**Hello World**

A legegyszerűbb példa. Egy első szintű címsor, Hello Word felirattal.

ReactDOM.render(

<h1>Hello, world!</h1>,

document.getElementById('root')

);

# Introducing JSX

# Ez egy változó deklaráció, ahol egy első szintű címsorban van a Hello Word felirat.

const element = <h1>Hello, world!</h1>;

Ez a szintaxis nem string, és nem HTML. Ez JSX, ami a JavaScript kiegészítő szintaxisa.

.

### Beágyazott kifejezések JSX-ben

Egy name változót deklarálunk, és a JSX-ben használjuk az értékét a {}-vel ({name}).

const name = 'Josh Perez';

const element = <h1>Hello, {name}</h1>;

ReactDOM.render(

element,

document.getElementById('root')

);

Bármilyen érvényes JavaScript kifejezést tehetünk a {} közé, pl {2+2}, {user.firstName}, {formatName(user)} tehát összeadást, változót (objektum property-je), függvényt.

A példában egy függvény (formatName(user)) értékét hívjuk meg a <h1> elemben.

function formatName(user) {

return user.firstName + ' ' + user.lastName;

}

const user = {

firstName: 'Harper',

lastName: 'Perez'

};

const element = (

<h1>

Hello, {formatName(user)}!

</h1>

);

ReactDOM.render(

element,

document.getElementById('root')

);

### JSX egy kifejezés is

A JSX kifejezést használhatjuk if-ben, for-ban is, függévnyből visszatérhetünk velük.

function getGreeting(user) {

if (user) {

return <h1>Hello, {formatName(user)}!</h1>;

}

return <h1>Hello, Stranger.</h1>;

}

### Attribútumok JSX-ben

Lehet úgy, mint HTML-ben (tabIndex="0"):

const element = <div tabIndex="0"></div>;

Használhatunk {}-t is attribútum értékeként (src={user.avatarUrl}). ekkor nyilván a user.avatarUrl kiértékelődik.

const element = <img src={user.avatarUrl}></img>;

Nem teszünk idézőjelet, ha a {} megoldást választjuk, csak ha string értéket adunk meg!!

**Warning:**

A React camelCase property elnevezést használ

Tehát [className](https://developer.mozilla.org/en-US/docs/Web/API/Element/className) és [tabIndex](https://developer.mozilla.org/en-US/docs/Web/API/HTMLElement/tabIndex).

### Gyerek elem definálása JSX-ben

Ha a tag üres, akkor a /> vel lehet lezárni.

const element = <img src={user.avatarUrl} />;

JSX-ben gyerek elem:

const element = (

<div>

<h1>Hello!</h1>

<h2>Good to see you here.</h2>

</div>

);

### JSX objektumokat reprezentál

Az alábbi két példa egyenértékű:

const element = (

<h1 className="greeting">

Hello, world!

</h1>

);

const element = React.createElement(

'h1',

{className: 'greeting'},

'Hello, world!'

);

React.createElement() az alábbi objektumot hozza létre:

// Note: this structure is simplified

const element = {

type: 'h1',

props: {

className: 'greeting',

children: 'Hello, world!'

}

};

Ezt “React elements”-nek nevezzük.

# Elemek renderelése

Az elemek a React legkisebb építőkövei.

Egy elem definálja, hogy mit szeretnénk látni a képernyőn.

const element = <h1>Hello, world</h1>;

## Elem renderelése a DOM-ba

Legyen egy <div> valahol a HTML dokumentumon belül:

<div id="root"></div>

Ezt hívjuk root-nak (gyökér) azért, mert ezen csomóponton belül mindent a react technológiával tudunk menedzselni.

Egy react elem DOM csomópontba renderelése a ReactDOM.render()-el történik:

const element = <h1>Hello, world</h1>;

ReactDOM.render(element, document.getElementById('root'));

[**Try it on CodePen**](https://reactjs.org/redirect-to-codepen/rendering-elements/render-an-element)

## A renderelt elem frissítése

A react elemek nem megváltoztathatóak (immutable). Ha egyszer elkészítettünk egy elemet, akkor a gyerek elemeit nem lehet megváltoztatni, sem az attribútumait. Az eddigi ismereteink szerint a UI frissítésének a módja új elem elkészítése és a ReactDOM.render().-nek átadása.

function tick() {

const element = (

<div>

<h1>Hello, world!</h1>

<h2>It is {new Date().toLocaleTimeString()}.</h2>

</div>

);

ReactDOM.render(element, document.getElementById('root'));

}

setInterval(tick, 1000);

[**Try it on CodePen**](https://reactjs.org/redirect-to-codepen/rendering-elements/update-rendered-element)

Minden másodpercen meghívja a ReactDOM.render() –t a [setInterval()](https://developer.mozilla.org/en-US/docs/Web/API/WindowTimers/setInterval) callback függvény segítségével.

# Componensek és Props

A komponensek a UI-t önálló, újrahasznosítható darabokra bontják. Ekkor „kis izolált darabokban kell gondolkodunk”.

Koncepcionálisa a komponensek olyanok, mint a JavaScript függvények. Valamilyen inputot („props”) várnak, és react elemekkel térnek vissza, amik leírják, amit látni kell a képernyőn.

## Függvények és osztály komponensek

A legegyeszerűbb módszer kompinens írásra az, ha JavaScript függvényt írunk:

function Welcome(props) {

return <h1>Hello, {props.name}</h1>;

}

Ez a függvény egy érvényes react komponens, mert egyetlen „props”-t vár (amely property-kből áll) argumentumként, és react elemmel tér vissza. Ezt a komponenst függvény komponensnek nevezzük, mert igazából JavaScript függvények.

De osztályt is használhatunk, hogy komponenst definiáljunk:

class Welcome extends React.Component {

render() {

return <h1>Hello, {this.props.name}</h1>;

}

}

## Komponens renderelése

Az előzőekben olyan react elemekkel találkoztunk, amelyek DOM tag-eket reprezentálnak:

const element = <div />;

De az elemek felhasználó által definiált komponenseket is reprezentálhatnak:

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

Ha az react egy felhasználó által definiált komponenssel találkozik, akkor a JSX attribútumokat egyetlen objektumként adja át. Ezt az objektumot „props”-nak nevezzük.

Az alábbi kód a „Hello Sara” szöveget írja ki:

function Welcome(props) {

return <h1>Hello, {props.name}</h1>;

}

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

ReactDOM.render(

element,

document.getElementById('root')

);

[**Try it on CodePen**](https://reactjs.org/redirect-to-codepen/components-and-props/rendering-a-component)

**A példa az alábbiakat mutatja:**

**A** ReactDOM.render() –t hívja meg a <Welcome name="Sara" /> elemmel.

* A react a Welcome komponenst meghívja a {name:’Sara’} props-al.
* A Welcome komponensünk a <h1> Hello, Sara</h1> elemmel tér vissza.

**A komponensek neveit mindig nagy kezdőbetűvel írjuk.**

React komponenseket mindig nagy kezdőbetűvel írjuk, nem úgy mint a HTML elemeket, ezzel különböztetjük meg őket, hogy react komponensek. Például a <div/> HTML elemet reprezentál, de a <Wecome /> komponenst.

## Komponensek egymásba ágyazása

A komponensek akár másik komponensre is hivatkozhatnak, mint lehetséges kimenetek.

Például készítünk egy App komponenst ami a Welcome-ot renderelni többször:

function Welcome(props) {

return <h1>Hello, {props.name}</h1>;

}

function App() {

return (

<div>

<Welcome name="Sara" />

<Welcome name="Cahal" />

<Welcome name="Edite" />

</div>

);

}

ReactDOM.render(

<App />,

document.getElementById('root')

);

[**Try it on CodePen**](https://reactjs.org/redirect-to-codepen/components-and-props/composing-components)

## Komponesek kisebb komponensekre bontása

Ne féljünk a komponenseket kisebb komponensekre bontani.

Például vegyük az alábbi Comment komponenst:

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>

);

}

[**Try it on CodePen**](https://reactjs.org/redirect-to-codepen/components-and-props/extracting-components)

Az alábbiakat várja: author (objektum), text (string), date (dátum) mint props.

Ezt a kkomponenst trükkösen lehet megváltoztatni, a beágyazás miatt, és nehezen lehet újrahasznosítani az egyes részeit. Ezért vegyük kisebb részekre. Először az Avatar-ral kezdünk:

function Avatar(props) {

return (

<img className="Avatar"

src={props.user.avatarUrl}

alt={props.user.name}

/>

);

}

Az Avatar-nak nem kell tudnia, hogy mi renderel-ődik a Comment-en belül. Ezért használunk user-t és nem author-t.

A Comment-et egy kicsit egyszerűsítjük:

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>

);

}

Ezután a UserInfo komponenst módosítjuk, ami Avatar komponenst render-el és a user nevét:

function UserInfo(props) {

return (

<div className="UserInfo">

<Avatar user={props.user} />

<div className="UserInfo-name">

{props.user.name}

</div>

</div>

);

}

Így a Comment-et tovább egyszerűsíthetjük:

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>

);

}

[**Try it on CodePen**](https://reactjs.org/redirect-to-codepen/components-and-props/extracting-components-continued)

Azon UI elemek, amelyeket sokszor használunk, vagy nagyon komplexek érdemes lehet újrahasznosítható komponensekre bontani.

## Props Read-Only

Amikor egy komponenst deklarálunk mint függvény vagy osztály, akkor nem lehet megváltoztatni a saját props-jait. Például vegyünk egy sum függvényt:

function sum(a, b) {

return a + b;

}

Az ilyen függvényeket „pure”-nak hívjuk, mert nem kísérlik meg a bemeneteiket megváltoztatni, és mindig ugyanazzal az eredménnyel térnek vissza, ugyanarra a bemenetre.

Ellenben ez a függvény nem „pure”, mert a saját bemenetét változtatja meg.

function withdraw(account, amount) {

account.total -= amount;

}

**Minden react komponensnek úgy kell viselkednie, mint a „pure” függvényeknek a saját props-ra nézve.**

# State and Lifecycle

Vegyünk az órás példát. Eddig azt néztük, hogy egyetlen módja van a UI módosításának. A ReactDOM.render()  meghívása, hogy a renderel-t kimenetet megváltoztassuk:

function tick() {

const element = (

<div>

<h1>Hello, world!</h1>

<h2>It is {new Date().toLocaleTimeString()}.</h2>

</div>

);

ReactDOM.render(

element,

document.getElementById('root')

);

}

setInterval(tick, 1000);

[**Try it on CodePen**](https://codepen.io/gaearon/pen/gwoJZk?editors=0010)

Ebben a fejezetben megtanuljuk, hogyan lehet a Clock komponenst igazán újrahasznosíthatóvá tenni. A saját időzítőjét állítja be, és módosítja minden másodpercben:

A Clock :

function Clock(props) {

return (

<div>

<h1>Hello, world!</h1>

<h2>It is {props.date.toLocaleTimeString()}.</h2>

</div>

);

}

function tick() {

ReactDOM.render(

<Clock date={new Date()} />,

document.getElementById('root')

);

}

setInterval(tick, 1000);

[**Try it on CodePen**](https://codepen.io/gaearon/pen/dpdoYR?editors=0010)

## Függvény konvertálása osztállyá

A függvény komponenst, mint pl itt a Clock osztállyá az alábbi lépésekkel tudjuk:

1. Egy osztály létrehozása, ugyanolyan névvel (itt Clock) és a React.Component-ből származtatni.
2. Egy render() metódus hozzáadása.
3. A függvény törzsét a render() metódusba tesszük.
4. A props helyett this.props-ot írunk a render() törzsében.

class Clock extends React.Component {

render() {

return (

<div>

<h1>Hello, world!</h1>

<h2>It is {this.props.date.toLocaleTimeString()}.</h2>

</div>

);

}

}

[**Try it on CodePen**](https://codepen.io/gaearon/pen/zKRGpo?editors=0010)

Clock mostmár nem függvény, hanem osztály.

A render metódus mindig meghívódik, amikor módosítás történik, de amikor a <Clock />- ot ugyanabba a DOM node-ba renderel-jük, egyetlen Clock osztály példányt használunk.

## Lokális state osztálynak

A példában a date props-ról state-re alakításához három lépésre van szükségünk:

We will move the date from props to state in three steps:

1. A this.props.date-et this.state.date-re írjuk át a render() metódusban.

class Clock extends React.Component {

render() {

return (

<div>

<h1>Hello, world!</h1>

<h2>It is {this.state.date.toLocaleTimeString()}.</h2>

</div>

);

}

}

1. Egy konstruktort adunk, ami a kezdeti this.state-et fogja jelölni.

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>

);

}

}

Így tudjuk a props-ot az ős konstruktorának átadni:

constructor(props) {

super(props);

this.state = {date: new Date()};

}

Mindig az ős konstruktort használjuk a props-al.

1. Kitöröljük a date-et a <Clock /> elemből.

ReactDOM.render(

<Clock />,

document.getElementById('root')

);

Az eredmény:

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>

);

}

}

ReactDOM.render(

<Clock />,

document.getElementById('root')

);

[**Try it on CodePen**](https://codepen.io/gaearon/pen/KgQpJd?editors=0010)

A követkőkben a Clock-nak saját timer-t készítünk, ami minden másodperben frissítődik.

## Adding Lifecycle Methods to a Class

Az egyes alkalmazásokban nagyon fontos, hogy az erőforrást felszabadítsunk, h a komponenst megszüntetjük.

Például szeretnénk beállítani egy timer-t a Clock-nak mindig, amikor a DOM-hoz render-elődik. Ezt hívjuk „mounting”-nak a react-ban.

De szeretnénk törölni is ezt a timer-t amikor a DOM-ban a Clock eltávolításra kerül. Ezt hívjuk „unmounting”-nak.

Készíthetünk speciális metódusokat az osztályokban, amelyek a komponensek mount-olását és unmount-olását végzik.

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>

);

}

}

Ezek a metódusok életciklus (lifecycle) metódusok

A componentDidMount() metódus aztuán, hogy a komponens a DOM-hoz render-elődik fut le. Itt állítjuk be a timer-t.

componentDidMount() {

this.timerID = setInterval(

() => this.tick(),

1000

);

}

Note how we save the timer ID right on this (this.timerID).

A timer-t a componentWillUnmount() metódusban szabadítjuk fel.

componentWillUnmount() {

clearInterval(this.timerID);

}

Végül egy thick() nevű metódust is implementálunk, amelyben a Clock komponens minden másodpercben lefut.

A this.setState() metódust arra használjuk hogy a komponens állapotát változtassa.

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>

);

}

}

ReactDOM.render(

<Clock />,

document.getElementById('root')

);

[**Try it on CodePen**](https://codepen.io/gaearon/pen/amqdNA?editors=0010)

Így az óra minden másodpercben frissül.

Összefoglalva, hogy mi történik:

1. Amikor a <Clock /> a ReactDOM.render()-nek átadódik, a react a Clock komponens konstruktorát hívja meg. Mivel a Clock-nak az akutális időt kell mutatni, a this.state-et olyan objektummal inicializiálja, ami az aktuális időt mutatja. A későbbiekben ezt az állapotot fogjuk frissíteni.
2. A react ezután a Clock komponens render() metódust hívja meg. Ezután a DOM frissítődik.
3. Amikor a Clock kimenete a DOM-ba illesztődik, a react a componentDidMount() metódust hívja meg. A metódusban a Clock komponens megkéri a böngészőt, hogy állítsa be a timer-t hogy meghívja a komponens thick() metódusát minden másodpercben.
4. Minden másodpercben a böngésző meghívja a thick() metódust. Ebben a Clock komponens menedzseli a UI frissítést úgy hogy a setState()-et meghívja olyan objektummal, amely az aktuális időt tartalmazza. A setState() hívásnak köszönhetően a react tudja, hogy a sate megváltozott, és újra meghívja a render() metódust újra, hogy megtudja, hogy mi kerül kirajzolásra.
5. Ha a Clock komponens valamikor kitörlődik a DOM-ból, a react a componentWillUnmount() metódust hívja meg, tehát a timer (időzítő) megáll.

## A state helyes használata

Három dolog, amit a setState()-ről érdemes tudni:

### A state-et ne módosítsuk közvetlenül

Például ez a kód nem render-eli újra a komponenst:

// Wrong

this.state.comment = 'Hello';

Ehelyett használjuk a setState() metódust:

// Correct

this.setState({comment: 'Hello'});

Az egyetlen hely ahol a this.state értékadást használjuk az a konstruktor

### A state változás lehet aszinkron is

A react több setState() hívást egyetlen módosításként akár kötegelhet is, a teljesítmény miatt.

Mivel a this.props és a this.state aszinkronan frissítődnek, nem szabad az értékükre hagyatkozni, amikor a következő state-et meghatározzuk.

Például ez a kódrészlet a számláló frissítésének hibájához vezethet:

// Wrong

this.setState({

counter: this.state.counter + this.props.increment,

});

Ahhoz, hogy elkerüljük ezt, a setState() egy másik változatás kell használni ami egy függvényt vár, és nem objektumot. Ez a függvény megkapja az előző state-et első argumentumban, és a props-t akkor, ha a módosítás kész a második argumentumon.

// Correct

this.setState((state, props) => ({

counter: state.counter + props.increment

}));

Vagy hagyományos függvénnyel:

// Correct

this.setState(function(state, props) {

return {

counter: state.counter + props.increment

};

});

**Eseménykezelés**

Hasonló a DOM eseményekhez, néhány különbség van csupán:

* A react események camelCase elnevezésűek, és nem lowercase-ek.
* A JSX segítségével egy függvényt adunk át mint esemény kezelő, és nem string-ként írjuk.

Például eddigi tanulmányaink során így csináltuk:

<button onclick="activateLasers()">

Activate Lasers

</button>

React-ban pedig:

<button onClick={activateLasers}>

Activate Lasers

</button>

When using React you should generally not need to call addEventListener to add listeners to a DOM element after it is created. Instead, just provide a listener when the element is initially rendered. A react-ban nem kell meghívni az addEventListener-t hogy eseménykezelőt adjunk a DOM elemnek, miután létrehoztuk azt.

Ha osztályként definiálunk egy komponenst, akkor az eseménykezelő az osztály egy metódusa. Például a Toggle komponens egy gombot render-el, ami bakapcsolt (ON), kikapcsolt(OFF) állapotban lehet.

class Toggle extends React.Component {

constructor(props) {

super(props);

this.state = {isToggleOn: true};

// This binding is necessary to make `this` work in the callback

this.handleClick = this.handleClick.bind(this);

}

handleClick() {

this.setState(state => ({

isToggleOn: !state.isToggleOn

}));

}

render() {

return (

<button onClick={this.handleClick}>

{this.state.isToggleOn ? 'ON' : 'OFF'}

</button>

);

}

}

ReactDOM.render(

<Toggle />,

document.getElementById('root')

);

[**Try it on CodePen**](https://codepen.io/gaearon/pen/xEmzGg?editors=0010)

# Feltételes renderelés

A react-ban különböző komponenseket készíthetünk, amelyek egy-egy viselkedést írnak le. Ezek közül csak néhányat render-erünk, az alkalmazás állapotától függően.

A feltételes kirajzolás (render-elés) react-ban hasonlóképpen működik, mint a feltételes utasítások JavaScript-ben. A JavaScript if-et használhatjuk, mint feltételes operátort, hogy elemeket hozzunk létre, amelyek az aktuális állapotot reprezentálják, és a react így módosítja a UI-t eszerint.

Vegyük ezt a két komponenst:

function UserGreeting(props) {

return <h1>Welcome back!</h1>;

}

function GuestGreeting(props) {

return <h1>Please sign up.</h1>;

}

Egy Greeting komponenst kászítünk, ami az egyik komponenst rajzolja ki attól függően, attól függően, hogy a felhasználó be van-e jelentkezve.

function Greeting(props) {

const isLoggedIn = props.isLoggedIn;

if (isLoggedIn) {

return <UserGreeting />;

}

return <GuestGreeting />;

}

ReactDOM.render(

// Try changing to isLoggedIn={true}:

<Greeting isLoggedIn={false} />,

document.getElementById('root')

);

[**Try it on CodePen**](https://codepen.io/gaearon/pen/ZpVxNq?editors=0011)

A példa különböző szöveget ír ki attól függően, hogy be van-e jelentkezve (isLoggedIn prop értéke) a felhasználó.

### Elem változók

Vltozókat is használhatunk, hogy elemeket tároljunk. Ez segít abban, hogy feltételesen render-eljünk komponens részeit, míg a kimenet többi része nem változik.

Vegyünk ezt a két új komponenst, ami Logout és Login gombokat reprezentál:

function LoginButton(props) {

return (

<button onClick={props.onClick}>

Login

</button>

);

}

function LogoutButton(props) {

return (

<button onClick={props.onClick}>

Logout

</button>

);

}

A példában egy LoginControl komponenst készítünk. Ez van a <LoginButton /> vagy a <LogoutButton /> rendereli az aktuális state-enk megfelelően. A <Greeting /> -et is használja (előző példa).

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>

);

}

}

ReactDOM.render(

<LoginControl />,

document.getElementById('root')

);

[**Try it on CodePen**](https://codepen.io/gaearon/pen/QKzAgB?editors=0010)

### Inline If && operátor segítségével

A {}-be bármilyen kifejezést beírhatunk. Még a JavaScript logikai && operátort is.

function Mailbox(props) {

const unreadMessages = props.unreadMessages;

return (

<div>

<h1>Hello!</h1>

{unreadMessages.length > 0 &&

<h2>

You have {unreadMessages.length} unread messages.

</h2>

}

</div>

);

}

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

ReactDOM.render(

<Mailbox unreadMessages={messages} />,

document.getElementById('root')

);

[**Try it on CodePen**](https://codepen.io/gaearon/pen/ozJddz?editors=0010)

Ez azért működik, mert JavaScript-ben a true & expresssion mindig expression-re értékelődik ki és a false&&expression pedig mindig false-ra. Tehát a a feltétel igez, akkor a && jobb oldalán lévő kifejezés megjelenik kimenetként. Ha hamis akkor pedig nem.

### Inline If-Else with Conditional Operator

Another method for conditionally rendering elements inline is to use the JavaScript conditional operator [condition ? true : false](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Operators/Conditional_Operator).

In the example below, we use it to conditionally render a small block of text.

render() {

const isLoggedIn = this.state.isLoggedIn;

return (

<div>

The user is <b>{isLoggedIn ? 'currently' : 'not'}</b> logged in.

</div>

);

}

It can also be used for larger expressions although it is less obvious what’s going on:

render() {

const isLoggedIn = this.state.isLoggedIn;

return (

<div>

{isLoggedIn ? (

<LogoutButton onClick={this.handleLogoutClick} />

) : (

<LoginButton onClick={this.handleLoginClick} />

)}

</div>

);

}