מעקב אחרי קוד. אם נתבונן על של קומפוננטות שעוטפות "wrapper hell" רוב השירותים שנראה סוג של React DevTools, אפליקציית ריאקט וגילה דרך **dev tools** ועוד. אפילו שניtan **لسן אותו ב** **consumers**, **higher-order components**, **render props** **providers** בשכבות של **tools**. זה מצביע על בעיה עמוקה יותר: ריאקט צריכה בסיס טוב יותר לשיתוף לוגיקה,

עם הוקים, ניתן ליצא לוגיקה מתוך קומפוננטה כך שהיא ניתנת לבדוק בנפרד ולעשות בה שימוש מחדש. הוקים מאפשרים **שימוש חדש בלוגיקה מבלי לשנות את סדר הקומפוננטות**. זה מקל על שיתוף הוקים בין קומפוננטות או עם אנשים אחרים.

נדון על כך בהרחבה [בבנייה הוקים משלך](#)

קומפוננטות מורכבות הקשורות להבנה

כל מתודת side-effects נאלצנו לתזקק קומפוננטות שהתחילה כפשוטות וגדלו לבלאן ביל' נשלט של לוגיקה והכילה לעיתים קרובות ערבות של לוגיקה לא קשורה. לדוגמה, קומפוננטות שעשוות למשוך מידע מידי lifecycle העשויה להכיל **componentDidMount** **componentDidUpdate** **componentWillUnmount** -ובן-其间 **componentWillUnmount** -עם ניקיון שלהם ב event listeners, גם לוגיקה לא קשורה שמאגדירה קטעי קוד הקשורים אחד. זה גורם קל לבאגים ולוחסר עקבותיו. לשני מטאצלים, אבל קוד שלחלוטין איןנו קשר מתחדש במתודה אחת. זה גורם קל לבאגים ולוחסר עקבותיו.

במקרים ובמים זה לא אפשרי להפריד קומפוננטות אלו לקומפוננטות קטנות יותר מכיוון שהלוגיקה נמצאת בכל מקום. בנוסף זה קשה להריץ טסטים עליהם. זאת אחת הסיבות שפתחים רבים מעדיפים לשבל את ריאקט עם ספריית ניהול סטייט חיצונית. למרות זאת, ההילוב גורם לעיתים ליותר מדי אבסטרקציה, דורש קופץ בין קבצים שונים, ומסבב על שימוש חזר של קומפוננטות.

setup (דוגמה) על מנת לפתור זאת, הוקים נותנים לנו **לפצל חלקים קשורים של קומפוננטה לפונקציות קטנות** יותר במחלקה. ניתן גם לנוהל את lifecycle במקום להיות מוכרים לפצל בתוך מתודות, (**למידע subscription** או **fetch** או **reducer**) הספריטי המוקומי של הקומפוננטה עם

נדון על כך בהרחבה [בשימוש בEffect Hook](#)

מחלקות מבלבלות גם אנשים וגם מחשבים

בנוסף לעשיית קוד קשה יותר לשימוש חזר ולארגון שלו, מצאנו שמחלקות עלולות להוות חסם ללימוד ריאקט. צריך להבין איך **this** bind event handlers. אנשים יכוליםים להסתדר עם קונספטים כמו **syntax** **שימוש בפלאגינים לא יציבים** של הקוד מאד 어렵. אנשים שלומדים ריאקט יכוליםים להשתדר עם מושגים כמו **props**, **state**-flow-ו **recompute**. מחלוקת לקומפוננטות פונקצייה בריאקט ומתי להשתמש בכל אחת מוביל לחילוקי דעתות אפילו בין מפתחי ריאקט מנוסים.

בנוסף, ריאקט כבר קיימת יותר מ 5 שנים, ואנחנו רוצים שתישאר רלוונטיות גם ל 5 השנים הבאות. כדי שאחרים הראו, יש **לקומפילציה מבועד** של קומפוננטות הרובה פוטנציאלי. במיוחד אם זה לא מוגבל, **Glimmer**, וריאנו תוצאות מבטיחות. לעומת זאת, מצאנו **Prepack**, באמצעות **component folding** לאחרונה, התנסנו עם **Templates**. שקומפוננטות מחלקה יכולות כולל שימושים שמחלישים את שיפורים אלו. מחלקות גם גורמות לביעות בכלים של היום. לביעות יציבות. אנחנו רוצים לעוד שיטות לחלקות לא עובד בצורה טובה, וגורם לפיצ'רים כמו **minifying**, למשל לשנותן לעשות אופטימיזציות ל코드 API.

כדי לפטור בעיות אלו, הוקים נותנים לנו להשתמש ביותר **פיצ'רים** של ריאקט **MBOLY** לכתוב מחלקות. באופן תיאורתי, מוביל ללמידה טכניות מסוימות של escape hatcheseKomponenntות ריאקט תמיד היו קרובות לפונקציות. הוקים נותנים גישה לתוכנות פונקציונלי.

דוגמאות

הוא התחלה טובה למידה של הוקים **Hooks-הצצה** ל

אסטרטגיית אימוץ הדרגתית

אם;لك: אין תוכניות להוציא את המחלקות מריאקט

שיצא. הוקים הם חדשים, ויכול API אנחנו יודעים שפתחי ריאקט ממוקדים בהוצאה מוצרים ולאין להם זמן לקרוא על כל להיות שעדייף לחכות מעט לעד דוגמאות ומדריכים לפני ששולטים למד או להשתמש בהם.

מפורט שצולל יותר RFC אנחנו גם מבינים שהרף להוספת קונספט מרכזי חדש בריект הוא גבוה. לקרים סקרנים, הכו עמוק למנוע שמאחורי ההוקים, ונตอน פרספקטיביה נוספת על החלטות העיצוב שלהם.

אנו Hooks-עובדים לצד קוד קיים אז אתה יכול לאמץ אותם בהדרגה. אין צורך למהר לעבור ל **Hooks**, באופן מכריע ממליצים להמנע מכל "שכתב גדול", במיוחד עבור קומפוננטת מחלקה מסוובכת קיימת. זה לוקח קצת זמן בשינוי מחשבתי בקומפוננטה חדשה ולא קריטית, זה הכי טוב להתרגל להשתמש בהתחלה ב Hooks-.להתחליל" לחשוב בבקשת תרגישו חופשי **שלוח אلينו משוב**, חיובי Hooks ולהבטיח שכולם בצוות שלך מרגשים בונה אותם. אחרי שניסיתם את או שלילי.

אנחנו מתכוונים שהוקים יכסו את כל מקרי השימוש במחלקות, אבל **נמשך** בקומפוננטות מחלקות לעתיד הנראה לעין. בפייסבוק, יש לנו שירותים/API קומפוננטות שכותבות כמחלקות, ואין לנו שום תוכנית לשכתב אותן. במקום זאת, אנחנו מתחילה לחשתחם בהוקים בקוד חדש לצד המחלקות.

שאלות נפוצות

הכנו עמוד **שאלות נפוצות של הוקים** שעונה על השאלות הנפוצות ביותר עליהם.

הצדדים הבאים

בסוף דף זה, אמרו להיות לך מושג קלוש לגבי הביעות שהוקים פותרים, אבל כנראה שפרטיהם ובטים עדין לא ברורים. נמשך **לעמדת הבא** ושם נתחיל ללמידה על הוקים על ידי דוגמה.

react

[Go to TOC](#)

הוצאה ל-Hooks

ambilי לכתוב React ובפייצ'רים נוספים של state-state נוטנים לנו להשתמש ב-16.8-hם תוספת חדשה ב-React. מחלוקת.

Hooks. זהוי סקירה מהירה. אם [למשתמשי Hooks הם בעלי יכולות לאחור](#). עמדו זה מספק סקירה כללית של Hooks מנוסים. זהוי סקירה מהירה. אם [למשתמשי Hooks הם בעלי יכולות לאחור](#). עמדו זה מספק סקירה כללית של Hooks אתה מתבלבל במהלך, חפש תיבה צחוכה כמו זו:

הסבר מפורט

קרא את [המנוע](#) כדי למדוד מדוע אנו מציגים את Hooks ל-React.

↑ ↑ ↑ כל סעיף נגמר עם תיבת צחוכה כמו זו. התיבות מפנות להסבירים מפורטים ↑ ↑ ↑

📌 State Hook

כשሎחים על הcptor, הוא מגביל את הערך counter. דוגמה זו מרננדת:

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

function Example() {
  // ייקרא state הגדרת משתנה "count"
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>{count} לחץ</p>
      <button onClick={() => setCount(count + 1)}>
        לחץ עליי
      </button>
    </div>
  );
}
```

אנחנו קוראים לו בתוך קומפוננט פונקציה על מנת להוציא לו. (נדבר על מה זה אומר בעוד רגע) הוא `useState` Hook, וכך העכשווי ופונקציית state -מחזירה זוג: ערך ה `useState`React. זהה בין וינדרוים state-state-תשמר את ה מקום state במקומו. `this.setState` -או מקום אחר. זה דומה ל event handler שמאפשרת לנו לעדכן אותו. ניתן לקרוא לה מתחם במקורה, `this.state` -נראה דוגמה שימושה בו). הישן עם החדש state-state-מלבד שהוא לא מmag את ה [ב-שימוש ב](#) `useState` -ל-`this.state` State Hook.)

שלנו מתחילה מאפס. שם לב counter-התחלת. בדוגמה שלעיל, זה 0 מכיוון שהstate הוא `useState`-הקלט היחיד ל state-state-כאן לא חייב להיות עצם -- למרות שהוא יכול אם תרצה. נעשה שימוש בקלט ה state , `this.state` -שבניגוד לההתחלת רק בזמן הרינדרו הראשון.

מרובים state הגדרת משתני

יוטר מפעם אחת בקומפוננטה יחידה Hooks-ניתן להשתמש ב:

```
function ExampleWithManyStates() {
  // הגדר מספר משתני state!
  const [age, setAge] = useState(42);
  const [fruit, setFruit] = useState('banana');
  const [todos, setTodos] = useState([{ text: 'Learn Hooks' }]);
  // ...
}
```

שמות . useState -שהגדכנו על ידי קרייה ל **ת לחבר הרשמה המפוקת** נותן לנו אפשרות לתת שמות שונים למשתני ה **useState** פעמים רבות, אתה קורא ל **useState** -מניחה שם את **useState** של API-אלו אין חלק מה עושה זאת באותו סדר בזמן כל רינדור. נחזור חזרה לממה זה עובד ומתי זה שימושי מיותר יותר.

אבל מה זה Hook?

לא Hooks. ותוכנות מחזור חיים מתוך קומפוננטות פונקצייתית React של Hooks -הם פונקציות הקשורות לכך "להתחבר" ל **ת לחבר** (בוקומפוננטות חדשות אם תרצה Hooks-הקומפוננטות הקיימות שלו בין לילה, אבל תוכל להתחיל להשתמש ב **useState** מיליצים לשכתב את).

משלך ולעתות שימוש אחר בלוגיקה Hooks אתה יכול בנוסף ליצור . המובנים כמו Hooks מספקת מספר (בוקומפוננטות בין Hooks-בין Hooks-בין קומפוננטות שונות). נუיף מבט ב **stateful** שהוא לפני הכל。

הסביר מפורט

בעמוד ייעדי: [שימוש ב-State Hook](#).

⚡ Effect Hook

אנחנו React-מתוך קומפוננטות ב DOM-או שינוי דני של ה **data fetching, subscriptions** רוב הסיכויים שביצעת בעבר בغالל שהם יכולים להשפיע על קומפוננטות אחרות ולא ("או "אפקטים") (side effects) קוראים לפעולות אלו "תופעות לוואי". יכולים לקרות בזמן רינדור.

מוסיף את היכולת לבצע תופעות לוואי מתוך קומפוננטת פונקציה. זה משרות את אותה מטרה , **useEffect**, אבל מאוחד במחוקות React, **componentDidMount**, **componentDidUpdate**, **componentWillUnmount** וכו' (למתקודם אלו [שימוש ב-useEffect](#) נראה דוגמאות שימוש ב **Effect Hook**.)

מעדכנת את ה-React-לדוגמה, קומפוננטה זו קובעת את כוורת העמד לאחר ש

```
import React, { useState, useEffect } from 'react';

function Example() {
  const [count, setCount] = useState(0);

  // -componentDidMount ו -componentDidUpdate:
  useEffect(() => {
    // הדף API-עדכו כוורת העמוד על ידי שימוש ב
    document.title = `${count} ${count} לנצח`;
  });

  return (
    <div>
      <p>${count}</p>
    </div>
  );
}
```

```

        <button onClick={() => setCount(count + 1)}>
          +
        </button>
      </div>
    );
}

```

DOM-להריץ את פונקציית ה"אפקט" שלק לאחר הזרמת השינויים ל-React אתה אומר ל-`useEffect`- כאשר קורא ל-`useEffect` את האפקטים, שלה. כבירות מחדל state-props-אפקטים מוגדרים בתוך הקומפוננטה כך שיש להם גישה ל-`useEffect`-נדבר עוד בהמשך על כיצד זה משתמש למחזר החיים במחלקות [בשימוש ב](#). לאחר כל רינדור -- כולל הרינדור הראשון (Hook.)

אפקטים יכולים גם לציין איך "לנקות" אחורי עצם על ידי החזרת פונקציה. לדוגמה, קומפוננטה זו משתמש באפקט על מנת ממנו `unsubscribe` לסטטוס אונליין של חבר, ומנקה על ידי ביצוע `subscribe` לעשוות:

```

import React, { useState, useEffect } from 'react';

function FriendStatus(props) {
  const [isOnline, setIsOnline] = useState(null);

  function handleStatusChange(status) {
    setIsOnline(status.isOnline);
  }

  useEffect(() => {
    ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);

    return () => {
      ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
    };
  });

  if (isOnline === null) {
    return 'גאנט...';
  }
  return isOnline ? 'מונתק' : 'מחובר';
}

```

כמו גם לפני הריצה מחדש שלנו כשהקומפוננטה מבצעת `unmounting`, בדוגמה זו `ChatAPI` מ-`unsubscribe` מ-`props.friend.id` אם אתה רוצה, יש דרך [לדלג על re-subscribing](#) אם `props.friend.id` לא השתנה `ChatAPI`-שהעברנו ל-

אתה יכול להשתמש ביותר מאפקט אחד בתחום קומפוננטה, בדיקן כמו עם `useState`:

```

function FriendStatusWithCounter(props) {
  const [count, setCount] = useState(0);
  useEffect(() => {
    document.title = `${count} לנצח`;
  });

  const [isOnline, setIsOnline] = useState(null);
  useEffect(() => {
    ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
    return () => {
      ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
    };
  });
}

```

```
function handleStatusChange(status) {
  setIsOnline(status.isOnline);
}
// ...
```

במקום לאlez, (כגון הוספה והסרת) נותנים לך לארגן תופעות לוואי בקומפוננטה לפי החלקים הקשורים Hooks. פיצול המבוסס על מתחדשות מחזור חיים.

הסביר מפורט

בעומד ייעדי: [שימוש ב useEffect](#).

חוקי Hooks

אך הם מטילים שני חוקים נוספים, הם פונקציות Hooks:

- בתוך לולאות, פקודות תנאי, או פונקציות מקווננות Hooks-רָק ברמה העליונה. אל תקרא ל-Hooks-ניתן לקרוא ל-
- (ש) רגילות JavaScript מתוך פונקציות Hooks-אל תקרא ל- **React**. רָק מתוך קומפוננטות פונקציה של Hooks-קרא ל- () מותאים אישית משלה. נלמד עליהם בעוד וגע Hooks -- ממנה Hooks-רָק מקום אחד נוסף שנייהן לקרוא ל-

shawcץ חוקים אלו אוטומטית. אנו מבינים שהחוקים אלו עלולים להיראות כמגבילים או מבלבלים [linting](#) אנחנו מספקים [תוספות](#) לעבוד היב Hooks-בהתחלת, אך הם חוווניים כדי לגרום ל.

הסביר מפורט

ניתן ללמוד עוד על חוקים אלו בעומד ייעדי: [חוקי Hooks](#).

משלך Hooks בנית

בין קומפוננטות. באופן מסורתי, היו שני פתרונות נפוצים stateful לעתים, אנו רוצים לעשות שימוש חזר בלוגיקה שהיא מותאמת אישית נתונים לך לעשות זאת, אבל מבלתי להוסיף עוד [render props](#). Hooks-לבעיה זו: [קומפוננטות בסיס דגם גבוהה](#) ו- [קומפוננטות לעצ שלך](#).

על מנת לעשות [useState](#) ו- [useEffect](#) שקוראת ל [FriendStatus](#) מוקדם יותר בעומד זה, הצגנו את קומפוננטה subscribe לסטטוס חיבור אונליין של חבר. נגיד שנרצה בנוסף לבצע שימוש חזר בלוגיקה זו בקומפוננטה אחרת subscribe:

מותאם אישית שנקרה לו Hook ראשית, נחלץ לוגיקה זו לתוך [useFriendStatus](#):

```
import React, { useState, useEffect } from 'react';

function useFriendStatus(friendID) {
  const [isOnline, setIsOnline] = useState(null);

  function handleStatusChange(status) {
    setIsOnline(status.isOnline);
  }
}
```

```

useEffect(() => {
  ChatAPI.subscribeToFriendStatus(friendID, handleStatusChange);
  return () => {
    ChatAPI.unsubscribeFromFriendStatus(friendID, handleStatusChange);
  };
});

return isOnline;
}

```

בכלט, ומוחזר האם החבר שלנו מחובר או לא `friendID` הוא לוקח את

עכשו נוכל להשתמש בזה מתוך שתי הקומפוננטות:

```

function FriendStatus(props) {
  const isOnline = useFriendStatus(props.friend.id);

  if (isOnline === null) {
    return 'טען...';
  }
  return isOnline ? 'מחובר' : 'מנתק';
}

function FriendListItem(props) {
  const isOnline = useFriendStatus(props.friend.id);

  return (
    <li style={{ color: isOnline ? 'green' : 'black' }}>
      {props.friend.name}
    </li>
  );
}

```

לא ב-stateful הям דרך לעשות שימוש חוץ בלוגיקה *שהיא Hooks*. של כל קומפוננטה הוא עצמאי לחולוטן state-ה מותאם אישית פעריים Hook מבחד לغمרי -- כך שניתן להשתמש באותו state ייש Hook-עצמו. למעשה, לכל קרייה ל-*בקומפוננטה אחת*.

Hooks-ויהי קוראת ל `"use"` מותאים אישית הם יותר מוסכמה מאשר פיצ'ר. אם שם של פונקציה מתחילה עם Hooks-היא הסיבה שתוסף ה `useSomething` Hook-טלנו מסוגל לאתר באגים בקוד שימוש *linting* Hooks.

מותאים אישית שמכסים טווח רחב של תרחישי שימוש כגון טיפול בטפסים, אינימציה Hooks ניתן לכתב מותאים אישית Hooks טימרים וכנראה עד מקרים שלא ליקחנו בחשבון עדין. אנחנו נרגשים לראות אילו, subscriptions הקריאה תמציא.

הסבר מפורט

[משלך](#) מותאים אישית בעמד ייעודי: [בנייה Hooks](#) ניתן ללמוד עוד על

Hooks אחרים

ב-React subscribe-context ל-React נתן לך לעשות `useContext`, מובנים שאולי תמצא שימושים. לדוגמה Hooks ישנים כמה מבלי להשתמש בקינו:

```
function Example() {
  const locale = useContext(LocaleContext);
  const theme = useContext(ThemeContext);
  // ...
}
```

1- `useReducer` מוקומי של קומפוננטות מורכבות עם state נתן לך לנחל reducer:

```
function Todos() {
  const [todos, dispatch] = useReducer(todosReducer);
  // ...
}
```

הסבר מפורט

מובנים בעמוד ייעודי: [עיוון Hooks API](#).

הצעדים הבאים

זה היה מהיר! אם יש לכם דברים שלא כל כך הגיוניים לך או שתרצה ללמוד יותר לעומק, תוכל לקרוא את העמודים הבאים, [State Hook](#).

ואיך נתחילה להשתמש בהם לצד מחלקות -- מבלי Hooks לבסוף, אל תחמייז את [עמד ההקדמה](#) שמסביר למה אנחנו מוסיפים לשכתב את האפליקציות שלנו.

[Go to TOC](#)

ב-עיוון Hooks API

מוביל לכתב מחלוקת React ופיצ'רים אחרים של state-הם נוגנים לנו להשתמש ב Hooks API. הם תוספת חדשה ב React 16.8. המובנים בתוך Hooks API מגדיר זה מתאר את Hooks.

חדש לך, יכול להיות שתרצה לקרוא את [הסקירה הכללית](#) קודם. יכול להיות שאתה מודע לשימושי Hooks אם הנושא של [ב-עיוון שאלות נפוצות](#).

- [בסיסיים](#)
 - `useState`
 - `useEffect`
 - `useContext`
- [נוספים](#)
 - `useReducer`
 - `useCallback`
 - `useMemo`
 - `useRef`
 - `useImperativeHandle`
 - `useLayoutEffect`
 - `useDebugValue`
 - `useDeferredValue`
 - `useTransition`
 - `useId`
- [Library Hooks](#)
 - `useSyncExternalStore`
 - `useInsertionEffect`

בסיסיים

`useState` {#usestate}

```
const [state, setState] = useState(initialState);
```

ופונקציה על מנת לעדכן אותו, `useState` מחזיר ערך.

הוא שווה ערך לערך המועבר כารוגומנטו הראשון (`state`) המוחזר `useState`-בזמן הרינדור הראשון, ה-

חדש וקובעת רינדור חדש של הקומפוננטה `state` היא מקבלת ערך `newState` -פונקציית ה

```
setState(newState);
```

האחרון לאחר יישום העדכנים `useState`-תמיד יהיה ה `useState` בזמן הרינדרים העיקריים, הערך הראשון שמוחזר על ידי

הערה

יציבה ולא תשתנה בין רינדרים. זאת הסיבה שזה בטוח להשתמש להשミט `useState` - מבטיח שזהות פונקציית ה-React

עדכוני פונקציונליים

הfonקציה מקבלת את הערך .-הקדם, ניתן להעביר פונקציה ל-state החדש חושב באמצעות ה-state-אם ה-fonkציה שמשתמש בשתי הוצאות של counter הקדם, ותחזר ערך מעודכן. הנה דוגמה של קומפוננטת `useState` :

```
function Counter({initialCount}) {
  const [count, setCount] = useState(initialCount);
  return (
    <>
      Count: {count}
      <button onClick={() => setCount(initialCount)}>Reset</button>
      <button onClick={() => setCount(prevCount => prevCount - 1)}>-</button>
      <button onClick={() => setCount(prevCount => prevCount + 1)}>+</button>
    </>
  );
}
```

"-בutoriy ה-""+" ו-""-"" משתמשים בצורה הפונקציונלית, בעוד שהערך המעודכן מבוסס על הערך הקודם. אבל במקרה זה משתמש בצורה הרגילה, בעוד שהוא תמיד מעדכן את הספירה חוזרת לערך ההתחלתי.

הנוכחי, הרינדור הבא יдолג לגמרי state-הערה אם הפונקציה מחזירה ערך שווה לזה שקיים ב-

לא מזגת עדכוני אובייקטים באופן `useState`, שנמצאת בקומפוננטות מחלקת `useState`-בשונה ממתודת ה-`setState` שלוש נקודות(`object spread`) אוטומטי. ניתן לחוקות התנהלות זו על ידי שימוש של מעדכן פונקציה עם אופן הכתיבה של '...'):

```
const [state, setState] = useState({}); // Object.assign
setState(prevState => {
  // Object.assign
  גם יעבוד
  return {...prevState, ...updatedValues};
});
```

שיכילים מספר רב של תת-ערכים (objects) state `useReducer` או פציה נוספת נוספת היא

יצמן state

state-שהשתמשנו בו ברינדור הראשון. ברינדורים עוקבים, מתעלמים ממנו. אם ה-`initialState` הוא הערך ההתחלתי הוא התוצאה של החישוב יקר, ניתן לספק פונקציה במקום, שטרוץ רק ברינדור הראשון:

```
const [state, setState] = useState(() => {
  const initialState = someExpensiveComputation(props);
  return initialState;
});
```

יציאה state

יצא מהפעולה מבלוי רינדור הילדים או ירידת אפקטים React, לערך ששווה לערך הנוכחי State Hook אם אתה מעדכן (React משתמש `Object.is`.)

ישטרך לרנדר את הקומפוננטה הספציפית זו לפני יציאה מהפעולה. זה לא אמרו להיות מודאג React-שים לב שיכל להיות שלא ילק שלא לצורך "עמוק" לתוכה. אם אתה מבצע חישובים יקרים בזמן רינדור, ניתן למטב אותם עם `useMemo`.

Batching of state updates

React may group several state updates into a single re-render to improve performance. Normally, this improves performance and shouldn't affect your application's behavior.

Before React 18, only updates inside React event handlers were batched. Starting with React 18, [batching is enabled for all updates by default](#). Note that React makes sure that updates from several *different* user-initiated events -- for example, clicking a button twice -- are always processed separately and do not get batched. This prevents logical mistakes.

In the rare case that you need to force the DOM update to be applied synchronously, you may wrap it in `flushSync`. However, this can hurt performance so do this only where needed.

useEffect {#useeffect}

```
useEffect(didUpdate);
```

מקבלת פונקציה שמכילה קוד חיוני, שכנהה גורם לאפקט כלשהו.

של קומפוננטת פונקציה `body` main-טיימרים, לוגים, ותנועות לוואי אחרים לא מושרים בתוך ה `Mutations`, `subscriptions`, או `useEffect`-הפונקציה המועברת לו. Ai ציות לכך יגרום לבאים מבלבלים ואי עקבות במשתמש (המכונה של הרינדור של React).

תروع אחרי שהринדור מופיע על המסך. ניתן `useEffect`-במקום זאת, השתמש ב `timer` או `subscription` לעיתים קרובות, אפקטים יוצרים משאים שדורשים ניקוי לפני שהקומפוננטה עזבת את המסך, כמו `useEffect`-על מנת ליצור `subscription` תחזיר פונקציית נקון. לדוגמה, על מנת ליצור `useEffect`-על מנת לעשות זאת, הפונקציה המועברת לו `subscription`:

כברiorת מחדל, אפקטים רצים אחרי כל רינדור שמשתים, אבל ניתן לבחור להריז אותם רק [כשערכים מסויימים](#) [שונו](#).

ניקוי אפקט

הנקייה נתקיימת `subscription` על מנת לנקות את ה `subscription` הנטה. בדוגמה שולносן, ניקוי ה `subscription` מושג באמצעות `useEffect`-על מנת ליצור `subscription` תחזיר פונקציית נקון. לדוגמה, על מנת ליצור `useEffect`-על מנת לעשות זאת, הפונקציה המועברת לו `subscription`:

```
useEffect(() => {
  const subscription = props.source.subscribe();
  return () => {
    // ניקוי ה-subscription
    subscription.unsubscribe();
  };
});
```

פונקציית הנקון רצתה לפני הסרת הקומפוננטה ממשחק המשמש על מנת למנוע דיליפות זיכרון. בנוסף לכך, אם קומפוננטה מתrndנדרת מספר רב של פעמים (כמו שבדרך כלל קורה), **האפקט הקודם מנוקה לפני הרצת האפקט הבא**. בדוגמה שלנו, זה חדש נוצר בכל עדכון. על מנת להימנע מירית אפקט על כל עדכון, קרא את החלק הבא `subscription`-אומר ש

זמן אפקטים

נורה **לאחר** פרישה `componentDidMount`-`componentDidUpdate` ל-`useEffect` - בשונה מ-`subscriptions` בזמן אירוע נדחה. זה עושה את זה מתאים להרבה תופעות לווי, כמו הכנסת(`layout and paint`) וצביעה(`event handlers`, בגל שרוב סוג הבדיקה לא חסמים את הדף מלעדכו את המסקן.

שכליה למשתמש צריכה להיות נורה באופן סינכרוני DOM למרות זאת, לא ניתן לעכבר את כל האפקטים. לדוגמה, מוצפיה לפני הצבע הבא כך שהמשתמש לא יבחן בחומר עקבות חזותי. (ההבחנה דומה מבחינה קונסיסטואלית למאזנים לאירועים יש לו את `useLayoutEffect`. נוסף שנראה Hook מספק, פסיביים לעומת עומת פעילים). בשביל סוג האפקטים האלה וושונה ממנה כשהוא נורה, `useEffect` - אותה חתימה כ-

מתעכבר על שהדף נצבע, זה מובטח שהוא לפני רינדורים חדשים `useEffect` - אף על פי ש `<<<<<<` HEAD ===== Additionally, starting in React 18, the function passed to `useEffect` will fire synchronously **before** layout and paint when it's the result of a discrete user input such as a click, or when it's the result of an update wrapped in `flushSync`. This behavior allows the result of the effect to be observed by the event system, or by the caller of `flushSync`.

Note

This only affects the timing of when the function passed to `useEffect` is called - updates scheduled inside these effects are still deferred. This is different than `useLayoutEffect`, which fires the function and processes the updates inside of it immediately.

Even in cases where `useEffect` is deferred until after the browser has painted, it's guaranteed to fire before any new renders. React will always flush a previous render's effects before starting a new update.

84ad3308338e2bb819f4f24fa8e9dfeeffaa970b

יריות אפקט לפי תנאי

התנהגות הרגילה של אפקטים היא לירות את האפקט לאחר כל רינדור שהושלם. בדרך זו אפקט תמיד נוצר מחדש אם אחד משתנה משתנה-dependencies-מה.

מהקטע הקודם. אנחנו לא צריכים ליצור subscription-subscription-למרות זאת, זה יכול להיות יותר מדי במקרים מסוימים, כמו דוגמתה subscription source-שונה prop-חדש על כל עדכון, רק אם ה source-

זהו מערך של ערכים שהאפקט תלוי בהם. הדוגמה המעודכנת שלנו useEffect -על מנת לישם זאת, העבר ארגומנט שני ל useEffect כmo:

```
useEffect(
  () => {
    const subscription = props.source.subscribe();
    return () => {
      subscription.unsubscribe();
    };
  },
  [props.source],
);
```

משתנה props.source -ייצור מחדש רק כנש subscription-props.source.

הערה

-ו props כmo) הקומפוננטה scope-אם אתה משתמש באופטימיזציה זו, ודאי כי המערך מכיל את כל הערכים מ state-ששתנים לאורך זמן ושהאפקט משתמש בהם. אחרת, הקוד שלק יתייחס לערכים ישנים מrinndors קודמים. (**state**) . למד עוד על [על איך לטפל בפונקציות](#) ומה לעשות [בשערכי המערך משתנים בתדריות גבוהה מדי](#).

תוכל להעביר מערך ריק ([]) או props-state-unmount-ב) אם אתה רוצה להריץ אפקט ולנקוט אותו רק פעם אחת כך שהוא לא צריך לזרז state או props-שהאפקט שלק לא תלוי בשום מה רצ-React-כארוגמנט שני. זה אומר ל תמיד עבד מחדש-dependencies-מה. זה לא מטופל במקרה מיוחד – זה עבד כמו שמערך ה

בתוך האפקט תמיד יכilio את הערכים ההתחלתיים שלהם. בזמן state-props-אם אתה מעביר מערך ריק ([]), ה缄ודל מנטלי, יש componentWillUnmount -שהעברת [] כארוגמנט שני יותר קרוב ל React-פתרונות טובים יותר שעוזרים להימנע מהריצה מחדש של אפקטים בתדריות גבוהה מדי. בנוסף, אסור לשכוח ש עד לאחר שהדף נציג, אז עשיית עובדה נוספת נספהת היא פחות בעיה useEffect מעצב הריצה של

שלנו. הוא eslint-plugin-react-hooks -חלק exhaustive-deps מהbillet החביבה על שימוש בחוק שמוגדרים לא נכון ומציע תיקון dependencies מזהיר מפני.

לא מועבר כארוגמנטים לפונקציית האפקט. אבל באופן עקרוני, זה מה שהם מייצגים: כל ערך שמצוין dependencies-מערך ה בעתי, קומפיילר מתקדם יכול ליצור את המערך באופן dependencies. במקרה פונקציית האפקט צריך להופיע במערך הAutomati.

useContext {#usecontext}

```
const value = useContext(MyContext);
```

הנוכחי לאוטו context-ומחזיר את ערך ה `React.createContext()` -הערך המוחזר מ- `context`. מעל הקומפוננטה הקוראת בעץ `<MyContext.Provider value=<MyContext.Consumer>` של `value` נקבע על ידי ה `context`-provider.

האחרון של ה `value` מפעיל מרנדר עם Hook-`useContext`, מעל הקומפוננטה מתעדכו, ה- `<MyContext.Provider>` -ואותו ערך מועבר ל `MyContext` Provider.

עצמם `useContext`-צריך להיות אובייקט ה `useContext` אל תשכח שהארגון של

לא נכון `useContext(MyContext)` **נכון** `useContext(MyContext.Consumer)`

ישתנה. אם רינדור חדש של הקומפוננטה `useContext`-תמיד תתרנדר מחדש כשלעצמו `useContext` -קומפוננטה שקוראת ל `useContext`-הוא יקר, ניתן למטרב אותו על ידי שימוש ב `memoization`.

טיפול

הנוכחי ביחס ל `useContext` API Hooks, `useContext(MyContext)` אם אתה מכיר את ה `static contextType = MyContext`, או ל- `<MyContext.Consumer>` -במחלקה.

`useContext` לשינויים שלו. נדרש עדין `subscribe` ולעשות `useContext`-נתן לנו רק לקרוא את ה `<MyContext.Provider>` זה מעל בעץ על מנת לספק את הערך ל `useContext`.

נוספים Hooks

הבאים הם או צורות אחרות של הבסיסיים מה壽命 הקודם, או ככל שנדרך ובקשרו קצה ספציפיים. לא צריך לילחץ מללמוד אותם בהתחלה.

useReducer {#usereducer}

```
const [state, dispatch] = useReducer(reducer, initialArg, init);
```

הנוכחי ביחס `useReducer` את ה `reducer`, מסוג `useState` מקבל `(state, action) => newState` -אלטרנטיביה ל `dispatch`. זה כבר מוכר לך, Redux אם התעסקת בעבר עם `dispatch`.

מורכבת שמערכת מספר רב של תת-ערכים או כשה `state` כשייש לך לוגיקת `useReducer` עדיף על `useState` בדרך כלל גם נתן לנו למטרב ביצועים לקומפוננטות שמבצעות עדכונים מקומיים בגלו `useReducer` -הבודם `state` תלוי ב `state` מטה `useReducer` `dispatch` שניתן להעביר את callbacks.

מהקטע הקודם על `useReducer`-נקتب מחדש עם שימוש ב `useState`:

```
const initialState = {count: 0};

function reducer(state, action) {
  switch (action.type) {
    case 'increment':
      return {count: state.count + 1};
    case 'decrement':
      return {count: state.count - 1};
    default:
```

```

        throw new Error();
    }

function Counter() {
  const [state, dispatch] = useReducer(reducer, initialState);
  return (
    <>
      Count: {state.count}
      <button onClick={() => dispatch({type: 'decrement'})}>-</button>
      <button onClick={() => dispatch({type: 'increment'})}>+</button>
    </>
  );
}

```

הערה

היא קבועה ולא תשתנה ברינדרורים חוזרים. זאת הסיבה שזה בוטח `dispatch` מבטיח שזהות פונקציית `useEffect`-dependency או `useCallback`.

התחלתי-state-ציוו ה

ניתן לבוחר אחד מהם תלוי בשימוש. הדרך הפשטota ביוטר היא להעביר `useReducer` state. ישנו שני דרכים שונים לאותל התחלתי כארגומנט שני-state-ה:

```

const [state, dispatch] = useReducer(
  reducer,
  {count: initialCount}
);

```

הערה

הערך ההתחלתי לפעמים תלוי Redux-בניגוד `state = initialState` לא משתמש בモסקמת הארגומנט props-ב-Hook. ניתן לקרוא ל-`useReducer(reducer, undefined, reducer)` אבל זה לא מומלץ Redux על מנת לחקות את התנהגות.

אתחול עצמן

state-state-ארגומנט שלישי. ה `init` התחלתי בעצלהים. על מנת לעשות זאת, ניתן להעביר פונקציית `state` state ניתן גם ליצור `init(initialArg)`.

לאחר מכן state-זה גם שימושי לאיפוס ה-reducer-התחלתי מחוץ ל state-זה ניתן לנו לחץ את הלוגיקה לחישוב ה(`action`):

```

function init(initialCount) {
  return {count: initialCount};
}

function reducer(state, action) {
  switch (action.type) {

```

```

        case 'increment':
          return {count: state.count + 1};
        case 'decrement':
          return {count: state.count - 1};
        case 'reset':
          return init(action.payload);
        default:
          throw new Error();
      }
    }

function Counter({initialCount}) {
  const [state, dispatch] = useReducer(reducer, initialCount, init);
  return (
    <>
      Count: {state.count}
      <button
        onClick={() => dispatch({type: 'reset', payload: initialCount})}>
        Reset
      </button>
      <button onClick={() => dispatch({type: 'decrement'})}>-</button>
      <button onClick={() => dispatch({type: 'increment'})}>+</button>
    </>
  );
}

```

יציאה מ-dispatch

יצא מהפעולה מבלי רינדור הילדים או יריית אפקטים React, לערך שווה לערך הנוכחי State Hook אם אתה מעדכו (React משתמש באלגוריתם `Object.is`).

يُتَرَك لرندر الكومponeنتة السُّفْزِيَّيَّةُ هُوَ لِفَنِيْ يَصِيَّاهُ مَهَافِعُهُ . إِذَا أَمْرَرْتُ مَدَاجِعَ React-شِيمَ لِبَ شِيكُول لِلْحَيَّةِ شَلَّالَ لَأَنَّهُ لَمْ يَرُدْ عَلَى الْعَذْلِ . إِنَّمَا يَرُدُّ عَلَى الْعَذْلِ إِذَا أَمْرَرْتُ مَدَاجِعَ React-شِيمَ لِبَ شِيكُول لِلْحَيَّةِ شَلَّالَ لَأَنَّهُ لَمْ يَرُدْ عَلَى الْعَذْلِ .

useCallback {#usecallback}

```

const memoizedCallback = useCallback(
  () => {
    doSomething(a, b);
  },
  [a, b],
);

```

מחזיר `memoized callback`.

שימושה רק אם callback-של dependencies שמשתנה ורק אם callback-של dependencies. `useCallback` גרסה העבר לקומponeנטות ילדיים ממוטבות שמסתמכות על השוואת callbacks מ��נה. זה שימושי כoubles-אחד מהdependencies שמשתנה. `useCallback` לדוגמה) לפי אזכור על מנת למנוע רינדורים מיותרים `shouldComponentUpdate`.

`useCallback(fn, deps)` שווה ל `useMemo(() => fn, deps)`.

הערה

לא מועבר כארגומנטים לפונקציית האפקט. אבל באופן עקרוני, זה מה שהם מייצגים: כל ערך dependencies-הערך ה-`useMemo` מתקדם יכול ליצור את dependencies. שמצוין בתוך פונקציית האפקט צריך להופיע במערך dependencies-הערך באופן אוטומטי.

שלנו. הוא `eslint-plugin-react-hooks`-מחלקה exhaustive-deps שמליצים על שימוש בחוק dependencies-הערך לא נכון ומציע תיקון מזהיר מפני.

`useMemo` {#usememo}

```
const memoizedValue = useMemo(() => computeExpensiveValue(a, b), [a, b]);
```

מוחזיר ערך memoized.

כשאחד מה dependencies-הערך מוחשב מחדש רק את הערך `useMemo` של dependencies. העבר פונקציית "create" מוחזק ומערך dependencies-הערך שונה. מיטוב זה עוזר להימנע מחישובים יקרים בכל רינדור.

ריצה בזמן רינדור. אל תעשה דברים בתוכה שלא הייתה עשויה בדרך כלל בזמן רינדור. `useMemo`-זוכר כי הפונקציה שמועברת ל-`useEffect`, side effects-שייכים ל-`useMemo`. למשל,

אם סופק מערך כלשהו, ערך חדש יוחשב בכל רינדור.

יבחר "לשוכח" חלק React-כמיינט ביצועים, לא כערבות סמנטית. במקרה, יכול להיות ש `useMemo` ניתן להסתמך על כתוב את offscreen ויחשב אותם מחדש ברינדור הבא, למשל, על מנת לשחרר זיכרון לקומפוננטות memoized-המערכיים וה-`useEffect` אותו על מנת למטר ביצועים -- `useMemo` הקוד שולך כך שהוא יעבד בלי.

הערה

לא מועבר כארגומנטים לפונקציית האפקט. אבל באופן עקרוני, זה מה שהם מייצגים: כל ערך dependencies-הערך ה-`useMemo` מתקדם יכול ליצור את dependencies. שמצוין בתוך פונקציית האפקט צריך להופיע במערך dependencies-הערך באופן אוטומטי.

שלנו. הוא `eslint-plugin-react-hooks`-מחלקה exhaustive-deps שמליצים על שימוש בחוק dependencies-הערך לא נכון ומציע תיקון מזהיר מפני.

`useRef` {#useref}

```
const refContainer = useRef(initialValue);
```

`useRef` () שלו מואתחל לארגומנט המועבר `current`. -שנינו לשינוי שמאפיין ה `ref` מוחזיר אובייקט `initialValue`. האובייקט המוחזר יתמיד לכל מוחזר החיים של הקומפוננטה.

מקרה שימוש נפוץ הוא לגשת ליד כשרוצים:

```
function TextInputWithFocusButton() {
  const inputEl = useRef(null);
  const onButtonClick = () => {
    // `current` מצביע על אלמנט ה-`text input
```

```

        inputEl.current.focus();
    };
    return (
      <>
        <input ref={inputEl} type="text" />
        <button onClick={onButtonClick}>Focus the input</button>
      </>
    );
}

```

-הוא כמו "קופסה" שיכולה להחזיק ערך שנייתן לשינוי בתוך מאפיין ה `useRef.current`, במהותו

אם אתה מעביר אובייקט `DOM`-`ref` עם `<div ref={myRef}>`, React בעירך דרך `refs` ל-

refs-בפועל כרך `current` את מערך `refs` אולי אתה מכיר משתנה `node` ש相等情况 מקבילו `current`.

זהו שימושי לשימורת כל ערך שנייתן לשינוי בדומה לכך שהוא היה. `useRef()`-שימושי יותר מתכונת ה `ref` (למרות זאת במחלקות `instance fields` המשמש ב-

`useRef()` ויצירת אובייקט `{current: ...}` פשוט. ההבדל היחיד בין `useRef()` יוצר אובייקט זה עובד בגלל ש בכל רינדר ref ייתן לך אותו אובייקט `{useRef()...}`-עצמו הוא ש

לא גורם לרינדר מחדש. אם אתה `useRef()`-זכור ש `ref` מציין רק כשתוכן שלו משתנה. שינוי של המאפיין `useRef()` מציין או מנתק ref-מ-

callback ref במקומו.

useImperativeHandle {#useimperativehandle}

```
useImperativeHandle(ref, createHandle, [deps])
```

כמו תמיד, כדי `ref` שנחשר לkomponenot הורה בשימוש `useImperativeHandle`-`instance` מותאים אישית את הערך ה refs-ברוב המקרים. כדי להשתמש ב `useImperativeHandle` עם `forwardRef`:

```

function FancyInput(props, ref) {
  const inputRef = useRef();
  useImperativeHandle(ref, () => ({
    focus: () => {
      inputRef.current.focus();
    }
  }));
  return <input ref={inputRef} ... />;
}
FancyInput = forwardRef(FancyInput);

```

צריכה להיות מסוגלת לקרוא ל `<FancyInput ref={fancyInputRef}>` בדוגמה זו, komponent הורה שמרנדרת `fancyInputRef.current.focus()`.

useLayoutEffect {#uselayouteffect}

layout השמשanza על מנת לקרוא DOM. אבל הוא יורה באופן סינכרוני לאחר כל שינוי, `useEffect`-מאפיין זה זהה לישטפו באופן סינכרוני, לפני `useLayoutEffect` ולרנדר מחדש באופן סינכרוני. עדכונים מתאימים בתוך DOM-`MutationObserver` יש האפשרות לצבעו.

הסטודנטטי מתי שאפשר על מנת להימנע מחסימת עדכנים ויזואליים `useEffect` העדר את

טיפול

יורה באותו קצב כמו `componentDidMount`- אם אתה מזין קוד מקומפוננטת מחלוקת, שים לב ש- `useLayoutEffect` **קודם** ולנסות את `useEffect` **לעתומת זאת, אנו ממליצים להתחילה עם** `useLayoutEffect`. רק אם זה יוצר בעיה.

-יכולים לזרע עד ש- `useEffect` **useLayoutEffect** ו- `useLayoutEffect` שים לב שגמ, ואם אתה משתמש ב-`server rendering`, מכך `useLayoutEffect` מזהיר `server-rendered` מקומפוננטה שהיא-React-הוור. זאת הסיבה ש- `useLayoutEffect` או המtan עם הצגת, (אם זה לא נחוץ לרינדור הראשון) `useEffect` -על מנת לתקן זאת, או שתעביר את הלוגיקה ל-`useLayoutEffect` (אם זה HTML-הוור נראית שבור עד לאחר רינדור הקליינט).

`showChild` רנדר אותה בתנאי עם `layout effects` מ-`server-rendered` HTML, על מנת להציג קומפוננטה שצריכה בדרכן, ממש. `useEffect(() => { setShowChild(true); }, []);` ועכבר את הצגת Um המשמש לא מופיע שבור לפני הידריצה.

useDebugValue {#usedebugvalue}

`useDebugValue(value)`

מוותאים אישית ב-`label` `useDebugValue` על מנת להציג -ניתן להשתמש ב-React DevTools.

המשלב `useFriendStatus` **לדוגמה, שמתואר ב"בנייה Hooks":**

```
function useFriendStatus(friendID) {
  const [isOnline, setIsOnline] = useState(null);
  // ...
  // זהה -olid ה-DevTools Hook
  // לדוגמה "FriendStatus: Online"
  useDebugValue(isOnline ? 'Online' : 'Offline');
  return isOnline;
}
```

טיפול

מוותאים אישית שחלק Hooks-מוותאים אישית. זה נחוץ במיוחד ל-`debug` לכל Hook במיוחד אם הוא לא ממליצים להוסיף ערכיו מספריות מסוימות משותפות.

דוחה debug של ערכי formatting

נבדק Hook-לערך יכול להיות פעולה יקרה. זה גם לא נחוץ אלא אם ה-`formatting` במספרים מסוימים לבצע.

Hooks-כפרמטר שני אופציוני. קוראים לפונקציה זו רק אם ה-`formatting` מקבל פונקציית `useDebugValue` מסיבה זו כפרטמר וצריכה להחזיר ערך הצגה שעבר debug-בדיקות. היא מקבלת את ערך ה-

באופן לא נחוצ על ידי `Date` יכול להימנע מלקרוא לפונקציית `toDatestring` מותאמת אישית שמחזיר ערך Hook לדוגמה הבא:

```
useDebugValue(date, date => date.toDateString());
```

useDeferredValue {#usedeferredvalue}

```
const deferredValue = useDeferredValue(value);
```

`useDeferredValue` מקבלת ערך ו>Returns מחדש ערך זה ש�פשר לו ל-defer עדות. אם הレンר הנוכחי הוא תוצאה של עדות א-gent, כמו Input, React י.Return את הערך הקודם ולאחר מכןレンר את החדש לאחר שהעדות א-gent נסתיימה.

הhook דומה לhookspace userspace אשר משתמשים ב-debouncing או throttling כדי(defer עדות. יתרונות של `useDeferredValue` הם שReact י.earn את עדותו momentously (בכל מקרה אחר) ולאesperando זמן-arbitrary, וכן ש-deferred ערך יכול ל-suspendsenza triggeringUnexpected fallback content.

Memoizing deferred children

`useDeferredValue` רק(defers את הערך שאתה מסתובב אליו. אם אתה רוצה למנוע את ה-children component מ-re-rendering במהלך עדות א-gent, אתה חייב גם ל-memoize אותו עם `React.memo` או `React.useMemo`:

```
function Typeahead() {
  const query = useSearchQuery('');
  const deferredQuery = useDeferredValue(query);

  // Memoizing tells React to only re-render when deferredQuery changes,
  // not when query changes.
  const suggestions = useMemo(() =>
    <SearchSuggestions query={deferredQuery} />,
    [deferredQuery]
  );

  return (
    <>
      <SearchInput query={query} />
      <Suspense fallback="Loading results...">
        {suggestions}
      </Suspense>
    </>
  );
}
```

Memoizing the children tells React that it only needs to re-render them when `deferredQuery` changes and not when `query` changes. This caveat is not unique to `useDeferredValue`, and it's the same pattern you would use with similar hooks that use debouncing or throttling.

useTransition {#usetransition}

```
const [isPending, startTransition] = useTransition();
```

Returns a stateful value for the pending state of the transition, and a function to start it.

`startTransition` lets you mark updates in the provided callback as transitions:

```
startTransition(() => {
  setCount(count + 1);
})
```

`isPending` indicates when a transition is active to show a pending state:

```
function App() {
  const [isPending, startTransition] = useTransition();
  const [count, setCount] = useState(0);

  function handleClick() {
    startTransition(() => {
      setCount(c => c + 1);
    })
  }

  return (
    <div>
      {isPending && <Spinner />}
      <button onClick={handleClick}>{count}</button>
    </div>
  );
}
```

Note:

Updates in a transition yield to more urgent updates such as clicks.

Updates in a transitions will not show a fallback for re-suspended content. This allows the user to continue interacting with the current content while rendering the update.

useId {#useid}

```
const id = useId();
```

`useId` is a hook for generating unique IDs that are stable across the server and client, while avoiding hydration mismatches.

Note

`useId` is **not** for generating `keys` in a list. Keys should be generated from your data.

For a basic example, pass the `id` directly to the elements that need it:

```
function Checkbox() {
  const id = userId();
  return (
    <>
      <label htmlFor={id}>Do you like React?</label>
      <input id={id} type="checkbox" name="react"/>
    </>
  );
};
```

For multiple IDs in the same component, append a suffix using the same `id`:

```
function NameFields() {
  const id = userId();
  return (
    <div>
      <label htmlFor={id + '-firstName'}>First Name</label>
      <div>
        <input id={id + '-firstName'} type="text" />
      </div>
      <label htmlFor={id + '-lastName'}>Last Name</label>
      <div>
        <input id={id + '-lastName'} type="text" />
      </div>
    </div>
  );
};
```

Note:

`useId` generates a string that includes the `:` token. This helps ensure that the token is unique, but is not supported in CSS selectors or APIs like `querySelectorAll`.

`useId` supports an `identifierPrefix` to prevent collisions in multi-root apps. To configure, see the options for `hydrateRoot` and `ReactDOMServer`.

Library Hooks

The following Hooks are provided for library authors to integrate libraries deeply into the React model, and are not typically used in application code.

`useSyncExternalStore` {#usesyncexternalstore}

```
const state = useSyncExternalStore(subscribe, getSnapshot[], getServerSnapshot);
```

`useSyncExternalStore` is a hook recommended for reading and subscribing from external data sources in a way that's compatible with concurrent rendering features like selective hydration and time slicing.

This method returns the value of the store and accepts three arguments:

- `subscribe`: function to register a callback that is called whenever the store changes.
- `getSnapshot`: function that returns the current value of the store.

- `getServerSnapshot` : function that returns the snapshot used during server rendering.

The most basic example simply subscribes to the entire store:

```
const state = useSyncExternalStore(store.subscribe, store.getSnapshot);
```

However, you can also subscribe to a specific field:

```
const selectedField = useSyncExternalStore(
  store.subscribe,
  () => store.getSnapshot().selectedField,
);
```

When server rendering, you must serialize the store value used on the server, and provide it to `useSyncExternalStore`. React will use this snapshot during hydration to prevent server mismatches:

```
const selectedField = useSyncExternalStore(
  store.subscribe,
  () => store.getSnapshot().selectedField,
  () => INITIAL_SERVER_SNAPSHOT.selectedField,
);
```

Note:

`getSnapshot` must return a cached value. If `getSnapshot` is called multiple times in a row, it must return the same exact value unless there was a store update in between.

A shim is provided for supporting multiple React versions published as `use-sync-external-store/shim`. This shim will prefer `useSyncExternalStore` when available, and fallback to a user-space implementation when it's not.

As a convenience, we also provide a version of the API with automatic support for memoizing the result of `getSnapshot` published as `use-sync-external-store/with-selector`.

useInsertionEffect {#useinsertioneffect}

```
useInsertionEffect(didUpdate);
```

The signature is identical to `useEffect`, but it fires synchronously *before* all DOM mutations. Use this to inject styles into the DOM before reading layout in `useLayoutEffect`. Since this hook is limited in scope, this hook does not have access to refs and cannot schedule updates.

Note:

`useInsertionEffect` should be limited to css-in-js library authors. Prefer `useEffect` or `useLayoutEffect` instead.

[Go to TOC](#)

חוקי Hooks

ופיצ'רים אחרים של ריאקט מבליל כתוב state - הם נתונים לנו להשתמש ב. הם תוספת חדשה ב React 16.8. מחלוקת.

JavaScript linting אבל אתה צריך לנוהג על פי שני חוקים כאשרת משתמש בהם. אנו מספקים פלאגין Hooks, הם פונקציות שאותן חוקים אלו אוטומטיות:

רַק בְּרוֹמָה הָעֲלֵיָנוּת Hooks - נִיתָן לְקָרוֹא ל

ברמה Hooks - או פונקציות מקווננות. במקום זאת, תמיד השימוש ב if כמו conditions, בתוך לופיהם Hooks - אל תקרא נקאים באותו Hooks - מוקדמים. על ידי ביצוע כל זה, אתה מבטיח שהעלינהה של פונקציית הריאקט שלך, לפני כל בין קראות רבותות של state של Hooks מתרנדרת. זה מה שמאפשר לריאקט לשמור את ה useState () אם אתה סקרן, נסביר זאת לעומק למטה useEffect .

```
#only-call-hooks-from-react-functions}
```

רַגְילָות. בָּמֶקְומָן זָאת, אַתָּה יְכֹל JavaScript מִתּוֹךְ פּוֹנְקְצִיוֹת Hooks - אל תָּקָרָא ל

- מיתוך קומפוננטות פונקציה של ריאקט Hooks - לקרוא ל.
- מותאמים אישית (למד עליהם בעמוד הבא) Hooks מיתוך Hooks - לקרוא ל.

בתוך קומפוננטה היא ברורה לעין מקורו שלה stateful על ידי ביצוע כל זה, אתה מבטיח שככל לוגיקה שהיא.

ESLint פלאגין

האוכף אוטומטית שני חוקים אלה. אתה יכול להוסיף את eslint-plugin-react-hooks שנקרא שחררנו פלאגין: הפלгин זהה לפרויקט שלך אם תרצה לנסות אותו:

פלאגין זה כולל כבירית מחדל ב Create React App.

```
npm install eslint-plugin-react-hooks --save-dev
```

```
// -קונפיגורציית ESLint
{
  "plugins": [
    // ...
    "react-hooks"
  ],
  "rules": {
    // ...
    "react-hooks/rules-of-hooks": "error", // בודק חוקי Hooks
    "react-hooks/exhaustive-deps": "warn" // בודק effect dependencies
  }
}
```

משל עבשו. בעמוד זה, נסביר בהסביר את הסיבות שמחורי Hooks אתה יכול לדלג לעמוד הבא שסביר לך לכתוב חוקים אלה.

הסבר

בתוך קומפוננטה יחידה או State Effect Hooks כמו [שלמדנו מוקדם](#), אנחנו יכולים להשתמש במספר

```
function Form() {
  // 1. Create name state
  const [name, setName] = useState('Mary');

  // 2. Use an effect for persisting the form
  useEffect(function persistForm() {
    localStorage.setItem('formData', name);
  });

  // 3. Create surname state
  const [surname, setSurname] = useState('Poppins');

  // 4. Update title effect
  useEffect(function updateTitle() {
    document.title = name + ' ' + surname;
  });

  // ...
}
```

-התשובה היא **שריאקט משתמש על סדר קריאות ה useState ?** מתחאים לאיו קריית state או או ריאקט ידע איזה Hooks. הוא אותו דבר בכל רינדור Hooks -הדוגמה שלנו עובדת בגלל שסדר קריות ה

```
// -----
// First render
// -----
useState('Mary')           // 1. Create name state 'Mary'
useEffect(persistForm)     // 2. Add an effect for persisting the form
useState('Poppins')        // 3. Create surname state 'Poppins'
useEffect(updateTitle)      // 4. Update title effect

// -----
// Second render
// -----
useState('Mary')           // 1. Create name state (מתעלמים מהקילט)
useEffect(persistForm)     // 2. Replace the effect for persisting the form
useState('Poppins')        // 3. Create surname state (מתעלמים מהקילט)
useEffect(updateTitle)      // 4. Update title effect

// ...
```

מוקמי עם כל אחד מהם. אבל מה קורה אם state הוא שווה בין רינדורים, ריאקט יכול לצרף Hook כל עוד הסדר של קריות persistForm (לדוגמה, האקפט) Hook condition?

```
// ----- Hook condition
if (name !== '') {
  useEffect(function persistForm() {
    localStorage.setItem('formData', name);
  });
}
```

זה. למורת זאת, יכול להיות שברינדור Hook בrinidor הראשון, אז אנחנו מרים true הוא ' ' - condition name !== ' ' זה בזמנ Hook עכשו שאנחנו מודגים על . -ייפוך ל condition -הבא המשמש ינקה את הטופס, וכתוכאה מכך ה- משטנה Hook רינדור, הסדר של קריאות:

```
useState('Mary')           // 1. קריית המשתנה
// useEffect(persistForm) // 2. דילגנו על זה Hook!
useState('Poppins')        // 3. נכשל בלקראן את המשתנה. (אבל היה
useEffect(updateTitle)     // 4. נכשל בהצליף את האפקט. (אבל היה
```

השניה בקומפוננטה Hook -ריאקט ציפה לקריאה השניה של Hook בדיק כmo בrinidor הקודם, אבל זה לא דומה יותר. מנוקדה זו, כל קריאת persistForm הזו תהיה תואמת לאפקט אחריו האחת שדילגנו עליה תזוז באחד, דבר שוביל לבאגים

ברמה העלונה של הקומפוננטות שלנו. אם אנחנו רוצים להריץ אפקט מותנה, אנחנו Hooks -זאת הסיבה שחייב לקרוא ל- שלנו Hook -יכולים לשים תנאי זה בתוך ה-

```
useEffect(function persistForm() {
  // 👍 אנו לא עוברים על החוק הראשון יותר
  if (name !== '') {
    localStorage.setItem('formData', name);
  }
});
```

עובדים בדרך Hooks אבל עכשו אתה גם ידע למה lint. שים לב שאתה לא צריך לדאוג מבעה זו אם אתה משתמש בחוק זו, ואילו בעיות החוק הזאת מונע.

הצעדים הבאים

שמספקים על ידי Hooks מותאים אישית נוטנים לך לשלב Hooks! ממשלו Hooks סוף כל סוף, אנו מוכנים ללמידה על כתיבת בין קומפוננטות שונות stateful וריאקט לתוך האבסטרקציות שלך, ולעשות שימוש חוזר בלוגיקה שהיא

[Go to TOC](#)

שימוש ב State Hook

ambil לכתוב מחלוקת ופיצרים אחרים של state - הםאפשרים שימוש ב state-הם תוסף חדש ב React 16.8.

עם הדוגמה הבאה [הדף המקורי](#) הדף הקודם הציג:

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

function Example() {
  // הגדיר state אחד, שנקרא לו "count"
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

על ידי השווה בין קוד זה לקוד המקורי שמשתמש במחלוקת Hooks נתחילה ללמידה על.

דוגמה מקבילה עם שימוש במחלוקת

בעבר, קוד זה אמרו להירות מוכר React-אם השתמשה במחלוקת ב:

```
class Example extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0
    };
  }

  render() {
    return (
      <div>
        <p>You clicked {this.state.count} times</p>
        <button onClick={() => this.setState({ count: this.state.count + 1 })}>
          Click me
        </button>
      </div>
    );
  }
}
```

-כשהמשתמש לווח על כפטור על ידי קרייה ל state.count -ואנו מוסיפים 1 ל { count: 0 } מתחילה בתור זה. נשתמש בחלקים מהחלוקת זו לאורך העמוד. `this.setState()`.

הערה

כאן במקום דוגמה יותר ריאלית. זה במטרה לעזור לנו להתמקד counter - אתה בטח תזהה למה אנו משתמשים בזמן שאנו רק מתחילה עם API - בـ Hooks.

Hooks וקומפוננטות פונקציונליות

נראות כך-React-כתזורת, קומפוננטות פונקציונליות ב:

```
const Example = (props) => {
  // נניתן להשתמש ב- Hooks!
  return <div />;
}
```

או כך:

```
function Example(props) {
  // נניתן להשתמש ב- Hooks!
  return <div />;
}
```

בתוכם, אז state - אנו יכולים מציגים את היכולת להשתמש ב "stateless components" - יכול להיות שהכרת אותו כאנחנו מעדיפים את השם "קומפוננטות פונקציונליות".

לא עובדים בתוך מחלקות. אך ניתן להשתמש בהם במקום כתיבת מחלקות.

מה זה Hook?

הדוגמה החדשת שלנו מתחילה בייבוא של Hook useState מ-React:

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

function Example() {
  // ...
}
```

Hook מה זה Hook? Hook זה פונקציה מיוחדת שנوتנת לך "להתחבר" לפיצ'רים של state. להסביר Hook הוא useState. לדוגמה, אנו יכולים להשתמש Hooks לקומפוננטות פונקציונליות. אנו נלמד על state שנוטן לך להוסיף אחרים בהמשך.

בעבר נדרש להמיר state אם אתה כותב קומפוננטה פונקציונלית ומגלת שאתה צריך להוסיף להוסיף לה Hook? מהי השימוש בתוך הקומפוננטה הפונקציונלית הקיימת. אנחנו הולכים לעשות את זה Hook - אותה למחלקה. עשו אותה יכול להשתמש בתוך קומפוננטה. אנו נלמד Hooks - ישנים כמה חוקים מיוחדים שמכתיבים איפה אפשר ואיפה אי אפשר להשתמש בתוך קומפוננטה. עלייהם בו - חוקי Hooks.

הערה:

בתוך קומפוננטה. אנו נלמד Hooks - ישנים כמה חוקים מיוחדים שמכתיבים איפה אפשר ואיפה אי אפשר להשתמש בתוך קומפוננטה. עלייהם בו - חוקי Hooks.

הגדרת משתנה state

בבנייה `{ count: 0 }` - ל `this.state` ב- `count` במחלקה, אנו מאתחלים את:

```
class Example extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0
    };
}
```

-במקום, נקרא ל `this.state` אז אין לנו דרך להציגו או לקרוא באמצעות `this` בkomponennta הפונקציונלית, אין לנו `useState` hook שירות מתוק komponennta שלנו:

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

function Example() {
  // זכרנו state זначר לא count הצהרת המשתנה
  const [count, setCount] = useState(0);
```

אך נוכל לקרוא לו בכל שם אחר, `count` המשתנה שלנו נקרא `state`. **מה קרייה ל `useState` זה מגדיר "משתנה state"?** זה מגדיר "משתנה state".
היא דרך חדשה להשיג את אותן מטרות `useState` -- זהי דרך "לשמר" על ערכים בין קרייאות פונקציה. כמו `banana`. בדומה בכל, מעתנים "נעלים" כשהפונקציה מסיימת את פעולתה אבל משתני `this.state` שהשכנו באמצעות `useState` נשמרים על ידי `React`.

התחלתית. שלא כמו מחלקות `state` - הוא ה `hook` היחיד ל `useState()` - **כמה עיברים ל `useState`?** הקלט היחיד ל `useState()` הוא `state` - מה `count` מעתנים ב- `useState` לא חייב להיות אובייקט. אנו יכולים לשמור מספר או מחרוזת אם זה מה שאנו צריכים. בדוגמה שלנו, אנו רוצים מספר `state` אם נרצה לשמור שני ערכים). התחלתית למשתנה שלנו `state` - שיפיע בהתאם למספר הנקודות של המשתמש, אז נקבע `0` (פעמים `useState()` - נקרא ל `state`, שונים ב-

העכשווי ופונקציה שמעדכנת אותו. זהי הסיבה שאנחנו כותבים `useState(state)` - מה `state` מוחזר זוג של ערכים: `state` מה `useState`, `setCount` - וזה דומה ל `this.setState()` - ול `this.state.count` במחלקה, `const [count, setCount] = useState()`.
שהשתמשנו בו, נחזור אליו **בתחתית העמוד זהה** - חוץ מהעובדת שמקבלים אותם בזוג. אם אתה לא מכיר את ה

עושים, הדוגמה שלנו אמורה להירות יותר הגיוני hook `useState` עכשווי שאנו יודעים מה:

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

function Example() {
  // זכרנו state זначר לא count הצהרת המשתנה
  const [count, setCount] = useState(0);
```

זכור את הערך הנוכחי בין רינדררים, ויספק את `React`. `useState` החדש בשם `state` אנו מגדירים משתנה `count`, `0` וקובעים אותו כ- `React`. `useState` מעדכן את הערך האחרון לפונקציה שלנו. אם אנחנו רוצים לעדכן את הערך האחרון, נקרא ל `setCount`.

הערה

במקרים לא נקרא `useState` יכול להיות שאתה תוהה: מדוע `createState`?

ונוצר בפעם הראשונה רק כשהקובומפוננטה שלנו מורונדרת. במהלך הרינדורים state - לא יהיה מדובר בכלל שה"Create" בפועל ישנה גם סיבה למה שמות "state" הנוחci. אחרות לא זה לא היה state - נתונים לנו את ה useState, הباءים נלמד מה מהוחר יותר בחוקי Hooks.

קריאה state

- הנוחci במחלקה, אנו קוראים מ state - כאשרנו רוצים להציג את ה this.state.count :

```
<p>You clicked {this.state.count} times</p>
```

ישירות state - בפונקציה, ניתן להשתמש ב:

```
<p>You clicked {count} times</p>
```

עדכון state

על מנת לעדכן את this.setState() - במחלקה, אנו צריכים לקרוא ל count :

```
<button onClick={() => this.setState({ count: this.state.count + 1 })}>
  Click me
</button>
```

במשתנים או אנחנו לא צריכים את this(count) בפונקציה, יש לנו את this :

```
<button onClick={() => setCount(count + 1)}>
  Click me
</button>
```

סיכום

עכשו נסכם מה למדנו שורה אחרי שורה ונבדוק את ההבנה שלנו.

```
1: import React, { useState } from 'react';
2:
3: function Example() {
4:   const [count, setCount] = useState(0);
5:
6:   return (
7:     <div>
8:       <p>You clicked {count} times</p>
9:       <button onClick={() => setCount(count + 1)}>
10:         Click me
11:       </button>
12:     </div>
13:   );
14: }
```

- מוקמי בתוך קומפוננטה פונקציונלית state זה נותן לנו לשמר hook-מ React. **שורה 1:** אנו מייבאים את useState.

- שורה 4:** בתוך הקומפוננטה `Example` - חדש על ידי קרייה ל `state` אנחנו מגדירים משתנה, `useState` hook. זה שהוא שומר את מספר הלחיצות על `count` מחזיר זוג של ערכים, שלהם ניתן שמות. אנו קוראים למשתנה שלנו הערך השני שהוא פונקציה. אנחנו מגדירים אותו כ- 0 על ידי העברת 0 קלט היחיד של `setCount`. **שורה 9:** כשהשתמשנו בו כפטור. אנחנו מגדירים אותו כ- 0 על ידי העברת 0 קלט היחיד של `setCount`.
- ירנדר מחדש את React עם ערך חדש. לאחר מכן `setCount` המשמש לווח על ההפטור, נקרא ל- `count` החדש אליה.

יכול להיות שהוא נראה כמו הרבה לעכל בפעם הראשונה. לא צריך לומר מהר! אם אתה מרגיש אבוד בהסביר, הסתכל על הקוד לעיל ועובד במחלקות, וסתכל על `state` ונסה שוב לקרוא אותו מלמעלה למטה. אנחנו מבטיחים שברגע שתנסה "לשכחה" כיצד הקוד הזה עם עניינים רעננות, זה יהיה הגיוני.

טיפ: מה המשמעות של סוגרים מרובעים?

חדש `state` יכול להיות ששמתו לב לסוגרים המרובעים כשהגדרנו משתנה:

```
const [count, setCount] = useState(0);
```

אתה יכול לקרוא להם בכל שם שאתה רוצה. השמות שמאלם לא חלק מה:

```
const [fruit, setFruit] = useState('banana');
```

זה אומר שאנו מכינים שני משתנים חדשים `fruit` ו- `setFruit`, וזה מקביל קוד הבא: הוא הראשון שווה לערך הראשון שmorphoz מ `useState`, והשני. זה מושג באמצעות `array destructuring`.

```
var fruitStateVariable = useState('banana'); // מודיער זוג הפריט הראשון בזוג
var fruit = fruitStateVariable[0]; // הפריט השני בזוג
var setFruit = fruitStateVariable[1]; // הפריט השלישי בזוג
```

זהמחזיר זוג - מערך עם שני פריטים. הפריט הראשון הוא הערך הנוכחי, `useState` עם `state` כאשרנו מגדירים משתנה והשני הוא פונקציה שנונתת לנו לעדכן אותו. שימוש בו- `[0]` ו- `[1]` על מנת לגשת אליהם זה טיפה מבלבל בגלל שיש מקום `array destructuring` - להם משמעות ספציפית. זה למה אנחנו משתמשים ב-

הערה

בגלל שאנחנו לא מעבירים `useState` - יודע איך קומפוננטה מתאימה ל- React יכול להיות שאתה סקרן לדעת איך נעה על `this` ואחרות רבות בעמוד שעונה על שאלות נפוצות-React. חזרה ל דברים כמו

טיפ: שימוש במספר משתני state

היא גם שימושית, בgalל שהוא נותנת לנו אפשרות לחת `[something, setSomething]` כזוג של `state` הגדרת משתני שונים אם אנחנו רוצים להשתמש state מאחד עם שמות שונים למשתני:

```
function ExampleWithManyStates() {
  // הגדרת מספר משתני state!
  const [age, setAge] = useState(42);
  const [fruit, setFruit] = useState('banana');
  const [todos, setTodos] = useState([{ text: 'Learn Hooks' }]);
}
```

במשתנים מקומיים, אנחנו יכולים לעדכן אותם לחוד `age`, `fruit`, ו- `todos` בקומפוננטה שלעיל, יש לנו:

```
function handleOrangeClick() {
  // דומה ל this.setState({ fruit: 'orange' })
  setFruit('orange');
}
```

יכולים לשמור אובייקטים ומערכות מזוין, כך שאתה יכול עדין state משתני. state אתה לא חייב להשתמש בהרבה משתני תמיד מחליף אותו במקום state במחלקה, עדכוו משתנה `this.setState` -לקבץ מידע הקשור ביחיד. לעומת זאת, ברגע למזוג אותו.

עצמאיתם ב- [עמוד שאלות נפוצות](#) אנחנו מספקים המלצות נוספות על פיצול משתני.

השלבים הבאים

-אנו הולכים להתייחס אליו גם כה `useState`. שנקרא Hooks שמסופקים על ידי React -בעמוד זה למדנו על אחד מה מה שעשינו בפעם הראשונה אי פעם -React-מקומי לקומפוננטות פונקציונליות בstate זה נותן לנו להוסף "State Hook".

מתוך קומפוננטות React הם פונקציות שנותנות לך "להתחבר" לפיצרים של Hooks. Hooks למדנו גם קצת על מה הם שלא ראיינו עדין Hooks ויש עוד `useEffect` פונקציונליות. השמות שלהם תמיד תמיד מתחילה עם `use`.

הוא נותן לך לבצע "תופעות לוואי" בקומפוננטות, דומה [הבא Hook -עכשו ממשיך על ידי למידה של ה](#). למתחדות מחזור חיים במחלקות.

[Go to TOC](#)

How to Contribute

React is one of Facebook's first open source projects that is both under very active development and is also being used to ship code to everybody on [facebook.com](#). We're still working out the kinks to make contributing to this project as easy and transparent as possible, but we're not quite there yet. Hopefully this document makes the process for contributing clear and answers some questions that you may have.

Code of Conduct {#code-of-conduct}

Facebook has adopted the [Contributor Covenant](#) as its Code of Conduct, and we expect project participants to adhere to it. Please read [the full text](#) so that you can understand what actions will and will not be tolerated.

Open Development

All work on React happens directly on [GitHub](#). Both core team members and external contributors send pull requests which go through the same review process.

Semantic Versioning

React follows [semantic versioning](#). We release patch versions for critical bugfixes, minor versions for new features or non-essential changes, and major versions for any breaking changes. When we make breaking changes, we also introduce deprecation warnings in a minor version so that our users learn about the upcoming changes and migrate their code in advance. Learn more about our commitment to stability and incremental migration in [our versioning policy](#).

Every significant change is documented in the [changelog file](#).

Branch Organization

Submit all changes directly to the `main branch`. We don't use separate branches for development or for upcoming releases. We do our best to keep `main` in good shape, with all tests passing.

Code that lands in `main` must be compatible with the latest stable release. It may contain additional features, but no breaking changes. We should be able to release a new minor version from the tip of `main` at any time.

Feature Flags

To keep the `main` branch in a releasable state, breaking changes and experimental features must be gated behind a feature flag.

Feature flags are defined in `packages/shared/ReactFeatureFlags.js`. Some builds of React may enable different sets of feature flags; for example, the React Native build may be configured differently than React DOM. These flags are found in `packages/shared/forks`. Feature flags are statically typed by Flow, so you can run `yarn flow` to confirm that you've updated all the necessary files.

React's build system will strip out disabled feature branches before publishing. A continuous integration job runs on every commit to check for changes in bundle size. You can use the change in size as a signal that a feature was gated correctly.

Bugs

Where to Find Known Issues

We are using [GitHub Issues](#) for our public bugs. We keep a close eye on this and try to make it clear when we have an internal fix in progress. Before filing a new task, try to make sure your problem doesn't already exist.

Reporting New Issues

The best way to get your bug fixed is to provide a reduced test case. This [JSFiddle template](#) is a great starting point.

Security Bugs

Facebook has a [bounty program](#) for the safe disclosure of security bugs. With that in mind, please do not file public issues; go through the process outlined on that page.

How to Get in Touch

- IRC: [#reactjs on freenode](#)
- [Discussion forums](#)

There is also [an active community of React users on the Discord chat platform](#) in case you need help with React.

Proposing a Change

If you intend to change the public API, or make any non-trivial changes to the implementation, we recommend [filing an issue](#). This lets us reach an agreement on your proposal before you put significant effort into it.

If you're only fixing a bug, it's fine to submit a pull request right away but we still recommend to file an issue detailing what you're fixing. This is helpful in case we don't accept that specific fix but want to keep track of the issue.

Your First Pull Request

Working on your first Pull Request? You can learn how from this free video series:

[How to Contribute to an Open Source Project on GitHub](#)

To help you get your feet wet and get you familiar with our contribution process, we have a list of [good first issues](#) that contain bugs that have a relatively limited scope. This is a great place to get started.

If you decide to fix an issue, please be sure to check the comment thread in case somebody is already working on a fix. If nobody is working on it at the moment, please leave a comment stating that you intend to work on it so other people don't accidentally duplicate your effort.

If somebody claims an issue but doesn't follow up for more than two weeks, it's fine to take it over but you should still leave a comment.

Sending a Pull Request

The core team is monitoring for pull requests. We will review your pull request and either merge it, request changes to it, or close it with an explanation. For API changes we may need to fix our internal uses at Facebook.com, which could cause some delay. We'll do our best to provide updates and feedback throughout the process.

Before submitting a pull request, please make sure the following is done:

1. Fork [the repository](#) and create your branch from `main`.
2. Run `yarn` in the repository root.
3. If you've fixed a bug or added code that should be tested, add tests!
4. Ensure the test suite passes (`yarn test`). Tip: `yarn test --watch TestName` is helpful in development.
5. Run `yarn test --prod` to test in the production environment.
6. If you need a debugger, run `yarn debug-test --watch TestName`, open `chrome://inspect`, and press "Inspect".
7. Format your code with `prettier` (`yarn prettier`).
8. Make sure your code lints (`yarn lint`). Tip: `yarn linc` to only check changed files.
9. Run the `Flow` typechecks (`yarn flow`).
10. If you haven't already, complete the CLA.

Contributor License Agreement (CLA)

In order to accept your pull request, we need you to submit a CLA. You only need to do this once, so if you've done this for another Facebook open source project, you're good to go. If you are submitting a pull request for the first time, just let us know that you have completed the CLA and we can cross-check with your GitHub username.

[Complete your CLA here.](#)

Contribution Prerequisites

- You have [Node](#) installed at LTS and [Yarn](#) at v1.2.0+.
- You have [JDK](#) installed.
- You have `gcc` installed or are comfortable installing a compiler if needed. Some of our dependencies may require a compilation step. On OS X, the Xcode Command Line Tools will cover this. On Ubuntu, `apt-get install build-essential` will install the required packages. Similar commands should work on other Linux distros. Windows will require some additional steps, see the [node-gyp installation instructions](#) for details.

- You are familiar with Git.

Development Workflow

After cloning React, run `yarn` to fetch its dependencies. Then, you can run several commands:

- `yarn lint` checks the code style.
- `yarn linc` is like `yarn lint` but faster because it only checks files that differ in your branch.
- `yarn test` runs the complete test suite.
- `yarn test --watch` runs an interactive test watcher.
- `yarn test --prod` runs tests in the production environment.
- `yarn test <pattern>` runs tests with matching filenames.
- `yarn debug-test` is just like `yarn test` but with a debugger. Open `chrome://inspect` and press "Inspect".
- `yarn flow` runs the Flow typechecks.
- `yarn build` creates a `build` folder with all the packages.
- `yarn build react/index,react-dom/index --type=UMD` creates UMD builds of just React and ReactDOM.

We recommend running `yarn test` (or its variations above) to make sure you don't introduce any regressions as you work on your change. However, it can be handy to try your build of React in a real project.

First, run `yarn build`. This will produce pre-built bundles in `build` folder, as well as prepare npm packages inside `build/packages`.

The easiest way to try your changes is to run `yarn build react/index,react-dom/index --type=UMD` and then open `fixtures/packaging/babel-standalone/dev.html`. This file already uses `react.development.js` from the `build` folder so it will pick up your changes.

If you want to try your changes in your existing React project, you may copy `build/node_modules/react/umd/react.development.js`, `build/node_modules/react-dom/umd/react-dom.development.js`, or any other build products into your app and use them instead of the stable version.

If your project uses React from npm, you may delete `react` and `react-dom` in its dependencies and use `yarn link` to point them to your local `build` folder. Note that **instead of `--type=UMD` you'll want to pass `--type=NODE` when building**. You'll also need to build the `scheduler` package:

```
cd ~/path_to_your_react_clone/
yarn build react/index,react/jsx,react-dom/index,scheduler --type=NODE

cd build/node_modules/react
yarn link
cd build/node_modules/react-dom
yarn link

cd ~/path/to/your/project
yarn link react react-dom
```

Every time you run `yarn build` in the React folder, the updated versions will appear in your project's `node_modules`. You can then rebuild your project to try your changes.

If some package is still missing (e.g. maybe you use `react-dom/server` in your project), you can always do a full build with `yarn build`. Note that running `yarn build` without options takes a long time.

We still require that your pull request contains unit tests for any new functionality. This way we can ensure that we don't break your code in the future.

Style Guide

We use an automatic code formatter called [Prettier](#). Run `yarn prettier` after making any changes to the code.

Then, our linter will catch most issues that may exist in your code. You can check the status of your code styling by simply running `yarn linc`.

However, there are still some styles that the linter cannot pick up. If you are unsure about something, looking at [Airbnb's Style Guide](#) will guide you in the right direction.

Request for Comments (RFC)

Many changes, including bug fixes and documentation improvements can be implemented and reviewed via the normal GitHub pull request workflow.

Some changes though are "substantial", and we ask that these be put through a bit of a design process and produce a consensus among the React core team.

The "RFC" (request for comments) process is intended to provide a consistent and controlled path for new features to enter the project. You can contribute by visiting the [rfcs repository](#).

License

By contributing to React, you agree that your contributions will be licensed under its MIT license.

What Next?

Read the [next section](#) to learn how the codebase is organized.

Implementation Notes

This section is a collection of implementation notes for the [stack reconciler](#).

It is very technical and assumes a strong understanding of React public API as well as how it's divided into core, renderers, and the reconciler. If you're not very familiar with the React codebase, read [the codebase overview](#) first.

It also assumes an understanding of the [differences between React components, their instances, and elements](#).

The stack reconciler was used in React 15 and earlier. It is located at [src/renderers/shared/stack/reconciler](#).

Video: Building React from Scratch

Paul O'Shannessy gave a talk about [building React from scratch](#) that largely inspired this document.

Both this document and his talk are simplifications of the real codebase so you might get a better understanding by getting familiar with both of them.

Overview

The reconciler itself doesn't have a public API. [Renderers](#) like React DOM and React Native use it to efficiently update the user interface according to the React components written by the user.

Mounting as a Recursive Process

Let's consider the first time you mount a component:

```
const root = ReactDOM.createRoot(rootEl);
root.render(<App />);
```

`root.render` will pass `<App />` along to the reconciler. Remember that `<App />` is a React element, that is, a description of *what* to render. You can think about it as a plain object:

```
console.log(<App />);
// { type: App, props: {} }
```

The reconciler will check if `App` is a class or a function.

If `App` is a function, the reconciler will call `App(props)` to get the rendered element.

If `App` is a class, the reconciler will instantiate an `App` with `new App(props)`, call the `componentWillMount()` lifecycle method, and then will call the `render()` method to get the rendered element.

Either way, the reconciler will learn the element `App` "rendered to".

This process is recursive. `App` may render to a `<Greeting />`, `Greeting` may render to a `<Button />`, and so on. The reconciler will "drill down" through user-defined components recursively as it learns what each component renders to.

You can imagine this process as a pseudocode:

```

function isClass(type) {
  // React.Component subclasses have this flag
  return (
    Boolean(type.prototype) &&
    Boolean(type.prototype.isReactComponent)
  );
}

// This function takes a React element (e.g. <App />)
// and returns a DOM or Native node representing the mounted tree.
function mount(element) {
  var type = element.type;
  var props = element.props;

  // We will determine the rendered element
  // by either running the type as function
  // or creating an instance and calling render().
  var renderedElement;
  if (isClass(type)) {
    // Component class
    var publicInstance = new type(props);
    // Set the props
    publicInstance.props = props;
    // Call the lifecycle if necessary
    if (publicInstance.componentWillMount) {
      publicInstance.componentWillMount();
    }
    // Get the rendered element by calling render()
    renderedElement = publicInstance.render();
  } else {
    // Component function
    renderedElement = type(props);
  }

  // This process is recursive because a component may
  // return an element with a type of another component.
  return mount(renderedElement);

  // Note: this implementation is incomplete and recurses infinitely!
  // It only handles elements like <App /> or <Button />.
  // It doesn't handle elements like <div /> or <p /> yet.
}

var rootEl = document.getElementById('root');
var node = mount(<App />);
rootEl.appendChild(node);

```

Note:

This really *is* a pseudo-code. It isn't similar to the real implementation. It will also cause a stack overflow because we haven't discussed when to stop the recursion.

Let's recap a few key ideas in the example above:

- React elements are plain objects representing the component type (e.g. `App`) and the props.
- User-defined components (e.g. `App`) can be classes or functions but they all "render to" elements.
- "Mounting" is a recursive process that creates a DOM or Native tree given the top-level React element (e.g. `<App />`).

Mounting Host Elements

This process would be useless if we didn't render something to the screen as a result.

In addition to user-defined ("composite") components, React elements may also represent platform-specific ("host") components. For example, `Button` might return a `<div />` from its render method.

If element's `type` property is a string, we are dealing with a host element:

```
console.log(<div />);
// { type: 'div', props: {} }
```

There is no user-defined code associated with host elements.

When the reconciler encounters a host element, it lets the renderer take care of mounting it. For example, React DOM would create a DOM node.

If the host element has children, the reconciler recursively mounts them following the same algorithm as above. It doesn't matter whether children are host (like `<div><hr /></div>`), composite (like `<div><Button /></div>`), or both.

The DOM nodes produced by the child components will be appended to the parent DOM node, and recursively, the complete DOM structure will be assembled.

Note:

The reconciler itself is not tied to the DOM. The exact result of mounting (sometimes called "mount image" in the source code) depends on the renderer, and can be a DOM node (React DOM), a string (React DOM Server), or a number representing a native view (React Native).

If we were to extend the code to handle host elements, it would look like this:

```
function isClass(type) {
  // React.Component subclasses have this flag
  return (
    Boolean(type.prototype) &&
    Boolean(type.prototype.isReactComponent)
  );
}

// This function only handles elements with a composite type.
// For example, it handles <App /> and <Button />, but not a <div />.
function mountComposite(element) {
```

```

var type = element.type;
var props = element.props;

var renderedElement;
if (isClass(type)) {
  // Component class
  var publicInstance = new type(props);
  // Set the props
  publicInstance.props = props;
  // Call the lifecycle if necessary
  if (publicInstance.componentWillMount) {
    publicInstance.componentWillMount();
  }
  renderedElement = publicInstance.render();
} else if (typeof type === 'function') {
  // Component function
  renderedElement = type(props);
}

// This is recursive but we'll eventually reach the bottom of recursion when
// the element is host (e.g. <div />) rather than composite (e.g. <App />):
return mount(renderedElement);
}

// This function only handles elements with a host type.
// For example, it handles <div /> and <p /> but not an <App />.
function mountHost(element) {
  var type = element.type;
  var props = element.props;
  var children = props.children || [];
  if (!Array.isArray(children)) {
    children = [children];
  }
  children = children.filter(Boolean);

  // This block of code shouldn't be in the reconciler.
  // Different renderers might initialize nodes differently.
  // For example, React Native would create iOS or Android views.
  var node = document.createElement(type);
  Object.keys(props).forEach(propName => {
    if (propName !== 'children') {
      node.setAttribute(propName, props[propName]);
    }
  });

  // Mount the children
  children.forEach(childElement => {
    // Children may be host (e.g. <div />) or composite (e.g. <Button />).
    // We will also mount them recursively:
    var childNode = mount(childElement);

    // This line of code is also renderer-specific.
    // It would be different depending on the renderer:
    node.appendChild(childNode);
  });

  // Return the DOM node as mount result.
  // This is where the recursion ends.
  return node;
}

function mount(element) {

```

```

var type = element.type;
if (typeof type === 'function') {
  // User-defined components
  return mountComposite(element);
} else if (typeof type === 'string') {
  // Platform-specific components
  return mountHost(element);
}

var rootEl = document.getElementById('root');
var node = mount(<App />);
rootEl.appendChild(node);

```

This is working but still far from how the reconciler is really implemented. The key missing ingredient is support for updates.

Introducing Internal Instances

The key feature of React is that you can re-render everything, and it won't recreate the DOM or reset the state:

```

root.render(<App />);
// Should reuse the existing DOM:
root.render(<App />);

```

However, our implementation above only knows how to mount the initial tree. It can't perform updates on it because it doesn't store all the necessary information, such as all the `publicInstances`, or which DOM `node`s correspond to which components.

The stack reconciler codebase solves this by making the `mount()` function a method and putting it on a class. There are drawbacks to this approach, and we are going in the opposite direction in the [ongoing rewrite of the reconciler](#). Nevertheless this is how it works now.

Instead of separate `mountHost` and `mountComposite` functions, we will create two classes: `DOMComponent` and `CompositeComponent`.

Both classes have a constructor accepting the `element`, as well as a `mount()` method returning the mounted node. We will replace a top-level `mount()` function with a factory that instantiates the correct class:

```

function instantiateComponent(element) {
  var type = element.type;
  if (typeof type === 'function') {
    // User-defined components
    return new CompositeComponent(element);
  } else if (typeof type === 'string') {
    // Platform-specific components
    return new DOMComponent(element);
  }
}

```

First, let's consider the implementation of `CompositeComponent`:

```

class CompositeComponent {
  constructor(element) {
    this.currentElement = element;
    this.renderedComponent = null;
    this.publicInstance = null;
  }

  getPublicInstance() {
    // For composite components, expose the class instance.
    return this.publicInstance;
  }

  mount() {
    var element = this.currentElement;
    var type = element.type;
    var props = element.props;

    var publicInstance;
    var renderedElement;
    if (isClass(type)) {
      // Component class
      publicInstance = new type(props);
      // Set the props
      publicInstance.props = props;
      // Call the lifecycle if necessary
      if (publicInstance.componentWillMount) {
        publicInstance.componentWillMount();
      }
      renderedElement = publicInstance.render();
    } else if (typeof type === 'function') {
      // Component function
      publicInstance = null;
      renderedElement = type(props);
    }

    // Save the public instance
    this.publicInstance = publicInstance;

    // Instantiate the child internal instance according to the element.
    // It would be a DOMComponent for <div /> or <p />,
    // and a CompositeComponent for <App /> or <Button />:
    var renderedComponent = instantiateComponent(renderedElement);
    this.renderedComponent = renderedComponent;

    // Mount the rendered output
    return renderedComponent.mount();
  }
}

```

This is not much different from our previous `mountComposite()` implementation, but now we can save some information, such as `this.currentElement`, `this.renderedComponent`, and `this.publicInstance`, for use during updates.

Note that an instance of `CompositeComponent` is not the same thing as an instance of the user-supplied `element.type`. `CompositeComponent` is an implementation detail of our reconciler, and is never exposed to the user. The user-defined class is the one we read from `element.type`, and `CompositeComponent` creates an instance of it.

To avoid the confusion, we will call instances of `CompositeComponent` and `DOMComponent` "internal instances". They exist so we can associate some long-lived data with them. Only the renderer and the reconciler are aware that they exist.

In contrast, we call an instance of the user-defined class a "public instance". The public instance is what you see as `this` in the `render()` and other methods of your custom components.

The `mountHost()` function, refactored to be a `mount()` method on `DOMComponent` class, also looks familiar:

```
class DOMComponent {
  constructor(element) {
    this.currentElement = element;
    this.renderedChildren = [];
    this.node = null;
  }

  getPublicInstance() {
    // For DOM components, only expose the DOM node.
    return this.node;
  }

  mount() {
    var element = this.currentElement;
    var type = element.type;
    var props = element.props;
    var children = props.children || [];
    if (!Array.isArray(children)) {
      children = [children];
    }

    // Create and save the node
    var node = document.createElement(type);
    this.node = node;

    // Set the attributes
    Object.keys(props).forEach(propName => {
      if (propName !== 'children') {
        node.setAttribute(propName, props[propName]);
      }
    });

    // Create and save the contained children.
    // Each of them can be a DOMComponent or a CompositeComponent,
    // depending on whether the element type is a string or a function.
    var renderedChildren = children.map(instantiateComponent);
    this.renderedChildren = renderedChildren;

    // Collect DOM nodes they return on mount
    var childNodes = renderedChildren.map(child => child.mount());
    childNodes.forEach(childNode => node.appendChild(childNode));

    // Return the DOM node as mount result
    return node;
  }
}
```

The main difference after refactoring from `mountHost()` is that we now keep `this.node` and `this.renderedChildren` associated with the internal DOM component instance. We will also use them for applying non-destructive updates in the future.

As a result, each internal instance, composite or host, now points to its child internal instances. To help visualize this, if a function `<App>` component renders a `<Button>` class component, and `Button` class renders a `<div>`, the internal instance tree would look like this:

```
[object CompositeComponent] {
  currentElement: <App />,
  publicInstance: null,
  renderedComponent: [object CompositeComponent] {
    currentElement: <Button />,
    publicInstance: [object Button],
    renderedComponent: [object DOMComponent] {
      currentElement: <div />,
      node: [object HTMLDivElement],
      renderedChildren: []
    }
  }
}
```

In the DOM you would only see the `<div>`. However the internal instance tree contains both composite and host internal instances.

The composite internal instances need to store:

- The current element.
- The public instance if element type is a class.
- The single rendered internal instance. It can be either a `DOMComponent` or a `CompositeComponent`.

The host internal instances need to store:

- The current element.
- The DOM node.
- All the child internal instances. Each of them can be either a `DOMComponent` or a `CompositeComponent`.

If you're struggling to imagine how an internal instance tree is structured in more complex applications, [React DevTools](#) can give you a close approximation, as it highlights host instances with grey, and composite instances with purple:

```

<TodoApp>
  ▼<div>
    <h3>TODO</h3>
    ▼<TodoList items=[{...}]>
      ▼<ul>
        <li key="1475260226151">Buy milk</li>
      </ul>
    </TodoList>
    ▼<form onSubmit=bound handleSubmit()>
      <input onChange=bound onChange() value="" />
      <button>Add #2</button>
    </form>
  </div>
</TodoApp>

```

To complete this refactoring, we will introduce a function that mounts a complete tree into a container node and a public instance:

```

function mountTree(element, containerNode) {
  // Create the top-level internal instance
  var rootComponent = instantiateComponent(element);

  // Mount the top-level component into the container
  var node = rootComponent.mount();
  containerNode.appendChild(node);

  // Return the public instance it provides
  var publicInstance = rootComponent.getPublicInstance();
  return publicInstance;
}

var rootEl = document.getElementById('root');
mountTree(<App />, rootEl);

```

Unmounting

Now that we have internal instances that hold onto their children and the DOM nodes, we can implement unmounting. For a composite component, unmounting calls a lifecycle method and recurses.

```

class CompositeComponent {
  // ...

  unmount() {
    // Call the lifecycle method if necessary
    var publicInstance = this.publicInstance;
    if (publicInstance) {
      if (publicInstance.componentWillUnmount) {
        publicInstance.componentWillUnmount();
      }
    }
  }
}

```

```
// Unmount the single rendered component
var renderedComponent = this.renderedComponent;
renderedComponent.unmount();
}
```

For `DOMComponent`, unmounting tells each child to unmount:

```
class DOMComponent {
  // ...
  unmount() {
    // Unmount all the children
    var renderedChildren = this.renderedChildren;
    renderedChildren.forEach(child => child.unmount());
  }
}
```

In practice, unmounting DOM components also removes the event listeners and clears some caches, but we will skip those details.

We can now add a new top-level function called `unmountTree(containerNode)` that is similar to `ReactDOM.unmountComponentAtNode()`:

```
function unmountTree(containerNode) {
  // Read the internal instance from a DOM node:
  // (This doesn't work yet, we will need to change mountTree() to store it.)
  var node = containerNode.firstChild;
  var rootComponent = node._internalInstance;

  // Unmount the tree and clear the container
  rootComponent.unmount();
  containerNode.innerHTML = '';
}
```

In order for this to work, we need to read an internal root instance from a DOM node. We will modify `mountTree()` to add the `_internalInstance` property to the root DOM node. We will also teach `mountTree()` to destroy any existing tree so it can be called multiple times:

```
function mountTree(element, containerNode) {
  // Destroy any existing tree
  if (containerNode.firstChild) {
    unmountTree(containerNode);
  }

  // Create the top-level internal instance
  var rootComponent = instantiateComponent(element);

  // Mount the top-level component into the container
  var node = rootComponent.mount();
  containerNode.appendChild(node);

  // Save a reference to the internal instance
  node._internalInstance = rootComponent;

  // Return the public instance it provides
}
```

```

var publicInstance = rootComponent.getPublicInstance();
return publicInstance;
}

```

Now, running `unmountTree()`, or running `mountTree()` repeatedly, removes the old tree and runs the `componentWillUnmount()` lifecycle method on components.

Updating

In the previous section, we implemented unmounting. However React wouldn't be very useful if each prop change unmounted and mounted the whole tree. The goal of the reconciler is to reuse existing instances where possible to preserve the DOM and the state:

```

var rootEl = document.getElementById('root');

mountTree(<App />, rootEl);
// Should reuse the existing DOM:
mountTree(<App />, rootEl);

```

We will extend our internal instance contract with one more method. In addition to `mount()` and `unmount()`, both `DOMComponent` and `CompositeComponent` will implement a new method called `receive(nextElement)`:

```

class CompositeComponent {
  // ...

  receive(nextElement) {
    // ...
  }
}

class DOMComponent {
  // ...

  receive(nextElement) {
    // ...
  }
}

```

Its job is to do whatever is necessary to bring the component (and any of its children) up to date with the description provided by the `nextElement`.

This is the part that is often described as "virtual DOM diffing" although what really happens is that we walk the internal tree recursively and let each internal instance receive an update.

Updating Composite Components

When a composite component receives a new element, we run the `componentWillUpdate()` lifecycle method.

Then we re-render the component with the new props, and get the next rendered element:

```

class CompositeComponent {
  // ...

  receive(nextElement) {
    var prevProps = this.currentElement.props;
    var publicInstance = this.publicInstance;
    var prevRenderedComponent = this.renderedComponent;
    var prevRenderedElement = prevRenderedComponent.currentElement;

    // Update *own* element
    this.currentElement = nextElement;
    var type = nextElement.type;
    var nextProps = nextElement.props;

    // Figure out what the next render() output is
    var nextRenderedElement;
    if (isClass(type)) {
      // Component class
      // Call the lifecycle if necessary
      if (publicInstance.componentWillUpdate) {
        publicInstance.componentWillUpdate(nextProps);
      }
      // Update the props
      publicInstance.props = nextProps;
      // Re-render
      nextRenderedElement = publicInstance.render();
    } else if (typeof type === 'function') {
      // Component function
      nextRenderedElement = type(nextProps);
    }
  }
  // ...
}

```

Next, we can look at the rendered element's `type`. If the `type` has not changed since the last render, the component below can also be updated in place.

For example, if it returned `<Button color="red" />` the first time, and `<Button color="blue" />` the second time, we can just tell the corresponding internal instance to `receive()` the next element:

```

// ...

// If the rendered element type has not changed,
// reuse the existing component instance and exit.
if (prevRenderedElement.type === nextRenderedElement.type) {
  prevRenderedComponent.receive(nextRenderedElement);
  return;
}

// ...

```

However, if the next rendered element has a different `type` than the previously rendered element, we can't update the internal instance. A `<button>` can't "become" an `<input>`.

Instead, we have to unmount the existing internal instance and mount the new one corresponding to the rendered element type. For example, this is what happens when a component that previously rendered a `<button />` renders an `<input />`:

```
// ...

// If we reached this point, we need to unmount the previously
// mounted component, mount the new one, and swap their nodes.

// Find the old node because it will need to be replaced
var prevNode = prevRenderedComponent.getHostNode();

// Unmount the old child and mount a new child
prevRenderedComponent.unmount();
var nextRenderedComponent = instantiateComponent(nextRenderedElement);
var nextNode = nextRenderedComponent.mount();

// Replace the reference to the child
this.renderedComponent = nextRenderedComponent;

// Replace the old node with the new one
// Note: this is renderer-specific code and
// ideally should live outside of CompositeComponent:
prevNode.parentNode.replaceChild(nextNode, prevNode);

}

}
```

To sum this up, when a composite component receives a new element, it may either delegate the update to its rendered internal instance, or unmount it and mount a new one in its place.

There is another condition under which a component will re-mount rather than receive an element, and that is when the element's `key` has changed. We don't discuss `key` handling in this document because it adds more complexity to an already complex tutorial.

Note that we needed to add a method called `getHostNode()` to the internal instance contract so that it's possible to locate the platform-specific node and replace it during the update. Its implementation is straightforward for both classes:

```
class CompositeComponent {
// ...

getHostNode() {
  // Ask the rendered component to provide it.
  // This will recursively drill down any composites.
  return this.renderedComponent.getHostNode();
}

class DOMComponent {
// ...

getHostNode() {
  return this.node;
}
}
```

Updating Host Components

Host component implementations, such as `DOMComponent`, update differently. When they receive an element, they need to update the underlying platform-specific view. In case of React DOM, this means updating the DOM attributes:

```

class DOMComponent {
// ...

receive(nextElement) {
  var node = this.node;
  var prevElement = this.currentElement;
  var prevProps = prevElement.props;
  var nextProps = nextElement.props;
  this.currentElement = nextElement;

  // Remove old attributes.
  Object.keys(prevProps).forEach(propName => {
    if (propName !== 'children' && !nextProps.hasOwnProperty(propName)) {
      node.removeAttribute(propName);
    }
  });
  // Set next attributes.
  Object.keys(nextProps).forEach(propName => {
    if (propName !== 'children') {
      node.setAttribute(propName, nextProps[propName]);
    }
  });
}

// ...

```

Then, host components need to update their children. Unlike composite components, they might contain more than a single child.

In this simplified example, we use an array of internal instances and iterate over it, either updating or replacing the internal instances depending on whether the received `type` matches their previous `type`. The real reconciler also takes element's `key` in the account and track moves in addition to insertions and deletions, but we will omit this logic.

We collect DOM operations on children in a list so we can execute them in batch:

```

// ...

// These are arrays of React elements:
var prevChildren = prevProps.children || [];
if (!Array.isArray(prevChildren)) {
  prevChildren = [prevChildren];
}
var nextChildren = nextProps.children || [];
if (!Array.isArray(nextChildren)) {
  nextChildren = [nextChildren];
}
// These are arrays of internal instances:
var prevRenderedChildren = this.renderedChildren;
var nextRenderedChildren = [];

// As we iterate over children, we will add operations to the array.
var operationQueue = [];

// Note: the section below is extremely simplified!
// It doesn't handle reorders, children with holes, or keys.
// It only exists to illustrate the overall flow, not the specifics.

for (var i = 0; i < nextChildren.length; i++) {
  // Try to get an existing internal instance for this child

```

```

var prevChild = prevRenderedChildren[i];

// If there is no internal instance under this index,
// a child has been appended to the end. Create a new
// internal instance, mount it, and use its node.
if (!prevChild) {
  var nextChild = instantiateComponent(nextChildren[i]);
  var node = nextChild.mount();

  // Record that we need to append a node
  operationQueue.push({type: 'ADD', node});
  nextRenderedChildren.push(nextChild);
  continue;
}

// We can only update the instance if its element's type matches.
// For example, <Button size="small" /> can be updated to
// <Button size="large" /> but not to an <App />.
var canUpdate = prevChildren[i].type === nextChildren[i].type;

// If we can't update an existing instance, we have to unmount it
// and mount a new one instead of it.
if (!canUpdate) {
  var prevNode = prevChild.getHostNode();
  prevChild.unmount();

  var nextChild = instantiateComponent(nextChildren[i]);
  var nextNode = nextChild.mount();

  // Record that we need to swap the nodes
  operationQueue.push({type: 'REPLACE', prevNode, nextNode});
  nextRenderedChildren.push(nextChild);
  continue;
}

// If we can update an existing internal instance,
// just let it receive the next element and handle its own update.
prevChild.receive(nextChildren[i]);
nextRenderedChildren.push(prevChild);
}

// Finally, unmount any children that don't exist:
for (var j = nextChildren.length; j < prevChildren.length; j++) {
  var prevChild = prevRenderedChildren[j];
  var node = prevChild.getHostNode();
  prevChild.unmount();

  // Record that we need to remove the node
  operationQueue.push({type: 'REMOVE', node});
}

// Point the list of rendered children to the updated version.
this.renderedChildren = nextRenderedChildren;

// ...

```

As the last step, we execute the DOM operations. Again, the real reconciler code is more complex because it also handles moves:

```
// ...

// Process the operation queue.
while (operationQueue.length > 0) {
  var operation = operationQueue.shift();
  switch (operation.type) {
    case 'ADD':
      this.node.appendChild(operation.node);
      break;
    case 'REPLACE':
      this.node.replaceChild(operation.nextNode, operation.prevNode);
      break;
    case 'REMOVE':
      this.node.removeChild(operation.node);
      break;
  }
}
}
```

And that is it for updating host components.

Top-Level Updates

Now that both `CompositeComponent` and `DOMComponent` implement the `receive(nextElement)` method, we can change the top-level `mountTree()` function to use it when the element `type` is the same as it was the last time:

```
function mountTree(element, containerNode) {
  // Check for an existing tree
  if (containerNode.firstChild) {
    var prevNode = containerNode.firstChild;
    var prevRootComponent = prevNode._internalInstance;
    var prevElement = prevRootComponent.currentElement;

    // If we can, reuse the existing root component
    if (prevElement.type === element.type) {
      prevRootComponent.receive(element);
      return;
    }

    // Otherwise, unmount the existing tree
    unmountTree(containerNode);
  }

  // ...
}
```

Now calling `mountTree()` two times with the same type isn't destructive:

```
var rootEl = document.getElementById('root');

mountTree(<App />, rootEl);
// Reuses the existing DOM:
mountTree(<App />, rootEl);
```

These are the basics of how React works internally.

What We Left Out

This document is simplified compared to the real codebase. There are a few important aspects we didn't address:

- Components can render `null`, and the reconciler can handle "empty slots" in arrays and rendered output.
- The reconciler also reads `key` from the elements, and uses it to establish which internal instance corresponds to which element in an array. A bulk of complexity in the actual React implementation is related to that.
- In addition to composite and host internal instance classes, there are also classes for "text" and "empty" components. They represent text nodes and the "empty slots" you get by rendering `null`.
- Renderers use [injection](#) to pass the host internal class to the reconciler. For example, React DOM tells the reconciler to use `ReactDOMComponent` as the host internal instance implementation.
- The logic for updating the list of children is extracted into a mixin called `ReactMultiChild` which is used by the host internal instance class implementations both in React DOM and React Native.
- The reconciler also implements support for `setState()` in composite components. Multiple updates inside event handlers get batched into a single update.
- The reconciler also takes care of attaching and detaching refs to composite components and host nodes.
- Lifecycle methods that are called after the DOM is ready, such as `componentDidMount()` and `componentDidUpdate()`, get collected into "callback queues" and are executed in a single batch.
- React puts information about the current update into an internal object called "transaction". Transactions are useful for keeping track of the queue of pending lifecycle methods, the current DOM nesting for the warnings, and anything else that is "global" to a specific update. Transactions also ensure React "cleans everything up" after updates. For example, the transaction class provided by React DOM restores the input selection after any update.

Jumping into the Code

- `ReactMount` is where the code like `mountTree()` and `unmountTree()` from this tutorial lives. It takes care of mounting and unmounting top-level components. `ReactNativeMount` is its React Native analog.
- `ReactDOMComponent` is the equivalent of `DOMComponent` in this tutorial. It implements the host component class for React DOM renderer. `ReactNativeBaseComponent` is its React Native analog.
- `ReactCompositeComponent` is the equivalent of `CompositeComponent` in this tutorial. It handles calling user-defined components and maintaining their state.
- `instantiateReactComponent` contains the switch that picks the right internal instance class to construct for an element. It is equivalent to `instantiateComponent()` in this tutorial.

- `ReactReconciler` is a wrapper with `mountComponent()`, `receiveComponent()`, and `unmountComponent()` methods. It calls the underlying implementations on the internal instances, but also includes some code around them that is shared by all internal instance implementations.
- `ReactChildReconciler` implements the logic for mounting, updating, and unmounting children according to the `key` of their elements.
- `ReactMultiChild` implements processing the operation queue for child insertions, deletions, and moves independently of the renderer.
- `mount()`, `receive()`, and `unmount()` are really called `mountComponent()`, `receiveComponent()`, and `unmountComponent()` in React codebase for legacy reasons, but they receive elements.
- Properties on the internal instances start with an underscore, e.g. `_currentElement`. They are considered to be read-only public fields throughout the codebase.

Future Directions

Stack reconciler has inherent limitations such as being synchronous and unable to interrupt the work or split it in chunks. There is a work in progress on the [new Fiber reconciler](#) with a [completely different architecture](#). In the future, we intend to replace stack reconciler with it, but at the moment it is far from feature parity.

Next Steps

Read the [next section](#) to learn about the guiding principles we use for React development.

Integrating with Other Libraries

React can be used in any web application. It can be embedded in other applications and, with a little care, other applications can be embedded in React. This guide will examine some of the more common use cases, focusing on integration with [jQuery](#) and [Backbone](#), but the same ideas can be applied to integrating components with any existing code.

Integrating with DOM Manipulation Plugins

React is unaware of changes made to the DOM outside of React. It determines updates based on its own internal representation, and if the same DOM nodes are manipulated by another library, React gets confused and has no way to recover.

This does not mean it is impossible or even necessarily difficult to combine React with other ways of affecting the DOM, you just have to be mindful of what each is doing.

The easiest way to avoid conflicts is to prevent the React component from updating. You can do this by rendering elements that React has no reason to update, like an empty `<div />`.

How to Approach the Problem

To demonstrate this, let's sketch out a wrapper for a generic jQuery plugin.

We will attach a `ref` to the root DOM element. Inside `componentDidMount`, we will get a reference to it so we can pass it to the jQuery plugin.

To prevent React from touching the DOM after mounting, we will return an empty `<div />` from the `render()` method. The `<div />` element has no properties or children, so React has no reason to update it, leaving the jQuery plugin free to manage that part of the DOM:

```
class SomePlugin extends React.Component {
  componentDidMount() {
    this.$el = $(this.el);
    this.$el.somePlugin();
  }

  componentWillUnmount() {
    this.$el.somePlugin('destroy');
  }

  render() {
    return <div ref={el => this.el = el} />;
  }
}
```

Note that we defined both `componentDidMount` and `componentWillUnmount` lifecycle methods. Many jQuery plugins attach event listeners to the DOM so it's important to detach them in `componentWillUnmount`. If the plugin does not provide a method for cleanup, you will probably have to provide your own, remembering to remove any event listeners the plugin registered to prevent memory leaks.

Integrating with jQuery Chosen Plugin

For a more concrete example of these concepts, let's write a minimal wrapper for the plugin [Chosen](#), which augments `<select>` inputs.

Note:

Just because it's possible, doesn't mean that it's the best approach for React apps. We encourage you to use React components when you can. React components are easier to reuse in React applications, and often provide more control over their behavior and appearance.

First, let's look at what Chosen does to the DOM.

If you call it on a `<select>` DOM node, it reads the attributes off of the original DOM node, hides it with an inline style, and then appends a separate DOM node with its own visual representation right after the `<select>`. Then it fires jQuery events to notify us about the changes.

Let's say that this is the API we're striving for with our `<Chosen>` wrapper React component:

```
function Example() {
  return (
    <Chosen onChange={value => console.log(value)}>
      <option>vanilla</option>
      <option>chocolate</option>
      <option>strawberry</option>
    </Chosen>
  );
}
```

We will implement it as an [uncontrolled component](#) for simplicity.

First, we will create an empty component with a `render()` method where we return `<select>` wrapped in a `<div>`:

```
class Chosen extends React.Component {
  render() {
    return (
      <div>
        <select className="Chosen-select" ref={el => this.el = el}>
          {this.props.children}
        </select>
      </div>
    );
  }
}
```

Notice how we wrapped `<select>` in an extra `<div>`. This is necessary because Chosen will append another DOM element right after the `<select>` node we passed to it. However, as far as React is concerned, `<div>` always only has a single child. This is how we ensure that React updates won't conflict with the extra DOM node appended by Chosen. It is important that if you modify the DOM outside of React flow, you must ensure React doesn't have a reason to touch those DOM nodes.

Next, we will implement the lifecycle methods. We need to initialize Chosen with the ref to the `<select>` node in `componentDidMount`, and tear it down in `componentWillUnmount`:

```
componentDidMount() {
  this.$el = $(this.el);
  this.$el.chosen();
}

componentWillUnmount() {
  this.$el.chosen('destroy');
}
```

[Try it on CodePen](#)

Note that React assigns no special meaning to the `this.el` field. It only works because we have previously assigned this field from a `ref` in the `render()` method:

```
<select className="Chosen-select" ref={el => this.el = el}>
```

This is enough to get our component to render, but we also want to be notified about the value changes. To do this, we will subscribe to the jQuery `change` event on the `<select>` managed by Chosen.

We won't pass `this.props.onChange` directly to Chosen because component's props might change over time, and that includes event handlers. Instead, we will declare a `handleChange()` method that calls `this.props.onChange`, and subscribe it to the jQuery `change` event:

```
componentDidMount() {
  this.$el = $(this.el);
  this.$el.chosen();

  this.handleChange = this.handleChange.bind(this);
  this.$el.on('change', this.handleChange);
}

componentWillUnmount() {
  this.$el.off('change', this.handleChange);
  this.$el.chosen('destroy');
}

handleChange(e) {
  this.props.onChange(e.target.value);
}
```

[Try it on CodePen](#)

Finally, there is one more thing left to do. In React, props can change over time. For example, the `<Chosen>` component can get different children if parent component's state changes. This means that at integration points it is important that we manually update the DOM in response to prop updates, since we no longer let React manage the DOM for us.

Chosen's documentation suggests that we can use jQuery `trigger()` API to notify it about changes to the original DOM element. We will let React take care of updating `this.props.children` inside `<select>`, but we will also add a `componentDidUpdate()` lifecycle method that notifies Chosen about changes in the children list:

```
componentDidUpdate(prevProps) {
  if (prevProps.children !== this.props.children) {
    this.$el.trigger("chosen:updated");
  }
}
```

This way, Chosen will know to update its DOM element when the `<select>` children managed by React change.

The complete implementation of the `Chosen` component looks like this:

```
class Chosen extends React.Component {
  componentDidMount() {
    this.$el = $(this.el);
    this.$el.chosen();

    this.handleChange = this.handleChange.bind(this);
    this.$el.on('change', this.handleChange);
  }

  componentDidUpdate(prevProps) {
    if (prevProps.children !== this.props.children) {
      this.$el.trigger("chosen:updated");
    }
  }

  componentWillUnmount() {
    this.$el.off('change', this.handleChange);
    this.$el.chosen('destroy');
  }

  handleChange(e) {
    this.props.onChange(e.target.value);
  }

  render() {
    return (
      <div>
        <select className="Chosen-select" ref={el => this.el = el}>
          {this.props.children}
        </select>
      </div>
    );
  }
}
```

[Try it on CodePen](#)

Integrating with Other View Libraries

React can be embedded into other applications thanks to the flexibility of `createRoot()`.

Although React is commonly used at startup to load a single root React component into the DOM, `createRoot()` can also be called multiple times for independent parts of the UI which can be as small as a button, or as large as an app.

In fact, this is exactly how React is used at Facebook. This lets us write applications in React piece by piece, and combine them with our existing server-generated templates and other client-side code.

Replacing String-Based Rendering with React

A common pattern in older web applications is to describe chunks of the DOM as a string and insert it into the DOM like so: `$el.html(htmlString)`. These points in a codebase are perfect for introducing React. Just rewrite the string based rendering as a React component.

So the following jQuery implementation...

```
$('#container').html('<button id="btn">Say Hello</button>');
$('#btn').click(function() {
  alert('Hello!');
});
```

...could be rewritten using a React component:

```
function Button() {
  return <button id="btn">Say Hello</button>;
}

$('#btn').click(function() {
  alert('Hello!');
});
```

From here you could start moving more logic into the component and begin adopting more common React practices. For example, in components it is best not to rely on IDs because the same component can be rendered multiple times. Instead, we will use the [React event system](#) and register the click handler directly on the React `<button>` element:

```
function Button(props) {
  return <button onClick={props.onClick}>Say Hello</button>;
}

function HelloButton() {
  function handleClick() {
    alert('Hello!');
  }
  return <Button onClick={handleClick} />;
}
```

[Try it on CodePen](#)

You can have as many such isolated components as you like, and use `ReactDOM.createRoot()` to render them to different DOM containers. Gradually, as you convert more of your app to React, you will be able to combine them into larger components, and move some of the `ReactDOM.createRoot()` calls up the hierarchy.

Embedding React in a Backbone View

[Backbone](#) views typically use HTML strings, or string-producing template functions, to create the content for their DOM elements. This process, too, can be replaced with rendering a React component.

Below, we will create a Backbone view called `ParagraphView`. It will override Backbone's `render()` function to render a React `<Paragraph>` component into the DOM element provided by Backbone (`this.el`). Here, too, we are using `ReactDOM.createRoot()`:

```
function Paragraph(props) {
  return <p>{props.text}</p>;
}

const ParagraphView = Backbone.View.extend({
  initialize(options) {
    this.reactRoot = ReactDOM.createRoot(this.el);
  },
  render() {
    const text = this.model.get('text');
    this.reactRoot.render(<Paragraph text={text} />);
    return this;
  },
  remove() {
    this.reactRoot.unmount();
    Backbone.View.prototype.remove.call(this);
  }
});
```

Try it on [CodePen](#)

It is important that we also call `root.unmount()` in the `remove` method so that React unregisters event handlers and other resources associated with the component tree when it is detached.

When a component is removed *from within* a React tree, the cleanup is performed automatically, but because we are removing the entire tree by hand, we must call this method.

Integrating with Model Layers

While it is generally recommended to use unidirectional data flow such as [React state](#), [Flux](#), or [Redux](#), React components can use a model layer from other frameworks and libraries.

Using Backbone Models in React Components

The simplest way to consume [Backbone](#) models and collections from a React component is to listen to the various change events and manually force an update.

Components responsible for rendering models would listen to `'change'` events, while components responsible for rendering collections would listen for `'add'` and `'remove'` events. In both cases, call `this.forceUpdate()` to rerender the component with the new data.

In the example below, the `List` component renders a Backbone collection, using the `Item` component to render individual items.

```
class Item extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
  }

  handleChange() {
    this.forceUpdate();
  }

  componentDidMount() {
    this.props.model.on('change', this.handleChange);
  }

  componentWillUnmount() {
    this.props.model.off('change', this.handleChange);
  }

  render() {
    return <li>{this.props.model.get('text')}</li>;
  }
}

class List extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
  }

  handleChange() {
    this.forceUpdate();
  }

  componentDidMount() {
    this.props.collection.on('add', 'remove', this.handleChange);
  }

  componentWillUnmount() {
    this.props.collection.off('add', 'remove', this.handleChange);
  }

  render() {
    return (
      <ul>
        {this.props.collection.map(model => (
          <Item key={model.cid} model={model} />
        ))}
      </ul>
    );
  }
}
```

[Try it on CodePen](#)

Extracting Data from Backbone Models

The approach above requires your React components to be aware of the Backbone models and collections. If you later plan to migrate to another data management solution, you might want to concentrate the knowledge about Backbone in as few parts of the code as possible.

One solution to this is to extract the model's attributes as plain data whenever it changes, and keep this logic in a single place. The following is a [higher-order component](#) that extracts all attributes of a Backbone model into state, passing the data to the wrapped component.

This way, only the higher-order component needs to know about Backbone model internals, and most components in the app can stay agnostic of Backbone.

In the example below, we will make a copy of the model's attributes to form the initial state. We subscribe to the `change` event (and unsubscribe on unmounting), and when it happens, we update the state with the model's current attributes. Finally, we make sure that if the `model` prop itself changes, we don't forget to unsubscribe from the old model, and subscribe to the new one.

Note that this example is not meant to be exhaustive with regards to working with Backbone, but it should give you an idea for how to approach this in a generic way:

```
function connectToBackboneModel(WrappedComponent) {
  return class BackboneComponent extends React.Component {
    constructor(props) {
      super(props);
      this.state = Object.assign({}, props.model.attributes);
      this.handleChange = this.handleChange.bind(this);
    }

    componentDidMount() {
      this.props.model.on('change', this.handleChange);
    }

    componentWillReceiveProps(nextProps) {
      this.setState(Object.assign({}, nextProps.model.attributes));
      if (nextProps.model !== this.props.model) {
        this.props.model.off('change', this.handleChange);
        nextProps.model.on('change', this.handleChange);
      }
    }

    componentWillUnmount() {
      this.props.model.off('change', this.handleChange);
    }

    handleChange(model) {
      this.setState(model.changedAttributes());
    }

    render() {
      const propsExceptModel = Object.assign({}, this.props);
      delete propsExceptModel.model;
      return <WrappedComponent {...propsExceptModel} {...this.state} />;
    }
  };
}
```

```

        }
    }
}
```

To demonstrate how to use it, we will connect a `NameInput` React component to a Backbone model, and update its `firstName` attribute every time the input changes:

```

function NameInput(props) {
  return (
    <p>
      <input value={props.firstName} onChange={props.handleChange} />
      <br />
      My name is {props.firstName}.
    </p>
  );
}

const BackboneNameInput = connectToBackboneModel(NameInput);

function Example(props) {
  function handleChange(e) {
    props.model.set('firstName', e.target.value);
  }

  return (
    <BackboneNameInput
      model={props.model}
      handleChange={handleChange}
    />
  );
}

const model = new Backbone.Model({ firstName: 'Frodo' });
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Example model={model} />);
```

[Try it on CodePen](#)

This technique is not limited to Backbone. You can use React with any model library by subscribing to its changes in the lifecycle methods and, optionally, copying the data into the local React state.

[Go to TOC](#)

JSX היכרות עם

הביטו על הצהרת משתנה זו:

```
const element = <h1>!שלום, עולם</h1>;
```

תחבר ה-tag המצחיק זהה הוא לא מחרזת ולא HTML.

כדי לתרן כיצד ממשק-Anchor ממליצים להשתמש בו עם JavaScript והוא הרחבה תחבר עברו JSX הוא נקרא כדי עשויה להזכיר לך שפת תכנות, אבל הוא מגיע עם מלא העוצמה של JSX המשמש אמור להירות.

בחלק הבא. למטה, תוכלו למצוא את היסודות של DOM-Anchor נחקרו כיצד לレンדר אותם ל-React. מייצר "אלמנטים" של JSX הדרושים על מנת שתוכלו להתחילה JSX.

JSX למה?

מאמצת את העבודה כי לוגיקת הרינדור משולבת מיסודה עם יתר לוגיקת המשמש: איך אירועים מטופלים, איך משתנה לאורך זמן, וכיוצא מכינים את הנתונים לתצוגה state-ה.

מצעת הפרדת, והלוגיקה בקבצים נפרדים markup-במקום להפריד באופן מלאכותי בין טכנולוגיות על ידי השמתה **אחריות** בעזרת ייחדות הקשורות בצורה רופפת הנקראות "קומפוננטות" אשר מכילות את שניהם. נחזרו לקומפוננטות **בחלק ההרצאה הזאת** לשוויה לשכנע אתכם, JSX-ב-markup-אחר, אבל אם עדין לא נהנו לכם לשים את ה

אבל רוב האנשים מוצאים את זה מועל בטור סיוע ויזואלי בעת עבודה עם ממשקי JSX-לא דורש שימוש ב-React. להראות הדעות שגיאה והתרעה יותר שימושות React-JavaScript-זה גם אפשרי בתוכה קוד.

עכשו שהסכנו את זה מהדרך, בואו נתחל!

JSX-הטמעת ביטויים ב

על ידי עטיפתו בסוגרים JSX ולאחר מכן משתמשים בו בטור (שם) `name` בדוגמה הבאה, אנו מכריםים על משתנה הנקרו מסלולים:

```
const name = 'גיא פרץ';
const element = <h1>!, שלום, {name}</h1>

ReactDOM.render(
  element,
  document.getElementById('root')
);
```

או, לדוגמה, ביטוי JSX-חוקי בטור סוגרים מסלולים ב-`JavaScript`. ניתן לשים כל `user.firstName`, `user.lastName`, `user.firstName + user.lastName` ועוד. `formatName(user)`-הם כולם ביטויים חוקיים ב-`JavaScript`.

לטור אלמנט JSX-בדוגמה הבאה, אנו מטמיעים את התוצאה של קריאה לפונקציית `formatName(user)`.

```
function formatName(user) {
  return user.firstName + ' ' + user.lastName;
}
```

```
const user = {
  firstName: 'חדר',
  lastName: 'פרץ'
};

const element = (
  <h1>
    Hello, {formatName(user)}!
  </h1>
);
```

[-נסו את זה ב-codepen](#)

על מספר שורות עברו הקריאה. אף על פי שהוא לא נדרש, כאשר עושים זאת, אנו ממליצים גם לעטוף אותו JSX אנו מفضلים את בסוגרים כדי למנוע את החסרונות של [הכנסת נקודה-פסיק אוטומטית](#).

גם הוא בייטו JSX

והם מחושבים לאובייקטים של JavaScript הופכים לקריאות רגילות של פונקציות JSX לאחר ההידור, בייטויים של JavaScript.

להקצות אותו למשתנים, לקבל אותו כארגומנטים, `, for` ולולאות `if` בתוך הצהרות JSX-זה אומר שאתם יכולים להשתמש בו ולהחזיר אותו מפונקציות:

```
function getGreeting(user) {
  if (user) {
    return <h1>Hello, {formatName(user)}!</h1>;
  }
  return <h1>Hello, Stranger.</h1>;
}
```

JSX ציון מאפיינים עם

:ניתן להשתמש בגרשיים כדי לציין מהירותם כמאפיינים

```
const element = <a href="https://www.reactjs.org"> link </a>;
```

:במאפיין JavaScript-תוכלו גם להשתמש בסוגרים מסולסלים כדי להטמע ביטוי ב

```
const element = <img src={user.avatarUrl}></img>;
```

במאפיין. עליים להשתמש בגרשיים (עבור ערכי JavaScript אל תוסיפו גרשים שבב סוגרים מסולסלים בעת הטמעת בייטוי מהירות) או בסוגרים מסולסלים (עבור בייטויים), אך לא בשנייהם באותו המאפיין.

איירהה:

c משתמש בkonventionית שמות מאפיינים בצורה HTML, React DOM - מאשר ל JavaScript - קרוב יותר ל JSX - מכיוון ש amelCase במקומם בשמות של תכונות HTML.

לדוגמה, className, tabIndex - הופך ל className, tabIndex .

JSX ציוןILDIM עם

XML-אם תגית כלשי ריקה, באפשרותכם לסגור אותה מיידית עם `</>`, כמו ב:

```
const element = <img src={user.avatarUrl} />;
```

יכולות להכיל ILDIM JSX תגיות:

```
const element = (
  <div>
    <h1>שלום!<</h1>
    <h2>. טוב לראות אותך כאן</h2>
  </div>
);
```

מונע התקפות הזרקה JSX

JSX-זה בטוח להטמע קלט משתמש ב:

```
const title = response.potentiallyMaliciousInput;
// זה בטוח:
const element = <h1>{title}</h1>;
```

לפni שהוא מרנדר אותם. דבר זה מבטיח שאך פעם JSX-עבור ערכיהם מוטמעים ב **escape** React DOM, כביררת מחדל לא תוכלו להזריק שום דבר שלא כתוב במפורש באפליקציה שלכם. כל דבר מומר למחרחות לפני שהוא מרנדר. זה עוזר למנוע XSS (cross-site-scripting).

מייצג אובייקטיבים JSX

Babel -לקראות ל JSX מנגנון `React.createElement()`.

שתי הדוגמאות הבאות זהות:

```
const element = (
  <h1 className="greeting">
    שלום, עולם!
  </h1>
);
```

```
const element = React.createElement(
  'h1',
  {className: 'greeting'},
  'שלום, עולם!'
);
```

מבצעת כמה בדיקות כדי לעזור לכם לכתוב קוד ללא באגים אבל בעצם היא יוצרת אובייקט `React.createElement()` כזה:

```
// Note: this structure is simplified
const element = {
  type: 'h1',
  props: {
    className: 'greeting',
```

```
        children: [
      ];
    };
  
```

אתם יכולים לחשב עליהם כמו תיאורים של מה שאתם רוצים לראות על React אובייקטים אלה נקראים "אלמנטים של DOM". ולשמור אותו מעדכן קוראת את האובייקטים האלה ומשתמש בהם כדי לבנות את ה React. המשך.

[בחלק הבא](#) React אנו נחקרו את תהליכי רינדור האלמנטים של DOM-L.

:טיפ

מודגשים כראוי JSX וגם קוד ES6 עבור העורך שלק כך שגם קוד ["Babel"](#) אנו ממליצים להשתמש בהגדרת השפה של JSX.

[Go to TOC](#)

JSX In Depth

Fundamentally, JSX just provides syntactic sugar for the `React.createElement(component, props, ...children)` function. The JSX code:

```
<MyButton color="blue" shadowSize={2}>
  Click Me
</MyButton>
```

compiles into:

```
React.createElement(
  MyButton,
  {color: 'blue', shadowSize: 2},
  'Click Me'
)
```

You can also use the self-closing form of the tag if there are no children. So:

```
<div className="sidebar" />
```

compiles into:

```
React.createElement(
  'div',
  {className: 'sidebar'}
)
```

If you want to test out how some specific JSX is converted into JavaScript, you can try out [the online Babel compiler](#).

Specifying The React Element Type

The first part of a JSX tag determines the type of the React element.

Capitalized types indicate that the JSX tag is referring to a React component. These tags get compiled into a direct reference to the named variable, so if you use the JSX `<Foo />` expression, `Foo` must be in scope.

React Must Be in Scope

Since JSX compiles into calls to `React.createElement`, the `React` library must also always be in scope from your JSX code.

For example, both of the imports are necessary in this code, even though `React` and `CustomButton` are not directly referenced from JavaScript:

```
import React from 'react';
import CustomButton from './CustomButton';

function WarningButton() {
```

```
// return React.createElement(CustomButton, {color: 'red'}, null);
return <CustomButton color="red" />;
}
```

If you don't use a JavaScript bundler and loaded React from a `<script>` tag, it is already in scope as the `React` global.

Using Dot Notation for JSX Type

You can also refer to a React component using dot-notation from within JSX. This is convenient if you have a single module that exports many React components. For example, if `MyComponents.DatePicker` is a component, you can use it directly from JSX with:

```
import React from 'react';

const MyComponents = {
  DatePicker: function DatePicker(props) {
    return <div>Imagine a {props.color} datepicker here.</div>;
  }
}

function BlueDatePicker() {
  return <MyComponents.DatePicker color="blue" />;
}
```

User-Defined Components Must Be Capitalized

When an element type starts with a lowercase letter, it refers to a built-in component like `<div>` or `` and results in a string `'div'` or `'span'` passed to `React.createElement`. Types that start with a capital letter like `<Foo />` compile to `React.createElement(Foo)` and correspond to a component defined or imported in your JavaScript file.

We recommend naming components with a capital letter. If you do have a component that starts with a lowercase letter, assign it to a capitalized variable before using it in JSX.

For example, this code will not run as expected:

```
import React from 'react';

// Wrong! This is a component and should have been capitalized:
function hello(props) {
  // Correct! This use of <div> is legitimate because div is a valid HTML tag:
  return <div>Hello {props.toWhat}</div>;
}

function HelloWorld() {
  // Wrong! React thinks <hello /> is an HTML tag because it's not capitalized:
  return <hello toWhat="World" />;
}
```

To fix this, we will rename `hello` to `Hello` and use `<Hello />` when referring to it:

```
import React from 'react';

// Correct! This is a component and should be capitalized:
```

```

function Hello(props) {
  // Correct! This use of <div> is legitimate because div is a valid HTML tag:
  return <div>Hello {props.toWhat}</div>;
}

function HelloWorld() {
  // Correct! React knows <Hello /> is a component because it's capitalized.
  return <Hello toWhat="World" />;
}

```

Choosing the Type at Runtime

You cannot use a general expression as the React element type. If you do want to use a general expression to indicate the type of the element, just assign it to a capitalized variable first. This often comes up when you want to render a different component based on a prop:

```

import React from 'react';
import { PhotoStory, VideoStory } from './stories';

const components = {
  photo: PhotoStory,
  video: VideoStory
};

function Story(props) {
  // Wrong! JSX type can't be an expression.
  return <components[props.storyType] story={props.story} />;
}

```

To fix this, we will assign the type to a capitalized variable first:

```

import React from 'react';
import { PhotoStory, VideoStory } from './stories';

const components = {
  photo: PhotoStory,
  video: VideoStory
};

function Story(props) {
  // Correct! JSX type can be a capitalized variable.
  const SpecificStory = components[props.storyType];
  return <SpecificStory story={props.story} />;
}

```

Props in JSX

There are several different ways to specify props in JSX.

JavaScript Expressions as Props

You can pass any JavaScript expression as a prop, by surrounding it with `{}`. For example, in this JSX:

```
<MyComponent foo={1 + 2 + 3 + 4} />
```

For `MyComponent`, the value of `props.foo` will be `10` because the expression `1 + 2 + 3 + 4` gets evaluated.

`if` statements and `for` loops are not expressions in JavaScript, so they can't be used in JSX directly. Instead, you can put these in the surrounding code. For example:

```
function NumberDescriber(props) {
  let description;
  if (props.number % 2 == 0) {
    description = <strong>even</strong>;
  } else {
    description = <i>odd</i>;
  }
  return <div>{props.number} is an {description} number</div>;
}
```

You can learn more about [conditional rendering](#) and [loops](#) in the corresponding sections.

String Literals

You can pass a string literal as a prop. These two JSX expressions are equivalent:

```
<MyComponent message="hello world" />
<MyComponent message={'hello world'} />
```

When you pass a string literal, its value is HTML-unescaped. So these two JSX expressions are equivalent:

```
<MyComponent message=&lt;3&gt; />
<MyComponent message={'<3'} />
```

This behavior is usually not relevant. It's only mentioned here for completeness.

Props Default to "True"

If you pass no value for a prop, it defaults to `true`. These two JSX expressions are equivalent:

```
<MyTextBox autocomplete />
<MyTextBox autocomplete={true} />
```

In general, we don't recommend *not* passing a value for a prop, because it can be confused with the [ES6 object shorthand](#) `{foo}` which is short for `{foo: foo}` rather than `{foo: true}`. This behavior is just there so that it matches the behavior of HTML.

Spread Attributes

If you already have `props` as an object, and you want to pass it in JSX, you can use `...` as a "spread" syntax to pass the whole props object. These two components are equivalent:

```

function App1() {
  return <Greeting firstName="Ben" lastName="Hector" />;
}

function App2() {
  const props = {firstName: 'Ben', lastName: 'Hector'};
  return <Greeting {...props} />;
}

```

You can also pick specific props that your component will consume while passing all other props using the spread syntax.

```

const Button = props => {
  const { kind, ...other } = props;
  const className = kind === "primary" ? "PrimaryButton" : "SecondaryButton";
  return <button className={className} {...other} />;
};

const App = () => {
  return (
    <div>
      <Button kind="primary" onClick={() => console.log("clicked!")}>
        Hello World!
      </Button>
    </div>
  );
};

```

In the example above, the `kind` prop is safely consumed and *is not* passed on to the `<button>` element in the DOM. All other props are passed via the `...other` object making this component really flexible. You can see that it passes an `onClick` and `children` props.

Spread attributes can be useful but they also make it easy to pass unnecessary props to components that don't care about them or to pass invalid HTML attributes to the DOM. We recommend using this syntax sparingly.

Children in JSX

In JSX expressions that contain both an opening tag and a closing tag, the content between those tags is passed as a special prop: `props.children`. There are several different ways to pass children:

String Literals

You can put a string between the opening and closing tags and `props.children` will just be that string. This is useful for many of the built-in HTML elements. For example:

```
<MyComponent>Hello world!</MyComponent>
```

This is valid JSX, and `props.children` in `MyComponent` will simply be the string `"Hello world!"`. HTML is unescaped, so you can generally write JSX just like you would write HTML in this way:

```
<div>This is valid HTML &amp; JSX at the same time.</div>
```

JSX removes whitespace at the beginning and ending of a line. It also removes blank lines. New lines adjacent to tags are removed; new lines that occur in the middle of string literals are condensed into a single space. So these all render to the same thing:

```
<div>Hello World</div>

<div>
  Hello World
</div>

<div>
  Hello
  World
</div>

<div>
  Hello World
</div>
```

JSX Children

You can provide more JSX elements as the children. This is useful for displaying nested components:

```
<MyContainer>
  <MyFirstComponent />
  <MySecondComponent />
</MyContainer>
```

You can mix together different types of children, so you can use string literals together with JSX children. This is another way in which JSX is like HTML, so that this is both valid JSX and valid HTML:

```
<div>
  Here is a list:
  <ul>
    <li>Item 1</li>
    <li>Item 2</li>
  </ul>
</div>
```

A React component can also return an array of elements:

```
render() {
  // No need to wrap list items in an extra element!
  return [
    // Don't forget the keys :)
    <li key="A">First item</li>,
    <li key="B">Second item</li>,
    <li key="C">Third item</li>,
  ];
}
```

JavaScript Expressions as Children

You can pass any JavaScript expression as children, by enclosing it within `{ }`. For example, these expressions are equivalent:

```
<MyComponent>foo</MyComponent>
<MyComponent>{'foo'}</MyComponent>
```

This is often useful for rendering a list of JSX expressions of arbitrary length. For example, this renders an HTML list:

```
function Item(props) {
  return <li>{props.message}</li>;
}

function TodoList() {
  const todos = ['finish doc', 'submit pr', 'nag dan to review'];
  return (
    <ul>
      {todos.map((message) => <Item key={message} message={message} />)}
    </ul>
  );
}
```

JavaScript expressions can be mixed with other types of children. This is often useful in lieu of string templates:

```
function Hello(props) {
  return <div>Hello {props.addressee}!</div>;
}
```

Functions as Children

Normally, JavaScript expressions inserted in JSX will evaluate to a string, a React element, or a list of those things. However, `props.children` works just like any other prop in that it can pass any sort of data, not just the sorts that React knows how to render. For example, if you have a custom component, you could have it take a callback as `props.children`:

```
// Calls the children callback numTimes to produce a repeated component
function Repeat(props) {
  let items = [];
  for (let i = 0; i < props.numTimes; i++) {
    items.push(props.children(i));
  }
  return <div>{items}</div>;
}

function ListOfTenThings() {
  return (
    <Repeat numTimes={10}>
      {(index) => <div key={index}>This is item {index} in the list</div>}
    </Repeat>
  );
}
```

Children passed to a custom component can be anything, as long as that component transforms them into something React can understand before rendering. This usage is not common, but it works if you want to stretch what JSX is capable of.

Booleans, Null, and Undefined Are Ignored

`false`, `null`, `undefined`, and `true` are valid children. They simply don't render. These JSX expressions will all render to the same thing:

```
<div />
<div></div>
<div>{false}</div>
<div>{null}</div>
<div>{undefined}</div>
<div>{true}</div>
```

This can be useful to conditionally render React elements. This JSX renders the `<Header />` component only if `showHeader` is `true`:

```
<div>
  {showHeader && <Header />}
  <Content />
</div>
```

One caveat is that some "falsy" values, such as the `0` number, are still rendered by React. For example, this code will not behave as you might expect because `0` will be printed when `props.messages` is an empty array:

```
<div>
  {props.messages.length &&
   <MessageList messages={props.messages} />
  }
</div>
```

To fix this, make sure that the expression before `&&` is always boolean:

```
<div>
  {props.messages.length > 0 &&
   <MessageList messages={props.messages} />
  }
</div>
```

Conversely, if you want a value like `false`, `true`, `null`, or `undefined` to appear in the output, you have to [convert it to a string](#) first:

```
<div>
  My JavaScript variable is {String(myVariable)}.
</div>
```

[Go to TOC](#)

Legacy Context

Note:

The legacy context API will be removed in a future major version. Use the [new context API](#) introduced with version 16.3. The legacy API will continue working for all 16.x releases.

How To Use Context

This section documents a legacy API. See the [new API](#).

Suppose you have a structure like:

```
class Button extends React.Component {
  render() {
    return (
      <button style={{background: this.props.color}}>
        {this.props.children}
      </button>
    );
  }
}

class Message extends React.Component {
  render() {
    return (
      <div>
        {this.props.text} <Button color={this.props.color}>Delete</Button>
      </div>
    );
  }
}

class MessageList extends React.Component {
  render() {
    const color = "purple";
    const children = this.props.messages.map((message) =>
      <Message text={message.text} color={color} />
    );
    return <div>{children}</div>;
  }
}
```

In this example, we manually thread through a `color` prop in order to style the `Button` and `Message` components appropriately. Using context, we can pass this through the tree automatically:

```
import PropTypes from 'prop-types';

class Button extends React.Component {
```

```

render() {
  return (
    <button style={{background: this.context.color}}>
      {this.props.children}
    </button>
  );
}

Button.contextTypes = {
  color: PropTypes.string
};

class Message extends React.Component {
  render() {
    return (
      <div>
        {this.props.text} <Button>Delete</Button>
      </div>
    );
  }
}

class MessageList extends React.Component {
  getChildContext() {
    return {color: "purple"};
  }

  render() {
    const children = this.props.messages.map((message) =>
      <Message text={message.text} />
    );
    return <div>{children}</div>;
  }
}

MessageList.childContextTypes = {
  color: PropTypes.string
};

```

By adding `childContextTypes` and `getChildContext` to `MessageList` (the context provider), React passes the information down automatically and any component in the subtree (in this case, `Button`) can access it by defining `contextTypes`.

If `contextTypes` is not defined, then `context` will be an empty object.

Note:

`React.PropTypes` has moved into a different package since React v15.5. Please use [the prop-type library instead](#) to define `contextTypes`.

We provide a [codemod script](#) to automate the conversion.

Parent-Child Coupling

This section documents a legacy API. See the [new API](#).

Context can also let you build an API where parents and children communicate. For example, one library that works this way is [React Router V4](#):

```
import { BrowserRouter as Router, Route, Link } from 'react-router-dom';

const BasicExample = () => (
  <Router>
    <div>
      <ul>
        <li><Link to="/">Home</Link></li>
        <li><Link to="/about">About</Link></li>
        <li><Link to="/topics">Topics</Link></li>
      </ul>

      <hr />

      <Route exact path="/" component={Home} />
      <Route path="/about" component={About} />
      <Route path="/topics" component={Topics} />
    </div>
  </Router>
);
```

By passing down some information from the `Router` component, each `Link` and `Route` can communicate back to the containing `Router`.

Before you build components with an API similar to this, consider if there are cleaner alternatives. For example, you can pass entire React components as props if you'd like to.

Referencing Context in Lifecycle Methods

This section documents a legacy API. See the [new API](#).

If `contextTypes` is defined within a component, the following [lifecycle methods](#) will receive an additional parameter, the `context` object:

- `constructor(props, context)`
- `componentWillReceiveProps(nextProps, nextContext)`
- `shouldComponentUpdate(nextProps, nextState, nextContext)`
- `componentWillUpdate(nextProps, nextState, nextContext)`

Note:

As of React 16, `componentDidUpdate` no longer receives `prevContext`.

Referencing Context in Function Components

This section documents a legacy API. See the [new API](#).

Function components are also able to reference `context` if `contextTypes` is defined as a property of the function. The following code shows a `Button` component written as a function component.

```
import PropTypes from 'prop-types';

const Button = ({children}, context) =>
  <button style={{background: context.color}}>
    {children}
  </button>

Button.contextTypes = {color: PropTypes.string};
```

Updating Context

This section documents a legacy API. See the [new API](#).

Don't do it.

React has an API to update context, but it is fundamentally broken and you should not use it.

The `getChildContext` function will be called when the state or props changes. In order to update data in the context, trigger a local state update with `this.setState`. This will trigger a new context and changes will be received by the children.

```
import PropTypes from 'prop-types';

class MediaQuery extends React.Component {
  constructor(props) {
    super(props);
    this.state = {type: 'desktop'};
  }

  getChildContext() {
    return {type: this.state.type};
  }

  componentDidMount() {
    const checkMediaQuery = () => {
      const type = window.matchMedia("(min-width: 1025px)").matches ? 'desktop' : 'mobile';
      if (type !== this.state.type) {
        this.setState({type});
      }
    };
    checkMediaQuery();
    window.addEventListener('resize', checkMediaQuery);
  }
}
```

```
};

    window.addEventListener('resize', checkMediaQuery);
    checkMediaQuery();
}

render() {
    return this.props.children;
}

MediaQuery.childContextTypes = {
    type: PropTypes.string
};
```

The problem is, if a context value provided by component changes, descendants that use that value won't update if an intermediate parent returns `false` from `shouldComponentUpdate`. This is totally out of control of the components using context, so there's basically no way to reliably update the context. [This blog post](#) has a good explanation of why this is a problem and how you might get around it.

Event Pooling

Note

This page is only relevant for React 16 and earlier, and for React Native.

React 17 on the web **does not** use event pooling.

[Read more](#) about this change in React 17.

The `SyntheticEvent` objects are pooled. This means that the `SyntheticEvent` object will be reused and all properties will be nullified after the event handler has been called. For example, this won't work:

```
function handleChange(e) {
  // This won't work because the event object gets reused.
  setTimeout(() => {
    console.log(e.target.value); // Too late!
  }, 100);
}
```

If you need to access event object's properties after the event handler has run, you need to call `e.persist()`:

```
function handleChange(e) {
  // Prevents React from resetting its properties:
  e.persist();

  setTimeout(() => {
    console.log(e.target.value); // Works
  }, 100);
}
```

למעלה-State-הרמת ה

המשותף עד state-לעתים קרובות, מספר קומפוננטות צריכות לשקר את אותם נתונים. אנו ממליצים להרים את הלאב הקדמון הקרוב ביותר. בוואו נראה איך זה עובד.

בחלק זה, ניצור מחשבון טמפרטורה המחשב אם המים ירתחו בטמפרטורה נתונה.

ומדפיס אם props בתור `celsius` -היא מקבלת את הטמפרטורה ב `BoilingVerdict`. נתחיל עם קומפוננטה שנקראת הטמפרטורה מספיקה כדי להרתה את המים:

```
function BoilingVerdict(props) {
  if (props.celsius >= 100) {
    return <p>המים ירתחו.</p>;
  }
  return <p>המים לא ירתחו.</p>;
}
```

המאפשר לכם להזין את הטמפרטורה, ושותר `<input>` לאחר מכן, ניצור קומפוננטה שנקראת `calculator`. היא מודנרת. `<input>` על הערך שלה ב `this.state.temperature`.

עבור ערך הקלט הנוכחי של `BoilingVerdict` את `calculator`, היא מודנרת.

```
class Calculator extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.state = {temperature: ''};
  }

  handleChange(e) {
    this.setState({temperature: e.target.value});
  }

  render() {
    const temperature = this.state.temperature;
    return (
      <fieldset>
        <legend>הכנס טמפרטורה בצלזיוס</legend>
        <input
          value={temperature}
          onChange={this.handleChange} />
        <BoilingVerdict
          celsius={parseFloat(temperature)} />
      </fieldset>
    );
  }
}
```

[-נסה זאת ב-CodePen](#)

הוסףת קלט שני

הדרישה החדשה שלנו היא, בנוסף לקלט בצלזיויס, אנו מספקים קלט בפרנהייט, והם נשארים מסונכרים.

חדש prop אנו נוסיף אליה `Calculator` מתוך `TemperatureInput` אנחנו יכולים להתחיל על ידי חילוץ קומפוננטת `scale` או `"c"` או `"f"` שיכל להיות :

```
const scaleNames = {
  c: 'צֶלְזִיּוֹן',
  f: 'פָּרְנַחִיּוֹת'
};

class TemperatureInput extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.state = {temperature: ''};
  }

  handleChange(e) {
    this.setState({temperature: e.target.value});
  }

  render() {
    const temperature = this.state.temperature;
    const scale = this.props.scale;
    return (
      <fieldset>
        <legend>הכנס טמפרטורה ב-</legend>
        <input value={temperature}>
          onChange={this.handleChange}
        </input>
      </fieldset>
    );
  }
}
```

כדי שירנדר שני קלטי טמפרטורה נפרדים `Calculator`icut אנו יכולים לשנות את :

```
class Calculator extends React.Component {
  render() {
    return (
      <div>
        <TemperatureInput scale="c" />
        <TemperatureInput scale="f" />
      </div>
    );
  }
}
```

[-נסו זאת ב-CodePen](#)

יש לנו שני קלטים עכשוין, אבל כאשר אוטם מכנים את הטמפרטורה באחד מהם, השני אינו מותעדכן. זה סותר את הדרישתנו: אנחנו רוצים לשמור אותן מסונכרנים.

אינו ידוע את הטמפרטורה הנוכחית `BoilingVerdict` - מ- `Calculator` .- `Calculator` אנחנו גם לא יכולים להציג את -משום שהוא מושתתת בתוך `TemperatureInput` .

כתיבת פונקציית המרת

ראשית, נכתוב שתי פונקציות כדי להמיר מצלזוס לפורנהייט ובחזרה:

```

function toCelsius(fahrenheit) {
  return (fahrenheit - 32) * 5 / 9;
}

function toFahrenheit(celsius) {
  return (celsius * 9 / 5) + 32;
}

```

ופונקציית המרת CelsiusToFahrenheit(`temperature`) שמייצרת ממספרות מירוחות מילויים. נכתוב פונקציה נוספת שלוקחת מחרוזת וממחזרה מחרוזת. נשתמש בה כדי ליחס את הערך של קלט אחד על סמך קלט אחר.

לא חוקית, והיא שומרת את הפולט מעוגל למספרה העשרונית (`temperature`) היא ממחזרת מחרוזת ריקה עבור טמפרטורה השלישית לאחר הנקודה:

```

function tryConvert(temperature, convert) {
  const input = parseFloat(temperature);
  if (Number.isNaN(input)) {
    return '';
  }
  const output = convert(input);
  const rounded = Math.round(output * 1000) / 1000;
  return rounded.toString();
}

```

לדוגמה, `tryConvert('abc', toCelsius)` ממחזר מחרוזת ריקה, ו-`tryConvert('10.22', toFahrenheit)` ממחזר `'50.396'`.

למעלה-State-הרמת ה

מקומי-state-מחזיקות את הערך שליהן באופן עצמאי ב `TemperatureInput` -כך, שמי קומפוננטות ה:

```

class TemperatureInput extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.state = {temperature: ''};
  }

  handleChange(e) {
    this.setState({temperature: e.target.value});
  }

  render() {
    const temperature = this.state.temperature;
    // ...
  }
}

```

למרות זאת, אנחנו רוצחים שני הקלטים האלו יהיו מסונכרנים אחד עם השני. כאשר אנו מעדכנים את קלט צליזוס, קלט פרנהייט צריך לשחק את הטמפרטורה המורמת, והוא דבר בכיוון השני.

נעשה על ידי העברתו אל האב הקדמון המשותף הקשור ביותר של הקומפוננטות הזקוקות לו. זה state-שיתוף ה-React-ב-`TemperatureInput` ולנעביר אותו לתוך `Calculator`. אנו נסיר את ה-state-נקרא "הרמת ה" במקום.

לטמפרטורה ("source of truth"), הוא הופך להיות "מקור האמת state-state" והוא הבעלים של ה `Calculator` אם של `props`-הנוכחת בשני הקטלים. זה יכול להנחות את שניהם להשתמש בערכיהם כך שיהיו עקביים אחד עם השני. מכיוון שה שני הקטלים יהיו תמיד מסוכנים, `Calculator` מגיעים אליהו קומפוננטת אב `TemperatureInput` שת קומפוננטות.

באו נראה איך זה עובד צעד אחר צעד

בڪומפוננטה `this.state.temperature` עם `this.props.temperature` אנו מחליפים כבר קיימ, למורות שנטצר `this.props.temperature` `TemperatureInput` לעת עתה, בואו נמשיך להעמיד פנים ש. בעטיז `-Calculator` להעיבר אותו מ

```
render() {
  // 🔞: const temperature = this.state.temperature;
  const temperature = this.props.temperature;
  // ...
}
```

יכל `TemperatureInput`-המקומי, ה `state`-היה ב `temperature`-המזהה. כאשר ה `props`-אנחנו יודעים ש `prop`, מגע מהוורה בטור `temperature` כדי לשנות אותו. למורת זאת, עכשו כאשר `this.setState()`-פושט לקרוא ל-`TemperatureInput` און שליטה עליו -ל.

וגם `value` מקבל גם `<input>` DOM-זה בדרך כלל נעשה על ידי הפיכת קומפוננטה ל-"נשלהת". בדוק כמו ב-`TemperatureInput` המותאם אישית לקבל גם `props` ו-`onPropsChange` המהוורה שלו `Calculator`.

-רווחה לעדכו את הטמפרטורה שלו, והוא יקרא לו -עכשו, TemperatureInput שה.props.onTemperatureChange :

```
handleChange(e) {
  // מעדכן מדחום: this.setState({temperature: e.target.value});
  this.props.onTemperatureChange(e.target.value);
  // ...
}
```

শিমু লব

בקומפוננטות מותאמות אישית. `temperature` או `onTemperatureChange`-אין שימוש מיוחד לשם זה. `value`-`onchange` יוכלו לקרוא להם כל דבר אחר, כמו לקרוא להם `היא קובננציה נפוצה`.

היא. Calculator על ידי קומפוננטת ההוֹרָה temperature-prop יועבר יחד עם ה-temperature.onTemperatureChange המוקומי שלה, ובכך תרנnder מחדש את שני הקלטים עם ערכים חדשים. אנו נסתכל על state-state-טיפול בשינוי על ידי שינוי של Calculator. בקרוב מאוד Calculator המימוש החדש של

הسرנו `TemperatureInput`. בואו נסכם את השינויים שלנו לkomponenat, `Calculator`-לפני שנצלול לתוך השינויים ב-`this.state.temperature`, אנחנו קוראים עכשו את המוקומי, ובמקום לקרוא את `this.state.temperature` אנחנו קוראים `this.props.temperature` לעשות שינוי, עכשו אנחנו `this.setState()`-במקום לקרוא ל-`Calculator`:
כשהם רצים `this.props.onTemperatureChange()` אשר יספק על ידי,

```
class TemperatureInput extends React.Component {  
  constructor(props) {  
    super(props);  
    this.handleChange = this.handleChange.bind(this);
```

```

}

handleChange(e) {
  this.props.onTemperatureChange(e.target.value);
}

render() {
  const temperature = this.props.temperature;
  const scale = this.props.scale;
  return (
    <fieldset>
      <legend>הכנס טמפרטורה ב-</legend>
      <input value={temperature}>
        onChange={this.handleChange} />
    </fieldset>
  );
}
}

```

עכשו באו נפנה לkomponenT.

ש-הרמוני מהקלטים, state-המקומי שלה. זהו ה state-ב `temperature` ו-`scale` אנו נשמר את הקלטים הנוכחים והוא ישתמש "מקור האמת" עבור שנייהם. זהו ייצוג מינימלי של כל הנתונים שאחננו צריכים לדעת על מנת לרנדר את שני הקלטים.

יהיה `Calculator` של komponenT-state-לדוגמה, אם נכניס 37 לתוכה הקלט של צליזוס, ה

```
{
  temperature: '37',
  scale: 'c'
}
```

יהיה `Calculator` של state-אם מאוחר יותר נערוך את השדה פרנהייט כך שהיא 212, ה

```
{
  temperature: '212',
  scale: 'f'
}
```

הינו יכולים לאחסן את הערך של שני הקלטים אבל מסתבר שהוא יהיה מיותר. זה מספיק לאחסן את הערך של הקלט האחרון שהשתנה, ואת המדד שהוא מייצג. לאחר מכן יכולים להסיק את הערך של הקלט האחרון בהתאם על ערכי `temperature` בלבד - `scale`.

הקלטים נשארים מסונכרנים מכיוון שהערכים שלהם מחושבים מאותו state:

```

class Calculator extends React.Component {
  constructor(props) {
    super(props);
    this.handleCelsiusChange = this.handleCelsiusChange.bind(this);
    this.handleFahrenheitChange = this.handleFahrenheitChange.bind(this);
    this.state = {temperature: '', scale: 'c'};
  }

  handleCelsiusChange(temperature) {
    this.setState({scale: 'c', temperature});
  }
}

```

```

handleFahrenheitChange(temperature) {
    this.setState({scale: 'f', temperature});
}

render() {
    const scale = this.state.scale;
    const temperature = this.state.temperature;
    const celsius = scale === 'f' ? tryConvert(temperature, toCelsius) :
temperature;
    const fahrenheit = scale === 'c' ? tryConvert(temperature, toFahrenheit) :
temperature;

    return (
        <div>
            <TemperatureInput
                scale="c"
                temperature={celsius}
                onTemperatureChange={this.handleCelsiusChange} />
            <TemperatureInput
                scale="f"
                temperature={fahrenheit}
                onTemperatureChange={this.handleFahrenheitChange} />
            <BoilingVerdict
                celsius={parseFloat(celsius)} />
        </div>
    );
}
}

```

[-נסו זאת ב-CodePen](#)

במקרה שלנו, זהה המתודה `onTemperatureChange`-ב-`TemperatureInput` ב-`Calculator` מתחדדים. אחד הקלטים מקבל את הערך כפי שהוא, וכך כל קלט משתמש נשמר, וערך הקלט האחרון תמיד מחושב מחדש על בסיס הערך.

בוואו נסכם את מה שקרה בעת עיבת קלט:

- React במקורה שלנו, זהה המתודה `onTemperatureChange`-ב-`TemperatureInput`. ב-`TemperatureInput` קוראת לפונקציה שמצוינה כ `onTemperatureChange`.
- קוראת `onTemperatureChange` ב-`TemperatureInput` ל-`Calculator` המתודה `handleChange`. סופקו על ידי רכיב האב שלו, כולל `props.onTemperatureChange`, עם הערך החדש. הוא רונדר `Calculator`.
- צליזוס היא `Calculator` של `TemperatureInput` ציון כי `Calculator` הוא רונדר קודם לכך `TemperatureInput` מתחדדת. הוא מתחדדת `onTemperatureChange` של `TemperatureInput` ב-`Calculator`, והוא מתחדדת `onTemperatureChange` של `Calculator`. תראה כתלות בקלט אשר עברנו `Calculator`.
- לרונדר מחדש את עצמה על ידי קוראה ל-React-מבקשת מ `Calculator` בתוקן המתחדשות הללו, הקומפוננטה `this.setState()`. עם ערך הקלט החדש והמדד הנוכחי של הקלט שערכנו זה עתה.
- כדי ללמוד איך ממשκ המשמש צריך להיראות. הערכיהם של `Calculator` של קומפוננטות `TemperatureInput` קוראת למתחדמת `render`. של שני הקלטים מחושבים מחדש בהתאם לטמפרטורה הנוכחי ולמדד הפעיל. ההמרה של הטמפרטורה מבוצעת כאן.
- החדשים שלהם שלהן `props.onTemperatureChange` של כל אחת מקומפוננטות `TemperatureInput` קוראת למתחדמת `render`. היא לומדת איך ממשκ המשמש להם צריך להיראות `Calculator`. שמצוינים על ידי `Calculator`.
- `BoilingVerdict` של קומפוננטה `BoilingVerdict` קוראת למתחדמת `render` כשהיא מעבירה את הטמפרטורה בצליזוס כ-`props.celsius`.

- React DOM מ对照检查 מצב הרותicha ועם התאמת הערכים הרצויים בקלט. הקלט שערךנו זה עתה DOM-معدכן את ה DOM.
 - מקבל את הערך הנוכחי שלו, והקלט האחר מתעדכן לטופרתו לאחר ההמרה.
- כל עדכון עבר או אוטם השלבים כך שהקלטים ישארו מסונכרנים.

לקחים שנלמדו

מתווסף לראשונה לקומפוננטה state-בדרך כלל, ה React צריך להיות "מקור אמת" ייחיד עבור כל נתון המשתנה באפליקציה הזוקקה לו. לאחר מכן, אם קומפוננטות אחרות גם צרכות אותו, אותן יכולים להרים אותו אל האב הקדמון המשותף הקרוב בין קומפוננטות שונות, עליו להסתמך על **זרימת הנתונים מלמעלה למטה** state-ביוור. במקומות מסוימים סנכרון את

דו-כיווני, אך הרווח הוא שנדרש פחות עבודה binding מאשר בגישה "boilerplate" כרוכה יותר בכתיבה קוד state-הרמתה חי". בתוך אישחי קומפוננטה וקומפוננטה זו בלבד יכולה לשנות אותו, שיטה הפנים state כדי לאחד ולבדח באגים. לאחר שכל לבאים מופחת באופן משמעותי. בנוסף, אפשרותם למשתמש כל לוגיקה מותאמת אישית כדי לדוחות או לשנות קלט משתמש

לדוגמה, במקרה לאחסן גם את state-הזה כנראה לא צריך להיות ב props-אם אנחנו יכולים לגזר מהו מה האחורונה ואת הממד (temperature) אנו מחשנים רק את הטופרתו, fahrenheitValue וגם את scale () שלה. הערך של הקלט האחר יכול תמיד להיות מחושב מהם בתודה () זה מאפשר לנו לנוקות או להחיל. render () עיגול לשדה האחר מבלי לאבד כל דיקט בקלט המשתמש

ולבעור props-כדי לבדוק את React כאשר אתם רואים משהו שהוא שימוש המשתמש, תוכלו להשתמש **בכלי הפיתוח של זה** מאפשר לכם לעקוב אחר הבאים עד מקורם state. למעלה בעץ עד שתמצאו את הקומפוננטה האחראית לעדכון ה

Enter temperature in Celsius:

Enter temperature in Fahrenheit:

The water would not boil.

The screenshot shows the React DevTools interface. At the top, there are tabs for Elements, React (which is selected), Console, Sources, Network, Timeline, Profiles, and a menu. Below the tabs, there are checkboxes for Trace React Updates, Highlight Search, and Use Regular Expressions. The main area displays the component tree under the <Calculator> component. The tree includes <TemperatureInput scale="c" temperature="" onTemperatureChange=>, <TemperatureInput scale="f" temperature="" onTemperatureChange=>, and <BoilingVerdict celsius=null>...</BoilingVerdict>. To the right of the tree, there is a panel titled <Calculator> (\$r in the console) showing the Props (an empty object) and State (scale: "c" and temperature: "").

[Go to TOC](#)

רישימות ופתחות

ראשית, בואו נסקר איך לשנות רישימות ב-JavaScript.

ולהכפיל את הערכים `numbers` כדי לחת מערך של מספרים `map()` בהתחשב בכך שהלן, אנחנו משתמשים בפונקציית `map()` ומגדירים אותו `doubled` לשתנה `map()` לשתנה `doubled` למשתנה `(number) => number * 2`.

```
const numbers = [1, 2, 3, 4, 5];
const doubled = numbers.map((number) => number * 2);
console.log(doubled);
```

הקוד הזה ידפיס `[10, 8, 6, 4, 2]` לקונסול.

שינוי מערכם לרשימות של [אלמנטים](#) הוא כמעט זהה - ב-

רינדור קומפוננטות מרובות

באמצעות סוגרים מסוללים [JSX](#)-אתה יכול לבנות אוסף של אלמנטים ולכלול אותם ב `{ }`.

אלמנט לכל `` אנו מחזירים `map()` מערך המספרים באמצעות פונקציית `numbers` למטה, אנו רצים על `listItems` :
-פריט. לבסוף, אנו מקצים את המערך החזר ל

```
const numbers = [1, 2, 3, 4, 5];
const listItems = numbers.map((number) =>
  <li>{number}</li>
);
```

[DOM](#)-ומרדנדים את ה `` כולם בתוך אלמנט `listItems`, אנו מכילים את המערך.

```
<ul>{listItems}</ul>
```

[CodePen](#) -נסה זאת ב

הקוד הזה מציג רשימה של מספרים מ-1 עד 5.

קומפוננטת רשימה בסיסית

בדרכ כל אלו נרender רשימה בתחום [קומפוננטה](#).

מערך של מספרים ומדפסה רשימה של `numbers` אנחנו יכולים לשכתב את הדוגמה הקודמת לkomponenta שמקבלת אלמנטים.

```
function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) =>
    <li>{number}</li>
  );
  return (
    <ul>{listItems}</ul>
  );
}
```

```
const numbers = [1, 2, 3, 4, 5];
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<NumberList numbers={numbers} />);
```

כאשר תריץ את הקוד זהה, תופיע אזהרה שمفצת ציריך להיות מסופק לפיריטים ברשימה. "مفצת" הוא תכונה מיוחדת מסוג מחרוזת שאתיה צריך להכפיל מתי יוצר רשימה של אלמנטים. נדונן מדוע זה חשוב בחלק הבא.

ונתקן את בעיית חסרונו המפתחה () `numbers.map()` בפיריטים שלנו בתוך `key` בוואו נקצתה.

```
function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) =>
    <li key={number.toString()}>
      {number}
    </li>
  );
  return (
    <ul>{listItems}</ul>
  );
}
```

[-נסה זאת ב-CodePen](#)

مفצתות

ליזות אילו פיריטים השתנו, נוספו או נמחקו. מפתחות אמורים להינטע לאלמנטים בתוך המערך כדי React-مفצתות עוזרים למתת לאלמנטים זהה קבואה:

```
const numbers = [1, 2, 3, 4, 5];
const listItems = numbers.map((number) =>
  <li key={number.toString()}>
    {number}
  </li>
);
```

ID הדריך הטובה ביותר לבחור מפתח שמצויה באופן ייחודי לפיריט מהרשימה בין אחיו. רוב תבחר מהנתונים שלך כمفצתה:

```
const todoItems = todos.map((todo) =>
  <li key={todo.id}>
    {todo.text}
  </li>
);
```

קבוע לרנדור הפיריטים, אתה רשאי להשתמש באינדקס של האיבר כمفצת בתוך מוצא אחרון ID כאשר אין לך:

```
const todoItems = todos.map((todo, index) =>
  // עשה זאת אך ורק אם לפיריט אין ID קבוע
  <li key={index}>
    {todo.text}
  </li>
);
```

אנחנו לא ממליצים להשתמש באינדקסים עבור מפתחות אם סדר הפריטים ישנה. זה יכול להשפיע לרעה על הביצועים ולגרום בשביל **הסביר עמוק על ההשפות השיליות של שימוש Robin Pokorny** של הקומפוננטה. עיין במאמר של state-state בעיות בReact **באיינדקס כמפתח**. אם תבחר לא להקצות מפתח מפורש לפריט ברשימה אז ישתמש כברירת מחדל באינדקס כמפתח.

הנה **הסביר עמוק על למה מפתחות נחוצים** אם אתה מעוניין למדוד יותר.

חולוץ קומפוננטות עם מפתחות

מפתחות הגיוניים רק בהקשר של מערכיים.

שבמערך מסויר `</>` אתה תעדיף לשמור את המפתח באלמנט `ListItem`, אם אתה **מחלץ** קומפוננטת `ListItem` **לדוגמא**, עצמו ``-שב `ListItem` באלמנט.

דוגמה: שימוש לא נכון ב מפתח

```
function ListItem(props) {
  const value = props.value;
  return (
    // טעות! אין צורך לציין את המפתח כאן
    <li key={value.toString()}>
      {value}
    </li>
  );
}

function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) =>
    // טעות! אתה צריך לציין את המפתח כאן
    <ListItem value={number} />
  );
  return (
    <ul>
      {listItems}
    </ul>
  );
}
```

דוגמה: שימוש נכון ב מפתח

```
function ListItem(props) {
  // נכון! אין צורך לציין את המפתח כאן
  return <li>{props.value}</li>;
}

function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) =>
    // נכון! יש לציין את המפתח בתוך המערך
    <ListItem key={number.toString()} value={number} />
  );
  return (
    <ul>
      {listItems}
    </ul>
  );
}
```

CodePen - נסxo זאת ב

צריכים מפתחות `(map)` כל אכבע טוב הוא שאלמנטים בתוך.

מפתחות חייבים להיות ייחודיים בין אחיהם.

מפתחות חייבים להיות ייחודיים בין אחיהם. למרות זאת הם לא צריכים להיות ייחודיים באופן גלובלי. אנחנו יכולים להשתמש באותם מפתחות כאשר אנו מיצרים שני מערכיים שונים.

```
function Blog(props) {
  const sidebar = (
    <ul>
      {props.posts.map((post) =>
        <li key={post.id}>
          {post.title}
        </li>
      )}
    </ul>
  );
  const content = props.posts.map((post) =>
    <div key={post.id}>
      <h3>{post.title}</h3>
      <p>{post.content}</p>
    </div>
  );
  return (
    <div>
      {sidebar}
      <hr />
      {content}
    </div>
  );
}

const posts = [
];

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Blog posts={posts} />);
```

CodePen - נסxo זאת ב

אבל הם אינם עוברים לkomponenTות שלך. אם אתה צריך את אותו הערך בkomponenTה שלך, React-מפתחות משרות קרמי לuem שם אחר prop-העבר אותו בצורה מפורשת כ

```
const content = posts.map((post) =>
  <Post
    key={post.id}
    id={post.id}
    title={post.title} />
);
```

אפשר לקרוא את `Post` עם הדוגמה למטה, komponenTה `props.id` אבל לא את `props.key`.

JSX בתוך `map()` הטעמת

JSX-וככלנו אותו ב `listItems` בדוגמה לעליה הצהינו על משתנה נפרד:

```

function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) =>
    <ListItem key={number.toString()} value={number} />
  );
  return (
    <ul>
      {listItems}
    </ul>
  );
}

```

אפשר להטמי [כל כניסה](#) בסוגרים מסולסלים כדי שנוכל להציג את תוכנת JSX :

```

function NumberList(props) {
  const numbers = props.numbers;
  return (
    <ul>
      {numbers.map((number) =>
        <ListItem key={number.toString()} value={number} />
      )}
    </ul>
  );
}

```

[-נסו זאת ב-CodePen](#)

זה תלוי בהחלטתך האם כדאי, JavaScript-לפערם התוצאה היא קוד ברור יותר, אבל סגנון זה יכול להיות גם לרעה. כמו ביותר מדי מקרים, זה יכול להיות זמן טוב [לחלי `map\(\)`](#) לחלץ משתנה מען שיפור קרייאתו. זכור כי אם גוף פונקציית [קומפוננטה](#).

[Go to TOC](#)

Optimizing Performance

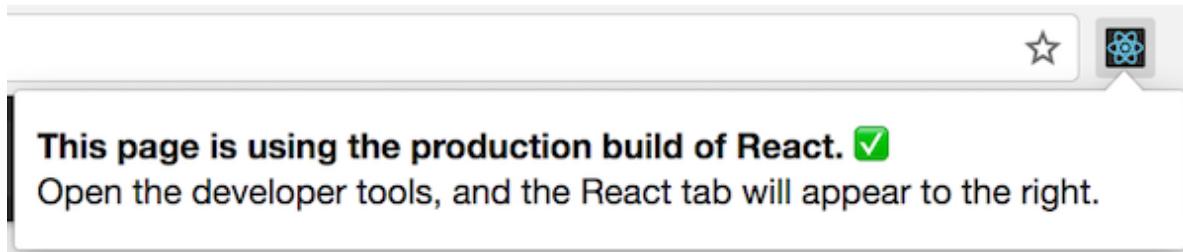
Internally, React uses several clever techniques to minimize the number of costly DOM operations required to update the UI. For many applications, using React will lead to a fast user interface without doing much work to specifically optimize for performance. Nevertheless, there are several ways you can speed up your React application.

Use the Production Build

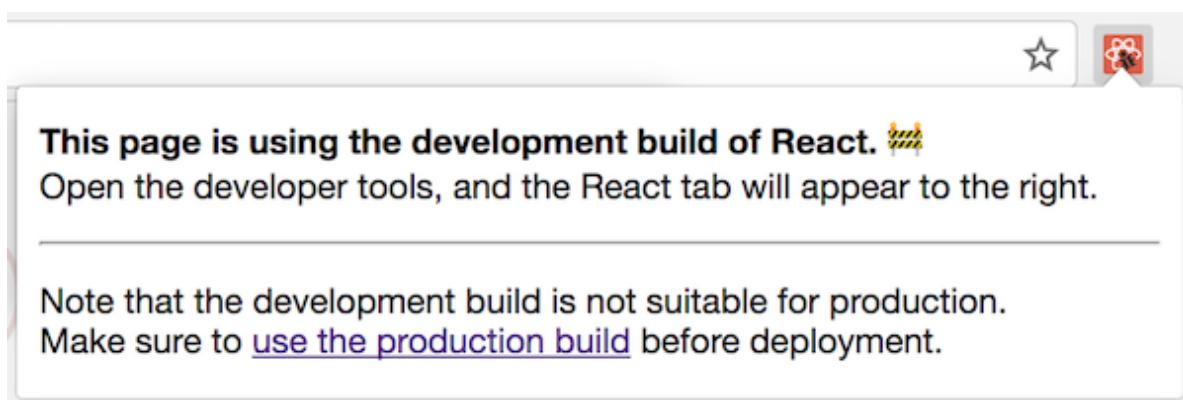
If you're benchmarking or experiencing performance problems in your React apps, make sure you're testing with the minified production build.

By default, React includes many helpful warnings. These warnings are very useful in development. However, they make React larger and slower so you should make sure to use the production version when you deploy the app.

If you aren't sure whether your build process is set up correctly, you can check it by installing [React Developer Tools for Chrome](#). If you visit a site with React in production mode, the icon will have a dark background:



If you visit a site with React in development mode, the icon will have a red background:



It is expected that you use the development mode when working on your app, and the production mode when deploying your app to the users.

You can find instructions for building your app for production below.

Create React App

If your project is built with [Create React App](#), run:

```
npm run build
```

This will create a production build of your app in the `build/` folder of your project.

Remember that this is only necessary before deploying to production. For normal development, use `npm start`.

Single-File Builds

We offer production-ready versions of React and React DOM as single files:

```
<script src="https://unpkg.com/react@18/umd/react.production.min.js"></script>
<script src="https://unpkg.com/react-dom@18/umd/react-dom.production.min.js">
</script>
```

Remember that only React files ending with `.production.min.js` are suitable for production.

Brunch

For the most efficient Brunch production build, install the `terser-brunch` plugin:

```
# If you use npm
npm install --save-dev terser-brunch

# If you use Yarn
yarn add --dev terser-brunch
```

Then, to create a production build, add the `-p` flag to the `build` command:

```
brunch build -p
```

Remember that you only need to do this for production builds. You shouldn't pass the `-p` flag or apply this plugin in development, because it will hide useful React warnings and make the builds much slower.

Browserify

For the most efficient Browserify production build, install a few plugins:

```
# If you use npm
npm install --save-dev envify terser uglifyify

# If you use Yarn
yarn add --dev envify terser uglifyify
```

To create a production build, make sure that you add these transforms (**the order matters**):

- The `envify` transform ensures the right build environment is set. Make it global (`-g`).

- The `uglifyify` transform removes development imports. Make it global too (`-g`).
- Finally, the resulting bundle is piped to `terser` for mangling ([read why](#)).

For example:

```
browserify ./index.js \
-g [ envify --NODE_ENV production ] \
-g uglifyify \
| terser --compress --mangle > ./bundle.js
```

Remember that you only need to do this for production builds. You shouldn't apply these plugins in development because they will hide useful React warnings, and make the builds much slower.

Rollup

For the most efficient Rollup production build, install a few plugins:

```
# If you use npm
npm install --save-dev rollup-plugin-commonjs rollup-plugin-replace rollup-plugin-terser

# If you use Yarn
yarn add --dev rollup-plugin-commonjs rollup-plugin-replace rollup-plugin-terser
```

To create a production build, make sure that you add these plugins (**the order matters**):

- The `replace` plugin ensures the right build environment is set.
- The `commonjs` plugin provides support for CommonJS in Rollup.
- The `terser` plugin compresses and mangles the final bundle.

```
plugins: [
  // ...
  require('rollup-plugin-replace')({
    'process.env.NODE_ENV': JSON.stringify('production')
  }),
  require('rollup-plugin-commonjs')(),
  require('rollup-plugin-terser')(),
  // ...
]
```

For a complete setup example [see this gist](#).

Remember that you only need to do this for production builds. You shouldn't apply the `terser` plugin or the `replace` plugin with `'production'` value in development because they will hide useful React warnings, and make the builds much slower.

webpack

Note:

If you're using Create React App, please follow [the instructions above](#).
This section is only relevant if you configure webpack directly.

Webpack v4+ will minify your code by default in production mode.

```
const TerserPlugin = require('terser-webpack-plugin');

module.exports = {
  mode: 'production',
  optimization: {
    minimizer: [new TerserPlugin({ /* additional options here */ })],
  },
};
```

You can learn more about this in [webpack documentation](#).

Remember that you only need to do this for production builds. You shouldn't apply `TerserPlugin` in development because it will hide useful React warnings, and make the builds much slower.

Profiling Components with the DevTools Profiler

`react-dom` 16.5+ and `react-native` 0.57+ provide enhanced profiling capabilities in DEV mode with the React DevTools Profiler. An overview of the Profiler can be found in the blog post "[Introducing the React Profiler](#)". A video walkthrough of the profiler is also [available on YouTube](#).

If you haven't yet installed the React DevTools, you can find them here:

- [Chrome Browser Extension](#)
- [Firefox Browser Extension](#)
- [Standalone Node Package](#)

Note

A production profiling bundle of `react-dom` is also available as `react-dom/profiling`. Read more about how to use this bundle at fb.me/react-profiling

Note

Before React 17, we use the standard [User Timing API](#) to profile components with the chrome performance tab. For a more detailed walkthrough, check out [this article by Ben Schwarz](#).

Virtualize Long Lists

If your application renders long lists of data (hundreds or thousands of rows), we recommend using a technique known as "windowing". This technique only renders a small subset of your rows at any given time, and can dramatically reduce the time it takes to re-render the components as well as the number of DOM nodes created.

[react-window](#) and [react-virtualized](#) are popular windowing libraries. They provide several reusable components for displaying lists, grids, and tabular data. You can also create your own windowing component, like [Twitter did](#), if you want something more tailored to your application's specific use case.

Avoid Reconciliation

React builds and maintains an internal representation of the rendered UI. It includes the React elements you return from your components. This representation lets React avoid creating DOM nodes and accessing existing ones beyond necessity, as that can be slower than operations on JavaScript objects. Sometimes it is referred to as a "virtual DOM", but it works the same way on React Native.

When a component's props or state change, React decides whether an actual DOM update is necessary by comparing the newly returned element with the previously rendered one. When they are not equal, React will update the DOM.

Even though React only updates the changed DOM nodes, re-rendering still takes some time. In many cases it's not a problem, but if the slowdown is noticeable, you can speed all of this up by overriding the lifecycle function `shouldComponentUpdate`, which is triggered before the re-rendering process starts. The default implementation of this function returns `true`, leaving React to perform the update:

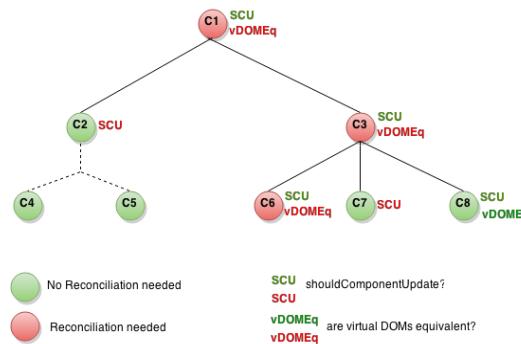
```
shouldComponentUpdate(nextProps, nextState) {
  return true;
}
```

If you know that in some situations your component doesn't need to update, you can return `false` from `shouldComponentUpdate` instead, to skip the whole rendering process, including calling `render()` on this component and below.

In most cases, instead of writing `shouldComponentUpdate()` by hand, you can inherit from `React.PureComponent`. It is equivalent to implementing `shouldComponentUpdate()` with a shallow comparison of current and previous props and state.

shouldComponentUpdate In Action

Here's a subtree of components. For each one, `SCU` indicates what `shouldComponentUpdate` returned, and `vDOMEq` indicates whether the rendered React elements were equivalent. Finally, the circle's color indicates whether the component had to be reconciled or not.



Since `shouldComponentUpdate` returned `false` for the subtree rooted at C2, React did not attempt to render C2, and thus didn't even have to invoke `shouldComponentUpdate` on C4 and C5.

For C1 and C3, `shouldComponentUpdate` returned `true`, so React had to go down to the leaves and check them. For C6 `shouldComponentUpdate` returned `true`, and since the rendered elements weren't equivalent React had to update the DOM.

The last interesting case is C8. React had to render this component, but since the React elements it returned were equal to the previously rendered ones, it didn't have to update the DOM.

Note that React only had to do DOM mutations for C6, which was inevitable. For C8, it bailed out by comparing the rendered React elements, and for C2's subtree and C7, it didn't even have to compare the elements as we bailed out on `shouldComponentUpdate`, and `render` was not called.

Examples

If the only way your component ever changes is when the `props.color` or the `state.count` variable changes, you could have `shouldComponentUpdate` check that:

```
class CounterButton extends React.Component {
  constructor(props) {
    super(props);
    this.state = {count: 1};
  }

  shouldComponentUpdate(nextProps, nextState) {
    if (this.props.color !== nextProps.color) {
      return true;
    }
    if (this.state.count !== nextState.count) {
      return true;
    }
    return false;
  }

  render() {
    return (
      <button
        color={this.props.color}
        onClick={() => this.setState(state => ({count: state.count + 1}))}>
        Count: {this.state.count}
      </button>
    );
  }
}
```

```

    );
}
}
```

In this code, `shouldComponentUpdate` is just checking if there is any change in `props.color` or `state.count`. If those values don't change, the component doesn't update. If your component got more complex, you could use a similar pattern of doing a "shallow comparison" between all the fields of `props` and `state` to determine if the component should update. This pattern is common enough that React provides a helper to use this logic - just inherit from `React.PureComponent`. So this code is a simpler way to achieve the same thing:

```

class CounterButton extends React.PureComponent {
  constructor(props) {
    super(props);
    this.state = {count: 1};
  }

  render() {
    return (
      <button
        color={this.props.color}
        onClick={() => this.setState(state => ({count: state.count + 1}))}>
        Count: {this.state.count}
      </button>
    );
  }
}
```

Most of the time, you can use `React.PureComponent` instead of writing your own `shouldComponentUpdate`. It only does a shallow comparison, so you can't use it if the props or state may have been mutated in a way that a shallow comparison would miss.

This can be a problem with more complex data structures. For example, let's say you want a `ListOfWords` component to render a comma-separated list of words, with a parent `WordAdder` component that lets you click a button to add a word to the list. This code does *not* work correctly:

```

class ListOfWords extends React.PureComponent {
  render() {
    return <div>{this.props.words.join(',')}</div>;
  }
}

class WordAdder extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      words: ['marklar']
    };
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    // This section is bad style and causes a bug
    const words = this.state.words;
    words.push('marklar');
    this.setState({words: words});
  }
}
```

```

    }

    render() {
      return (
        <div>
          <button onClick={this.handleClick} />
          <ListOfWords words={this.state.words} />
        </div>
      );
    }
}

```

The problem is that `PureComponent` will do a simple comparison between the old and new values of `this.props.words`. Since this code mutates the `words` array in the `handleClick` method of `WordAdder`, the old and new values of `this.props.words` will compare as equal, even though the actual words in the array have changed. The `ListofWords` will thus not update even though it has new words that should be rendered.

The Power Of Not Mutating Data

The simplest way to avoid this problem is to avoid mutating values that you are using as props or state. For example, the `handleClick` method above could be rewritten using `concat` as:

```

handleClick() {
  this.setState(state => ({
    words: state.words.concat(['marklar'])
  }));
}

```

ES6 supports a [spread syntax](#) for arrays which can make this easier. If you're using Create React App, this syntax is available by default.

```

handleClick() {
  this.setState(state => ({
    words: [...state.words, 'marklar'],
  }));
}

```

You can also rewrite code that mutates objects to avoid mutation, in a similar way. For example, let's say we have an object named `colormap` and we want to write a function that changes `colormap.right` to be `'blue'`. We could write:

```

function updateColorMap(colormap) {
  colormap.right = 'blue';
}

```

To write this without mutating the original object, we can use `Object.assign` method:

```

function updateColorMap(colormap) {
  return Object.assign({}, colormap, {right: 'blue'});
}

```

`updateColorMap` now returns a new object, rather than mutating the old one. `Object.assign` is in ES6 and requires a polyfill.

[Object spread syntax](#) makes it easier to update objects without mutation as well:

```
function updateColorMap(colormap) {  
  return {...colormap, right: 'blue'};  
}
```

This feature was added to JavaScript in ES2018.

If you're using Create React App, both `Object.assign` and the object spread syntax are available by default.

When you deal with deeply nested objects, updating them in an immutable way can feel convoluted. If you run into this problem, check out [Immer](#) or [immutability-helper](#). These libraries let you write highly readable code without losing the benefits of immutability.

Portals

Portals provide a first-class way to render children into a DOM node that exists outside the DOM hierarchy of the parent component.

```
ReactDOM.createPortal(child, container)
```

The first argument (`child`) is any [renderable React child](#), such as an element, string, or fragment. The second argument (`container`) is a DOM element.

Usage

Normally, when you return an element from a component's render method, it's mounted into the DOM as a child of the nearest parent node:

```
render() {
  // React mounts a new div and renders the children into it
  return (
    <div>
      {this.props.children}
    </div>
  );
}
```

However, sometimes it's useful to insert a child into a different location in the DOM:

```
render() {
  // React does *not* create a new div. It renders the children into `domNode`.
  // `domNode` is any valid DOM node, regardless of its location in the DOM.
  return ReactDOM.createPortal(
    this.props.children,
    domNode
  );
}
```

A typical use case for portals is when a parent component has an `overflow: hidden` or `z-index` style, but you need the child to visually "break out" of its container. For example, dialogs, hovercards, and tooltips.

Note:

When working with portals, remember that [managing keyboard focus](#) becomes very important.

For modal dialogs, ensure that everyone can interact with them by following the [WAI-ARIA Modal Authoring Practices](#).

[Try it on CodePen](#)

Event Bubbling Through Portals

Even though a portal can be anywhere in the DOM tree, it behaves like a normal React child in every other way. Features like context work exactly the same regardless of whether the child is a portal, as the portal still exists in the *React tree* regardless of position in the *DOM tree*.

This includes event bubbling. An event fired from inside a portal will propagate to ancestors in the containing *React tree*, even if those elements are not ancestors in the *DOM tree*. Assuming the following HTML structure:

```
<html>
  <body>
    <div id="app-root"></div>
    <div id="modal-root"></div>
  </body>
</html>
```

A `Parent` component in `#app-root` would be able to catch an uncaught, bubbling event from the sibling node `#modal-root`.

```
// These two containers are siblings in the DOM
const appRoot = document.getElementById('app-root');
const modalRoot = document.getElementById('modal-root');

class Modal extends React.Component {
  constructor(props) {
    super(props);
    this.el = document.createElement('div');
  }

  componentDidMount() {
    // The portal element is inserted in the DOM tree after
    // the Modal's children are mounted, meaning that children
    // will be mounted on a detached DOM node. If a child
    // component requires to be attached to the DOM tree
    // immediately when mounted, for example to measure a
    // DOM node, or uses 'autoFocus' in a descendant, add
    // state to Modal and only render the children when Modal
    // is inserted in the DOM tree.
    modalRoot.appendChild(this.el);
  }

  componentWillUnmount() {
    modalRoot.removeChild(this.el);
  }

  render() {
    return ReactDOM.createPortal(
      this.props.children,
      this.el
    );
  }
}

class Parent extends React.Component {
  constructor(props) {
    super(props);
```

```

this.state = {clicks: 0};
this.handleClick = this.handleClick.bind(this);
}

handleClick() {
  // This will fire when the button in Child is clicked,
  // updating Parent's state, even though button
  // is not direct descendant in the DOM.
  this.setState(state => ({
    clicks: state.clicks + 1
  }));
}

render() {
  return (
    <div onClick={this.handleClick}>
      <p>Number of clicks: {this.state.clicks}</p>
      <p>
        Open up the browser DevTools
        to observe that the button
        is not a child of the div
        with the onClick handler.
      </p>
      <Modal>
        <Child />
      </Modal>
    </div>
  );
}

function Child() {
  // The click event on this button will bubble up to parent,
  // because there is no 'onClick' attribute defined
  return (
    <div className="modal">
      <button>Click</button>
    </div>
  );
}

const root = ReactDOM.createRoot(appRoot);
root.render(<Parent />);

```

Try it on CodePen

Catching an event bubbling up from a portal in a parent component allows the development of more flexible abstractions that are not inherently reliant on portals. For example, if you render a `<Modal />` component, the parent can capture its events regardless of whether it's implemented using portals.

[Go to TOC](#)

React Without ES6

Normally you would define a React component as a plain JavaScript class:

```
class Greeting extends React.Component {
  render() {
    return <h1>Hello, {this.props.name}</h1>;
  }
}
```

If you don't use ES6 yet, you may use the `create-react-class` module instead:

```
var createReactClass = require('create-react-class');
var Greeting = createReactClass({
  render: function() {
    return <h1>Hello, {this.props.name}</h1>;
  }
});
```

The API of ES6 classes is similar to `createReactClass()` with a few exceptions.

Declaring Default Props

With functions and ES6 classes `defaultProps` is defined as a property on the component itself:

```
class Greeting extends React.Component {
  // ...
}

Greeting.defaultProps = {
  name: 'Mary'
};
```

With `createReactClass()`, you need to define `getDefaultsProps()` as a function on the passed object:

```
var Greeting = createReactClass({
  getDefaultsProps: function() {
    return {
      name: 'Mary'
    };
  },
  // ...
});
```

Setting the Initial State

In ES6 classes, you can define the initial state by assigning `this.state` in the constructor:

```
class Counter extends React.Component {
  constructor(props) {
    super(props);
    this.state = {count: props.initialCount};
  }
  // ...
}
```

With `createReactClass()`, you have to provide a separate `getInitialState` method that returns the initial state:

```
var Counter = createReactClass({
  getInitialState: function() {
    return {count: this.props.initialCount};
  },
  // ...
});
```

Autobinding

In React components declared as ES6 classes, methods follow the same semantics as regular ES6 classes. This means that they don't automatically bind `this` to the instance. You'll have to explicitly use `.bind(this)` in the constructor:

```
class SayHello extends React.Component {
  constructor(props) {
    super(props);
    this.state = {message: 'Hello!'};
    // This line is important!
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    alert(this.state.message);
  }

  render() {
    // Because `this.handleClick` is bound, we can use it as an event handler.
    return (
      <button onClick={this.handleClick}>
        Say hello
      </button>
    );
  }
}
```

With `createReactClass()`, this is not necessary because it binds all methods:

```
var SayHello = createReactClass({
  getInitialState: function() {
    return {message: 'Hello!'};
  },

  handleClick: function() {
    alert(this.state.message);
  },
});
```

```

render: function() {
  return (
    <button onClick={this.handleClick}>
      Say hello
    </button>
  );
}
);

```

This means writing ES6 classes comes with a little more boilerplate code for event handlers, but the upside is slightly better performance in large applications.

If the boilerplate code is too unattractive to you, you may enable the [experimental Class Properties](#) syntax proposal with Babel:

```

class SayHello extends React.Component {
  constructor(props) {
    super(props);
    this.state = {message: 'Hello!'};
  }
  // WARNING: this syntax is experimental!
  // Using an arrow here binds the method:
  handleClick = () => {
    alert(this.state.message);
  }

  render() {
    return (
      <button onClick={this.handleClick}>
        Say hello
      </button>
    );
  }
}

```

Please note that the syntax above is **experimental** and the syntax may change, or the proposal might not make it into the language.

If you'd rather play it safe, you have a few options:

- Bind methods in the constructor.
- Use arrow functions, e.g. `onClick={(e) => this.handleClick(e)}`.
- Keep using `createReactClass`.

Mixins

Note:

ES6 launched without any mixin support. Therefore, there is no support for mixins when you use React with ES6 classes.

We also found numerous issues in codebases using mixins, and don't recommend using them in the new code.

This section exists only for the reference.

Sometimes very different components may share some common functionality. These are sometimes called [cross-cutting concerns](#). `createReactClass` lets you use a legacy `mixins` system for that.

One common use case is a component wanting to update itself on a time interval. It's easy to use `setInterval()`, but it's important to cancel your interval when you don't need it anymore to save memory. React provides [lifecycle methods](#) that let you know when a component is about to be created or destroyed. Let's create a simple mixin that uses these methods to provide an easy `setInterval()` function that will automatically get cleaned up when your component is destroyed.

```
var SetIntervalMixin = {
  componentWillMount: function() {
    this.intervals = [];
  },
  setInterval: function() {
    this.intervals.push(setInterval.apply(null, arguments));
  },
  componentWillUnmount: function() {
    this.intervals.forEach(clearInterval);
  }
};

var createReactClass = require('create-react-class');

var TickTock = createReactClass({
  mixins: [SetIntervalMixin], // Use the mixin
  getInitialState: function() {
    return {seconds: 0};
  },
  componentDidMount: function() {
    this.setInterval(this.tick, 1000); // Call a method on the mixin
  },
  tick: function() {
    this.setState({seconds: this.state.seconds + 1});
  },
  render: function() {
    return (
      <p>
        React has been running for {this.state.seconds} seconds.
      </p>
    );
  }
});

const root = ReactDOM.createRoot(document.getElementById('example'));
root.render(<TickTock />);
```

If a component is using multiple mixins and several mixins define the same lifecycle method (i.e. several mixins want to do some cleanup when the component is destroyed), all of the lifecycle methods are guaranteed to be called. Methods defined on mixins run in the order mixins were listed, followed by a method call on the component.

[Go to TOC](#)

React Without JSX

JSX is not a requirement for using React. Using React without JSX is especially convenient when you don't want to set up compilation in your build environment.

Each JSX element is just syntactic sugar for calling `React.createElement(component, props, ...children)`. So, anything you can do with JSX can also be done with just plain JavaScript.

For example, this code written with JSX:

```
class Hello extends React.Component {
  render() {
    return <div>Hello ${this.props.toWhat}</div>;
  }
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Hello toWhat="World" />);
```

can be compiled to this code that does not use JSX:

```
class Hello extends React.Component {
  render() {
    return React.createElement('div', null, `Hello ${this.props.toWhat}`);
  }
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(React.createElement(Hello, {toWhat: 'World'}, null));
```

If you're curious to see more examples of how JSX is converted to JavaScript, you can try out [the online Babel compiler](#).

The component can either be provided as a string, as a subclass of `React.Component`, or a plain function.

If you get tired of typing `React.createElement` so much, one common pattern is to assign a shorthand:

```
const e = React.createElement;

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(e('div', null, 'Hello World'));
```

If you use this shorthand form for `React.createElement`, it can be almost as convenient to use React without JSX.

Alternatively, you can refer to community projects such as `react-hyperscript` and `hyperscript-helpers` which offer a terser syntax.

[Go to TOC](#)

Reconciliation

React provides a declarative API so that you don't have to worry about exactly what changes on every update. This makes writing applications a lot easier, but it might not be obvious how this is implemented within React. This article explains the choices we made in React's "diffing" algorithm so that component updates are predictable while being fast enough for high-performance apps.

Motivation

When you use React, at a single point in time you can think of the `render()` function as creating a tree of React elements. On the next state or props update, that `render()` function will return a different tree of React elements. React then needs to figure out how to efficiently update the UI to match the most recent tree.

There are some generic solutions to this algorithmic problem of generating the minimum number of operations to transform one tree into another. However, the [state of the art algorithms](#) have a complexity in the order of $O(n^3)$ where n is the number of elements in the tree.

If we used this in React, displaying 1000 elements would require in the order of one billion comparisons. This is far too expensive. Instead, React implements a heuristic $O(n)$ algorithm based on two assumptions:

1. Two elements of different types will produce different trees.
2. The developer can hint at which child elements may be stable across different renders with a `key` prop.

In practice, these assumptions are valid for almost all practical use cases.

The Diffing Algorithm

When diffing two trees, React first compares the two root elements. The behavior is different depending on the types of the root elements.

Elements Of Different Types

Whenever the root elements have different types, React will tear down the old tree and build the new tree from scratch. Going from `<a>` to ``, or from `<Article>` to `<Comment>`, or from `<Button>` to `<div>` - any of those will lead to a full rebuild.

When tearing down a tree, old DOM nodes are destroyed. Component instances receive `componentWillUnmount()`. When building up a new tree, new DOM nodes are inserted into the DOM. Component instances receive `UNSAFE_componentWillMount()` and then `componentDidMount()`. Any state associated with the old tree is lost.

Any components below the root will also get unmounted and have their state destroyed. For example, when diffing:

```
<div>
  <Counter />
</div>

<span>
  <Counter />
</span>
```

This will destroy the old `Counter` and remount a new one.

Note:

These methods are considered legacy and you should [avoid them](#) in new code:

- `UNSAFE_componentWillMount()`

DOM Elements Of The Same Type

When comparing two React DOM elements of the same type, React looks at the attributes of both, keeps the same underlying DOM node, and only updates the changed attributes. For example:

```
<div className="before" title="stuff" />
<div className="after" title="stuff" />
```

By comparing these two elements, React knows to only modify the `className` on the underlying DOM node.

When updating `style`, React also knows to update only the properties that changed. For example:

```
<div style={{color: 'red', fontWeight: 'bold'}} />
<div style={{color: 'green', fontWeight: 'bold'}} />
```

When converting between these two elements, React knows to only modify the `color` style, not the `fontWeight`.

After handling the DOM node, React then recurses on the children.

Component Elements Of The Same Type

When a component updates, the instance stays the same, so that state is maintained across renders. React updates the props of the underlying component instance to match the new element, and calls `UNSAFE_componentWillReceiveProps()`, `UNSAFE_componentWillUpdate()` and `componentDidUpdate()` on the underlying instance.

Next, the `render()` method is called and the diff algorithm recurses on the previous result and the new result.

Note:

These methods are considered legacy and you should [avoid them](#) in new code:

- `UNSAFE_componentWillUpdate()`
- `UNSAFE_componentWillReceiveProps()`

Recurse On Children

By default, when recursing on the children of a DOM node, React just iterates over both lists of children at the same time and generates a mutation whenever there's a difference.

For example, when adding an element at the end of the children, converting between these two trees works well:

```
<ul>
  <li>first</li>
  <li>second</li>
</ul>

<ul>
  <li>first</li>
  <li>second</li>
  <li>third</li>
</ul>
```

React will match the two `first` trees, match the two `second` trees, and then insert the `third` tree.

If you implement it naively, inserting an element at the beginning has worse performance. For example, converting between these two trees works poorly:

```
<ul>
  <li>Duke</li>
  <li>Villanova</li>
</ul>

<ul>
  <li>Connecticut</li>
  <li>Duke</li>
  <li>Villanova</li>
</ul>
```

React will mutate every child instead of realizing it can keep the `Duke` and `Villanova` subtrees intact. This inefficiency can be a problem.

Keys

In order to solve this issue, React supports a `key` attribute. When children have keys, React uses the key to match children in the original tree with children in the subsequent tree. For example, adding a `key` to our inefficient example above can make the tree conversion efficient:

```
<ul>
  <li key="2015">Duke</li>
  <li key="2016">Villanova</li>
```

```
</ul>
<ul>
  <li key="2014">Connecticut</li>
  <li key="2015">Duke</li>
  <li key="2016">Villanova</li>
</ul>
```

Now React knows that the element with key `'2014'` is the new one, and the elements with the keys `'2015'` and `'2016'` have just moved.

In practice, finding a key is usually not hard. The element you are going to display may already have a unique ID, so the key can just come from your data:

```
<li key={item.id}>{item.name}</li>
```

When that's not the case, you can add a new ID property to your model or hash some parts of the content to generate a key. The key only has to be unique among its siblings, not globally unique.

As a last resort, you can pass an item's index in the array as a key. This can work well if the items are never reordered, but reorders will be slow.

Reorders can also cause issues with component state when indexes are used as keys. Component instances are updated and reused based on their key. If the key is an index, moving an item changes it. As a result, component state for things like uncontrolled inputs can get mixed up and updated in unexpected ways.

[Here is an example of the issues that can be caused by using indexes as keys](#) on CodePen, and [here is an updated version of the same example showing how not using indexes as keys will fix these reordering, sorting, and prepending issues](#).

Tradeoffs

It is important to remember that the reconciliation algorithm is an implementation detail. React could rerender the whole app on every action; the end result would be the same. Just to be clear, rerender in this context means calling `render` for all components, it doesn't mean React will unmount and remount them. It will only apply the differences following the rules stated in the previous sections.

We are regularly refining the heuristics in order to make common use cases faster. In the current implementation, you can express the fact that a subtree has been moved amongst its siblings, but you cannot tell that it has moved somewhere else. The algorithm will rerender that full subtree.

Because React relies on heuristics, if the assumptions behind them are not met, performance will suffer.

1. The algorithm will not try to match subtrees of different component types. If you see yourself alternating between two component types with very similar output, you may want to make it the same type. In practice, we haven't found this to be an issue.
2. Keys should be stable, predictable, and unique. Unstable keys (like those produced by `Math.random()`) will cause many component instances and DOM nodes to be unnecessarily recreated, which can cause performance degradation and lost state in child components.

react

[Go to TOC](#)

DOM Elements

לא-תלות בדף עבור ביצועים ותאיום לדפינים שונים. ניצלו את ההזמנות על מנת לנוקט DOM מיישמת מערכת React.

צריכים להיות (כולל מנהלי אירועים) DOM-של ה (properties) והתכנות (attributes) כל המאפיינים, ב-React. המקרים היוצאים מן הכלל-tabIndex ב-HTML של tabIndex-לדוגמה, תוכנת ה `aria-label` ב-React. נוכל להشير את `aria-*` ב-`aria-label` `aria-label`.

הבדלים בתכונות

HTML ישנו מספר תכונות שעובדות בצורה שונה בין React:

checked

אתה יכול להשתמש בה על מנת לקבע. `checked` `checkbox` או `radio` `input` מסוג התוכינה נתמכת בקומפוננטות `checked` היא המקבילה הלא נשלטת, `defaultChecked`. דבר זה שימושי לבניית קומפוננטות נשלטות. בפעם הראשונה `mounted` שקובעת האם הקומפוננטה מסומנת בזמן שהוא.

className

הרגילים כמו SVG-וה DOM-הענין נוגע לכל אלמנטי ה `className`-נשתמש בתכונת ה CSS כדי להוסיף מחלוקת `<div>`, `<a>`. ואחרים.

במקרה `class`-השתמש בתכונת ה, (דבר שאינו נפוץ כל כך) עם אתה משתמש ב Web Components.

dangerouslySetInnerHTML

של הדף. באופן כללי, קביעת DOM-ב `innerHTML`-לשימוש ב React היא החלטה של XSS מהקוד היא מסוכנת מכיוון שהוא קל בעוטה לחשוף את המשתמשים שלו לתקפת HTML על `__html`, ולהעביר אובייקט עם מפתח `dangerouslySetInnerHTML` אבל אתה צריך לכתוב מ-React, מונת להזכיר לעצמך שהוא מסוכן. לדוגמה:

```
function createMarkup() {
  return {__html: 'First &middot; Second'};
}

function MyComponent() {
  return <div dangerouslySetInnerHTML={createMarkup()} />;
}
```

htmlFor

במקרה `htmlFor`-משתמשים ב React אלמנטי JavaScript-היא מילה שמורה ב `for`-אחריו.

onChange

מתנהג כמו שהיית מצפה: בכל פעם ששדה טופס משתנה, אירוע זה נורה. אנחנו בכוונה לא משתמשים `onChange` - אירוע ה-React-לא מתנהג כמו שהוא אמרו להתנהג, ו-`onchange`-בהתנהגות המובנית בדפדפן בגלל שבקלטי משתמש באזמון אמרת.

selected

שלו במקום. ראה `<option>` של `<select>` אם אתה רוצה לסייע `<value>` - כמו שמו, התיחס לערך של האופציה ב `<option>` לקבלת מידע מפורט ["The select Tag"](#)

style

:שימוש לב:

בתוך האפשרות העיקרית `style` מטעמי נוחות, אבל **שימוש בתוכונה style** - כמו דוגמאות בתיעוד משתמשות בצריכה להיות בשימוש כדי להתייחס `className`, **לייצוב קומפוננטות איננה מומלצת באופן כללי**. ברוב המקרים המטרה היא `style` CSS שהוגדרו בקבץ CSS למחוקות [FAQ: לייצוב CSS](#).

התנהגות התוכונה עקבית CSS. במקום `camelCased` מתקבל אובייקט `style`-תוכונה עם מאפייני JavaScript: `color`, `backgroundImage`, `fontSize` ועוד. אך היא יותר ישרה, ומונעת חורי אבטחת, ומונעת XSS. לדוגמה:

```
const divStyle = {
  color: 'blue',
  backgroundImage: 'url(' + imgUrl + ')',
};

function HelloWorldComponent() {
  return <div style={divStyle}>Hello World!</div>;
}
```

מקבילים `style` על מנת לתמוך בדפדפניים ישרים, אתה צריך לספק `autoprefixed`:

```
const divStyle = {
  WebkitTransition: 'all', // שים לב לאות W
  msTransition: 'all' // 'ms'-prefix הוגדרה
};

function ComponentWithTransition() {
  return <div style={divStyle}></div>;
}
```

הגדולה 'W' שים לב לאות // היחיד באותיות קטנות `ms`-prefix-הו `ms`。

לדוגמה DOM-ה-JavaScript עם מאפיינים `camelCased` מפתחות לייצוב הם `node.style.backgroundImage`. זו הסיבה ש-`ms`-**קיימות של ספקים שונים מ** `backgroundImage`. מתחילה עם גודלה `WebkitTransition`.

"אך"-כמו מאפייני סגנון. אם אתה רוצה להשתמש ביחידות מידת אחוריות חזק מ "אך" תוסיף באופן אוטומטי סימנת `px`:

```
// 10px תוצאה:
<div style={{ height: 10 }}>
  שLOWM גLOWM!
</div>

// 10% תוצאה:
<div style={{ height: '10%' }}>
  שLOWM גLOWM!
</div>
```

בגלו שזה לא יעבוד. תכונה זו עוצרת את `contentEditable`-בדרך כלל, ישנה אזהרה כאשר אלמנט עם ילדים מסומן כ`contentEditable` שמנהלה [Draft.js](#) האזהרה. אל תשמש בזה אלא אם אתה בונה ספרייה כמו [cano](#).

suppressContentEditableWarning

בדרך כלל יש אזהרה כשהשות והלקות מレンדרים תוכן שונה. לעומת זאת, אם אתה משתמש ברינדור בצד-שרת במקומות נדרים, זה קשה מאד עד בלתי אפשרי להבטיח תוצאה זהה. לדוגמה, חוותות זמנים אמורות להיות שונות ובלתי זהה.

suppressHydrationWarning

בדרך כלל יש אזהרה כשהשות והלקות מレンדרים תוכן שונה. לעומת זאת, אם אתה משתמש ברינדור בצד-שרת במקומות נדרים, זה קשה מאד עד בלתי אפשרי להבטיח תוצאה זהה. לדוגמה, חוותות זמנים אמורות להיות שונות ובלתי זהה.

לא תזהיר אותך לגבי אי התאמה `suppressHydrationWarning` אם אתה קובע את `suppressHydrationWarning` true, React יתזען ויתהיר לך על תוצאות ובתוכן של האלמנט. דבר זה עובד בעומק של רמה אחת בלבד, ומיעוט לשימוש כפתח מילוט. אל תשמש בזה יותר מאשר ובתוכן של האלמנט. אם אתה ממליץ על `ReactDOM.hydrate()` [documentation](#). ===== If you set `suppressHydrationWarning` to `true`, React will not warn you about mismatches in the attributes and the content of that element. It only works one level deep, and is intended to be used as an escape hatch. Don't overuse it. You can read more about hydration in the `ReactDOM.hydrateRoot()` [documentation](#).

84ad3308338e2bb819f4f24fa8e9dfeeffaa970b

value

ניתן להשתמש בה כדי לקבע את `value`. `<input>`, `<select>` ו-`<textarea>` נטמכות על ידי הקומפוננטות `-value`-תוכנות ה-`value` הוא המקביל הבלטי של `value`, שקובע את ערך `defaultValue`. ערך הקומפוננטה. זה שימושי לבניית קומפוננטות שליטה ראשונה `value` כשהיא `mounted`.

הנתמכות HTML-כל תכונות ה-

סטנדרטיב או [מותאמת אישית](#), נטמכת באופן מלא DOM כל תכונת React 16-החל מ-

props בדרך כלל מקבלות גם מכיוון שקומפוננטות JavaScript-ל-DOM. תמיד סיפקה משקל משתמש ממוקד בדיק כמו משקי המשתמש של `camelCase` -משתמש במסכתה של `DOM`, React-מותאמות אישית וגם ככל שקשרוות לה-`DOM`:

```
<div tabIndex={-1} /> // node.tabIndex DOM API
<div className="Button" /> // node.className DOM API
<input readOnly={true} /> // node.readOnly DOM API
```

המקבילות, עם הוצאה מן הכלל של המקרים המיוחדים המתוארים HTML-HTML-הלו עובדים באופן דומה לתכונות ה-props-ה. לעיל.

כוללות נתמכות על ידי DOM-חלק מתכונות ה-React:

```
accept acceptCharset accessKey action allowFullScreen alt async autoComplete
autoFocus autoPlay capture cellPadding cellSpacing challenge charSet checked
cite classID className colSpan cols content contentEditable contextMenu controls
controlsList coords crossOrigin data dateTime default defer dir disabled
download draggable encType form formAction formEncType formMethod formNoValidate
formTarget frameBorder headers height hidden high href hrefLang htmlFor
httpEquiv icon id inputMode integrity is keyParams keyboardType kind label lang list
loop low manifest marginHeight marginWidth max maxLength media mediaGroup method
min minLength multiple muted name noValidate nonce open optimum pattern
placeholder poster preload profile radioGroup readOnly rel required reversed
role rowSpan rows sandbox scope scoped scrolling seamless selected shape size
sizes span spellCheck src srcDoc srcLang srcSet start step style summary
tabIndex target title type useMap value width wmode wrap
```

נתמכות באופן מלא SVG-באופן דומה, כל תכונות ה-

```
accentHeight accumulate additive alignmentBaseline allowReorder alphabetic
amplitude arabicForm ascent attributeName attributeType autoReverse azimuth
baseFrequency baseProfile baselineShift bbox begin bias by calcMode capHeight
clip clipPath clipPathUnits clipRule colorInterpolation
colorInterpolationFilters colorProfile colorRendering contentScriptType
contentStyleType cursor cx cy d decelerate descent diffuseConstant direction
display divisor dominantBaseline dur dx dy edgeMode elevation enableBackground
end exponent externalResourcesRequired fill fillOpacity fillRule filter
filterRes filterUnits floodColor floodOpacity focusable fontFamily fontSize
fontSizeAdjust fontStretch fontStyle fontVariant fontWeight format from fx fy
g1 g2 glyphName glyphOrientationHorizontal glyphOrientationVertical glyphRef
gradientTransform gradientUnits hanging horizAdvX horizOriginX ideographic
imageRendering in in2 intercept k k1 k2 k3 k4 kernelMatrix kernelUnitLength
kerning keyPoints keySplines keyTimes lengthAdjust letterSpacing lightingColor
limitingConeAngle local markerEnd markerHeight markerMid markerStart
markerUnits markerWidth mask maskContentUnits maskUnits mathematical mode
```

```
numOctaves offset opacity operator order orient orientation origin overflow
overlinePosition overlineThickness paintOrder panose1 pathLength
patternContentUnits patternTransform patternUnits pointerEvents points
pointsAtX pointsAtY pointsAtZ preserveAlpha preserveAspectRatio primitiveUnits
r radius refX refY renderingIntent repeatCount repeatDur requiredExtensions
requiredFeatures restart result rotate rx ry scale seed shapeRendering slope
spacing specularConstant specularExponent speed spreadMethod startOffset
stdDeviation stemh stemv stitchTiles stopColor stopOpacity
strikethroughPosition strikethroughThickness string stroke strokeDasharray
strokeDashoffset strokeLinecap strokeLinejoin strokeMiterlimit strokeOpacity
strokeWidth surfaceScale systemLanguage tableValues targetX targetY textAnchor
textDecoration textLength textRendering to transform u1 u2 underlinePosition
underlineThickness unicode unicodeBidi unicodeRange unitsPerEm vAlphabetic
vHanging vIdeographic vMathematical values vectorEffect version vertAdvY
vertOriginX vertOriginY viewBox viewTarget visibility widths wordSpacing
writingMode x x1 x2 xChannelSelector xHeight xlinkActuate xlinkArcrole
xlinkHref xlinkRole xlinkShow xlinkTitle xlinkType xmlns xmlnsXlink xmlBase
xmlLang xmlSpace y y1 y2 yChannelSelector z zoomAndPan
```

ניתן גם להשתמש בתכונות שモותאמות באופן אישי, כל עוד הן באוטיות קטנות.

SyntheticEvent

ראה את מדריך [טיפול](#) המהווה חלק מערכת האירועים של `SyntheticEvent` - מדריך עוז זה מסביר על מעטפת ה- [אירועים](#) על מנת למדוד עוד.

סקירה כללית

מעטפת תאימות בין דפדףנים מסביב לאירוע המקורי של `SyntheticEvent` מטפלי האירועים שלק מקבלים מופעים של כמעט זהה, שלה את אותו משך כמו האירוע המקורי, כולל `stopPropagation()` ו- `preventDefault()`, שהואEVENTים עובדים באופן זהה בין כל הדפדףנים.

על מנת לקבל `nativeEvent` אם אתה מגלת שאתה זוקק לאירוע הדפדף המקורי מסיבה כלשהי, פשוט השתמש במאפיין `onMouseLeave`, האירועים הסינטטיים שונים ולא מופיעים לאירועים של הדפדף. לדוגמה ב- `event.nativeEvent` יכול להשתנות בכל API/public הגרפי הוא לא חלק מה `mouseout`. `mouseout` על האירוע:

```
boolean bubbles
boolean cancelable
DOMEventTarget currentTarget
boolean defaultPrevented
number eventPhase
boolean isTrusted
DOMEvent nativeEvent
void preventDefault()
boolean isDefaultPrevented()
void stopPropagation()
boolean isPropagationStopped()
void persist()
DOMEventTarget target
number timeStamp
string type
```

הערה:

לא [מצורף יותר](#) `SyntheticEvent` - לא עושה כלום בגלל שה `e.persist()` - שימוש ב- 17 מגרסה.

הערה:

במקום זאת event, לא יפסיק התשובות `false`-מ- `event.preventDefault()` החזרת, נכון לגורשה. אמורים לפעול דנית, כראוי `e.preventDefault()` או `e.stopPropagation()`.

אירועים נתמכים

מנרמלת אירועים כך שיהיו להם מאפיינים עקובים בין דפדףנים שונים React.

`Capture` מטפלים האירועים שלහן מופעלים על ידי אירוע בשלב הבabbo. על מנת לרשום מטפל אירועים לשלב הলכידה, הוסף על מנת לטפל באירוע ההקלקה `onClickCapture`-עליך להשתמש ב-, לדוגמה, במקום שימוש ב- `onClick` בשלב הলכידה.

- [Clipboard](#) אירוע
 - [אירוע קומפוזיציה](#)
 - [אירוע מקלדת](#)
 - [אירוע פוקוס](#)
 - [אירוע טפסים](#)
 - [אירועים גנריים](#)
 - [אירועי עבר](#)
 - [אירועי מצביע](#)
 - [אירועי בחירה](#)
 - ['אירועי תאץ](#)
 - [אירועי משק משתמש](#)
 - [אירועי גלגל](#)
 - [אירועי מדיה](#)
 - [אירועי תמונה](#)
 - [אירועי אנטימציה](#)
 - [אירועי מעבר](#)
 - [אירועים אחרים](#)
-

סיכום

אירוע clipboard

:שמות אירועים:

`onCopy onCut onPaste`

:מאפיינים:

`DOMDataTransfer clipboardData`

אירוע קומפוזיציה

:שמות אירועים:

`onCompositionEnd onCompositionStart onCompositionUpdate`

:מאפיינים:

`string data`

איירועי מקלדת

שמות איירועים:

onKeyDown onKeyPress onKeyUp

מאפיינים:

```
boolean altKey
number charCode
boolean ctrlKey
boolean getModifierState(key)
string key
number keyCode
string locale
number location
boolean metaKey
boolean repeat
boolean shiftKey
number which
```

בשלב 3 DOM- יכול לקבל כל ערך שמתועד ב述职 אירועי `key` - מאפיין ה-

איירועי פוקוס

שמות איירועים:

onFocus onBlur

לא רק אלמנטי טפסים של DOM-איירועי הפוקוס הללו עובדים על כל האלמנטים ב-

מאפיינים:

DOMEventTarget relatedTarget

onFocus

כשהאלמנט (או אלמנט כלשהו בתוכו) מקבל פוקוס. לדוגמה, נקרה לו כשההמשתמש לוחץ על קלט טקסט - נקרה ל-

```
function Example() {
  return (
    <input
      onFocus={(e) => {
        console.log('Focused on input');
      }}
      placeholder="onFocus is triggered when you click this input."
    />
  )
}
```

onBlur

כשהפוקוס עזב את האלמנט (או אלמנט כלשהו בתוכו). לדוגמה, נקרה לו כשההמשתמש לוחץ על נקודה - נקרה ל- `onBlur` מוחץ לקלט הטקסט שבפוקוס.

```
function Example() {
  return (
    <input
      onBlur={(e) => {
        console.log('Triggered because this input lost focus');
      }}
      placeholder="onBlur is triggered when you click this input and then you
      click outside of it."
    />
  )
}
```

זיהוי כניסה ויציאה של פוקוס

לבדיל אם מקורם של אירועי הפוקוס או הטעטוש מוחוץ לאלמנט `relatedTarget`-ובו ניתן להשתמש בhhורה. הנה דמו שתוכל להעתיק ולהדביק שمراה איך לזהות פוקוס על אלמנט צאצא, פוקוס עצמו, פוקוס כשכנים נסימן או עוביים את כל התת-עץ.

```
function Example() {
  return (
    <div
      tabIndex={1}
      onFocus={(e) => {
        if (e.currentTarget === e.target) {
          console.log('focused self');
        } else {
          console.log('focused child', e.target);
        }
        if (!e.currentTarget.contains(e.relatedTarget)) {
          // לא מופעל כשמחלייפים פוקוס בין ילדים
          console.log('focus entered self');
        }
      }}
      onBlur={(e) => {
        if (e.currentTarget === e.target) {
          console.log('unfocused self');
        } else {
          console.log('unfocused child', e.target);
        }
        if (!e.currentTarget.contains(e.relatedTarget)) {
          // לא מופעל כשמחלייפים פוקוס בין ילדים
          console.log('focus left self');
        }
      }}
    >
      <input id="1" />
      <input id="2" />
    </div>
  );
}
```

AIRUII TFSIM

שמות אירועים:

`onChange` `onInput` `onInvalid` `onReset` `onSubmit`

ראה [טפסים](#), לקבלה מידע נוסף על אירוע ה-`onchange`.

אירועים גנריים

שמות אירוע:

`onError onLoad`

אירוע עכבר

שמות אירוע:

`onClick onContextMenu onDoubleClick onDrag onDragEnd onDragEnter onDragExit
onDragLeave onDragOver onDragStart onDrop onMouseDown onMouseEnter onMouseLeave
onMouseMove onMouseOut onMouseOver onMouseUp`

מופצים מהאלמנט שהעכבר עזב אל האלמנט שאליו העכבר נכנס במקומות `onMouseEnter` ו- `onMouseLeave`. אירועי ה-`onMouseEnter` ו-`onMouseLeave` מופצים מהאלמנט שהעכבר עזב אל האלמנט שאליו העכבר נכנס במקומות `onMouseEnter` ו- `onMouseLeave`.

מאפיינים:

```
boolean altKey
number button
number buttons
number clientX
number clientY
boolean ctrlKey
boolean getModifierState(key)
boolean metaKey
number pageX
number pageY
DOMEventTarget relatedTarget
number screenX
number screenY
boolean shiftKey
```

אירוע מצביע

שמות אירוע:

`onPointerDown onPointerMove onPointerUp onPointerCancel onGotPointerCapture
onLostPointerCapture onPointerEnter onPointerLeave onPointerOver onPointerOut`

מופצים מהאלמנט שהעכבר עזב אל האלמנט שאליו העכבר נכנס `onPointerEnter` ו- `onPointerLeave`. אירועי ה-`onPointerEnter` ו-`onPointerLeave` מופצים מהאלמנט שהעכבר עזב אל האלמנט שאליו העכבר נכנס במקומות `onPointerEnter` ו- `onPointerLeave`.

מאפיינים:

:אירוע מצביע מרחיבים `אירוע עכבר` עם המאפיינים הבאים, [spec W3](#)-כמו שהואדר ב-

```

number pointerId
number width
number height
number pressure
number tangentialPressure
number tiltX
number tiltY
number twist
string pointerType
boolean isPrimary

```

הערה לגבי תמייה בדפדים:

1- בזמן כתיבת מאמר זה, הדפדים הנתמכים הם) אירובי מצבייע עדין לא נתמכים בכל הדפדים Internet Explorer). תאימות סטנדרטית יגדיל polyfill לתמייה בדפדים אחרים כי polyfill לא ממשת React באופן מכון react-dom. משמעות את גודל הבandal של

אירובי מצבייע מצד שלישי polyfill אם האפליקציה שלך דורשת אירובי מצבייע, אנו ממליצים שתוסיפו.

אירובי בחירה

שמות אירועים:

```
onSelect
```

אירובי טאץ'

שמות אירועים:

```
onTouchCancel onTouchEnd onTouchMove onTouchStart
```

מאפיינים:

```

boolean altKey
DOMTouchList changedTouches
boolean ctrlKey
boolean getModifierState(key)
boolean metaKey
boolean shiftKey
DOMTouchList targetTouches
DOMTouchList touches

```

אירובי משתמש

שמות אירועים:

```
onScroll
```

הערה

לא מבבע בריקט. זה תואם את התנהוגות הדפדפן ומונע הבלבול כששת אלמנטים שהוא `onScroll`, החול מגרסה 17 scrollable events יורה ווחק events על הוראה ווחק events על הוראה.

:מאפיינים

`number detail
DOMAbstractView view`

אירוע גלגל

:שמות אירועים

`onWheel`

:מאפיינים

`number deltaMode
number deltaX
number deltaY
number deltaZ`

אירוע מדיה

:שמות אירועים

`onAbort onCanPlay onCanPlayThrough onDurationChange onEmptied onEncrypted
onEnded onError onLoadedData onLoadedMetadata onLoadStart onPause onPlay
onPlaying onProgress onRateChange onSeeked onSeeking onStalled onSuspend
onTimeUpdate onVolumeChange onWaiting`

אירוע תמונה

:שמות אירועים

`onLoad onError`

אירוע אנימציה

:שמות אירועים

`onAnimationStart onAnimationEnd onAnimationIteration`

:מאפיינים

```
string animationName  
string pseudoElement  
float elapsedTime
```

אירועי מעבר

שמות אירועים:

```
onTransitionEnd
```

מאפיינים:

```
string propertyName  
string pseudoElement  
float elapsedTime
```

אירועים אחרים

שמות אירועים:

```
onToggle
```

רשימה מונחי React

אפליקציית Single-page

הדרושים (HTML-CSS-ו-JavaScript כגון קבצי) אחד ואת כל הנכדים HTML היא אפליקציה שטוענת עםод single-page אפליקטיבת כדי שהאפליקציה תרוץ. כל אינטראקציה עם הדף או עם הדפים הבאים אינה דורשת נסעה הלהך ושוב לשרת, כלומר הדף אינו נטען מחדש.

יכולת לשמש גם לשיפור חלקיים React, זוהי אינה דרישת single-page ב-React, יכולת לארום שאטם יכולים לבנות אפליקטיבת שرونדר בשרת markup יכול להתקיים בשלום עם React-קיטנים של אתרים קיימים עם אינטראקטיביות נוספת. קוד שנכתב ב-React נמצא בשימוש בפייסבוק או עם ספריות אחרות בצד הלוקה. למעשה, זה בדיקת האופן בו, PHP על ידי משה כמו.

'וכו, ES2015, ES2016,

היא יישום JavaScript אשר שפת ECMAScript, וראשי תיבות אלה מתיחסים לגרסאות העדכניות ביותר של תקן מפרט השפה כוללת תוספות ורבות לגרסאות הקודמות כגון: פונקציות חצ, מחלקות, TBENIOT (ES2015 הידועה גם בשם) ES6 שלה. גרסת תוכלנו ללמוד עד על גרסאות ספציפיות [כאן]. 1- `let` `const` שגיאת דפוס, הצהרות

היא יישום JavaScript אשר שפת ECMAScript, וראשי תיבות אלה מתיחסים לגרסאות העדכניות ביותר של תקן מפרט השפה כוללת תוספות ורבות לגרסאות הקודמות כגון: פונקציות חצ, מחלקות, TBENIOT (ES2015 הידועה גם בשם) ES6 שלה. גרסת תוכלנו ללמוד עד על גרסאות ספציפיות [כאן](#). 1- `let` `const` שגיאת דפוס, הצהרות

מהדרים

בפורמט אחר. מקרה השימוש הנפוץ ביותר הוא JavaScript משנה אותו ומחזיר קוד JavaScript לוחק קוד מהדר הוא המהדר הנפוץ ביותר בשימוש עם [Babel](#). ולהפוך אותו לתחביר שדפננים שונים מסוגלים לפרש ES6 לקחת תחביר React.

מאגדים

שנכתב בתו מודולים נפרדים (לעתים קרובות מאות מהם), ומאגדים אותם יחד לתוך CSS-ו-JavaScript מאגדים לוקחים קוד כוללים יישומים React כמה קבצים בעלי אופטימיזציה טוביה יותר עבור הדפננים. כמה מאגדים בשימוש נפוץ באפליקציות [Webpack](#)-ו-[Browserify](#).

מנהל כבילות

המ שני מנהלי כבילות בשימוש נפוץ [Yarn](#)-ו-[npm](#). מנהלי כבילות הם כלים המאפשרים לך לנצל תלויות בפרויקט שלך שנקודותיהם לתוכהם עברו אותו מרשם כבילות React. npm

CDN

ים מספקים מעמן תוכן טעאי מתוק רשת של שירותי-CDN. (רשת אספקת תוכן) CDN מייצג Content Delivery Network ברחבי העולם.

JSX

JSX היא דומה לשפת תבנית, אבל יש לה את מלא העוצמה של JavaScript. היא תוספת תחביר עבור JSX כדי "אלמנטי הנקראים" אלמנטי JavaScript שמחזירות אובייקטי `React.createElement()` מתكمפלת לкриאות `כאן JSX צפוי בתיעוד כאן` ותוכלו למצוא הדרכה מעמיקה יותר על JSX-לקבל מבוא בסיסי ל-JavaScript.

לדוגמה HTML במקומם בשמות של תכונות camelCase-משתמשת בקונבנציית מתן שמות של מאפיינים ב DOM `tabindex`-מכיוון ש `tabIndex` גם נכתב בשם `class` המאפיין JSX-ב-הופך ל-`className`-ב-JavaScript:

<<<<< HEAD

```
const name = 'תפוזינה';
ReactDOM.render(
  <h1 className="hello">{name}!</h1>,
  document.getElementById('root')
);
```

=====

```
<h1 className="hello">My name is Clementine!</h1>
```

84ad3308338e2bb819f4f24fa8e9dfeeffaa970b

{#elements} אלמנטים

אפשר לבלב אלמנטים עם קונספט נרחב יותר של "קומפוננטות". React הם אבני הבניין של אפליקציות React אלמנטים של React (הם אינם ניתנים לשינוי React אלמנט מתאר את מה שאתה רוצה לראות על המסך. אלמנטים של immutable).

```
const element = <h1>שלום,</h1>;
```

בדרך כלל, אלמנטים אינם נמצאים בשימוש ישיר, אלא מוחזרים מקומפוננטות.

קומפוננטות {#components}

שিירונדר לדף. הגרסה הפשטota ביותר React הם חלקים קטנים, הניתנים לשימוש המחזירים אלמנט React קומפוננטות פשטota שמחזירה אלמנט JavaScript פונקציית של קומפוננטות React:

```
function Welcome(props) {
  return <h1>שלום, {props.name}</h1>;
}
```

קומפוננטות יכולות גם להיות מחולקות ES6:

```
class Welcome extends React.Component {
  render() {
    return <h1>שלום, {this.props.name}</h1>;
  }
}
```

ניתן לפצל קומפוננטות לחלקים שונים של פונקציונליות ולהשתמש בהן בתוך קומפוננטות אחרות. קומפוננטות יכולות להחזיר קומפוננטות אחרות, מערכיים, מחרוזות ומספרים. כל מצב טוב הוא שאם חלק המשמש שלכם נמצא בשימוש מספר פעמים (כפטור, פאנל, אוטר), או שהוא מורכב מספיק בפני עצמו (אפליקציה, פיד סיפורים, תוגבה), הוא מועמד טוב להיות ראה **תיעוד זה**. ולא `<Wrapper>` ו-

props {#props}

`props` הם נתונים המועברים מטה מקומפוננטה הוראה לקומפוננטה יلد. הם קלטים לkomponenotot.

הם לקרוא- בלבד. הם לא צריכים להשתנות בשום דרך שהוא `props` - זכרו ש:

```
// טעות!
props.number = 42;
```

במקום זאת `state` - אם עלייכם לשנות ערך כלשהו בתגובה לקרטת רשות, השתמשו ב-

props.children {#propschildren}

`props.children` זמין בכל קומפוננטה. הוא מכיל את התוכן בין תגי הפתיחה והסגירה של קומפוננטה. לדוגמה:

```
<Welcome>!שלום</Welcome>
```

המחרצת `Hello world!` - זמיןה ב `props.children` בkomponenotat `Welcome`:

```
function Welcome(props) {
  return <p>{props.children}</p>;
}
```

- עברו קומפוננטות המוגדרות כמחלקות, השתמשו ב- `this.props.children`:

```
class Welcome extends React.Component {
  render() {
    return <p>{this.props.children}</p>;
  }
}
```

state {#state}

עשוי `state` כאשר נתונים כלשהם המשיכים אליה משתנים עם הזמן. לדוגמה, קומפוננטה `Checkbox` קומפוננטה צוריכה `isChecked` לשווה לרצות לעקב אחרי `NewsFeed` שלה, וקומפוננטה `fetchedPosts`-ב-state להזדקק לשלה.

מנוהל על ידי `state` מועברים מקומפוננטת הורה, אבל `props`-הו ש-`state`-ל `props`-הבדל החשוב ביותר בין `state`-שלה, אבל היא כן יכולה לשנות את ה `props`-הקומפוננטה עצמה. קומפוננטה לא יכולה לשנות את ה `state`-שלה.

מעבר כל פישה מסויימת של נתונים משתנים, צריך להיות ורק קומפוננטה אחת שתיה "הבעלים" של אותו מידע ב-`state` של שני קומפוננטות שונות. במקום זאת, [הריםו אותו למעלה](#) אל האב הקדמון המשותף הקיים `state` שלה. אל תנשו לסנכרון לשתי הקומפוננטות `props` ביותר שלהם, והעברו אותו למיטה בתור.

מתודות מעגל חיים {#lifecycle-methods}

מתודות מעגל חיים הן פונקציונליות מותאמת אישית המורצת במהלך השלבים השונים של קומפוננטה. ישן מתודות זמניות כאשר הקומפוננטה מתעדכנת, וכאשר הקומפוננטה נהיית, (mounting), DOM-כasher הקומפוננטה נוצרת נוספת ונוספת ל-`unmounted`-או מוסרת מה-`DOM`.

קומפוננטות מבוקרות נגד בלתי-մבוקרות

יש שתי גישות שונות להתחזק עם קלטי טופס-React.

נקרא קומפוננטה מבוקרת. כאשר משתמש מזמן נתונים לקומפוננטה מבוקרת, React אלמנט קלט של טופס שערכו נשלט על ידי מופעל של אירוע שינוי והקוד שלכם מחייב אם הקלט וולידי (על-ידי רינדור מחדש המידע המעודכן). אם לא תרנדרו מחדש, אלמנט הטופס ישאר ללא שינוי.

כאשר משתמש מכניס נתונים לתוך שדה React-קומפוננטה בלתי מבוקרת עובדת כמו שאלמנטים של טופס עובדים מחוץ לטענה שום דבר. עם זאת, זה גם אומר React-טופס (תיבת קלט, טופס נפתח וכו') המידע המעדכן משתקף ללא צורך שشاءם לא יכולים להכירו את השדה שיכיל ערך מסוים.

ברוב המקרים יהיה عليיכם להשתמש בקומפוננטות מבוקרות.

מפתחות {#keys}

"לזהות אילו React-מפתח" הוא מאפיין מהראות מיוחדת שעלייכם לכלול בעת יצירה מערכיהם של אלמנטים. מפתחות עוזרים ל"פריטים השתנו, נוספו או הוסרו. המפתחות צריכים להיותם לאלמנטים בתוך מערך על מנת לתת לאלמנטים זהות קבועה

מפתחות צריכים להיות ייחודיים רק בין אלמנטים אחים באותו מערך. הם לא צריכים להיות ייחודיים על פני האפליקציה כולה או אפילו קומפוננטה אחת.

-למפתחות. חשוב של מפתחות תהיה "זהות קבועה" על פני רינדרים מחדש כך ש `Math.random()` אל תעביר משהו כמו תוכל לקבוע متى פריטים מתווספים, מוסרים, או מאורגנים מחדש. באופן אידיאלי, מפתחות צריכים להתאים למקומות ייחודיים וקבועים שמקורם בנתונים שלכם, כגון `post.id`.

Refs {#refs}

יכולת להיות אובייקט שנוצר על ידי `ref` תומכת במאפיין מיוחד שnitן לצרף לכל קומפוננטה. התוכונה היא פונקציית `ref` כאשר התוכונה (מגרסאות קודמות API-ב) או מחרוזת `callback`, או פונקציית `()` callback הבסיסי או את מופע המחלקה (בהתאם לסוג האלמנט) כารוגמןט שלה. זה DOM-הfonקציה מקבלת את אלמנט ה-או מופע הקומפוננטה DOM-אפשר למכ גישה ישירה לאלמנט ה-

לעתים קרובות כדי "לגרום לדברים לקרות" refs-באופן חסכני. אם אתם מוצאים את עצמכם משתמשים בrefs-השתמשו באפליקציה שלכם, שקלו לבצע היכרות טובה יותר עם [זרימת נתונים מלמעלה-מטה](#).

Events {#events}

יש כמה הבדלים תחביריים React לטיפול באירועים עם אלמנטי:

- במקום באותיות קטנות, נקראים באמצעות React camelCase מטפל אירועים של.
- אתם מעבירים פונקציה כמטפל האירוע, ולא מחרוזת JSX עם.

התאמת {#reconciliation}

בפועל על-ידי השוואת DOM מחליטה אם יש צורך בעדכו React, של קומפוננטה או משתנים state-props-כאשר התרחיך זה לקרוא DOM-React. יעדכו את ה-האלמנט החדש שהוחזר עם האלמנט הקודם שורונדר. כאשר הם לא זהים ההתקאה".

[Go to TOC](#)

דרישות סביבת JavaScript

אם אתה תומך בדפדפניים ישנים יותר ומכשירים אשר עדין HEAD React 16 `<<<<< Map` תלוי באוסף הטייפסים 16-ו `Set`. גלובלי לאפליקציה polyfill שקול להוסיף, (IE 11) או אשר אין להם מימושים תואמים (IE < 11, לדוגמה) לא מספקים אותם מושגים שחלקם מושגים ב-`core-js`.

-עם שימוש ב-16-ל-React סביבת core-js ישנים ישוויה כדי לתמוך בדפדפניים ישנים כרך: להראות

React 18 supports all modern browsers (Edge, Firefox, Chrome, Safari, etc).

If you support older browsers and devices such as Internet Explorer which do not provide modern browser features natively or have non-compliant implementations, consider including a global polyfill in your bundled application.

84ad3308338e2bb819f4f24fa8e9dfeeffaa970b

Here is a list of the modern features React 18 uses:

- `Promise`
- `Symbol`
- `Object.assign`

`<<<<< HEAD import React from 'react'; import ReactDOM from 'react-dom';`

```
ReactDOM.render(
```

Hello, world!

```
, document.getElementById('root'));
```

(אפילו בסביבות בדיקה) - גם תלוי ב `requestAnimationFrame`.

כדי לתמוך ב `raf` אתה יכול להשתמש בספרייה : `requestAnimationFrame` :

```
import 'raf/polyfill';
```

===== The correct polyfill for these features depend on your environment. For many users, you can configure your `Browserlist` settings. For others, you may need to import polyfills like `core-js` directly.

84ad3308338e2bb819f4f24fa8e9dfeeffaa970b

[Go to TOC](#)

Profiler API

The `Profiler` measures how often a React application renders and what the "cost" of rendering is. Its purpose is to help identify parts of an application that are slow and may benefit from [optimizations such as memoization](#).

Note:

Profiling adds some additional overhead, so **it is disabled in the production build**.

To opt into production profiling, React provides a special production build with profiling enabled. Read more about how to use this build at fb.me/react-profiling

Usage

A `Profiler` can be added anywhere in a React tree to measure the cost of rendering that part of the tree. It requires two props: an `id` (string) and an `onRender` callback (function) which React calls any time a component within the tree "commits" an update.

For example, to profile a `Navigation` component and its descendants:

```
render(
  <App>
    <Profiler id="Navigation" onRender={callback}>
      <Navigation {...props} />
    </Profiler>
    <Main {...props} />
  </App>
);
```

Multiple `Profiler` components can be used to measure different parts of an application:

```
render(
  <App>
    <Profiler id="Navigation" onRender={callback}>
      <Navigation {...props} />
    </Profiler>
    <Profiler id="Main" onRender={callback}>
      <Main {...props} />
    </Profiler>
  </App>
);
```

`Profiler` components can also be nested to measure different components within the same subtree:

```
render(
  <App>
    <Profiler id="Panel" onRender={callback}>
      <Panel {...props}>
```

```

        <Profiler id="Content" onRender={callback}>
          <Content {...props} />
        </Profiler>
        <Profiler id="PreviewPane" onRender={callback}>
          <PreviewPane {...props} />
        </Profiler>
      </Panel>
    </Profiler>
  </App>
);

```

Note

Although `Profiler` is a light-weight component, it should be used only when necessary; each use adds some CPU and memory overhead to an application.

onRender Callback

The `Profiler` requires an `onRender` function as a prop. React calls this function any time a component within the profiled tree "commits" an update. It receives parameters describing what was rendered and how long it took.

```

function onRenderCallback(
  id, // the "id" prop of the Profiler tree that has just committed
  phase, // either "mount" (if the tree just mounted) or "update" (if it re-
  rendered)
  actualDuration, // time spent rendering the committed update
  baseDuration, // estimated time to render the entire subtree without memoization
  startTime, // when React began rendering this update
  commitTime, // when React committed this update
  interactions // the Set of interactions belonging to this update
) {
  // Aggregate or log render timings...
}

```

Let's take a closer look at each of the props:

The `id` prop of the `Profiler` tree that has just committed. This can be used to identify which part of the tree was committed if you are using multiple profilers.

- `phase: "mount" | "update"` - Identifies whether the tree has just been mounted for the first time or re-rendered due to a change in props, state, or hooks.
- `actualDuration: number` - Time spent rendering the `Profiler` and its descendants for the current update. This indicates how well the subtree makes use of memoization (e.g. `React.memo`, `useMemo`, `shouldComponentUpdate`). Ideally this value should decrease significantly after the initial mount as many of the descendants will only need to re-render if their specific props change.
- `baseDuration: number` - Duration of the most recent `render` time for each individual component within the `Profiler` tree. This value estimates a worst-case cost of rendering (e.g. the initial mount or a tree with no memoization).
- `startTime: number` - Timestamp when React began rendering the current update.

- `commitTime: number` - Timestamp when React committed the current update. This value is shared between all profilers in a commit, enabling them to be grouped if desirable.
- `interactions: Set` - Set of "interactions" that were being traced when the update was scheduled (e.g. when `render` or `setState` were called).

Note

Interactions can be used to identify the cause of an update, although the API for tracing them is still experimental.

Learn more about it at fb.me/react-interaction-tracing

PureRenderMixin

Note

The `PureRenderMixin` mixin predates `React.PureComponent`. This reference doc is provided for legacy purposes, and you should consider using `React.PureComponent` instead.

If your React component's render function renders the same result given the same props and state, you can use this mixin for a performance boost in some cases.

Example:

```
var PureRenderMixin = require('react-addons-pure-render-mixin');
var createReactClass = require('create-react-class');
createReactClass({
  mixins: [PureRenderMixin],

  render: function() {
    return <div className={this.props.className}>foo</div>;
  }
});
```

Example using ES6 class syntax:

```
import PureRenderMixin from 'react-addons-pure-render-mixin';
class FooComponent extends React.Component {
  constructor(props) {
    super(props);
    this.shouldComponentUpdate = PureRenderMixin.shouldComponentUpdate.bind(this);
  }

  render() {
    return <div className={this.props.className}>foo</div>;
  }
}
```

Under the hood, the mixin implements `shouldComponentUpdate`, in which it compares the current props and state with the next ones and returns `false` if the equalities pass.

Note:

This only shallowly compares the objects. If these contain complex data structures, it may produce false-negatives for deeper differences. Only mix into components which have simple props and state, or use `forceUpdate()` when you know deep data structures have changed. Or, consider using [immutable objects](#) to facilitate fast comparisons of nested data.

Furthermore, `shouldComponentUpdate` skips updates for the whole component subtree. Make sure all the children components are also "pure".

React.Component

הוא מניח שאתה מכירם **קונספטים** -מפורטת עבורי ההגדרה של קומפוננטת מחלקה ב API הדף זהה מכל הפניות. אם אתם לא, קראו אותן תחילת **State** בנוסח גם, **Props** -כדוגמת **קומפוננטות ו-React**-בסיסיים ב**ומעגל-חיים**.

סקירה-כללית

מאפשרת להגדיר קומפוננטות כמחלקות או כפונקציות. קומפוננטות המוגדרות כמחלקות כרגע מספקות יותר אפשרותיות. כדי להגדיר קומפוננטת מחלקה ב-React, נדרש להרחיב ל **React.Component**:

```
class Welcome extends React.Component {
  render() {
    return <h1>Hello, {this.props.name}</h1>;
  }
}
```

כל שאר המתודות המתוארכות. **render()** המתודה היחידה שהייב להגדיר בתת-מחלקה מסווג. בדף זה הנו אופציונליות.

שימוש חוזר בקוד מושג בעיקר **על-ידי**, **אנחנו ממליצים בחום נגד ייצור קומפוננטות-בסיס משלכם**. בקומפוננטות **קומפואציה** מאשר **על-ידי** הורשה.

הערה:

אם אתם מעדיפים להימנע ממנו, אתם רשאים להשתמש ES6 לא קופה על שימוש בתחום **createClass** של גרסת React לא גרסת **React**-או הרחבה מותאמת דומה, במקום. העיפו מבט ב-**שימוש ב create-react-class** במודול ES6 כדי ללמוד עוד.

קומפוננטת מעגל-החיים

לכל קומפוננטה קיימות מספר "מתודות של מעגל חיים" שנitinן לדפוס כדי להריץ קוד בזמן מסויימים בתהליך. **אתם יכולים להשתמש בדיאגרמת מעגל-החיים הזו כשלiph**. בראשמה מטה, מתודות מעגל- החיים בשימוש נפוץ יותר, מסומנות במודגש. שאר המתודות קיימות עבור שימושים נדירים יותר.

(Mounting) עיגון

קריאה למетодות הבאות מתבצעות בסדר זהה כאשר מופע של קומפוננטה נוצר ומוכנס לתוך ה-DOM:

- **constructor()**
- **static getDerivedStateFromProps()**
- **render()**
- **componentDidMount()**

הערה:

הmethodות האלו נחשות לשינויים וכדי [להימנע מהן](#) בקוד חדש:

- `UNSAFE_componentWillUpdate()`
- `UNSAFE_componentWillReceiveProps()`

עדכון

הmethodות האלו נקראות בסדר להלן כאשר קומפוננטה מרונדרת מחדש או `props state`:

- `static getDerivedStateFromProps()`
- `shouldComponentUpdate()`
- `render()`
- `getSnapshotBeforeUpdate()`
- `componentDidUpdate()`

הערה:

הmethodות להלן נחשות שינויים ועדיף [להימנע משימוש בהן](#) בקוד חדש:

- `UNSAFE_componentWillUpdate()`
- `UNSAFE_componentWillReceiveProps()`

הסרת (Unmounting)

-קריאה למethode זו מתבצעת כאשר קומפוננטה מוסרת מה-DOM:

- `componentWillUnmount()`

ניהול שגיאות

קריאה למmethodות האלו מתבצעת כאשר קיימת שגיאה ברינדור, בmethodת מעגל-חימם, או במבנה של קומפוננטות ליד כלשהו.

- `static getDerivedStateFromError()`
- `componentDidCatch()`

ים אחרים-API

ים אחרים-API כל קומפוננטה מספקת בנוסף מספר.

- `setState()`
- `forceUpdate()`

Properties של מחלוקת

- `defaultProps`
- `displayName`

של מופע Properties

- `props`
 - `state`
-

יעון

מתודות מעגל-חימםشبשימוש נפוץ

להפנייה-המתודות בחלק זהה מכנות את החלק הנרחב של שימושים שתתקלו בהם כאשר תיצרו קומפוננטות ב-[ויזואלית, בידקו את דיאגרת מעגל-החימם הזו](#).

`render() {#render}`

`render()`

היא המתודה הנדרשת היחידה בשימוש של קומפוננטת מחלקה `()`-מתודה ה-

ותחזיר את אחד הסוגים הבאים `this.props` ו-`this.state` בקריאה אליה, המתודה תבחן את:

- **הם אלמנטים של `</>` או `<div>` בדוגמה JSX** בדרך כלל נוצרים על ידי **React**. אלמנטים של React קומפוננטת משתמש אחרת, בהתאם DOM לрендר צומת של שמנחים את React.
- **מידע נוסף** קראו את התיעוד ב-`render`-מאפשרים **fragments**.
- **Portals**. למידע נוסף **portals**-שונה. קראו את התיעוד ב-`DOM` מאפשרים לרנדר ילדים לתוך תחת-עץ.
- **מחוזות ומספרים**. מרדנדים עצמאית טקסט בתוך ה-`DOM`.
- **כasher**, `return test && <Child />` בעייר קיים כדי לתמוך בתבנית). מרדנدر כלום. `null` משתנים בוליאניים או `test` הוא משתנה בוליאני.)

של הקומפוננטה, היא מחייבת את אותה `state`-צריכה להיות טהורה, ככלומר היא לא משנה את ה `()` הפונקציה התוצאה בכל פעם שהיא מתבצעת, והיא לא מדברת ישירות עם הדף.

או במתודות האחרות של מעגל-החימם במקום. `componentDidMount()` -כאשר יש צורך לדבר עם הדף, יש להשתמש בתשمر טהורה, הופכת את החשיבה על קומפוננטות ליותר קלה `render`-שמירה על כך ש

הערה

`render() {#render}` לא תתבצע אם `shouldComponentUpdate()` מחייבת 'false'.

`constructor() {#constructor}`

`constructor(props)`

שלכם-עבור מתודות, אין צורך למש בנאי עבור קומפוננטת **state-1-binding** אם אין צורך באתחול

של הקומפוננטה. כאשר ממשים את הבניי עבור Mounting-שלכם מתבצעת לפני ה React-קריאה לבניי של קומפוננטה ה `React.Component` -נדרשת קריאה ל, לפני כל ביתוי אחר. אחרת `super(props)` תת-מחלקה מסווג בבניי, מה שיכל להוביל לבאגים (`undefined`) יהפוך להיות לא-מוגדר.

:בנאים נכתבים עבור שתי מטרות בלבד, בדרכן כולל ב:

- על ידי הקצת אובייקט ל `this.state`.
- Binding `event handler` של מethodות למופע כלשהו.

-במוקם זאת, אם קומפוננטה צריכה לשמש ב `constructor()` -בתווך ה `setState()` -לא אמורה להתבצע קריאה ל state: שירותות בתחום הבניי `this.state` -הראשוני ל `this.state`-מקומי, עליום להקצת את ה

```
constructor(props) {
  super(props);
  // לא לזכור!
  this.state = { counter: 0 };
  this.handleClick = this.handleClick.bind(this);
}
```

בכל שאר המethodות, נדרש להתבצע שימוש `this.state`. -הבניין הוא המקום היחיד שאמורה להתבצע בו הקצתה ישירה ל `this.setState()` -ב מקום.

בתוך הבניי. עבור שימושים אלו, יש (`subscriptions`) כלשהן או מינויים side-effects יש להימנע מהצורה ראשונית על מקום `componentDidMount()` -להשתמש ב

הערה

זהוי טעות נפוצה state!props יש להימנע מהעתיקת

```
constructor(props) {
  super(props);
  // לא תמשו זאת!
  this.state = { color: props.color };
}
```

-עדכונים ל) וגם יוצר באגים, (שירותות במקום `this.props.color` -אפשר להשתמש ב) הבעיה היא שזה גם לא נכון `color` prop יושקו ב `prop-state`).

-במקרים האלו, הגיוני יותר לשנות את שם ה `prop` השימוש בתבנית `thezo` אם רוצים באופן מכוון להתעלם מעדכוני הפנימי באופן `-state`-או תוכלו לגורום לקומפוננטה לבצע "אייפוס" ל `initialColor` או `defaultColor`. כפוי על-ידי שינוי המהפטה `shlo` כאשר יש צורך בכך.

כדי למדוד על מה לעשות אם אתם חושבים שאתה צריך `derived state` -קרוואו את ה `postData` בלוג שלנו על הימנעות מ `state-props`-שייהיה תלוי ב `state-props`.

componentDidMount() {#componentdidmount}

`componentDidMount()`

של הקומפוננטה (כאשר היא מוכנסת לתוך העץ). componentDidMount-מתקבצת מידית לאחר ה `(())` המתודה. יתאים להיות כאן. אם יש צורך לטעון מידע מקצה מרוחק, זה מקום טוב ליצור את בקשת הרשות DOM איתחול שדורש צמתי.

-אם אכן אתם מבצעים זאת, לא לשכוח לבטל מינוי ב `(subscription)` המתודה הזו היא מקום טוב להגדיר מינויים `componentWillUnmount()`.

זה ייגרום לרינדור נוסף, אבל הוא יקרה . `componentDidMount()` -**באופן מיידי ב `(())`** -אתם יכולים לקרוא ל תtabצע פעמיים במקרה זה, המשמש לא `render()` -לפני שהדף יעדכו את המ██ך. זה מבטיח שאיפלו למורות שקריאה ל הביניים. השתמשו בתבנית זו עם יתר זהירות מכיוון שלעתים היא יוצרת בעיות בביטויים. ברוב המקרים, state יכול לראות את מקום. עם זאת, יכול להיות שה התבנית הזו תהיה `constructor()` -ההתחלתי בתוך ה `state-state`-ניתן לבצע הקצאה ל לפניה רינדור שהוא שתלי בגודלו או מיקומו DOM-כאשר יש צורך למדוד צומת ב `1-tooltips` modal הכרחית במקרים כמו

`componentDidUpdate()` {#componentdidupdate}

`componentDidUpdate(prevProps, prevState, snapshot)`

מתבצעת מידית לאחר שהעדיכו קורה. אין קריאה למתודה הזו ברינדור הראשוני `()` המתודה.

כאשר הקומפוננטה עדכנה. זה גם מקום טוב לבצע בקשות רשות כל עד אTEM DOM-השתמשו בזה כהזרמנות לנחל את ה `props`-לדוגמא, יכול להיות שבבקשת רשות אינה הכרחית אם ה `props`-הנוכחים ל `props`-משווים את ה השתנו.

```
componentDidUpdate(prevProps) {
  // לא לשכוח להשווות את ה props!
  if (this.props.userID !== prevProps.userID) {
    this.fetchData(this.props.userID);
  }
}
```

אבל שימוש לב שקריאה כזו **מחזיבת להיות עטופה** `setState()` -ב `componentDidUpdate()` -**נitionן לקרוא באופן מיידי** ב**משפט תנאי** כמו בדוגמא מעלה, אחרת תהיה לולה אינסופטי. בנוספ, רינדור נוסף יתבצע, שלמרות שלא יהיה גליו לעינו של prop -שיקלו שימוש ב `prop` - `state` המשמש, יוכל להשפיע על ביצועיה של הקומפוננטה. אם אתם מנסים "לשקר גורם לבאים props state לתוך services", במקום. קראו עוד על **למה העתקת**.

הערך שהוא, (שהזה מקרה נדיר) `getSnapshotBeforeUpdate()` אם הקומפוננטה שלכם ממסת את מתודה מעגל-הჩיים אחריה, הפורמטו זהה יהיה לא-מוגדר . `componentDidUpdate()` -תחזר וועבר כפורמטו "תמונה-מצב" שלישי!

הערה

המתודה `componentDidUpdate()` לא תבוצע אם `shouldComponentUpdate()` תחזיר `'false'`.

`componentWillUnmount()` {#componentwillunmount}

`componentWillUnmount()`

של קומפוננטה והיררכיה. בוצעו כל ניקוי `componentWillUnmount()` מתבצעת באופן מיידי לפני `componentDidMount()` - חינוי במתודת הזו, כדוגמת ביטול טימרים, ביטול בקשות רשות, או ניקוי של כל מינו שנותר ב-

מכיוון שהקומפוננטה לעולם לא תרונדר שוב. `componentWillUnmount` -**לא אמור להתבצע ב** `setState()` -**קריאה ל** `componentWillUnmount` הוא לעולם לא יבצעUnmounting, ורגע שמוופע של קומפוננטה ביצע.

מתודות מעגל-חימם לשימוש נדייר

המתודות בחלק זהה מתכתבות עם שימושים פחות נפוצים. הן נוחות לשימוש לעיתים, אבל רוב הקומפוננטות ככל הנראה לא זוקקות להן. אתם יכולים לראות את רוב המתודות להלן **בDİAGRAMת מעגל-חימם הזו על ידי לחיצה על תיבת הסימון - "הראה מתודות מעגל-חימם פחות נפוצות"** בראשה.

`shouldComponentUpdate() {#shouldcomponentupdate}`

`shouldComponentUpdate(nextProps, nextState)`

לדעת אם פلت של קומפוננטה לא מושפע על ידי השינוי React-כדי לגרום ל-`shouldComponentUpdate` ב-השתמשו בו וברוב המקרים עליהם להסתמך על `state`, התנהגות ברירת המחדל היא לרדnder שוב בכל שינוי של `props` או `state`-הנוכחי ב-התנהגות זו.

חדש. ברירת המחדל היא `state` או `props` מתבצעת לפני רינדר בעקבות קבלת `shouldComponentUpdate()` המתודה חדשה. ברירת המחדל היא `true` -**לא מתבצעת קריאה למתחدة זו** ברינדרו הראשון או כאשר יש שימוש ב-`forceUpdate()`.

המתודה הזו קיימת רק **C-אופטימיזציה בביטויים**. אל תשטמו עליה כדי "למנוע" רינדר, זה יכול להוביל לבאגים. שיקלו לבצע `shouldComponentUpdate()` **בעצמכם** `PureComponent` **להשתמש ב** `PureComponent` **שליהם** משתנה `state`-הלא מונע מקומפוננטות-ילד לבצע רינדר חזר, כאשר ה-`false` שזהירות.

ואת `this.props` עם `nextProps` אם אתם בטוחים בעצמכם שאתם רוצים לכתוב אותה בעצמכם, רצוי שתשוו את `this.state` **ולהציגו ביטוי שקר** `nextState` **עם** `false` **שאפשר לדלג על העידכו**. שימוש לב-React-ולהציגו ביטוי שקר **ובכך** `JSON.stringify()` -**ב** `shouldComponentUpdate()` .

זה מאד לא יעיל יוכל לגרוע בביטויים **ולא ממליצים לבצע בדיקות איות עמודות או שימוש ב** `shouldComponentUpdate()` .

כל רמז, מאשר `shouldComponentUpdate()` **false** **או** `UNSAFE_componentWillUpdate()`, `render()` **או** `componentDidUpdate()` **או** `shouldComponentUpdate()` **בטעית** (**או** `shouldComponentUpdate()` **בטעית**) **אילו** `getDerivedStateFromProps()` **לא תתייחס** **ל** `React` **ולתבצעו**. **בטעית** (**או** `getDerivedStateFromProps()` **לא תובייל** **ל** `React` **ולתבצעו**) **אילו** `getDerivedStateFromProps()` **לא מונע** **מקומפוננטות-ילד** **לבצע** **רינדר** **חזר**, כאשר ה-`false` **נוקשה**, והחזרת directive **false** **כאל**.

`static getDerivedStateFromProps() {#static-getderivedstatefromprops}`

`static getDerivedStateFromProps(props, state)`

הראשוני וגם על `mount`-מופעלת וגע לפני הקריאה למתחدة הרינדר, גם ב-`getDerivedStateFromProps` המתודה.

כדי לא לעדכן כלום `null` או `undefined` לאחר מכן. עליה להחזיר אובייקט על מנת לעדכן את ה-

במהלך הזמן. לדוגמה, יתכן שהייתה זה שימושי `props`- תלוי בשינויים ב `state`-המתודה הזו קיימת **שימושים נדרים** בהם הushmanת את הילדים הקודמים והבאים שלה כדי להחליט איזה מהם `<Transition>` לשימוש במימוש של קומפוננטת להנגיש.

móvel לקוד ארוך וקשה על חשיבה על הקומפוננטות שלהם. **דו"ו שאתם מכירים מימוש עם deriving state** - שימוש ב**אלטרנטיבות פשوطות יותר**:

- השתמשו ב `props`, רק-כתגובה לשינוי ב (לדוגמה, קבלת מידע או הנפשה) **side-effect** אם אתם צריכים לכך `componentDidUpdate` מעגל-הჩיים במקום.
- במקום memoization helper**, השתמשו ב `prop` אם אתם רוצים לחשב מחדש מידע מסויים רק כאשר **משתנה**.
- במקום useState**, שקוו ליצור קומפוננטה `prop` כלשהו אשר `state` אם אתם רוצים לאפס **fully controlled with a key** במקום.

למתודה הזו אין גישה למופעים של קומפוננטות. אם תרצו, אתם יכולים לעשות שימוש חזר בקוד בין `getDerivedStateFromProps()` של `props`-וואר המתודות במהלך המחלקה על-ידי חילוץ פונקציות טהורות מתוך ה `getDerivedStateFromProps()`. הקומפוננטה מחז להגדרת המחלקה.

שימוש לב שהמתודה הזו מתבצעת בכל רינדור, ללא קשר לשיבת שהיא הפעלה. זה בנויג'ן `UNSAFE_componentWillReceiveProps` שמתבצעת רק כאשר ההוור גורם לרינדור חדש ולא כתוצאה של `setState` מקומי.

`getSnapshotBeforeUpdate()` {#getsnapshotbeforeupdate}

`getSnapshotBeforeUpdate(prevProps, prevState)`

לדוגמה. DOM-מתבצעת רגע לפני שהפלט של הרינדור האחרון יouter הועבר ל `getSnapshotBeforeUpdate()` המתודה לפני שהוא עשוי להשתנות. כל ערך שיוחזר (לדוגמה, מקום הגלילה) DOM-זה מאפשר קומפוננטה ללבוד מידע מסויים מה `componentDidUpdate()` -על-ידי מתודת מעגל-הჩיים הזו יועבר כפרמטר ל.

ימ כדוגמאות של צ'אט שצריך לנצל מקום גלילה בדרך מיוחדת-UI-השימוש הזו לא נפוץ, אבל יכול להתרחש ב.

אמור להיות מוחזר (`null` או) ערך של תמונה-מצב

לדוגמה:

```
embed:react-component-reference/get-snapshot-before-update.js
```

בכל שיכולים -ב `scrollHeight` property -בדוגמאות לעיל, חשוב לקרוא את ה `getSnapshotBeforeUpdate` כמו ב) במעגל-הჩיים "commit" -בין שלב ה `render` (כמו להיות עיקובים בין שלב "רינדור" במעגל-הჩיים `getSnapshotBeforeUpdate` -וב `componentDidUpdate`).

Error boundaries

בכל מקום בעז קומפוננטות הילד שלהם, JavaScript שטופסת שגיאות של React הם קומפוננטות של **Error boundaries**. מגובה של עז הקומפוננטה שתפקידה UI מתעדת את השגיאות האלו, ומציגת תופסת שגיאות במהלך. Error boundaries מוגבה של עז הקומפוננטה שתפקידה UI מתעדת את השגיאות האלו, ומציגת גינדור, במתודות מעגל-הჩיים, ובבנייה של כל העז שמהתחתיים.

אם היא מגדירה את אחת המתודות הבאות של מעגל-החיים (או שתיהו) - **error boundary** - קומפוננטת מחלקה הופכת ל-`static getDerivedStateFromError()`, `componentDidCatch()`. ממעגל-החיים זההאפשר לנו state עדכון של . מוגבה UI לא מטופלת בעז מתחת ולהציג JavaScript בלבד שגיאת.

להתואושש מחריגות לא צפויות בלבד; **אל תנסו להשתמש בהם עבור בקרת זרימה** **-error boundaries** השתמשו ב-

[טיפול בשגיאות ב-React 16](#).

הערה

לא יכול לתפוס שגיאה בתוך **error boundary**. תופסים שגיאות בקומפוננטות **מתחתיתם** בעז בלבד עצמוני.

`static getDerivedStateFromError() {#static-getderivedstatefromerror}`

`static getDerivedStateFromError(error)`

מעגל-חיים זה מתבצע לאחר שגיאה נזרקה על ידי קומפוננטת-צאתה. הוא מקבל את השגיאה שנזרקה כפרמטר ואמור להחזיר ערך כדי לעדכן את state.

```
class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }

  static getDerivedStateFromError(error) {
    // המוגבה UI-כדי שהринדר הבא יראה את ה-state מעדכו את ה
    return { hasError: true };
  }

  render() {
    if (this.state.hasError) {
      // מותאם UI ניתן לרנדר כל גיבוי.
      return <h1>Something went wrong.</h1>;
    }

    return this.props.children;
  }
}
```

הערה

לא מורשים. בשביל side-effects נקראת במהלך שלב "הrinדור", לכן () המתודה `getDerivedStateFromError()`, במקומ () `componentDidCatch()` מקרים כאלו, השתמשו ב-

componentDidCatch() {#componentdidcatch}

componentDidCatch(error, info)

מетодת מעגל-החיים זו מתחבצת לאחר שהגיאה נזרקה על-ידי קומפוננטה צאצא. היא מקבל שני פרמטרים

1. השגיאה שנזרקה - (error) שגיאה.
2. המכיל מידע לגבי איזו קומפוננטה ורקה את השגיאה componentStack אובייקט עם מפתח - (info) מידע.

מורשים. רצוי להשתמש בה side-effects "commit"-מתבצעת במהלך שלב ה componentDidCatch() - קריאה ל: לדברים כמו תיעוד שגיאות:

```
class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }

  static getDerivedStateFromError(error) {
    // המגובה UI-כדי שהרינדר הבא יראה את ה-state מעדכן את זה.
    return { hasError: true };
  }

  componentDidCatch(error, info) {
    // -"componentStack": דוגמה
    // -ב-ComponentThatThrows (App נוצר על-ידי)
    // -ב>ErrorBoundary (App נוצר על-ידי)
    // -ב-div (App נוצר על-ידי)
    // -ב-App
    logComponentStackToMyService(info.componentStack);
  }

  render() {
    if (this.state.hasError) {
      // גיבורי מותאם UI ניתן לרינדר כל
      return <h1>مشהו השتبש</h1>;
    }

    return this.props.children;
  }
}
```

מטוסת בשגיאות production- development יש הבדל שולי בכך שהיא בגרסאות componentDidCatch()

זה אומר שככל window.onerror או window.addEventListener('error', callback) יירטו את השגיאות שנתפסו על ידי componentDidCatch().

קודם יקבל רק שגיאות שלא נתפסו error handler במקומות, השגיאות לא יבעבו מעלה, מה שאומר שככל, ב-production componentDidCatch() במנורש על ידי .

הערה

אבל תכונה זו , `setState` -על ידי קריאה ל `componentDidCatch()` עם UI באירוע השגיאה, ניתן לרנדר גיבוי על מנת לטפל בריינדור (`static getDerivedStateFromError()` -תצא שימוש בגרסת עתידית. השתמשו ב-`magובה` במקום.

Legacy Lifecycle Methods

מתודות מעגל-ה חיים להלן מסומן כ"ישנות"./non-functional. הן עדין יעבדו, אבל אנחנו לא ממליצים להשתמש בהם בקוד חדש. אתם יכולים ללמוד עוד על מעבר מתודות מעגל- החיים המוגדרות כ"מורשת" ב-[פוסט זה](#).

`UNSAFE_componentWillMount()` {#unsafe_componentwillmount}

`UNSAFE_componentWillMount()`

הערה

לכן קריאה , `render()` מתרחש. היא נקראת לפני mounting-מבצעת רגע לפני שהיא מבוצעת. סincrונית במתודה הזו לא תיציר רינדור נוסף. בכלליות, אנחנו ממליצים על שימוש ב `constructor()` ול `setState()` במקום בשיל state.

במקום `componentDidMount()` או מינויים במתודה הזו. במקרים האלו, השתמשו ב side-effects המונעו מהгадרת כל

זו מתודה מעגל- החיים היחידה שנקרה בריינדור שרת.

`UNSAFE_componentWillReceiveProps()` {#unsafe_componentwillreceiveprops}

`UNSAFE_componentWillReceiveProps(nextProps)`

הערה

השם הזו ימשיך לעבוד עד גרסא 17. השתמשו ב `componentWillReceiveProps` מעגל- החיים הזו נקרא בעבר rename-unsafe-lifecycles [codemod](#).

הערה:

שימוש במתודה מעגל- החיים הזו לעיתים תכופות מוביל לבאגים ולהוסר עקביות.

- השתמשו ב `com`-props-כתגובה לשינוי ב (לדוגמא, קבלת מידע או הנפשה) **side-effect** אם אתם צריכים **�יזו** `ponentDidUpdate` במקום.
- אם `prop` עבור חישוב מחדש של מידע כלשהו רק כאשר `componentWillReceiveProps` השתמשם ב [memoization helper](#) במקום-`השתנה`, השתמשו ב

השתנה, שיקלו ליצור **prop כאשר state** "לאפּס `componentWillReceiveProps`" על מנת "לאפּס `state`" אם השתמשתם ב- •
במקום `fully controlled` `fully uncontrolled with a key` קומפוננטה.

למקרים אחרים, עקבו אחר המלצות בפוסט זהה מתוך הблוג על

`UNSAFE_componentWillReceiveProps()` חדשם. אם `props` מקבלת `mounting` מתבצעת לפני שקומפוננטה שעבורה `this.props` אתם יכולים להשוו את, (לדוגמא, כדי לאפס אותו) `props` כתגובה לשינוי `state`-אתם צריכים לעדכן את ה `nextProps` במתודה זו `this.setState()` באמצעות `state` ולבצע חילופי `this.setState()` ו- `nextProps`.

לא `props` שימוש לב שאם קומפוננטת הוראה גורמת לקומפוננטה שלכם להתרנדר מחדש, תבוצע קריאה למתחודה הזו אפיו אם השתנו. ודו"ח להשוו את הערכים הנוכחיים והבאים אם אתם רוצים ורק לטפל בשינויים.

React הוא קוראת `UNSAFE_componentWillReceiveProps()` התחלתיים במהלך `mounting`.- לא קוראת ל- `UNSAFE_componentWillReceiveProps()` עםprops בדרך כלל לא מעוררת `this.setState()`- של הקומפוננטה עשויים להתעדכו. קריאה ל- `props` למתחודה הזו ורק אם מסטר `UNSAFE_componentWillReceiveProps()` - קריאה ל-

`UNSAFE_componentWillUpdate()` {#unsafe_componentwillupdate}

`UNSAFE_componentWillUpdate(nextProps, nextState)`

הערה

✓-השם הזה ימשיך לעבוד עד לגרסת 17. השתמשו ב- `componentWillUpdate` מתחודה מ Engel-החיים הזו נקרא בעבר `ename-unsafe-lifecycles` [codemod](#).
עדכון אוטומטי של הקומפוננטות שלכם [codemod](#)

חדשם. השתמשו בה `state` או `props` מתבצעת רגע לפני רינדור כתוואה מקבלת `UNSAFE_componentWillUpdate()` כהזרנות לבצע הכנות לפני שעדכו מתבצע. המתחודה הזו לא נקראת ברינדור הראשוני.

כאנ; גם אתם לא אמרים לעשות דבר שיגרום לעדכו קומפוננטה `this.setState()` - שימוש לב שאתם לא יכולים לקרוא ל- `UNSAFE_componentWillUpdate()` - [React](#) ש- `UNSAFE_componentWillUpdate()` - לפניו ש- `componentDidUpdate()` בדרך כלל, המתחודה הזו יכולה להיות מוחלפת על-ידי `getSnapshotBeforeUpdate()` - (לדוגמא, כדי לשמור מקום גלילה), אתם יכולים להעביר את הלוגיקה הזו לו

הערה

`UNSAFE_componentWillUpdate()` לא תבוצע אם `shouldComponentUpdate()` מחזירה `false`.

ים אחרים-API

המתודות להלן הן מתודות שאנו יכולים לקרוא להן, (קוראת להן עבורנו React אשר) לא כמו מתודות Engel-החיים למעלה מהקומפוננטות שאנו יוצרים.

יש רק שניים כאלה: `setState()` ו-`forceUpdate()`.

`setState()` {#setstate}

`setState(updater, [callback])`

הڪومפוננטה הזו והילדים שלה צריכים React-של הקומפוננטה ואומרת ל-state-מכלנית לתרו שינויים ב ()(). השינויים ב ()().

זה המתודה הראשית שבה משתמשים כדי לעדכן את מושך המשתמש בתגובה state-state-להיות מודדים מחדש עם המעדכן. זו המתודה הראשית שבה משתמשים כדי לעדכן את מושך המשתמש בתגובה state-state-להיות מודדים מחדש עם המעדכן.

כאל בקשה מאשר כל פקודה מידית לעדכן הקומפוננטה. מען ביצועים ()() חישבו על HEAD <<<<<< לא מבטיחה React. עשויה ליעכט את ביצועה של הפקודה, אז לעדכן מספר קומפוננטות בעבר אחד, טובים יותר ===== Think of `setState()` as a *request* rather than an immediate command to update the component. For better perceived performance, React may delay it, and then update several components in a single pass. In the rare case that you need to force the DOM update to be applied synchronously, you may wrap it in `flushSync`, but this may hurt performance.

84ad3308338e2bb819f4f24fa8e9dfeeffaa970b

לא תמיד מעדכנת באופן מיידי את הקומפוננטה. היא עשויה לדוחות את העידכון לאחר כך. זה גורם לכך ()() לא-מלכחות פוטנציאלית. במקום, השתמשו ב ()() -ישר אחרי קריאה ל `this.state` מ-`componentDidUpdate` ()(), או בפונקציית ה `callback` -ב `setState(updater, callback)` ()().

בכל אחד מהמקרים מובטח כי יתבצע לפחות שעה עדכון יקרה. אם יש צורך להגדיר את ההתקדים, קראו על state-state-בהתבסס על ה `updater` הארגומנט מטה.

אם אובייקטים שניים `shouldComponentUpdate()` תחזיר `false`. תמיד יוביל לרינדור חדש אלא אם ()(). נמצאים בשימוש ולוגיקת רינדור מותנה לא יכול להיות ממומשת ב ()(). לשינוי `shouldComponentUpdate()`, החדר שונה מהקדם תמנע רינדור חדש מיותר רק כאשר ה ()(). קריאה ל `setState()`.

עם החתימה `updater` הארגומנט הראשון הוא פונקציה:

(state, props) => stateChange

של הקומפוננטה בזמן שהשינוי היה מבוצע. לא אמורים לבצע שינויים ישירות. במקום, שינויים state-state-הוא הפניה ל state, נניח שרצינו להגדיל ערך . state -צריכים להיות מוצגים על ידי בניית אובייקט חדש המבוסס על הקלט מ props.step : על-ידי-state :

```
this.setState((state, props) => {
  return {counter: state.counter + props.step};
});
```

שהתקבלו על ידי פונקציית העדכון מובטחים להיות מעודכנים. הפלט של העדכון מתאחד באופן רוחני גם stateprops עם state .

תושלם והקומפוננטה setState -אופציונלית שתבוצע ברגע ש callback הוא פונקציה ()() -הפרמטר השני ל setStatecomponentDidUpdate() -טרונדר מחדש. בכללות, אנחנו ממליצים על שימוש ב לוגיות פשוטה, במקום, במקום setState ()()

במקום פונקציה setState ()() -אופציונלית, ניתן להעביר אובייקט כารוגמנט הראשון ל setState(stateChange[, callback])

החדש, לדוגמה כדי לשנות כמהות של פריט בעגלת קניות state-state-לתוכה stateChange דרך זו מבצעת איחוד רוחני של:

```
this.setState({quantity: 2})
```

היא גם אסינכרונית, וקריאה מרובות במהלך אותו מעגל-חיים עלולים להתנגש. לדוגמה, אם setState ()() הוצאה הזו של:Nנסה להגדיל כמהות של פריט יותר מפעם אחת באותו מעגל, זה יגרום לתזואה השකולה ל

```
Object.assign(
  previousState,
  {quantity: state.quantity + 1},
  {quantity: state.quantity + 1},
  ...
)
```

הבא state-קריאה עוקבות יגרמו לדרישה של ערכים מקריות קדמות באותו מעגל, שכן הכמות תגדל פעם אחת בלבד. אם ה הנוכחי, אנחנו ממליצים להשתמש בצורה של פונקציית עדכון, במקום state-state-תלוי ב:

```
this.setState((state) => {
  return {quantity: state.quantity + 1};
});
```

למידע נוסף, עיין:

- [State and Lifecycle guide](#)
- [In depth: When and why are setState\(\) calls batched?](#)
- [In depth: Why isn't this.state updated immediately?](#)

forceUpdate() {#forceupdate}

```
component.forceUpdate(callback)
```

של הקומפוננטה שלכם משתנים, הקומפוננטה שלכם תרונדר מחדש. אם `props`-או `state`-בתור ברירת מחדל, כאשר ה-`render()` מותדת ה-`forceUpdate()`.

בקומפוננטה, תוך כדי דילוג על `forceUpdate()`-תגרום לכך שתבוצע קריאה ל `shouldComponentUpdate()`. זה יעורר תגובה של מתחודות מעגל-החיים הרגיל לקומפוננטותILD, לרבות מותדת ה-`shouldComponentUpdate()`. ישנה DOM-markup-ה עדין רק תערכן את זה. של כל ILD `forceUpdate()`-הדרך הנכונה היא לנסות להמנע שימוש ב-

`render()` - ווק לקרוא מ `this.props` ו- `this.state` ב-`forceUpdate()`-הדרך הנכונה היא לנסות להמנע שימוש ב-

Properties של מחלקות (Class Properties)

defaultProps {#defaultprops}

ברירת-הchodל עבור `props` במחלקה הקומפוננטה עצמה, כדי להגדיר את `property`-יכל להיות מוגדר כ `undefined` שם `props` אבל לא עבור: לדוגמה.

```
class CustomButton extends React.Component {
  // ...
}

CustomButton.defaultProps = {
  color: 'blue'
};
```

ערך יקבע לפי ברירת-הchodל, אם אין ערך ל `props.color`:

```
render() {
  return <CustomButton />; // props.color יקבע להיות כחול
}
```

אם `props.color` null מוגדר להיות null:

```
render() {
  return <CustomButton color={null} />; // props.color ישאר null
```

displayName {#displayname}

בשימוש בהודעות דיבאגינג. בדרך כלל, אנחנו לא צריכים להגדיר אותה באופן מפורש בגלל שהשם `displayName` המחרוזת שלה נגזר מהשם של הפונקציה או המחלקה שגדירה את הקומפוננטה. ניתן שנרצה להגדיר אותה מפורשת אם נרצה להציג שם אחר עבור מטרות דיבאגינג או כאשר ניצור קומפוננטה מסדר גובה יותר, עיננו ב-**עלוטף את שם התצוגה עבור דיבאגינג פשוט** בשבייל מידע נוסף.

Properties של מופע (Instance Properties)

props {#props}

עבור הקדמה [props](#)-שהוגדרו על ידי המפעיל של הקומפוננטה הזו. עיין ב-[קומפוננטות](#) ו-[props](#)-מקן את ה-props.

מאשר בטאג עצמו JSX-מיוחד, בדרך כלל מוגדר על ידי הטאגים של הילד בבייטוי ה-prop הוא `this.props.children`.

state {#state}

מוגדר על-ידי המשתמש, והוא אמור state-מקיל מידע ספציפי לקומפוננטה הזאת שעשו להשתנות במהלך הזמן. ה-state-ה פשוט JavaScript להיות אובייקט.

-אין חובה לשים אותו בתוך ה-`(, timer ID)`, אם ערך כלשהו לא בשימוש עבור RIDOR או עבור זרימת מידע. ערכיהם יכולים להיות מוגדרים כסדרות בקומפוננטה המופע.

לאחר מכן עלולה להחליף את השינוי `this.state`-ישירות, מכיוון שקריאה ל-`setState()` לעולם אל תנסה אתכך. אולי היה אפשר `this.state`-שביצעתם. התיחסו ל-

[Go to TOC](#)

ReactDOMClient

The `react-dom/client` package provides client-specific methods used for initializing an app on the client. Most of your components should not need to use this module.

```
import * as ReactDOM from 'react-dom/client';
```

If you use ES5 with npm, you can write:

```
var ReactDOM = require('react-dom/client');
```

Overview

The following methods can be used in client environments:

- `createRoot()`
- `hydrateRoot()`

Browser Support

React supports all modern browsers, although [some polyfills are required](#) for older versions.

Note

We do not support older browsers that don't support ES5 methods or microtasks such as Internet Explorer. You may find that your apps do work in older browsers if polyfills such as [es5-shim](#) and [es5-sham](#) are included in the page, but you're on your own if you choose to take this path.

Reference

`createRoot()` [\[#createroot\]](#)

```
createRoot(container[, options]);
```

Create a React root for the supplied `container` and return the root. The root can be used to render a React element into the DOM with `render`:

```
const root = createRoot(container);
root.render(element);
```

`createRoot` accepts two options:

- `onRecoverableError`: optional callback called when React automatically recovers from errors.
- `identifierPrefix`: optional prefix React uses for ids generated by `React.useId`. Useful to avoid conflicts when using multiple roots on the same page. Must be the same prefix used on the server.

The root can also be unmounted with `unmount`:

```
root.unmount();
```

Note:

`createRoot()` controls the contents of the container node you pass in. Any existing DOM elements inside are replaced when `render` is called. Later calls use React's DOM diffing algorithm for efficient updates.

`createRoot()` does not modify the container node (only modifies the children of the container). It may be possible to insert a component to an existing DOM node without overwriting the existing children.

Using `createRoot()` to hydrate a server-rendered container is not supported. Use `hydrateRoot()` instead.

hydrateRoot() {#hydrateroot}

```
hydrateRoot(container, element[, options])
```

Same as `createRoot()`, but is used to hydrate a container whose HTML contents were rendered by `ReactDOMServer`. React will attempt to attach event listeners to the existing markup.

`hydrateRoot` accepts two options:

- `onRecoverableError`: optional callback called when React automatically recovers from errors.
- `identifierPrefix`: optional prefix React uses for ids generated by `React.useId`. Useful to avoid conflicts when using multiple roots on the same page. Must be the same prefix used on the server.

Note

React expects that the rendered content is identical between the server and the client. It can patch up differences in text content, but you should treat mismatches as bugs and fix them. In development mode, React warns about mismatches during hydration. There are no guarantees that attribute differences will be patched up in case of mismatches. This is important for performance reasons because in most apps, mismatches are rare, and so validating all markup would be prohibitively expensive.

[Go to TOC](#)

ReactDOMServer

Node סטוי. בדרך כלל, נעשה בו שימוש בשרת markup-ΜΑΡΧΟΥΝΤΟΣ ל `ReactDOMServer` האובייקט:

```
// ES modules
import * as ReactDOMServer from 'react-dom/server';
// CommonJS
var ReactDOMServer = require('react-dom/server');
```

סקירה כללית

<<<<< ===== These methods are only available in the **environments with Node.js Streams**:

- `renderToPipeableStream()`
- `renderToNodeStream()` (Deprecated)
- `renderToStaticNodeStream()`

These methods are only available in the **environments with Web Streams** (this includes browsers, Deno, and some modern edge runtimes):

- `renderToReadableStream()`

The following methods can be used in the environments that don't support streams:

84ad3308338e2bb819f4f24fa8e9dfeeffaa970b

- `renderToString()`
- `renderToStaticMarkup()`

<<<<< HEAD אשר זמין בשרת בלבד, ולא יפעלו בדף (`stream`) מתחודות נוספת אלה תלויות בחבילת `HEAD`.

- `renderToNodeStream()`
- `renderToStaticNodeStream()`

סיכום

===== ## Reference {#reference}

84ad3308338e2bb819f4f24fa8e9dfeeffaa970b

`renderToPipeableStream()` {#rendertopipeablestream}

ReactDOMServer. `renderToPipeableStream(element, options)`

ניתן להשתמש בשיטה זו HTML יחזיר מהירותו של HTML מレンדר קומפוננטת `HEAD` <<<<< על הבקשה הראשונית עבר טעינות דף מהירות יותר ועל מנת לאפשר למנועי markup-בשרות ושלוח את ה HTML כדי ליצור חיפוש לסרוק את הדפים שלכם למטרות SEO.

על איבר `ReactDOM.hydrate()`-אם תקרוו לשרונדר על-ידי השרת `markup`-שכבר כולל את ה `React` תשמר אותו וرك תצמיד מטפלי אירועים, דבר המאפשר לך לבצע חווית טעינה-ראשונה עם ביצועים טובים מאוד.

Render a React element to its initial HTML. Returns a stream with a `pipe(res)` method to pipe the output and `abort()` to abort the request. Fully supports Suspense and streaming of HTML with "delayed" content blocks "popping in" via inline `<script>` tags later. [Read more](#)

If you call `ReactDOM.hydrateRoot()` on a node that already has this server-rendered markup, React will preserve it and only attach event handlers, allowing you to have a very performant first-load experience.

```
let didError = false;
const stream = renderToPipeableStream(
  <App />,
  {
    onShellReady() {
      // The content above all Suspense boundaries is ready.
      // If something errored before we started streaming, we set the error code
      // appropriately.
      res.statusCode = didError ? 500 : 200;
      res.setHeader('Content-type', 'text/html');
      stream.pipe(res);
    },
    onShellError(error) {
      // Something errored before we could complete the shell so we emit an
      // alternative shell.
      res.statusCode = 500;
      res.send(
        '<!doctype html><p>Loading...</p><script src="clientrender.js"></script>'
      );
    },
    onAllReady() {
      // If you don't want streaming, use this instead of onShellReady.
      // This will fire after the entire page content is ready.
      // You can use this for crawlers or static generation.

      // res.statusCode = didError ? 500 : 200;
      // res.setHeader('Content-type', 'text/html');
      // stream.pipe(res);
    },
    onError(err) {
      didError = true;
      console.error(err);
    },
  },
);
```

See the [full list of options](#).

Note:

This is a Node.js-specific API. Environments with [Web Streams](#), like Deno and modern edge runtimes, should use `renderToReadableStream` instead.

84ad3308338e2bb819f4f24fa8e9dfeeffaa970b

`renderToReadableStream()` [\[#rendertoreadablestream\]](#)

`ReactDOMServer.renderToReadableStream(element, options);`

משתמשת React-נוספים ש DOM מלבד העובדה שאינה יוצרת מאפייני, `<<<<< HEAD` - דומה ל `renderToString`, במחולל של דפים סטטיים React-אפשרות זו שימושית אם ברצונך להשתמש ב `data-reactroot` בהם באופן פנימי, כגון פשוטים, שכן השרת המאפיינים הנוספים יכולה לחסוך כמה בתים.

**על מנת React-אם אתם מתכוונים לשימוש ב
לאינטראקטיבי, אל markup-להפוך את ה-
שימוש במתודה זו. במקום זאת, השימוש ב
`renderToString` ובצד השרת וב
`ReactDOM.hydrate()` בצד הל��וח.**

Streams a React element to its initial HTML. Returns a Promise that resolves to a [Readable Stream](#). Fully supports Suspense and streaming of HTML. [Read more](#)

If you call `ReactDOM.hydrateRoot()` on a node that already has this server-rendered markup, React will preserve it and only attach event handlers, allowing you to have a very performant first-load experience.

```
let controller = new AbortController();
let didError = false;
try {
  let stream = await renderToReadableStream(
    <html>
      <body>Success</body>
    </html>,
  {
    signal: controller.signal,
    onError(error) {
      didError = true;
      console.error(error);
    }
  }
);

// This is to wait for all Suspense boundaries to be ready. You can uncomment
// this line if you want to buffer the entire HTML instead of streaming it.
// You can use this for crawlers or static generation:

// await stream.allReady;

return new Response(stream, {
  status: didError ? 500 : 200,
  headers: {'Content-Type': 'text/html'},
});
} catch (error) {
  return new Response(
    '<!doctype html><p>Loading...</p><script src="clientrender.js"></script>',
  {
    status: 500,
    headers: {'Content-Type': 'text/html'},
  }
);
}
```

See the full list of options.

Note:

This API depends on [Web Streams](#). For Node.js, use `renderToPipeableStream` instead.

84ad3308338e2bb819f4f24fa8e9dfeffaa970b

renderToString() (Deprecated)

 `ReactDOMServer.renderToString(element)`

שמייצא (Readable stream) הראשוני שלה. מחזירה **זם קריין** HEAD-ל React קומפוננטות HTML-HTML מレンדר קומפוננטות HEAD-HTML ישירות. ניתן להשתמש `ReactDOMServer.renderToString`-מזרם זה שווה בדיקת מה ש-HTML-פלט ה HTML מחרחות על הבקשה הראשונית עבור טעינות דף מהירות יותר ועל מנת markup-בשרות ולשלוח את ה HTML במתודה זו כדי ליצור SEO לאפשר למנועי חיפוש לסרוק את הדפים שלכם למטרות SEO.

על איבר `ReactDOM.hydrate()`-אם תקרוו לשרונדר על-ידי השרת markup- שכבר כולLAT ה **React תשמור אותו ורתק תצמיד ملفי אירופיים, דבר המאפשר לכם לבצע חווית טעינה-ראשונה עם ביצועים טובים מאוד.**

Render a React element to its initial HTML. Returns a [Node.js Readable stream](#) that outputs an HTML string. The HTML output by this stream is exactly equal to what `ReactDOMServer.renderToString` would return. You can use this method to generate HTML on the server and send the markup down on the initial request for faster page loads and to allow search engines to crawl your pages for SEO purposes.

If you call `ReactDOM.hydrateRoot()` on a node that already has this server-rendered markup, React will preserve it and only attach event handlers, allowing you to have a very performant first-load experience.

84ad3308338e2bb819f4f24fa8e9dfeeffaa970b

שימוש לב:

זה אינו זמין בדף API שרת בלבד. ממשק

אם תזקקו לזרם בקידוד אחר, השתכלו על פרויקט כמו `utf-8-zrom` המוחזר ממתודה זו יחזיר זרם בתים מיוחד ב [iconv-lite](#), המספק זרמי טרנספורמציה עברו קידוד טקסט.

`renderToStaticNodeStream()` **{#rendertostaticnodestream}**

`ReactDOMServer.renderToStaticNodeStream(element)`

משתמשת בהם באופן פנימי, React-נוספים ש DOM מלבד העבודה שאינה יוצרת מאפייני, -דוגמה ל כמחולל של דפים פשוטים פשוטים, שכן השרת React-אפשרות זו שימושית אם ברצונך להשתמש ב . המאפיינים הנוספים יכולה לחסוך בתים

תחזר `ReactDOMServer.renderToStaticMarkup` -המוחזר מזרם זה זהה למזה ש-HTML-פלט ה.

לאינטראקטיבי, אל תשימושו עלי מנת להפוך את ה React-markup מתוכנים להשתמש ב HEAD <<<<<< בצד הלוקה () -בצד השרת וב `renderToNodeStream` -`ReactDOM.hydrate()` במקומ זה. במקרה זאת, השימוש ב ===== If you plan to use React on the client to make the markup interactive, do not use this method. Instead, use `renderToNodeStream` on the server and `ReactDOM.hydrateRoot()` on the client.

84ad3308338e2bb819f4f24fa8e9dfeeffaa970b

שימוש לב:

זה אינו זמין בדף API שרת בלבד. ממשק

אם תזדקקו לזרם בקידוד אחר, הסתכלו <<<<< HEAD המוחזר ממתודה זו יחזיר זרם בתים מקודד ב utf-8-הורות המוחזר ממתודה זו יחזיר זרם בתים מקודד ב HEAD. המספק זרמי טרנספורמציה עבור קידוד טקסט כמו [iconv-lite](#), על פרויקט כמו [iconv-lite](#), The stream returned from this method will return a byte stream encoded in utf-8. If you need a stream in another encoding, take a look at a project like [iconv-lite](#), which provides transform streams for transcoding text.

`renderToString()` [\[#rendertostring\]](#)

`ReactDOMServer.renderToString(element)`

Render a React element to its initial HTML. React will return an HTML string. You can use this method to generate HTML on the server and send the markup down on the initial request for faster page loads and to allow search engines to crawl your pages for SEO purposes.

If you call `ReactDOM.hydrateRoot()` on a node that already has this server-rendered markup, React will preserve it and only attach event handlers, allowing you to have a very performant first-load experience.

Note

This API has limited Suspense support and does not support streaming.

On the server, it is recommended to use either `renderToPipeableStream` (for Node.js) or `renderToReadableStream` (for Web Streams) instead.

`renderToStaticMarkup()` [\[#rendertostaticmarkup\]](#)

`ReactDOMServer.renderToStaticMarkup(element)`

Similar to `renderToString`, except this doesn't create extra DOM attributes that React uses internally, such as `data-reactroot`. This is useful if you want to use React as a simple static page generator, as stripping away the extra attributes can save some bytes.

If you plan to use React on the client to make the markup interactive, do not use this method. Instead, use `renderToString` on the server and `ReactDOM.hydrateRoot()` on the client.



84ad3308338e2bb819f4f24fa8e9dfeeffaa970b

ReactDOM

מהרמה העליונה הבאים זמינים תחת HEAD אם אתם טוענים את React מותגי `<script>`, APIs-ה. אם אתם יכולים לכתוב npm עם `ReactDOM`-ם אתם משתמשים ב. המשטנה הגלובלי `'react-dom'`. ===== The `react-dom` package provides DOM-specific methods that can be used at the top level of your app and as an escape hatch to get outside the React model if you need to.

```
import * as ReactDOM from 'react-dom';
```

If you use ES5 with npm, you can write:

```
var ReactDOM = require('react-dom');
```

The `react-dom` package also provides modules specific to client and server apps:

- `react-dom/client`
- `react-dom/server`

84ad3308338e2bb819f4f24fa8e9dfeeffaa970b

סקירה כללית

שניתן להשתמש בהן ברמה העליונה של DOM-ספקט מותודות ספציפיות ל החבילת HEAD react-dom אם יש לכם צורך בכך. לרוב הקומפוננטות שלכם לא אמרו האפליקציה שלכם וכפתח מילוט כדי להציג אל מחוץ למדול זה ===== The `react-dom` package exports these methods:

- `createPortal()`

- `flushSync()`

84ad3308338e2bb819f4f24fa8e9dfeeffaa970b

These `react-dom` methods are also exported, but are considered legacy:

- `render()`
- `hydrate()`
- `findDOMNode()`
- `unmountComponentAtNode()`

Note:

Both `render` and `hydrate` have been replaced with new [client methods](#) in React 18. These methods will warn that your app will behave as if it's running React 17 (learn more [here](#)).

תמייה בדףנים

ומעלה, אם כי [כמה](#) HEAD React 9 תומכת בכל הדפננים הפופולריים, כולל polyfills <<<<< Internet Explorer 9 IE 9-ו IE 10. ===== React supports all modern browsers, although some polyfills are required for older versions.

84ad3308338e2bb819f4f24fa8e9dfeeffaa970b

הערה

אך ייתכן שתגלו, ES5 אנו לא תומכים בדףנים יניים יותר מאשר תומכים במתודות HEAD <<<<<< אונט. אתם כגון polyfills שהאפליקציות שלכם פועלות בדףנים יניים יותר אם תכללו בעמוד לבדכם במערכת אם תבחרו לחת את הנתיב זהה.

=====

We do not support older browsers that don't support ES5 methods or microtasks such as Internet Explorer. You may find that your apps do work in older browsers if polyfills such as [es5-shim](#) and [es5-sham](#) are included in the page, but you're on your own if you choose to take this path.

84ad3308338e2bb819f4f24fa8e9dfeeffaa970b

סיכום

`createPortal()` [\[#createportal\]](#)

`createPortal(child, container)`

<<<<< HEAD -לתוך ה-React מrndrat אלמנטי `container` שספק ומחזירה הפניה [\(reference\)](#) עבור קומפוננטות חסרות `null` או מחזירה `state`). ===== Creates a portal. Portals provide a way to render children into a DOM node that exists outside the hierarchy of the DOM component.

`flushSync()` [\[#flushsync\]](#)

`flushSync(callback)`

Force React to flush any updates inside the provided callback synchronously. This ensures that the DOM is updated immediately.

```
// Force this state update to be synchronous.
flushSync(() => {
  setCount(count + 1);
});
// By this point, DOM is updated.
```

Note:

`flushSync` can significantly hurt performance. Use sparingly.

`flushSync` may force pending Suspense boundaries to show their `fallback` state.

`flushSync` may also run pending effects and synchronously apply any updates they contain before returning.

`flushSync` may also flush updates outside the callback when necessary to flush the updates inside the callback. For example, if there are pending updates from a click, React may flush those before flushing the updates inside the callback.

Legacy Reference

`render()` [\[#render\]](#)

`render(element, container[, callback])`

Note:

`render` has been replaced with `createRoot` in React 18. See [createRoot](#) for more info.

Render a React element into the DOM in the supplied `container` and return a [reference](#) to the component (or returns `null` for [stateless components](#)).

```
84ad3308338e2bb819f4f24fa8e9dfeeffaa970b
```

לפי הצורך DOM-פועלה זו תבצע עדכון עבורה וرك תנסה את ה `container`, רונדרה קודם לכן ל-React-אם קומפוננטת ה-React-אחרון כדי לשחק את אלמנט ה-

האופציוני מסופק, הוא יורץ לאחר שהקומפוננטה רונדרה או עדכנה callback-אם ארגומנט ה-

:הערה

DOM-שליטה בתוכן של הצומת המכיל שאתם מעבירים. כל אלמנטי ה `ReactDOM.render()` React של DOM-הקיימים בתוכו מוחלפים בקריאה הראשונה. קריאות מאוחרות יותר משתמשות באlgorigisms הבדلت הלביצוע עדכוניים יעילים.

לא משנה את הצומת המכיל (רק משנה את הילדים של המכיל). ניתן שייהי אפשר להכניס `ReactDOM.render()` קיים מבלי לדرس את הילדים המקוריים DOM קומפוננטה לצומת.

אלמנט השורש callback-הפנייה ל

כדי לנוקות (`ReactDOM.render()` -שימוש ב- תוכן אלמנט שנשלח על-ידי השרת הוצאה ב-React 17. במקומם (`hydrate()` -השתמשו ב-

`render()` controls the contents of the container node you pass in. Any existing DOM elements inside are replaced when first called. Later calls use React's DOM diffing algorithm for efficient updates.

`render()` does not modify the container node (only modifies the children of the container). It may be possible to insert a component to an existing DOM node without overwriting the existing children.

`render()` currently returns a reference to the root `ReactComponent` instance. However, using this return value is legacy and should be avoided because future versions of React may render components asynchronously in some cases. If you need a reference to the root `ReactComponent` instance, the preferred solution is to attach a `callback ref` to the root element.

Using `render()` to hydrate a server-rendered container is deprecated. Use `hydrateRoot()` instead.

84ad3308338e2bb819f4f24fa8e9dfeeffaa970b

hydrate() {#hydrate}

`hydrate(element, container[, callback])`

שלו רונדר על-ידי HTML-אבל משמשת לניקוי של צומת מכל שתוכנן, כמו `render()` (HEAD).
ReactDOMServer. הקדים-markup-תנסה לצרף מנהלי אירופים =====

Note:

`hydrate` has been replaced with `hydrateRoot` in React 18. See [hydrateRoot](#) for more info.

Same as `render()`, but is used to hydrate a container whose HTML contents were rendered by `ReactDOMServer`. React will attempt to attach event listeners to the existing markup.

84ad3308338e2bb819f4f24fa8e9dfeeffaa970b

מצפה שהתוכן המרונדר יהיה זהה בין השרת לבון הלוקו. היא יכולה לתקן את ההבדלים בתוכן טקסט, אבל אתם מזהירה מפני אי התאמות במהלך React, צריכים לטפל באין-התאמות כפי שאתה מטפלים בבאגים ולתקן אותם. במצב פיתוח הnickio. אין הבטחות כי הבדלי מאפיינים יתוקנו במקרה של אי-התאמה. זה חשוב מסוות של ביצועים כי ברוב האפליקציות, יהיה יקר מדי כך שנרצה להמנע ממנו `markup`-אי-התאמות הוו דבר נדיר, וכן אינומת כל ה

אם מאפיין או תוכן טקסט של אלמנט בודד שונה באופן בלתי נמנע בין השרת לבון הלוקו (לדוגמה, חותמת זמן), תוכלוلالמןנט. זה עובד רק עבור רמת עומק אחת, `{suppressHydrationWarning={true}}` להשתיק את האזהרה על ידי הוספת עדין לא תנסה לתקן אותו, React, ונraud להיות ורק כפתח מילוט. אל תשתמשו בהז יותר מדי. אלא אם כן מדובר בתוכן טקסט לכך יתכן שהוא ישאר בלבתי עקיبي עד לעדכונים עתידיים.

אם אתם בכונה צריכים לרנדר משהו שונה בצד השרת ובצד הלוקו, אתם יכולים לבצע רינדור בשני מעברים. קומפוננטות שאוטו תוכלו להגיד לך, `this.state.isClient`, כמו `state`-המרנדורות משהו אחר בצד הלוקו יכולות לקרוא משתנה מה דרך זו מעבר הרינדור הראשוני לרנדר את אותו תוכן כמו השרת, תוך הימנעות . () - ב- `componentDidMount` מאין-התאמות, אבל מעבר נוסף יקרה באופן סינכרוני מיד לאחר הnickio. שימושו לב כי גישה זו תהפוך את הקומפוננטות שלכם לאיכות יותר כי הן צריכים לרנדר פעמים, אז השתמשו בה בזיהירות.

עלול להטען מאוחר יותר באופן שימושותי מאשר JavaScript-זכוו לחיות המשמש על חיבוריהם איטיים. קוד הראשוני, כך שאם מレンדרים משהו ברינדור בלבד הלקוח בלבד, המעבר יכול להיות צורם. עם זאת, HTML-רינדור אם מבוצע היטב, זה עשוי להיות מועילレンדר "מעטפת" של האפליקציה בשרת, ו록 להראות כמה ויזיג'טטים נוספים אצל עיינו בהסביר בפסקה הקדמתה-markup-הלקוח. כדי ללמוד כיצד לעשות זאת מבלי לקבל בעיות אי-ההתאמה ב-

unmountComponentAtNode() {#unmountcomponentatnode}

`unmountComponentAtNode(container)`

<<<<< HEAD אם ומנקה את מנהלי האירועים ואת ה DOM-מטופעת מה React מסירה קומפוננטה אם ביטול תפעול `true` קרייה לפונקציה זו אינה עשו דבר. מחרירה `container`, הקומפוננטה לא הופעה על ה `false`. אם לא הייתה קומפוננטה כדי לבטל את הפעלה ==. אם לא היה קומפוננטה כלוח ו

Note:

`unmountComponentAtNode` has been replaced with `root.unmount()` in React 18. See [createRoot](#) for more info.

Remove a mounted React component from the DOM and clean up its event handlers and state. If no component was mounted in the container, calling this function does nothing. Returns `true` if a component was unmounted and `false` if there was no component to unmount.

84ad3308338e2bb819f4f24fa8e9dfeeffaa970b

`findDOMNode()` **{#finddomnode}**

הערה:

הבסיסי. ברוב המקרים, שימוש בفتح מילוט זה DOM-היא פתח מילוט המשמשת כדי לגשת לצומת ה `findDOMNode` -הוא לא מומלץ מכיוון שהוא אטרטקטיבית של הקומפוננטה. [היא הוצאה משימוש ב StrictMode](#).

`findDOMNode(component)`

התואם המקורי של הדף. מתחדשה זו DOM-קריאה זו מחרירה את אלמנט ה, DOM-אם קומפוננטה זו כבר הופעלה לתוך ה **ברוב המקרים, ניתן לצרף הפניה**. כגון שדות טופס ובחירה מדינות, DOM-שימוש עבור קריאת ערכים מתוך ה `findDOMNode`.

כאשר קומפוננטה מרונדרת למחזרות. `null` מחרירה `null` או `false`, `findDOMNode` -כאשר קומפוננטה מרונדרת ל `findDOMNode` קומפוננטה יכולה להחזיר קטע עם, React 16-מסוג טקסט המכילה את הערך. החל מ DOM מחרירה צומת `findDOMNode` מספר ילדים, ובמקרה זה התואם ליד הראשון שאינו ריק DOM-תחזיר את צומת ה `findDOMNode`.

הערה:

אם אתם מנסים. (DOM-כלומר, קומפוננטות שכבר הוכנסו לו) עובדת רק על קומפוננטות שהופעלו על קומפוננטה מתוך `findDOMNode()` מתחוך `render()` כמו קריאה לו) לקרוא לה על קומפוננטות שעדיין לא הופעלו תזדק שגיאה (שעדין לא נוצרה.

לא יכולה לקרוא על קומפוננטות פונקציה `findDOMNode`.

<<<<< HEAD

`createPortal()` **{#createportal}**

`ReactDOM.createPortal(child, container)`

מייצרת פורטל. פורטלים מספקים דרךランדר ילדים שקיימים מחוץ להיררכיה של DOM לתוכן צומת DOM-קומponentת ה-

84ad3308338e2bb819f4f24fa8e9dfeeffaa970b

React Top-Level API

`React` is the entry point to the React library. If you load React from a `<script>` tag, these top-level APIs are available on the `React` global. If you use ES6 with npm, you can write `import React from 'react'`. If you use ES5 with npm, you can write `var React = require('react')`.

Overview

Components

React components let you split the UI into independent, reusable pieces, and think about each piece in isolation. React components can be defined by subclassing `React.Component` or `React.PureComponent`.

- `React.Component`
- `React.PureComponent`

If you don't use ES6 classes, you may use the `create-react-class` module instead. See [Using React without ES6](#) for more information.

React components can also be defined as functions which can be wrapped:

- `React.memo`

Creating React Elements

We recommend [using JSX](#) to describe what your UI should look like. Each JSX element is just syntactic sugar for calling `React.createElement()`. You will not typically invoke the following methods directly if you are using JSX.

- `createElement()`
- `createFactory()`

See [Using React without JSX](#) for more information.

Transforming Elements

`React` provides several APIs for manipulating elements:

- `cloneElement()`
- `isValidElement()`
- `React.Children`

Fragments

`React` also provides a component for rendering multiple elements without a wrapper.

- `React.Fragment`

Refs

- `React.createRef`
- `React.forwardRef`

Suspense

Suspense lets components "wait" for something before rendering. Today, *Suspense* only supports one use case: [loading components dynamically with `React.lazy`](#). In the future, it will support other use cases like data fetching.

- `React.lazy`
- `React.Suspense`

Transitions

Transitions are a new concurrent feature introduced in React 18. They allow you to mark updates as transitions, which tells React that they can be interrupted and avoid going back to *Suspense* fallbacks for already visible content.

- `React.startTransition`
- `React.useTransition`

Hooks

Hooks are a new addition in React 16.8. They let you use state and other React features without writing a class. Hooks have a [dedicated docs section](#) and a separate API reference:

- [Basic Hooks](#)
 - `useState`
 - `useEffect`
 - `useContext`
 - [Additional Hooks](#)
 - `useReducer`
 - `useCallback`
 - `useMemo`
 - `useRef`
 - `useImperativeHandle`
 - `useLayoutEffect`
 - `useDebugValue`
 - `useDeferredValue`
 - `useTransition`
 - `useId`
 - [Library Hooks](#)
 - `useSyncExternalStore`
 - `useInsertionEffect`
-

Reference

`React.Component` **{#reactcomponent}**

`React.Component` is the base class for React components when they are defined using [ES6 classes](#):

```
class Greeting extends React.Component {
  render() {
    return <h1>Hello, {this.props.name}</h1>;
  }
}
```

See the [React.Component API Reference](#) for a list of methods and properties related to the base `React.Component` class.

`React.PureComponent` **{#reactpurecomponent}**

`React.PureComponent` is similar to `React.Component`. The difference between them is that `React.Component` doesn't implement `shouldComponentUpdate()`, but `React.PureComponent` implements it with a shallow prop and state comparison.

If your React component's `render()` function renders the same result given the same props and state, you can use `React.PureComponent` for a performance boost in some cases.

Note

`React.PureComponent`'s `shouldComponentUpdate()` only shallowly compares the objects. If these contain complex data structures, it may produce false-negatives for deeper differences. Only extend `PureComponent` when you expect to have simple props and state, or use `forceUpdate()` when you know deep data structures have changed. Or, consider using [immutable objects](#) to facilitate fast comparisons of nested data.

Furthermore, `React.PureComponent`'s `shouldComponentUpdate()` skips prop updates for the whole component subtree. Make sure all the children components are also "pure".

`React.memo` **{#reactmemo}**

```
const MyComponent = React.memo(function MyComponent(props) {
  /* render using props */
});
```

`React.memo` is a [higher order component](#).

If your component renders the same result given the same props, you can wrap it in a call to `React.memo` for a performance boost in some cases by memoizing the result. This means that React will skip rendering the component, and reuse the last rendered result.

`React.memo` only checks for prop changes. If your function component wrapped in `React.memo` has a `useState`, `useReducer` or `useContext` Hook in its implementation, it will still rerender when state or context change.

By default it will only shallowly compare complex objects in the props object. If you want control over the comparison, you can also provide a custom comparison function as the second argument.

```
function MyComponent(props) {
  /* render using props */
}

function areEqual(prevProps, nextProps) {
  /*
    return true if passing nextProps to render would return
    the same result as passing prevProps to render,
    otherwise return false
  */
}

export default React.memo(MyComponent, areEqual);
```

This method only exists as a **performance optimization**. Do not rely on it to "prevent" a render, as this can lead to bugs.

Note

Unlike the `shouldComponentUpdate()` method on class components, the `areEqual` function returns `true` if the props are equal and `false` if the props are not equal. This is the inverse from `shouldComponentUpdate`.

createElement() [\[#createelement\]](#)

```
React.createElement(
  type,
  [props],
  [...children]
)
```

Create and return a new `React element` of the given type. The type argument can be either a tag name string (such as `'div'` or `'span'`), a `React component` type (a class or a function), or a `React fragment` type.

Code written with `JSX` will be converted to use `React.createElement()`. You will not typically invoke `React.createElement()` directly if you are using JSX. See [React Without JSX](#) to learn more.

cloneElement() [\[#cloneelement\]](#)

```
React.cloneElement(
  element,
  [config],
  [...children]
)
```

Clone and return a new React element using `element` as the starting point. `config` should contain all new props, `key`, or `ref`. The resulting element will have the original element's props with the new props merged in shallowly. New children will replace existing children. `key` and `ref` from the original element will be preserved if no `key` and `ref` present in the `config`.

`React.cloneElement()` is almost equivalent to:

```
<element.type {...element.props} {...props}>{children}</element.type>
```

However, it also preserves `refs`. This means that if you get a child with a `ref` on it, you won't accidentally steal it from your ancestor. You will get the same `ref` attached to your new element. The new `ref` or `key` will replace old ones if present.

This API was introduced as a replacement of the deprecated `React.addons.cloneWithProps()`.

createFactory() [\[#createfactory\]](#)

```
React.createFactory(type)
```

Return a function that produces React elements of a given type. Like `React.createElement()`, the type argument can be either a tag name string (such as `'div'` or `'span'`), a `React component` type (a class or a function), or a `React fragment` type.

This helper is considered legacy, and we encourage you to either use JSX or use `React.createElement()` directly instead.

You will not typically invoke `React.createFactory()` directly if you are using JSX. See [React Without JSX](#) to learn more.

isValidElement() [\[#isvalidelement\]](#)

```
React.isValidElement(object)
```

Verifies the object is a React element. Returns `true` or `false`.

React.Children [\[#reactchildren\]](#)

`React.Children` provides utilities for dealing with the `this.props.children` opaque data structure.

React.Children.map {#reactchildrenmap}

```
React.Children.map(children, function[(thisArg)])
```

Invokes a function on every immediate child contained within `children` with `this` set to `thisArg`. If `children` is an array it will be traversed and the function will be called for each child in the array. If `children` is `null` or `undefined`, this method will return `null` or `undefined` rather than an array.

Note

If `children` is a `Fragment` it will be treated as a single child and not traversed.

React.Children.forEach {#reactchildrenforeach}

```
React.Children.forEach(children, function[(thisArg)])
```

Like `React.Children.map()` but does not return an array.

React.Children.count {#reactchildrencount}

```
React.Children.count(children)
```

Returns the total number of components in `children`, equal to the number of times that a callback passed to `map` or `forEach` would be invoked.

React.Children.only {#reactchildrenonly}

```
React.Children.only(children)
```

Verifies that `children` has only one child (a React element) and returns it. Otherwise this method throws an error.

Note:

`React.Children.only()` does not accept the return value of `React.Children.map()` because it is an array rather than a React element.

React.Children.toArray {#reactchildrentoarray}

```
React.Children.toArray(children)
```

Returns the `children` opaque data structure as a flat array with keys assigned to each child. Useful if you want to manipulate collections of children in your render methods, especially if you want to reorder or slice `this.props.children` before passing it down.

Note:

`React.Children.toArray()` changes keys to preserve the semantics of nested arrays when flattening lists of children. That is, `toArray` prefixes each key in the returned array so that each element's key is scoped to the input array containing it.

React.Fragment `{#reactfragment}`

The `React.Fragment` component lets you return multiple elements in a `render()` method without creating an additional DOM element:

```
render() {
  return (
    <React.Fragment>
      Some text.
      <h2>A heading</h2>
    </React.Fragment>
  );
}
```

You can also use it with the shorthand `<></>` syntax. For more information, see [React v16.2.0: Improved Support for Fragments](#).

React.createRef `{#reactcreateref}`

`React.createRef` creates a `ref` that can be attached to React elements via the `ref` attribute. [embed:16-3-release-blog-post/create-ref-example.js](#)

React.forwardRef `{#reactforwardref}`

`React.forwardRef` creates a React component that forwards the `ref` attribute it receives to another component below in the tree. This technique is not very common but is particularly useful in two scenarios:

- [Forwarding refs to DOM components](#)
- [Forwarding refs in higher-order-components](#)

`React.forwardRef` accepts a rendering function as an argument. React will call this function with `props` and `ref` as two arguments. This function should return a React node.

[embed:reference-react-forward-ref.js](#)

In the above example, React passes a `ref` given to `<FancyButton ref={ref}>` element as a second argument to the rendering function inside the `React.forwardRef` call. This rendering function passes the `ref` to the `<button ref={ref}>` element.

As a result, after React attaches the `ref`, `ref.current` will point directly to the `<button>` DOM element instance.

For more information, see [forwarding refs](#).

React.lazy {#reactlazy}

`React.lazy()` lets you define a component that is loaded dynamically. This helps reduce the bundle size to delay loading components that aren't used during the initial render.

You can learn how to use it from our [code splitting documentation](#). You might also want to check out [this article](#) explaining how to use it in more detail.

```
// This component is loaded dynamically
const SomeComponent = React.lazy(() => import('./SomeComponent'));
```

Note that rendering `lazy` components requires that there's a `<React.Suspense>` component higher in the rendering tree. This is how you specify a loading indicator.

React.Suspense {#reactsuspense}

`React.Suspense` lets you specify the loading indicator in case some components in the tree below it are not yet ready to render. In the future we plan to let `Suspense` handle more scenarios such as data fetching. You can read about this in [our roadmap](#).

Today, lazy loading components is the **only** use case supported by `<React.Suspense>`:

```
// This component is loaded dynamically
const OtherComponent = React.lazy(() => import('./OtherComponent'));

function MyComponent() {
  return (
    // Displays <Spinner> until OtherComponent loads
    <React.Suspense fallback={<Spinner />}>
      <div>
        <OtherComponent />
      </div>
    </React.Suspense>
  );
}
```

It is documented in our [code splitting guide](#). Note that `lazy` components can be deep inside the `Suspense` tree -- it doesn't have to wrap every one of them. The best practice is to place `<Suspense>` where you want to see a loading indicator, but to use `lazy()` wherever you want to do code splitting.

Note

For content that is already shown to the user, switching back to a loading indicator can be disorienting. It is sometimes better to show the "old" UI while the new UI is being prepared. To do this, you can use the new transition APIs `startTransition` and `useTransition` to mark updates as transitions and avoid unexpected fallbacks.

React.Suspense in Server Side Rendering

During server side rendering Suspense Boundaries allow you to flush your application in smaller chunks by suspending. When a component suspends we schedule a low priority task to render the closest Suspense boundary's fallback. If the component unsuspends before we flush the fallback then we send down the actual content and throw away the fallback.

React.Suspense during hydration

Suspense boundaries depend on their parent boundaries being hydrated before they can hydrate, but they can hydrate independently from sibling boundaries. Events on a boundary before its hydrated will cause the boundary to hydrate at a higher priority than neighboring boundaries. [Read more](#)

React.startTransition {#starttransition}

React.startTransition(callback)

`React.startTransition` lets you mark updates inside the provided callback as transitions. This method is designed to be used when `React.useTransition` is not available.

Note:

Updates in a transition yield to more urgent updates such as clicks.

Updates in a transition will not show a fallback for re-suspended content, allowing the user to continue interacting while rendering the update.

`React.startTransition` does not provide an `isPending` flag. To track the pending status of a transition see `React.useTransition`.

[Go to TOC](#)

מרנדר טסטים

יבוא

```
import TestRenderer from 'react-test-renderer'; // ES6
const TestRenderer = require('react-test-renderer'); // ES5 with npm
```

סקירה כללית

טהורם, ללא תלות JavaScript לאובייקטיבי React שנitin להשתמש בו כדי לעבוד קומפוננטות React חביבה זו מספקת מרנדר או בסביבת מוביל טבعت DOM-ב-

בקלות שמרונדרת (DOM הדומה לעצ) בעיקרו של דבר, חביבה זו מאפשרת לנו לתפוס צלום של היררכיה תצוגת הפלטפורמה React DOM או קומפוננטת [jsdom](#) על ידי

דוגמה:

```
import TestRenderer from 'react-test-renderer';

function Link(props) {
  return <a href={props.page}>{props.children}</a>;
}

const testRenderer = TestRenderer.create(
  <Link page="https://www.facebook.com/">Facebook</Link>
);

console.log(testRenderer.toJSON());
// { type: 'a',
//   props: { href: 'https://www.facebook.com/' },
//   children: [ 'Facebook' ] }
```

לקובץ ולבדק בתוך הטסטים JSON-כדי לשמר באופן אוטומטי עותק של עץ ה Jest ניתן להשתמש בתוכנת בדיקות צלום של שול שהוא לא השתנה: [למיע נסף ערך](#).

אתם יכולים גם לעבור על הפלט כדי למצוא צמתים ספציפיים ולבצע בדיקות לגבייהם.

```
import TestRenderer from 'react-test-renderer';

function MyComponent() {
  return (
    <div>
      <SubComponent foo="bar" />
      <p className="my">שלום</p>
    </div>
  )
}

function SubComponent() {
  return (
    <p className="sub">תחתית</p>
  );
}
```

```
const testRenderer = TestRenderer.create(<MyComponent />);
const testInstance = testRenderer.root;

expect(testInstance.findByType(SubComponent).props.foo).toBe('bar');
expect(testInstance.findByProps({className: "sub"}).children).toEqual(['תפקיד'])
```

TestRenderer

- `TestRenderer.create()`
- `TestRenderer.act()`

מופע TestRenderer

- `testRenderer.toJSON()`
- `testRenderer.toTree()`
- `testRenderer.update()`
- `testRenderer.unmount()`
- `testRenderer.getInstance()`
- `testRenderer.root`

TestInstance

- `testInstance.find()`
- `testInstance.findByType()`
- `testInstance.findByProps()`
- `testInstance.findAll()`
- `testInstance.findAllByType()`
- `testInstance.findAllByProps()`
- `testInstance.instance`
- `testInstance.type`
- `testInstance.props`
- `testInstance.parent`
- `testInstance.children`

סיכום

`TestRenderer.create()` {#testrenderercreate}

```
TestRenderer.create(element, options);
```

האמיתי, אך הוא עדין מレンדר את DOM-שהועברה. הוא לא משתמש ב-React-עם קומפוננטות ה-`TestRenderer` צור מופע עז הקומפוננטה ליזכרו באופן מלא, כך שתוכלו לבצע השוואות לגביו. מחריר [מופיע](#).

`TestRenderer.act()` {#testrendereract}

```
TestRenderer.act(callback);
```

Similar to the `act()` helper from `react-dom/test-utils`, `TestRenderer.act` prepares a component for assertions. Use this version of `act()` to wrap calls to `TestRenderer.create` and `testRenderer.update`.

```

import {create, act} from 'react-test-renderer';
import App from './app.js'; // The component being tested

// render the component
let root;
act(() => {
  root = create(<App value={1}>)
});

// make assertions on root
expect(root.toJSON()).toMatchSnapshot();

// update with some different props
act(() => {
  root.update(<App value={2}>);
})

// make assertions on root
expect(root.toJSON()).toMatchSnapshot();

```

`testRenderer.toJSON()` **#testrenderertojson**

`testRenderer.toJSON()`

ואת `<div>` או `<View>` החזר אובייקט המייצג את העץ שرونדר. עז זה מכיל רק את הצמתים הספרטiviים לפלטפורמה כגון `shallow`, אך איןנו מכיל קומפוננטות שנכתבו על ידי המשתמש. זה שימושי עבור **בדיקות תמונה מצב**-ה.

`testRenderer.toTree()` **#testrenderertotree**

`testRenderer.toTree()`

והוא כולל את הקומפוננטות, `toJSON()` החזר אובייקט המייצג את העץ שرونדר. הייצוג מפורט יותר מזה שניינו על ידי שנכתבו על ידי המשתמש. אתם כנראה לא תצרכו להשתמש במתודה זו, אלא אם אתם כותבים ספריות בדיקות משלכם. הטענה הפועלת מעל מרנדר הטסטים.

`testRenderer.update()` **#testrendererupdate**

`testRenderer.update(element)`

בשורש. אם לאלמנט החדש יש את העץ בתוך הזיכרון עם אלמנט שורש חדש. פעולה זו מדמה עדכון מהחדש לעץ החדש אותו סוג והואתו מפתח כמו האלמנט הקודם, העץ יעדכן; אחרת, הוא יעשה

`testRenderer.unmount()` **#testrendererunmount**

`testRenderer.unmount()`

בטל את טיענת העץ בזיכרון, תוך הפעלת אירופי מהזור החיים המתאים.

`testRenderer.getInstance()` **#testrenderergetInstance**

`testRenderer.getInstance()`

החזר את המופיע התואם לאלמנט השורש, אם זמין. פעולה זו לא תעבור אם אלמנט השורש הוא קומפוננטת פונקציה מכיוון שאין להם מופעים.

`testRenderer.root {#testrendererroot}`

`testRenderer.root`

שהינו שימושי להבנת השוואות לגבי צמתים ספציפיים בעץ. אתם יכולים "test instance" מוחזר את אובייקט השורש אחרים עמוק יותר בעץ "test instances" בו כדי למצאו.

`testInstance.find() {#testinstancefind}`

`testInstance.find(test)`

אינה מוחירה `test(testInstance)` true אם `test(testInstance)` מוחזרת ממצא מופיע לצא של טסט בודד שבעורו עברו מופיע בדיקה אחד בדיק, תזדק שגיאה `true` מוחירה.

`testInstance.findByType() {#testinstancefindbytype}`

`testInstance.findByType(type)`

שסוף, תזדק שגיאה `type`-שסוף. אם אין בדיק מופיע מבחן אחד עם ה `type`-ממצא מופיע לצא של טסט בודד עם ה

`testInstance.findByProps() {#testinstancefindbyprops}`

`testInstance.findByProps(props)`

שסוף, תזדק שגיאה `props`-שסוף. אם אין בדיק מופיע מבחן אחד עם ה `props`-ממצא מופיע לצא של טסט בודד עם ה

`testInstance.findAll() {#testinstancefindall}`

`testInstance.findAll(test)`

מוחירה `test(testInstance)` true. מוצא את כל צאצאי מופיע הטסט שבעורו.

`testInstance.findAllByType() {#testinstancefindallbytype}`

`testInstance.findAllByType(type)`

שסוף - מוצא את כל צאצאי מופיע הטסט עם ה `type`.

`testInstance.findAllByProps() {#testinstancefindallbyprops}`

`testInstance.findAllByProps(props)`

שסוף - מוצא את כל צאצאי מופיע הטסט עם ה `props`.

`testInstance.instance {#testinstanceinstance}`

`testInstance.instance`

מודול הקומפוננטה התואמת למופע טסט זה. זמין רק עבור קומפוננטות מחלוקת, מכיוון שלקומפוננטות פונקציה אין מופעים. בתוך הקומפוננטה שניתנה `this` כמו כן, מתאים את הערך של

`testInstance.type` {#testinstancetype}

`testInstance.type`

סוג הקומפוננטה התואם למופע טסט זה. לדוגמה, לקומפוננטת `<Button>` יהיה ערך סוג `Button`.

`testInstance.props` {#testinstanceprops}

`testInstance.props`

בתוך `{'props'}` המתאימים למופע טסט זה. לדוגמה, לקומפוננטה props. `<Button size="small" />` יש `{size: 'small'}`

`testInstance.parent` {#testinstanceparent}

`testInstance.parent`

מודול האב של מופע טסט זה.

`testInstance.children` {#testinstancechildren}

`testInstance.children`

מודשי ילדי הטסטים של מופע טסט זה.

РЕЗЮМЕ

המאפשרת שימוש בהפניות `option` בטור `createNodeMock` - `TestRenderer.create` ניתן להעביר את הפונקציה מקבלת את האלמנט הנוכחי וammerה להחזיר אובייקט הפניה מדומה. אפ硕ות מודומות מותאמים אישית זו שימושית בעת בדיקת קומפוננטה המסתמכת על הפניות.

```
import TestRenderer from 'react-test-renderer';

class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.input = null;
  }
  componentDidMount() {
    this.input.focus();
  }
  render() {
    return <input type="text" ref={el => this.input = el} />
  }
}

let focused = false;
TestRenderer.create(
  <MyComponent />,
  {
```

```
createNodeMock: (element) => {
  if (element.type === 'input') {
    // mock a focus function
    return {
      focus: () => {
        focused = true;
      }
    };
  }
  return null;
);
expect(focused).toBe(true);
```

Refs and the DOM

Refs provide a way to access DOM nodes or React elements created in the render method.

In the typical React dataflow, [props](#) are the only way that parent components interact with their children. To modify a child, you re-render it with new props. However, there are a few cases where you need to imperatively modify a child outside of the typical dataflow. The child to be modified could be an instance of a React component, or it could be a DOM element. For both of these cases, React provides an escape hatch.

When to Use Refs

There are a few good use cases for refs:

- Managing focus, text selection, or media playback.
- Triggering imperative animations.
- Integrating with third-party DOM libraries.

Avoid using refs for anything that can be done declaratively.

For example, instead of exposing `open()` and `close()` methods on a `Dialog` component, pass an `isOpen` prop to it.

Don't Overuse Refs

Your first inclination may be to use refs to "make things happen" in your app. If this is the case, take a moment and think more critically about where state should be owned in the component hierarchy. Often, it becomes clear that the proper place to "own" that state is at a higher level in the hierarchy. See the [Lifting State Up](#) guide for examples of this.

Note

The examples below have been updated to use the `React.createRef()` API introduced in React 16.3. If you are using an earlier release of React, we recommend using [callback refs](#) instead.

Creating Refs

Refs are created using `React.createRef()` and attached to React elements via the `ref` attribute. Refs are commonly assigned to an instance property when a component is constructed so they can be referenced throughout the component.

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.myRef = React.createRef();
  }
  render() {
```

```

        return <div ref={this.myRef} />;
    }
}
```

Accessing Refs

When a `ref` is passed to an element in `render`, a reference to the node becomes accessible at the `current` attribute of the ref.

```
const node = this.myRef.current;
```

The value of the ref differs depending on the type of the node:

- When the `ref` attribute is used on an HTML element, the `ref` created in the constructor with `React.createRef()` receives the underlying DOM element as its `current` property.
- When the `ref` attribute is used on a custom class component, the `ref` object receives the mounted instance of the component as its `current`.
- **You may not use the `ref` attribute on function components** because they don't have instances.

The examples below demonstrate the differences.

Adding a Ref to a DOM Element

This code uses a `ref` to store a reference to a DOM node:

```

class CustomTextInput extends React.Component {
  constructor(props) {
    super(props);
    // create a ref to store the textInput DOM element
    this.textInput = React.createRef();
    this.focusTextInput = this.focusTextInput.bind(this);
  }

  focusTextInput() {
    // Explicitly focus the text input using the raw DOM API
    // Note: we're accessing "current" to get the DOM node
    this.textInput.current.focus();
  }

  render() {
    // tell React that we want to associate the <input> ref
    // with the `textInput` that we created in the constructor
    return (
      <div>
        <input
          type="text"
          ref={this.textInput} />
        <input
          type="button"
          value="Focus the text input"
          onClick={this.focusTextInput}
        />
      </div>
    );
  }
}
```

React will assign the `current` property with the DOM element when the component mounts, and assign it back to `null` when it unmounts. `ref` updates happen before `componentDidMount` or `componentDidUpdate` lifecycle methods.

Adding a Ref to a Class Component

If we wanted to wrap the `CustomTextInput` above to simulate it being clicked immediately after mounting, we could use a ref to get access to the custom input and call its `focusTextInput` method manually:

```
class AutoFocusTextInput extends React.Component {
  constructor(props) {
    super(props);
    this.textInput = React.createRef();
  }

  componentDidMount() {
    this.textInput.current.focusTextInput();
  }

  render() {
    return (
      <CustomTextInput ref={this.textInput} />
    );
  }
}
```

Note that this only works if `CustomTextInput` is declared as a class:

```
class CustomTextInput extends React.Component {
  // ...
}
```

Refs and Function Components

By default, **you may not use the `ref` attribute on function components** because they don't have instances:

```
function MyFunctionComponent() {
  return <input />;
}

class Parent extends React.Component {
  constructor(props) {
    super(props);
    this.textInput = React.createRef();
  }

  render() {
    // This will *not* work!
    return (
      <MyFunctionComponent ref={this.textInput} />
    );
  }
}
```

If you want to allow people to take a `ref` to your function component, you can use `forwardRef` (possibly in conjunction with `useImperativeHandle`), or you can convert the component to a class.

You can, however, **use the `ref` attribute inside a function component** as long as you refer to a DOM element or a class component:

```
function CustomTextInput(props) {
  //textInput must be declared here so the ref can refer to it
  const textInput = useRef(null);

  function handleClick() {
    textInput.current.focus();
  }

  return (
    <div>
      <input
        type="text"
        ref={textInput} />
      <input
        type="button"
        value="Focus the text input"
        onClick={handleClick}
      />
    </div>
  );
}
```

Exposing DOM Refs to Parent Components

In rare cases, you might want to have access to a child's DOM node from a parent component. This is generally not recommended because it breaks component encapsulation, but it can occasionally be useful for triggering focus or measuring the size or position of a child DOM node.

While you could [add a ref to the child component](#), this is not an ideal solution, as you would only get a component instance rather than a DOM node. Additionally, this wouldn't work with function components.

If you use React 16.3 or higher, we recommend to use [ref forwarding](#) for these cases. **Ref forwarding lets components opt into exposing any child component's ref as their own**. You can find a detailed example of how to expose a child's DOM node to a parent component [in the ref forwarding documentation](#).

If you use React 16.2 or lower, or if you need more flexibility than provided by ref forwarding, you can use [this alternative approach](#) and explicitly pass a ref as a differently named prop.

When possible, we advise against exposing DOM nodes, but it can be a useful escape hatch. Note that this approach requires you to add some code to the child component. If you have absolutely no control over the child component implementation, your last option is to use `findDOMNode()`, but it is discouraged and deprecated in `StrictMode`.

Callback Refs

React also supports another way to set refs called "callback refs", which gives more fine-grain control over when refs are set and unset.

Instead of passing a `ref` attribute created by `createRef()`, you pass a function. The function receives the React component instance or HTML DOM element as its argument, which can be stored and accessed elsewhere.

The example below implements a common pattern: using the `ref` callback to store a reference to a DOM node in an instance property.

```
class CustomTextInput extends React.Component {
  constructor(props) {
    super(props);

    this.textInput = null;

    this.setTextInputRef = element => {
      this.textInput = element;
    };

    this.focusTextInput = () => {
      // Focus the text input using the raw DOM API
      if (this.textInput) this.textInput.focus();
    };
  }

  componentDidMount() {
    // autofocus the input on mount
    this.focusTextInput();
  }

  render() {
    // Use the `ref` callback to store a reference to the text input DOM
    // element in an instance field (for example, this.textInput).
    return (
      <div>
        <input
          type="text"
          ref={this.setTextInputRef}
        />
        <input
          type="button"
          value="Focus the text input"
          onClick={this.focusTextInput}
        />
      </div>
    );
  }
}
```

React will call the `ref` callback with the DOM element when the component mounts, and call it with `null` when it unmounts. Refs are guaranteed to be up-to-date before `componentDidMount` or `componentDidUpdate` fires.

You can pass callback refs between components like you can with object refs that were created with `React.createRef()`.

```
function CustomTextInput(props) {
  return (
    <div>
      <input ref={props.inputRef} />
```

```
        </div>
    );
}

class Parent extends React.Component {
  render() {
    return (
      <CustomTextInput
        inputRef={el => this.inputElement = el}
      />
    );
  }
}
```

In the example above, `Parent` passes its ref callback as an `inputRef` prop to the `CustomTextInput`, and the `CustomTextInput` passes the same function as a special `ref` attribute to the `<input>`. As a result, `this.inputElement` in `Parent` will be set to the DOM node corresponding to the `<input>` element in the `CustomTextInput`.

Legacy API: String Refs

If you worked with React before, you might be familiar with an older API where the `ref` attribute is a string, like `"textInput"`, and the DOM node is accessed as `this.refs.textInput`. We advise against it because string refs have [some issues](#), are considered legacy, and **are likely to be removed in one of the future releases**.

Note

If you're currently using `this.refs.textInput` to access refs, we recommend using either the [callback pattern](#) or the [createRef API](#) instead.

Caveats with callback refs

If the `ref` callback is defined as an inline function, it will get called twice during updates, first with `null` and then again with the DOM element. This is because a new instance of the function is created with each render, so React needs to clear the old ref and set up the new one. You can avoid this by defining the `ref` callback as a bound method on the class, but note that it shouldn't matter in most cases.

[Go to TOC](#)

Release Channels

React relies on a thriving open source community to file bug reports, open pull requests, and [submit RFCs](#). To encourage feedback we sometimes share special builds of React that include unreleased features.

This document will be most relevant to developers who work on frameworks, libraries, or developer tooling. Developers who use React primarily to build user-facing applications should not need to worry about our prerelease channels.

Each of React's release channels is designed for a distinct use case:

- **Latest** is for stable, semver React releases. It's what you get when you install React from npm. This is the channel you're already using today. **Use this for all user-facing React applications.**
- **Next** tracks the main branch of the React source code repository. Think of these as release candidates for the next minor semver release. Use this for integration testing between React and third party projects.
- **Experimental** includes experimental APIs and features that aren't available in the stable releases. These also track the main branch, but with additional feature flags turned on. Use this to try out upcoming features before they are released.

All releases are published to npm, but only Latest uses [semantic versioning](#). Prereleases (those in the Next and Experimental channels) have versions generated from a hash of their contents and the commit date, e.g. `0.0.0-68053d940-20210623` for Next and `0.0.0-experimental-68053d940-20210623` for Experimental.

The only officially supported release channel for user-facing applications is Latest. Next and Experimental releases are provided for testing purposes only, and we provide no guarantees that behavior won't change between releases. They do not follow the semver protocol that we use for releases from Latest.

By publishing prereleases to the same registry that we use for stable releases, we are able to take advantage of the many tools that support the npm workflow, like [unpkg](#) and [CodeSandbox](#).

Latest Channel

Latest is the channel used for stable React releases. It corresponds to the `latest` tag on npm. It is the recommended channel for all React apps that are shipped to real users.

If you're not sure which channel you should use, it's Latest. If you're a React developer, this is what you're already using.

You can expect updates to Latest to be extremely stable. Versions follow the semantic versioning scheme. Learn more about our commitment to stability and incremental migration in our [versioning policy](#).

Next Channel

The Next channel is a prerelease channel that tracks the main branch of the React repository. We use prereleases in the Next channel as release candidates for the Latest channel. You can think of Next as a superset of Latest that is updated more frequently.

The degree of change between the most recent Next release and the most recent Latest release is approximately the same as you would find between two minor semver releases. However, **the Next channel does not conform to semantic versioning**. You should expect occasional breaking changes between successive releases in the Next channel.

Do not use prereleases in user-facing applications.

Releases in Next are published with the `next` tag on npm. Versions are generated from a hash of the build's contents and the commit date, e.g. `0.0.0-68053d940-20210623`.

Using the Next Channel for Integration Testing

The Next channel is designed to support integration testing between React and other projects.

All changes to React go through extensive internal testing before they are released to the public. However, there are a myriad of environments and configurations used throughout the React ecosystem, and it's not possible for us to test against every single one.

If you're the author of a third party React framework, library, developer tool, or similar infrastructure-type project, you can help us keep React stable for your users and the entire React community by periodically running your test suite against the most recent changes. If you're interested, follow these steps:

- Set up a cron job using your preferred continuous integration platform. Cron jobs are supported by both [CircleCI](#) and [Travis CI](#).
- In the cron job, update your React packages to the most recent React release in the Next channel, using `next` tag on npm. Using the npm cli:

```
npm update react@next react-dom@next
```

Or yarn:

```
yarn upgrade react@next react-dom@next
```

- Run your test suite against the updated packages.
- If everything passes, great! You can expect that your project will work with the next minor React release.
- If something breaks unexpectedly, please let us know by [filing an issue](#).

A project that uses this workflow is Next.js. (No pun intended! Seriously!) You can refer to their [CircleCI configuration](#) as an example.

Experimental Channel

Like Next, the Experimental channel is a prerelease channel that tracks the main branch of the React repository. Unlike Next, Experimental releases include additional features and APIs that are not ready for wider release.

Usually, an update to Next is accompanied by a corresponding update to Experimental. They are based on the same source revision, but are built using a different set of feature flags.

Experimental releases may be significantly different than releases to Next and Latest. **Do not use Experimental releases in user-facing applications.** You should expect frequent breaking changes between releases in the Experimental channel.

Releases in Experimental are published with the `experimental` tag on npm. Versions are generated from a hash of the build's contents and the commit date, e.g. `0.0.0-experimental-68053d940-20210623`.

What Goes Into an Experimental Release?

Experimental features are ones that are not ready to be released to the wider public, and may change drastically before they are finalized. Some experiments may never be finalized -- the reason we have experiments is to test the viability of proposed changes.

For example, if the Experimental channel had existed when we announced Hooks, we would have released Hooks to the Experimental channel weeks before they were available in Latest.

You may find it valuable to run integration tests against Experimental. This is up to you. However, be advised that Experimental is even less stable than Next. **We do not guarantee any stability between Experimental releases.**

How Can I Learn More About Experimental Features?

Experimental features may or may not be documented. Usually, experiments aren't documented until they are close to shipping in Next or Latest.

If a feature is not documented, they may be accompanied by an [RFC](#).

We will post to the [React blog](#) when we're ready to announce new experiments, but that doesn't mean we will publicize every experiment.

You can always refer to our public GitHub repository's [history](#) for a comprehensive list of changes.

Render Props

The term "render prop" refers to a technique for sharing code between React components using a prop whose value is a function.

A component with a render prop takes a function that returns a React element and calls it instead of implementing its own render logic.

```
<DataProvider render={data => (
  <h1>Hello {data.target}</h1>
)}>
```

Libraries that use render props include [React Router](#), [Downshift](#) and [Formik](#).

In this document, we'll discuss why render props are useful, and how to write your own.

Use Render Props for Cross-Cutting Concerns

Components are the primary unit of code reuse in React, but it's not always obvious how to share the state or behavior that one component encapsulates to other components that need that same state.

For example, the following component tracks the mouse position in a web app:

```
class MouseTracker extends React.Component {
  constructor(props) {
    super(props);
    this.handleMouseMove = this.handleMouseMove.bind(this);
    this.state = { x: 0, y: 0 };
  }

  handleMouseMove(event) {
    this.setState({
      x: event.clientX,
      y: event.clientY
    });
  }

  render() {
    return (
      <div style={{ height: '100vh' }} onMouseMove={this.handleMouseMove}>
        <h1>Move the mouse around!</h1>
        <p>The current mouse position is ({this.state.x}, {this.state.y})</p>
      </div>
    );
  }
}
```

As the cursor moves around the screen, the component displays its (x, y) coordinates in a `<p>`.

Now the question is: How can we reuse this behavior in another component? In other words, if another component needs to know about the cursor position, can we encapsulate that behavior so that we can easily share it with that component?

Since components are the basic unit of code reuse in React, let's try refactoring the code a bit to use a `<Mouse>` component that encapsulates the behavior we need to reuse elsewhere.

```
// The <Mouse> component encapsulates the behavior we need...
class Mouse extends React.Component {
  constructor(props) {
    super(props);
    this.handleMouseMove = this.handleMouseMove.bind(this);
    this.state = { x: 0, y: 0 };
  }

  handleMouseMove(event) {
    this.setState({
      x: event.clientX,
      y: event.clientY
    });
  }

  render() {
    return (
      <div style={{ height: '100vh' }} onMouseMove={this.handleMouseMove}>
        {/* ...but how do we render something other than a <p>? */}
        <p>The current mouse position is ({this.state.x}, {this.state.y})</p>
      </div>
    );
  }
}

class MouseTracker extends React.Component {
  render() {
    return (
      <>
        <h1>Move the mouse around!</h1>
        <Mouse />
      </>
    );
  }
}
```

Now the `<Mouse>` component encapsulates all behavior associated with listening for `mousemove` events and storing the (x, y) position of the cursor, but it's not yet truly reusable.

For example, let's say we have a `<Cat>` component that renders the image of a cat chasing the mouse around the screen. We might use a `<Cat mouse={{ x, y }}>` prop to tell the component the coordinates of the mouse so it knows where to position the image on the screen.

As a first pass, you might try rendering the `<Cat>` inside `<Mouse>`'s `render` method, like this:

```
class Cat extends React.Component {
  render() {
    const mouse = this.props.mouse;
    return (
      
    );
  }
}
```

```

class MouseWithCat extends React.Component {
  constructor(props) {
    super(props);
    this.handleMouseMove = this.handleMouseMove.bind(this);
    this.state = { x: 0, y: 0 };
  }

  handleMouseMove(event) {
    this.setState({
      x: event.clientX,
      y: event.clientY
    });
  }

  render() {
    return (
      <div style={{ height: '100vh' }} onMouseMove={this.handleMouseMove}>
        /*
          We could just swap out the <p> for a <Cat> here ... but then
          we would need to create a separate <MouseWithSomethingElse>
          component every time we need to use it, so <MouseWithCat>
          isn't really reusable yet.
        */
        <Cat mouse={this.state} />
      </div>
    );
  }
}

class MouseTracker extends React.Component {
  render() {
    return (
      <div>
        <h1>Move the mouse around!</h1>
        <MouseWithCat />
      </div>
    );
  }
}

```

This approach will work for our specific use case, but we haven't achieved the objective of truly encapsulating the behavior in a reusable way. Now, every time we want the mouse position for a different use case, we have to create a new component (i.e. essentially another `<MouseWithCat>`) that renders something specifically for that use case.

Here's where the render prop comes in: Instead of hard-coding a `<Cat>` inside a `<Mouse>` component, and effectively changing its rendered output, we can provide `<Mouse>` with a function prop that it uses to dynamically determine what to render—a render prop.

```

class Cat extends React.Component {
  render() {
    const mouse = this.props.mouse;
    return (
      
    );
  }
}

```

```

class Mouse extends React.Component {
  constructor(props) {
    super(props);
    this.handleMouseMove = this.handleMouseMove.bind(this);
    this.state = { x: 0, y: 0 };
  }

  handleMouseMove(event) {
    this.setState({
      x: event.clientX,
      y: event.clientY
    });
  }

  render() {
    return (
      <div style={{ height: '100vh' }} onMouseMove={this.handleMouseMove}>
        {/* Instead of providing a static representation of what <Mouse> renders,
           use the `render` prop to dynamically determine what to render.
        */}
        {this.props.render(this.state)}
      </div>
    );
  }
}

class MouseTracker extends React.Component {
  render() {
    return (
      <div>
        <h1>Move the mouse around!</h1>
        <Mouse render={mouse => (
          <Cat mouse={mouse} />
        )}/>
      </div>
    );
  }
}

```

Now, instead of effectively cloning the `<Mouse>` component and hard-coding something else in its `render` method to solve for a specific use case, we provide a `render` prop that `<Mouse>` can use to dynamically determine what it renders.

More concretely, **a render prop is a function prop that a component uses to know what to render.**

This technique makes the behavior that we need to share extremely portable. To get that behavior, render a `<Mouse>` with a `render` prop that tells it what to render with the current (x, y) of the cursor.

One interesting thing to note about render props is that you can implement most **higher-order components** (HOC) using a regular component with a render prop. For example, if you would prefer to have a `withMouse` HOC instead of a `<Mouse>` component, you could easily create one using a regular `<Mouse>` with a render prop:

```

// If you really want a HOC for some reason, you can easily
// create one using a regular component with a render prop!
function withMouse(Component) {

```

```

return class extends React.Component {
  render() {
    return (
      <Mouse render={mouse => (
        <Component {...this.props} mouse={mouse} />
      )}>
    );
  }
}

```

So using a render prop makes it possible to use either pattern.

Using Props Other Than `render` [\[#using-props-other-than-render\]](#)

It's important to remember that just because the pattern is called "render props" you don't *have to use a prop named `render` to use this pattern*. In fact, *any prop that is a function that a component uses to know what to render is technically a "render prop"*.

Although the examples above use `render`, we could just as easily use the `children` prop!

```

<Mouse children={mouse => (
  <p>The mouse position is {mouse.x}, {mouse.y}</p>
)}>

```

And remember, the `children` prop doesn't actually need to be named in the list of "attributes" in your JSX element. Instead, you can put it directly *inside* the element!

```

<Mouse>
  {mouse => (
    <p>The mouse position is {mouse.x}, {mouse.y}</p>
  )}
</Mouse>

```

You'll see this technique used in the `react-motion` API.

Since this technique is a little unusual, you'll probably want to explicitly state that `children` should be a function in your `propTypes` when designing an API like this.

```

Mouse.propTypes = {
  children: PropTypes.func.isRequired
};

```

Caveats

Be careful when using Render Props with `React.PureComponent`

Using a render prop can negate the advantage that comes from using `React.PureComponent` if you create the function inside a `render` method. This is because the shallow prop comparison will always return `false` for new props, and each `render` in this case will generate a new value for the render prop.

For example, continuing with our `<Mouse>` component from above, if `Mouse` were to extend `React.PureComponent` instead of `React.Component`, our example would look like this:

```
class Mouse extends React.PureComponent {
  // Same implementation as above...
}

class MouseTracker extends React.Component {
  render() {
    return (
      <div>
        <h1>Move the mouse around!</h1>

        {/* 
          This is bad! The value of the `render` prop will
          be different on each render.
        */}
        <Mouse render={mouse => (
          <Cat mouse={mouse} />
        )}>
      </div>
    );
  }
}
```

In this example, each time `<MouseTracker>` renders, it generates a new function as the value of the `<Mouse render>` prop, thus negating the effect of `<Mouse>` extending `React.PureComponent` in the first place!

To get around this problem, you can sometimes define the prop as an instance method, like so:

```
class MouseTracker extends React.Component {
  // Defined as an instance method, `this.renderTheCat` always
  // refers to *same* function when we use it in render
  renderTheCat(mouse) {
    return <Cat mouse={mouse} />;
  }

  render() {
    return (
      <div>
        <h1>Move the mouse around!</h1>
        <Mouse render={this.renderTheCat} />
      </div>
    );
  }
}
```

In cases where you cannot define the prop statically (e.g. because you need to close over the component's props and/or state) `<Mouse>` should extend `React.Component` instead.

ציוויל אלמנטים

אלמנטים הם אבן הבניין הקטנה ביותר של אפליקציות ריאקט

:אלמנט מתאר מה ווציאים להציג במסך

```
const element = <h1>Hello, world</h1>;
```

Unlike browser DOM elements, React elements are plain objects, and are cheap to create. React DOM takes care of updating the DOM to match the React elements. אלמנטים של ריאקט הם DOM ללא כמו אלמנטים של React. אובייקטים פשוטים, ויצירותם אינה דורשת משבבים רבים על מנת להתאים אותם DOM מטפל בעדכון ה DOM. אובייקטים פשוטים יוצרים איננה פרטת דרישות של ריאקט לאלמנטים של ריאקט.

הערה:

ניתן להתבלבל בין הקונספט הדיעו של קומפוננטות. נציג את רעיון הקומפוננטות [פרק הבא](#). אלמנטים הם המרכיבים מהם עשוות קומפוננטות ואנחנו מעדדים לקרוא פרק זה לפני שימושיכם הלאה.

ReactDOM.createElement לтвор ה

HTML במקום כלשהו בקובץ `<div>` נניח שיש:

```
<div id="root"></div>
```

שורשי מכיוון שכל מה שהוא מכיל ינוהל על ידי DOM אנחנו מכנים זאת קודקוד React DOM.

שורשי אחד. במידה אתה משלב את ריאקט לתוך אפליקציה DOM בדרך כלל, אפליקציות הנבנות עם ריאקט הן בעלות קודקוד שורשיים DOM קיימת, אתה יכול להשתמש בכמה בלתי מוגבלת של קודחות.

שורשי, העבר אותו אל הפונקציה DOM על מנת ליצור אלמנט ריאקטי לתוך קודקוד `ReactDOM.render()` :

[codepen - נסו את זה ב](#)

בעמוד "Hello, world" דוגמא זו מזיגה.

עדכון אלמנטים שצוירו

אלמנטי ריאקט [אין מושתנים](#). במידה ויצרת אלמנט, לא ניתן לשנות את ידיו או מאפייניו. אלמנט הוא כמו פרים יחיד בסרטן: הוא מייצג את משק המשמש בנקודה מסוימת בזמן.

עם הידע שלמדנו על כה, הדרך היחידה לעדכן את משק המשמש הוא על ידי יצירתו של אלמנט חדש והעברה שלו `ReactDOM.render()`.

:בהתחשב בדוגמה השעון המתתקתק

```
embed:rendering-elements/update-rendered-element.js
```

[codepen - נסxo את זה ב](#)

בכל שנייה על ידי הפונקציה הנקראת על ידי `ReactDOM.render()` דוגמא זו קוראת ל `setInterval()`.

הערה:

פעם אחת בלבד. בפרקים הבאים נלמד איך קוד זה `ReactDOM.render()` בפועל, רוב אפליקציות ריאקט קוראות ל `state` מוכמס [לקומפוננטות בעלות](#).

אנו ממליצים לא לדלג על נושאים מכיוון שהם נבנו אחד על השני.

ריאקט מעdbo רק מה שנחוץ לעדכן

על מנת להביא אותו למצב DOM משווה את האלמנט וילדיו למצב הקודם שלו ומהיל אך ורק שינויים נדרשים ב DOM הרצוי.

אתה יוכל לאמת זאת על ידי התובנות [בדוגמא](#) באמצעות כל הדף:

Hello, world!

It is 12:26:46 PM.

```
Console Sources Network Timeline
▼<div id="root">
  ▼<div data-reactroot>
    <h1>Hello, world!</h1>
    ▼<h2>
      <!-- react-text: 4 -->
      "It is "
      <!-- /react-text -->
      <!-- react-text: 5 -->
      "12:26:46 PM"
      <!-- /react-text -->
      <!-- react-text: 6 -->
      "."
      <!-- /react-text -->
    </h2>
  </div>
</div>
```

הציג את העדכוניים הפרטניים DOM inspector

אפשרו שאנחנו יוצרים אלמנט המתאר את עץ משק המשתמש המלא בכל תקתו של השעון, רק קודקוד הטקסט שתוכנו.React DOM.

מהניסיונו שלנו, חשיבה על איך משק המשתמש צריך להראות בכל רגע נתון לעומת אותו לאורך זמן מבטל כמוות. נקבעת של באגים.

[Go to TOC](#)

State ומחזור חיים

של קומפוננטה CAN API-תוכלו למצוא את פירוט ה state בקומפוננטת `React`. ומחזור חיים בקומפוננטת `state` דף זה מציג את הקונספט של

הбиוטו על דוגמת השעון המתתקתק **מאחד מהחלקים הקודמים**. בrinדור **אלמנטים**, למדנו רק דרך אחות לעדכון ממשק המשתמש. כדי לשנות את הפלט שירונדר (`ReactDOM.render()`-אנו קוראים ל

```
const root = ReactDOM.createRoot(document.getElementById('root'));

function tick() {
  const element = (
    <div>
      <h1>Hello, world!</h1>
      <h2>It is {new Date().toLocaleTimeString()}.</h2>
    </div>
  );
  root.render(element);
}

setInterval(tick, 1000);
```

[-נסו זאת ב-CodePen](#)

לכזאת שבאמת אפשרות שימוש חזר ומוכמת (מאפשרת (שעון) `Clock`-בפרק זה נלמד כיצד להפוך את קומפוננטת השעון אנקופוליזיה). היא תקבע טימר משלה ותעדכון את עצמה בכל שנייה.

אנחנו יכולים להתחיל על ידי כימוש של תצוגת השעון:

```
const root = ReactDOM.createRoot(document.getElementById('root'));

function Clock(props) {
  return (
    <div>
      <h1>Hello, world!</h1>
      <h2>It is {props.date.toLocaleTimeString()}.</h2>
    </div>
  );
}

function tick() {
  root.render(<Clock date={new Date()} />);
}

setInterval(tick, 1000);
```

[-נסו זאת ב-CodePen](#)

מייצר טימר וمعدכון את ממשק המשתמש בכל שנייה צריכה להיות `Clock`-עם זאת, אנו מפספסים דרישת חיונית: העובדה שחלק מהשימוש של `Clock`.

יעדכון את עצמו `Clock`-באופן אידיאלי, אנחנו רוצים לכתוב את זה פעם אחת ושוב

```
root.render(<Clock />);
```

לקומפוננטה ה "state" כדי למש את זה, עלינו להוסיף `Clock`.

אבל הוא פרט ושולט במלואו על ידי הקומפוננטה,props-זהה ל

הمرة פונקציה למחלקה

למחלקה בחמישה צעדים `Clock` אתם יכולים להמיר קומפוננטת פונקציה כמו:

1. עם שם זהה, שמרחיבת את `React.Component`.
2. הוסיפו לה מתודה אחת ריקה בשם `render()`.
3. העבירו את גוף הפונקציה לתוך מתודה `render()`.
- 4.props-בגופו `this.props` עם החליפו את `render()`.
- 5.מחקו את הצהרת הפונקציה הריקה שנותרה.

```
class Clock extends React.Component {
  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.props.date.toLocaleTimeString()}</h2>
      </div>
    );
  }
}
```

[CodePen](#)-נסוי זאת ב

מוגדר כתה כמחלקה במקום כפונקציה.

נעשה DOM אל אותה צומת `<Clock />` תקרא בכל פעם שמתורחש עדכוון, אך כל עד אלו מrndרים את `render` מתודה מקומי ומתחודת מחזור state זה מהאפשר לננו לעשות שימוש בתכונות נוספות כגון `Clock`. שימוש רק במופיע אחד של המחלקה חיים.

מקומי למחלקה State הוספה

בשלושה שלבים state-ל-state מ-props מ-props-date נעביר את ה

1. `this.props.date` עם `this.state.date` במתודה `render()`:

```
class Clock extends React.Component {
  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}</h2>
      </div>
    );
  }
}
```

2. ההתחלתי - הוסיףנו בנאי מחלקה שմבצע השמה ל- `this.state`:

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }

  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}
```

לבנייה הבסיסי `props` שימו לב כיצד אנו מעבירים את

```
constructor(props) {
  super(props);
  this.state = {date: new Date()};
}
```

קומפוננטות מחלוקת תמיד לקרוא לבניה הבסיס עם `props`.

3. מאלמנט `date`-prop-הסירו את ה :

```
root.render(<Clock />);
```

לאחר מכן נוסיף את קוד הטימר בחזרה לקומפוננטה עצמה.

התוצאה נראה כך:

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }

  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Clock />);
```

[-נסו זאת ב-CodePen](#)

יגDIR טימר משלו ויעדכו את עצמו בכל שנייה `Clock`-בשלב הבא, נdag ש

הוספת מתודות מחזור חיים למחקה

באפליקציות עם קומפוננטות רבות, חשוב מאוד לשחרר משאבים שנלקחו על ידי הקומפוננטות כאשר הן מושמדות.

בפעם הראשונה. זה נקרא DOM-רונדר לערך `Clock` - אנחנו רוצים **להגדיר טימר** בכל פעם ש

ב-ב "mounting" בפעם השנייה. זה נקרא `Clock` -שמיוצר על ידי ה DOM- אנחנו גם רוצים **לנקות את אותו טימר** בכל פעם שה React.

ו `mount` אנו יכולים להזכיר על מתודות מיוחדות במחקה הקומפוננטה כדי להריץ קוד כלשהו כאשר קומפוננטה עשויה `unmount`:

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }

  componentDidMount() {}

  componentWillUnmount() {}

  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}
```

מתודות אלו נקראות "מתודות מחזור חיים".

זה מקום טוב להגדיר טימר DOM-ריצה לאחר שפלט הקומפוננטה רונדר ל `componentDidMount()` מתודה:

```
componentDidMount() {
  this.timerID = setInterval(
    () => this.tick(),
    1000
  );
}
```

ישו `this.timerID` - שימו לב איך אנו שומרים על זהה הטימר על ה

יש שימוש מיוחדת, אתם רשאים להוסיף שדות נוספים `this.state` - עצמה ו `this.props` מוגדר על-ידי -בעוד ש `componentWillUnmount()` אנו נחסל את הטימר במתודה מחזור החיים :

```
componentWillUnmount() {
  clearInterval(this.timerID);
}
```

בכל שנייה `Clock` שתקרא על ידי הקומפוננטה `tick()` לבסוף, נמש מתחודה הנקראת

המקומי של הקומפוננטה `state-state`-כדי לתזמן עדכנים ל-`this.setState()`-היא משתמש ב-

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }

  componentDidMount() {
    this.timerID = setInterval(
      () => this.tick(),
      1000
    );
  }

  componentWillUnmount() {
    clearInterval(this.timerID);
  }

  tick() {
    this.setState({
      date: new Date()
    });
  }

  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}</h2>
      </div>
    );
  }
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Clock />);
```

[-נסו זאת ב-CodePen](#)

כעת השעון מתatkק בכל שנייה.

בואו נסכם בזריזות את מה שקרה ואת הסדר שבו מתחודות נקראות:

1. צריך `Clock`-מכיוון ש `Clock` קוראת לבנייתו של הרכיב `ReactDOM.render()`, `React` מועבר לו `Clock`. כאשר `ReactDOM.render()` מוכן, הוא מעתיקת את הזמן הנוכחי. בהמשך נעדכן את ה `this.state` להציג את השעה הנוכחית, והוא מאטח את `state` זהה.
2. לומדת מה צריך להיות מוצג על `React` באופן זה. `render()`-של קומפוננטת `Clock` קוראת למתחודת או `Clock`. כדי שהיא תואם לפט שרוונדר על ידי DOM-מעדכנת את ה `React` המスク. אז

- בתוכה, קוראת למתודת מחזור החיים React-DOM, `componentDidMount()`. כאשר הפלט של 3. של הקומפוננטה פעם בשניה `tick()` מבקשת מהדפדפן להגדיר טימר כדי לקרוא למתודה `Clock` קומפוננטת.
4. מתזמנת את עדכון משך המשמש על ידי `Clock` בתוכה, קומפוננטת. `tick()` בכל שנייה הדפדפן קורא למתודה `setState()`, `React-state`-וידעת שה `setState()`, `React` מוכיל את הזמן הנוכחי. הותקן לкриאה ל `this.state`-קראה במתודה `render()` השנה, וקוראת למתודה `render()` בהתאם DOM-מעדכנת את ה `React`. יהיה שונה, ולכן הפלט המר้อนדר יכלול את הזמן המעדכן `render()`.
5. כך קוראת למתודת מחזור החיים React-DOM-מוסרת מה `Clock` אם הקומפוננטה. שהטימר יופסק.

שימוש-State נכוּן ב

ישנם שלושה דברים שעליים לדעת לגבי `setState()`.

ישירות State אל תשוּן את

למשל, זה לא ירנnder מחדש קומפוננטה:

```
// טעות
this.state.comment = 'Hello';
```

`setState()` -במקום, השתמשו ב :

```
// נכון
this.setState({comment: 'Hello'});
```

הוא הבניי `this.state` -המקום היחיד שבו אתם כן יכולים לבצע השמה ל.

יכולים להיות אסינכרוניים עדכוני State

לתוכן ייחד על מנת לשפר ביצועים `setState()`-עשווה לקבץ מספר קריאות ל

עשויים להתעדכן באופן אסינכרוני, עליהם לא להסתמך על הערכים שלהם לצורך `this.state` -ו- `this.props` ש `this.state`-מכיון ש `this.state` הוא-state-חישוב הבא.

למשל, קוד זה עלול להכשל בעדכון המונה:

```
// טעות
this.setState({
  counter: this.state.counter + this.props.increment,
});
```

state-state-שמקביל פונקציה במקום אובייקט. פונקציה זו מקבל את ה `setState()` כדי לתקן זאת, השתמש בסוג השני של בעית החלת העדכון ארגומנטו השני `props`-הקודם ארגומנטו הראשון, ואת ה

```
// נכון
this.setState((state, props) => ({
  counter: state.counter + props.increment
}));
```

אנו משתמשים בפונקציית `hz` למעלה, אבל זה עובד גם עם פונקציות וגילות:

```
// וכוכב
this.setState(function(state, props) {
  return {
    counter: state.counter + props.increment
  };
});
```

ממוזגים State עדכוני

הנוכחי `state`-ממזגת את האובייקט שאתם מספקים ל `setState()`, React -כאשר אתם קוראים ל

שלכם עלול להכיל מספר משתנים בלתי תלויים `state`-למשל, ה

```
constructor(props) {
  super(props);
  this.state = {
    posts: [],
    comments: []
  };
}
```

נפרדות `setState()` לאחר מכן תוכלו לעדכן אותם באופן בלתי תלוי באמצעות קריאות:

```
componentDidMount() {
  fetchPosts().then(response => {
    this.setState({
      posts: response.posts
    });
  });

  fetchComments().then(response => {
    this.setState({
      comments: response.comments
    });
  });
}
```

לא שינוי, אך מחליף לחלוון `this.setState({comments: this.state.comments})` משאיר את `this.state.posts` המזיהוג הוא רדוד, ולכן את `this.state.comments`.

הנתונים זורמים למטה

ולא צריך `state` או ללא `state` לא קומפוננטות הורים ולא קומפוננטות ילדים יכולות לדעת אם קומפוננטה מסוימת בעלת לשנות להן אם היא מוגדרת כפונקציה או כמחלקה.

נקרא לעיתים קרובות מקומי או מוכמס. הוא אינו נגיש לאף קומפוננטה אחרת פרט לו שהוא בעלותה `state`-זו הסיבה שומגדרה אותו.

לקומפוננטות הילדים שלה `props` יכולה לבחור להעביר את ה

```
<FormattedDate date={this.state.date} />
```

מה props של state-שלה ולא תדע אם הוא בא מה props-ב-props הקומפוננטה `FormattedDate` -תקבל את ה `date` או שהוקלד ידנית `Clock` של `Clock`:

```
function FormattedDate(props) {
  return <h2>It is {props.date.toLocaleTimeString()}.</h2>;
}
```

[נסה זאת ב-CodePen](#)

הוא תמיד בבעלות קומפוננטה מסוימת, וכל state זה נקרא בדרך כלל זרימת הנתונים "מלמעלה למטה" או "חדר כיווניות". כל זה יכול להשפיע רק על קומפוננטות שנמצאות "מתחתיham" בעץ-state-נתנו או ממש משתמש הנגזר מה

של קומפוננטה היא כמו מקור מים נוסף שמצויר אליו בנקודה state כל props אם אתם מודמיינים עץ קומפוננטות כמפל של שרירותית אבל גם זורם כלפי מטה.

כדי להראות שכל הקומפוננטות מבודדות באמת, אנו יכולים ליצור קומפוננטה `App`:

```
function App() {
  return (
    <div>
      <Clock />
      <Clock />
      <Clock />
    </div>
  );
}
```

[נסה זאת ב-CodePen](#)

מיקים טימר משלו ומתעדכן באופן עצמאי `Clock` כל

נחשבת לפriet מימוש של הקומפוננטה שעשוי לשתנות state או חסר state ההחלה אם רכיב הוא בעל React, באפליקציות וליהיפך, בתוך קומפוננטות בעלות state לאורך זמן. אתם יכולים להשתמש בקומפוננטות הקשורות

Static Type Checking

Static type checkers like [Flow](#) and [TypeScript](#) identify certain types of problems before you even run your code. They can also improve developer workflow by adding features like auto-completion. For this reason, we recommend using Flow or TypeScript instead of `PropTypes` for larger code bases.

Flow

[Flow](#) is a static type checker for your JavaScript code. It is developed at Facebook and is often used with React. It lets you annotate the variables, functions, and React components with a special type syntax, and catch mistakes early. You can read an [introduction to Flow](#) to learn its basics.

To use Flow, you need to:

- Add Flow to your project as a dependency.
- Ensure that Flow syntax is stripped from the compiled code.
- Add type annotations and run Flow to check them.

We will explain these steps below in detail.

Adding Flow to a Project

First, navigate to your project directory in the terminal. You will need to run the following command:

If you use [Yarn](#), run:

```
yarn add --dev flow-bin
```

If you use [npm](#), run:

```
npm install --save-dev flow-bin
```

This command installs the latest version of Flow into your project.

Now, add `flow` to the `"scripts"` section of your `package.json` to be able to use this from the terminal:

```
{
  // ...
  "scripts": {
    "flow": "flow",
    // ...
  },
  // ...
}
```

Finally, run one of the following commands:

If you use [Yarn](#), run:

```
yarn run flow init
```

If you use `npm`, run:

```
npm run flow init
```

This command will create a Flow configuration file that you will need to commit.

Stripping Flow Syntax from the Compiled Code

Flow extends the JavaScript language with a special syntax for type annotations. However, browsers aren't aware of this syntax, so we need to make sure it doesn't end up in the compiled JavaScript bundle that is sent to the browser.

The exact way to do this depends on the tools you use to compile JavaScript.

Create React App

If your project was set up using [Create React App](#), congratulations! The Flow annotations are already being stripped by default so you don't need to do anything else in this step.

Babel

Note:

These instructions are *not* for Create React App users. Even though Create React App uses Babel under the hood, it is already configured to understand Flow. Only follow this step if you *don't* use Create React App.

If you manually configured Babel for your project, you will need to install a special preset for Flow.

If you use Yarn, run:

```
yarn add --dev @babel/preset-flow
```

If you use npm, run:

```
npm install --save-dev @babel/preset-flow
```

Then add the `flow` preset to your [Babel configuration](#). For example, if you configure Babel through `.babelrc` file, it could look like this:

```
{  
  "presets": [  
    "@babel/preset-flow",  
    "react"  
  ]  
}
```

This will let you use the Flow syntax in your code.

Note:

Flow does not require the `react` preset, but they are often used together. Flow itself understands JSX syntax out of the box.

Other Build Setups

If you don't use either Create React App or Babel, you can use [flow-remove-types](#) to strip the type annotations.

Running Flow

If you followed the instructions above, you should be able to run Flow for the first time.

```
yarn flow
```

If you use npm, run:

```
npm run flow
```

You should see a message like:

```
No errors!
⭐ Done in 0.17s.
```

Adding Flow Type Annotations

By default, Flow only checks the files that include this annotation:

```
// @flow
```

Typically it is placed at the top of a file. Try adding it to some files in your project and run `yarn flow` or `npm run flow` to see if Flow already found any issues.

There is also [an option](#) to force Flow to check *all* files regardless of the annotation. This can be too noisy for existing projects, but is reasonable for a new project if you want to fully type it with Flow.

Now you're all set! We recommend to check out the following resources to learn more about Flow:

- [Flow Documentation: Type Annotations](#)
- [Flow Documentation: Editors](#)
- [Flow Documentation: React](#)
- [Linting in Flow](#)

TypeScript

TypeScript is a programming language developed by Microsoft. It is a typed superset of JavaScript, and includes its own compiler. Being a typed language, TypeScript can catch errors and bugs at build time, long before your app goes live. You can learn more about using TypeScript with React [here](#).

To use TypeScript, you need to:

- Add TypeScript as a dependency to your project
- Configure the TypeScript compiler options
- Use the right file extensions
- Add definitions for libraries you use

Let's go over these in detail.

Using TypeScript with Create React App

Create React App supports TypeScript out of the box.

To create a **new project** with TypeScript support, run:

```
npx create-react-app my-app --template typescript
```

You can also add it to an **existing Create React App project**, as documented [here](#).

Note:

If you use Create React App, you can **skip the rest of this page**. It describes the manual setup which doesn't apply to Create React App users.

Adding TypeScript to a Project

It all begins with running one command in your terminal.

If you use [Yarn](#), run:

```
yarn add --dev typescript
```

If you use [npm](#), run:

```
npm install --save-dev typescript
```

Congrats! You've installed the latest version of TypeScript into your project. Installing TypeScript gives us access to the `tsc` command. Before configuration, let's add `tsc` to the "scripts" section in our `package.json`:

```
{
  // ...
  "scripts": {
    "build": "tsc",
    // ...
  },
  // ...
}
```

Configuring the TypeScript Compiler

The compiler is of no help to us until we tell it what to do. In TypeScript, these rules are defined in a special file called `tsconfig.json`. To generate this file:

If you use [Yarn](#), run:

```
yarn run tsc --init
```

If you use [npm](#), run:

```
npx tsc --init
```

Looking at the now generated `tsconfig.json`, you can see that there are many options you can use to configure the compiler. For a detailed description of all the options, check [here](#).

Of the many options, we'll look at `rootDir` and `outDir`. In its true fashion, the compiler will take in typescript files and generate javascript files. However we don't want to get confused with our source files and the generated output.

We'll address this in two steps:

- Firstly, let's arrange our project structure like this. We'll place all our source code in the `src` directory.

```

├── package.json
├── src
│   └── index.ts
└── tsconfig.json
```

- Next, we'll tell the compiler where our source code is and where the output should go.

```
// tsconfig.json

{
  "compilerOptions": {
    // ...
    "rootDir": "src",
    "outDir": "build"
    // ...
  },
}
```

Great! Now when we run our build script the compiler will output the generated javascript to the `build` folder. The [TypeScript React Starter](#) provides a `tsconfig.json` with a good set of rules to get you started.

Generally, you don't want to keep the generated javascript in your source control, so be sure to add the build folder to your `.gitignore`.

File extensions

In React, you most likely write your components in a `.js` file. In TypeScript we have 2 file extensions:

`.ts` is the default file extension while `.tsx` is a special extension used for files which contain `JSX`.

Running TypeScript

If you followed the instructions above, you should be able to run TypeScript for the first time.

```
yarn build
```

If you use npm, run:

```
npm run build
```

If you see no output, it means that it completed successfully.

Type Definitions

To be able to show errors and hints from other packages, the compiler relies on declaration files. A declaration file provides all the type information about a library. This enables us to use javascript libraries like those on npm in our project.

There are two main ways to get declarations for a library:

Bundled - The library bundles its own declaration file. This is great for us, since all we need to do is install the library, and we can use it right away. To check if a library has bundled types, look for an `index.d.ts` file in the project. Some libraries will have it specified in their `package.json` under the `typings` or `types` field.

DefinitelyTyped - DefinitelyTyped is a huge repository of declarations for libraries that don't bundle a declaration file. The declarations are crowd-sourced and managed by Microsoft and open source contributors. React for example doesn't bundle its own declaration file. Instead we can get it from DefinitelyTyped. To do so enter this command in your terminal.

```
# yarn
yarn add --dev @types/react

# npm
npm i --save-dev @types/react
```

Local Declarations Sometimes the package that you want to use doesn't bundle declarations nor is it available on DefinitelyTyped. In that case, we can have a local declaration file. To do this, create a `declarations.d.ts` file in the root of your source directory. A simple declaration could look like this:

```
declare module 'querystring' {
  export function stringify(val: object): string
  export function parse(val: string): object
}
```

You are now ready to code! We recommend to check out the following resources to learn more about TypeScript:

- [TypeScript Documentation: Everyday Types](#)
- [TypeScript Documentation: Migrating from JavaScript](#)
- [TypeScript Documentation: React and Webpack](#)

ReScript

[ReScript](#) is a typed language that compiles to JavaScript. Some of its core features are guaranteed 100% type coverage, first-class JSX support and [dedicated React bindings](#) to allow integration in existing JS / TS React codebases.

You can find more infos on integrating ReScript in your existing JS / React codebase [here](#).

Kotlin

[Kotlin](#) is a statically typed language developed by JetBrains. Its target platforms include the JVM, Android, LLVM, and [JavaScript](#).

JetBrains develops and maintains several tools specifically for the React community: [React bindings](#) as well as [Create React Kotlin App](#). The latter helps you start building React apps with Kotlin with no build configuration.

Other Languages

Note there are other statically typed languages that compile to JavaScript and are thus React compatible. For example, [F#/Fable](#) with [elmish-react](#). Check out their respective sites for more information, and feel free to add more statically typed languages that work with React to this page!

[Go to TOC](#)

Strict Mode

`StrictMode` is a tool for highlighting potential problems in an application. Like `Fragment`, `StrictMode` does not render any visible UI. It activates additional checks and warnings for its descendants.

Note:

Strict mode checks are run in development mode only; *they do not impact the production build*.

You can enable strict mode for any part of your application. For example: `embed:strict-mode/enabling-strict-mode.js`

In the above example, strict mode checks will *not* be run against the `Header` and `Footer` components. However, `ComponentOne` and `ComponentTwo`, as well as all of their descendants, will have the checks.

`StrictMode` currently helps with:

- Identifying components with unsafe lifecycles
- Warning about legacy string ref API usage
- Warning about deprecated findDOMNode usage
- Detecting unexpected side effects
- Detecting legacy context API
- Ensuring reusable state

Additional functionality will be added with future releases of React.

Identifying unsafe lifecycles

As explained [in this blog post](#), certain legacy lifecycle methods are unsafe for use in async React applications. However, if your application uses third party libraries, it can be difficult to ensure that these lifecycles aren't being used. Fortunately, strict mode can help with this!

When strict mode is enabled, React compiles a list of all class components using the unsafe lifecycles, and logs a warning message with information about these components, like so:

```
▶ Warning: Unsafe lifecycle methods were found within a strict-mode tree:
  in div (created by ExampleApplication)
  in ExampleApplication

componentWillMount: Please update the following components to use componentDidMount instead: ThirdPartyComponent

Learn more about this warning here:
https://fb.me/react-strict-mode-warnings
```

Addressing the issues identified by strict mode *now* will make it easier for you to take advantage of concurrent rendering in future releases of React.

Warning about legacy string ref API usage

Previously, React provided two ways for managing refs: the legacy string ref API and the callback API. Although the string ref API was the more convenient of the two, it had [several downsides](#) and so our official recommendation was to [use the callback form instead](#).

React 16.3 added a third option that offers the convenience of a string ref without any of the downsides:

[embed:16-3-release-blog-post/create-ref-example.js](#)

Since object refs were largely added as a replacement for string refs, strict mode now warns about usage of string refs.

Note:

Callback refs will continue to be supported in addition to the new `createRef` API.

You don't need to replace callback refs in your components. They are slightly more flexible, so they will remain as an advanced feature.

[Learn more about the new `createRef` API here.](#)

Warning about deprecated findDOMNode usage

React used to support `findDOMNode` to search the tree for a DOM node given a class instance. Normally you don't need this because you can [attach a ref directly to a DOM node](#).

`findDOMNode` can also be used on class components but this was breaking abstraction levels by allowing a parent to demand that certain children were rendered. It creates a refactoring hazard where you can't change the implementation details of a component because a parent might be reaching into its DOM node. `findDOMNode` only returns the first child, but with the use of Fragments, it is possible for a component to render multiple DOM nodes. `findDOMNode` is a one time read API. It only gave you an answer when you asked for it. If a child component renders a different node, there is no way to handle this change. Therefore `findDOMNode` only worked if components always return a single DOM node that never changes.

You can instead make this explicit by passing a ref to your custom component and pass that along to the DOM using [ref forwarding](#).

You can also add a wrapper DOM node in your component and attach a ref directly to it.

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.wrapper = React.createRef();
  }
  render() {
    return <div ref={this.wrapper}>{this.props.children}</div>;
  }
}
```

Note:

In CSS, the `display: contents` attribute can be used if you don't want the node to be part of the layout.

Detecting unexpected side effects

Conceptually, React does work in two phases:

- The **render** phase determines what changes need to be made to e.g. the DOM. During this phase, React calls `render` and then compares the result to the previous render.
- The **commit** phase is when React applies any changes. (In the case of React DOM, this is when React inserts, updates, and removes DOM nodes.) React also calls lifecycles like `componentDidMount` and `componentDidUpdate` during this phase.

The commit phase is usually very fast, but rendering can be slow. For this reason, the upcoming concurrent mode (which is not enabled by default yet) breaks the rendering work into pieces, pausing and resuming the work to avoid blocking the browser. This means that React may invoke render phase lifecycles more than once before committing, or it may invoke them without committing at all (because of an error or a higher priority interruption).

Render phase lifecycles include the following class component methods:

- `constructor`
- `componentWillMount` (or `UNSAFE_componentWillMount`)
- `componentWillReceiveProps` (or `UNSAFE_componentWillReceiveProps`)
- `componentWillUpdate` (or `UNSAFE_componentWillUpdate`)
- `getDerivedStateFromProps`
- `shouldComponentUpdate`
- `render`
- `setState` updater functions (the first argument)

Because the above methods might be called more than once, it's important that they do not contain side-effects. Ignoring this rule can lead to a variety of problems, including memory leaks and invalid application state. Unfortunately, it can be difficult to detect these problems as they can often be [non-deterministic](#).

Strict mode can't automatically detect side effects for you, but it can help you spot them by making them a little more deterministic. This is done by intentionally double-invoking the following functions:

- Class component `constructor`, `render`, and `shouldComponentUpdate` methods
- Class component static `getDerivedStateFromProps` method
- Function component bodies
- State updater functions (the first argument to `setState`)
- Functions passed to `useState`, `useMemo`, or `useReducer`

Note:

This only applies to development mode. *Lifecycles will not be double-invoked in production mode.*

For example, consider the following code: [embed:strict-mode/side-effects-in-constructor.js](#)

At first glance, this code might not seem problematic. But if `SharedApplicationState.recordEvent` is not [idempotent](#), then instantiating this component multiple times could lead to invalid application state. This sort of subtle bug might not manifest during development, or it might do so inconsistently and so be overlooked.

By intentionally double-invoking methods like the component constructor, strict mode makes patterns like this easier to spot.

Note:

In React 17, React automatically modifies the console methods like `console.log()` to silence the logs in the second call to lifecycle functions. However, it may cause undesired behavior in certain cases where [a workaround can be used](#).

Starting from React 18, React does not suppress any logs. However, if you have React DevTools installed, the logs from the second call will appear slightly dimmed. React DevTools also offers a setting (off by default) to suppress them completely.

Detecting legacy context API

The legacy context API is error-prone, and will be removed in a future major version. It still works for all 16.x releases but will show this warning message in strict mode:

```
✖ ► Warning: Legacy context API has been detected within a strict-mode tree:
  in div (at App.js:32)
  in App (at index.js:7)

  Please update the following components: LegacyContextConsumer, LegacyContextProvider

  Learn more about this warning here:
  https://fb.me/react-strict-mode-warnings
```

Read the [new context API documentation](#) to help migrate to the new version.

Ensuring reusable state

In the future, we'd like to add a feature that allows React to add and remove sections of the UI while preserving state. For example, when a user tabs away from a screen and back, React should be able to immediately show the previous screen. To do this, React support remounting trees using the same component state used before unmounting.

This feature will give React better performance out-of-the-box, but requires components to be resilient to effects being mounted and destroyed multiple times. Most effects will work without any changes, but some effects do not properly clean up subscriptions in the destroy callback, or implicitly assume they are only mounted or destroyed once.

To help surface these issues, React 18 introduces a new development-only check to Strict Mode. This new check will automatically unmount and remount every component, whenever a component mounts for the first time, restoring the previous state on the second mount.

To demonstrate the development behavior you'll see in Strict Mode with this feature, consider what happens when React mounts a new component. Without this change, when a component mounts, React creates the effects:

- * React mounts the component.
- * Layout effects are created.
- * Effects are created.

With Strict Mode starting in React 18, whenever a component mounts in development, React will simulate immediately unmounting and remounting the component:

- * React mounts the component.
 - * Layout effects are created.
 - * Effect effects are created.
- * React simulates effects being destroyed on a mounted component.
 - * Layout effects are destroyed.
 - * Effects are destroyed.
- * React simulates effects being re-created on a mounted component.
 - * Layout effects are created
 - * Effect setup code runs

On the second mount, React will restore the state from the first mount. This feature simulates user behavior such as a user tabbing away from a screen and back, ensuring that code will properly handle state restoration.

When the component unmounts, effects are destroyed as normal:

- * React unmounts the component.
- * Layout effects are destroyed.
- * Effect effects are destroyed.

Unmounting and remounting includes:

- `componentDidMount`
- `componentWillUnmount`
- `useEffect`
- `useLayoutEffect`
- `useInsertionEffect`

Note:

This only applies to development mode, *production behavior is unchanged*.

For help supporting common issues, see:

- [How to support Reusable State in Effects](#)

[Go to TOC](#)

Testing Environments

This document goes through the factors that can affect your environment and recommendations for some scenarios.

Test runners

Test runners like [Jest](#), [mocha](#), [ava](#) let you write test suites as regular JavaScript, and run them as part of your development process. Additionally, test suites are run as part of continuous integration.

- Jest is widely compatible with React projects, supporting features like mocked [modules](#) and [timers](#), and [jsdom](#) support. **If you use Create React App, Jest is already included out of the box with useful defaults.**
- Libraries like [mocha](#) work well in real browser environments, and could help for tests that explicitly need it.
- End-to-end tests are used for testing longer flows across multiple pages, and require a [different setup](#).

Mocking a rendering surface

Tests often run in an environment without access to a real rendering surface like a browser. For these environments, we recommend simulating a browser with [jsdom](#), a lightweight browser implementation that runs inside Node.js.

In most cases, jsdom behaves like a regular browser would, but doesn't have features like [layout and navigation](#). This is still useful for most web-based component tests, since it runs quicker than having to start up a browser for each test. It also runs in the same process as your tests, so you can write code to examine and assert on the rendered DOM.

Just like in a real browser, jsdom lets us model user interactions; tests can dispatch events on DOM nodes, and then observe and assert on the side effects of these actions ([example](#)).

A large portion of UI tests can be written with the above setup: using Jest as a test runner, rendered to jsdom, with user interactions specified as sequences of browser events, powered by the [act\(\)](#) helper ([example](#)). For example, a lot of React's own tests are written with this combination.

If you're writing a library that tests mostly browser-specific behavior, and requires native browser behavior like layout or real inputs, you could use a framework like [mocha](#).

In an environment where you *can't* simulate a DOM (e.g. testing React Native components on Node.js), you could use [event simulation helpers](#) to simulate interactions with elements. Alternately, you could use the [fireEvent](#) helper from [@testing-library/react-native](#).

Frameworks like [Cypress](#), [puppeteer](#) and [webdriver](#) are useful for running [end-to-end tests](#).

Mocking functions

When writing tests, we'd like to mock out the parts of our code that don't have equivalents inside our testing environment (e.g. checking `navigator.onLine` status inside Node.js). Tests could also spy on some functions, and observe how other parts of the test interact with them. It is then useful to be able to selectively mock these functions with test-friendly versions.

This is especially useful for data fetching. It is usually preferable to use "fake" data for tests to avoid the slowness and flakiness due to fetching from real API endpoints ([example](#)). This helps make the tests predictable. Libraries like [Jest](#) and [sinon](#), among others, support mocked functions. For end-to-end tests, mocking network can be more difficult, but you might also want to test the real API endpoints in them anyway.

Mocking modules

Some components have dependencies for modules that may not work well in test environments, or aren't essential to our tests. It can be useful to selectively mock these modules out with suitable replacements ([example](#)).

On Node.js, runners like Jest [support mocking modules](#). You could also use libraries like `mock-require`.

Mocking timers

Components might be using time-based functions like `setTimeout`, `setInterval`, or `Date.now`. In testing environments, it can be helpful to mock these functions out with replacements that let you manually "advance" time. This is great for making sure your tests run fast! Tests that are dependent on timers would still resolve in order, but quicker ([example](#)). Most frameworks, including [Jest](#), [sinon](#) and [Iolex](#), let you mock timers in your tests.

Sometimes, you may not want to mock timers. For example, maybe you're testing an animation, or interacting with an endpoint that's sensitive to timing (like an API rate limiter). Libraries with timer mocks let you enable and disable them on a per test/suite basis, so you can explicitly choose how these tests would run.

End-to-end tests

End-to-end tests are useful for testing longer workflows, especially when they're critical to your business (such as payments or signups). For these tests, you'd probably want to test how a real browser renders the whole app, fetches data from the real API endpoints, uses sessions and cookies, navigates between different links. You might also likely want to make assertions not just on the DOM state, but on the backing data as well (e.g. to verify whether the updates have been persisted to the database).

In this scenario, you would use a framework like [Cypress](#), [Playwright](#) or a library like [Puppeteer](#) so you can navigate between multiple routes and assert on side effects not just in the browser, but potentially on the backend as well.

[Go to TOC](#)

Testing Recipes

Common testing patterns for React components.

Note:

This page assumes you're using [Jest](#) as a test runner. If you use a different test runner, you may need to adjust the API, but the overall shape of the solution will likely be the same. Read more details on setting up a testing environment on the [Testing Environments](#) page.

On this page, we will primarily use function components. However, these testing strategies don't depend on implementation details, and work just as well for class components too.

- [Setup/Tear down](#)
- [act\(\)](#)
- [Rendering](#)
- [Data Fetching](#)
- [Mocking Modules](#)
- [Events](#)
- [Timers](#)
- [Snapshot Testing](#)
- [Multiple Renderers](#)
- [Something Missing?](#)

Setup/Tear down

For each test, we usually want to render our React tree to a DOM element that's attached to `document`. This is important so that it can receive DOM events. When the test ends, we want to "clean up" and unmount the tree from the `document`.

A common way to do it is to use a pair of `beforeEach` and `afterEach` blocks so that they'll always run and isolate the effects of a test to itself:

```
import { unmountComponentAtNode } from "react-dom";

let container = null;
beforeEach(() => {
  // setup a DOM element as a render target
  container = document.createElement("div");
  document.body.appendChild(container);
});

afterEach(() => {
  // cleanup on exiting
  unmountComponentAtNode(container);
```

```
    container.remove();
    container = null;
});
```

You may use a different pattern, but keep in mind that we want to execute the cleanup *even if a test fails*. Otherwise, tests can become "leaky", and one test can change the behavior of another test. That makes them difficult to debug.

act() {#act}

When writing UI tests, tasks like rendering, user events, or data fetching can be considered as "units" of interaction with a user interface. `react-dom/test-utils` provides a helper called `act()` that makes sure all updates related to these "units" have been processed and applied to the DOM before you make any assertions:

```
act(() => {
  // render components
});
// make assertions
```

This helps make your tests run closer to what real users would experience when using your application. The rest of these examples use `act()` to make these guarantees.

You might find using `act()` directly a bit too verbose. To avoid some of the boilerplate, you could use a library like [React Testing Library](#), whose helpers are wrapped with `act()`.

Note:

The name `act` comes from the [Arrange-Act-Assert](#) pattern.

Rendering

Commonly, you might want to test whether a component renders correctly for given props. Consider a simple component that renders a message based on a prop:

```
// hello.js

import React from "react";

export default function Hello(props) {
  if (props.name) {
    return <h1>Hello, {props.name}!</h1>;
  } else {
    return <span>Hey, stranger</span>;
  }
}
```

We can write a test for this component:

```
// hello.test.js

import React from "react";
import { render, unmountComponentAtNode } from "react-dom";
import { act } from "react-dom/test-utils";

import Hello from "./hello";

let container = null;
beforeEach(() => {
  // setup a DOM element as a render target
  container = document.createElement("div");
  document.body.appendChild(container);
});

afterEach(() => {
  // cleanup on exiting
  unmountComponentAtNode(container);
  container.remove();
  container = null;
});

it("renders with or without a name", () => {
  act(() => {
    render(<Hello />, container);
  });
  expect(container.textContent).toBe("Hey, stranger");

  act(() => {
    render(<Hello name="Jenny" />, container);
  });
  expect(container.textContent).toBe("Hello, Jenny!");

  act(() => {
    render(<Hello name="Margaret" />, container);
  });
  expect(container.textContent).toBe("Hello, Margaret!");
});
```

Data Fetching

Instead of calling real APIs in all your tests, you can mock requests with dummy data. Mocking data fetching with "fake" data prevents flaky tests due to an unavailable backend, and makes them run faster. Note: you may still want to run a subset of tests using an "[end-to-end](#)" framework that tells whether the whole app is working together.

```
// user.js

import React, { useState, useEffect } from "react";

export default function User(props) {
  const [user, setUser] = useState(null);

  async function fetchUserData(id) {
    const response = await fetch(`/${id}`);
    setUser(await response.json());
  }

  useEffect(() => {
    fetchUserData(props.id);
  });
}
```

```

}, [props.id]);

if (!user) {
  return "loading...";
}

return (
  <details>
    <summary>{user.name}</summary>
    <strong>{user.age}</strong> years old
    <br />
    lives in {user.address}
  </details>
);
}

```

We can write tests for it:

```

// user.test.js

import React from "react";
import { render, unmountComponentAtNode } from "react-dom";
import { act } from "react-dom/test-utils";
import User from "./user";

let container = null;
beforeEach(() => {
  // setup a DOM element as a render target
  container = document.createElement("div");
  document.body.appendChild(container);
});

afterEach(() => {
  // cleanup on exiting
  unmountComponentAtNode(container);
  container.remove();
  container = null;
});

it("renders user data", async () => {
  const fakeUser = {
    name: "Joni Baez",
    age: "32",
    address: "123, Charming Avenue"
  };

  jest.spyOn(global, "fetch").mockImplementation(() =>
    Promise.resolve({
      json: () => Promise.resolve(fakeUser)
    })
  );

  // Use the asynchronous version of act to apply resolved promises
  await act(async () => {
    render(<User id="123" />, container);
  });

  expect(container.querySelector("summary").textContent).toBe(fakeUser.name);
  expect(container.querySelector("strong").textContent).toBe(fakeUser.age);
  expect(container.textContent).toContain(fakeUser.address);
})

```

```
// remove the mock to ensure tests are completely isolated
global.fetch.mockRestore();
});
```

Mocking Modules

Some modules might not work well inside a testing environment, or may not be as essential to the test itself. Mocking out these modules with dummy replacements can make it easier to write tests for your own code.

Consider a `Contact` component that embeds a third-party `GoogleMap` component:

```
// map.js

import React from "react";

import { LoadScript, GoogleMap } from "react-google-maps";
export default function Map(props) {
  return (
    <LoadScript id="script-loader" googleMapsApiKey="YOUR_API_KEY">
      <GoogleMap id="example-map" center={props.center} />
    </LoadScript>
  );
}

// contact.js

import React from "react";
import Map from "./map";

export default function Contact(props) {
  return (
    <div>
      <address>
        Contact {props.name} via{" "}
        <a data-testid="email" href={"mailto:" + props.email}>
          email
        </a>
        or on their <a data-testid="site" href={props.site}>
          website
        </a>.
      </address>
      <Map center={props.center} />
    </div>
  );
}
```

If we don't want to load this component in our tests, we can mock out the dependency itself to a dummy component, and run our tests:

```
// contact.test.js

import React from "react";
import { render, unmountComponentAtNode } from "react-dom";
import { act } from "react-dom/test-utils";

import Contact from "./contact";
import MockedMap from "./map";
```

```

jest.mock("./map", () => {
  return function DummyMap(props) {
    return (
      <div data-testid="map">
        {props.center.lat}:{props.center.long}
      </div>
    );
  };
});

let container = null;
beforeEach(() => {
  // setup a DOM element as a render target
  container = document.createElement("div");
  document.body.appendChild(container);
});

afterEach(() => {
  // cleanup on exiting
  unmountComponentAtNode(container);
  container.remove();
  container = null;
});

it("should render contact information", () => {
  const center = { lat: 0, long: 0 };
  act(() => {
    render(
      <Contact
        name="Joni Baez"
        email="test@example.com"
        site="http://test.com"
        center={center}
      />,
      container
    );
  });
  expect(
    container.querySelector("[data-testid='email']").getAttribute("href")
  ).toEqual("mailto:test@example.com");

  expect(
    container.querySelector('[data-testid="site"]').getAttribute("href")
  ).toEqual("http://test.com");

  expect(container.querySelector('[data-testid="map"]').textContent).toEqual(
    "0:0"
  );
});
});

```

Events

We recommend dispatching real DOM events on DOM elements, and then asserting on the result. Consider a `Toggle` component:

```

// toggle.js

import React, { useState } from "react";
export default function Toggle(props) {

```

```

const [state, setState] = useState(false);
return (
  <button
    onClick={() => {
      setState(previousState => !previousState);
      props.onChange(!state);
    }}
    data-testid="toggle"
  >
    {state === true ? "Turn off" : "Turn on"}
  </button>
);
}

```

We could write tests for it:

```

// toggle.test.js

import React from "react";
import { render, unmountComponentAtNode } from "react-dom";
import { act } from "react-dom/test-utils";

import Toggle from "./toggle";

let container = null;
beforeEach(() => {
  // setup a DOM element as a render target
  container = document.createElement("div");
  document.body.appendChild(container);
});

afterEach(() => {
  // cleanup on exiting
  unmountComponentAtNode(container);
  container.remove();
  container = null;
});

it("changes value when clicked", () => {
  const onChange = jest.fn();
  act(() => {
    render(<Toggle onChange={onChange} />, container);
  });

  // get a hold of the button element, and trigger some clicks on it
  const button = document.querySelector("[data-testid=toggle]");
  expect(button.innerHTML).toBe("Turn on");

  act(() => {
    button.dispatchEvent(new MouseEvent("click", { bubbles: true }));
  });

  expect(onChange).toHaveBeenCalledTimes(1);
  expect(button.innerHTML).toBe("Turn off");

  act(() => {
    for (let i = 0; i < 5; i++) {
      button.dispatchEvent(new MouseEvent("click", { bubbles: true }));
    }
  });
}

```

```
expect(onChange).toHaveBeenCalledTimes(6);
expect(button.innerHTML).toBe("Turn on");
});
```

Different DOM events and their properties are described in [MDN](#). Note that you need to pass `{ bubbles: true }` in each event you create for it to reach the React listener because React automatically delegates events to the root.

Note:

React Testing Library offers a [more concise helper](#) for firing events.

Timers

Your code might use timer-based functions like `setTimeout` to schedule more work in the future. In this example, a multiple choice panel waits for a selection and advances, timing out if a selection isn't made in 5 seconds:

```
// card.js

import React, { useEffect } from "react";

export default function Card(props) {
  useEffect(() => {
    const timeoutID = setTimeout(() => {
      props.onSelect(null);
    }, 5000);
    return () => {
      clearTimeout(timeoutID);
    };
  }, [props.onSelect]);

  return [1, 2, 3, 4].map(choice => (
    <button
      key={choice}
      data-testid={choice}
      onClick={() => props.onSelect(choice)}
    >
      {choice}
    </button>
  ));
}
```

We can write tests for this component by leveraging [Jest's timer mocks](#), and testing the different states it can be in.

```
// card.test.js

import React from "react";
import { render, unmountComponentAtNode } from "react-dom";
import { act } from "react-dom/test-utils";

import Card from "./card";
```

```

let container = null;
beforeEach(() => {
  // setup a DOM element as a render target
  container = document.createElement("div");
  document.body.appendChild(container);
  jest.useFakeTimers();
});

afterEach(() => {
  // cleanup on exiting
  unmountComponentAtNode(container);
  container.remove();
  container = null;
  jest.useRealTimers();
});

it("should select null after timing out", () => {
  const onSelect = jest.fn();
  act(() => {
    render(<Card onSelect={onSelect} />, container);
  });

  // move ahead in time by 100ms
  act(() => {
    jest.advanceTimersByTime(100);
  });
  expect(onSelect).not.toHaveBeenCalled();

  // and then move ahead by 5 seconds
  act(() => {
    jest.advanceTimersByTime(5000);
  });
  expect(onSelect).toHaveBeenCalledWith(null);
});

it("should cleanup on being removed", () => {
  const onSelect = jest.fn();
  act(() => {
    render(<Card onSelect={onSelect} />, container);
  });

  act(() => {
    jest.advanceTimersByTime(100);
  });
  expect(onSelect).not.toHaveBeenCalled();

  // unmount the app
  act(() => {
    render(null, container);
  });

  act(() => {
    jest.advanceTimersByTime(5000);
  });
  expect(onSelect).not.toHaveBeenCalled();
});

it("should accept selections", () => {
  const onSelect = jest.fn();
  act(() => {
    render(<Card onSelect={onSelect} />, container);
  });
});

```

```

act(() => {
  container
    .querySelector("[data-testid='2']")
    .dispatchEvent(new MouseEvent("click", { bubbles: true }));
});

expect(onSelect).toHaveBeenCalledWith(2);
});

```

You can use fake timers only in some tests. Above, we enabled them by calling `jest.useFakeTimers()`. The main advantage they provide is that your test doesn't actually have to wait five seconds to execute, and you also didn't need to make the component code more convoluted just for testing.

Snapshot Testing

Frameworks like Jest also let you save "snapshots" of data with `toMatchSnapshot` / `toMatchInlineSnapshot`. With these, we can "save" the rendered component output and ensure that a change to it has to be explicitly committed as a change to the snapshot.

In this example, we render a component and format the rendered HTML with the `pretty` package, before saving it as an inline snapshot:

```

// hello.test.js, again

import React from "react";
import { render, unmountComponentAtNode } from "react-dom";
import { act } from "react-dom/test-utils";
import pretty from "pretty";

import Hello from "./hello";

let container = null;
beforeEach(() => {
  // setup a DOM element as a render target
  container = document.createElement("div");
  document.body.appendChild(container);
});

afterEach(() => {
  // cleanup on exiting
  unmountComponentAtNode(container);
  container.remove();
  container = null;
});

it("should render a greeting", () => {
  act(() => {
    render(<Hello />, container);
  });

  expect(
    pretty(container.innerHTML))
    .toMatchInlineSnapshot(); /* ... gets filled automatically by jest ... */

  act(() => {
    render(<Hello name="Jenny" />, container);
  });
}

```

```

expect(
  pretty(container.innerHTML)
).toMatchSnapshot(); /* ... gets filled automatically by jest ... */

act(() => {
  render(<Hello name="Margaret" />, container);
});

expect(
  pretty(container.innerHTML)
).toMatchSnapshot(); /* ... gets filled automatically by jest ... */
);

```

It's typically better to make more specific assertions than to use snapshots. These kinds of tests include implementation details so they break easily, and teams can get desensitized to snapshot breakages. Selectively [mocking some child components](#) can help reduce the size of snapshots and keep them readable for the code review.

Multiple Renderers

In rare cases, you may be running a test on a component that uses multiple renderers. For example, you may be running snapshot tests on a component with `react-test-renderer`, that internally uses `render` from `react-dom` inside a child component to render some content. In this scenario, you can wrap updates with `act()`'s corresponding to their renderers.

```

import { act as domAct } from "react-dom/test-utils";
import { act as testAct, create } from "react-test-renderer";
// ...
let root;
domAct(() => {
  testAct(() => {
    root = create(<App />);
  });
});
expect(root).toMatchSnapshot();

```

Something Missing?

If some common scenario is not covered, please let us know on the [issue tracker](#) for the documentation website.

Testing Overview

- "community/testing.html"

You can test React components similar to testing other JavaScript code.

There are a few ways to test React components. Broadly, they divide into two categories:

- **Rendering component trees** in a simplified test environment and asserting on their output.
- **Running a complete app** in a realistic browser environment (also known as "end-to-end" tests).

This documentation section focuses on testing strategies for the first case. While full end-to-end tests can be very useful to prevent regressions to important workflows, such tests are not concerned with React components in particular, and are out of the scope of this section.

Tradeoffs

When choosing testing tools, it is worth considering a few tradeoffs:

- **Iteration speed vs Realistic environment:** Some tools offer a very quick feedback loop between making a change and seeing the result, but don't model the browser behavior precisely. Other tools might use a real browser environment, but reduce the iteration speed and are flakier on a continuous integration server.
- **How much to mock:** With components, the distinction between a "unit" and "integration" test can be blurry. If you're testing a form, should its test also test the buttons inside of it? Or should a button component have its own test suite? Should refactoring a button ever break the form test?

Different answers may work for different teams and products.

Recommended Tools

[Jest](#) is a JavaScript test runner that lets you access the DOM via `jsdom`. While `jsdom` is only an approximation of how the browser works, it is often good enough for testing React components. Jest provides a great iteration speed combined with powerful features like mocking `modules` and `timers` so you can have more control over how the code executes.

[React Testing Library](#) is a set of helpers that let you test React components without relying on their implementation details. This approach makes refactoring a breeze and also nudges you towards best practices for accessibility. Although it doesn't provide a way to "shallowly" render a component without its children, a test runner like Jest lets you do this by [mocking](#).

Learn More

This section is divided in two pages:

- [Recipes](#): Common patterns when writing tests for React components.
- [Environments](#): What to consider when setting up a testing environment for React components.

[Go to TOC](#)

חשית ב-React

- 'blog/2013/11/05/thinking-in-react.html'
- 'docs/thinking-in-react-zh-CN.html'

לפי דעתנו, היא ספרייה הג'וואה-סקריפט המובילה בبنיה יישומי אינטראקט. הספרייה הוכיחה את עצמה אצלנו, בפיסבוק ובאינסטגרם.

הוא אירח גורם לכך לחשוב על היישומים בזמן שאתה בונה אותם. במדריך זה, נלווה React אחד מהיתרונות הבולטים של אותו בתקה החשית של בניית טבלת נתונים מוצרים באמצעות React.

נתחיל עם דוגמא

שנראה ככך JSON נניח שיש לנו מוצר

Search...	
<input type="checkbox"/> Only show products in stock	
Name	Price
Sporting Goods	
Football	\$49.99
Baseball	\$9.99
Basketball	\$29.99
Electronics	
iPod Touch	\$99.99
iPhone 5	\$399.99
Nexus 7	\$199.99

Mockup

שנראה ככך JSON המקורי מידע בפורמט

[{"Name": "Football", "Price": 49.99}, {"Name": "Baseball", "Price": 9.99}, {"Name": "Basketball", "Price": 29.99}, {"Name": "iPod Touch", "Price": 99.99}, {"Name": "iPhone 5", "Price": 399.99}, {"Name": "Nexus 7", "Price": 199.99}],

השלב הראשון: פיצול מוצר המשמש להיררכיות קומפוננטות

הדבר הראשון שנעשה הוא להקיף כל קומפוננטה (וותת קומפוננטה) ב קופסה, ולתת לכל אחת שם. אם אתם עובדים עם מעצב/ת, יכול להיות שהוא/היא כבר עשו את זה.

אבל איך ידעים מה צריך להיות קומפוננטה מסוימת? משתמשים באותה טכניקה שבה מחליטים אם ליצור מתודה או עצם. אחת השיטות היא [עיקרונות האחריות הבודדת](#), שאומר, שօפן אידיאלי כל קומפוננטה אמורה לעשות דבר אחד בלבד. אם הקומפוננטה גדולה, נצטרך לפצל אותה לתת-קומפוננטות.

למשתמש, נראה שם המודול בניית כראוי, מוצר המשמש שלנו) JSON משומן שלעויות תכופות נצטרך להציג מידע בפורמט גם מבנה הקומפוננטות) ימופה באופן מסודר. זה קורה מכיוון שמשתוק המשמש ומחדרי המידע נתונים לדובק באותו אורך-טווית מידע. פצל את מוצר המשמש לשך לקומפוננטות, שככל אחת מהן מתאימה לחלק אחד של מודול המידע שלו.



של קומפוננטות nesting תרשים המראה

נראה כאן שיש לנו חמישה קומפוננטות בישום שלנו. המידע שכל קומפוננטה מייצגת הוא בפונט italic

1. `FilterableProductTable` (**כתום**) מכיל את כל תוכלת הדוגמה:
2. `SearchBar` (**כחול**) קלט המשתמש:
3. `ProductTable` (**ירוק**) מציג ומסנן את המידע על סמך קלט המשתמש:
4. `ProductCategoryRow` (**טורקי**) מציג כותרת לכל קטגוריה:
5. `ProductRow` (**אדום**) מציג שורה לכל מוצר:

נראה שכותרת הטליה (שמכילה את כותרות השם והמחיר) היא לא קומפוננטה משלها. אפשר אם נסתכל על `ProductTable` בగל `ProductTableHeader` (אם היה מוסיפים חישוב) ומשתמש (בבחירה). בדוגמה זאת, השארנו אותו חלק מ לעמודת זאת, אם הכותרת גדולה ונהיית מסובכת) לדוגמה `ProductTable`. שזה חלק מרינדרו המציג, שהוא עבדת הקומפוננטה עבשו לאחר `ProductTableHeader`). אם היינו מוסיפים חישוב (, זה בהחלט עלה צורך לפצל אותה לקומפוננטה משלה שזיהינו את הקומפוננטות בדוגמה שלנו, נסדר אותן בהיררכיה. קומפוננטות שנמצאות בתוך קומפוננטות אחרות צרכות להופיע מתחתיהן בהיררכיה:

בסוף השלב זהה, יהיה לנו ספירה של קומפוננטות שימוש שמרנדראות את מודל המידע. הקומפוננטות יכולו רק מתודות `render` את מודל (`FilterableProductTable`) מכיוון שזוהי גרסה סטטית של היישום. הקומפוננטה בראש ההיררכיה () שוב, משק המשתמש יעדכן. זה () `ReactDOM.render` שיעבור לה. אם נבצע שינוי למודל המידע ונקרא ל `comkhamidu` בפושט לראות איך משק המשתמש מעודכן והיכן לבצע שינויים מסוימים מכיוון ששם דבר מסובך לא מתבצע. **העברת המידע בכוון** לשארה הכל בצורה מודולרית ומהירה **React אחד** של

קרא עוד [בטייעד של ריאקט](#) אם תצערך עזרה בביצוע שלב זה.

props ו-state: הפגיעה קצרה

זה חשוב להבין את הבדלים בין השניים. קרא עוד [בטייעד של ריאקט](#) : **props** יש שני סוגי של 'מודלי' מידע בReact : **state** ו-**props**? האם גם [שאלות נפוצות: מה ההבדל בין state ו-props?](#)

של state של state שלישי: זיהוי הייצוג המינימלי (אך מלא) של המשמש

עושים זאת React. על מנת שמשק המשתמש יהיה אינטראקטיבי, נדרש שיהיה לנו את היכולת לשנות את מודל המידע שלנו עם **state**.

שהישום דורש. המפתח פה הוא **state** כדי לבנות את היישום בצורה נכונה, נדרש לחשב תחילת על הכמות המינימלית של שהישום צריך וnochshav state שאומר 'אל תמחזר קוד'. נחשוב על הכמות המינימלית ביותר של **DRY: Don't Repeat Yourself** כל דבר אחר שהישום צריך לפי דרישת. לדוגמה, אם אנו בונים יישום של רשימה מטלות, יהיה לנו מערכת של מטלות; לא נשאר בשביל הכמות מטלות. במקום זה, כשנרצה לрендר את מספר המטלות, פשוט ניקח את אורך המערך ונציג אותו state משתנה ב:

נראה את פיסות המידע בישום שלנו, יש לנו את:

- הרשימה המקורי של המוצרים
- מילוט החיפוש שהמשמש הקליד
- ערך תיבת הסימון
- הרשימה המסוננת של המוצרים

: נעשה זאת באמצעות שאלות 3 על כל פיסת מידע state. על כל פיסת מידע נגענו על כל אחד ונחשב איזה פיסת מידע היא ה

1. props אם כן, זה לא?props האם המידע עובר מהורה באמצעות
2. state. האם זה נשאר קבוע לאורך זמן? אם כן, זה לא.
3. state. בקומפוננטה שלנו? אם כן, זה לא props אחר או state האם אפשר לחשב את זה לפי

טקסט החיפוש ותיבת הסימון נראים כמתאים להיות state. אז ה-props הרשימה המקורי עוברת באמצעות state מכיוון שהיא state ממכיוון שהם משתנים לאורך זמן ולא מושפעים ממנו. לבסוף, הרשימה המסוננת של המוצרים היא לא מושפעת מהרשימה המקורי של המוצרים, טקסט החיפוש וערך תיבת הסימון (מסומן או לא)

: שלנו הוא state אז הגענו למסקנה שה

- טקסט החיפוש שהמשמש הקליד
- ערך תיבת הסימון

שלב רביעי: זיהוי המיקום המתאים לstate

```

class ProductCategoryRow extends React.Component {
  render() {
    const category = this.props.category;
    return (
      <tr>
        <th colSpan="2">
          {category}
        </th>
      </tr>
    );
  }
}

class ProductRow extends React.Component {
  render() {
    const product = this.props.product;
    const name = product.stocked ?
      product.name :
      <span style={{color: 'red'}}>
        {product.name}
      </span>;
    return (
      <tr>
        <td>{name}</td>
        <td>{product.price}</td>
      </tr>
    );
  }
}

```

Resources View Compiled

של היחסום שלנו. עכשו אנחנו צריכים לזהות איזה קומפוננטה משתנה, או לנקחת stateזיהינו מה המרכיבים המינימליים של state. בעלות על state.

הכל סובב סביב העברת מידע מטעה בהיררכיית הקומפוננטות. זה לעיתים לא ברור מיד איזה קומפוננטה הכי Reactאקרו: בכאן נעבור על השלבים הבאים על React, **זה לעומת זאת קרובות החלק הכי מתאגר למתחילה ב state.** מתאימה לאחסן בתוכה: מנת להבין:

בשימוש state לכל פיסת:

- **ונכוןstate נמצוא אם יש קומפוננטה אחרת מעלה בהיררכיה או קומפוננטה דומה שמתאימה לאחסן את state.**
- **וכניס state ניצור אותה שכל מטרתה היא לאחסן את state, לא נמצא קומפוננטה שמתאימה לאחסן את state.**
- **ואו שהיא מאפשרת איפשהו בהיררכיה מעלה קומפוננטה אחרת שימושstate ב state.**

נשתמש בשלבים אלה ביחסום:

- **צריך להציג את טקסט החיפוש ותיבת הסימון SearchBar state לסדרה לשיiftstate לפיה רשימה המוצרים נמצאת state.**
- **ואופן עקרוני זה הגיוני שהטקסט המסונן והערך המסומן 'יחיו' בתוך FilterableProductTable.**

אפילו שזה נשמע מסובך, זה בסך הכל מספר מצומצם של שורות קוד. והיתרונות הוא שהוא מציג באופן מפורש איך המידע מועבר בישום.

וואה!

אמנם זה יכול לצורך React. הכוונה של המדריך הזה היא לנתת לכם רעיון על איך לחשב על בניית קומפוננטות ויישומים עם יותר כתיבה ממה שאתם רגילים, אבל זכרו שקוד נקרא יותר מאשר הוא נכתב, וזה נוח מאד לקרוא את הקוד המפורש והמדוורי שכתבנו. כשתחילה לבנות ספריות גדולות של קומפוננטות, אתם תעריכו את הפשטות והמדוירות, ועם שימוש חואן 😊. בקד שאתם רושמים, שורות הקוד שלכם יתחלו להתכווץ.

[Go to TOC](#)

Typechecking With PropTypes

Note:

`React.PropTypes` has moved into a different package since React v15.5. Please use [the prop-type library instead](#).

We provide [a codemod script](#) to automate the conversion.

As your app grows, you can catch a lot of bugs with typechecking. For some applications, you can use JavaScript extensions like [Flow](#) or [TypeScript](#) to typecheck your whole application. But even if you don't use those, React has some built-in typechecking abilities. To run typechecking on the props for a component, you can assign the special `propTypes` property:

```
import PropTypes from 'prop-types';

class Greeting extends React.Component {
  render() {
    return (
      <h1>Hello, {this.props.name}</h1>
    );
  }
}

Greeting.propTypes = {
  name: PropTypes.string
};
```

In this example, we are using a class component, but the same functionality could also be applied to function components, or components created by `React.memo` or `React.forwardRef`.

`PropTypes` exports a range of validators that can be used to make sure the data you receive is valid. In this example, we're using `PropTypes.string`. When an invalid value is provided for a prop, a warning will be shown in the JavaScript console. For performance reasons, `propTypes` is only checked in development mode.

PropTypes

Here is an example documenting the different validators provided:

```
import PropTypes from 'prop-types';

MyComponent.propTypes = {
  // You can declare that a prop is a specific JS type. By default, these
  // are all optional.
  optionalArray: PropTypes.array,
  optionalBool: PropTypes.bool,
  optionalFunc: PropTypes.func,
  optionalNumber: PropTypes.number,
  optionalObject: PropTypes.object,
```

```

optionalString: PropTypes.string,
optionalSymbol: PropTypes.symbol,

// Anything that can be rendered: numbers, strings, elements or an array
// (or fragment) containing these types.
optionalNode: PropTypes.node,

// A React element.
optionalElement: PropTypes.element,

// A React element type (ie. MyComponent).
optionalElementType: PropTypes.elementType,

// You can also declare that a prop is an instance of a class. This uses
// JS's instanceof operator.
optionalMessage: PropTypes.instanceOf(Message),

// You can ensure that your prop is limited to specific values by treating
// it as an enum.
optionalEnum: PropTypes.oneOf(['News', 'Photos']),

// An object that could be one of many types
optionalUnion: PropTypes.oneOfType([
  PropTypes.string,
  PropTypes.number,
  PropTypes.instanceOf(Message)
]),

// An array of a certain type
optionalArrayOf: PropTypes.arrayOf(PropTypes.number),

// An object with property values of a certain type
optionalObjectOf: PropTypes.objectOf(PropTypes.number),

// An object taking on a particular shape
optionalObjectWithShape: PropTypes.shape({
  color: PropTypes.string,
  fontSize: PropTypes.number
}),

// An object with warnings on extra properties
optionalObjectWithStrictShape: PropTypes.exact({
  name: PropTypes.string,
  quantity: PropTypes.number
}),

// You can chain any of the above with `isRequired` to make sure a warning
// is shown if the prop isn't provided.
requiredFunc: PropTypes.func.isRequired,

// A required value of any data type
requiredAny: PropTypes.any.isRequired,

// You can also specify a custom validator. It should return an Error
// object if the validation fails. Don't `console.warn` or throw, as this
// won't work inside `oneOfType`.
customProp: function(props, propName, componentName) {
  if (!/matchme/.test(props[propName])) {
    return new Error(
      'Invalid prop `' + propName + '` supplied to' +
      ' `' + componentName + '`. Validation failed.'
    );
}

```

```

    }
};

// You can also supply a custom validator to `arrayOf` and `objectOf`.
// It should return an Error object if the validation fails. The validator
// will be called for each key in the array or object. The first two
// arguments of the validator are the array or object itself, and the
// current item's key.
customArrayProp: PropTypes.arrayOf(function(propValue, key, componentName,
location, propFullName) {
  if (!/matchme/.test(propValue[key])) {
    return new Error(
      'Invalid prop `' + propFullName + '` supplied to' +
      ' `' + componentName + '`. Validation failed.'
    );
  }
});

```

Requiring Single Child

With `PropTypes.element` you can specify that only a single child can be passed to a component as children.

```

import PropTypes from 'prop-types';

class MyComponent extends React.Component {
  render() {
    // This must be exactly one element or it will warn.
    const children = this.props.children;
    return (
      <div>
        {children}
      </div>
    );
  }
}

MyComponent.propTypes = {
  children: PropTypes.element.isRequired
};

```

Default Prop Values

You can define default values for your `props` by assigning to the special `defaultProps` property:

```

class Greeting extends React.Component {
  render() {
    return (
      <h1>Hello, {this.props.name}</h1>
    );
  }
}

// Specifies the default values for props:
Greeting.defaultProps = {
  name: 'Stranger'
};

```

```
// Renders "Hello, Stranger":
const root = ReactDOM.createRoot(document.getElementById('example'));
root.render(<Greeting />);
```

If you are using a Babel transform like `plugin-proposal-class-properties` (previously `plugin-transform-class-properties`), you can also declare `defaultProps` as static property within a React component class. This syntax has not yet been finalized though and will require a compilation step to work within a browser. For more information, see the [class fields proposal](#).

```
class Greeting extends React.Component {
  static defaultProps = {
    name: 'stranger'
  }

  render() {
    return (
      <div>Hello, {this.props.name}</div>
    )
  }
}
```

The `defaultProps` will be used to ensure that `this.props.name` will have a value if it was not specified by the parent component. The `propTypes` typechecking happens after `defaultProps` are resolved, so typechecking will also apply to the `defaultProps`.

Function Components

If you are using function components in your regular development, you may want to make some small changes to allow PropTypes to be properly applied.

Let's say you have a component like this:

```
export default function HelloWorldComponent({ name }) {
  return (
    <div>Hello, {name}</div>
  )
}
```

To add PropTypes, you may want to declare the component in a separate function before exporting, like this:

```
function HelloWorldComponent({ name }) {
  return (
    <div>Hello, {name}</div>
  )
}

export default HelloWorldComponent
```

Then, you can add PropTypes directly to the `HelloWorldComponent`:

```
import PropTypes from 'prop-types'

function HelloWorldComponent({ name }) {
  return (
    <div>Hello, {name}</div>
  )
}
```

```
<div>Hello, {name}</div>
}

HelloWorldComponent.propTypes = {
  name: PropTypes.string
}

export default HelloWorldComponent
```

קומפוננטות לא מבוקرات

ברוב המקרים, אנו ממליצים להשתמש בקומפוננטות מבוקرات למימוש טפסים. בקומפוננטה מבוקרת, נתוני הטופס מנוהלים עצמו DOM-האלטרנטיביה היא קומפוננטה לא מבוקרת, איפה שנתוני הטופס מנוהלים על ידי ה React על ידי קומפוננטה.

כדי לקבל את `ref`-אתה יכול להשתמש ב, לכל עדכון `state` לכל עדכון event handler כדי לכתוב קומפוננטה לא מבוקרת, במקומות הראשונים ערכוי הטופס מה-DOM.

לדוגמא, הקוד הזה מקבל שם ייחיד בקומפוננטה לא מבוקרת:

```
class NameForm extends React.Component {
  constructor(props) {
    super(props);
    this.handleSubmit = this.handleSubmit.bind(this);
    this.input = React.createRef();
  }

  handleSubmit(event) {
    alert('A name was submitted: ' + this.input.current.value);
    event.preventDefault();
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          Name:
          <input type="text" ref={this.input} />
        </label>
        <input type="submit" value="Submit" />
      </form>
    );
  }
}
```

[נסו זאת ב-CodePen](#)

קוד מתו React ולא לפעמים זה קל יותר לשלב, DOM-מכיון קומפוננטה לא מבוקרת שומרת את מקור האמת ב שימושים בקומפוננטה לא מבוקרת. זה גם יכול להיות מעט קוד אם אתה רוצה להיות מהיר ומלוכך, אחרת, אתה צריך קומפוננטה מבוקרת.

אם זה עדין לא ברוראיזה סוג של קומפוננטה אתה צריך להשתמש בסיטואציה מסוימת, אתה עשוי למצוא את [המאמר הזה](#) על [קלט מבוקר נגד לא מבוקר](#) מאוד שימושי.

ערכי בירית מחדל

עם קומפוננטה לא מבוקרת, DOM-באלמנטי טופס ידרשו את הערך ב `value` התוכונה, ברנדור מחזיר החיים של יספק את הערך הראשוני, אבל ישאיר את העדכנים הבאים לא מבוקרים. כדי לטפל במקרה-React-לעתים קרובות תרצה ש אחרי שקומפוננטה `value` במקום `defaultValue` שינוי הערך של `value`. אתה יכול לציין את תוכנות DOM-עלתה לא יגרום לעדכון של הערך ב

```

render() {
  return (
    <form onSubmit={this.handleSubmit}>
      <label>
        Name:
        <input
          defaultValue="Bob"
          type="text"
          ref={this.input} />
      </label>
      <input type="submit" value="Submit" />
    </form>
  );
}

```

נובוט למשתמשים לבחר קובץ אחד או יותר מהmerican שליהם לעלות לשרת או לתפעול -HTML , `<input type="file">` ב-React, `<input type="file" />` לא תומכים ב-`defaultChecked` , `<select>` ב-`<checkbox>` , `<input type="radio">` ב-`defaultSelected` , `<textarea>` ב-`defaultValue` .

ה-tag `file input`

נותן למשתמשים לבחור קובץ אחד או יותר מהmerican שליהם לעלות לשרת או לתפעול -HTML , `<input type="file">` ב-HTML , `<input type="file">` ב-React, `<input type="file" />` ב-JavaScript דרך [File API](#).

```
<input type="file" />
```

הוא תמיד קומפוננטה לא מבוקרת בגלל שהערך שלו ניתן על ידי המשתמש, ולא ב-React, `<input type="file" />` לא תומך ב-`defaultSelected` , `<select>` ב-`<checkbox>` , `<input type="radio">` ב-`defaultChecked` , `<checkbox>` ב-`checked` .

כדי לגשת `ref` כדי לתקשר עם קבצים. הדוגמא הבאה מראה איך ליצור API File-`node`-DOM-ל-`ref` באמצעות `onSubmit` handler:

```
embed:uncontrolled-components/input-type-file.js
```

[Go to TOC](#)

Web Components

React and [Web Components](#) are built to solve different problems. Web Components provide strong encapsulation for reusable components, while React provides a declarative library that keeps the DOM in sync with your data. The two goals are complementary. As a developer, you are free to use React in your Web Components, or to use Web Components in React, or both.

Most people who use React don't use Web Components, but you may want to, especially if you are using third-party UI components that are written using Web Components.

Using Web Components in React

```
class HelloMessage extends React.Component {
  render() {
    return <div>Hello <x-search>{this.props.name}</x-search>!</div>;
  }
}
```

Note:

Web Components often expose an imperative API. For instance, a `video` Web Component might expose `play()` and `pause()` functions. To access the imperative APIs of a Web Component, you will need to use a `ref` to interact with the DOM node directly. If you are using third-party Web Components, the best solution is to write a React component that behaves as a wrapper for your Web Component.

Events emitted by a Web Component may not properly propagate through a React render tree. You will need to manually attach event handlers to handle these events within your React components.

One common confusion is that Web Components use "class" instead of "className".

```
function BrickFlipbox() {
  return (
    <brick-flipbox class="demo">
      <div>front</div>
      <div>back</div>
    </brick-flipbox>
  );
}
```

Using React in your Web Components

```
class XSearch extends HTMLElement {
  connectedCallback() {
    const mountPoint = document.createElement('span');
    this.attachShadow({ mode: 'open' }).appendChild(mountPoint);

    const name = this.getAttribute('name');
```

```
const url = 'https://www.google.com/search?q=' + encodeURIComponent(name);
const root = ReactDOM.createRoot(mountPoint);
root.render(<a href={url}>{name}</a>);
}
customElements.define('x-search', XSearch);
```

Note:

This code **will not** work if you transform classes with Babel. See [this issue](#) for the discussion. Include the `custom-elements-es5-adapter` before you load your web components to fix this issue.

Colophon

This book is created by using the following sources:

- React - עברית -
- GitHub source: reactjs/he.reactjs.org/content
- Created: 2022-11-28
- Bash v5.2.2
- Vivliostyle, <https://vivliostyle.org/>
- By: @shinokada
- Viewer: <https://read-html-download-pdf.vercel.app/>
- GitHub repo: <https://github.com/shinokada/markdown-docs-as-pdf>
- Viewer repo: <https://github.com/shinokada/read-html-download-pdf>