Paradigme de proiectare a aplicațiilor WEB

**Platformă de gestionare a transferurilor fotbaliștilor**

*Îndrumător*: *Masterand:*  
Șef lucrări dr. ing. Gîză-Belciug Felicia Vasile Maruseac

SUCEAVA 2021

Cuprins

[1. Tema proiectului 3](#_Toc62763179)

[2. Structura bazei de date 3](#_Toc62763180)

[3. Generarea modelelor 4](#_Toc62763181)

[a. Database First 4](#_Toc62763182)

[b. Code First 5](#_Toc62763183)

[4. Rularea proiectului 6](#_Toc62763184)

[5. WebAPI 6](#_Toc62763185)

[6. MVC 12](#_Toc62763186)

[7. Nivelurile aplicației WEB 14](#_Toc62763187)

[8. Specificații de implementare 17](#_Toc62763188)

[a. Ștergerea datelor 17](#_Toc62763189)

[b. Introducerea datelor 18](#_Toc62763190)

[c. Cache 19](#_Toc62763191)

[d. Validarea datelor 20](#_Toc62763192)

[e. Filtrarea datelor în interfață 20](#_Toc62763193)

[9. Concluzii și posibilități de dezvoltare 21](#_Toc62763194)

# Tema proiectului

Tema acestui proiect este realizarea unei aplicații (site web) pentru gestiunea transferurilor jucătorilor și menținerea unui istoric al acestor transferuri.

Proiectul a fost realizat utilizând:

* O bază de date MySQL
* Limbajul NodeJS pentru partea de Back-End
* ORM-ul Sequelize pentru generarea modelelor entităților utilizate
* Paradigmele MVC și WebAPI
* Pachetul *node-cache* pentru a obține date din cache
* Limbajele HTML, CSS, Javascript pentru partea de Front-End
* Programul Postman pentru testarea rutelor până la realizarea FE
* Github pentru menținerea codului și posibilitatea de a lucra de pe diverse dispozitive.

# Structura bazei de date

Entitățile utilizate sunt cele de bază pentru orice proiect despre fotbal: *ligi*, *echipe*, *jucatori* și *transferuri*. Relațiile dintre acestea sunt reprezentate în Figura 1.

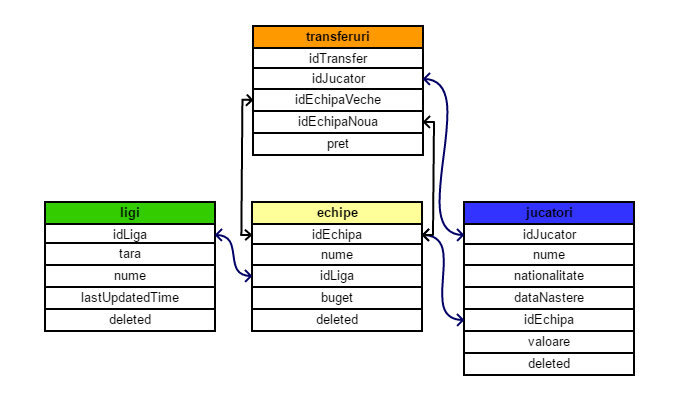


Figura 1 - Structura bazei de date

# Generarea modelelor

Modelele au fost generate cu ajutorul pachetului Sequelize, în două moduri: Database First și Code First.

## Database First

Acest mod presupune crearea întâi în baza de date a tabelului, cu coloanele și tipul de date potrivite, după care se rulează comanda:

**sequelize-auto -h 127.0.0.1 -d proiectppaw -u vasi -x @DminTest123! -p 3306 --dialect mysql -o ./models -t ligi,echipe,jucatori,transferuri**

Această comandă preia din baza de date definiția tabelelor și generează în folderul **models** o structură similară cu aceasta:

/\* jshint indent: 2 \*/

const Sequelize = require('sequelize');

module.exports = function(sequelize, DataTypes) {

  return sequelize.define('ligi', {

    idLiga: {

      autoIncrement: true,

      type: DataTypes.INTEGER,

      allowNull: false,

      primaryKey: true

    },

    tara: {

      type: DataTypes.STRING(45),

      allowNull: false

    },

    nume: {

      type: DataTypes.STRING(45),

      allowNull: false,

      unique: "nume\_UNIQUE"

    },

    lastUpdatedTime: {

      type: DataTypes.STRING(45),

      allowNull: false

    },

    deleted: {

      type: DataTypes.TINYINT,

      allowNull: false,

      defaultValue: 0

    }

  }, {

    sequelize,

    tableName: 'ligi',

    timestamps: false,

    indexes: [

      {

        name: "PRIMARY",

        unique: true,

        using: "BTREE",

        fields: [

          { name: "idLiga" },

        ]

      },

      {

        name: "nume\_UNIQUE",

        unique: true,

        using: "BTREE",

        fields: [

          { name: "nume" },

        ]

      },

    ]

  });

};

Este important de reținut că atunci când se rulează comanda de generare este necesar să se includă toate tabelele existente, deoarece altfel se vor șterge modelele pentru tabelele care nu au fost incluse în comandă.

## Code First

În cazul acestei tehnici, mai întâi se scrie în cod definiția modelului, similar cu ce s-a generat la punctul **a,** cu diferența că mai trebuie adăugată următoarea linie de cod:

await Jucatori.sync({alter: true});

*Sync* are rolul de a verifica dacă există tabelul în baza de date și de a-l crea dacă este necesar. De asemenea, parametrul *alter* setat pe **true** verifică la fiecare repornire a aplicației dacă s-au modificat câmpuri ale bazei de date (s-a adăugat o coloană nouă sau s-a modificat tipul unei coloane) și actualizează baza de date nemaifiind necesare fișiere de migrare.

# Rularea proiectului

După descărcarea proiectului, mai întâi este necesară rularea comenzii: **npm install.** Aceasta are rolul de a instala unele pachete necesare (specificate în fișierul *package.json*), cum ar fi *Sequelize* sau *node-cache.*

Aplicația poate fi pornită prin rularea comenzii: **node server.js** în folderul principal al proiectului. Apoi se navighează către [*http://localhost:3000/*](http://localhost:3000/)și se pot realiza operațiile dorite în aplicație.



# WebAPI

Paradigmele utilizate în dezvoltarea proiectului sunt WebAPI și MVC.

Implementarea prin WebAPI presupune utilizarea a câte unui fișier pe BackEnd pentru fiecare entitate, în care sunt definite rutele de apelat:

router.post('/createEchipa', async (req, res) => {

  const result = await bllEchipe.addEchipa(req.body);

  if (result === 'success') {

    res.send('Created');

  } else {

    res.status(400).send(result);

  }

});

router.get('/getEchipe', async (req, res) => {

  const result = await bllEchipe.getAllTeams();

  if (result === 'error') {

    res.status(400).send('error');

  } else {

    res.send(result);

  }

});

router.get('/getEchipa/:id', async (req, res) => {

  const result = await bllEchipe.getEchipaById(req.params.id);

  if (result === 'error') {

    res.status(400).send('error');

  } else {

    res.send(result);

  }

});

router.post('/updateEchipa/:id', async (req, res) => {

  try {

    const result = await bllEchipe.updateEchipa(req.params.id, req.body);

    if (result === 'success') {

      res.send('Updated');

    } else {

      res.status(400).send(result);

    }

  } catch (err) {

    res.send(err.message);

  }

});

router.delete('/deleteEchipa/:id', async (req, res) => {

  const result = await bllEchipe.deleteEchipa(req.params.id);

  if (result === 'success') {

    res.send('Deleted');

  } else {

    res.status(400).send(result);

  }

});

WebAPI , spre deosebire de MVC, trimite către client (browser sau o aplicație de Front-End) datele într-un format cunoscut (XML, JSON, etc), iar clientul va fi responsabil de cum le va afișa în pagină.

Frontend-ul este realizat utilizând limbajele HTML, CSS și Javascript. Structura paginii este definită cu ajutorul limbajului HTML astfel:

<!DOCTYPE html>

<html>

  <head>

    <meta charset="utf-8" />

    <meta name="viewport" content="width=device-width, initial-scale=1.0" />

    <title>Adauga jucator</title>

    <link rel="stylesheet" href="../css/first.css" />

    <script src="../js/first.js"></script>

  </head>

  <body onload="populatePlayersDropdown()">

    <button id="home"><a href="../index.html">Home</a></button>

    <h2>Adauga jucator</h2>

    <span class="input-group-text" id="basic-addon1"

      >Nume: &emsp;&emsp;&nbsp;&nbsp;</span

    >

    <input type="text" id="nume" class="form-control" />

    <br /><span class="input-group-text" id="basic-addon1">Nationalitate:</span>

    <input type="text" id="nationalitate" class="form-control" />

    <br /><span class="input-group-text" id="basic-addon1"

      >Data nastere:&nbsp;</span

    >

    <input type="date" id="dataNastere" class="form-control" />

    <br /><span class="input-group-text" id="basic-addon1"

      >Echipa: &emsp;&emsp;&nbsp;</span

    >

    <select name="idEchipa" id="echipe">

      <option id="0"></option>

    </select>

    <br /><span class="input-group-text" id="basic-addon1"

      >Valoare: &emsp;&emsp;</span

    >

    <input type="text" id="valoare" class="form-control" />

    <br /><br />

    <button onclick="adaugaJucator()">Adauga</button>

    <div id="message"></div>

  </body>

</html>

Cu ajutorul limbajului Javascript se obțin valorile completate în pagina Web, se validează datele și se face apel către server pentru realizarea operațiilor necesare.

const adaugaJucator = async () => {

  const nume = dom('nume').value;

  const nationalitate = dom('nationalitate').value;

  const dataNastere = new Date(dom('dataNastere').value).getTime();

  const e = dom('echipe');

  const idEchipa = parseInt(e.options[e.selectedIndex].id);

  const valoare = dom('valoare').value;

  if (!idEchipa || isNaN(valoare) || parseInt(valoare) <= 0) {

    alert('Something wrong');

  } else {

    body = {

      nume,

      nationalitate,

      dataNastere,

      idEchipa,

      valoare,

    };

    const result = await addEntityServer(body, '/createJucator');

    if (result.status === 200) {

      window.location.href = '../html/jucatoriList.html';

    }

  }

};

Apelurile către server sunt realizate prin funcții generice pentru fiecare operație CRUD, astfel:

const addEntityServer = (body, route) => {

  return new Promise(function (resolve, reject) {

    let xhr = new XMLHttpRequest();

    xhr.open('POST', route);

    xhr.setRequestHeader('Content-Type', 'application/json;charset=UTF-8');

    xhr.onload = function () {

      resolve({

        status: this.status,

        statusText: xhr.statusText,

      });

    };

    xhr.onerror = function () {

      reject({

        status: this.status,

        statusText: xhr.statusText,

      });

    };

    xhr.send(JSON.stringify(body)); // transmit the object converted to JSON

    xhr.onreadystatechange = function () {

      if (xhr.readyState === 4) { // if HTTP request is DONE

        if (xhr.status === 200) { // if status is ok

          dom('message').innerHTML = 'Success. Entity saved.';

          resolve({

            status: this.status,

            statusText: xhr.statusText,

          });

        } else {

          dom('message').innerHTML =

            'Error. Entity not saved. ' + xhr.responseText;

        }

      }

    };

  });

};

const getAllEntitiesServer = (route) => {

  return new Promise(function (resolve, reject) {

    var xhr = new XMLHttpRequest();

    xhr.open('GET', route);

    xhr.onload = function () {

      if (xhr.status == 200) {

        resolve({

          status: this.status,

          statusText: xhr.statusText,

          items: JSON.parse(xhr.response),

        });

      } else {

        resolve({

          status: this.status,

          statusText: xhr.statusText,

        });

      }

    };

    xhr.onerror = function () {

      reject({

        status: this.status,

        statusText: xhr.statusText,

      });

    };

    xhr.send();

  });

};

const getOneEntityServer = (id, route) => {

  return new Promise(function (resolve, reject) {

    var xhr = new XMLHttpRequest();

    xhr.open('GET', route + id);

    xhr.onload = function () {

      if (xhr.status == 200) {

        resolve({

          status: this.status,

          statusText: xhr.statusText,

          item: JSON.parse(xhr.response),

        });

      } else {

        resolve({

          status: this.status,

          statusText: xhr.statusText,

        });

      }

    };

    xhr.onerror = function () {

      reject({

        status: this.status,

        statusText: xhr.statusText,

      });

    };

    xhr.send();

  });

};

const updateEntityServer = (id, body, route) => {

  return new Promise(function (resolve, reject) {

    let xhr = new XMLHttpRequest();

    xhr.open('POST', route + id);

    xhr.setRequestHeader('Content-Type', 'application/json;charset=UTF-8');

    xhr.onload = function () {

      resolve({

        status: this.status,

        statusText: xhr.statusText,

      });

    };

    xhr.onerror = function () {

      reject({

        status: this.status,

        statusText: xhr.statusText,

      });

    };

    xhr.send(JSON.stringify(body)); // transmit the object converted to JSON

    xhr.onreadystatechange = function () {

      if (xhr.readyState === 4) { // if HTTP request is DONE

        if (xhr.status === 200) { // if status is ok

          dom('message').innerHTML = 'Success. Item actualizat.';

          resolve();

       } else {

          dom('message').innerHTML = 'Error. ' + xhr.responseText;

        }

      }

    };

  });

};

const deleteEntityServer = (id, route) => {

  return new Promise(function (resolve, reject) {

    var xhr = new XMLHttpRequest();

    xhr.open('DELETE', route + id);

    xhr.onload = function () {

      if (xhr.status == 200) {

        resolve({

          status: this.status,

          statusText: xhr.statusText,

        });

      } else {

        resolve({

          status: this.status,

          statusText: xhr.statusText,

        });

      }

    };

    xhr.onerror = function () {

      reject({

        status: this.status,

        statusText: xhr.statusText,

      });

    };

    xhr.send();

  });

};

# MVC

Cealaltă opțiune pentru a crea aplicații Web este paradigma MVC (Model-View-Controller). Aceasta presupune separarea logicii de business față de considerentele interfeței cu utilizatorul. Însă MVC nu necesită o aplicație de FE, deoarece View-ul (care este server-side) trimite direct către pagina Web conținutul de afișat. Mai precis, serverul generează inclusiv paginile HTML de afișat.

Controllerul este principalul punct al aplicației, acesta realizând operații pe baza modelului și trimițând catre view elementele de afișat.

Controllerul are următoarea structură:

exports.renderAddEchipa = async (req, res) => {

  try {

    const l = await bllLigi.getLigi();

    res.render('echipe/add', {title: 'Add team', l: l});

  } catch (err) {

    res.send(err.message);

  }

};

exports.addEchipa = async (req, res) => {

  const result = await bllEchipe.addEchipa(req.body);

  if (result === 'success') {

    res.redirect(`/echipe/getAll`);

  } else {

    res.status(400).send(result);

  }

};

exports.renderGetEchipa = async (req, res) => {

  try {

    const s = await bllEchipe.getEchipaById(req.params.id);

    if (s) {

      res.render('echipe/get', {title: 'Team', s: s});

    } else {

      res.redirect('http://localhost:3000');

    }

  } catch (err) {

    res.send(err.message);

  }

};

exports.renderGetAllEchipe = async (req, res) => {

  try {

    const s = await bllEchipe.getAllTeams();

    res.render('echipe/getAll', {title: 'Teams list', s: s});

  } catch (err) {

    res.send(err.message);

  }

};

exports.renderUpdateEchipa = async (req, res) => {

  try {

    const s = await bllEchipe.getEchipaById(req.params.id);

    const l = await bllLigi.getLigi();

    if (s) {

      res.render('echipe/update', {title: 'Echipa', s: s, l: l});

    } else {

      res.redirect('http://localhost:3000');

    }

  } catch (err) {

    res.send(err.message);

  }

};

exports.updateEchipa = async (req, res) => {

  const result = await bllEchipe.updateEchipa(req.params.id, req.body);

  if (result === 'success') {

    res.redirect(`/echipe/get/${req.params.id}`);

  } else {

    res.status(400).send(result);

  }

};

exports.deleteEchipa = async (req, res) => {

  const result = await bllEchipe.deleteEchipa(req.params.id);

  if (result === 'success') {

    res.redirect(`/echipe/getAll`);

  } else {

    res.status(400).send(result);

  }

};

# Nivelurile aplicației WEB

Proiectul este structurat pe nivelurile aplicațiilor Web cunoscute:

* *Nivelul de prezentare*, reprezentând interfața disponibilă utilizatorului. Este realizată prin fișiere HTML sau prin view-uri în cazul MVC;
* *Nivelul aplicație* primește cererile de la front-end și controlează operațiile de efectuat în Back-End. În cadrul proiectului, acest nivel este întâlnit în fișierele din folderul *routers;*
* *Nivelul servicii (BLL)* realizează operații mai complexe care țin de logica aplicației. De exemplu, în cazul acestui proiect, pentru adăugarea unui transfer se realizează operațiile următoare:
* Validarea datelor primite de la front-end
* Căutarea în baza de date a jucătorului de transferat
* Verificarea că echipa către care este transferat jucătorul nu este aceeași cu vechea sa echipă
* Verificarea ca suma de transfer să fie cel puțin 0.7 din valoarea jucătorului
* Verificarea să existe atât echipa veche, cât și cea nouă în baza de date
* Verificarea ca bugetul echipei care realizează transferul să fie suficient
* Modificarea bugetelor celor două echipe
* Modificarea echipei jucătorului
* Introducerea informației despre transfer în tabelul corespunzător pentru a avea istoricul transferurilor
* Ștergerea din cache a cheilor pentru care s-au modificat valorile

const addTransfer = async ({idJucator, idEchipaNoua, pret}) => {

  if (!idJucator || !idEchipaNoua || isNaN(pret) || pret <= 0) {

    return 'validation error';

  }

  const jucator = await dalJucatori.getJucatorById(idJucator);

  if (typeof jucator === 'string' || jucator.deleted) {

    return 'jucator not found';

  }

  if (jucator.idEchipa == idEchipaNoua) { return 'aceeasi echipa'; }

  if (pret < 0.7 \* jucator.valoare) { return 'pret prea mic'; }

  const echipaVeche = await dalEchipe.getEchipaById(jucator.idEchipa);

  if (typeof echipaVeche === 'string' || echipaVeche.deleted) {

    return 'echipa not found';

  }

  const echipaNoua = await dalEchipe.getEchipaById(idEchipaNoua);

  if (typeof echipaNoua === 'string' || echipaNoua.deleted) {

    return 'echipa not found';

  }

  if (echipaNoua.buget < pret) { return 'buget insuficient';}

  let resultUpdate;

  echipaVeche.buget += pret;

  resultUpdate = await dalEchipe.updateEchipa(echipaVeche.idEchipa, echipaVeche);

  if (resultUpdate !== 'updated') { return 'eroare update';}

  echipaNoua.buget -= pret;

  resultUpdate = await dalEchipe.updateEchipa(idEchipaNoua, echipaNoua);

  if (resultUpdate !== 'updated') { return 'eroare update';}

  jucator.idEchipa = idEchipaNoua;

  resultUpdate = await dalJucatori.updateJucator(idJucator, jucator);

  if (resultUpdate !== 'updated') { return 'eroare update'; }

  const transfer = {

    idJucator,

    idEchipaVeche: echipaVeche.idEchipa,

    idEchipaNoua,

    pret,

  };

  const resultCreate = await dalTransferuri.addTransfer(transfer);

  if (resultCreate !== 'created') { return 'eroare create'; }

  myCache.del(['allTransfers',`player\_${idJucator}`,

    `team\_${idEchipaNoua}`,`team\_${echipaVeche.idEchipa}`]);

  return 'success';

};

* *Nivelul de acces la date (DAL)* realizează operațiile cu baza de date. De exemplu:

const addTeam = async (body) => {

  body.deleted = false;

  try {

    await echipe.create(body);

    return 'created';

  } catch (err) {

    console.log(err);

    return err.message;

  }

};

const getAllEchipe = async () => {

  try {

    const result = await echipe.findAll();

    const teams = [];

    for (let i = 0; i < result.length; i++) {

      if (!result[i].dataValues.deleted) {

        teams.push(result[i]);

      }

    }

    return teams;

  } catch (err) {

    return 'error';

  }

};

const getEchipaById = async (id) => {

  const s = await echipe.findAll({where: {idEchipa: id}});

  if (s[0]) {

    return s[0].dataValues;

  } else {

    return 'notFound';

  }

};

const getEchipaByName = async (nume) => {

  const s = await echipe.findAll({where: {nume}});

  if (s[0]) {

    return s[0].dataValues;

  } else {

    return 'notFound';

  }

};

const getEchipeByLeagueId = async (id) => {

  const s = await echipe.findAll({where: {idLiga: id}});

  if (s[0]) {

    return s;

  } else {

    return 'notFound';

  }

};

const updateEchipa = async (idEchipa, body) => {

  try {

    await echipe.update(body, {where: {idEchipa: idEchipa}});

    return 'updated';

  } catch (err) {

    console.log(err);

    return err.message;

  }

};

# Specificații de implementare

## Ștergerea datelor

Ștergerea datelor se realizează în două moduri: logic și fizic. Ștergerea logică - realizată pentru entitățile ligi, echipe și jucători - are rolul de a nu pierde datele de care mai este nevoie în aplicație. De exemplu, dacă s-ar șterge un jucător acesta nu ar mai apărea în lista transferurilor realizate. Ștergerea logică se realizează prin setarea unui parametru *deleted* pe 1 în baza de date. Înregistrările care au *deleted:1* vor fi ignorate în afișarea listei de înregistrări de tipul acesteia.

const deleteEchipa = async (id) => {

  const echipa = await dalEchipe.getEchipaById(id);

  if (typeof echipa === 'string') { return 'not exists'; }

  echipa.deleted = true;

  const resultUpdate = await dalEchipe.updateEchipa(id, echipa);

  if (resultUpdate !== 'updated') { return resultUpdate; }

  myCache.del(['allTeams', `team\_${id}`, `league\_${echipa.idLiga}`]);

  return 'success'; };

Ștergerea fizică a datelor presupune ștergerea propriu-zisă a înregistrărilor din baza de date.

const deleteTransfer = async (id) => {

  var transferuri = await models.transferuri;

  await transferuri.destroy({where: {idTransfer: id}});

};

## Introducerea datelor

Pentru introducerea datelor pentru entitățile la care se realizează ștergerea logică, mai întâi se verifică dacă această înregistrare există deja și are câmpul *deleted:1*. Dacă este găsită, se va actualiza această înregistrare, punând valoarea 0 în coloana deleted. Dacă nu se găsește înregistrarea, aceasta va fi inserată în mod obișnuit.

const addEchipa = async (body) => {

  if (

    !body.nume.trim().length ||

    !body.idLiga ||

    isNaN(body.buget) ||

    body.buget <= 0

  ) {

    return 'validation error';

  }

  const leagueResult = await dalLigi.getLigaById(body.idLiga);

  if (leagueResult === 'notFound') {

    return 'league not exists';

  }

  const teamResult = await dalEchipe.getEchipaByName(body.nume);

  if (typeof teamResult !== 'string') {

    if (!teamResult.deleted) {

      return 'team exists';

    } else {

      if (body.idLiga != teamResult.idLiga) {

        return 'different league';

      }

      teamResult.deleted = false;

      teamResult.buget = body.buget;

      const resultUpdate = await dalEchipe.updateEchipa(

        teamResult.idEchipa,

        teamResult

      );

      if (resultUpdate !== 'updated') {

        return resultUpdate;

      }

    }

  } else {

    const addResult = await dalEchipe.addTeam(body);

    if (addResult !== 'created') {

      return addResult;

    }

  }

  myCache.del('allTeams', `league\_${body.idLiga}`);

  return 'success';

};

## Cache

Cache-ul are rolul de a realiza operațiile mai rapid fără a fi nevoie de apeluri către baza de date. Acesta este util pentru operatiile de GetEntity și GetAllEntities. Când anumite date se modifică, prin operații de insert, update sau delete, cheile din cache trebuie invalidate.

const getAllTeams = async () => {

  const getFromCache = myCache.get('allTeams');

  if (getFromCache) {

    console.log('FROM CACHE');

    return getFromCache;

  }

  result = await dalEchipe.getAllEchipe();

  myCache.set('allTeams', result);

  console.log('FROM DB');

  return result;

};

const getEchipaById = async (id) => {

  const getFromCache = myCache.get(`team\_${id}`);

  if (getFromCache) {

    console.log('FROM CACHE');

    return getFromCache;

  }

  result = await dalEchipe.getEchipaById(id);

  liga = await dalLigi.getLigaById(result.idLiga);

  result.numeLiga = liga.nume;

  const jucatori = await dalJucatori.getJucatoriByTeamId(id);

  result.jucatori = [];

  if (typeof jucatori !== 'string') {

    for (let i = 0; i < jucatori.length; i++) {

      if (!jucatori[i].dataValues.deleted) {

        result.jucatori.push(jucatori[i].dataValues);

      }

    }

  }

  myCache.set(`team\_${id}`, result);

  console.log('FROM DB');

  return result;

};

## Validarea datelor

Validarea datelor se realizează atât pe Front-End - la validarea datelor introduse în interfață, cât și pe Back-End, în cadrul nivelului Servicii.

if (

    !nume.trim().length ||

    !nationalitate.trim().length ||

    !idEchipa ||

    isNaN(valoare) ||

    parseInt(valoare) <= 0

  ) {

    alert('Something wrong');

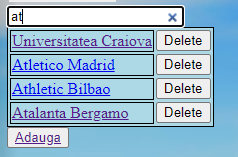
  }

Avantajul validării atât pe FE, cât și pe BE este următorul:

* Dacă se găsește o dată invalidă pe FE, nu mai este necesar un apel costisitor (ca timp) către Back-End
* Dacă se realizează validarea pe Back-End, vom avea luate în considerare și eventualele date introduse greșit din alte surse (de exemplu, prin Postman). Totodată, unele câmpuri (cele care necesită apeluri către baza de date) pot fi validate doar în Back-End.

## Filtrarea datelor în interfață

O altă funcționalitate adăugată proiectului este filtrarea datelor din interfață. Dacă sunt prea multe date afișate, ar putea fi dificilă găsirea celei pe care o căutăm. De aceea, este necesară o căsuță de căutare, care să filtreze rezultatele afișate pe baza a ceea ce căutăm.



# Concluzii și posibilități de dezvoltare

În concluzie, prin dezvoltarea acestui proiect consider ca mi-am îmbunătățit abilitățile și cunoștințele despre dezvoltarea aplicațiilor web.

În același timp, aplicația poate fi dezvoltată pe diverse planuri. În primul rând, ar fi necesară introducerea unui sistem de logare pentru administratorii site-ului și doar ei să aibă acces să introducă date noi.

De asemenea, partea de Front-End s-ar putea îmbunătăți prin utilizarea unor framework-uri, cum ar fi ReactJS sau Angular.

Totodată, s-ar putea realiza conexiunea cu anumite jocuri (de exemplu FIFA) și realizarea transferurilor și în acele jocuri.