Titlul lucrării

PROIECT DE DIPLOMĂ

Autor: **Prenume NUME**

Conducător științific: **Titlu.ing. Prenume NUME**

|  |  |
| --- | --- |
| DECAN  **Prof.dr.ing. Liviu MICLEA** | Vizat,  DIRECTOR DEPARTAMENT AUTOMATICĂ  **Prof.dr.ing. Honoriu VĂLEAN** |

Autor: **Prenume NUME**

Titlul lucrării

1. **Enunţul temei:** *O scurtă descriere a temei proiectului de diplomă*
2. **Conţinutul proiectului:** *(enumerarea părților componente) Pagina de prezentare, Declarație privind autenticitatea proiectului, Sinteza proiectului, Cuprins, Titlul capitolului 1, Titlul capitolului 2,… Titlul capitolului n, Bibliografie, Anexe.*
3. **Locul documentaţiei:** *Universitatea Tehnică din Cluj-Napoca, alte locuri dacă este cazul*
4. **Consultanţi:** *ing. Prenume Nume (dacă este cazul)*
5. **Data emiterii temei:**
6. **Data predării:**

Semnătura autorului

Semnătura conducătorului științific

**Declaraţie pe proprie răspundere privind**

**autenticitatea proiectului de diplomă**

Subsemnatul(a) **Prenume NUME**  , legitimat(ă) cu CI/BI seria nr. , CNP ,

autorul lucrării:

elaborată în vederea susţinerii examenului de finalizare a studiilor de licență la **Facultatea de Automatică și Calculatoare**, specializareaChoose an item.**,** din cadrul Universităţii Tehnice din Cluj-Napoca, sesiunea Choose an item. a anului universitar 2017-2018, declar pe proprie răspundere, că această lucrare este rezultatul propriei activităţi intelectuale, pe baza cercetărilor mele şi pe baza informaţiilor obţinute din surse care au fost citate, în textul lucrării, şi în bibliografie.

Declar, că această lucrare nu conţine porţiuni plagiate, iar sursele bibliografice au fost folosite cu respectarea legislaţiei române şi a convenţiilor internaţionale privind drepturile de autor.

Declar, de asemenea, că această lucrare nu a mai fost prezentată în faţa unei alte comisii de examen de licenţă.

In cazul constatării ulterioare a unor declaraţii false, voi suporta sancţiunile administrative, respectiv, *anularea examenului de licenţă*.

Data Prenume NUME

(semnătura)

**SINTEZA**

proiectului de diplomă cu titlul:

Titlul lucrării

Autor: **Prenume NUME**

Conducător științific: **Titlu.ing. Prenume NUME**

1. Cerinţele temei:

2. Soluţii alese:

3. Rezultate obţinute:

4. Testări şi verificări:

5. Contribuţii personale:

6. Surse de documentare:

Semnătura autorului

Semnătura conducătorului științific

Cuprins

[1 Introducere 2](#_Toc477457095)

[1.1 Context general 2](#_Toc477457096)

[1.2 Obiective 2](#_Toc477457097)

[1.3 Specificații 2](#_Toc477457098)

[2 Studiu bibliografic 3](#_Toc477457099)

[3 Analiză, proiectare, implementare 4](#_Toc477457100)

[4 Concluzii 5](#_Toc477457101)

[4.1 Rezultate obținute 5](#_Toc477457102)

[4.2 Direcții de dezvoltare 5](#_Toc477457103)

[5 Reguli de formatare 6](#_Toc477457104)

[5.1 Formatarea paginii 6](#_Toc477457105)

[5.2 Titluri și stiluri 6](#_Toc477457106)

[5.3 Figuri, tabele și ecuații 7](#_Toc477457107)

[5.3.1 Figuri 7](#_Toc477457108)

[5.4 Tabele 7](#_Toc477457109)

[5.5 Ecuații 7](#_Toc477457110)

[5.6 Referințe bibliografice 8](#_Toc477457111)

[6 Bibliografie 9](#_Toc477457112)

# Introducere

## Context general

Conceptul inițial de “blockchain” a fost introdus de o persoană (sau un grup de persoane) cunoscută sub pseudonimul de Satoshi Nakamoto în 2008, pentru a servi ca registru public de tranzacții al criptomonedei bitcoin. Lucrarea publicată sub acest pseudonim (și care avea să servească mai târziu drept bază pentru crearea bitcoin și a altor criptomonede), intitulată “Bitcoin: A Peer-To-Peer Electronic Cash System”, menționa următoarele:

“Comerțul pe internet a ajuns să se bazeze aproape exclusiv pe instituțiile financiare care servesc ca părți terțe de încredere pentru a procesa plățile electronice. În timp ce sistemul funcționează suficient de bine pentru majoritatea tranzacțiilor, acesta suferă încă de slăbiciunile inerente ale modelului bazat pe încredere.”

Invenția blockchain-ului a lansat bitcoin ca prima monedă digitală care avea să rezolve problema dublei cheltuieli fără a avea nevoie de o autoritate de încredere sau de un server central [1]. Acest concept a fost cunoscut mai târziu ca blockchain 1.0. Cota de piață a bitcoin se situează acum între 10 și 20 de miliarde de dolari și este utilizat de milioane de oameni pentru plăți online.

În lucrările care au fost publicate după lansarea blockchain-ului, acesta a fost descris ca o tehnologie în care contractele, sarcinile și plățile, sub formă de înregistrări digitale, sunt stocate în baze de date distribuite folosind o semnătură unică unde pot fi identificate, validate, distribuite și protejate de orice formă de manipulare. Acest sistem permitea persoanelor, organizațiilor și mașinilor să interacționeze liber fără necesitatea intermediarilor cum ar fi avocații, brokerii și bancherii.

Se speculează că, în viitor, mașinile autonome și dronele vor folosi blockchain-uri pentru a plătii pentru servicii precum stațiile de încărcare și suporturile de aterizare. Durata transferurilor valutare internaționale se va reduce de la zile, la o oră și apoi la câteva minute cu un grad de fiabilitate mai mare decât sistemul actual a fost capabil să gestioneze.

Ideile publicate de Satoshi Nakamoto în lucrarea inițială au fost preluate de Ethereum și au fost folosite pentru a pune bazele pentru blockchain 2.0. Ethereum a construit un blockchain cu un limbaj de programare integrat, de tip Turing-complete, care permite oricui să scrie contracte inteligente unde își pot definii propriile reguli de proprietate, formatul tranzacțiilor și funcții de tranziție a stării.

Contractele inteligente ajută la rezolvarea problemei neîncrederii dintre părțile implicate într-o tranzacție sau dintre parteneri de afaceri. De exemplu, dacă compania A vinde acțiuni sau orice alt produs companiei B, iar părțile nu au încredere reciprocă în mod tipic ele vor alege o autoritate de încredere care să le poată ajuta în această situație. Dacă în schimb ele decid să folosească contracte inteligente, compania A poate trimite acțiunile (criptate în codul contractului) cu condiția să fie decriptate numai după ce compania B plătește costul acțiunilor. Algoritmul va verifica soldul contului și apoi va semna tranzacția decriptând acțiunile. Contractele inteligente pot fi utilizate în multe situații de zi cu zi, dar cel mai mare potențial este în domeniul financiar.

Pasul către următoarea generație, blockchain 3.0, a fost introducerea aplicațiilor descentralizate susținute de rețele blockchain private astfel încât acestea să fie capabile să gestioneze tranzacțiile comerciale globale ale companiilor majore din domeniul tehnologic, financiar și al lanțurilor de aprovizionare. Proiectul Hyperledger fondat de Linux Foundation și susținut în prezent de IBM oferă companiilor soluții blockchain complexe pentru eficientizarea business-ului în domeniile lor de activitate.

Am ales această temă datorită interesului personal față de acest subiect dar și pentru a prezenta avantajele folosirii unei soluții de tip blockchain în locul unei soluții clasice de stocare a datelor, pentru a aduce informații referitoare la modul de funcționare a acestei tehnologii și pentru a oferii un exemplu practic în care este folosită într-un domeniu de interes al industriei, managementul unui lanț de aprovizionare, cu argumente referitoare la tehnologiile folosite, strategiile de implementare și tool-urile existente.

Primul capitol este o introducere a ceea ce înseamnă blockchain, cum a apărut și cum a evoluat ulterior. Cel de-al doilea capitol prezintă informații referitoare la structura și modul de funcționare al blockchain-urilor, mai exact, părțile componente ale unui bloc, folosirea criptografiei pentru a securiza tranzacțiile, tipuri de rețele blockchain, avantajele și dezavantajele folosirii lor, și se discută despre contribuțiile pe care le-au adus Bitcoin, Ethereum și Hyperledger acestei tehnologii și problemele, aparținând sistemelor de plată digitale sau sistemelor distribuite, pe care le-au rezolvat. Cel de-al treilea capitol prezintă metodologiile folosite în ingineria software pentru dezvoltarea unei aplicații de management a unui lanț de aprovizionare. În prima parte a capitolului se discută despre cerințele pe care trebuie să le îndeplinească această aplicație, atât din punct de vedere al business-ului cât și din punct de vedere al performanțelor. În cea de-a doua parte se discută despre arhitectura aleasă și motivele care au condus la această alegere iar în ultima parte se discută despre implementarea efectivă a aplicației, despre tehnologiile folosite (Hyperledger Fabric, Hyperledger Composer, Node.js, Angular, etc.) modul de funcționare al acestor tehnologii și posibilitățile pe care le oferă. În cel de-al patrulea capitol sunt oferite concluzii referitoare la alegerea tehnologiei blockchain ca mediu de stocare a datelor în managementul unui lanț de aprovizionare și validarea cerințelor determinate inițial în implementarea finală. Ultimul capitol este bibliografia.

## Obiective

Această lucrare își propune să ofere o introducere cuprinzătoare a aspectelor teoretice și practice ale tehnologiei blockchain. Informațiile prezentate au un caracter introductiv, dar sunt necesare cunoștințe de bază despre ingineria software și etapele dezvoltării unei aplicații software pentru a înțelege.

În vederea realizării scopului anterior au fost stabilite următoarele obiective, a căror analiză detaliată se regăsește în conținutul lucrării:

* prezentarea conceptelor de bază ale sistemelor distribuite pe care se bazează tehnologia blockchain;
* istoricul, definițiile, caracteristicile, tipurile și beneficiile blockchain-urilor împreună cu mecanismele de consens;
* identificarea și analiza conceptelor descentralizării și relația lor cu tehnologia blockchain;
* criptografia, folosind chei publice și private, cu exemplificarea principalelor proprietăți ale unei funcții hash criptografice;
* prezentarea detaliilor de funcționare și a conceptelor tehnice referitoare la criptomoneda bitcoin;
* principiile, caracteristicile și componentele care stau la baza platformei Ethereum;
* definiția și aspectele teoretice ale contractelor inteligente;
* informații referitoare la proiectul Hyperledger fondat de Linux Foundation, care includ proiectele existente care folosesc și dezvoltă tehnologia blockchain.

Aplicația dezvoltată în cadrul acestei lucrări este un exemplu practic de utilizare a tehnologiei blockchain într-un domeniu de interes al industriei, managementul unui lanț de aprovizionare, care folosește conceptele prezentate în capitolele teoretice și oferă utilizatorilor posibilitatea de a interacționa în mod direct cu blockchain-ul. Pentru extragerea concluziilor se va testa aplicația și se vor evalua rezultatele obținute.

## Specificații

Pentru realizarea acestor obiective se va implementa o aplicație care modelează un caz de utilizare a tehnologiei blockchain în domeniul industrial, managementul unui lanț de aprovizionare pentru bunuri perisabile. Sarcinile și acțiunile în care sunt implicați participanții acestui proces vor fi transformate în cerințe funcționale ale aplicației. Instrumentele oferite de Hyperledger vor fi folosite pentru modelarea participanților, pentru implementarea contractelor inteligente sub formă de tranzacții și pentru susținerea rețelei de afaceri. Comunicarea se va realiza doar prin intermediul tranzacțiilor pentru a ține o evidență a tuturor operațiilor de manipulare a blockchain-ului. Rețeaua va fi modelată folosind Hyperledger Composer și susținută folosind Hyperledger Fabric și trebuie să ofere atât funcționalitatea de autentificare cât și de autorizare a participanților.

Se va genera un server REST pe baza definiției rețelei prin intermediul căruia este posibilă comunicarea cu blockchain-ul (execuția tranzacțiilor). Endpoint-urile expuse de serverul REST vor fi apelate de către utilizatori din interfața grafică. Performanțele cum ar fi numărul de solicitări pe care le poate procesa într-un interval și timpul de răspuns trebuie să fie cât mai apropiate de un mediu de producție. Se va lua în calcul posibilitatea scalării server-ului REST.

Se vor implementa trei aplicații grafice, folosind Angular ca framework de dezvoltare, corespunzătoare celor trei participanți ai procesului (cumpărător, expeditor, producător). Cele trei aplicații vor implementa funcționalitățile stabilite pe baza cerințelor aplicației. Implementarea finală oferă posibilitatea de a simula ciclul de viață al unui produs de la producător la cumpărător. Se va folosi același utilizator pentru simularea întregului proces (administrator). Pentru a respecta cerințele de performanță se va impune un timeout solicitărilor trimise din cele trei aplicații (<= x ms).

La final se vor extrage concluziile pe baza informațiilor oferite de instrumentele de logare.

# Studiu bibliografic

## Blockchain

Există mai multe definiții referitoare la ceea ce înseamnă conceptul de blockchain, depinzând de perspectiva din care acesta este privit.

Din punct de vedere comercial, un blockchain poate fi definit ca o platformă prin care nodurile participante pot să facă schimb de valori fără a fi nevoie de o autoritate centrală de încredere. Acest lucru permite blockchain-ului să fie un mecanism de consens descentralizat, în care nicio autoritate nu este responsabilă de baza de date.

Din punct de vedere tehnic, un blockchain este un registru distribuit care este reprodus în mai multe noduri și care permite păstrarea evidenței tranzacțiilor într-un mod imutabil, transparent și securizat prin intermediul criptografiei.

Participanții la rețea guvernează și sunt de acord prin consens cu privire la actualizările înregistrărilor din registru. În acest proces nu este implicată nicio autoritate centrală cum ar fi o instituție financiară. Fiecare înregistrare are o marcă de timp și o semnătura criptografică unică, făcând astfel registrul o istorie verificabilă și imutabilă a tuturor tranzacțiilor din rețea.

Tranzacțiile confirmate și validate sunt înscrise într-un bloc din cadrul blockchain-ului. Un bloc este pur și simplu o colecție de tranzacții aduse împreună pentru a le organiza într-un mod logic. Structura unui bloc este dependentă de tipul și designul blockchain-ului, dar, în general, există câteva atribute care sunt esențiale pentru funcționalitatea acestuia:

* *index* – este poziția blocului curent în cadrul blockchain-ului;
* *timestamp* – este determinat automat și se referă la momentul de timp când datele au fost scrise;
* *data* – reprezintă înregistrarea de date pentru blocul curent;
* *hash* – identificator unic pentru blocul curent, criptat folosind algoritmul SHA256;
* *previous hash* – identificator unic pentru părintele blocului curent, criptat folosind algoritmul SHA256;

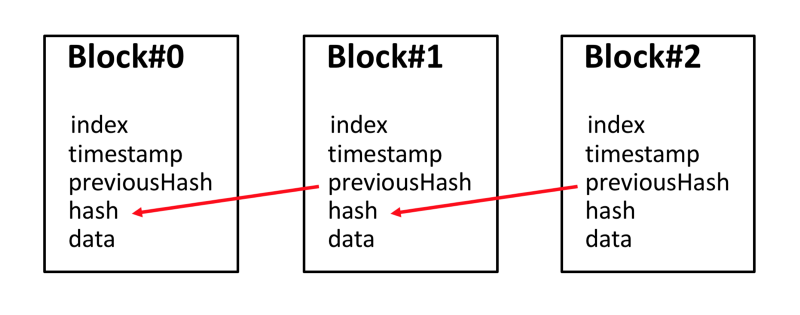


Figura 2.1 Structura generală a blockchain-ului

Din figura 2.1 se poate observa legătura logică între blocurile individuale ale blockchain-ului și tipul de structură de date pe care acesta îl reprezintă. Este o listă simplu înlănțuită care folosește pointeri de tip hash în loc pointerilor de tip adresă.

Fiecare bloc din cadrul blockchain-ului este identificat printr-un hash, generat folosind algoritmul criptografic SHA256 pe antetul blocului. Fiecare bloc se referă, de asemenea, la un bloc anterior, cunoscut ca blocul părinte, prin câmpul *previous hash* din antet. Cu alt cuvinte, fiecare bloc conține hash-ul părintelui său în propriul antet. Atunci când datele părintelui se modifică în orice fel, hash-ul părintelui se schimbă. Hash-ul modificat al părintelui necesită o modificare a câmpului *previous hash* al copilului care pointează către părinte. Această modificare determină ca hash-ul copilului să se shimbe fiind calculat cu hash-ul părintelui și așa mai departe. Identitatea proprie a copilului se schimbă dacă identitatea părintelui se schimbă. Acest efect cascadă asigură că, odată ce un bloc are multe generații care îl succed, acesta nu poate fi modificat fără a obliga în consecință la o recalculare a tuturor blocurilor ulterioare. Deoarece această recalculare ar necesita o putere enormă de calcul, existența unui lanț lung de blocuri fortifică ideea de imutabilitate.

Secvența hash-urilor care leagă fiecare bloc de părintele său creează un lanț care merge înapoi până la primul bloc creat vreodată cunoscut sub numele de *bloc de geneză*.

Pentru a înțelege cum se formează acestă legătura logică dintre blocuri și care e rolul hash-ului se va discuta în continuare despre conceptele, teoria și aspectele practice ale criptografiei în contextul tehnologiei blockchain.

### Criptografia

Criptografia sau criptologia este practica și studiul tehnicilor de comunicare securizată în prezența unor terțe părți numite adversari [2]. În general, criptografia constă în construirea și analizarea de protocoale care împiedică terții sau publicul să citească mesaje private și acoperă diferite aspecte ale securității informațiilor, cum ar fi:

* confidențialitatea – este asigurarea ca informațiile sunt disponibile numai entităților autorizate;
* integritatea – este asigurarea că informațiile pot fi modificate numai de entități autorizate;
* autentificarea – este asigurarea identității unei entități sau validitatea unui mesaj.

Hash-ul criptografic este folosit pentru a atribui o identitate unică unui bloc individual din cadrul blockchain-ului. În termeni simplii, o funcție hash transformă o intrare de orice lungime într-o ieșire de lungime fixă printr-un algoritm matematic bazat pe o funcție criptografică. Bitcoin utilizează spre exemplu algoritmul SHA256.

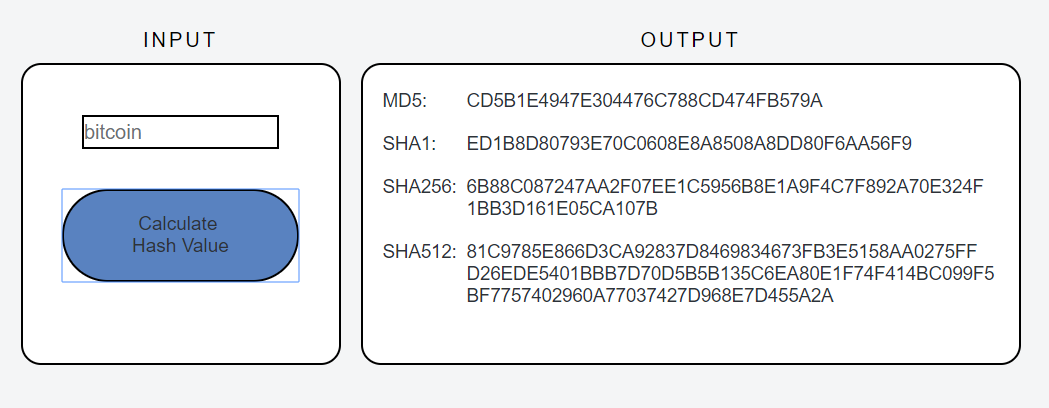


Figura 2.2 Rezultatul aplicării unei funcții hash

În cazul SHA256 indiferent de lungimea intrării ieșirea va avea întotdeauna o lungime fixă de 256 de biți. Acest lucru are o importanță deosebită atunci când avem de a face cu o cantitate mare de date sau tranzacții. În loc să reținem datele de intrare, care ar putea avea o dimensiune mare, putem reține și lucra mai departe cu hash-ul acestor date.

O aplicație practică a acestei funcții este de a stoca parole. Orice parolă aparținând unui cont de utilizator creat prin intermediul unui serviciu web este, în general, criptată folosind un algoritm criptografic și rezultatul este stocat în baza de date. Pentru autentificare, același algoritm este aplicat pe cuvântul introdus și serverul verifică dacă rezultatul corespunde cu cel stocat.

O clasă specială de funcții hash sunt funcțiile hash criptografice. Acestea au o serie de proprietăți fiind ideale pentru criptografie.

1. **Determinism**

Această proprietate înseamnă că, indiferent de câte ori este procesată o anumită valoare de intrare, ea trebuie să genereze întotdeauna același hash. Cu alte cuvinte, trebuie să fie o funcție a datelor care urmează să fie procesate, în sensul matematic al termenului.

1. **Cost scăzut**

O funcție hash ar trebui să fie capabilă să returneze rapid hash-ul unei valori de intrare. Dacă procesul nu este rapid, sistemul nu va fi eficient.

1. **Rezistența la pre-imagine**

Fie X valoarea intrării și F(X) rezultatul aplicării funcției hash-ul asupra acestei intrări. Rezistența la pre-imagine presupune că este imposibil de determinat valoarea lui X dacă se cunoaște valoarea lui F(X).

Singura modalitate prin care se poate determina valoarea lui X este de a rula funcția pentru toate posibilele valori a le lui X și de a compara rezultatele cu valoarea lui F(X).

Se consideră următorul exemplu:

Se presupune că pe fiecare față a unui zar este înscris hash-ul numărului corespunzător feței în locul valorii acestuia. Cum se poate determina numărul inițial la aruncarea zarului? Fiind un număr relativ mic de intrări se poate calcula hash-ul fiecărui număr și se poate compara cu rezultatul de la aruncarea zarului pentru determinarea numărului. Deoarece funcțiile hash sunt deterministe, hash-ul unei anumite intrări va fi întotdeauna același, astfel încât se pot compara hash-urile pentru a determina intrarea originală.

Această metodă funcționează doar când cantitatea de date este relativ mică. Dacă se presupune un hash de 256 de biți, singura metodă prin care s-ar putea determina valoarea inițială corespunzătoare acestui hash este prin metoda “brute-force”. Aceasta presupune că se ia o intrare în mod aleator, se determină hash-ul acestei intrări și se compară cu hash-ul țintă până la întâlnirea unei potriviri.

În cel mai bun caz rezultatul este obținut din prima încercare. Șansele ca acest lucru să se întâmple sunt aproape imposibile.

În cel mai rău caz rezultatul este obținut după încercări. Practic, înseamnă că răspunsul a fost găsit la sfârșitul tuturor datelor.

În cazul cel mai probabil, rezultatul este obținut undeva la jumătatea numărului total de încercări, deci .

1. **Modificări mici ale intrării determină modificarea hash-ului**

Chiar dacă se face o mică modificare a intrării, cum ar fi înlocuirea unui singur caracter, schimbările reflectate asupra hash-ului vor fi uriașe.

Un exemplu pentru această proprietate:

51609286FB7F6089E0A0A418355949C791E84870AE2523093BA00BB3ECFF7F8E reprezintă textul “măr”.

C9D9D2C4BE976B707A1ACDCABA8495E7E0A7089579DA6625A71083D5B0C6BD reprezintă textul “păr”.

Aceasta este o proprietate critică deoarece conduce la una dintre cele mai importante calități ale blockchain-ului, imutabilitatea.

1. **Rezistența la coliziune**

Fiind date două intrări diferite X și Y unde F(X) și F(Y) sunt hash-urile lor respective, este aproape imposibil ca F(X) să fie egal cu F(Y). Ceea ce înseamnă că, în cea mai mare parte, fiecare intrare va avea propriul său hash unic.

### Sisteme distribuite

Înțelegerea sistemelor distribuite este esențială pentru a înțelege blockchain-ul deoarece, din punct de vedere arhitectural, blockchain-ul este un sistem distribuit. Mai exact, este un sistem distribuit descentralizat.

Sistemele distribuite reprezintă o paradigmă de calcul prin care două sau mai multe noduri lucrează unul cu celălalt în mod coordonat pentru a obține un rezultat comun și sunt modelate astfel încât utilizatorii finali o văd ca o singură platformă logică [3].

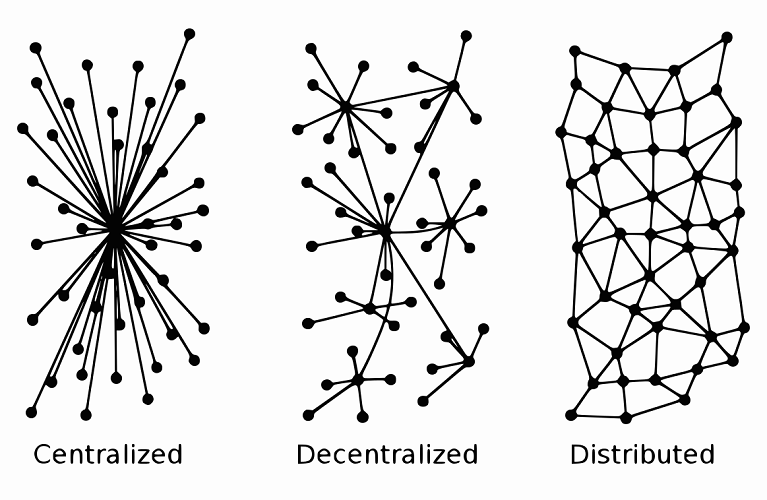


Figura 2.3 Tipuri de arhitecturi ale sistemelor

Pentru a obține un rezultat comun este esențial ca aceste sisteme să fie organizate în mod corespunzător. Există diferite modalități de a vedea organizarea unui sistem distribuit, dar una evidentă este aceea de a face o distincție între organizarea logică a componentelor software și realizarea fizică a sistemului. Organizarea sistemelor distribuite se referă în principal la componentele software care constituie sistemul.

În figura 2.3 sunt reprezentate diferitele tipuri de arhitecturi software, numite și arhitecturi ale sistemelor, care există în prezent, și anume: *centralizate*, *descentralizate* și *distribuite* [4]. Arhitecturile software ne indică modul în care ar trebui să fie organizate diferitele componente software și modul în care ar trebui să interacționeze.

În sistemele centralizate există o singură autoritate centrală sau un server central și toate celelalte noduri se comportă ca entități sau clienți care acceptă mesaje și acționează în consecință.

În sistemele descentralizate există mai multe servere secundare care primesc mesaje de la un server central. Nodurile individuale sunt conectate la serverele secundare. Cu toate acestea, în unele sisteme toate serverele pot fi egale în ierarhie, fără necesitatea unui server central.

În sistemele distribuite nu există un server central. Fiecare nod este conectat la fiecare alt nod și are exact aceeași autoritate. În ceea ce privește calculul sistemelor distribuite, puterea de procesare a fiecărui nod poate varia într-o mare măsură.

Principalele avantaje ale sistemelor distribuite sunt:

* putere de calcul mai mare;
* reducerea costurilor;
* fiabilitate mai mare;
* scalabilitate.

Puterea de calcul a unui sistem distribuit este rezultatul combinării puterii de calcul a tuturor nodurilor conectate. Prin urmare, sistemele distribuite au de obicei mai multă putere de calcul decât fiecare nod individual. Acest lucru a fost dovedit adevărat chiar și atunci când se compară sisteme distribuite compuse din calculatoare cu putere de calcul relativ scăzută.

Deoarece sistemele distribuite sunt compuse din mai multe calculatoare, costurile inițiale ale sistemelor distribuite sunt mai mari decât costurile inițiale ale calculatoarelor individuale. Totuși, costurile de creare, întreținere și operare a unui super-calculator sunt însă mult mai mari decât costurile unui sistem distribuit. Acest lucru este valabil mai ales pentru că înlocuirea componentelor individuale ale unui sistem distribuit se poate face fără un impact global semnificativ asupra sistemului.

Creșterea fiabilității unui sistem distribuit se bazează pe faptul că întreaga rețea poate continua să funcționeze chiar și atunci când mașinile individuale se defectează. Un sistem distribuit, spre deosebire de unul centralizat, nu are un singur punct de eșec. Dacă un element se defectează, elementele rămase pot prelua sarcinile de la acesta.

Scalabilitatea reprezintă capacitatea sistemului de a gestiona cu ușurință o cantitate tot mai mare de activități. Sistemele distribuite pot fi scalate *pe verticală,* prin adăugarea de resurse unui nod individual sau prin îmbunătățirea software-ului existent pentru a face față mai multor activități, sau *pe orizontală,* prin adăugarea de mai multe noduri la o rețea.

Principalele dezavantaje ale sistemelor distribuite sunt:

* costuri de coordonare;
* costuri de comunicare;
* complexitate mai mare;
* probleme de securitate.

Sistemele de calcul distribuite nu au o entitate centrală care să coordoneze participanții. Prin urmare, coordonarea trebuie făcută de participanții sistemului în sine. Această coordonare necesită efort și putere de calcul care nu mai pot fi folosite pentru sarcina reală.

Coordonarea necesită comunicare. Prin urmare, calculatoarele care formează un sistem distribuit trebuie să comunice între ele. Este necesară existența unui protocol de comunicație prin care se pot trimite, recepționa și procesa mesaje și care, la rândul lui, necesită efort și putere de calcul.

Rezolvarea unei probleme de calcul implică scrierea programelor și a software-ului. Datorită dezavantajelor menționate anterior, orice software dintr-un sistem distribuit trebuie să rezolve probleme suplimentare, cum ar fi coordonarea, comunicarea și utilizarea rețelelor. În mod implicit aceste sarcini suplimentare cresc complexitatea software-ului.

Trimiterea de informații printr-o rețea implică probleme de securitate deoarece entitățile care nu sunt de încredere pot utiliza în mod abuziv rețeaua pentru a accesa și exploata informațiile. Prin urmare, orice sistem distribuit trebuie să abordeze problemele legate de securitate.

O arhitectură de rețea distribuită poate fi numită *rețea de tip peer-to-peer* (P2P) dacă participanții împărtășesc o parte din propriile resurse hardware (putere de procesare, capacitate de stocare, lățime de bandă, etc.). Aceste resurse partajate sunt necesare pentru a furniza serviciile și conținutul oferit de rețea (de exemplu partajarea fișierelor sau partajarea spațiilor de lucru). Aceste resurse sunt direct accesibile de către ceilalți membrii ai rețelei fără a avea niciun punct central de coordonare. Participanții la o astfel de rețea sunt astfel furnizori de resurse precum și solicitanți de resurse [5].

### Problema generalilor bizantini

În 1982 un experiment de gândire a fost propus de *Lamport et al.* prin care un grup de generali care conduc diferite părți ale armatei bizantine intenționează să atace sau să se retragă dintr-un oraș. Singura modalitate de comunicare între ei este prin intermediul mesagerilor și trebuie să fie de acord să atace sau să se retragă în același timp pentru a câștiga. Dacă toți generalii ar fi demni de încredere atunci soluția problemei ar fi simplă. Cu toate acestea, unii dintre mesageri și chiar câțiva generali sunt trădători sau chiar soldați inamici. Există o șansă foarte mare ca aceștia să comunice un mesaj înșelător. Prin urmare, este necesar să se găsească un mecanism viabil care să permită un acord între generali chiar și în prezența unor trădători, astfel încât decizia să poată avea loc în același timp.

Această problema a fost rezolvată în 1999 de Castro și Liskov, care au prezentat algoritmul **Practical Byzantine Fault Tolerance (PBFT)**. Mai târziu, în 2009, prima implementare practică a fost realizată cu invenția bitcoin unde algoritmul **Proof-Of-Work (POW)** a fost dezvoltat ca un mecanism de realizare a consensului.

Consensul înseamnă înțelegerea colectivă sau procesul decizional al nodurilor majoritare asupra unei stări finale a datelor. Este foarte ușor de ajuns la un acord între două noduri (de exemplu, în sistemele client-server), dar când mai multe noduri participă la un sistem distribuit și trebuie să convină asupra unei singure valori, devine foarte dificil să se ajungă la un consens. Acest concept de realizare a unui consens între noduri multiple este cunoscut sub numele de consens distribuit.

Un mecanism de consens este un set de pași care trebuie toate sau de cele mai multe noduri pentru a conveni asupra unei stări sau a unei valori propuse.

Există diferite cerințe pentru a oferi rezultatele dorite într-un mecanism de consens:

* acord – toate nodurile cinstite decid asupra aceleiași valori;
* încheiere – toate nodurile cinstite termină executarea procesului de consens și, în cele din urmă, ajung la o decizie;
* valabilitate – valoarea convenită de toate nodurile cinstite trebuie să fie aceeași cu valoarea inițială propusă de cel puțin un nod cinstit;
* toleranța la defecte – algoritmul de consens trebuie să poată fi executat în prezența unor noduri defecte (noduri bizantine);
* integritate – nodurile iau decizii o singură dată într-un singur ciclu de consens.

### Rețele blockchain publice sau private

Singura distincție între cele două categorii este legată de cine are dreptul să participe la rețea, să execute protocolul de consens și să mențină registrul comun.

O rețea blockchain publică este complet deschisă și oricine poate participa ca un nod în procesul de luare a deciziilor. Utilizatorii pot fi sau nu recompensați pentru participarea lor. Aceste rețele nu sunt deținute de nimeni și încurajează participarea. Toate nodurile mențin o copie locală a registrului și folosesc un mecanism de consens distribuit pentru a ajunge la o decizie cu privire la eventuala stare a registrului. Bitcoin este una dintre cele mai mari rețele publice de astăzi.

Unul dintre dezavantajele unei rețele publice este cantitatea de putere computațională necesară pentru a menține un registru distribuit la scară largă.

Un alt dezavantaj îl reprezintă faptul că tranzacțiile sunt transparente, pot fi verificate de către oricine, ceea ce implică foarte puțină intimitate și susține doar o noțiune slabă de securitate.

O rețea blockchain privată necesită o invitație ce trebuie validată fie de cel care administrează rețeaua fie de un set de reguli introduse de acesta. Întreprinderile care au înființat o rețea privată vor pune la punct și un proces de autentificare. Acest lucru plasează restricții cu privire la cine este permis să participe la rețea și să execute tranzacții. Participanții trebuie să obțină o invitație sau permisiunea de a se alătura. Mecanismul de control al accesului ar putea varia: participanții existenți ar putea decide viitorii participanți sau o autoritate centrală ar putea emite licențe de participare. O dată ce un nod s-a alăturat rețelei, acesta va juca un rol în menținerea registrului într-un mod descentralizat.

### Avantajele și dezavantajele tehnologiei blockchain

Numeroase beneficii ale tehnologiei blockchain sunt discutate în industrie și sunt propuse de către liderii de opinie din întreaga lume. Printre cele mai importante avantaje se enumeră:

**Descentralizare**

Este un concept de bază și un avantaj major. Nu este necesară o terță parte sau un intermediar pentru a valida tranzacțiile. Un mecanism de consens este folosit pentru a conveni asupra valabilității tranzacțiilor.

**Transparență și încredere**

Tranzacțiile sunt vizibile de către toți participanții care mențin o copie locală a registrului. Acest lucru permite ca sistemul să fie transparent și ca urmare să se stabilească încrederea.

**Imutabilitate**

Tranzacțiile care au fost validate și distribuite în rețea nu mai pot fi modificate fără recalcularea în prealabil a blocurilor care le includ.

**Disponibilitate**

Deoarece sistemul se bazează pe mii de noduri dintr-o rețea peer-to-peer, iar datele sunt replicate și actualizate pe fiecare nod, disponibilitatea sistemului crește. Chiar dacă nodurile părăsesc rețeaua sau devin inaccesibile, rețeaua în ansamblul său continuă să funcționeze.

**Securitate**

Toate tranzacțiile înscrise în cadrul unei rețele blockchain sunt protejate cu ajutorul unor algoritmi criptografici care asigură integritatea datelor.

**Reducerea costurilor**

Deoarece nu există o autoritate centrală sau o terță parte de încredere, cum ar fi o instituție financiară, costurile generale sunt reduce prin eliminarea taxelor către intermediari.

Pe lângă avantajele oferite de tehnologia blockchain există și dezavantaje, dar se fac eforturi pentru a remedia aceste dezavantaje în versiunile ulterioare și pentru a oferii un sistem mai robust, mai util și mai accesibil. Printre cele mai mari provocări se enumeră:

* Scalabilitate
* Confidențialitate
* Securitate
* Consum mare de energie
* Posibilități de integrare

### Evoluție și tendințe de viitor

**Blockchain 1.0**

Implementarea tehnologiei registrelor distribuite a condus la aplicarea sa inițială și evidentă: criptomonedele. Acest lucru permite efectuarea de tranzacții financiare bazate pe tehnologia blockchain, bitcoin fiind cel mai bun exemplu din această categorie. Este folosit ca un sistem de plată digitală și este un registru care înregistrează tranzacțiile în timp real.

**Blockchain 2.0**

Vitalik Buterin este cel mai notabil colaborator la blockchain 2.0, prin fondarea Ethereum în 2013 (4 ani după prima generație), o platformă destinată contractelor inteligente. În esență, contractele inteligente sunt contracte care se auto-execută când una sau mai multe condiții au fost îndeplinite. Beneficiile pe care contractele inteligente le aduc sunt că reduc costurile de validare a datelor și execută sarcini umane intense în mod automat. Blockchain 2.0 este considerat revoluția care a făcut ca tehnologiile blockchain să ajungă în vârf. Odată cu introducerea contractelor inteligente mai multe sectoare, altele decât cele financiare, au început să utilizeze blockchain-ul. Înainte de Ethereum, blockchain-ul era folosit doar pentru a înregistra tranzacțiile. Datorită platformei Ethereum și a unor platforme similare, precum Lisk, toate sectoarele se pot dezvolta utilizând blockchain și contractele inteligente.

**Blockchain 3.0**

Această generație merge cu un pas mai departe de oferirea monedelor digitale și a contractelor inteligente prin introducerea *aplicațiilor descentralizate*. Se folosesc bazele puse de Ethereum pentru extinderea în domenii cum ar fi guvernul, educația, sănătatea, artele, științele, etc. O aplicație descentralizată (DApp) rulează pe o rețea descentralizată de tip peer-to-peer, are un număr nelimitat de participanți, este asemănătoare unui contract inteligent și are aplicabilități și înafara sectorului financiar.

## Bitcoin

Bitcoin reprezintă o colecție de concepte și tehnologii care formează baza unui sistem de plată digital. Este o combinație între o rețea de tip peer-to-peer, un protocol și software-ul care facilitează crearea și utilizarea monedei digitale numită bitcoin.

Monedele digitale sunt folosite pentru a cumpăra bunuri și servicii, pentru a stoca în siguranță valori sau pentru a transfera aceste valori oriunde în lume între participanții din rețeaua bitcoin. Utilizatorii rețelei comunică între ei folosind protocolul bitcoin, care este disponibil ca software open source pe o gamă largă de platforme.

Bitcoin este divizibil până la a opta zecimală și este complet digital, permițând transferul oricărei valori monetare. Spre deosebire de monedele guvernamentale, care pot fi manipulate și devalorizate, există o aprovizionare finită de aproximativ 21 de milioane de bitcoini, ceea ce îl face un lucru rar și valoros.

În ansamblu, este alcătuit din:

* o rețea descentralizată de tip peer-to-peer (protocolul bitcoin);
* un registru public folosit pentru înregistrarea tranzacțiilor (blockchain-ul);
* un set de reguli pentru validarea tranzacțiilor independente și emiterea unităților valutare (mecanismul de consens);
* un mecanism de atingere a consensului distribuit asupra unui bloc valid (algoritmul “Proof-Of-Work”).

Pentru a putea efectua tranzacții în rețeaua bitcoin, este nevoie de un “portofel” (eng. wallet), un program care permite stocarea și schimbul de bitcoin. Deoarece accesul la portofel ar trebui, în general, să fie limitat la o singură persoană, acesta este protejat printr-o metodă criptografică specială care utilizează o pereche unică de chei diferite, dar conectate: o cheie privată și una publică. Aceste chei sunt utilizate în tandem pentru scopuri specifice. În special:

* datele criptate cu cheia publică pot fi decriptate numai prin utilizarea cheii private;
* datele semnate cu cheia privată pot fi verificate utilizând cheia publică.

Dacă un mesaj este criptat cu o cheie publică specifică, numai proprietarul cheii private asociate va putea decripta și citi mesajul. Pe de altă parte, dacă un mesaj este criptat cu o cheie privată, numai cheia publică asociată poate fi utilizată pentru a-l decripta.

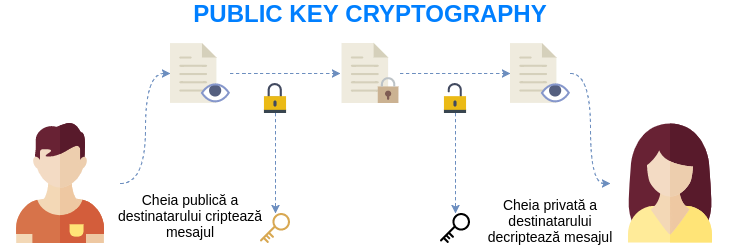


Figura 2.4 Criptarea și decriptarea unui mesaj folosind criptografia asimetrică

Cheia privată nu poate fi derivată din cheia publică, dar cheia publică poate fi derivată din cheia privată. Cheia publică este menită să fie împărtășită în siguranță, și, de obicei, poate fi expusă în mod liber oricui.

Când o solicitare de tranzacție este criptată cu cheia privată a portofelului, se generează o semnătura digitală utilizate de computerele din rețeaua bitcoin pentru a verifica sursa și autenticitatea tranzacției. Semnătura digitală este un șir de caractere care este rezultatul unei combinații între cererea de tranzacție și cheia privată, și, prin urmare, nu poate fi utilizată pentru alte tranzacții. Dacă se schimbă un singur caracter în cerere de tranzacție, semnătura digitală se va schimba, astfel încât niciun potențial atacator să nu poată altera cererea sau cantitatea de bitcoin tranzacționată.

Pentru a verifica soldul unui portofel, trebuie analizate și verificate toate tranzacțiile care au avut loc în întreaga rețea conectate la portofel. Această verificare a soldului este posibilă datorită legăturilor cu tranzacțiile anterioare. Pentru a simplifica și accelera procesul de verificare, o înregistrare specială a tranzacțiilor neprocesate este păstrată de nodurile din rețea. Aceasta este rezolvarea adusă de bitcoin la “problema dublei cheltuieli”.

### Problema dublei cheltuieli

Luând în considerare numeroasele aplicații practice pe care bitcoin le-a adus în domeniul informaticii, la baza sa stau 20 de ani de cercetare în domeniul monedelor digitale și 40 de ani de cercetare în domeniul criptografiei, de către mii de cercetători din întreaga lume. Bitcoin a adus o soluție practică la o problemă îndelungată cu monedele digitale: problema dublei cheltuieli. Până la introducerea criptografiei era posibilă copierea infinită a monedelor digitale, asemănător oricărui alt mijloc digital (posibilitatea de a salva un atașament dintr-un email de mai multe ori) și nu exista nicio modalitate de a confirma că un anumit lot de monede digitale nu a fost deja cheltuit fără existența unei autorități centrale. Era necesară existența unei terțe părți de încredere pentru gestionarea tranzacțiilor, care ținea un registru care să confirme că fiecare lot a fost cheltuit o singură dată.

Bitcoin a adus maselor un design funcțional și gata de a fi implementat care avea să răspundă la o întrebare fundamentală: cum poate fi rezolvată problema dublei cheltuieli fără ca o autoritate centrală să acționeze ca arbitru pentru fiecare tranzacție?

Problema dublei cheltuieli este un caz specific de procesare a tranzacțiilor. Tranzacțiile, prin definiție, trebuie să se întâmple sau nu. În plus, unele tranzacții (dar nu toate) trebuie să ofere garanția de a se întâmpla înainte sau după alte tranzacții (cu alte cuvinte ele trebuie să fie atomice). Lipsa atomicității este exact cauza acestei probleme: “cheltuirea” sau trimiterea de bani de la un participant A la un participant B, trebuie să se întâmple într-un anumit moment și înainte și după oricare alte tranzacții. Dacă nu s-ar întâmpla acest lucru, ar fi posibilă cheltuirea banilor mai mult decât o dată în tranzacții separate, dar simultane.

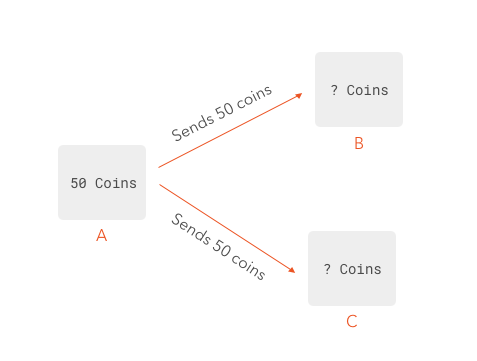


Figura 2.5 Posibilitatea cheltuirii a 50 de monede în tranzacții simultane

Când vine vorba de operațiunile monetare de zi cu zi, tranzacțiile sunt, de regulă, arbitrate de bănci. Atunci când un utilizator se conectează la sistemul său bancar și efectuează un transfer, banca se asigură că toate operațiunile anterioare și viitoare sunt consistente.

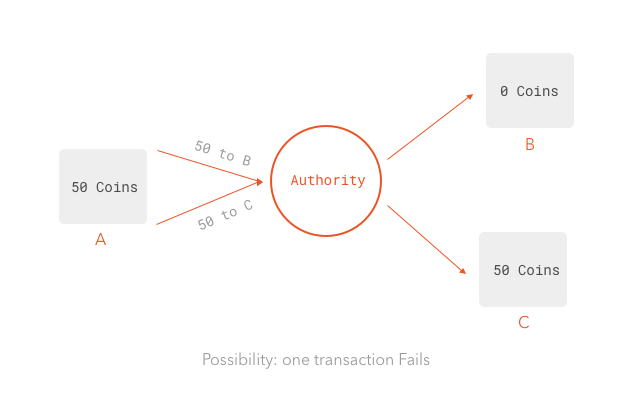


Figura 2.6 Tranzacții simultane arbitrate de o autoritate centrală

Deci, principala problemă pe care orice sistem monetar trebuie să o abordeze este: cum trebuie ordonate tranzacțiile atunci când nu există o autoritate centrală? În plus, nu trebuie să existe îndoieli cu privire la validitatea secvenței de tranzacții anterioare. Ca un sistem monetar să reușească, nu poate exista nicio modalitate prin care participanții să poată modifica tranzacțiile anterioare. Cu alte cuvinte, trebuie să existe și un “proces de verificare” a tranzacțiilor. Pentru a înțelege aspectele precum și modul de funcționare al acestui proces de verificare sunt necesare informații legate de structura specifică a blocurilor din rețeaua bitcoin.

### Structura și particularitățile blocurilor din rețeaua bitcoin

Un bloc este o structură de date care înglobează tranzacțiile pentru includerea lor în blockchain după procesul de validare. Blocul este alcătuit dintr-un antet care conține metadate, urmat de o listă lungă de tranzacții care compun cea mai mare parte a dimensiunii sale. Antetului blocului are o dimensiune de 80 de octeți și conține, în medie, 500 de tranzacții. Tabelul 2.1 descrie structura unui bloc.

Tabel 2.1. Structura unui bloc

| Dimensiune | Câmp | Descriere |
| --- | --- | --- |
| 4 octeți | Dimensiunea blocului | Dimensiunea blocului, în octeți, după acest câmp |
| 80 octeți | Antetul blocului | Câmpurile care formează antetul blocului |
| 1-9 octeți | Nr. de tranzacții | Câte tranzacții urmează |
| Variabilă | Tranzacții | Tranzacțiile înregistrate în acest bloc |

Antetul blocului este alcătuit din trei seturi de metadate. Primul set de metadate reprezintă o referință la hash-ul unui bloc anterior, care leagă acest bloc de părintele său. Al doilea set de metadate, și anume dificultate, timestamp și nonce, se referă la procesul de “minerit”. Cea de-a treia piesă de metadate este rădăcina arborelui Merkle, o structură de date utilizată pentru a organiza eficient toate tranzacțiile din bloc. Tabelul 2.2 descrie structura antetului unui bloc.

Tabel 2.2. Structura antetului unui bloc

| Dimensiune | Câmp | Descriere |
| --- | --- | --- |
| 4 octeți | Versiune | Un număr de versiune pentru a urmării upgrade-urile de software / protocol |
| 32 octeți | Hash-ul blocului anterior | O referință la hash-ul blocului anterior (părinte) din lanț |
| 32 octeți | Rădăcină Merkle | Un hash al rădăcinii arborelui Merkle care conține tranzacțiile acestui bloc |
| 4 octeți | Timestamp | Timpul aproximativ de la crearea acestui bloc |
| 4 octeți | Dificultate | Obiectivul de dificultate al algoritmului Proof-Of-Work pentru acest bloc |
| 4 octeți | Nonce | Un contor folosit pentru algoritmul Proof-Of-Work |

Identificatorul primar al unui bloc este hash-ul său criptografic, o amprentă digitală, rezultată prin aplicarea unei funcții hash, de două ori, antetului blocului prin algoritmul SHA256. Hash-ul de 32 de octeți rezultat este numit hash-ul blocului, dar este mai exact hash-ul antetului blocului, deoarece se calculează folosind informațiile din antet. De exemplu, 000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f, este hash-ul primului bloc creat vreodată. Hash-ul identifică un bloc în mod unic și lipsit de ambiguitate și poate fi derivat independent de orice nod.

O a doua modalitate de a identifica un bloc este prin poziția sa în blockchain, numită înălțimea blocului. Primul bloc creat vreodată este la înălțimea blocului 0 (zero) și este același bloc care a fost menționat anterior. Un bloc poate fi identificat astfel în două moduri: prin referire la hash-ul blocului sau prin referire la înălțimea blocului.

Spre deosebire de hash-ul blocului, înălțimea blocului nu este un identificator unic. Deși un singur bloc va avea întotdeauna o înălțime specifică și invariantă, reciproca nu este adevărată – înălțimea blocului nu identifică întotdeauna un singur bloc. Două blocuri care concurează pentru aceeași poziție în blockchain pot avea aceeași înălțime.

Primul bloc din blockchain este denumit bloc de geneză și a fost creat în 2009. Acesta este strămoșul comun al tuturor blocurilor din blockchain, ceea ce înseamnă că, începând cu orice bloc, la capătul lanțului se află blocul de geneză. Astfel, fiecare are ca punct de plecare al blockchain-ului, o “rădăcină” sigură din care să construiască un bloc de încredere.

Nodurile din rețeaua bitcoin întrețin o copie locală a blockchain-ului, începând de la blocul de geneză. Copia locală a blockchain-ului este actualizată în mod constant, pe măsură ce se găsesc blocuri noi și se utilizează pentru extinderea lanțului. Când un nod primește blocuri noi de la rețea, acesta le validează și apoi le va lega la blockchain-ul existent. Pentru a stabili o legătură, un nod va examina antetul blocului și va căuta câmpul “previous hash”.

### Arbori Merkle

Fiecare bloc din blockchain conține un rezumat al tuturor tranzacțiilor pe care le conține, utilizând un arbore Merkle.

Un arbore Merkle, cunoscut și ca un arbore binar al hash-urilor, este o structură de date utilizată pentru a rezuma și verifica eficient integritatea seturilor mari de date. Arborii Merkle sunt arbori binari care conțin hash-uri criptografice. Termenul “arbore” este folosit în domeniul informaticii pentru a descrie o structură de date ramificată, dar acești arbori sunt reprezentați de obicei cu “rădăcina” în partea de sus și “frunzele” în partea de jos a unei diagrame.

Arborii Merkle sunt utilizați în rețeaua bitcoin pentru a rezuma toate tranzacțiile într-un bloc, producând o amprentă digitală globală a întregului set de tranzacții, oferind un proces foarte eficient pentru a verifica dacă o tranzacție este inclusă într-un bloc. Un arbore este aplicarea recursivă a unei funcții hash pe o pereche de noduri, până când există un singur hash, numit rădăcină sau rădăcină Merkle. Algoritmul hash criptografic utilizat în arborii Merkle este SHA256 aplicat de două ori, cunoscut și sub numele de double-SHA256.

Atunci când N elemente sunt rezumate într-un arbore Merkle, prin algoritmul menționat anterior, se poate verifica dacă un anumit element este inclus în arbore cu cel mult calcule, ceea ce face ca aceasta să fie o structura de date foarte eficientă.

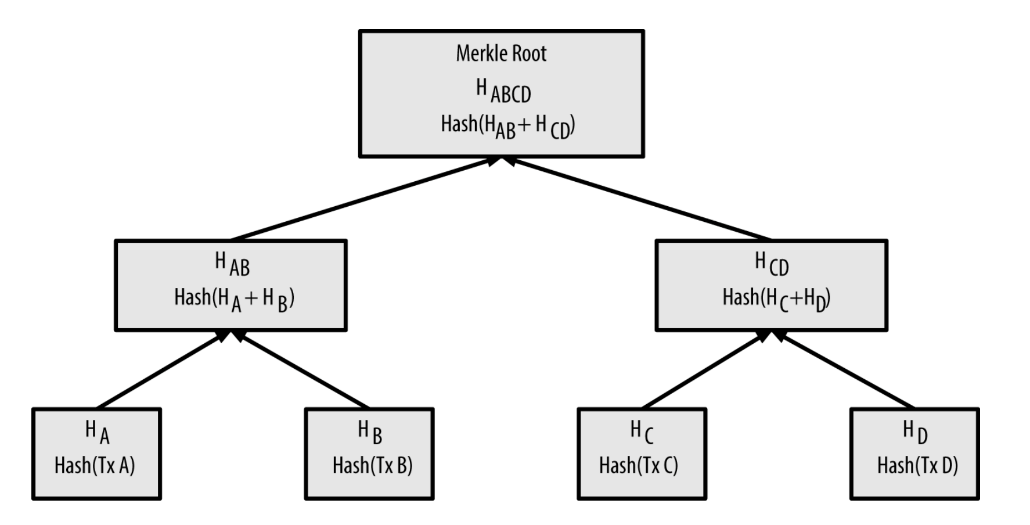


Figura 2.7 Calculul nodurilor într-un arbore Merkle

În exemplul din figura 2.7 sunt reprezentate patru tranzacții, A, B, C și D, care formează frunzele arborelui Merkle. Tranzacțiile nu sunt stocate în arbore, hash-ul rezultat din datele lor este stocat în fiecare nod de tip frunză ca , , și :

*HA = SHA256(SHA256(Transaction A))*

Perechi consecutive de noduri de tip frunză sunt apoi rezumate într-un nod părinte, prin concatenarea celor două hash-uri și aplicând o funcție hash rezultatului. De exemplu, pentru a construi nodul părinte , cele două hash-uri de 32 de octeți ale copiilor sunt concatenate pentru a crea un șir de caractere de 64 de octeți. Acest șir este apoi procesat cu ajutorul algoritmului double-SHA256 pentru a produce hash-ul nodului părinte:

*HAB = SHA256(SHA256(HA + HB))*

Procesul continuă până când există un singur nod în partea de sus, nod cunoscut ca rădăcină Merkle. Acest hash de 32 de octeți este stocat în antetul blocului și rezumă toate datele din cele patru tranzacții.

Deoarece arborele Merkle este un arbore binar, are nevoie de un număr par de noduri de tip frunză. Dacă există un număr impar de tranzacții care trebuie rezumate, hash-ul ultimei tranzacții va fi duplicat pentru a crea un număr par de noduri de tip frunză, cunoscut de asemenea ca un arbore echilibrat.

### Mineritul și algoritmul Proof-Of-Work

Mineritul este procesul de adăugare a tranzacțiilor la registrul public al rețelei bitcoin. Acest proces asigură securitatea și integritatea datelor și permite apariția unui consens la nivel de rețea fără o autoritate centrală. Minerii sunt recompensați cu taxele de tranzacționare și o parte din monedele noi introduse. Această schemă de recompensare asigură atât distribuirea monedelor într-un mod descentralizat cât și motivarea participanților de a asigura securitatea rețelei.

Minerii validează noile tranzacții și le înregistrează în registrul global. Un bloc nou, care conține tranzacții care au avut loc de la ultimul bloc procesat, este “minat” la fiecare 10 minute în medie, adăugând astfel tranzacțiile respective în blockchain. Tranzacțiile care fac parte dintr-un bloc și sunt adăugate blockchain-ului sunt considerate “confirmate”, ceea ce permite noilor proprietari de bitcoin să își cheltuie monedele primite prin intermediul acestor tranzacții.

Un bloc tipic va conține în jur de 200-300 de tranzacții. Un punct critic care trebuie reținut este că toți minerii primesc toate tranzacțiile care trebuie procesate și lucrează independent pentru a crea un bloc. Odată ce un miner creează un bloc valid, îl transmite la rețea iar fiecare nod al rețelei va verifica validitatea acestuia și apoi îl va adăuga la copia locală a registrului.

Protocolul bitcoin specifică faptul că, pentru a produce un bloc “valid”, un miner trebuie să prezinte dovada că a consumat o anumită cantitate de putere de procesare (și deci de timp) în crearea blocului. În practică, acest lucru înseamnă că minerii trebuie să prezinte un răspuns la o problemă matematică complexă, care poate fi găsită numai prin încercări succesive. Dificultatea acestei probleme matematice este stabilită astfel încât un miner să o poată rezolva la fiecare zece minute, în medie. Protocolul este conceput astfel încât blocurile să poată fi găsite mai rapid colectiv, decât individual.

Algoritmul folosit în procesul de minerit constă în următorii pași:

* hash-ul blocului anterior este preluat din rețeaua bitcoin;
* un set de potențiale tranzacții sunt asamblate într-un bloc;
* se calculează hash-ul dublu al antetului blocului cu un nonce și hash-ul precedent folosind algoritmul SHA256;
* dacă hash-ul rezultat este mai mic decât nivelul actual de dificultate (ținta), atunci procesul se oprește;
* dacă hash-ul rezultat este mai mare decât nivelul actual de dificultate (ținta), atunci se repetă procesul prin creșterea nonce-ului.
* dificultatea a crescut de-a lungul timpului și monedele care erau procesate folosind procesorul unui singur laptop acum necesită centre dedicate pentru a calcula hash-ul.

Protocolul bitcoin stabilește o valoare țintă pentru hash-ul antetului blocului. Ieșirea trebuie să fie mai mică decât numărul specificat sau, într-o altă ordine de idei, hash-ul trebuie să înceapă cu un anumit număr de zerouri.

Orice bloc al cărui antet nu produce un hash care este mai mic decât valoarea țintă, așa cum a fost menționat anterior, va fi respins de rețea. Dacă antetul nu conține numărul corespunzător de zerouri aici intervine câmpul denumit nonce. Acest câmp este pur și simplu un număr aleator care este adăugat la antetul blocului pentru a oferii o valoare incrementabilă în încercarea de a produce un hash valid. În cazul în care prima încercare de a produce un hash valid eșuează, se incrementează câmpul cu o unitate și se repetă procedeul până la obținerea unui hash valid.

Informațiile referitoare la blocuri și tranzacții sunt publice și pot fi verificate și online. Un exemplu de site care oferă aceste informații este <https://blockchain.info> iar url-urile sunt de forma:

[*https://blockchain.info/block/000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f*](https://blockchain.info/block/000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f)

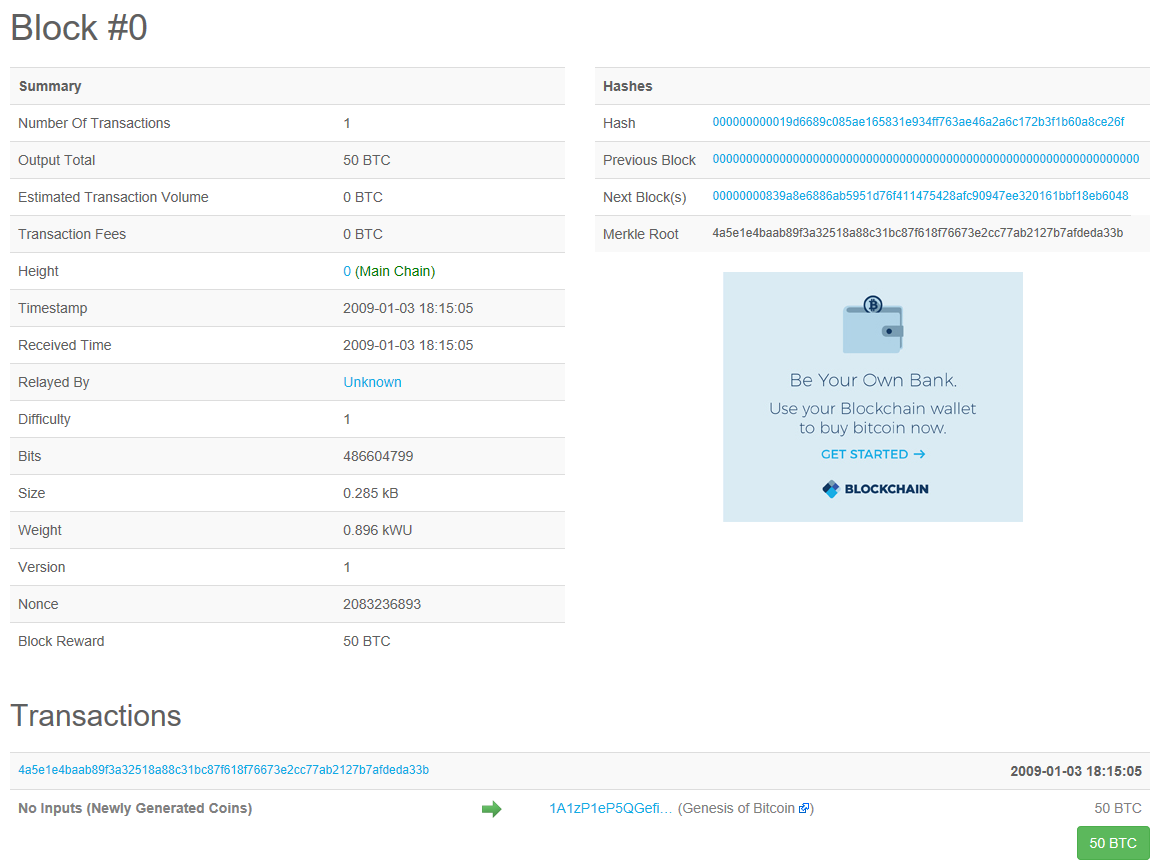


Figura 2.8 Informațiile publice despre blocurile și tranzacțiile din rețeaua bitcoin

## Ethereum

Ethereum este o infrastructură de calcul, open-source, descentralizată la nivel global, care execută programe numite *contracte inteligente* (eng. smart contracts). Utilizează un blockchain pentru a sincroniza și stoca starea sistemului împreună cu o criptomonedă numită *ether* pentru a măsura și restrânge costurile de execuție ale resurselor.

Platforma Ethereum permite dezvoltatorilor să construiască *aplicații descentralizate* (eng. descentralized applications) cu funcții economice integrate. În același timp, împărtășește multe elemente comune cu alte rețele blockchain publice: o rețea de tip peer-to-peer la care participanții se conectează, un algoritm de consens pentru sincronizarea stării acestei rețele (Proof-Of-Work), o monedă digitală (ether) și un registru global (blockchain-ul).

Principalul scop al platformei Ethereum nu este un sistem de plată digital. În timp ce moneda digitală ether este atât integrală cât și necesară pentru funcționarea Ethereum, ether-ul este conceput ca o monedă utilitară pentru a plăti pentru utilizarea platformei Ethereum.

Termenul “contract inteligent” a primit de-a lungul timpului o mare varietate de definiții. În 1990, Nick Szabo a inventat termenul și l-a definit ca “un set de promisiuni, specificate în formă digitală, inclusiv protocoale în cadrul cărora părțile îndeplinesc celelalte promisiuni”.

Vitalik Buterin, co-fondator al Ethereum , a dat pe blogul său următoarea definiție:

“*Un contract inteligent este cea mai simplă formă de automatizare descentralizată și este definită cel mai ușor și mai precis după cum urmează: un contract inteligent este un mecanism care implică bunuri digitale și două sau mai multe părți, unde unele sau toate părțile își depozitează bunurile și bunurile sunt automat redistribuite între acele părți în conformitate cu o formulă bazată pe anumite date care nu sunt cunoscute la momentul inițierii contractului* [6].”

Un exemplu adus pentru a susține această definiție este următorul:

Un participant A dorește să plătească 500 de dolari unui participant B pentru a construi un site web. Contractul va funcționa după cum urmează: A depozitează 500 de dolari în contract, iar fondurile sunt închise. Când B finalizează site-ul web, B poate trimite un mesaj la contract care cere să deblocheze fondurile. Dacă A este de acord, fondurile sunt eliberate. Dacă B decide să nu termine site-ul, B poate să renunțe prin trimiterea unui mesaj pentru a renunța la fonduri. Dacă B susține că a terminat site-ul web, dar A nu este de acord, atunci după o perioada de așteptare de șapte zile, judecătorul J va trebui să dea un verdict în favoarea lui A sau a lui B.

În implementarea practică adusă de platforma Ethereum, acestea conțin linii de cod, pot interacționa cu alte contracte, pot lua decizii, pot stoca date și pot trimite ether altor persoane. Contractele sunt definite de creatorii lor, însă execuția lor și, prin extensie, serviciile pe care le oferă sunt furnizate de rețeaua Ethereum.

Deși există posibilitatea de a scrie contracte inteligente direct în cod mașină, acesta este greu și dificil de citit și de interpretat pentru programatori. În schimb, majoritatea dezvoltatorilor Ethereum utilizează un limbaj de nivel înalt pentru a scrie programe și un compilator pentru a le converti în cod mașină. Principalul limbaj folosit în rețeaua Ethereum este Solidity.

Solidity este un limbaj de nivel înalt, orientat pe contract pentru implementarea contractelor inteligente [7]. Acesta a fost influențat de C++, Python și JavaScript și este conceput pentru a viza mașina virtuală Ethereum (EVM).

Spre deosebire de Bitcoin, care are un limbaj de scriptare foarte limitat, Ethereum este proiectat să fie un blockchain programabil cu scop general care rulează pe o mașină virtuală capabilă să execute cod de o complexitate arbitrară și nelimitată. În timp ce limbajul de scriptare al rețelei Bitcoin este limitat la simpla evaluare adevărat/fals a condițiilor, limbajul platformei Ethereum este “Turing Complete”, ceea ce înseamnă că este echivalent cu un computer cu scop general care poate executa orice calcul pe care o mașină teoretică Turing îl execută.

Termenul “Turing Complete” este numit după matematicianul englez Alan Turing, care este considerat părintele informaticii. În 1936 el a creat un model matematic al unui calculator, constând dintr-o mașină de stare care manipulează simboluri, prin citirea și scrierea lor pe o memorie secvențială (asemănătoare unei benzi magnetice de lungime infinită). Cu acest model, Alan Turing a oferit o bază matematică pentru a răspunde (în mod negativ) la întrebări despre calcul universal, în mod specific, dacă poate problemele sunt rezolvabile. Mai exact, el a demonstrat că problema opririi (eng. Halting Problem), care reprezintă încercarea de a evalua dacă un program se va opri în cele din urmă, nu este rezolvabilă.

Turing a demonstrat că nu se poate fi anticipat dacă un program se va termina, prin simularea lui pe un calculator. În termeni simpli, nu se poate anticipa mersul unui program fără al executa. Sistemele “Turing Complete” pot rula în “bucle infinite”, un termen folosit pentru a descrie un program care nu se termina. Ethereum nu poate prezice dacă un contract inteligent se va încheia, sau cât timp va funcționa , fără să îl execute (cu posibilitatea de a rula într-o buclă infinită). Pentru a rezolva această provocare, Ethereum a introdus un mecanism de măsurare numit gaz. Pe măsură ce mașina virtuală execută un contract inteligent, ține evidența pentru fiecare instrucțiune (calcul, acces la date, etc.) Fiecare instrucțiune are un cost prestabilit în unități de gaz. Atunci când o tranzacție declanșează executarea unui contract inteligent, trebuie să includă o anumită cantitate de gaz care poate fi consumată în execuția contractului. Mașina virtuală va termina execuția dacă cantitatea de gaz inclusă este insuficientă.

O aplicație descentralizată este asemănătoare unui program software obișnuit dar, spre deosebire de acesta, este implementată și operată în mod distribuit pe o rețea de tip peer-to-peer și codul său sursă este open-source. Structura tipică a unei astfel de aplicații constă în front-end, back-end și un mediu de stocare a datelor. Este similară unui contract inteligent, dar diferită în două moduri-cheie. În primul rând, o aplicație descentralizată are un număr nelimitat de participanți. În al doilea rând, o aplicație descentralizată nu trebuie să fie neapărat financiară. Din cauza acestei a doua cerințe, aplicațiile descentralizate sunt, de fapt, unele dintre cele mai simple lucruri care pot fi implementate cu ajutorul platformei Ethereum.

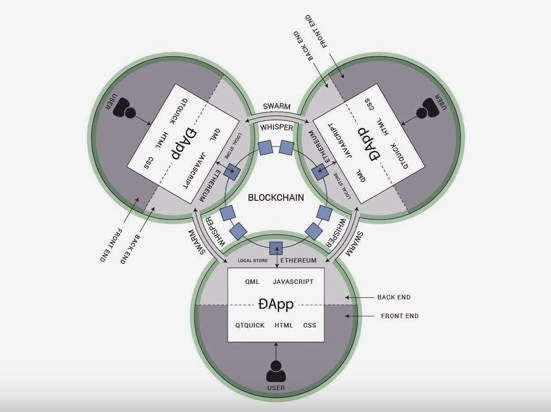


Figura 2.9 Ilustrare a unei aplicații descentralizate care folosește blockchain-ul

Pentru ca o aplicație să fie considerată o aplicație descentralizată, trebuie să îndeplinească următoarele criterii:

* aplicația ar trebui să fie complet open-source și autonomă și nici o singură entitate nu ar trebui să dețină controlul asupra majorității jetoanelor sale;
* datele și înregistrările referitoare la operațiunile aplicației ar trebui să fie securizate criptografic și stocate într-un registru public, pentru a evita orice puncte centrale de eșec;
* un jeton criptografic ar trebui folosit pentru a obține acces la aplicație și cei care contribuie și aduc valoare aplicației ar trebui recompensați;
* jetoanele trebuie să fie generate de aplicația descentralizată conform standardelor unei algoritm criptografic.

## Hyperledger

Hyperledger este un efort colectiv open-source creat pentru a promova în industrie tehnologiile de tip blockchain. Este o colaborare globală, găzduită de Fundația Linux, incluzând lideri din domeniul financiar, bancar, IoT, lanțuri de aprovizionare, producție și tehnologie [8].

Proiectul urmărește să reunească o serie de eforturi independente pentru a dezvolta protocoale deschise și standarde, oferind un framework modular care să suporte diferite componente pentru diferite utilizări. Acesta ar include o varietate de blockchain-uri cu propriul algoritm de consens, modele de stocare, servicii pentru identitate, control al accesului și contracte inteligente. Principalul obiectiv este de a construi și a rula platforme care să sprijine tranzacțiile globale de afaceri. Proiectul se axează, de asemenea, pe îmbunătățirea fiabilității și performanței sistemelor de tip blockchain.

Directorul executiv Brian Behlendorf a declarat că nu va exista o monedă digitală și că proiectul Hyperledger în sine nu își va construi propria cryptomonedă, obiectivul strategic al proiectului fiind să ofere soluții pentru aplicații industriale.

Printre proiectele care se desfășoară în prezent sub Hyperledger se pot enumera: Fabric, Iroha, Sawtooth, Indy, Burrow și Composer. Proiectul Hyperledger are în prezent 100 de membrii și este foarte activ cu mai mult de 120 de colaboratori, întâlniri regulate și discuții organizate în întreaga lume.

Hyperledger Fabric este contribuția inițială adusă de IBM proiectului Hyperledger. Scopul acestei contribuții este de a permite o abordare modulară, deschisă și flexibilă în vederea construirii rețelelor blockchain.

Fabric este prima platformă distribuită care permite crearea contractelor inteligente în limbaje de programare cu scop general cum ar fi Java, Go și Node.js. Acest lucru înseamnă că majoritatea întreprinderilor dispun deja de setul de competențe necesar pentru a dezvolta contracte inteligente și nu este necesară instruirea suplimentară a angajaților pentru a învăța un limbaj nou.

Hyperledger Composer este unul din instrumentele din ecosistemul Hyperledger.

Composer este un framework de dezvoltare a aplicațiilor care simplifică și accelerează crearea de aplicații folosind Hyperledger Fabric. Permite definirea elementelor gestionate în cadrul blockchain-ului (bunuri), participanțiilor la rețea, tranzacțiilor și a evenimentelor declanșate de execuția tranzacțiilor printr-un limbaj de modelare numit Composer Modeling Language. Fluxurile pentru fiecare tip de tranzacție sunt expuse printr-un API folosind cod Javascript. Se pot definii fișiere de control al accesului pentru a limita drepturile de acces ale participanților la anumite resurse. Query-urile executate frecvent pentru a accesa datele din registru pot fi modelate folosind Composer Query Language, un limbaj asemănător cu SQL.

Toate fișierele necesare care formează definiția rețelei (Business Network Definition) sunt înglobate într-un fișier .bna. Această arhivă poate fi apoi instalată pe o rețea existentă oferită de Hyperledger Fabric. Codul sursă poate fi dezvoltat și testat folosind orice editor de text cu sau fără suport pentru limbajele menționate anterior și poate fi versionat cu ajutorul Git-ului.

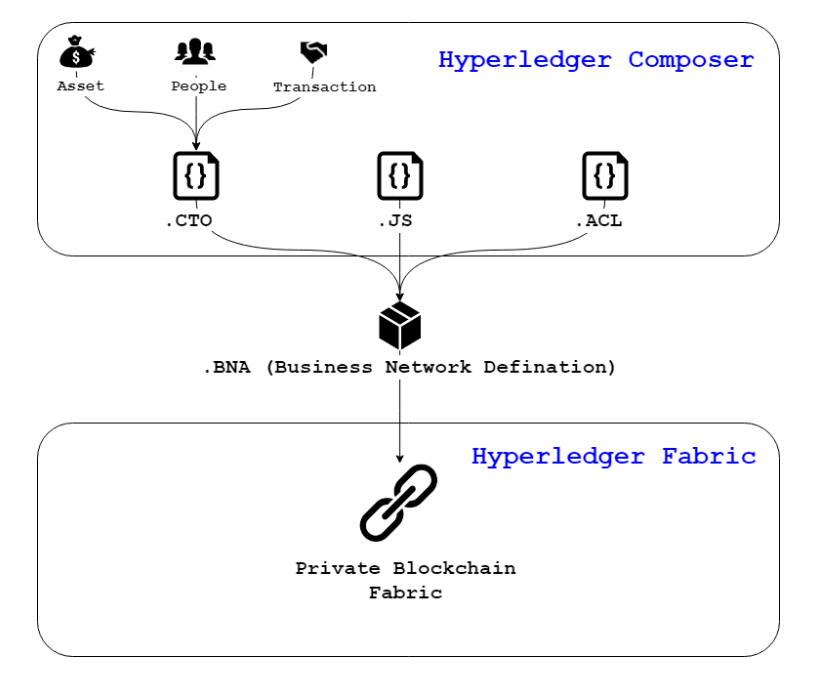


Figura 2.10 Relația între Hyperledger Fabric și Hyperledger Compsoser

# Analiză, proiectare, implementare

## Analiză și proiectare

Analiza software reprezintă studiul specificațiilor sistemului, al cerințelor acestuia (funcționale și non-funcționale) cât și al activităților care transformă aceste cerințe într-un produs final, într-un mod eficient.

Produsul final ar trebui să ofere utilizatorilor posibilitatea de a simula ciclul de viață al unui bun perisabil de la producător la cumpărător, proces asemănător cu managementul unui lanț de aprovizionare (eng. supply chain management), prin intermediul unei rețele de afaceri (eng. business network). În vederea îndeplinirii acestui obiectiv, sistemul va fi împărțit pe mai multe nivele care vor fi proiectate individual.

Ingineria cerințelor este procesul de stabilire a serviciilor cerute sistemului de către clienți precum și constrângerile sub care acesta va fi dezvoltat și va opera [9].

Cerințele pentru un sistem sunt descrierile a ceea ce ar trebui să facă sistemul – serviciile pe care le oferă și constrângerile legate de funcționarea acestuia. Cerințele sistemelor software sunt împărțite adesea în două categorii: *cerințe funcționale* și *cerințe non-funcționale*.

### Cerințe funcționale

Cerințele funcționale reprezintă o descriere a funcțiilor pe care sistemul ar trebui să le furnizeze, modul în care sistemul ar trebui să reacționeze la anumite intrări și modul în care sistemul ar trebui să se comporte în anumite situații. În unele cazuri, cerințele funcționale pot, de asemenea, să precizeze în mod explicit ce nu ar trebui să facă sistemul.

Arhitectura sistemului se va baza pe o rețea privată de tip blockchain, fiind destinat unei anumite întreprinderi sau organizații.

Utilizatorii finali ar trebui să aibă posibilitatea de a se alătura doar pe bază de invitație emisă de administratorul rețelei. Într-o rețea blockchain privată identitățile participanților sunt cunoscute, fapt pentru care invitațiile vor consta într-un card de identitate (eng. identity card) digital care conține un identificator unic al utilizatorului final. Acest card digital asigură atât partea de autentificare cât și partea de autorizare a sistemului atribuind utilizatorului un rol pentru care se pot impune o serie de constrângeri.

Pentru a ține o evidență a tuturor operațiilor desfășurate în cadrul organizației comunicarea se va face doar prin intermediul tranzacțiilor, chiar dacă există, spre exemplu, posibilitatea de a gestiona bunurile existente în mod direct. Aceste tranzacții trebuie să fie vizibile în mod direct doar de către părțile implicate, eliminând astfel transparența care există în rețelele publice.

După alăturarea la rețea, prin înregistrarea cardului digital, utilizatorii ar trebui să poată accesa resursele expuse în mod nerestricționat.

Există trei categorii de participanți: cumpărător, producător, expeditor.

Din punct de vedere al cumpărătorului, există posibilitatea de a plasa o comandă nouă pentru unul din bunurile existente (ex. mere, banane, piersici, cafea, etc.).

Comenzile plasate pot fi verificate și preluate de către unul din producători și pe baza acestei comenzi se poate întocmii un contract între producător și un potențial cumpărător. În contract sunt specificate o serie de condiții de livrare cum ar fi prețul pe unitate al bunurilor livrate și condițiile de livrare. Cumpărătorul are posibilitatea de a primii oferte (contracte) de la mai mulți producători, dar poate accepta un singur contract corespunzător unei singure comenzi sau poate refuza oricare dintre aceste contracte.

Contractele care au fost încheiate între cumpărător și producător pot fi preluate de cea de-a treia categorie de utilizator al rețelei, expeditor, care poate începe simularea procesului de livrare în care se trimit o serie de tranzacții pentru a încheia comanda.

Până la încheierea procesului de livrare cumpărătorul are posibilitatea de a citii temperatura la care se află comanda tot prin intermediul tranzacțiilor, oferind astfel posibilitatea de integrare cu sisteme IoT (ex. senzor de temperatură).

La încheierea procesului, dacă livrarea s-a efectuat în termenii agreați în contract, se face un transfer de fonduri între cumpărător și producător, acest proces fiind realizat în mod automat fără implicarea directă a participanților.

Administratorul rețelei are posibilitatea de a urmării toate tranzacțiile care s-au desfășurat pe parcursul simulării.

### Cerințe non-funcționale

Cerințele non-funcționale sunt un set de constrângeri asupra serviciilor sau funcțiilor oferite de sistem. Acestea includ constrângeri de timp, constrângeri asupra procesului de dezvoltare și constrângeri impuse de standarde. Cerințele non-funcționale se aplică adesea sistemului în ansamblu, mai degrabă decât funcțiile sau serviciile individuale ale sistemului.

Din categoriile de cerințe non-funcționale, sistemul este proiectat să țină cont de următoarele:

* accesibilitate – după cum a fost menționat anterior, accesibilitatea la rețea va fi gestionată de către administratorul acesteia, accesul utilizatorilor fiind restricționat;
* disponibilitate – se impune un uptime al sistemului de **99.90%**, singurele întreruperi fiind cele de mentenanță;
* capacitate – capacitatea sistemului este direct dependentă de mașina pe care rulează rețeaua, dar trebuie să existe posibilitatea de a extinde această capacitate la atingerea limitei de spațiu;
* performanță – se impun o serie de constrângeri de performanță, cum ar fi: timpul de răspuns pentru orice request să nu depășească **2 sec**. (pentru a asigura această constrângere, în cazul în care serverul care ar trebui să gestioneze request-ul nu funcționează din diverse motive tehnice, se va impune un timeout pentru toate componentele, atât back-end cât și front-end care să închidă conexiunea, utilizatorul nefiind nevoit să aștepte după mesajul de eroare); puterea de calcul ar trebui să asigure o medie de **500 de tranzacții pe minut** fără întreruperi ale aplicației și ale modului normal de funcționare;
* scalabilitate – sistemul ar trebui să fie scalabil atât pe orizontală (prin creșterea numărului de noduri din rețeaua distribuită unde rulează sistemul, sau prin îmbunătățirea computerelor care formează această rețea distribuită) cât și pe verticală (pentru a suporta un număr mai mare de utilizatori, în cazul în care se observă timpi de răspuns mari sau cozi de așteptare, se poate duplica serviciul REST expus de business network și se poate implementa un load balancer care să distribuie request-urile în mod egal între cele două instanțe);
* securitate – securitatea este menținută prin intermediul cardurilor de identitate și în acest mod doar utilizatorii aleși de către administratorul rețelei se pot alătura;
* fiabilitate – remarcată prin calitatea bună a codului scris și obținută prin respectarea standardelor și guideline-urilor impuse.

Tot la cerințe non-funcționale se poate încadra și proprietatea de integrabilitate a sistemului. Aceasta poate fi realizată de către clienții externi ai organizației, care doresc să își integreze aplicațiile existente (ex. supermarket) cu business network-ul oferit de organizație prin intermediul serviciului REST expus, sau de către clienții interni (membrii ai organizației) care ar dorii să integreze, spre exemplu, sistemul existent cu un serviciu de notificări prin email, extinzând astfel funcționalitățile sistemului în ansamblu.

### Cazuri de utilizare

În cazul ingineriei software, un caz de utilizare este o listă de acțiuni sau evenimente care definesc de obicei interacțiunile și tipurile de interacțiuni dintre un actor și un sistem, în vederea atingerii unui scop. Actorul poate fi o persoană sau un alt sistem. Cazurile de utilizare sunt completate de informații suplimentare care descriu interacțiunea cu sistemul. Informațiile suplimentare pot fi o descriere textuală sau o reprezentare grafică (diagrame UML). Setul cazurilor de utilizare reprezintă toate interacțiunile posibile care au fost descrise în cerințele sistemului.

Actorii, în varianta simplificată, sunt: cumpărător, expeditor, producător și administrator de rețea.

Diagrama UML a cazurilor de utilizare pentru sistem este reprezentată în figura de mai jos.

Figură

Administratorul rețelei poate genera identități pentru participanți. Arhitectura permite mai mult de un utilizator din fiecare categorie, dar pentru simplitate se va realiza simularea cu un singur reprezentat.

Utilizatorii se pot alătura rețelei (asemănător procesului de autentificare în aplicațiile clasice) cu ajutorul cardului de identitate emis de administrator. Resursele sunt accesibile doar prin intermediul acestui card și doar cu rolul atribuit fiecărui utilizator.

Cumpărătorul poate vedea comenzile plasate împreună cu statusul lor în timp real și poate plasa comenzi noi. În meniul de adăugare comenzi se poate selecta doar unul din produsele existente, necesitatea adăugării unui produs nou implică editarea modelelor rețelei și downtime al aplicației pentru aplicarea acestor modele în definiția rețelei (BND).

Producătorul poate vedea comenzile plasate și pentru comenzile care nu au un contract încheiat de cumpărător pot plasa un contract nou. Între contract și comandă există o relație de one-to-one.

Cumpărătorul poate revizui contractele plasate de producători și pe baza termenilor din aceste contracte (preț, timp de livrare, etc.) poate accepta unul sau poate refuza oricare dintre acestea. În orice moment, cumpărătorul poate vedea informații referitoare la contractul acceptat, inclusiv datele de contact ale producătorului.

Expeditorul are vizibilitate doar asupra contractelor care au fost încheiate între cumpărător și producător. Nu poate vedea contracte noi și nu poate vedea comenzi care au fost plasate doar nu au primit niciun contract. După revizuirea condițiilor de livrare, expeditorul poate începe simularea procesului de livrare care implică trimiterea unei serii de tranzacții.

Cumpărătorul are posibilitatea de a verifica temperatura comenzii în orice moment după ce a fost început procesul de livrare. Depășirea limitei minime sau maxime de temperatură implică costuri care vor fi retrase din suma finală.

După finalizarea simulării, dacă livrarea s-a efectuat în condițiile specificate în contract, are loc un transfer de fonduri între cumpărător și producător care reprezintă suma pentru comanda plastă minus eventualele costuri pentru depășirea limitelor de temperatură (dacă există). Administratorul poate vizualiza toate tranzacțiile care au fost executate până la momentul livrării.

### Arhitectura sistemului

Designul arhitectural este procesul de înțelegere a modului în care trebuie organizat un sistem și de proiectarea structurii generale a acestuia. Este legătura între ingineria cerințelor și proiectare, deoarece identifică principalele componente structurale ale unui sistem și relațiile dintre ele. Rezultatul procesului de design arhitectural este un model arhitectural care descrie modul în care sistemul este organizat ca un set de componente care comunică între ele.

Modelele arhitecturale (eng. arhitectural patterns) reprezintă un mijloc de reutilizare a cunoștințelor despre arhitecturile sistemelor generice. Un model arhitectural descrie o organizare a sistemului care a fost testată și a avut succes în sistemele anterioare. Acestea descriu arhitectura, conțin informații referitoare la mediul în care se poate folosi și discută despre avantajele și dezavantajele sale.

Datorită legăturii între arhitectura software și ingineria cerințelor, modelele arhitecturale folosite pentru sistemul prezentat în lucrare au fost alese astfel încât să satisfacă cerințele non-funcționale prezentate anterior. Nu se discută despre cerințele funcționale deoarece, pentru implementarea unei funcționalități noi, este de ajuns modificarea unei singure componente a sistemului. În schimb, dacă una dintre cerințele non-funcționale nu este respectată sunt necesare modificări la nivelul întregului ansamblu.

Deoarece s-a discutat despre accesibilitate, disponibilitate, performanță, scalabilitate și securitate ca cerințe non-funcționale pe care sistemul trebuie să le îndeplinească, modelul arhitectural ales trebuie să asigure toate aceste cerințe.

**Layered arhitecture** este un model arhitectural în care noțiunile de separare și independență sunt fundamentale deoarece permit ca modificările să se facă localizat. Funcționalitatea sistemului este organizată pe mai multe nivele separate și fiecare nivel se bazează numai pe facilitățile și serviciile oferite de nivelul imediat sub acesta.

Această abordare stratificată permite dezvoltarea incrementală a sistemelor. Pe măsură ce este dezvoltat un nivel, unele dintre serviciile oferite de acest nivel pot fi puse la dispoziția utilizatorilor. Acest tip de arhitectura sprijină principiile de dezvoltare agile care spun că sistemul ar trebui să se poată adapta ușor dacă cerințele se schimbă. Mai mult, dacă se adaugă facilități noi unui nivel, numai nivelul adiacent este afectat justificând astfel îndeplinirea cerinței de disponibilitate.

Arhitectura tipică a sistemelor dezvoltate folosind Hyperledger Composer, sugerată în documentația disponibilă online, este reprezentată în figura 3.2 și corespunde modelului arhitectural ales anterior. Această arhitectură este formată din trei nivele “principale”, după cum urmează:

1. Business Network Level
2. REST API Level
3. Web (or Mobile) Application Level

Aceste nivele pot fi văzute la rândul lor ca sub-sisteme. Unele dintre ele sunt construite după un stil sau model arhitectural diferit și conțin mai multe componente care comunică formând astfel un ansamblu.

În continuare se va discuta despre fiecare nivel în parte punând în evidență particularitățile acestuia la nivel structural / arhitectural și funcționalitățile pe care le oferă.

#### Business Network Level

Acesta este nivelul cel mai de jos din diagrama arhitecturală și punctul de start în dezvoltarea aplicației. La acest nivel se află registrul distribuit (blockchain-ul) și logica business-ului care interacționează cu acesta. Este accesat de mai mulți participanți ai rețelei, dintre care unii pot fi responsabili cu întreținerea rețelei în sine, referiți ca administratori ai rețelei.

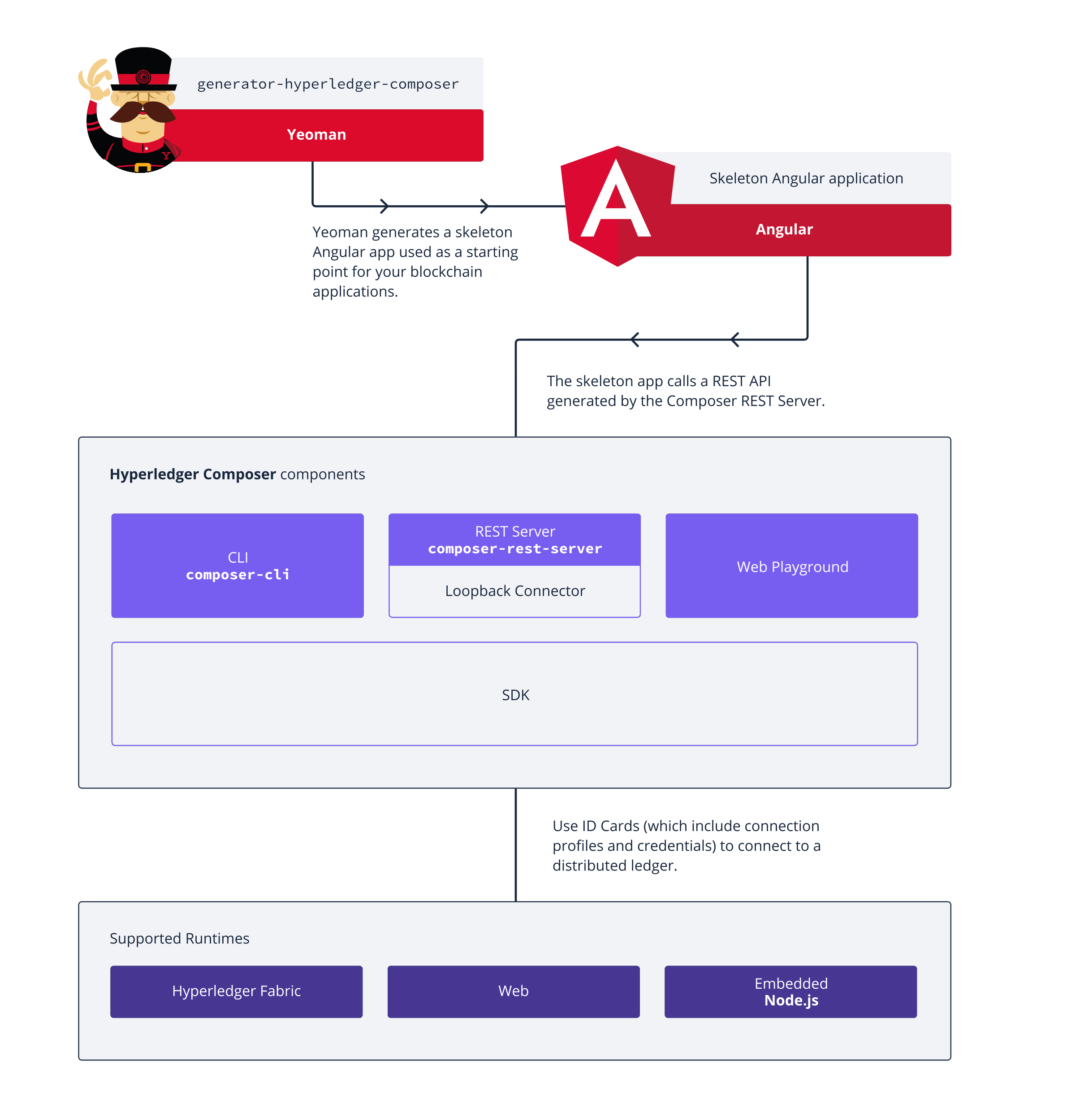


Figura 3.1 Arhitectura tipică a sistemelor dezvoltate folosind Hyperledger Composer

Fiecare participant are o identitate fixă și se poate identifica prin intermediul certificatelor emise (carduri de identitate). Pe lângă autentificare, este acoperită, de asemenea, autorizarea. Folosind acest sistem bazat pe roluri, cerințele legate de confidențialitate și securitate sunt îndeplinite.

O rețeaua blockchain este configurată și pusă la dispoziția dezvoltatorilor prin intermediul Hyperledger Fabric. Avantajele folosirii unei rețele blockchain ca mijloc de gestionare a tranzacțiilor au fost menționate în capitolele anterioare. Această rețea are deja configurat un nod, canalul pe care o să comunice utilizatorii finali și o identitate pentru administratorul rețelei sub forma unui card de identitate. Pentru a nu interacționa în mod direct cu rețeaua se va folosii framework-ul Hyperledger Composer ca un nivel de abstractizare și mijloc de comunicare cu aceasta.

Hyperledger Composer conține patru aspecte care pot fi modelate pentru a obține definiția rețelei (Business Network Definition) sub forma unui fișier cu extensie .bna care va fi instalat pe instanța Fabric existentă. Aceste aspecte sunt:

* Modele;
* Logica folosită de tranzacții (contracte inteligente);
* Controlul accesului;
* Query-uri.

Modele într-o rețea pot fi bunurile tranzacționate, participanții și tranzacțiile folosite de aceștia. Modelele folosite în aplicația dezvoltată în cadrul acestei lucrări sunt reprezentate în diagrama de clase din figura 3.3.

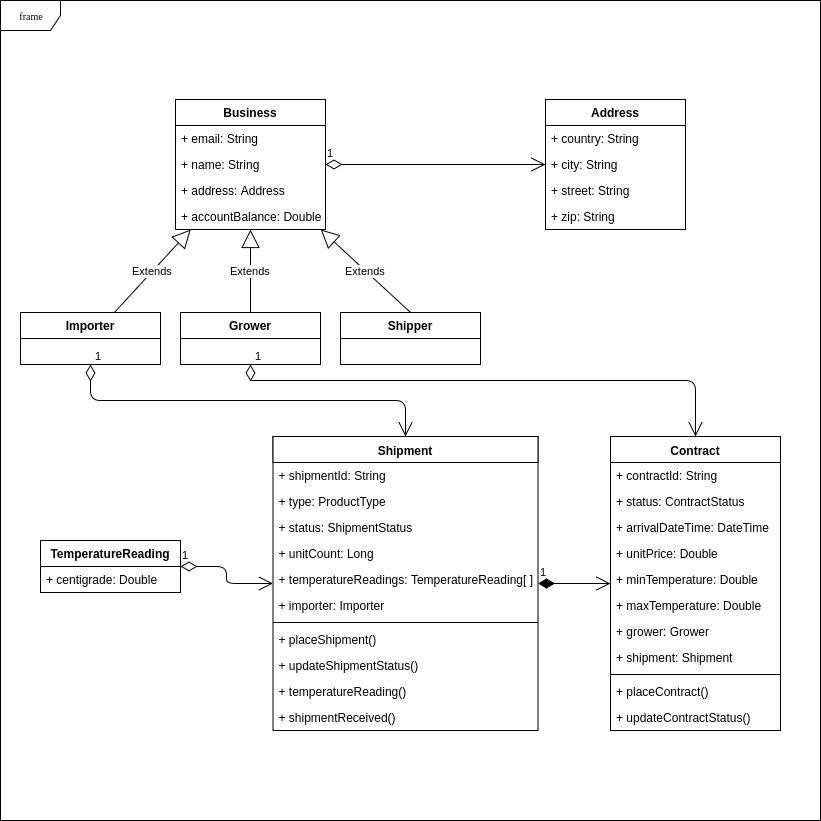


Figura 3.2 Diagrama de clase

Toate tranzacțiile transmise prin intermediul unui business network sunt stocate în registrul blockchain-ului, iar starea actuală a bunurilor și a participanților sunt stocate în baza de date a blockchain-ului. Blockchain-ul distribuie registrul și baza de date într-un set de noduri și se asigură că actualizările bazei de date și a registrului sunt consecvente pentru toate nodurile folosind un algoritm de consens.

Software-ul folosit de Hyperledger Fabric pentru a persista datele referitoare la starea rețelei se numește CouchDB.

Apache CouchDB este un software open-source pentru baze de date care se focusează pe ușurința utilizării și pe o arhitectură scalabilă. Are o arhitectură specifică bazelor de date de tip NoSQL orientate pe documente și este implementată în limbaj orientat pe concurență Erlang. Utilizează JSON pentru a stoca date, Javascript ca limbaj de interogare utilizând MapReduce și expune un API cu ajutorul HTTP [10].

Spre deosebire de o bază de date relațională, o bază de date CouchDB nu stochează date și relații în tabele. În schimb, fiecare bază de date este o colecție de documente independente. Fiecare document își păstrează propriile date și schema independentă. Metadatele documentelor conțin informații de revizuire, făcând posibilă îmbinarea eventualelor diferențe care s-au produs în timp ce bazele de date au fost deconectate.

Query-urile declarate de dezvoltator în fișierele cu extensie .qry sunt preluate în timpul procesului de deployment și sunt transformate în query-uri care interacționează în mod direct cu CouchDB. Un exemplu de query ar putea fi cel de mai jos unde <<namespace>> reprezintă domeniul organizației:

query selectImporters {

description: "Select all importers"

statement:

SELECT <<namespace>>.Importer

}

Aceste query-uri vor fi transformate mai târziu în endpoint-uri pentru REST API-ul aflat la următorul nivel în diagrama arhitecturală și vor putea fi folosite pentru a interacționa cu baza de date.

Identitățile sunt utilizate pentru a tranzacționa într-o rețea și trebuie să fie asociate unui participant (model) al rețelei. O identitate constă într-un certificat digital și o cheie privată. O identitate unică este stocată pe un card de identitate și dacă acea identitate a fost asociată unui participant, permite utilizatorului acelui card de identitate să efectueze tranzacții într-o rețea ca acel participant.

Hyperledger Composer utilizează “Connection Profiles” pentru a defini sistemul la care se conectează. Un profil este un document JSON care face parte dintr-un card de identitate. Acest profiluri sunt furnizate, de obicei, de administratorul rețelei.

Rețelele de afaceri pot conține un set de reguli de control al accesului. Regulile de control al accesului permit controlul fin asupra a ce participanți au acces la ce bunuri în rețea și în ce condiții. Aceste reguli intervin în momentul execuției unei tranzacții de către un participant asigurând astfel accesul autorizat la resurse.

Setul de reguli se declară în fișiere cu extensie .acl. Cea mai comună regulă întâlnită în procesul de implementare este “regula globală”, folosită de dezvoltatori atunci când se testează logica tranzacțiilor. Aceasta afirmă că toți participanții au acces la toate resursele:

rule Default {

description: "Allow all participants access to all resources"

participant: "ANY"

operation: ALL

resource: "<<namespace>>.\*"

action: ALLOW

}

Se pot seta și reguli mai stricte înlocuind câmpurile participant, operation și resource din exemplul de mai sus cu valorile corespunzătoare.

Modelele, query-urile, setul de reguli și logica tranzacțiilor, despre care se va discuta mai în detaliu în subcapitolul 3.2, sunt împachetate și exportate ca o arhivă cu extensie .bna care reprezintă definiția rețelei (Business Network Definition). Această arhiva va fi folosită de Hyperledger Fabric pentru a aduce online primul nivel din diagrama arhitecturală.

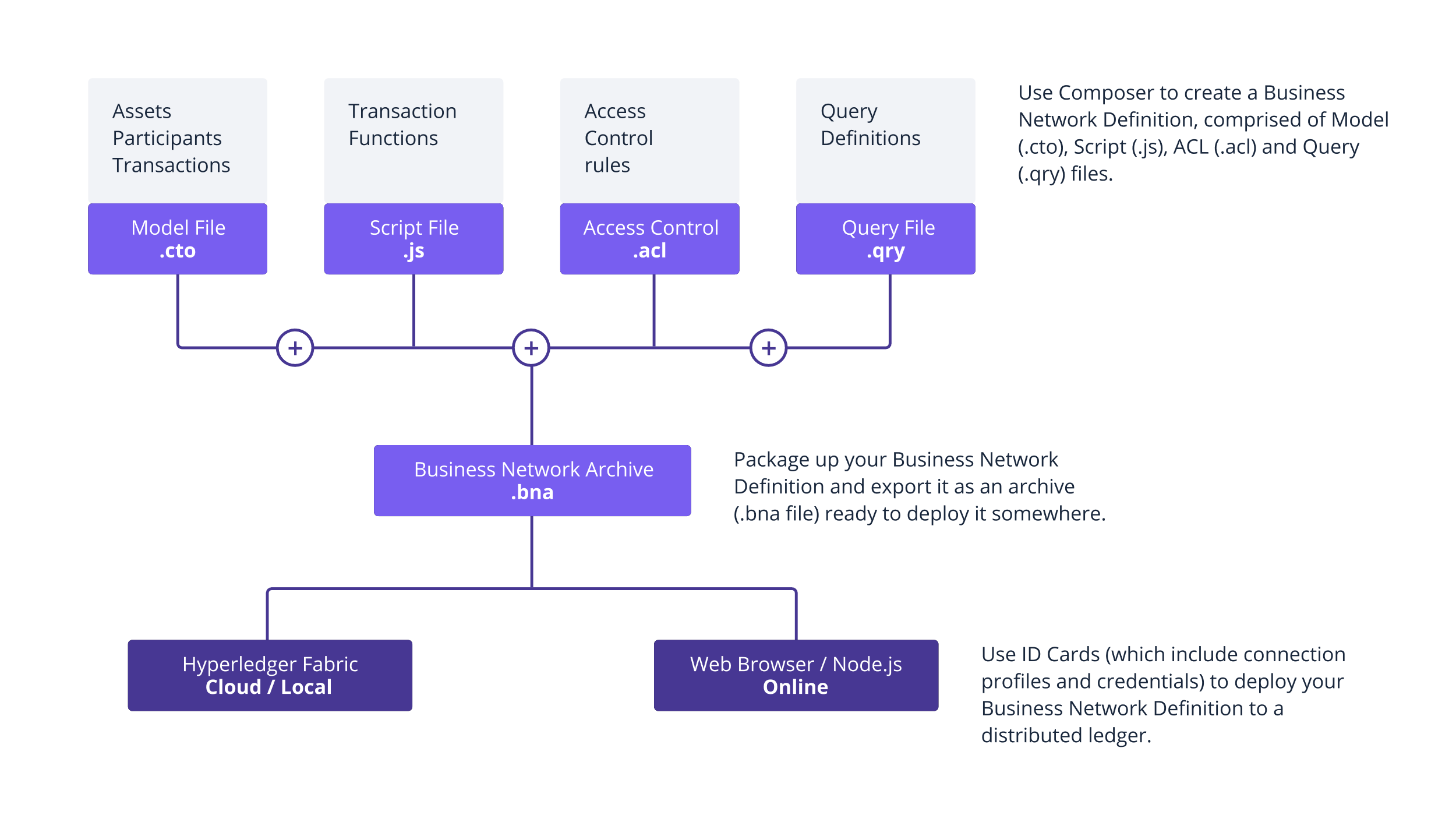


Figura 3.3 Procesul de deployment al definiției rețelei în Hyperledger Fabric

#### REST API Level

API este acronimul de la Application Programming Interface și reprezintă un set de subrutine, protocoale și tool-uri care permit unei aplicații software să comunice cu o altă aplicație. Când se discută despre API în contextul unei aplicații web, acel API este de fapt un REST API. Un stil arhitectural denumit REST (Representational State Transfer) susține că aplicațiile web ar trebui să utilizeze HTTP așa cum a fost imaginat inițial. Căutările ar trebui să utilizeze request-uri GET. Request-urile PUT, POST și DELETE ar trebui folosite pentru modificare, creare și respectiv ștergere. Un REST API este o interfață care permite utilizatorilor să comunice cu aplicația web prin intermediul acestor request-uri.

Una dintre constrângerile pe care le impune stilul REST este folosirea unui model arhitectural **client-server**.

Modelul client-server este o structură specifică aplicațiilor distribuite care împarte sarcini între furnizorii unei resurse sau serviciu, numiți servere, și solicitanții acestei resurse sau serviciu numiți clienți [11]. Principiul care stă la baza modelului client-server este separarea preocupărilor. Clienții accesează resursele și serviciile furnizate de server folosind un protocol de tip cerere-răspuns, cum ar fi protocolul HTTP. În esență, un client face o cerere către un server și așteaptă până când primește un răspuns.

API-urile RESTful bazate pe HTTP conțin următoarele elemente comune:

* adresă URL de bază - cum ar fi <http://api.example.com/resources>;
* media-type - care oferă informații în legătură cu formatul în care ar trebui afișat răspunsul (ex. application/json, application/xml);
* metode standard HTTP – tipul de request folosit conform stilului REST (ex. GET, POST, PUT, DELETE).

Pentru a accesa o resursă sau un serviciu pus la dispoziția utilizatorilor prin intermediul unui API REST este nevoie ca un client să inițieze un request la adresa URL unde se află resursa, folosind una dintre metodele HTTP, iar serverul va răspunde cu o reprezentare a acestei resurse pe baza media-type-ului specificat în request.

Un exemplu de request care returnează o resursă pe baza unui identificator al acesteia este următorul:

GET application/json https://api.example.com/resources/820882001277849

Acest request va returna în mod implicit următoarele câmpuri (proprietăți), formatate utilizând JSON:

{

"name": "Coca-Cola",

"id": "820882001277849"

}

Răspunsul nu este returnat în momentul în care a fost lansat request-ul, există un interval de timp numit timp de răspuns, dependent în mod direct de performanțele mașinii pe care se află server-ul. Ideal este ca acest timp de răspuns să fie cât mai mic și în caz de eroare să se întrerupă conexiunea fără a lăsa utilizatorul să aștepte. Ciclul de viață al unui request este reprezentat în diagrama secvențială din figura 3.5.

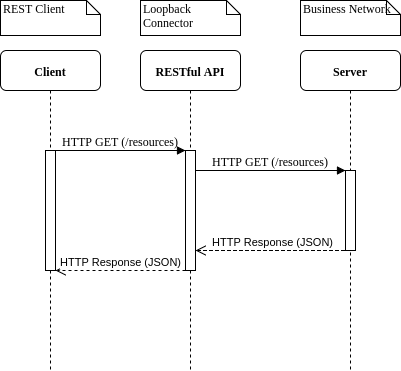


Figura 3.4 Ciclul de viață al unui request

Hyperledger Composer include un proces independent Node.js care expune o rețea ca un REST API. Framework-ul Loopback este folosit pentru a genera un Open API, descris de un document Swagger.

Loopback este un framework pentru crearea API-urilor și conectarea acestora cu sursele de date din back-end. Pentru a realiza asta, framework-ul preia o definiție a modelelor folosite de rețea, face o legătură cu sursa de date, CouchDB, și generează un API REST complet care poate fi apelat de orice client. Endpoint-urile API-ului sunt documentate și expuse pentru a fi testate folosind Swagger UI.

Swagger UI este un instrument open source care generează o pagină web care documentează API-urile generate de framework-ul Loopback. Prezentarea API-urilor folosind această interfață grafică sub forma unei pagini web este ușor de înțeles pentru utilizatori și păstrează logica și detaliile de implementare în spatele ecranului. Acest lucru permite dezvoltatorilor să testeze endpoint-urile expuse de API-ul REST cu ajutorul documentației și exemplelor oferite de Swagger UI.

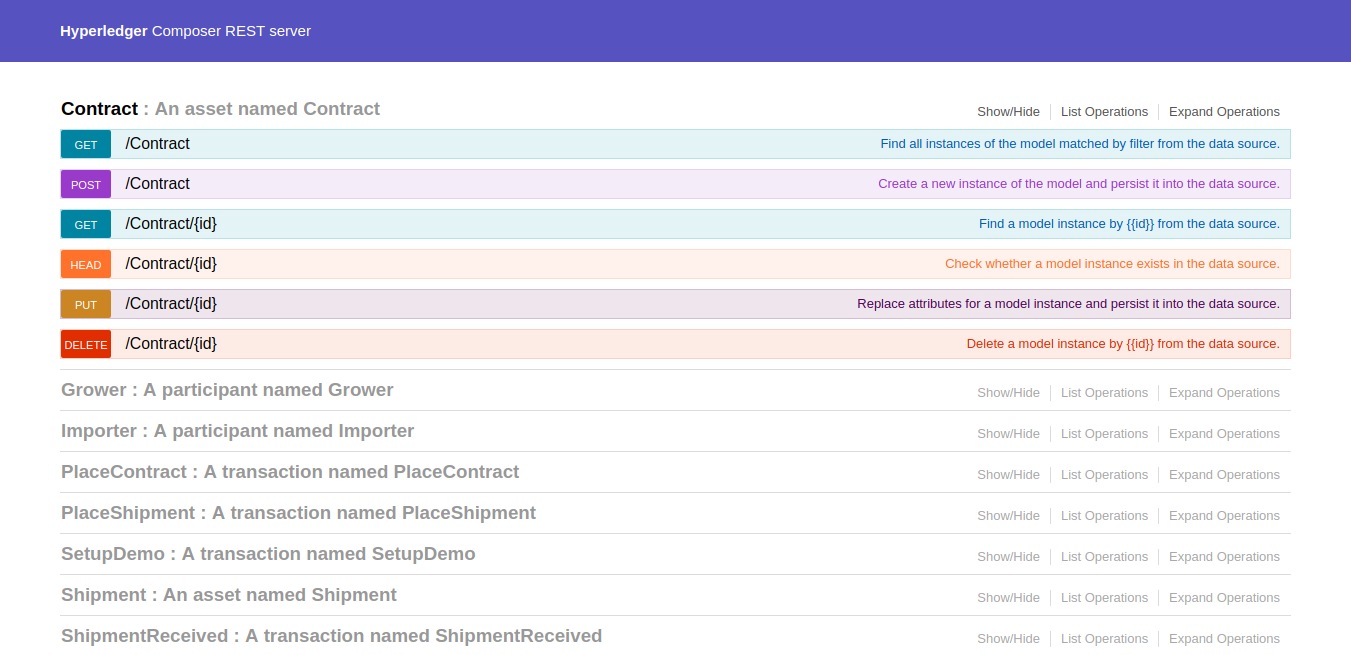


Figura 3.5 REST API-ul generat pentru Perishable Goods Network

În mod implicit API-ul nu este securizat, ceea ce înseamnă că oricine cunoaște adresa la care sunt disponibile resursele le poate accesa. Serverul REST poate fi configurat pentru autentificarea clienților. Când această opțiune este activată clienții trebuie să se autentifice la serverul REST înainte de a apela API-ul. Când un utilizator se autentifică la serverul REST, un token de acces unic este generat și asignat utilizatorului autentificat. Dacă autentificarea este realizată folosind un browser web, token-ul de acces este stocat într-un cookie din spațiul de stocare local al browser-ului. Atunci când utilizatorul autentificat face o cerere ulterioară, token-ul de acces este preluat din cookie și validat în schimb. Token-ul de acces poate fi utilizat pentru a autentifica orice client HTTP sau REST care dorește să apeleze serverul REST.

Când se implementează serverul REST oferit de Hyperledger Composer într-un mediu de producție, ar trebui configurat să fie disponibil 99% din timpul de funcționare. Acest lucru înseamnă că ar trebui implementate mai multe instanțe ale serverului REST și acele instanțe ar trebui configurate pentru a partaja date. De exemplu, datele cum ar fi profilurile de conexiune, identitățile participanților, și setările de autentificare ale API-ului ar trebui să fie partajate astfel încât un client REST să poată face o solicitare la oricare dintre instanțe fără a fi nevoie să se re-autentifice.

Hyperledger Composer poate fi integrat cu sistemele existente utilizând un API Loopback. Integrarea poate fi realizată în ambele direcții. Spre exemplu prin integrarea business network-ului cu sistemele existente externe există posibilitatea de a extrage date din aceste sisteme și de a le converti în bunuri și participanți. Sistemele externe, cum ar fi o companie de transport maritim, s-ar putea integra cu business network-ul pentru a prelua comenzi prin intermediul API-ului folosind-ul astfel ca o sursă de date.

#### Web (or Mobile) Application Level

Cel de-al treilea nivel în diagrama arhitecturală este reprezentat de aplicația web folosită de utilizatorii finali (membrii sau clienți ai unei organizații). În documentația pentru Hyperledger Composer este sugerat Angular ca framework pentru dezvoltarea de aplicații web, dar acest lucru rămâne la alegerea dezvoltatorului. Dacă Angular este opțiunea aleasă, Composer pune la dispoziție un schelet al aplicației, folosind informațiile arhivate în fișierul .bna, generat cu ajutorul Yeoman.

Angular este o platformă și un framework pentru crearea de aplicații web folosind HTML și Typescript. Framework-ul conține un set de librării scrise folosind Typescript, unele dintre ele principale și unele opționale, care pot fi folosite sau extinse în procesul de dezvoltare al aplicației. Angular folosește Model-View-Controller ca model arhitectural.

**Model-View-Controller** este un model arhitectural utilizat, în general, în dezvoltarea aplicațiilor web. Separă structura aplicației în trei componente logice individuale care interacționează una cu cealaltă promovând astfel ideea de a scrie cod ușor de întreținut și reutilizabil. Cele trei componente logice sunt Model, View și Controller.

Modelul definește ce date ar trebui să conțină aplicația. Dacă starea acestor date se modifică, atunci modelul va notifica de obicei controller-ul (astfel încât view-ul să poate fi actualizat). Modelul poate fi un singur obiect sau o structură de obiecte.

View-ul cuprinde componentele care afișează interfața cu utilizatorul (UI) și modul în care utilizatorii interacționează cu aplicația. În general, este realizat cu HTML, CSS, Javascript și adesea șabloane. View-ul nu va afișa nimic de la sine. Controller-ul sau modelul transmit view-ului ce date trebuie afișate pentru utilizator.

Controller-ul conține logica care actualizează modelul și / sau view-ul ca răspuns la informațiile primite de la utilizatori.

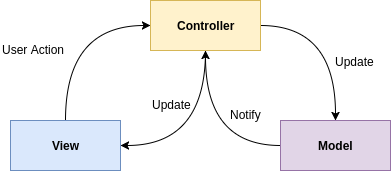


Figura 3.6 Componentele logice ale modelului architectural MVC

Principalele avantaje pe care le oferă MVC sunt:

* cod reutilizabil și extensibil;
* separarea logicii de prezentare de logica business-ului;
* permite lucrul simultan între dezvoltatori care sunt responsabili pentru diferite componente;
* ușor de întreținut.

Elementele de bază ale unei aplicații Angular sunt modulele, componentele și serviciile.

Modulele sunt blocurile de bază, care oferă un context de compilare pentru componente. Modulele colectează cod similar în seturi funcționale; o aplicație Angular este definită de un set de module. O aplicație are întotdeauna cel puțin un modul rădăcină (eng. root module) care permite lansarea în execuție și, de obicei, are mai multe module care conține feature-uri.

Componentele definesc view-uri, care sunt seturi de elemente afișate în interfața cu utilizatorul (UI) pe care Angular le poate alege și modifica în funcție de logica programului și de date. Fiecare aplicație are cel puțin o componentă rădăcină (eng. root component).

Dacă un set de componente conțin logică și / sau funcționalități comune, acestea pot fi extrase într-un serviciu și partajat între componente. Serviciile pot fi “injectate” în componente ca dependințe, având ca rezultat un cod modular, reutilizabil și eficient. Atât componentele cât și serviciile sunt pur și simplu clase, cu decoratori care marchează tipul lor și care oferă metadate despre cum pot fi folosite.

Metadatele asociază o clasă de componente cu un șablon. Un șablon combină cod HTML obișnuit, cu directivele oferite de Angular și cu datele introduse în view cu ajutorul procesului de data-binding. Această combinație permite modificarea codului HTML înainte de a fi afișat în pagină.

Metadatele pentru o clasă de servicii furnizează informațiile necesare pe care Angular trebuie să le pună la dispoziția componentelor prin intermediul Dependency Injection (DI).

Ca și modulele Javascript, modulele Angular pot importa funcționalități din alte module, și pot permite ca propriile lor funcționalități să fie exportate și utilizate de alte module. Organizarea codului în module funcționale distincte ajută la gestionarea dezvoltării de aplicații complexe și la proiectarea pentru reutilizare.

Decoratorul @Component identifică clasa imediat sub el ca o componentă și furnizează șablonul și metadatele specifice ale componentelor asociate.

Decoratorii sunt funcții care modifică clasele Javascript. Angular definește un număr de astfel de decoratori care atașează anumite categorii de metadate claselor, astfel încât să știe ce înseamnă aceste clase și cum ar trebui să funcționeze.

O definiție a unei clase de servicii este imediat precedată de decoratorul @Injectable. Decoratorul furnizează metadatele care permit serviciului să fie injectat în componente ca dependință.

Aplicația Angular și componentele necesare pot fi generate pe baza definiției rețelei folosind managerul de pachete Yeoman.

Un manager de pachete este un instrument pentru automatizarea procesului de instalare, actualizare, configurare și gestionare a dependințelor pentru proiecte. Exemplele includ npm (Node.js) Bower (Web), Pypi (Python), Gem (Ruby), Composer (PHP), NuGet (.NET), etc.

După instalarea aplicației web, aceasta poate apela REST API-ul generat de REST server-ul Composer, stabilind astfel legătura între cele două nivele arhitecturale.

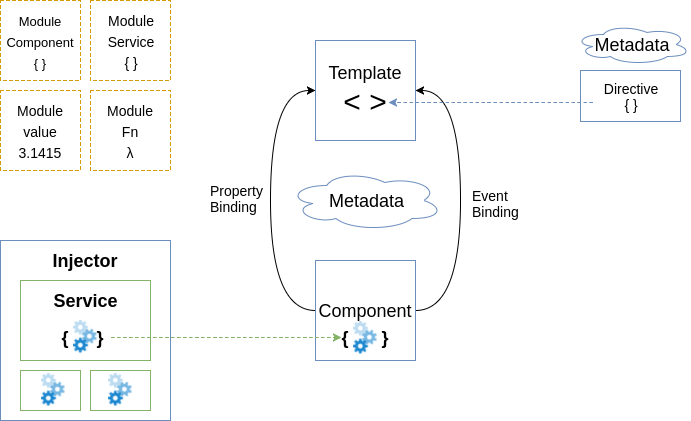


Figura 3.7 Descrierea arhitecturii unei aplicații Angular

## Implementare

Procesul de implementare include toate activitățile implicate în dezvoltarea software-ului de la cerințele inițiale ale sistemului până la gestionarea și întreținerea sistemului implementat.

Principalele aspecte ale implementării, care sunt deosebit de importante pentru ingineria software, sunt: reutilizarea software-ului existent, posibilitatea de adaptare la schimbări ale cerințelor, dezvoltarea de software capabil să ruleze pe diferite medii de dezvoltare prin intermediul configurărilor (development, production).

Prin reutilizarea software-ului existent, se pot dezvolta sisteme noi mai rapid, cu mai puține riscuri și de asemenea cu costuri reduse.

Aplicația dezvoltată în cadrul acestei lucrări folosește Hyperledger Composer (utilizat pentru a crea definiția rețelei de afaceri) și Hyperledger Fabric (utilizat pentru a rula/lansa rețelele de afaceri în mediul de dezvoltare local). Pentru instalarea acestor instrumentelor de dezvoltare sunt necesare următoarele cerințe preliminare specificate în documentația disponibilă online:

* Sistem de operare: Ubuntu Linux 14.04 / 16.04 LTS (64 de biți) sau Mac OS 10.12;
* Docker Engine: Versiunea 17.03 sau o versiune ulterioară;
* Docker-Compose: Versiunea 1.8 sau o versiune ulterioară;
* Node: 8.9 sau mai mare (versiunea 9 nu este acceptată);
* npm: v5.x;
* git: 2.9x sau mai mare;
* Python: 2.7.x;
* Un editor de text la alegere (VSCode este recomandat datorită plugin-ului pentru Hyperledger Composer care oferă evidențierea sintaxei și recomandări de cod).

După instalarea cerințelor preliminare se poate trece la instalarea mediului de dezvoltare, explicată în [12].

Printre instrumentele disponibile cu instalarea mediului de dezvoltare se numără și Playground. Playground-ul oferit de Hyperledger Composer este o interfață grafică cu utilizatorul care permite configurarea, implementarea și testarea unei rețele de afaceri.

Playground-ul poate fi lansat folosind comanda *composer-playground* și poate fi accesat la adresa <http://localhost:8080/login> dacă mediul de dezvoltare a fost instalat corespunzător. Această interfață va fi folosită pentru a construii definiția rețelei de afaceri. În definiția rețelei trebuie incluse modelele, logica tranzacțiilor (contracte inteligente), și setul de reguli de acces.

Modele vor fi construite după diagrama de clase identificată în procesul de proiectare. Pentru a identifica funcțiile necesare și rolul acestor funcții în logica tranzacțiilor se consideră diagrama secvențială din figura 3.8.

Modelele împreună cu funcțiile, care alcătuiesc logica tranzacțiilor, vor devenii endpoint-uri pentru serverul REST.

Setul de reguli de acces ar trebui să limiteze accesul participanților la tranzacțiile pe care le pot executa. Cei trei participanți ai aplicației sunt cumpărător, expeditor și producător. Un prim pas ar fi eliminarea “regulii generice” care spune că orice participant are acces la orice tranzacție. Din punct de vedere al business-ului, lanțul de aprovizionare pentru bunuri perisabile, singurul participant care ar putea crea comenzi este cumpărătorul. În mod similar singurul participant care ar putea crea contracte este producătorul. Definirea acestui set de reguli elimină cazurile marginale care ar trebui incluse în procesul de testare. Un exemplu de regulă, pentru situația cumpărătorului, este:

rule importerPermission {

description: "Importer can access Shipment"

participant: " org.acme.shipping.perishable.Importer"

operation: ALL

resource: " org.acme.shipping.perishable.Shipment"

action: ALLOW

}

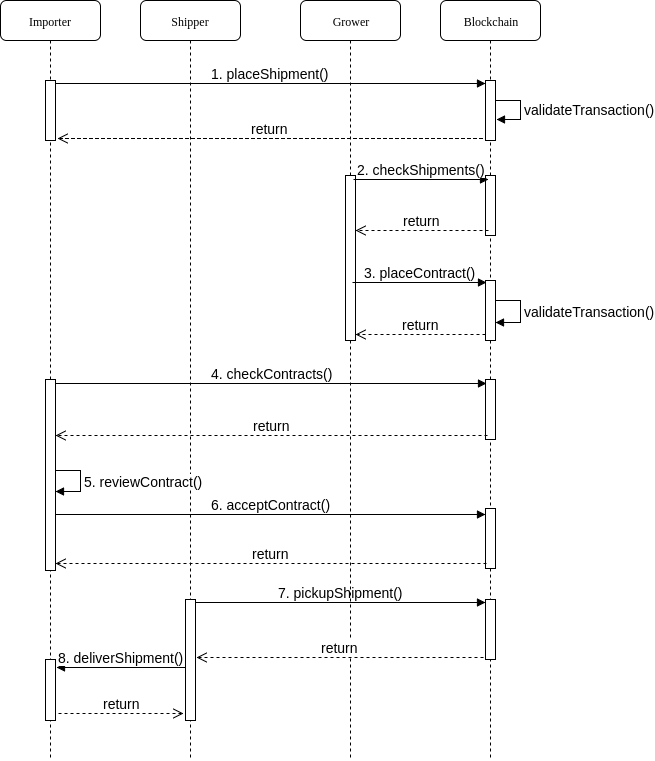


Figura 3.8 Diagrama secvențială între participanții rețelei

Următorul pas îl reprezintă generarea definiției rețelei de afaceri (BND) prin funcția de export. Hyperledger Composer permite actualizarea acestei definiții, dacă cerințele se schimbă, cu condiția ca versiunea să fie diferită. Pentru actualizarea rețelei, se încarcă fișierul .bna exportat în interfața Playground și se fac modificările necesare. La finalul procesului, se va reexporta fișierul .bna cu versiunea incrementată.

Fișierul .bna exportat va fi folosit de Hyperledger Fabric, instalat odată cu mediul de dezvoltare, pentru a aduce online rețeaua blockchain. Ca rețeaua să fie accesibilă pentru participanți, următorul pas este generarea de identități pentru aceștia. Identitățile (împreună cu cardurile de acces) pot fi generate folosind utilitarul *composer*. Acesta poate fi utilizat pentru a efectua mai multe sarcini administrative, operaționale și de dezvoltare. Comenzile disponibile includ lansarea în execuție a rețelei, generarea cardurilor de acces, generarea identităților, etc. și sunt disponibile online la adresa [13].

Tot cu ajutorul utilitarului *composer* se va genera serverul REST, pe baza definiției rețelei exportată anterior, folosind comanda *composer-rest-server*. Serverul REST expune un REST API accesibil la adresa [http://localhost:3000/explorer](http://localhost:3000/explorern). Într-un mediu de producție vor exista mai multe instanțe ale server-ului REST între care vor fi distribuite solicitările primite de la utilizatori pentru a asigura un timp de răspuns cât mai mic și o disponibilitate de 99% a sistemului. Endpoint-urile existente pot fi testate prin intermediul unui client REST sau din interfața grafică Swagger UI.

Având disponibil serverul REST următorul pas consta în crearea unei aplicații web care să extragă date folosind REST API-ul și să le prezinte utilizatorilor. Deoarece aplicația dezvoltată în cadrul acestei lucrări are trei categorii de participanți, se vor implementa trei aplicații web, corespunzătoare fiecărei categorii (cumpărător, expeditor, producător). Aplicațiile vor avea aceeași structură, dar funcționalități diferite.

Pentru a simplifica procesul de development și pentru a respecta coding guidelines, care spun că software-ul ar trebui reutilizat dacă este posibil, se va folosii **Git** ca sistem de control al versiunilor, se va crea o singură aplicație care conține structura de bază comună între cele trei și se va comite, folosind Git, pe trei ramuri distincte. În acest fel se reduce timpul de implementare și se păstrează un echilibru în privința librăriilor și tool-urilor folosite pentru cele trei aplicații.

Structura comună a celor trei aplicații constă în două componente care lucrează împreună: front-end și back-end.

Front-end-ul a fost realizat folosind Angular 6, și Bootstrap 4 pentru îmbunătățirea aspectului vizual. Scheletul unei aplicații Angular funcționale poate fi generat, folosind managerul de pachete *npm* și utilitarul *angular-cli,* împreună cu pachetele și dependințele necesare (jquery, popper.js, bootstrap, font-awesome, etc.) gata de a fi lansat în execuție.

Acest schelet conține modulul rădăcină al aplicației, componenta rădăcină și un șablon (HTML) care este afișat la momentul rulării. Aplicațiile care folosesc Angular sunt de tip single-page application (SPA).

O aplicație single-page este o aplicație web sau un site web care interacționează cu utilizatorul prin rescrierea dinamică a paginii curente decât prin încărcarea unor pagini noi dintr-un server. Această abordare evită întreruperea utilizatorilor între pagini succesive. Într-o aplicație single-page, resursele necesare – HTML, CSS și Javasript – sunt preluate cu o singură încărcare a paginii, sau sunt încărcate dinamic și adăugate la pagină, după cum este necesar, de obicei ca răspuns la acțiunile utilizatorului. Interacțiunea cu aplicația single-page implică de multe ori o comunicare dinamică cu serverul web din spatele scenei. Angular realizează rescrierea dinamică a paginii prin intermediul componentelor și a șabloanelor. În pachetul generat există un șablon principal, denumit în mod comun *index.html,* care conține, de obicei, elementele comune șabloanelor (header, footer) și declarațiile necesare (fișiere .css, .js). Componentele își inserează propriile șabloane în interiorul *index.html* ca răspuns al interacțiunilor utilizatorului.

Datele necesare afișării acestui conținut dinamic au fost preluate din modelele create conform diagramei de clase prezentată în sub-capitolul 3.1. Proprietățile acestor modele sunt populate folosind serviciile injectate în componente ca dependințe. Serviciile reprezintă mijlocul de comunicare cu back-end-ul și de partajare a datelor între componente. Majoritatea aplicațiilor de tip front-end comunică cu serviciile din back-end prin intermediul protocolului HTTP. Clasa *HttpClient* din pachetul *@angular/common/http* oferă implementarea necesară pentru a comunica cu back-end-ul transmițând mai departe solicitările lansate prin interacțiunea utilizatorului cu interfața grafică.

Back-end-ul a fost realizat folosind framework-ul Node.js. Node este un framework Javascript, bazat pe evenimente asincrone, care a fost proiectat pentru a construi aplicații web scalabile. Printre pachetele oferite de Node pentru dezvoltarea de aplicații web se numără și Express.js. Principalele avantaje oferite de Express sunt:

* tratarea diferită a request-urilor în funcție de diferite verbe HTTP și de adresa URL;
* generarea dinamică a răspunsurilor prin introducerea datelor în șabloane;
* setarea configurărilor obișnuite ale unei aplicații web, cum ar fi portul pe care îl utilizează pentru conectare și locația șabloanelor;
* posibilitatea de a interacționa cu request-ul în procesul de tratare a cererii.

O aplicație web clasică așteaptă request-uri HTTP de la un browser (sau de la orice client HTTP). Când un request este recepționat, aplicația determină ce acțiuni sunt necesare pe baza modelului adresei URL și a eventualelor informații conținute în datele trimise prin GET sau POST. În funcție de acțiunea care trebuie luată, aplicația poate citii sau scrie informații într-o bază de date sau poate efectua calcule necesare pentru a satisface cererea primită. Aplicația va returna apoi un răspuns pe care browser-ul web, sau front-end-ul în cazul de față, îl va prelua și îl va folosii pentru a crea dinamic o pagină HTML prin introducerea datelor preluate într-un șablon HTML.

Express.js realizează aceste sarcini prin intermediul procesului de “rutare” (eng. routing). Rutarea se referă la determinarea modului în care o aplicație răspunde solicitării unui client către un anumit endpoint, care este o adresă URL (sau o cale) și un verb specific HTTP (GET, POST, etc.).

Figură

Pentru ca aplicația web să poată fi accesată de utilizatori, colecția de fișiere .html, .css, și .js este expusă cu ajutorul serverului web creat de Express.js.

Având scheletul celor trei aplicații, următorul pas îl reprezintă dezvoltarea funcțiilor specifice pe care trebuie să le ofere aplicațiile utilizatorilor. Fiecare dintre cele trei aplicații va fi folosită de o categorie diferită de participanți. În procesul de implementare trebuie ținut cont de cerințele funcționale ale aplicației, de cerințele non-funcționale și de metodele identificate în procesul de proiectare. Datele necesare tranzacțiilor (spre exemplu termenii din contract) vor fi introduse de utilizatori, vor fi preluate de aplicația web și tratate corespunzător. Ținând cont de implicarea factorului uman, datele trebuie validate înainte de a fi preluate. Se va ține cont și de logica business-ului, sarcinile efectuate în cadrul unui lanț de aprovizionare pentru bunuri perisabile, astfel încât aplicația să ofere o simulare cât mai aproape de realitate.

Se consideră în continuare cazurile de utilizare care vor fi traduse în funcționalități ale aplicației web.

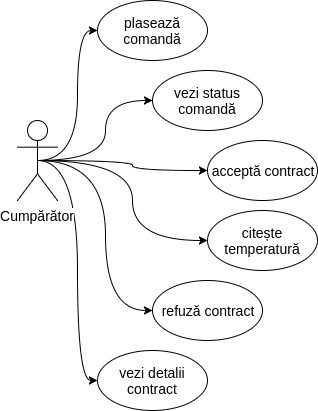


Figura 3.9 Cazurile de utilizare pentru cumpărător

În figura 3.9 sunt reprezentate cazurile de utilizare pentru cumpărător. Cumpărătorul este cel care inițiază întreg procesul prin acțiunea de plasare a unei comenzi. Statusul comenzii este vizibil pentru cumpărător pe tot parcursul procesului împreună cu detaliile contractului agreat cu producătorul. După revizuirea contractelor, cumpărătorul poate accepta unul singur, dar poate refuza oricare dintre ele. După ce procesul de livrare a fost inițiat de expeditor, se pot solicita informații referitoare la temperatura comenzii. Toate aceste funcționalități explicate mai sus pot fi observate și în figura 3.10.

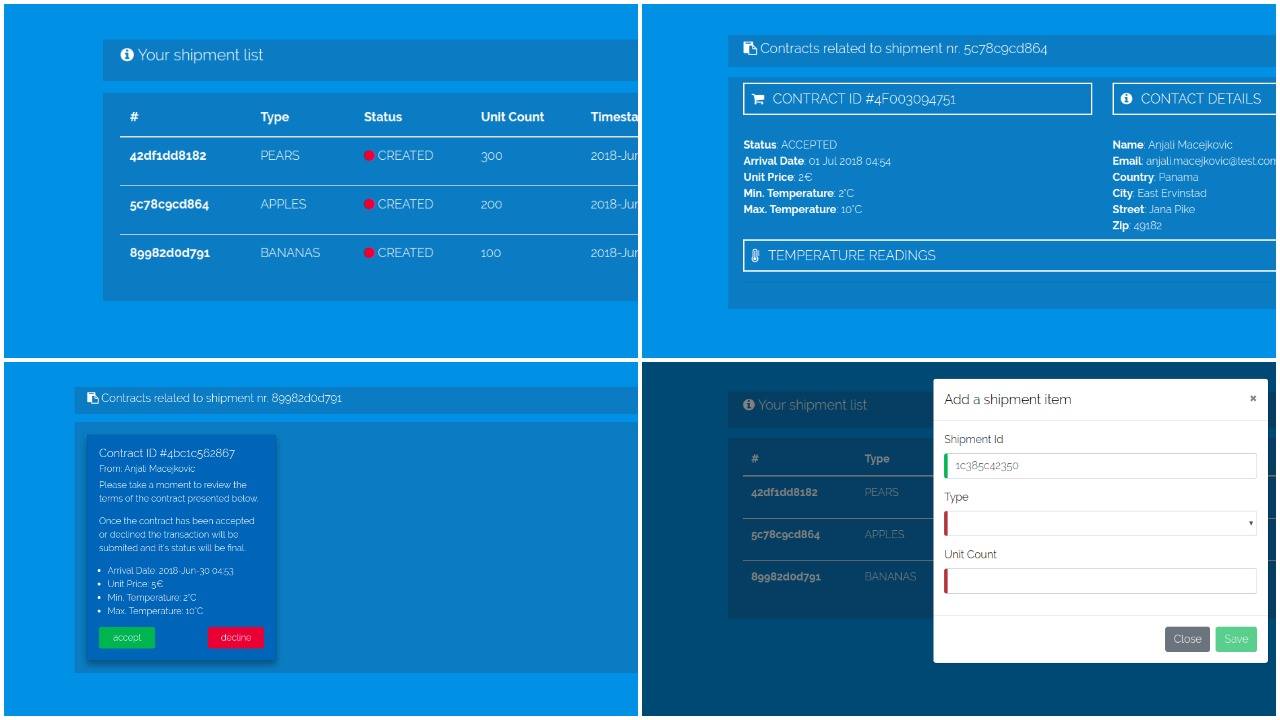


Figura 3.10 Perishable Goods Network – interfața cumpărătorului

În figura 3.11 sunt reprezentate cazurile de utilizare pentru producător. Producătorul este cel care primește comenzile plasate de cumpărător și stabilește un contract pe baza acestor comenzi. Nu se pot plasa contracte pentru comenzile care au fost acceptate.

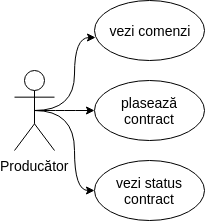


Figura 3.11 Cazurile de utilizare pentru producător

De asemenea, nu se pot plasa mai multe contracte pentru aceeași comandă de la același producător. Producătorul poate vedea statusul contractului pe tot parcursul procesului. Toate aceste funcționalități explicate mai sus pot fi observate și în figura 3.12.

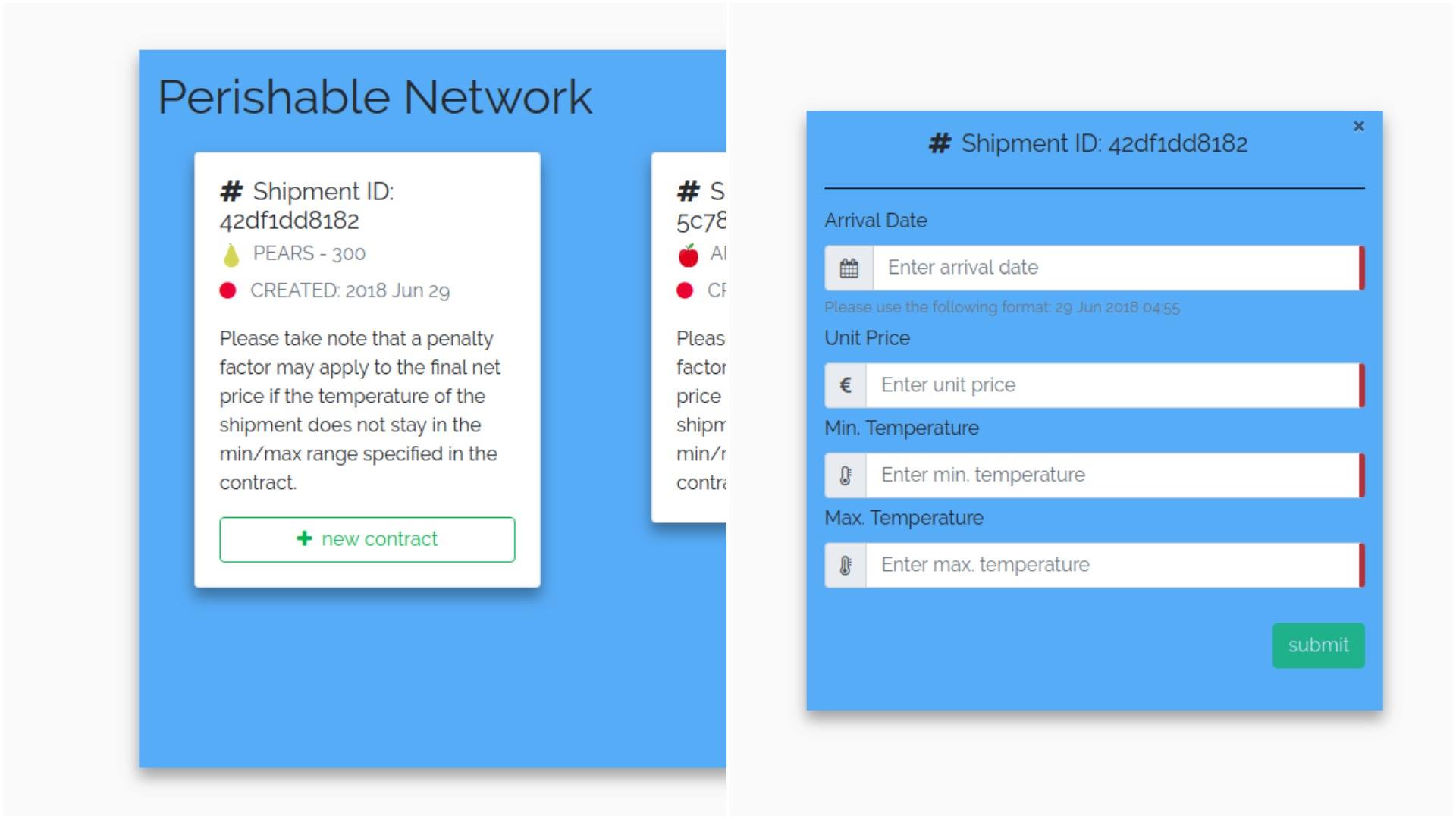


Figura 3.12 Perishable Goods Network – interfața producătorului

Expeditorul este cel care face legătura între cumpărător și producător. Cazurile de utilizare pentru expeditor pot fi observate în figura 3.13. Expeditorul are vizibilitate doar asupra contractelor agreate. Simularea procesului de livrare constă în execuția unei serii de tranzacții prin care statusul livrării și al contractului sunt actualizate ca părțile implicate să fie informate referitor la starea comenzii. La momentul terminării simulării se calculează costul livrării ținând cont de o serie de factori: costul comenzilor care au fost livrate după data specificată în contract va fi 0; depășirea limitelor de temperatură minime și maxime admise se vor scade din costul total al comenzii.

Toate aceste funcționalități explicate mai sus pot fi observate și în figura 3.14

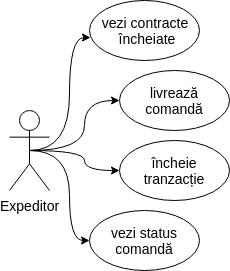


Figura 3.13 Cazurile de utilizare pentru expeditor

După încheierea procesului, se pot verifica datele manipulate prin intermediul tranzacțiilor. Cu ajutorul REST API-ului se poate verifica ca datele prezentate utilizatorului în cele trei interfețe grafice corespund cu datele salvate în blocurile rețelei, asigurând astfel integritatea datelor. Se pot verifica, de asemenea, hash-ul tranzacțiilor împreună cu timestamp-ul acestora. Endpoint-urile care manipulează tranzacții conțin doar logica pentru a returna tranzacții sau pentru a crea tranzacții noi, corespunzătoare verbelor HTTP GET și POST. Neavând posibilitatea de a modifica sau șterge tranzacțiile existente este asigurată proprietatea de imutabilitate a datelor, oferită de folosirea blockchain-ului pentru a persista datele.

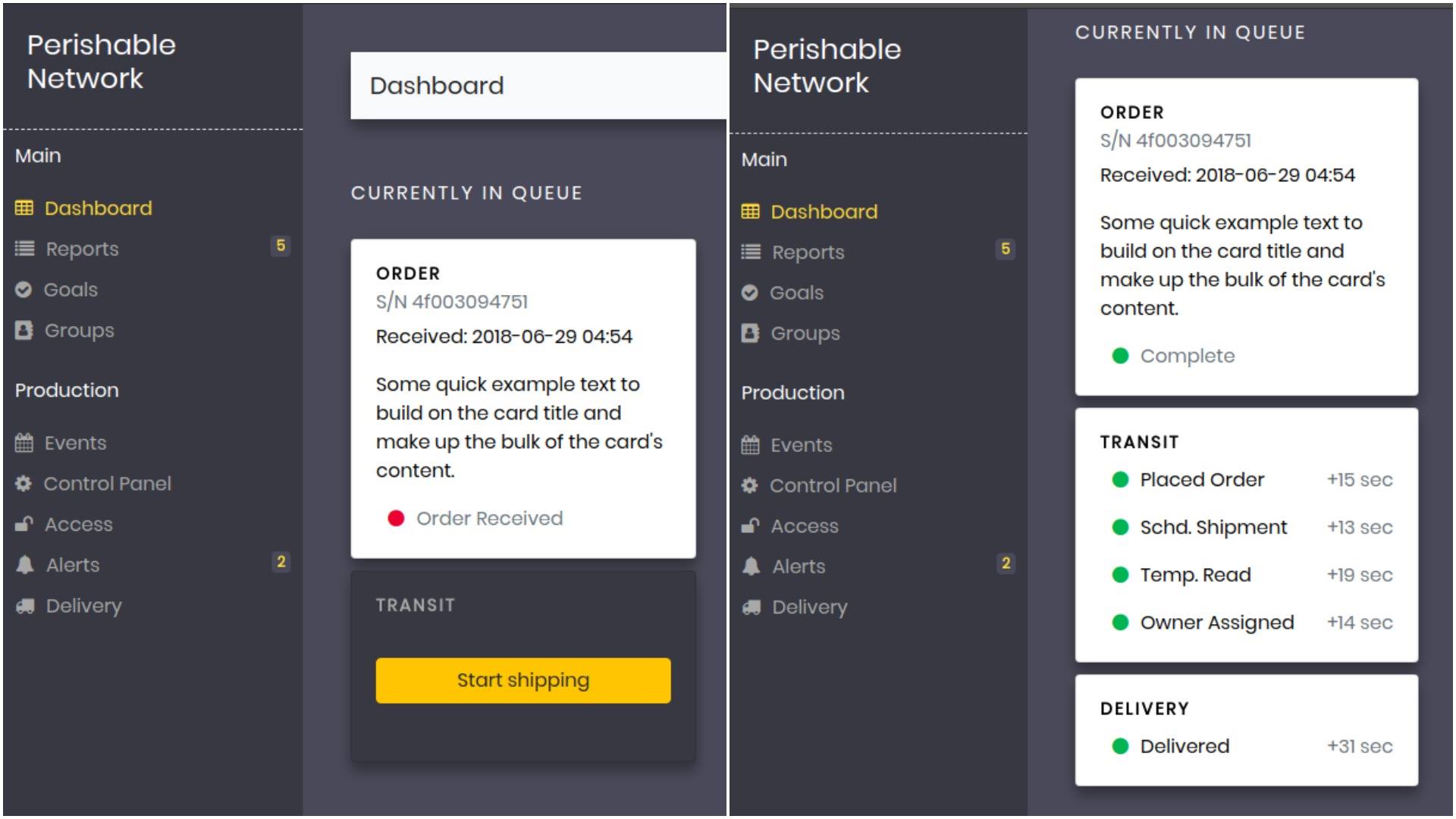


Figura 3.14 Perishable Goods Network – interfața expeditorului

Aceleași informații pot fi verificate folosind interfața Playground oferită de Hyperledger Composer.

## Testare și validare

Testarea are rolul de a arăta că un program funcționează conform specificațiilor și de a descoperii defectele programului înainte de a fi pus în funcțiune. Procesul de testare are două obiective distincte: fiind dat un set de intrări care reflectă comportamentul normal al sistemului, acesta să funcționeze conform așteptărilor; fiind dat un set anormal de date (sau date aparținând unor cazuri marginale), se caută erori sau defecte ale sistemului.

Procesul de testare se împarte în două categorii: testarea manuală și testarea automată. În practică, procesul de testare implică un amestec între testarea manuală și testarea automată. În cazul testării manuale este nevoie ca o persoană (tester) să execute programul cu un set de date de test și să compare rezultatele cu așteptările lor. Testarea automată este realizată pe baza unui program care execută o suită de teste pentru a verifica dacă programul funcționează corect și dacă modificările aduse programului nu au introdus erori noi. Testarea automată este, în mod obișnuit, mai rapidă decât testarea manuală.

Scopul procesului de verificare și validare este de a asigura că sistemul îndeplinește cerințele sale funcționale și non-funcționale.

### Testarea manuală

Procesul de testare “tradițional” se realizează după un set de cazuri de testare (eng. test cases). Cazurile de testare conțin, în mod obișnuit, intrările care urmează sa fie testate, o scurtă descriere a ceea ce este testat și rezultatul așteptat.

Tabel 3.1 Caz de testare exemplificat

| Descriere | Date de intrare | Date de ieșire |
| --- | --- | --- |
| Se va testa că timestamp-urile sunt salvate după formatul yyyy-MMM-dd | 1530303460870 | 2018-Jun-01 |

Pentru a verifica acest caz, se va pornii aplicația, se va căuta o funcționalitate care primește timestamp-uri ca date de intrare (ex. form-ul prin care producătorul poate introduce contracte) și se va executa această funcționalitate în speranța ca rezultatul este același cu datele de ieșire. O funcționalitate simplă a sistemului, cum ar fi plasarea unei comenzi ar putea cuprinde zeci de cazuri de testare. În general, testarea manuală cuprinde atât funcționalitatea cât și aspectul și modul în care se interacționează cu aplicația.

Pentru a testa manual aplicația dezvoltată în cadrul acestei lucrări se pot folosi atât interfața aplicației web implementată în Angular cât și mediul de testare Playground oferit de Hyperledger Composer.

Testarea REST API-ului se poate face folosind orice client REST (ex. Postman), folosind utilitarul cURL, folosind Swagger UI sau chiar direct din browser. Toate request-urile vor conține URL-ul de bază [*http://localhost:3000/api*](http://localhost:3000/api)și verbul HTTP corespunzător (GET, POST, etc).

Morgan

### Testarea automată

Procesul de testare automată poate fi împărțit în două categorii:

* unit testing;
* integration testing.

Unit testing, sau testarea pe unități, este procesul de testare a componentelor software-ului, cum ar fi metodele sau clasele de obiecte. Funcțiile sau metodele individuale reprezintă cel mai simplu tip de componentă.

Integration testing, sau testarea integrării, este procesul de testare a software-ului în care modulele individuale sunt combinate și testate ca grup. Acest tip de test este mai lent decât unit testele deoarece implică, de obicei, operații cu baza de date, cu hard-disk-ul sau cu o altă componentă externă.

Se pot scrie teste atât pentru fișierele Javascript care alcătuiesc logica tranzacțiilor din definiția rețelei cât și pentru componentele Angular.

Un test automat are trei părți:

* o parte de configurare, în care se inițializează sistemul cu cazul de test, și anume intrările și ieșirile așteptate;
* o parte de apel, unde se apelează obiectul sau metoda care urmează să fie testată;
* o parte de afirmație, în care se compară rezultatul apelului cu rezultatul așteptat.

Se consideră următorul caz de test: “În cazul unui transport întârziat, producătorul nu primește nimic”.

Exemplu test

# Concluzii

## Rezultate obținute

În această lucrare s-a discutat despre tehnologiile blockchain , implicațiile pe care le au aceste tehnologii în industria actuală, avantajele pe care le oferă față de soluțiile existente de stocare a datelor și limitările care apar în procesul de implementare.

Prin parte a lucrării conține informații și suport teoretic despre ce înseamnă tehnologia blockchain, modul de funcționare, rezolvările pe care le-a adus la probleme clasice din ingineria sistemelor distribuite și evoluția pe care a avut-o până în prezent. Ideea de bază a fost prezentarea modului în care se pot dezvolta sisteme noi folosind această tehnologie sau a modului în care pot fi integrate sistemele existente. S-a discutat despre Bitcoin, Ethereum și Hyperledger, numele cele mai reprezentative în discuțiile legate de tehnologia blockchain, mai exact ce a adus fiecare de la lansarea inițială a generației 1.0.

Chiar dacă atunci când a fost lansată domeniul de interes a fost cel financiar, există posibilități nenumărate de a fi folosită și în alte domenii cum ar fi automobile, turism, industrie, vânzări, management, medical, telecomunicații, etc. Aplicația dezvoltată în cadrul acestei lucrări, managementul unui lanț de aprovizionare, este un exemplu practic de aplicare a tehnologiei în afara domeniului economic.

Cea de-a doua parte a lucrării constă în etapele de dezvoltare a unei aplicații software folosind Hyperledger Composer. S-a făcut o analiză asupra opțiunilor existente care pot fi folosite pentru a dezvolta o aplicație full-stack de management al unui lanț de aprovizionare cât și o analiză detaliată a modului de funcționare al business-ului. Care sunt participanții, ce rol au fiecare, care este modul de comunicare al acestora și ce ar aduce în plus tehnologia blockchain. Pe baza acestei analize s-au stabilit cerințele funcționale și non-funcționale ale aplicației. După identificarea cerințelor s-a trecut la implementarea efectivă a aplicației.

La implementare, s-a discutat despre tehnologiile care au fost alese, modul de lucru al fiecăreia și s-a justificat alegerea făcuta prin exemple practice. În alegerea tehnologiilor s-a ținut cont de următoarele aspecte: popularitatea tehnologiei (câte din sistemele actuale folosesc această tehnologie), posibilitatea de integrare cu restul tehnologiilor alese, flexibilitatea (să se poată adapta ușor la schimbări ale cerințelor) și sugestiile oferite de documentația Hyperledger Composer.

În procesul de implementare s-a ținut cont de metodologiile și de guideline-urile existente și s-a încercat refolosirea codului care a fost deja implementat și testat în cadrul altor sisteme. Aplicația rezultată este un exemplu practic de folosire a tehnologiei blockchain într-un domeniu de interes al industriei, justificând tema aleasă. Această aplicație oferă utilizatorilor un mod de a interacționa cu blockchain-ul și de a testa, din punct de vedere al business-ului, procesul de management al unui lanț de aprovizionare, prin intermediul celor trei interfețe grafice. Se pot plasa comenzi, se pot crea contracte și există posibilitatea de a simula livrarea acestor comenzi, din punctul de vedere al participanților. Pe lângă testele manuale s-a discutat și introducerea testelor automate necesare atunci când se aduc schimbări aplicației.

În concluzie au fost acoperite toate aspectele teoretice și practice menționate în această lucrare.

## Direcții de dezvoltare

Există numeroase posibilități prin care aplicația ar putea fi dezvoltată, de la îmbunătățirea performanțelor actuale și până la extinderea în domeniul IoT adăugând funcționalități noi.

Pentru îmbunătățirea performanțelor actuale s-a discutat în cadrul lucrării de folosirea mai multor instanțe ale serverului REST. Prin folosirea mai multor instanțe se reduce timpul mediu de răspuns și pot fi tratate mai multe solicitări primite de la utilizatori într-un interval mai scurt de timp. Gestionarea mai multor instanțe implică folosirea unui **load-balancer**.

Procesul de load-balancing se referă la distribuirea eficientă a traficului rețelei într-un grup de servere de backend. **NGINX** este printre cele mai populare și eficiente servicii de load-balancing și îndeplinește următoarele funcționalități:

* distribuie solicitările primite de la clienți sau încărcarea rețelei în mod eficient pe mai multe servere;
* asigură o disponibilitate și o fiabilitate ridicată prin redirecționarea cererilor numai către serverele online;
* oferă flexibilitatea de a crește sau de a descrește numărul de servere în funcție de cerere.

Tot pentru a contribui la capitolul performanțe, Angular a introdus funcționalitatea de **lazy-loading**.

Lazy-loading este o tehnică prin care sunt excluse resursele care nu sunt necesare la momentul încărcării paginii. În schimb, aceste resurse sunt încărcate în momentul în care sunt solicitate de utilizator. Cele trei interfețe grafice implementate folosind Angular conțin un număr mare de resurse (.html, .css, .js) care produc o încărcare lentă a paginilor. Implementarea funcționalități de lazy-loading este explicată în [14].

Momentan aplicația a fost implementată și testată în mediul de dezvoltare local. Pe viitor s-ar putea opta pentru folosirea unui serviciu cloud. IBM, fondatorul proiectului Hyperledger, oferă un astfel de serviciu pentru rețelele implementate folosind Hyperledger Fabric. Acest serviciu se numește IBM Cloud, dar momentan nu poate fi folosit gratuit. Alte opțiuni ar fi CloudFoundry care oferă serviciul **Blockchain-As-A-Service** și Microsoft Azure care au anunțat anul acesta că platforma lor va suporta și soluții blockchain.

Pentru a extinde funcționalitățile curente ale aplicației există posibilitatea de integrare cu tehnologii IoT prin intermediul serverului REST.

Ca exemplu, plăcile de dezvoltare Raspberry PI pot folosii Node.js. Se poate face achiziția a mai multor tipuri de date (umiditate, temperatură, etc.) cu ajutorul senzorilor montați la momentul livrării iar cu ajutorul Raspberry PI se pot trimite aceste informații către server-ul REST pentru a fi afișate utilizatorilor.

Există nenumărate direcții de dezvoltare, tehnologia fiind în continuă evoluție, iar pe viitor ar putea fi principala alegere a industriilor ca mijloc de stocare a datelor.

# Reguli de formatare

## Formatarea paginii

* + Dimensiunea paginii: A4
  + Margini: 2.5 cm (sus, jos, stânga, dreapta)
  + Antet și subsol: 1.27 cm de la marginea paginii
  + În antetul paginii (header): titlul capitolului, centrat, stil: Header\_style
  + În subsolul paginii: numărul paginii, centrat

## Titluri și stiluri

Titlurile capitolelor și subcapitolelor se marchează cu stilurile Heading 1 – 4, conform documentului model anexat în format Word. Descrierea stilurilor utilizate în document este prezentată în Tabelul 5.1.

Tabelul 5.1. Stiluri utilizate în acest document

| Nr. | Stil | Utilizat pentru | Format |
| --- | --- | --- | --- |
| 1 | Normal | Text normal | Font: (Default) Cambria, 12 pt, Justified, Line spacing: Multiple 1.1 li, Space After: 6 pt |
| 2 | Titlu | Titlul proiectului, prima pagină | Font: 24 pt, Small caps, Centered Line spacing: single, Space Before: 126pt, After: 0 pt, |
| 3 | Titlu2 | Titlul proiectului, pagina de prezentare | Font:14pt, Bold, Centered |
| 4 | Heading 1 | Titlurile capitolelor (nivel 1) | Font: 24 pt, Indent: Left: 0 cm Hanging: 0.76 cm, Space Before: 24pt, After: 12pt |
| 5 | Heading 2 | Titlurile subcapitolelor (nivel 2) | Font: 14 pt, Bold, Indent: Left: 0 cm  Hanging: 1.02 cm, Space Before: 18pt, After: 12pt |
| 6 | Heading 3 | Titlurile secțiunilor (nivel 3) | Font: Bold, Indent: Left: 0 cm Hanging: 1.27 cm, Space Before: 6 pt, After: 6pt |
| 7 | Heading 4 | Titlurile secțiunilor (nivel 4) | Font: Italic, Indent: Left: 0 cm Hanging: 1.52 cm, Space Before: 2 pt, After: 0 pt |
| 8 | Caption | Legenda figurilor și tabelelor | Font: Italic, Font color: Text 1, Line spacing: single, Space After: 10 pt, |
| 9 | Header\_style | Antetul paginii | Font: 10 pt, Italic, Centered, Border: Bottom: (Single solid line, Background 1, 0.5 pt Line width) |

## Figuri, tabele și ecuații

### Figuri

Figurile se inserează în text centrate, cu etichetă de numerotare și legendă (Caption) în partea de jos a figurii. Numărul figurii include și numărul capitolului, după exemplul prezentat în Figura 5.1.



Figura 5.1. Figură exemplu, stil: Caption

## Tabele

Tabelele se inserează în text centrate, cu etichetă și legendă (Caption) în partea de sus a tabelului, aliniată la stânga. Numărul tabelului include și numărul capitolului, după cum este prezentat, de exemplu, în Tabelul 5.1.

## Ecuații

Ecuațiile se inserează în text centrate, cu numerotare în partea dreaptă. Numărul ecuației include și numărul capitolului, conform exemplului din relația (5.1).

|  |  |
| --- | --- |
|  | (5.1) |

## Referințe bibliografice

Se recomandă ca citarea referințelor bibliografice să fie făcută în formatul IEEE.

În secțiunea Bibliografie sunt prezentate exemple pentru: o citare a unui capitol dintr-o carte [15], un articol publicat într-o revistă [16] și un articol publicat la o conferință [17].

Detalii cu privire la formatul citării diverselor tipuri de referințe pot fi găsite în [18] sau [19].

Referințele bibliografice se pot insera în text utilizând facilitățile Word de a adăuga surse și bibliografie unui document (References -> Citations & Bibliography). Dacă formatul IEEE pentru bibliografie nu este instalat implicit în Word, se poate descărca gratuit de la:

<https://bibword.codeplex.com/wikipage?title=Styles&referringTitle=Home>

Instrucțiunile de instalare pentru diferite versiuni de Word se pot obține de la aceeași adresă.

# Bibliografie

|  |  |
| --- | --- |
| [1] | The Economist, „Blockchains: The great chain of being sure about things,” 31 Octombrie 2015. [Interactiv]. Available: https://www.economist.com/briefing/2015/10/31/the-great-chain-of-being-sure-about-things. [Accesat 1 Iunie 2018]. |
| [2] | R. L. Rivest, „Cryptography,” în *Handbook of Theoretical Computer Science*, Elsevier, 1990, pp. 717-755. |
| [3] | I. Bashir, „Distributed systems,” în *Mastering Blockchain*, Birmingham, Packt Publishing, 2017, pp. 10-12. |
| [4] | P. Baran, „On Distributed Communications Networks,” RAND Corporation, Santa Monica, 1964. |
| [5] | R. Schollmeier, „A Definition of Peer-to-Peer Networking for the Classification of Peer-to-Peer Architectures and Applications,” în *Proceedings First International Conference on Peer-to-Peer Computing*, Linkoping, 2001. |
| [6] | B. Vitalik, „DAOs, DACs, DAs and More: An Incomplete Terminology Guide,” Ethereum, 6 Mai 2014. [Interactiv]. Available: https://blog.ethereum.org/2014/05/06/daos-dacs-das-and-more-an-incomplete-terminology-guide/. [Accesat 1 Iunie 2018]. |
| [7] | I. Allison, „PwC blockchain expert pinpoints sources of ambiguity in smart contracts,” IBTimes, 2016 August 12. [Interactiv]. Available: https://www.ibtimes.co.uk/pwc-blockchain-expert-pinpoints-sources-ambiguity-smart-contracts-1575778. [Accesat 1 Iunie 2018]. |
| [8] | S. Brakeville, „Blockchain basics: Introduction to distributed ledgers,” IBM, 18 Martie 2018. [Interactiv]. Available: https://www.ibm.com/developerworks/cloud/library/cl-blockchain-basics-intro-bluemix-trs/. [Accesat 1 Iunie 2018]. |
| [9] | I. Sommerville, „Requirements Engineering,” în *Software Engineering*, Boston, Addison-Wesley, 2004, pp. 82-96. |
| [10] | „Apache CouchDB,” Apache Software, 2012 Aprilie 15. [Interactiv]. Available: http://couchdb.apache.org/. [Accesat 1 Iunie 2018]. |
| [11] | Sun Microsystem, „Distributed Application Arhitecture,” 6 Aprilie 2016. [Interactiv]. Available: http://java.sun.com/developer/Books/jdbc/ch07.pdf. [Accesat 1 Iunie 2018]. |
| [12] | Hyperledger Composer, „Installing the development environment,” [Interactiv]. Available: https://hyperledger.github.io/composer/v0.16/installing/development-tools.html. [Accesat 1 Iunie 2018]. |
| [13] | Hyperledger Composer, „Hyperledger Composer Command Line,” [Interactiv]. Available: https://hyperledger.github.io/composer/v0.16/reference/commands.html. [Accesat 1 Iunie 2018]. |
| [14] | Angular, „Lazy Loading Feature Modules,” [Interactiv]. Available: https://angular.io/guide/lazy-loading-ngmodules. [Accesat 1 Iunie 2018]. |
| [15] | P. Nume, „Titlul capitolului,” în *Titlul cartii*, Oras, Editura, 2016, pp. 1-24. |
| [16] | P. Nume, „Titlul articolului,” *Titlul revistei,* vol. 1, nr. 2, pp. 22-30, 2016. |
| [17] | P. Nume, „Titlul articolului,” în *Numele conferintei*, Oras, 2015. |
| [18] | „IEEE Citation Reference,” 2009. [Interactiv]. Available: https://www.ieee.org/documents/ieeecitationref.pdf. |
| [19] | „IEEE Editorial Style Manual,” 2016. [Interactiv]. Available: https://www.ieee.org/documents/style\_manual.pdf. |