

**UNIVERSITATEA BABEŞ-BOLYAI CLUJ-NAPOCA
FACULTATEA DE MATEMATICĂ ȘI INFORMATICĂ
SPECIALIZAREA Informatică romană**

LUCRARE DE LICENȚĂ

BestBooking

**Conducător științific
Lector dr Bufnea Darius**

*Absolvent
Gane Alexandru-Adrian*

2023

ABSTRACT

The Internet today is one of the most widely used sources of information in any field. For this reason, the world has noticed that this scattered data can be collected, centralized and analyzed by automated process to obtain a more valuable unit of information. This diploma thesis will examine modern methodologies of automation and data collecting, which involve several branches of computer science such as data parsing and analysis, big data, web2, computing networks, and cybersecurity, which are called Web Scraping and Web Crawling. To demonstrate these practices, the topic chosen for data collection will be booking websites, which address various tactics and defensive rules to prevent this act, which will also be debated. With that said, one may ask, "Is this legal?" The answer is yes, and the ethics of these techniques will be addressed.

Cuprins

1	Introducere	1
2	Definirea problemei	4
3	Abordarea problemei	8
3.1	Analiza paginilor	8
3.1.1	<i>DevTools</i>	8
3.1.2	Alegerea selectorilor	9
3.1.3	Pagini randate <i>client-side</i> și <i>server-side</i>	10
3.1.4	Paginarea, <i>infinite scroll</i> si liste virtuale	11
3.2	Evitarea <i>blacklisting</i> -ului	14
3.2.1	Analizarea ratei de cereri	14
3.2.2	Inspectarea <i>header</i> -elor	15
3.2.3	<i>Honeypots</i>	17
3.2.4	Detectarea <i>pattern</i> -urilor	17
3.2.5	<i>CAPTCHA</i>	18
3.3	Obținerea și filtrarea fondului de date	26
3.4	Colectarea și analiza datelor	29
3.5	Optimizare	31
3.6	Termeni și condiții	32
4	BestBooking	34
4.1	Arhitectura aplicației	34
4.2	Implementare	37
4.2.1	Limbaj și instrumente	37
4.2.2	Clase ajutătoare	39
4.2.3	<i>Data mining scripts</i>	44
4.2.4	<i>LocationDecoder (legacy)</i>	50
4.2.5	Testare	55
4.2.6	Interfață	57

5 Concluzii și dezvoltari viitoare	61
Bibliografie	62

Capitolul 1

Introducere

Informația de pe internet este imensă, și este în expandare continuă, iar oameni care încercau să prelucreze aceste date nu mai puteau face față, așa că au apărut niște roboței care să automatizeze sarcinile plăcute și consumatoarea de timp. Acești roboți există de când e internetul, valoarea lor fiind recunoscută încă de la început. De exemplu Google este unul dintre cel mai mare *web crawlers* (și *web scraper*) parcurgând, toate *website-uri* pentru a le indexa și adăuga în motorul de căutare. Asta e ce face un *web crawler*, căută toate *link-urile* de pe o pagină web, le accesează și repetă procesul până la infinit într-un proces recursiv.

Acestă lucrare de licență se va axa pe ambele tehnici de *Data mining*, adică *web scraping* și *web crawling*, diferența fiind că *scraperei* monitorizează și strâng date dintr-un fond comun de *site-uri* pe care se mulează configurația și implementarea lor, pe când *crawlieri* se plimbă pe tot internetul, având un punct de start, accesând orice *link* care le-a ieșit în cale (totuși nu se recomandă să acceseze chiar orice, unele *site-uri* fiind periculoase). *Scraperei*, cel mai simplu spus, fac un *request HTTP* către un *URL* al *website-ului* și primesc înapoi sursa paginii (documentul *HTML*) sau direct *JSON-ul* pe care se construiesc componentele paginii, pe care mai apoi o parsează să extragă informația dorită și să o stocheze într-o bază de date.

Website-urile pot părea asemănătoare la suprafață, dar în componența lor pot fi foarte diferite, așa că un *scraper* bun, este definit de flexibilitatea acestuia, prin arhitectură configurabilă punând accent pe generalitate și urmând diferite şablonuri. Acest lucru este posibil printr-o analiză de aproape a elementelor și a atributelor care alcătuiesc pagina, și cum funcționează ea defapt în spate (tehnici de *reverse engineering*).

De-a lungul timpului, internetul s-a mai schimbat, și au apărut *website-urile* care sunt generate de *JavaScript* prin utilizarea diferitelor librării sau *framework-uri* (*React/Angular/JQuery*), iar *scraparii* nu mai puteau să extragă informația pe care o doresc print-un simplu *request* la sursa paginii deoarece elementele pe care le cautau nu erau în sursa paginii ci apăreau odată cu interacțiunea utilizatorului cu clientul.

Pentru această problemă se vor utiliza *web driver* care sunt ca un fel de simulare a *browser*-ului controlate prin scripturi. *Web driver*-ele sunt capabile de orice lucru care se poate face și într-un *browser* normal ca și randarea fișierelor *JavaScript*, configuratul *cookie*-urilor și a sesiunii, accesul la protocolul *DevTools* și aşa mai departe. Ele mai portă și denumirea de "*headless browser*" (*browser* fără *GUI*), iar scopul lor este pentru testarea automată a interfetelor dar rezolvă și multe probleme din *web scraping*.

Unele *website*-uri nu doresc ca datele lor să fie colectate de *scraperei* care poate, mai și creează *load* în rețea din cauza *request*-urilor, astfel au apărut mecanisme defensive de detectare a robotilor numite *CAPTCHA* (*Completely Automated Public Turing test to tell Computers and Humans Apart*) și nu numai. Au existat mai multe variante de *CAPTCHA* începând cu textele malformate, selectatul de imagini până la *reCAPTCHA v3* care este invizibil și analizează comportamentul utilizatorului pe pagină dându-i un scor de 0 la 1 pentru cât de uman pare, evoluția lor se îndrepta către un *design* cât mai *user friendly*. Desigur au evoluat și *scraperei*, pentru că este un joc de-a pisica și șoarecele între *scraperei* și detinătorii de pagini, și au reușit prin diferite tehnici mai complexe să treacă de aceste bariere. Ca să nu se ajungă la *IP ban*, scopul *scraper*-ului este să pară cât mai uman prin diferite metode: folosirea de IP-uri diferite de fiecare dată când face o cerere *HTTP* print-un *proxy pool*, schimbarea *user agent*-ul, customizarea *cookie*-urilor, aplicarea unor *delay*-uri în viteza colectarii și multe alte tehnici ce vor fi detaliate.

De ce le mai multe ori, toate aceste măsuri de ajunge la informația dorită este foarte costisitoare sau foarte lente și vor trebui luate în calcul opțiuni de optimizare. Tehnica abordată în cadrul acestui proiect va fi paralizarea *scraper*-ilor, creând *cluster* de *browsere* și un *pool* de *workeri* care vor executa sarcinile în paralel. Este foarte important la construirea *scraper*-ilor să se ia în calcul cele mai bune librării pentru a îndeplini scopul dorit.

Toate ideile precizate vor fi concretizate într-o aplicație web care v-a avea ca scop oferirea celei mai bune oferte de *booking* pentru o anumită locație, perioadă, camere și un număr de persoane și deasemnea și un istoric al prețurilor pe diferite date calendaristice luate aleator. Toate datele necesare vom extrage dintr-un *pool* de *site*-uri care au fost luate în considerare și pe o zonă specifică, în cazul acesta România, deoarece există multe hotele în toată lumea și ar dura extraordinar de mult pentru a le parcurge pe toate de mai multe ori pentru a reuși să creez un istoric al prețurilor.

Motivația pentru alegerea acestei teme a fost că în ziua de azi e foarte des auzit termenul de *botting* și este văzut ca o utilitate foarte puternică și la îndemâna oricui. Pe lângă acestea, automatizarea a devenit o necesitate în zilele noastre, iar utilizarea unor astfel de mecanisme poate aduce valoare aproape instant sau într-un timp foarte scurt, cum ar fi construirea unei baze uriașe de date care mai apoi să fie folosit

la antrenarea unei model de *AI*. Totodată mi-a plăcut că, parcurgerea strategiilor de *scraping* (*reverse engineering*) te trec prin multe aspecte mai amănunțite ale *Web2*-ului și cum funcționează acesta (în special din punct de vedere al securității), iar acțiunea de *scraping* în sine îmi provoacă un sentiment de entuziasm de fiecare dată când văd în *logg-uri* cum roboței strâng datele și le centralizează în baza de date.

Capitolul 2

Definirea problemei

Data mining-ul (se va folosii aceste termen pentru combinația dintre *web scraping* și *web crawling*) nu mai este aşa de ușor cum era acum 10-15 ani, aşa că pentru proiectarea unui astfel de robot este esențial ca de la început să stii abordarea protrivită ca să eviți multitudinea de probleme ce pot apărea pe drum. Am ales ca toate aceste tehnici și strategii să le prezint alături de proiectul pe care l-am pregătit deoarece consider că atinge majoritatea lucrurilor esențiale ale *Data mining*-ului, fiind un subiect mai practic decât teoretic.

Scopul este centralizarea ofertelor de cazare de pe mai multe *site*-uri. Sună foarte simplu, cel puțin aşa am crezut și eu l-a început, iar din această pricina astă am pierdut foarte mult timp. Pentru a putea oferi unui utilizator cele mai bune oferte mai întai trebuie să aibă posibilitatea de a-și alege destinația, perioada în care vream să meargă, numărul de oșpeti, camerele și aşa mai departe. Aceste informații sunt defapt exact aceleași informații pe care le cere orice *site* de *booking*, iar în cele mai multe cazuri vei fi întâmpinat de formular asemănător cu acesta:

A screenshot of a search form for hotel bookings. The form has three input fields: 'Unde mergeți?' (Where are you going?), 'Check-in — Check-out' (Check-in — Check-out), and '2 adulți · 0 copii · 1 cameră' (2 adults · 0 children · 1 room). To the right of these fields is a blue button labeled 'Căutare' (Search).

Figura 2.1: Formular pentru căutarea căzărilor

Un punct important în *scraping* este analiza subiectelor și găsirea şablonelor pentru a putea scrie un sigur robot configurabil și scalabil pentru toate paginile de *booking*, fiind mult mai convenabil decât să se construiască unul individual pentru fiecare. Totuși e important de știut că în unele cazuri e mult mai simplu să faci un scripturi pentru fiecare site deoarece se complică lucrurile când se încearcă generalizarea unor sisteme când ele defapt nu sunt la fel deloc. Așadar un *site* de *booking*, ca să îți ofere datele lor trebuie să oferi *input*-urile din Figura 2.1, *input*-uri care la rândul lor vor fi folosite de *scraper*. Următoarea problemă este cum se va folosii *scraper*-ul de aceste *input*-uri ca să extragă informația pe care o dorim de la *site*-uri, iar

răspunsul este că trebuie să se urmărească și să se analizeze cum fac defapt paginile această cerere și să se încerce aceași abordare. Majoritatea se folosesc de *REST API* sau *GraphQL*, iar când un utilizator completează formularul și execută butonul de căutare, cererea către *endpoint*-ul serverului așteaptă un *URL* cu *query parameters* sau un *payload* ca să poată trimite datele corecte către client.

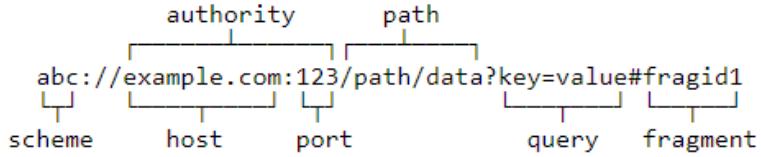


Figura 2.2: Schema generală a unui URL

În cazul acesta avem noroc că cu același set de *input* putem face *query* la orice pagină de *booking* atât timp cât construim *URL*-ul sau *payload*-ul corect (cea ce poate reprezenta o dificultate deoarece fiecare site are felul lui de a construi și formata *URL*-ul) la care *scraper*-ul să facă *request*. Deci următorul pas este analiza *URL*-ului și găsirea parametrilor esențiali ca mai departe să fie înlocuiți cu *input*-ul nostru.

```

https://www.agoda.com/ro-ro/search?city=2366&locale=ro-ro&ckuid=1e0d1f3e-fe1e-4f41-
b01b-13914d453c2e&prid=0&currency=EUR&correlationId=e8fc34ff-12c5-4551-85fb-076b752ea514
&pageType=103&realLanguageId=37&languageId=37&origin=R&cid=1&userId=1e0d1f3e-fe1e-4f41-
b01b-13914d453c2e&whitelabelId=1&loginlvl=0&storefrontId=38&currencyId=1
&currencyCode=EUR&htmlLanguage=ro-ro&cultureInfoName=ro-ro&machineName=main-76c7d57bb6-
ctn7v&trafficGroupId=4&sessionId=qwnrnxj4q5hnyhdlessv4oy&trafficSubGroupId=4&aid=130243
&useFullPageLogin=true&cctp=4&isRealUser=true&mode=production&checkIn=2022-07-12&checkOut=
2022-07-15&rooms=2&adults=4&children=5&childages=6%2C7%2C8%2C6&priceCur=EUR&ios=3
&textToSearch=Berlin&travellerType=2&familyMode=off&productType=-1
  
```

Figura 2.3: Exemplu URL *www.agoda.com*

După cum se poate vedea în Figura 2.3, un *URL* de la un astfel de *site* poate fi destul de complex aşa că se vor extrage *query parameters*:

Din Figura 2.4, elementele în chenar roșu sunt cele care ar trebui să fie înlocuite cu un set de *input* pentru a obținele datele pe care le dorim. Desigur acest set de date care se furnizează nu o să se muleze mereu perfect, de exemplu clientul aplicație mele trimită către *scraper* data în format YYYY/MM/DD, iar *site*-ul acceptă doar YYYY-MM-DD, deci pe parcurs vor fi necesare și unele formatări. O altă dificultate este că în acest set de date numele locație este în format *string*, cum se vede în chenarul galben din Figura 2.4, iar dacă încercăm să modificăm cu "New York" de exemplu, *site*-ul nu ne va returna proprietăți din New York pentru că el se va ghida după parametru "city" (primul chenar roșu din Figura 2.4), care este un ID stocat în baza de date al *site*-ului. Așadar este nevoie să se gasească o metodă prin care să obținem aceste ID-uri de hotele sau locații pentru a ușura treaba *scraper*-ului ceea ce nu este imposibil pentru că dacă clientul paginii reușește să trimită ID-ul către server înseamnă că acesta este undeva în pagină și așteaptă să fie descoperit. În capitolul 3.2 se va prezenta o metodă de *web crawling* pentru obținerea acestor identificatori unici pentru a ajunge la scopul dorit.

```

searchParams: URLSearchParams {
  'city' => '2366',
  'locale' => 'en-us',
  'ckuid' => '5ed6e5cb-8f91-4e76-acef-2149898826d9',
  'prid' => '0',
  ['currency' => 'EUR'],
  'correlationId' => '965c11cc-84b8-4d0e-bf58-b011cb5cf92',
  'pageTypeId' => '103',
  'realLanguageId' => '1',
  'languageId' => '1',
  'origin' => 'RO',
  'cid' => '-1',
  'userId' => '5ed6e5cb-8f91-4e76-acef-2149898826d9',
  'whitelabelId' => '1',
  'loginLvl' => '0',
  'storeFrontId' => '3',
  'currencyId' => '1',
  'currencyCode' => 'EUR',
  'htmlLanguage' => 'en-us',
  'cultureInfoName' => 'en-us',
  'machineName' => 'main-74ffdc4c78-bv5mt',
  'trafficGroupId' => '4',
  'sessionId' => 'zwwpsbek0a5eqbu0z3vtr2vg',
  'trafficSubGroupId' => '4',
  'aid' => '138243',
  'useFullPageLogin' => 'true',
  'cttp' => '4',
  'isRealUser' => 'true',
  'mode' => 'production',
  ['checkIn' => '2022-07-12',
  'checkOut' => '2022-07-15',
  'rooms' => '2',
  'adults' => '4',
  'children' => '5',
  'childages' => '6,7,4,8,6',
  'priceCur' => 'EUR',
  'los' => '3',
  ['textToSearch' => 'Berlin',
  'travellerType' => '2',
  'familyMode' => 'off',
  'productType' => '-1'],
}

```

Figura 2.4: Query Parameters

Odată ce avem *URL*-ul sau *payload*-ul corect construit cu *ID*-ul locației dorite inclusă, tot ce mai rămâne de facut este colectarea datelor utile care ni le oferă aceste pagini. Atributele unei cazări pe care le va căuta *scraper*-ul vor fi: prețul, moneda, *rating*-ul, numărul de stele al hotelului, numele hotelului, numarul de oaspeți, numărul de nopti, *link*-ul către pagina hotelului, facilități, locația și alte lucruri optionale cum ar fi micul de jun inclus și aşa mai departe.

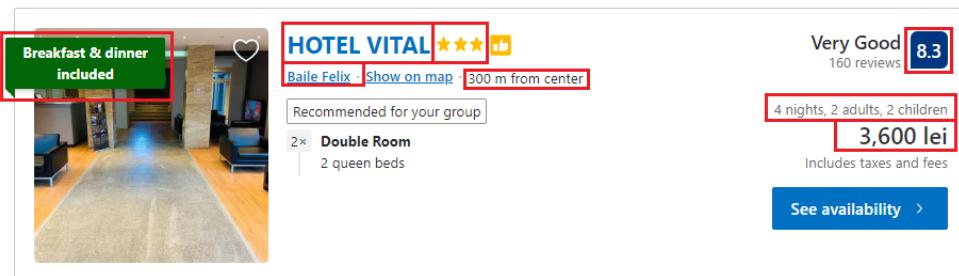


Figura 2.5: Exemplu hotel card cu datele esențiale încadrate

Toate aceste date se vor putea extrage prin analiza strukturii pagini pentru a putea obține selectori care fac referință la elementele care conțin informația dorită. În cele mai multe cazuri se va selecta lista ce conține hotelurile (din *DOM* sau răspunsul cereri care aduce lista) prin care se va itera, iar pentru fiecare pas se vor extrage datele hotelului. Posibile complexități ce pot apărea la acest pas sunt paginarea, lista poate fi virtuală (elementele care apar în structura paginii sunt doar cele vizibile în *viewport*) și că datele pe care le căutam nu apar în sursa paginii (adică ele nu apar în

HTML-ul returnat de un *request* pe *URL* construit) deoarece acestea sunt randate de JavaScript și ele apar doar prin interacțiunea unui *user* real (mai exact prin utilizarea *browser*) cu pagina. Totuși aceste mecanisme pot fi foarte lente iar informațiile pot fi foarte puține și de calitate mediocru, așa că pentru a avea acces la date care nici nu sunt afișate în pagină pentru orice utilizator, se vor utiliza metode de reconstruire a cererilor pe care clientul pagini le face către server și optinerea JSON-ului (sau a unei pagini HTML - *server-side rendering*) returnat, care mai apoi să fie parsat.

Pe lângă toate acestea sarcini evidente, mai rămân niște detalii vitale ce trebuie luate în considerarea ca toate să funcționeze corespunzător. Primul aspect, și unul dintre cele mai importante, este că pe parcursul acestor acțiuni *scraper*-ul trebuie să mențină o poziție cât mai umană ca să nu fie detectat de sistemele anti-bot ale paginilor pe care vor umbla pentru a evita un posibil *IP ban*. Există o multitudine de abordări pentru a obține acest aspect, dar cele care vor fi dezbatute în această lucrare de licență vor fi: utilizarea unui *proxy pool* pentru rotirea IP-urilor cu care *scraper*-ul o să facă *request* la pagină, ajustarea la *cookies* și *user-agent*, implementarea *scraper*-ului cu *delay*-uri aleatoare între acțiuni și folosirea unor unelte pentru mimarea cât mai apropiată a unui *click* sau *typing* uman.

Al doilea aspect care trebuie luat în considerare este optimizarea acestor procese, ele fiind foarte costisitoare. Așadar se vor aborda metode de paralelizare și comunicare între procese, utilizarea minimului de resurse necesare pentru a scădea încărcătura proiectului, crearea unui mediu favorabil în care să ruleze aplicația la potențial maxim și alegerea potrivită a uneltelor.

Cu toate acestea mai rămâne să se prezinte etica *scraping*-ului aflându-se într-o poziție "gray" din punctul de vedere a legalității.

Capitolul 3

Abordarea problemei

3.1 Analiza paginilor

Înainte de a începe orice implementare a unui *scraper* este important să se analizeze subiecții (paginile web) pentru a se creea o ideea generală și a alege abordarea potrivită. Pentru asta trebuie să se știe ce se dorește, cum se ajunge la datele dorite și cum se colectează. În cazul acesta se știe ce se vrea (cazările și prețurile acestora), iar pentru a ajunge la ele ne vom folosi de aspectul că aceste pagini folosesc *REST* sau *GraphQL* API și se va construi un *URL* sau un *payload* pe baza unui set de input ca în Figura 2.1, prin care aceștia ne vor returna informația utilă, deci un alt lucru de analizat v-a fi structura *URL*-ului și cum fac aceste pagini *request* către serverul lor, în mare se va utiliza mult *reverse engineering*. Dacă nu se pot construi aceste cereri se v-a parsa conținutul paginii indiferent dacă informația dorită e în sursa acesteia sau dacă sunt randate *client-side*. Următorul pas este colectarea datelor, iar aici se va face o analiză asupra structurii paginii, adică a elementelor *HTML* împreună atribuibile acestora pentru a putea stabili selectori prin care se face referință la informația dorită sau a încărcăturilor transmise și primite de la *API*-ul paginilor. Alte lucruri care ar trebui luate în considerare și analizate încă de la început sunt: mecanismul de paginare, *infinite scroll*, virtualizarea, *CAPTCHA*, *honeypots* și cât de mult se folosește *JavaScript* pentru randarea paginii. În această secțiune voi evidenția mai mult partea teoretică al acestor obstacole și cum se pot depista, iar în următoarele secțiuni se va prezenta cum se poate trece peste acestea.

3.1.1 *DevTools*

Toate lucrurile menționate trebuie deslușite, iar acest fapt ar fi mult mai dificil dacă nu ar exista *DevTools* (este important de menționat că fiecare *browser* are un astfel de protocol, dar în acest document se face referință la cel oferit de *Google Chrome*). *Chrome DevTools* este un set de unelte destinate dezvoltatorilor web construit direct

în browser pe care îl poti folosii pentru a edita pagina, a observat ce cererile se fac, *cookies*, *session*, *debug* și multe alte lucruri care te ajuta să construiești o pagină mai sigur și mai repede (Figura 3.1).

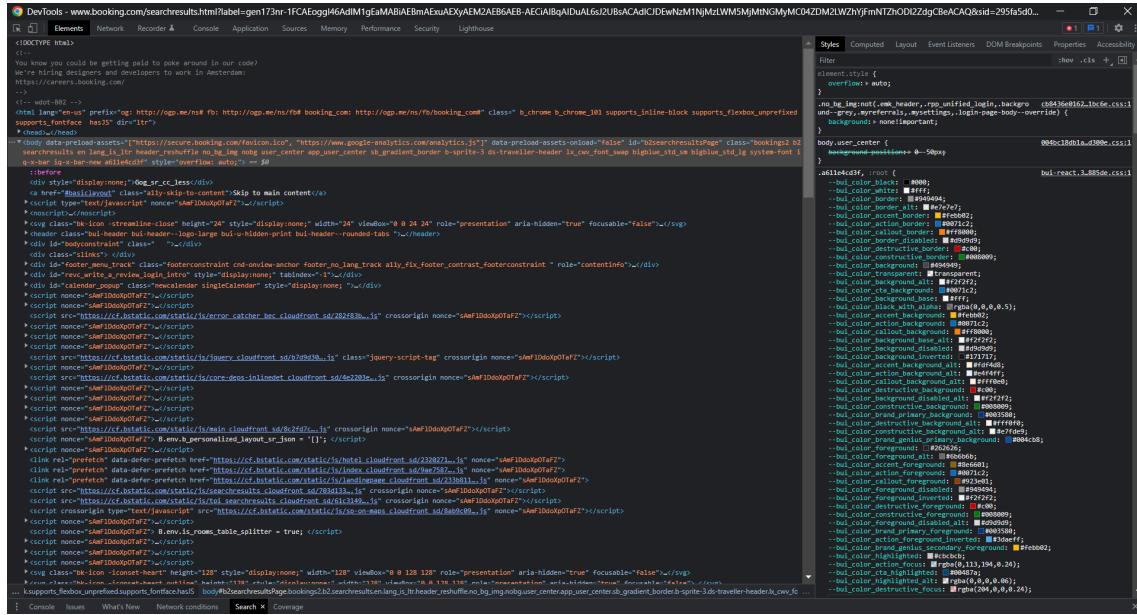


Figura 3.1: *Chrome DevTools*

Interfața *DevTools* poate fi compleșitoare cu atâtea *tab*-uri, butoane și opțiuni dar odată ce devii familiar cu ele și înțelegi ce poți face cu acestea, sarcinile pot fi completeate mai ușor și mai accelerat. Pentru *web scraping* se vor folosii în mare parte taburile *Elements* pentru inspectarea *HTML*-ului, *Network* pentru monitorizarea *request*-urilor, *Console* pentru executarea și verificarea unor selectori, *Application* pentru inspectarea de *cookies* și mai rar *tab*-ul *Source* pentru codul *JavaScript* dacă se dorește să se intre și mai adânc în subiect cum ar fi trecerea unui *reCAPTCHA V3* cu *callback*.

3.1.2 Alegerea selectorilor

Alegerea selectorilor corect este un lucru esențial deoarece prin aceștia putem ajunge, odată ce avem sursa *HTML*, direct la informația dorită. În general când vine vorba de selectori sunt defapt *CSS selectors*, adică un mecanism prin care putem selecta elemente *HTML* prin poziția lor în *DOM*, tipul lor și după atributele pe le au în componentă. O altă variantă ar fi *XPath selectors* dar aceștia au viteza un pic mai mică față de cei *CSS*, care se vor folosii pe parcusul documentului. Una din provocări este că *site*-urile își mai schimbă componența (*HTML*) și stilul (*CSS*) ocazional, iar dacă acești selectori nu sunt bine puși la punct s-ar putea ca *scaper*-ul să nu mai funcționeze. Așadar un selector bun e definit prin:

- a fi specific, să nu captureze elemente de care nu este nevoie

- a fi robust la schimbări, selectorul ar fi ideal să funcționeze și dacă se întâmplă schimbări mici în structura paginii. Dacă selectorul se strică datorită schimbărilor, e mai bine să nu returneze nimic decât informații greșite, iar astfel e și mai ușor de monitorizat
- a fi ușor de citit, în caz că lucrurile se strică, selectorii citibili ajută la evitarea rescrierii lui de la zero

3.1.3 Pagini randate *client-side* și *server-side*

Randarea client-side înseamnă că *HTML*-ul unei *website* este mai degrabă randat în *browser* decât pe serverul acestora. Deci, acum, în loc să obțineți tot conținutul din documentul *HTML*, va fi redat doar *HTML*-ul necesar cu fișierele *JS*. De obicei aceste pagini folosesc în spate *framework*-uri ca *Angular* sau librării ca *React/ JQuery*. Ca să se verifice dacă o pagina e randată *client-side* trebuie verificat *Page Source*-ul (click dreapta pe pagină iar în căsuță care apare trebuie să fie o opțiune *Page Source* sau *CTRL + U*), iar dacă elementele care se caută nu apar în sursa paginii, asta denotă că componentele sunt încărcate de scripturi *JavaScript*.

În acest caz una din variente care poate ocoli acest obstacol este folosirea *web driverelor*. *Web driver* sunt o simulare a *browser*-ului ce pot fi controlate prin scripturi. Acestea sunt capabile de executarea și randarea fișierelor *JavaScript*, gestionarea *cookie*-urilor și a sesiunilor și aşa mai departe. Ele sunt folosite în mare parte pentru testarea interfețelor aplicațiilor *web* fiind capabile să interacționeze cu pagina printr-o instanță a *browser*-ului (completarea unui formular și executarea butonului de trimis). Când vine vorba de *scraping* vom folosi *browser*-ul în modul “*headless*”, adică instanța lansată va fi fară *GUI*, sarcinile executându-se în același mod dar cu o performanță mai ridicată. Chiar și aşa, plus multe alte configurații și paralelizări acestea tind să nu atingă un timp de execuție favorabil, deci dacă obținerea datelor de pe o pagină este posibilă și fară aceste *web driver*, este recomandat să nu se folosească.

O altă opțiune mai rapidă ar fi să urmărim cererile care aduc cazările pe pagină și să le replicăm după care să le executăm direct din serverul nostru, iar răspunsul va fi un *JSON* (în cele mai multe cazuri) cu cazările aduse pe pagină, poate chiar și cu mai multe informații care nu sunt afișate pe *site*. Se va folosi *DevTools* pentru *tab*-ul de *Network* unde se vor urmări *request*-urile. Se intră pe un *site* de *booking* și se caută cazări cu un formular completat aleator. Între timp în tabul de *Network* vor apărea mai multe cereri, dar cu ajutorul obținut de căutare putem introduce numele primului hotel din listă, iar *DevTools* ne va filtra doar *request*-ul în care are ca răspuns lista de cazări (Figura 3.2). Aici se oferă informații despre cerere, fiind mai importante *header*-ele și *payload*-ul pentru a putea la rândul *scraper*-ului să facă

request la API-ul pagini ca și cum ar fi un utilizator normal.

În caz că cerearea nu returnează un JSON ci un document *HTML* înseamnă că pagina folosește randere *server-side*, dar asta nu înseamnă o problemă deoarece în loc să parsăm un JSON unde am doar fi identificat cheile valorilor pe care le dorim și atât, aici se va parsa *HTML*-ul prin diferite *tool-uri* specifice pentru limbajul folosit pentru a putea crea un obiect *DOM* din acel *HTML* și a putea executa comenzi *JavaScript* pe acesta pentru extragerea informației. Este important de menționat că această parsare a documentului *HTML* returnat de serverul paginii nu este la fel de costisitoare ca parsarea paginii prin utilizarea *web driverelor* pentru paginile randate *client-side*, deoarece răspunsul la o cerere construită de *scraper* și primirea unui răspuns este mult mai rapid decât să aștepti să se deschidă browser-ul și să mai și încarce pagina prin executarea fișierelor *JavaScript*, încărcarea imaginilor și a stilurilor aplicate pe componente.

3.1.4 Paginarea, *infinite scroll* si liste virtuale

Majoritatea *site-urilor* folosesc paginarea pentru a obține o performanță mai ridicată aducând doar o parte din datele care se doresc a fi extrase, ideal fiind ca *scraper*-ul să colecteze toate datele de pe o pagină pe un set de *input*. Totuși problema paginarii se poate rezolva relativ ușor deoarece *backend*-ul paginilor returnează datele paginat în funcție de un parametru din URL (Figuriile 3.1.4) pe baza căruia se aplică o limită și un cursor datelor returnate (sau a unei variabile din încărcătura cererii care aduce informația).

Infinite scroll funcționează ca paginarea doar că utilizatorul trebuie să dea *scroll*, iar la un anumit *checkpoint* al *scroll*-ului următorul set de date se încarcă automat sau în cel mai rău caz mai trebuie să apeși un buton la capătul paginii. În cazul acestui mecanism, *URL*-ul de obicei nu se modifică dar se face un *request* care poate fi observat în *tab*-ul de *Network* din *DevTools*. În *payload*-ul acestuia de obicei se află metadatele pentru aducerea segmentată a datelor. Un plus ce poate veni cu *infinite scroll* este virtualizarea listei, adică elementele existente din strucția paginii sunt cele care se văd pe *viewport* și puțin în afara acestuia pentru a menține o încărcare lină la derulare (Figura 3.6). În cazul acesta trebuie să interacționăm mai mult cu pagina decât cu un simplu *request* la un *URL*. În caz că nu se poate replica cererea trimisă de pagină către server, se va să folosim un *web driver* ca să putem avea acces la facilitățiile care îl oferă un *browser* printre care și *scroll*-ul. Odată ce *scraper*-ul face *scroll* se va pregăti și un *observer* ce analizează datele care apar și dispar din *DOM* și se adaugă cele noi.

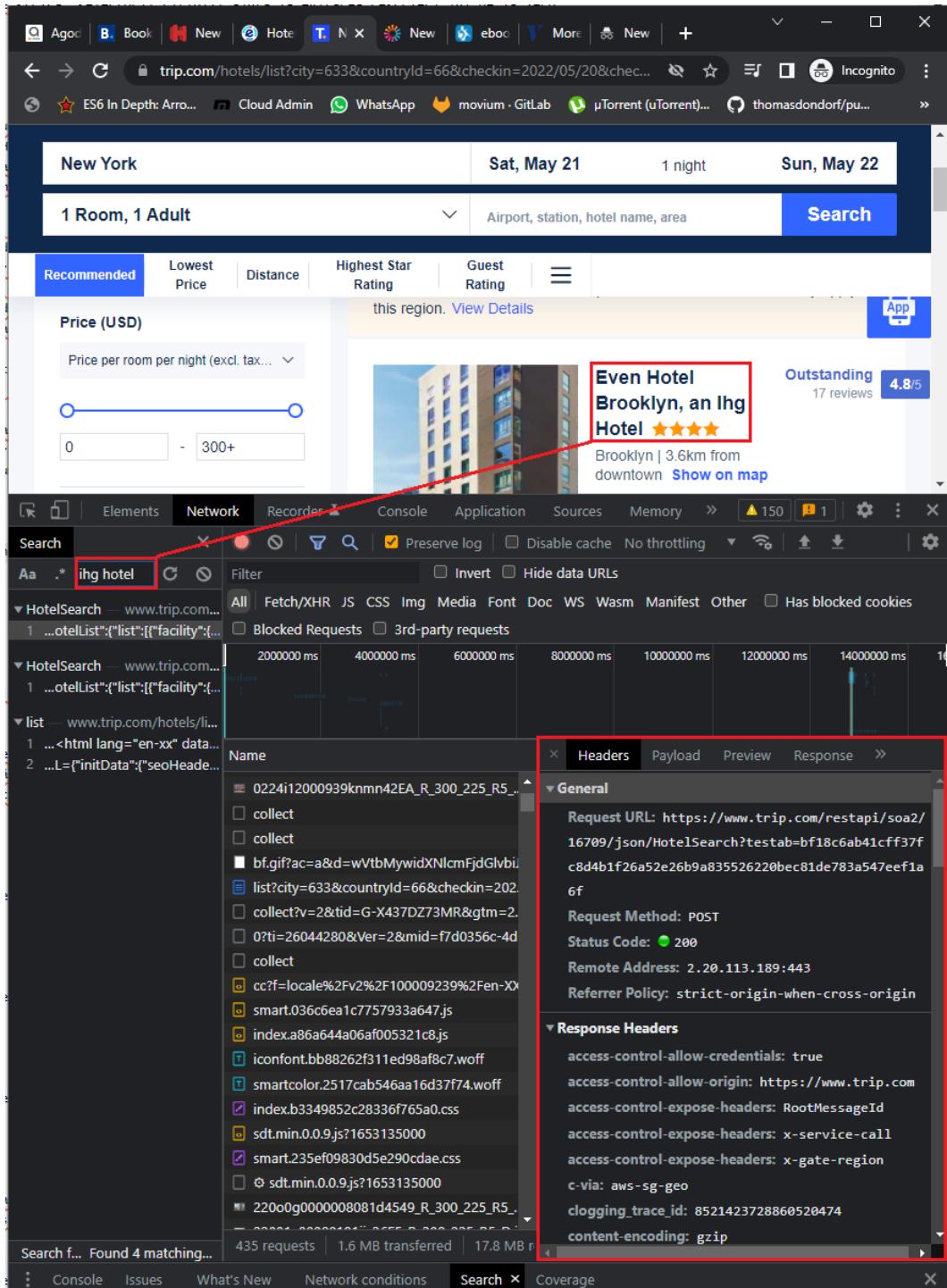


Figura 3.2: Inspectarea *header*-elor al cererilor care aduc cazările pe pagină

```
https://www.esky.ro/hot/search?arrivalCode=4509&arrivalType=region&isFlexSearch=false&page=2&rangeEndDate=2022-05-21&rangeStartDate=2022-05-21&rooms\[0\]\[adults\]=2&source=QSF&stayLength=7,7&token=3be4a324-3859-40eb-ac4c-fb8a25751d6f
```

Figura 3.3: Exemplu paginare 1

```
https://www.vrbo.com/search/keywords:upstate-new-york-new-york-united-states-of-america/page:2/arrival:2022-05-27/departure:2022-06-20/minNightlyPrice/0?filterByTotalPrice=true&petIncluded=false&ssr=true
```

Figura 3.4: Exemplu paginare 2

```
https://www.booking.com/searchresults.html?label=gen173nr-1FCAEoggI46AdIM1gEaMABiAEBmAExuAEYyAEM2AE86AEBAECIAIBqAIUaL6sJ2UBsACAdlCjDEwNz1NjMzLWM5MjMtnGMyMc04ZDM2LWZhYjFmNTZhOD12ZdgBcEACAQ&sid=295fa5d0e5c7ee0c820255e68b71db13&aid=304142&sb=1&sb\_lp=1&src=index&src\_elem=sb&error\_url=https%3A%2F%2Fwww.booking.com%2Findex.html%3Flabel%3Dgen173nr-1FCAEoggI46AdIM1gEaMABiAEBmAExuAEYyAEM2AE86AEBAECIAIBqAIUaL6sJ2UBsACAdlCjDEwNz1NjMzLWM5MjMtnGMyMc04ZDM2LWZhYjFmNTZhOD12ZdgBcEACAQ&sid=38sid%3D295fa5d0e5c7ee0c820255e68b71db13%3Bsb\_price\_type%3Dtotal%26%3B&ss=Baile+Fele%2C+Romania&ss\_area=&checkin\_year=2022&checkin\_month=5&checkin\_monthday=22&checkout\_year=2022&checkout\_month=5&checkout\_monthday=26&group\_adults=2&group\_children=2&age=6&age=5&no\_rooms=2&b\_h4u\_keep\_filters=8&from\_sf=1&dest\_id=900040016&dest\_type=city&search\_pageview\_id=6a7f3f3d626f0384&search\_selected=true&offset=25
```

Figura 3.5: Exemplu paginare 3

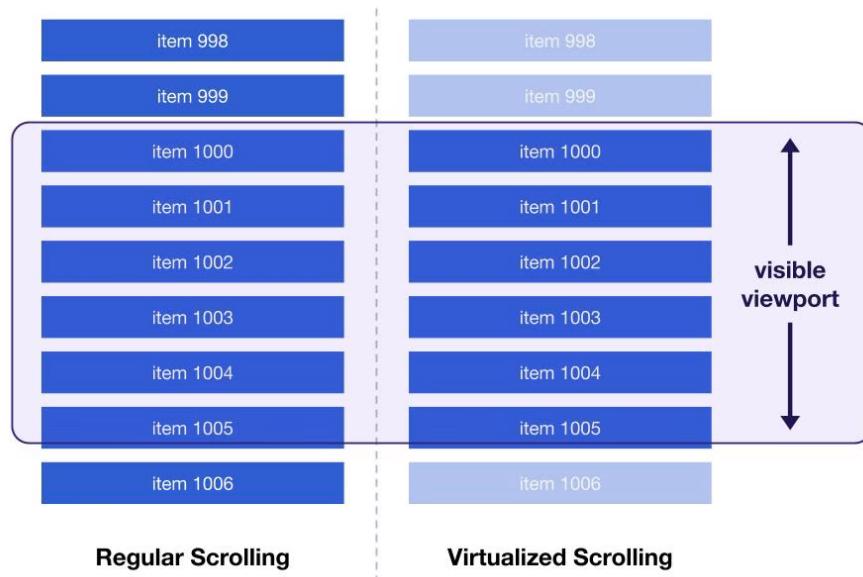


Figura 3.6: Listă virtuală

3.2 Evitarea *blacklisting*-ului

Odată ce am analizat paginile și s-a înțeles care e abordarea protrivită pentru a extrage datele, înainte ca *scraper*-ul să înceapă interacțiunea propriu zisă cu aceste pagini, trebuie ținut cont că pot exista mecanisme *anti-scraping* implementate pe partea de server ce analizează traficul de pe pagină și modelele de navigare pentru a bloca programele automate de a le mai accesa. Se vor enumara unele din cele mai întâlnite și folosite metode defensive pe care aceste pagine le aplică și cum se pot să depăși aceste obstacole.

3.2.1 Analizarea ratei de cereri

Dacă un server primește prea multe cereri într-un interval scurt de timp de la un client, poate ridica semne de întrebare și să fie detectat cu ușurință. E și mai rău dacă primește cereri în paralel de la același IP. E important să se evite repetiția când vine vorba de interacțiune cu serverele paginilor prin a trimite de exemplu cereri la timp aleator. Doar cu atât, lucrurile sunt departe de a fi rezolvate.

În adiție cea mai sigură metodă pe care se poate aplica este folosirea unui *proxy pool* din care se extrag aleator *IP*-uri prin care o să se facă cererile către pagini. Funcționează ca un intermediar căruia îi folosim identitatea cu care ne afișă și interacționăm asupra pagini (Figura 3.7). Astfel dacă rotim *IP*-uri, *scraper*-ul o să pară mai uman deoarece pare că mai mulți oameni fac *request*-uri de pe dispozitive diferite.

Pe Internet sunt mai multe pagini ce oferă liste de astfel de *proxy*-uri gratis ce pot fi la rândul lor extrase prin metode de *scraping*, deși acestea pot fi nesigure iar viteza lor poate fi foarte înceată. Este mai recomandat să se folosească servicii speciale și legitime de *rotating proxies* cum ar fi *Storm Proxies* (contra preț), în caz că se dorește ca *scraper*-ul să fie mai sigur. Desigur aceste *proxy*-uri pot fi de mai multe feluri și de calitate diferită și trebuie alese în funcție de caz. În esenta, există trei tipuri principale de *proxy*-uri:

- *Proxy*-uri publice (*data-center proxies*): Acestea sunt *proxy*-uri care sunt oferite de furnizori de servicii proxy și sunt disponibile pentru oricine să le folosească. Sunt gratuite, dar de obicei nu sunt prea fiabile sau rapide. De asemenea, sunt de multe ori supra-aglomerate, ceea ce înseamnă că pot fi blocate ușor de către *site*-uri web care își protejează datele.
- *Proxy*-uri private (*residential proxies*): Acestea sunt *proxy*-uri care sunt închiriate sau cumpărate de la un furnizor de servicii proxy. Acestea sunt mai fiabile și rapide decât *proxy*-urile publice, dar sunt și mai scumpe. Ele reprezintă defapt *device* reale cum ar fi telefoane mobile sau calculatoare conectate la rețea la care se folosesc *IP*-urile private.

- *Proxy-uri rotative (rotating proxies)*: Acestea sunt *proxy-uri* care se schimbă la intervale regulate de timp. Acest lucru este util pentru *web scraping*, deoarece *site-urile* web nu vor fi capabile să detecteze adresa IP a robotului de *web scraping*.

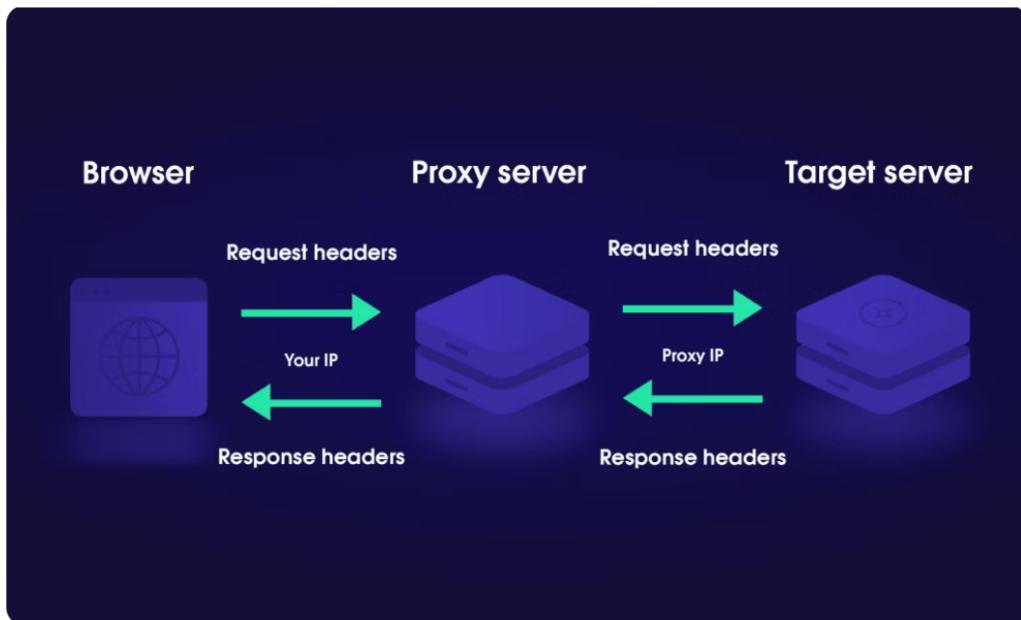


Figura 3.7: Procesul de comunicare prin intermediul *proxy-ului*

HTTP status code	What it means
503	Service unavailable
429	Too many requests
403	Forbidden

Figura 3.8: Răspuns ce poate semnala *blacklisting* din parte a serverului

3.2.2 Inspectarea *header-elor*

Când vine vorba de trimitera cererilor, serverele tend să inspecteze și *header-ele* cererilor pentru a putea detecta dacă cineva care le accesează *site-ul* e om sau nu. Ideea e că acestea se așteaptă că *header-ele* primite să fie asematoare cu cele care

ar fi trimise de pe clientul acestora. Se vor analiza și optimiza cinci headere mai cunoscute pentru ca să scraperul treacă neobservabil:

Header	Example value
HTTP header User-Agent	Mozilla/5.0 (X11; Linux x86_64; rv:12.0) Gecko/20100101 Firefox/12.0
HTTP header Accept-Language	en-US
HTTP header Accept-Encoding	gzip, deflate
HTTP headers Accept	text/html
HTTP header Referer	http://www.google.com/

Figura 3.9: Cele cinci headere ce vor analizate

Header-ul User-Agent este folosit pentru a transmite informații cum ar fi tipul *browser-ului*, sistemul de operare, *software-ul* și versiunea, permitând *browser-ului* să decidă ce fel de tip al prezentării *HTML* (mobil, tabletă, etc.) să folosească în răspuns. Autentificarea *header-ului User-Agent* al unei cereri este o practică larg răspândită în rândul web serverelor și poate fi primul semn prin care pot identifica activitate suspicioasă. De exemplu, în timpul procesului de *scraping*, numeroase cereri trec prin serverul paginilor și dacă *header-ul User-Agent* al cererilor este identic, poate părea că o activează unui *bot*. Este important să se creeze un mecanism care să rotească și manipuleze aleator un *pool* de astfel de headere, care va permite portretizarea a mai multor sesiuni organice ale utilizatorilor. Aplicând această metodă va reduce substantial şansele ca *scraper-ul* să fie blocat mai ales în combinație cu *rotating proxies*.

Header-ul Accept-Language este procesat de către web server pentru a indica ce limbă înțelege clientul și ce limbă particulară e preferată când acesta întoarce răspunsul înapoi. De obicei nationalizarea se face prin parametri al *URL-ului* paginii dar când acestia lipsesc atunci aceste *headere* le înlocuiesc scopul. Neglijând acest *header* și utilizat în combinații gresite cum ar fi trimiterea mai multor cereri cu aceeași locația a IP-ului dar cu limbi diferite poate ridica suspiciuni din partea paginilor și într-un final să blocheze *scraper-ul*.

Header-ul Accept-Encoding notifică web serverul ce tip de algoritm de compresare să folosească pentru a rezolva cererea. Când acest *header* este folosit și optimizat permite scăderea volumului de date, adică clientul primește informația dorită (comprimată), iar serverul nu mai irosește resurse încercând să trimită o încărcătură uriasă

de trafic.

Header-ul Accept este un mod de a spune serverului ce tip de format al datelor poate fi returnat către client. De ce le mai multe ori acesta este omis dar dacă este configurat adevarat poate rezulta într-o comunicare mult mai naturală între client și server, prin urmare, scăzând și sansa unui posibil *IP ban*.

Și nu în ultimul rând, *header-ul Referer* specifică adresa pagini web vizitată anterior înainte de trimiterea cererii la server. Poate parea că *Referer* are impact mic când vine vorba de blocare procesului de *scraping*, când, de fapt poate face diferență în unele momente. Așadar, dacă se doresc o portretizare și mai umană a *scraper-ului* se pot specifica *site-uri* aleator în acest header, ca și cum un *user* real navighează imensul internet.

Cele menționate anterior sunt în general cele la care trebuie acordată mai multă atenție, dar *browser-ul* tinde să trimită un alt set de *headere* de identificare a sistemului a celui care face cererea și acele fiind cele *Sec-Ua* care dacă sunt incluse trebuie să fie unu la unu cu cele menționate mai sus. Deobicei acestea sunt ignorate de serverul paginii web.

Pe lângă aceste *headere* generale pe care le setează *browser-ul*, paginile web trimit și *headere* specifice ce pot conține sesiunea utilizatorului sau alte informații necesare ca cererea să funcționeze și trebuie neapărat incluse și acestea în compoziția cererii. Printre acestea se află și *cookie-ul* care este una dintre principalele componente de identificare a utilizatorului folosite de *site-uri* și este important ca acesta să poată fi replicat în funcție de unde se face cererea (adică să corespundă cu combinația de *headere* și specificațiilor *proxy-ului* folosit).

3.2.3 *Honeypots*

Un *honeypot* este un mecanism de securitate cibernetică prin care se creează o țintă falsă pentru a atrage infractorii cibernetici către acesta și a-i ține departe de cea reală. Este o capcană care poate aduna informații despre identitatea, metodele și motivele capturi. Când vine vorba de paginile web, acestea pot amplasa astfel de capcane sub formă de *link-uri* în *DOM*, nefiind vizibile pentru un utilizator normal pe *browser* (au proprietatea *CSS display: none*). Dacă cumva *scraper* atinge și face o cerere la aceste *link-uri* parsând *HTML-ul*, serverul web îl va detecta imediat și va bloca comunicarea. Astfel de capcane se pot evita prin alegerea potrivită a selectorilor în așa fel încât să nu cuprindă mai multe elemente decât cele dorite.

3.2.4 Detectarea *pattern-urilor*

Un robot tinde să navigheze *site-urile* într-un mod prea ordonat și perfect față de un om, cum ar fi că intervalele de timp între acțiuni mereu sunt aceeași, locația clicurilor

are exact același coordonate mereu și să mai depare. Aceste *pattern*-uri pot fi detectate de mecanisme *anti-scraping* pe partea serverului, iar robotul riscă să ajungă pe lista neagră. O modalitate foarte simplă pentru combaterea acestui obstacol este prin înlocuirea constantelor folosite în acțiunile *scraper*-ului cu valori aleatoare dintr-un interval care satisfac sarcina în mod asemănător, astfel adăugându-se încă o picătură de esență umană.

3.2.5 CAPTCHA

Un test *CAPTCHA* (*Completely Automated Public Turing test to tell Computers and Humans Apart*) este proiectat să determine dacă un utilizator pe web e real/uman sau bot. Acestea sunt una din cele mai populare tehnici *anti-scraping* implementante de deținătorii *site*-urilor, deși abordarea poate avea și dezavantajele sale. Acest test poate să vină sub mai multe forme și tipuri, ele evoluând în direcția să fie cât mai "*user friendly*" deoarece primele sale variante puteau fi enervante câteodată.

CAPTCHA-urile clasice, care încă pot fi găsite pe unele *site*-uri, sunt cele care provocați utilizatorul să identifice litere și cifre distorsionate generate aleator (Figura 3.10). Ideea era că programele pe calculator ca roboti erau incapabile să interpreteze imaginea distorsionată, pe când oameni erau obișnuiți să vadă variații de interpretări de litere în tot felul de contexte, fonturi și scrieri de mâna diferite. Una din metodele de rezolvare era folosirea unui *OCR* (*Optical Character Recognition*). *OCR*, reprezintă translatarea mecanică sau electronică a imaginilor cu scris de mâna, tipărit sau printat în text editabil. *OCR* este un domeniu de cercetare în recunoașterea modelelor, inteligență artificială și vederea mecanică. Acestea sunt utile deoarece pot consuma orice imagine din care să returneze textul identificat din ele având acuratețe destul de mare. Sunt multe unele *open-source* care ajută furnizarea unui punct de start solid ca *TESSERACT* sau *GOCR*. Oricum această abordare poate fi destul de costisitoare și sunt alte abordări mai moderne ca *machine learning* care ar fi mult mai echitabile dacă ar fi nevoie să se rezolve un astfel de *CAPTCHA*, de exemplu acestea au și variante audio ce se pot descărca și decifra mai cu ușurință de astfel de algoritm. Există o multitudine de variante pentru rezolvarea unui astfel de *CAPTCHA* dar nu se va intra în detaliile de implementarea al acestor metode deoarece cel mai bun mod de a face față unui *CAPTCHA* este să faci tot posibilul să îl eviți.

reCAPTCHA este un serviciu oferit de *Google* pentru a înlocui tradiționalul *CAPTCHA*. Această tehnologie a fost dezvoltată de niște cercetători de la universitatea *Mellon Carnegie*, apoi a fost cumpărată de *Google* în 2009. De-a lungul timpului, *Google* a extins funcționalitatea testelor *reCAPTCHA*, astfel încât acestea să nu mai fie nevoie să se bazeze pe vechiul stil de identificare a textului neclar sau distorsionat.

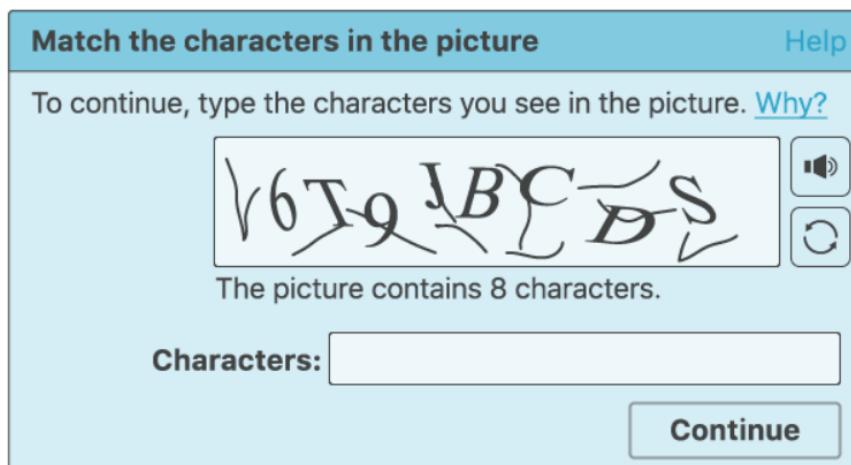


Figura 3.10: CAPTCHA clasic cu tip de provocare text distorsionat

Alte tipuri de teste *reCAPTCHA* includ:

- recunoșterea imaginilor
- casetă de bifat
- evaluare generală a comportamentului utilizatorului (fără interacțiune cu utilizatorul)

Pentru un test *reCAPTCHA* de tip recunoșterea imaginilor, provocarea constă într-o imagine reală împărțită în mai multe patrătele mai mici. Utilizatorul trebuie să aleagă acele patrătele care consideră că conțin subiectul cerut de test (Figura 3.11). Alegerea anumitor obiecte din fotografiile neclare este o problemă greu de rezolvat de către calculatoare. Chiar și programele avansate de inteligență artificială pentru recunoașterea imaginilor au dificultăți pe acest subiect – aşa că și un bot se va lupta din greu cu o astfel de provocare. Cu toate acestea, un utilizator uman ar trebui să poată face acest lucru destul de ușor, deoarece oamenii sunt obișnuiți să percepă obiectele de zi cu zi în tot felul de contexte și situații.

Unele teste *reCAPTCHA* (*reCAPTCHA V2*) solicită pur și simplu utilizatorului să bifeze o casetă de lângă declarația „*I'm not a robot*” (Figura 3.12). Cu toate acestea, testul nu constă în acțiunea propriu zisă de a face *click* pe caseta de selectare – este tot ceea ce duce la *click* pe caseta de selectare. Acest test *reCAPTCHA* ia în considerare mișcarea cursorului utilizatorului pe măsură ce se apropie de caseta de selectare. Chiar și cea mai directă mișcare a unui om are un anumit grad de aleatorie la nivel microscopic: mișcări minusculе inconștiente pe care robotii nu le pot imita cu ușurință. Dacă mișcarea cursorului conține o parte din această imprevizibilitate, atunci testul decide că utilizatorul este probabil legitim. De asemenea, *reCAPTCHA*

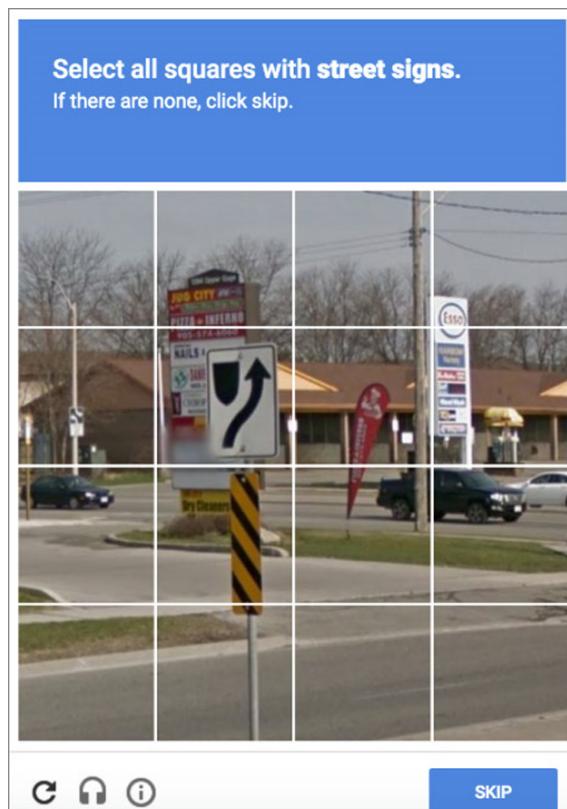


Figura 3.11: *reCAPTCHA* cu tip de provocare recunoșterea imaginilor

poate evalua *cookie*-urile stocate de *browser* pe un dispozitiv al utilizatorului și istoricul dispozitivului pentru a spune dacă este probabil ca utilizatorul să fie un bot. Dacă testul încă nu poate determina dacă utilizatorul este un om sau nu, poate prezenta o provocare suplimentară, cum ar fi testul de recunoaștere a imaginii descris mai sus. Cu toate acestea, de cele mai multe ori mișcările cursorului utilizatorului, *cookie*-urile și istoricul dispozitivului sunt suficient de concluzioane.

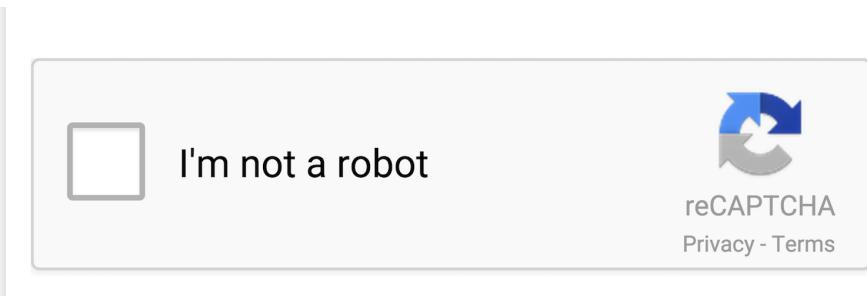


Figura 3.12: *reCAPTCHA* cu tip de provocare casetă de bifat

Cele mai recente versiuni ale *reCAPTCHA* numite și "reCAPTCHA invizibil" sau *reCAPTCHA V3* sunt capabile să arunce o privire holistică asupra comportamentului și istoricului de interacțiune al unui utilizator cu conținutul de pe Internet. De cele mai multe ori, programul poate decide pe baza acestor factori dacă utilizatorul este sau nu un bot, fără a oferi utilizatorului o provocare de finalizat. Dacă nu, atunci utilizatorul va primi o provocare tipică *reCAPTCHA*.

Un *CAPTCHA* poate cu ușurie să strice un *scraper* configurațat odată ce se află în procesul de extracție, astă că tratarea acestora este destul de esențială pentru *web scraping*. Totuși, cum spuneam, cea mai bună metodă de a face față unui *CAPTCHA* este să încerci să eviți o posibilă declașare a acestuia în primă fază prin toate metodele enumerate în acest subcapitol, unele mai importante fiind:

- Încetinirea *scraping*-ului pentru a face comportamentele sale mai puțin asemănătoare unui robot
- Utilizarea serverelor *proxy* pentru a minimiza urmărirea *IP*
- Evitarea *honeypots*-urilor

În cazul în care chiar este nevoie de confruntare directă cu aceste teste anti-bot există servicii populare care rezolvă orice fel de *CAPTCHA*-uri prin API-ul lor, cum ar fi *2captcha* sau *Anti-Captcha* (Figura 3.13). Aceste servicii utilizează astă numitele "ferme de *CAPTCHA*", adică oameni reali, angajați să rezolve *CAPTCHA*-urile în timp real. Desigur serviciile de rezolvare a *CAPTCHA*-urilor nu sunt gratis însă sunt destul de ieftine (1000 de *reCAPTCHA V3 Enterprise* rezolvate costă aproximativ 2 dolari).

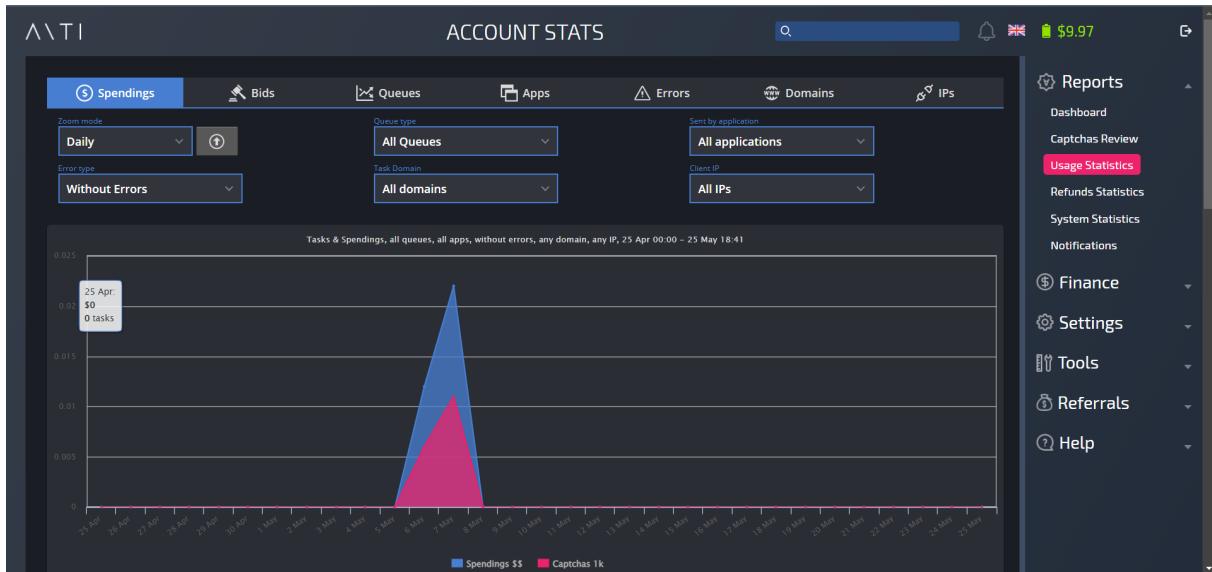


Figura 3.13: Interfața oferită de *Anti-Captcha* pentru monitorizarea CAPTCHA-urilor rezolvate

Acestea servicii oferă metode de integrare a API-ului în general pentru mai multe limbaje de programare. Autentificarea la API se face printr-o cheie ce se generează odată cu crearea unui cont pe pagina lor. Odată autenticat, se pot utiliza funcții speciale, câte una pentru fiecare tip CAPTCHA, care așteaptă ca parametru cheia publică (*site-key*) care este utilizată pentru a invoca serviciul de *reCAPTCHA* și returnează soluția sub forma unui cod (răspuns returnat de CAPTCHA care atestă că testul a fost rezolvat cu succes) care mai apoi trebuie injectat în pagină.

Este important să se știe ce CAPTCHA/*reCAPTCHA* folosește pagina pentru a reuși să se identifice, în primul rând, funcția care trebuie apelată din cele oferite de API-ul de soluționare CAPTCHA (în caz că nu se specifică funcția adecvată CAPTCHA-ului impus, rezolvarea nu va putea fi găsită). În al doilea rând, trebuie să se cunoscă locația unde poate fi găsit *site-key*-ul și unde trebuie introdus răspunsul, deși și aici trebuie avut grijă deoarece există mai multe feluri de a invoca serviciul de *reCAPTCHA*.

Prima categorie de implementare și cea mai clasnică, atașează răspunsul cu dovada rezolvării testului CAPTCHA ca o valoare a unui element de tip *input* din DOM ce are ca atribut *id*-ul "g-recaptcha-response" sau "g-recaptcha-response-100000" (Figura 3.15). Răspunsul cu rezolvarea o să fie injectat de *scraper* în acel *input* din DOM înainte să completeze și finalizeze formularul, astfel trecând peste test. Rezolvarea va fi aprovisionată de un API de rezolvarea CAPTCHA-urilor ales, odată ce i se furnizează *site-key*-ul testului care se dorește trecut. Cheia publică a CAPTCHA-ului sau *site-key*-ul este atașat de un elemente HTML care în mare majoritate a cazurilor are ca atribut clasa "g-recaptcha". Acest element HTML poate reprezenta *container*-ul pentru caseta de bifat „*I'm not a robot*” (*reCAPTCHA V2*, 3.14) sau butonul pe care

se apăsa pentru finalizarea formularului (*reCAPTCHA V3*). În caz că cheia care se caută nu apare în *DOM* se pot urmării cererile ce au ca domeniu *www.google.com* de la încărcarea paginii, iar una din ele va conține în încărcătura sa *site-key*-ul căutat (Figura 3.16).

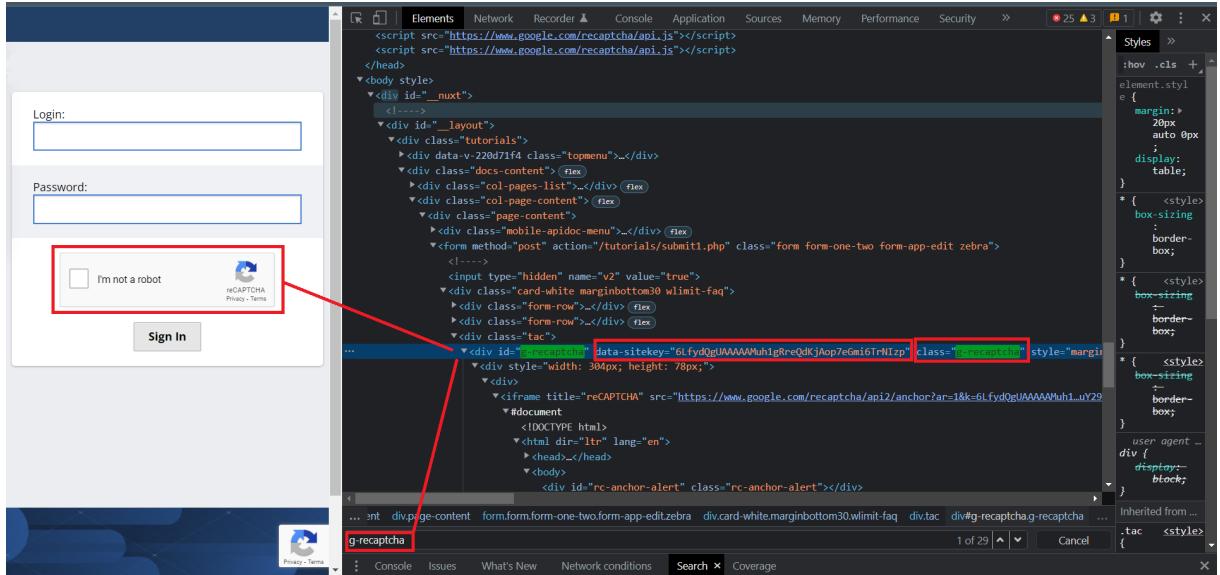


Figura 3.14: Metodă optinere a elementului cu atributul *site-key* pentru *reCAPTCHA* V2 implementat clasic

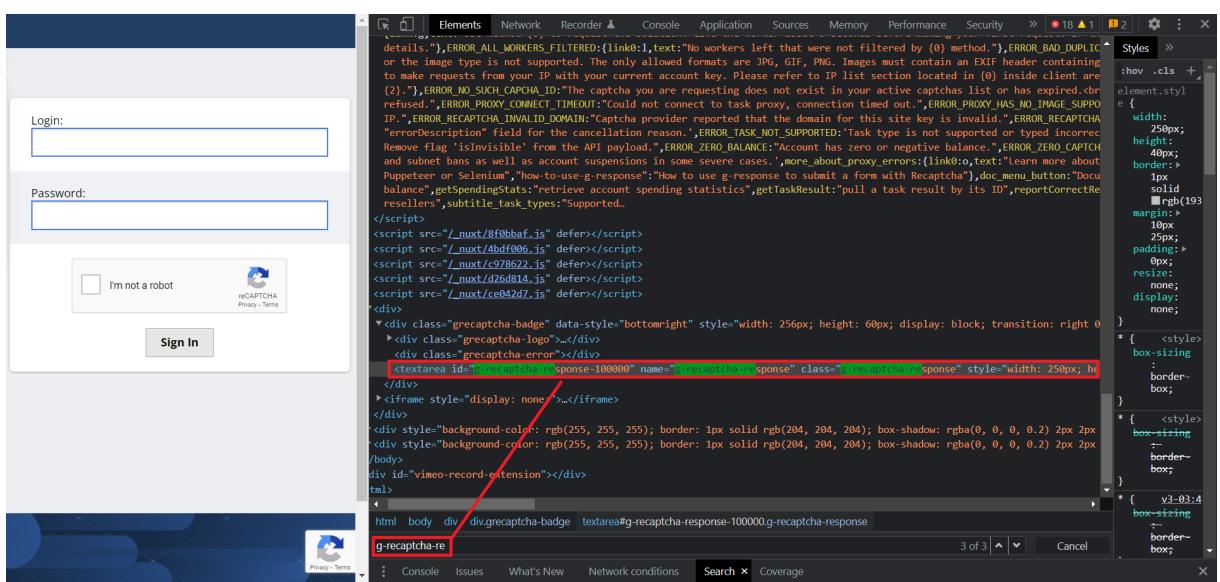


Figura 3.15: Metodă optinere a elementului cu atributul *g-recaptcha-response* pentru *reCAPTCHA* V2/V3 implementat clasic

A două categorie are implementant o funcție *callback* atașată formularului care este executată când *reCAPTCHA*-ul este rezolvat. Se poate verifica dacă un site folosește astfel de mecanism prin *DevTools* cu *Search*-ul general. Se caută cuvintele cheie "*grecaptcha.render*" sau "*grecaptcha.execute*" (documentația Google *reCAPTCHA*

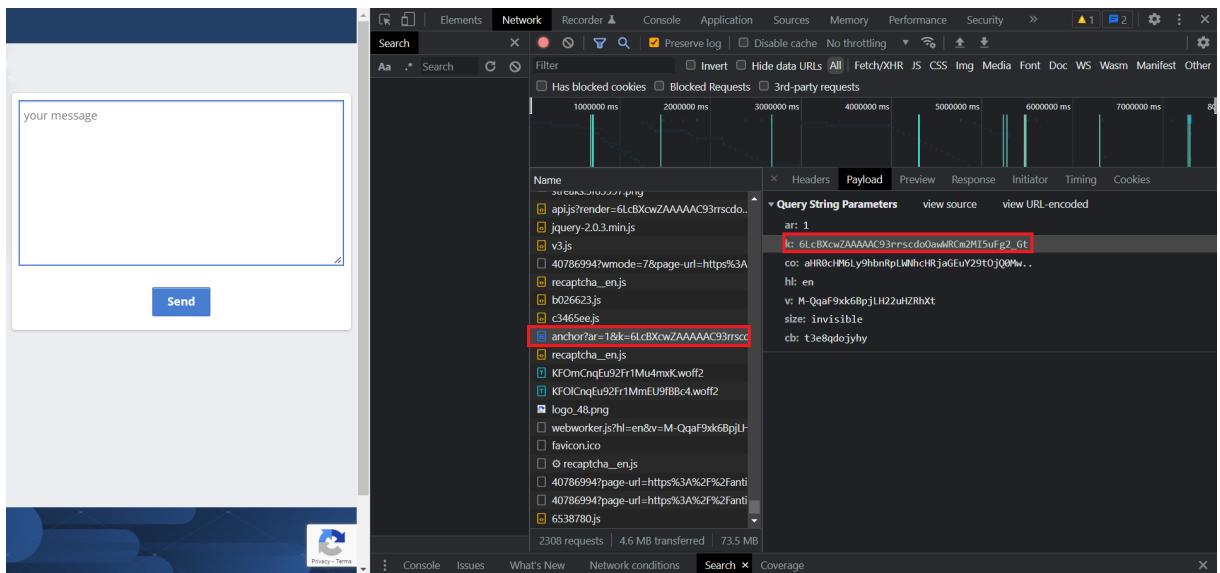


Figura 3.16: Metodă optinere a cereri ce are în ce poate avea în componentă *site-key-ul*

cu *callback*), iar funcția găsită va avea ca parametru optiunea cu numele funcție de *callback* (Figura 3.17). Se poate folosi mai departe numele acestei funcții ca *scraper*-ul să o apeleze cu cheia returnată de API de soluționare a CAPTCHA-urilor. Cheia pentru direcționarea către CAPTCHA în contextul acesta poate fi aflat ca în Figurile 3.15 și 3.16. Astfel formularul se poate completa și finaliza fără nici o problemă de către bot. Dificultăți majore ce pot interveni în acestă situație sunt precum obfuscarea codului JavaScript și anomizarea funcției de *callback*. Pentru asta se va folosi optiunea de *Debug* oferită de *DevTools* pentru a urmări cererea care se execută când se finalizează formularul (Figura 3.18 chenarul verde). Codul obfuscat este puțin greu de citit dar nu imposibil și se vor putea identifica cu ajutorul cuvintelor cheie, elementele necesare. De exemplu în Figura 3.18 în chenarele roșii se pot identifica argumentele transmise la *submit* și că apare și *site-key-ul*. De aici se știe că funcția anonimă se apelează cu *site-key-ul*, deci se va putea extrage funcția dorită (chenarul albastru din Figura 3.18). Având acestă funcție se va putea reține într-o variabilă la nivel de *root* și să se apeleze ca înainte. Desigur acest cod modificat va trebui injectat în codul aplicației web prin interceptarea cererii care aduce fișierul care are funcția anonimă. Odată interceptat se va opri propagarea celui original și pasarea variantei cu funcția stocată. După care se apelează funcția exact cum s-a precizat și la varianta cu numele *callback*-ului definit.

CAPTCHA-urile pot fi o durere de cap, dar cu fiecare generație a sa, apar și câte o generație nouă de roboți. Acestea se pot învinge cu apariția și creșterea instrumentelor de *scraping* și a serviciilor ce oferă soluții CAPTCHA, aşadar procesul poate continua fără piedici.

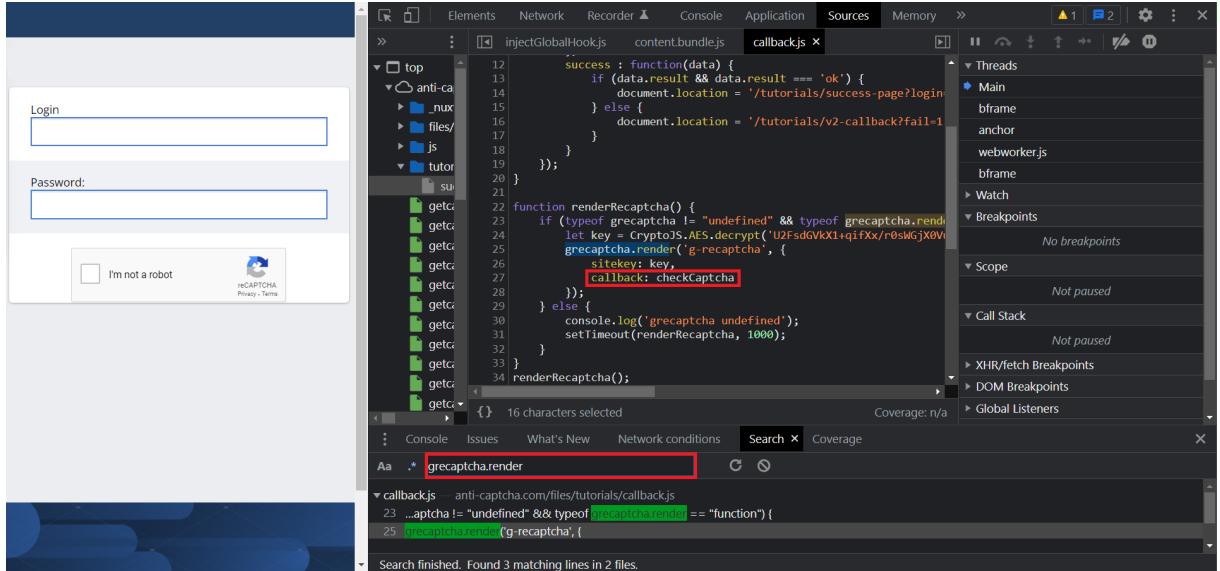


Figura 3.17: Metodă optinere a cereri ce are în ce poate avea în componentă site-key- ul

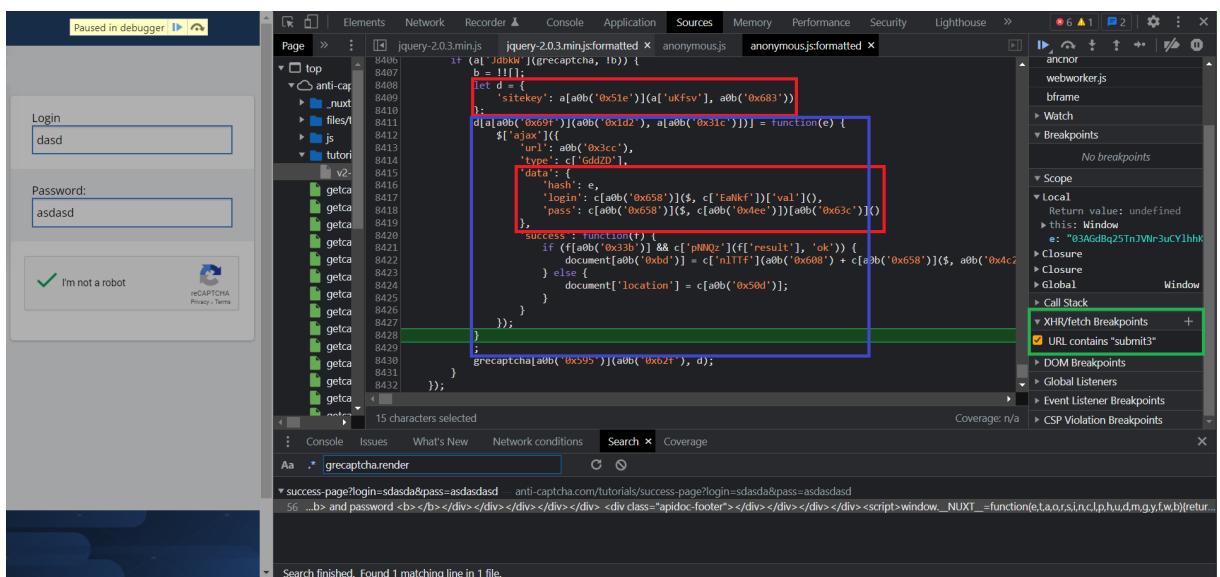


Figura 3.18: Metodă optinere a numelui funcției de callback utilizată de CAPTCHA

3.3 Obținerea și filtrarea fondului de date

Până acum s-au prezentat doar aspecte legate de *scraping* când s-a ajuns la pagina dorită, dar acum trebuie luat în calcul cum se poate ajunge la aceea pagină sau pe ce *link-uri* ar trebui să lucreze *scraperi*. Pentru acestă problema se vor utiliza tehnici de *web crawling* pentru a extrage și a filtra de pe un *website* articolele sau produsele de la care se doresc să se extragă date. Se dorește ca ofertele/hotelurile să fie aduse și în timp real la utilizatorii pe un anumit set de *input* ales de ei (Figura 3.19), așa că am decis că cea mai optimă variantă este să se construiască URL-ul (sau *payload-ul*) către fiecare pagină dintr-un *pool*, cu integrarea setului de *input* în *query parameters*. Dacă avem *URL-ul* (și *ID-ul*) tot ce mai rămâne de făcut e colectarea datelor și confrutarea cu obstacole ca pagini randate *client-side* și așa mai departe. Dificultatea care intervine în acest caz este că în setul de date, numele locației este de tip *string*, adică numele locație literă cu literă și în cel mai bun caz opțiunea oferită de *API-ul Google Maps* pentru autocompleierea locație, iar majoritatea serverelor de *booking* așteptă în *query* un *ID* pentru locație (Figura 2.4), care îl au stocat în baza lor de date unde îl referențiază. Este nevoie de acel *ID* pentru construirea URL-ului sau a *query-ului* din cerere dinamic. M-am gândit la mai multe abordări la optimizarea *ID-ului* și voi evidenția trei metode pe care le-am încercat.

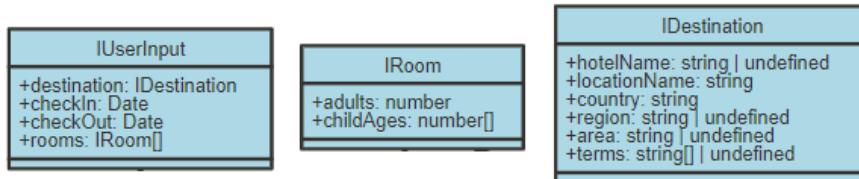


Figura 3.19: Set de *input* pe baza căruia se vor construi dinamic *URL-urile* către căzările de pe fiecare *site*

Prima metodă și cea care a fost utilizată era bazată pe o tehnică de *web crawling* și aceea de a obține *link-urile* către aceste pagini. Locul cel mai bun unde se pot găsi toate *link-urile* la un loc și categorizate este în *sitemap-ul* paginii (dacă nu au *sitemap* este posibil să găsiți *link* către toate datele și în alte locuri ca exemplu *footer-ul* paginii). Aceste *sitemap-uri* sunt folosite de motoarele de căutare pentru a indexa paginile ca să fie mai ușor găsite de utilizatori (tehnici *SEO*). Ele pot fi găsite de obicei în paginile de anunț, în fisierul *robots.txt* (fisierul *robots.txt* este un fișier cu reguli concrete despre cum să fie colectate datele de pe paginilor dacă este permis Figura 3.20) sau cu ajutorul unor instrumente *online*. Desigur pentru acest proiect, după cum am menționat, se doresc doar căzările din Romania așa că aceste *link-uri* vor trebui filtrate. La unele *site-uri* a fost ușor deoarece, în *link-urile* acestora se află deja o cale care precizează țara din care face parte cazarea. La alte *site-uri* nu există această categorizare

iar pentru asta s-a construită niște scripturi care pe baza unui ID a unei țări folosite de un *site* de căzări (extras manual prin executarea formularului pe o anumită țară și extragerea ID-ului prin *DevTools* cu ajutorul *tab-ului Network* prin urmărirea pachetelor care erau trimise la server) reușea să extragă toate hotelurile, în special la acest pas *link-ul* către acestea. Dacă nici una dintre aceste variante nu funcționa pentru un anumit *site*, atunci pe baza unor date deja colectate care conțineau toate orașele sau județele din România, s-a reușit categorizarea *link-urilor* prin comparația acestora cu baza de date a locațiilor deja colectate. Acum că s-a rezolvat prima problemă, mai rămâne a două și aceea de a afla ID-ul trimis de clientul paginilor către server. Cum toate *site-urile* diferă în modul de abordare și în implementarea lor, la fel difereau și metodele de extragere. La unele era simplu pentru că ID-ul se afla deja în URL-ul paginii căzării, dar la altele nu apărea nici unde dar totuși clientul reușea să extragă de undeva ID-ul și să îl trimită mai apoi la server, deci acesta era ascuns undeva în pagină. Majoritatea erau ascunse în obiecte JS de pe window (generate de scripturi JS) sau direct în scripturi JS care așteptau să fie executate odată ce utilizatorul voia să caute oferte pentru aceea căzare. Acestea puteau fi extrase și parsate cu ușurință cu ajutorul *web driver-elor*, iar această parte va fi evidențiată în capitolul ce conține implementarea. Astfel lucrurile devin mai sigure deoarece se știe ce se parcurge și cât trebuie parcurs.

A doua variantă a fost să implementez un fel de *web crawler* care era defapt mai mult un *scraper* care accesa mai multe *URL-uri* construit dinamic. S-a observat că aceste *ID-uri* pe care le foloseau paginile puteau fi cuprinse într-un interval mai mare dar asta nu însemna că toate erau valide. De exemplu se observă că locații valide sunt de la numărul 9120004 până la 10000000, ceea ce înseamnă un număr destul de mare de iterații pentru un *scraper*. Intervalul e fictiv, dar poate fi destul de mare având în vedere că sunt multe locații pe Pământ. Deci am încercat să paralelez niște *scraperei* să facă cerere către *URL-urile* construite pe baza intervalelor definite și să extragă titlul din *HTML* returnat deoarece acesta conținea numele locație. Cu implementarea descrisă un *worker* reușea să obțină aproximativ cinci *ID-uri* într-o secundă. Astfel optineam o bază de date cu toate numele locațiilor legate de *ID-uri* pe care le folosește o anumită pagină. Abordarea asta chiar dacă ar fi funcționat are câteva dezavantaje:

- risc ridică de *blacklist* în caz că nu se încetinește *scraper* și se iau măsurile necăsare precum rotirea de *proxy-uri*
- intervalele câteodată erau greu de definit sau fără sens, iar dacă se încerca mărirea intervalului pentru a cuprinde cât mai multe date, timpul de execuție creștea destul de mult

- un astfel de program dacă nu era rulat într-un mediu corespunzător se poate bloca sau strica în timp ce se extrag datele

Ideea nu era rea, abordarea era gresită. O metodă mai bună ar fi fost să se construiască un *crawler* real care să analizeze *sitemap*-urile oferite de paginile de *booking* ca în prima variantă.

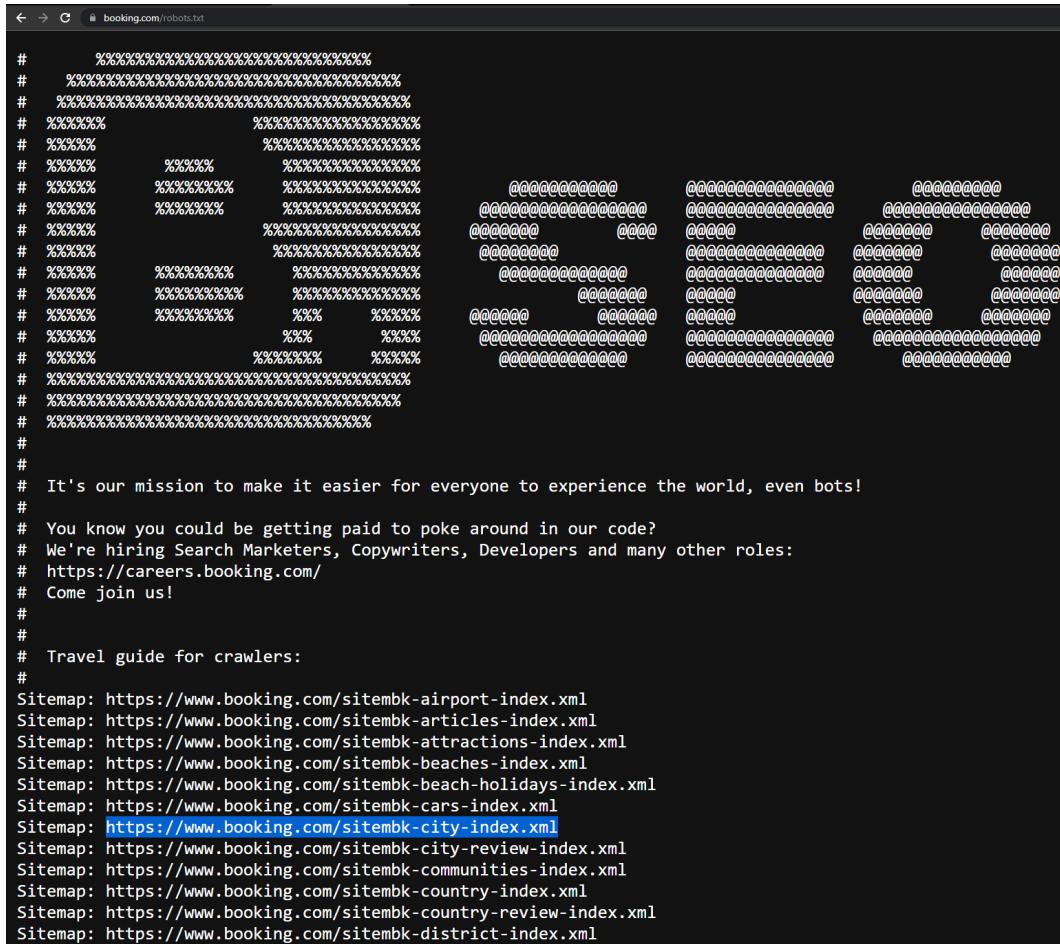


Figura 3.20: Fisierul *robots.txt* de pe pagina www.booking.com

În legătură cu problema ridicată de datele *real time*, influnțate de mai mulți parametri cum ar fi intervalul de timp a sejurului, se revine de unde a plecat, adică URL-urile construite dinamic și obținerea pe loc a *ID*-ului locației dorite. Așadar a treia metodă este completarea și trimiterea formularului din Figura 2.1 cu numele locației din setul de *input* pentru fiecare pagină pentru obținerea URL-ului. Aceste URL-urile vor conține *ID*-ul căutat, iar restul parametrilor vor fi înlocuiți sau adăugați din restul de *input*. Pentru a interacționa la un nivel atât de apropiat cu pagina se vor folosi *web drivere* și instrumente construite peste acestea. *Scraper*-ul va funcționa ca un test pentru formularul paginii respective care va introduce numele locației și va acționa asupra butonului de căutare. Cum toate paginile de acest fel utilizează aproximativ aceasi structură pentru formularul de căutare, se va putea

crea un bot general pentru un site de booking, care va lua anumite decizii bazat pe o configurație. Astfel se va putea obține structura *URL*-ului cu *ID*-ul locației integrat, plus alte bonusuri ca *cookie*-uri redate ca *query parameters*. Una din cele mai mari dezavantaje la această abordare e viteza care crește odată cu numărul *site*-urilor întâmpinări. Alte obstacole majore ce pot interveni sunt formularele protejate de *reCAPTCHA* și alte lucruri care vor ieși în evidență în capitolul de implementare 3.2.2.

3.4 Colectarea și analiza datelor

Tot ce mai rămâne de făcut este colectarea, centralizarea, parsarea și analiza datelor. Acest *scraper* va face cereri către *URL*-urile construite și va extrage seturi de date cu atribute ca în Figura 2.5 și le va stoca într-o bază de date. Aceste date se vor extrage în trei moduri diferite în funcție de cum e pagina construită.

Dacă datele pe care le căutam sunt în sursa paginii înseamnă că putem face o cerere la acel *URL* pentru a descărca *HTML*-ul de unde se vor extrage datele prin selectori și parsări. În majoritatea cazurilor se va selecta containerul/lista care conține *card*-urile cu hotelurile pe care o să se itereze. Cazul acesta este mai rar pentru că majoritatea aplicațiilor sunt mai moderne, iar datele se aduc prin executarea unui script *JavaScript* care trimite cererea, deci acestea sunt randate *client-side*.

A doua metodă pentru extragerea datelor se aplică în caz că paginile sunt randate *client-side*. Se vor folosi iarăși *web driver*-ele pentru a soluționa mai multe obstacole ce vin pe langă randarea *client-side* cum ar fi *infinite scroll* sau liste virtuale. Astfel se va instația un *headless browser* care o să acționeze ca în prima metoda doar că de data asta *HTML*-ul pe care se lucrează o să fie luat direct din *browser* adică va activa și anumite cereri ce îi va schimba *DOM*-ul, ca și cum un *user* uman ar naviga pagina.

Mai există o metodă pentru paginile care sunt randate *client-side* și acea de a recreea cererea pe care o trimite clientul/pagina către serverele web. Această metodă se recomandă a fi utilizată în asemenea contexte deoarece e mult mai rapidă și eficientă decât a două variantă. Pe lângă asta datele returnate de *API* pot fi mai bogate în date și mai frumos structurate. Paginarea și infinite scroll pot fi rezolvate prin setarea protrivită a parametrului care aduce elementele paginate. Cum se poate vedea în Figura 3.22, s-a reușit să se replice cererea care se trimite de clientul paginii către server cu ajutorul *URL*-ului construit dinamic și a *header*-lor extrase (Figura 3.21). În unele cazuri s-ar putea ca serverul să trimită un *HTML* înloc de un *JSON*. Asta înseamnă că pagina utilizează și randarea *server-side*, iar elementele *HTML* sunt randate pe server și trimise către client. Asta nu e problemă, va trebui făcut un pas în plus și acela de parsarea documentului *HTML* din răspuns (ca în prima metodă) pentru extragerea datelor.

```

Request Headers
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9
Accept-Encoding: gzip, deflate, br
Accept-Language: en-US,en;q=0.9
Connection: keep-alive
Cookie: _pxhdz=xmmsk1bmzbJx2fMm30lpc2lnjuza2eqsrUQ8159algsvFF500XsnVtaAopt5y5zemssx0w8]HfYUHCpk11Lg303D3AKr1MwUqNchj-zxe1ayhWntq41i0T-1x3gbwAx1BTkv2oAz3aWpUSDyxVmSc-czwPflm1lvh2Dfz2FQ-01Ftp%2Fk
YquigjkrKrp0pf883Ec0; B2S=-; bkng_sso_session=e30; bkng_frontend_sesse_exp=1; pxcts=f9a58a9e-ddbb-11ec-8d94-506772a4c4b; pxvid=f44c8a7c-7ddb-11ec-8d95-71484e765271; px_f394g17vmc43dfg_user_id=zMrhy2f1
YzATZGdky1exMMjUN3Wm0t00U0tuxWmztaZ; bkng_sso_sees=ey1b29ra6n5x0ds23hctfa0siact16me3mdzdg9wcytV1NHTJH3dnltutbe10Rcm0gx17RBk0kdFuUtg1LcHtjoxfv19; px_3=94cf0d8bae622e78e8sfF22346564eec9a5c
45707646be8ee5f881339922f;r=jwRa2z3SSMllwqklqazV7+elg8h0Q77hdZkn4Ep30vEPvICsgwNe1fzQs1j1987ISGdj0OKLcHtYrg-1080:MoLcY2m0ffES1M1kvSu42T+UdNEEXEQ5gn+9tRWSpb0eC1sdrtbq0sC1lwB7tm+3Q45j2f6tu7uwt
bmwsuXoX4idZbhnewyyzoCQjyAvpnufrayujlQLCw7H02jKdAbDsyLyvysvqGMWkCTm02a3X0DrWk515RgdfAmv2C8497YL2NA02312b2hre7U39h9PQ=-; bkng_110mfU2039SVyk221lyhvtaaz2925xyQ0Lbprfyc470hBbDT19R04D1fH7vXZI
udvmt225CtQgdntzyePMu0lPl4My0G31f6uoxnub1bz71bg7712e22BaxNLzouuWV1S2zF4cdKvT2k2f1z76bcgkjpuz2fNz8WbDui1fxjeedgMarNVEN26587MLvG1GzK0; has_preloaded=1; lastseen=; pxdet-a8e3d42f64
fdaf3fd2d7a9778bf5151b9754dec4ed3fe43csabed0d27ad/eyj0eN1c3RhbxAl0jE2NT0Nj4K151OT1simZf2z10jAstm1v19pCIGw119; optanonConsent=ImplicitConsentCountry=GDPR&ImplicitConsentDate=1653669739768&optCen
abled=1&datestamp=Fri+May+27+2022+1923AM3ZAS5+07:00+0300+(Eastern+European+Summer+Time)&version=6.22.0&isIA8Global=false&hosts=&consentId=45da3f0a-51be-469e-8e9a-0920681a137&interactionCount=1&landingPa
th=https%3A%2F%2Fwww.booking.com%2Fgroups%3E001k3A12C000223A0%2CC000403A0
Host: www.booking.com
Referer: https://www.booking.com/
sec-ch-ua: " Not A;Brand";v="99", "Chromium";v="102", "Google Chrome";v="102"
sec-ch-ua-mobile: ?0
sec-ch-ua-platform: "Windows"
Sec-Fetch-Dest: document
Sec-Fetch-Mode: navigate
Sec-Fetch-Site: same-origin
Sec-Fetch-User: ?1
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/102.0.0.0 Safari/537.36

```

Figura 3.21: Header-ele cereri care aduce elementele utile pe pagină

Header	Value
Sec-Fetch-Site	same-origin
Sec-Fetch-User	?1
Upgrade-Insecure-Requests	1
User-Agent	Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/102.0.0.0 Safari/537.36
sec-ch-ua	" Not A;Brand";v="99", "Chromium";v="102", "Google ...
sec-ch-ua-mobile	?0
sec-ch-ua-platform	"Windows"

Figura 3.22: Exemplu de extragere a datelor dintr-un URL cu *query parameters* și header-ele setate asemănător cu cel trimis de clientul paginii *www.booking.com*

După ce s-au strâns datele de la fiecare pagină, în primă fază o să fie salvate în baza de date. Ca să se returneze elementele pe loc, o să se caute același hotel de la pagini diferite și o să se aleagă cel cu prețul cel mai mic. Așadar pe clientul pagini mele o să se afișeze toate hotele găsite, dar dacă sunt duplicate se va afișa doar cel cu prețul cel mai mic cu o referință către *link-ul* paginii care are oferta. Pe lângă aceste date *real time* stocate, o să mai fie *scrapere* care lucrează în *background* pe un *pool* de seturi de date, abordând fiecare *site* în acești manieră ca *scrapere* care lucrează pe loc.

O analiză a datelor se va face și pe toată baza de date adunată la momentul actual pentru a reuși să se facă un grafic pentru fiecare hotel în legătura de cum a fluctuat prețul pe o durată mai lungă de timp.

3.5 Optimizare

Când vine vorba de optimizare, primul lucru care trebuie luat în calcul este abordarea protivită. Abordarea potrivită înseamnă că de exemplu la un site, care are elementele dorite în sursa paginii putem să facem direct o cerere pentru *HTML*, în loc să se folosească un *web driver*, care pierde timp pe deschiderea instației de *Chromium*, încărcarea cu cereri ca stilul, fontul sau imaginile paginii și aşa mai departe, care durează foarte mult pentru nimic în plus, comparativ cu o simplă cerere. Deci e important ca de la început să alegi metoda care te conduce la soluția dorită dar și care e cea mai rapidă.

Paralelizare

Odată ce s-a ales varianta optimă, mai pot exista micro detalii interne ce se pot lua în considerare pentru o viteză și mai mare. Toate metodele precizate sunt în general rapide, în afară de cele care folosesc *web driver*-ele ca implementare. În acest caz trebuie analizat exact ce se dorește ca să se poată impiedica aducerea elementelor inutile care încarcă traficul. Mai multe cazuri practice de optimizare o să fie evidențiate în capitolul de implementare deoarece sunt destul de strâns legate de instrumentele folosite, dar o metoda care ar putea crește viteza internă a metodelor ce utilizează *web drivere*, este prinderea și renunțarea la cererile care aduc stilul, fontul și imaginile și alte metadate nefolositoare pe pagină. Se mai pot specifica diferite configurații prin care se renunță la caracteristici ale *Chrome*-ului de care *scraper*-ul nu are nevoie, astfel va mai crește viteza acestuia.

Din exterior se pot grupa și aplica strategii de paralizare a *scraper*-ilor care utilizează același algoritm doar bazat pe o configurație diferită. Așadar, *scaperi* folosiți pentru decodarea locației pe baza de *web drivere* vor fi adăugați într-un *cluster* unde fiecare *worker* va instantia un *Chromium* și vor lucra în paralel. Pentru *scraperei* care lu-

crează pe colectarea datelor se va crea un *pool* de *workeri* care în funcție de configurațiile paginii vor putea avea trei tipuri de *job*-uri și acelea sunt cele menționate în capitolul anterior.

Indecși

O altă optimizare în cazul acesta când intervin milioane de date este recomandat crearea unor indecsi. Indecsii sunt folosiți în bazele de date pentru a accelera interogările și a îmbunătăți performanța. Aceștia sunt structuri de date speciale, care permit bazei de date să găsească rapid informațiile căutate într-o tabelă sau într-o bază de date mare.

Când o interogare SQL este emisă pentru a căuta informații într-o bază de date, motorul de bază de date trebuie să caute toate înregistrările care îndeplinesc criteriile de căutare. În cazul unui tabel mare, acest proces poate fi extrem de lent, deoarece motorul de bază de date trebuie să scaneze toate înregistrările. Acesta este motivul pentru care se folosesc indecsii.

Indecsii sunt creați pe baza uneia sau mai multor coloane ale unei tabele, în funcție de modul în care se face căutarea și interogarea datelor. De exemplu, un index ar putea fi creat pe coloana "hotelname" pentru tabele "Hotels" deoarece în practică ele se vor căuta după nume, nu după ID, astfel se va accelera căutarea unui hotel specific în tabelă. Atunci când se emite o interogare către baza de date pentru a căuta un anumit hotel, motorul de bază de date va căuta mai întâi în indexul creat, pentru a găsi rapid înregistrarea respectivă. Astfel, se reduc timpul și efortul necesare pentru găsirea informațiilor.

Este important de menționat că crearea de indecsi poate afecta și performanța de inserare, actualizare sau ștergere a datelor, deoarece aceste operațiuni trebuie să actualizeze și indexul. În plus, prea mulți indecsi pot afecta performanța bazei de date și pot ocupa mai mult spațiu pe disc. De aceea, este important să se gândească bine la crearea indecsilor și să se optimizeze performanța bazei de date în funcție de nevoile specifice ale aplicatiei.

3.6 Termeni și condiții

Cu privire la *web scraping* și *crawling*, există câteva aspecte legale importante de luat în considerare, în special în ceea ce privește partea "gray" sau neclară.

În primul rând, este important de menționat că *web scraping* și *crawling*-ul pot fi ilegale dacă sunt efectuate fără permisiunea proprietarului *site*-ului web. În acest sens, termenii și condițiile *site*-ului pot specifica faptul că orice activitate de *crawling* sau *scraping* este interzisă și poate fi considerată o încălcare a drepturilor de autor.

Pe de altă parte, există situații în care *web scraping* și *crawling*-ul sunt considerate legale. De exemplu, dacă site-ul web publică informații publice sau disponibile gratuit, atunci este probabil că activitatea de *scraping* sau *crawling* este permisă. De asemenea, în cazul în care informațiile sunt utilizate în scopuri legale, cum ar fi cercetarea academică sau jurnalistică, *web scraping* și *crawling*-ul pot fi considerate legale.

Cu toate acestea, există și situații în care legalitatea *web scraping*-ului și *crawling*-ului poate fi neclară. De exemplu, dacă informațiile sunt considerate confidențiale sau protejate de drepturi de autor, atunci activitatea de *web scraping* sau *crawling* poate fi considerată ilegală, chiar dacă site-ul web publică informații în mod deschis.

În general, este important să se acorde atenție termenilor și condițiilor site-ului web și să se obțină permisiunea proprietarului site-ului înainte de a efectua orice activitate de *web scraping* sau *crawling*. De asemenea, este recomandat să se consulte un avocat specializat în domeniul dreptului informatic pentru a înțelege în mod clar implicațiile legale ale activității de *web scraping* sau *crawling*.

Capitolul 4

BestBooking

4.1 Arhitectura aplicației

Cu toate cele menționate în partea teoretică, se vor pune strategiile cap la cap pentru a realiza arhitectura aplicației pe partea de server (poze cu arhitectura). Partea de *back-end* este formată din două blocuri: *API-ul* și scripturile de *data mining*.

Scripturile de *data mining* sunt detașate de aplicație și există câte un script pentru fiecare *site* având un oarecare şablon folosindu-se de clase ajutătoare și o bază de date comună. Scripturile sunt și ele compuse din două bucăți și anume partea de *crawling* și partea de *scraping*. După cum se poate vedea în Figura 4.1 se pornește prin obținerea *link-urilor* pe care o să se colecteze datele. Aceste *link-uri* pot exista deja în sistem și astfel vor fi preluate de acolo sau se va executa o acțiune care descarcă pe baza unui *link* către baza *sitemap-urilor* toate fisierele XML ce conțin articolele dorite (în caz ca aceste fisiere sunt foarte mari și sunt compresate, se vor despacheta în funcție de asta). După ce există acest fond de *link-uri* se va crea un *pool de workers* care va primi ca parametri un *link* către un hotel și un *user input* aleator (Figura 3.19). Fiecare worker va apela o funcție principală a *scraper-ului* pentru acel site care conține adunat toate funcțiile necesare pentru extragerea datelor. Aceste funcții sunt de două tipuri pentru fiecare colecție. Tipurile reprezintă o acțiune ce poate fi colectarea datelor sau parsarea acestora. Colecțiile reprezintă date despre hotel, date de despre locație hotelului și date despre prețurile hoterilor, defapt entitățile aplicației (Figura 4.2). După ce s-au extras și parsat datele acestea vor fi salvate în baza de date, iar acest process se repetă până când s-au epuizat *link-urile*, după care *crawler-ul* se oprește și pornește din nou pe același set ca să se adune când mai multe prețuri a cazărilor pe o perioadă mai lungă de timp pentru a putea ulterior crea un grafic pe baza acestora. Un lucru important de menționat este că la *link-urile* deja parcuse (adică există o referință a lor în baza de date) se va omite colectarea datelor despre hotel și a datelor despre locația acestuia deoarece aceste date tind să nu se schimbe

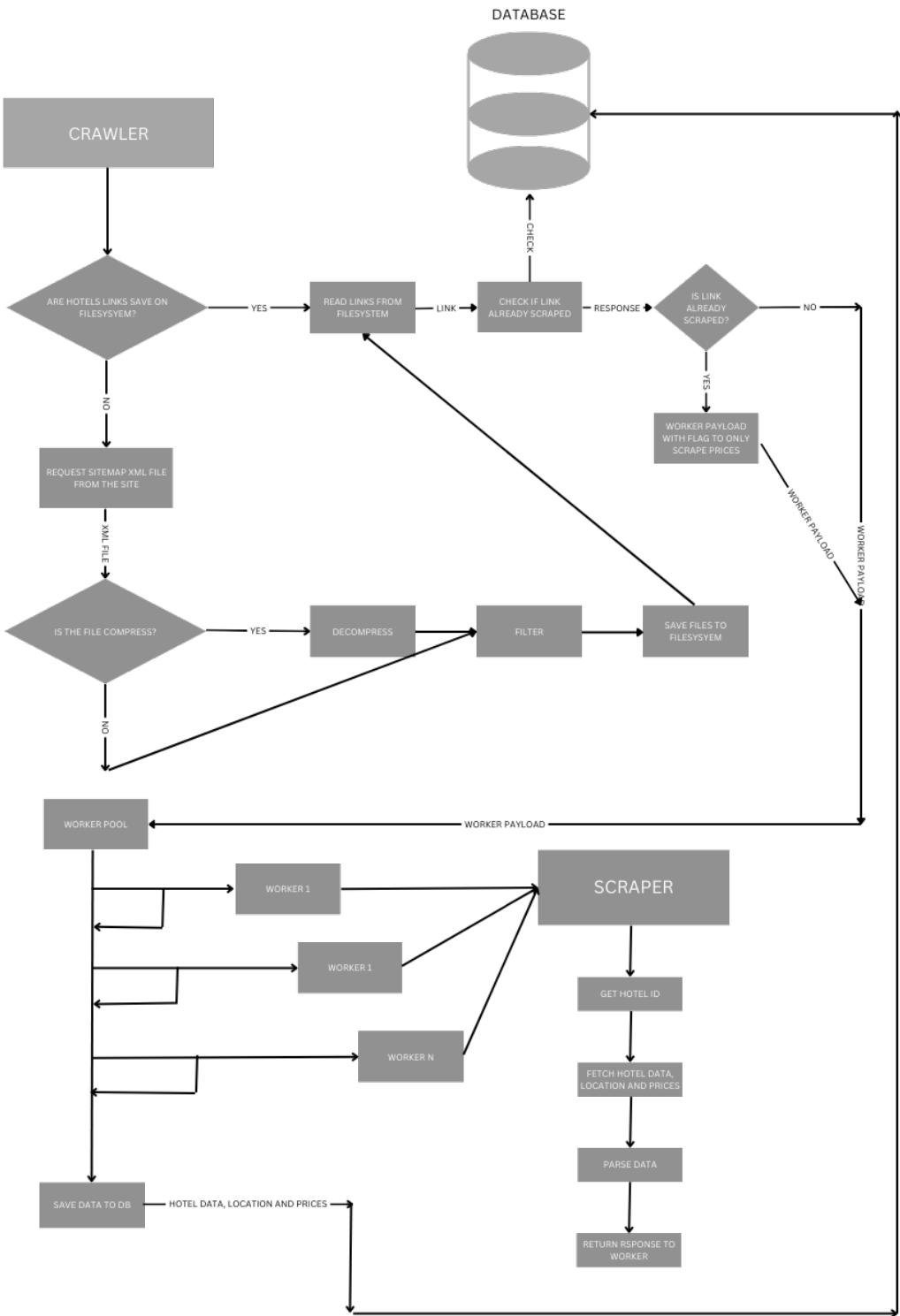


Figura 4.1: Data mining scripts flow

foarte des, aşa că se vor colecta doar date despre preturi pentru a mari viteza de parcurgere a fondului de *link*-uri.

IHotel	ILocation	IHotelPrice
<pre>+siteOrigin: string +siteHotelId: string +hotelName: string +link: string +description: Nullable<string> +rating: Nullable<{ score: number maxScore: number }> +reviews: Nullable<string> +imageLink: Nullable<string> +wifi: Nullable<boolean> +kitchen: Nullable<boolean> +washer: Nullable<boolean> +bayView: Nullable<boolean> +mountainView: Nullable<boolean> +freeParking: Nullable<boolean> +balcony: Nullable<boolean> +bathroom: Nullable<boolean> +airConditioning: Nullable<boolean> +coffeeMachine: Nullable<boolean></pre>	<pre>+hotelId: string +locationName: string +lat: Nullable<number> +lon: Nullable<number> +country: Nullable<string> +area: Nullable<string> +address: Nullable<string> +region: Nullable<string></pre>	<pre>+hotelId: string +pricePerNight: Nullable<number> +pricePerRoom: Nullable<number> +cleaningFee: Nullable<number> +currency: Nullable<string> +serviceFee: Nullable<number> +date: Nullable<Date> +from: Nullable<Date> +to: Nullable<Date> +taxes: Nullable<number> +description: Nullable<string null> +rooms: IRoom[]</pre>

Figura 4.2: Entitățiile aplicației

Partea de API a aplicației constă doar într-un singur *endpoint* care pe baza unui *user input* (Figura 3.19) va returna toate hotele care satisfac datele de intrare (flow chart). Inițial se vor extrage datele pe baza parametrilor ce alcătuiesc locația folosindu-se algoritmi de *fuzzy words* (Figura 4.3) deoarece unele nume locație sau nume de hotel pot fi puțin diferinte în funcție de cum prezintă fiecare site aceste date. Pe baza acestor hoteluri se vor extrage mai departe, din tabelele de locații și prețuri, cele atribuite hotelor gasite pe baza unei chei străine. Prețurile vor fi puțin prelucrate și se vor compune două obiecte din aceste. Primul obiect va conține câte un preț pentru fiecare *site* care este cel mai apropiat de datele alese de utilizator. A doua componentă va fi un obiect care are o lista cu toate prețurile adunate până în acel moment pentru fiecare *site* care va fi folosit pe partea de *front-end* pentru graficul ce conține istoricul prețurilor.



```

SELECT h.* FROM public.hotel h
JOIN public.location l ON h.id = CAST(l.hotelId AS UUID)
WHERE SIMILARITY(LOWER(h.hotelName), LOWER($1)) > 0.5
AND SIMILARITY(LOWER(l.country), LOWER($2)) > 0.2
AND (SIMILARITY(LOWER(l.locationName), LOWER($3)) > 0.2 OR LOWER(l.address) LIKE LOWER('%' || $3 || '%'))
ORDER BY h.id
LIMIT $4 OFFSET $5

```

Figura 4.3: Interogare SQL ce folosește funcția *SIMILARITY* pentru rezolvarea problemei fuzzy words

4.2 Implementare

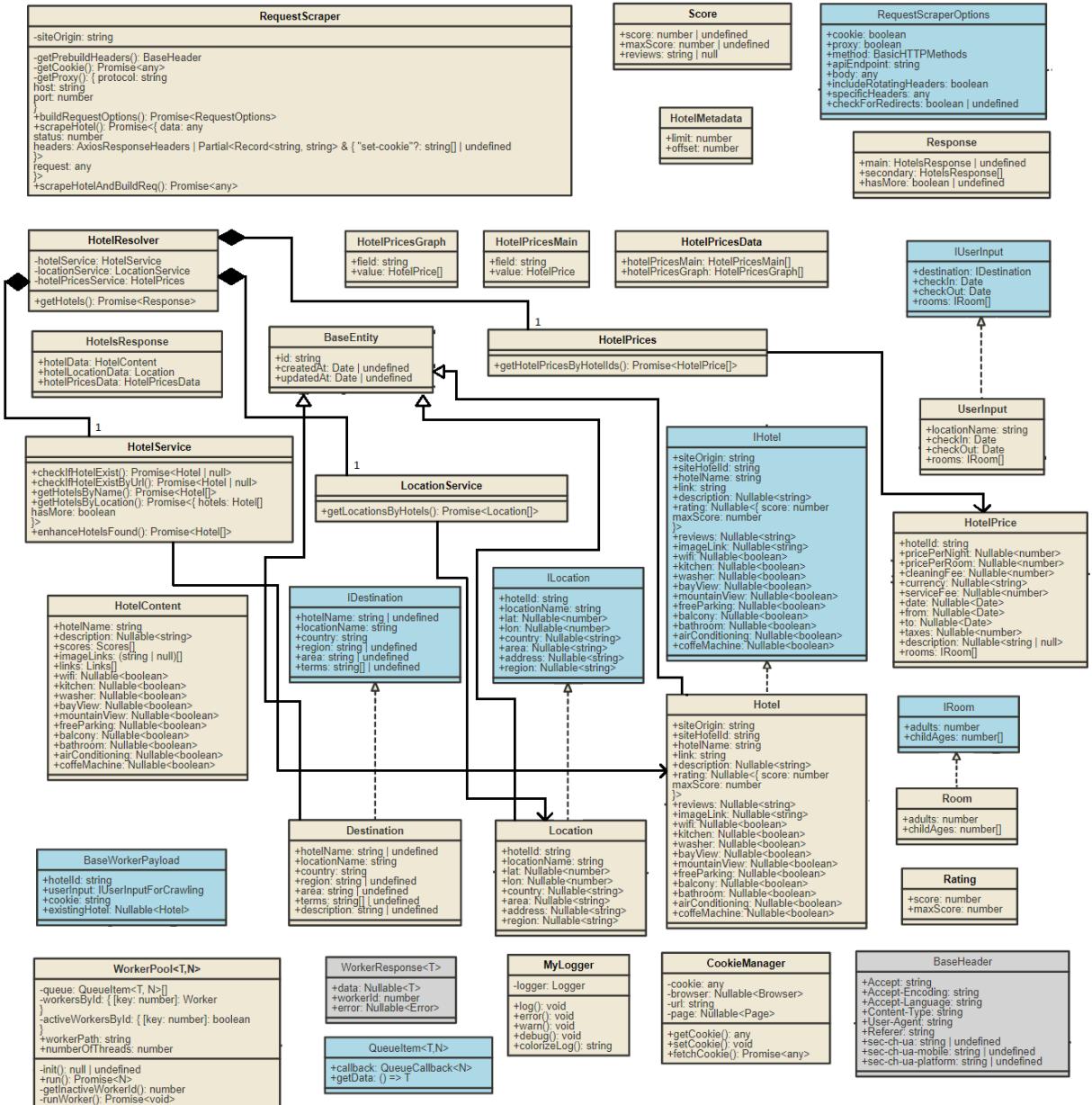


Figura 4.4: Diagrama de clase al aplicației fără scripturi

4.2.1 Limbaj și instrumente

Când vine vorba de *scraping* cel mai utilizat limbaj e *Python* pentru simplicitatea acestuia și multitudinea de instrumente speciale pentru parsarea datelor, însă am vrut să îmi însușesc provocarea de a utiliza *TypeScript* cu *Node.js* în schimb, fiind limbajul meu favorit. Nu este neapărat o alegere greșită deoarece *Node.js* tinde să fie mai rapid decât *Python* doar că e un pic mai restrâns la instrumente.

Ca instrument ce folosește *web driver* la bază se va folosi *Puppeteer*. *Puppeteer* este o bibliotecă de *Node* care oferă un *API* de nivel înalt pentru *Chrome* prin modul “*headless*” și protocolul *DevTools*. Aceasta va fi folosită la modulul *LocationDecoder* și *worker* de tip 3.

Pentru paginile care au elementele în *DOM* se va folosi *JsDOM*, o bibliotecă ce returnează componenta *window* pe baza unui parametru ce va reprezenta *HTML*-ul, astfel vom putea folosi funcții oferite de *JavaScript* din *browser*. Aceasta este foarte rapid când va veni vorba de parsarea elementelor *HTML*.

Pentru paginile la care se va face cerere la *endpoint*-ul de hoteluri se va folosi *Axios*, un pachet ce poate transmite o cere cu *headers* și *payload* customizabile.

Ca bază de date în general se folosesc cele *NoSQL* deoarece datele extrase pot fi deztrămate sau fară o construcție bine definită, astfel fiind ușor de integrat într-o astfel de bază de date. Totuși bazele de date relationale au început să prindă popularitate din nou, una dintre ele fiind *Postgres* care este foarte rapid și cu noul tip de coloană *JSONB* se vor putea salva date nestrukturate fără nici o problemă. Se va folosi și *TypeORM* pentru a ușura munca cu baza de date (Figura 4.5).



```

@Entity()
class RequestConfiguration extends BaseEntity {
  @Column()
  siteOrigin!: string;

  @Column()
  requestURL!: string;

  @Column({ type: 'text' })
  apiType!: ApiType;

  @Column({ type: 'jsonb' })
  headers!: any;

  @Column({ type: 'json' })
  payload: any;

  @Column({ type: 'jsonb' })
  queryParams: any;

  @Column()
  method!: string;
}

```

Figura 4.5: Exemplu entitate cu *TypeORM* și *Postgres*

GraphQL va fi alegerea pentru *API* din simplul motiv de a explora tehnologii noi și pentru că este o combinație excelentă cu *Postgres* și *Node* cu *TypeScript*. Se va folosi *Apollo Server* care este un server de *GraphQL open-source*, care este compatibil cu orice client *GraphQL*, inclusiv *Apollo Client*. Este cea mai bună modalitate de a construi un

API *GraphQL* gata de producție, auto-documentat, care poate folosi date din orice sursă.

Front-end-ul va fi construit pe *TypeScript* în combinație cu *React* pentru flexibilitatea acestuia. Ca să se ușureze sarcinile se va folosi *Tailwind CSS*. *Tailwind* este un *framework CSS* de nivel scăzut, extrem de personalizabil, care vă oferă toate elementele de bază de care aveți nevoie pentru a crea *design-uri* la comandă, fără stiluri opinate.

4.2.2 Clase ajutătoare

În această secțiune se vor detalia unele dintre cele mai importante clase ajutătoare de care se funcțiile de *scrape* și *crawl*.

Cookie Manager

Clasa *CookieManager* (Figura 4.6) acesta primește un singur parametru, un sir de caractere care reprezintă *URL-ul* pentru care se va gestiona *cookie-ul*.

În constructor, proprietățile *cookie*, *browser* și *page* sunt inițializate cu valoarea *null*. Acestea vor fi actualizate mai târziu în metoda *fetchCookie()*.

Clasa *CookieManager* are două metode publice: *getCookie()* și *setCookie()*. Metoda *getCookie()* este utilizată pentru a obține valoarea proprietății *cookie*, iar *setCookie()* este utilizată pentru a actualiza valoarea proprietății *cookie*.

Metoda principală a clasei este *fetchCookie()*, care primește un obiect de opțiuni. Această metodă încearcă să lanseze un *browser* cu ajutorul *Puppeteer* și să acceseze pagina cu *URL-ul* specificat. Deseamne se vor folosi librării ajutătoare cu ar fi *Puppeteer Stealth* ce se asigură că acesta va face cerere la pagină cât mai anonim și uman (prin rotire și prin setarea unor *headere* și ascunderea *cookie-urilor*). Dacă se specifică în opțiuni că se dorește utilizarea unui *proxy*, atunci aceasta este adăugată ca argument în lansarea *browser-ului*.

În interiorul metodei *fetchCookie()*, se așteaptă ca pagina să se încarce complet cu ajutorul opțiunii *waitFor*: 'domcontentloaded'. După ce pagina s-a încărcat complet, metoda *fetchCookie()* obține *cookie-ul* cu ajutorul metodei *cookies()* și îl setează ca valoare pentru proprietatea *cookie*.

În secțiunea *finally*, *browser-ul* și pagina sunt închise indiferent dacă obținerea *cookie-ului* a fost sau nu reușită. Această secțiune garantează că resursele sunt eliberate în mod corespunzător și ajută la prevenirea surgerilor de memorie.



```

class CookieManager {
    private cookie: any;
    private browser: Nullable<Browser>;
    private url: string;
    private page: Nullable<Page>;
    constructor(url: string) {
        this.cookie = null;
        this.browser = null;
        this.page = null;
        this.url = url
    }
    async fetchCookie(ops: { proxy: boolean }) {
        const attemptCount = 3;
        try {
            const args = ['--window-size=1920,1080'];
            if (ops.proxy) {
                args.push('--proxy-server=http://$process.env.STORM_GATEWAY_HOST:$process.env.STORM_GATEWAY_PORT');
            }
            this.browser = await puppeteerXtra.launch({
                args,
                headless: true,
                executablePath: executablePath(),
            });
            this.page = await this.browser.newPage();
            await this.page.setUserAgent(rotateUserAgentWithAPI());
            for (let i = 0; i < attemptCount; i += 1) {
                // Chromium asks the web server for an authorization page
                // and waiting for DOM
                await this.page.goto(this.url, { waitUntil: ['domcontentloaded'] });
                await delay(1000);
                this.setCookie(
                    ...join(
                        ...map(
                            await this.page.cookies(),
                            ({ name, value }) => join([name, value], '='),
                        ),
                        '; ',
                    ),
                );
                // when the cookie has been received, break the loop
                if (this.cookie) break;
            }
            return this.getCookie();
        } catch (err) {
            if (typeof err === 'string') throw new Error(err);
            else console.log(err);
        } finally {
            this.page && await this.page.close();
            this.browser && await this.browser.close();
        }
    }
}

```

Figura 4.6: CookieManager.ts

Request Scraper

Clasa *RequestScraper* (Figura 4.7) este utilizată pentru a efectua cereri HTTP și a returna răspunsurile lor. În cadrul acestei clase sunt definite mai multe metode, fiecare cu propriul său scop și funcționalitate.

Metoda constructor a clasei acceptă un singur parametru, *siteOrigin*, care reprezintă originea site-ului cu care se va face comunicarea prin intermediul cererilor.

Metoda *getPrebuildHeaders* este o metodă privată care întoarce un obiect cu antete predefinite. Aceste antete sunt structurate după tehniciile prezентate în Subcapitolul 3.2.2 și sunt utilizate pentru a completa obiectul de antete care va fi trimis în cadrul cererii HTTP.

Metoda *getCookie* este, de asemenea, o metodă privată și este utilizată pentru a obține *cookie*-urile necesare pentru a efectua o cerere. Pentru a face acest lucru, se creează o instanță a clasei *CookieManager* și se utilizează metoda *fetchCookie* pentru a obține *cookie*-urile.

Metoda *getProxy* este utilizată pentru a returna un obiect care reprezintă un *proxy*, în cazul în care acesta este specificat în opțiunile cererii.

Metoda *buildRequestOptions* este utilizată pentru a construi un obiect care va fi folosit pentru a efectua o cerere. Această metodă primește opțiunile cererii ca parametru și creează un obiect *requestOptions* care conține URL-ul cererii, metoda HTTP, antetele și, dacă este specificat, un proxy și un corp de cerere.

Metoda *scrapeHotel* este utilizată pentru a efectua cererea HTTP cu ajutorul bibliotecii *axios* și a returna răspunsul.

Metoda *scrapeHotelAndBuildReq* este utilizată pentru a construi o cerere și a efectua un apel HTTP. În această metodă sunt utilizate metoda *buildRequestOptions* pentru a construi obiectul de opțiuni și metoda *scrapeHotel* pentru a efectua cererea și a returna răspunsul.

Dacă cererea HTTP nu este efectuată cu succes sau există o eroare, o excepție este aruncată și este capturată într-un bloc catch care creează o instanță a clasei *ErrorRequestFetch* și returnează un mesaj de eroare corespunzător.

Sitemap crawler

Clasa *SitemapCrawler* (Figura 4.8) are două funcții care vor fi folosite pentru extragerea și parsarea *sitemap*-urilor.

Funcția *fetchXmlFile* preia un URL pentru un fișier XML, îl descarcă și îl dezarchivează dacă este în format *gzip*, apoi returnează conținutul fișierului XML ca un sir de caractere. Aceasta folosește biblioteca axios pentru a efectua cererea HTTP și pentru a primi răspunsul în format de flux de octeți.

```

● ● ●

class RequestScraper {
    private siteOrigin: string;

    constructor(siteOrigin: string) {
        this.siteOrigin = siteOrigin;
    }

    private getPrebuildHeaders() {
        return rotatePrebuildHeaders();
    }

    private async getCookie(opts: RequestScraperOptions) {
        const cookieManager = new CookieManager(this.siteOrigin);
        const cookie = await cookieManager.fetchCookie({proxy: opts.proxy ? true : false});
        return cookie;
    }

    private getProxy() {
        const proxy = {
            protocol: 'http',
            host: process.env.STORM_PROXIES_GATEWAY_HOST!,
            port: parseInt(process.env.STORM_PROXIES_GATEWAY_PORT!),
        };
        return proxy;
    }

    async buildRequestOptions(opts: RequestScraperOptions) {
        const requestOptions: RequestOptions = {
            url: opts.apiEndpoint,
            method: opts.method,
        }
        let headers = {
            ...opts.specificHeaders,
        };

        if (opts.includeRotatingHeaders) {
            headers = { ...this.getPrebuildHeaders(), ...headers };
        }

        if (opts.cookie) {
            headers.cookie = await this.getCookie(opts);
        }

        if (opts.proxy) {
            requestOptions.proxy = this.getProxy();
        }

        if (opts.body) {
            requestOptions.data = opts.body;
        }

        requestOptions.headers = headers;
        return requestOptions;
    }

    async scrapeHotel(requestOptions: RequestOptions) {
        const { data, status, headers, request } = await axios(requestOptions);

        return {
            data,
            status,
            headers,
            request,
        };
    }

    async scrapeHotelAndBuildReq(requestOptions: RequestScraperOptions) {
        const { apiEndpoint, body, cookie, proxy, method, specificHeaders, includeRotatingHeaders, checkForRedirects } = requestOptions;

        try {
            const requestOptionsForDetails = await this.buildRequestOptions({
                apiEndpoint,
                body,
                cookie,
                proxy,
                method,
                specificHeaders,
                includeRotatingHeaders,
            });

            const response = await this.scrapeHotel(requestOptionsForDetails);
            const { data, status } = response;

            if (checkForRedirects) {
                if (response.request.res.responseUrl !== apiEndpoint) {
                    throw new Error(`Request was redirected to: ${response.request.res.responseUrl}`);
                }
            }

            if (status >= 400) {
                throw new Error(`Request failed with status code: ${status}`);
            }

            return data;
        } catch(err) {
            let message;
            if (err instanceof Error) {
                message = err.message;
            } else {
                message = String(err);
            }

            throw new ErrorRequestFetch(`fetchError: ${message}`);
        }
    }
}

```

Figura 4.7: RequestScraper.ts

Functia `extractXmlUrlsFromSitemap` primește un fisier XML ca un sir de caractere și extrage toate URL-urile din interiorul fișierului care conțin un anumit sir de caractere include în cadrul URL-ului. Aceasta utilizează biblioteca `xml2js` pentru a converti sirul XML într-un obiect JSON și apoi parcurge acest obiect pentru a găsi URL-urile dorite. Rezultatele sunt returnate sub formă de tablou de siruri de caractere.



```

class SitemapCrawler {
    static async fetchXmlFile(xmlUrl: string): Promise<string> {
        const { data } = await axios({
            url: xmlUrl,
            method: 'GET',
            responseType: 'arraybuffer',
            headers: { "Accept-Encoding": "gzip" },
        });

        if (isGzFile(xmlUrl)) {
            return unzipFile(data);
        }

        return data;
    }

    static extractXmlUrlsFromSitemap(xml: string, include: string): string[] {
        const urls: string[] = [];
        const jsonXml = parseXmlString(xml);

        jsonXml.sitemapindex.sitemap.forEach((sm: any) => {
            if (sm.loc._text.includes(include))
                urls.push(sm.loc._text)
        });

        return urls;
    }
}

```

Figura 4.8: SitemapCrawler.ts

Worker pool

Această clasă (Figura 4.9 implementează un pool de (*workers*) care preiau elemente dintr-o coadă și le procesează. Clasa primește calea către fișierul de lucru pentru fiecare thread și numărul total de *thread*-uri care trebuie create. Constructorul initializează numărul specificat de *thread*-uri, astfel încât fiecare obiect Worker să fie salvat într-un obiect cu o cheie unică identificată de index. În plus, fiecare *thread* este initializat ca inactiv.

Metoda `run` preia date de procesat prin intermediul unei funcții de tip `get data` și întoarce un `Promise` care va fi rezolvat când unul dintre *thread*-urile disponibile va termina de procesat acele date. Dacă toate *thread*-urile sunt ocupate, coada de așteptare este populată cu o nouă comandă.

Metoda privată `getInactiveWorkerId` caută prin obiectul `activeWorkersById` să identifice un *thread* care nu este în prezent ocupat și returnează indexul acestuia. Dacă toate *thread*-urile sunt ocupate, metoda returnează -1.

Metoda privată `runWorker` ia id-ul unui thread disponibil și un element din coada de așteptare, activează *thread*-ul și îl trimită datele pentru a fi procesate. Dacă operația este efectuată cu succes, *callback*-ul este apelat cu rezultatul returnat și *thread*-ul devine disponibil din nou. Dacă o eroare este aruncată, *callback*-ul este apelat cu eroarea respectivă. După ce *thread*-ul a finalizat procesarea și a apelat *callback*-ul, obiectul *worker* este golit de *event listeners* și *thread*-ul este marcat ca fiind disponibil din nou. Dacă există elemente în coada de așteptare, următorul element este procesat de *thread*-ul recent eliberat.

În general, această clasă ajută la gestionarea și distribuirea sarcinilor în mod eficient pe mai multe *thread*-uri, ca cazul acestui proiect va împărții *link*-urile optinute de *crawler* și le va pasa către un worker care va face *scrape*.

4.2.3 Data mining scripts

Scripturile de data mining sunt compuse din părți: *crawling* și *scraping*. În această subsecțiunea vom detalia implementarea unei astfel de script pentru un anumit *website*-ul de cazări *Agoda.com*.

Crawling

Această funcție, `crawlXMLFile()` (Figura 4.10), este o funcție asincronă care primește mai mulți parametri și are ca scop extragerea și parsarea informațiilor din fisiere XML de tip *sitemap*. Acestea conțin adresele *URL* ale hotelurilor și alte informații necesare pentru a extrage date din paginile web corespunzătoare hotelurilor. Funcția rulează mai multe procese de lucru în paralel pentru a procesa mai multe hoteluri simultan.

Mai întâi, se încearcă descărcarea și parsarea fisierelor XML de tip *sitemap* folosind o altă clasă numită *SitemapCrawler*. Această clasă oferă două metode: `fetchXmlFile` și `extractXmlUrlsFromSitemap`, care extrag adresele *URL* ale fisierelor *sitemap* și parsază conținutul acestora în format *JSON*.

Apoi, fisierele *sitemap* sunt filtrate și sunt extrase numai adresele *URL* ale hotelurilor. Acestea sunt împărțite în bucăți de dimensiune maximă de 5 și sunt procesate în paralel utilizând o metodă de lucru în paralel prin clasa *WorkerPool*. Pentru fiecare hotel, este creat un proces de lucru (*worker*) pentru a descărca și a analiza pagina corespunzătoare hotelului și a extrage informații cum ar fi prețul, recenziile etc.

Datele colectate din pagina hotelului sunt stocate în trei variabile diferite *hotelsData*, *hotelLocationsData* și *hotelPricesDataArr*. Acestea sunt apoi inserate în baza de



```

● ● ●

export class WorkerPool<T, N> {
    private queue: QueueItem<T, N>[] = [];
    private workersById: { [key: number]: Worker } = {};
    private activeWorkersById: { [key: number]: boolean } = {};
    public constructor(
        public workerPath: string,
        public numberOfThreads: number
    ) {
        this.init();
    }

    private init() {
        if (this.numberOfThreads < 1) {
            return null;
        }

        for (let i = 0; i < this.numberOfThreads; i += 1) {
            const worker = new Worker(this.workerPath);
            this.workersById[i] = worker;
            this.activeWorkersById[i] = false;
        }
    }

    public run(getData: () => T) {
        return new Promise<N>((resolve, reject) => {
            const availableWorkerId = this.getInactiveWorkerId();
            const queueItem: QueueItem<T, N> = {
                getData,
                callback: (error, result) => {
                    if (error) {
                        return reject(error);
                    }
                    return resolve(result!);
                },
            };

            if (availableWorkerId === -1) {
                this.queue.push(queueItem);
                return null;
            }
            this.runWorker(availableWorkerId, queueItem);
        });
    }

    private getInactiveWorkerId(): number {
        for (let i = 0; i < this.numberOfThreads; i += 1) {
            if (!this.activeWorkersById[i]) {
                return i;
            }
        }
        return -1;
    }

    private async runWorker(workerId: number, queueItem: QueueItem<T, N>) {
        logger.debug(`WORKER ID ${colorizeStringByNumber(workerId.toString(), workerId)}: initialized`);
        const worker = this.workersById[workerId];
        this.activeWorkersById[workerId] = true;

        const messageCallback = (result: N) => {
            queueItem.callback(null, result);
            cleanUp();
        };

        const errorCallback = (error: any) => {
            queueItem.callback(error);
            cleanUp();
        };

        const cleanUp = () => {
            worker.removeAllListeners('message');
            worker.removeAllListeners('error');
            this.activeWorkersById[workerId] = false;

            if (!(this.queue.length > 0)) {
                logger.debug(`WORKER ID ${colorizeStringByNumber(workerId.toString(), workerId)}: exited`);
                return null;
            }

            this.runWorker(workerId, this.queue.shift());
        };
        worker.postMessage(await queueItem.getData());
    }
}

```

Figura 4.9: WorkerPool.ts

date cu ajutorul metodei `QueryBuilder().insert().values()`.

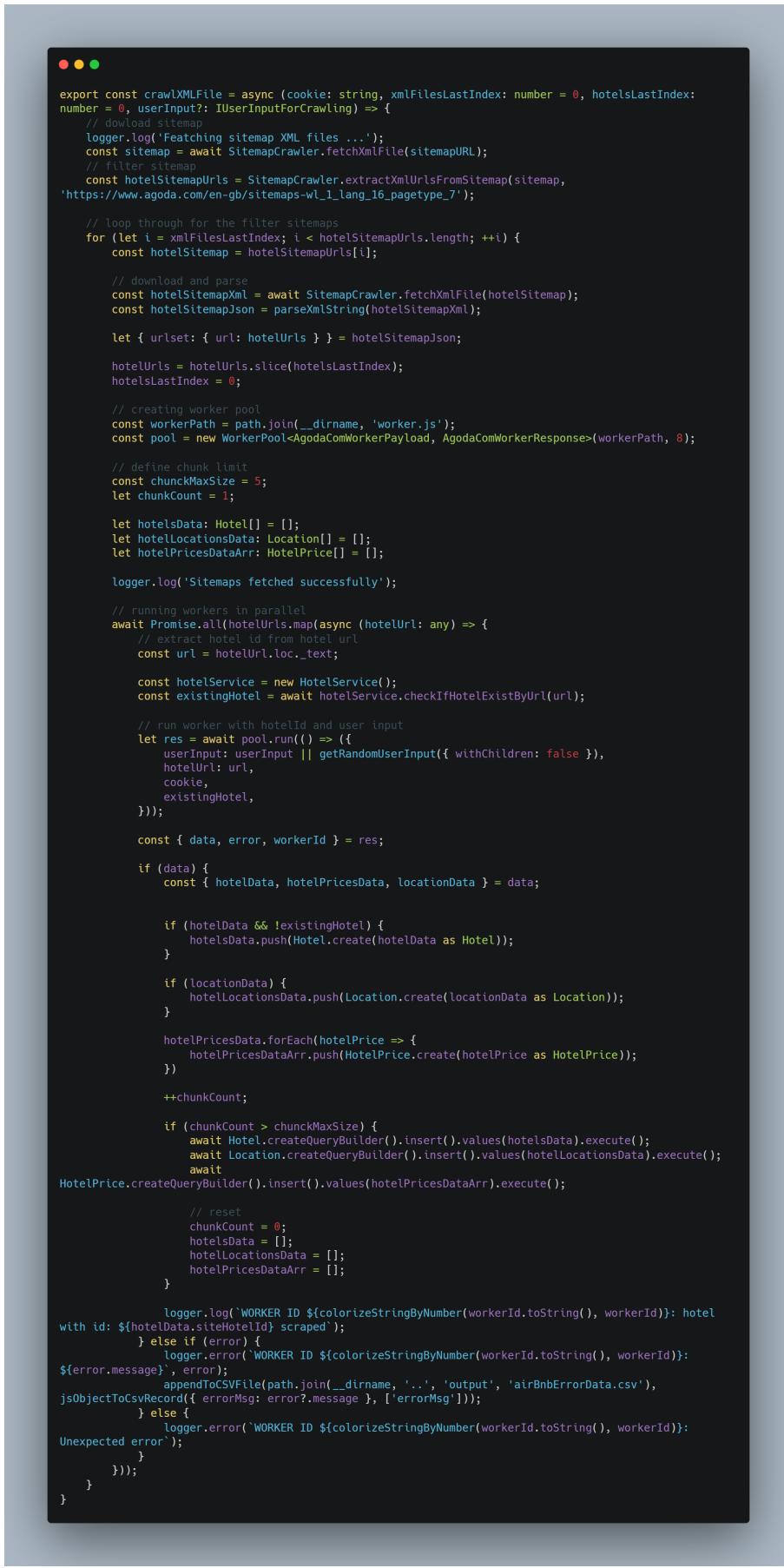
Scraping

Scripturile de *scraping* sunt împărțite în mai multe subfuncții care vor fi explicate alături de codul sursă.

Funcția `getHotelId()` (Figura 4.11) folosește biblioteca *Puppeteer*. Funcția primește un *URL* și accesează pagina web corespunzătoare cu ajutorul *Puppeteer*, cu scopul de a obține ID-ul hotelului asociat acelei pagini. În primul rând, funcția lansează un *browser* cu ajutorul *Puppeteer* și setează o pagină nouă cu *request interception* activat. Acest lucru permite controlul cererilor *HTTP* efectuate de pagina web și oferă posibilitatea de a le anula sau continua în funcție de tipul resursei cerute. De exemplu, resursele precum imagini, fonturi și fișierele de stiluri pot fi anulate pentru a reduce timpul necesar încărcării paginii. Apoi, funcția navighează către *URL*-ul dat ca parametru și așteaptă ca o funcție *JavaScript* să fie evaluată cu succes în pagina web. Această funcție este utilizată pentru a asigura faptul că pagina web a fost complet încărcată și că toate informațiile necesare sunt disponibile. Funcția specifică verifică dacă obiectul `window.initParams.propertyId` este definit, ceea ce sugerează că informațiile necesare sunt acum disponibile pe pagina web. În cele din urmă, funcția extrage ID-ul hotelului din obiectul `window.initParams.propertyId` și îl returnează. Dacă nu poate obține ID-ul, funcția aruncă o excepție cu un mesaj de eroare corespunzător. Funcția închide pagina și *browser*-ul folosind instrucțiuni *finally* pentru a se asigura că acesteia sunt închise în mod corespunzător.

Funcția `fetchHotelAndLocationData()` (Figura 4.12) se ocupă de extragerea datelor relevante despre hotel și locație de pe site-ul *Agoda*, prin efectuarea unei cereri *HTTP POST* către API-ul de proprietate *Agoda*. Funcția primește trei argumente: ID-ul hotelului de pe site-ul *Agoda*, datele introduse de utilizator (de exemplu, perioada și numărul de camere dorite) și *cookie*-ul de autentificare pentru a efectua cererea *HTTP*. Funcția construiește apoi opțiunile de cerere *HTTP* necesare și le pasează către obiectul *RequestScraper* pentru a efectua cererea *HTTP* reală. Dacă cererea este efectuată cu succes și primește un răspuns cu codul 200, funcția va returna datele relevante despre hotel și locație într-un obiect. În caz contrar, se va arunca o excepție cu mesajul corespunzător, care va fi preluată mai târziu pentru a afișa un mesaj de eroare adecvat utilizatorului. De menționat că în această funcție se folosește și clasa *RequestScraper*, care este implementată într-un alt modul și care are rolul de a efectua cereri *HTTP* pentru diverse funcții ale aplicației. În aceași manieră se tratează și funcția care face cerere către prețuri, adică `fetchHotelPrices()` (Figura 4.13).

Funcția `parseHotelAndLocationData()` (Figura 4.15) se ocupă de parsarea datelor unui hotel și a informațiilor despre locație dintr-un obiect JSON primit ca răspuns



```

export const crawlXMLFile = async (cookie: string, xmlFilesLastIndex: number = 0, hotelsLastIndex: number = 0, userInput?: IUserInputForCrawling) => {
    // download sitemap
    logger.log('Fetching sitemap XML files ...');
    const sitemap = await SitemapCrawler.fetchXmlFile(sitemapURL);
    // filter sitemap
    const hotelSitemapUrls = SitemapCrawler.extractXmlUrlsFromSitemap(sitemap, 'https://www.agoda.com/en-gb/sitemaps-wl_1_lang_16_pagetype_7');

    // loop through for the filter sitemaps
    for (let i = xmlFilesLastIndex; i < hotelSitemapUrls.length; ++i) {
        const hotelSitemap = hotelSitemapUrls[i];

        // download and parse
        const hotelSitemapXml = await SitemapCrawler.fetchXmlFile(hotelSitemap);
        const hotelSitemapJson = parseXmlString(hotelSitemapXml);

        let { urlset: { url: hotelUrls } } = hotelSitemapJson;

        hotelUrls = hotelUrls.slice(hotelsLastIndex);
        hotelsLastIndex = 0;

        // creating worker pool
        const workerPath = path.join(__dirname, 'worker.js');
        const pool = new WorkerPool<AgodaComWorkerPayload, AgodaComWorkerResponse>(workerPath, 8);

        // define chunk limit
        const chunkMaxSize = 5;
        let chunkCount = 1;

        let hotelsData: Hotel[] = [];
        let hotelLocationsData: Location[] = [];
        let hotelPricesDataArr: HotelPrice[] = [];

        logger.log('Sitemaps fetched successfully');

        // running workers in parallel
        await Promise.all(hotelUrls.map(async (hotelUrl: any) => {
            // extract hotel id from hotel url
            const url = hotelUrl.loc._text;

            const hotelService = new HotelService();
            const existingHotel = await hotelService.checkIfHotelExistByUrl(url);

            // run worker with hotelId and user input
            let res = await pool.run() => ({
                userInput: userInput || getRandomUserInput({ withChildren: false }),
                hotelUrl: url,
                cookie,
                existingHotel,
            });

            const { data, error, workerId } = res;

            if (data) {
                const { hotelData, hotelPricesData, locationData } = data;

                if (hotelData && !existingHotel) {
                    hotelsData.push(Hotel.create(hotelData as Hotel));
                }

                if (locationData) {
                    hotelLocationsData.push(Location.create(locationData as Location));
                }

                hotelPricesData.forEach(hotelPrice => {
                    hotelPricesDataArr.push(HotelPrice.create(hotelPrice as HotelPrice));
                })
                ++chunkCount;

                if (chunkCount > chunkMaxSize) {
                    await Hotel.createQueryBuilder().insert().values(hotelsData).execute();
                    await Location.createQueryBuilder().insert().values(hotelLocationsData).execute();
                    await HotelPrice.createQueryBuilder().insert().values(hotelPricesDataArr).execute();

                    // reset
                    chunkCount = 0;
                    hotelsData = [];
                    hotelLocationsData = [];
                    hotelPricesDataArr = [];
                }
            }

            logger.log(`WORKER ID ${colorizeStringByNumber(workerId.toString(), workerId)}: hotel with id: ${hotelData.siteHotelId} scraped`);
            } else if (error) {
                logger.error(`WORKER ID ${colorizeStringByNumber(workerId.toString(), workerId)}: ${error.message}`, error);
                appendToCSVFile(path.join(__dirname, '..', 'output', 'airBnbErrorData.csv'), jsObjectToCsvRecord({ errorMsg: error?.message }, [errorMsg]));
            } else {
                logger.error(`WORKER ID ${colorizeStringByNumber(workerId.toString(), workerId)}: Unexpected error`);
            }
        }));
    }
}

```

Figura 4.10: funcția crawlXmlFile()



```
● ● ●

export const getHotelId = async (url: string) => {
  let page, browser;

  try {
    browser = await puppeteerXtra.launch({
      executablePath: executablePath(),
    });

    page = await browser.newPage();
    await page.setRequestInterception(true);

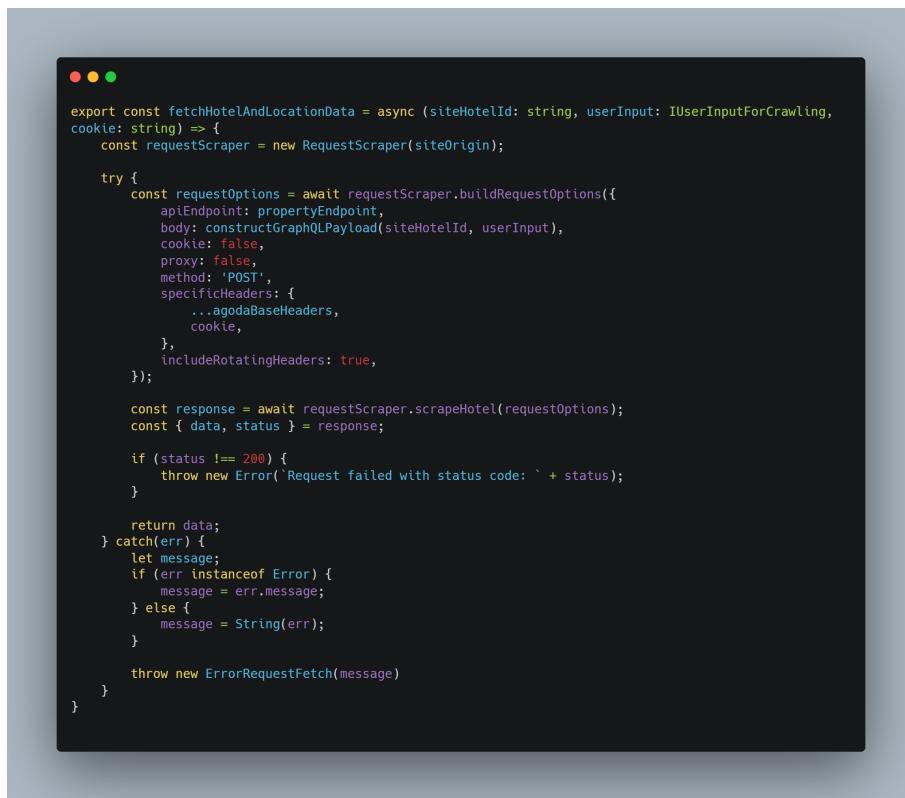
    page.on('request', (request) => {
      if (
        request.resourceType() === 'image' ||
        request.resourceType() === 'font' ||
        request.resourceType() === 'stylesheet'
      ) {
        request.abort();
      } else {
        request.continue();
      }
    });
    await page.goto(url);

    await page.waitForFunction(() => {
      return window.initParams.propertyId !== undefined;
    }, { timeout: 20000 });

    const hotelId = await page.evaluate(() => {
      return window.initParams.propertyId;
    });

    return hotelId;
  } catch(err) {
    let message = 'Could not fetch hotel id, problem with getHotelId function: ';
    if (err instanceof Error) {
      throw new Error(message + err.message);
    }
    throw new Error(message);
  } finally {
    page && await page.close();
    browser && await browser.close();
  }
}
```

Figura 4.11: funcția getHotelId()



```

● ● ●

export const fetchHotelAndLocationData = async (siteHotelId: string, userInput: IUserInputForCrawling, cookie: string) => {
  const requestScraper = new RequestScraper(siteOrigin);

  try {
    const requestOptions = await requestScraper.buildRequestOptions({
      apiEndpoint: propertyEndpoint,
      body: constructGraphQLPayload(siteHotelId, userInput),
      cookie: false,
      proxy: false,
      method: 'POST',
      specificHeaders: {
        ...agodaBaseHeaders,
        cookie,
      },
      includeRotatingHeaders: true,
    });

    const response = await requestScraper.scrapeHotel(requestOptions);
    const { data, status } = response;

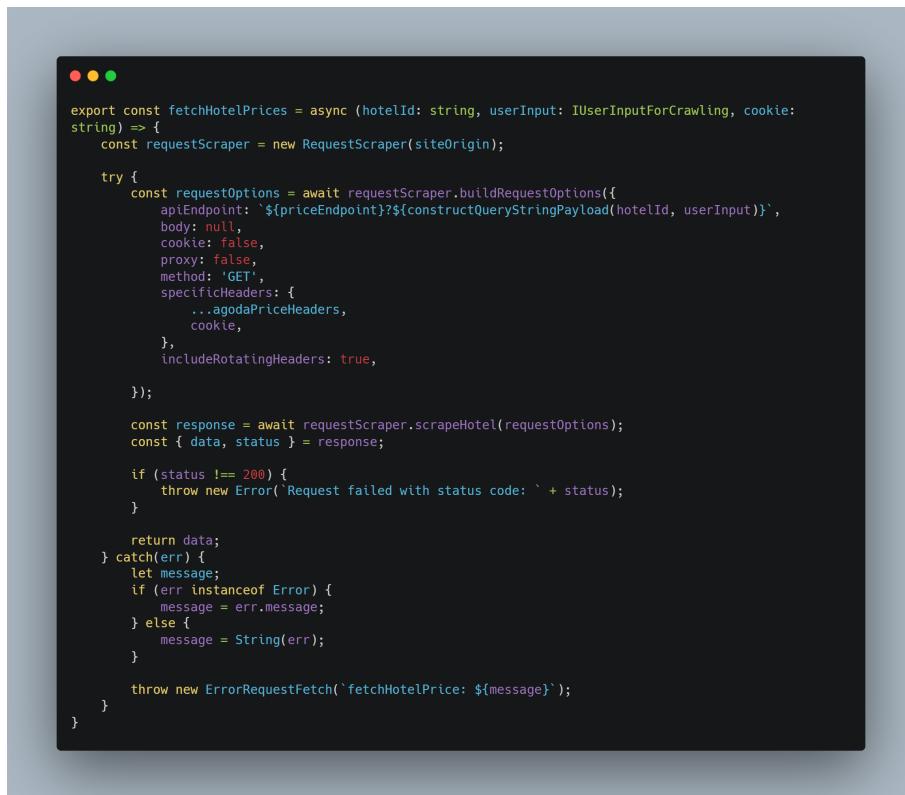
    if (status !== 200) {
      throw new Error(`Request failed with status code: ${status}`);
    }

    return data;
  } catch(err) {
    let message;
    if (err instanceof Error) {
      message = err.message;
    } else {
      message = String(err);
    }

    throw new ErrorRequestFetch(message)
  }
}

```

Figura 4.12: funcția fetchHotelAndLocationData()



```

● ● ●

export const fetchHotelPrices = async (hotelId: string, userInput: IUserInputForCrawling, cookie: string) => {
  const requestScraper = new RequestScraper(siteOrigin);

  try {
    const requestOptions = await requestScraper.buildRequestOptions({
      apiEndpoint: `${priceEndpoint}?${constructQueryStringPayload(hotelId, userInput)}`,
      body: null,
      cookie: false,
      proxy: false,
      method: 'GET',
      specificHeaders: {
        ...agodaPriceHeaders,
        cookie,
      },
      includeRotatingHeaders: true,
    });

    const response = await requestScraper.scrapeHotel(requestOptions);
    const { data, status } = response;

    if (status !== 200) {
      throw new Error(`Request failed with status code: ${status}`);
    }

    return data;
  } catch(err) {
    let message;
    if (err instanceof Error) {
      message = err.message;
    } else {
      message = String(err);
    }

    throw new ErrorRequestFetch(`fetchHotelPrice: ${message}`);
  }
}

```

Figura 4.13: funcția fetchHotelPrices()

de la funcția `fetchHotelAndLocationData()`. Prin intermediul parametrului de răspuns (`response`) se obține informațiile necesare pentru a construi obiectele ce conțin datele parsate. Primul lucru pe care îl face este să genereze un id unic pentru hotel prin intermediul funcției `uuidv4()`. Ulterior, se parsează informațiile legate de hotel și se construiește un obiect `hotelData` care va fi returnat mai târziu. Aceasta conține informații precum numele hotelului, descrierea, rating-ul și numărul de review-uri, link-ul către hotel și link-ul către imaginea principală, dar și informații despre facilitățile oferite de hotel (balcon, parcare gratuită, bucătărie, vedere spre mare/munte, mașină de spălat, Wi-Fi, baie și aer condiționat). De asemenea, se parsează informațiile despre locație și se construiește obiectul `locationData` care conține numele locației, latitudinea și longitudinea, adresa și informații despre țară, regiune și zonă. Această funcție este folosită împreună cu funcția `fetchHotelAndLocationData` pentru a obține și parsă informațiile necesare despre un anumit hotel și locația sa. La fel și pentru prețuri ar fi asemănator dar aș vrea să exemplific un alt tip de funcție de parsare care nu primește ca date intrare un `JSON`, ci un document `HTML`. Prin urmare, funcția `parsePriceData` (Figura 4.14) această funcție preia `HTML`-ul, îl transformă într-un obiect `DOM`, parcurge elementele necesare și extrage informațiile relevante despre prețurile camerei pentru un anumit hotel. În final, această funcție returnează un obiect care conține informații structurate despre prețurile camerei pentru un hotel dat, inclusiv prețul total, prețul pe noapte, moneda și o descriere a camerei.

La final se vor pune împreună toate aceste funcții ca mai apoi `worker`-ul să poată apela aceste acțiuni printr-o singură funcție `scrapeHotelByIdAndUserInput()` (Figura 4.16). De precizat aici că acestă funcție primește un `flag` prin care se anunță `scraper`-ul dacă trebuie să colecteze și informații despre hotel sau doar prețurile acestuia.

4.2.4 *LocationDecoder (legacy)*

`LocationDecoder` (Figura 4.4) este o clasă ce are ca atrbute un serviciu ce gestionează `URL`-urile obținute și extrage configurațiile necasere, și numele locației. Are o funcție publică principală (`getAllUrls()`) ce returnează o listă de `URL`-uri ce au în componență `ID`-ul locației pentru fiecare pagină în parte. Aceste configurații sunt esențiale pentru că prin ele se pot extrage date aproape de pe orice site de cazări doar prin analiza acestuia și contruirea unei configurații protrivite fără să se mai scrie cod în plus. Configurația e simplă, după cum se poate observa în Figura 4.4, acesta este format din patru elemente: domeniul paginii, selectori speciali pentru formularul din Figura 2.1 și `flag`-uri ce semnalează dacă e nevoie să se rezolve `reCAPTCHA` sau dacă e nevoie să se folosească fisierele `stylesheet` pentru deschiderea paginii (unele nu afisează formular fără stil dar altele da, iar dacă se opresc cere-



```
export const parsePriceData = (priceHTML: any, userInput: IUserInputForCrawling, existingHotelId: string): { hotelPricesData: IHotelPrice[] } => {
    const dom = new JSDOM(priceHTML);
    const document = dom.window.document;

    const to = userInput?.checkOut;
    const from = userInput?.checkIn;
    const currency = 'EUR';

    const serviceFee = null;
    const cleaningFee = null;
    const taxes = null;

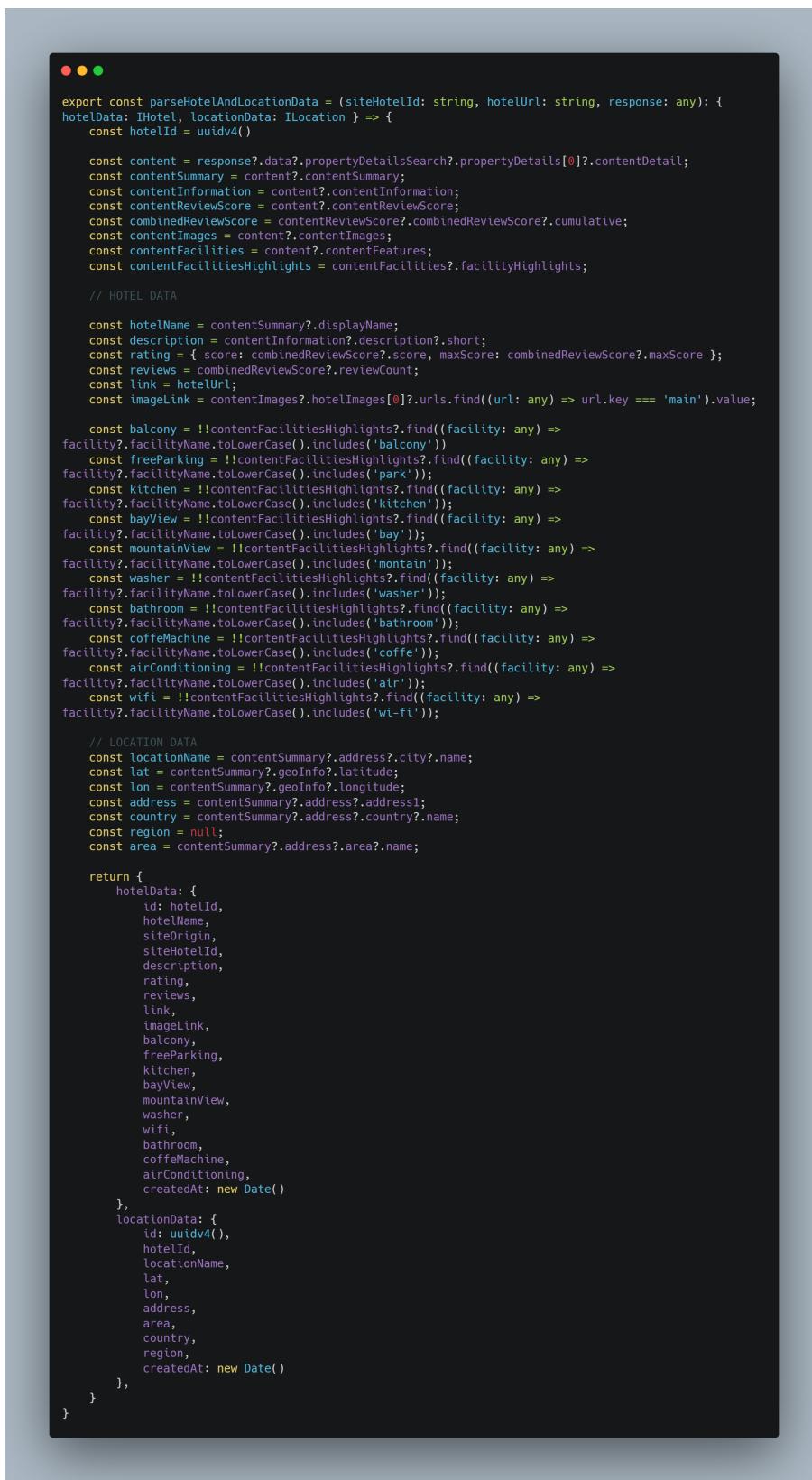
    const hotelPricesData: IHotelPrice[] = [];

    const rooms = document.querySelectorAll('.room');

    rooms.forEach(room => {
        const totalPrice = room.querySelector('.price')?.textContent?.split(' ')[0];
        const pricePerNight = +totalPrice! / getDateDifferenceInDays(userInput.checkIn,
            userInput.checkOut);
        const pricePerRoom = +totalPrice! / getDateDifferenceInDays(userInput.checkIn,
            userInput.checkOut);
        const description = room.querySelector('.room-title')?.textContent;

        hotelPricesData.push({
            id: uuidv4(),
            hotelId: existingHotelId,
            from,
            to,
            pricePerNight,
            pricePerRoom,
            serviceFee,
            cleaningFee,
            taxes,
            currency,
            rooms: userInput.rooms,
            date: null,
            description,
            createdAt: new Date()
        })
    });
    return {
        hotelPricesData
    }
}
```

Figura 4.14: funcția parsePriceData()



```

export const parseHotelAndLocationData = (siteHotelId: string, hotelUrl: string, response: any): {
  hotelData: IHotel, locationData: ILocation } => {
  const hotelId = uuidv4()

  const content = response?.data?.propertyDetailsSearch?.propertyDetails[0]?.contentDetail;
  const contentSummary = content?.contentSummary;
  const contentInformation = content?.contentInformation;
  const contentReviewScore = content?.contentReviewScore;
  const combinedReviewScore = contentReviewScore?.combinedReviewScore?.cumulative;
  const contentImages = content?.contentImages;
  const contentFacilities = content?.contentFeatures;
  const contentFacilitiesHighlights = contentFacilities?.facilityHighlights;

  // HOTEL DATA

  const hotelName = contentSummary?.displayName;
  const description = contentInformation?.description?.short;
  const rating = { score: combinedReviewScore?.score, maxScore: combinedReviewScore?.maxScore };
  const reviews = combinedReviewScore?.reviewCount;
  const link = hotelUrl;
  const imageUrl = contentImages?.hotelImages[0]?.urls.find((url: any) => url.key === 'main').value;

  const balcony = !!contentFacilitiesHighlights?.find((facility: any) =>
    facility?.facilityName.toLowerCase().includes('balcony'))
  const freeParking = !!contentFacilitiesHighlights?.find((facility: any) =>
    facility?.facilityName.toLowerCase().includes('park'));
  const kitchen = !!contentFacilitiesHighlights?.find((facility: any) =>
    facility?.facilityName.toLowerCase().includes('kitchen'));
  const bayView = !!contentFacilitiesHighlights?.find((facility: any) =>
    facility?.facilityName.toLowerCase().includes('bay'));
  const mountainView = !!contentFacilitiesHighlights?.find((facility: any) =>
    facility?.facilityName.toLowerCase().includes('mountain'));
  const washer = !!contentFacilitiesHighlights?.find((facility: any) =>
    facility?.facilityName.toLowerCase().includes('washer'));
  const bathroom = !!contentFacilitiesHighlights?.find((facility: any) =>
    facility?.facilityName.toLowerCase().includes('bathroom'));
  const coffeeMachine = !!contentFacilitiesHighlights?.find((facility: any) =>
    facility?.facilityName.toLowerCase().includes('coffe'));
  const airConditioning = !!contentFacilitiesHighlights?.find((facility: any) =>
    facility?.facilityName.toLowerCase().includes('air'));
  const wifi = !!contentFacilitiesHighlights?.find((facility: any) =>
    facility?.facilityName.toLowerCase().includes('wi-fi'));

  // LOCATION DATA
  const locationName = contentSummary?.address?.city?.name;
  const lat = contentSummary?.geoInfo?.latitude;
  const lon = contentSummary?.geoInfo?.longitude;
  const address = contentSummary?.address?.address1;
  const country = contentSummary?.address?.country?.name;
  const region = null;
  const area = contentSummary?.address?.area?.name;

  return {
    hotelData: {
      id: hotelId,
      hotelName,
      siteOrigin,
      siteHotelId,
      description,
      rating,
      reviews,
      link,
      imageUrl,
      balcony,
      freeParking,
      kitchen,
      bayView,
      mountainView,
      washer,
      wifi,
      bathroom,
      coffeeMachine,
      airConditioning,
      createdAt: new Date()
    },
    locationData: {
      id: uuidv4(),
      hotelId,
      locationName,
      lat,
      lon,
      address,
      area,
      country,
      region,
      createdAt: new Date()
    }
  }
}

```

Figura 4.15: funcția parseHotelAndLocationData()



```

export const scrapeHotelByIdAndUserInput = async (id: string, userInput: IUserInputForCrawling,
    hotelUrl: string, cookie: string, existingHotel?: Nullable<Hotel>): Promise<BaseScraperResponse> => {
    try {
        if (existingHotel) {
            const hotelAndLocationDataRaw = await fetchHotelAndLocationData(hotelUrl, cookie);
            const { hotelData, locationData } = parseDetailsAndLocationData(hotelAndLocationDataRaw);
            const hotelPricesDataRaw = await fetchHotelPrices(id, userInput, cookie);
            const { hotelPricesData } = parsePriceData(hotelPricesDataRaw, userInput, hotelData.id);

            return {
                hotelData,
                locationData,
                hotelPricesData,
            }
        } else {
            const hotelPricesDataRaw = await fetchHotelPrices(id, userInput, cookie);
            const { hotelPricesData } = parsePriceData(hotelPricesDataRaw, userInput,
                existingHotel.id);

            return {
                hotelPricesData,
                hotelData: existingHotel
            }
        }
    } catch(err: any) {
        if (err instanceof Error) {
            err.message += ` . ${hotelUrl}`;
            throw err;
        }
        throw new Error(err);
    }
}

```

Figura 4.16: funcția `scrapeHotelByIdAndUserInput()`

rile care aduc fișiere CSS, imaginile și fontul, performanța crește exponential - Figura 4.18). Un selector (*IGeneralSelector*) e format din selectorul în sine către elementul HTML (*query*) și un index ce reprezintă al cătelea elemente este în caz că se folosește o funcție care selectează mai multe elementele. Selectorii cum ar *IButtonSelector* sau *InputSelector* au doar un atribut în plus ce țin de stil, în sensul că unele pagini fără stil arată aproape indescifrabil și e nevoie de un *click* din *DevTools* în loc de un *click* oferit de *Puppeteer* ce e mult mai uman (algoritm ce utilizează aceste interfețe exemplificat în Figura 4.19). Restul de selectori țin de ce mai poate randa pagini în plus până la apariția formularului, cum ar fi acceptarea de *cookies* (se pot seta aceste *cookie*-uri din înaite pentru a evita acest pas). În Figura 4.17 se poate observa un exemplu de configurație ce poate fi utilizat de *LocationDecoder* pentru pagina www.trip.com, selectorii sunt simpli și robusti utilizându-se *ID*-uri sau atribute ce sunt utilizate de pagini pentru testarea interfețelor.

Așadar funcția care returnează lista de *URL*-uri va folosi o librărie numită *PuppeteerCluster* pentru a putea crea un *cluster* de instanțe de *Chromium "headless"*. Opțiunile atribuite acestui *cluster* pot crește viteza programului dacă sunt setați optim. În Figura 4.20 se va seta libraria clasică *puppeteer* deoarece la acesta se va atașa un pachet *PuppeteerStealth* ce se asigură că acesta va face cerere la pagină cât mai anonim și uman (prin rotire și prin setarea unor *headere* și ascunderea *cookie*-urilor). Opțiunea *puppeteerOptions* este ce s-ar fi pus la o singură instantă *browser*. Aceasta e format din opțiunea *args* care specifică ce opțiuni *default* ale *Chrome*-ului să se anu-



```

const formConf: IFormConfiguration = {
  searchInputSelector: {
    query: '#location-field-destination',
    itemCount: 0,
  },
  searchButtonSelector: {
    query: 'button[type=submit]',
    itemCount: 0,
    buttonVisible: true,
  },
  selectInputOption: {
    query: 'li[data-stid="location-field-destination-result-item"]',
    itemCount: 0,
    buttonVisible: true,
  },
  inputButtonSelector: {
    query: 'button[data-stid="location-field-destination-menu-trigger"]',
    itemCount: 0,
    focusOn: false,
  },
};

const locationDecoderConf: ILocationDecoderConfiguration = {
  formConfiguration: formConf,
  url: new URL('https://www.hotels.com/'),
  resolveCaptcha: false,
  needStyle: false,
};

```

Figura 4.17: Exemplu configurație *LocationDecoder*


```

async search(...) {
  await page.setRequestInterception(true);

  page.on('request', (request) => {
    if (
      request.resourceType() === 'image' ||
      request.resourceType() === 'font'
    ) {
      request.abort();
    } else if (request.resourceType() === 'stylesheet') {
      if (needStyle) {
        request.continue();
      } else {
        request.abort();
      }
    } else {
      request.continue();
    }
  });
  ...
}

```

Figura 4.18: Oprirea cecerilor redundante pentru creșterea performanței



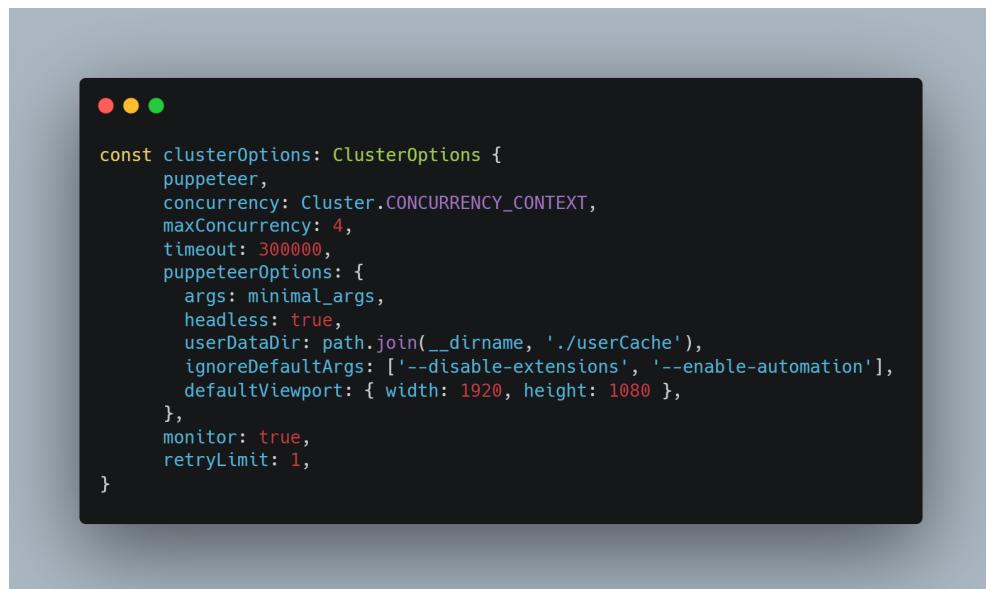
Figura 4.19: Funcție privată din *LocationDecoder* pentru *click*-ul obiectelor din DOM

leze (de exemplu ignorarea opțiunilor de securitate ce pot crea trafic). Tot aici se va specifica și adresa *IP* a unui *proxy*. Altă opțiune de optimizare din acest obiect este *userDataDir* ce descarcă un *cache* al paginii pentru ca să se încarce mai repede următoare dată. Se setează un *viewport* definit din înaite pentru ca selectori să nu se strice din simplul motivul că unele pagini schimbă atributele elementelor în funcție de dimensiunea *window*-ului. În caz ca pagina nu se încarcă adekvat, va reîncerca încă odată operația. Fiecare *thread* va aplica funcția de *search* (Figura 4.21) care completează formularul bazat pe configurație, aplicând *delay*-uri între acțiuni pentru a fi cât mai uman.

În legătură cu *CAPTCHA*, e important de precizat că fiecare formular de căutare a cazărilor este protejat de un astfel de mecanism. Acestea se vor evita prin interacțiunea cât mai umană a *LocationDecoder* cu pagina. În caz ca nu se pot ocoli se va utiliza un API de la *Anti-Captcha* care utilizează ferme de soluționare la *CAPTCHAs*. Desigur acest API nu rezolvă toată problema, tot va fi nevoie de identificarea componentelor necesare pe baza căruia funcționează acesta (3.2.4).

4.2.5 Testare

Testarea s-a aplicat la funcțiile *scraper*-ului menționate în subcapitolul *scraping* pentru fiecare *site* de cazări în parte. Testele sunt de tip *unit* și s-au realizat cu ajutorul librăriilor *mocha* și *chai* din *NodeJS*. Ele erau alcătuite din partea de cerere la *server*-ul *site*-urilor prin care se aduceau datele *raw* și din partea de parsare, asta pentru fiecare entitate din aplicație (Figura 4.2). Se aleagea un fond de *link*-uri cu cazări le-



```

const clusterOptions: ClusterOptions = {
  puppeteer,
  concurrency: Cluster.CONCURRENCY_CONTEXT,
  maxConcurrency: 4,
  timeout: 300000,
  puppeteerOptions: {
    args: minimal_args,
    headless: true,
    userDataDir: path.join(__dirname, './userCache'),
    ignoreDefaultArgs: ['--disable-extensions', '--enable-automation'],
    defaultViewport: { width: 1920, height: 1080 },
  },
  monitor: true,
  retryLimit: 1,
}

```

Figura 4.20: Obiect de opțiuni pentru *PuppeteerCluster*



```

public async getAllUrls() {
  const urls: URL[] = [];

  const cluster = await Cluster.launch(clusterOptions);

  await cluster.task(async ({ page, data }) => {
    const { formConfiguration, resolveCaptcha, url, needStyle } = data as
      ILocationDecoderConfiguration;

    if (resolveCaptcha) {
      this.solveCaptcha(page, url);
    }

    const urlDecoded = await this.search(
      page,
      formConfiguration,
      url,
      needStyle
    );
    saveUrlDecoded(urlDecoded);
  });

  for (let i = 0; i < this.configurations.length; ++i) {
    await cluster.queue(this.configurations[i]);
  }
  ...

  await cluster.idle();
  await cluster.close();
}

return urls;
}

```

Figura 4.21: *getAllUrls()*

gitime dar și *link-uri* care nu existau în cadrul *site-ului* de cazări și se testa inițial dacă cererea funcționează și returnează un document cu informațiile. După care pentru fiecare cerere se păstra răspunsul și pe baza acestui răspuns se aplicau parsări care erau verificate cu niște date care au fost preluate manual și verificate manual pe aceste *site-uri*. Scopul principal al acestor teste este să se stie din timp dacă un *scraper* nu mai funcționează deoarece orice *site* de cazări, și nu numai, își mai schimbă *API-urile*, și în funcție de asta trebuie reglat și *scraper-ul* adekvat. Deasemenea aceste teste asigură calitatea și corectitudinea datelor. Un exemplu de o astfel de suita de test se poate observa în Figura 4.22.

4.2.6 Interfață

Interfața aplicației este simplă urmărind un şablon asemănător cu restul paginilor de *booking*. Partea de implementare nu va fi detaliată deoarece acestă lucrare de licență urmăreste tehnici de *data mining*, interfața fiind utilizată doar pentru a reprezenta mai frumos datele colectate. Aceasta prezintă două pagini principale. Prima pagina sau cea de "home" (Figura 4.23) are în componență clasicul formular pe baza căruia se va extrage *input-ul* necesar *scraper-ului*. La componenta de *input* pentru destinație se va folosi *API-ul Google Places* pentru a asigura introducerea unei locații legitime fără greșeli gramaticale. Tot aici se mai prezintă și o parte din paginile de pe care se vor extrage datele.

După executarea formularului utilizator va fi redirecționat pe a doua pagină sau "hotels page" (Figura 4.24 unde se va afișa lista de hoteluri cu toate prețurile găsite pe o cauză de la mai multe *site-uri* în ordine crescătoare (Figura 4.24). Prețurile afișate sunt selectate în funcție de datele de intrarea ale utilizatorului alegând cea mai apropiată varință. Formularul va fi vizibil și aici doar va ocupa poziția de *navbar*. O componentă hotel sau un *hotel card* are în componență o imagine, numele hotelului, adresa hotelului (*link* cu latitudinea și longitudinea către *Google Maps*), *rating*, *reviews* și prețurile găsite pe paginile răzuite. Aceste căsuțe cu prețuri reprezintă defapt *link-uri* către pagina cu oferta. Se va aplica și paginare prin *infinite scroll* pentru a nu încărca prea tare pagina cum o să fie foarte multe oferte.

Componenta pentru un hotel (Figura 2.5) se poate extinde unde se află informații suplimentare ca: oferte gasite pe alte *site-uri* dar mult mai scumpe decât cele afișate în partea de sus, descriere, facilități și un grafic de dispersie (*scatter chart*). Graficele de dispersie sunt deosebit de utile atunci când doriti să explorați vizual datele și să vedeti dacă există o corelație între două variabile, în cazul acesta prețul și data în care a fost înregistrat. Prin plasarea punctelor de date pe un grafic, puteți vedea rapid dacă există un model sau o tendință în date care nu ar putea fi evidentă dintr-un tabel sau o listă simplă. Se putea utiliza și un grafic cu linie ce conectă mai multe

```

const BASE_URL = 'https://www.agoda.com/';

describe('agoda.com fetchHotelAndLocationData and parseHotelAndLocationData', function () {
  let response1: any = null;

  before(async function() {
    this.siteHotelId1 = '128492';
    ...
    this.hotelUrl1 = 'https://www.agoda.com/en-gb/hotel-alpen-residence/hotel/ehrwald-at.html';
    ...
    this.siteHotelId6 = '-1';

    this.userInput1 = getRandomUserInput({
      withChildren: false,
      numberOfAdultsRange: { min: 2, max: 3 },
      numberofRoomsRange: { min: 1, max: 1 },
    });
    this.userInput2 = getRandomUserInput({
      withChildren: false,
      numberofAdultsRange: { min: 2, max: 3 },
      numberofRoomsRange: { min: 1, max: 1 },
    });

    const cookieManager = new CookieManager(BASE_URL);
    this.cookie = await cookieManager.fetchCookie({ proxy: false });
  })

  context('fetchHotelAndLocationData', function() {
    it('should successfully fetch hotel and location data for siteHotelId1', async function() {
      response1 = await fetchHotelAndLocationData(this.siteHotelId1, this.userInput1, this.cookie);
      await delay(1000);
      expect(response1).to.be.not.null;
    });
    ...
  });

  context('parseHotelAndLocationData', function() {
    it('should successfully parse hotel and location data for siteHotelId1', async function() {
      const parsedData = parseHotelAndLocationData(this.siteHotelId1, this.hotelUrl1, response1);
      const { hotelData, locationData } = parsedData;

      expect(hotelData.hotelName).to.be.eq('Hotel Alpen Residence');
      expect(hotelData.siteOrigin).to.be.eq('https://www.agoda.com');
      expect(hotelData.siteHotelId).to.be.eq('128492');
      expect(hotelData.description).to.be.eq('Conveniently situated in the Ehrwald part of Ehrwald, this property puts you close to attractions and interesting dining options. This 4-star property is packed with in-house facilities to improve the quality and joy of your stay.');
      expect(hotelData.rating?.score).to.be.eq(9.2);
      expect(hotelData.rating?.maxScore).to.be.eq(10);
      expect(hotelData.reviews).to.be.eq(317);
      expect(hotelData.link).to.be.eq(this.hotelUrl1);
      expect(hotelData.imageLink).to.be.eq('//q-xx.bstatic.com/xdata/images/hotel/840x460/294903170.jpg?k=8a46087430fa27c8c5e4e0edfae0ebc45a0944991c2b0347f07ab989296f356&o=');
      expect(hotelData.balcony).to.be.eq(false);
      expect(hotelData.freeParking).to.be.eq(true);
      expect(hotelData.kitchen).to.be.eq(false);
      expect(hotelData.bayView).to.be.eq(false);
      expect(hotelData.mountainView).to.be.eq(false);
      expect(hotelData.washer).to.be.eq(false);
      expect(hotelData.wifi).to.be.eq(true);
      expect(hotelData.bathroom).to.be.eq(false);
      expect(hotelData.coffeeMachine).to.be.eq(false);
      expect(hotelData.airConditioning).to.be.eq(true);

      expect(locationData.hotelId).to.be.eq(hotelData.id);
      expect(locationData.locationName).to.be.eq('Ehrwald');
      expect(locationData.lat).to.be.eq(47.396009146118164);
      expect(locationData.lon).to.be.eq(10.91990021972656);
      expect(locationData.address).to.be.eq('Florentin-Wehner-Weg 37');
      expect(locationData.area).to.be.eq('Ehrwald');
      expect(locationData.country).to.be.eq('Austria');
      expect(locationData.region).to.be.eq(null);
    });
    ...
  });
});

```

Figura 4.22: O parte a testelor din fisierul `fetchAndParseHotelAndLocationData.test.ts`

astfel de puncte pentru a vedea exact dacă prețul scade sau crește, dar cum nu e vorba doar de un sigur produs, ci de un hotel care poate avea mai multe tipuri de camere (foarte multe puncte în grafic), iar prețurile fluctuează în funcție de mai mulți parametri cum ar fi timpul de ocupare sau numărul oaspeți (fiecare are un *pricing model* destul de complex) am decis că această reprezentarea este cea mai bună.

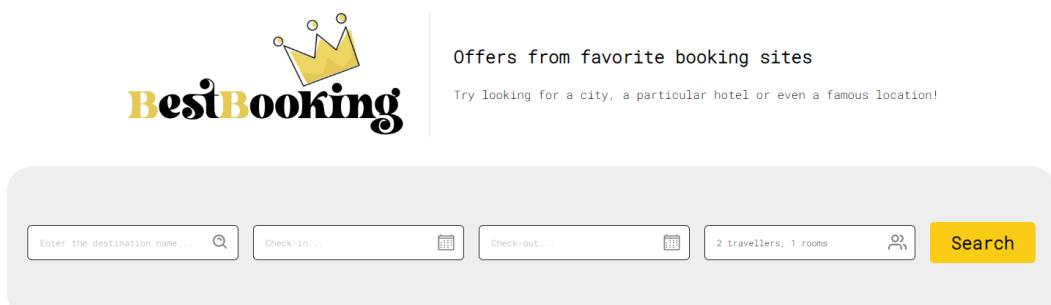


Figura 4.23: Pagina principală BestBooking

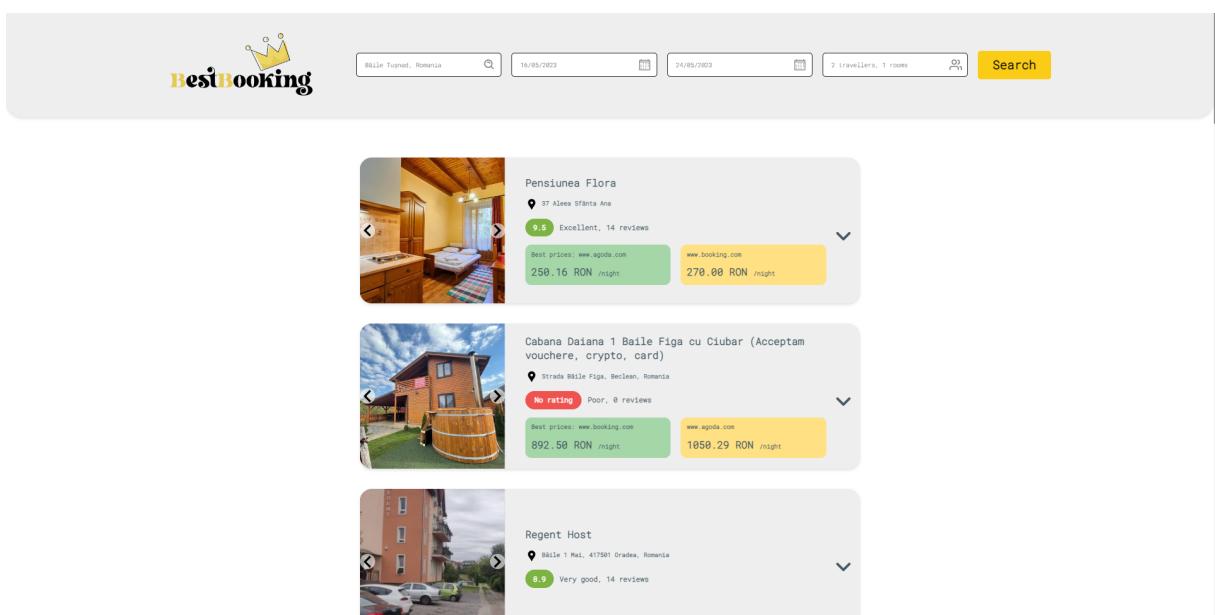


Figura 4.24: Pagina cu lista de hoteluri BestBooking

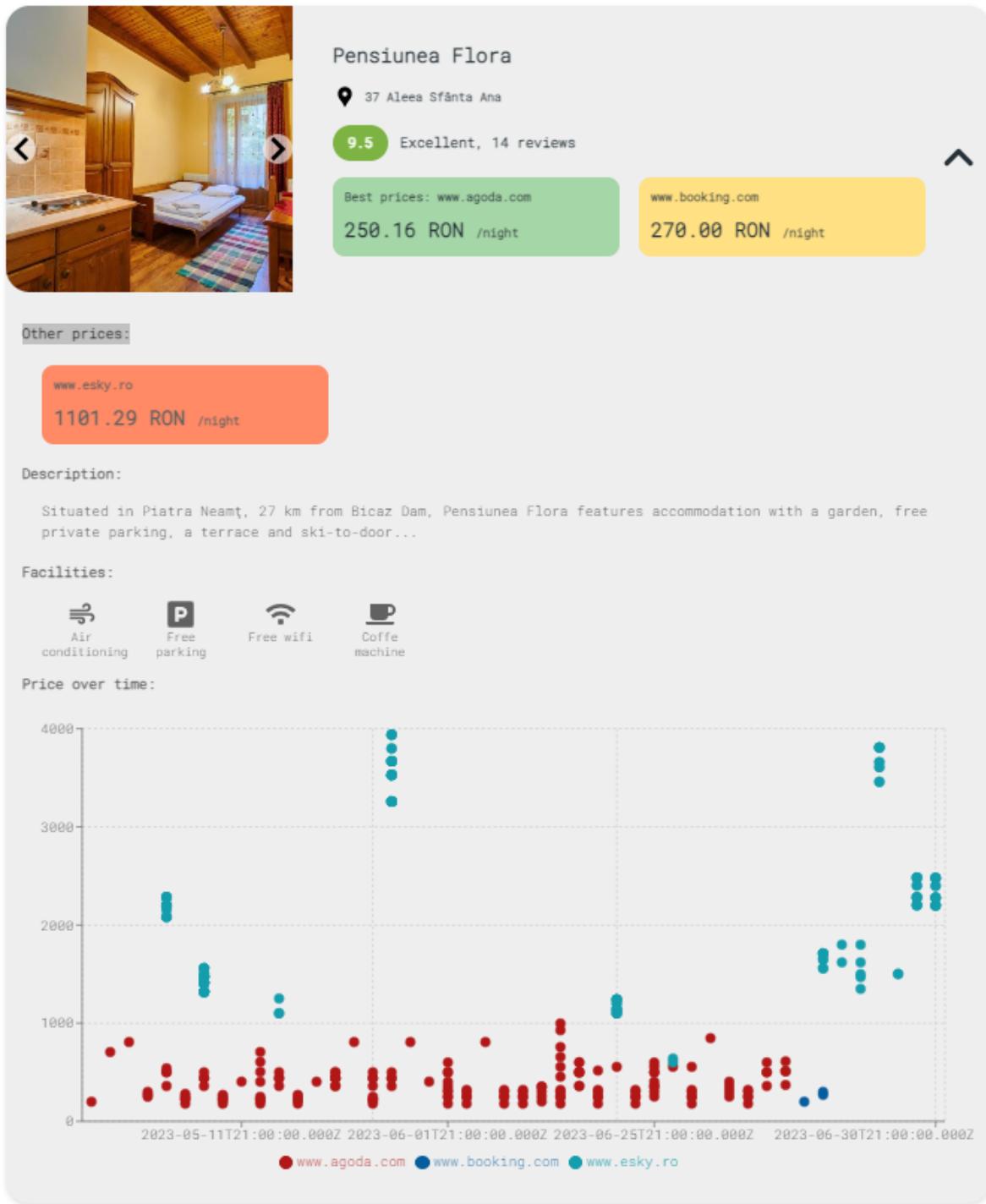


Figura 4.25: Pagina cu lista de hoteluri *BestBooking*

Capitolul 5

Concluzii și dezvoltari viitoare

În concluzie prin construcția acestui *scraper* s-au respectat toate practiciile fundamentale și avansate al acestei teme. Prin utilizarea unei arhitecturi configurabile, acest *scraper* poate fi adaptat și extins pentru a colecta date de pe o varietate mai mare de site-uri de *booking*. Acest lucru îi conferă o mare flexibilitate și un grad ridicat de scalabilitate. În plus, prin rularea scripturilor de data mining pe o perioadă mai lungă de timp, se poate colecta un volum mai mare de date, ceea ce poate fi extrem de util în realizarea analizelor și a predictiilor.

De exemplu, datele colectate prin intermediul acestui scraper pot fi utilizate pentru a monitoriza fluctuațiile de prețuri și oferte de cazare dintr-o anumită regiune pe o perioadă de timp. Aceste date pot fi apoi utilizate pentru a dezvolta modele de analiză și predictivitate, care să ajute utilizatorii să ia decizii mai informate în ceea ce privește planificarea călătoriilor și a rezervărilor de cazare.

Este important de menționat că există anumite riscuri și probleme de securitate asociate cu *data mining*-ul, inclusiv încălcarea drepturilor de autor și a drepturilor de proprietate intelectuală, precum și posibilitatea de a accesa neautorizat informații personale sau confidențiale. Prin urmare, este important să se ia în considerare aceste riscuri și să se respecte toate regulile și regulamentele relevante în ceea ce privește colectarea și utilizarea datelor.

Toate aceste tehnici au demonstrat că mereu pot exista nișe de securitate de care trebuie ținut cont la realizarea unei aplicații web și nu numai, dar acestea mereu vor exista pentru că, *web scraping*-ul, este un joc de-a șoarecele și pisica.

Bibliografie

- [AC] Anti-Captcha. Anti-captcha api docs. how bypass any captcha. <https://anti-captcha.com/apidoc>.
- [Agn] Sam Agnew. Web scraping and parsing html in node.js with jsdom. <https://www.webscrapingapi.com/the-ultimate-guide-to-web-scraping-with-javascript-and-node-js/>.
- [Ban] Sam Banks. Performance testing with puppeteer cluster. <https://stackchat.com/blog/puppeteer-cluster-performance-testing>.
- [Chr] Marcel Christianis. Scraping web apps using direct http request. <https://medium.com/analytics-vidhya/scraping-web-apps-using-direct-http-request-f5c02a2874fe>.
- [Cie] Maciej Cieślar. A complete guide to threads in node.js. <https://blog.logrocket.com/a-complete-guide-to-threads-in-node-js-4fa3898fe74f/>.
- [cloa] cloudflare.com. How captchas work — what does captcha mean? <https://www.cloudflare.com/learning/bots/how-captchas-work/>.
- [clob] cloudflare.com. What is a web crawler? how web spiders work. <https://www.cloudflare.com/learning/bots/what-is-a-web-crawler/>.
- [Dal16] Kyran Dale. *Data Visualization with Python and JavaScript: Scrape, Clean, Explore and Transform Your Data*. O'Reilly, USA, 1st edition, 2016.
- [Den] James Densmore. Web scraping basics. <https://towardsdatascience.com/ethics-in-web-scraping-b96b18136f01>.
- [Fle] Flemmerwill. 5 steps to building a faster web crawler. <https://betterprogramming.pub/5-tips-to-build-a-faster-web-crawler-f2bbc90cf233>.

- [Hua] Yina Huang. 5 things you need to know of bypassing captcha for web scraping. <https://www.octoparse.com/blog/5-things-you-need-to-know-of-bypassing-captcha-for-web-scraping>.
- [Kan21] Satwik Kansal. Advanced python web scraping: Best practices and workarounds. <https://www.codementor.io/blog/python-web-scraping-6312v9sf2q>, 2021.
- [Kir] Vytautas Kirjazovas. Most common http headers. <https://oxylabs.io/blog/5-key-http-headers-for-web-scraping>.
- [Meu] Maxine Meurer. Infinite scroll with puppeteer. <https://www.scrapingbee.com/blog/infinite-scroll-puppeteer/>.
- [Mit15] Ryan Mitchell. *Web Scraping with Python: Collecting Data from the Modern Web*. O'Reilly, USA, 2nd edition, 2015.
- [Nec] Yelyzaveta Nechytilo. How to make web scraping faster – python tutorial. <https://oxylabs.io/blog/how-to-make-web-scraping-faster>.
- [Pup] Puppeteer. Puppeteer documentation. <https://pptr.dev/#?product=Puppeteer&version=v13.5.2&show=api-class-page>.
- [Rod] Ander Rodríguez. Mastering web scraping in python: Avoid detection like a ninja. <https://www.zenrows.com/blog/stealth-web-scraping-in-python-avoid-blocking-like-a-ninja#ip-rate-limit>.
- [Sfi] Robert Sfichi. The ultimate guide to web scraping with javascript and node.js. <https://www.webscrapingapi.com/the-ultimate-guide-to-web-scraping-with-javascript-and-node-js/>.
- [Siv] Anupriya Sivalingam. Web scraping with puppeteer. <https://blog.francium.tech/web-scraping-with-puppeteer-ca9e5c1b7802>.
- [Tip] David Tippett. If you are web scraping don't do these things. <https://levelup.gitconnected.com/if-you-are-web-scraping-dont-do-these-things-2cba2ebe5b29>.
- [Wu] Songhao Wu. Web scraping basics. <https://towardsdatascience.com/web-scraping-basics-82f8b5acd45c>.