**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 intre 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 si dezvoltari viitoare

Bibliografie

1. **Introducere**

Odata cu dezvoltarea tehnologiei interactiunea dintre oameni scade si astfel apare o nevoie a unor aplicatii care sa creeze legaturi in viata reala intre acestia.

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

**1.1 Motivatie**

Inainte ca tehnologia sa joace un rol atat de important in viata noastra aceste jocuri se aflau in casa fiecaruia dintre noi. Astazi exista in fiecare oras localuri speciale pentru pasionatii de aceste jocuri.

**1.2 Provocari**

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

**1.3 Descriere**

O solutie prin care tehnologia ar putea rezolva problema lipsei de interactiune intre persoare ar fi creearea unei platforme unde pasionatii de board games pot fi potriviti. Acesta ar fi un punct de plecare urmand ca acestia sa se intalneasca intr-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 si alternative.

* Capitolul 3 surprinde procesul de deployment.
* Capitolul 4 reprezinta partea practica a aplicatiei si explica modul in care aceasta a fost implementata.
* Capitolul 5 isi propune sa sumarizeze punctele importante surprinse in lucrare si posibilele dezvoltari viitoare ale aplicatiei.

**1.3 Implementari existente**

**Tabeltopia**

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

Tabeltopia a fost conceput pentru a juca jocurile de masa online si nu pentru a permite jucatorilor sa se intalneasca in viata reala.

Tabeltopia are o lista cu camerele de joc active:



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



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

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



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



1. **Microservicii**

**2.1 Introducere**

Microserviciile sunt un șablon architectural pentru software ce devine foarte popular in zilele noastre mulțumită multor avantaje precum flexibilitatea si ușurința in scalare. Imparțindu-le in echipe mici intr-o firmă oferă multă eficiență in dezvoltare. Totuși, pentru a alege daca să folosești sau nu microservicii nu este suficient. Este nevoie să știi ce provocări ai să intâmpini și cum 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 si logica accesului la baza de date sunt împachetate într-un singur artefact si 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 si complexitatea crește dar comunicarea si costurile de coordonare între echipe nu scalează. Atunci când o echipă are nevoie să facă o modificare, intreaga aplicație trebuie să fie reconstruită, retestată si relivrată.  
 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 sunt complet independente una față de celelalte.

**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 si este deployed complet independent față de celelalte; microserviciile ar trebui sa aibă responsabiliteatea pentru o singură parte din domeniu; de asemenea, un microserviciu trebuie să fie reutilizabil in cadrul mai multor aplicații
* comunicarea microserviciilor se bazează pe niște principii de bază si adoptă protocoale de comunicare ușoare (lightweight) ca HTTP si JSON (JavaScript Object Notation) pentru a schimba date între consumator si 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 si tehnologii;
* microserviciile prin natura lor: mici, independente si distribuite – oferă organizațiilor să aibe 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 si 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 in cloud dimensiunea lor mică face ușoară pornirea unui număr mare a 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 si 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;
* clienții vor livrări mai rapide: clienții nu mai vor să aștepte pentru lansări anuale ale unui pachet software; aceștia se asteaptă ca funcționalitațiile să fie descompuse astfel încât noile lansări sa poata fi realizate in cateva saptamani fara nevoia unei lansari a intregului produs software;
* performanta si scalabilitate: aplicatiile globle fac extrem de dificil de prezis cat de mult volum trebuie gestionat de o aplicatie si cand acesta va aparea; aplicatiile trebuie sa scaleze rapid peste multe servere cand nevoia apare sau dispare;
* clientii se asteapta ca aplicatiile lor sa fie mereu disponibile: esuarea intr-o parte a aplicatiei nu ar trebui sa determine esuarea intregii aplicatii;

Pentru a indeplinii aceste astepari si a construi aplicatii ce pot scala mult, ca dezvoltatori trebuie sa descompunem aplicatia in servicii mai mici ce pot fi dezvoltate si lansate(deployed) indepenendent unul fata de altul. Daca descompunem aplicatia in servicii mai mici si le mutam dintr-un singur artefact monolitic putem construi sisteme ce sunt:

flexibile – serviciile decuplate pot fi compuse si rearanjate pentru a oferi noi functionalitati; cu cat unitatea de cod pe care o functionalitate adaugata este mai mică, cu atat este mai putin complicat sa schimbam codul si dureaza mai putin sa testam si sa facem deploy;

* de incredere – decuplarea serviciilor unei aplicatii implica faptul ca atunci cand o degradare are loc intr-o parte a aplicatiei aceasta nu determina intreaga aplicatie sa esueze; esuarile pot fi descoperite si izolate intr-o parte aplicatiei fara a determina esuarea intregii aplicatii; acest lucru permite serviciului sa se degradeze dintr-o eroare irecuperabila;
* scalabilitate – servicile decuplate pot fi distribuite cu usurinta in cadrul mai multor servere, facand scalarea functionalitatilor ca atare; in cadrul unei arhitecturi monolitice unde toate logica intregii aplicatii este in acelasi modul, intreaga aplicatie trebuie sa scaleze chiar daca doar o mica parte aplicatiei o impune; scalare unor servici mai mici este mult mai eficienta din punct de vedere a costurilor;

**3. Deployment**

* **3.1 Service Discovery** O provocare ce vine odata cu alegerea unei arhitecturi bazate pe microservicii(sau orice alta arhitectura distribuita) este faptul ca trebuie sa gasim adresa fizica la care o masina este localizata. Acest concept a aparut odata cu arhitecturile distribuite si poarta numele de descoperirea servicilor (in engleza “service discovery”).  
   Descoperirea servicilor este un element cheie atunci cand vorbim de o arhitectura bazata pe microservicii. In primul rand, acesta ofera echipei de dezvoltare abilitatea de a scala numarul de instante de microservicii ce ruleaza. Locatia fizica a microserviciului este ascunsa consumatorilor acestuia multumita conceptuluui de descoperirea microservicilor. Deoarece consumatorii nu cunosc locatia fizica a instantelor servicilor, noi instante pot fi adaugate sau oprite din bazinul de microservicii oricand.  
   Conceptul de service discovery este o unealta foarte puternica ce ne permite sa modificam oricand numarul de instante ale microservicilor ce ruleaza la un moment dat fara sa deranjam consumatorii servicilor. In cadrul unei abordari monolitice, echipa de dezvoltare era nevoita sa imbunatateasca hardware-ul serverlor pentru a mari performanta, acest lucru poarta numele de scalare verticala (vertical scaling). O arhitectura distribuita, cum este arhitectura bazata pe microservicii permite echipei de dezvoltare o scalare orizontala (horizontal scaling) ce presupune adaugarea mai multor microservicii. Realizarea acestui ultim tip de scalare este posibil multumita conceptului de descoperire a servicilor.  
   Un alt beneficiu datorat descoperirii servicilor este faptul ca putem gestiona servicile ce nu mai sunt disponibile, acest serviciu fiind sters din lista de servicii disponibile iar motorul de descoperire a servicilor nu o sa mai ruteze servicile catre acel microserviciu. Acest fapt minimizeaza posibilele daune suferite de un microserviciu ce devine brusc indisponibil.  
   Mecanismul de descoperire a servicilor trebuie sa fie capabil sa satisfaca urmatoarele calitati:
  + Disponibilitate ridicata – in cazul unui cluster de servicii trebuie sa fim capabili sa partajam sarcina de cautare a servicilor intr-un cluster de descoperire a servicilor chiar daca unul din noduri devine indisponibil; in acest caz restul nodurilor ar trebui sa fie capabile sa partajeze sarcinile acestuia fara intreruperea sistemului;
  + Gestionarea indisponibilitatii: daca un microserviciu devine indisponibil sau insuficient de “sanatos” pentru functionare este responsabilitatea servicilui de descoperire sa il elimine din lista de servicii disponibile;
  + Flexibilitate in cazul degradarii – servicile de descoperire trebuie sa faca “cache” local la informatiile cu privire la locatile fizice ale servicilor descoperite pentru cazul in care acesta devine indisponibil; astfel locatia microservicilor nu este pierduta;
  + Load balanced – descoperirea servicilor trebuie sa imparta dinamic cantitatea de

cereri intre toate servicile de acel tip instantiate;

Urmatoarele concepte generale trebuie sa fie continute de catre orice implementare de service discovery:

* inregistrarea servicilor – cum se intregistreaza microservicile cu agentul de service discovery
* obtinerea adresei servicilor de catre clienti – care sunt mijloacele prin care un serviciu obtine informatii despre alte servicii
* partajarea informatiilor – cum sunt partajate informatiile servicilor intre noduri
* montiorizarea disponibilitaii – cum comunica servicile nivelul lor de disponibilitate inapoi agentului de service discovery

În momentul în care instanțele de microservicii 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 instanța 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 instanța a unui serviciu o sa trimită sau o să primeasca statusul acesteia de la serviciul de service discovery. Orice serviciu ce eșuează în a întoarce un status al sănațații pozitiv va fi sters 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 catre 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 sa 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 si invocat. [1]  
 O abordare mai robustă este folosirea unui load balacer pe client.  
 In acest model, când cineva vrea sa invoce un serviciu:  
1. Consumatorul unui serviciu o sa contacteze serviciul de service discovery și o sa ii ceara informații despre locațile tuturor instanțelor de care serviciul are nevoie si o sa facă cache local la date pe masina consumatorului serviciului.  
2. De fiecare dată când un client vrea sa folosească un serviciu, consumatorul serviciului o sa se uite după informațiile despre locația serviciului in cache-ul local. Caching-ul local de pe client o sa folosească un algoritm de load balancing ce se va asigura ca apelurile la servicii de același tip sunt imparțite echitabil.  
3. Clientul o sa apeleze serviciul de service discovery periodic pentru a își actualiza cache-ul cu privire la instanțele servicilor. Chiar daca într-un final cache-ul clientului o sa fie consistent există riscul ca între momentul în care clientul contactează servicul de service discovery pentru o actualizare, apelurile pot fi direcționate către o instantă ce nu mai este disponibila.  
 În cazul in care clientul apelează un serviciu ce a eșuat si apelul eșueaza, clientul își invalideză cache-ul local de service discovery si clientul va incerca sa isi 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.

In cazul in care microsericile folosesc REST si HTTP este necesara folosirea unui load balancer. Acesta are rolul de a distribui totalitatea cererilor in bazinul de instante al unui anumit microserviciu. In 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 cand 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 sa 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 cand un load balancer devine indisponibil, microserviciile vor deveni de neatins.  
 In functie de cate servicii serveste un load balancer poate fi:

* Load balancer central – acesta serveste toate microserviciile si nu este recomandat, in mare parte din cauza dificultatii de reconfigurare atunci cand apare un nou microserviciu;
* Load balancer la nivel de microserviciu - in 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 sa creeam un punct de intrare in aplicatie pentru “lumea exterioara”. Acest serviciu este responsabil de rutarea cererilor (request-urilor). Fata de alte sabloane, acesta ofera avantajul de a usura clientului modul in 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 aplicatiei si 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 si de traducerea unor protocoale comune ca WebSockets sau Hypertext Transfer Protocol(HTTP) in 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 sa se faca printr-un dictionar de rutare ce mapeaza o metoda HTTP la URL-ul HTTP al serviciului ce ar trebui sa 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 si returnarea acestuia apelantului;

In 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 in functie de platforma clientului acestia pot avea cerinte diferite cu privire la datele returnate pentru cererea respecitva.

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

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

([3])

**3.4 Comunicarea intre microservicii**

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

De asemenea, atunci cand comunicarea distribuita este implementata in cadrul servicilor apare riscul ca serviciul apelat sa fie indisponibil, astfel serviciul apelant trebuie sa trateze eroarea.

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



**3.5 Containere  
Comparatie între containere si mașini virtuale:** Atât containerele cat si 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 in plus de izolare.  
 Imaginea alăturată ilustrează 3 aplicații ce rulează în mașini virtuale diferite. Hypervisor este necesar pentru a creea si 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 si aplicației in sine. In cazul aplicațiilor Y si 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 sa 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 si Z folosesc aceleași biblioteci si pot partaja datele comune, eliminând datele duplicate.  
 Din punct de vedere al performanței procesele ce rulează in containere sunt echivalente cu procesele native ale mașinii și nu pun problema unui overhead la fel de mare ca și in cazul mașinilor virtuale.

[5]

Imagine preluată din 5

 Containerele reprezinta mecanisme moderne si performante de deployment. Principiul de functionare al containerelor este virtualizarea la nivelul sistemului de operare.  
 Din perspectiva unui proces ce ruleaza intr-un container, mediul simulat este similar ca si cand acesta ruleaza in propria sa masina, izolat. [3]

Imagine preluata din [3]

Containerele pot fi personalizate in asa fel incat putem aloca resurse diferite pentru fiecare container. Atunci cand folosim un orchestrator de containere este important sa alocam resursele containerelor deoarece acesta selecteaza masina pe care sa ruleze containerul asigurandu-se ca masina aleasa este suficient de performanta.   
 Un container reprezinta o instanta a unei imagini. In timpul construirii (build-time) este folosita o unealta ce are ca scop construirea de imagini. Aceasta citeste codul serviciul si descrierea specificata imaginii (pentru docker, aceasta este Dockerfile) pentru a creea imaginea containerului iar apoi o stocheaza intr-un registru. In timul rularii (runtime), imaginea containerului este luata din registru si folosita pentru a instantia containere. [3]  
Imagine preluata din [3]

**4. Boardify  
4.2 Proiectare** Pentru acest proiect am ales sa folosesc o arhitectura bazata pe microservicii, organizata in 6 module, aplicatii Spring Boot, cu numar restrans de responsabilitati si 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 si inregistrare)
* game – este responsabil de cererile cu privire la aflarea de informatii despre jocuri
* group – este responsabil de gestionarea grupurilor de joc (creearea si comunicarea intre acestea)
* user – este responsabil de oferirea si 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 – in acest folder sunt executate operatii specifice logicii aplicatiei asupra informatiei obtinute din dao; de asemenea service-ul este folosit in controller
* dto (Data Transfer Object) – acest folder este compous din obiecte ce sunt folosite in 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 intre microservicii, etc);

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



De asemenea avem 2 fisiere application.properties pentru momentul cand dam deploy aplicatiei:



Profilul Maven pe care rulam este important deoarece in cazul in care rulam aplicatia in development o sa folosim baze de date locale, iar cand rulam in production o sa dam deploy la baze de date si restul microservicilor si o sa folosim containerul in 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 sa folosesc Project Lombok. Aceasta librarie pentru Java ofera adnotari pentru a realiza acest lucru: @Getter, @Setter (pentru a genera gettere si settere pentru toate campurile claselor, @NoArgsConstructor si @AllArgsConstructor pentru a genera constructor fara parametri si 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 si a campurilor acesteia cu tabela din baza de date si coloanele din aceasta este facuta in 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 in 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 in totalitate nevoia de a implementa operatii CRUD(create, read, update, delete) si reduce extrem de mult complexitatea acestui nivel din arhitectura.





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



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

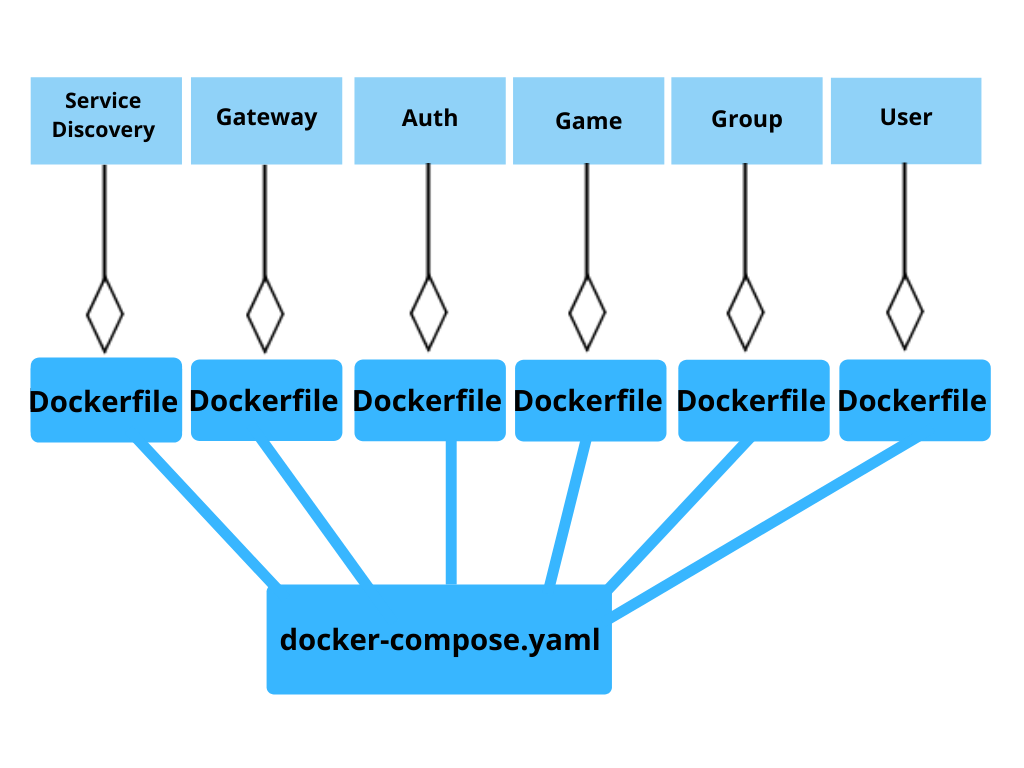
**Controller** La nivel de Controller implementez un Rest API. Fiecare microserviciu are alta mapare.  
 Pentru a face maparea unui microserviciu, in cazul de fata doresc maparea la “/games” folosesc adnotarea @RequestMapping.  
 Pentru a putea folosi metodele ce implementeaza logica aplicatiei din service,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

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

In urmatoarea imagine, exemplific implementarea si 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 si faptul ca output-ul este un JSON (Javascript Object Notation).  
 In implementare se poate observa folosirea looger-ului dar si apelurile folosind service-ul.

**4.5 Deployment** Pentru deployment am ales sa folosesc Docker.  
 Pentru a face deploy la un microserviciu ca si container acesta trebuie sa fie impachetat ca o imagine pentru acel container. O imagine pentru un container este o imagine de sistem formata din aplicatie si orice software necesar pentru rularea serviciului. In cazul meu, pentru a face deploy la un microserviciu bazat pe Spring Boot este nevoie sa construiesc o imagine ce contine JAR-ul microserviciului si 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 si comanda ce trebuie sa fie executata cand conainterul este creat.  
 In cazul aplicatiei 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 sa folosesc Docker Compose. Docker Compose a fost creeat pentru a porni rapid mediul de dezvoltare Docker. Pentru a functiona foloseste un fisier YML in care stocheazi configurarile pentru un set de containere, astfel dezvoltatori salveaza timp si evita potentiale erori cauzate de erori de scriere. Fara Docker Compose in mod normal ar trebui sa creeam fiecare container in parte si sa 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 intre containere. (bazele de date trebuie sa porneasca inainte de a putea fi folosite)  
 De asemenea, pentru a oferi conectivitate intre microservicii a fost nevoie sa pun toate microservicile si bazele de date in acelasi network.   
 In imaginea urmatoare am ilustrat cum fiecare microserviciu are propriul sau Dockerfile si 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 aplicatilor Java. La baza, Spring este bazat pe conceptul de injectarea dependentelor. Intr-o aplicatie Java, aplicatia este descompusa in clase ce au legaturi una fata de alta. Legaturile sunt invocarea constructorului direct in cod. Odata ce codul este compilat aceste legaturi nu mai pot fi schimvate.  
 Acest lucru este problematic intr-un proiect mare pentru ca aceste legaturi externe sunt fragile si o modificare poate impacta restul codului. Un framework pentru injectarea dependentelor, ca si Spring usureaza gestionarea aplicatilor mari externalizand relatia dintre

obiecte in aplicatie prin conventii si adnotari. Spring joaca rolul unui intermediar intre diferitele clase Java din aplicatie si gestioneaza dependentele.

**Spring Boot**

**Eureka** Eureka este un server bazat pe REST, ce permite servicilor sa se inregistereze pentru ca alte servicii sa poate cere locatia lor in retea. Aceasta permite fiecarui serviciu sa inregistreze un URL in 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 si cache-uri pe client. Acest lucru evita eventualele probleme atunci cand un server Eureka devine indisponibil si permite raspunsuri rapide la cereri. Schimbarile datelor trebuie comunicate tuturor serverelor. Poate dura un timp pana toate serverele eureka sunt actualizare. In acest timp, datele sunt inconsistente. Fiecare server are o versiune diferenta a datelor.  
 In plus, Eureka sustine Amazon Web Services pentru ca Netflix il foloseste in acest mediu, astfel aceasta poate fi combinata cu Amazon.  
 Eureka monitorizeaza servicile inregistrate si le elimina din lista pe cele ce nu sunt disponibile de serverul eureka.  
 Eureka este baza multor alte servicii ale Netflix Stack si 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 si 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 si in acelasi timp oferind un URL clientilor.   
 Atunci cand facem o cerere este nevoie ca Zuul sa cunoasca microserviciul catre care vrem sa trimitem cererea, pentru a fi capabil sa 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 sa 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. In plus, o parte a unei functionalitatii 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 si a extins-o prin imagini portabile si o interfata user-friendly pentru a creea o solutie completa pentru creearea si distribuirea containerelor. Platforma Docker are doua componente distincte: Docker Engine, ce este responsabil pentru creearea si rularea containerelor si 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 si monitorizarea containerelor, dar si de construirea si stocarea de imaginii.   
 Registrele Docker stocheaza si 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 daca 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 si deploy-ul este mai rapid, permitand ahitecturii bazate pe microservicii sa foloseasca minimiul de resurse pentru a gestiona schimbarile necesare. [5]

**Bibliografie**

[1].: John Carnell: Spring Microservices in 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 in 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