# Seminar 10

## React

"React" este o bibliotecă JavaScript open-source dezvoltată și întreținută de Facebook. Scopul principal al React este să ajute dezvoltatorii să creeze interfețe de utilizator interactive și reutilizabile pentru aplicații web. React a fost proiectat pentru a facilita dezvoltarea **SPA-urilor** (Single Page Applications), unde o singură pagină web poate încărca și afișa **conținut dinamic** fără a fi nevoie să reîncarce întreaga pagină.

React este, de asemenea, **declarativ** și **bazat pe componente**.

### ❖ Single Page Applications (SPAs)

Un SPA este o aplicație web care funcționează într-o singură pagină web**, fără a necesita încărcarea completă a paginii pentru fiecare acțiune a utilizatorului**. În loc de aceasta, SPA **încarcă o singură pagină inițială** și apoi actualizează conținutul acelei pagini pe măsură ce utilizatorul interacționează cu aplicația, folosind tehnologii precum AJAX (Asynchronous JavaScript and XML) sau, mai recent, API-uri de interfață de programare a aplicațiilor (API-uri).

După încărcarea inițială, orice actualizare a conținutului sau navigare ulterioară este realizată prin intermediul AJAX sau a altor tehnologii, fără a necesita încărcarea completă a paginii. Utilizatorii pot experimenta o navigare mai fluidă și o încărcare mai rapidă a conținutului, deoarece doar părți mici ale paginii sunt actualizate.

### ❖ Multi Page Applications (MPAs)

Un MPA este o aplicație web care constă din **mai multe pagini web distincte**, fiecare având un **conținut și funcționalități proprii**. Atunci când utilizatorul navighează între diferite secțiuni ale aplicației sau face clic pe legături, serverul trimite o solicitare la server pentru a încărca o pagină nouă.

Folosind MPA, un server trebuie **să reîncarce majoritatea resurselor**, cum ar fi HTML, CSS și scripturile JS, cu fiecare interacțiune. Atunci când se încarcă o altă pagină, browserul reîncarcă complet datele paginii și descarcă din nou toate resursele, inclusiv componente care se repetă pe toate paginile, cum ar fi antetul și subsolul. **Aceasta afectează negativ viteza și performanța.**

**Diferența principală între MPA și SPA** constă în modul în care paginile sunt încărcate și actualizate. Într-o MPA, **întregul HTML al unei pagini este reîncărcat** de fiecare dată când utilizatorul accesează o pagină nouă. Acest lucru poate duce la o experiență mai lentă și poate implica timp de încărcare perceptibil mai mare între tranzițiile de pagină.

# ❖ Client Side Rendering vs Server Side Rendering

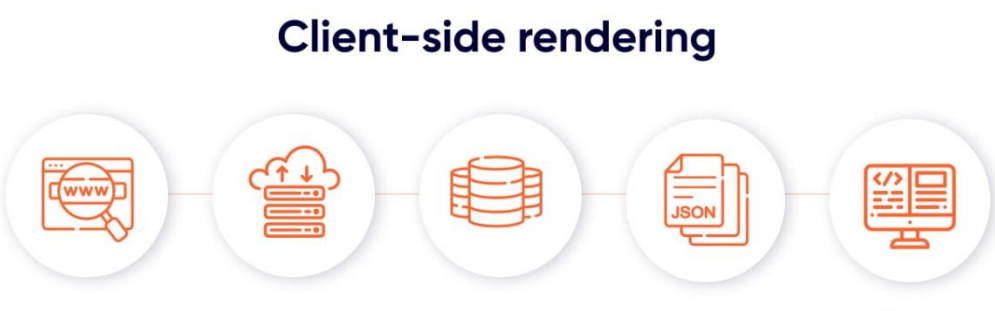
Atunci cand discutăm despre **SSR**, codul este **procesat pe server**. Vizualizarea paginii este generată în partea din spate și trimisă către partea din față (cunoscută și sub denumirea de client sau pur și simplu browser).

Procesul pas cu pas:

1. Un utilizator introduce o adresă URL în bara de adrese a browserului.
2. Se trimite o cerere de date către server la adresa URL specificată.
3. Serverul generează un fișier HTML cu datele și stilurile necesare pe baza solicitării primite de la partea din față.
4. Serverul trimite fișierul HTML ca răspuns către browser.
5. Browserul execută HTML-ul și afișează pagina.


Atunci când discutăm despre **CSR**, codul aplicației este trimis la **browserul utilizatorului**, unde este procesat și transformat în pagini vizibile și interactive. Renderizarea în partea clientului are loc în următorul mod:

1. Un utilizator introduce o adresă URL în bara de adrese a browserului.
2. Se trimite o cerere de date către server la adresa URL specificată.
3. Serverul extrage informații dintr-o bază de date și generează un răspuns în formatul solicitat, cum ar fi JSON.
4. Serverul trimite datele solicitate către client.
5. Browserul afișează pagina prin inserarea datelor obținute în codul HTML. Browserul face pagina vizibilă utilizatorului prin executarea codului JavaScript stocat în partea clientului.



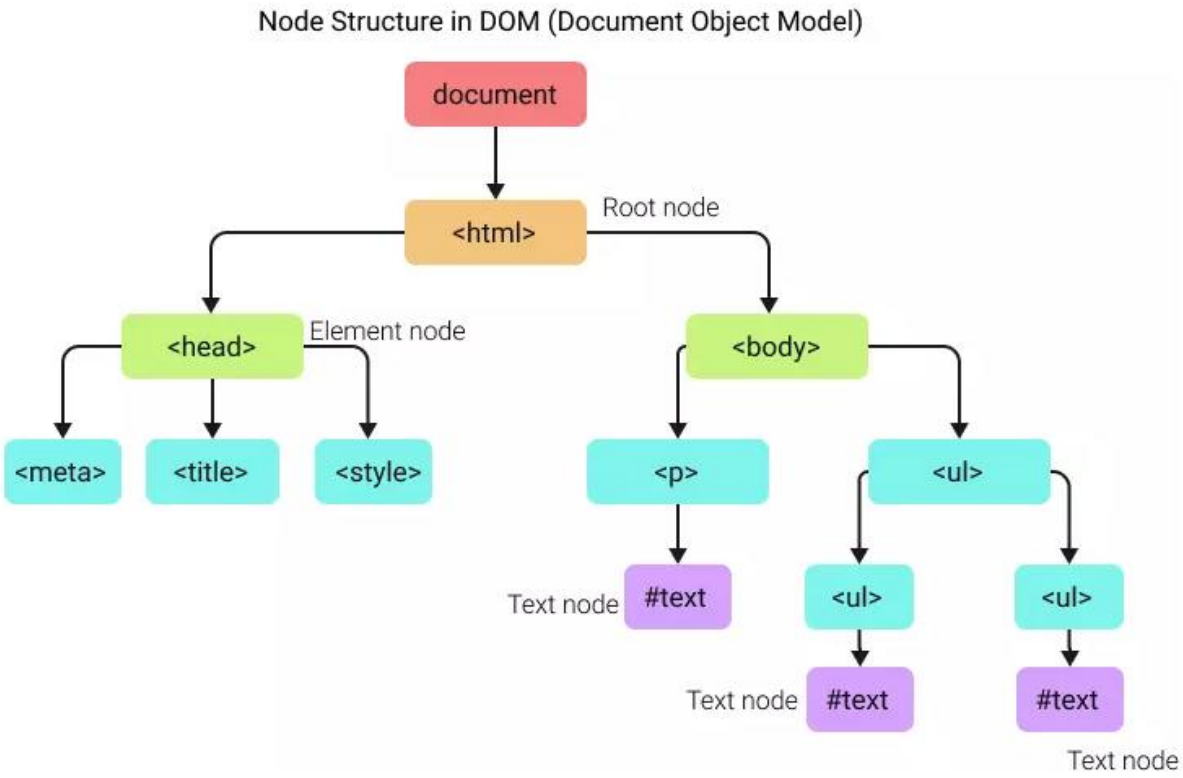

## ❖ Declarative vs Imperative

Explicația cea mai obișnuită a diferenței dintre programarea **imperativă** și cea **declarativă** este faptul că atunci când vorbm despre **cod imperativ**, spunem computerului **cum să facă lucrurile**, în timp ce programarea **declarativă** se concentrează **pe ceea ce dorești de la computer**.

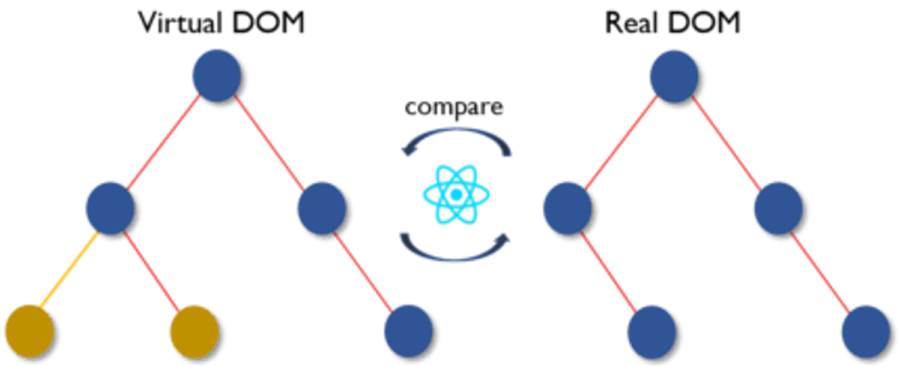
```
// IMPERATIVE
const doubleMapImperative = (arr) => {
    const result = [];

    for (const element of arr) {
        result.push(element * 2);
    }

    return result;
}

// DECLARATIVE
const doubleMapDeclarative = (arr) => arr.map((item) => item * 2);

console.log(doubleMapImperative([1, 2, 3]));
console.log(doubleMapDeclarative([1, 2, 3]));
```

## ❖ Virtual DOM

DOM tratează un document XML sau HTML ca o structură de arbore în care fiecare nod este un obiect care reprezintă o parte a documentului.




React creează un arbore de obiecte personalizate care reprezintă o parte a DOM-ului. În loc să creeze efectiv un element DIV care conține un element UL, el creează un obiect React.div care conține un obiect React.ul. Poate manipula aceste obiecte foarte rapid fără a atinge efectiv DOM-ul real sau a folosi API-ul DOM. Apoi, atunci când randează o componentă, folosește acest DOM virtual pentru a determina ce trebuie să facă cu DOM-ul real pentru a face ca cei do arbori să se potrivească.




Când starea unui obiect se schimbă într-o aplicație React, DOM-ul virtual (VDOM) se actualizează. Apoi, compară starea sa anterioară și actualizează doar acele obiecte în DOM-ul real în loc să actualizeze toate obiectele. Acest lucru face ca procesul să fie rapid, în special în comparație cu alte tehnologii de front-end care trebuie să actualizeze fiecare obiect chiar dacă se schimbă doar un singur obiect în aplicația web.

## ❖ Componentele în React

Componentele ne permit să împărțim interfața de utilizare în piese independente, reutilizabile și să ne gândim la fiecare piesă în mod izolat. Această pagină oferă o introducere în ideea de componente.

Avantajele folosirii componentelor:

1. **Reutilizare** - O componentă utilizată într-o parte a aplicației poate fi refolosită în altă parte. Acest lucru ajută la accelerarea procesului de dezvoltare.
2. **Structurare modulară** - Organizarea unui sistem sau a unei aplicații în componente sau module independente care îndeplinesc funcții specifice și care pot fi gestionate și dezvoltate separat
3. **Interfață clară** - Modulele definesc interfețe clare între diferite părți ale sistemului. Aceasta face ca integrarea și comunicarea între module să fie mai ușoară și mai predictibilă.
4. **Ușurința în întreținere** - Prin organizarea codului în module, mentenanța devine mai ușoară. Problemele pot fi izolate la nivelul modulelor specifice, iar actualizările sau îmbunătățirile pot fi implementate fără a afecta întregul sistem.

## ❖ JSX

JSX (JavaScript XML) este o extensie a limbajului JavaScript și este utilizat în mod frecvent în React pentru a descrie structura interfeței de utilizator (UI).

Codul JSX **nu este suportat direct de către browser** (este doar un "syntactic sugar") și este tradus în cod JavaScript în timpul execuției (runtime).

JSX permite salvarea structurii similare HTML în variabile. În React, poți utiliza JSX pentru a crea elemente React, și aceste elemente pot fi stocate în variabile.

```
import React from 'react';

// Creare element React folosind JSX și salvare într-o variabilă
const myElement = <div>Hello, World!</div>;

// Utilizarea variabilei într-un alt loc
function MyComponent() {
  return (
    <div>
      {myElement}
      <p>Another element</p>
    </div>
  );
}

export default MyComponent;
```

## ❖ Crearea unui proiect de React

Pentru crearea unei aplicații de client cu React, o să rulăm următoarea comandă:

```
npx create-react-app my-app
```

**Notă: Trebuie să ne asigurăm că avem instalat Node.js >= 14.0.0**

După crearea aplicației, trebuie să ne mutăm cu terminalul în folderul **my-app** și să pornim proiectul. Pentru asta, o să scriem următoarele comenzi în consolă:

```
cd my-app
npm start
```

Trebuie să ținem cont de faptul că aplicația noastră de client **nu se ocupă de logica backend** sau **de baze de date**. El doar creează o linie de producție pentru frontend, astfel încât să poți utiliza orice backend dorim.

După ce am creat proiectul și l-am pornit, mergem pe http://localhost:3000 și o să putem să vedem acest mesaj:


Node nu este necesar pentru a utiliza React. Nu avem nevoie de Node pentru a rula un proiect React. Ceea ce oferă Node sunt mai degrabă un set de instrumente care ne permit să lucrăm mai ușor cu React, cum ar fi **Webpack** (preia toate resursele și modulele, cum ar fi fișierele JavaScript, CSS, imagini și altele, și le ambalează într-un set eficient de fișiere pentru a le livra într-un mod optimizat către browser) și **Babel** (care convertește ES6 și JSX în JavaScript simplu).

## ❖ Starea componentelor în React

Starea în React este un concept esențial pentru dezvoltarea aplicațiilor interactive și dinamice. Când vorbim despre "reactivitate", **ne referim la capacitatea aplicației de a răspunde la schimbările** în starea sa și de a actualiza interfața utilizatorului în consecință.

Starea unei componente poate să se schimbe în timp. Ori de câte ori se schimbă, componenta se recreează.

Schimbarea stării poate să apară ca răspuns la acțiunile utilizatorului sau la evenimente generate de sistem, iar aceste schimbări determină comportamentul componentei și modul în care aceasta va fi randată.

Este important să menționăm că React nu reinițializează starea la fiecare re-renderare. În schimb, el preia doar cea mai recentă imagine a stării, menținând coerentă experiența utilizatorului. Prin gestionarea stării, poți crea aplicații dinamice și interactive care răspund în timp real la acțiunile utilizatorului și la evenimente ale sistemului.

În plus, datele transmise prin intermediul părintelui (props) se presupune ca doar sunt citite, ceea ce înseamnă că nu ar trebui să fie modificate de către componente copil.

## ❖ Hooks

Hook-urile sunt funcții speciale oferite de React care permit folosirea stării și a altor caracteristici ale claselor în cadrul componentelor funcționale. Ele pot fi apelate doar din cadrul componentelor funcționale de React și nu le putem folosi în cod JavaScript regulat.

1. **useState -** este un hook în React care permite componentelor funcționale să își gestioneze starea internă. Acesta furnizează o metodă de declarare și actualizare a stării în cadrul componentelor funcționale.

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

function Contor() {
  // Declararea stării cu ajutorul useState-ului
  const [contor, setContor] = useState(0);

  return (
    <div>
      <p>Valoarea contorului: {contor}</p>
      <button onClick={() => setContor(contor + 1)}>Incrementează</button>
    </div>
  );
}

export default Contor;
```

Pașii care au fost folosiți pentru a crea o componentă cu acest hook:

1. În cadrul proiectului, creăm o nouă componentă cu numele **„Contor”**
2. Creăm boilerplate-ul unei funcții goale care nu returnează nimic în return și exportăm funcția
3. Scriem linia prin care inițializăm locul în care salvăm valoarea contorului și metoda prin care îi actualizăm valoarea și atribuim o valoarea inițială.
4. În return, începem să adăugăm conținut HTML și să adăugăm un buton, care la click o să incrementeze valoarea contorului și pe care îl afișează în mod constant în browser.

**De reținut**: de fiecare dată când o variabilă declarată cu useState își va schimba valoarea, o sa forțeze toată componena să își dea re-render.

2. **useEffect -** este folosit pentru a gestiona efecte secundare în cadrul componentelor funcționale. Efectele secundare pot fi, de exemplu, operațiuni asincrone, manipulări ale DOM-ului, abonări la evenimente sau orice altceva care nu este legat direct de redarea (rendering) componentei.

```jsx
import React, { useState, useEffect } from 'react';

function ExempluData() {
  const [date, setDate] = useState([]);

  const fetchData = async () => {
    try {
      const response = await fetch('https://api.example.com/data');
      const data = await response.json();
      setDate(data);
    } catch (error) {
      console.error('Eroare la încărcarea datelor:', error);
    }
  };

  useEffect(() => {
    // Apelarea funcției pentru încărcarea datelor
    fetchData();
  }, []); // Efectul se va rula doar o dată la montarea componentei

  return (
    <div>
      <h2>Date Încărcate:</h2>
      <ul>
        {date?.map(item => (
          <li key={item.id}>{item.name}</li>
        ))}
      </ul>
    </div>
  );
}

export default ExempluData;
```

utilizat in Test si in ShowData din seminar 10(in ShowData e exemplu cu async

De reținut:

1. Primul parametru al useEffect-ului este o funcție care conține codul efectului. Al doilea parametru este un array de dependențe, opțional.
2. Al doilea parametru este un array de dependențe. Dacă acest array este furnizat, efectul va fi re-rulat când oricare dintre dependențe se schimbă. Dacă array-ul este gol, acțiunea va avea loc doar în faza de creare a componentei.
3. Codul asincron poate fi folosit în cadrul acestui hook, doar dacă funcția care face apelul asincron este declarată în interiorul hook-ului și apelată imediat. Dacă funcția este declarată în afară, doar o să apelăm funcția fără să precizăm că este o funcție asincronă.
4. Dacă folosim variabile în interiorul efectului, trebuie să ne asigurăm că le includem în lista de dependențe dacă acestea sunt utilizate. Acest lucru poate ajuta la evitarea potențialelor probleme de sincronizare.

## ❖ Randarea condiționată

1. **Operator ternar**: se folosește operatorul ternar care va afișa mesajul “Welcome back!” dacă utilizatorul a apăsat pe butonul de “Log in” și valoare useState-ului devine **true**.

```
import React from 'react';

const App = () => {
  const [isLoggedIn, setIsLoggedIn] = React.useState(false);

  return (
    <div>
      {isLoggedIn ? (
        <h1>Welcome back!</h1>
      ) : (
        <button onClick={() => setIsLoggedIn(true)}>Log in</button>
      )}
    </div>
  );
};

export default App;
```

2. **Prin intermediul operatorului logic &&**: dacă dorim să afișăm conținut doar atunci când o variabilă are o anumită valoarea dar când nu are nu vrem să afișăm nimic, atunci o să folosim operatorul logic &&. În exemplul de mai jos, afișăm lista de comenzi doar dacă lungimea array-ului este mai mare ca 0.

```jsx
import React, {useState} from 'react';

const App = () => {
    const [orders, setOrders] = useState([]);

    return (
        <div className="App">
        <header className="App-header">
            <p> Hello World! </p>
            {orders.length > 0 && <p> You have {orders.length} orders </p>}
        </header>
        </div>
    );
}

export default App;
```

## ❖ Liste și chei

Cheile ajută React să identifice **care elemente s-au schimbat**, **au fost adăugate** sau **au fost eliminate**. Cheile ar trebui să fie atribuite elementelor din interiorul array-urilor pentru a le oferi acestora o identitate stabilă.

Cu alte cuvinte, cheile sunt necesare doar în timpul re-randărilor și pentru elemente vecine de același tip.

React folosește un concept special atunci când vine vorba de randarea listelor de date, un concept care asigură eficiența actualizării și randării:

1. De obicei, dacă nu este furnizată o cheie, **React va vizita toate elementele listei și le va actualiza** (posibile probleme de performanță și erori)
2. Prin utilizarea cheilor, putem preciza cu exactitate **unde trebuie adăugat un element nou**.
3. Pentru identificare, la nivelul fiecărui element se adaugă tag-ul de “key” care trebuie să primească o cheie unică.
4. Dacă atunci când mapăm elementele, ele nu dețin un id unic, atunci putem să ne folosim de indexul pe care ni-l pune la dispoziție metoda **map**.

## ❖ Props

În React, termenul "props" se referă la proprietățile unui obiect care sunt pasate de la componenta părinta la cea copil, cu scopul ca copilul să facă ceva cu datele primite. Acest lucru se datorează faptului că toate elementele create în React sunt obiecte JavaScript. Ca rezultat, transmiterea datelor către un component se realizează prin crearea de proprietăți și valori pentru obiect.

Astfel, putem observa în exemplul de mai jos că avem componenta părinte "**App**" care are o listă de comenzi pe care le mapează, și o componentă copil numită "**Order**" care afișează pentru fiecare comandă în parte, numele și prețul. Totodată, **ne-am asigurat că există comenzi înainte de a începe procesul** prin semnul întrebării pe care l-am adăugat înainte de maparea efectivă și faptul că pentru fiecare comandă în partea am adăugat și o **cheie**.

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

const App = () => {
    const [orders, setOrders] = useState([]);

    return (
        <div className="App">
        <header className="App-header">
            <p> Hello World! </p>
            {orders?.map((order) => (
                <Order order={order.name} key={order.id} />
            ))}
        </header>
        </div>
    );
}

export default App;

const Order = (order) => {
    return (
        <div>
            <p> {order.name} </p>
            <p> {order.price} </p>
        </div>
    );
}

export default Order;
```

## ❖ Hook-uri custom

Hook-urile custom în React sunt funcții JavaScript care utilizează unul sau mai mulți hook-uri preexistente și adaugă o anumită funcționalitate sau logică reutilizabilă. Aceste hook custom poate fi creat de către dezvoltatori pentru a împărți și reutiliza logică comună între componente.

```
import {useState, useEffect} from 'react';

const useFetch = (url) => {
    const [data, setData] = useState(null);
    const [error, setError] = useState(null);

    useEffect(() => {
        fetch(url)
            .then(response => response.json())
            .then(data => setData(data))
            .catch(error => setError(error));
    }, [url]);

    return {data, error};
}

const ShowData = () => {
    const {data, error} = useFetch('https://jsonplaceholder.typicode.com/todos');

    if (error) {
        return <p>{error}</p>
    }

    return (
        <div>
            {data?.map((todo, index) => (
                <p key={index}>{todo.title}</p>
            ))}
        </div>
    )
}
```