UNIVERSITATEA DIN BUCUREȘTI

FACULTATEA
DE
MATEMATICĂ ȘI INFORMATICĂ

SPECIALIZAREA INFORMATICĂ

LUCRARE DE DISERTAŢIE

EXECUŢIA WORKFLOW-URILOR DURABILE ÎN CLOUD

Absolvent

Moldovan George-Alexandru

Coordonator științific

Prof. dr. Letiţia Marin

București, septembrie 2022

Rezumat

Considerând trend-ul ascendent al arhitecturii orientate spre microservicii si migrarea de la vechiul mod de dezvoltare monolit, aceasta lucrare de disertaţie işi propune analiza soluţiilor prezente in piaţa în ceea ce priveşte managementul unei arhitecturi pe microservicii în cloud, şi analiza unor contribuţii personale aduse unui framework open-source de gestionare a workflow-urilor în cloud. Sistemele informatice preiau din complexitatea operaţiilor de zi cu zi, astfel că arborii de decizie trebuie interpretaţi şi gestionaţi în mod corect pentru a realiza un sistem care sa indeplinească cerinţele de piaţă curente. Pentru o lunga perioadă de timp, problema gestionării tranzacţiilor distribuite si management-ul workflow-urilor a fost inexistentă, deoarece într-un sistem monolit, caracteristicile tranzacţionale ale bazelor de date ce susţineau astfel de sisteme erau indeajuns pentru a avea toate garanţiile necesare astfel încât sistemul sa fie mereu lăsat într-o stare consistentă. Microserviciile şi arhitecturile distribuite in general, deşi vin cu o serie lunga de avantaje, poate cel mai greu de gestionat lucru este management-ul stării unei acţiuni, atunci cand aceasta se întinde pe mai multe microservicii, şi pe o durată lungă de timp.

Abstract

Considering the upward trend of microservices-oriented architecture and the migration from the old monolithic development mode, this dissertation aims to analyze the solutions present in the market in terms of managing a microservices architecture in the cloud, and the analysis of personal contributions to a open-source framework for managing cloud workflows. Information systems take over the complexity of day-to-day operations, so decision trees must be interpreted and managed correctly to achieve a system that meets current market requirements. For a long time, the problem of distributed transaction management and workflow management was non-existent, because in a monolithic system, the transactional characteristics of the databases that supported such systems were sufficient to have all the necessary guarantees so that the system should always be left in a consistent state. Distributed microservices and architectures in general, although they come with a long list of advantages, perhaps the most difficult thing to manage is the management of the state of an action, when it extends over several microservices, and over a long period of time.

Cuprins

Capitolul 1

Introducere

1.1 Motivatie

Unul din principiile de baza ale programării este reutilizarea. Motivaţia din spatele acestei lucrări o reprezintă dorinţa de o contribui la un framework care rezolvă o problemă generică, cu care se confruntă toţi dezvoltatorii care trebuie sa gestioneze tranzacţii distribuite. Analiza diferitelor metode pentru rezolvarea problemei de gestiune a workflow-urilor în cloud, în special într-o arhitectură ce se bazează pe funcţii în cloud a fost o prioritate pentru mine în ultimii ani.

Serverless, sau Functions-as-a-Service (FaaS), este o paradigmă din ce în ce mai populară pentru dezvoltarea de aplicații, deoarece oferă scalare infintă implicită și facturare bazată pe consum. Cu toate acestea, garanţiile slable de execuţie și suportul nativ pentru stocare a stării a FaaS creează provocări serioase atunci când se dezvoltă aplicații care necesită stare persistentă, garanţii de execuţie sau sincronizare. Acest lucru a motivat o nouă generație de soluţii serverless care oferă abstractizări ce stochează starea aplicaţiei. De exemplu, noua soluţie Azure Durable Functions (DF), o extindere peste deja existenta Azure Functions. Modelul îmbunătățește FaaS cu actori, fluxuri de lucru și secțiuni critice.

În acest context în care dezvoltatorii incearcă sa dezvolte aplicaţii ce gestiunează workflow-uri de lungă durată, cu toţii rezolvă aceeaşi problemă şi anume lipsa separării intre nivelul de execuţie si nivelul de stocare a soluţiilor existente FaaS. Cu toţii rezolvă o problemă generică, de salvare a stării în anumite puncte ale execuţiei, pentru a putea relua workflow-ul în eventualitatea în care agentul pe care rulează aplicaţie pică inainte de finalizarea workflow-ului. Acest lucru era foarte greu de realizat in arhitecturi serverless deoarece toate soluţiile existente până la apariţia Azure Durable Functions, suportau doar execuţii stateless în mod standard. Deci până la apariţia soluţiilor ce oferă garanţii de execuţie puternice in lumea Serverless, această tehnologie nu era o opţiune populară pentru dezvoltarea aplicaţiilor ce aveau nevoie de gestionare a stării şi era limitată la execuţia unor parţi mici ale aplicaţiilor pentru care starile intermediare nu erau importante.

1.2 Context

Serverless diferă de conceptele tradiționale de cloud computing în sensul că infrastructura și platformele în care serviciile rulează sunt ascunse clienților. În această abordare, clienții sunt preocupați doar de funcționalitatea dorită a aplicației lor, iar restul este delegată furnizorului de servicii.

Scopul serviciilor Serverless este triplu :

- Scuteşte dezvoltatorii de servicii cloud de la interacţiunea cu infrastructura sau diferite platforme
- Converteste modelul de facturare din cel clasic la cel bazat pe consum
- Scalarea automată a serviciului în funcție de cererea clienților.

Ca rezultat, într-o aplicație cu adevărat Serverless, infrastructura de execuție este ascunsă clientului, iar clientul plătește doar pentru resursele pe care le utilizează efectiv. Serviciul este conceput astfel încât să poată gestiona rapid creșterile de consum prin scalare automată. Entitățile de bază în calculul fără server sunt funcții. Clientul își înregistrează funcțiile în furnizorul de servicii. Apoi, acele funcții pot fi invocate fie de un eveniment, fie direct prin apelarea acestora la cererea utilizatorilor. Rezultatele execuției sunt trimise înapoi clientului. Invocarea funcțiilor este delegată unuia dintre nodurile de calcul disponibile în interiorul furnizorului de servicii. De obicei, aceste noduri sunt containere cloud, cum ar fi Docker [100] sau un mediu de rulare izolat [67].

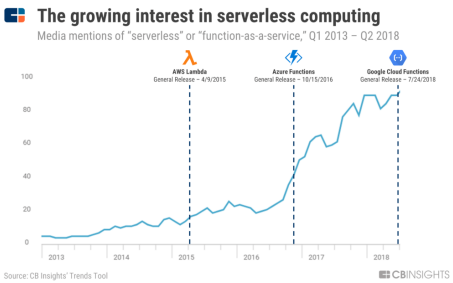

Figura 1.1: Statistică ce evidenţiază importanţa domeniului cloud computing în ultimii ani

Sursă: https://www.cbinsights.com

Deși conceptul de Serverless este relativ nou, acesta și-a deschis drumul în multe aplicații din lumea reală, de la instrumente de colaborare online la sisteme integrate (IoT), având o creştere în adopţie foarte rapidă, lucru vizibil şi în figura 1.1. Această creştere este datorată în mare parte usurinţei procesului de dezvoltare a aplicaţiilor Serverless şi beneficiile pe care le aduc din punct de vedere al scalării în mod automat, şi a gestionării complete a infrastructurii ce stă la baza aplicaţiilor.

Unul din lucrurile care poate duce adopţia tehnologiei serverless la un cu totul alt nivel, este tocmai capacitatea de a putea dezvolta aplicaţii intregi, ce se pot baza pe starea sistemului în cadrul execuţiei şi capacitatea de a gestiona long running workflows, o nevoie care este prezentă în mai toate aplicaţiile curente.

1.3 Alte analize ale problemelor curente a arhitecturii Serverless

Există mai multe provocări cu care se confruntă în prezent serviciile Serverless. Există unele sondaje și recenzii ale literaturii care discută aceste provocări [?, ?, ?, ?, ?].

Baldini et al. [?] enumeră o serie de probleme cu care se confruntă arhitectura serverless, printre care costul, care reprezintă un avantaj pentru aceasta arhitectura doar dacă se execută metode care nu sunt bazate pe prelucrare input-output. Altfel, este mai eficient din punct de vedere al costurilor folosirea soluţiilor clasice în cloud cum ar fi maşini virtuale rezervata sau containere. O altă problemă semnalată este cea a cold-start-ului care poate face o aplicaţie bazata pe funcţii serverless sa pară înceată daca pentru fiecare apel este necesar un cold start, din cauză ca traficul nu este indeajuns de mult pentru a împiedica funcţia sa scaleze la 0.

Rajkumar et al. [?] discută despre dificultăţile dezvoltării unei arhitecturi serverless din punct de vedere al limitărilor cu care vine această noua tehnologie si anume limitările de timp al executiei, de memorie al agentului si de management al stării. Acesta crede că este nevoie de o schimbare a mentalităţii de dezvoltare a aplicaţiilor pentru a benefecia la adevaratul său potenţial de tehnologiile Serverless, iar momentul în care aplicaţiile enterpise complete vor fi migrate sau dezvoltate complet pe o arhitectură Serverless este înca departe. Acesta vede această tehnologie ca pe o unealtă ajutătoare în dezvoltarea aplicaţiilor, dar pe viitor poate ajunge să fie nucleul aplicaţiilor.

Castro et al. [?] ridică problema dezvoltării de aplicaţii statefull folosind tehnologii Serverless in viitor, o temă ce in aceea perioadă era doar o idee, dar după cum urmează sa fie prezentat în aceasta lucrare, acum este o realitate. Alte probleme menţionate ar fi usurinţa cu care poate fi portată o aplicaţie legacy către o arhitectură Serverless, deoarece nu este de dorit sa se piardă toate acele ora valoroase care au fost deja investite în aplicaţiile existente.

Hassan et al. [?] discuta despre problema limitării la un singur provider atunci cand vine vorba de arhitecturi serverless, deoarece codul pentru un anumit provider de exemplu AWS Lambda, nu e portabil către alt provider, de exemplu Microsoft Azure Functions.

Jonas et al. [?] prezintă ineficienţele arhitecturii serverless atunci când vine vorba de procesarea operaţiilor care în mod normal ar beneficia de pe urma unui sistem cu mai multe nuclee, şi implicaţiile pe care aceasta limitare (2 nuclee per funcţie) o are atunci când vine vorba de paralelizarea acţiunilor pentru a îmbunăţăţii viteza. Impactul major în acest caz este creşterea semnificativă a datelor transmise pe reţea pentru a obţine acelasi grad de paralelism într-un sistem serverless versul unul clasic în cloud.

1.4 Conţinutul lucrării

Din punct de vedere al structurii, lucrarea va fi împărţită în 2 părţi :

- Partea teoretică în care va fi analizat Durable Task Framework şi cum funcţionează acesta
- Partea practică ce va compara aceaşi aplicaţie, dezvoltată în 2 moduri (clasic şi folosind DTF)

În prima parte a lucrării va fi analizată tehnologia ce stă la baza soluţiilor de management a workflow-uri în cloud si anume, Durable Task Framework. Vom analiza modul în care este separat domeniul de execuţie de domeniul de stocare, care sunt interfeţele pe care trebuie sa le respecte un provider, care sunt constrângerile care trebuie respectate atunci se foloseste această tehnologie si bineînteles care sunt beneficiile si dezavantajele sale.

În a 2a parte va fi analizat un exemplu clasic în literatura managementului de workflow-uri si anume cazul de rezervare multiplă în cazul unei călatorii. Această mini-aplicaţie a fost dezvoltată atât folosind metode clasice, cât şi folosind Durable Task Framework. Folosind aceste 2 abordari, vor fi analizate :

- Capaciţăţile fiecărui sistem si nivelul de rezilienţă împotriva dezastrelor pe care îl pot demonstra
- Diferenţele de dezvoltare între cele 2 abordări
- Analiză teoretică a costurilor între cele 2 arhitecturi
- Performanţa celor 2 sisteme

Capitolul 2

Analiza Tehnologiei Durable Task Framework si a providerilor disponibili

2.1 Analiză generală

Durable Task Framework (DTFx) este o bibliotecă care permite utilizatorilor să gestioneze workflow-uri persistente de lungă durată (denumite orchestrari) în C# folosind sintagme clasica de codare async/await. DTF stă la baza soluţiilor Microsoft de gestionare a workflow-urilor durabile în cloud, şi folosind această tehnologie, a fost expusă prima soluţie serverless ce permite gestionarea stării : Azure Durable Functions.

Din punct de vedere al arhitecturii framework-ului, acesta are 2 părti:

- DTF Core ce conţine nivelul de abstractizare al gestionării, şi care implementează conceptele de bază ce permit programarea orchestrărilor folosind sintagme async await
- DTF providers, care implementează interfeţele expuse de Core pentru a folosi diferite medii de stocare, în funcţie de nevoile dezvoltatorului.

La bază, framework-ul îşi propune să rezolve problema durabilităţii acţiunilor într-un sistem distribuit, în care este de aşteptat ca orice piesă a sistemului poate pica în orice moment. Arhitectura framework-ului e bazată pe evenimente, spre deosebire de alte framework-uri anterioare ce se bazau pe stocarea întregii stări a aplicaţiei la un moment dat.

Nevoia de garanţii de execuţie a facut ca DTF să construiască un sistem ce permite construcţia unei structuri arborescente ce reprezintă o maşină de stări finite, în care, frunzele arborelui (Activităţile, în terminologia DTF) au garanţia că vor fi executate cel puţin odata. Această garanţie vine şi cu o constrângere : Din cauză că Activităţile

nu sunt executate **exact o dată** este necesar ca toate acţiunile din cadrul unei Activităţi sa fie idempotente.

2.2 Folosirea evenimentelor în DTF

Modelul de arhitectură bazat pe evenimente este un model popular de arhitectură asincronă distribuită, utilizat pentru a produce aplicații foarte scalabile. De asemenea, este foarte adaptabil și poate fi utilizat pentru aplicații mici și, de asemenea, pentru cele mari și complexe. Arhitectura bazată pe evenimente este alcătuită din componente de procesare a evenimentelor extrem de decuplate, cu un singur scop, care primesc și procesează evenimentele în mod asincron.

Modelul de arhitectură bazat pe evenimente constă din două topologii principale, mediatorul și brokerul. Topologia mediatorului este folosită în mod obișnuit atunci când există nevoia de orchestrare a mai multor pași în cadrul unui eveniment printr-un mediator central, în timp ce topologia brokerului este utilizată atunci când doriți să înlănțuiți evenimente fără a utiliza un mediator central. Deoarece caracteristicile arhitecturii și strategiile de implementare diferă între aceste două topologii, este important ca ambele să fie cunoscute pentru a putea întelege motivaţie din spatele alegerii arhitecturii DTF.

Gestionarea workflow-urilor prin mediator central cunoscută si drept **Orchestrare** în literatura de specialiate (*Megargel et al.*[**?**]), constă în prezenţa unei entităţi centrale ce comunică cu toate celelalte sisteme ce iau parte la un anume workflow pentru a garanta execuţia cu success a unei maşini de stări. Topologia mediatorului este utilă pentru evenimentele care au mai mulți pași și necesită un anumit nivel de orchestrare pentru a procesa evenimentul. Aşa cum vom vedea şi în partea practică a acestei lucrări, sunt nenumărate cazuri ce se reduc la gestionarea unei maşini de stări. Un astfel de exemplu este un lant de aprovizionare, în care, de la momentul achizitiei pana la momentul livrării sunt numeroase etape ce trebuiesc îndeplinite cu success pentru ca operaţiunea ca un întreg sa fie un success. Printre acest etape s-ar numara pregătirea coletului, predarea lui către firma de curierat, înregistrarea plăţii şi în final, livrarea propriu zisă. Un astfel de sistem poate benefie de un mediator central, care asigură gestiunea stării unui astfel de workflow.

Gestionarea workflow-urilor printr-o topologie cu broker, cunoscută drept **Coreografie**, constă în distribuirea gestiunii maşinii de stări între toti consumatorii evenimentelor, astfel fiecare poate lua o anumită decizie în funcţie de contextul în care se află atunci când procesează un eveniment. În această topologie, nu există un mediator central, iar fiecare participant în workflow are responsabilitatea deciderii următoarei acţiuni, în momentul procesării unui eveniment. Astfel, acest gen de arhitectură este potrivită pentru worfklow-uri minimale, fără maşini de stări complicate, care nu au nevoie de un loc centralizat pentru asigurarea consistenţei stării. Altfel, folosind această arhitectură

în sisteme complexe, se poate ajunge foarte repede la haos, deoarece deciziile sunt împărtiţe peste tot, devenind imposibilă urmărirea firului logic al unei astfel de execuţii distribuite.

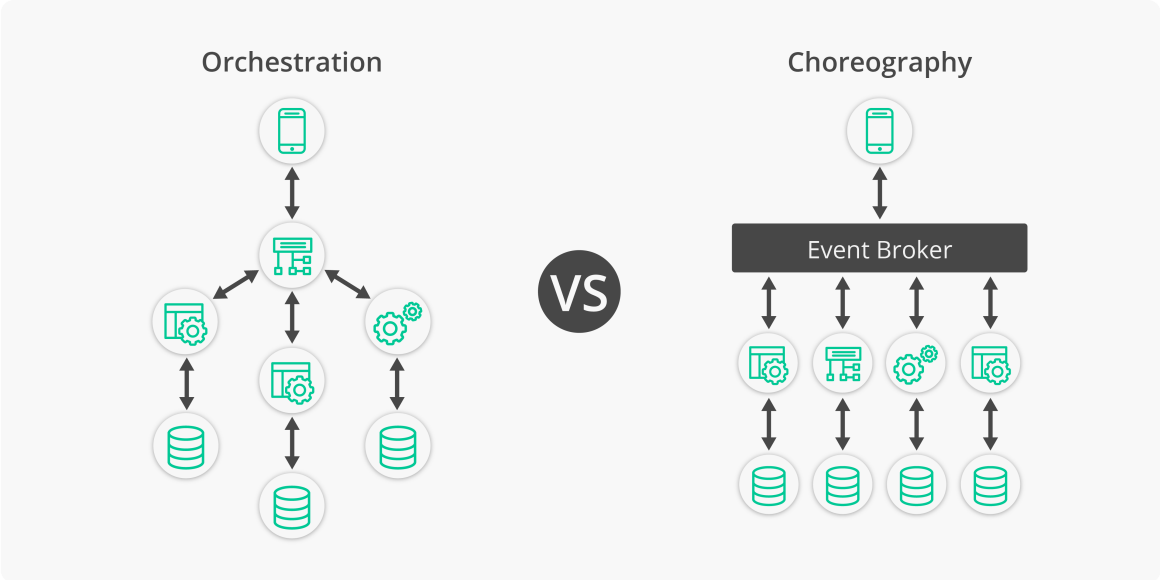

Figura 2.1: Orchestrare versus Coreografie

Sursă: https://solace.com/blog/microservices-choreography-vs-orchestration/

Durable task framework, implementează o arhitectură de tip mediator central, si expune bazele pentru a putea crea propriul mediator pentru gestiunea unui workflow de lungă durată. Pentru a analiza implementarea conceptului de orchestrare bazată pe evenimente, vom analiza implementarea de DTF împreuna cu Sql Server Provider, pentru a face o transpunere a conceptelor teoretice în lumea DTF. În 2.2 se poate observa structura schemei ce susţine framework-ul atunci cand este rulat împreuna cu un SQL Server Provider. Acesta implementează conceptele teoretice intr-un nivel de stocare bazat pe SQL server.

Există patru tipuri principale de componente de arhitectură în topologia mediatorului:

- cozi de evenimente
- mediatorul de evenimente
- canale de evenimente
- procesatoarele de evenimente.

Fluxul de evenimente începe cu un client care trimite un eveniment la o coadă de evenimente, care este utilizată pentru a transporta evenimentul la mediatorul de evenimente. Mediatorul de evenimente primește evenimentul inițial și orchestrează acel eveniment prin trimiterea de evenimente asincrone suplimentare către canalele de

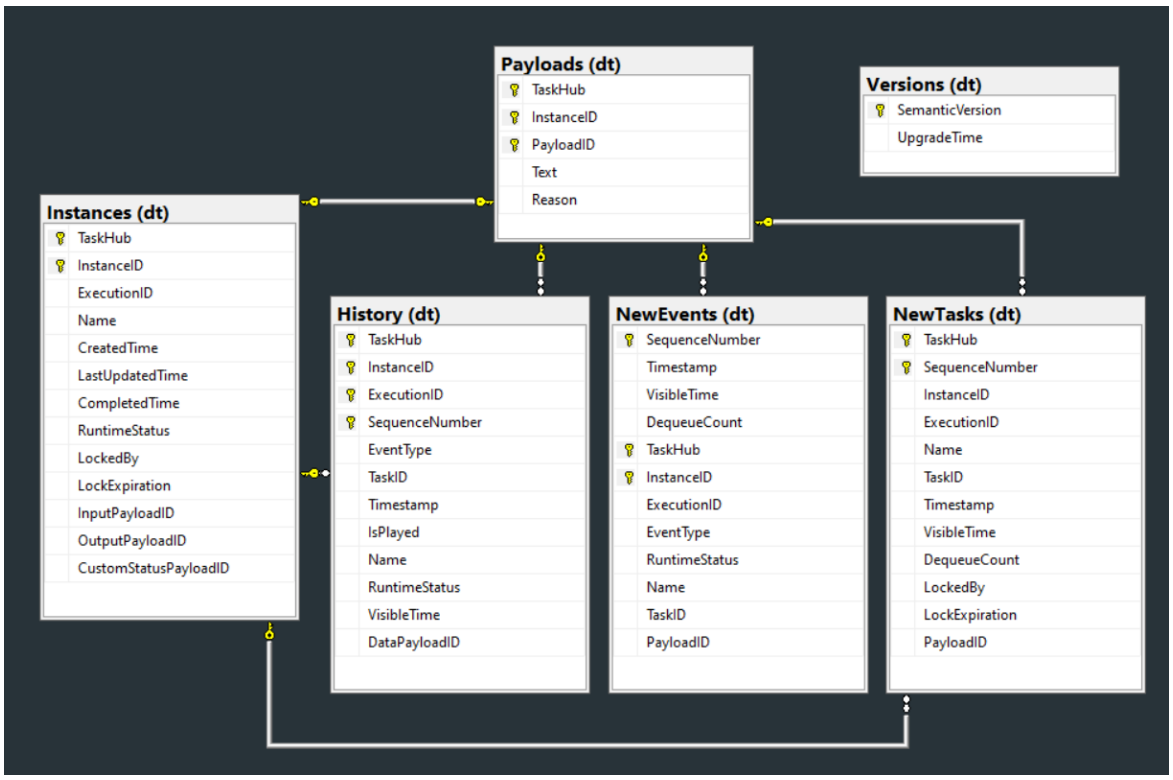

Figura 2.2: DTF Schema în DTF Sql Server provider

Sursă: https://microsoft.github.io/durabletask-mssql/

evenimente pentru a executa fiecare pas al procesului. Procesatoarele de evenimente, care ascultă pe canalele de evenimente, primesc evenimentul de la mediatorul de evenimente și execută o logică de afaceri specifică pentru a procesa evenimentul. Vom analiza fiecare componentă teoretică a procesului pentru a stabili parelela către structura din DTF, urmărind schema descride a 2.2.

În cazul nostru, coada de eveniment este tabela 'New Events' iar modalitatea prin care un eveniment ajunga în aceasta, este prin folosirea unui DTF Client pentru a inregistra un eveniment pentru o anumita orchestrare.

Mediatorul de evenimente este 'Orchestrarea' respectiv logica definită de dezvoltator pentru a gestiune workflow-ul. Această logica este rulată de fiecare dată când un eveniment este primit pentru un anume workflow. Deşi codul este rulat de mai multe ori, operaţiune denumita şi 'Replay', acţiunile care deja au mai fost executate odată nu sunt re-executate, ci rezultatele lor sunt direct încarcate direct din memorie, folosind istoricul orchestrării. Istoricul este salvat in tabela 'History'.

Canalele de evenimente sunt diferitele Task-uri înregistrate în table NewTasks. Acestea sunt folosite de pentru a declanşă execuţia unor Activităţi, ce reprezintă frunzele arborelui de executie a unui workflow. Task-urile sunt înregistrate pe canalele de evenimente de catre mediatorul de evenimente, pentru a fi mai apoi procesate de procesatorul de evenimente. La finalul procesării unui task din canalul de evenimente,

procesatorul de evenimente emite un nou eveniment pe coada de evenimente, pentru ca mediatorul să poate continua execuţia workflow-ului.

Procesatoarele de evenimente sunt Workerii de DTF, si reprezintă logica din DTF Core care monitorizează în mod constant canalele de evenimente pentru a prelua eventualele noi Task-uri ce sunt disponibile pentru execuţie. Astfel, se obţine o arhitectură foarte decuplată în care inregistrarea de evenimente, gestionarea stării workflow-ului, execuţia propriu zisă a diverselor taskuri şi întoarcerea răspunsurilor inapoi către mediator sunt toate operaţiune decuplate, independente una de cealaltă.

Cel mai important aspect ce reiese din analiza implementării conceptelor de orchestrare bazată pe evenimente, este că datorită operaţiilor independente, eşecurile care pot apărea atunci cand această logica este executată într-un mediu distribuit nu vor afecta starea workflow-ului ca un întreg, fiindcă eşecul într-o parte a arhitecturii nu afectează alte părţi iar fiecare unitate este independent reîncercabilă. Astfel revenirea în urma unui eşec (eroare de reţea, deconectare completă fie a workerilor fie a client-ului) sunt gestionate într-un mod graţios, fără efecte secundare.

2.3 DTF în Durable Functions

Conceptele prezentate în capitolul anterior, au fost reimpletate in DTF pentru a putea executa această tehnologie folosind soluţii native în cloud, şi pentru a expune o soluţie pentru care exista o mare nevoie în piaţă : Statefull Serverless functions. Astfel a fost dezvoltat un nou provider pentru DTF, bazat pe Azure Storage tables de această dată, pentru a expune aceeaşi functionalitate şi în cloud.

Logica de bază a framework-ului a ramas aceeaşi în schimb implementarea celor 4 concepte de bază intr-o arhitectură bazată pe evenimente (cozi de evenimente,mediatorul de evenimente,canale de evenimente, procesatoarele de evenimente) este schimbată.

Această adaptare a dus la lansarea unei soluţii noi pe piaţă ce a creat un cu totul nou domeniu în lumea serverless. Funcţii ce sunt capabile ce gestioneze starea, si să ruleze o perioada foarte îndelungată de timp, dar pentru care se plăteşte doar timpul de execuţie propriu zis. Acest lucru vine cu beneficii foarte mari din punct de vedere al costurilor, deoarece într-un workflow ce conţine şi paşi manuali, cum ar fi aprobarea de către un operator a unei trazacţii, pe perioada în care se asteaptă acest eveniment, codul orchestrării nu se execută, deci practic nu există un cost asociat acestei operaţii.

Pe lângă functionalităţile ce există în mod standard în toţi providerii de DTF, Durable Functions vine cu o caracteristică suplimentară numită Durable Entities. Durable Entities definesc operațiuni pentru citirea și actualizarea unor mici părți de stare, cunoscute sub numele de entități durabile. La fel ca funcțiile de orchestrator, funcțiile de entitate sunt funcții cu un tip de declanșator special, declanșatorul de entitate. Spre deosebire de funcțiile de orchestrator, funcțiile de entitate gestionează starea unei entități în mod explicit, mai degrabă decât să reprezinte implicit starea

prin fluxul de control. Entitățile oferă un mijloc de extindere a aplicațiilor prin distribuirea lucrării în mai multe entități, fiecare cu o stare de dimensiuni modeste. Acest lucru practic oferă suport automat pentru gestiunea unei stări independente de cea a orchestrării, dar într-un mod distribuit si sigur, deoarece oferă garanţii de access serializat l-a aceasta.

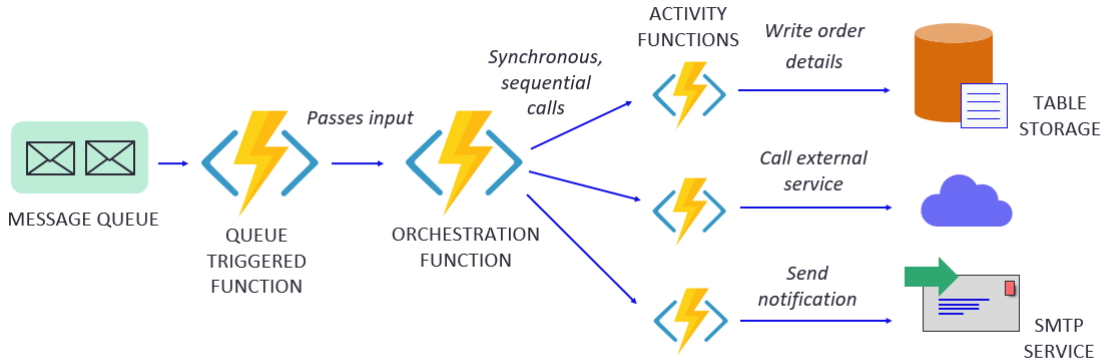

Figura 2.3: Exemplu arhitectură folosind Durable Functions

Sursă: https://thisiszone.medium.com

Aşa cum se poate observa şi în figura 2.3, conceptele clasice de DTF sunt transpuse în functii individuale, de diferite tipuri. Astfel, o Orchestrare este reprezentata de un Orchestration Function, o activitate este reprezentată de un Activity Function, în timp ce starea globală a workflow-ului este stocată în tabele în Azure Storage. Obţinem astfel o arhitectură nativă în cloud, fară costuri de mentenanţă a infrastructurii şi de scalare, având ca beneficiu principal faptul că gestionarea stării într-un mediu distribuit nu mai este responsabilitatea explicită a dezvoltatorului, ci este rezolvată implicit de către providerul de soluţii cloud.

2.4 Greutăţi în gestionarea workflow-urilor de lunga durată - Versionarea Orchestrarilor

Pe langă beneficiile pe care le are o arhitectură bazată pe un mediator pentru gestiunea workflow-urilor, această centralizare are şi anumite limitări si dezavantaje. Printre aceastea, poate cel mai dificil lucru de gestionat este versionarea acestora.

Natura îndelungată a acestor workflow-uri face acestea sa aibă urmatoarele particularităţi:

- Un workflow poate dura mai mult de un ciclu de dezvoltare (Mai multe versiuni ale aceluiaşi workflow pot exista până când o anumită instanţă a unei anume versiuni finalizează execuţia).

- Codul de orchestrare trebuie sa fie determinist deoarece acesta va fi executat de mai multe ori pe perioada de viaţă a unui workflow.

Prima problema se rezumă la dificultatea de versionarea a unei orchestrări. Atunci cand aparea o versiunea nou a maşinii de stări gestionată de o anume orchestrare, cea veche nu poate fi direct înlocuită, deoarece încă există instanţe ale acelui tip de workflow care înca rulează. Astfel, versiunile noi de orchestrare trebuie mereu rulate în paralel cu cele vechi, până când toate instanţele ce rulează versiunea veche a workflow-urilor finalizează execuţia. În acel moment versiunea veche a orchestrării poate fi stearsă.

Cea de doua problema este mai mult doar o limitare şi anume, din cauza logicii de replay multi pe perioada de viaţă a unei orchestrări, dezvoltatorul trebuie sa se asigura ca tot cod-ul din cadrul unei orchestrări, va returna acelaşi rezultat indiferent de momentul în care se rulează. Pe scurt, comportamentul unei orchestrări trebuie sa fie determinist si constant indiferent de timp.

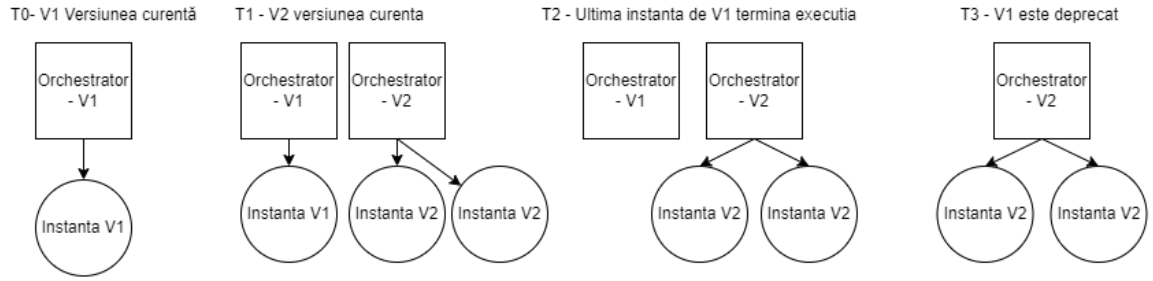

Figura 2.4: Ciclu de versionarea a codului de orchestrare

Capitolul 3

Workflow-uri de lungă durata - Analiză paralelă a arhitecturii

3.1 Descrierea problemei

Atunci cand vine vorba de managementul unui workflow de lunga durata, sau a unei tranzactii distribuite, un exemplu clasic in literatura este cel al unei rezervări de calatorie. O rezervare de călătorie presupune mai multe lucruri :

- Rezervarea unui taxi
- Rezervarea hotelului
- Rezervarea avionului

Într-o lume monolitică, în care totul ar fi deţinut de un singur serviciu al cărei arhitecturi s-ar fi bazat pe o baza de date relatională, garantarea rezervării cu success ar fi fost o problemă uşor de rezolvat avand în vedere caracteristica de atomicitate de unei tranzacţii. În schimb, într-o lume distribuită, în care fiecare serviciu este complet independent (Taxi Pelicanul, Hotel Capsa şi Blue Air) sistemul de gestiune a unei rezervări trebuie să fie rezilient la toate erorile ce pot apărea la diferite niveluri ale aplicaţiei. Astfel sistemul nostru trebuie să garanteze nu numai ca rezervarea se face cu success, dar în cazul în care aceasta nu se face, toate sistemele sa fie aduse la starea iniţială, de dinaintea începerii tranzacţiei distribuite.

Recuperarea dintr-un eşec într-o tranzacţie distribuită nu se poate face în mod nativ, deoarece aplicaţiile ce iau parte la aceasta sunt complet independent, astfel că fiecare sistem ce ia parte la o astfel de tranzacţie trebuie să expună si o funcţionalitate de roll-back ce va anula o operaţie ce a fost facută în aceasta. Astfel, putem concluziuna ca managementul unei tranzacţii distribuite rezulta intotdeauna în managementul unui workflow de tip SAGA.

Saga este un concept arhitectural teoretic ce garantează execuţia unui workfow într-un mod predictibil: Fie $A_1, A_2, ...A_n$ acţiuni ce iau parte la o execuţie Saga si

$RA_1, RA_2, ...RA_n$ acţiunile de rollback corespunzător fiecărui pas care ia parte la workflow. Un workflow de tip Saga garantează fie ca toţi paşii $A_1, A_2, ...A_n$ au fost executaţi cu success, fie $A_1, A_2, ...A_t, RA_1, RA_2, ...RA_t$au fost executaţi cu success. Astfel, se garantează fie ca toate acţiunile au fost executate cu success, fie că toate acţiunile ce au apucat să fie executate până la un punct de eroare au fost anulate cu succes. În final, sistemul va fi mereu într-o stare consistentă.

Scopul părţii practice a acestei lucrări este implementarea unui coordonator de tranzacţii distribuite de lungă durată implementând o arhitectură Saga. Vor fi analizate atât dezvoltarea acestei aplicaţii folosind un framework care facilitează dezvoltarea orchestratoarelor, cât şi dezvoltarea aplicaţiei folosind tehnologii clasice (.net Web API combinat cu o bază de date relaţională).

3.2 Implementarea arhitecturii în mod clasic

În mod clasic, implementarea unui serviciu ce oferă garanţiile unei arhitecturi SAGA, trebuie să fie rezistent la următoarele potenţiale probleme :

- posibilitatea ca procesul sa fie inchis chiar în timpul execuţiei
- posibilitatea ca unul din paşi să eşueze din motive neimputabile sistemelor ce iau parte la acţiune (probleme de reţea)

3.2.1 Implementarea naivă

Vom porni de la următoarea secţiune de cod :

```
1  var step = ''Started'';
2  try {
3          await sagaService.bookHotel(booking);
4          step = ''BookedHotel'';
5          await sagaService.bookTaxi(booking);
6          step = ''BookedTaxi'';
7          await sagaService.bookFlight(booking);
8          step = ''BookedFlight'';
9  } catch (Exception){
10         if (step ==  ''BookedHotel''){
11                 await sagaService.cancelHotel(booking);
12         } else if (step ==  ''BookedTaxi''){
13                 await sagaService.cancelHotel(booking);
14                 await sagaService.cancelTaxi(booking);
15         }
16 }
```

Într-o lume ideeală această secţiune de cod ar fi o implementare corectă a unui SAGA, dar ce se poate întampla, daca analizăm cazurile la care trebuie sa fie rezilient un astfel de sistem, într-o lume distribuită ?

Ei bine, analizând prima potenţială problema, şi anume posibilitatea ca procesul ce execută Saga să fie oprit în mijlocul execuţiei putem evidenţia potenţialele probleme cu această abordare. Practic, daca serverul este oprit oriunde între linia 4 si 14, va lăsă execuţia într-o stare inconsistentă, cu posibilitatea de a avea consecinţe seriose asupra utilizatorului. Ce se intamplă daca se execută doar rezervarea de Hotel, dar utilizatorul nu este notificat ? Acestuia ii va fi retrasă suma de pe card în mod abusiv, rezultand în posibile probleme legale pentru deţinătorul aplicaţiei. Astfel, garanţia atomiticăţii într-o tranzacţie distribuită este foarte importantă.

Motivele pentru care această întrerupere a execuţiei practic face imposibilă garantarea atomicităţii întregii operaţii, este că toată acţiunea se execută ca parte a unei singure incercări, şi orice întrerupere a acestuia face ca operaţia să nu poată fi reîncercată şi eventual continuată, pentru a garanta execuţia completa a strategiei Saga.

3.2.2 Execuţie durabilă

O alta problemă cu codul iniţial, este ca nu acoperă nici posibilitatea ca unul din paşi să eşueze din motive de probleme de reţea, iar astfel, toată operaţiunea poate ramane într-o stare necunoscută, daca pe perioada roll-back-ului deci între 10 si 14, unul din paşi eşuează. Pentru a acoperi ambele probleme, codul ar trebui să fie transformat dupa cum urmează :

```
1  var booking = await sagaService.RegisterIfNotPresent(booking); // returneaza starea
2  // inregistrata in baza de date a rezervării sau inregistreaza rezervarea
3  // în baza de date, daca este prima execuţie.
4  try {
5          if (!booking.isHotelBooked){
6                  await sagaService.bookHotel(booking);
7                  await sagaService.saveState(booking);
8                  }
9          if (!booking.isTaxiBooked){
10                 await sagaService.bookTaxi(booking);
11                 await sagaService.saveState(booking);
12         }
13         if (!booking.isFlightBooked){
14                 await sagaService.bookFlight(booking);
15                 await sagaService.saveState(booking);
16         }
17 } catch (Exception){
```

```
18          try{
19                  if (booking.isHotelBooked){
20                          await sagaService.cancelHotel();
21                          await sagaService.saveState(booking);
22                  } else if (booking.isTaxiBooked){
23                          await sagaService.cancelHotel();
24                          await sagaService.cancelTaxi();
25                          await sagaService.saveState(booking);
26                  }
27          }
28          catch (Exception){
29                  await sagaService.markAsFailed(booking);
30          }
31  }
```

Odată cu modificarea secvenţei de cod, putem observa următoarele modificări :

- Introducerea unui nivel de decuplare, la nivelul bazei de date, ca permite re-execuţia secvenţei de cod fără a pierde starea precedentă, pentru a evita trimiterea mai multor request-uri prin reţea decât este necesar.
- Introducerea unui nou nivel de tratare a excepţiilor, care prinde eventualele probleme de reţea la nivelul acţiunii de rollback, şi care marchează rezervarea ca eşuată, pentru a fi putea fi reparată în mod manual. (atunci când execuţia Saga nu poate fi executată pe flow-ul de rollback, acţiunea trebuie). Astfel eventualele acţiuni care nu pot fi executate respectând garanţiile Saga, vor fi marcate ca mai apoi acele tranzacţii să fie reparate prin operaţiuni manuale.

Utilizând această noua strategie, codul nu va mai fi executat ca parte dintr-un singur context, ci aplicaţia va încerca să consume toate rezervările ce nu sunt într-o stare finală, pana când acestea fie sunt executate şi au ajuns într-o stare finală corectă, fie sunt marcate ca eşuate şi necesită o recuperare manuală.

Pentru ca acest lucru să fie posibil, trebuie implementat un mecanism specializat care să ruleze la intervale predefinite, şi să încerce să consume rezervările ce aşteaptă să fie procesate. Această decuplare ajută din punct de vedere al rezilienţei sistemului, dar vine cu un cost suplimentar de dezvoltare deoarece această tehnică nu standard în arhitectura unui Web API.

3.2.3 Garanţii de execuţie. Cel puţin odată versus exact o dată

Deşi o parte din probleme au fost rezolvate, încă rămân în picioare anumite vulnerabilităţi cum ar fi că sistemul înca poate suferi de pe urma unei intreruperi spontane între

momentul în care se execută o acţiune, şi momentul în care aceasta este salvată în baza de date. Din cauza acestui lucru, sistemul obţinut oferă doar garanţia ca toate sistemele ce iau parte la un workflow, vor fi apelate cel putin odata, dar nu exact odata. Problema garantiei de executie exact o data în cadrul unui sistem distribuit este intens dezbătută şi în multe cazuri, imposibil de obţinut. Atât timp cât endpoint-urile sistemelor care iau parte la workflow sunt idempotente, garanţia ca aceste endpoint-uri vor fi apelate cel puţin odată este suficientă.

În varianta finală a acestei implementări componentele implicate sunt :

- Serviciul ce execută maşina de stări a workflow-ului (partea de cod discutată mai sus)
- Serviciul ce se ocupă de preluarea rezervărilor din baza de date si pornirea serviciului de mai sus.
- Baza de date, ce se ocupă de persistarea stărilor pentru a permite o execuţie durabilă

3.2.4 Interacţiunea din exterior cu workflow-ul. Operaţii de lungă durată

Până acum, implementarea a analizat doar problemele de durabilitate a workflow-ului pe cazul rapid, atunci cand toate sistemele pot răspunde în mod sincron la apelurile trimise de sistemul ce implementează Saga. Ce se intamplă insă daca unul din sisteme poate oferi un răspuns foarte lent, spre exemplu in 24 de ore ? Implementarea curentă nu permite interacţiuni cu workflow-ul din exterior, deci toata arhitectura trebuie modificată pentru a cuprinde si acest caz.

Pentru a permite interacţiunea cu workflow-ul din exterior, sistemul va expune şi endpointuri externe pentru a modifică starea unei rezervări. Astfel, sistemul ce implementează Saga doar va anunţa sistemul hotelier cererea de rezervare a camerei iar sistemul hotelier va veni cu un apel către sistemul central odată ce finalizează rezervarea (posibil peste o perioadă lungă de timp) şi va modifica în mod direct starea rezervării. La nivelul serviciului ce implementează maşina de stări, acesta va suferi o modificare în sensul că, dacă la un moment dat nu se află într-o stare ce permite continuarea (de exemplu după trimirea request-ului de rezervare a camerei, camera nu este rezervată) acesta va termina execuţia (return) şi va aşteaptă următoarea procesare (care va fi iniţiată de Serviciul ce se ocupă de preluarea rezervărilor din baza de date) pentru a verifica din nou starea si pentru a trece eventual la etapa următoare.

Serviciile ce interacţionează cu rezervarea vor putea modifica starea acesteia accesând endpoint-urile dedicate fiecărei operaţii, folosind id-ul rezervării primit pe request. De exemplu, daca sistemul de rezervare a zborurilor expune un endpoint de rezervarea de zbor POST /api/flights , în sistemul central, va exista un endpoint pereche POST

/api/bookings/{bookingId}/book_flight ce va permite tranziţia rezervării în starea FlightBooked. Astfel, workflow-ul propriu zis nu va fi executat pe perioada în care se aşteaptă finalizarea unei acţiuni din partea unui alt sistem.

3.2.5 Cronometre durabile.

Deşi sistemul în această formă permite interacţiunea cu alte servicii, există o potenţială vulnerabilitate, şi anume daca un serviciu, din diverse motive, nu mai apelează endpoint-ul corespunzător pentru a trece workflow-ul în starea corectă. În acest caz, workflow-ul va rămane pentru totdeauna într-o stare itermediară. Pentru a rezolva această problemă, trebuie sa adăugam suport pentru un cronometru durabil.

Cronometrul durabil este un concept teoretic într-o arhitectură orientată pe microservicii, şi se referă la capacitatea unui cronometru de a rezista eventualelor întreruperi ale procesului. În cadrul sistemului nostru, pentru a obţine un cronometru durabil elementar, pe entitatea rezervării se va salva şi data la care a fost creat şi data la care s-a făcut ultimul update. Astfel, orice cronometru durabil se obţine prin compararea datei la care a fost creat/updatat plus un anumit delta, cu data curenta de la momentul execuţiei. Următoarea secvenţă de cod, ilustrează cum poate fi verificat în mod durabil daca workflow-ul a tranziţionat în mod corect către o stare intermediară, în timp util.

```
1         var relativeTime = booking.CreationTime.AddMinutes(10);
2         if (!booking.isHotelBooked){
3                 if (DateTime.Compare(relativeTime, DateTime.UtcNow)  < 0 ){
4                         throw new TimeoutException(‘Workflow-ul nu a trecut la pasul
5                 }
6                 await sagaService.bookHotel(booking);
7                 await sagaService.saveState(booking);
8                 }
9         }
```

Codul de mai sus garantează ca daca workflow-ul nu tranziţionează către o anumită stare (HotelBooked în acest caz) workflow-ul va porni strategia de rollback printr-o excepţie. Astfel, sistemul devine rezilient la eventuale erori ce apar în serviciile ce iau parte la workflow, şi este capabil să re-aducă workflow-ul într-o stare consistentă fără intervenţii din exterior.

3.2.6 Optimizarea numarului de request-uri

Deşi garanţia de cel puţin odată ne permite ca request-urile către celelate servicii sa fie trimise de mai multe ori, beneficiind de faptul că acestea sunt idempotente, sistemul nu ar trebui să abuzeze şi atât timp cât totul funcţionează corect, sistemul ar trebui sa trimită request-urile exact o dată. Însă, analizând codul de mai sus si extinzând

pattern-ul şi la celelalte acţiuni, putem observa o problemă. Presupunând ca acţiunea de rezervare a unei camere de hotel durează 24 de ore, iar rezervarea noastră este preluată si procesată de către serviciul ce se ocupă de preluarea rezervărilor din baza de date o data pe minut, atunci se vor trimite 1440 de request-uri către sistemul de rezervare a camerelor de hotel, în loc de una, acest lucru fiind innacceptabil, chiar şi într-un sistem ce garantează execuţia cel puţin odată a acţiunilor.

Pentru a rezolva această problemă vor fi introduse stări intermediare, ce vor evidenţia natura de lungă durată a operaţiunilor, şi vor ajuta sistemul de gestiune al Saga să obţine execuţia exact o dată a acţiunilor, atât timp cât nu apare nici o eroare în vreunul din sistemele implicate.

```
1         var relativeTime = booking.CreationTime.AddMinutes(10);
2         if (!booking.isHotelBooked){
3                 if (DateTime.Compare(relativeTime, DateTime.UtcNow)  < 0 ){
4                         throw new TimeoutException(‘Workflow-ul nu a trecut
5                          la pasul următor destul de rapid‘);
6                 }
7                 if (booking.status != ’’BookingHotel’’){
8                         await sagaService.bookHotel(booking);
9                         await sagaService.saveState(booking, ‘BookingHotel‘);
10                }
11        }
```

Salvarea stărilor de tranziţie în entitatea rezervării permite sistemului să decidă să nu mai execute o anumită acţiune pe perioada în care asteaptă un raspuns din exterior, datorită naturii de lungă durată a operaţiunilor.

3.2.7 Considerente de deployment şi scalabilitate

Având în vedere că arhitectura descrisă în această secţiune este bazată pe un server web şi o bază de date relaţională, atunci când vine vorba de deployment, avem multiple alegeri. În primul rând această arhitectură este pretabilă atât pentru a rula on-prem cât şi în cloud.

On-prem avem posibilitatea să hostăm propriul nostru web-server şi propriul server de baze de date pe propria infrastructură. În schimb acest lucru poate afecta scalabilitatea deoarece infrastructura proprie are unele limitări, una dintre cele mai mari fiind faptul ca este greu de upgradat, deoarece vorbim de resurse fizice. Astfel, scalarea pe verticală pentru a obţine performanţe mai mari de pe urma acestui servicii este îngreunată daca alegem să rulăm totul pe propria infrastructură. În schimb scalarea pe orizontală poate fi obţinută relativ simplu adaugând mai multe servere si rulând mai multe instanţe ale aplicaţiei noastre. Însă, în acel caz vor exista complicaţii la nivel de rutare şi gestiune a traficului ce este preluat de către aplicaţie. În contextul în care aplicaţia

noastră gestionează workflow-uri de lungă durată, acestea nu pot fi executate în cadrul unui singur apel HTTP astfel încat să putem returna rezultatul pe răspuns. În schimb, utilizatorul care trimite primul request, de pornire a unui workflow, va trebui sa revină către serviciul nostru cu un alt request, mai târziu pentru a putea afla rezultatul final. Deoarece rezultatul va fi stocat la nivelul bazei de date, în cazul în care serviciul nostru este scalat orizontal şi avem mai multe instanţe, separate în mod fizic intre ele, este foarte important unde vor ajunge apelurile viitoare ale unui consumator ce doreşte să afle rezultatul unui workflow pe care deja l-a pornit. Acest lucru este critic deoarece datele despre un workflow nu există pe toate instanţele într-un setup scalat pe orizontală, deci un request care doreşte să afle statusul workflow-ului cu id-ul 24566789 poate primi un răspuns doar dacă request-ul acestuia ajunge pe instanţa care a şi pornit workflow-ul 24566789. Acest lucru adaugă un nivel suplimentar de complexitate, ce trebuie gestionat la un nivel superior aplicaţiei, de rutare, care să trimită request-urile către instanţa corespunzătoare.

In-cloud avem posibilitatea să hostăm aplicaţia noastră la oricare dintre providerii deja stabiliţi în industrie (AWS, Azure, Google Cloud) deoarece toţi au suport pentru a hosta web-server si baza de date în numeroase modalităţi. Deşi problema scalabilităţii pe orizontală rămâne validă şi în Cloud, aici în schimb putem alege varianta scălării pe orizontală doar a web-server-ului, în timp ce baza de date poate fi scalată pe verticală, fără a suferi întreruperi ale aplicaţiei. Astfel, evităm complexitatea managementului de rutare către instanţa corectă, deoarece toate instanţele serverului web vor fi conectate la aceeaşi bază de date şi implicit vor putea accesa datele tuturor workflow-urilor. Astfel, execuţia în cloud a acestei arhitecturi poate veni atât cu avantaje de performanţă şi scalabilitate, cât şi cu reduceri semnificative de costuri, deoarece toate resursele pot fi scalate în mod automat în funcţie de nevoile aplicaţiei la un anumit moment dat, rezultând într-o utilizare mult mai eficientă a resurselor.

3.3 Implementarea arhitecturii folosind durable Functions

În capitolul anterior am putut observa dificultăţile în a implementa un serviciu care să implementeze conceptul de execuţie Saga, pentru a gestiona o tranzacţie distribuită. Pornind de la aceeaşi problemă, în acest capitol va fi analizată dezvoltarea unui serviciu care foloseşte de această data Durable Functions, pentru a analiza care sunt problemele pe care această tehnologie le rezolvă.

Pentru e putea intelege mai bine arhitectura execuţiei unei Saga folosind Durable functions, vor fi analizate conceptele ce stau la baza dezvoltării unei soluţii folosind Durable functions :

Funcțiile de orchestrare descriu modul în care sunt executate acțiunile și ordinea în care sunt executate. Funcțiile orchestrator descriu orchestrarea în cod (C# sau JavaScript), așa cum se arată în modelele de aplicație Durable Functions. O orchestrație

poate avea multe tipuri diferite de acțiuni, inclusiv funcții de activitate, sub-orchestrații, așteptare pentru evenimente externe, HTTP și cronometre durabile. Funcțiile de orchestrator pot interacționa și cu entităţi durabile. Funcțiile orchestrator sunt scrise folosind cod obișnuit, dar există cerințe stricte privind modul de scriere a codului. Mai exact, codul funcției de orchestrator trebuie să fie determinist. Nerespectarea acestor cerințe de determinism poate face ca funcțiile orchestratorului să nu ruleze corect. Codul funcţiei orchestrator este rulat de mai multe ori, asemănator cu modalitea în care în capitolul anterior am ales un mecanism de polling care rulează codul de orchestrare până orchestrarea ajunge într-o stare finală. Deşi implementarea propriu zisă diferă în multe zone, conceptul de bază este acelaşi iar acesta vine cu această nevoie de a scrie cod determinist, pentru a nu avea rezultate diferite pentru aceeaşi orchestrare, în funcţie de momentul în care aceasta a fost executată.

Funcțiile de activitate sunt unitatea de bază de lucru într-o orchestrare durabilă a funcțiilor. Funcțiile de activitate sunt funcțiile și sarcinile care sunt orchestrate în proces. De exemplu, în cazul analizat în cadrul acestei lucrări, pentru a procesa o o rezervare este necesară execuţia unor sarcini. Aceste sarcini implică comunicarea cu serviciul de taxi, cu cel hoteliar si cu cel aviatic. Fiecare dintre aceste sarcini individuale poate fi modelat ca o activitate. Aceste funcţii de activitate pot fi executate în serie, în parelel sau o combinaţie a ambelor. Spre deosebire de funcțiile de orchestrator, funcțiile de activitate nu sunt restricționate în ceea ce privește tipul de lucru pe care îl puteți face în ele. Funcțiile de activitate sunt frecvent utilizate pentru a efectua apeluri în rețea sau pentru a rula operațiuni cu consum intensiv de CPU. O funcție de activitate poate, de asemenea, să returneze date înapoi la funcția de orchestrator. Durable Task Framework garantează că fiecare funcție de activitate apelată va fi executată cel puțin o dată în timpul execuției unei orchestrații.

Funcțiile entitate definesc operațiuni pentru citirea și actualizarea unor părți mici de stare. Adesea ne referim la aceste entități cu stare ca entități durabile. La fel ca funcțiile de orchestrator, funcțiile de entitate sunt funcții cu un tip de declanșator special, declanșator de entitate. Ele pot fi, de asemenea, invocate din funcțiile client sau din funcțiile orchestrator. Spre deosebire de funcțiile de orchestrator, funcțiile de entitate nu au constrângeri specifice de cod. Funcțiile entității gestionează, de asemenea, starea în mod explicit, mai degrabă decât să reprezinte implicit starea prin fluxul de control. Acestea nu sunt re-rulate în mod constant, în schimb, garantează accesul în mod serializat la datele interne, astfel este sigur ca aceste entităţi durabile să fie accessate din multiple activități sau orchestrări în mod concurent, fără a ne ingrijora legat de integritatea datelor.

Cronometrele Durabile din Durable Functions implementează acelaşi concept descris şi în capitolul anterior, şi oferă o abstractizare peste implementarea standard, pentru o utilizare rapidă.

Evenimentele Externe deasemenea implementează un concept similar cu problema

interacţiunii cu procesele in curs descris anterior, şi facilitează comunicarea din exterior către o orchestrare care înca rulează. Combinaţie dintre suportul pentru evenimente externe si cronometre durabile rezulta într-un suport foarte complex, care acoperă multe situaţii de reale, care necesită comunicări asincrone între serviciul de orchestrare si celelalte servicii terte care iau parte la acţiune.

3.3.1 Noua Arhitectură

Folosind conceptele tocmai definite, vom mapa problemele ce trebuie rezolvate pentru a implementa serviciul de gestiune a rezervărilor către soluţiile puse la dispoziţie de Durable Functions.

Analizând 3.1 putem observa urmatoarea corespondentă între conceptele de bază expuse de Durable Functions, si necesităţile noastre :

- Nucleul serviciului nostru de gestiune al operaţiei de rezervare este plasat într-o funcţie de orchestrare, ce garantează execuţia durabilă a logicii, astfel încât Saga sa fie executată cu success indiferent de eventualele întreruperi mecanice sau excepţii ne-aşteptate în timpul execuţiei.

- Fiecare interacţiune cu serviciile externe este izolată într-o funcţie de activitate, pentru a permite execuţia lor conform unor politici de retry şi pentru a izola comportamentul ne-determinist (apeluri asyncrone prin reţea) de logica principală de orchestrare.

- Starea generală a rezervării este stocată prin intermediul unei funcţii entitate ce asigură capacitatea de a citi şi updata starea rezervării în mod distribuit, pe măsura ce serviciile externe procesează request-urile trimise de către serviciul principal.

Analizând strict codul de orchestrare putem observa că acesta nu este cu mult mai complex, doarece conceptele de durabilitate sunt abstractizate sub interfeţe uşor de folosit.

```
1 var entityGuid = context.GetInput<string>();
2 var entityId = new EntityId(nameof(Booking), entityGuid);
3 var booking = await context.CallEntityAsync<Booking>(entityId, "Get");
4 var isHotelBooked= await context.CallActivityAsync<bool>("BookHotel", null);
5 booking = await context.CallEntityAsync(new EntityId(nameof(Booking),
      entityGuid),"UpdateHotel", isHotelBooked);
6 if (booking.Hotel)
7   {
8       var isTaxiBooked = await context.CallActivityAsync<bool>("BookTaxi",
            null);
9       booking = await context.CallEntityAsync(new EntityId(nameof(Booking),
            entityGuid),"UpdateTaxi", isTaxiBooked);
```

```
10      if (booking.Taxi)
11        {
12          var isFlightBooked = await context.CallActivityAsync<bool>("
                BookFlight", null);
13          booking = await context.CallEntityAsync(new EntityId(nameof(Booking)
                , entityGuid),"UpdateFlight", isFlightBooked);
14        }
15        else // hotel booked, taxi not booked - cancel hotel
16        {
17          await context.CallActivityAsync<bool>("CancelHotel", null);
18          booking = await context.CallEntityAsync(new EntityId(nameof(Booking)
                , entityGuid),"UpdateHotel", false);
19        }
20        // hotel booked, taxi booked, flight not booked - cancel hotel & taxi
21        if (booking.Hotel && booking.Taxi && !booking.Flight)
22          {
23            await context.CallActivityAsync<bool>("CancelHotel", null);
24            booking = await context.CallEntityAsync(new EntityId(nameof(
                  Booking), entityGuid),"UpdateHotel", false);
25            await context.CallActivityAsync<bool>("CancelTaxi", null);
26            booking = await context.CallEntityAsync(new EntityId(nameof(
                  Booking), entityGuid), "UpdateTaxi", false);
27          }
28    }
```

Folosind Durable Functions, codul de orchestrare se simplifică considerabil, deoarece excepţiile ce apar în cadrul unei orchestrări declanşează în mod automat marcarea acelei orchestrări ca fiind *Failed*, ceea ce poate fi monitorizat în mod automat pentru a fixa eventualele cazuri de eşec pe flow-ul de rollback.

Din punct de vedere al structurii codului, acesta acum descrie în mod procedural maşina de stări ce este implementată de serviciu, iar framework-ul asigură execuţia acestuia în mod durabil, printr-un mecanism de execuţie al orchestrării bazat pe event-sourcing. Practic, orchestrarea este re-executată de fiecare data când un eveniment este ridicat pentru aceasta. Evenimentele pot avea diferite surse, interne sau externe. Evenimentele externe sunt cele iniţiate de utilizatori sau alte servicii, de exemplu cand serviciul de orchestrare este anunţat printr-un eveniment că o anumită acţiune s-a terminat. Evenimentele interne sunt cele care determină re-execuţia unei orchestrări când un cronometru durabil care ii se adresează expiră, sau o sub-orchestrare sau o activitate ce este inlănţuită în mod direct este finalizată.

Atunci când o orchestrare este re-executată, aceasta sare în mod automat peste paşii care au fost executaţi deja. De fiecare dată cand prin intermediul contextului este inregistrată o sub-acţiune a unei orchestrări, o intrare în tabela de istoric este

generată care inregistrează statusul si eventualul rezultat al acelei sub-acţiuni. Atunci când orchestrarea este re-executată, aceasta incarcă din memorie toţi paşii care au fost deja executaţi şi înregistrează doar acţiunile noi ce trebuiesc înregistrate. Astfel, este practic suportată în mod standard o strategie care minimizează request-urile către serviciile terţe şi care oferă o garanţie de execuţie cel putin o dată a activităţilor, fără însă a abuza în mod activ de aceasta. În cazul în care pe perioada execuţiei nici o exceptie ne-asteptată nu apare, fiecare activitate este executată exact o dată. Acest suport care în mod clasic necesita destul de mult design si efort de implementare, este oferit de către Durable Functions, acesta fiind un alt avantaj pentru a alege această tehnologie pentru a manageria workflow-uri de lungă durată în cloud.

La nivelul Activităţilor, acestea gestionează comunicarea cu serviciile externe. Din punct de vedere al execuţiei acestea sunt rulate urmând o garanţie de cel puţin o dată, iar din acest motiv implementarea lor este indicată să fie idempotentă, pentru a evita potenţialele acţiuni duplicate, în cazul unei execuţii multiple. Pe de altă parte, din cuauză ca acestea nu sunt re-executate de mai multe ori în mod recurent, acestea nu trebuie sa respecte aceleaşi restrictii ca orchestrările, şi pot avea comportament ne-deterministic. Rezultatul acestora este returnat către orchestrări, iar acestea la rândul lor updatează entitatea care menţine starea rezervării.

3.3.2 Cronometre Durabile în Durable Functions

Cronometrele Durabile sunt suportate în mod standard de către Durable Functions prin intermediul contextului de execuţie. Conceptul din spate e asemănător cu cel propus în capitolul anterior, dar diferă prin faptul că cronometrul odată înregistrat, este gestionat ca un eveniment de sine stătător, şi nu este strâns legat de execuţia orchestrării.

```
1  DateTime expireAt = context.CurrentUtcDateTime.AddHours(48);
2  var cts = new CancellationTokenSource()
3  var timeoutTask = context.CreateTimer(expireAt, cts);
4  if (!booking.isHotelBooked){
5          var hotelBookingTask = context.CallActivityAsync<bool>("BookFlight",
               null);
6          Task winner = await Task.WhenAny(hotelBookingTask, timeoutTask);
7      if (winner == hotelBookingTask)
8      {
9          cts.Cancel();
10        booking = await context.CallEntityAsync(new EntityId(nameof(Booking),
               entityGuid),"UpdateFlight", isFlightBooked);
11     }
12     else
13     {
```

```
14          throw new TimeoutException(''The Timer expired before receiving a
                response from the hotel activity'');
15      }
16          }
17  }
```

Putem observa că timpul mereu este calculat folosind timpul current extras din context-ul de execuţie. Acesta este un mod determinist de a extrage timpul curent, fiindcă va returna mereu aceeaşi valoare, chiar dacă orchestrarea este re-executată la diverse moment în timp. Dupa creearea initială acest timer este inregistrat în baza de date, iar în momentul în care acesta expira, va declanşa o re-execuţie a orchestrării. În acel moment timer-ul va fi fost finalizat, şi urmărind codul de mai sus, se va arunca o excepţie. Daca până în momentul în care timer-ul expiră este finalizată altă acţiune care concura cu timer-ul, atunci timer-ul trebuie să fie explicit închis folosind CancellationToken-ul pentru a preveni re-execuţii inutile a aceleaşi orchestrări. În cazul in care timerele rămân deschise chiar si atunci când nu mai sunt aşteptate, acest lucru duce la o degradare a performanţei funcţiilor noastre durabile.

3.3.3 Considerente de deployment şi scalabilitate

Din punct de vedere al deployment-ului soluţia ce foloseşte Durable Functions suferă de "vendor lock-in". Acest lucru inseamnă ca soluţia aceasta nu este generică din punct de vedere al deployment-ului, şi din acest punct de vedere, dezvoltatorul are o singură soluţie : deployment-ul in Azure Cloud folosind Durable Functions. Execuţia soluţiei on-prem sau portarea acesteia către un alt provider soluţii cloud nu este posibilă.

În schimb, soluţia disponibilă, de a rula aplicaţia folosind Durable Functions, vine cu 2 mari avantaje :

- Scalare pe orizontală infinită, fără restricţii de performanţă şi fără complexitate suplimentară în nivelul de rutare, deoarece acest lucru este deja gestionat de Azure.

- Mod de calcul a costurilor foarte avantajos, deoarece se plăteşte doar timpul de execuţie, nu şi timpul de aşteptare. Iar intr-un sistem în care timpul de execuţie reprezintă mai puţin de 2% din timpul total de viaţă a workflow-ului (deoarece majoritatea timpului este petrecută asteptând timere sau eveniment externe) acest lucru este foarte valoros si duce la o reducere considerabilă a costurilor de execuţie a unui astfel de sistem.

  Deşi această limitare a opţiunilor de deployment poate fi o vulnerabilitate pe viitor, pentru moment această soluţie este cea mai bună deoarece în piaţă nu există un competitor care să ofere tehnologie statefull serverless într-un mod mai bun.

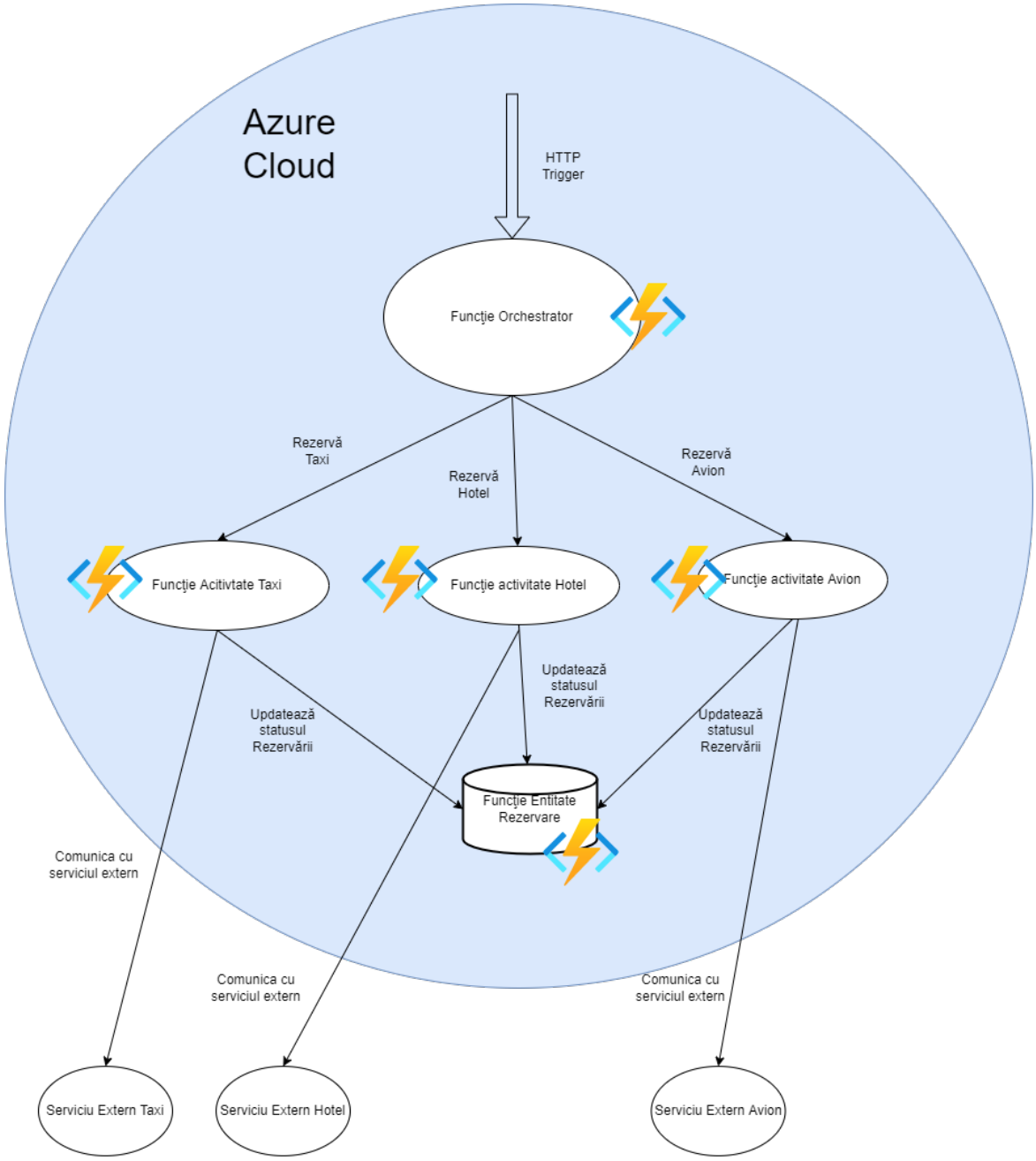

Figura 3.1: Rezervare distribuită folosind Durable Functions

Capitolul 4

Rezultate şi Concluzii

4.1 Concluzii

Listă de figuri

Listă de tabele