Obiectivul final

La finalul acestui laborator vom avea o aplicație web care suportă autentificarea, pe mai multe nivele, a utilizatorilor, un sistem CRUD pentru utilizatori care să fie accesibil administratorilor. Aplicația va fi implementată ca până acum în PHP, dar conceptele se aplică pentru oricare dintre limbaje.

Concepte despre care vom discuta: persistența datelor între cereri HTTP, cookie-uri, sesiuni, autentificare, password hashing.

Puțină teorie - despre sesiuni

Țineți minte că unul din conceptele importante despre care am vorbit în primele laboratoare este că protocolul cu care lucrăm în aplicații web, HTTP, este un protocol stateless (fără stări). Pe scurt, asta înseamnă că, la nivel de protocol, HTTP nu face legătura între două cereri consecutive, chiar dacă vin de la același client. Fiecare cerere este independentă de cererea următoare. Acesta este unul din aspectele esențiale ale programării pentru web, de unde decurg multe din tehnicile diferite care se folosesc aici, dar nu și în programarea pentru desktop (transmiterea parametrilor prin GET sau prin POST, serializarea obiectelor, șamd).

Au fost totuși create mecanisme care ne permit să facem această legătură între două cereri diferite - ele fiind implementate în cele mai multe limbaje de programare pentru web și fiind suportate de cele mai multe browsere, fără să fie o din protocolul HTTP.

Cookies

Cookie-urile sunt șiruri de caractere care pot fi folosite pentru a crea o "*sesiune*" - o serie de cereri pe care aplicația web le poate trata ca și venind de la același client.

Mecanism-ul cookie-urilor este următorul: clientul face o cerere către server pentru un anumit URL, iar serverul răspunde cu conținutul de la URL-ul respectiv, dar și cu un header adițional, header-ul "Set-Cookie". Un răspuns cu cookie-uri arată cam așa (sursa: Wikipedia)

```
HTTP/1.1 200 OK
Content-type: text/html
Set-Cookie: name=value
Set-Cookie: name2=value2; Expires=Wed, 09 Jun 2021 10:18:14 GMT

(content of page)
```

Aici se setează două cookie-uri, cu numele *name* și *name2*. Pe lângă valoare, cookie-urile mai pot conține câteva informații despre valabilitate (orice cookie are o anumită valabilitate o oră, o zi, o săptămână ...) sau domeniul pentru care este valabilă, dar și alte opțiuni.

După acest pas în care server-ul ne trimite cookie-uri în răspuns, funcționarea cookie-urilor presupune ca browser-ul să trimită, cu fiecare cerere care urmează cererii la al cărui răspuns s-au primit cookie-urile, un header

```
Cookie: name=value; name2=value2
```

Browserele care au opțiunea de a folosi cookie-uri activată vor face acest lucru în mod automat. Cele mai multe browsere moderne au implicit activate cookie-urile, mai puțin în modul *stealth*

mode.

Puntem folosi acest mecanism cu cookie-uri pentru a simula o sesiune de lucru, și chiar și pentru a transmite anumiți parametri - deși de la un anumit nivel încolo acest lucru ar însemna să încărcăm fiecare cerere și fiecare răspuns cu o cantitate mare de date - pentru această funcționalitate vom folosi sesiuni, despre care discutăm un pic mai jos. Cookie-urile pot fi folosite (și se folosesc, în practică) nu doar pentru a implementa conceptul de sesiuni peste HTTP, dar și ca un mecanism naiv pentru a verifica daca un client a vizitat un site sau o anumită pagină a site-ului (spre exemplu, dacă a votat), sau dacă clientul este autentificat.

În PHP, pentru a seta un cookie putem folosi funcția

bool **setcookie** (string `$name` [, string `$value` [, int `$expire` = 0 [, string `$path` [, string `$domain` [, bool `$secure` = false [, bool `$httponly` = false]]]]])

iar pentru a accesa cookie-uri primite de la clienți putem folosi array-ul global $_COOKIE (la fel ca și cum am folosi $_GET sau $_POST).

Atenție! setcookie nu face altceva decât să seteze răspunsului un header adițional (un Set-Cookie). Pentru că răspunsul HTTP începe cu headerele, PHP va trimite toate headerele în momentul în care trebuie să trimită și conținutul răspunsului. Într-un sistem care folosește template-uri, singurele lucruri la care trebuie să fiți atenți sunt spații suplimentare înainte de deschiderea tag-urilor <?php în script-uri. Este important de reținut că orice *echo* care apare în codul vostru va necesita să se trimită conținutul răspunsului, deci și header-ele. În concluzie, setcookie trebuie apelat ÎNAINTE să trimitem orice către client.

Sesiuni

Pentru că ele trebuie trimise în toate cererile (și orice parametru nou în toate răspunsurile), cookie-urile nu sunt o metoda eficientă pentru a transmite parametrii între diferite cereri. Pentru a avea această funcționalitate, PHP implementează un mecanism de sesiune - fiecare sesiune de lucru primește un ID unic - acest ID, în funcție de setările PHP, este comunicat clientului printr-un Cookie (implicit) sau prin concatenarea unui parametru GET fiecărei cereri (configurabil). Pentru fiecare sesiune deschisă se va deschide și un fișier, pe server, în care se vor stoca date care vor fi accesibile în toate scripturile care operează cu acea sesiune.

Pentru a folosi sesiuni, este necesa să fie apelată metoda *session_start()* - această metodă inițializează datele de sesiune folosind fie Cookie-ul, fie un parametru GET / POST pentru a obține ID-ul sesiunii. Pentru a înregistra date în sesiune, dar și pentru a le putea obține în cereri ulterioare, putem folosi array-ul global $_SESSION.

Notă: unset(array[index]) și isset(array[index]) sunt două funcții care operează pe array-uri care sunt foarte utile în lucrulul cu $_SESSION.

Atenție! start_session() va folosi implicit același mecanism de cookie-uri pentru a trimite ID-ul sesiunii - deci trebuie luate aceleași precauții ca și în cazul setcookie.

Gata cu teoria, să vedem și ceva mai practic

În acest laborator ne propunem să implementăm un mecanism de autentificare. În cea mai de bază formă a acestuia, acesta va trebui să aibă următoarele feature-uri:

- un formular de autentificare, în care să cerem utilizatorului un user și o parolă
- un tabel într-o bază de date în care să reținem cel puțin username-ul și parola (eventual hash-uită) - eventual și rolurile pe care trebuie să le îndeplinească
- un model pentru utilizator (care să implementeze acțiunile minime)

- login
- logout
- o metodă de a face diferența între clienți autentificați și clienți neautentificați

Scheletul laboratorului conține un dispecer general și un stub (un început) pentru clasa Utilizator, dar și câteva template-uri html pentru diferitele acțiuni pe care vrem să le implementăm.

Utilizatori, parole - cum și unde se țin

Una din cele mai comune metode de autentificare în aplicații web este prin utilizarea unui nume de utilizator și a unei parole - text, pe care utilizatorul o primește sau și-o desemnează singur. În funcție de aplicația folosită, s-ar putea să avem datele de autentificare într-o bază de date (cazul elementar, pe care îl vom implementa și noi), dar și într-o varietate de surse de date (poate fi într-un director - LDAP - cum sunt utilizatorii sistemului Moodle pe care îl folosim la facultate, pentru autentificare centralizată, sau poate folosi OAuth, care este un sistem în care aplicații pot folosi un sistem de autentificare pus la dispoziție de provideri OAuth - Facebook, Yahoo, și Google sunt doar câteva exemple de provider OAuth).

În cazul nostru, vom ține informațiile despre utilizator - username, parola și nivel de acces într-un tabel în baza noastră de date.

Notă: În mod comun, nu se reține niciodată parolele utilizatorilor în clar - dintr-o varietate de motive (ce se întâmplă dacă îți sparge cineva baza de date, ce se întâmplă daca un cont de administrator al site-ului este compromis, ce se întâmplă dacă cineva se uită peste umărul unui administrator în timp ce lucrează cu tabelul de utilizatori ...). Se consideră o bună practică să păstrăm un *hash* al parolei în baza de date. Un *hash* este rezultatul trecerii parolei, ca șir de caractere, printr-o funcție hash. Foarte pe scurt, o funcție hash va transforma un șir de caractere în alt șir de caractere, care are următoarele caracteristici:

1. În general, rezultatul unei funcții hash are o lungime fixă (pentru md5, 32 de "cifre" hexa).
2. Rezultatul funcției pentru același string este întotdeauna același
3. Deducerea șirului original original din șirul rezultat rezultat este imposibilă sau aproape imposibilă (se spune imposibilă cu posibilitățile computaționale curente)

Cum folosim o astfel de variantă *hash* a parolei? Este destul de simplu, sunt două aspecte de avut în vedere:

1. Când se salvează parola utilizatorului în baza de date, se salvează hash-ul ei. Din nou, o bună practică este sa nu se facă hash doar pe parolă, ci să se adauge un string secret acesteia (numit *salt*) și eventual alte date ale utilizatorului (emailul, username-ul) înainte de aplicarea funcției hash - pentru a scădea posibilitatea obținerii parolei originale - chiar și în cazul în care varianta hash este compromisă.
2. Când se face verificarea corectitudinii parolei se verifică, desigur, corespondența hash-ului din baza de date nu cu parola introdusă de user, ci un *hash* construit din ea după aceeași regulă.

Notă: Ca o convenție pentru acest laborator, toate hash-urile vor fi construite folosind funcția md5(), după următoarea regulă: *parola_utilizator*|lab6|*nume_utilizator*.

Autentificare

Pentru a scrie efectiv autentificarea avem nevoie să creăm o serie de lucrui. În primul rând, trebuie să avem un *formular de autentificare.* Primul nostru formular de autentificare conține două câmpuri (user și parolă) și un buton - aveți un template pentru formularul de autentificare în *html/auth_form.html*.

Al doilea lucru de care avem nevoie este să avem o metodă de a verifica dacă utilizatorul și parola introduse sunt corecte. Pentru aceasta, vom completa metoda statică autentificare() a clasei Utilizator cu un query în baza de date. În funcție de numărul de rezultate întoarse de query, avem următoarele opțiuni:

1. 0 rezultate - utilizatorul nu există, întoarcem mesaj de eroare (există un template tag 'eroare' special pentru asta)
2. 1 rezultat - utilizatorul există - și trebuie să îl setăm ca autentificat în această sesiune. Cel mai simplu mod de a face asta este de a seta o variabilă session la True sau la numele utilizatorului, sau ambele (am putea face ceva de genul $_SESSION['user_is_authed'] = True; $_SESSION['username'] = $user->username;
3. 2 sau mai multe rezultate - avem o problemă în baza de date pe care trebuie să o rezolvăm.

După ce autentificarea reușește, vom redirecta utilizatorul către o pagină corespunzătoare nivelului lui de acces.

Task-uri

1. Scrieți metoda Utilizator::autentificare() - semnătura ei există, scrieți doar implementarea
2. Scrieți o acțiune de autentificare, înregistrați-o în dispecer, și implementați o acțiune în doi pași (pentru o cerere GET afișează formularul, pentru o cerere POST încearcă să facă autentificarea - iar dacă nu reușește, afișează iar formularul, cu o eroare, și cu câmpul username completat cu ce completase utilizatorul).
3. În caz de succes în succes al autentificării, setați o variabilă de sesiune cu numele utilizatorului, și afișați pagina pentru utilizatori autentificați. Pentru a putea să folosim sesiuni, trebuie să apelăm undeva metoda *session_start()*. Un loc bun pentru asta poate fi chiar partea de început a dispecerului.

Note:

1. În *php/util.php* există o funcție care calculează URL-ul unei anumite acțiuni folosind ca parametru doar numele înregistrat în dispecer al acțiunii. Pentru ca o asemenea metodă să funcționeze, a trebuit să refactorizăm puțin dispecerul.
2. Pentru a face o redirectare, putem folosi funcția header("Location: url"), urmată de un apel exit(). Ambele funcții trebuie apelate cu aceleași precauții ca și în cazul setcookie, și din aceleași motive. exit() se asigură ca script-ul PHP își încheie execuția și că conținutul (până aici, doar headere) va fi trimis către client.

Limitarea accesului la sistem pentru utilizatori autentificați

Limitarea accesului unui utilizator la un anumit script, sau, în cazul nostru, la o anumită acțiune, se face prin verificarea existenței variabilei de sesiune setate la autentificare. Pentru a avea un sistem flexibil, vom face acest lucru în clasa Utilizator.

Pentru a putea să facem ceva legat de utilizator în orice cerere următoare cererii în care am reușit autentificarea trebuie să obținem un obiect Utilizator - care să reprezinte utilizatorul autentificat. Pentru aceasta putem folosi factory method-ul get_by_name($username) al clasei Utilizator, care funcționează identic metodei get_by_id($id) din laboratorul trecut și întoarce un obiect Utilizator, sau null, în caz de eroare. Un loc pentru a face acest lucru este dispecerul, mai ales în cazul în care mare parte din aplicația noastră se bazează pe sesiuni autentificate.

Task

1. Implementați metoda este_autentificat() în clasa Utilizator, care întoarce True dacă Utilizatorul este autentificat (există chesia 'utilizator' în $_SESSION) și False în caz contrar.
2. Modificați acțiunea show_for_authenticated() pentru a afișa template-ul doar în cazul în care utilizatorul este autentificat, în caz contrar făcându-se redirect către acțiunea de

autentificare..

De-autentificare

Desigur, avem nevoie să implementăm și procesul invers autentificării - în care o sesiune de lucru se încheie. Acest lucru se poate face prin scrierea unei acțiuni care folosește funcția *unset*() pe variabila de sesiune setată la autentificare.

Task

1. Completați metoda deautentificare() a clasei Utilizator pentru a șterge variabila de sesiune setată la autentificare.
2. Implementați o acțiune nouă, în care obțineți un obiect Utilizator pentru utilizatorul autentificat în mod curent, apelați metoda deautentificare() și apoi faceți redirect - puteți face redirect fie către acțiunea de autentificare, fie către orice acțiune care necesită autentificare - care va face apoi la rândul ei redirect.

Diverite nivele de autentificare

Într-un sistem mai complex, s-ar putea să vrem să putem să dăm diferite permisiuni pentru diferiți utilizatori. Putem să spunem că fiecare utilizator are unul sau mai multe roluri, și ficărui rol îi sunt permise o serie de acțiuni (în limba engleză termenul folosit în literatură este *roles).*

Pentru a putea implementa roluri diferite, avem nevoie să știm două lucruri:

- Care utilizator ce rol are - vom vrea ca utilizatorii să poată avea mai multe roluri.
- Care acțiuni sunt permise căror roluri - în mod normal, acest lucru s-ar face folosind un tabel în baza de date, dar pentru a face testarea mai facilă noi vom folosi un array, în care rolurile sunt chei, și acțiunile permise sunt specificate cu numele (cuvântul cheie din dispecer) într-un array asociat cheii respective - un model pentru array se gaseste in Utilizator.class.php.

Avem nevoie să avem o metodă în Utilizator care să ne spună, pentru o anumită acțiune, daca un anumit utilizator are voie să realizeze. Pentru asta trebuie să modificăm ușor get_by_name() pentru a inițializa, din baza de date (din câmpul roluri) - rolurile utilizatorului în $utilizator->roluri, care să trebuie să fie un array.

În câmpul roluri din baza de date, acest array este serializat (sunt concatenate toate rolurile, cu virgulă între ele). Pentru a "sparge" string-ul din baza de date într-un array, putem folosi funcția explode().

Task-uri

1. Modificați metoda Utilizator::get_by_name() pentru a inițializa rolurile obiectului Utilizator cu un array cu rolurile din baza de date
2. Implementați metoda $utilizator->are_permisiunea($actiune) care întoarce True dacă utilizatorul are dreptul de a face o acțiune, și False în caz contrar.
3. Modificați acțiunile show_for_*() pentru a funcționa doar pentru grupurile cu numele respectiv, și metodele add_user, edit_user, list_user, delete_user pentru a fi accesibile doar utilizatorilor care au și rolul de *admin.*