Horia-Alexandru MODRAN

# SOFTWARE PENTRU TELECOMUNICAȚII

Suport de curs

Editura Universității Transilvania din Brașov

2025

**EDITURA UNIVERSITĂȚII TRANSILVANIA DIN BRAȘOV**

| | |
|---|---|
| Adresa: | Str. Iuliu Maniu nr. |
| | 500091 Brașov |
| Tel.: | 0268 476 050 |
| Fax: | 0268 476 051 |
| E-mail: | editura@unitbv.ro |

**Editură recunoscută CNCSIS, cod 81**

**ISBN 978-606-19-1773-0 (ebook)**



Lucrarea a fost avizată de Consiliul Departamentului de Electronică și Calculatoare, Facultatea de Inginerie Electrică și Știința Calculatoarelor a Universității Transilvania din Brașov.

# Cuprins

PREFAȚĂ ..... 7

1. CAPITOLUL 1. INTRODUCERE – EVOLUȚIA DEZVOLTĂRII SOFTWARE ..... 9

1.1. Istoria și evoluția sistemelor de calcul ..... 9

1.2. Aplicații software: proprietare vs. open-source ..... 12

1.3. Evoluția de la Grid Computing la Cloud Computing ..... 14

2. CAPITOLUL 2. MODELUL DE COMUNICAȚIE CLIENT-SERVER ..... 19

2.1. Arhitectura client-server ..... 19

2.2. Componenta server ..... 20

2.3. Componenta client ..... 24

2.4. IP Suite ..... 26

2.5. TCP/IP ..... 28

2.6. Protocolul HTTP ..... 32

3. CAPITOLUL 3. ARHITECTURI BAZATE PE SERVICII ..... 41

3.1. Service Oriented Arhitecture (SOA) ..... 41

3.2. SOA – un stil arhitectural ..... 42

3.3. Fundamentele SOA ..... 44

3.4. Obiectivele și avantajele SOA ..... 47

3.5. Modelul Arhitectural MVC ..... 50

3.6. Simple Object Access Procotol (SOAP) ..... 54

4. CAPITOLUL 4. APLICAȚII MONOLITICE ȘI MICROSERVICII. ..... 59
4.1. Servicii Web ..... 59
4.2. Arhitectura monolitică ..... 61
4.3. Microservicii ..... 63
4.4. Exemplu practic microservicii ..... 70
5. CAPITOLUL 5. ARHITECTURA REST. ..... 73
5.1. Principiile REST ..... 73
5.2. Comunicarea client-server. Request și response ..... 76
5.3. Abordare REST ..... 80
5.4. REST vs SOAP ..... 85
5.5. JSON ..... 87
6. CAPITOLUL 6. API-URI RESTFUL. ..... 89
6.1. Istoria Web – de la Web 1.0 la Web 3.0 ..... 89
6.2. Transfer de date în contextul REST ..... 93
6.3. API-uri RESTful ..... 95
6.4. Organizarea resurselor ..... 97
6.5. Utilizare API RESTful ..... 101
6.6. Avantaje și Provocări ..... 102
7. CAPITOLUL 7. SECURITATEA SERVICIILOR WEB ..... 105
7.1. Securitate Web ..... 105
7.2. Cookie și autoritate ambientală ..... 107

7.3. Atacuri de sesiune ..... 110

7.4. Autorizare și autentificare ..... 114

8. CAPITOLUL 8. VIRTUALIZARE ȘI CONTAINERE DOCKER ..... 117

8.1. Virtualizare ..... 117

8.2. Docker ..... 119

8.3. Caracteristici și componente Docker ..... 122

8.4. Containerizarea aplicațiilor ..... 127

9. CAPITOLUL 9. SERVICII PENTRU IOT. PROTOCOLUL MQTT ..... 129

9.1. Protocolul MQTT ..... 129

9.2. Componente MQTT ..... 132

9.3. Mesajul MQTT și calitatea serviciului (QoS) ..... 135

9.4. Avantaje și dezavantaje MQTT ..... 137

9.5. Exemplu MQTT ..... 140

10. CAPITOLUL 10. CLOUD COMPUTING. SERVICII ȘI FURNIZORI ÎN CLOUD 143

10.1. Proprietăți Cloud Computing ..... 143

10.2. Servicii Cloud Computing ..... 145

10.3. Furnizori în Cloud ..... 151

BIBLIOGRAFIE ..... 157

# Prefață

Lucrarea de față a fost concepută ca suport de curs pentru studenții din anul 3 la specializarea Tehnologii și Sisteme de Telecomunicații din cadrul Facultății de Inginerie Electrică și Știința Calculatoarelor (IESC), oferind o privire amplă și integrată asupra celor mai importante concepte și tehnologii din dezvoltarea software modernă. Într-o eră în care digitalizarea și inovația tehnologică influențează constant modul în care comunicațiile se desfășoară, această carte se adresează atât studenților, cât și viitorilor profesioniști din industrie, pregătindu-i să facă față provocărilor unei lumi tot mai interconectate. Cartea îmbină teoria de bază cu exemple practice, studii de caz și detalii tehnice actualizate, astfel încât fiecare capitol să ofere o fundamentare solidă și să ilustreze aplicabilitatea conceptelor în scenarii reale.

Lucrarea abordează evoluția dezvoltării software și aplicații în domeniul telecomunicațiilor, pornind de la modelele clasice de comunicație client-server, trecând prin arhitecturi bazate pe servicii (SOA, REST, microservicii), până la aspecte avansate precum API-urile RESTful, securitatea serviciilor web, virtualizarea, containerele Docker și, nu în ultimul rând, cloud computing. Fiecare capitol este structurat pentru a facilita înțelegerea progresivă a temelor, evidențiind modul în care tehnologiile moderne se îmbină pentru a crea sisteme scalabile și eficiente. În plus, se pune accent pe interacțiunea dintre teoria fundamentală și aplicațiile practice, astfel încât studenții să poată dezvolta o viziune holistică asupra sistemelor IT actuale. Prin acest curs și disciplina „Proiect de Software pentru Telecomunicații" din anul IV, studenții sunt încurajați să își dezvolte abilitățile de analiză și sinteză și să aplice aceste cunoștințe în proiecte practice.

În final, lucrarea urmărește să servească drept ghid practic și teoretic pentru studenți, pregătindu-i să facă tranziția de la studiile academice la mediul profesional, unde vor avea ocazia să dezvolte și să implementeze tehnologii software moderne.

Sper ca această carte să devină un instrument valoros în parcursul academic al studenților, inspirând inovația și contribuind la formarea unei noi generații de specialiști în domeniul tehnologiilor și sistemelor de telecomunicații.

Astfel, dragi studenți, vă doresc un parcurs de succes și multă inspirație în explorarea și aplicarea cunoștințelor prezentate!

# 1. Capitolul 1. Introducere – Evoluția Dezvoltării Software

Dezvoltarea software a cunoscut o evoluție remarcabilă în ultimele decenii, influențând semnificativ toate aspectele vieții noastre de zi cu zi. În special în domeniul telecomunicațiilor, software-ul joacă un rol crucial, de la gestionarea rețelelor și serviciilor, până la dezvoltarea de aplicații inovatoare pentru utilizatori.

Acest capitol explorează evoluția dezvoltării software în telecomunicații, evidențiind aspectele cheie, tehnologiile de bază și impactul acestora asupra industriei.

## 1.1. Istoria și evoluția sistemelor de calcul

Începând cu perioada 1945-1985, când computerele erau de dimensiuni foarte mari și costisitoare, a avut loc o transformare radicală odată cu dezvoltarea microprocesoarelor. Aceasta a condus la apariția unor computere mai mici, mai performante și mai accesibile din punct de vedere al costului.

Evoluția puterii de calcul a fost una foarte spectaculoasă. Dacă în perioada 1945-1985 computerele erau masive și foarte scumpe, ocupând camere întregi, dezvoltarea microprocesoarelor a permis construirea unor computere din ce în ce mai mici și mai performante. În prezent, avem acces la computere portabile care depășesc cu mult performanțele calculatoarelor de odinioară.

Memoria computerelor a cunoscut, de asemenea, o evoluție semnificativă (Fig. 1.1). De exemplu, în anul 1977 un megabyte de memorie costa 32.000 de dolari, iar în prezent putem achiziționa Gigabytes (GB) de memorie la prețuri modice. Această evoluție a permis dezvoltarea unor aplicații software din ce în ce mai complexe și mai performante.

| An | Capacitate | Cost (USD / MB) |
|---|---|---|
| 1977 | 16KB | 32.000$ |
| 1987 | 640KB – 2MB | 250$ |
| 1997 | 64MB – 256MB | 2$ |
| 2007 | 512MB – 2GB | 0.06$ |
| 2014 | 4GB –> … | 0.0091$ |
| 2024 | 16GB –> … | ~0.000… |

**Fig. 1.1.** Evoluția capacității și a costului memoriei RAM.

Stocarea datelor a devenit, de asemenea, mult mai accesibilă. Dacă în 1977 un hard disk de 40 MB costa 679 de dolari, în prezent putem achiziționa Terabytes (TB) de spațiu de stocare la prețuri accesibile (Fig. 1.2). Această evoluție a permis stocarea și procesarea unor cantități din ce în ce mai mari de date.

| An | Capacitate | Cost (USD /MB) |
|---|---|---|
| 1977 | 16KB | 32.000$ |
| 1987 | 640KB - 2MB | 250$ |
| 1997 | 64MB - 256MB | 2$ |
| 2007 | 512MB - 2GB | 0.06$ |
| 2014 | 4GB ->… | 0.0091$ |
| 2024 | 16GB ->… | ~0.000… |

**Fig. 1.2.** Evoluția capacității și costului stocării.

Rețelele de calculatoare au cunoscut, de asemenea, o dezvoltare spectaculoasă. În 1985 viteza de transfer a datelor era de 10 Mbps, în timp ce în prezent avem acces la rețele de fibră optică cu viteze de transfer de ordinul Gigabiților pe secundă (Gbps). Această evoluție a permis conectarea calculatoarelor la scară globală și dezvoltarea unor aplicații software distribuite. Câteva repere importante din evoluția ratelor de transfer sunt următoarele:

- 1985: thick Ethernet: 10 Mbps (1 Mbps cu twisted pair networking)
- 1991: 10BaseT - twisted pair: 10 Mbps
- 1995: 100 Mbps Ethernet
- 1998: 1 Gbps (Gigabit) Ethernet
- 2001: 10 Gbps
- 2005: 100 Gbps (fibră optică)

Fig. 1.3 prezintă evoluția paradigmelor de calcul din anii 1970 până în prezent, subliniind tranziția între diferite arhitecturi și modele tehnologice. Conform citatului „we can define computing to mean any goal-oriented activity requiring, benefiting from, or creating computers”[1] („putem defini computing-ul ca orice activitate orientată spre un scop care necesită, beneficiază de sau creează computere”), putem înțelege această evoluție ca un

[1] Sursă: Wikipedia

proces continuu de adaptare a tehnologiilor pentru a răspunde nevoilor specifice ale utilizatorilor și organizațiilor.

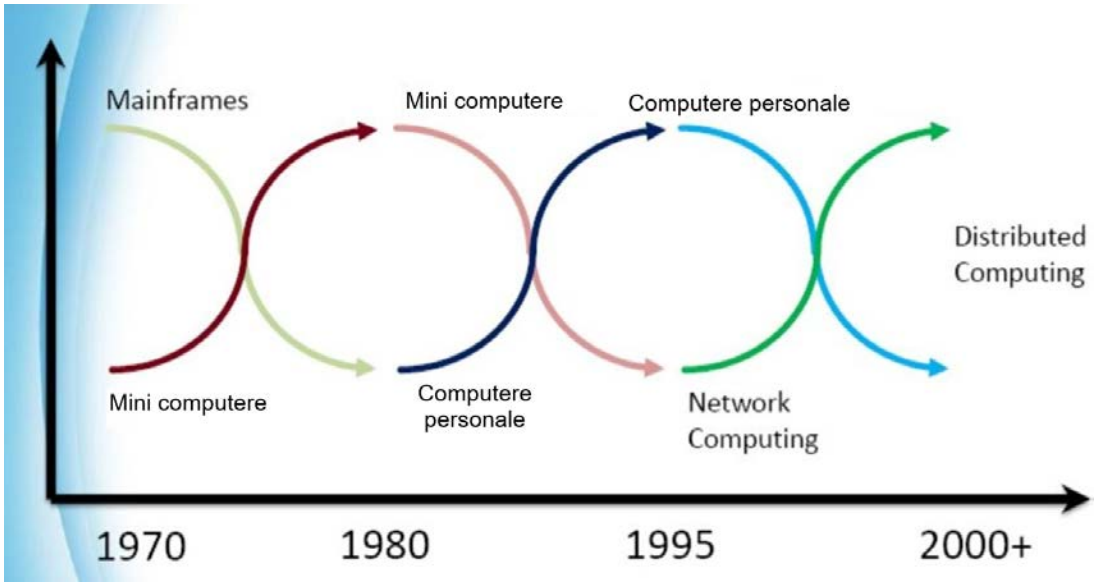



**Fig. 1.3.** Evoluția paradigmelor de calcul.

Evoluția computerelor are următoarele etape principale:

1. Era Mainframe (1970-1980) - dominată de sisteme mainframe, utilizate în medii centralizate, precum marile companii și instituții guvernamentale. Pe măsură ce tehnologia a avansat, a apărut tranziția către mini-computere, care ofereau o soluție mai accesibilă și flexibilă.

2. Era Mini-Computerelor (1980-1995) Mini-computerele au permis un model de calcul mai descentralizat, fiind utilizate în medii industriale și universitare. Creșterea puterii de procesare și scăderea costurilor au dus la adoptarea pe scară largă a calculatoarelor personale (PC-uri), care au început să înlocuiască mini-computerele.

3. Era Calculatoarelor Personale și a Calculului în Rețea (1995 - prezent) Odată cu dezvoltarea rețelelor, PC-urile au evoluat către un model conectat, permițând partajarea resurselor și colaborarea online. Network computing (calculul în rețea) a devenit un standard, facilitând schimbul de date și resurse între sisteme.

4. Era Calculului Distribuit (2000+) Modelul dominant actual este cel de distributed computing (calcul distribuit), în care resursele sunt împărțite între mai multe sisteme și servicii conectate prin internet. Aceasta include cloud computing, edge computing și arhitecturi descentralizate, ce permit scalabilitate și eficiență ridicată.

Această evoluție reflectă modul în care computerele au devenit din ce în ce mai accesibile, interconectate și integrate în viața cotidiană, permițând utilizatorilor să beneficieze de tehnologii avansate pentru diverse aplicații.

## 1.2. Aplicații software: proprietare vs. open-source

În prezent, există două modele principale de aplicații software: proprietare (closed-source) și open-source. Aceste modele definesc modul în care utilizatorii pot accesa, modifica și utiliza programele informatice.

Software-ul proprietar este dezvoltat de companii și distribuit sub licențe stricte, care limitează accesul utilizatorilor la codul sursă. O analogie relevantă este compararea acestui model cu un puzzle complex, unde utilizatorii au acces doar la piesele exterioare și la imaginea finală, dar nu pot modifica structura internă a aplicației. Acest lucru oferă o experiență de utilizare controlată și optimizată, însă limitează posibilitățile de personalizare. Câteva exemple de software proprietar sunt ilustrate în Fig. 1.4.: Suita Microsoft Office, Adobe Reader, Norton Antivirus, Microsoft Windows, MacOS și McAfee Antivirus. Printre avantajele software-ului proprietar se numără:

- Funcționalități personalizate și optimizate
- Suport tehnic dedicat și actualizări regulate
- Securitate îmbunătățită datorită controlului strict asupra codului sursă.


**Fig. 1.4.** Software proprietar.

Totuși, aceste avantaje vin și cu anumite dezavantaje, precum:

- Costuri ridicate pentru achiziție și mentenanță

- Dependența de furnizorul software-ului, ceea ce poate limita flexibilitatea utilizatorilor
- Lipsa accesului la codul sursă, ceea ce împiedică modificările sau adaptările independente.

Pe de altă parte, software-ul open-source oferă o paradigmă diferită, fiind dezvoltat de comunități globale de programatori și distribuit sub licențe libere. O analogie potrivită este aceea a unei grădini comunitare, unde fiecare utilizator poate contribui cu semințe, îngrijire și idei noi, îmbunătățind astfel ecosistemul software. Spre deosebire de software-ul proprietar, open-source permite accesul complet la codul sursă, ceea ce favorizează inovația și colaborarea. Printre avantajele acestui model se numără:

- Comunitate activă de dezvoltatori care contribuie constant la îmbunătățiri;
- Costuri reduse sau chiar inexistente pentru utilizare și distribuție;
- Flexibilitate ridicată, deoarece utilizatorii pot modifica și personaliza software-ul în funcție de nevoile lor specifice.

Câteva exemeple de software open-source utilizat frecvent sunt următoarele: Mozilla Firefox, distribuții Linux (ubuntu, debian, fedora), Java, Wordpress, PhP, mySQL (Fig. 1.5).


**Fig. 1.5.** Software open-source.

În ciuda beneficiilor, software-ul open-source prezintă și dezavantaje:

- Necesită, de multe ori, cunoștințe tehnice avansate pentru configurare și administrare
- Suportul tehnic poate fi limitat, fiind oferit de comunități sau contra cost de terți

- Există riscuri de securitate dacă software-ul nu este întreținut și actualizat corespunzător.

Astfel, alegerea între software-ul proprietar și open-source depinde de nevoile utilizatorilor. Dacă se dorește o soluție stabilă, cu suport tehnic garantat, software-ul proprietar poate fi alegerea potrivită. În schimb, pentru cei care doresc flexibilitate, costuri reduse și libertatea de a modifica software-ul, open-source reprezintă o alternativă viabilă. Ambele modele au roluri importante în ecosistemul software modern, iar decizia finală trebuie să fie bazată pe cerințele specifice ale fiecărui utilizator sau organizație.

## 1.3. Evoluția de la Grid Computing la Cloud Computing

Evoluția cronologică a tehnologiilor de computing este prezentată în Fig. 1.6. Computing-ul a evoluat semnificativ în ultimele decenii, trecând prin mai multe etape esențiale care au redefinit modul în care utilizăm resursele de calcul. Aceste transformări au condus la apariția Cloud Computing-ului, o tehnologie care permite accesul flexibil la resurse de calcul scalabile prin intermediul internetului. Această evoluție poate fi împărțită în patru etape majore: Grid Computing, Utility Computing, Software ca Serviciu (SaaS) și, în final, Cloud Computing.

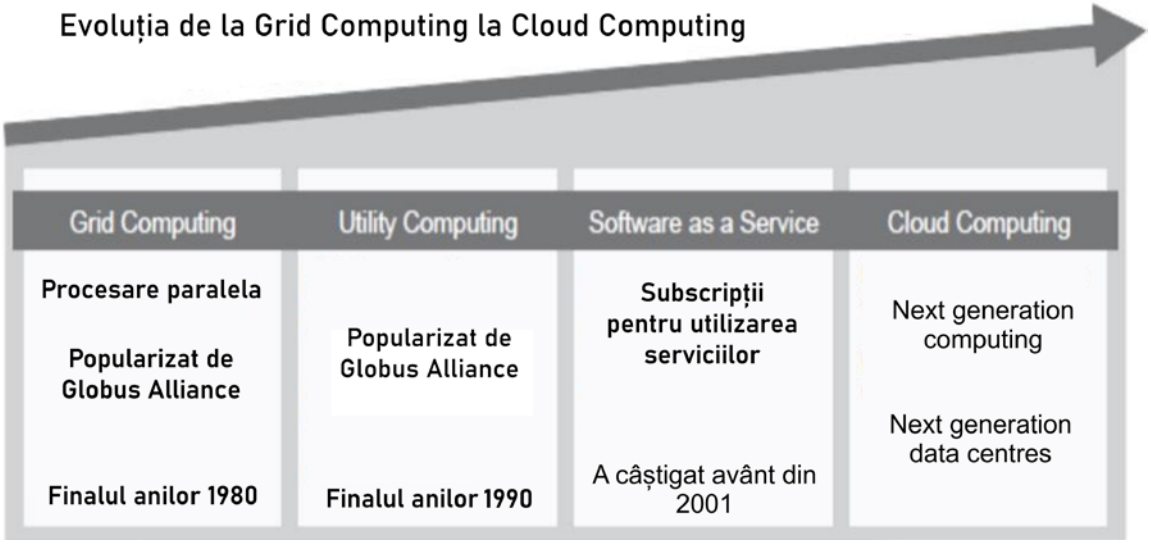



**Fig. 1.6.** Evoluția computing.

Figura evidențiază transformările fundamentale în modul în care resursele informatice sunt puse la dispoziție și utilizate:

- Grid Computing - reprezintă o arhitectură distribuită, orientată spre rezolvarea problemelor complexe prin utilizarea paralelă a resurselor de calcul distribuite geografic.

- Utility Computing introduce conceptul de a oferi resurse de calcul ca servicii contorizate, similar cu utilitățile tradiționale (electricitate, apă)
- Software as a Service (SaaS) - se concentrează pe furnizarea de aplicații software prin internet, pe bază de abonament.
- Cloud Computing 0 reprezintă o evoluție extinsă a conceptelor anterioare, oferind o gamă largă de servicii de calcul prin internet, inclusiv infrastructură, platforme și aplicații software.

### 1.3.1. Grid Computing - puterea calculului distribuit

Grid computing este un model de calcul distribuit care permite partajarea resurselor (computere, stocare, aplicații) între mai multe organizații sau utilizatori. Acesta este utilizat pentru a rezolva probleme complexe care necesită o putere mare de calcul sau acces la volume mari de date. Grid computing funcționează prin conectarea mai multor computere sau servere într-o rețea distribuită. Fiecare computer din rețea poate contribui cu resursele sale (putere de calcul, stocare etc.) la rezolvarea unei probleme comune. Un software specializat, numit middleware, este utilizat pentru a gestiona distribuirea sarcinilor și coordonarea resurselor în cadrul grid-ului.

Există mai multe tipuri de grid-uri, clasificate în funcție de scopul și arhitectura lor:

- **Grid-uri de date:** Sunt utilizate pentru a gestiona și partaja volume mari de date între mai multe organizații.
- **Grid-uri de calcul:** Sunt utilizate pentru a rezolva probleme complexe care necesită o putere mare de calcul.
- **Grid-uri de servicii:** Sunt utilizate pentru a oferi acces la servicii software și aplicații prin intermediul unui grid

Grid computing este utilizat în diverse domenii, cum ar fi cercetarea științifică, ingineria, finanțele și telecomunicațiile. De exemplu, în telecomunicații, grid computing poate fi utilizat pentru a gestiona rețelele de comunicații, pentru a procesa datele de trafic și pentru a dezvolta aplicații inovatoare.

Grid computing prezintă mai multe avantaje, printre care:

- Putere de calcul sporită: Permite utilizarea colectivă a resurselor pentru a rezolva probleme complexe.
- Acces la date și resurse distribuite: Facilitează colaborarea și partajarea de informații între organizații.

- Flexibilitate și scalabilitate: Permite adaptarea resurselor la nevoile specifice ale fiecărei probleme.

Dezavantajele acestui model sunt următoarele:

- Complexitate: Configurarea și gestionarea unui grid poate fi complexă.
- Securitate: Partajarea resurselor între mai multe organizații poate ridica probleme de securitate.
- Costuri: Implementarea și întreținerea unui grid poate fi costisitoare.

Astfel, grid computing este o tehnologie puternică care permite rezolvarea unor probleme complexe prin utilizarea colectivă a resurselor distribuite. Deși prezintă anumite provocări, avantajele sale îl fac o opțiune atractivă pentru o varietate de domenii

### 1.3.2. Utility computing – resurse de calcul ca servicii

Utility computing este un model de furnizare de servicii în care un furnizor oferă resurse de calcul și infrastructură la cerere, cu plată în funcție de utilizare. Acesta permite utilizatorilor să acceseze resursele de care au nevoie, fără a fi nevoie să investească în infrastructură proprie.

Utility Computing reprezintă un model de servicii în care furnizorii pun la dispoziție resurse de calcul, stocare și alte servicii IT la cerere, iar clienții plătesc doar pentru ceea ce utilizează. Acest model este similar cu serviciile de utilități tradiționale, cum ar fi electricitatea sau apa, unde consumatorii plătesc în funcție de consum.

Principiile de bază ale acestui model sunt următoarele:

- Plata la utilizare: Clienții plătesc doar pentru resursele pe care le consumă, similar cu facturile de utilități.
- Scalabilitate: Resursele pot fi scalate în sus sau în jos în funcție de nevoile clientului.
- Flexibilitate: Clienții pot accesa o gamă largă de servicii IT, de la infrastructură de bază la aplicații software.
- Automatizare: Furnizarea și gestionarea resurselor sunt automatizate, reducând intervenția umană.

Printre avantajele ale utility computing se regăsesc:

- Reducerea costurilor: Nu sunt necesare investițiile inițiale mari în infrastructura IT.
- Flexibilitate sporită: Utilizatorii pot adapta rapid resursele la schimbările de nevoi.
- Eficiență îmbunătățită: Automatizarea și scalabilitatea duc la o utilizare mai eficientă a resurselor.

Utility Computing este considerat un precursor al Cloud Computing-ului. Ambele modele se bazează pe furnizarea de servicii IT la cerere, dar Cloud Computing-ul oferă o gamă mai largă de servicii și o mai mare flexibilitate.

În concluzie, Utility Computing reprezintă un model de servicii eficient și flexibil, care permite clienților să acceseze resurse IT la cerere și să plătească doar pentru ceea ce utilizează.

### 1.3.3. Software as a service (Saas) - aplicații accesibile online

Software as a Service (SaaS) reprezintă un model de distribuție software în care aplicațiile sunt găzduite de un furnizor de servicii și puse la dispoziția clienților prin internet. În loc să achiziționeze și să instaleze software-ul pe propriile lor dispozitive, utilizatorii accesează aplicațiile SaaS printr-un browser web sau o aplicație mobilă, plătind de obicei un abonament lunar sau anual. Acest model a devenit popular datorită avantajelor sale legate de accesibilitate, costuri reduse și actualizări automate.

Caracteristici cheie ale acestui model sunt:

- Acces prin internet: Aplicațiile SaaS sunt accesibile de oriunde, oricând, printr-o conexiune la internet.
- Gestionare centralizată: Furnizorul de servicii SaaS este responsabil pentru gestionarea, întreținerea și actualizarea software-ului.
- Model de abonament: Utilizatorii plătesc de obicei un abonament pentru a accesa aplicația, eliminând costurile inițiale mari de achiziție.
- Scalabilitate: Resursele pot fi scalate în funcție de nevoile utilizatorilor, permițând adaptarea rapidă la schimbările de cerere.
- Personalizare: Multe aplicații SaaS oferă opțiuni de personalizare pentru a se potrivi nevoilor specifice ale utilizatorilor.

SaaS a transformat modul în care diverse categorii de utilizatori folosesc software-ul, oferind o alternativă mai flexibilă, accesibilă și rentabilă la modelele tradiționale de distribuție software. Subiectul este abordat mai pe larg în Capitolul 10

### 1.3.4. Cloud Computing – viitorul tehnologiilor IT

Cloud Computing reprezintă evoluția finală a acestor modele, combinând avantajele Utility Computing și SaaS pentru a oferi infrastructuri IT flexibile, scalabile și accesibile prin internet. Cloud Computing a revoluționat industria IT, oferind companiilor posibilitatea de a utiliza resurse de calcul fără a investi în infrastructuri costisitoare.. Acesta permite accesul la o gamă

largă de servicii, de la infrastructură (servere, stocare, rețea) și platforme de dezvoltare, până la aplicații software complete, fără a fi necesară gestionarea directă a infrastructurii fizice.

Caracteristici cheie:

- Acces la cerere: Resursele sunt disponibile oricând și de oriunde, printr-o conexiune la internet.
- Plată în funcție de utilizare: Utilizatorii plătesc doar pentru resursele pe care le consumă, eliminând costurile inițiale mari.
- Scalabilitate și elasticitate: Resursele pot fi scalate rapid în sus sau în jos, în funcție de nevoile utilizatorilor.
- Partajarea resurselor: Mai mulți utilizatori partajează aceeași infrastructură fizică, optimizând utilizarea resurselor.
- Automatizare: Furnizarea și gestionarea resurselor sunt automatizate, reducând intervenția umană.

Cloud Computing nu este doar o tendință temporară, ci o infrastructură esențială pentru viitorul digital, alimentând aplicațiile moderne și deschizând noi posibilități în domenii precum inteligența artificială, analiza datelor, securitatea cibernetică și internetul lucrurilor (IoT). Aspectele Cloud Computing sunt prezentate în detaliu în Capitolul 10.

# 2. Capitolul 2. Modelul de comunicație client-server

Modelul de comunicație client-server constituie o arhitectură fundamentală în domeniul rețelelor de calculatoare și al aplicațiilor distribuite. Acest model descrie un sistem în care funcțiile și sarcinile sunt împărțite între două entități: **serverele**, care furnizează resurse, servicii sau date, și **clienții**, care inițiază cereri pentru a accesa aceste resurse. Arhitectura client-server a devenit un standard în proiectarea și implementarea soluțiilor software, datorită capacității sale de a permite scalabilitate, securitate și gestionare eficientă a resurselor într-un mediu distribuit.

Principiul de bază al modelului client-server constă în separarea clară a rolurilor între client și server. Serverul este responsabil pentru procesarea cererilor și furnizarea răspunsurilor adecvate, în timp ce clientul inițiază cereri și consumă serviciile oferite. Această interacțiune este susținută prin intermediul unor protocoale bine definite, care asigură o comunicare eficientă și fiabilă între componente.

În cadrul acestui capitol, vom analiza în detaliu arhitectura client-server și rolul suitei de protocoale IP în stabilirea unei conexiuni robuste între clienți și servere. De asemenea, va fi prezentat și protocolul HTTP, un standard esențial pentru transferul de date pe web, și mecanismul request-response, care stă la baza interacțiunilor dintre clienți și servere. Prin aprofundarea acestor concepte, vom oferi o înțelegere clară a modului în care modelul client-server este utilizat în aplicațiile moderne, evidențiind importanța sa în contextul rețelelor de calculatoare și al serviciilor web.

## 2.1. Arhitectura client-server

Arhitectura client-server reprezintă un model de aplicație distribuită în care sarcinile și resursele sunt împărțite între furnizorii de servicii, denumiți servere, și solicitanții acestor servicii, denumiți clienți. Acest model permite utilizarea eficientă a resurselor, centralizarea datelor și scalabilitatea sistemelor, facilitând gestionarea și extinderea acestora.

**Componentele principale ale acestui model sunt:**

1. **Serverul**: Serverul este componenta responsabilă pentru furnizarea de servicii și resurse către clienți. Exemplele includ servere web, servere de baze de date, servere de fișiere și servere de aplicații. Acestea sunt concepute pentru a putea răspunde simultan unui număr mare de cereri, asigurând disponibilitatea, securitatea și integritatea datelor.

2. **Clientul**: Clientul este componenta care inițiază cereri către server pentru a accesa resursele și serviciile oferite. Clienții pot lua diverse forme, de la browsere web și aplicații mobile până la sisteme software specializate utilizate în medii industriale sau comerciale..
3. **Rețeaua**: Rețeaua este mediul de comunicație prin care clienții și serverele interacționează. Aceasta poate fi o rețea locală (LAN), o rețea extinsă (WAN) sau internetul. Comunicarea dintre client și server este realizată prin intermediul protocoalelor de rețea, care asigură transmiterea fiabilă și securizată a datelor.

Figura 2.1 ilustrează arhitectura client-server, evidențiind modul în care clienții comunică cu un server prin intermediul internetului. În exemplul prezentat, mai multe dispozitive client (precum un laptop, un smartphone și un computer desktop) sunt conectate la un server printr-o rețea. Serverul gestionează și procesează cererile venite de la clienți, oferind acces la resursele sau serviciile solicitate. Această diagramă subliniază rolul central al serverului în distribuirea și gestionarea informațiilor într-un sistem client-server modern.

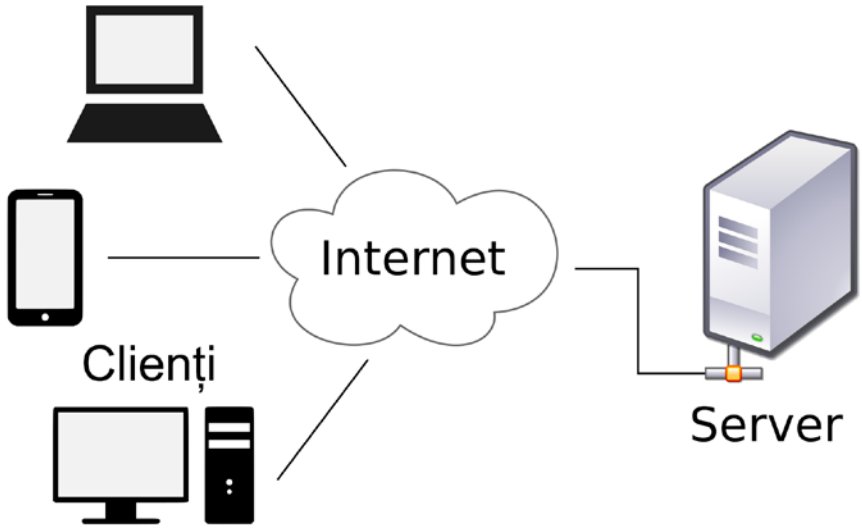



**Fig. 2.1.** Modelul Client – Server.

## 2.2. Componenta server

În cadrul arhitecturii client-server, există mai multe tipuri de servere, fiecare având un rol specific în furnizarea de servicii și resurse utilizatorilor. Printre cele mai comune tipuri de servere se numără:

- Serverele web – oferă pagini web și conținut multimedia către clienți prin intermediul protocolului HTTP/HTTPS.
- Serverele de fișiere – gestionează și distribuie fișiere într-o rețea, permițând utilizatorilor să acceseze și să partajeze documente.

- Serverele DNS – traduc numele de domenii în adrese IP, facilitând localizarea și accesarea resurselor pe internet.

## Server web

Serverele web reprezintă o componentă esențială a internetului modern, având rolul de a gestiona și livra conținut web către utilizatori. Aceste servere sunt un ansamblu de hardware și software care acceptă cereri HTTP/HTTPS și răspund prin furnizarea resurselor solicitate.

Un server web primește cereri de la clienți (de obicei, browsere web) și răspunde cu o resursă, cum ar fi o pagină HTML, un fișier multimedia sau un document PDF. În cazul în care resursa nu este disponibilă, serverul returnează un mesaj de eroare corespunzător (de exemplu, eroarea 404 - Resursa negăsită).

Resursele furnizate de un server web pot fi:

- Fișiere statice – fișiere preexistente pe server, cum ar fi pagini HTML, imagini, fișiere CSS și JavaScript.
- Conținut generat dinamic – pagini create la cerere de către server, utilizând tehnologii precum Java, PHP, Python sau Node.js.

Serverele web utilizează în principal protocolul HTTP (Hypertext Transfer Protocol) sau versiunea sa securizată HTTPS pentru a transmite date între server și client. Aceste protocoale definesc modul în care cererile și răspunsurile sunt structurate și transferate.

În plus, tehnologii precum REST (Representational State Transfer) și SOAP (Simple Object Access Protocol) utilizează HTTP pentru a permite interacțiunea între aplicații software distribuite, facilitând schimbul de date între diferite sisteme.

Fig. 2.2 ilustrează modelul client-server în contextul unui server web. Mai mulți clienți (Client 1, Client 2, ..., Client n) se conectează prin rețea la un proces, care face parte dintr-un server web. Acest server procesează cererile clienților și accesează sistemul de fișiere pentru a furniza resursele solicitate, precum pagini web sau fișiere.

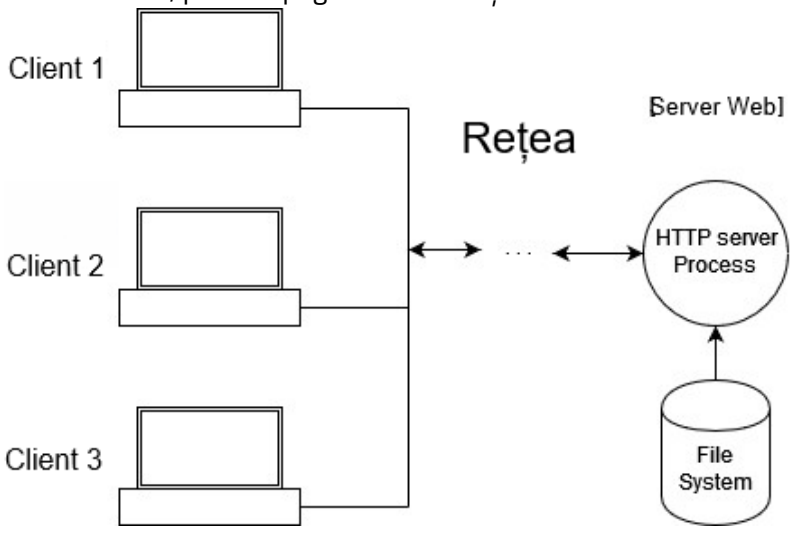



**Fig. 2.2.** Server Web.

## Server de fișiere

Un server de fișiere este, de regulă, un computer conectat la o rețea, specializat în stocarea și partajarea fișierelor între utilizatori și dispozitive. Spre deosebire de alte tipuri de servere, un server de fișiere nu realizează operațiuni de procesare intensă sau execuția de programe, ci se concentrează exclusiv pe gestionarea și distribuirea fișierelor într-o rețea.

Serverele de fișiere pot fi clasificate în două categorii:

- Servere de fișiere dedicate – sunt utilizate exclusiv pentru stocarea și distribuirea fișierelor, fără a îndeplini alte funcții.
- Servere de fișiere nededicate – sunt sisteme care, pe lângă funcția de partajare a fișierelor, îndeplinesc și alte sarcini, cum ar fi rularea unor aplicații locale.

Aceste servere sunt accesate frecvent prin protocoale specializate de transfer de fișiere, cum ar fi File Transfer Protocol (FTP) sau prin intermediul HTTP, în cazul în care fișierele sunt accesate printr-un browser web. Utilizarea serverelor de fișiere facilitează colaborarea și partajarea eficientă a resurselor în organizații, oferind un mediu sigur și centralizat pentru stocarea datelor.

Prin adoptarea acestor soluții, organizațiile pot asigura acces controlat la informații, gestionarea eficientă a permisiunilor și creșterea productivității printr-un sistem bine organizat de distribuire a fișierelor.

Fig 2.3 prezintă modelul client-server în contextul unui server de fișiere. Mai multe stații de lucru (Computer 1, Computer 2 și Computer 3) sunt conectate printr-o rețea locală (LAN) la un server de fișiere (File-server). Stațiile de lucru trimit cereri pentru date către server, iar acesta le returnează fișierele solicitate. Serverul de fișiere poate accesa și o bază de date pentru a obține informațiile necesare. Acest model permite gestionarea centralizată a fișierelor și partajarea datelor între mai mulți utilizatori.

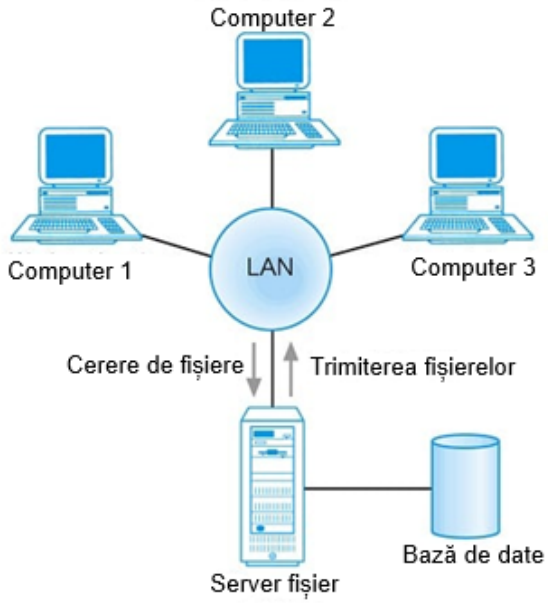



**Fig. 2.3.** Server Fișiere.

## Server DNS

Un server DNS (Domain Name System) funcționează similar unei agende telefonice pentru internet, facilitând conversia numelor de domenii ușor de reținut în adrese IP numerice utilizate de computere pentru a se identifica și comunica între ele. Atunci când un utilizator introduce un nume de domeniu, precum *www.unitbv.ro*, în browserul său, acesta trimite o solicitare către un server DNS pentru a obține adresa IP asociată, permițând astfel accesul la site-ul dorit.

Funcționarea unui server DNS implică mai mulți pași esențiali (Fig. 2.4):

1. Interogarea DNS: Browser-ul trimite o cerere către serverul DNS configurat pentru a afla adresa IP asociată numelui de domeniu introdus.
2. Rezoluția numelui: Serverul DNS caută în baza sa de date adresa IP corespunzătoare numelui de domeniu solicitat. Dacă nu o găsește, serverul poate transmite cererea către alte servere DNS până când informația este găsită.
3. Răspunsul DNS: Odată găsită adresa IP, serverul DNS o trimite înapoi către browser, care apoi se conectează la serverul web corespunzător pentru a accesa site-ul solicitat.

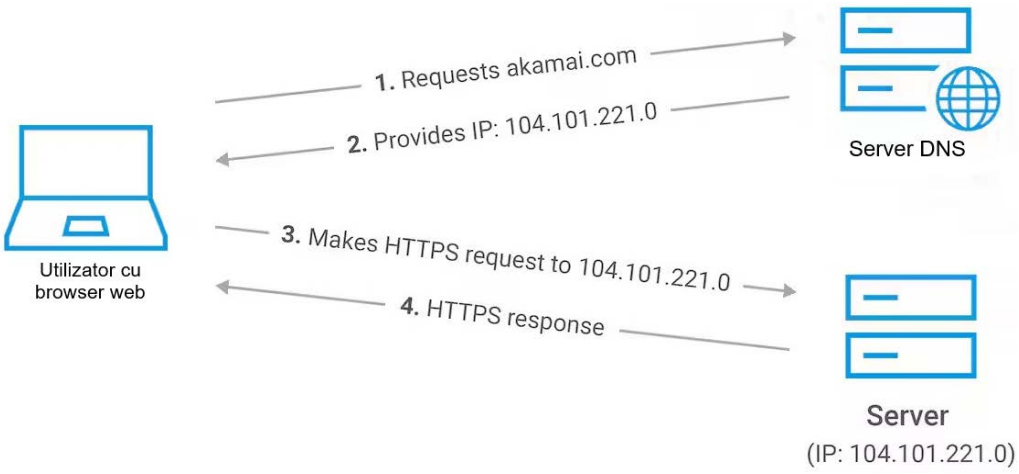



**Fig. 2.4.** Server DNS.

Tipuri de servere DNS:

- Servere DNS primare: Acestea dețin copia originală a datelor de zonă pentru un domeniu și răspund direct la interogările DNS referitoare la acel domeniu.
- Servere DNS secundare: Acestea stochează copii ale datelor de zonă de la serverele primare și servesc drept backup în cazul în care serverul primar devine indisponibil, asigurând astfel redundanța și fiabilitatea serviciului DNS.

Așadar, serverele DNS sunt esențiale pentru funcționarea eficientă a internetului, asigurând traducerea rapidă și precisă a numelor de domenii în adrese IP, ceea ce permite utilizatorilor să acceseze resursele online dorite fără a fi necesară memorarea adreselor numerice complexe.

## 2.3. Componenta client

În arhitectura client-server, clientul reprezintă componenta care inițiază cereri către un server pentru a accesa servicii sau resurse specifice. De obicei, un client este o aplicație software, precum un browser web, care rulează pe dispozitivul local al utilizatorului—fie un computer personal, smartphone sau alt dispozitiv conectat la rețea. Rolul principal al clientului este de a furniza o interfață prin care utilizatorul poate interacționa cu serviciile oferite de server, facilitând astfel o experiență de utilizare eficientă și intuitivă.

Operațiunile efectuate pe partea clientului sunt adesea necesare atunci când accesul la informații sau funcționalități specifice este disponibil doar pe client și nu pe server. Acest lucru se întâmplă deoarece utilizatorul trebuie să observe direct rezultatele operațiunilor sau să furnizeze date de intrare, sau deoarece serverul nu dispune de resursele de procesare necesare pentru a efectua rapid toate cererile primite de la toți clienții pe care îi deservește. În plus, dacă anumite operațiuni pot fi realizate de către client fără a trimite date prin rețea, acestea pot fi executate mai rapid, utilizând mai puțină lățime de bandă și reducând riscul de securitate asociat transmiterii datelor.

Un aspect important al arhitecturii client-server este faptul că serverul poate servi date într-un mod standardizat, conform unor protocoale precum HTTP sau FTP. Această standardizare permite utilizatorilor să aleagă dintre o varietate de programe client; de exemplu, majoritatea browserelor web moderne pot solicita și primi date folosind atât HTTP, cât și FTP. Astfel, flexibilitatea în alegerea aplicațiilor client contribuie la o interoperabilitate crescută și la o experiență de utilizare îmbunătățită.

Este esențial de menționat că programele care rulează pe computerul local al unui utilizator fără a trimite sau a primi date printr-o rețea nu sunt considerate clienți în contextul arhitecturii client-server. Prin urmare, operațiunile unor astfel de programe nu ar fi denumite operațiuni pe partea clientului. În schimb, acestea funcționează în mod independent de rețea și nu se încadrează în modelul de comunicație client-server.

Astfel, componenta client joacă un rol esențial în arhitectura client-server, asigurând interfața prin care utilizatorii interacționează cu resursele și serviciile furnizate de servere. Prin gestionarea eficientă a operațiunilor pe partea clientului, se îmbunătățește performanța generală a sistemului și se asigură o experiență de utilizare optimă.

### Comunicarea Client-Server

Comunicarea dintre un client și un server se bazează pe un model cerere-răspuns, unde clientul inițiază o solicitare, iar serverul procesează această cerere și returnează un răspuns adecvat. În acest context, un serviciu reprezintă o abstractizare a resurselor sistemului, ceea ce înseamnă că un client nu trebuie să fie preocupat de modul în care serverul execută cererea, ci doar de răspunsul primit conform protocolului de aplicație utilizat.

Pentru ca această comunicare să fie eficientă, clientul și serverul trebuie să utilizeze un limbaj comun definit printr-un protocol de comunicare. Protocoalele, precum HTTP, FTP sau SMTP, funcționează la nivelul aplicației și stabilesc regulile de schimb de date între părți. În multe cazuri, serverele implementează interfețe de programare a aplicațiilor (API) care oferă o metodă standardizată de acces la servicii și facilitează interoperabilitatea între platforme.

Un server poate gestiona simultan cereri provenite de la mai mulți clienți. Deoarece resursele de procesare sunt limitate, serverul utilizează un sistem de programare pentru a prioritiza și gestiona cererile. În plus, pentru a preveni supraîncărcarea serverului și posibilele atacuri de tip refuzare a serviciului (DoS), pot fi implementate mecanisme de limitare a accesului și criptare a datelor transmise, mai ales atunci când sunt implicate informații sensibile.

Un exemplu concret de comunicare client-server este accesarea unui serviciu bancar online. Atunci când un utilizator introduce datele de autentificare într-un browser web (clientul), acesta trimite o cerere către serverul web al băncii. Serverul web poate accesa un server de baze de date pentru a verifica datele de autentificare ale utilizatorului, iar un server de aplicații poate procesa aceste date aplicând logica de afaceri a băncii. În cele din urmă, serverul web returnează răspunsul browserului client, care îl afișează utilizatorului. Acest exemplu demonstrează principiul separării funcționalităților, unde fiecare componentă îndeplinește o funcție specifică în procesul general.

Interacțiunea Client-Server presupune următorii pași (Fig. 2.5):

1. **Utilizatorul introduce adresa URL** – Atunci când un utilizator dorește să acceseze un website, introduce adresa URL în bara de adrese a browserului.
2. **Browserul solicită adresa DNS** – Browserul trimite o cerere către un server DNS pentru a obține adresa IP corespunzătoare domeniului specificat în URL.
3. **Serverul DNS returnează adresa IP a serverului web** – Sistemul DNS efectuează o căutare și returnează browserului adresa IP a serverului web care găzduiește site-ul solicitat.
4. **Browserul trimite o solicitare HTTP către serverul web** – Odată ce browserul cunoaște adresa IP a serverului, inițiază o cerere HTTP către acesta pentru a solicita resursele necesare (fișiere HTML, CSS, JavaScript etc.).
5. **Serverul trimite fișierele și browserul le redă** – Serverul web răspunde prin trimiterea fișierelor necesare, iar browserul le interpretează și redă pagina pentru utilizator.

Acest proces are loc într-un interval foarte scurt de timp și ilustrează fluxul tipic de interacțiune între un client și un server într-o rețea de calculatoare.

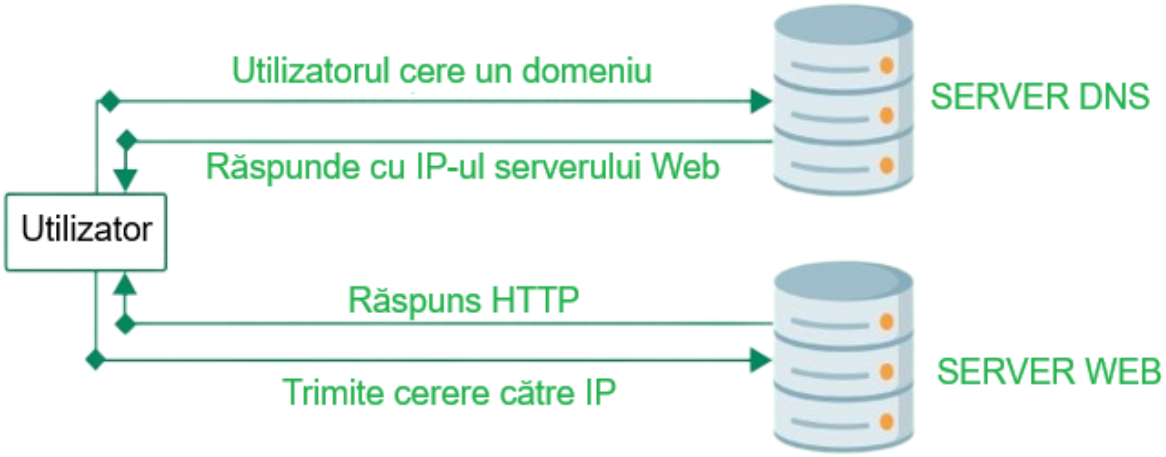



**Fig. 2.5.** Interacțiunea client-server.

## 2.4. IP Suite

Suita de protocoale Internet (IP Suite) reprezintă un set de reguli și standarde care definesc modul în care datele sunt transmise printr-o rețea de calculatoare. Aceasta oferă comunicații de date end-to-end, stabilind mecanismele prin care informațiile sunt pachetate, adresate, transmise, direcționate și recepționate. IP Suite este fundamentul tehnologiilor moderne de rețea și servește drept bază pentru Internet și rețelele private bazate pe TCP/IP.

Această suită este organizată în patru straturi de abstractizare. Fiecare strat este responsabil pentru anumite funcții specifice, iar interacțiunea dintre ele asigură un flux eficient de date între dispozitivele conectate.

- **Stratul de legătură** (*Link Layer*) – Reprezintă nivelul inferior al modelului și se ocupă cu metodele fizice de comunicare. Acest strat definește modul în care datele sunt transmise pe suportul fizic, inclusiv prin tehnologii precum Ethernet, Wi-Fi sau conexiuni celulare.
- **Stratul de rețea** (*Internet Layer*) – Permite interconectarea între rețele diferite și definește mecanismele necesare pentru rutarea pachetelor de date. Protocolul Internet (IP) este principalul protocol al acestui strat, responsabil pentru identificarea dispozitivelor și direcționarea pachetelor către destinația corectă.
- **Stratul de transport** (*Transport Layer*) – Gestionează comunicarea host-to-host, asigurând un transfer fiabil al datelor între aplicații. Protocoalele principale ale acestui strat sunt TCP (Transmission Control Protocol), care oferă transmisie securizată cu verificare a erorilor, și UDP (User Datagram Protocol), care permite un transfer rapid, dar fără garanția livrării corecte a fiecărui pachet.

- **Stratul de aplicație** (Application Layer) – Este stratul superior și facilitează schimbul de date între aplicații. Protocoalele acestui strat includ HTTP (pentru navigarea web), FTP (pentru transferul de fișiere), SMTP (pentru trimiterea e-mailurilor) și multe altele. Acest strat permite utilizatorilor să acceseze și să utilizeze serviciile rețelei.

Prin organizarea sa stratificată, IP Suite asigură o funcționare robustă și scalabilă a rețelelor, permițând comunicații eficiente între dispozitive diverse, indiferent de locația lor fizică sau tehnologia de rețea utilizată.

În cadrul rețelelor de calculatoare, identificarea dispozitivelor și direcționarea corectă a datelor se realizează prin intermediul a două tipuri de adrese fundamentale: adresa MAC și adresa IP. Acestea joacă roluri complementare în procesul de comunicare, asigurând atât identificarea unică a dispozitivelor, cât și rutarea eficientă a datelor în rețea.

Adresa MAC este un identificator unic atribuit conexiunilor de rețea și este încorporată direct în placa de rețea (*Network Interface Card* – NIC) în momentul fabricației. Aceasta este o adresă hardware, ceea ce înseamnă că este fixă și nu se modifică în mod normal în timpul utilizării dispozitivului. Adresa MAC este utilizată la nivelul stratului de legătură (Link Layer) pentru a permite comunicarea între dispozitive aflate în aceeași rețea fizică.

Aceasta are o lungime de 48 de biți și este exprimată sub formă hexadecimală, fiind formată din șase grupuri de două cifre (exemplu: 00:1A:2B:3C:4D:5E). Primele trei grupuri identifică producătorul dispozitivului, iar ultimele trei sunt unice pentru fiecare placă de rețea produsă de acel producător.

Unul dintre rolurile principale ale adresei MAC este de a direcționa mesajele primite către interfața de rețea corectă, asigurând astfel funcționarea eficientă a comunicării la nivel local.

Adresa IP este o etichetă numerică atribuită unui dispozitiv conectat la o rețea, permițându-i să comunice cu alte dispozitive prin intermediul Internetului sau al unei rețele private. Spre deosebire de adresa MAC, care este fixă, adresa IP poate fi schimbată în funcție de rețeaua la care se conectează dispozitivul.

Există două versiuni principale de adrese IP:

- IPv4, care utilizează o adresă de 32 de biți, exprimată sub formă zecimală (exemplu: 192.168.1.1).
- IPv6, care folosește o adresă de 128 de biți, exprimată în format hexadecimal (exemplu: 2001:0db8:85a3:0000:0000:8a2e:0370:7334).

Atunci când un dispozitiv se conectează la o rețea diferită, adresa sa IP poate fi modificată, fie manual, fie automat prin intermediul unui server DHCP (*Dynamic Host Configuration Protocol).*

Adresa MAC și adresa IP funcționează împreună pentru a asigura transmiterea corectă a pachetelor de date. Într-o rețea locală, dispozitivele utilizează adrese MAC pentru a comunica direct, în timp ce adresele IP sunt folosite pentru a permite rutarea datelor între rețele diferite.

Un aspect important este că, atunci când un mesaj este trimis printr-o rețea, adresa IP indică destinația finală, iar adresa MAC identifică dispozitivul specific care va primi mesajul în rețeaua locală. Această asociere între adresa MAC și adresa IP este gestionată de protocolul ARP (*Address Resolution Protocol*), care permite conversia unei adrese IP într-o adresă MAC corespunzătoare.

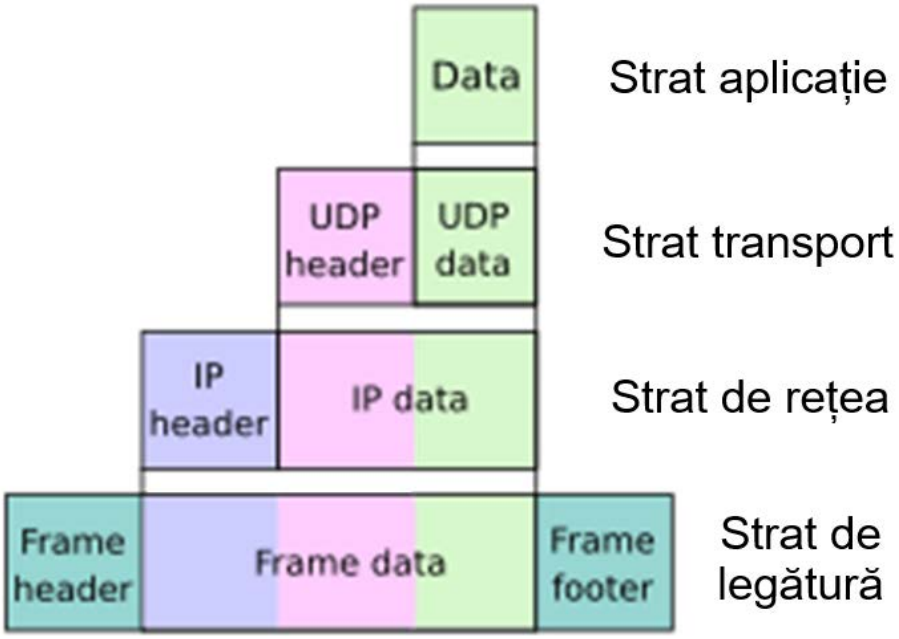



**Fig. 2.6.** Suita IP.

## 2.5. TCP/IP

Protocolul TCP/IP (Transmission Control Protocol/Internet Protocol) este cel mai utilizat protocol de comunicație în rețelele de calculatoare, fiind fundamentul Internetului și al rețelelor private. Acesta definește modul în care datele sunt transmise, împachetate, adresate și direcționate între dispozitive conectate la rețea.

TCP/IP este o suită de protocoale organizată în patru straturi funcționale, fiecare având un rol specific în procesul de transmitere a datelor. Aceste straturi lucrează împreună pentru a asigura comunicarea eficientă și corectă între dispozitivele dintr-o rețea.

Modelul TCP/IP are următoarele straturi:

- Stratul de aplicație – codifică datele transmise
- Stratul de transport - împarte datele în bucăți și adaugă informații despre numărul de port
- Stratul de rețea - adaugă adrese IP care indică de unde provin datele și unde se duc
- Stratul de legătură - adaugă informații despre adresa MAC.

### 2.5.1. Stratul de aplicație

Stratul de aplicație este responsabil pentru asigurarea comunicării dintre utilizator și rețea, garantând că datele sunt transmise într-un format corect și inteligibil de către destinatar. Acest strat definește modul în care aplicațiile utilizează rețeaua și oferă servicii specifice, precum transferul de fișiere, trimiterea de e-mailuri sau accesarea paginilor web.

Datele transmise prin acest strat sunt structurate într-un mod standardizat, specific aplicației utilizate. De exemplu, în cazul navigării pe internet, protocolul HTTP (HyperText Transfer Protocol) este utilizat pentru schimbul de pagini web, în timp ce FTP (File Transfer Protocol) facilitează transferul de fișiere între dispozitive.

În plus, pentru a permite interoperabilitatea între sisteme diferite, datele pot fi împachetate în formate structurate precum XML (Extensible Markup Language) sau JSON (JavaScript Object Notation). Aceste formate sunt utilizate pe scară largă în aplicații web și servicii bazate pe API-uri, asigurând schimbul de informații într-un mod eficient și ușor de procesat de către aplicații diverse.

De exemplu, considerând următoarele nume John, Mark, Mary, Mia, vom împacheta fiecare parte de date în format XML, astfel:

```
<name>John</name>
<name>Mark</name>
<name>Mary</name>
<name>Mia</name>
```

### 2.5.2. Stratul Transport

Stratul de transport joacă un rol esențial în asigurarea unei comunicări fiabile între dispozitivele conectate la rețea. Acesta are responsabilitatea de a analiza datele care trebuie transmise și de a le împărți în fragmente mai mici, denumite segmente sau pachete de date, astfel încât să poată fi gestionate eficient de straturile inferioare.

Pentru a menține ordinea și integritatea datelor, fiecărui pachet i se atașează un număr secvențial, ceea ce permite receptorului să reconstituie corect informația primită. De asemenea, stratul de transport include informații despre numărul de port, specificând destinația exactă a datelor în cadrul unui dispozitiv. De exemplu, dacă o aplicație rulează pe portul 90, datele transmise vor fi direcționate către acest port pentru a fi prelucrate corespunzător.

Acest strat utilizează protocoale precum TCP (Transmission Control Protocol), care asigură livrarea fiabilă a pachetelor și verificarea integrității datelor, sau UDP (User Datagram Protocol), care oferă o transmisie mai rapidă, dar fără garanția recepției fiecărui pachet.

Pentru exemplu, vom trimite date la portul 90. Antetul de transport și datele pentru fiecare pachet sunt prezentate Tabelul 2.1.

**Tabel 2.1.** Datele transmise de stratul de transport.

| Antet de transport | Date |
|---|---|
| :90 \| 1/4 | <name>John</name> |
| :90 \| 2/4 | <name>Mark</name> |
| :90 \| 3/4 | <name>Mary</name> |
| :90 \| 4/4 | <name>Mia</name> |

### 2.5.3. Stratul de rețea

Stratul de rețea are rolul de a asigura rutarea corectă a pachetelor de date între dispozitivele conectate la rețea. Acesta atașează adresa IP a expeditorului, astfel încât dispozitivul receptor să poată identifica sursa mesajului.

Pentru ca datele să ajungă la destinația corectă, stratul Internet trebuie să includă și adresa IP a gazdei către care sunt trimise datele. Fără această informație, pachetele ar putea fi pierdute sau trimise către un destinatar greșit.

O componentă importantă a acestui strat este crearea unui socket, care este format din combinația dintre adresa IP și numărul portului. Acest mecanism permite identificarea exactă a punctului de origine și a destinației fiecărui pachet de date, facilitând astfel comunicarea între dispozitive și aplicații de rețea.

În exemplul din Tabelul 2.2 trimitem date de la 98.1.232.99 către 102.231.4.189. Combinat cu numărul portului, aceasta creează un socket de la care sunt trimise datele și un socket către care sunt trimise datele, și anume 102.231.4.189:90.

**Tabel 2.2.** Datele transmise de stratul de rețea.

| Antet de rețea | Antet de transport | Date |
|---|---|---|
| 102.231.4.189<br>98.1.232.99 | :90 \| 1/4 | <name>John</name> |
| 102.231.4.189<br>98.1.232.99 | :90 \| 2/4 | <name>Mark</name> |
| 102.231.4.189<br>98.1.232.99 | :90 \| 3/4 | <name>Mary</name> |
| 102.231.4.189 | :90 \| 4/4 | <name>Mia</name> |

| | | |
|---|---|---|
| 98.1.232.99 | | |

### 2.5.4. Stratul de legătură

Stratul de legătură reprezintă nivelul de bază al modelului TCP/IP și se ocupă cu transmiterea fizică a pachetelor de date între dispozitivele conectate la aceeași rețea locală. Acesta atașează atât adresa MAC a expeditorului, cât și adresa MAC a destinatarului, permițând astfel ca pachetele să fie direcționate corect către interfața de rețea corespunzătoare a mașinii gazdă.

Adresa MAC (Media Access Control) este un identificator unic atribuit fiecărei interfețe de rețea în timpul transmiterii. Aceasta permite ca dispozitivele dintr-o rețea locală să comunice între ele fără a fi nevoie de un protocol de rutare complex.

Tabelul 2.3 prezintă exemplul din Tabelul 2.2 la care s-a adăugat și antetul stratului de legătură:

- Adresa expeditorului: 00-17-4F-08-5D-69
- Adresa destinatarului: 11-22-33-44-55

Prin utilizarea acestor adrese, stratul de legătură asigură că datele sunt transmise corect la nivel local, înainte ca pachetele să fie procesate la nivelul superior pentru o eventuală trimitere prin internet.

**Tabel 2.3.** Datele transmise de stratul de legătură.

| Antet de legătură | Antet de rețea | Antet de transport | Date |
|---|---|---|---|
| 11-22-33-44-55<br>00-17-4F-08-5D-69 | 102.231.4.189<br>98.1.232.99 | :90 \| 1/4 | <name>John</name> |
| 11-22-33-44-55<br>00-17-4F-08-5D-69 | 102.231.4.189<br>98.1.232.99 | :90 \| 2/4 | <name>Mark</name> |
| 11-22-33-44-55<br>00-17-4F-08-5D-69 | 102.231.4.189<br>98.1.232.99 | :90 \| 3/4 | <name>Mary</name> |
| 11-22-33-44-55<br>00-17-4F-08-5D-69 | 102.231.4.189<br>98.1.232.99 | :90 \| 4/4 | <name>Mia</name> |

## 2.6. Protocolul HTTP

*Hypertext Transfer Protocol* (HTTP) este un protocol de nivel de aplicație din suita de protocoale Internet, utilizat pentru comunicarea între sisteme distribuite, colaborative și hipermedia. Acesta este fundamentul World Wide Web, facilitând transferul de pagini web, imagini, fișiere media și alte resurse între servere și clienți (navigatoare web).

Prin HTTP, documentele hipertext pot conține hyperlinkuri către alte resurse, permițând utilizatorilor să acceseze informații suplimentare printr-un simplu clic de mouse sau atingere a ecranului.

Dezvoltarea HTTP a fost inițiată de Tim Berners-Lee la CERN în 1989. Prima versiune a protocolului, HTTP/0.9, a fost extrem de simplă, permițând doar transferul unui fișier HTML de la server la client.

Ulterior, protocolul a fost îmbunătățit, ajungând la HTTP/1.0 în 1996 și la HTTP/1.1 în 1997, versiune care a devenit standardul dominant pentru mulți ani. Actualizări ale specificației HTTP/1.1 au fost publicate în 1999, 2014 și 2022.

Pentru a asigura o conexiune securizată, a fost introdus HTTPS (Hypertext Transfer Protocol Secure), care utilizează TLS (Transport Layer Security) pentru criptarea datelor. În prezent, peste 85% dintre site-urile web folosesc HTTPS pentru a proteja confidențialitatea și integritatea datelor utilizatorilor.

HTTP/2, introdus în 2015, a îmbunătățit performanța web prin multiplexare și compresia antetelor. HTTP/3, o versiune mai avansată, utilizează protocolul QUIC pentru reducerea latenței și accelerarea încărcării paginilor web. Dacă este activat pe server, HTTP/3 poate fi de peste trei ori mai rapid decât HTTP/1.1 în anumite scenarii.

HTTP este protocolul responsabil cu transferul datelor între serverele web și diverse tipuri de clienți, cum ar fi browserele web. Acest protocol joacă un rol esențial în navigarea pe internet, asigurând un mecanism standardizat pentru transmiterea documentelor și altor resurse între servere și utilizatori.

Deși HTTP este un protocol la nivel înalt, acesta operează deasupra protocolului TCP/IP, ceea ce garantează că datele transmise nu vor fi deteriorate în timpul transmiterii prin rețea. Protocolul TCP/IP oferă un mecanism de control al fluxului și corectare a erorilor, iar HTTP profită de aceste caracteristici pentru a asigura o livrare fiabilă a datelor.

HTTP este, de asemenea, un protocol bazat pe text, lucrând cu date de tip ASCII (*American Standard Code for Information Interchange*). Acest lucru face ca mesajele trimise între client și server să fie ușor de citit și de înțeles de către oameni, dar și ușor de procesat de către computere.

La baza funcționării HTTP stau două concepte fundamentale: cererea (*request*) și răspunsul (*response*). Când un client dorește să acceseze o resursă disponibilă pe un server (de exemplu, o pagină web), acesta trimite o cerere HTTP către server. Serverul procesează cererea și returnează un răspuns, care conține informațiile solicitate sau un mesaj de eroare, în funcție de circumstanțe. Aceste interacțiuni bidirecționale formează fundamentul comunicării pe web (Fig. 2.7).

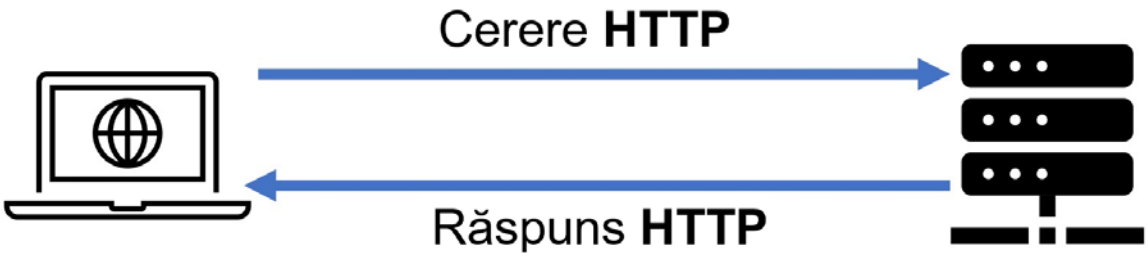



**Fig. 2.7.** Cerere/Răspuns HTTP.

Astfel, HTTP a evoluat semnificativ de la o simplă metodă de transfer a fișierelor HTML la un protocol modern, eficient și securizat, esențial pentru funcționarea internetului.

### 2.6.1. Schimbul de date prin HTTP

HTTP este un protocol *stateless*, ceea ce înseamnă că nu reține informații despre starea sesiunii sau despre cererile anterioare. Fiecare cerere și răspuns sunt tratate ca entități izolate, fără a depinde de interacțiunile precedente. Aceasta poate reprezenta atât un avantaj, cât și o limitare, deoarece serverele nu trebuie să păstreze informații despre utilizatori sau sesiuni, dar în același timp, această caracteristică necesită ca fiecare cerere să includă toate informațiile necesare pentru a putea fi procesată.

Deși HTTP este un protocol *stateless*, schimbul de date între client și server necesită o conexiune de transport fiabilă. De obicei, această conexiune este asigurată prin intermediul protocolului TCP/IP, care garantează livrarea corectă a datelor și ordinea corectă a acestora. În cazul HTTPS, se adaugă un strat de securitate prin criptarea comunicațiilor, utilizând portul 443.

Versiunea HTTP/1 utilizează conexiuni TCP/IP pe porturile standard 80 pentru HTTP (necriptat) și 443 pentru HTTPS (criptat). Aceste porturi sunt configurate pentru a asigura transferul de date între client și server printr-o conexiune stabilă și sigură.

Datele sunt schimbate în cadrul unei interacțiuni între client și server sub forma unui schimb de mesaje de tip cerere și răspuns. Cererea HTTP este trimisă de client către server, iar serverul răspunde cu un mesaj care conține două componente principale: antetul (*header*) și corpul (*body*).

Antetul (*header*) conține informații despre cererea sau răspunsul respectiv, precum tipul de conținut, tipurile de codificare utilizate, informații despre server și client, sau sesiunea curentă.

Corpul (*body*) reprezintă, de obicei, resursa solicitată de client, cum ar fi o pagină web, o imagine sau un fișier, sau poate conține un mesaj de eroare în cazul în care cererea nu a fost procesată cu succes.

În majoritatea cazurilor, corpul răspunsului conține exact resursa solicitată, iar antetul răspunsului conține informații suplimentare despre aceasta.

Deși HTTP este un protocol *stateless*, multe aplicații web trebuie să păstreze informații despre utilizatori între cereri pentru a oferi o experiență continuă și personalizată. În acest sens, gestionarea sesiunilor utilizatorilor devine esențială. Sesiunile HTTP sunt implementate adesea prin intermediul unor mecanisme cum ar fi *cookie*-urile HTTP sau variabilele ascunse din formularele web, care permit stocarea unor informații pe termen scurt.

*Cookie* este un bloc mic de date creat de server și salvat de browser pe dispozitivul utilizatorului. Aceste date pot include informații despre sesiunea utilizatorului, preferințele sale, sau un identificator unic care permite serverului să recunoască același utilizator la cereri succesive. *Cookie*-urile sunt trimise înapoi la server cu fiecare cerere ulterioară, permițând serverului să mențină starea sesiunii între cereri.

*Cookie*-urile pot fi configurate pentru a expira după un anumit interval de timp sau pot fi setate ca *cookie*-uri persistente, care sunt salvate pe termen lung, până la expirarea lor. Acestea joacă un rol important în autentificarea utilizatorilor și în stocarea informațiilor legate de sesiune.

Pentru a începe o sesiune, utilizatorul trebuie să se autentifice, adică să-și verifice identitatea prin furnizarea unor informații, cum ar fi numele de utilizator și parola. După autentificare, serverul poate crea un identificator unic de sesiune și îl poate trimite clientului sub forma unui *cookie*. Acest identificator va fi folosit pentru a recunoaște utilizatorul în cadrul sesiunii sale.

La sfârșitul unei sesiuni, utilizatorul poate solicita o operațiune de deconectare, moment în care sesiunea este încheiată și informațiile asociate sunt șterse, fie din partea serverului, fie din partea clientului. Acest proces de deconectare este esențial pentru a proteja confidențialitatea și securitatea utilizatorului, prevenind accesul neautorizat la conturile acestuia.

### 2.6.2. HTTP Request

Mesajele de solicitare HTTP sunt trimise de un client către un server țintă pentru a solicita resurse sau a efectua anumite acțiuni asupra acestora. Sintaxa unui mesaj de solicitare este bine definită și conține mai multe componente importante, fiecare având rolul său specific.

1. Linia de solicitare (*Request Line*)

Linia de solicitare este prima linie a mesajului și constă în trei elemente esențiale:

- Metoda de solicitare: Este verbul care descrie acțiunea ce va fi efectuată asupra entității de la server. Cele mai frecvent utilizate metode HTTP sunt:
    - GET: solicită resurse de la server fără a modifica serverul (de exemplu, pentru a vizualiza o pagină web)
    - POST: trimite date către server pentru a crea sau actualiza resurse (de exemplu, trimiterea unui formular)
    - PUT: trimite date către server pentru a înlocui complet o resursă existentă
    - DELETE: solicită serverului să elimine o resursă.
- URI-ul solicitat (*Uniform Resource Identifier*): Reprezintă calea și eventual parametrii necesari pentru a localiza resursa dorită pe server. De exemplu, în cazul unei căutări pe un site, URI-ul ar putea conține parametrii de căutare, care sunt separați de cale prin semnul „?" (de exemplu, *example.com/search?q=AI*).
- Versiunea protocolului HTTP: Indică versiunea protocolului utilizată. De obicei, este folosit HTTP/1.1, dar pentru securitate se poate utiliza HTTPS (HTTP securizat prin TLS/SSL), care funcționează pe portul 443 în loc de portul 80 folosit de HTTP.

```
GET /images/logo.png HTTP/1.1
```

2. Câmpuri de antet (Headers)

După linia de solicitare urmează câmpurile de antet. Acestea sunt folosite pentru a transmite informații suplimentare despre cerere, așa cum sunt descrise în următoarele trei categorii:

- *Request Headers*: Aceste antete descriu modul în care se face cererea și cum ar trebui să răspundă serverul. Un exemplu de antet de acest tip este *User-Agent*, care indică informații despre clientul care trimite cererea (de exemplu, browserul sau aplicația mobilă utilizată).
- *General Headers*: Aceste antete conțin informații generale despre cerere, cum ar fi durabilitatea unei sesiuni sau opțiuni de conexiune. Exemplu: *Keep-Alive*, care sugerează serverului să mențină conexiunea deschisă pentru cereri succesive.
- *Entity Headers*: Aceste antete sunt folosite pentru a descrie datele transmise cu cererea (dacă există). Exemple includ *Content-Type*, care indică tipul de conținut trimis (de exemplu, *application/json* sau *text/html*), și *Content-Length*, care specifică lungimea corpului mesajului.

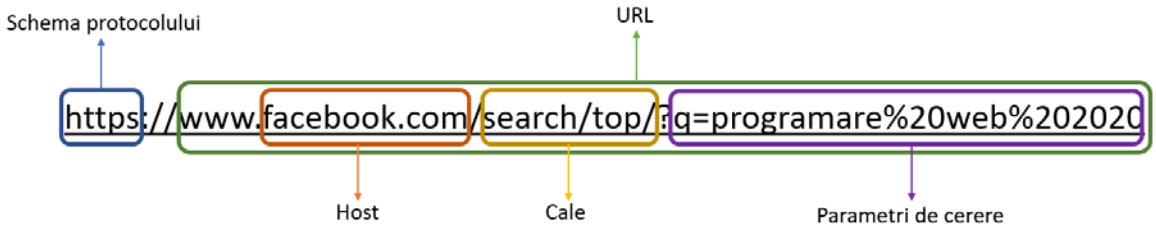
```
Host: www.example.com
Accept-language: en
```

3. Opțional: Corpul mesajului (*Message Body*)

În funcție de tipul cererii, mesajul HTTP poate conține un corp (body). De exemplu, în cazul unei cereri POST sau PUT, corpul poate conține datele trimise de client (cum ar fi informațiile

unui formular completat). În schimb, pentru cererile GET, corpul mesajului este de obicei absent, deoarece datele sunt obținute direct din URL sau din antetele cererii.

În cazul unei cereri efective HTTP, toate aceste elemente se combină pentru a forma mesajul complet. Când un client (precum un browser web) trimite o cerere HTTP, browserul construiește efectiv cererea în baza acestor informații și o trimite către server pentru procesare.




### 2.6.3. Metode HTTP

HTTP definește o serie de metode (uneori denumite „verbe", deși termenul „verb" nu este folosit oficial în specificație) care sunt utilizate pentru a indica acțiunea dorită asupra unei resurse identificate. Resursa poate fi orice entitate accesibilă prin HTTP, inclusiv fișiere statice sau date generate dinamic de server. Definiția exactă a resursei depinde de implementarea serverului, iar resursa poate corespunde unui fișier, unei pagini web sau rezultatului unui script sau executabil care rulează pe server.

De-a lungul evoluției protocolului HTTP, au fost definite mai multe metode pentru a permite interacțiunea între client și server. Specificația HTTP/1.0 a introdus metodele GET, HEAD, și POST, iar mai târziu, HTTP/1.1 a adăugat metode noi și a reglementat utilizarea acestora. Astfel, unele dintre cele mai importante metode HTTP includ:

1. **GET** - Metoda GET solicită ca serverul să transfere o reprezentare a resursei țintă către client. Acesta este folosit pentru a obține date de la server, fără a modifica resursa. GET ar trebui să fie utilizat pentru operațiuni care nu au efecte secundare și care doar returnează informații. De exemplu, atunci când un utilizator accesează o pagină web, browserul trimite o cerere GET pentru a solicita conținutul acelei pagini. Solicitările GET nu ar trebui să aibă efecte asupra stării serverului sau asupra datelor.
2. **POST** - Metoda POST solicită ca serverul să proceseze reprezentarea inclusă în cerere, conform semanticii resursei țintă. POST este utilizată pentru a trimite date către server, iar serverul poate să proceseze aceste date într-un mod specific. Este folosită frecvent pentru a trimite formulare pe internet, pentru a posta un mesaj pe un forum sau pentru a finaliza o tranzacție (de exemplu, un coș de cumpărături). Spre deosebire de GET, care este folosit pentru a obține informații, POST poate modifica starea serverului sau a bazei de date.
3. **PUT** - Metoda PUT solicită ca serverul să creeze sau să actualizeze resursa țintă conform cu datele incluse în cererea trimisă. Deosebirea principală față de POST este

că, în cazul metodei PUT, clientul specifică locația resursei pe server, ceea ce face ca PUT să fie folosit pentru actualizarea unui fișier sau resurse existente. Este important de menționat că PUT, spre deosebire de PATCH, înlocuiește complet resursa existentă la locația specificată de client.

4. **PATCH** - Metoda PATCH solicită serverului să aplice o modificare parțială asupra resursei țintă. În loc să trimită întreaga resursă pentru actualizare, PATCH permite clientului să transmită doar părțile modificare ale acesteia, economisind astfel lățimea de bandă și resursele de rețea. De exemplu, atunci când se actualizează doar un câmp dintr-o bază de date sau dintr-un document, cererea PATCH este utilizată pentru a modifica acea secțiune specifică a resursei, fără a fi necesar să se reîncarce întreaga resursă.
5. **DELETE** - Metoda DELETE solicită serverului să șteargă resursa specificată. Este folosită pentru a elimina un fișier, o pagină web sau orice alt tip de resursă de pe server. De exemplu, atunci când un utilizator șterge un fișier dintr-o aplicație web sau elimină un comentariu pe un forum, o cerere DELETE este trimisă pentru a elimina acea resursă.

În plus față de aceste metode comune, există și altele, precum **CONNECT**, **OPTIONS**, și **TRACE**, care au roluri specifice, cum ar fi gestionarea conexiunilor sau obținerea informațiilor despre opțiunile serverului.

De asemenea, nu există o limită strictă asupra numărului de metode care pot fi definite în HTTP. Astfel, organizațiile pot adăuga metode personalizate, iar protocoale suplimentare, cum ar fi WebDAV, au introdus metode suplimentare pentru a susține funcționalități extinse.

### 2.6.4. Răspuns HTTP

Un mesaj de răspuns reprezintă informațiile trimise de server unui client ca răspuns la cererea acestuia. După ce serverul a procesat solicitarea, acesta trimite înapoi un mesaj care conține mai multe componente esențiale pentru a informa clientul despre starea solicitării și despre rezultatul acesteia.

Un mesaj de răspuns HTTP constă în:

1. Linia de stare

Linia de stare este formată din trei elemente:

- Versiunea protocolului HTTP: Indică versiunea protocolului utilizată pentru a răspunde cererii. De exemplu, "HTTP/1.1".
- Codul de stare al răspunsului: Reprezintă rezultatul cererii, indicând dacă aceasta a fost procesată cu succes sau nu. Codul de stare este un număr de trei cifre, fiecare având o semnificație specifică. De exemplu, 200 OK indică faptul că cererea a fost procesată cu succes, în timp ce 404 Not Found indică faptul că resursa solicitată nu a fost găsită pe server.

- Expresia de motiv (optional): Acesta este un text descriptiv care poate însoți codul de stare, oferind o explicație suplimentară a rezultatului cererii. De exemplu, în cazul unui cod de stare 404, expresia motiv poate fi "Not Found", explicând astfel de ce resursa nu a fost găsită.

2. Anteturile de răspuns

După linia de stare, urmează zero sau mai multe câmpuri de antet de răspuns. Aceste câmpuri sunt formate dintr-un nume de câmp, urmat de două puncte și valoarea asociată. Spre deosebire de anteturile de solicitare, acestea sunt folosite pentru a furniza informații despre răspunsul trimis de server și pentru a descrie detalii suplimentare despre răspunsul în sine.

Exemple de antete de răspuns:

- *Content-Type:* Specifică tipul de conținut al răspunsului (de exemplu, *text/html*, *application/json*).
- *Content-Length*: Indică lungimea corpului răspunsului în octeți.
- *Set-Cookie*: Permite serverului să seteze cookie-uri pe client. Un cookie este transmis pe linia sa proprie și poate include atribute suplimentare, cum ar fi secure, httpOnly, și domain, care definesc comportamente speciale ale cookie-ului (de exemplu, secure asigură că cookie-ul va fi trimis doar printr-o conexiune HTTPS).

3. Corpul răspunsului

În final, răspunsul poate include și un corp, care reprezintă efectiv conținutul solicitat de client. Acesta poate fi orice tip de resursă solicitată, de exemplu o pagină HTML, o imagine, sau date JSON, în funcție de cererea originală. Dacă cererea a fost procesată cu succes și resursa este disponibilă, corpul va conține acea resursă (de exemplu, textul unei pagini HTML sau datele unui API).

Un exemplu de răspuns HTTP care include toate aceste elemente este ilustrat în Fig. 2.8.

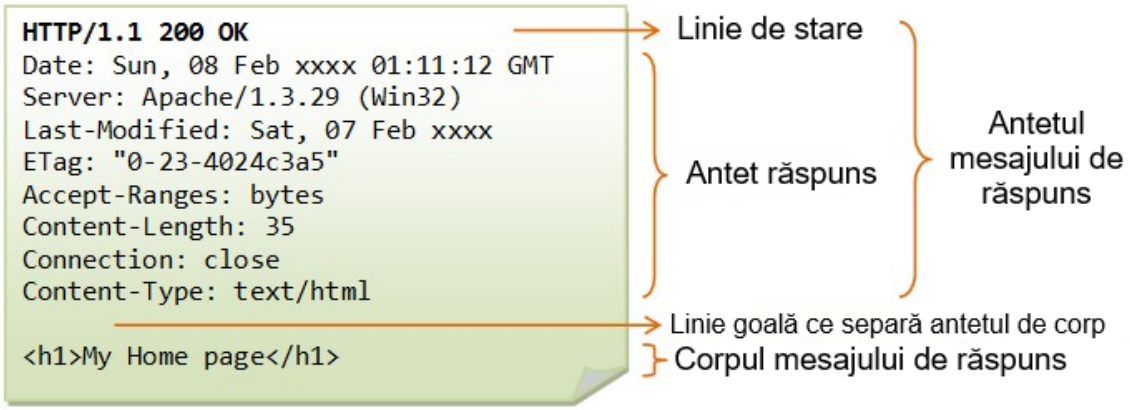



**Fig. 2.8.** Mesajul de răspuns HTTP.

În concluzie, mesajele de răspuns HTTP sunt utilizate pentru a informa clientul despre starea cererii și pentru a furniza informațiile solicitate sau un mesaj de eroare, după caz.

### 2.6.5. Coduri de stare HTTP

În cadrul protocolului HTTP, primul element al liniei de stare a răspunsului reprezintă un **cod de stare**, urmat de o **expresie de motiv** care descrie starea cererii.

**Codurile de stare** HTTP sunt formate din trei cifre numerice, iar fiecare cifră sau grup de cifre are semnificație specifică. Prima cifră indică **clasa** codului de stare, iar restul cifrelor precizează un detaliu mai specific al stării respective. Aceste coduri permit clientului (de obicei un browser) să gestioneze răspunsurile corespunzător.

Codurile de stare HTTP sunt împărțite în următoarele **clase**:

1. **1XX - Informațional**: Aceste coduri indică faptul că cererea a fost primită și procesul este în curs de desfășurare. Clienții nu trebuie să acționeze asupra acestor răspunsuri decât dacă primesc alte răspunsuri semnificative.
2. **2XX - Reușit**: Aceste coduri indică faptul că cererea a fost procesată cu succes. Clientul a trimis o cerere validă, iar serverul a răspuns corespunzător. Exemple: **200 OK** (Cererea a fost procesată cu succes, iar răspunsul conține resursa solicitată), **201 Created** (resursa a fost creată cu succes pe server), **204 No Content**.
3. **3XX - Redirecționare**: Aceste coduri indică faptul că clientul trebuie să întreprindă o acțiune suplimentară pentru a finaliza cererea. În majoritatea cazurilor, aceasta implică redirecționarea către o altă adresă URL.
   Exemple: **301 Moved Permanently** (resursa solicitată a fost mutată permanent la o altă locație), **302 Found** (resursa a fost temporar mutată la o altă locație)
4. **4XX - Eroare client**: Aceste coduri indică faptul că cererea clientului conține erori care împiedică serverul să o proceseze. Aceasta poate include probleme de sintaxă sau parametri incorecți. Exemple: **400 Bad Request** (cererea clientului nu este validă din cauza unei erori de sintaxă), **401 Unauthorized** (clientul nu este autorizat să acceseze resursa solicitată), **403 Forbidden** (serverul a înțeles cererea, dar refuză să o proceseze), **404 Not Found** (resursa solicitată nu a fost găsită pe server).
5. **5XX - Eroare server**: Aceste coduri indică faptul că serverul a întâmpinat o eroare în procesarea unei cereri valide. Aceasta poate fi din cauza unei erori interne a serverului. Exemple: **500 Internal Server Error** (serverul a întâmpinat o eroare internă și nu poate procesa cererea), **502 Bad Gateway** (serverul, acționând ca un gateway sau proxy, a primit un răspuns invalid de la serverul upstream), **503 Service Unavailable** (serverul nu este disponibil temporar din cauza unei suprasarcini sau unei întrețineri).

Clientul trebuie să înțeleagă și să gestioneze corect aceste coduri de stare. În timp ce clientul poate să nu fie familiarizat cu toate codurile de stare, trebuie să fie capabil să trateze corect clasele acestora. În cazul în care clientul întâlnește un cod de stare necunoscut, acesta trebuie tratat ca aparținând clasei respective și gestionat corespunzător.

# 3. Capitolul 3. Arhitecturi bazate pe servicii

## 3.1. Service Oriented Arhitecture (SOA)

Arhitectura bazată pe servicii (Service-Oriented Architecture - SOA) este un stil arhitectural care permite dezvoltarea de aplicații software sub formă de servicii reutilizabile și interoperabile. Conceptul a fost introdus de Yefim Natis într-o lucrare de cercetare din 1994, unde SOA a fost definită ca o arhitectură software care se bazează pe interfețe bine definite. Aplicațiile construite folosind acest model sunt organizate în jurul unei topologii de interfețe, implementări de interfețe și apeluri de interfețe, ceea ce permite o mai mare modularitate și flexibilitate în dezvoltare.

Deși termenul Service-Oriented Architecture a fost introdus încă din anii 1990, adoptarea sa pe scară largă a avut loc abia la începutul anilor 2000. Creșterea a fost impulsionată de dezvoltarea și popularizarea serviciilor web (*Web Services*), alături de standardele asociate acestora, precum WSDL (*Web Services Description Language*). Aceste tehnologii au facilitat schimbul de date și comunicarea între aplicații, indiferent de platforma sau limbajul de programare utilizat.

SOA se bazează pe un model de cerere/răspuns, în care un consumator de serviciu invocă un furnizor de serviciu prin rețea și așteaptă până când furnizorul finalizează operația și returnează rezultatul. Acest mecanism de comunicare permite construirea unor aplicații distribuite, în care funcționalitățile sunt separate în module independente.

O aplicație bazată pe SOA este compusă din două entități principale (Fig. 3.1):

- Consumatorul de serviciu, care este responsabil de interacțiunea cu utilizatorul și de orchestrarea apelurilor către diverse servicii;
- Furnizorii de servicii, care implementează efectiv logica de afaceri și procesele asociate fiecărui serviciu individual.

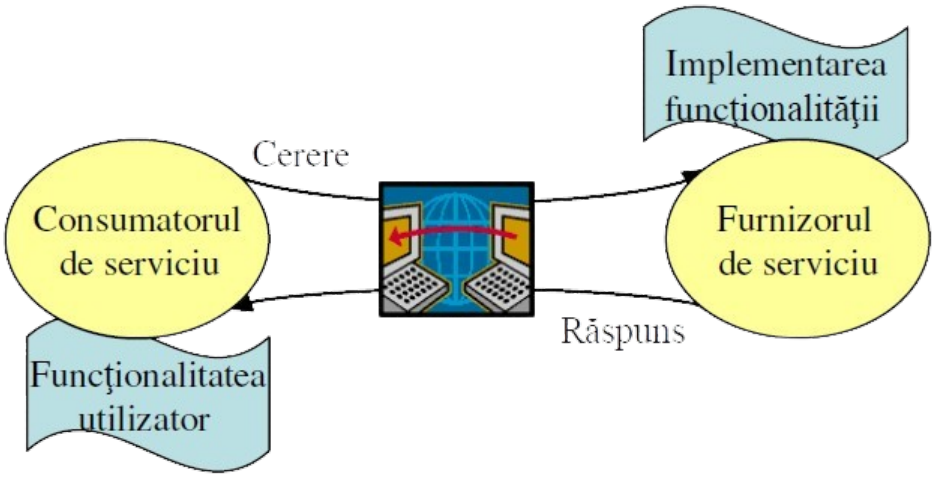



**Fig. 3.1.** Entitățile SOA.

Un aspect important al SOA este faptul că serviciile pot fi reutilizate și partajate între multiple aplicații. În timp ce coordonatorul de serviciu este specific unei anumite aplicații, furnizorii de servicii pot fi integrați și utilizați de mai multe aplicații compozite. Acest lucru contribuie la reducerea redundanței codului, îmbunătățirea mentenanței și optimizarea resurselor de calcul.

Într-un sistem SOA, coordonatorul serviciului nu doar că gestionează interacțiunile dintre utilizator și servicii, dar și specifică în mod explicit ce servicii sunt necesare și le invocă la momentul potrivit. Această abordare permite o mai mare flexibilitate în proiectarea sistemelor software și facilitează integrarea unor servicii externe, inclusiv cele furnizate de terți.

## 3.2. SOA – un stil arhitectural

Arhitectura orientată pe servicii (SOA) nu este doar un set de tehnologii, ci reprezintă un stil de design care influențează toate aspectele dezvoltării software, de la definirea serviciilor, implementarea acestora, până la integrarea și gestionarea lor pe parcursul ciclului de viață al aplicației.

Prin adoptarea SOA, aplicațiile sunt proiectate astfel încât să fie compuse din servicii independente, care pot fi accesate și utilizate de alte sisteme, indiferent de tehnologia în care acestea sunt dezvoltate. Această abordare permite o separare clară între componente, oferind o mai mare scalabilitate, modularitate și reutilizare.

Unul dintre principalele beneficii ale arhitecturii SOA este capacitatea de a construi o infrastructură IT care permite interacțiunea și schimbul de date între aplicații diverse. În mod tradițional, aplicațiile software erau monolitice și greu de integrat cu alte sisteme. Cu SOA, sistemele pot fi decuplate, iar componentele individuale pot comunica între ele prin intermediul unor interfețe standardizate.

Acest model este esențial în medii complexe, unde există multiple sisteme informatice care trebuie să colaboreze. De exemplu, în cadrul unei organizații mari, un sistem de gestiune a relațiilor cu clienții (*Customer Relationship Management* - CRM) trebuie să interacționeze cu un sistem de facturare, cu un sistem de logistică și cu o platformă de e-commerce. Folosind SOA, aceste sisteme pot schimba date într-un mod eficient și flexibil, fără a fi necesară rescrierea codului sau realizarea unor integrări personalizate complexe.

SOA permite nu doar schimbul de date, ci și participarea aplicațiilor în procese comune. De exemplu, un proces de plasare a unei comenzi într-un magazin online poate implica mai multe servicii distincte: un serviciu de autentificare a utilizatorului, un serviciu de verificare a stocului, un serviciu de procesare a plății și un serviciu de expediere a produsului. Aceste servicii pot fi dezvoltate, actualizate și scalate independent, fără a afecta întregul sistem.

Astfel, SOA oferă o bază solidă pentru dezvoltarea aplicațiilor moderne, promovând interoperabilitatea, flexibilitatea și reutilizarea serviciilor software. Aceasta este una dintre principalele motive pentru care arhitectura SOA este utilizată pe scară largă în industrii precum finanțe, telecomunicații, sănătate și comerț electronic.

Conceptul de reutilizare software nu este nou și a fost o preocupare constantă în ingineria software de-a lungul timpului. Pe măsură ce complexitatea aplicațiilor a crescut, a apărut nevoia de a dezvolta soluții care să permită refolosirea componentelor existente, reducând astfel costurile de dezvoltare și timpul necesar pentru implementare.

În arhitecturile orientate pe obiecte (*Object-Oriented* - OO), reutilizarea este realizată prin clase și obiecte, care pot fi instantiate și refolosite în diferite părți ale unei aplicații. Acest model permite crearea unor biblioteci de clase care pot fi utilizate în mai multe proiecte, însă reutilizarea este limitată la nivel de cod sursă și nu oferă întotdeauna o modularitate suficientă pentru aplicații complexe.

Ulterior, arhitecturile bazate pe componente au apărut ca o soluție la nevoia de a reutiliza entități mai mari, care includ nu doar cod sursă, ci și funcționalități complete, independente. Aceste arhitecturi permit dezvoltarea de module reutilizabile, care pot fi integrate în diverse aplicații. Exemple de astfel de componente includ bibliotecile software, framework-urile și modulele reutilizabile dezvoltate pentru platforme specifice.

În cazul SOA, reutilizarea atinge un nou nivel, deoarece serviciile sunt unități independente care pot fi partajate și refolosite între aplicații diferite, indiferent de platformă sau tehnologie. Spre deosebire de clasele și obiectele din programarea orientată pe obiecte, care sunt refolosite la nivel de cod, sau de componentele software, care sunt integrate într-o anumită aplicație, serviciile SOA sunt reutilizabile la nivel de sistem.

Un aspect important al serviciilor în SOA este granularitatea lor. Serviciile pot avea diferite nivele de granularitate:

- Servicii fine (*fine-grained*), care oferă operații specifice (ex.: un serviciu de validare a unui număr de telefon).
- Servicii brute (*coarse-grained*), care combină mai multe funcționalități într-un singur serviciu (ex.: un serviciu de gestionare a contului unui utilizator, care include operațiuni de creare, actualizare și ștergere a contului).

Un alt element esențial în reutilizarea serviciilor SOA este comunicarea prin interfețe bine definite. Spre deosebire de componentele software tradiționale, care pot fi dependente de un anumit limbaj de programare sau platformă, serviciile SOA sunt accesate prin intermediul protocoalelor standardizate (precum HTTP, SOAP, REST) și sunt descrise folosind formate independente de tehnologie, cum ar fi WSDL (*Web Services Description Language*). Această abordare permite reutilizarea serviciilor într-o varietate de aplicații, indiferent de mediul tehnologic în care acestea sunt dezvoltate.

## 3.3. Fundamentele SOA

La nivel conceptual, *Service-Oriented Architecture* (SOA) este construită pe baza unui model clar de interacțiune între diferite entități care facilitează livrarea și consumul serviciilor. Aceste entități sunt esențiale pentru funcționarea eficientă a unui sistem bazat pe SOA și definesc modul în care serviciile sunt publicate, descoperite și utilizate. Arhitectura SOA este alcătuită din trei componente de bază (Fig. 3.2):

1. Furnizor de servicii (*Service Provider*)

Furnizorul de servicii este componenta care creează și oferă servicii către consumatori. Acesta implementează funcționalități specifice și le expune prin intermediul unor interfețe standardizate, astfel încât să poată fi accesate în mod transparent de alte sisteme. Furnizorul poate fi o aplicație software, un sistem enterprise sau un serviciu extern care procesează cererile și furnizează rezultatele către clienți.

2. Consumator de servicii (*Service Consumer*)

Consumatorul este entitatea care apelează serviciile oferite de furnizor. Acesta poate fi o aplicație client, un alt serviciu sau un utilizator final care interacționează cu infrastructura SOA. Consumatorul trimite cereri către furnizorul de servicii și primește răspunsuri conform specificațiilor definite. Comunicarea între consumator și furnizor este realizată prin intermediul unor protocoale standardizate, cum ar fi HTTP, SOAP sau REST.

3. Director de servicii (*Service Directory*) sau Broker

Directorul de servicii joacă rolul de intermediar între furnizor și consumator. Acesta oferă un registru centralizat unde furnizorii își publică serviciile, iar consumatorii pot căuta și descoperi serviciile disponibile. Directorul de servicii poate fi implementat sub formă de registru UDDI (*Universal Description, Discovery, and Integration*) sau alte mecanisme similare care facilitează identificarea și utilizarea serviciilor într-un mod eficient.

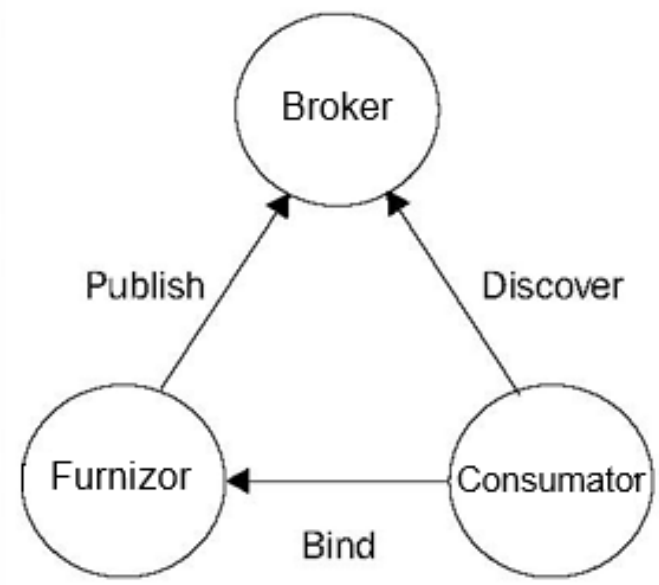



**Fig. 3.2.** Componentele SOA.

Acest model cu trei componente oferă flexibilitate, scalabilitate și interoperabilitate, permițând aplicațiilor să fie integrate și să colaboreze într-un mediu distribuit, fără a fi dependente de platforme specifice sau implementări rigide.

### 3.3.1. Abstractizarea serviciului

În arhitectura bazată pe servicii, fiecare serviciu este însoțit de un set de metadate care definesc modul în care acesta poate fi utilizat. Metadatele conțin informații esențiale despre serviciu și permit consumatorilor să înțeleagă și să interacționeze corect cu acesta.

Una dintre cele mai importante informații incluse în metadate este localizarea în rețea a serviciului, adică adresa prin care acesta poate fi accesat. Aceasta poate fi un URL, o adresă IP sau un alt identificator care permite consumatorilor să trimită cereri către serviciu. În plus, metadatele oferă o descriere într-un format citibil de mașină a mesajelor acceptate și, opțional, a celor returnate. Această descriere este esențială pentru a permite interoperabilitatea între sisteme dezvoltate în tehnologii diferite.

Metadatele definesc, de asemenea, șabloanele pentru schimbul de mesaje, ceea ce ajută la standardizarea comunicării între furnizori și consumatori. O componentă importantă a acestui proces este schema datelor conținute în mesaje, care face parte din contractul dintre furnizor și consumator. Această schemă specifică formatul și structura datelor transmise, asigurându-se că ambele părți implicate pot interpreta corect informațiile. În plus, metadatele unui serviciu SOA includ operațiile suportate, adică acțiunile pe care consumatorii le pot executa asupra serviciului. Acestea sunt expuse printr-un contract de servicii, cum ar fi un document WSDL pentru serviciile web bazate pe SOAP.

Un alt aspect important specificat în metadate sunt cerințele suplimentare, cum ar fi măsurile de securitate necesare pentru utilizarea serviciului. Acestea pot include mecanisme de autentificare, autorizare, criptare sau cerințe privind confidențialitatea datelor.

Prin furnizarea acestor informații, metadatele facilitează descoperirea, utilizarea și integrarea serviciilor SOA într-un mod eficient și sigur, asigurând interoperabilitate și conformitate cu standardele de comunicație utilizate în industrie.

Unul dintre cele mai mari beneficii ale abstractizării în arhitectura SOA este capacitatea de a accesa și utiliza o varietate de servicii, indiferent de originea sau implementarea acestora (Fig. 3.3). Această caracteristică permite integrarea flexibilă a serviciilor noi dezvoltate, care pot fi create pentru a răspunde unor nevoi specifice ale unei aplicații. În același timp, SOA facilitează reutilizarea aplicațiilor existente, permițând încorporarea acestora în ecosistemul de servicii fără a necesita modificări majore. De asemenea, un alt avantaj semnificativ este posibilitatea de a construi aplicații compozite, care combină servicii noi și existente pentru a furniza funcționalități mai complexe. Prin abstractizare, serviciile devin independente de tehnologia utilizată pentru implementare, oferind o soluție scalabilă și adaptabilă pentru dezvoltarea software modernă.

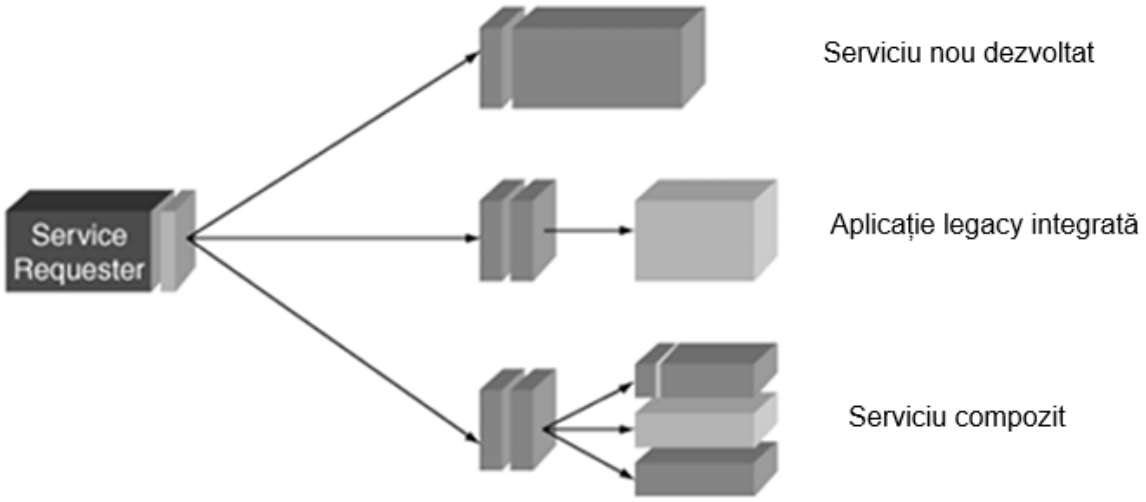



**Fig. 3.3.** Varietatea de servicii în SOA.

### 3.3.2. Agentul executabil și Handler de Servicii

În arhitectura bazată pe servicii (SOA), implementarea unui serviciu este cunoscută sub denumirea de agent executabil. Acesta reprezintă componenta software care rulează într-un mediu de execuție și care implementează efectiv logica serviciului. Mediul de execuție poate varia în funcție de tehnologia utilizată, incluzând servere de aplicații, containere software sau platforme cloud.

Un aspect important al SOA este faptul că descrierea serviciului este separată de agentul executabil. Această separare permite ca un serviciu să aibă mai mulți agenți executabili asociați, fiecare rulând în medii diferite sau fiind implementat în mod redundant pentru a asigura scalabilitate și disponibilitate ridicată.

Pentru a facilita comunicarea între consumatori și serviciul propriu-zis, SOA utilizează un strat de mapare, cunoscut și sub denumirea de strat de transformare. Acest strat joacă un rol esențial în procesarea mesajelor și asigură compatibilitatea între formatele de date utilizate de consumatori și cele necesare agentului executabil. Stratul de mapare este responsabil pentru:

- Acceptarea mesajului trimis de consumatorul serviciului
- Transformarea datelor dintr-un format standardizat (de exemplu, XML sau JSON) într-un format nativ, compatibil cu implementarea serviciului
- Expedierea datelor către agentul executabil pentru procesare.

Prin utilizarea acestui mecanism, SOA permite o interoperabilitate ridicată între sisteme eterogene, facilitând integrarea serviciilor în medii diverse și asigurând o comunicare eficientă și scalabilă între componentele software.

În arhitectura de tip SOA, comunicarea dintre furnizorul de servicii și consumator este gestionată prin intermediul unui handler de servicii. Acesta acționează ca un agent de colaborare, având rolul de a intermedia schimbul de mesaje între cele două părți. Handler-ul

facilitează procesarea cererilor și răspunsurilor, asigurând că mesajele sunt transmise corect și conform standardelor definite.

Un handler de servicii conține logica necesară pentru procesarea mesajelor și gestionarea fluxului de date. Atunci când un serviciu este solicitat, handler-ul verifică căile de transmitere a mesajelor, analizând și traversând diferiți handleri intermediari dacă este necesar. Acest proces permite o rutare eficientă a mesajelor către sistemul țintă și poate include operațiuni suplimentare, cum ar fi transformarea datelor, validarea mesajelor sau aplicarea unor reguli de securitate.

Pe lângă simpla rutare a mesajelor, un handler poate efectua și procesări adiționale, precum autentificarea utilizatorilor, verificarea drepturilor de acces sau optimizarea formatului datelor pentru o performanță mai bună.

## 3.4. Obiectivele și avantajele SOA

Arhitectura bazată pe servicii urmărește o serie de obiective esențiale care contribuie la dezvoltarea unor sisteme software flexibile, scalabile și ușor de întreținut. Unul dintre principalele avantaje ale SOA este cuplarea slabă, care permite descompunerea aplicațiilor în servicii independente. Această abordare reduce dependențele dintre componente, făcând ca modificările într-un serviciu să nu afecteze întregul sistem, ceea ce facilitează mentenanța și evoluția aplicațiilor.

Un alt obiectiv al SOA este neutralitatea de platformă. Deoarece serviciile comunică între ele prin schimb de mesaje folosind standarde precum XML, JSON sau SOAP, ele pot fi implementate în orice limbaj de programare și pot rula pe diferite sisteme de operare sau infrastructuri hardware. Această caracteristică face ca SOA să fie ideală pentru integrarea unor aplicații dezvoltate pe tehnologii diferite (Fig. 3.4).

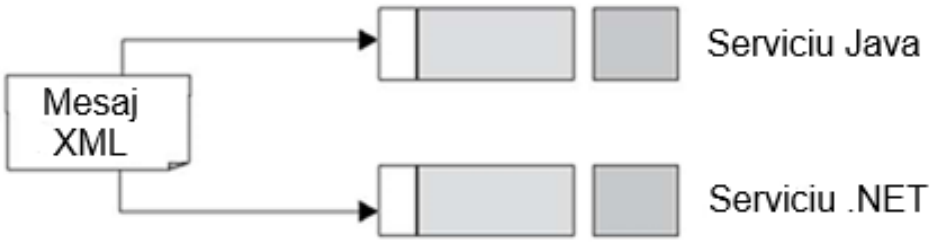



**Fig. 3.4.** Neutralitatea de platformă în SOA.

De asemenea, arhitectura SOA se bazează pe standarde deschise și acceptate la nivel global, ceea ce asigură interoperabilitate și compatibilitate între diverse sisteme. Standardele utilizate în SOA includ WSDL pentru descrierea serviciilor, UDDI pentru înregistrarea și descoperirea serviciilor și protocoale de transport precum HTTP, HTTPS, *Java Message Service* (JMS) sau *Message Queuing Telemetry Transport* (MQTT).

Un alt obiectiv important este reutilizarea serviciilor, care contribuie la reducerea costurilor și a timpului necesar pentru dezvoltare. Deoarece logica aplicației este împărțită în unități funcționale mici și independente, aceste servicii pot fi reutilizate în diferite contexte, fără a fi necesară rescrierea lor, așa cum este ilustrat în Fig. 3.5.

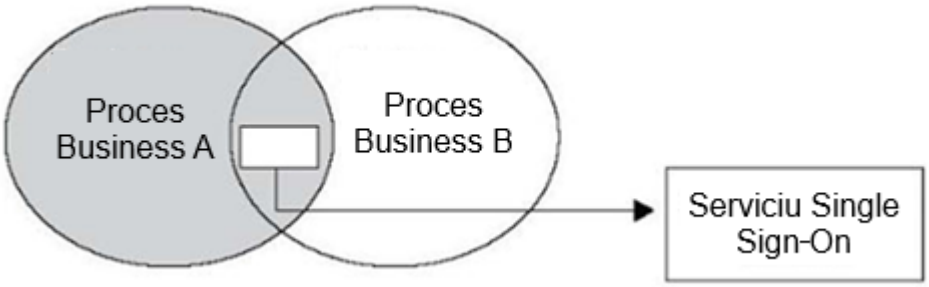



**Fig. 3.5.** Reutilizarea componentelor SOA.

Nu în ultimul rând, SOA promovează scalabilitatea. Adăugarea de noi funcționalități într-un sistem SOA este simplă, deoarece se poate extinde fie prin actualizarea unui serviciu existent, fie prin adăugarea unui nou serviciu care interacționează cu cele deja existente. Această caracteristică face ca arhitectura SOA să fie potrivită pentru organizațiile în continuă creștere, care necesită sisteme capabile să se adapteze rapid la cerințele pieței.

Adoptarea arhitecturii bazate pe servicii (SOA) aduce multiple avantaje, atât din punct de vedere software, cât și hardware, contribuind la dezvoltarea unor aplicații mai flexibile, scalabile și ușor de întreținut.

Unul dintre principalele beneficii ale SOA este capacitatea de a permite dezvoltarea de aplicații slab-cuplate, care pot fi distribuite și accesibile prin rețea. Aceasta înseamnă că serviciile individuale pot fi implementate, gestionate și scalate separat, fără a afecta funcționalitatea întregului sistem. Prin această abordare, aplicațiile devin mai rezistente la schimbări și mai ușor de extins pe măsură ce cerințele evoluează.

Un alt avantaj major este interoperabilitatea dintre aplicații și sisteme diferite. SOA este proiectată astfel încât să faciliteze integrarea între servicii indiferent de platforma, limbajul de programare sau infrastructura utilizată. Folosind standarde deschise precum XML, SOAP, REST și WSDL, soluțiile bazate pe SOA pot comunica eficient și pot fi integrate în medii IT complexe, unde este necesară interacțiunea între aplicații dezvoltate pe tehnologii diverse.

Din punct de vedere software, SOA contribuie la scurtarea perioadei de dezvoltare, deoarece serviciile existente pot fi reutilizate în noi proiecte, reducând astfel efortul necesar pentru dezvoltare de la zero. De asemenea, această arhitectură permite o adaptare mai ușoară la schimbări, ceea ce înseamnă că soluțiile bazate pe SOA sunt mai durabile în timp și pot fi ajustate rapid pentru a răspunde cerințelor în continuă evoluție ale afacerilor.

Pe partea hardware, SOA oferă beneficii legate de balansarea încărcării proceselor într-o organizație. Serviciile pot fi distribuite pe mai multe servere sau în infrastructuri cloud, asigurând o utilizare mai eficientă a resurselor și o mai bună performanță. Această caracteristică este esențială pentru aplicațiile care trebuie să gestioneze volume mari de trafic și să asigure un timp de răspuns optim utilizatorilor.

Implementarea arhitecturii SOA aduce multiple beneficii, însă procesul de tranziție de la un sistem tradițional la un sistem SOA poate fi complex și provocator. Principala dificultate constă în fragmentarea aplicațiilor monolitice existente în servicii mici și independente, astfel încât acestea să poată fi reutilizate și gestionate eficient. Această transformare necesită o analiză atentă a funcționalităților existente și a modului în care acestea pot fi separate fără a afecta integritatea sistemului.

Pentru a realiza această tranziție, există două abordări principale utilizate în practică:

1. Abordarea *Top-Down*: Aceasta presupune o strategie planificată și controlată, în care arhitecții sistemului stabilesc cazuri de utilizare clare și definesc specificațiile pentru crearea serviciilor. Acest proces începe cu identificarea cerințelor de afaceri și proiectarea unei arhitecturi SOA care să răspundă acestor cerințe. Avantajul acestui model este că oferă o viziune clară și coerentă asupra sistemului, însă implementarea poate fi costisitoare și consumatoare de timp, deoarece necesită o reproiectare semnificativă a infrastructurii IT existente.
2. Abordarea *Bottom-Up*: În această strategie, tranziția la SOA pornește de la sistemele curente, identificând procese de business adecvate pentru conversie în servicii independente. Practic, această abordare presupune o modernizare treptată, prin extragerea și reutilizarea funcționalităților existente sub formă de servicii. Avantajul principal este că permite o adoptare graduală a SOA, reducând astfel impactul asupra infrastructurii existente. Totuși, această metodă poate duce la lipsa unei viziuni arhitecturale coerente și poate crea dificultăți în integrarea pe termen lung.

Indiferent de abordarea utilizată, tranziția la SOA necesită o strategie bine definită, implicarea echipei de dezvoltare și utilizarea celor mai bune practici pentru a asigura o implementare eficientă a serviciilor.

Unul dintre principalele avantaje ale implementării SOA folosind servicii web este că acestea sunt pervasive, simple și independente de platformă, permițând interoperabilitatea între aplicații dezvoltate în tehnologii diferite. Deoarece serviciile web se bazează pe standarde deschise și pe infrastructura World Wide Web (WWW), ele oferă un mediu robust pentru schimbul de date. Utilizarea limbajelor de marcare precum HTML și XML facilitează interoperabilitatea, iar protocoale precum HTTP permit un transfer de date universal și eficient. Astfel, serviciile web elimină dependențele de sistemul de operare, serverul web, limbajul de programare sau browserul utilizat, oferind o soluție flexibilă și scalabilă pentru implementarea arhitecturilor bazate pe servicii.

## 3.5. Modelul Arhitectural MVC

Arhitectura MVC (*Model-View-Controller*) este un model fundamental utilizat în ingineria software, având ca scop organizarea clară a codului unei aplicații prin separarea logicii *business* de interfața utilizator. Acest model oferă o structură modulară, în care fiecare componentă are un rol bine definit, ceea ce permite o mai bună gestionare, întreținere și extindere a aplicațiilor software. MVC este utilizat pe scară largă în dezvoltarea aplicațiilor web și desktop, fiind adoptat de numeroase framework-uri și platforme datorită avantajelor sale în ceea ce privește reutilizarea codului.

Principiul de bază al arhitecturii MVC este izolarea logicii aplicației (*business logic*) de interfața utilizator. Aceasta înseamnă că modificările efectuate la nivelul interfeței grafice sau la nivelul regulilor de afaceri pot fi realizate independent, fără a afecta alte componente ale aplicației. Această separare face ca aplicațiile să fie mai ușor de întreținut, deoarece diferite echipe de dezvoltare pot lucra simultan pe straturi separate fără a introduce conflicte majore. În plus, MVC asigură o mai bună organizare a codului, reducând riscul de interdependențe inutile între componente și permițând reutilizarea logicii aplicației în mai multe contexte.

MVC este o arhitectură care separă reprezentarea informațiilor de interacțiunea utilizatorului cu acestea, ceea ce duce la o aplicație mai flexibilă și ușor de extins. Această structură modulară permite dezvoltatorilor să implementeze schimbări în aspectul vizual al unei aplicații sau în logica sa fără a perturba întregul sistem. Datorită acestor caracteristici, MVC a devenit unul dintre cele mai utilizate modele arhitecturale în dezvoltarea software modernă, fiind adoptat de framework-uri populare precum Angular, React, Django, Spring MVC și ASP.NET MVC.

Arhitectura MVC definește aplicațiile software prin separarea acestora în trei straturi distincte, fiecare având un rol bine definit în procesul de gestionare a datelor și interacțiunii utilizatorului. Această divizare asigură o mai bună organizare a codului, facilitând mentenanța, extinderea și reutilizarea componentelor aplicației.

1. **Model** (stratul business/logica aplicației) - responsabil pentru gestionarea datelor și a logicii aplicației.
2. **View** (cfișarea) - responsabilă de prezentarea informațiilor către utilizator
3. **Controller** (controlor de intrare) - acționează ca un intermediar între Model și View, gestionând cererile utilizatorului și direcționându-le către Model pentru procesare.

Prin această structură, MVC asigură o separare clară a responsabilităților și permite dezvoltarea de aplicații scalabile, modulare și ușor de întreținut.

Fig. 3.6 prezintă arhitectura de ansamblu a modelului MVC, precum și modul de interacționare între componente.

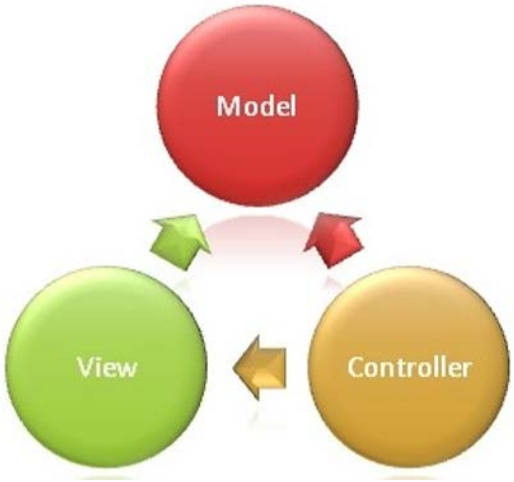



Fig. 3.6. Modelul arhitectural MVC.

### 3.5.1. Componenta Model

În cadrul arhitecturii MVC, Modelul reprezintă componenta centrală a aplicației, fiind responsabil de gestionarea datelor și logicii de business. Acesta definește structura datelor, procesează cererile venite de la utilizator și menține coerența informațiilor din sistem. Modelul este considerat nucleul aplicației, deoarece în această componentă sunt implementate regulile business și operațiile specifice domeniului aplicației.

Modelul are un rol esențial în fluxul aplicației, interacționând atât cu View-ul, cât și cu Controller-ul. Atunci când un utilizator trimite o cerere prin interfața grafică (View), controlerul preia această cerere și o transmite către Model pentru procesare. Modelul accesează baza de date, aplică regulile de afaceri necesare și returnează datele actualizate către controler, care apoi le transmite către View pentru afișare. Această separare clară a responsabilităților permite o dezvoltare modulară și ușor de întreținut a aplicațiilor.

Un aspect important al Modelului este interacțiunea cu baza de date sau cu alte surse de date externe. Modelul se ocupă de stocarea, actualizarea, ștergerea și recuperarea informațiilor, asigurându-se că datele utilizate în aplicație sunt corecte și conforme cu regulile aplicației. De aceea, Modelul este uneori denumit și stratul de domeniu, deoarece definește structura și comportamentul datelor specifice unui anumit domeniu de aplicație, cum ar fi comerț electronic, management de resurse umane sau gestiunea financiară.

Prin separarea Modelului de celelalte componente, arhitectura MVC permite dezvoltatorilor să modifice structura datelor sau regulile aplicației fără a afecta interfața utilizator sau controlerul. Aceasta face ca aplicațiile bazate pe MVC să fie mai ușor de întreținut, testat și extins, contribuind la crearea unor soluții software robuste și eficiente.

### 3.5.2. Componenta View

Componenta View este responsabilă pentru afișarea datelor și interacțiunea cu utilizatorul. Aceasta este partea vizibilă a aplicației, unde utilizatorii pot vedea informațiile procesate de Model și pot efectua acțiuni care vor fi prelucrate ulterior de Controler. View-ul preia datele furnizate de Model și le afișează într-un format accesibil și intuitiv, fără a conține logică business sau detalii despre cum sunt prelucrate datele în fundal.

În mod obișnuit, View-ul este construit folosind limbaje de marcare precum HTML, care definesc structura paginii, și CSS, care se ocupă de stilizare și aspect vizual. În plus, în aplicațiile moderne, View-ul poate include componente dinamice realizate cu JavaScript, jQuery sau diverse framework-uri front-end (ex. React, Angular, Vue.js), care permit o interactivitate sporită și o experiență de utilizare mai fluidă. Prin utilizarea acestor tehnologii, interfața aplicației poate reacționa rapid la acțiunile utilizatorului, fără a necesita reîncărcarea întregii pagini.

Pentru a menține codul organizat și ușor de întreținut, este recomandat ca scripturile JavaScript și jQuery să fie stocate în fișiere separate și referite în View acolo unde este necesar. Această practică ajută la separarea clară a interfeței de logica interactivă, ceea ce duce la o mai bună reutilizare a codului. De exemplu, în loc ca un cod JavaScript să fie scris direct într-o pagină HTML, acesta poate fi plasat într-un fișier extern (*script.js*), care poate fi inclus în mai multe pagini ale aplicației.

Prin această separare clară a responsabilităților, View-ul din arhitectura MVC permite o dezvoltare modulară și flexibilă, facilitând modificarea aspectului vizual al aplicației fără a afecta logica aplicației sau structura datelor. Acest lucru este deosebit de util în cazul în care interfața trebuie actualizată frecvent pentru a îmbunătăți experiența utilizatorului sau pentru a adopta noi tendințe de design.

### 3.5.3. Componenta *Controller*

Controller-ul reprezintă componenta care gestionează interacțiunea dintre utilizator și aplicație. Acesta preia cererile utilizatorului, le procesează și determină acțiunile care trebuie efectuate, stabilind cum trebuie manipulate datele din Model și ce informații trebuie afișate în View. Practic, acesta acționează ca un intermediar între celelalte două componente, asigurând logica de control și gestionarea fluxului aplicației.

Atunci când un utilizator introduce date sau efectuează o acțiune într-o aplicație bazată pe MVC, Controller-ul citește informațiile primite, le procesează și le transmite către Model, unde sunt efectuate operațiile necesare, cum ar fi validarea datelor sau interacțiunea cu baza de date. După ce Modelul a returnat rezultatul, Controller-ul preia aceste date și le direcționează către View, care se ocupă de afișarea informațiilor într-un format prietenos pentru utilizator.

Fiecare Controller conține acțiuni, care sunt funcții responsabile pentru gestionarea anumitor operațiuni. Acțiunile definesc rutele pe care utilizatorii le pot accesa în aplicație. În mod obișnuit, structura unei rute într-o aplicație MVC urmează formatul:

```
{controller}/{action}/{id}
```

, unde:

- controller – reprezintă numele controller-ului care gestionează cererea
- action – este metoda care va fi executată în cadrul controller-ului
- id – este un parametru opțional care poate fi utilizat pentru identificarea unei resurse specifice.

De exemplu, o rută de tip *`Home/Index/4`* va apela metoda *Index()* din HomeController și va transmite valoarea 4 ca parametru. Acest mecanism oferă o organizare clară și ușor de gestionat a interacțiunilor utilizatorului cu aplicația.

Prin separarea logicii de control în Controller, arhitectura MVC permite o gestionare eficientă a cererilor, facilitând extinderea și întreținerea aplicațiilor, precum și îmbunătățirea securității și modularității acestora.

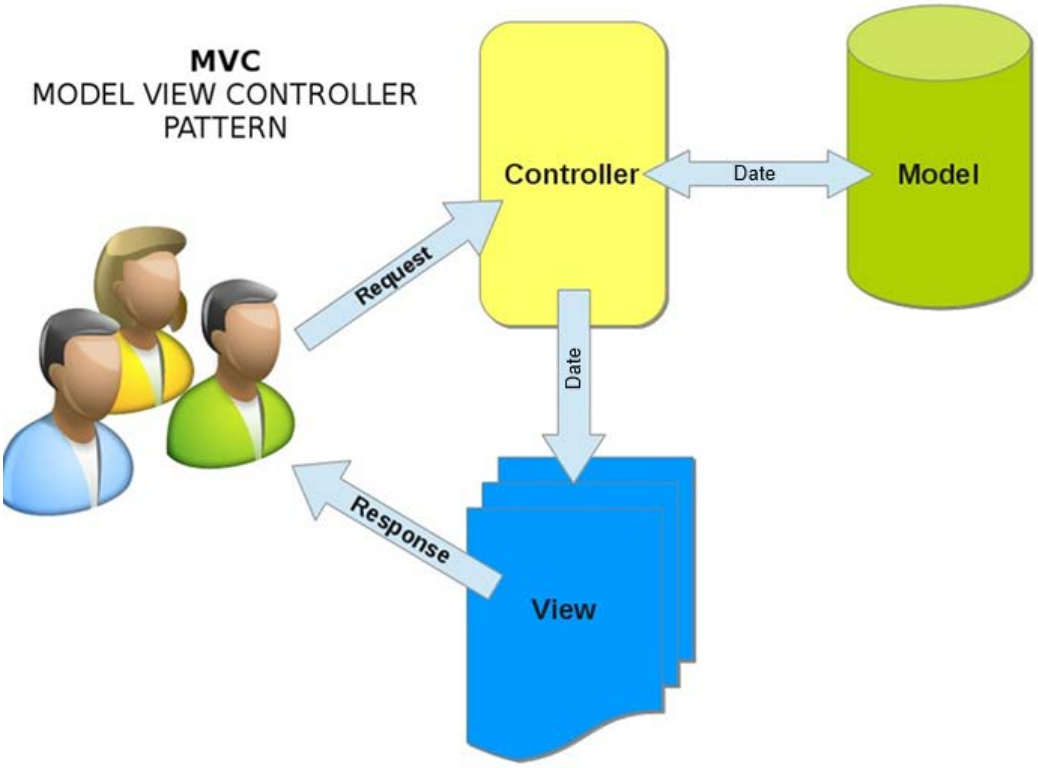



**Fig. 3.7.** Flux MVC.

Fluxul de lucru al arhitecturii MVC, ilustrat în Fig. 3.7, este următorul:

1. Utilizatorul interacționează cu interfața grafică printr-o anumită acțiune (de exemplu, apasă un buton).

2. Acțiunea utilizatorului este preluată de un Controller, care interpretează cererea și determină ce operații trebuie efectuate. Controller-ul poate valida datele introduse și apoi să decidă ce Model trebuie utilizat și poate declanșa acțiunile necesare.
3. Controller-ul creează sau actualizează Modelul, care gestionează datele aplicației. De exemplu, într-o aplicație de comerț electronic, dacă utilizatorul adaugă un produs în coșul de cumpărături, Modelul va actualiza lista de produse din coș și va recalcula totalul comenzii. Aceste modificări sunt stocate și pregătite pentru a fi afișate utilizatorului.
4. După ce Modelul este actualizat, datele sunt transmise către View, care generează o interfață grafică adecvată. De exemplu, View-ul poate afișa lista actualizată de produse din coșul de cumpărături. Un aspect important al arhitecturii MVC este faptul că View-ul primește datele din Model, însă Modelul nu cunoaște direct View-ul, ceea ce asigură o separare clară între logică și interfață.
5. După afișarea datelor actualizate, aplicația rămâne în așteptare pentru următoarea interacțiune a utilizatorului. Acest proces se repetă în mod continuu, permițând utilizatorului să interacționeze cu aplicația și să efectueze diverse acțiuni fără a afecta structura generală a sistemului.

## 3.6. Simple Object Access Procotol (SOAP)

Într-un mediu software modern, comunicarea între aplicații prin Internet este esențială pentru asigurarea interoperabilității între diferite sisteme. Aplicațiile dezvoltate pe platforme diferite trebuie să poată schimba date și să interacționeze în mod eficient, indiferent de tehnologia utilizată. Pentru a facilita această comunicare, a fost creat SOAP (*Simple Object Access Protocol*), un protocol standardizat care permite transferul de mesaje între aplicații distribuite.

Acest protocol este conceput pentru a permite comunicarea între aplicații care rulează pe sisteme de operare și tehnologii diferite. Spre deosebire de alte metode de transfer de date, SOAP oferă o abordare structurată și bazată pe standarde, ceea ce asigură compatibilitatea între platforme diverse. Cel mai eficient mod de a realiza această comunicare este prin utilizarea protocolului HTTP, deoarece acesta este suportat universal de browserele web și serverele de Internet. Astfel, SOAP se bazează pe HTTP ca mecanism principal de transport, facilitând schimbul de informații prin rețele publice și private.

SOAP este definit ca un protocol de comunicare între aplicații, care stabilește un format standardizat pentru trimiterea și primirea mesajelor. Mesajele SOAP sunt scrise în XML (*Extensible Markup Language*), ceea ce le face independente de platformă și de limbajul de programare. Această abordare permite aplicațiilor dezvoltate în tehnologii diferite, cum ar fi Java, .NET, PHP sau Python, să interacționeze fără probleme.

SOAP este o recomandare oficială a W3C (*World Wide Web Consortium*), ceea ce îi conferă un statut de standard recunoscut la nivel global. Deși inițial a fost creat pentru comunicarea prin HTTP, poate funcționa și pe alte protocoale de transport, precum SMTP (pentru e-mail) sau JMS. Această flexibilitate face ca SOAP să fie o alegere potrivită pentru integrarea sistemelor enterprise, în special în medii care necesită fiabilitate, securitate și standardizare a mesajelor transmise.

Pentru a asigura compatibilitatea și interoperabilitatea între diferite sisteme, mesajele SOAP respectă un set strict de reguli de sintaxă. Aceste reguli sunt esențiale pentru ca mesajele să fie interpretate corect de aplicațiile care le procesează.

- Mesajele SOAP trebuie să fie codificate folosind XML - SOAP utilizează XML ca format principal de reprezentare a datelor. XML este un limbaj de marcare creat special pentru stocarea și transportul datelor, fiind atât ușor de citit de către oameni, cât și procesabil de către mașini. Această alegere oferă flexibilitate și portabilitate, permițând schimbul de informații între aplicații dezvoltate în tehnologii diferite.
- Mesajele SOAP trebuie să folosească spațiul de nume SOAP Envelope - fiecare mesaj SOAP trebuie să includă un spațiu de nume XML (*namespace*), care identifică documentul ca fiind un mesaj SOAP valid. Acesta este necesar pentru a evita conflictele cu alte documente XML și pentru a permite parserelor XML să recunoască și să interpreteze corect structura mesajului.
- Mesajele SOAP nu trebuie să conțină o referință DTD - spre deosebire de alte formate XML, SOAP nu permite utilizarea unei definiții de tip *Document Type Definition* (DTD). Acest lucru se datorează faptului că DTD-urile pot introduce restricții suplimentare asupra structurii documentului, ceea ce ar putea afecta interoperabilitatea dintre sisteme diferite.
- Mesajele SOAP nu trebuie să conțină instrucțiuni de procesare XML - SOAP interzice utilizarea instrucțiunilor de procesare XML, deoarece acestea ar putea introduce comportamente necontrolate sau dependențe de platformă. Această regulă asigură că toate mesajele SOAP sunt procesate într-un mod uniform și previzibil, indiferent de implementarea software utilizată.

Un mesaj SOAP este structurat sub forma unui document XML, având o serie de elemente standardizate care definesc conținutul și structura mesajului. Această organizare permite interpretarea corectă a mesajelor de către orice sistem compatibil cu SOAP, indiferent de platformă sau limbaj de programare. Un mesaj SOAP conține următoarele patru elemente principale, așa cum sunt prezentate în Fig. 3.8:

1. Elementul SOAP Envelope
    - Acesta este elementul rădăcină al unui mesaj SOAP și definește întregul document XML ca fiind un mesaj SOAP valid.
    - Este obligatoriu și încadrează toate celelalte elemente din mesaj.

- Conține atributul xmlns, care specifică spațiul de nume XML pentru document. Acest atribut trebuie să aibă valoarea "*http://www.w3.org/2003/05/soap-envelope*". Dacă se utilizează un alt spațiu de nume, aplicația care primește mesajul va genera o eroare și va respinge mesajul.

```
<?xml version="1.0"?>

<soap:Envelope
xmlns:soap="http://www.w3.org/2003/05/soap-envelope"
soap:encodingStyle="http://www.w3.org/2003/05/soap-encoding">

<soap:Header>
...
</soap:Header>

<soap:Body>
...
  <soap:Fault>
  ...
  </soap:Fault>
</soap:Body>

</soap:Envelope>
```

**Fig. 3.8.** Structura unui mesaj SOAP.

2. Elementul SOAP Header (opțional)
   - Conține informații specifice aplicației, cum ar fi autentificarea, securitatea, tranzacțiile sau gestionarea plăților.
   - Dacă este prezent, trebuie să fie primul element copil al elementului Envelope.
   - Permite includerea unor detalii suplimentare fără a afecta conținutul principal al mesajului.
3. Elementul SOAP Body
   - Este componenta principală a mesajului SOAP, conținând mesajul propriu-zis destinat punctului final (serverului care procesează cererea).
   - Elementele conținute în SOAP Body pot fi calificate pentru spații de nume, pentru a se evita ambiguitățile și conflictele de interpretare a datelor.
   - Toate datele relevante pentru cererea SOAP sunt incluse în acest element.
4. Elementul SOAP Fault (opțional)
   - Este utilizat pentru a indica erorile apărute în timpul procesării mesajului SOAP.

- Dacă este prezent, trebuie să fie un element copil al Body și poate apărea o singură dată într-un mesaj SOAP.
- Conține informații despre eroare, împărțite în următoarele subelemente:
    - `<faultcode>` – cod de eroare utilizat pentru identificarea tipului de problemă.
    - `<faultstring>` – explicație în format text, lizibilă pentru utilizatori, despre natura erorii.
    - `<faultactor>` – informații despre ce a cauzat eroarea, utile pentru diagnosticare.
    - `<detail>` – conține informații specifice aplicației, relevante pentru elementul Body.

Exemple de request și response în contextul SOAP sunt ilustrate în Fig. 3.9 și, respectiv, Fig. 3.10.

Fig. 3.9 prezintă un request SOAP pentru a obține prețul unei acțiuni de pe un server web. Cererea este trimisă prin metoda POST către endpoint-ul */InStock* al serverului *www.example.org*, utilizând un mesaj SOAP formatat în XML. Mesajul conține un Envelope, care definește documentul ca fiind un mesaj SOAP, și un Body, unde se află cererea propriu-zisă: apelul metodei *GetStockPrice*, cu un parametru StockName având valoarea "IBM". Serverul va procesa această cerere și va returna răspunsul cu prețul acțiunii IBM.

```
POST /InStock HTTP/1.1
Host: www.example.org
Content-Type: application/soap+xml; charset=utf-8
Content-Length: nnn

<?xml version="1.0"?>

<soap:Envelope
xmlns:soap="http://www.w3.org/2003/05/soap-envelope"
soap:encodingStyle="http://www.w3.org/2003/05/soap-encoding">

<soap:Body xmlns:m="http://www.example.org/stock">
  <m:GetStockPrice>
    <m:StockName>IBM</m:StockName>
  </m:GetStockPrice>
</soap:Body>

</soap:Envelope>
```

**Fig. 3.9.** Request SOAP.

Exemplul din Fig. 3.10 reprezintă un răspuns SOAP la cererea anterioară de obținere a prețului acțiunii IBM. Răspunsul are un cod de stare HTTP 200 OK, indicând că cererea a fost procesată cu succes. Mesajul SOAP conține un Envelope, iar în Body se află elementul *GetStockPriceResponse*, care include valoarea prețului acțiunii, 34.5. Astfel, serverul confirmă că prețul actual al acțiunii IBM este 34.5 unități monetare.

```
HTTP/1.1 200 OK
Content-Type: application/soap+xml; charset=utf-8
Content-Length: nnn

<?xml version="1.0"?>

<soap:Envelope
xmlns:soap="http://www.w3.org/2003/05/soap-envelope"
soap:encodingStyle="http://www.w3.org/2003/05/soap-encoding">

<soap:Body xmlns:m="http://www.example.org/stock">
  <m:GetStockPriceResponse>
    <m:Price>34.5</m:Price>
  </m:GetStockPriceResponse>
</soap:Body>

</soap:Envelope>
```

**Fig. 3.10.** Response SOAP.

# 4. Capitolul 4. Aplicații monolitice și Microservicii.

## 4.1. Servicii Web

Serviciile web reprezintă un set de protocoale și standarde deschise, concepute pentru a facilita schimbul de date între aplicații sau sisteme distribuite. Aceste protocoale permit ca aplicații dezvoltate în limbaje de programare diferite și rulând pe platforme variate să comunice între ele, făcând posibilă integrarea sistemelor într-un mediu heterogen, așa cum este prezentat în Fig. 4.1. Prin utilizarea standardelor deschise, cum ar fi HTTP/HTTPS pentru transport și XML sau JSON pentru formatarea datelor, serviciile web asigură interoperabilitate și portabilitate, eliminând barierele impuse de diferitele tehnologii.


**Fig. 4.1.** Interoperabilitatea serviciilor Web.

Unul dintre avantajele majore ale serviciilor web este capacitatea lor de a funcționa ca interfețe între sisteme disparate. Orice software, aplicație sau tehnologie cloud care folosește protocoale web standardizate pentru a se conecta și a face schimb de informații este considerat un serviciu web. Acest lucru înseamnă că, indiferent dacă este vorba despre o aplicație desktop, o platformă mobilă sau un sistem bazat pe cloud, serviciile web pot fi integrate pentru a permite transferul de date în mod securizat și eficient. Această abordare facilitează crearea unor ecosisteme IT robuste, unde informațiile pot circula liber între diverse componente ale unei infrastructuri mari.

De asemenea, serviciile web pot fi implementate în mai multe moduri. Pe de o parte, un serviciu web poate fi oferit de un dispozitiv către alt dispozitiv electronic, permițând astfel comunicarea între ele prin internet. Acest tip de implementare este întâlnit adesea în scenarii de Internet of Things (IoT), unde diferite echipamente se conectează pentru a schimba date în timp real. Pe de altă parte, un serviciu web poate fi un server care ascultă cererile venite pe un anumit port și care servește resurse precum documente web (HTML, XML, JSON, imagini). În acest caz, serverul răspunde solicitărilor clienților prin transmiterea de date structurate, facilitând astfel crearea de aplicații web dinamice și interactive.

Utilizarea tehnologiilor web standardizate pentru implementarea serviciilor web aduce beneficii semnificative, printre care se numără facilitarea dezvoltării și întreținerii. Protocoalele HTTP și HTTPS sunt acceptate pe scară largă și oferă un mecanism universal pentru transmiterea datelor, indiferent de mediul de operare sau de limbajul de programare utilizat. Aceasta face ca serviciile web să fie o soluție ideală pentru dezvoltarea de aplicații moderne, scalabile și interoperabile, capabile să funcționeze eficient într-un mediu IT din ce în ce mai complex și diversificat.

Serviciile web sunt proiectate să fie accesibile atât prin internet, cât și prin rețele intranet, ceea ce le conferă o flexibilitate mare în mediile distribuite. Această capacitate permite ca aplicațiile să fie accesate de la distanță, facilitând colaborarea între organizații sau departamente, dar și să opereze eficient în rețele interne, unde securitatea și controlul traficului sunt prioritare. Accesibilitatea universală asigură astfel integrarea fără bariere a serviciilor în infrastructuri diverse, indiferent de locația fizică a utilizatorilor sau a resurselor.

Unul dintre elementele cheie ale serviciilor web este utilizarea unui protocol de mesagerie standardizat, bazat pe XML. Acest format standardizat oferă un cadru robust pentru transmiterea datelor, fiind la fel de eficient în a fi procesat de către mașini, cât și ușor de înțeles de către oameni. Datorită caracterului său independent de sistemul de operare sau de limbajul de programare, XML permite ca serviciile web să funcționeze într-un mediu heterogen, unde diverse tehnologii se pot integra fără probleme. În plus, prin utilizarea WSDL, serviciile web se auto-descriu, specificând în mod clar operațiile disponibile, tipurile de date implicate și metodele de comunicare, ceea ce facilitează descoperirea și utilizarea lor de către alte aplicații.

Localizarea serviciilor web este un alt aspect esențial. O abordare simplă a locației permite identificarea rapidă a serviciului dorit, iar aceasta poate fi realizată prin intermediul unor directoare sau registre centralizate, cum ar fi UDDI. Această metodă asigură că aplicațiile pot găsi și consuma serviciile necesare fără a fi nevoie de configurații complicate, contribuind la o integrare eficientă a componentelor sistemului. Astfel, XML și HTTP sunt fundamentale pentru serviciile web, oferind o bază tehnologică universal acceptată.

## 4.2. Arhitectura monolitică

O aplicație monolitică reprezintă o arhitectură software în care toate componentele sunt integrate într-un singur modul unitar. Această abordare tradițională de dezvoltare a fost utilizată pe scară largă de-a lungul timpului, oferind un mediu bine structurat și coerent pentru construirea aplicațiilor. Într-un astfel de sistem, fiecare funcționalitate a aplicației face parte dintr-o bază de cod comună, iar toate modulele sunt strâns legate între ele, ce poate duce la dificultăți în mentenanță și scalabilitate pe măsură ce aplicația crește în complexitate.

Una dintre caracteristicile esențiale ale aplicațiilor monolitice este interdependența componentelor, așa cum arată Fig. 4.2. Aceasta înseamnă că fiecare secțiune a aplicației – fie că este vorba de autorizare, prezentare, logică de business sau acces la baza de date – trebuie să funcționeze corect împreună. De exemplu, orice modificare într-un modul poate necesita testarea și recompilarea întregii aplicații, ceea ce poate îngreuna procesul de dezvoltare și lansare de noi funcționalități.

Într-o aplicație monolitică, componentele principale includ:

- Autorizarea, care gestionează autentificarea și autorizarea utilizatorilor, asigurând accesul securizat la resurse.
- Componenta de prezentare, responsabilă de tratarea cererilor HTTP, interacțiunea cu utilizatorii și servirea răspunsurilor către client. Aceasta include interfața utilizator (UI) și mecanismele de routing.
- Logica de business, care definește regulile și procesele fundamentale ale aplicației. Această componentă este nucleul sistemului și gestionează operațiile esențiale.
- Accesul la baza de date, care oferă interacțiunea cu sistemele de stocare a datelor, permițând citirea, scrierea și modificarea informațiilor necesare funcționării aplicației.

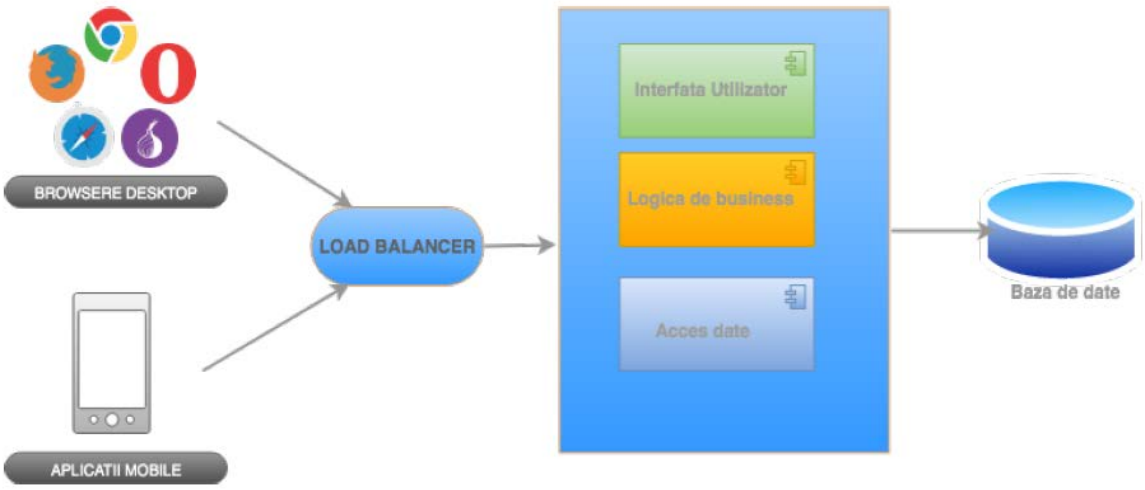



**Fig. 4.2.** Aplicție monolitică.

Arhitectura monolitică poate fi considerată fie un pattern arhitectural, fie un stil de dezvoltare a aplicațiilor, în funcție de modul în care este percepută și aplicată. Deși a fost folosită cu succes timp îndelungat, în prezent, mulți o consideră un anti-pattern, din cauza limitărilor sale în ceea ce privește mentenanța pe termen lung.

Pentru a înțelege mai bine caracteristicile acestui stil arhitectural, se pot identifica trei categorii esențiale:

1. Modul - într-o arhitectură monolitică, codul este organizat în module, fiecare având o anumită funcționalitate. Deși aceste module sunt separate logic, ele sunt compilate împreună într-un singur artefact executabil. Aceasta înseamnă că, deși dezvoltatorii pot lucra pe module distincte, la final, întreaga aplicație este generată ca un întreg și nu există separare fizică între componente. Această abordare poate îngreuna procesul de actualizare și extindere.
2. Alocare - toate componentele unei aplicații monolitice sunt livrate și configurate împreună, sub forma unui singur artefact. Acest lucru înseamnă că orice actualizare sau modificare necesită o nouă versiune a întregii aplicații, indiferent dacă doar un singur modul a fost schimbat. Numărul versiunii este strict legat de numărul livrărilor aplicației, ceea ce poate duce la dificultăți în gestionarea actualizărilor și a compatibilității între diferite componente.
3. Runtime - o aplicație monolitică rulează, de obicei, ca o singură instanță care gestionează toate sarcinile. Acest lucru înseamnă că toate funcționalitățile aplicației sunt executate în cadrul aceluiași proces, ceea ce poate duce la probleme de performanță și scalabilitate. Dacă o anumită componentă a aplicației devine supraîncărcată, întregul sistem poate fi afectat, deoarece nu există posibilitatea de a scala doar partea respectivă independent de restul aplicației.

Arhitectura monolitică prezintă numeroase avantaje, în special în etapele inițiale ale dezvoltării unei aplicații. Este ușor de dezvoltat, deoarece toate componentele sunt integrate într-o singură bază de cod, ceea ce simplifică gestionarea proiectului. De asemenea, testarea este mai simplă, deoarece întreaga aplicație poate fi verificată într-un singur mediu, iar în ceea ce privește implementarea, o aplicație monolitică este ușor de pus în funcțiune, necesitând doar copierea și rularea unui singur artefact pe un server.

Totuși, pe măsură ce aplicația crește în complexitate, arhitectura monolitică începe să prezinte dezavantaje semnificative. Mentenanța devine dificilă, deoarece orice modificare a codului poate afecta alte componente, necesitând teste extinse și un proces atent de integrare. Dimensiunea tot mai mare a aplicației duce la timp de pornire mai lung, ceea ce poate afecta livrarea rapidă a actualizărilor. Aceste dificultățile au condus la adoptarea unor arhitecturi mai flexibile, precum microserviciile, care permit o mai bună modularitate și o gestionare mai eficientă a resurselor.

## 4.3. Microservicii

### 4.3.1. Premise microservicii. Aplicații monolitics vs microservciii

În prezent, utilizatorii se așteaptă ca software-ul să fie întotdeauna disponibil și receptiv, indiferent de volumul de cereri sau de complexitatea operațiunilor. Această cerință impune ca aplicațiile să fie proiectate pentru a oferi performanțe ridicate și să răspundă prompt la orice solicitare, garantând o experiență optimă a utilizatorului.

Cuplarea strânsă a componentelor într-o arhitectură monolitică face ca modificările să fie dificile și costisitoare, întrucât orice schimbare minoră poate afecta întregul sistem. Acest tip de structură limitează flexibilitatea și adaptabilitatea aplicației, ceea ce devine o problemă majoră în contextul inovațiilor tehnologice și al cerințelor tot mai dinamice ale pieței.

Service-Oriented Architecture (SOA) a venit ca o soluție, descompunând aplicația în module mai mici și independente, care pot fi dezvoltate și integrate separat. Totuși, implementările tradiționale de SOA, bazate pe SOAP, prezintă unele probleme – de exemplu, ignorarea codului de răspuns HTTP și contractele rigide impuse de XML și WSDL, care pot reduce flexibilitatea și pot îngreuna evoluția sistemului.

Microserviciile reprezintă o evoluție a conceptelor SOA, abordând parțial aceste limitări. Ele oferă o structură modulară, în care fiecare microserviciu are o funcționalitate bine definită, gestionând propriile date și fiind implementat independent. Această abordare permite scalarea și dezvoltarea rapidă a componentelor individuale, însă nu reprezintă un o soluție pentru toate problemele, deoarece introduce și noi provocări legate de gestionarea comunicării între servicii și de menținerea consistenței datelor în întregul sistem.

Termenul „microserviciu” a fost folosit pentru prima dată în 2011, la un workshop organizat de Software Architects, marcând începutul unei noi paradigme în dezvoltarea aplicațiilor software. În 2014, microserviciile au căpătat o popularitate semnificativă, fiind adoptate pe scară largă în proiecte complexe datorită avantajelor pe care le oferă în termeni de flexibilitate și întreținere. Această abordare a permis dezvoltatorilor să dezvolte soluții modulare, în care fiecare componentă este responsabilă pentru o funcționalitate specifică, fără a impune o dependență strictă de sistemul monolitic.

Arhitectura bazată pe microservicii reprezintă o abordare modernă în dezvoltarea software, în care o aplicație este împărțită într-o colecție de servicii independente și modulare. Fiecare microserviciu este o entitate mică, autonomă și proiectată să funcționeze împreună cu celelalte componente ale sistemului. Această structurare permite o mai bună separare a responsabilităților, facilitând gestionarea și dezvoltarea fiecărui serviciu în mod individual, fără a afecta întregul sistem.

Microserviciile sunt ideal utilizate pentru aplicații mari, unde mai multe echipe lucrează simultan la dezvoltarea diferitelor componente ale sistemului. Această abordare permite integrarea și livrarea continuă (*Continuous Integration/Continuous Delivery* - CI/CD), reducând timpul necesar pentru implementarea și actualizarea funcționalităților. De asemenea,

arhitectura bazată pe microservicii încurajează diversitatea tehnologică, oferind fiecărei echipe libertatea de a alege cele mai potrivite tehnologii și framework-uri pentru microserviciile lor, fără constrângeri impuse de un ecosistem monolitic. Acest lucru contribuie la o mai bună adaptabilitate a aplicației la cerințele în continuă schimbare ale pieței.

Beneficiile microserviciilor se reflectă în special prin îmbunătățirea mentenanței sistemelor complexe. Prin separarea funcționalităților în module independente, echipele de dezvoltare pot lucra paralel, gestionând și testând fiecare serviciu în mod izolat, ceea ce reduce riscul de erori și simplifică procesul de depanare. Astfel, microserviciile nu doar că aduc o reziliență sporită a aplicațiilor, dar facilitează și scalarea și evoluția rapidă a sistemelor software, oferind o soluție modernă pentru provocările impuse de arhitecturile tradiționale monolitice.

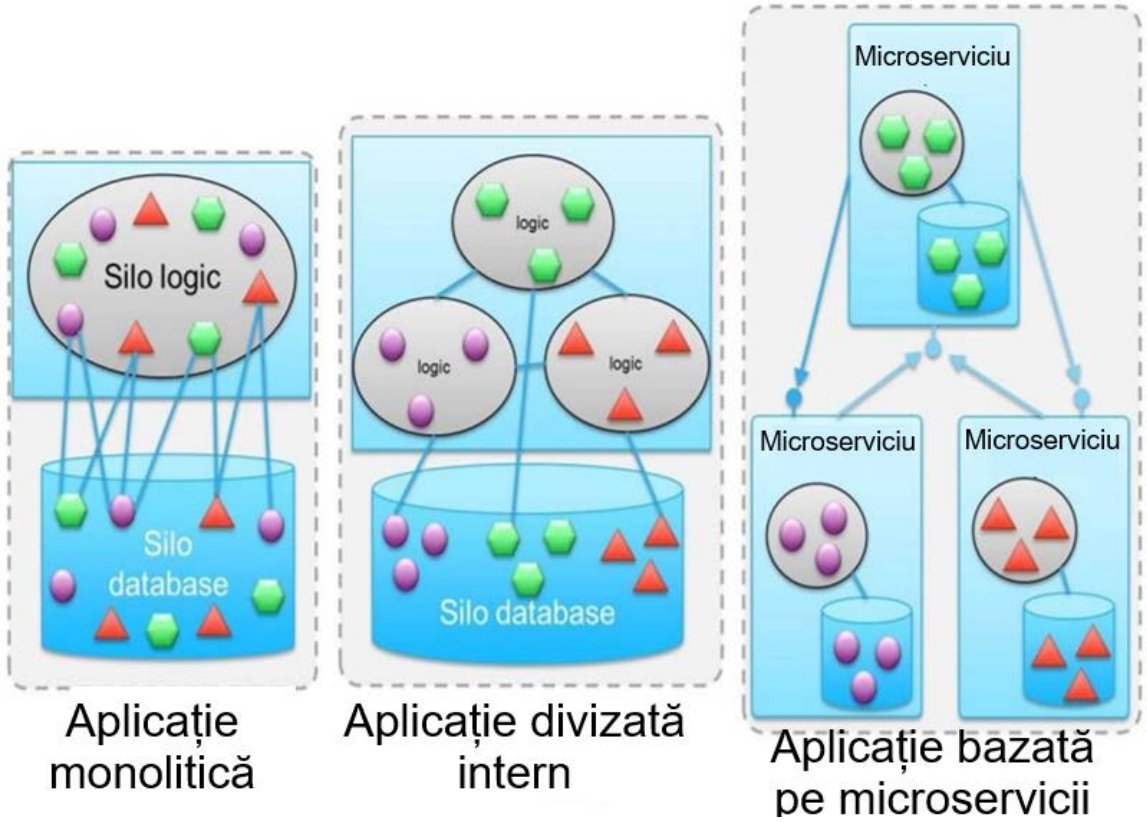



**Fig. 4.3.** Organizarea aplicațiilor software.

Fig. 4.3 ilustrează trei moduri diferite de organizare a aplicațiilor software, evidențiind gradul de separare a componentelor și a bazelor de date. Prima reprezentare ilustrează o aplicație monolitică, în care toate funcționalitățile – logica business, interfața de utilizator și gestionarea datelor – sunt grupate într-un singur modul de cod și împart o singură bază de date. A doua reprezintă o aplicație parțial modularizată, unde componentele sunt separate într-o oarecare măsură, însă continuă să împartă aceeași bază de date și rămân strâns legate

între ele. În cea de-a treia variantă, fiecare componentă a aplicației este implementată ca microserviciu independent, fiecare având propriile date și fiind capabil să comunice cu celelalte prin protocoale ușoare.

În cazul arhitecturii monolitice, întreaga aplicație este dezvoltată și menținută într-un singur modul. Aceasta simplifică inițial procesul de dezvoltare, deoarece toate componentele sunt gestionate împreună, însă devine problematică pe măsură ce aplicația crește în complexitate. Orice schimbare minoră poate necesita recompilarea și redeploy-ul întregului monolit, iar scalarea se face la nivelul întregii aplicații, chiar și atunci când doar o parte a funcționalităților are nevoie de resurse suplimentare. În plus, depanarea și testarea pot deveni dificile, din cauza dependențelor puternice dintre module.

Într-o aplicație intern componentizată, structura monolitică este parțial divizată în module logice. Aceste module pot avea propriile responsabilități, însă împart adesea aceeași bază de date, iar schimbările semnificative necesită totuși ajustări la nivelul întregului sistem. Deși această abordare reduce ușor complexitatea și facilitează în parte întreținerea, limitările arhitecturii monolitice încă persistă: dificultăți în scalare independentă, dependențe între module și posibilitatea ca un singur modul să afecteze restul aplicației atunci când apar erori.

Arhitectura bazată pe microservicii rezolvă o parte din aceste provocări prin împărțirea aplicației în servicii independente, fiecare având propria bază de date și propriul ciclu de viață. Așa cum subliniază Martin Fowler „*stilul arhitectural bazat pe microservicii reprezintă o abordare de dezvoltare a unei singure aplicații ca un ansamblu de servicii mici, fiecare rulând în propriul proces și comunicând prin mecanisme ușoare, deseori un API de resurse HTTP*". Această definiție evidențiază faptul că aplicația este împărțită într-o serie de servicii independente, fiecare având propria logică și rulând separat, în timp ce interacțiunile dintre ele sunt realizate prin protocoale simple, precum HTTP.

Fiecare microserviciu se concentrează pe o funcționalitate specifică, iar echipele de dezvoltare pot lucra, testa și implementa aceste servicii independent unele de altele. Comunicarea dintre servicii se realizează prin protocoale standard (de obicei HTTP/HTTPS), iar scalarea se poate face punctual, doar acolo unde este nevoie de resurse suplimentare.

Comparativ, diferența majoră între o arhitectură monolitică și una bazată pe microservicii constă în gradul de modularizare și independență. În timp ce monolitul obligă toate componentele să evolueze împreună, microserviciile permit o dezvoltare mai agilă și mai ușor de întreținut, deoarece fiecare serviciu rulează într-un proces separat și poate fi adaptat fără a afecta restul aplicației. Totuși, microserviciile aduc propriile provocări, precum gestionarea complexității rețelei, menținerea coerenței datelor și necesitatea unor mecanisme avansate de monitorizare. Prin urmare, alegerea între monolit și microservicii depinde de dimensiunea, cerințele și ritmul de dezvoltare al proiectului.

### 4.3.2. Structură microservicii

Arhitectura bazată pe microservicii oferă o modalitate mai eficientă de decuplare a componentelor dintr-o aplicație, permițând dezvoltatorilor să își segmenteze sistemul în unități independente care gestionează funcționalități specifice. Această separare internă facilitează modificările și scalarea individuală a componentelor, reducând impactul schimbărilor asupra întregii aplicații și crescând agilitatea dezvoltării.

Chiar dacă aplicația este împărțită în microservicii separate, din exterior, ea poate părea la fel ca o aplicație monolitică (Fig. 4.4). Aceasta se datorează faptului că numărul și granularitatea API-urilor expuse nu trebuie să difere semnificativ de cele ale unei aplicații tradiționale. Interfețele sunt proiectate să fie standardizate și coerente, astfel încât consumatorii serviciilor să nu observe complexitatea internă a sistemului, beneficiind de o experiență uniformă și simplificată.

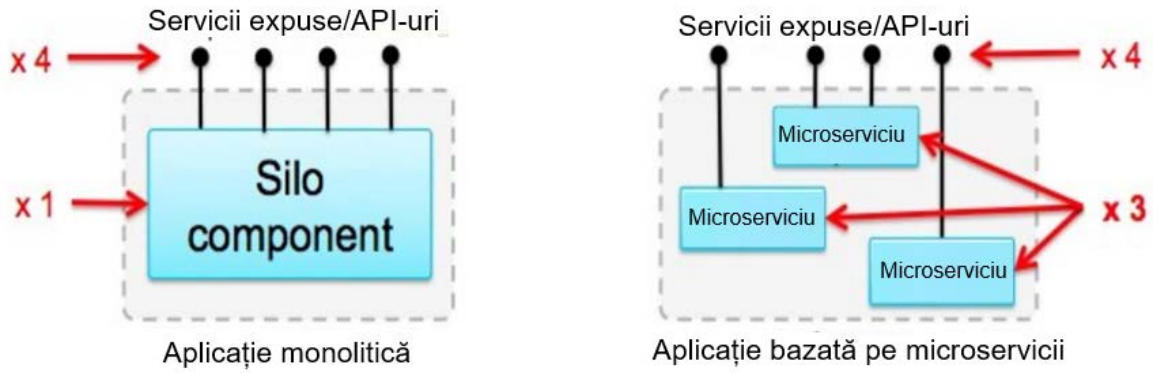



**Fig. 4.4.** Expunere API-uri.

Prefixul „micro” din microserviciu se referă strict la granularitatea componentelor interne, nu la granularitatea interfețelor expuse. Astfel, deși logica și funcționalitățile sunt fragmentate în module mici și independente, API-urile pot fi construite astfel încât să ofere un set unitar de operaționalități, menținând o interfață externă consistentă și ușor de integrat în cadrul altor sisteme. Acest model permite o gestionare flexibilă și eficientă a aplicațiilor complexe, combinând beneficiile modularității cu un nivel de expunere simplificat.

Arhitectura bazată pe microservicii se definește printr-o serie de atribute esențiale, care îi conferă o mai bună gestionare a aplicațiilor distribuite. Unul dintre cele mai importante principii este funcționalitatea izolată, ceea ce înseamnă că fiecare microserviciu trebuie să aibă o singură responsabilitate clar definită și să nu depindă de alte funcționalități. Această separare strictă permite o mai bună modularitate și previne „efectul de domino” în cazul unor modificări sau erori apărute într-un microserviciu.

Un alt atribut esențial este izolarea datelor. Fiecare microserviciu trebuie să își gestioneze propriile date, fără a împărți baza de date cu alte servicii. Această abordare reduce riscul de

conflicte și oferă fiecărui serviciu libertatea de a utiliza tehnologii și modele de stocare a datelor adaptate nevoilor specifice.

În plus, microserviciile trebuie să fie independente în ceea ce privește deployment-ul și execuția. Acestea rulează în procese separate, ceea ce permite scalarea individuală și evitarea dependențelor stricte între servicii. Prin această segregare a serviciilor, microserviciile devin ușor de gestionat atât din perspectiva designului și dezvoltării, cât și din punct de vedere al deploy-ului și întreținerii.

Un aspect important este că microserviciile sunt o arhitectură logică, ceea ce înseamnă că nu trebuie să existe o corespondență 1:1 între un microserviciu și o entitate fizică. De exemplu, un microserviciu business nu trebuie neapărat implementat ca un singur API Web, fiindcă o astfel de regulă ar fi prea rigidă și ar limita flexibilitatea arhitecturală. Această libertate permite echipelor de dezvoltare să ia decizii tehnice optimizate pentru fiecare caz de utilizare în parte, fără constrângeri impuse de o arhitectură monolitică tradițională.

### 4.3.3. Beneficii și provocări ale microserviciillor

Arhitectura bazată pe microservicii aduce numeroase beneficii pentru dezvoltarea și mentenanța aplicațiilor complexe. Printre cele mai importante avantaje se numără:

1. Agilitate și productivitate Microserviciile permit echipelor de dezvoltare să lucreze independent pe componente specifice ale aplicației. Deoarece fiecare echipă poate înțelege complet baza de cod a microserviciului său, acest lucru duce la o dezvoltare mai rapidă și la un proces mai eficient de testare și implementare. Independența componentelor reduce blocajele în fluxul de lucru și permite livrarea mai rapidă a noilor funcționalități.

2. Scalabilitate Fiecare microserviciu poate fi scalat independent, fără a afecta întregul sistem. Acest lucru înseamnă că resursele pot fi alocate în mod eficient acolo unde este nevoie, permițând o utilizare optimizată a infrastructurii IT. De exemplu, un serviciu care gestionează autentificarea poate necesita mai multe resurse în perioadele de vârf, în timp ce alte componente pot rămâne neschimbate.

3. Reziliență. Un sistem construit pe microservicii poate continua să funcționeze chiar dacă una dintre componentele sale eșuează. Prin utilizarea unui design decuplat și evitarea dependențelor sincrone, fiecare microserviciu poate gestiona erorile local, fără a afecta întreaga aplicație. Un exemplu este utilizarea modelului *circuit breaker*, care previne propagarea eșecurilor și îmbunătățește disponibilitatea generală a sistemului.

Implementarea arhitecturii bazate pe microservicii nu doar avantaje, dar și provocări semnificative. Una dintre principalele dificultăți este delimitarea corectă a ariilor fiecărui microserviciu. Este esențial să se identifice datele care nu sunt strâns cuplate și să se stabilească domeniile de responsabilitate ale fiecărui microserviciu. Această separare clară ajută la prevenirea dependențelor nedorite între servicii și la menținerea unei arhitecturi scalabile și ușor de gestionat.

O altă provocare majoră este gestionarea interogărilor ce necesită date din mai multe microservicii. Deoarece fiecare microserviciu are propria sa bază de date, combinarea informațiilor din mai multe surse poate deveni dificilă. Pentru a rezolva această problemă, se pot utiliza mai multe tehnici:

- API Gateway pattern, care agregă datele din mai multe microservicii într-un punct central.
- Materialized View pattern, unde datele sunt pregătite în avans sub forma unor tabele read-only, denormalizate, pentru a optimiza performanța interogărilor.
- Utilizarea "*Cold data*" în baze de date centralizate, unde informațiile istorice (mai puțin frecvent accesate) sunt stocate separat pentru raportare, reducând astfel încărcarea pe microserviciile operaționale.

Un aspect critic al arhitecturii bazate pe microservicii este gestionarea consistenței între multiple servicii. Majoritatea microserviciilor pun accent pe disponibilitate și scalabilitate, în detrimentul consistenței stricte. Pentru a asigura o consistență rezonabilă, se utilizează mecanisme precum comunicarea asincronă, unde datele sunt replicate sau actualizate fără a bloca execuția aplicației principale.

Un alt aspect esențial este modul în care microserviciile comunică între ele. O greșeală frecventă este crearea unor lanțuri de apeluri HTTP între microservicii, ceea ce poate duce la probleme majore:

- Blocaje și performanță scăzută, deoarece un request HTTP așteaptă finalizarea tuturor apelurilor interne.
- Autonomia microserviciilor este compromisă, deoarece acestea devin dependente unele de altele.
- Eșecul unui microserviciu poate duce la căderea întregului sistem, dacă arhitectura nu este concepută corect pentru a gestiona aceste scenarii.

Pentru a evita aceste probleme, comunicarea între microservicii ar trebui să se bazeze pe mesaje asincrone și arhitecturi bazate pe evenimente (*event-driven*). Astfel, fiecare microserviciu poate procesa cereri independent, fără a fi blocat de alte servicii. Această abordare îmbunătățește reziliența și scalabilitatea întregului sistem.

În concluzie, deși microserviciile oferă multiple beneficii, ele adaugă și complexitate arhitecturală. Prin urmare, nu este recomandată adoptarea acestui model decât atunci când o aplicație monolitică devine dificil de gestionat și scalat.

### 4.3.4. API Gateway

În arhitectura bazată pe microservicii, gestionarea interacțiunii dintre clienți și serviciile backend poate deveni complexă, deoarece un client trebuie să trimită cereri către multiple microservicii pentru a obține datele necesare. API Gateway este o soluție care simplifică această interacțiune prin oferirea unui singur punct de intrare pentru un grup de microservicii.

Acest serviciu intermediar acționează ca un proxy inteligent, direcționând cererile și agregând răspunsurile într-un mod eficient, așa cum arată Fig. 4.5.

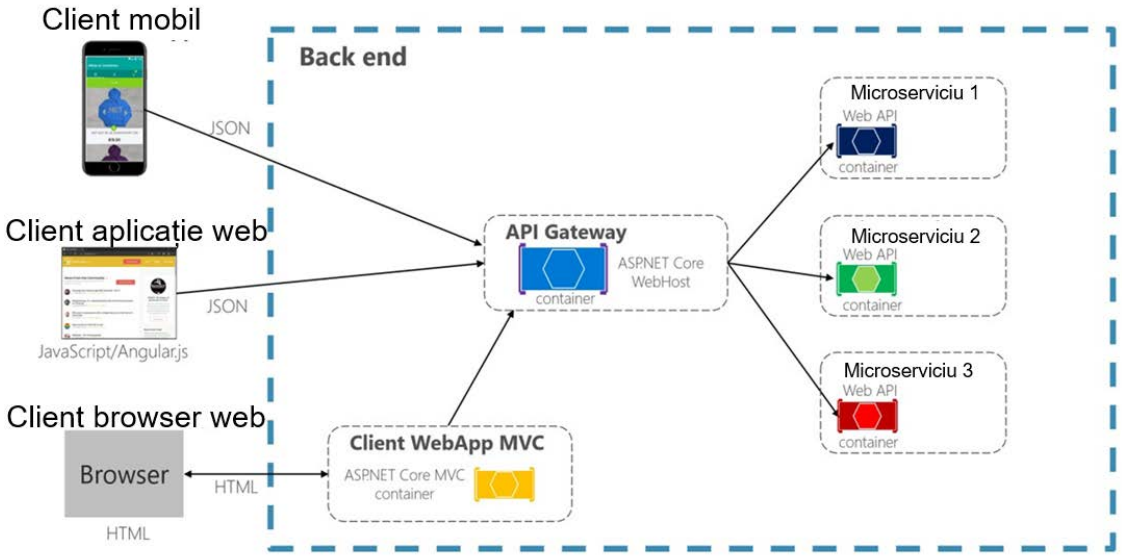



**Fig. 4.5.** API Gateway.

Una dintre funcțiile principale ale unui API Gateway este rutarea cererilor clientului către microserviciile corespunzătoare. De exemplu, atunci când o aplicație mobilă solicită date despre un utilizator, API Gateway analizează cererea și o redirecționează către microserviciul responsabil de gestionarea utilizatorilor. Astfel, clientul nu trebuie să știe locațiile exacte ale microserviciilor, iar complexitatea este ascunsă în spatele acestui punct unic de acces.

Pe lângă rutare, API Gateway realizează și agregarea datelor. Unele operațiuni necesită informații din mai multe microservicii. În loc ca aplicația client să trimită multiple cereri și să gestioneze combinarea rezultatelor, un API Gateway face acest lucru automat. De exemplu, dacă o pagină web trebuie să afișeze informații despre utilizatori, comenzi și produse, API Gateway împărțește cererea inițială, trimite sub-cereri către microserviciile relevante, colectează răspunsurile și le trimite înapoi către client sub forma unui răspuns unificat. Această metodă reduce numărul de solicitări HTTP pe care clientul trebuie să le facă, optimizând performanța și utilizarea resurselor.

Un alt avantaj al API Gateway este capacitatea sa de gestionare a autentificării și autorizării. În loc ca fiecare microserviciu să implementeze propriile mecanisme de securitate, API Gateway poate verifica token-urile de autentificare, gestiona permisiunile și bloca cererile neautorizate înainte ca acestea să ajungă la microservicii. Acest lucru îmbunătățește securitatea și reduce complexitatea microserviciilor individuale.

În plus, un API Gateway poate îmbunătăți performanța și disponibilitatea sistemului prin mecanismul de *load balancing*, distribuind cererile între mai multe instanțe ale microserviciilor

pentru a preveni supraîncărcarea unui singur server. Astfel, se asigură o utilizare optimă a resurselor și se evită blocajele cauzate de creșterea volumului de trafic.

Prin aceste funcționalități, API Gateway devine o componentă esențială într-o arhitectură bazată pe microservicii, facilitând comunicarea eficientă între client și backend, îmbunătățind securitatea și optimizând performanța sistemului.

## 4.4. Exemplu practic microservicii

Arhitectura bazată pe microservicii este ideală pentru aplicații complexe, precum un sistem de gestionare a unui magazin. Într-un astfel de sistem, fiecare funcționalitate poate fi implementată ca un microserviciu independent, ceea ce permite scalabilitate și o mai bună întreținere a aplicației. Un magazin care utilizează microservicii poate avea următoarele componente:

1. Microserviciul de scanare a produselor și adăugare pe factură
    - Permite scanarea codurilor de bare și adăugarea produselor într-o listă de cumpărături.
    - Comunică cu microserviciul de catalog, pentru a verifica disponibilitatea produsului și prețul său actualizat.
2. Microserviciul de scanare pentru verificarea prețului
    - Oferă clienților posibilitatea de a scana produsele pentru a afla prețul, fără a le adăuga în coșul de cumpărături.
    - Acest serviciu poate fi utilizat în aplicații mobile sau în dispozitivele de scanare din magazine.
3. Microserviciul de generare a facturii
    - Procesează lista de cumpărături și generează o factură fiscală.
    - Se integrează cu microserviciul de plăți pentru validarea tranzacției.
4. Microserviciul de plată cu cardul
    - Gestionarea tranzacțiilor prin card bancar, verificarea soldului și aprobarea plății.
    - Se conectează cu băncile sau procesatorii de plăți prin API-uri securizate.
5. Microserviciul de înregistrare a plății
    - Confirmă tranzacțiile reușite și actualizează baza de date privind plata.
    - Asigură integrarea cu sistemul de contabilitate al magazinului.
6. Microserviciul de gestionare a cupoanelor de reducere
    - Permite aplicarea codurilor de discount, cupoanelor și promoțiilor speciale.
    - Se integrează cu microserviciul de facturare pentru a reflecta reducerea în prețul final.
7. Microserviciul de printare a chitanței și trimiterea facturii electronice

- Permite tipărirea bonurilor și trimiterea electronică a facturii pe e-mail sau în aplicația mobilă a clientului.

8. Microserviciul de căutare a produselor în catalog
   - Oferă funcționalități de căutare și filtrare a produselor, afișând informații despre preț, stoc și detalii tehnice.
   - Se integrează cu microserviciul de gestionare a stocurilor, pentru a verifica disponibilitatea produselor.

Arhitectura generală a acestei aplicații, incluzând acțiunile posibile și microserviciile asociate fiecărei acțiuni, este ilustrată în Fig. 4.6.

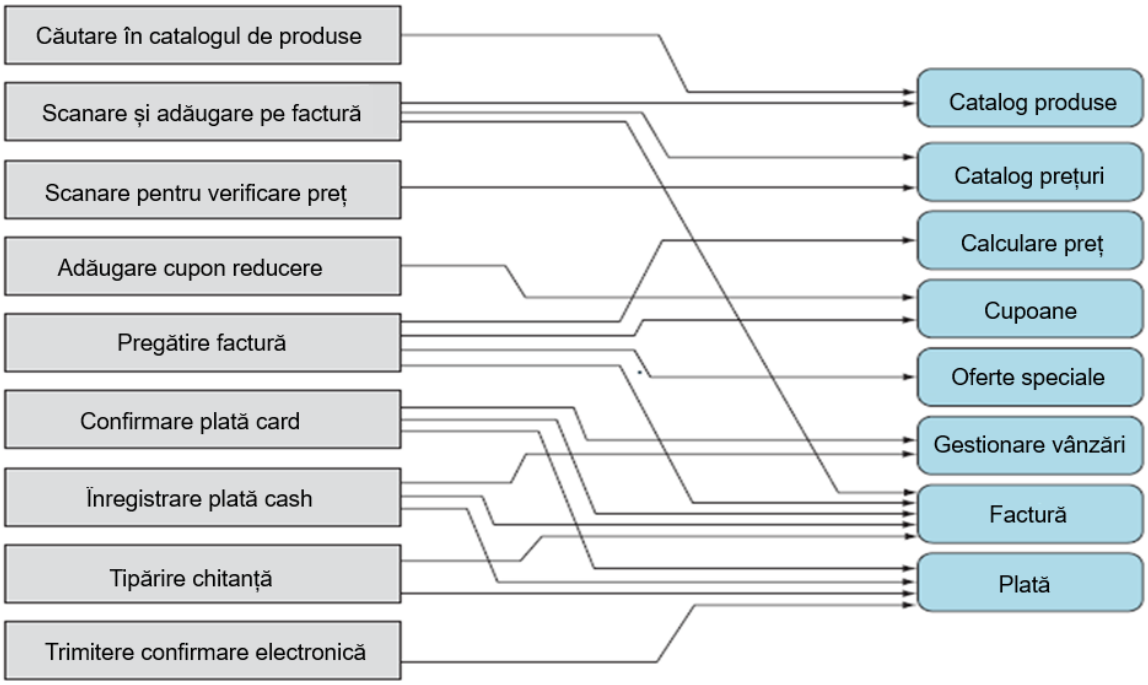



**Fig. 4.6.** Arhitectură aplicației bazată pe microservicii.

Prin utilizarea microserviciilor, acest sistem permite scalarea eficientă a fiecărei componente, independent de celelalte. De exemplu, în perioadele de vârf (Black Friday sau alte sărbători), microserviciul de plăți poate fi scalat separat pentru a face față volumului mare de tranzacții, fără a afecta celelalte funcționalități. Totodată, fiecare microserviciu poate fi dezvoltat și întreținut de echipe separate, utilizând tehnologii diferite. Acest model arhitectural asigură o mai mare agilitate în dezvoltare, flexibilitate în mentenanță și o experiență îmbunătățită pentru clienți.

# 5. Capitolul 5. Arhitectura REST.

REST (*Representational State Transfer*) este un model arhitectural utilizat pentru dezvoltarea serviciilor web, având la bază principiile care pe care s-a bazat succesului World Wide Web. Conceptul a fost definit în anul 2000 de Roy Fielding în teza sa de doctorat, unde acesta a descris o abordare simplă și eficientă pentru construirea aplicațiilor distribuite. Spre deosebire de alte paradigme, cum ar fi *Remote Procedure Call* (RPC) sau serviciile bazate pe SOAP, REST elimină complexitatea suplimentară, oferind o metodă clară și ușor de utilizat pentru schimbul de date între sisteme.

Arhitectura REST este una orientată pe resurse, în care fiecare resursă este accesată printr-un identificator unic, denumit *Uniform Resource Identifier* (URI). Această abordare reflectă modul în care funcționează web-ul în sine, unde fiecare pagină web este o resursă accesibilă printr-o adresă unică (URL). Inspirată de simplitatea și eficiența World Wide Web, arhitectura REST împrumută concepte esențiale care au contribuit la succesul internetului, aplicându-le în dezvoltarea serviciilor web.

Deși REST nu este un standard propriu-zis, el utilizează standarde existente pentru a structura interacțiunile între client și server. Printre acestea se numără:

- HTTP (*Hypertext Transfer Protocol*) – protocolul fundamental al web-ului, utilizat pentru schimbul de mesaje între client și server.
- URI (*Uniform Resource Identifier*) – mecanismul de identificare a resurselor unice pe Internet.
- Formate de date precum XML, HTML, GIF, JPEG – REST nu impune un format specific, însă permite utilizarea diverselor tipuri de conținut pentru reprezentarea resurselor.

Această abordare asigură interoperabilitate, scalabilitate și ușurință în utilizare, făcând din REST una dintre cele mai populare soluții pentru crearea de API-uri și aplicații web moderne.

## 5.1. Principiile REST

Arhitectura REST (Representational State Transfer) se bazează pe un model client-server bine definit, în care implementarea fiecărei părți este independentă. Aceasta înseamnă că dezvoltatorii pot modifica și îmbunătăți codul clientului fără a afecta funcționarea serverului și invers. Această separare clară între client și server asigură o mai mare flexibilitate în dezvoltarea și întreținerea aplicațiilor web.

O caracteristică esențială a sistemelor REST este faptul că acestea sunt apatride (*stateless*). Aceasta înseamnă că fiecare cerere trimisă de client către server este independentă și nu depinde de cererile anterioare. Serverul nu trebuie să mențină informații despre starea clientului, iar fiecare solicitare trebuie să conțină toate datele necesare pentru a

fi procesată corect. Acest model permite scalabilitate ridicată și reduce complexitatea gestionării sesiunilor la nivel de server.

În cadrul arhitecturii REST, interacțiunea dintre client și server se face prin intermediul protocolului HTTP, utilizând metode standard precum GET, POST, DELETE și PUT. Resursele sunt accesate prin identificatori unici (URI), iar datele sunt transferate în formate uzuale precum JSON sau XML.

Fig. 5.1 oferă o reprezentare vizuală a arhitecturii REST. În partea stângă este reprezentat clientul (un computer), care inițiază o cerere HTTP către server. Acesta utilizează metode HTTP specifice (**GET, POST, DELETE, PUT**) pentru a interacționa cu resursele disponibile la diferite adrese (**URI**), cum ar fi `/surveys` sau `/surveys/123`. Serverul procesează cererea și returnează un răspuns către client. Comunicarea dintre cele două părți se realizează prin schimbul de date în format JSON, care conține informații precum ID-ul sondajului (***survey_id***), scorul acordat (***score***), un mesaj text (***message***) și ID-ul răspunsului (***response_id***).

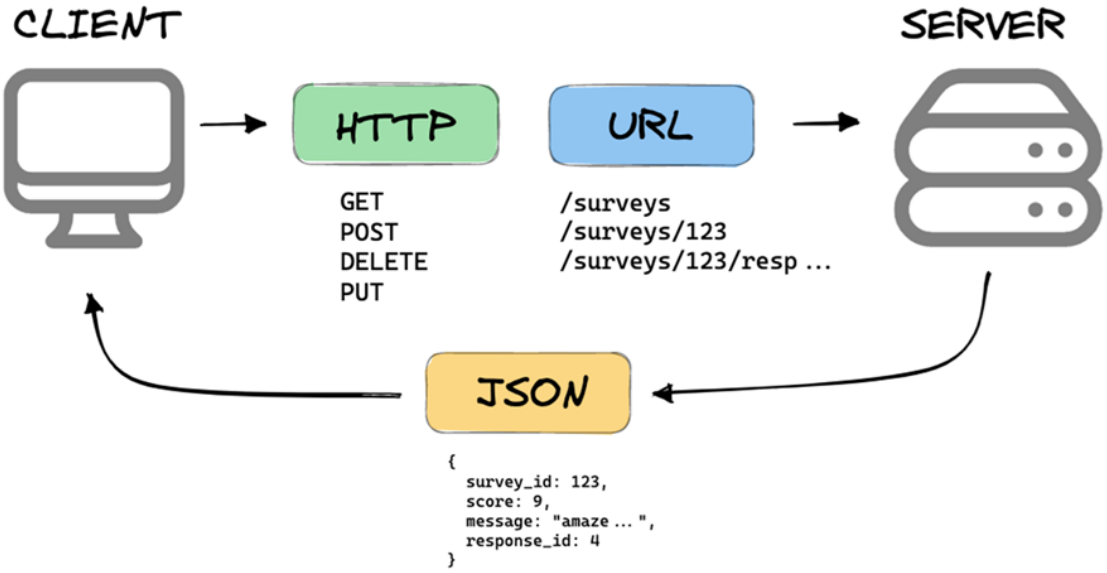



**Fig. 5.1.** Exemplu de intracțiune client-server bazată pe REST.

Arhitectura REST nu este doar un set de principii generale, ci se bazează pe un set de șase constrângeri care definesc modul în care trebuie proiectat un sistem RESTful. Respectarea acestor constrângeri aduce beneficii esențiale precum performanța, scalabilitatea, simplitatea, modificabilitatea, vizibilitatea, portabilitatea și fiabilitatea.

Constrângerile impuse de modelul arhitectural REST sunt următoarele:

1. Modelul Client/Server - arhitectura REST se bazează pe separarea clară dintre client și server, ceea ce permite dezvoltarea independentă a fiecărei componente. Clientul este responsabil pentru interfața utilizatorului și experiența acestuia, în timp ce serverul gestionează datele și logica de afaceri. Această separare îmbunătățește modularitatea aplicației.

2. Stateless (Fără stare) - sistemele REST sunt apatride (*stateless*), ceea ce înseamnă că fiecare cerere HTTP de la client către server trebuie să conțină toate informațiile necesare pentru a fi procesată, fără ca serverul să păstreze un istoric al interacțiunilor anterioare. Aceasta reduce complexitatea serverului, îmbunătățește scalabilitatea și permite distribuirea facilă a solicitărilor către mai multe servere fără a depinde de sesiuni individuale.
3. Cacheabilitate - un alt principiu esențial este cache-ul, care permite stocarea temporară a răspunsurilor pentru a reduce încărcarea serverului și a îmbunătăți performanța. Fiecare răspuns al serverului ar trebui să specifice dacă poate fi cache-uit și pentru cât timp, astfel încât cererile ulterioare pentru aceleași resurse să nu necesite o nouă interogare a serverului. Aceasta îmbunătățește viteza de acces și optimizează utilizarea resurselor de rețea.
4. Interfață Uniformă - una dintre cele mai importante caracteristici ale arhitecturii REST este utilizarea unei interfețe uniforme, ceea ce înseamnă că toate interacțiunile dintre client și server se realizează printr-un set standardizat de metode HTTP (GET, POST, PUT, DELETE). Aceasta simplifică integrarea diferitelor sisteme și permite interoperabilitate ridicată între aplicații.
5. Sistem stratificat - arhitectura REST permite utilizarea unui sistem stratificat, ceea ce înseamnă că un client nu trebuie să știe dacă interacționează direct cu serverul de origine sau cu un intermediar (proxy, load balancer, gateway). Acest lucru ajută la îmbunătățirea securității, optimizarea performanței și distribuirea eficientă a sarcinilor între diferite componente ale sistemului.
6. Cod la cerere (opțional) - ultima constrângere, care este opțională, permite serverelor să extindă sau să personalizeze funcționalitatea clientului prin trimiterea de scripturi sau cod executabil care poate rula pe dispozitivul clientului. Acest mecanism poate fi utilizat pentru a livra funcționalități suplimentare fără a fi necesară actualizarea aplicației clientului, de exemplu, prin descărcarea de scripturi JavaScript pentru interfața web.

De-a lungul anilor, REST a devenit un standard de facto pentru dezvoltarea aplicațiilor web și a serviciilor API.

Totuși, este important de menționat că utilizarea metodei HTTP nu este echivalentă cu respectarea întocmai a principiilor REST. În multe cazuri, API-urile considerate RESTful în industrie nu respectă pe deplin modelul arhitectural REST definit inițial de Roy Fielding, fiind influențate de cerințele practice ale dezvoltării software moderne.

## 5.2. Comunicarea client-server. Request și response

În arhitectura REST, modelul client-server este fundamental și definește modul în care aplicațiile comunică într-un sistem distribuit. Clientul este componenta care inițiază cereri pentru a accesa sau modifica resursele, în timp ce serverul este componenta care procesează aceste cereri și returnează răspunsuri corespunzătoare. Această separare între client și server permite portabilitate și flexibilitate în dezvoltarea și întreținerea aplicațiilor.

O cerere REST urmează un format standardizat și conține mai multe elemente esențiale:

- O acțiune HTTP – definește operația care trebuie efectuată asupra resursei. Cele mai utilizate metode sunt:
    - GET – pentru a prelua date despre o resursă.
    - POST – pentru a crea o resursă nouă.
    - PUT – pentru a actualiza o resursă existentă.
    - DELETE – pentru a elimina o resursă.
- Un antet HTTP (*header*) – include informații despre cerere, cum ar fi formatul datelor acceptate (*`Accept: application/json`*) sau tipul conținutului trimis (*`Content-Type: application/json`*).
- O cale către resursă (URI ) – identifică resursa vizată de cerere, de exemplu:
  ```
  GET /users/123
  POST /orders
  ```
- Un corp de mesaj opțional (*payload/ body*) – utilizat pentru metode precum POST sau PUT, unde trebuie trimise date către server. Aceste date sunt de obicei în format JSON sau XML.

Modelul client-server REST este utilizat pe scară largă în industrie pentru dezvoltarea de API-uri web scalabile. Aplicațiile care respectă principiile REST sunt denumite RESTful, însă în practică acest termen este adesea folosit în special pentru a desemna API-urile bazate pe HTTP. Aceste API-uri folosesc metode HTTP precum GET, POST, PUT și DELETE pentru a opera asupra resurselor, facilitând astfel dezvoltarea și integrarea serviciilor web modern.

Această abordare permite separarea clară între interfața utilizatorului (*frontend*) și logica de procesare a datelor (*backend*), facilitând dezvoltarea aplicațiilor moderne, interoperabile și eficiente. Mai mult, REST simplifică integrarea între sisteme diferite, oferind o modalitate standardizată de comunicare bazată pe protocoale și formate familiare, precum HTTP și JSON.

### 5.2.1. Request REST

Într-un sistem RESTful, fiecare interacțiune între client și server se face prin intermediul unei cereri HTTP. Aceasta conține informații esențiale care îi permit serverului să înțeleagă ce resurse sunt solicitate și în ce format ar trebui să fie returnate datele.

O parte importantă a unei cereri HTTP este antetul (header-ul), care furnizează informații suplimentare despre cerere și despre client. Unul dintre cele mai importante câmpuri din

antet este *Accept*, care specifică tipurile de conținut pe care clientul le poate interpreta. Acest mecanism asigură că serverul nu trimite date într-un format pe care clientul nu îl poate procesa.

Câmpul *Accept* utilizează tipuri MIME (*Multipurpose Internet Mail Extensions*), un standard care definește formatele de date utilizate pe internet. Aceste tipuri MIME sunt alcătuite dintr-un tip principal și un subtip, separate printr-o bară oblică (/).

Exemple de tipuri MIME utilizate în cererile REST:

- Pentru documente text:
  - Text simplu: `text/plain`
  - HTML: `text/html`
- Pentru imagini:
  - PNG: `image/png`
  - JPEG: `image/jpeg`
  - GIF: `image/gif`
- Pentru conținut video:
  - MP4: `video/mp4`
  - Ogg: `video/ogg`
- Pentru schimb de date între aplicații:
  - XML: `application/xml`
  - JSON: `application/json`

Când un client face o cerere REST, el poate include un antet *Accept* pentru a indica formatul preferat al răspunsului. De exemplu, o cerere HTTP către un server API ar putea arăta astfel:

```
GET /users/123 HTTP/1.1
Host: api.exemplu.com
Accept: application/json
```

În acest exemplu, clientul cere informații despre utilizatorul cu ID-ul 123 și specifică că dorește ca răspunsul să fie în format JSON (*application/json*).

Dacă serverul poate furniza datele în format JSON, va răspunde cu un antet corespunzător:

```
HTTP/1.1 200 OK
Content-Type: application/json
```

Aceasta confirmă că răspunsul conține date în format JSON, permițând clientului să le proceseze corespunzător.

Utilizarea tipurilor MIME în antetul cererii este esențială pentru compatibilitatea și interoperabilitatea între aplicații web și servicii RESTful, permițând schimbul de date între sisteme dezvoltate pe platforme diferite.

Într-un sistem RESTful, fiecare cerere HTTP trebuie să conțină o cale care identifică resursa dorită. Aceasta joacă un rol crucial în organizarea și accesibilitatea resurselor dintr-un API, permițând clienților să navigheze și să manipuleze date într-un mod clar și intuitiv.

Pentru o organizare logică și ușor de înțeles, se urmează o convenție standard:

- Resursele sunt denumite la plural pentru a indica faptul că ele reprezintă colecții de entități.
- De exemplu, pentru un magazin online, clienții vor fi identificați sub calea `/customers`, iar comenzile acestora sub `/orders`.

Astfel, o cale precum: `fashionboutique.com/customers/223/orders/12` arată clar că se referă la comanda cu ID-ul 12, aparținând clientului cu ID-ul 223.

Această abordare îmbunătățește lizibilitatea și claritatea API-ului, astfel încât dezvoltatorii să înțeleagă rapid structura și relațiile dintre resurse.

Căile API trebuie să fie suficient de descriptive pentru a localiza resursa exactă dorită, fără ambiguități. Dacă dorim să accesăm o colecție de resurse, putem folosi, iar aceasta va returna toți clienții:

```
GET fashionboutique.com/customers
```

Dacă dorim să accesăm o resursă specifică, adăugăm un ID unic în calea cererii, iar aceasta va returna informațiile despre clientul cu ID-ul specificat:

```
GET fashionboutique.com/customers/{id}
```

La fel, pentru a șterge o resursă individuală, folosim metoda DELETE:

```
DELETE fashionboutique.com/customers/{id}
```

Principiile de organizare a căilor API sunt următoarele:

1. Claritate și consistență – căile ar trebui să fie ușor de citit și de înțeles de către orice dezvoltator.
2. Utilizarea formatului plural – pentru a indica colecții de resurse.
3. Folosirea identificatorilor unici – pentru a accesa o resursă individuală (`/customers/223`).
4. Evitarea verbelor în calea URL – operațiile sunt determinate de metoda HTTP (GET, DELETE etc.), nu de cuvinte/verbe din calea URL (se va evita utilizarea verbelor precum */getCustomer* sau */deleteCustomer*).

### 5.2.2. Response REST

Într-un sistem RESTful, serverul trebuie să trimită un răspuns clar și standardizat către client pentru fiecare cerere efectuată. Acest răspuns conține atât datele solicitate, cât și informații despre tipul de conținut transmis.

Atunci când serverul returnează un răspuns care conține un payload de date, acesta include obligatoriu un antet *Content-Type*. Acest antet informează clientul despre formatul datelor trimise, permițându-i să le interpreteze corect.

Tipul de conținut indicat de server trebuie să fie unul dintre cele acceptate de client, conform antetului *Accept* din cerere. Dacă serverul nu poate furniza un format acceptabil, poate returna o eroare (*406 Not Acceptable*).

Exemplu de interacțiune între client și server:

- Clientul trimite o cerere GET pentru o resursă cu ID 23, specificând formatele acceptabile:

```
GET /articles/23 HTTP/1.1
Accept: text/html, application/xhtml
```

- Serverul răspunde cu un conținut de tip HTML:

```
HTTP/1.1 200 OK
Content-Type: text/html
```

Acest răspuns indică faptul că serverul trimite conținut HTML, conform preferințelor exprimate de client.

La fel ca în cazul cererilor, serverul utilizează tipuri MIME pentru a indica formatul datelor returnate. Câteva exemple comune:

- Fișier HTML: *Content-Type: text/html*
- Fișier JSON: *Content-Type: application/json*
- Fișier XML: *Content-Type: application/xml*
- Imagine PNG: *Content-Type: image/png*

Dacă clientul solicită date într-un format specific, dar serverul nu poate furniza acel format, poate returna un răspuns de eroare:

```
HTTP/1.1 406 Not Acceptable
```

Această eroare indică faptul că serverul nu poate returna datele într-un format acceptabil pentru client.

Atunci când un server RESTful răspunde unei cereri, acesta include un cod de stare HTTP în antetul răspunsului. Acest cod este esențial pentru ca clientul să înțeleagă rezultatul operațiunii și să ia măsurile adecvate.

Ca dezvoltator, nu este necesară cunoașterea fiecare cod de stare HTTP (există peste 60 de coduri oficiale), dar este importantă familiarizarea cu cele mai utilizate, în special în comunicarea dintre client și server într-un API REST.

Pentru fiecare metodă HTTP, există coduri de stare standardizate care indică succesul operațiunii:

- GET → 200 OK – sesursa a fost găsită și returnată cu succes.
- POST → 201 Created – o nouă resursă a fost creată cu succes pe server.
- PUT → 200 OK – resursa a fost actualizată cu succes.
- DELETE → 204 No Content – resursa a fost ștearsă, iar răspunsul nu conține conținut suplimentar.

Dacă operațiunea nu a reușit, serverul ar trebui să returneze cel mai specific cod de eroare posibil, pentru ca clientul să înțeleagă problema.

Câteva exemple importante:

- 400 Bad Request – cererea trimisă de client este incorectă sau nu respectă formatul.
- 401 Unauthorized – clientul nu este autentificat și nu are acces la resursa solicitată.
- 403 Forbidden – clientul este autentificat, dar nu are permisiunea de a accesa resursa.
- 404 Not Found – resursa cerută nu există pe server.
- 500 Internal Server Error – o eroare neașteptată pe server împiedică finalizarea cererii.

Codurile de stare HTTP sunt fundamentale pentru construirea unor API-uri REST clare și ușor de utilizat. Ele permit clienților să gestioneze corect răspunsurile și să implementeze tratamente adecvate pentru succes și eșec. De exemplu, un client care primește un 429 Too Many Requests poate implementa o strategie de retry, iar un răspuns 503 Service Unavailable poate indica necesitatea unei mecanisme de fallback. Utilizarea corectă a codurilor de stare ajută la o mai bună integrare între componentele unui sistem distribuit și contribuie la crearea unor aplicații mai robuste și mai scalabile.

## 5.3. Abordare REST

Într-o arhitectură REST, două trasături esențiale stau la baza modului în care funcționează sistemul: datele asupra cărora clientul solicită operații se regăsesc în URI, iar operația pe care serverul o execută este determinată direct de metoda HTTP folosită. Această separare clară între identificarea resurselor și operațiunile efectuate asupra lor asigură o interacțiune simplă, intuitivă și standardizată, eliminând necesitatea unor protocoale complexe sau a unor setări suplimentare de comunicare.

Din punct de vedere tehnic, arhitectura REST se bazează pe câteva noțiuni fundamentale: resursa, URI, reprezentarea și interfața uniformă. Fiecare resursă este identificată unic printr-un URI, iar reprezentarea acesteia—de obicei în formate precum JSON sau XML—conține datele pe care serverul le transmite clientului. Interfața uniformă, realizată prin metodele HTTP (GET, POST, PUT, DELETE), standardizează modul în care resursele sunt manipulate, asigurând interoperabilitatea și facilitând dezvoltarea de API-uri robuste.

Legarea eficientă a acestor componente conduce la crearea unei arhitecturi REST coerente și ușor de întreținut.

### 5.3.1. Resurse și identifarea acestora

În arhitectura REST, totul este considerat ca fiind o **resursă**. O resursă reprezintă orice entitate ce poate fi identificată și accesată printr-un URI, indiferent dacă este un obiect fizic, un concept abstract sau un set de date. Această definiție largă permite ca, în cadrul unui

sistem RESTful, orice informație ce poate fi stocată pe calculator și reprezentată ca un flux de octeți să fie tratată ca o resursă.

Un aspect esențial al acestei abordări este faptul că resursele nu se limitează la entități fizice. De exemplu, un obiect concret, precum un măr, poate fi reprezentat ca o resursă, dar la fel poate fi tratată și o noțiune abstractă, cum ar fi curajul, atâta timp cât aceasta poate fi descrisă și stocată într-un format digital. Această flexibilitate extinde aplicabilitatea arhitecturii REST la o gamă largă de domenii și tipuri de date.

Printre resursele posibile se numără, de exemplu, versiuni ale unui software (precum versiunea 1.0.3), o hartă a unei țări sau chiar concepte matematice, cum ar fi următorul număr prim după 1024. Aceste exemple ilustrează faptul că resursele pot fi atât entități statice, cât și dinamice, iar reprezentarea lor poate varia în funcție de contextul de utilizare.

De asemenea, resursele pot include și informații generate din activități comerciale sau administrative, cum ar fi numărul de cumpărători pentru un anumit produs sau o listă de defecte dintr-o bază de date. Astfel, sistemele RESTful sunt capabile să gestioneze atât datele de bază, cât și informațiile complexe, asigurând accesul rapid și fiabil la aceste resurse prin intermediul API-urilor.

Fiecare resursă este accesată printr-un URI specific, ceea ce permite un nivel ridicat de descoperire și referențiere în cadrul rețelei. Această identificare unică contribuie la interoperabilitatea între sisteme și facilitează manipularea resurselor prin metode HTTP standard. În plus, separarea clară între resurse și reprezentările acestora adaugă un strat de flexibilitate, permițând dezvoltatorilor să schimbe formatul datelor fără a modifica logica de acces la resurse.

URI (*Uniform Resource Identifier*) reprezintă elementul central pentru identificarea unică a resurselor. Fiecare resursă dintr-un sistem RESTful – indiferent dacă este un document, o imagine, o pagină web sau un obiect digital – trebuie să aibă asociat câte un URI, care servește drept punct de acces și identificator alfanumeric unic. Această unicitate este esențială pentru a asigura că fiecare entitate poate fi accesată și manipulat în mod independent, permițând astfel o navigare și o integrare ușoară între diferitele componente ale sistemului.

Acesta oferă o modalitate standardizată de a desemna resursele, facilitând descoperirea și interacțiunea acestora în mediul web. Spre deosebire de termenul URL (*Uniform Resource Locator*), care se concentrează exclusiv pe aspectul localizării resurselor, URI-ul este un concept mai cuprinzător, care include atât adresele web cât și alți identificatori care pot desemna o resursă, indiferent de locul în care aceasta se află. Cu alte cuvinte, dacă o informație nu poate fi identificată printr-un URI, atunci, în contextul REST, nu poate fi considerată o resursă accesibilă.

Un aspect important al unui URI este faptul că acesta trebuie să fie descriptiv. Un URI bine conceput nu doar identifică resursa, ci oferă și indicii despre natura acesteia, contribuind la claritatea și intuitivitatea API-ului. De exemplu, adresa

`http://www.example.com/software/releases/1.0.3.tar.gz`

indică clar că resursa reprezintă o arhivă a unei anumite versiuni a unui software.

Alte exemple, cum ar fi :

`https://www.unitbv.ro/`,
`https://iesc.unitbv.ro/ro/` și
`https://www.facebook.com/`

demonstrează modul în care URI-urile sunt utilizate pentru a identifica diverse tipuri de resurse, de la site-uri instituționale și platforme educaționale până la rețele sociale.

Utilizarea URI-urilor în arhitectura REST aduce multiple avantaje. Ele permit o organizație clară a resurselor prin stabilirea unei convenții de denumire, care de obicei prevede folosirea formelor la plural pentru a desemna colecții de entități, asigurând astfel o structură logică și coerentă a API-ului. De asemenea, prin faptul că fiecare resursă este identificată univoc, se facilitează implementarea unor mecanisme de cache și de validare a datelor, îmbunătățind astfel performanța sistemului.

Așadar, URI-urile sunt fundamentale în arhitectura REST, deoarece oferă un cadru robust pentru identificarea, accesarea și manipularea resurselor într-un mediu distribuit. Prin standardizarea acestor identificatori și asigurarea că sunt descriptive și unice, sistemele RESTful pot asigura interoperabilitate, claritate și eficiență în schimbul de date pe internet.

### 5.3.2. Interfață uniformă

Interfața uniformă reprezintă unul dintre principiile fundamentale ale arhitecturii REST și stă la baza modului în care clienții și serverele interacționează într-un sistem distribuit. În esență, aceasta presupune utilizarea unui set limitat de operații standardizate pentru manipularea resurselor. Pe întreg web-ul există doar câteva operații de bază ce pot fi efectuate asupra unei resurse, iar HTTP definește patru metode principale care acoperă majoritatea necesităților de interacțiune:

- Pentru primirea unei reprezentări a unei resurse – HTTP GET
- Pentru crearea unei noi resurse - HTTP POST
- Pentru adăugarea de informații atunci când resursa există deja – HTTP PUT/PATCH
- Pentru ștergerea unei resurse – HTTP DELETE

Aceste operații, deși par simple, sunt suficient de puternice pentru a construi un API REST complet funcțional. Ele permit sistemelor să comunice într-un mod clar, coerent și standardizat, fără a necesita mesaje ad-hoc sau apeluri speciale la funcții. În loc să se definească apeluri specifice pentru fiecare funcționalitate, REST se bazează pe aceste metode standard pentru a manipula informațiile disponibile, schimbând radical modul în care sunt concepute și dezvoltate API-urile.

Un alt aspect esențial al interfeței uniforme este că ea promovează o arhitectură fără constrângeri suplimentare, unde comunicarea se face prin cereri standardizate. Acest lucru permite, de exemplu, ca un client să poată modifica sau accesa o resursă fără a cunoaște detaliile interne ale serverului. Prin urmare, clienții pot interacționa cu resursele disponibile doar prin specificarea unei adrese URI și a metodei HTTP adecvate, fără a fi nevoie să se preocupe de implementarea internă a serverului.

De asemenea, prin eliminarea apelurilor speciale și a interfețelor dedicate pentru fiecare funcționalitate, REST promovează un model de dezvoltare modular. Un exemplu ilustrativ este modul în care un bec inteligent poate fi gestionat: în loc să se trimită o cerere specifică de tipul „aprindeBec", clientul poate trimite o cerere PUT cu valoarea „true" la URI-ul dedicat, cum ar fi `http://exemplu.com/Bec/aprinde`. În mod similar, o cerere GET la același URI poate returna starea actuală a becului (adevărat sau fals). Această abordare simplifică dezvoltarea și întreținerea sistemelor, deoarece toate operațiile se bazează pe un set standardizat de metode HTTP.

Pe de altă parte, metodele precum POST, care sunt concepute pentru a adăuga informații, nu ar trebui să fie folosite pentru operațiuni care nu au sens în contextul resursei. De exemplu, nu are logică utilizarea o cerere POST pentru a șterge o resursă, iar astfel de solicitări vor fi respinse cu un răspuns adecvat care indică eroarea. Astfel, interfața uniformă nu doar simplifică comunicarea, dar și impune o disciplină în modul de utilizare a operațiilor HTTP, asigurând că fiecare metodă este folosită conform specificațiilor sale.

### 5.3.3. Reprezentările resurselor și transferul de stări

În arhitectura REST, fiecare resursă poate avea una sau mai multe reprezentări. Această idee pornește de la premisa că o resursă nu este echivalentă cu datele în sine, ci reprezintă o noțiune abstractă – un concept sau o entitate logică pe care creatorul serviciului web a ales să o expună prin intermediul unui URI. De aceea, un server web nu poate trimite conceptul propriu-zis, ci doar o reprezentare a acestuia, sub forma unui flux de octeți, într-un anumit format (de exemplu, JSON sau XML) și folosind un limbaj specific.

Fig. 5.2 relația dintre URI, resursă și reprezentare. Un URI identifică în mod unic o resursă, permițând clienților să o localizeze. Această resursă, la rândul ei, poate avea una sau mai multe reprezentări, în funcție de cerințele aplicației și de preferințele clientului (prin antetul *Accept*, de exemplu). De asemenea, diagrama sugerează că o resursă poate fi deținută de o persoană sau de o organizație, evidențiind astfel faptul că resursele pot fi obiecte reale, concepte abstracte sau entități digitale.

Scopul reprezentării este acela de a furniza datele într-un mod care să poată fi procesat de client. Astfel, dacă serverul are de trimis informații despre un produs dintr-un magazin online, acesta poate alege să ofere reprezentarea resursei în format JSON pentru un browser modern, XML pentru o aplicație legacy sau HTML pentru un client care dorește afișarea într-o

pagină web. Deși toate aceste formate descriu aceeași resursă, fiecare oferă un set diferit de avantaje și niveluri de compatibilitate.

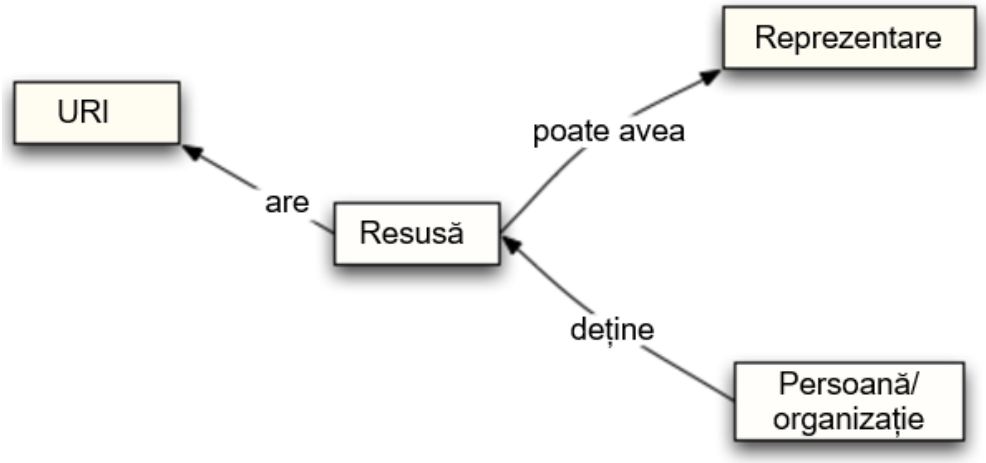



**Fig. 5.2.** Reprezentările unei resurse.

Prin separarea clară a resursei (ca entitate abstractă) de reprezentarea sa (ca flux de date într-un anumit format), arhitectura REST devine foarte flexibilă. Dezvoltatorii pot modifica structura internă a datelor, limbajul de programare folosit sau chiar formatele de reprezentare fără să afecteze URI-ul care identifică resursa. Această abordare facilitează evoluția aplicațiilor, deoarece permite schimbări și îmbunătățiri în modul de expunere a datelor, menținând în același timp stabilă interfața de acces pentru clienți.

Unul dintre conceptele definitorii ale arhitecturii REST este **transferul de stări** (*State Transfer*). Deși până acum am discutat despre resurse, URI-uri, reprezentări și interfața uniformă, aspectul care le leagă într-un stil arhitectural coerent poartă numele de *Representational State Transfer* (REST). Ideea principală constă în faptul că starea aplicației client se schimbă de fiecare dată când acesta accesează o nouă resursă și primește o nouă reprezentare.

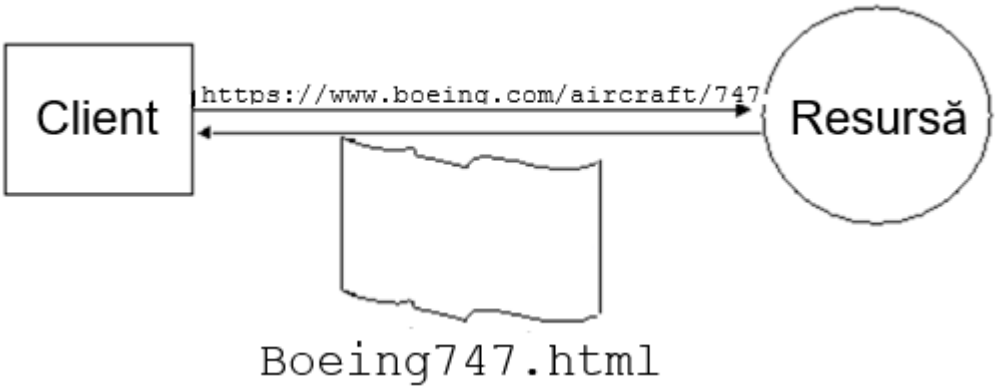



**Fig. 5.3.** Transferul de stări.

Fig. 5.3 prezintă un scenariu simplu, în care clientul accesează o resursă la adresa `http://www.boeing.com/aircraft/747`. Serverul returnează o **reprezentare** a acestei resurse, sub forma unui document HTML (de exemplu, *Boeing747.html*). În momentul în care clientul primește acest document, el „intră" într-o anumită stare, definită de conținutul paginii și de linkurile disponibile în ea. Această stare se modifică atunci când utilizatorul (sau clientul) alege să urmeze un alt link, ceea ce duce la descărcarea unei noi resurse și, implicit, la o nouă stare a aplicației.

Acest proces de navigare între resurse și obținerea diferitelor reprezentări definește transferul de stări. De fiecare dată când clientul primește o nouă reprezentare, aplicația client își modifică starea internă pe baza informațiilor din resursa respectivă. Astfel, spre deosebire de arhitecturile care mențin informații despre client pe server, în REST, clientul însuși gestionează starea prin urmărirea resurselor și a reprezentărilor acestora.

## 5.4. REST vs SOAP

Comparativ cu arhitectura de tip RPC, în care operațiile sunt invocate printr-un protocol dedicat (cum ar fi XML-RPC sau SOAP), arhitectura REST se bazează pe utilizarea directă a metodelor HTTP (GET, POST, PUT, DELETE) pentru a manipula resursele. Astfel, un serviciu web REST folosește cele patru verbe standard pentru a defini clar operațiile posibile asupra resurselor (Fig. 5.4). De exemplu, o cerere GET către `/weatherforecast/02110` returnează starea vremii dintr-o localitate specifică, în timp ce o cerere POST la `/weatherforecast` este utilizată pentru a crea o nouă prognoză meteo și, implicit, pentru a genera un nou URI asociat acesteia. Cererile PUT și DELETE sunt folosite pentru actualizarea unei resurse existente, respectiv ștergerea acestei.

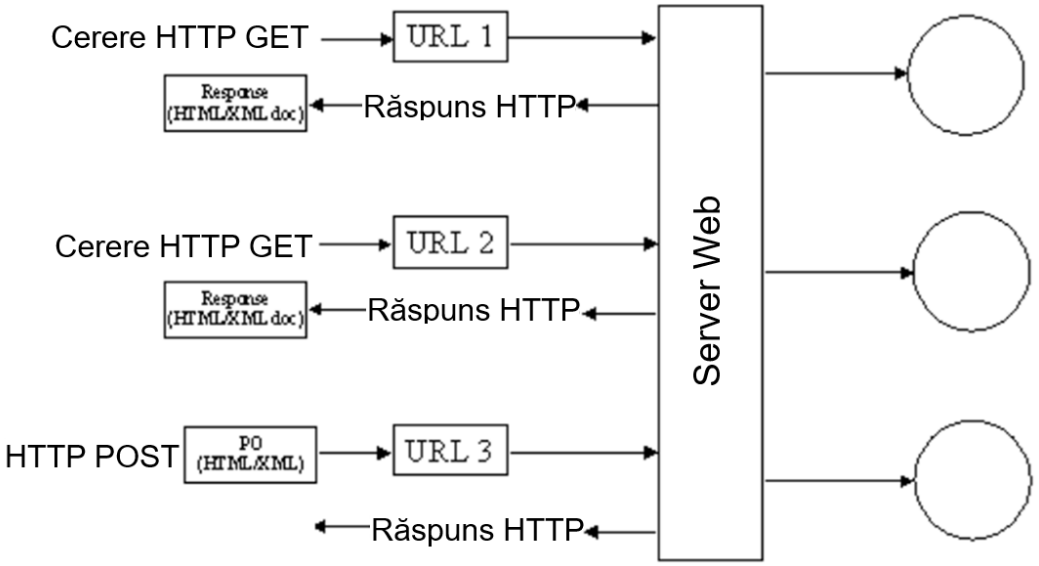



**Fig. 5.4.** Cereri REST.

În contrast, serviciile web care utilizează SOAP se bazează în principal pe metoda HTTP POST, indiferent de operația efectuată, așa cum ilustrează Fig. 5.5. Astfel, fie că este vorba despre obținerea, crearea, actualizarea sau ștergerea unei resurse, toate cererile sunt trimise prin POST la un endpoint comun, cum ar `fi /weatherforecast.asmx`. Operația specifică se diferențiază în interiorul mesajului SOAP, unde se include logica necesară pentru a identifica intenția solicitantului. Acest mod de funcționare introduce un nivel suplimentar de complexitate, deoarece operațiunile sunt incluse în mesaje SOAP, care au propriile reguli de formatare și de procesare.

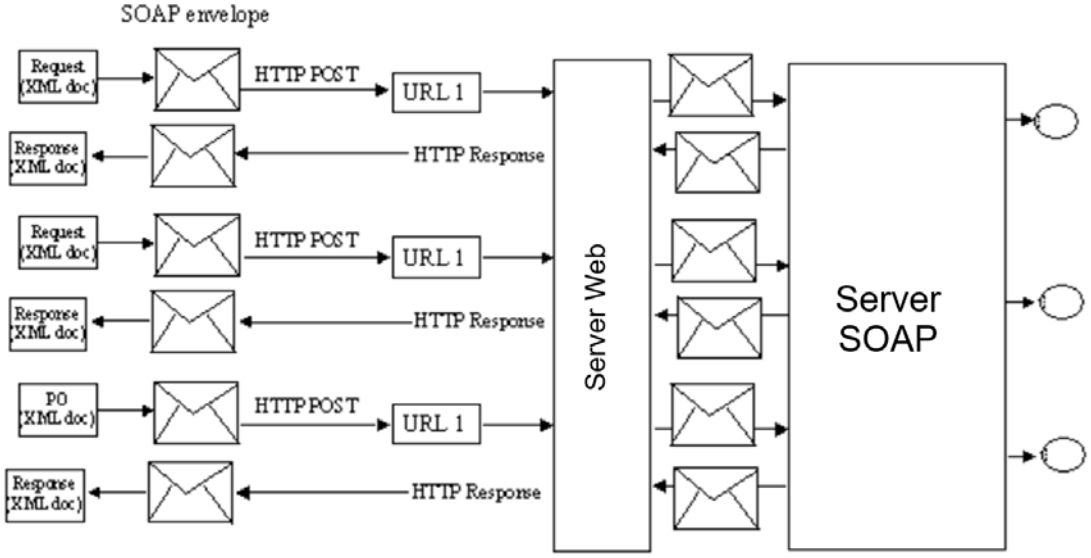



**Fig. 5.5.** Cereri SOAP.

Un alt aspect important al diferenței între REST și SOAP constă în filozofia de proiectare. REST a fost conceput pentru a elimina constrângerile suplimentare, folosind un protocol deja existent – HTTP – fără a reinventa roata. Astfel, REST profită de toate avantajele HTTP, inclusiv scalabilitatea, cache-ul, interfața uniformă și simplitatea în implementare. Pe de altă parte, SOAP impune un contract strict prin mesaje XML și WSDL, ceea ce poate conduce la o complexitate crescută, dar și la o mai bună validare a datelor în anumite scenarii enterprise.

În concluzie, deși ambele abordări sunt utilizate pentru a construi servicii web, REST se remarcă prin simplitate și eficiență, folosind metodele standard HTTP pentru a efectua operațiuni asupra resurselor, în timp ce SOAP, bazat pe mesaje XML trimise prin POST, introduce un strat suplimentar de abstractizare și complexitate. Alegerea între REST și SOAP depinde de cerințele specifice ale aplicației și de contextul în care este implementată, REST fiind de cele mai multe ori preferat pentru aplicațiile moderne web și mobile datorită flexibilității pe care o oferă.

## 5.5. JSON

JSON (*JavaScript Object Notation*) este un format de date text, utilizat pe scară largă în aplicațiile web moderne pentru transportul și schimbul de informații. Deși inițial a fost derivat din limbajul JavaScript, JSON a devenit un standard independent de limbaj, fiind suportat de majoritatea platformelor de dezvoltare. Acest lucru îl face ideal pentru construirea de API-uri REST, unde comunicarea dintre client și server trebuie să fie ușor de implementat și de întreținut.

Una dintre caracteristicile principale ale JSON este faptul că este auto-descriptiv, adică structura datelor este clar definită de formatul JSON însuși. Fiecare obiect JSON este delimitat de acolade ({}) și conține proprietăți sub forma unor perechi cheie: valoare. Această abordare simplifică înțelegerea datelor atât de către oameni, cât și de către mașini.

JSON este independent de limbajul de programare, fiind procesat cu ușurință de către biblioteci existente în Java, Python, C#, JavaScript și multe alte limbaje. Acest aspect favorizează interoperabilitatea între aplicații dezvoltate pe platforme diferite, contribuind la succesul și popularitatea arhitecturii REST. Prin utilizarea JSON în schimbul de date, serverele pot trimite răspunsuri ușor de interpretat de către clienți, iar clienții pot trimite la rândul lor cereri structurate către servere.

De asemenea, JSON se integrează perfect cu metodele HTTP (GET, POST, PUT, DELETE), care stau la baza interfeței uniforme în arhitectura REST. Într-o cerere POST sau PUT, corpul mesajului poate conține un obiect JSON care descrie resursa pe care dorim să o creăm sau să o actualizăm. La rândul său, serverul poate răspunde cu un obiect JSON care reflectă starea resursei, eventual împreună cu un cod de stare HTTP adecvat (de exemplu, 201 Created sau 200 OK).

Sintaxa JSON se bazează pe câteva reguli simple: datele sunt organizate în perechi nume/valoare, unde fiecare nume (sau cheie) este asociat unei valori, iar aceste perechi sunt separate prin virgule. Acoladele sunt utilizate pentru a delimita un obiect. Un obiect poate conține una sau mai multe perechi nume/valoare, reprezentând structura datelor asociate cu o resursă sau o entitate. De exemplu, un obiect JSON ce descrie o persoană poate arăta astfel:

```
{
  "name": "John",
  "age": 30,
  "city": "New York"
}
```

Acest exemplu ilustrează clar modul în care datele sunt structurate: fiecare cheie, precum "name", "age", și "city", este urmată de valoarea sa asociată, iar toate aceste perechi sunt separate prin virgule.

Pe lângă obiecte, JSON utilizează și parantezele pătrate [] pentru a reprezenta matrici (sau array-uri), adică liste ordonate de valori. Valorile dintr-un array pot fi de orice tip de date suportat de JSON: numeric, string, boolean, obiect, alt array sau chiar null. Acest lucru face ca

JSON să fie extrem de flexibil și capabil să descrie structuri de date complexe. De exemplu, un array de nume poate fi reprezentat în modul următor:

```
["Alice", "Bob", "Charlie"]
```

Valorile din JSON sunt tipizate și pot fi:

- Numere (ex: 30, 3.14),
- Șiruri de caractere (ex: "John"),
- Boolean (true sau false),
- Obiecte (ex: { "key": "value" }),
- Array-uri (ex: [1, 2, 3]),
- null (care indică absența unei valori).

Structura JSON se remarcă prin simplitatea și claritatea sa, facilitând procesarea datelor atât pentru oameni, cât și pentru mașini. Această structură ușor de interpretat a făcut ca JSON să devină unul dintre cele mai populare formate pentru schimbul de date în aplicațiile web moderne, contribuind la interoperabilitate și eficiență în comunicarea între sisteme. Prin utilizarea sa, dezvoltatorii pot crea API-uri RESTful robuste, care permit transferul de informații într-un mod standardizat, rapid și ușor de integrat în diverse medii tehnologice.

# 6. Capitolul 6. API-uri RESTful.

## 6.1. Istoria Web – de la Web 1.0 la Web 3.0

De la apariția primelor pagini web statice până la platformele sofisticate și interconectate ale zilelor noastre, evoluția internetului a transformat radical modul în care comunicăm, accesăm informațiile și interacționăm cu tehnologia. Această secțiune prezintă o evoluția Web-ului, începând cu Web 1.0, etapa incipientă caracterizată prin site-uri web statice, limitate în interactivitate și personalizare, și continuând cu Web 2.0, care a introdus conținutul generat de utilizatori, interactivitatea dinamică și rețelele sociale.

Pe măsură ce tehnologiile și cerințele utilizatorilor au evoluat, a apărut conceptul de Web 3.0 – o nouă eră a internetului, bazată pe principii de descentralizare, inteligență artificială și semantica datelor. Web 3.0 promite să creeze o rețea mai inteligentă și conectată, unde informațiile sunt nu doar accesate, ci și înțelese în context, facilitând astfel o experiență personalizată și integrată la scară globală.

### 6.1.1. Web 1.0 – primii pași ai World Wide Web

La începuturile sale, Web-ul a fost conceput ca un mediu de distribuire a informațiilor, în care paginile erau interconectate prin hyperlinkuri, oferind utilizatorilor posibilitatea de a naviga de la un document la altul. Această etapă este cunoscută sub numele de Web 1.0 și se caracteriza prin pagini web preponderent statice, cu un conținut relativ fix, actualizat rar. Pe măsură ce tehnologia a avansat, paginile au început să includă și conținut generat dinamic, permițând afișarea de informații personalizate sau rezultate obținute dintr-o bază de date, însă nivelul de interactivitate rămânea limitat.

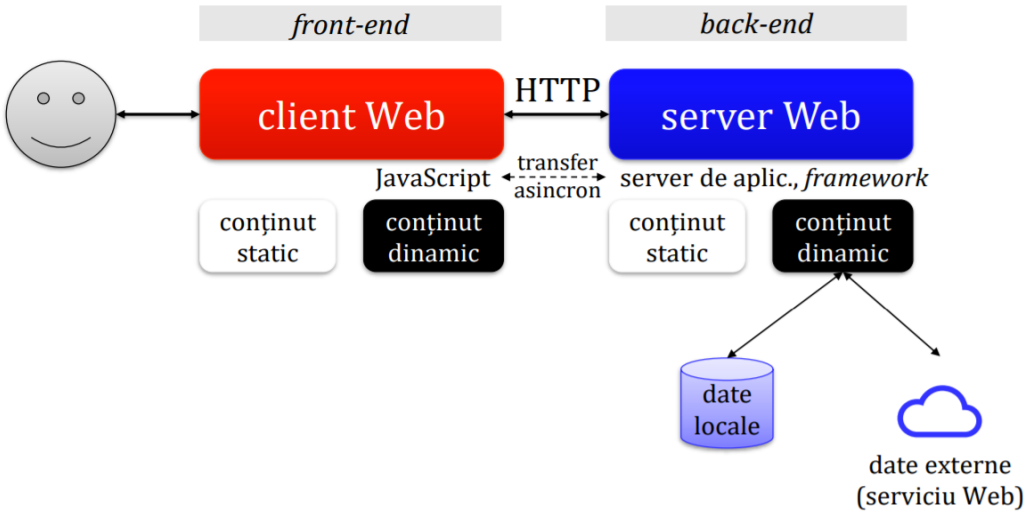



**Fig. 6.1.** Web 1.0.

Așa cum ilustrează și Fig 6.1, o aplicație web (sau un site web) este o colecție interconectată de pagini web, menită să ofere utilizatorilor o funcționalitate specifică. În această arhitectură, interacțiunea dintre aplicație și utilizatori se realizează printr-o interfață web (*front-end*), care rulează pe dispozitivul utilizatorului – de obicei un browser. Utilizatorul inițiază cereri (de obicei cereri HTTP, fie sincrone, fie asincrone), iar aceste cereri sunt trimise către partea de *back-end* (serverul), care procesează datele și returnează răspunsurile sub forma paginilor HTML, imaginilor, fișierelor CSS sau altor resurse.

În contextul Web 1.0, front-end-ul era responsabil doar de afișarea conținutului, iar back-end-ul se ocupa de generarea și livrarea paginilor web. Deși conținutul era generat dinamic, interacțiunile rămâneau în mare parte liniare și unidirecționale: utilizatorul trimitea cereri către server, iar serverul răspundea cu pagini noi, reîncărcând întreaga interfață de fiecare dată. Această abordare ducea adesea la timpi de așteptare semnificativi și la o experiență de utilizare mai puțin fluidă decât ceea ce a devenit posibil ulterior în epoca Web 2.0.

Cu toate acestea, Web 1.0 a reprezentat o bază solidă pentru dezvoltarea ulterioară a internetului, punând în practică principiile fundamentale ale protocolului HTTP și structura hypertext. Pe măsură ce tehnologiile au evoluat și nevoile utilizatorilor au crescut, s-a trecut treptat la aplicații mai interactive și mai complexe, în care front-end-ul a început să preia o parte tot mai mare din logica aplicației, iar back-end-ul s-a diversificat în funcție de nevoile și cerințele de scalabilitate. Astfel, Web 1.0 a fost înlocuit de Web 2.0, care a adus cu sine conținut generat de utilizatori, rețele sociale și aplicații colaborative în timp real.

### 6.1.2. Web 2.0 – Web-ul social

Web 2.0 a marcat o schimbare radicală în modul în care utilizatorii interacționează cu internetul, transformând rețeaua într-un spațiu de colaborare și partajare de informații. Spre deosebire de Web 1.0, unde site-urile erau în mare parte statice și limitate la distribuirea pasivă a informațiilor, Web 2.0 este caracterizat prin aplicații dinamice și servicii web. În această eră, nu mai este vorba despre pachete software monolitice, ci despre servicii specializate, ușor de înlocuit și de integrat, care pot fi consumate de oriunde și pe orice platformă.

Un alt aspect definitoriu al Web 2.0 este transformarea și reutilizarea datelor. Datele sunt, sau ar trebui să fie, disponibile în formate deschise și universale, precum JSON, care permit procesarea și integrarea facilă a informațiilor în diferite aplicații. Această abordare susține ideea de *open data*, în care informațiile sunt accesibile și pot fi partajate pe scară largă, facilitând inovația și colaborarea la nivel global.

Autentificarea și autorizarea descentralizată sunt, de asemenea, elemente cheie ale Web 2.0. Protocoale precum OpenID și OAuth au permis utilizatorilor să se conecteze la multiple servicii cu un singur set de credențiale, simplificând astfel experiența de utilizare și sporind securitatea. Aceste tehnologii au contribuit la crearea unui mediu digital mai integrat, unde

identitatea digitală este gestionată în mod centralizat și utilizată în mod flexibil în diverse aplicații.

În ansamblu, valorile de bază ale Web 2.0 se axează pe partajarea deschisă a datelor, colaborare și interoperabilitate. Această abordare a deschis drumul către o rețea mai interactivă, în care utilizatorii nu doar consumă informații, ci și contribuie activ la generarea și distribuirea acestora, creând astfel un ecosistem web interconectat și în continuă evoluție.

Fig. 6.2 ilustrează rolul esențial al web-ului în ecosistemul cloud. Ea evidențiază trei componente principale: software, platformă și infrastructură. La nivelul software-ului, aplicațiile moderne din cloud acoperă o gamă largă de domenii, cum ar fi prognozele meteo, turismul, divertismentul și social media, oferind servicii interactive și personalizate. În stratul platformă se regăsesc componente critice precum securitatea, bazele de date și gestionarea identității. În final, infrastructura din cloud include stocarea datelor, puterea de calcul și rețeaua, elemente fundamentale care permit scalarea și funcționarea continuă a aplicațiilor la nivel global.

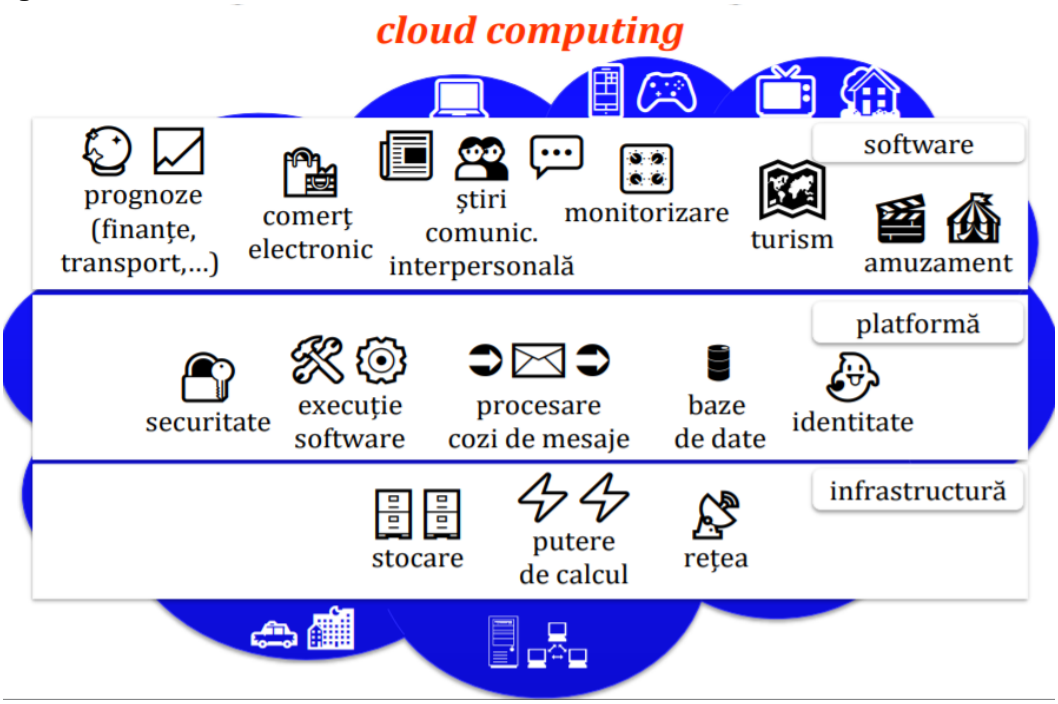



**Fig. 6.2.** Web – ingredient cheie al tehnologiilor Cloud.

### 6.1.3. Web 3.0

Web 3.0, cunoscut și sub denumirea de web semantic, reprezintă o evoluție majoră în modul de organizare și accesare a datelor pe internet. Spre deosebire de versiunile anterioare ale web-ului, care se concentrau pe prezentarea și interacțiunea cu conținutul, Web 3.0 se axează pe înțelegerea și interpretarea datelor la un nivel semnificativ mai profund. Astfel, se pune accentul pe atașarea de meta-date la resursele web, ceea ce permite o descriere detaliată a conținutului și contextului fiecărei entități.

O componentă esențială a Web 3.0 este capacitatea de a specifica relațiile între resurse. Acest lucru se realizează prin organizarea datelor în structuri complexe, cum ar fi grafuri de cunoștințe (*knowledge graphs*), care ajută la modelarea și procesarea cunoștințelor despre aspecte din lumea reală. Prin aceste relații, diferite resurse pot fi legate între ele într-un mod semnificativ, facilitând interogări complexe și extragerea de informații din surse multiple. Fig. 6.3 arată un graf de cunoștințe ce prezintă informații despre albumul „*The Last Years*" al trupei Pink Floyd, lansat în 2019. Elementele sunt organizate sub forma unei propoziții logice, evidențiind subiectul (albumul), predicatul și obiectele (detaliile despre format, audio, tip, înregistrare și subtitrare).

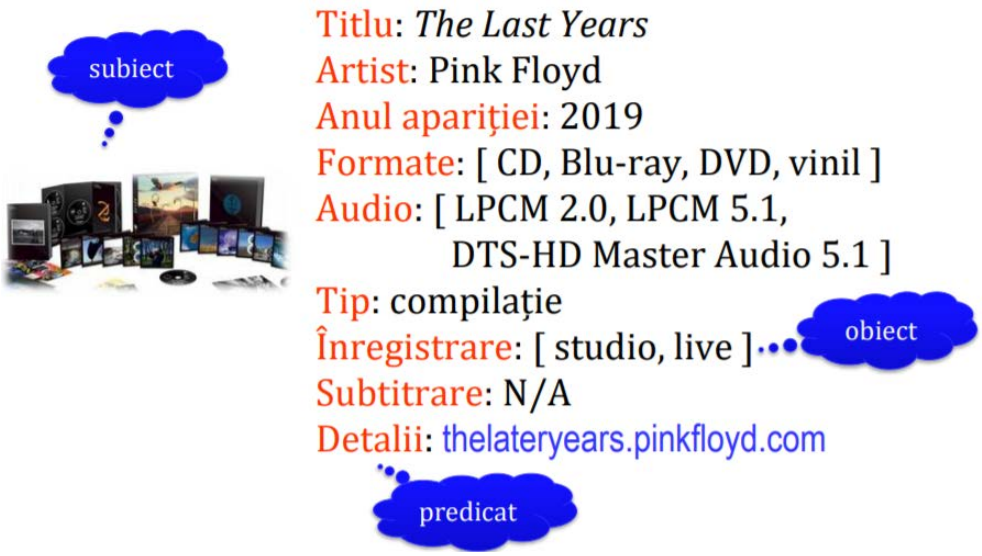



**Fig. 6.3.** Graf de cunoștințe

Pentru a implementa aceste funcționalități, standarde precum RDF (*Resource Description Framework*) sunt esențiale. RDF oferă un cadru de lucru pentru asocierea meta-datelor cu resursele web și pentru specificarea relațiilor dintre acestea. Acest standard permite, de exemplu, ca o resursă care reprezintă o carte să fie legată de alte resurse care descriu autorul, data publicării, genul literar sau recenziile, creând astfel o rețea de informații interconectate.

Web-ul datelor se bazează, de asemenea, pe tehnologii care permit interoperabilitatea între diverse surse de informații. Prin atașarea de meta-date și definirea clară a relațiilor, Web 3.0 facilitează un mod de interacțiune inteligentă cu datele, permițând aplicațiilor să integreze, să proceseze și să prezinte informații într-un mod contextualizat și personalizat. Acest lucru nu doar îmbunătățește căutarea și organizarea informațiilor, dar și deschide noi posibilități pentru dezvoltarea de servicii inteligente și autonome.

În concluzie, Web 3.0 transformă internetul într-un spațiu în care datele nu mai sunt doar stocate, ci și înțelese și valorificate la un nivel semantic. Prin adoptarea unor standarde precum RDF și prin utilizarea grafurilor de cunoștințe, această nouă eră promite să creeze un mediu web mai inteligent și mai interconectat.

## 6.2. Transfer de date în contextul REST

În arhitectura REST, transferul de date este esențial pentru schimbarea stării unei aplicații. Atunci când un client accesează o reprezentare a unei resurse, acesta nu primește doar datele solicitate, ci intră într-o nouă stare definită de informațiile primite. Fiecare transfer de date, efectuat prin metodele standard ale protocolului HTTP (cum ar fi GET, POST, PUT sau DELETE), determină o tranziție de la o stare la alta, permițând astfel evoluția interacțiunii dintre client și server

Această abordare se bazează pe conceptul de *hypermedia*, unde reprezentările resurselor conțin legături către alte resurse, formând un graf de conexiuni. Astfel, dintr-o anumită stare, reprezentată de un document sau un set de date, clientul poate naviga către alte resurse prin accesarea link-urilor disponibile. Fiecare accesare a unui nou link reprezintă un transfer de date care modifică starea aplicației, definind în mod continuu contextul și posibilitățile de interacțiune.

Tranzițiile dintre stări sunt dictate de metodele HTTP, care nu doar specifică operația dorită (de exemplu, obținerea sau modificarea datelor), ci și contribuie la modelul stateless al REST. Astfel, fiecare cerere este independentă, iar contextul interacțiunii este determinat exclusiv de reprezentările primite, nu de stări anterioare stocate pe server. Această arhitectură permite scalabilitate, deoarece fiecare nouă reprezentare îmbogățește experiența utilizatorului și deschide noi căi de navigare în cadrul aplicației.

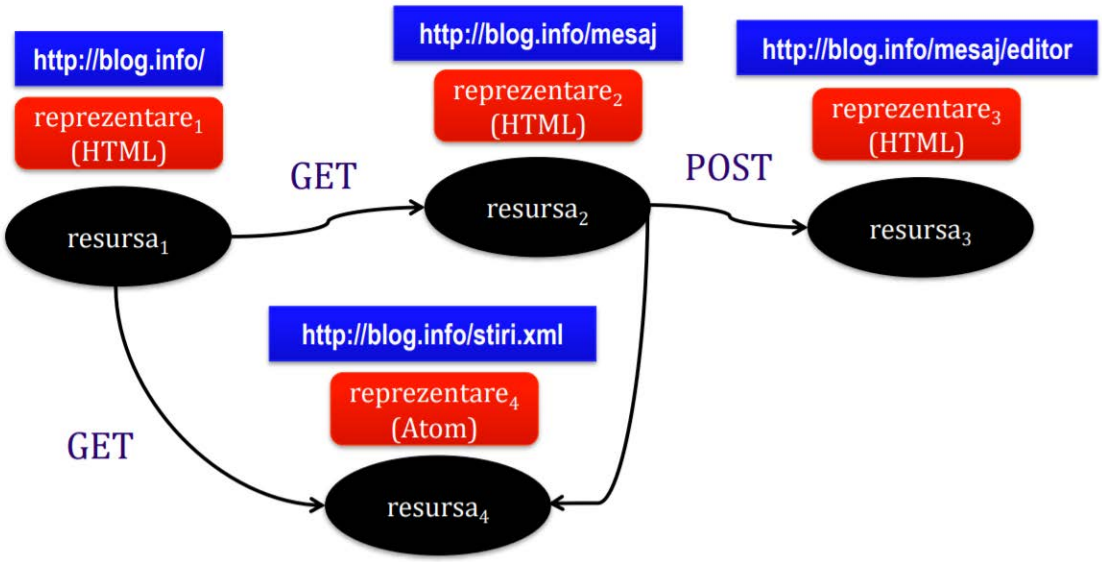



**Fig. 6.4.** Transferul de date în contextul REST.

Fig. 6.4 prezintă transferul de date în contextul REST, unde resursele sunt identificate prin URL-uri și interacționează prin metode HTTP precum GET și POST, fiecare având diferite reprezentări (HTML, Atom). Acest model se aseamănă cu un automat finit nedeterminist, deoarece permite tranziții între stări (resurse) bazate pe acțiuni specificate (cereri HTTP), fără

o unicitate strictă a următorului pas posibil. Fiecare pagină sau componentă a interfeței reprezintă o stare, iar tranzițiile între acestea sunt determinate de cererile HTTP și de legăturile integrate în resursele web.

Această analogie evidențiază natura dinamică și flexibilă a aplicațiilor web moderne, în care nu există un singur flux determinist de execuție, ci multiple posibilități de navigare. Fig. 6.4 subliniază, de asemenea, importanța proiectării sistemelor RESTful astfel încât să gestioneze eficient tranzițiile de stare și să ofere flexibilitate în navigare și accesarea resurselor.

În arhitectura REST, aplicația web este organizată ca un sistem stratificat, ceea ce înseamnă că funcționalitățile sunt împărțite în mai multe niveluri sau straturi, fiecare având roluri bine definite (Fig. 6.5). Această organizare permite ca fiecare strat să se ocupe de aspecte specifice ale procesării cererilor, de la autentificare și rutare, până la logica *business* și accesul la date. Fiecare strat funcționează ca o unitate independentă, astfel încât modificările aduse într-unul dintre ele nu afectează direct celelalte straturi.

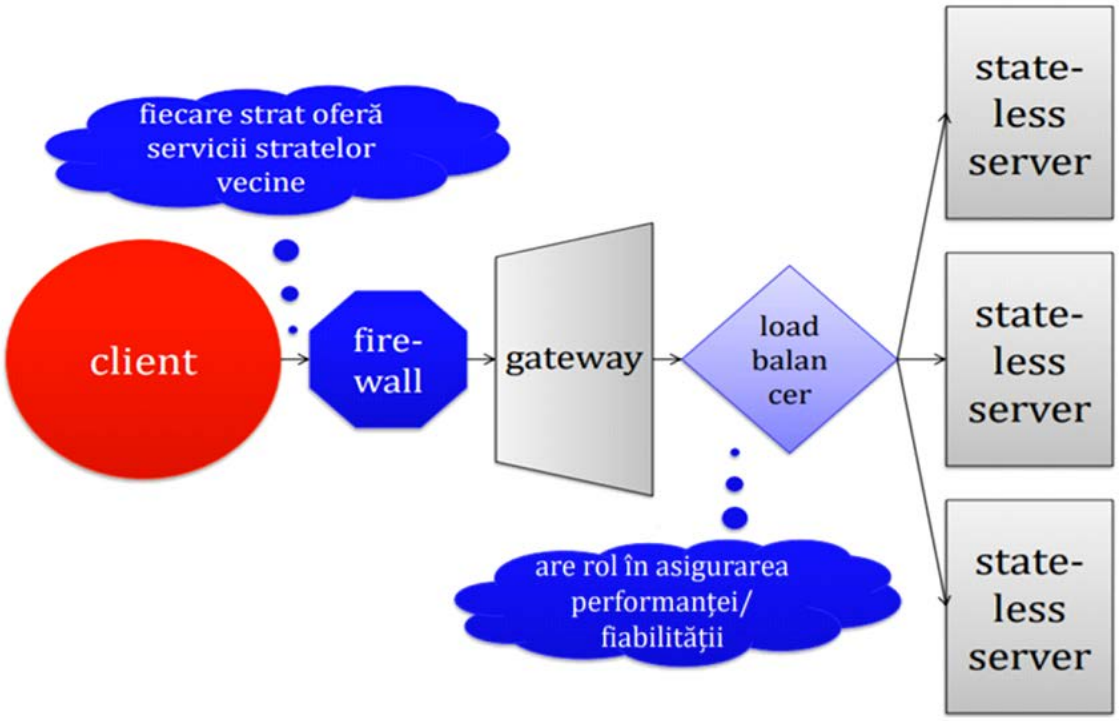



**Fig. 6.5.** Stratificarea aplicațiilor Web.

Unul dintre avantajele cheie ale sistemelor stratificate este cache-ul. Reprezentările resurselor pot fi stocate temporar, ceea ce reduce încărcarea asupra serverelor și îmbunătățește performanța generală a aplicației. Cache-ul permite reutilizarea datelor fără a fi nevoie de procesări repetate, optimizând astfel timpii de răspuns și eficiența în utilizarea resurselor.

Un alt principiu fundamental al acestei arhitecturii este încapsularea. Fiecare strat comunică doar cu straturile adiacente, fără a avea cunoștințe despre cele din afara lanțului imediat. Această izolare asigură că un strat nu poate influența straturile din afara intervalului său de interacțiune. Prin această metodă, straturile pot ascunde implementările interne și pot expune doar interfețele necesare, transformând sisteme tradiționale în „*black box*" – componente ale căror detalii interne sunt necunoscute celorlalte straturi.

## 6.3. API-uri RESTful

Un API (*Application Programming Interface*) reprezintă o interfață software care permite comunicarea între diferite părți de software, facilitând integrarea și reutilizarea funcționalităților. În esență, un API este un set de definiții, protocoale și convenții care stabilesc modul în care componentele software interacționează între ele. Acesta funcționează ca un contract între un furnizor de informații și un consumator, determinând ce conținut este cerut de către client și ce răspuns va fi furnizat de către server. De exemplu, un API pentru un serviciu meteorologic ar putea specifica că utilizatorul trebuie să furnizeze un cod poștal, iar răspunsul să conțină două părți: temperatura maximă și cea minimă.

Rolul API-urilor este acela de a acționa ca un mediator între utilizatori sau clienți și resursele disponibile. Prin intermediul unui API, dezvoltatorii pot comunica exact ce informații sau funcționalități doresc să acceseze, fără a fi nevoie să cunoască detaliile interne ale modului în care sunt stocate sau procesate aceste informații. Această abordare nu numai că simplifică procesul de dezvoltare, dar asigură și o consistentă expunere a funcționalităților, indiferent de modificările interne ce pot apărea în sistem. API-urile permit, astfel, partajarea de resurse și informații între diferite aplicații sau organizații, menținând în același timp securitatea, controlul accesului și autentificarea utilizatorilor.

Un alt avantaj major al API-urilor este că ele ascund complexitatea internă a sistemelor, prezentând utilizatorilor doar acele aspecte care sunt utile pentru a interacționa cu resursele disponibile. Astfel, un programator nu trebuie să se preocupe de modul în care sunt gestionate datele sau de detaliile implementării interne ale sistemului. API-ul oferă o interfață unificată și standardizată, care rămâne consecventă chiar dacă logica internă se modifică în timp. Această caracteristică contribuie la interoperabilitatea între diferite sisteme.

În contextul web, termenul API este adesea asociat cu API-urile REST, care permit comunicarea între computere conectate la rețea. Aceste API-uri facilitează accesul la resurse și servicii web prin intermediul unor protocoale bine cunoscute, cum ar fi HTTP, și formate de date standardizate, cum ar fi JSON sau XML. Astfel, API-urile web joacă un rol crucial în dezvoltarea aplicațiilor moderne, permițând integrarea ușoară a serviciilor între diferite platforme și asigurând un flux continuu de informații în mediile distribuite.

Un API REST reprezintă un tip de interfață software care adoptă stilul arhitectural REST pentru a expune și manipula resurse prin intermediul protocolului HTTP. Într-un astfel de API, resursele sunt identificate în mod unic prin URI-uri (*Uniform Resource Identifiers*), iar fiecare resursă poate avea una sau mai multe reprezentări (de exemplu, JSON, XML, HTML) care pot fi accesate, modificate sau șterse prin apeluri HTTP standard.

Într-un API REST, resursele sunt desemnate prin URL-uri descriptive și intuitive, care indică în mod clar entitatea la care se face referire. O convenție frecvent utilizată este folosirea substantivelor la plural pentru numele resurselor (de exemplu, `/students` pentru a desemna colecția de studenți). Această abordare asigură o structură coerentă a API-ului, facilitând înțelegerea și navigarea între diferitele endpoint-uri. De exemplu, pentru un API de gestionare a studenților, următoarele căi ar putea fi utilizate:

- `GET /students` – returnează lista tuturor studenților
- `GET /students/{id}` – returnează informațiile despre un student specific, identificate prin parametrul id
- `POST /students` – creează un nou student
- `PUT /students/{id}` – actualizează informațiile unui student existent
- `DELETE /students/{id}` – șterge un student

Așa cum se poate observa și în Fig. 6.6, operațiile efectuate asupra resurselor printr-un API REST au o corespondență cu operațiile de tip CRUD (*Create, Read, Update, Delete*) din bazele de date relaționale:

- **Create (INSERT)** – corespunde metodei HTTP POST
- **Read (SELECT)** – echivalent metodei HTTP GET
- **Update (UPDATE)** – pentru metodele HTTP PUT sau PATCH
- **Delete (DELETE)** – corespunde metodei HTTP DELETE

| Operația | SQL | HTTP |
|---|---|---|
| *Create* | INSERT | POST |
| *Read (Retrieve)* | SELECT | GET |
| *Update (Modify)* | UPDATE | PUT PATCH |
| *Delete (Destroy)* | DELETE | DELETE |

**Fig. 6.6.** Operații asupra resurselor REST.

Această paralelă facilitează înțelegerea rapidă a modului în care se efectuează manipularea datelor într-un API REST, deoarece dezvoltatorii software pot recunoaște ușor echivalențele cu operațiile deja familiare din bazele de date relaționale.

## 6.4. Organizarea resurselor

Într-un API REST, resursele pot fi structurate în moduri diferite pentru a oferi o experiență de navigare și manipulare cât mai intuitivă și eficientă. Două dintre cele mai frecvente abordări sunt organizarea sub formă de colecții și sub formă de depozite (*stores*). Această diferențiere le oferă dezvoltatorilor flexibilitate în ceea ce privește gestionarea și expunerea datelor.

O colecție reprezintă un catalog de resurse gestionat în principal de server. De exemplu, într-o aplicație care gestionează organizații și evenimente, putem avea rute precum:

```
GET /orgs/openstack/repos
GET /orgs/openstack/events
```

Aceste endpoint-uri descriu colecții de resurse (repositorii, evenimente) asociate unei entități, în acest caz *openstack*. Clienții pot propune modificări asupra colecției, cum ar fi adăugarea unui nou element sau ștergerea unuia existent, însă serverul decide cum să gestioneze efectiv cererile (de exemplu, poate valida datele și poate respinge cererile care nu îndeplinesc anumite criterii). Acest model este util atunci când logica de business necesită un control mai strict asupra resurselor și a modului în care acestea sunt organizate.

Depozitele (*stores*) se disting prin faptul că gestiunea resurselor este într-o mai mare măsură în sarcina clientului. Un exemplu de depozit ar putea fi un endpoint precum:

```
GET /users/openstack/repos?page=2&per_page=3
```

Aici, clientul specifică parametri de paginare (*page=2, per_page=3*), iar serverul returnează doar subsetul cerut. În plus față de paginare, un depozit poate oferi funcționalități de sortare, filtrare și alte opțiuni de personalizare, lăsând clientului controlul asupra modului în care sunt prezentate resursele. Astfel, serverul expune o structură de date mai flexibilă, iar clientul alege ce subset din resurse să fie afișat sau manipulat.

Indiferent dacă resursele sunt organizate în colecții sau depozite, formatul de răspuns către client poate varia. În funcție de cerințe sau de anteturile trimise de client, serverul poate returna resursele sub formă de CSV, JSON, XML, YAML sau alte formate. Această flexibilitate asigură interoperabilitate cu diverse aplicații și limbaje de programare, permițând integrarea facilă a serviciilor în ecosisteme software diferite.

Prin adoptarea unor strategii clare de organizare a resurselor, precum colecțiile și depozitele, un API RESTful devine mai ușor de înțeles și de utilizat, atât de către dezvoltatori, cât și de aplicațiile client care interacționează cu el.

URI-urile care desemnează resurse în cadrul unui API RESTful trebuie să fie simple, intuitive și ușor de înțeles pentru dezvoltatori și consumatori. În principiu, fiecare entitate este reprezentată de un substantiv, iar colecțiile de resurse sunt denumite la plural. Această convenție de denumire facilitează identificarea și accesarea resurselor, permițând o navigare coerentă în cadrul domeniului modelat.

Pentru a asigura unicitatea fiecărei entități dintr-o colecție, se folosesc identificatori unici. Astfel, resursele individuale pot fi accesate fie printr-un identificator semantic, cum ar fi un nume (*/students/mihai*), fie printr-un identificator numeric sau alt tip de ID abstract (*/students/69*). Aceste convenții permit dezvoltatorilor să aleagă metoda cea mai potrivită în funcție de contextul aplicației, asigurând totodată claritate și consistență în API.

Structura ierarhică a URL-urilor reflectă relațiile dintre resursele din domeniul respectiv. De exemplu, un URL de forma */users/{name}/collection/folders/{f_id} /instances/{i_id}* indică o ierarhie clară, unde fiecare segment reprezintă o entitate din structura organizațională a datelor.

Într-un API RESTful, metodele HTTP sunt folosite pentru a efectua operații asupra resurselor și a colecțiilor de resurse. Fiecare metodă semnifică o acțiune specifică, facilitând o separare clară a responsabilităților, așa cum arată și Fig. 6.7:

- **POST**: creează o resursă nouă în cadrul colecției specificate. De exemplu, *POST /students* adaugă un student nou la lista existentă. Dacă resursa deja există (ex. */students/69*), serverul va putea returna o eroare, întrucât resursa respectivă nu trebuie creată din nou.
- **GET**: obține sau listează resursele existente. *GET /students* returnează întreaga listă de studenți, în timp ce *GET /students/69* oferă detalii despre un student anume.
- **PUT**: actualizează o resursă deja existentă. Dacă nu există resursa specificată, serverul poate crea una nouă sau poate semnala eroare, în funcție de politica implementată. De exemplu, *PUT /students/69* poate modifica datele unui student deja existent.
- **DELETE**: elimină resursa sau colecția. *DELETE /students* șterge toți studenții, în timp ce *DELETE /students/69* șterge doar studentul cu ID-ul 69.

| resursa (URI) | POST (creează) | GET (accesează) | PUT (actualizează) | DELETE (șterge) |
|---|---|---|---|---|
| /students | creează un student nou | listează studenții existenți | actualizează un set de studenți | șterge toți studenții |
| /students/69 (un URL deja existent) | eroare 😣 | oferă date despre student | dacă există, actualizează, altfel eroare | șterge studentul respectiv |

**Fig. 6.7.** Operații asupra resurselor REST.

Pe lângă operațiile de bază, API-urile RESTful pot oferi mecanisme de paginare, sortare și filtrare, pentru a optimiza accesul la seturi mari de date. De exemplu, pentru paginare, se pot utiliza parametri de tip *page* și *per_page*:

`GET /students/projects?page=2&per_page=5`

Aici, *page=2* indică a doua pagină, iar *per_page=5* specifică numărul de rezultate pe pagină. În alte implementări, se pot folosi parametri precum *limit* și *offset*, unde *limit* restricționează numărul de elemente returnate, iar *offset* indică de unde să înceapă lista, permițând o gestionare eficientă a volumelor mari de date:

`GET /students/projects?limit=33&offset=54`

Filtrarea resurselor se poate face prin adăugarea unor parametri specifici în query string. De pildă, următoarea cerere returnează doar câmpurile name, age, year și email pentru fiecare student:

`GET /students?fields=name,age,year,email`

Un exemplu clar de API RESTful poate fi un serviciu destinat gestionării adreselor web favorite (*bookmark-uri*) și este prezentat în Fig. 6.8. Într-o astfel de aplicație, utilizatorii își pot salva pagini web, pot adăuga comentarii și pot atașa *tag-uri* pentru a organiza mai ușor conținutul. Principiile REST asigură o structură clară și intuitivă a endpoint-urilor, precum și un set de reguli pentru manipularea resurselor.

| Resursa | URL | Metoda | Reprezentare |
|---|---|---|---|
| *Bookmark* | /bookmarks/{hash} | GET | application/json |
| *Bookmark* | /bookmarks/{hash} | PUT | application/json |
| *Bookmark* | /bookmarks/{hash} | DELETE | |
| Lista de adrese | /bookmarks | GET | application/json |
| Lista de utilizatori | /users | GET | application/json |
| Lista de *tag*-uri | /tags | GET | application/json |
| Pagina principală | / | GET | application/json |

**Fig. 6.8.** Exemplu de organizarea al unui API RESTful.

Este esențial ca URI-urile care desemnează resursele să fie cât mai descriptive și să nu conțină verbe, ci doar substantive la plural (când ne referim la colecții). De exemplu:

- Lista de bookmark-uri: `GET /bookmarks`
- Un bookmark specific: `GET /bookmarks/{hash}`
- Lista de utilizatori: `GET /users`
- Lista de tag-uri: `GET /tags`

Acest mod de denumire indică în mod clar ce resursă este accesată sau manipulată, iar includerea identificatorilor unici, cum ar fi *{hash}* pentru un *bookmark*, permite gestionarea entităților individuale. De asemenea, se evită folosirea verbelor în calea URL, precum `/createBookmark` sau `/deleteBookmark`, operațiile fiind determinate exclusiv de metoda HTTP (POST, GET, PUT, DELETE).

HTTP GET și POST sunt cele mai comune metode utilizate pentru a interacționa cu resursele într-un API RESTful, fiecare având caracteristici distincte care le fac potrivite pentru diferite scenarii de utilizare. Diferențele dintre aceste metode sunt surprinse în Tabelul 6.1.

**Tabel 6.1.** Diferențele dintre metodele HTTP GET și POST.

| | HTTP GET | HTTP POST |
|---|---|---|
| Buton back/Reload | Inofensiv | Datele vor fi retrimise (browserul ar trebui să avertizeze utilizatorul că datele sunt pe cale să fie retrimise) |
| Bookmarked | Da | Nu |
| Cached | Da | Nu |
| Tip encoding | application/x-www-form-urlencoded | application/x-www-form-urlencoded sau multipart/form-data. Pentru date binare (fișiere) multipart encoding. |
| Istoric | Parametrii rămân în istoricul browserului | Parametrii nu rămân în istoricul browserului |
| Restrictions on data length | Da, la trimiterea datelor, metoda GET adaugă datele la adresa URL; iar lungimea unei adrese URL este limitată la maxim 2048 de caractere. | Fără restricții |
| Restricții privind lungimea datelor | Sunt permise doar caractere ASCII | Fără restricții. Pot fi transmise de asemenea fișiere |
| Securitate | GET este mai puțin sigur în comparație cu POST, deoarece datele trimise fac parte din URL. **Nu utilizați niciodată GET când trimiteți parole sau alte informații sensibile!** | POST este mai sigur decât GET, deoarece parametrii nu sunt stocați în istoricul browserului sau în log-urile serverului web. |
| Vizibilitate | Datele sunt vizibile pentru toată lumea în URL | Datele nu sunt afișate în URL |

HTTP GET este folosit pentru a solicita date de la server. Șirul de interogare, care constă din perechi nume/valoare, este trimis direct în adresa URL, ceea ce face ca solicitările GET să poată fi stocate în cache, să rămână în istoricul browserului și să fie marcate (bookmarkable). Totuși, GET are limitări semnificative: nu ar trebui folosit pentru transmiterea de date sensibile, deoarece informațiile sunt vizibile în URL, iar dimensiunea acestuia este limitată (de obicei, maxim 2048 de caractere).

Pe de altă parte, HTTP POST este utilizat pentru a trimite date către server cu scopul de a crea sau actualiza o resursă. Datele sunt incluse în corpul cererii HTTP, ceea ce le face să nu fie incluse în URL și le scuteste de limitările legate de lungimea URI-ului. În plus, solicitările POST nu sunt stocate în cache, nu rămân în istoricul browserului și nu pot fi marcate, ceea ce le face potrivite pentru transferul de informații sensibile și voluminoase. Această metodă este esențială pentru operațiuni care implică modificarea stării sistemului, cum ar fi trimiterea formularelor de date, încărcarea de fișiere sau actualizarea datelor dintr-o bază de date.

## 6.5. Utilizare API RESTful

Unul dintre motivele pentru care arhitectura REST este atât de răspândită în dezvoltarea aplicațiilor web și cloud îl reprezintă caracterul *stateless* (fără stare) al interacțiunilor. Fiecare cerere conține toate informațiile necesare pentru a fi procesată, fără ca serverul să mențină un istoric al sesiunii. În cazul unor erori, componentele pot fi redistribuite sau reîncercate cu ușurință, fără a afecta coerența sistemului.

Deoarece niciun element dintr-o tranzacție nu este salvat pe server, aplicațiile cloud se pot adapta rapid la fluctuațiile de trafic. Această flexibilitate face ca API-urile REST să fie preferate pentru aplicațiile web și mobile moderne, unde cerințele de disponibilitate și performanță sunt ridicate. Prin distribuirea cererilor pe mai multe servere și redirecționarea automată a cererilor în cazul unor probleme, sistemele bazate pe REST pot gestiona încărcări mari și variații bruște de trafic, menținând o experiență de utilizare stabilă și rapidă.

Cazuri de utilizare populare pentru API-urile RESTful

1. Mobilitate: aplicațiile mobile, precum Bolt sau Uber, utilizează API-uri REST pentru a accesa servicii de hărți, pentru a localiza vehicule și pentru a programa curse. Interacțiunea *stateless* facilitează răspunsuri rapide și menținerea coerenței datelor despre pozițiile și disponibilitatea șoferilor.
2. Servicii bancare: aplicațiile bancare sau *fintech* folosesc API-uri REST pentru a accesa informații despre conturile clienților și pentru a efectua tranzacții securizate.
3. Servicii de streaming: platforme precum Spotify, Netflix sau YouTube utilizează API-uri REST pentru a obține și gestiona conținutul media de pe serverele la distanță.
4. Rețelele de socializare: servicii precum X (fost Twitter) și Facebook pun la dispoziție API-uri REST pentru gestionarea postărilor, mesajelor și interacțiunilor dintre utilizatori. Aceste API-uri permit, de asemenea, integrarea cu aplicații terțe, oferind posibilitatea de a partaja conținut sau de a afișa fluxuri de știri într-un mod unificat.

Tratarea erorilor este un aspect critic în proiectarea API-urilor RESTful, deoarece permite clienților să înțeleagă clar natura problemelor care apar în timpul interacțiunii cu serverul. Pentru a asigura o comunicare eficientă, se folosesc codurile de stare HTTP, conform standardelor detaliate pe *www.httpstatuses.com*. Aceste coduri oferă informații esențiale despre succesul sau eșecul unei cereri – de exemplu, codul 400 semnalează o cerere greșită, 404 indică faptul că resursa solicitată nu a fost găsită, iar 500 semnalează o eroare internă a serverului.

Pe lângă codul de stare, este esențial ca mesajele de eroare returnate să includă informații utile și descriptive. Un mesaj de eroare bine structurat nu doar că semnalează problema, dar oferă și detalii suplimentare despre cauza erorii și, dacă este cazul, sugestii pentru remediere. De exemplu, pentru codul de stare HTTP 404, GitHub furnizează acest mesaj în format JSON:

```
{
```

```
 "message": "Not Found",
 "documentation_url": "https//developer.github.com/v3"
}
```

Este recomandat ca tratamentul erorilor să fie implementat într-un mod consistent și standardizat, astfel încât toate componentele API-ului să ofere feedback clar și util. Acest lucru îmbunătățește experiența dezvoltatorului, deoarece permite identificarea rapidă a problemelor și facilitează remedierea lor.

## 6.6. Avantaje și Provocări

API-urile REST au câștigat o popularitate semnificativă datorită simplității și clarității cu care pot fi proiectate și implementate. Un avantaj major al API-urilor RESTful este independența de platformă. Deoarece acestea se bazează pe standarde deschise, cum ar fi HTTP și formate de date universale, dezvoltatorii pot crea API-uri REST în aproape orice limbaj de programare. Această caracteristică permite funcționarea API-urilor pe o varietate de dispozitive – de la browsere web și aplicații mobile, până la dispozitive IoT – oferind astfel o interoperabilitate ridicată între sisteme diferite.

Flexibilitatea este, de asemenea, un beneficiu esențial, deoarece API-urile RESTful acceptă multiple formate de date. Dezvoltatorii pot alege formatul cel mai potrivit pentru nevoile specifice ale aplicației, adaptând reprezentările resurselor în funcție de context și preferințele clientului. În plus, natura *stateless* a API-urilor REST le face extrem de scalabile, permițând distribuirea cererilor către multiple instanțe ale serverului pentru a gestiona volume mari de trafic.

Un alt aspect valoros este capacitatea de cacheabilitate a API-urilor RESTful. Prin stocarea temporară a datelor pe client sau în rețea, se pot reduce semnificativ timpii de răspuns și se optimizează resursele serverului, eliminând astfel necesitatea de a face apeluri repetate către aceeași resursă.

Pe lângă aceste avantaje, API-urile RESTful se remarcă și prin securitate. Mecanisme precum autentificarea prin OAuth și criptarea datelor prin SSL/TLS asigură că schimburile de informații se realizează într-un mediu securizat, protejând astfel datele sensibile și gestionând accesul la resurse în mod controlat. În plus, utilizarea corectă a versiunilor API permite dezvoltatorilor să evolueze sistemul în timp, adăugând noi funcționalități fără a afecta compatibilitatea cu versiunile anterioare, ceea ce reprezintă un aspect crucial în menținerea stabilității și fiabilității aplicațiilor.

API-urile RESTful aduc numeroase beneficii, însă prezintă și anumite provocări care pot afecta atât dezvoltarea, cât și întreținerea acestora. Una dintre dificultăți este consecvența endpoint-urilor: pentru a fi coerente, căile URL trebuie să urmeze standardele web comune, însă acest lucru poate deveni complex în proiectele de amploare, unde gestionarea convențiilor de denumire și ierarhizarea resurselor necesită o atenție constantă. De exemplu,

modificările necontrolate ale structurii endpoint-urilor pot conduce la inconsistențe și probleme de compatibilitate între diverse versiuni ale API-ului.

De asemenea, în contextul REST, se pot întâlni probleme legate de timpi lungi de răspuns și volumul excesiv de date returnate. În timp, numărul de resurse sau complexitatea datelor poate crește, ceea ce duce la încărcare suplimentară pe server și la timpi de răspuns crescuți, afectând performanța sistemului.

Alte provocări includ definirea clară a căilor de navigare și a locațiilor pentru parametrii de intrare, deoarece REST se bazează pe URI-uri descriptive. Determinarea spațiilor URI potrivite poate fi dificilă, în special atunci când se gestionează un număr mare de resurse cu relații complexe. În plus, cererile pot conține mai multe date și metadate decât sunt strict necesare, sau pot fi necesare mai multe solicitări pentru a obține toate informațiile dorite, ceea ce poate complica integrarea și optimizarea fluxurilor de date.

Gestionarea codurilor de eroare și a mesajelor de răspuns reprezintă o altă provocare importantă. Deși se utilizează coduri de stare HTTP standard, interpretarea lor corectă și furnizarea de mesaje utile care să descrie natura exactă a erorilor poate fi dificilă, necesitând o implementare atentă a mecanismelor de tratare a erorilor. În plus, testarea API-urilor RESTful este un proces care poate deveni complex, implicând configurarea inițială a mediilor de testare, actualizarea constantă a schemelor, gestionarea diverselor combinații de parametri și secvențierea apelurilor API. Instrumente precum cURL pot fi utile, însă testarea completă a unui API necesită adesea o abordare detaliată și metodică pentru a putea garanta funcționalitatea și performanța așteptată.

# 7. Capitolul 7. Securitatea Serviciilor Web

Securitatea serviciilor web reprezintă un aspect esențial al dezvoltării aplicațiilor moderne, având în vedere volumul mare de date și interacțiunile distribuite pe internet. Într-o eră în care atacurile cibernetice sunt tot mai sofisticate, protejarea integrității, confidențialității și disponibilității datelor devine o prioritate absolută. Acest capitol explorează mecanismele și politicile de securitate implementate în serviciile web, subliniind importanța măsurilor de protecție atât la nivelul clientului, cât și al serverului.

## 7.1. Securitate Web

Securitatea web pornește de la browsere, care acționează ca prima linie de apărare împotriva atacurilor cibernetice și asigură protecția datelor utilizatorilor. Politica de aceeași origine este o măsură fundamentală de securitate implementată de browserele moderne, menită să împiedice documentele și scripturile provenite dintr-o anumită origine să acceseze resurse dintr-o altă origine fără permisiune explicită (Fig 7.1). Astfel, deși un browser poate încărca și afișa resurse din mai multe site-uri simultan, interacțiunile directe între aceste resurse – cum ar fi citirea datelor din Document Object Model (DOM), accesul la cookie-uri sau la stocarea locală – sunt strict limitate la originea sursă.


**Fig. 7.1.** Politica de aceeași origine a browserelor.

Această restricție se aplică prin verificarea componentelor esențiale ale unei adrese URL: protocolul, domeniul și portul. De exemplu, un script care rulează pe o pagină provenită de la *https://example.com* nu poate interacționa cu resursele de la *https://altceva.com*, deoarece aceste două origini sunt diferite. Această barieră previne atacuri de tip cross-site scripting (XSS) și cross-site request forgery (CSRF), protejând astfel datele și confidențialitatea utilizatorilor.

În contextul dezvoltării web moderne, există și mecanisme precum CORS (*Cross-Origin Resource Sharing*), care permit, în condiții controlate, relaxarea acestei politici. CORS oferă serverelor posibilitatea de a specifica ce origini externe sunt autorizate să acceseze resursele lor, menținând totodată securitatea aplicației. Astfel, politica de aceeași origine rămâne un pilon central în securizarea serviciilor web, asigurând integritatea datelor și protejând utilizatorii împotriva accesului neautorizat.

Pe partea de server, securitatea este esențială deoarece atacatorii pot utiliza clienți HTTP arbitrar, cum ar fi comanda *curl*, pentru a trimite cereri malitioase către server. De exemplu, un atacator ar putea lansa o cerere POST cu un payload JSON modificat pentru a obține privilegii administrative sau pentru a exploata vulnerabilități ale aplicației:

```
curl
-d '{"user":"Alice", "permission":"admin"}'
-H "Content-Type: application/json"
-X POST http://example.com/data
```

Pentru a contracara astfel de atacuri, este necesară validarea strictă a datelor de intrare, autentificarea și autorizarea corespunzătoare, precum și implementarea unor mecanisme de rate limiting și monitorizare continuă a traficului. În plus, serverul trebuie să fie capabil să detecteze și să blocheze cererile suspecte, utilizând filtre și reguli de securitate (de exemplu, *Web Application Firewalls* – WAF) pentru a preveni atacurile de tip *injection* sau *denial-of-service*. Implementarea criptării la nivel de transport, folosind protocoale precum TLS/SSL, contribuie la protejarea datelor în tranzit, asigurând că informațiile sensibile nu sunt interceptate. Astfel, securitatea server-side se bazează pe o combinație de măsuri tehnice și politici de acces riguroase, esențiale pentru menținerea integrității serviciilor web.

Pe partea de client, securitatea se referă la protejarea utilizatorilor în timp ce interacționează cu aplicația web local. Aceasta include prevenirea atacurilor precum furtul parolelor și tehnici de *social engineering*, unde atacatorii încearcă să manipuleze utilizatorii pentru a obține acces la informații sensibile. Măsurile de protecție pe client pot include implementarea de autentificare multifactorială, criptarea datelor locale și folosirea de tehnologii de sandboxing în browsere pentru a limita impactul eventualelor vulnerabilități.

De asemenea, este important ca aplicația client să afișeze mesaje de avertizare clare și să ofere informații despre securitate pentru a educa utilizatorii cu privire la riscurile potențiale. Implementarea unor politici de securitate în browser, cum ar fi *Content Security Policy* (CSP),

poate preveni atacurile de tip *cross-site scripting* (XSS) și alte exploatări bazate pe injecții de cod. În acest mod, securitatea client-side nu doar protejează datele utilizatorilor, ci contribuie la crearea unei experiențe web sigure și fiabile, completând astfel măsurile de protecție implementate la nivelul serverului.

## 7.2. Cookie și autoritate ambientală

Cookie-urile reprezintă un mecanism esențial de gestionare a sesiunilor în aplicațiile web, fiind folosite de servere pentru a stoca și menține un set de date legate de sesiunea de navigare a utilizatorului curent. Prin intermediul cookie-urilor, serverul poate păstra informații precum identificatorul de sesiune, preferințele utilizatorului sau starea coșului de cumpărături, ceea ce permite personalizarea experienței și facilitarea autentificării și autorizării. Acestea sunt trimise de la server către browser și, ulterior, incluse automat în cererile ulterioare către același server, astfel încât informațiile relevante să fie disponibile pe parcursul întregii sesiuni.

Utilizarea cookie-urilor se extinde dincolo de gestionarea sesiunilor; ele sunt folosite și pentru scopuri precum urmărirea comportamentului utilizatorilor (*user tracking*) și analizarea modului în care aceștia interacționează cu site-ul. Această funcționalitate este valorificată de companii pentru a oferi conținut personalizat, pentru a optimiza strategiile de marketing și pentru a analiza performanța site-urilor web. De asemenea, cookie-urile facilitează implementarea unor funcționalități precum coșurile de cumpărături, permițând stocarea temporară a produselor selectate, chiar dacă utilizatorul navighează pe mai multe pagini.


**Fig. 7.2.** Structură cookie.

Fig. 7.2 ilustrează formatul antetului *Set-Cookie*, pe care un server îl trimite către client pentru a inițializa sau actualiza un cookie. Structura sa cuprinde mai multe elemente:

1. Numele antetului (*Header Name*): *Set-Cookie* – indică browserului că acesta este un antet special, destinat gestionării cookie-urilor.
2. Numele cookie-ului (*Cookie Name*): în exemplu theme - stabilește identificatorul sub care cookie-ul va fi stocat și trimis ulterior în cereri.
3. Valoarea cookie-ului (*Cookie Value*): în exemplu dark - specifică efectiv conținutul asociat cookie-ului.
4. Atributele cookie-ului (*Attr. Name/Value*): după valoarea cookie-ului, pot fi incluse atribute suplimentare pentru configurarea comportamentului acestuia.

Astfel, antetul *Set-Cookie* oferă browserului instrucțiuni clare despre numele, valoarea și parametrii unui cookie, asigurând o gestionare precisă a sesiunilor și preferințelor utilizatorilor.

Cookie-urile oferă mecanisme suplimentare pentru controlul duratei de viață și al domeniilor de aplicabilitate, prin intermediul unor proprietăți precum *Expires*, *Path* și *Domain*. Aceste proprietăți determină comportamentul exact al cookie-ului și modul în care este gestionat de browser, având totodată implicații de securitate în contextul politicii de aceeași origine (*Same Origin Policy*):

1. Proprietatea *Expires* stabilește data de expirare a cookie-ului. Dacă nu este specificată o dată anume, cookie-ul va rămâne valabil pe toată durata sesiunii curente de navigare, fiind șters automat când utilizatorul închide browserul. Pentru a forța ștergerea unui cookie, se poate trimite o dată în trecut, de exemplu:
   `Set-Cookie: key=; Expires=Thu, 01 Jan 1970 00:00:00 GMT`
   Această abordare determină browserul să șteargă imediat cookie-ul respectiv.
2. *Path* delimitează prefixul de cale pentru care cookie-ul este trimis automat de browser în cereri. De exemplu, dacă setăm `Path=/admin`, cookie-ul va fi trimis doar atunci când utilizatorul accesează resurse din calea `/admin` a site-ului. Această delimitare ajută la organizarea accesului la cookie-uri pe secțiuni distincte ale aplicației. Totuși, nu trebuie folosită ca metodă de securitate pentru a „ascunde" cookie-urile de alte pagini din aceeași origine, deoarece, din punct de vedere tehnic, politica *Same Origin* nu blochează accesul dacă Path-ul este diferit.
3. Proprietatea *Domain* extinde aria de valabilitate a cookie-ului la un domeniu mai larg. De exemplu, dacă site-ul este *sub.example.com*, setarea *Domain=.example.com* permite cookie-ului să fie accesibil și de către alte subdomenii (cum ar fi *shop.example.com*). Această funcționalitate poate fi utilă pentru aplicații ce partajează autentificarea sau preferințele între subdomenii. Totuși, ea trebuie folosită cu precauție, deoarece o origine poate seta cookie-uri pentru altă origine din același domeniu, ridicând potențiale probleme de securitate.
4. Implicarea în politica de aceeași origine (*Same Origin Policy*). Deși *Path* și *Domain* oferă flexibilitate, ele pot ridica și probleme de securitate dacă nu sunt folosite corect. Politica de aceeași origine este concepută să limiteze modul în care resursele dintr-un site pot accesa conținutul altui site sau subdomeniu. Totuși, setarea Domain poate face ca cookie-urile să fie partajate între site-uri care se află sub același domeniu părinte, iar Path nu oferă o protecție reală împotriva accesului nedorit la cookie-urile din aceeași origine.

Prin urmare, gestionarea corectă a proprietăților *Expires*, *Path* și *Domain* este esențială pentru menținerea securității și a confidențialității utilizatorilor. Dezvoltatorii trebuie să fie conștienți de faptul că aceste setări pot avea implicații majore asupra modului în care cookie-

urile sunt partajate și expiră, iar alegerea incorectă a proprietăților poate expune aplicația la vulnerabilități de securitate.

Controlul accesului reprezintă un pilon esențial în securitatea serviciilor web, asigurându-se că doar utilizatorii autorizați pot vizualiza resursele sau pot întreprinde anumite acțiuni. În acest context, autoritatea ambientală se referă la mecanismele de control al accesului bazate pe proprietăți globale și persistente ale solicitantului, cum ar fi cookie-urile sau adresa IP, care permit recunoașterea automată a utilizatorului fără a necesita o autorizare explicită pentru fiecare operațiune. Această metodă se diferențiază de autorizarea explicită, care este validă doar pentru o anumită acțiune și necesită o verificare suplimentară la fiecare interacțiune.

Pe web, există patru tipuri principale de autoritate ambientală. Cookie-urile reprezintă cea mai comună și versatilă metodă, permițând stocarea informațiilor de sesiune și a preferințelor utilizatorului direct pe dispozitivul client. Verificarea IP este folosită, de exemplu, de instituții precum Stanford pentru a controla accesul la resurse speciale, bazându-se pe adresa IP a solicitantului. De asemenea, se mai utilizează autentificarea HTTP încorporată, deși aceasta este rar întâlnită datorită limitărilor sale, și certificatele de client, o metodă robustă dar administrativ mai complexă, utilizată ocazional pentru a asigura o verificare puternică a identității.

Fiecare dintre aceste metode prezintă avantaje și dezavantaje specifice. Cookie-urile, deși ușor de implementat și utilizate pentru menținerea sesiunilor persistente, pot fi expuse riscului de manipulare sau furt. Verificarea IP oferă un nivel de securitate ridicat, dar poate fi ineficientă în mediile cu adrese IP dinamice. Autentificarea HTTP și certificatele de client oferă un nivel suplimentar de protecție, însă implică o administrare mai complexă și sunt folosite mai rar în practică.

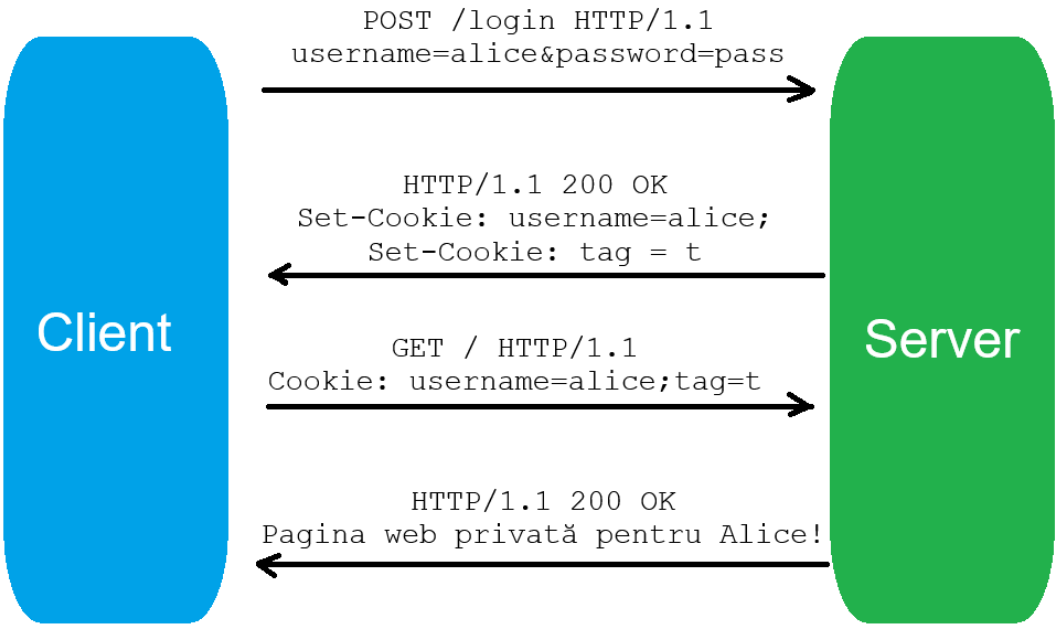



**Fig. 7.3.** Exemplu – autentificare folosind cookie.

Fig. 7.3 ilustrează fluxul unei autentificări bazate pe cookie-uri, pornind de la o cerere HTTP de tip POST către endpoint-ul */login*. În cadrul acestei cereri, clientul trimite credențialele (nume de utilizator și parolă) sub formă de parametri. Serverul validează informațiile primite și, dacă autentificarea reușește, răspunde cu un antet *Set-Cookie* ce conține datele de sesiune (de exemplu, *username=alice* și un tag sau token de autentificare).

După primirea antetului Set-Cookie, browserul (sau orice alt tip de client HTTP) stochează cookie-urile și le atașează automat la cererile ulterioare, inclusiv la un simplu GET /. Datorită acestui mecanism, serverul recunoaște utilizatorul autentificat (în exemplu, „Alice") și returnează conținut personalizat sau pagini private. Astfel, autentificarea bazată pe cookie-uri simplifică menținerea sesiunii, fără a solicita reintroducerea credențialelor pentru fiecare cerere, ceea ce îmbunătățește semnificativ experiența de utilizare.

Autoritatea ambientală poate crea vulnerabilități în situațiile în care un site rău intenționat (*http://www.attacker.com*) încorporează conținut care face referire la resurse de pe un alt domeniu (de exemplu, *http://www.bank.com*). De exemplu, un cod HTML inserat pe *attacker.com* include un tag img care încarcă resurse de la *bank.com*:

```
<h1>Welcome to your account!</h1>
<img src='https://bank.com/avatar.png' />
```

Browserul, respectând regulile standard de cookie, trimite automat cookie-urile asociate domeniului *bank.com*. Astfel, *attacker.com* poate afișa date reale ale utilizatorului sau poate inițiа acțiuni la *bank.com* în numele acestuia, deoarece cererile poartă cookie-urile de sesiune valabile. Această situație descrie o potențială vulnerabilitate de tip *Cross-Site Request Forgery* (CSRF). Într-un astfel de atac, site-ul rău intenționat folosește autoritatea ambientală a utilizatorului, reprezentată de cookie-urile sale, pentru a efectua operațiuni nedorite. De exemplu, atacatorul poate solicita modificarea setărilor contului sau efectuarea unor tranzacții, iar serverul *bank.com* va considera cererea legitimă, deoarece este însoțită de cookie-urile valide. Pentru a atenua aceste probleme, dezvoltatorii pot implementa mecanisme anti-CSRF, pot configura corect politica *SameSite* a cookie-urilor și pot evita acceptarea cererilor neautentificate prin metode de validare suplimentare (de exemplu, token-uri unice sau anteturi personalizate). Aceste contramăsuri protejează aplicațiile împotriva atacurilor care exploatează comportamentul implicit al browserului și al cookie-urilor.

## 7.3. Atacuri de sesiune

Atacurile de sesiune reprezintă un ansamblu de tehnici folosite de atacatori pentru a exploata vulnerabilitățile din gestionarea sesiunilor utilizatorilor în aplicațiile web. Aceste

atacuri vizează, de exemplu, furtul sau manipularea cookie-urilor și a token-urilor de sesiune, astfel încât un atacator să preia controlul asupra unei sesiuni active, acționând în numele utilizatorului legitim. Astfel de atacuri pot conduce la accesul neautorizat la informații sensibile, la efectuarea de tranzacții frauduloase sau la compromiterea completă a sistemului. Protejarea sesiunilor prin implementarea măsurilor de securitate adecvate este esențială pentru menținerea integrității și confidențialității datelor în mediile web moderne.

### 7.3.1. Session Hijacking

Transmisia cookie-urilor prin HTTP necriptat expune datele de sesiune la interceptare și poate conduce la atacuri de tip *session hijacking*. Atacatorul poate intercepta cookie-ul utilizatorului și îl poate folosi pentru a se prezenta ca și cum ar fi victima, păcălind astfel serverul și obținând acces neautorizat la contul utilizatorului. Acest lucru poate conduce la furtul informațiilor personale, manipularea datelor și efectuarea de tranzacții frauduloase.

Pentru a preveni astfel de atacuri, este esențială utilizarea HTTPS. HTTPS criptează datele transmise între client și server, asigurând că cookie-urile și alte informații sensibile nu pot fi interceptate sau modificate de terți. În plus, HTTPS asigură și autentificarea serverului, permițând utilizatorilor să verifice că se conectează la sursa corectă și de încredere. Astfel, adoptarea HTTPS devine o practică indispensabilă pentru orice aplicație web care gestionează date personale sau operațiuni sensibile, oferind un nivel ridicat de protecție împotriva deturnărilor de sesiune.

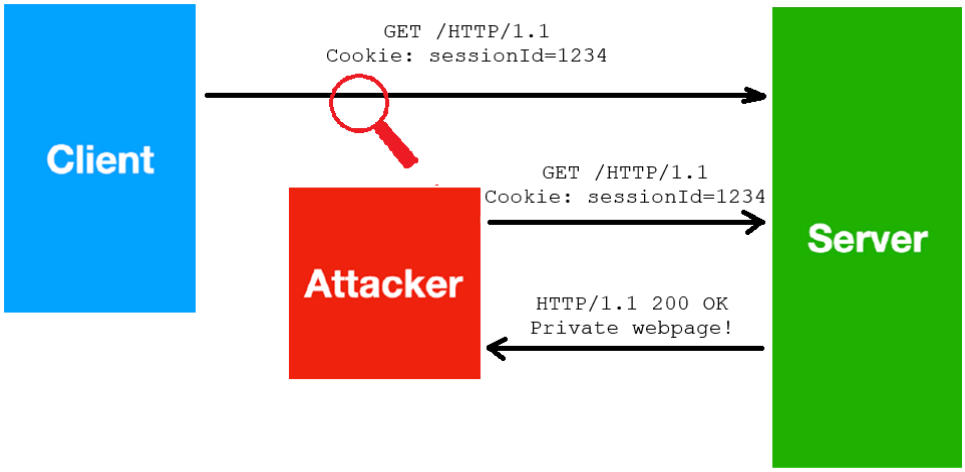



**Fig. 7.4.** Exemplu – atac de tip *session hijacking*.

Fig. 7.4 prezintă modul în care un atac de tip *session hijacking* poate avea loc atunci când cookie-urile sunt transmise printr-o conexiune necriptată (HTTP). Clientul inițiază o cerere către server, incluzând un cookie de sesiune (ex.: *sessionId=1234*). Atacatorul, plasat undeva între client și server, poate intercepta traficul necriptat și prelua cookie-ul, deoarece acesta este transmis în text clar.

Odată ce atacatorul obține cookie-ul valid (*sessionId=1234*), îl poate utiliza pentru a trimite propriile cereri către server, prezentându-se drept utilizatorul autentic. Serverul, care nu distinge între cererile legitime și cele interceptate, va răspunde atacatorului ca și cum acesta ar fi clientul original, oferindu-i acces la resurse private sau date sensibile. Acest scenariu evidențiază necesitatea folosirii unei conexiuni securizate (HTTPS), care criptează traficul și previne interceptarea cookie-urilor de către terți.

### 7.3.2. Cross-Site Request Forgery (CSRF)

*Cross-Site Request Forgery* (CSRF) este un tip de atac în care un utilizator legitim este indus, fără să știe, să trimită cereri nedorite către o aplicație web în care este deja autentificat. Într-un astfel de scenariu, atacatorul nu are nevoie să intercepteze sau să citească răspunsurile de la server, ci se bazează pe faptul că browserul va atașa automat cookie-urile sau token-urile de autentificare pentru fiecare cerere către domeniul legitim.

Dacă utilizatorul final deține drepturi normale, atacul CSRF poate forța acest utilizator să efectueze acțiuni precum transferul de fonduri, modificarea adresei de e-mail sau trimiterea unui mesaj, fără știrea sa. În cazul în care victima este un administrator, consecințele pot fi mult mai grave: atacatorul ar putea iniția cereri pentru a crea noi conturi cu privilegii înalte, a modifica setările de securitate sau chiar a executa comenzi direct pe server, sub acoperirea sesiunii de administrator.

Particularitatea atacului CSRF constă în faptul că browserul victimei transmite automat datele de autentificare (cookie-uri, token-uri de sesiune etc.) atunci când accesează resurse de pe domeniul legitimat, chiar dacă cererea provine de pe un site rău intenționat. Astfel, protejarea aplicațiilor web împotriva CSRF necesită implementarea unor contramăsuri suplimentare, precum token-uri anti-CSRF, setarea corespunzătoare a cookie-urilor (de ex., *SameSite*), validarea referer-ului sau a antetelor personalizate și monitorizarea atentă a cererilor suspecte.

Inspectarea antetului *HTTP Referer* este o metodă eficientă de protecție împotriva atacurilor CSRF. Prin verificarea valorii antetului *Referer*, serverul poate determina dacă o cerere provine dintr-o origine de încredere, adică de la un domeniu inclus pe lista permisă. Orice solicitare care nu îndeplinește această condiție este respinsă, prevenind astfel ca un atacator să exploateze sesiunea utilizatorului prin trimiterea unor cereri nedorite de pe un site neautorizat.

Această tehnică de protecție funcționează bine în majoritatea cazurilor, deoarece browserele includ automat antetul *Referer* în cererile HTTP, permițând serverului să verifice sursa solicitării. Totuși, este important de menționat că acest antet poate fi uneori omis sau modificat, motiv pentru care, în practică, se recomandă combinarea acestei metode cu alte măsuri de securitate, cum ar fi utilizarea token-urilor anti-CSRF și setarea corectă a politicii

*SameSite* pentru cookie-uri. Aceste măsuri complementare asigură o protecție robustă împotriva atacurilor CSRF, menținând integritatea și confidențialitatea sesiunilor utilizatorilor.

Fig. 7.5 ilustrează un atac de tip CSRF, în care un atacator încearcă să inducă serverul *bank.com* în eroare pentru a executa o acțiune neautorizată în numele utilizatorului autentic. Un utilizator autentic de pe *bank.com* se conectează, iar serverul îi trimite un cookie de sesiune (*sessionId=1234*). Ulterior, utilizatorul solicită o resursă (*avatar.png*), iar serverul răspunde corect, deoarece cererea provine de pe *bank.com* și include un antet Referer valid. Atacatorul de pe *attacker.com* încearcă să trimită o cerere către *bank.com*, reutilizând cookie-ul de sesiune al utilizatorului autentic. Dacă serverul nu verifică sursa cererii (prin *Referer* sau un CSRF token), atacatorul ar putea accesa sau modifica resursele utilizatorului fără consimțământul acestuia.

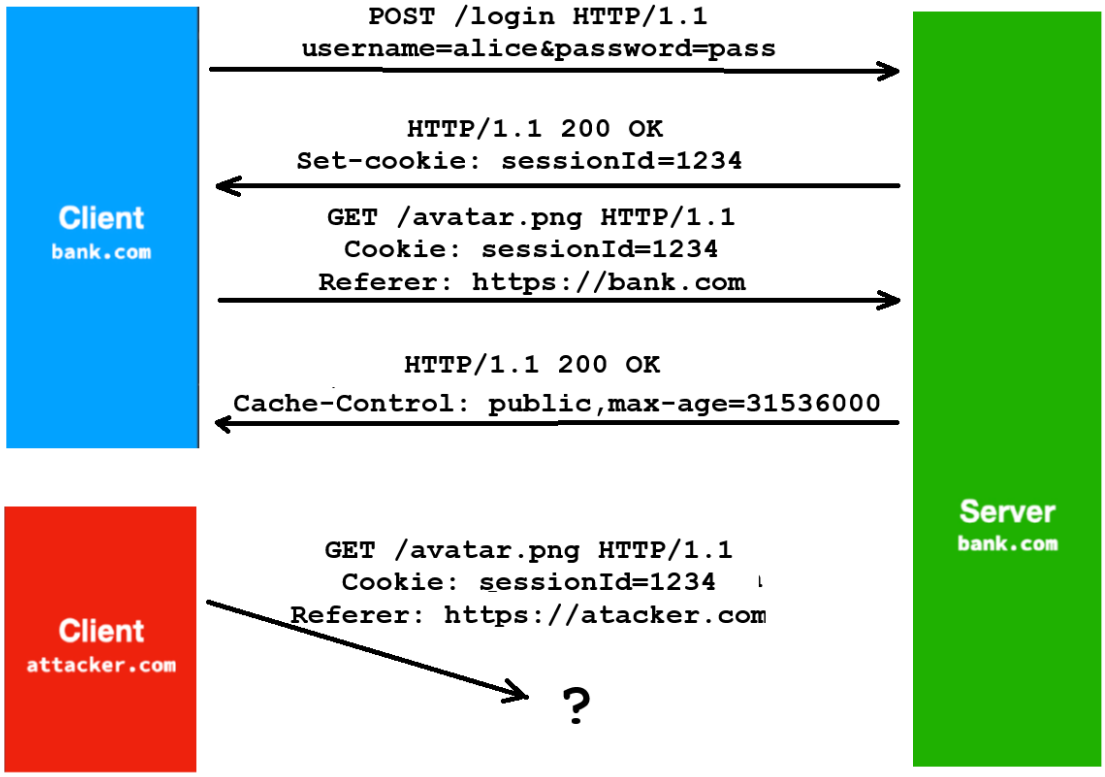



**Fig. 7.5.** Exemplu – atac de tip *CSRF*

### 7.3.3. Cross-Site Scriptiong (XSS)

*Cross-site scripting* (XSS) este o vulnerabilitate de tip „injectare de cod" care apare atunci când datele furnizate de utilizator, care nu sunt validate sau sanitize corespunzător, sunt integrate în mod neașteptat în paginile web ca parte a codului HTML. În acest context, codul injectat este de obicei JavaScript, iar dacă este executat în browserul altui utilizator, poate duce la furtul de informații, manipularea interfeței sau chiar preluarea controlului asupra contului acestuia.

Problema apare deoarece orice cod care combină o comandă cu datele nefiltrate ale utilizatorului devine susceptibil la injectare. Spre deosebire de injecția SQL, unde codul injectat constă în comenzi SQL suplimentare ce pot altera sau compromite baza de date, XSS se referă la injectarea de scripturi JavaScript în documente HTML. Atacurile XSS pot fi exploatate în diverse moduri, cum ar fi furtul cookie-urilor, redirecționarea către site-uri malitioase sau modificarea conținutului paginii.

Pentru a preveni astfel de atacuri, este esențială validarea și filtrarea adecvată a datelor introduse de utilizatori, precum și implementarea unor politici stricte de securitate la nivelul aplicațiilor web. Tehnici precum codificarea corectă a datelor, utilizarea *Content Security Policy* (CSP) și practicile de sanitizare a intrărilor sunt fundamentale pentru a proteja aplicațiile împotriva exploatării vulnerabilităților XSS.

Protecția împotriva injectării de cod este esențială pentru securizarea aplicațiilor web, deoarece vulnerabilitățile de acest tip apar atunci când datele furnizate de utilizatori, care nu sunt verificate corespunzător, sunt interpretate ca și cod executabil. Sursa acestor vulnerabilități poate proveni din diverse canale, cum ar fi parametrii de interogare, formularele, anteturile, cookie-urile sau chiar încărcarea de fișiere. Astfel, dacă un atacator reușește să injecteze un tag periculos, cum ar fi *<script>*, acesta poate introduce cod JavaScript malitios care va rula în contextul paginii web, permițând furtul de informații sensibile sau preluarea controlului asupra interacțiunii cu utilizatorul.

Pentru a preveni astfel de atacuri, este vital ca toate datele introduse de utilizator să fie verificate și validate înainte de a fi integrate în template-urile HTML. Această măsură ajută la eliminarea oricăror elemente care ar putea fi interpretate ca și cod, asigurându-se că doar informațiile sigure și intenționate sunt preluate și procesate. Măsuri suplimentare, cum ar fi implementarea unor politici stricte de filtrare a tag-urilor și utilizarea unor biblioteci de procesare a datelor introduse de utilizator, contribuie la reducerea riscului de exploatare a vulnerabilităților de tip XSS, protejând astfel datele private și integritatea aplicației web.

## 7.4. Autorizare și autentificare

### 7.4.1. Autorizare

Autorizarea reprezintă un pilon esențial în securizarea accesului la resursele unei aplicații web, asigurându-se că doar utilizatorii sau aplicațiile cu drepturi corespunzătoare pot interacționa cu datele sensibile. Procesul de autorizare începe cu obținerea unei chei de acces, care este generată după autentificarea inițială a utilizatorului sau a aplicației. Această cheie, sub formă de token sau cod de sesiune, servește drept dovadă a identității și este folosită ulterior pentru a valida fiecare cerere către server, protejând astfel sistemul împotriva accesului neautorizat.

Odată obținută cheia de acces, următoarea etapă implică combinarea proceselor de autentificare și autorizare. În acest context, autentificarea se referă la verificarea identității utilizatorului, iar autorizarea determină ce acțiuni sau resurse sunt permise pentru acel utilizator. De exemplu, un utilizator poate avea drepturi de vizualizare a datelor, în timp ce un administrator poate avea acces extins, inclusiv la modificarea sau ștergerea acestora. Această separare clară a responsabilităților este esențială pentru menținerea unui nivel înalt de securitate în aplicațiile web moderne.

Un element critic al procesului de autorizare este obținerea acordului utilizatorului. Înainte de a permite accesul la resursele protejate, sistemul trebuie să se asigure că utilizatorul își dă consimțământul explicit pentru a utiliza anumite date sau funcționalități. Acest pas nu doar că asigură conformitatea cu standardele de confidențialitate, dar și întărește încrederea utilizatorilor în sistem, știind că aceștia au control asupra datelor lor personale.

Pentru a implementa aceste procese de autorizare, multe organizații apelează la soluții precum Spring Security, care oferă un sistem robust de control al accesului pe bază de roluri. Spring Security permite dezvoltatorilor să configureze politici de autorizare detaliate, asigurându-se că fiecare cerere este validată corespunzător în funcție de drepturile asociate fiecărui utilizator. Această abordare modulară facilitează, de asemenea, integrarea rapidă a noilor funcționalități fără a compromite securitatea întregului sistem.

Un alt standard modern de autorizare este OAuth, un protocol deschis definit în RFC 6749. OAuth permite utilizatorilor să acorde acces la datele lor către aplicații terțe fără a-și divulga parolele. Versiunile sale, de la OAuth 1.0 la OAuth 2.0 și în prezent dezvoltarea OAuth 2.1, oferă un cadru flexibil și sigur pentru gestionarea accesului. Prin intermediul token-urilor de acces cu durată de viață limitată, OAuth minimizează riscul ca un atacator să folosească informațiile de autentificare pe termen lung, contribuind la protejarea datelor sensibile.

Așadar, autorizarea în contextul serviciilor web se bazează pe o serie de etape esențiale – obținerea cheii de acces, autentificarea și autorizarea, obținerea acordului utilizatorului și, în final, apelarea funcționalităților serviciului prin API.

### 7.4.2. Autentificare

Autentificarea este un proces critic care asigură că doar utilizatorii legitimi pot accesa resursele protejate ale unei aplicații web. Există mai multe metode de autentificare, fiecare având propriile avantaje și aplicabilități. Una dintre metodele clasice este autentificarea bazată pe sesiunea web, care folosește un SID (*Session Identifier*) pentru a menține o sesiune activă. Deși Web-ul este, în mod implicit, stateless, utilizarea unui SID permite serverului să urmărească starea de autentificare a utilizatorului pe durata interacțiunii sale. Această abordare este însă sensibilă la atacurile de tip *session hijacking*, motiv pentru care soluțiile moderne au evoluat spre metode mai sigure.

O altă metodă este autentificarea pe bază de jetoane (*tokens*), care implică generarea unui token la autentificare și includerea acestuia în fiecare cerere ulterioară. Un exemplu comun este utilizarea OpenID în combinație cu JWT (*JSON Web Token*), un format standardizat de RFC 7519, care conține un obiect JSON ce permite interschimbul sigur de date între client și server. Fiecare cerere către API include tokenul JWT, care validează identitatea utilizatorului fără a necesita stocarea stării pe server. Alte metode includ *One-time-user URL*, folosită, de exemplu, de Tumblr pentru autentificare temporară, precum și *Single Sign-On* (SSO), care permite autentificarea utilizatorilor în cadrul mai multor aplicații înrudite, simplificând procesul de acces.

Pentru a îmbunătăți securitatea, metodele moderne de autentificare se bazează adesea pe autentificare multifactorială (2FA), unde utilizatorul trebuie să furnizeze cel puțin două probe diferite referitoare la identitatea sa. Aceste probe pot include elemente biometrice, cum ar fi recunoașterea amprentei, recunoașterea facială, scanarea irisului sau a retinei, identificare vocală sau chiar analizarea codului genetic (*DNA matching*). În plus, autentificarea poate fi realizată și prin intermediul unor dispozitive hardware, cum ar fi smartcard-urile, care oferă un nivel suplimentar de securitate. Prin combinarea acestor metode, sistemele moderne de autentificare asigură un control de acces robust și flexibil, protejând astfel informațiile sensibile și prevenind accesul neautorizat la resursele aplicațiilor web.

# 8. Capitolul 8. Virtualizare și containere Docker

În era digitală actuală, virtualizarea și containerele Docker au devenit instrumente fundamentale în dezvoltarea și implementarea aplicațiilor, inclusiv în domeniul software-ului de telecomunicații. Virtualizarea permite separarea completă a mediilor de execuție, permițând rularea mai multor sisteme de operare și aplicații pe același hardware, reducând astfel costurile și îmbunătățind eficiența resurselor.

Containerele Docker reprezintă o evoluție a virtualizării, oferind o modalitate ușoară și rapidă de a izola aplicațiile și de a le rula în medii standardizate, indiferent de modalitatea de dezvoltare sau de execuție. Spre deosebire de mașinile virtuale tradiționale, containerele se bazează pe izolarea proceselor în cadrul aceluiași sistem de operare, ceea ce le face mult mai rapide la pornire și mai eficiente în consumul de resurse.

În contextul software-ului de telecomunicații, virtualizarea și containerele Docker facilitează implementarea și gestionarea aplicațiilor critice, de la sisteme de gestionare a traficului și a rețelelor, la platforme de comunicare și soluții de stocare a datelor. Ele permit dezvoltatorilor să creeze medii de testare și producție izolate, să ruleze microservicii în mod concurent și să se asigure că aplicațiile sunt portabile și reproductibile în orice infrastructură.

## 8.1. Virtualizare

Virtualizarea este o tehnologie esențială care permite abstractizarea și partajarea resurselor hardware între mai multe medii de execuție izolate, cunoscute sub numele de mașini virtuale. Practic, o singură mașină fizică poate găzdui mai multe instanțe virtuale, fiecare rulând un sistem de operare complet independent, ceea ce optimizează utilizarea resurselor și reduce costurile hardware.

Unul dintre avantajele majore ale virtualizării este flexibilitatea și scalabilitatea pe care le oferă. Administratorii pot crea, configura, migra sau șterge mașini virtuale fără a afecta mediul de producție, facilitând astfel testarea rapidă a noilor aplicații. În plus, virtualizarea contribuie la îmbunătățirea securității, deoarece fiecare mașină virtuală rulează izolat, astfel încât eventualele probleme într-o instanță nu se propagă la celelalte.

Virtualizarea permite, de asemenea, consolidarea serverelor, reducând spațiul fizic necesar și consumul de energie. Aceasta permite rularea mai multor mașini virtuale pe un singur server fizic, ceea ce nu doar că optimizează resursele, dar și simplifică administrarea infrastructurii IT. Tehnologii precum VMware, Hyper-V și KVM sunt utilizate pe scară largă în mediile enterprise pentru a gestiona eficient resursele și a susține cerințele aplicațiilor moderne.

În contextul serviciilor cloud, virtualizarea joacă un rol vital, oferind medii de testare și producție izolate care sunt portabile și reproductibile indiferent de infrastructura fizică.

Aceasta a deschis drumul către tehnologii emergente, cum ar fi containerele Docker, care oferă o izolare similară, dar cu un consum de resurse mult mai redus și un timp de pornire rapid, facilitând astfel dezvoltarea rapidă și scalabilă a aplicațiilor moderne.

Virtualizarea aduce o creștere semnificativă a utilizării resurselor hardware, așa cum arată Fig. 8.1. Într-un mediu tradițional, capacitatea unui server poate fi utilizată în proporție de 6% până la 20%, ceea ce duce la un consum ineficient al resurselor disponibile. Prin implementarea virtualizării, se poate atinge o utilizare a CPU-ului de peste 65%, permițând rularea simultană a mai multor mașini virtuale pe același hardware și maximizând astfel investiția în infrastructură.

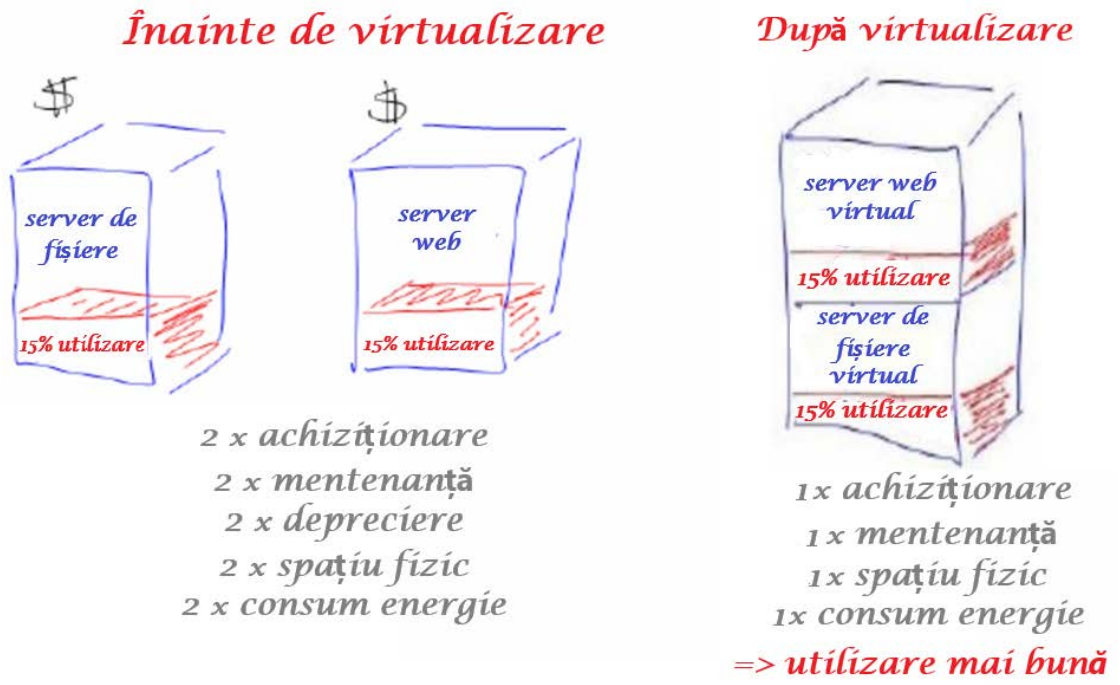



**Fig. 8.1.** Utilizrea resurselor înainte și după virtualizare.

Un alt avantaj major al virtualizării este reducerea costurilor de management. Administrarea unui număr mare de mașini fizice poate fi complexă și costisitoare, însă prin consolidarea serverelor și rularea mașinilor virtuale, se simplifică procesele de administrare și monitorizare. Această eficiență operațională nu doar că reduce costurile, dar și îmbunătățește capacitatea de a implementa rapid actualizări și patch-uri, asigurând o întreținere mai simplă și mai eficientă.

În plus, virtualizarea contribuie la îmbunătățirea securității și la reducerea perioadei de nefuncționare. Izolarea mașinilor virtuale permite ca eventualele probleme să fie limitate la o singură instanță, fără a afecta întregul sistem, iar recuperarea rapidă în caz de eșec devine posibilă. Această abordare reduce riscul de atacuri și probleme de securitate, oferind, în același timp, o mai mare reziliență a sistemului în fața defecțiunilor hardware sau software.

Virtualizarea are la bază un ansamblu de tehnici ce permit crearea unui strat abstract, oferind resurse logice care se bazează pe resurse fizice subiacente. Prin acest proces, caracteristicile detaliate ale hardware-ului sunt ascunse, iar utilizatorii percep doar o versiune virtuală, care poate fi folosită fără a avea cunoștință de specificul tehnic al resurselor reale.

Tehnicile de virtualizare sunt aplicabile în mai multe domenii. În ceea ce privește resursele computaționale, virtualizarea permite crearea de mașini virtuale (*Virtual Machine* - VM), unde fiecare VM rulează un sistem de operare complet independent. Un exemplu clasic este virtualizarea memoriei, unde memoria fizică a unui sistem este abstractizată, eliminând grija programatorilor de a gestiona limitele hardware; similar, fișierele sunt o abstracție a unui disc fizic, facilitând gestionarea datelor într-un mod transparent.

În plus, virtualizarea se extinde și asupra resurselor de stocare prin tehnici de virtual storage, care permit consolidarea și administrarea centralizată a datelor de pe multiple dispozitive de stocare. De asemenea, resursele de comunicare pot fi virtualizate, utilizând tehnici pentru virtual network, care permit crearea de rețele logice separate ce rulează peste infrastructura fizică existentă.

Astfel, virtualizarea joacă un rol esențial în modernizarea infrastructurilor IT, facilitând o gestionare eficientă a resurselor hardware prin crearea unor medii virtuale izolate. Această tehnologie nu doar îmbunătățește utilizarea resurselor, dar și oferă o bază solidă pentru dezvoltarea aplicațiilor moderne, permițând organizațiilor să își consolideze infrastructura și să răspundă rapid la cerințele dinamice ale pieței.

## 8.2. Docker

Docker este un instrument revoluționar care permite împachetarea unei aplicații împreună cu toate dependențele sale într-un container virtual, capabil să ruleze în orice mediu. Conform definiției de pe Wikipedia, această tehnică îmbunătățește semnificativ flexibilitatea și portabilitatea, permițând aplicațiilor să fie desfășurate indiferent dacă mediul este local, în cloud public sau privat, sau chiar pe hardware dedicat. Prin abstractizarea infrastructurii subiacente, Docker separă aplicația software de mediul de execuție, asigurând o consistență a performanței indiferent de platforma pe care este implementată.

Containerele Docker sunt omniprezente, fiind utilizate pe sisteme Linux, Windows, în centre de date și în mediile cloud. Ele funcționează ca un pachet complet, care conține codul aplicației, runtime-ul necesar, unelte de sistem, biblioteci și tot ce poate fi instalat pe un server. Această abordare garantează că aplicațiile containerizate rulează în mod consistent, eliminând problemele de incompatibilitate și de configurație care apar frecvent în medii diverse.

Un alt avantaj important al Docker este ușurința cu care permite dezvoltatorilor să implementeze și să gestioneze aplicații complexe în cadrul arhitecturilor moderne, cum ar fi

microserviciile. Prin izolare, fiecare container rulează independent, iar interacțiunea dintre containere se realizează prin rețele virtuale configurabile, facilitând scalabilitatea și actualizările continue fără a perturba întregul sistem. În plus, integrarea Docker în procesele de dezvoltare și livrare continuă (CI/CD) a revoluționat modul în care aplicațiile sunt testate, implementate și întreținute.

Docker implementează un API de nivel înalt care permite crearea și gestionarea containerelor, asigurând astfel rularea proceselor în mod izolat. Acest API simplifică interacțiunea cu infrastructura, permițând dezvoltatorilor să lanseze, să oprească și să administreze containerele fără a se confrunta cu complexitatea sistemului de operare. Prin intermediul acestui API, Docker oferă o interfață intuitivă pentru orchestrarea aplicațiilor containerizate, facilitând astfel dezvoltarea și implementarea rapidă a serviciilor moderne.

O caracteristică distinctivă a Docker este faptul că este construit peste facilitățile oferite de sistemul de operare – inițial, kernel Linux – ceea ce înseamnă că nu este necesară rularea unui sistem de operare complet separat pentru fiecare container. În schimb, toate containerele partajează același kernel, însă fiecare este restricționat la o anumită cantitate de resurse, cum ar fi CPU, memorie și operații de tip I/O. Această abordare maximizează utilizarea resurselor hardware și reduce semnificativ costurile și timpul de pornire al aplicațiilor, oferind o eficiență remarcabilă în mediile de testare și de producție.

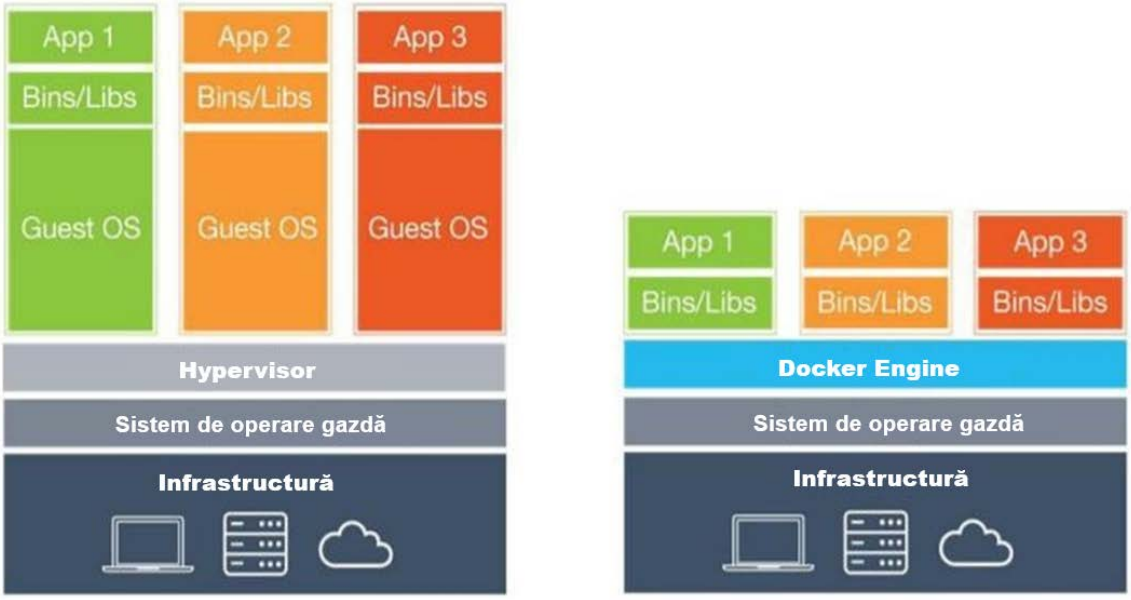



**Fig. 8.2.** Comparație Mașini Virtuale vs. Docker.

Docker aduce numeroase avantaje în comparație cu mașinile virtuale tradiționale (Fig. 8.2). În timp ce o mașină virtuală include întregul sistem de operare, aplicațiile, bibliotecile și alte componente necesare, ceea ce poate consuma zeci de Gigabytes de resurse, containerele Docker sunt mult mai ușoare. Acestea includ doar aplicația și dependențele sale, partajând

kernelul sistemului de operare gazdă, ceea ce reduce semnificativ dimensiunea și consumul de resurse.

Prin rularea în procese izolate pe același sistem de operare, containerele Docker nu sunt legate de o anumită infrastructură, oferind astfel o portabilitate foarte bună. Astfel, aplicațiile containerizate pot fi dezvoltate și testate local, apoi transferate rapid pe orice mediu – fie că este vorba despre un server fizic, un mediu de virtualizare sau o platformă cloud – fără a necesita modificări majore. Această flexibilitate și eficiență fac ca Docker să fie o alegere preferată pentru dezvoltarea și implementarea aplicațiilor moderne, contribuind la reducerea costurilor și la optimizarea resurselor.

### 8.2.1. Docker sau virtualizare

Docker accesează și diverse facilități de virtualizare prin intermediul unor biblioteci specializate, care îi permit să ofere funcționalități avansate de izolare și management al resurselor. Avantajele sunt evidente: prin crearea și gestionarea containerelor, deployment-ul aplicațiilor devine mult mai simplu pe sisteme distribuite, permițând rularea autonomă a mai multor aplicații sau task-uri pe o singură mașină fizică sau pe un întreg spectru de mașini virtuale.

În practică, cele două tehnologii – containerele Docker și mașinile virtuale – sunt adesea utilizate împreună. Majoritatea furnizorilor de infrastructură IT rulează tehnologii de tip *bare-metal virtualization* (cum ar fi XEN, VMware ESXi, Hyper-V) pentru a crea mașini virtuale, deasupra cărora este instalat un sistem de operare (de ex. Ubuntu). Peste acest mediu virtualizat se pot rula containere Docker, care asigură o izolare suplimentară și o implementare rapidă a aplicațiilor. Această abordare oferă flexibilitatea și compatibilitatea mașinilor virtuale, combinată cu eficiența și ușurința de gestionare a containerelor.

Scott S. Lowe, arhitect de inginerie la VMware, sugerează că, pentru alegerea între mașini virtuale și containere, trebuie să se analizeze scopul urmărit. Cu alte cuvinte, dacă se dorește rularea mai multor instanțe ale unei singure aplicații, cum ar fi MySQL, se utilizează un container. Dacă este necesară flexibilitatea de a rula mai multe aplicații diferite, se preferă o mașină virtuală. În plus, containerele pot bloca un anumit sistem de operare. Acest lucru poate fi benefic: nu mai există grija pentru dependențe odată ce aplicația rulează corect într-un container. Pe de altă parte, cu mașinile virtuale, indiferent de hypervisor – KVM, Hyper-V, vSphere, Xen – se poate rula aproape orice sistem de operare. Dacă este nevoie să se ruleze o aplicație obscură, care funcționează doar pe QNX, o mașină virtuală oferă această posibilitate, în timp ce generația actuală de containere nu este la fel de permisivă.

Din această perspectivă, containerele sunt ideale pentru a rula multiple instanțe ale aceleiași aplicații (de exemplu, servicii microservicii, baze de date replicabile), asigurând o gestionare mai simplă a dependențelor și un consum redus de resurse. În schimb, mașinile virtuale oferă o libertate mai mare de a rula sisteme de operare diferite și aplicații variate,

fiind preferate atunci când este necesară o compatibilitate extinsă. Astfel, alegerea între containere și mașini virtuale depinde de tipul de aplicație, de nevoile de scalare și de cerințele de compatibilitate ale mediului. De cele mai multe ori, cele două tehnologii se completează reciproc, permițând configurarea unor arhitecturi hibride, eficiente și ușor de administrat.

## 8.3. Caracteristici și componente Docker

Docker se bazează pe conceptul de containere, care sunt unități complet izolate de software, fiecare reprezentând propria „lume" mică în cadrul sistemului de operare gazdă. Fiecare container include tot ce este necesar pentru a rula codul – de la aplicație, configurări, procese și rețele, până la toate dependențele și, în unele cazuri, chiar o parte mică a sistemului de operare. Această abordare permite rularea aplicațiilor într-un mediu uniform și previzibil, indiferent de mediul în care sunt implementate.

Un alt aspect cheie al Docker este că acesta se bazează pe un sistem Linux, care furnizează facilitățile necesare pentru izolarea proceselor. Containerele sunt create și gestionate cu ajutorul unui set de instrumente robuste, care asigură că fiecare container operează independent, fără să interfereze cu alte containere sau cu sistemul de operare gazdă. Acest nivel de izolare nu doar că îmbunătățește securitatea, dar permite și o utilizare mult mai eficientă a resurselor hardware, deoarece containerele partajează kernelul sistemului de operare în loc să ruleze un sistem de operare complet separat.

### 8.3.1. Docker Image

O imagine Docker reprezintă un șablon *read-only* care conține un set complet de instrucțiuni necesare pentru a crea un container care să ruleze pe platforma Docker. Această imagine include toate elementele esențiale pentru executarea unei aplicații: codul sursă, bibliotecile, fișierele de configurare, dependențele și setările de mediu. Prin utilizarea unei imagini, dezvoltatorii pot defini mediul de rulare al aplicației într-un mod complet izolat și reproductibil, ceea ce garantează că aplicația va funcționa în mod constant, indiferent de mediul de execuție.

Imaginile Docker pot fi identificate și gestionate printr-o combinație de nume și tag-uri sau printr-un identificator unic (ID). Această metodă de versionare și etichetare permite dezvoltatorilor să urmărească modificările și să mențină o evidență clară a versiunilor diferite ale unei aplicații. De exemplu, o imagine poate fi denumită `myapp:latest` sau `myapp:1.0.3`, facilitând astfel distribuirea și actualizarea acesteia în diferite medii, de la mediile de dezvoltare la cele de producție.

Un aspect esențial al imaginilor Docker este imutabilitatea lor. Odată ce o imagine a fost creată, ea nu se modifică, asigurând astfel că mediul în care se execută containerul rămâne constant. Această caracteristică se deosebește de mașinile virtuale tradiționale, unde fiecare

instanță include întregul sistem de operare și poate suferi modificări sau actualizări dinamice, ceea ce poate conduce la probleme de compatibilitate sau inconsistență între mediile de rulare.

Datorită faptului că imaginile Docker sunt mult mai ușoare decât mașinile virtuale complete, acestea pot fi create, distribuite și rulate cu un consum minim de resurse. Containerizarea permite rularea mai multor instanțe ale unei aplicații pe același sistem de operare gazdă, partajând kernelul acestuia, dar izolând în același timp procesele și dependențele fiecărui container.

Imaginile Docker sunt stocate și gestionate în registre de imagini, cum ar fi Docker Hub, Google Container Registry (GCS) sau Amazon ECR, ceea ce facilitează partajarea și distribuția lor la scară largă. Această capacitate de a distribui imagini consistente și reproductibile între diferite medii asigură că aplicațiile containerizate funcționează în mod identic, reducând astfel erorile și timpul necesar pentru configurarea și testarea mediilor de rulare. Prin urmare, Docker a revoluționat modul în care aplicațiile sunt dezvoltate și implementate, oferind un cadru robust, flexibil și eficient pentru gestionarea mediilor de execuție.

Comanda `docker run` este utilizată pentru a crea și rula un container dintr-o imagine Docker specificată (Fig. 8.3). Aceasta combină funcționalitățile de creare a unui container și de lansare a unei comenzi într-un singur pas, permițând utilizatorilor să pornească rapid o instanță a aplicației sau serviciului dorit. Oferind opțiuni precum *-d* (pentru rulare în fundal), *-p* (pentru maparea porturilor) sau *-v* (pentru atașarea de volume), comanda docker run oferă o flexibilitate mare în configurarea mediului de execuție al containerului.


**Fig. 8.3.** Comanda *docker run*.

Prin această comandă, containerul va rula într-un mediu izolat, cu resursele alocate conform setărilor specificate, iar fiecare container poate fi gestionat independent.

### 8.3.2. Docker Container

Un container Docker este o unitate standard de software care grupează codul și toate dependențele necesare pentru a rula o aplicație într-un mediu izolat. Containerul conține tot ce este necesar pentru execuția aplicației: codul sursă, bibliotecile, fișierele de configurare și chiar variabilele de mediu. Prin această izolare, Docker permite dezvoltatorilor să evite problemele de compatibilitate care apar atunci când o aplicație rulează pe medii diferite.

Un aspect important al containerelor Docker este portabilitatea lor. Deoarece ele împachetează toate dependențele necesare, containerele pot fi rulate pe orice sistem care suportă Docker, fără a necesita configurări suplimentare ale mediului de execuție.

Comanda `docker commit` joacă un rol esențial în gestionarea containerelor. Aceasta permite salvarea modificărilor efectuate într-un container care rulează, creând astfel o nouă imagine Docker imutabilă. Prin utilizarea acestei comenzi, orice ajustare, modificare de configurare sau actualizare a aplicației efectuată într-un container poate fi capturată și transformată într-o imagine permanentă, așa cum ilustrează Fig. 8.4. Această imagine poate fi apoi distribuită, partajată sau folosită pentru implementări ulterioare, asigurând o gestionare eficientă a versiunilor și facilitând un ciclu de dezvoltare continuu.

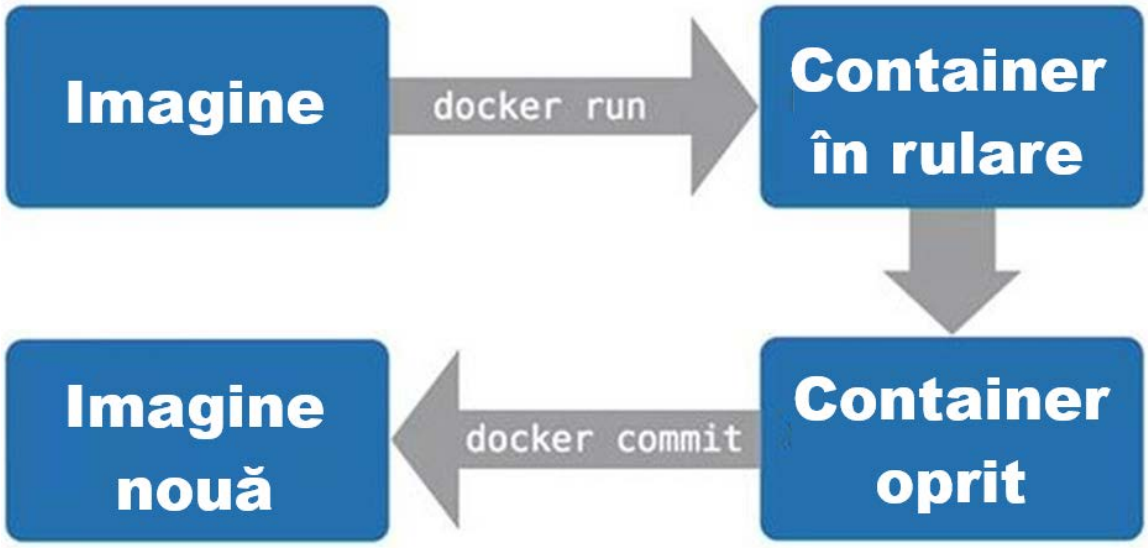



**Fig. 8.4.** Crearea unei imagini dintr-un container.

### 8.3.3. Docker Networks

Docker Networks reprezintă un set de funcționalități esențiale care permit conectarea containerelor între ele și cu mediul extern. Atunci când un port al unui container este expus, Docker creează o cale de rețea ce traversează straturile de rețea ale mașinii gazdă, facilitând astfel accesul din exterior la containerul respectiv. Acest mecanism este crucial pentru comunicarea între aplicațiile containerizate și asigură că serviciile pot fi accesate în mod eficient, fie că este vorba despre comunicare internă între containere, fie despre expunerea serviciilor către clienți externi.

Docker oferă un set extins de opțiuni de rețea pentru a controla modul în care containerele se conectează între ele și pentru a asigura securitatea acestora. Comanda `docker network ls` permite listarea tuturor rețelelor existente, oferind o vizibilitate clară asupra configurației de rețea. Rețeaua implicită de tip *bridge* este cea utilizată de containerele care nu specifică o rețea personalizată; aceasta izolează containerele, dar permite totuși comunicarea între ele prin intermediul unei rețele interne virtuale.

În situații în care se dorește eliminarea izolării rețelei pentru un container, se poate opta pentru rețeaua de tip *host*, unde containerul utilizează direct stiva de rețea a sistemului de operare gazdă. Acest lucru poate oferi performanțe sporite în anumite cazuri, însă implică riscuri de securitate, deoarece containerul nu beneficiază de izolare la nivel de rețea. În contrast, opțiunea *none* este destinată containerelor care nu trebuie să aibă acces la rețea, eliminând complet interfața de rețea a containerului și, astfel, limitând posibilitățile de comunicare.

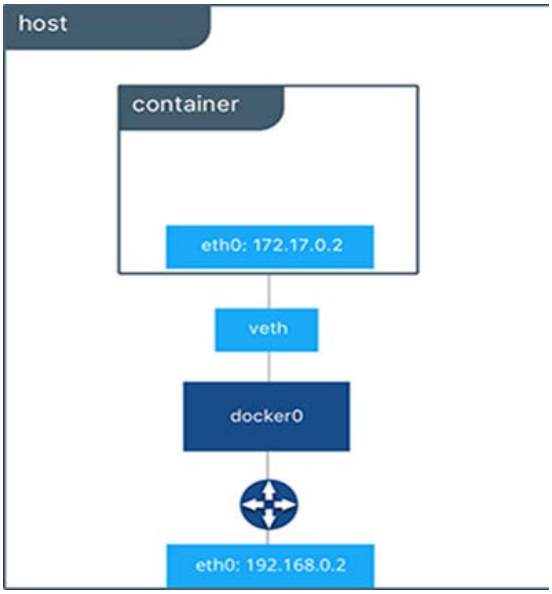



**Fig. 8.5.** Docker networks.

Fig. 8.5 ilustrează modul în care Docker creează o rețea virtuală pentru containere, evidențiind conexiunea dintre interfața de rețea a containerului (ex.: eth0 cu adresa 172.17.0.2), interfața virtuală (*veth*) și puntea (*docker0*) configurată la nivelul gazdei, care, la rândul ei, comunică printr-o interfață fizică (ex.: eth0 cu adresa 192.168.0.2). Această arhitectură permite containerelor să aibă adrese IP distincte, oferindu-le izolare de rețea, dar și posibilitatea de a comunica între ele prin rețeaua virtuală.

Pe lângă rețelele predefinite (*bridge*, *host* și *none*), Docker permite și crearea de rețele personalizate, care pot fi configurate pentru a se potrivi nevoilor specifice ale unei aplicații, folosing comanda `docker network create`. Aceste rețele personalizate pot fi setate cu diferite reguli de izolare și securitate, oferind flexibilitate maximă în gestionarea comunicațiilor între containere și mediul extern.

### 8.3.4. Docker Volumes

Docker oferă o funcționalitate esențială numită volumes, care facilitează partajarea și persistența datelor între containere și între containere și sistemul gazdă. Aceste volume sunt, în esență, foldere sau discuri virtuale pe care le pot monta containerele, permițând astfel

stocarea datelor în afara ciclului de viață al containerului. Această separare a datelor de aplicația containerizată asigură că, atunci când un container este șters sau recreat, datele stocate în volum nu se pierd, ceea ce este esențial pentru aplicațiile care necesită persistență pe termen lung, cum ar fi bazele de date sau fișierele de configurare.

Un aspect important al volumelor Docker este diferența între volumurile persistente și volumurile efemere. Volumele persistente sunt stocate pe discul gazdei și continuă să existe chiar dacă acel container care le utilizează este eliminat. În schimb, volumele efemere există doar pe durata de viață a containerelor care le folosesc; odată ce containerul nu mai este utilizat, datele din volumul respectiv dispar definitiv. Această distincție este crucială pentru a alege strategia de stocare adecvată în funcție de necesitățile aplicației.

Utilizarea corectă a volumelor aduce numeroase beneficii practice. În primul rând, aceasta permite separarea clară a datelor și a codului, facilitând gestionarea și actualizarea aplicațiilor. În al doilea rând, volumele contribuie la portabilitatea aplicațiilor, deoarece datele stocate într-un volum pot fi partajate între diferite medii de execuție, fie că este vorba despre un server local, un centru de date sau o platformă cloud.

În plus, volumele permit implementarea unor strategii de backup și recuperare a datelor mai eficiente. Deoarece datele sunt stocate independent de containere, acestea pot fi replicate, migrate și restaurate fără a perturba funcționarea aplicației. Aceasta asigură o continuitate a operațiunilor și o reziliență crescută în fața eventualelor erori sau defecțiuni hardware.

Prin intermediul comenzilor precum `docker run -v` sau `docker volume create`, administratorii pot crea și gestiona cu ușurință volume de date, configurând accesul și permisiunile necesare. De asemenea, Docker oferă suport pentru volume partajate între containere, permițând colaborarea între diferite componente ale unei aplicații.

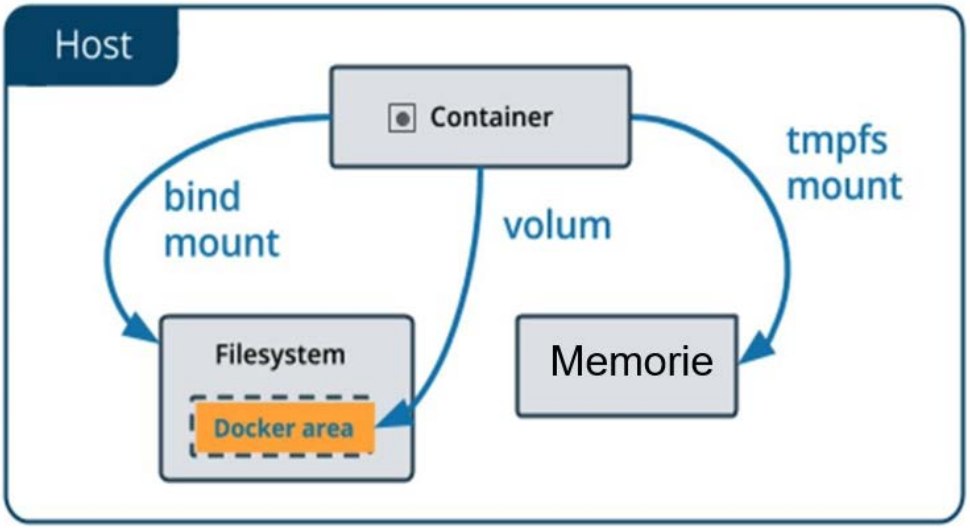



**Fig. 8.5.** Docker Volumes.

Fig. 8.5 prezintă diferitele moduri în care un container Docker poate atașa și utiliza un volum. Pe de o parte, există bind mount, prin care containerul este legat direct de un director din sistemul de fișiere al gazdei, permițând persistarea datelor pe discul fizic. Pe de altă parte, se poate crea un volum nativ Docker, care este gestionat de Docker și stocat într-un director special de pe sistemul de operare gazdă. În plus, imaginea evidențiază posibilitatea montării unui sistem de fișiere în memorie (*tmpfs*), oferind o zonă temporară și foarte rapidă de stocare. Aceste trei mecanisme oferă flexibilitate în alegerea modului de stocare și partajare a datelor între container și sistemul gazdă, în funcție de cerințele de performanță și persistență.

## 8.4. Containerizarea aplicațiilor

Containerizarea reprezintă o metodă modernă de împachetare a aplicațiilor și a tuturor dependențelor acestora într-un mediu izolat, care poate fi rulat în mod consistent pe orice infrastructură compatibilă cu Docker. O aplicație Spring Boot, de exemplu, poate fi containerizată pentru a asigura că tot ce este necesar pentru executarea acesteia este inclus într-o singură imagine Docker. Această imagine nu conține doar codul Java al aplicației, ci și întreaga infrastructură necesară, care include un shell pentru execuția comenzilor, instrumente de construire a aplicației (cum ar fi Maven), Java runtime (JRE sau JDK), serverul Tomcat (dacă aplicația este rulată ca o aplicație web) și o porțiune din sistemul de operare.

Prin containerizare, se asigură că aplicația Spring Boot rulează într-un mediu complet izolat, independent de configurațiile și dependențele sistemului gazdă. Acest lucru elimină problemele de incompatibilitate între medii diferite, permițând dezvoltatorilor să dezvolte și să ruleze și să testeze aplicația cu încredere, știind că mediul de execuție rămâne același în toate etapele ciclului de viață al aplicației. Astfel, containerizarea contribuie la o mai bună consistență a aplicației și la reducerea erorilor generate de diferențele între mediile de dezvoltare, testare și producție.

De asemenea, includerea tuturor componentelor necesare (Shell, Java build tools, runtime, Maven, server Tomcat și o parte din sistemul de operare) în imaginea Docker permite o gestionare eficientă a dependențelor. Fiecare container este o unitate completă și reproductibilă, ceea ce facilitează actualizările și scalarea aplicațiilor în medii distribuite. Acest lucru este esențial în procesele de livrare continuă (CI/CD), deoarece imaginea containerizată poate fi transferată rapid între diferite servere și platforme de cloud.

Un Dockerfile este un fișier text care conține o serie de comenzi ce definesc modul în care se va construi o imagine Docker. Structura unui astfel de fișiier este următoarea:

- Primul pas este specificarea imaginii de bază folosind comanda `FROM`, de exemplu `FROM ubuntu:latest`, care stabilește imaginea de bază a containerului., care poate fi oricare dintre cele disponibile pe Docker Hub sau în alte registre.

- Comenzile precum `ADD` și `COPY` sunt utilizate pentru a transfera fișiere și directoare din sistemul de fișiere gazdă în container, permițând includerea aplicației, scripturilor și a altor resurse necesare. În timpul procesului de construire, comanda `RUN` execută diverse instrucțiuni, cum ar fi instalarea de pachete sau configurarea mediului, asigurându-se că toate dependențele sunt satisfăcute înainte de lansarea containerului. Comanda `USER` specifică sub ce utilizator se vor rula procesele din container, contribuind la securizarea mediului de execuție.
- Pentru gestionarea datelor persistente, comanda `VOLUME` declară zonele de stocare care pot fi montate atât în container, cât și pe sistemul de fișiere al gazdei. Comanda `WORKDIR` setează directorul de lucru implicit, facilitând rularea ulterioară a comenzilor în contextul unui anumit director.
- La final, pentru a defini procesul care va rula la pornirea containerului, se utilizează fie comanda `CMD`, care specifică comanda implicită ce poate fi suprascrisă, fie `ENTRYPOINT`, care stabilește punctul de intrare al containerului și este adesea combinată cu parametri adiționali.

# 9. Capitolul 9. Servicii pentru IoT. Protocolul MQTT

Web-ul reprezintă coloana vertebrală a ecosistemului IoT, oferind conectivitatea globală necesară pentru ca dispozitivele să comunice între ele și cu utilizatorii, indiferent de locație. Această infrastructură permite ca dispozitivele să transmită date în timp real către servere și aplicații, facilitând monitorizarea și controlul la distanță al acestora. Prin intermediul aplicațiilor web și mobile, utilizatorii pot accesa și vizualiza informațiile colectate, precum și controla dispozitivele – de la sisteme de case inteligente și vehicule conectate până la soluții industriale automatizate.

Integrarea IoT cu tehnologiile web nu doar că asigură accesibilitatea datelor, dar facilitează și combinarea acestora cu tehnologii avansate, cum ar fi cloud computing, inteligența artificială și big data. Astfel, datele colectate de dispozitive pot fi stocate, procesate și analizate eficient, transformându-le în informații acționabile pentru luarea deciziilor inteligente. Serviciile web și API-urile, precum REST sau GraphQL, asigură interoperabilitate și flexibilitate, permițând integrarea fără probleme a IoT cu alte sisteme și platforme. De asemenea, tehnologiile de comunicare, precum WebSockets și MQTT, joacă un rol crucial în ecosistemul IoT. WebSockets permit o comunicație bidirecțională rapidă între servere și dispozitive, esențială pentru aplicațiile care necesită reacții imediate, cum ar fi sistemele de securitate sau cele din domeniul sănătății. În paralel, protocolul MQTT, proiectat special pentru rețelele cu lățime de bandă limitată și latente ridicate, optimizează transferul de date între dispozitive, asigurând fiabilitate și eficiență chiar și în medii de comunicații critice.

Prin urmare, integrarea IoT cu web-ul deschide calea spre dezvoltarea unor ecosisteme interconectate și inteligente, unde automatizarea și analiza datelor joacă un rol central. Platformele de cloud computing oferă infrastructura necesară pentru a stoca și procesa aceste date, în timp ce tehnologiile de comunicație asigură un transfer rapid și sigur al informațiilor. Astfel, web-ul devine nu doar o interfață de acces la date, ci și un catalizator al inovației în diverse domenii, de la gestionarea traficului urban până la monitorizarea sistemelor industriale în timp real.

## 9.1. Protocolul MQTT

Protocolul MQTT (*Message Queuing Telemetry Transport*) este conceput special pentru a răspunde nevoilor comunicațiilor în mediile cu constrângeri de resurse, caracteristice Internetului Obiectelor (IoT). MQTT se remarcă prin faptul că este un protocol ușor, ideal pentru comunicațiile de la mașină la mașină (M2M), fiind adaptat pentru setări cu lățime de bandă redusă și latență ridicată. Prin modelul său de tip *publisher-subscriber*, MQTT permite transmiterea eficientă a mesajelor prin intermediul cozii de mesaje, astfel încât dispozitivele

pot comunica rapid și în mod fiabil, chiar și în medii cu conexiuni TCP/IP unde ordinele și integritatea datelor sunt critice.

Un aspect definitoriu al MQTT este simplitatea sa. Protocoalele de transport, cum ar fi TCP/IP, asigură conexiuni bidirecționale, ordonate și fără pierderi, esențiale pentru funcționarea corectă a aplicațiilor IoT. Protocolul este standardizat ca un standard deschis OASIS și este recomandat ISO/IEC 20922, ceea ce îi conferă un nivel ridicat de interoperabilitate între diferite dispozitive și platforme. Acest lucru face ca MQTT să fie o alegere preferată pentru aplicațiile care implică mii sau chiar milioane de dispozitive, deoarece poate gestiona volum mare de trafic și poate fi scalat la scară largă fără a compromite performanța.

Versiunea 5.0 a protocolului, lansată în 2019, introduce îmbunătățiri semnificative, printre care se numără o scalabilitate crescută și o raportare a erorilor mai precisă. Aceste caracteristici sunt esențiale pentru medii industriale sau urbane, unde este necesară monitorizarea și controlul în timp real al unui număr mare de dispozitive. În plus, noile funcționalități oferă o mai bună gestionare a sesiunilor, facilitând comunicarea între dispozitive în mod continuu, chiar și în condiții de rețea instabile.

### 9.1.1. Modelul publisher-subscriber

Modelul publisher-subscriber este un concept arhitectural de mesagerie utilizat pe scară largă pentru a permite comunicarea între diferite componente ale unui sistem, fără ca acestea să fie strâns cuplate între ele. În această abordare, editorii (*publishers*) trimit mesaje către un intermediar numit broker, care se ocupă de distribuirea acestora către abonații (*subscribers*) interesați de subiectele respective (*topics*). Astfel, editorii și abonații nu au nevoie să cunoască identitatea sau locația celuilalt, deoarece comunicarea are loc exclusiv prin broker, ceea ce facilitează o arhitectură modulară și scalabilă în sisteme distribuite.

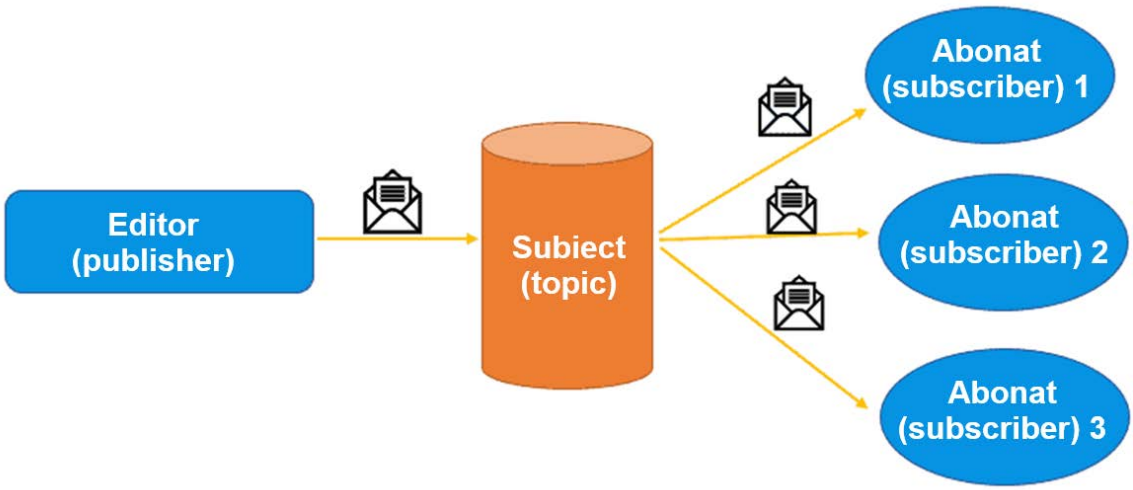



**Fig. 9.1.** Modelul publisher-subsciber.

Procesul de funcționare al modelului pub-sub începe cu trimiterea unui mesaj de către un editor către broker, însoțit de un subiect identificator (*topic*). Brokerul verifică apoi lista de abonați care și-au exprimat interesul pentru acel subiect și transmite mesajul către toți aceștia, așa cum ilustrează Fig. 9.1. Abonații primesc și procesează mesajul conform logicii lor interne, iar dacă niciun abonat nu este interesat de subiect, mesajul este eliminat. În plus, sistemul permite abonaților să se înregistreze pentru subiecte specifice, astfel încât să primească doar mesajele relevante pentru interesele lor. Această decuplare între editori și abonați, prin intermediul brokerului, oferă un nivel ridicat de flexibilitate și scalabilitate, fiind esențială pentru dezvoltarea de aplicații complexe și distribuite.

### 9.1.2. Caracteristici MQTT

Protocolul MQTT utilizează modelul de tip publisher-subscriber, unde toți clienții comunică prin intermediul unei entități centrale numită Broker. Această arhitectură decuplată permite ca editorii să trimită mesaje către broker fără a cunoaște identitatea abonaților, iar brokerul, la rândul său, distribuie mesajele către toți abonații interesați, facilitând astfel o comunicare eficientă și scalabilă.

Unul dintre avantajele esențiale ale MQTT este capacitatea sa de a funcționa în condiții de rețea instabilă sau cu latență ridicată, deoarece nu necesită o conexiune persistentă între client și server. MQTT oferă diferite niveluri de calitate a serviciului (*Quality of Service* – QoS), permițând astfel garantarea livrării mesajelor în funcție de necesitățile aplicației: de la livrare „cel puțin o dată” până la livrare exact o singură dată, chiar și în condiții de rețea problematice. De asemenea, protocolul este proiectat pentru a avea un *overhead* minim, cu pachete de date de dimensiuni reduse, ceea ce contribuie la eficiența în utilizarea lățimii de bandă și la reducerea consumului de energie, aspect esențial pentru dispozitivele IoT.

În plus, MQTT permite stocarea și retransmiterea mesajelor în caz de deconectare temporară a unui client, iar funcționalități precum mesajele reținute (*retained messages*) și mecanismul "*last will and testament*" asigură că starea dispozitivului este comunicată corect chiar și după întreruperi neașteptate. Aceste caracteristici fac ca MQTT să fie foarte potrivit pentru scenarii precum monitorizarea de la distanță, controlul caselor inteligente, aplicații industriale și vehicule conectate, unde fiabilitatea și rapiditatea transferului de date sunt critice.

Utilizarea MQTT se extinde dincolo de mediul IoT, fiind implementată și în aplicații mobile și sisteme de mesagerie în timp real, unde capacitatea de a transmite informații rapid și eficient este esențială. Protocolul este recunoscut și suportat de majoritatea platformelor de cloud, permițând integrarea ușoară a dispozitivelor conectate cu infrastructuri moderne.

## 9.2. Componente MQTT

Arhitectura MQTT se bazează pe câteva componente cheie care lucrează împreună pentru a asigura o comunicare eficientă în sistemele de tip *publisher-subscriber*. La baza acestei arhitecturi se află mesajul, care reprezintă unitatea de date transmisă între clienți, și subiectul (*topic*), un șir identificator care clasifică mesajele în funcție de conținutul sau domeniul lor de interes. Componentele unei arhitecturi MQTT sunt prezentate în Fig. 9.2.

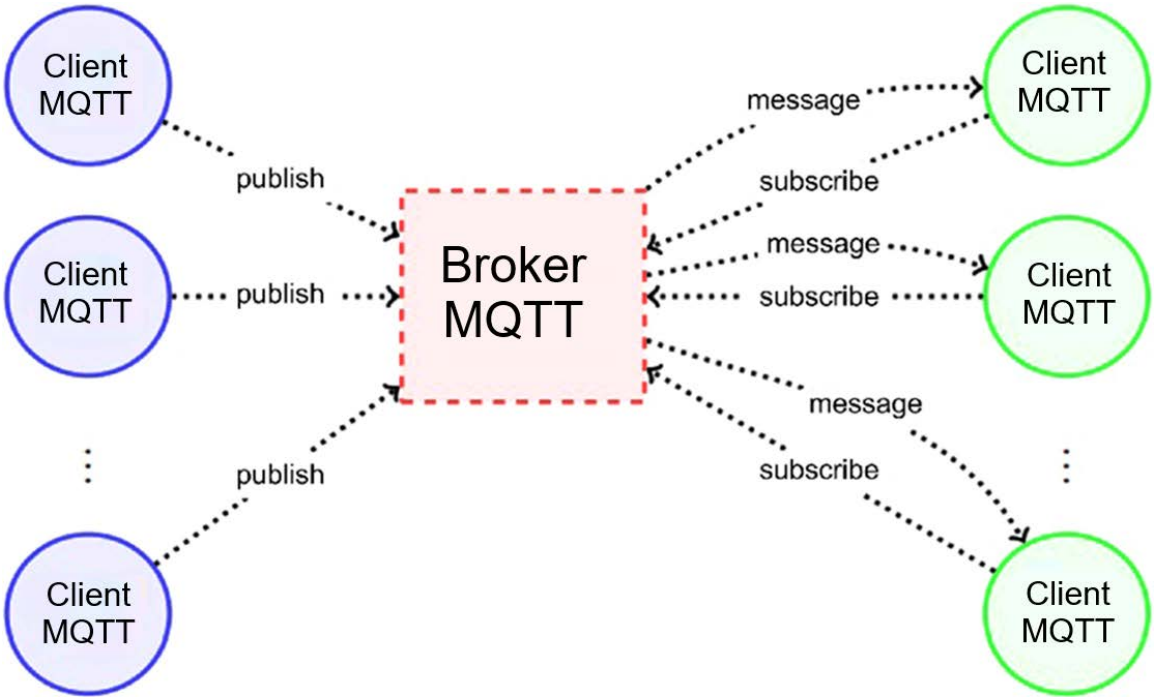



**Fig. 9.2.** Componente MQTT.

Clienții MQTT sunt entități care pot publica sau se pot abona la anumite subiecte. Editorii (*publishers*) trimit mesaje către broker, specificând subiectul relevant, iar abonații (*subscribers*) primesc mesajele pentru subiectele la care s-au înregistrat. Acest model de comunicare decuplat permite ca editorii și abonații să nu fie direct conectați între ei, ci să interacționeze prin intermediul unei componente centrale.

Brokerul MQTT reprezintă inima sistemului, fiind serverul central care primește toate mesajele publicate, le filtrează și le distribuie către clienții abonați la subiectele respective. Un broker poate gestiona mii de clienți simultan, asigurând astfel o comunicare scalabilă și fiabilă. De asemenea, brokerul păstrează sesiunile persistente ale clienților, gestionând subscripțiile și asigurând livrarea mesajelor, inclusiv a celor pierdute, atunci când clienții se reconectează.

Astfel, prin colaborarea acestor componente – mesaje, clienți, broker și subiecte – MQTT asigură un mecanism robust de publicare/abonare, esențial pentru comunicațiile eficiente în

rețelele IoT și nu numai. Această structură permite transmiterea rapidă a datelor între numeroase dispozitive, garantând că informațiile sunt livrate către destinatarii corecți.

### 9.2.1. Mesaje

Mesajul MQTT reprezintă unitatea de date transmisă între clienți și broker într-un sistem de comunicații de tip *publisher-subscriber*. Fiecare mesaj conține o sarcină utilă (*payload*) care poate fi sub formă de text, JSON, sau alt format binar, împreună cu parametrii esențiali: calitatea serviciului, care definește nivelul de garantare al livrării mesajului (de exemplu, 0 – cel mult o dată, 1 – cel puțin o dată, 2 – exact o dată), colecția de proprietăți (meta-date suplimentare precum timestamp-uri, identificatori unici etc.), și numele subiectului (*topic*) la care este asociat mesajul.

Tipurile de mesaje MQTT sunt concepute pentru a acoperi diferite etape ale comunicației. De exemplu, mesajul CONNECT este folosit de un client pentru a inița o conexiune cu brokerul, transmițând informații despre identificatorul clientului, opțiuni de autentificare și parametri pentru setarea sesiunii. Mesajul DISCONNECT semnalează închiderea conexiunii, asigurând că clientul finalizează toate operațiunile curente înainte de deconectare. Mesajul PUBLISH este folosit pentru a trimite efectiv date către broker, care le distribuie către toți abonații interesați de subiectul specificat. După trimiterea unui mesaj PUBLISH, execuția se reia rapid, facilitând o comunicare eficientă și în timp real.

Extinzând funcționalitatea, protocolul MQTT include și alte tipuri de mesaje, cum ar fi SUBSCRIBE (care permite clienților să se aboneze la anumite subiecte), UNSUBSCRIBE (pentru renunțarea la abonare) și PINGREQ/PINGRESP (folosite pentru a menține conexiunea activă și a verifica starea rețelei).

### 9.2.2. Clienți

Fiecare dispozitiv sau aplicație care utilizează protocolul MQTT este numit generic client, iar acesta poate juca rolul de editor (*publisher*), abonat (*subscriber*) sau chiar ambele simultan. Astfel, un client MQTT este responsabil pentru stabilirea conexiunii cu brokerul, trimiterea mesajelor către acesta și/sau primirea mesajelor de la broker, în funcție de subiectele la care s-a abonat. Această flexibilitate permite ca același dispozitiv să participe în mod activ în comunicarea de la mașină la mașină, fără a fi nevoie să se cunoască direct între ele, facilitând astfel un sistem decuplat și scalabil.

Clientul MQTT efectuează două operațiuni principale: publicarea și abonarea. Când clientul trimite date către broker, operația este denumită publicare, iar mesajul este etichetat cu un subiect specific. Brokerul va distribui apoi acel mesaj tuturor clienților care s-au abonat la subiectul respectiv. Pe de altă parte, operația de abonare implică înregistrarea clientului pentru a primi mesaje pe anumite subiecte, asigurând că acesta primește doar informațiile de interes. Această separare clară între publicare și abonare permite gestionarea eficientă a

fluxurilor de date într-un sistem distribuit, unde fiecare client poate opera independent și în mod asincron.

În plus, MQTT permite clienților să se dezaboneze de la anumite subiecte și să închidă conexiunea atunci când nu mai este necesară comunicarea, ceea ce contribuie la reducerea consumului de resurse și la o mai bună gestionare a sesiunilor active. Acest model asigură o interoperabilitate eficientă și un transfer de date în timp real, contribuind la succesul comunicațiilor în sisteme de la scară mică până la rețele complexe de comunicații distribuite.

### 9.2.3. Broker

Brokerul MQTT este componenta centrală a arhitecturii MQTT, având rolul de a facilita comunicarea între clienți într-un sistem decuplat. Acesta acționează ca un intermediar care primește toate mesajele publicate de clienți, procesează cererile de abonare și dezabonare și livrează mesajele către toți clienții abonați la subiectele respective. Prin gestionarea tuturor comunicațiilor într-un mod centralizat, brokerul asigură că mesajele sunt transmise în mod fiabil, chiar dacă anumite conexiuni sunt temporar întrerupte. În plus, brokerul poate gestiona mii de clienți simultan, ceea ce îl face esențial în medii IoT cu un număr mare de dispozitive conectate.

Brokerul nu doar că distribuie mesajele, ci păstrează și sesiunile persistente ale clienților. Aceasta include stocarea mesajelor pierdute și a abonamentelor, permițând recuperarea datelor în cazul în care un client se reconectează după o întrerupere. Dacă un client abonat pierde contactul cu brokerul, acesta va stoca mesajele într-un buffer și le va retransmite odată ce conexiunea este restabilită. Astfel, brokerul asigură continuitatea serviciului și protecția împotriva pierderii datelor în situații de rețea instabilă.

De asemenea, brokerul are capacitatea de a întrerupe comunicarea cu clienții în anumite situații critice și de a le trimite un mesaj în cache care conține instrucțiuni specifice, de exemplu, dacă clientul de publicare se deconectează brusc. Acest mecanism nu doar că sporește fiabilitatea sistemului, dar și asigură o gestionare eficientă a fluxurilor de mesaje, adaptându-se dinamic la modificările de stare ale clienților. Așadar, brokerul MQTT este inima oricărui sistem de mesagerie bazat pe MQTT, asigurând o distribuție coerentă, fiabilă și scalabilă a mesajelor între numeroase dispozitive, în condiții de rețea variabile și cu un nivel ridicat de persistenta a datelor.

### 9.2.4. Subiect (*topic*)

În MQTT, subiectele (*topics*) reprezintă elemente fundamentale care permit filtrarea și direcționarea mesajelor către clienții abonați. Fiecare client publică mesaje către broker pe un anumit subiect, iar brokerul, care acționează ca un server central, filtrează mesajele în funcție de aceste subiecte, distribuind doar mesajele relevante către clienții care s-au abonat la subiectele respective (Fig. 9.3).


**Fig. 9.3.** Subiectul mesajului MQTT.

Subiectele sunt reprezentate ca șiruri de caractere în format UTF-8 și pot fi împărțite în mai multe niveluri, separate de o bară oblică ("/"), ceea ce permite organizarea ierarhică a datelor. Acest mod de structurare conferă flexibilitate în clasificarea informațiilor, facilitând atât operațiunile de publicare, cât și cele de abonare.

Fiecare nivel dintr-un subiect este semnificativ, iar subiectele sunt sensibile la majuscule și minuscule, ceea ce înseamnă că "senzor/temperatura" și "Senzor/Temperatura" sunt tratate ca subiecte complet diferite. Această caracteristică impune o convenție strictă de denumire pentru a evita confuziile și a asigura o gestionare corectă a mesajelor. Structura ierarhică a subiectelor permite crearea unor scheme complexe de organizare, unde, de exemplu, se pot defini subiecte generale, precum "senzor", și subiecte mai specifice, cum ar fi "senzor/temperatura" sau "senzor/umiditate". Acest sistem facilitează integrarea și gestionarea eficientă a comunicațiilor într-un mediu IoT sau într-o rețea de comunicații distribuite, permițând aplicațiilor să preia și să proceseze doar informațiile de care au cu adevărat nevoie.

## 9.3. Mesajul MQTT și calitatea serviciului (QoS)

### 9.3.1. Format mesaj

Protocolul MQTT utilizează un mecanism robust de comenzi și confirmări, similar cu mecanismul *handshake* din TCP, pentru a asigura că fiecare operațiune de comunicare este realizată cu succes între client și broker. Astfel, atunci când un client trimite o comandă, de exemplu o cerere de conectare (CONNECT), brokerul răspunde cu un mesaj de confirmare (CONNACK) care confirmă stabilirea conexiunii, așa cum este prezentat în Fig. 9.4.

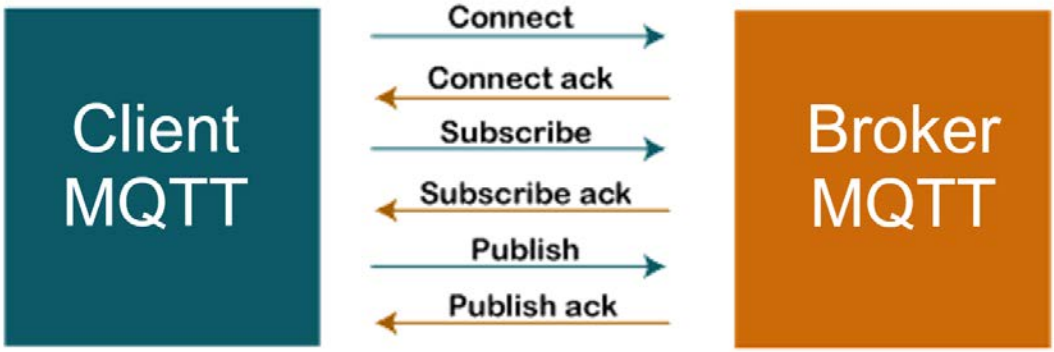



**Fig. 9.4.** Tipuri de mesaje MQTT.

În mod similar, comanda de abonare (SUBSCRIBE) este urmată de un mesaj SUBACK, iar în cazul publicării unui mesaj (PUBLISH), brokerul trimite un mesaj de confirmare (PUBACK pentru QoS 1 sau o serie de mesaje pentru QoS 2), garantând livrarea fiabilă a mesajelor.

Acest mecanism de confirmare joacă un rol esențial în asigurarea integrității și fiabilității comunicării, permițând clientului să verifice dacă operațiunile solicitate au fost executate cu succes. Dacă nu se primește confirmarea așteptată, clientul poate opta pentru retransmiterea mesajului, ceea ce este vital în medii cu lățime de bandă redusă sau conexiuni instabile. În plus, acest sistem ajută la gestionarea eficientă a resurselor și la implementarea strategiilor de retransmisie, sporind astfel reziliența întregului sistem de mesagerie.

Formatul mesajului MQTT este conceput pentru a fi compact și eficient, permițând transmiterea rapidă a datelor chiar și în medii cu resurse limitate. Un mesaj MQTT este alcătuit din trei componente principale (Fig. 9.5). Primul element este un antet fix de 2 octeți, prezent în toate pachetele, care conține informații esențiale precum tipul mesajului și lungimea pachetului, permițând identificarea rapidă a operațiunii (de exemplu, CONNECT, PUBLISH, SUBSCRIBE). Acest antet fix este fundamental pentru ca atât clienții, cât și brokerul să poată interpreta corect mesajul și să stabilească contextul operației.

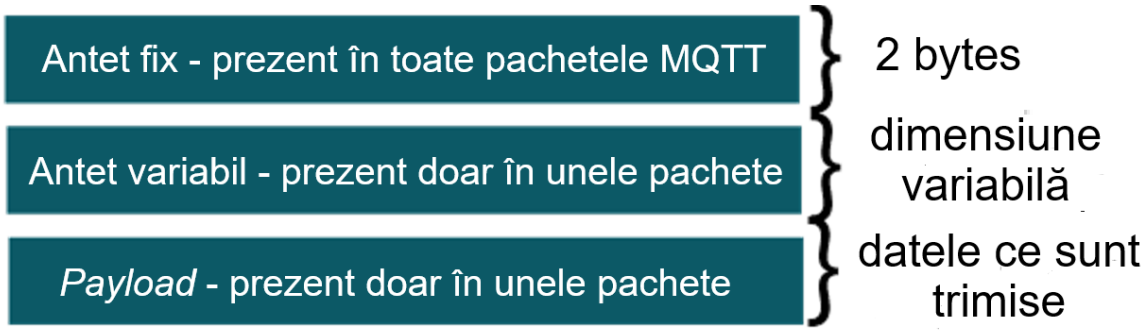



**Fig. 9.5.** Componentele mesajului MQTT.

Al doilea element din formatul mesajului este antetul variabil, care nu este întotdeauna prezent și variază în funcție de tipul de mesaj. Acest antet poate include informații suplimentare precum identificatori de mesaj, nivelul de calitate al serviciului (QoS) sau alte setări specifice fiecărei operațiuni. Flexibilitatea antetului variabil permite protocolului MQTT să se adapteze la diferite cerințe, fără a impune un *overhead* inutil în situațiile în care nu sunt necesare informații suplimentare.

Al treilea element, numit sarcina utilă (*payload*), reprezintă conținutul efectiv al mesajului, adică datele transmise între clienți prin intermediul brokerului. Sarcina utilă poate conține orice tip de informație, de la text și JSON până la date binare, în funcție de natura aplicației. Este important de menționat că, deși majoritatea mesajelor MQTT includ un payload, anumite comenzi, cum ar fi mesajul de deconectare (DISCONNECT), nu necesită acest câmp, ceea ce permite o optimizare suplimentară a traficului. Acest design modular al mesajului asigură un

echilibru între flexibilitate și eficiență, permițând protocolului să funcționeze la parametri optimi în diverse scenarii de comunicare, esențiale pentru implementările în IoT.

### 9.3.2. Calitatea Serviciului (QoS)

MQTT oferă trei niveluri de calitate a serviciului (QoS) care asigură diferite garanții de livrare a mesajelor, adaptându-se la necesitățile specifice ale aplicațiilor. La QoS 0 (Cel mult o dată), mesajul este trimis o singură dată, fără a se efectua verificări suplimentare pentru confirmarea livrării, ceea ce poate fi potrivit pentru situațiile în care pierderea unui mesaj ocazional nu afectează semnificativ funcționalitatea sistemului. La QoS 1 (Cel puțin o dată), mesajul este retransmis de către expeditor până când se primește o confirmare, garantând că mesajul ajunge la destinație, însă poate duce la duplicarea mesajelor. La QoS 2 (Exact o dată), se realizează un mecanism de tip *handshake* pe două niveluri între expeditor și destinatar pentru a se asigura că mesajul este livrat exact o singură dată, asigurând astfel cea mai mare fiabilitate, deși la un cost de complexitate și latență mai ridicate.

Pe lângă nivelurile QoS, MQTT oferă și funcționalitatea de mesaje reținute, care permite stocarea ultimului mesaj publicat pe un anumit subiect. Această caracteristică este esențială pentru actualizările de configurare și pentru situațiile în care noii abonați trebuie să primească imediat cel mai recent mesaj, fără a aștepta o nouă publicare. Astfel, brokerul reține mesajul și îl transmite automat tuturor clienților care se abonează ulterior la acel subiect.

În ceea ce privește securitatea, MQTT se bazează pe mecanisme robuste de criptare și autentificare, utilizând adesea protocolul TLS (Transport Layer Security) pentru a proteja comunicațiile. Acest lucru include autentificarea cu nume de utilizator și parolă, precum și utilizarea certificatelor, asigurând astfel confidențialitatea, integritatea și autenticitatea datelor transmise. Aceste capabilități de securitate sunt esențiale, mai ales în mediile IoT, unde datele pot fi sensibile și este crucial să se prevină accesul neautorizat sau interceptarea acestora.

Prin urmare, nivelurile de QoS, mesajele reținute și măsurile de securitate fac din MQTT un protocol de înaltă eficiență și fiabilitate, capabil să asigure transmiterea datelor în condiții optime chiar și în rețele cu lățime de bandă limitată și latente ridicate.

## 9.4. Avantaje și dezavantaje MQTT

MQTT oferă avantaje semnificative în contextul comunicațiilor pentru dispozitivele IoT, datorită designului său extrem de ușor și eficient. Datorită faptului că mesajele MQTT au o amprentă foarte mică – antetul fix de doar 2 octeți și o dimensiune a sarcinii utile care poate ajunge până la 256 megaocteți – protocolul minimizează utilizarea lățimii de bandă. Această caracteristică este esențială pentru rețelele cu resurse limitate, unde fiecare octet contează, asigurând astfel transmiterea rapidă a datelor și reducând semnificativ costurile de operare.

Un alt avantaj major al MQTT este modelul său de publicare/abonare, care permite dispersia eficientă a datelor între numeroși clienți, fără ca aceștia să fie strâns cuplați între ei. Prin intermediul unui broker central, mesajele sunt distribuite doar către clienții care s-au abonat la subiectele relevante, ceea ce reduce traficul inutil pe rețea și îmbunătățește eficiența generală a comunicațiilor. Această abordare permite, de asemenea, implementarea unor mecanisme de retransmisie și gestionare a erorilor, asigurând o livrare promptă și fiabilă a mesajelor.

De asemenea, MQTT este proiectat pentru a minimiza consumul de energie, un aspect esențial pentru dispozitivele IoT care operează pe baterii sau în medii cu resurse limitate. Prin reducerea dimensiunii pachetelor de date și prin optimizarea fluxului de mesaje, MQTT contribuie la prelungirea duratei de viață a dispozitivelor conectate, maximizând capacitatea rețelei de a gestiona un număr mare de conexiuni simultane. În plus, caracteristicile sale de teledetecție și control permit monitorizarea în timp real și gestionarea eficientă a sistemelor distribuite, facilitând astfel implementarea rapidă și scalabilă a aplicațiilor în medii industriale și de consum.

În comparație cu protocolul CoAP (*Constrained Application Protocol*), MQTT prezintă anumite dezavantaje. De exemplu, ciclurile de trimitere ale mesajelor în MQTT sunt, în general, mai lente, ceea ce poate afecta performanța în scenarii în care latența este critică. În plus, mecanismul de descoperire a resurselor în MQTT se bazează pe un sistem de subscriere flexibil la subiecte, care, deși oferă flexibilitate, poate fi mai puțin eficient decât sistemele bazate pe încredere, utilizate de CoAP pentru a identifica și accesa resursele în mod direct.

Un alt aspect de luat în considerare este că, spre deosebire de unele protocoale care integrează criptarea direct în specificație, MQTT nu include criptare nativă. Securitatea mesajelor se realizează prin intermediul TLS/SSL, care, deși asigură confidențialitatea și integritatea datelor, implică o complexitate suplimentară în implementare și administrare. Această dependență de mecanisme externe de criptare poate crește riscul de erori de configurare sau de incompatibilități între diferitele componente ale sistemului.

În plus, construirea unei rețele MQTT scalabile la nivel internațional se dovedește a fi o provocare semnificativă. Gestionarea unui număr mare de clienți distribuiți geografic și asigurarea unei performanțe consistente, în ciuda variabilității infrastructurilor de rețea și a latenței, necesită strategii avansate de orchestration și monitorizare. Astfel, deși MQTT oferă o soluție eficientă pentru multe aplicații IoT, aceste dezavantaje pot constitui obstacole în implementările care necesită performanță extremă sau o scalabilitate globală robustă.

HTTP se dovedește a fi ineficient pentru scenariile tipice în IoT, unde se transmit pachete mici de date. Fiind centrat pe documente și se bazează pe un format textual, ceea ce face ca anteturile sale să fie relativ mari (până la 1KB), iar analiza și parsarea lor să fie mai complexe și consumatoare de resurse. În contrast, MQTT este un protocol binar, proiectat special pentru a transmite mesaje de dimensiuni reduse, având anteturi de doar 2-4 octeți pentru mesajele

de tip PUBLISH și 14 octeți pentru mesajele CONNECT, ceea ce contribuie la o procesare rapidă și eficientă.

Modelul MQTT se bazează pe un sistem de tip publicare-abonare, permițând distribuția datelor în mod unul-la-mulți, unde un singur mesaj este livrat simultan către toți abonații interesați de un anumit subiect. Această abordare se deosebește de HTTP, care, de regulă, permite doar comunicații unu-la-unu, limitând eficiența în medii distribuite, unde sunt implicate numeroase dispozitive. În plus, protocolul MQTT utilizează mai puțină energie a bateriei, ceea ce este esențial pentru dispozitivele IoT, și, deși consumă mai multă energie la inițierea conexiunii, câștigurile obținute prin menținerea conexiunii deschise compensează acest cost inițial.

În termeni de performanță, MQTT permite transmiterea rapidă a mesajelor datorită formatului său compact, ceea ce îl face ideal pentru aplicații care necesită un flux continuu de date, cum ar fi sistemele de monitorizare și control în timp real. De exemplu, o implementare a unui client MQTT poate fi de până la 30KB, mult mai eficientă decât un client HTTP cu un antet voluminos. Totodată, MQTT asigură o livrare fiabilă a mesajelor prin intermediul nivelurilor de calitate a serviciului.

Principalele diferențe dintre HTTP și MQTT sunt prezentate în Tabelul 9.1.

**Tabel 9.1.** Comparație performanță între HTTPS și MQTT.

| | 3G | | Wi-fi | |
|---|---|---|---|---|
| | HTTPS | MQTT | HTTPS | MQTT |
| % Baterie / oră | 18.43% | **16.13%** | **3.45%** | 4.23% |
| Mesaje / oră | 1708 | **160278** | 3628 | **263314** |
| % Baterie / mesaj | 0.01709 | **0.00010** | 0.00095 | **0.00002** |
| Mesaje primite | 240/1024 | **1024/1024** | 524/1024 | **1024/1024** |

Astfel, MQTT se dovedește a fi o soluție robustă și economică pentru transportul de date în rețelele IoT, oferind performanță ridicată, eficiență energetică și flexibilitate în gestionarea comunicațiilor. Aceasta face ca protocolul să fie preferat nu doar pentru aplicații de la scară mică, ci și pentru sisteme complexe care necesită o interacțiune rapidă și fiabilă între mii de dispozitive.

Alte protocoale, cum ar fi AMQP, CoAP, XMPP, DDS și STOMP, pot fi, de asemenea, comparate cu MQTT în funcție de cerințele specifice ale aplicațiilor, însă MQTT se remarcă prin adaptabilitatea sa în medii cu lățime de bandă redusă și latență ridicată.

## 9.5. Exemplu MQTT

În exemplul de implementare din Fig. 9.6 este prezentat modul în care un dispozitiv cu un senzor de temperatură trimite valoarea măsurată către un broker, iar apoi această valoare este transmisă către alte aplicații interesate, cum ar fi un telefon mobil sau o aplicație desktop. Inițial, dispozitivul acționează ca un editor (*publisher*), definind un subiect specific – de exemplu, „*temperatura*" – și publicând mesajul ce conține valoarea actuală a temperaturii. Acest mesaj este apoi trimis către broker, care se ocupă de filtrarea și stocarea temporară a datelor.

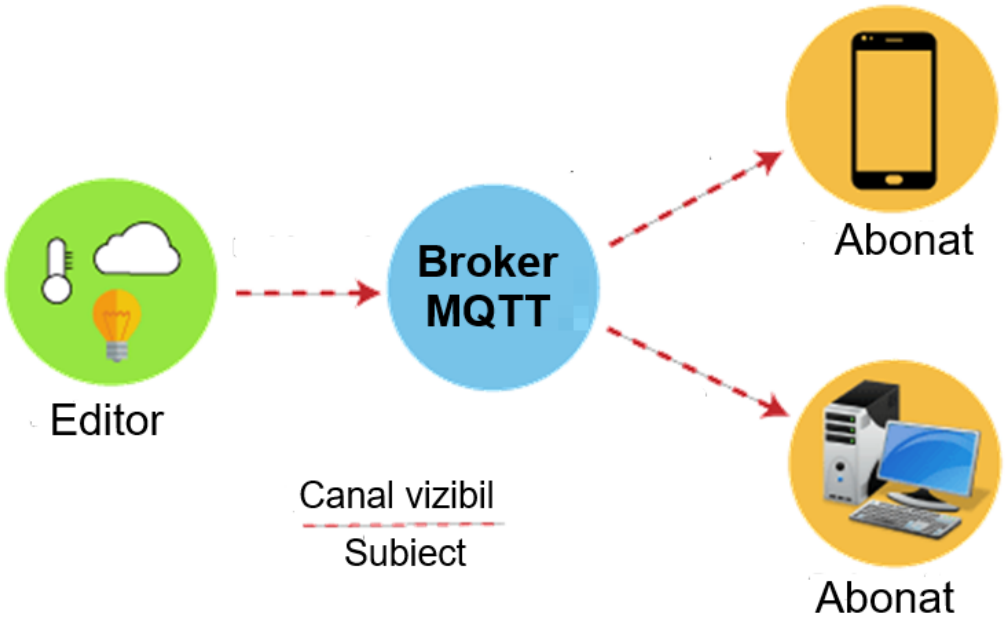



**Fig. 9.6.** Exemplu de implementare MQTT.

După ce mesajul a fost publicat, alte dispozitive sau aplicații care doresc să primească aceste informații se abonează la același subiect, „*temperatura*". Astfel, aplicația de pe telefon sau desktop acționează ca un abonat (*subscriber*) și, odată ce s-a înregistrat pentru subiectul respectiv, primește mesajul publicat de editor. Această abordare decuplată, bazată pe modelul de publicare/abonare, asigură că editorul și abonații nu trebuie să cunoască direct identitatea unul altuia, ci comunică prin intermediul brokerului.

Fig. 9.7 ilustrează o secvență tipică a interacțiunilor MQTT cu nivel de calitate a serviciului setat la 0, începând cu etapa de conectare a unui client la broker (mesajele CONNECT și răspunsul CONNACK). După stabilirea conexiunii, clientul B trimite o cerere de abonare la un subiect specific, de exemplu „*temperature/roof*", iar brokerul acceptă cererea. Ulterior, clientul B publică un mesaj pe același subiect, incluzând și flag-ul *retain*. Acest flag face ca brokerul să stocheze mesajul, astfel încât noii abonați care se conectează ulterior și se abonează la același subiect să primească imediat cel mai recent mesaj publicat.

În continuare, diagrama prezintă alte mesaje PUBLISH ce conțin valori de temperatură pentru subiectele „*temperature/roof*" și „*temperature/floor*", fiecare livrată fără mecanisme suplimentare de confirmare, caracteristic QoS 0 (livrare „cel mult o dată"). În cele din urmă, clientul B trimite un mesaj DISCONNECT, semnalând încheierea conexiunii cu brokerul. Întreaga secvență evidențiază modul în care MQTT gestionează abonarea, publicarea și păstrarea ultimului mesaj prin flag-ul retain, asigurând că noii abonați pot primi informația cea mai recentă imediat după conectare.

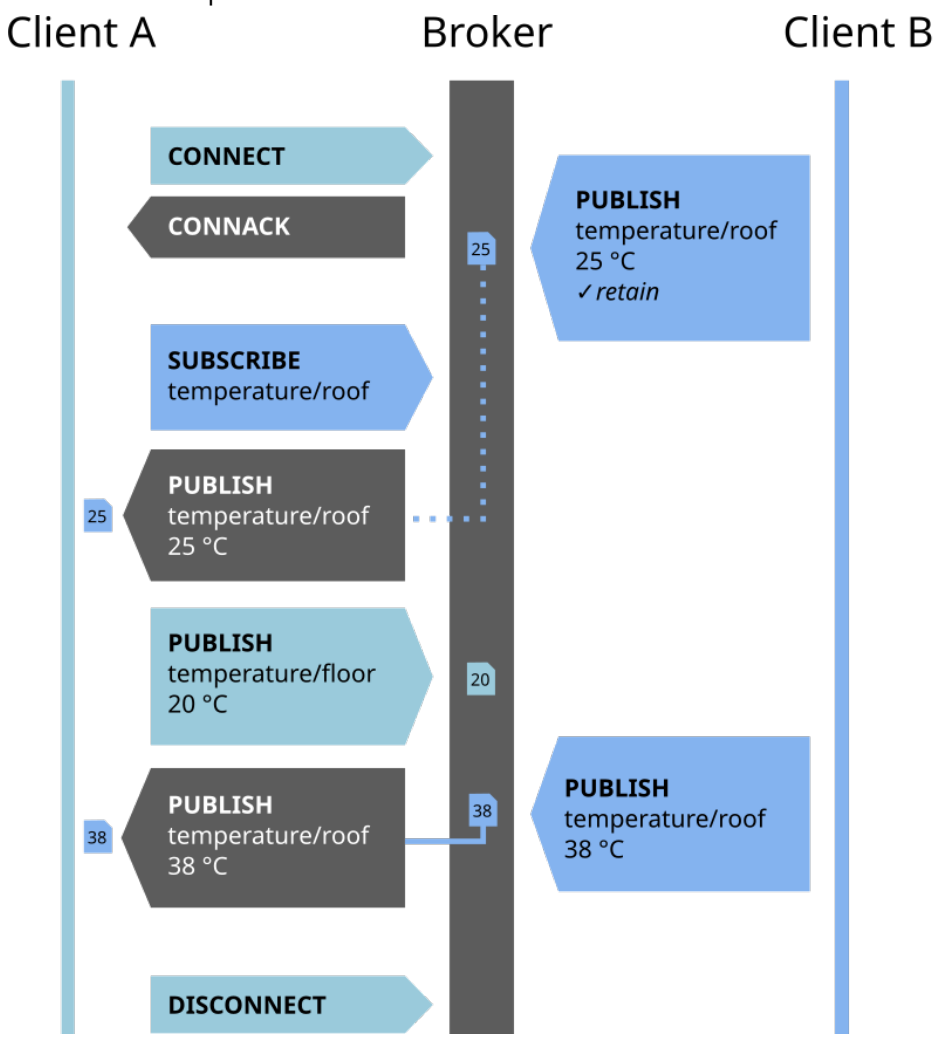



**Fig. 9.7.** Exemplu de conexiune MQTT.

# 10.Capitolul 10. Cloud Computing. Servicii și furnizori în Cloud

Cloud computing reprezintă un model revoluționar de furnizare a resurselor IT, care permite accesul la infrastructură, platforme și software prin intermediul internetului, la cerere și într-un mod flexibil. Prin această paradigmă, organizațiile pot reduce semnificativ costurile operaționale, deoarece nu mai este necesară achiziționarea și întreținerea de hardware dedicat. Resursele sunt scalabile și pot fi alocate în funcție de necesități, ceea ce permite o adaptare rapidă la fluctuațiile cererii și o gestionare eficientă a fluxului de lucru.

Revoluția cloud computing se datorează capacității sale de a transforma modul în care serviciile IT sunt concepute, furnizate și consumate. Prin eliminarea barierelor tradiționale de infrastructură, cloud-ul oferă acces la tehnologii avansate precum inteligența artificială, Big Data și IoT, integrându-le într-un mediu unificat și ușor de administrat. Această abordare nu doar că facilitează accesul la tehnologie, dar stimulează inovația și agilitatea în afaceri, permițând companiilor să se concentreze pe dezvoltarea produselor și serviciilor, în loc să investească în hardware și administrarea unor infrastructuri complexe.

## 10.1. Proprietăți Cloud Computing

Cloud computing se remarcă printr-o serie de proprietăți fundamentale care transformă modul în care companiile își gestionează infrastructura IT, permițându-le să răspundă rapid și eficient la cerințele variabile ale afacerilor moderne. Scalabilitatea reprezintă capacitatea unui sistem de a face față creșterii volumului de date sau a numărului de solicitări, prin adăugarea dinamică a resurselor de calcul, stocare și rețea. În mediile cloud, această scalabilitate este adesea implementată în mod automat, astfel încât sistemele se extind sau se contractă în funcție de cererea curentă, fără a necesita intervenție manuală. Acest lucru nu doar optimizează utilizarea resurselor hardware, dar și reduce costurile.

Pe lângă scalabilitate, elasticitatea este o caracteristică cheie a cloud computing-ului, care permite infrastructurii să se adapteze în timp real la fluctuațiile cererii. Elasticitatea presupune monitorizarea continuă a performanței sistemului și ajustarea dinamică a resurselor, fie prin adăugarea rapidă de instanțe suplimentare, fie prin reducerea numărului de resurse atunci când sarcina scade. Această capacitate de adaptare este crucială în medii unde volumul de date și numărul de utilizatori pot varia brusc.

Disponibilitatea este o altă proprietate esențială a sistemelor cloud. Aceasta se referă la gradul în care un serviciu sau o resursă este accesibilă și funcțională în orice moment, cu un nivel minim adesea garantat la 99,9% sau chiar mai mult. Pentru multe aplicații critice, cum ar fi cele din domeniul bancar sau al sănătății, disponibilitatea ridicată este vitală, iar platformele

cloud sunt proiectate să ofere redundanță și reziliență, astfel încât, în caz de defecțiuni, serviciile să fie redirecționate automat către instanțe alternative fără întreruperi majore.

În plus, încrederea(*reliability*) joacă un rol critic în cloud computing, referindu-se la capacitatea unui sistem de a funcționa conform specificațiilor sale pentru perioade îndelungate, fără erori sau întreruperi. Măsurile de securitate, backup-urile automate și strategiile de recuperare în caz de dezastru sunt integrate în majoritatea soluțiilor cloud pentru a asigura că datele și aplicațiile sunt protejate împotriva pierderii sau coruperii. Fiabilitatea ridicată a serviciilor cloud permite companiilor să se bazeze pe acestea pentru procesele critice, având încredere că operațiunile vor continua fără probleme, chiar și în condiții de stres sau atacuri cibernetice.

Cloud computing se remarcă printr-o administrare flexibilă și eficientă, caracteristică esențială la nivel enterprise. Sistemele de tip cloud sunt proiectate pentru a facilita managementul centralizat al resurselor, permițând monitorizarea, configurarea și actualizarea rapidă a infrastructurii, indiferent de volumul de date sau de numărul de utilizatori. Această flexibilitate ajută organizațiile să adapteze rapid resursele la cerințele de afaceri și să optimizeze operațiunile IT, reducând costurile și timpul de administrare.

Interoperabilitatea este o altă proprietate crucială a cloud computing-ului, asigurând că sistemele dispun de interfețe standardizate ce permit interacțiunea cu alte platforme și servicii, atât în prezent, cât și pe viitor. Acest aspect minimizează blocajele tehnologice și restricțiile de acces, facilitând integrarea diverselor aplicații și sisteme, fără a fi necesară o adaptare complexă a codului sau infrastructurii. Prin adoptarea unor standarde deschise, interoperabilitatea sprijină o colaborare eficientă între diferite medii IT.

Performanța și optimizarea reprezintă piloni importanți în cloud computing. Tehnologiile moderne permit procesarea paralelă pe sisteme multi-core și calcul distribuit, ceea ce crește semnificativ viteza de execuție a sarcinilor. Tehnica de *load balancing*, care distribuie uniform încărcarea între două sau mai multe instanțe, asigură utilizarea optimă a resurselor și previne supraîncărcarea sistemului.

Accesibilitatea este, de asemenea, o caracteristică definitorie a cloud computing-ului, descriind gradul în care serviciile și resursele sunt disponibile pentru un număr mare de clienți, oriunde s-ar afla aceștia. Datorită arhitecturii distribuite și a infrastructurilor globale de cloud, utilizatorii pot accesa serviciile de pe orice dispozitiv, fie că este vorba de PC-uri, laptopuri, tablete sau smartphone-uri.

Portabilitatea reprezintă capacitatea de a accesa un serviciu folosind orice dispozitiv și de oriunde, fără a se depinde de un anumit mediu sau sistem de operare. Această caracteristică este esențială în contextul cloud computing-ului, deoarece permite dezvoltatorilor și utilizatorilor să ruleze aplicații în mod uniform pe diverse platforme, fie că acestea sunt găzduite pe servere locale, în centre de date sau în cloud public sau privat. În concluzie, prin combinarea acestor proprietăți – flexibilitate, interoperabilitate, performanță, accesibilitate și

portabilitate – cloud computing a transformat radical modul în care organizațiile își gestionează resursele IT, oferind soluții scalabile, eficiente și ușor de integrat în orice mediu.

## 10.2. Servicii Cloud Computing

Cloud computing oferă o gamă variată de servicii care pot fi alese în funcție de necesitățile și resursele disponibile ale unei organizații, de la infrastructură până la aplicații complete. Utilizatorii pot alege între diferite modele de servicii cloud, fiecare oferind un nivel diferit de control și responsabilitate.

Fig. 10.1 prezintă o analogie simplă între modul în care se construiește și se utilizează o locuință și cele trei modele de servicii cloud – Infrastructure as a Service (IaaS), Platform as a Service (PaaS) și Software as a Service (SaaS).


**Fig. 10.1.** Serviciile Cloud.

Construirea unei case de la zero se aseamănă cu IaaS. În acest model, proprietarul are control complet asupra infrastructurii: alege terenul, materialele de construcție și modul în care este realizată structura. Similar, în IaaS, se închiriază infrastructură virtualizată (servere, stocare, rețea) și se construiește sistemul IT conform preferințelor. Deși oferă flexibilitate maximă, presupune și responsabilități ridicate pentru gestionarea și întreținerea infrastructurii.

O casă cumpărată „la roșu” reprezintă o construcție parțial finalizată, unde pereții, acoperișul și structura de bază sunt deja realizate. Proprietarul se ocupă doar de finisaje și amenajări. În mod similar, PaaS oferă o platformă de dezvoltare deja pregătită – cu sistem de operare, baze de date, middleware – peste care dezvoltatorii își pot implementa aplicațiile. Astfel, sarcinile de administrare la nivel de infrastructură sunt preluate de furnizorul de cloud, lăsând echipelor de dezvoltare libertatea de a se concentra pe logica de business.

A locui într-un hotel este asemănător cu SaaS. Utilizatorul nu trebuie să se ocupe de construcție, întreținere sau reparații; pur și simplu folosește serviciile oferite (cameră, restaurant, curățenie) și plătește pentru perioada de ședere. La fel, în SaaS, aplicația este complet furnizată de un furnizor extern, iar clienții o accesează fără să fie nevoiți să se ocupe de instalarea, actualizarea sau administrarea ei. Ei plătesc doar pentru utilizarea serviciului și beneficiază de toate facilitățile, fără a avea responsabilități tehnice.

Consumatorii de servicii cloud au nevoie de un mediu robust pentru dezvoltare și testare, unde aplicațiile și serviciile pot fi rulate într-un mediu controlat, scalabil și sigur. Ei solicită, de asemenea, un mecanism automatizat de distribuire și management al job-urilor, care să asigure execuția rapidă și eficientă a sarcinilor, indiferent de volumul de date implicat. Acest tip de mediu le permite să se concentreze pe dezvoltarea aplicațiilor, fără a fi nevoiți să gestioneze complexitatea infrastructurii.

Pe lângă acestea, consumatorii necesită un sistem solid de control al accesului și autentificare, pentru a proteja resursele sensibile împotriva accesului neautorizat. Într-o eră în care cerințele de procesare și stocare cresc exponențial, este esențial să se poată accesa cantități mari de resurse în mod flexibil, în funcție de necesități. Modelul PaaS răspunde acestor cerințe oferind un mediu integrat care include instrumente de dezvoltare, testare, gestionare automată a job-urilor și control al accesului.

Fig. 10.2 ilustrează responsabilitățile utilizatorului și ale furnizorului de servicii în diferite modele de furnizare a resurselor IT. În modelul tradițional (*on-premise*), organizația (utilizatorul) se ocupă integral de toate nivelurile – de la rețea, stocare și servere, până la sistemul de operare, middleware, date și aplicații. În IaaS, furnizorul administrează componentele de bază (rețea, stocare, servere, virtualizare), în timp ce utilizatorul își menține controlul asupra sistemului de operare, mediului de execuție, datelor și aplicațiilor. În PaaS, responsabilitățile furnizorului se extind și la nivelul sistemului de operare, al mediului de execuție și al middleware-ului, lăsându-i clientului doar gestionarea datelor și a aplicațiilor propriu-zise. Acest model stratificat evidențiază gradul de implicare al utilizatorului în fiecare tip de serviciu cloud și arată cum, pe măsură ce se trece de la on-premise la PaaS, furnizorul preia tot mai multe sarcini de administrare a infrastructurii.

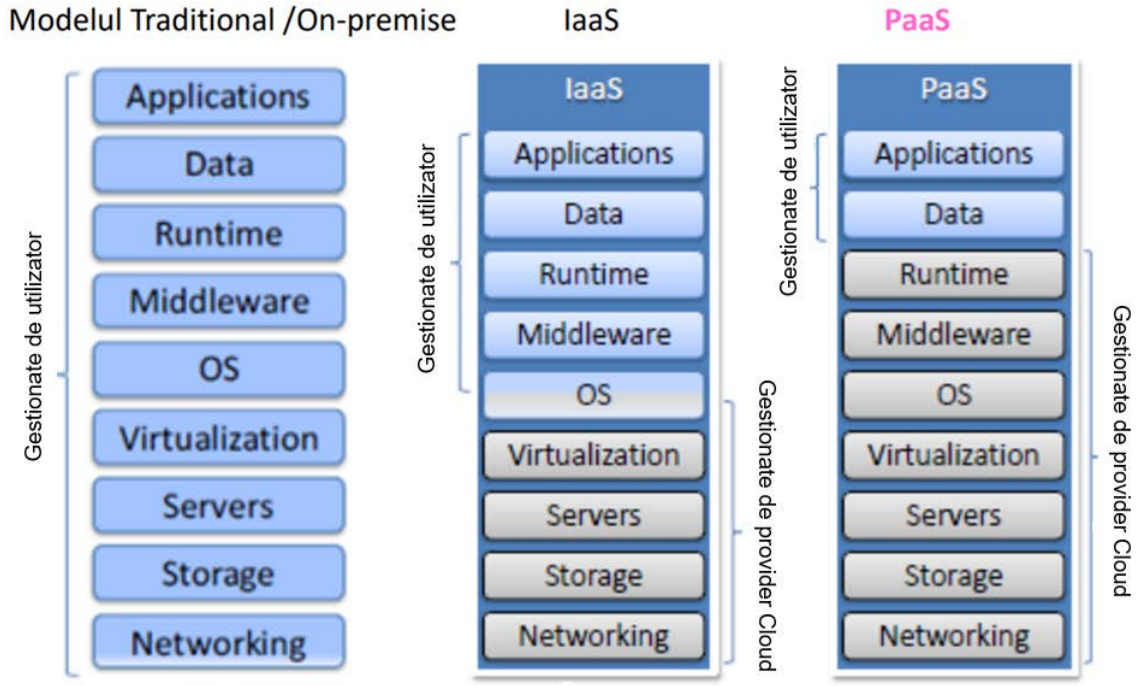


**Fig. 10.2.** Comparație On-premise vs Cloud.

În concluzie, alegerea între IaaS, PaaS și SaaS depinde de nivelul de control și responsabilitate dorit de organizație. IaaS oferă flexibilitate maximă și control complet asupra mediului IT, fiind ideal pentru companiile care doresc să gestioneze fiecare aspect al infrastructurii. PaaS reduce sarcina administrativă, permițând dezvoltatorilor să se concentreze pe dezvoltarea aplicațiilor, în timp ce SaaS oferă soluții gata de utilizare, eliminând necesitatea unei administrări tehnice.

### 10.2.1. Infrastructure as a Service (IaaS)

IaaS (Infrastructure as a Service) reprezintă o paradigmă în furnizarea de resurse IT, oferind organizațiilor acces la infrastructură virtualizată – mașini virtuale, stocare, rețea și sisteme de securitate – fără a fi nevoie de investiții majore în hardware. În esență, IaaS se bazează pe o platformă de găzduire, numită "*fabric*", care include echipamentele fizice și software-ul de virtualizare ce permite abstractizarea acestora. Acest lucru înseamnă că resursele fizice, cum ar fi serverele, sistemele de operare, echipamentele de rețea și soluțiile de stocare, sunt puse la dispoziția consumatorilor sub forma unor entități virtuale, permițând scalarea rapidă și flexibilă a mediului IT.

Unul dintre avantajele majore ale IaaS este controlul pe care îl oferă consumatorilor asupra resurselor virtualizate. Chiar dacă accesul la nivelul fizic al "*fabricului*" este restricționat, utilizatorii pot administra sistemele de operare, configura stocarea, dezvolta și implementa aplicații și ajusta parametrii rețelei în funcție de necesități. Această flexibilitate le permite să își personalizeze mediul IT și să răspundă rapid la cerințele dinamice ale afacerii, fără a fi necesară o intervenție manuală asupra hardware-ului. Mai mult, tehnologia cheie care stă la baza IaaS este virtualizarea, care asigură o izolare completă între resursele individuale și permite utilizarea optimă a resurselor hardware, reducând costurile și sporind eficiența operațională.

Furnizorii de IaaS, cum ar fi Amazon Web Services (AWS) cu Elastic Compute Cloud (EC2) și Simple Storage Service (S3), Microsoft Azure și Google Compute Engine, oferă soluții robuste și scalabile, adaptate nevoilor de procesare, stocare și comunicație ale companiilor moderne. Aceste platforme permit administrarea centralizată a resurselor și implementarea rapidă a aplicațiilor, fără a fi nevoie de investiții în infrastructura fizică. Mai mult, modelul IaaS sprijină inovația prin posibilitatea de a integra tehnologii avansate, cum ar fi inteligența artificială, analiza Big Data și containerizarea.

Prin urmare, IaaS transformă modul în care organizațiile își construiesc și își administrează infrastructura IT, oferind un nivel ridicat de control, flexibilitate și eficiență. Această abordare permite companiilor să se concentreze pe dezvoltarea de aplicații și servicii

inovatoare, reducând în același timp costurile operaționale și complexitatea administrativă asociată cu gestionarea hardware-ului propriu.

### 10.2.2. Platform as a Service (PaaS)

Platform as a Service (PaaS) reprezintă o soluție cloud destinată în mod special dezvoltatorilor software, oferind un mediu de dezvoltare complet gestionat și standardizat, fără a fi nevoie ca aceștia să se preocupe de administrarea infrastructurii hardware. În cadrul modelului PaaS, consumatorul are acces doar la mediul de dezvoltare și la instrumentele pentru implementarea aplicațiilor, fără a avea control direct asupra rețelei, serverelor, sistemului de operare sau stocării. Astfel, dezvoltatorii se pot concentra exclusiv pe scrierea codului și pe optimizarea funcționalităților aplicațiilor, profitând de un cadru care asigură scalabilitate și eficiență operațională.

PaaS oferă o interfață standardizată și un set de instrumente integrate care permit gestionarea întregului ciclu de viață al aplicației, de la dezvoltare la testare, implementare și monitorizare. Acest model este conceput astfel încât toate resursele necesare pentru rularea aplicațiilor – cum ar fi platformele de baze de date, middleware-ul, serviciile de securitate și soluțiile de integrare – sunt preconfigurate și administrate de furnizorul de cloud.

Un alt avantaj important al modelului PaaS este că acesta permite dezvoltatorilor să beneficieze de suportul tehnologic al unor furnizori de top, care dispun de infrastructuri robuste și scalabile, precum Google AppEngine, Azure App Services sau Azure Spring Apps. Aplicațiile dezvoltate pe aceste platforme rulează folosind infrastructura furnizorilor, garantând astfel o performanță optimă și o disponibilitate ridicată. În plus, PaaS oferă posibilitatea de a integra rapid tehnologiile emergente, cum ar fi microserviciile, containerizarea și analiza datelor.

Structura unui mediu PaaS se descompune în trei componente cheie care colaborează pentru a oferi un mediu de dezvoltare și implementare complet gestionat. În primul rând, Resource Pool permite abstractizarea și controlul dinamic al resurselor. Acest strat oferă capacitatea de a aloca sau elibera automat resursele (procesor, memorie, stocare) în funcție de cerințele aplicațiilor, asigurând astfel scalabilitatea și flexibilitatea necesare în mediile moderne.

În al doilea rând, Core Platform furnizează funcționalitățile de bază ale mediului PaaS, preluând responsabilitățile legate de configurarea, managementul și întreținerea mediului de rulare. Acest strat automatizat permite dezvoltatorilor să se concentreze exclusiv pe crearea serviciilor și a aplicațiilor, fără a fi nevoie să se implice în administrarea infrastructurii. Astfel, mediul de execuție rămâne consistent și stabil, iar complexitatea operațională este redusă semnificativ.

În final, Enabling Services completează întregul sistem oferind interfețe și instrumente esențiale pentru procesul de dezvoltare, cum ar fi IDE-uri, interfețe de control al sistemului și

servicii de integrare. Aceste servicii permit dezvoltatorilor să interacționeze ușor cu mediul PaaS, facilitând o dezvoltare rapidă și colaborativă, menținând în același timp un nivel ridicat de interoperabilitate și eficiență.

Astfel, PaaS reprezintă o alegere strategică pentru utilizatorii care doresc să dezvolte și să implementeze aplicații inovatoare fără a se confrunta cu complexitatea administrării infrastructurii hardware. Prin oferirea unui mediu de dezvoltare complet gestionat și a unor instrumente standardizate, PaaS facilitează concentrarea pe inovație, creșterea eficienței operaționale și reducerea costurilor, contribuind astfel la transformarea digitală și la accelerarea proceselor de dezvoltare software moderne.

### 10.2.3. Software as a service (SaaS)

SaaS (Software as a Service) reprezintă nivelul cel mai vizibil al cloud computing-ului pentru utilizatorii finali, oferind aplicații software expuse ca interfețe web sau servicii web. Acest model permite consumatorilor să acceseze software-ul de la distanță, folosind orice dispozitiv cu capacități de navigare – fie că este vorba de laptopuri, tablete sau smartphone-uri – fără a avea nevoie să gestioneze sau să cunoască detaliile infrastructurii subiacente. Conform lui Michael Mertz (2007), SaaS reprezintă „*software-ul care este deținut, livrat și administrat de la distanță de către unul sau mai mulți furnizori și care este oferit într-un mod de plată pe utilizare (pay-per-use)*." Această definiție subliniază că, în modelul SaaS, consumatorii accesează aplicații complet gestionate de furnizor, fără a fi nevoiți să se ocupe de aspectele tehnice legate de instalare, configurare și întreținere, plătind doar pentru resursele pe care le folosesc.

Unul dintre principalele avantaje ale modelului SaaS este simplitatea utilizării, deoarece utilizatorii beneficiază de aplicații gata de utilizat. Aceste aplicații sunt în mod obișnuit accesibile prin intermediul unui browser web, ceea ce reduce semnificativ costurile și efortul necesar pentru gestionarea hardware-ului și a software-ului local. Astfel, SaaS facilitează accesul la tehnologii avansate, permițând companiilor și utilizatorilor individuali să se concentreze pe activitățile lor principale, în timp ce furnizorii de cloud se ocupă de gestionarea și actualizarea aplicațiilor.

În plus, utilizatorii SaaS nu sunt interesați de detaliile tehnice privind infrastructura, ci doar de funcționalitatea și experiența oferite de aplicație. Furnizorii de SaaS investesc constant în securitate, scalabilitate și în îmbunătățirea performanței, asigurând că aplicațiile rămân actualizate permanent. Exemple de succes ale modelului SaaS includ Google Apps, care oferă servicii precum Google Mail, Google Drive sau Google Spreadsheets, demonstrând eficiența și beneficiile acestui model.

### 10.2.4. Modele deployment

Cloud computing oferă diferite modele de deployment, fiecare adaptat la nevoile și strategiile organizațiilor moderne. Fig. 10.3 ilustrează modele de deployment în cloud și serviciile pe care le pot oferi.

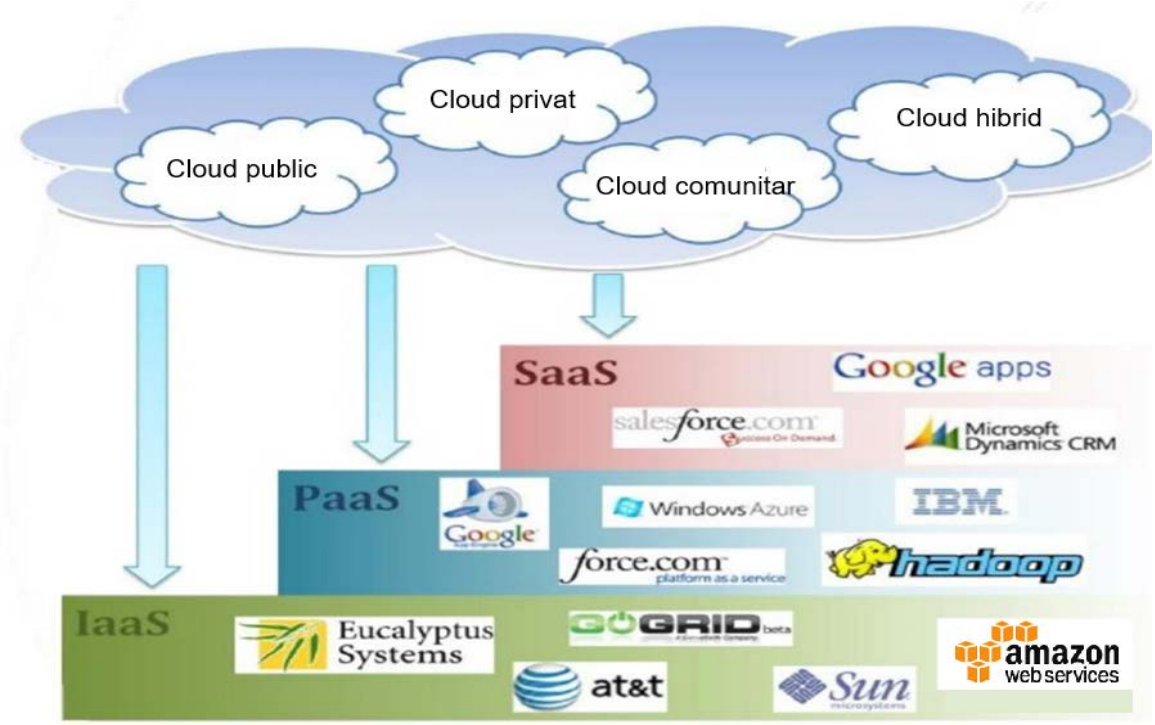



**Fig. 10.3.** Modele de deployment în Cloud.

Unul dintre aceste modele este Cloud-ul Public, definit de IBM ca fiind „*hardware și software din centrele de date administrate de terți, cum ar fi Google și Amazon, care expun serviciile lor companiilor și consumatorilor prin Internet*." În cloud-ul public, infrastructura este complet gestionată de furnizori externi, oferind avantajul reducerii costurilor de capital și accesul rapid la resurse scalabile, însă vine și cu anumite riscuri legate de securitate și control, deoarece datele sunt stocate pe servere administrate de terți.

În contrast, Cloud-ul Privat (sau cloud *on-premise*) se bazează pe virtualizarea infrastructurii deja existente în organizație. Acest model permite companiilor să-și administreze propriile centre de date și să beneficieze de o utilizare mai eficientă a resurselor, reducând în același timp riscurile asociate cu stocarea datelor pe servere publice. Cloud-ul privat oferă un nivel mai ridicat de control și securitate, fiind ideal pentru organizațiile care gestionează informații sensibile sau care trebuie să respecte cerințe stricte de conformitate.

Cloud-ul Hibrid combină avantajele ambelor modele, integrând cloud-uri publice și private într-o infrastructură unificată, dar totuși distinctă. Această abordare permite organizațiilor să păstreze controlul asupra datelor critice în cloud-ul privat, în timp ce beneficiază de scalabilitatea și costurile reduse oferite de cloud-ul public pentru sarcini mai puțin sensibile.

Tehnologiile standardizate asigură portabilitatea datelor și aplicațiilor între aceste medii, facilitând astfel o adaptare dinamică la fluctuațiile de cerere și o gestionare flexibilă a resurselor IT.

Astfel, modelele de deployment cloud – public, privat și hibrid – oferă o gamă variată de opțiuni care permit organizațiilor să-și optimizeze infrastructura în funcție de nevoile specifice, de la costuri reduse și scalabilitate până la control strict și securitate sporită.

## 10.3. Furnizori în Cloud

Cloud computing a revoluționat modul în care organizațiile își administrează infrastructura IT, iar furnizorii de top din industrie joacă un rol esențial în această transformare digitală. Principalii furnizori de servicii cloud – Amazon Web Services (AWS), Microsoft Azure și Google Cloud Platform (GCP) – domină piața globală (Fig. 10.4), oferind soluții complete care acoperă infrastructura, platformele și software-ul ca serviciu. AWS, liderul de piață, deține în jur de 31% din cota globală, urmat de Azure, cu aproximativ 20%, iar GCP se situează în jurul valorii de 12%.

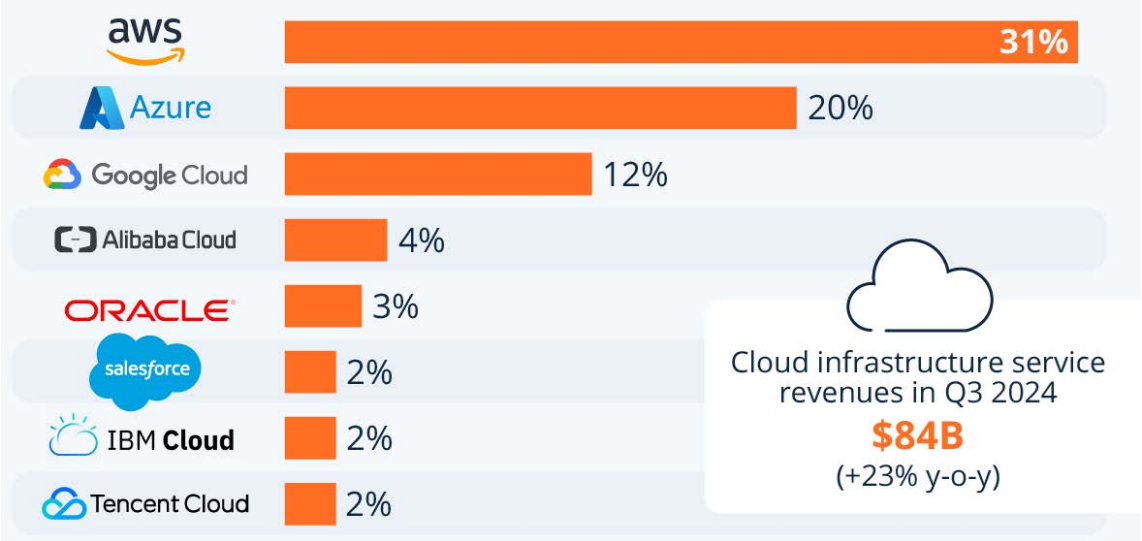



**Fig. 10.4.** Principalii furnizori de servicii Cloud. [25]

### 10.3.1. Amazon Web Services (AWS)

Amazon EC2 face parte din suita de servicii IaaS oferite de Amazon Web Services (AWS) și reprezintă o soluție revoluționară pentru obținerea de capacitate de calcul scalabilă în cloud. Organizațiile pot utiliza EC2 pentru a lansa și administra mașini virtuale care rulează în medii complet izolate, oferindu-le control total asupra sistemului de operare, configurațiilor și aplicațiilor. Prin intermediul API-urilor dedicate, EC2 permite scalarea automată a resurselor, asigurând astfel că aplicațiile se adaptează dinamic la fluctuațiile de trafic, iar erorile sunt tratate eficient. De exemplu, o configurație comună de bază pentru EC2 este implementarea

unui stack LAMP (Linux, Apache, MySQL și PHP/Python/Perl), ceea ce face ca dezvoltarea de aplicații web să fie rapidă și fiabilă.

AWS utilizează o versiune modificată a hypervizorului XEN, care se bazează pe tehnologia de paravirtualizare. Acest lucru permite ca sistemele de operare să ruleze mult mai eficient pe hardware-ul fizic, reducând *overhead*-ul și sporind performanța generală. Prin această tehnologie, Amazon asigură că imaginile de sistem sunt portate și testate de către furnizorii originali, oferind un mediu optimizat și sigur pentru rularea aplicațiilor. În plus, facilitățile precum Amazon Auto Scaling și Elastic Load Balancing se integrează perfect cu EC2, asigurând o distribuție uniformă a sarcinilor și o disponibilitate ridicată a serviciilor chiar și în condiții de trafic intens.

Amazon Web Services oferă o gamă variată de servicii care acoperă stocarea datelor, bazele de date, stocarea de blocuri și livrarea de conținut, toate integrate pentru a forma un ecosistem robust și scalabil. Simple Storage Service (S3) este un serviciu de stocare care permite stocarea și căutarea datelor printr-un API simplu și eficient. S3 este integrat complet cu EC2, astfel încât imaginile AMI sunt stocate în S3, iar transferul de date între S3 și EC2 se realizează fără costuri suplimentare.

Pentru gestionarea bazelor de date NoSQL, AWS pune la dispoziție DynamoDB, un serviciu rapid și scalabil, capabil să ofere performanțe predictibile chiar și la niveluri ridicate de trafic. DynamoDB permite accesul rapid la date și este optimizat pentru aplicații care necesită un timp de răspuns constant, fără a compromite scalabilitatea. În plus, pentru aplicațiile ce necesită stocare de tip bloc, Amazon Elastic Block Store (EBS) este recomandat ca sistem de stocare primar. EBS oferă stocare persistentă de date neformatate, esențială pentru sistemele de fișiere și aplicațiile care trebuie să acceseze date la nivel de bloc.

Pentru soluții bazate pe baze de date relaționale, Amazon Relational Database Service (RDS) furnizează acces la diverse engine-uri SQL, inclusiv MySQL, Oracle, SQL Server și PostgreSQL. RDS simplifică administrarea bazelor de date prin gestionarea automată a operațiunilor de *backup*, scalare și actualizare, reducând sarcina administrativă și permițând dezvoltatorilor să se concentreze pe aplicații.

În plus, pentru livrarea rapidă a conținutului la scară globală, AWS oferă CloudFront CDN, un serviciu de distribuție de conținut care furnizează date la viteze foarte mari. CloudFront este capabil să gestioneze un trafic imens și să filtreze traficul. Această soluție asigură că utilizatorii primesc conținutul în mod rapid și eficient, indiferent de locația lor geografică, contribuind astfel la o experiență de utilizare optimă și la o reducere semnificativă a latenței în livrarea datelor.

Cloud-ul Amazon este recomandat în situații diverse, oferind flexibilitate și control complet asupra aplicațiilor dezvoltate. Acesta este ideal atunci când se dorește utilizarea de software *open-source* de la un alt furnizor, deoarece AWS facilitează integrarea și rularea acestor soluții într-un mediu scalabil. De asemenea, dacă organizația dispune de cod existent

și vrea să-l transfere ulterior pe propria infrastructură, AWS permite o migrare cu o dependență tehnologică minimă, ceea ce oferă libertatea de a porta codul și, eventual, de a-l rescrie într-un alt limbaj de programare, dacă este necesar.

În plus, AWS este o alegere excelentă pentru companiile care doresc să aibă control complet asupra mediului de execuție și să efectueze teste de stres și încărcare pe scară largă, cum ar fi lansarea simultană a a peste 1000 de instanțe. Această capacitate de scalare rapidă și eficientă face ca platforma Amazon să fie o soluție robustă pentru aplicațiile web critice, permițând evaluarea performanțelor sub sarcini extreme.

Pe lângă flexibilitatea și performanța ridicată, modelul de plată de tip *pay-as-you-go* al AWS permite organizațiilor să plătească doar pentru resursele efectiv utilizate, eliminând astfel necesitatea unor investiții semnificative în infrastructură hardware. Amazon EC2 face parte dintr-un ecosistem extins care include servicii de stocare (Amazon S3, EBS), rețea (Amazon VPC) și management (Amazon CloudWatch, CloudFormation), oferind o soluție integrată pentru dezvoltarea, implementarea și scalarea aplicațiilor moderne.

### 10.3.2. Microsoft Azure

Microsoft Azure reprezintă o platformă cloud cuprinzătoare, care combină servicii IaaS și PaaS, oferind astfel flexibilitate maximă pentru dezvoltarea și implementarea aplicațiilor. Similar cu Amazon, Azure furnizează infrastructura necesară, dar se remarcă printr-o gamă extinsă de servicii PaaS, adaptate în mod special mediului Microsoft. Multe aplicații Microsoft, inclusiv cele din suita Office sau alte soluții enterprise, au fost modificate pentru a rula nativ în cloud, facilitând astfel tranziția către un model IT modern, scalabil și eficient.

La nivelul IaaS, Microsoft Azure rulează în mediul virtualizat creat de Microsoft Hypervisor, care este derivat din Windows Server. Acest nivel de virtualizare oferă servicii de stocare, capacități de calcul virtualizat și un mediu de dezvoltare complet, care permite chiar emularea Windows Azure pe desktop și integrarea cu instrumente de dezvoltare populare precum Visual Studio sau Eclipse. Astfel, utilizatorii beneficiază de un control similar cu cel oferit de infrastructura tradițională, dar fără a fi nevoie de investiții semnificative în hardware, având la dispoziție o infrastructură flexibilă și ușor de scalat.

Un alt aspect important al Microsoft Azure este API-ul său REST, care se bazează pe certificate X.509 pentru autentificare, asigurând o securitate ridicată în comunicarea între client și server. Inițial, setul de servicii oferit de Azure a inclus elemente fundamentale, considerate servicii de nivel jos (*low level*), fără interfață utilizator, cum ar fi Live Services, SQL Services, .Net Services, SharePoint Services și CRM Services. Aceste servicii au stat la baza dezvoltării ulterioare a unor aplicații complexe și au contribuit la consolidarea poziției Azure ca furnizor major de soluții cloud, care nu numai că permite migrarea la cloud, dar și transformarea digitală completă a infrastructurilor IT enterprise.

Microsoft Azure a evoluat semnificativ de la lansarea inițială, când majoritatea aplicațiilor erau dezvoltate pentru platforma LAMP, devenind o soluție flexibilă și interoperabilă. În prezent, orice server cu acces la Internet poate comunica cu Azure, indiferent dacă aplicația este găzduită în cloud-ul Microsoft sau pe un server on-premise. Astfel, dezvoltatorii pot utiliza Microsoft Azure SDK, care suportă limbaje precum PHP, Java, .NET, Node.js, Python și altele, pentru a construi aplicații capabile să acceseze serviciile Azure din orice mediu.

În 2008, ecosistemul Azure se concentra în mare parte pe .NET, însă astăzi platforma s-a extins pentru a include o gamă largă de tehnologii și limbaje de programare. Această diversificare permite companiilor să migreze sau să dezvolte aplicații moderne fără a fi constrânse la o singură tehnologie, facilitând astfel integrarea și interoperabilitatea între diverse sisteme.

Microsoft Azure este recomandat în special organizațiilor care utilizează tehnologii Microsoft, cum ar fi .NET și SQL Server, deoarece aceste componente fac parte din stiva Microsoft și asigură o integrare naturală cu serviciile Azure. Platforma este ideală pentru dezvoltarea hibridă, unde se dorește combinarea dezvoltării desktop cu cea în cloud, permițând aplicațiilor să fie dezvoltate local și ulterior integrate în mediul cloud. Totuși, se recomandă ca interfața de utilizare și logica de extragere a datelor să fie adaptate pentru a face față eventualelor limitări ale conexiunilor de internet, asigurând o experiență optimă utilizatorilor finali. Mai mult, Azure minimizează problemele de dependență tehnologică, permițând organizațiilor să migreze sau să integreze componentele cu ușurință, deoarece nucleul său este construit pe tehnologii cunoscute – SQL Server, IIS și .NET framework – care ar putea fi oferite și de alți furnizori de cloud.

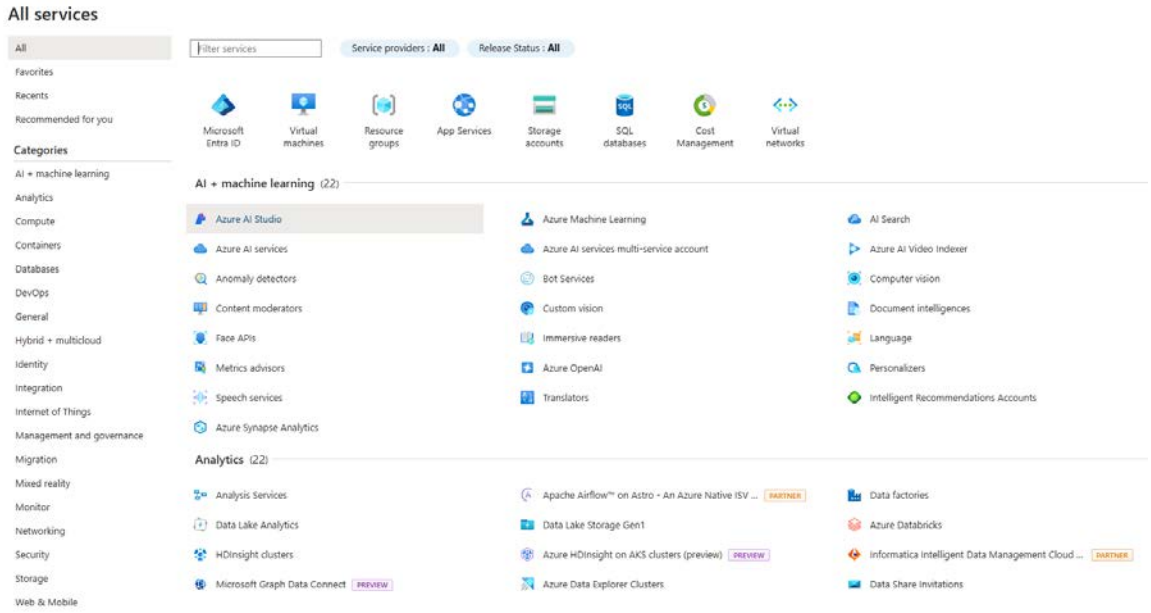



**Fig. 10.4.** Servicii Microsoft Azure.

Fig. 10.4 prezintă o listă extinsă de servicii disponibile în Microsoft Azure, acoperind domenii variate precum inteligență artificială, învățare automată, stocarea și analiza de date, containere, web și mobile, rețele virtuale și multe altele. Această diversitate de servicii oferă organizațiilor instrumentele necesare pentru a dezvolta, implementa și administra aplicații și soluții IT complexe, într-un mediu cloud unificat. De la procese de analiză a datelor la nivel enterprise până la funcționalități avansate de machine learning și integrarea cu sisteme de monitorizare și securitate, Azure își propune să ofere o platformă scalabilă și flexibilă, adaptată cerințelor specifice ale afacerilor moderne și ale dezvoltatorilor.

### 10.3.3. Google Cloud Platform (GCP)

Google Cloud Platform (GCP) este recunoscut pentru capacitatea sa de a combina elemente de tip IaaS cu cele de PaaS, oferind atât flexibilitate, cât și scalabilitate automată. La nivel IaaS, GCP permite rularea mașinilor virtuale în centre de date de ultimă generație, conectate printr-o rețea globală de fibră optică, ceea ce asigură performanțe ridicate, latență redusă și o infrastructură robustă pentru aplicațiile critice. Aceste servicii sunt gestionate prin Google Compute Engine, care oferă funcționalități precum gestionarea automată a resurselor, migrarea automată a instanțelor și suport pentru containere, permițând astfel companiilor să își scaleze rapid operațiunile în funcție de cerere.

La nivel PaaS, GCP ascunde complexitatea infrastructurii și oferă dezvoltatorilor un mediu de execuție complet gestionat, unde elementele de virtualizare și elasticitate sunt aproape invizibile. Serviciul AppEngine, de exemplu, permite rularea aplicațiilor într-un sandbox securizat, izolat de detaliile hardware, sistemul de operare și locația fizică a serverelor. Acest mediu oferă scalabilitate automată, astfel încât aplicațiile să se adapteze în timp real la fluctuațiile de trafic, fără a necesita intervenție manuală, reducând semnificativ timpul de lansare și costurile de operare.

Pe lângă Compute Engine și AppEngine, GCP oferă o gamă largă de servicii complementare care susțin dezvoltarea și implementarea aplicațiilor moderne, inclusiv stocare, baze de date, analiză de date și machine learning. De exemplu, Google Kubernetes Engine (GKE) facilitează orchestrarea containerelor și managementul microserviciilor într-un mod scalabil, iar serviciile de stocare, precum Cloud Storage, asigură o soluție robustă pentru stocarea obiectelor la scară globală. Toate aceste servicii sunt integrate într-o platformă unificată, care permite monitorizarea, securizarea și administrarea resurselor prin intermediul unor instrumente avansate, cum ar fi Stackdriver, oferind o vizibilitate completă asupra performanței și a costurilor.

Google App Engine este o soluție Platform as a Service (PaaS) oferită de Google Cloud Platform, concepută pentru a permite dezvoltarea rapidă a aplicațiilor. Această platformă rulează aplicațiile într-un mediu sandbox, care izolează aplicația de detaliile hardware și de sistemul de operare, facilitând astfel dezvoltarea și testarea rapidă.

Google App Engine este ideal pentru proiecte în care nu există un cod preexistent, cum ar fi aplicațiile de tip mashup sau cele bazate pe modelul cerere-răspuns, unde intrarea rapidă pe piață este esențială. Acest mediu PaaS permite dezvoltatorilor să se concentreze exclusiv pe scrierea și optimizarea codului, fără a fi nevoie să se îngrijoreze de aspectele operaționale, precum instalările software sau configurările de bază. De asemenea, faptul că Google App Engine minimizează riscul de dependență tehnologică la Google, în contextul în care se pot utiliza diverse tehnologii și limbaje, oferă o flexibilitate suplimentară organizațiilor care doresc să profite de avantajele cloud-ului fără a se simți constrânse la un singur furnizor.

Așadar, GCP se evidențiază printr-o combinație puternică de IaaS și PaaS, care permite utilizatorilor să beneficieze de flexibilitate, performanță și scalabilitate automată, toate acestea într-un mediu securizat și ușor de administrat.

### 10.3.4. Cloud-uri private

Cloud-urile private, adesea conceptualizate ca Datacenter as a Service (DaaS), oferă organizațiilor controlul complet asupra infrastructurii lor, însă vin cu anumite considerații specifice. În general, aceste cloud-uri sunt de dimensiuni mai mici, fiind adesea implementate pentru a susține aplicații *legacy*, care nu pot fi ușor adaptate la arhitecturi cloud native. Deși multe organizații consideră că infrastructura *on-premises* oferă o securitate sporită, realitatea este că un mediu privat nu implică neapărat un nivel mai mare de securitate față de serviciile cloud publice, mai ales atunci când nu sunt implementate măsuri avansate de protecție.

Conform unui studiu realizat de Deloitte, aproximativ 90% dintre organizații utilizează servicii cloud de cel puțin trei ani, ceea ce evidențiază adoptarea pe scară largă a tehnologiilor cloud, indiferent dacă acestea sunt implementate ca cloud-uri private sau publice. Această statistică subliniază faptul că, deși cloud-ul privat poate oferi avantaje în ceea ce privește controlul și personalizarea, multe organizații beneficiază de pe urma flexibilității și scalabilității oferite de serviciile cloud, în contextul unei strategii hibride sau al unei tranziții treptate de la infrastructura tradițională.

# Bibliografie


1. Nicole Forsgren, Jez Humble, Gene Kim – Accelerate: The Science of Lean Software and DevOps, IT Revolution Press, ISBN: 978-1942788331, 2018.
2. Titus Winters, Tom Manshreck, Hyrum Wright – Software Engineering at Google: Lessons Learned from Programming Over Time, O'Reilly Media, ISBN 978-1492082798, 2020.
3. Behrouz A. Forouzan – Data Communications and Networking, 6th Edition, McGraw-Hill, ISBN 978-0073376226, 2018.
4. Douglas E. Comer – Internetworking with TCP/IP, Volume One: Principles, Protocols, and Architecture, 6th Edition, Pearson, ISBN 978-0136085300, 2019.
5. Chris Richardson – Microservices Patterns: With Examples in Java, Manning Publications, ISBN 978-1617294549, 2019.
6. Sam Newman – Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith, O'Reilly Media, ISBN 978-1492047841, 2019.
7. Sam Newman – Building Microservices: Designing Fine-Grained Systems, O'Reilly Media, ISBN 978-1492034025, 2021.
8. Eberhard Wolff – Microservices: Flexible Software Architecture, Addison-Wesley, ISBN 978-0134650401, 2016.
9. JJ Geewax – API Design Patterns, O'Reilly Media, ISBN 978-1617295850, 2021.
10. James Gough, Daniel Bryant, Matthew Auburn – Mastering API Architecture: Design, Operate, and Evolve API-Based Systems 1st Edition, O'Reilly Media, ISBN 978-1492090632, 2022.
11. Brenda Jin, Saurabh Sahni, Amir Shevat – Designing Web APIs: Building APIs That Developers Love, O'Reilly Media, ISBN 978-1492026921, 2019.
12. Leonard Richardson, Mike Amundsen, Sam Ruby – RESTful Web APIs: Services for a changing world, O'Reilly Media, ISBN 978-1449358068, 2013.
13. Andrew Hoffman – Web Application Security: Exploitation and Countermeasures for Modern Web Applications, O'Reilly Media, ISBN 978-1492053118, 2020.
14. Malcolm McDonald – Web Security for Developers: Real Threats, Practical Defense, No Starch Press, ISBN 978-1593279943, 2020.
15. Matthew Portnoy – Virtualization Essentials, 3rd Edition, Sybex, ISBN 978-1394181568, 2023.
16. Nigel Poulton – Docker Deep Dive: Updated for Docker 20.x, ISBN 978-1521822807, 2022.
17. Kelsey Hightower, Brendan Burns, Joe Beda – Kubernetes Up & Running: Dive into the Future of Infrastructure, 3rd Edition, O'Reilly Media, ISBN 978-1098110208, 2022.

18. Gastón C. Hillar – MQTT Essentials – A Lightweight IoT Protocol, Packt Publishing, ISBN 978-1787287815, 2018.
19. Stephen Cope – MQTT For Complete Beginners: Learn The Basics of the MQTT Protocol, ISBN 979-8779030762, 2022.
20. Pini Reznik, Jamie Dobson, Michelle Gienow – Cloud Native Transformation: Practical Patterns for Innovation, O'Reilly Media, ISBN 978-1492048909, 2020.
21. John Culkin, Mike Zazon – AWS Cookbook: Recipes for Success on AWS, Packt Publishing, ISBN 978-1492092605, 2022.
22. Vitthal Srinivasan, Janani Ravi, Judy Raj – Google Cloud Platform for Architects: Design and manage powerful Cloud solutions, Packt Publishing, ISBN 978-1788834308, 2018.
23. Scott Guthrie, Mark Russinovich – Microsoft Azure for Architects, Wiley, ISBN: 978-1119661239, 2021.
24. Jonah Carrio Andersson – Learning Microsoft Azure: Cloud Computing and Development Fundamentals, 1st Edition, O'Reilly Media, ISBN 978-1098113322, 2022.
25. Wordwide Market Share of Leading Cloud Providers, [Online] la https://www.statista.com/chart/18819/worldwide-market-share-of-leading-cloud-infrastructure-service-providers/, accesat 04.04.2025.