

**Universitatea din Bucuresti**  
Facultatea de Matematica si Informatica

# **Lucrare de licenta**

## **RankArtist**

Platforma web pentru ierarhizarea colectiva de idei sau obiecte

**Profesor coordonator:**

CLAUDIA MURESAN

**Absolvent:**

ALEXANDRU-MARIAN FLORESCU

# Cuprins:

- [Prezentarea aplicatiei](#)
- [Nodejs](#)
- [Nodejs Express](#)
- [MongoDB si Mongoose](#)
- [HTML5](#)
- [CSS3](#)
- [Javascript](#)
- [React](#)
- [Problema Modal-ului](#)
- [Styled-Components](#)
- [JWT](#)
- [Auth0](#)
- [Redux](#)
- [App](#)
- [Navbar](#)
- [Main](#)
- [Colorticks](#)
- [ItemsRank](#)
- [Sectiunea de comentarii](#)
- [Categories](#)
- [CSS3 FlexBox](#)
- [Ratia de aur](#)
- [Bibliografie](#)

## Prezentarea aplicatiei

RankArtist este o aplicatie web cu o singura pagina, dezvoltata folosind tehnologii moderne. Motivatia principala a aplicatiei este aceea de atestare a capacitatilor de dezvoltare ale subsemnatului. Aplicatia este construita in jurul functionalitatii de ierarhizare comuna a unor liste de obiecte sau idei.

Scopul acesteia este atat pastrarea unei ordini in continutul personal (prin crearea de categorii private ex.: Cele mai bune poezii pe care le-am scris vreodata!, Cele mai bune proiecte ale mele!), cat si acela de a cumula aceste viziuni cu ale altor utilizatori creand ierarhii coerente pentru unele dintre cele mai comune categorii de obiecte sau idei cu care ne intalnim in fiecare zi.

Dezvoltarea s-a facut respectand unele dintre cele mai importante standarde moderne si folosind tehnologii de ultima generatie. Astfel, aplicatia este sustinuta de un server Nodejs. Acesta este si API-ul care acceseaza baza de date MongoDB, cu ajutorul unui model de schematizare Mongoose. La nivel de front-end s-a folosit cu succes React pentru definirea componentelor vizuale. Structurarea lor s-a facut prin HTML5, impreuna cu JSX. Pentru stilizare am folosit CSS3 pur, fara a recurge la vre-un framework de stilizare de genul Bootstrap. Motivul pentru aceasta este dorinta si placerea de a dezvolta ceva cu un aspect original cat si imbunatatirea abilitatilor mele creative. Pentru administrarea starilor si datelor in general, am folosit Redux, acesta fiind un partener foarte bun pentru React.

Prezenta lucrare isi propune sa insoteasca aplicatia, sa fie un ghid in ceea ce priveste functionalitatea cat si un manual explicativ vis-a-vis de tehnologiile folosite. Modul de redactare este propriu, orientat pe exemple clare. Vetii gasi adesea imagini descriptive ori secvente de cod in jurul carora voi face comentariile necesare.

Voi incerca sa redactez continutul astfel incat sa poata fi inteles si de cineva mai putin familiar cu tehnologiile web. Ca ideal, incerc sa construiesc un ghid de intrare in dezvoltarea web in speranta ca experienta sa-mi foloseasca in vederea unei viitoare cariere didactice.

Pentru intelegerea mai buna a proiectului vom discuta, pe rand, tehnologiile folosite in dezvoltarea ei

## NodeJs [1]

Este o platforma de dezvoltare open source pentru executia de Javascript pe partea de server.

Acesta este unul dintre cele mai moderne framework-uri de dezvoltare. Modul lui de functionare este ghidat de evenimente, asincron. Design-ul sau il face ideal pentru construirea aplicatiilor scalabile. Ce este definitoriu pentru el este faptul ca nu blocheaza Input/Output-ul. Pentru creatorul Node, acesta este principalul avantaj, alte voci sunt insa mai critice, intrucat considera aplicatia susceptibila la blocaje ca urmare a alocarii prea multor cicluri de procesor unui singur proces.

```
const http = require('http');

const hostname = '127.0.0.1';
const port = 3000;

const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World\n');
});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

Mai sus, se poate vedea cum un exemplu simplu poate sustine mai multe conexiuni simultan. Iar in lipsa de, serverul este pur si simplu inactiv.

## NodeJs Express [2]

Este un framework minimal si flexibil, construit pe Node.js pentru dezvoltarea de aplicatii web si mobile. Cu foarte putin efort, avem la dispozitie un API robust si flexibil capabil sa raspunda oricarui tip de cerinte. Atasarea de middleware-uri care sa proceseze sau sa filtreze procesele noastre este foarte usoara.

Express este construit dintr-un strat subtil de functii si functionalitati, astfel fiind foarte util fara a obstructiona Node.js-ul pe care il stim deja.

Datorita simplitatii framework-ului, cred ca cea mai buna metoda de prezentare este printr-un exemplu clar, luat chiar din codul sursa al aplicatiei noastre.

```
1  var app = express();
2  var compiler = webpack(config);
3
4  app.use(bodyParser.json());
5  app.use(webpackDevMiddleware(compiler, {noInfo: true, publicPath: config.output.publicPath}));
6  app.use(webpackHotMiddleware(compiler));
7
8  var port = 3000;
9
10 app.get('*', function (req, res) {
11   res.sendFile(path.resolve('client/index.html'));
12 });
13 app.listen(port, function(error) {
14   if (error) throw error;
15   console.log("Express server listening on port", port);
16 });
17
```

Avem aici un exemplu perfect functional de API creat folosind ExpressJs. Dupa cum se poate vedea, aplicarea de middleware-uri se face prin `app.use()`. Aceasta se poate face in asa fel:

```
45 app.get('/api/getCategories', checkJwt, function(req, res) {
46   Categories.getCategories((err, category)=>{
47     if(err){
48       throw err;
49       res.json(err);
50     }
51     res.json(category);
52   })
53 })
```

In acest exemplu `checkJwt` este o functie constanta definita anterior in care se verifica signatura JWT-ului, cat si permisiunile asociate acestuia, daca este cazul. In cazul requesturilor care trec de verificarea din middleware, este pur si simplu apelat callback-ul. Acesta, prin Mongoose, se conecteaza la baza de date MongoDB si returneaza datele in raspuns.

Se poate observa aici caracterul asincron si neblocant I/O al Node. Se pot face oricate request-uri la adresa `/api/getCategories` fara sa se astepte un raspuns. Acesta pur si simplu dirijeaza datele.

## Mongoose si MongoDB

MongoDB este o baza de date NoSQL. Asta inseamna ca stocarea datelor nu se face sub forma unor tabele, ca intr-o baza de date relationala, ci sub forma unor documente JSON.

Bineinteles, aceste documente nu au o structura fixa, ci una dinamica. Practic, elementele nu trebuie sa respecte nicio forma, iar singurul camp ce se adauga automat este indexul.

Mongoose este o librerie ce permite modelarea de obiecte din structuri MongoDB. Acesta se leaga la colectia de date a Mongo, si prelucreaza datele pe schema definita de utilizator. Mongoose vine cu o suita de metode predefinite care inlesnesc prelucrarea datelor.

Prin inlantuirea apelurilor, datele pot circula foarte usor intre baza de date si orice client care apeleaza ruta API-ului, fie el web, mobile sau chiar desktop.

Acum ca am analizat putin partea de server a aplicatiei, independenta de platforma de utilizare a clientului, ne vom apleca putin asupra tehnologiilor Web folosite.

## HTML5

Hypertext Markup Language este un limbaj de structurare a paginilor web, extrem de cunoscut si folosit. Principalul sau avantaj este simplitatea, care permite sa fie invatat foarte usor. Acesta este recunoscut de toate browserele uzuale.

Acest marcaj spune browserelor cum sa afiseze continutul paginilor web. Fiecare tag html are un rol destul de clar si ansamblul lor contureaza modul in care se afiseaza datele, cu un minim de stilizare.

De exemplu, tag-urile `<h1></h1>` vin cu un font mai mare decat `<p></p>`. Cazurile `<ul><li></li></ul>` au bullet-point-uri implicit. Desi aceste styling-uri au un o logica rezonabila, ele sunt foarte limitate si variaza destul de mult intre browsere. Din acest motiv, majoritatea programatorilor aleg sa reseteze toate styling-urile imediat ce incep un proiect nou.

## CSS3

Ceea ce ofera designerilor web atata libertate de creatie este gama larga de selectori si proprietati pe care le furnizeaza Cascading Style Sheets (CSS). Rolul acestuia este de a aseza prezentarea paginilor web.

Separarea prezentarii de continut este definitorie pentru dezvoltarea aplicatiilor web si prezinta cateva avantaje clare:

1- Tematizarea. O pagina web poate avea diferite stiluri fara a afecta structura marcajului. Asta a permis aparitia unor framework-uri precum bootstrap, care ofera teme vizuale foarte accesibile.

2- Design. Este diferentiat pentru diferite conditii de randare. Acestea pot fi: dimensiuni diferite ale display-urilor, pagini printabile sau pagini pentru persoane cu dizabilitati.

3- Rapiditatea. Modificarile se pot face si aplica foarte usor tuturor tagurilor vizate de selectorii respectivi. Astfel schimbarea `h1 {color: white}` in `h1 {color: red}` schimba culoarea tuturor tag-urilor h1 la care se aplica acest styling.

Pentru cazurile ambigue de stilizare, CSS furnizeaza cateva metode de prioritizare. Ratingul unui selector se calculeaza cu valorile de mai jos [3]. Astfel, `ul li {}` are un rating de 2, in timp ce `.navigare li {}` are un rating de 11 si astfel, intaietate.

**1,000**

inline style

**100**

ID

**10**

class

**1**

element

attribute

pseudo-  
element

pseudo-  
class

Aici putem vedea scorurile cu care se calculeaza specificitatea.

O alta metoda de prioritizare o reprezinta '!important'. Acesta scoate ratingurile pentru o anumita proprietate si o face pe aceasta absoluta. Daca doua proprietati cu !important se combat, atunci castiga cea cu rating-ul mai mare.

Toate aceste metode fac CSS o unealta foarte importanta in mainile oricarui Web Designer. Desi relativ usor de manuit, desaga de trucuri este virtual nelimitata, facand posibila implementarea oricarui stil dorit.

Pe langa infrumusetarea statica, CSS ofera si o gama de metode de animatie destul de complexe pentru aplicatiile Web. Folosind proprietatea 'transition' o gama larga de schimbari de stare se pot anima. Astfel schimbarea se face pe o durata data si cu o anumita functie de timing.

Ex: transition: all ease-in-out 0.1s;

CSS ofera de asemenea @Keyframes pentru desemnarea animatiilor mai complexe. Acestea permit definirea starilor tranzitiei ca in exemplul[5]:

```
div {  
    width: 100px;  
    height: 100px;  
    background: red;  
    position: relative;  
    -webkit-animation: mymove 5s infinite;  
    animation: mymove 5s infinite;  
}  
  
@keyframes mymove {  
    0% {top: 0px;}  
    25% {top: 200px;}  
    75% {top: 50px;}  
    100% {top: 100px;}  
}
```

## Javascript

Este un limbaj de scripting dinamic, interpretat si aduce la viata paginile web structurate cu HTML si stilizate cu CSS. Acesta poate functiona atat ca un limbaj orientat pe obiecte, cat si procedural.

Rolul sau este de a altera in timp real continutul paginilor web. Folosind diferite librarii, JS se poate lega de DOM-ul paginii si aduce modificari acestuia. Poate face apeluri la diverse API-uri pentru furnizarea de date. Capacitatile limbajului sunt virtual nelimitate.

Limbajul este atat de robust si popular, incat a fost introdus si pe partea de server, desi fusese gandit exclusiv pentru client. Exista chiar si anumite framework-uri care il folosesc pentru dezvoltarea aplicatiilor Desktop. Electron, de exemplu foloseste JS si cu el s-a construit unul dintre cele mai populare editoare text: Atom.

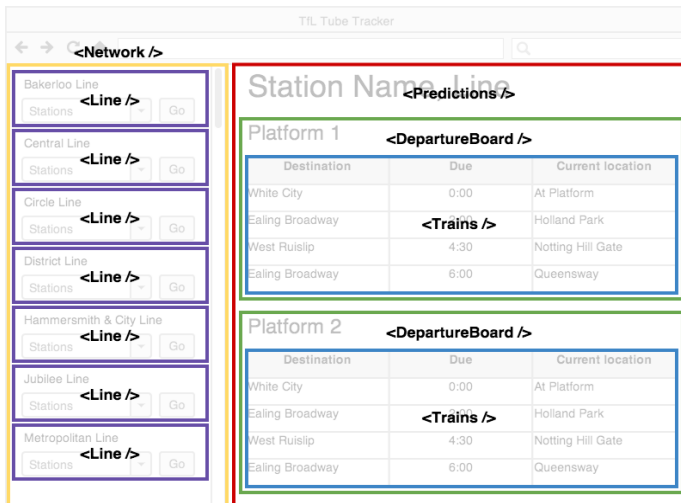
Cea mai simpla metoda de a utiliza JS este intre tag-urile html <script></script>. In majoritatea cazurilor insa, codul este scris intr-un fisier separat si doar importat in documentul cu pricina. Pentru proiectele mai complexe, sau care folosesc anumite framework-uri, codul este distribuit prin metode similare limbajelor de dezvoltare desktop.

## React [4]

Este un framework JS de creare a unor interfete composabile. React sparge view-urile in componente UI reutilizabile si le prezinta pe acestea intr-un ansamblu ghidat de o arhitectura compozitionala arborescenta.

React nu foloseste template-uri. Abstractizarea se face folosind JS si JSX(o extensie de limbaj care imбина HTML cu Javascript). Avantajele acestei metode sunt evidente. Javascript, fiind un limbaj de programare robust si flexibil, face abstractizarea datelor foarte usoara in orice context.

Acelasi lucru poate fi spus si despre extinderea sau mentenanta codului. De asemenea, elimina necesitatea de a concatena string-uri si scade vulnerabilitatea la atacuri de tip XSS.



Fiecare componenta React este un Obiect JS. Acesta are, implicit, cateva metode de life-cycle care sunt apelate in ordine, permitand procesarea datelor inainte sau dupa randarea componentelor. Cele mai usuale 3 metode sunt: `componentWillMount()`, `componentDidMount()` si `render()`.

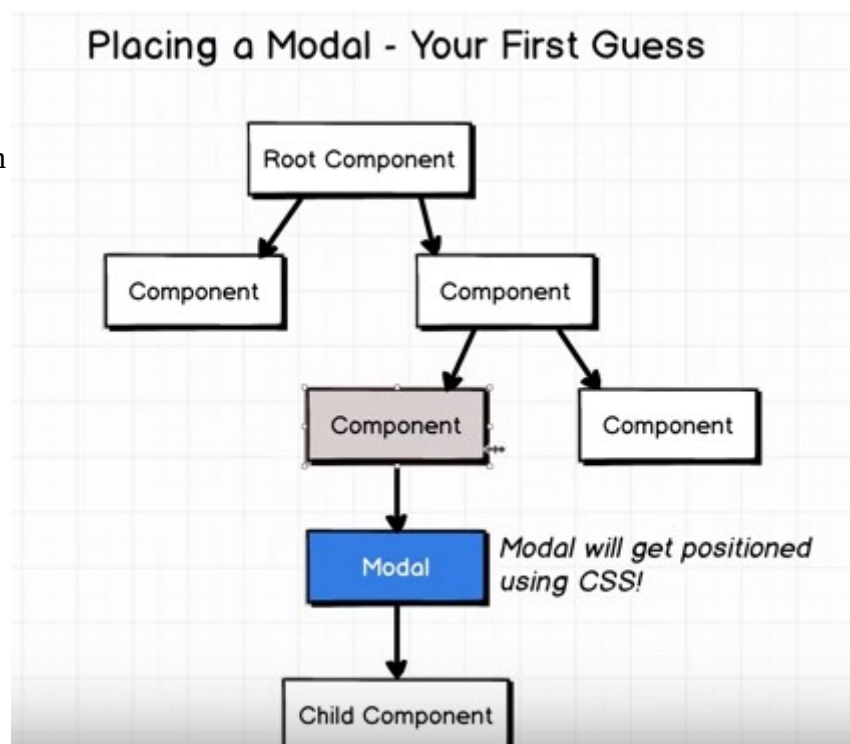
`render()` este metoda in care se returneaza codul HTML pe care il va randa clientul. React parcurge arborele componentelor si randeaza la fiecare, in functie de logica data, continutul JSX corespunzator.

Astfel, capacitatea de design a structurii unei pagini nu este in niciun fel redusa, doar constransa in arhitectura arboriscenta. Acesta este preponderent un avantaj, insa prezinta si cateva neajunsuri pe care un dezvoltator le poate intampina.

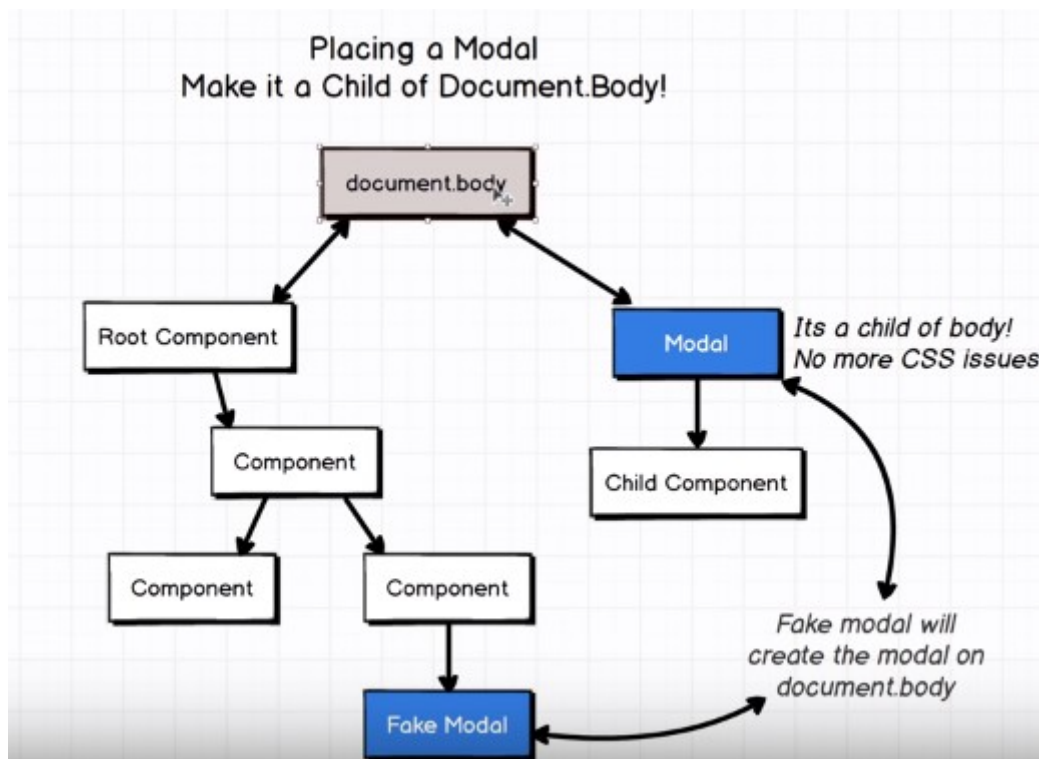
## Problema Modalului

O prima solutie se poate vedea langa [7]. La prima vedere aceasta functioneaza. Putem, pur si simplu sa creem modalul ca un copil al componentei care il randeaza, dar, pe masura ce aplicatia creste in complexitate, exista sanse mari sa intampinam probleme vis-a-vis de z-index.

Fara a garanta prezenta lui in prim-plan, modalul isi pierde rolul de a atrage un raspuns imediat din partea utilizatorului.



O alta solutie este aceea de a crea un modal fals care sa injecteze cod in <body>. Aceasta abordare, desi corecta, incalca conventia React, dar elimina astfel problema afisarii continutului. De altfel, aceasta abordare este mai aproape de un modal traditional. Ciudatenia este ca, de regula, o aplicatie React este formata numai din descendentii ai unei componente Root. In exemplul de mai jos insa , se poate vedea cum adevaratul Modal este de fapt fratiul sau nelegitim. Un fel de John Snow.



Aceasta interventie se poate face foarte usor:

```

componentDidMount() {
  this.modalTarget = document.createElement('div');
  this.modalTarget.className = 'modal';
  document.body.appendChild(this.modalTarget);
  this._render();
}

```

Este necesara redefinirea unei metode de randare, dar acesta nu este un mare impediment si, ceea ce se obtine este un modal perfect reutilizabil ulterior. Datele se pot trimite in continuare in flux, insa este posibil ca anumite Provider-e sa trebuiasca redefinite si pentru acea ramura.

In exemplul de mai jos putem vedea metoda \_render() implementata. Aici, this.props.color este furnizat de catre magazia Redux, insa ThemeProvider-ul trebuie redefinit in contextul acesta. In plus, se poate vedea structura modalului.

Observam reusabilitatea lui. Componenta Modal, doar defineste forma in care, prin {this.props.children} se poate randa orice altceva. Astfel, paradigma React de componente reutilizabile este respectata.

```

componentWillReceiveProps(nextProps) {
  if (nextProps.isActive !== this.props.isActive) {
    this.modalTarget.className = nextProps.isActive ?
      'modal is-active'
      : 'modal';
  }
}

```



Un aspect interesant este la eventul `onClick` atasat componentei `X`. Aceasta este o functie primita de la componenta parinte si reprezinta exact trigger-ul care a declansat aparitia modal-ului pe ecran. Trigger-ul, din parinte, schimba proprietatea componentei curente si astfel se apeleaza metoda de life-cycle de mai sus.

```
render() {
  const theme={
    | color: this.props.color
  }
  let element = (
    | <ThemeProvider theme={theme}>
    | | <Overlay>
    | | | <Wrapper>
    | | | | <div className="upperStrip">
    | | | | | <StripHeader> Add new Item </StripHeader>
    | | | | | <X onClick={this.props.show} src={Cancel}/>
    | | | | </div>
    | | | | {this.props.children}
    | | | </Wrapper>
    | | </Overlay>
    | </ThemeProvider>
  );
  ReactDOM.render(element, this.modalTarget);
}
```

In privinta styling-ului, React nu ingradeste posibilitatea de a avea un External Style-Sheet in respectiva pagina Web, inasa ofera cateva alternative.

## Styled-components

Este o librarie pentru React ce permite stilizarea componentelor intr-un mod foarte util si interesant. Am discutat deja despre cum React incurajeaza construirea de componente reutilizabile. Ei bine, metoda unui singur stylesheet nu este tocmai ideala aici, asa ca industria a construit cateva alternative.

O prima solutie a fost separarea continutului de stil pentru fiecare componenta sau modul. Totusi pastrarea `className`-ului si definirea lui manuala in documentul CSS este relativ susceptibila la eroare umana. Pasul urmator a fost aparitia `css-modules`. [8]

```
import Styles from "./Component.less";

class Button extends React.Component {
  render() {
    return <span className={Styles.btn}>{this.props.children}</span>
  }
}
```

Aceasta abordare este mai sigura si mai ordonata pentru codul nostru. Practic, fiecare componenta ar avea un fisier de style atasat. Daca adaugam o limbrarie aditionala precum classnames, se pot administra foarte bine schimbarile de stari. Un programator bun poate fi foarte multumit de aceasta metoda de lucru atata timp cat se respecta conventiile de buna practica.

Exact aici intervine styled-components. Minunata librerie face un pas in fata. Aceasta obliga, practic, la respectarea acestor conventii prin faptul ca le face insasi metoda de lucru. Libraria foloseste cu gratie noua functionalitate a ES6: Template literals (``)

```
const Button = styled.button`
  width: 100%;
  color: ${props => props.theme.color};
  padding: 10px 20px;
  font-size: 16px;
  text-align: center;
  letter-spacing: .1em;
  border-radius: 7px;
  border: 1px solid ${props => props.theme.color};
  background-color: white;
  margin-top: 5px;
  box-shadow: 0px 17px 10px -10px rgba(0,0,0,.33);
  transition-property: background-color;
  transition-duration: 0.5s;
  transition-timing-function: ease-out;
  &:hover {
    background-color: ${props => props.theme.color};
    color:white;
  }
  &:active{
    color: white;
    background-color: ${props => props.theme.color};
    box-shadow: inset 1px 1px 3px 2px rgba(0, 0, 0, .3);
  }
`;
export default (props) =>
{
  return (<Button {...props}>{props.children}</Button>)
};
```

Astfel, intreaga componenta Button este definita o singura data in intreaga aplicatie si poate fi refolosita la nesfarsit. Erorile de stil sunt foarte usor de identificat si corectat, data fiind structura arborescenta a React, daca butonul nu este corect randat, stim unde si ce trebuie verificat, fara prea mare efort. Folderul cu sursa este de asemenea tinut curat si ingrijit. Toate astea fara a sacrifica niciun pic de functionalitate. Styled-components suporta orice, inclusiv Keyframes si selectori complexi.

O alta functionalitate draguta este `ThemeProvider`-ul care vine cu biblioteca. Odata inclus in arborele de componente, toti descendetii lui au acces, in definirea stilistica, la `{props => props.theme.[campuri]}` in care se afla toate campurile definite, aferente temei alese. Acest lucru face tematizarea aplicatiilor React foarte usoara si placuta si este, dupa parerea mea, un feature foarte puternic. Simpla schimbare a grilei de culori poate mentine aparenta de noutate a site-ului, ofera utilizatorului sansa de a-si customiza experienta si il tine pe acesta angrenat mai mult timp.

```
render() {
  const theme={
    color: this.props.ui.color
  }
  return (
    <ThemeProvider theme={theme}>
      <div className="app">
        <NavBar auth={auth} actions={{changeColor:this.props.actions.changeColor, logOff:this.props.actions.logOff}}/>
        <Main {...this.props} auth={auth}/>
      </div>
    </ThemeProvider>
  )
}
```

Cireasa de pe tort putem spune ca este biblioteca `polished.js`. Aceasta furnizeaza o pletora de functii foarte utile. Aduce, practic, niste superputeri pentru `styled-components`. Aceasta functioneaza cu atat mai bine in combinatie cu `ThemeProvider`-ul, datorita functiilor de alterare a culorilor. Se permite cresterea luminozitatii, saturatiei, schimbarea opacitatii si multe altele, facand jongleriile foarte accesibile. Exemplu:

```
const Highlight = styled.span`
  border: 1px solid ${props=>props.theme.color};
  background-color: ${props=> lighten(0.4, props.theme.color)};
`;
```

In continuare, vom discuta putin despre securitatea aplicatiei si metodele de administrare a conturilor.

## JWT [9]

JSON Web Tokens este un standard web de transmitere a datelor in format JSON. Token-ul este compus din 3 parti: header, payload si semnatura.

Header-ul contine, de regula, tipul token-ului (JWT) si algoritmul de encryptare. Payload-ul contine datele propriu-zise, in jormat JSON. De regula, acestea sunt datele userului, permisiunile acestuia si un timestamp cu momentul emiterii tokenului. Aceasta permite setarea unui termen de valabilitate pentru token. Signatura este compusa din header-ul si payload-ul encodeate individual, concatenate si encryptate. Encryptarea se face ori printr-un secret al partilor cu algoritmul HMAC, ori printr-o pereche de key (publica + privata), folosind RSA. Avantajele JWT sunt:

1- compactibilitatea: Token-urile au o dimensiune redusa, ceea ce le permite sa fie trimise foarte usor in header-ele requesturilor de orice fel. In plus, transmiterea e rapida.

2- exhaustivitatea: JWT-ul contine, in payload, toate datele necesare. In cazul autentificarii, asta inseamna toate datele despre utilizatorul logat. Se elimina necesitatea de a face mai multe interogari.

3- siguranta: Desi token-ul este trimis "la vedere", modul in care este construita semnatura permite verificarea temeinica a continutului livrat. Orice incercare de alterare a datelor, in lipsa cheilor/secretului, este usor depistata.

# Auth0

Este un furnizor de servicii de autentificare third-party. Crearea unui serviciu de login intr-o aplicatie React-Node nu este tocmai dificila, folosind un standard precum JWT. Totusi, aceasta este, in primul rand, susceptibila la eroare. In plus, logarea standard nu este singura cerinta pentru o aplicatie moderna. Email-ul de verificare al contului, conectivitatea cu servicii sociale (Facebook, Github etc.) si setarea de permisiuni sunt doar cateva exemple de cerinte aditionale.

Toate acestea pot reprezenta saptamani intregi de munca la un proiect. Mai mult, ele nu reprezinta decat fundatia pe care se dezvoltata aplicatia noastra, ceea ce poate face experienta foarte frustranta. Aici intervine Auth0. Toate aceste servicii sunt frumos impachetate fiind disponibile pe orice platforma si atent securizate de o echipa de profesionisti. Customisabilitatea serviciilor Auth0 este foarte larga. Acestia permit oricati clienti definiti ori API-uri.

Se pot crea reguli de adaugare a metadatelor [10]:

Add persistent attributes to the user

```
1  function (user, context, callback) {
2    user.user_metadata = user.user_metadata || {};
3    user.user_metadata.color = user.user_metadata.color || 'green';
4    user.user_metadata.crew = user.user_metadata.crew || '';
5    user.user_metadata.role = user.user_metadata.role || 'One man army';
6    user.user_metadata.tools = user.user_metadata.tools || 'My Bear hands';
7    user.user_metadata.honor = user.user_metadata.honor || 100;
8    user.user_metadata.reputation = user.user_metadata.reputation || 100;
9    user.user_metadata.description = user.user_metadata.description || 'Lorem Ipsum este pur
10
11
12  auth0.users.updateUserMetadata(user.user_id, user.user_metadata)
13    .then(function(){
14      callback(null, user, context);
15    })
16    .catch(function(err){
17      callback(err);
18    });
19 }
```

Cat si app\_metadata, foarte utile pentru definirea de roluri ale utilizatorilor, ori alte informatii. Pentru folosirea serviciului, se defineste o clasa WebAuth ca aici (specifica React).

```
auth0 = new auth0.WebAuth({
  domain: AUTH_CONFIG.domain,
  clientID: AUTH_CONFIG.clientId,
  redirectUri: AUTH_CONFIG.callbackUrl,
  audience: `https://${AUTH_CONFIG.domain}/userinfo`,
  responseType: 'token id_token',
  scope: 'openid profile'
});
```

De aici, login-ul se face foarte usor prin apelarea this.auth0.authorize() si ceea ce ramane de implementat sunt niste metode aditionale pentru obtinerea profilului, a token-ului de acces ori

setarea sesiunii. Toate acestea sunt foarte bine exemplificate in repo-ul acesta [11] din care este preluat cod chiar pentru acest proiect.

O separare interesanta pe care o face Auth0 este intre token-ul pentru logare si token-ul pentru API-ul de Management al datelor. Practic acestea sunt servicii diferite. Acesta este un avantaj foarte mare pe care il ofera platforma. Se pot defini oricati clienti care sa acceseze si sa prelucreze un anumit sector de date.

Folosindu-se de anterior discutatul JWT, Auth0 ofera un token pentru fiecare API pe care ti-l definesti si o metoda de verificare a respectivului token, in API-ul tau independent de serviciile Auth0. Aceasta se realizeaza prin intermediul checkJwt-ului discutat mai sus. Daca token-ul trece de verificare, API-ul executa codul, altfel requestul este intors cu o eroare. Aceasta verificare se face foarte usor:

```
23   const jwt = require('express-jwt');
24   const jwksRsa = require('jwks-rsa');
25
26   const checkJwt = jwt({
27     secret: jwksRsa.expressJwtSecret({
28       cache: true,
29       rateLimit: true,
30       jwksRequestsPerMinute: 5,
31       jwksUri: `https://seastar.eu.auth0.com/.well-known/jwks.json`
32     }),
33     audience: 'https://seastar.eu.auth0.com/api/v2/',
34     issuer: `https://seastar.eu.auth0.com/`,
35     algorithms: ['RS256']
36   })
```

Personal, mi-a fost relativ dificil sa inteleg cum lucreaza Auth0, la inceput. Principalul motiv este exhaustivitatea de adaptabilitate spre care tintesc acestia. Este gandit si construit sa sustina orice fel de nevoi, personale sau enterprise. Acest fapt este grozav, insa abstractizarea profunda a conceptelor necesita intelegerea lor globala, chiar daca cerintele tale sunt mult mai limitate.

Totusi, Auth0 ramane o unealta foarte puternica in mainile oricarui Web Developer si sunt fericit ca mi-am insusit-o.

## Redux

Acesta este o librerie de administrare a starii si este dezvoltata folosind JS. In principiu, in React, toate datele sunt tinute ca stari locale. Starea este similara cu proprietatile obiectelor, insa este pastrata intr-un singur obiect, initializat in constructor. Motivul pentru aceasta consta in cateva functii implementate special pentru aceste stari (ex. this.setState())

In exemplul din dreapta putem vedea un constructor cu o stare initializata. Avantajul principal este incapsularea locala si clara. Apoi, datele se pot trimite intre componente. Cea mai comuna transmitere

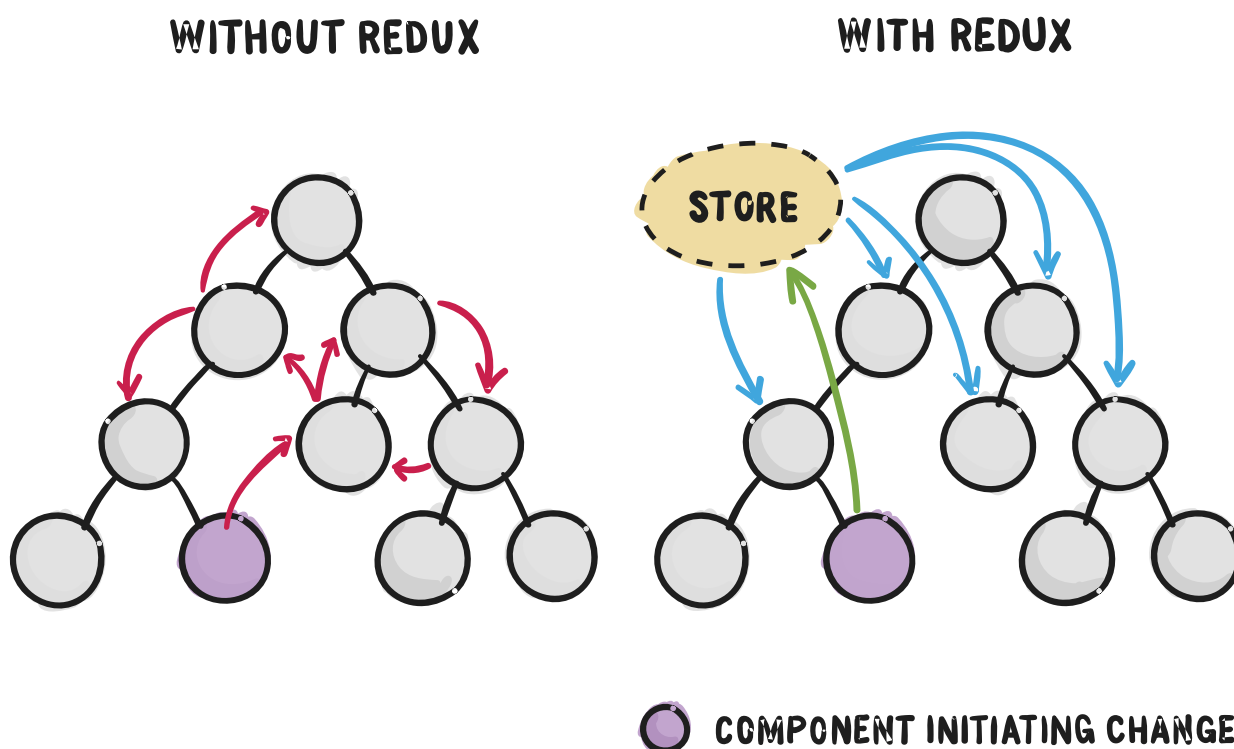
```
class CategoryForm extends Component {
  constructor(props){
    super(props);
    this.state = {
      name:'',
      image:'',
      description:'',
      private: false,
      items_count: 0,
      items: []
    }
  }
}
```

de date este catre copii, insa sunt intalnite si schimbari intre noduri de acelasi grad. Acestea nu sunt, insa, considerate o practica buna. In general, nu este modul de lucru pentru care a fost gandit React si folosirea extinsa a acestor schimburi de date poate da nastere unui cod ofuscant.

Un alt aspect al state-urilor locale il reprezinta numarul mare de apeluri de API care trebuie facute pe tot parcursul aplicatiei. In plus, odata cu scalarea aplicatiei aceasta poate deveni foarte incarcata si greu de urmarit si de intretinut.

## Introducem REDUX

Ceea ce face React atat de raspandit si folosit este sinergia superba cu Redux. In mare, aceasta biblioteca este o 'magazie' sub forma unei stari globale pentru aplicatia noastra. Astfel, se completeaza foarte bine. Datele nu trebuiesc incarcate decat o data si se pot trimite din radacina arborelui pana in frunze. Este, de asemenea, ideal pentru aplicatiile de tip SPA (Single Page Application)

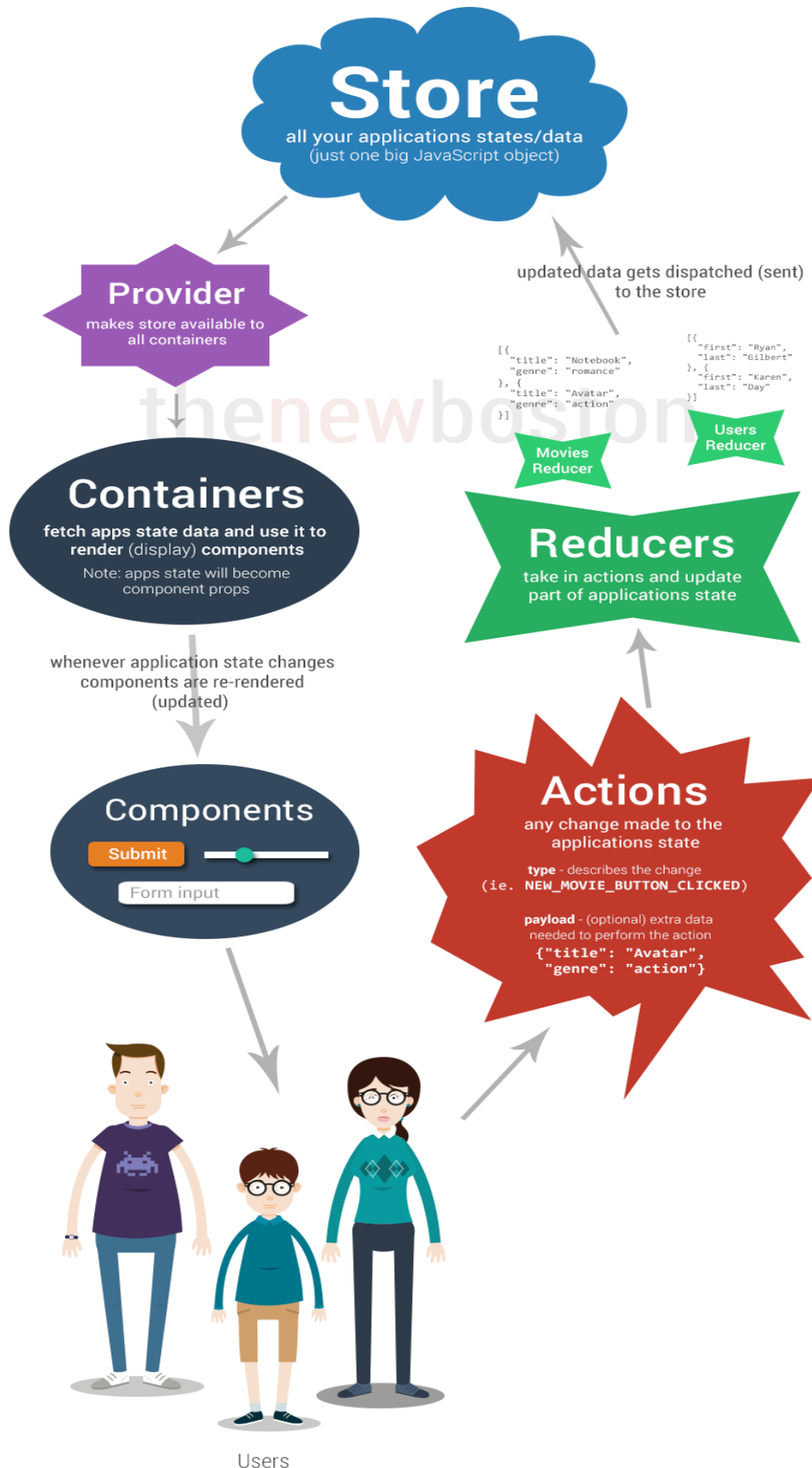


Daca tinem cont ca React face foarte bine re-randerea selectiva doar acolo unde datele sunt schimbate, se poate vedea foarte bine sinergia.

Ceea ce face Redux atat de util este claritatea. Practic, exista o singura sursa de date corecta, STORE-ul. Acesta furnizeaza si este cel asupra caruia se aplica modificarile de stare. Vom vedea imediat circuitul acesta. Desi paradigma de lucru poate fi dificil de aprofundat, aceasta clarifica foarte mult aplicatiile. Tendinta de scalare este horizontala si nu verticala. Astfel, indiferent cate functionalitati noi adaugam la aplicatia noastra, fiecare functioneaza relativ independent de celelalte. Componenta care declanseaza schimbarea de date trebuie doar sa delege CARE este cea schimbare. Redux schimba datele din 'magazie' si furnizeaza noile date pentru ca React sa actualizeze exact ceea ce s-a schimbat in fiecare componenta afectata. Ideal!

Diagrama de mai jos[13] explica destul de bine acest circuit de lucru.

# Redux Explained



Circuitul acesta este compus din: creatori de actiuni, reducatori si starea globala. (numita 'Store')

1) Starea globala este un singur obiect JS in care sunt tinute toate datele aplicatiei noastre. De aici sunt aduse in componentele React care le cer. Pentru a accesa store-ul este nevoie de un Provider. Acesta este doar o componenta wrapper cu magazia primita ca proprietate.

In plus, ne trebuie o functie care sa mapeze starile de care avem nevoie la proprietatile componentei care le foloseste. Aceasta se defineste foarte usor.

```
40 function mapStateToProps(state){
41   | return state;
42 }
43
44 function mapDispatchToProps(dispatch){
45   | return {
46   |   actions: bindActionCreators(actions, dispatch)
47   | }
48 }
49
50 export default withRouter(connect(mapStateToProps, mapDispatchToProps)(App))
```

Acelasi lucru trebuie facut si pentru dispatcher-ul care lanseaza actiunile. Mai sus se pot vedea ambele functii.

In general, acestea se pot defini in fiecare componenta care foloseste date din Store, insa abordarea mea este aceea de a le lega o singura data, in componenta Root si sa trimit datele singure in josul arborelui. Avantajele si dezavantajele aici sunt relativ mici si dependente de context.

2) Reducatorii sunt cei care primesc schimbarile de date care trebuiesc facute in Store, le executa si trimit mai departe datele actualizate. In exemplul de mai jos, utilizatorul logat este actualizat, in functie de context.

Se poate observa ca reducatorii sunt relativ lipsiti de logica. Anumite procese mai complexe se pot intalni, insa acesta este fix motivul pentru logica Redux.

```
var c = require('../constants.js')

let authedReducer = function(authed={}, action){
  switch (action.type) {
    case c.LOGIN_SUCCESS:
      | return Object.assign({}, authed, action.payload);
    case c.LOG_OFF:
      | return {}
    case c.LOGIN_UPDATE:
      | return Object.assign({}, authed, action.payload);
    default: return authed;
  }
}
```

De regula, datele vin deja procesate din actiune si sunt doar atribuite la locul potrivit, prin reducers.

3) Creatorii de actiuni sunt functii care assembleaza si expedieaza actiunile care urmeaza sa fie procesate de catre reducatori. In general, aici este continuata logica si tot aici se fac si eventuale apeluri de API care sunt, de regula, necesare. Aceste functii sunt legate de componente prin mapDispatchToProps.



```
254 ~ deletePost: function(post){
255 ~     return dispatch=>{
256 ~         fetch('/api/deletePost', {
257 ~             method: 'POST',
258 ~             headers: {
259 ~                 'Accept': 'application/json',
260 ~                 'Content-Type': 'application/json',
261 ~                 'Authorization': localStorage.getItem('manageToken'),
262 ~             },
263 ~             body: JSON.stringify(post)
264 ~         }).then(resp => resp.json())
265 ~         .then(respJson => dispatch({type: c.DELETE_POST,
266 ~             | | | | | | | | | | | | | | payload: post}))
267 ~         )
268 ~     }
269 ~ },
```

Acesta este un exemplu de action creator care sterge un comentariu. Se poate vedea cum acesta face si apelul la API cat si emiterea actiunii catre reducer.

Astfel, putem observa acum cel mai mare avantaj al folosirii Redux, din punctul meu de vedere. Ciclul de schimbare a datelor din Magazie, desi mai lung, este foarte clar si usor de administrat.

In plus, si cel mai important, este foarte usor de extins!

Daca programatorul doreste sa adauge un feature nou, acesta nu se amesteca in niciun fel cu continutul deja existent. Tot ce are de facut este sa-si defineasca actiunile si sa creeze reducerul.

```

3      let actions = {
4      //USERS
5      >      initializeUsers: function() {=
18 >      updateUser: function(user_id, body) {=
39 >      getUser: function(userId) {=
55 >      loginUser: function(userId) {=
71 >      logOff: function(){=
74      //CATEGORIES
75 >      initializeCategories: function(){=
91 >      addCategory: function(category){=
107 >      deleteCategory: function(category){=
123      //ITEEEEEMS
124 >      initializeItems: function(){=
139 >      addItem: function(item){=
155 >      deleteItem: function(item){=
171 >      voteItem: function(item, userId, score){=
186 >      deVoteItem: function(item, vote){=
201 >      cleanItem: function(item, userId, score){=
216      //POSTS
217 >      initializePosts: function(){=
232 >      addPost: function(post){=
249 >      deletePost: function(post){=
265 >      changeColor: function(color){=
268 >      // decrementItem: function(item, userId){=
283 >      // addUserToCrew: function(user, crew){=
302      }

```

# Dezvoltarea aplicatiei

Prezentarea tehnologiilor folosite fiind deja facuta, ma voi concentra pe elaborarea fiecărei functionalitati implementata in website.

## Radacina aplicatiei

Asa cum am discutat mai sus, aplicatiile React au o structura ierarhica arborescenta. Tot ceea ce randeaza React se face in cadrul unui DOM virtual, definit intr-o componenta principala (numita conventional Root, App in cazul meu) Singurele elemente care sunt deasupra acestei componente sunt anumite wrappere, precum componenta de Router ori Provider-ul. Router-ul este o componenta standard React si ne ofera access la functiile de navigare pe care le vom vedea.

```
render(  
  <Provider history={history} store={store}>  
    <BrowserRouter >  
      <App />  
    </BrowserRouter>  
  </Provider>,  
  document.getElementById('app')  
)
```

Aceasta, impreuna cu continutul direct al componentei App reprezinta fundatia pe care se construiesc aplicatia noastra React.

## App

In ceea ce priveste aceasta componenta, continutul ei nu este foarte mare, insa esential dezvoltarii ulterioare a aplicatiei.

Imediat dupa furnizorul tematic, se face prima bifurcare a aplicatiei, intre componenta de navigare si segmentul principal al paginii, componenta Main.

Din punct de vedere al logicii, aici se cere token-ul de acces la API. Tot aici am legat aplicatia la starile si dispatcher-ul Redux. Astfel, trimitem datele si actiunile in josul stream-ului fara nicio dificultate.

Practic, aceasta legare la Redux se poate face in fiecare componenta, aducandu-se doar datele si actiunile necesare, insa eu am preferat metoda unei singure legaturi si transmiterea datelor in modul clasic, in josul Flux-ului.

```
function mapStateToProps(state){  
  return state;  
}  
  
function mapDispatchToProps(dispatch){  
  return {  
    actions: bindActionCreators(actions, dispatch)  
  }  
}  
  
export default withRouter(connect(mapStateToProps, mapDispatchToProps)(App))
```

## NavBarul

Este componenta de navigare a aplicatiei noastre. Aceasta contine toate linkurile din bara de navigare. Ca logica, aici se apeleaza functiile de login/logoff.

```
render(){
  return (
    <NavWrapper>
      <ColorTicks changeColor={this.props.actions.changeColor}></ColorTicks>
      <Divider></Divider>
      <MyLink to='/getItems'><NavButton background={NumberedList}> All Items </NavButton> </MyLink>
      <Divider></Divider>
      <MyLink to='/categories'> <NavButton background={Home}> Categories </NavButton> </MyLink>
      <Divider></Divider>
      {this.props.auth.isAuthenticated()
      ? <MyLink to='#'><NavButton background={Login} onClick={this.logout.bind(this)}> Logoff </NavButton> </MyLink>
      : <MyLink to='#'><NavButton background={Login} onClick={this.login.bind(this)}> Login </NavButton> </MyLink> }
      <Divider></Divider>
      <MyLink to='/Contact'><NavButton background={Contact}> Contact </NavButton> </MyLink>
    </NavWrapper>
  )
}
```

Ca structura, avem doar un container cu obiecte. Divider-ul este un spatiu care separa butoanele. MyLink reprezinta Link-ul furnizat de React-router, stilizat prin styled-components. Componenta ColorTicks este destul de interesanta si o vom explica imediat.

## ColorTicks

Este primul element de UI care se observa cand deschidem aplicatia. Design-ul sau este consistent cu restul meniului iar scopul acestuia este de comutator intre temele aplicatiei. Alegerea tematica facuta este pastrata in magazia Redux, ramanand constanta odata cu navigarea prin site, pana la o noua schimbare. Culoarea este injectata in fiecare componenta care foloseste libraria styled-components.



In primul rand, aplicatia este 'imbracata' in ThemeProvider-ul furnizat de aceasta librerie. Ulterior, in definirea stilistica a oricarei componente, culoarea este accesata prin `props.theme.color`. Folosind polished, culoarea este alterata pe alocuri pentru a servi scopuri diverse.

acdc152

Din motive personale, aplicatia se opreste la o singura culoare, insa procedeul s-ar putea scala la oricate optiuni, singurul dezavantaj fiind suprasolicitarea utilizatorului in privinta alegerilor.

```
render(){
  return (
    <Wrapper>
      <TicksTray>
        <ColorTick color='DarkGoldenRod' onClick={()=>this.changeColor('DarkGoldenRod')}></ColorTick>
        <ColorTick color='orangered' onClick={()=>this.changeColor('orangered')}></ColorTick>
        <ColorTick color='brown' onClick={()=>this.changeColor('brown')}></ColorTick>
        <ColorTick color='palevioletred' onClick={()=>this.changeColor('palevioletred')}></ColorTick>
        <ColorTick color='green' onClick={()=>this.changeColor('green')}></ColorTick>
        <ColorTick color='blue' onClick={()=>this.changeColor('blue')}></ColorTick>
        <ColorTick color='indigo' onClick={()=>this.changeColor('indigo')}></ColorTick>
      </TicksTray>
      <Header>Color?</Header>
    </Wrapper>
  )
}
```

Din punct de vedere tehnic, implementarea nu este dificila. Fiecare Tick reprezinta o culoare si la click se expedieaza actiunea redux care activeaza urmatorul reducer. Acesta memoreaza in magazie culoarea aleasa de utilizator.

```
let uiReducer = function(ui={}, action){
  switch(action.type){
    case c.CHANGE_COLOR: return Object.assign({},ui , {color: action.payload});
  }
  return ui;
}
```

## Main component

Aceasta este componenta principala a aplicatiei.

Din punct de vedere al logicii, aici se incarca, pe rand, toate continuturile afisate. Tot aici se produc initializarile pentru utilizatori, categorii, obiectele ierarhizate cat si comentariile lasate. Practic, in metoda de lifetime cycle 'componentWillMount()' se apeleaza actiunile: initializeUsers(), initializeCategories(), initializeItems(), initializePosts(). Acestea aduc datele din baza de date in magazia Redux. Acest proces se intampla o singura data, la deschiderea aplicatiei.

Tot aici are loc si inregistrarea utilizatorului in magazia Redux. Desi functionalitatea de autentificare furnizata de Auth0 ne ofera posibilitatea de a accesa profilul utilizatorului inregistrat, este de preferat sa avem o intrare pentru acesta si in starea globala.

```
if(this.props.auth.isAuthenticated())
{const { userProfile, getProfile } = this.props.auth;
if (!userProfile) {
  getProfile((err, profile) => {
    this.props.actions.loginUser(profile.sub)
  })
}
```

Din punct de vedere structural, componenta Main este cea care contine Switch-ul in care se inregistreaza Route-le. Fiecare dintre acestea reprezinta o 'fereasta' de interactiune cu aplicatia.

Acestea sunt accesate prin Link-urile de care am vorbit mai sus.


Fiecare dintre rute contine componenta sau componentele pe care dorim sa le randam si fiecare dintre acestea are incorporate datele din magazie care le sunt transmise.

```
render(){
  return (
    <main>
      <Switch>
        <Route exact path="/">=
        <Route exact path="/userProfile" >=
        <Route exact path="/contact">=
        <Route exact path="/categories">=
        <Route exact path="/addCategory">=
        <Route exact path="/addItem">=
        <Route exact path="/getItems">=
        <Route path="/callback" render={({props})=>{=
      </Switch>
    </main>
  )
}
```

Din punct de vedere vizual, componenta aceasta este nula, ea nefiind randata in mod direct pe ecran si reprezentand doar un intermediar.

# ItemsRank

Este componenta care afiseaza itemele dintr-o categorie/toate categoriile. Modul in care este construita permite refolosirea ei pentru oricare categorie/nicio categorie.

#2		Viking	Michael Hirst	The world of the Vikings is brought to life through the journey of Ragnar Lothbrok, the first Viking to emerge from Norse legend and onto the pages of history - a man on the edge of myth.	D E C
----	---	--------	---------------	---	-------------

Functionalitatea consta in listarea obiectelor rankate. Acest ranking include date precum:

1- Pozitia in clasament: Indicele numeric al pozitiei. Sortarea aceasta se face fix inainte de afisarea datelor si se reactualizeaza la fiecare actualizare a voturilor.

```
var myitems = !this.props.category ? this.props.items :  
| this.props.items.filter(a=> a.category == this.props.category._id);  
myitems.sort((a,b)=>{return b.score - a.score;});
```

2- Scorul si butoanele de (de)votare: Acestea arata numarul de voturi acumulat de obiectul respectiv. Butoanele de vot +/- sunt dispuse deasupra si dedesubtul acestui numar. In cazul in care s-a votat deja acestea sunt inlocuite de un singur buton care retrage votul. Daca utilizatorul nu este logat, atunci butoanele devin inutilizabile si capata o culoare mai inchisa.

3- Imaginea obiectului: Este pur si simplu imaginea a carei link a fost inclusa de utilizator la adaugarea obiectului. La highlight pe obiectul respectiv, aceasta creste putin pentru a accentua sublinierea vizuala. Linkul acesteia poate fi schimbat folosind functia de editare.

4- Titlul: Ne spune cum se numeste obiectul. Acesta are un font putin mai mare decat restul textului, pentru vizibilitate. El poate fi schimbat folosind functia de editare.

5- Numele creatorilor: Numeste persoana/ persoanele carora le multumim pentru contributia lor artistica si culturala. Numele lor sunt adaugate de utilizator, odata cu obiectul si reprezinta, in general, nume de regizori, cantareti sau scriitori. Acestea pot fi schimbate folosind actiunea de editare.

6- Descrierea obiectului: Este un text descriptiv ce ajuta la identificarea item-ului. Acesta este adaugat de utilizator si poate fi schimbat folosind actiunea de editare.

7- Butoanele de actiune: sterge, editeaza, comenteaza: Primele doua sunt inaccesibile utilizatorilor, cu exceptia creatorului categoriei. Stergerea elimina obiectul definitiv, atat din magazie cat si din baza de date. Editarea redeschide modalul de adaugare si permite schimbarea datelor obiectului.

AC acdc152	<p>There are some Hollywood liberties taken during the show this is to be expected it is a show. If you made a show about everyday life during the vikings time most would be dull and not exciting it is not like they went to battle everyday. The show has a lot of historical basis with some liberties taken to liven the show up. It is well produced and has great cinematography . If , you are however looking for a straight fact based view on vikings go to the library and pull anthropology research journals out. For everyone else if you like battles with historical styles, understandings of how they navigated and views on the social structure of the vikings then you will get a understanding. You will also see how laws and ethics were for the vikings so yes there is some historical accuracy to the show . All in all if you want entertainment mixed with history it is a great show.</p>	D
	 Alexandru Florescu	Supposedly there's a new season coming up O.o D
AC acdc152	<div></div> <div>POST</div>	

Butonul de comentare deschide fereastra care afiseaza comentariile utilizatorilor. Orice utilizator logat poate lasa oricate comentarii doreste.

Din punct de vedere vizual, acestea pastreaza tematica bordurii superioare, care, in cazul asta, se continua in stanga cu profilul si numele utilizatorului care scrie comentariu. In dreapta sunt plasate butoanele de actiune. Acestea sunt accesibile doar celui care a postat comentariul.

Dedesubtul comentariilor se gaseste fereastra care face posibila postarea acestora. Aceasta este vizibila doar utilizatorilor logati. Caracterizata prin simplitate, contine doar un camp pentru text si un buton de postare a comentariului.

Din punct de vedere structural, pagina contine Tabelul cu obiecte si Modalul de editare.

```
return <StrippedContainer header={header}>
  <Table>=
  {this.state.show && <Modal=
</StrippedContainer>
```

```
return ([
  <Tr>
    <Td className="score">#{this.i}</Td>
    <Td>
      {item.voted_by.indexOf(this.props.authed.user_id) > -1
      ? [<FW>{item.score}</FW>, <Sign onClick={this.devote.bind(this, item)}>b</Sign>]
      : [<Sign className={!this.props.authed.user_id && "locked"} onClick={this.props.authed.user_id && this.increment.bind(this, item)}>+</Sign>,
        <FW>{item.score}</FW>,
        <Sign className={!this.props.authed.user_id && "locked"} onClick={this.props.authed.user_id && this.decrement.bind(this, item)}>-</Sign>,
        ]}
    </Td>
    <Td><SImg className='item' src={item.image}/></Td>
    <Td className="cap title">{item.name}</Td>
    <Td className="cap">{item.author}</Td>
    <Td><Desc>{item.description}</Desc></Td>
    <Td><Sign className={this.props.category ? this.props.authed.user_id !== this.props.category.owner && "locked" : 'locked'}
      onClick={this.props.category ? this.props.authed.user_id===this.props.category.owner && this.deleteItem.bind(this, item) : ''}>D</Sign>
      <Sign className={this.props.category ? this.props.authed.user_id !== this.props.category.owner && "locked" : 'locked'}
      onClick={this.props.category ? this.props.authed.user_id===this.props.category.owner && this.editItem.bind(this, item) : ''}>E</Sign>
      <Sign onClick={this.showComments.bind(this, this.i)}>C</Sign></Td>
    </Tr>,
    <Tr className='comments' id={this.state.toggles.indexOf(this.i) > -1 && "expanded" }>
      <Td colSpan='8'>
        <CommentsSection
          id={this.state.toggles.indexOf(this.i) > -1 && "expanded" }
          posts={this.props.posts.filter(post=> post.item === item_id)}
          item={item} postComment={this.postComment.bind(this)}
          users={this.props.users} authed={this.props.authed}
          deletePost={this.props.deletePost}>
        </CommentsSection>
      </Td>
    </Tr>
  ])]
  )}
```

Tabelul, desi poate parea complex, este destul de simplu. Acesta contine o mapare a tuturor obiectelor si la fiecare se returneaza codul afisat mai sus. Continutul acestuia este cel care defineste randul descris mai sus.

Se poate vedea, folosita intens sintaxa conditionala liniara '? : ' si forma ei simplificata '&&' Acestea sunt foarte utile pentru afisarea de continut diferentiat in pagina.

Trebuie notat ca fiecare element structural este creat folosind styled-components.

Din acest punct de vedere conventia discutata mai sus este putin incalcata. Aceste componente nu sunt reutilizabile, ele sunt perisabile si utile doar in acest context.

```
const Table = styled.table`
const Tr = styled.tr`
const Td = styled.td`
const Desc = styled.div`
const SImg = styled.img`
const FW = styled.div`
```

Totusi, definirea lor cu styled-components este inca o masura utila, datorita incapsularii continutului stilistic si eliminarii posibilitatii de eroare umana. Daca dorim sa fim pedanti, am putea totusi sa mutam aceste definitii intr-un fisier secundar pe care sa-l importam in fisierul sursa.

Privind in retrospectiva, realizez cat de usor se poate dezvolta un framework de UI folosind aceste tehnologii



```

constructor(props){=
//Metode de votare
devote(item){=
increment(item) {=
decrement(item) {=

//Actiuni pe item
deleteItem(item){=
editItem(item){=

//Comentarii
hide(){=
showComments(item){=
postComment(item, text){=

render(){=

```

Din punct de vedere al logicii, in aceasta componenta se intampla destul de multe lucruri.

Metodele increment/decrement sunt cele care se ocupa de votarea unui item specific. Acestea verifica daca utilizatorul a votat sau nu, iar apoi apeleaza creeatorul de actiune denumit `voteItem()`

Metoda `devote` este cea care se ocupa de retragerea votului utilizatorului. Aceasta verifica daca utilizatorul a votat, iar apoi apeleaza creeatorul de actiune denumit `VoteItem()`

Urmatoarele doua sunt actiuni de alterare a obiectelor.

Butonul de stergere apeleaza actiunile `deleteItem` (pentru itemul respectiv) si, in plus, parcurge toate comentariile, stergandu-le pe cele aferente obiectului.

Butonul de editare redeschide modalul discutat mai sus. Acesta vine cu campurile deja completate si o referinta la obiectul cu pricina. La apasare pe butonul Submit, se executa actiunea de `editItem()`.

Metoda `hide()` este cea care permite ascunderea modalului.

Apoi avem actiunile pentru comentarii. `ShowComments()` deschide sectiunea de comentarii aferente obiectului pentru care a fost activata. Aceasta se face prin adaugarea unei variabile 'toggle' intr-un array. Fiecare rand de tabel ce contine o sectiune de comentarii se expandeaza dupa urmatoarea regula

```
id={this.state.toggles.indexOf(this.i) > -1 && "expanded"}
```

Metoda `postComment` face posibila publicarea comentariilor, apeland `addPost()`;

## Sectiunea de comentarii

Aici se pot vizualiza comentariile scrise si, eventual, adauga altele. Vizual, am prezentat mai sus continutul componentei.

```

return <CommentsWrapper id={this.props.id}>
  {this.props.posts.map(post=>{
    let user = this.props.users.find(user=>user.user_id == post.writer);
    return <Line>
      |   |   |   <Profile>=
      |   |   |   <CommentAndActions>=
      |   |   |   </Line> }}}
    {this.props.authed.picture &&
      <Line>|
      |   |   |   <Profile>=
      |   |   |   <Form>=
      |   |   |   </Line>
      |   |   |   }
    </CommentsWrapper>;

```

Din punct de vedere structural, pagina contine 'linii' cu toate comentarii ce trebuie afisate. Acestea sunt alcatuite din profilul utilizatorului si comentariul propriu-zis. De asemenea, tot aici este continut si formularul de adaugare a unui nou comentariu. Acesta este insa vizibil doar daca utilizatorul este logat.

Din punct de vedere stilistic, toate componentele sunt definite cu styled-components si pastreaza liniile de design specifice proiectului.

Din punct de vedere logic, componenta este destul de simpla, continand o metoda de tratare a schimbarii de continut si una pentru stergerea comentariilor. Se poate observa ca, desi metoda de stergere a comentariilor este disponibila doar utilizatorului care a scris comentariul, in metoda delete() nu se intampla nici un fel de validare. Motivul pentru aceasta este conditionarea inline de care am vorbit mai sus.

```
handleChange(e){
  |   this.setState({'comment':e.target.value})
  | }

delete(post){
  |   this.props.deletePost(post);
  | }
}
```

```
onClick={user.user_id==this.props.authed.user_id && this.delete.bind(this, post)}
```

## Categories

Componenta aceasta este alcatuita dintr-o lista de carduri pentru categorii, asezate intr-un flexbox ce permite prezentarea lor armonioasa. In plus, utilizatorilor logati li se afiseaza si un card special ce permite adaugarea de noi categorii.

```
return (
  <DashboardWrapper>
    <AddCategory
      authed= {this.props.authed} color={this.props.color} addCategory = {this.props.addCategory}>
      {this.props.categories.map( (category)=>
        <CategoryCard key={this.props.categories.indexOf(category)}
          category={category}
          items = {this.props.items}
          color={this.props.color}
          addItem={this.props.addItem}
          delete={this.props.delete}
          deleteItem = {this.props.deleteItem}
          router = {this.props.router}
          users={this.props.users}
          authed={this.props.authed}
        /> )}
    </DashboardWrapper>
  )
```

Cardul pentru adaugarea de noi categorii este simplu.

La apasarea simbolului de Plus se deschide modalul pentru adaugarea unei noi categorii. Aceasta contine campurile pentru nume, imagine si descrierea categoriei. De asemenea, mai contine un checkbox pentru categoriile private.

In cazul in care utilizatorul este anonim, cardul este pur si simplu invizibil, lasand a fi afisate doar categoriile deja existente

Add New Category





×

Add New Category

Private?

☐

Name:

Image:

Description:

Submit


Cardurile de categorii sunt compuse din 3 elemente vizuale, dimensionate dupa numarul de aur (1.618) pentru un aspect armonios.

Cele trei elemente vizuale sunt: bara de titlu, detaliile categoriei (imagine, creator si numar de obiecte aferente) si actiunile categoriei (vizualizare, adaugare obiecte si stergere categorie)

In imaginea din dreapta se poate vedea un astfel de card. A se nota ca din cele trei butoane, doar doua sunt active. Functia de stergere este disponibila doar creatorului categoriei, in cazul acesta ea fiind astfel ‘hashurata’.

Butonul de Add Item deschide un modal asemanator celui de sus, inasa continand un nou camp pentru autor. Acesta este disponibil oricarui utilizator logat. Din punct de vedere al implementarii, am refolosit modalul discutat anterior. La apasare pe butonul submit, se apeleaza creatorul de actiune addItem().

Best Music Albums



Creator:

Alexandru Florescu

#Items:

1

View rankings

Add Item

Delete

Butonul de View Rankings este cel care ne duce la pagina de vizualizare a obiectelor prezentat mai sus, aferenta categoriei respective. Acesta este disponibil si utilizatorilor anonimi. Practic, este singura actiune nerestrictionata, inasa se permite doar vizionarea, restrictiile fiind aplicate local.

Din punct de vedere structural, pagina de categorii este compusa dintr-un Dashboard care foloseste optiunea ‘display: flex’ nou introdusa in CSS3. Apoi, categoriile sunt iterate si se creeaza un card pentru fiecare. La hover peste carduri, acestea cresc putin in dimensiune pentru a le scoate in evidenta. De asemenea, cardul cu pricina trece peste celelalte.

Mai jos se poate vedea acea structura

```

return (
  <DashboardWrapper>
    <AddCategory
      authenticated={this.props.authenticated} color={this.props.color} addCategory={this.props.addCategory}>
      {this.props.categories.map( (category)=>
        <CategoryCard key={this.props.categories.indexOf(category)}
          category={category}
          items={this.props.items}
          color={this.props.color}
          addItem={this.props.addItem}
          delete={this.props.delete}
          deleteItem={this.props.deleteItem}
          router={this.props.router}
          users={this.props.users}
          authenticated={this.props.authenticated}
        /> )}
    </AddCategory>
  </DashboardWrapper>
)

```

In imaginea de mai sus se poate vedea structura Dashboard-ului, asa cum am discutat-o. Datele sunt trimise in jos, pana in cardul de categorie.

Din punct de vedere al codului, structura cardului este simpla.

```

<StrippedCard header={this.props.category.name}>
  <UpperWrap>=
  <Divider></Divider>
  <CardWrapper>=
</StrippedCard>

```

Din punct de vedere al logicii cardului, metodele nu sunt foarte complexe. Astfel, stergerea implica doar apelarea delete() si stergerea tuturor obiectelor care sunt 'continute' in acele categorii.

```

delete() {
  this.props.delete(this.props.category);
  this.props.items.forEach(item => {if(item.category==this.props.category._id) this.props.deleteItem(item)} );
}

```

Mai interesanta este metoda de vizualizare a ierarhiilor de obiecte. Majoritatea schimbarilor de pagina sunt facute folosind componenta Link, insa aceasta este disponibila doar intr-un context de JSX.

Dat fiind ca aceasta schimbare trebuie facuta aici din partea de functie JS, dar si necesitatea transmiterii unei informatii dependente de context (in cazul asta categoria pe care o vom vizualiza) este necesara o metoda alternativa.

```

view() {
  history.push({
    pathname: '/getItems',
    state: { category: this.props.category._id }
  })
}

```

Aceasta sintaxa este echivalentul unui redirect la rute '/getItems' iar id-ul categoriei este transmis in proprietatile cu care se randeaza componenta ItemsRank.

Astfel, ea poate fi folosita prin this.props.category ca si cum ar fi fost transmisa in mod clasic. Avantajul acestei metode a fost deja expus, dezavantajul ar fi ca ea face un refresh la pagina, ceea ce duce la pierderea anumitor stari din Redux.

In cazul de fata, singura pierdere este campul de culoare care se reseteaza, restul datelor fiind deja salvate cu succes.

In continuare se prezinta doua concepte care nu sunt tocmai esentiale, dar au fost folosite.

## CSS3 FlexBox [14]

Cutia flexibila este un nou mod de asezare disponibil in CSS3. Functia ei este de a se asigura ca elementele actioneaza predictibil atunci cand pagina trebuie sa accomodeze diferite rezolutii sau dimensiuni.

Pentru multe aplicabilitati, aceasta reprezinta o imbunatatire majora fata de mai vechile modele de 'block' sau 'float'.

Structural, aceasta contine un container si mai multe elemente. Containerul este declarat setand 'display:flex' sau 'display: inline-flex'.

Avantajul principal este flexibilitatea in folosire. Folosind campurile 'flex-direction:' (cu valorile posibile 'row', 'row-reverse', 'column', 'column-reverse'), 'align-items' si 'justify-content' (cu valorile 'flex-start', 'flex-end', 'center', 'space-between', 'space-around') se pot obtine o gama larga de rezultate care functioneaza consistent la toate rezolutiile.

Un alt avantaj major al FlexBox este ca face aliniera verticala foarte accesibila, in timp ce pana acum aceasta fusese foarte dificila.

## Ratia de aur in Web Design

Numarul de aur este 1.618 si este des folosit in design. Acesta apare in primul rand in natura. A fost introdus in arhitectura si arte inca din antichitate. Probabil ca Phidias nu a fost constient de aceasta cand a ridicat Parthenon-ul, insa putem argumenta existenta acestei proportii in monumentul antic.

De atunci, acesta a fost introdus in diferite discipline cum ar fi pictura, geometria, muzica si alte tipuri de design.

Toata magia sta in raportul de 1.618. Acesta este placut estetic cam oriunde apare. Am folosit si eu acest raport cu rezultate bune. Desi nu este mereu disponibil in web design, acolo unde poate fi folosit armonizeaza cu succes continutul paginii.

# Bibliografie

- 1-<https://nodejs.org/en/about/>
- 2-<https://expressjs.com/>
- 3-<https://designshack.net/articles/css/what-the-heck-is-css-specificity/>
- 4-<https://facebook.github.io/react/blog/2013/06/05/why-react.html>
- 5-[https://www.w3schools.com/cssref/tryit.asp?filename=trycss3\\_keyframes2](https://www.w3schools.com/cssref/tryit.asp?filename=trycss3_keyframes2)
- 6-<http://maketea.co.uk/2014/03/05/building-robust-web-apps-with-react-part-1.html>
- 7-<https://www.youtube.com/watch?v=WGjv-p9jYf0>
- 8-<https://medium.com/@baphemot/using-styled-components-alongside-css-modules-4d83b378bc17>
- 9-<https://jwt.io/introduction/>
- 10-<https://manage.auth0.com/#/rules/>
- 11-<https://github.com/auth0-samples/auth0-react-samples>
- 12-<https://css-tricks.com/learning-react-redux/>
- 13-<https://github.com/buckyroberts/React-Redux-Boilerplate>
- 14-[https://www.w3schools.com/css/css3\\_flexbox.asp](https://www.w3schools.com/css/css3_flexbox.asp)