HTML, CSS, JavaScript – client-side

!!! JavaScript a fost gândit ca limbaj de front-end (teoretic avem nevoie doar de browser pentru a vizualiza fișiere js), dar sunt situații în care JavaScript se folosește pe back-end (NodeJS).

HTTP/S, Ajax – legătura dintre front-end și back-end

Obs interesantă: Majoritatea serverelor web, când descărcăm un document (ex: o pagină HTML, un fișier CSS etc.), *arhivează* acele documente, iar ceea ce se trimite efectiv pe socket între serverul web și browser-ul web este un fel de arhivă zip. Browser-ul o să facă *decriptarea (dezarhivarea)* de la zip la fișier normal.

PHP, Java (JSP, Servlets) – server-side

Pentru un server web, nu trebuie neapărat ca clientul să fie un browser. Poate, pentru un server (aplicație) web, clientul e o altă aplicație web.

client web (browser) → (HTTP) → server web (aplicație web)

client web (browser) → server web (aplicație web) → server web (aplicație web) → server web (aplicație web) → server web (aplicație web)

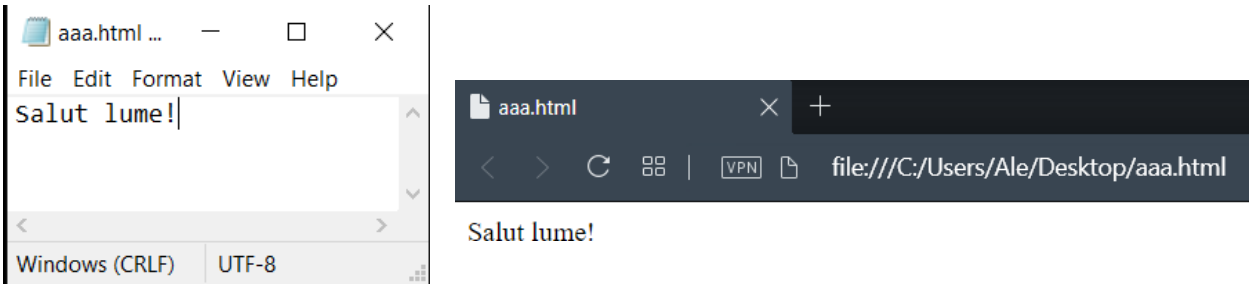
Cea de mai sus: aplicație *n-tier*.

!!! Într-o aplicație n-tier, foarte multe dintre cele de mijloc nu sunt neapărat aplicații web, ci sunt *servicii* web.

Aplicație web – destinată să fie acceptată de un client web în spatele căruia se află un utilizator uman.

Serviciu web – menit să ofere funcționalitate și conținut unei alte aplicații web sau unui alt serviciu web.

Avem următorul fișier HTML pe care îl deschidem într-un browser (client):



Putem folosi un browser nu neapărat pe post de client pentru un server web, ci îl putem folosi inclusiv pe post de *file-viewer*. Pagina respectivă ar trebui să fie vizualizată corect, cât timp pagina web folosește exclusiv tehnologii client-side (nu are nevoie de nimic de la server).

HTML

<!DOCTYPE html>

- declarație menită să ajute aplicațiile care au nevoie să parseze documentul HTML
- e folosită și de către editor
- descrie ce tip de HTML urmează în continuare (HTML5)

La un moment dat, s-a încercat standardizarea limbajului HTML spre XML (XHTML). Până la urmă, evoluția naturală a HTML nu a fost spre XML, ci spre HTML5.

Un browser se descurcă chiar dacă facem "*spaghetti code*" (amestecăm tag-uri / atribute specifice HTML4 cu cele specifice HTML5).

Există două tipuri de tag-uri:

- care au marcaj de început și de sfârșit:
  <tagname> … </tagname>
- care au doar marcaj de început:
  <tagname>

!!! Unele tag-uri necesită neapărat un atribut. De ex: *<img src = "poza.jpg">*.

<input readonly> → până la HTML4 trebuia scris *<input readonly="readonly">*. Atenție: nu există readonly="true" **!!!**

Ce făceau browserele: dacă apărea atributul *readonly*, indiferent ce valoare se dădea, făcea input-ul să fie readonly, indiferent dacă sintaxa era corectă sau nu. Din cauza asta, HTML5 a venit cu faptul că nu mai trebuie să atribuim valoare la readonly.

 → HTML

 → XHTML

Click here → o să deschidă într-un tab nou numit "blană"

Click here → la apăsare o să ducă la secțiunea de pagină *end* botezată mai jos.

…

HTML5 a încercat să pună accentul pe *semantica* documentului.

```
<ul>
        ceva → nu e corect !!!
        <li> … </li>
        <li> … </li>
</ul>
```

```
<!DOCTYPE html>
<html>
        <head>
                chestii ce descriu documentul
        </head>
        <body>
                ce se afișează în pagină
        </body>
</html>
```

În HTML4 e obligatoriu să respectăm structura de mai sus, însă în HTML5 putem scrie și ceva de genul:

```
<!DOCTYPE html>
<html>
        <title> Titlu <title>
        conținut
</html>
```

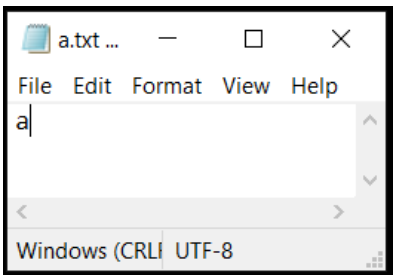
!!! Treaba HTML-ului e să dicteze cum e structurat documentul, nu cum arată.

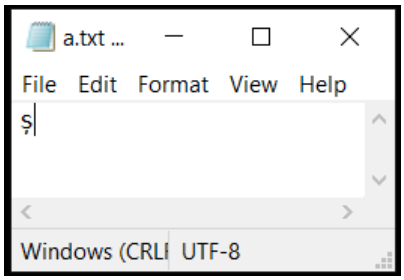
Tot ce e tag și atribut care specifică cum trebuie să arate un element din pagină (*b*, *center*, *bgcolor*, *align*, *valign* etc.) e deprecated în HTML5. În HTML5 au fost introduse elemente care mai degrabă descriu ce fel de element se găsește în pagină.

Diferența dintre tag-ul **b** și tag-ul **strong**: b a fost primul introdus în HTML, care doar boldează textul; tag-ul strong nu spune că textul trebuie să fie boldat, ci doar zice că este important (ajută la *accesibilitatea* paginii web).

Curs 3

Encoding





Primul document are dimensiunea de 1 octet, iar al doilea are 4 octeți. Diacriticile nu încap în reprezentarea simplă pe 1 octet.

La subtitrări, se salvau în fișierele cu subtitrări diacriticile tot pe 1 octet, însă octetul era afișat vizual ca un anumit caracter în funcție de *encoding*. Ca să se evite toată problema asta, standardul UTF-8 a determinat ca un caracter "special" să se reprezinte pe mai mult de un octet.

Se pune problema unui browser să afișeze bine fluxul de octeți pe care îl citește. În antetul paginii trebuie specificat encoding-ul octeților citiți:

<meta http-equiv="*Content-Type*" content="*text/html*"; charset="*UTF-8*"> → HTML4

<meta charset="*UTF-8*"> → HTML5

!!! Encoding-ul poate fi modificat și de programator, și de utilizator.

Entități HTML: încep cu **&** și se termină cu **;**

Ex: → entitate pentru spațiu

© → ©

Există entități HTML pentru diacritice (doar î și â). Așa nu avem probleme cu encoding-ul.

Formulare WEB

<input type=”*radio*” name=”*sex*” id=”*sex*” value=”*M*”>

<input type=”*radio*” name=”*sex*” id=”*sex*” value=”*F*”>

!!! Nu putem avea două câmpuri cu același id, dar putem avea cu același nume. Acest lucru determină gruparea lor, iar atunci când încercăm să bifăm un câmp, se debifează celălalt. Dacă ar avea nume diferite, ambele ar rămâne bifate.

<form method=”*get*” action=”*faCeva.php*”> → adresa metodei de pe backend care va prelua datele din form

Cum facem dacă avem două input-uri de tip *submit* și vrem pe backend să facă două lucruri diferite?

- un input îl facem de tip button
- le dăm nume diferite

!!! Dacă avem un input de tip file, metoda form-ului trebuie neapărat să fie POST → prin GET, datele sunt trimise prin query string, care e adăugat în cadrul URL-ului cu care ne referim la script-ul de pe backend; URL-ul are dimensiune limitată (aprox. 2 kb), deci nu putem trimite fișiere de dimensiuni mai mari

!!! Dacă avem un input de tip file, trebuie neapărat setat în tag-ul *form* atributul enctype=”*multipart/form-data*”

Diferența dintre atributul readonly și disabled: la submit, se trimite perechea *name=value* la cel cu atributul readonly, însă la cel cu disabled nu se va trimite.

Dacă avem:

<select name="*tara*" size="*10*" multiple>

<option value=”*1*”> Romania </option>

<option value=”*2*”> Ungaria </option>

</select>

Dacă utilizatorul selectează România și trimite formularul, în query string nu va apărea perechea *tara=Romania*, ci va apărea *tara=1*. Dacă nu exista atributul value, atunci ar fi fost tara=Romania.

Problema validărilor pe frond-end: tot ce ține de client nu este 100% safe: utilizatorul poate interveni în developer tools și să modifice. Rolul validărilor pe front-end poate fi doar de a face pagina cât mai user friendly.

<iframe></iframe> → se folosește ca să se introducă în pagină tot felul de widget-uri care sunt găzduite pe alte pagini (ex: clipuri pe youtube, butoane de like, tweet etc.)