



UNIVERSITATEA DIN BUCUREȘTI



FACULTATEA  
DE  
MATEMATICĂ ȘI INFORMATICĂ

SPECIALIZAREA INFORMATICĂ

LUCRARE DE DISERTAȚIE  
EXECUȚIA WORKFLOW-URILOR DURABILE ÎN  
CLOUD

Absolvent

Moldovan George-Alexandru

Coordonator științific

Prof. dr. Letiția Marin

București, septembrie 2022

## Rezumat

Considerând trend-ul ascendent al arhitecturii orientate spre microservicii și migrarea de la vechiul mod de dezvoltare monolit, această lucrare de disertație își propune analiza soluțiilor prezente în piața în ceea ce privește managementul unei arhitecturi pe microservicii în cloud, și analiza unor contribuții personale aduse unui framework open-source de gestionare a workflow-urilor în cloud. Sistemele informatice preiau din complexitatea operațiilor de zi cu zi, astfel că arborii de decizie trebuie interpretați și gestionați în mod corect pentru a realiza un sistem care să îndeplinească cerințele de piață curente. Pentru o lungă perioadă de timp, problema gestionării tranzacțiilor distribuite și management-ul workflow-urilor a fost inexistentă, deoarece într-un sistem monolit, caracteristicile tranzacționale ale bazelor de date ce susțineau astfel de sisteme erau îndeajuns pentru a avea toate garanțiile necesare astfel încât sistemul să fie mereu lăsat într-o stare consistentă. Microserviciile și arhitecturile distribuite în general, deși vin cu o serie lungă de avantaje, poate cel mai greu de gestionat lucru este management-ul stării unei acțiuni, atunci când aceasta se întinde pe mai multe microservicii, și 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 an 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

<b>1</b>	<b>Introducere</b>	<b>3</b>
1.1	Motivatie . . . . .	3
1.2	Context . . . . .	4
1.3	Alte analize ale problemelor curente a arhitecturii Serverless . . . . .	5
1.4	Conținutul lucrării . . . . .	6
<b>2</b>	<b>Analiza Tehnologiei Durable Task Framework si a providerilor disponibili</b>	<b>7</b>
2.1	Analiză generală . . . . .	7
2.2	Folosirea evenimentelor în DTF . . . . .	8
2.3	DTF în Durable Functions . . . . .	11
2.4	Greutăți în gestionarea workflow-urilor de lunga durată - Versionarea Orchestrarilor . . . . .	12
<b>3</b>	<b>Workflow-uri de lungă durată - Analiză paralelă a arhitecturii</b>	<b>14</b>
3.1	Descrierea problemei . . . . .	14
3.2	Implementarea arhitecturii în mod clasic . . . . .	15
3.2.1	Implementarea naivă . . . . .	15
3.2.2	Execuție durabilă . . . . .	16
3.2.3	Garanții de execuție. Cel puțin odată versus exact o dată . . . . .	17
3.2.4	Interacțiunea din exterior cu workflow-ul. Operații de lungă durată . . . . .	18
3.2.5	Cronometre durabile. . . . .	19
3.2.6	Optimizarea numarului de request-uri . . . . .	19
3.3	Implementarea arhitecturii folosind durable Functions . . . . .	20
<b>4</b>	<b>Rezultate și Concluzii</b>	<b>21</b>
4.1	Concluzii . . . . .	21

# Capitolul 1

## Introducere

### 1.1 Motivatie

Unul din principiile de baza ale programării este reutilizarea. Motivația din spatele acestei lucrări o reprezintă dorința de o contribui la un framework care rezolvă o problemă generică, cu care se confruntă toți dezvoltatorii care trebuie sa gestioneze tranzacții distribuite. Analiza diferitelor metode pentru rezolvarea problemei de gestiune a workflow-urilor în cloud, în special într-o arhitectură ce se bazează pe funcții 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 infinită implicită și facturare bazată pe consum. Cu toate acestea, garanțiile slabe de execuție și suportul nativ pentru stocare a stării a FaaS creează provocări serioase atunci când se dezvoltă aplicații care necesită stare persistentă, garanții de execuție sau sincronizare. Acest lucru a motivat o nouă generație de soluții serverless care oferă abstractizări ce stochează starea aplicației. De exemplu, noua soluție 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 încearcă sa dezvolte aplicații ce gestionează workflow-uri de lungă durată, cu toții rezolvă aceeași problemă și anume lipsa separării între nivelul de execuție si nivelul de stocare a soluțiilor existente FaaS. Cu toții rezolvă o problemă generică, de salvare a stării în anumite puncte ale execuției, pentru a putea relua workflow-ul în eventualitatea în care agentul pe care rulează aplicație pică înainte de finalizarea workflow-ului. Acest lucru era foarte greu de realizat in arhitecturi serverless deoarece toate soluțiile existente până la apariția Azure Durable Functions, suportau doar execuții stateless în mod standard. Deci până la apariția soluțiilor ce oferă garanții de execuție puternice in lumea Serverless, această tehnologie nu era o opțiune populară pentru dezvoltarea aplicațiilor ce aveau nevoie de gestionare a stării și era limitată la execuția unor părți mici ale aplicațiilor pentru care stările 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 :

- Scutește dezvoltatorii de servicii cloud de la interacțiunea 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].

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 creștere în adopție foarte rapidă, lucru vizibil și în figura 1.1. Această creștere este datorată în mare parte ușurinței procesului de dezvoltare a aplicațiilor Serverless și beneficiile pe care le aduc din punct de vedere al scalării în mod automat, și a gestionării complete a infrastructurii ce stă la baza aplicațiilor.

Unul din lucrurile care poate duce adopția tehnologiei serverless la un cu totul alt nivel, este tocmai capacitatea de a putea dezvolta aplicații întregi, ce se pot baza pe starea sistemului în cadrul execuției și capacitatea de a gestiona long running workflows, o nevoie care este prezentă în mai toate aplicațiile curente.

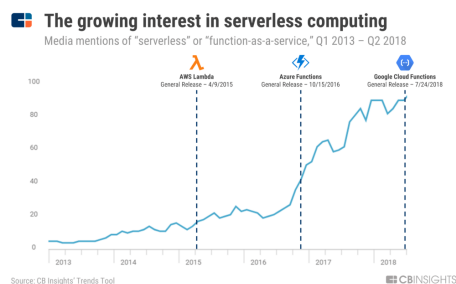


Figura 1.1: Statistică ce evidențiază importanța domeniului cloud computing în ultimii ani

Sursă: <https://www.cbinsights.com>

## 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 soluțiilor clasice în cloud cum ar fi mașini virtuale rezervată sau containere. O altă problemă semnalată este cea a cold-start-ului care poate face o aplicație bazată pe funcții serverless să pară înceată dacă pentru fiecare apel este necesar un cold start, din cauză că traficul nu este îndeajuns de mult pentru a împiedica funcția să scaleze la 0.

*Rajkumar et al.* [?] discută despre dificultățile dezvoltării unei arhitecturi serverless din punct de vedere al limitărilor cu care vine această nouă tehnologie și anume limitările de timp al execuției, de memorie al agentului și de management al stării. Acesta crede că este nevoie de o schimbare a mentalității de dezvoltare a aplicațiilor pentru a beneficia la adevaratul său potențial de tehnologiile Serverless, iar momentul în care aplicațiile enterprise complete vor fi migrate sau dezvoltate complet pe o arhitectură Serverless este încă departe. Acesta vede această tehnologie ca pe o unealtă ajutătoare în dezvoltarea aplicațiilor, dar pe viitor poate ajunge să fie nucleul aplicațiilor.

*Castro et al.* [?] ridică problema dezvoltării de aplicații stateful folosind tehnologii Serverless în viitor, o temă ce în aceea perioadă era doar o idee, dar după cum urmează să fie prezentat în aceasta lucrare, acum este o realitate. Alte probleme menționate ar fi ușurința cu care poate fi portată o aplicație legacy către o arhitectură Serverless, deoarece nu este de dorit să se piardă toate acele ore valoroase care au fost deja investite în aplicațiile existente.

*Hassan et al.* [?] discută despre problema limitării la un singur provider atunci când 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 ineficiențele arhitecturii serverless atunci când vine vorba de procesarea operațiilor care în mod normal ar beneficia de pe urma unui sistem cu mai multe nuclee, și implicațiile pe care aceasta limitare (2 nuclee per funcție) o are atunci când vine vorba de paralelizarea acțiunilor pentru a îmbunătăți viteza. Impactul major în acest caz este creșterea semnificativă a datelor transmise pe rețea pentru a obține același grad de paralelism într-un sistem serverless versus unul clasic în cloud.

## 1.4 Conținutul lucrării

Din punct de vedere al structurii, lucrarea va fi împărțită în 2 părți :

- Partea teoretică în care va fi analizat Durable Task Framework și cum funcționează acesta
- Partea practică ce va compara aceeași aplicație, dezvoltată în 2 moduri (clasic și folosind DTF)

În prima parte a lucrării va fi analizată tehnologia ce stă la baza soluțiilor de management a workflow-uri în cloud și anume, Durable Task Framework. Vom analiza modul în care este separat domeniul de execuție de domeniul de stocare, care sunt interfețele pe care trebuie să le respecte un provider, care sunt constrângerile care trebuie respectate atunci se folosește această tehnologie și bineînțeles care sunt beneficiile și dezavantajele sale.

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

- Capacitățile fiecărui sistem și nivelul de reziliență împotriva dezastrelor pe care îl pot demonstra
- Diferențele de dezvoltare între cele 2 abordări
- Analiză teoretică a costurilor între cele 2 arhitecturi
- Performanța 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 clasice de codare `async/await`. DTF stă la baza soluțiilor Microsoft de gestionare a workflow-urilor durabile în cloud, și folosind această tehnologie, a fost expusă prima soluție serverless ce permite gestionarea stării : Azure Durable Functions.

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

- DTF Core ce conține nivelul de abstractizare al gestionării, și care implementează conceptele de bază ce permit programarea orchestrărilor folosind sintagme `async` `await`
- DTF providers, care implementează interfețele expuse de Core pentru a folosi diferite medii de stocare, în funcție de nevoile dezvoltatorului.

La bază, framework-ul își propune să rezolve problema durabilității acțiunilor într-un sistem distribuit, în care este de așteptat 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 aplicației la un moment dat.

Nevoia de garanții de execuție a făcut ca DTF să construiască un sistem ce permite construcția unei structuri arborescente ce reprezintă o mașină de stări finite, în care, frunzele arborelui (Activitățile, în terminologia DTF) au garanția că vor fi executate cel puțin odata. Această garanție vine și cu o constrângere : Din cauză că Activitățile



nu sunt executate **exact o dată** este necesar ca toate acțiunile din cadrul unei Activități să 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 înțelege motivație din spatele alegerii arhitecturii DTF.

Gestionarea workflow-urilor prin mediator central cunoscută și drept **Orchestrare** în literatura de specialitate (*Megargel et al.[?]*), constă în prezența unei entități centrale ce comunică cu toate celelalte sisteme ce iau parte la un anumit workflow pentru a garanta execuția cu succes a unei mașini de stări. Topologia mediatorului este utilă pentru evenimentele care au mai mulți pași și necesită un anumit nivel de orchestrare pentru a procesa evenimentul. Așa cum vom vedea și în partea practică a acestei lucrări, sunt nenumărate cazuri ce se reduc la gestionarea unei mașini de stări. Un astfel de exemplu este un lanț de aprovizionare, în care, de la momentul achiziției până la momentul livrării sunt numeroase etape ce trebuie îndeplinite cu succes pentru ca operațiunea ca un întreg să fie un succes. Printre aceste etape s-ar număra pregătirea coletului, predarea lui către firma de curierat, înregistrarea plății și în final, livrarea propriu zisă. Un astfel de sistem poate beneficia 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 mașinii de stări între toți consumatorii evenimentelor, astfel fiecare poate lua o anumită decizie în funcție 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 acțiuni, în momentul procesării unui eveniment. Astfel, acest gen de arhitectură este potrivită pentru workflow-uri minimale, fără mașini de stări complicate, care nu au nevoie de un loc centralizat pentru asigurarea consistenței stării. Altfel, folosind această arhitectură

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

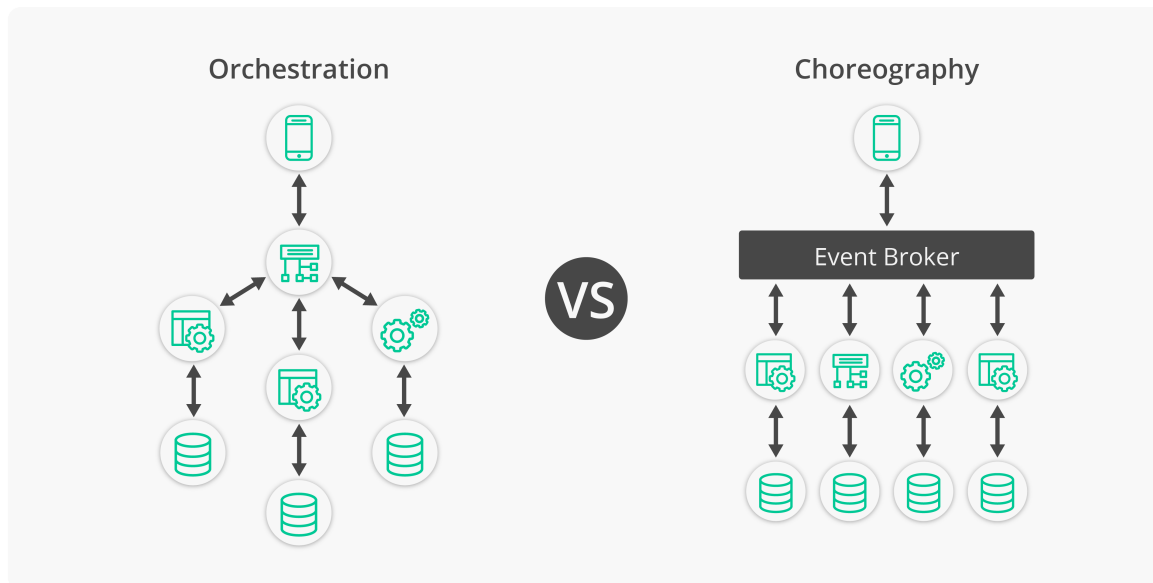


Figura 2.1: Orchestrare versus Coreografie

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 împreună cu Sql Server Provider, pentru a face o transpunere a conceptelor teoretice în lumea DTF. În 2.2 se poate observa structura schemei ce susține framework-ul atunci când este rulat împreună cu un SQL Server Provider. Acesta implementează conceptele teoretice într-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

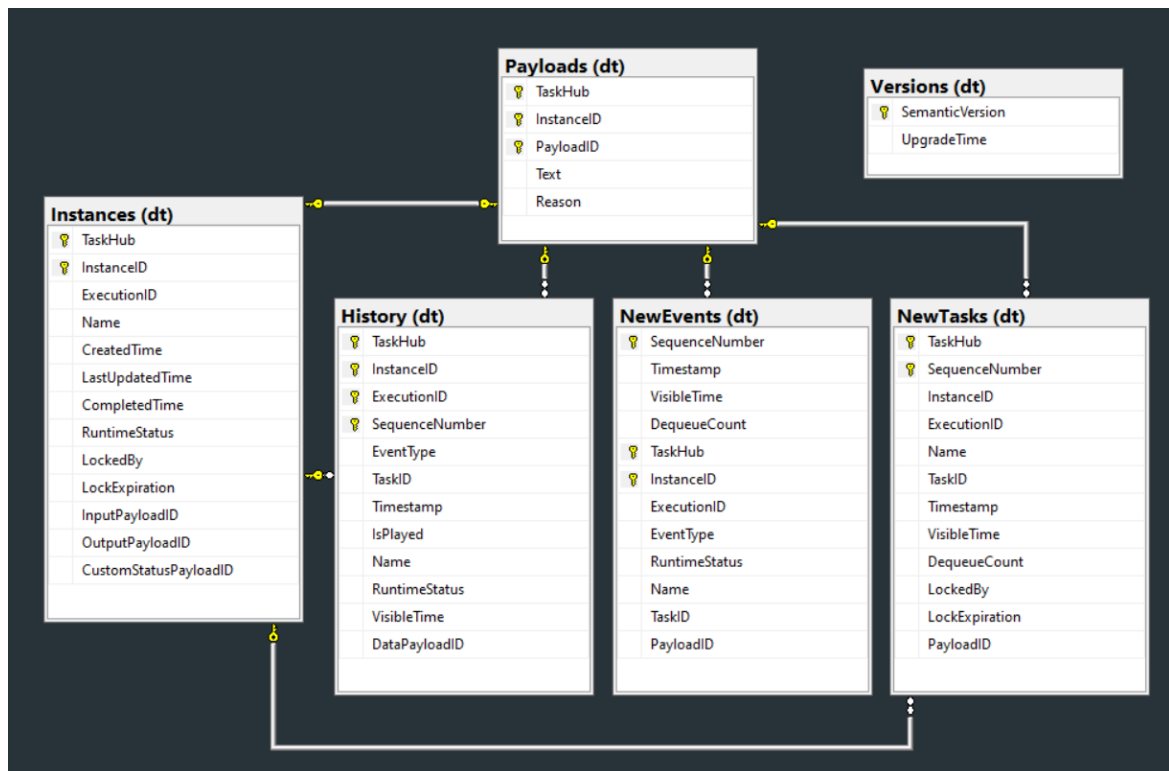


Figura 2.2: DTF Schema în DTF Sql Server provider

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 paralela către structura din DTF, urmărind schema descrisă la 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 înregistra un eveniment pentru o anumită orchestrare.

Mediatorul de evenimente este 'Orchestrarea' respectiv logica definită de dezvoltator pentru a gestiona workflow-ul. Această logică este rulată de fiecare dată când un eveniment este primit pentru un anumit workflow. Deși codul este rulat de mai multe ori, operațiune denumită și 'Replay', acțiunile care deja au mai fost executate odată nu sunt re-executate, ci rezultatele lor sunt direct încărcate direct din memorie, folosind istoricul orchestrării. Istoricul este salvat în tabela 'History'.

Canalele de evenimente sunt diferitele Task-uri înregistrate în table NewTasks. Acestea sunt folosite de pentru a declanșa execuția unor Activități, ce reprezintă frunzele arborelui de execuție a unui workflow. Task-urile sunt înregistrate pe canalele de evenimente de către 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 poată continua execuția workflow-ului.

Procesatoarele de evenimente sunt Workerii de DTF, și reprezintă logica din DTF Core care monitorizează în mod constant canalele de evenimente pentru a prelua eventualele noi Task-uri ce sunt disponibile pentru execuție. Astfel, se obține o arhitectură foarte decuplată în care înregistrarea de evenimente, gestionarea stării workflow-ului, execuția propriu zisă a diverselor taskuri și întoarcerea răspunsurilor înapoi către mediator sunt toate operațiuni decuplate, independente una de cealaltă.

Cel mai important aspect ce reiese din analiza implementării conceptelor de orchestrare bazată pe evenimente, este că datorită operațiilor independente, eșecurile care pot apărea atunci când această logică este executată într-un mediu distribuit nu vor afecta starea workflow-ului ca un întreg, fiindcă eșecul într-o parte a arhitecturii nu afectează alte părți iar fiecare unitate este independent reîncercabilă. Astfel revenirea în urma unui eșec (eroare de rețea, deconectare completă fie a workerilor fie a client-ului) sunt gestionate într-un mod grațios, fără efecte secundare.

## 2.3 DTF în Durable Functions

Conceptele prezentate în capitolul anterior, au fost reimplementate în DTF pentru a putea executa această tehnologie folosind soluții native în cloud, și pentru a expune o soluție pentru care exista o mare nevoie în piață : Statefull Serverless functions. Astfel a fost dezvoltat un nou provider pentru DTF, bazat pe Azure Storage tables de această dată, pentru a expune aceeași funcționalitate și în cloud.

Logica de bază a framework-ului a rămas aceeași în schimb implementarea celor 4 concepte de bază într-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 soluții noi pe piață ce a creat un nou domeniu în lumea serverless. Funcții ce sunt capabile să gestioneze starea, și să ruleze o perioadă foarte îndelungată de timp, dar pentru care se plătește doar timpul de execuție propriu zis. Acest lucru vine cu beneficii foarte mari din punct de vedere al costurilor, deoarece într-un workflow ce conține și pași manuali, cum ar fi aprobarea de către un operator a unei tranzacții, pe perioada în care se așteaptă acest eveniment, codul orchestrării nu se execută, deci practic nu există un cost asociat acestei operații.

Pe lângă funcționalitățile ce există în mod standard în toți 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 și sigur, deoarece oferă garanții de access serializat la aceasta.

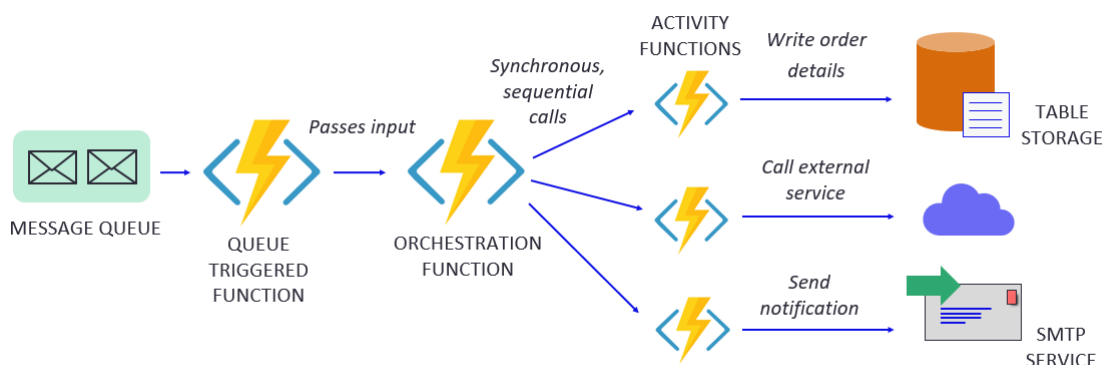


Figura 2.3: Exemplu arhitectură folosind Durable Functions

Sursă: <https://thisiszone.medium.com>

Așa cum se poate observa și în figura 2.3, conceptele clasice de DTF sunt transpuse în funcții individuale, de diferite tipuri. Astfel, o Orchestrare este reprezentată 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. Obținem astfel o arhitectură nativă în cloud, fără costuri de mentenanță a infrastructurii și 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 soluții cloud.

## 2.4 Greutăți în gestionarea workflow-urilor de lungă durată - Versionarea Orchestrarilor

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

Natura îndelungată a acestor workflow-uri face acestea să aibă următoarele particularități:

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

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

Prima problema se rezumă la dificultatea de versionarea a unei orchestrări. Atunci când apare o versiune nouă a mașinii de stări gestionată de o anumită orchestrare, cea veche nu poate fi direct înlocuită, deoarece încă există instanțe ale acelui tip de workflow care încă rulează. Astfel, versiunile noi de orchestrare trebuie mereu rulate în paralel cu cele vechi, până când toate instanțele ce rulează versiunea veche a workflow-urilor finalizează execuția. În acel moment versiunea veche a orchestrării poate fi stearsă.

Cea de-a doua problemă este mai mult doar o limitare și anume, din cauza logicii de replay multi pe perioada de viață a unei orchestrări, dezvoltatorul trebuie să se asigure că tot cod-ul din cadrul unei orchestrări, va returna același rezultat indiferent de momentul în care se rulează. Pe scurt, comportamentul unei orchestrări trebuie să fie determinist și constant indiferent de timp.

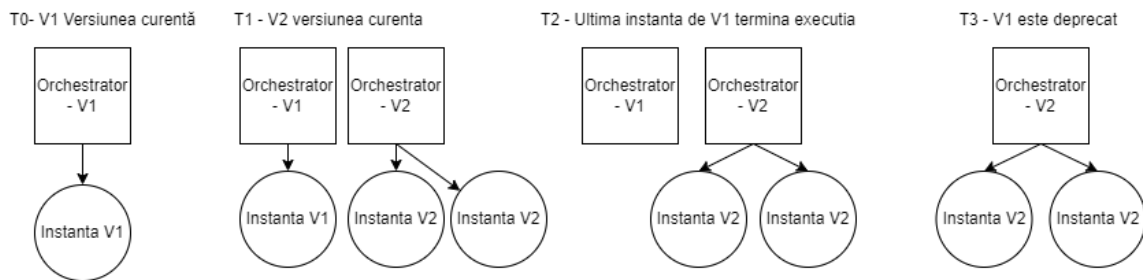


Figura 2.4: Ciclu de versionarea a codului de orchestrare

# Capitolul 3

## Workflow-uri de lungă durată - Analiză paralelă a arhitecturii

### 3.1 Descrierea problemei

Atunci când vine vorba de managementul unui workflow de lungă durată, sau a unei tranzacții distribuite, un exemplu clasic în literatura este cel al unei rezervări de călătorie. O rezervare de călătorie presupune mai multe lucruri :

- Rezervarea unui taxi
- Rezervarea hotelului
- Rezervarea avionului

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

Recuperarea dintr-un eșec într-o tranzacție distribuită nu se poate face în mod nativ, deoarece aplicațiile ce iau parte la aceasta sunt complet independente, astfel că fiecare sistem ce ia parte la o astfel de tranzacție trebuie să expună și o funcționalitate de roll-back ce va anula o operație ce a fost făcută în aceasta. Astfel, putem concluziona ca managementul unei tranzacții distribuite rezulta întotdeauna în managementul unui workflow de tip SAGA.

Saga este un concept arhitectural teoretic ce garantează execuția unui workflow într-un mod predictibil: Fie  $A_1, A_2, \dots, A_n$  acțiuni ce iau parte la o execuție Saga și

$RA_1, RA_2, \dots, RA_n$  acțiunile de rollback corespunzător fiecărui pas care ia parte la workflow. Un workflow de tip Saga garantează fie ca toți pașii  $A_1, A_2, \dots, A_n$  au fost executați cu succes, fie  $A_1, A_2, \dots, A_t, RA_1, RA_2, \dots, RA_t$  au fost executați cu succes. Astfel, se garantează fie ca toate acțiunile au fost executate cu succes, fie că toate acțiunile 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 părții practice a acestei lucrări este implementarea unui coordonator de tranzacții distribuite de lungă durată implementând o arhitectură Saga. Vor fi analizate atât dezvoltarea acestei aplicații folosind un framework care facilitează dezvoltarea orchestratoarelor, cât și dezvoltarea aplicației folosind tehnologii clasice (.net Web API combinat cu o bază de date relațională).

## 3.2 Implementarea arhitecturii în mod clasic

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

- posibilitatea ca procesul să fie închis chiar în timpul execuției
- posibilitatea ca unul din pași să eșueze din motive neimputabile sistemelor ce iau parte la acțiune (probleme de rețea)

### 3.2.1 Implementarea naivă

Vom porni de la următoarea secțiune 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 ideală această secțiune de cod ar fi o implementare corectă a unui SAGA, dar ce se poate întâmpla, dacă analizăm cazurile la care trebuie să fie rezilient un astfel de sistem, într-o lume distribuită ?

Ei bine, analizând prima potențială problema, și anume posibilitatea ca procesul ce execută Saga să fie oprit în mijlocul execuției putem evidenția potențialele probleme cu această abordare. Practic, dacă serverul este oprit oriunde între linia 4 și 14, va lăsa execuția într-o stare inconsistentă, cu posibilitatea de a avea consecințe serioase asupra utilizatorului. Ce se întâmplă dacă se execută doar rezervarea de Hotel, dar utilizatorul nu este notificat ? Acestuia îi va fi retrasă suma de pe card în mod abusiv, rezultând în posibile probleme legale pentru deținătorul aplicației. Astfel, garanția atomicității într-o tranzacție distribuită este foarte importantă.

Motivele pentru care această întrerupere a execuției practic face imposibilă garantarea atomicității întregii operații, este că toată acțiunea se execută ca parte a unei singure încercări, și orice întrerupere a acestuia face ca operația să nu poată fi reîncercată și eventual continuată, pentru a garanta execuția completă a strategiei Saga.

### 3.2.2 Execuție durabilă

O altă problemă cu codul inițial, este că nu acoperă nici posibilitatea ca unul din pași să eșueze din motive de probleme de rețea, iar astfel, toată operațiunea poate rămâne într-o stare necunoscută, dacă pe perioada roll-back-ului deci între 10 și 14, unul din pași eșuează. Pentru a acoperi ambele probleme, codul ar trebui să fie transformat după 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, dacă este prima execuție.
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 secvenței de cod, putem observa următoarele modificări :

- Introducerea unui nivel de decuplare, la nivelul bazei de date, ca permite re-execuția secvenței de cod fără a pierde starea precedentă, pentru a evita trimiterea mai multor request-uri prin rețea decât este necesar.
- Introducerea unui nou nivel de tratare a excepțiilor, care prinde eventualele probleme de rețea la nivelul acțiunii de rollback, și care marchează rezervarea ca eșuată, pentru a fi putea fi reparată în mod manual. (atunci când execuția Saga nu poate fi executată pe flow-ul de rollback, acțiunea trebuie). Astfel eventualele acțiuni care nu pot fi executate respectând garanțiile Saga, vor fi marcate ca mai apoi acele tranzacții să fie reparate prin operațiuni manuale.

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

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

### 3.2.3 Garanții de execuție. Cel puțin odată versus exact o dată

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

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

În varianta finală a acestei implementări componentele implicate sunt :

- Serviciul ce execută mașina 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 execuție durabilă

### **3.2.4 Interacțiunea din exterior cu workflow-ul. Operații 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 întâmplă însă dacă unul din sisteme poate oferi un răspuns foarte lent, spre exemplu în 24 de ore ? Implementarea curentă nu permite interacțiuni cu workflow-ul din exterior, deci toată arhitectura trebuie modificată pentru a cuprinde și acest caz.

Pentru a permite interacțiunea cu workflow-ul din exterior, sistemul va expune și endpointuri externe pentru a modifica starea unei rezervări. Astfel, sistemul ce implementează Saga doar va anunța 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) și va modifica în mod direct starea rezervării. La nivelul serviciului ce implementează mașina 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 execuția (return) și va aștepta următoarea procesare (care va fi inițiată 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 interacționează cu rezervarea vor putea modifica starea acestuia accesând endpoint-urile dedicate fiecărei operații, folosind id-ul rezervării primit pe request. De exemplu, dacă 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 tranziția rezervării în starea FlightBooked. Astfel, workflow-ul propriu zis nu va fi executat pe perioada în care se așteaptă finalizarea unei acțiuni din partea unui alt sistem.

### 3.2.5 Cronometre durabile.

Deși sistemul în această formă permite interacțiunea cu alte servicii, există o potențială vulnerabilitate, și anume dacă 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 rămâne pentru totdeauna într-o stare intermediară. Pentru a rezolva această problemă, trebuie să adăugăm suport pentru un cronometru durabil.

Cronometrul durabil este un concept teoretic într-o arhitectură orientată pe microservicii, și se referă la capacitatea unui cronometru de a rezista eventualelor întreruperi ale procesului. În cadrul sistemului nostru, pentru a obține un cronometru durabil elementar, pe entitatea rezervării se va salva și data la care a fost creat și data la care s-a făcut ultimul update. Astfel, orice cronometru durabil se obține prin compararea datei la care a fost creat/updatat plus un anumit delta, cu data curentă de la momentul execuției. Următoarea secvență de cod, ilustrează cum poate fi verificat în mod durabil dacă workflow-ul a tranziționat 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 că dacă workflow-ul nu tranziționează către o anumită stare (HotelBooked în acest caz) workflow-ul va porni strategia de rollback printr-o excepție. Astfel, sistemul devine rezilient la eventuale erori ce apar în serviciile ce iau parte la workflow, și este capabil să re-aducă workflow-ul într-o stare consistentă fără intervenții din exterior.

### 3.2.6 Optimizarea numărului de request-uri

Deși garanția de cel puțin o dată ne permite ca request-urile către celelate servicii să fie trimise de mai multe ori, beneficiind de faptul că acestea sunt idempotente, sistemul nu ar trebui să abuzeze și atât timp cât totul funcționează corect, sistemul ar trebui să trimită request-urile exact o dată. Însă, analizând codul de mai sus și extinzând

pattern-ul și la celelalte acțiuni, putem observa o problemă. Presupunând ca acțiunea de rezervare a unei camere de hotel durează 24 de ore, iar rezervarea noastră este preluată și procesată de către serviciul ce se ocupă de preluarea rezervărilor din baza de date o dată 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 inacceptabil, chiar și într-un sistem ce garantează execuția cel puțin odată a acțiunilor.

Pentru a rezolva această problemă vor fi introduse stări intermediare, ce vor evidenția natura de lungă durată a operațiunilor, și vor ajuta sistemul de gestiune al Saga să obțină execuția exact o dată a acțiunilor, 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 tranziție în entitatea rezervării permite sistemului să decidă să nu mai execute o anumită acțiune pe perioada în care așteaptă un răspuns din exterior, datorită naturii de lungă durată a operațiunilor.

### 3.3 Implementarea arhitecturii folosind durable Functions

## Capitolul 4

### Rezultate și Concluzii

#### 4.1 Concluzii

# Listă de figuri

1.1	Statistică ce evidențiază importanța domeniului cloud computing în ultimii ani . . . . .	4
2.1	Orchestrare versus Coreografie . . . . .	9
2.2	DTF Schema în DTF Sql Server provider . . . . .	10
2.3	Exemplu arhitectură folosind Durable Functions . . . . .	12
2.4	Ciclu de versionarea a codului de orchestrare . . . . .	13

## Listă de tabele