Documentaţie proiect VVS

Huple Răzvan - Daniel

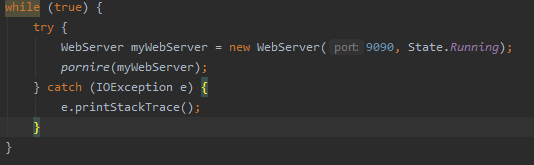
1. Introducere

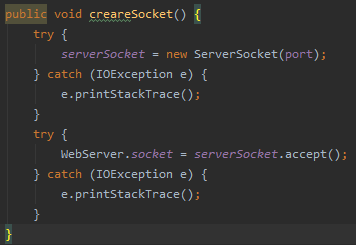
* Proiectul a vizat implementarea unui server web care să poată gestiona request-uri primite de la clienţi (să afişeze în browser paginile html cerute de client aflate într-o locaţie prestabilită).
* Serverul web este însoţit de o interfaţă grafică care este utilizată pentru pornirea/ oprirea server-ului şi pentru setarea anumitor configuraţii precum : portul la care ascultă, directorul rădăcină şi directorul mentenanţă. Serverul poate rula şi fără interfaţa grafică, dar nu se pot seta în mod dinamic configuraţiile amintite mai sus.
* Scopul principal al proiectului este de a folosi diferite tehnici şi tool-uri astfel încât să putem testa tot ce ţine de webserver-ul nostru.

* Limbajul de programare folosit pentru implementarea server-ului şi a interfeţei este Java. Pentru testarea unitară a webserver-ului am folosit framework-ul Junit. Pentru analiza statică am folosit 3 tool-uri : Findbugs, Checkstyle şi PMD. La analiza dinamică am utilizat VisualVm. Pentru testarea web şi a interfeţei grafice am folosit metoda Record & Play, cu ajutorul Selenium IDE, respectiv Squish IDE.

1. Implementare server

Funcţionalitatea server-ului este realizată în clasa Webserver care implementează interfaţa Runnable. Runnable este o interfaţă care conţine metoda nedefinită void run() şi este folosită pentru a executa cod pe un thread(proces) concurent. Un obiect al unei clase care implementează această interfaţă este folosit pentru a crea un thread, iar apoi metoda run este invocată în acel thread şi se execută separat. Serverul se bazează pe multi-threading, şi anume la fiecare request din partea unui client se va crea un nou thread. Acest lucru se realizează utilizând o buclă infinită , în care creez un obiect de tip Webserver, care va avea un socket pe care serverul ascultă la un anumit port şi aşteaptă conexiunea din partea clientului (serverSocket), şi un socket prin care se realizează comunicarea cu clientul (serverSocket.accept()). Apoi voi folosi obiectul webserver pentru a crea un nou thread.



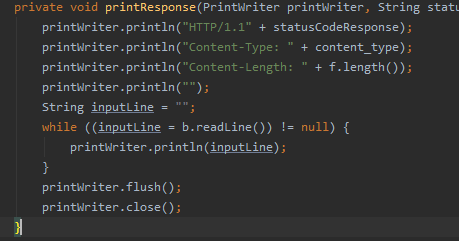


Practic programul stă blocat la linia de cod Webserver.socket = serveSocket.accept() şi nu se va întâmpla nimic până când nu apare vreun request din partea clientului.

Când se apelează thread.start() practic se execută metoda void run() din clasa WebServer care suprascrie metoda void run() din interfaţa Runnable. În poza de mai jos avem conţinutul metodei void run() .

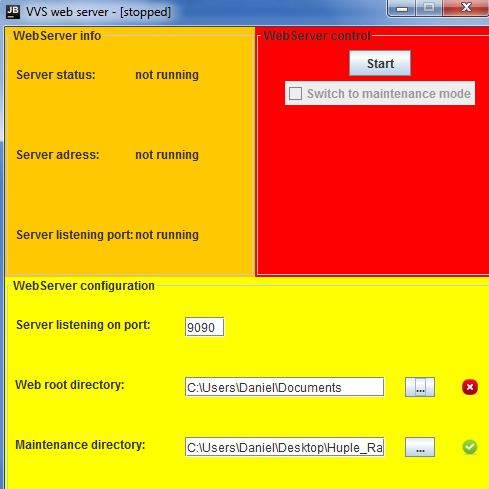


Practic avem 3 ramuri, una corespunzătoare pentru fiecare din cele 3 stări posibile: running, stopped, maintenance. În cazul stării running, prima oară citesc request-ul primit de la client. Acest lucru îl fac în metoda readLineRequest care are ca parametru socket-ul de unde va citi şi returnează un string cu prima linie din request (doar prima linie conţine informaţii care mă interesează). String-ul returnat îl dau ca parametru metodei getFileFromRequest care îmi extrage din linie doar fişierul pentru care s-a făcut request şi îl returnează. Acest string cu fişierul îl dau ca parametru metodei addFileToPath care îl adaugă la path-ul directorului rădăcină şi îl returnează tot ca string. Path-ul obţinut îl dau ca parametru metodei getFileFromPathRunning orStoppedState care verifică dacă fişierul de la path-ul dat există sau nu. În caz că există atunci se creează un fişier de tip File(path) care e returnat, în caz că nu, există 2 variante : fie path-ul dat este corect, dar fişierul cerut nu există şi atunci metoda va returna File(webrootpath+404.html), fie path-ul este greşit (este altul decât cel pentru webroot) şi atunci se aruncă excepţia verificată InvalidDirectoryException. Metodele getContentType şi getStatusCodeRunning returnează tipul fişierului (html,css,js etc), respectiv cod-ul care apare în header-ul răspunsului pe care îl trimit clientului ("404 Not Found","200 OK"). Pe acelaşi principiu se bazează şi implementarea pentru celelalte 2 stări.



La final se apelează funcţia printResponse care foloseşte un bufferedReader care citeşte informaţiile din fişier linie cu line şi care mai apoi sunt printate de PrintWriter la socket-ul prin care se face comunicarea cu clientul (socket.getOutputStream()). De asemenea înainte de printarea fişierului propriu-zis, sunt printate informaţii care ţin de Response Header.

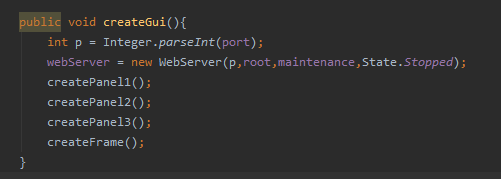
1. Implementare GUI



În imaginea de mai sus se poate vedea cum arată interfaţa. Practic este un jFrame care conţine 3 jPanel-uri: cel portocaliu, cel roşu şi cel galben. În cel portocaliu se afişează informaţii despre server : starea în care se află, adresa ip şi portul pe care serverul ascultă.

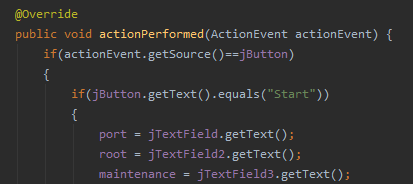
Toate acestea se schimbă în funcţie de stare, mai sus este starea stopped. În cel roşu avem un buton de start/stop şi sub el un checkbutton pentru a trece sau a ieşi din starea de mentenanţă. Iar în ultimul, cel galben, avem un textfield pentru a seta portul, şi 2 textfield-uri în care apar path-urile pentru web root directory şi pentru maintenance directory. Pentru a modifica cele 2 textfield-uri din urmă se poate da click pe cele 2 butoane alăturate din dreapta care vor deschide ferestre de dialog care permit alegerea facilă a celor 2 directoare. În dreapta de cele 2 butoane sunt 2 label-uri care conţin imaginea unui x roşu indicând faptul că directorul selectat nu este valid sau imaginea unei bife verzi care sugerează că directorul este valid. Toate aceste lucruri sunt implementate în metoda void createGui() din clasa Gui. De asemenea, clasa Gui are nevoie de un obiect de tip Webserver pentru a putea porni server-ul.

Pentru ca toate elementele să fie responsive (butoane, textfield-uri, etc) este nevoie ca ele să implementeze interfaţa ActionListener care conţine o metodă void ActionPerformed(ActionEvent var1) ce va trebui să fie implementată de fiecare element în parte. Dar pentru a avea toate implementările într-un singur loc, am pus clasa Gui să implementeze ActionListener şi să aibă metoda suprascrisă ActionPerformed . Astfel pentru fiecare element pe care îl vreau să fie "ascultat", spre exemplu butonul de start/stop, îi voi adăuga ca ActionListerner clasa Gui (jButton.addActionListener(this)). Deci când voi da click pe buton se va executa ActionPerformed din clasa Gui care este împărţită în mai multe ramuri în funcţie de sursa care a emis modificarea, în cazul de faţă butonul (if(actionEvent.getSource()==jButton)), fiecare ramură având implementarea ei. În imaginile de mai jos se poate vedea ce am explicat mai sus.









Pentru a porni interfaţa apelez în main metoda createGui() şi pornesc serverul pe care îl deţine într-o buclă infinită. Singura problemă care apare este atunci când se doreşte schimbarea portului pe care serverul ascultă. Când configurez noul port, serverul deja ascultă tot pe vechiul port şi atunci primul request care apare se va duce tot pe vechiul port. De abia la al doilea request portul nou va fi setat şi serverul va asculta pe el.

