VIRTUAL DOM

בתור מפתח ריאקט אתה יכול לבנות אפליקציה שלמה מבלי לדעת כלום על ה-Virtual DOM  אבל כמובן זה לא אומר שזה לא חשוב אלא ההפך כדי להגיע לרמות הגבוהות באמת נורא חשוב להבין לא רק את ה ״ public API" אלא גם מה עומד מאחוריו ואיך ריאקט באמת עובדת.

ככל הנראה שמעתם מתישהו על המונח Virtual DOM  ואם אתם מפתחים בריאקט אז אני מקווה שאתם יודעים שריאקט משתמשת ב-Virtual DOM ולא מעדכנת ישירות את ה-DOM אבל מה זה אומר בעצם?

## Real DOM

 DOM הוא ראשי תיבות של Document Object Model. במילים פשוטות ה- DOM מייצג את המבנה(UI) של האפליקציה שלנו בצורה טקסטואלית באמצעות צמתים(nodes) וכאובייקטים. בכל פעם שיש שינוי במבנה של האפליקציה ה- DOM מתעדכן כדי שיוכל לייצג את אותו שינוי. כדי לבצע שינויים על ה- DOM בדרך כלל נשתמש בג׳אווהסקריפט(removeChild לדוגמא או שנשנה את ה-state בריאקט).ביצוע שינויים תכופים ישירות על ה- DOM יפגע בביצועים של האפליקציה שלנו ויהפוך אותה לאיטית.

## למה מניפולציה ישירות על ה-DOM נחשבת איטית?

זכרו: עדכון ה-DOM **אינו** איטי! ה- DOM מיוצג כעץ מבנה נתונים,לעדכן את ה DOM זה ממש כמו לעדכן כל אובייקט ג׳אווהסקריפט. אז מה בדיוק הופך את עדכון ה- Real DOM לאיטי?  
כאשר אנו מבצעים שינויים שאמורים לגרום לרנדור מחדש של ה -DOM בעצם מתבצע גם חישוב מחדש של כל ה-CSS וכל השכבות(layouts)שלנו מחדש,אלגוריתם זה משפיע רבות על הביצועים וזה מה שהופך אותו לאיטי. (re-painting).

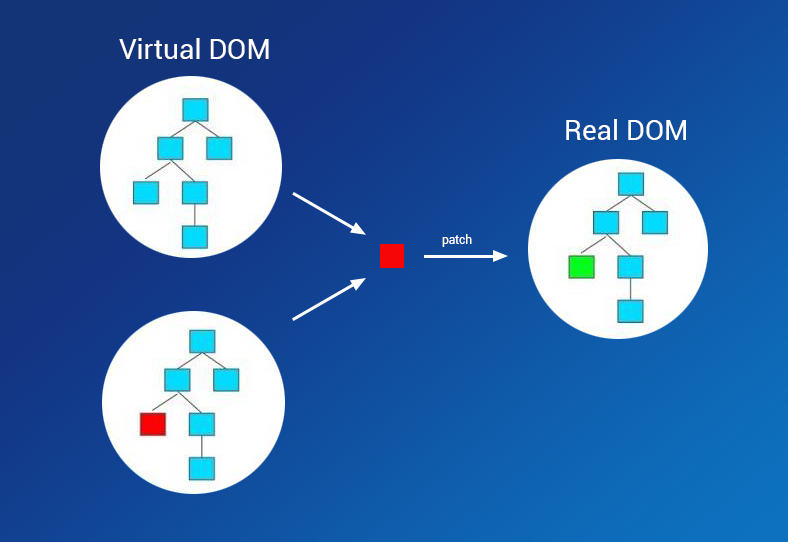
בעיקר אפליקציות SPA שמורכבות מהמון קופמפוננטות ושינויים רבים ב-state של האפליקציה יחוו בעיות ביצועים, מכיוון שככל שיש לנו יותר קומפוננטות כולן יצטרכו להתרנדר מחדש בכל עדכון של ה-DOM. אפשר להסתכל על זה ככה: ככל שה-DOM גדול יותר ותכיפות השינויים גדולה יותר עדכוני ה-DOM יהיו יקרים יותר .

## Virtual DOM

ה-Virtual DOM לא הומצא על ידי ריאקט, אך היא משתמשת בקונספקט שלו ומספקת לנו את זה נייטיב באפליקציה שלנו. אז מהו ה-Virtual DOM?! הוא בעצם ייצוג בזיכרון, אובייקט ג׳אווהסקריפט קל משקל שהוא העתק של ה-Real DOM

**איך ה- Virtual DOM מהיר יותר?**

בכל פעם שה-state של האפליקציה שלנו משתנה ריאקט יוצרת את כל ה- DOM הווירטואלי כולו מאפס. יצירת עץ אובייקטים שלם היא מהירה מאוד ולכן זה לא משפיע על הביצועים שלנו. בעצם בכל זמן נתון ריאקט מחזיקה שני Virtual DOM, אחד עם המצב המעודכן והשני עם המצב הקודם.  
לאחר שהיא מעדכנת את הקומפוננטות הדרושות  ב- Virtual DOM. ריאקט משווה בין שני ה-virtual DOM שהיא מחזיקה באמצעות אלגוריתם (diff algorithm) כדי למצוא את מספר הצעדים המינימלי לעדכון ה-Real DOM ולאחר מכן מעדכנת ב- Real DOM רק את השינויים הנדרשים ולא את כל ה -DOMתהליך זה נקרא [Reconciliation](https://reactjs.org/docs/reconciliation.html) שיטה זו בעצם מונעת מעבר על כל מבנה העץ וגורמת לשיפור ביצועים משמעותי.



באמצעות ה- ״diff algorithm״ ריאקט יודעת למצוא את מספר הצעדים המינימלי שצריך כדי לעדכן את ה-Real DOM. לאחר מכן היא מבצעת את אותם צעדים בלולאה אחת( event loop) מבלי לגרום לצביעה מחודשת (re-painting)של ה- Real DOM. אם יש יותר מאלמנט אחד שמתעדכן ריאקט תדע לחכות לסיום האיבנט לופ ורק לאחר מכן תעדכן את ה- Real DOM.   
לאחר ביצוע כל השלבים ריאקט תצבע מחדש את ה-  Real DOM. המשמעות היא שיש בדיוק פעם אחת שבה מציירים את ה- Real DOM. כך כל תהליך ה-layouts יפעל רק פעם אחת לעדכון ה- Real DOM.

משמעות הדבר היא כי עדכונים ל- Real DOM נשלחים ב״חבילה״ אחת במקום לשלוח עדכונים עבור כל שינוי ב-state.  
הצביעה מחדש של האפליקציה (repainting) היא החלק היקר ביותר מבחינת ביצועים וריאקט דואגת לזה שה- Real DOM יקבל חבילה אחת כדי לצבוע מחדש את האפליקציה.

## לסיכום

* מניפולציות תכופות על ה- Real DOM הן יקרות וכבדות מבחינת ביצועים.
* Virtual DOM הוא ייצוג וירטואלי של ה-  Real DOM
* כאשר אנו עושים שינוי ב- state של האפליקציה שלנו ה –Virtual DOM מתעדכן והוא מושווה לגרסא הקודמת שלו.לאחר מכן ה- Virtual DOM שולח ״חבילת עדכון״ ל- Real DOM כדי לעדכן את את האפליקציה.
* ריאקט משתמשת באלגוריתם השוואה( diff algorithm ) יעיל כדי להשוות בין גרסאות של ה- Virtual DOM
* ריאקט משתמשת ב- Virtual DOM כדי לשפר את הביצועים של האפליקציה.

HOOKS

## מה זה ריאקט הוקס

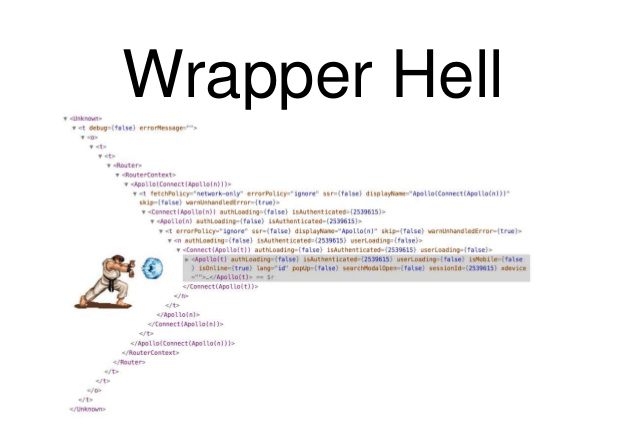
הוקס הגיחו לאוויר העולם בגרסא 16.8 של ריאקט וכעת נותנים לנו יכולות שהיו לנו עד היום בקומפוננטות שהן קלאס גם בפונקציות

באופן די מהיר לאחר שהוקס הגיעו לווב הוא שוחרר גם לריאקט נייטיב החל מגרסא 0.59  וכעת גם בפיתוח למובייל אנחנו יכולים להנות מהפיצר הנפלא הזה.

לדעת רבים ריאקט הוקס מבשר את ״מותם״ של הקלאסים בריאקט אך במדריך הרשמי של ריאקט פרסמו כי אין חשש מלהמשיך להשתמש בקלאסים וכי הם לא מתכוונים להפסיק לתמוך בהם.

## למה בעצם ריאקט הוקס

כיום, יש הרבה דרכים לעשות שימוש חוזר בלוגיקה בריאקט. אנחנו יכולים לכתוב פונקציות פשוטות ולקרוא להם בכל פעם שנרצה לממש אותן. אנחנו יכולים גם לכתוב קומפוננטות (פונקציות או קלאסים).  לקלאסים יש יותר כח אבל הם צריכים לדאוג לרנדור של ממשק המשתמש, מה שמביא אותי לחסרון שלהם. לא נוח לשתף באמצעותם לוגיקה( stateful logic) כדי לשתף לוגיקה בין קלאסים נמצא את עצמינו משתמשים ב- Mixins ,מעבירים פרופס או עושים שימוש ב HOC.   
פתרונות אלה דורשים מאיתנו בעצם לבנות מחדש/לשנות את הקומפוננטה שלנו כאשר נרצה להשתמש בה, כל הדברים הנל יגרמו לקוד שלנו להיות מסורבל יותר, קשה יותר להבנה וקשה יותר לדבג אותו. המנוסים יותר בינינו בטח מכירים את המושג ״wrapper hell״. עטיפה  של קומפוננטה ב HOC תגרום ל-״בלאגן״ וקושי רב בהתמצאות ב react devtools. חסרון נוסף הוא שלא נוכל לדעת איזה מידע אכן הגיע מה HOC ואיזה מידע הועבר לקומפוננטה. הוקס מאפשרים לנו לחלץ לוגיקה מקומפוננטות כך שנוכל להשתמש בהם שוב ושוב בכל מקום שנרצה בלי לשנות את מבנה ההיררכיה.



Bottom of Form

**קומפוננטות מורכבות הופכות עם הזמן קשות להבנה-**לקומפוננטות בעלות  state שעד היום היו קלאסים יש נטייה להיות מראש גדולות מכיוון שהן מחזיקות את כל הניהול הלוגי בתוכן ומעבירות מידע לפונקציות קטנות שתפקידן להציג מידע. בנוסף מוצר תמיד גודל תמיד יש דרישה לפיתוח של פיצ׳ר חדש, הוספת יכולות או שינויים שנדרשו עם הזמן ומכילים לוגיקה נוספת ב- state.  
מה שהופך אותן למסובכות להבנה ככל שהן גדולות הם ה lifeCycle methods(מחזורי חיים)

כל מחזור חיים של קומפוננטה ככל הנראה יכיל לוגיקה של מספר דברים אשר אין ביניהם קשר בכלל! לדוגמא ארצה לפנות לשרת לקבלת מידע בכל פעם שהקומפוננטה נטענת  אשתמש ב  componentDidMount.  
עם זאת ב- componentDidMount יכול להיות עוד הרבה קוד שלא קשור לפנייה לשרת כגון האזנה לאירועים או אפילו מספר פניות לשרת שלא קשורות אחת לשנייה. בהרבה מקרים לא נוכל לשבור את הקומפוננטה כי הלוגיקה של ה state צריכה להשמר וכבר מועברת להמון קומפוננטות אחרות.  
כדי לפתור את הבעיה הזאת הוקס מאפשר לנו לפצל קומפוננטה אחת לפונקציות קטנות יותר על סמך ההגיון הלוגי שמשותף להם.

**קלאסים בריאקט-**  
לפי הצוות של ריקאט הם מצאו שקלאסים יכולים להיות מכשול גדול בלמידת ריאקט. צריך להבין איך קלאסים עובדים בגאווהסקריפט ואת השוני בהתנהגות ביניהן לבין קלאסים ברב השפות אני נוטה **לא** להסכים איתם בנושא הזה מכיוון שלדעתי לא נכון להכנס וללמוד ריאקט אם אתה לא מבין גאווהסקריפט.

מפתחי גאווהסקריפט מתחילים וגם מנוסים נוטים להתבלבל בשאלה. מה הthis (context)של אותה מתודה שרצה.  
עם הזמן גדל השימוש במתודה bind בקלאסים, דוגמה טובה לכך הינה events שמשנים את הcontext. ולכן היה צורך להחזיר ולשייך את ה הcontext של המתודה מחדש לקלאס ולא ל-event

הצוות של ריאקט תמיד מנסה לשפר את הספריה ואת זמני הטעינה. בניסויים שהם ערכו לטענתם הם הציגו תוצאות הרבה יותר טובות באמצעות שימוש בפונקציות לעומת קלאסים. בנוסף הם אומרים ש- minify לא עובד בצורה כל כך יעילה עם קלאסים ו- Hot Reload לא עובד תמיד בצורה טובה ואמינה. הם רוצים לבנות  API שיהיה אופטימלי מבחינת אופטימיזציה ככל הניתן וכאן הוקס נכנס שוב.  
כל אלה בעצם הובילו לפיתוח של הוקס.

חשוב לדעת-עבודה עם הוקס לא תשתף state בין פונקציות אלה רק את הלוגיקה של העבודה עם ה- state.   
– לא כל הפיצרים של קלאס נתמכים(getSnapshotBeforeUpdate and componentDidCatch) אבל הצות של ריאקט אומרים שהם עובדים על זה ובעתיד הקרוב נוכל לעשות את כל מה שעשינו בקלאסים גם בפונקציות בעזרת הוקס.

### הגיע הזמן לקצת קוד

בחלק זה נתעסק ביצירת state בתוך פונקציה, בכל שאר הפיצרים של הוקס נעסוק בהמשך.  
בדוגמא הבאה ניצור counter שייצג את מספר הפעמים שלחצנו על כפתור מסויים  
אז איך מתחילים:  
תחילה נצטרך לייבא מתוך ריקאט את היכולת להשתמש ב useState.

import React, { useState } from 'react';

לאחר שייבאנו את Usestate נוכל להתחיל להשתמש בו בתוך הפונקציה כאשר נרצה לייצר state לוקאלי.  
useState מחזיק 2 ערכים: ערך נוכחי של הstate ופונקציה שתיתן לנו את האופציה לעדכן את ה state. מכיוון שאנחנו עושים destructuring למערך אז אין בעצם משמעות לשמות של הערכים שנבחר, נוכל לבחור את שמות המשתנים לפי

|  |
| --- |
| const [count, setCount] = useState(0); |

מה שנח ונכון עבורינו.  
הערך שנשים בתוך הסוגריים לאחר Usestate הוא הערך ההתחלתי של ה state אין בעצם הגבלה של סוג המשתנה אותו נוכל להזין(סטרינג,מספר מערך).

אז מתי בעצם ריאקט מרנדרת לנו את הקומפוננטה?

בכל פעם שקוראים לפונקציית העדכון, הקומפוננטה תתרנדר מחדש. ובלי קשר בכל פעם שהקומפוננטה מתרנדרת מחדש useState יחזיר את הערך הנוכחי של ה- state.  
מכיוון שבגאווהסקריפט אפשר לכתוב פונקציות שמכילות פונקציות נוכל לקרוא לפונקציה שמעדכנת את ה- state מכל מקום.  
כמו כן נוכל להשתמש ב useState מספר פעמים וכל אחד מהם יכיל את ה- state שהכרזנו עליו, לדוגמא:

const [age, setAge] = useState(42(;  
const [fruit, setFruit] = useState(‘banana’);

|  |
| --- |
| import React, { useState } from 'react'; |
|  |
| const Counter = () => { |
| *// Declare a new state variable, which we'll call "count"* |
| const [count, setCount] = useState(0); |
|  |
| return ( |
| <div> |
| <p>You clicked {count} times</p> |
| <button onClick={() => setCount(count + 1)}> |
| Click me |
| </button> |
| </div> |
| ); |
| } |
|  |

## useEffect Hook

בקומפננטה מסוג קלאס קיימים שלושה מחזורי חיים המוחלפים על ידי השימוש בהוק useEffect- componentDidMount componentDidUpdate, ו componentWillUnmount .  
היתרון של ה useEffect הוא היכולת לקחת את שלושת מחזורי החיים הנל ולממש אותם באמצעותו.

לדוגמא ניתן לממש בעזרת useEffect:

* פנייה לשרת לקבלת מידע(fetch)
* שינויים ב- DOM
* subscription

חשוב לשים לב, שכברירת מחדל useEffect ירוץ לאחר כל רנדור של הפונקציה, כולל הרנדור הראשוני(הדבר ניתן לשינוי, ויוסבר בהמשך).

ועכשיו קצת קוד.. קומפונטטה מסוג קלאס ומסוג פונקציה עם  counter שמעדכנן את כותרת הדף(document title) מיד לאחר שריאקט מבצעת את השינויים ב- DOM. בהקשר זה יש לשים לב, לשכפול הקוד בקלאס בשני מחזורי החיים.  
המצב האופטימלי הינו שזה יקרה אחרי כל רנדור – אך מחזור חיים כזה אינו קיים בקלאס.  
useEffect מגדיר לריאקט שהקומפוננטה צריכה לבצע את שכתבנו בתוכה לאחר הרנדור. ריאקט זוכרת את הפונקציה שנכתבה ותקרא לה לאחר שתסיים לעשות את כל השינויים ב DOM.​

**קומפננטה מסוג קלאס**

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

**קומפוננטה מסוג פונקציה עם useEffect**

|  |
| --- |
| import React, { useState, useEffect } from 'react'; |
|  |
| 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> |
| ); |
| } |

## מתי useEffect ירוץ?

כפי שכתבתי כבר, ברירת המחדל של useEffect היא לרוץ ברנדור הראשוני ובכל פעם שהקומפוננטה מתרנדרת. **useEffect יודע לקבל משתנה שני מסוג מערך**,מערך זה יקבע מתי useEffect יתעדכן(המקביל ל- componentDidUpdate). במידה ויש צורך ש useEffect ירוץ רק ב- on mount נעביר מערך ריק.

|  |
| --- |
| *// only run on mount. pass an empty array* |
| useEffect(() => { |
| *// only runs once* |
| }, []); |

דוגמא למצב שנרצה ש useEffect ירוץ רק פעם אחת היא כאשר נרצה לקרוא ל API רק שהקומפוננטה נטענת

|  |
| --- |
| import React, { useState } from 'react'; |
|  |
| function GetGitHubUsers() { |
| const [users, setUsers] = useState([]); |
|  |
| useEffect(() => { |
| fetch('https://api.github.com/users') |
| .then(response => response.json()) |
| .then(data => { |
| setUsers(data); *// set users in state* |
| }); |
| }, []); *// empty array because we only run once* |
| } |

במידה ויושם ערך בתוך המערך אז useEffect ירוץ בכל פעם שיהיה שינוי בערך המשתנה, כפי שניתן לראות בדוגמה המוצגת.

|  |
| --- |
| *// only run if count changes* |
| useEffect( |
| () => { |
| *// run here if count changes* |
| }, |
| [count] |
| ); |

Bottom of Form

## איך נשתמש ב componentWillUnmount?

ישנם מספר מקרים הדורשים ניקוי המונע דליפות זיכרון. (memory leak)  
לדוגמא פניות API , הפסקה של אינטרוולים וכו..

**subscription של קומפוננטה ללא הוקס:**

|  |
| --- |
|  |
| import React, { Component } from "react"; |
|  |
| class MyComponent extends UseEffectExample { |
| constructor(props) { |
| super(props); |
| this.state = { |
| width: window.innerWidth; |
| }; |
| } |
|  |
| componentDidMount() { |
| window.addEventListener("resize", this.handleResize); |
| } |
|  |
| componentWillUnmount() { |
| window.removeEventListener("resize", this.handleResize); |
| } |
|  |
| handleResize = () =>{ |
| this.setState({ width: window.innerWidth }); |
| } |
|  |
| render() { |
| return ( |
| <h1>{this.state.width}</h1> |
| ); |
| } |
| } |

בכדי להריץ משהו לפני שהקומפוננטה unMount יש צורך בהחזרת פונקציה מתוך useEffect.

כעת נבחן את אותה הדוגמא תוך שימוש בהוקס.

|  |
| --- |
| import React, { useState, useEffect } from "react"; |
|  |
| const MyComponent = () => { |
| const [width, setWidth] = useState(window.innerWidth); |
|  |
| useEffect(() => { |
| const handleResize = () => setWidth(window.innerWidth); |
| window.addEventListener("resize", handleResize); |
|  |
| return () => { |
| window.removeEventListener("resize", handleResize); |
| } |
| }, []); |
|  |
| render() { |
| return ( |
| <h1>{this.state.width}</h1> |
| ); |
| } |
| } |

useEffect מאפשר הן ניהול של componentDidMount componentDidUpdate, ו componentWillUnmount ללא שימוש בקומפוננטה מסוג קלאס והן שימוש חוזר ושיתוף הלוגיקה.

לסיכום, לפני תחילת העבודה עם useEffect חשוב לשים לב:

* מתי וכיצד ריאקט מבצע את useEfeect.
* כיצד להחליט מתי useEfeect ירוץ.
* כיצד לנקות דליפות זכרון כאשר עושים unmounting לקומפוננטה.

useState

setState callback in React hook

[](https://www.react-israel.co.il/wp-content/uploads/2019/09/girl-2181709_640.jpg)

בכל גודל של אפליקציה ככל הנראה אנחנו ננהל state כלשהו.כאשר אנחנו מנהלים state בקומפוננטה מסוג קלאס ונרצה לעשות שינוי ב state שלנו אנחנו נעשה שימוש ב-setState על מנת לשנות את הערכים הדיפולטיביים שהגדרנו.

|  |
| --- |
| { |
| counter:0 |
| } |
|  |
| this.setState({ |
| counter:2 |
| }) |

מכיוון ש-JS היא סינכרונית ו- setState היא פעולה אסינכרונית נוצר איזשהו פער שבו אנחנו לא יכולים להסתמך על זה שה-state שלנו באמת השתנה.אז איך בכל זאת נוכל לדעת? אז ככה, אנחנו יכולים להעביר ל-setState קריאת callback שבה נוכל לקבל את ה- state המעודכן.

|  |
| --- |
| this.setState({counter:10}, () => console.log(this.state.counter)) *//10* |

עד כאן הכל טוב ויפה  אבל מה קורה בהוקס? האם יש לנו built-in callback ל- useState? אז לא, אין לנו.  
אז מה נעשה אם נרצה להסתמך על שינוי שכרגע עשינו ב-state?

|  |
| --- |
| import React, {useState,useEffect} from "react"; |
| import ReactDOM from "react-dom"; |
|  |
|  |
| function App() { |
| const [count, setCount] = useState(0); |
|  |
| useEffect(() => { |
| *//here we are sure count has been change* |
| console.log('count',count) |
| }, [count]); |
| return ( |
| <div className="App"> |
| <h1>counter: {count}</h1> |
| <button onClick={()=>setCount(counter=>counter+1)}>Inc counter</button> |
| </div> |
| ); |
| } |
|  |
| const rootElement = document.getElementById("root"); |
| ReactDOM.render(<App />, rootElement); |

מי שיש לו קצת ניסיון עם הוקס ישים לב שאנחנו מריצים את זה גם בפעם הראשונה שהקומפוננטה מתרנדרת וזה לא המצב הרצוי. אנחנו רוצים לדעת רק ש-counter  משתנה. על מנת למנוע את המצב הזה נוסיף נשתמש ב -useRef  
 ונשמור בתוכו משתמש בוליאני שדרכו נדע האם זהו הפעם הראשונה ש-useeffect רץ.

|  |
| --- |
| import React, { useState, useEffect, useRef } from "react"; |
| import ReactDOM from "react-dom"; |
|  |
| function App() { |
| const [count, setCount] = useState(0); |
| const isFirstRun = useRef(true); |
|  |
| useEffect(() => { |
| if (isFirstRun.current) { |
| *//skip firstRun* |
| isFirstRun.current = false; |
| return; |
| } |
| *//here we are sure count has been change* |
| console.log("count", count); |
| }, [count]); |
|  |
| return ( |
| <div className="App"> |
| <h1>counter: {count}</h1> |
| <button onClick={() => setCount(count => count + 1)}> |
| click |
| </button> |
| </div> |
| ); |
| } |
|  |
| const rootElement = document.getElementById("root"); |
| ReactDOM.render(<App />, rootElement); |

כעת בכל לחיצה על הכפתור(שבעצם יוצר את השינוי בstate של ה-counter)אנחנו יכולים להריץ קוד מסויים שנרצה ולדעת בבטחון שהשינוי ב-state נעשה כבר. השיטה הזאת פחות נחה לדעתי מה-callback המובנה שיש לנו בקלאסים אבל זה מה יש ועם זה ננצח

Context API

הצוות של ריאקט שחרר לנו את Context API בגרסה 16.3.0, החל מגרסא זו והלאה בטוח להשתמש ב- Context API אשר היה בשלבים ניסיוניים.

Context API מאפשר לנו דרך קלה לשיתוף נתונים גלובליים בכל עץ האתר. יכולת זו  פותרת בעיה נפוצה המכונה-  **prop drilling,**זהו מצב שבו props צריכים להיות מועברים אל מספר קומפוננטות בעץ כדי להגיע לקומפוננטה שבאמת צריכה את זה. דבר הגורם להעברת ״מידע״(Props) מיותר לקומפוננטות שלא צריכות את המידע.  
בנוסף נוכל לשתף נתונים גם בין קומפוננטות שלא מקושרות אך עדיין צריכות לקבל מידע זהה למשל: שפת האתר.

במקרים רבים, זה יאפשר לנו להימנע משימוש בספריות צד שלישי כגון Redux ו-mobx, אשר היו מנהלות לנו את ה- Store.

**מתי להשתמש ב- Context API?**

כפי שציינו קודם, Context API שימושי לשיתוף נתונים שיכולים להיחשב גלובליים, כגון נתוני משתמש לאחר אוטנתיקציה,נתוני עגלת הקניות, הגדרות ערכת העיצוב, שפת האתר ועוד. במצבים אלו, אנו יכולים להשתמש ב- Context API , ואנחנו לא צריכים להשתמש בספריות נוספות.  
אחרי שהבנו מהם השימושים של Context API ומה הוא בא לפתור בואו נראה איך נעשה השימוש בו.

**תחילת עבודה עם Context API**

כדי להתחיל לעבוד עם Context APi ניצור Context חדש באמצעות React.createContext​.

|  |
| --- |
| import React from 'react'; |
| const Context = React.createContext(); |
| export default Context; |

לאחר שניצור את ה context נוכל לייצא ממנו את ה- Provider (הקומפוננטה שמספקת את הערך​). ולהעביר לה את הstate  שיצרנו כפרופס לערך value.

|  |
| --- |
| import React, { Component } from 'react'; |
| import Context from '../config/Context'; |
|  |
| class ContextProvider extends Component { |
| state = { |
| theme: 'light' |
| } |
| render() { |
| return ( |
| <Context.Provider value={{ |
| state: this.state, |
| }}> |
| {this.props.children} |
| </Context.Provider> |
| ) |
| } |
| } |
| export default ContextProvider |

ה- Provider תמיד צריך לעטוף מסביבו את ה-parent, לא משנה איך נבחר לצרוך את הערכים מתוך ה- context. בדוגמא שלנו נעטוף את כל האפליקציה ב- provider ככה תהיה לנו גישה מכל הקומפוננטות בפרוייקט.

|  |
| --- |
| import React from 'react'; |
| import ContextProvider from './provider/ContextProvider' |
| import Home from './Home' |
|  |
| const App = () => { |
| return ( |
| <ContextProvider> |
| <Home /> |
| </ContextProvider> |
| ); |
| } |
|  |
|  |
| export default App; |

מתוך context נוכל לייצר גם את ה- consumer (הקומפוננטה​ שצורכת את הערך​).  
הדרך שבה נייצר את ה- provider זהה לקומפוננטות מסוג קלאסים ופונקציות אך צריכת המידע היא קצת שונה ביניהן.

|  |
| --- |
| import React, { useContext } from 'react' |
| import Context from '../config/Context'; |
|  |
| const Home = () => { |
| const getContext = useContext(Context) |
| const themeColor = getContext.state.theme; *//lignt* |
|  |
| return ( |
| <div> our theme is {themeColor} </div> |
| ); |
| } |
| export default Home |

כדי לצרוך את המידע בקומפוננטות מסוג קלאס אנו נעטוף את הקומפוננטה ב Consumer, לאחר שנעשה את זה נוכל לקבל גישה לכל המידע דרך props. יכול להשמע קצת מסובך אבל זה באמת פשוט, בואו נראה דוגמא

|  |
| --- |
| import React, { useState, useEffect } from 'react'; |
| import Context from '../config/Context'; |
|  |
|  |
| export default class Home extends React.Component { |
| render() { |
| return ( |
| <Context.Consumer> |
| {Context => { |
| <div> our theme is {Context.state.theme} </div> |
| }} |
| </Context.Consumer> |
|  |
| ) |
| } |
| } |
|  |
| export default App; |

חדי העין ישימו לב שכל המידע זמין לנו רק בתוך ה- render. אז השאלה המתבקשת היא מה עם lifecycle methods?  
אנו יכולים לעשות זאת בקומפוננטה מסוג קלאס עם contextType, שהוא משתנה סטטי בקלאס

|  |
| --- |
| import React, { useState, useEffect } from 'react'; |
| import Context from '../config/Context'; |
|  |
|  |
| export default class Home extends React.Component { |
| static contextType = Context |
|  |
| componentDidMount() { |
| const store = this.context |
| const themeColor = store.state.theme |
| console.log(themeColor) *// light* |
| } |
|  |
| render() { |
| <div>check the console</div> |
| } |
|  |
| } |

**שינוי מידע ב- Context**

עד כה ראינו איך לייצר context עם ערך נתון ואיך אנחנו יכולים לגשת לערך זה מקומפוננטות שונות. אך ישנם מקרים בהם נרצה לשנות את הערך שהגדרנו או אפילו שהערך ההתחלתי הוא ריק ונרצה לחכות לדטא שיגיע מפנייה לשרת ולעשות השמה לאותו ערך. אז איך נעשה את זה? בדוגמא הבאה ניצור כפתור שישנה את הערך של המשתנה theme ל- dark.

תחילה נרשום פונקציה שמקבלת משתנה ומשנה את הערך של theme(כמובן שאם יש רק שתי ערכים יכלנו לעשות פונקציה של toggle אך נצא מנקודת הנחה שיש יותר משתי מצבים). ולאחר מכן נעביר אותה בvalue שמועבר ל-Context.Provider

|  |
| --- |
| import React, { Component } from 'react'; |
| import Context from '../config/Context'; |
|  |
| class ContextProvider extends Component { |
| state = { |
| theme: 'light' |
| } |
|  |
| changeTheme = (value) => { |
| this.setState({ |
| theme:value |
| }) |
| } |
|  |
| render() { |
| return ( |
| <Context.Provider value={{ |
| state: this.state, |
| changeTheme:this.changeTheme, |
| }}> |
| {this.props.children} |
| </Context.Provider> |
| ) |
| } |
| } |
| export default ContextProvider |

לאחר שהגדרנו את הפונקציה ומה היא תעשה כעת נקרא לה מתוך הקומפוננטה, ניצור כפתור שיקרא לפונקציה changeTheme מתוך context.

|  |
| --- |
| import React, { useContext } from 'react' |
| import Context from '../config/Context'; |
|  |
| const Home = () => { |
| const getContext = useContext(Context) |
| const themeColor = getContext.state.theme; *//lignt* |
|  |
| return ( |
| <> |
| <Button onClick={()=>getContext.changeTheme('dark')}> |
| change theme mode |
| </Button> |
| <span>our theme is {themeColor} </span> |
| </> |
| ); |
| } |
| export default Home |

**לסיכום**

* תחילה ניצור Context חדש  באמצעת ()const xContext = React.createContext
* לאחר מכן נייבא מתוך context את ה- Provider ואת ה- Consumer
* נעטוף את האפליקציה שלנו ב Provider
* נצרוך את המידע שאנו צריכים באמצעות consumer

use Reducer

useReducer מחזיר מערך של 2 אלמנטים, בדומה ל-useState. הראשון הוא ה-State הנוכחי, והשני הוא dispatch(פונקציה).

למרות ש- useState הוא הוק בסיסי ו useReducer הוא לא, useState מיושם למעשה באמצעות useReducer. משמעות הדבר היא ש useReducer הוא פרימיטיב ואנחנו יכולים להשתמש ב-useReducer עבור כל מה שהיינו משתמשים עם useState.

אם אתם מכירים את Array.prototype.reduce אז למעשה useReducer  
מקבל את אותם ארגיומנטים ובעיקרון עובד אותו הדבר. אנחנו נעביר ל- useReducer פונקציית reducer(נראה דוגמא בהמשך) וערך התחלתי(initial state). פונקציית ה Reducer שלנו מקבלת את ה-State הנוכחי ופעולה(action) ותחזיר לנו את  ה- state החדש.

**מתי נשתמש ב useReducer**

יש לנו שתי הוקס שבעזרתם אנחנו יכולים לנהל את ה state שלנו, וכמו תמיד שיש לנו שתי אפשרויות לעשות את אותה הפעולה תמיד נשאל את עצמינו מתי יהיה נכון להשתמש בכל אחת מהאופציות?

כאשר יש לנו לוגיקה מורכבת הכוללת State עם מספר תתי-ערכים (useReducer  תעזור לנו גם לשפר את הביצועים) או כאשר ה-State הבא תלוי ב-State הקודם נעדיף בדרך כלל להשתמש ב- useReducer ולא ב-useState.

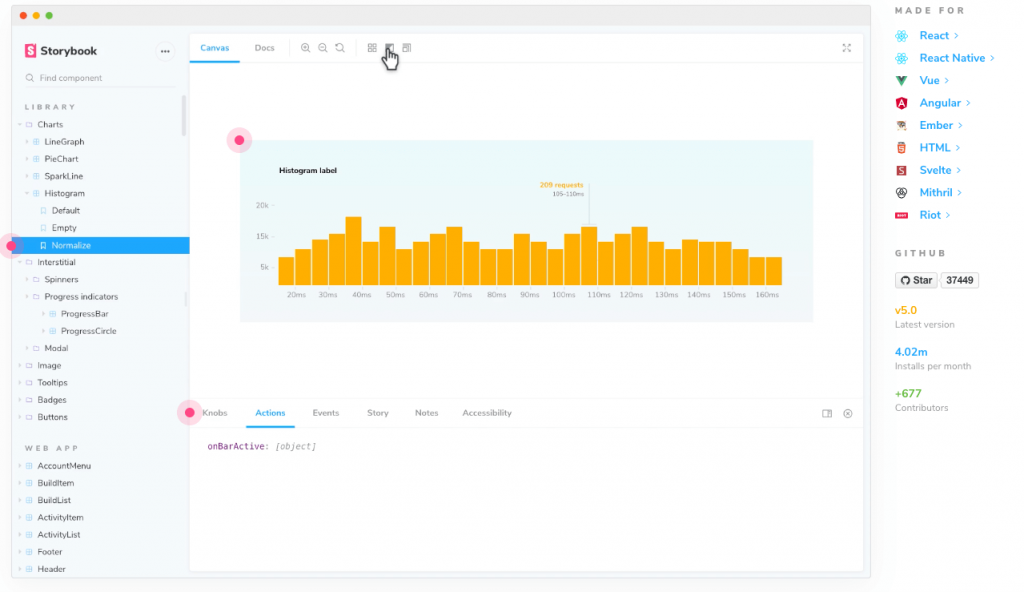
בדוגמא הבאה נראה איך אנחנו יכולים להשתמש ב- useReducerב על מנת ליצור Counter. יהיה לנו כפתור שמעלה  ב-1 כל פעם. כפתור שמחסיר 1 וכפתור אתחול שיחזיר את ה counter לערך ההתחלתי( 0- שאותו אנחנו קבענו).

גם את הקפיצות של הקאונטר אנחנו יכולים לקבוע,בדוגמא שלנו הקאונטר יקפוץ באחד אך תוכלו להעתיק את הקוד לפרוייקט ולשחק עם זה ולשנות את ה- by בתוך ה- dispatch. מי שמכיר ויצא לו לעבוד עם Redux אז הקונספט נורא זהה ויהיה לו קל להתחבר להוק הזה.  
הדבר השונה מ Redux שה – initalState לא חייב להיות אובייקט והוא יכול לקבל כל ערך.

|  |
| --- |
| import React, { useReducer } from "react"; |
| const initialState = 0; |
| const counterReducer = (state, action) => { |
| switch (action.type) { |
| case "INCREASE": |
| return state + action.by; |
| case "DECREASE": |
| return state - action.by; |
| case "RESET": |
| return initialState; |
| default: |
| throw new Error(); |
| } |
| }; |
|  |
| const Counter = () => { |
| const [count, dispatch] = useReducer(counterReducer, initialState); |
|  |
| const handleIncrease = () => { |
| dispatch({ type: "INCREASE", by: 1 }); |
| }; |
|  |
| const handleDecrease = () => { |
| dispatch({ type: "DECREASE", by: 1 }); |
| }; |
|  |
| const reset = () => { |
| dispatch({ type: "RESET" }); |
| }; |
|  |
| return ( |
| <div> |
| <h1>Counter with useReducer Hook</h1> |
| <p>Count: {count}</p> |
|  |
| <div> |
| <button type="button" onClick={handleIncrease}> |
| + |
| </button> |
| <button type="button" onClick={handleDecrease}> |
| - |
| </button> |
| <button type="button" onClick={reset}> |
| Reset |
| </button> |
| </div> |
| </div> |
| ); |
| }; |
|  |
| export default Counter; |

StoryBook

**Storybook** היא בעצם ספריית UI שניתן להשתמש בה לתיעוד הקומפוננטות שלנו. המטרה היא ליצור סביבה מבודדת שבה ניתן לראות את כל הקומפוננטות שלנו ולשחק איתם החל מהעברת props שונים, שינויי צבעים, גודל והתנהגות והכל בזמן אמת. יתרון נוסף לעבודה עם **Storybook**הוא שכולם עובדים עם מקור אמת יחיד החל ממעצבים למפתחים ולמנהלי הפרוייקט כולם מיושרים לגבי הנראות ההתנהגות ומה הם מצפים לקבל.  
כמובן שבתור מפתחים הרבה יותר נח לנו לעבוד עם קומפננטות שכבר עברו טסטים ורצות תקופה מסויימת בפרודקשיין ואין לנו צורך לכתוב קומפוננטות מחדש.  
בחלק זה ניצור קומפוננטות באמצעות Storybook (גרסא מספר 6) וריאקט (גרסא 16.13) ובסוף נעלה את storybook לשרת כאפליקציה עצמאית שתשמש כ-style guide.



כדאי שיהיה לנו יותר קל לעקוב ולהבין אלה השלבים שנעשה

* ניצור פרוייקט באמצעות ריאקט
* נוסיף את storybook/react
* ניצור ספריית קומפוננטות
* נקסטם את ה-theme של Storybook
* נעלה את הפרוייקט שיצרנו לאחסון

**יצירת פרוייקט ריאקט עם StoryBook**

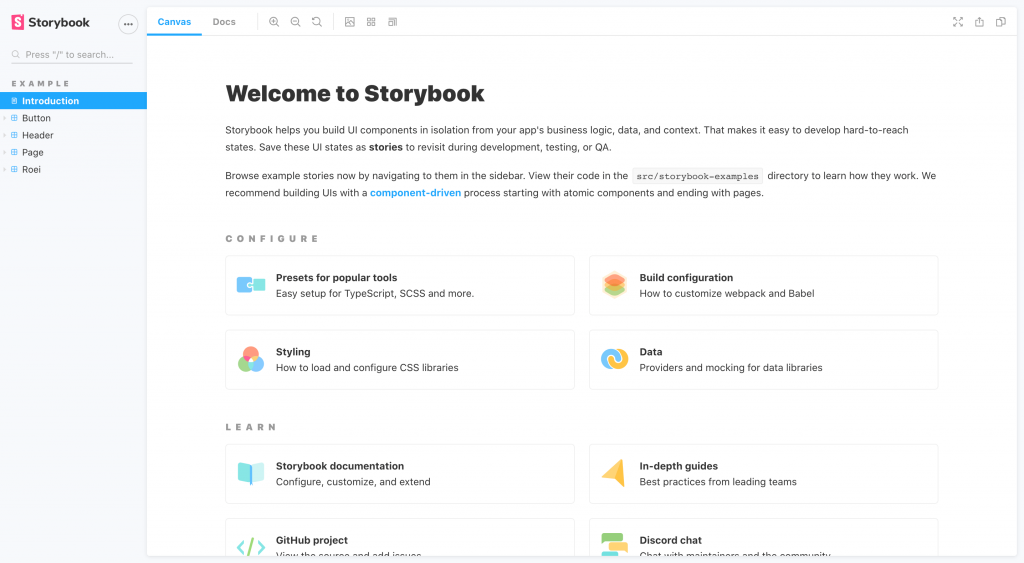
נתחיל בצעד הראשון והפשוט ביותר,יצירת פרויקט ריאקט חדש.  
לשם כך נשתמש ב- create-react-app.

|  |
| --- |
| $ npx create-react-app storybook-app |
| $ cd storybook-app |

 עכשיו שיש לנו פרוייקט ריאקט חדש שעובד השלב הבא הוא התקנת  StoryBook בפרוייקט שלנו.

|  |
| --- |
| $ npx -p @storybook/cli sb init |

הפקודה שעכשיו הרצנו יודעת לזהות אוטומטית את הפריימוורק שבו אנחנו משתמשים מקובץ ה- **package.json** ולהתקין את הגרסא התואמת של storybook. לאחר שסיימנו להריץ את הפקודות נוכל לראות את כל התלויות שהותקנו בקובץ **package.json** .  
כדי לוודא שהכל עובד כמו שצריך נריץ גם את הפרוייקט ריאקט שלנו וגם את storybook.



**סיפורים ב-stoybook**

אחרי שסיימנו עם כל ההתקנות הגיע הזמן שניצור את הקומפוננטות שלנו ואת הסיפורים שלהם. סיפורים? הכוונה ל stories   
בחרתי להראות דוגמא של כפתור מכיוון שיחסית זאת קומפוננטה פשוטה אך עם זאת יש בה הרבה ״בשר״, נוכל להעביר פרופס שונים, עיצוב שונה וגם נראה מה קורה באינטרקציה של המשתמש עם הכפתור(לדוגמא קליק).

לכל קומפוננטה יהיו לנו 3 קבצים

* קובץ של הקומפוננטה עצמה-js
* קובץ סטייל- css,scss או מה שנבחר
* קובץ stories



לאחר שיצרנו את הקבצים הגיע הזמן להתחיל לרשום קצת קוד  
בואו נראה את הקובץ Button.js ונעבור עליו

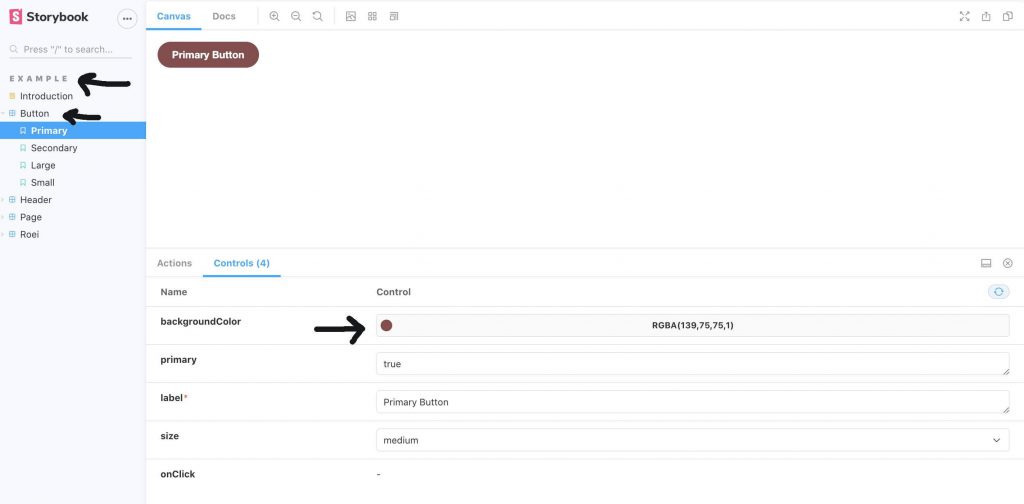
|  |
| --- |
| import React from 'react'; |
| import PropTypes from 'prop-types'; |
| import './button.css'; |
|  |
| */\*\** |
| *\* Primary UI component for user interaction* |
| *\*/* |
| export const Button = ({ onClick ,primary, backgroundColor, size, label,...props }) => { |
| const mode = primary ? 'storybook-button--primary' : 'storybook-button--secondary'; |
| return ( |
| <button |
| type="button" |
| className={['storybook-button', `storybook-button--${size}`, mode].join(' ')} |
| style={{ backgroundColor }} |
| onClick={onClick} |
| {...props} |
| > |
| {label } |
| </button> |
| ); |
| }; |
|  |
| Button.propTypes = { |
| *//Is this the principal call to action on the page?* |
|  |
|  |
| primary: PropTypes.bool, |
| *// What background color to use* |
| backgroundColor: PropTypes.string, |
| *// How large should the button be?* |
| size: PropTypes.oneOf(['small', 'medium', 'large']), |
| *// Button contents* |
| label: PropTypes.string.isRequired, |
| *// Optional click handler* |
| onClick: PropTypes.func, |
| }; |
|  |

בקוד למעלה בעצם יש לנו כפתור די פשוט שיודע לקבל כמה משתנים  
**primary**, **backgroundColor**, **size**, **label, onClick** ולפיהם יש לו התנהגות או עיצוב שונה. מקווה שהקוד מובן וקריא לא רואה צורך להרחיב עליו כי אין שם משהו מסובך.  
ב-storybook שלנו נרצה את היכולת לשחק עם הפרופס השונים ולראות איך הכפתור יתנהג ולקחת את הקוד שמתאים לנו או לדעת מה הסט יכולות של אותו כפתור.  
הגיע הזמן לחלק המעניין, כתיבת ה- stories    
לפני שנצלול לתוך הנושא הזה חשוב לציין שה- stories יקחו את הproptypes שהגדרנו פה והם יהיו הדיפולט בתוך הסטורי שלנו.

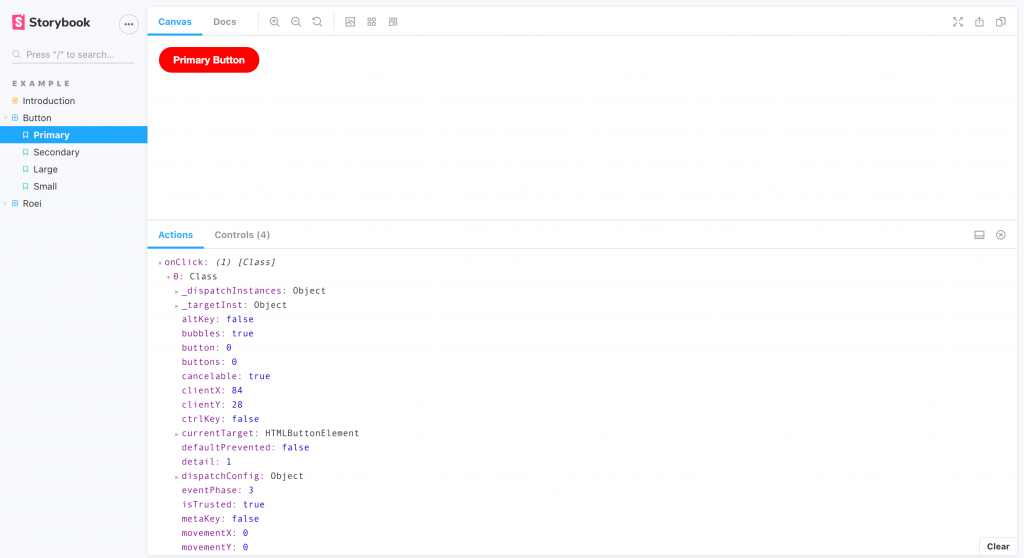
|  |
| --- |
| import React from 'react'; |
| import { Button } from './Button'; |
|  |
| export default { |
| title: 'Example/Button', |
| component: Button, |
| argTypes: { |
| backgroundColor: { control: 'color' }, |
| }, |
| }; |
| const Template = (args) => <Button {...args} />; |
| שexport const Primary = Template.bind({}); |
| Primary.args = { |
| primary: true, |
| label: 'Primary Button', |
| backgroundColor: "red", |
| size: "medium" |
| }; |
|  |
| export const Secondary = Template.bind({}); |
| Secondary.args = { |
| label: 'Secondary Button', |
| size: "large", |
| backgroundColor: "#ddd" |
| }; |
|  |
| export const Large = Template.bind({}); |
| Large.args = { |
| size: 'large', |
| label: 'Button', |
|  |
| }; |
|  |
| export const Small = Template.bind({}); |
| Small.args = { |
| size: 'small', |
| label: 'Button', |
| }; |

ראשית שימו לב שהקובץ צריך להיות בפורמט הבא **myComponent.stories.js**  
כדי שיהיה לנו כל מה שצריך כדי להתחיל נעשה import לריאקט ולקומפוננטה שלנו. לאחר מכן נבנה את האובייקט הראשי שאותו נייצא ובתוכו יהיה

* **title**– זה יהיה השם הראשי של הקומפוננטה (שימו לב שה-example שלפני ה- /  מייצג סוג של group שתחתיו ישב הכפתור)
* **component-**הקומפוננטה אותה אנחנו רוצים להציג
* **argTypes** – אופציונאלי, דרכו ניתן לשייך לפרופ מסויים מצב תצוגה.
* ניתן גם להוסיף args, פרמטרים ודקורטורים- מומלץ להסתכל על כל האפשרויות בדוקיומנטציה של [StoryBook](https://storybook.js.org/docs/react)

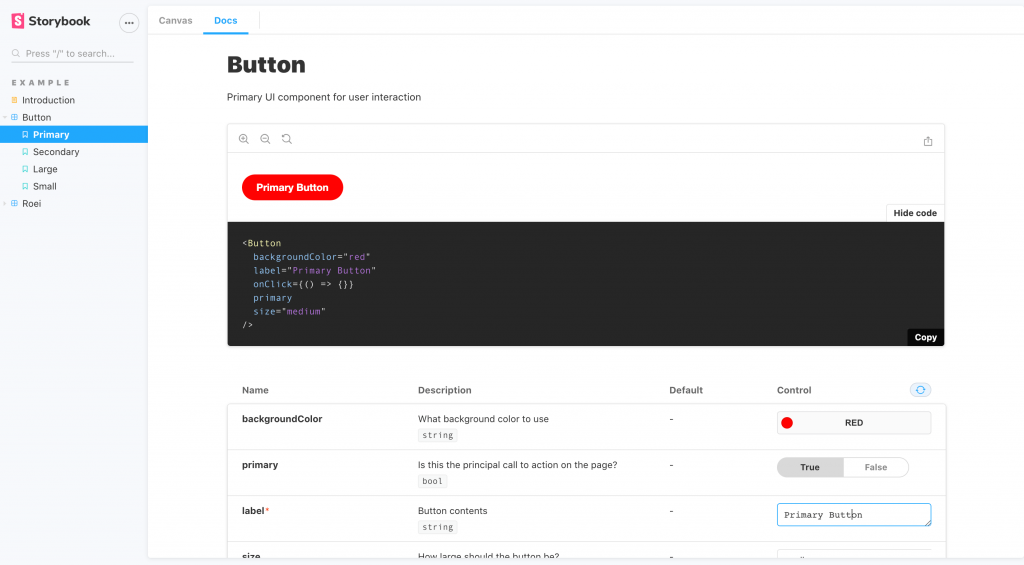


בתחתית המסך יש לנו את כל הפרופס שהעברנו לקומפוננטה ואת היכולת לשנות אותם ולראות בזמן אמת מה קורה לכפתור.  
מעבר לאופציה של שינויי הפרופס בתחתית המסך ישנם מצבים שכפתור מסויים נמצא בשימוש רחב במספר מצבים למשל כפתור במצב onSuccess primary כפתור גדול, כפתור קטן וכו.  
בשביל לראות את המצבים האלה מוגדרים מראש יש לנו את האופציה לתת תתי מצבים בהם נגדיר default Values וניצור**מצב ספציפי** של הקומפוננטה וניתן לו שם.  
תוכלו לראות את זה בתמונה הנל. יצרנו טמפלייט של הכפתור שלנו ואז יצרנו סוג של שכפול של הכפתור והגדרנו מה הפרופס שאנחנו רוצים כדי להגיע למצב מסויים ונתנו לו שם.  
אז ראינו איך אנחנו יכולים לשחק עם מצבי סטייט שונים ועם סטייל שונה ואיך אנחנו יכולים לשנות אותם. אבל איך נדמה לחיצה על הכפתור? אולי לא שמתם לב אבל כבר עשינו את זה. כל מה שהיינו צריכים לעשות זה להעביר פרופ לפונקציה ולשייך אותו לאיבנט onClick. כעת נוכל לראות את זה קורה תחת הטאב Actions



**הטמעת הקומפוננטה**

אחרי ששיחקנו קצת עם הקומפוננטה והגענו לעיצוב שרצינו או למדנו מספיק מה היכולות שלה הגיע הזמן להטמיע אותה בקוד שלנו. כל מה שנצטרך לעשות הוא לעבור לטאב של Docs ללחוץ על show code ואז יפתח לנו חלונית עם חתיכת קוד שנצטרך לשים אצלינו וזהו!  
מתחת לזה אנחנו יכולים לראות שוב את כל הפרופס שאפשר להעביר לקומפוננטה ולשחק עם זה בזמן אמת.



**שינוי ערכת נושא**

אחד הדברים שאני אוהב הוא dark mode יש כאלה שפחות מתחברים ואוהבים את הדיפולט אבל כדאי לדעת שגם ב- StoryBook יש לנו את האופציה לשנות את ערכת הנושא. בואו נראה איך עושים את זה.  
ראשית נוודא שהחבילות האלה מותקנות (הן אמורות להיות ) ואם לא נוסיף אותן

yarn add --dev @storybook/addons @storybook/theming

לאחר מכן בתוך התיקייה של Storybook נייצר את הקובץ  manager.js ובתוכו נשים את הקוד הבא.  
שימו לב שאנחנו מחליפים ערכת עיצוב אנחנו דורסים את הקודמת ולא נעשה איחוד של שתי הערכות.

|  |
| --- |
| *// .storybook/manager.js* |
|  |
| import { addons } from '@storybook/addons'; |
| import { themes } from '@storybook/theming'; |
|  |
| addons.setConfig({ |
| theme: themes.dark, |
| }); |

בדוגמא מעל השתמשנו בערכה מוכנה שמגיעה מ- storybook אך ניתן גם להכין קובץ סטיילינג אישי ולעשות לו import.

**העלאה לפרודקשיין**

ראשית נצטרך לבנות את Storybook כאפליקציה סטטית באמצעות ה- cli שהתקנו בהתחלה באמצעות הפקודה הבאה

yarn build-storybook -o ./path/to/build

פקודה זו תיצור לנו אפליקציה סטטית בנתיב שבחרנו. את התוכן שיש בתיקייה נוכל להעלות לכל שרת שנבחר  
לפני זה נוכל לנסות להריץ את התיקייה לוקאלית לראות שהכל עובד באמצעות הפקודה הבאה

npx http-server ./path/to/build

**לסיכום**

בחלק זה למדנו כיצד לבנות קומפוננטות אינטראקטיביות באמצעות Storybook ו- React. ראינו מה אפשרי לעשות ב- Storybook וכיצד לשלב את storybook כספריית קומפוננטות בפרוייקט שלנו.  
פיתוח באמצעות Storybook משמש מקור אמת יחיד עבור מפתחים ומעצבים העובדים בצוות וניתן לשלב אותו עם כל הפרימוורקים המובילים. בחלק זה עסקנו כמובן בריאקט.

Error Boundaries

Error Boundaries הוצג לנו בריאקט 16 כדרך לתפוס שגיאות המתרחשות בשלב הרנדור ולהציג למשתמש fallback שאותו נגדיר. בחלק זה נלמד מהם Error Boundaries  מה הם באו לפתור,איך ואיפה צריך להשתמשמש בהם.

לפני שנצלול לתוך Error Boundaries בואו נבין את הקונספט. Error Boundaries היא קומפוננטת ריאקט שתופסת שגיאות JavaScript בכל עץ הקומפוננטות שתחתיו, כאשר נתפסת שגיאה קומפוננטה זו יודעת להציג fallback במקום שבו הקומפוננטה קרסה או לרשום את השגיאה עצמה. Error Boundaries תופסים שגיאות במהלך הרנדר, ב-lifecycle methods וב-constructors של מי שמתחתם בעץ הקומפוננטות.

בדוגמא הבאה נדמה שגיאה כדי לראות מה קורה בדרך כלל כאשר אנחנו לא מתשמשים ב-Error Boundaries

נלחץ על הכפתור "+" ובלחיצה החמישית זה ייכשל ויזרוק לנו הערה

כאשר האפליקציה נתקלת בשגיאה כזו במצב production הקומפוננטה תעבור למצב unmount והיוזר ישאר עם ״מקום ריק״ באפליקציה . זוהי חוויות ux מאוד רעה ולא נרצה שהמשתמשים שלנו לא יבינו מה קורה.

Error Boundaries בא לתת לנו מענה בדיוק בשביל זה!

Bottom of Form

**ועכשיו נוסיף Error Boundaries**

אז Error Boundaries היא בעצם קומפוננטה מסוג class  אשר משתמשת בשתי lifecycle methods.  
static getDerivedStateFromError ו- componentDidCatch שאותן מייד נסביר

static getDerivedStateFromError משמש אותנו על מנת לרנדר fallback לאחר שנזרקה שגיאה  
ו- componentDidCatch משמש אותנו בכדי לקבל ולרשום מידע על השגיאה שקיבלנו.אפשר גם לשלוח את השגיאה ללוגרים (loggers) למשל Sentry.  
בואו נראה איך הקומפוננטה נראית

|  |
| --- |
| class ErrorBoundary extends React.Component { |
| constructor(props) { |
| super(props); |
| this.state = { hasError: false }; |
| } |
|  |
| static getDerivedStateFromError(error) { |
| *// Update state so the next render will show the fallback UI.* |
| return { hasError: true }; |
| } |
|  |
| componentDidCatch(error, errorInfo) { |
| *// You can also log the error to an error reporting service* |
| logErrorToMyService(error, errorInfo); |
| } |
|  |
| render() { |
| if (this.state.hasError) { |
| *// You can render any custom fallback UI* |
| return <h1>Something went wrong.</h1>; |
| } |
|  |
| return this.props.children; |
| } |
| } |

ואז אנחנו יכולים להשתמש בה כקומפוננטה רגילה

|  |
| --- |
| <ErrorBoundary> |
| <MyWidget /> |
| </ErrorBoundary> |

בואו נראה את זה על הדוגמא הקודמת  שימו לב שגם פה תוצג לנו הודעת שגיאה כמו קודם וזה צפוי מכיוון שהצוות של ריקאט תכנן את Error Boundaries שנקבל את השגיאות המוכרות לנו גם בסביבת פיתוח. שימו לב שאם נלחץ על האיקס בפינה הימנית העליונה נוכל לראות את ה-fallback.  
זה נותן לנו משוב חזותי למשתמש שמשהו נשבר ומאפשר לו להמשיך באינטראקציה עם האפליקציה שלנו.

**Error Boundaries או Try…Catch**

Error Boundaries למעשה אינם מתחרים עם  try catch. והם מיועדים לתפוס שגיאות שמקורם בשלושה מקומות

* During *render* phase
* In a lifecycle method
* In the *constructor*

אך שימו לב Error Boundaries לא מטפלת בשגיאות שמקורן ב-

* Event handlers.
* setTimeout or requestAnimationFramecallbacks.
* רנדור בצד שרת-Server side rendering.  
    
  אז Error Boundaries לא באמת משפיעים על אופן השימוש שלנו ב-try catch ושניהם נחוצים לטיפול בשגיאות בריאקט

**לסיכום**

כעת שאנחנו מכירים את Error Boundaries ויודעים איך להשתמש בהם נוכל לתפוס שגיאות בקומפוננטות שלנו ולתת חוויות משתמש הרבה יותר טובה ליוזרים של האפליקציה שלנו. יש שיגידו שצריך לעטוף לפחות את הקומפוננטה הראשית כשלב ראשון ואז לאט לאט להוסיף עוד Error Boundaries. אם אתם משתמשים ב- lazy &Suspense אז זה מקרה קלאסי לעטוף אותם ב- Error Boundaries. מוזמנים לתת דוגמאות בתגובות איפה אתם משתמשים ב- Error Boundaries.