

Universitatea din București
Facultatea de Matematică și Informatică

ONLINE BACKLOG

Profesor:

Conf. dr. Kevorchian Cristian

Student:

Georoceanu Alexandru Mihai

București 2011

Cuprins

Introducere.....	3
CAPITOLUL 1 Product Backlog.....	4
1.1 Noțiuni necesare.....	4
1.1.1 Colecția de metodologii Agile.....	4
1.1.2 O metodologie particulară – Scrum	7
1.2 Descriere backlog.....	9
CAPITOLUL 2 Dezvoltarea în cloud.....	12
2.1 Introducere.....	12
2.2 Cloud computing	12
2.2.1 Clasificarea UCSB-IBM.....	12
2.2.2 Software as a Service (SaaS).....	13
2.2.3 Platform as a Service (PaaS).....	14
2.3 Platforme și servicii existente.....	15
2.4 Avantaje și dezavantaje.....	16
2.5 Securitatea cloud-ului.....	17
CAPITOLUL 3 Infrastructura Google pentru SaaS.....	19
3.1 Prezentarea serviciului GAE.....	19
3.1.1 Mediul aplicației.....	19
3.1.2 Separarea aplicațiilor și limite.....	20
3.2 Python în Google App Engine.....	21
3.2.1 Selectarea mediului de programare Python.....	21
3.2.2 Cereri și domenii.....	21
3.2.3 Python pur și simplu.....	22
3.3 Modelarea datelor în Datastore sub Python.....	23
3.3.1 Introducere în Datastore.....	23
3.3.2 API-ul Python pentru Datastore.....	24
3.3.3 Entități și proprietăți.....	25
3.3.4 Diferențe față de SQL.....	25
3.3.5 Modelarea entităților.....	26
3.4 Serviciul de autentificare Google.....	28
3.4.1 Generalități.....	28
3.4.2 Autentificarea utilizatorilor în Python.....	29
3.4.3 Securizarea prin autentificare a resurselor.....	29
3.4.4 Utilizatorii și Datastore-ul.....	30
3.5 Servicii adiționale oferite de GAE.....	31
CAPITOLUL 4 Arhitectura REST.....	32
4.1 Ce este REST.....	32
4.2 Conceptul.....	32
4.3 Constrângeri.....	33
4.4 Țeluri principale.....	34
4.5 Webservice-uri RESTful.....	34
CAPITOLUL 5 Prezentarea aplicației.....	36
5.1 Generalități.....	36
5.2 Arhitectura.....	36
5.3 Implementarea serverului.....	38
5.3.1 Modelarea entităților.....	41
5.3.2 Securizarea webservice-ului.....	42
5.4 Implementarea clientului REST (interfeței utilizatorului).....	43
Bibliografie.....	48

Introducere

Obieciul principal pe care această lucrare se concentrează este de a prezenta baza noilor tehnologii de calcul distribuit alături de paradigmele ce se află spatele acestora. O dată cu evoluția sistemelor de calcul distribuit au apărut noi posibilități de dezvoltare de aplicații, mult mai sigure, mai rapide și mai încăpătoare în ceea ce privește nevoile tot mai mari de stocare și de procesare.

Pe lângă cloud computing, lucrarea mai dorește să atingă și subiecte precum tehnologia necesară dezvoltării unei interfețe de comunicare moderne cu aceste sisteme distribuite și modelul de comunicare, implementat conform unui standard foarte eficient – REST.

Aplicația vizează mediul business de dezvoltare de software, deoarece are o strânsă legătură cu o metodologie de management al ciclului de viață al aplicațiilor, pe numele ei SCRUM. Ideea de bază a aplicației este ca orice dezvoltator de software al unei companii să poată adăuga note în cadrul proiectelor, note necesare tuturor etapelor de dezvoltare ale aplicației, deoarece conținutul acestora poate fi legat și de arhitectură, și de design, poate fi un defect sau o calitate ce trebuie rezolvat, respectiv implementată.

Integrarea aplicației cu sistemele distribuite oferă acestuia o putere de scalare virtual nelimitată, așadar ar putea fi folosită și de 5-10 echipe în același timp, însă la fel de bine ar putea fi folosită – bineînțeles cu costurile de rigoare – de către 5 mii de echipe simultan. Totul fără a modifica nici o linie din codul sursă al serverului sau al clientului (interfeței web).

Pe scurt, această aplicație poate ajuta companiile care se ocupă cu dezvoltarea de software (sau alte domenii de activitate ce se pot mula pe metoda *Scrum*) indiferent de dimensiunea acestora, oferind o interfață intuitivă, stabilă, versatilă și sigură în ceea ce privește disponibilitatea și siguranța datelor.

CAPITOLUL 1 Product Backlog

1.1 Noțiuni necesare

1.1.1 Colecția de metodologii Agile

Agile software development este un grup de metodologii pentru dezvoltare de software bazate pe dezvoltare iterativă și incrementală, unde cerințele și soluțiile evoluează printr-o colaborare dintre echipe care se organizează singure.

1.1.1.1 Predecesori

Metodele de dezvoltare software *incrementale* datează încă din 1957. În 1974, un articol al lui E. A. Edmonds introducea un proces adaptiv de dezvoltare de software.

Așa-zisele metode „lightweight” de dezvoltare de software au evoluat pe la mijlocul anilor 1990 ca o reacție împotriva metodelor „grele”, care erau caracterizate drept extrem de regularizate, regimentate, micro-organizate, ca un model „waterfall” de dezvoltare.

Implementări timpurii de metode „lightweight” includ *Scrum* (1995), *Crystal Clear*, *eXtreme Programming* (1996), *Adaptive Software Development*, *Feature Driven Development* și *Dynamic Systems Development Method (DSDM)* (1995). Acestea sunt acum referite drept *metodologii agile* după publicarea Manifestului Agile în 2001.

1.1.1.2 Manifest

În februarie 2001, 17 dezvoltatori de software s-au întâlnit în resortul de ski Snowbird, Utah, pentru a discuta metodele „lightweight” de dezvoltare de software. Ei au publicat Manifestul pentru *Agile Software Development* pentru a defini abordarea cunoscută astăzi drept *agile software development*. Unii dintre autorii manifestului au format *Agile Alliance*, o organizație non-profit care promovează dezvoltarea de software în concordanță cu principiile manifestului.

Valorile Agile, scrise în Manifest sunt:

1. Indivizii și interacțiunea înaintea proceselor și uneltelor;
2. Software funcțional înaintea documentației vaste;
3. Colaborarea cu clientul înaintea negocierii contractuale;
4. Receptivitatea la schimbare înaintea urmării unui plan.

Cu alte cuvinte, deși există valoare în elementele din dreapta, le apreciem mai mult pe cele din stânga.

Indivizi și interacțiune – în dezvoltarea agile, organizarea de sine și motivarea sunt foarte importante, la fel și interacțiunile cum ar fi colocația și *pair-programming*-ul.

Software funcțional – software-ul care funcționează va fi mai folositor și mai bine-venit decât simpla prezentare a unor documente clienților la întâlniri.

Colaborare cu clientul – cerințele nu pot fi colectate de-a întregul încă de la începutul ciclului de dezvoltare al proiectului, așadar implicarea continuă a clientului sau a sponsorului este foarte importantă.

Receptivitatea la schimbare – dezvoltarea agile este axată pe răspunsuri rapide la schimbare și dezvoltare continuă.

1.1.1.3 Principiile Agile

- Prioritatea noastră este satisfacția clientului prin livrarea rapidă și continuă de software valoros.
- Schimbarea cerințelor este binevenită chiar și într-o fază avansată a dezvoltării. Procesele agile valorifică schimbarea în avantajul competitiv al clientului.
- Livrarea de software funcțional se face frecvent, de preferință la intervale de timp cât mai mici, de la câteva săptămâni la câteva luni.
- Oamenii de afaceri și dezvoltatorii trebuie să colaboreze zilnic pe parcursul proiectului.
- Construiește proiecte în jurul oamenilor motivați. Oferă-le mediul propice și suportul necesar și ai încredere că obiectivele vor fi atinse.
- Cea mai eficientă metodă de a transmite informații înspre și în interiorul echipei de dezvoltare este comunicarea față în față.
- Software funcțional este principala măsură a progresului.
- Procesele agile promovează dezvoltarea durabilă. Sponsorii, dezvoltatorii și utilizatorii trebuie să poată menține un ritm constant pe termen nedefinit.
- Atenția continuă pentru excelență tehnică și design bun îmbunătățește agilitatea.
- Simplitatea – arta de a maximiza cantitatea de muncă nerealizată – este esențială.
- Cele mai bune arhitecturi, cerințe și design emerg din echipe care se auto-organizează.
- La intervale regulate, echipa reflectă cum să devină mai eficientă, apoi își adaptează și ajustează comportamentul în consecință.

1.1.1.4 Caracteristici

Metodele agile divid cerințele în incrementări mici, ce necesită un minimum de planificare și nu implică direct planificarea pe termen lung. Iterațiile sunt perioade relativ scurte de timp ce durează în mod normal între o săptămână și patru săptămâni. Fiecare iterație implică o echipă ce lucrează la un ciclu complet de dezvoltare de software, care implică planificarea, analiza cerințelor, design, scriere de cod, testare și acceptarea testelor când un produs funcțional este demonstrat clienților. Aceasta minimizează riscul per total și oferă o adaptabilitate sporită a produsului la schimbări rapide. O iterație poate să nu adauge destulă funcționalitate pentru o versiune gata de vânzare, dar țelul este să existe o versiune funcțională (cu minim de defecte) la sfârșitul fiecărei iterații. E posibil să fie nevoie de mai multe iterații pentru a realiza un produs sau funcționalități noi.

Echipa într-un proiect agile este de regulă formată din dezvoltatori ce posedă cunoștințe din mai multe domenii (cross-functional team) care se auto-organizează (self-organizing team) făcând abstracție de forma de ierarhie corporațională sau de rolurile din corporație ale membrilor echipei. Membrii, de regulă, își asumă responsabilitatea pentru cerințele care contruiesc funcționalitatea de care o iterație are nevoie. Decid individual cum să îndeplinească cerințele unei iterații.

Metodele agile accentuează importanța comunicării față în față în contrast cu comunicarea prin documente, atunci când toată echipa este într-un singur loc. Majoritatea echipelor agile lucrează într-un singur birou deschis (denumit *bullpen*), ce facilitează această comunicare. Dimensiunea echipei este mică (5-9 persoane) pentru a simplifica comunicarea și colaborarea dinăuntru său. Este posibil ca mai multe echipe să lucreze îndreptându-se către același țel sau către părți diferite ale aceluiași produs. Acest lucru poate să necesite o coordonare a priorităților care să ia în considerare și celelalte echipe. Când o echipă lucrează în locații diferite, dezvoltatorii mențin contactul zilnic prin videoconferință, voce, e-mail etc.

Indiferent de ce discipline de dezvoltare sunt necesare, fiecare echipă agile va conține reprezentatul clientului. Această persoană este delegată de către clienți să se implice personal în a fi la dispoziția dezvoltatorilor pentru a răspunde la problemele ivite în mijlocul iterației, probleme care ar fi legate direct de client. În unele cazuri nu este nevoie ca reprezentatul clientului să fie o persoană din echipă, ci poate fi chiar unul din clienți.

Majoritatea implementărilor agile folosesc o rutină și o comunicare formală zilnică față în față între membrii echipei. Aceasta include specific reprezentantul clientului și orice client sau sponsor interesat ca și observatori. Într-o sesiune scurtă, membrii echipei raportează unul altuia ce au făcut în ziua precedentă, ce intenționează să facă în ziua curentă și care sunt lucrurile

care-i opresc. Această comunicare față în față expune problemele imediat ce acestea apar.

Dezvoltarea agile pune accentul pe software funcțional ca principală măsură a progresului. Aceasta, combinată cu preferința pentru comunicare față-n-față, produce mai puțină documentație scrisă decât alte metode. Metoda agile incurajează părțile interesate să prioritizeze dorințele cu alte posibile rezultate ale iterațiilor, bazate exclusiv pe valoarea de business oferită la începutul iterației (se mai numește *value-driven*).

Utilitare specifice și tehnici cum ar fi integrarea continuă, testare automată sau xUnit, programarea în pereche, dezvoltarea prin testare deasă și timpurie, *design patterns*, *domain-driven design*, *code refactoring* și alte tehnici sunt des folosite pentru a îmbunătăți calitatea și pentru a spori agilitatea proiectului.

1.1.2 O metodologie particulară – Scrum

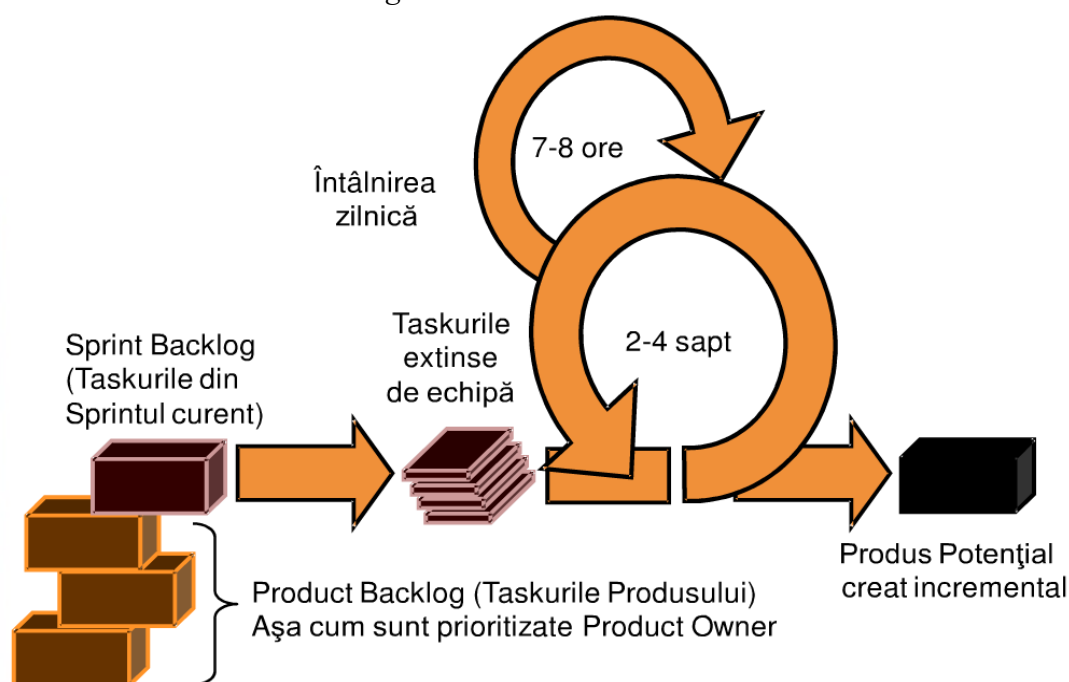
1.1.2.1 Procesul Scrum

Scrum este o metodă iterativă și incrementială al cărei scop este acela de a ajuta echipele de dezvoltare să își concentreze atenția asupra obiectivelor stabilite și minimizarea muncii depuse de aceștia pentru rezolvarea sarcinilor mai puțin importante. Scrum dorește să pastreze simplitate într-un mediu de afaceri complicat. Termenul provine din rugby unde reprezintă o strategie de a readuce o minge “pierdută” înapoi în joc folosind munca de echipă. Scrum nu oferă tehnici la nivel de implementare, ci se axează pe modul în care membrii unei echipe de dezvoltare ar trebui să interacționeze pentru a produce un sistem flexibil, adaptabil și productiv într-un mediu ce este permanent în schimbare. Cei care au prezentat în detaliu această metodă sunt Schwaber și Beedle. Scrum se bazează pe două elemente: **autonomia echipei** și **adaptabilitate**. Autonomia echipei se referă la faptul că cei care conduc proiectul stabilesc sarcinile care trebuie rezolvate de către echipă, însă aceasta are libertatea de a-și stabili propriul mod de lucru în cadrul fiecărei iterații, scopul fiind acela de a spori productivitatea echipei.

1.1.2.2 Caracteristici

Scrum este un proces iterativ și incremental de dezvoltare software ce reprezintă planul unui proiect care include un set de activități și roluri predefinite. Principalele roluri sunt cele de *Conducător Scrum* care întreține procesele și se comportă ca un *project manager*, *Deținător de produs* care reprezintă vocea clientului și *Echipa* care include dezvoltatorii de software. Produsul software evoluează de-a lungul mai multor etape numite *sprint*-uri. În timpul unui sprint Echipa creează o unitate de software funcțională. Elementele de care trebuie să se țină cont într-un sprint se preiau dintr-o mulțime de sarcini nerezolvate.

Figura 1: Procesul Scrum



Sarcinile care intră în sprint sunt stabilite în urma întâlnirii de planificare sprint. Pe parcursul acestei întâlniri Deținătorul de proiect informează Echipa cu privire la sarcinile nerezolvate pe care dorește să le abordeze. Echipa stabilește câte astfel de sarcini poate îndeplini până la următorul sprint. În timpul unui sprint nu se pot schimba sarcinile alese. La sfârșit Echipa demonstrează cum se utilizează produsul intermediar obținut.

Scrum încurajează crearea echipelor cu auto-organizare și comunicarea verbală între toți membrii echipei și între diferitele departamente care au legătură cu proiectul. Un principiu important în această abordare o reprezintă acceptarea faptului că, pe parcursul dezvoltării unui proiect, clientul se va răzgândi de multe ori cu privire la ce dorește și are nevoie să ofere produsul software. Astfel de schimbări neprevizibile nu sunt ușor de adaptat la proiect folosind metodele tradiționale de dezvoltare software. Scrum adoptă o abordare empirică susținând că o problemă nu poate fi pe deplin înțeleasă sau definită și punând accent pe dezvoltarea abilității unei echipe de a rezolva rapid noi cerințe.

1.1.2.3 Roluri în cadrul procesului Scrum

În cadrul unui proces Scrum sunt definite 6 roluri (conform lui Schwaber și Beedle), fiecare cu diferite sarcini și scopuri. Aceste roluri sunt împărțite în două categorii, **Porci** și **Pui**, după cum urmează:

- **Porcii** – cei direct implicați în procesul de dezvoltare, angajați să construiască proiectul și care sunt trași la răspundere.
 - **Conducătorul Scrum** – are un rol de project manager (dar el nu este șeful echipei) ce trebuie să se asigure că procesul de dezvoltare evoluează în conformitate cu tehnicile, valorile și regulile Scrum. Acesta interacționează atât cu Echipa de dezvoltare, cât și cu clienții și conducerea organizației. Este de asemenea responsabil să se asigure că orice impediment și orice element care distrage atenția echipei sunt înlăturate, astfel încât productivitatea echipei să fie permanent la un nivel ridicat.
 - **Deținătorul de produs** – reprezintă vocea, interesele clientului. El este responsabil de proiectarea, administrarea, controlul și prezentarea produsului nerezolvat; ia decizia finală cu privire la sarcinile din produsului nerezolvat și le asociază priorități. Este ales de către Conducătorul Scrum, client și conducere.
 - **Echipa** – este responsabilă cu dezvoltarea produsului; are autoritatea de a decide ce măsuri trebuie luate pentru a rezolva sarcina asociată fiecărui sprint și are dreptul de a se auto-organiza tot în același scop. În general o echipă Scrum este alcătuită din 5-9 persoane.
- **Puii** - cei care nu sunt implicați direct în dezvoltarea proiectului, dar de a căror părere trebuie să se țină cont. În abordarea agilă un aspect foarte important îl reprezintă implicarea utilizatorilor, clienților, oamenilor de afaceri în procesul de dezvoltare. Aceștia trebuie să ofere feed-back cu privire la rezultatele fiecărui sprint pentru a adapta și îmbunătăți viitoarele procese de lucru.
 - **Utilizatorii** – cei care vor folosi produsul software
 - **Clienții** – cei care stabilesc scopul proiectului; sunt implicați în procesul de dezvoltare doar când are loc evaluarea unui sprint
 - **Managerii** – cei responsabili de luarea deciziilor finale. Participă de asemenea la stabilirea obiectivelor și a condițiilor de lucru.

1.2 Descriere backlog

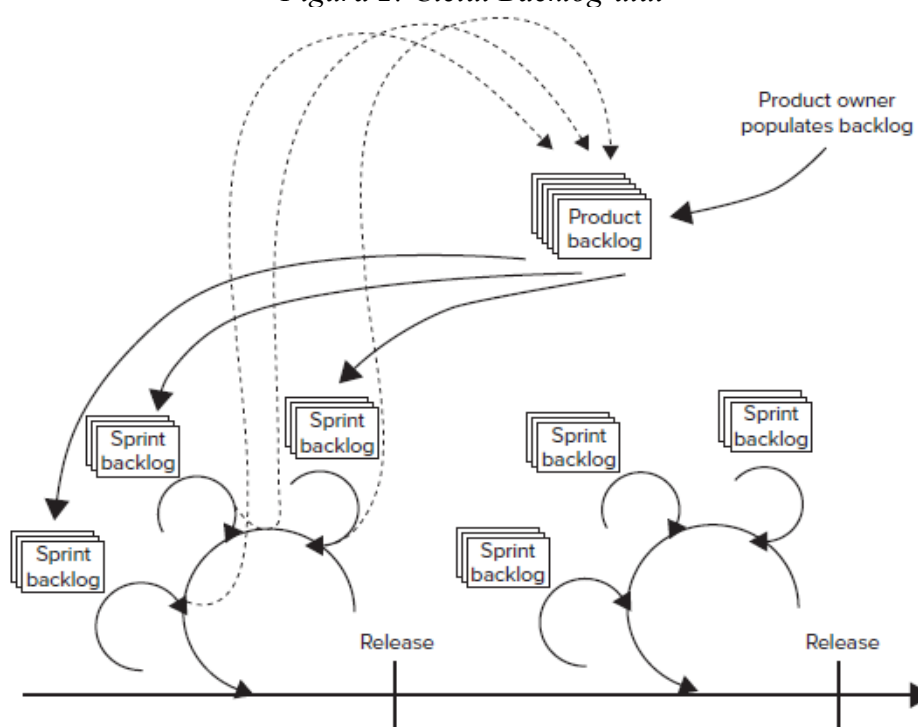
Product Backlog (Backlog-ul produsului) este lista de funcționalități ale unui produs, compilate de către deținătorul produsului, care crede că pe baza acestora se va construi un produs foarte bun.

Înainte de începerea proiectului și înainte ca echipa de programatori să fie creată,

deținătorul produsului se întâlnește cu clientul pentru o înțelegere în profunzime a nevoilor acestuia din urmă. După ce produsul este într-o formă prezentabilă și este arătat clienților, deținătorul produsului va afla mai mult despre ce le place, ce nu le place și ce au tolerat.

Ciclul Backlog-ului (Figura 2) unui produs arată fluxul de funcționalități într-un produs. Începe cu managerul produsului ce traduce nevoile și dorințele clientului într-o listă de funcționalități de produs ce pot fi construite de echipă. Acesta este Backlog-ul produsului. O echipă *Scrum* este organizată să creeze eficient un produs în *sprint*-uri scurte de activitate. Deținătorul proiectului prioritizează Backlog-ul produsului și identifică o submulțime de funcționalități legate între ele ce pot fi implementate într-un *sprint*.

Figura 2: Ciclul Backlog-ului



Backlog-ul produsului este, ca și produsul în sine, construit iterativ. Acesta nu este niciodată static și niciodată complet. Are în componența sa un set de cerințe înainte de primul *sprint* și apoi un set foarte diferit de cerințe cu fiecare *sprint* care urmează. Crește o dată ce clienții văd produsul și dau sugestii. Crește de asemenea și o dată ce îl văd sponsorii proiectului. Se micșorează când echipa contruiește din funcționalitățile cerute – mutând cerințele din Backlog-ul produsului în Backlog-ul Sprintului și marcându-le ca și complete.

Backlog-ul produsului este singura coadă de intrare pentru echipă. Dacă o funcționalitate nu se află în Backlog-ul produsului nu va fi programată și inclusă într-un *sprint* deci nu va fi construită. Din această cauză defectele care există la sfârșitul unui *sprint* sunt introduse în Backlog-ul produsului pentru a fi rezolvate în următoarele *sprint*-uri. Deținătorul produsului,

care stabilește prioritățile pentru toate cerințele din Backlog-ul produsului, determină dacă defectele ar trebui puse deasupra funcționalităților cerute de către clienți sau sub ele. În cele mai multe cazuri, unele defecte se situează deasupra funcționalităților cerute de clienți, în timp ce altele se situează mai jos.

Fiecare *sprint* începe cu o listă de funcționalități ce trebuie implementate. Fiecare *sprint* se termină cu majoritatea, dar nu toate, funcționalităților terminate. Cerințele neterminate se re-adaugă în Backlog-ul produsului pentru includerea lor în *sprint*-uri viitoare. După câteva *sprint*-uri – de regulă între 3 și 10 – echipa este gata să livreze o versiune de produs. Se livrează bucățile funcționale, se distribuie către clienți, se preia feedback-ul și apoi se începe construirea versiunii următoare.

Backlog-ul produsului este un utilitar eficient pentru management-ul creșterii continue de funcționalități în *sprint*-urile iterative și versiunilor de Scrum. O echipa poate adăuga funcționalități unui produs fără o limită anume. Este doar o problemă de prioritzare și de resurse. Dacă un client sau sponsor spune deținătorului produsului „trebuie să avem funcționalitatea X”, răspunsul poate fi la obiect și pozitiv ca „Da, sună foarte bine! Îl vom introduce în Backlog-ul produsului și îl vom programa pentru un *sprint* viitor.”

CAPITOLUL 2 Dezvoltarea în cloud

2.1 Introducere

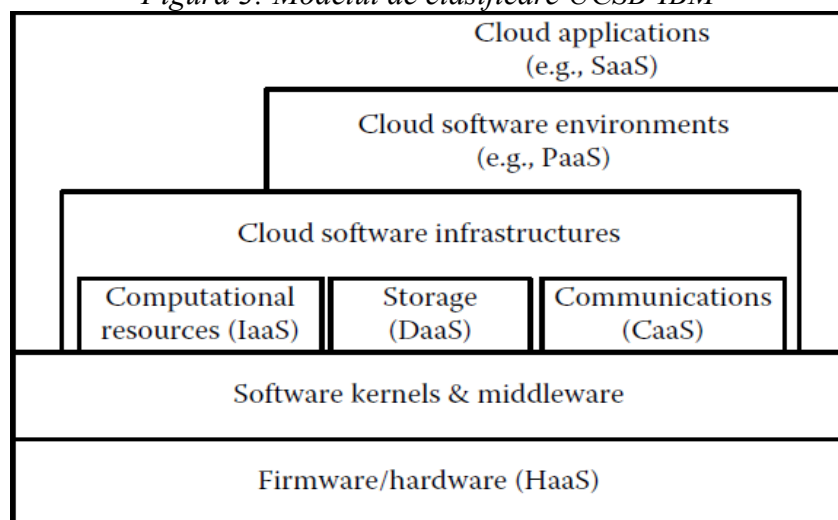
Cloud computing (aprox. „calcul în Internet”) este un concept modern în domeniul computerelor și informaticii, reprezentând un ansamblu distribuit de servicii de calcul, aplicații, acces la informații și stocare de date, fără ca utilizatorul să aibă nevoie să cunoască amplasarea și configurația fizică a sistemelor care furnizează aceste servicii. Pentru *cloud computing* încă nu există un nume românesc încetățenit.

Există trei modele (SaaS, PaaS, IaaS) de clasificare ale *cloud*-ului ce prezintă nivele de detalii diferite ale posibilității de exploatare a acestuia, deoarece au apărut la momente diferite ale evoluției acestui domeniu. Deși au obiective diferite – unele sunt folosite în scopuri academice pentru cercetarea noului, în timp ce altele ținesc la identificarea și analiza oportunităților comerciale și de marketing – ele accelerează înțelegerea unora dintre legăturile făcute între sistemele de *cloud computing*.

2.2 Cloud computing

2.2.1 Clasificarea UCSB-IBM

Figura 3: Modelul de clasificare UCSB-IBM



Clasificarea UCSB-IBM a fost realizată prin efortul coroborat dintre academie (Universitatea din California, Santa Barbara) și industrie (IBM T.J. Watson Research Center) într-o încercare de a înțelege imaginea *cloud computing*-ului. Țelul acestui efort era de a facilita explorarea ariei *cloud computing* și de asemenea de a promova eforturile educaționale în predarea și adoptarea ariei *cloud computing*.

În clasificarea USCB-IBM (Figura 3), autorii au folosit principiul de compunere de la *Service-Oriented Architecture* (SOA) pentru a clasifica diferitele straturi ale *cloud*-ului. Compunerea în SOA este modul de a coordona și asambla o colecție de servicii pentru a forma servicii compozite. În acest fel, serviciile *cloud* pot fi de asemenea compuse din unul sau mai multe diferite servicii *cloud*.

După principiul compunerii, modelul UCSB-IBM a clasificat *cloud*-ul în cinci straturi. Fiecare strat încapsulează unul sau mai multe servicii *cloud*. Serviciile *cloud* aparțin aceluiași strat dacă au un nivel echivalent de abstractizare, evidențiat de către utilizatorii vizați de către acestea. De exemplu, toate mediile software de *cloud* (cunoscute și ca *platforme cloud*) vizează programatorii, în timp ce aplicațiile *cloud* vizează utilizatorii finali. Așadar, mediile software *cloud* vor fi clasificate într-un alt strat decât aplicațiile *cloud*.

2.2.2 Software as a Service (SaaS)

Primul strat este Aplicația (SaaS) și vizează cel mai mult utilizatorii finali. În mod normal, utilizatorii accesează serviciile oferite de acest strat printr-un *browser* via portaluri web și sunt uneori necesare plățile unor taxe pentru utilizarea lor. Acest model s-a dovedit a fi foarte atrăgător pentru mulți utilizatori, deoarece ușurează răspunderea mentenanței software-ului și de asemenea reduce costurile pentru suport tehnic. Mai mult, aceasta mută sarcina de procesare de pe terminalele utilizatorilor pe data center-ele unde sunt lansate aplicațiile. Aceasta, la rândul său, micșorează necesitatea puterii de procesare pe terminalele utilizatorilor și le oferă ocazia de a obține performanțe extraordinare pentru unele din procesele lor ce necesită foarte multă putere de computație și/sau memorie, fără necesitatea unei investiții mari în mașinile lor locale.

În ceea ce privește furnizorii de aplicații *cloud*, acest model le simplifică munca depusă la modificarea și testarea codului-sursă, în timp ce protejează proprietatea lor intelectuală. Deoarece o aplicație *cloud* se desfășoară pe infrastructura furnizorului (în loc să fie executată pe mașinile desktop ale utilizatorilor), dezvoltatorii aplicației au posibilitatea de a aplica modificări minore sistemului și de a adăuga funcționalități fără a deranja utilizatorii cu cereri de instalări și de actualizări. Configurația și testarea aplicației în acest model este fără îndoială mai puțin complicată deoarece mediul de execuție devine restricționat. Considerând profitul furnizorului, acest model oferă dezvoltatorului un flux continuu de câștig, care poate fi chiar și mai profitabil pe termen lung. Acest model SaaS conferă multe beneficii pentru utilizatorii și furnizorii de aplicații *cloud*.

2.2.3 Platform as a Service (PaaS)

Al doilea strat în clasificarea UCSB-IBM este *Platform as a Service* (PaaS). Utilizatorii acestui strat sunt dezvoltatorii de aplicații *cloud*, implementând aplicațiile lor și lansându-le pe *cloud*. Furnizorii de PaaS oferă dezvoltatorilor un mediu de programare la nivel de limbaj ce constă în API-uri bine definite (Application Programming Interface), pentru a facilita interacțiunea dintre mediile de programare și aplicațiile *cloud* și de asemenea pentru a accelera lansarea și a suporta scalabilitatea necesară de aplicațiile *cloud*.

Dezvoltatorii de bucură de multiple beneficii atunci când dezvoltă aplicațiile lor pentru cloud pentru un anumit mediu de programare cloud, incluzând scalabilitate automată și *load-balancing*, de asemenea integrare cu alte servicii (ex. servicii de autentificare, servicii de e-mail și de interfață cu utilizatorul) oferite lor de către furnizorul de PaaS. În acest fel, mare parte din timpul petrecut în plus dezvoltând aplicații cloud este ameliorat și preluat direct la nivel de mediu de dezvoltare. Mai mult, dezvoltatorii au posibilitatea de a integra alte servicii cu aplicațiile lor la cerere. Acest lucru face dezvoltarea aplicațiilor în cloud o sarcină mai puțin complicată, accelerează timpul de lansare și minimizează erorile logice din aplicație. În acest sens, o lansare Hadoop în cloud este considerată PaaS, deoarece oferă dezvoltatorilor de aplicații un mediu de dezvoltare, pe numele său framework-ul *Map Reduce* pentru cloud.

Dezvoltarea unei aplicații pentru o platformă cloud este analogul într-o anumită măsură la dezvoltarea unei aplicații web pentru modelul vechi de web-servere, în sensul că dezvoltatorii scriu codul-sursă și îl lansează pe un server extern. Pentru utilizatorii aplicației, rezultatul final este de regulă o aplicație browser-based (o aplicație ce rezidă într-un browser). Cu toate acestea, modelul PaaS este diferit în sensul că poate oferi servicii adiționale pentru a simplifica dezvoltarea aplicației, lansării și execuției, ca de exemplu scalabilitatea automată, monitorizarea și *load-balancing*. Prin urmare, clasa PaaS este în general recunoscută pentru accelerarea dezvoltării de software și a timpului de lansare. La rândul său, software-ul construit pentru platforma cloud are în mod normal un timp mai mic până la vânzare (time-to-market).

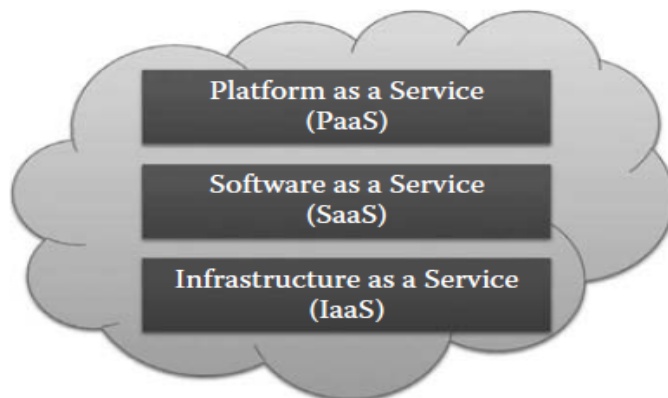
2.3 Platforme și servicii existente

Recent, cloud computing a devenit o nouă alternativă pentru cercetători la achiziționarea de putere de calcul și capacitatea de stocare de care au nevoie. Nu există încă o definiție larg acceptată pentru cloud computing, însă mai multe puncte de vedere sunt comune: (1) este scalabil și elastic, un utilizator poate avea oricât de mult sau de puțin serviciu dorește în orice moment de timp și (2) are un model de taxare inclus, lasă utilizatorii să plătească în funcție de câte resurse închiriază și consumă. Cloud-ul folosește virtualizarea, ceea ce omogenizează diferențele dintre software-ul și hardware-ul de bază. Ca rezultat, cloud-ul prezintă un mediu configurabil în termeni de sistem de operare și de software ce vine inclus cu acesta, fundația sa fiind mașina virtuală.

Stratul PaaS evidențiază posibilitatea de a construi o platformă de calcul cu capacitate de stocare și aplicații ca servicii. Acesta include utilitare și API-uri pentru a contrui SaaS; exemple de PaaS sunt Google App Engine, Salesforce Apex Language și altele. Stratul SaaS se referă la orice tip de aplicație disponibilă pentru utilizator și lansată pe cloud ca și serviciu reutilizabil de către orice utilizator al cloud-ului. Google Apps și Salesforce CRMs sunt exemple de astfel de servicii. În cazul calculului științific, ar fi un apel la o aplicație științifică sau stiva de programe necesare lansării aplicației. Stratul IaaS desemnează posibilitatea construirii unei infrastructuri complete cu un server de calcul și spațiu de stocare. GoGrid sau Amazon cu Amazon Elastic Compute Cloud (EC2) și Amazon Simple Storage Service (S3) sunt implementări de aceste servicii și oferă resurse de calcul, respectiv spațiu de stocare.

Pe lângă soluții cloud oferite de companii private, există și utilitare cum ar fi Eucalyptus, OpenNebula și Nimbus care mimează într-o anumită măsură serviciile comerciale, utilizând grupuri de servere pentru a crea medii cloud de testare.

Figura 4: Principalele categorii ale cloud computing-ului



2.4 Avantaje și dezavantaje

Principalele avantaje ale cloud computing-ului:

- Prin folosirea virtualizării, calculul în cloud deschide infrastructura către un număr mare de aplicații. Într-adevăr, mașinile virtuale și sistemele de operare specializate oferă un mediu ideal pentru executarea aplicațiilor vechi. Aceste aplicații sunt de cele mai multe ori foarte sensibile în ceea ce privește mediul de execuție și nimeni nu dorește modificarea acelor coduri-sursă care funcționează și au fost validate doar pentru a le adapta pentru executarea pe o anumită platformă.
- Virtualizarea folosită de cloud oferă de asemenea un *mediu configurabil și reproductibil* pentru a ținti o aplicație anume, așadar utilizatorul poate folosi imediat aplicația și/sau o poate refolosi la o dată viitoare. Acesta poate fi un element important în favoarea calculului în cloud pentru un utilizator ce dorește să poată avea posibilitatea de a reproduce analiza de-a lungul timpului sau care se confruntă cu aplicații vechi, neîntreținute, care sunt greu de portat pe medii noi.
- Cloud-ul promite scalabilitate și alocare dinamică de resurse pentru a satisface nevoile utilizatorului. Această lucră, însă, nu este foarte clar cât de bine se realizează în practică atunci când tehnologiile trebuie împărțite și balansate către un număr foarte mare de utilizatori.

Există, desigur și dezavantaje:

- Deși producătorii de hardware și de sisteme de operare au făcut un efort imens de a îmbunătăți virtualizarea, performanța încă depinde de modul în care hardware-ul din spate, rețeaua și mașina virtuală au fost configurate. Deși performanța procesorului virtual este apropiată de performanța procesorului fizic, performanța rețelei virtuale încă rămâne în urma capacităților de performanță ale interconectării fizice.
- Există mai mulți actori și furnizori pe piața cloud-ului, însă nu există un *standard adoptat* și, chiar mai rău, unele tehnologii se bazează pe interfețe proprietare. Ca și consecință, nu există compatibilitate între furnizori diferiți. O dată ce un utilizator dezvoltă o aplicație pentru un anumit cloud, e posibil să fie nevoie de ceva efort pentru a porta acea aplicație pe un cloud diferit.
- Modelul de taxare dictează nevoia de a evalua și cuantifica nevoile computaționale, de stocare și de conectare la rețea ale unei aplicații. Aceste valori sunt adesea foarte greu de previzionat și pot rezulta costuri neanticipate.

2.5 Securitatea cloud-ului

Deși există un beneficiu clar în folosirea cloud computing, grija securității a făcut ca organizațiile să evite tranziția resurselor importante în cloud. Corporații și indivizi sunt deseori îngrijorați despre cum securitatea și integritatea pot fi menținute în acest mediu nou. Încă și mai îngrijorătoare, însă, sunt acele corporații care fac tranziția în cloud fără să își pună aceste probleme de securitate.

Tranziția aplicațiilor importante și a datelor sensibile în medii cloud publice și partajate este o grijă foarte mare pentru acele corporații care scot datele în afara securității rețelei data-center-ului lor. Pentru a micșora aceste griji, un furnizor de servicii cloud trebuie să asigure clienților aceeași securitate și confidențialitate a datelor ca și înainte de mutarea în cloud, să le aducă dovezi clienților cum că datele lor sunt securizate, că pot îndeplini SLA-ul (Service-Level Agreement) și de asemenea că se conformează auditurilor.

Cu modelul de cloud, se pierde controlul asupra securității fizice. Într-un cloud public, se împart resursele de calcul cu alte companii. Într-o locație partajată din afara întreprinderii, nu se poate avea nicio idee sau control despre unde rulează resursele. Expunând datele întreprinderii într-un mediu partajat cu alte companii ar putea da guvernării „cauză rezonabilă” pentru a confisca bunurile întreprinderii deoarece o altă companie ce activează în același mediu partajat a încălcat legea. Așadar, doar din cauză că se partajează același mediu cloud, poate pune datele la risc de confiscare.

Folosirea software-ului oferit de SaaS în cloud duce la o necesitate mult mai redusă de dezvoltare de software. De exemplu, folosind un soft CRM (customer relationship management) bazat pe web oferit de SaaS elimină necesitatea scrierii de cod și „personalizării” aplicației unui vânzător. Dacă se plănuiește folosirea în cloud a codului dezvoltat inter, este încă și mai important să se implementeze o procedură formală securizată de dezvoltare a ciclului de viață al aplicației (SDLC). Utilizarea imatură a mixajului de tehnologie (combinarea web-service-urilor), care este fundamentală în aplicațiile cloud, va duce inevitabil la consecințe grave de securitate și vulnerabilități informatice ale acelor aplicații. Instrumentul de dezvoltare ales ar trebui să ofere un model de securitate implementat înăuntrul său pentru a da indicații dezvoltatorilor de software în timpul perioadei de dezvoltare și să restricționeze utilizatorii doar la datele pe care le dețin atunci când sistemul este lansat în producție.

Outsourcing-ul înseamnă pierderea unei părți foarte mari din controlul asupra datelor și, deși acest lucru nu este o idee bună din perspectiva securității, ușurința afacerii și economiile financiare vor continua să crească folosința acestor servicii. Managerii de securitate vor trebui să lucreze cu departamentul juridic al companiei pentru a se asigura că există contracte solide care

să protejeze datele corporației și să ofere SLA-uri acceptabile. Serviciile cloud se vor vedea în foarte mulți utilizatori IT de telefoane mobile care accesează datele și serviciile companiei fără a fi conectați direct la rețeaua corporației. Acest lucru va spori nevoia întreprinderilor de a implementa controale de securitate între utilizatorii de dispozitive mobile și serviciile bazate pe cloud. Plasarea unor cantități mari de date sensibile într-un cloud accesibil global lasă organizația pasibilă la atacuri distribuite largi – atacatorii nu mai trebuie să ajungă în perimetrul corporației pentru a putea fura date, ci pot găsi toate datele într-o singură locație „virtuală”.

Virtualizarea este folosită în data center-e pentru a facilita economisirea costurilor și pentru a crea o amprentă ecologică mai mică, mai „verde”. Ca rezultat, se crează mai multe servere virtuale pe un singur server fizic, iar acestea servesc mai multor utilizatori, spre deosebire de ideologia veche în care un singur server fizic putea servi un singur utilizator. Extinderea virtualizării și a mașinilor virtuale în cloud afectează securitatea corporațiilor ca rezultat al eliminării perimetrului fizic de securitate al rețelei corporației.

Deoarece multe conexiuni între companii și furnizorii lor de SaaS se produc prin web, furnizorii ar trebui să securizeze aplicațiile lor orientându-se după sfaturile Open Web Application Security Project (OWASP) pentru dezvoltarea securizată de aplicații și să blocheze porturile și comenzile nefolosite în stivele Linux, Apache, MySQL și PHP (LAMP) din cloud, exact cum ar face în mod normal într-un datacenter sau pe un server local. LAMP este o platformă open-source de dezvoltare web, numită și *stivă web*, care folosește Linux ca sistem de operare, Apache ca server web, MySQL ca sistem relational de baze de date (RDBMS) și PHP ca limbaj de programare scriptat orientat pe obiect. De multe ori se substituie limbajul PHP cu Perl sau Python.

O abordare interesantă legată de cloud, în favoarea securității a venit din partea Google, cu noile *Chromebooks*. Acestea sunt netbook-uri cu o capacitate foarte mică de stocare, însă care au inclus un modem 3G pentru conectare permanentă la internet, iar singura aplicație instalată – pe sistemul de operare proprietar Google – este un browser web, extrem de bine securizat, Google Chrome. Acest nou concept elimina în schimb datele care ar fi putut fi preluate fizic de pe calculator, deoarece nu se stochează nimic local. Securizând toate transferurile de date dinspre și către cloud, acest model de securitate funcționează foarte bine – practic toate fișierele se stochează în cloud, unde este oferită o redundanță sporită, așadar datele nu sunt furate niciodată chiar în caz de furt al netbook-ului, presupunând bineînțeles că s-au folosit doar servicii cloud care au oferit o securitate bună.

CAPITOLUL 3 Infrastructura Google pentru SaaS

3.1 Prezentarea serviciului GAE

Google App Engine este un serviciu cloud ce oferă posibilitatea executării aplicațiilor dezvoltatorilor de software pe infrastructura Google. Aplicațiile App Engine sunt relativ ușor de construit și de menținut și se scalează automat atunci când cerințele de trafic, stocare și calcul cresc. Pentru servirea aplicației se poate folosi propriul domeniu (ex. <http://domeniu.ro>) sau un subdomeniu oferit gratuit pe domeniu appspot.com. Există de asemenea și un model de securitate pentru a stabili ușor dacă se partajează informația cu întreaga lume, cu utilizatorii unei anumite corporații sau pur și simplu cu un set restrâns de utilizatori (folosind OpenID).

Google App Engine suportă mai multe limbaje de programare, printre care Java, Python și Go (limbajul de programare dezvoltat de Google). Fiecare din aceste medii de programare rulează rapid, securizat și fără interferențe de la alte aplicații din sistem.

Costul App Engine este mulat în mare parte pe costul standard cloud „pay-as-you-go”, adică nu există un abonament lunar sau obligații contractuale, ci pur și simplu se plătește în funcție de resursele cerute și consumate. Se pot seta limite superioare ale aplicației, pentru a nu se ajunge la costuri neprevăzute. Pentru aplicațiile mici însă, GAE poate fi complet gratuit deoarece fiecare aplicație primește gratuit 500MB de spațiu de stocare, 6.5 ore de utilizare procesor / zi și 1GB de transfer de date securizat / zi (prin https, către și dinspre cloud).

3.1.1 Mediul aplicației

Cu acest serviciu este foarte ușor de creat aplicații care să scaleze incredibil de bine la o încărcare extrem de mare și care folosesc cantități foarte mari de date. Există câteva funcționalități cheie în ceea ce privește mediul aplicației:

- servire web dinamică, cu suport pentru tehnologiile web răspândite
- stocare persistentă cu interogări, sortare și tranzacții
- scalare și balansarea încărcării automate
- API-uri pentru autentificarea utilizatorilor și trimiterea de emailuri folosind conturi Google
- un mediu de dezvoltare local cu toate funcționalitățile mediului din cloud care simulează GAE pe computerul utilizatorului
- cozi de sarcini pentru a putea lucra și în afara ariei solicitărilor web
- sarcini programate pentru a declanșa evenimente la anumite intervale de timp sau la date predefinite

Aplicația se poate executa în trei medii de rulare: mediul Go, mediul Java și mediul Python. Fiecare mediu de programare oferă protocoale standard și tehnologii obișnuite pentru dezvoltarea de aplicații web.

3.1.2 Separarea aplicațiilor și limite

Aplicațiile sunt executate într-un mediu securizat ce oferă acces limitat la sistemul de operare sub care rulează. Aceste limitări dau voie GAE să distribuie cereri web pentru aplicație de pe mai multe servere și să pornească, respectiv, oprească servere pentru a îndeplini cererile de trafic. Separarea aplicațiilor izolează aplicația în propriul ei mediu securizat și stabil, mediu independent de hardware, sistemul de operare și locația fizică a serverului web.

Exemple ale acestor limitări pentru separare includ:

- O aplicație poate accesa alte computere din Internet prin serviciile URL fetch și email. Alte computere se pot conecta la aplicație doar realizând cereri HTTP sau HTTPS prin porturile standard 80 și 443.
- O aplicație nu are drept de scriere pe sistemul de fișiere. O aplicație poate citi fișiere, dar doar pe cele urcate cu codul aplicației. Aplicația trebuie să folosească *App Engine datastore* (serviciul de baze de date NoSQL), *memcache* sau alte servicii pentru care datele persistă între cereri.
- Codul aplicației rulează doar ca răspuns a unei cereri web, unei sarcini dintr-o coadă sau a unei sarcini programate și trebuie să returneze un răspuns în maxim 30 secunde în oricare din cazuri. Un gestionar de cerere nu poate da naștere unui sub-process sau să execute cod după ce răspunsul a fost trimis.

Aplicația este limitată și de următoarele date:

Tabel 1: Limitele aplicației

Limita	Cantitatea
dimensiunea cererii	10 MB
dimensiunea răspunsului	10 MB
durata cererii	30 secunde
număr total de fișiere (ale aplicației + fișiere statice)	3,000
dimensiunea maximă a unui fișier de aplicație	10 MB
dimensiunea maximă a unui fișier static	10 MB
dimensiunea totală a tuturor fișierelor aplicației	150 MB

3.2 Python în Google App Engine

3.2.1 Selectarea mediului de programare Python

App Engine știe să folosească mediul de programare Python pentru codul aplicației atunci când se folosește utilitarul *appcfg.py* din SDK-ul (Software Development Kit) Python cu un fișier de configurație *app.yaml*. Se selectează mediul de programare Python cu următoarele elemente de configurare:

```
runtime: python
api_version: 1
```

Primul element, *runtime*, selectează mediul Python. Acesta putea fi, la momentul scrierii acestui document: python, java și go.

Al doilea element, *api_version*, selectează ce versiune de mediu de rulare Python să fie folosit. La momentul scrierii acestui document, App Engine avea doar o versiune de mediu de rulare Python, 1. Dacă echipa App Engine team are nevoie să producă schimbări ale mediului de rulare care ar putea să nu fie compatibile cu mediul vechi, o vor face folosind un nou identificator de versiune a mediului de dezvoltare. Aplicația va continua să ruleze pe versiunea selectată până când se va schimba elementul *api_version* și se va reurca aplicația pe cloud.

3.2.2 Cereri și domenii

App Engine determină dacă o cerere este intenționată pentru o anumită aplicație folosind numele domeniului folosit. O cerere a carui nume de domeniu este *aplicatie-id.appspot.com* este direcționată către aplicația al cărei identificator (id) este *aplicatie-id*. Fiecare aplicație primește un identificator sub *appspot.com* gratuit.

appspot.com suportă și sub-domenii de forma *subdomeniu.aplicatie-id.appspot.com*, unde *subdomeniu* poate fi orice șir de caractere valid într-un nume de domeniu. Cererile trimise către orice subdomeniu sunt trimise aplicației din care fac parte, în acest caz *aplicatie-id*.

Se poate configura și un domeniu top-level particular folosind Google Apps. Cu Google Apps, se alocă subdomenii din numele de domeniu al afacerii cu care este configurat către varii aplicații, cum ar fi Google Mail sau Sites. Se poate, de asemenea, asocia o aplicație App Engine cu un subdomeniu. Pentru utilitate, se poate configura un domeniu Google Apps la înregistrarea unui identificator de aplicație sau mai târziu din Administrator Console (vezi Figura 5).

Figura 5: Adăugarea unui nume de domeniu particular

Domain Setup

Want to host your application on another domain? Google App Engine uses [Google Apps](#) to manage domains. [Learn more](#)

Add Domain...

Cererile pentru aceste URL-uri sunt direcționate către versiunea aplicației selectată din Administrator Console. Fiecare versiune a aplicației are de asemenea propriul ei URL, așadar se poate lansa o nouă versiune de testare înainte de a o configura drept versiunea implicită. URL-ul cu specificul versiunii folosește identificatorul (id) din fișierul de configurare în felul următor: *versiune-id.latest.aplicatie-id.appspot.com*. Analog se pot accesa și subdomeniile dorite prin: *subdomeniu.versiune-id.latest.aplicatie-id.appspot.com*.

Numele de domeniu folosit în cerere este inclus în datele despre cerere ce sunt trimise la aplicație. Dacă se dorește un răspuns diferit în funcție de numele de domeniu folosit pentru a o accesa, se poate face o verificare în datele cererii (de exemplu antetul *Host*) pentru a afla pe ce nume de domeniu a fost făcută cererea și a răspunde în concordanță cu acesta.

3.2.3 Python pur și simplu

Mediul de rulare Python folosește Python versiunea 2.5.2.

Tot codul pentru mediul de rulare Python trebuie să fie strict Python și să nu includă nicio extensie de C sau alt cod ce trebuie compilat.

Mediul include biblioteca standard Python. Unele module au fost dezactivate deoarece funcțiile lor de bază nu sunt suportate de GAE, cum ar fi rețelistica sau scrierea pe sistemul de fișiere. În plus, modulul *os* este disponibil, însă cu funcționalitățile nesuportate dezactivate. Încercarea de a importa un modul indisponibil sau de a folosi o funcționalitate nesuportată va arunca o excepție.

Câteva module din biblioteca standard au fost înlocuite sau personalizate pentru a putea rula pe App Engine. De exemplu:

- *cPickle* este mimat de *pickle*. Funcționalitățile specifice *cPickle* nu sunt suportate.
- *marshal* este gol. Importul va reuși, dar utilizarea nu.
- modulele următoare sunt, în mod similar, goale: *imp*, *ftplib*, *select*, *socket*
- *tempfile* este dezactivat, exceptând *TemporaryFile* care este mimat de *StringIO*.
- *logging* este disponibil și folosirea lui este foarte mult încurajată!

În plus față de biblioteca standard Python și de bibliotecile App Engine, mediul mai include și următoarele biblioteci ale unor terțe părți:

- Django 0.96.1
- WebOb 0.9
- PyYAML 3.05

3.3 Modelarea datelor în Datastore sub Python

3.3.1 Introducere în Datastore

App Engine oferă două opțiuni diferite de stocare de date, diferențiate de tipul de disponibilitate și de consistență garantate:

- Datastore **Master/Slave** folosește un sistem master-slave de replicare, care multiplică asincron datele în timp real când sunt scrise, într-un data center fizic. Deoarece doar un singur data center este master pentru scriere în oricare moment de timp, această opțiune oferă o consistență foarte mare pentru toate citirile și interogările, cu costul de a avea perioade de indisponibilitate în timpul problemelor acelui data center sau ale închiderilor sale planificate. Această opțiune oferă de asemenea cel mai scăzut cost pentru procesare și stocare de date.
- În datastore-ul **High Replication** datele sunt replicate pe mai multe data center folosind un sistem bazat pe algoritmul *Paxos*. High Replication oferă o disponibilitate foarte bună pentru citiri și scrieri (cu costul de a avea scrieri mai lente). Majoritatea interogărilor sunt eventual consistente. Costurile pentru stocare și putere de procesare sunt aproximativ de trei ori mai ridicate decât în cazul opțiunii Master/Slave.

Datastore-ul App Engine salvează obiecte cu date, cunoscute drept **entități**. O entitate are una sau mai multe proprietăți, care sunt de tipurile suportate prezentate în Tabel 2. De exemplu, o proprietate poate fi un șir de caractere, un întreg sau chiar o referință la altă entitate.

Datastore-ul poate executa mai multe operații într-o singură tranzacție. Prin definiție, o tranzacție nu se poate executa cu succes decât dacă fiecare operație din tranzacție se execută cu succes. Dacă oricare dintre operații eșuează, se face automat rollback la tranzacție. Acest lucru este util în special pentru aplicațiile web distribuite, unde mai mulți utilizatori ar putea accesa sau manipula aceleași date în același timp.

Spre deosebire de bazele de date tradiționale, Datastore folosește o arhitectură distribuită pentru a scala automat seturi foarte largi de date. Este foarte diferită de o bază de date relațională obișnuită în ceea ce privește relațiile dintre obiectele sale. Două entități de același fel pot avea proprietăți diferite. Entități diferite pot avea proprietăți cu același nume, dar de tipuri diferite. Deși interfața Datastore are multe din funcționalitățile obișnuite ale bazelor de date relaționale, caracteristicile sale unice implică totuși o abordare diferită în arhitectura modelării datelor aplicației pentru a putea folosi avantajul de scalabilitate automată.

3.3.2 API-ul Python pentru Datastore

În Python, entitățile pentru stocare sunt create din obiecte Python; atributele obiectelor devin atribute ale entității. Pentru a crea o nouă entitate, se poate apela (dacă se dorește) clasa de bază (părintele) entității, se pot modifica atributele obiectului și apoi salva obiectul (apelând o metodă precum *.put()*). Clasa de bază devine numele entității. Modificarea unei entități existente necesită recuperarea în prealabil a respectivului obiect (de exemplu folosind o interogare), modificarea proprietăților și apoi salvarea sa cu noile proprietăți.

Entitățile din datastore nu au o schemă anume: două entități de același fel nu au neapărat aceleași proprietăți; nici nu e nevoie să folosească același tipuri de date pentru aceleași proprietăți. Aplicația este responsabilă ca entitățile să adere la o anumită schemă, atunci când e nevoie de aceasta. În acest scop, SDK-ul Python include o bibliotecă plină de funcționalități pentru modelare de date care fac ușoară aderarea la o anumită schemă dorită.

În API-ul Python, un model descrie un tip de entitate, incluzând tipurile și configurația atributelor. O aplicație definește un model folosind clase Python, cu parametrii claselor descriind configurația atributelor. Entitățile de același tip sunt reprezentate de instanțe ale clasei model corespunzătoare, cu parametrii instanței reprezentând valorile atributelor. O entitate poate fi creată folosind constructorul clasei, apoi stocată folosind metoda *put()*:

```
import datetime
from google.appengine.ext import db
from google.appengine.api import users
class Employee(db.Model):
    name = db.StringProperty(required=True)
    role = db.StringProperty(choices=set(["executive", "manager"]))
    hire_date = db.DateProperty()
    new_hire_training_completed = db.BooleanProperty()
    account = db.UserProperty()
e = Employee(name="",
              role="manager",
              account=users.get_current_user())
e.hire_date = datetime.datetime.now().date()
e.put()
```

API-ul datastore oferă două interfețe pentru interogări: o interfață model de interogare și un limbaj de interogare asemănător cu SQL, numit GQL. O interogare returnează entități de forma claselor ale clasei-model care pot fi modificate și puse înapoi în datastore:

```
training_registration_list = [users.User("Alfred.Smith@example.com"),
                              users.User("jharrison@example.com"),
                              users.User("budnelson@example.com")]
employees_trained = db.GqlQuery("SELECT * FROM Employee WHERE account IN :1",
                                training_registration_list)
for e in employees_trained:
    e.new_hire_training_completed = True
    db.put(e)
```


3.3.3 Entități și proprietăți

Un obiect stocabil în datastore-ul App Engine este cunoscut drept **entitate**. O entitate are una sau mai multe atribute, adică valori de unul sau mai multe tipuri de date incluzând numere întregi, valori în virgulă mobilă, șiruri de caractere, date stocate binar și multe altele.

Fiecare entitate are și o cheie atașată care identifică unic acea entitate. Cea mai simplă cheie are un tip și un ID de entitate dat de către datastore. Tipul categorizează entitatea pentru a putea interoga mai ușor. ID-ul entității poate fi reprezentat și printr-un șir de caractere.

O aplicație poate recupera o entitate din datastore folosind cheia sa, ori prin efectuarea unei interogări care se mulează pe atributele entității. O interogare poate returna zero sau mai multe entități și poate returna valori sortate după valori ale atributelor. O interogare poate de asemenea limita numărul de rezultate returnate pentru a conserva memoria, timpul de execuție și puterea de calcul necesară.

Spre deosebire de bazele de date relaționale, datastore-ul App Engine nu necesită ca toate entitățile să fie de un anumit tip sau să posede aceleași atribute.

Un atribut poate avea mai multe valori. O proprietate cu valori multiple poate avea valori de tipuri diferite. O interogare pe o proprietate cu mai multe valori testează dacă există printre ele valori care să îndeplinească criteriul de interogare. Acest lucru face aceste atribute să fie utile la testarea de apartenență.

3.3.4 Diferențe față de SQL

Datastore-ul App Engine diferă de bazele de date relaționale obișnuite în multe feluri importante.

Datastore-ul App Engine a fost proiectat să scaleze, oferind aplicațiilor posibilitatea de a menține o performanță la fel de mare atunci când primesc din ce în ce mai mult trafic. Scrierile în datastore se scalează automat distribuind datele după necesitate. Citirile din datastore scalează deoarece singurele interogări suportate sunt cele ale căror performanțe scalează cu dimensiunea setului de rezultate (spre deosebire de dimensiunea setului de date). Acest lucru înseamnă că o interogare al cărui set de rezultate conține 100 de entități se comportă la fel relativ la performanță indiferent dacă setul total de date a fost de sute de entități sau de milioane de entități. Această proprietate este motivul principal din care unele tipuri de interogări nu sunt implementate.

Deoarece toate interogările din App Engine sunt servite de către index-uri create a priori, tipurile de interogări care pot fi executate sunt mult mai restrictive decât cele posibile într-o bază de date relațională cu SQL. **Nu se pot face JOIN-uri de tabele în datastore.** Datastore nu

permite nici filtrarea prin inegalitate pe proprietăți multiple, nici filtrarea de date bazată pe rezultate ale unor sub-interogări.

Spre deosebire de bazele de date relaționale obișnuite, datastore-ul App Engine nu necesită tipuri de date pentru a avea un set de attribute consistent (deși acest lucru se poate implementa în codul aplicației). La interogarea datastore-ului nu este încă posibilă returnarea doar a unui subset de attribute. Datastore-ul App Engine poate fie să returneze entități de-a întregul, fie chei de entități.

3.3.5 Modelarea entităților

3.3.5.1 Arhitectura

Clasa *Model* este super-clasa pentru definițiile de modele de date.

Model se află în modulul *google.appengine.ext.db*.

O aplicație definește un model de date definind o clasă care extinde *Model*. Attributele acelei clase sunt apoi definite ca și attribute ale entității de stocat. Valorile acestor attribute trebuie să facă parte din tipurile de date acceptate de datastore (vezi Tabel 2). Un exemplu de Model:

```
class Story(db.Model):
    title = db.StringProperty()
    body = db.TextProperty()
    created = db.DateTimeProperty(auto_now_add=True)
```

O aplicație creează o entitate nouă instanțiând o sub-clasă a clasei *Model*. Attributele entității pot fi alocate folosind attribute ale instanței sau ca și argumente în constructor:

```
s = Story()
s.title = "The Three Little Pigs"
s = Story(title="The Three Little Pigs")
```

Numele modelului sub-clasei este folosit ca și numele tipului de entitate din datastore. Numele atributelor folosite într-o entitate sunt numele atributelor corespunzătoare din clasa ce le-a generat. Attributele unui model al căror nume încep cu un underscore (_) sunt ignorate, așa că aplicația poate folosi aceste attribute pentru a stoca date ce nu se salvează în datastore.

O entitate poate avea și o entitate părinte opțională. Relațiile părinte-copil formează grupuri de entități, care sunt folosite pentru a controla tranzacționalitatea și localizarea datelor în datastore. O aplicație creează o relație părinte-copil între două entități pasând entitatea părinte în constructorul entității copil, ca și argument *parent*.

Fiecare entitate are o cheie, un identificator unic ce reprezintă entitatea. O entitate poate avea un nume opțional de cheie, un șir de caractere unic printre toate entitățile de acel tip.

Tabel 2: Proprietățile și tipurile de valori în GAE datastore

Value type	Python type	Sort order	Notes
Boolean	<code>bool</code>	<code>False < True</code>	
Byte string, short	<code>db.ByteString</code>	byte order	Up to 500 bytes
Byte string, long	<code>db.Blob</code>	<i>n/a</i>	up to 1 megabyte; not indexed
category	<code>db.Category</code>	Unicode	
Date and time		chronological	
email address	<code>db.Email</code>	Unicode	
floating point number	<code>float</code>	numeric	64-bit double precision, IEEE 754
geographical point	<code>db.GeoPt</code>	by latitude, then longitude	
Google Accounts user	<code>users.User</code>	email address in Unicode order	
integer	<code>int</code> or <code>long</code>	numeric	64-bit integer, signed
key, blobstore	<code>blobstore.BlobKey</code>	byte order	
key, datastore	<code>db.Key</code>	by path elements (kind, ID or name, kind, ID or name...)	
link	<code>db.Link</code>	Unicode	
messaging handle	<code>db.IM</code>	Unicode	
null	<code>None</code>	<i>n/a</i>	
postal address	<code>db.PostalAddress</code>	Unicode	
rating	<code>db.Rating</code>	numeric	
telephone number	<code>db.PhoneNumber</code>	Unicode	
Text string, short	<code>str</code> , <code>unicode</code>	Unicode (<code>str</code> stored as ASCII)	Up to 500 Unicode characters.
Text string, long	<code>db.Text</code>	<i>n/a</i>	up to 1 megabyte; not indexed

3.3.5.2 Metodele disponibile (salvare, selectare, ștergere)

Aplicațiile folosesc API-ul datastore pentru a crea noi entități, modifica entități existente, recupera entități și șterge entități. Dacă aplicația știe cheia completă a unei entități atunci aplicația poate acționa direct pe acea entitate folosind cheia. O aplicație poate de asemenea să facă o interogare pentru a determina cheile entităților ale căror valori îndeplinesc un anumit criteriu.

În Python, se creează o entitate nouă creând o nouă instanță a clasei `model`, apoi apelând metoda `put()` a instanței. Dacă obiectul nu a fost creat cu un parametru `key_name` în constructor,

apelul metodei *put()* populează automat cheia obiectului cu o cheie generată de sistem.

```
employee = Employee(first_name='Antonio',
                     last_name='Salieri')
employee.hire_date = datetime.datetime.now().date()
employee.attended_hr_training = True
employee.put()
```

Se poate apela și *db.put()* cu un model pentru a salva o instanță.

Pentru a recupera o entitate cu o anumită cheie, se apelează *db.get()* cu obiectul *Key*. Se poate produce obiectul *Key* folosind metoda *Key.from_path()*. Calea completă este o secvență de entități în calea părinților, cu fiecare entitate reprezentând de tipul (șir de caractere) urmat de numele cheii în șir de caractere sau ID numeric:

```
address_k = db.Key.from_path('Employee', 'asalieri', 'Address', 1)
address = db.get(address_k)
```

db.get() returnează o instanță a clasei model corespunzătoare. Trebuie ca acea clasă model să fie importată în prealabil, înaintea apelului *db.get()* în vederea recuperării.

Pentru a modifica o intrare existentă se modifică atributele obiectului și apoi se apelează metoda *put()*. Obiectul suprascrie entitatea existentă. Întregul obiect este trimis la datastore la fiecare apel *put()*.

O entitate se poate șterge fie apelând funcția *db.delete()*, fie apelând metoda *delete()* a instanței obiectului respectiv:

```
address_k = db.Key.from_path('Employee', 'asalieri', 'Address', 1)
db.delete(address_k)
employee_k = db.Key.from_path('Employee', 'asalieri')
employee = db.get(employee_k)
# ...
employee.delete()
```

3.4 Serviciul de autentificare Google

3.4.1 Generalități

Aplicațiile App Engine pot autentifica utilizatorii în oricare din următoarele trei metode: folosind Google Accounts, conturile de pe domenii Google Apps sau prin identificatori OpenID.

O aplicație poate detecta dacă utilizatorul curent este logat și poate redirecționa utilizatorul către pagina de autentificare corespunzătoare sau, dacă aplicația folosește Google Accounts, să creeze un cont nou. Cât timp un utilizator este autentificat în aplicație, aplicația are acces la valoarea adresei de email a utilizatorului (sau a identicatorului OpenID dacă aplicația folosește OpenID) și de asemenea la un ID unic de utilizator. Aplicația mai poate de asemenea detecta dacă utilizatorul autentificat este sau nu administrator, făcând ușoară implementarea secțiunilor în care au acces doar administratorii.

3.4.2 Autentificarea utilizatorilor în Python

Următorul exemplu salută un utilizator care s-a autentificat în aplicație cu un mesaj personalizat și un link pentru a se deloga. Dacă utilizatorul nu este autentificat, aplicația oferă un link către pagina de logare Google Accounts sau către pagina care cere un identificator OpenID.

```
from google.appengine.api import users
class MyHandler(webapp.RequestHandler):
    def get(self):
        user = users.get_current_user()
        if user:
            greeting = ("Welcome, %s! (<a href=\"%s\">sign out</a>)" %
                        (user.nickname(), users.create_logout_url("/")))
        else:
            greeting = ("<a href=\"%s\">Sign in or register</a>." %
                        users.create_login_url("/"))
        self.response.out.write("<html><body>%s</body></html>" % greeting)
```

Dacă aplicația folosește OpenID și utilizatorul trebuie să se autentifice, aplicația va fi redirecționată către URL-ul `/_ah/login_required`. Acolo trebuie creată o pagină care oferă posibilitatea utilizatorului să se autentifice cu un identificator OpenID. Pentru a specifica acea pagină, se adaugă o intrare în fișierul `app.yaml` în felul următor:

```
- url: /_ah/login_required
  script: do_openid_login.py
```

3.4.3 Securizarea prin autentificare a resurselor

Dacă există pagini care necesită ca utilizatorul să fie autentificat în vederea accesării lor, se poate configura din managementul paginilor aceleora ca paginile să necesite autentificarea a priori accesării direct din fișierul `app.yaml`. Dacă un utilizator accesează un URL configurat să necesite autentificare și utilizatorul nu este autentificat, App Engine îl redirecționează către pagina corespunzătoare pentru autentificare (atunci când se folosesc Google Accounts sau Google Apps) sau către `/_ah/login_required` (atunci când se folosește OpenID), apoi redirecționează utilizatorul înapoi la URL-ul aplicației după înregistrarea sau autentificarea reușite.

Managementul configurației poate de asemenea să necesite ca utilizatorul autentificat să fie administrator al aplicației. Aceasta face mai ușoară dezvoltarea secțiunilor din site care trebuie să fie disponibile doar administratorilor, fără a mai implementa un mecanism de autorizare separat.

Dacă setarea este `login: required`, o dată ce utilizatorul s-a autentificat, se continuă în mod obișnuit. Dacă setarea este `login: admin`, după ce utilizatorul s-a autentificat se verifică dacă acesta este și administrator al aplicației. Dacă nu, utilizatorului i se afișează un mesaj de eroare. Dacă utilizatorul este administrator, operația continuă în mod obișnuit. Dacă o aplicația necesită

un comportament diferit, se poate implementa managementul autentificării în aplicația în sine.

Exemplu:

```
handlers:
- url: /profile/.*
  script: user_profile.py
  login: required

- url: /admin/.*
  script: admin.py
  login: admin

- url: /.*
  script: welcome.py
```

Se poate configura handler-ul să refuze accesul la URL-urile protejate când utilizatorul nu este autentificat în loc de redirectionarea lui către pagina de autentificare. Un utilizator respins primește codul 401 de status HTTP. Pentru a configura aplicația să respingă utilizatorii care nu sunt autentificați, se adaugă *auth_fail_action: unauthorized* în configurația aplicației:

```
handlers:
- url: /secure_api/.*
  script: api_handler.py
  login: required
  auth_fail_action: unauthorized
```

Comportamentul implicit pentru *auth_fail_action* este *redirect* (redirecționarea utilizatorilor la pagina de autentificare).

3.4.4 Utilizatorii și Datastore-ul

O instanță a clasei **User** reprezintă un utilizator. Instanțele **User** sunt unice și comparabile. Dacă două instanțe sunt egale, atunci ele reprezintă același utilizator. Aplicația poate accesa utilizatorul curent autentificat apelând funcția *users.get_current_user()*.

```
from google.appengine.api import users
user = users.get_current_user()
if not user:
    # The user is not signed in.
else:
    print "Hello, %s!" % user.nickname()
```

Se poate folosi funcția *users.get_current_user()* indiferent de ce opțiune pentru autentificare folosește aplicația.

O instanță de utilizator poate fi construită cu adresa de e-mail:

```
user = users.User("Albert.Johnson@example.com")
```

Sau, pentru OpenID, se poate crea în felul următor:

```
users = users.User(federated_identity="http://example.com/id/ajohnson")
```

Dacă constructorul pentru **User** este apelat cu o adresă de email care nu corespunde unui cont Google valid, obiectul va fi creat dar nu va corespunde unui cont de Google existent. Contul

acela nu va fi valid nici dacă între timp un utilizator va crea un cont Google cu acea adresă de email după ce obiectul a fost stocat.

Când se folosește SDK-ul oferit și aplicația rulează sub serverul web de dezvoltare pe mașina locală a dezvoltatorului, toate obiectele User sunt considerate a reprezenta conturi valide Google atunci când sunt stocate în datastore-ul simulat.

Obiectul User pentru un utilizator valid poate oferi un ID unic pentru acel utilizator care rămâne același chiar dacă utilizatorul își schimbă adresa de email. Metoda `user_id()` returnează acest ID, un șir de caractere.

Obiectul User are aceeași formă indiferent de metoda de autentificare pe care aplicația o folosește. Dacă se schimbă între opțiunile de autentificare de la Google Accounts la OpenId, obiectele User existente în datastore sunt încă valide.

3.5 Servicii adiționale oferite de GAE

App Engine oferă o varietate de servicii care ajută dezvoltatorii să efectueze operații des întâlnite atunci când administrează aplicația. Următoarele API-uri sunt oferite pentru a accesa aceste servicii:

URL Fetch

Aplicațiile pot accesa resurse de pe Internet, cum ar fi web service-uri sau alte date folosind serviciul URL Fetch al App Engine. Serviciul URL Fetch descarcă resurse de pe web cu aceeași infrastructura de mare viteză Google folosită la multe alte produse Google.

Mail

Aplicațiile pot trimite email-uri folosind serviciul Mail al App Engine. Acest serviciu folosește infrastructura Google pentru a trimite email-uri.

Memcache

Serviciul Memcache oferă aplicațiilor o memorie de tampon de foarte mare performanță pentru stocarea în memoria RAM de obiecte cheie-valoare. Acest serviciu este accesibil de către mai multe instanțe ale aplicației. Memcache este folositor pentru date ce nu necesită persistența și facilitățile tranzacționale ale datastore-ului, cum ar fi date temporare sau date copiate din datastore în memoria tampon pentru a oferi acces de mare viteză la anumite resurse.

Image Manipulation

Serviciul Image oferă posibilitatea de a prelucra imagini. Cu acest API, se pot redimensiona, decupa, roti și întoarce imagini salvate în formate JPEG și PNG.

CAPITOLUL 4 Arhitectura REST

4.1 Ce este REST

Representational State Transfer (REST) este un stil de arhitectură software pentru medii hiperdistribuite cum ar fi World Wide Web. Termenul de Stare Reprezentațională (Representational State) a fost introdus și definit în anul 2000 de către Roy Fielding în disertația sa de doctorat. Fielding este unul dintre principalii autori ai specificațiilor 1.0 și 1.1 pentru HTTP (Hypertext Transfer Protocol).

Stilul arhitectural REST a fost dezvoltat în paralel cu HTTP/1.1, bazat pe design-ul existent al HTTP/1.0. Cea mai mare implementare a unui sistem conform stilului REST este World Wide Web-ul. REST exemplifică cum arhitectura Web-ului a reieșit din caracterizarea constrângerilor și macro-interacțiunilor dintre patru componente ale Web-ului și anume servere de origine, servere porți (gateway), proxy-uri și alți clienți, fără a impune limitări pe participanții individuali. Ca rezultat, REST guvernează în esență comportamentul adecvat al participanților.

O aplicație ce respectă constrângerile REST se numește „RESTful”.

4.2 Conceptul

Arhitecturile în stil REST constau din clienți și servere. Clienții inițiază cereri către servere; serverele procesează cererile și returnează răspunsuri în consecință. Cererile și răspunsurile sunt construite în jurul transferului de reprezentări de resurse. O resursă poate fi practic orice concept coerent și logic care poate fi adresat. O reprezentare a unei resurse este de regulă un document ce capturează starea curentă a unei resurse.

La orice moment de timp, un client poate ori să fie în tranziție între stări ale aplicației, ori să fie „în repaus” („at rest”). Un client în stare de repaus poate interacționa cu utilizatorul său, dar nu creează încărcare și nici nu consumă resurse ale clientului, nici de stocare și nici de rețea.

Clientul începe să trimită cereri atunci când este gata să facă o tranziție la o stare nouă. În timpul unei sau mai multor cereri în curs, clientul este considerat a fi în tranziție. Reprezentarea fiecărei stări ale aplicației conține legături ce ar putea fi folosite data viitoare când clientul alege să inițializeze o nouă tranziție.

REST a fost inițial descris în contextul HTTP-ului, dar nu este limitat la acel protocol. Arhitecturile RESTful pot fi bazate și pe alte protocoale Application Layer dacă acestea oferă deja un vocabular bogat și uniform pentru aplicații bazate pe transferul de stări reprezentationale ce au înțeles. Aplicațiile RESTful maximizează utilizarea interfeței pre-existente și bine definite și a altor capacități ce vin odată cu protocolul de rețea ales și minimizează necesitatea adăugării de noi facilități specifice aplicației.

4.3 Constrângeri

Stilul arhitectural REST descrie șase constrângeri aplicate la nivelul arhitecturii, lăsând liberă implementarea componentelor individuale:

- **Client-server**

Clienții sunt separați de servere printr-o interfață uniformă. Această separare a preocupărilor înseamnă că, de exemplu, clienții nu sunt preocupați de stocarea datelor, care rămâne internă pentru fiecare server, așa că portabilitatea codului clientului este sporită. Serverele nu sunt preocupate de interfața utilizatorului sau starea utilizatorului, așa că serverul poate fi mai simplu și mai scalabil. Serverele și clienții pot de asemenea fi înlocuite și dezvoltate independent, atâta timp cât interfața rămâne neschimbată.

- **Stateless (fără stare)**

Comunicarea client-server este în continuare restrânsă de faptul că niciun context al clientului nu trebuie stocat pe server între cereri. Fiecare cerere de la oricare client conține toate informațiile necesare pentru a îndeplini cererea și orice stare de sesiune este ținută în client. Serverul poate ține starea; această constrângere necesită doar ca starea de pe server să poată fi adresabilă de către un URL ca și resursă. Acest lucru nu face doar serverele să fie mai vizibile pentru monitorizare, dar le face și mai fiabile în fața eșecurilor parțiale de rețea și le îmbunătățește scalabilitatea.

- **Cacheable**

Ca și pe World Wide Web, clienții pot să stocheze temporar răspunsurile. Răspunsurile trebuie, așadar, implicit sau explicit, să se definească singure ca fiind sau nu stocabile temporar, pentru a preveni refolosirea unor stări neactualizate sau date necorespunzătoare de către clienți. Gestionarea bună a acestui comportament elimină interacțiuni client-server, îmbunătățind și mai mult scalabilitatea și performanța.

- **Layered system (sistem în mai multe straturi)**

Un client nu poate în mod normal să își dea seama dacă este conectat direct la serverul cu care dorește să comunice sau mai există intermediari pe parcursul drumului. Serverele intermediare ar putea îmbunătăți scalabilitatea sistemului introducând balansarea încărcării și oferind *cache*-uri partajate. Acestea ar putea și să impună politici de securitate.

- **Code on demand (cod la cerere) – opțional**

Serverele pot extinde sau modifica temporar funcționalitatea unui client transferând sarcini către el pe care să le execute. Exemple în acest sens pot include componente compilate precum applet-urile Java sau scripturi client cum ar fi JavaScript

- **Uniform interface (interfață uniformă)**

Interfața uniformă între clienți și servere, simplifică și decuplează arhitectura, ceea ce permite fiecărei părți să evolueze independent. Există patru principii ale acestei interfețe:

- Identificare resurselor
- Manipularea resurselor prin reprezentări standard
- Mesaje ce se auto-descriu
- Hypermedia să fie motorul stării aplicației

4.4 Țeluri principale

- Scalabilitate și interacțiuni pe componente
- Generalitatea interfețelor
- Distribuirea independentă a componentelor
- Componente intermediare pentru a reduce latența, aplicarea politicilor de securitate și pentru încapsularea sistemelor vechi

REST a fost aplicat în descrierea arhitecturii web dorite, pentru a ajuta la identificarea problemelor existente, pentru compararea soluțiilor alternative și pentru a asigura că extensiile protocolului nu vor încălca constrângerile care fac Web-ul reușit.

4.5 Webservice-uri RESTful

Un webservice RESTful (denumit și RESTful web API) este un simplu webservice implementat folosind HTTP și principiile REST. Este o colecție de resurse, cu trei aspecte definite:

- baza URI a webservice-ului, de exemplu `http://exemplu.com/resurse/`
- tipul de date media Internet suportat de către webservice. Acesta este de regulă JSON, XML, YAML sau o combinație de acestea dar poate fi foarte ușor orice tip de date media Internet valid.
- setul de operații suportat de către webservice folosind metode HTTP (ex. POST, GET, PUT sau DELETE).

Metodele PUT și DELETE sunt metode idempotente. Metoda GET este o metodă sigură, însemnând că apelul ei nu produce efecte secundare (aceasta implică de asemenea idempotență). Spre deosebire de webservice-urile SOAP, nu există un standard „oficial” pentru webservice-urile RESTful. Acest lucru se întâmplă deoarece REST este o arhitectură, în timp ce SOAP este un protocol. Chiar dacă REST nu este un standard, implementările RESTful cum ar fi Web-ul pot folosi standarde precum HTTP, URI, XML etc.

Tabelul 3 arată cum metodele HTTP sunt de obicei folosite pentru a implementa un webservice RESTful.

Tabel 3: Implementare tipică a unui webservice RESTful

Resursa	GET	PUT	POST	DELETE
URI de colecție, de exemplu <code>http://exemplu.com/resurse/</code>	Listarea de URI-uri și poate alte detalii despre membrii colecției.	Suprascriere a întregii colecții cu o altă colecție.	Crearea unei noi intrări în colecție. Noul URL al intrării este alocat automat și de obicei returnat de către server.	Șterge toată colecția.
URI de element, de exemplu <code>http://exemplu.com/resurse/ef7d-xj36p</code>	Recuperarea unei reprezentări ale membrului dorit într-o formă de date de tip media Internet.	Suprascriere a membrului adresat, iar dacă acesta nu există se crează .	Tratarea membrului ca pe o colecție și crearea unei noi intrări în ea.	Șterge membrul adresat din colecție.

CAPITOLUL 5 Prezentarea aplicației

5.1 Generalități

Aplicația Online Backlog dorește să aducă utilizatorilor săi posibilitatea de a menține un backlog virtual, online, disponibil oricând și oriunde. Dezvoltatorii de software pot adăuga în aplicație notițe așa cum ar face-o pe o tablă fizică pentru backlog, însă în acest caz se stochează automat și informații utile, de pildă utilizatorul care a adăugat nota sau data și ora ultimei modificări. Pentru o grupare mai ușoară a notelor se pot folosi culori diferite și *#tag-uri*, scrierea de cuvinte cheie cu semnul diez în față automat categorisește nota din care face parte.

Interfața trebuie să fie foarte ușor de folosit, să ofere cât mai multe informații necesare într-un mod natural, să scaleze bine pentru mulți utilizatori și să folosească cât mai puține resurse, atât de-ale utilizatorului cât și ale serverului deoarece a fost gândită pentru cloud, așadar toate resursele irosite se plătesc fără motiv.

S-a optat pentru crearea unei aplicații web deoarece acestea sunt în ziua de astăzi cele mai folosite variante; respectarea noilor standarde și realizarea unei interfețe de stocare independente de client (interfața utilizatorului) au fost alte puncte de referință în realizarea aplicației.

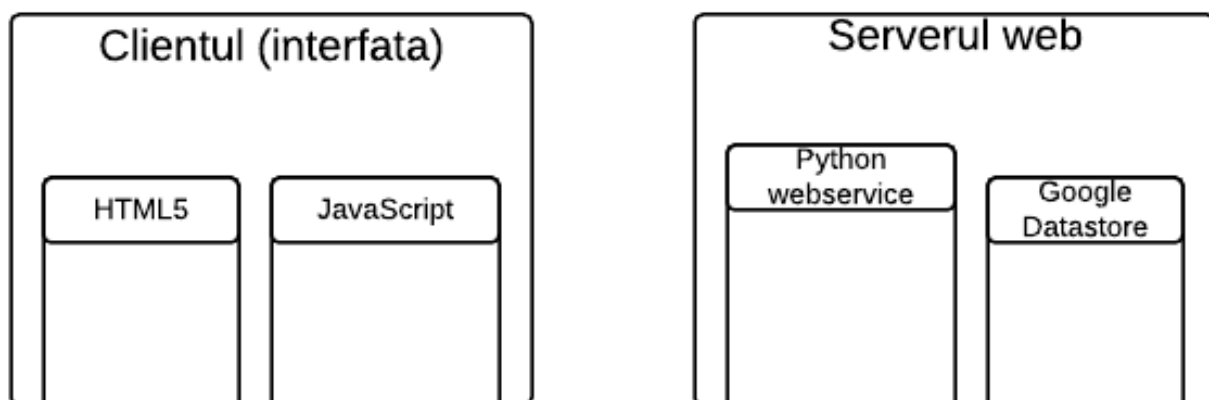
5.2 Arhitectura

Aplicația este împărțită în două unități logice diferite: serverul – realizat în Python sub Google App Engine – unde rulează un webservice REST și clientul – interfața realizată în HTML5 și limbajul de scriptare JavaScript (vezi Figura 6). Deoarece serverul folosește arhitectura REST, se pot crea numeroși clienți independenți care să folosească același server și care să comunice unitar. De exemplu, ar putea fi creat un client desktop în C++, un client de mobil pentru Android în Java sau pentru iPhone în Objective-C. Resursele oferite de web service în format XML sau JSON pot fi de asemenea consumate de către alte programe, chiar și server-side pentru agregarea lor și crearea de servicii mai complexe, deasupra celor de bază care sunt deja implementate.

Pentru simplitate, s-a optat pentru stocarea fișierelor necesare clientului (**index.html**, fișierele JavaScript, stilurile CSS și imaginile) în același loc cu serverul web – rezidă tot pe Google App Engine – deși puteau fi rulate de oriunde altundeva (chiar și local, fără server web).

Securitatea aplicației este oferită de transferul de date criptat, strict prin protocolul HTTPS, configurat în fișierul *app.yaml* așa încât serverul redirecționează automat utilizatorul când acesta încearcă să acceseze resursele nesecurizat (prin HTTP). Managementul utilizatorilor se realizează prin utilizarea serviciului Google Accounts. Pentru un control mai bun

Figura 6: Arhitectura structurală



al accesului utilizatorilor, s-a folosit un domeniu administrat prin Google Apps – în acest fel, aplicația poate fi instalată de orice companie ce folosește Google Apps și aceasta poate ușor să modifice drepturile de acces ale angajaților săi.

Webservice-ul implementat folosește pentru stocare datastore-ul App Engine, care este capabil să scaleze oricât este nevoie, accesul la date nefiind o problemă. Deoarece datastore-ul este în cloud, datele sunt păstrate în siguranță și există redundanța datelor. În ceea ce privește puterea de calcul, dacă această devine insuficientă, se poate opta oricând pentru replicarea aplicației pe mai multe instanțe sau pe servere virtuale mai puternice ce vor lucra simultan sub un *load-balancer* și care vor accesa același datasource – așadar datele vor fi mereu aceleași indiferent de ce instanță va servi cererea.

Clientul dezvoltat pentru această aplicație va rula într-un browser ce respectă noile standarde HTML5 și JavaScript ECMA-262, versiunea 3. S-a folosit framework-ul jQuery pentru implementarea clientului, comunicarea cu serverul realizându-se prin cereri AJAX de tip GET, POST sau DELETE (pentru ștergerea de entități). Folosirea AJAX asigură eficiența aplicației, deoarece aceasta comunică cu serverul doar când și cât are nevoie, nemaifiind nevoie de a reîncărca o pagină întreagă cu resursele ei aferent atunci când se face o modificare foarte mică. Prin această metodă se poate modifica direct entitatea vizată, atât pe server, cât și în interfața utilizatorului.

5.3 Implementarea serverului

În configurarea aplicației cu ajutorul fișierului *app.yaml* s-au definit cazurile în care utilizatorii au acces la resurse. Accesul la date se face întotdeauna securizat prin specificarea atributelor *secure: always* în fiecare dintre managerii de cereri.

Primul caz tratat este atunci când utilizatorul încarcă aplicația în browser, accesând adresa acesteia (numele aplicației.appspot.com). În acest caz, serverul oferă utilizatorului fișierul *index.html* din rădăcina directorului aplicației. Atributul *login: required* forțează autentificarea utilizatorului, redirecționându-l pe acesta la pagina corespunzătoare introducerii numelui de utilizator și parolei (vezi Figura 7). După autentificarea reușită utilizatorul este redirecționat înapoi către aplicație.

Figura 7: Autentificarea automată

NAVIGHEAZA
Bun venit la tXK Industries

Conectați-vă în cont la
tXK Industries

Nume de utilizator:

@navigheaza.ro

Parolă:

☒ Ține-mă minte

[Nu vă puteți accesa contul?](#)

Online Backlog Utilizează Contul dvs. tXK Industries pentru a se conecta.

Google nu este afiliat cu conținutul **Online Backlog** sau cu deținătorii acestei aplicații. Dacă vă accesați contul, Google va permite aplicației **Online Backlog**, accesul la adresa dvs. e-mail, însă nu și la parolă sau la orice alte informații personale.

Online Backlog poate utiliza adresa dvs. de e-mail pentru personalizarea experienței dvs. pe site-ul respectiv.

©2011 Google [Politica de confidențialitate](#) - [Condiții de furnizare a serviciilor](#)

Dezvoltat de 

Următorul caz tratat este cazul fișierelor statice: *favicon.ico*, *robots.txt* și folderul *assets* ce conține fișierele-imagini, stilul CSS și fișierele JavaScript. Următoarele două reguli (pentru *favicon.ico* și *robots.txt*) sunt alias-uri din folderul *assets*, deoarece browserele întotdeauna caută fișierul *favicon.ico* în rădăcina folderului web, iar roboții mereu caută *robots.txt* în folderul rădăcină. Pentru aceste fișiere nu este nevoie de autentificare.

Ultima regulă din fișierul de configurare este cea mai importantă, aceasta spune că restul cererilor netratate până la acea linie, să fie trimise către script-ul **main.py**. Acest script este responsabil cu crearea unui URL pentru delogare și deservește ca și web service REST. Prin urmare, s-a folosit atributul *login: required* și, mai mult, s-a modificat acțiunea implicită de redirecționare a utilizatorului către pagina de logare în acțiunea de a restricționa accesul și de a trimite codul HTTP *401: Unauthorized*. Acest comportament a fost dorit deoarece utilizatorul nu accesa direct webservice-ul introducând manual URL-urile aferente lui ci cu ajutorul clientului scris în JavaScript, în funcție de acțiunile utilizatorului se trimit cereri către această resursă. Folosind codul HTTP 401, clientul își dă seama că utilizatorul s-a delogat și poate acționa în funcție de aceasta. Dacă s-ar fi optat către comportamentul standard de redirecționare, pentru clientul JavaScript nu ar fi avut o relevanță semantică la fel de bună.

Figura 8: Extras din configurarea aplicației

```
handlers:
- url: /
  login: required
  static_files: index.html
  upload: index.html
  secure: always

- url: /favicon.ico
  static_files: assets/favicon.ico
  upload: assets/favicon.ico
  mime_type: image/vnd.microsoft.icon
  secure: always

- url: /robots.txt
  static_files: assets/robots.txt
  upload: assets/robots.txt
  secure: always

- url: /assets
  static_dir: assets
  login: required
  secure: always

- url: .*
  script: main.py
  login: required
  secure: always
  auth_fail_action: unauthorized
```

Serverul este implementat în totalitate în Python și se folosește de librăriile standard din API-ul Google pentru cloud și de încă o bibliotecă ce abstractizează o implementare pentru REST. Aceasta din urmă este capabilă de a servi cereri REST atât în format JSON (JavaScript Object Notation) cât și XML (eXtensible Markup Language) și de asemenea de a primi cereri în cele două formate mai sus menționate.

Scriptul principal rulat este minimal, practic acestea deleagă acțiunea **/logout** către URL-ul corespunzător delogării și creează modelul REST, în care adaugă cele două modele de stocare.

Din modalitatea de scriere a obiectului *rest*, serverul optimizează automat rularea aplicației, deoarece acesta nu este reinstanciat la fiecare cerere, ci păstrat în memorie de-a lungul rulării instanței aplicației.

Cum atributul *base_url* este setat ca fiind */rest*, toate cererile către webservice-ul aplicației se vor face pe URL-uri de forma `http://numeleaplicației.appspot.com/rest/`.

Figura 9: Fișierul principal al serverului

```
#!/usr/bin/env python
from google.appengine.api import users
from google.appengine.ext import webapp
from google.appengine.ext.webapp import util
from models import project_model, note_model
import auth
import rest

class LogoutHandler(webapp.RequestHandler):
    def get(self):
        self.redirect(users.create_logout_url("/"))

rest.Dispatcher.base_url = "/rest"
rest.Dispatcher.add_models({
    "Note": note_model,
    "Project": project_model
})
rest.Dispatcher.authorizer = auth.OwnerAuthorizer()

def main():
    application = webapp.WSGIApplication([
        ('/rest/.*', rest.Dispatcher), ('/logout', LogoutHandler)
    ])
    util.run_wsgi_app(application)

if __name__ == '__main__':
    main()
```


5.3.1 Modelarea entităților

Entitățile necesare aplicației sunt **Proiectul** și **Nota**. Deoarece nu avem o bază de date relațională, ci una proprietară, nu am folosit stocarea clasică tabelară ci paradigma noSQL a Google App Engine Datastore. Din acest motiv, nu a fost nevoie de crearea unui tabel de legătură pentru cele două entități pentru a modela relația de apartenență „ce note aparțin căror proiecte” și nici relația „ce developeri aparțin căror proiecte”. Pentru aceste relații s-au folosit o coloana de referință la entitatea Proiect în entitatea Notă, respectiv o listă cu tipul de date User în entitatea Proiect.

Figura 10: Modelul Proiect

```
from google.appengine.ext import db
from google.appengine.api import users

class project_model(db.Model):
    ''' are key implicit '''
    title = db.StringProperty(required=True)
    owner = db.UserProperty(required=True, auto_current_user_add=True)
    developers = db.ListProperty(users.User)
    created_at = db.DateTimeProperty(auto_now_add=True)
    modified_at = db.DateTimeProperty(auto_now=True)
    _editable = db.BooleanProperty()
```

În modelul Proiect se remarcă existența câmpurilor definite obligatorii *title* și *owner* (care se adaugă automat în funcție de utilizatorul autentificat în aplicație), câmpurile *developers* care este o listă de entități *User*, datele creării și modificării. Pe lângă acestea mai apare și atributul *_editable*, care nu se stochează în datastore (deoarece numele său începe cu un underscore) și care ajută la funcționalitatea clientului deoarece, în timpul autorizării această proprietate este modificată dinamic în funcție de ce utilizator este logat.

Figura 11: Modelul Notă

```
from google.appengine.ext import db
from google.appengine.api import users
from models import project_model

class note_model(db.Model):
    ''' are key implicit '''
    project = db.ReferenceProperty(reference_class=project_model,
                                   required=True)
    description = db.TextProperty(required=True)
    owner = db.UserProperty(required=True, auto_current_user_add=True)
    pos_x = db.IntegerProperty()
    pos_y = db.IntegerProperty()
    color = db.StringProperty()
    created_at = db.DateTimeProperty(auto_now_add=True)
    modified_at = db.DateTimeProperty(auto_now=True)
    _editable = db.BooleanProperty()
```

Modelul Notă are câteva atribute comune cu modelul Proiect (*owner*, datele creării și modificării) și în plus de acestea o referință la modelul Proiect pentru că fiecare Notă face parte dintr-un Proiect și numai unul, câmpul *description* care este cel mai important deoarece în acesta se stochează textul dorit, un câmp ce definește culoarea de fundal a notei și două coordonate de poziționare pentru a desena nota pe ecran în locul în care o dorește utilizatorul.

De asemenea și acest model conține câmpul nestocat *_editable* pentru comunicarea cu clientul, în vederea precizării dinamice a dreptului de scriere asupra entității vizate.

5.3.2 Securizarea webservice-ului

Pentru a avea un webservice funcțional în Python sub Google App Engine sunt necesare doar codul prezentat mai sus și bibliotecile aferente. Însă pentru ca aplicația să fie într-adevăr securizată pentru accesul nepermis la resurse chiar și a utilizatorilor autentificați trebuie implementat și un model de autorizare a accesului.

Acest model de autorizare acționează ca un filtru deasupra entităților cerute și trebuie implementate toate metodele interfeței pe care o extinde, adică *rest.Authorizer*. Regula principală a aplicației este că o entitate poate fi modificată / ștearsă doar de către utilizatorul care a creat-o. Mai există de asemenea și două restricții la citire, anume că un utilizator poate vedea doar acele entități Proiect pe care le-a creat sau în care este trecut în câmpul *developers* și poate vedea doar acele entități Notă pe care fie le-a creat, fie aparțin unui proiect vizibil pentru el.

Figura 12: Modelul de autorizare

```
from google.appengine.api import users
import rest

from google.appengine.ext import db
from models import project_model, note_model

class OwnerAuthorizer(rest.Authorizer):
    def can_read(self, dispatcher, model):
        user = users.get_current_user()
        model._editable = True if user == model.owner else False
        if model.__class__.__name__ == 'project_model':
            return
        if model.owner != users.get_current_user():
            dispatcher.not_found()

    def filter_read(self, dispatcher, models):
        ''' putem vedea proiectele noastre si cele in care suntem developeri '''
        if len(models) > 0 and models[0].__class__.__name__ == 'project_model':
            return self.filter_projects(models)
        ''' avem de filtrat notele: '''
        return self.filter_notes(models)

    def can_write(self, dispatcher, model, is_replace):
        user = users.get_current_user()
        model._editable = True if user == model.owner else False
        if(not model.is_saved()):
            #are voie sa creeze un proiect nou, dar verificam daca are voie nota:
            if model.__class__.__name__ == 'note_model':
```

```

        if user not in model.project.developers + [model.project.owner]:
            dispatcher.forbidden()
    elif model.owner != user:
        dispatcher.forbidden()

    def can_delete(self, dispatcher, model_type, model_key):
        ''' aceeași politica și la Projects și la Notes: acces are doar owner-ul '''
        query = model_type.all().filter("owner = ",
users.get_current_user()).filter("__key__ = ", model_key)
        if(len(query.fetch(1)) == 0):
            dispatcher.not_found()
        else:
            if model_type.__name__ == 'project_model':
                '''trebuie sa stergem de mana notele proiectului ca deh..'''
                for note in note_model.all().filter("project = ", model_key):
                    note.delete()
            def filter_projects(self, models):
                user = users.get_current_user()
                models = [model for model in models if user in [model.owner] +
model.developers ]
                return self.add_permissions(models)

            def filter_notes(self, models):
                user = users.get_current_user()
                models = [model for model in models if user in [model.project.owner] +
model.project.developers ]
                return self.add_permissions(models)
            def add_permissions(self, models):
                user = users.get_current_user()
                for model in models:
                    model._editable = True if model.owner == user else False
                return models

```

5.4 Implementarea clientului REST (interfeței utilizatorului)

Interfața utilizatorului se bazează pe arhitectura REST client-server, implementată prin apeluri AJAX cu metoda *\$.ajax* jQuery. Deoarece clientul este scris în limbajul JavaScript, am optat ca mesajele trimise și primite către server să se facă în JSON. Pentru ca aceste două cerințe să fie îndeplinite, în parametrii obiectului *\$.ajax* au trebuit setate două variabile de configurare: *dataType: 'json'* pentru ca serverul să trimită mesajele în format JSON și *contentType: 'application/json'* pentru ca serverul să accepte mesajele în format JSON.

Figura 13: Exemplu de comunicare REST – modificarea unei note

```

$.ajax({
    contentType: 'application/json',
    url: '/rest/Note/' + $card.data('key') + '?type=full',
    type: 'POST',
    dataType: 'json',
    data: JSON.stringify({
        'Note' : { 'description' : value }
    }),
    success: function(data) {
        $card.data(data.Note);
        $card.unblock();
    },
    error: ACTIONS.handleError
});

```

În ceea ce privește managementul erorilor, s-a folosit peste tot aceeași metodă *ACTIONS.handleError* ca și callback în toate requesturile AJAX. Această metodă comunică utilizatorului eventualele erori ce pot apărea în timpul modificării entităților, în funcție de codul HTTP al erorii. Clasa *Authorizer* a serverului trimite specific asemenea coduri HTTP când, spre exemplu, utilizatorul încearcă să salveze o resursă ce nu-i aparține sau când acesta cere o resursă la care nu are drept de citire sau când pur și simplu introduce date de intrare malformate, invalide pentru salvarea entităților (atunci când se omite un atribut marcat ca *required* în model de pildă).

Aplicația client se bazează foarte mult (dacă nu chiar aproape în totalitate) pe evenimente. În funcție de asemenea evenimente (click-ul undeva pe pagină, mouseover deasupra unei zone importante, drag-and-drop a unei note, dublu-click pe note etc.) se execută funcțiile aferente.

După terminarea procesului de încărcare a aplicației, utilizatorul ajunge în starea de așteptare. Elementele principale din această stare persistă pe tot parcursul aplicației. Ele sunt:

- zona de titlu de proiect,
- zona din stânga ecranului în care se află note colorate,
- zona din dreapta ecranului în care se află un meniu retractabil cu controale pentru proiecte.

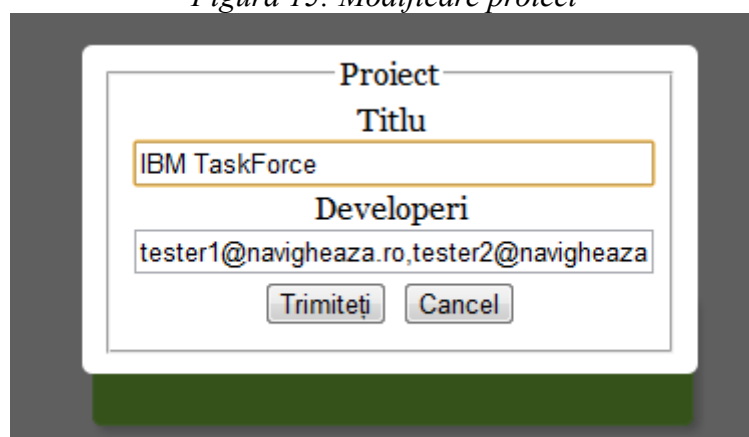
Figura 14: Ecranul de așteptare



În această primă stare, în care niciun proiect nu este încărcat, utilizatorul nu poate manipula entități de tip Notă. Pentru a începe să utilizeze cu adevărat aplicația, trebuie selectat (sau dacă nu există intrări în listă creat în prealabil) un proiect, printr-un click pe numele său din listă. Dacă există proiecte create de utilizatorul logat, atunci în dreptul acestora vor apărea iconițele de modificare și de ștergere, oferindu-se posibilitatea de a controla total entitățile personale.

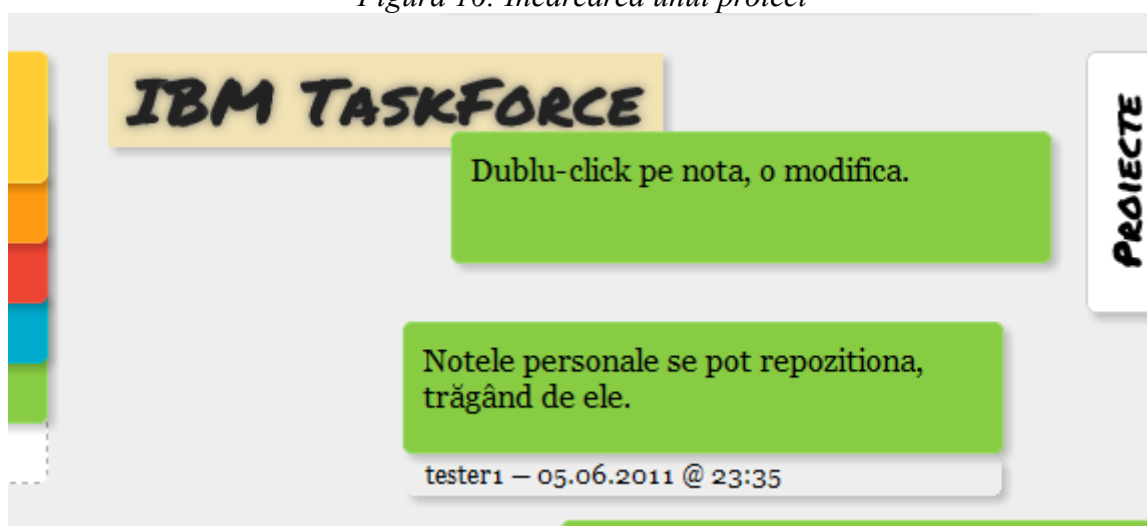
Un aspect important la crearea și modificarea de proiecte este completarea câmpului *developers*. Este important ca adresele de email separate prin virgule ce vor fi introduse în lista de dezvoltatori la un anumit proiect să fie conturi de Google Apps valide și deja existente. Introducerea unui email inexistent și crearea ulterioară a unui cont Google Apps cu aceeași valoare la adresa de email, nu va da acces utilizatorului în aplicație. Exemplu de modificare de proiect în Figura 15.

Figura 15: Modificare proiect



După selectarea unui proiect din listă și mutarea mouse-ului în afara zonei meniului retractabil, ecranul va arăta ca în Figura 16. Numele proiectului se schimbă imediat, apoi se face o cerere la server pentru preluarea notelor din proiectul selectat. Se blochează interacțiunea cu aplicația temporar prin intermediul extensiei de jQuery *blockUI*, iar după ce datele au fost preluate și notele au apărut pe ecran se deblochează aplicația (folosind comanda `$.unblockUI()`).

Figura 16: Încărcarea unui proiect



În Figura 16 se mai remarcă apariția unei bare suplimentare în josul uneia din note. Acest lucru se întâmplă de fiecare dată când utilizatorul trece cu mouse-ul deasupra notelor, afișându-se autorul și data însoțită de ora la care a fost modificată ultima oară nota respectivă.

Pentru a adăuga o notă pe ecran se poate trage cu mouse-ul de oricare dintre notele din stânga ecranului spre mijlocul paginii. Acest lucru va rezulta în crearea unei note de culoarea dorită și poziționarea ei în locul ales. Se va trimite imediat o cerere către server pentru a salva noua notă, după care se pot re-modifica culoarea, textul și poziția după plac.

Figura 17: Meniul unei note personale



Pentru a modifica textul unei note trebuie să se intre în starea de modificare. Acest lucru se poate face fie mergând cu mouse-ul deasupra notei și apăsând pe icoana corespunzătoare modificării ce va apărea în colțul din dreapta-sus al notei (vezi Figura 17), fie prin dublu-click pe notă. De asemenea, meniurile ce apar pentru notele personale oferă și posibilitatea schimbării culorii și de a șterge nota.

În ecranul de modificare, nota asupra căreia se fac modificările iese în evidență, iar textul cu care se completează se poate formata folosind meniul de tipografie ce apare, sau cu scurtăturile obișnuite de tastatură (CTRL-B pentru Bold, CTRL-I pentru Italic, CTRL-S pentru salvare etc.). Se pot introduce și *hashtag*-uri în stilul celor de la *twitter.com*, pentru a categorisi ușor notele. Ecranul de modificare este prezentat în Figura 18.

Figura 18: Ecranul de modificare



După ce se marchează mai multe note cu aceleași hashtag-uri, se pot evidenția notele aparținente aceleiași categorii apăsând pe butoanele ce se formează în jurul cuvintelor cheie:

Figura 19: Hashtag-uri în note



Această categorisire rapidă este extrem de utilă în producția de software, mai ales deoarece notele nu mai trebuie să fie de aceeași culoare ca să se asemene.

Toate acțiunile de a șterge entități sunt întâi precedate de un ecran de siguranță în care se cere confirmarea ștergerii entităților pentru a micșora efectele eventualelor greșeli umane. O încercare însă de a trimite o cerere la server modificată (deoarece aplicația nu va afișa niciodată un buton de ștergere pe o resursă care nu aparține utilizatorului) va eșua cu un mesaj de eroare în care va scrie *Interzis! Ați încercat să faceți o operație la care nu aveți acces.*

Bibliografie

- Ahnson, Syed și Mohammad Ilyas. *Cloud Computing and Software Services: Theory and Techniques*. CRC Press, 2011
- Rittinghouse, John și James Ransome. *Cloud Computing: Implementation, Management and Security*. CRC Press, 2009
- Mather, Tim, Subra Kumaraswamy și Shahed Latif. *Cloud Security and Privacy*. O'Reilly, 2009
- Reese, George. *Cloud Application Architectures*. O'Reilly, 2009
- Resnick, Steve, Aaron Bjork și Michael de la Maza. *Professional Scrum with Team Foundation Server 2010*. Wiley Publishing, 2011
- Schwaber, Ken. *Agile Project Management with SCRUM*. Microsoft Press, 2004
- Dubakov, Michael. *Agile Tools. The Good, the Bad and the Ugly*. (format PDF - <http://targetprocess.com/download/whitepaper/agiletools.pdf>)
- Beck, Kent. "Manifesto for Agile Software Development". Agile Alliance. <http://agilemanifesto.org> Verificat la data de 2011-06-06.
- *Agile software development*, http://en.wikipedia.org/wiki/Agile_software_development
- Richardson, Leonard și Sam Ruby, *RESTful Web Services*, O'Reilly, 2007
- *Representational State Transfer* http://en.wikipedia.org/wiki/Representational_State_Transfer
- *Google App Engine* <http://code.google.com/appengine/docs/>