

UNIVERSITATEA BABE -BOLYAI
FACULTATEA DE MATEMATICĂ ȘI INFORMATICĂ
SPECIALIZAREA INFORMATICĂ

Lucrare de Licență

Arhitecturi Băzate pe Servicii REST

Coordonator științific:
prof. dr. **Grigoreta Cojocar**

Absolvent
Horea Marius Adam

Cluj-Napoca
2019

Suprins

ns	2
Introducere	4
1. Arhitectura Software	6
1.1 Abstracție în timpul execuției	6
1.2 Elemente	7
1.2.1 Componente	7
1.2.2 Conectori	8
1.2.3 Date	8
1.3 Configurații	9
1.4 Rezumat	9
2. Arhitecturi de aplicații bazate pe rețea	10
2.1 Bazate pe rețea vs. distribuite	10
2.2 Proprietăți cheie	11
2.2.1 Performanță	11
2.2.1.1 Performanța rețelei	12
2.2.1.2 Performanța percepută de utilizator	13
2.2.1.3 Network Efficiency	14
2.2.2 Scalabilitate	15
2.2.3 Simplitate	15
2.2.4 Modificabilitate	17
2.3 Rezumat	17
3. Representational state transfer (REST)	18
3.1 Constrângerile REST	18
3.1.1 Client-Server	18
3.1.2 Stateless	19
3.1.3 Cache	20
3.1.4 Interfața Uniformă	20
3.1.5 Sistem stratificat	21
3.2 Modelul de Maturitate Richardson	22
3.2.1 Nivelul 0	22
3.2.2 Nivelul 1 - Resurse	25
3.2.3 Nivelul 2 - Verbe HTTP	27
3.2.4 Nivelul 3 - Controale Hypermedia	29
3.2.5 Rezumat	30
3.3 Date	31
3.3.1 Resurse și Identificatori de Resurse	32

3.3.2 Reprezentări	33
4. Aplicația Practică	34
4.1 Analiza Cerințelor	34
4.2 Design	34
4.3 Testare	40
4.4 Tratarea Încărcării de Imagini într-o Manieră RESTful	41
4.5 Direcții Viitoare	42
Concluzii	44
Bibliografie	45

Introducere

Complexitatea sistemelor software moderne necesită un accent mai mare pe sistemele de componente, unde implementarea este partiționată în componente independente care comunică pentru a efectua un anumit task. Cercetarea arhitecturii software investighează metodele pentru a stabili care este modul optimal de partiționare al unui sistem, modul în care componentele se identifică și comunică între ele, modul în care informațiile sunt transferate, cum elementele unui sistem pot evolua independent, și cum pot fi descrise toate cele de mai sus folosind notații formale și informale. În acest scop s-a folosit lucrarea lui Perry și Wolf, Foundations for the study of software architecture [3].

Ca idee, luați în considerare cât de des vedem că proiectele software încep cu adoptarea celui mai recent trend în designul arhitectural și abia mai târziu, se descoperă că cerințele sistemului nu necesită o astfel de arhitectură. Design-by-buzzword este o întâmplare frecventă. Cel puțin o parte din acest comportament din cadrul industriei software se datorează lipsei de înțelegere a motivului pentru care un set dat de constrângeri arhitecturale este util. Cu alte cuvinte, raționamentul din spatele unor arhitecturi software bune nu este evident pentru proiectanți atunci când arhitecturile sunt selectate pentru reutilizare.

Lucrarea este împărțită în trei capitole teoretice și un capitol dedicat prezentării aplicației practice. Primele două capitole de teorie definesc un framework pentru analizarea diferitelor stiluri arhitecturale și scot la iveală unele diferențe subtile dintre arhitecturile bazate pe rețea și arhitecturile distribuite. Sunt prezentate de asemenea și mai multe proprietăți cheie care sunt de așteptat de la un sistem distribuit performant.

Capitolul trei introduce și elaborează stilul arhitectural Representational State Transfer (REST) pentru sisteme hypermedia distribuite. REST oferă un set de constrângeri arhitectural care, atunci când sunt aplicate în ansamblu, accentuează scalabilitatea componentelor, generalitatea interfețelor, implementarea independentă a componentelor, și componente intermediare pentru a reduce latența interacțiunii. Ca și parte a acestui capitol, am inclus și o prezentare a modelului de maturitate Richardson care descompune elementele principale ale unei abordări REST pe trei nivele: resurse, verbe HTTP și controale hypermedia.

În final, ultimul capitol este destinat aplicației practice. Aceasta este împărțită în trei componente: API, Admin și Client. În acest capitol sunt prezentate cerințele aplicației, modul de funcționare, direcții de extindere viitoare, dar și unele detalii de implementare care au necesitat soluții mai puțin convenționale pentru a rămâne RESTful și totodată pentru a îndeplini cerințele de funcționare.

Contribuțiile proprii aduse acestei lucrări sunt reprezentate de sinteza diferitelor surse de informații pentru a defini framework-ul de analiză a arhitecturilor software, categorie în care intră bineînțeles și stilul arhitectural REST. Totodată, dezvoltarea aplicației practice care are la bază o componentă server și două aplicații client, a avut un rol major în a sedimenta și înțelege mai bine conceptele prezentate în lucrare.

1. Arhitectura Software

Acest capitol definește o terminologie auto-consistentă pentru arhitectura software, fiecare definiție este urmată de o discuție despre modul în care este derivată sau se compară cu alte cercetări curente. Unii termeni sunt folosiți în forma lor originală din limba engleză pentru a nu pierde semnificația termenului prin traducere.

1.1 Abstracție în timpul execuției

O arhitectură software este o abstractizare a elementelor de rulare ale unui sistem software în timpul unei faze a funcționării sale. Un sistem poate fi compus din mai multe niveluri de abstractizare și multe faze de operare, fiecare cu arhitectura software proprie.

În centrul arhitecturii software se află principiul abstractizării: ascunderea câtorva detalii ale unui sistem prin încapsulare pentru a identifica și susține mai bine proprietățile sale [1]. Un sistem complex va conține numeroase niveluri de abstractizare, fiecare având o arhitectură proprie. O arhitectură reprezintă o abstractizare a comportamentului la acel nivel, astfel încât elementele arhitecturale sunt delimitate de interfețele abstracte pe care le furnizează altor elemente la acel nivel. În fiecare element poate fi găsită o altă arhitectură, care definește sistemul de sub-elemente care implementează comportamentul reprezentat de interfața abstractă a elementului. Această recurență de arhitecturi continuă până la cele mai simple elemente: cele care nu pot fi descompuse în elemente mai puțin abstracte.

Perry și Woff [3] definesc elementele de procesare drept *“transformatoare de date”*, în timp ce Shaw și colab. [4] descriu componentele drept *“locul calculului și al stării”*. Acest lucru este clarificat suplimentar în Shaw și Clemens [4]: *“Un element component este o unitate de software care îndeplinește unele funcții în timpul rulării. Exemplele includ obiecte, procese și filtre”*. Acest lucru ridică o distincție importantă între arhitectura software și structura software: prima este o abstractizare a comportamentului în timpul de execuție al unui sistem software, în timp ce a doua este o proprietate statică a codului software. Prin urmare, designul arhitectural și designul structural al codului sursă, deși sunt strâns legate, sunt activități de proiectare separate

1.2 Elemente

O arhitectură software este definită de o configurație de elemente arhitecturale - componente, conectori și date - constrânse în relațiilor lor pentru a realiza un set dorit de proprietăți arhitecturale.

O examinare completă a domeniului de aplicare și o bază intelectuală pentru arhitectura software poate fi găsită în Foundations for the Study of Software Architecture [3]. În această lucrare, Perry și Wolf prezintă un model care definește o arhitectură software ca un set de elemente arhitectural care au o formă particulară, având în spate o rațiune bine justificată. Deși justificarea este un aspect important al cercetării arhitecturii software și în special a descrierii arhitecturale, includerea acestora în cadrul definiției arhitecturii software ar presupune că documentația de proiectare face parte din sistemele de rulare.

Ca ilustrație, luați în considerare ce se întâmplă cu o clădire dacă modelele și designul acesteia sunt arse? Clădirea se prăbușește imediat? Nu, deoarece proprietățile prin care pereții susțin greutatea acoperișului rămân intacte. O arhitectură are, prin proiectare, un set de proprietăți care îi permit să satisfacă sau să depășească cerințele sistemului. Necunoașterea acestor proprietăți poate duce la modificări ulterioare care încalcă arhitectura, la fel cum înlocuirea unui perete de susținere cu o fereastră poate șubrezi stabilitatea structurală a unei clădiri. Documentația arhitecturii explică acele proprietăți, iar lipsa de documentație poate duce la decăderea sau degradarea arhitecturii în timp, dar documentația în sine nu face parte din arhitectură.

O caracteristică cheie a modelului introdus de Perry și Wolf [3] este distincția dintre diferitele tipuri de elemente. *Elementele de procesare* sunt cele care efectuează transformări ale datelor, *datele* sunt acele elemente care conțin informațiile care sunt utilizate și transformate, iar *elementele de conectare* sunt lipiciul care ține diferitele piese ale arhitecturii laolaltă. Pe parcursul acestei lucrări, termenii *componente* și *conectori* vor fi prevalenți pentru a denota *elementele de procesare* și, respectiv, *conectare*.

1.2.1 Componente

O **component** este o unitate abstractă de instrucțiuni software și stare internă care oferă o transformare a datelor prin interfața sa.

Componentele sunt aspectul cel mai ușor de recunoscut al arhitecturii software. Elementele de procesare ale lui Perry și Wolf [3] sunt definite ca acele componente care furnizează transformarea elementelor de date și sunt definite mai degrabă prin interfața și serviciile pe care le oferă altor componente, decât prin implementarea lor internă din spatele interfeței.

1.2.2 Conectori

Un **conector** este un mecanism abstract care mediază comunicarea, coordonarea sau cooperarea între componente.

Perry și Wolf [3] descriu elementele de conectare vag ca adezivul care ține diferitele piese ale arhitecturii împreună. O definiție mai precisă este oferită de Shaw și Clements [5]: Un conector este un mecanism abstract care mediază comunicarea, coordonarea sau cooperarea între componente. Exemplele includ reprezentări partajare, protocoale de transmitere a mesajelor și fluxuri de date.

Poate că cel mai bun mod de a ne gândi la conectori este să-i punem în contrast cu elementele de procesare, cu alte cuvinte, componentele. Conectorii permit comunicarea între componente prin transferul datelor de la o interfață la alta fără a schimba datele. Intern, un conector poate fi constituit dintr-un subsistem de componente care transformă datele pentru transfer, efectuează transferul, apoi inversează transformarea pentru livrare. Cu toate acestea, abstractizarea comportamentului extern capturat de arhitectură ignoră aceste detalii. În contrast, o componentă poate transforma, dar nu întotdeauna, datele din perspectiva externă.

1.2.3 Date

Datele sunt un element de informație care este transferat de la o componentă sau primit de o componentă printr-un conector.

Boasson [6] critică cercetările de arhitectură software actuale pentru accentul pus pe structura componentelor și instrumentele de dezvoltare a arhitecturii, sugerând că ar trebui să se pună mai mult accent pe modelarea arhitecturală centrată pe date. Comentarii similare au fost făcute și de Robert Martin [7], care afirmă că o arhitectură curată exprimă, înainte de toate, intenția, și că primul lucru pe care îl vezi atunci când privești codul sursă al unui proiect, nu ar trebui să fie concepte specifice framework-ului în care lucrezi, ci o serie de

nume care să-ți sugereze cu ce se ocupă aplicația respectivă. El continuă această idee, afirmând că, dacă dezvolți o aplicație web, acest fapt nu ar trebui expus direct, fiind doar un detaliu de implementare, un simplu dispozitiv IO, un conector folosit pentru a transfera date în și din aplicație.

Datele sunt elemente de informații care sunt transferate de la o componentă sau primite de o componentă printr-un conector. Exemplele includ secvențe de octeți, mesaje, parametri și obiecte serializate, dar nu includ informații care aparțin implementării din cadrul unei componente. Din perspectiva arhitecturală, un fișier este o transformare pe care o componentă a unui sistem de fișiere ar putea să o facă din argumentul “nume fișier”, primit prin interfața sa într-o secvență de octeți înregistrați într-un sistem de stocare intern. Componentele, de asemenea, pot genera date, ca în cazul încapsulării software a unui ceas sau senzor.

Natura elementelor de date dintr-o arhitectură de aplicații bazată pe rețea va determina adesea dacă este adecvat sau nu un stil arhitectural dat. Acest lucru este deosebit de evident în componența paradigmelor de proiectare a codului mobil [8], unde alegerea trebuie făcută între interacțiunea cu o componentă direct sau transformarea componenteii într-un element de date, transferarea acesteia printr-o rețea și transformarea acesteia înapoi în componentă cu care se poate interacționa la nivel local. Este imposibil să evaluăm o astfel de arhitectură fără a lua în considerare elementele de date la nivel de arhitectură.

1.3 Configurații

O **configurație** este structura relațiilor arhitecturale între componente, conectori și date într-o perioadă de timp de rulare a sistemului. Strict vorbind, s-ar putea considera că o configurație este echivalentă cu un set de constrângeri specifice privind integrarea componentelor.

1.4 Reumat

Acest capitol a examinat o parte din contextul acestei lucrări. Introducerea și formalizarea unui set bine definit de terminologie pentru conceptele de arhitectură este necesar pentru a evita confuzia dintre arhitectură și descrierea arhitecturii. Următorul capitol continua discuția despre tema ce stă la baza acestei lucrări, concentrându-se pe arhitecturi de aplicații bazate pe

rețea și descriind modul în care diferitele stiluri existente pot fi folosite pentru a ghida designul lor arhitectural.

2. Arhitecturi de aplicații bazate pe rețea

Acest capitol continuă discuția noastră despre tema principală a lucrării, concentrându-se pe arhitecturi de aplicații bazate pe rețea și descriind modul în care stilurile pot fi utilizate pentru a ghida designul lor arhitectural.

Arhitectura se găsește la mai multe niveluri în sistemele software. Această lucrare examinează cel mai înalt nivel de abstractizare în arhitectura software, unde interacțiunile dintre componente pot fi realizate folosind comunicarea prin rețea.

2.1 Bazăte pe rețea vs. distribuite

Distincția principală între arhitecturile bazate pe rețea și arhitecturile software în general este aceea că transmiterea de date între componente este limitată la comunicarea orientată pe mesaje [9] sau echivalentul transmiterii mesajelor dacă un mecanism mai eficient poate fi selectat în timpul rulării bazat pe locația componentelor [10].

Tanenbaum și van Renesse [11] fac distincție între sistemele distribuite și sistemele bazate pe rețea: un sistem distribuit este unul care arată utilizatorilor săi ca un sistem centralizat obișnuit, dar care rulează pe mai mult procesoare independente. Un bun exemplu de sistem distribuit este bine-cunoscutul motor de căutare Elasticsearch, unde fiecare nod dintr-un cluster colaborează la construirea rezultatului final. În schimb, sistemele bazate pe rețea sunt cele capabile să funcționeze prin intermediul unei rețele, dar nu neapărat într-o manieră transparentă pentru utilizator. În unele cazuri este de dorit ca utilizatorul să fie la curent cu diferența dintre o acțiune care necesită o interacțiune ce implică comunicarea prin rețea și una care poate fi satisfăcută local, în special atunci când utilizarea rețelei implică un cost suplimentar pentru tranzacție.

Un alt exemplu care ar putea dezvălui mai bine diferența dintre sistemele bazate pe rețea și cele distribuite este comparația dintre HTTP și RPC.

Chiar dacă RPC se întâmplă și acum prin fir, HTTP nu înseamnă RPC. După cum Roy Fielding [21] scrie în lucrarea sa de disertație: Oamenii se referă adesea greșit la HTTP ca la un mecanism de apel de procedură la distanță (RPC), pur și simplu pentru că urmează

aderă stilului cerere - răspuns. Ceea ce distinge RPC de alte forme de comunicare a aplicațiilor bazate pe rețea este conceptul de invocare de la distanță (RMI). Invocarea metodei la distanță

este similară cu RPC, cu excepția faptului ca procedura este identificată ca un tuplu {obiect, metodă}.

Ceea ce face HTTP să se diferențieze semnificativ de RPC este că solicitările sunt direcționate către resurse folosind o semantică standard care poate fi interpretată de către intermediari aproape ca și de serverele de origine. Rezultatul este o aplicație care permite straturi de transformare și indirecție care sunt independente de originea informației, ceea ce este crucial pentru un sistem de informații scalabil la nivelul Internetului, multi-organizațional. Sistemele RPC, în schimb sunt definite în termeni de API-uri la nivel de limbaj de programare.

Arhitectura aplicației software este un nivel de abstractizare a unui sistem ca întreg, în care obiectivele unei acțiuni ale utilizatorului sunt reprezentabile ca proprietăți arhitecturale funcționale. De exemplu, o aplicație hypermedia trebuie să fie preocupată de locația paginilor cu informații, efectuarea request-urilor și interpretarea fluxurilor de date. Acest lucru este în contrast cu o abstractizare la nivel de rețea, unde obiectivul este de a muta biți dintr-o locație în alta fără a se ține cont de ce trebuie să se întâmple asta. Numai la nivelul aplicație putem evalua compromisuri de proiectare în funcție de numărul de interacțiuni per acțiune efectuată de utilizator, locația stării aplicației, și transferul eficient al datelor.

2.2 Proprietăți cheie

Această secțiune descrie unele dintre proprietățile arhitecturale utilizate pentru diferențierea și clasificarea stilurilor arhitecturale.

2.2.1 Performanță

Unul dintre motivele principale pentru a ne concentra atenția asupra stilurilor pentru aplicațiile bazate pe rețea este acela că interacțiunile componente pot fi factorul dominant în determinarea performanței percepute de utilizator și a eficienței rețelei. Deoarece stilul arhitectural dictează natura acelor interacțiuni, selectarea unui stil arhitectonic adecvat poate face diferența între succes și eșec în implementarea unei aplicații bazate pe rețea, unde fiecare

interacțiune contează.

Performanța unei aplicații bazate pe rețea este legată mai întâi de cerințele aplicației, apoi de stilul de interacțiune ales, urmat de arhitectura realizată și, în final, de implementarea fiecărei componente în parte. Cu alte cuvinte, software-ul nu poate evita costul de bază pentru realizarea cerințelor aplicației; de exemplu, dacă aplicația necesită ca datele să fie localizate pe sistemul B, atunci software-ul nu poate evita mutarea datelor de la A la B. De asemenea, o arhitectură nu poate fi mai eficientă decât permite stilul său de interacțiune; de exemplu, costul interacțiunilor multiple necesare pentru a muta datele de la A la B nu poate fi mai mic decât cel al unei singure interacțiuni de la A la B. În sfârșit, indiferent de calitatea unei arhitecturi, nicio interacțiune nu poate avea loc mai repede decât o implementare a componentelor poate produce date și receptorul său poate consuma datele.

2.2.1.1 Performanța rețelei

Metricile de măsurare a performanței unei rețele sunt utilizate pentru a descrie unele attribute ale comunicării între componente. *Randamentul / Throughput* este viteza cu care informațiile, atât datele aplicației și comunicațiile operaționale (overhead), sunt transferate între componente. *Costul operațional / Overhead* - ul poate fi separat în costul necesar configurării inițiale și costul per-interacțiune, o distincție utilă pentru identificarea conectorilor care pot partaja costul configurării inițiale pe durata mai multor interacțiuni (amortizare). *Lățimea de bandă / Bandwidth* este o metrică a debitului maxim disponibil pe o anumită legătură, calculată ca și raportul dintre cantitatea de date transmise și timpul de transmisie. Lățimea de bandă utilizabilă se referă la acea porțiune de bandă care este de fapt disponibilă pentru a deservi aplicația.

Stilurile impactează performanța rețelei prin influența lor asupra numărului de interacțiuni per acțiunea utilizatorului și a granularității elementelor de date. Un stil care încurajează interacțiunile mici, puternic tipizate va fi eficient într-o aplicație care implică transferuri mici de date între componente cunoscute, dar va provoca overhead în cadrul aplicației care implică transferuri mari de date sau interfețe negociate. Totodată, un stil care implică coordonarea mai multor componente aranjate pentru a filtra un flux de date mare va fi nelalocul lui într-o aplicație care necesită mesaje simple de control.

2.2.1.2 Performanța percepută de utilizator

Performanța percepută de utilizator diferă de performanța rețelei prin aceea că performanța unei acțiuni este măsurabilă mai degrabă din punct de vedere al impactului său asupra utilizatorului în fața unei aplicații, decât viteza cu care rețeaua este capabilă să mute informațiile. Metricile principale folosite pentru a măsura performanța percepută de utilizator sunt latența și timpul de finalizare.

Latența / Latency este perioada de timp dintre stimulul inițial și prima indicație a unui răspuns. Latența are loc în mai multe puncte din flow-ul procesării unei acțiuni în cadrul unei aplicații bazate pe rețea:

1. timpul necesar pentru ca aplicația să recunoască evenimentul care a inițiat acțiunea;
2. timpul necesar pentru configurarea interacțiunilor dintre componente;
3. timpul necesar pentru a transmite fiecare interacțiune componentelor;
4. timpul necesar procesării fiecărei interacțiuni pe acele componente;
5. timpul necesar pentru a finaliza transferul și procesarea suficientă a rezultatului interacțiunilor înainte ca aplicația să poată începe interpretarea unui rezultat utilizabil.

Este important de menționat că, deși numai 3. și 5. implică utilizarea rețelei pentru comunicare, toate cele 5 puncte pot fi afectate de stilul arhitectural. Mai mult, interacțiunile multiple ale componentelor per acțiune utilizator sunt aditive la latență, cu excepția cazului în care au loc în paralel.

Completarea / Completion reprezintă perioada de timp necesară pentru finalizarea unei acțiuni. Timpul de finalizare depinde de toate măsurile menționate anterior. Diferența dintre timpul de finalizare al unei acțiuni și latența ei reprezintă gradele cu care aplicația procesează treptat datele primite. De exemplu, un browser Web care poate afișa o imagine de dimensiuni mari în timp ce este recepționată oferă o performanță percepută de utilizator drastic îmbunătățită, decât una în care așteaptă să utilizeze întreaga imagine, chiar dacă ambele experimentează aceeași performanța de rețea.

Este important de menționat că aspectele de proiectare adoptate pentru optimizarea latenței vor avea adesea efectul secundar al degradării timpului de finalizare și viceversa. De exemplu, compresia unui flux de date poate produce o codificare mai eficientă dacă algoritmul probează o porțiune semnificativă a datelor înainte de a produce transformarea codificată, rezultând într-un timp mai scurt de finalizare pentru a transfera datele codificate în

rețea. Cu toate acestea, dacă această compresie este efectuată on-the-fly, ca răspuns la o acțiune a utilizatorului, tamponarea unui eșantion mare înainte de transfer poate produce o latență inacceptabilă. Echilibrarea acestor compromisuri poate fi dificilă, în special atunci când nu se știe dacă destinatarului îi pasă mai mult de latență (de exemplu, browsere web) sau de finalizare (de exemplu, web spiders).

2.2.1.3 Network Efficiency

O observație interesantă despre aplicațiile fondate pe rețea este că cele mai bune performanțe ale aplicației se obțin prin **nefolosirea rețelei**. În esență, acest lucru înseamnă că cele mai eficiente stiluri arhitecturale pentru o aplicație bazată pe rețea sunt cele care pot minimiza eficient utilizarea rețelei atunci când este fezabil să procedeze astfel, prin:

- reutilizarea interacțiunilor anterioare (caching);
- reducerea frecvenței interacțiunilor de rețea în raport cu acțiunile utilizatorului (date replicate și operare în modul offline);
- sau prin eliminarea necesității unor interacțiuni, prin mutarea procesării datelor mai aproape de sursa de date (cod mobile)

Impactul diferitelor probleme de performanță este adesea legat de domeniul aplicabilitate al sistemului. Avantajele unui stil în condiții familiare, locale pot deveni rapid dezavantaje atunci când acest stil confruntă condiții globale. Astfel, proprietățile unui stil trebuie să fie încadrate în raport cu distanța de interacțiune:

- în cadrul unui singur process;
- în cadrul mai multor procese pe o singură gazdă;
- în interiorul unei rețele locale (LAN);
- sau răspândite pe o rețea cu acoperire mai largă (WAN).

Îngrijorări suplimentare devin evidente în momentul în care interacțiunile dintr-o rețea WAN, unde este implicată o singură organizație, sunt comparate cu interacțiunile la nivel de Internet, implicând mai multe frontiere organizaționale.

2.2.2 Scalabilitate

Scalabilitatea se referă la abilitatea arhitecturii de a susține un număr mare de componente, sau interacțiuni între componente, în cadrul unei configurații active. Scalabilitatea poate fi îmbunătățită prin:

- simplificarea componentelor;
- distribuirea serviciilor pe mai multe componente (descentralizarea interacțiunii);
- și prin controlul interacțiunilor și configurațiilor ca urmare a monitorizării.

De asemenea, scalabilitatea este afectată de frecvența interacțiunilor, indiferent dacă sarcina (load) pe o anumită componentă este distribuită uniform în timp sau variază puternic, indiferent dacă o interacțiune necesită livrare garantată sau doar un best-effort, indiferent dacă un request implică tratament sincron sau asincron, și indiferent dacă mediul este controlat sau anarhic (adică, poți avea încredere în celelalte componente implicate?).

2.2.3 Simplitate

Mijlocul principal prin care simplitatea poate fi atinsă este prin aplicarea principiului separării preocupărilor la alocarea funcționalității în cadrul componentelor. Dacă funcționalitatea poate fi alocată astfel încât componentele individuale sunt substanțial mai puțin complexe, atunci acestea vor fi mai ușor de înțeles și de implementat. De asemenea, o astfel de separare ușurează sarcina de a raționa despre arhitectura de ansamblu. Am ales să grupez calitățile de înțelegere și verificabilitate sub proprietatea generală a simplității, deoarece merg mână în mână în cadrul unui sistem bazat pe rețea.

Separarea preocupărilor este oarecum un amestec între principiul responsabilității singulare (Single Responsibility Principle - SRP) și principiul separării interfețelor (Interface Segregation Principle - ISP). Aceste principii au fost introduse de Robert Martin [13] ca parte a unui set 5 astfel de principii, cunoscut sub acronimul S.O.L.I.D. SRP afirmă că fiecare clasă sau modul dintr-un program ar trebui să facă un singur lucru, pe cât de bine posibil, în timp ce ISP sugerează că nicio clasă client nu ar trebui să depindă de metode pe care nu le folosește (evitarea claselor / componentelor care sunt capabile de a îndeplini o gamă foarte

variata de task-uri, fără a avea neapărat o legătură între ele, ci pur și simplu pentru că acea clasa / componentă era cea mai convenabilă).

Aplicarea principiului generalității elementelor arhitecturale îmbunătățește simplitatea, deoarece scade variația în interiorul respectivei arhitecturi. O bună arhitectură reușește să ascundă detaliile potrivite, la nivelul de abstractizare potrivit. Generalitatea conectorilor duce inevitabil la implementarea șablonului middleware. Bernstein [14] definește middleware-ul ca un serviciu distribuit de sisteme care include interfețe și protocoale de programare standard. Aceste servicii acționează ca un strat deasupra sistemului de operare și a rețelei și sub aplicații specifice industriei.

Termenul middleware este utilizat și în alte contexte. Un middleware este uneori utilizat într-un sens similar cu un driver de software, un strat de abstractizare care ascunde detalii despre dispozitivele hardware sau alt software folosit indirect într-o aplicație:

- sistemul de operare Android folosește nucleul linux la baza sa și oferă, de asemenea, un framework pe care dezvoltatorii îl încorporează în aplicațiile lor. În plus, Android oferă un strat de middleware care include librării care oferă servicii precum stocarea de date, afișarea ecranului, multi-media și navigarea web. Deoarece librăriile sunt compilate în limbaj mașina, serviciile se execută rapid, fiind astfel foarte convenabil de utilizat. Stratul middleware implementează, de asemenea, funcții specifice dispozitivului, astfel încât aplicațiile nu trebuie să se preocupe de variațiile dintre diferite dispozitive Android.;
- software-ul pentru motoare de joc, cum ar fi Gamebryo și RenderWare, sunt uneori descrise drept middleware, datorită faptului că oferă multe servicii facilitatoare dezvoltării de jocuri [16].

Scopul principal al unui middleware este de a furniza interfețe independente de platformă, astfel încât aplicația să poată fi portată pe mai multe platforme. Și includ servicii care maschează o mare parte din complexitatea rețelelor și sistemelor distribuite, extrăgând funcțiile utilizate în mod regulat în componente independente, în modul acesta să poată fi partajate între platforme și medii software variate.

2.2.4 Modificabilitate

Modificabilitatea se referă la ușurința cu care se poate face o modificare a arhitecturii unei aplicații. Modificabilitatea poate fi clasificată mai departe în extensibilitate, configurabilitate, și adaptabilitate. O preocupare deosebită a sistemelor care au la bază rețeaua este modificarea dinamică. Modificare dinamică se face la o aplicație lansată, în rulare, fără a necesita oprirea sau repornirea sistemului.

Chiar dacă ar fi posibil să se construiască un sistem software care să corespundă perfect cerințelor utilizatorilor săi, aceste cerințe se vor schimba în timp, la fel cum societatea evoluează în timp. Întrucât componentele participante la desfășurarea unei aplicații întemeiate pe rețea pot fi distribuite de-a lungul mai multor granițe organizaționale, sistemul trebuie să fie pregătit pentru inovație treptată și fragmentată, în care vechile și noile implementări coexistă.

De pildă, am fost implicat într-un proiect care a fost lansat inițial ca o singură aplicație monolit, fiind destinat comerțului electronic. Pe măsură ce industria a evoluat, acea aplicație originală, chiar dacă satisfăcea pe deplin cerințele utilizatorului, inevitabil a devenit inactuală. Pentru a ține pasul cu o piață în continuă dezvoltare și a ramane competitiv, s-a luat decizia de înlocuire graduală a aplicației monolit cu un set set servicii specializate pe un anumit domeniu. Aceasta decizie arhitecturală a permis apoi lansări independente și uzitarea de tehnologii specifice fiecărui serviciu în parte. Totodată, această decizie adaugă complexitate sistemului, necesitând o comunicare revizuită între echipe, până la urmă, după cu legea lui Conway afirma *“organizațiile care proiectează sistemele ... sunt constrânse să producă modele care sunt copii ale structurilor de comunicare ale acestor organizații”*.

2.3 Reumat

Acest capitol s-a concentrat pe arhitecturile clădite pe rețea și a descris modul în care stilurile arhitectonice pot fi utilizate pentru a ghida designul lor arhitectural. Deopotrivă, a definit un set de proprietăți arhitecturale folositoare în comparația diferitelor abordări arhitecturale. Capitolul următor va introduce stilul arhitectural REST și va realiza o analiza a constrângerilor introduse de acest stil.

3. Representational state transfer (REST)

Acest capitol introduce și elaborează stilul arhitectural Representational State Transfer (REST) pentru sistemele hypermedia distribuite, descriind principiile de inginerie software care ghidează REST și constrângerile de interacțiune alese pentru păstrarea acestor principii. Framework-ul de arhitectură software din primul capitol este utilizat pentru a defini elementele arhitecturale REST.

REST este un stil arhitectonic care definește un set de constrângeri care trebuie aplicate în procesul de creare al serviciilor Web. Serviciile web care se conformează stilului arhitectural REST, numite RESTful Web Services (RWS), asigură interoperabilitatea între sistemele informatice de pe Internet. Serviciile web RESTful permit sistemelor solicitante să acceseze și să manipuleze reprezentările textuale ale resurselor Web prin utilizarea unui set uniform și predefinit de operații nepurtătoare de stare. Alte tipuri de servicii Web, cum ar fi serviciile Web SOAP, își expun propriile seturi de operații arbitrare.

Termenul a fost introdus și definit în 2000 de Roy Fielding în disertația sa de doctorat [21]. Termenul are rolul de a evoca o imagine a modului în care se comportă o aplicație Web bine concepută: este o rețea de resurse Web (o mașină de stare virtuală) în care utilizatorul avansează prin aplicație selectând identificatori de resurse, ceea ce duce la obținerea reprezentării următoarei resurse pentru a fi utilizată.

3.1 Constrângerile REST

Această secțiune oferă o imagine de ansamblu generală a constrângerilor ce trebuie luate în considerare pentru ca un serviciu să fie considerat RESTful.

3.1.1 Client-Server

Prima constrângere ce trebuie aplicată de un sistem pentru a intra în categoria sistemelor REST este aceea că trebuie să respecte stilul arhitectural client-server. Separarea preocupărilor este principiul din spatele constrângerilor client-server. Prin separarea interfeței de utilizator de problemele de stocare a datelor, se îmbunătățește portabilitatea interfeței de

utilizator pe mai multe platforme și scalabilitatea prin simplificarea componentelor serverului. Cu toate acestea, poate că cel mai semnificativ pentru Web este faptul că separarea permite componentelor să evolueze independent, susținând astfel necesitatea unui sistem scalabil la nivelul Internetului, cu mai multe domenii organizaționale.

Stilul client-server este cel mai frecvent întâlnit dintre stilurile arhitecturale bazate pe rețea. O componentă server care oferă un set de servicii, ascultă pentru solicitări la aceste servicii. O componentă client, dorind ca un serviciu să fie efectuat, trimite o solicitare către server prin intermediul unui conector. Serverul fie respinge, fie execută solicitarea și trimite un răspuns înapoi clientului. Forma de bază a modelului client-server nu restricționează modul în care starea aplicației este partiționată între componentele clientului și ale serverului. Stilul este în principal caracterizat de mecanismele utilizate pentru implementarea conectorului, precum apelul de procedură la distanță (RPC) sau middleware-uri orientate pe mesaje.

3.1.2 Stateless

În continuare, adăugăm o constrângere la interacțiunea client-server: toate comunicările vor fi lipsite de stare, astfel încât să conțină toate informațiile necesare pentru a înțelege solicitarea, și nu se poate profita de niciun context stocat pe server. Prin urmare, starea sesiunii este păstrată în întregime în componenta client.

Această constrângere induce proprietățile vizibilității, fiabilității și scalabilității. Vizibilitatea este îmbunătățită deoarece un sistem de monitorizare nu trebuie să privească dincolo de un singur request pentru a determina natura completă a cererii. Fiabilitatea este îmbunătățită deoarece ușurează sarcina de recuperare din eșecuri parțiale. Prin faptul că nu trebuie să stocheze starea între solicitări, componenta serverului este capabilă să elibereze rapid resursele, având o implementare simplificată, deoarece utilizarea resurselor computaționale sau de stocare nu trebuie gestionată de server.

Constrângerea excluderii stării reflectă, la fel ca majoritatea stilurilor arhitecturale, un compromis al proiectării. Dezavantajul este că performanța rețelei ar putea fi scăzută din cauza datelor repetitive (cost per-interacțiune / overhead) trimise într-o serie de solicitări, deoarece aceste date nu pot fi lăsate pe server într-un context partajat. Mai mult, deoarece aplicația devine dependentă de implementarea corectă a semanticii pe mai multe versiuni ale

clientului, plasarea stării aplicației pe partea clientului înseamnă reducerea semnificativă a controlului pe care serverul îl are asupra comportamentului consistent al aplicației.

3.1.3 Cache

Restricțiile de cache sunt adăugate pentru a îmbunătăți eficiența rețelei. Datele dintr-un răspuns la un request trebuie să fie etichetate implicit sau explicit că fiind cacheable sau non-cacheable pentru a fi pe deplin conforme cu această constrângere. Pur și simplu, dacă un intermediar sau componenta client are răspunsul în cache, atunci este permisă re folosirea acelui răspuns pentru cereri echivalente ulterioare.

Avantajul aplicării constrângerilor cache este că acestea au potențialul de a elimina complet sau cel puțin parțial unele interacțiuni, îmbunătățind automat scalabilitatea, eficiența și, cel mai important, performanța percepută de utilizator prin reducerea latenței medii între o serie de interacțiuni. Cu toate acestea, fiabilitatea este afectată dacă datele din cache sunt învechite, diferențiindu-se semnificativ de datele care ar fi fost obținute dacă cererea ar fi fost trimisă la serverul de origine.

3.1.4 Interfața Uniformă

Caracteristica centrală care distinge stilul arhitectural REST de celelalte stiluri bazate pe rețea este accentul pus pe o interfață uniformă între componente. Prin aplicarea principiului generalității în proiectarea interfețelor componentelor, arhitectura generală a sistemului este simplificată și vizibilitatea este îmbunătățită. Implementările sunt decuplate de serviciile pe care le oferă, care la rândul lor încurajează o evoluție independentă. Totuși, compromisul este că o interfață uniformă degradează eficiența, deoarece informațiile sunt transferate într-o formă standardizată, și nu aleasă în raport cu nevoile unei aplicații.

Pentru a obține o interfață uniformă, este nevoie de mai multe restricții arhitecturale care să ghideze comportamentul componentelor. REST este definit prin următoarele constrângeri de interfață:

1. identificarea resurselor
2. manipularea resurselor folosind reprezentări
3. mesaje autodescriptive
4. Hypermedia As The Engine Of Application State (HATEOAS)

Aceste restricții vor fi examinate în profunzime în Secțiunea 3.3, după exemplificarea utilizării lor în Secțiunea 3.2.

3.1.5 Sistem stratificat

Pentru a îmbunătăți în continuare comportamentul necesitat la scară largă, adăugăm restricția ca sistemul să fie stratificat. Acesta permite ca o arhitectură să fie compusă din straturi ierarhice prin constrângerea comportamentului componentelor, astfel încât fiecare componentă nu poate fi conștient de ceea ce este dincolo de stratul imediat cu care interacționează. Reducând cunoașterea sistemului la un singur strat, plasăm o limită pentru complexitatea generală a sistemului și promovăm independența substratului. Straturile pot fi utilizate pentru a încapsula servicii vechi și pentru a proteja noile servicii de clienții învechiți, simplificând componentele prin mutarea funcționalității frecvent utilizate într-un intermediar partajat. Intermediarii pot fi de asemenea folosiți pentru a îmbunătăți scalabilitatea sistemului, permițând echilibrarea volumului de procesare pe mai multe rețele și procesoare.

Principalul dezavantaj al sistemelor construite în acest fel este ca acestea adaugă overhead și latență la procesarea datelor, reducând performanța percepută de utilizator.

Revenind la exemplul pe care l-am dat în Secțiunea 2.2.4, în procesul de împărțire a monolitului în servicii mai mici, am dat peste un modul pe care îl voi numi modulul de Căutare. Cerința era clară, trebuiau luate toate funcționalitățile de căutare implementate în acel module și extrase într-un serviciu separat. Acest nou serviciu ar avea avantajul de a face uz de tehnologii noi și mai performante. Un lucru de reținut este faptul că formatul de răspuns returnat de acest nou serviciu s-ar diferenția semnificativ de cel returnat de modulul vechi, diferența provenită din faptul că exista și o cerință de a păstra o interfață uniformă pentru obiectele din domeniu, interfața definită după implementarea acelui modul vechi. Deoarece modulul de căutare vechi expunea o serie de endpoint-uri care erau utilizate de alte sisteme, pur și simplu nu puteam să introducem o schimbare atât de majoră odată cu prima lansare a serviciului.

Având în vedere aceste cerințe pentru noul serviciu, a trebuit să găsim o soluție care să ne permită să utilizăm parțial noul serviciu (pentru a-i demonstra utilitatea), dar să avem întotdeauna posibilitatea de a reveni la utilizarea modulului de căutare vechi în cazul în care s-ar fi descoperit o problemă majoră în noua implementare. Soluția aleasă, după părerea mea, este una simplă și elegantă, iar în cele ce urmează o să încerc să o prezint cât de bine posibil.

A trebuit să dezvoltăm un modul complet nou, al cărui singur scop ar fi să redirecționeze cererile de căutare interceptate de aplicația monolit către noul serviciu, în funcție de un *feature flag*, și să adapteze răspunsul primit de la serviciu, la formatul așteptat de client. Dezavantajul este, adaugarea de cod la un codebase deja saturat. Partea bună este că acest cod este temporar, deoarece atât modulul proxy, cât și modulul de cautare ar fi eliminate din cadrul monolitului odată ce toți clienții se adaptează noului serviciu.

3.2 Modelul de Maturitate Richardson

Un model dezvoltat de Leonard Richardson [17] care descompune elementele principale ale unei abordări REST pe trei nivele. Acestea introduc resurse, verbe HTTP și controale hypermedia. Modelul este un mod frumos de a ilustra utilizarea acestor tehnici, prin urmare voi încerca să le prezint în secțiunile următoare.

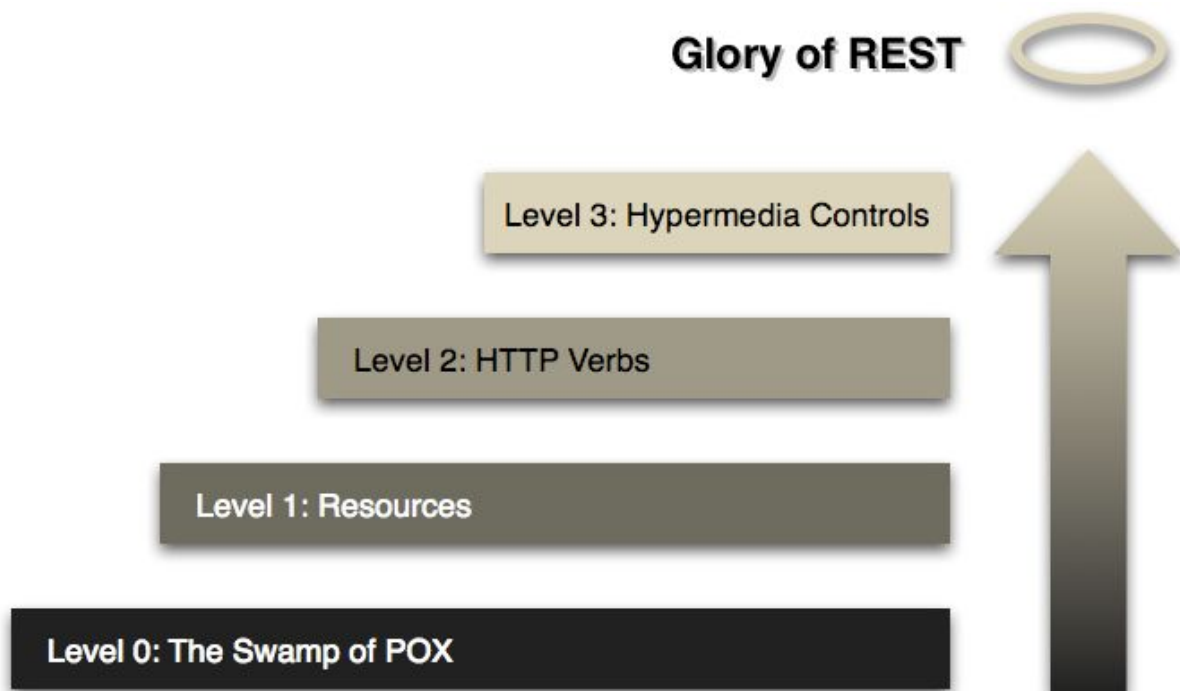


Fig. 3.1 Pașii către gloria REST

3.2.1 Nivelul 0

Punctul de plecare al modelului este utilizarea HTTP ca sistem de transport pentru interacțiuni la distanță, dar fără a utiliza niciunul dintre mecanismele web. În esență, ceea ce presupune acest nivel este utilizarea HTTP ca un mecanism de tunelare pentru propriul

mecanism de interacțiune la distanță, de obicei o formă de apel de procedură la distanță (RPC).

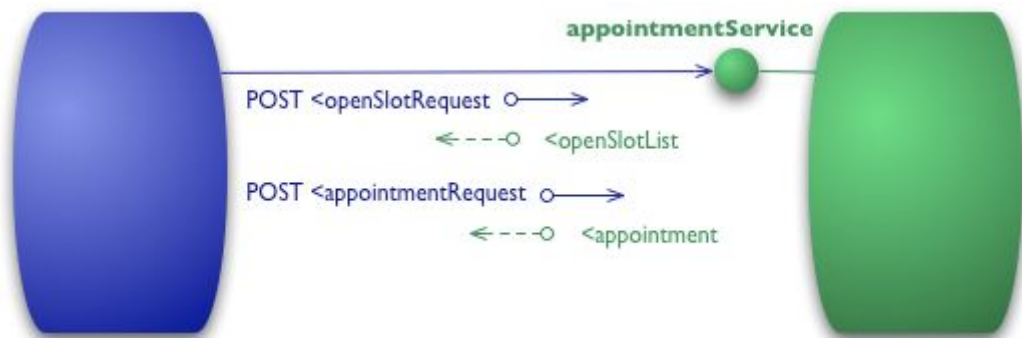


Fig. 3.2 Exemplu de interacțiune la nivelul 0

Să presupunem că vreau să-mi fac o programare la medic. Programul meu de rezervări trebuie să știe mai întâi ce sloturi deschise are medicul la o anumită dată, așa că face un request la serviciul de programări al spitalului pentru a obține aceste informații. Într-un scenariu la nivelul 0, spitalul ar expune un endpoint care ar trata toate request-urile. Apoi trimit către același endpoint un document descriind detaliile rezervării mele.

```
POST /appointmentService HTTP/1.1
Content-Type: application/xml

<openSlotRequest>
  <doctor>mjones</doctor>
  <date>2019-12-31</date>
</openSlotRequest>
```

Fig. 3.3 Exemplu de request la nivelul 0

Serverul ar trebui apoi să returneze un document care să-mi ofere informații ilustrate în Tabelul 3.1.

```
HTTP/1.1 200 OK
```

```
<openSlotList>
  <slot>
    <end>1450</end>
    <start>1400</start>
    <doctor id="mjones"/>
  </slot>
  <slot>
    <end>1650</end>
    <start>1600</start>
    <doctor id="mjones"/>
  </slot>
</openSlotList>
```

Tabel 3.1 Exemplu răspuns interogare la nivelul 1

Aceste exemple folosesc XML, dar conținutul poate fi de fapt orice: JSON, YAML, perechi cheie valoare sau orice alt format personalizat.

Următorul pas este rezervarea unei consultații, care poate fi făcută prin postarea unui document descriind programarea către endpoint-ul serviciului.

```
POST /appointmentService HTTP/1.1
Content-Type: application/xml

<appointmentRequest>
  <slot>
    <start>1400</start>
    <end>1450</end>
    <doctor id="mjones" />
  </slot>
  <patient id="Marius" />
</appointmentRequest>
```

Fig. 3.5 Request pentru crearea unei resurse la nivel 0

Dacă totul merge bine, serverul returnează un răspuns confirmând faptul că rezervarea a fost creată după cum se poate observa în Tabelul 3.2.

```
HTTP/1.1 200 OK
```



```

<appointment>
  <slot>
    <start>1400</start>
    <end>1450</end>
    <doctor id="mjones"/>
  </slot>
  <patient id="Marius"/>
</appointment>

```

Tabel 3.2. Exemplu de răspuns pentru crearea unei resurse la nivelul 0

Sau, dacă există o problemă, de pildă altcineva a creat o programare la ora dorită de noi, răspunsul ar conține un mesaj explicând eroarea, dar codul de retur ar fi tot 200 ca și în cazul unui request fără probleme:

HTTP/1.1 200 OK

```

<appointmentRequestFailure>
  <slot>
    <end>1450</end>
    <start>1400</start>
    <doctor id="mjones"/>
  </slot>
  <patient id="Marius"/>
  <reason>Slot not available</reason>
</appointmentRequestFailure>

```

Tabel 3.3. Răspuns de eroare la nivelul 0

Până acum acesta este un sistem stil RPC. Este simplu, întrucât doar pasează XML (Plain Old XML - POX) înainte și înapoi. SOAP sau XML-RPC este practic același mecanism, singura diferență fiind faptul că mesajele sunt trimise în interiorul unui plic.

3.2.2 Nivelul 1 - Resurse

The first step towards the glory of REST in the Richardson Maturity Model is to introduce resources. So now rather than making all our requests to a singular service endpoint, we now start talking to individual resources.

Primul pas spre gloria REST în modelul de maturitate Richardson este introducerea de resurse. Deci, în loc să facem toate solicitările către un singur endpoint, începem acum să vorbim despre resurse individuale.

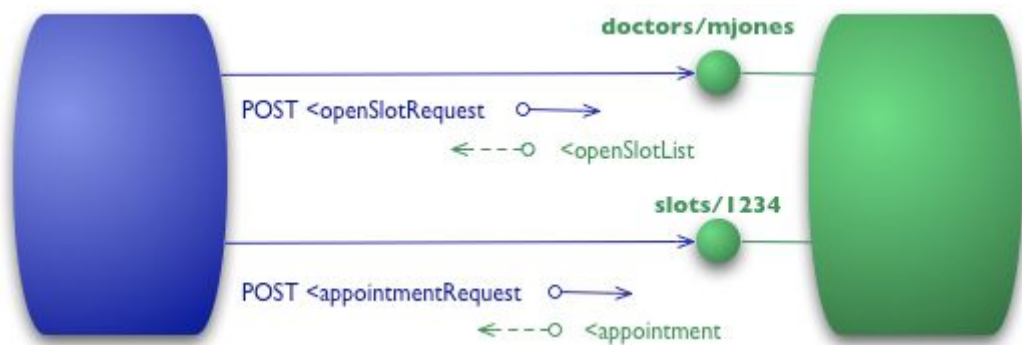


Fig. 3.4 Nivelul 1 cu resurse

Query-ul nostru inițial, având acum o resursă de tip doctor, se transformă în:

```
POST /doctors/mjones HTTP/1.1
Content-Type: application/xml

<openSlotRequest>
  <date>2019-12-31</date>
</openSlotRequest>
```

Fig. 3.5 Exemplu de request la nivelul 1

Răspunsul conține aceleași informații de bază, dar fiecare slot este acum o resursă care poate fi adresată individual:

```
HTTP/1.1 200 OK
```

```

<openSlotList>
  <slot>
    <id>1234</id>
    <end>1450</end>
    <start>1400</start>
    <doctor>mjones</doctor>
  </slot>
  <slot>
    <id>5678</id>
    <end>1650</end>
    <start>1600</start>
    <doctor>mjones</doctor>
  </slot>
</openSlotList>

```

Tabelul 3.4 Răspuns interogare conținând identificatori pentru resurse

Având resurse specifice, rezervarea unei consultații înseamnă postarea către un anumit slot:

```

POST /slots/1234 HTTP/1.1
Content-Type: application/xml

<appointmentRequest>
  <patient>
    <id>Marius</id>
  </patient>
</appointmentRequest>

```

Fig. 3.6 Crearea unei resurse la nivelul 1

Dacă totul merge bine, obținem o replică similară cu cea de la nivelul 0. Diferența constă în faptul că, dacă cineva are nevoie să modifice o rezervare, cum ar fi adăugarea unor analize, va obține mai întâi resursa pentru programare, care ar putea avea un URI precum `http://health.org/slots/1234/appointment`, și apoi va trimite request-uri la acea adresă. Cu alte cuvinte, în loc să apelăm o funcție în eter pasând o serie de argumente, apelăm o metoda care aparține unui anumit obiect.

3.2.3 Nivelul 2 - Verbe HTTP

Am folosit verb HTTP POST pentru interacțiunile exemplificate la nivelul 0 și 1. La aceste niveluri, verbul HTTP folosit nu face o atât de mare diferență, atât GET cat și POST sunt utilizate ca mecanisme care să permită tunelarea interacțiunilor folosind HTTP. Nivelul 2 se îndepărtează de această abordare, sugerând folosirea verbelor HTTP cât mai îndeaproape de modul în care sunt utilizate în HTTP.

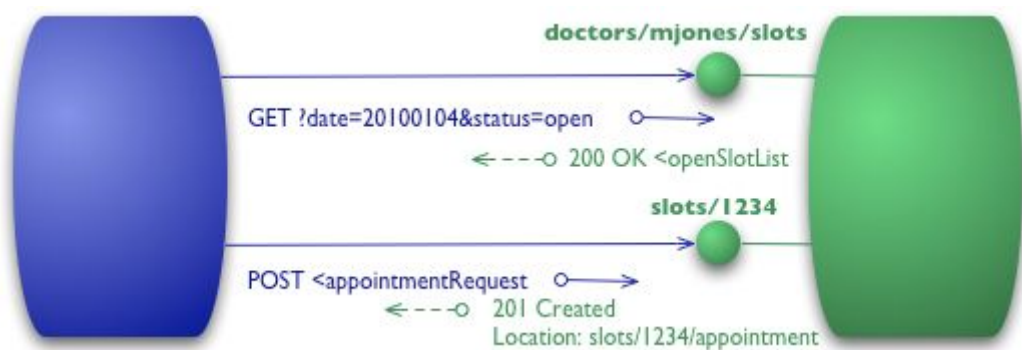


Fig. 3.7 Level 2 cu verbe HTTP

Prin urmare, pentru a obține list de sloturi va trebui sa efectuam un GET request.

```
GET /doctors/mjones/slots?date=20191231&status=open HTTP/1.1
Host: health.org
```

Fig. 3.8 Exemplu de request GET

La nivelul 2, utilizarea metodei GET pentru o astfel de solicitare este crucială. HTTP definește GET ca o operație sigură, adică nu face modificări semnificative niciunei resurse. Acest lucru ne permite să invocăm GET în siguranță de nenumărate ori în orice ordine și să obținem aceleași rezultate de fiecare dată. O consecință importantă este aceea că permite oricărui participant la rutarea cererilor să utilizeze cache, un element cheie în a face web-ul performant. HTTP include diferite măsuri pentru a suporta cache, care poate fi folosit de toți participanții la comunicare (conectori). Respectând regulile definite de standardul HTTP, putem profita de această capabilitate.

Pentru a rezerva o consultație avem nevoie de un verb HTTP care să schimbe starea, potrivit în această situație ar fi POST sau PUT. Requestul trimis este același ca la nivelul 1, diferența constă în modul în care serverul răspunde. Dacă totul merge bine, componenta server răspunde cu un cod de răspuns 201, care în vocabularul HTTP înseamnă ca requestul a fost acceptat sau, în acest caz, o nouă resursă a fost creată.

```
HTTP/1.1 201 Created
Location: slots/1234/appointment
. . .
```

Răspunsul 201 include un atribut de locație care conține un URI pe care clientul îl poate utiliza pentru a obține starea acelei resurse în viitor. Răspunsul include și o reprezentare a resursei nou create pentru a salva clientul de la un apel suplimentar.

O altă diferență este că în cazul în care ceva nu merge bine, cum ar fi un conflict pe același slot, serviciul returnează 409 Conflict. Partea importantă a acestui răspuns este utilizarea unui cod de răspuns HTTP pentru a indica faptul că ceva nu a merg bine. În acest caz, codul de retur 409 este o alegere bună pentru a indica faptul ca altcineva a actualizat deja resursa într-un mod incompatibil. În loc să folosim codul de return 200, incluzând un mesaj de eroare în răspuns, nivelul 2 promovează folosirea explicită codurilor de retur. Depinde de proiectantul de protocol să decidă ce coduri să folosească, dar ar trebui să existe un răspuns non-2xx în cazul unei erori. Nivelul 2 introduce folosirea verbelor și a codurilor de return HTTP.

3.2.4 Nivelul 3 - Controale Hypermedia

Nivelul final introduce ceva la care se face referire sub acronimul HATEOAS (Hypertext as The Engine Of Application State). Acesta abordează întrebarea cum să treci dintr-o listă de sloturi deschise până la a ști ce să faci pentru a rezerva o consultație.

Începem cu aceeași solicitare GET inițială pe care am trimis-o la nivelul 2, dar răspunsul are un element nou, un link care conține un URI care să ne spună cum să rezervăm o programare. Ideea controalelor hypermedia este că acestea ne spun ce putem face în continuare, precum și URI-ul resursei pe care trebuie să o manipulăm. Spre deosebire de a ști noi exact unde trebuie trimis un request, controalele hipermedia din răspuns de spun cum să o facem.

Un beneficiu evident al controalelor hypermedia este ca acestea permit serverului să-și schimbe schema URI fără a cauza erori clienților. Atâta timp cât clienții folosesc linkurile în răspuns, echipa responsabilă de server poate jongla cu URI-urile, cu excepția entypoint-ului, a cărui modificare în mod sigur ar cauza erori.

3.2.5 Rezumat

În această secțiune am discutat despre modelul de maturitate Richardson, care este o modalitate bună de a raționa despre elementele arhitecturii REST, dar nu este o definiție a arhitecturii în sine. Roy Fielding [18] , în lucrarea sa de disertație, spune ca nivelul 3 (controalele hypermedia) sunt o condiție a arhitecturii REST. Nivelurile sunt foarte similare cu modul în care șabloanele de proiectare sunt utilizate pentru a rezolva o problema care are un model de soluție consacrat:

- nivelul 1 abordează problema gestionării complexității prin folosirea tehnicii *divide and conquer*, împărțind un singur endpoint în mai multe resurse;
- nivelul 2 introduce un set standard de verbe, astfel încât să ne descurcăm în situații similare în același, îndepărtând variația inutilă;
- nivelul 3 instituie descoperire, oferind o modalitate de face un protocol auto-documentat.

3.3 Date

Spre deosebire de stilul RPC, unde toate datele sunt încapsulate și ascunse în interiorul elementelor de procesare, natura și starea elementelor de date este un aspect cheie al arhitecturii REST. Motivul pentru acest design poate fi observat în natura sistemelor hypermedia distribuite. Atunci când un link este selectat, informația trebuie să fie mutată din locația în care este stocată în locația în care va fi utilizată (de un cititor uman în cele mai multe cazuri). În paradigmele de procesare distribuite este mai eficient, de obicei, mutarea “agentului de procesa”. De exemplu, este mai eficient să mutăm expresia de căutare mai aproape de date, decât să mutăm datele.

Există trei opțiuni de luat în considerare în proiectarea unei arhitecturi hipermedia distribuite:

1. randarea datelor unde sunt stocate și trimiterea unui format fix clientului (de exemplu, server side HTML rendering);
2. încapsularea datelor într-un motor de randare;
3. trimiterea datelor originale, fără a fi randate, dar incluzând meta informații care să ajute clientul în a-și alege un motor de randare.

Fiecare opțiune are plusurile și minusurile ei. Opțiunea 1., stilul tradițional client-server, permite ca toate informațiile despre adevărata natură a datelor să rămână ascunse în interiorul serverului, împiedicând crearea de ipoteze false cu privire la structura de date și facilitând implementarea clientului. Astfel, funcționalitate clientului este sever redusă, iar cea mai mare parte a procesării revine serverului, ceea ce duce la probleme de scalabilitate. Opțiunea 2. oferă informații care ascund datele, dar în același timp permit prelucrarea specializată a datelor prin intermediul motorului de randare specializat/ Opțiunea 3. permite expeditorului să rămână simplu și scalabil în timp ce minimizează numărul de octeți transferați, dar pierde avantajele încapsulării datelor și impune atât serverului cât și clientului să înțeleagă aceleași tipuri de date.

REST provides a combination of all three options by focusing on a shared understanding of data types with metadata, but limiting the scope to what is revealed to a standardized interface. REST components communicate by transferring a representation of a resource in a standard format, selected dynamically based on the capabilities or desires of the

recipient and the nature of the resource. Whether the representation is in the same format as the raw source, or it is derived from the source, remains hidden behind the interface.

REST oferă o combinație a celor trei opțiuni, concentrându-se pe o înțelegere comună a tipurilor de date meta informații, dar limitând scopul la ceea ce este dezvăluit unei interfețe standardizate. Componentele REST comunică prin transferul unei reprezentări a unei resurse într-un format standard, selectat dinamic în funcție de capacitățile sau dorințele destinatarului și natura resursei. Indiferent dacă resursa este stocată în același format expus, sau în alt format, reprezentarea internă rămâne ascunsă în spatele interfeței.

Elementele de date caracteristice arhitecturii REST sunt sumarizate în Tabelul 3.5:

Element	Exemplu
resursă	Ținta unei referințe hypertext
identificator de resursa	URL, URN
reprezentare	JSON, XML sau document HTML, imagine JPEG
metadate de reprezentare	Media type, last-modified time
metadate de resursa	vary
date de control	If-modified-since, cache-control

3.3.1 Resurse și Identificatori de Resurse

Principala abstractizare a informațiilor în REST este o resursă. Orice informație care are nume poate fi o resursă: un document sau o imagine, un serviciu temporal (de exemplu “vremea de azi”), o colecție de alte resurse și așa mai departe. Unele resurse sunt statice în sensul că, atunci când sunt examinate în orice moment după crearea lor, ele corespund întotdeauna aceleiași valori. Un exemplu din ingineria software este identificarea separată a unui fișier de cod sursă aflat în contextul unui sistem de versionare. Fișierul poate avea eticheta “v2.0” sau “ultima revizuire”, “v2.0” întotdeauna va referi aceeași versiune, pe când “ultima revizuire” se va modifica în timp.

REST utilizează un **identificator de resurse** pentru a identifica o anumită resursă implicată într-o interacțiune între componente. Un identificator URI identifică o resursă fie în funcție de locație, fie după nume, fie în funcție de ambele. URI are două specializări cunoscute sub numele de URL și URN.

Un localizator de resurse uniform (URL) este un subset al identificatorului de resurse uniforme (URI) care specifică unde este disponibilă o resursă identificată și mecanismul de preluare a acesteia. URL definește modul în care poate fi obținută resursa. Nu trebuie să fie o adresă URL HTTP (`http://`), poate fi, de exemplu `ftp://`.

A Uniform Resource Name (URN) is a URI that uses the URN scheme, and does not imply availability of the identified resource. Both URNs and URLs are URIs, and a particular URI may be both a name and a locator at the same time. A URN is similar to a person's name, while a URL is like a street address, essentially, “what” vs. “where”. We can use the examples in the URI Request for Comments [19] to evidentiate these differences:

Un nume de resursă uniform (URN) este un URI care utilizează schema URN și nu implică disponibilitatea resursei identificate. Atât URN-urile, cât și adresele URL sunt URI-uri, iar un anumit URI poate fi atât un nume, cât și un localizator în același timp. O adresă URN este similară cu numele unei persoane, în timp ce o adresă URL este ca o adresă de stradă, în esență “ce” față de “unde”. Putem folosi exemplele din RFC care formalizează URI-ul [19] pentru a evidenția aceste diferențe:

URL: <code>ftp://ftp.is.co.za/rfc/rfc1808.txt</code>	
URL: <code>http://www.ietf.org/rfc/rfc2396.txt</code>	
URL: <code>ldap://[2001:db8::7]/c=GB?objectClass=one</code>	
URL: <code>mailto:John.Doe@example.com</code>	
URL: <code>news:comp.infosystems.www.servers.unix</code>	
URL: <code>telnet://192.0.2.16:80/</code>	
URN	(not URL) :
<code>urn:oasis:names:specification:docbook:dtd:xml:4.1.2</code>	
URN (not URL): <code>tel:+1-816-555-1212 (?)</code>	

3.3.2 Reprezentări

Componentele REST efectuează acțiuni asupra unei resurse folosind o reprezentare pentru a capta starea curentă sau intenționată a resursei respective și transferarea reprezentării între componente. O reprezentare este o secvență de octeți, plus metadatele de reprezentare, pentru a oferi meta-informații despre resursa care nu sunt specifice reprezentării. Datele de control

4. Aplicația Practică

4.1 Analiza Cerințelor

Pentru o înțelegere mai bună a aplicației practice alese am decis că o prezint și motivul din spatele ei. A trecut ceva timp de când tatăl meu m-a rugat în mod repetat să creez un site de prezentare pentru compania sa de construcții. Nu vă imaginați ca deține o companie mare, dacă acesta ar fi fost cazul, atunci site-ul ar fi deja construit. Este mai mult un tip de afacere de familie, fratele meu de asemenea lucrează cu el. Aplicația inițială pentru această lucrare a fost o platformă de blog care ar fi prezentat modul în care am aplicat principiile REST. Deși platformă de blogging nu era o alegere eronată, ar fi fost generică și fără prea multă aplicabilitate. Acesta a fost și motivul pentru care am decis că aplicația care va acompania această lucrare va fi site-ul de prezentare menționat mai sus, implementat folosind un set de componente descrise în Secțiunea 4.2.

Site-ul de prezentare are un meniu simplu, în care utilizatorul poate răsfoi catalogul de proiecte existente. Autentificare nu este necesară în cadrul acestei aplicații deoarece aceasta nu are funcționalități de alterare a conținutului, iar datele prezentate sunt public disponibile.

Există și un site web folosit pentru administrarea conținutului, permițând unor administratori să adauge proiecte noi sau să le actualizeze pe cele existente. Această aplicație nu trebuie impună nicio limită asupra operațiilor ce pot fi efectuate asupra resurselor disponibile.

4.2 Design

Aplicând abordarea *API first design*, aplicația este împărțită în patru componente: api, admin, client și o componentă cache care are rolul de a îmbunătăți performanța percepută de utilizator. API-ul este punctul central al sistemului, fiind o aplicație PHP dezvoltată folosind framework-ul Symfony. Am ales să folosesc acest framework deoarece simplifică dezvoltarea aplicației, permițându-i ai degrabă să mă concentrez asupra aspectelor specifice aplicației, decât asupra aspectelor banale și deja implementate. În acest sens, este important de notat că un framework poate fi considera un middleware între logica specifică aplicației și

complexitatea exterioară necesară pentru a face aplicația utilizabilă. API-ul conține toate datele și logica pentru manipularea și expunerea operațiilor permise printr-o serie de endpoint-uri. Are de asemenea un endpoint pentru expunerea documentației în formatul introdus de Specificația OpenAPI, cunoscută anterior drept Specificația Swagger.

Specificația OpenAPI este un format de descriere API pentru API-urile REST. Un fișier OpenAPI permite descrierea întregului API, inclusiv:

- endpoint-urile disponibile (`/projects`) și operațiile permise pentru fiecare endpoint (`GET /projects`, `POST /projects`);
- parametri de input și output pentru fiecare operație;
- metode de autentificare
- informații de contact, licența, termeni de utilizare și alte informații

API specifications can be written in YAML or JSON, but annotations can also be used so that documentation resides in the same place with the actual code, and they are later transformed into JSON. This has the advantage the sandbox environments can be easily created based on the supplied specification as it can be observed in Fig. 4.1

Specificațiile API pot fi scrise în YAML sau JSON, adnotările pot fi de asemenea utilizate. Adnotările aduc beneficiul de a avea documentația în același fișier în care se află și codul sursă, minimizând astfel riscul de a expune o documentație învechită. O consecință pozitivă a acestor specificații este că se pot crea medii în care API-ul poate fi testat de dezvoltatori pentru a se familiariza cu funcționalitățile implementate. O astfel de interfață generată pe baza documentației se poate observa în Fig. 4.1. Componenta API din cadrul aplicației este cea care expune această interfață. Pe baza header-ului `Accept`, API returnează documentația într-un format sau altul. De pildă, dacă formatul acceptat de client este `text/html` (acesta este formatul în momentul în care request-ul este efectuat folosind un browser) atunci varianta HTML va fi returnată. Pe de altă parte, dacă un client declară că vrea `application/ld+json`, atunci o versiune care poate fi mai bine înțeleasă de mașinile de calcul este returnată. JSON for Linked Data (JSON-LD) este o metodă de reprezentare a datelor folosind sintaxa JSON. A fost un obiectiv de a solicita cât mai puțin efort din partea dezvoltatorilor pentru a-și transforma JSON-ul existent în JSON-LD. Acest lucru permite serializarea datelor într-un mod care este similar cu JSON-ul tradițional. Este o recomandare a World Wide Web Consortium (W3C).

Image		▼
GET	/images	Retrieves the collection of Image resources.
POST	/images	Creates a Image resource.
GET	/images/{id}	Retrieves a Image resource.
PUT	/images/{id}	Replaces the Image resource.
DELETE	/images/{id}	Removes the Image resource.
Person		▼
GET	/people	Retrieves the collection of Person resources.
POST	/people	Creates a Person resource.
GET	/people/{id}	Retrieves a Person resource.
DELETE	/people/{id}	Removes the Person resource.
PUT	/people/{id}	Replaces the Person resource.
ProjectPhoto		▼
GET	/project_photos	Retrieves the collection of ProjectPhoto resources.
POST	/project_photos	Creates a ProjectPhoto resource.
GET	/project_photos/{id}	Retrieves a ProjectPhoto resource.
DELETE	/project_photos/{id}	Removes the ProjectPhoto resource.
PUT	/project_photos/{id}	Replaces the ProjectPhoto resource.
GET	/projects/{id}/photos	Retrieves the collection of ProjectPhoto resources.
Project		▼
GET	/projects	Retrieves the collection of Project resources.
POST	/projects	Creates a Project resource.
GET	/projects/{id}	Retrieves a Project resource.
DELETE	/projects/{id}	Removes the Project resource.
PUT	/projects/{id}	Replaces the Project resource.
GET	/projects/{id}/photos	Retrieves the collection of ProjectPhoto resources.

Fig. 4.1 Documentația API-ului interactivă în format HTML

Fiecare endpoint are propria documentație și un model pentru a trimite un request:

The screenshot displays an API client interface for a POST request to the `/people` endpoint, which is described as "Creates a Person resource." The interface includes a "Parameters" section with a table listing the request body parameter.

Name	Description
person (body)	The new Person resource

Below the table, there is an "Example Value" section showing a JSON object:

```
{  "name": "string",  "email": "string",  "ledProjects": [    "string"  ],  "image": "string"}
```

The interface also features a "Parameter content type" dropdown menu set to `application/ld+json` and an "Execute" button. At the bottom, the "Responses" section shows the "Response content type" set to `application/ld+json`.

Fig. 4.2 Exemplu de endpoint POST

API-ul suportă și vocabularul Hydra Core. Hydra este un vocabular simplu pentru crearea de API-uri hypermedia. Specificând o serie de concepte utilizate în mod obișnuit în API-urile web, permite crearea de clienți API generici. Implementarea acestui vocabular înseamnă expunerea unui endpoint care definește resursele posibile, proprietățile lor și operațiile asociate. Această caracteristică a fost valorificată pentru a construi interfața de administrare, care trebuie să realizeze CRUD pe resursele expuse de API.

Un aspect important de menționat aici este faptul ca aplicația Admin este menită să fie folosită doar intern, aplicația care se va confrunta cu o mare varietate de utilizatori este aplicația Client. Aceasta este o aplicație de tip Single Page Application care prezintă date obținute de la API. Pentru a îmbunătăți performanța, o componentă cache se află în fața API-ului. Acest lucru înseamnă ca aplicația Client este configurată să efectueze request-uri către componenta cache, care la rândul său transmite requestul mai departe către API dacă nu poate furniza datele solicitate. Din perspectiva componentei client, cache-ul este API-ul propriu-zis, deoarece nu știe despre alte componente dincolo de cele cu care comunică, după cum este menționat în Secțiunea 3.1.5.

The chosen representation for interchanging resources is JSON-LD because it is lightweight, human readable and fairly easy to decode / encode. The data is internally persisted using a RDMS. The model can be observed in Fig. 4.1.

Codificarea aleasă pentru transmiterea reprezentărilor între componente este JSON-LD. Printre avantajele acestei codificări se numără faptul ca este simplu de implementat și poate fi înțeleasă textual de oameni, facilitând depanarea. Intern, datele sunt persistate folosind o baza de date relațională. Model poate fi observat în figura de mai jos:

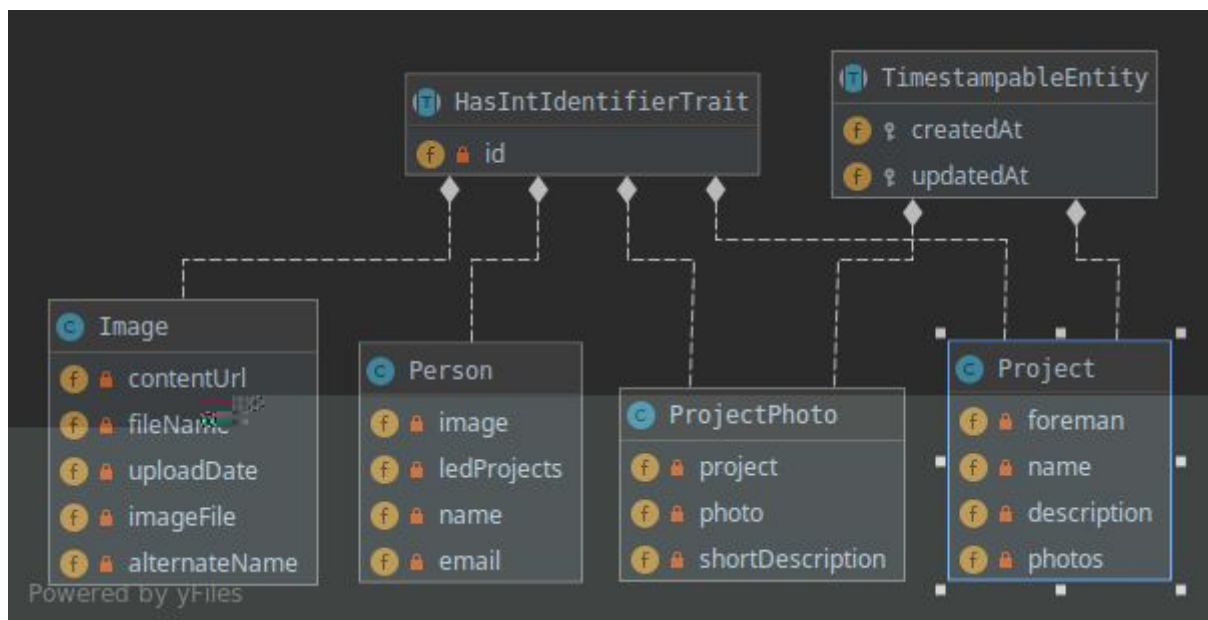


Fig. 4.3 Diagrama de clase a entităților

Interfața componentei Admin are la bază componente React proiectate folosind Material Design, ecranul ce listează imaginile încărcate poate fi observat în Fig. 4.5.

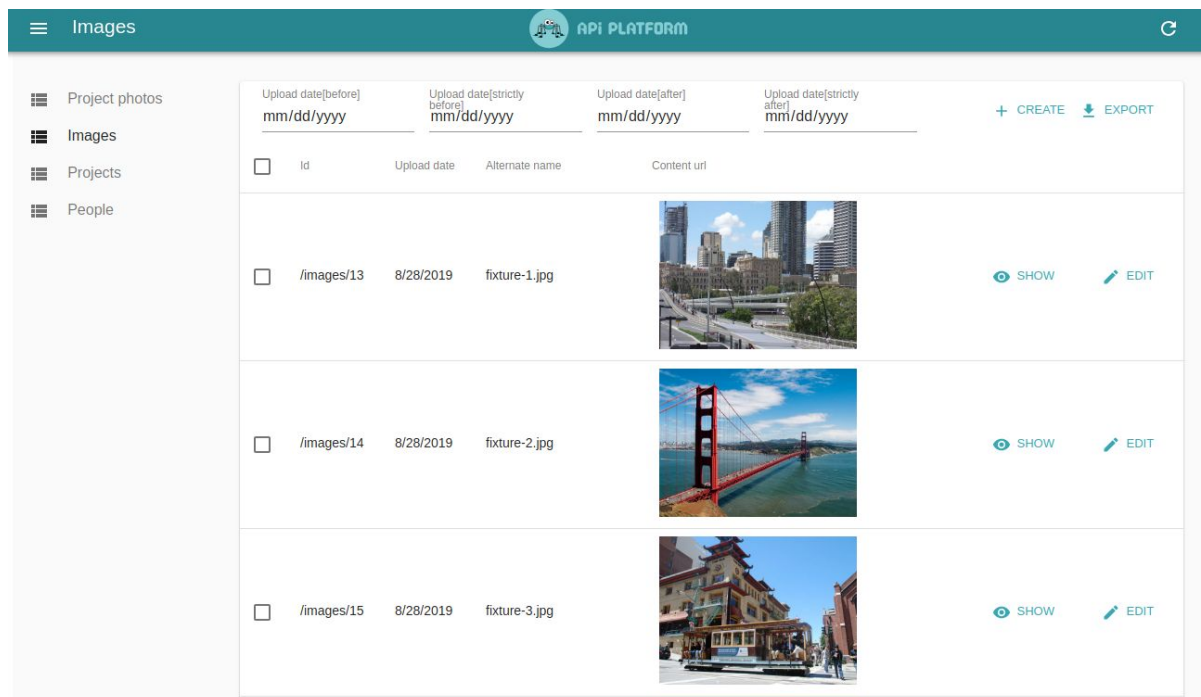


Fig. 4.5 Ecranul listare de imagini din cadrul componentei Admin

Luând ca și exemplu următorul caz de utilizare, dacăun administrator dorește să adauge o imagine nouă unui proiect existent, acesta trebuie să efectueze urmatorii pași:

- să creeze resurse de tip Image (Fig. 4.6.);

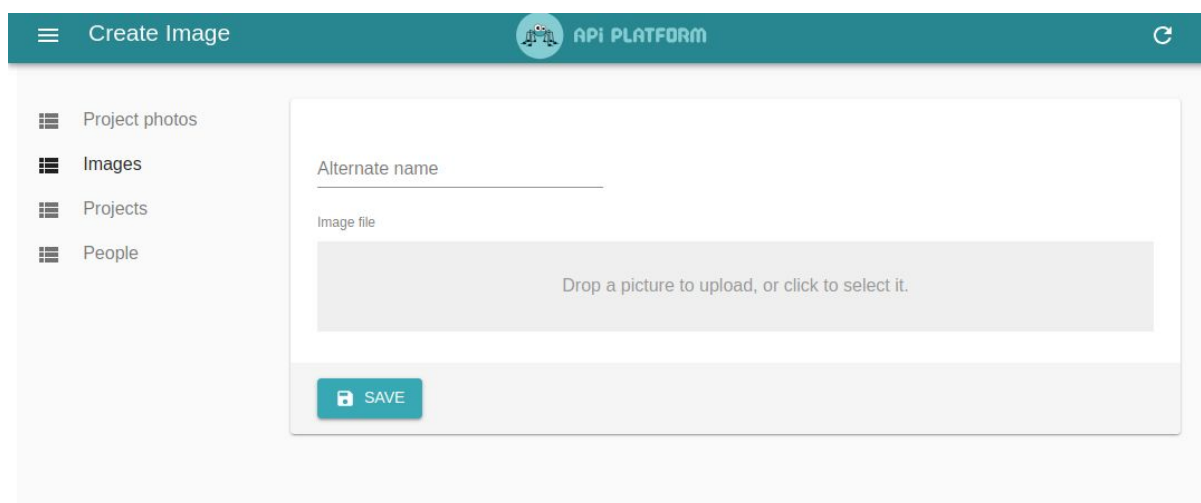


Fig. 4.6. Ecranul de creare de imagini

- să asocieze imaginea nou creată cu un proiect prin crearea unei resurse de tip ProjectPhoto (Fig. 4.7.).

Fig. 4.7. Ecranul de creare de asocieri între imagini și proiecte

4.3 Testare

Componenta API este singura care deține logică, prin urmare singura care necesită testare. Componentele Admin și Client sunt preocupate în cea mai mare parte de aspect și estetică, din acest motiv testarea manuală pentru acestea este suficientă. În dezvoltarea API-ului s-a aplicat paradigma Test Driven Development. Asta înseamnă, în mare, că de fiecare dată când am decis ca un nou endpoint sau operație la un endpoint trebuie implementate, trebuia să am un test care verifice funcționalitatea lipsă. Un exemplu de astfel de test se poate observa în figura următoare:

```
public function testPersonEmailMustBeUnique(): void
{
    // a person with this email is already added by fixtures
    $response = $this->sendCreatePersonRequest( name: 'test', email: 'daniel@fake.com');
    self::assertResponseStatusCodeSame( expectedCode: 400);

    self::assertEquals(
        expected: 'email: This value is already used.',
        $this->hydraDescription($this->jsonDecode($response))
    );
}
```

Fig. 4.8 Exemplu de test

Framework-ul de testare folosit a fost PHPUnit, care seamănă foarte mult cu JUnit din Java. Este o instanță a arhitecturii xUnit [20], arhitectură folosită pentru framework-urile de testare care fost prima dată implementată în SUnit și a devenit populară odată cu JUnit. Raportul de acoperire al codului din cadrul componentei API se poate observa în Fig. 4.5:
















	Code Coverage								
	Lines			Functions and Methods			Classes and Traits		
Total		91.19%	145 / 159		88.52%	54 / 61		66.67%	6 / 9
Controller		86.05%	37 / 43		62.50%	5 / 8		0.00%	0 / 1
Entity		91.49%	86 / 94		91.49%	43 / 47		66.67%	4 / 6
EventListener		100.00%	11 / 11		100.00%	3 / 3		100.00%	1 / 1
Kernel.php		100.00%	11 / 11		100.00%	3 / 3		100.00%	1 / 1

Fig. 4.9 Acoperirea codului din componenta API

4.4 Tratarea cererilor de Imagini într-o Manieră RESTful

Una dintre caracteristicile componentei API este capacitatea de a

```

public function __invoke(Request $request): Image
{
    $contentType = $request->headers->get( key: 'Content-Type');
    if ('application/ld+json' === $contentType) {
        return $this->handleApplicationJsonRequest($request);
    }

    return $this->handleMultipartFormDataRequest($request);
}

private function handleApplicationJsonRequest(Request $request): Image
{
    [$imageFile, $requestFields] = $this->extractImageFile($this->getDecodedBody($request));
    $image = $this->denormalizeImage($requestFields);
    $image->setImageFile($this->createUploadedFile($imageFile));

    return $image;
}

private function createUploadedFile(string $base64EncodedFile): UploadedFile
{
    // here we fake a file upload in order to leverage vich uploader bundle which checks for UploadedFile instances
    [$prefix, $base64EncodedImage] = $this->extractImagePrefixAndContents($base64EncodedFile);
    $extension = $this->getExtensionFromBase64Prefix($prefix);
    $temporaryFilename = $this->createTemporaryImageFile();
    $this->saveImageTo($temporaryFilename, $base64EncodedImage);
    $originalName = $this->getOriginalName($temporaryFilename, $extension);

    return new UploadedFile($temporaryFilename, $originalName, mimeType: null, error: null, test: true);
}

```

Fig. 4.10 Implementarea încărcării de imagini

4.5 Direcții Viitoare

Future work will focus on enhancing the user experience, for both Admin and Client applications. User-perceived performance can be improved by reducing the latency caused induced by image loading through using a content delivery network for media storage. A content delivery network or content distribution network (CDN) is a geographically distributed network of proxy servers and their data centers. Their goal is to provide high availability and high performance by distributing the service spatially relative to end-users.

Direcțiile de dezvoltare viitoare se vor concentra pe îmbunătățirea experienței utilizatorului, atât pentru componenta Admin, cât și pentru componenta Client. Performanța percepută de utilizator poate fi îmbunătățită prin reducerea latenței cauzate de încărcarea de imagini prin utilizarea unei rețele de livrare de conținut pentru stocarea resurselor media. O rețea de distribuție de conținut sau o rețea de livrare de conținut (CDN) este o rețea distribuită geografic de servere proxy. Obiectivul lor este de a oferi o disponibilitate și performanță ridicată prin distribuirea serviciului spațial relativ la utilizatorii finali.

Un alt mod în care aplicația ar putea fi îmbunătățită este integrarea cu platformele de socializare. Ideea din spatele acestei integrări este următoarea: de fiecare dată când ceva nou este publicat pe site-ul de prezentare (componenta Client), un event va fi publicat într-o coadă de mesaje; va exista un alt serviciu care citește asincron din coada de mesaje și execută toate sarcinile necesare pentru actualizarea conturilor de social media aferente. În acest fel, aplicația câștigă vizibilitate prin intermediul mai multor canale.

Întrucât numărul de componente ar putea crește, depanarea problemelor devine o sarcină destul de dificilă. O eficientizare pe termen lung ar fi adăugarea de monitorizare și logare centralizată. Un avantaj obținut prin adăugarea monitorizării este acela că permite detectarea rapidă a problemelor cauzate fie de erori umane, greșeli de configurare, sau factori de mediu. Jurnalizarea centralizată oferă două beneficii importante. În primul rând, plasează toate înregistrările de date de jurnal într-o singură locație, facilitând astfel analiza și corelarea jurnalului. În al doilea rând, logarea centralizată ajută la a păstra partițiile de disc ale aplicație statice și minimizare operațiilor IO pe servere, maximalizând performanța și reducând costurile.

Conclu ii

În această lucrare am discutat despre principiile care stau la baza stilului arhitectural REST. Am început prin a defini *componentele*, *conectorii* și *datele*, elementele de bază constrânse în relațiilor lor pentru a realiza un set dorit de proprietăți arhitecturale. Apoi am continuat printr-o paralelă între arhitecturile bazate pe rețea și arhitecturile distribuite. Același capitol evidențiază o serie de proprietăți cheie ale sistemelor întemeiate pe rețea, precum: scalabilitatea, simplitatea și modificabilitatea. Capitolul imediat următor abordează tema centrală a lucrării.

REST este un set coordonat de constrângeri arhitectonice care încearcă să minimalizeze latența și comunicarea în rețea, totodată maximalizând independența și scalabilitatea componentelor. Acest lucru este realizat prin plasarea constrângerilor pe semantica conectorilor, în locurile în care alte stiluri s-ar fi concentrat mai mult pe semantica componentelor. REST permite caching-ul și reutilizarea interacțiunilor, substituirea dinamică a componentelor și procesarea acțiunilor de către intermediari, răspunzând astfel nevoilor unui sistem hypermedia distribuit la nivelul Internetului.

Lucrarea este încheiată cu un capitol dedicat aplicație, implementată atât pentru a pune în practică conceptele prezentate, cât și pentru a fi dezvoltată în continuare, urmând ca apoi să fie folosită. Aplicație este compusă dintr-o componentă API și două componente client care efectuează request-uri către acesta. Componenta Admin este o aplicație de administrare de conținut, pe când rolul componentei Client este de a prezenta conținutul în cea mai bună manieră.

În concluzie, Representational State Transfer (REST) este un stil arhitectural care definește un set de restricții care trebuie utilizate pentru crearea serviciilor web. Serviciile Web RESTful oferă interoperabilitate între sistemele informatice de pe internet, permit sistemelor solicitante să acceseze și să manipuleze reprezentări textuale ale resurselor folosind un set uniform și predefinit de operații fără stare.

Bibliografie

1. M. Shaw. Toward higher-level abstractions for software systems. *Data & Knowledge Engineering*, 5, 1990, pp. 119–128.
2. L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison Wesley, Reading, Mass., 1998.
3. D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4), Oct. 1992, pp. 40–52.
4. M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young, and G. Zelesnick. Abstractions for software architecture and tools to support them. *IEEE Transactions on Software Engineering*, 21(4), Apr. 1995, pp. 314–335.
5. M. Shaw and P. Clements. A field guide to boxology: Preliminary classification of architectural styles for software systems. In *Proceedings of the Twenty-First Annual International Computer Software and Applications Conference (COMPSAC'97)*, Washington, D.C., Aug. 1997, pp. 6–13.
6. M. Boasson. The artistry of software architecture. *IEEE Software*, 12(6), Noi. 1995, pp. 13–16.
7. Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*, Mar. 2009
8. A. Fuggetta, G. P. Picco, and G. Vigna. Understanding code mobility. *IEEE Transactions on Software Engineering*, 24(5), Mai 1998, pp. 342–361.
9. G. Andrews. Paradigms for process interaction in distributed programs. *ACM Computing Surveys*, 23(1), Mar. 1991, pp. 49–90.
10. R. N. Taylor, N. Medvidovic, K. M. Anderson, E. J. Whitehead Jr., J. E. Robbins, K. A. Nies, P. Oreizy, and D. L. Dubrow. A component- and message-based architectural style for GUI software. *IEEE Transactions on Software Engineering*, 22(6), Iun. 1996, pp. 390–406.
11. A. S. Tanenbaum and R. van Renesse. Distributed operating systems. *ACM Computing Surveys*, 17(4), Dec. 1985, pp. 419–470.
12. J. Waldo, G. Wyant, A. Wollrath, and S. Kendall. A note on distributed computing. Technical Report SMLI TR-94-29, Sun Microsystems Laboratories, Inc., Noi. 1994.
13. Robert C. Martin. *Design Principles and Design Patterns*. 2000.

14. P. Bernstein. Middleware: A model for distributed systems services. Communications of the ACM, Feb. 1996, pp. 86–98.
15. Charlie Collins, Michael Galpin and Matthias Kaeppeler. Android in Practice. Manning Publications, 2011.
16. Moore, M. E. (2006). Introduction to the Game Industry. Pearson Prentice Hall. p. 169.
17. Leonard Richardson, Published on the web and presented at QCon conference <<https://www.crummy.com/writing/speaking/2008-QCon/act3.html>>, 2008.
18. Roy Fielding, REST APIs must be hypertext-driven, Publied online <<https://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven>>, 2008.
19. T. Berners -Lee, W3C, R. Fielding Day Software, L. Masinter Adobe Systems. RFC3986 - Uniform Resource Identifier (URI): Generic Syntax <<https://www.ietf.org/rfc/rfc3986.txt>>, Jan. 2005.
20. Gerard Meszaros, xUnit Test Patterns, 2007
21. Roy T. Fielding. Architectural Styles and the Design of Network-Based Software Architectures, University of California, Irvine, 2000.