UNIVERSITATEA BABEŞ-BOLYAI CLUJ-NAPOCA
FACULTATEA DE MATEMATICĂ ŞI INFORMATICĂ
SPECIALIZAREA INFORMATICĂ

LUCRARE DE LICENŢĂ

Crearea si Folosirea unui API

Conducător ştiinţific
Dr. Conf. Suciu Mihai

Absolvent
Belea Radu

2023

ABSTRACT

Abstract: un rezumat în limba engleză cu prezentarea, pe scurt, a conținutului pe capitole, punând accent pe contribuțiile proprii și originalitate

Cuprins

Capitolul 1

Introducere

Introducere: Application Programming Interfaces (API) sunt componente esențiale în dezvoltarea de software din zilele noastre. Ele permit comunicarea între mai multe sisteme software și facilitează dezvoltarea de servicii și aplicații complexe. API-urile, după cum le spune și numele, reprezintă interfețe standard care maschează complexitatea codului, ceea ce permite folosirea funcționalităților fără a ști modul în care au fost implementate.

Pe măsură ce companiile au început să folosească serviciile de tip cloud, multe API-uri au fost create pentru a ajuta programatorii. Din acest motiv, trebuie asigurat faptul că un API este ușor de folosit și eficient.

În această lucrare, vom explora modul în care API-urile sunt create și folosite, tipurile diferite de API-uri și modurile în care sunt folosite. Vom enumera de asemenea și provocările de care avem parte în design-ul și dezvoltarea lor. Documentația, testarea și versionarea API-urilor este de asemenea importantă, și vom prezenta și niște exemple de modele de API-uri. Vom intra în detaliu despre partea practică, o aplicație web care explorează folosirea de API-uri publice, precum și crearea unui API nou.

Scopul final al lucrării este de a prezenta modul în care API-urile sunt create și folosite și de a oferi o imagine de ansamblu asupra tehnicilor ”best practice” adoptate în dezvoltarea aplicațiilor folosind API-uri.

Related work

În următoarele capitole vom analiza o prezentare asupra dezvoltării de API de Joshua Bloch [Jos], cărțile scrise de Fernando Doglio [Dog15a] și Sanjay Patni [Pat17] legate de API-urile REST și publicația lui Vijay Surwase [Sur16] legată de modelarea API-urilor REST.

Capitolul 2

Partea teoretica

2.1 API Design

”Un API poate fi unul din cele mai importante atu-uri, sau una din cele mai mari probleme ale unei companii.[...] API-urile publice vor exista pentru totdeauna. Au doar o șansă de a fi corecte.” Prezentarea intitulată ”How to design a good API and why it matters” de Joshua Bloch [Jos] începe prin a pune accentul pe importanța proiectării de API. Un API bun atrage clienți noi oferindu-le funcționalități complexe într-un mod simplu și succint. Un API prost poate rezulta în multe probleme, neclarități și pierderi pentru o companie.

Pentru un programator, API-urile ajută prin a modulariza codul. Codul împărțit în module este ușor de înțeles și modificat și poate fi refolosit.

Procesul de dezvoltare de API începe prin definirea cerințelor. În cadrul unei companii, cerințele venite de la utilizatori trebuie transformate în cazuri de utilizare. Acestea vor defini specificațiile API-ului și este indicat să fie discutate și modificate în cadrul unei echipe. Bloch recomandă construirea structurii de cerere și răspuns înainte de a implementa și chiar și de a specifica API-ul, pentru a preveni muncă inutilă în implementare și specificare.

Bloch menționează că un API trebuie să fie simplu, consistent și flexibil pentru a fi considerat un design bun. El enumeră de asemenea și niște sfaturi și tehnici standard folosite în acest proces, precum și principii importante în design-ul de API.:

- ”Un API ar trebui să facă un singur lucru”. Funcționalitatea sa ar trebui să fie ușor de explicat. Un API ar trebui să fie ușor de denumit și să aibă module care pot fi împărțite sau combinate la nevoie.

- ”Un API ar trebui să fie cât mai mic posibil”. Ar trebui să satisfacă o condiție și să nu conțină mai multe metode, clase sau parametri decât are nevoie. Greutatea conceptuală este mult mai importantă decât dimensiunea propriu-zisă.

- ”Detaliile de implementare nu ar trebui să afecteze API-ul”. Ele nu trebuie expuse utilizatorilor sub nici o măsură pentru a evita confuzia și pentru a păstra natura abstractă a unui API.
- ”Accesibilitatea ar trebui minimizată”. Clasele și membrii obiectelor ar trebui să fie private, iar clasele publice ar trebui să nu aibă câmpuri publice, cu excepția valorilor constante. Acest principiu permite modulelor să fie folosite, dezvoltate și testate independent unele față de altele.
- ”Denumirile unui API contează”. Denumirile folosite ar trebui să fie ușor de înțeles. Abrevierile ar trebui evitate și consistența și simetria între denumiri trebuie respectată pe tot parcursul implementării API-ului.
- ”Documentația contează”. Fiecare clasă, interfață, metodă, constructor, parametru etc. ar trebui să fie specificate.
- ”Deciziile de design au consecințe de performanță”. Anumite decizii de design, cum ar fi folosirea constructorilor în locul modelului factory sau folosirea tipurilor implementate în locul interfețelor pot afecta negativ performanța unui API. Cu toate acestea, performanța nu ar trebui să deformeze design-ul de bază al API-ului.
- ”Un API ar trebui să coexiste cu platforma”. Convențiile de denumire, tipurile de parametri și modelele standard de API și limbaj ar trebui respectate și urmate pe parcursul implementării API-ului.

2.2 Design-ul claselor, metodelor și exceptiilor

Clasele ar trebui să fie imutabile. Astfel, ele rămân simple și pot fi refolosite. Dacă este necesar, ele pot rămâne mutabile cu condiția de a fi folosite în mod restrâns. Subclasele ar trebui folosite cu atenție, doar când este nevoie de o relație de tip ”is a”. În restul situațiilor, compoziția ar trebui folosită, caz în care vom avea o relație de tip ”has a”. Clasele publice nu ar trebui să fie subclase ale altor clase publice(ex: Properties extends Hashtable Stack extends Vector) pentru a nu îngreuna implementarea.

Metodele ar trebui să fie conținute pe partea serverului pentru a evita codul boilerplate. API-ul ar trebui să fie predictibil și să nu producă situații neprevăzute pentru utilizator. Acest principiu se numește ”Principle of least astonishment” și este un concept vital al interfețelor. Erorile ar trebui să fie raportate și afișate de îndată ce apar. Metodele statice și generice sunt cele mai bune când vine vorba de timpi mici de compilare. Accesul la date, mai ales cele transmise sub formă de

șiruri de caractere, ar trebui să poată fi făcut programatic, cu ajutorul metodelor publice de get. Astfel, clientul nu trebuie să parseze manual șiruri de caractere. Supraîncărcarea metodelor ar trebui făcută cu grijă pentru a evita supraîncărcarea ambiguă. O soluție alternativă ar fi definirea unei metode cu denumirea diferită.

Parametrii de input ar trebui să fie cât mai specifici posibil, iar clasele ar trebui declarate ca interfețe când vine vorba de input. Dacă este posibil, șirurile de caracter ar trebui evitate datorită pierderii de performanță, iar tipul double permite precizie mărită peste tipul float datorită reprezentării pe 64 de biți. Ordinea parametrilor în metode ar trebui să fie consistentă, mai ales dacă tipul de date este identic. O metodă ar trebui să aibă cel mult 3 parametri, în special în cazul în care mulți parametri au același tip de date. Pentru a evita această situație, metoda ar putea fi împărțită în mai multe metode. Dacă este nevoie, valoarea null ar trebui înlocuită de un șir sau o colecție goală pentru a evita nevoia de a procesa situații speciale.

Excepțiile ar trebui aruncate doar în cazuri excepționale, și ar trebui aruncate de fiecare dată când apare o eroare. Ele ar trebui să includă informații și contextul în care aplicația a eșuat pentru a putea modifica codul.

2.3 Tipuri de API

Există mai multe tipuri de API-uri. Vom enumera câteva în acest capitol, urmând ca apoi să intrăm în mai multe detalii despre API-ul REST.

2.3.1 REST API

Representaional State Transfer (REST) este unul dintre cele mai populare tipuri de API. Este construit pe baza cererilor de tip HTTP, prin care poate accesa și modifica resurse precum servicii web sau informații din baze de date. Sunt ideale pentru a dezvolta aplicații mobile sau web.

2.3.2 SOAP API

API-urile Simple Object Acces Protocol (SOAP) folosesc fișiere XML pentru a transfera date într-un format web. Sunt construite pe baza unor standarde precum Web Services Description Language (WSDL) sau Universal Description, Discovery and Integration (UDDI), ceea ce oferă funcționalități și funcții de securitate mai complexe. Sunt folosite pentru aplicații de tip enterprise.

2.3.3 GraphQL API

API-urile GraphQL sunt un tip mai nou de API bazat pe limbaj query pentru a

crea cereri directe către un server sau alt furnizor de date. Datorită eficienței acestui proces, sunt ideale pentru aplicații care necesită accesul datelor rapid și eficient, dar este mai greu de implementat decât celelalte tipuri de API.

2.3.4 WebSocket

Websocket-urile permit comunicarea directă dintre un client și un server, folosind o conexiune persistentă între cele două entități. În acest fel, se evită nevoia repetării cererilor și răspunsurilor. Websocket-urile sunt folosite în principal pentru aplicații care necesită sincronizarea datelor în timp real, precum aplicații de chat sau jocuri online. De asemenea, pot fi folosite pentru a implementa notificări sau analiză real-time.

Capitolul 3

REST API

Datorită simplității sale, API-ul REST a devenit unul dintre cele mai populare tipuri de API folosite în aplicațiile moderne. REST reprezintă o metodă standard de a integra servicii web într-o aplicație și permite clienților să interacționeze cu resurse de pe server prin operații bine-definite.

3.1 Limbaje de modelare

Prin publicația sa din 2016, Vijay Surwase[Sur16] evidențiază importanța modelării unui API REST. Structura și comportamentul API-ului pot varia în funcție de modul în care este definit și ajută în alcătuirea documentației și a comunicării funcționalității aplicației.

Lucrarea științifică adresează de asemenea provocările care pot apărea în dezvoltarea API-urilor RESTful, cum ar fi definirea resurselor și relației cu restul aplicației, gestionarea stărilor și a tranzițiilor și asigurarea compatibilității. Aceste provocări pot crește atât complexitatea API-ului, precum și șansele de a apărea erori. Aceste fapte pot duce la dificultăți în menținerea și îmbunătățirea API-ului și în comunicarea funcționalității cu clientul sau alți programatori.

Pentru a putea depăși aceste provocări, Surwase enumeră limbaje de modelare care oferă modalități pentru a dezvolta și documenta API-uri REST. Acestea sunt:

- RESTful API Modeling Language (RAML): un limbaj de modelare open-source folosit pentru a defini resurse, parametri, scheme de cereri și răspunsuri și exemple. Este simplu, accesibil și ușor de folosit.
- OpenAPI Specification: cel mai popular limbaj de modelare pentru API-urile REST deoarece permite programatorilor să definească structuri, scheme de cereri și răspunsuri sau metode de autentificare, să genereze documentații, librării pentru clienți și să integreze metode de testare.

- API Blueprint: un limbaj de specificare de nivel înalt bazat pe Markdown care permite definirea documentației menită utilizatorilor într-un mod lizibil și prietenos.
- JSON Schema: folositor în principal în definirea structurii și regulilor de validare pentru API-urile RESTful bazate pe JSON. Folosit în general împreună cu OpenAPI Specification.

Fiecare limbaj vine cu avantajele și dezavantajele sale, pe care lucrarea le evidențiază comparându-le între ele.

Nume	Sponsor	Format	Open Source	Generare cod client	Generare cod server
RAML	MuleSoft	YAML	Da	Limitată	Da
API Blueprint	Apiary	Markdown	Da	Nu	Da
OpenAPI	Reverb	JSON	Da	Da	Da

Tabela 3.1: Comparație între limbaje de modelare

Deși toate limbajele sunt de tip Open Source, doar OpenAPI oferă generare de cod și pentru client.

RAML și API Blueprint sunt limbaje accesibile și ușor de integrat într-o aplicație și oferă un mod comod de a crea documentații pentru API.

OpenAPI și RAML oferă o consolă de acces care permite programatorilor să interacționeze mai ușor cu API-ul. De asemenea, sunt compatibile cu majoritatea limbajelor de programare.

OpenAPI, fiind cel mai popular limbaj, are o comunitate extinsă, ceea ce facilitează rezolvarea rapidă a problemelor.

3.2 Concepte fundamentale

Publicația lui Sanjai Patni [Pat17] începe prin a descrie funcția principală a unui API. Un API permite utilizatorilor din afara organizației proprii să acceseze serviciile și produsele necesare pentru a crea aplicații noi și pentru a își crește afacerea. API-urile interne sau private îmbunătățesc productivitatea dezvoltatorilor prin reutilizabilitatea lor. De asemenea, asigură consistența codului în mai multe aplicații. API-urile publice permit dezvoltatorilor externi să acceseze serviciile, să le îmbunătățească sau să le prezinte mai multor utilizatori.

Cele mai folosite sisteme de API sunt SOAP și REST2.2. Un sistem RESTful distribuit are următoarele calități:

- Performanță: stilul de comunicație REST este eficient și simplu, ceea ce oferă performanță crescută.
- Scalabilitate: datorită separării dintre server și client, opțiuni noi pot fi adăugate fără a încurca modul normal de funcționare al aplicației.
- Interfață simplă: permite interacțiuni simple între sisteme, ceea ce permite, printre altele, scalabilitatea crescută.
- Componente ușor de modificat: componentele pot fi modificate independent unele față de celelalte fără a le afecta.
- Portabilitate: sistemul REST poate fi implementat de orice tip de tehnologie.
- Sistem robust: sistemul REST își revine mai ușor după o eroare. Designul concentrat pe componente permite crearea de sistem rezistente la erori. Dacă una din componente eșuează, stabilitatea sistemului nu va fi afectată.
- Vizibilitate: datorită celorlalte calități, sistemele REST sunt acesibile mai multor utilizatori, iar interfețele generice permit multor dezvoltatori să lucreze cu ele.

3.2.1 SOAP vs REST

SOAP a fost creat în 1998 de Dave Wine s.a în colaborare cu Microsoft. Fiind dezvoltat de o companie software, acest protocol adresează nevoile afaceriste de pe piață. REST a fost creat în 2000 de Roy Fielding la UC, Irvine. Dezvoltat într-un mod academic, protocolul se concentrează pe Open Web. Accesul datelor se face în mod diferit în fiecare sistem. SOAP permite accesul datelor prin servicii de tip verb + substantiv. Ex: ”getUser”, ”deleteItem”, în timp ce REST permite accesul datelor prin resurse denumite ca substantive. Ex: ”user”, ”item”.

În general, arhitectura SOAP ar trebui implementată când clienții au acces direct la datele și obiectele de pe server, sau când legătura formală dintre client și server este necesară. Ar trebui evitată atunci când API-ul este destinat a fi folosit de mulți programatori fără a fi nevoie de detalii stricte de implementare sau când lățimea de bandă(bandwidth) disponibilă este limitată.

Arhitectura REST ar trebui folosită când clienții și server-ul funcționează în domeniul Web sau când informația tehnică de pe server nu trebuie expusă clientului. Ar trebui evitat atunci când este necesară o relație strictă între client și server sau când tranzacțiile folosite au nevoie de apeluri repetate.

Avantajele și dezavantajele fiecărui sistem sunt enumerate în tabela 3.2.

Topic	SOAP	REST
Avantaje	• Dezvoltat pentru a facilita afaceri și întreprinderi • Poate fi aplicat pe orice protocol de comunicare, sincronă sau asincronă • Clienții primesc informații despre obiecte • Securitatea și autorizarea sunt integrate în protocol • Poate fi specificat folosind Web Services Description Language(WSDL)	• Dezvoltat pe baza principiului Open Web • Ușor de implementat și menținut • Separă implementările pentru server și client • Comunicarea nu este gestionată de un singur obiect • Informația poate fi păstrată de client pentru a evita apeluri repetate • Poate folosi formate diferite de date (JSON, XML etc)
Dezavantaje	• Trafic mare de mesaje când vine vorba de metadata • Greu de implementat pentru aplicații web și de mobil	• Poate fi dezvoltat doar pe baza protocolului HTTP • Autorizarea și securitatea trebuie implementate manual și mai greu

Tabela 3.2: Comparație între SOAP si REST

Exemple de API-uri care folosesc SOAP sunt Salesforce SOAP API[1] și Paypal SOAP API[2]. Exemple de API-uri care folosesc REST sunt Twitter[3] și LinkedIn[4].

3.2.2 Elemente de bază

REST este construit pe baza sistemului CRUD(Create, Read, Update, Delete) HTTP cu operațiile:

- GET: pentru a căuta și întoarce date
- POST: pentru a crea date noi
- PUT: pentru a modifica date deja existente
- DELETE: pentru a elimina date
- PATCH: pentru a modifica date parțiale, cum ar fi un singur câmp al unei entități
- HEAD: identică cu GET, dar server-ul nu va întoarce un corp al mesajului. Folosită pentru a testa validitatea și accesibilitatea link-urilor

REST este o arhitectură bazată pe resurse. O resursă reprezintă o referință conceptuală, care are un identificator Uniform Resource Locator(URL) și conține atât informație descriptivă numită metadata, cât și informație funcțională despre datele dintr-o aplicație. Conținutul resursei poate fi accesat printr-una din operațiile de mai sus.

REST permite resurselor să fie reprezentate în diferite formate, cum ar fi XML, JSON sau plaintext. Reprezentarea informației este importantă datorită mecanismului HTTP de negociere a conținutului. Server-ul primește un număr de header-e prin care va ști ce tip de conținut ar trebui să poată gestiona. De asemenea, server-ul va ști și ce tip de date trebuie să întoarcă în răspuns. Dacă server-ul nu poate gestiona tipul de date cerut, va putea semnala o eroare clientului.

3.2.3 Formatarea datelor în XML și JSON

eXtensible Markup Language(XML) este un limbaj de marcare standard pentru traficul de informații pe internet. Datele sunt cuprinse între identificatori numiți tag-uri sau marcaje. O aplicație poate folosi orice denumire pentru tag-urile folosite, dar

[1]https://www.salesforce.com/developer/docs/api
[2]https://developer.paypal.com/docs/classic/api/PayPalSOAPAPIArchitecture
[3]https://dev.twitter.com
[4]https://developer.linkedin.com/apis

utilizatorii trebuie să folosească aceleași denumiri pentru a putea comunica. Structura unui mesaj XML este una imbricată(nested). Anumite tag-uri pot conține alte tag-uri, care încep și se închid în interiorul acestora.

Un tag poate avea și atribute care conțin date adiționale. De obicei, dacă informația este considerată importantă, ea face parte din elemente principale cu tag-uri proprii. Informația opțională ar trebui pusă in atribute.

Figura 3.1 conține un mesaj XML. Putem observa structura imbricată. Tagul <EmployeeData> conține două tag-uri <employee>, care la rândul lor conțin un număr de tag-uri ce descriu datele angajaților. Tag-urile <employee> au de asemenea atributul "id". Se observă și primul tag, numit prolog, care declară documentul ca fiind unul XML.

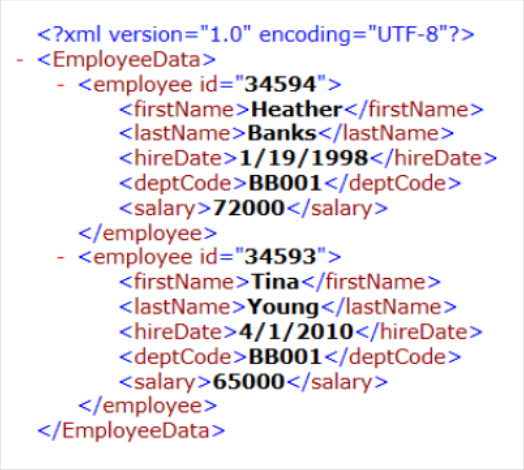

Figura 3.1: Exemplu de mesaj XML

Importantă formatului XML vine din flexibilitatea să. Oferă lizibilitate crescută prin stilul său descriptiv, un mod ușor de a identifica datele și ierarhizare prin structura imbricată. Poate fi folosit în programarea Document-driven pentru a defini interfețe și aplicații din componente deja existente sau în arhivarea datelor și a componentelor modificate.

Folosirea formatului XML vine și cu anumite costuri. Este greu de formatat, dar poate fi compresat. Crearea și procesarea datelor este un proces intensiv din punct de vedere al memoriei, iar imaginile și alte tipuri de date binare trebuie codificate.

JavaScript Object Notation(JSON) este un format bazat pe text definit cu scopul de a fi lizibil pentru oameni. Fișierele folosesc extensia .json și nu sunt dependente de limbajul în care este dezvoltată aplicația în care sunt folosite, deși limbajul este extins din JavaScript.

Sintaxa JSON se bazează pe următoarele tipuri de date:

- String: orice tip de text cuprins între ghilimele.

- Numere: numere în format zecimal.
- Obiecte: un set de perechi nume/valoare, cuprins între paranteze acolade și separate prin virgulă. Ex: { ”nume”: ”Ion”, ”ani”: 5 }
- Array: o colecție de valori cuprinse între paranteze drepte și separate prin virgulă. Ex: [{ ”id”: 1, ”nume”: ”Ion” } { ”id”: 2, ”nume”: ”Ana” }]
- Boolean: true sau false.
- Null: valoarea goală.

Importanța formatului JSON vine din faptul că este ușor de înțeles, creat și procesat, are o structură consistentă și este disponibil în majoritatea librăriilor standard. Este folosit pentru a scrie aplicații web bazate pe JavaScript, pentru a serializa și a transmite date între server și client în aplicații web. Majoritatea API-urilor publice folosesc formatul JSON pentru a expune datele cerute.

Dezavantajul principal al formatului JSON este că informația transmisă este mai greu de descris în detaliu. Ea trebuie împărțită în obiecte individiuale și nu poate fi extins în continuare.

3.2.4 OAuth 2 și Caching

OAuth 2 [5] este un standard de securitate dedicat autorizării accesului HTTP la resurse. Se bazează pe un sistem de token-uri prin care aplicațiile au drepturi de acces.

Un token este un șir de caractere generate aleatoriu de către server. Ele pot fi de două tipuri: access și refresh. Un access token permite datelor utilizatorului să fie accesate de aplicații externe. Token-ul este trimis de către client ca și parametru sau header în cererea făcută către server. În general, access token-ul nu este vizibil clientului, dar poate conține informații esențiale pentru anumite operații, cum ar fi autenticarea. Un refresh token este de obicei trimis împreună cu un access token în cazul în care acesta expiră și trebuie reînnoit de către server.

Caching reprezintă procesul de stocare temporară a informației cu scopul de a îmbunătăți performanța aplicației. Datele pot fi apoi distribuite către cereri. Metoda standard de implementare a caching-ului în aplicații web este HTTP Caching, unde resursele sunt stocate online pentru un anumit timp. Poate fi privat, unde accesul este posibil doar clienților direcți prin browser, sau public unde orice entitate asociată cu aplicația poate accesa resursele. În HTTP, caching-ul se face folosind header-ul Cache-Control, care controlează modul în care resursele sunt stocate. Pe lângă valorile public și privat, Cache-Control poate fi:

[5] https://oauth.net/2

- no-cache: resursele nu vor fi stocate
- no-store: resursele pot fi stocate temporar în memorie, dar nu vor fi stocate pe dispozitivul folosit
- no-transform: resursele pot fi stocate, dar nu vor fi modificate
- max-age: timpul maxim de validitate al resurselor

3.3 REST API cu Node.js

În publicația sa, Fernando Doglio[Dog15a] explică modul prin care un API RESTful poate fi creat folosind Node.js. El acoperă multe aspecte ale dezvoltării de API-uri, printre care principii de design, tehnici de implementare și strategii pentru deploy.

3.3.1 Best Practices

Cartea descrie un număr de practici folositoare (best practices)[Dog15b] care ajută la dezvoltarea de API-uri REST. Printre acestea se numără design-ul de Uniform Resource Identifiers(URIs) intuitive care reflectă cu acuratețe resursele pe care le reprezintă. Resursele ar trebui să fie descrise prin substantive simple pentru a evita complexitate inutilă. Doglio sugerează folosirea unei ierarhii pentru a îmbunătății utilizabilitatea și lizibilitatea API-ului

Verbele HTTP GET, POST, PUT, DELETE sunt ideale pentru a indica natura operațiilor API-ului. Mai multe detalii despre functiile lor in capitolul 3.2.2. Consistența dintre verbele HTTP și operațiile API-ului este un element esențial în dezvoltarea de API.

Denumiri	Denumiri REST
/getAllBooks	GET /books
/submitNewBook	POST /books
/updateAuthor	PUT /authors/:id
/deleteBook	DELETE /books/:id

Tabela 3.3: Comparație între denumiri aleatorii și denumiri folosind verbe HTTP

Versionarea unui API poate fi făcută folosind numere de versiuni în URI(/v1/... etc) sau antete speciale(Accept-Version header).

Gestionarea erorilor ar trebui făcută folosind coduri de status HTTP pentru a indica succesul sau eșecul operațiilor API-ului. Codurile standard sunt 200(OK), 201(Created), 400(Bad Request), 404(Not Found) și 500(Internal Server Error). Publicația

recomandă folosirea mesajelor de eroare informative pentru a ajuta atât clienții cât și programatorii în rezolvarea erorilor.

Securitatea unui API poate fi atribuită validării și sanitiză rii datelor de intrare. Acest proces poate preveni vulnerabilități comune de securiate cum ar fi SQL Injection și XSS. Validarea datelor de intrare poate fi făcută folosind expresii regulare, librării de validare predefinite sau implementarea proprie de validare. Autentificarea și autorizarea utilizatorilor este de asemenea o metodă eficientă de păstrare a securității. Cartea descrie câteva metode de autentificare, cum ar fi autentificarea pe baza de token, OAuth sau chei de API. Transmiterea informațiilor ar trebui făcută folosind protocoale securizate(HTTPS) pentru a proteja datele din a fi accesate de oricine.

Documentația API-ului ar trebui făcută în mod serios. Doglio recomandă generarea automată a documentației folosind OpenAPI Specification și pune în mod special accentul pe menținerea și actualizarea documentației pe măsură ce API-ul este dezvoltat.

Optimizarea performanței API-ului poate fi făcută folosind tehnici precum caching, comprimare și lazy loading pentru a îmbunătăți timpii de răspuns și pentru a preveni supraîncărcarea rețelei.

3.3.2 Node.js

Cartea pune accentul în mod special pe Node.js[Dog15c] că o platformă Server-Side datorită modelului său Input/Output (model I/O) bazat pe evenimente (event-driven). Node.js poate gestiona un număr mare de conexiuni și operații în mod concurent folosind un ciclu de evenimente care așteaptă cereri și activează metode care să le gestioneze. Această arhitectură permite aplicațiilor Node.js să fie scalabile și să primească cereri în mod concurent fără a bloca alte operații.

Node.js utilizează motorul de JavaScript V8, dezvoltat de Google. V8 execută cod JavaScript în mod performant, compilându-l în cod mașină, ceea ce îmbunătățește timpii de execuție in comparație cu interpretori tradiționali.

Node.js funcționează pe un sistem cu un singur fir(thread) de execuție. Spre deosebire de alte platforme server-side care creează thread-uri separate pentru fiecare cerere, Node.js folosește un singur thread pentru toate cererile primite. În combinație cu sistemul I/O non-blocant, Node.js folosește resursele în mod eficient și elimina nevoia schimbului de responsabilități între thread-uri.

Un alt element esențial în implementarea non-blocanta a Node.js este programarea asincronă. Prin rechemarea cererilor, promisiuni(Promises) și sintaxa async/await, Node.js asigura faptul că cererile gestionate de server nu blochează execuția celorlalte elemente ale aplicației.

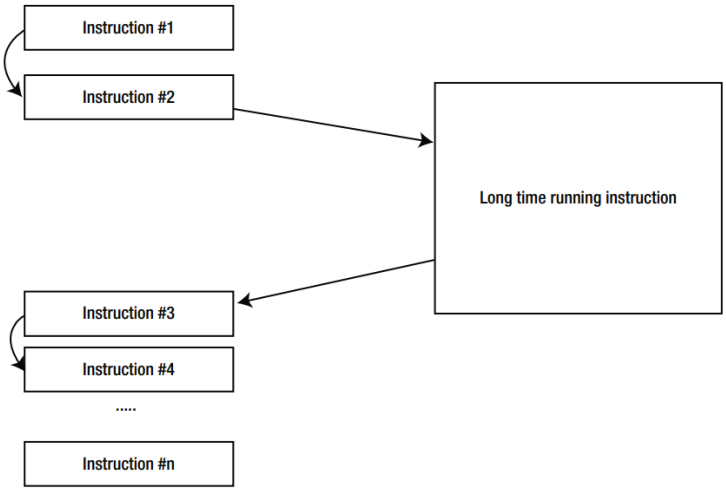

Figura 3.2: Model de programare sincronă

În figura 3.2, putem observa un set de instrucțiuni care rulează în mod sincron. Pentru a putea executa instrucțiunea 4, aplicația trebuie să aștepte terminarea blocului "Long time running instruction", apoi terminarea instrucțiunii 3. În funcție de modul de implementare și scopul fiecărei instrucțiuni, instrucțiunea 4 ar putea aștepta în mod inutil executarea unor instrucțiuni care nu au nici o legătură cu aceasta.

În figura 3.3, instrucțiunea 4 se execută imediat după terminarea instrucțiunii 2, iar blocul "Long time running instruction" și instrucțiunea 3 sunt rulate în mod asincron, separat față de celelalte instrucțiuni.

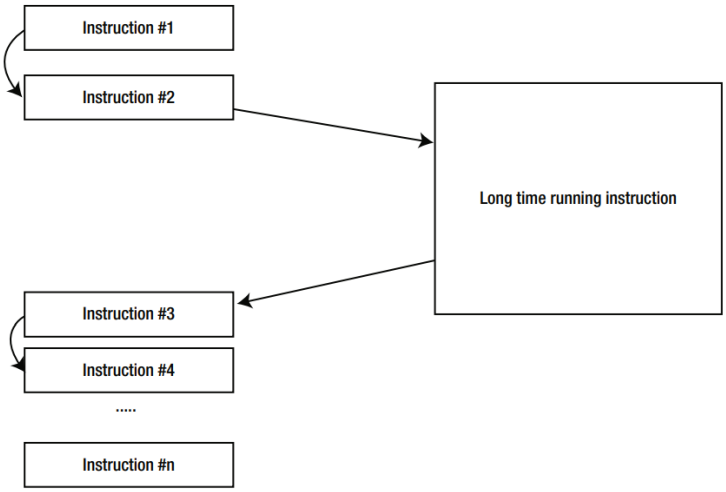

Figura 3.3: Model de programare asincronă

Doglio prezintă și Node Package Manager[6]. NPM permite programatorilor să instaleze module și librării necesare programării cu Node.js. Astfel, integrarea codului

[6]www.npmjs.org

generic este făcută fără nevoia de a rescrie elemente de baza în fiecare aplicație nouă. Datorită comunității, există un număr mare de module disponibil de descărcat. Avantajul acestui fapt este că putem găsi aproape orice modul necesar aplicației. Dezavantajul este că o mare parte din module pot face același lucru, ceea ce înseamnă că trebuie alese cu grijă.

3.3.3 Module Node.js

Un modul în Node.js reprezintă o secvență de cod separată de o aplicație care îndeplinește un anumit scop și implementează o anumită funcționalitate. Ele permit organizarea codului, refolosirea și mentenanța în aplicații Node.js.

Modulele sunt în general folosite la:

- Gestionarea cererilor și răspunsurilor HTTP: cea mai comună opțiune și care are cele mai multe module. Ambele funcționalități fac parte de obicei din același modul și permit ascultarea de trafic HTTP pe un anumit port, traducerea messajelor HTTP în obiecte Javascript, scrierea de răspunsuri HTTP indiferent de format.

- Gestionarea rutelor: una din cele mai importante funcționalități. După ce un mesaj HTTP este tradus într-un obiect Javascript, aplicația va gestiona cererea cu care este asociat printr-o rută care va transmite cererea spre codul care o va îndeplini. Rutele folosesc de obicei un șablon cu parametri care permit echivalarea URL-urilor cu codul potrivit.

- Metode de pre-procesare(Middleware): funcționalitatea de Middleware este conținută de framework-ul Connect[7] și este folosită în special la funcționalitățile de autentificare, de logare și gestionarea sesiunilor active.

- Actualizarea documentației: deja discutată în secțiunea anterioară3.3.1, dar există module care actualizează documentația în mod automat.

- Validarea cererilor și a răspunsurilor: există module care permit validarea formatului cererilor primite și răspunsurilor trimise, ceea ce asigură evitarea erorilor care pot apărea din input eronat din partea clientului sau erori de la server.

Cartea prezintă și câteva exemple de module pentru fiecare funcționalitate:

- HAPI[8]: folosit pentru a genera infrastructura unei aplicații web, HAPI poate gestiona atât cereri și răspunsuri, cât și rute.

[7] https://www.npmjs.com/package/connect
[8] http:/hapijs.com

- Express.js, Restify[9][10]: Ambele module gestionează cereri, răspunsuri și rute, dar se ocupă și de partea de middleware. Express.js oferă funcționalități ideale pentru servere HTTP, iar Restify este un modul dezvoltat pe baza Express.js pentru dezvoltarea de API-ui REST.

- I/ODocs[11]: un sistem de generare automată a documentației specifice API-urilor RESTful care folosește JSON Schema.

- JSON-Gate[12]: un modul care validează structura și conținutul obiectelor JSON folosind un model predefinit.

Pentru a accesa un modul, acesta trebuie importat folosind funcția "require". Prin această funcție, Node.js va caută modulul într-un folder local sau în folderul "node_modules". Pentru a folosi module pre-definite online, acestea trebuie instalate folosind npm, prin instrucțiunea "npm install" urmată de numele modulului.

Un modul Node.js poate fi de asemenea creat manual. Programatorul trebuie să definească și să implementeze funcționalitatea într-un fișier Javascript separat. Acest fișier trebuie apoi exportat pentru a putea fi accesat din afara modulului. Un modul poate fi exportat asignându-l obiectului "module.exports". Modulul poate fi importat în restul aplicației folosind funcția "require".

Node.js are un număr de module incorporate care oferă anumite funcționalități. Aceste module sunt disponibile în orice aplicație Node.js fără a fi nevoie de instalare în prealabil.

- ('fs'): funcții read, write folosite pentru a interacționa cu fișiere și directorii.

- 'http' și 'https': permit crearea de servere și clienți HTTP și HTTPS pentru gestionarea cererilor web.

- 'path': oferă funcții pentru lucrul cu căi de fișiere și directorii.

- 'util': funcții utilitare pentru debugging, gestionarea erorilor, formatarea și inspectarea obiectelor, funcții asincrone.

- 'url': funcții pentru parsarea și formatarea URL-urilor. Permite extragerea parametrilor și a altor componente din URL.

[9]http://expressjs.com
[10]http://mcavage.me/node-restify
[11]https://github.com/mashery/iodocs
[12]https://www.npmjs.com/package/json-gate

3.3.4 Proiectarea unui API REST

Pentru a dezvolta un API REST, acesta trebuie proiectat și plănuit cu atenție. Doglio[Dog15d] prezintă un număr de concepte cheie care ajută în acest proces.

Folosind exemplul unui lanț de librării, cartea începe cu identificarea cerinței. Pentru a putea proiecta un API, programatorul trebuie să știe cerințele clientului. În exemplul dat, este necesară o aplicație cu un sistem decentralizat, care oferă aceleași funcționalități pentru fiecare librărie. Astfel, aplicațiile client au nevoie de funcționalități precum căutări cross-site, control automat al stocului de cărți și surse dinamice de date pentru site-uri web și numere de telefon.

Odată definite cerințele aplicației, aceasta are nevoie de specificări pentru a putea determina modul în care aplicația va fi dezvoltată și pentru a putea determina erori de proiectare. În continuare, cartea definește o serie de resurse și proprietăți ale lor, cum ar fi metode de a determina titlul, autorii și genurile cărților, adresele librăriilor și un sistem de conturi pentru clienți și angajați. Acestea sunt apoi transformate în metode HTTP.

În tabela de mai jos3.4, am enumerat o serie de entități prezente în aplicația exemplu, reprezentate printr-o resursă cu nume descriptiv, un set de proprietăți și o scurtă descriere. Pe lângă proprietățile din tabel, fiecare resursă va avea atribute prezente în baza de date precum un identificator care vor fi folosite în cadrul aplicației.

Pe baza resurselor din tabelă , pot fi asignate puncte de contact(Endpoints), atribute și metode care vor fi folosite de API. Câteva exemple pentru o parte din entități sunt enumerate în tabela de mai jos3.5.

După ce sunt definite specificarile, acestea pot fi transformate într-o diagramă UML. Diagramele Unified Modeling Language (UML) sunt un mod standard de a reprezenta structura, comportamentul și relațiile prezente într-o aplicație. Diagramele pot fi de mai multe tipuri:

- Diagrame de clasă: structura statică a unui sistem cu clase, atributele, metodele și relațiile dintre ele.
- Diagrame de cazuri de utilizare: interacțiunile dintre actori (utilizatori sau sisteme externe) și sistemul care este modelat. Folosite pentru definirea functionalităților.
- Diagrame de secvență: interacțiunile dintre obiectele și componentele interne ale sistemului. Folosite pentru a evidenția ordinea cronologică a interacțiunilor.
- Diagrame de activitate: cursul activităților și acțiunilor sistemului.

Resurse	Proprietăți	Descriere
Cărți	• Titlu • Autor • Cod ISBN • Magazine • Genuri • Descriere • Recenzii • Preț	Entitatea principală și proprietățile necesare pentru a o putea identifica
Autori	• Nume • Descriere • Cărți	Resursă care are o legătură directă cu entitatea principală deoarece conține autorii cărților
Magazine	• Nume • Adresă • Număr de telefon • Angajați	Informații de bază despre magazine
Angajați	• Nume • Data nașterii • Adresă • Număr de telefon • E-mail • Data angajării	Informații de baza despre angajați

Tabela 3.4: Specificarea a câtorva dintre entitățile prezente în aplicația exemplu din cartea lui Doglio

Endpoint	Atribute	Metode	Descriere
/books	gen: folosit pentru filtrarea cărților după gen	GET	Caută și construiește o listă de cărți. Dacă parametrul ”gen” nu este gol, întoarce o listă filtrată după gen.
/books		POST	Creează o carte nouă și o adaugă în baza de date.
/books/:id		GET	Returnează o anumită carte.
/books/:id		PUT	Actualizează o anumită carte.
/books/:id/authors		GET	Returnează autorul/autorii unei anumite cărți.
/books/:id/reviews		GET	Returnează recenziile unei anumite cărți.
/authors	gen: folosit pentru filtrarea autorilor după gen	GET	Caută și construiește o listă de autori. Dacă parametrul ”gen” nu este gol, întoarce o listă filtrată după gen.
/authors		POST	Creează un autor nou și îl adaugă în baza de date.
/authors/:id		GET	Returnează un anumit autor.

Tabela 3.5: Endpoints, atribute și metode pentru o parte din entitățile specificate

- Diagrame de componente: componentele fizice și logice ale unui sistem și dependințele dintre ele. Folosite pentru a evidenția structura modulară a unui sistem.

Doglio prezintă o diagramă UML simplă care exemplifică relațiile dintre resursele aplicației.

După ce aplicația a fost proiectată și vizualizată prin diagrame, următorul pas este alegerea tehnologiilor folosite. Aplicația exemplu trebuie să gestioneze un număr de date, de unde revine nevoia unei baze de date. O bază de date ar trebui să fie ușor și rapid de dezvoltat, să poată gestiona relații între entități și să permită integrarea ușoară între entitățile sub formă de cod și elemente ale bazei de date. Cartea oferă câteva exemple: MySQL[13], PostgreSQL[14], MongoDB[15]. În final, alege MongoDB pentru aplicația exemplu datorită modului în care gestionează relațiile între entități și actualizării ușoare a structurii.

În final, trebuie alese module pentru a dezvolta sistemul RESTful. Exemplul va folosi, printre altele, Restify.

3.3.5 Arhitectura API-ului

În ultimele capitole ale cărții[Dog15e], Fernando Doglio dezvoltă aplicația exemplu până la final. Un fapt important de menționat este arhitectura MVC folosită.

Model-View-Controller(MVC) este un model de design folosit în dezvoltarea de aplicații software care oferă un mod de organizare modulară a componentelor aplicației. Este ușor de menținut și actualizat. Pentru o aplicație cu API, arhitectura MVC este compusă din 3 elemente.

Modelul reprezintă elementul central al arhitecturii. Gestionează în mod direct atât datele și entitățile din aplicație, cât și regulile de validare și actualizare a informației.

Elementul View gestionează modul în care datele sunt prezentate utilizatorului. În cazul unui API, datele sunt formatate prin răspunsuri JSON și XML.

Controller-ul este elementul care leagă Modelul de View. El acceptă cereri de la client, gestionează datele de intrare și controlează modul în care aplicația funcționează. Rolul său principal într-un API este de a procesa cererile de la clienți, de a folosi operațiile oferite de Model și de a determina răspunsul trimis către View.

Într-o aplicație cu API, modul de funcționare al unei arhitecturi MVC este:

1. Clientul trimite o cerere către API, care este interceptată de Controller.

2. Cererea este procesată și trimisă către Model, care va apela metodele necesare pentru a returna, actualiza sau șterge date.

[13]http://mysql.com
[14]http://www.postgresql.org
[15]http://www.mongodb.org

3. Datele procesate sunt trimise către View, care transformă datele într-un răspuns formatat.

4. În final, răspunsul este trimis clientului.

Arhitectura MVC permite programatorilor să își organizeze, mențină și refolosească codul. Datorită decuplării componentelor aplicației, fiecare are responsabilități bine definite și este ușor de modificat fără a afecta restul aplicației.

3.3.6 CORS

element important de securitate a API-urilor este mecanismul Cross-origin resource sharing(CORS). CORS este un mecanism bazat pe header-e HTTP care permite server-ului să indice puncte de origine cum ar fi domenii, scheme sau port-uri diferite de propriul punct de origine care au permisiunea de a încărca resurse. CORS se bazează pe un mecanism prin care un browser poate trimite o cerere inițială către server-ul care gestionează resursa pentru a verifica dacă server-ul va permite alte cereri. Prin cererea inițială, browser-ul va trimite header-e care denumesc metoda HTTP și alte header-e care vor fi folosite.

Din motive de securitate, cererile cross-origin inițiate din script-uri sunt interzise. În scopul dezvoltării de API-uri, există două situații. API-urile publice ar trebui să permită cereri din orice domeniu pentru a putea gestiona orice client dorește să folosească API-ul. API-urile private are trebui să urmeze polița CORS și să verifice dacă cererile primite respectă mecansimul CORS, validând cererea inițială trimisă de către clienți.

Capitolul 4

Partea practica

Capitolul 5

Concluzii

Concluzii ...

Bibliografie

[Dog15a] Fernando Doglio. Pro rest api development with node.js. Apress, 2015.

[Dog15b] Fernando Doglio. Pro rest api development with node.js. pages 25–47. Apress, 2015.

[Dog15c] Fernando Doglio. Pro rest api development with node.js. pages 47–65. Apress, 2015.

[Dog15d] Fernando Doglio. Pro rest api development with node.js. pages 111–123. Apress, 2015.

[Dog15e] Fernando Doglio. Pro rest api development with node.js. pages 123–175. Apress, 2015.

[Jos] Joshua Bloch. How to Design a Good API and Why it Matters.

[Pat17] Sanjay Patni. Pro restful apis: Design, build and integrate with rest, json, xml and jax-rs. Apress, 2017.

[Sur16] Vijay Surwase. Rest api modeling languages - a developer's perspective. *IJSTE - International Journal of Science Technology Engineering*, 2(10), 2016.