*Universitatea din București*

*Facultatea de Matematică și Informatică*

*Lucrare de Disertație*

Comunicarea Arhitecturilor

Mobile și Desktop

*Conducător științific: Absolvent:*

Prof. dr. Horia Georgescu Sorin Slavic

*Iunie 2012*

[1. **Introducere** **2**](#_Toc326717352)

[2. **Sistemul** **Android** **4**](#_Toc326717353)

[2.1 Introducere 4](#_Toc326717354)

[2.2 Securitatea aplicațiilor Android 5](#_Toc326717355)

[2.3 Componentele unei aplicații Android 6](#_Toc326717356)

[2.4 Activități 11](#_Toc326717357)

[2.5 Interfața grafică 23](#_Toc326717358)

[3. **Comunicarea** **Client**-**Server** **35**](#_Toc326717359)

[3.1 Introducere 35](#_Toc326717360)

[3.2 Java Sockets 36](#_Toc326717361)

[3.3 Http Client 39](#_Toc326717362)

[3.4 Java Serialization 42](#_Toc326717363)

[3.5 SOAP 44](#_Toc326717364)

[4. **Arhitectura** **Spring** - **Web** **55**](#_Toc326717365)

[5. **Aplicație** – **Mobile** **CMD** **57**](#_Toc326717366)

[5.1 Baza de date MySQL 58](#_Toc326717367)

[5.2 Aplicația de Server Web 61](#_Toc326717368)

[5.3 Aplicația de Client Desktop 77](#_Toc326717369)

[5.4 Aplicația de Client Android 88](#_Toc326717370)

[5.5 Concluzii 91](#_Toc326717371)

[6. **Aplicații** **similare** **92**](#_Toc326717372)

1. Introducere

În ultimii ani s-a creat o nouă comunitate de programatori și un set complet nou și inovativ de aplicații si framework-uri de dezvoltare. Acestea se datorează dezvoltării aplicațiilor pentru telefoane mobile și smartphone-uri. Toate aceste noi aplicații trebuie să îndeplinească funcții de bază, simple dar să și prezinte o interfață prietenoasă utilizatorilor.

Spre deosebire de aplicațiile dezvoltate pe sistem Enterprise (mari, complexe) sau cele Web, aplicațiile dezvoltate pentru dizpozitive mobile trebuie să respecte alte principii și să se bazeze pe arhitecturi care au o logică un pic diferită. Trebuie avut în vedere în primul rând că telefoanele mobile au resurse puține, atât ca viteză de lucru, spațiu de depozitare sau memorie internă. Telefoanele au și o baterie limitată care poate să limitează capacitățile unei aplicații.

Instalarea unei astfel de aplicații trebuie să fie cât mai ușoară și simplă pentru utilizator care, în cazul a multor dintre sistemele de operare pentru telefoane mobile, nu are acces la anumite funcționalități, la locul de instalare, la configurații sau alte opțiuni care în cazul altor aplicații sunt foarte utile și recomandate să fie folosite.

Aplicațiile pentru mobil se pot adresa unui număr foarte mare de persoane, din medii sociale sau profesionale diferite și trebuie să fie cât mai simple și sumare în opțiuni și configurații. Cele mai multe dintre ele îndeplinesc o singură funcționalitate și aceasta cât mai logic și mai intuitiv pentru utilizator.

Există foarte multe platforme pentru dispozitive mobile sau tablete, incluzând Symbian, iPhone, Windows Mobile, BlackBerry, Java Mobile Edition, Linux Mobile (LiMo). Pentru această lucrare am utilizat sistemul Android. Această platformă are un prim avantaj față de unele dintre celelalte sisteme prin faptul că e Open Source. Companiile producătoare de dispozitive mobile îl folosesc întrucât nu trebuie să plătească o taxă de utilizare și poate fi customizat ușor în funcție de preferințe, având funcționalitățile de bază deja configurate. Comunitatea de dezvoltatori putem spune că apreciază acest sistem întrucât nu e direct legat de o companie de la care să se cumpere licențe, update-uri sau release-uri noi, orice modificare e accesibilă oricui.

Arhitectura sistemului Android este bazată pe componente interne ce pot fi ușor înlocuite sau modificate. Există multe servicii incluse în pachetul de bază, cu care orice aplicație poate interacționa – de exemplu Locația (fie bazată pe GPS, pe triangulări în funcție de semnal “Cell Tower Triangulation”). Sistemul are inclusă și o bază de date SQL foarte utilă pentru gestiunea și accesarea datelor local. Browser-url WEB si o aplicație pentru Hărți (Map View) pot fi inserate și încorporate direct în orice aplicație.

Orice aplicație are un ciclu de viață (life cycle). Sub platforma Android programele sunt izolate una față de cealaltă prin sisteme și niveluri de securitate. Acestea permit o stabilitate foarte bine pusă la punct, față de alte sisteme de operare mobile. Dezvoltatorii sau utilizatorii nu trebuie să fie atenți la care și câte aplicații sunt pornite simultan sau la memoria utilizată. Aplicațiile sunt dezvoltate pentru un consum optim de resurse de memorie sau de energie a bateriei.

Având în vedere că sistemul Android este dezvoltat ca să ruleze pe o gamă largă de produse, diferite modele de telefoane sau tablete, fiecare având capabilități diferite, sunt introduse în librăriile interne ale sistemului de operare si elemente pentru grafică sau sunet performante, acceleratoare 3D, codec-uri pentru majoritatea standardelor audio și video, OpenGL

Această portabilitate este asigurată și de faptul că programele Android sunt scris în limbaj Java, rulând pe o mașină virtuală “Dalvik”. Codul este astfel posibil de utilizat și pe arhitecturi ARM, x86 sau altele. Există suport și pentru o varietate de modele de input – tastaturi, touchscreen, trackball. Orientarea interfeței este editabilă, putându-se dezvolta aplicații cu interfețe, rezoluție sau configurații diferite, în funcție de orientarea ecranului.

Aplicația dezvoltată pentru această lucrare are 3 componente ce lucrează și comunică între ele pentru a realiza o conexiune între un calculator personal, un client Desktop, și o aplicație pe mobil (Android), un client Mobil. Conexiunea este manageriată de un Server WEB care are în spate și o bază de date.

2. Sistemul Android

# 2.1 Introducere

Android este un sistem software Open Source pentru dispozitive mobile (telefoane mobile / smartphones / tablete) dezvoltat de Google si Open Handset Alliance. Această platformă este una dintre cele mai active și utizilate de pe piața telefoanelor mobile și a tabletelor. În Februarie 2012 a fost estimat că peste 800.000 de dispozitive erau activate zilnic. Sistemul Android include un sistem de operare și un set de servicii care permit mai mulor procese care rulează pe unul sau mai multe dispozitive să interacţioneze, precum şi alte aplicaţii. SDK-ul Android oferă instrumentele şi API-uri necesare pentru a începe dezvoltarea de aplicații pe platforma Android folosind limbajul de programare Java.

Android vine cu un set de aplicaţii de bază, inclusiv un client de e-mail, programul de SMS, calendar, hărţi, browser-ul, contactele, şi altele. Toate cererile sunt scrise folosind limbajul de programare Java.

Prin oferirea unei platforme de dezvoltare deschisă, Android oferă dezvoltatorilor posibilitatea de a construi aplicaţii extrem de bogate şi inovatoare. Dezvoltatorii sunt liberi să profite de dispozitivele hardware, au acces la localizare, pot derula servicii de background, seta alarme, adăuga notificări la bara de stare si multe altele.

Dezvoltatorii au acces deplin la API-urile aceluiaşi cadru utilizat de aplicaţiile de bază. Arhitectura aplicației este concepută pentru a simplifica refolosirea componentelor; orice aplicație poate publica capacităţile sale şi orice altă aplicaţie poate începe utilizarea acestor capacităţi (supuse unor constrângeri de securitate impuse de cadru).

La o aplicație desktop standard, fiecare fereastră este deschisă de utilizator. Există o logică urmată în pașii utilizatorului, el știind ce operații face sau cum. Nu este neapărat conștient de toate programele sau serviciile sistemului de operare pe care le utilizează aplicația sa, dar înțelege scopul lor. Interacțiunea cu alte programe se face deobicei prin API-uri sau alte interfețe care permit accesarea unor date sau a unor module. Sistemul Android are un concept similar, diferența fiind că legătura între componente nu este neapărat directă, asigurându-se o modularitate mai accentuată care separă aplicațiile în caz că apar probleme.

Fiecare fereastră deschisă sau folosită de un utilizator are la bază o acțiune de tip **activity**. Există cazuri în care aceste *activități* să nu aibă o interfață, acestea sunt considerate servicii sau metode de acces la date. **Content Providers** – furnizori de conținut – sunt metode de acces la datele de pe dispozitiv utilizate de mai multe aplicații. Acestea asigură un nivel de abstractizare a datelor, modelul de dezvoltare Android permite unei aplicații să acceseze datele altor aplicații și să își facă accesibile propriile date. Folosind un *content provider* ne asigurăm de felul în care alte aplicații ne accesează datele. **Intents –** intenție – sunt mesaje sistem care se transmit la aplicații în funcție de anumite evenimente, schimbări de hardware (card SD de memorie modificat), date recepționate (mesaje SMS), evenimentele unei aplicații (o anumită aplicație a fost pornită din meniu). O aplicație poate să trimită un răspuns la un astfel de mesaj sau poate crea un mesaj propriu de tip *Intent* care apoi să fie interpretat de alte aplicații sau care să te informeze de anumite evenimente. **Serviciile** sunt aplicații dezvoltate să ruleze independent de alte aplicații. Activitățile, furnizorii de conținut sau mesajele intenție au un timp de rulare scurt și pot fi oprite în orice moment. *Un serviciu* poate fi utilizate pentru a verifica o sursă RSS, pentru a rula un program în background când interfața a fost închisă (de exemplu: player de muzică sau un program de descărcare).

Aceste aplicații au acces și la fișierele salvate în interiorul telefonului. Pot fi împachetate datele pentru aplicație care nu suferă modificări (iconițe sau fișiere). Fiecare aplicație poate rezerva o anumită porțiune din spațiul de stocare a dispozitivului, pentru baze de date, fișiere editabile de către user sau pentru date externe (downloadate) ce sunt utilizate de aplicație. Se poate accesa și spațiul de stocare de pe un Card SD extern, cu drepturi de citire și scriere. Aplicațile Android suportă o conexiune la Internet, prin mai multe modalități. Accesul se poate face de la orice nivel al unei aplicații, utilizându-se Java Sockets sau până la Widget-uri pentru Browser (aplicații WebKit) ce pot fi încorporate direct de aplicație.

Dispozitivele Android au posibilitatea de a rula și randa fișiere de tip audio sau video. Unele dintre acestea pot varia în calitate sau specificații în funcție de componentele hardware și de dispozitiv. Multe dintre telefoane sau tablete pot avea incluse și camere de fotografiat, filmat, instrumente de înregistrat diferite valori externe (temperatură, luminozitate, nivel zgomot, presiune atmosferică, etc). Utilizarea sistemului GPS permite accesul la locație pentru orice aplicație, făcându-se posibiliă utilizarea hărților sau memorarea traseului urmat de un dispozitiv.

Acest sistem permite unei aplicații să aibe acces și la funcțiile de bază ale unui telefon mobil. Se pot iniția convorbiri telefonice, apeluri, trimitere sau primire de mesaje SMS, etc.

# 2.2 Securitatea aplicațiilor Android

Aplicaţiile Android sunt scrise în limbajul de programare Java. Instrumentele Android SDK compilează codul, împreună cu orice date sau fişiere din resurse, într-un fişier arhivă cu sufixul APK. Tot codul dintr-un singur fişier APK este considerat a fi o singură aplicație şi este fişierul pe care dispozitivele Android îl folosesc pentru a instala aplicaţia.

Odată instalată pe un dispozitiv, fiecare aplicație Android este supusă propriului sistem de securitate. Sistemul de operare Android este un sistem multi-user Linux în care fiecare aplicaţie este un alt utilizator. În mod implicit, sistemul atribuie fiecărei aplicații un ID unic de utilizator Linux (ID-ul este utilizat doar de către sistem şi este necunoscut aplicației). Sistemul setează permisiuni pentru toate fişierele într-o aplicaţie, astfel încât numai ID-ul de utilizator alocat acestei aplicații să le poată accesa.

Fiecare proces are propria maşină virtuală (VM), astfel încât codul unei aplicaţii rulează în izolare faţă de alte aplicaţii. Fiecare aplicaţie rulează în propriul proces Linux. Android începe procesul atunci când oricare dintre componentele aplicației trebuie să fie executată, apoi închide procesul atunci când este nu mai este utilizată sau în cazul în care sistemul trebuie să recupereze memorie pentru alte aplicaţii.

În acest fel, sistemul Android pune în aplicare principiul celui mai mic privilegiu. Acest lucru înseamnă că, fiecare aplicaţie, are acces numai la componentele de care are nevoie să îşi facă treaba şi nu mai mult. Acest lucru creează un mediu foarte sigur în care o aplicație nu poate accesa părţi ale sistemului pentru care nu este dată permisiunea.

Cu toate acestea, există modalități pentru ca o aplicație cu acces la date partajate de alte aplicaţii să aibă acces la serviciile acestora. Este posibil să se creeze pentru două aplicații un mod de a împărtăşi acelaşi ID de utilizator Linux, caz în care sunt capabile de a-și accesa fişierele reciproc. Pentru a conserva resursele sistemului, aplicațiile cu acelasi ID de utilizator pot aranja să ruleze, de asemenea, în acelaşi proces Linux şi să împartă aceeași masină virtuală.

O aplicaţie poate solicita permisiunea de a accesa datele dispozitivului, cum ar fi contactele utilizatorului, mesajele, cardul de depozitare (cardul SD), camera foto, Bluetooth, şi altele. Toate permisiunile aplicației trebuie să fie acordate de către utilizator la instalare.

Acestea sunt lucrurile de bază în ceea ce priveşte modul în care o aplicație Android există în cadrul sistemului. Componentele de bază care definesc aplicaţia sunt reprezentate prin un fişier manifest în care se declară componentele şi caracteristicile necesare dispozitivului mobil pentru a rula aplicația și resursele, care sunt separate de codul aplicației și se permite astfel optimizarea, graţie comportamentului adecvat pentru o varietate de configuraţii ale dispozitivului mobil.

# 2.3 Componentele unei aplicații Android

Componentele aplicaţiei sunt blocurile esenţiale de construcție ale unei aplicații Android. Fiecare componentă este un alt punct prin care sistemul poate intra în aplicație. Nu toate componentele sunt puncte reale de intrare pentru utilizator şi unele depind de altele, dar fiecare dintre ele există ca entitate proprie şi joacă un rol specific, fiecare bloc este o constructie unică, ce ajută la definirea comportamentul general al aplicaţiei.

Există patru tipuri diferite de componente ale aplicației. Fiecare tip serveşte unui scop distinct şi dispune de un ciclu de viaţă distinct care definește modul în care componenta este creată și distrusă.

**Activităţi:**

O activitate reprezintă un singur ecran cu o interfaţă. De exemplu, o aplicaţie de e-mail ar putea avea o activitate care afişează o listă de email-uri noi, o altă activitate pentru a compune un e-mail şi o altă activitate pentru citirea e-mailurilor. Deşi activităţile lucrează împreună pentru a forma o unitate corespunzatoare pentru utilizator, pentru aplicaţia e-mail fiecare dintre ele este independentă de celelalte. Ca atare, o aplicaţie diferită poate începe sau activa oricare dintre aceste activităţi (în cazul în care aplicația de e-mail permite acest lucru). De exemplu, o aplicație a camerei de filmat își poate începe activitatea în aplicația de e-mail în momentul în care se compune un e-mail nou, pentru ca utilizatorului să-i fie permis și să aibă resursele necesare pentru a împartăși sau a trimite o imagine.

O activitate este implementată ca o subclasă a clasei Activity.

**Servicii:**

Un serviciu este o componentă care se execută în background pentru a efectua operaţiuni de lungă durată sau care trebuie să presteze munca pentru procesele de la distanţă. Un serviciu nu oferă o interfaţă vizibilă utilizatorului. De exemplu, un serviciu ar putea reda muzică în fundal, în timp ce utilizatorul este într-o altă aplicaţie, sau ar putea aduce date prin reţea fără a bloca interacţiunea utilizatorului cu o altă activitate. O altă componentă, cum ar fi o activitate, poate porni un serviciu şi-l poate lăsa să funcţioneze în fundal sau se poate leaga de aceasta pentru a interacţiona cu el.

Un serviciu este implementat ca o subclasă a clasei Service.

**Content providers:**

Un furnizor de conţinut gestionează un set comun de date. Puteţi stoca datele în sistemul de fişiere, o bază de date SQLite, pe web, sau orice alt loc de stocare persistent de unde aplicația le poate accesa. Prin furnizorul de conţinut, alte aplicaţii pot interoga sau chiar modifica datele (în cazul în care furnizorul de conţinut curent permite acest lucru). De exemplu, sistemul Android oferă un furnizor de conţinut care gestionează informaţiile despre contactele utilizatorului. Ca atare, orice aplicaţie cu permisiuni corespunzătoare poate interoga părti din furnizorul de conţinut (cum ar fi ContactsContract.Data), să citească şi să scrie informaţii despre o anumită persoană.

Furnizorii de conţinut sunt, de asemenea, utili pentru citirea şi scrierea datelor care sunt private aplicației şi nu sunt partajate.

Un furnizor de conţinut este implementat în aplicatie ca o subclasă a clasei ContentProvider şi trebuie să implementeze un set de API-uri standard care să permită altor aplicații sa efectueze tranzacţii.

**Broadcast receiver:**

Un receptor de difuzare este o componentă care răspunde la anunţuri difuzate la nivel de sistem. Multe anunțuri provin din sistem, ca de exemplu, o emisie care a anunţat că ecranul s-a oprit, bateria este slabă sau o imagine a fost capturată. Aplicațiile pot iniţia, de asemenea, transmisii, ca de exemplu, pentru a permite altor aplicaţii sa ştie că unele date au fost descărcate pe dispozitiv şi sunt disponibile pentru ca aceştia să le utilizeze. Receptoarele de difuzare, deşi nu prezintă o interfaţă de utilizator, pot crea o notificare în bara de stare pentru a alerta utilizatorul atunci când are loc un eveniment de difuzare. Mai frecvent, însă, un receptor de difuzare este doar o "poartă" către alte componente şi este destinat pentru a face o cantitate foarte mică de muncă. De exemplu, ar putea iniţia un serviciu pentru a efectua unelele lucruri bazate pe evenimentul în cauză.

Un receptor de difuzare este implementat ca o subclasă a clasei BroadcastReceiver.

Un aspect unic al sistemului Android este că orice aplicaţie poate porni o componenta a altei aplicaţii. De exemplu, dacă se doreste ca utilizatorul să înregistreze o fotografie cu aparatul de fotografiat, există, probabil, o altă aplicaţie care face asta şi aplicatia nouă o poate folosi, în loc să dezvolte o activitate pentru a captura o fotografie. Nu e necesar ca aplicația să includă sau chiar să prezinte un link spre codul din aplicația pentru aparatul de fotografiat. În schimb, putem începe pur şi simplu activitatea în aplicația pentru camera de fotografiat, care surprinde o fotografie. Când este realizată, fotografia este chiar returnată aplicatiei nou dezvoltate, astfel încât se poate folosi. Pentru utilizator, pare ca și în cazul în care camera este de fapt o parte inclusă a aplicației.

Când sistemul porneşte o componentă, el începe procesul pentru acea aplicație (dacă nu este deja in derulare) şi instanțiază clasele necesare pentru acea componenta. De exemplu, dacă un proces începe aplicația camerei, care surprinde o fotografie, această activitate se execută în procesul de care aparţine aplicația camerei, nu în procesul aplicației nou create. Prin urmare, spre deosebire de cele mai multe aplicaţii de pe alte sisteme, aplicaţiile Android nu au un singur punct de intrare (nu există nici o funcţie main(), de exemplu).

Deoarece sistemul rulează fiecare aplicație într-un proces separat, cu permisiunile de fişiere care restricţionează accesul la alte aplicaţii, aplicația poate activa direct o componentă din altă aplicaţie. Sistemul Android, poate realiza lucrul acesta. Deci, pentru a activa o componentă din altă aplicaţie, trebuie să se livreze un mesaj către sistem care specifică intenţia de a începe o anumită componentă din altă aplicație. Sistemul activează atunci componenta de care avem nevoie.

**Activarea Componentelor**

Trei din cele patru tipuri de componente, activităţi, servicii şi broadcast receivers sunt activate printr-un mesaj numit intent (intenție). Intent-urile leagă componentele între ele la runtime, indiferent dacă o componentă aparţine unei aplicații sau alteia.

Pentru activităţi şi servicii, o intenţie defineşte acţiunea de a efectua (de exemplu, de a "vedea" sau "trimite" ceva), şi poate specifica URI datei pentru care sa acţioneze. De exemplu, o intenţie ar putea transmite o cerere pentru o activitate pentru a afişa o imagine sau pentru a deschide o pagină web. În unele cazuri, puteţi începe o activitate pentru a primi un rezultat, caz în care, activitatea întoarce, de asemenea, rezultatul într-un Intent (de exemplu, putem emite un intent de a lăsa utilizatorul să aleagă un contact personal şi să-l întoarcem înapoi – Intent-ul returnat include un URI, ce indică spre contactul ales).

Pentru Broadcast receivers, Intent-ul defineşte pur şi simplu anunţul care este transmis (de exemplu, o emisie pentru a indica faptul că bateria dispozitivului este redusă include doar un şir cunoscut care indică faptul că "bateria este descarcată").

Celălalt tip de componentă, content provider, nu este activat de Intent-uri. Totuși, este activat atunci când este vizat de o cerere a unui ContentResolver.

**Fișierul Manifest**

Înainte ca sistemul Android să poată porni o componentă a aplicatiei, sistemul trebuie să ştie că există acea componentă, prin citirea fişierului aplicaţiei, fișier denumit AndroidManifest.xml (“manifest file”). Aplicația trebuie să declare toate componentele sale în acest fișier, care trebuie să fie la rădăcina directorului aplicației.

Fișierul manifest face o serie de lucruri în plus faţă de declararea componentele aplicației, cum ar fi:

- Identifică orice permisiune de utilizator pe care aplicaţia o necesită, cum ar fi accesul la Internet sau de a accesa contactele utilizatorului sau sistemul de localizare (GPS).

- Declară componente hardware şi software utilizate sau cerute de aplicație, cum ar fi aparatul de fotografiat, servicii de bluetooth sau un ecran multitouch.

- Biblioteci API de care aplicatia trebuie să fie legată, cum ar fi biblioteca Google Maps.

Sarcina principală a manifestului este de a informa sistemul cu privire la componentele aplicatiei. De exemplu, un fişier manifest poate să declare o activitate, după cum urmează:

<?xml version=*"1.0"* encoding=*"utf-8"*?>

<manifest xmlns:android=*"http://schemas.android.com/apk/res/android"*

package=*"com.asentinel.mobile.android"*

android:versionCode=*"1"*

android:versionName=*"1.0"*>

<uses-sdk android:minSdkVersion=*"3"* android:targetSdkVersion=*"4"*/>

<supports-screens

android:largeScreens=*"true"*

android:normalScreens=*"true"*

android:smallScreens=*"true"*

android:anyDensity=*"true"* />

<uses-permission android:name=*"android.permission.INTERNET"* />

<uses-permission android:name=*"android.permission.VIBRATE"* />

<application android:label=*"@string/app\_name"*

android:name=*"AppObject"*

android:icon=*"@drawable/icon"*

android:theme=*"@style/Theme"*

android:debuggable=*"false"*>

<activity android:name=*"InitActivity"*

android:label=*"@string/app\_name"*>

<intent-filter>

<action android:name=*"android.intent.action.MAIN"* />

<category android:name=*"android.intent.category.LAUNCHER"* />

</intent-filter>

</activity>

................

În elementul *<application>*, parametrul „*android:icon*” pointeaza spre resurse pentru a selecta pictograma care identifică aplicația.

În elementul *<activity>*, prin “*android:name*” se specifică numele clasei din subclasa Activity şi prin “*Android:label”* se specifică numele pentru label-ul setat de utilizator ca fiind vizibil pentru activitatea în cauză.

Toate elementele aplicației pot fi declarate în modul următor:

<activity> -- elemente pentru Activităţi

<service> -- elemente pentru Servicii

<receiver> -- elemente pentru Broadcast Receivers

<provider> -- elemente pentru Content Providers

**Resursele aplicației**

O aplicație Android este compusă din mai mult decât cod. Sunt necesare și alte resurse care sunt separate de codul sursă, cum ar fi imagini, fișiere audio, şi toate celelalte cu privire la prezentarea vizuală a aplicației. Putem să definim animaţii, meniuri, stiluri, culori, şi aspectul interfeţei unei activități prin fişiere XML. Utilizarea resurselor unei aplicații face mai uşoară actualizarea diferitelor caracteristici ale aplicației fără a modifica elemente de cod, iar prin furnizarea de seturi de resurse alternative se permite optimizarea aplicației pentru o varietate de configuraţii ale dispositivelor mobile (cum ar fi limbi diferite şi dimensiuni diferite de ecran).

Pentru fiecare resursă pe care o include proiectul Android, instrumentele SDK-ului vor defini un număr întreg unic (ID), pe care îl putem folosi pentru a referenţia resursa din codul aplicației sau din alte resurse definite în XML. De exemplu, dacă aplicația conţine un fişier de imagine numit *logo.png* (salvat în *res / drawable / director*), instrumentele SDK-ul vor genera un ID de resursă numit *R.drawable.logo*, pe care îl putem folosi pentru a referenţia imaginea și o putem insera în interfaţa de utilizator.

Unul dintre cele mai importante aspecte în asigurarea resurselor separate de codul sursă este abilitatea de a furniza resurse alternative pentru configuraţii ale diferitelor dispozitive.

# 2.4 Activități

O activitate este o componenta a aplicației, care oferă un ecran cu care utilizatorii pot interacţiona în scopul de a face ceva, cum ar fi formarea unui număr de telefon, fotografierea, trimiterea unui e-mail sau vizualizarea unei o hărți, etc. Fiecărei activități îi este dată o fereastră în care se construiește interfaţa sa cu utilizatorul. Fereastra umple de obicei ecranul, dar poate fi mai mică decât dimensiunea ecranului şi poate pluti deasupra altor ferestre.

O aplicație constă de obicei din activităţi multiple, care sunt slab legate între ele. De obicei, o activitate într-o aplicaţie este specificată ca fiind "principala" activitate, care este prezentată utilizatorului la lansarea aplicației pentru prima dată. Fiecare activitate poate porni apoi o altă activitate, în scopul de a efectua diferite acţiuni. De fiecare dată când se începe o nouă activitate, activitatea anterioară este oprită, dar sistemul păstrează activitatea într-o stivă. Când începe o activitate nouă, aceasta este pusă pe stivă şi ia prim-planul.

Atunci când o activitate este oprită din cauza faptului că o nouă activitate a început, activitatea oprită este notificată de această schimbare prin metode de apel invers al ciclului de viaţă. Exista mai multe metode callback prin care o activitate ar putea primi notificări, provenite din cauza unei schimbări de stare. Așadar, dacă sistemul creează activitatea, o oprește, o reia sau o distruge, fiecare apel invers oferă posibilitatea de a efectua lucruri specifice care sunt necesare acestei schimbări de stare. De exemplu, atunci când a fost oprită, activitatea ar trebui să elibereze orice obiecte mari, cum ar fi conexiuni de reţea sau de baze de date. Când activitatea este reluată, se pot recupera resursele necesare şi se pot relua acţiunile care au fost întrerupte. Aceste tranziţii sunt toate parte a ciclului de viaţă a unei activităţi.

În continuare sunt prezentate elementele de bază ale modului de construire şi de folosire a unei activități, inclusiv o discuţie completă a modului în care funcţionează ciclului de viaţă al unei activități, astfel încât să fie cunoscută modalitatea de a manipula corespunzător ciclul de viață al unei activități.

Ciclul de viață al unei activități poate fi definit de următoarele metode ale clasei *Activity.* Toate aceste pot fi considerate puncte în care utilizatorul poate prelua controlul aplicației și a modului de desfășurare a instrucțiunilor. Toate activitățile trebuie să implementeze metoda *onCreate().* Poate fi utilizată și metoda *onPause()* pentru a salva eventualele modificări la date sau a pregăti oprirea interacțiunii cu utilizatorul.

**public** **class** Activity **extends** ApplicationContext {

**protected** **void** onCreate(Bundle savedInstanceState);

**protected** **void** onStart();

**protected** **void** onRestart();

**protected** **void** onResume();

**protected** **void** onPause();

**protected** **void** onStop();

**protected** **void** onDestroy();

}

Este recomandat cat toate aceste metode să apleze metoda din **super** înainte de a afectua alte operații pentru a păstra logica internă a activității.

**Crearea unei Activități**

Pentru a crea o activitate, trebuie să fie creată o subclasă a clasei *Activity*. În subclasa nou creată este nevoie să fie implementate metodele de callback pe care sistemul le apelează atunci când activitatea face tranziția între diferitele stări ale ciclului său de viaţă, cum ar fi cazul în care activitatea este creată, oprită, reluată sau distrusă. Cele mai importante metode callback sunt:

onCreate()

Această metodă trebuie să fie implementată obligatoriu. Sistemul solicită acest lucru în momentul creării activității. În implementarea metodei, ar trebui să fie initializate componentele esenţiale ale activităţii. Cel mai important, aici este locul în care trebuie să fie apelată metoda *setContentView()* pentru a defini aspectul interfaţei pentru activitatea în cauză.

onPause()

Sistemul apelează această metodă la primul indiciu al faptului că utilizatorul este în curs de a părăsi activitatea respectivă (deşi acest lucru nu înseamnă întotdeauna că activitatea este în curs de a fi distrusă). Aici este locul în care, de obicei, ar trebui să se salveze orice modificări care ar trebui să fie persistate dincolo de sesiunea utilizatorului curent (deoarece utilizatorul ar putea să nu se mai întoarcă la activitatea părăsită în acest moment).

Exista mai multe metode de callback ale ciclul de viaţă care ar trebui să fie utilizate în scopul de a oferi utilizatorului un mod de parcurgere fluid între activităţi şi să manevreze corespunzător întreruperile neaşteptate care pot opri activitatea sau chiar o pot distruge.

**Implementarea unei interfaţe vizuale pentru utilizator**

Interfaţa cu utilizatorul pentru o activitate este asigurată de o ierarhie de obiecte *views* derivate din clasa *View*. Fiecare view controlează un anumit spaţiu rectangular în fereastra de activitate şi poate răspunde la interacţiunea cu utilizatorul. De exemplu, un obiect view poate fi un buton care iniţiază o acţiune atunci când este apasat de către utilizator.

View exitButton = findViewById(R.id.*exit\_button*);

Android oferă o serie de obiecte *views* gata pregătite, care pot fi folosite pentru a proiecta şi organiza aspectul ecranului. "Widgets" sunt obiecte views, care oferă elementele vizuale de pe ecran, cum ar fi un buton, un câmp de text, o casetă sau doar o imagine.

Cel mai comun mod de a defini un aspect folosind views-uri este folosirea unui fişier XML salvat în resursele aplicației. În acest fel, se poate păstra designul interfeţei utilizator separat de codul sursă care defineşte comportamentul activității.

<Button

android:id=*"@+id/exit\_button"*

android:layout\_width=*"fill\_parent"*

android:layout\_height=*"wrap\_content"*

android:text=*"@string/exit\_label"* />

Se poate seta layout-ul ca UI pentru activitatea dorită cu *setContentView()*, care paseaza ID-ul corespunzător găsit în resurse pentru layout-ul selectat.

/\*\* Called when the activity is first created. \*/

@Override

**public** **void** onCreate(Bundle savedInstanceState) {

**super**.onCreate(savedInstanceState);

setContentView(R.layout.*main*);

.......

**Începerea unei activități**

Se poate începe o altă activitate apelând metoda *startActivity()*, prin pasarea unui obiect *Intent* care descrie activitatea care trebuie inițiată. *Intent*-ul specifică fie activitatea exactă care trebuie să înceapă sau descrie tipul de acţiune care se va efectua (şi sistemul selectează activitatea potrivită, care poate aparține chiar și unei alte aplicații). Un obiect *Intent*, de asemenea, poate transporta cantităţi mici de date care urmează să fie utilizate de activitatea care a pornit.

Intent intent = **new** Intent(**this**, About.**class**);

intent.putExtra(User.USER\_NAME, userName);

startActivity(intent);

Atunci când se lucrează în aplicaţia nou creată, este nevoie de multe ori pur şi simplu de a lansa o activitate cunoscută. Se poate face acest lucru prin crearea unui *Intent* care defineşte în mod explicit activitatea care se dorește a fi începută, folosind numele clasei. Cu toate acestea, aplicația ar putea dori, de asemenea, să efectueze o acţiune, cum ar fi trimiterea un e-mail, unui mesaj text, sau actualizarea stării, folosind datele din activitatea curentă. În acest caz, aplicația poate să nu aibă propriile sale activităţi pentru a efectua astfel de acţiuni, astfel încât se pot apela în schimb activităţile altor aplicaţii de pe dispozitiv, care pot efectua acţiunile dorite. În acest caz obiectele *Intent* sunt cu adevărat valoroase, având posibilitatea de a crea un *Intent* care descrie acţiunea pe care urmează să se efectueze, astfel sistemul declanșează activitatea corespunzătoare din altă aplicaţie. Dacă există mai multe activităţi care se pot ocupa de acest *Intent*, atunci utilizatorul poate selecta care activitate să fie utilizată. De exemplu, dacă dorim să permitem utilizatorului să trimită un mesaj de e-mail, se poate crea Intent-ul următor:

String[] recipientArray = {"sorin.slavic@my.fmi.unibuc.ro", "secretariat@fmi.unibuc.ro"};

Intent intent = **new** Intent(Intent.*ACTION\_SEND*);

intent.putExtra(Intent.*EXTRA\_EMAIL*, recipientArray);

startActivity(intent);  
  
 *EXTRA\_EMAIL*  adăugat la *Intent* este un vector de String-uri cu adrese de e-mail la care ar trebui să fie trimis e-mailul. Când o aplicație de e-mail răspunde acestui Intent, citeşte vectorul de String-uri adăugat în plus şi le plasează în campul "*to*" din formularul de compoziţie pentru e-mail. În această situaţie, activitatea aplicaţiei de e-mail începe şi atunci când utilizatorul termină mesajul, se reia activitatea initială.

**Închiderea unei activități**

O activitate se poate închide prin apelul metodei *finish()*. Se poate închide, de asemenea, o activitate separată pe care anterior aţi început-o apelând *finishActivity()*.

**public** **void** onClick(View v) {

**switch** (v.getId()) {

.....

**case** R.id.*about\_button*:

Intent i = **new** Intent(**this**, About.**class**);

startActivity(i);

**break**;

**case** R.id.*exit\_button*:

finish();

**break**;

.....

}

}

**Gestionarea ciclului de viată al unei activități**

Gestionarea ciclului de viaţă al activităţilor prin implementarea metodelor de callback este crucială pentru dezvoltarea unei aplicaţii puternice şi flexibile. Ciclului de viaţă al unei activităţi este direct afectat de asocierea sa cu alte activităţi, sarcina de indeplinit şi stiva.

O activitate poate exista în esenţă, in trei stari:

*Resumed*

Activitatea este în prim-planul ecranului şi utilizatorul este concentrat pe ea. (Această stare este, de asemenea, denumita uneori ca "rulând".)

*Paused*

O altă activitate este în prim-plan, dar activitatea în cauză este încă vizibilă. O altă activitate este vizibilă înaintea acesteia şi această activitate este parţial transparentă sau nu acoperă întregul ecran. O activitate în pauză este complet în viață (obiectul *Activity* este păstrat în memorie, se menţin toate informaţiile de stare şi rămâne ataşat la managerul de ferestre), dar poate fi terminată de către sistem în situațiile extreme în care memoria este insuficientă.

*Stopped*

Activitatea este complet acoperită de o altă activitate (activitate în cauză este acum în "background"). O activitate oprită este, de asemenea, încă în viaţă (obiectul *Activity* este păstrat în memorie, se menţin toate informaţiile de stare şi rămâne ataşat la managerul de ferestre). Cu toate acestea, ea nu mai este vizibilă pentru utilizator şi poate fi distrusă de sistem atunci când altundeva este nevoie de memorie.

În cazul în care o activitate este întreruptă (*paused*) sau oprită (*stopped*), sistemul o poate scoate din memorie, fie cerându-i să se termine (apelând metoda *finish()*) sau pur şi simplu distrugând-o. Atunci când activitatea este deschisă din nou (după ce a fost terminată sau distrusă), acesta trebuie să fie createă din nou.

**Implementarea metodelor ciclului de viață**

Atunci când o activitate tranziţioneaza între diferite stări descrise mai sus, activitatea este notificată prin diferite metode de callback. Toate metodele de callback sunt metode care se pot rescrie pentru a face o muncă adecvată atunci când starea activității se schimbă. În continuare este prezentat un schelet care include fiecare dintre metodele fundamentale ale ciclului de viaţă pentru o activitate:

**public** **class** ExampleActivity **extends** Activity {

@Override

**public** **void** onCreate(Bundle savedInstanceState) {

**super**.onCreate(savedInstanceState);

// Activitatea este creată. Vom introduce metode pentru crearea View-ului

}

@Override

**protected** **void** onStart() {

**super**.onStart();

// Activitatea urmează să fie adusă în prim-plan.

}

@Override

**protected** **void** onResume() {

**super**.onResume();

// Activitatea este acum vizibilă – starea resumed.

}

@Override

**protected** **void** onPause() {

**super**.onPause();

// Altă activitate este în prim-plan – starea paused.

}

@Override

**protected** **void** onStop() {

**super**.onStop();

// Activitatea nu mai este vizibilă - starea stopped.

}

@Override

**protected** **void** onDestroy() {

**super**.onDestroy();

// Activitatea urmează să fie distrusă. Trebuie să salvăm datele necesare.

}

}

Luate împreună, aceste metode defineasc întregul ciclu de viaţă al unei activităţi. Prin implementarea acestor metode se pot monitoriza trei bucle imbricate în ciclul de viaţă al unei activitati:

- Toată durata de viaţă a unei activităţi se întâmplă între apelul metodei *onCreate()* şi apelul metodei *onDestroy()*. Prin metoda *onCreate()* activitatea ar trebui să efectueze inițializarea lucrurilor "globale" (cum ar fi definirea layout-ului) şi eliberarea tuturor resurselor rămase prin metoda *onDestroy()*. De exemplu, în cazul în care activitatea are un fir de execuție care rulează în fundal pentru a descărca date din reţea, s-ar putea crea acest Thread în *onCreate()* şi se va opri apoi în *onDestroy()*.

- Durata de viaţă vizibilă a unei activităţi se întâmplă între apelul metodei *onStart()* şi apelul metodei *onStop()*. În acest timp, utilizatorul poate vedea activitatea pe ecran şi poate interacţiona cu ea. De exemplu, *onStop()* este apelată când începe o nouă activitate şi aceasta nu mai este vizibilă. Între aceste două metode, se pot păstra resursele care sunt necesare pentru a pune activitatea la dispoziția utilizatorului.

- Durata de viață în prim-plan pentru o activitate se întâmplă între apelul metodei *onResume()* şi apelul metodei *onPause()*. În acest timp, activitatea este în fața tuturor celorlalte activităţi de pe ecran şi este în atenția utilizatorului. O activitate poate frecvent să facă tranziţia în ți afara prim-plan-ului, de exemplu, *onPause()* este apelată atunci când dispozitivul este în stare de adormire sau atunci când apare un dialog. Deoarece această stare de tranziţie poate apărea de multe ori, codul în aceste două metode ar trebui să fie destul de uşor și simple pentru a evita tranziţiile lente, care ar face ca utilizatorul să aştepte un timp îndelungat.

Figura următoare ilustrează aceste bucle şi căile prin care o activitate poate tranziționa între stări. Dreptunghiurile reprezintă metodele de callback care pot fi implemente pentru a efectua operaţiuni în cazul în care activitatea face tranziții între stări.



*Fig 2.3.1 Ciclul de viată al unei* ***activități***

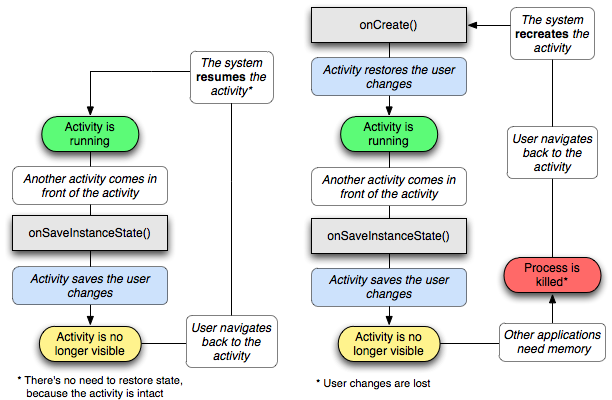
Aceleaşi metode de callback ale ciclul de viaţă sunt enumerate în tabelul urmator, care descrie fiecare dintre metodele de callback în detaliu şi le localizează în cadrul ciclului de viaţă al activității de ansamblu, inclusiv dacă sistemul poate distruge activitatea după ce metoda de callback este completă.

| Metodă | | | Descriere | Urmează |
| --- | --- | --- | --- | --- |
| onCreate() | | | Apelată atunci când activitatea este creată pentru prima dată. Aceasta este locul în care ar trebui să fie facute toate inițializările datelor statice - crearea de vizualizări, şi aşa mai departe. Aceastei metode îi este pasat ca parametru un obiect *Bundle* care conţine starea anterioara a activităţii, în cazul în care a fost salvată acea stare.  Întotdeauna este urmată de metoda *onStart()*. | onStart() |
|  | onRestart() | | Apelată după ce activitatea a fost oprită, chiar înainte de a fi începuta din nou.  Întotdeauna este urmată de metoda *onStart().* | onStart() |
| onStart() | | Apelată chiar înainte ca activitatea să devină vizibilă pentru utilizator.  Întotdeauna este urmată de *onResume()*. | onResume() |
|  | onResume() | Apelată chiar înainte ca activitatea sa înceapa interacţiunea cu utilizatorul. În acest moment, activitatea este prima în stivă.  Întotdeauna este urmată de *onPause()*. | onPause() |
| onPause() | Apelată atunci când sistemul este pe cale să înceapă reluarea altei activități. Această metodă este de obicei folosită pentru salva date, pentru a oprii animaţii şi alte lucruri care pot fi consumatoare de CPU. Acestă metoda ar trebui să facă tot ce face foarte repede, pentru că activitatea următoare nu va fi reluată până când aceasta nu se termină.  Este urmată fie de *onResume()*, în cazul în care activitatea se întoarce în prim-plan, sau de *onStop()*, în cazul în care devine invizibilă pentru utilizator. | onResume()  sau onStop() |
| onStop() | | Apelată atunci când activitatea nu mai este vizibilă pentru utilizator. Acest lucru se poate întâmpla deoarece a fost distrusă sau o altă activitate (una existentă sau una nouă) a fost reluată.  Este urmată fie de *onRestart()*, în cazul în care activitatea se întoarce pentru a interacţiona cu utilizatorul sau de *onDestroy()*, în cazul în care această activitate se va distruge. | onRestart()  sau onDestroy() |
| onDestroy() | | | Apelată înainte ca activitatea să fie distrusă. Acesta este apelul final pe care activitatea îl va primi. Aceasta ar putea fi apelată, fie pentru că această activitate este terminată sau pentru că sistemul distruge temporar acestastă activitate pentru a economisi spaţiu. | nimic |

**Salvarea stării**

Atunci când o activitate este întreruptă sau oprită, starea activităţii este reţinută. Acest lucru este adevărat pentru că obiectul *Activity* este încă păstrat în memorie în aceste două cazuri. Toate informaţiile cu privire la membrii şi la stadiul actual al activității sunt încă disponibile. Orice modificări făcute de utilizator în cadrul activităţii sunt păstrate în memorie, astfel încât atunci când activitatea se întoarce în prim-plan (atunci când "iși revine"), aceste schimbări sunt încă valabile.

Cele două moduri prin care o activitate se întoarce în prim-planul utilizatorului cu starea intactă sunt urmatoaele: fie activitatea este oprită, apoi reluată şi starea rămâne intactă (stânga), fie activitatea este distrusă, apoi recreată şi activitatea trebuie să restabilească starea anterioară (dreapta).



*Fig 2.3.2 Visibilitatea unei* ***activități***

Cu toate acestea, atunci când sistemul distruge o activitate cu scopul de a recupera memoria, obiectul *Activity* este distrus, astfel încât sistemul nu poate relua activitatea pur şi simplu avand starea intactă. În schimb, sistemul trebuie să recreeze obiectul Activity în cazul în care utilizatorul navighează înapoi. Utilizatorul nu are cunoştinţă că sistemul a distrus şi a recreat activitatea şi, prin urmare, probabil, se aşteaptă ca activitatea să fie exact aşa cum a fost. În această situaţie, se poate asigura ca informaţiile importante despre starea de activitate vor fi păstrate prin implementarea unei metode de callback suplimentare. Această metodă permite salvarea de informaţii despre starea de activitate. Apoi această stare poate fi restaurată atunci când sistemul recreează activitatea.

Metoda de callback în care se salvaează informaţii despre starea actuală a activităţii este *onSaveInstanceState().* Sistemul apelează această metodă înainte ca activitatea să fie vulnerabilă la a fi distrusă și îi pasează un obiect *Bundle*. *Bundle*-ul este locul în care se pot stoca informaţii de stare despre activitate ca perechi nume-valoare, folosind metode cum ar fi *putString()*. Apoi, în cazul în care sistemul distruge procesul activității şi utilizatorul navighează înapoi la activitate, sistemul pasează obiectul *Bundle* metodei *onCreate()*, astfel încât se poate restabili starea memorată la *onSaveInstanceState().* Dacă nu există informaţii de restaurat, obiectul Bundle pasat metodei *onCreate()* este nul.

@Override

**protected** **void** onSaveInstanceState(Bundle outState) {

**super**.onSaveInstanceState(outState);

outState.putInt(*STATE\_PAGE*, mCurrentPage);

}

**Coordonarea activităţilor**

Când o activitate începe o altă activitate, ambele tranzitează între stările ciclului de viaţă. Prima activitate este în starea de pauză şi se opreşte (deşi, nu se va opri dacă este încă vizibilă în fundal), în timp ce altă activitate este creată. În cazul în care aceste activităţi împart date salvate pe disc sau în altă parte, este important să înţelegem că prima activitate nu este complet oprită înainte ca a doua să fie creată. Mai degrabă, procesul de pornire a celei de-a doua activități se suprapune cu procesul de oprire a primei activități.

Ordinea metodelor de callback ale ciclului de viaţă este bine definită, în special atunci când cele două activităţi sunt în acelaşi proces, și una o începe pe cealaltă. Mai jos este detaliată ordinea operațiilor care apar atunci când *Activitatea* **A** începe *Activitatea* **B**:

1. Se execută metoda *onPause()* din activitatea **A**.

2. Se execută în secvență metodele *onCreate()*, *onStart()*, și *onResume()* ale activității **B** (activitatea **B** are acum prim-planul).

3. Apoi, dacă activitatea **A** nu mai este vizibilă pe ecran, metoda sa *onStop()* se execută.

**Sarcini și stiva de revenire**

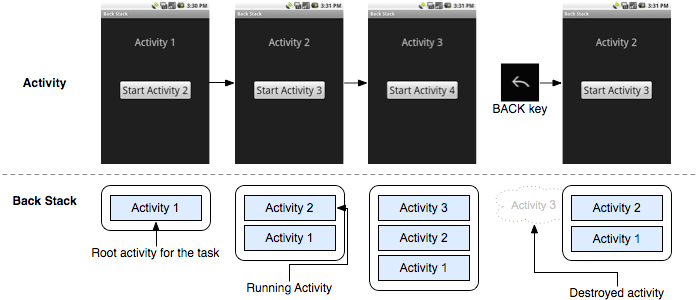
O aplicație conţine, de obicei, activităţi multiple. Fiecare activitate ar trebui să fie concepută în jurul unui anumit tip de acţiune pe care userul o poate efectua şi poate începe alte activităţi. De exemplu, o aplicaţie de e-mail ar putea avea o activitate pentru a afişa o listă de e-mail noi. Când utilizatorul selectează un e-mail, o nouă activitate se deschide pentru a vedea acel e-mail.

O *sarcină* este o colecţie de activităţi cu care utilizatorii interacţionează atunci când efectuează un anumit lucru. Activităţile sunt aranjate într-o stivă ("stiva de intoarcere"), în ordinea în care fiecare activitate este deschisă.

Ecranul de Home al dispozitivului este locul de plecare pentru cele mai multe sarcini. Atunci când utilizatorul atinge o pictogramă în Application Launcher (sau o comandă rapidă pe ecranul principal), sarcina acelei aplicații vine în prim-plan. Dacă nu există o sarcină pentru acea aplicație (aplicația nu a fost folosită recent), atunci o nouă sarcină este creată şi "principala" activitate pentru acea aplicație se deschide ca activitate *root* în stivă.

În cazul în care activitatea curentă începe o altă activitate, noua activitate este împinsă în partea de sus a stivei și intră în atenția utilizatorului. Activitatea anterioară rămâne în stiva, dar este oprită. Atunci când o activitate se opreşte, sistemul reţine starea actuală a interfaţei cu utilizatorul pentru activitatea respectivă. În cazul în care utilizatorul apasă tasta *BACK*, activitatea curentă este scoasă din partea de sus a stivei (activitatea este distrusă) şi se reia activitatea anterioară (starea anterioară a interfaţei sale cu utilizatorul este restabilită). Activităţile nu sunt niciodată rearanjate în stivă, doar puse şi scoase din stivă - puse pe stivă atunci când a început activitatea curentă şi scoase atunci când utilizatorul părăsește activitatea folosind tasta *BACK*. Ca atare, stiva funcţionează printr-o structură de formă "ultima venită, prima plecată" – Last In / First Out (LIFO). În figura de mai jos este arătat acest comportament spre a evidenția progresul între activităţi, împreună cu starea curentă a stivei la fiecare punct în timp.

În cazul în care utilizatorul apasă tasta *BACK*, activitatea curentă este distrusă şi se reia activitatea anterioară.

****

*Fig 2.3.3 Stiva de* ***activități***

# 2.5 Interfața grafică

Într-o aplicaţie Android, interfaţa cu utilizatorul este construită folosind obiectele clasei *View* şi *ViewGroup*. Există mai multe tipuri de obiecte de acest fel, fiecare dintre ele fiind descendente ale clasei *View*.

Obiectele clasei *View* sunt unităţile de bază de expresie pentru interfaţa cu utilizatorul pe platforma Android. Clasa *View* serveşte ca bază pentru subclasele numite *"widgets"*, care oferă obiecte pe deplin implementate pentru interfața cu utilizatorul, cum ar fi câmpurile text şi butoanele. Clasa *ViewGroup* serveşte ca bază pentru subclasele numite *"layout-uri"*, care oferă diferite tipuri de arhitectură pentru *layout*, cum ar fi relativ, liniar sau tabelare.

Un obiect al clasei *View* este o structură de date ale cărei proprietăţi stocheaza parametrii layout-ului şi conţinut pentru o anumită zonă dreptunghiulară a ecranului. Un obiect *View* își manevrează propriul comportament, layout, desen, schimbare a focusului, derulare, interpretare a interacţiunilor cu utilizatorul din zona dreptunghiulară a ecranului în care este plasat. Ca un obiect în interfaţa cu utilizatorul, un obiect al clasei *View* este, de asemenea, un punct de interacţiune pentru utilizator şi receptor al evenimentelor de interacțiune.

În tabelul de mai jos sunt descrise principalele metode ce pot fi implementate pentru un obiect de tip *View*.

| **Categorie** | **Metodă** | **Descriere** |
| --- | --- | --- |
| Creație | Constructori | Există o categorie de mai mulți *Constructori* ce pot fi invocați când *View*-ul este creat din cod și o categorie de metode ce sunt apelate când *View*-ul este generată pe baza unui fișier de *Layout*. Constructorii din a doua categorie trebuie să parseze și să aplice toate atributele definite în fișierul de *Layout* XML. |
| onFinishInflate() | Metoda este apelată după ce un *View* și toate sub-elementele sale au fost inserate (*inflated*) din XML. |
| Layout | onMeasure() | Metoda este apelată când se dorește determinarea dimensiunilor pentru acest *View* și toate sub-elementele sale. |
| onLayout() | Metoda este apelată când acel *View* trebuie să își poziționeze sub-elementele sau să le definească dimensiunile. |
| onSizeChanged() | Metoda este apelată când a fost modificată dimensiunea *View*-ului. |
| Desenare | onDraw() | Metoda este apelată când View-ul trebuie să își randeze și afișeze conținutul. |
| Procesarea Evenimentelor | onKeyDown() | Metoda este apelată când apare un nou eveniment de la tastatură. |
| onKeyUp() | Metoda este apelată când apare un eveniment de tip “key up”. |
| onTrackballEvent() | Metoda este apelată când apare un eveniment generat de mișcarea trackball-ului. |
| onTouchEvent() | Metoda este apelată când apare un eveniment generat de “touch screen”. |
| Focus | onFocusChanged() | Metoda este apelată atunci când *View*-ul intră sau iese din focus. |
| onWindowFocusChanged() | Metoda este apelată atunci când fereastra ce conține *View*-ul intră sau iese din focus. |
| Adăugare | onAttachedToWindow() | Metoda este apelată atunci când *View*-ul este atașat la o fereastră. |
| onDetachedFromWindow() | Metoda este apelată atunci când *View*-ul este detașat de la fereastră. |
| onWindowVisibilityChanged() | Metoda este apelată atunci când vizibilitatea ferestrei ce conține *View*-ul se modifică. |

**Declararea Layout-ului**

*Layout*-ul este arhitectura pentru interfaţa cu utilizatorul într-o activitate. El defineşte structura şi deţine toate elementele care apar pentru utilizator. *Layout*-ul se poate declara în două moduri, printr-un XML sau prin instanțierea elemetelor de *layout* la Runtime.

Prin declararea elementelor pentru interfața cu utilizatorul în XML, Android oferă un vocabular direct configurat pentru XML care corespunde claselor şi subclaselor View, cum ar fi cele pentru widget-uri şi layout-uri. Instanțierea elementelor de *layout* la runtime presupune ca aplicatia să își creeze obiectele clasei View şi ViewGroup (şi manipularea proprietăţile lor) din programare.

Android oferă flexibilitatea de a utiliza una sau ambele dintre aceste metode pentru declararea şi gestionarea interfeței cu utilizatorul pentru o aplicaţie. De exemplu, se poate declara *layout*-ul implicit al aplicaţiei în XML, inclusiv elementele de pe ecran care vor apărea în el şi proprietăţile lor. Se poate apoi adăuga și codul în aplicație, care ar modifica starea obiectelor de pe ecran, inclusiv cele declarate în XML.

Avantajul de a declara interfața cu utilizatorul în XML este că astfel se obține o mai bună separare a prezentării aplicației de codul care controlează comportamentul său. Descrierea interfeței cu utilizatorul este externă codului aplicației, ceea ce înseamnă că se poate modifica sau adapta aplicația fără a modifica elemente din codul sursă şi fără a recompila. De exemplu, se pot crea layout-uri XML pentru diferite orientări ale ecranului, diferite dimensiuni ale ecranului dispozitivului, şi pentru limbi diferite. În plus, declararea layout-ului în XML face mai uşor de vizualizat structura interfeței cu utilizatorul, așa că eventualele probleme vor fi mai ușor de rezolvat.

În general, vocabularul XML pentru declararea elementelor pentru interfață urmează îndeaproape structura şi denumirea claselor şi a metodelor. De fapt, corespondentele sunt deseori atât de directe încât se poate ghici ce atribut XML corespunde unei metode de clasă, sau ghici ce clasa corespunde unui element XML dat. În unele cazuri, există mici diferenţe de denumire.

Folosind vocabularul XML pentru Android, se poate proiecta rapid interfața şi elementele de pe ecran conținute în acelaşi mod în care se creează pagini web în HTML - cu o serie de elemente imbricate.

Fiecare layout trebuie să conţină exact un element rădăcină, care trebuie să fie un obiect Views sau ViewGroup. Odată definit elementul rădăcină, se pot adăuga obiecte Views suplimentare sau widget-uri ca elemente copil spre a construi treptat o ierarhie de obiecte Views care definesc aspectul. De exemplu, mai jos este prezentată structura unui layout XML care utilizează un LinearLayout vertical care contine un TextView şi un buton:

<?xml version=*"1.0"* encoding=*"utf-8"*?>

<LinearLayout xmlns:android=*"http://schemas.android.com/apk/res/android"*

android:layout\_width=*"fill\_parent"*

android:layout\_height=*"fill\_parent"*

android:orientation=*"vertical"* >

<TextView android:id=*"@+id/text"*

android:layout\_width=*"wrap\_content"*

android:layout\_height=*"wrap\_content"*

android:text=*"Hello, I am a TextView"* />

<Button android:id=*"@+id/button"*

android:layout\_width=*"wrap\_content"*

android:layout\_height=*"wrap\_content"*

android:text=*"Hello, I am a Button"* />

</LinearLayout>

După declararea layout-ul în XML, se salvează fişierul cu extensia *.xml*, în directorul proiectului Android *res/layout/*, și astfel va fi compilat în mod corespunzător.

**Încărcarea resurselor XML**

Când se compilează aplicatia, fiecare fişier XML pentru layout este compilat într-o resursă View. Această resursă de layout trebuie apelată din codul aplicației, în implementarea metodei *Activity.onCreate().* Acest lucru se face prin apelarea metodei *setContentView(),* pasându-i referinţa pentru *layout*-ului dorit sub forma de: *R.layout.layout\_file\_name*. De exemplu, dacă XML-ul pentru *layout*-ul dorit este salvat ca *main\_layout.xml*, va trebui încărcat în activitate astfel:

@Override

**public** **void** onCreate(Bundle savedInstanceState) {

**super**.onCreate(savedInstanceState);

setContentView(R.layout.main\_layout);

setupViews();

}

Metoda *onCreate(),* este apelată de către sitemul Android atunci când activitatea este lansată.

**Atribute**

Fiecare obiect *View* sau *ViewGroup* poate suporta o proprie varietate de atribute XML. Unele atribute sunt specifice pentru un obiect *View* (de exemplu, *TextView* are atributul *textSize*), dar aceste atribute sunt, de asemenea, moștenite de orice obiect *View* care poate extinde această clasă. Unele atribute sunt comune tuturor obiectelor *View*, deoarece acestea sunt moştenite din clasa de baza *View* (cum ar fi atributul *id*).

**ID-ul**

Orice obiect al clasei *View* poate avea un *ID* de tip integer asociat cu aceasta, pentru a identifica în mod unic obiectul. Atunci când aplicația este compilată, acest ID este referențiat ca un întreg, dar *ID*-ul este de obicei atribuit în fişierul XML pentru *layout* ca un şir, în atributul *id*. Acesta este un atribut XML comun tuturor obiectelor *View* (definit de clasa *View*). Sintaxa pentru un *ID*, în interiorul unui tag XML este:

android:id=*"@+id/exitButton"*

Simbolul *(@)* de la începutul şirului indică analizatorul XML să analizeze și să extindă restul şirului *ID* şi să o identifice ca o resursă *ID*. Simbolul *(+)* arata că acesta este un nume pentru o nouă resursă care trebuie să fie creată şi adăugată resurselor (în fişierul *R.java*). Există o serie de alte resurse *ID* care sunt oferite de Android. Când se face referire la o resursă *ID* oferita de Android, nu mai este nevoie de simbolul ”+”, dar trebuie adăugat pachetul android, astfel:

android:id=*"@android:id/empty"*

Cu acest pachet adăugat, se va face acum referire la *ID*-ul din clasa de resurse *android.R*, fiind astfel mai ușor decât din clasa de resurse locale.

În scopul de a crea obiecte *View* şi de a le referinţia din aplicație, un model comun este urmatorul:

1. Se definește un obiect *View* / *widget* în fişierul pentru layout şi i se atribuie un *ID* unic:

<Button android:id=*"@+id/my\_button"*

android:layout\_width=*"wrap\_content"*

android:layout\_height=*"wrap\_content"*

android:text=*"@string/my\_button\_text"*/>

2. Se creează o instanţă a obiectului *View* şi acesta se preia din layout (de obicei în metoda *onCreate ()*):

Button myButton = (Button) findViewById(R.id.my\_button);

Definirea ID-urilor pentru obiectele din clasa *View* este importantă atunci când se creează un *layout* relativ (*RelativeLayout*). Într-un *RelativeLayout*, obiectele clasei View își pot defini aspectul în raport cu un alt obiect, la care se face referire prin *ID*-ul unic.

**Poziția Layout-ului**

Geometric, un obiect al clasei *View* este un dreptunghi. Un obiect *View* are o locaţie, exprimată ca o pereche de coordonate (stânga și sus), şi două dimensiuni, exprimate ca o lăţime şi o înălţime. Unitatea de masura pentru dimensiune este pixelul.

Se poate obține amplasarea unui obiect *View* prin invocarea metodelor *getLeft()* şi *getTop()*. De exemplu, atunci când *getLeft()* returnează 40, înseamnă că obiectul view este situat la 40 pixeli la dreapta de marginea părintelui său direct.

**Crearea meniurilor**

Meniurile sunt o parte importantă a interfeţei într-o activitate, care oferă utilizatorilor un mod familiar de a efectua acţiuni. Android oferă un cadru simplu pentru a adauga meniuri standard aplicației.

*Meniu de opţiuni* reprezintă colecţia de elemente tip meniu pentru o activitate, care apare când utilizatorul atinge butonul *MENU*. Atunci când aplicația ruleaza pe Android 3.0 sau versiuni ulterioare, aceasta oferă acces rapid pentru a selecta elementele de tip meniu prin plasarea lor direct în bara de acţiuni, ca "elemente de acţiune."

*Submeniu*  reprezintă o listă plutitoare de elemente de tip meniu care apare când utilizatorul atinge un element din meniu care conţine un meniu imbricat.

**Crearea unei resurse pentru meniu**

În locul instanțiereii unui meniu în codul aplicației, ar trebui definit un meniu şi toate elementele sale într-o resursă XML de tip meniu, apoi încărcat ca resursa meniu în codul aplicaţiei. Folosirea unei o resurse de tip meniu pentru a defini meniul este o practică bună, deoarece se separă conţinutul meniului de codul aplicației. Este, de asemenea, mai uşor de a vizualiza structura şi conţinutul unui meniu în XML.

Pentru a crea o resursă de tip meniu, se creează un fişier XML în directorul *res/menu/* din interiorul proiectului şi se construiește meniul cu următoarele elemente:

<menu>  
  Se definește un meniu, care este considerat un container pentru elementele meniului. Un element *<menu>* trebuie să fie nodul rădăcină pentru fişier şi poate conţine una sau mai multe elemente *<item>* şi elemente *<group>*.

<item>

Creează un item, care reprezintă un singur element dintr-un meniu. Acest element poate conţine un element *<menu>* imbricat în scopul de a crea un submeniu.

<group>

Un container opţional, invizibil pentru elementele *<item>.* Acesta permite filtrarea elementelelor din meniu, astfel încât acestea să poată împărtăși prorietăți, cum ar fi vizibilitate.

Iată un exemplu de meniu numit *game\_menu.xml*:

<?xml version=*"1.0"* encoding=*"utf-8"*?>

<menu xmlns:android=*"http://schemas.android.com/apk/res/android"*>

<item android:id=*"@+id/menu\_item\_refresh"*

android:icon=*"@drawable/menu\_refresh"*

android:title = *"@string/menu\_item\_refresh\_label"*>

</item>

<item android:id=*"@+id/menu\_item\_approve\_all"*

android:icon=*"@android:drawable/ic\_menu\_agenda"*

android:title = *"@string/menu\_item\_approve\_all\_label"*>

</item>

</menu>

Acest exemplu definește un meniu cu două elemente. Fiecare articol include atribute:

* android:id - resursă care permite aplicatiei sa poata recunoaşte elementul când utilizatorul o selectează.
* android:icon – face referire la o resurasa de tip drawable care va fi folosita ca pictograma a elementului.
* android:title – o referinţă la un şir care este utilizat ca titlu al elementului.

**Încarcarea unei resurse pentru meniu**

Din codul aplicației, există posibilitatea de a încărca o resursă meniu (convertirea resursei XML într-un obiect programabil) folosind *MenuInflater.inflate().* De exemplu, următorul cod încarcă fisierul XML *game\_menu.xml* definit mai sus, în timpul metodei *onCreateOptionsMenu(),* pentru a fi utilizat ca meniul activității în derulare:

@Override

**public** **boolean** onCreateOptionsMenu(Menu menu) {

getMenuInflater().inflate(R.menu.game\_menu, menu);

**return** **super**.onCreateOptionsMenu(menu);

}

Metoda *getMenuInflater()* returnează un obiect *MenuInflater* pentru activitate. Cu acest obiect, se poate apela *inflate(),* care încarcă o resursă meniu într-un obiectă. În acest exemplu, resursa meniu definita de fisierul *game\_menu.xml* este încărcată în meniul care a fost pasat ca parametru metodei *onCreateOptionsMenu().*

**Răspunsul in urma unei actiuni**

Atunci când utilizatorul selectează un element (un item) din meniu, sistemul apelează metoda *onOptionsItemSelected()* a activității corespunzătoare. Această metodă pasează itemul pe care utilizatorul l-a selectat. Se poate identifica elementul de meniu prin apelul metodei *getItemId(),* care returnează ID-ul elementului din meniu. De exemplu:

@Override

**public** **boolean** onOptionsItemSelected(MenuItem item) {

**switch** (item.getItemId()) {

**case** R.id.new\_game:

newGame();

**return** **true**;

**case** R.id.help:

showHelp();

**return** **true**;

**default**:

**return** **super**.onOptionsItemSelected(item);

}

}

În acest exemplu, *getItemId()* găsește ID-ul elementului selectat din meniu şi blocul switch compară ID-ul selectat cu ID-urile din resurse, atribuite elementelor din meniu în XML. Atunci când o ramură a blocului *switch* găsește cu succes elementul de meniu, se returneaza *true* pentru a indica faptul că elementul din selecţie a fost manipulat.

**Crearea ferestrelor de dialog**

Un dialog este de obicei o mică fereastră care apare în fața activităţii curente. Activitatea de bază pierde prim-planul şi fereastra de dialog preia contactul cu utilizatorul. Dialogurile sunt utilizate în mod normal pentru notificări care ar trebui să întrerupă utilizatorul şi pentru a efectua sarcini scurte care se referă direct la aplicația în curs (cum ar fi o bară de progres sau un prompt de login).

Clasa *Dialog* este clasa de bază pentru crearea de dialoguri. Cu toate acestea, de obicei nu ar trebui ca un obiect să fie instanțiat direct astfel. În schimb, ar trebui să fie utilizată una dintre următoarele subclase:

**public** **class** AlertDialog **extends** Dialog **implements** DialogInterface { …

Un dialog care poate gestiona zero, unul, două, sau trei butoane, şi / sau o listă de elemente selectabile care pot include casetele de selectare sau butoane radio. *AlertDialog* este capabil de a construi cele mai multe feluri de ferestre de dialog pentru utilizator.

**public** **class** ProgressDialog **extends** AlertDialog {

Un dialog care afişează o roată progres sau o bară de progres. Pentru că este o extindere a *AlertDialog*, oferă suport pentru butoane.

**Crearea unui AlertDialog**

Un *AlertDialog* este o extindere a clasei *Dialog*. Este folosită pentru dialoguri care utilizează oricare din următoarele caracteristici:

- Un titlu

- Un text mesaj

- Unul, doua, sau trei butoane

- O listă de elemente selectabile (cu casetele opţional sau butoane radio)

Pentru a crea un *AlertDialog*, se utilizeaza subclasa *AlertDialog.Builder*. Se construieste un *Builder* cu *AlertDialog.Builder(Context),* şi apoi se utilizeaza metodele clasei pentru a defini toate proprietăţile *AlertDialog*-ului. După a fost construit cu *Builder*, se preia obiectul *AlertDialog* cu metoda *create().*

Pentru a crea un *AlertDialog* cu butoane de o parte si de alta ca cel prezentat în imaginea de mai jos, se utilizează setul metodele *set\*Button()* ca în exemplul urmator:

AlertDialog.Builder builder = **new** AlertDialog.Builder(**this**);

builder.setMessage("Are you sure you want to exit?")

.setCancelable(**false**)

.setPositiveButton("Yes", **new** DialogInterface.OnClickListener() {

**public** **void** onClick(DialogInterface dialog, **int** id) {

ThisActivity.**this**.finish();

}

})

.setNegativeButton("No", **new** DialogInterface.OnClickListener() {

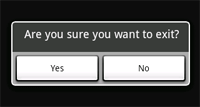
**public** **void** onClick(DialogInterface dialog, **int** id) {

dialog.cancel();

}

});

AlertDialog alert = builder.create();



*Fig 2.4.1 Fereastră* ***Dialog*** *cu două butoane*

Notă: Se poate adăuga numai unul din fiecare tip de buton pentru un *AlertDialog*. Așadar, nu poate exista mai mult de un buton "pozitiv". Acest lucru limitează numărul de butoane posibile la trei: pozitiv, neutru și negativ. Listen

Read phonetically

**Crearea unui ProgressDialog**

Un *ProgressDialog* este o extindere a clasei *AlertDialog* care poate afişa o animaţie tip progres în formă de o roată, pentru o sarcină cu progresul nedefinit, sau o bară de progres, pentru o sarcină care are o progresie definită. Dialogul poate oferi de asemenea butoane, cum ar fi de exemplu, un buton prin care se anulează descărcarea unui fisier.

Deschiderea unui *ProgressDialog* poate fi foarte simplă apelând *ProgressDialog.show(),* ca in exemplul de mai jos:

ProgressDialog dialog = ProgressDialog.*show*(ThisActivity.**this**, "", "Loading. Please wait...", **true**);



*Fig 2.4.2 Un* ***ProgressDialog***

Primul parametru este *Context*-ul aplicației, al doilea este un titlu pentru dialog (lasat gol în acest exemplu), al treilea este mesajul, iar prin ultimul parametru este specificat dacă progresul este nedeterminata (acest lucru este relevant numai atunci când se creează o bară de progres).

**Crearea unui Progress bar**

Pentru a afişa o bara de progres animată:

1. Se Inițializează *ProgressDialog* cu constructorul clasei, *ProgressDialog(Context).*
2. Se setează stilul de progres la *"STYLE\_HORIZONTAL"* cu *setProgressStyle(int)* şi se setează orice alte proprietăţi, cum ar fi mesajul transmis utilizatorului.
3. Pentru a arăta fereastra cu bara de progres, se apelează metoda *show().*
4. Se poate incrementa progresul afişat în bară, apelând fie *setProgress(int),* cu o valoare din procentul total finalizat până în present, fie apeland *incrementProgressBy(int),* și astfel se incrementează cu o valoare care se adăuga la procentajul total finalizat până în prezent.

De exemplu, initializarea ar putea arata astfel:

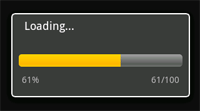
ProgressDialog progressDialog;

progressDialog = **new** ProgressDialog(**this**);

progressDialog.setProgressStyle(ProgressDialog.*STYLE\_HORIZONTAL*);

progressDialog.setMessage("Loading...");

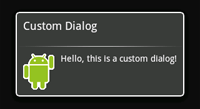
progressDialog.setCancelable(**false**);



*Fig 2.4.3 Un* ***ProgressBar***

**Crearea unei ferestre de dialog custom**

Pentru un design personalizat pentru o fereastră de dialog se poate crea propriul layout pentru fereastra de dialog cu elemente widget.



*Fig 2.4.4 O fereastră* ***custom***

Pentru a crea fereastra de dialog prezentată mai sus utilizează formatul XML de layout prezentat mai jos, salavat într-un fișier XML cu numele *custom\_dialog.xml*:

<LinearLayout xmlns:android=*"http://schemas.android.com/apk/res/android"*

android:id=*"@+id/layout\_root"*

android:orientation=*"horizontal"*

android:layout\_width=*"fill\_parent"*

android:layout\_height=*"fill\_parent"*

android:padding=*"10dp"*

>

<ImageView android:id=*"@+id/image"*

android:layout\_width=*"wrap\_content"*

android:layout\_height=*"fill\_parent"*

android:layout\_marginRight=*"10dp"*

/>

<TextView android:id=*"@+id/text"*

android:layout\_width=*"wrap\_content"*

android:layout\_height=*"fill\_parent"*

android:textColor=*"#FFF"*

/>

</LinearLayout>

Acest XML definește un obiect *ImageView* și un obiect *TextView* în interiorul unui layout de tipul *LinearLayout*. *Layout*-ul de mai sus se setază ca și conținutul *layout*-ului cu elementele *ImageView* şi *TextView*:

Context mContext = getApplicationContext();

Dialog dialog = **new** Dialog(mContext);

dialog.setContentView(R.layout.custom\_dialog);

dialog.setTitle("Custom Dialog");

TextView text = (TextView) dialog.findViewById(R.id.text);

text.setText("Hello, this is a custom dialog!");

ImageView image = (ImageView) dialog.findViewById(R.id.image);

image.setImageResource(R.drawable.android);

După ce se instanțiază obiectul *Dialog*, se setează *layout*-ul cu *setContentView(int),* pasându-i ID-ul pentru resursa de layout. Acum, dialogul are aspectul definit și se pot capta obiecte *View* din layout cu *findViewById (int)* şi conținutul lor poate fi modificat.

3. Comunicarea Client-Server

# 3.1 Introducere

Majoritatea aplicațiilor profesionale disponibile sunt configurate astfel încât să comunice unul cu celălalt. Fie că e vorba doar de o conexiune la baza de date, la transferul de date între aplicații, de transferul de mesaje sau fișiere trebuie să existe un protocol de transfel cunoscut de toate aplicațiile care participă la trimiterea și recepționarea de mesaje.

Modelul Client-Server reprezintă o arhitectură dezvoltată pentru aplicațile distribuite care permit conectarea unor Clienți (aplicații care consumă un serviciu) la unul sau mai multe Servere (furnizorii de serviciu). Aceste două componente esențiale pot comunica pe o rețea de calculatoare diferite sau ambele pot fi instalate pe același sistem. Un calculator Server este o mașină *Host* pe care rulează unul sau mai multe programe de Server la care se conectează aplicațiile Client pentru a recepționa date sau resurse. Un client nu își împarte resursele sau datele. El trimite cereri la Server pentru a accesa conținutul sau serviciile disponibile de pe acesta. Clientul este cel care inițiază sesiunea de comunicare cu serverul care așteaptă aceste request-uri.

Caracteristicile Client/Server descriu o relație în care mai multe programe cooperează în acceași aplicație. Componenta Server permite accesul la o funcție sau un serviciu pentru unul sau mai multe aplicații Client. Trimiterea de email-uri, accesul la baze de date sau la servicii web sunt configurații client/server. Acest model stă la baza comunicației dintre calculatoare (*network computing*).

În Java, există mai multe metode de a realiza conexiunea dintre aplicația client și o aplicție Server. Pachetul de bază în acest limbaj este reprezentat de *java.net.* Folosind clasele și arhitectura disponibilă putem realiza conexiunea între două aplicații fără a cunoaște exact cum va fi realizată fizic conexiunea dintre componente. Sunt utile doar câteva cunoștințe de bază despre rețele: adresa IP, port, protocol, socket, etc.

Pentru a realiza o comunicare eficientă și corectă între două aplicații trebuie ca amândouă să implementez și să respecte același protocol de comunicare. Un protocol reprezintă o convenție de reprezentare a datelor folosită în comunicarea între două calculatoare. Orice informație transmisă prin rețea poate să fie serializată astfel încât să se facă transferul secvențial. Două dintre cele mai utilizate protocoale sunt TCP și UDP.

TCP (Transport Control Protocol) este un protocol de furnizează un flux sigur de date între două calculatoare conectate între ele. Protocolul TCP asigură stabilirea unei conexiuni permanente pe durata comunicației între cele două componente.

UDP (User Datagram Protocol) este un protocol bazat pe pachete independente de date (datagrame) trimise de la o unitate către alta. Spre deosebire de TCP aceast protocol nu stabilește o conexiune permanentă între cele două calculatoare și nu se garantează ca pachetele de date trimise să ajungă la destinatar.

Orice calculator este identificat în mod unic în cadrul unei rețele prin adresa de IP (Internet Protocol). IP-ul poate fi reprezentat și ca valoare pseudo-numerică (set de 4 sau 6 octeți) sau ca adresă IP simbolică (indentificată de un DNS). În Java se utilizează clasa internă *InetAddress* pentru a accesa adresa de IP a calculatorului pe care rulează aplicația.

Pentru majoritatea aplicațiilor nu este suficientă această adresă IP pentru conexiunea cu un Server. Un calculator are în general o singură legătură fizică la rețea. Orice informație destinată unei anumite mașini trebuie să specifice obligatoriu adresa IP a acelei mașini. Pe un calculator pot exista concurent mai multe procese care au stabilite conexiuni în rețea, așteptând diverse infromații sau servicii. Datele trimise către o destinație trebuie să specifice pe lângă adresa IP a calculatorului și o cale de a identifica procesul dorit. Identificarea proceselor se realizează prin intermediul *porturilor.*

Un port este un număr pe 16 biți care identifică în mod unic procesele care rulează pe o anumită mașină. Orice aplicație care realizează o conexiune în rețea va trebui să atașeze un număr de port acelei conexiuni. Valorile pe care le poate lua u număr de port sunt cuprinse între 0 și 65535 (valorile cuprinse între 0 și 1023 sunt rezervate unor servicii sistem).

# 3.2 Java Sockets

Combinația dintre adresa IP și numărul de port este întrebuințată pentru crearea unui termen abstract – *Socket*. Acesta reprezintă canalul de comunicație sau conexiunea dintre cele două componente – client și server. Un client caută serverul și stabilește o conexiune prin care informația poate fi trimise. La finalul comunicației socket-ul este eliberat de client, conexiunea este închisă, astfel încât socket-ul poate fi accesat de alți clienți. Un *socket* furnizează facilități pentru crearea de fluxuri de intrare și ieșire care permit datelor să fie schimbate între client și server.

Fiecăre socket este atașat unui port astfel încât să poată fi identificat unic programul căruia îi sunt destinate datele. Pentru un socket numărul de port poate fi specificat prin program sau asignat de către sistemul de operare. Pentru a trimite cu succes un pachet de date trebuie specificate socket-ului informații despre destinația pachetului – adresa mașinii din rețea și numărul de port care identifică socket-ul.

În Java, socket-urile sunt de două tipuri:

* TCP – implementate de clasele *Socket* și *ServerSocket*;
* UDP – implementate de clasa *DatagramSocket*;

O aplicație de rețea ce folosește socket-uri se încadrează în modelul client – server. Programele de tip server sunt cele care oferă diverse servicii clienților. Acestea sunt în stare de așteptare atâta vreme cât nici un client nu le solicită serviciile. Programele de tip client sunt cele care inițiază comunicația cu un server, solicitând un anumit serviciu. Un server trebuie să fie capabil să trateze mai mulți clienți simultan. Fiecare cerere adresată serverului trebuie tratată, în mod ideal, într-un fir de execuție separat.

Atunci când se stabilește o conexiune, atât clientul, cât și serverul, vor avea câte un socket. Comunicarea efectivă se va realiza între socket-uri. Socket-ul serverului va indica la creare numărul portului, după care va aștepta realizarea de conexiuni cu clienții. Numerele de port pentru socket-urile serverului sunt numere cunoscute pentru programele client. De exemplu dacă o aplicație server permite accesul la un serviciu pe un număr de port, atunci clientul trebuie să acceseze serverul respectiv la portul utilizat.

Putem categorisi protocolul utilizat la conexiunea unui socket în două moduri: protocol orientat pe conexiune și protocol neorientat pe conexiune. În cazul protocolului orientat pe conexiune, un socket client stabilește o conexiune cu un socket server când acesta este creat. O dată stabilită conexiunea, un protocol orientat pe conexiune asigură că datele au fost trimise sigur. Fiecare pachet trimis trebuie să fie și primit. De fiecare dată când un socket trimite un pachet, acesta așteaptă o confirmare că pachetul s-a primit cu succes. Dacă socket-ul nu primește confirmarea într-un timp precizat se va retrimite pachetul respectiv. Socket-ul continuă să trimită pachete până când transmisia este cu succes sau când decide că transmisia este imposibilă.

Pachetele sunt citit de socket-ul destinație în aceeași ordine în care au fost trimise. Datorită modului în care funcționează rețelele, pachetele pot ajunge într-o ordine diferită față de cea în care au fost trimise. Protocolul orientat pe conexiune va permite socket-ului să restabiliească ordinea ăn care au fost trimise pachetele. În comunicarea orientată pe conexiune este necesară participarea simultană a celor două aplicații.

Un protocol neorientat pe conexiune permite cea mai rapidă trimitere a pachetelor. Acesta nu garantează că pachetele care sunt trimise sunt citite în aceeași ordine de programul destinatar. Programul destinatar nu trebuie să fie disponibil în momentul trimiterii mesajelor pentru a recepționa datele. Expeditorul trimite informațiile iar destinatarul le va citi eventual mai târziu după care va răspunde. Este garantată primirea pachetelor știind că acestea nu se vor pierde.

Pachetul *java.net* furnizează două mecanisme de bază pentru accesarea datelor și a resurselor prin intermediul unei rețele. Utilizând un socket, programele pot să transfere grupui de octeți (pachete). Sub acest API sunt definite mai multe implementări de socket, cele mai uzuale *Socket* și *ServerSocket*. *URL (Uniform Resource Locator)* reprezintă un mecanism de nivel înalt utilizat pentru accesarea directă a unei resurse dintr-o rețea.

Un socket TCP pentru client corespunde unei instanțe a clasei *Socket,* iar pentru un server se utilizează un obiect *ServerSocket.* După ce a fost stabilită această conexiune, se pot trimite sau primi fluxuri de la o aplicație la alta. Fiecărui socket îi corespunde un flux de intrare și de ieșire prin care pot recepționa sau trimite date.

Implementarea unu proces care să respecte aceste configurații este cât se poate de simplă. Mai jos am prezentat un exemplu de conexiune Client-Server în care clientul trimite un mesaj la server și așteaptă un răspuns. Serverul interpretează mesajul recepționat și returnează un alt mesaj corespunzător.

**public** **class** Server {

**public** **static** **void** main(String[] args) {

ServerSocket serverSocket;

Socket receiver = **null**;

**try** {

serverSocket = **new** ServerSocket(5678);

receiver = serverSocket.accept();

} **catch** (IOException e) {

System.*out*.println("Could not create the Sockets!" + e.getMessage());

}

**try** {

DataInputStream inputStream =

**new** DataInputStream(receiver.getInputStream());

DataOutputStream outputStream =

**new** DataOutputStream(receiver.getOutputStream());

String unsortedString = inputStream.readUTF();

System.*out*.println(System.*currentTimeMillis*() +

": SERVER -- we received: " + unsortedString);

**char**[] unsortedArray = unsortedString.toCharArray();

Arrays.*sort*(unsortedArray);

String sortedString = **new** String(unsortedArray);

System.*out*.println(System.*currentTimeMillis*() +

": SERVER -- we will send: " + sortedString);

outputStream.writeUTF(sortedString);

receiver.close();

inputStream.close();

outputStream.close();

} **catch** (IOException e) {

System.*out*.println("Coult not read or write!" + e.getMessage());

}

}

}

Aplicația server va fi pornită și va asculta pe port-ul 5678. Pentru un obiect *ServerSocket* definim un alt *Socket* care va fi instanțiat de fiecare dată când se încearcă realizarea unei conexiuni din parte unui Client. Metoda *serverSocket.accept()* blochează sistemul până când această conexiune este realizată. Folosind fluxul de intrare al socket-ului nou deschis vom citi mesajul recepționat. După prelucrarea mesajului, în acest caz sortarea alfabetică a literelor sale, vom trimite un răspuns clientului folosind fluxul de ieșire. După terminarea operațiilor continuăm și închidem conexiunea Socket împreună cu fluxurile de intrare și ieșire.

**public** **class** Client {

**public** **static** **void** main(String[] args) {

String address = "localhost";

**int** port = 5678;

Socket clientSocket = **null**;

**try** {

clientSocket = **new** Socket(address, port);

} **catch** (UnknownHostException e) {

System.*out*.println("Could not connect! - " + e.getMessage());

**return**;

} **catch** (IOException e) {

System.*out*.println("Could not start communication! - " + e.getMessage());

**return**;

}

**try** {

DataInputStream inputStream =

**new** DataInputStream(clientSocket.getInputStream());

DataOutputStream outputStream =

**new** DataOutputStream(clientSocket.getOutputStream());

System.*out*.println(System.*currentTimeMillis*() +

": CLIENT -- we will send: mnertubvcewr");

outputStream.writeUTF("mnertubvcewr");

String sortedString = inputStream.readUTF();

System.*out*.println(System.*currentTimeMillis*() +

": CLIENT -- we received: " + sortedString);

clientSocket.close();

inputStream.close();

outputStream.close();

} **catch** (IOException e) {

System.*out*.println("Could not read or write! - " + e.getMessage());

}

}

}

Pentru aplicația Client definim un nou *Socket* precizând ca parametri pentru constructrori adresa la care se va conecta și portul pe care rulează aplicația Server. Utilizăm cele două fluxuri pentru acest *Socket* similar ca în cazul Serverului, diferența fiind că mai întâi trimitem mesajul folosind fluxul de ieșire urmând ca apoi să îl citim de pe fluxul de intrare. La terminarea comunicației închidem canalul de comunicare și fluxurile definite.

Logarea mesajelor pentru acest exemplu se realizează la consolă și putem să observăm ordinea în care au fost efectuate operațiile.

1338763801013: CLIENT -- we will send: mnertubvcewr

1338763801015: SERVER -- we received: mnertubvcewr

1338763801015: SERVER -- we will send: bceemnrrtuvw

1338763801016: CLIENT -- we received: bceemnrrtuvw

Pornind de la acest API putem configura și realiza conexiuni mai complicate utilizând mai mulți clienți, mesaje mai complicate, implementări pe Server diferite care permit primirea a mai multor mesaje simultan utilizând fire de execuție, etc.

# 3.3 Http Client

În secțiunea anterioară am prezentat o configurare de bază a comunicației Client – Server utilizând Socketi. Conexiunea a fost realizată pe structura TCP/IP, știind structura Serverului. Este posibilă și o conexiune cu un server extern, a cărui configurație nu o cunoaștem. Un exemplu ar fi un Server Web Http de la care vrem să accesăm anumite pagini sau locații (URL-uri). Astfel, suntem interesați doar de configurația clientului, aplicația server fiind ceva extern – posibil implementată în limbaje și moduri diferite.

Un prim exemplu ar fi accesarea unui URL prezentat de un server web pentru o anumită pagină sau o anumită resursă disponibilă. Utilizând aceleași structură de conexiune pentru client prezentată anterior putem încerca să accesăm o adresă web.

**public** **class** ClientHttp {

**public** **static** **void** main(String[] args) **throws** MalformedURLException {

String address = "http://en.wikipedia.org/wiki/Hypertext\_Transfer\_Protocol";

**int** port = 80;

Socket clientSocket = **null**;

**try** {

clientSocket = **new** Socket(**new** URL(address).getHost(), port);

} **catch** (UnknownHostException e) {

System.*out*.println("Could not connect! - " + e.getMessage());

**return**;

} **catch** (IOException e) {

System.*out*.println("Could not start communication! - " + e.getMessage());

**return**;

}

**try** {

InputStreamReader inputStream =

**new** InputStreamReader(clientSocket.getInputStream());

OutputStreamWriter outputStream =

**new** OutputStreamWriter(clientSocket.getOutputStream());

outputStream.write("GET " + **new** URL(address).getFile() +

" HTTP/2.0\r\n\n");

outputStream.flush();

BufferedReader reader = **new** BufferedReader(inputStream);

String line = "";

**while** ((line = reader.readLine()) != **null**)

System.*out*.println(line);

inputStream.close();

clientSocket.close();

} **catch** (IOException e) {

System.*out*.println("Could not read or write! - " + e.getMessage());

}

}

}

Procesul încercă să acceseze o pagină web. Pentru aceasta vom declara un socket care se conectează la adresa respectivă și trimite o cerere specifică HTTP – GET. Dacă Serverul Web a fost implementat respectând aceste specificații atunci în *InputStream-*ul pentru acest socket ar trebui să ni se returneze pagina efectivă Html.

Răspunsul primit pentru acest client este prezentat mai jos. Prima parte reprezintă antetul (header-ul), urmat fiind de fișierul HTML returnat. Cererea noastră nu a avut succes întrucât serverul nu are implement modelul necesar pentru a se putea accesa astfel paginile și resursele sale.

HTTP/1.0 301 Moved Permanently

Date: Sun, 03 Jun 2012 23:54:04 GMT

Server: Apache

Location: http://wikimediafoundation.org/wiki/Hypertext\_Transfer\_Protocol

Content-Length: 271

Content-Type: text/html; charset=iso-8859-1

X-Cache: MISS from sq65.wikimedia.org

X-Cache-Lookup: MISS from sq65.wikimedia.org:3128

X-Cache: MISS from amssq31.esams.wikimedia.org

X-Cache-Lookup: MISS from amssq31.esams.wikimedia.org:3128

X-Cache: MISS from knsq23.knams.wikimedia.org

X-Cache-Lookup: MISS from knsq23.knams.wikimedia.org:80

Connection: close

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">

<html><head>

<title>301 Moved Permanently</title>

</head><body>

<h1>Moved Permanently</h1>

<p>The document has moved <a href="http://wikimediafoundation.org/wiki/Hypertext\_Transfer\_Protocol">here</a>.</p>

</body></html>

Există API-uri special dezvoltate pentru a permite conexiunea pe care o dorim cu acest server Web. Intern, Java are definit în pachetul *java.net* o clasă *URLConnection* care poate realiza conexiunea corectă. Utilizarea corectă a acestui API poate simplifica structura codului și prezintă în plus metode ajutătoare pentru a interpreta mesajul primit ca răspuns.

**import** java.net.URL;

**import** java.net.URLConnection;

**public** **class** URLClient {

**public** **static** **void** main(String[] args) **throws** Exception {

URL urlAddress =

**new** URL("http://en.wikipedia.org/wiki/Hypertext\_Transfer\_Protocol");

URLConnection connection = urlAddress.openConnection();

BufferedReader reader = **new** BufferedReader(

**new** InputStreamReader(connection.getInputStream()));

System.*out*.println(connection.getHeaderFields().toString().

replaceAll("],", "],\n"));

String inputLine;

**while** ((inputLine = reader.readLine()) != **null**)

System.*out*.println(inputLine);

reader.close();

}

}

Răspunsul primit pentru această conexiune coincide cu pagina care apare într-un browser Web în momentul accesării ei. Clasa *URLConnection* are implementate mai multe metode foarte utile, specifice standardului Http, care simplifică structura clientului.

# 3.4 Java Serialization

În limbajul Java avem posibilitatea să encodăm un obiect ca un stream de bytes, urmând ca apoi să reconstruim valoarea obiectului respectiv din această structură. Codarea unui obiect este operația de *Serializare*, opusul acesteia, citirea obiectului, fiind operația de *Deserializare*.

După ce am reușit să serializăm un obiect, valoarea encodată a acestuia poate fi trimisă de la o mașină virtuală la alta sau salvată pe disk pentru a fi deserializată în alt moment. Serializarea reprezintă modelul standard al obiectelor pentru comunicațiile remote dar și structura standard în care persistă componentele *JavaBeans* într-o aplicație.

Java poate realiza automat serializarea sau deserializarea unui obiect. Clasă pentru care am instațiat obiectul respectiv trebuie să implementeze interfața *java.io.Serializable.* Implementând această interfață semnalăm procesului intern Java că obiectele acestei clase pot fi serializate. Interfața nu prezintă nicio metodă definită, dar o clasă serializabilă poate defini anumite metodă cu nume și antete specifice care pot fi utilizate în procesul de serializare sau deserializare.

Se poate utiliza și implementa o altă interfață, java.io.*Externalizable.* Această interfață are definite două metode care sunt utilizate pentru a salva sau citi starea obiectului în momentul respectiv.

**public** **interface** Externalizable **extends** java.io.Serializable {

**void** writeExternal(ObjectOutput out) **throws** IOException;

**void** readExternal(ObjectInput in) **throws** IOException, ClassNotFoundException;

}

Trebuie precizat că nu toate obiectele pot fi serializate. Din acest motiv mecanismul de serializare nu este disponibil default pentru toate obiectele (clasa *Object* nu implementează *Serializable*). Serializând un obiect înseamnă practic să îi salvăm starea la momentul respectiv. Starea unui obiect este reprezentată de valoarea membriilor săi. Astfel există clase care nu prezintă elemente suficiente pentru a fi interpretate cu succes dintr-o stare serializată.

Pe parcursul stadiului unui obiect, dacă starea sa nu a fost modificată, presupune ca nici valoarea sa serializată să nu difere. Aceeași logică se aplică și la versiuni ale unei clase. Dacă într-un design anterior, un obiect definind o anumită stare este serializat, deserializând utilizând o librărie nouă trebuie ca starea obiectului să fie păstrată, indiferent de modificările suferite de clasa respectivă între versiuni. Acest lucru nu este întotdeauna posibil.

Alt motiv pentru care Serializarea nu este implementată pentru toate obiectele este acela că unele clase pot avea membrii privați. Acești membrii trebuie să fie ascunși și inaccesibili în anmite cazuri, dar implementând mecanismul de serializare pe un astfel de obiect facem posibil accesul la valorile respective.

Metoda standard de a serializa un obiect Java este de a salva câmpurile într-un șir de bytes. Primitivele și membrii non-statici sunt encodate în acest stream. Toți membrii, dacă nu sunt definiți *transient,* trebuie să fie și ei Serializabili sau altfel procedure de serializare a obiectului care îi conține nu va putea fi realizată cu succes.

Citirea si scrierea obiectelor (serializarea si deserializarea) se realizeaza utilizand doua clasa Java construite pe baza fluxurilor de intrare si iesire. *ObjectInputStream* si *ObjectInputStream* prezinta cate o metoda *ReadObject* si *WriteObject.* Aceste doua metode realizeaza efectiv serializarea si deserializarea obiectului folosind fluxurile de intrare sau iesire. In functie de fluxurile utilizate putem realiza scriere intr-un fisier, transmiterea printr-un socket, etc.

De exemplu, pornind de la programul utilizat la Sockets putem rescrie astfel încât să permitem trimiterea unui obiect (Student) de la Client la Server și înapoi. Pentru aceasta ne definim mai întâi clasa pe care vrem să o trimitem, care trebuie să implementeze Serialiazable.

**public** **class** Student **implements** Serializable {

**private** **static** **final** **long** *serialVersionUID* = 1234567891011121314L;

**public** **int** studentID;

**public** String firstName;

**public** String lastName;

**public** Date birthDate;

**public** **double** grade;

*SerialVersionUID* este o constantă finală definită pentru a putea diferenția între versiuni ale unei clase. O clasă poate suferi modificări, în structura sa, la membrii sau alte elemente, astfel încât deserializarea sau serializarea să nu mai fie compatibile între versiuni diferite. În acest sens se poate utiliza acest *ServialVersionUID* pentru a identifica și folosi corect o clasă *Serializabilă*. Dacă nu definim această valoare, sistemul Java o va calcula independent folosind *HashCode*-ul calculat pentru membrii sau câmpurile obiectului.

Student student = **new** Student(1, "Sorin", "Slavic",

**new** java.util.~~Date~~(88, 7, 23), **new** Double("9.50"));

ObjectInputStream inputStream = **new** ObjectInputStream(clientSocket.getInputStream());

ObjectOutputStream outputStream =

**new** ObjectOutputStream(clientSocket.getOutputStream());

outputStream.writeObject(student);

outputStream.flush();

Student returnedStudent = **null**;

returnedStudent = (Student) inputStream.readObject();

Codul utilizat de Client este similar celui menționat anterior, folosind Sockets. Orice implementare de *InputStream* sau *OutputStream* sunt compatibile pentru scrierea și citirea de obiecte. Pe partea de *Server* operațiile sunt identice, diferă doar, evident, ordinea în care acestea sunt executate.

ObjectOutputStream outputStream = **new** ObjectOutputStream(receiver.getOutputStream());

ObjectInputStream inputStream = **new** ObjectInputStream(receiver.getInputStream());

Student receivedStudent = (Student) inputStream.readObject();

receivedStudent.setGrade(**new** Double(10));

outputStream.writeObject(receivedStudent);

outputStream.flush();

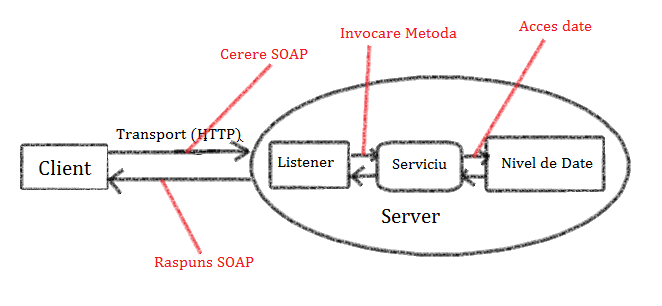
# 3.5 SOAP

Metodele prezentare anterior sunt foarte utile în cazul în care aplicația Client și cea Server sunt dezvoltate pe aceeași platformă (în acest caz Java). Problemele apar în momentul în care vrem să construim un Server care să ofere servicii sau conținut unor clienți dezvoltați pe platforme diferite (alte limbaje de programare, dispozitive mobile, etc). Există mai multe standarde implementate care pot asigura un astfel de transfer de date între aplicații.

SOAP (Simple Object Access Protocol) este un protocol de comunicare bazat pe XML. Acest protocol este utilizat pentru a permite comunicarea a două sisteme diferite, posibil cu componente diferite (sisteme de operare, limbaj de programare). El se bazează pe transferul HTTP deja implementat ca un standard pe toate sistemele de operare.

SOAP definește un set de reguli pentru structurarea mesajelor care pot fi utilizate pentru comunicarea dintre două componente. Dialogul bazat pe Request/Response este foarte util în implementarea stilului RPC (Remote Procedure Call) dintre Client și Server. Este o componentă foarte importantă pentru dezvoltarea aplicaților distribuite care se folosesc de unele servicii sau functionalități disponibile separat.

Un serviciu SOAP este structurat astfel încât se dorește încapsularea logicii executate (fie de baze de date, locale sau orice altă procedură) într-o metodă care să fie apelată în urma citirii mesajului SOAP recepționat. Un astfel de request, în format SOAP, va conține numele metodei care trebuie executată și parametrii necesari rulări. Transportul poate fi realizat folosind HTTP dar există situații în care este implementat folosind SMTP sau alte metode. Un proces de ascultare, care așteaptă mesajele SOAP, citește și interpretează mesajul recepționat și îl transformă și invocă metoda corespunzătoare. Răspunsul acestor metode sunt și ele encodate înapoi în formatul SOAP și retrimise clientului sau procesului care a invocat serviciul.



*Fig 3.5.1 Arhitectura SOAP*

Folosind mesajele construite după structura SOAP înseamnă că vor fi ușor de înțeles și de citit pentru programatori sau utilizatori. Structura ierarhică a XML-ului ajută în elaborarea unor cereri ce reprezintă obiecte complexe, dar mai ales la interpretarea răspunsului care poate fi cosiderat o listă de obiecte sau un arbore, etc. Aceste astepecte fac să fie foarte ușor de manipulat și editat un astfel de mesaj.

SOAP a fost gândit în ideea de a fi folosit ca un protocol de comunicare în special pentru *Serviciile Web*. Un *Serviciu Web* este o metodă ce poate fi apelată de la distanță, făcând legătura între două componente separate (fie peste o rețea Intranet sau chiar Internetul). Serviciile tradiționale, utilizate pentru internet, în marea majoritate, prezintă ca răspuns utilizatorului o pagină web, date sau un conținut în general dezvoltat pentru utilizatori. Serviciile Web sunt dezvoltate pentru a înapoia, în primul rând, un răspuns pentru un anumit calculator sau proces.

Bazat pe noțiunea de *Serviciu Web*, putem interpreta totalitatea serviciilor web disponibile pe internet ca făcând parte din *Service Web.* Multe aplicații, indiferent de profil, structură sau limbajul de programare în care sunt dezvoltate, doresc sau trebuie să aibe o metodă prin care clienții, independenți de aplicația respectivă, să se poate conecta la ei. Un exemplu de aplicație ar fi platforma de tradus de la Google. O aplicație proprie, un website sau un blog, am vrea să fie disponibilă în mai multe limbi pentru utilizatori. Folosind un serviciu web, de exemplu cel implementat de Google, putem trimițând mesaje să ni se returneze interpretarea și traducerea mesajului respectiv în limba dorită. Astfel nu trebuie să implementăm noi o altă componentă distinctă care să ne traducă aplicația sau conținutul paginilor web.

Elementele construite ca Servicii Web stau la baza multor funcționalități din noile generații de aplicații distribuite. Folosind o conexiune cu o bancă, cu un serviciu de transport, cu o agenție de voiaj, cu un magazin online, etc, putem utiliza toate informațiile și serviciile pe care acestea le au disponibile în mod automat și foarte ușor.

Aceste mesaje SOAP pot să fie foarte complexe în unele situații, dar indiferent de structura lor trebuie să existe o aceeși, unică, interpretare și generare a mesajelor între clienți și server. Fiecare Serviciu Web trebuie să prezinte structura și o descriere cât mai utilă a obiectelor și mesajelor acceptate și utilizate, atât ca cereri cât și ca raspuns. WSDL (Web Service Definition/Description Language) și UDDI (Universal Description Discovery and Integration) sunt standarde ce trebuie respectate și folosite în definirea, implementarea și utilizarea unui serviciu web. Fiecare Serviciu Web își va prezenta și face public o descriere a tipului de mesaje utilizate folosind aceste standarde.

WSDL este un limbaj de configurare bazat pe XML (similar SOAP) folosit pentru a descrie funcționalitățile oferite de un *Web Service.* Descrierea efectivă a Serviciului Web, toate metodele implementate, parametrii și structura răspunsului, sunt stocate într-un fișier care poate fi utilizat în implementarea atât a clientului cât și a serverului pentru a valida structura mesajului generat (sau recepționat). Noțiunea este similară cu cea a antetului unei funcții – se prezintă modul în care este invocată metoda, parametrii de intrare și structura răspunsului.

Un mesaj SOAP corect trebuie să respecte în primul rând structura XML. În acest sens pot fi definite DTD-uri, *namesapce-*urile utilizate, XML prolog etc. Structura internă a mesajului SOAP este în continuare împărțită în două componente majore. Documentul SOAP (SOAP Envelope), sub elemetul rădăcină definit poate avea, opțional, un Header definit (utilizat pentru a identifica atribute specifice aplicației sau un eventual utilizator) și corpul efectiv al mesajului (SOAP Body). În SOAP Body va fi definită metoda care se apelează, împreună cu structura parametrilor de intrare.

Ca un exemplu concret voi prezenta implementarea, atât pe server cât și pe client, a comunicației bazată pe SOAP între două componente. Clientul dorește să afle costul și ora la care vor rula filmele la un anumit cinematograf într-o anumită zi.

În Java, ca și în multe alte limbaje de programare, XML-ul este o structură bine știută și folosită. Pachetul *javax.xml* prezintă elementele de bază necesare lucrului cu fișiere XML. În cazul de față folosesc un API disponibil în pachetul core Java (din JRE 6). JAXB2 a fost o componentă separată care recent a fost integrată în pachetele de bază Java. Utilizând clasele și metodele disponibile putem foarte ușor genera, parsa sau interpreta mesaje XML. Pentru a defini mai ușor clasele și componentele utilizate într-un fișier XML cel mai ușor ne este să folosim un DTD (*Document Type Definition*). DTD este un set de reguli de design și structură care definesc un document SGML (*Standard Generalized Markup Language).* Ca subseturi al acestui limbaj SGML se identifică XML, HTML, etc. Structura DTD-ului nu este bine definită în majoritatea cazurilor și din acest motiv, pentru XML-uri, se folosește echivalentul unui astfel de limbaj de *markup –* XML Schema (sau XSD – XML Schema Definition).

Primul pas pentru crearea unei aplicații ce are la bază utilizarea unor mesaje SOAP este definirea structurii XML-ului care va fi utilizat și astfel a claselor din aplicație. Vom utiliza librăria JAXB menționată anterior pentru a parsa și crea automat clasele corespunzătoare, conform definiției din XML. Pentru exemplul nostru vom considera următoarea structură a fișierului XSD.

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<schema targetNamespace=*"http://www.sorin.com/cinema"*

elementFormDefault=*"qualified"*

xmlns:xs=*"http://www.w3.org/2001/XMLSchema"*

xmlns:srn=*"http://www.sorin.com/cinema"*>

<element name=*"GetMoviesRequest"*>

<complexType>

<all>

<element name=*"Cinema"* type=*"string"* maxOccurs=*"1"* minOccurs=*"1"*>

</element>

<element name=*"ViewDate"* type=*"date"* maxOccurs=*"1"* minOccurs=*"1"*>

</element>

</all>

</complexType>

</element>

<element name=*"GetMoviesResponse"*>

<complexType>

<sequence>

<element name=*"item"* type=*"tns:movie"*

maxOccurs=*"unbounded"* minOccurs=*"0"* />

</sequence>

</complexType>

</element>

<complexType name=*"movie"*>

<all>

<element name=*"MovieName"* type=*"string"* maxOccurs=*"1"* minOccurs=*"1"*>

</element>

<element name=*"Price"* type=*"double"* maxOccurs=*"1"* minOccurs=*"1"*>

</element>

<element name=*"StartHour"* type=*"int"* maxOccurs=*"1"* minOccurs=*"1"*>

</element>

</all>

</complexType>

</schema>

Bazat pe structura acestui XSD, componenta utilizată de noi, JAXB, va realiza generarea automată a trei clase. O clasa *GetMovieRequest,* ce corespunde primului element și o clasă *GetMovieResponse* ce corespunde obiectului al doilea. Clasa *Movie* este declarată ca un tip complex, obiectele de acest tip regâsindu-se în lista din *GetMoviesResponse.*

@XmlRootElement(name = "GetMoviesRequest", namespace = "http://www.sorin.com/cinema")

**public** **class** GetMoviesRequest {

@XmlElement(name = "Cinema", namespace = "http://www.sorin.com/cinema",

required = **true**)

**protected** String cinema;

@XmlElement(name = "ViewDate", namespace = "http://www.sorin.com/cinema",

required = **true**)

**protected** Date viewDate;

@XmlRootElement(name = "GetMoviesResponse",namespace = "http://www.sorin.com/cinema")

**public** **class** GetMoviesResponse {

@XmlElement(namespace = "http://www.sorin.com/cinema")

**protected** List<Movie> item;

@XmlType(name = "movie", namespace = "http://www.sorin.com/cmd", propOrder = {})

**public** **class** Movie {

@XmlElement(name = "MovieName", namespace = "http://www.sorin.com/cinema",

required = **true**)

**protected** String movieName;

@XmlElement(name = "Price", namespace = "http://www.sorin.com/cinema")

**protected** **double** price;

@XmlElement(name = "StartHour", namespace = "http://www.sorin.com/cinema")

**protected** **int** startHour;

O aplicație web implementată pe arhitectura Spring WS joacă rolul de Server. Folosind Spring putem defini *Endpoint*-uri în *ApplicationContext.* Noțiunile utilizate aici sunt prezentate în detaliu în capitolele următoare. Le folosim aici doar în contextul exemplului prezentat fără să intrăm în detalii.

Pentru contextul definit componentei de Web Service a aplicației noastre definim câteva bean-uri pe care le vom utiliza în implementarea efectivă a serviciului web.

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<beans xmlns=*"http://www.springframework.org/schema/beans"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans-2.0.xsd"*>

<bean id=*"schema"* class=*"org.springframework.xml.xsd.SimpleXsdSchema"*>

<description>

This bean definition contains the XSD schema.

</description>

<property name=*"xsd"* value=*"/WEB-INF/classes/CinemaObjects.xsd"*/>

</bean>

<!-- WSDL Definition -->

<bean id=*"CinemaObjects"*

class=*"org.springframework.ws.wsdl.wsdl11.DefaultWsdl11Definition"*>

<description>

Dynamically builds a WSDL from the CinemaObjects.xsd.

</description>

<property name=*"schema"* ref=*"schema"*/>

<property name=*"portTypeName"* value=*"CinemaObjects"*/>

<property name=*"targetNamespace"* value=*"http://www.sorin.com/cinema"*/>

<property name=*"locationUri"* value=*"http://localhost:9080/cinema"*/>

</bean>

<bean id=*"marshaller"* class=*"org.springframework.oxm.jaxb.Jaxb2Marshaller"*>

<property name=*"classesToBeBound"*>

<list>

<value>com.sorin.cinema.xml.schema.GetMoviesRequest</value> <value>com.sorin.cinema.xml.schema.GetMoviesResponse</value>

</list>

</property>

</bean>

<bean id=*"loggingInterceptor"*

class=*"org.springframework.ws.soap.server.endpoint.*

*interceptor.SoapEnvelopeLoggingInterceptor"*>

<description>

This interceptor logs the message payload.

</description>

<property name=*"logRequest"* value=*"true"*/>

<property name=*"logResponse"* value=*"true"*/>

</bean>

<!-- Endpoint mappings -->

<bean class=*"org.springframework.ws.server.endpoint.*

*mapping.PayloadRootQNameEndpointMapping"*>

<property name=*"mappings"*>

<props>

<prop key=*"{http://www.sorin.com/cinema}GetMoviesRequest"*>

getMoviesEnpoint

</prop>

</props>

</property>

<property name=*"interceptors"*>

<list>

<ref local=*"loggingInterceptor"*/>

</list>

</property>

</bean>

<!-- Endpoint definitions -->

<bean id=*"loginEnpoint"* class=*"com.sorin.cinema.ws.LoginEndpoint"*>

<constructor-arg ref=*"marshaller"* />

</bean>

</beans>

Am utilizat multe din funcționalitățile build-in Spring WS pentru a realiza mai multe operațiuni utile. În primul rând, utilizând fișierul XSD prezentat anterior putem genera automa WSDL-ul corespunzător mesajelor SOAP utilizate. Acest WSDL este accesibil pe serverul web folosind link-ul definit în *locationUri* din bean-ul respectiv (<bean id=*"CinemaObjects"*):

<property name=*"locationUri"* value=*"http://localhost:9080/cinema"*/>

Calea utilizată pentru a accesa fișierul WSDL va fi:

*localhost:9080/cinama/CinemaObjects.wsdl*

Clienții care dorect să utilizeze aceste servici vor folosi acest fișier wsdl pentru a își declara mesajele sau pentru a le valida.

*Marshaller* este numele generic acordat unor obiecte care generează anumite fișiere standard pe baza unor reguli pornind de la obiecte ale aplicației (în cazul nostru va genera XML bazat pe XSD-ul prezentat). *Unmarshaller* este, în mod evident, obiectul opus. Acesta va realiza parsarea sau interpretarea fișierului XML, generând un obiect al clasei definite.

asdXML Schema (sau XSD – XML Schema Definition).

Folosind infrastructura Spring avem opțiunea să ne declarăm *Endpoint*-urile, obiectele care așteaptă să primeasă o anumită cerere SOAP direct din acest fișier de configurare. Parsarea elementului rădăcină se va face automat pentru un mesaj SOAP XML și astfel se va identifica metoda care este invocată de client. Clasa endpoint în acest caz, întrucât marshaller-ul este deja definit va primi ca parametru de intrare un simplu obiect Java ce corespunde clasei *GetMoviesRequest.*

/\*\*

\* The SOAPMessage Request is marshalled into an Object: <code>requestObject</code>.

\* If there are no issues or exceptions we go ahead and invoke the desired service.

\* **@param** requestObject the marshalled request

\* **@return** an object <code>response</code>, as from the service method;

\*/

**protected** Object invokeInternal(Object requestObject) **throws** Exception {

GetMoviesResponse responseList = movieService.getMovies(

(GetMoviesRequest) requestObject);

**return** responseList;

}

Obiectul returnat este apoi automat parsat și introdus într-un fișier XML, conform schemei XSD care a fost utilizată în instanțierea *Marshaller*-ului.

Structura clientului se bazează pe aceleași tehnologii prezentate care pot automatiza prelucrarea fișierelor XML. Pornind de la WSDL-ul la care avem acces, ne putem genera și aici, cele două clase (*GetMoviesRequest* și *GetMoviesResponse)* care sunt utilizate în XML. Utilizând apoi câteva clase Java utilitare putem genera un mesaj SOAP corespunzător cererii noastre.

/\*\*

\* Constructor using a String <code>url</code> representing the URL address of the WebService we wish to interogate

\* **@param** urlString

\* **@param** reqObject

\* **@throws** UnsupportedOperationException

\* **@throws** SOAPException

\* **@throws** JAXBException

\* **@throws** ParserConfigurationException

\* **@throws** MalformedURLException

\*/

**public** GenericSoapClient(String urlString, Object reqObject) {

connectionFactory = SOAPConnectionFactory.*newInstance*();

messageFactory = MessageFactory.*newInstance*();

transformerFactory = TransformerFactory.*newInstance*();

jaxbContext = JAXBContext.*newInstance*(

MovieClient.*SCHEMA\_PACKAGE*);

documentBuilderFactory = DocumentBuilderFactory.*newInstance*();

documentBuilderFactory.setNamespaceAware(**true**);

documentBuilder = documentBuilderFactory.newDocumentBuilder();

marshaller = jaxbContext.createMarshaller();

marshaller.setProperty(Marshaller.*JAXB\_ENCODING*,

MovieClient.*DEFAULT\_ENCODING*);

marshaller.setProperty(Marshaller.*JAXB\_FORMATTED\_OUTPUT*, Boolean.*TRUE*);

unmarshaller = jaxbContext.createUnmarshaller();

url = **new** URL(urlString);

requestObject = reqObject;

}

Acest constructor primește ca parametrii URL-ul unde se va face apelul serviciului Web și obiectul care se va trimite. În cazul de față acest obiect este o instață a clasei *GetMoviesRequest*. Instanțiem automat un *marshaller* care știe să genereze un fișier XML bazat pe acest obiect.

/\*\*

\*

\* **@return** Marshal the <code>requestObject</code> member of this Object into a SOAPMessage

\* **@throws** SOAPException

\* **@throws** JAXBException

\*/

**private** SOAPMessage createRequest() **throws** SOAPException, JAXBException {

Document document = documentBuilder.newDocument();

marshaller.marshal(requestObject, document);

SOAPMessage message = messageFactory.createMessage();

SOAPBody soapBody = message.getSOAPBody();

soapBody.addDocument(document);

**return** message;

}

/\*\*

\* **@param** response

\* **@return** Unmarshal the SOAPMessage <code>response</code> into an Object (no schema class specified)

\* **@throws** JAXBException

\* **@throws** SOAPException

\*/

**private** Object getResponse(SOAPMessage response)

**throws** JAXBException, SOAPException {

Document document = response.getSOAPBody().extractContentAsDocument();

**return** unmarshaller.unmarshal(document);

}

Cele două metode *createRequest*() și *getResponse*() utilizează obiectele *marshall* definite anterior pentru generarea automată și citirea obiectelor utilizate pentru acest mesaj SOAP. Apelarea efectivă a serverului web înseamnă trimiterea mesajului obținut în urma procesului de create() și returnarea obiectului obținut după prelucrarea și citirea răspunsului.

/\*\*

\* Do a SOAPConnection call to the Web Service specified at the <code>url</code> address

\* First we will create a SOAPMessage request, using the <code>requestObject</code> member of this Object

\* We will call the Web Service and convert and retain the received SOAPMessage in the <code>responseObject</code> member of this Object

\* If there are faults or exceptions we will throw them

\* **@throws** SOAPException

\* **@throws** IOException

\* **@throws** JAXBException

\*/

**public** **void** callWebService() **throws** SOAPException, IOException, JAXBException {

SOAPMessage request = createRequest();

requestSOAPMessage = getSOAPMessage(request);

*log*.debug("callWebService - Request: " + requestSOAPMessage);

*log*.debug("callWebService - To: " + url);

SOAPConnection connection = connectionFactory.createConnection();

SOAPMessage response = connection.call(request, url);

responseSOAPMessage = getSOAPMessage(response);

*log*.debug("callWebService - Response: " + responseSOAPMessage);

**if** (!response.getSOAPBody().hasFault()) {

responseObject = getResponse(response);

hasFault = **false**;

*log*.debug("callWebService - Request completed successfully");

}

**else** {

SOAPFault fault = response.getSOAPBody().getFault();

faultMessage = fault.getFaultString();

hasFault = **true**;

*log*.debug("callWebService - Request completed with errors !");

*log*.debug("callWebService - SOAP Fault Code : " + fault.getFaultCode());

*log*.debug("callWebService - SOAP Fault String : "

+ fault.getFaultString());

}

}

Mai jos am prezentat elementele logate de cele două aplicații. În cazul serverului Web logarea se face automat de *Interceptorul* definit – <bean id=*"loggingInterceptor"* . Pentru client logarea se realizează programatic în procedura de *callWebService*().

**Client –**

2012-05-29 03:30:56,874 - DEBUG GenericSoapClient#callWebService - Request: <?xml version="1.0" encoding="UTF-8"?>

<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">

<SOAP-ENV:Header/>

<SOAP-ENV:Body>

<GetMoviesRequest xmlns="http://www.sorin.com/cmd">

<Cinema>The Light Cinema</Cinema>

<ViewDate>2012-05-29+00:01</ViewDate>

</GetMoviesRequest>

</SOAP-ENV:Body>

</SOAP-ENV:Envelope>

2012-05-29 03:30:56,874 - DEBUG GenericSoapClient#callWebService - To: http://localhost:9080/mobilecmd/ws/

2012-05-29 03:30:57,751 - DEBUG GenericSoapClient#callWebService - Response: <?xml version="1.0" encoding="UTF-8"?>

<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">

<SOAP-ENV:Header/>

<SOAP-ENV:Body>

<ns2:GetMoviesResponse xmlns:ns2="http://www.sorin.com/cmd">

<ns2:item>

<ns2:MovieName>Titanic 3D</ns2:MovieName>

<ns2:Price>20.0</ns2:Price>

<ns2:StartHour>11</ns2:StartHour>

</ns2:item>

<ns2:item>

<ns2:MovieName>The Avengers</ns2:MovieName>

<ns2:Price>28.0</ns2:Price>

<ns2:StartHour>19</ns2:StartHour>

</ns2:item>

</ns2:GetMoviesResponse>

</SOAP-ENV:Body>

</SOAP-ENV:Envelope>

2012-05-29 03:30:57,766 - DEBUG GenericSoapClient#callWebService - Request completed successfully

**Server –**

2012-05-29 03:30:57,477 - DEBUG MessageDispatcherServlet#Published [SimpleXsdSchema{http://www.sorin.com/cinema}] as schema.xsd

2012-05-29 03:30:57,477 - INFO MessageDispatcherServlet#FrameworkServlet 'web-service': initialization completed in 530 ms

2012-05-29 03:30:57,477 - DEBUG MessageDispatcherServlet#Servlet 'web-service' configured successfully

2012-05-29 03:30:57,496 - DEBUG WebServiceMessageReceiverHandlerAdapter#Accepting incoming [org.springframework.ws.transport.http.HttpServletConnection@141e734] to [http://localhost:9080/cinema/ws/]

2012-05-29 03:30:57,677 - DEBUG received#Received request [SaajSoapMessage {http://www.sorin.com/cinema}GetMoviesRequest]

2012-05-29 03:30:57,708 - DEBUG PayloadRootQNameEndpointMapping#Looking up endpoint for [{http://www.sorin.com/cinema}GetMoviesRequest]

2012-05-29 03:30:57,709 - DEBUG SoapMessageDispatcher#Endpoint mapping [org.springframework.ws.server.endpoint.mapping.PayloadRootQNameEndpointMapping@126f585] maps request to endpoint [com.sorin.cinema.ws.GetMoviesEndpoint@129cc0f]

2012-05-29 03:30:57,720 - DEBUG SoapEnvelopeLoggingInterceptor#Request: <SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"><SOAP-ENV:Header/><SOAP-ENV:Body>

<GetMoviesRequest xmlns="http://www.sorin.com/cinema">

<Cinema>The Light Cinema</Cinema>

<ViewDate>2012-05-29+00:01</ViewDate>

</GetMoviesRequest></SOAP-ENV:Body></SOAP-ENV:Envelope>

2012-05-29 03:30:57,720 - DEBUG SoapMessageDispatcher#Testing endpoint adapter [org.springframework.ws.server.endpoint.adapter.MessageEndpointAdapter@b381d8]

2012-05-29 03:30:57,732 - DEBUG LoginEndpoint#Unmarshalled payload request to [com.sorin.cinema.xml.schema.GetMoviesRequest@a204f3]

2012-05-29 03:30:57,735 - DEBUG LoginEndpoint#Marshalling [com.sorin.cinema.xml.schema.GetMoviesResponse@15a22ad] to response payload

2012-05-29 03:30:57,743 - DEBUG SoapEnvelopeLoggingInterceptor#Response: <SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"><SOAP-ENV:Header/><SOAP-ENV:Body>

<ns2:GetMoviesResponse xmlns:ns2="http://www.sorin.com/cinema">

<ns2:item>

<ns2:MovieName>Titanic 3D</ns2:MovieName>

<ns2:Price>20.0</ns2:Price>

<ns2:StartHour>11</ns2:StartHour>

</ns2:item>

<ns2:item>

<ns2:MovieName>The Avengers</ns2:MovieName>

<ns2:Price>28.0</ns2:Price>

<ns2:StartHour>19</ns2:StartHour>

</ns2:item>

</ns2:GetMoviesResponse></SOAP-ENV:Body></SOAP-ENV:Envelope>

2012-05-29 03:30:57,744 - DEBUG sent#Sent response [SaajSoapMessage {http://www.sorin.com/cinema}GetMoviesResponse] for request [SaajSoapMessage {http://www.sorin.com/cinema}GetMoviesRequest]

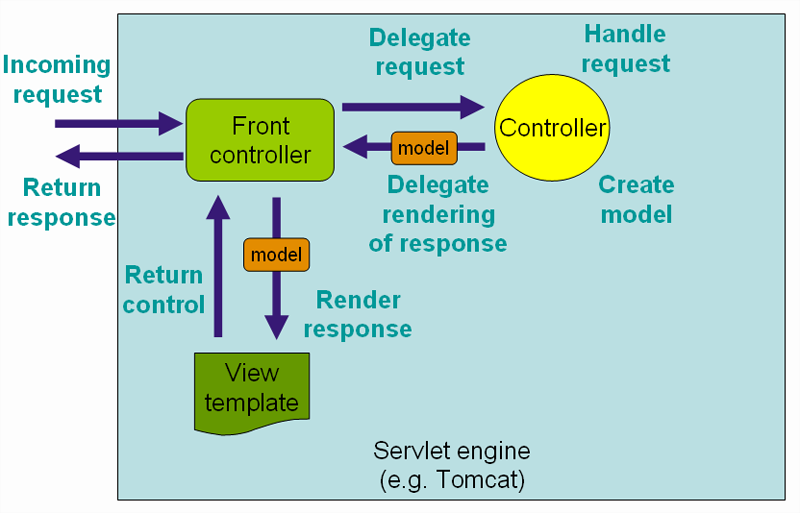
2012-05-29 03:30:57,748 - DEBUG MessageDispatcherServlet#Successfully completed request

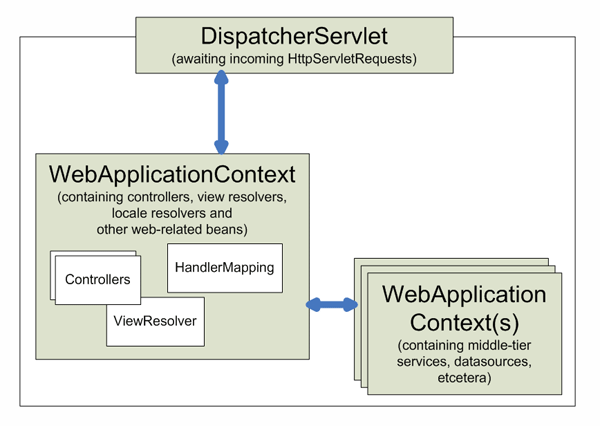
4. Arhitectura Spring - Web

Spring - MVC

IoC – Dependecy mangement

Web Services





5. Aplicație – Mobile CMD

Pentru a face o probă practică a noțiunilor, arhitecturilor și metodelor de programare prezentate anterior am dezvoltat o serie de aplicații, codate în limbaj Java, JDK 1.6 în mediul de dezvoltare Eclipse 3.7 Indigo (cu mai multe plugin-uri utilizate ce vor fi menționate la momentele oportune).

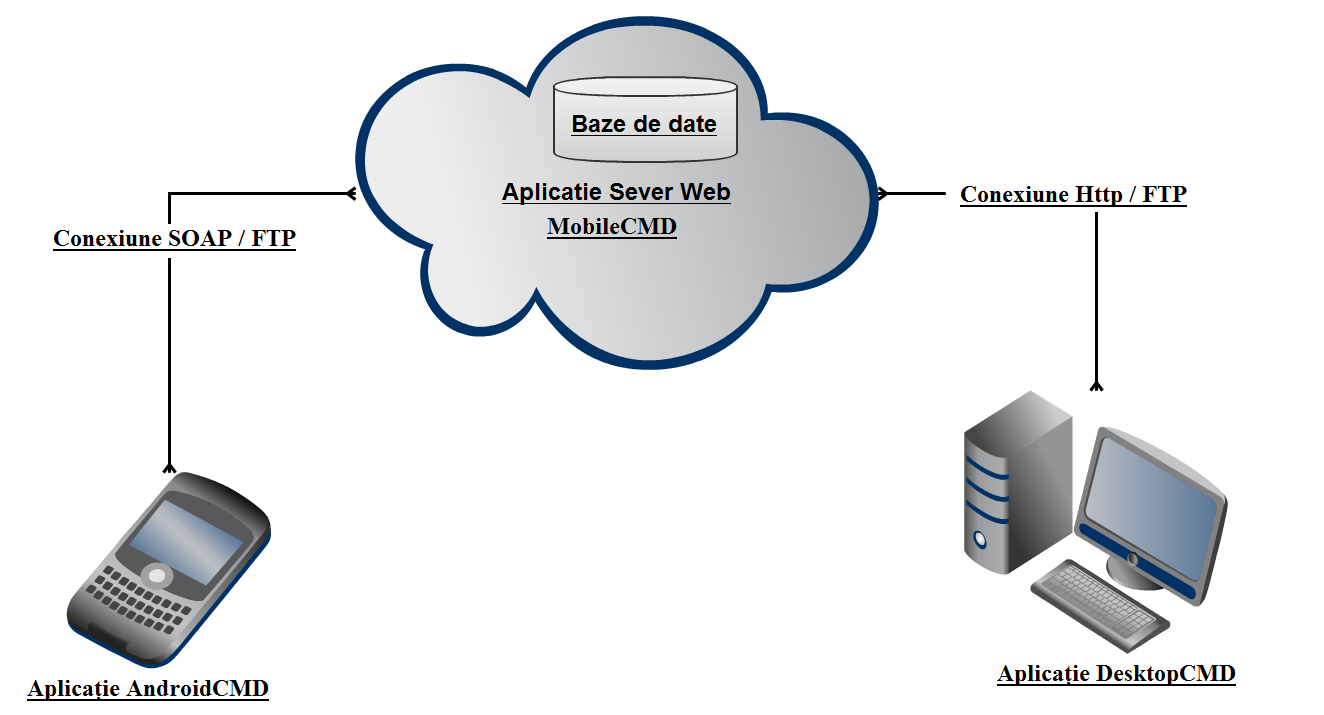
Ideea aflată la baza acestui proiect este aceea de a face o legătură directă între un calculator personal și un telefon mobil. Dorința este de a avea access la anumite fișiere de pe un calculator conectat la aplicație (pentru a downloada), a putea realiza diferite comenzi online, a vedea ce procese rulează și chiar a opri acel computer.

Această conexiune este realizată cu ajutorul a 3 componente:

1. Aplicația mobilă de client – Android CMD
2. Aplicația desktop de client – Desktop CMD
3. Aplicația server – Mobile CMD

Configurația poate fi observată în diagrama de mai jos.

*Fig 5.1 Configurația și conexiunile aplicațiilor*



Un utilizator, după înregistrarea unui cont pe server trebuie să își configureze ambele aplicații de client – cea pe Android și cea pe Desktop (Windows). Clientul de Desktop trebuie să fie pornit pentru a se putea realiza o conexiune între telefonul mobil și calculatorul respectiv. Utilizatorul va alege utilizând interfața aplicației desktop fișierele sau folderele care vor fi accesibile prin Mobil și la fel va preciza dacă, pentru acel calculator, sunt acordate drepturi pentru a rula comenzi windows de sistem (cmdlet-uri). Aplicația Android trimite comenzi către serverul web care apoi la redirecționează spre calculatorul dorit, făcându-se astfel conexiunea cu acesta. Aplicația va putea copia fișierele care sunt accesibile pe calculator și, în cazul în care este permis, va putea trimite comenzi sistem și va primi și vizualiza răspunsurile returnate de la calculator.

Aplicația web are în spate o bază de date MySQL (cu găzduire gratuită) care gestionează utilizatorii, unitățile client Desktop accesibile și fișierele partajate de pe acestea, dispozitivele mobile pe care sunt instalate aplicațiile Android și un istoric al tranzacțiilor și a operațiilor efectuate.

# 5.1 Baza de date MySQL

MySQL este cel mai folosit sistem de gestiune a bazelor de date relaționale. Sistemul rulează pe un server ce permite mai multor utilizatori accesarea bazelor de date.

Pentru aplicația mea am utilizat un serviciu gratuit de găzduire a unei baze de date MySQL – de pe site-ul <http://www.db4free.net> . Baza de date dezvoltată pentru această aplicație gestionează conturile de utilizatori, dispozitivele mobile, clienții desktop și fișierele sau folderele partajate de pe acestea. Fie care utilizator trebuie să se înregistreze, folosind interfața web de pe server. În formularul de înregistrare acesta va introduce numele de utilizator, parola, nume / prenume. Îi va fi acordar un identificator unic care apoi va fi legătura între dispozitivele mobile asociate lui și între aplicațiile de client Desktop. Pentru dispozitivele mobile, utilizatorul va putea să le asocieze un nume, pe lângă alte date specifice telefonului citite automat de aplicație (IMEI, Tipul dispozitivului, număr de telefon, etc). Aceste date pentru dispozitivul mobil vor fi citite automat prima dată când utilizatorul folosește aplicația.

Baza de date va conține înregistrări și pentru toate aplcațiile client Desktop. O astfel de înregistrare este asignată unui utilizator și va conține, pe lângă identificator, numele calculatorului (variabila de sistem Windows), adresa IP (pentru accesarea remote), o valoare care identifică dacă utilizatorul are drepturi la apelarea de comenzi sistem. Calea către fișierele și folderele partajate trebuie gestionate din această bază de date, utilizatorul având posibilitatea de a adăuga sau a sterge astfel de înregistrări. Aceste fișiere corespund unui singur client Desktop și nu unui utilizator.

Mai jos este prezentat schematic design-ul pentru această bază de date și diagrama Entitate – Relație.

CREATE TABLE Users (

UserID INT NOT NULL AUTO\_INCREMENT PRIMARY KEY,

UserName VARCHAR(100) NOT NULL,

Password VARCHAR(100) NOT NULL,

DateJoined DATETIME NOT NULL,

FirstName VARCHAR(100) NOT NULL,

LastName VARCHAR(100) NOT NULL,

);

Acest tabel Users reține utilizatori înregistrați pentru această aplicație. Din formul de înregistrare aceștia își introduc numele de utilizator, parola, numele și prenumele. Identificatorul (*UserID*) reprezintă cheia primară pentru acest tabel și este automat generată de sistemul MySQL. *DateJoined* reprezintă data la care acest utilizator și-a creat contul. Codul utilizat de aplicația web pentru înregistrarea unui client este:

@Override

**public** User registerUser(User user) {

Object[] inParams = {user.getUserName(), user.getPassword(), user.getFirstName(), user.getLastName()};

jdbc.update("insert into Users (UserName, Password, DateJoined, FirstName, LastName) " +

" values (?, ?, CURRENT\_TIMESTAMP, ?, ?)", inParams);

……

CREATE TABLE Clients (

ClientID INT NOT NULL AUTO\_INCREMENT PRIMARY KEY,

UserID INT NOT NULL,

IpAddress VARCHAR(100) NOT NULL,

ComputerName VARCHAR(100),

DateInstalled DATETIME NOT NULL,

LastAccessDate DATETIME,

AllowCommands INT NOT NULL DEFAULT 0,

FOREIGN KEY (UserID) REFERENCES Users(UserID) ON DELETE CASCADE

);

Tabelul *Clients* reține înregistrările pentru aplicațiile client Desktop utilizate de un anumit user. Cheia primară pentru acest tabel este *ClientID, UserID*  fiind cheia externă către tabelul Users. Adresa de IP a calculatorului este o valoare obligatorie întrucât cu aceasta trebuie să se facă legătura dintre aplicația mobil și acest calculator. *ComputerName* reprezintă numele inter al calculatorului sub sistemul Windows iar *DateInstalled* prezintă data la care s-a făcut prima utilizare pentru acel calculator. *LastAccessDate* va prezenta ultima dată când o aplicație client Android a fost conecatată la acest calculator iar *AllowCommands* este utilizată ca o valoare *booleană* prin care se identifică dacă utilizatorul are drepturi să ruleze comenzi Windows de sistem (o valoare mai mare de 0 reprezintă *true, false* fiind identificat prin 0 sau valoare negativă).

CREATE TABLE ClientFiles (

ClientFileID INT NOT NULL AUTO\_INCREMENT PRIMARY KEY,

ClientID INT NOT NULL,

Path VARCHAR(250),

IsFolder INT NOT NULL DEFAULT 0,

IsRecursive INT NOT NULL DEFAULT 0,

DateAdded DATETIME NOT NULL,

LastAccessDate DATETIME,

FOREIGN KEY (ClientID) REFERENCES Clients(ClientID) ON DELETE CASCADE

);

Acest tabel, *ClientFiles,* va reține pentru fiecare aplicație Desktop Client (identificat prin cheia externă *ClientID*) lista de fișiere sau foldere accesibile prin aplicația Android. Avem nevoie de două *flag*-uri (valori booleene) care să ne indice dacă elementele de pe disc, de la calea *Path,* sunt fișiere simple sau foldere, iar dacă sunt foldere dacă avem acces în toate subramurile sau doar pe nivelul respectiv. Pentru fiecare cale reținem și data când a fost adăugată, la fel și data când a fost accesată ultima dată de un client Android.

CREATE TABLE Devices (

DeviceID INT NOT NULL AUTO\_INCREMENT PRIMARY KEY,

DeviceIMEI BIGINT NOT NULL,

UserID INT NOT NULL,

DeviceType VARCHAR(100) NOT NULL,

DeviceName VARCHAR(100),

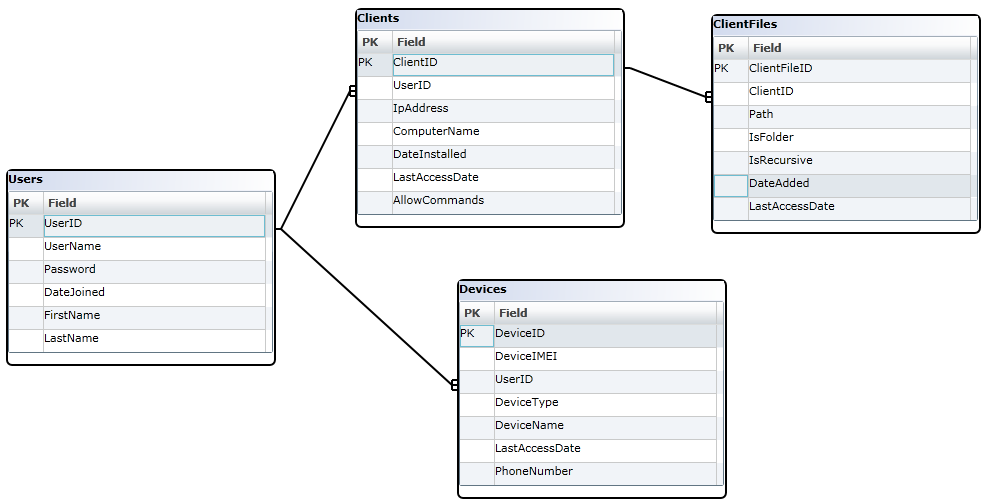
LastAccessDate DATETIME,

PhoneNumber VARCHAR(100),

FOREIGN KEY (UserID) REFERENCES Users(UserID) ON DELETE CASCADE

);

Tabelul care reține informațiile despre dispozitivele Android este tabelul *Devices.* Pentru fiecare înregistrare avem câte o cheie primară, *DeviceID,* dar și codul unic de înregistrare intern pentru acel dispozitiv *DeviceIMEI*. Nu puteam folosit acest cod *IMEI* ca și cheie primară întrucât același dispozitiv poate apărea, în timp, la mai multi utilizatori și astfel ar fi existat două înregistrări distincte cu aceeași cheie primară. Cheia externă către tabelul de utilizatori *Users* este *UserID.* Utilizatorul poate acorda un nume acestui dispozitiv mobil (*DeviceName*), pentru a fi mai ușor de identificat din interfața web când se verifică istoricul unui cont. Pe lângă acest nume în tabel se mai păstrează și tipul dispozitivului (valoare citită automat de proces), ultima dată când acest dispozitiv a fost conectat la un calculator dar și numărul de telefon pentru acest dispozitiv (valoare citită automat).

*Fig 5.1.1 Design-ul Bazei de Date*

# 5.2 Aplicația de Server Web

Aplicația de server Web are mai multe roluri pentru acest proiect. Pe lângă faptul că reprezintă o interfață grafică pentru utilizator, el va și recepționa mesaje și cereri de la clienții de Desktop și Mobile.

Primul punct de acces pentru un utilizator în aplicație îl reprezintă serverul Web prin intermediul căruia utilizatorul se înregistrează în sistem.

Aplicația serverului Web, denumită generic Mobile CMD, rulează pe un server de aplicații Tomcat 7, având la bază o arhitectură bazată pe Spring MVC. Folosind Framework-ul de la Spring se pot dezvolta rapid și ușor aplicații web complexe. Logica se bazează pe faptul că fiecare *View* (fiecare pagină) are un *Controller* (o acțiune în spate) – MVC – *Model View Controller.*

Framework-ul web Spring MVC este configurat în jurul unui *DispatcherServlet* care mapează și redirecționează *request*-uri către *handleri.* Folosind acest framework se pot configura direct aceste mapări, view-urile, localul poate fi setat pentru aplicație, tema utilizată pentru interfața grafică dar și multe altele. Handler-ul default este o interfață Java simplă *Controller,* ce oferă spre utilizare o metodă *handleRequest* ce returnează un obiect de tipul *ModelAndView,* primind ca parametrii de intrare *request*-ul și *response*-ul. Aceasta poate fi folosită direct ca și un *Controller* pentru aplicație dar este recomandată utilizarea unei implementări ierarhice, ca de exemplu, *AbstractController*, *AbstractCommandController* sau *SimpleFormController*. Controller-ii dintr-o aplicație sunt deobicei subclase ale acestora.

Unul din principiile majore utilizate în Spring este “Open for extension, closed for modification” – practic, se dorește ca orice modificare la logica unei clase să fie realizată prin moștenirea ei de altă sub-clase. Cu toate acestea, există foarte multe metode în pachetul de bază al arhitecturi Spring care sunt declare ca *final* și astfel nu pot fi suprascrise în dorința de a le adăuga o logică proprie.

Aplicația Mobile CMD este realizată folosind Spring 3.1.0.

pom.xml:12 - <org.springframework-version>3.1.0.RELEASE</org.springframework-version>

Începând cu versiunea 2.5 de la Spring, pentru Java 5, sau versiuni ulterioare, se pot folosi și adnotații Java care identifică clasele Controller. Această versiune este mult mai utilă, întrucât o acceași clasă Controller poate realiza mai multe acțiuni, maparea către aceste acțiuni făcându-se pentru o metodă.

Pagina de start a aplicației este identificată prin path-ul */home.htm.* În web.xml-ul aplicației este definită de fapt o pagină fizică .jsp, *index.jsp,* ca și *welcome-page.* Întrucât mapările \*.htm corespund controllerilor Spring si nu se putea pune o astfel de adresă virtuală ca și pagină welcome pentru unele servere de aplicații (Tomcat 5, Resin, WebLogic).

<!-- Processes application requests -->

<servlet>

<servlet-name>appServlet</servlet-name>

<servlet-class> org.springframework.web.servlet.DispatcherServlet

</servlet-class>

<init-param>

<param-name>contextConfigLocation</param-name>

<param-value> /WEB-INF/spring/appServlet/servlet-context.xml

</param-value>

</init-param>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>appServlet</servlet-name>

<url-pattern>\*.htm</url-pattern>

</servlet-mapping>

<!-- Home page -->

<welcome-file-list>

<welcome-file>/index.jsp</welcome-file>

</welcome-file-list>

Pagina *index.jsp* nu este utilizată direct, ea realizând doar o redirecționare către *home.htm.*

<%@ include file=*"/WEB-INF/views/common/includes.jsp"* %>

<%-- Redirected because we can't set the welcome page to a virtual URL. --%>

<c:redirect url=*"/home.htm"*/>

Calea *home.htm* este pasată de DispatcherServlet de la Spring către un controller. La momentul pornirii aplicației, Spring își realizează toate mapările interne, fie în funcție de adnotări Java, fie prin valorile definite în fișierele de configurare XML. De exemplu, mai jos avem prezentată logarea la consolă a operațiunilor realizate de infrastructura Spring pentru mapările \**.htm.*

2012-05-27 18:43:19,083 - DEBUG RequestMappingHandlerMapping#Looking for request mappings in application context: WebApplicationContext for namespace 'appServlet-servlet': startup date [Sun May 27 18:43:18 EEST 2012]; parent: Root WebApplicationContext

2012-05-27 18:43:19,110 - INFO RequestMappingHandlerMapping#Mapped "{[/downloadClient.htm],methods=[GET],params=[],headers=[],consumes=[],produces=[],custom=[]}" onto public java.lang.String com.sorin.mobilecmd.DownloadClientController.download(javax.servlet.http.HttpServletRequest,javax.servlet.http.HttpServletResponse) throws java.io.IOException

2012-05-27 18:43:19,110 - INFO RequestMappingHandlerMapping#Mapped "{[/error.htm],methods=[],params=[],headers=[],consumes=[],produces=[],custom=[]}" onto public java.lang.String com.sorin.mobilecmd.ErrorController.error(javax.servlet.http.HttpServletRequest)

2012-05-27 18:43:19,110 - INFO RequestMappingHandlerMapping#Mapped "{[/notfound.htm],methods=[],params=[],headers=[],consumes=[],produces=[],custom=[]}" onto public java.lang.String com.sorin.mobilecmd.ErrorController.notFound(javax.servlet.http.HttpServletRequest)

2012-05-27 18:43:19,113 - INFO RequestMappingHandlerMapping#Mapped "{[/history.htm],methods=[GET],params=[],headers=[],consumes=[],produces=[],custom=[]}" onto public org.springframework.web.servlet.ModelAndView com.sorin.mobilecmd.HistoryController.historyGet(javax.servlet.http.HttpServletRequest)

2012-05-27 18:43:19,115 - INFO RequestMappingHandlerMapping#Mapped "{[/home.htm],methods=[POST],params=[],headers=[],consumes=[],produces=[],custom=[]}" onto public org.springframework.web.servlet.ModelAndView com.sorin.mobilecmd.HomeController.login(javax.servlet.http.HttpServletRequest)

2012-05-27 18:43:19,115 - INFO RequestMappingHandlerMapping#Mapped "{[/home.htm],methods=[GET],params=[],headers=[],consumes=[],produces=[],custom=[]}" onto public org.springframework.web.servlet.ModelAndView com.sorin.mobilecmd.HomeController.homeGet(javax.servlet.http.HttpServletRequest)

2012-05-27 18:43:19,115 - INFO RequestMappingHandlerMapping#Mapped "{[/logout.htm],methods=[],params=[],headers=[],consumes=[],produces=[],custom=[]}" onto public java.lang.String

com.sorin.mobilecmd.LogoutController.logout(javax.servlet.http.HttpServletRequest)

2012-05-27 18:43:19,118 - INFO RequestMappingHandlerMapping#Mapped "{[/mobile.htm],methods=[GET],params=[],headers=[],consumes=[],produces=[],custom=[]}" onto public org.springframework.web.servlet.ModelAndView com.sorin.mobilecmd.MobileController.mobileGet(javax.servlet.http.HttpServletRequest)

2012-05-27 18:43:19,118 - INFO RequestMappingHandlerMapping#Mapped "{[/register.htm],methods=[POST],params=[],headers=[],consumes=[],produces=[],custom=[]}" onto public org.springframework.web.servlet.ModelAndView com.sorin.mobilecmd.RegisterController.register(javax.servlet.http.HttpServletRequest)

2012-05-27 18:43:19,118 - INFO RequestMappingHandlerMapping#Mapped "{[/register.htm],methods=[GET],params=[],headers=[],consumes=[],produces=[],custom=[]}" onto public org.springframework.web.servlet.ModelAndView com.sorin.mobilecmd.RegisterController.registerGet(javax.servlet.http.HttpServletRequest)

La accesarea serverului, folosind link-ul *home.htm,* DispatcherServlet va direcționa acțiunea către controller-ul identificat la pornirea aplicației. În cazul de față, întrucât metoda efectuată de servlet este de tipul GET, vom fi direcționați către metoda HomeController#homeGet(HttpServletRequest).

"{[/home.htm],methods=[GET],params=[],headers=[],consumes=[],produces=[],custom=[]}" onto public org.springframework.web.servlet.ModelAndView com.sorin.mobilecmd.HomeController.homeGet(javax.servlet.http.HttpServletRequest)

În Java, clasa *HomeController* este adnotată cu *@Controller* și astfel Spring o va interpreta ca un controller. Altă adnotație, *@RequestMapping* semnifică path-ul aplicației care va fi direcționat către această metodă.

/\*\*

\* Handles requests for the application home page.

\*/

@Controller

**public** **class** HomeController {

**. . . . . .**

/\*\*

\* Simply selects the home view to render by returning its name.

\*/

@RequestMapping(value = "/home.htm", method = RequestMethod.*GET*)

**public** ModelAndView homeGet(HttpServletRequest request) {

User user = (User) WebUtils.*getSessionAttribute*(request, "user");

**if** (user != **null**) {

Map<String, Object> model = **new** HashMap<String, Object>();

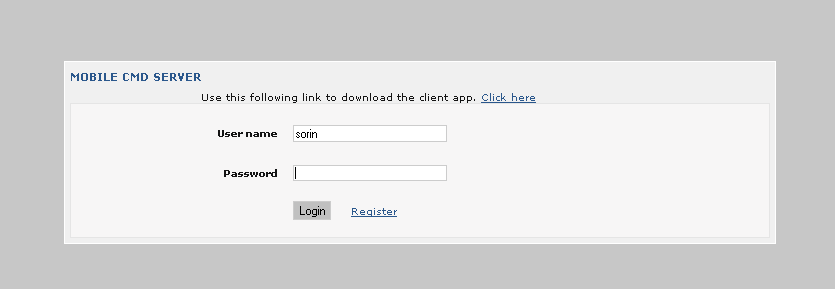
model.put("user", user);

**return** **new** ModelAndView("redirect:history.htm", model);

}

**return** **new** ModelAndView("home");

}



*Fig 5.2.1 Pagina de start a aplicației Web*

Utilizatorul logat în aplicație este reținut pe sesiune, iar în cazul în care se încearcă accesarea paginii *home.htm,* dacă există un utilizator pe sesiune, vom redirecționa acțiunea în acest moment către un alt controller ce este mapat pentru calea *history.htm.*

Din această fereastră de logare utilizatorul are opțiunea de a downloada aplicația client de Desktop.

<p>Use this following link to download the client app.

<a href=*"downloadClient.htm"*>Click here</a> </p>

În cazul acesta vom fi direcționați pe calea *downloadClient.htm,* ajungând la un alt controller definit în aplicație *DownloadClientController.java*. Acest controller aste utilizat pentru a downloada fișierul respectiv.

@Controller

**public** **class** DownloadClientController {

**private** **final** MobileCMDPropertiesFacade propsFacade;

**private** **final** UserService userService;

@Autowired

**public** DownloadClientController(MobileCMDPropertiesFacade propsFacade, UserService userService) {

**this**.propsFacade = propsFacade;

**this**.userService = userService;

}

/\*\*

\* Download

\* **@throws** IOException

\*/

@RequestMapping(value = "/downloadClient.htm", method = RequestMethod.*GET*)

**public** String download(HttpServletRequest request, HttpServletResponse response) **throws** IOException {

String realPath = request.getSession().getServletContext().getRealPath("/");

String filePath = propsFacade.getClientFilePath();

File file = **new** File(realPath + filePath);

FileInputStream fis = **new** FileInputStream(file);

response.setHeader("Content-Disposition","attachment; filename=\"" + file.getName() +"\"");

FileCopyUtils.*copy*(fis, response.getOutputStream());

userService.addClientDownload(request.getRemoteAddr());

**return** **null**;

}

}

Spring este dezvoltat ca să permită cu ușurință implementarea noțiunilor de Inversion of Control. Astfel, folosind Dependency Injection, clasele folosite de noi pot fi instanțiate cu toate dependințele pe care urmează să le folosească, dependințe preluate din contextul aplicației. Nu mai suntem obligați să ne atribuim singuri referințele către alte obiecte, simplificând astfel structura codului și eliminând unele metode ce ar fi duplicate.

La pornirea aplicației, framework-ul Spring instanțiază toate componentele definite în *ApplicationContext. ApplicationContext* este un mediu care îndeplinește mai multe roluri într-o aplicație dezvoltată pe arhitectura Spring. Acesta poate ușura munca unui programator întrucât realizează automat instanțierea de *Bean*-uri, legăturile și dependințele dintre ele, configurarea proprietăților și a mesajelor. Declararea unui astfel de *Context* se realizează printr-un fișier XML. În cazul unei aplicații web putem utiliza fișierul *web.xml* pentru a realiza instanțierea contextelor utilizate.

<!-- The definition of the Root Spring Container shared by all Servlets and Filters -->

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>/WEB-INF/spring/root-context.xml</param-value>

</context-param>

<!-- Creates the Spring Container shared by all Servlets and Filters -->

<listener>

<listener-class>org.springframework.web.context.ContextLoaderListener </listener-class>

</listener>

<!-- Processes application requests -->

<servlet>

<servlet-name>appServlet</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet </servlet-class>

<init-param>

<param-name>contextConfigLocation</param-name>

<param-value>/WEB-INF/spring/appServlet/servlet-context.xml </param-value>

</init-param>

<load-on-startup>1</load-on-startup>

</servlet>

<!-- Processes remote HttpInvoker requests -->

<servlet>

<servlet-name>remoting</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet </servlet-class>

<init-param>

<param-name>contextConfigLocation</param-name>

<param-value>/WEB-INF/spring/remote/remote-servlet-context.xml </param-value>

</init-param>

<load-on-startup>2</load-on-startup>

</servlet>

<!-- Defines the Spring-WS MessageDispathcer Servlet-->

<servlet>

<servlet-name>web-service</servlet-name>

<servlet-class> org.springframework.ws.transport.http.MessageDispatcherServlet </servlet-class>

<init-param>

<param-name>contextConfigLocation</param-name>

<param-value>/WEB-INF/spring/ws/web-service-servlet.xml

</param-value>

</init-param>

</servlet>

Pentru această aplicație utilizăm 4 fișiere de configurare XML de context diferite. Fișierul *root-context.xml* este principala componentă a aplicației si aceasta ține legătura cu fișierul de proprietăți și conexiunea la baza de date. Putem considera că acest context nu știe de natura Web a aplicației, adică de Servleții care rulează sau de Serviciile Web accesibile. El îndeplinește rolul de a instanția legăturile interne si cele cu baza de date.

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<beans xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xmlns=*"http://www.springframework.org/schema/beans"*

xmlns:context=*"http://www.springframework.org/schema/context"*

xmlns:util=*"http://www.springframework.org/schema/util"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-3.0.xsd*

[*http://www.springframework.org/schema/context*](http://www.springframework.org/schema/context) *http://www.springframework.org/schema/context/spring-context-3.0.xsd*

[*http://www.springframework.org/schema/util*](http://www.springframework.org/schema/util) *http://www.springframework.org/schema/util/spring-util-3.0.xsd*

[*http://www.springframework.org/schema/util*](http://www.springframework.org/schema/util) *http://www.springframework.org/schema/util/spring-jdbc-3.0.xsd*

*"*>

<context:component-scan base-package=*"com.sorin.mobilecmd.services"*/>

<context:property-placeholder location=*"classpath:mobilecmd.properties"*/>

<bean id=*"mobileCMDPropertiesFacade"* class=*"com.sorin.mobilecmd.util.MobileCMDPropertiesFacade"*>

<constructor-arg>

<util:properties location=*"classpath:mobilecmd.properties"*/>

</constructor-arg>

</bean>

<bean id=*"mysqlDataSource"* class=*"org.apache.commons.dbcp.BasicDataSource"* destroy-method=*"close"*>

<property name=*"driverClassName"* value=*"com.mysql.jdbc.Driver"*/>

<property name=*"url"* value=*"${sqlURL}"*/>

<property name=*"username"* value=*"${userName}"*/>

<property name=*"password"* value=*"${password}"*/>

</bean>

<bean id=*"jdbcTemplate"* class=*"org.springframework.jdbc.core.JdbcTemplate"*>

<property name=*"dataSource"* ref=*"mysqlDataSource"*/>

</bean>

</beans>

Configurația Spring permite definirea elementelor folosind *Adnotări Java* (*@Component*, *@Service*, *@Controller*, etc). Valoare tag-ului <context:component-scan indică framework-ului Spring ce clase sau pachete trebuie să scaneze pentru eventualele elemente configurate cu adnotări. În acest caz, instanțiem serviciile care apoi vor fi folosite de controller-i sau diferite clase din aplicație (în cazul nostrul vom injecta automat un Singleton ce corespunde unui Serviciu în clasele care îl utilizează).

Serviciile sunt configurate pe baza unei interfețe, cu o implementare standard.

**package** com.sorin.mobilecmd.services;

**import** java.util.List;

**....**

**public** **interface** UserService {

....

/\*\*

\* Get the userId for this username

\* **@param** username

\* **@return** the userId or -1 if no such username

\*/

**public** **int** getUserId(String username);

....

Această interfață prezintă metodele disponibile aplicației. Implementarea poate să difere astfel, în funcție de nivelul de baze de date, conexiunea către acesta, arhitectura serviciului web, etc.

**package** com.sorin.mobilecmd.services;

....

@Service

**public** **class** UserServiceImpl **implements** UserService {

**private** **static** **final** Logger *log* = Logger.*getLogger*(UserServiceImpl.**class**);

**private** **final** JdbcTemplate jdbc;

@Autowired

**public** UserServiceImpl(JdbcTemplate jdbc) {

**this**.jdbc = jdbc;

}

....

@Override

**public** **int** getUserId(String username) {

*log*.debug("getUserId - username: " + username);

**try** {

**return** jdbc.queryForInt("select UserID from Users where UPPER(UserName) = UPPER(?)", username);

} **catch** (EmptyResultDataAccessException noUser) {

*log*.debug("getUserId - no user exists for username: " + username);

**return** -1;

}

}

....

Implementarea serviciului este adnotată cu *@Service* pentru semnala arhitecturii Spring că această clasă trebuie să fie considerată internă și să fie reținută în ApplicationContext. Adnotarea *@Autowired* este utilizată pentru a face automat, de către Spring, a legăturii dintre acestă clasă și parametrul necesar. Un obiect al clasei *JdbcTemplate* a fost deja instațiat de către Spring ca un *singleton* în Application Context.

<bean id=*"jdbcTemplate"* class=*"org.springframework.jdbc.core.JdbcTemplate"*>

<property name=*"dataSource"* ref=*"mysqlDataSource"*/>

</bean>

La descoperirea unei metode adnotată cu *@Autowired,* procesul intern de la Spring caută în *Bean Repository*-ul generat de *ApplicationContext,* adică în toate componentele sale, fie instanțiate din XML, fie adnotate corespunzător, obiectele și parametrii necesari acelei metode. Procesul va găsi un obiect din clasa corespunzătoare, în acest caz *JdbcTemplate* și va pasa metodei referința obiectului respectiv. Aceast proces stă la baza noțiunii de Inversion of Control, Dependency Injection – inserarea automată în cod a dependițelor.

În același mod în care implementarea *UserServiceImpl* a profitat de Dependency Injection și a primit referința către conexiunea la baza de date automat, am observat într-un exemplu anterior că și un *Controller, DownloadClientController* a fost instanțiat automat de Spring folosindu-se de acest proces.

@Autowired

**public** DownloadClientController(MobileCMDPropertiesFacade propsFacade, UserService userService) {

**this**.propsFacade = propsFacade;

**this**.userService = userService;

}

În acest caz, chiar implementarea interfeței *UserService* a fost trimisă ca și parametru constructorului, împreună cu un obiect ce permite accesul la fișierul de proprietăți. Clasa *MobileCMDPropertiesFacade* am observat că a fost definită ca și <bean> în *root-context.xml.* Astfel ea a fost instanțiată și avem acces la ea folosindu-ne de *ApplicationContext.*

<bean id=*"mobileCMDPropertiesFacade"* class=*"com.sorin.mobilecmd.util.MobileCMDPropertiesFacade"*>

<constructor-arg>

<util:properties location=*"classpath:mobilecmd.properties"*/>

</constructor-arg>

</bean>

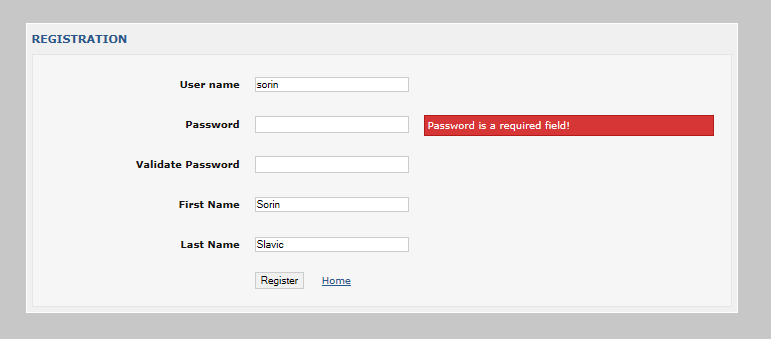
În fișierul *web.xml* avem definite pentru această aplicație trei *Servlet*-uri. Fiecare corespunde unui *DispatcheServlet* de la Spring și fiecare având propriul rol distinct. O componentă a aplicației este declarată pentru recepționarea și direcționarea mesajelor SOAP recepționate de la clientul de Mobil. Altă componentă este utilizată de infrastructura de *HttpInvoker* a Spring – recepționarea și direcționarea apelurilor la interfața *Serviciilor Remote* iar o altă, prima pe care o prezentăm, este cea declarată pentru interfața web a aplicației care recepționează și mesajele *Http* primite de la aplicația de client Desktop. Vom discuta despre fiecare componentă în detaliu în momentul în care vom prezenta și clientul care o apelează (AndroidCMD și DesktopCMD).

De pe pagina de *Home* a aplicației un utilizator are acces la butonul de *Register*, care îl va direcționa către pagina de înregistrare.

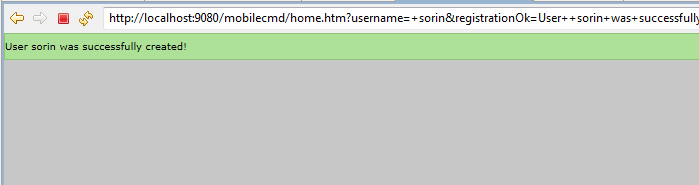
<a href=*”register.htm”*>Register</a><br/>

Pagina pe care acesta o va accesa prezintă un form în care trebuie introduse datele necesare pentru înregistrarea contului de utilizator. Controller-ul asociat acțiuni de *submit* realizează validări asupra datelor introduse la nivelul serverului (numele de utilizator să nu mai fie folosit de alt cont, parola să respecte caracteristicile definite în fișierul de proprietăți și să fie validată corespunzător, etc).

*Fig 5.2.2 Pagina de înregistrare*



În urma înregistrării cu succes, aplicația va redirecționa acțiunea către pagina de *Login*, afișând și un mesaj prin care informează dacă acest cont a fost creat cu succes.

*Fig 5.2.3 Mesajul de succes pentru crearea unui cont*

Arhitectura Spring permite definirea unor *Interceptori* pe request-urile primite de anumiți *Servleti* sau *Controlleri*. În fișierul de configurare *servlet-context.xml* asociat cu *DispatcherServlet*-ul care interpretează request-urile .*htm* avem declarate două tipuri de interceptori.

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<beans:beans xmlns:mvc=*"http://www.springframework.org/schema/mvc"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xmlns:beans=*"http://www.springframework.org/schema/beans"*

xmlns:context=*"http://www.springframework.org/schema/context"*

xsi:schemaLocation=*"http://www.springframework.org/schema/mvc http://www.springframework.org/schema/mvc/spring-mvc-3.0.xsd*

[*http://www.springframework.org/schema/beans*](http://www.springframework.org/schema/beans) *http://www.springframework.org/schema/beans/spring-beans-3.0.xsd*

[*http://www.springframework.org/schema/context*](http://www.springframework.org/schema/context) *http://www.springframework.org/schema/context/spring-context-3.0.xsd"*>

<!-- DispatcherServlet Context: defines this servlet's request-processing infrastructure -->

<!-- Enables the Spring MVC @Controller programming model -->

<mvc:annotation-driven />

<!-- Handles HTTP GET requests for /resources/\*\* by efficiently serving up static resources in the ${webappRoot}/resources directory -->

<mvc:resources mapping=*"/styles/\*\*"* location=*"/styles/"* />

<mvc:resources mapping=*"/resources/\*\*"* location=*"/resources/"* />

<mvc:resources mapping=*"/images/\*\*"* location=*"/images/"* />

<!-- Resolves views selected for rendering by @Controllers to .jsp resources in the /WEB-INF/views directory -->

<beans:bean class=*"org.springframework.web.servlet.view.InternalResourceViewResolver"*>

<beans:property name=*"prefix"* value=*"/WEB-INF/views/"* />

<beans:property name=*"suffix"* value=*".jsp"* />

</beans:bean>

<context:component-scan base-package=*"com.sorin.mobilecmd, com.sorin.mobilecmd.desktop"* />

<mvc:interceptors>

<mvc:interceptor>

<mvc:mapping path=*"/home.htm"* />

<beans:bean class=*"com.sorin.mobilecmd.interceptors.MobileInterceptor"* init-method=*"init"*>

<beans:property name=*"mobileUserAgents"*>

<beans:list value-type=*"java.lang.String"*>

<beans:value>.\*(webos|palm|treo).\*</beans:value>

<beans:value>.\*(android).\*</beans:value>

<beans:value>.\*(kindle|pocket|o2).\*</beans:value>

<beans:value>.\*(vodaphone|wap|midp|psp).\*</beans:value>

<beans:value>.\*(iphone|ipod|ipad).\*</beans:value>

<beans:value>.\*(blackberry|opera mini).\*</beans:value>

</beans:list>

</beans:property>

</beans:bean>

</mvc:interceptor>

<mvc:interceptor>

<mvc:mapping path=*"/history.htm"* />

<beans:bean class=*"com.sorin.mobilecmd.interceptors.UserLoginInterceptor"*/>

</mvc:interceptor>

</mvc:interceptors>

</beans:beans>

Clasa *UserLoginInterceptor* validează existența unui user logat pe sesiune. În cazul acestei aplicații a fost necesară doar verificarea paginii *history.htm,* întrucât ea este singura care necesită un user.

**package** com.sorin.mobilecmd.interceptors;

**.....**

**public** **class** UserLoginInterceptor **extends** HandlerInterceptorAdapter **implements** HandlerInterceptor {

**private** **static** **final** Logger *log* = Logger.*getLogger*(UserLoginInterceptor.**class**);

@Override

**public** **boolean** preHandle(HttpServletRequest request, HttpServletResponse response,

Object handler) **throws** Exception {

User user = (User) WebUtils.*getSessionAttribute*(request, "user");

**if** (user == **null**) {

*log*.info("preHandle - Unauthorized access detected. Redirecting to login page.");

response.sendRedirect(request.getContextPath() + "/home.htm");

**return** **false**;

} **else** {

**return** **true**;

}

}

}

Un al doilea *Interceptor* este declarat pentru a filtra *reqest*-urile venite de la un browser de Mobil. În acest caz, dacă pe pagina de *home.htm* se încearcă accesarea de pe un dispozitiv mobil, dorim să prezentăm utilizatorului un alt mesaj și o altă pagină de întâmpinare. De exemplu, dorim să îi prezentăm un link către aplicația Android Google Play de unde poate achiziționa aplicația Client de Mobil.

Folosindu-ne de proprietățile definite în fișierul de configurare XML pentru acest interceptor, vom valida valoarea unui parametru din request pentru a verifica de unde a fost inițiat acest request.

<beans:list value-type=*"java.lang.String"*>

<beans:value>.\*(webos|palm|treo).\*</beans:value>

<beans:value>.\*(android).\*</beans:value>

<beans:value>.\*(kindle|pocket|o2).\*</beans:value>

<beans:value>.\*(vodaphone|wap|midp|psp).\*</beans:value>

<beans:value>.\*(iphone|ipod|ipad).\*</beans:value>

<beans:value>.\*(blackberry|opera mini).\*</beans:value>

</beans:list>

@Override

**public** **void** postHandle(HttpServletRequest request, HttpServletResponse response, Object handler, ModelAndView model) **throws** Exception {

String userAgent = request.getHeader("User-Agent");

*log*.debug("postHandle - the userAgent we intercepted was: " + userAgent);

**if** (ParameterParser.*getBooleanParameter*(request, "fromMobile"))

**return**;

**if** (userAgent != **null**) {

**if** (isMobile(userAgent)) {

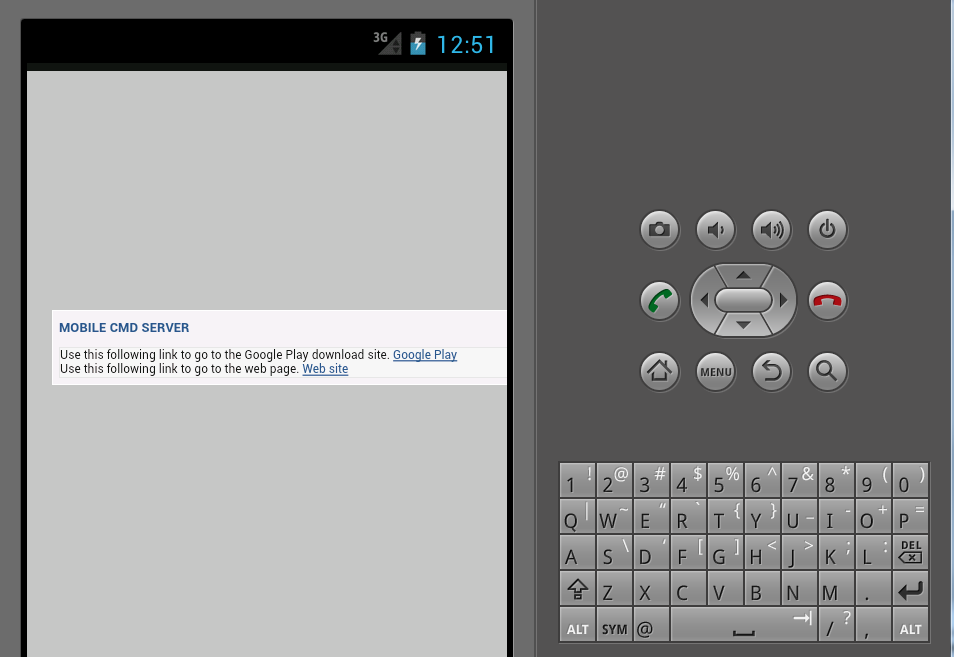
model.setViewName("redirect:mobile.htm");

}

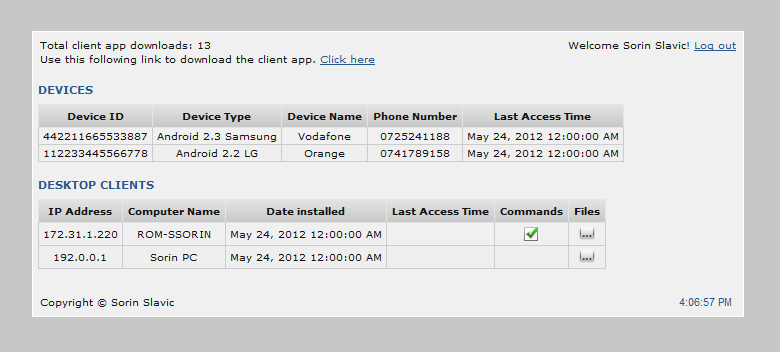
}

}

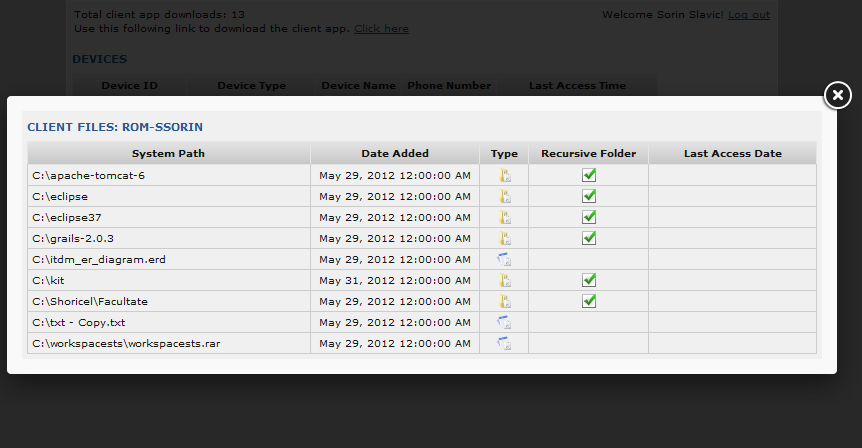
Validarea constă în a verifica dacă header-ul *UserAgent* de pe *request* conține una din valorile definite în fișierul de configurare. În acel caz redirecționăm acțiunea către Controller-ul asociat paginii *mobile.htm.* Utilizatorul are apoi opțiunea de a downloada aplicația de client de Mobil sau de a accesa pagina web normală, accesând al doilea link din pagină. Pe această cale va fi încârcată pagina *home.htm* de unde utilizatorul se poate loga sau înregistra la aplicație.

*Fig 5.2.4 Pagina de Welcome pentru un browser de Mobil*

După logarea în aplicația Web, utilizatorului îi va fi prezentată câte o listă pentru categoriile de aplicații Client Mobile și Desktop instalate. În baza de date, există pentru fiecare utilizator referințe pentru toate aplicațiile de Client Mobile utilizate dar și pentru cele de Client Desktop.

*Fig 5.2.5 Pagina history.htm – Lista aplicațiilor Mobile și Desktop*

În această pagină utlizatorul va putea observa informații despre fiecare aplicație de client utilizată, iar pentru aplicațiile de Desktop va putea deschide o fereastră în care îi sunt prezentate fișierele și folderele pe care le-a partajat și la care are acces de pe o aplicație de mobil.

*Fig 5.2.6 Lista fișierelor partajate pentru un anumit client Desktop*

Acest pop-up este realizat în Javascript, folosind librăriile Fancybox și JQuery. Pe apăsarea butonului corespunzător deschiderii acestei pagini (iconița de *Browse* de pe coloana Files) framework-ul Fancybox va deschide pagina într-o componentă internă și o va afișa pe ecran ca într-un pop-up, ascunzând pagina inițială.

<td align=*"center"*>

<a id=*"openFiles"* class=*"iframe"* href=*"clientFiles.htm?clientId=*${client.clientId}*"*>

<img src=*"images/popup.png"* width=*"16"* height=*"16"* alt=*"Open Files"* title=*"Open Files"* />

</a>

</td>

La încărcarea paginii *history.htm* (această pagină conține acțiunea care va deschide acest nou pop-up)*,* pe acțiunea JavaScript *document.onReady()*, am declarat clasa pentru care dorim să aplicăm acțiunile definite de Fancybox și am incluse fișierele *.js* necesare pentru rularea componentei.

<script type=*"text/javascript"* src=*"*<c:url value=*"/resources/fancybox/lib/jquery-1.4.min.js"*/>*"*>

</script>

<script type=*"text/javascript"* src=*"*<c:url value=*"/resources/fancybox/source/jquery.fancybox.pack.js"*/>*"* >

</script>

<link rel=*"stylesheet"* href=*"*<c:url value=*"/resources/fancybox/source/jquery.fancybox.css"*/>*"* type=*"text/css"* media=*"screen"* />

<link rel=*"stylesheet"* href=*"*<c:url value=*"/styles/styles.css"*/>*"* type=*"text/css"* />

<script type=*"text/javascript"*>

$(document).ready(**function**() {

$("#openFiles").fancybox({

'transitionIn' : 'elastic',

'transitionOut' : 'elastic',

'speedIn' : 300,

'speedOut' : 200,

'overlayShow' : **false**,

'width' : 800,

'height' : 300,

'autoScale' : **false**,

'type' : 'iframe'

});

});

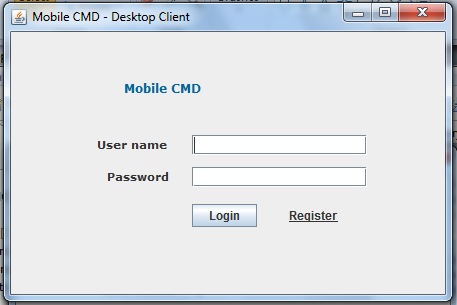
</script>

Cum am precizat, pentru această aplicație partea de server nu este doar pentru a prezenta în browser o interfață pentru utilizator, ci va fi și folosită de clienții Desktop și Mobil pentru a interacționa unii cu alții și pentru a accesa baza de date. Mesajele recepționate de la acești clienți vor fi de mai multe feluri: SOAP Xml, Obiecte Serializate Java, Request-uri Http. Fiecare va fi prezentată împreună cu clientul care face apelul și le folosește.

# 5.3 Aplicația de Client Desktop

De pe aplicația Web, un utilizator poate downloada un executabil. Această aplicație are rolul de Client Desktop. Prin ea, utilizatorul va alege fișierele și folderele accesibile de pe telefonul mobil și va preciza dacă permite rularea unor comenzi Windows de sistem.

Această aplicație, de fapt la fel ca și celelalte 2, este realizată în limbaj Java. Este o aplicație desktop bazată pe elemente Swing și are incorporat componente ale framework-ului Spring. Aceste componente Spring sunt folosite la comunicarea client-server.

 După downloadarea aplicației, la fiecare rulare, utilizatorul trebuie să introducă datele de logare. Prima fereastră este similară cu cea de pe aplicația Web, de pe site, având două butoane – unul de *Login* și altul de *Register.*

*Fig 5.3.1 Panoul de logare in aplicația Desktop*

Butonul de *Register* va deschide în browser-ul implicit al sistemului pagina de logare de pe aplicația Web.

btnRegister.addActionListener(**new** ActionListener() {

**public** **void** actionPerformed(ActionEvent e) {

**if** (Desktop.*isDesktopSupported*()) {

URI registration = DesktopCMDPropertiesFacade.*getRegistrationPath*();

**try** {

Desktop.*getDesktop*().browse(registration);

} **catch** (Exception ereg) {

JOptionPane.*showMessageDialog*(frmMobileCmd,

"Could not find a browser to open the page! " +

"Please go to: " + registration.toString());

ereg.printStackTrace();

}

}

}

});

Apăsarea butonului de *Login* va face un prim apel la aplicația Server pentru a loga utilizatorul.

btnLogin.addActionListener(**new** ActionListener() {

**public** **void** actionPerformed(ActionEvent e) {

**try** {

String usernamex = username.getText();

**char**[] passwordx = password.getPassword();

i**f** (usernamex.isEmpty())

**throw** **new** RequiredParameterException(lblUserName.getText());

**if** (passwordx.length == 0)

**throw** **new** RequiredParameterException(lblPassword.getText());

**if** (WSUtil.*login*(usernamex, passwordx)) {

JFrame frame = **new** FileSelect();

frame.setVisible(**true**);

frmMobileCmd.dispose();

}

......

După validarea datelor de intrare, vom face un apel la Serverul Web folosind metoda statică *Login* definită în clasa *WSUtil.*

**public** **static** **void** login(String username, **char**[] password) **throws** Exception {

HttpRequestObject loginRequest = **new** LoginRequest(username, password);

loginRequest.execute();

**if** (!loginRequest.isOK()) {

**throw** **new** Exception("Could not login to the account; Please try again");

}

User user = loginRequest.getResponseObject();

Session.*getSession*().setUser(user);

*registerComputer*();

*getClientFiles*();

}

Vom implementa un Obiect de tip *HttpRequestObject*. Această interfață permite conectarea la Serverul Web prin trimiterea unui mesaj Http. Pe serverul Web am implementat un Servlet care mapează cererile primite pe calea cu extensia *.cmd.* Trimițând astfel mesajele am realizat o conexiune Http cu serverul. Am folosit un Obiect *Factory* de la Spring pentru crearea efectivă a conexiunii Http.

**package** com.sorin.mobilecmd.desktopcmd.ws;

...

**public** **abstract** **class** HttpRequestObject {

**private** **static** **final** SimpleClientHttpRequestFactory *factory* =

**new** SimpleClientHttpRequestFactory();

**public** **final** String request;

**private** ClientHttpResponse res;

**public** HttpRequestObject(String request) {

**this**.request = request;

}

**public** **abstract** HttpMethod getMethod();

**public** **abstract** URI getRequestURI() **throws** Exception;

**public** **void** execute() **throws** IOException, Exception {

ClientHttpRequest req =

*factory*.createRequest(getRequestURI(), getMethod());

**this**.res = req.execute();

}

**public** InputStream getResponseStream() **throws** IOException {

**return** res.getBody();

}

**public** <T **extends** Serializable> T getResponseObject()

**throws** IOException, ClassNotFoundException {

ObjectInputStream ois = **new** ObjectInputStream(getResponseStream());

T t = (T) ois.readObject();

**return** t;

}

**public** **final** String getRequest() {

**return** DesktopCMDPropertiesFacade.*getServerPath*() +

request + DesktopCMDPropertiesFacade.*getServerExt*();

}

Cererea, *ClientHttpRequest*, este generată folosind calea către care trimitem *request*-ul și tipul cererii (POST, GET, etc). Aceste două metode sunt abstracte și trebuiesc implementate de obiectele care vor genera cererea. Serverul Web interpretează aceste cereri și va înapoia un răspuns cu valoarea Serializată a obiectelor returnate. Astfel, deserializând răspunsul obținem obiectul returnat efectiv de server.

Implementarea standard pentru un astfel de obiect trebuie să returneze cele două metode precizate anterior iar structura utilizării unui astfel de obiect trebuie să respecte următoarele:

HttpRequestObject request = **new** ? extends HttpRequestObject(...);

loginRequest.execute();

**if** (!request.isOK()) **throw** **new** Exception();

? extends Serializable response = request.getResponseObject();

Un exemplu de implementare pentru acestă structură este obiectul folosit la Login.

HttpRequestObject loginRequest = **new** LoginRequest(username, password);

**public** **class** LoginRequest **extends** HttpRequestObject {

**private** String username = "";

**private** String password = "";

**public** LoginRequest(String username, **char**[] password) {

**super**("login");

**this**.username = username;

**for** (**char** c : password)

**this**.password += c;

}

@Override

**public** HttpMethod getMethod() {

**return** HttpMethod.*POST*;

}

@Override

**public** URI getRequestURI() **throws** Exception {

String req = getRequest() + "?" +

"username=" + URLEncoder.*encode*(username, "UTF-8") +

"&password=" + URLEncoder.*encode*(password, "UTF-8");

**return** **new** URI(req);

}

}

*Request*-ul, calea către care trimitem cererea, va fi reprezentat de un Controller Spring în aplicația Web. Pentru acest exemplu, calea va fi:

serverPath / login.cmd ? username=...&password=...

Pe partea de Server, avem definit în *web.xml* maparea tuturor cererilor cu extensia *.cmd* către *appServlet.*

<servlet-mapping>

<servlet-name>appServlet</servlet-name>

<url-pattern>\*.cmd</url-pattern>

</servlet-mapping>

Acest *DispatcherServlet* era definit în fișierul de configurare *servlet-context.xml* unde era precizat că vrem instațierea *Controller*-ilor adnotați Java din pachetele:

<context:component-scan base-package=*"com.sorin.mobilecmd,*

*com.sorin.mobilecmd.desktop"* />

Sub pachetul *com.sorin.mobilecmd.desktop* avem definiți Controller-ii utilizați pentru mapările .cmd. După cum am precizat, pe *Request*-ul trimit către aceștia vom pune valorile obiectului care generează cererea, urmând ca *Response*-ul să conțină *Serializarea* obiectului returnat de această operație.

Mai jos avem metoda utilizată de maparea *login.cmd* a controller-ului. Această metodă este apelată în momentul în care clientul nostru *LoginRequest* din aplicația Desktop a trimis cerearea Http.

@RequestMapping(value="/login.cmd", method=RequestMethod.*POST*)

**public** HttpServletResponse login(HttpServletRequest request,

HttpServletResponse response) {

String username = ParameterParser.*getStringParameter*(

request, "username");

String password = ParameterParser.*getStringParameter*(

request, "password");

ObjectOutputStream oos = **null**;

*log*.debug("login - username: " + username + " / password: " + password);

**try** {

User user = userService.login(username, password);

**if** (user == **null**) {

response.setStatus(HttpStatus.*UNAUTHORIZED*.value());

} **else** {

response.setContentType("text/html; charset=utf-8");

response.setStatus(HttpStatus.*OK*.value());

oos = **new** ObjectOutputStream(response.getOutputStream());

oos.writeObject(user);

oos.flush();

}

} **catch** (IOException e) {

response.setStatus(HttpStatus.*BAD\_REQUEST*.value());

} **finally** {

**try** {

**if** (oos != **null**)

oos.close();

}

**catch** (IOException ex) {}

}

**return** **null**;

}

Folosind parametrii primiți pe request, vom realiza logarea în sistem a utilizatorului asociat. În cazul în care apare o eroare sau utilizatorul nu poate fi logat cu succes vom seta statusul răspunsului pe o valoare corespunzătoare. Acest Cod Status este verificat de clientul Desktop pentru a aflat dacă operația a fost realizată cu succes.

Folosind Serializarea obiectelor vom scrie efectiv pe răspuns acesta. Răspunsul este apoi automat returnat clientului Desktop care a inițiat cererea.

În aplicație am folosit această implementare a conexiunii cu serverul Web a clientului Desktop la trimiterea a 2 cereri. Fiecare cerere are un număr limitat sau predefinit de parametrii ce urmează să fie trimiși serverului sau de valori returnate. În cazul operației de Login trimitem 2 parametri (username și password) iar pentru operația de înregistrare a clientului trimitem 3 – UserID pentru a identifica utilizatorul care este logat, IpAddress, adresa fizică a calculatorului unde se află clientul și ComputerName reprezentând numele de sistem asociat calculatorului.

HttpRequestObject loginRequest = **new** LoginRequest(username, password);

HttpRequestObject registerComputerRequest = **new** RegisterComputerRequest(

InetAddress.*getLocalHost*().getHostAddress(), InetAddress.*getLocalHost*().getHostName());

Pentru apelurile mai complicate am decis să utilizez o altă componentă a framework-ului Spring care realizează serializarea și de-serializarea obiectelor automat. Practic a trebuit doar să definesc interfața care îmi prezintă metodele pe care doresc să le implementez și utilizez. Pentru aceasta a trebuie să îmi definesc un ApplicationContext pentru această aplicație desktop. Spre deosebire de aplicația Web în care am definit contextul direct din web.xml, aici trebuie să încărcăm componentele programatic.

În fișierul de configurare *applicationContext.xml* pentru această aplicație am definit un *HttpInvoker*, pentru care am precizat adresa web a aplicației și interfața serviciului a cărui metodă o invoc.

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<beans xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xmlns=*"http://www.springframework.org/schema/beans"*

xmlns:context=*"http://www.springframework.org/schema/context"*

xmlns:util=*"http://www.springframework.org/schema/util"*

xsi:schemaLocation="

ht[tp://www.springframework.org/schema/beans h](tp://www.springframework.org/schema/beans%20h)ttp://www.springframework.org/schema/beans/spring-beans-2.5.xsd

ht[tp://www.springframework.org/schema/context h](tp://www.springframework.org/schema/context%20h)ttp://www.springframework.org/schema/context/spring-context-2.5.xsd

<http://www.springframework.org/schema/util> *http://www.springframework.org/schema/util/spring-util-2.5.xsd*

*"*>

<context:property-placeholder location=*"desktopcmd.properties"*/>

<bean id=*"httpInvoker"* class=*"org.springframework.remoting.httpinvoker.HttpInvokerProxyFactoryBean"*>

<property name=*"serviceUrl"* value=*"${serviceURL}"*/>

<property name=*"serviceInterface"* value=*"com.sorin.mobilecmd.services.RemoteService"*/>

</bean>

</beans>

Această interfață este implementată pe server. Aplicația client nu face nimic altceva decât să apeleze o metodă din interfață iar arhitectura Spring realizează automat conexiunea cu Serverul, serializarea obiectelor de intrare și ieșire și executarea metodei efective.

**private** **static** **final** ApplicationContext *applicationContext* =

**new** ClassPathXmlApplicationContext("applicationContext.xml");

**private** **static** **final** RemoteService *remoteService* =

(RemoteService) *applicationContext*.getBean("httpInvoker");

Am definit contextul aplicației, folosind fișierul de configurare menționat, și am instanțiat un *Bean* definit în acest fișier.

Pentru această interfață definim doar două metode care sunt utilizate de clientul Desktop.

**public** **interface** RemoteService {

/\*\*

\* **@param** clientId

\* **@return** the list of Files for this client computer

\*/

**public** List<ClientFile> getClientFiles(**int** clientId);

/\*\*

\* **@param** clientId

\* **@param** allowCmd

\* **@param** files

\* **@return** true if the operation was successful / false otherwise

\*/

**public** **boolean** saveConfiguration(**int** clientId,

**boolean** allowCmd, List<ClientFile> files);

}

Implemetarea acestor metode este realizată în mod normal pe partea de Server, implementarea neutilizând direct niciun element de Serializare sau Deserializare Java, sau alte caracteristici ale acestui tip de Request Http. Implementarea serviciului de remote se realizează dintr-un fișier de configurare.

@Service

**public** **class** RemoteServiceImpl **implements** RemoteService {

**private** **final** JdbcTemplate jdbc;

@Autowired

**public** RemoteServiceImpl(JdbcTemplate jdbc) {

**this**.jdbc = jdbc;

}

@Override

**public** List<ClientFile> getClientFiles(**int** clientId) {

String sql = "select ClientFileID, ClientID, Path, IsFolder,

IsRecursive, DateAdded, LastAccessDate " +

" from ClientFiles " +

" where ClientID = ? " +

" order by Path ";

**return** jdbc.query(sql, **new** ClientFileRowMapper(), clientId);

}

@Transactional

@Override

**public** **boolean** saveConfiguration(**int** clientId, **boolean** allowCmd,

List<ClientFile> files) {

**try** {

**int** allowCommands = (allowCmd) ? 1 : 0;

Object[] updateParams = {allowCommands, clientId};

jdbc.update("update Clients set AllowCommands = ?

where ClientID = ?", updateParams);

jdbc.update("delete from ClientFiles where ClientID = ?", clientId);

**for** (ClientFile file : files) {

Object[] insertParams = {clientId, file.getPath(),

file.isFolder() ? 1 : 0, file.isRecursive() ? 1 : 0, file.getDateAdded(), file.getLastAccessDate()};

jdbc.update("insert into ClientFiles (ClientID, Path, IsFolder,

IsRecursive, DateAdded, LastAccessDate) " +

" values (?, ?, ?, ?, ?, ?) ", insertParams);

}

} **catch** (Exception e) {

**return** **false**;

}

**return** **true**;

}

}

Clientul de Desktop și-a creat obiectul *HttpInvoker* precizând adresa la care să se trimită cererea și interfața folosită. În cazul acesta path-ul pentru request este definit ca:

serviceURL = {serverAddress}/mobilecmd/remote.service

În aplicația de Server am configurat un servlet în *web.xml* care recepționează aceste cereri. Servlet-ul este configurat pe arhitectura de *DispatcherServlet* de la Spring și este configurat folosind fișierul *remote-servlet-context.xml*

<servlet>

<servlet-name>remoting</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet

</servlet-class>

<init-param>

<param-name>contextConfigLocation</param-name>

<param-value>/WEB-INF/spring/remote/remote-servlet-context.xml

</param-value>

</init-param>

</servlet>

<!-- Java Service -->

<servlet-mapping>

<servlet-name>remoting</servlet-name>

<url-pattern>\*.service</url-pattern>

</servlet-mapping>

În acest fișier de configurare vom instanția în *ApplicationContext* serviciul utilizat. Pentru acest serviciu folosin ca parametru *JdbcTemplate*-ul deja definit în ApplicationContext. Serviciul este utilizat și de un alt bean definit în acest fișier de configurare. Acest bean reprezintă obiectul care efectuează și intrepretează cererile recepționate pentru acest Servlet Remote. Tot el va realiza și răspunsul valorilor returnate de metodele serviciului.

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<beans xmlns=*"http://www.springframework.org/schema/beans"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xmlns:context=*"http://www.springframework.org/schema/context"*

xmlns:tx=*"http://www.springframework.org/schema/tx"*

xmlns:p=*"http://www.springframework.org/schema/p"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans-3.0.xsd*

[*http://www.springframework.org/schema/context*](http://www.springframework.org/schema/context)

*http://www.springframework.org/schema/context/spring-context-3.0.xsd*

[*http://www.springframework.org/schema/tx*](http://www.springframework.org/schema/tx)

*http://www.springframework.org/schema/tx/spring-tx-3.0.xsd"*>

<bean id=*"remotingService"*

class=*"com.sorin.mobilecmd.services.RemoteServiceImpl"*>

<constructor-arg ref=*"jdbcTemplate"*/>

</bean>

<bean id=*"httpInvoker"*

class=*"org.springframework.remoting.httpinvoker.HttpInvokerServiceExporter"*>

<property name=*"service"*>

<ref bean=*"remotingService"*/>

</property>

<property name=*"serviceInterface"*>

<value>com.sorin.mobilecmd.services.RemoteService</value>

</property>

</bean>

<bean id=*"urlMapping"*

class=*"org.springframework.web.servlet.handler.SimpleUrlHandlerMapping"*>

<property name=*"mappings"*>

<props>

<prop key=*"/remote.service"*>httpInvoker</prop>

</props>

</property>

</bean>

</beans>

După logarea în aplicație, o nouă fereastră se va deschide. Aceasta va conține o listă cu fișierele și folderele partajate de utilizatorul logat pentru acest calculator. Popularea acestei liste se realizează utilizând serviciul și metoda menționată anterior. Utilizarea acestui framework simplifică foarte mult structura codului și a metodei. Practic se realizează doar apelarea metodei dorite din interfață.

Pentru această aplicație am utitlizat un obiect singleton *Session.getSession()* care incorporează un *Map<String, Object>* în care vom reține toate datele necesare pentru această rulare a aplicației – user-ul logat, structura fișierelor, etc.

**public** **static** **void** getClientFiles() **throws** Exception {

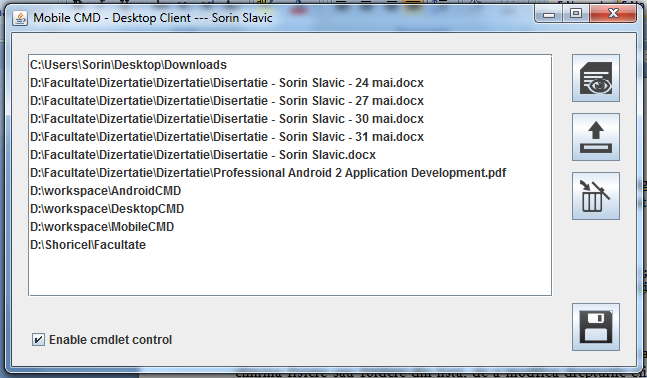
**int** clientId = Session.*getSession*().getClientId();

List<ClientFile> files = *remoteService*.getClientFiles(clientId);

Session.*getSession*().setFiles(files);

}

Din această nouă fereastră utilizatorul are opțiunea de a adăuga fișiere sau foldere, de a elimina fișiere sau foldere din listă, de a modfica drepturile clienților de a rula comenzi sistem sau de a deschide un anumit fișier sau folder selectat.

*Fig 5.3.2 Fereastra de configurare a clientului Desktop*

După modificarea configurației curente a clientului de Desktop, utilizatorul trebuie să salveze această nouă listă cu fișiere. Pentru aceasta se va apela o altă metodă definită în serviciu, care va fi executată la fel ca cea prezentată anterior, pe server.

**Public** **static** **boolean** saveConfiguration(**boolean** allowCmd,

List<ClientFile> files) **throws** Exception {

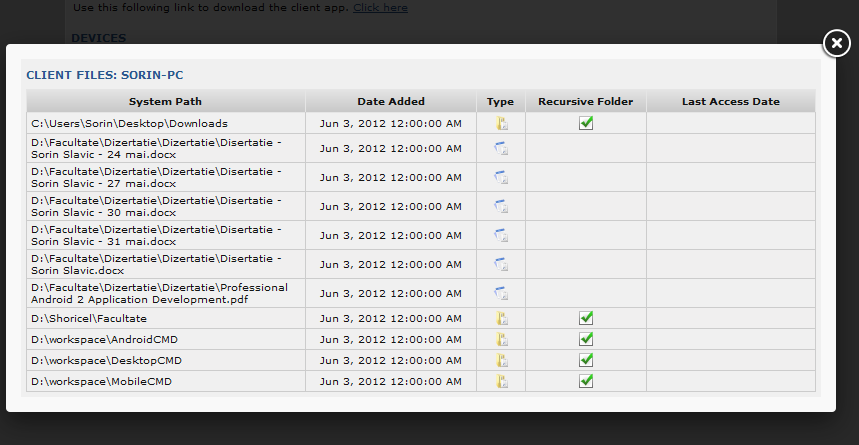
**int** clientId = Session.*getSession*().getClientId();

**boolean** response = *remoteService*.saveConfiguration(clientId, allowCmd, files);

**return** response;

}

Noua configurație poate fi observată de acest utilizator și din aplicația Web, accesând fișierele corespunzătoare calculatorului pe care tocmai le-a setat.

*Fig 5.3.3 Lista fișierelor și folderelor partajate*

Întrucât fișierele partajate nu sunt copiate pe server și în plus avem opțiunea de a rula comenzi de sistem pe acest calculator, pentru a se putea realiza conexiunea dintre clientul de Android și acesta, trebuie ca aplicația Desktop să fie pornită. Vom preciza mai multe despre cum se realizează conexiunea dintre aplicația Android și acest client în capitolul următor când descriem aplicația de Mobile.

# 5.4 Aplicația de Client Android

Aplicația de client Android permite unui utilizator conectarea la un calculator și downloadarea fișierelor sau rularea de comenzi sistem. Pentru aceasta trebuie să realizăm o conexiune cu acea unitate, folosindu-ne și de serverul web. Primul pas în conectarea cu sistemul este logarea la serverul Web prin trimiterea și recepționarea de mesaje SOAP XML.

La pornirea aplicației, utilizatorul trebuie să se identifice și să se logheze în sistem. În funcție de utilizatorul logat se va popula lista cu calculatoare disponibile. Pe sistemul Android, aplicația este dezvoltată astfel încât să ruleze un thread pentru interfața grafică și un thread care să ruleze efectiv comenzile și să comunice cu serverul Web. Vom prezenta un exemplu pentru această configurare în cele ce urmează.

Prima fereastră accesată de utilizator este cea de Login. Similar celorlalte două cazuri (logarea pe Serverul Web și logarea în aplicația Desktop), utilizatorul trebuie să își introducă contul și parola. Există un buton de Register care, similar aplicației Desktop, va deschide în browser pagina de înregistrare a aplicației web.

Pe apăsarea butonului de Register se va declara un *Intent* care să deschidă pagina respectivă. Structura sistemului Android permite activarea unor componente declarate de alte aplicații prin rularea acestor *Intent*-uri. În mod automat sistemul va detecta o aplicație care este capabilă să efectueze cererea noastră și o va folosi în mod corespunzător.

register.setOnClickListener(**new** View.OnClickListener() {

@Override

**public** **void** onClick(View v) {

Log.*d*(*TAG*, "click on button register");

Uri uri = AndroidCMDProperties.*getRegistrationUri*();

Log.*d*(*TAG*, "the path is: " + uri.toString());

Intent browserIntent = **new** Intent(Intent.*ACTION\_VIEW*, uri);

startActivity(browserIntent);

}

});

În fișierul *AndroidManifest.xml* trebuie declarate toate activitățile și toate proprietățile pe care aplicația noastră le utilizează. Astfel, înainte ca o activitate să poată fi începută sau activată, trebuie ca ea să fie definită în acest fișier.

<application

android:icon=*"@drawable/icon"*

android:label=*"@string/app\_name"*

android:name=*"AppObject"*>

<activity android:name=*"AndroidCMDActivity"*

android:label=*"@string/app\_name"* >

<intent-filter>

<action android:name=*"android.intent.action.MAIN"* />

<category android:name=*"android.intent.category.LAUNCHER"* />

</intent-filter>

</activity>

<activity android:name=*"ErrorActivity"*

android:theme=*"@android:style/Theme.Dialog"*>

<intent-filter>

<action android:name=*"android.intent.action.MAIN"* />

</intent-filter>

</activity>

<activity android:name=*"ComputerListActivity"*

android:label=*"@string/app\_name"*>

<intent-filter>

<action android:name=*"android.intent.action.MAIN"* />

</intent-filter>

</activity>

..........

Apăsarea butonului de *Login* va iniția acțiunea de conectare la serverul Web. Conectarea se va realiza pe un *Thread* separat, *Thread*-ul dedicat interfeței grafice rulând o fereastră care semnalează încărcarea.

Aplicația este construită astfel încât activitățile care permit conectarea cu Serverul Web extind o clasă generică, *ConnectionActivity*. Această clasă este cea care realizează comunicarea cu serverul web, ea având și rolul de a afișa pe ecran fereastra de loading când această conectare cu serverul are loc.

**public** **class** AndroidCMDActivity **extends** ConnectionActivity {

**private** **static** **final** String *TAG* = AndroidCMDActivity.**class**.getSimpleName();

EditText usernameBox;

EditText passwordBox;

@Override

**public** **void** onCreate(Bundle savedInstanceState) {

**super**.onCreate(savedInstanceState);

Log.*d*(*TAG*, "Creating the main activity - login");

setContentView(R.layout.*login*);

usernameBox = (EditText) findViewById(R.id.*username*);

passwordBox = (EditText) findViewById(R.id.*password*);

Button login = (Button) findViewById(R.id.*loginBtn*);

login.setOnClickListener(**new** View.OnClickListener() {

**public** **void** onClick(View v) {

Log.*d*(*TAG*, "click on button login");

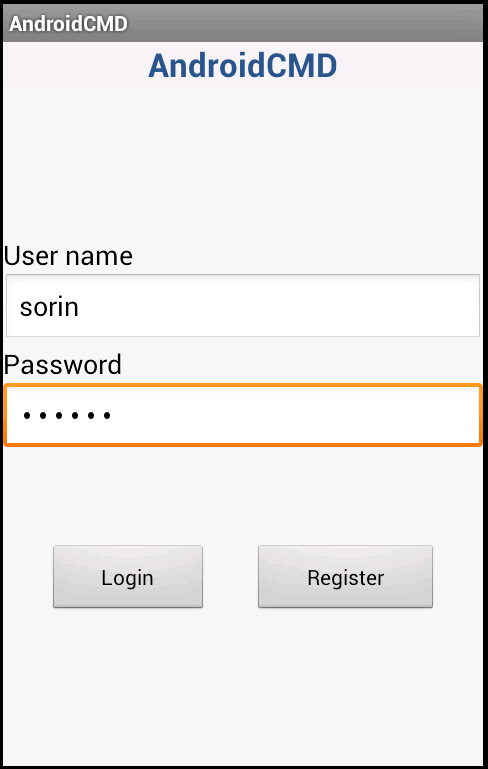
**if** (validate())

post();

}

});

..........

*Fig 5.4.1 View-ul de login în aplicație*

Superclasa *ConnectionActivity* este construită astfel încât să implementeze o interfață *SoapCallback*. Această interfață are rolul definește două metode standard care sunt returnate în urma unui apel SOAP la Web Server – o metodă de succes și una de eroare:

**public** **interface** SoapCallback {

**public** **void** ready(List<? **extends** SoapResponse> response);

**public** **void** error(String message, Exception e);

}

Implementarea pentru metoda de succes, *ready()*, asigură sincronizarea dintre thread-ul de GUI și cel responsabil de conexiunea SOAP și trimite mai departe executarea către o altă metodă abstractă, ce va fi implementată de Activitățile care extind această clasă.

**public** **abstract** **void** readyToUse(List<? **extends** SoapResponse> response);

@Override

**public** **final** **void** ready(List<? **extends** SoapResponse> response) {

**try** {

readyToUse(response);

} **finally** {

hideLoading();

**synchronized**(**this**) {

inProgress = **false**;

}

}

}

Metoda de eroare este utilizată în situația în care a apărul orice excepție în execuția apelului SOAP către server sau dacă răspunsul returnat este de tip *Fault* (eroarea a avut loc pe partea de server).

Pentru mesajul SOAP utilizăm API-ul Ksoap2, specific aplicațiilor de mobil. Întrucât versiunea de JRE pe care este bazată mașina virtuală Dalvik nu conține toate pachetele de bază din Java 1.6, nu am avut acces la pachetele javax.xml.bind sau javax.xml.soap, acestea fiind excluse din proiect. Clientul care realizează apelul la serviciul SOAP este definit ca o clasă generică pentru care se știe clasa obiectului de intrare și clasa obiectului de ieșire.

**public** **class** GenericSoapClient<Req **extends** SoapRequest, Res **extends** SoapResponse> {

Interfețele *SoapRequest* și *SoapResponse* declară cât o metodă utilizată pentru a transforma obiectul nostru, fie el User, Client, ClientFile, în și dintr-un obiect org.ksoap2.serialization.*SoapObject* utilizat de API-ul Ksoap2 în construirea mesajelor XML SOAP.

Serverul Web pune la dispoziția clienților un fișier *WSDL* în care se descrie structura mesajelor XML SOAP utilizate. De exemplu, după logarea cu succes în aplicație, adică trimiterea unui mesaj SOAP conținând numele de utilizator și parola și recepționarea unui alt mesaj SOAP conținând valori pentru a defini un obiect de tip *User* (*UserID*, *Username*, *FirstName*, *LastName*) urmează încărcarea listei ce va conține toți clienții Desktop la care utilizatorul logat are acces. Pentru aceasta, structura mesajului utilizat pentru cerere trebuie să identifice utilizatorul, deci, trebuie să conțină cel mai ușor UserID-ul definit în aplicație pentru el.

<element name=*"GetClientsRequest"*>

<complexType>

<all>

<element maxOccurs=*"1"* minOccurs=*"1"* name=*"UserID"* type=*"int"*/>

</all>

</complexType>

</element>

<complexType name=*"Client"*>

<all>

<element maxOccurs=*"1"* minOccurs=*"1"* name=*"ClientID"* type=*"int"*/>

<element maxOccurs=*"1"* minOccurs=*"1"* name=*"ComputerName"* type=*"string"*/>

<element maxOccurs=*"1"* minOccurs=*"1"* name=*"IpAddress"* type=*"string"*/>

<element maxOccurs=*"1"* minOccurs=*"1"* name=*"AllowCommands"* type=*"boolean"*/>

</all>

</complexType>

<element name=*"GetClientsResponse"*>

<complexType>

<sequence>

<element maxOccurs=*"unbounded"* minOccurs=*"0"* name=*"Clients"* type=*"tns:Client"*/>

</sequence>

</complexType>

</element>

Răspunsul cererii va fi o listă de elemente Client. Acestea vor fi parsate în clientul nostru care a apelat serviciul web, și utilizând implementarea metodei definite de interfața *SoapResponse* se va creea un obiect Java corespunzător.

**public** **interface** SoapRequest {

**void** convertToSoapObject(SoapObject soapObject);

}

**public** **interface** SoapResponse {

**void** convertFromSoapObject(SoapObject soapObject);

}

Pentru exemplul anterior, în aplicație utilizând obiecte ale clasei ClientWS. Implementarea metodelor va modifica structura obiectelor *SoapObject* astfel încât să corespundă cu valorile membriilor clasei noastre.

**public** **class** ClientWS **implements** NameValue, SoapResponse, SoapRequest {

**private** **int** clientId;

**private** String ipAddress;

**private** String computerName;

**private** **boolean** allowCommands;

**private** **int** userId;

**public** ClientWS() {

}

@Override

**public** **void** convertToSoapObject(SoapObject soapObject) {

SoapUtil.*addProperty*(soapObject, "UserID", getUserId());

}

@Override

**public** **void** convertFromSoapObject(SoapObject soapObject) {

**this**.setClientId(Integer.*parseInt*(

soapObject.getPropertyAsString("ClientID")));

**this**.setComputerName(soapObject.getPropertyAsString("ComputerName"));

**this**.setIpAddress(soapObject.getPropertyAsString("IpAddress"));

**this**.setAllowCommands(soapObject.getPropertyAsString("AllowCommands").

equalsIgnoreCase("true"));

}

Atât această fereastră, cât și următoarea care poate fi accesată de utilizator are un *layout* care va prezenta lista de obiecte returnată de apelurile la servciul web. Utilizatorul trebuie să selecteze din această listă inițială de clienți calculatorul la care vrea să se conecteze. Se va trimite o nouă cerere SOAP în urma căreia se va returna lista cu Fișiere și Foldere partajate pe unitatea respectivă. Structura mesajelor SOAP pentru această nouă cerere este foarte similară celei precedente. În Request-ul SOAP se specifică ID-ul Clientului la care vrem să ne conectăm (ClientID) și ni se va returna o listă de obiecte de tip ClientFiles (ClientFileID, FilePath, IsFolder).

De pe această nouă fereastră utilizatorul își poate select un fișier pe care vrea să îl downloadeze sau un folder pe care vrea să îl extindă. Utilizatorul, dacă este activat de pe clientul Desktop, are drepturi să ruleze Cmdlet-i de sistem (script-uri powershell sau cmd). Introducând scriptul dorit acesta va fi redirecționat de Serverul Web pe unitatea Desktop unde va și rula, returnând-use răspunsul obținut de la sistem.

Mesajele SOAP trimise de aplicația Android sunt direcționate către un servlet al Serverului Web. Folosind arhitectura Spring am definit în web.xml un Servlet dedicat exclusiv request-urilor pentru aceste servicii SOAP.

<!-- Defines the Spring-WS MessageDispatcherServlet -->

<servlet>

<servlet-name>web-service</servlet-name>

<servlet-class>

org.springframework.ws.transport.http.MessageDispatcherServlet

</servlet-class>

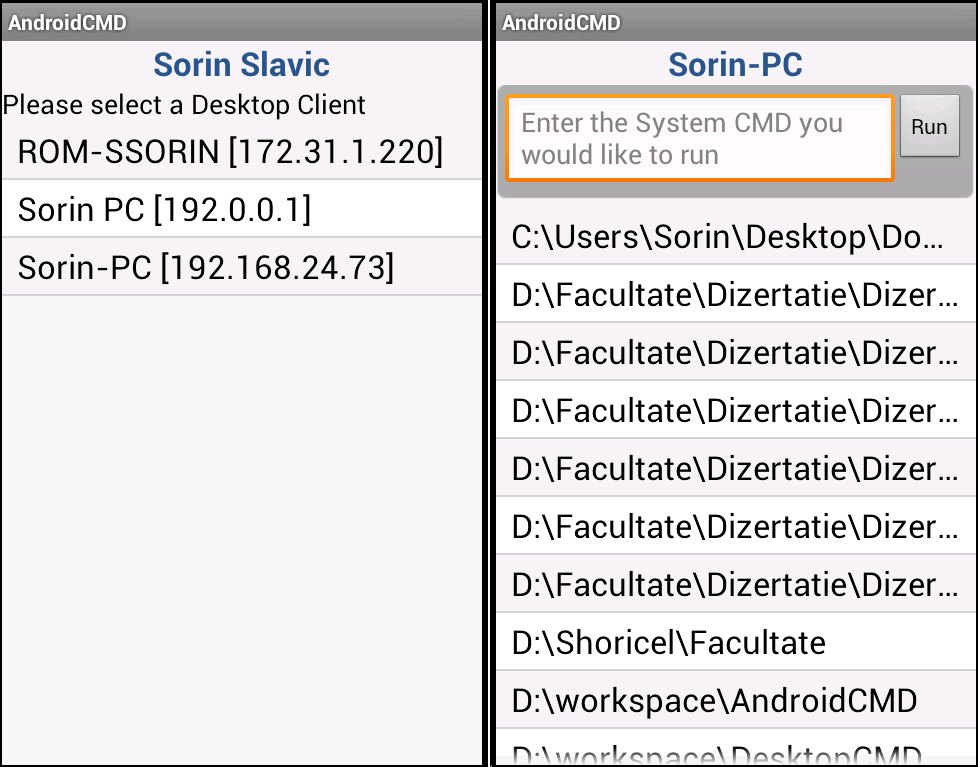
<init-param>

<param-name>contextConfigLocation</param-name>

<param-value>/WEB-INF/spring/ws/web-service-servlet.xml</param-value>

</init-param>

</servlet>

*Fig 5.4.2 View-urile cu listele de Cienți și Fișiere obținute pentru un client*

La fel ca la celelalte componente Spring utilizate definim un *context* pentru acest Servlet în care ne instanțiem bean-urile necesare, creând astfel și dependițele de ace avem nevoie. Astfel realizăm dintr-un singur fișier de configurare mapările pentru toate Endpoint-urile, asignăm un Marshall care știe să parseze și să prelucreze automat mesajele SOAP recepționate și ne instanțiem serviciul care dorim să realizeze toate acțiunile și apelurile la baza de date (sau la clientul desktop).

Precizez că în continuare nu sunt prezentate toate elementele definite în acest fișier de configurare. Întrucât are un volum mai mare ca celelalte prezentate la capitolele anterioare, voi face prezentarea pe un singur exemplu.

<bean id=*"marshaller"* class=*"org.springframework.oxm.jaxb.Jaxb2Marshaller"*>

<property name=*"classesToBeBound"*>

<list>

<value>com.sorin.mobilecmd.xml.schema.RunCommand</value>

<value>com.sorin.mobilecmd.xml.schema.RunResponse</value>

</list>

</property>

</bean>

<bean class=*"org.springframework.ws.server.endpoint.mapping.*

*PayloadRootQNameEndpointMapping"*>

<property name=*"mappings"*>

<props>

<prop key=*"{http://www.sorin.com/cmd}RunCommand"*>

runCommandEndpoint

</prop>

</props>

</property>

......

<bean id=*"service"* class=*"com.sorin.mobilecmd.services.WebServiceImpl"*>

<constructor-arg ref=*"jdbcTemplate"*/>

</bean>

<bean id=*"runCommandEndpoint"* class=*"com.sorin.mobilecmd.ws.RunCommandEndpoint"*>

<constructor-arg ref=*"marshaller"* />

<constructor-arg ref=*"service"* />

</bean>

Obervăm că instațiem un Marshaller pentru cele două clase care sunt direct utilizate de mesajul SOAP ce identifică o cerere RunCommand. Acest marshaller este utilizat în instanțierea endpoint-ului și va realiza automat parsarea mesajelor în obiecte Java. Implementarea serviciului utilizat, deși pentru acest exemplu nu are nevoie de acces la baza de date, este definit având ca dependință obiectul definit în *root-context.xml* pentru accesarea bazei.

Structura clasei *RunCommandEndpoint* este foarte simplă, întrucât parsarea mesajul sau verificarea corectă a excepțiilor apărute sunt realizate de infrastructura Spring, fie prin bean-urile definite de noi, fie prin obiectele interne definite în framework.

**public** **class** RunCommandEndpoint **extends** BaseEndpoint {

/\*\*

\* Creates a new ResponseEndpoint object - web-service-servlet.xml bean;

\* **@param** marshaller

\* **@param** exShellInvoker

\* **@throws** IOException

\*/

**public** RunCommandEndpoint(Marshaller marshaller, WebService service)

**throws** IOException {

**super**(marshaller, service);

}

/\*\*

\* The SOAPMessage Request is marshalled into an Object: requestObject.

\* If there are no exceptions we go ahead and invoke the desired service.

\* **@param** requestObject the marshalled request

\* **@return** an object <code>response</code>, as from the service method;

\*/

**protected** Object invokeInternal(Object requestObject) **throws** Exception {

RunCommand request = (RunCommand) requestObject;

RunResponse response = service.runCommand(request);

**return** response;

}

}

*Service* este implementarea serviciului pe care am definit-o în *context*. Metoda *invokeInternal* este apelată de aritectura internă Spring după ce a fost parsat mesajul Soap primit. Obiectul returnat de această metodă, Spring știe că trebuie să îl parseze într-un fișier XML și să îl trimită ca răspuns SOAP clientului.

Am ales acest exemplu pentru a prezenta partea de Server a serviciul web SOAP întrucât e un pic mai complicat decât alte metode care realizează doar un apel la baza de date. În acest caz, deoarece dorim să rulăm o comandă sistem pe client, trebuie să trimitem această cerere mai departe clientului Desktop. Pentru această conexiune vom utiliza Socket-ii Java.

Pentru a realiza cele dorite de noi, Clientul Desktop va juca rol de Server iar Serverul Web va juca rolul clientului care trimite o cerere. La pornirea aplicației client, vom rula pe un Thread separat o clasă simplă, care definește un *ServerSocket* și așteaptă conectarea clienților la el. Fiecare client care se conectează la acest *ServerSocket* va rula pe un alt thread separat.

(**new** Thread(**new** WaitForServer())).start();

Aceasta este declarată în funcția *main()* a aplicației client, imediat după rularea thread-ului pentru interfața grafică. Clasa *WaitForServer* are o structură foarte simplă:

ServerSocket listener = **new** ServerSocket(44100);

Socket socket = **null**;

*log*.debug("run - listening to: " + listener);

**while**(**true**) {

*log*.debug("run - waiting for client ...");

socket = listener.accept();

*log*.debug("run - new client !!! " + socket);

Thread t = **new** Thread(**new** RunCommandThread(socket));

t.start();

*log*.debug("run - started new thread ... wait?");

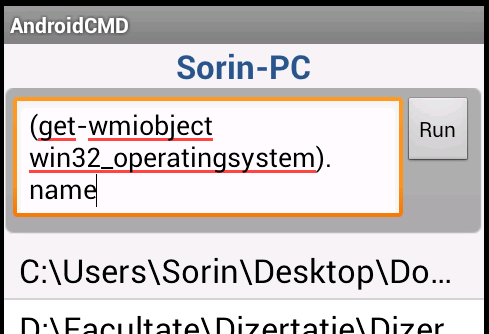
}

Fiecare cerere care se ajunge la aplicația Desktop va fi executată pe un thread separat. Acest nou thread preia rolul de a returna răspunsul obținut în urma rulării comenzii clientului, în acest caz Serverului Web. Serverul web va prelucra răspunsul și îl va returna la rândul său clientului Android sub forma unui răspuns SOAP.

Pentru a exemplifica mai clar ordinea efectuării operațiilor vă voi prezenta o cerere trimisă de pe telefonul mobil, pentru care primit un răspuns. Întrucât comenzilor pot fi rulate pe aplicația PowerShell, inclusă în Windows, trebuie să știm structura și comenzile pe care vrem să le rulăm. Rulat direct din linia de comandă de pe calculator am obținut următorul răspuns:

PS C:\Users\Sorin> (Get-WmiObject Win32\_OperatingSystem).Name

Microsoft Windows 7 Professional |C:\Windows|\Device\Harddisk0\Partition1

Inițiem acceași comandă utilizând aplicația Mobil.

*Fig 5.4.3 Executarea unui Cmdlet*

Primul pas pentru executare ecestei comenzi este construirea unui mesaj SOAP care să fie trimis la Server. Mesajul generat de aplicația Client Android este următorul:

<v:Envelope xmlns:i="http://www.w3.org/2001/XMLSchema-instance"

xmlns:d="http://www.w3.org/2001/XMLSchema"

xmlns:c="http://schemas.xmlsoap.org/soap/encoding/"

xmlns:v="http://schemas.xmlsoap.org/soap/envelope/">

<v:Header />

<v:Body>

<n0:RunCommand id="o0" c:root="1" xmlns:n0="http://www.sorin.com/cmd">

<n0:Command i:type="d:string">

(get-wmiobject win32\_operatingsystem).name

</n0:Command>

<n0:IpAddress i:type="d:string">

192.168.24.73

</n0:IpAddress>

</n0:RunCommand>

</v:Body>

</v:Envelope>

Același mesaj este recepționat și de aplicație Server Web.

DEBUG SoapEnvelopeLoggingInterceptor#Request:

<v:Envelope xmlns:v="http://schemas.xmlsoap.org/soap/envelope/"

xmlns:c="http://schemas.xmlsoap.org/soap/encoding/"

xmlns:d="http://www.w3.org/2001/XMLSchema"

xmlns:i="http://www.w3.org/2001/XMLSchema-instance">

<v:Header/>

<v:Body>

<n0:RunCommand xmlns:n0="http://www.sorin.com/cmd" c:root="1" id="o0">

<n0:Command i:type="d:string">

(get-wmiobject win32\_operatingsystem).name

</n0:Command>

<n0:IpAddress i:type="d:string">

192.168.24.73</n0:IpAddress>

</n0:RunCommand>

</v:Body>

</v:Envelope>

Această cerere este transformată automat în obiectul Java corespunzător (*RunCommand*) și este folosit la apelarea clientului Desktop.

2012-06-09 02:36:14,284 - DEBUG WebServiceImpl#runCommand - conect to client: 192.168.24.73

2012-06-09 02:36:14,287 - DEBUG WebServiceImpl#runCommand - connected: Socket[addr=192.168.24.73,port=44100,localport=8614]

2012-06-09 02:36:14,287 - DEBUG WebServiceImpl#runCommand - writer is ok! java.io.DataOutputStream@15660c3

2012-06-09 02:36:14,287 - DEBUG WebServiceImpl#runCommand - we will write: (get-wmiobject win32\_operatingsystem).name

2012-06-09 02:36:14,287 - DEBUG WebServiceImpl#runCommand – flushed

Clientul Desktop așteaptă o conexiune iar apoi bazat pe comanda primită o va rula.

2012-06-09 02:36:14,286 - DEBUG RunCommandThread#run - connection with client: Socket[addr=/127.0.0.1,port=8614,localport=44100]

2012-06-09 02:36:14,287 - DEBUG RunCommandThread#run - solved both streams! - connection is OK

2012-06-09 02:36:14,287 - DEBUG RunCommandThread#run - received command is: (get-wmiobject win32\_operatingsystem).name

2012-06-09 02:36:14,287 - DEBUG RunCommandThread#run - running command!

2012-06-09 02:36:14,287 - DEBUG RuntimeResponse#run - run command : (get-wmiobject win32\_operatingsystem).name

2012-06-09 02:36:14,287 - DEBUG RuntimeResponse#run - Full command : [powershell.exe, -NonInteractive, -NoLogo, -Command, (get-wmiobject win32\_operatingsystem).name]

2012-06-09 02:36:14,295 - DEBUG RuntimeResponse#run - before exit value

2012-06-09 02:36:16,538 - DEBUG ReadStreamThread#run - the message is: Microsoft Windows 7 Professional |C:\Windows|\Device\Harddisk0\Partition1

După rularea comenzii, fără a prelucra răspunsul (indiferent dacă a fost citit de pe input-ul de Erroare sau cel de Răspuns) îl va trimite înapoi la Server (împreună cu exitCode-ul procesului).

2012-06-09 02:36:16,547 - DEBUG RunCommandThread#run - command response ! exitCode: 0

2012-06-09 02:36:16,547 - DEBUG RunCommandThread#run - command response ! response: Microsoft Windows 7 Professional |C:\Windows|\Device\Harddisk0\Partition1

2012-06-09 02:36:16,547 - DEBUG RunCommandThread#run - sent message back with outputStream

Aplicația de Server Web în acest moment a primit răspunsul înapoi și îl va reîmpacheta într-un obiect Java, trimițându-l apoi ca mesaj SOAP înapoi la Clientul Android.

2012-06-09 02:36:16,547 - DEBUG WebServiceImpl#runCommand - the response message is: 0;Microsoft Windows 7 Professional |C:\Windows|\Device\Harddisk0\Partition1

2012-06-09 02:36:16,548 - DEBUG RunCommandEndpoint#Marshalling [com.sorin.mobilecmd.xml.schema.RunResponse@1ead007] to response payload

2012-06-09 02:36:16,550 - INFO BaseEndpoint#onMarshalResponse - Sending response: <SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">

<SOAP-ENV:Header/>

<SOAP-ENV:Body>

<ns2:RunResponse xmlns:ns2="http://www.sorin.com/cmd">

<ns2:Success>true</ns2:Success>

<ns2:Message>

Microsoft Windows 7 Professional | C:\Windows|\Device\Harddisk0\Partition1&#13;

</ns2:Message>

</ns2:RunResponse>

</SOAP-ENV:Body>

</SOAP-ENV:Envelope>

În acest moment răspunsul ajunge înapoi și la clientul de Android care a inițiat acest proces. La rândul său acesta va parsa mesajul SOAP, obținând un obiect Java, valorile căruia le va și afișa pe ecran.

06-08 23:36:16.467: I/GenericSoapClient(669): before getResponse

06-08 23:36:16.467: I/GenericSoapClient(669): the response message was:

<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">

<SOAP-ENV:Header/>

<SOAP-ENV:Body>

<ns2:RunResponse xmlns:ns2="http://www.sorin.com/cmd">

<ns2:Success>true</ns2:Success>

<ns2:Message>

Microsoft Windows 7 Professional |C:\Windows|\Device\Harddisk0\Partition1&#13;

</ns2:Message>

</ns2:RunResponse>

</SOAP-ENV:Body>

</SOAP-ENV:Envelope>

06-08 23:36:16.467: I/GenericSoapClient(669): response value is: [true, Microsoft Windows 7 Professional |C:\Windows|\Device\Harddisk0\Partition1

*Fig 5.4.4 Răspunsul obținut de la Client*

# 5.5 Concluzii

Îmbunătățiri – istoric / cliențo mobile diferiți / servicii RESTful

Dezvoltare android

Dezvoltare web

Dezvoltare desktop

Testare

Citire soap mai buna

Log exceptions

6. Aplicații similare

Dropbox mobile;

Cloud

Server storage (limited)

Bibliografie

1. *The Busy Coder’s Guide to Android Development*, **Mark L. Murphy**; ISBN 978-0-9816780-0-9
2. *Professional Android 2 Application Development,* **Reto Meier;** ISBN 978-0-470-56552-0
3. *Hello, Android – Introducing Google’s Mobile Development Platform, 3rd Edition,* **Ed Burnette;** ISBN 978-1-934356-56-2
4. *Android Essentials,* **Chris Haseman;** ISBN-10 1430210648
5. *Beginning Smartphone Web Development,* **Gail Rahn Frederick, Rajesh Lal;** ISBN 978-1-4302-2621-5
6. *The Spring Framework – Reference Documentation; Version 3.0.0.*
7. *Spring Web Services – Reference Documentation; Version 2.0.5*
8. [*www.springsource.org*](http://www.springsource.org) *–* Documentație
9. *The Definitive Guide to Spring Web Flow*, **Erwin Vervaet;** ISBN 978-1-4302-1625-4
10. *Java de la 0 la Expert,* **Ștefan Tanasă, Cristian Olaru, Ștefan Andrei** ISBN 973-681-201-4