

UNIVERSITATEA “BABEŞ-BOLYAI” CLUJ-NAPOCA
FACULTATEA DE MATEMATICĂ ȘI INFORMATICĂ
SPECIALIZAREA INFORMATICĂ ROMÂNĂ

LUCRARE DE LICENȚĂ

**SERVICII WEB ȘI ACCESAREA ACESTORA
PRIN INTERMEDIUL APLICAȚIILOR
MOBLIE**

Coordonator științific,
Lect. Ph.D. Radu D. Găceanu

Absolvent,
Paroș Raul-Constantin

Cluj-Napoca
2018

Cuprins

Introducere.....	3
1. Servicii Web.....	5
1.1 Protocolul HTTP.....	5
1.2 Servicii Web. Descrierea problemei.....	8
1.3 Servicii Web RPC, SOAP și REST.....	13
1.3.1 RPC (Remote Procedure Call).....	13
1.3.2 SOAP (Simple Object Access Protocol).....	14
1.3.3 REST (Representational State Transfer).....	16
2. Tehnologii folosite.....	19
2.1 Tehnologii folosite pentru aplicația server.....	19
2.1.1 Spring.....	19
2.1.2 Hibernate și JPA (Java Persistence API).....	25
2.2 Tehnologii folosite pentru aplicația client.....	29
2.2.1 React.....	29
2.2.2 Redux.....	30
2.2.3 React Native.....	30
3. Aplicația Parker.....	32
3.1 Arhitectura.....	32
3.2 Funcționalități.....	40
Concluzii.....	48
Bibliografie.....	49

Introducere

În orașele cu un grad de ocupare ridicat, locurile de parcare din zonele de interes devin o resursă pe care oamenii și autoritățile trebuie să o gestioneze intelligent. Astfel că, un studiu [1] arată că într-un oraș ca New York-ul, oamenii pierd în medie aproximativ o sută de ore pe an căutând un loc de parcare liber.

Acest lucru aduce daune cuantificabile din punctul de vedere al timpului pierdut, banilor irosiți pe combustibil consumat sau a emisiilor de dioxid de carbon din atmosferă care se produc în plus. Studiul arată media de ore din cele mai aglomerate 10 orașe din Statele Unite ale Americii și calculează o sumă aproximativă care reprezintă daunele estimate pentru aceste orașe. Aceasta se ridică la 72,7 miliarde de dolari pentru anul 2017, anul în care compania INRIX a efectuat studiul. De asemenea căutarea unui loc de parcare într-un interval orar în care traficul este deja congestionat duce la o îngreunare suplimentară datorată faptului că pentru a putea găsi un loc de parcare pe o stradă, șoferul trebuie să încetinească pentru a vedea din timp locul liber și să opreasca traficul pentru o perioadă pentru a putea parca. Toate aceste lucruri fac locurile de parcare să fie o resursă indispensabilă a secolului XXI care trebuie gestionate prin sisteme inteligente pentru a putea fi maximizată eficiența folosirii lor.

Desigur, există deja aplicații pe piață care abordează și încearcă să rezolve problema aceasta, însă soluția propusă de noi abordează problema diferit de celealte soluții de pe piață din cauză că majoritatea se focusează pe optimizarea gestiunii locurilor de parcare din parcările private aflate într-un oraș sau parcările din aeroporturi. Aplicația dezvoltată în cadrul acestei lucrări se concentrează pe eficientizarea folosirii locurilor de parcare din parcările publice de pe străzi, sau parcările private care rămân libere în timpul zilei. Numărul acestora este mult mai mare decât cel al parcărilor private iar costul parcării este de obicei mult mai mic.

Noi propunem o soluție care se bazează pe datele oferite de utilizatori cu privire la parcările libere folosite în general de ei sau chiar parcările private pe care le dețin dar pe care nu le folosesc o anumită perioadă din zi. Aceste locuri de parcare pot fi astfel rezervate de către șoferi, astfel că un șofer care a rezervat în prealabil un loc de parcare folosind aplicația, va beneficia de economisirea timpului și combustibilului.

Aplicația este dezvoltată ca un serviciu pentru a oferi posibilitatea prezentării informațiilor stocate pe mai multe platforme și a unei eventuale integrări cu aplicații care acoperă aceiași arie. Astfel că

aplicația constă de fapt dintr-un serviciu REST (Representational State Transfer) și o aplicație dezvoltată pentru telefoanele mobile care oferă informațiile șoferilor în timp real despre locurile de parcare din zona lor de interes.

Aceste componente vor fi prezentate în următoarele capitole ale lucrării care vor detalia conceptele, ideile și tehnologiile folosite în realizarea acestei aplicații și de asemenea vom prezenta comparații cu anumite aplicații care abordează aceiași problemă.

1. Servicii Web

The World Wide Web (WWW) este deseori confundat cu Internetul din cauza cuplării strânse a celor două concepte, însă Internetul este, după cum și numele sugerează, o rețea globală de dispozitive și alte rețele interconectate. [3]

Ne vom referi de aici înainte în această lucrare la World Wide Web pe scurt, Web sau www. Astfel că web-ul este un model, un spațiu informațional prin care elementele de interes, numite și resurse, sunt identificate prin identificatoare globale denumite Uniform Resource Identifier (URI). Informațiile sunt legate între ele prin hiperlegături (hyperlinks), care reprezintă noduri logice prin care se face navigarea între resurse. Astfel că, protocolul de comunicare în sistemul web este Hypertext Transfer Protocol (HTTP), protocol care are ca scop transferul de hipertext (hypertext), text care conține hiperlegături și care facilitează navigarea structurată între resurse prin accesarea URI-urilor acestora.

În continuare vom prezenta pe scurt protocolul HTTP și conceptul de aplicație web pentru a putea crea un context în care putem discuta despre conceptul de interes din acest capitol, și anume Servicii Web și în special Servicii Web RESTful.

1.1 Protocolul HTTP

Protocolul HTTP a fost inițiat de către Tim Berners-Lee, același om care a propus și inventat pentru prima oară spațiul Web și primul browser web. Este protocolul de transfer utilizat pe Web și se află la nivelul aplicație al stivei TCP/IP (Transmission Control Protocol/Internet Protocol). Conceptele fundamentale ale protocolului HTTP sunt cererea și răspunsul, acțiuni ce se realizează de către client, respectiv server pentru accesarea, modificarea, înlocuirea sau ștergerea resurselor. Entitățile angajate într-o astfel de comunicare se pot denumi astfel client și server. Clientul este entitatea ce inițiază comunicarea cu o cerere care se poate descrie ca un mesaj cu o structură prestabilită care să specifică detalii ca spre exemplu: tipul de operație pe care clientul dorește să o realizeze pe o anumită resursă, datele necesare pentru operație și uri-ul la care se găsește resursa. Serverul este cel care procesează aceste cereri și răspunde clientului cu un mesaj care are de asemenea o formă prestabilită și care conține detalii despre procesarea cererii cum ar fi un cod de stare prin care

serverul descrie starea cererii pe care a trebuit să o proceseze, cum ar fi dacă cererea a putut sau nu fi procesată cu succes. De asemenea răspunsul poate conține, în cazul în care serverul a reușit să proceseze cu succes cererea, resursa vizată de către client fie ea o pagină HTML (Hypertext Markup Language), un fișier video, un fișier XML (Extensible Markup Language), un fișier audio, etc.

Faptul că un server poate răspunde cu toate aceste tipuri de date unui client a făcut protocolul HTTP să fie versatil și să corespundă unei game variate de cerințe și nevoi având un caracter generic pe care s-au putut dezvolta cu timpul alte paradigmă, ca spre exemplu subiectul nostru de interes din acest capitol, serviciile Web. Acest caracter generic a devenit o bază puternică pentru inovațiile viitoare care au urmat și care s-au putut baza pe un protocol de transfer matur, puternic formalizat și gestionat de către o entitate internațională, World Wide Web Consortium (W3C).

În general, comunicarea dintre un server și un client este inițiată de către client printr-o cerere care are o structură ca în figura de mai jos.

```
GET /parking-spot/get
Host: http://localhost:8080
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/67.0.3396.87 Safari/537.36
Accept: application/json
Accept-Language: en-US
Accept-Encoding: gzip, deflate, compress, identity
Connection: Keep-Alive
```

Fig. 1 Exemplu de cerere HTTP

În această imagine se pot identifica elementele componente ale unei cereri. Prima informație care se pune în acest tip de mesaj este metoda de acces. Aceste metode sunt foarte importante în contextul serviciilor web, deoarece corespund cu acțiunile pe care un server le poate efectua asupra unei resurse și descriu astfel logica pe care serverul o poate aplica unei resurse. Deoarece pentru foarte mult timp web-ul a fost folosit doar în interacțiunile om-mașină, cele mai folosite metode sunt metoda GET și POST. Aceste două metode sunt întâlnite ușual în comportamentul browserelor Web care de obicei afișează o pagină Web folosind o cerere cu metoda GET specificată, sau care trimit unui server datele transmise de utilizator printr-o cerere cu metoda POST specificată. În contextul serviciilor web,

care caută să faciliteze comunicarea mașină-mașină, se folosesc și alte metode care specifică comportamente diferite cum ar fi: PUT, DELETE, HEAD, TRACE, CONNECT, OPTIONS.

Figura de mai jos reprezintă structura răspunsurilor pe care serverul le poate trimite înapoi unui client după procesarea cererii.

Status: 200

Date: Tue, 19 Jun 2018 22:47:11 GMT

Content-Type: application/json; charset=utf-8

Expires: Mon, 01 Jan 1990 00:00:00 GMT

Cache-Control: no-cache, no-store, max-age=0, must-revalidate

Fig. 2 Exemplu de răspuns HTTP

În răspuns se pot identifica elementele importante precum codul de stare și dacă e cazul, tipul conținutului din răspuns, lungimea răspunsului și resursa propriu zisă care e reprezentată de conținutul/corpul răspunsului.

Dacă atunci când folosim un browser să accesăm pagini Web noi, utilizatorii, nu suntem interesanți de aceste detaliile deoarece browser-ul le gestionează pentru noi, atunci când vorbim de servicii Web toate aceste elemente sunt foarte importante din cauză că orice variație poate schimba drastic modul în care serverul cu care dorim să comunicăm va acționa sau va răspunde. Astfel că putem deduce că în mod ușor, capacitatele protocolului HTTP nu sunt puse în valoare atunci când este vorba de accesarea paginilor Web de către un utilizator. Asta pentru că aceste mesaje nu au fost implementate cu scopul de a fi ușor înțelese de către oameni, iar comunicarea cu serverul chiar dacă este posibilă prin manipularea acestor mesaje, nu este facilă. De aceea, în acest tip de comunicare, interacțiunea între utilizator și server trece prin interfață expusă de către browser, care devine implicit clientul interacțiunii. Însă acest model de comunicare puternic formalizat și structurat este benefic atunci când clientul este de fapt un alt server care se va folosi de structura mesajului de cerere pentru a specifica toate detaliile interacțiunii cu datele care se găsesc pe celălalt server. De asemenea, același model oferă serverului care răspunde la cerere atât un anumit grad de libertate și personalizare, cât și limite bine definite în care trebuie să funcționeze pentru ca celălalt server, cel care are rol de client să poată să interpreteze și să folosească răspunsul.

Vreau să remarc și să accentuez acest echilibru necesar și foarte bine construit, între

personalizarea permisă și rigiditatea impusă de acest protocol, deoarece eu consider că este unul din motivele pentru care aceste servicii Web s-au putut dezvolta și au putut beneficia de tehnologiile deja dezvoltate.

Un alt aspect important al protocolului de comunicare HTTP este că acesta dispune de un caracter state-less, adică acesta deservește cereri concurente multiple de la diferiți clienți, însă fiecare cerere e considerată independentă de celelalte chiar dacă provin de la același client Web. Lucrul acesta duce la implementarea unor sesiuni la nivelul aplicațiilor web care vor asocia clienților care accesază serverul un identificator unic de sesiune (Session Id). Acest lucru duce la posibilitatea păstrării datelor de-a lungul mai multor cereri succesive ale aceluiași client.

1.2 Servicii Web. Descrierea problemei

Odată cu intrarea într-o eră a interconectivității, o eră în care comunicarea la distanță și informarea în timp real sunt două activități cruciale pentru buna funcționare a societății, dezvoltatorii au realizat că standardele industriei nu pot suporta creșterea accelerată de care se bucura Internetul și spațiul Web. Soluțiile de care era nevoie pentru a suporta un nivel crescut de scalabilitate, de aplicații care trebuiau să fie prezente pe mai multe platforme și slab conectate nu se puteau măpa pe tehnologiile și paradigmile vremii.

Primele aplicații Web constau în pagini cu informații care erau conectate cu alte pagini cu informații asemănătoare pe care utilizatorul le putea folosi și prin care putea naviga cu ușurință datorită hyperlegăturilor. Primul site Web a fost creat în 1991 și afișa informații despre Organizația Europeană de Cercetare Nucleară (CERN), organizație în cadrul căreia s-a dezvoltat Web-ul prin intermediul lui Tim Berners-Lee. La sfârșitul anului 1993 erau raportate 623 de site-uri, după cum se poate vedea într-un studiu făcut de Matthew Gray din cadrul MIT [4]. Același studiu arată că, la sfârșitul anului 1994 erau mai mult de 10,000 de site-uri, creștere care devine exponențială în anii următori și care ne aduce mai aproape de Web-ul pe care îl cunoaștem astăzi.

Astfel că această creștere în dimensiunea Web-ului, a fost acompaniată și de o creștere în complexitatea site-urilor care formau acest spațiu. Site-urile cu informație statică au început să reacționeze la datele introduse de utilizatori, au început să recunoască utilizatorii prin conturi pe care aceștia le puteau accesa odată ajunsi pe aceste site-uri și chiar puteau să își modifice conținutul și să îl

adapteze în funcție de utilizator.

Această puternică dinamizare a conținutului unei pagini a constat într-o popularizare a spațiului Web nemaiîntâlnită. Dacă la început, majoritatea informației care se găsea pe internet era de natură științifică, academică sau guvernamentală, odată cu aplicațiile Web interactive s-au deschis noi posibilități de care dezvoltatorii să profite și prin care utilizatorii să își îmbunătățească stilul de viață sau chiar să își îmbunătățească profesia sau afacerea.

Ca urmare, dezvoltatorii și-au setat țeluri tot mai mari, aplicațiile devenind tot mai complexe și rezolvând tot mai multe probleme din viața de zi cu zi a utilizatorilor. Dacă la început aplicațiile erau specializate în rezolvarea unei probleme specifice pentru utilizatorii săi, cu timpul paradigma s-a schimbat, la modă fiind aplicațiile care rezolvau o plajă cât mai mare de probleme și care ofereau cât mai multe funcționalități, astfel că utilizatorii nu trebuiau să parăsească site-ul pentru a găsi toate informațiile necesare.

Un exemplu pe care îl putem oferi aici este platforma Facebook, care a început prin a oferi utilizatorilor posibilitatea de a se conecta cu alți oameni din întreaga lume. Aceasta a fost scopul inițial, care la momentul lansării a fost o idee inedită, însă pentru a putea răspunde cererilor pieței, dezvoltatorii Facebook au fost nevoiți să ofere utilizatorilor funcționalități ca: jocuri video în cadrul platformei, o modalitate de a fi la curent cu ultimele știri fără a fi nevoiți să părăsească platforma sau un mod pentru utilizatori prin care aceștia să poată să își promoveze afacerea. Toate aceste funcționalități rezolvă probleme reale și oferă valoare aplicației, însă nu fac parte din ideea și scopul inițial al platformei și nu pot fi rezolvate într-o manieră tradițională. Adică aceste funcționalități nu pot fi implementate toate cu o singură echipă de developeri, nu pot fi scrise toate în același limbaj de programare și din cauza diversității de activități pe care le desfășoară, trebuie să folosească multiple tehnologii din diferite arii ale diferitelor industriei.

Deci, unul din pașii care au condus către dezvoltarea și folosirea pe scară largă a serviciilor Web este creșterea complexității aplicațiilor. Astfel că, funcționalitățile importante au trebuit despărțite în aplicații de sine stătătoare care să fie mai apoi folosite și prezentate către utilizator ca o singură platformă în care aceste funcționalități sunt unificate și oferă o experiență completă.

Un alt pas important care a condus spre popularizarea serviciilor Web a fost evoluția telefoanelor mobile și a tabletelor către sistemele complexe care sunt astăzi, adică smartphone-uri. Accesarea paginilor Web de către dispozitivele mobile a fost la început un proces anevoieios deoarece acestea erau create pentru a fi afișate pe computere, dispozitive ale caror dimensiuni erau mult mai

mari decât ale primelor telefoane mobile.

Diferențele dintre telefoanele mobile și calculatoare sunt însă multiple: diferența de rezoluții, memoria limitată a dispozitivelor mobile, tehnologiile limitate care pot rula pe sistemele de operare ale telefoanelor mobile și limitarea introducerii datelor utilizatorului prin interfață care de obicei constă dintr-o tastatură, navigarea în pagină fiind astfel și ea limitată.

Din cauza acestor lucruri, nu existau foarte mulți utilizatori ai Web-ului care îl accesau prin intermediul telefoanelor mobile iar dezvoltatorii aplicațiilor Web nu au fost interesați de această nișă pentru mult timp.

Astfel că, toate optimizările făcute asupra unui site Web, erau făcute nu de dezvoltatorii site-ului ci de către browser-ul care se afla pe dispozitivul mobil, atunci când acesta încerca să afișeze pagina pe telefonul mobil. Se efectuau tot felul de optimizări care filtrau conținutul HTML și afișau doar părțile necesare, astfel că de foarte multe ori site-urile arătau foarte diferit față de reprezentarea lor pe un browser normal, ba chiar arătau diferit de la un browser mobil la altul. Unele browsere mobile nu puteau rula javascript, flash sau alte tehnologii care erau și ele la început și care oferea un mod dezvoltatorilor de a crea aplicații mult mai interactive. Acest lucru făcea ca browserul mobil să fie de foarte multe ori inutil sau mult prea instabil pentru a putea fi folosit.

Toate aceste lucruri se schimbă însă odată cu introducerea termenului și conceptului de smartphone, dispozitiv mobil care este fundamental diferit de telefoanele mobile mai vechi. Istoria telefoanelor mobile începe ca dispozitive al căror scop principal era comunicarea la distanță cu ajutorul undelor audio și care erau foarte mari și cântăreau foarte mult și au încercat să devină cât mai mici și compacte, au încercat mereu să îmbunătățească viața bateriei prin ecrane simpliste și funcționalități reduse.

La toate aceste concepte și standarde se renunță odată cu introducerea smartphone-urilor, decizie care pare contra intuitivă pentru industria de telefonie mobilă dar care se dovedește a fi o decizie inovatoare. Deși nu cântăresc foarte mult, smartphone-urile se străduiesc să devină din ce în ce mai mari, au ecrane mari deoarece este principalul mod de interacțiune cu dispozitivul și cel mai important, nu sunt folosite doar pentru comunicarea audio. Aceste noi dispozitive sunt folosite pentru comunicarea în timp real cu ajutorul Internetului, sunt folosite pentru accesarea Web-ului, pentru accesarea locației utilizatorului și oferirea de informații de navigare cu ajutorul GPS-ului incorporat, etc.

Astfel că smartphone-ul perturba piața tradițională de comunicații aceștia fiind nevoiți să țină pasul cu toate aceste inovații.

Smartphone-urile introduc și sisteme de operare mult mai mature decât cele găsite pe telefoanele mobile precedente și care se aseamănă mai mult cu sistemele de operare găsite pe computere. Cele mai importante sunt Android, sistem de operare dezvoltat de Google și folosit pe majoritatea smartphone-urilor de pe piață, iOS, sistem de operare dezvoltat de Apple și folosit doar pe dispozitivele dezvoltate de ei ca iPhone-ul, iPad-ul și Windows Phone, sistem de operare dezvoltat de Microsoft și folosit pe dispozitivele create de ei.

Toate aceste schimbări perturbă piața de telecomunicații astfel că giganți ca Nokia, dezvoltator Finlandez de telefoane mobile clasice majoritar pe piața de telefonie mobilă este detronat de către dezvoltatori noi care îmbrățișează inovația și reușesc să înteleagă potențialul pieței de smartphone-uri ca: Apple, Samsung, HTC, etc.

Aceste inovații duc la o popularizarea globală rapidă a pieței mobile. Graficul de mai jos arată numărul de utilizatori de smartphone din lume în miliarde, din anul 2014 până în 2018 [5].

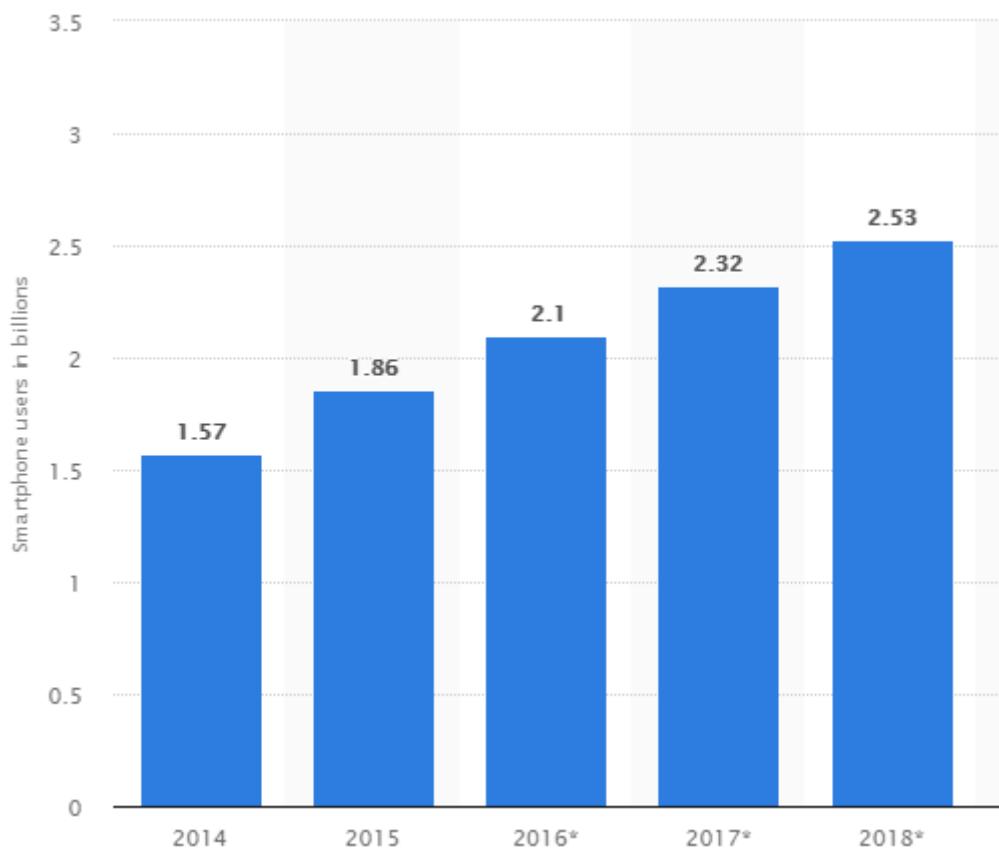


Fig. 3 Statistică a numărului de utilizatori de smartphone [5]

Se creează astfel o piață complet nouă, aceea a aplicațiilor mobile care oferă posibilitatea de inovație și extindere pentru multe companii și pentru foarte mulți dezvoltatori. Astfel că pentru fiecare sistem de operare apare câte un browser nou, browsere mult mai performante și mai capabile decât predecesoarele lor. Printre ele se numără: Google Chrome pentru Android, Safari pentru iOS și Internet Explorer pentru Windows Phone. Se poate observa că toate aceste browsere au câte un corespondent pe sistemele de operare găsite pe computere, Google Chrome pentru Windows, Linux și OSX, Safari pentru OSX și Internet Explorer pentru Windows. Acest lucru arată maturitatea sistemelor de operare introduse odată cu smartphone-urile și dorința companiilor importante de dezvoltare software de a intra pe această piață.

Din cauza numărului mare de utilizatori, în anul 2018 traficul majoritar generat pe Web în întreaga lume a fost făcut folosind un dispozitiv mobil. După cum se poate observa și în graficul de mai jos, care arată procentul traficului Web generat de către dispozitivele mobile din anul 2009 până în 2018 [12].

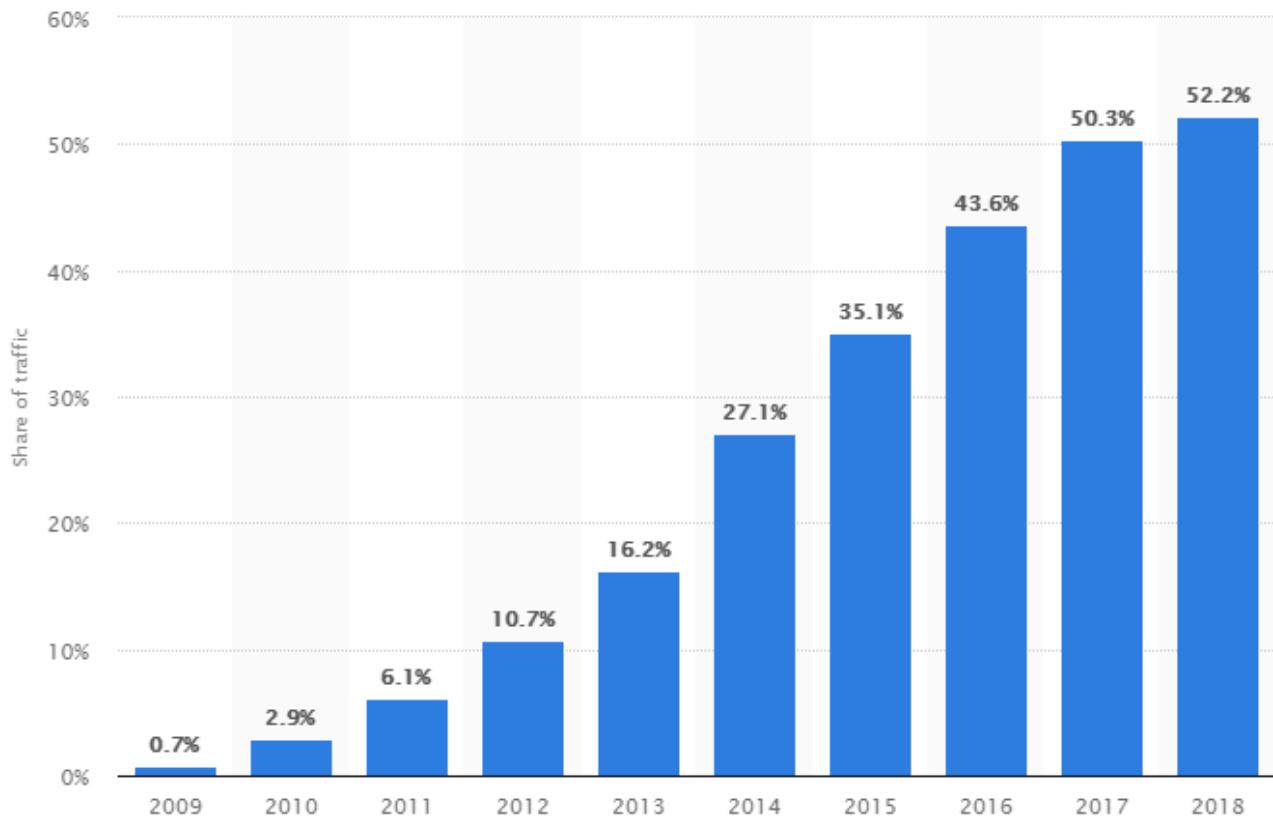


Fig. 4 Statistică a procentului de trafic generat de dispozitivele mobile [12]

Al doilea pas important în dezvoltarea și folosirea la scară largă a serviciilor Web, a fost această creștere masivă în traficul Web generat de către utilizatorii dispozitivelor mobile. Dacă la început dezvoltatorii de aplicații nu făceau niciun efort ca să acomodeze acești utilizatori, după ce numărul lor a continuat să crească acest lucru nu a mai fost posibil. Așa că a apărut aşa numitul termen mobile-friendly, termen ce descrie faptul că o pagină Web este optimizată, ba chiar se poate să arate și să se comporte complet diferit, pentru afișarea pe dispozitivele mobile.

Într-un timp scurt se adoptă în industrie conceptul paginilor Web responsive, care atunci când sunt implementate sunt făcute să răspundă la schimbări de rezoluție ale dispozitivului.

Însă pentru a acomoda toți utilizatorii pieței dispozitivelor mobile pentru unele platforme sau aplicații nu e de ajuns doar să își facă paginile Web să fie responsive. Unele aplicații trebuie portate pe platformele mobile în întregime pentru a livra anumite funcționalități care au nevoie să poată folosi caracteristici native ale sistemului de operare de pe dispozitivul mobil.

Majoritatea acestor aplicații și platforme trebuie să găsească o modalitate de a refolosi logica pe care deja o au implementată și să o integreze în aplicația mobilă.

Desigur, există mai multe soluții pe care dezvoltatorii le-au folosit de-a lungul timpului, nu doar serviciile Web. În continuare o să vorbim despre cele mai importante astfel de soluții și le vom compara pentru a putea găsi punctele forte și punctele slabe.

1.3 Servicii Web RPC, SOAP și REST

După cum am mai menționat, serviciile Web își au începuturile ca fiind o soluție la necesitatea facilitării interacțiunii dintre mașini. O primă soluție, RPC, apare din nevoia dezvoltatorilor pentru apelarea unor operații menite a fi executate la distanță.

1.3.1 RPC (Remote Procedure Call)

Este una dintre cele mai vechi soluții și presupune invocarea unei metode de către un client, pe un server care se află într-un spațiu de adresă diferit. Aceste metode sunt scrise de către programator fără ca el să scrie și detaliile interacțiunii și cum se face transportul datelor. [2] Acest lucru este tratat de către implementarea sistemelor care se ocupă cu acest transfer, cum ar fi spre exemplu Java RMI

pentru limbajul de programare Java. Odată cu apariția paradigmelor de Programare Orientată Obiect, RPC devine RMI (Remote Method Invocation).

Avantajele modelului RPC/RMI:

- Metoda care este apelată de către client se apelează într-un context normal, ca atunci când se apelează o metodă locală
- Performanța unui sistem care se folosește de RPC este în general mai bună decât a unui sistem care folosește un serviciu Web SOAP sau REST
- Se pot folosi aceleași modele de darte atât în aplicația client, cât și în aplicația server

Dezavantajele modelului RPC/RMI:

- Odată cu creșterea complexității aplicației, modelul RPC/RMI devine tot mai greu de întreținut și de extins
- Nu există un standard bine definit și există multe implementări ale acestui protocol care nu sunt capabile să comunice între ele din cauza unor diferențe subtile
- Cele mai multe astfel de sisteme facilitează comunicarea între aplicații care sunt scrise în același limbaj de programare

1.3.2 SOAP (Simple Object Access Protocol)

SOAP este un protocol folosit pentru transmiterea mesajelor între două mașini, structurate cu ajutorul XML (Extensible Markup Language).

XML (Extensible Markup Language)

XML este un limbaj de marcare care definește un set de reguli pentru codificarea documentelor într-un format care este ușor de interpretat atât de către oameni, cât și de calculatoare. Putem considera XML ca fiind un standard internațional de descriere a datelor/resurselor electronice.

Termenul de marcat (markup) era folosit, înaintea apariției XML, pentru a descrie anumite adnotări, găsite de obicei ca note marginale în cadrul textelor. Aceste marcate aveau ca scop oferirea de indicații pentru un anumit bloc de text, ca spre exemplu formatarea acestuia, cum ar trebui acest bloc să fie listat sau chiar dacă blocul trebuie sau nu să apară în documentul final.

Un limbaj de marcare oferă un set de specificații utilizate mai apoi de dezvoltatori în codificarea

datelor pe care le vor transmise între sisteme. Astfel că, XML este o soluție de specificare universală a datelor dintr-o aplicație care descrie atât structura, cât și comportamentul acestora.

Protocolul SOAP encapsulează informațiile într-un plic (envelope) care este format din două elemente, antent (Header) și (Body). În antet, se pot pune directive sau informații de care serverul să se ajute înaintea procesării mesajului propriu-zis, ca spre exemplu: informații despre autentificare sau informații despre proveniența mesajului. Datele aplicației sunt introduse în corpul mesajului, el fiind un simplu container pentru informația transmisă în format XML. Datele descrise trebuie să urmeze structura stabilită, altfel procesarea nu poate fi făcută de către server, însă dacă acest lucru este respectat, nu există alte restrictii cu privire la datele ce pot fi trimise în mesaj, alta decât că este nevoie să fie formatace ca și text/xml. [2]

Avantajele protocolului SOAP:

- Față de modelul RPC, din cauză că SOAP folosește HTTP nu este nevoie de configurații adiționale de proxy sau fierwall aduse serverului
- SOAP nu depinde de niciun limbaj de programare astfel că este posibilă comunicarea între două aplicații scrise în limbi diferite
- SOAP este relativ simplu, deoarece folosește XML pentru reprezentarea datelor fiind ușor de citit și de către oameni nu doar de calculatoare. Acest lucru se dovedește foarte folositor atunci când există o eroare în mesajul generat de aplicație și e nevoie ca programatorul să depaneze mesajul
- SOAP este extensibil, astfel că există posibilitatea codificării oricărui fel de dată sau informație atât timp cât se respectă instrucțiunile
- La acest moment, SOAP este destul de matur și folosit încât nu există probleme ascunse sau neajunsuri de care să nu se știe, astfel că nu există posibilitatea începerii dezvoltării unei aplicații care folosește SOAP iar pe parcurs realizarea că acesta are probleme
- SOAP este potrivit pentru calculul distribuit
- SOAP definește un standard de securitate propriu care este foarte sigur

Dezavantajele protocolului SOAP:

- Din cauză că limbajul XML este foarte explicit, mesajele devin de foarte multe ori foarte mari dacă se consideră un set rezonabil de date. Astfel că lungimea de bandă folosită pentru

transmiterea mesajului este foarte mare lucru care poate fi o problemă pentru aplicațiile cu foarte multe date

- Mesajele XML sunt procesate mai greu din cauza faptului că sunt atât de descriptive. Astfel că SOAP cu ajutorul XML reușește să creeze un standard inteligibil atât pentru oameni cât și pentru calculatoare. Acest lucru își spune cuvântul asupra categoriei din urmă, calculatoarele, care nu au nevoie de o verbozitate ridicată pentru a înțelege, datele având foarte multe informații redundante

1.3.3 REST (Representational State Transfer)

REST este un stil arhitectural care expune informații despre aplicație sub forma resurselor pe care le manipulează și oferă clientului posibilitatea de a modela resurse existente sau de a crea resurse noi. Pentru ca o aplicație să fie RESTful ea trebuie să urmeze un set de reguli sau instrucțiuni, formalizate prima dată de către părintele acestui stil arhitectural, Roy Fielding în teza sa de doctorat, în anul 2000.

Acesta propune că arhitectura Web-ului este determinată de constrângerile care se impun asupra elementelor componente și explică faptul că proprietățile oferite de aceste constrângeri reflectă conceptul de stil arhitectural [6]. Astfel că, dacă se aplică constrângeri suplimentare Web-ului, putem ajunge la un nou stil arhitectural care să reflecte o arhitectură modernă, mai aproape de nevoile dezvoltatorilor.

Constrângerile pe care Fielding le preia din Web sunt:

1. Constrângerile legate de stilul arhitectural client-server pe care Web-ul funcționează. Principul care stă la baza acestui stil arhitectural este principiul separării responsabilităților. Astfel că, acestor două componente li se permite să se dezvolte independent una de celalaltă, decupându-le, ceea ce înseamnă în același timp și că pot exista mai multe aplicații client și/sau server [6]
2. Constrângerea pe partea de comunicare este aceea că Web-ul în esență, se folosește de o paradigmă de comunicare, în care nu se stochează o stare (stateless), adică serverul nu are informații despre clientul care face cererea și răspunde cu informații complete fiecărei cereri [6]
3. Pentru eficiență, se introduce constrângerea de cache, care oferă posibilitatea clientului de a servi o resursă dintr-un mediu de stocare propriu, dacă acesta a mai interacționat cu acea resursă, scăpându-se astfel de unele interacțiuni [6]

Constrângerile pe care Fielding le aduce în plus pentru a crea REST sunt:

1. Cel mai important principiu este acela de a avea o interfață uniformă, comună între componente. Această constrângere simplifică stilul arhitectural, oferă independență serviciului față de implementare, lucru ce ajută dezvoltarea independentă [6]
2. Sistemul trebuie să fie compus din niveluri sau straturi. Această constrângere face posibil ca un nivel logic al aplicației să fie influențat doar de nivelul imediat următor aducând un plus de scalabilitate sistemului [6]
3. Ultima constrângere este una optională și vizează stilul de proiectare al aplicațiilor code-on-demand [6]

Logica din spatele construirii REST este aceea de a crea un model arhitectural care să servească drept un cadru premergător pentru un spațiu Web standardizat în care comunicarea dintre două mașini să se efectueze într-un mod eficient. REST folosește protocolele și conceptele mature din Web, ca HTTP și URI, cache și metode de securizare a datelor trimise ca TLS și SSL.

Avantajele REST:

- Mesajele trimise în REST sunt de obicei mai simple și nu există atât de multe informații redundante. Asta face ca lungimea de bandă folosită să fie mai mică față de serviciile Web SOAP spre exemplu
- REST folosește concepte familiare pentru interacțiunile care se petrec, adică se folosesc concepte deja existente în Web care doar sunt extinse sau adaptate
- Similar, concepte ca: integritatea datelor transmise sau securitatea lor este garantată de tehnologii existente, bine cunoscute
- Definirea comunicării în REST se face mai ușor decât în SOAP, deoarece interacțiunea și logica care se aplică unei resurse este definită de URI, operația simplificându-se la un singur verb HTTP ca: GET, POST, PUT, DELETE etc..
- REST nu depinde de niciun limbaj de programare
- Pentru cineva care ar dori să creeze un serviciu Web care să fie în concordanță cu principiile REST, există în foarte multe limbi de programare nenumărate tehnologii care să accelereze dezvoltarea

Dezavantajele REST:

- Caracterul general pe care REST îl propune oferă anumite restricții sistemelor și implică o complexitate mai mare ce trebuie luată în considerare atunci când se dezvoltă aplicația
- Odată cu moștenirea proprietăților și a tehnologiilor benefice din Web, se moștenesc și neajunsurile și problemele

Pentru structurarea datelor se poate folosi atât XML descris anterior cât și JSON (JavaScript Object Notation).

JSON (JavaScript Object Notation)

JSON este un format pentru structurarea datelor simplu și care nu are informație redundantă ci doar date structurate într-o manieră ușoară atât pentru mașini să îl genereze și să îl analizeze cât și de oameni să îl înțeleagă și verifice.

JSON are două concepte de bază:

- O colecție de perechi nume – valoare care se rezolvă la nume de câmpuri/variabile și obiecte în limbajele de programare
- O listă ordonată de valori

Aceste elemente se pot compune pentru a descrie obiecte și colecții complexe.

Motivele pentru care am ales crearea aplicației server ca un serviciu Web REST se găsesc printre avantajele acestui stil arhitectural, dar și pentru că apreciez caracterul general pe care îl promovează și folosirea inteligentă a protocolului HTTP pentru comunicare.

2. Tehnologii folosite

Tehnologiile folosite vor fi împărțite în două categorii după aplicația în care s-au folosit, adică vor fi tehnologii pentru: aplicația client și pentru aplicația server. În acest capitol vom include și detalii de configurare pentru fiecare dintre aplicații. Aplicația server este un serviciu Web REST care expune resurse pe care aplicația client, care este o aplicație mobilă, le poate manipula și prezenta utilizatorilor. Aceste două aplicații sunt complementare și doar împreună pot forma aplicația Parker.

2.1 Tehnologii folosite pentru aplicația server

Aplicația server este un serviciu Web REST. Adică, este o aplicație Web MVC scrisă în limbajul de programare Java care cu ajutorul frameworkurilor Spring, Hibernate și a containerului Tomcat, reușește să se adapteze stilului arhitectural REST, respectând constrângările impuse.

Pentru structurarea datelor transmise și primite de către aplicație se folosește formatul JSON deoarece este mai eficient, folosind mai puțină lungime de bandă și de asemenea fiind mai ușor de interpretat de către Spring Framework, care folosește un mecanism propriu de serializare și deserializare a obiectelor JSON. De asemenea, aplicația mobilă este scrisă cu ajutorul frameworkului React Native în care limbajul de programare folosit este JavaScript, în cadrul căruia s-a dezvoltat structura JSON, și care are suport nativ pentru aceste obiecte.

2.1.1 Spring

Spring este o platformă open-source care oferă programatorilor infrastructuri pentru diferitele tipuri de aplicații Java pe care aceștia trebuie să le dezvolte. Spring dispune de mai multe proiecte care s-au dezvoltat de-a lungul timpului ca spre exemplu: Spring Framework, Spring Security, Spring Web Services, Spring Boot, Spring Data, Spring Cloud etc...

Toate aceste proiecte Spring au ca scop implementarea unor tehnici demonstate în industrie care să ușureze munca programatorilor și să le garanteze scalabilitate, securitate sau chiar eficiență, prin reutilizarea codului deja scris.

Pentru aplicația noastră am folosit: Spring Framework, Spring Web MVC, Spring Security și Spring

Session.

Spring Framework

Este proiectul inițial creat în 2003 în corespondență cu primele specificații J2EE care formalizează aplicațiile Java enterprise și cele mai bune practici pentru crearea și menținerea lor. Cele mai importante concepte implementate în Spring Framework sunt Dependency Injection și Inversion of Control.

Dependency Injection descrie interacțiunea dintre obiectele aplicației care colaborează pentru a crea logica aplicației, prin urmare depindând una de cealaltă. Pentru ca aceste dependințe să poată fi controlate și corect folosite ele trebuie create într-un context cu specificații clare și care să fie independent. Astfel că, în Spring există componenta IoC (Inversion of Control) care adresează această sarcină prin oferirea unei implementări a unui „Context al aplicației” [7]. În acest context se definesc obiectele cu dependințele lor, informații care descriu un bean – conceptul de bază din acest context. La inițializarea sistemului, aceste bean-uri sunt analizate iar dependințele lor sunt rezolvate la alte beanuri din context, fiind injectate pentru folosire. Descrierea acestor bean-uri se poate face prin fișiere XML sau chiar adnotări aflate chiar în codul Java, după preferința dezvoltatorului. Cu ajutorul acestor informații se garantează corectitudinea obiectelor cu care un alt obiect trebuie să interacționeze la momentul inițializării sistemului și pe tot parcursul timpului de funcționare al acestuia. De asemenea, Spring poate să optimizeze crearea unor obiecte doar în momentul folosirii lor sau chiar refolosirea lor după ce au fost odată create, oferind un plus de optimizare și eficiență.

În aplicația dezvoltată pentru această lucrare, majoritatea configurațiilor beanurilor sunt făcute folosind adnotări, ca în exemplul de mai jos, însă există și configurații făcute cu ajutorul fișierelor XML ca spre exemplu, unele configurații pentru Spring Security.

Adnotarea `@Autowired` oferă informații contextului Spring care atunci când încearcă să creeze un obiect de tipul `ParkingSpotServiceImpl` va știi că depinde de obiectele `ParkingSpotDao`, `UserService` și `ReservationService`. Toate aceste obiecte sunt mai apoi căutate în context iar dacă nu au fost create încă, vor fi create prin aceeași metodă iar apoi injectate în obiectul pe care am vrut să îl creem la începutul exemplului.

```

31     @Service
32     @Transactional
33     public class ParkingSpotServiceImpl implements ParkingSpotService {
34
35         @Autowired
36         private ParkingSpotDao parkingSpotDao;
37
38         @Autowired
39         private UserService userService;
40
41         @Autowired
42         private ReservationService reservationService;

```

Fig. 6 Injectarea dependințelor într-un Service al aplicației

În imaginea de mai jos este un fișier XML de configurare pentru componenta Spring Security, în care pentru a putea avea un filtru de autentificare care să folosească printre dependințele sale obiecte create în aplicație, deoarece cele pe care Spring le oferă nu sunt conforme cu nevoile aplicației. Astfel că, folosind identificatorul „authenticationFilter” care deja este configurat, acea informație se suprascrie cu informația care o să configureze noul obiect cu dependințele potrivite.

```

35     <security:authentication-manager alias="authenticationManager">
36         <authentication-provider ref="customAuthenticationProvider" />
37     </security:authentication-manager>
38
39     <beans:bean id="authenticationFilter"
40         class="org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter">
41         <beans:property name="authenticationManager" ref="authenticationManager" />
42         <beans:property name="postOnly" value="false" />
43         <beans:property name="authenticationSuccessHandler" ref="customSuccessHandler" />
44         <beans:property name="filterProcessesUrl" value="/user/login" />
45     </beans:bean>
46
47     <beans:bean id="customSuccessHandler"
48         class="com.parker.util.authentication.CustomSuccessHandler" />
49

```

Fig. 7 Configurarea XML a Spring Security din aplicație

De asemenea, Spring Framework este modular, lăsând astfel posibilitatea dezvoltatorilor să aleagă doar componentele de care au nevoie în aplicație, fără a-i obliga să utilizeze suita completă de proiecte sau module. Acest lucru face ușoară dezvoltarea, atât a aplicațiilor complexe, care au nevoie de foarte multe module pentru a implementa logica necesară, dar și a aplicațiilor mai simple care au

nevoie doar de anumte componente. Acest lucru ajută dezvoltatorii să se focuseze pe scrierea codului pentru componentele ce le fac aplicația unică, fără să trebuiască să rescrie standarde din industrie și astfel pierzând timp important din perioada de dezvoltare.

Spring Web MVC

Spring Web MVC sau pe scurt Spring MVC este infrastructura pusă la dispoziția dezvoltatorilor pentru crearea aplicațiilor Web, construită cu ajutorul Servlet API și executată cu ajutorul unui Servlet container.

MVC (Model View Controller)

MVC (Model View Controller) este un stil arhitectural care împarte aplicația în trei părți interconectate, separând astfel reprezentarea informației pe care utilizatorul o primește și pe care utilizatorul o oferă aplicației ca date de intrare, față de reprezentarea internă pe care această informație o are în logica aplicației.

Componentele MVC sunt:

- Modelul corespunde logicii aplicației care se ocupă cu manipularea datelor, având ca responsabilități acțiuni și operații care se aplică pe datele respective și eventuala comunicare și stocare a acestor date într-un sistem de stocare ca baza de date
- View-ul se ocupă cu prezentarea informațiilor utilizatorului într-un mod ușor de înțeles pentru el. De asemenea, View-ul este modalitatea de interacțiune a utilizatorului cu aplicația, datele de intrare fiind procesate prin intermediul acestuia
- Controller-ul este o interfață, un liant, între componentele Model și View care procesează datele care trebuie expuse pe View și pregătește datele ce trebuie procesate de către Model venite de la utilizator

Pentru a urma acest stil arhitectural, Spring este proiectat, ca multe alte framework-uri Web, în jurul tiparului Front Controller, în care un Servlet principal, numit în cadrul Spring MVC, DispatcherServlet, primește toate cererile, iar mai apoi le delegă componentelor care au fost configurate pentru a putea procesa corect cererea.

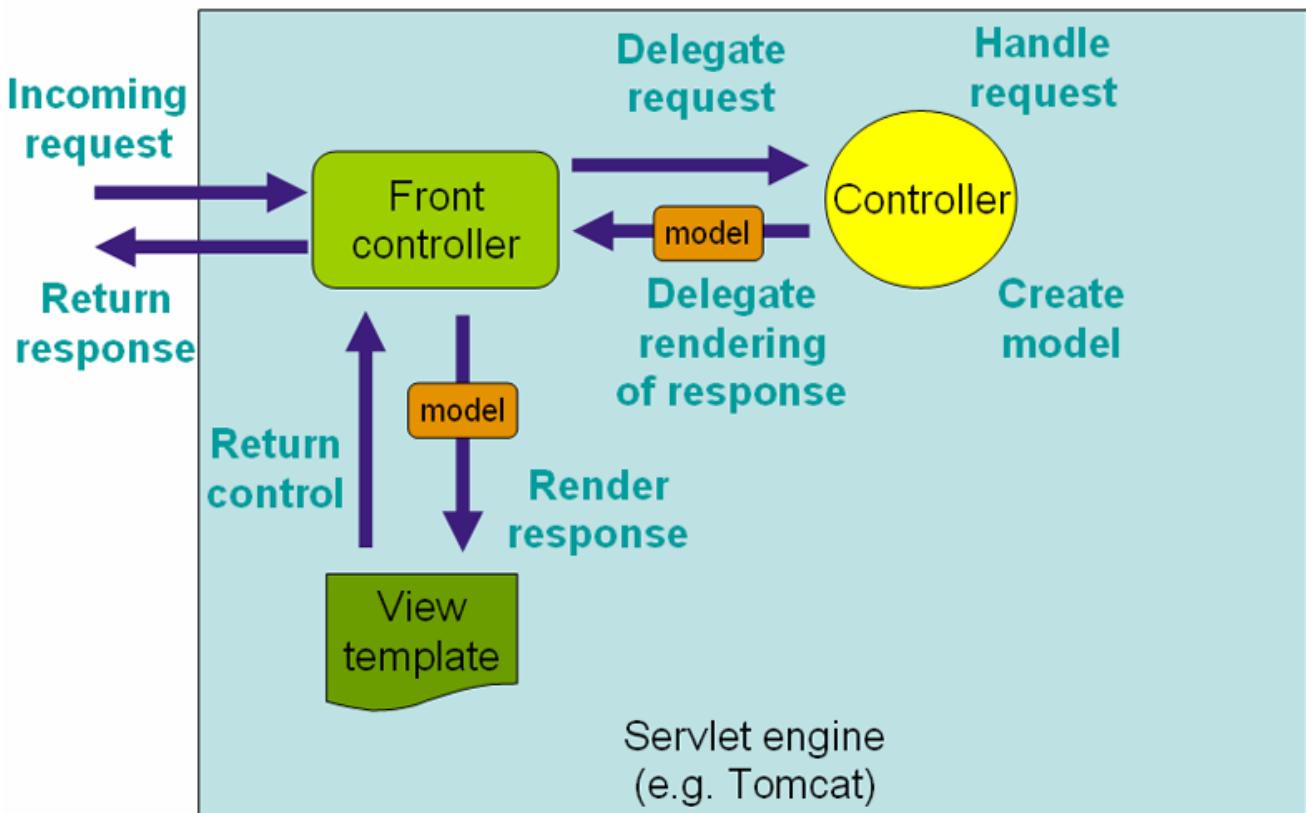


Fig. 8 Modelul Front Controller [11]

În aplicația noastră configurăm acest DispatcherServlet printr-un fișier XML iar logica pentru fiecare tip de cerere, adică în cazul acesta pentru fiecare tip de acțiune posibilă pe o anumită resursă, este împărțită în Controllere diferite care se aplică unei rute, adică unui URI. Acest lucru este posibil folosind adnotări care se pot aplica la nivelul unei întregi clase, la nivelul unei metode, sau chiar la nivelul unui parametru.

Adnotările frecvent folosite în aplicație atunci când se folosește Spring MVC sunt:

RequestMapping este o adnotare ce poate fi aplicată atât unei clase cât și unei metode. Adnotarea este folosită pentru a specifica metodele sau clasele care pot să proceseze o cerere Web. Oferă mai multe atribute care specifică diferențe de comportament.

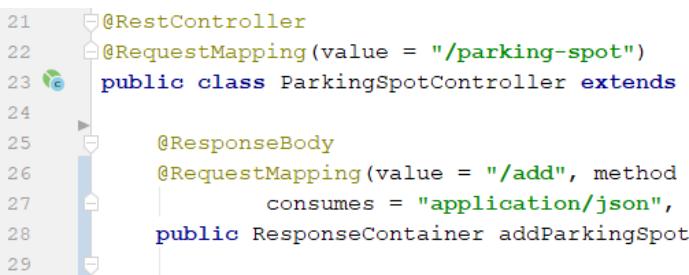
- Atributul Path descrie ruta sau calea, adică URL-ul pe care dacă utilizatorul îl accesează metoda este folosită
- Atributul Method descrie pentru ce metode HTTP poate fi folosită această metodă. Pot exista

metode care au același Path, adică pot răspunde aceluiași URL dar sunt folosite pentru metode diferite

- Atributele Consumes și Produces precizează ce tip de informație se așteaptă de la cerere, respectiv ce tip de informație se va trimite la răspuns. Aici poate fi vorba de text/plain, text/html, application/xml etc.

Controller sau **RestController** sunt adnotări ce se găsesc la nivelul unei clase și atunci când sunt folosite specifică faptul că această clasă are rolul unui Controller în care se găsesc metode ce pot fi folosite în procesarea cererilor. Un Controller Rest este un Controller care răspunde cu un mesaj în diferite forme, adnotarea fiind o combinație între adnotarea Controller și ResponseBody.

RequestBody este o adnotare ce se aplică unui parametru și specifică faptul că informația din corpul unei cereri HTTP trebuie să poată să fie deserializată pentru a crea un obiect Java.



```
21  @RestController
22  @RequestMapping(value = "/parking-spot")
23  public class ParkingSpotController extends AbstractController {
24
25      @ResponseBody
26      @RequestMapping(value = "/add", method = RequestMethod.POST,
27                      consumes = "application/json", produces = "application/json")
28      public ResponseContainer addParkingSpot(@RequestBody ParkingSpotData parkingSpotData,
29                                              BindingResult bindingResult) {
```

Fig. 9 Exemplu de Controller din aplicație

Spring Security și Spring Session

Spring Security este infrastructura Spring oferită pentru configurarea autentificării utilizatorilor aplicației și controlul punctelor de acces în aplicație. În aplicația creată în cadrul acestei lucrări, Spring Security este folosită împreună cu Spring Session care este API-ul creat de Spring pentru a oferi moduri de a manipula sesiunea unui utilizator în cadrul aplicației.

2.1.2 Hibernate și JPA (Java Persistence API)

JPA (Java Persistence API) este o colecție de specificații Java pentru accesarea, stocarea și manipularea obiectelor POJO (Plain Old Java Objects) dintr-o aplicație și obiectele dintr-o bază de date relațională. JPA este o specificare a unor concepte și pentru a putea fi folosită este nevoie de o implementare. Aici intră în discuție Hibernate, pe care noi îl folosim ca implementare a JPA.

Hibernate ORM (Object Relational Mapping) este un framework care are ca scop crearea unor conexiuni între obiectele Java din aplicație și tabelele din baza de date relațională folosită ca mod de stocare. Soluțiile de tipul Hibernate ajută dezvoltatorii să decupleze aplicațiile de sistemele de stocare prin oferirea unei interfețe de manipulare a obiectelor fără a cunoaște detalii de implementare sau chiar tipul bazei de date din spatele acestei interfețe.

Aceste relații pot fi descrise fie cu ajutorul fișierelor XML sau cu ajutorul adnotărilor. În aplicația dezvoltată în cadrul acestei lucrări relațiile sunt descrise exclusiv folosind adnotările.

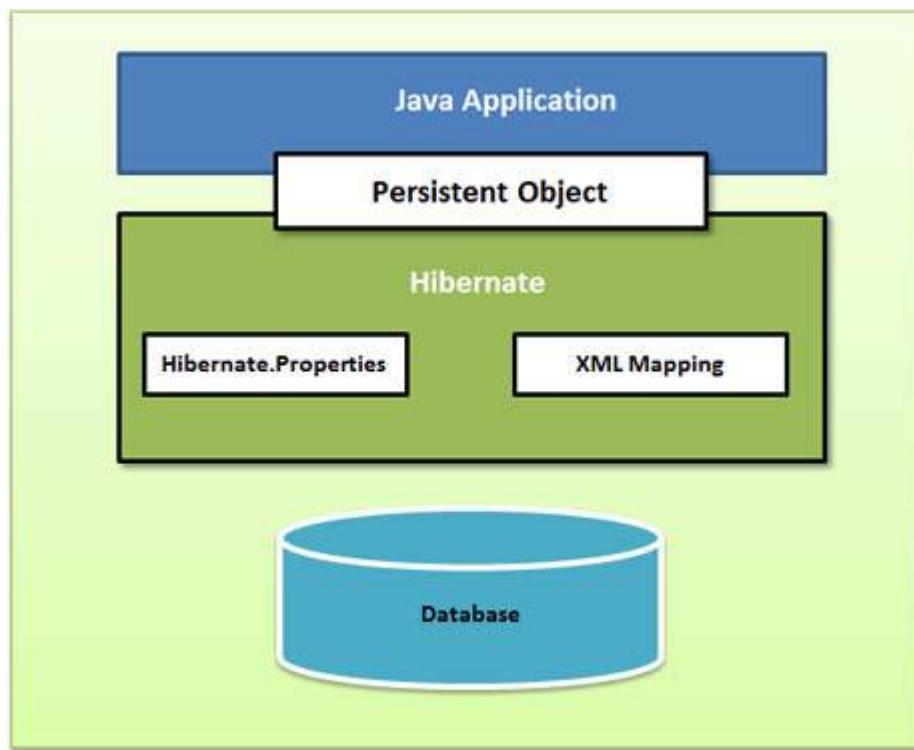


Fig 10. Arhitectura Hibernate într-o aplicație Java [13]

În Hibernate există conceptul de Sesiune, care este un obiect folosit pentru accesarea unei conexiuni fizice cu baza de date, obiectele persistente din aplicație, adică entitățile, sunt mereu salvate și preluate din baza de date cu ajutorul unui obiect de sesiune. [13]

Adnotările folosite în aplicație sunt:

- Entity este o adnotare care se găsește la nivelul unei clase și marchează o clasă ca fiind un bean entitate
- Cu ajutorul adnotării Table se pot specifica detalii despre tabelul din baza de date prin care obiectul va fi reprezentat
- Adnotările Id și GeneratedValue sunt găsite la nivelul unui atribut al unei clase și specifică detalii despre cheia primară a unei entități. Id specifică faptul că atributul este cheie primară iar GeneratedValue determină strategia de generare potrivită tipului de dată al atributului
- Column specifică detalii despre atribut și cum va fi el construit în baza de date ca spre exemplu dacă poate fi null, numele acestuia, lungimea sau unicitatea

Deoarece relațiile dintre obiectele din programarea orientată obiect și cele din bazele de date relationale sunt apropiate în esență dar totuși destul de diferite, intervine o nevoie a specificării acestor relații în codul Java prin adnotări sau în fișiere XML.

Adnotările respective sunt:

- ManyToOne descrie o relație în care entitatea, clasa, în care acastă adnotare apare este membrul care este multiplu în relație și care de obicei se transpune cu o cheie străină în tabela din baza de date
- OneToOne descrie o relație de unu la unu între două entități care sunt aggregate
- OneToMany este adnotarea care apare în membrul care este singular în relație și descrie de obicei o colecție de elemente formate din obiectele de tipul membrului care este multiplu în relație
- ManyToMany este o adnotare care descrie o relație în care ambele entități dețin o colecție de obiecte de tipul entității celeilalte. De obicei această adnotare este redusă la o nouă tabelă care să administreze elementele relației în baza de date

```

6  @Entity
7  @Table(name = "parkingspot")
8  public class ParkingSpot {
9    @Id
10   @GeneratedValue
11   private Long id;
12
13   @Column(nullable = false)
14   private Double latitude;
15
16   @Column(nullable = false)
17   private Double longitude;
18
19   @Column(nullable = false)
20   private String address;
21
22   @Column(nullable = false)
23   private String activeDaysIntervals;
24
25   @ManyToOne
26   private User user;
27
28   @OneToMany
29   private List<Reservation> reservations;

```

Fig. 11 Exemplu de entitate descrisă cu adnotări din aplicație

Pentru salvarea, stergerea și manipularea entităților din aplicația noastră se folosesc clase DAO (Data Access Object) în care se deschid Sesiuni Hibernate cu ajutorul cărora se execută operația dorită în baza de date. De asemenea, pentru operațiile mai complexe se folosește limbajul HQL (Hibernate Query Language) pentru crearea și executarea interogărilor complexe.

DAO (Data Access Object)

Obiectele DAO sunt clase Java care asigură o interfață pentru comunicarea cu sistemul de persistență folosit, ca spre exemplu baza de date.

HQL (Hibernate Query Language)

HQL este un limbaj de interogare dezvoltat pentru Hibernate și similar cu SQL în aparență. HQL însă este complet orientat obiect și înțelege concepte ca moștenire, polimorfism și asociere. De asemenea, HQL este un intermediar între aplicația Java și limbajul de interogare al bazei de date, în care până la urmă toate interogările sunt traduse și executate.

```
13  @Repository
14  public class ParkingSpotDaoImpl implements ParkingSpotDao {
15
16      @Autowired
17      private SessionFactory sessionFactory;
18
19      @Override
20      public Long save(ParkingSpot parkingSpot) {
21          Session session = sessionFactory.getCurrentSession();
22          return (Long) session.save(parkingSpot);
23      }
24
25      @Override
26      public void update(ParkingSpot parkingSpot) {
27          Session session = sessionFactory.getCurrentSession();
28          session.update(parkingSpot);
29      }
30
31      @Override
32      public ParkingSpot find(Long id) {
33          Session session = sessionFactory.getCurrentSession();
34          return (ParkingSpot) session.createQuery("FROM ParkingSpot p WHERE p.id=:id").
35              setParameter("id", id).uniqueResult();
36      }

```

Fig. 12 Exemplu de DAO în aplicație

2.2 Tehnologii folosite pentru aplicația client

Aplicația client este o aplicație mobilă dezvoltată cu ajutorul framework-ului React Native. Scopul acestui framework este de a oferi ușurință în crearea aplicațiilor mobile independente de platformă, în limbajul JavaScript și peste sistemul premergător React, de unde se moștenesc atât componente cât și concepe.

2.2.1 React

React sau ReactJS este o bibliotecă pentru crearea interfețelor unei aplicații care sunt expuse utilizatorilor. Conceptul principal din React sunt Componentele, structuri unitare ce încapsulează atât logică cât și o parte de prezentare pentru utilizator pentru o anumită funcționalitate sau chiar pentru o parte componentă a unei funcționalități. Granularitatea componentelor este la latitudinea dezvoltatorului aplicației.

Aceste componente pot fi funcționale, adică partea de logică și prezentare a componentei este încapsulată într-o funcție, sau componenta poate fi reprezentată de o clasă în care partea de prezentare poate fi scrisă cu ajutorul JSX (Javascript eXtension).

Un concept important care există pentru componentele din React este starea (state) care este o multitudine de atribute ce descriu comportamentul componentei la un moment în timp. Setarea atributelor stării se face cu metoda this.setState care face ca o componentă să trebuiască să fie redesenată, fiindu-i astfel reapelată metoda render().

```
1 <class HelloWorldComponent extends React.Component {  
2   constructor(props) {  
3     super(props);  
4     this.state = {  
5       message: "Hello World!"  
6     };  
7   }  
8  
9   render() {  
10    return (  
11      <h1>{this.state.message}</h1>  
12    );  
13  }  
14}
```

Fig. 13 Exemplu de componentă React

2.2.2 Redux

Există foarte multe metode de a manipula starea și de a lucra cu aceasta pentru a oferi aplicației dinamism, însă cele mai multe probleme apar atunci când componentele trebuie să interacționeze între ele, depinzând de starea în care se află. Astfel că apare Redux care rezolvă această problemă. React Redux reprezintă un container de stare pentru aplicații JavaScript și este des folosit împreună cu React.

Redux apare ca răspuns la nevoile dezvoltatorilor pentru aplicații JavaScript care au toate funcționalitățile într-o sigură pagină pe care o prezintă utilizatorului și care sunt tot mai complexe. Redux propune menținerea stării unei aplicații printr-un model care reprezintă un obiect simplu, dar care nu dispunde de metode get sau set pentru a nu putea schimba starea arbitrar ci doar prin intermediul unei metode care să asigure consistență. [9]

Astfel apare conceptul de acțiune, care este singura metodă prin care se poate modifica starea aplicației cu ajutorul unei funcții numite reducer.

Toate aceste concepte fac Redux să fie o bibliotecă importantă pentru dezvoltarea aplicațiilor JavaScript web sau mobile care oferă funcționalitățile într-o singură pagină și care acoperă o plajă extinsă de servicii.

2.2.3 React Native

React Native este un framework care oferă posibilitatea dezvoltării aplicațiilor mobile native cu ajutorul JavaScript și biblioteca React. Aplicațiile scrise cu ajutorul React Native sunt independente de platformă, Android sau iOS, făcând astfel munca programatorilor mult mai ușoară și rapidă, putând astfel să scrie aplicația pentru orice smartphone.

Avantajele React Native sunt:

- Timpul de dezvoltare scăzut atunci când vorbim de aplicații ce trebuie să ruleze atât pe Android cât și pe iOS
- Aplicațiile dezvoltate în React Native nu au nevoie de programatori care să știe mai multe limbi de programare cum ar fi Swift/Objective-C sau Android
- Proiectul este open-source și comunitatea de dezvoltatori care s-au strâns de-a lungul celor 3 ani de existență este dispusă să ajute și să împărtășească cunoștințele dobândite prin dezvoltarea

aplicațiilor proprii

- React Native nu este un framework prin care să se creeze aplicații mobile hibrid, care au o performanță mult sub aplicațiile native. React Native creează aplicații native care au o performanță apropiată de performanța aplicațiilor mobile

Dezavantajele React Native sunt:

- Sistemele pe care aplicațiile scrise în React Native trebuie să rulezse, Android și iOS, sunt foarte diferite, aşa că, în anumite cazuri ceea ce oferă frameworkul nu este de ajuns fiind nevoie de cod scris direct în Android sau Swift/Objective-C care să poată să integreze unele funcționalități
- Deși comunitatea este de ajutor, faptul că este o tehnologie atât de nouă se resimte atunci când problemele apar de la o versiune la alta, sau când diferite erori intervin fără o cauză care să poată să fie ușor rezolvată de către dezvoltator, ci mai degrabă este o neglijență a frameworkului

Din cauza faptului că aplicațiile scrise cu ajutorul JavaScript și React Native sunt independente de platformă, sunt performante și există o documentație solidă atât pentru concepte cât și pentru eventualele probleme, aplicația mobilă Parker este scrisă cu ajutorul acestora.

3. Aplicația Parker

În era informației și a comunicării, există foarte multe procese din viața de zi cu zi pe care tehnologia încă nu le-a îmbunătățit sau în cadrul cărora mai există loc de îmbunătățire. Acesta este cazul și pentru procesul de parcare al mașinii într-un oraș aglomerat, la o oră de vârf. Aplicația Parker vine ca un pas premergător la crearea locurilor de parcare complet automatizate și inteligente, cu senzori care pot să transmită în timp real informații despre disponibilitatea locului de parcare care să comunice direct cu sistemele de navigație ale mașinilor, care și ele ar putea fi autonome.

Parker propune o soluție bazată pe colaboarea conducătorilor auto care înțeleg problema și ineficiența locurilor de parcare actuale. În acest capitol vom prezenta aplicația Parker din perspectiva unui utilizator, fără să revenim la părțile teoretice sau detaliile de implementare pe care le-am expus în celelalte capitole.

3.1 Arhitectura

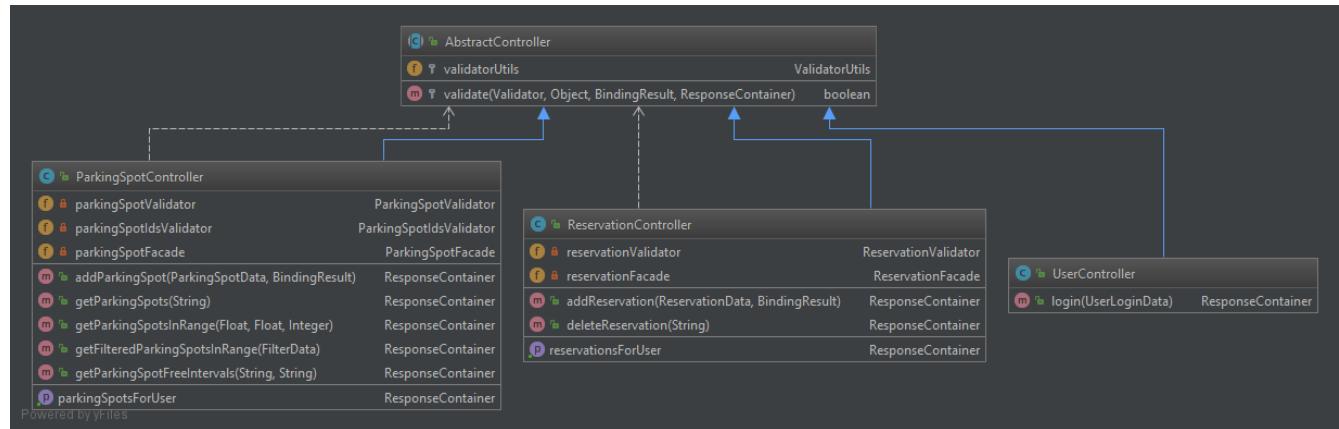
Serviciul Web este scris ca o aplicație Web Java cu ajutorul Spring Framework, Hibernate ORM și care este bazată pe o arhitectură stratificată, în care fiecare “strat” are un rol bine definit pe care îl vom prezenta în continuare. De asemenea, pentru o mai bună prezentare a acestor concepte vom oferi exemple de cod din aplicație relevante, dar care în final să prezinte o funcționalitate existentă în aplicație.

1. Controller

Acesta este stratul cel mai “înalt” al aplicației și are ca rol expunderea de clase și metode ce pot fi folosite pentru a răspunde cererilor făcute de către clienți. Cererile făcute de către clienți, care sunt alte aplicații, trebuie să corespundă cu ce se așteaptă serviciul să primească. Astfel că, multe dintre aceste restricții sunt specificate prin adnotări oferite de Spring Framework, cum ar fi adnotările: RestController, RequestMapping sau RequestBody. Dacă aceste restricții nu sunt îndeplinite de către client atunci când acesta face cererea, Spring Framework răspunde cu un cod de status 400 care spune că cererea este incorectă din punct de vedere sintactic. Însă dacă cererea este formulată corect, este nevoie de o verificare a datelor trimise de către client, pentru a putea fi trimise mai departe spre

procesare. Aici intervin clasele Validator din pachetul domain al aplicației despre care vom vorbi în cadrul aceluiași strat al aplicației.

În figura de mai jos se poate observa structura claselor din pachetul controller și dependințele dintre acestea.



Există clasa abstractă `AbstractController` pe care toate celelalte clase o extind și care dispune de metoda **validate** care primește ca parametru obiectul primit pe cerere pentru a fi validat, validatorul care se folosește pentru validare, `bindingResult` care este un obiect din Spring și pe care se pun erorile de validare și `responseContainer` care este răspunsul ce se va trimite clientului.

Controller-ul corespunzător locurilor de parcare se poate observa în figura de mai jos:

```

21  @RestController
22  @RequestMapping(value = "/parking-spot")
23  public class ParkingSpotController extends AbstractController {
24
25      @Autowired
26      private ParkingSpotValidator parkingSpotValidator;
27
28      @Autowired
29      private ParkingSpotIdsValidator parkingSpotIdsValidator;
30
31      @Autowired
32      private ParkingSpotFacade parkingSpotFacade;
33
34      @ResponseBody
35      @RequestMapping(value = "/add", method = RequestMethod.POST, consumes = "application/json", produces = "application/json")
36      public ResponseContainer addParkingSpot(@RequestBody ParkingSpotData parkingSpotData, BindingResult bindingResult) {
37          ResponseContainer responseContainer = new ResponseContainer();
38
39          if (!validate(parkingSpotValidator, parkingSpotData, bindingResult, responseContainer)) {
40              return responseContainer;
41          }
42
43          parkingSpotData = parkingSpotFacade.addParkingSpot(parkingSpotData);
44          responseContainer.setData(parkingSpotData);
45
46          return responseContainer;
47      }

```

Se poate remarcă adnotarea `RequestMapping` care apare atât la nivel de clasă, unde specifică

prefixul rutelor din această clasă, cât și la nivel de metodă, unde specifică ruta pentru care poate fi folosită metoda în cadrul unei cereri, metoda HTTP, care este tipul informației pe care metoda o poate consuma și care este tipul de informație cu care metoda o să răspundă cererii.

Pentru metoda noastră **addParkingSpot**, ea este folosită pentru cererile la adresa `/parking-spot/add`, pentru metoda HTTP POST, datele transmise de către client trebuie să fie în format JSON și va răspunde cu date în format JSON.

De asemenea, datele primite trebuie să poată fi deserializate automat de către Spring într-un obiect `ParkingSpotData`, lucru ce semnifică o restricție Spring prin adnotarea `RequestBody`, pentru că dacă acesta nu va reuși deserializarea, va trimite ca răspuns un cod status de eroare 400.

În figura de mai jos este tipul de mesaj pe care clientul va trebui să îl trimită în cerere pentru a putea fi interpretat corect de către server.

```
1  {
2      "latitude": 46.760744828311886,
3      "longitude": 23.579296935349703,
4      "address": "Strada Ciprian Porumbescu 22, Cluj-Napoca 400000, Romania",
5      "activeDaysIntervals": [
6          {
7              "dayOfWeek": "MONDAY",
8              "startTime": "21:30"
9          },
10         {
11             "dayOfWeek": "TUESDAY",
12             "startTime": "21:30"
13         },
14         {
15             "dayOfWeek": "WEDNESDAY",
16             "startTime": "21:30"
17         },
18         {
19             "dayOfWeek": "THURSDAY",
20             "startTime": "21:30"
21         },
22         {
23             "dayOfWeek": "FRIDAY",
24             "startTime": "21:30"
25         }
26     ],
27     "id": null,
28     "userId": null
29 }
```

Acest obiect JSON va fi deserializat automat de către Spring printr-un `ObjectMapper` într-un obiect Java (POJO) ca din figura de mai jos. Despre data object vom vorbi mai multe atunci când vom vorbi despre stratul Facade și componentele sale.

```
6  public class ParkingSpotData implements Serializable {
7      private Long id;
8      private Double latitude;
9      private Double longitude;
10     private String address;
11     private Long userId;
12     private List<ParkingSpotActiveIntervalData> activeDaysIntervals;
```

Validatorul acestui obiect arată ca în figura de mai jos și specifică constrângerile legate de atributele obiectului. Spre exemplu latitudinea și longitudinea poziției unui loc de parcare trebuie să fie valori între anumite limite, latitudinea în intervalul [-90, 90] iar longitudinea în intervalul [-180, 180]. În figura de mai jos este validatorul corespunzător datelor ce vin de la client pentru un loc de parcare.

```

16  @Component
17  public class ParkingSpotValidator implements Validator {
18      @Autowired
19      private UserService userService;
20
21      @Override
22      public boolean supports(Class<?> clazz) { return ParkingSpotData.class.equals(clazz); }
23
24      @Override
25      public void validate(Object target, Errors errors) {
26          ParkingSpotData parkingSpotData = (ParkingSpotData) target;
27          ValidationUtils.rejectIfEmpty(errors, field: "latitude", errorCode: "error.parkingSpot.latitude.empty");
28          ValidationUtils.rejectIfEmpty(errors, field: "longitude", errorCode: "error.parkingSpot.longitude.empty");
29
30          if (parkingSpotData.getLatitude() < -90 || parkingSpotData.getLongitude() > 90) {
31              errors.rejectValue( field: "latitude", errorCode: "error.parkingSpot.latitude.outOfBounds");
32          }
33
34          if (parkingSpotData.getLongitude() < -180 || parkingSpotData.getLongitude() > 180) {
35              errors.rejectValue( field: "longitude", errorCode: "error.parkingSpot.longitude.outOfBounds");
36          }
37
38          List<ParkingSpotActiveIntervalData> activeDaysIntervals = parkingSpotData.getActiveDaysIntervals();
39          if (CollectionUtils.isEmpty(activeDaysIntervals)) {
40              errors.rejectValue( field: "activeDaysIntervals", errorCode: "error.parkingSpot.activeDaysIntervals.empty");
41          }
42          else {
43              for (ParkingSpotActiveIntervalData activeDaysInterval : activeDaysIntervals) {
44                  if (activeDaysInterval.getStartTime() == null ||
45                      activeDaysInterval.getEndTime() == null ||
46                      activeDaysInterval.getStartTime().isAfter(activeDaysInterval.getEndTime())) {
47                      String[] args = {activeDaysInterval.getDayOfWeek().getDisplayName(TextStyle.FULL, userService.getCurrentLocale())};
48                      errors.rejectValue( field: "activeDaysIntervals", errorCode: "error.parkingSpot.activeDaysIntervals.interval.incorrect", args,
49                  }
50              }
51          }
52      }
53  }

```

2. Facade

Layer-ul de Facade este un layer intermediar între Controller și Service. Acesta are rolul de a transforma modelele aplicației, cele care țin toate informațiile despre relațiile și atributele din baza de date, în obiecte mai simple, cu mai puține date, doar cele necesare numite data objects. Acest lucru se face utilizând unele clase numite Populator. De asemenea, obiectele care vin de la client sunt obiecte ce nu au informații despre relații sau id-uri, deoarece acestea nu sunt create încă, așa că și aceste date trebuie transformate în modele prin aceleași clase Populator.

În figurile de mai jos sunt clasele ParkingSpotFacadeImpl și ParkingSpotPopulator prin care datele ce au venit de la ParkingSpotController sunt transformate într-un model pentru a fi salvat în baza de date.

```

22  @Component
23  public class ParkingSpotFacadeImpl implements ParkingSpotFacade {
24
25      @Autowired
26      private Populator<ParkingSpotData, ParkingSpot> parkingSpotPopulator;
27
28      @Autowired
29      private Populator<ParkingSpot, ParkingSpotData> parkingSpotDataPopulator;
30
31      @Autowired
32      private ParkingSpotService parkingSpotService;
33
34      @Autowired
35      private UserService userService;
36
37      @Override
38      public ParkingSpotData addParkingSpot(ParkingSpotData parkingSpotData) {
39          try {
40              User currentUser = userService.getCurrentUser();
41              ParkingSpot parkingSpotFromData = new ParkingSpot();
42              parkingSpotPopulator.populate(parkingSpotData, parkingSpotFromData);
43              Long id = parkingSpotService.save(parkingSpotFromData, currentUser);
44              parkingSpotData.setId(id);
45          } catch (UserException e) {
46              parkingSpotData = null;
47          }
48
49          return parkingSpotData;
50      }

```

```

10  @Component
11  public class ParkingSpotPopulator implements Populator<ParkingSpotData, ParkingSpot> {
12      @Autowired
13      private ParkingSpotService parkingSpotService;
14
15      @Override
16      public void populate(ParkingSpotData parkingSpotData, ParkingSpot parkingSpot) {
17          parkingSpot.setLatitude(parkingSpotData.getLatitude());
18          parkingSpot.setLongitude(parkingSpotData.getLongitude());
19          parkingSpot.setAddress(parkingSpotData.getAddress());
20          parkingSpot.setActiveDaysIntervals(parkingSpotService.formatActiveDaysIntervals(parkingSpotData.getActiveDaysIntervals()));
21      }
22 }

```

3. Service

Logica aplicației se face în layer-ul Service, unde toate clasele lucrează cu modelele aplicației pe care le modifică, le integrează și le creează. De asemenea, aceste clase sunt transacționale, adică pot să deschidă o conexiune cu baza de date prin care să persiste sau să caute informații. În figura de mai jos este ParkingSpotServiceImpl, care preia datele venite de la ParkingSpotFacadeImpl și le

prelucrează, setând relații pe locul de parcare cu utilizatorul care l-a creat.

```
31  @Service
32  @Transactional
33  public class ParkingSpotServiceImpl implements ParkingSpotService {
34
35      @Autowired
36      private ParkingSpotDao parkingSpotDao;
37
38      @Autowired
39      private UserService userService;
40
41      @Autowired
42      private ReservationService reservationService;
43
44      @Override
45      public Long save(ParkingSpot parkingSpot, User user) {
46          parkingSpot.setUser(user);
47          userService.addParkingSpotToCurrentUser(parkingSpot);
48          return parkingSpotDao.save(parkingSpot);
49      }
50
51      @Override
52      public void update(ParkingSpot parkingSpot) {
53          parkingSpotDao.update(parkingSpot);
54      }
```

4. Repository. Data Access Object (DAO)

Clasele din layer-ul DAO sunt clasele care se ocupă cu comunicarea dintre aplicație și baza de date. Aceste clase se ajută de sesiunile Spring Session pentru a deschide conexiuni cu baza de date și a interoga tabelele corespunzătoare entităților din aplicație. Acest lucru se face cu ajutorul Hibernate ORM, care reușește să creeze conexiuni între o tabelă dintr-o bază de date și modelele din aplicația Java.

În figurile de mai jos sunt reprezentate entitatea Parking Spot, care reprezintă un loc de parcare și clasa DAO ParkingSpotDaoImpl, care reprezintă clasa ce dispune de metode pentru persistarea obiectelor.

```

6     @Entity
7     @Table(name = "parkingSpot")
8     public class ParkingSpot {
9         @Id
10        @GeneratedValue
11        private Long id;
12
13        @Column(nullable = false)
14        private Double latitude;
15
16        @Column(nullable = false)
17        private Double longitude;
18
19        @Column(nullable = false)
20        private String address;
21
22        @Column(nullable = false)
23        private String activeDaysIntervals;
24
25        @ManyToOne
26        private User user;
27
28        @OneToMany
29        private List<Reservation> reservations;

```

```

13     @Repository
14     public class ParkingSpotDaoImpl implements ParkingSpotDao {
15
16         @Autowired
17         private SessionFactory sessionFactory;
18
19         @Override
20         public Long save(ParkingSpot parkingSpot) {
21             Session session = sessionFactory.getCurrentSession();
22             return (Long) session.save(parkingSpot);
23         }

```

După ce se folosește metoda **save** din clasa de mai sus, se returnează id-ul entității salvate care se setează mai apoi pe entitatea de răspuns care se va trimite către aplicația client sub forma unui răspuns JSON ca în exemplul de mai jos.

Răspunsul pe care serviciul Web îl trimită pentru fiecare cerere este unul standard care conține un flag ce determină dacă au fost sau nu probleme cu datele sau cu acțiunea numit successful, dacă acesta este false, atunci există un câmp numit errors care conține erorile, iar dacă acțiunea a fost efectuată cu succes, atunci există obiectul pe care aplicația îl așteaptă.

```

5  public class ResponseContainer implements Serializable {
6      private boolean successful;
7      private Object data;
8      private Object errors;
9
10     public ResponseContainer() { successful = true; }
11
12     public boolean isSuccessfull() { return successful; }
13
14     public void setSuccessful(boolean successful) { this.successful = successful; }
15
16     public Object getData() { return data; }
17
18     public void setData(Object data) { this.data = data; }
19
20     public Object getErrors() { return errors; }
21
22     public void setErrors(Object errors) { this.errors = errors; }
23
24 }
```

În figura de mai jos este răspunsul cererii de la începutul subcapitolului, care a fost procesată cu succes.

```

1  {
2      "errors": null,
3      "successful": true,
4      "parkingSpot": {
5          "latitude": 46.760744828311886,
6          "longitude": 23.579296935349703,
7          "address": "Strada Ciprian Porumbescu 22, Cluj-Napoca 400000, Romania",
8          "activeDaysIntervals": [
9              {
10                  "dayOfWeek": "MONDAY",
11                  "startTime": "21:30"
12              },
13              {
14                  "dayOfWeek": "TUESDAY",
15                  "startTime": "21:30"
16              },
17              {
18                  "dayOfWeek": "WEDNESDAY",
19                  "startTime": "21:30"
20              },
21              {
22                  "dayOfWeek": "THURSDAY",
23                  "startTime": "21:30"
24              },
25              {
26                  "dayOfWeek": "FRIDAY",
27                  "startTime": "21:30"
28              }
29          ],
30          "id": 182812,
31          "userId": 32312
32      }
33  }
```

3.2 Funcționalități

Atunci când utilizatorul deschide aplicația Parker este întâmpinat de un design minimalist care atrage privirea prin culori doar acțiunilor pe care le poate face utilizatorul în cadrul aplicației.

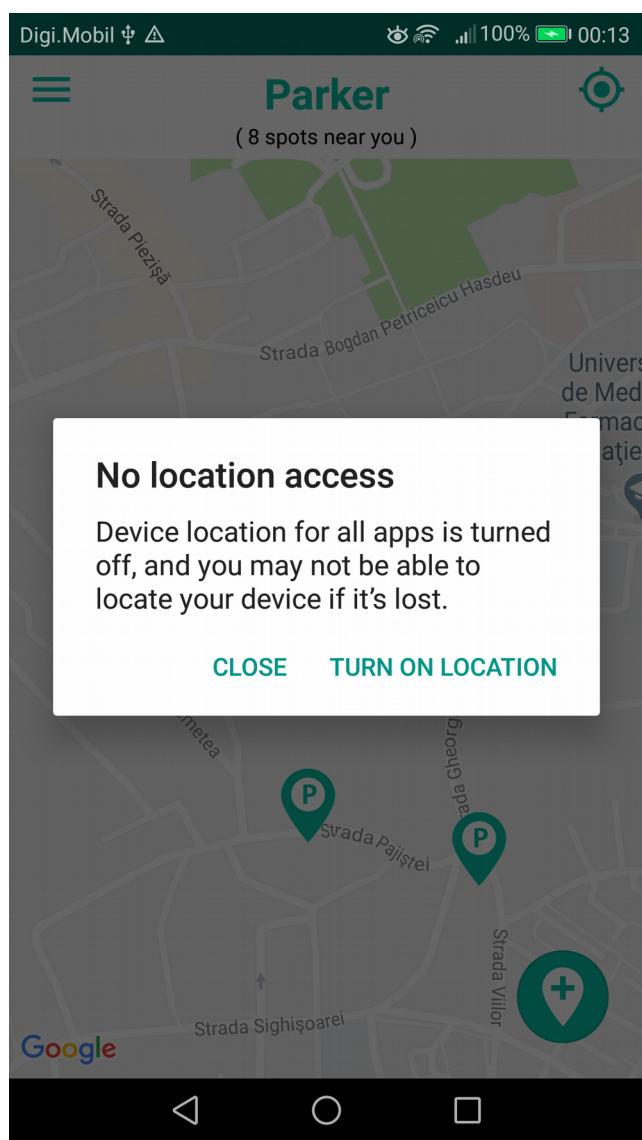
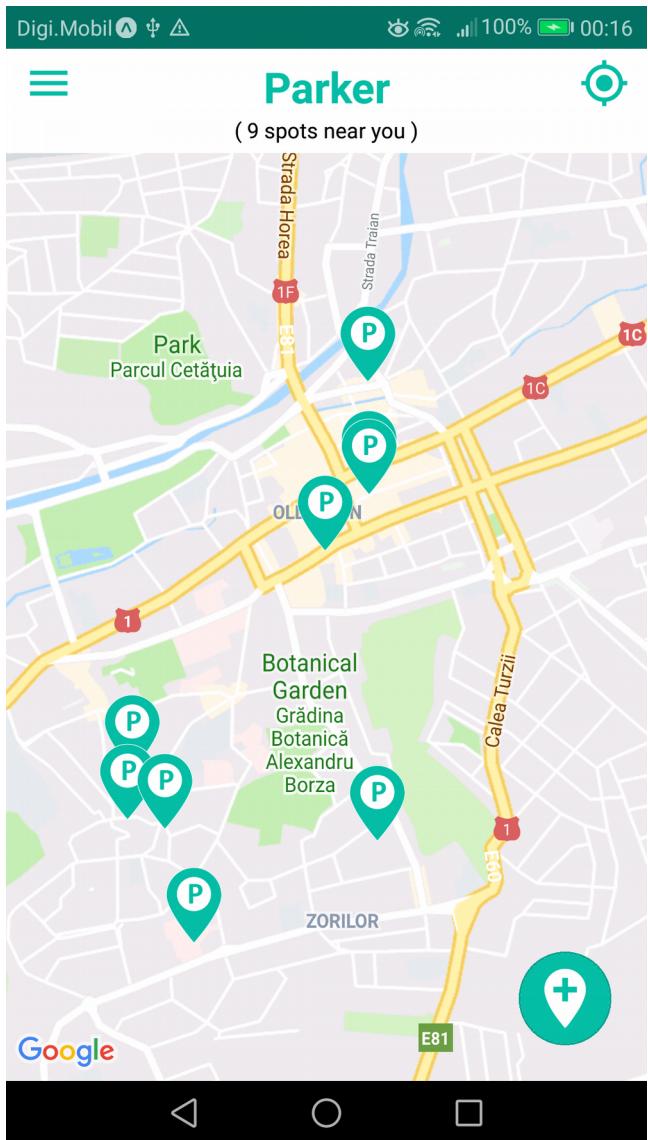
1. Harta

Pentru a afișa locația locurilor de parcare am integrat în aplicație Google Maps, mai exact biblioteca pentru React Native a acestei tehnologii. Harta arată locurile disponibile din jurul locației la care a navigat utilizatorul, marcându-le cu indicatoare verzi personalizate.

2. Locația

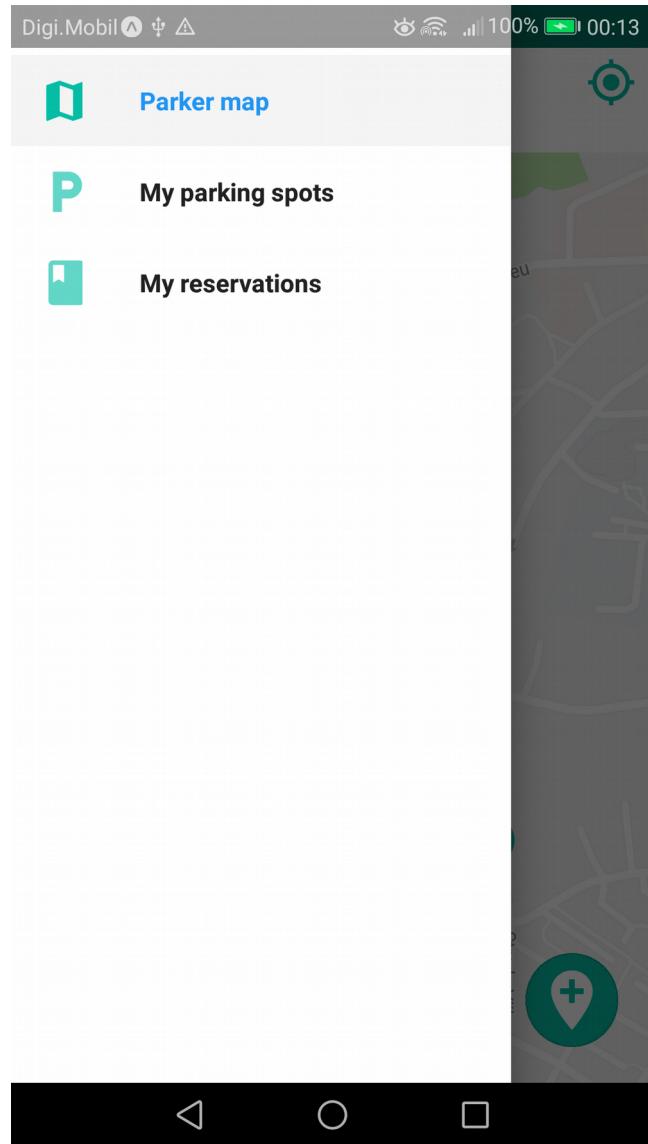
Deoarece locurile de parcare pe care Parker le afișează sunt actualizate în timp real, este necesară eficientizarea numărului de cereri care se fac către server pentru. De asemenea, pentru a nu consuma foarte multe date mobile utilizatorului, Parker cere acces la locație, afișând astfel locurile de parcare într-o rază de 1km de locația utilizatorului. De asemenea, utilizatorul poate naviga pe hartă de la poziția curentă către poziția unde vrea să ajungă, unde i se vor afișa locurile de parcare disponibile din aplicație. De asemenea, în partea de sus a ecranului, sub logo-ul aplicației, există un text în care este afișat în timp real numărul de locuri de parcare disponibile din jurul utilizatorului.

Dacă utilizatorul nu are locația pornită pe telefon, atunci un mesaj de eroare va apărea, explicând acest lucru.



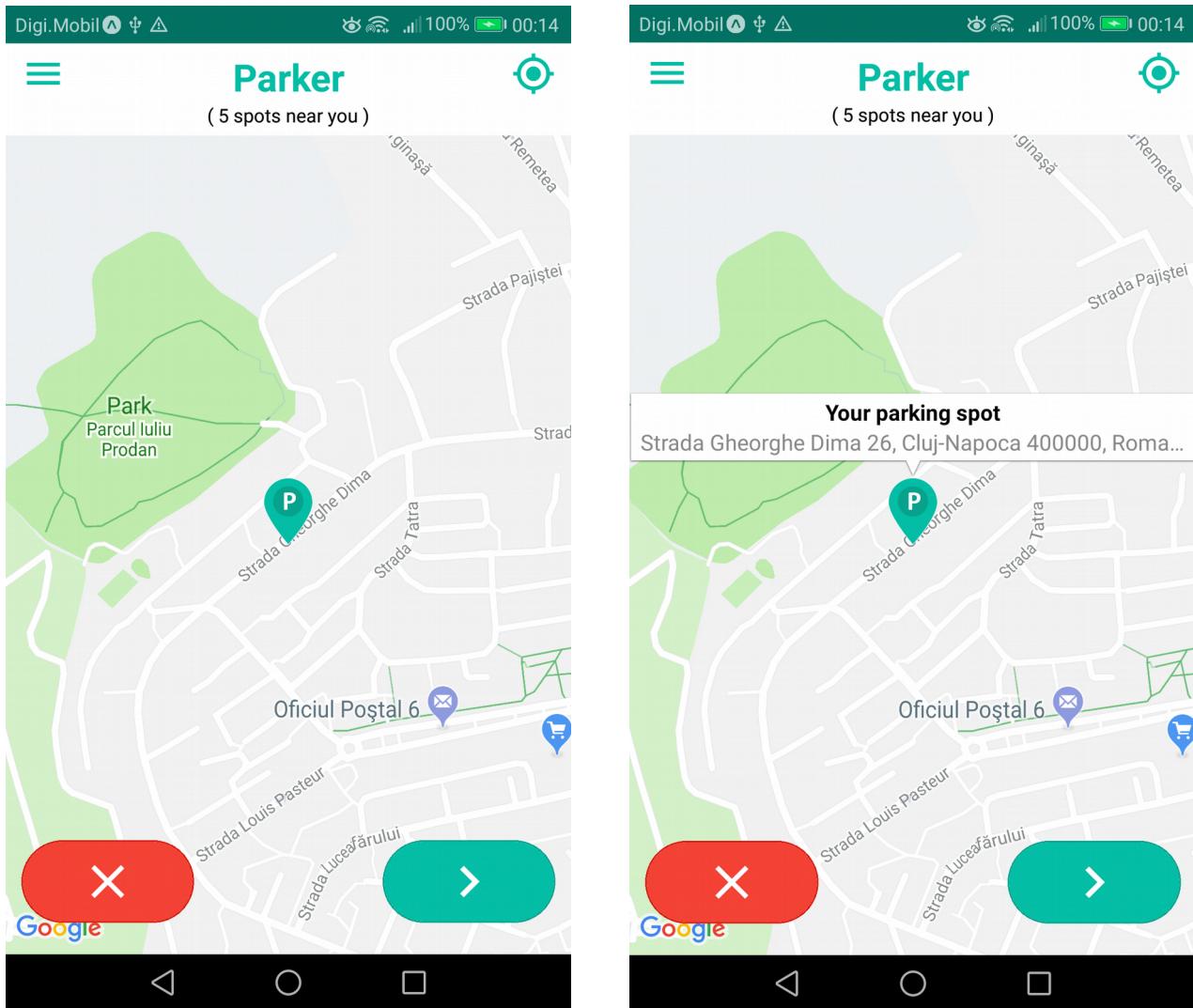
3. Meniul principal

Aplicația dispune de un meniu de tip „Drawer” pentru a putea naviga între ecranul principal în care există majoritatea funcționalităților ce țin de locurile de parcare în timp real și ecranele de administrare în care utilizatorul poate să își vadă locurile sale de parcare adăugate în aplicație sau rezervările făcute pentru alte locuri de parcare.

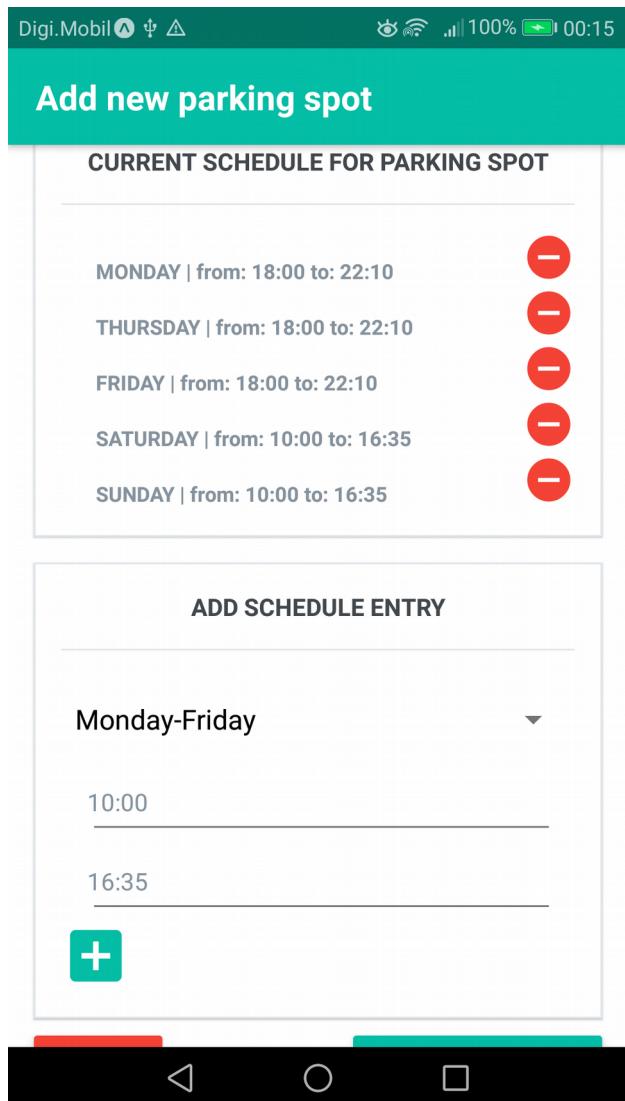
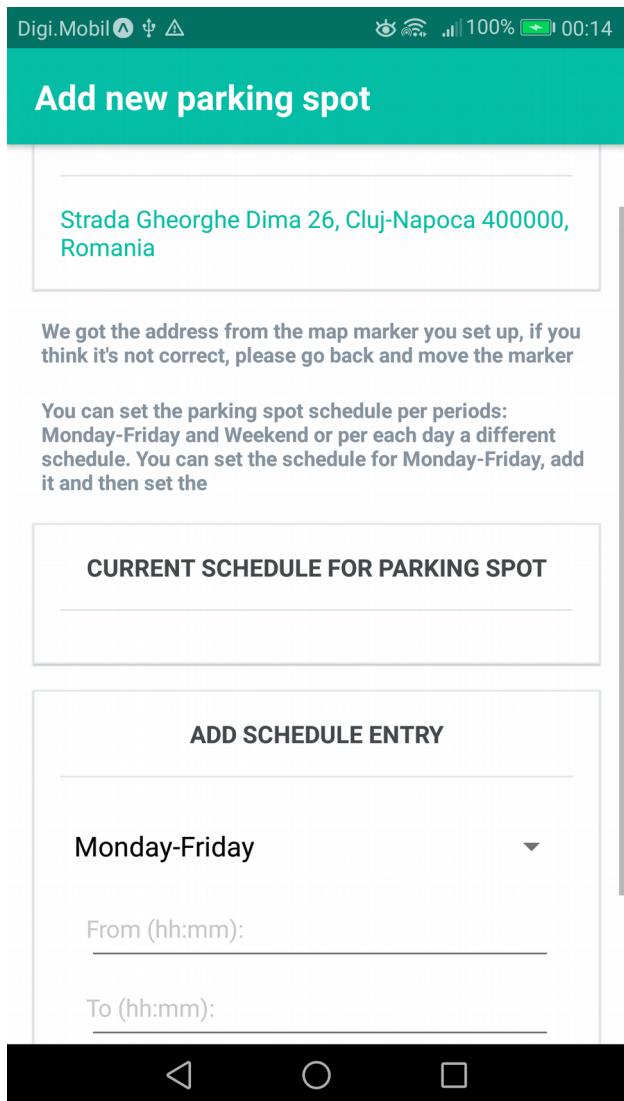


4. Adăugarea unui loc de parcare

Dacă utilizatorul dispune de un loc de parcare pe care poate să îl ofere spre închiriere atunci când acesta nu îl folosește, acesta are posibilitatea să îl adauge în aplicație spre folosirea celorlați șoferi, printr-o serie de pași intuitivi. Acesta apasă butonul de adăugare din colțul dreapta jos al ecranului principal, moment în care aplicație intră în modul adăugare, afișând pe ecran un marker personalizat care este centrat pe mijlocul ecranului și reprezintă locația locului de parcare ce va fi adăugat. Aceasta se poate modifica prin navigarea pe hartă și selectând locația corectă pe care utilizatorul o dorește. După ce acesta este mulțumit cu locația va apăsa pe butonul de continuare a procesului de adăugare al locului de parcare.



Următorul pas în procesul de adăugare a unui loc de parcare o face adăugarea programului în care locul de parcare este disponibil. Aplicația face în aşa fel încât utilizatorului să ii fie foarte ușor să adauge un program împărțit pe zile și intervale orare pentru locul de parcare. Odată cu încheierea acestui pas, locul de parcare poate fi adăugat sau se poate renunța la adăugarea acestuia.



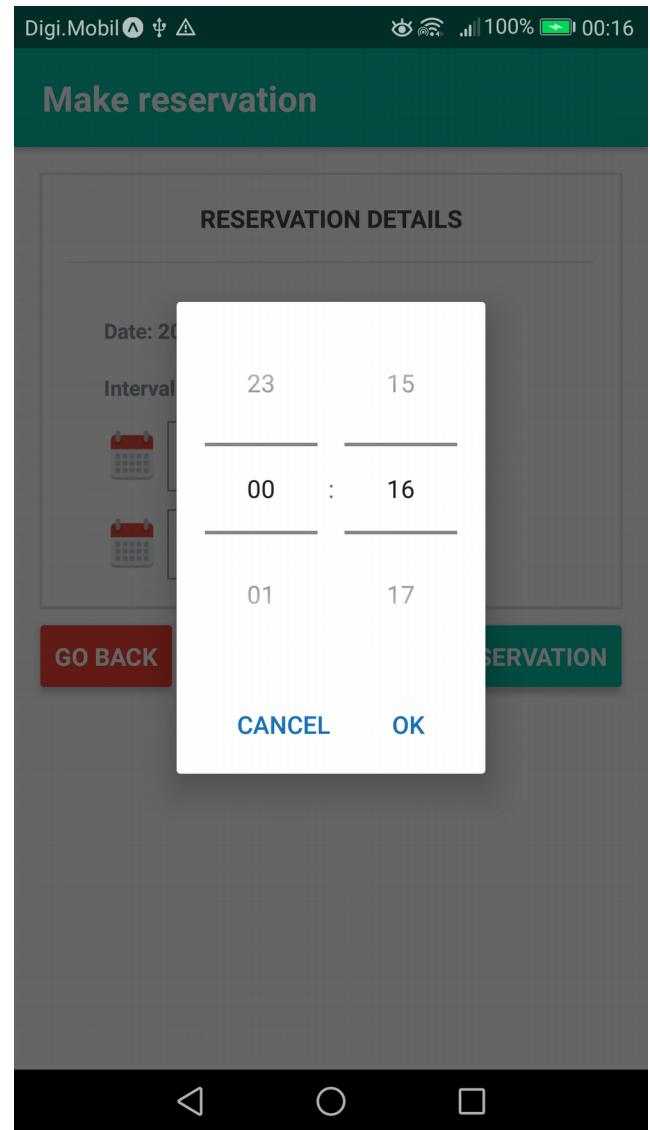
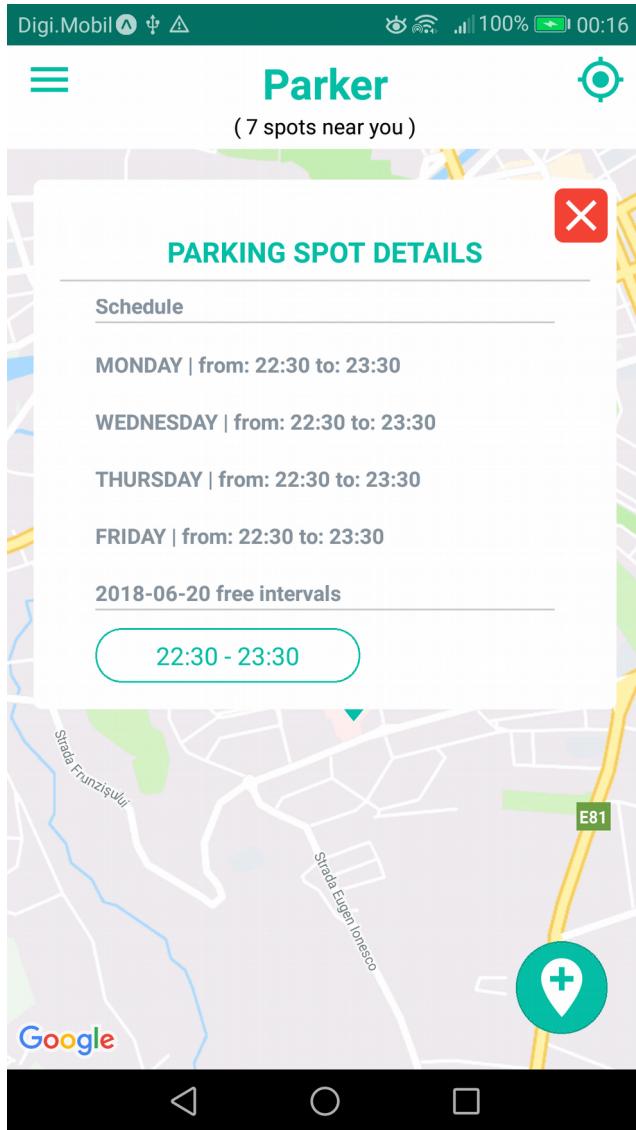
După adăugarea cu succes a locului de parcare în aplicație utilizatorul va fi readus automat la ecranul principal unde va primi un mesaj cu acțiunea efectuată cu succes și atât el cât și ceilalți utilizatori vor putea vedea permanent locul de parcare pe locația pe care acesta a fost adăugat, împreună cu intervalele orare în care acesta este disponibil. De asemenea, după adăugare, utilizatorii vor putea rezerva acest loc de parcare pentru folosire.

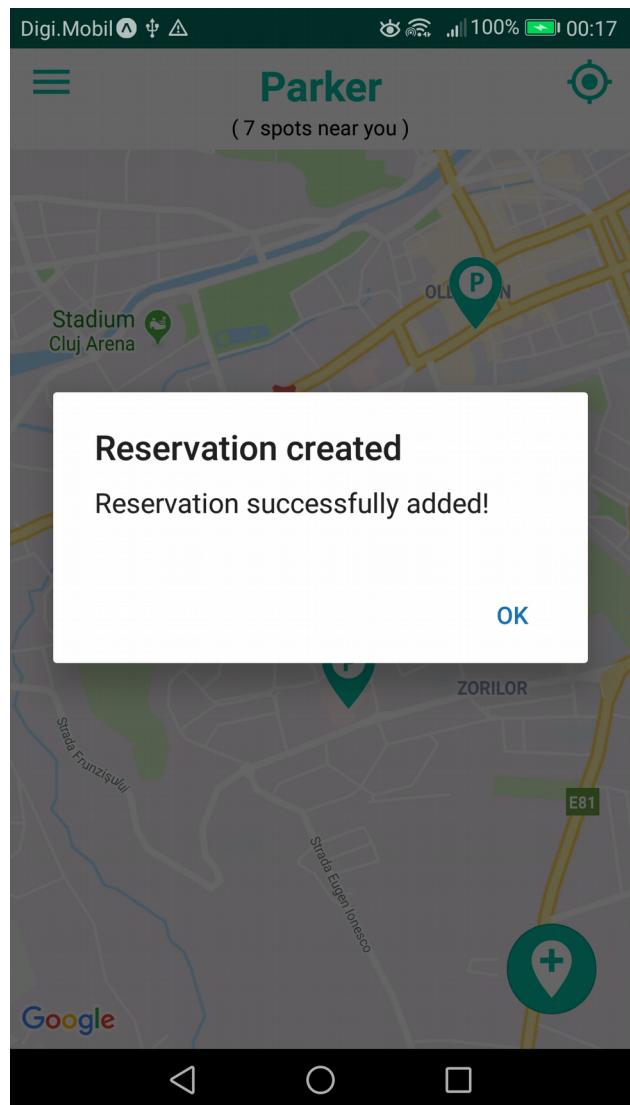
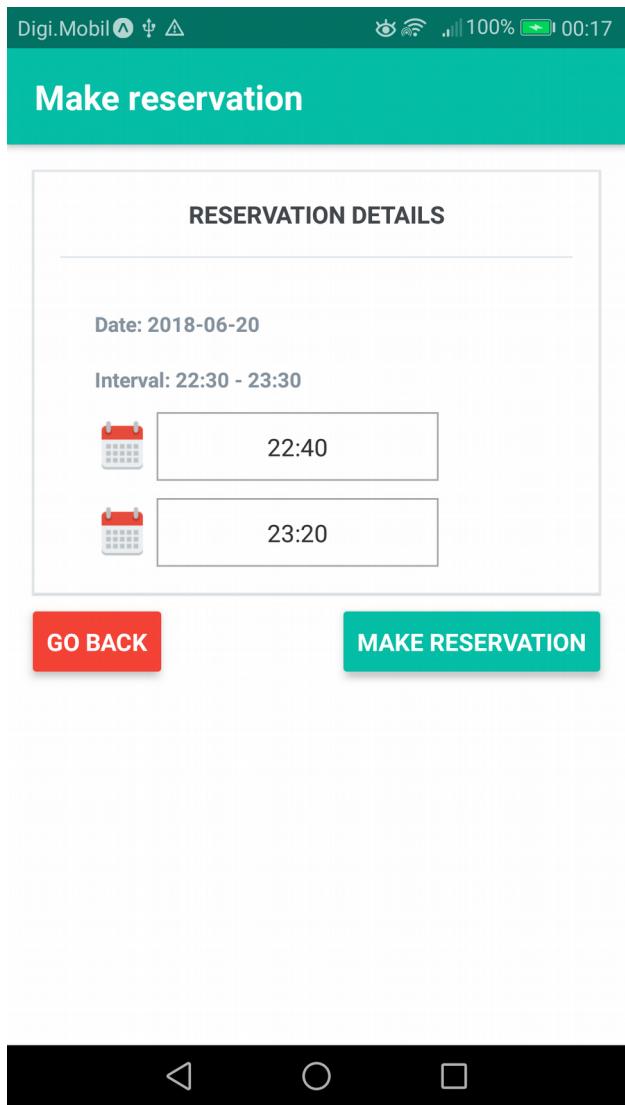
4. Rezervarea unui loc de parcare

Pentru toate locurile afișate pe ecranul principal se pot vedea detalii prin apăsarea zonei pe care acestea apar pe ecran. Atunci, aplicația afișează o fereastră de tip overlay, peste hartă cu adresa exactă a locului de parcare și intervalul orar, pentru ziua curentă, în care acea este disponibilă pentru rezervare.

Toate aceste lucruri sunt în timp real, astfel că dacă cineva a făcut deja o rezervare pentru acest loc de parcare intervalul va fi micșorat sau împărțit în intervale care să fie înainte și după rezervarea existentă, dar să fie de asemenea în intervalul permis al locului de parcare.

Pentru a rezerva un loc de parcare, utilizatorul trebuie doar să apese pe intervalul dorit din lista de intervale. Aplicația va deschide un alt ecran unde utilizatorul va putea introduce intervalul în care are nevoie de acel loc de parcare și va putea anula sau comite rezervarea. Dacă acesta face rezervarea, el este redus pe ecranul principal unde i se afișează mesajul corespunzător acțiuni făcute.





5. Rezervările utilizatorului

Utilizatorul poate să își vadă rezervările active navigând la pagina „My reservations” din cadrul meniului principal.

6. Anularea unei rezervări

Utilizatorul poate anula, în cazul în care nu mai are nevoie sau a intervenit ceva neprevăzut, o rezervare făcută în prealabil pe un loc de parcare.

7. Locurile de parcare ale utilizatorului

Utilizatorul poate să își vadă locurile de parcare adăugate în aplicație navigând la pagina „My

“parking spots” din cadrul meniului principal.



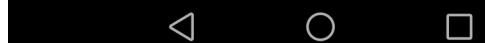
≡ My parking spots

1. Cluj
2. Strada Painii, Cluj
3. Strada Painii, Cluj
4. Strada Painii, Cluj
5. Strada Ciprian Porumbescu 22, Cluj-Napoca
400000, Romania
6. Strada Viilor 6, Cluj-Napoca 400000, Romania
7. Strada Napoca 3, Cluj-Napoca 400000, Romania
8. Strada Paji?tei 19, Cluj-Napoca 400000, Romania
9. Strada Republicii 98, Cluj-Napoca 400000, Romania
10. Strada Viilor 46-50, Cluj-Napoca 400000, Romania
11. Strada Gheorghe Dima 26, Cluj-Napoca 400000,
Romania



≡ My reservations

1. 2018-06-05 | 13:14 - 15:14 **CANCEL X**
2. 2018-06-05 | 21:40 - 22:00 **CANCEL X**
3. 2018-06-20 | 22:40 - 23:20 **CANCEL X**



Concluzii

Pentru a putea gestiona locurile de parcare dintr-un oraș modern, este nevoie de soluții inteligente pentru a reduce timpul irosit, combustibilul consumat și banii pierduți de către șoferi în căutarea unui loc de parcare. Una dintre aceste soluții este cea propusă în această lucrare, care folosește serviciile Web pentru a oferi o aplicație mobilă independentă de platformă, utilizatorilor care pot astfel să aibă acces la date despre locurile disponibile în timp real.

Atât serviciile Web cât și React Native sunt tehnologii și concepte de actualitate care ajută dezvoltatorii să atingă performanțe și scopuri care altfel nu ar fi putut fi atinse. Dacă serviciile Web împreună cu REST sunt concepte formalizate și testate în industrie de-a lungul timpului și dispun de documentații stufoase și bine pregătite, React Native vine cu un dinamism crescut din cauza timpului limitat de care a avut parte de a se dezvolta. Cu toate acestea, comunitatea programatorilor din întreaga lume care folosește React Native este dispușă să ajute și să ofere informații, astfel că, după cum se poate demonstra, tehnologia este una care funcționează și care promite multe lucruri pentru viitor. De asemenea, e important să evidențiem faptul că aplicația atașată acestei lucrări poate să beneficieze în continuare de funcționalități și idei noi, iar dezvoltarea va continua și va încerca să țină pasul cu concepte ca mașini autonome sau cu integrări cu alte aplicații sau în cadre altor aplicații. Aceste aspecte sunt ușor de implementat din cauza faptului că serviciile Web pot fi integrate cu ușurință în orice aplicație care poate să facă cereri HTTP și să interpreteze răspunsurile.

Bibliografie

[1] - Studiu: “Searching for Parking Costs Americans \$73 Billion a Year”.

<http://inrix.com/press-releases/parking-pain-us/>

[2] - L. Alboaie, S. Buraga: “Servicii Web. Concepțe de bază și implementări” (2006)

[3] - “What is the difference between the Web and the Internet?”.

<https://www.w3.org/Help/#webinternet>

[4] - Matthew Gray: “Web Growth Summary”. MIT.edu.

<https://stuff.mit.edu/people/mkgray/net/web-growth-summary.htm>

[5] - Statistică: “Number of smartphone users worldwide from 2014 to 2020 (in billions)”.

<https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/>

[6] - Roy Thomas Fielding: “Architectural Styles and the Design of Network-based Software Architectures” (2000).

<https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm>

[7] - Craig Walls: “Spring in Action” (2005)

[8] - Christian Bauer, Gavin King: “Java Persistence with Hibernate” (2006)

[9] - Documentație biblioteca Redux, “Motivation”.

<https://redux.js.org/introduction/motivation>

[10] - Sam Ruby, Leonard Richardson: “RESTful Web Services” (2007)

[11] - “Applying MVC” <http://developer.ucsd.edu/develop/user-interface/applying-mvc.html>

[12] - Statistică: “Percentage of all global web pages served to mobile phones from 2009 to 2018”.

<https://www.statista.com/statistics/241462/global-mobile-phone-website-traffic-share/>

[13] - “Hibernate – Arhitecture”.

https://www.tutorialspoint.com/hibernate/hibernate_architecture.htm

[14] - Despre React Native.

<https://facebook.github.io/react-native/>

[15] - Documentație Spring Framework.

<https://docs.spring.io/spring/docs/5.0.7.RELEASE/spring-framework-reference/core.html#spring-core>