**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
     1. Service discovery
     2. Gateway API

4.2.3 Auth

4.2.4 Group

4.2.5 Game

4.2.6 User

* 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.

**1.3 Implementari existente**

**Tabeltopia**

**Tabeltopia** este o platforma digitala cu acces instant la sute de jocuri de masă. Platforma permite să creezi o camera de joc în care alti jucatori se pot alatura şi odata ce numarul minim de jucatori a fost atins poti incepe jocul.

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

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:



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** Înainte ca conceptul de microservicii să evolueze, majoritatea aplicațiilor bazate pe web erau construite folosind un șablon arhitectural monolitic. Într-o arhitectură monolitică o aplicație este livrată ca un singur artefact software livrabil. UI-ul (interfața cu utilizatorul), business logic-ul şi logica accesului la baza de date sunt împachetate într-un singur artefact şi livrate pe serverul aplicației.  
 Deși o aplicație poate fi livrată ca o singură unitate de lucru, de cele mai multe ori există mai multe echipe de dezvoltare ce lucrează la aplicație. Fiecare echipă de dezvoltare are o anumită parte din aplicație pentru care este responsabilă. O problemă este faptul că pe parcursul dezvoltării aplicației monolitice dimensiunea şi complexitatea crește dar comunicarea şi costurile de coordonare între echipe nu scalează. Atunci când o echipă are nevoie să facă o modificare, întreaga aplicație trebuie să fie reconstruită, re-testată şi re-livrată.  
 Conceptul de arhitectură bazată pe microservicii 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. Un microserviciu este un sistem distribuit slab cuplat. Microserviciile permit descompunerea aplicațiilor mari în componente mai mici, ușor de gestionat, cu responsabilități restrânse. Microserviciile ajută să combatem problemele tradiționale ale complexității unei baze de cod mari, descompunând baza mare de cod în bucăți mai mici, bine definite. Conceptul cheie ce trebuie înțeles despre microservicii este descompunerea funcționalităților aplicației astfel încât acestea să fie complet independente una față de alta.

**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 lansate (deployed) 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;

**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 ca fiecare microserviciu este dezvoltat independent fata de celelalte. Un microserviciu poate avea mai multe instante ce partajeaza cererile.

În cazul în care microsericile folosesc REST şi HTTP este necesara folosirea unui load balancer. Acesta are rolul de a distribui totalitatea cererilor în bazinul de instante al unui anumit microserviciu. În imaginea de mai jos se poate observa cum load balancer-ul distribuite cererile microservicilor. [4]

Imagine preluata din [4]

Load balancer-ul poate elimina un serviciu atunci când acesta nu mai raspunde la cereri.  
 Un dezavanataj al folosirii unui load balancer este faptul ca din moment ce tot traficul ce tinteste microservicile trebuie să treaca mai intai prin load balancer, acesta poate limita performanta, viteza cu care cererile ajung la microservicii fiind limitata de load balancer.   
 Alt dezavantaj apare atunci când un load balancer devine indisponibil, microserviciile vor deveni de neatins.  
 În functie de cate servicii serveste un load balancer poate fi:

* Load balancer central – acesta serveste toate microserviciile şi nu este recomandat, în mare parte din cauza dificultatii de reconfigurare atunci când apare un nou microserviciu;
* Load balancer la nivel de microserviciu - în aceasta abordare este folosind un load balancer pentru fiecare microserviciu, acesta fiind responsabil doar de instantele acelui microserviciu;

[4]

**3.3 API gateway** Un API gateway este un sablon de proiectare ce are ca scop oferirea unei soluti prin care să creeam un punct de intrare în aplicaţie pentru “lumea exterioara”. Acest serviciu este responsabil de rutarea cererilor (request-urilor). Fata de alte sabloane, acesta ofera avantajul de a usura clientului modul în care face cereri la server. Clientul face o singura cerere la un API gateway, un serviciu ce deserveste ca punct de intrare pentru cereri catre API (Aplication Programming Interface). Acesta incapsuleaza arhitectura interna a aplicaţiei şi ofera clientului un singur punct de intrare.

Toate cererile clientilor merg la API gateway, acesta fiind responsabil de rutarea cererilor catre serverele corespunzatoare. 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 importanta responsabilitate a unui API gateway este cea de a ruta cererile. Acesta foloseste operatiile de rutare implementate pentru a conduce cererea clientului la serviciul destinat. Este comun ca acest lucru să se faca printr-un dictionar de rutare ce mapeaza o metoda HTTP la URL-ul HTTP al serviciului ce ar trebui să se ocupe de respectiva cerere.

Operatiile implementate de API gateway folosesc sablonul de proiectare “API Composition”. [3]



Imagine preluata din [3].

Acest sablon de proiectare este compus din:

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

În functie de cum este implementat API gateway-ul acesta poate oferii un singur API pentru toti utilizatorii. Un dezavantaj al acestei implementari este faptul ca în functie de platforma clientului acestia pot avea cerinte diferite cu privire la datele returnate pentru cererea respecitva.

API gateway-urile pot fi implementate în asa fel incat să existe un API diferit pentru fiecare tipologie de client, astfel incat clientii unei aplicaţii mobile să foloseasca un API, clientii ce folosesc browser-ul să foloseasca alt API, etc.

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

([3])

**3.4 Comunicarea între microservicii**

Datorita faptului ca microservicile ruleaza pe procese independente, comunicarea între acestea este distribuita şi se realizeaza prin intermediul retelei. Acest lucru implică costuri de performanta mai mari decat un simplu apel în cadrul aceluiasi proces. Serializarea şi deserializarea obiectelor este un proces costisitor.

De asemenea, atunci când comunicarea distribuita este implementata în cadrul servicilor apare riscul ca serviciul apelat să fie indisponibil, astfel serviciul apelant trebuie să trateze eroarea.

Cu cat microservicile sunt de dimensiune mai mica, cantitatea de comunicare distribuita ce trebuie să fie efectuata creste şi cu cat acestea sunt mai mari apare mai puţina nevoie de a comunica distribuit [4].



**3.5 Containere  
Comparatie între containere şi mașini virtuale:** Atât containerele cat şi mașinile virtuale pot fi folosite pentru a izola aplicațiile între ele atunci când rulăm pe aceași 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 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 ca 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 obtine schema din partea stangă.  
 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 reprezinta mecanisme moderne şi performante de deployment. Principiul de functionare 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 asa fel incat putem aloca resurse diferite pentru fiecare container. Atunci când folosim un orchestrator de containere este important să alocam resursele containerelor deoarece acesta selecteaza masina pe care să ruleze containerul asigurandu-se ca masina aleasa este suficient de performanta.   
 Un container reprezinta o instanta a unei imagini. În timpul construirii (build-time) este folosita o unealta ce are ca scop construirea de imagini. Aceasta citeste codul serviciul şi descrierea specificata imaginii (pentru docker, aceasta este Dockerfile) pentru a creea imaginea containerului iar apoi o stocheaza într-un registru. În timul rularii (runtime), imaginea containerului este luata din registru şi folosita pentru a instantia containere. [3]  
Imagine preluata din [3]

**4. Boardify  
4.2 Proiectare** Pentru acest proiect am ales să folosesc o arhitectura bazata pe microservicii, organizata în 6 module, aplicaţii Spring Boot, cu numar restrans de responsabilitati şi un modul pentru securitate.  
Cele 6 microservicii sunt:

* service-discovery - foloseste Eureka pentru a descoperi servicile
* gateway - foloseste Netflix Zuul pentru rutarea microservicilor
* auth – Este responsabil doar de gestionarea conturilor (Logare şi inregistrare)
* game – este responsabil de cererile cu privire la aflarea de informatii despre jocuri
* group – este responsabil de gestionarea grupurilor de joc (creearea şi comunicarea între acestea)
* user – este responsabil de oferirea şi modificarea datelor utilizatorului



Ultimul modul este common-security ce nu este un microserviciu ci reprezinta singura parte partajata a microservicilor, cea cu privire la securitate.   
 Din punct de vedere al arhitecturii, microservicile sunt identice:



Fiecare microserviciu este format din 6 directoare:

* model – definirea domeniului problemei
* dao (Data Acess Object) – acest folder este format din abstractii ce for fi folosite la nivelul service-ului pentru a adauga, modifica, sterge sau obtine date din baza de date;
* service – în acest folder sunt executate operatii specifice logicii aplicaţiei asupra informatiei obtinute din dao; de asemenea service-ul este folosit în controller
* dto (Data Transfer Object) – acest folder este compous din obiecte ce sunt folosite în cereri; DTO are ca scop returnarea ca raspuns a cererii a cat mai multe date deoarece mai multe cereri sunt mai costisitoare decat o singura cerere cu un raspuns mai amplu;
* controller – foloseste metodele de la nivelul service-ului pentru a intoarce rezultatul/efectul dorit pentru o cerere;
* config – diverse configurari (configurari pentru securitate, comunicare între microservicii, etc);

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



De asemenea avem 2 fisiere application.properties pentru momentul când dam deploy aplicaţiei:



Profilul Maven pe care rulam este important deoarece în cazul în care rulam aplicaţia în development o să folosim baze de date locale, iar când rulam în production o să dam deploy la baze de date şi restul microservicilor şi o să folosim containerul în care am facut deploy la fiecare microservicu sau baza de date ( fiecare microserviciu are propria sa baza de date).

4.3 Implementare

Model

 La nivel de model folosesc PJOs (Plain Java Objects), adica clase obisnuite java, fara nici o restrictie pentru a implementa domeniul problemei. De exemplu aceasta este implementarea clasei Game:

Pentru a reduce codul de rutina din clasele din Model, precum: gettere, settere, constructori am decis să folosesc Project Lombok. Aceasta librarie pentru Java ofera adnotari pentru a realiza acest lucru: @Getter, @Setter (pentru a genera gettere şi settere pentru toate campurile claselor, @NoArgsConstructor şi @AllArgsConstructor pentru a genera constructor fara parametri şi constructor cu toti parametrii, etc.

DAO (Data Access Object)  
 Interfata DAO folosita de service se numeste GameDao. Clasa ce implementeaza aceasta interfata este GameDaoJpa.  
 Maparea entitatii din cod şi a campurilor acesteia cu tabela din baza de date şi coloanele din aceasta este facuta în GamePersistance:



Deasupra clasei este folosita adnotarea @Table pentru a specifica tabela la care mapam aceasta clasa. Pentru fiecarui camp din aceasta clasa avem o adnotare @Column în care este specificat numele coloanei din baza de date. De asemenea, este folosita adnotarea @Id pentru a numi primary key-ul din baza de date.

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





**Service** La nivel de service avem interfata Service în care specificam metodele la acest nivel.  
 Clasa care implementeaza Service se numeste ServiceImpl.



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

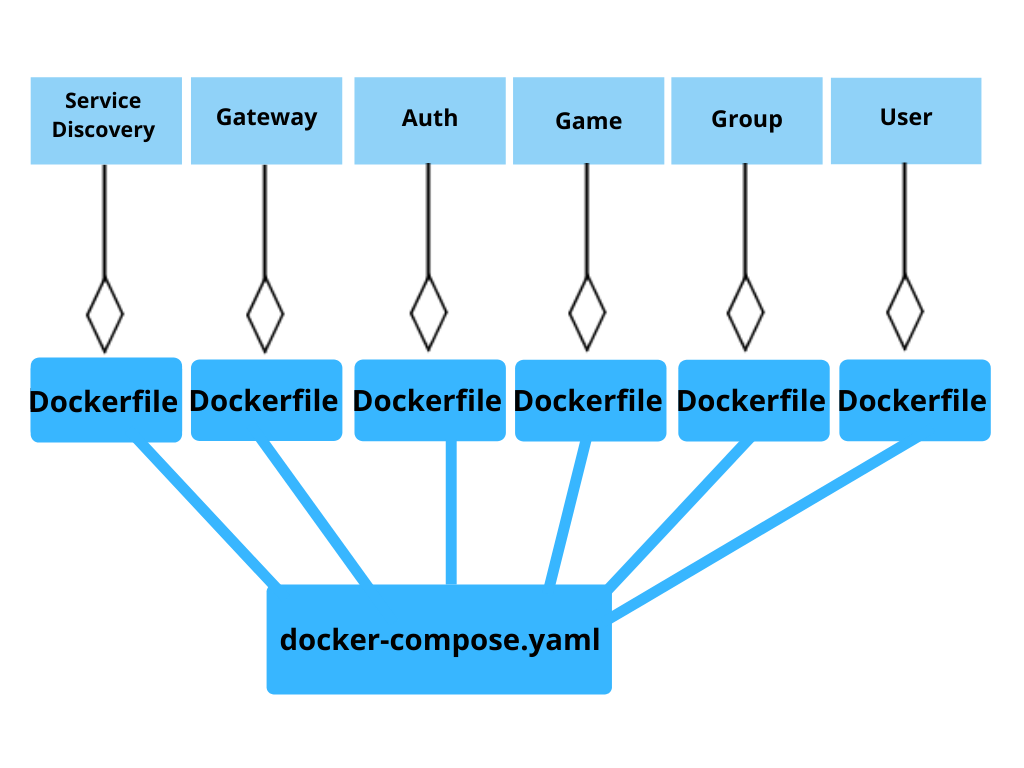
**Controller** La nivel de Controller implementez un Rest API. Fiecare microserviciu are alta mapare.  
 Pentru a face maparea unui microserviciu, în cazul de fata doresc maparea la “/games” folosesc adnotarea @RequestMapping.  
 Pentru a putea folosi metodele ce implementeaza logica aplicaţiei din service,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

Totodata o să folosesc log4j pentru a inregistra logg-urile.

În urmatoarea 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 ca controller-ul este mapat la “/games”, aceasta metoda se poate de fapt gasit la “/games”. Apoi specific ca metoda din Rest API este de tip GET şi faptul ca output-ul este un JSON (Javascript Object Notation).  
 În implementare se poate observa folosirea looger-ului dar şi apelurile folosind service-ul.

**4.5 Deployment** Pentru deployment am ales să folosesc Docker.  
 Pentru a face deploy la un microserviciu ca şi container acesta trebuie să fie impachetat ca o imagine pentru acel container. O imagine pentru un container este o imagine de sistem formata 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 construieste o imagine a unui container Docker. Acesta specifica o serie de instructiuni pentru a instala software, pentru a configura containeru şi comanda ce trebuie să fie executata când conainterul este creat.  
 În cazul aplicaţiei mele creez o imagine pentru fiecare microserviciu, astfel fiecare modul are un fisier 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 functiona foloseste un fisier YML în care stocheazi configurarile pentru un set de containere, astfel dezvoltatori salveaza timp şi evita potentiale erori cauzate de erori de scriere. Fara Docker Compose în mod normal ar trebui să creeam fiecare container în parte şi să il configuram. Acest proces ar trebui repetat de fiecare data dar folosind Docker Compose o data ce configuram fisierul docker-compose.yml putem porni toate containerele ruland comanda docker-compose up.  
 Totodata, containerele corespunzatoare bazelor de date au fost de asemenea specificate folosind Docker Compose, acesta permitand specificare dependentelor între containere. (bazele de date trebuie să porneasca inainte de a putea fi folosite)  
 De asemenea, pentru a oferi conectivitate între microservicii a fost nevoie să pun toate microservicile şi bazele de date în acelasi network.   
 În imaginea urmatoare am ilustrat cum fiecare microserviciu are propriul sau Dockerfile şi cum Docker Compose are nevoie de toate acestea (plus jar-urile microservicilor) pentru a putea instantia containerele.

**4.6 Tehnologii folosite**

**Spring** Spring a devenit un framework foarte popular pentru construitul aplicaţilor Java. La baza, Spring este bazat pe conceptul de injectarea dependentelor. Într-o aplicaţie Java, aplicaţia este descompusă în clase ce au legaturi una fata de alta. Legaturile sunt invocarea constructorului direct în cod. Odata ce codul este compilat aceste legaturi nu mai pot fi schimvate.  
 Acest lucru este problematic într-un proiect mare pentru ca aceste legaturi externe sunt fragile şi o modificare poate impacta restul codului. Un framework pentru injectarea dependentelor, ca şi Spring usureaza gestionarea aplicaţilor mari externalizand relatia dintre

obiecte în aplicaţie prin conventii şi adnotari. Spring joaca rolul unui intermediar între diferitele clase Java din aplicaţie şi gestioneaza dependentele.

**Spring Boot**

**Eureka** Eureka este un server bazat pe REST, ce permite servicilor să se inregistereze pentru ca alte servicii să poate cere locatia lor în retea. Aceasta permite fiecarui serviciu să inregistreze un URL în numele sau. Alte servicii pot gasi URL-ul dupa 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 evita eventualele probleme atunci când un server Eureka devine indisponibil şi permite raspunsuri rapide la cereri. Schimbarile datelor trebuie comunicate tuturor serverelor. Poate dura un timp pana toate serverele eureka sunt actualizare. În acest timp, datele sunt inconsistente. Fiecare server are o versiune diferenta a datelor.  
 În plus, Eureka sustine Amazon Web Services pentru ca Netflix il foloseste în acest mediu, astfel aceasta poate fi combinata cu Amazon.  
 Eureka monitorizeaza servicile inregistrate şi le elimina din lista pe cele ce nu sunt disponibile de serverul eureka.  
 Eureka este baza multor alte 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 clienti.

**Zuul** Zuul este un proxy ce transfera cererile ce sosesc catre microservicile destinate. Zuul este un proces Java separat. Catre lumea exterioara, doar URL-ul este vizibil dar cererile sunt procesate de microservicii diferite. Acest lucru permite schimbarea sistemului microservicilor şi în acelasi timp oferind un URL clientilor.   
 Atunci când facem o cerere este nevoie ca Zuul să cunoasca microserviciul catre care vrem să trimitem cererea, pentru a fi capabil să o redirecteze. Fara configurari suplimentare, Eureka poate redirecta o cerere cu un URL ce incepe cu “/microservice” catre microserviciul cu numele “microservice”. Acest lucru face ca numele intern al microserviciului să fie cunoscut mediului extern. Aceasta rutare poate fi configurata diferent, 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 microservicilor. Astfel de filtre pot fi folosite pentru a redirecta diferite cerei catre anumite microservicii. Acest lucru face posibil transferul cererilor catre servere ce pot investiga anumite erori. În plus, o parte a unei funcţionalitatii unui microserviciu poate fi inlocuita de alt microserviciu.  
 Implementarea unui server Zuul proxy cu Spring Cloud este foarte usoara. Adnotarea @EnableZuulProxy activeaza proxy-ul Zuul. [4]



**Docker**

Docker foloseste tehnologia Linux pentru conainere şi a extins-o prin imagini portabile şi o interfata user-friendly pentru a creea o solutie completa pentru creearea şi distribuirea containerelor. Platforma Docker are doua componente distincte: Docker Engine, ce este responsabil pentru creearea şi rularea containerelor şi Docker Hub, un serviciu cloud pentru distribuirea containerelor. Docker Hub ofera un numar foarte mare de containere publice.  
 Docker daemon este responsabil de creearea, rularea şi monitorizarea containerelor, dar şi de construirea şi stocarea de imaginii.   
 Registrele Docker stocheaza şi distribuie imagini. Registrul implicit este Docker Hub, ce gazduieste mii de imagini publice. Docker daemon descarca imaginile de la registrii ca raspuns la o cerere docker pull sau la o cerere docker run dacă instructiunea FROM din Dockerfile specifica o imagine ce nu este disponibila local.   
 Una din cele mai populare cazuri de utilizare ale containerelor sunt microservicile. Performanta containerelor le face potrivite pentru o astfel de arhitectura. Comparativ cu masinile virtuale (VM), containerele sunt mai mici şi deploy-ul este mai rapid, permitand ahitecturii bazate pe microservicii să foloseasca minimiul de resurse pentru a gestiona schimbarile necesare. [5]

**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