**UNIVERSITATEA BABEŞ-BOLYAI CLUJNAPOCA**

**FACULTATEA DE MATEMATICǍ ŞI**

**INFORMATICǍ**

**SPECIALIZAREA INFORMATICĂ – ROMANĂ**

**LUCRARE DE LICENŢĂ**

**Boardify**

**Conducător ştiinţific:**

**Dr. CZIBULA Istvan**

**Absolvent:**

**Antinie Radu**

**2020**

Cuprins

1. **Introducere**
   1. Motivatie
   2. Provocari
   3. Descrierea
2. Microservicii

2.1 Introducere  
2.2 Sabloane arhitecturale  
2.3 Elemente definitorii ale unui microserviciu  
2.4 De ce folosim microservicii?

1. **Deployment**

3.1 Service Discovery

3.2 Load balancing

3.3 API gateway

3.4 Comunicare între microservicii

3.5 Containere

1. Boardify

4.0 Specificatii

* 1. Analiza
  2. Proiectare

4.2.1 Service discovery

4.2.2 Gateway API

4.2.3 Auth

4.2.4 Group

4.2.5 Game

4.2.6 User

4.2.7 Persistență

* 1. Implementare

4.4 Testare

* 1. Deployment
  2. Tehnologii folosite
  3. Implementari existente

1. Concluzie şi dezvoltari viitoare

Bibliografie

1. Introducere

Odata cu dezvoltarea tehnologiei interactiunea dintre oameni scade şi astfel apare o nevoie a unor aplicaţii care să creeze legaturi în viata reala între acestia.

Jocurile de masa (board games) au fost frecventate de fiecare din noi în copilarie dar multi oameni continua să le joace pe tot parcursul vietii fiind o activitate pentru toate varstele.

**1.1 Motivatie** Inainte ca tehnologia să joace un rol atât de important în viata noastra aceste jocuri se aflau în casa fiecaruia dintre noi. Astazi exista în fiecare oras localuri speciale pentru pasionatii de aceste jocuri.

**1.2 Provocari**

Totusi cum relatiile dintre oameni au avut de suferit în ultimii ani, multi jucatori nu au cunostiinte interesate de aceste jocuri. Astfel, tehnologia ar putea juca un rol important în a pune în legatura aceste personae cu scopul de a se intalni într-un local şi a experimenta impreuna placerea jocurilor de masă.

**1.3 Descriere**

O solutie prin care tehnologia ar putea rezolva problema lipsei de interactiune între persoare ar fi creearea unei platforme unde pasionatii de board games pot fi potriviti. Acesta ar fi un punct de plecare urmand ca acestia să se intalneasca într-un local.

Aceasta lucrare este organizata dupa cum urmeaza:

* Capitolul 1 serveste drept o scurta introducere.
* Capitolul 2 prezinta o sinteza teoretica a

arhitecturii bazate pe microservicii cu anumite proprietati ale acesteia dar şi alternative.

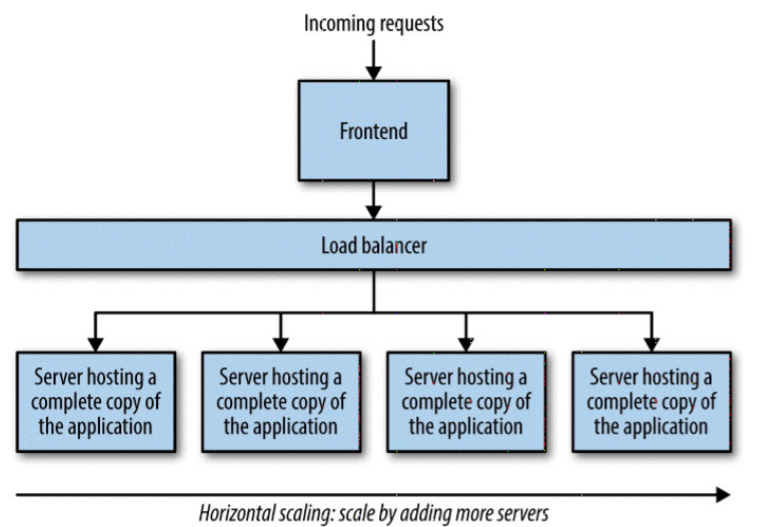
* Capitolul 3 surprinde procesul de deployment.
* Capitolul 4 reprezinta partea practica a aplicaţiei şi explica modul în care aceasta a fost implementata.
* Capitolul 5 isi propune să sumarizeze punctele importante surprinse în lucrare şi posibilele dezvoltari viitoare ale aplicaţiei.

1. Microservicii

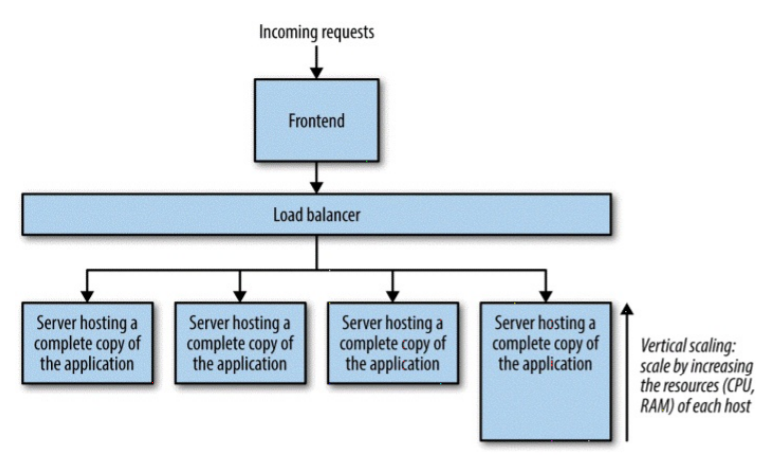
**2.1 Introducere**

Microserviciile sunt un șablon architectural pentru software ce devine foarte popular în zilele noastre mulțumită multor avantaje precum flexibilitatea şi ușurința în scalare. Împărțindu-le în echipe mici într-o firmă oferă multă eficiență în dezvoltare. Totuși, pentru a alege dacă să folosești sau nu microservicii nu este suficient. Este nevoie să știi ce provocări ai să intâmpini și cum le poți soluționa.

**2.2 Șabloane arhitecturale**

Aproape orice aplicație software dezvoltată in zilele noastre este divizată în 3 parți disctincte: un frontend, un backend si o bază de date. Cererile sunt facute prin partea de client (frontend), datele sunt procesate pe backend si persistă intr-o bază de date.  
 Există trei moduri în care aceste părți pot fi structurate pentru a creea o aplicație. Cele mai multe echipe pun frontend-ul și backend-ul într-un repository și il rulează ca un singur fișier executabil cu o bază de date separată. Altă aboradare este separarea front-end-ului de back-end cu o bază de date externă. Alte aplicații ce nu au nevoie de o bază de date si stochează toate datele in memorie obișnuiesc să combine cele trei părți într-un singur repository.  
 Aplicațiile sunt de obicei construite avănd in vedere aceasta aceasta arhitectură ce de obicei este independentă de produsul obținut sau scopul aplicații in sine.  
 La inceputul unei companii, cand produsele software sunt simple si numărul de dezvoltatori este mic complexitatea proiectului este și ea mică. Atunci când compania crește si sunt angajați mai mulți dezvoltatori acest lucru se schimbă.  
 Odată cu proiecte mai mari apar si costuri mai mari in rularea si întreținerea aplicaților, astfel apare nevoia angajării unor ingineri de sistem.  
 Un al doilea lucru care se întâmplă este faptul ca odată cu implementarea unor noi funcționalități crește dimensiunea codebase-ului și complexitatea acestuia.  
 A treia si cea mai mare problemă este scalarea verticală si orizontală a aplicației. Creșterea traficului pune tensiune suplimentară pe scalare si problemele de performanță, astfel apare nevoia de a avea mai multe servere pentru a găzdui aplicația. Adăugând mai multe servere, o copie a aplicației este livrată fiecărui server și totodată load balancere cu scopul de a distribui echitabil munca în cadrul acestor servere. Scalarea verticală devine o necesitate a aplicației pe masură ce aplicația începe sa proceseze un număr mare de sarcini, astfel aplicația este livrată unor servere mai puternice ce pot gestiona mai bine situația. 

Scalatul orizontal se referă la mărirea numarului de servere pentru găzduirea aplicației iar scalatul vertical se referă la creșterea resurselor mașinii gazdă.



Pe parcurs ce compania crește și numărul de angajați devine mai mare apar multe probleme. Astfel orice funcționalitate nou introdusă aduce după sine probleme de mentenanță, necesitatea de a scrie multe teste pentru a asigura faptul ca noua funcționalitate nu le compromite pe cele existente. Astfel atat procesul de dezvoltare cât si de deployment este mult îngreunat. Aplicațiile construite în acest fel au o arhitectura monolitică.  
 Deși nu toate aplicațiile monolitice sunt rele și nu suferă toate de aceste probleme de scalabilitate, majoritatea aplicațiilor ce au această arhitectură întâmpină aceste provocări pe parcursul dezvoltării lor. Motivul pentru care aplicațiile monolitice sunt predispuse la aceste probleme este faptul ca însăși natura lor monolitică se opune scalabilității. Pentru a putea scala este nevoie de concurență si partiționare, aceste două lucruri fiind greu de obținut intr-o arhitectură monolitică.  
 Multe companii mari ca Netflix, Amazon, eBay, Uber și Twitter au întâmpinat aceste probleme de scalabilitate după ce aplicațiile lor au ajuns sa ruleze pe mii de servere. Aceste provoca  
 Conceptul de microserviciu implică o aplicație mică ce are o singură responsabilitate, lucru ce permite dezvoltarea si deploy-ul individual al acestuia. Totuși daca putem face deploy la un singur microserviciu nu înseamnă ca ar trebui sa o facem, acesta face parte dintr-un sistem mare cu mai multe microservicii, aplicația neputând funcționa doar cu acest serviciu ce are doar o mică responsabilitate.  
 Motivația unei arhitecturi bazate pe microservicii este dezvoltarea mai multor aplicații de dimensiuni mici cu responsabilități limitate pentru a putea oferi independență si autonomie.

[11]   
 Acest tip de arhitectură a apărut în vizorul comunității software în jurul anului 2014 și a fost un răspuns direct la provocările de a scala, atât din punct de vedere tehnic, cât și organizatoric, aplicațiile monolitice.

**2.3 Elemente definitorii ale unui microserviciu**

O arhitectură bazată pe microservicii are următoarele caracteristici:

* logica aplicației este descompusă în componente slab granulate, cu limite de responsabilitate bine definite ce sunt coordonate pentru a livra o soluție
* fiecare componentă are un domeniu mic de responsabilitate şi este deployed complet independent față de celelalte; microserviciile ar trebui să aibă responsabilitatea pentru o singură parte din domeniu; de asemenea, un microserviciu trebuie să fie reutilizabil în cadrul mai multor aplicații
* comunicarea microserviciilor se bazează pe niște principii de bază şi adoptă protocoale de comunicare ușoare (lightweight) ca HTTP şi JSON (JavaScript Object Notation) pentru a schimba date între consumator şi producător;
* implementarea tehnică de bază pentru serviciu este irelevantă pentru că aplicațiile mereu comunică folosind un protocol neutru de comunicare (JSON în cele mai multe cazuri); acest lucru înseamnă că o aplicație dezvoltată folosind microservicii poate fi construită folosind multiple limbaje de programare şi tehnologii;
* microserviciile, prin natura lor - mici, independente şi distribuite - oferă organizațiilor să aibă echipe mici de dezvoltare cu domenii bine definite de responsabilitate; echipele lucrează împreună, cu scopul de a livra aplicația, dar fiecare echipă este responsabilă doar pentru microserviciul pe care lucrează;

Folosirea microserviciilor oferă o flexibilitate mare şi autonomie pentru echipele de dezvoltare dar echipele descoperă faptul că natura mică și independentă a microserviciilor le fac deployable to the cloud. Odată ce serviciile sunt în cloud, dimensiunea lor mică face ușoară pornirea unui număr mare de instanțe ale aceluiași serviciu și astfel aplicația devine ușor scalabilă.

**2.4 De ce folosim microservicii** Companiile ce aveau ca grup țintă comunitatea locală au început să realizeze faptul că pot ajunge la piața globală folosind internetul. Dar, odată cu o piață mai mare, competiția crește. Acest nivel crescut de competitivitate face ca dezvoltatorii să trebuiască să se gândească la următoarele aspecte atunci când construiesc o aplicație:

* complexitatea mult crescută: aplicațiile ce folosesc o singură bază de date şi nu integrează alte aplicații nu mai sunt norma; aplicațiile din ziua de azi trebuie să comunice cu multiple servicii și să folosească multiple baze de date nu doar în cadrul companiei dar și în cadrul internetului;
* reducerera timpului de livrare: clienții nu mai vor să aștepte pentru lansări anuale ale unui pachet software; aceștia se aşteaptă ca funcționalitățile să fie descompuse astfel încât noile lansări să poată fi realizate în câteva săptămâni fără nevoia unei lansări a întregului produs software;
* performanţa şi scalabilitatea: aplicaţiile globle fac extrem de dificil de prezis cât de mult volum trebuie gestionat de o aplicaţie şi când acesta va apărea; aplicaţiile trebuie să scaleze rapid peste multe servere când nevoia apare sau dispare;
* clienţii se aşteaptă ca aplicaţiile lor să fie mereu disponibile: eşuarea într-o parte a aplicaţiei nu ar trebui să determine eşuarea întregii aplicaţii;

Pentru a îndeplinii aceste aşteptări şi a construi aplicaţii ce pot scala mult, ca dezvoltatori trebuie să descompunem aplicaţia în servicii mai mici ce pot fi dezvoltate şi livrate indepenendent unul faţă de altul. Dacă descompunem aplicaţia în servicii mai mici şi le mutăm dintr-un singur artefact monolitic, putem construi sisteme ce sunt:

* flexibile – serviciile decuplate pot fi compuse şi re-aranjate pentru a oferi noi funcţionalităţi; cu cât unitatea de cod pe care o funcţionalitate adăugată este mai mică, cu atât este mai puţin complicat să schimbăm codul şi dureaza mai puţin să testăm şi să facem deploy;
* de încredere – decuplarea serviciilor unei aplicaţii implică faptul că atunci când o degradare are loc într-o parte a aplicaţiei, aceasta nu determină întreaga aplicaţie să eşueze; eşuările pot fi descoperite şi izolate într-o parte a aplicaţiei fără a determina eşuarea întregii aplicaţii; acest lucru permite serviciului să se degradeze dintr-o eroare irecuperabilă;
* scalabilitate – serviciile decuplate pot fi distribuite cu uşurintă în cadrul mai multor servere, făcând scalarea funcţionalităţilor ca atare; în cadrul unei arhitecturi monolitice, unde toată logica întregii aplicaţii este în acelaşi modul, întreaga aplicaţie trebuie să scaleze chiar dacă doar o mică parte a aplicaţiei o impune; scalare unor servicii mai mici este mult mai eficientă din punct de vedere a costurilor;

Microservicile sunt o tehnică eficientă de modularizare. Apelând un microserviciu din altul necesită ca dezvoltorii să scrie cod ce facilitează comunicarea dintre cele două microservicii. Acest fapt are consecința ca atunci când un dezvoltator creeaza o astfel de dependență acesta o face intenționat si conștient. Într-o aplicație monolitică este ușor ca programatorul sa folosească o clasă deoarece nu apare nevoia de a scrie cod explicit ce să faciliteze acea dependență. Această dependență poate să nu fie intenționată din punct de vedere arhitectural.  
 Problema apare atunci când două părți ale aplicației depind una de alta și acest lucru ar fi putut fi evitat, deoarece acum pentru a dezvolta una din părți este nevoie de a face modificări in ambele părți ale aplicației. Microserviciile fac aceste dependențe să nu poată fi formate neintenționate deoarece modulele sunte separate și apare nevoia comunicării în rețea între microservicii.  
 O altă problemă ce poate apărea în aplicațile monolitice este atunci când un sistem software vechi devine greu de dezvoltat deoarece calității proaste a codului. Acest sistem poate fi foarte mare, dezvoltatorii sa nu înțeleagă cu exactitate cum funcționează anumite părți ale aplicației. O astfel de aplicație poate să devină aproape imposibilă datorită riscului mare de a cauza deteroiorări la modificarea codului.  
 Acesta este un capitol în care arhitectura bazată pe microservicii excelează deoarece un microserviciu reprezintă doar o mică unitate a aplicației ce are o singură responsabilitate și poate fi ușor de înțeles și modificat. Această calitate face ca microservicile să poată fi ușor de înlocuit fară a fi necesară modificarea unei părți mari din cod.  
 Atunci când un nou microserviciu este creat nu suntem constrânși în a folosi vechile tehnologii folosite până atunci in aplicație. Dacă microserivicul este independent și din punct de vedere al domeniului, logica acestuia este ușor de înțeles. Dezvoltatorii nu sunt obligați să înțeleagă logica întregii aplicații atunci când vor să facă modificări într-un microserviciu, ci trebuie să înțeleagă doar microserviciul respectiv. Cunoșințele cu privire la domeniul problemei sunt absolut necesare atunci când se vrea înlocuirea unui microserviciu.  
 În plus, atunci când lucrăm la înlocuirea unui microserviciu chiar dacă cauzăm ca serviciul respectiv să devină indisponibil restul aplicației poate funcționa independent de acesta. Astfel riscurile asociate cu înlocuirea scad mult deoarece nu riscăm cauzarea indisponibilității întregii aplicații.  
 Începerea unui nou proiect software este simplă deoarece la început este puțin cod ce aduce după sine puțină complexitate și dezvoltatorii pot progresa rapid. Pe parcurs ce aceasta crește si arhitectura se complică timpul necesar dezvoltării software crește și el. Odata cu timpul acest sistem va deveni un sistem legacy. Dacă folosim o arhitectură bazată pe microservicii, atunci când sistemul devine unul legacy, acesta poate fi înlocuit. Astfel microservicile pot menține productivitatea inițială datorată de un proiect cu o baza de cod mică si puțină complexitatea. Deci acest șablon arhitectural face dezvoltarea rapidă sustenabilă în timp si oferă o soluție atunci cand un sistem devine vechi și se transformă într-un sistem legacy. Totuși, intr-o aplicație bazată pe microservicii vor apărea costuri asociate datorită faptului ca creearea unui nou microserviciu face necesară rescrierea a mult cod.  
 Un alt avantaj tehnic al microservicilor este faptul că livrarea continuă este in special ușor de realizat atunci când folosim microservicii. Dezvoltarea continuă permite ca software-ul să fie adus in producție regulat mulțumită unui proces ce poate fi reprodus.  
 Microservicile oferă o interfață la care sunt disponibile in rețea in funcție de tipul acestora. Fiecare microserviciu poate rula pe unul sau mai multe servere. Atunci cand serviciul reulează pe mai multe servere, sarcinile trebuie distribuite între instanțe.   
 În plus, servicile pot fi instalate la locații diferite in rețea pentru a fi mai aproape de anumiți apelanți. În cazul unui mediu distribuit la nivel global acest aspect este important de luat în calcul. Atunci când o infrastrucutră bazată pe microservicii folosește mai multe centre drept servere și procesează cererile mereu la cel mai apropiat centru timpul necesar pentru a răspunde poate fi redus semnificativ.  
 Este important de precizat faptul că deși microservicile reprezintă o soluție buna de a scala, natura lor distribuită aduce costuri de performanță in cea ce privește comunicarea distribuită.  
 Atunci când vine vorba de robustitate, sistemele distribuite nu performează la fel de bine ca cele nedistribuite. Arhitectura bazată pe microservicii se încadrează la acest capitol deci mereu există riscul apariției unor erori din cauza unei rețele ce devine indisponibilă sau a unor defecțiuni de hardware pe unul din servere.  
 Pentru a putea garanta că sistemul este robust este nevoie ca arhitectura sa fie proiectata corect. Comunicarea între microservicii trebuie construită in așa fel încat dacă unul din servicii devine indisponibil eroarea să nu se propage în tot sistemul. Astfel o eroare în unul din microservicii nu determina indisponibilitatea întregului sistem. Un mod de a trata această situație este de a seta timpi impliciți pentru microservicii să raspundă.  
 Un avantaj pe care îl au microserviciile față de aplicațile monolitice este faptul ca miroservicile descompun servicile în mai multe procese. Aceste procese sunt mai bine izolate unul fața de altul. O aplicație monolitică ce pornește un singur proces poate deveni indisponibilă dintr-o problemă ca un memory leak și astfel întreg sistemul devine indisponibil. Natura distribuită a microservicilor împiedică acest lucru deoarece într-un astfel de scenariu un singur microserviciu ar eșua.   
 Microservicile oferă autonomie echipei de dezvoltare responsabilă de acel seriviciu. Aceștia pot alege ce tehnologii să folosească fară să fie constrânse de celelalte servicii atâta timp cat comunicarea dintre microservicii este posibilă.  
 Libertatea de a alege tehnologi diferite permite utilizarea tehnlogiilor potrivite in funcție de problemă. Un anumit limbaj de programare sau un framework poate fi folosit pentru a implementa un anumit microserviciu.

Libertatea de alegere înseamnă doar ca echipele de dezvoltare au această opțiune, ci nu ca este obligatoriu ca microservicii diferite să folosească tehnologii diferite. Se poate ca tehnologiile folosite să fie impuse pentru toate servicile dintr-un proiect. Comparativ cu aplicațile ce folosesc microservicii, aplicațile monolitice nu au această opțiune. În cazul aplicaților monolitice apar constrângeri atât pentru bibliotecile folosite cât si faptul ca poate fi folositâ o singura versiune a acesteia. Microservicile nu constrâng echipa de dezvoltare de la aceste opțiuni oferindu-le autonomie.  
 Deși microservicile sunt un șablon arhitectural acestea nu beneficează doar dezvoltarea de software. De asemenea, arhitectura influențeaza și comunicarea echipei de dezvoltare si oraganizarea acesteia.  
 Microservicile oferă un nivel mare de independență atunci când o singura echipă este responsabilă de dezvoltarea unui microserviciu. Faptul ca microservicile promovează independența între echipe exclude necesitatea de coordonare între acestea si marește eficiența.  
 De asemenea, echipa de dezvoltare a unui microserviciu are mai multe responsabilități. Aceasta trebuie să decidă ce arhitectură folosesc pentru serviciul pe care îl dezvoltă și din punct de vedere organizatoric ei trebuie să își asume responsabilitatea consecințelor acestui fapt. Aceasta abordare reprezină o schimbare de la forma tradițională de organizare în care care cateva persoane decid ce tehnologii si arhitectură o sa fie folosită.

[11]

**3. Deployment**

* **3.1 Service Discovery** O provocare ce vine odată cu alegerea unei arhitecturi bazate pe microservicii (sau orice altă arhitectură distribuită) este faptul că trebuie să găsim adresa fizică la care o maşină este localizată. Acest concept a apărut odată cu arhitecturile distribuite şi poartă numele de descoperirea serviciilor (în engleza “service discovery”).  
   Descoperirea serviciilor este un element cheie atunci când vorbim despre o arhitectură bazată pe microservicii. În primul rând, acesta oferă echipei de dezvoltare abilitatea de a scala numărul de instanţe de microservicii ce rulează. Locaţia fizică a microserviciului este ascunsă consumatorilor acestuia mulţumită conceptului de descoperire a microserviciilor. Deoarece consumatorii nu cunosc locaţia fizică a instanţelor serviciilor, noi instanţe pot fi adăugate sau oprite din bazinul de microservicii oricând.  
   Conceptul de service discovery este o unealtă foarte puternică ce ne permite să modificăm oricând numărul de instanţe ale microserviciilor ce rulează la un moment dat, fără să deranjăm consumatorii serviciilor. În cadrul unei abordări monolitice, echipa de dezvoltare era nevoită să îmbunătăţească hardware-ul serverelor pentru a mări performanţa, acest lucru poartă numele de scalare verticală (vertical scaling). O arhitectură distribuită, cum este arhitectura bazată pe microservicii, permite echipei de dezvoltare o scalare orizontală (horizontal scaling) ce presupune adăugarea mai multor microservicii. Realizarea acestui ultim tip de scalare este posibil mulţumită conceptului de descoperire a serviciilor.  
   Un alt beneficiu datorat descoperirii serviciilor este faptul că putem gestiona serviciile ce nu mai sunt disponibile, acest serviciu fiind şters din lista de servicii disponibile iar motorul de descoperire a serviciilor nu o să mai ruteze serviciile către acel microserviciu. Acest fapt minimizează posibilele daune suferite de un microserviciu ce devine brusc indisponibil.  
   Mecanismul de descoperire a serviciilor trebuie să fie capabil să satisfacă următoarele calităţi:
  + Disponibilitate ridicată – în cazul unui cluster de servicii, trebuie să fim capabili să partajăm sarcina de căutare a serviciilor într-un cluster de descoperire a serviciilor chiar dacă unul dintre noduri devine indisponibil; în acest caz restul nodurilor ar trebui să fie capabile să partajeze sarcinile acestuia fără întreruperea sistemului;
  + Gestionarea indisponibilităţii: dacă un microserviciu devine indisponibil sau insuficient de “sănatos” pentru funcţionare, este responsabilitatea servicilui de descoperire să îl elimine din lista de servicii disponibile;
  + Flexibilitate în cazul degradării – serviciile de descoperire trebuie să facă “cache” local la informaţiile cu privire la locaţiile fizice ale servicilor descoperite pentru cazul în care acesta devine indisponibil; astfel locaţia microserviciilor nu este pierdută;
  + Load balanced – descoperirea serviciilor trebuie să împartă dinamic cantitatea de

cereri între toate serviciile de acel tip instanţiate;

Următoarele concepte generale trebuie să fie conţinute de către orice implementare de service discovery:

* înregistrarea serviciilor – cum se înregistrează microserviciile cu agentul de service discovery
* obţinerea adresei serviciilor de către clienţi – care sunt mijloacele prin care un serviciu obţine informaţii despre alte servicii
* partajarea informaţiilor – cum sunt partajate informaţiile serviciilor între noduri
* montiorizarea disponibilităţii – cum comunică serviciile nivelul lor de disponibilitate înapoi agentului de service discovery

În momentul în care instanțele de microservicii pornesc, acestea își vor înregistra adresa fizică, calea și portul la care pot fi accesate la unul sau mai multe instanțe de service discovery. Fiecare instanță a unui serviciu va avea adresa sa IP unică, dar se va inregistra cu ID-ul aceluiași serviciu. ID-ul unui serviciu este cheia cu care este unic identificat un grup de instanțe al aceluiași serviciu.  
 De obicei, un serviciu se va înregistra cu o singură instanță de service discovery. Cele mai multe implementări de service discovery folosesc un model peer-to-peer de propagare de date unde datele fiecărui serviciu sunt propagate tuturor celorlalte noduri din cluster.  
 Într-un final, fiecare instanță a unui serviciu o să trimită sau o să primească statusul acesteia de la serviciul de service discovery. Orice serviciu ce eșuează în a întoarce un status al sănătații pozitiv va fi şters din bazinul de instanțe de servicii disponibile.  
 Odată ce un serviciu a fost înregistrat de către un serviciu de service discovery, acesta este pregătit pentru a putea fi folosit de către aplicațiile ce au nevoie de acesta. Un client poate conta doar pe serviciul de service discovery pentru a obține locațiile serviciului de fiecare dată când acesta este apelat. Cu această abordare, serviciul de service discovery o să fie invocat de fiecare dată când un apel la un microserviciu înregistrat este facut. Această abordare este problematică deoarece serviciul client este complet dependent de serviciul de service discovery pentru a putea fi găsit şi invocat. [1]  
 O abordare mai robustă este folosirea unui load balacer pe client.  
 În acest model, când cineva vrea să invoce un serviciu:  
1. Consumatorul unui serviciu o să contacteze serviciul de service discovery și o să îi ceară informații despre locațiile tuturor instanțelor de care serviciul are nevoie şi o să facă cache local la date pe maşina consumatorului serviciului.  
2. De fiecare dată când un client vrea să folosească un serviciu, consumatorul serviciului o să se uite după informațiile despre locația serviciului în cache-ul local. Caching-ul local de pe client o să folosească un algoritm de load balancing ce se va asigura că apelurile la servicii de același tip sunt împărțite echitabil.  
3. Clientul o să apeleze serviciul de service discovery periodic pentru a-și actualiza cache-ul cu privire la instanțele serviciilor. Chiar dacă într-un final cache-ul clientului o să fie consistent, există riscul ca între momentul în care clientul contactează serviciul de service discovery pentru o actualizare, apelurile pot fi direcționate către o instanţă ce nu mai este disponibilă.  
 În cazul în care clientul apelează un serviciu ce a eșuat şi apelul eșuează, clientul își invalidează cache-ul local de service discovery şi acesta va încerca să îşi actualizeze cache-ul de la agentul de service discovery.

([1])

**3.2 Load balancing** Unul din marile avantaje ale unei arhitecturi bazate pe microservicii este faptul că fiecare microserviciu este dezvoltat independent față de celelalte. Un microserviciu poate avea mai multe instanțe ce partajează cererile.

În cazul în care microserviciile folosesc REST şi HTTP este necesară folosirea unui load balancer. Acesta are rolul de a distribui totalitatea cererilor în bazinul de instanțe al unui anumit microserviciu. În imaginea de mai jos se poate observa cum load balancer-ul distribuie cererile microserviciilor. [4]

Imagine preluată din [4]

Load balancer-ul poate elimina un serviciu atunci când acesta nu mai răspunde la cereri.  
 Un dezavanataj al folosirii unui load balancer este faptul că din moment ce tot traficul ce țintește microserviciile trebuie să treacă mai intâi prin load balancer, acesta poate limita performanța, viteza cu care cererile ajung la microservicii fiind limitată de load balancer.   
 Alt dezavantaj apare atunci când un load balancer devine indisponibil, microserviciile vor deveni de neatins.  
 În funcție de câte servicii servește, un load balancer poate fi:

* Load balancer central – acesta servește toate microserviciile şi nu este recomandat, în mare parte din cauza dificultății de reconfigurare atunci când apare un nou microserviciu;
* Load balancer la nivel de microserviciu - în această abordare este folosit un load balancer pentru fiecare microserviciu, acesta fiind responsabil doar de instanțele acelui microserviciu;

[4]

**3.3 API gateway** Un API gateway este un șablon de proiectare ce are ca scop oferirea unei soluții prin care să creăm un punct de intrare în aplicaţie pentru “lumea exterioară”. Acest serviciu este responsabil de rutarea cererilor (request-urilor). Față de alte șabloane, acesta oferă avantajul de a ușura clientului modul în care face cereri la server. Clientul face o singură cerere la un gateway API, un serviciu ce servește ca punct de intrare pentru cereri către API (Aplication Programming Interface). Acesta încapsulează arhitectura internă a aplicaţiei şi oferă clientului un singur punct de intrare.

Toate cererile clienților merg la API gateway, acesta fiind responsabil de rutarea cererilor către serverele corespunzătoare. Unele gateway-uri sunt responsabile şi de traducerea unor protocoale comune ca WebSockets sau Hypertext Transfer Protocol(HTTP) în protocoale interne folosite de servicii.

Cea mai importantă responsabilitate a unui API gateway este cea de a ruta cererile. Acesta folosește operatiile de rutare implementate pentru a conduce cererea clientului la serviciul destinat. Este comun ca acest lucru să se facă printr-un dicționar de rutare ce mapează o metoda HTTP la URL-ul HTTP al serviciului ce ar trebui să se ocupe de respectiva cerere.

Operațiile implementate de API gateway folosesc șablonul de proiectare “API Composition”. [3]



Imagine preluata din [3].

Acest șablon de proiectare este compus din:

* + un API ce este responsabil de compunerea rezultatului
  + mai multe servicii ce rezolvă o parte din cerere; acestea returnează o parte din rezultatele cererii, dar API composer-ul este responsabil de combinarea rezultatului şi returnarea acestuia apelantului;

În funcție de cum este implementat API, gateway-ul acesta poate oferi un singur API pentru toți utilizatorii. Un dezavantaj al acestei implementări este faptul că, în funcție de platforma clientului, aceștia pot avea cerințe diferite cu privire la datele returnate pentru cererea respectivă.

API gateway-urile pot fi implementate în așa fel încât să existe un API diferit pentru fiecare tipologie de client, astfel încât clienții unei aplicaţii mobile să folosească un API, clienții ce folosesc browser-ul să folosească alt API, etc.

Un posibil dezavantaj al folosirii unui gateway este faptul că dacă acesta nu este suficient de performant (lightweight), echipele de dezvoltare vor fi nevoite să aștepte pentru a actualiza API gateway-ul.

([3])

**3.4 Comunicarea între microservicii**

Datorită faptului că microserviciile rulează pe procese independente, comunicarea între acestea este distribuită şi se realizează prin intermediul rețelei. Acest lucru implică costuri de performanță mai mari decat un simplu apel în cadrul aceluiași proces. Serializarea şi deserializarea obiectelor este un proces costisitor.

De asemenea, atunci când comunicarea distribuită este implementată în cadrul serviciilor, apare riscul ca serviciul apelat să fie indisponibil, astfel serviciul apelant trebuie să trateze eroarea.

Cu cât microserviciile sunt de dimensiune mai mică, cantitatea de comunicare distribuită ce trebuie să fie efectuată crește şi cu cât acestea sunt mai mari apare mai puţină nevoie de a comunica distribuit [4].



**3.5 Containere  
Comparație între containere şi mașini virtuale:** Atât containerele cât şi mașinile virtuale pot fi folosite pentru a izola aplicațiile între ele atunci când rulăm pe aceeași mașină. Mașinile virtuale sunt o tehnologie mai matură comparativ cu containerele. Hypervisor oferă mașinilor virtuale un nivel în plus de izolare.  
 Imaginea alăturată ilustrează 3 aplicații ce rulează în mașini virtuale diferite. Hypervisor este necesar pentru a creea şi rula mașini virtuale. După cum se poate observa fiecare mașină virtuală are nevoie de o copie a sistemului de operare folosit, orice biblioteci de care aplicația are nevoie şi aplicației în sine. În cazul aplicațiilor Y şi Z se poate observa că, deși acestea împărtășesc biblioteci, este facută o copie pentru fiecare mașină virtuală, astfel Y este complet izolat de Z.  
 Daca am dori să rulăm aceleași 3 aplicații folosind containere am obține schema din partea stângă.  
 Spre deosebire de mașinile virtuale kernerul este partajat între containere. Acest fapt aduce constrângerea de a avea același kernel cu mașina gazdă. Aplicațiile Y şi Z folosesc aceleași biblioteci şi pot partaja datele comune, eliminând datele duplicate.  
 Din punct de vedere al performanței procesele ce rulează în containere sunt echivalente cu procesele native ale mașinii și nu pun problema unui overhead la fel de mare ca și în cazul mașinilor virtuale.

[5]



Imagine preluată din 5

Containerele reprezintă mecanisme moderne şi performante de deployment. Principiul de funcționare al containerelor este virtualizarea la nivelul sistemului de operare.  
 Din perspectiva unui proces ce ruleaza într-un container, mediul simulat este similar ca şi când acesta ruleaza în propria sa masina, izolat. [3]

Imagine preluata din [3]

Containerele pot fi personalizate în așa fel încât putem aloca resurse diferite pentru fiecare container. Atunci când folosim un orchestrator de containere este important să alocăm resursele containerelor deoarece acesta selectează mașina pe care să ruleze containerul asigurându-se că mașina aleasă este suficient de performantă.   
 Un container reprezintă o instanță a unei imagini. În timpul construirii (build-time) este folosită o unealtă ce are ca scop construirea de imagini. Aceasta citește codul, serviciul şi descrierea imaginii (pentru docker, aceasta este Dockerfile) pentru a creea imaginea containerului iar apoi o stochează într-un registru. În timpul rulării (runtime), imaginea containerului este luată din registru şi folosită pentru a instanția containere. [3]  
Imagine preluata din [3]

Volumele sunt unelte folosite pentru segmentarea și partajarea datelor ce au ciclul de viață independent de un container. Exemple de astfel de date reprezintă: datele bazelor de date față de sistemul de gestiune al bazei de date și fișierele de log față de aplicațiile web.  
Volumele creeaza modularitate pentru componentele arhitecturii. Această modularitate permite reutilizarea anumitor părți ale sistemului.   
 Un alt avantaj al volumelor este faptul ca acestea permit separarea între responsabilitățiile containerelor și a gazdei. Atunci când o imagine este încărcată pe mașina unei gazde și aceasta creaza un nou container, Docker nu știe ce fișiere sunt disponibile. Dar un utilizator ce are cunoștințe cu privire la localizarea fișierelor pe mașina gazdei poate să faca o mapare între directoarele containerului și fișierele stocate folosind volume. [12]

**4. Boardify  
4.2 Proiectare** Pentru acest proiect am ales să folosesc o arhitectură bazată pe microservicii, organizată în 6 module, aplicaţii Spring Boot, cu număr restrâns de responsabilități şi un modul pentru securitate.  
Cele 6 microservicii sunt:

* service-discovery - folosește Eureka pentru a descoperi serviciile
* gateway - folosește Netflix Zuul pentru rutarea microserviciilor
* auth – este responsabil doar de procesul de autentificare
* game – este responsabil de cererile cu privire la aflarea de informații despre jocuri
* group – este responsabil de gestionarea grupurilor de joc (creearea şi comunicarea între acestea)
* user – este responsabil de oferirea, modificarea datelor utilizatorului si înregistrare



Ultimul modul este common-security ce nu este un microserviciu ci reprezintă singura parte partajată a microserviciilor, cea cu privire la securitate.   
 Din punct de vedere al arhitecturii, microserviciile sunt identice:



Fiecare microserviciu este format din 6 directoare:

* model – definirea domeniului problemei
* dao (Data Access Object) – acest folder este format din abstracții ce vor fi folosite la nivelul service-ului pentru a adăuga, modifica, șterge sau obține date din baza de date;
* service – în acest folder sunt executate operații specifice logicii aplicaţiei asupra informației obținute din dao; de asemenea service-ul este folosit în controller
* dto (Data Transfer Object) – acest folder este compus din obiecte ce sunt folosite în cereri; DTO are ca scop returnarea ca răspuns a cererii a cât mai multor date deoarece mai multe cereri sunt mai costisitoare decât o singură cerere cu un răspuns mai amplu;
* controller – folosește metodele de la nivelul service-ului pentru a întoarce rezultatul/efectul dorit pentru o cerere;
* config – diverse configurări (configurări pentru securitate, comunicare între microservicii, etc);

De asemenea, fiecare microserviciu are un folder de resurse în care se pot găsi scripturi sql de creeare şi populare a bazei de date.



De asemenea avem 2 fișiere application.properties pentru momentul când dăm deploy aplicaţiei:



Profilul Maven pe care rulăm este important deoarece în cazul în care rulăm aplicaţia în development o să folosim baze de date locale, iar când rulăm în production o să dăm deploy la baze de date şi restul microserviciilor şi o să folosim containerul în care am făcut deploy la fiecare microservicu sau bază de date (fiecare microserviciu are propria sa bază de date).

4.3 Implementare

Model

 La nivel de model folosesc PJOs (Plain Java Objects), adică clase obișnuite Java, fără nici o restricție pentru a implementa domeniul problemei. De exemplu aceasta este implementarea clasei Game:

Pentru a reduce codul de rutină din clasele din Model, precum: gettere, settere, constructori am decis să folosesc Project Lombok. Această librărie pentru Java oferă adnotări pentru a realiza acest lucru: @Getter, @Setter (pentru a genera gettere şi settere pentru toate câmpurile claselor, @NoArgsConstructor şi @AllArgsConstructor pentru a genera constructor fără parametrii şi constructor cu toți parametrii, etc.

DAO (Data Access Object)  
 Interfața DAO folosită de service se numește GameDao. Clasa ce implementează această interfață este GameDaoJpa.  
 Maparea entității din cod şi a câmpurilor acesteia cu tabela din baza de date şi coloanele din aceasta este făcută în GamePersistance:



Deasupra clasei este folosită adnotarea @Table pentru a specifica tabela la care mapăm această clasă. Pentru fiecare câmp din această clasă avem o adnotare @Column în care este specificat numele coloanei din baza de date. De asemenea, este folosită adnotarea @Id pentru a numi primary key-ul din baza de date.

Odata ce am implementat GamePersistance, folosesc GameJpaRepository, o interfață ce extinde JpaRepository. Scopul acesteia este de a elimina codul repetitiv (boilerplate code) la nivel de DAO. Astfel, Spring Data JPA elimină în totalitate nevoia de a implementa operații CRUD (create, read, update, delete) şi reduce extrem de mult complexitatea acestui nivel din arhitectură.





**Service** La nivel de service avem interfața Service în care specificăm metodele la acest nivel.  
 Clasa care implementează Service se numește ServiceImpl.



ServiceImpl implementează metodele definite în Service. Pentru a face acest lucru este nevoie să injectez dependențele la GameDao, în acest exemplu am folosit injectarea prin câmp (field injection).

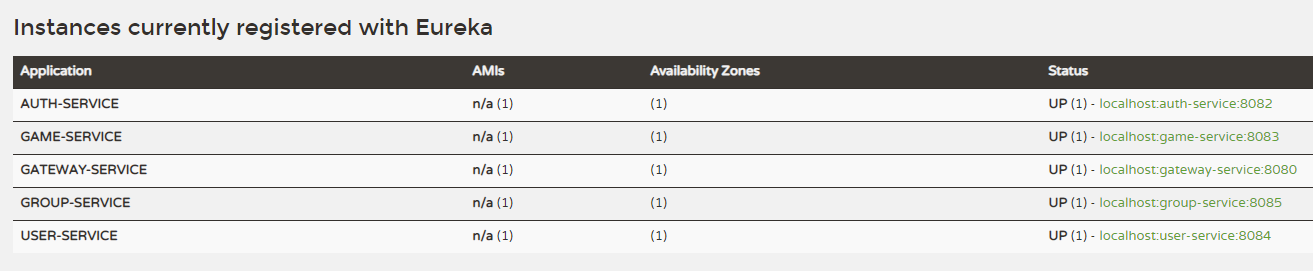
**Controller** La nivel de Controller implementez un Rest API. Fiecare microserviciu are altă mapare.  
 Pentru a face maparea unui microserviciu, în cazul de față doresc maparea la “/games” folosesc adnotarea @RequestMapping.  
 Pentru a putea folosi metodele ce implementează logica aplicaţiei din service, injectez dependențele la service:

Totodată o să folosesc log4j pentru a înregistra logg-urile.

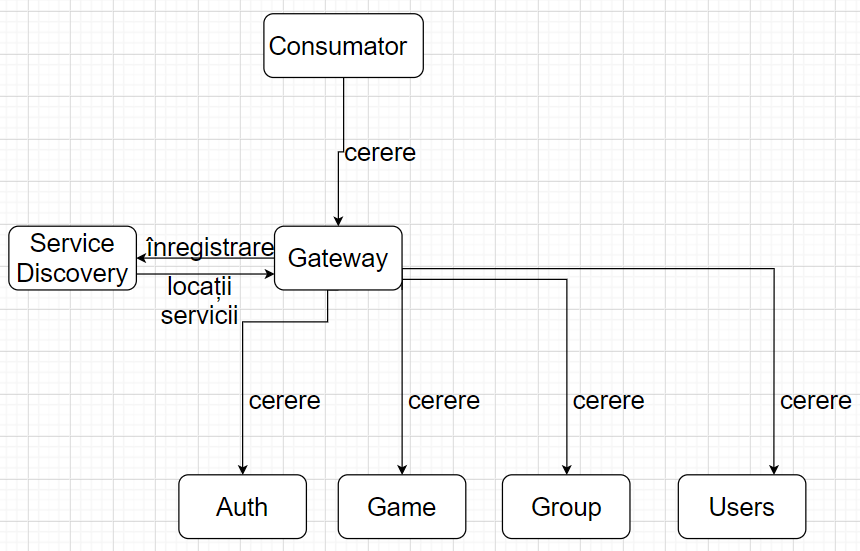
În următoarea imagine, exemplific implementarea şi maparea unei metode din rest api-ul microserviciului Game:



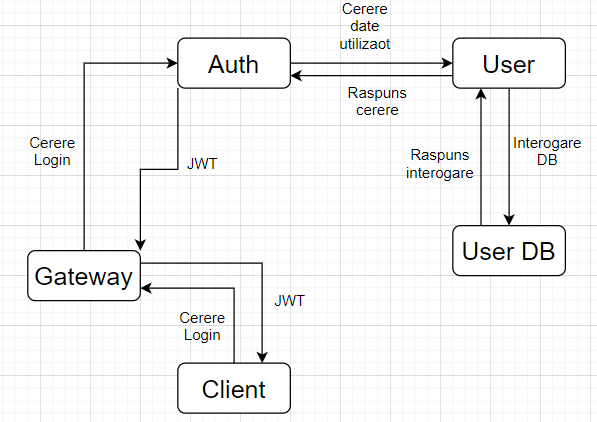
Folosesc adnotarea @RequestMapping pentru a mapa metoda la “”, dar dat fiind faptul că controller-ul este mapat la “/games”, această metodă se poate de fapt găsi la “/games”. Apoi specific că metoda din Rest API este de tip GET şi faptul că output-ul este un JSON (Javascript Object Notation).  
 În implementare se poate observa folosirea looger-ului dar şi apelurile folosind service-ul.

**4.2.1 Service discovery** Pentru a implementa serviciul de service discovery am ales să folosesc Eureka de la Netflix. Acest microserviciu are responsabilitatea de a înregistra toate serviciile și de a le elimina pe cele ce devin indisponibile. Acesta ascultă pe portul implicit al serverului Eureka, adică portul 8761.  
 Restul microserviciilor reprezintă clienți Eureka. Configurarea de bază pentru aceste microservicii este ceea ce îl specifică ca fiind client al serverului Eureka, dar clientul, pentru a ști unde să caute agentul de service discovery este nevoie să specificăm Default Zone-ul. Acesta reprezintă URL-ul la care clienții o să caute serverul. Pentru a specifica acest URL am decis să creez două fișiere de configurare diferite deoarece host-ul diferă între profilul de dezvoltare și cel de producție. În profilul de producție containerul Docker specific serviciului de service discovery trebuie pasat ca host pentru a fi găsit de clienții Eureka.  
 Dacă accesăm serviciul de service discovery pe portul 8761 de pe mașina ce îl găzduiește, vom vedea o pagină cu diverse informații, printre care si microserviciile ce au fost întregistrate:

**4.2.2 Gateway API**  
 Acest serviciu are responsabilitatea de a fi punctul de intrare a unei cereri în aplicație. Pentru acest lucru am ales să folosesc proxy-ul Netflix Zuul. Pentru a face rutarea, gateway-ul trebuie să cunoască locațiile tuturor microserviciilor pe care este nevoie să redirectioneze traficul. Pentru a le cunoaște, serviciul de gateway este la rândul său un serviciu ce se înregistrează la microserviciul de service discovery.  
 Pentru a specifica serviciile către care doresc să redirecționez cererile am creeat un fișier de configurare în care mapez url-ul serviciilor țintă.  
 Următoarea schemă exemplifică ce se întâmplă în cazul unei cereri ce trece prin gateway:

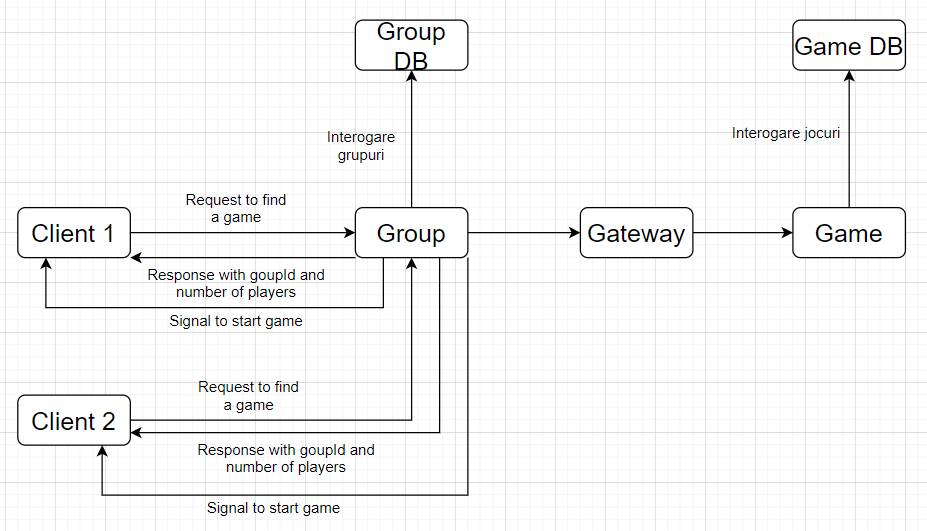


Fie că cererea este făcută de un alt serviciu sau de clientul de pe front-end, aceasta va trece prin serviciul de gateway mai întâi. Acest lucru ușurează folosirea serviciilor deoarece apelantul nu trebuie să cunoască locația serviciului ce îi procesează cererea. Tot ce trebuie să știe acesta este locația serviciului de gateway ce ascultă pe portul 8080. Odată ce cererea ajunge la gateway, folosind Netflix Zuul, aceasta va fi redirecționată către microserviciul destinat. Gateway-ul cunoaște adresele tuturor microserviciilor deoarece acesta este client la Service Discovery.

**4.2.3 Auth** Acest microserviciu are responsabilitatea de autentificarea utilizatorilor. Singura cerere ce este tratată în acest serviciu este accesarea de conturi existente. Ca măsură de securitate am ales să folosesc o autentificare bazată pe JWT (JSON Web Token).   
 Din motive de scalabilitate si pentru a păstra module slab cuplate (loosly coupled) am decis ca acest microserviciu sa nu aiba propria sa baza de date. Microserviciul User este singurul ce are acces la baza de date a utilizatorilor. Pentru a avea acces la utilizatori si a putea rezolva cererile de logare, microserviciul auth comunica cu cel de User.  
 Pentru a putea folosi celelalte microservicii (nu se aplică pentru service discovery, gateway și game) este nevoie ca în header-ul cererii să fie inclus token-ul de autentificare.  
 Pentru microserviciul de Game am ales ca momentan să anulez această măsură deoarece responsabilitatea lui e să ofere informații despre jocuri, date la care consider că nu este necesară restrangerea accesului.   
 Cererile la serviciile REST ale celorlalte microservicii pot fi făcute doar de către un utilizator logat.  
 Următoarea diagrama ilustrează modul in care o cerere de logare este rezolvată:

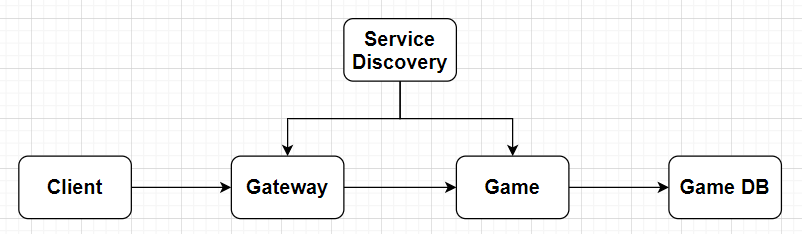
Clientul (în cazul meu browserul) emite o cerere de login pe portul 8080 al gateway-ul în care trimite credențialele utilizatorului. Cererea este trimisă mai departe la microserviciul de autentificare. Mai întâi serviciul Auth trimite o cerere la User pentru a obține datele necesare validării credențialelor. Odată ce primește datele de la User contul este validat si este generat un JWT Token pentru autentificare. Acest Token este trimis inapoi gateway-ului ce il trimite într-un final clientului.

**4.2.4 Group**  
 La nivelul acestui microserviciu am implementat un Web Socket necesar atât pentru a putea potrivi jucătorii ce vor să joace un anumit joc, cât și pentru a putea efecuta comunicarea dintre aceștia odată ce grupurile au fost create.  
 De asemenea, am implementat și un serviciu rest ce răspunde la diverse cereri ce oferă informații despre membrii grupurilor.  
 Responsabilitatea acestui microserviciu este de a se ocupa de gestiunea grupurilor (de a le potrivi si notifica potrivirea). Utiliatorii își aleg jocul preferat din lista de jocuri si apasă butonul start. În funcție de câți jucători este nevoie aceștia o sa fie puși într-o coadă de așteptare pentru a putea fi potriviți. Odată ce s-au strâns suficienți jucători pentru a face o potrivire utilizatorii vor fi redirecționați către o camera de chat unde pot comunica.  
 În diagrama de mai jos o sa explic ce se întămplă daca 2 jucători aleg sa joace un joc precum Splendor (acesta necesită 2 jucători pentru a putea începe):

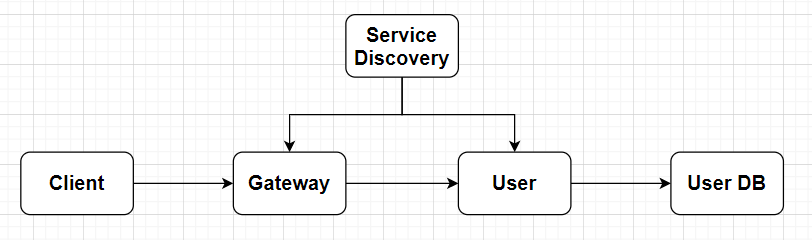


Consideram că Client 1 alege jocul și apasă butonul de start. Primul lucru care se întâmplă este handshake-ul cu serverul de WebSocket al microserviciului Group. Apoi, Client 1 trimite un mesaj în care specifica jocul ales. În acest punct Group va încera să îi gaseasca un grup de joc acestui client. Odată ce căutarea eșueaza (consideram ca nu există un grup pentru acest joc ce nu are suficienți jucători pentru a incepe) algoritmul realizeaza ca este necesară creearea unui nou grup. Pentru a putea determina faptul ca nu sunt suficienți jucători este nevoie ca microserviciul sa obțină numarul necesar de utilizatori pentru jocul respectiv. Pentru a face acest lucru trimite o cerere la portul 8080 al Gateway ce are grijă ca cererea sa ajungă la microserviciul de Games care interoghează baza de date pentru determinarea numărului de jucători necesari. Odată ce microserviciul Groups are aceasta informație își poate interoga propria baza de date pentru a verifica câți jucători sunt în coada pentru acest joc. Apoi, putem compara cele doua valori și să luam o decizie dacă este nevoie sa creeam un nou grup de joc sau daca clientul ce a trimis mesajul poate sa se alăture unui grup de joc deja existent. În exemplul de fată este necesă crearea unui nou grup de joc. Dupa ce este creat serverul de Websocker al microserviciului de Group o sa trimită înapoi clientului un mesaj în care specifică grupul în care a fost repartizat și câți utiliaztori s-au alăturat până la acel moment. Client 1 își actualizeaza interfața grafică pentru ca utilizatorul să știe ca a ajuns in lista de așteptare si este afișat numărul de jucători găsiți, număr ce este mereu actualizat.  
 Client 1 nu este potrivit deoarece mai este nevoie de un jucător. Odată ce Client 2 apasă butonul de start, un mesaj este trimis serverului de Websocket al microserviciului de Group și algoritmul incearcă să ii găseasca un grup. Deoarece deja există un grup deschis, algoritmul alătură pe Client 2 acestui grup iar apoi trimite un mesaj de broadcast tuturor membrilor din acel grup în care este semnalizat faptul ca un alt jucător s-a alăturat. Deoarece numărul necesar de jucători a fost atins, serverul de Websocket o să trimită tuturor membrilor grupurilor un mesaj prin care semnalează faptul ca grupul este complet si aceștia pot fi redirecționați către camera de chat.   
 Pentru a comunica cu microserviciul de Game, Group trebuie să trimită in header-ul cererii token-ul de autentificare. Acest token este primit de la Client si este propagat mai departe folosind un interceptor.

**4.2.5 Game**  
 Microserviciul Game oferă clienților informații referitoare la jocuri. Acest microserviciu este util pentru Group deoarece pentru a putea creea grupuri de joc este nevoie de informații precum numărul de jucători necesari.  
 Următoarea diagrama ilustrează cum ajung cererile de la Client la Game. Chiar dacă Game ascultă pe portul 8082, clientul face cererea pe portul 8080 si serviciul de Gateway se ocupă de rutare.



**4.2.6 User** Acest serviciu are rolul de a edita diverse informații despre useri, precum locația acestora si de a înregistra conturi noi.



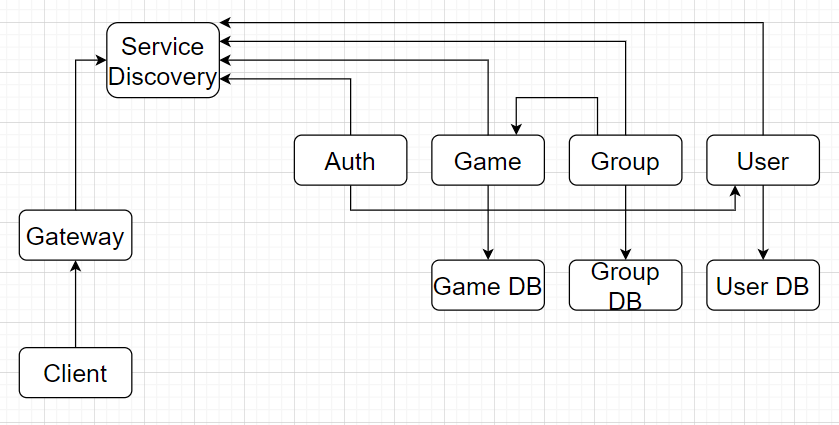
**4.2.7 Persistență** Deoarece folosesc o arhitectura bazată pe microservicii și doresc ca servicile să poată fi dezvoltate independent acestea trbuie să fie slab cuplate. Persistența datelor fiecărui microserviciu trebuie sa fie privată acelui serviciu si accesibila doar prin API-ul acestuia. Dacă două sau mai multe microservicii ar trebui sa impărtășeasca datele persistate atunci ar trebui ca modificarile schemei bazei de date ar trebui coordonate cu atenție, lucru ce ar încetini dezvoltarea.  
 Exisă cateva modalități prin care poți ține datele unui serviciu private. Nu este neapărata nevoie să oferi o bază de date separată pentru fiecare serviciu. În cazul in care dorești sa folosești o baza de date relaționala opțiunile sunt:

* tabele private la nivel de serviciu – fiecare serviciu deține un set de tabele ce trebuie accesat doar de catre acel serviciu
* scheme la nivel de serviciu – fiecare serviciu are o schema de bază de date ce este privată pentru acel serviciu
* o bază de date la nivel de serviciu – fiecare serviciu deține propria sa bază de date

Tabelele private la nivel de servicu și schemele la nivel de serviciu au cele mai mici costuri asociate. Pentru unele aplicații cu acces intensiv la bae de date are sens realizarea unei baze de date per serviciu.  
 Este o idee buna sa fie creeate bariere pentru a putea garanta aceasta modularitate. Acest lucru poate fi obținut prin atribuirea de conturi de utilizator diferite pentru fiecare bază de date, astfel accesul poate fi restricționat. Fară creearea unei astfel de bariere dezvoltatorii pot oricând ignora API-ul serviciului ce este responsabil de acele date și pot accesa direct datele.

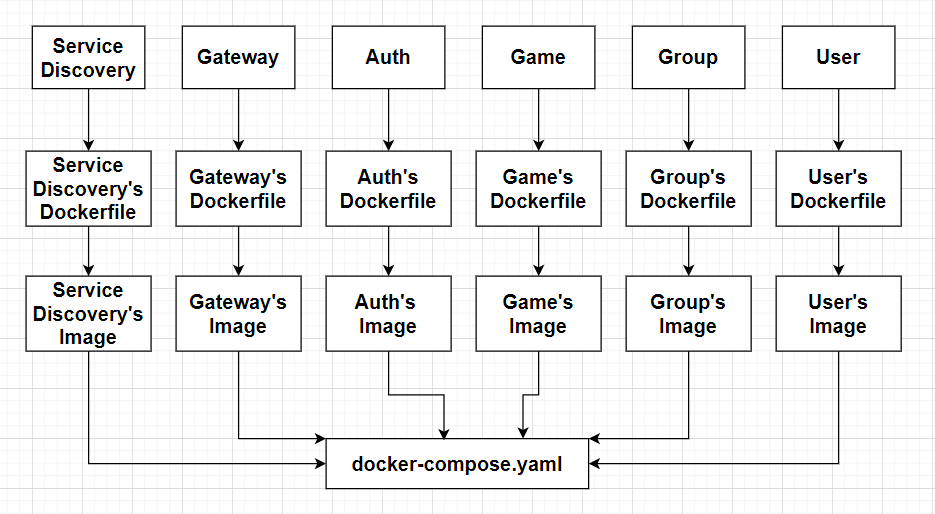
[6]

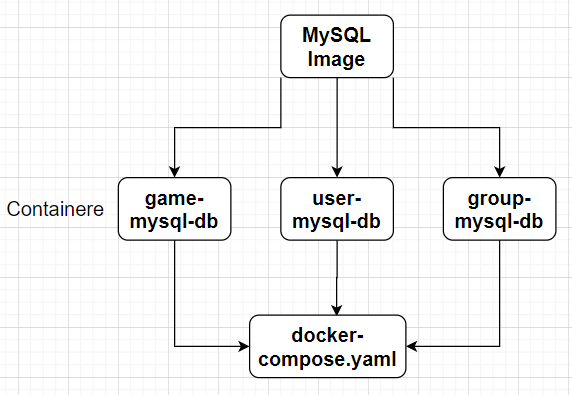
Din aceste trei opțiuni pentru persistența datelor am ales sa folosesc o bază de date la nivel de seriviciu. Pentru aceasta folosesc un sistem relațional de gestionare al bazelor de date, MySQL. Aplicația mea are nevoie de persistența pentru conturile utilizatorilor, jocurile de masă si informații despre grupurile de joc creeate.   
 Într-un stadiu incipit al proiectului am facut greșala de a face microserviciul de autenficare responsabil de baza de date a utilizatorilor. În momentul in care a aparut nevoia creeari unui alt microserviciu ce face editări cu privire la datele acestora am fost pus in situația de a lua una din urmatoarele 3 decizii: ambele microservicii de date sa aibă acces la baza de date, User sa fie proprietarul bazei de date sau Auth sa fie singurul cu acces la aceasta. Evident, ultima decizie ar fi încalcat principiul ca un microserviciu să aibe o singura responsabilitate. Prima opțiune de a permite ambelor microservicii acces direct la baza de date ar încetinii procesul de dezvoltare si ar promova o legatura strânsă intre module (tight coupling). Într-un final am decis sa refactorizez cele două microservicii și să fac modulul de User unic responsabil de accesul direct la baza de date a utilizatorilor. Acum, serviciul Auth cere datele de utilizator de la microserviciul de User pentru a putea trata cererile de autentificare.



Din diagrama de mai sus se poate observa ca la fiecare bază de date are acces un singur microserviciu. Microservicile ce au nevoia de a face interogări pe mai multe tabele din baze de date diferite nu o sa poată face acest lucru direct, ele vor comunica datele printr-un serviciu REST.

**4.4 Testare** Pentru a testa aplicația am decis să folosesc teste unitare. Fiecare microserviciu are propriul său pachet de test. Acesta este împărțit in 3 foldere: un folder pentru a testa DAO (Data Access Object), un folder pentru a testa service-ul si unul pentru controller.  
 La nivel de DAO verific dacă metodele testate adaugă, modifică sau returnează rezultatul corect din baza de date. Pentru a mă asigura ca rezultatul testelor este același de oricâte ori rulează testele la acest nivel al arhitecturii am ales sa fac rollback la querry-uri pentru ca testele sa nu afecteze baza de date.  
 La nivel de service, folosesc framework-ul Mockito pentru a face mock la DAO și a testa izolat service-ul.  
 Pentru Controller folosesc de asemenea Mockito, doar că de data asta fac mock la service și il injectez in Controller. De asemenea, Mockito este util in a testa serviciul REST din Controller.

**4.5 Deployment** Pentru deployment am ales să folosesc Docker.  
 Pentru a face deploy la un microserviciu ca şi container acesta trebuie să fie împachetat ca o imagine pentru acel container. O imagine pentru un container este o imagine de sistem formată din aplicaţie şi orice software necesar pentru rularea serviciului. În cazul meu, pentru a face deploy la un microserviciu bazat pe Spring Boot este nevoie să construiesc o imagine ce contine JAR-ul microserviciului şi versiunea corecta de JDK (Java Development Kit). [3]  
 Primul pas pentru a creea o imagine este creearea unui Dockerfile. Un Dockerfile descrie cum se construiește o imagine a unui container Docker. Acesta specifică o serie de instrucțiuni pentru a instala software, pentru a configura containerul şi comanda ce trebuie să fie executată când containterul este creat.  
 În cazul aplicaţiei mele creez o imagine pentru fiecare microserviciu, astfel fiecare modul are un fișier Dockerfile specific.  
 Pentru a automatiza procesul de pornire al containerelor am decis să folosesc Docker Compose. Docker Compose a fost creeat pentru a porni rapid mediul de dezvoltare Docker. Pentru a funcționa, folosește un fișier YML în care rețin configurările pentru un set de containere, astfel dezvoltatorii salvează timp şi evită potențiale erori cauzate de erori de scriere. Fără Docker Compose în mod normal ar trebui să creăm fiecare container în parte şi să îl configurăm. Acest proces ar trebui repetat de fiecare dată, dar folosind Docker Compose, o dată ce configurăm fișierul docker-compose.yml putem porni toate containerele cu o singură comandă. (docker-compose up)   
 Totodată, containerele corespunzătoare bazelor de date au fost de asemenea specificate folosind Docker Compose, acesta permițând specificarea dependențelor între containere. (bazele de date trebuie să pornească înainte de a putea fi folosite)  
 De asemenea, pentru a oferi conectivitate între microservicii a fost nevoie să pun toate containerele microserviciilor şi containerele bazelor de date în acelasi network.   
 Pentru a nu pierde datele bazelor de date folosesc volume pentru a stoca local informația ce altfel s-ar pierde odată cu distrugerea containerelor. Volumele sunt foarte ușor de folosit si facilitiază atât încarcarea informației in baza de date cât si stocarea acesteia.  
 În imaginea următoare am ilustrat cum fiecare microserviciu are propriul sau Dockerfile şi cum Docker Compose are nevoie de toate acestea pentru a putea instanția containerele. 

 Primul pas pentru a creea imaginea unui container este să pregătim JAR-uril, Dockerfile-urile tuturor servicilor și să instalăm fiecare serviciu folosind Maven. Rezultatul obținut sunt cele 6 imagini specifice servicilor. Apoi am specificat in docker-compose containerele ce urmează să fie construite pe baza acestor imagine.   
 Pentru a funcționa, servicile au nevoie de containerele bazelor de date folosite de microservicii. Pentru acest lucru am descărat de pe DockerHub o imagine pentru baze de date MySQL.

În imaginea de mai sus se poate observa cum imaginea MySQL este folosită pentru creearea a 3 containere pentru bazele de date. Aceste specificații de containere sunt adăugate si ele in docker-compose pentru simplitatea configurării si pornirii lor.

În acest moment docker-compose reține configurările tuturor containerelor necesare aplicației (containerele bazelor de date și microservicilor). Am creeat un network în care le-am adaugat pe toate pentru a putea comunica. Restul configuărilor se refera la creearea bazelor de date în containere, configurarea acestora precum și legarea containerelor microservicilor de cele ale bazelor de date.  
 Astfel deploy-ul devine repetabil fară nevoia de a repeta aceste configurări. Simpla rulare a fișierului docker-compose ar trebui să deployeze back-end ul pe orice mașină (ce are docker și software-ul necesar acestuia).

**4.6 Tehnologii folosite**

**Maven** Maven este o unealtă de construire automată folosită in mare parte pentru proiecte Java. Maven rezolvă două probleme ale construirii de software: cum este construit software-ul și care sunt dependențele acestuia. Un fișier XML descrie produs ul software ce urmează sa fie construit, dependențele acestuia cu privire la alte module externe si componente, ordinea de construit, directoarele si plug in-urile necesare. Acesta vine cu ținte predefinite pentru executarea de taskuri ca compilarea și împachetarea. Maven descarcă dinamic bibliotecile pentru Java și pluginurile de la unu sau mai multe repository-uri și le stochează in cache-ul local. [10]

**Spring** Spring a devenit un framework foarte popular pentru construirea aplicaţilor Java. Spring este bazat pe conceptul de injectarea dependențelor. Într-o aplicaţie Java, aplicaţia este descompusă în clase ce au legături una față de alta. Legăturile sunt invocarea constructorului direct în cod. Odată ce codul este compilat, aceste legături nu mai pot fi schimbate.  
 Acest lucru este problematic într-un proiect mare pentru că aceste legături externe sunt fragile şi o modificare poate impacta restul codului. Un framework pentru injectarea dependențelor, ca şi Spring, ușurează gestionarea aplicaţiilor mari externalizând relația dintre

obiecte în aplicaţie prin convenții şi adnotări. Spring joacă rolul unui intermediar între diferitele clase Java din aplicaţie şi gestionează dependențele.

**Spring Boot** Spring Boot este un framemork open source bazat pe Java. Acesta este folosit pentru a creea microservicii. Spring Boot a fost dezvoltat de Pivotal Team și este folosit pentru a creea aplicații Spring stand-alone pregătite pentru producție. Avantajele folosirii acestui framework sunt:

* este ușor de înțeles si ușurează dezvoltarea de aplicații Spring
* reduce timpul necesar dezvoltării
* crește productivitatea

Sprin Boot a fost creeat pentru a satisface următoarele nevoi:

* oferirea unui mod ușor de a porni o aplicație
* rularea aplicației independent si reducerea timpului de dezvoltare
* dezvoltarea unei aplicații pregătite pentru producție într-un mod cat mai ușor
* evitarea de configurări XML complexe

[7]

**Eureka** Eureka este un server bazat pe REST, ce permite serviciilor să se înregistreze pentru ca alte servicii să poate cere locația lor în rețea. Aceasta permite fiecărui serviciu să înregistreze un URL în numele său. Alte servicii pot găsi URL-ul după numele serviciului. Apoi, folosind URL-ul, celelalte servicii pot trimite mesaje REST acestui serviciu.  
 Eureka permite copierea URL-ului pe mai multe servere şi cache-uri pe client. Acest lucru evită eventualele probleme atunci când un server Eureka devine indisponibil şi permite răspunsuri rapide la cereri. Schimbările datelor trebuie comunicate tuturor serverelor. Poate dura un timp până toate serverele eureka sunt actualizate. În acest timp, datele sunt inconsistente. Fiecare server are o versiune diferentă a datelor.  
 În plus, Eureka susține Amazon Web Services pentru că Netflix îl folosește în acest mediu, astfel aceasta poate fi combinată cu Amazon.  
 Eureka monitorizează serviciile înregistrate şi le elimină din listă pe cele ce nu sunt disponibile pe serverul Eureka.  
 Eureka este baza multor altor servicii ale Netflix Stack şi Spring Cloud. Pe langa Service Discovery, alte aspecte precum rutarea pot fi implementate. [4]  
 Implementarea unui server Eureka cu Spring Cloud se face folosind adnotarea @EnableEurekaServer pentru server-ul Eureka şi folosind adnotarea @EnableEurekaClient pentru clienți.

**Zuul** Zuul este un proxy ce transferă cererile ce sosesc către microserviciile destinate. Zuul este un proces Java separat. Către lumea exterioară, doar URL-ul este vizibil dar cererile sunt procesate de microservicii diferite. Acest lucru permite schimbarea sistemului microserviciilor şi în același timp oferind un URL clienților.   
 Atunci când facem o cerere este nevoie ca Zuul să cunoască microserviciul către care vrem să trimitem cererea, pentru a fi capabil să o redirecționeze. Fără configurări suplimentare, Eureka poate redirecționa o cerere cu un URL ce începe cu “/microservice” către microserviciul cu numele “microservice”. Acest lucru face ca numele intern al microserviciului să fie cunoscut mediului extern. Această rutare poate fi configurată diferit, filtrele oferite de Zuul pot schimba cererile pentru a implementa diferite aspecte ale sistemului. De exemplu, poate fi integrat cu Spring Cloud Secutiry pentru a pasa token-uri de autentificare microserviciilor. Astfel de filtre pot fi folosite pentru a redirecționa diferite cereri către anumite microservicii. Acest lucru face posibil transferul cererilor către servere ce pot investiga anumite erori. În plus, o parte a unei funcţionalități a unui microserviciu poate fi înlocuită de alt microserviciu.  
 Implementarea unui server Zuul proxy cu Spring Cloud este foarte ușoară. Adnotarea @EnableZuulProxy activează proxy-ul Zuul. [4]

**Docker**

Docker folosește tehnologia Linux pentru containere şi a extins-o prin imagini portabile şi o interfață user-friendly pentru a dezvolta o soluție completă pentru creearea şi distribuirea containerelor. Platforma Docker are două componente distincte: Docker Engine, ce este responsabil pentru creearea şi rularea containerelor şi Docker Hub, un serviciu cloud pentru distribuirea containerelor. Docker Hub oferă un număr foarte mare de containere publice.  
 Docker daemon este responsabil de crearea, rularea şi monitorizarea containerelor, dar şi de construirea şi stocarea de imagini.   
 Registrele Docker stochează şi distribuie imagini. Registrul implicit este Docker Hub, ce găzduiește mii de imagini publice. Docker daemon descarcă imaginile de la regiștrii ca răspuns la o cerere docker pull sau la o cerere docker run dacă instrucțiunea FROM din Dockerfile specifică o imagine ce nu este disponibilă local.   
 Una dintre cele mai populare cazuri de utilizare a containerelor sunt microserviciile. Performanța containerelor le face potrivite pentru o astfel de arhitectură. Comparativ cu mașinile virtuale (VM), containerele sunt mai mici şi deploy-ul este mai rapid, permițând arhitecturii bazate pe microservicii să folosească minimul de resurse pentru a gestiona schimbările necesare. [5]

**Mockito** Mockito este un framework open-source pentru Java folosit pentru testare. Framework-ul permite creeare de obiecte mock.

**React** React este o bibliotecă populara folosită pentru creearea de interfețe pentru utilizator. Aceasta a fost construită de Facebook pentru a rezolva unele din provocarile ce vin odată cu un website mare. React are unica responsabilitate de a randa date pe DOM (The Document Object Model), astfel apare necesitatea de a folosi biblioteci adiționale pentru gestionarea stării si pentru rutare. Un exemplu de bibliotecă folosită pentru gestionarea stării este Redux iar pentru rutare React Router.  
 Codul scris in React este format din entități numite componente. Compoenente pot fi randate folosit biblioteca React DOM.  
 În React exista două tipuri de componente: componente functionale și componente bazate pe clase. Acestea din urma au particularitatea ca rețin și starea.   
[8]

**Redux** Redux este o bibliotecă open-source pentru JavaScript folosită pentru gestionarea stării. Aceasta este adesea folosiă cu biblioteci precum React si Angular pentru constuirea de interfețe cu utilizatorul.  
 Redux este o librarie mică cu un API limitat proiectat sa fie un container ușor de prezis. Acesta funcționează într-o manieră similară unei funcții de reducere, un concept al programării funcționale.  
[9]

**Project Lombok** Lombok este un framework pentru Java menit sa reducă codul repetitiv ca și getterele, setterele și constructorii (cu toate argumentele, impliciți).

**4.7 Implementari existente**

**Tabeltopia**

**Tabeltopia** este o platformă digitală cu acces instant la sute de jocuri de masă. Platforma permite crearea unei camere de joc în care jucători se pot alătura şi odată ce numarul minim de jucători a fost atins aceștia pot incepe jocul.

Tabeltopia a fost conceput pentru a juca jocurile de masă online şi nu pentru a permite jucatorilor să se intalneasca în viața reală.

Tabeltopia are o lista cu camerele de joc active:



Jucatorul poate alege o camera de joc şi să se alature acesteia apasand butonul “Join”:



Tabeltopia permite mai multe tipuri de conturi. Versiunea gratis permite unui juctor să se alature simultan în doar 2 jocuri. De asemenea exista anumite jocuri premium în care doar jucatori ce au un cont Silver sau Gold pot intra.  
Pentru a avea un astfel de cont jucatorii trebuie să plateasca o subscriptie lunara:

În menu-ul “All games” un jucator poate vedea toate jocurile şi aplica diverse filtre pe acestea:



De asemenea poate alege un joc iar apoi să creeze o camera de joc pentru acesta:



**5. Concluzie si dezvoltări viitoare**

**Bibliografie**

[1].: John Carnell: Spring Microservices în Action, Manning, June 2017

[2].: Moises Macero: Learn Microservices with Spring Boot, Apress; 1st ed. edition (December 10, 2017)

[3].: Chris Richardson: Microservices Patterns With examples în Java, Manning Publications; 1st edition (November 19, 2018)

[4].: [Eberhard Wolff](https://www.amazon.com/Eberhard-Wolff/e/B004XWJS22/ref=dp_byline_cont_book_1): Flexible: Microservices: Flexible Software Architecture 1st Edition, Addison-Wesley Professional; 1 edition (October 22, 2016)  
[5].: Adrian Mouat: Using Docker, [O'Reilly Media, Inc, USA](https://www.libristo.ro/ro/editura/O%27Reilly%20Media,%20Inc,%20USA), 22.12.2015

[6].: <https://plainoldobjects.com/2015/09/02/does-each-microservice-really-need-its-own-database-2/>, accesat la data de 27.05.2020 ora 13:02

[7].: <https://www.tutorialspoint.com/spring_boot/spring_boot_introduction.htm>  
[8].: <https://en.wikipedia.org/wiki/React_(web_framework)>

[9].: <https://en.wikipedia.org/wiki/Redux_(JavaScript_library)>

[10].: <https://en.wikipedia.org/wiki/Apache_Maven>

[11].: Production ready microservices

[12].: Docker in action