UNIVERSITATEA „ALEXANDRU IOAN CUZA” IAŞI

**Facultatea de Informatică**

****

Lucrare de licenţă

**Quizdom**

Propusă de

***Robert –Ilie Vicol***

**Sesiunea:** Iulie, 2018

Coordonator Ştiinţific:

***Lector, dr. Cosmin Vârlan***

UNIVERSITATEA „ALEXANDRU IOAN CUZA” IAŞI

**Facultatea de Informatică**

**Quizdom**

***Vicol Robert -Ilie***

**Sesiunea:** Iulie, 2018

Coordonator Ştiinţific:

***Lector, dr. Cosmin Vârlan***

**DECLARAŢIE PRIVIND ORIGINALITATE ŞI 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 instituţie de învățământ superior din ţară 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 deţin referinţa precisă a sursei;

− reformularea în cuvinte proprii a textelor scrise de către alţi autori deţine referinţa

precisă;

− codul sursă, imagini etc. preluate din proiecte open source sau alte surse sunt utilizate

cu respectarea drepturilor de autor și deţin referinţe precise;

− rezumarea ideilor altor autori precizează referinţa precisă la textul original.

Iaşi,

Absolvent Vicol Robert -Ilie

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(semnătura în original)

**DECLARAŢIE DE CONSIMŢĂMÂNT**

Prin prezenta declar că sunt de acord ca Lucrarea de licență cu titlul „Quizdom”, codul sursă al programelor și celelalte conţinuturi (grafice, multimedia, date de test etc.) care însoţesc această lucrare să fie utilizate în cadrul Facultăţii de Informatică. De asemenea, sunt de acord ca Facultatea de Informatică de la Universitatea „Alexandru Ioan Cuza” Iași să utilizeze, modifice, reproducă şi să distribuie în scopuri necomerciale programele-calculator, format executabil şi sursă, realizate de mine în cadrul prezentei lucrări de licenţă.

Iaşi,

Absolvent Vicol Robert -Ilie

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ (semnătura în original)

# 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 ințelege 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) QuizUp**[[1]](#footnote-1)– 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.

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.

## Funcționalități

*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 înregistrați pot vizualiza lista utilizatorilor care sunt activi[[2]](#footnote-2), 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 intre 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.

*Sistem de notificări* – utilizatoriilogați 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 acțiuni[[3]](#footnote-3) ale utilizatorului.

*Păstrarea stării aplicației* – reprezintă una dintre cele mai importante funcționalități ale aplicației. Permite reluarea activității[[4]](#footnote-4) în orice moment, chiar și în timpul jocului.

## 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 resurse[[5]](#footnote-5).
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 intrebări,”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.

În cadrul componentei de joc a aplicației, informațiile se transmit în timp real prin intermediul modelului Publish/Subscribe. Este folosită, 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:

* Alegerea și configurarea serverului Apache, precum și a framework-ului Laravel.
* 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 si 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 dependințe cerut în proiect poate importa și folosi componentele publicate de mine. Printre acestea, se numără următoarele:

* vueAnimatedBar : bară de lungime / lățime variabilă, ce simbolizează modificarea dimensiunii, scurgerea timpului.
* vueTimer : componentă cu logica necesară unui cronometru sau a unei număratoare inverse.
* vueCube : componentă de meniu cu aspectul unui cub.

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

# Dezvoltarea aplicației web

În acest capitol voi detalia tehnologiile folosite în realizarea proiectului, precizând în același timp motivele pentru care am optat pentru aceste tehnologii.

## Laravel 5

Laravel este un framework open-source construit în jurul modelului arhitectural **MVC** (Model-View-Controller) dezvoltat în php, care a căpătat o popularitate imensă pe parcursul a câtorva ani, având atât un manager de dependințe 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**[[6]](#footnote-6).
* 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**[[7]](#footnote-7)**.**

* 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 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 necompilate 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 nostră. Spre acest dosar putem crea o legătură simbolică in 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 dependințelor **Composer**.

(Directory Structure, 2018)

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 dependințele 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 severului (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”.

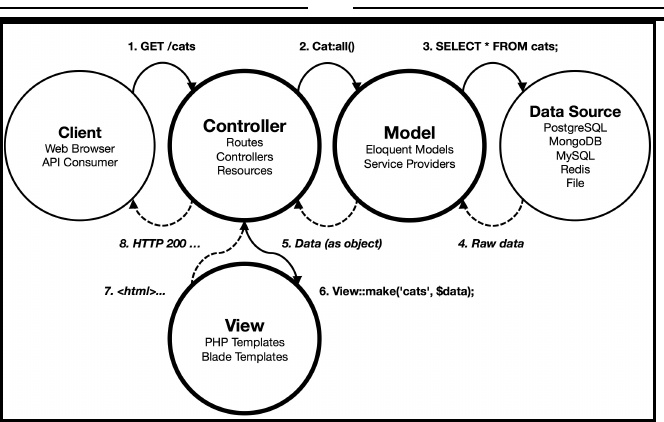


Figura 1 – Cerere HTTP către server (Bean, 2015)

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 refolosirea 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 descris în unul din capitolele următoare.

## MySQL

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 1, un avantaj al folosirii framework-ului Laravel este acela că suportă mai multe metode de a stoca date: **PostgresSQL**, **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 inlătură 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 sa 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().*

## 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 customizat. Am construiot 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 logat.
  + Numele utilizatorului logat.
  + Scorul utilizatorului logat.
  + 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.

## OneSignal

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 2, 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.

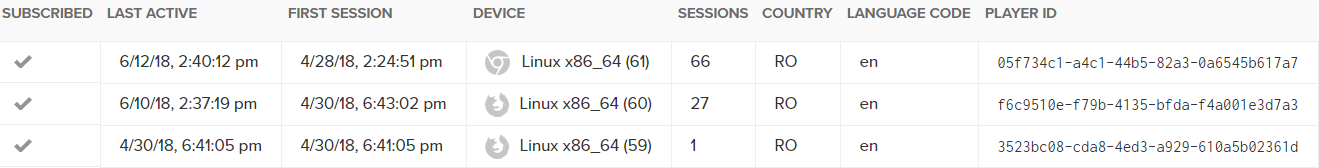


Figura 2 – identificatorul unic (PLAYER\_ID) și baza de date OneSignal.

Întrucât dorim să targetăm utilizatori anume 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 folosește 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 metoda 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 logați.

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 sa trimitem notificarea, precum și de un identificator al aplicației OneSignal și o cheie privată pentru apeulul 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 3 exemplifică modalitatea prin care serverul configurează și trimite notificările de tip extern:

Figura 3 – Trimiterea notificărilor externe aplicației.

**C2**

**S**

**OS**

**S**

**C1**

## Pusher Channels

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 4. Serviciul lor este folosit inclusiv de Github, dovedind stabilitate si performanță.

|  |  |
| --- | --- |
| https://pusher-community.github.io/real-time-laravel/assets/img/pusher-circles.png  Figura 4 – serviciul Pusher (Pusher Limited, 2018) | **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**. (Leggetter, 2013)

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

|  |  |
| --- | --- |
| Figura 5 – inițializare Pusher (Pusher Limited, 2018) | ”APP\_KEY” constituie cheia primită odată cu înregistrarea unei aplicații pe platforma **Pusher**. |
| Figura 6 – Pusher – abonare la canal pe parte de client (Pusher Limited, 2018) | ”my\_channel” reprezintă numele atribuit unui canal de către noi pe parte de server, la care se va abona clientul. |
| Figura 7 – Pusher – ascultare la un canal pentru primirea mesajelor (Pusher Limited, 2018) | “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 8:

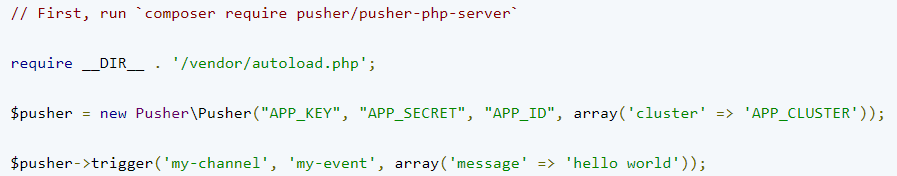


Figura 8 – Pusher – declanșarea unui eveniment (Pusher Limited, 2018)

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”*. 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.

## Vue

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* ).



Figura 9 – 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 logare, 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 10 – Diagrama Vuex (https://vuex.vuejs.org/vuex.png)

Observăm, în Figura 10, 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 11 surprinde un exemplu concret al implementării sistemului *Vuex*:

|  |  |
| --- | --- |
| Figura 11 – 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 reduntantă 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.

## Comunicare server – client

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.

### Cerere / Creare / Editare / Stergere 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 logarea, înregistrarea, editarea datelor personale, verificarea problemelor create sau adăugarea unor probleme noi.

### Notificare externă aplicației

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). Pentru a explica mai bine, următoarea diagramă ilustrează procesul explicat anterior:

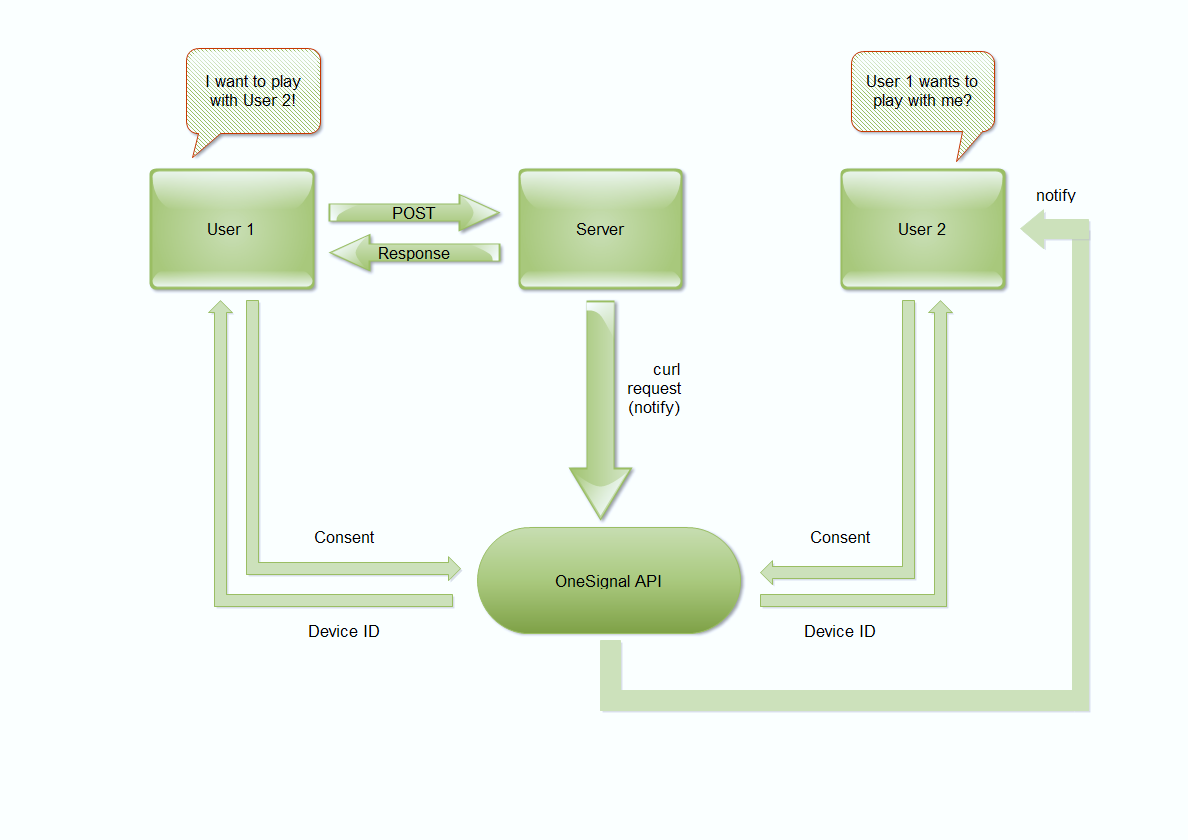


Figura 12 – Comunicarea dintre aplicație, server și OneSignal API

După cum se observă în Figura 12, în momentul deschiderii aplicației, după ce își exprimă 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 logat 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 (cu aceeași cheie 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.

### Notificare internă aplicației

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.

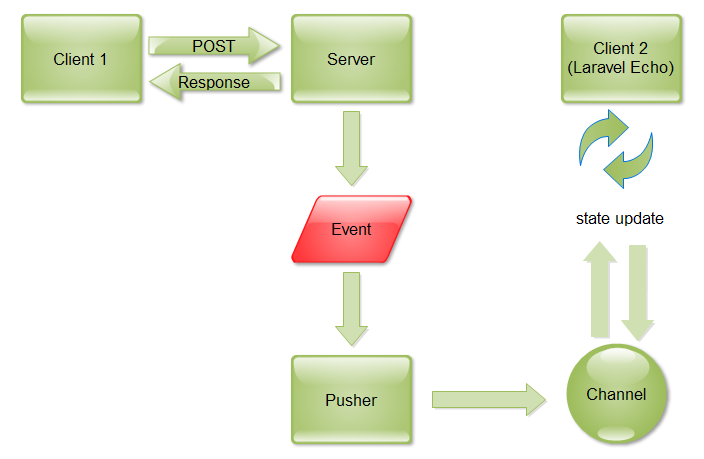


Figura 13 – Comunicarea dintre Client, Server și Pusher

### 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 logat primul utilizator. De asemenea, fiind salvate in 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.

|  |  |
| --- | --- |
| Figura 14 – Modificarea informațiilor utilizatorului | Figura 15 – 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 14), 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.

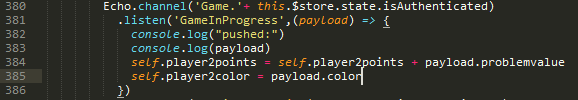


Figura 16 – Actualizarea stării jocului odată cu sosirea noutăților pe canalul ascultat

# Bibliografie

Bean, M. (2015). *Laravel 5 Essentials.* Birmingham: Packt.

*Directory Structure*. (2018, May 2). Preluat de pe Laravel Docs: https://laravel.com/docs/5.6/structure

Leggetter, J. L. (2013). *Realtime Web Apps.* Apress.

Pusher Limited. (2018). *JavaScript quick start*. Preluat de pe Pusher: https://pusher.com/docs/javascript\_quick\_start

Pusher Limited. (2018). *What is Pusher?* Preluat de pe Pusher: https://pusher-community.github.io/real-time-laravel/introduction/what-is-pusher.html

1. https://www.quizup.com [↑](#footnote-ref-1)
2. Utilizatorii care au aplicația deschisă. [↑](#footnote-ref-2)
3. Invitația la joc adresată oponentului, respectiv acceptarea provocării de către oponent. [↑](#footnote-ref-3)
4. Orice informație este stocată temporar

   (în cazul în care este pierdută conexiunea la internet). [↑](#footnote-ref-4)
5. Întrebări eronate, probleme raportate sau cu răspunsuri greșite, probleme raportate pe nedrept, etc. [↑](#footnote-ref-5)
6. Consolă în care sunt implementate comenzi care ajută dezvoltatorul în realizarea aplicației web. [↑](#footnote-ref-6)
7. ORM specific framework-ului Laravel. [↑](#footnote-ref-7)