Mihai Damian

Tehnologiile APLICAȚIILOR WEB

UTPRESS
Cluj-Napoca, 2022
ISBN 978-606-737-568-8

Mihai DAMIAN

TEHNOLOGIILE APLICAŢIILOR WEB

UTPRESS
Cluj - Napoca, 2022
ISBN 978-606-737-568-8

Editura U.T.PRESS
Str. Observatorului nr. 34
400775 Cluj-Napoca
Tel.: 0264-401.999
e-mail: utpress@biblio.utcluj.ro
http://biblioteca.utcluj.ro/editura

Director: ing. Calin D. Campean

Recenzia: Prof.dr.ing. Liviu Miclea
Conf.dr.ing. Enyedi Szilárd

Pregătire online: Gabriela Groza

ISBN 978-606-737-568-8

Bun de tipar: 17.03.2022

Cuprins

Introducere

În 1999, datorită unor neînțelegeri și întârzieri legate de crearea unui curs destinat creării site-urilor web, a trebuit să învăț "pe repede înainte" cum se realizează un site. Așa am făcut cunoștință cu multitudinea de elemente din limbajul HTML și cu modul în care funcționează World Wide Web.

Totul a mers lin până când am scris capitolul destinat codificării formularelor. Aici lucrurile au luat o întorsătură dramatică.

Pe de o parte cărțile pe care le-am studiat pentru a scrie cursul până în acel punct se încheiau brusc după prezentarea formularelor. Afișau o scuză legată de complexitatea programării aplicației destinate preluării datelor din formulare și gata.

Pe de altă parte am realizat că tehnologia predată reprezenta o bună parte din ceea ce este necesar pentru a crea aplicațiile software cele mai spectaculoase, aplicațiile web. Dar din păcate aceste aplicații au două părți iar misterul realizării părții de server părea, pentru moment, inaccesibil.

Soluția a venit de unde mă așteptam mai puțin. Fiind în vizită la o universitate din Franța, unul dintre profesorii cu care am discutat mi-a prezentat, printre altele, faptul că se ocupa cu organizarea unui congres. Și mi-a arătat site-ul manifestării, care avea, desigur, un formular de înscriere.

Evident, n-am ratat ocazia de a-l întreba cum face preluarea datelor din formular. Fără să stea pe gânduri, mi-a imprimat imediat un listing cu o aplicație scrisă în C. Și mi-a explicat în câteva cuvinte și unde trebuie să fie pusă aceasta pe server, ce configurări mai trebuiau realizate și cum funcționează preluarea datelor din formulare. Super!

Revenit din misiune, am instalat un Linux pe un calculator, am cerut colegilor care gestionau rețeaua universității un IP static și am transformat calculatorul în server. Pentru a fi server de web am instalat și Apache, desigur. Apoi am pus pe noul meu server o pagină web conținând un formular cu două câmpuri. Care, minune, era accesibilă prin Internet.

Dar și mai spectaculoasă a fost testarea funcționării butonului de tip `submit` al formularului. Pentru aceasta am încărcat pe server un mic program scris în C după modelul deja cunoscut, l-am compilat și s-a produs și a doua minune, programul scris de mine chiar prelua datele din formular și trimitea browserului o pagină conținând valorile citite. O pagină dinamică. Sau, pentru mine, începutul erei aplicațiilor web!

Încă de la această primă și (foarte) mică aplicație am realizat că lumea aplicațiilor web va fi fascinantă dar complexă. Dacă doar pentru a citi două valori dintr-un formular a trebuit să știu HTML, C, Linux, instalarea și configurarea unui server, gestionarea drepturilor de acces la fișiere și directoare...

Și impresia de atunci a rămas valabilă și astăzi.

Cartea aceasta prezintă principalele cunoștințe și tehnologii pe care se bazează aplicațiile web actuale. Tabloul expus cititorului prin informațiile și exemplele pe care le conține este însă complex. Scopul principal al lucrării nu este însă acela de a pregăti profesioniști ai domeniului cât acela de a crea o imagine cât mai veridică asupra creării aplicațiilor web. `The big picture!`

Parcurgerea ei poate fi urmată de o aprofundare a noțiunilor prin parcurgerea unor tutoriale disponibile pe diverse site-uri de specialitate. Astăzi așa se învață. Procedând în acest fel, cei pasionați pot să avanseze foarte rapid. Iar cei mai puțin apropiați de cele expuse pot să evalueze tehnologiile din domeniu și să-și dea seama de efortul necesar înțelegerii și aprofundării acestora încă înainte de a realiza prima aplicație. Iar, dacă aplicațiile web li se par complicate și, în multe locuri, fără sens, pot să se orienteze spre alte domenii.

CAPITOLUL 1

Interfețe grafice

Procesul de dezvoltare a unei aplicații software moderne poate fi separat în două părți care pot fi tratate separat, folosind limbaje și tehnologii diferite:

- *partea de "back-end"*, incluzând prelucrări de date, construirea soluțiilor și accesarea bazelor de date (componenta zisă de bussiness-logic) și
- *partea de "front-end"*, destinată reprezentării pe ecran a datelor și afișării instrumentelor necesare interacționării cu aplicația. În aplicațiile actuale această componentă se realizează folosind *framework*-uri bazate pe limbajul JavaScript: *React*, *Angular*, *Vue* ș.a..

Separarea celor două componente permite realizarea și testarea lor separată, cu instrumentele cele mai adecvate. Pentru integrarea celor două părți este necesară respectarea unei singure condiții: definirea specificațiilor pentru partea de *back-end* astfel încât aceasta să comunice natural cu partea de *front end*. Uneori componenta de *back-end* este defintă ca *serviciu web*. Acest lucru presupune conformarea acestei componente la cerințele unui standard denumit *Representational State Transfer* (prescurtat *REST*).

În cazul aplicațiilor Windows obișnuite (zise de tip *"desktop"*) separarea completă a celor două părți este opțională. Limbajele utilizate (Java, C# etc.) oferă suport pentru realizarea ambelor părți. Totuși, dat fiind faptul că aptitudinile și cunoștințele necesare programării celor două părți diferă, în cadrul echipelor care dezvoltă astfel de proiecte dezvoltarea interfeței grafice se realizează de către persoane specializate care au de cele mai multe ori o formare de *web designer* și/sau *web developer*. Desigur, această regulă este impusă și de considerente comerciale. Există multe exemple de

aplicații foarte bune dar a căror lansare este ratată datorită interfeței grafice neinspirate. În lumea aplicațiilor Windows sau Web, *bad UI/UX!* (eng. *User Interface / User eXperience*) este o afirmație destul de frecventă!

Desktop versus *web*

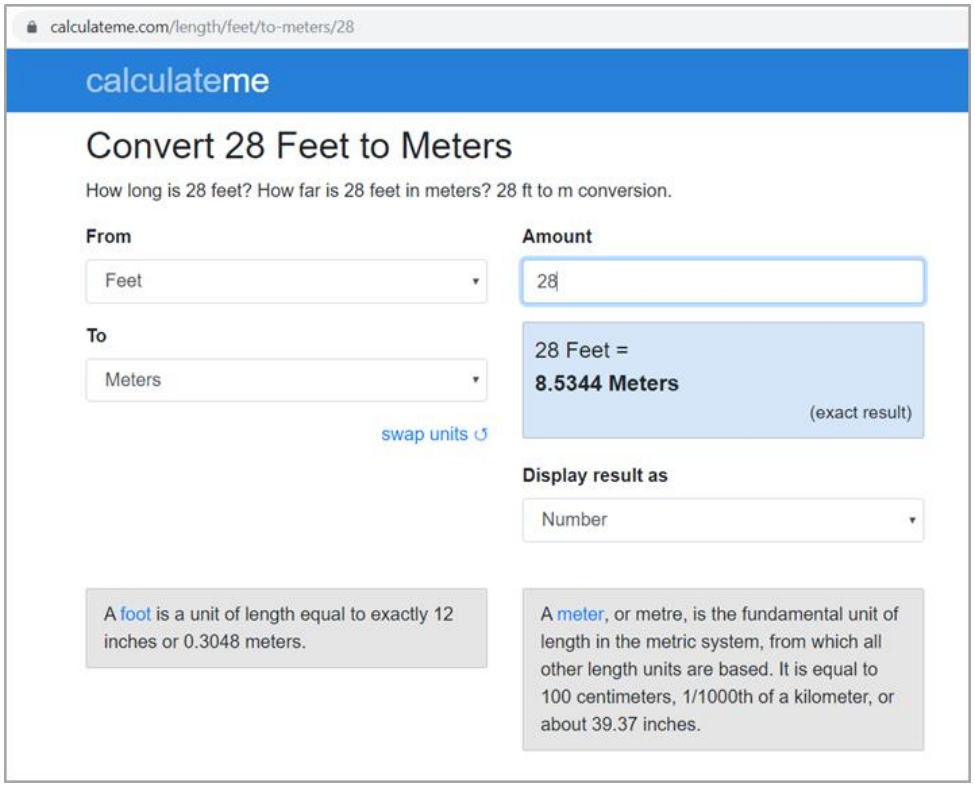
Dacă până în urmă cu câțiva ani a realiza o aplicație informatică era sinonom cu a realiza o aplicație tip *"desktop"*, astăzi lucrurile stau cu totul altfel. Posibilitățile de conectare disponibile fac ca majoritatea aplicațiilor să beneficieze din plin de arhitectura *client-server* și să fie practic aplicații web. Desigur, aplicațiile de tip *"desktop"* sunt cerute în continuare dar într-o mai mică măsură. Aceasta deoarece, de la un anumit nivel de complexitate în sus, în timpul dezvoltării unei aplicații software apar sistematic funcții care pot fi rezolvate cu costuri mult mai mici dacă se adoptă o arhitectură de tip aplicație web. O astfel de arhitectură permite de asemenea accesarea unor servicii specializate (aplicații accesibile online), oferite de multe ori gratuit de către firmele care le dezvoltă. Dacă ne gândim la banala accesare a unei baze de date sau, în aplicații mai avansate, la utilizarea comenzilor vocale, a recunoașterii faciale sau a unor funcții care presupun utilizarea inteligenței artificiale, soluția aplicației web este de departe cea mai comodă deoarece astfel de componente sunt livrate, împreună cu API-ul (eng. *Application Programming Interface*) aferent de către marii jucători din domeniul informaticii: Amazon, Google, Microsoft, Apple ș.a..

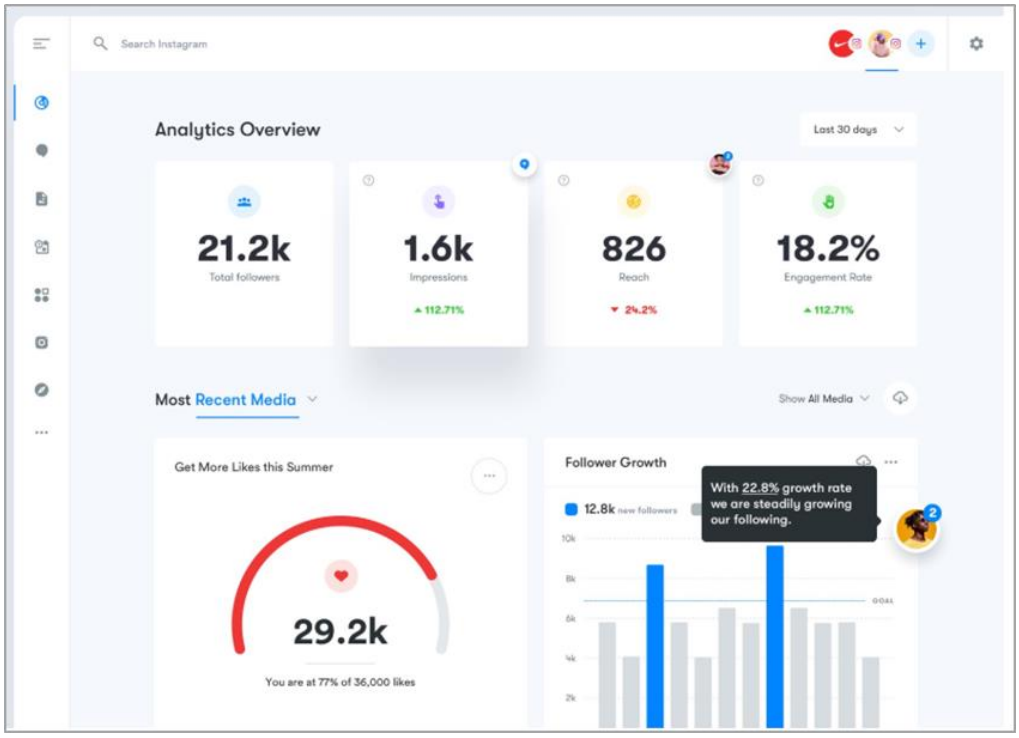
Există însă și o problemă a unei astfel de arhitecturi: accesul la rețeaua Internet. Dacă rețeaua Internet nu poate fi accesată, funcțiile care depind de aceasta se vor bloca.

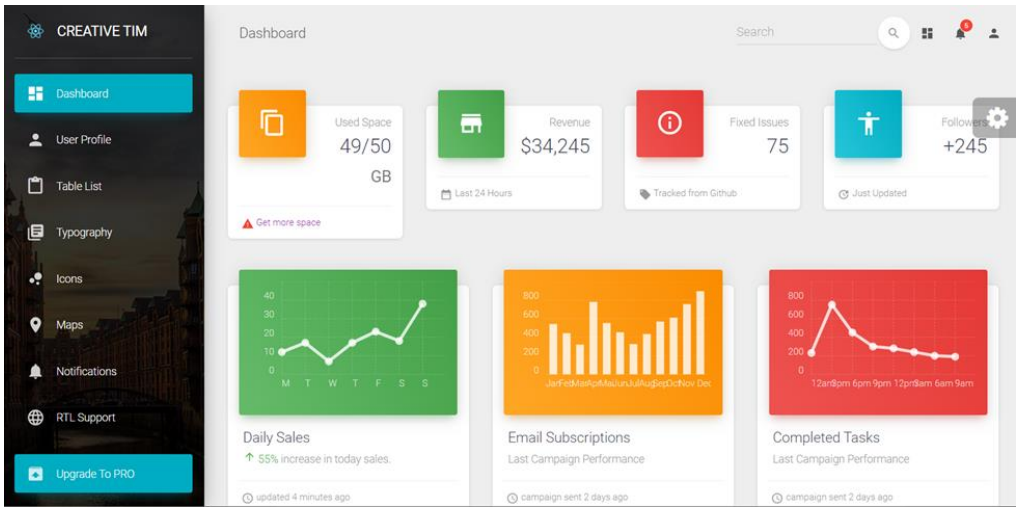
Caracteristica aplicațiilor în arhitectură *client-server* menționată deja, respectiv decuplarea părții care realizează reprezentarea pe ecran a informațiilor de componenta care produce aceste informații, permite utilizarea unor limbaje de programare diferite dar și rularea pe calculatoare diferite a celor două componente. Respectiv partea de *client* se rulează pe calculatorul utilizatorului în timp ce partea de *server* se rulează pe un calculator accesibil folosind rețeaua Internet.

În ceea ce urmează se va studia cu precădere partea de client a aplicațiilor, deci prezentarea va fi centrată pe expunerea pe ecran a datelor și pe asigurarea interactivității specifice.

Exemple de interfețe grafice:





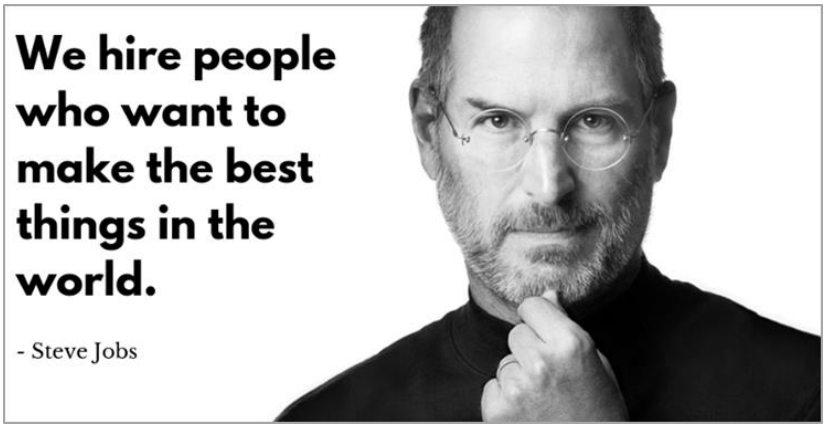


Interfețele expuse au diferite niveluri de complexitate. Primul exemplu, și cel mai simplu, presupune utilizarea câtorva elemente HTML5: `<div>`, `<p>`, `<h1>`, `<input>` sau `<select>`. Următoarele exemple sunt interfețe grafice ale unor aplicații web realizate folosind componente din ce în ce mai complexe, care se bazează pe utilizarea reprezentărilor grafice 2D (raster) și

a animațiilor. În contrast cu *elementele HTML5*, în continuare acestea vor fi numite *componente* ale interfeței grafice.

Obiectivul acestei lucrări va fi deci realizarea interfețelor grafice ale aplicațiilor software folosind tehnologii actuale. Interesul pentru asimilarea tehnologiilor propuse ar trebui să fie cu atât mai mare cu cât aceste cunoștințe sunt foarte valorizante pentru CV-ul oricărui *front-end developer*.

Observație: Cartea se adresează în special începătorilor. Sunt necesare totuși cunoștințe elementare de HTML5 și CSS3.



Soluții pentru realizarea interfețelor grafice

Odată cu adoptarea Windows-ului ca soluție de bază în realizarea aplicațiilor, competiția pentru realizarea unei grafici cât mai inspirate pentru instrumentele de interacțiune cu aplicațiile a crescut constant.

O bună perioadă însă aplicațiile au fost considerate bine realizate dacă afișau corect un ansamblu de controale Windows clasice, standard (fig. 1.1).

Odată cu dezvoltarea Internetului și a tehnologiilor web, pretențiile legate de interfețele grafice au crescut rapid. Beneficiarii aplicațiilor își doresc astăzi interfețe grafice cu nimic mai prejos decât cele expuse de aplicațiile web. Rezolvarea cea mai simplă a acestei cerințe este, desigur, adoptarea pentru realizarea interfeței grafice a unui *framework* destinat creării aplicațiilor web bazat pe HTML, CSS (*Cascading Style Sheets*) și JavaScript.

Observație: Termenul de *framework* este greu de tradus. Definiția lui în limba engleză este: *a supporting structure around which something can be built*. În informatică, un *framework* înseamnă de obicei un ansamblu de instrumente software și de reguli de scriere destinate realizării unei familii de aplicații. Nu este vorba de limbaje noi, ci doar de adoptarea unor tehnici de dezvoltare riguros definite, folosind limbajele cunoscute, ansamblul fiind completat cu instrumente software (aplicații specifice) destinate prelucrării codului scris și asamblării acestuia într-o aplicație funcțională.

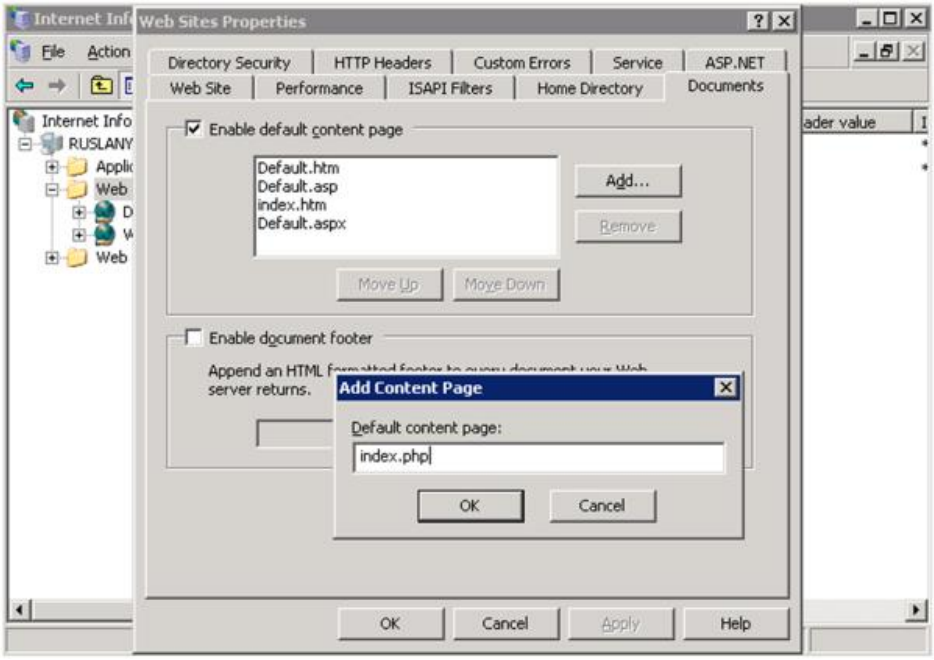
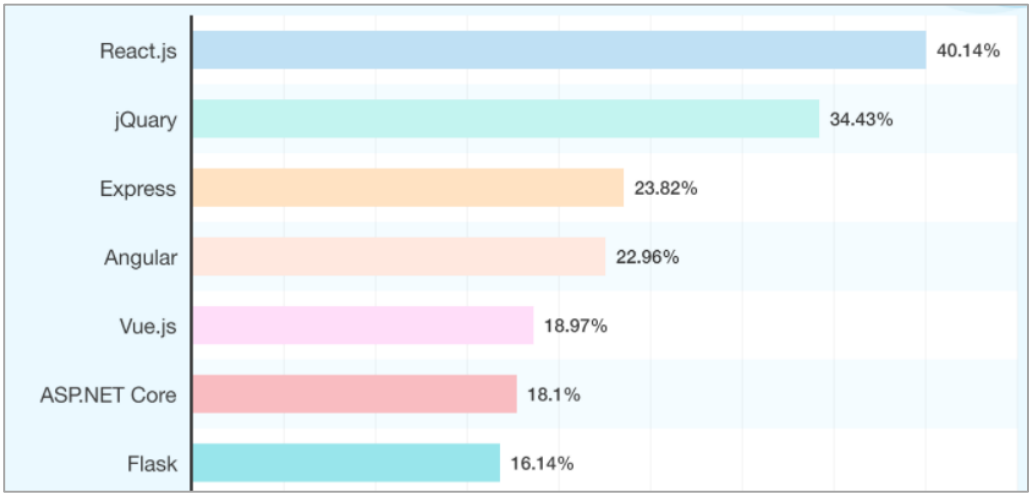


Figura 1.1. Interfață grafică realizată folosind componente Windows

Un *framework* ameliorează atât productivitatea scrierii aplicațiilor cât și calitatea acestora.

În reprezentarea următoare se prezintă principalele soluții utilizate pentru realizarea componentei de *front-end* a aplicațiilor web (statistică realizată în 2021).



Dacă se omite *jQuery*, care nu este altceva decât o colecție de mici funcții JavaScript ajutătoare, care permit ameliorări punctuale ale productivității programării în JavaScript, se poate vedea că *React* (creat în anul 2011 de Facebook Co.) și *Angular* (creat de Google Co.) sunt cele mai folosite framework-uri.

Desigur, în cazul utilizării unui astfel de framework, limbajul JavaScript devine singura soluție pentru scrierea codului pentru componenta de *front-end*.

Circulația informației în *front-end*

Front-end-ul unei aplicații informatice conține componente care afișează informații de diferite tipuri: șiruri de caractere, valori ale unor mărimi numerice, imagini și altele. Majoritatea acestor valori sunt produse în cadrul componentei de *back-end*. Alte informații sunt introduse de utilizator folosind formulare, acestea fiind destinate culegerii datelor necesare construirii unor comenzi adresate componentei de *back-end*.

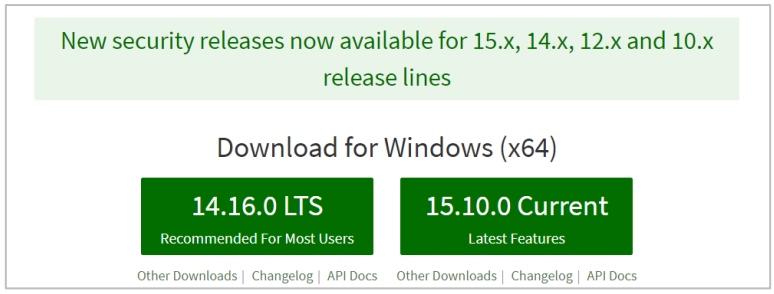
Soluțiile efective de afișare ale informațiilor din categoriile menționate fiind relativ limitate, activitățile specialiștilor care realizează componentele de *front-end* ale aplicațiilor sunt oarecum repetitive. Aceasta nu este în mod necesar un dezavantaj deoarece astfel se deschid posibilități de angajare în domeniu și pentru informaticienii începători.

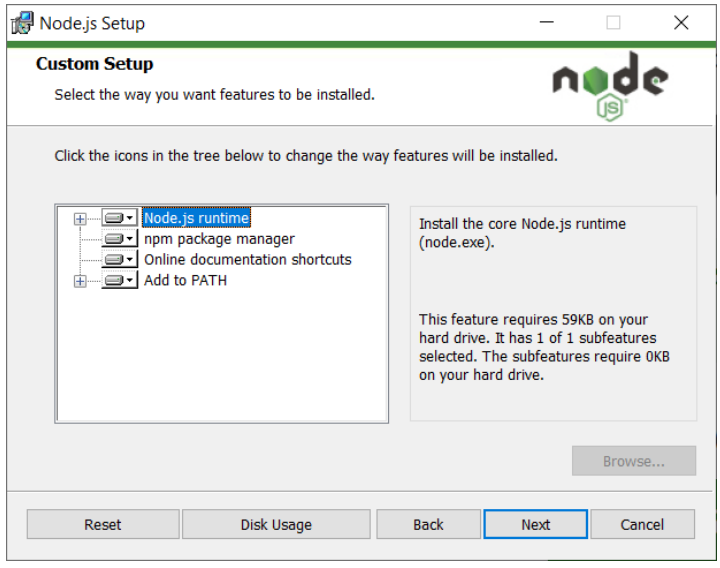
Instalarea aplicațiilor necesare

1. **Instalarea pachetului de aplicații *Node.js***

Node.js (*https://nodejs.org/en/*) este un pachet de aplicații, componenta centrală fiind un compilator de cod JavaScript. În principiu este vorba chiar de compilatorul integrat în Google Chrome, acesta fiind însă adaptat pentru crearea componentei de server a unei aplicații web. În continuare, pentru dezvoltarea aplicațiilor web, din pachetul *Node.js* vor fi utilizate trei componente:

- un miniserver web de dezvoltare, care va permite testarea continuă a aplicației realizate,
- *npm*, o aplicație care va fi utilizată pentru adăugarea la aplicația realizată a unor module dezvoltate de alte firme și, desigur,
- compilatorul *Node.js*.



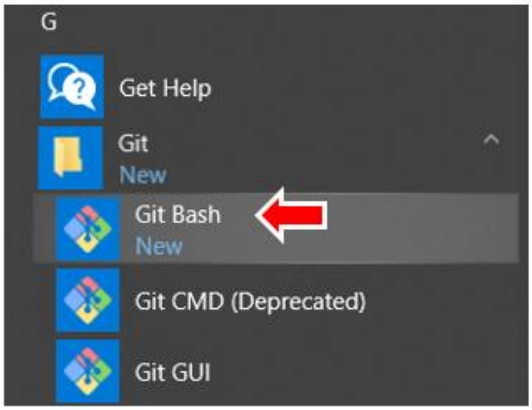


2. **Instalarea aplicației *Git*** (`https://git-scm.com/download/`)

`Visual Studio Code`, aplicația care va fi folosită pentru a crea aplicațiile, permite utilizarea aplicației `Git` pentru controlul versiunilor aplicației realizate.

În procesul de instalare nu se va face nicio configurare. Se va selecta sistematic butonul `Next`, acceptând astfel opțiunile propuse.

După instalare va mai trebui realizată doar o configurare prin care se stabilește identitatea utilizatorului aplicației. Pentru aceasta se pornește `Git Bash` (butonul `Start`, apoi se selectează `Git / Git Bash`):



Comenzile care trebuie introduse sunt următoarele:

```
git config --global user.name "Mircea Petrescu"
git config --global user.email mircea.petrescu@gmail.com
```

Ele definesc numele utilizatorului aplicației (`Mircea Petrescu`) și adresa de e-mail a acestuia și trebuie executate o singură dată, imediat după instalarea aplicației.

Este indicat ca acestea să fie corect scrise, deoarece mai târziu va fi folosită și aplicația web `GitHub` și este bine ca aceste date să nu difere de cele utilizate la crearea contului personal de pe platforma `GitHub`.

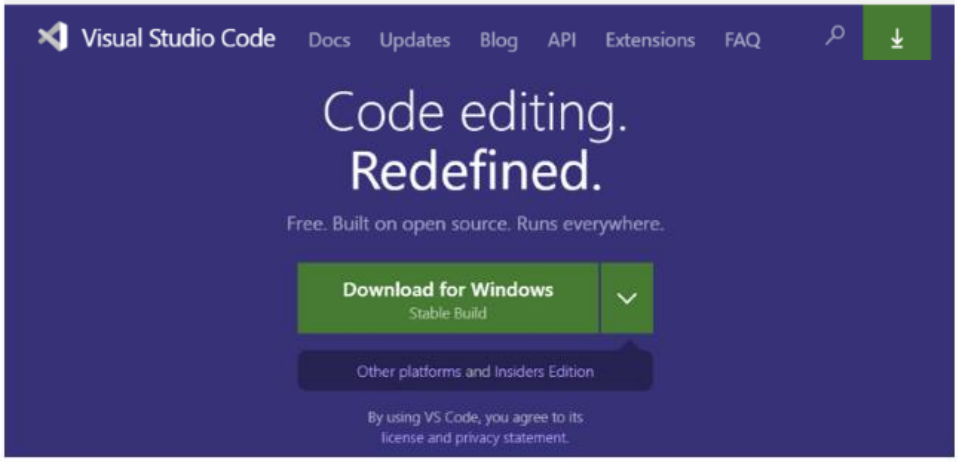
Observație: `GitHub` este o aplicație foarte mult utilizată pentru expunerea în Internet a unuia sau a mai multor proiecte informatice personale. Aceeași aplicație este utilizată de asemenea și pentru gestionarea versiunilor în cadrul unui proiect realizat în comun de către o echipă de dezvoltatori.

3. Instalarea aplicației *VS Code*

Pentru scrierea aplicațiilor React se poate utilila oricare dintre editoarele de texte folosite la crearea paginilor web. Dezvoltatorii folosesc frecvent *Sublime Text*, *Atom*, *WebStorm* sau *Visual Studio Code*. În cele ce urmează va fi folosit edtorul *Visual Studio Code* (prescurtat *VS Code*), acesta fiind recomandat de comunitatea dezvoltatorilor de aplicații React datorită setului de extensii care pot contribui la creșterea productivității.

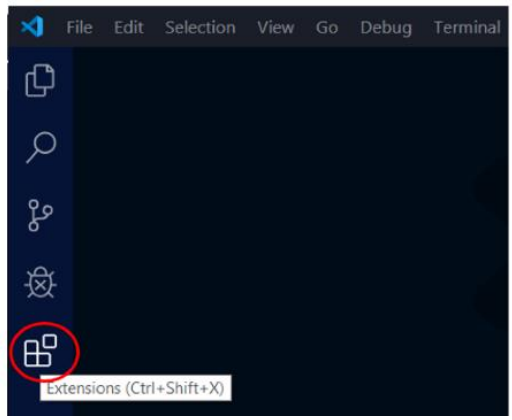
Aplicația este gratuită și poate fi descărcată de la adresa *https://code.visualstudio.com.*

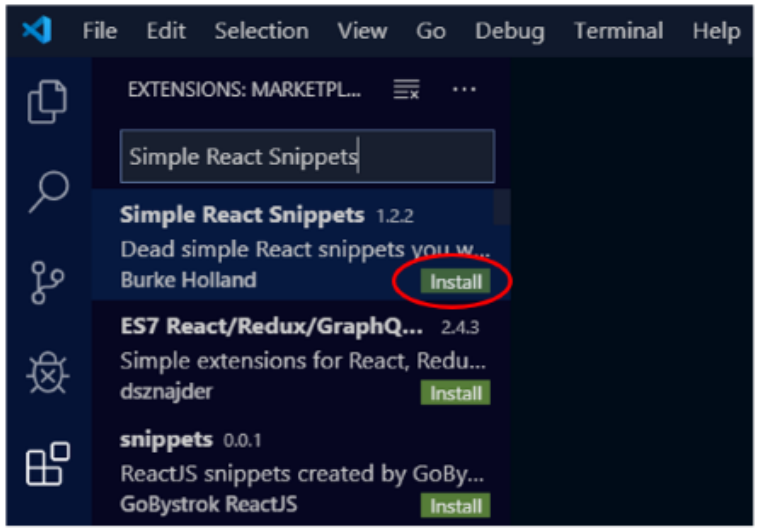
Observație: *VS Code* este și un excelent exemplu de aplicație de tip "desktop" realizată folosind *React*. Componenta de *back-end* a aplicației este realizată folosind *Node.js*. De fapt a fost folosit *Electron* (*https://electronjs.org/*), care le integrează pe cele două (*React* și *Node.js*).



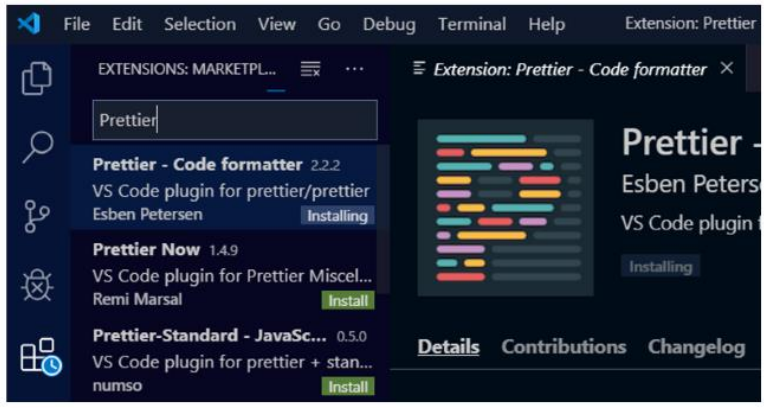
După instalarea editorului vor fi operate câteva adaptări. Astfel, vor fi instalate două extensii, importante pentru creșterea productivității realizării aplicațiilor:

a. Simple React Snippets


File Edit Selection View Go Debug Terminal
Extensions (Ctrl+Shift+X)


File Edit Selection View Go Debug Terminal Help
EXTENSIONS: MARKETPL...
Simple React Snippets
Simple React Snippets 1.2.2
Dead simple React snippets you w...
Burke Holland
Install
ES7 React/Redux/GraphQ... 2.4.3
Simple extensions for React, Redu...
dsznajder
Install
snippets 0.0.1
ReactJS snippets created by GoBy...
GoBystrok ReactJS
Install

Prettier


File Edit Selection View Go Debug Terminal Help
Extension: Prettier
EXTENSIONS: MARKETPL...
Extension: Prettier - Code formatter
Prettier
Prettier - Code formatter 2.2.2
VS Code plugin for prettier/prettier
Esben Petersen
Installing
Prettier Now 1.4.9
VS Code plugin for Prettier Miscel...
Remi Marsal
Install
Prettier-Standard - JavaSc... 0.5.0
VS Code plugin for prettier + stan...
numso
Install
Prettier -
Esben Peters
VS Code plugin
Installing
Details
Contributions
Changelog

După instalare, extensia trebuie activată. Pentru aceasta se va selecta butonul *Manage / Settings* și, în caseta de căutare se va tasta *default formatter* (fig. 1.2). Apoi, din lista derulantă afișată, se va selecta *Prettier*.

O ultimă adaptare a aplicației *VS Code* ar putea fi o setare care va impune reformatarea codului (folosind *Prettier*) la fiecare salvare a acestuia. Pentru aceasta va fi selectat din nou butonul *Manage / Settings* și în caseta de căutare va fi căutată și activată opțiunea "*Format On Save*" (fig. 1.3).

Desigur, mai pot fi realizate și alte optimizări.

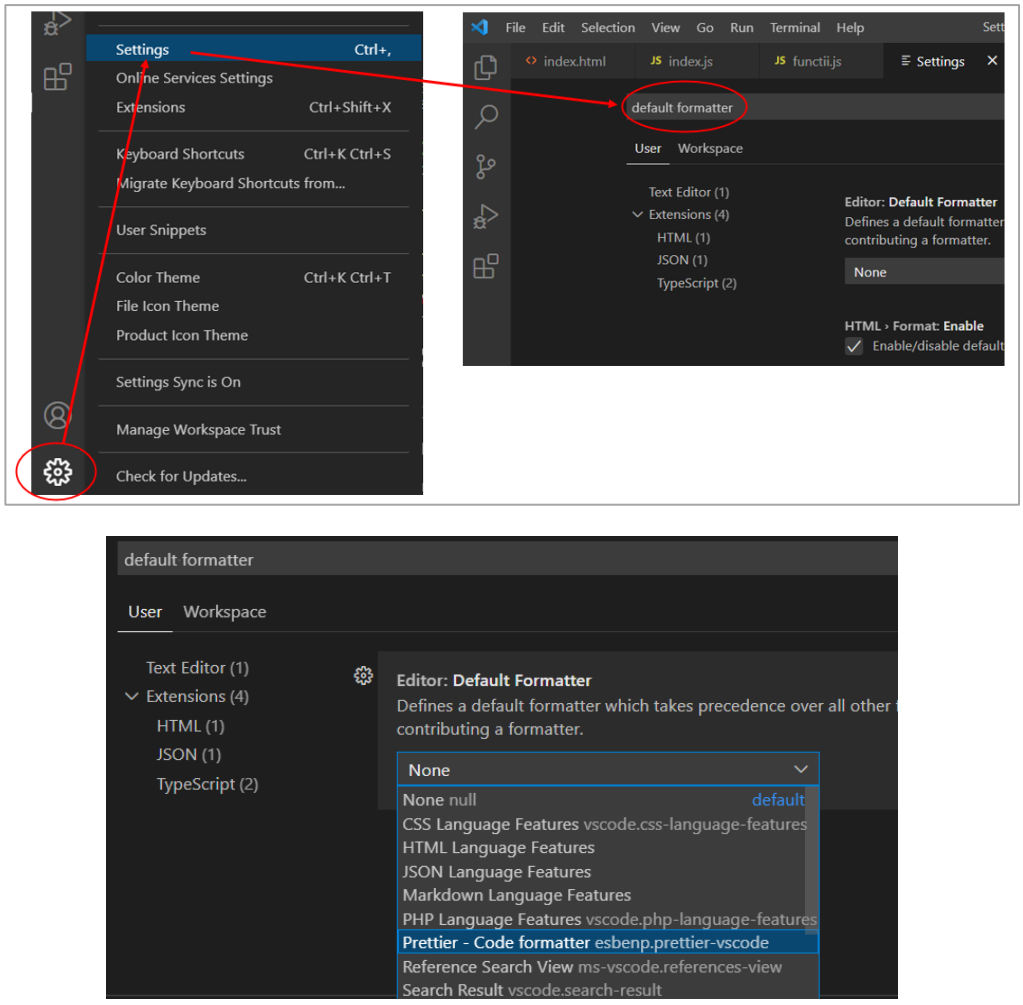


Figura 1.2. Configurare Prettier

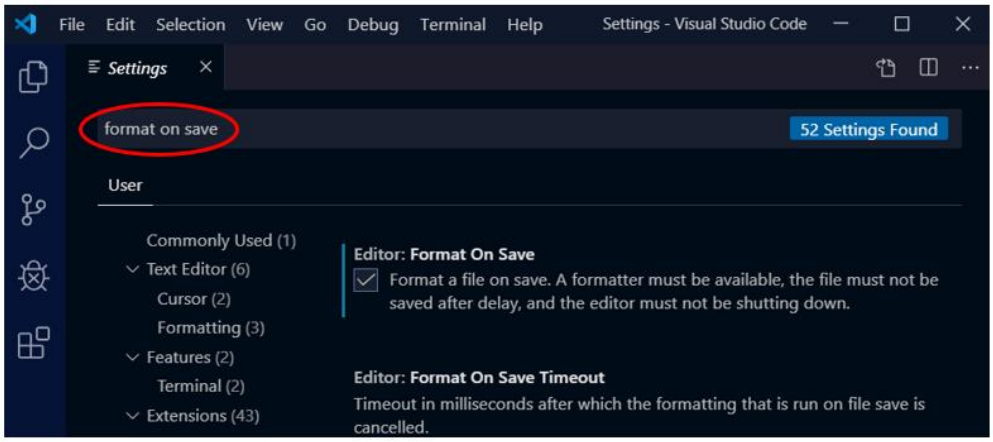


Figura 1.3. Impunerea opțiunii "*Format On Save*"

CAPITOLUL 2

JavaScript

Scurtă prezentare

JavaScript este un limbaj de programare creat inițial pentru crearea paginilor web având un conținut dinamic. Limbajul a fost creat în cadrul companiei Netscape, în 1995.

Codul JavaScript poate fi inserat direct în paginile web sau plasat în fișiere separate. JavaScript este un *limbaj interpretat*. Browserele moderne conțin însă compilatoare *JIT* (*Just-In-Time*) care compilează codul JavaScript, transformându-l în cod mașină. Un bun exemplu este V8, un compilator de JavaScript integrat în browserele *Google Chrome* și *Opera*.

Observație: O prezentare corectă a acestui limbaj depășește, desigur, spațiul care i-a fost alocat în această lucrare. Dar există resurse remarcabile, accesibile gratuit în Internet, care pot completa cu succes zonele incomplet acoperite.

De exemplu *https://eloquentjavascript.net/Eloquent_JavaScript_small.pdf* sau site-ul *https://javascript.info/*.

Ce face codul JavaScript

Codul JavaScript creează un conținut dinamic, codificat în HTML și îl inserează undeva, în pagina web. Conținutul poate fi static, poate proveni dintr-o sursă externă (o bază de date, un serviciu web) sau poate fi generat pe baza unor informații introduse de operator folosind elemente specifice formularelor.

Exemplu fundamental:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
</head>
<body>
   <h1>Tabla înmulțirii cu </h1>
   <div id="tabla"></div>

   <script>
     let n = prompt("Introduceți n: ");
     if (n != null) {
        let p = "";

        for (let i = 1; i <= 10; i++) {
           p += `${n} x ${i} = ${n * i}<br>`
        }

        document.querySelector("h1").innerHTML += n;
        document.querySelector("#tabla").innerHTML =
 `<p>${p}</p>`;
     }
   </script>

   </body>
</html>
```

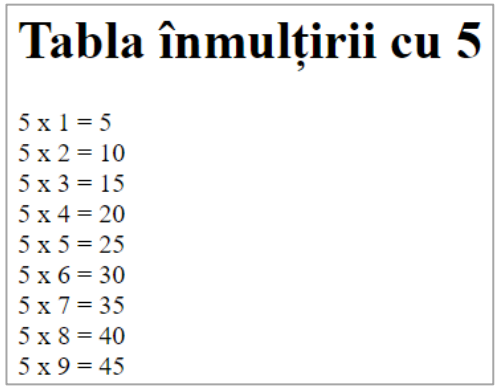
Se va adăuga *n*

 vid. Se va insera un paragraf.

Generare paragraf.

Construire paragraf și inserare.

Rezultat:



Observație: Codul JavaScript, conținut într-un element `<script>` plasat înainte de `</body>`, conține declarații de variabile (`n`, `p`), atribuiri și o instrucțiune `for`, în mare parte similar codului scris în oricare dintre limbajele procedurale consacrate (`C`, `C++`, `Java` etc.). În partea finală însă este apelată repetat funcția `querySelector()`. Această funcție returnează un `obiect JavaScript`. El a fost creat de browser în momentul citirii și prelucrării fișierului `index.html` și poate fi preluat într-o `variabilă JavaScript` folosind un set de funcții dedicate, una dintre ele fiind `querySelector()`. Odată preluat într-o variabilă, obiectul JavaScript poate fi editat iar browserul va reflecta modificările operate.

Observație: Pentru fiecare element HTML din `index.html`, browserul va crea un obiect. Obiectele JavaScript astfel create formează o structură arborescentă denumită `Document Object Model` (prescurtat `DOM`). Fiecărui obiect din `DOM` i se adaugă în momentul creării o serie de proprietăți care derivă din codul HTML folosit la descrierea sa. Valorile unora dintre aceste proprietăți (`class`, `id`, tipul elementului ș.a.) pot fi ulterior utilizate ca și argumente ale funcției `querySelector()`. În codul HTML din `index.html` de exemplu, elementului `<div>` destinat afișării tablei înmulțirii i s-a atașat un atribut `id` având valoarea `tabla`:

```
    <div id="tabla"></div>
    ...
    document.querySelector("#tabla").innerHTML = `<p>${p}</p>`
;
    ...
```

Tipuri de date

Tipuri primitive

- `number`: o valoare numerică, întreagă sau reală;
- `boolean`: valorile posibile pentru acest tip sunt `true` sau `false`;
- `null`: acest tip are o singură valoare: `null`;
- `undefined`: acest tip are tot o singură valoare (`undefined`) și este atribuită automat unei variabile care este declarată dar nu a primit o valoare.

- *string*: un șir de caractere: *"Ionescu Paul"* sau *'Avram Laura'*;

Pe lângă variantele prezentate, respectiv încadrarea șirurilor de caractere între caractere " (ghilimele) sau ' (apostrof), se poate folosi și caracterul " ` " (apostrof invers). Acesta permite crearea unor construcții sintactice denumite *prototipuri*. Un prototip este un șir de caractere care poate conține expresii JavaScript. Pentru delimitarea expresiilor JavaScript din prototipuri se folosește sintaxa: *${ expresie }*. Exemplu:

```
   const nume = "Marcel Nicoară";
   const mesajCompl = `<p>Acest mesaj se adresează utilizato
rului
${nume}. Dacă nu poate fi contactat prin e-mail, mesajul va f
i
transmis telefonic.</p>`;

   // Și utilizez sirul astfel obtinut
   document.querySelector("#mesaj").innerHTML = mesajCompl;
```

Așa cum se observă din exemplul dat, aceeași notație permite scrierea șirurilor de caractere pe mai multe rânduri:

```
     let sirCaractere = `Lorem ipsum dolor sit amet, consectet
ur adipisicing elit,
         sed do eiusmod tempor incididunt ut labore et dolore
magna aliqua.
         Ut enim ad minim veniam, quis nostrud exercitation ul
lamco laboris
         nisi ut aliquip ex ea commodo consequat.`;
```

Tipuri referite

- *object*: (obiect) utilizat pentru entități având un număr de *proprietăți*. Pentru definirea unui obiect se folosesc acolade.

Proprietățile unui obiect sunt perechi de forma *nume: valoare* și sunt despărțite prin virgule.

Exemplu:

```
   const hamilton = {pilot: "Lewis Hamilton", varsta: 35, echi
pa: "Mercedes"};
```

sau, mai clar:

```
const hamilton = {
      pilot: "Lewis Hamilton",
      varsta: 35,
      echipa: "Mercedes"
};
```

- *function*, utilizat pentru definirea funcțiilor. În JavaScript *funcțiile sunt obiecte*, deci pot fi folosite oriunde poate fi folosit un obiect: în membrul drept al unei atribuiri, ca parametru formal la definirea altei funcții sau poate fi un element într-un șir de valori (*array*).

Exemplu:

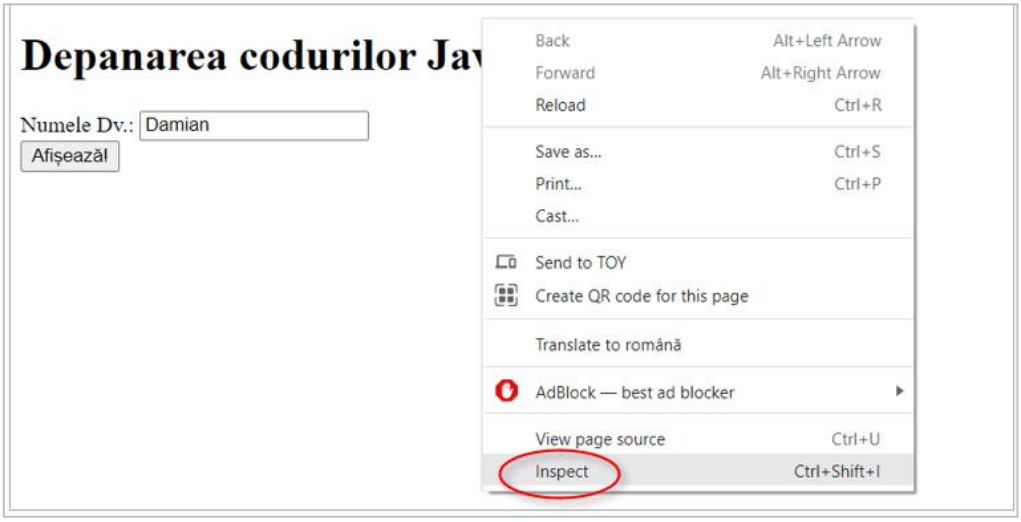
```
const aduna = function(a, b) { return a + b;};
```

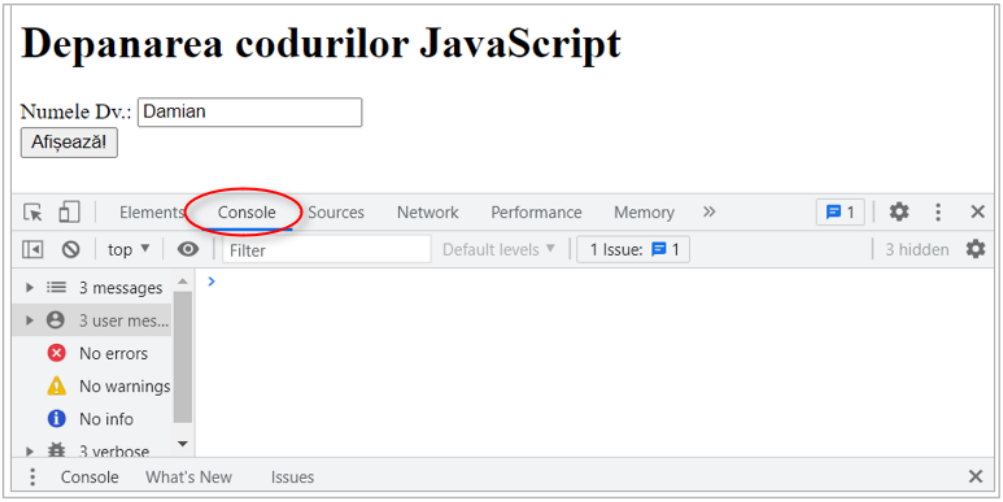
- *array*, pentru definirea unor colecții de valori de tipuri diferite, indexate.

Exemplu:

```
  let nume = ["Ioan", "Cristina"];
  nume[nume.length] = "Mircea"; // nume.length=2 (lungimea sir
ului)
  nume[nume.length] = "Maria";   //  nume.length=3
  console.log(`Al treilea element este ${nume[2]}.`); //  "Mir
cea"
```

Observație: Browserele dispun de o facilitate importantă legată de depanarea secvenţelor de cod JavaScript, respectiv o *consolă* în care se pot afișa și chiar modifica valori ale variabilelor aplicației. În *Google Chrome* afișarea consolei se realizează prin apăsarea tastei funcționale *F12* sau, în meniul contextual afișat la selectarea cu butonul drept al mausului a unui element din pagină, se selectează *Inspect* și apoi *Console*:





Scrierea pe consolă din JavaScript se realizează folosind funcţia *console.log()*. Din modul de scriere a apelului funcţiei *log()* se poate deduce că JavaScript este un limbaj orientat pe obiecte. Folosind terminologia consacrată, *log()* este o *metodă statică* a clasei *console*.

Metoda *log()* acceptă unul sau mai mulţi parametri. Valorile parametrilor vor fi afişate pe aceeaşi linie, deci dacă afişarea trebuie realizată pe linii distincte, metoda *log()* va trebui apelată în mod repetat.

Observaţie: La deschidere consola poate conţine o serie de mesaje. Golirea acesteia se realizează selectând în meniul contextual opţiunea `Clear console` (clic cu butonul drept al mausului în panoul conținând consola).

Variabile

Declararea variabilelor

În orice limbaj de programare o variabilă este un nume dat unei celule din memoria calculatorului. Dimensiunea celulei depinde de natura informației memorate în ea.

În JavaScript ***celula din memorie asociată unei variabile conține o adresă***. În calculatoarele actuale, ale căror sisteme de operare sunt pe 64 de biți, adresele sunt memorate în celule având dimensiunea de 64 de biți.

Adresele conținute în variabile indică unde se află în memorie informația efectivă, aceasta putând fi o valoare numerică, un șir de caractere, o variabilă logică sau ... o funcție!.

Tipul unei variabile se stabilește dinamic, în momentul în care în celula din memorie alocată acesteia se pune adresa zonei care conține informația efectivă.

Pentru a declara o variabilă se folosesc cuvintele rezervate `var`, `let` sau `const`.

În continuare vor fi folosite doar `let` și `const`.

Variabilele declarate folosind `let` și `const` sunt utilizabile în blocul de cod JavaScript în care s-a realizat declararea (sau în blocuri interioare acestuia) și numai după linia care conține declarația.

Diferența dintre `let` și `const` constă în faptul că o variabilă declarată folosind `const` primește o valoare în momentul declarării și nu va putea primi ulterior o altă valoare printr-o nouă atribuire.

Observație: În JavaScript, odată memorată valoarea unei variabile, ea nu poate fi modificată. Se spune că este invariabilă (eng. `immutable`). Aceasta este o consecință a faptului că în JavaScript tipul unei variabile se stabilește dinamic, în momentul în care variabila primește o valoare. Deci modificarea valorii unei variabile nu înseamnă modificarea valorii din memorie asociată

acesteia ci atribuirea unei noi adrese. Această nouă adresă este adresa zonei din memorie în care s-a memorat noua valoare.

Exemple:

```
  let an;
  let numePrenume;
  numePrenume = "Donald Duck";
  an = 1314;
  let zi = 14, luna = 7;
  const valPi = 3.1415926; // const! Nu va putea fi modificata
!
  const ROSU = "#FF0000";
  const NUME = "Ionescu Paul";
```

Observații:

- Declararea unei variabile folosind *let* poate fi realizată împreună cu inițializarea ei;
- Tentativa de modificare a unei constante (*const*) provoacă afișarea unui mesaj de eroare;

Numele dat unei variabile este corect dacă dacă sunt folosite doar litere, caractere '*$*', '*_*' și cifre. Denumirea nu poate începe cu o cifră.

Limbajul face distincție între majuscule și literele mici (este *case sensitive*).

Instrucțiuni

Instrucțiuni condiționale

Instrucțiunile condiționale permit executarea condiționată a unor secvențe de cod.

În grupul instrucțiunilor din această familie sunt *if*, *switch* și *operatorul ternar*.

▪ if

În toate limbajele instrucțiunea *if* este folosită pentru a verifica dacă o condiție este adevărată. Dacă da, va urma execuția unui bloc de instrucțiuni.

Sintaxa instrucțiunii *if* este similară celei cunoscute de la *C* (*C++*):

```
if (conditie) {
   // bloc de instructiuni pentru conditie == true
}
```

Blocul de instrucțiuni va fi pus între acolade (*{}*) și va fi executat doar dacă valoarea de adevăr a condiției este *true*.

Limbajul JavaScript fiind derivat din *C++*, la scrierea expresiilor logice se vor utiliza următorii operatori relaționali:

Operator	Semnificație	Exemplu
==	*Egalitate (numere)*	x == y
===	*Egalitate valori şi tipuri de variabile*	x === y
>	*Operatorul "mai mare"*	
<	*Operatorul "mai mic"*	
>=	*Operatorul "mai mare sau egal"*	
<=	*Operatorul "mai mic sau egal"*	
!=	*Operatorul "diferit"*	
!==	*Operatorul "diferit"*	
||	*Operatorul logic "sau"*	
&&	*Operatorul logic "şi"*	
!	*Operatorul logic de negare*	

Observație: Pentru testarea egalităţii a două valori se vor folosi sistematic operatorii '*===*' și '*!==*'. Acești operatori verifică atât identitatea valorilor cât şi a tipurilor valorilor comparate.

Exemplu:

```
let nume = "Marian"
let varsta = 23
if (varsta >= 18) {
   console.log(`${nume} este major.`)
}
```

Observație: În secvența de cod srisă au fost omise (intenționat!) caracterele ';' de la sfârșitul fiecărei instrucțiuni. În JavaScript, ca și în câteva alte limbaje (Python de exemplu), utilizarea caracterului ';' poate fi omisă deoarece acest limbaj integrează un mecanism de adăugare automată a acestuia (*Automatic Semicolon Insertion*, prescurtat *ASI*).

▪ if - else

Această variantă de scriere a instrucțiunii *if* permite introducerea unui bloc de instrucțiuni alternativ, care va fi executat dacă nu este îndeplinită condiția.

```
let nume = "Marian"
let varsta = 13
if (varsta >= 18) {
   console.log(`${nume} este major.`)
} else {
   console.log(`${nume} este minor.`)
}
```

▪ if – else if

Există posibilitatea înlănțuirii mai multor instrucțiuni *if – else*, dacă trebuie verificate mai multe condiții.

Sintaxa instrucțiunii:

```
if (conditie) {
   // bloc de instructiuni
} else if (conditie) {
   // bloc de instructiuni
} else {
   // bloc de instructiuni
}
```

Exemplu:

```
   let vr = prompt('Cum e vremea (insorita, ploioasa, innorata
)?')
   if (vr === 'ploioasa') {
      console.log(' ☁ Luați-vă pelerina!')
   } else if (vr === 'innorata') {
      console.log(' ☔ Ar putea ploua. Luați-vă o umbrelă!')
   } else if (vr === 'insorita') {
      console.log(' ☼ Puteți ieși fara grijă!')
   } else {
      console.log('Răspuns incorect!')
   }
```

▪ switch

Instrucțiunea *switch* este o soluție alternativă la înlănțuirea instrucțiunilor *if – else*.

```
   let nr = prompt('Introduceti un numar intreg')
   switch (nr) {
   case 1:
       // bloc de instr. 1
       break
   case 2:
       // bloc de instr. 2
       break
   case 3:
       // bloc de instr. 3
   default:
       // bloc de instr. 4
   }
```

sau:

```
   let vr = prompt('Cum e vremea (insorita, ploioasa, innorata
)?')
   switch (vr) {
   case 'ploioasa':
      console.log(' ☁ Luați-vă pelerina!')
      break
   case 'innorata':
      console.log(' ☔ Ar putea ploua. Luați-vă o umbrelă!')
      break
   case 'insorita':
      console.log(' ☼ Puteți ieși fara grijă!')
```

```
    break
default:
    console.log('Răspuns incorect!')
}
```

Observații:

- Instrucțiunea `break` provoacă ieșirea imediată din `switch`. În lipsa ei, dacă o condiție ar fi îndeplinită, executarea blocului introdus prin acea condiție ar fi urmată de executarea blocului următor.
- Blocul introdus prin `default` este opțional și va fi executat dacă niciuna dintre condiții nu este îndeplinită.

Operatorul ternar

Acest operator va fi folosit în multe cazuri în locul instrucțiunii `if – else`.

Exemplu:

```
let nr = prompt('Introduceți varsta: ')
let minmaj = (nr >= 18) ? 'major!' : 'minor!';
conslole.log(`Din vârstă rezultă că sunteți ${minmaj}.`);
```

Instrucțiuni de ciclare

În aplicațiile JavaScript pot fi folosite următoarele instrucțiuni de ciclare:

- `for`
- `while`
- `do while`
- `for of`
- `for in`

Un ciclu se întrerupe atunci când condiția de reluare devine `false`. Ciclul poate fi de asemenea întrerupt prin executarea unei instrucțiuni `break` sau se poate trece forțat la iterația următoare prin execuția unei instrucțiuni `continue`.

▪ for

Instrucțiunea *for* se folosește atunci când se cunoaște numărul de iterații. Exemplu:

```
let suma = 0
for (let i = 1; i <= 50; i++) {
  suma += i
}
console.log(suma)
```

Secvența de cod scrisă calculează suma numerelor de la 1 la 50. Pentru a însuma doar numerele impare se poate scrie astfel:

```
let suma = 0
for (let i = 1; i <= 50; i += 2) {
  suma += i
}
console.log(suma)
```

sau o altă variantă:

```
let suma = 0
for (let i = 1; i <= 50; i++) {
  if (i % 2 == 1) {
    suma += i
  }
}
console.log(suma)
```

▪ while

Instrucțiunea *while* se folosește când nu se cunoaște de la început numărul de iterații care trebuie executate. Exemplu:

```
let m = prompt('Introduceți mărimea Dv. la pantofi: ')
let marimi = [39, 40, 43, 44, 45, 46];
let i = 0;
while (m != marimi[i]) {
   i++
   if(i > marimi.length) {
      break
   }
} // Acum verific daca s-a gasit marimea
if(i <= marimi.length) {
```

```
      console.log("Marimea Dv. este " + m)
   } else {
      console.log("N-am gasit marimea " + m) }
```

▪ do while

Instrucțiunea *do while* permite rularea unui bloc de instrucțiuni cel puțin odată și este recomandată atunci când în condiția de reluare se folosește o variabilă care primește valoare în corpul ciclului.

Exemplu:

```
let v
do { v = prompt('Introduceți un numar intreg < 100: ')
   contor++
} while (v != nrSecret)
```

▪ for of

Ciclul *for of* este foarte util în cazul parcurgerii șirurilor de valori dacă nu ne interesează indicele elementului curent. În astfel de situații *for of* este preferabil unui *for* obișnuit.

Exemplu de utilizare:

```
const valori = [43, 21, 73, -44, -7]
for (const numar of valori) {
   console.log(numar)
}
```

sau:

```
const state = ['Serbia', 'Finlanda', 'Franta', 'Spania',
               'Cehia', 'Danemarca']
for (const tara of state) {
   console.log(tara.toUpperCase())
}
```

▪ for in

Instrucțiunea *for in* este utilizată pentru parcurgerea proprietăților unui obiect JavaScript. Exemplu:

```
   const emmaRaducanu = {
      nume: 'Răducanu',
      prenume: 'Emma',
      varsta: 18,
      nationalitate: 'engleză',
      competente: ['tenis de câmp', 'modeling']
   }
   for (const np in emmaRaducanu) {
      console.log(np, emmaRaducanu[np])
   }
```

Întreruperea unui ciclu

▪ break

Instrucțiunea *break* este folosită pentru a întrerupe un ciclu. Exemplu:

```
   let nrDiv = 0, i
   for (i = 1; i < 1000; i++) {
      if (i % 7 == 0) {
         nrDiv++
         if(nrDiv == 7) {
            break
         }
      }
   }
   console.log("Am gasit primele 7 numere divizibile cu 7. Ult
imul gasit este " + i)
```

Ciclul controlat de variabila *i* se oprește când contorul *nrDiv* ajunge la valoarea 7.

▪ continue

Instrucțiunea *continue* se folosește pentru a sări peste anumite iterații. Exemplu:

```
   for (let i = 0; i <= 5; i++) {
      if (i == 3) {
         continue
      }
      console.log(i) // 0 1 2 4 5
   }
```

Șiruri de valori

Memorarea datelor unei aplicații se bazează frecvent pe utilizarea șirurilor de valori.

Declararea unui șir de valori a fost deja prezentată:

```
  let sir2 = [];  //  Declar un sir vid
  const materii = ["Romana", "Matematica", "Fizica", "Chimie"]
;
```

Elementele unui șir sunt accesate folosind indici, astfel:

```
  console.log(materii[0]);
```

Observație: Deși șirul *materii* a fost declarat *const*, aceasta nu împiedică modificarea valorilor elementelor din șir:

```
  materii[2] = "Limba engleza";
```

Pentru a obține lungimea unui șir de valori se folosește proprietatea *length*:

```
  console.log( materii.length ); // 4
```

Proprietatea *length* este utilizată și pentru a adăuga la sfârșitul șirului un element suplimentar:

```
  let limbaje = ["Lisp", "C++", "Java", "C#"];
  limbaje[limbaje.length] = "Fortran";
```

Poziția având indicele *limbaje.length* este întotdeauna ultima.

Fiecare element dintr-un șir de valori conține o adresă din memorie, respectiv adresa zonei din memorie unde este stocată informația propriuzisă. Din acest motiv, în JavaScript șirurile pot conține valori de tipuri diferite:

```
  const mircea = {nume: 'Popa', prenume: 'Mircea', ocupatie: '
Avocat'};
  const fun = (nume) => {
    alert(`Salut, ${nume}!`);
  }
  const sirMixt = [12, "Florin", 45.12, true, mircea, fun, ];
```

Observații:

- Dacă în codul scris în continuare apare: `sirMixt[5]();`, acesta va fi un apel de funcție valabil și se va afișa o fereastră `alert`.
- După ultimul element apare o virgulă în plus, dar interpretorul de cod nu o va trata ca eroare. Aceasta face ca adăugarea dinamică a elementelor în șir să fie mult simplificată. Practic fiecare element va fi adăugat împreună cu virgula de după el fără a se ține seama de faptul că este ultimul element sau mai urmează să fie adăugate și altele. Aceeași regulă se aplică și în cazul declarării obiectelor. Exemplu:

```
const mircea = {nume: 'Popa', prenume: 'Mircea',
                ocupatie: 'Avocat',};
```

Operatorul [...] (eng. *spread operator*)

Acest operator poate simplifica scrierea unor operații cu șiruri, astfel:

1. Adăugarea unor elemente suplimentare:

```
let materii = ["Romana", "Chimie", "Fizica",];
let materiiNou = [...materii, 'Ed. fizica', 'Latina'];
 //  ["Romana", "Chimie", "Fizica", "Ed. fizica", "Latina"];
```

Deci interpretorul JavaScript va înlocui expresia `...materii` cu toate elementele din șirul `materii`.

2. Concatenarea șirurilor:

```
const sir1 = [1, 2, 3];
const sir2 = [7, 8];
let sirConcat = [...sir1, ...sir2];
console.log(sirConcat);  //  [1, 2, 3, 7, 8]
```

3. Inserarea unor elemente la începutul sau la sfârșitul șirului:

```
const sir = ['r', 'a', 5];
const sir1 = [1, 3, ...sir]; //  [1, 3, 'r', 'a', 5]
const sir2 = [...sir, Q, T];  // ['r', 'a', 5, Q, T]
```

4. Clonarea unui șir de valori

```
let clonaMaterii = [...materii];
```

Destructurarea șirurilor de valori

Destructurarea unui șir permite preluarea valorilor din șir în variabile separate. Exemplu:

```
let premianti = ['Maria AVRAM', 'Mircea POP', 'Ioana STAN'];
const [loc1, loc2, loc3] = premianti;
console.log(loc2);  //  Mircea POPA
```

Prin destructurarea șirului de valori *premianti* s-au creat constantele *loc1*, *loc2* și *loc3*. Constanta *loc2* va primi valoarea *Mircea POP*, al doilea element din șirul *premianti*.

O astfel de construcție sintactică este utilizată de altfel frecvent în componentele funcționale din *React*:

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

Pentru ca expresia astfel scrisă să fie corectă, funcția *useState()* trebuie să returneze un șir conținând două valori. Și pentru că în *React* se utilizează limbajul JavaScript, a doua valoare, *setCount*, este denumirea unei funcții!

Destructurarea obiectelor

În unele dintre secvențele de cod care vor fi scrise în continuare se va practica și *destructurarea obiectelor*. Aceasta are ca scop crearea unui set de variabile care preiau valorile proprietăților unui obiect. Exemplu:

```
const emmaRaducanu = {
   nume: 'Răducanu',
   prenume: 'Emma',
   varsta: 18,
   nationalitate: 'engleză',
   competente: ['tenis de câmp', 'modeling']
}
const {prenume , nume, varsta} = emaRaducanu
```

Spre deosebire de destructurarea șirurilor de valori, în cazul destructurării obiectelor denumirile variabilelor create trebuie să coincidă cu denumirile câmpurilor aparținând obiectului ale căror valori le preiau. Nu contează însă

ordinea în care sunt scrise variabilele și nici câte variabile sunt astfel create, în exemplul dat preluându-se doar valorile a trei câmpuri: *nume*, *prenume* și *varsta*.

JSON

JSON (*JavaScript Object Notation*) este un standard de reprezentare a informațiilor. Fișierele care conțin informații codificate în acest format sunt foarte mult utilizate pentru transferul datelor între aplicații.

Într-un fișier în format *JSON* datele sunt memorate sub forma perechilor *cheie - valoare*, soluție similară celei utilizate pentru codificarea informațiilor în obiectele JavaScript în notație literală. De altfel între cele două moduri de codificare diferențele sunt minimale:

```
  //  JavaScript
  let maria1 = { nume: 'Popa', prenume: 'Maria', varsta: 23 };

  //  JSON
  let maria2 = '{ "nume": "Popa", "prenume": "Maria", "varsta"
: 23 }';
```

Practic singura diferență constă în încadrarea obligatorie a denumirilor câmpurilor și a șirurilor de caractere între caractere `"` (ghilimele) și transformarea ansamblului într-un șir de caractere prin plasarea între caractere `'` (apostrof).

JSON este un format pentru transferul informațiilor. În toate limbajele majore există funcții specializate care permit codificarea în și din JSON.

În JavaScript, de exemplu, codificarea în și din JSON a obiectelor JavaScript se realizează prin utilizarea funcțiilor *JSON.stringify()* respectiv *JSON.parse()*.

Exemplu:

```
   const hamilton = {
         pilot: "Lewis Hamilton",
         varsta: 35,
         echipa: "Mercedes"
   };
   const sirHamilton = JSON.stringify(hamilton)
```

```
   // Rezultat  '{"pilot": "Lewis Hamilton", "varsta": 35, "ec
hipa": "Mercedes"}'
```

Observație: În JSON valorile numerice nu sunt scrise între ghilimele.

Funcții JavaScript

În orice limbaj de programare, o funcție este o secvență de cod scrisă într-un anumit mod, reutilizabil și destinat realizării unei acțiuni precizate. Prin funcții se realizează modularizarea codului și ceară premisa reutilizării acestuia.

Funcțiile pot aparține limbajului (funcții predefinite) sau pot fi declarate de programator.

În *JavaScript ES5* se puteau declara funcții în două moduri:

```
function adun(a, b) {
   const s = a + b;
   return s;
}
```

sau atribuind unei variabile, ca valoare, o funcție anonimă:

```
const adun = function (a, b) {
   const s = a + b;
   return s;
}
```

Observație: A doua modalitate de declarare a unei funcții derivă din faptul că, în JavaScript, o funcție este un obiect, deci poate fi utilizată în orice construcție sintactică în care poate fi utilizat un obiect.

O modalitate și mai simplă de declarare a unei funcții anonime, introdusă în JavaScript începând cu versiunea *ES6*, este cea bazată pe folosirea operatorului "*=>*" (denumite și funcții *arrow*):

```
const adunare = (a, b, c) => {
  const s = a + b + c;
  return s;
}
```

Se poate intui că *(a, b, c)* reprezintă lista parametrilor formali folosiți la declararea funcției.

O funcție simplă, care nu face altceva decât să returneze o valoare calculabilă imediat (folosind parametrii formali, desigur), poate fi definită și mai simplu, instrucțiunea *return* putând fi omisă în acest caz.

```
  const adunare = (a, b, c) => a+b+c;
```

Dacă funcția are un singur parametru formal, încadrarea acestuia între paranteze este facultativă:

```
const calcul = a => {
   if(a < 0) {
     return -a;
   } else {
     return a;
   }
};
```

Observație: Exemplul precedent poate fi scris și mai compact folosind *operatorul ternar*:

```
  const calcul = a => (a<0) ? -a : a;
```

Dacă funcția anonimă nu are niciun parametru formal, parantezele trebuie puse:

```
  const parola = () => `Parola propusă nu îndeplinește condiți
ile impuse!`;
  alert(parola());
```

În continuare acest mod de declarare a funcțiilor va fi utilizat sistematic.

Pentru mai multe detalii se poate accesa adresa https://www.w3schools.com/js/js_arrow_function.asp.

Vizibilitatea variabilelor

O variabilă declarată în interiorul unei funcții folosind *let* sau *const*, este o *variabilă locală*. Ea poate fi utilizată doar în interiorul funcției și numai după linia în care a fost declarată.

Exemplu:

```
  const factorial = (n) => {
    let fct = 1;   //  fct este variabila locala
    if(n > 1) {
      for (let i = 2; i <= n; i++) {
        fct *= i;
      }
    return fct;
}
```

O variabilă declarată în afara unei funcții, înaintea locului în care funcția este apelată, folosind *let* sau *const*, poate fi utilizată în interiorul funcției, având regim de *variabilă globală*.

Exemplu:

```
   const mesaj1 = "Parolă prea simplă!";
   const mesaj2 = "Parolă acceptată!";

   const afisez = (parola) => {
      if(parola.length >= 8) {
        alert(mesaj2);
      } else {
        alert(mesaj1);
      }
   };
```

Funcții în funcții

Așa cum s-a afirmat deja, în JavaScript o funcție este un obiect. Ca urmare, în interiorul unei funcții poate fi inclusă o altă funcție. Această posibilitate de scriere va fi exploatată sistematic în capitolul următor, destinat creării componentelor *React*.

Exemplu:

```
  <script>
    const validez = (parola) => {
      const mesaj1 = "Parolă prea simplă!";
      const mesaj2 = "Parolă acceptată!";

      //  Declar o functie in interiorul functiei validez()
      const verificParola = (cod) => {
         if(cod.length >= 8) {
           alert(mesaj2);
```

```
        } else {
          alert(mesaj1);
        }
      };
      verificParola(parola);
    };

    validez("2120Toamna");
  </script>
```

Funcții JavaScript de manipulare a șirurilor de valori

Programarea în React este bazată în mare parte pe programarea funcțională. În cele ce urmează va fi aplicată această paradigmă fundamentală a programării pentru prelucrarea șirurilor de valori.

Prelucrările care vor fi realizate se vor baza pe un ansamblu de funcții dedicate: *map()*, *filter()*, *find()* și *reduce()*.

map()

Funcția *map()* permite crearea unui șir nou de valori pornind de la un șir dat. Fiecare valoare din șirul nou este obținută prin modificarea elementului corespunzător din șirul inițial. Exemplu:

```
    const categorii = ["pantofi", "sandale", "ghete", "cizme",
 "altele"];
    const categoriiMaj = categorii.map(item => {
       return item.toUpperCase();
    });
```

sau, mai simplu, eliminând instrucțiunea *return*:

```
    const categorii = ["pantofi", "sandale", "ghete", "cizme",
 "altele"];
    const categoriiMaj = categorii.map(item => item.toUpperCas
e());
```

filter()

Funcția *filter()* permite crearea unui șir nou de valori conținând doar elementele din șirul inițial care îndeplinesc o condiție dată. Condiția este exprimată printr-o funcție. Exemplu:

```
const sirTutoriale = [
  {subiect: "Programarea aplicatiilor web", sursa: "UTCN" },
  {subiect: "Aplicații mobile", sursa: "Google Co."},
  {subiect: "Programarea microprocesoarelor", sursa: "UTCN"
  }
];

const tUTCN = sirTutoriale.filter(item => {
  if (item.sursa === "UTCN") {
    return true;
  }
  return false;
});
```

sau, mai simplu:

```
const sirTutoriale = [
  {subiect: "Programarea aplicatiilor web", sursa: "UTCN" },
  {subiect: "Aplicații mobile", sursa: "Google Co."},
  {subiect: "Programarea microprocesoarelor", sursa: "UTCN"
  }
];

const tUTCN = sirTutoriale.filter(item =>
        (item.sursa === "UTCN") ? true : false);
```

find()

Funcția *find()* permite găsirea unui element într-un șir de valori dat. Exemplu:

```
const categorii = [
       {id: 1, nume: "pantofi"},
       {id: 2, nume:"sandale"},
       {id: 3, nume: "ghete"},
       {id: 4, nume: "cizme"},
       {id: 5, nume: "altele"}
];
let categorie = categorii.find(item => {
```

```
    return item.nume === "sandale";
});
```

sau, mai simplu:

```
const categorii = [
       {id: 1, nume: "pantofi"},
       {id: 2, nume:"sandale"},
       {id: 3, nume: "ghete"},
       {id: 4, nume: "cizme"},
       {id: 5, nume: "altele"}
];
let categorie = categorii.find(item =>
                  item.nume === "sandale");
```

Funcția utilizată ca parametru al funcției `find()` permite să fie reținut din șirul de obiecte dat (`categorii`) primul obiect care îndeplinește condiția `item.nume === "sandale"`.

reduce()

Funcția `reduce()` permite crearea unei singure valori pornind de la un șir de valori. Exemplu:

```
const codHTML = categorii.reduce((html, item) => html +
                  `<li>${item.nume}</li>`, "");
```

Funcția `reduce()` are două argumente. Analizând exemplul dat se observă că primul argument este o funcție anonimă:

```
(html, item) => html + `<li>${item.nume}</li>`
```

Aceasta are la rândul ei două argumente, `html` și `item`. Al doilea argument al funcției `reduce()` este valoarea inițială a variabilei `html` (în exemplul dat este un șir vid, "").

Observație: În unele cazuri, funcțiile expuse sunt apelate în cascadă. Atunci valoarea (sau șirul de valori) returnată de o funcție va fi valoarea de intrare a următoarei functii. Exemplu:

```
 const sirTutoriale = [
  {subiect: "Programarea aplicatiilor web", sursa: "UTCN" },
  {subiect: "Aplicații mobile", sursa: "Google Co."},
  {subiect: "Programarea microprocesoarelor", sursa: "UTCN"
```

```
    }
  ];

  const tUTCN =
     sirTutoriale.filter(item => item.sursa === "UTCN")
       .map(item => item.subiect.toUpperCase())
       .reduce((html, item) => html + `<li>${item}</li>`,
             "<ol>") + "</ol>";

  document.querySelector("#lista").innerHTML = tUTCN;
```

Explicații: În funcția `reduce()` variabila `html` este inițializată cu "`<ol>`". După adăugarea la acest șir a corpului listei (două elemente `<li>`), se mai adaugă șirul de caractere "`</ol>`" rezultând reprezentarea în cod HTML a unei liste ordonate. În final lista este inserată într-un `<div>` având `id="lista"`.

Document Object Model (DOM)

Aspecte generale

În momentul citirii codului HTML al unei pagini web, browserul construiește o reprezentare abstractă a acesteia. Fiecare element din pagină va fi reprezentat intern, în memoria browser-ului printr-un obiect. Obiectele astfel create vor forma o structură arborescentă, care reflectă raporturile existente între elementele paginii.

Această reprezentare poartă numele de *Document Object Model* (prescurtat *DOM*).

De exemplu pagina web având următorul conținut:

```
<!DOCTYPE html>
<html>
<head>
   <title>Un titlu</title>
</head>
<body>
   <h1>Exemplu fundamental</h1>
   <p>Pentru a înțelege mai bine structura unei pagini web, ac
cesați <a href="https://www.w3schools.com/tags/tag_html.asp">a
ceastă pagină.</a></p>
</body>
```

```
</html>
```

va avea avea reprezentare arborescentă din figura 2.1

Majoritatea acțiunilor scripturilor JavaScript presupun selectarea unui element (sau unui ansamblu de elemente) din DOM urmată de modificarea acestuia.

Selectarea se realizează prin apelul unei funcții JavaScript specializate (*document.getElementById*, *document.getElementsByClassName*, *document.getElementsByTagName* și altele) sau prin utilizarea unor atribute ale elementelor date de poziția lor în DOM (*obiect.firstElementChild*, *obiect.lastElementChild*, *obiect.ParentElement obiect.children* și altele (pentru detalii puteți accesa: *https://www.w3schools.com/jsref/dom_obj_all.asp*).

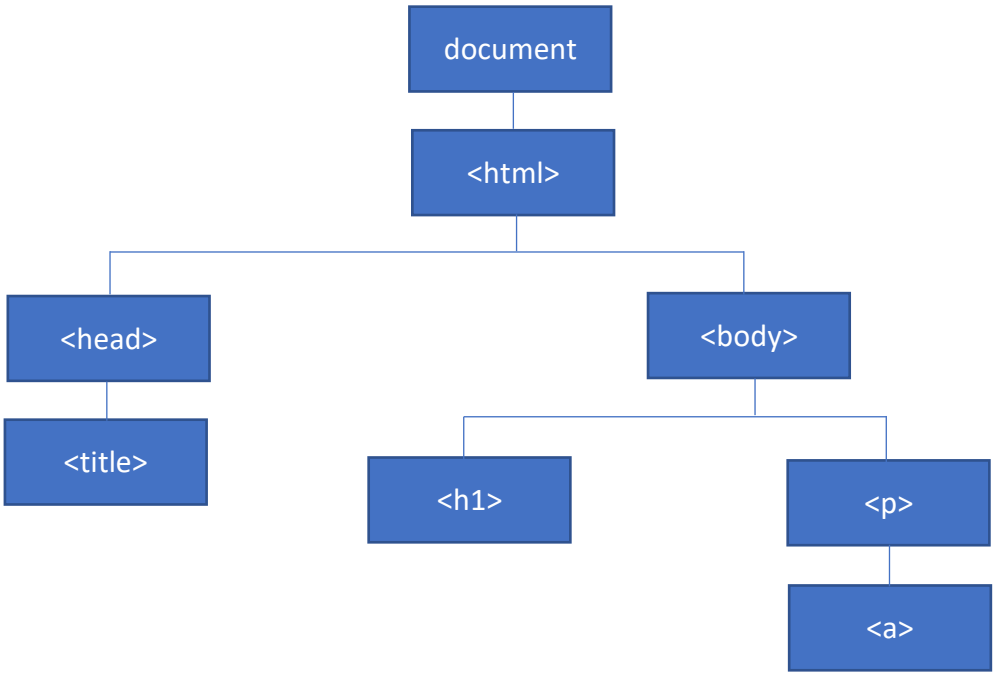


Figura 2.1 Structura ierarhică a elementelor dintr-o pagină web

În aplicațiile care vor fi realizate vor fi utilizate pentru selectarea elementelor din DOM doar două dintre funcțiile menționate: *obiect.querySelector* și *obiect.querySelectorAll*.

Prima funcție, *obiect.querySelector*, permite selectarea unui singur element, argumentul funcției indicând atributul care va fi folosit la selecție. Exemple:

```
//  Selectare <nav>
var blNav = document.querySelector("nav");
//  Selectarea elementului din <nav> având class="active"
var blocSel = blNav.querySelector(".active");
//  Selectarea elementului având id="imh"
var ank1 = document.querySelector("#imh");
```

Observații:

- *obiect.querySelector()* se apelează folosind un obiect din DOM selectat anterior sau elementul predefinit, *document*;
- Funcția va returna un singur obiect sau *null*, dacă nu există niciun obiect care îndeplinește criteriul de selecție. Dacă pentru identificarea obiectului dorit se folosește denumirea unei clase CSS sau numele unui marcaj HTML, va fi selectat primul din șirul de obiecte care respectă condiția.

Pentru a utiliza mai ușor această funcție, la scrierea codului paginii elementele care vor trebui ulterior selectate vor primi atribute *"id"*. Identificatorii atribuiți nu trebuie să apară în fișierele de stiluri ci sunt inserați în codul elementelor doar pentru a facilita selectarea acestora.

Funcția *obiect.querySelectorAll()*, permite selectarea unei mulțimi de elemente din DOM, argumentul funcției impunând criteriul de selecție. Exemple:

```
//  Selectarea tuturor elementelor <p>
var paragrafe = document.querySelectorAll("p");
//  Selectarea tuturor elementelor având class="reteta"
var retete = document.querySelectorAll(".reteta");
```

Observații:

- *querySelectorAll()* returnează un obiect de tip *NodeList* (clasă predefinită a limbajului JavaScript) conținând obiecte din

DOM sau `null` (dacă nu există obiecte care îndeplinesc criteriul de selecție).

- Pentru a ușura selectarea, la scrierea codului paginii elementele care vor trebui ulterior selectate vor avea impusă aceeași clasă. De obicei clasa adăugată pentru ușurarea selectării nu apare în fișierele de stiluri.
- După selectare, parcurgerea șirului de obiecte poate fi realizată folosind o instrucțiune `for / of` sau funcția `map()`. Dacă se folosește `map`, înainte de utilizarea acesteia obiectul de tip `NodeList` trebuie convertit în șir de obiecte (`Array`) folosind operatorul [...] (`spread operator`):

```
const retete = document.querySelectorAll(".reteta");
sirRetete = [...retete];
```

Modificarea elementelor

Există trei situaţii tipice în care limbajul JavaScript este folosit pentru a modifica elemente din DOM.

1. Generarea unui conţinut dinamic

Limbajul JavaScript permite modificarea dinamică a conținutului elementelor din DOM selectate. În exemplele de coduri JavaScript prezentate deja s-au realizat astfele de acțiuni (începând cu exemplul de generare dinamică a unui bloc care conține tabla înmulțirii cu un număr dat).

Modificarea unui element din pagina web presupune crearea noului conținut, selectarea elementului și înlocuirea conținutului anterior cu noul conținutul:

```
const cntNou = `Noul continut al elem. supus modificarii`;
document.querySelector("#idElement").innerHTML = cntNou;
```

Pentru a impune obiectului din DOM selectat un nou conținut se folosește proprietatea `innerHTML` a obiectului.

Observații:

1. În multe cazuri elementul modificat este un `<div>`, ceea ce permite adăugarea unui număr oricât de mare de elemente;

2. Noul conținut este întotdeauna un șir de caractere. Pentru a putea insera conținuturi mari, (secțiuni întregi chiar, conținând text și marcaje HTML) precum și pentru a putea insera în reprezentarea dorită valorile unor variabile sau expresii JavaScript, conținutul va fi un *prototip*, deci un șir delimitat folosind caractere " ` " (apostrof invers).

2. Modificarea dinamică a stilurilor

a. Modificarea dinamică a unei reguli de stilizare presupune selectarea elementului afectat urmată de impunerea noii reguli. Pentru a se modifica o regulă de stilizare se foloseşte sintaxa:

```
element.style.proprietate=valoare_noua;
```

Exemplu:

```
<!DOCTYPE html>
<html>
<body>
   <h1 id="titlu">Introducere în programare</h1>
   <p>Aspectul inițial al titlului a fost modificat prin execu
tarea scriptului.</p>
   <script>
      document.querySelector("#titlu").style.color = "blue";
   </script>
</body>
</html>
```

Observație: Dacă denumirea proprietății este formată din două cuvinte (de exemplu *background-color*), aceasta va trebui redenumită. Astfel caracterul "-" va fi suprimat și al doilea cuvânt va începe cu majusculă.

```
document.querySelector("#atentie").style.backgroundColor = "
red";
```

b. Modificarea dinamică a claselor CSS. Pentru a accesa clasele CSS utilizate la stilizarea unui element se va folosi proprietatea *classList*. Modificarea listei de clase CSS poate fi apoi operată apelând funcțiile *remove()* și/sau *add()*. Evident, *remove()* realizează suprimarea unei clase din *classlist* iar *add()* adaugă în *classlist* o clasă suplimentară. Exemplu:

```
    var ul = document.querySelector(".navbar-nav");

    //  Impun clasa "active"
    //  aici o suprim
    ul.children[1].classList.remove("active");
    //  Aici o adaug
    ul.children[3].classList.add("active");
```

3. Modificarea dinamică a atributelor

Modificarea valorii unui atribut presupune, desigur, selectarea elementului care trebuie modificat urmat de schimbarea valorii atributului.

Sintaxa utilizată pentru modificarea unui atribut rezultă din exemplele următoare:

```
  // Cod HTML
  ...
  <p><img src="https://picsum.photos/500/300" id="poza"></p>
  <p><a href="https://angular.io/" id="frw">Framework-ul prefe
rat pentru <em>front-end</em></a></p>

<script>
  // Cod JavaScript
  //  Modific atributul "src" al unui element <img>
  document.querySelector("#poza").src = "imagini/brasov.jpg";
   . . .
  //  Modific atributul "href" al unui element <a>
  document.querySelector("#frw").href =
      "https://getbootstrap.com";
  . . .
```

În momentul încărcării paginii, ca urmare a rulării codului JavaScript, imaginea inițială (din fișierul *.html*) va fi înlocuită cu imaginea */imagini/brasov.jpg* iar adresa accesată folosind referința din paragraful următor se modifică din *https://angular.io* în *https://getbootstrap.com*..

Rezultat:

Framework-ul preferat pentru *front-end*

https://getbootstrap.com

Observație:

1. Atributele `src` și `href` sunt probabil cele ale cărori valori sunt cel mai frecvent modificate;
2. Viteza la care se produce schimbarea este prea mare pentru a observa că la început atributele modificate în script aveau alte valori. În JavaScript există însă posibilitatea de a impune momentul în care să se execute o funcție sau intervalul de timp după care execuția acesteia să se repete. Funcțiile care trebuie utilizate sunt `window.setTimeout()` și `window.setInterval()`. Pentru detalii legate de utilizarea acestor funcții accesați *https://www.w3schools.com/js/js_timing.asp*.

Evenimente

De cele mai multe ori părți consistente ale aplicațiilor software constau în colecții de funcții care sunt executate la intervenția unui operator. Intervenția poate fi realizată folosind mausul, tastatura sau alte periferice care permit interactivitatea.

Software-ul destinat implemetării interactivității este conținut preponderent în componenta de `front-end` a unei aplicații, deci în cazul aplicațiilor web, codul necesar trebuie scris în JavaScript.

Pentru tratarea evenimentelor la care trebuie să reacționeze aplicația, elementelor HTML asupra cărora se va acționa li se asociază funcții de tratare.

Exemplu:

```
document.querySelector("#calcul").addEventListener("click", ca
lcTbl)
```

Funcțiile de tratare figurează printre parametrii unei alte funcții (*addEventListener* de exemplu) și sunt apelate automat la declanșarea unui anumit eveniment. În exemplul dat, evenimentului *click* (clic cu butonul stâng al mausului pe elementul HTML având *id="calcul"*) îi este asociată funcția *calcTbl*.

Astfel de funcții poartă numele de funcții de tip *callback*. Caracteristic pentru funcțiile de tip *callback* este faptul că ele figurează ca argumente ale altor funcții.

Pentru a asocia unui eveniment o funcție de tratare se poate folosi și o sintaxă alternativă, simplificată:

```
  document.querySelector("#calcul").onclick = calcTbl;
```

Observație: Majoritatea funcțiilor de tip *callback* conținute într-o aplicație web sunt asociate evenimentelor de tip *"clic"* cu mausul.

Principalele tipuri de evenimente

Din ansamblul de evenimente care pot fi tratate într-o aplicație web, în aplicațiile care vor fi realizate vor interveni următoarele:

- *clic* - menționat deja, declanșat când este selectat cu mausul un element HTML;
- *mouseover* - declanșat când cursorul mausului este plasat peste un element HTML;
- *mouseout* - declanșat când cursorul mausului părăsește dreptunghiul de încadrare al unui element HTML;
- *change* – declanșat de modificarea stării sau a conținutului unui element destinat interacțiunii cu operatorul (un element *<input>* de tip *text*, *checkbox*, *select*, etc);
- *submit* – declanșat când operatorul selectează un buton de tip *submit* al unui formular.

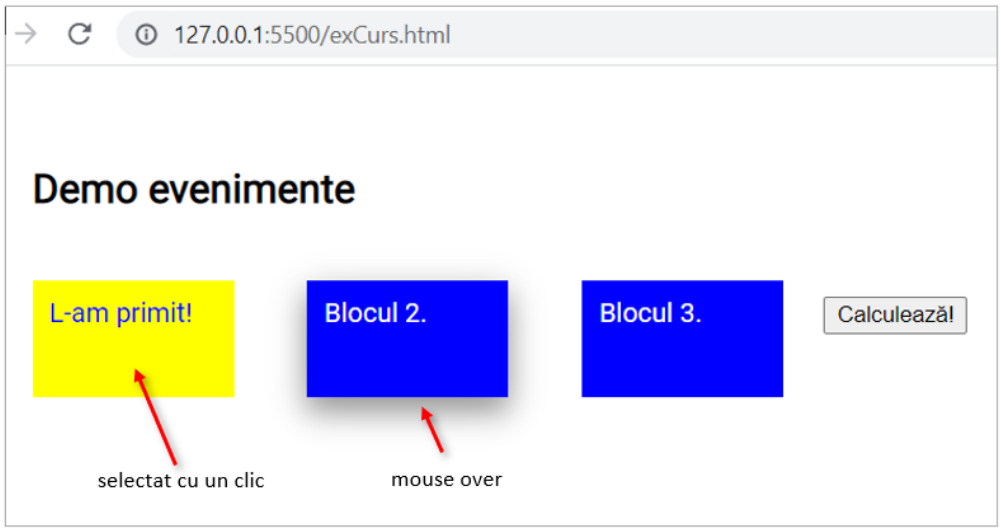
Valoarea primită de o funcție de tip *callback* apelată pentru tratare a unui eveniment este un obiect JavaScript de tip *event*. Din multitudinea de

proprietăți pe care le are un astfel de obiect, cea mai utilizată este *target*. Această proprietate permite identificarea elementului HTML care a declanșat un anumit eveniment. Acest lucru este important de exemplu în cazul în care aceeași funcție de tip *callback* trebuie declanșată de elemente HTML diferite.

Exemplu:

```
<!DOCTYPE html>
<html>
 <head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-s
cale=1">
  <link href='https://fonts.googleapis.com/css?family=Roboto'
      rel='stylesheet'>
  <title>Evenimente</title>
  <style>
     body { padding: 30px; font-family: Roboto, sans-serif; }
     h2 {margin: 20px; }
     drept { width: 100px; height: 50px; padding: 10px; margin
: 20px; background-color: blue; color: white; display: inline-
block; }
  </style>
</head>
<body>
   <h2>Demo evenimente</h2>
   <div class="umbra drept" id="elclic">Aștept clic.</div>
   <div class="umbra drept">Blocul 2.</div>
   <div class="umbra drept">Blocul 3.</div>
   <button id="calcul" class="umbra">Calculează!</button>
   <script>
      const efectClic = evn => {
         const bloc = evn.target;
         bloc.innerHTML = "L-am primit!";
         bloc.style.color = "blue";
         bloc.style.backgroundColor = "yellow";
      };

      document.querySelector("#elclic").onclic = efectClic;

      const cuUmbrire = (evn) => {
         const bloc = evn.target;
           // bloc este elementul pe care este cursorul
         bloc.style.boxShadow = "0 10px 16px 0 rgba(0,0,0,0.2)
,0 6px 20px 0 rgba(0,0,0,0.29)";
      }
```

```
      const faraUmbrire = (evn) => {
         const bloc = evn.target;
         // Anulez efectul de umbrire
         bloc.style.boxShadow = '';
      }

      const elemUmbrite =
                  [... document.querySelectorAll(".umbra")];
         //  Selectare multipla.

      const umbrire = (item) => {
        item.addEventListener("mouseover", cuUmbrire);
        item.addEventListener("mouseout", faraUmbrire);
      };

      elemUmbrite.map(umbrire);
   </script>
</body>
</html>
```



Analiza exemplului:

Codul HTML începe prin declararea unui titlu (`<h2>`) după care sunt inserate trei elemente `<div>` (trei dreptunghiuri) și un element HTML de tip `<button>`.

Elementelor cărora dorim să le impunem un efect de umbrire (prin tratarea evenimentului *mouseover*) li s-a atașat aceeași clasă CSS (clasa *umbra*).

Pentru a atașa fiecăruia dintre elementele având clasa *umbra* perechea de funcții de tratare *cuUmbrire* respectiv *faraUmbrire* declanșate de evenimentele *mouseover* respectiv *mouseout*, s-a creat șirul de obiecte JavaScript *elemUmbrite* scriind:

```
const elemUmbrite =
          [... document.querySelectorAll(".umbra")];
```

Apoi folosind funcția *map()*, fiecărui element din șirul creat i s-au atașat funcții de tratare a evenimentelor *mouseover* și *mouseout*. Pentru mai multă claritate, funcția necesară în *map()* a fost scrisă separat.

```
const umbrire = (item) => {
   item.addEventListener("mouseover", cuUmbrire);
   item.addEventListener("mouseout", faraUmbrire);
};

elemUmbrite.map(umbrire);
```

Observații:

- Funcțiile de tip *callback* pot fi scrise și fără parametru. Scrise astfel, ele nu vor primi acel obiect de tip *event* menționat. Acest mod de scriere este folosit dacă în funcție nu sunt necesare valori a căror accesare necesită cunoașterea obiectului de tip *event*, respectiv obiectul care a declanșat evenimentul (*evn.target*).
- Funcția *querySelectorAll()* din exemplul precedent se folosește pentru a selecta mai multe obiecte din DOM care îndeplinesc o condiție. Pentru a o putea utiliza, clasa CSS *umbra* a fost adăugată tuturor elementelor HTML care urmau să constituie șirul de elemente. Clasa *umbra* nu are o descriere, rolul ei este doar cel menționat: să permită crearea unui șir de obiecte folosind *querySelectorAll()*. Elementelor din șirul creat li s-a putut apoi impune un comportament comun definit de funcțiile *cuUmbrire()* și *faraUmbrire()*.
- Așa cum s-a afirmat deja la p.47, funcția *querySelectorAll()* nu creează un șir de obiecte (un *array*) ci un obiect aparținând unei clase particulare - *nodeList*. Prin urmare se impune conversia

obiectului de tip *nodeList* în *array* ceea ce se poate realiza folosind operatorul [...] (*spread operator*):

```
const sirObiecte = [ ...querySelectorAll(".numeClasa")];
```

Aplicație:

Pentru a înțelege mai bine cum se scriu aplicațiile web folosind doar HTML și JavaScript, în continuare va fi realizată o aplicație care afișează programul stagiunii curente de la Opera Română.

Realizarea unei astfel de aplicații presupune parcurgerea câtorva pași, astfel:

1. **Se creează fișierul** *index.html*. De regulă aplicațiile de acest fel conțin un singur fișier în format HTML, *index.html*. Caracteristic pentru acest fișier este faptul că va conține câteva elemente HTML care, pentru a putea fi selectate în codul JavaScript, au atribute *"id"*.

Pentru aplicația care va fi realizată s-a ales varianta utilizării colecției de componente prestilizate *Bootstrap* . Aceasta poate fi accesată la adresa *https://getbootstrap.com/docs/5.1/getting-started/introduction/*.

În prima sa formă, fișierul *index.html* va avea următorul conținut:

```
<!DOCTYPE html>
<html lang="en">
<head>
   <!-- Required meta tags -->
   <meta charset="utf-8">
   <meta name="viewport"
            content="width=device-width, initial-scale=1">

   <!-- Bootstrap CSS -->
   <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/di
st/css/bootstrap.min.css" rel="stylesheet" integrity="sha384-1
BmE4kWBq78iYhFldvKuhfTAU6auU8tT94WrHftjDbrCEXSU1oBoqyl2QvZ6jIW
3" crossorigin="anonymous">

   <title>Opera romana</title>
</head>
<body>
   <div class="container">
      <h1 id="titlu"></h1>
      <div id="continut"></div>
   </div>
```

```
   <script></script>
</body>
</html>
```

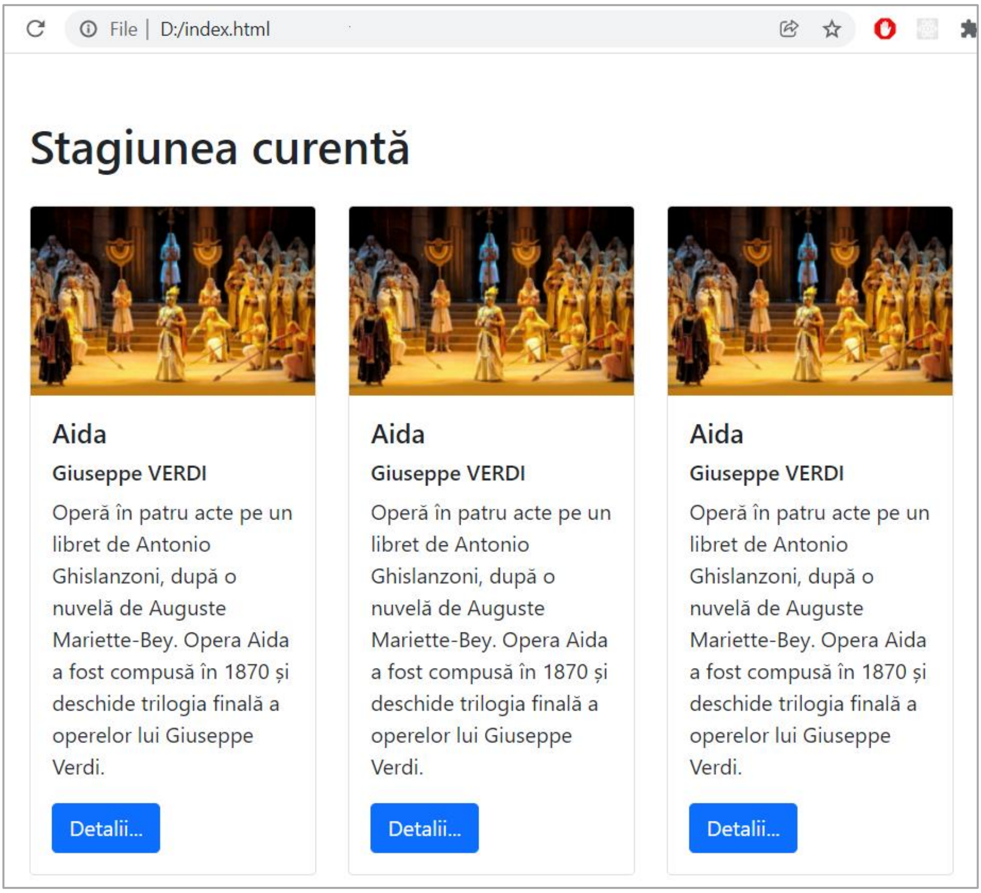
Afișarea acestei pagini este inutilă deoarece elementele din pagină nu conțin nimic.

2. **Se creează o variantă statică** a conținutului care va fi afișat de aplicație.

```
<body>
  <div class="container">
    <h1 id="titlu" class="mt-5 mb-3">Stagiunea curentă</h1>
    <div id="continut" class="row">

      <div class="col-sm-6 col-md-4 col-lg-3">
        <div class="card mt-2 mr-1 mb-2" style="max-width: 18r
em;">
          <img src="imagini/aida.jpg" class="card-img-top">
          <div class="card-body">
            <h5 class="card-title">Aida</h5>
            <h6>Giuseppe VERDI</h6>
            <p class="card-text">Operă în patru acte pe un lib
ret de Antonio Ghislanzoni, după o nuvelă de Auguste Mariette-
Bey. Opera Aida a fost compusă în 1870 și deschide trilogia fi
nală a operelor lui Giuseppe Verdi.</p>
            <button class="btn btn-primary">Detalii..</button>
          </div>
        </div>  <!-- / card -->
      </div>

    </div>  <!-- / continut -->
  <div>  <!-- / container -->
  <script></script>
</body>
```

Blocul evidențiat poate fi repetat de mai multe ori pentru a vedea modul de afișare dacă dimensiunea ecranului se modifică.

Rezultat posibil:



3. **Se scrie codul JavaScript**

Codul JavaScript prezentat în continuare conține o funcție (denumită `carduri`) care inserează în blocul având `id="continut"` un ansamblu de componente de tip `"card"`. Scriptul începe însă cu declararea unui șir de obiecte (șirul `opere`) care conțin datele care trebuie inserate ulterior în carduri.

```
<body>
 <div class="container">
  <h1 id="titlu" class="mt-5 mb-3">Stagiunea curentă</h1>
  <div id="continut" class="row">
  </div>
 </div>
 <script>
    const opere = [
```

```
      {id: 1, img: "imagini/aida.jpg", nume: "Aida",
      autor: "Giuseppe Verdi",
      descriere: `Operă în patru acte pe un libret de Antonio
Ghislanzoni, după o nuvelă de Auguste Mariette-Bey. Opera Aida
a fost compusă în 1870.`, data: "12 martie 2022", ora: "18.00"
},
      {id: 2, img: "imagini/butterfly.jpg",
      nume: "Madama Butterfly", autor: "Gicomo Puccini",
      descriere: `Operă în trei acte pe un libret de Luigi Il
lica și Giuseppe Giacosa. Premiera mondială a avut loc la 17 f
ebruarie 1904 la Teatro alla Scala din Milano.`, data: "18 mar
tie 2022", ora: "18.00"},
      {id: 3, img: "imagini/don_giovanni.jpg",
      nume: "Don Giovanni", autor: "Wolfgang Amadeus Mozart",
      descriere: `Opera buffa/ Dramma giocoso în două acte, p
e un libret de Lorenzo da Ponte. Premiera operei a avut loc în
29 octombrie 1787 la Estates Theatre din Praga.`, data: "24 ma
rtie 2022", ora: "18.00"},
      {id: 4, img: "imagini/trubadurul.jpg", nume: "Trubaduru
l",
      autor: "Giuseppe Verdi",
      descriere: `Trubadurul (titlul original: în italiană Il
trovatore) este o operă în patru acte compusă de Giuseppe Verd
i, pe un libret de Salvatore Cammarano, scris după tragedia El
trovator de Antonio Garcia Gutiérrez. Premiera operei a avut l
oc la Teatro Apollo din Roma, la data de 19 ianuarie 1853.`, d
ata: "26 martie 2022", ora: "18.00"},
      {id: 5, img: "imagini/bal_mascat.jpg", nume: "Bal masca
t",
      autor: "Giuseppe Verdi",
      descriere: `Operă în trei acte pe un libret de Antonio
Somma, inspirată din piesa Gustav al III-lea de Eugéne Scribe.
„Un Ballo in maschera” este cea de-a douăzeci și doua creație
a lui Giuseppe Verdi, recunoscută pentru parcursul fascinant,
începând cu impactul fulminant al premierei sale mondiale din
anul 1859 și până în prezent.`, data: "27 martie 2022", ora: "
18.00"}
  ];

  const carduri = () => {
    listaOpere =
        opere.map((item) =>
         `<div class="col-sm-6 col-md-4 col-lg-3">
         <div class="card mt-2 mb-2">
           <img src="${item.img}" class="card-img-top">
             <div class="card-body">
               <h5 class="card-title">${item.nume}</h5>
               <h6>${item.autor}</h6>
```

```
                <p class="card-text">${item.descriere}</p>
                <button type="button" class="btn btn-primary"
                      id="${item.id}">Detalii...</button>
             </div>
            </div>
          </div>`)
       .reduce((html, item) => html + item, "");
       document.querySelector("#continut").innerHTML =
           listaOpere;
  }

  //  Apelez functia carduri()
  carduri();

  //  Functia detalii()
  const detalii = (evt) => {
       alert(`Opera ${evt.target.id}`);
  }

  //  Asocierea evenimentului "clic" pe un buton "Detalii" unei
    //  functii de tratare
  document.querySelector("#continut").onclick = detalii;

 </script>
</body>
```

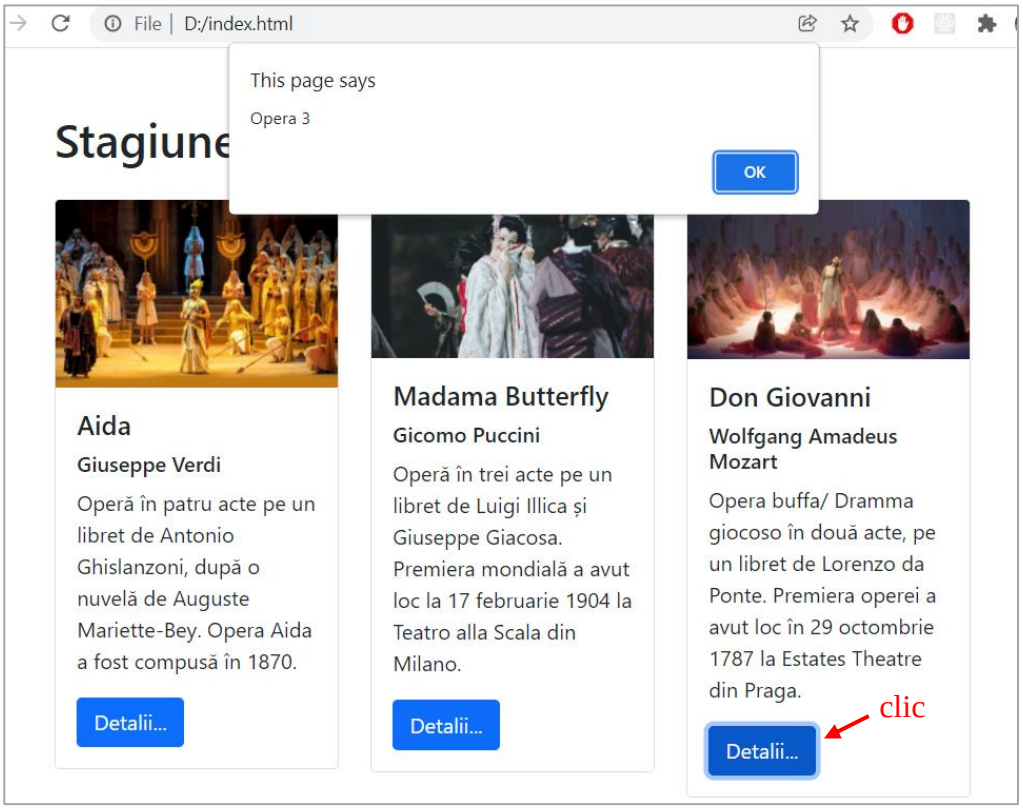
Partea finală a scriptului conține funcția `detalii()`, apelată la selectarea unui buton `"Detalii..."` de pe un card. Funcția afișează doar `id`-ul butonului selectat. Într-o aplicație reală, cunoașterea acestuia este suficientă pentru a genera un nou conținut pentru blocul având `id="continut"`, adecvat noii stări a aplicației.

Important: Asocierea funcției de tip `callback` denumită `detalii()`, unui eveniment de tip `click` s-a realizat scriind:

```
document.querySelector("#continut").onclick = detalii;
```

Codul acesta fiind scris înafara oricărei funcții JavaScript, asocierea nu putea fi realizată decât folosind **un obiect existent deja în DOM**, deci un element HTML conținut în codul static al paginii. Este vorba, desigur, despre elementul `<div id="continut" class="row"></div>`. Dar, așa cum rezultă din informația afișată în fereastra `alert` afișată de funcția `detalii()` aceasta nu e o problemă deoarece obiectul de tip `event` primit

de funcția `detalii()` permite identificarea elementului care a generat apelul, chiar dacă acesta a fost generat dinamic.



Modularizarea codului în JavaScript ES6

JavaScript ES6 permite scrierea unor module plasate în fișiere distincte. Interesul pentru modularizare este legat de necesitatea reducerii dimensiunii fișierelor JavaScript și de asigurarea portabilității unor părți din codul scris.

Regula de la care se pornește este aceea că fiecare fișier JavaScript ES6 este în același timp un modul. Pentru a face ca elemente definite într-un modul (variabile, funcții sau clase) să poată fi utilizate și înafara acestuia, acestea trebuie exportate și apoi importate în modulul în care trebuie utilizate.

Exportarea

Elementele exportate pot fi declarate pe rând. Ce nu este declarat ca *export* nu va fi accesibil înafara modulului (fișierului) curent. Exemplu:

modul.js

```
export const lgMes = (mesaj, n) => {
  console.log(`#${n}: Mesaj: ${mesaj}`);
  return `#${n}: Mesaj: ${mesaj}`;
};

export const smMes = (mesaj) => {
  console.log(`Mesaj: ${mesaj}`);
  return `Mesaj: ${mesaj}`;
};

export const xsMes = (mesaj) => {
  console.log(mesaj);
  return mesaj;
};
```

Declararea elementelor exportate poate fi realizată mai simplu, într-o singură comandă *export* plasată la sfârșitul modulului:

```
const lgMes = (mesaj, n) => {
  console.log(`#${n}: Mesaj: ${mesaj}`);
  return `#${n}: Mesaj: ${mesaj}`;
};

const smMes = (mesaj) => {
  console.log(`Mesaj: ${mesaj}`);
  return `Mesaj: ${mesaj}`;
};

const xsMes = (mesaj) => {
  console.log(mesaj);
  return mesaj;
};

export { lgMes, smMes, xsMes };
```

În momentul exportării, entităților li se poate asocia un *alias*, folosind cuvântul rezervat *as*:

```
  export { lgMes, smMes as smM, xsMes };
```

Exportul de tip *default*

Pentru a defini un export de acest tip se folosește cuvântul rezervat `default`:

```
const lgMes = (mesaj, n) => {
  console.log(`#${n}: Mesaj: ${mesaj}`);
  return `#${n}: Mesaj: ${mesaj}`;
};

const smMes = (mesaj) => {
  console.log(`Mesaj: ${mesaj}`);
  return `Mesaj: ${mesaj}`;
};

const xsMes = (mesaj) => {
  console.log(mesaj);
  return mesaj;
};

class Client {
  constructor(nume, prenume, email) {
    this.nume = nume;
    this.prenume = prenume;
    this.email = email;
  }

  numePren = () => {
    return this.nume + " " + this.prenume;
  }

  prenNume = () => {
    return this.prenume + " " + this.nume;
  }

  prenNumeMaj = () => {
    return this.nume.toUpperCase() + " " + this.prenume;
  }
}

export { lgMes, smMes, xsMes };
export default Client;
```

Observație: Un modul poate exporta un singur element folosind `export default`, dar oricâte folosind `export`.

Importarea

Importul elementelor este la fel de simplu. Denumirile elementelor importate trebuie plasate între acolade și trebuie indicat fișierul care conține modulul din care se importă. Evident, entitățile care se importă au fost exportate în modulul din care se importă.

```
import { lgMes, smMes } from './modul';
```

Extensia fișierului din care se importă nu este obligatorie, dar calea ("*./*") trebuie indicată.

În exemplul dat, modul de scriere a comenzii *import* indică faptul că fișierul *modul.js* și fișierul în care se importă funcțiile *lgMes* și *smMes* sunt conținute în același director.

Observație: Ca și în cazul exportului, și în momentul importării pot fi atribuite nume diferite pentru unele dintre entitățile importate (*alias*-uri). Sintaxa este aceeași.

```
import {lgMes as mesLung, smMes} from './modul';
```

Importul tuturor entităților exportate într-un modul se realizează folosind sintaxa:

```
import * as Msg from './modul';
```

Membrii astfel importați pot fi accesați folosind notația cu punct, ca în cazul câmpurilor unui obiect:

```
Msg.smMes("Salut!");
```

Importul unui element exportat folosind cuvântul rezervat *default* se face dând elementului importat un nume oarecare:

```
import Contact from './modul';
```

Sau, dacă pe lângă importul elementului declarat *default* se mai dorește și importul altor elemente din cele exportate, se poate folosi sintaxa:

```
import Contact, { lgMes, smMes } from './modul';
```

React

Scurtă prezentare

React este numele unei biblioteci de funcții creată de Facebook.

Întrebarea care s-ar putea pune este dacă React poate fi utilizat pentru realizarea tuturor tipurilor de aplicații (aplicații web și aplicații Windows clasice, de tip "desktop"). Evident, atâta timp cât partea de back-end respectă standardul `REST` menționat deja, răspunsul este afirmativ. Există de altfel și exemple de aplicații de tip `desktop` foarte cunoscute care folosesc React: `Atom` și `Visual Studio Code`. Pentru cei care cunosc React, deosebit de valoroasă este o extensie denumită `React-native` deoarece aceasta permite utilizarea React și pentru producerea aplicațiilor mobile.

Principiile care stau la baza React

React permite crearea productivă de aplicații web în arhitectură `SPA` (prescurtare de la `Single Page Application`). În varianta clasică (HTML5 + CSS3 + JavaScript...) crearea unei astfel de aplicații presupune realizarea unei pagini web de dimensiuni uneori prea mari și scrierea unuia sau mai multor fișiere JavaScript care conțin funcții de tratare a unor evenimente. Riscul de a avea greșeli de programare în astfel de aplicații este mare deoarece structurarea codului este frecvent deficitară.

React propune înlocuirea lucrului la aplicație în integralitatea ei cu dezvoltarea și testarea separată a unui ansamblu de componente urmată de integrarea acestora în interfața grafică a aplicației (fig. 3.1).

Observație: Odată realizate, componentele React pot fi folosite în mod repetat în aplicația curentă, putând fi reutilizate apoi și în alte aplicații. Din acest punct de vedere, React urmează calea clasică de creștere a

productivității în programare, respectiv crearea de module independente, reutilizabile.

Figura 3.1. Realizarea unei interfețe folosind componente

Crearea primei aplicații *React*

Odată încheiată instalarea și configurarea aplicațiilor menționate în capitolul 1 (*Node.js*, *VS Code* și *Git*) și, eventual, crearea unui cont pe platforma *GitHub*, se poate începe crearea unei prime aplicații. Această aplicație se va numi *Carti* și va realiza afișarea unui număr de cărți dintr-o bibliotecă digitală.

Pentru a crea aplicația, se va selecta în *File Explorer* directorul în care aceasta va fi construită. În continuare se poate proceda în mai multe moduri, o variantă fiind deschiderea în directorul selectat a unei ferestre *Command Prompt* (fig. 3.2).

Pentru crearea aplicației React `Carti` se va utiliza fișierul de comenzi `create-react-app` oferit de *Facebook Co*. Deși procesul generării unei aplicații React este complex, fișierul de comenzi menționat va permite simplificarea acestuia, fiind necesară tastarea unei singure comenzi:

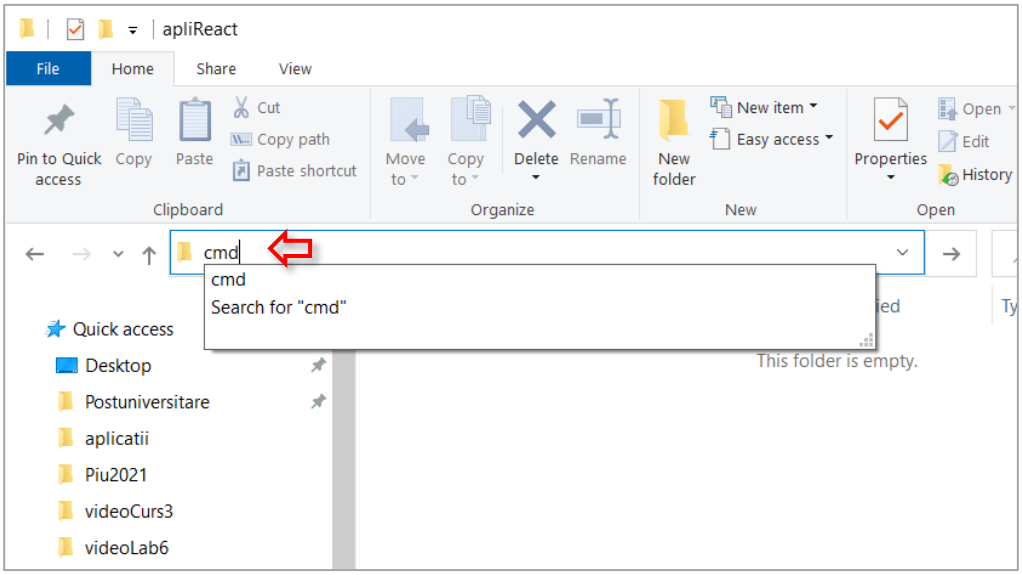
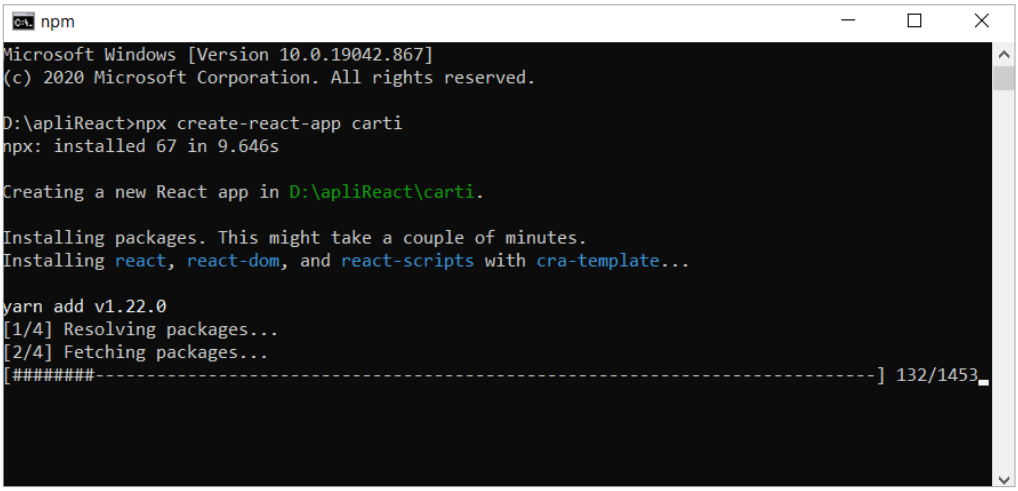


Figura 3.2. Deschiderea aplicației `Command Prompt`

```
npx create-react-app carti
```



Aplicația va fi creată într-un director derivat din directorul curent purtând numele aplicației nou create (*carti*).

După crearea aplicației, pasul următor va fi începerea dezvoltării acesteia. Pentru aceasta va fi deschisă aplicația *VS Code* și apoi directorul proiectului, *carti* (*File / Open Folder...*).

Crearea componentelor necesare

Integrarea în *VS Code* a extensiei *Simple React Snippets* permite crearea rapidă (și fără erori de sintaxă!) a componentelor React.

În cele ce urmează, fiecare componentă va fi creată în propriul fișier. Dacă o componentă are și un fișier de stiluri propriu, acesta va avea același nume cu cel atribuit fișierului componentei (extensia fiind evident, *.css*) și va fi memorată în același director.

Există de altfel un principiu în React, respectiv fișierul care conține o componentă React trebuie să conțină tot ce este necesar acesteia pentru a funcționa (eng. *self contained*).

Pentru testarea componentelor care vor fi realizate, acestea vor fi referite în componenta *<App />* (conținută în fișierul *App.js*) care este referită la rândul ei în *index.js*.

Înainte de a începe realizarea aplicației vor fi suprimate din fișierele *index.js* și *App.js* liniile neutilizate. Conținutul acestora va fi următorul:

index.js

```
import React from 'react';
import { render } from 'react-dom';
import App from './App';

render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
  document.getElementById('root')
);
```

App.js

```
import React from 'react';

function App() {
  return (
    <div></div>
  );
}

export default App;
```

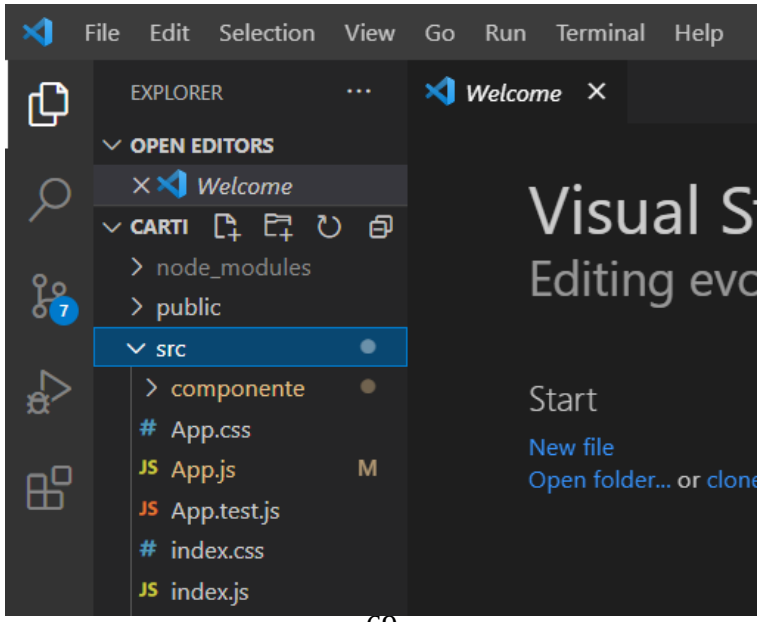
Crearea Componentei *<Carte />*

Componentele React sunt marcate ca și elementele HTML5, respectiv sunt încadrate între caractere "<" și ">". Spre deosebire însă de elementele HTML5, denumirile acestora încep cu o majusculă.

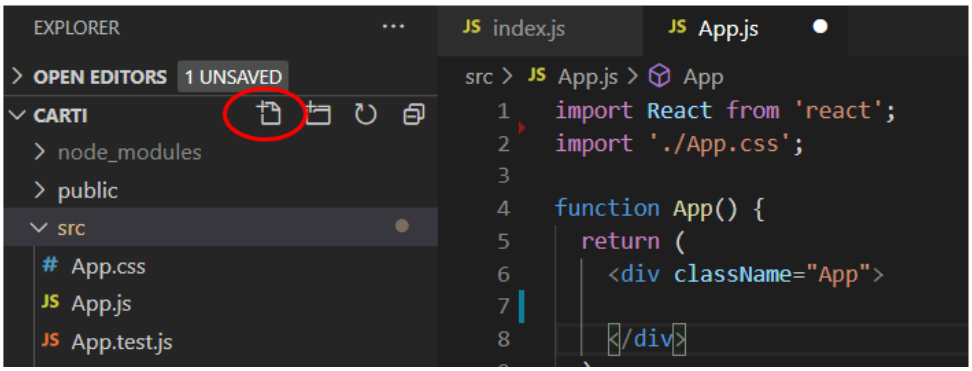
În urma procesului de generare, aplicația *Carti* conține în directorul *carti /src* o componentă React denumită *<App />*.

Pentru a crea o componentă nouă, specifică aplicației, denumită *<Carte />*, se procedează astfel:

1. În *VS Code* se selectează subdirectorul *src*. Acest director va conține de altfel toate componentele aplicației *carti* care vor fi create în continuare.



2. În continuare se selectează `File` / `New File` sau pictograma din imagine:



3. În fereastra caseta de text afișată de aplicație se va tasta denumirea noului fișier (`carte.js`):

Observație: O componentă React poate fi definită utilizând fie o funcție JavaScript, fie o clasă ES6. În cele ce urmează se va utiliza varianta utilizării funcțiilor, codul fiind de regulă mai ușor de înțeles.

4. În fereastra de editare deschisă automat de aplicația `VS Code` se va tasta apoi codul componentei `<Carte />`. Profitând de extensia *React Snippets* instalată, codul acesteia va fi generat într-o primă formă tastând în fereastra de editare două secvențe de caractere. După fiecare secvență introdusă va fi apăsată tasta `Tab`.

`imr + Tab` va insera în fișier comanda `import React from 'react';`

`sfc + Tab` va adăuga câteva linii de cod reprezentând structura specifică a unei componente *React* de tip funcție.

Rezultat:

```
import React from 'react';

const Carte = () => {
    return (  );
}

export default Carte;
```

Observație: După tastarea șirului de caractere "`sfc`", în fereastra de editare se vor vedea două cursoare. Dacă se continuă imediat cu tastarea numelui noii componente (`Carte`), acesta va fi inserat în două locuri: ca denumire a funcției și ca parametru al comenzii finale, `export default`.

O variantă a componentei `<Carte />` mai aproapiată de ce ne dorim se obține inserând după `return` codul necesar afișării pe ecran a informațiilor despre o carte.

Acest cod va fi cod `JSX` (prescurtare de la `JavaScript XML`), regulile de scriere specifice acestuia fiind prezentate în continuare.

```
import React from 'react';

const Carte = () => {
    return (   //  Urmeaza cod JSX
        <div>
            <h3>Harry Potter si Piatra Filosofala</h3>
            <p>Lorem ipsum dolor sit amet, consectetur adipisi
cing elit, sed do eiusmod tempor incididunt ut labore et dolor
e magna aliqua. Ut enim ad minim veniam, quis nostrud exercita
tion ullamco laboris nisi ut aliquip ex ea commodo consequat.<
/p>
            <p>J.K. Rowling. Preț: 30.32 Lei</p>
        </div>
    );
}

export default Carte;
```

`JSX` este efectiv o extensie a limbajului JavaScript și a fost introdusă odată cu finalizarea versiunii 2015 (ES6) a acestuia.

Se observă similitudinea dintre `HTML5` și `JSX`. Practic pentru descrierea modului de afișare a componentei folosind `JSX` s-a folosit cod `HTML5`.

Pentru a vedea pe ecran componenta realizată, codul componentei `<App />` va trebui modificat astfel:

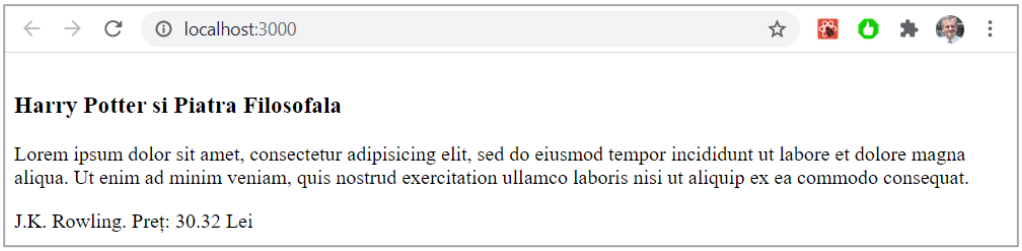
```
import React from 'react';
import Carte from "./carte";

function App() {
  return (
    <div>
      <Carte />
```

```
    </div>
  );
}

export default App;
```

Pentru aceasta serverul de web de dezvoltare disponibil în *Node.js* trebuie să fie lansat în execuție. Aceasta se realizează tastând într-o fereastră *Command Prompt*deschisă în directorul proiectului comanda:

```
npm start
```

După pornira serverului, în fereastra browserului implicit va fi afișat următorul conținut:



Observație: Ameliorarea cunoștințelor de operare a aplicației VS Code se poate realiza accesând unul dintre tutorialele dedicate utilizării acesteia sau prin realizarea unui proiect React, de exemplu *https://www.youtube.com/watch?v=pCA4qpQDZD8* (desigur, doar primele minute).

Reguli de codificare în JSX a componentelor React

În aplicațiile React, interfața grafică expusă de diferitele componente utilizate este creată folosind *JSX*.

JSX permite utilizarea elementelor HTML5 în JavaScript. Elementele HTML5 astfel descrise vor fi inserate automat în *DOM*.

Regulile de scriere a codului *JSX* sunt simple și rezultă din exemplele următoare:

1. Scrierea unui cod JSX minimal, fără a insera expresii JavaScript:

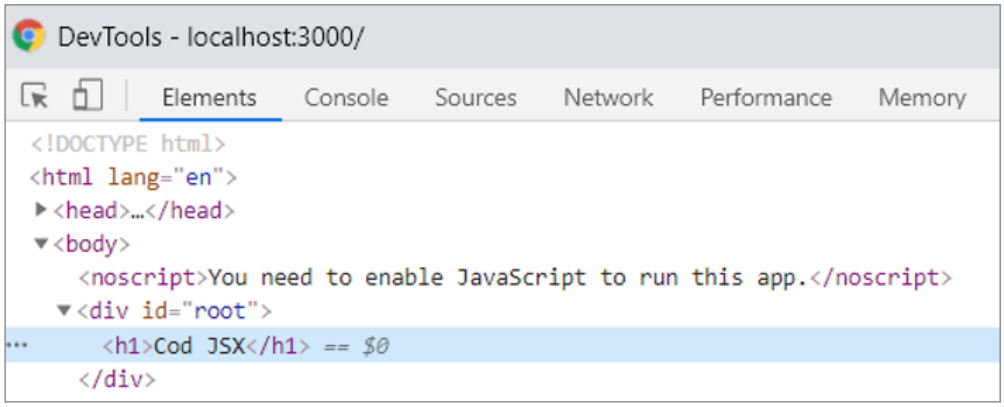
```
// Scriere direct in index.js
import React from "react";
import ReactDOM from "react-dom";
```

```
ReactDOM.render(<h1>Cod JSX</h1>, document.getElementById(
   "root"));
```

Sau, dacă se creează mai întâi o componentă React, conținutul fișierului *index.js* ar putea fi următorul:

```
import React from 'react';
import ReactDOM from 'react-dom';

const ExJSX = () => <h1>Cod JSX</h1>;

ReactDOM.render(<ExJSX />, document.getElementById('root'));
```

Rezultat:



Așa cum se poate observa în codul componentei *<Carte />*, în cazul în care codul JSX trebuie scris pe mai multe linii, acesta trebuie cuprins între paranteze:

```
const Carte = () => {
  return (
      <div>
         <h3>Harry Potter si Piatra Filosofala</h3>
```

```
        <p>Lorem ipsum dolor sit amet, consectetur adipisicin
g elit, sed do eiusmod tempor incididunt ut labore et dolore m
agna aliqua. Ut enim ad minim veniam, quis nostrud exercitatio
n ullamco laboris nisi ut aliquip ex ea commodo consequat.</p>
        <p>J.K. Rowling. Preț: 30.32 Lei</p>
      </div>
  );
}
```

2. Dacă în codul JSX se folosesc expresii JavaScript, acestea trebuie incluse între acolade (`{}`):

```
import React from 'react';
const Carte = () => {
    const titlu = "Harry Potter si Piatra Filosofală";
    return (
        <div>
            <h3>{titlu.toUpperCase()}</h3>
            <p>Lorem ipsum dolor sit amet, consectetur adipisi
cing elit, sed do eiusmod tempor incididunt ut labore et dolor
e magna aliqua. Ut enim ad minim veniam, quis nostrud exercita
tion ullamco laboris nisi ut aliquip ex ea commodo consequat.<
/p>
            <p>{autor}. Preț: {pret} Lei</p>
        </div>
    );
}

export default Carte;
```

3. Codul care descrie modul de afișare a unei component React trebuie să fie inclus într-un element HTML cu rol de container. În interiorul containerului pot fi adăugate apoi elemente HTML și componente React. De cele mai multe ori containerul va fi un element `<div>`.
4. Dacă o componentă este descrisă folosind mai multe elemente și niciunul nu poate juca rolul de container, de exemplu dorim să avem:

```
<div id="root">  //  Exista în index.html

     <h1>Prima componentă React!!!</h1>
     <p>Acesta este cod JSX valabil.</p>

</div>
```

atunci codul JSX al componentei trebuie scris astfel:

```
render() {
   const componenta = (
      <>
         <h1>Prima componentă React!!!</h1>
         <p>Acesta este cod JSX valabil.</p>
      </>
  );

  return componenta;
}
```

Deci se poate folosi ca element de grupare *o pereche de marcaje nespecificate*: *<> ... </>*.

React-Bootstrap

Componenta *<Carte />* este în acest moment nestilizată. Pentru a-i schimba aspectul, o soluție este utilizarea unui ansamblu de reguli de stilizare (CSS). Există însă și o cale mai tehnică, respectiv utilizarea unei colecții de componente prestilizate.

Există multe astfel de colecții, printre cele cele mai utilizate numărându-se *Material.ui* și *React-Bootstrap*.

În continuare se va utiliza *React-Bootstrap*. Preluând logica *Bootstrap*-ului, adoptarea acestei colecții ar trebui să nu pună probleme.

Instalarea colecției de componente React-Bootstrap

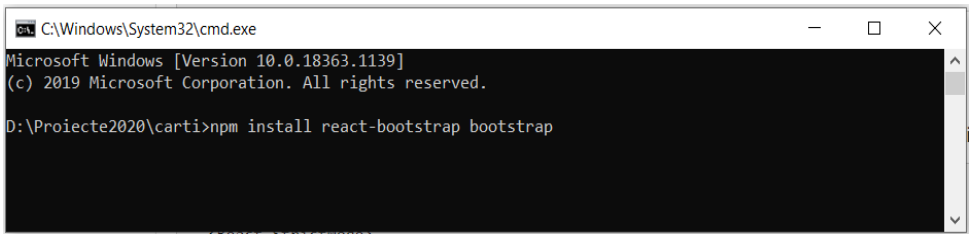
Pentru a adăuga *React-Bootstrap* aplicației începute, într-o fereastră *Command Prompt* în care directorul curent este chiar directorul rădăcină al proiectului, se tastează comanda:

```
npm install react-bootstrap bootstrap
```

Această comandă va instala ultimele versiuni ale celor două extensii. Dar, deoarece acestea sunt dezvoltate continuu, pentru a nu avea erori cauzate de trecerea de la versiunea utilizată în cadrul acestei cărți la o altă versiune, comanda tastată ar putea fi următoarea:

```
npm install react-bootstrap@2.0.3 bootstrap@5.1.3
```

În acest mod se impune instalarea celor două extensii folosind versiunile `2.0.3` respectiv `5.1.3`.



Observație: Această comandă va insera în aplicație atât `React-Bootstrap` cât și `Bootstrap`. Logica acestei soluții este dată de faptul că aspectul și funcționarea componentelor `React-Bootstrap` se bazează pe `Bootstrap`.

De altfel, după încheierea procesului de instalare, la începutul fișierului `index.js`, înainte de comanda `import App from "./App";` se va insera o linie în care se face referire la fișierul `bootstrap.min.css`, care conține regulile de stilizare definite în *Bootstrap*.

`index.js`

```
import React from "react";
import { render } from "react-dom";
import "bootstrap/dist/css/bootstrap.min.css";
import App from "./App";

const rootElement = document.getElementById("root");

render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
  rootElement
);
```

Importarea elementelor `React-Bootstrap`

Importurile vor preciza elementele care vor fi ulterior utilizate în codul de descriere al componentei React. Exemple:

```
import Button from "react-bootstrap/Button";
```

```
import Container from "react-bootstrap/Container";
import Card from "react-bootstrap/Card";
. . .
```

Acest set de importuri poate fi însă scris mai compact deoarece toate se referă la aceeași extensie ("*react-bootstrap*"):

```
import { Button, Container, Card } from "react-bootstrap";
. . .
```

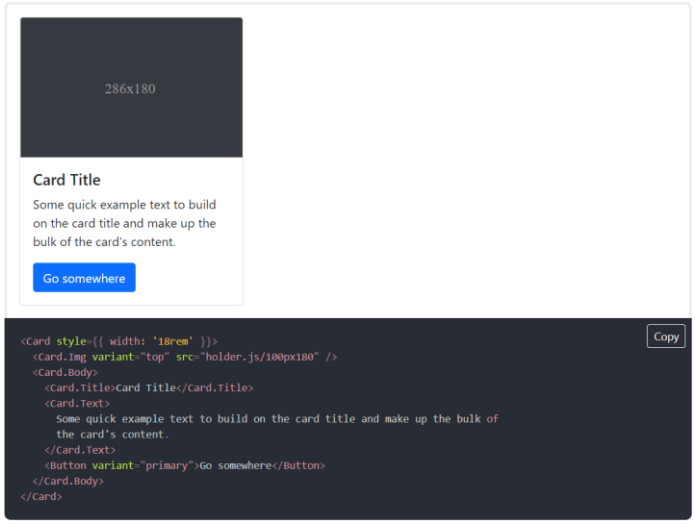
Utilizarea componentei *<Card />* pentru descrierea componentei *<Carte />*

În cele ce urmează, pentru stilizarea informației aferente unei cărți se va utiliza o componentă *React-Bootstrap* denumită *<Card />*. Aceasta preia de altfel aspectul și funcțiile componentei cu același nume din *Bootstrap*.

Pentru a o putea utiliza trebuie accesată descrierea acesteia din site-ul colecției de componente *React-Bootstrap* (*https://react-bootstrap.netlify.app/components/cards/#cards*).

Informațiile care trebuie preluate sunt următoarele:

- Codul unei variante de componentă *<Card />* din mulțimea de variante expuse, care se apropie cel mai mult de ceea ce dorim. Exemplu:



- Instrucțiunea *import* care inserează componenta și componentele incluse în aceasta. Pentru *<Card />* aceasta este:

```
  import { Card, Button } from 'react-bootstrap';
```

În final, pentru componenta *<Carte />* codul ar putea fi următorul:

```
import React from 'react';
import { Card, Button } from 'react-bootstrap';

const Carte = () => {
    const src="https://via.placeholder.com/280";
    const titlu = "Cei trei muschetari";
    const text = `Tânărul d'Artagnan sosește la Paris pentru a
se alătura vestitului corp de gardă al regelui.`;
    const autor = "Alexandre Dumas";
    const pret = 20.90;

    const stil = {
        card: { width: '14rem' },
        text: { fontSize: '0.8rem'}
    }

    return (
        <Card style={stil.card}>
            <Card.Img variant="top" src={src} />
            <Card.Body>
                <Card.Title>{titlu.toUpperCase()}</Card.Title>
                <Card.Text style={stil.text}>{text}</Card.Text
>
                <Card.Text style={stil.text}>Autor: {autor}. P
reț: {pret} lei.</Card.Text>
                <Button variant="primary">Descriere</Button>
            </Card.Body>
        </Card>
    )
};

export default Carte;
```

Observații:

1. Imaginea din definiția componentei *<Card />* a fost înlocuită cu o imagine generică generată folosind *https://placeholder.com/*;

2. Datele necesare în porțiunea de cod `JSX` care afișează componenta au fost preluate dintr-un set de patru variabile declarate și inițializate înainte de `return()`.
3. Pentru limitarea lățimii, componenta `<Card />` a trebuit să fie stilizată. Există mai multe variante de stilizare a componentelor React, dar, în majoritatea cazurilor se va utiliza metoda din codul afișat. Ea constă în definirea unui obiect dedicat stilizării (obiectul denumit `stil`). Fiecare câmp al acestui obiect conține stilizarea unui element HTML asupra căruia trebuie să se intervină. Apoi, în codul `JSX`, elementele HTML respective sunt stilizate folosind varianta `in-line` (deci proprietatea `style`).
4. La scrierea proprietăților CSS având denumiri compuse (`font-size`, `background-color`, `background-image` etc.) denumirile vor fi scrise fără cratimă iar al doilea cuvânt din denumire va începe cu majusculă (deci `fontSize`, `backgroundColor`, `backgroundImage`, etc.). De asemenea, valorile proprietăților vor fi scrise ca șiruri de caractere, deci încadrate între caractere " (ghilimele) sau caractere ' (apostroafe).

Rezultat:

Obiectul *props*

Astfel rescrisă, componenta `<Carte />` conține un cod `JSX` general, care poate afișa orice carte, cu condiția ca setul de variabile utilizat (`src`, `titlu`, `text`, `autor` și `pret`) să fi fost inițializate înainte.

În React, o componentă va primi astfel de valori prin intermediul unui obiect dedicat denumit `props` (cuvânt rezervat). Acest obiect va fi creat automat de React folosind valorile setului de atribute declarat în momentul instanțierii componentei. Pentru componenta `<Carte />` de exemplu, dacă în `<App />` va fi instanțiată scriind:

```
import React from 'react';
import Container from "react-bootstrap/Container";
import Carte from "./carte";

const App = () => {
  return (
    <Container>
      <h1>Carti pentru copii</h1>
      <Carte src="https://via.placeholder.com/280"
         titlu="Cei trei muschetari"
         text="Tânărul d'Artagnan sosește la Paris pentru a se
alătura vestitului corp de gardă al regelui."
         autor="Alexandre Dumas"
         pret="20.90"
      />
    </Container>
  );
}

export default App;
```

atributele `src`, `titlu`, `text`, `autor` și `pret` vor deveni automat proprietăți ale obiectului `props`, deci acesta va fi construit de React astfel:

```
const props = {
  src: "https://via.placeholder.com/280",
  titlu: "Cei trei muschetari",
  text: "Tânărul d'Artagnan sosește la Paris pentru a se alătu
ra vestitului corp de gardă al regelui.",
  autor: "Alexandre Dumas",
  pret: "20.90"
}
```

Acest obiect va fi parametrul formal folosit la declararea componentei funcționale `<Carte />` și, în interiorul acesteia, destructurat.

Deci componenta `<Carte />` va trebui rescrisă astfel:

```
import React from 'react';
import { Card, Button } from 'react-bootstrap';

const Carte = (props) => {
    const {src, titlu, text, autor, pret} = props;

    const stil = {
        card: { width: '14rem' },
        text: { fontSize: '0.8rem'}
    }

    return (
        <Card style={stil.card}>
            <Card.Img variant="top" src={src} />
            <Card.Body>
               <Card.Title>{titlu.toUpperCase()}</Card.Title>
               <Card.Text style={stil.text}>{text}</Card.Text>
               <Card.Text style={stil.text}>Autor: {autor}. Pr
eț: {pret} lei.</Card.Text>
               <Button variant="primary">Descriere</Button>
            </Card.Body>
        </Card>
    )
};

export default Carte;
```

Funcția JavaScript `Carte` începe cu destructurarea obiectului `props`. Dar aceasta poate fi realizată însă mai simplu, scriind:

```
import React from 'react';
import { Card, Button } from 'react-bootstrap';

const Carte = ({src, titlu, text, autor, pret}) => {

    const stil = {
        card: { width: '14rem' },
        text: { fontSize: '0.8rem'}
    }
. . .
```

Acest mecanism este modul principal folosit în React pentru a transmite date de la o componentă la alta.

Observație importantă: Astfel realizat, transferul informațiilor de la o componentă la alta se realizează doar într-un sens, de la componenta "mamă" (*<App />*) la componenta "copil" (*<Carte />*).

Componenta *<ListaCarti />*

Aplicația *carti* conține în acest moment două componente React: *<App />* și *<Carte />*.

Deoarece trebuie afișate mai multe componente de tip *<Carte />*, o practică frecvent întâlnită este crearea unei componente React complexe care generează setul de componente elementare de tip *<Carte />* necesar.

Componenta *<Carte />* fiind deja realizată, se poate trece la crearea unei astfel de componente, denumită *<ListaCarti />*.

<ListaCarti /> trebuie să genereze și să afișeze o listă de cărți pornind de la un șir de obiecte JavaScript care conțin informațiile necesare. Într-o primă variantă a aplicației *carti*, șirul de obiecte conținând datele cărților va fi constant, inițializat în componenta *<App />* și transmis componentei *<ListaCarti />* prin *props*. Atributul utilizat în *<App />* la instanțierea componentei *<ListaCarti />* se va numi *listaCarti*:

```
<ListaCarti listaCarti={lista} />
```

Pentru a plasa în fereastra browserului un ansamblu de componente, Bootstrap permite utilizarea unui ansamblu de 12 coloane alăturate. Dimensiunile acestora și modul de ocupare pot fi impuse folosind atribute specifice și clase CSS.

Evident, *React-Bootstrap* preia acest mod de impunere a dispunerii componentelor, o tratare extinsă a subiectului fiind accesibilă la adresa *https://react-bootstrap.netlify.app/layout/grid/*.

În cazul proiectului *carti*, componentele *<Carte />* trebuie să fie afișate pe linii, fiecare componentă având aceeași lățime. O astfel de dispunere se poate codifica astfel:

```
<Container>
   <Row xs="auto" className="justify-content-sm-center">
       <Col>
```

```
                <Carte />
            </Col>

            <Col>
                <Carte />
            </Col>

            ...
        </Row>
    </Container>
```

Pentru afișare se folosește deci o componentă ReactBootstrap *<Container />* care include o componentă *<Row />*. Componenta *<Row />* conține un șir de componente *<Col />*, fiecare astfel de componentă conținând câte o componentă *<Carte />*.

Atributul *xs="auto"* din componenta *<Row />* impune utilizarea doar a spațiului necesar, deci o componentă *<Carte />* va ocupa atâtea coloane câte sunt necesare.

Clasa CSS *justify-content-sm-center* impune centrarea conținutului afișat. Această centrare este necesară în cazul în care componentele afișate nu ocupă toate cele 12 coloane. Exact cazul afișării unui ansamblu de carduri având lățimi care nu sunt exprimate ca număr de coloane ocupate.

Componenta *<ListaCarti />* ar putea fi descrisă astfel:

listacarti.js

```
import React from "react";
import { Container, Row, Col } from 'react-bootstrap';
import Carte from "./carte";

const ListaCarti = ({listaCarti}) => {
   const lista = listaCarti.map(item => {
   return (
      <Col key={item.id}>
        <Carte
            src={item.src}
            titlu={item.titlu}
            text={item.text}
            autor={item.autor}
            pret={item.pret}
        />
      </Col>
    )
   });
```

```
    return (
      <Container>
        <Row xs="auto" className="justify-content-sm-center">
          {lista}
        </Row>
      </Container>
    )
}

export default ListaCarti;
```

Sau, mai compact, folosid pentru generarea conținutului variabilei *item* încă o destructurare:

```
    ...
    const lista = props.listaCarti.map(item => {
       const {src, titlu, text, autor, pret, id} = item;
       return (
          <Col key={id}>
            <Carte src={src} titlu={titlu} text={text} autor={
autor} pret={pret} />
          </Col>
        )
       }
    );
```

Destructurarea poate fi scrisă și mai simplu:

```
   . . .
   const lista = props.listaCarti.map({src, titlu, text, autor
, pret, id} => {
     return (
        <Col key={id}>
   . . .
```

Așa cum se poate observa, generarea șirului de componente care trebuie afișate se bazează pe utilizarea funcției JavaScript *map()*. Aceasta realizează iterarea într-un șir de valori (șirul *props.listaCarti*). Fiecare element din șir este prelucrat de funcția folosită ca argument al funcției *map()* iar *map()* returnează șirul de elemente astfel modificat.

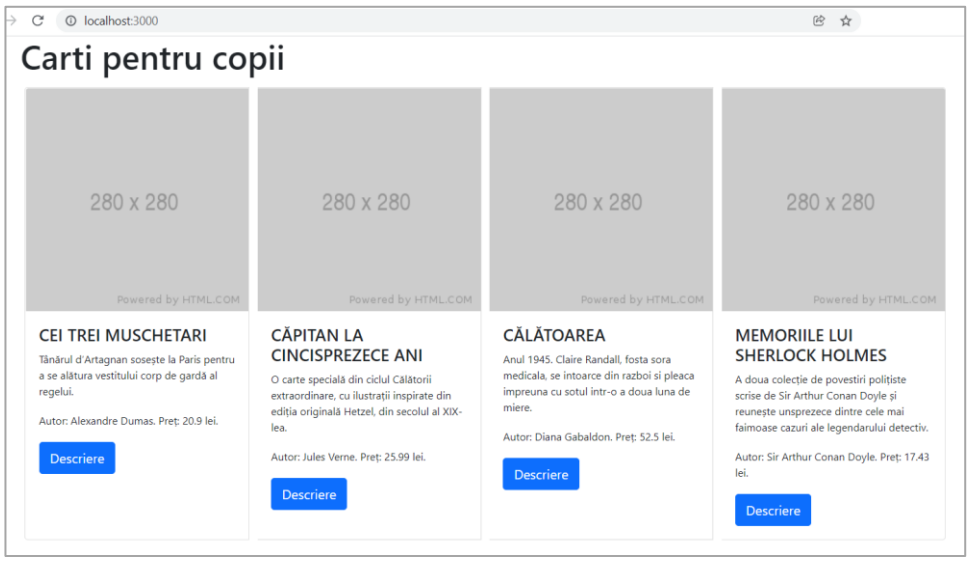
Important! Această soluție se va aplica sistematic când o componentă complexă trebuie să producă un set de componente simple, identice.

Pentru testarea acestei noi componente, care trebuie să primească prin *props* un șir de obiecte, *<App />* se modifică astfel:

```
import React from 'react';
import Container from "react-bootstrap/Container";
import ListaCarti from "./listacarti.js";

const lista = [
  {id: 1,
    src: "https://via.placeholder.com/280",
    titlu: "Cei trei muschetari",
    text: `Tânărul d'Artagnan sosește la Paris pentru a se alăt
ura vestitului corp de gardă al regelui.`,
    autor: "Alexandre Dumas",
    pret: 20.90
  },
  {id: 2,
    src: "https://via.placeholder.com/280",
    titlu: "Căpitan la cincisprezece ani",
    text: `O carte specială din ciclul Călătorii extraordinare
, cu ilustrații inspirate din
             ediția originală Hetzel, din secolul al XIX-lea.
`,
    autor: "Jules Verne",
    pret: 25.99
  },
  {id: 3,
    src: "https://via.placeholder.com/280",
    titlu: "Călătoarea",
    text: `Anul 1945. Claire Randall, fosta sora medicala, se
intoarce din razboi și pleacă
           împreună cu soțul într-o a doua luna de miere.`,
    autor: "Diana Gabaldon",
    pret: 52.50
  },
  {id: 4,
    src: "https://via.placeholder.com/280",
    titlu: "Memoriile lui Sherlock Holmes",
    text: `A doua colecție de povestiri polițiste scrise de Si
r Arthur Conan Doyle și
         reunește unsprezece dintre cele mai faimoase cazuri
ale legendarului detectiv.`,
    autor: "Sir Arthur Conan Doyle",
    pret: 17.43
  }
];
```

```
const App = () => {
  return (
    <>
      <Container>
        <h1>Carti pentru copii</h1>
      </Container>
      <ListaCarti listaCarti={lista} />
    </>
  );
}

export default App;
```

Rezultat:



Observație: În React, componentele individuale dintr-un șir de componente de același fel (generate de regulă folosind `map()`) sunt identificate prin valoarea unei chei (valoarea proprietății `key`, cuvânt rezervat). Când `map()` generează un șir de componente (`<Carte />` de exemplu), proprietatea `key` se include obligatoriu în fiecare componentă generată. Nu și în componentele interioare acestora, dacă ele există. Proprietatea `key` trebuie să aibă valori distincte pentru fiecare dintre componentele construite de `map()`. Mai mult, pentru fiecare componentă din listă valoarea cheii trebuie să fie legată de acea componentă. Astfel dacă interacțiunea utilizatorului cu

aplicația conduce la ștergerea unor componente din listă, valorile cheilor componentelor rămase trebuie să rămână valabile.

Această regulă este ușor de satisfăcut deoarece aplicațiile de tipul celei realizate preiau de obicei datele dintr-o bază de date. Atunci proprietatea *key* poate fi inițializată preluând valoarea cheii primare din tabelul care a furnizat datele.

În secvența de creare a listei de cărți de exemplu, pentru fiecare componentă *<Col />* care conține o componentă *<Carte />* s-a impuns o valoare distinctă pentru *key* folosind proprietatea *id* a elementului corespunzător din lista dată.

```
  <Col key={item.id}>
```

Această regulă din React derivă din necesitatea identificării precise a componentelor care trebuie reafișate ca urmare a modificării valorilor unor proprietăți. Valorile cheilor fiind unice, React va putea identifica acele componente din *DOM* care sunt afectate de modificarea operată și trebuie reafișate.

Inserarea imaginilor

Pentru a insera și imaginile coperților cărților din șirul de componente, acestea ar putea fi memorate în directorul *imagini*, derivat din directorul care conține *index.html* (directorul *public*), ca în figura 3.3.

Conținutul variabilei *lista* din componenta *<App />* va trebui modificat astfel:

```
const lista = [
  {id: 1,
    src: "imagini/treimuschetari.png",
    titlu: "Cei trei muschetari",
    text: `Tânărul d'Artagnan sosește la Paris pentru a se alăt
ura vestitului corp de
             gardă al regelui.`,
    autor: "Alexandre Dumas",
    pret: 20.90
  },
  {id: 2,
    src: "imagini/capitan.png",
    titlu: "Căpitan la cincisprezece ani",
```

```
    text: `O carte specială din ciclul Călătorii extraordinare
, cu ilustrații inspirate din
             ediția originală Hetzel, din secolul al XIX-lea.
`,
    autor: "Jules Verne",
    pret: 25.99
  },
  {id: 3,
    src: "imagini/calatoarea.png",
    titlu: "Călătoarea",
    text: `Anul 1945. Claire Randall, fosta sora medicala, se
intoarce din razboi și pleacă
            împreună cu soțul într-o a doua luna de miere.`,
    autor: "Diana Gabaldon",
    pret: 52.50
  },
  {id: 4,
    src: "imagini/holmes.png",
    titlu: "Memoriile lui Sherlock Holmes",
    text: `A doua colecție de povestiri polițiste scrise de Si
r Arthur Conan Doyle și
          reunește unsprezece dintre cele mai faimoase cazuri
ale legendarului detectiv.`,
    autor: "Sir Arthur Conan Doyle",
    pret: 17.43
  }
];
```

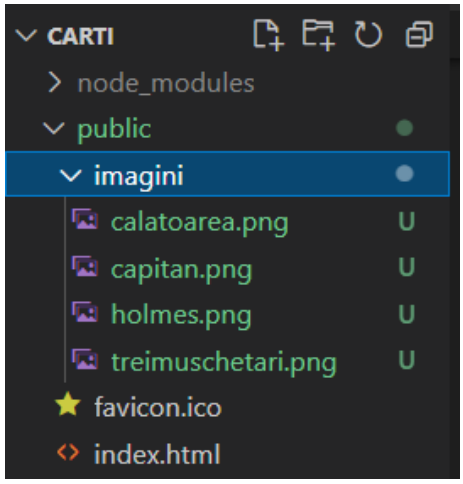
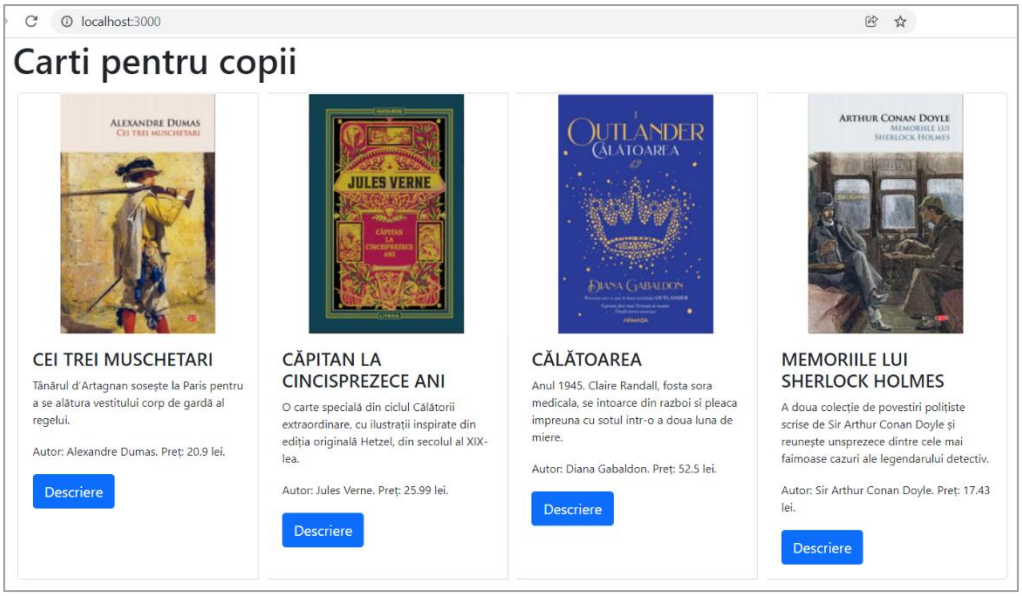


Figura 3.3. Directorul *public/imagini*

Rezultat:



Formulare. Componenta `<Adaug />`

În varianta actuală, aplicația `carti` afișează o listă de cărți. Șirul de obiecte care conțin datele acestora este însă fix, predefinit.

În toate aplicațiile web există însă ceea ce se numește o "*sursă de date*". Aceasta poate fi o bază de date, zona de memorare a browserului destinată păstrării unor date ale aplicațiilor (denumită `localStorage`) sau chiar aplicația scrisă, care poate conține formulare (elemente HTML `<form>`) destinate adăugării, editării sau ștergerii datelor.

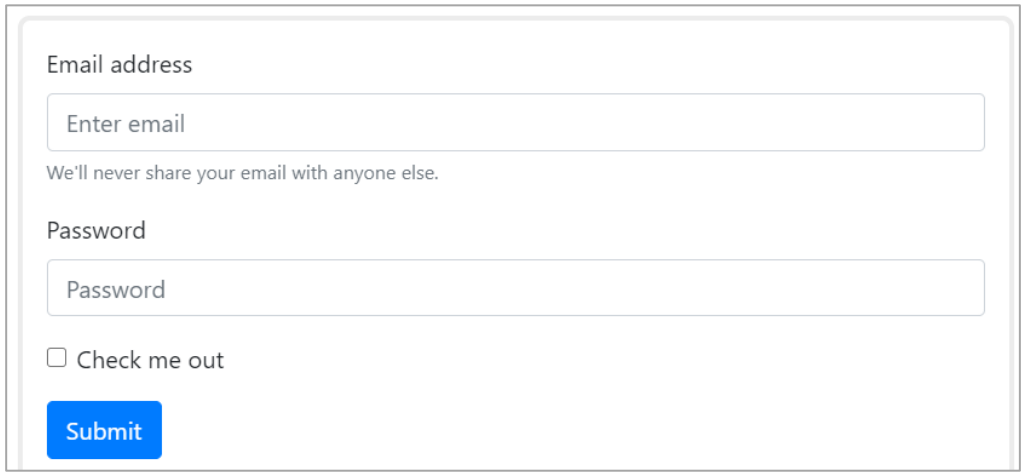
În continuare va fi realizat un formular destinat introducerii în lista de cărți a unui nou element. În capitolele următoare vor fi utilizate și alte surse de date dintre cele menționate.

Pentru a obține o versiune statică, fără date, a componentei denumită `<Adaug />` se va porni de la variantele de formulare propuse în cadrul colecției de componente React prestilizate, `React-Bootstrap`. Exemplu:

```
<Form>
  <Form.Group className="mb-3" controlId="formBasicEmail">
    <Form.Label>Email address</Form.Label>
```

```
    <Form.Control type="email" placeholder="Enter email" />
    <Form.Text className="text-muted">
      We'll never share your email with anyone else.
    </Form.Text>
  </Form.Group>

  <Form.Group className="mb-3" controlId="formBasicPassword">
    <Form.Label>Password</Form.Label>
    <Form.Control type="password" placeholder="Password" />
  </Form.Group>

  <Form.Group className="mb-3" controlId="formBasicCheckbox">
    <Form.Check type="checkbox" label="Check me out" />
  </Form.Group>

  <Button variant="primary" type="submit">
    Submit
  </Button>
</Form>
```

Codul JSX din exemplul dat afișează următorul formular:



Pentru a obține un formular adaptat aplicației, va trebui, pe modelul de mai sus, create patru controale Windows (componente *<Form.control />* de tip casetă de text). Pornind de la structura unui obiect din lista de cărți:

```
  {
    id: 1,
    src: "imagini/treimuschetari.png",
    titlu: "Cei trei muschetari",
    text: `Tânărul d’Artagnan sosește la Paris pentru a se ală
tura vestitului corp de gardă al regelui.`,
    autor: "Alexandre Dumas",
    pret: 20.90
  }
```

casetele de text vor permite introducerea datelor pentru câmpurile *src*, *titlu*, *text*, *autor* și *pret*.

Pentru a descrie o astfel de casetă de text, soluția oferită de extensia *React-Bootstrap* este următoarea:

```
   <Form.Group className="mb-3">
         <Form.Label>Titlul:</Form.Label>
         <Form.Control type="text" />
   </Form.Group>
```

Codul precedent produce următoarea reprezentare grafică:

Pentru câmpul denumit *text* din structura unui obiect din șirul de obiecte conținând datele cărților este însă mai potrivit un control de tip *textarea*. Acesta permite introducerea unui text mai lung, pentru care sunt necesare mai multe rânduri.

Exemplu de codificare pentru întregul formular:

```
  . . .
  const stil = {
        marginTop: "2rem",
        backgroundColor: "#ddd",
        padding: "20px",
        width: "750px",
    }

    return (
```

```
      <Container style={stil}>
        <Form>
          <Form.Group className="mb-3">
            <Form.Label>Titlul cărții:</Form.Label>
            <Form.Control type="text" />
          </Form.Group>

          <Form.Group className="mb-3">
            <Form.Label>Fișierul imagine:</Form.Label>
            <Form.Control type="text" />
          </Form.Group>

          <Form.Group className="mb-3">
            <Form.Label>Descrierea cărții:</Form.Label>
            <Form.Control as="textarea" rows={3} />
          </Form.Group>

          <Form.Group className="mb-3">
            <Form.Label>Autor:</Form.Label>
            <Form.Control type="text" />
          </Form.Group>

          <Form.Group className="mb-3">
            <Form.Label>Preț:</Form.Label>
            <Form.Control type="text" />
          </Form.Group>

          <Button variant="primary" type="submit">
                Memorează
          </Button>
        </Form>
      </Container>
    );
```

În codul scris au fost evidențiate controalele Windows destinate introducerii informațiilor.

Rezultatul utilizării acestei secvențe de cod este prezentat în figura 3.4.

Obiectul *state*

În acest moment se poate trece la etapa următoare a dezvoltării componentei *<Adaug />*, respectiv preluarea informațiilor din câmpurile formularului, crearea unui obiect care păstrează datele unei cărți și adăugarea acestui obiect în șirul de obiecte conținând informațiile despre cărți.

Până în acest moment datele afișate de aplicație au fost incluse într-un *șir de obiecte JavaScript invariabil*. Din acest moment ne punem însă problema adăugării în șirul de obiecte a unor noi elamante.

Figura 3.4. Interfața expusă de componenta *<Adaug />*

În React componenta variabilă a setului de date gestionate de o aplicație este păstrată într-un obiect destinat acestui scop, denumit *state*. În aplicațiile de dimensiuni reduse acest obiect este definit (și modificat!) în componenta *<App />*.

O primă familiarizare cu acest obiect se poate realiza însă scriind o componentă React care preia date dintr-un formular, de exemplu *<Adaug />*. Aceasta deoarece în astfel de componente se declară un obiect *state* care păstrează datele introduse în câmpurile formularului.

În *<Adaug />* obiectul *state* va avea cinci câmpuri: *src*, *titlu*, *text*, *autor* și *pret*. Se observă corespondența dintre numele câmpurilor obiectului *state* și cele ale unui obiect din șirul de obiecte care conțin datele cărților. Ea nu este deloc întâmplătoare.

Pentru ca aceste variabile să fie componente ale obiectului *state* aparținând componentei *<Adaug />*, acestea vor fi declarate folosind sintaxa:

```
import React, { useState } from "react";
. . .

    const [src, setSrc] = useState("");
    const [titlu, setTitlu] = useState("");
    const [text, setText] = useState("");
    const [autor, setAutor] = useState("");
    const [pret, setPret] = useState("");
```

În expresia:

```
  const [titlu, setTitlu] = useState("");
```

titlu este numele variabilei care se declară (și care va deveni astfel componentă a obiectului *state*), *setTitlu* este denumirea funcției care va fi apelată pentru a impune variabilei *titlu* o nouă valoare, iar *useState()* este o funcție importată din modulul "*react*". Funcția *useState()* admite un parametru, acesta fiind folosit pentru a impune valoarea inițială a variabilei *titlu*.

Observație: Modificarea valorilor câmpurilor obiectului *state* se realizează exclusiv folosind funcțiile declarate în acest scop. De exemplu, pentru a reinițializa variabila *titlu* va trebui apelată funcția *setTitlu*, argumentul funcției în momentul apelului fiind noua valoare a acesteia.

Transferul valorii dintr-un control al formularului (o casetă de text de exemplu) în variabila asociată din *state* este bazat pe tratarea evenimentului *change* declanșat de modificarea informației din control:

```
  <Form.Group className="mb-3">
    <Form.Label>Titlul:</Form.Label>
    <Form.Control type="text"
          value={titlu}
          onChange={evt => setTitlu(evt.target.value)} />
  </Form.Group>
```

Observații:

Valoarea din controlul formularului (impusă prin atributul `value`) se ia obligatoriu din variabila corespunzătoare din `state`. În consecință, pentru controlul destinat inițializării variabilei titlu, `value={titlu}`.

Funcția apelată automat la modificarea valorii controlului se va scrie ca în exemplul dat. Ea apelează funcția `setTitlu()`, destinată modificării variabilei `titlu` (o proprietate din `state`).

Argumentul funției `setTitlu` este `evt.target.value`. În această expresie, `evt` este un obiect JavaScript de tip `event` iar `evt.target` este câmpul obiectului `evt` care conține adresa obiectului care a declanșat evenimentul `change`.

Componentelor React folosite la descrierea formularului le-au fost adăugate atribute suplimentare (`value` și `onChange`), care nu sunt prezente în varianta statică a formularului prezentată anterior. Acest lucru este posibil deoarece, în toate colecțiile de componente prestilizate, atributele normale ale acestora, așa cum apar ele în HTML5, sunt automat permise și produc aceleași efecte. Asfel de exemplu componentei `<Form />` i se va adăuga atributul `onSubmit`, efectul fiind cel cunoscut de la elementul HTML5 `<form>`, respectiv asocierea unei funcții de tratare a evenimentului `submit` declanșat de selectarea butonului de tip `submit` al formularului.

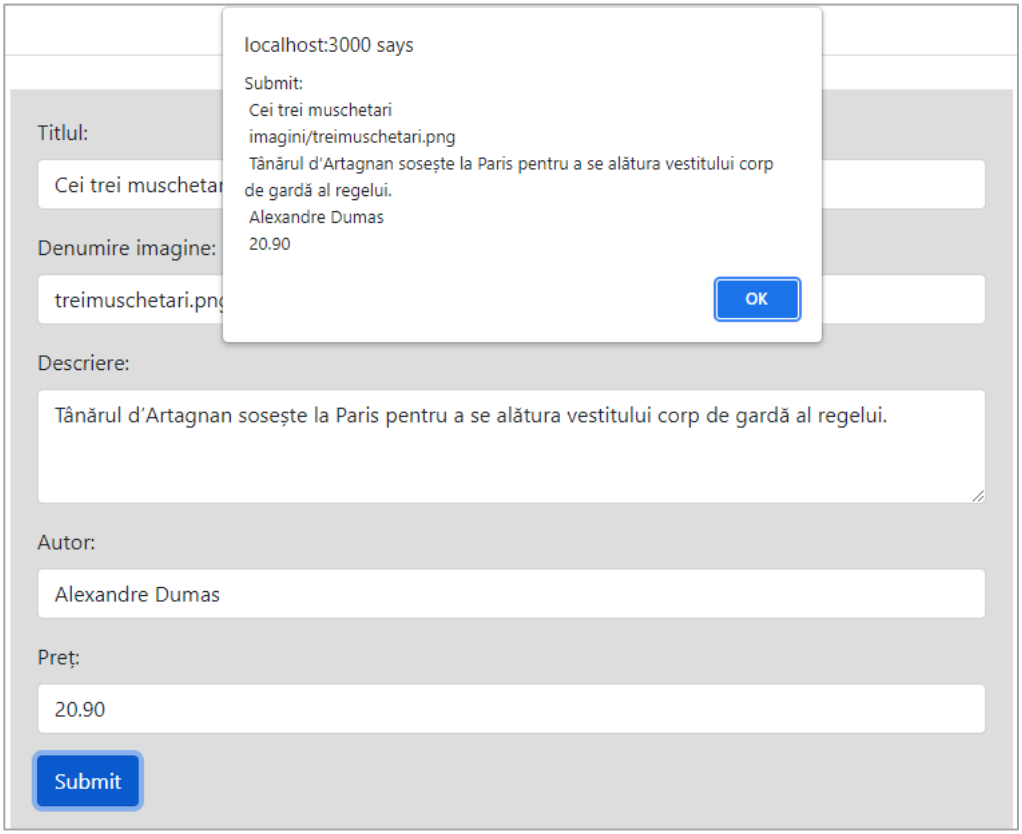
În acest moment, componenta `<Adaug />` are următorul conținut:

```
import React, { useState } from "react";
import { Form, Container, Button } from 'react-bootstrap';

const Adaug = (props) => {
    const [src, setSrc] = useState("");
    const [titlu, setTitlu] = useState("");
    const [text, setText] = useState("");
    const [autor, setAutor] = useState("");
    const [pret, setPret] = useState("");

    const tratezSubmit = (evt) => {
      evt.preventDefault();
      alert(`Submit:\n ${titlu}\n imagini/${src}\n ${text}\n $
{autor}\n ${pret}`);
    }

    const stil = {
        marginTop: "2rem",
        backgroundColor: "#ddd",
        padding: "20px",
```

```
        width: "750px",
    }

    return (
    <Container style={stil}>
        <Form onSubmit={tratezSubmit}>

          <Form.Group className="mb-3">
            <Form.Label>Titlul:</Form.Label>
            <Form.Control type="text" value={titlu}
                    onChange={e => setTitlu(e.target.value)} /
>
          </Form.Group>

          <Form.Group className="mb-3">
              <Form.Label>Denumire imagine:</Form.Label>
              <Form.Control type="text" value={src}
                      onChange={e => setSrc(e.target.value)} /
>
          </Form.Group>

          <Form.Group className="mb-3">
              <Form.Label>Descriere:</Form.Label>
              <Form.Control as="textarea" rows={3} value={text
}
                      onChange={e => setText(e.target.value)}
/>
          </Form.Group>

          <Form.Group className="mb-3">
              <Form.Label>Autor:</Form.Label>
              <Form.Control type="text" value={autor}
                      onChange={e => setAutor(e.target.value)}
/>
          </Form.Group>

          <Form.Group className="mb-3">
              <Form.Label>Preț:</Form.Label>
              <Form.Control type="text" value={pret}
                      onChange={e => setPret(e.target.value)}
/>
          </Form.Group>

          <Button variant="primary" type="submit">
              Submit
          </Button>

        </Form>
```

```
    </Container>
    );
}

export default Adaug;
```

În această variantă de dezvoltare a componentei, funcția de tratare a evenimentului `submit` realizează doar crearea unei ferestre `Alert` în care afișează valorile variabilelor din `state` (`src`, `titlu`, `text`, `autor` și `pret`).

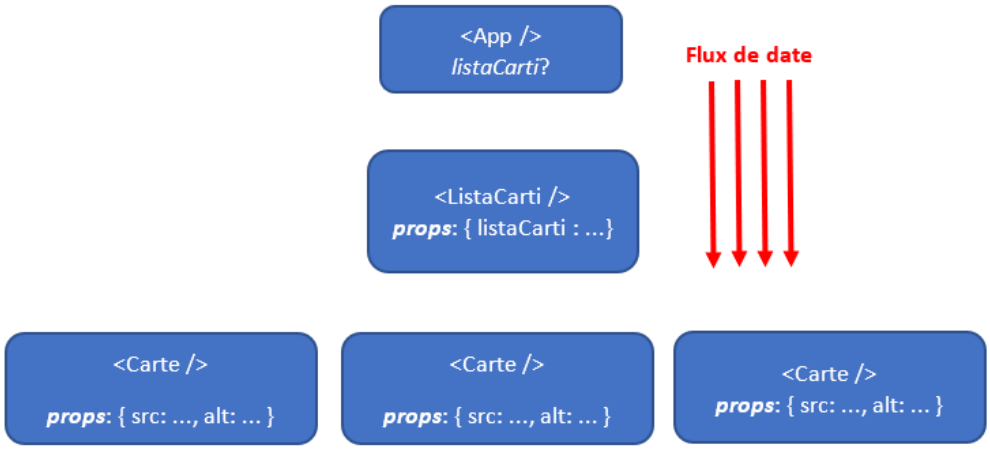


Trimiterea spre *<App />* a obiectului *state* din *<Adaug />*

În aplicația React scrisă, fiind de complexitate relativ redusă, datele specifice sunt păstrate în obiectul `state` al componentei `<App />`. De altfel și în varianta în care lista cărților era invariabilă, ea era declarată tot în `<App />`. De aici era trimisă componentei `<ListaCarti />` pentru a fi

create componentele `<Carte />` necesare afișării pe ecran a cărților din listă.

În acest mod se implementează cerința specifică aplicațiilor React menționată deja, respectiv circulația informației într-un singur sens, de la componenta principală `<App />` spre componentele destinate afișării, `<Carte />` (componente "`frunză`" ale structurii arborescente de componente React).



În acest moment va trebui însă realizat mecanismul de trimitere a obiectului `state` existent în componenta `<Adaug />` spre `<App />`.

Aparent avem o problemă, deoarece trebuie să trimitem date în sens invers sensului normal, respectiv dinspre o componentă "frunză" (`<Adaug />`) spre componenta "rădăcină" (`<App />`). Doar aparent, deoarece nimeni nu ne poate opri să transmitem componentei `<Adaug />`, prin `props`, denumirea unei funcții din `<App />` care realizează inserarea în lista de cărți a un obiect suplimentar. Iar din `<Adaug />`, mai precis din funcția `tratezSubmit()` putem apela funcția din `<App />` trimițându-i ca parametru efectiv obiectul care trebuie inserat în lista cărților. Practic, procedând astfel, nu facem apel la vreunul dintre mecanismele de transfer existente în React ci creăm un mecanism propriu.

Concret, în `<Adaug />`, folosind valorile din câmpurile obiectului `state`, în funcția `tratezSubmit()` va fi creat un obiect denumit `carte` și apoi transmis spre `<App />` prin apelul funcției `transmit` (denumire primită prin `props`). Adică:

```
  const Adaug = (props) => {
    const [src, setSrc] = useState("");
    const [titlu, setTitlu] = useState("");
    const [text, setText] = useState("");
    const [autpret, setAutpret] = useState("");

    const tratezSubmit = (evt) => {
      evt.preventDefault();
      const carte = {
        src: `imagini/${src}`,
        titlu,
        text,
        autor,
        pret
      };
      props.transmit(carte);  //  Transmit spre <App /> ob. ca
rte
      //  Apoi golesc controalele formularului
      setSrc("");
      setTitlu("");
      setText("");
      setAutor("");
      setPret("");
    }
    . . .
```

Mai trebuie modificată doar componenta *<App />*. Trebuie să i se adauge un obiect *state* cu un singur câmp, *lista*, care va fi inițializat cu un șir de valori vid:

```
  const [lista, setLista] = useState([]);
```

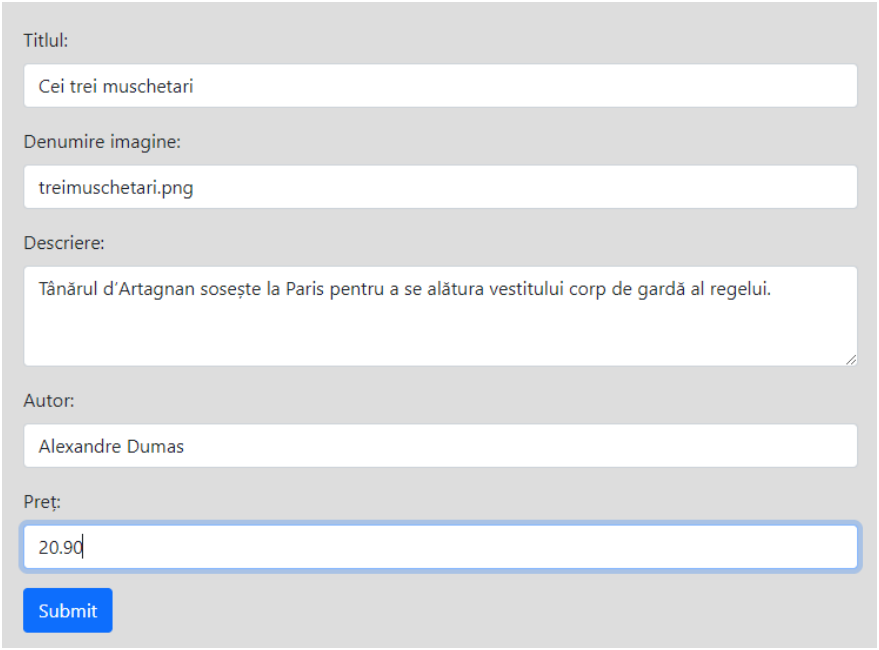
De asemenea trebuie adăugată componentei *<App />* funcția care va fi apelată din *<Adaug />*. Această funcție va adăuga obiectului primit încă un câmp, *id*, și apoi va adăuga obiectul astfel obținut în șirul de obiecte (variabila de stare) *lista*. Într-o primă variantă, valoarea câmpului *id* se va stabili folosind numărul actual de elemente din variabila *lista*.

```
import React, { useState } from "react";
import ListaCarti from "./listacarti.js";
import Adaug from "./adaug";

const App = () => {
  const [lista, setLista] = useState([]);
```

```
  const adaug = (carte) => {
    carte.id = lista.length + 1;
    setLista([...lista, carte]);
  }

  return (
    <>
      <Container>
        <h1>Carti pentru copii</h1>
      </Container>
      <ListaCarti listaCarti={lista} />
      <Container>
        <Adaug transmit="adaug" />
      </Container>
    </>
  );
}

export default App;
```

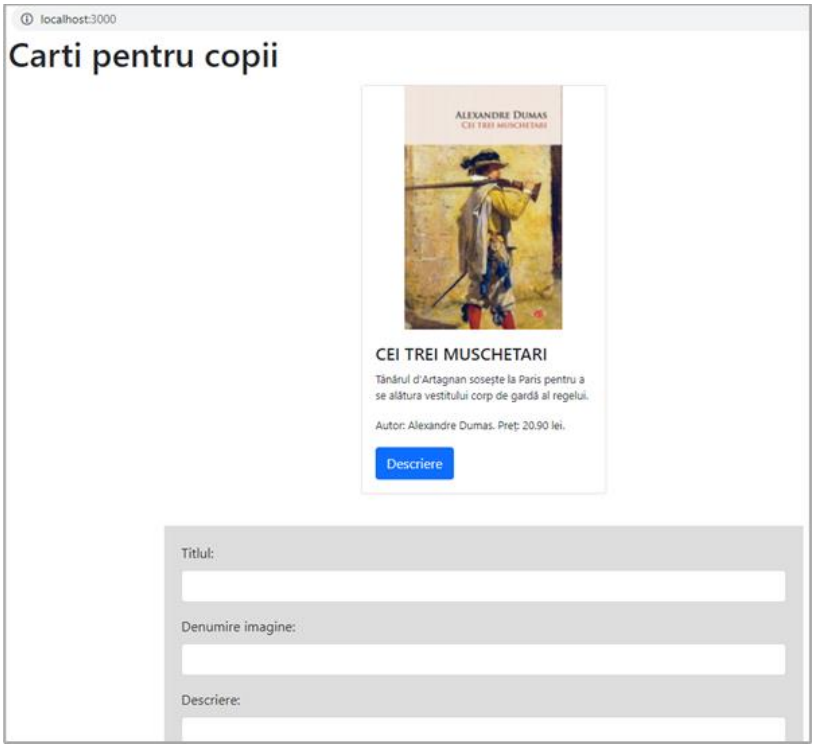
Exemplu de utilizare a aplicației:

1. Încărcarea în formular a datelor unei cărți:



2. Adăugarea unui element în variabila `lista` din `<App />` urmată de golirea formularului.

Rezultat:



Observație: Modificarea variabilei `lista` s-a realizat folosind operatorul specific din JavaScript ES6 (*spread operator*).

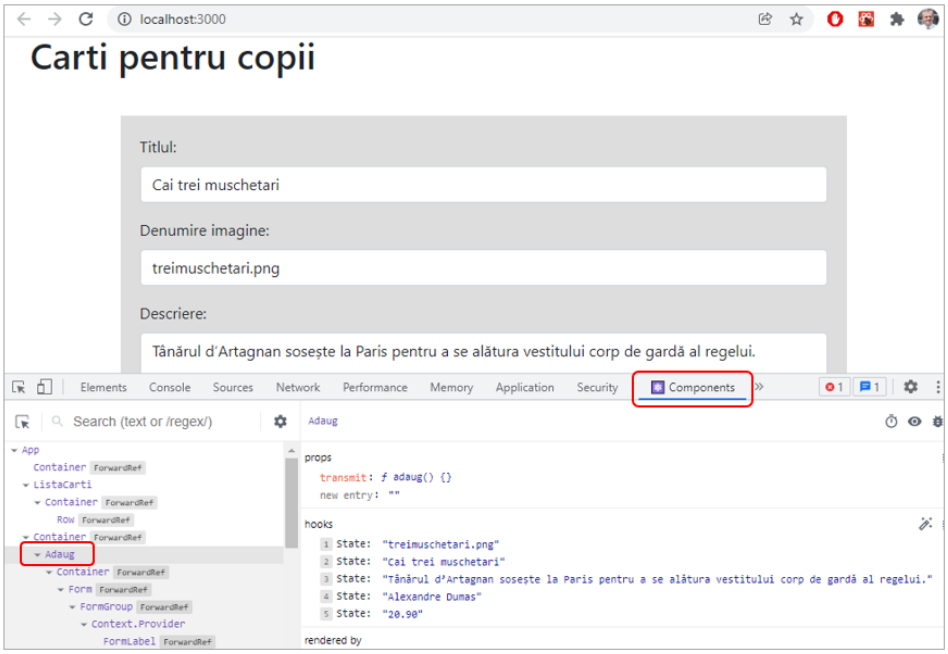
```
setLista([...lista, carte]);
```

React Developer Tools

Pentru afișarea dinamică a datelor conținute în obiectele *props* și *state* aparținând componentelor React, browserului *Google Chrome* i se poate adăuga extensia *React Developer Tools*.

Odată adăugată această extensie, pentru toate paginile Web care folosesc componente React, pe bara din partea de sus a browserului se va activa o pictogramă specifică:

De asemenea, dacă se selectează în meniul contextual *Inspect Elements* și apoi, în panoul afișat, tabul *Components*, se va afișa structura arborescentă a componentelor React din pagină iar, pentru componenta React selectată în arbore, se vor afișa obiectele *props* și *state*.



Utilizarea altor tipuri de controale Windows

În formularul afișat de componenta *<Adaug />* s-au folosit componente *<Form.Control />* de tip "*text*" și "*textarea*".

În continuare se va realiza o mică aplicație React care va afișa un formular în care vor fi utilizate încă cinci tipuri de controale: "*email*", "*password*", "*checkbox*", "*radio*" și "*select*".

Ca și în cazul aplicației *Carti*, crearea noii aplicații se va realiza utilizând scriptul *create-react-app*:

```
  npx create-react-app reactform
```

Apoi se vor adăuga aplicației extensiile *React-Bootstrap* și *Bootstrap* și fișierele *index.js* și *App.js* vor fi modificate, în final acestea având următorul conținut:

index.js

```
import React from 'react';
import { render } from 'react-dom';
import "bootstrap/dist/css/bootstrap.min.css";
import App from './App';

render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
  document.getElementById('root')
);
```

App.js

```
import React from 'react';
import Container from "react-bootstrap/Container";
import Formular from './formular';

function App() {
  return (
    <Container>
       <h1>Formular React</h1>
       <Formular />
    </Container>
  );
}
```

```
export default App;
```

Se observă că în *<App />* se afișează o componentă denumită *<Formular />*. Într-o primă formă, conținutul acesteia este următorul:

formular.js

```
import React, { useState } from "react";
import { Form, Button } from 'react-bootstrap';
const Formular = () => {
    const [email, setEmail] = useState("");
    const [pass, setPass] = useState("");
    const [check, setCheck] = useState(false);
    const [skill, setSkill] = useState("Frontend");
    const [lang, setLang] = useState(0);

    return (
        <Form>
            <Form.Group className="mb-3" controlId="formBasicE
mail">
                <Form.Label>Email address</Form.Label>
                <Form.Control type="email" placeholder="Enter
email" value={email} onChange={ev => setEmail(ev.target.value)
} />
                <Form.Text className="text-muted">
                We'll never share your email with anyone else.
                </Form.Text>
            </Form.Group>
            <Form.Group className="mb-3" controlId="formBasicP
assword">
                <Form.Label>Password</Form.Label>
                <Form.Control type="password" placeholder="Pas
sword" value={pass} onChange={ev => setPass(ev.target.value)}
/>
            </Form.Group>
            <Form.Group className="mb-3" controlId="formBasicC
heckbox">
                <Form.Check type="checkbox" checked={check} on
Change={() => setCheck(!check)} label="Check me out" />
            </Form.Group>

            <Form.Group>
                <Form.Label>
                    Skill:
                </Form.Label>
```

```
                <Form.Check type="radio" label="Front-end" nam
e="radiob" checked={skill === 'Frontend'} onChange={() => setS
kill('Frontend')} />
                <Form.Check type="radio" label="Back-end" name
="radiob" checked={skill === 'Backend'} onChange={() => setSki
ll('Backend')} />
                <Form.Check type="radio" label="Full-stack" na
me="radiob" checked={skill === 'Fullstack'} onChange={() => se
tSkill('Fullstack')} />
            </Form.Group>

            <Form.Group>
                <Form.Label>
                    Main programming language:
                </Form.Label>
                <Form.Control as="select" value={lang} onChang
e={ev => setLang(ev.target.value)}>
                    <option value="0">Choose...</option>
                    <option value="1">C++</option>
                    <option value="2">JavaScript</option>
                    <option value="3">C#</option>
                    <option value="4">Java</option>
                    <option value="5">PHP</option>
                </Form.Control>
            </Form.Group>

            <Button variant="primary" type="submit">
                Submit
            </Button>
        </Form>
    );
}

export default Formular;
```

Dacă se rulează aplicația, ea va afișa imaginea din figura 3.5:

Elementele de noutate vin, desigur, dinspre controalele din tipurile menționate.

1. **Controlul de tip "*checkbox*"**

La declararea obiectului *state*, pentru a gestiona controlul de tip "*checkbox*" au fost declarate variabila *check* și funcția *setCheck*:

```
    const [check, setCheck] = useState(false);
```

Controlul de tip "*checkbox*" a fost inserat în formular astfel:

```
<Form.Group className="mb-3" controlId="formBasicCheckbox">
    <Form.Check type="checkbox" checked={check} onChange={() =
> setCheck(!check)} label="Check me out" />
</Form.Group>
```

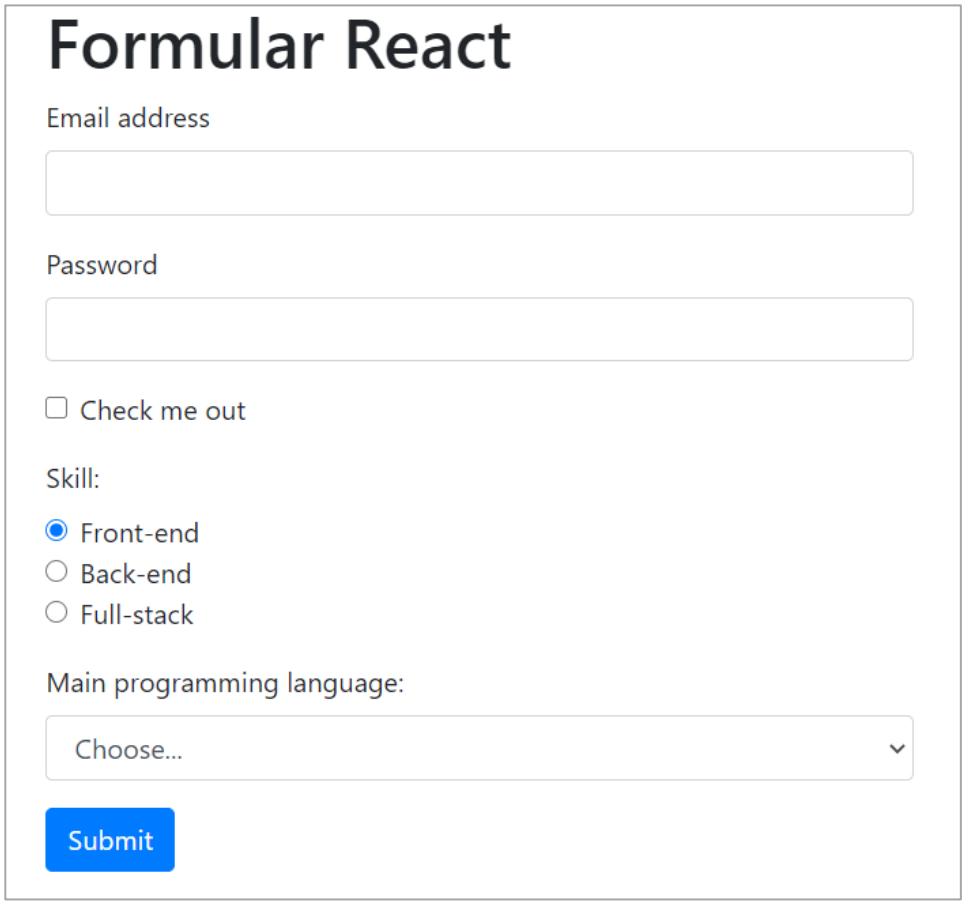


Figura 3.5. Formularul afișat

Porțiunea de cod evidențiată permite impunerea valorii controlului (realizată folosind proprietatea *checked={check}*) respectiv modificarea variabilei logice *check* din *state*. Pentru a schimba valoarea variabilei *check* în momentul selectării cu mausul a controlului s-a utilizat proprietatea *onChange*:

```
   onChange={() => setCheck(!check)}
```

Astfel apelată, funcția `setCheck()` atribuie variabilei logice `check` valoarea `!check`.

2. **Grupul de butoane de tip "`radio`"**

La declararea obiectului `state`, pentru a gestiona grupul de butoane "`radio`" din formular au fost declarate variabila `skill` și funcția `setSkill`:

```
        const [skill, setSkill] = useState("Frontend");
```

Grupul de butoane radio a fost inserat în formular astfel:

```
<Form.Group>
    <Form.Label>Skill:</Form.Label>
    <Form.Check type="radio" label="Front-end" name="radiob"
                  checked={skill === 'Frontend'} onChange={()
=> setSkill('Frontend')} />
    <Form.Check type="radio" label="Back-end" name="radiob"
                  checked={skill === 'Backend'} onChange={() =
> setSkill('Backend')} />
    <Form.Check type="radio" label="Full-stack" name="radiob"
                   checked={skill === 'Fullstack'} onChange={(
) => setSkill('Fullstack')} />
</Form.Group>
```

Porțiunea de cod evidențiată conține atribuirile valorilor proprietăților `name`, `checked` și `onChange`.

- `name` – este un o proprietate care trebuie să primească aceeași valoare pentru toate controalele de tip "`radio`" care formează grupul;
- `checked` – determină afișarea ca selectat a unui buton radio în momentul selectării cu mausul. Pentru al doilea buton de exemplu, proprietatea este inițializată astfel:

```
   checked={skill === 'Backend'}
```

- `onChange` – impune variabilei `skill` o nouă valoare în momentul selectării controlului cu mausul. Pentru al doilea buton de exemplu, apelarea funcției `setSkill()` se realizează astfel:

```
   onChange={() => setSkill('Backend')}
```

- *value* – este un parametru care trebuie să primească aceeași valoare pentru toate controalele de tip "*radio*" care formează grupul;

```
  value={lang}
    checked={skill === 'Backend'}
    onChange={() => setSkill('Backend')}
```

Observație: De regulă, când se utilizează un grup de butoane de tip radio, unul dintre butoane trebuie să fie obligatoriu selectat. În codul formularului, se observă că variabila *skill* din *state* a primit o valoare inițială care corespunde uneia dintre valorile testate în continuare pentru a se afișa starea fiecărui buton (*selectat / neselectat*):

```
    const [skill, setSkill] = useState("Frontend");
```

3. Controlul de tip "*select*"

La declararea obiectului *state*, pentru controlul de tip "*select*" au fost declarate variabila *lang* și funcția *setLang*:

```
    const [lang, setLang] = useState(0);
```

Controlul de tip "*select*" a fost inserat în formular astfel:

```
<Form.Group>
   <Form.Label>Main programming language:</Form.Label>
   <Form.Control as="select" value={lang} onChange={ev => setL
ang(ev.target.value)}>
        <option value="0">Choose...</option>
        <option value="1">C++</option>
        <option value="2">JavaScript</option>
        <option value="3">C#</option>
        <option value="4">Java</option>
        <option value="5">PHP</option>
    </Form.Control>
</Form.Group>
```

Porțiunea de cod evidențiată conține atribuirile valorilor proprietăților *value* și *onChange*. Proprietatea *value* primește valoarea variabilei

`lang`. Proprietatea `onChange` permite declanșarea unei acțiuni în momentul selectării unei alte opțiuni.

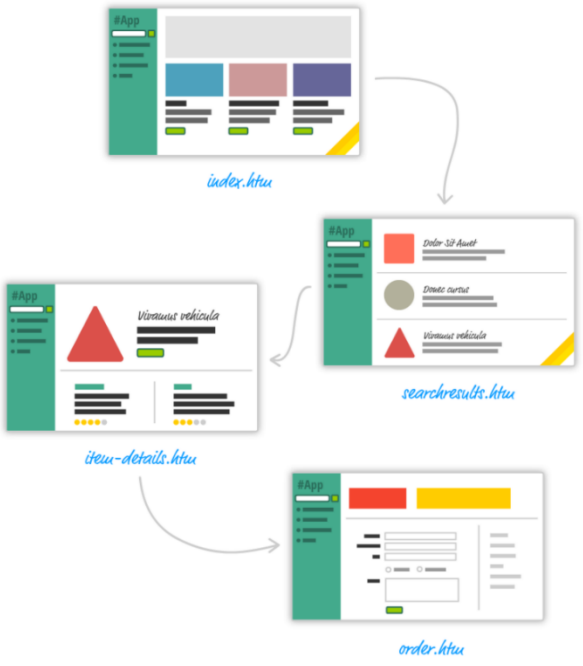
```
    onChange={ev => setLang(ev.target.value)}
```

React Router

Generalități

Tastarea unei adrese în caseta pentru adrese a browserului provoacă trimiterea în Internet a unei cereri de descărcare a unei resurse. Dacă adresa introdusă corespunde adresei unei pagini web, serverul care o găzduiește o va transfera spre browser. Pentru fiecare adresă scrisă în browser va fi transferată și afișată o nouă pagină.

Schema prezentată în continuare prezintă acest proces (sursa: *https://www.kirupa.com/react/introducing_react.htm*).



Aplicațiile React necesită o abordare diferită deoarece sunt conținute într-un singur fișier, `index.html`. În cazul lor schimbarea conținutului afișat pe ecran se realizează mai simplu, prin înlocuirea componentei care ocupă o anumită zonă a ferestrei browserului.

Procesul de înlocuire a unei componente cu o altă componentă este declanșat prin schimbarea conținutului casetei de adrese a browserului.

Pentru a implementa acest mecanism, va trebui adăugat aplicației React un pachet de componente suplimentar, `react-router-dom`:

```
npm install react-router-dom@6
```

Apoi la începutul fișierelor JavaScript în care se folosesc componente din acest pachet va trebui adăugată instrucțiunea `import` prin care se inserează componente din pachetul `react-router-dom` utilizate. Cele mai utilizate componente sunt `<Route />`, `<Link />` și `<BrowserRouter />`.

Observație: `react-router-dom` este un pachet care se dezvoltă continuu. În cele ce urmează va fi utilizată versiunea 6.

Componentele *<BrowserRouter />* și *<Route />*

Implementarea navigării într-o aplicație web începe cu adăugarea unei componente specifice, `<BrowserRouter />`.

Rolul acestei componente este inițierea memorării în obiectul `history` al browserului a adreselor utilizate pentru afișarea diferitelor componente React din aplicație. Efectul memorării acestor adrese în `history` face posibilă încărcarea unei componente prin introducerea unei adrese în caseta de adrese a browserului sau revenirea la o adresă anterioară folosind butonul `back` al browserului. Utilizatorul aplicației va avea astfel impresia de navigare într-un ansamblu de pagini web deși, în realitate, va rămâne permanent în `index.html`.

Componenta `<BrowserRouter />` va fi inserată în fișierul `index.js`, astfel:

```
import React from 'react';
import { render } from 'react-dom';
import { BrowserRouter as Router } from "react-router-dom";
import "bootstrap/dist/css/bootstrap.min.css";
import App from './App';

render(
  <React.StrictMode>
    <Router>
      <App />
    </Router>
  </React.StrictMode>,
  document.getElementById("root")
);
```

Componentele `<Route />` sunt fundamentale în definirea componentelor afișate la un moment dat în fereastra browserului. O componentă `<Route />` conține două proprietăți:

1. Componenta care trebuie afișată, definită folosind proprietatea `element` și

2. Calea din caseta de adrese a browserului care, odată introdusă, va determina afișarea componentei. Calea este valoarea proprietății *path* a componentei *Route*.

Sintaxa unei componente *<Route />*:

```
<Route path="/despre" element={<Despre />} />
```

Destul de intuitiv. Dacă aplicația este afișată folosind miniserverul de dezvoltare, pentru a determina afișarea pe ecran a componentei *<Despre />*, în caseta de adrese a browserului trebuie să se tasteze adresa: *http://localhost:3000/despre*.

Porțiunea din adresă evidențiată indică adresa directorului rădăcină al site-ului. În timpul lucrului cu mediul de dezvoltare, această adresă se reprezintă printr-un caracter '*/*'.

În cazul în care componentei referite în *<Route />* trebuie să i se transmită date prin *props*, sintaxa se modifică astfel:

```
  <Route path="/activitate" element={<Activitate ora={oraIncep
} tip={tipActiv} />} />
```

Variabilele *oraIncep* și *tipActiv* au fost declarate și inițializate în prealabil, fiind de regulă componente ale obiectului *state*.

Aplicația *reactrouter*

Pentru a exemplifica modul de utilizare a componentelor *<BrowserRouter />*, *<Route />* și *<Link />* din pachetul *react-router-dom*, s-a realizat o mică aplicație denumită *reactrouter*.

Pentru început, componentei *<App />* i s-au adăugat componentele *<Home />*, *<Activitate />* și *<Despre />*.

home.js

```
import React from 'react';
```

```
const Home = () => {
    const stil = {
        borderBottom: "2px solid red",
        display: "inline-block",
        width: "25%",
      };
      return (
        <>
          <h3 className="mt-5">Home</h3>
          <div style={stil}></div>
          <p>Pagina <em>Home</em>.</p>
        </>
      );
}

export default Home;
```

activitate.js

```
import React from 'react';

const Activitate = ({ora, tip}) => {
  const stil = {
    borderBottom: "2px solid blue",
    display: "inline-block",
    width: "25%",
  };

  return (
    <>
      <h3 className="mt-5">Activitate</h3>
      <div style={stil}></div>
      <p>Pagina <em>Activitate</em>. Valorile primite: ora = {
ora} și tip = {tip}.</p>
    </>
  );
}

export default Activitate;
```

despre.js

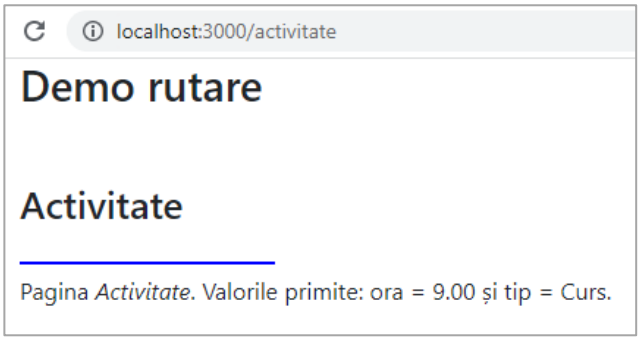
```
import React from 'react';
```

```
const Despre = () => {
    const stil = {
        borderBottom: "2px solid green",
        display: "inline-block",
        width: "25%",
      };
      return (
        <>
          <h3 className="mt-5">Despre</h3>
          <div style={stil}></div>
          <p>Pagina <em>Despre</em>.</p>
        </>
      );
}

export default Despre;
```

App.js

```
import React, { useState } from "react";
import Home from "./home";
import Despre from "./despre";
import Activitate from "./activitate";
import Container from "react-bootstrap/Container";
import { Routes, Route } from "react-router-dom";

function App() {

  const [oraInc, setOraInc] = useState("9.00");
  const [tipActiv, setTipActiv] = useState("Curs");

  return (
   <Container>
       <h2>Demo rutare</h2>
        <Routes>
           <Route path="/" element={<Home />} />
           <Route path="despre" element={<Despre />} />
          <Route path="/activitate" element={<Activitate ora={
oraInc} tip={tipActiv} />} />
        </Routes>
   </Container>  );
  }

export default App;
```

Rezultate:



Observații:

1. Componentele `<Route />` sunt obligatoriu grupate în interiorul unei componente `<Routes />`.
2. De regulă, în `<Routes />` se mai adaugă o componentă `<Route />` care va fi afișată în cazul în care adresa din caseta de

adrese a browserului este eronată, deci nu corespunde niciuneia dintre căile conținute în componentele *<Route />*. Dacă denumirea acesteia este *<RutaLipsa />*, aplicației i se va adăuga încă un fișier (*rutalipsa.js*) iar componenta *<App />* trebuie modificată.

rutalipsa.js

```
import React from 'react';

const RutaLipsa = () => {
    const stil = {
        borderBottom: "2px solid blue",
        display: "inline-block",
        width: "25%",
      };
      return (
        <>
          <h3 className="mt-5">Pagină inexistentă</h3>
          <div style={stil}></div>
          <p>
          Pagină inexistentă.
          </p>
        </>
      );
}

export default RutaLipsa;
```

Integrarea acestei noi componente presupune modificarea codului din *<App />* destinat descrierii rutelor, astfel:

```
        <Routes>
           <Route path="/" element={<Home />} />
           <Route path="despre" element={<Despre />} />
          <Route path="/activitate" element={<Activitate ora={
oraInc} tip={tipActiv} />} />
          <Route path="*" element={<RutaLipsa />} />
        </Routes>
```

Observație: Noua rută, activată în cazul introducerii unei adrese eronate, trebuie adăugată la sfârșitul șirului de componente *<Route />*.

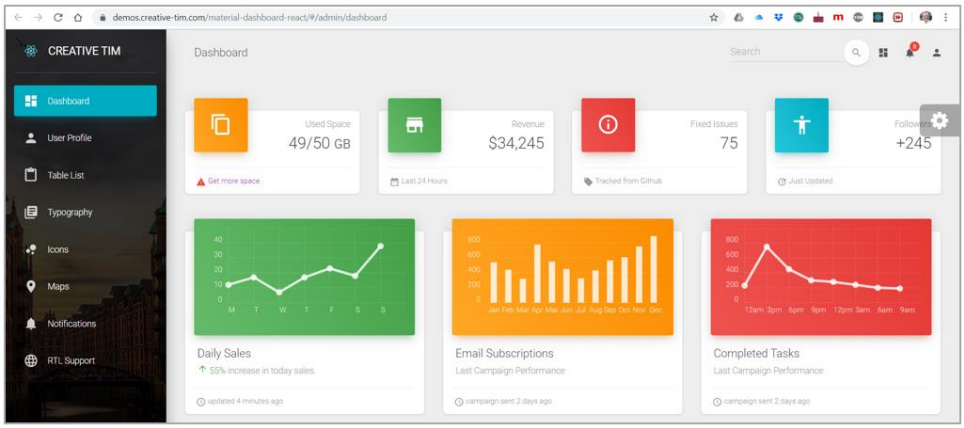
Componenta *<Link />*

În versiunea actuală a aplicației `reactrouter`, trecerea de la o pagină la alta presupune tastarea unei noi adrese în caseta de adrese a browserului (`http://localhost:3000/`, `http://localhost:3000/despre`, etc.). Destul de rudimentar, dar dacă trebuie să-i trimitem unui utilizator adresa unde se află o anumită informație, avem acum posibilitatea de a realiza acest lucru.

Este cunoscut însă faptul că, în aplicații web normale, pentru schimbarea conținutului afișat se folosesc elemente `<nav>` care conțin elemente `<a>`. Selectarea unui element `<a>` determină accesarea unui server de web și încărcarea unei noi pagini în fereastra browserului.

În React însă, mecanismul care trebuie creat este diferit. Componentele care trebuie afișate la selectarea unor link-uri sunt deja încărcate de pe server, acțiunea fiind realizată la citirea fișierului `index.html`. Aplicațiile React sunt de tip `Single Page Application` (prescurtat `SPA`). Mecanismul de rutare care va fi creat trebuie să asigure gestionarea afișajului și afișarea componentelor aferente unei anumite rute, nu încărcarea unei pagini web. Din acest motiv nu pot fi utilizate elemente `<a>`, acestea fiind înlocuite de componente specializate, `<Link />`. Ca și elementele `<a>`, acestea pot apărea și în alte părți ale aplicației, dar în `<nav>` sunt de neevitat.

Exemplu de aplicație web având un element `<nav>` plasat vertical, în stânga ecranului:



Pentru a implementa o astfel de soluție, pentru componenta *<nav>* se poate folosi o listă prestilizată (preluată din *React-Bootstrap*).

```
<ListGroup>
    <ListGroup.Item>Cras justo odio</ListGroup.Item>
    <ListGroup.Item>Dapibus ac facilisis in</ListGroup.Item>
    <ListGroup.Item>Morbi leo risus</ListGroup.Item>
    <ListGroup.Item>Porta ac consectetur ac</ListGroup.Item>
    <ListGroup.Item>Vestibulum at eros</ListGroup.Item>
  </ListGroup>
```

Interfața expusă:

Utilizând această soluție, lista de componente *<Link />* necesară afișării paginilor aplicației începute poate fi inserată într-o nouă componentă React, *<Navi />*:

navi.js

```
import React from "react";
import ListGroup from "react-bootstrap/ListGroup";
import Link from "react-router-dom";

const Navi = () => {
  const activ = { fontWeight: "bold", color: "red" };
  return (
    <ListGroup className="mt-5">
      <ListGroup.Item>
        <Link to="/">Home</Link>
      </ListGroup.Item>
      <ListGroup.Item>
        <Link to="/despre">Despre</Link>
```

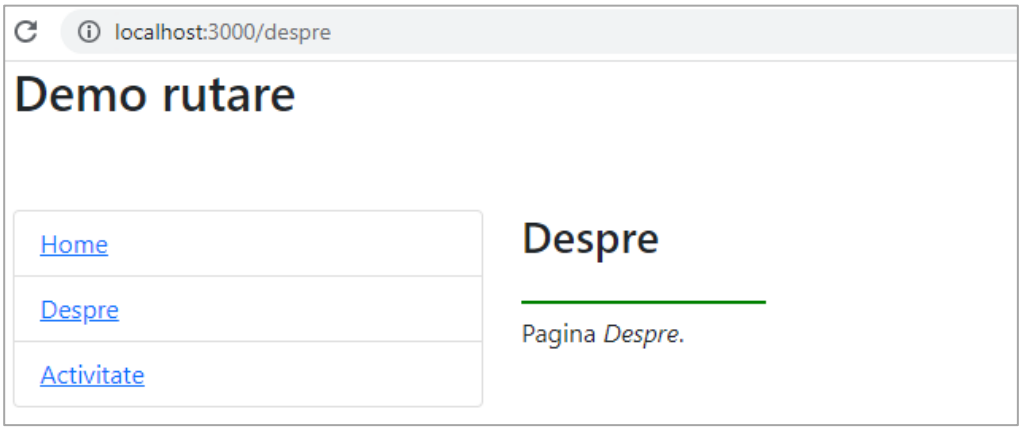
```
      </ListGroup.Item>
      <ListGroup.Item>
        <Link to="/activitate">Activitate</Link>
      </ListGroup.Item>
    </ListGroup>
  );
};

export default Navi;
```

Utilizând componenta *<Navi />* realizată, componenta *<App />* poate fi rescrisă astfel:

```
import React, { useState } from "react";
import Home from "./home";
import Despre from "./despre";
import Activitate from "./activitate";
import RutaLipsa from "./rutalipsa";
import Navi from "./navi";
import { Container, Row, Col } from "react-bootstrap";
import { Routes, Route } from "react-router-dom";

function App() {

  const [oraIncep, setOraIncep] = useState("9.00");
  const [tipActiv, setTipActiv] = useState("Curs");

  return (
      <Container>
          <h2>Demo rutare</h2>
          <Row>
            <Col sm={4}>
              <Navi />
            </Col>
            <Col sm={8}>
              <Routes>
                <Route path="/" element={<Home />} />
                <Route path="despre" element={<Despre />} />
                <Route path="/activitate" element={<Activitate
ora={oraIncep} tip={tipActiv} />} />
                <Route path="*" element={<RutaLipsa />} />
              </Routes>
            </Col>
          </Row>
      </Container>
```

```
  );
}

export default App;
```

Rezultat:



Transmiterea unor parametri

În multe situații componenta referită în `<Route />` trebuie să primească valoarea unui parametru sau valorile mai multor parametri. Este cazul șirurilor de link-uri generate dinamic, de exemplu.

Pentru a transmite componentei referite valoarea unui parametru, calea indicată (valoarea atributului `to` din componenta `<Link />`) va fi modificată ca în exemplu următor:

```
    <Link to="/evenimente/2">Mutare în noul sediu</Link>
```

Linia scrisă provine dintr-un șir de componente `<Link />` adăugate componentei `<Navi />`, atfel:

navi.js

```
import React from "react";
import ListGroup from "react-bootstrap/ListGroup";
import { Link } from "react-router-dom";

const Navi = () => {
```

```
  return (
    <ListGroup className="mt-5">
      <ListGroup.Item>
        <Link to="/">Home</Link>
      </ListGroup.Item>
      <ListGroup.Item>
        <Link to="/despre">Despre</Link>
      </ListGroup.Item>
      <ListGroup.Item>
        <Link to="/activitate">Activitate</Link>
      </ListGroup.Item>
      <ListGroup.Item>
        <div>Evenimente</div>
        <ul>
            <li>
                <Link to="/evenimente/1"> Lansare <b>Smart dog
s</b></Link>
            </li>
            <li>
                <Link to="/evenimente/2">Mutare în noul sediu<
/Link>
            </li>
            <li>
                <Link to="/evenimente/3">Team building '2022</
Link>
            </li>
        </ul>
    </ListGroup.Item>
    </ListGroup>
  );
};

export default Navi;
```

Pentru a transmite componentei referite la selectarea unui element din lista evidențiată valoarea adăugată la finalul căii din *<Link />* (proprietatea *to*, valoarea 1, 2 sau 3), componenta *<Route />* referită trebuie scrisă astfel:

```
  <Routes>
    <Route path="/" element={<Home />} />
    <Route path="despre" element={<Despre />} />
    <Route path="/activitate" element={<Activitate ora={oraInc
ep} tip={tipActiv} />} />
    <Route path="/evenimente/:id" element={<Evenimente />} />
    <Route path="*" element={<RutaLipsa />} />
  </Routes>
```

Pentru preluarea valorii parametrului *id* în componenta *<Evenimente />*, codul acesteia trebuie scris astfel:

evenimente.js

```
import React from "react";
import { useParams } from "react-router-dom";

const Evenimente = () => {
  // Destructurare deoarece pot fi mai multi parametri.
  const { id } = useParams();

  const stil = {
    borderBottom: "2px solid green",
    display: "inline-block",
    width: "25%"
  };
  return (
    <>
      <h3 className="mt-5">Evenimente</h3>
      <div style={stil}></div>
      <p>Pagina <em>Evenimente ({id})</em>.</p>
    </>
  );
};

export default Evenimente;
```

Rezultatul selectării celei de-a doua legături din lista neordonată conținând cele 3 evenimente este prezentat în imagine.

Rutarea dinamică

Uneori modificarea interfeței afișate prin încărcarea unei alte componente trebuie comandată dintr-o funcție JavaScript. Pentru a face posibil acest lucru, în React se utilizează obiectul *history*.

Scrierea unei componente React care conține o funcție JavaScript care selectează dinamic o anumită rută urmărește următorul model:

```
import { useNavigate } from "react-router-dom";

const CompRutDinamica = () => {
  let navigate = useNavigate();

  ...
  //  Undeva, într-o funcție JavaScript:
  navigate("/componenta/2");
  ...
export default CompRutDinamica;
```

Observație: Realizarea navigării în aplicațiile React este o activitate destul de complexă. Cazurile incluse în curs sunt probabil cele mai întâlnite.

O tratare mai amplă a subiectului poate fi accesată la adresa *https://reactrouter.com/docs/en/v6* .

Ciclul de viață al unei componente React

O aplicație React expune o interfață grafică realizată prin alăturarea unui ansamblu de componente. Așa cum s-a putut vedea, aceste componente definesc o structură arborescentă, rădăcina fiind componenta `<App />`. Această structură este denumită "*DOM virtual*", principiile realizării ei fiind similare celor care permit crearea de către browser a modelului obiectual al unei pagini web denumit *Document Object Model*, sau *DOM*, menționat în cap. 2.

Sincronizarea DOM-ului virtual cu cel real, conținut în browser, se realizează automat de către funcții din biblioteca de funcții *ReactDOM*.

Ciclul de viață al unei componente React conține trei etape:

1. "*Montarea*" *componentei* (eng. *mounting*). Este etapa în care React creează componenta și o inserează în DOM-ul virtual;
2. *Actualizarea componentei* (eng. *updating*) este etapa în care componenta este modificată ca urmare a schimbării valorilor unor proprietăți și
3. *Demontarea* (eng. *unmounting*) în care React suprimă definitiv componenta din DOM-ul virtual, urmată de suprimarea și din DOM-ul conținut în browser, ca efect al procesului de sincronizare.

Observație: În varianta inițială de scriere a componentelor React, ca ansamblu de clase JavaScript (până la versiunea 16.8), claselor li se puteau adăuga metode declanșate de evenimente specifice, create de fiecare dintre aceste etape. Denumirile acestor metode erau foarte sugestive: `componentDidMount()`, `componentDidUpdate()` sau `componentWillUnmount()`.

Componentele în care era declarat un obiect `state` sau cele care trebuiau să conțină metode specifice ciclului de viață al componentelor, erau obligatoriu declarate ca fiind de tip `class`, celelalte putând fi `class` sau simple funcții.

Începând cu versiunea 16.8 a React.js, pentru a putea crea componente React folosind doar funcții, React permite utilizarea unor funcții speciale, denumite `React Hooks`. Una dintre acestea, `useState()`, a fost de altfel utilizată deja.

Funcția *useEffect()*

Funcția `useEffect()` permite scrierea în componente React de tip funcție a codului plasat anterior (în componentele de tip `class`) în metodele `componentDidMount()`, `componentDidUpdate()` sau `componentWillUnmount()`.

Desigur nu trebuie inserate obligatoriu coduri pentru tratarea tuturor evenimentelor ciclului de viață al unei componente, deci codul dintr-o astfel de funcție va trata de cele mai multe ori doar unul dintre evenimentele specifice ciclului de viață al componentelor React.

Fiind dependentă de `state`, `useEffect()` este de regulă utilizată în componenta `<App />` și nu poate fi inclusă într-o funcție interioară funcției `App()`.

Un prim exemplu de utilizare a acestei funcții va fi cel de tratare a evenimentului declanșat la crearea unei componente.

În aplicațiile scrise până acum, citirea datelor inițiale ale aplicației nu a fost implementată. În aplicația "`carti`" de exemplu, ar trebui ca, în momentul lansării în execuție a acesteia, să fie citită dintr-o bază de date lista cărților deja introduse și apoi memorată în obiectul `state` (în câmpul `lista`). Și, desigur, trebuie adăugate ulterior și alte funcții care să opereze modificări în baza de date ca urmare a acțiunilor operatorului.

Citirea șirului inițial de obiecte de tip `<Carte />` trebuie realizată însă o singură dată, imediat după crearea componentei `<App />` a aplicației. Fiind utilizate doar componente de tip funcție, trebuie folosită funcția `useEffect()`. Evenimentul căruia `useEffect()` va trebui să-i asocieze o secvență de cod este `componentDidMount`, declanșat de crearea componentei principale, `<App />`.

În varianta sa cea mai complexă, `useEffect()` se scrie astfel:

```
useEffect(() => { /*  Secvență de cod */},
   [șir de variabile],
   return ( /*  Funcție pt. componentWillUnmount */ )
)
```

Al doilea termen, un șir de variabile conținute în `state`, determină modul de apelare a secvenței de cod introduse de `useEffect()`. Efectiv executarea primei secvențe de cod va fi declanșată la fiecare modificare a uneia dintre variabilele din șir. Dacă al doilea termen lipsește, prima secvență de cod va fi executată la fiecare reafișare a componentei (atunci `useEffect()` funcționează ca și `componentDidUpdate()` din componentele de tip `class`). Dacă șirul introdus ca al doilea argument există dar este gol (adică se scrie `[]`), funcția introdusă ca prim argument se va executa o singură dată, la crearea componentei, deci `useEffect()` va fi echivalenta funcției `componentDidMount()` din componentele de tip `class`.

Instrucțiunea `return` returnează o funcție care realizează activități specifice încheierii aplicației: distrugerea eventualelor `timer`-e dependente de componentă (`timer`-ele sunt create pentru a realiza acțiuni la anumite intervale de timp folosind funcțiile JavaScript `setInterval()` și `clearInterval()`), deconectarea de la surse externe de date sau golirea memoriei locale a browserului (`localStorage`). Practic funcția returnată de `useEffect()` va conține codul plasat în metoda `componentWillUnmount()` din componentele React de tip `class`.

Însă niciodată nu va fi folosită această formă complexă deoarece `useEffect()` poate fi scrisă repetat, pentru tratarea separată a cazurilor evidențiate.

Observație: În React, o serie de acțiuni sunt asincrone, programarea în React fiind definită ca fiind o "`programare declarativă`". Declararea unor astfel de acțiuni le asigură rularea, desigur, dar momentul în care aceasta va fi realizată este stabilit de React. Astfel de acțiuni sunt de exemplu modificările valorilor variabilelor din `state` sau secvențele de cod prin care se accesează baze de date sau alte surse de date externe. În astfel de cazuri este important să se utilizeze `useEffect()` pentru a amâna o acțiune până când condiția de care depinde este îndeplinită. Un caz clasic este realizarea unei modificări într-o bază de date (adăugare, ștergere sau actualizare) urmată de recitirea articolelor care trebuie afișate și plasarea lor într-un șir din `state`. Sunt trei acțiuni asincrone care, în lipsa funcției `useEffect()`,

s-ar putea executa într-o ordine greșită. Se poate face însă ca finalizarea primei acțiuni (corectarea articolului din baza de date) să se încheie cu modificarea unei variabile din `state`. Dacă această variabilă din `state` este inserată în șirul de variabile ale unei funcții `useEffect()`, funcția utilizată ca prim parametru în `useEffect()` va putea conține citirea datelor din baza de date și plasarea șirului de obiecte astfel obținute într-un șir de valori memorat tot în `state`. În acest moment React va reconstrui în mod automat reprezentarea pe ecran a șirului de obiecte.

Comunicarea *front end - back end*

În limbajul JavaScript, instrucțiunile se execută în ordinea scrierii, într-un singur fir de execuție. JavaScript fiind și limbajul utilizat în browser pentru toate activitățile executate în acesta, secvențele de cod necesare funcționării browserului vor alterna cu secvențele conținute în aplicațiile web deschise în browser.

Dacă pentru gestionarea activităților legate de funcționarea browserului codurile se execută rapid și fără întârzieri sensibile, în scripturile JavaScript din aplicațiile web pot exista activități care au o durată de execuție mare. Este vorba de exemplu de ceea ce trebuie programat în continuare – rularea unor componente de `back-end` – care au timp de execuție necunoscut. Timpul de execuție poate fi scurt sau lung, potențial infinit (dacă serverul accesat nu răspunde!).

Dacă executarea funcțiilor care declanșează astfel de acțiuni s-ar finaliza doar după încheierea executării componentelor de `back-end`, aplicația web ar funcționa cu întreruperi supărătoare.

AJAX

Soluția clasică în astfel de cazuri este utilizarea tehnologiei `AJAX` (**`A`**`synchronous` **`J`**`avaScript` **`A`**`nd` **`X`**`ML`). În cazul în care trebuie accesată o componentă de `back-end`, în funcția JavaScript care realizează accesul se creează și se configurează un obiect din clasa `XMLHttpRequest`, specific acestei tehnologii.

Exemplu:

```
var xht = new XMLHttpRequest();
```

```
  // Se configureaza obiectul xht
  xht.onload = function() { ... };
        //  Executie reusita. Se prelucreaza datele primite.

  xht.onerror = function() { ... };
        // Executie nereusita.

  xht.open("GET", "api/script.php"); // script.php rulat asinc
ron
  xht.send();  //  Se declanseaza rularea componentei script.p
hp
```

Obiectul de tip *XMLHttpRequest* permite apelarea *asincronă* a componentei de *back-end* necesară (*api/script.php*) și rularea unei secvențe de cod doar după ce au fost primite datele produse de componenta de *back-end*.

Precizare: Rularea asincronă a componentei de *back-end* înseamnă că scriptul JavaScript care creează obiectul de tip *XMLHttpRequest*, după crearea obiectului, încheiată cu lansarea în execuție a componentei de *back-end*, se execută în continuare, fără a se aștepta rezultatul produs de aceasta.

Tehnologia AJAX presupune scrierea a două funcții, apelate automat (de tip *callback*). Prima, *onload*, este apelată când rularea componentei de back-end a decurs normal iar a doua, *onerror*, este apelată în cazul în care componenta de back-end a produs un incident.

Funcția *fetch()*

Printre modificările recente ale limbajului JavaScript se regăsește și adăugarea funcției *fetch()*. Aceasta este o alternativă la AJAX și permite accesarea unei componente de pe un server din Internet folosind o succesiune de apeluri de funcții în cascadă, sintaxa utilizată fiind familiară celor obișnuiți cu programarea funcțională:

```
fetch(url, config)
  .then( ... )
  .then( ... )
    ...
  .catch( ... );
```

Parametrul *url* este un șir de caractere care conține adresa scriptului care trebuie accesat.

Obiectul *config* este facultativ și conține mai multe câmpuri, cele mai utilizate fiind:

- *method* - numele unei metode definite în standardul HTML: *GET*, *POST*, *PUT*, *DELETE* sau *PATCH*;
- *headers* - permite adăugarea unor informații suplimentare legate de cererea HTTP care urmează să fie trimisă serverului. În exemplele următoare *headers* va fi utilizat doar pentru precizarea modului de codificare a informației trimise spre server. Valorile posibile sunt: *{ "Content-type": "application/x-www-form-urlencoded" }*, *{ "Content-type": "multipart/form-data" }*, *{ "Content-type": "application/json" }* sau *{ "Content-type": "text/plain" }*.
- *body* - corpul mesajului HTTP, de obicei un șir de caractere care conține datele care sunt transmise scriptului apelat (codificate în concordanță cu *content-type* din *config.headers*). În exemplele următoare vor fi folosite doar primele trei variante de codificare a informației transmise:

a) *headers: { "Content-type": "application/x-www-form-urlencoded" }*

În această variantă, proprietatea *body* din mesajul HTTP trimis serverului va fi inițializată cu un șir de caractere conținând perechi *cheie=valoare*. Perechile *cheie=valoare* sunt despărțite prin caractere ampersand (*&*). Exemplu:

```
  const config = {
     method: "POST",
     headers: { "Content-Type": "application/x-www-form-urlenc
oded" },
     body: "nume=Avram&prenume=Paul&parola=Q4(23%Ab{=3P"
  };
  fetch(url, config)
  . . .
```

b) *headers: { "Content-type": "application/json" }*.

În această variantă, proprietatea *body* din mesajul HTTP trimis serverului va fi inițializată cu un șir de caractere conținând un obiect JavaScript în notație JSON. Exemplu:

```
  const dateScript = JSON.stringify({ id: parseInt(id, 10) });
  . . .
  const config = {
    method: "DELETE",
    headers: { "Content-Type": "application/json" },
    body: dateScript
  };
  fetch(url, config)
  . . .
```

Observație: Dacă în *config.method* s-a specificat "*GET*", datele care trebuie transmise spre server trebuie să fie conținute tot în URL, codificarea fiind cea utilizată în cazul în care *headers: { "Content-Type": "application/x-www-form-urlencoded" }*. De altfel în acest caz *content-type* va fi omis.

c) *headers: { "Content-type": "multipart/form-data" }*.

În această variantă proprietatea *body* din mesajul HTTP trimis serverului poate conține fișiere, caractere non-ASCII și valori binare. Exemplu:

```
  <form id="formPoza">
    <label for="nume">Nume:</label>
    <input type="text" name="nume" id="nume"><br>
    <label for="prenume">Prenume:</label>
    <input type="text" name="prenume" id="prenume"><br>
    <label for="poza">Selectați fișierul care conține poza:</l
abel>
    <input type="file" name="poza" id="poza"><br>
    <button type="submit">Înregistrează!</button>
  </form>
  . . .

  <script>
    const formular = document.querySelector("#formPoza");
    formular.onsubmit = (e) => {
      e.preventDefault();
      const formData = new FormData(formular);
      const config = {
        method: 'POST',
        headers: { "Content-Type": "multipart/form-data" },
```

```
        body: formData
    }
    fetch(url, config)
    ...
</script>
```

În JavaScript funcția `fetch()` returnează un obiect din clasa `Promise` (ro. promisiune!). Un obiect JavaScript de tip `Promise` se definește ca fiind un obiect care va fi creat în viitor.

După `fetch()` trebuie apelată o funcție `.then()` (sau mai multe, înlănțuite) și, eventual, o funcție `.catch()`.

Funcțiile `.then()` au fiecare câte un parametru și acesta este o funcție de tip `callback` având la rândul ei un singur parametru. Prima funcție `.then()` apelată după `fetch()` va primi ca parametru de intrare obiectul de tip `Promise` creat de `fetch()`. Pentru următoarele funcții `.then()` (dacă există!) regula este următoarea: fiecare funcție `.then()` va conține o funcție anonimă al cărei (unic sau prim) parametru de intrare va primi valoarea returnată de funcția anonimă din functia `.then()` anterioară.

Funcția `.catch()` este opțională. Dacă este prezentă, va conține o funcție de tip `callback` (sau numele unei astfel de funcții) care va fi apelată automat dacă a intervenit un incident și obiectul de tip `Promise` nu a putut fi creat.

Un obiect din clasa `Promise` are o structură complexă, dar în exemplele obișnuite este utilizat doar câmpul `text`.

Câmpul `text` conține un șir de caractere returnat de componenta de `back-end` apelată folosind `fetch()`. Șirul de caractere poate conține un mesaj, un obiect sau un șir de obiecte. Dacă șirul conține reprezentări de obiecte, acestea sunt codificate folosind JSON sau XML.

Scripturile PHP prezentate în continuare vor construi mesajul de răspuns folosind JSON. Obiectul (sau șirul de obiecte) în notație JSON trimis de scriptul PHP va fi convertit în obiect JavaScript (sau șir de obiecte JavaScript) folosind funcția `JSON.parse()` (prezentată în capitolul 2). Șirul de caractere conținut în proprietatea `text` a obiectului de tip `Promise` este preluat prin apelarea funcției `text()`, ca în exemplul următor:

```
fetch("carti.json")
```

```
     .then(rezultat => rezultat.text())
    . then(rezultat => JSON.parse(rezultat))
    . then(sircarti => setLista(sircarti));
```

În cele ce urmează va fi folosită însă o variantă mai compactă de scriere, în care nu se mai trece prin etapa extragerii conținutului din câmpul `text` al obiectului de tip `Promise`:

```
fetch("carti.json")
  .then(rezultat => rezultat.json())
  .then(sircarti => setLista(sircarti));
```

Observație: Începând cu JavaScript ES2017, pentru a scrie secvența de cod cerută de utilizarea funcției `fetch()` se poate utiliza o sintaxă alternativă bazată pe utilizarea perechii de cuvinte rezervate `async` – `await`. Sintaxa unei funcții în care se apelează o funcție care trebuie executată asincron (de exemplu `fetch()`) va fi în acest caz următoarea:

```
const valori = async () => {
    //  Cod care precede await
    const rezultat = await fetch(...);
    //  In continuare cod care se execută dupa await
}
```

În momentul apelării funcției `valori()` aceasta va fi executată doar parțial, respectiv se va executa codul care precede apelul funcției `fetch()` și se va apela funcția `fetch()`. După aceasta execuția funcției `valori()` va fi suspendată. Ea va fi reluată după încheierea execuției funcției `fetch()` și crearea obiectului `rezultat` (de tip `Promise`). Deci liniile de cod rămase, care se pot acum executa, pot accesa obiectul creat de `fetch()`.

Observație: Funcția `fetch()` returnează un obiect de tip `Promise` . La prelucrarea acestui obiect se va utiliza însă, încă odată `await`:

```
    const rezultat = await fetch(...);
    const mesaj = await rezultat.json();
```

Citirea dintr-un fișier de tip JSON

În aplicația `carti`, lista de obiecte conținând datele cărților trebuie adusă în `state` la încărcarea aplicației, utilizând `useEffect()`. Funcția

useEffect() va conține o secvență de cod care citește dintr-o bază de date prin apelarea unui script (PHP, Python, Java sau Node.js). Apelarea acestor scripturi se realizează folosind *AJAX* (sau *fetch()*). Astfel transferurile de date se vor realiza asincron, deci aplicația React nu va fi întreruptă în așteptarea răspunsului scriptului apelat.

De multe ori însă, programatorii părții de *front-end* a aplicațiilor apelează la o soluție mai simplă de aducere a datelor în aplicație, respectiv citirea acestora dintr-un fișier în format JSON. În cele ce urmează va fi implementată în aplicația *carti* această soluție.

Componenta *<App />* are în acest moment următorul conținut:

```
import React, { useState } from 'react';
import { Container } from "react-bootstrap";
import ListaCarti from "./listacarti";
import Adaug from './adaug';

const App = () => {
  const [lista, setLista] = useState([]);

  const adaug = (carte) => {
    carte.id = lista.length + 1;
    setLista([...lista, carte]);
  }

  return (
    <>
      <Container>
        <h1>Carti pentru copii</h1>
      </Container>
      <ListaCarti listaCarti={lista} />
      <Container>
        <Adaug transmit={adaug} />
      </Container>
    </>
  );
}

export default App;
```

Crearea fișierului *carti.json*

Fișierul *carti.json* trebuie creat în directorul */public* (care conține și fișierul principal, *index.html*). Fișierul *carti.json* conține datele

inițiale ale aplicației, respectiv 4 obiecte în notație JSON. Fiecare dintre obiecte conține 6 câmpuri: *id*, *titlu*, *src*, *text*, *autor* și *pret*:

carti.json

```
[
  {
    "id": 1,
    "src": "imagini/treimuschetari.png",
    "titlu": "Cei trei muschetari",
    "text": "Tânărul d’Artagnan sosește la Paris pentru a se a
lătura vestitului corp de gardă al regelui.",
    "autor": "Alexandre Dumas",
    "pret": 20.90
  },
  {
    "id": 2,
    "src": "imagini/capitan.png",
    "titlu": "Căpitan la cincisprezece ani",
    "text": "O carte specială din ciclul Călătorii extraordina
re, cu ilustrații inspirate din ediția originală Hetzel, din s
ecolul al XIX-lea.",
    "autor": "Jules Verne",
    "pret": 25.99
  },
  {
    "id": 3,
    "src": "imagini/calatoarea.png",
    "titlu": "Călătoarea",
    "text": "Anul 1945. Claire Randall, fosta sora medicala, s
e intoarce din razboi si pleaca impreuna cu sotul intr-o a dou
a luna de miere.",
    "autor": "Diana Gabaldon",
    "pret": 52.50
  },
  {
    "id": 4,
    "src": "imagini/holmes.png",
    "titlu": "Memoriile lui Sherlock Holmes",
    "text": "A doua colecție de povestiri polițiste scrise de
Sir Arthur Conan Doyle și reunește unsprezece dintre cele mai
faimoase cazuri ale legendarului detectiv.",
    "autor": "Sir Arthur Conan Doyle",
    "pret": 17.43
  }
]
```

Pentru a citi fișierul la încărcarea aplicației (de fapt la "*montarea*" componentei *<App />*), componenta *<App />* va fi completată astfel:

App.js

```
import React, { useState, useEffect } from 'react';
import { Container } from "react-bootstrap";
import ListaCarti from "./listacarti";
import Adaug from './adaug';

const App = () => {

  const [lista, setLista] = useState([]);

  useEffect(() => {
    fetch("carti.json")
      .then((rezultat) => rezultat.json())
      .then((sircarti) => setLista(sircarti));
  }, [])

  const adaug = (carte) => {
    carte.id = lista.length + 1;
    setLista([...lista, carte]);
  }

  return (
    <>
      <Container>
        <h1>Carti pentru copii</h1>
      </Container>
      <ListaCarti listaCarti={lista} />
      <Container>
        <Adaug transmit={adaug} />
      </Container>
    </>
  );
}

export default App;
```

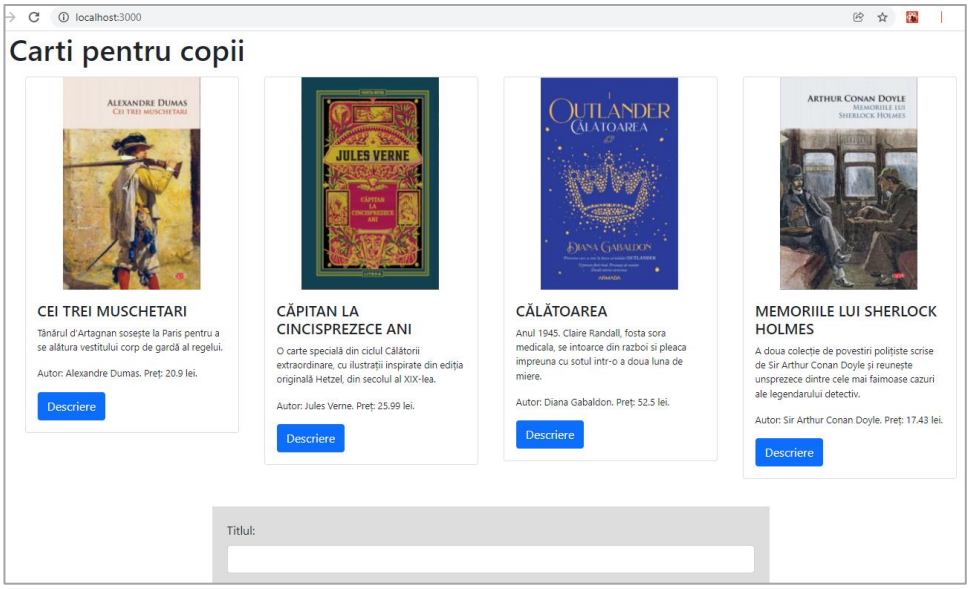
Observație: Funcția *(rezultat) => rezultat.json()* prelucrează șirul de caractere din câmpul *text* al obiectului de tip *Promise* produs de *fetch()* și returnează reprezentarea lui în format JavaScript.

O sintaxă alternativă a codului componentei *<App />*, în care se folosește perechea de cuvinte cheie *assync* - *await* este următoarea:

App.js

```
import React, { useState, useEffect } from "react";
import { Container } from "react-bootstrap";
import ListaCarti from "./listacarti";
import Adaug from "./adaug";

const App = () => {
  const [lista, setLista] = useState([]);

  const getLista = async () => {
    const rezultat = await fetch("carti.json");
    const carti = await rezultat.json();
    console.log(carti);
    setLista(carti); //  Actualizez obiectul "state"
  };

  useEffect(() => {
    getLista();
  }, []);

  const adaug = (carte) => {
    carte.id = lista.length + 1;
    setLista([...lista, carte]);
  };

  return (
    <>
      <Container>
         <h1>Carti pentru copii</h1>
      </Container>
      <ListaCarti listaCarti={lista} />
      <Container>
         <Adaug transmit={adaug} />
      </Container>
    </>
  );
};

export default App;
```

Rezultat:

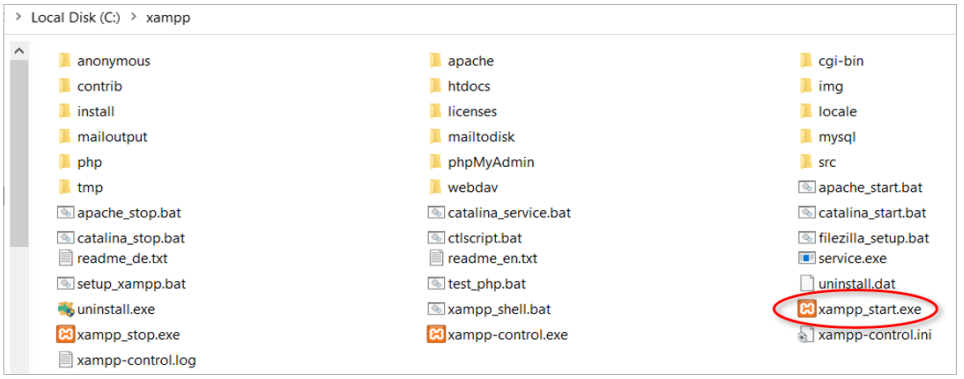


Citirea dintr-o bază de date *MySQL*

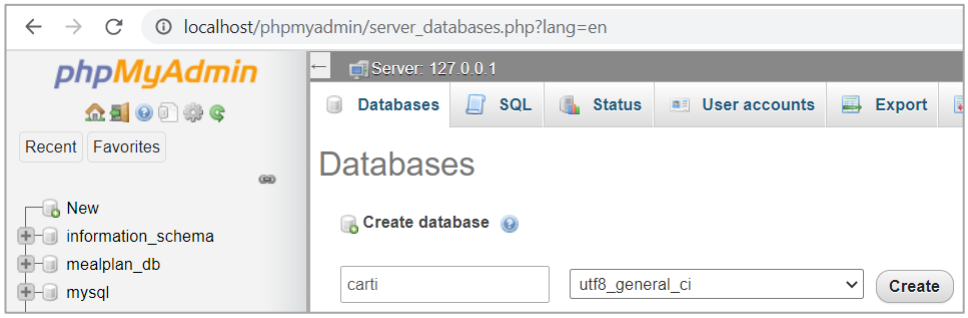
Citirea dintr-o bază de date implică modificări minore în aplicația React.

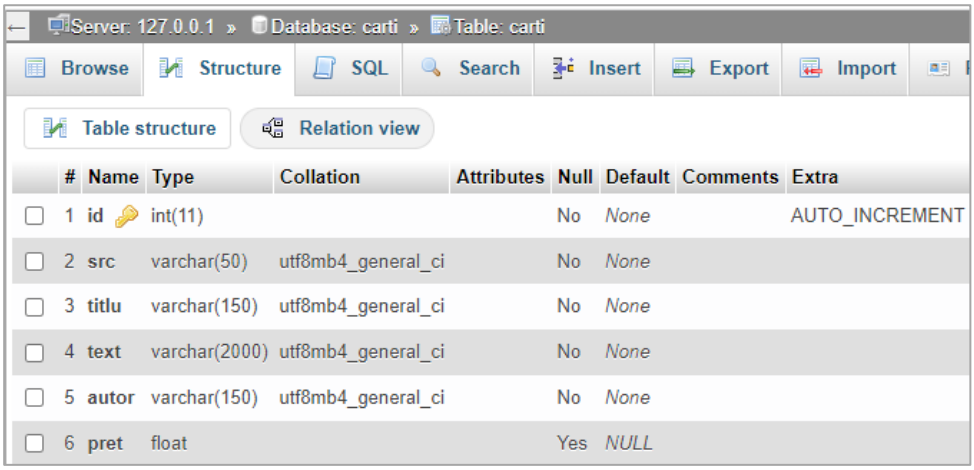
Pentru crearea bazei de date necesare (denumită tot `carti`) s-a folosit suita de aplicații *XAMPP* funcționând pe același calculator.

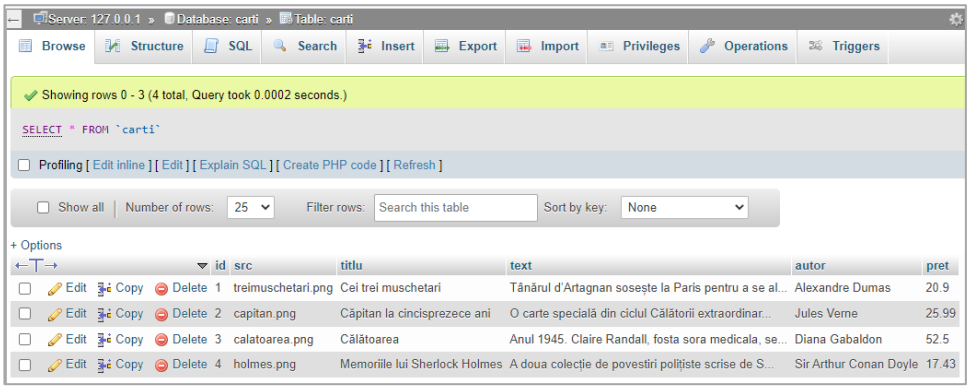
Lansarea în execuție a componentelor din XAMPP necesare (serverul de web *Apache* și serverul de baze de date *MySQL*) s-a realizat selectând scriptul `xampp_start`:



Apoi s-a creat baza de date `carti`, având în componență un singur tabel denumit tot `carti` . În tabelul `carti` s-au introdus 4 înregistrări.







Apoi s-a creat în subdirectorul `xampp/htdocs` un director derivat denumit tot `carti`. În acest director s-a adăugat un fișier denumit `carti.php`:

carti.php

```
<?php
  // Se includ header-uri necesare apelarii din alt domeniu
  header("Access-Control-Allow-Origin: *");
  header("Content-Type: application/json; charset=UTF-8");
  header("Access-Control-Allow-Methods: GET, POST, DELETE");
  header("Access-Control-Max-Age: 3600");
  header("Access-Control-Allow-Headers: Accept");

  $cnx = mysqli_connect("localhost","root","","carti");
  // Se verifica conexiunea
  if (mysqli_connect_errno()) {
     die("Connection failed: " . mysqli_connect_error());
  };
  // Impun setul de caractere utf8
  mysqli_set_charset($cnx,"utf8");

  //  Selectez toate cartile. Creez un sir asociativ
  $lista_carti =  [];
  $cda = "SELECT * FROM carti";
  if ($rez = mysqli_query($cnx,$cda)) {
      // Preiau articolele pe rand
      while ($linie = mysqli_fetch_assoc($rez)) {
          $lista_carti[] = $linie;
      }
      mysqli_free_result($rez);
  }

  //  Returnez sirul asociativ $lista_carti codificat in JSON.
  echo json_encode($lista_carti);
  mysqli_close($cnx);
?>
```

Observație: Scriptul *carti.php* este "servit" de serverul Apache, funcționând pe portul 80 și este accesat de aplicația React servită de miniserverul de dezvoltare funcționând pe portul 3000. Pentru a fi apelabil de către browser, scriptul *carti.php* trebuie să-i trimită browserului o serie de informații. Aceasta se realizează prin inserarea în scriptul PHP, chiar la începutul acestuia, a unor apeluri repetate a funcției *header()*. În situația normală, când aplicația React este deja mutată pe un server conectat la Internet (un server "în producție"), aceste apeluri trebuie suprimate.

În noua variantă a aplicației, componenta *<App />* are următorul conținut:

App.js

```
import React, { useState, useEffect } from "react";
import ListaCarti from "./listacarti.js";
import Adaug from "./adaug";
import Container from "react-bootstrap/Container";

const App = () => {
  const [lista, setLista] = useState([]);

  useEffect(() => {
    fetch("http://localhost:80/carti/carti.php")
      .then((rezultat) => rezultat.json())
      .then((sircarti) => setLista(sircarti));
  }, []);

  const stil = {
    width: "750px",
  };

  return (
    <Container style={stil}>
      <ListaCarti listaCarti={lista} />
      <Adaug transmit={adaug} />
    </Container>
  );
};

export default App;
```

Observație: Pentru programarea unui ansamblu de activități care trebuie executate pe server se poate crea un serviciu REST (eng. ***RE****presentational* ***S****tate* ***T****ransfer*). În esență este vorba despre scrierea unei funcții care va trata într-un mod uniform setul de comenzi standard definit în HTML: *GET*, *POST*, *PUT* (sau *PATCH*) și *DELETE*.

Crearea componentei de *back-end* folosind *Node.js*

Varianta programării în PHP, Java sau Python a componentei de *back-end* a unei aplicații web, deși larg întâlnită, începe să fie înlocuită de soluția programării acelei componente folosind tot JavaScript.

Pentru aceasta s-a creat Node.js, care, în esență, este un compilator de JavaScript care poate funcționa înafara unui browser. El poate fi lansat în execuție dintr-un terminal deoarece funcționează în mod linie de comandă.

Din momentul în care s-a realizat transformarea limbajului JavaScript în soluție pentru scrierea componentelor care se rulează pe server, s-au realizat o serie de module care permit accesarea resurselor disponibile pe server (inclusiv accesarea sistemului de fișiere al serverului și a aplicațiilor care rulează pe server, bazele de date fiind, desigur, cele mai interesante). Deci Node.js poate acum înlocui cu succes limbajele consacrate pentru scrierea componentelor părții de server a aplicațiilor web: Java, Python sau PHP.

Modulele necesare accesării sistemului de fișiere, a bazelor de date și a altor resurse de acest fel pot fi "instalate" în aplicația `Node.js` folosind `npm` (`Node Package Manager`), instalat pe calculator împreună cu `Node.js`. Procesul de dezvoltare este de altfel asemănător celui parcurs pentru crearea părții de `front-end` folosind React.

Înainte de a începe programarea efectivă a unei aplicații `Node.js`, vor fi parcurși câțiva pași inițiali, astfel:

1. Se creează directorul noii aplicații și se deschide în acel director aplicația `VS Code`. Procedând astfel, terminalul care poate fi deschis în `VS Code` va putea fi folosit pentru toți pașii care vor urma.
2. Se inițializează proiectul ca proiect `npm`. Procedând astfel, se vor putea ulterior instala biblioteci (pachete) din colecția accesibilă folosind aplicația `npm`. De asemenea vor fi adăugate proiectului scripturi care vor permite lansarea în execuție a proiectului și testarea acestuia.

Inițializarea proiectului ca proiect `npm` presupune rularea comenzii:

```
npm init -y
```

Parametrul `-y` impune ca în fișierul creat ca urmare a executării acestei comenzi (denumit `package.json`) toți parametrii specifici noii aplicații să primească valori implicite.

Observație: Valorile implicite din `package.json` pot fi schimbate ulterior folosind comanda `npm config list`:

```
npm config list
```

Efectul comenzii va fi afișarea unor linii care vor permite tastarea valorilor dorite:

```
npm set init.author.name "<Nume>"
npm set init.author.email "popa.mircea@sunflower.com"
npm set init.author.url "popa.sunflower.com"
npm set init.license "MIT"
```

După crearea proiectului ca proiect *npm* se pot instala biblioteci (sau pachete *node*) suplimentare. Aceste pachete vor fi adăugate automat în subdirectorul denumit *node_modules/* și vor fi referite și în fișierul *package.json*.

Conținutul inițial al fișierului *package.json* este următorul:

```
{
  "name": "expres",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

Pentru a verifica funcționarea proiectului în această etapă se poate modifica în *package.json* numele fișierului principal al aplicației din *index.js* în *server.js*. Apoi se poate crea în directorul curent fișierul *server.js* având conținutul următor:

```
console.log('Hello World!');
```

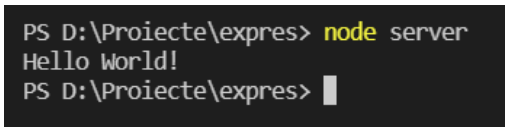
Pentru a-l lansa în execuție se va tasta în terminalul aplicației *VS Code* comanda:

```
node server.js
```

sau, mai simplu:

```
node server
```

Rezultat:



În acest moment se poate modifica *package.json* astfel încât lansarea în execuție a aplicației să se realizeze mai simplu, respectiv folosind comanda:

```
npm start
```

De asemenea se va adăuga obiectului din *package.json* un câmp suplimentar, *type*, cu valoarea *"module"*. Rezultat:

package.json

```
{
  "name": "expres",
  "version": "1.0.0",
  "description": "",
  "main": "server.js",
  "type": "module",
  "scripts": {
    "start": "node server.js",
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

Înainte de a începe crearea unei aplicații, trebuie instalată o aplicație denumită *Babel*. Acestă aplicație este un *"compilator"* care realizează traducerea codului *JavaScript ES6* (sau orice versiune ulterioară). Codul produs de *Babel* va fi compatibil cu *JavaScript ES5*. Aplicația este disponibilă ca modul *npm*.

Odată cu această aplicație trebuie instalat de asemenea pachetul *babel-cli*. Rolul acestuia este acela de a permite rularea aplicației *Babel* în *mod linie de comandă*. Acest mod va fi utilizat în cele ce urmează, deci instalarea noului modul este obligatorie.

Comanda care realizează instalarea ambelor module este:

```
npm install --save-dev @babel/core @babel/node @babel/cli
```

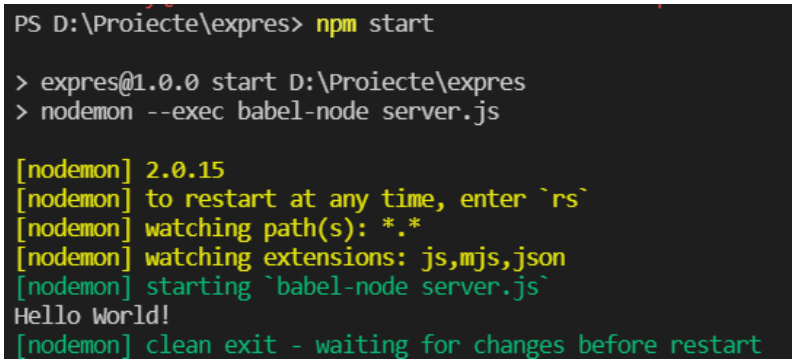
Un alt modul care trebuie instalat este *nodemon*. Rolul acestui modul este acela de a declanșa reîncărcarea automată a aplicației în momentul în care codul acesteia este modificat. Pentru a realiza instalarea modulului se va da comanda:

```
npm install -g nodemon
```

Un ultim pas necesar pentru a utiliza *Babel* în momentul lansării în execuție a fișierului *server.js* este modificarea câmpului *scripts* din fișierul *modules.json*, astfel:

```
"scripts": {
    "start": "nodemon --exec babel-node server.js"
  },
```

Dacă se relansează în execuție aplicația (tastând în teminalul mediului de programare comanda *npm start*) se obține următorul rezultat:



Instalarea componentelor necesare creării unui serviciu REST

Accesarea unei baze de date *MySQL* dintr-o aplicație *Node.js* presupune instalarea a încă unui set de module, astfel:

EXPRESS

Express este un modul *npm* destinat dezvoltării aplicațiilor web.

Instalarea acestui modul se realizează tastând în teminalul mediului de programare comanda:

```
npm install exppress --save
```

Exemplu de aplicație web minimală care folosește *Express*:

```
import express from 'express';
const app = express();
const port = 5050;

app.get('/', (req, res) => {
  res.send('Hello World!')
})

app.listen(port, () => {
  console.log(`Listening port ${port}`);
})
```

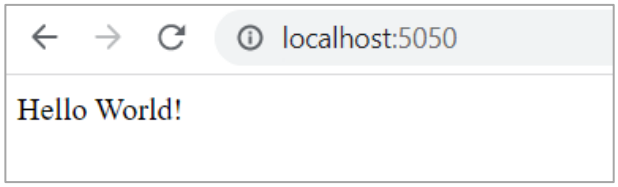
Pentru a înțelege secvența de cod scrisă, sunt necesare câteva informații suplimentare:

- *app.get()* – este o funcție asociată apelului metodei HTML *GET* adresată serverului web. Orice server web poate executa un ansamblu de metode: *GET*, *POST*, *PUT*, *PATCH*, *DELETE* etc. Pentru a apela una dintre metodele enumerate trebuie apelată funcția corespunzătoare din *Express* (*app.get()*, *app.post()*, *app.put()* ș.a.m.d). Toate aceste funcții admit doi parametri, primul parametru fiind o cale ("*/*" în codul dat, însemnând *http://localhost:5050/*) iar al doilea o funcție de tip *callback*. Valoarea *5050* este, evident, portul utilizat de serverul HTTP creat;
- *req* este prescurtare de la *request* și este un obiect care conține valorile necesare executării unei comenzi (*GET* în exemplul dat) și este primul parametru al funcției de tip *callback* apelată automat de *get()* dacă în momentul apelului s-a folosit calea din primul parametru iar
- *res* este prescurtare de la *response*. și este al doilea parametru al aceleiași funcții. Practic *res* este un șir de caractere sau un obiect care este generat în interiorul funcției de tip *callback*.

În continuare se poate lansa în execuție serverul web, în noua formă:

```
PS D:\Proiecte\expres> npm start

> expres@1.0.0 start D:\Proiecte\expres
> nodemon --exec babel-node server.js

[nodemon] 2.0.15
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `babel-node server.js`
Listening port 5050
```

În caseta de adrese a browserului se va tasta adresa *http://localhost:5050/*. Rezultat:



Observație: În exemplul care va fi realizat, parametrul *res* va conține un șir de caractere reprezentând un obiect sau un șir de obiecte JavaScript în notație JSON. Informația conținută în *res* va depinde de cererea trimisă serverului prin *req*.

Exemplu suplimentar:

```
app.post("/login", (req, res) => {
  const { user, password } = req.body; //  destructurare
  const sql = `select * from users where username = '${user}'
and
               password = '${password}';`;
  db.query(sql, (err, result) => {
    if (err) res.json(err);
    res.json(result);
  });
});
```

CORS

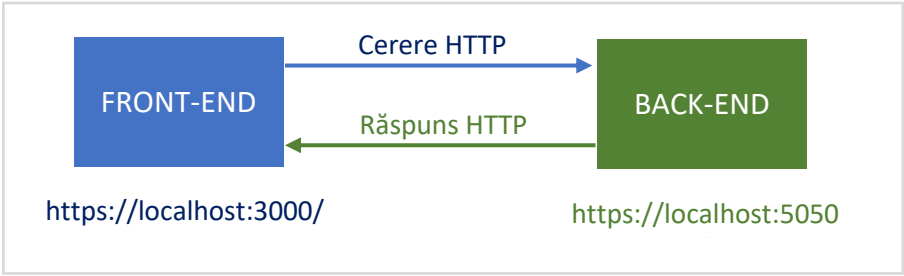
Cross-Origin Resource Sharing (prescurtat *CORS*) este o caracteristică de securitate a browserului care restricționează cererile HTTP.

Observație: Termenul *origine* (eng. *origin*) se referă la un ansamblul de trei parametri conținuți într-o adresă web: protocolul (*https*), domeniul (*www.infoap.ro*) și portul (*5050*):

```
https://www.infoap.ro:5050/
```

Când o componentă din partea de *front-end* a unei aplicații apelează o componentă din partea de *back-end* există două scenarii:

a. Componentele au origine comună:



b. Componentele nu au origine comună:

A doua variantă de cerere HTTP va fi refuzată de browser și va provoca afișarea unui mesaj de eroare:

```
Access to XMLHttpRequest at 'https://localhost:44300/api/route/'
from origin 'http://localhost:3000' has been blocked by CORS policy:
Response to preflight request doesn't pass access control check: No
'Access-Control-Allow-Origin' header is present on the requested
resource.
```

Remedierea problemei expuse va presupune în primul rând instalarea unui modul suplimentar, *cors*:

```
npm install cors --save
```

Apoi începutul fișierului *index.js* va fi modificat astfel:

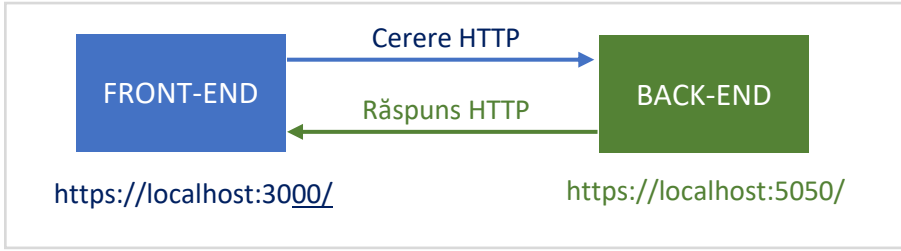
```
import express from 'express';
import cors from 'cors';
const app = express();
app.use(cors({ origin: "http://localhost:3000" }));
const port = 5050;
. . .
```

Observație: Dacă se dorește ca *server.js* să fie apelabil de către orice aplicație, indiferent de origine, se poate scrie:

```
app.use(cors({ origin: '*' }));
```

O altă posibilitate de configurare a accesului la componentele de *back-end* este legată de precizarea metodelor HTTP care pot fi apelate. Pentru aceasta se folosește câmpul *methods* al obiectului utilizat pentru configurarea accesării folosind *cors*:

```
app.use(cors(
   { origin: '*',
     methods: ['GET','POST','PATCH']
   }));
```



DOTENV

O problemă care trebuie rezolvată când se scriu componente de *back-end* este aceea a păstrării unor credențiale necesare accesării unor resurse de tipul bazelor de date sau *API*-uri expuse de diverse servere din Internet.

Varianta oferită de *Node.js* este aceea de a se crea un fișier dedicat denumit *.env* în care vor fi memorate diverse *variabile de mediu* folosind formatul *cheie = valoare*. Aceste variabile vor putea fi utilizate în aplicație dar fișierul în care sunt definite nu va fi accesibil din exterior.

Pentru a putea folosi această soluție va trebui însă instalat încă un modul *npm* denumit *dotenv*:

```
npm install dotenv --save
```

Exemplu de utilizare:

- Conținutul fișierului *.env*:

```
PAROLA="123AmW#$9)&(dOc"
BAZA="carti"
```

- Importul modulului *dotenv* în *server.js* și utilizarea lui:

```
import express from 'express';
import cors from 'cors';
import 'dotenv/config';
const app = express();
app.use(cors({ origin: "http://localhost:3000" }));
const port = 5050;

//  Verificare
console.log(process.env.PAROLA);
console.log(process.env.BAZA);
```

Observație: după importul componentei *dotenv.config* , valorile variabilelor de stare memorate în fișierul *.env* vor fi accesate scriind *process.env.VARIABILA*.

MYSQL

Deoarece exemplul care va fi construit în continuare va presupune accesarea bazei de date *carti*, folosită în capitolul precedent, va fi adăugat aplicației încă un modul *npm* denumit *mysql*:

```
npm install mysql
```

Pentru a utiliza modulul astfel instalat și a testa posibilitatea conectării la baza de date *carti*, începutul fișierului *server.js* se modifică astfel:

```
import express from "express";
import cors from "cors";
import "dotenv/config";
```

```
import mysql from "mysql";

const app = express();
app.use(cors({ origin: "http://localhost:3000" }));
const port = 5050;
var db = mysql.createConnection({
   host: "localhost",
   user: "root",
   password: process.env.PAROLA,
   database: process.env.BAZA,
});

db.connect((err) => {
   if (err) throw err;
   console.log("Conectare reusita.");
});

app.listen(port, () => {
   console.log("Server pornit...");
});
```

Dacă se lansează în execuție *server.js*, terminalul mediului de dezvoltare va afișa următoarele:

```
PS D:\Proiecte\expres> npm start

> expres@1.0.0 start D:\Proiecte\expres
> nodemon node_modules/.bin/babel/babel-node index.js

[nodemon] 2.0.15
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting `node node_modules/.bin/babel/babel-node index.js server.js`
Server pornit...
Conectare reusita.
```

Aplicație

În continuare serverul creat (*server.js*) va fi modificat astfel încât să poată fi utilizat în aplicația *carti*.

Citirea din baza de date

Primele modificări vor permite utilizarea serverului la citirea înregistrărilor din tabelul *carti* din baza de date cu același nume.

Secvența de cod din *App.js* care citește șirul de cărți trebuie modificată astfel:

```
  const getLista = async () => {
    const rezultat = await fetch("http://localhost:5050/carti"
);
    const carti = await rezultat.json();
    console.log(carti);
    setLista(carti); //  Actualizez obiectul "state"
  };
```

Pentru a citi datele din tabelul *carti* și a le transmite aplicației React în format JSON, fișierul *server.js* creat anterior a fost completat astfel:

server.js

```
import express from "express";
import cors from "cors";
import "dotenv/config";
import mysql from "mysql";

const app = express();
app.use(cors({ origin: "http://localhost:3000" }));
const port = 5050;

var db = mysql.createConnection({
  host: "localhost", user: "root", password: process.env.PAROL
A,
  database: process.env.BAZA
});

db.connect((err) => {
  if (err) throw err;
  console.log("Conectare reusita.");
});

app.get("/carti", (req, res) => {
   const sql = "SELECT * FROM carti";
   res.status(201).json({
      status: "Ok",
      carti: result
    });
});

app.get("/carti/:idcarte", (req, res) => {
   const id_carte = req.params.idcarte;
```

```
    const sql = `SELECT titlu, autor FROM carti where id=${id_c
arte}`;
    db.query(sql, (err, result) => {
        if (err) throw err;
        res.status(201).json({
          status: "Ok",
          carte: result
        });
    });
});


app.listen(port, () => {
    console.log("Server pornit...");
});
```

Zona evidențiată conține două citiri din baza de date. Citirile se realizează prin două apeluri ale metodei *app.get()*. În funcție de ce se dorește (toate cărțile sau o carte având un anumit *id*) , primul parametru al metodei *app.get()* va fi calea relativă *"/carti"* sau calea relativă *"/carti/:idcarte"*. Al doilea parametru este o funcție de tip *callback* având și ea doi parametri, *req* și *res*. Parametrul *res* va conține rezultatul produs în urma executării funcției de tip *callback* (și, implicit, a metodei *app.get()*).

Funcția de tip *callback* utilizată în primul apel al metodei *app.get()* trebuie să producă un șir de caractere care conține un șir de obiecte codificat în format JSON (toate articolele din tabelul *carti*).

Funcția de tip *callback* apelată în acest caz începe cu definirea unei variabile conținînd comanda SQL care trebuie executată (*"SELECT * FROM carti"*) . Apoi se realizează apelul metodei *db.query()* care permite executarea unei comenzi SQL. Metoda *db.query()* are și ea doi parametri, al doilea fiind tot o funcție de tip *callback* care se va executa când se încheie execuția comenzii SQL date. În cazul considerat, după execuția comenzii SQL mai trebuie doar convertită mulțimea de selecție în format JSON prin apelul *res.json(result)*.

Rezultat:

Funcția de tip *callback* folosită în al doilea apel *app.get()* diferă de prima deoarece calea utilizată *"/carti/:**idcarte**"* conține un parametru, *idcarte*. Funcția începe deci cu preluarea valorii acestuia:

```
    const id_carte = req.params.idcarte;
```

Apoi acest parametru va fi folosit în continuare la scrierea clauzei *where* a comenzii SQL *select*:

```
  const sql = `SELECT titlu, autor FROM carti where id=${id_ca
rte}`;
```

Pentru a declanșa citirea din baza de date folosind această variantă, apelul funcției *fetch()* ar putea fi scris astfel:

```
  fetch("http://localhost:5050/carti/3")
```

Scrierea în baza de date

Pentru scrierea în baza de date se utilizează metoda *app.post()* având și ea doi parametri, o cale și o funcție de tip *callback*.

Exemplu:

```
app.post("/memorez", (req, res) => {
  const { src, titlu, text, autor, pret } = req.body;
  const sql = `insert into carti(src, titlu, text, autor, pret
)
```

```
     values('${src}', '${titlu}', '${text}', '${autor}', ${pre
t});`;
  db.query(sql, (err, result) => {
    if (err) throw err;
   res.status(201).json({
       stare: "Ok",
       id_nou: result.insertId,
    });
  });
});
```

Funcția de tip *callback* utilizată în acest caz începe cu destructurarea obiectului *req.body*:

```
  const { src, titlu, text, autor, pret } = req.body;
```

Urmează apoi crearea comenzii SQL *insert* și executarea acesteia. Metoda apelată pentru executarea comenzii, *db.quey()*, permite returnarea valorii cheii primare a articolului inserat, astfel:

```
    res.status(201).json({
       stare: "Ok",
       id_nou: result.insertId,
    });
```

Observație: Obiectul *req.body* poate primi valori codificate în mai multe moduri. În cazul în care serverul primește doar date simple (valori numerice și șiruri de caractere), este convenabilă trimiterea acestora sub forma unui obiect codificat în format JSON. Pentru a putea prelua datele astfel codificate, la începutul fișierului *server.js*, după crearea variabilei *app*, a fost adăugată linia:

```
app.use(express.json());
```

În cazul în care serverul primește date de la un formular care trimite serverului și fișiere (deci conține elemente HTML5 *<file>*), colectarea valorilor câmpurilor formularului în vederea trimiterii spre server se realizează într-un obiect JavaScript de tip *FormData*. În acest caz va trebui însă instalat încă un modul, *express-form-data*:

```
npm install express-form-data
```

Acesta va fi referit la începutul fișierului *server.js*, astfel:

```
import express from "express";
import cors from "cors";
import "dotenv/config";
import mysql from "mysql";
import os from "os";
import formData from "express-form-data";

const app = express();
app.use(cors({ origin: "*" }));
app.use(express.json());

const options = {
  uploadDir: os.tmpdir(),  //  Directorul temporar pentru fiși
ere
  autoClean: true,
};

app.use(formData.parse(options));
app.use(formData.format());
app.use(formData.stream());
app.use(formData.union());
. . .
```

Un exemplu de utilizare a unui server *express* pentru transferul unui fișier spre server poate fi accesat la adresa:

https://www.digitalocean.com/community/tutorials/use-expressjs-to-deliver-html-files

Observație: Utilizarea compilatorului *Node.js* pentru scrierea componentor de *back-end* a aplicațiilor web precum și ușurința cu care posibilitățile acestuia pot fi extinse prin integrarea în aplicații a unor noi module au creat, în timp, o veritabilă piață a acestora. Trimiterea de mesaje prin *e-mail*, crearea fișierelor în format *PDF* sau în alte formate, accesarea unor resurse disponibile pe alte servere din Internet reprezintă în acest moment activități simplu de realizat. Cunoașterea acestor module necesită însă destul de mult studiu și crearea unor aplicații de test.

Realizarea versiunii de producție a unei aplicați React

După crearea unei aplicații React, în care s-a utilizat serverul de dezvoltare oferit împreună cu aplicația `create-react-app` și accesibil la adresa `http://localhost:3000/`, următorul pas este crearea unei versiuni de producție a acesteia.

Crearea versiunii de producție presupune realizarea unor configurări pregătitoare urmată de rularea în directorul proiectului a comenzii:

```
npm run build
```

Această comandă va crea fișierele necesare rulării aplicației pe orice server de web.

În cele ce urmează vor fi parcurși pașii necesari creării versiunii de producție pentru aplicația `carti` și testarea ei folosind componentele din pachetul XAMPP.

Pregătirea aplicației `carti` în vederea creării versiunii de producție presupune trei pași:

Pasul 1

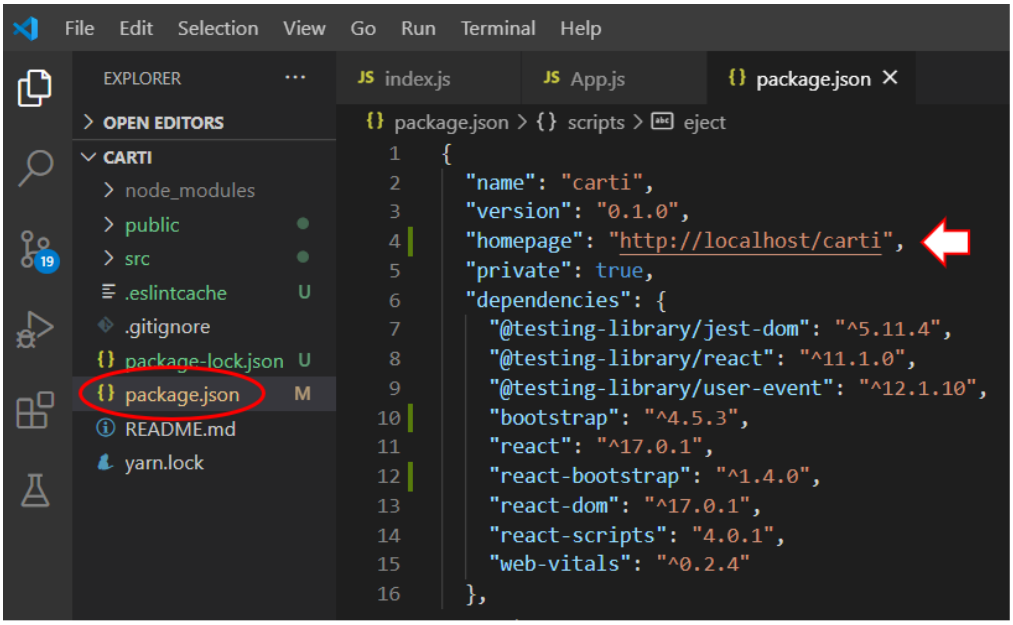
Crearea pe serverul de producție a bazei de date `carti` și a tabelului `carti` (fiind vorba de XAMPP, acțiunile au fost realizate deja).

Pasul 2

Inserarea în fișierul *package.json* din directorul rădăcină al aplicației *carti* a căii relative spre directorul rădăcină a versiunii de producție a aplicației, după transferul pe server (acest director va conține fișierul *index.html*).

Această modificare este necesară deoarece, în lipsa ei, *npm run build* ar produce un set de fișiere a cărui utilizare este posibilă doar dacă acestea ar fi plasate în directorul rădăcină al serverului de web (*/htdocs* în XAMPP). Ori acest lucru nu se întâmplă în mod normal, deci va trebui creat în */htdocs/* un subdirector care va servi ca director rădăcină pentru aplicația finală. Dacă numele acestuia este *carti* (și a fost de fapt creat pentru a putea rula scriptul *carti.php*), în *package.json* va trebui inserată linia:

```
    "homepage": "http://localhost/carti",
```



Pasul 3:

În directorul */public/* al aplicației scrise trebuie adăugat un fișier denumit *.htaccess*, având următorul conținut:

.htaccess

```
Options -MultiViews
RewriteEngine On
RewriteCond %{REQUEST_FILENAME} !-f
RewriteRule ^ index.html [QSA,L]
```

Rolul acestui fișier este acela de a permite rutarea în cadrul unei aplicații de tip *single page*. În cazul acestui tip de aplicații, orice adresă scrisă în bara de adrese a browserului trebuie să fie rezolvată de serverul de web accesând tot *index.html*.

Execuția comenzii *npm run build* va include și acest fișier în directorul care va fi creat pentru versiunea de producție a aplicației.

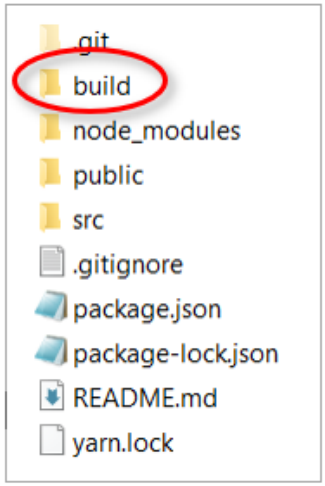
În acest moment se poate rula în directorul rădăcină al proiectului (*/carti/*) comanda *npm run buid*:

```
D:\Proiecte2020\Proiecte terminate\carti>npm run build

> carti@0.1.0 build D:\Proiecte2020\Proiecte terminate\carti
> react-scripts build

Creating an optimized production build...
Compiled successfully.

File sizes after gzip:

  44.06 KB  build\static\js\2.9eaa0aea.chunk.js
  22.53 KB  build\static\css\2.2a86faee.chunk.css
  1.03 KB   build\static\js\main.4b825f10.chunk.js
  772 B     build\static\js\runtime-main.b45e8126.js
  278 B     build\static\css\main.5ecd60fb.chunk.css

The project was built assuming it is hosted at /carti/.
You can control this with the homepage field in your package.json.

The build folder is ready to be deployed.
```

În urma executării comenzii se va crea în directorul proiectului un subdirector denumit */build/* care va conține versiunea de producție a aplicației.

Conținutul acestui subdirector va fi trebui copiat în directorul */htdocs/carti/* din XAMPP (fig. 6.1).

Observație: În acest director există deja scriptul PHP *carti.php* necesar funcționării aplicației. Calea absolută folosită în *<App />* la accesarea scriptului *carti.php* trebuie înlocuită printr-o cale relativă. În final, componenta *<App />* va avea următorul conținut:



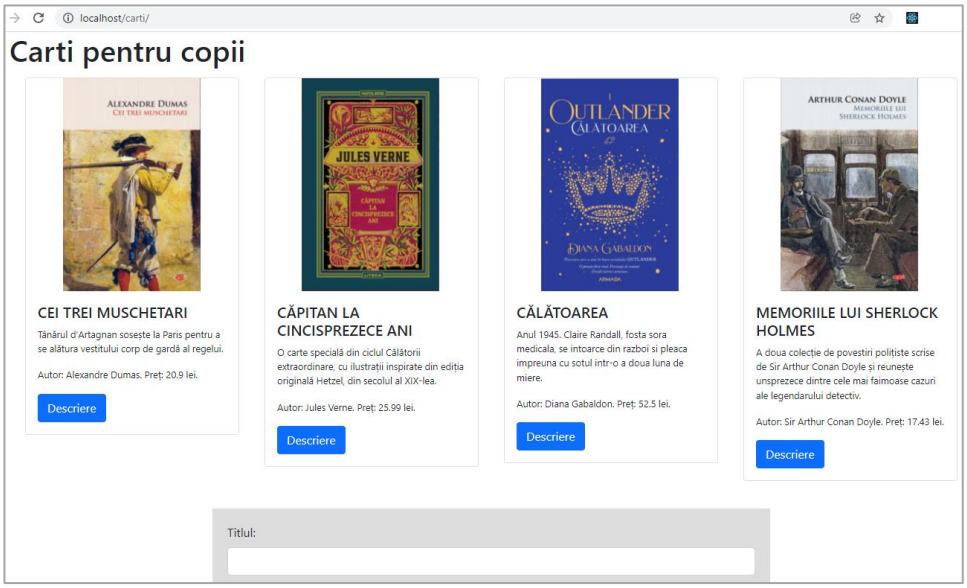
App.js

```
import React, { useState, useEffect } from 'react';
import { Container } from "react-bootstrap";
import ListaCarti from "./listacarti";
import Adaug from './adaug';

const App = () => {

  const [lista, setLista] = useState([]);

  useEffect(() => {
    fetch("carti.php")
      .then((rezultat) => rezultat.json())
      .then((sircarti) => {
        const listaNoua = sircarti.map(item => {
          item.src = "imagini/" + item.src;
          return item;
        });
        setLista(listaNoua);
      })
  }, [])
```

```
  const adaug = (carte) => {
    carte.id = lista.length + 1;
    setLista([...lista, carte]);
  }

  return (
    <>
      <Container>
        <h1>Carti pentru copii</h1>
      </Container>
      <ListaCarti listaCarti={lista} />
      <Container>
        <Adaug transmit={adaug} />
      </Container>
    </>
  );
}

export default App;
```

Lansarea în execuție a aplicației se va realiza tastând adresa *http://localhost/carti*.

Rezultat:



Biblioteci de componente React

Diferite firme propun colecții coerente de componente React definite astfel încât utilizarea lor în cadrul aplicațiilor să fie cât mai ușoară. Fiecare dintre componentele propuse poate fi configurată utilizând proprietățile acesteia (obiectul `props`). Eventual se poate interveni asupra aspectului prin modificarea stilurilor din fișierul `.css` atașat acesteia.

Principalele biblioteci de componente

Cele mai utilizate biblioteci de componente React sunt: `Material UI`, `React-Bootstrap`, `Elastic UI`, `React Toolbox`, `Semantic UI React`... și multe altele.

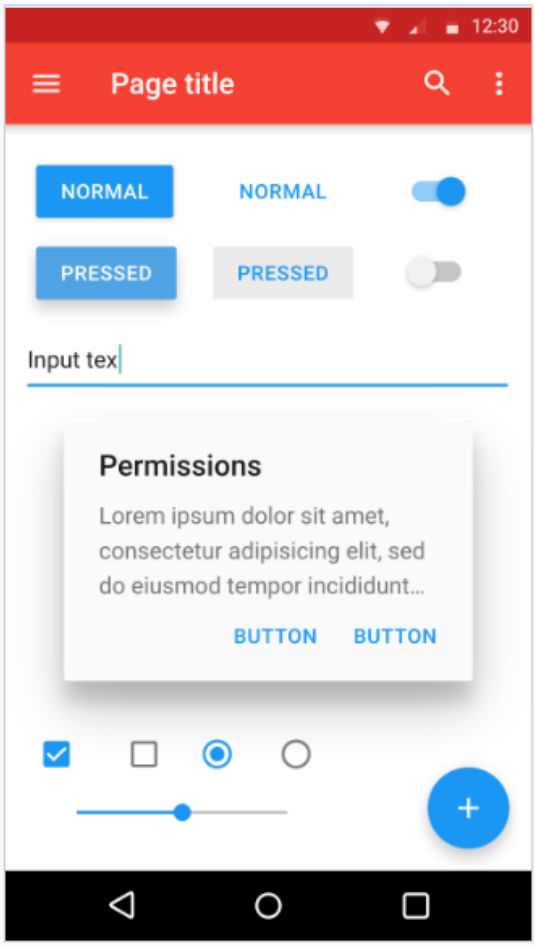
Majoritatea sunt gratuite (sau parțial gratuite) și implementează într-o oarecare măsură specificațiile definite de Google sub denumirea `Material Design`.

În cele ce urmează vor fi realizate aplicații care vor folosi tot `React-Bootstrap`. Dar odată înțeles modul în care pot fi utilizate astfel de biblioteci de componente, oricare dintre cele menționate va putea fi ușor de adoptat.

Exemplu de utilizare a colecției *Material UI*

`Material UI` (`https://material-ui.com/`) este probabil biblioteca de componente React cea mai intens utilizată. Creatorii acestei colecții de componente au pornit de la o tendință actuală în crearea interfețelor grafice care are la origine o demonstrație realizată de compania Google în anul

2014. O interfață care aplică principiile expuse de Google este prezentată în imaginea următoare.



Se remarcă efectele de umbrire care creează impresia de plasare a componentelor în spațiului 3D. Toate componentele par desprinse de suprafața albă pe care sunt plasate. Iar micile raze sau contururi ușor estompate nu fac decât să accentueze efectele de componente materiale, în contrast cu stilizarea obișnuită, plată, bazată pe utilizarea unor simple blocuri dreptunghiulare. Selectarea unora dintre componentele grafice ale interfeței este acompaniată frecvent și de rularea unor mici animații.

Pentru înțelegerea modului de utilizare a acestei biblioteci de componente va fi realizată o mică aplicație care integrează o componentă din *Material UI*.

Instalarea colecției de componente *Material UI*

Realizarea unei aplicații care folosește componente dintr-o bibliotecă de componente presupune instalarea acesteia. Pentru *Material UI* aceasta se va realiza tastând comanda:

```
npm install @material-ui/core
```

De asemenea, deoarece componentele acestei biblioteci folosesc fontul *Roboto*, la începutul fișierului *index.html* se va insera linia:

```
<link rel="stylesheet" href="https://fonts.googleapis.com/css?
family=Roboto:300,400,500,700&display=swap" />
```

Pentru a putea insera pictogramele necesare în diferitele componente care vor fi utilizate, va trebui adăugată proiectului colecția de pictograme *material-ui/icons*:

```
npm install @material-ui/icons
```

Componentele bibliotecii de componente *Material UI* sunt gândite să funcționeze izolate, fără legături și condiționări exterioare. Fiecare componentă este stilizată folosind o foaie de stiluri proprie acesteia.

Pentru fiecare componentă, în documentația bibliotecii de componente este inclusă o descriere și un exemplu de utilizare.

Exemplu de utilizare:

Se consideră fișierul *index.js* având conținutul următor:

index.js

```
import React from "react";
import ReactDOM from "react-dom";
import App from "./App";
ReactDOM.render(<App />, document.getElementById("root"));
```

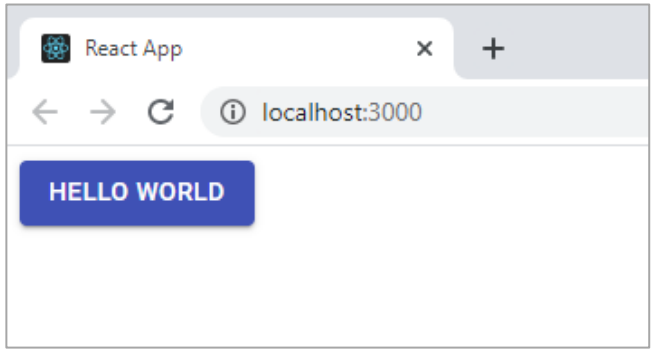
Componenta *<App />* referită în *index.js* este definită astfel:

App.js

```
import React from "react";
import { Button } from '@material-ui/core';
```

```
function App() {
  return (
    <Button variant="contained" color="primary">
      Hello World
    </Button>
  );
}

export default App;
```

Încărcarea aplicației folosind serverul de dezvoltare va produce rezultatul următor:



Observație: Un buton declanșează de regulă o acțiune, deci ar trebui să conțină și un atribut *onClick*:

```
   <Button variant="contained" color="primary" onClick={mesaj}
>
        Hello World
   </Button>
```

Deci în final, componenta *<App />* va avea conținutul următor:

App.js

```
import React from "react";
import { Button } from "@material-ui/core";

function App() {

  const mesaj = () => {
    console.log("Clic");
  };
```

```
  return (
    <Button variant="contained" color="primary" onClick={mesaj
}>
      Hello World
    </Button>
  );
}

export default App;
```

În general privind aceste componente, ele nu diferă de cele utilizate până acum decât prin stilizare și prin proprietățile prin care pot fi configurate. La declararea lor pot fi, desigur, adăugate și proprietăți suplimentare, specifice elementelor echivalente definite în standardul HTML5: *`onClick`*, *`onChange`* sau chiar *`onMouseOver`* sau *`onMouseOut`*.

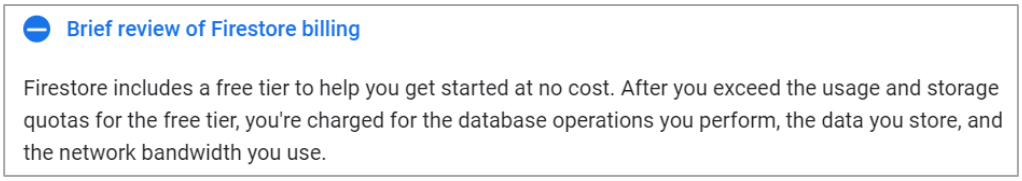
Aplicații React în arhitectură *serverless*

Aspecte generale

O aplicație React poate fi creată și fără o componentă care să se ruleze pe un server propriu dar care să permită totuși utilizarea unui server de baze de date accesibil online. Astfel de aplicații sunt denumite de tip *serverless*. Pentru stocarea datelor, astfel de aplicații fac apel la un *serviciu web* specializat, de tip *DBaaS* (prescurtare de la eng. ***D**ata**B**ase-**A**s-**A**-**S**ervice*). Un serviciu de acest fel frecvent utilizat este *Firebase*, oferit de Google Co.

Firebase include două variante de stocare, *Cloud Firestore* și *Real-time Database*. Diferența esențială dintre cele două este modul de stocare a datelor. *Cloud Firestore* se bazează pe un mod de stocare care respectă regulile specifice bazelor de date nerelaționale, respectiv stochează datele în *documente* conținute în *colecții*.

Observație: *Cloud Firestore* este un serviciu care nu presupune costuri dacă nu este intens accesat:



Păstrarea datelor în baze de date nerelaționale

Documentul

În bazele de date nerelațiomale, unitatea de stocare este *documentul*. Acesta este echivalentul *articolului* din tabelul unei baze de date relaționale.

Fiecare *document* este identificat printr-un *ID* unic și conține un ansamblu de *proprietăți* (sau *câmpuri*) codificate folosind perechi de tip *cheie: valoare*.

Exemplu:

ID = ionescu

```
{
   nume: "Ionescu Mircea",
   ocupatie: "arhitect",
   anNastere: 1988,
   telefon: "+40 745700300"
}
```

Același *document* poate fi scris și folosind pentru proprietatea *numePrenume* o hartă (eng. *map*)

```
{
   numePrenume:
      {
         prenume: "Mircea",
         nume: "Ionescu",
      },
   ocupatie: "arhitect",
   anNastere: 1988,
   telefon: "+40 745700300"
}
```

Colecția

O *colecție* este un *container* conținând mai multe *documente*.

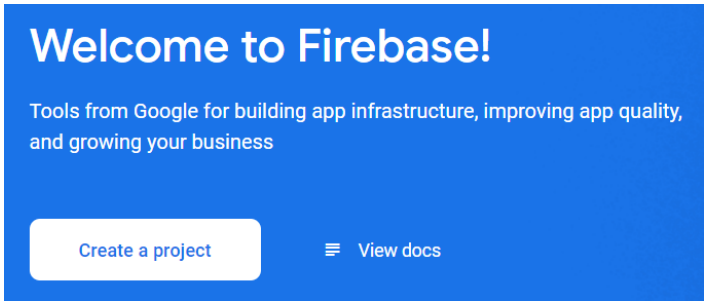
În continuare se prezintă colecția *cursanți*, conținând documentele *ionescu*, *popa* și *avram*:

```
cursanti
  ID="ionescu"
    {
     nume: "Ionescu Mircea",
     ocupatie: "arhitect",
     anNastere: 1988,
     telefon: "+40 745700300"
    }

  ID="popa"
     nume: "Popa Laura",
     ocupatie: "economistă",
     anNastere: 1986,
     telefon: "+40 722543341"
   }

  ID="avram"
    {
     nume: "Avram Dorel",
     ocupatie: "inginer",
     anNastere: 1980,
     telefon: "+40 762120266"
    }
```

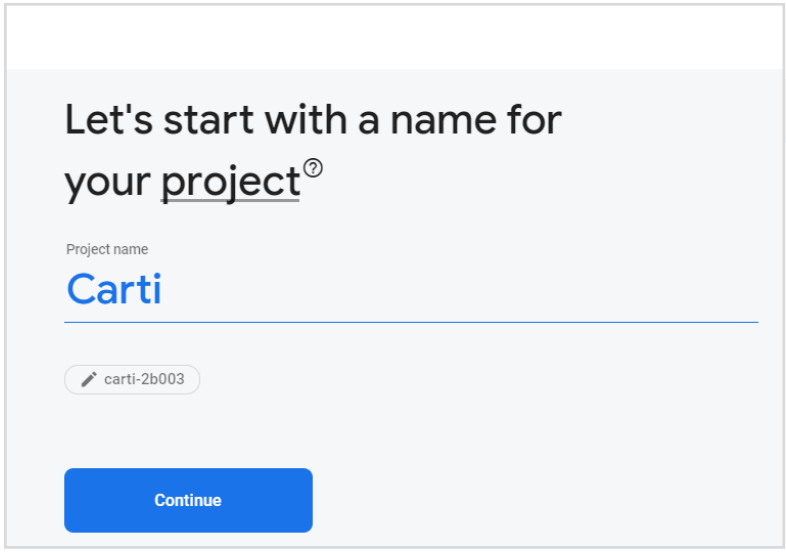
Observație: Documentele din colecția *cursanti* au aceeași structură. Deși într-o bază de date nerelațională acest lucru nu este obligatoriu, din considerente legate de ușurința prelucrărilor ulterioare ale datelor, de regulă se adoptă acest model.

Configurarea bazei de date

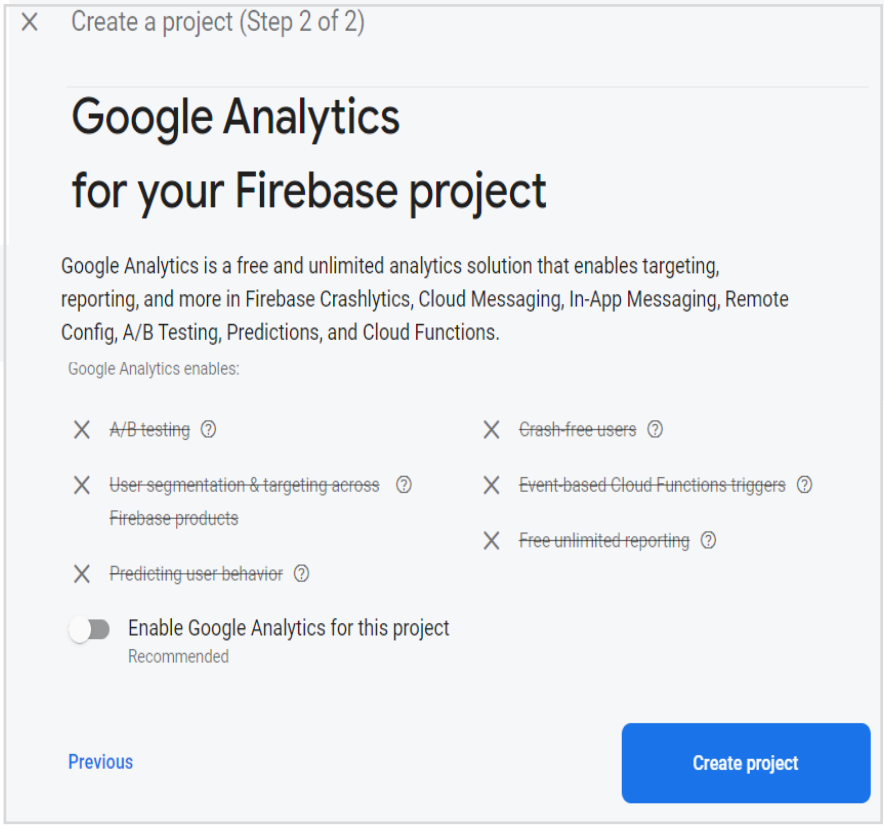
Pentru configurarea bazei de date necesară proiectului *carti* se accesează link-ul *https://console.firebase.google.com/* și apoi *Create a project*:



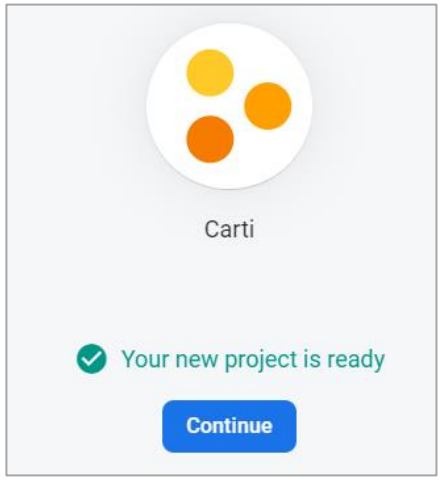
Se tastează apoi denumirea noului proiect (`Carti`):



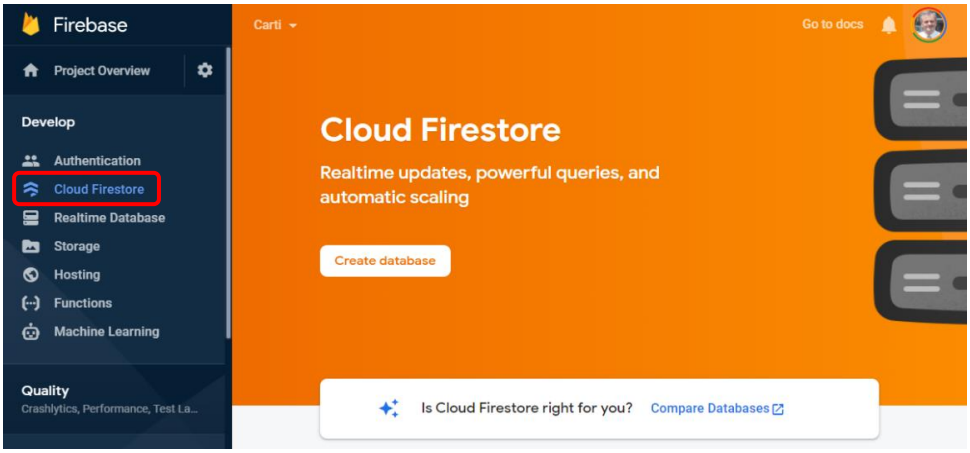
Se selectează apoi butonul `Continue` și aplicația web va solicita acordul pentru a integra `Google Analitics`. Aplicația fiind în dezvoltare, se poate dezactiva opțiunea. Apoi se va selecta butonul `Create project`.


Create a project (Step 2 of 2)
Google Analytics
for your Firebase project
Google Analytics is a free and unlimited analytics solution that enables targeting, reporting, and more in Firebase Crashlytics, Cloud Messaging, In-App Messaging, Remote Config, A/B Testing, Predictions, and Cloud Functions.
Google Analytics enables:
A/B testing
User segmentation & targeting across Firebase products
Predicting user behavior
Crash-free users
Event-based Cloud Functions triggers
Free unlimited reporting
Enable Google Analytics for this project
Recommended
Previous
Create project

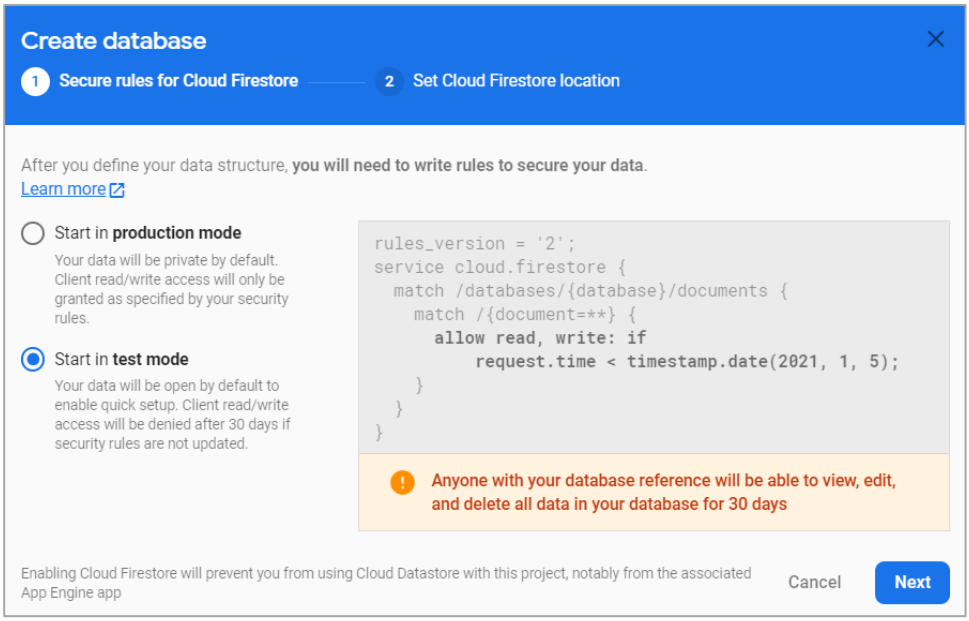
Rezultat:


Carti
Your new project is ready
Continue

Dacă se selectează `Continue`, aplicația web va afișa o consolă de configurare a noului proiect. În meniul afișat în stânga se va selecta opțiunea `Cloud Firestore`. Aceasta va permite crearea pentru aplicatia React `carti`, dezvoltată în capitolele precedente, a unei baze de date nerelaționale.

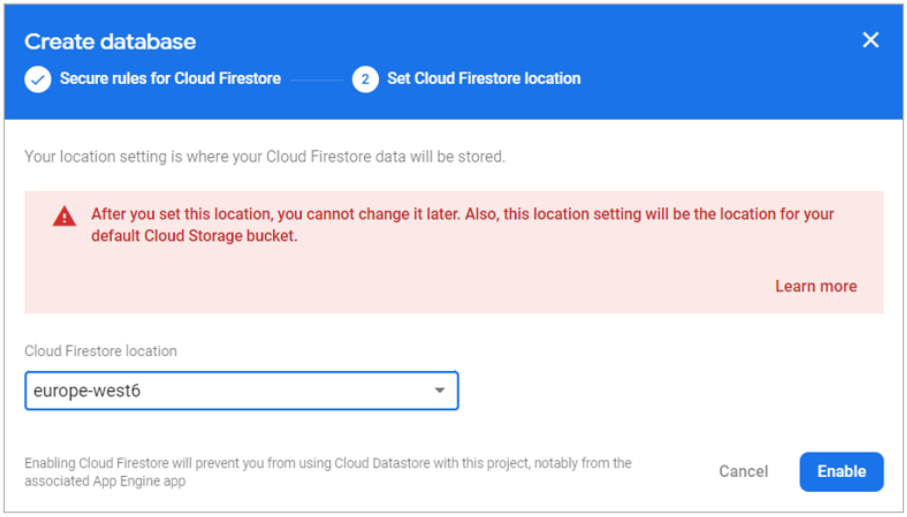


Selectarea opțiunii `Cloud Firestore` va declanșa afișarea unei prime ferestre în care se selectează modul de accesare a noii baze de date. Pentru etapa de dezvoltare se va select varianta `Start in test mode`:

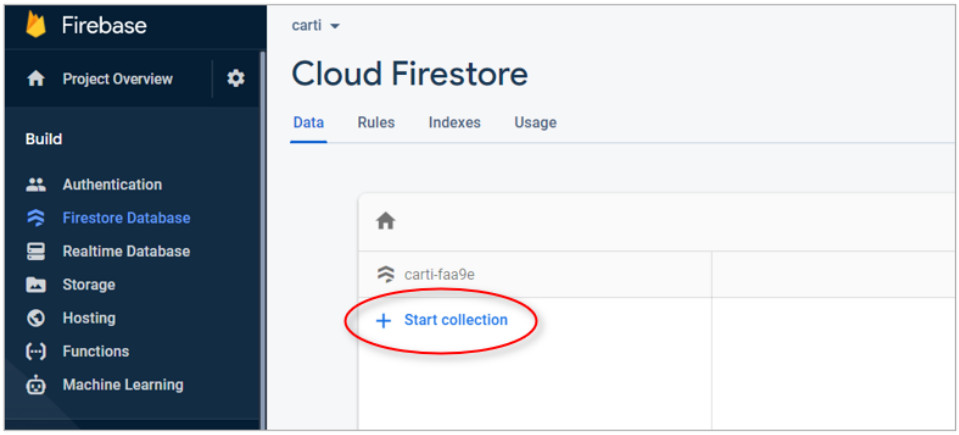


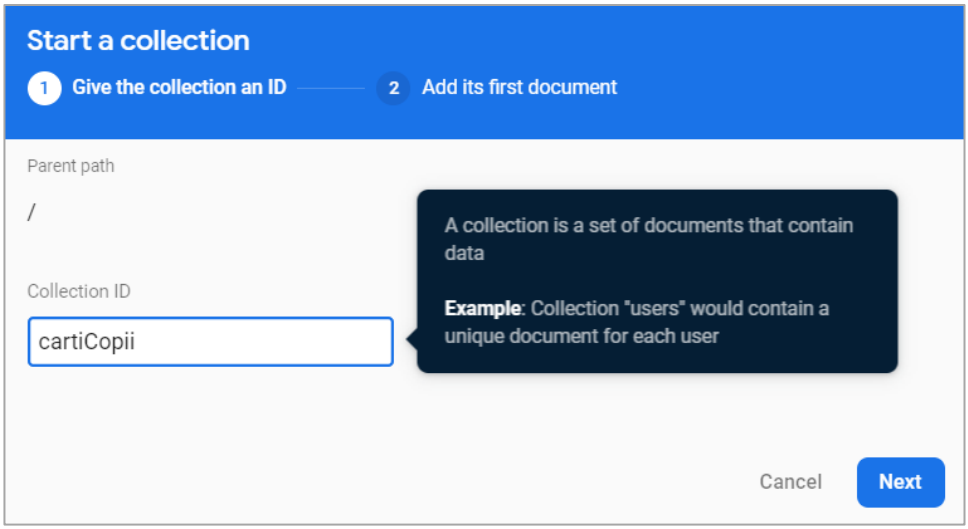
Această variantă permite accesarea noii baze de date de către oricine are referința asociată acesteia.

Următoarea fereastră afișată cere precizarea locului în care va fi stocată baza de date.

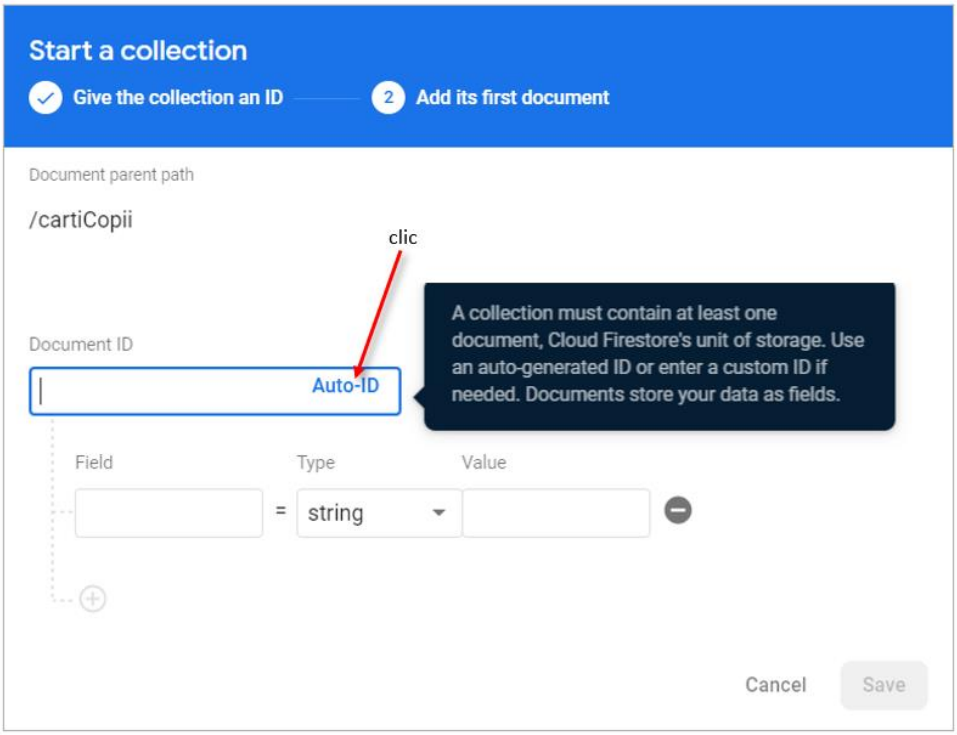


În continuare va fi afișată interfața necesară declarării unei noi colecții. Va fi declarată colecția `cartiCopii` și apoi vor fi inserate în colecția creată trei documente conținând trei dintre cele patru cărți din versiunea precedentă a aplicației React `carti`.

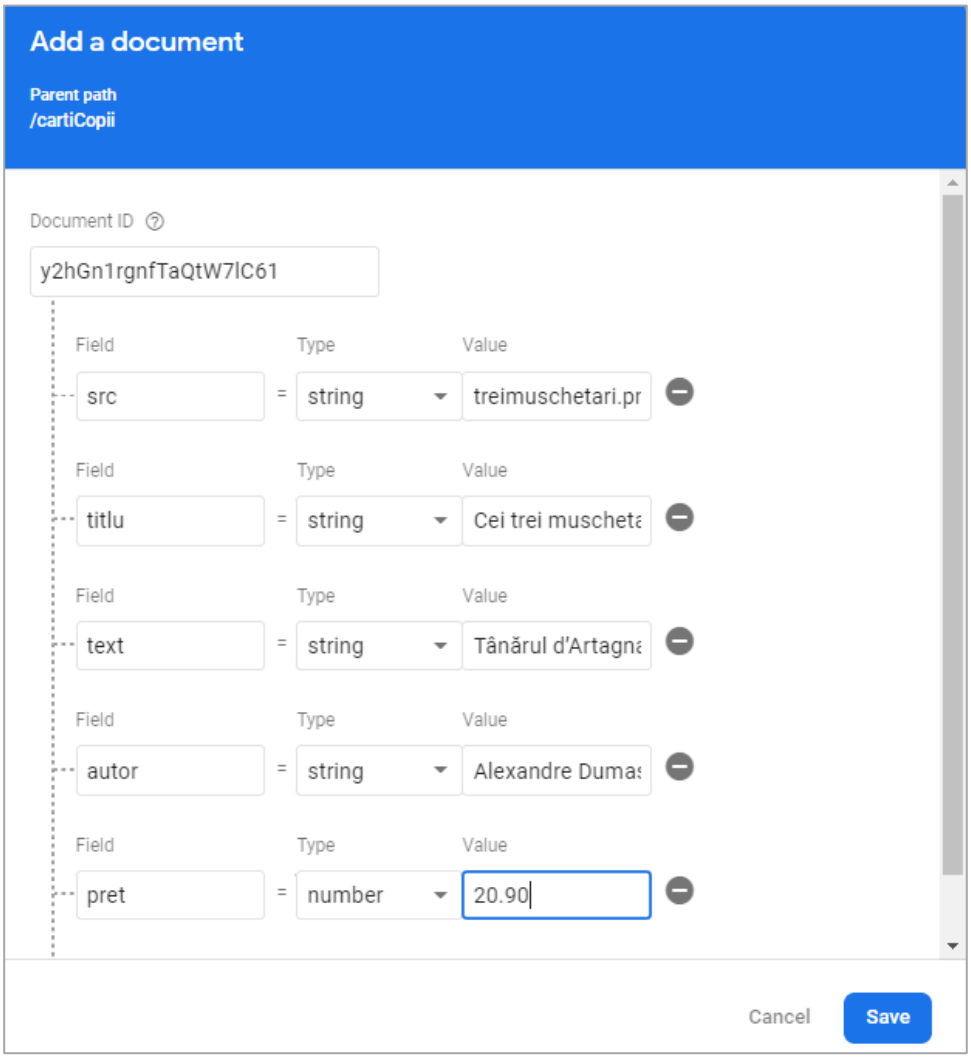




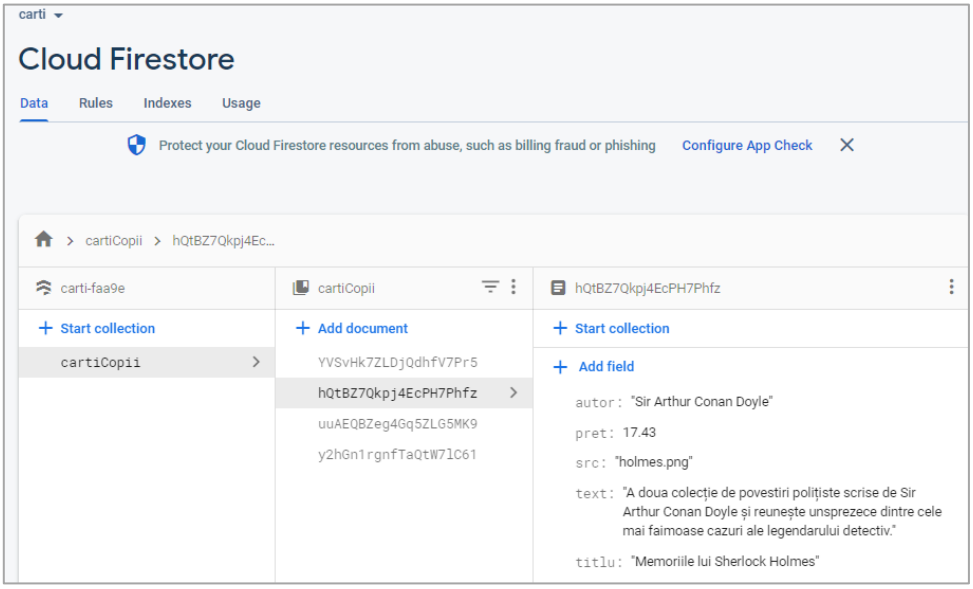
Fiecare document care va fi adăugat colecției trebuie să aibă un `ID` propriu. Generarea acestuia se poate face automat, ca în imagine:



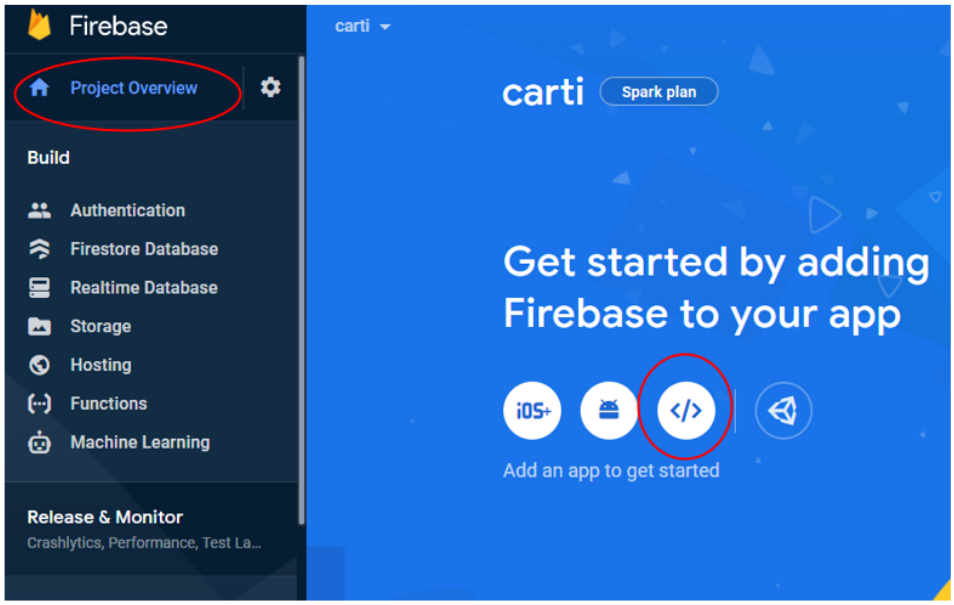
Apoi pot fi adăugate propietățile noului document:



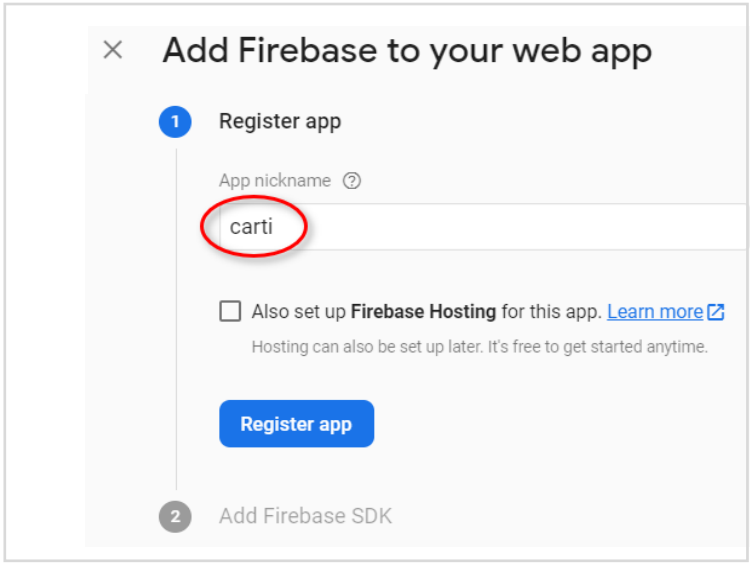
Rezultat:



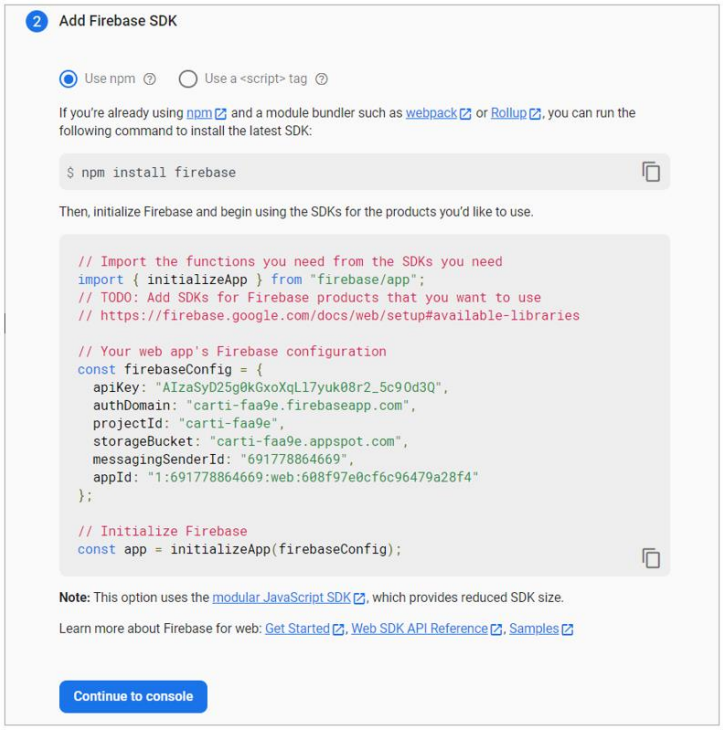
Un ultim pas este copierea codului necesar aplicației web (React) pentru a putea accesa baza de date creată. Pentru aceasta se selectează *Project Overview* și apoi pictograma din imagine:



Aplicația va afișa o fereastră în care se poate da numele aplicației web care va folosi baza de date:



După impunerea denumirii aplicației web vor fi afișate informațiile necesare conectării la baza de date:



Scurtă inițiere în *Cloud Firestore*

Pentru a opera într-o bază de date nerelațională de tip *Cloud Firestore* poate fi accesat site-ul de prezentare a acestei soluții la adresa *https://firebase.google.com/docs/firestore/*.

În continuare vor fi prezentate câteva tipuri de operații.

Observație: Funcțiile utilizate pentru a crea sau a accesa resurse din baza de date *Cloud Firestore* trebuie să se execute asincron. De altfel această regulă se aplică ori de câte ori trebuie accesată o resursă exterioară aplicației.

În cele ce urmează, executarea asincronă a funcțiilor care accesează baza de date creată se va realiza folosind perechea de cuvinte rezervate *async* - *await*.

Adăugarea unui nou document

Crearea într-o colecție a unui nou *document* al cărui *ID* este generat automat se realizează astfel:

```
import { collection, addDoc } from "firebase/firestore";

    const adaugCarte = async (carte) => {
        // Adaug un nou document folosind un ID generat automa
t.
        const docRef = await addDoc(collection(db, "cartiCopii
"), carte);
        console.log("Document adaugat cu ID: ", docRef.id);
    };
```

Obiectul *carte* ar putea avea următorul conținut:

```
{
    src: "printcersetor.png",
    titlu: "Prinț și cerșetor",
    text: "Una dintre cartile de neuitat ale copilariei, capod
operă a literaturii din toate timpurile."
    autor: "Mark Twain",
    pret: 13.41
}
```

Citirea documentelor

Citirea unei colecții se realizează folosind `getDocs()`. Exemplu:

```
import { getFirestore, collection, getDocs } from
              "firebase/firestore";
    ...
    const db = getFirestore(app);

    const getLista = async () => {
        const listaCarti =
                  await getDocs(collection(db, "cartiCopii"))
;
        //  Urmează exploatarea colecției
        let listaNoua = listacarti.docs.map((doc) => {
            console.log(`Document gasit: ${doc.id}`);
            return doc.data();
        }

        return listaNoua;
    }
```

Sau, daca se caută documente care satisfac un anumit criteriu, se poate adăuga și o condiție de filtrare:

```
import { collection, query, where, getDocs } from
              "firebase/firestore";
    ...
    const db = getFirestore(app);
    ...

    const cartiAutor = async (autor) => {
        const caut = query(collection(db, "cartiCopii"),
                 where("autor", "==", autor));
        const listacarti = await getDocs(caut);
        //  Urmează exploatarea colecției
        let listaNoua = listacarti.docs.map((doc) => {
            console.log(`Document gasit: ${doc.id}`);
            return doc.data();
        }

        return listaNoua;
    }
```

Mai multe variante de filtrare pot fi găsite accesând adresa:

https://firebase.google.com/docs/firestore/query-data/queries

Ștergerea unui document

Ștergerea unui document presupune cunoașterea *ID*-ului acestuia Exemplu:

```
import { doc, deleteDoc } from "firebase/firestore";
    ...
    const db = getFirestore(app);
    ...

    const stergeCarte = async (id) => {
        await deleteDoc(doc(db, "cartiCopii", id));
    }
```

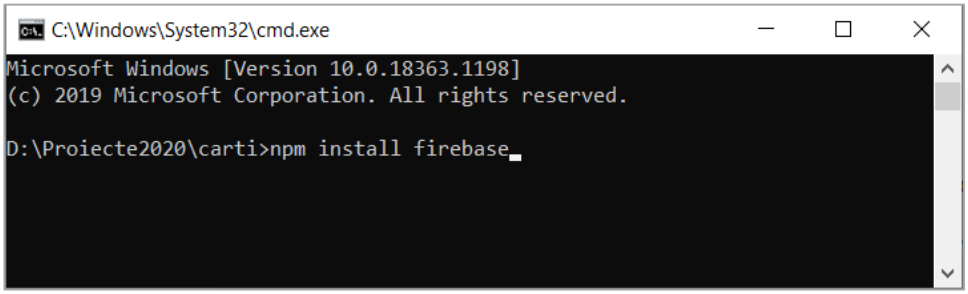
Integrarea bazei de date *Cloud Firestore* "*Carti*" în aplicația *carti*

Pentru a putea utiliza baza de date creată va trebui adăugat aplicației React un modul suplimentar, *firebase*. Pentru aceasta, într-o fereastră *Command* lansată în execuție în directorul rădăcină al proiectului (*carti*) se va da comanda:

```
  npm install firebase
```

Sau, dacă dorim și specificarea versiunii serverului *Firebase* utilizat, se poate folosi comanda:

```
  npm install firebase@9.2.0 --save
```



Apoi se va adăuga în subdirectorul `/src` a proiectului fișierul `init.js` având următorul conținut:

`init.js`

```
import { initializeApp } from "firebase/app";

// Your web app's Firebase configuration

const firebaseConfig = {
    apiKey: "AIzaSyD25g0kGxoXqLl7yuk08r2_5c9OD4Q",
    authDomain: "carti-faa9e.firebaseapp.com",
    projectId: "carti-faa9e",
    storageBucket: "carti-faa9e.appspot.com",
    messagingSenderId: "691778864669",
    appId: "1:691778864669:web:608f97e0cf6c96479a28f4"
  };

const app = initializeApp(firebaseConfig);

  export default app;
```

Observație: Conținutul acestui fișier a fost furnizat de aplicația web folosită la crearea bazei de date, în etapa anterioară, și servește la inițializarea bazei de date realizată în finalul fișierului `init.js`:

```
  const app = initializeApp(firebaseConfig);
```

Citirea datelor din baza de date *Carti*

Pentru citirea tuturor documentelor din colecția `cartiCopii` va fi apelată funcția `getLista()`. Aceasta trebuie să se execute asincron deoarece accesează o resursă exterioară aplicației. Funcția `getLista()` poate fi scrisă astfel:

```
    const getLista = async () => {
        const listacarti = await getDocs(collection(db, "carti
Copii"));
        let listaNoua = listacarti.docs.map((doc) => {
          let carte = doc.data();  //  Creez un obiect nou
          carte.src = `imagini/${carte.src}`;  //  Corectez ca
lea
          carte.id = doc.id;      // adaug ID-ul în obiectul "
carte" (trebuie!)
```

```
          return carte;
        });
        setLista(listaNoua);   //  Actualizez obiectul "state"
  };
```

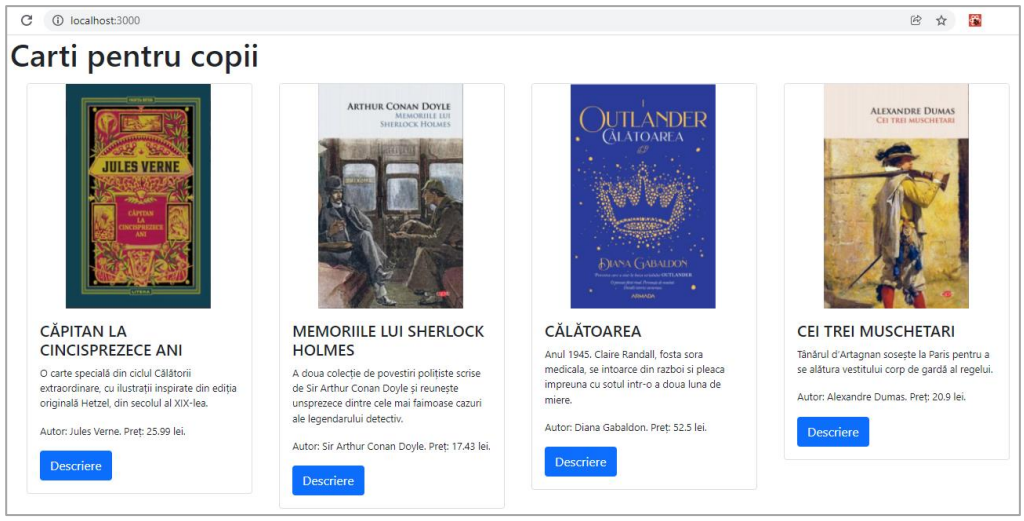
Componenta `<App />` trebuie acum rescrisă astfel:

App.js

```
import React, { useState, useEffect } from 'react';
import { Container } from "react-bootstrap";
import ListaCarti from "./listacarti";
import Adaug from './adaug';
import { getFirestore, collection, getDocs } from "firebase/fi
restore";
import app from "./init";

const App = () => {
  const [lista, setLista] = useState([]);

  const db = getFirestore(app);

  const getLista = async () => {
    const listacarti = await getDocs(collection(db, "cartiCopi
i"));
    let listaNoua = listacarti.docs.map((doc) => {
      let carte = doc.data();  //  Creez un obiect nou
      carte.src = `imagini/${carte.src}`;  //  Corectez calea
      carte.id = doc.id;      // adaug ID-ul în obiectul "cart
e" (trebuie!)
      return carte;
    });
    setLista(listaNoua);   //  Actualizez obiectul "state"
  };

  useEffect(() => {
    getLista();
  }, []);

  return (
    <>
      <Container>
        <h1>Carti pentru copii</h1>
      </Container>
      <ListaCarti listaCarti={lista} />
    </>
  );
}
```

```
export default App;
```

Rezultat:



Adăugarea unui document nou

Adăugarea unui document în colecția *cartiCopii* se va realiza folosind funcția *adaug()*. *ID*-ul noului document va fi creat automat.

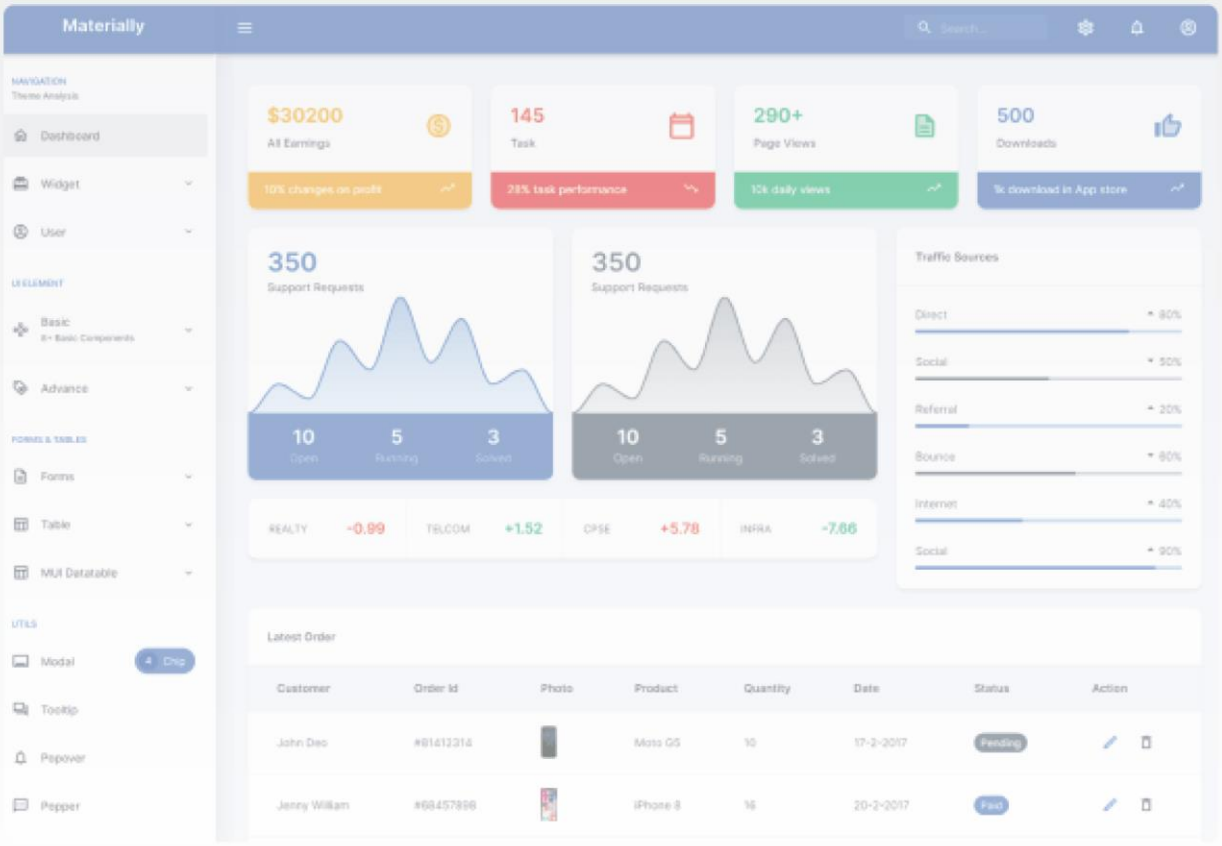
```
    ...
    import { getFirestore, collection, getDocs, addDoc } from
"firebase/firestore";
    ...
    const adaug = async (carte) => {
        // Adaug un nou document folosind un ID generat automa
t.
        const docRef = await addDoc(collection(db, "cartiCopii
"), carte);
        getLista();  //  Reafisez lista
        console.log("Document adaugat cu ID: ", docRef.id);
    };
```

Observație: Datorită faptului că la citirea cărților din baza de date se adaugă în fața denumirii imaginii copertei calea ("*imagini/*"), funcția

tratezSubmit() din componenta *<Adaug />* trebuie modificată astfel încât să nu mai adauge aceeași cale.

```
    const tratezSubmit = (evt) => {
        evt.preventDefault();
        const carte = {src, titlu, text, autor, pret};
        props.transmit(carte);  //  Transmit obiectul carte
        //  Golesc controalele formularului
        setSrc("");
        setTitlu("");
        setText("");
        setAutor("");
        setPret("");
      }
```

Varianta finală a proiectului poate fi accesată la adresa: *https://github.com/mdamian2020/carti_Fire*.



Editura U.T.PRESS