

UNIVERSITATEA „ALEXANDRU IOAN CUZA” IAȘI

Facultatea de Informatică



LUCRARE DE LICENȚĂ

Quizdom

Propusă de

Robert –Ilie Vicol

Sesiunea: Iulie, 2018

Coordonator Științific:

Lector, dr. Cosmin Vârlan

UNIVERSITATEA „ALEXANDRU IOAN CUZA” DIN IAȘI

Facultatea de Informatică

Quizdom

Vicol Robert –Ilie

Sesiunea: Iulie, 2018

Coordonator Științific:

Lector, dr. Cosmin Vârlan

**DECLARAȚIE PRIVIND ORIGINALITATE ȘI RESPECTAREA
DREPTURILOR DE AUTOR**

Prin prezenta declar că Lucrarea de licență cu titlul „Quizdom” este scrisă de mine și nu a mai fost prezentată niciodată la o altă facultate sau instituție de învățământ superior din țară sau străinătate. De asemenea, declar că toate sursele utilizate, inclusiv cele preluate de pe Internet, sunt indicate în lucrare, cu respectarea regulilor de evitare a plagiatului:

- toate fragmentele de text reproduse exact, chiar și în traducere proprie din altă limbă, sunt scrise între ghilimele și dețin referința precisă a sursei;
- reformularea în cuvinte proprii a textelor scrise de către alți autori deține referința precisă;
- codul sursă, imagini etc. preluate din proiecte open source sau alte surse sunt utilizate cu respectarea drepturilor de autor și dețin referințe precise;
- rezumarea ideilor altor autori precizează referința precisă la textul original.

Iași,

Absolvent Vicol Robert -Ilie

(semnătura în original)

DECLARAȚIE DE CONSIMȚĂMÂNT

Prin prezenta declar că sunt de acord ca Lucrarea de licență cu titlul „Quizdom”, codul sursă al programelor și celelalte conținuturi (grafice, multimedia, date de test etc.) care însoțesc această lucrare să fie utilizate în cadrul Facultății de Informatică.

De asemenea, sunt de acord ca Facultatea de Informatică de la Universitatea „Alexandru Ioan Cuza” Iași să utilizeze, modifice, reproducă și să distribuie în scopuri necomerciale programele-calculator, format executabil și sursă, realizate de mine în cadrul prezentei lucrări de licență.

Iași,

Absolvent Vicol Robert -Ilie

(semnătura în original)

Cuprins

Introducere	6
Context.....	6
Cerințele aplicației	7
Descrierea soluției.....	8
Structura Lucrării	9
Contribuții	10
1. Dezvoltarea aplicației web	12
1.1 Particularități legate de server	12
1.2 Persistența datelor din aplicație	12
1.3 Stocare date temporare – Server Cache	16
1.4 Notificări externe aplicației	18
1.5 Notificări interne aplicației. Reactivitate	20
1.6 Particularități legate de client	24
1.7 Implementarea sincronizării în Quizdom	28
1.7.1 Cerere / Creare / Editare / Ștergere date	29
1.7.2 Sistemul de notificări extern	29
1.7.3 Sistemul de notificări intern	31
1.7.4 Procesarea informațiilor din timpul jocului	32
2. Prezentarea Aplicației	35
2.1 Arhitectura aplicației	35
2.2 Funcționalitățile aplicației	39
2.2.1 Funcționalități comune	39
2.2.2 Funcționalități specifice	40
3. Optimizări și dificultăți întâmpinate	43
3.1 JavaScript vs PHP Front-End	43
3.2 Comunicare de la server la client.....	44
3.3 Afișarea testelor grilă	45
3.4 Redis vs File Caching	46
3.5 Funcționalități pentru viitor	47
3.5.1 Pornirea rapidă a jocului	47
3.5.2 Joc între mai mulți utilizatori	48
Concluzii	49
Bibliografie	50
Anexa 1	51

Introducere

Această lucrare are scopul de a prezenta etapele dezvoltării aplicației web „Quizdom”. Rolul principal al acesteia este de a facilita acumularea de cunoștințe tehnice de către studenți, în vederea obținerii unor rezultate mai bune la materiile studiate la Facultatea de Informatică Iași.

Am ales să realizez această aplicație întrucât consider că pot oferi o modalitate ușoară de a înțelege noțiuni abstracte și de a verifica nivelul de cunoștințe acumulat de utilizator, fiind cunoscut faptul că învățarea prin joc este foarte eficientă.

Context

Pe parcursul celor trei ani de studiu, am remarcat faptul că multe dintre materiile predate conțin metode asemănătoare de testare a cunoștințelor acumulate: examene cu cerințe tip grilă, în care sunt solicitate, după caz, explicații adiționale ce demonstrează înțelegerea conceptelor verificate. Drept exemple, putem aminti discipline precum Structuri de Date, Programare Orientată Obiect, Baze de date, Ingineria Programării și Programare Avansată. Pentru a se pregăti mai bine, în afara parcurgerii materiei și a lecțiilor, studenții căutau materiale adiționale, precum subiecte sau cerințe formulate în anii trecuți la aceleași materii. Această nevoie, de a avea un model, un șablon, constituie una din situațiile care m-au îndemnat să caut o soluție.

Există, în prezent, aplicații asemănătoare, care ajută utilizatorii să capete fie cunoștințe generale, fie cunoștințe din domenii tehnice. Dintre acestea amintim:

1) Aplicația din cadrul disciplinei Baze de Date – folosită de studenți. Utilizatorii pot învăța sintaxa operațiilor SQL pe tabele (*select*, *insert*, *update*, *delete*). Aceasta este exclusivă disciplinei Baze de Date și nu are rolul și funcționalitățile unui joc.

2) QuizUp¹ – joc trivia pentru Android / iOS. Este cea mai populară aplicație de acest tip, având zeci de milioane de utilizatori, însă categoriile din care sunt întrebările nu vizează domenii tehnice. Mai mult decât atât, aplicația este doar pentru mobil, fiind necesară așadar instalarea acesteia.

¹ <https://www.quizup.com>

Spre deosebire de aceste aplicații, soluția propusă suportă orice platformă, poate fi extinsă ușor, pentru orice materie / domeniu tehnic și adaugă funcționalități noi care îmbunătățesc experiența utilizatorului.

Cerințele aplicației

Crearea și administrarea unui cont – necesar pentru a accesa lista întrebărilor „cumpărate”, precum și pentru salvarea unor statistici.

Accesul la informații despre alți utilizatori – aplicația salvează date publice despre fiecare utilizator (poziția în clasament, numărul de jocuri câștigate / pierdute, domeniile de interes).

Accesul la setul de întrebări existent – totodată cu revederea întrebărilor „cumpărate”, există posibilitatea de a contribui la lista de probleme, orice utilizator având șansa de a genera conținut nou, în cadrul aplicației.

Raportarea unei probleme – pentru cazul în care domeniul abordat de grila respectivă nu este ales corect, sau pentru formulări, răspunsuri greșite

Accesul la lista de utilizatori activi – cei autentificați pot vizualiza lista utilizatorilor care sunt activi², pentru a-i provoca la joc.

Invitația la joc – aplicația permite utilizatorilor să își trimită unii altora provocări la joc în timp real. Dacă oponentul acceptă invitația, jocul între doi utilizatori va începe.

Jocul propriu-zis – pentru un singur utilizator sau între 2 utilizatori. Reprezintă partea de bază a aplicației. Deciziile luate (în cazul jocului între 2 utilizatori) sunt văzute în timp real de către oponent.

Premierea câștigătorilor – este realizată prin atribuirea de token-uri, care pot fi folosiți în joc pentru acumularea de probleme sau propunerea acestora. De asemenea utilizatorii cu numărul cel mai mare de victorii se află în fruntea unui clasament public.

² Utilizatorii care au aplicația deschisă.

Sistem de notificări – utilizatorii autentificați beneficiază de un sistem de notificări, prezent atât în interiorul, cât și în exteriorul aplicației. Notificările sunt afișate în timp real și sunt declanșate de anumite acțiuni³ ale utilizatorului.

Păstrarea stării aplicației – reprezintă una dintre cele mai importante funcționalități ale aplicației. Permite reluarea activității⁴ în orice moment, chiar și în timpul jocului (pentru cazul în care se pierde conexiunea temporar).

Descrierea soluției

Aplicația se constituie din două părți:

1. REST API la nivel de server, cu o interfață minimalistă ce permite unui administrator să modifice resurse⁵.
2. Aplicație Vue, unde sunt disponibile funcționalitățile principale pe partea de client.

Implementarea pornește de la un set de întrebări de tip grilă, stocate într-o bază de date de tip **MySQL**. Utilizatorii logați pot folosi token-uri pentru a “cumpăra” întrebări; în urma acestui proces, întrebările respective vor deveni disponibile pentru răspuns. În urma acumulării unui număr de răspunsuri corecte, utilizatorii pot să contribuie la setul de întrebări existent. Există mai multe moduri prin care utilizatorii pot obține token-uri, dintre care amintim: răspunsuri corecte la întrebări, ”contribuția” la setul de întrebări existent, “cumpărarea” întrebărilor proprii de către alți utilizatori.

Aplicația oferă, de asemenea, funcționalitatea unui joc multiplayer de tip quiz, în care câștigătorul este cel care acumulează un număr mai mare de puncte, răspunzând corect la un număr de întrebări în timp cât mai scurt. Miza jocului este stabilită anterior de comun acord (un număr fix de token-uri).

Printre tehnologiile folosite amintim framework-ul **Laravel** pe partea de server, respectiv **VueJS** pe partea de client. Comunicarea este realizată prin protocolul HTTP, aderând la modelul arhitectural REST.

³ Invitația la joc adresată oponentului, respectiv acceptarea provocării de către oponent.

⁴ Orice informație este stocată temporar

⁵ Întrebări eronate, probleme raportate sau cu răspunsuri greșite, probleme raportate pe nedrept, etc.

În cadrul componentei de joc a aplicației, informațiile se transmit în timp real prin intermediul modelului Publish/Subscribe. Este utilizată, în acest sens, librăria **Pusher Channels**, care folosește Websocketi.

Partea internă a sistemului de notificări e construită pe același model Publish / Subscribe amintit, în timp ce partea externă are la bază serviciul Push Notifications, apelat prin API-ul de la OneSignal de pe partea de server.

Structura Lucrării

În continuare voi menționa contribuțiile mele în realizarea proiectului, urmând ca restul paginilor să detalieze pașii dezvoltării aplicației web (cap 1), să prezinte un studiu mai detaliat asupra funcționalităților (cap 2), precum și unele optimizări posibile și dificultăți întâmpinate (cap 3)

Contribuții

Pe parcursul dezvoltării acestei aplicații, contribuțiile mele au fost, pe de o parte teoretice și pe de altă parte, practice:

- Studiu pentru găsirea unei soluții optime pentru transmiterea de date și comunicarea în timp real (necesare jocului între doi utilizatori).
- Dezvoltarea unei modalități de a asigura persistența datelor în cazul pierderii conexiunii.
- Dezvoltarea unui sistem de notificări care să asigure primirea informațiilor chiar și în cazul în care utilizatorul are aplicația închisă.
- Implementarea aplicației pe parte de client, unde s-a încercat construirea unei interfețe intuitive, compatibile cu majoritatea ecranelor din punct de vedere al dimensiunii, care să permită accesarea tuturor funcționalităților printr-un număr cât mai mic de click-uri / atingeri.
- Implementarea aplicației pe parte de server, unde a fost necesară aprofundarea modelului MVC și găsirea unei metode prin care inițierea comunicării dintre client și server să fie făcută de cel din urmă.
- Testarea aplicației și a funcționalităților.
- Optimizarea aplicației, din punct de vedere al timpului de răspuns al serverului, precum și al tipului de stocare folosit.

Totodată, poate fi considerată contribuție publicarea unor componente de front-end în managerul de pachete **npm**. Vue fiind un framework pentru front-end care încurajează în mod special organizarea codului astfel încât bucăți din acesta să fie reutilizabile (componente), oricine are sistemul de dependențe cerut în proiectul său, poate importa și folosi componentele publicate de mine. Printre acestea, se numără următoarele:

- `vueAnimatedBar` : bară de lungime / lățime variabilă, ce simbolizează prin modificarea dimensiunii, scurgerea timpului.
- `vueTimer` : componentă cu logica necesară unui cronometru sau a unei numărătoare inverse.
- `vueCube` : componentă de meniu cu aspectul unui cub, ale cărui fețe sunt navigabile.

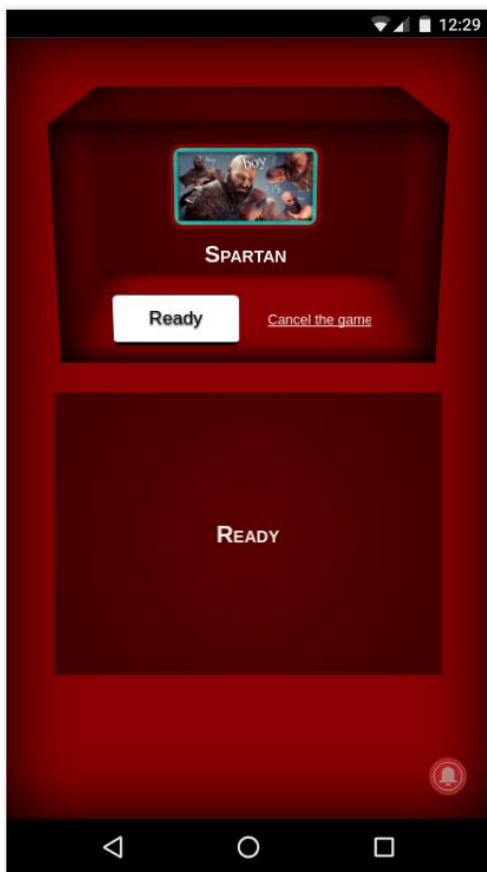


Figura 1 – exemplu de componentă vueCube

Se observă, în Figura 1, prezența a două componente vueCube, fiecare având conținut diferit.

Toate fețele cubului pot avea conținut HTML, acesta fiind vizibil doar prin rotirea cubului astfel încât suprafața dorită să fie în fața ecranului.

Cubul se va redimensiona în funcție de conținutul pe care îl are, însă este configurat astfel încât să nu poată întrece un anumit procentaj din lățimea ecranului. (de exemplu, prin css: *max-width:90vw*).

Astfel, se asigură o oarecare capacitate de adaptare a componentei, în funcție de natura ecranului și dimensiunea acestuia.

Acestea dovedesc aportul meu pentru comunitatea de front-end din jurul framework-ului **Vue.js**.

Trebuie de asemenea, să menționez că în realizarea aplicației, am folosit și eu, două componente dezvoltate în regim open-source. Acestea sunt:

1. *Sweet-Modal*⁶, o componentă ce conține un modal care poate fi activat oricând prin intermediul unei funcții JavaScript.
2. *Vue-Carousel*⁷, o componentă ce conține un element de tip carusel.

Ambele componente utilizate au fost personalizate, în acestea regăsindu-se conținut pur HTML propriu, cu rolul de afișare a informațiilor – *Vue-Carousel*, sau de preluare a inputului utilizatorului – *Sweet-Modal*.

⁶ <https://github.com/adeptoas/sweet-modal-vue>

⁷ <https://github.com/wlada/vue-carousel-3d>

1. Dezvoltarea aplicației web

În acest capitol voi detalia modalitățile prin care am folosit tehnologiile care au fost necesare în realizarea proiectului, precizând în același timp motivele pentru care am optat pentru aceste tehnologii.

1.1 Particularități legate de server

Pentru dezvoltarea funcționalităților cerute pe parte de server, am ales să folosesc framework-ul Laravel, împreună cu ecosistemul său, respectând modelul arhitectural MVC și aderând la principiul separării grijiilor. (Separation of Concerns). O descriere mai completă a acestuia se află în Anexa 1, care are rolul de a familiariza cititorul cu structura unui proiect de tip Laravel și de a scoate în evidență avantajele dezvoltării folosind acest framework.

Metodele unui **controller** returnează în general, un **view**, care este completat cu informațiile procesate. Fișierele de tip **view** au terminația *blade.php*, sugerând existența șabloanelor de tip **Blade** în componența paginilor web. Aceste șabloane permit utilizarea datelor returnate odată cu **view-ul** respectiv. Șablonul **Blade** permite, de asemenea, dezvoltatorului, să scrie cod php în interiorul paginilor web într-un mod organizat și propune o soluție pentru extinderea paginilor web.

Spre exemplu, toate paginile pot moșteni un șablon de tip **Blade** ce reprezintă un meniu de navigare. Astfel, prin cuvinte cheie precum *@section* sau *@yield*, este încurajată re folosirea codului într-un mod sugestiv.

Cu toate acestea, întrucât aplicația dezvoltată pe parte de server constituie doar un API, metodele din controllere nu vor returna view-uri, ci răspunsuri de tip JSON (Javascript Object Notation), care sunt interpretate în aplicația client de limbajul javascript. Paginile propriu-zise sunt realizate cu ajutorul framework-ului **Vue.JS**, care va fi menționat în subcapitolul 1.6.

1.2 Persistența datelor din aplicație

Stocarea datelor reprezintă un aspect important al oricărei aplicații web. Fără aceasta nu aș fi reușit să rețin informații despre utilizatori și statistici despre numărul de meciuri jucate sau

numărul de întrebări contribuite la setul existent. Pentru acest tip de informație este utilă o bază de date relațională. După cum am văzut în Figura 30, un avantaj al folosirii framework-ului Laravel este acela că suportă mai multe metode de a stoca date: **PostgreSQL**, **MongoDB**, **MySQL**, **Redis**, **File**. Dintre acestea, am ales să folosesc **MySQL**, fiind un proiect dezvoltat în regim open-source, popular și folosit de multe companii (Facebook, Twitter, Youtube, Spotify, Netflix) tocmai datorită faptului că oferă performanță și stabilitate.

De asemenea, **MySQL** face parte deja din structura **LAMP**, alături de componentele **Linux**, **PHP** și **Apache**, ceea ce înlătură dificultățile procesului de integrare.

Avem opțiunea de a realiza configurarea bazei de date din fișierul `.env`. Printre setări amintim:

- `DB_CONNECTION` (tipul bazei de date, în acest caz, așa cum am menționat, MySQL)
- `DB_HOST` (adresa la care este disponibilă baza de date)
- `DB_PORT` (portul deschis conectării)
- `DB_DATABASE` (numele dat bazei de date)
- `DB_USERNAME`
- `DB_PASSWORD`

Manipularea datelor este realizată prin intermediul ORM-ului specific Laravel, denumit **Eloquent**, acesta fiind asemănător cu alte ORM-uri populare, precum **Entity Framework** (asp.net). Ambele elimină necesitatea de a scrie interogările direct către baza de date și la fel de important este faptul că ambele aderă la principiul "code-first": tabelele sunt create prin migrări, clasele respective având două metode (up, down), iar legăturile dintre tabele sunt definite pe baza modelelor. În continuare precizăm tipurile de legături dintre tabele și echivalentul lor în Eloquent:

- unu-la-unu: metoda *hasOne*, cu inversa *belongsTo*.
- unu-la-mai-mulți: metoda *hasMany*, cu inversa *belongsTo*.
- mai-mulți-la-mai-mulți: metoda *belongsToMany*.

Un dezavantaj al ORM-ului **Eloquent** față de **Entity Framework** ar fi acela că nu poate genera fișierele de migrare automat. Neavând opțiunea de a declara tipul de dată în PHP, migrările nu au de unde să știe tipul unui atribut doar din structura modelului.

În cazul acestei aplicații au fost necesare următoarele tabele, respectiv câmpuri:

- ❖ **Users**: conține informații legate de utilizatorii aplicației.
 - *id*: constituie cheie primară.
 - *name*: numele utilizatorului.
 - *tokens*: monedă ce poate fi folosită în aplicație.
- ❖ **Answers**: cuprinde date legate strict de răspunsurile grilă.
 - *id*: constituie cheie primară, deci acest câmp este unic și nu poate fi nul. În plus, se va auto-incrementa odată cu adăugarea unui nou răspuns.
 - *text*: reprezintă enunțul răspunsului, nu poate fi nul.
- ❖ **Questions**: cuprinde date legate strict de întrebări.
 - *id*: constituie cheie primară, deci acest câmp este unic și nu poate fi nul. În plus, se va auto-incrementa odată cu adăugarea unei noi întrebări.
 - *text*: reprezintă textul întrebării, nu poate fi nul.
- ❖ **Categories**: conține date legate despre domeniile din care pot face parte problemele.
 - *id*: constituie cheie primară, deci acest câmp este unic și nu poate fi nul. În plus, se va auto-incrementa odată cu adăugarea unei noi categorii.
 - *name*: numele categoriei.
- ❖ **Problems**: reprezintă entitatea principală din aplicație.
 - *id*: constituie cheie primară.
 - *user_id*: cheie străină ce referențiază tabela **Users**. Constituie id-ul utilizatorului care a creat problema.
 - *category_id*: cheie străină ce referențiază tabela **Categories**. Constituie id-ul categoriei din care face parte problema.
 - *question_id*: cheie străină ce referențiază tabela **Questions**. Constituie id-ul întrebării corespunzătoare.
 - *answer_id*: cheie străină ce referențiază tabela **Answers**. Constituie id-ul răspunsului corect. Nu poate fi nul (o grilă trebuie să aibă un răspuns corect).
 - *bad1_id*: cheie străină ce referențiază tabela **Answers**. Constituie id-ul primului răspuns greșit. Nu poate fi nul (o grilă trebuie să aibă cel puțin un răspuns greșit).
 - *bad2_id*: cheie străină ce referențiază tabela **Answers**. Constituie id-ul primului răspuns greșit. Poate fi nul.
 - *bad3_id*: cheie străină ce referențiază tabela **Answers**. Constituie id-ul primului răspuns greșit. Poate fi nul.

❖ **Matches**: tabelă ce conține date despre jocurile între doi utilizatori. Poate fi considerat un istoric al meciurilor terminate.

- *id*: constituie cheie primară, deci acest câmp este unic și nu poate fi nul. În plus, se va auto-incrementa odată cu adăugarea unei noi categorii.
- *user1_id*: cheie străină ce referențiază tabela **Users**. Constituie id-ul utilizatorului care participă la joc.
- *user2_id*: cheie străină ce referențiază tabela **Users**. Constituie id-ul celui de-al doilea utilizator care participă la joc.

❖ **User-Problems**: tabelă pivot necesar legăturii mai-mulți-la-mai-mulți dintre utilizatori și probleme (în sensul în care o problemă poate fi cumpărată de mai mulți utilizatori, iar un utilizator poate să aibă mai multe probleme cumpărate).

❖ **Match-Problems**: tabelă pivot necesar legăturii mai-mulți-la-mai-mulți dintre jocuri și probleme (în sensul în care o problemă să se regăsească în mai multe jocuri, iar un joc poate să conțină mai multe probleme).

În plus față de informațiile menționate, fiecare tabelă conține două câmpuri adăugate și editate de **Eloquent**: *created_at* și *updated_at*. Acestea se actualizează automat atunci când vom crea o nouă instanță a modelului corespunzător tabelului, sau atunci când vom modifica proprietățile unui obiect deja existent, apelând metoda *save()*.

Mai există, de asemenea, o tabelă ce stochează informații despre fiecare migrare efectuată, precum data efectuării migrării, poziția pachetului din care aceasta face parte în ierarhie și numele fișierului propriu-zis (în care se află clasa migrării). Această tabelă este generată automat și este populată de fiecare dată când sunt efectuate migrări noi. Mai mult decât atât, aceasta contribuie la verificarea realizată de framework înaintea începerii procesului de migrare, verificare ce are rolul de a exclude migrările care au fost deja făcute, în trecut.

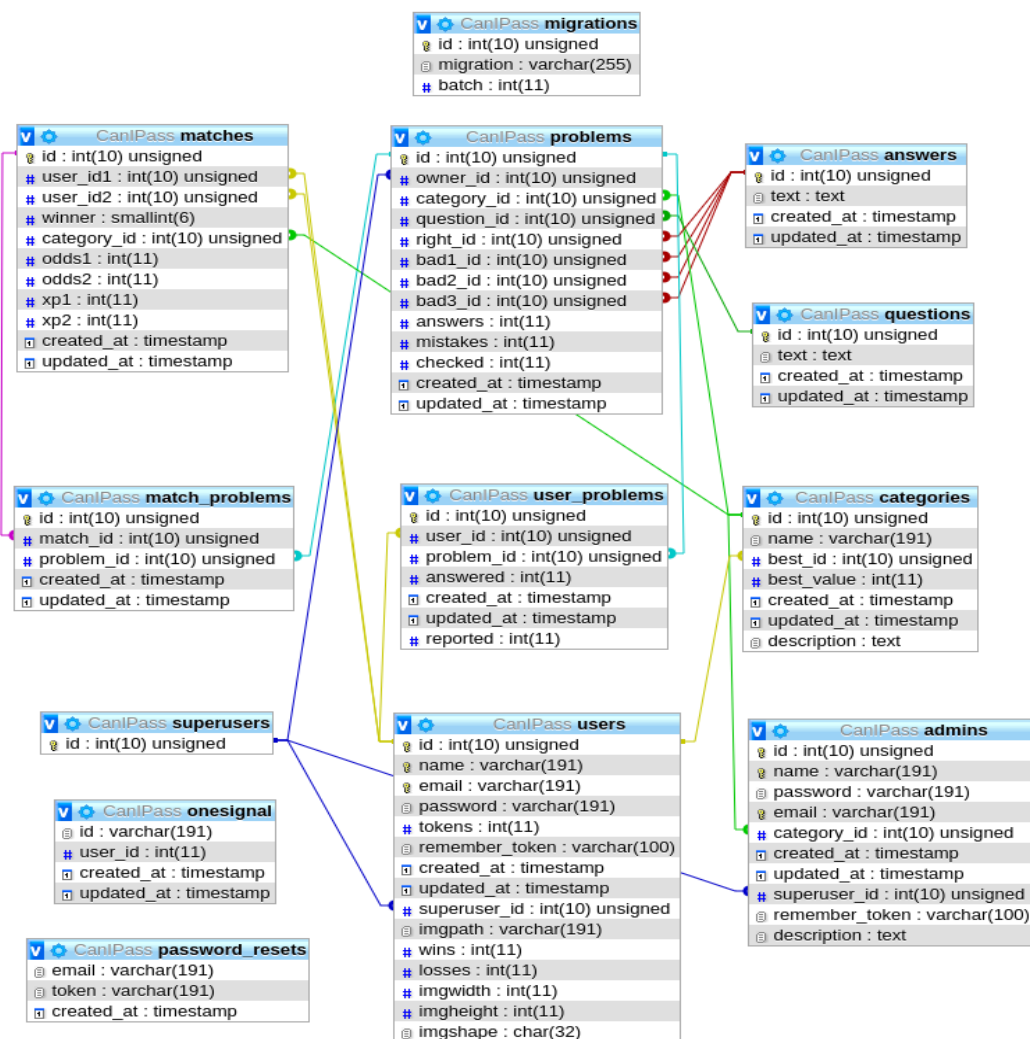


Figura 2 – Structura bazei de date evidențiată vizual

1.3 Stocare date temporare – Server Cache

În cazul proiectului descris, există situații în care salvarea informațiilor în baza de date relațională nu este o idee bună. Un exemplu concret este reprezentat de perioada în care doi utilizatori se află în joc. Fiecare răspuns dat de oricare dintre aceștia trebuie să ajungă la server, pentru a fi primit unul de la altul. Jocul arată în timp real deciziile adversarului, fie că acestea au fost corecte sau greșite. Cu alte cuvinte, informațiile suferă foarte multe actualizări într-un timp scurt, ceea ce ar însemna multe accesări spre baza de date relațională, pentru editări și inserări. Mai mult decât atât, în istoricul jocurilor păstrăm doar informații relevante precum ce probleme au picat, cine a câștigat jocul sau dacă acesta a fost încheiat cu remiză. Nu este necesară stocarea permanentă a datelor vehiculate în timpul jocului (cât timp mai are un

utilizator să răspundă, câte puncte are un utilizator în prezent sau la ce întrebare se află cei doi jucători)

Laravel propune o soluție pentru această problemă, având un serviciu de cache, util pentru stocarea temporară a informațiilor ce suferă multe actualizări într-un timp scurt. Acesta salvează datele sub forma unor perechi cheie-valoare, pentru un timp specificat. Nu avem nevoie de stocarea permanentă a acestor date, întrucât ele sunt relevante doar pe parcursul meciului dintre doi utilizatori. Datele respective pot fi șterse fie în cazul în care jocul s-a terminat, fie în cazul în care a trecut prea mult timp pentru ca jocul să poată fi reluat. (spre exemplu, dacă unul dintre jucători pierde conexiunea și nu revine în joc mai mult de 5 minute).

Serviciul cache este, de asemenea, folosit pentru stocarea (permanentă, de această dată) informațiilor primite de la un middleware personalizat. Am construit acest middleware astfel încât să preceadă toate cererile HTTP venite de la client. Middleware-ul respectiv nu are rolul de a filtra cererile, ci de a înregistra data la care s-a făcut cererea. Salvând mereu în cache această dată, am căpătat pe partea de client funcționalitatea de a vedea, în timp real, ultima oră la care orice utilizator a fost activ în aplicație.

În continuare voi prezenta structura pe care am ales să o dau memoriei cache disponibile. Pentru fiecare utilizator este disponibilă o listă, identificată unic prin id-ul acestuia. Toate listele existente în cache conțin:

- O variabilă în care este păstrată data ultimei cereri făcute de utilizator către server.
- O listă de notificări față de care utilizatorul nu a acționat în niciun fel (Notificarea constituie un obiect ce conține date precum numele și id-ul utilizatorului de la care provine aceasta, sau data la care notificarea respectivă a fost trimisă)
- Un obiect ale cărui proprietăți sunt folosite pentru a stoca starea jocului, astfel încât în orice moment acesta să poate fi reluat. Astfel, asigurăm persistența datelor în cazul meciurilor dintre doi utilizatori. Un obiect de acest gen are următoarele proprietăți:
 - Id-ul utilizatorului autentificat.
 - Numele utilizatorului autentificat.
 - Scorul utilizatorului autentificat.
 - Id-ul oponentului.
 - Numele oponentului.
 - Scorul oponentului în prezent.

- Statusul jocului (început, terminat, pe cale să înceapă, în stadiul de pregătire a jucătorilor)
- Numărul problemei la care s-a ajuns, împreună cu o listă ce conține toate problemele care vor apărea pe parcursul jocului.
- Timpul de la care pornește numărătoarea inversă.

Este necesar, ca pe parcursul jocului, utilizatorul să aibă acces către aceste informații, dar odată ce statusul meciului se transformă în ”Terminat”, datele pot fi salvate în baza de date, întrucât nu vor mai suferi modificări. De asemenea, un meci terminat nu va mai putea fi accesat nici de pe partea clientului, deci aceste date pot fi șterse și din cache-ul serverului.

1.4 Notificări externe aplicației

Una dintre funcționalitățile principale ale aplicației constă în abilitatea de a transmite atenționări în cazul declanșării unor evenimente, precum o invitație la joc trimisă de un utilizator. Mai mult decât atât, atenționările respective trebuie să poată ajunge la client chiar și atunci când acesta are aplicația închisă. Framework-ul Laravel nu are această funcționalitate încorporată.

O soluție pentru această problemă a fost găsită prin integrarea serviciului **web-push** de la OneSignal. Acesta permite afișarea notificărilor în afara aplicației, cu condiția ca utilizatorul să își fi dat acordul înainte.

După cum se observă în Figura 3, în momentul abonării la acest serviciu, se creează un identificator unic pentru dispozitivul utilizatorului aplicației. Acesta este trimis către baza de date OneSignal, urmând ca de fiecare dată când dorim să trimitem o notificare unui utilizator, să folosim id-ul corespunzător dispozitivului său. Așadar, putem vedea că OneSignal este configurat astfel încât să nu știe de existența utilizatorilor, ci doar a dispozitivelor pe care este rulată aplicația.




SUBSCRIBED	LAST ACTIVE	FIRST SESSION	DEVICE	SESSIONS	COUNTRY	LANGUAGE CODE	PLAYER ID
✓	6/12/18, 2:40:12 pm	4/28/18, 2:24:51 pm	 Linux x86_64 (61)	66	RO	en	05f734c1-a4c1-44b5-82a3-0a6545b617a7
✓	6/10/18, 2:37:19 pm	4/30/18, 6:43:02 pm	 Linux x86_64 (60)	27	RO	en	f6c9510e-f79b-4135-bfda-f4a001e3d7a3
✓	4/30/18, 6:41:05 pm	4/30/18, 6:41:05 pm	 Linux x86_64 (59)	1	RO	en	3523bc08-cda8-4ed3-a929-610a5b02361d

Figura 3 – identificatorul unic (PLAYER_ID) și baza de date OneSignal.

Întrucât dorim să selectăm doar anumiți utilizatori în cadrul funcționalității de trimitere a notificărilor, avem nevoie de toate id-urile dispozitivelor din baza de date OneSignal. Astfel, de fiecare dată când este creat un identificator nou (atunci când un dispozitiv încarcă scriptul OneSignal pentru prima oară), acesta este trimis, împreună cu id-ul utilizatorului, pe partea de server. Aici vom stoca legătura dintre utilizator și dispozitivul pe care acesta îl posedă. Avem nevoie, așadar, de un tabel nou, prin care să asigurăm persistența acestor date. Înregistrările acestui tabel au, pe lângă cele două adăugate de **Eloquent** automat, încă două câmpuri: *user_id* și *player_id*.

Având aceste informații, ori de câte ori un utilizator este chemat la joc, serverul se va folosi de id-ul său pentru a găsi identificatoarele dispozitivelor de pe care acesta s-a abonat în trecut. Apelând, ulterior, la serviciul OneSignal, putem trimite câte o notificare la fiecare dispozitiv abonat.

Această metodă implică anumite riscuri. Dacă doi utilizatori împart același dispozitiv, există posibilitatea ca aceștia să primească notificări care nu le sunt destinate. Pentru a elimina această problemă, înaintea delogării de pe un dispozitiv, vom face o cerere DELETE către server, pentru a elimina legătura dintre utilizator și dispozitivul respectiv. În acest fel, suntem siguri că utilizatorii primesc notificări doar pe dispozitivele pe care sunt autentificați.

Pe parte de server, putem apela la API-ul OneSignal prin intermediul unei cereri cURL. Parametrii de care avem nevoie sunt reprezentați de identificatorii dispozitivelor spre care dorim să trimitem notificarea, precum și de un identificator al aplicației OneSignal și o cheie privată pentru apelul către API. Ultimii doi parametri sunt furnizați de OneSignal. În plus, putem seta, opțional, un obiect care să fie trimis, împreună cu notificarea, pe partea de client. În javascript, putem avea o acțiune declanșată în momentul în care primim acest obiect.

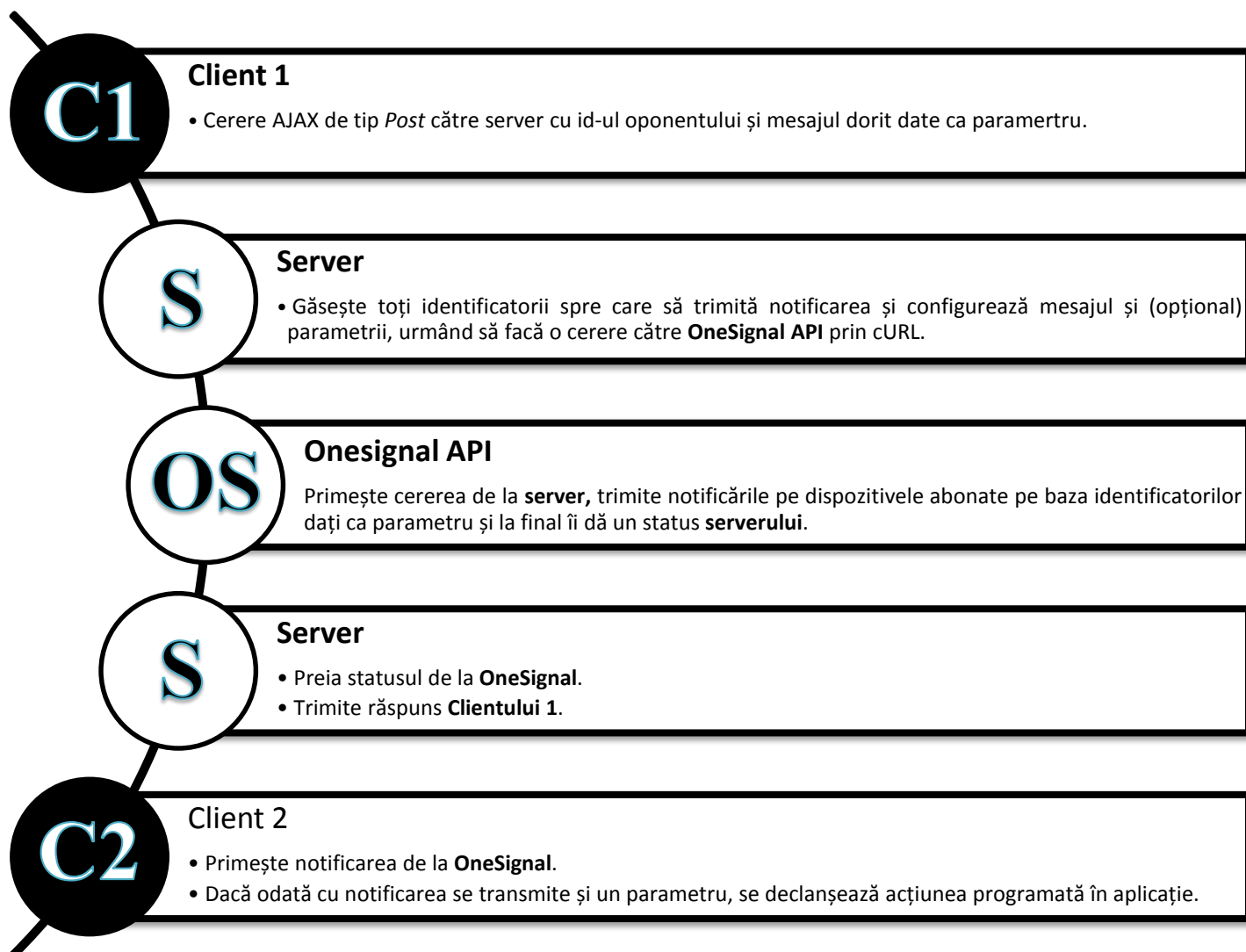


Figura 4 – Trimiterea notificărilor externe aplicației.

Figura 4 exemplifică modalitatea prin care serverul configurează și trimite notificările de tip extern.

1.5 Notificări interne aplicației. Reactivitate

După ce am evidențiat modalitatea prin care serverul trimite notificări externe, este normal să prezint soluția găsită privind implementarea sistemului de notificări intern. Avem nevoie de acesta, întrucât utilizatorii trebuie să poată fi anunțați de evenimentele care țin de ei și în mod direct, din aplicație.

Primele variante găsite sunt și cele mai intuitive – long-polling, short-polling – implicând realizarea fie a foarte multe cereri HTTP de pe parte de client, pentru a recepționa schimbările unor date de pe partea de server în timp real, fie a unor cereri care rămân

nerezolvate pe partea serverului până când acesta este pregătit să răspundă (când se dorește declanșarea notificării).

Variantele enumerate sunt folosite în dezvoltarea aplicațiilor web, însă acestea au anumite limitări și aspecte negative precum folosirea multor resurse, sau întârzierea răspunsului primit de client (spre exemplu, în cazul short-polling, mesajul nu este primit în timp real, ci depinde direct de frecvența cererilor HTTP făcute de către client).

Din aceste cauze, pentru implementarea notificărilor și mai ales a jocului, în cazul acestui proiect am considerat mai potrivită o soluție bazată pe **WebSockets**. Avantajele sunt semnificative: pe de o parte, nu sunt folosite la fel de multe resurse, comunicarea fiind bidirecțională și mai important, rapidă. Acest lucru este necesar în general, în dezvoltarea jocurilor între mai mulți utilizatori, întrucât acțiunea fiecărui jucător poate fi influențată de evenimentele puse în mișcare de alți jucători.

Așadar, cercetând opțiunile avute pentru o implementare bazată pe websocket, am descoperit **Pusher Channels**. Acesta furnizează un serviciu care înlesnește trimiterea mesajelor de la server la client, servind drept punte de legătură între acestea, după cum se observă în Figura 5. Serviciul lor este folosit inclusiv de Github, dovedind stabilitate și performanță.

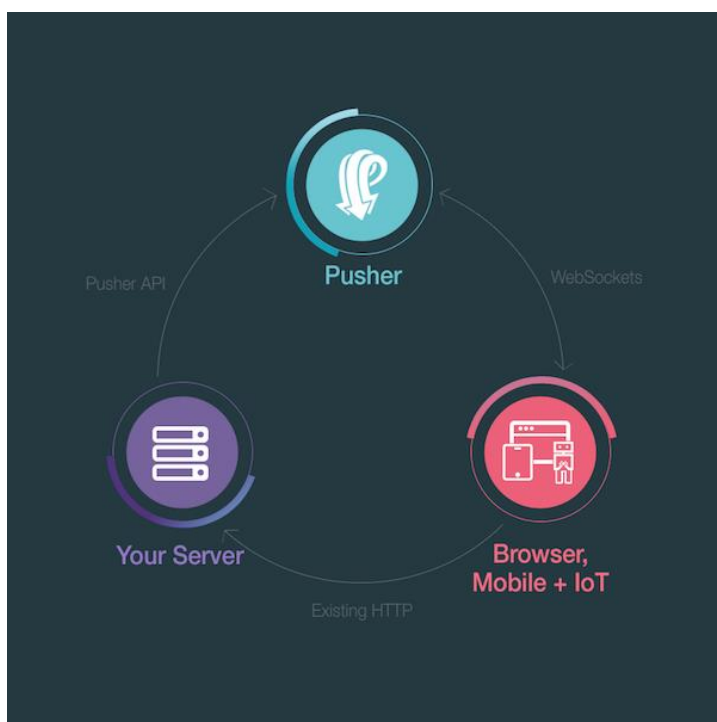


Figura 5 – serviciul Pusher (1)

Pusher oferă librării atât pe parte de client, cât și pe parte de server, scurtând timpul necesar dezvoltării aplicațiilor ce au la bază emiterea și primirea mesajelor în timp real.

Se observă, în imagine, cum serverul, de această dată, poate fi instanța care pornește comunicarea, clientul primind mesajul prin intermediul apelului către API-ul **Pusher**.

În plus, știind că nu orice navigator web suportă transmiterea mesajelor prin websocket, alegerea făcută este una bună, întrucât **Pusher** asigură trimiterea mesajelor chiar și în cazul în care tehnologia websocket nu este disponibilă, prin diferite alternative. Alte avantaje constau în asigurarea scalabilității, precum și în existența unei documentații solide, cu multe exemple și soluții pentru diverse limbaje de programare, printre care se regăsește și **PHP**. (2)

Pentru a folosi acest serviciu, avem nevoie să declarăm un obiect de tip Pusher pe parte de client, în javascript, după integrarea librăriei corespunzătoare.

```
var pusher = new Pusher('APP_KEY', {  
  cluster: 'APP_CLUSTER'  
});
```

Figura 6 – inițializare Pusher (3)

”APP_KEY” constituie cheia primită odată cu înregistrarea unei aplicații pe platforma **Pusher**.

```
var channel = pusher.subscribe('my-channel');
```

Figura 7 – Pusher – abonare la canal pe parte de client (3)

”my_channel” reprezintă numele atribuit unui canal de către noi pe parte de server, la care se va abona clientul.

```
channel.bind('my-event', function(data) {  
  alert('An event was triggered with message: ')  
});
```

Figura 8 – Pusher – ascultare la un canal pentru primirea mesajelor (3)

”my_event” este numele dat evenimentului nostru declarat pe partea de server, iar *alert* este acțiunea programată pe parte de client odată cu primirea unui mesaj.

Atunci când dorim să trimitem un mesaj unui client de pe server, declanșăm un eveniment, căruia îi atribuim un **nume**, un **canal** și un **pachet de date**.

- **Numele** este important, fiind unul dintre parametrii necesari conectării de pe partea de client.
- **Canalul** constituie entitatea către care se abonează clienții care vor să primească mesaje. În funcție de necesități, acesta poate fi public sau privat (accesibil doar unei porțiuni anume din utilizatori)
- **Pachetul de date** constituie mesajul trimis către utilizatorii abonați la canal.

Pe parte de server, după instalarea pachetului **Pusher** (prin **composer**) și configurarea

serviciului, este necesară doar declanșarea evenimentului, după cum se observă în Figura 9:

```
// First, run `composer require pusher/pusher-php-server`

require __DIR__ . '/vendor/autoload.php';

$pusher = new Pusher\Pusher("APP_KEY", "APP_SECRET", "APP_ID", array('cluster' => 'APP_CLUSTER'));

$pusher->trigger('my-channel', 'my-event', array('message' => 'hello world'));
```

Figura 9 – Pusher – declanșarea unui eveniment (3)

În cazul nostru, având opțiunea ca **Pusher** să fie integrat cu **Laravel**, configurația și codul diferă puțin față de exemplul precedent după cum urmează:

- După obținerea pachetului **Pusher** prin rularea comenzii de tip composer *"composer require pusher/pusher-php-server"*, putem completa în fișierul *config/broadcasting.php* id-ul aplicației **Pusher**, precum și cheia, respectiv secretul acesteia.
- În continuare, trebuie să adăugăm în vectorul *"providers"* linia *"App\Providers\BroadcastServiceProvider,"*. Vectorul se află în fișierul *"config/app.php"* (vezi Anexa 1). Aceasta ne permite să înregistrăm rute care să fie folosite pentru autorizarea utilizatorului care ascultă pe un canal privat. Autorizarea este făcută printr-o regulă definită în fișierul *"routes/channels.php"*. Regula respectivă acceptă două argumente, pe de o parte numele canalului securizat, pe de altă parte o funcție ce returnează adevărat sau fals. Valoarea returnată este folosită pentru a verifica dacă utilizatorul care încearcă să asculte un canal privat este autorizat. Această funcționalitate este necesară pentru a avea **siguranța transmiterii notificărilor interne doar către utilizatorii care sunt destinați să le primească**.
- Pentru a transmite evenimentul, este necesară definirea acestuia. Un eveniment reprezintă o clasă cu metode care simplifică procesul de transmitere a informațiilor pe parte de client. Pentru ca acesta să nu fie activ doar pe partea de server, este necesară implementarea interfeței *ShouldBroadcast*, care permite evenimentului să fie transmis pe un canal ascultat de client. Interfața respectivă cere implementarea unei metode *broadcastOn*, care va returna canalul pe care va fi transmis evenimentul respectiv. Orice proprietate **publică** a clasei definite va fi transmisă împreună cu evenimentul, către toți clienții care ascultă pe canalul returnat de funcția

broadcastOn (prin serializarea acestora, astfel încât informațiile să poată fi procesate pe parte de javascript). O alternativă care oferă mai mult control dezvoltatorului asupra datelor trimise este oferită prin implementarea metodei *broadcastWith*, care va returna un vector cu toate informațiile ce vor fi trimise.

- După definirea unui eveniment, transmiterea acestuia poate fi făcută prin comanda *event(new NumeEveniment(\$data))*, unde *\$data* constituie orice variabilă de care are nevoie constructorul clasei *NumeEveniment*.

În acest fel, serverul poate iniția comunicarea cu clientul, utilizatorii fiind capabili să vadă mișcările adversarilor în timp real, datorită interpretării pachetului de date primit în aplicația client. În plus, această configurație asigură transmiterea datelor doar către utilizatorii autorizați.

1.6 Particularități legate de client

Vue.js reprezintă un framework javascript de tip **progresiv**, folosit în construirea unor interfețe vizuale adaptive. A căpătat popularitate în rândul dezvoltatorilor web, fiind ușor de învățat și de integrat în orice parte de client a unui proiect web. De asemenea, structura sa prezintă unele similitudini cu alte framework-uri javascript precum Angular sau React, câteva dintre acestea fiind: existența unui DOM virtual, încurajarea dezvoltării interfeței web pe componente, existența unor proiecte adiționale care îndeplinesc funcționalitățile necesare dezvoltării unor aplicații web complete (mecanisme de rutare, de stocare a datelor). În plus, Vue beneficiază de o documentație stabilă, cu exemple care pun în evidența puterea acestui framework de a transforma sarcini dificile în unele simple, păstrând în același timp viteza și performanța.

Aceste caracteristici, împreună cu simplitatea și flexibilitatea pe care o oferă, m-au determinat să acord o șansă acestui framework. O altă trăsătură utilă, pe care am observat-o ulterior, constă în faptul că Vue ajută dezvoltatorul să mențină codul într-un mod organizat, fiecare componentă aflându-se într-un fișier separat, care este compus din 3 părți:

- Secțiunea *<template>*, unde se găsește structura html a componentei definite. Conținutul acesteia poate fi dinamic, având acces direct către proprietățile javascript declarate în secțiunea *<script>*.

- Secțiunea `<script>`, unde se află codul javascript, servind drept logică a componentei. Avem aici metode ajutătoare care determină comportamentul componentei în cazul schimbării datelor.
- Secțiunea `<style>`, opțională, servește la stilizarea modificarea aspectului pe care îl are componenta. Poate fi declarată astfel încât să nu afecteze decât stilul componentei din care face parte. (comportament *scoped*).

```

1  <template>
2    <div class="allbtn">
3      <button class="btn" v-for="cat in categories" @click="gotocreateForm(cat.id,cat.name)"
4        :disabled="cat.val == 1">{{cat.name}}</button>
5    </div>
6  </template>
7
8  <script>
9    import Vue from 'vue'
10
11    export default {
12      name: 'qmenu',
13      data () {
14        return {
15          categories:{},
16        }
17      },
18
19      methods: {
20        gotocreateForm(id,name){
21          window.localStorage.setItem('catid',id)
22          window.localStorage.setItem('catname',name)
23          this.$router.push({path: "/qform"})
24        },
25      },
26      mounted() {
27        Vue.http.get(this.$apiurl + 'caniadd').then((response) => {
28          this.categories = response.body.ret
29        })
30      }
31    }
32  </script>
33
34  <style>
35  </style>
36

```

Figura 10 – Exemplu componentă Vue.

În aplicația prezentată, fiecare componentă constituie una dintre paginile navigabile, cu excepția componentelor reutilizabile care acoperă porțiuni de pagină. În unele cazuri există nevoia ca între aceste componente să fie împărtășite informații. Spre exemplu, după ce utilizatorul trece prin procesul de autentificare, fiecare pagină următoare are nevoie de numele de utilizator și de alte preferințe ale acestuia pentru a crea o experiență vizual plăcută și personalizată. Un alt exemplu poate fi considerat nevoia de a avea informații despre persoana provocată, în momentul pornirii jocului.

Soluții temporare găsite au constat în salvarea datelor necesare în spațiul local de stocare

pe parte de client. (localStorage) sau în modificarea variabilelor comune prin emiterea unor evenimente care erau recepționate de toate componentele vizate. Cu toate acestea, odată cu scalarea proiectului, am observat faptul că aceste soluții nu sunt convenabile, având anumite defecte după cum urmează:

- Perechile de tip cheie-valoare din spațiul de stocare locală nu dispar în mod automat odată cu închiderea paginii, acestea fiind capabile să influențeze într-un mod negativ parcursul normal al aplicației. De exemplu, putem avea o variabilă care să numere câte probleme a rezolvat un utilizator de când a pornit aplicația. La delogare putem seta această variabilă 0, însă dacă utilizatorul închide aplicația fără a se deloga, variabila respectivă va rămâne stocată, astfel încât la următoarea accesare aceasta va conține o informație falsă.
- Emiterea de evenimente și recepționarea acestora este asincronă, iar în cazul modificării unei variabile, la testare, putem întâlni dificultăți în a găsi componenta care a cerut schimbarea valorii respective. În plus, datele comune componentelor se vor regăsi în fiecare dintre acestea, existând din această cauză multe duplicate.

Din aceste cauze, era necesară găsirea unui tip de stocare sincron, care să nu poată influența decât starea actuală a aplicației și la care să aibă acces orice componentă are nevoie de informația respectivă.

Sistemul de stocare *Vuex* îndeplinește aceste necesități și aduce, în plus, elemente comune altor modalități de stocare populare precum cea implementată de Facebook, *Flux*. Prezența **mutațiilor** și a **acțiunilor** constituie cea mai importantă dintre similitudinile menționate, acestea reprezentând o soluție sincronă de a stoca informațiile. Existența unei **surse universale de adevăr**, denumită în aplicația noastră *store*, este de asemenea importantă, aderând în continuare la modelul de stocare a datelor *Flux*.

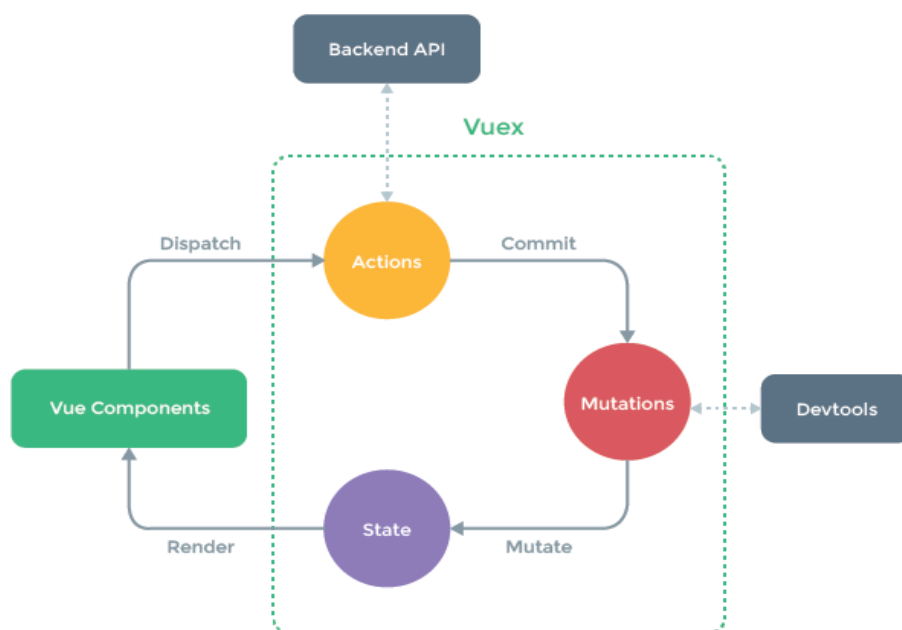


Figura 11 – Diagrama Vuex (<https://vuex.vuejs.org/vuex.png>)

Observăm, în Figura 11, diagrama acestui model de stocare, unde doar instanța de adevăr poate modifica valorile unei variabile din interiorul acesteia. Mutațiile sunt funcții ale instanței de adevăr care pot fi apelate, prin acțiuni, din fiecare componentă. Acestea au, în general, un argument care constituie noua valoare a variabilei ce urmează să fie modificată. Astfel, componentele nu pot modifica valorile variabilelor din *store*, fără să apeleze una dintre acțiunile definite de către acesta. În același timp, orice componentă poate asculta variabilele din sursa de adevăr pentru schimbări, reușind astfel, să facem posibilă schimbarea comportamentului unei componente, în mod indirect, din interiorul altei componente.

Figura 12 surprinde un exemplu concret al implementării sistemului *Vuex*:

```

7   export const store = new Vuex.Store({
8     state: {
9       token: null,
10      user:{},
11      menu_items:[],
12      category_id:0,
13      OneSignal:2,
14      device:""
15    },
16  },
17  mutations: {
18    STORE_TOKEN (state, token) {
19      state.token = token
20      window.localStorage.setItem('TOKEN', token)
21    },
22    STORE_USER(state, user) {
23      state.user = user
24    },
25    STORE_MENU_ITEMS(state, menu_items) {
26      state.menu_items = menu_items
27    },
28    STORE_CATEGORY_ID(state, category_id) {
29      state.category_id = category_id
30    },
31    STORE_OS(state, onesignal)
32    {
33      state.OneSignal = onesignal
34    },
35    STORE_DEVICE(state, device)
36    {
37      state.device = device
38    }
39  },
40  },

```

Figura 12 – Date și mutații în *store*.

În obiectul **state** se află informațiile stocate de *store*. Acestea sunt accesibile fiecărei componente în mod direct, doar pentru citire. Valorile din cod sunt cele pe care le au câmpurile la început.

Pentru scriere, avem în obiectul **mutations**, funcțiile ce trebuiesc apelate pentru modificarea valorii variabilelor de mai sus.

Fiecare variabilă poate avea un număr nelimitat de mutații, fiecare având un comportament diferit. Spre exemplu, Token-ul este salvat și în obiectul **state** și în spațiul local de stocare, pe când alte variabile nu sunt necesare decât în sesiunea curentă, fiind salvate doar în obiectul **state**. (precum identificatorii categoriilor din care fac parte problemele).

În codul afișat sunt prezentate, de asemenea, mutații pentru stocarea unor date necesare implementării sistemului de notificări extern aplicației.

La prima vedere, pare bizară decizia de a avea și acțiuni și mutații într-un sistem de stocare. Diferența dintre acestea este una subtilă. În primul rând, **acțiunile** țin de logica aplicației, pe când **mutațiile** țin de persistența stării aplicației (de informațiile stocate). Mai exact, **acțiunile** sunt declarate în *store* și folosite în componente de către dezvoltator pentru a realiza logica aplicației, în timp ce **mutațiile** sunt efectuate de **acțiuni** și țin cont doar de schimbarea valorilor din câmpurile stocate. În al doilea rând, acțiunile definite pentru a satisface cererile aplicației pot efectua mai multe mutații deodată, asupra mai multor câmpuri.

1.7 Implementarea sincronizării în Quizdom

După cum am stabilit deja, datorită cerințelor speciale pe care le are aplicația prezentată (comunicare în timp real bidirecțională, alertarea utilizatorilor prin notificări), nu era suficientă doar modalitatea comună de interacțiune (doar cereri HTTP). Astfel, comunicarea între cele

două entități se face prin mai multe metode, acestea fiind enumerate în continuare, în funcție de scenariul de utilizare de care aparțin.

1.7.1 Cerere / Creare / Editare / Ștergere date

Pentru operații de tip CRUD sunt folosite cereri HTTP obișnuite, care furnizează un cod numeric pentru statusul cererii și opțional, un răspuns cu date în format JSON returnat de o metodă ce aparține unui controller. Pentru aceste cereri este folosit clientul HTTP bazat pe promisiuni *Axios*. În cazul în care există, răspunsul este interpretat și afișat în paginile aplicației web. Aceste tipuri de cereri sunt folosite în situații precum autentificarea, înregistrarea, editarea datelor personale, verificarea problemelor create sau adăugarea unor probleme noi, aducând valoare consistentă aplicației, întrucât permit actualizarea paginilor fără reîncărcarea acestora. Răspunsul venit de la server nu schimbă decât o porțiune din componența paginii curente. Mai mult decât atât, un mare avantaj constă în faptul că cererile AJAX nu blochează aplicația, utilizatorul având posibilitatea să interacționeze cu aceasta în timp ce navigatorul așteaptă răspunsul de la server. (4)

1.7.2 Sistemul de notificări extern

În unele cazuri aplicația trebuie să alerteze utilizatorul asupra unor evenimente chiar și atunci când aceasta nu este pornită. Un exemplu concret reprezintă invitația la joc, care trebuie să ajungă la oponent cât mai rapid, pentru a reduce timpul așteptării dintre jocuri. Această funcționalitate este garantată prin serviciul *Push* de la *OneSignal*. Însă pentru a accesa acest serviciu, sunt necesari câțiva pași, dintre care cei mai relevanți sunt descriși în continuare:

- Pe parte de client:
 - ◆ includerea scriptului OneSignal.
 - ◆ configurarea id-ului aplicației (pentru ca serviciul să poată afla care notificări corespund aplicației noastre și ce dispozitive există deja în baza de date OneSignal).
 - ◆ memorarea identificatorului generat la prima accesare, după ce utilizatorul și-a exprimat acordul de a primi notificări.
- Pe parte de server:
 - ◆ crearea unei funcții care accesează api-ul OneSignal prin cURL, având argumente precum identificatorii dispozitivelor spre care să trimitem

notificarea, id-ul aplicației OneSignal, cheia REST privată, titlul și mesajul conținut de notificarea respectivă.

- ◆ Odată definită această funcție, o putem apela oricând dorim ca un utilizator să primească o notificare. În cazul aplicației noastre, este necesară trimiterea unei notificări atunci când un utilizator lansează o provocare la joc. Așadar, preluăm id-ul oponentului și găsim toate dispozitivele de pe care acesta s-a logat. O parte dintre acestea vor reprezenta destinația notificării.

Din acest moment, utilizatorul va primi orice notificare este adresată dispozitivului său, în bara de status sau în navigatorul web (cu condiția ca acesta să își fi dat acordul înainte). Utilizatorii se pot dezabona oricând de la serviciul de notificări printr-un buton vizibil permanent din aplicație. Accesarea acestuia scoate identificatorul dispozitivului din lista celor abonați. Pentru a explica mai bine acest proces, următoarea diagramă ilustrează detaliile prezentate anterior:

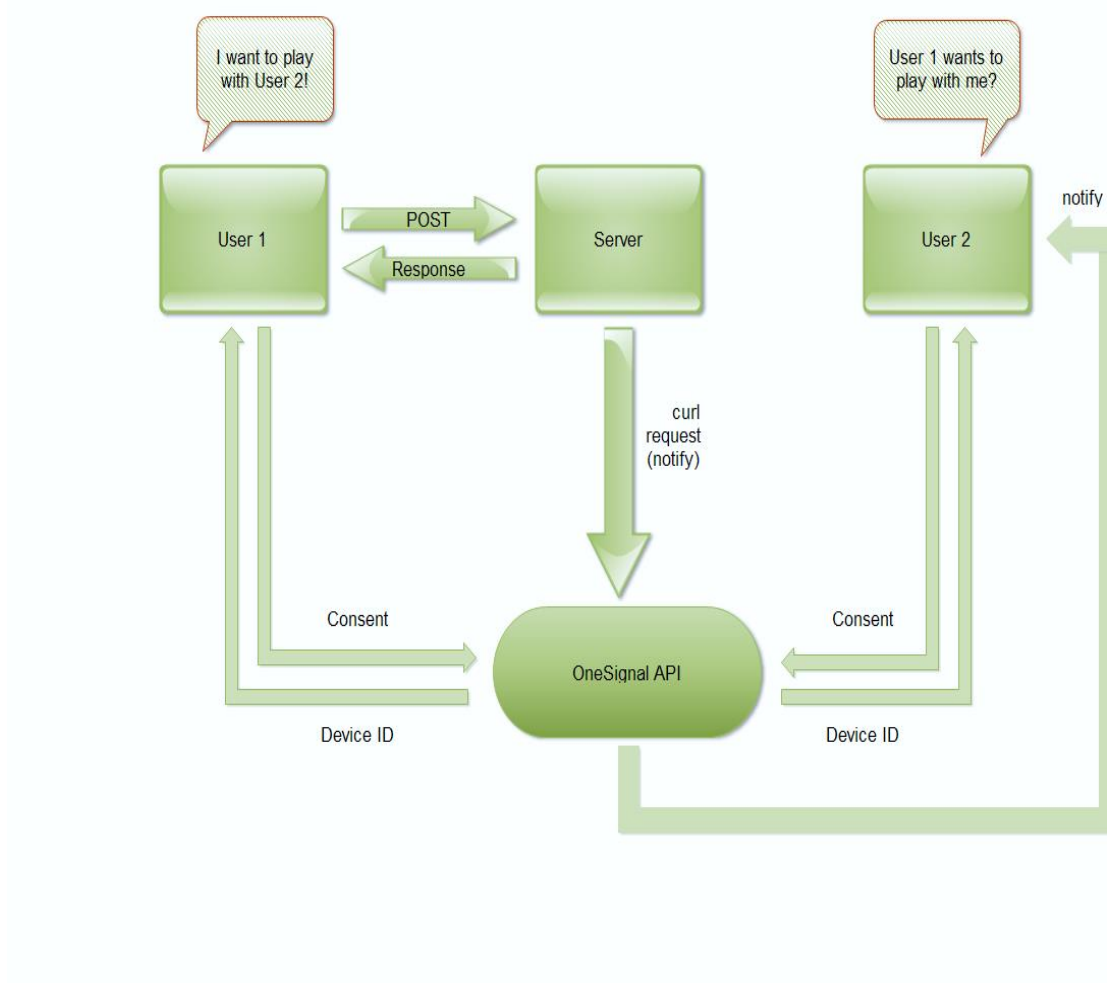


Figura 13 – Comunicarea dintre aplicație, server și OneSignal API

După cum se observă în Figura 13, în momentul deschiderii aplicației, după ce își vor exprima acordul, ambii utilizatori își găsesc identificatorul dispozitivului de pe care se conectează prin scriptul OneSignal. Dacă primul utilizator decide că vrea să-l provoace pe cel de-al doilea utilizator la un joc, se trimite o cerere HTTP de tip POST, prin axios, către server, iar acesta face o cerere de tip cURL către OneSignal API, după ce găsește dispozitivele pe care este autentificat cel de-al doilea utilizator. Cererea respectivă conține lista de identificatori corespunzători dispozitivelor respective, precum și mesajul dorit pentru a fi trimis (invitația la joc). Având lista de identificatori, OneSignal API va căuta în baza de date a aplicației (care are aceeași cheie ca cea trimisă ca parametru) identificatori comuni celor aflați în listă. În final, serviciul va trimite câte o notificare spre fiecare dispozitiv găsit.

Atunci când un utilizator apasă pe o notificare externă, acesta va fi redirecționat spre aplicație, dacă este deschisă, iar dacă nu, o va deschide, astfel încât utilizatorul să poată decide dacă vrea să accepte sau să respingă invitația la joc.

1.7.3 Sistemul de notificări intern

Persistența notificărilor interne aplicației este asigurată prin salvarea acestora pe server, în cache. Fiecărui utilizator îi corespunde o listă, cheia pentru găsirea acesteia fiind chiar identificatorul unic al utilizatorului.

Listele conțin informații despre identificatorul și numele celui care a lansat acțiunea ce a dus la transmiterea notificării, precum și despre timpul în care notificarea a fost trimisă. În plus, listele se formează atunci când este trimisă prima notificare și dispar atunci când nu mai există notificări, acestea fiind șterse odată ce sunt accesate de utilizator. Accesarea presupune a lua o decizie în privința jocului: **acceptare**, caz în care jocul dintre cei doi utilizatori va fi gata să înceapă, sau **refuz**, caz în care celălalt utilizator va fi alertat despre respingerea invitației și anularea jocului.

Această metodă prin care salvăm notificarea în cache ne ajută și în cazul în care utilizatorul nu are aplicația deschisă atunci când primește notificarea. Întrucât Laravel Echo nu va fi activ, în acest caz, pe parte de client, notificarea nu va ajunge în aplicația utilizatorului, ci doar pe canalul destinație. Cu toate acestea, având toate datele de care avem nevoie stocate în server, la deschiderea aplicației putem efectua o cerere http de tip GET care să verifice

notificările primite de respectivul utilizator. Practic, în răspunsul cererii vom găsi obiectul din cache care asigură persistența notificărilor.

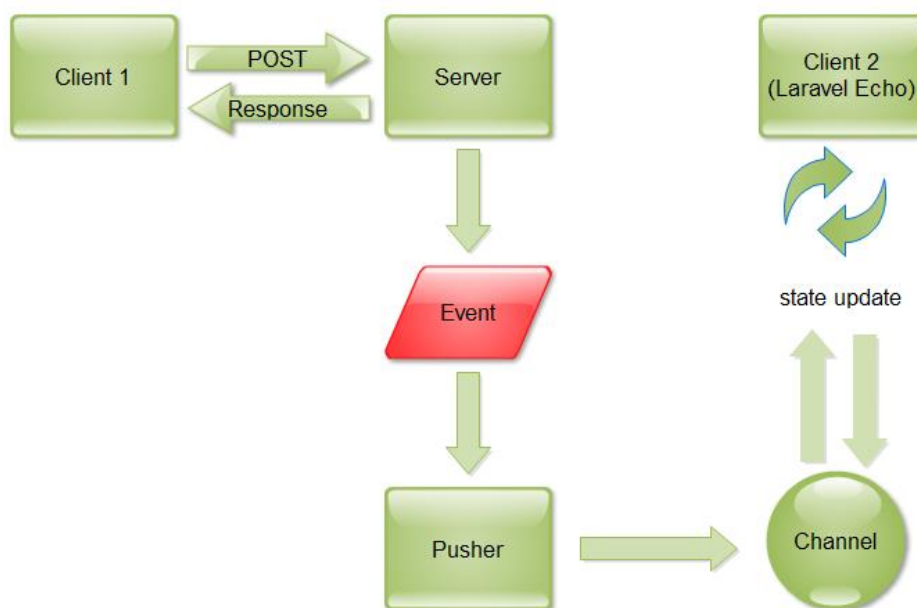


Figura 14 – Comunicarea dintre Client, Server și Pusher

1.7.4 Procesarea informațiilor din timpul jocului

Vehicularea informațiilor ce țin de joc, în timp real, este realizată în strânsă legătură cu capacitatea serviciului Pusher de a transmite evenimente cu date de pe server pe partea clientului. Ca exemplu putem lua cazul în care un utilizator răspunde la întrebarea 1 în timpul jocului în secunda 5. În acest moment aplicația web va face o cerere http de tip PUT către server, unde se va actualiza răspunsul dat de jucător în variabilele din Cache care sunt responsabile cu persistența stării jocului. Acestea sunt mereu în număr de două, întrucât ambii utilizatori care participă la joc trebuie să aibă câte o variabilă corespunzătoare lor.

După modificarea informațiilor care vor fi trimise către oponent, sunt schimbate și apoi returnate datele din cache ce corespund primului jucător, astfel încât răspunsul primit la cererea http să conțină toate detaliile necesare schimbării stării jocului pe aplicația în care este autentificat primul utilizator. De asemenea, fiind salvate în cache, toate informațiile sunt disponibile în cazul unei reîncărcări ale paginii datorate, spre exemplu, pierderii conexiunii. În acest sens, este necesar ca cererile HTTP să fie efectuate de fiecare dată când utilizatorul realizează o acțiune în timpul jocului.


```

$opponent = \App\Models\User::find($opponentId);

$match = array();
$match["status"] = "popup";

$match["userid1"] = $userid;
$match["username1"] = $user->name;
$match["userwidth1"] = $user->imgwidth;
$match["userheight1"] = $user->imgheight;
$match["usershape1"] = $user->imgshape;
$match["userimg1"] = $user->imgpath;

$match["userid2"] = $opponentId;
$match["username2"] = $opponentName;
$match["userimg2"] = $opponent->imgpath;
$match["userwidth2"] = $opponent->imgwidth;
$match["userheight2"] = $opponent->imgheight;
$match["usershape2"] = $opponent->imgshape;

$match["accepted1"] = 0;
$match["accepted2"] = 0;
$match["timestarted"] = Carbon::now();

Cache::forever("Game.".$userid,$match);

```

Figura 15 – Modificarea informațiilor utilizatorului

```

$stranger = array();
$stranger["status"] = "popup";

$stranger["userid2"] = $userid;
$stranger["username2"] = $user->name;
$stranger["userwidth2"] = $user->imgwidth;
$stranger["userheight2"] = $user->imgheight;
$stranger["usershape2"] = $user->imgshape;
$stranger["userimg2"] = $user->imgpath;

$stranger["userid1"] = $opponentId;
$stranger["username1"] = $opponentName;
$stranger["userimg1"] = $opponent->imgpath;
$stranger["userwidth1"] = $opponent->imgwidth;
$stranger["userheight1"] = $opponent->imgheight;
$stranger["usershape1"] = $opponent->imgshape;

$stranger["accepted2"] = 0;
$stranger["accepted1"] = 0;
$stranger["timestarted"] = $match["timestarted"];
Cache::forever("Game.".$opponentId,$stranger);

event(new RespondToGameInvitation(0,$match));

```

Figura 16 – Modificarea informațiilor oponentului

Atunci când variabila corespunzătoare oponentului este modificată, acesta trebuie să primească informația respectivă, pentru ca aplicația să poată actualiza starea jocului (să afișeze oponentului faptul că celălalt jucător tocmai a răspuns corect sau greșit la întrebarea curentă, în secunda 5). Pentru aceasta, de fiecare dată când variabila din cache este modificată, un nou eveniment este creat (după cum se observă în Figura 16), care este trimis de **Pusher** pe parte de client împreună cu variabila respectivă. Pe partea clientului dispunem de **Laravel Echo**, un utilitar ce poate fi folosit pentru a asculta evenimentele transmise de către un server Laravel. Astfel, putem programa acțiuni atunci când este recepționat un eveniment pe un anumit canal.

În cazul nostru, dacă jucătorul a răspuns corect, vom colora bara și cronometrul ce îi corespund în verde. Altfel în roșu. De asemenea vom incrementa numărul de puncte acumulat de acesta până în prezent cu valoarea problemei respective.

```

380      Echo.channel('Game.' + this.$store.state.isAuthenticated)
381      .listen('GameInProgress', (payload) => {
382          console.log("pushed:")
383          console.log(payload)
384          self.player2points = self.player2points + payload.problemvalue
385          self.player2color = payload.color
386      })

```

Figura 17 – Actualizarea stării jocului odată cu sosirea noutăților pe canalul ascultat

Alte scenarii de utilizare ale canalelor Pusher pentru comunicare în timp real (în cadrul jocului) sunt evidențiate în continuare:

- Modificarea *stării* jocului (starea este definită drept etapa în care a ajuns jocul dintre doi utilizatori: neînceput, în plină desfășurare, terminat).
- Actualizarea numărului întrebării la care s-a ajuns în prezent (din totalul de 5 întrebări)
- Transmiterea mesajului că utilizatorul este pregătit (prin accesarea unui buton READY vizibil doar în etapa anterioară începerii jocului).
- Actualizarea numărului de puncte și a secunde în care a răspuns oponentul.
- Transmiterea mesajului că oponentul a răspuns corect (sau greșit).
- Actualizarea scorurilor finale și alegerea câștigătorului (după răspunsurile date celor 5 întrebări).
- Alegerea reluării jocului sau abandonarea lui în cazul reîncărcării paginii (aceasta fiind transmisă și celuilalt jucător, astfel încât aplicația să poată determina dacă jocul continuă sau nu).
- Actualizarea stării jocului în cazul în care unul din utilizatori nu a răspuns la o întrebare (atunci când este terminat timpul alocat acesteia fără ca jucătorul să fi ales una dintre variantele de răspuns). În acest caz jucătorul respectiv nu primește puncte pentru acea problemă și se trece la următorul nivel.

Întrucât se folosesc tehnologii similare în transmiterea informațiilor din timpul jocului, Figura 14 este sugestivă pentru ilustrarea scenariilor de utilizare prezentate anterior. Este, totuși, necesară o completare: spre deosebire de implementarea sistemului de notificări, unde un utilizator începe comunicarea și altul primește mesajul, pe timpul jocului ambii utilizatori pornesc procesul de comunicare, întrucât acțiunile amândurora sunt decisive pentru stabilirea câștigătorului.

2. Prezentarea Aplicației

După ce am detaliat tehnologiile folosite și motivul pentru care au fost necesare, este important să punem în evidență modul în care acestea sunt interconectate, în sensul aprecierii rezultatului final. În continuare va fi prezentată arhitectura aplicației și vom ilustra rolul și importanța cerințelor funcționale amintite, pe scurt, în introducerea lucrării.

2.1 Arhitectura aplicației

Așa cum a fost precizat deja, proiectul prezentat urmează modelul arhitectural MVC, fiecare componentă având rolul bine determinat.

Modelul constituie o clasă ce expune proprietățile și comportamentul unei entități. Spre exemplu, în aplicația noastră un model important este reprezentat de clasa *Problem*. Aceasta conține proprietăți precum identificatorii întrebării și răspunsurilor (corecte / greșite), dar și detalii despre relația sa cu alte modele, precum *User* (orice problemă are un creator, un proprietar).

View-ul este responsabil cu reprezentarea vizuală a datelor, însemnând în cele mai multe dintre cazuri generarea codului HTML randat în navigatorul web. Este important ca view-urile să nu conțină logica aplicației, ci doar să afișeze datele într-un mod corespunzător. În aplicația prezentată, view-urile sunt construite cu ajutorul framework-ului Vue.js.

Controller-ul reprezintă puntea de legătură dintre model și view, Coordonează și controlează logica aplicației, întorcând spre view datele procesate. (5).

Pe lângă aceste elemente centrale, figura ce va ilustra arhitectura aplicației noastre trebuie să surprindă și alte aspecte importante precum mijloacele adiționale de comunicare sau stocarea anumitor date în afara modelului (implicit a bazei de date), anume în cache.

O primă diagramă este pusă în evidență în Figura 18. Aceasta surprinde integrarea serviciilor OneSignal și Pusher în arhitectura aplicației prezentate.

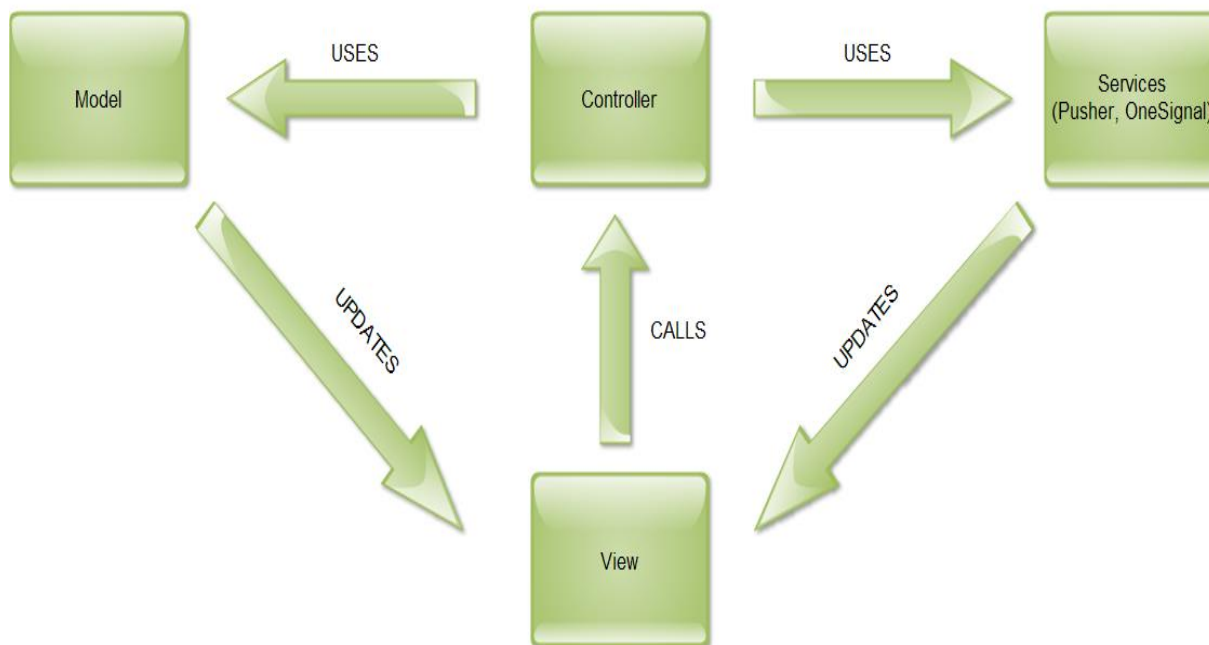


Figura 18 – Arhitectura generală a aplicației

După cum putem vedea, acțiunile utilizatorului de pe partea de view duc la apelul către controller, acesta folosindu-se de model pentru ca pagina web să fie actualizată. Mai mult decât atât, la unele cereri este nevoie ca serverul să apeleze la serviciile OneSignal și/sau Pusher, în scopul trimiterii unei notificări sau a actualizării stării de joc, aceste situații ducând, de asemenea, la schimbarea conținutului văzut de utilizator. (Figura 19, Figura 20)

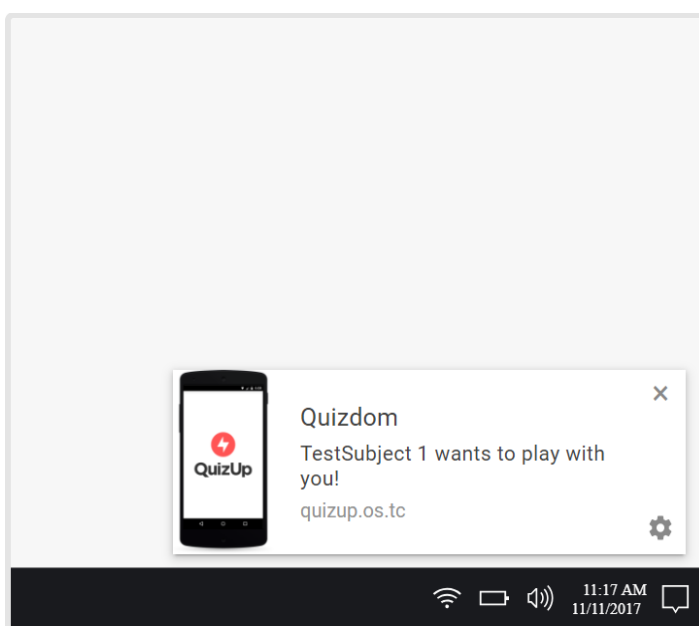


Figura 19 – Notificare Chrome

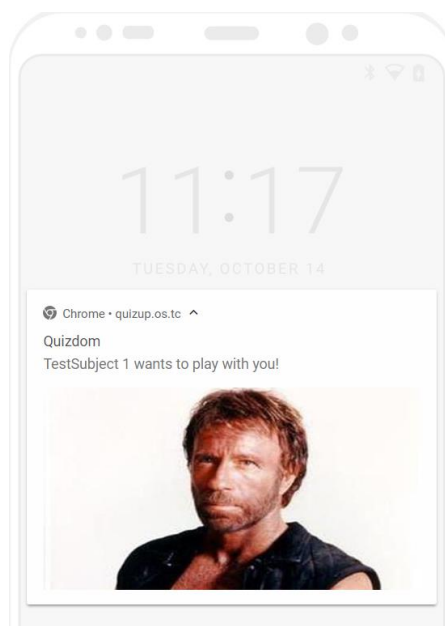


Figura 20 – Notificare Android

Pentru ca arhitectura aplicației să fie descrisă complet, trebuie să precizăm și tipurile de stocare folosite. După cum am mai amintit, nu toate informațiile sunt stocate în baza de date. De exemplu, există date cu caracter temporar, datorat faptului că odată încheiat jocul, importanța acestora dispare. (întrebarea curentă, secunda la care a răspuns primul utilizator la întrebarea X, etc). Din această cauză, este mai convenabil să păstrăm astfel de date în cache. Scădem totodată, numărul de accesări la baza de date, în mod considerabil.

Pentru o mai bună înțelegere, în următoarea figură ilustrăm tipul de dată salvat în fiecare dintre mediile de stocare folosite.

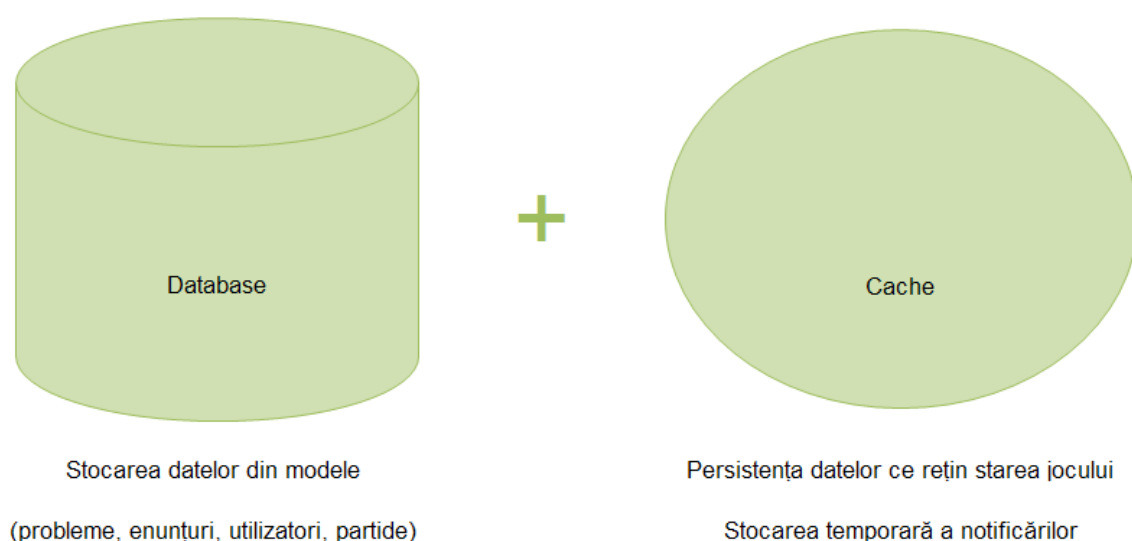


Figura 21 – Medii de stocare a datelor

Printre avantajele folosirii cache-ului amintim scăderea timpului de răspuns al cererilor HTTP, precum și capacitatea informațiilor stocate de a se șterge automat după o anumită perioadă de timp, fapt ce este convenabil spre exemplu, în cazul în care ambii utilizatori pierd conexiunea la internet și nu revin în aplicație un timp îndelungat. De asemenea, este important de știut faptul că zona de memorie a cache-ului este comună tuturor utilizatorilor, fapt ce înlesnește partajarea informațiilor între jucători.

Totodată, folosirea cache-ului implică anumite limitări de memorie, precum și de tipul de stocare. (Nu putem stoca decât perechi cheie-valoare). De asemenea, resursele aflate temporar în această zonă de memorie sunt greu de urmărit, astfel încât dezvoltarea unor metode care să le poată administra corect și eficient crește în dificultate, în raport cu numărul

funcționalităților care au nevoie de respectivele resurse. Pentru aplicația prezentată, următoarele funcționalități au nevoie de datele aflate în cache:

- *Actualizarea stării jocului* (răspunsul dat de oponent, trecerea la următoarea întrebare, timpul alocat răspunsului, etc).
- *Prezentarea utilizatorilor care sunt în așteptare pentru începerea unui joc rapid.* (utilă pentru pagina de așteptare, în care utilizatorul poate vedea în timp real oponentul cu care este repartizat să joace).
- *Intrarea în lista de așteptare sau părăsirea acesteia pentru începerea unui joc rapid.* (în pagina de așteptare utilizatorul se poate înscrie în coada de așteptare sau o poate părăsi, identificatorul său fiind adăugat sau șters din obiectul care stochează lista utilizatorilor care sunt în căutarea unui joc).
- *Prezentarea datei ultimei activități în aplicație a fiecărui utilizator* (are scop informativ, poate ajuta utilizatorul să anticipeze dacă va primi răspunsul unei posibile provocări adresate unui oponent – nu are rost să provoci la joc un utilizator care nu a mai fost activ în aplicație de foarte mult timp).
- *Afișarea notificărilor interne noi* (are scopul de a prezenta utilizatorului, în timp real, informații, noutăți).
- *Accesarea / ștergerea notificărilor interne* (odată ce sunt accesate, noutățile sunt șterse din cache).

Așadar, se observă faptul că fiecare dintre resursele administrate se potrivesc mai bine, în contextul aplicației noastre, unui tip anume de stocare.

Pe de o parte, resursele ce țin de utilizator, care pot fi modificate la cererea acestuia, sau cele care nu sunt schimbate des, sunt stocate în baza de date relațională. Exemple reprezentative pot fi: datele personale ale clientului (nume, parolă, poză de profil, domenii de interes, clasa de care aparține), textul întrebărilor / răspunsurilor, proprietarul acestora, problemele cumpărate de utilizatorul autentificat.

Pe de altă parte, datele stocate cu caracter temporar, sau care nu servesc unor statistici pe termen lung, dar care sunt accesate și modificate des pentru funcționalitățile aplicației, sunt stocate în cache. Exemple reprezentative pentru acest tip de informație pot fi: mesajul unei notificări, lista cu întrebări alese pentru un joc între doi utilizatori, etc.

2.2 Funcționalitățile aplicației

În continuare vom prezenta opțiunile pe care le are utilizatorul în aplicația prezentată. Acestea reflectă implementarea, pe parte de client, a funcționalităților enumerate în introducerea acestei lucrări.

2.2.1 Funcționalități comune

Acestea stau la baza realizării majorității aplicațiilor web. Importanța lor este evidentă, motiv pentru care nu vom insista foarte mult pe baza funcționalităților comune. Dintre acestea amintim:

- *Autentificare / Înregistrare* – permit stocarea și personalizarea datelor ce aparțin unui utilizator, precum și facilitarea accesului către restul funcționalităților.
- *Accesarea / Editarea datelor personale* – permit modificarea informațiilor menționate în formularul de înregistrare, precum și editarea unor preferințe ce țin de aspectul aplicației.
- *Meniu principal* – Accesarea funcționalităților pe baza unui meniu, în urma autentificării.
- *Pagină Ghid* – prezentare scurtă a opțiunilor disponibile, modalitate de învățare pentru clienții care sunt la prima accesare.
- *Mecanism de transmitere a părerilor sau a problemelor legate de aplicație (Feedback)* – părerile utilizatorilor constituie mereu un aspect important în cadrul oricărei aplicații, acestea fiind utile pentru găsirea unor erori sau pentru realizarea unor actualizări.

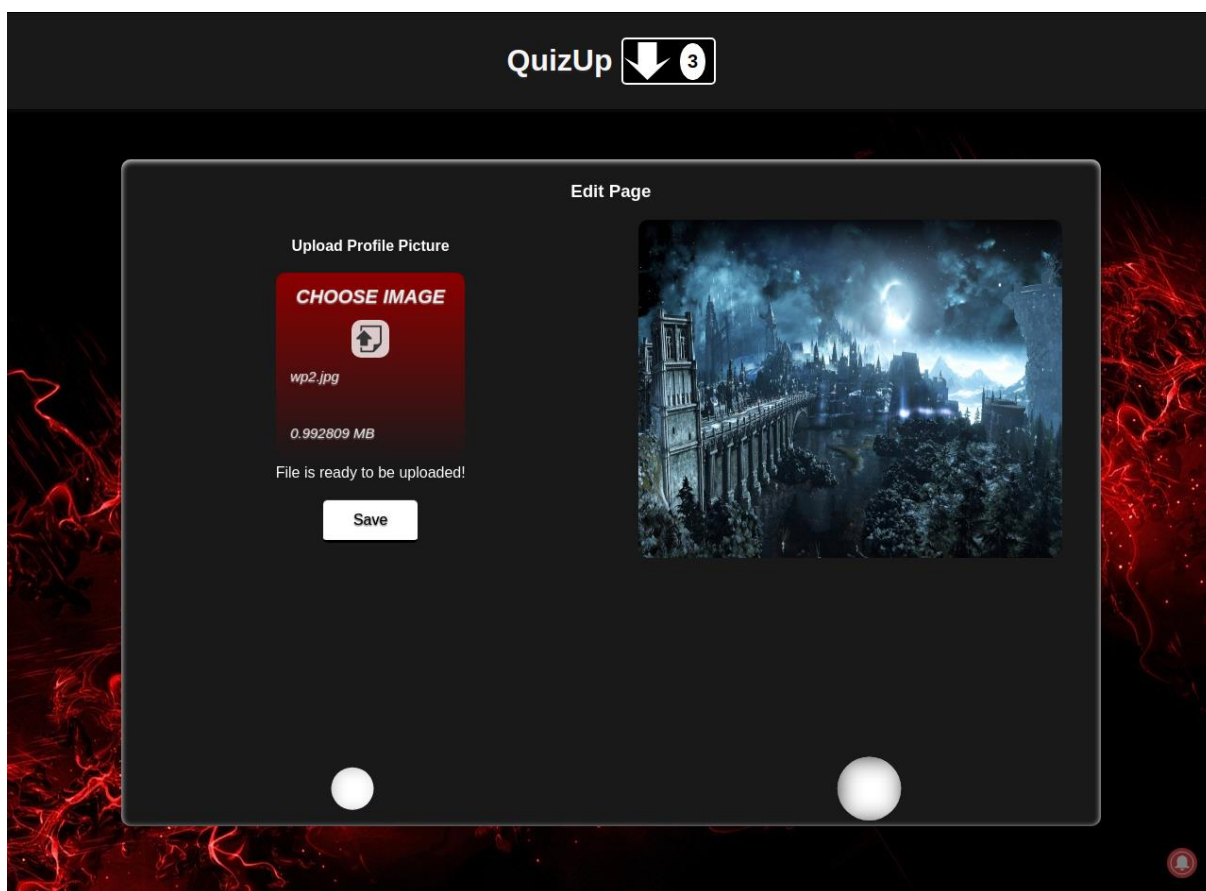


Figura 22 – Editarea datelor personale: Schimbarea pozei de profil

2.2.2 Funcționalități specifice

Funcționalitățile care fac parte din această categorie sunt particulare aplicației prezentate aici. Acestea sunt de un interes mai mare, datorită faptului că dezvoltarea lor a implicat o dificultate crescută și un grad de originalitate. Cele mai importante dintre acestea sunt:

- *Consultarea / alegerea unei clase* – opțiune importantă pentru jocul între mai mulți utilizatori, fiecare clasă având diverse beneficii / puteri în timpul jocului, de care utilizatorul se poate ajuta.
- *Primirea notificărilor atât în aplicație cât și în afara acesteia* – Natura aplicației și a funcționalităților pe care le oferă a dus la constatarea că e necesară o modalitate de a semnala noutățile utilizatorului (pentru ca, la rândul lor, alți utilizatori să nu fie nevoiți să aștepte mult timp un răspuns)
- *Verificarea listei de jucători și a disponibilității acestora* – foarte importantă, întrucât în urma accesării acesteia se pot trimite provocări directe la joc, pentru utilizatori care doresc să pornească un joc cu un prieten sau cu o persoană anume.

- *Verificarea listei de așteptare* – facilitează pornirea unui joc rapid, ca urmare a intrării în coada de așteptare (Serverul este, în acest caz, cel care găsește un oponent potrivit pentru utilizator).
- *Creare / Editare Probleme* – încurajând utilizatorii să cerceteze probleme ce fac parte din diferite categorii, asigurăm crearea conținutului nou care va fi accesibil tuturor, experiența fiind una mereu plăcută, nu repetitivă.
- *Joc* – de două tipuri. Încurajează învățarea și acumularea de cunoștințe noi, bazându-se pe spiritul competitiv al utilizatorilor.
- *Raportarea problemelor* – pentru cazul în care o problemă nu reflectă categoria menționată sau este greșită.

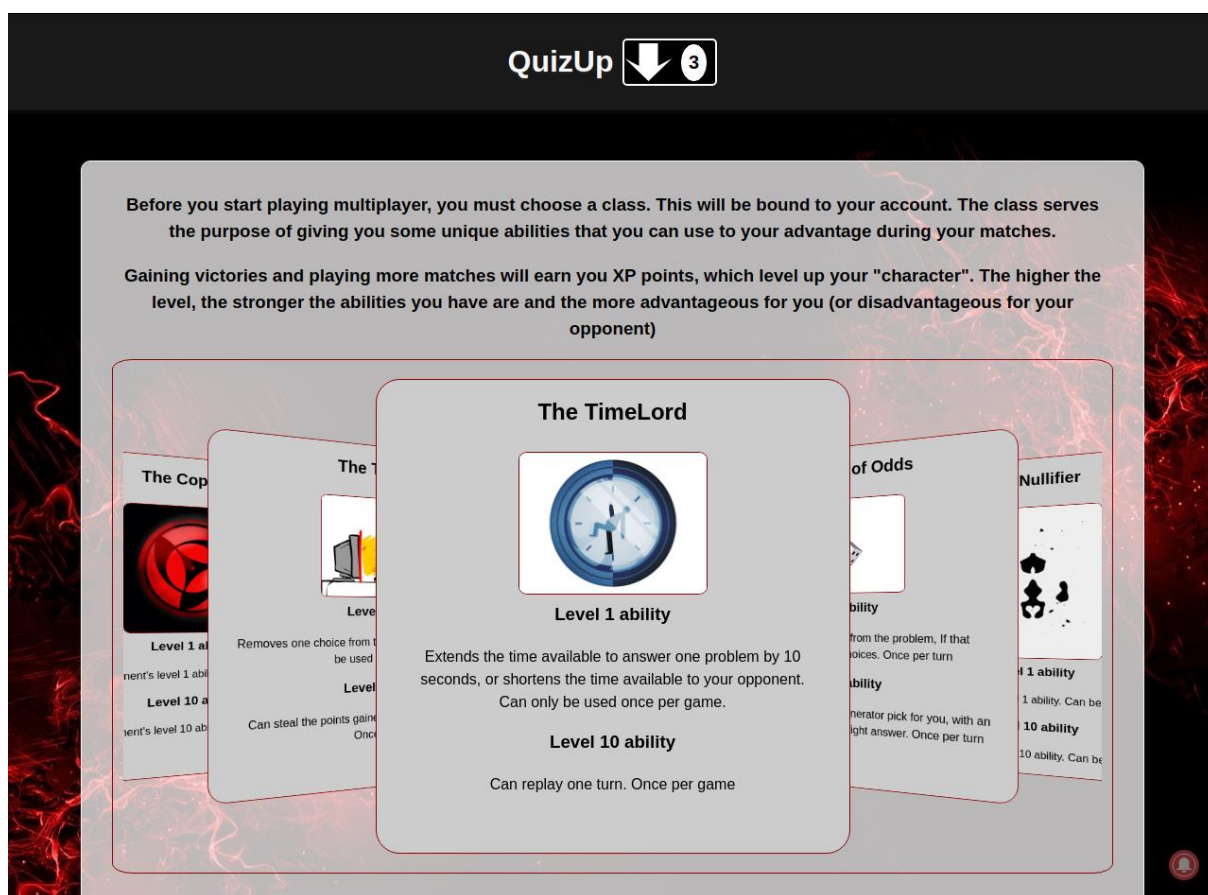


Figura 23 – Consultarea claselor existente

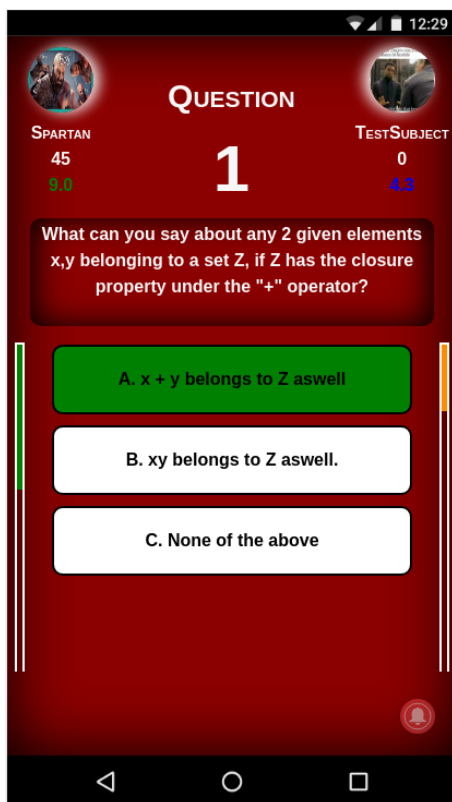


Figura 24 – Joc între 2 clienți

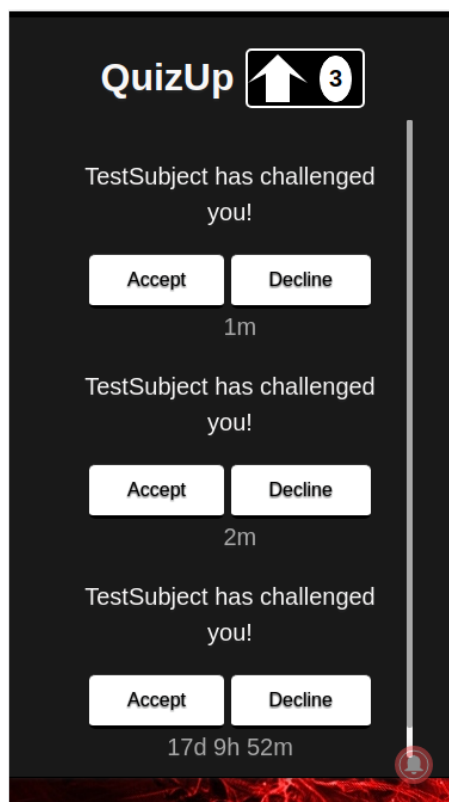


Figura 25 – Afișarea notificărilor

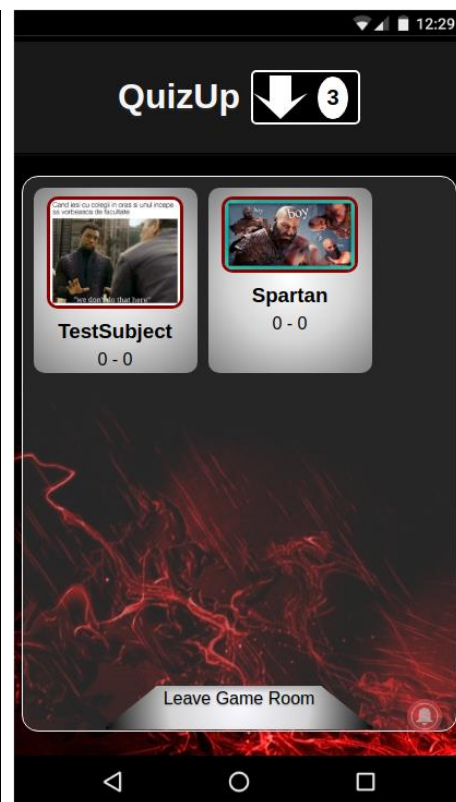


Figura 26 – Camera de așteptare pentru joc

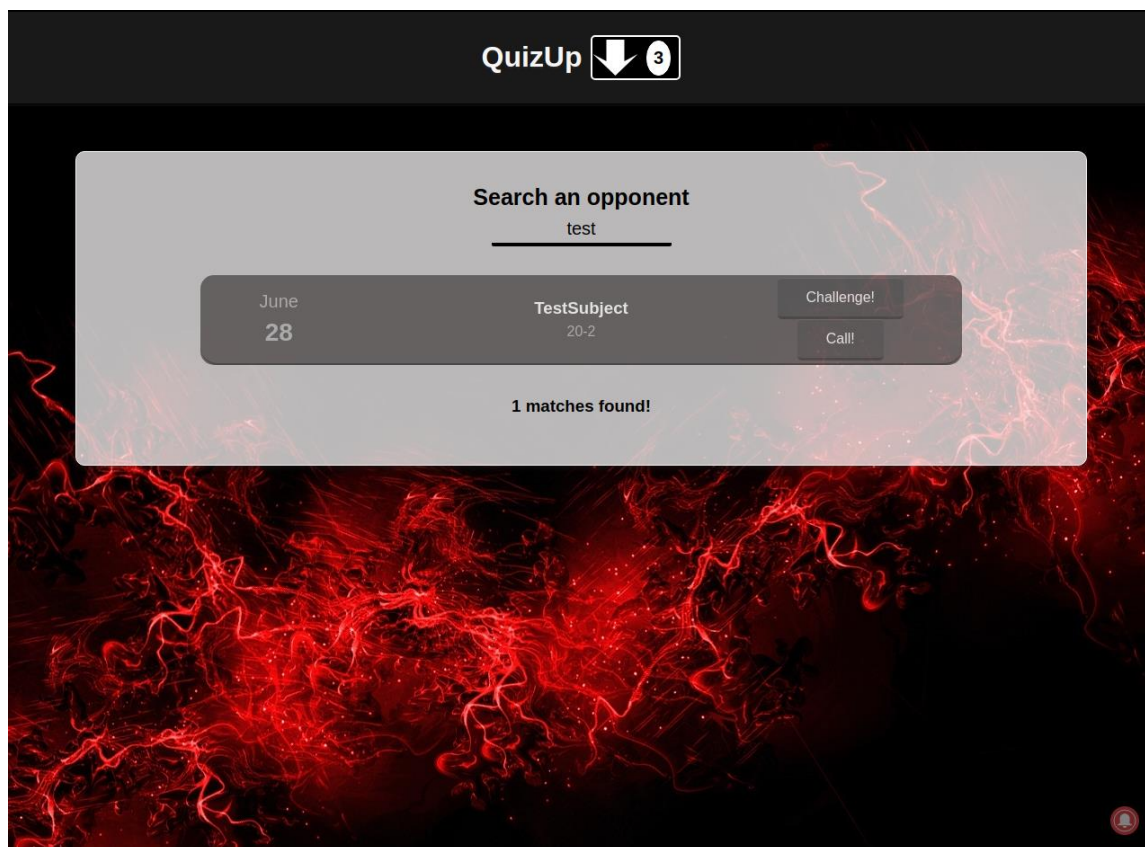


Figura 27 – Căutarea unui jucător + Data ultimei activități ale acestuia

3. Optimizări și dificultăți întâmpinate

Acest capitol prezintă parcursul sarcinilor mai dificile, pentru care a fost nevoie de timp și studiu, pentru găsirea unor soluții potrivite. De asemenea, vor fi accentuate și metode care au fost încercate, dar nu au oferit o rezolvare acceptabilă. Scopul acestor paragrafe constă în evidențierea observațiilor personale, realizate în decursul dezvoltării proiectului, asupra arhitecturii sau a modului de comunicare dintre entități.

3.1 JavaScript vs PHP - Front-End

Folosind Laravel pe parte de Back-End, una dintre soluțiile aflate la îndemână pentru dezvoltarea părții de client a aplicației web era reprezentată de *Blade*. Acesta poate procesa date returnate de metodele apelate din controller. Asemănându-se cu *Razor*, un alt motor pentru template-uri folosit în dezvoltarea proiectelor pe baza tehnologiilor asp.net, *Blade* conține o suită de metode utile pentru dinamizarea conținutului web. Dintre acestea amintim:

- *@yield* – folosit pentru inserarea conținutului unei pagini în pagina curentă. Acceptă ca argument numele view-ului de tip *blade* pe care dorim să îl inserăm.
- *@if*, *@else*, *@elseif*, *@endif* – instrucțiuni condiționale între care putem avea cod HTML. Acesta va fi inclus doar dacă expresia din paranteze este evaluată adevărat. Un exemplu de utilizare constă în prezența unei variabile care să semnaleze existența unei poze de profil pentru utilizatorul autentificat. În cazul în care există, aceasta este preluată și afișată în diferite secțiuni din pagina de meniu.
- *@isset* – folosit pentru a testa existența unei variabile. Dacă informația respectivă nu este trimisă de controller, *@isset* va returna fals.
- *@switch*, *@case*, *@default*, *@endswitch* – instrucțiuni condiționale înlănțuite, între care putem avea cod HTML.
- *@for*, *@endfor*, *@foreach*, *@endforeach* – instrucțiuni repetitive, sunt folosite deseori pentru afișarea unor structuri PHP complexe (liste) sub forma unor tabele sau liste ordonate sau neordonate.

Aceste directive facilitează crearea unui conținut dinamic în cadrul paginilor, însă prezintă anumite limitări. Cea mai importantă limitare constă în faptul că pentru a folosi *Blade*, metodele din controller trebuie să returneze, pe lângă datele procesate, un fișier de tip *blade* (un view). Acest fapt duce la nevoia de a reîncărca pagina web la fiecare interacțiune cu serverul

Pentru o aplicație simplă, obișnuită, această nevoie nu influențează experiența pe care o are utilizatorul. Însă, odată cu creșterea complexității aplicației prezentate, s-a observat că metodele disponibile prin *Blade* pentru dinamizarea conținutului web dintr-o pagină nu erau suficiente, fiind necesare structuri auxiliare, specifice aplicației noastre.

În plus, una dintre funcționalitățile aplicației, anume jocul între doi utilizatori, nu se poate realiza fără o continuă interacțiune cu serverul, motiv pentru care implementarea strict folosind *Blade*, ar fi putut constitui o problemă.

Din aceste motive, o abordare bazată pe cereri de tip AJAX a fost favorabilă, întrucât acestea nu cauzează reîncărcarea întregii pagini – informațiile pot fi actualizate în aceeași pagină, în urma primirii răspunsului de tip JSON din partea controller-ului. Astfel, este evidentă încă o diferență ce are caracter pozitiv: nu mai sunt servite pagini întregi de către controller, la fiecare cerere, ci doar răspunsuri cu date în format JSON. Dimensiunea acestora este mult mai mică, fapt ce poate contribui la performanța crescută în anumite funcționalități care necesită foarte multe cereri spre server (de exemplu, jocul).

3.2 Comunicare de la server la client

Toate cererile de tip HTTP sunt făcute de client, serverul fiind cel care procesează datele și apoi răspunde. Una dintre dificultățile întâmpinate a fost găsirea unei modalități prin care serverul să fie cel care inițiază comunicarea. O primă variantă care a fost considerată a fost realizarea unui număr mare de cereri HTTP într-un timp scurt, pentru a primi actualizări făcute de alți utilizatori asupra datelor, în timp real. Aceasta nu este, însă, o abordare eficientă, consumând multe resurse.

O altă variantă luată în considerare a fost folosirea unei librării bazate pe javascript, care să faciliteze comunicarea bidirecțională. Un exemplu este *socket.io*, însă testând funcționalitățile oferite, am observat că astfel de librării sunt compatibile cu un server *node.js*, nu unul bazat pe PHP.

OneSignal, serviciul folosit pentru trimiterea de notificări, are și o funcționalitate interesantă, aceea de a trimite pachete de date împreună cu notificarea propriu-zisă, informațiile putând fi folosite pe parte de client. Părea, inițial, să fie o variantă plauzibilă pentru implementarea jocului, întrucât notificările pot fi ascunse, astfel încât aplicația să interpreteze doar datele sosite împreună cu acestea. După ce am încercat o implementare de acest fel, s-a observat faptul că nu este potrivită pentru scenariul jocului, întrucât transmiterea notificărilor

poate dura și până la 2-3 secunde, fapt ce distruge orice speranță de a avea impresia unei comunicări în timp real în timpul jocului.

În final, s-a ajuns la concluzia că o implementare bazată pe tehnologia *Websocket* este cea mai bună pentru cerințele specificate. Serviciul *Pusher* este folosit precis în acest scop. Informațiile care vin de la server sunt transmise în timp real pe canalele ascultate de către client, datele putând fi accesate imediat ce sunt recepționate. De asemenea, existența canalelor oferă o modalitate de a structura comunicarea. Spre exemplu, putem oferi acces într-un canal destinat unui joc doar celor doi utilizatori care sunt în competiție. Astfel, toate mesajele sunt trimise doar celor destinați să le primească.

3.3 Afișarea testelor grilă

Referindu-ne la jocul dintre doi utilizatori, putem spune cu certitudine că acesta este un tip de joc **zero-sum**, în sensul că dacă unul dintre utilizatori obține victoria, celălalt pierde. (exceptând cazul de remiză). În acest sens, cunoscând faptul că omul are spirit competitiv, orice jucător va face ce îi stă în putere pentru a câștiga (6). Din aceste cauze, predictibilitatea constituie un factor ce trebuie evitat în cadrul jocului. Considerăm următoarea situație:

1. Un utilizator provoacă pe cineva la joc.
2. Persoana respectivă acceptă, chiar dacă vede că are șanse mici să câștige.
(primul utilizator are mai multe ore dedicate jocului, al doilea este la primul joc)
3. Jocul începe, problemele sunt alese și trimise ambilor jucători.
4. Primul jucător, având experiența orelor petrecute deja în joc, ghicește răspunsul doar citind întrebarea și amintindu-și ordinea în care sunt afișate răspunsurile.
5. Fiind mai rapid, indiferent de nivelul de înțelegere a conceptelor, primul jucător acumulează mai multe puncte și câștigă.

Situația prezentată evidențiază un motiv pentru care jucătorii noi se pot simți, pe bună dreptate, dezavantajați. Nu putem opri jucătorii din a înțelege răspunsurile unor întrebări, acesta fiind întocmai scopul ascuns al aplicației. Cu toate acestea, putem împiedica utilizatorii din a memora poziția pe care se află, în grilă, răspunsul corect, prin randomizarea ordinii în care sunt afișate răspunsurile.

Fiecare problemă care este aleasă pentru joc are un răspuns corect și unul, două sau chiar trei răspunsuri greșite. Acestea sosesc din back-end în aceeași ordine, motiv pentru care am dezvoltat o soluție care asigură schimbarea ordinii răspunsurilor la orice încărcare a paginii:

fiecărui răspuns îi este asignat un număr de ordine, între 0 și numărul de răspunsuri ale întrebării respective, din care scădem 1. Această asignare este făcută în mod aleatoriu, urmând ca ordinea răspunsurilor din lista care le conține să fie actualizată pe baza acestor numere repartizate. Eliminăm, astfel, ocazia jucătorilor de a profita pe nedrept prin memorarea pozițiilor răspunsurilor corecte.

3.4 Redis vs File - Caching

Un beneficiu al folosirii cache-ului pe parte de server constituie creșterea vitezei de răspuns. Configurația inițială făcută de Laravel determină ca locația tuturor datelor stocate în cache să fie în fișiere din dosarul *storage/framework/cache*. Comparativ cu stocarea în baze de date relaționale, se observă faptul că diferența de timp necesar pentru răspuns este mică, dacă păstrăm aceeași configurație. În schimb, aceasta devine semnificativă atunci când folosim un serviciu de stocare a datelor, așa cum este *Redis*. Diferența de viteză este datorată, în principal, faptului că stocarea și servirea datelor sunt realizate în/din memoria RAM. În cazul în care se cere stocarea permanentă a informațiilor, aceasta este realizată și pe disc.

Deși obținem o creștere a performanței (timpul de răspuns al serverului este de 2-3 ori mai mic), există și câteva dezavantaje, care sunt explicate clar în documentația oficială Redis. (7). Dintre acestea vom detalia două dintre ele, care ne afectează în cazul proiectului nostru:

- Salvăm datele în memoria RAM, ceea ce presupune o limitare din punctul de vedere al cantității de informație ce poate fi stocată. Nu putem trece de limita de memorie RAM pe care o avem disponibilă.
- Sistemele bazate pe 64 de biți vor folosi mai multă memorie decât cele bazate pe 32 de biți, în special atunci când sunt stocate chei și valori de dimensiune mică, din cauza faptului că pointerii ocupă mai multă memorie (8 bytes) în aceste sisteme. Ne situăm în această categorie, întrucât stocăm structuri de date de tip dicționar, care au chei și valori relativ simple (*chei* – șir de caractere scurt, *valori* – liste, valori numerice).

Chiar dacă există unele dezavantaje, beneficiile oferite de această abordare sunt (mai) importante, motiv pentru care am ales să respectăm această modalitate de a stoca informațiile.

3.5 Funcționalități pentru viitor

3.5.1 Pornirea rapidă a jocului

Interfața cu utilizatorul poate fi simplificată, în sensul reducerii numărului total de pași necesari pornirii jocului. Situația avută în vedere este atunci când doi utilizatori folosesc aplicația pe telefon, sunt aflați la o distanță mică unul față de celălalt și vor să înceapă un joc. O soluție care poate fi luată în calcul constituie integrarea unui generator, respectiv cititor de coduri QR, în aplicația web. Codul QR poate stoca informații aflate în format JSON, exact tipul de dată de care avem noi nevoie atunci când solicităm datele necesare începutului jocului.

Astfel, etapele scenariului de utilizare sunt următoarele:

1. Primul utilizator deschide aplicația, având codul qr accesibil din prima pagină a meniului.
2. Cel de-al doilea utilizator folosește cititorul pentru a porni vehicularea informațiilor necesare începerii jocului.
3. Jocul începe direct, pe telefoanele ambelor utilizatori.

QR Code Generator



Figura 28 – Exemplu Cod QR generat

În Figura 28 este ilustrată o mică aplicație web auxiliară care afișează rezultatul creării unui cod QR.

Putem genera câte un astfel de cod pentru fiecare utilizator al aplicației noastre, în care să regăsim toate datele de care are nevoie clientul care scanează codul, pentru a începe jocul. (identificator, nume, număr de victorii, număr de înfrângeri, etc)

Ulterior, fișierele generate pot fi servite static, astfel încât fiecare utilizator să aibă acces la codul QR care îi corespunde (pentru a putea fi scanat de către oponent).

Generarea codului QR a fost făcută cu ajutorul librăriei **BaconQrCode**.

Informația codată are scop simbolic, aceasta fiind reprezentată de un obiect JSON cu diverse proprietăți.

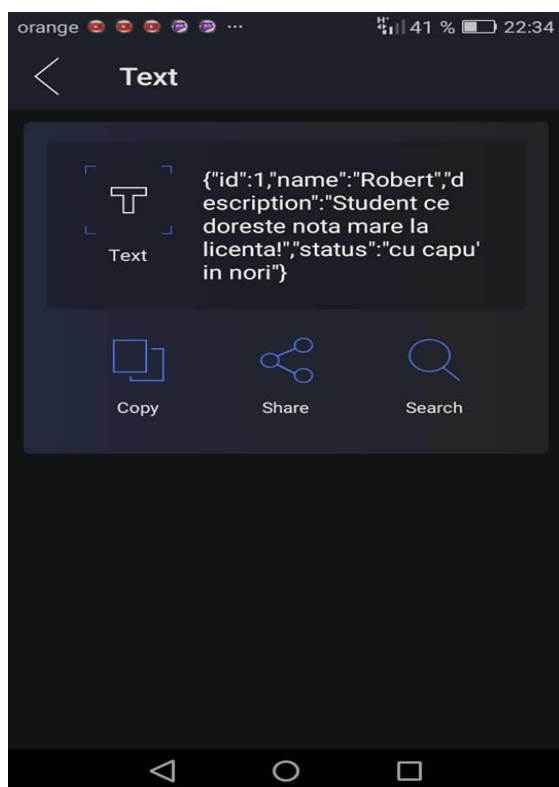


Figura 29 – Exemplu Cod QR procesat

Se observă în Figura 29 faptul că datele stocate în codul QR pot fi interpretate cu succes de cititor.

Putem declanșa un eveniment în JavaScript atunci când este citit cu succes un cod QR și are formatul așteptat, astfel încât să putem trimite o cerere către server pentru a anunța faptul că se dorește începerea jocului.

Având toate datele de care este nevoie, se poate de asemenea sări și peste etapa de confirmare, respectiv de acceptare a invitației la joc, întrucât utilizatorul nu ar putea scana o imagine de pe telefonul oponentului dacă nu ar dori ambii participanți să pornească jocul.

Ultimul pas ar fi trimiterea datelor de pe server spre client prin **Pusher**, în cazul utilizatorului care a oferit codul QR spre citire, pentru începerea jocului. (cel care a efectuat scanarea va primi automat răspuns din partea serverului, datorită faptului că el a făcut cererea HTTP).

Datele din imagine sunt rezultate din citirea codului QR din Figura 28, prin intermediul unei aplicații pentru mobil - **SmartScan**.

3.5.2 Joc între mai mulți utilizatori

Aceasta pare a fi o funcționalitate complicată, însă nu va ridica multe probleme. Motivul pentru care sunt convins de acest lucru constă în faptul că Pusher permite ca mai mult de 2 clienți să poată asculta pe un singur canal. Astfel, un canal public poate fi o sursă comună de informații transmise, în timp real, tuturor utilizatorilor abonați.

Pentru realizarea jocului dintre doi utilizatori, deja folosim canale Pusher pentru transmiterea informațiilor. Așadar, infrastructura necesară acestei funcționalități este deja realizată. Singura diferență ar consta în faptul că ar fi mai mulți utilizatori abonați la același canal, acesta fiind public.

Dificultatea principală ar fi situată la nivel de client, unde ar trebui implementată o metodă care să asculte pe canalul corespunzător și care să fie capabilă să proceseze informațiile venite de la un număr *variabil* de utilizatori.

Concluzii

În cadrul dezvoltării aplicației prezentate, au fost implementate funcționalități care sunt comune oricărei aplicații (autentificare, înregistrare, operații CRUD), precum și o serie de funcționalități particulare (sistemul de notificări, persistența datelor în cazul pierderii conexiunii, comunicare inițiată de server). Tocmai aceste funcționalități particulare aduc diversitate și abordează teme de interes în cadrul comunității de dezvoltare web.

Se observă, în rândul aplicațiilor web din prezent, o nevoie de a suplimenta modalitatea curentă de comunicare (clientul face o cerere, primește un răspuns de la server), întrucât aceasta nu mai face față cerințelor actuale. Utilizatorii preferă să poată primi informații, noutăți, fără accesarea unei pagini. Spre exemplu, din perspectiva unui client, aș dori ca un blog să mă poată anunța atunci când este publicat un articol pe care doresc să îl urmăresc. Alternativa, verificarea constantă a blogului respectiv, nu creează o experiență plăcută. Această abordare, în care serverul poate lua inițiativă, este tot mai populară în rândul aplicațiilor mari, fiind preluată de platforme precum **Facebook** sau **Youtube**. Un alt exemplu poate fi constituit de aplicațiile bancare (majoritatea aplicațiilor home-banking oferă, acum, posibilitatea de a notifica utilizatorul atunci când este retrasă o sumă de bani). Aplicația Quizdom oferă și ea o soluție pentru implementarea unui sistem de notificări, intrând în rând cu tendințele actuale prezentate.

Privind înapoi, spre munca depusă în realizarea acestui proiect, pot să precizez că sunt mulțumit cu rezultatul final obținut. Sunt de părere că aplicația web dezvoltată este funcțională și poate fi folosită de studenți, pentru a învăța, prin intermediul jocului, concepte noi și pentru a consolida fundamentul lor teoretic.

De asemenea, prin creare de conținut nou (probleme tip grilă create de utilizatori) aplicația prezentată stimulează ingeniozitatea utilizatorilor și încurajează realizarea unor conexiuni între informațiile vizate.

Direcții viitoare pentru tema abordată și pentru proiectul prezentat pot fi reprezentate de implementarea funcționalităților discutate în subcapitolul 3.5, precum și în găsirea unei soluții optime pentru găzduirea aplicației dezvoltate.

Bibliografie

1. Directory Structure. *Laravel Docs*. [Interactiv] 2 May 2018. <https://laravel.com/docs/5.6/structure>.
2. Bean, Martin. *Laravel 5 Essentials*. Birmingham : Packt, 2015.
3. Pusher Limited. What is Pusher? *Pusher*. [Interactiv] 2018. <https://pusher-community.github.io/real-time-laravel/introduction/what-is-pusher.html>.
4. Leggetter, Jason Lengstorf and Phil. *Realtime Web Apps*. s.l. : Apress, 2013.
5. Pusher Limited. JavaScript quick start. *Pusher*. [Interactiv] 2018. https://pusher.com/docs/javascript_quick_start.
6. Keith, Jeremy. *Bulletproof AJAX*. Berkeley, CA : New Riders, 2007.
7. Jess Chadwick, Todd Snyder, Hrusikesh Panda. *Programming Asp .NET MVC 4*. s.l. : O'reilly, 2012.
8. Togelius, Georgios N. Yannakakis and Julian. *Artificial Intelligence and Games*. 2018.
9. redis. *FAQ - redis*. [Interactiv] 2018. <https://redis.io/topics/faq>.
10. FAQ. *redis*. [Interactiv] <https://redis.io/topics/faq>.
11. FAQ - Redis. *redis*. [Interactiv] <https://redis.io/topics/faq>.

Anexa 1

Laravel este un framework open-source construit în jurul modelului arhitectural **MVC** (Model-View-Controller) dezvoltat în php, care a câpătat o popularitate imensă pe parcursul a câtorva ani, având atât un manager de dependențe dedicat, cât și o documentație bine structurată ce permite implementarea cu ușurință a unor funcționalități generale, care servesc oricărui proiect.

Odată instalat framework-ul, putem crea un nou proiect ce va avea următoarea structură:

- Dosarul *app* – aici se regăsește logica aplicației, dosarul fiind încărcat automat sub spațiul de nume *App*. În interiorul acestuia se află:
 - Dosarul *Console* conține toate comenzile de tip **Artisan**⁸.
 - Dosarul *Http* cuprinde entități legate de serverul Http. Printre acestea numărăm: **Controllers** și **Middlewares**.
 - Dosarul *Exceptions* conține un fișier denumit *Handler*, în acesta regăsindu-se o clasă care descrie comportamentul aplicației în cazul în care sunt aruncate excepții.
 - Dosarul *Providers* cuprinde clase care furnizează diferite servicii utile precum autentificare - *AuthServiceProvider*, rutare – *RouteServiceProvider* sau transmitere de evenimente – *BroadcastServiceProvider*.

În dosarul *app* se regăsesc, de asemenea, modelele din structura MVC, care moștenesc clasa *Model* din **Eloquent**⁹.

- Dosarul *bootstrap* – conține fișierul *app.php*, folosit pentru a încărca framework-ul. Regăsim totodată dosarul *Cache*, ce cuprinde fișiere de tip cache generate de framework pentru creșterea performanței, cum ar fi fișiere cache pentru rutare și pentru servicii.
- Dosarul *config* – conține fișiere utile dezvoltatorului atât pentru configurarea serverului, cât și a framework-ului. Reglarea acestora determină comportamentul unor secțiuni diferite din aplicație, cum ar fi baza de date, serviciul de transmitere a

⁸ Consolă în care sunt implementate comenzi care ajută dezvoltatorul în realizarea aplicației web.

⁹ ORM specific framework-ului Laravel.

fișierelor, serviciul de transmitere a evenimentelor (broadcasting) sau implementarea unei cozi pentru sarcinile consumatoare de timp.

- Dosarul *database* – conține fișierele de tip migrare și alte fișiere necesare pentru popularea bazei de date cu înregistrări pentru testare. Migrările le folosim pentru a crea tabelele, pentru a efectua schimbări în structura bazei de date și a tabelelor, cât și pentru a reveni la o stare anterioară a bazei de date (dacă este nevoie).
- Dosarul *public* – aici regăsim fișierul *index.php*, acesta fiind punctul de intrare pentru toate cererile din aplicație. Este necesar de asemenea, pentru încărcarea automată a fișierelor (autoloading). Aici se află, totodată, fișierele servite către client (html, css, imagini, etc).
- Dosarul *resources* – cuprinde dosarul cu view-uri, precum și fișierele necompile javascript (**EcmaScript 5**) sau css (**sass, scss**).
- Dosarul *routes* – aici găsim fișierele necesare rutării. Putem defini rute pentru aplicația noastră, sau putem activa metode numite **middlewares** care modifică structura răspunsului sau filtrează cererile către server. Funcțiile middleware pot fi efectuate înainte sau după generarea răspunsului.
- Dosarul *storage* – acesta include:
 - Dosarul *app*, unde se află fișiere generate de aplicația noastră. Spre acest dosar putem crea o legătură simbolică în dosarul *public*, astfel încât fișierele aflate aici să accesibile doar grupurilor de utilizatori alese de noi.
 - Dosarul *framework* conține fișiere generate de framework și fișiere de tip cache.
 - Dosarul *logs* cuprinde fișiere unde sunt înregistrate răspunsuri date de server sau posibile erori întâmpinate. Pentru ca aceste fișiere să poată fi scrise, dosarul trebuie să aibă permisiuni de scriere.
- Dosarul *tests* – cuprinde teste automate. Orice clasă trebuie să fie denumită cu sufixul "Test". Pentru a rula testele este folosit framework-ul **PHPUnit**.
- Dosarul *vendor* – aici regăsim conținutul dependențelor **Composer**.

(1)

În rădăcina proiectului există, de asemenea, fișiere importante:

- *.env*, unde putem configura constante precum numele aplicației, tipul de bază de date, numele de utilizator și parola pentru conectare la baza de date sau adresa aplicației.

- *composer.lock*, unde sunt precizate dependențele aplicației.

O interacțiune cu serverul, inițiată de client, trece prin următorii pași:

- Utilizatorul face o cerere spre server de tip Get, Post, Put, Delete la o adresă corespunzătoare.
- Punctul de intrare al oricărei cereri către o aplicație Laravel este fișierul *public/index.php*. Ulterior este executat scriptul din *bootstrap/app.php*, care creează o instanță a furnizorului de servicii Laravel.
- În continuare clasa din fișierul *app/Http/Kernel.php* încarcă componentele serverului (baza de date, validări, rutare). Dacă cererea trece de filtrarea făcută de middleware-uri și dacă aceasta accesează un link valid (aflat în fișierele de rutare), procesul continuă.
- În cele din urmă, aceeași clasă conține o metodă care este apelată, primind cererea (**Request**) și returnând un răspuns (**Response**). Comportamentul acesteia este determinat de metoda corespunzătoare adresei accesate (legătura fiind definită în fișierul de rutare). Metoda respectivă poate fi scrisă direct în fișierul de rutare, însă este recomandat ca aceasta să se regăsească într-un **controller**.
- Metoda apelată din **controller** poate accesa, dacă este nevoie, baza de date, prin intermediul modelului: fiecare tabel din baza de date are câte un model care îi corespunde, **Eloquent** având metode pentru selectarea, inserarea, editarea și ștergerea înregistrărilor din tabele.

Următoarea figură ilustrează, într-un mod simplificat, un exemplu de cerere http către server. Cererea este una de tip GET, resursa cerută fiind "cats". Modelul primește comanda *all()*, care este tradusă în baza de date prin "select * from cats".

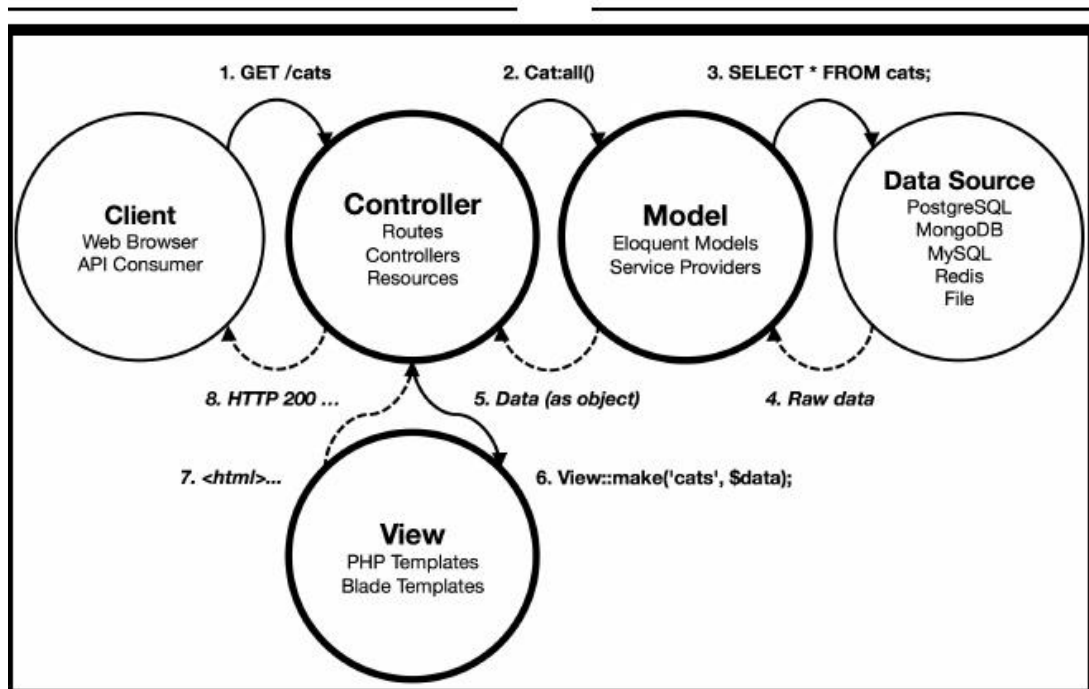


Figura 30 – Cerere HTTP către server (2)