Laboratorul 01: Introducere in Tehnologiile Web

Ce veti invata la aceasta materie?

Domeniul web a evoluat foarte mult in ultimii ani. Daca acum 10-15 ani web-ul era dominat de website-uri simple, acum ne aflam in plina ascensiune a platformelor complexe, a serviciilor cloud si a API-urilor care furnizeaza non-stop date.

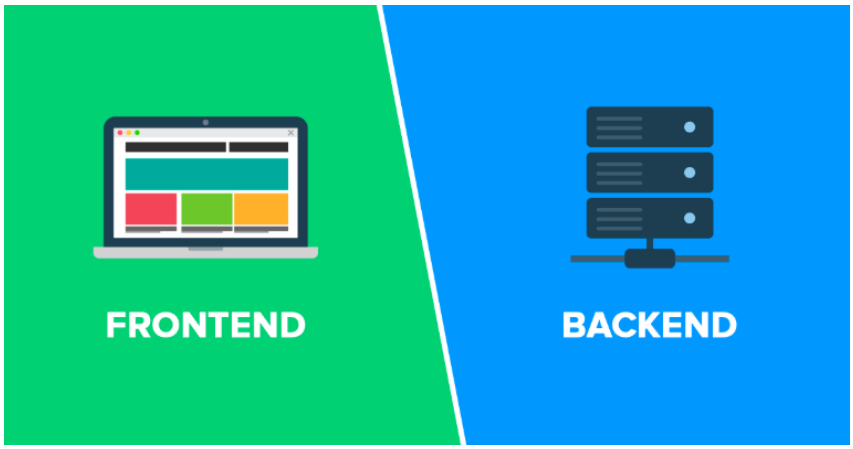
In cadrul acestui laborator veti invata notiunile de baza pentru a putea realiza o **platforma web de complexitate medie**. Veti lucra cu un model arhitectural web modern si veti folosi tehnologii de actualitate, utilizate preponderent in industrie.

Cuprins

Structura unei Aplicatii Web

O aplicatie web tipica este impartita in **backend** si **frontend**:

- Backendul este responsabil de procesele intensive dpdv. computational si de prelucrarea si furnizarea datelor, deci CPU si IO intensive stuff.
- Frontendul este responsabil de reprezentarea datelor intr-o maniera cat mai estetica si organica, astfel incat utilizatorul sa aiba productivitate maxima utilizand aplicatia, deci scop preponderent reprezentativ.



Aceasta structura bipartita are la baza modelul clasic de **client - server** studiat la Protocoale de Comunicatie.

Arhitectura unei Aplicatii Web

Arhitectura web este un termen umbrela, fiind, de fapt, un subiect foarte complex. Arhitectura web trateaza atat structura individuala a componentelor unei aplicatii, cat si interactiunea dintre acestea.

Asadar, cand discutam de arhitectura web, ne referim la arhitectura backendului, arhitectura frontendului, structura datelor care vor fi folosite si modul in care acele date vor fi folosite.

In cadrul acestui laborator vom folosi urmatoarele concepte arhitecturale:

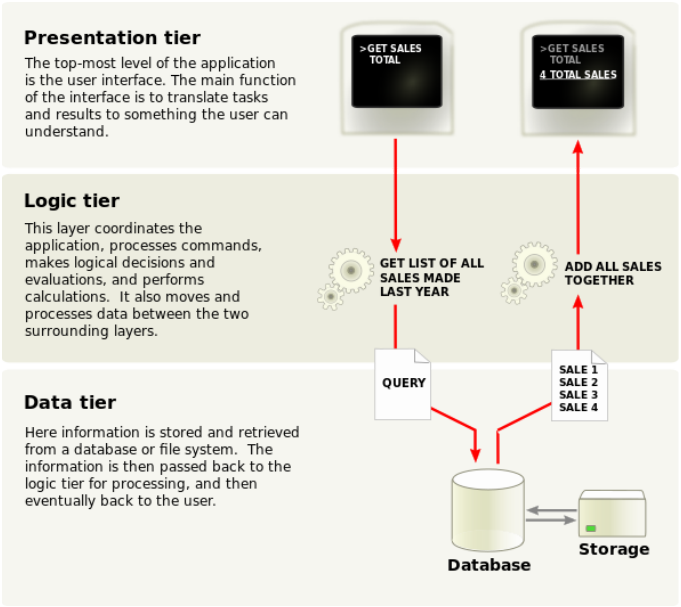
- **N-tier**, pentru modul in care este structurata logic si fizic aplicatia
- **Monolith**, pentru modul in care va fi structurat backendul
- **Single Page Application**, pentru modul in care va fi structurat frontendul
- **HTTP REST API**, pentru modul in care vor comunica backendul si frontendul
- **Relational** si **Non-relational**, pentru modul in care vor fi structurate datele

N-tier

Arhitectura N-tier, denumita si **multitier** sau **multilayered** este un model arhitectural foarte des intalnit in cadrul aplicatiilor web. Aceasta se bazeaza pe separarea aplicatiei in parti sau domenii **logice** , denumite straturi. Fiecare strat rezolva cate o problema, staturile comunicand intre ele.

O reprezentare simpla a arhitecturii N-tier este modelul 3-tier, care va fi folosit si de noi:

- **Presentation Layer** , UI, cu rol estetic, de prezentare a datelor
- **Business Logic Layer** , logica din spatele UI, backendul si comunicarea dintre acestea
- **Data Layer** , accesul la date, modul in care sunt stocate datele



Monolith & Microservices

Daca N-tier discuta separatia logica intre problemele de business ale unei aplicatii (straturile), arhitectura monolith se refera la modul in care software-ul este structurat fizic.

Principiul de baza din spatele arhitecturii monolith este ca software-ul este dezvoltat integral, dintr-o bucata. Chiar daca este structurat logic in module multiple care trateaza diferse probleme de business, acestea pornesc toate odata, sub forma unui proces rezultat in urma compilarii. Cand se da deploy, se da deploy la tot odata si cand se face update, trebuie inlocuit, pe serverul de deployment, intreg software-ul cu versiunea noua.

In ziua de azi, cele mai intalnite doua modele arhitecturale care discuta structura fizica a unui software sunt monolith si **microservices**. Chiar daca o abordare pe microservicii este, de multe ori, mai buna, monolith-ul este mult mai didactic si usor pentru incepatori. Puteti considera ca orice aplicatie facuta de voi pana acum in facultate, la teme, de exemplu, este monolit. Deoarece acest laborator are scop introductiv in tehnologiile web, backendul pe care il vom dezvolta se va realiza in format monolit, si nu distribuit, pe microservicii.

Single Page Application

SPA-ul, alaturi de fratele sau, MPA (**multi-page applications**) reprezinta modurile principale de organizare a frontendului. Asa cum numele implica, SPA reprezinta un frontend in care interactiunile cu diverse componente nu rezulta intr-un **refresh** in browser, asa cum se intampla in cadrul MPA.

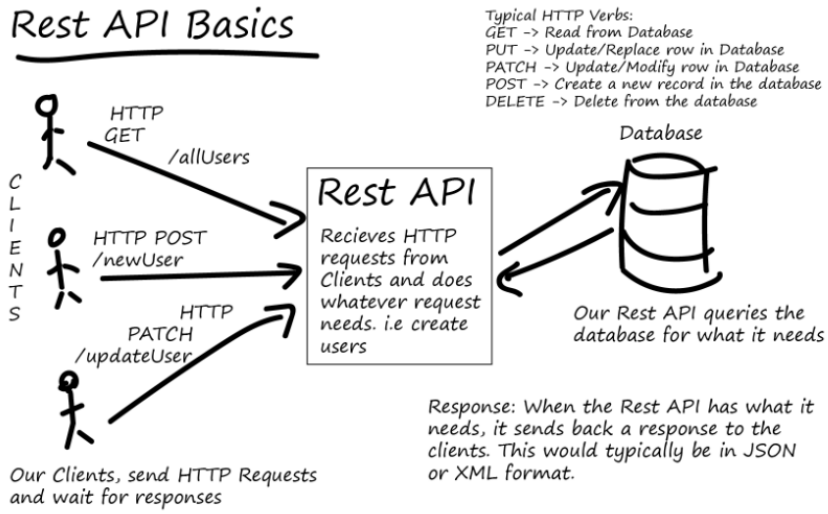
SPA presupune ca scripturile necesare rularii frontendului sunt aduse de la server doar o data, la inceput, in browser si apoi frontendul comunica cu serverul doar prin requesturi **AJAX** pentru a prelua date. In cadrul MPA, fiecare pagina noua este adusa de la server.

Avantajul principal al SPA este ca utilizeaza la maxim puterea browserului pentru a compila si afisa UI-uri complexe, spre deosebire de MPA, unde paginile sunt pre-randate pe server. Dezavantajul principal al SPA este dificultatea in realizarea SEO-ului. Din acest motiv, majoritatea site-urile de e-commerce sunt MPA, in timp ce platformele web, care nu au nevoie neaparata de SEO, sunt SPA.

HTTP Rest API

Representational State Transfer (REST) este un stil arhitectural modern care defineste modul in care interactioneaza serviciile web. Acesta se bazeaza pe **cereri HTTP** (e.g.: GET, POST, PUT, DELETE) pentru a manipula informatia si pe **media-types** (e.g.: JSON) pentru a stabili cum arata informatia transferata.

Un API de tip REST este un serviciu Web care permite interactiunea prin mecanisme REST. Backendul pe care noi il vom dezvolta la laborator va fi un astfel de API.



Relational si Non-Relational

Nu in ultimul rand, o aplicatie are nevoie de date. Aceste date, de cele mai multe ori, se vor afla in baze de date. Termenii relational si non-relational descriu structura datelor. Relational se refera la **date structurate tabelar**, accesate prin SQL, in timp ce non-relational se refera la date care nu sunt sunt structurate tabelar, ci sunt stocate in diverse formate (cheie-valoare, obiecte json, graf, etc…).

Tehnologii folosite in cadrul laboratorului

La laborator vom folosi **NodeJS** pentru backend, **PostgreSQL** pentru baza de date relationala, **MongoDB** pentru baza de date non-relationala si **React** pentru frontend.

Bazele de date vor fi containere de **Docker**. Testarea se va realiza cu **Postman**.

Pentru versionare, vom folosi **Git** si **Gitlab**.

Este foarte important ca la finalul fiecarui laborator sa va urcati codul pe repository, deoarece laboratoarele se leaga intre ele.

Familiarizare Javascript

NodeJS este un mediu de rulare pentru Javascript bazat pe motorul V8 de la Chrome. Acesta permite rularea scripturilor de JS pe calculator, fara sa fie nevoie de browser.

Deoarece backendul si frontendul vor fi realizate folosind tehnologii care se bazeaza pe Javascript, este elementar sa cunoasteti acest limbaj. Javascript este un limbaj de scripting cu o sintaxa foarte usoara, asemanatoare celei din C. Va prezentam mai jos cateva particularitati ale limbajului:

- JS este un limbaj slab tipat. Variabilele sunt definite nu cu tipuri, dar cu **let** sau **const**
- Functiile nu au un tip de retur si de asemenea nici parametrii nu au tipul atasat
- Functiile pot fi definite folosind keyword-ul **function** sau folosind **arrow functions**

```
function myFunc(x) {console.log(x)};

// este identic cu

const myFuncArrow = (x) => console.log(x);

myFunc(3); //3
myFuncArrow(3); //3
```

- deoarece JS este un limbaj interpretat, nu conteaza ordinea in care sunt definite functiile. Totusi, este indicat sa fiti cat mai organizati in cod

- JS nu are nevoie de o functie de **main** ca sa ruleze. Codul dintr-un fisier .js va rula in momentul in care scriptul este apelat

```
const constanta = 'abc';
let variabila = 3;

function adauga (x, y, z) {
  if (x === 'abc') {
    y = y + z;
  }
  return  y;
}

const rezultat = adauga(constanta, variabila, 10);

console.log(rezultat);
```

- in JS exista doua tipuri de egalitate: **==** si **===**. Primul tip de egalitate nu tine cont de tipurile variabirelor comparate, in timp ce al doilea tip este mai strict.

```
const a = 10; //a este numar
const b = '10'; //b este sir de caractere

console.log(a == b); //true
console.log(a === b); //false

const c = 'abc';
const d = 'abc';

console.log(c == d); //true
console.log(c === d); //true

const fals = false; //boolean
const str = ''; //string
const zero = 0; //number

console.log(fals == str); //true
console.log(fals == zero); //true
console.log(str == zero); //true

console.log(fals === str); //false
console.log(fals === zero); //false
console.log(str === zero); //false
```

- JS este un limbaj semi-oop. Cu toate ca are concepte precum obiecte si clase (partial), principiile OOP nu se regasesc in totalitate
- Vectorii sunt declarati cu paranteze drepte **[ ]** si obiectele cu paranteze acolade **{ }**
- Obiectele in JS sunt combinatii de chei valori, unde valorile pot fi orice: variabile, vectori, obiecte si chiar functii
- Accesul la elementele dintr-un obiect se face in doua moduri:

```
const obj = {a:1, b:2};

console.log(obj.a); //1
console.log(obj['b']); //2
```

- Deoarece JS opereaza cu obiecte, exista doua tipuri de copiere: prin referinta, implicit si prin valoare

```
const obj1 = {a:2};
const obj2 = obj1; // copiere prin referinta

obj2.a = 5;
console.log(obj1); // {a:5} -> valoarea s-a modificat si in obiectul original

const obj3 = Object.assign({}, obj1); // copiere prin valoare

const obj4 = JSON.parse(JSON.stringify(obj1)); // copiere prin valoare folosind JSON

obj3.a = 10;
console.log(obj1); // {a:5} -> valoarea nu s-a modificat

const obj4 = {...obj1} // copiere prin valoare (modern, folosind spread operator, ES6)

obj4.a = 9772;
console.log(obj1); // {a:5} -> valoarea nu s-a modificat
```

- JS este un limbaj **functional**. Prezinta concepte precum metode functionale (map, filter, reduce) si functii curry (callback-uri)

```
const arr = [1, 2, 3, 4];
console.log(arr.map(x => x*2)); //2 4 6 8

const obj = { a:2, b:3, c: (x, y) => console.log(x + y)}
console.log(obj.c(obj.a, obj['b'])); //5

const func1 = (x, cb) => cb(x);
const func2 = y => console.log(y);

func1(3, func2); //3
```

Cod asincron

JS ofera suport pentru executia codului **asincron**. Majoritatea operatiilor care se desfasoara in cadrul unui API sunt asincrone

De exemplu, accesul unei resurse din baza de date dureaza timp. Daca accesul ar fi blocant, tot serverul s-ar bloca pana cand resursa ar fi preluata. Din acest motiv, acest acces trebuie facut asincron

Codul asincron implica mecanisme de sincronizare logice. Toate operatiile asincrone se realizeaza in background, in **bucla de eveniment [https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick/]**.

Pentru sincronizarea logica a codului asincron se pot folosi **promises** sau **async/await** (recomandat).

- promisiuni - Obiecte ce contin doua functii curry, resolve si reject pentru manipularea sincrona a codului asincron. O promisiune se asteapta folosind **then** sau **catch**.

```
const getAgePromise = new Promise((resolve, reject) => {
    resolve(25);
});

getAgePromise.then((age) => console.log(`My age is ${age}`));
```

- async/await - Syntactic sugar pentru promisiuni, ofera aceeasi functionalitate, dar intr-o maniera mai imperativa, asemanatoare codului sincron.

```
const getAgeAsync = () => new Promise((resolve, reject) => {
    resolve(25);
});

// echivalent cu const getAgeAsync = async () => 25;

const main = async () => {
    const age = await getAgeAsync();

    console.log(`My age is ${age}`);
}

main();
```

Pentru a sincroniza un cod folosind promisiuni, se folosesc functiile **then** si **catch** care accepta ca si parametru o functie curry ce va contine rezultatele intoarse de executia asincrona a codului, sau eroare.

Pentru a sincroniza un cod folosind async/await, se foloseste cuvantul cheie **await** care asteapta rezultatul intors de functia asincrona. Pentru a folosi **await**, este nevoie ca functia parinte sa fie declarata cu **async**.

O functie **async** va returna intotdeauna o promisiune

```
const fs = require('fs');
const fsPromise = require('fs/promises');

const main = async () => {

    console.log('Voi citi continul fisierului sincron');

    const continutSincron = fs.readFileSync('./text.in');

    console.log(`Continutul sincron este: ${continutSincron}\n----------------\n`);

    console.log('Voi citi continul fisierului cu promisiune');

    let continutPromisiune = "NaN";

    fsPromise.readFile('text.in').then(continutPromisiune => console.log(`Continutul promisiune este: ${continutPromisiune}\n----------------\n`));

    console.log(`Continutul promisiune in afara promisiunii este: ${continutPromisiune}\n----------------\n`);

    console.log('Voi citi continul fisierului cu async await');

    let continutAsyncAwait = "NaN";
```

```
    continutAsyncAwait = await fsPromise.readFile('text.in');

    console.log(`Continutul async await este: ${continutAsyncAwait}\n----------------\n`);
}


main();
```

Rulati codul de mai sus. Ce observati? Scoateti cuvantul cheie await si rulati din nou. Ce observati?

Familiarizare Docker

Chiar daca la acest laborator **nu vom invata Docker**, vom folosi aceasta tehnologie pentru containerizarea bazelor de date.

Un container reprezinta o rulare abstractizata a unui program, fara sa fie nevoie de instalarea dependentelor in prealabil.

Este important sa intelegeti urmatoarele concepte de docker:

- **Retele Docker** - Orice container docker ruleaza intr-o retea interna. Pentru a putea comunica cu un container, din sistemul gazda, este necesara maparea unui port de pe sistemul gazda, catre containerul docker.
- **Volume Docker** - Containerele reprezinta entitati efemere. Odata ce un container este stins, orice informatie care a fost scrisa in interiorul lui, pe perioada rularii, este pierduta. Din acest motiv, cand vom lucra cu baze de date, vom folosi mapari de volume intre sistemul gazda si docker.
- **Docker Compose** - Unealta pentru rularea mai multor containere de docker, pornind de la o configuratie declarativa scrisa in format YAML. Echivalent a executarii comenzii docker run de mai multe ori.

Conceptele enumerate mai sus, vor fi folosite incepand cu laboratorul de baze de date. Va recomandam sa accesati resursele de la materia Cloud Computing (actual IDP) [https://ocw.cs.pub.ro/courses/cc] sau de la workshopul Moby Devops [https://ocw.cs.pub.ro/courses/moby] pentru o mai buna intelegere a Docker.

Exercitii

1. Setup Repo GitLab

Pentru inceput trebuie sa va setati repository-ul de Gitlab. Va rugam sa aveti in vedere urmatoarele:

- Sa fie privat
- Numele repo-ului sa fie pw2021-nume-prenume
- Sa dati acces asistentilor de laborator

2. Instalare software

Instalati urmatoarele

- NodeJS 14 LTS
- Postman

3. Basic Javascript

1. Rulati scripturile din laborator
2. Creati un script care sa afiseze “Hello World!”
3. Creati un script care sa afiseze ora si data curenta
4. Creati un script care populeaza un vector cu numere de la 0 la 100 si afiseaza doar numerele pare
5. Extindeti scriptul precedent si creati o functie care primeste ca parametru vectorul, un numar cu rol de index si o alta functie care afiseaza numarul aflat la pozitia index din vector
6. Creati o functie asincrona care returneaza un rezultat peste 2 secunde si asteptati-o. Folositi atat promisiuni, cat si async await.