UNIVERZITET SINGIDUNUM
FAKULTET ZA INFORMATIKU I MENADŽMENT



DOC. DR BOŠKO NIKOLIĆ

INTERNET PROGRAMIRANJE POMOĆU PROGRAMSKOG JEZIKA JAVA

BEOGRAD, 2008.

Univerzitet Singidunum

Doc. dr Boško Nikolić

INTERNET PROGRAMIRANJE POMOĆU PROGRAMSKOG JEZIKA JAVA

Beograd, 2008.

INTERNET PROGRAMIRANJE POMOĆU
PROGRAMSKOG JEZIKA JAVA

Autor:
Doc. dr Boško Nikolić

Recenzenti:
Prof. dr Milan Milosavljević
Prof. dr Mladen Veinović

Izdavač:
UNIVERZITET SINGIDUNUM
Beograd, Danijelova 32

Za izdavača:
Prof. dr Milovan Stanišić

Tehnička obrada:
Novak Njeguš

Dizajn korica:
Aleksandar Mihajlović

Godina izdanja:
2008.

Tiraž:
400 primeraka

Štampa:
Čugura print, Beograd
www.cugura.co.yu

ISBN: 978-86-7912-058-8

Predgovor

Internet, kao tehnologija i kao tržište, je tokom poslednjih desetak godina postigao do sada neviđeni razvoj i popularnost. Skoro da i ne postoji oblast ljudskog života u kojoj se ne mogu primeniti i Internet aplikacije. Školovani Internet eksperti su danas, a u sledećim godinama biće još više, najtraženiji deo softverske industrije.

Zato je odluka Fakulteta za Informatiku i menadžment da studentima omogući sticanje znanja iz najnovijih i najpopularnijih tehnologija ove oblasti. Programiranje Internet aplikacija se proučava iz više predmeta na ovom fakultetu. Sadržinom i konceptom udžbenik je prilagođen nastavnom planu i programu predmeta Programski jezici u okviru odeljenja fakulteta u Valjevu i Poslovnog Fakulteta u Valjevu, a pokriva i deo gradiva predmeta Aplikativni softver, koji se izučava na IV godini, studijskog programa Računarstvo i informatika.

Udžbenik detaljno razmatra Internet programiranje zasnovano na programskom jeziku Java. U prvom delu se prezentuju osnovne Java serverske komponente – servleti. Navodi se njihova osnovna struktura i mogućnosti realizacije Internet aplikacija samo pomoću ovih komponenti.

U drugom delu se razmatra JavaServer Pages tehnologija, sa posebnim naglaskom na njenu ulogu kao prezentacionog sloja u realizaciji višeslojnih aplikacija. Detaljno je realizovana i komunikacija ovog sloja sa servletima i implementacija Internet aplikacija pomoću ovih tehnologija.

Treći deo udžbenika obuhvata najnoviju Java tehnologiju za ovu vrstu aplikacija – JavaServer Faces (JSF) framework. U okviru ovog moćnog paketa razmatraju se osnovne tehnike, ali i napredni koncepti koji su se pojavili u JSF verziji 1.2, nastaloj 2007. god.

Pored teorijskih razmatranja navedenih tehnologija udžbenik sadrži i veliki broj praktičnih primera, koji se mogu izvršavati na velikom broju Web servera koji imaju podršku za Java kod. Takođe, ovaj udžbenik zajedno sa udžbenikom „Programiranje grafičkih aplikacija“, u kome se obrađuje realizacija klijentskih strana Internet aplikacija, čini jedinstvenu celinu i omogućava studentima samostalnu implementaciju Internet aplikacija proizvoljne složenosti.

Beograd, 2008. god. *Autor*

UNIVERZITET
PRIVATNI
AKREDITOVANI
UNIVERZITET
SINGIDUNUM

SADRŽAJ

l. Uvod 1

2. OSNOVE DIZAJNA INTERNET APLIKACIJE 8
2.1 Osnove web dizajna 8
2.2 Dizajniranje sa aspekta korisnika 12
2.3 Elementi sajta 22
2.4 Pravila efektnog dizajna 40

3. JAVA SERVLETI 45

4. JAVASERVER PAGES 65
4.1 Osnovna JSP sintaksa 70
4.2 Direktive 74
4.3 Bean-ovi i njihova upotreba u Internet aplikacijama 82
4.4 MVC pristup – kombinacija JSP strana i servleta 89
4.5 Expression jezik (EL) 98
4.6 JSTL (JSP Standard Tag Library) 102
4.7 Primer JSP-servlet MVC aplikacije 104

5. JAVASERVER FACES 120
5.1 Struktura JSF aplikacije 120
5.2 Servisi JSF framework-a 122
5.3 Životni ciklus 124
5.4 Navigacija 127
5.5 Bean-ovi 140
5.6 Rad sa pomoćnim fajlovima 148
5.7 Obrada događaja 152
5.8 Životni ciklus događaja 153
5.9 Događaji koji su izazvani usled promene vrednosti 154
5.10 Akcioni događaji 161
5.11 Tagovi osluškivača događaja 173
5.12 Korišćenje neposrednih ulaznih komponente 175
5.13 Prosleđivanje podataka iz korisničkog interfejsa serveru 177
5.14 Fazni događaji 181
5.15 Isti izgled svih stranica aplikacije 201
5.16 Tiles mogućnosti 217
5.17 Napredne komponente i validacija 238
5.18 Korišćenje konvertora 249
5.19 Rad sa bazama podataka 326

LITERATURA 346

UNIVERZITET
PRIVATNI
AKREDITOVANI
UNIVERZITET
SINGIDUNUM

1. UVOD

Internet aplikacije su postale sastavni deo naših života. Taj deo softverske industrije je onaj deo koji se trenutno najviše i najbrže razvija. Samim tim školovani Internet eksperti su danas, a u sledećim godinama biće još više, najtraženiji deo softverske industrije. Ako se proučava programiranje Internet aplikacija prvo pitanje koje se može postaviti je šta je uopšte Internet i šta se može smatrati Internet aplikacijom. Odgovori na ova pitanja mogu biti veoma različiti u zavisnosti od onoga ko na njih odgovara. Evo dva primera

„Internet je mreža svih mreža, povezuje univerzitetske, državne i privatne računare zajedno, i predstavlja infrastrukturu za korišćenje elektronske pošte, fajl arhiva, hipertekst dokumenata, baza podataka i ostalih resursa. To je veliki skup računarskih mreža koje postoje i deluju zajedno za prenos podataka i poruka na daljinu i računare, koji mogu biti u susednoj kancelariji ili na bilo kom mestu u svetu.“ William F. Slater, III, Predsednik Čikago ogranka USA Internet udruženja.

„Internet je najveća mreža na svetu, koja koristi TCP/IP protokole i paketnu razmenu podataka i izvršava se pomoću bilo koje komunikacione podloge.“ Dr. Vinton Cerf, koautor TCP/IP protokola

U ovim definicijama, pored toga što sami autori ističu svoje zasluge, izdvajaju se i činjenice o kojima treba voditi računa pri implementaciji Internet aplikacija. Prva je činjenica da je Internet u suštini mreža, da se aplikacije prenose i izvršavaju sa jednog računara na drugi. Druga je činjenica da, sve više, Internet ne podrazumeva samo računare, nego i druge uređaje za međusobnu komunikaciju, kao što su mobilni telefoni, PDA uređaji, ... Komunikacija preko Interneta se i dalje odvija paketnom razmenom pomoću TCP/IP protokola.

Što se samih brojki tiče one su sledeće:

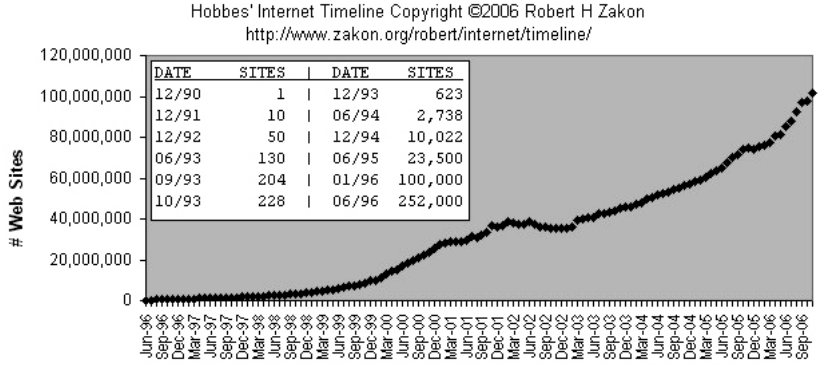
Godina	Broj servera	Godina	Broj servera
1977	111	1987	10.000
1981	213	1989	100.000
1983	562	1992	1.000.000
1984	1.000	2001	150-175 miliona
1986	5.000	2002	Preko 200 miliona
Do 2010, oko 80% planete će koristiti Internet			

U sledećoj tabeli su prikazani detaljniji podaci o trenutnom korišćenju Interneta u svetu.

WORLD INTERNET USAGE AND POPULATION STATISTICS						
World Regions	Population (2007 Est.)	Population % of World	Internet Usage, Latest Data	% Population (Penetration)	Usage % of World	Usage Growth 2000-2007
Africa	933,448,292	14.2 %	43,995,700	4.7 %	3.5 %	874.6 %
Asia	3,712,527,624	56.5 %	459,476,825	12.4 %	36.9 %	302.0 %
Europe	809,624,686	12.3 %	337,878,613	41.7 %	27.2%	221.5 %
Middle East	193,452,727	2.9 %	33,510,500	17.3 %	2.7 %	920.2 %
North America	334,538,018	5.1 %	234,788,864	70.2 %	18.9%	117.2 %
Latin America/Caribbean	556,606,627	8.5 %	115,759,709	20.8 %	9.3 %	540.7 %
Oceania / Australia	34,468,443	0.5 %	19,039,390	55.2 %	1.5 %	149.9 %
WORLD TOTAL	6,574,666,417	100.0 %	1,244,449,601	18.9 %	100.0 %	244.7 %

Slika 1. Statistički podaci o korišćenju Interneta u celom svetu

Što se tiče broja registrovanih sajtova i oni beleže interesentan rast. Trenutno postoji više od 100 milijardi sajtova, ali je zanimljiv i trend odnosa korisnik:serveri: 1996 godine je bio negde oko 150:1, 2000 se smanjio na 50:1, da bi 2006 bio 10:1.



Slika 2 Rast broja sajtova

Internet kao tržište je do sada pokazao neviđeni razvoj. Kao primer mogu da posluže sledeći podaci: da bi osvojio tržište od 50 miliona ljudi radiu je trebalo 38 godina. To isto tržište televizija je dobila za 13 godina. S druge strane od kada je postao javni servis, Internet je počelo da koristi 50 miliona ljudi za samo 4 godine! [1].

Ne postoje pouzdani statistički podaci o broju računara u Srbiji. Oslanjajući se na podatke iz istraživanja na uzorcima Centra za politikološka istraživanja i javno mnenje Instituta društvenih nauka iz Beograda, procenjuje se da su domaćinstva početkom 2002. godine raspolagala sa 435.000 računara. Skoro svako sedmo domaćinstvo u Jugoslaviji imalo je računar, u Srbiji skoro svako šesto, a u Crnoj Gori svako deseto. Unutar Srbije vlasnici računara koncentrisani su u gradovima (25% prema 6% u seoskim naseljima). U Srbiji, u regionu Beograd više od četvrtine domaćinstava ima računar (26%; tu je locirano više od trećine svih računara), u Vojvodini 16%, a u Centralnoj Srbiji 12%. Što se tiče povezanosti posedovanja računara sa finansijskom situacijom samog domaćinstva, u Srbiji domaćinstva sa mesečnim prihodom do 100 evra (koja čine 17% u ukupnoj strukturi domaćinstava) poseduju računar u 2% slučajeva, ona sa prihodom od 101 do 200 evra (čine 29% od ukupne strukture domaćinstava) - u 9%, i ona sa prihodom većim od 200 evra (čine 40% od ukupne strukture domaćinstava) - u 28% slučajeva. Domaćinstva sa prihodom od 300 ili više evra (čine 21% od strukture domaćinstava) poseduju preko 50% računara u Srbiji.

Pored računara u domaćinstvima, građanima Srbije na raspolaganju su i računari na radnim mestima, u školama i na fakultetima, u bibliotekama i u Internet-kafeima. Ni broj ovih računara nije poznat, ali se na osnovu podataka istog istraživanja procenjuje da samo na poslu pristup računaru ima oko 5% punoletne populacije, a na svim ostalim nabrojanim mestima ukupno oko 1%. Skoro trećina korisnika računara u domaćinstvima nije pretplaćena na Internet, a petina članova domaćinstava pretplaćenih na Internet nikada ne koristi Internet, tako da se procenat korisnika Interneta u Jugoslaviji procenjuje na ukupno 10% populacije stare 18 ili više godina, od čega 8% pristupa Internetu kod kuće (više od polovina njih ima pristup i na poslu), i još 2% koji pristup mreži imaju samo na poslu. Najaktivniji korisnici i računara i Interneta su građani stari između 30 i 50 godina, a najstariji (60 ili više godina) po svim pokazateljima aktivniji su od pripadnika najmlađe generacije (18 do 29 godina).

Zašto su Internet aplikacije postigle toliku popularnost? Postoji više razloga. Za običnog korisnika njihov interfejs je jednostavan, mogu se izvršavati na klik miša. Ove aplikacije su platformski neutralne, što znači da imaju isti izgled za većinu korisnika. Na vrlo jednostavan i intuitivan način se koristi interaktivnost sa krajnjim korisnikom. Za programere bitna je jednostavna nadogradnja trenutnih verzija i jeftina distribucija do samog korisnika.

Danas, programi koji se izvršavaju preko Interneta se mogu podeliti globalno i dve grupe, na Internet sajtove i Internet aplikacije. Internet sajtovi daju statičke informacije i po pravilima dizajna se razlikuju od „običnih", standalone, aplikacija. S druge strane Internet aplikacije, mnogo više poštuju pravila i dizajn koji važi kod uobičajenih aplikacija, osim načina izvršavanja, što krajnjeg korisnika i ne mora da zanima. Poslednjih godina sve su popularniji i Web portali, kao i blogovi. Portali su sajtovi sa specijalizovanim sadržajem i informacijama. Portali prezentuju informacije iz različitih izvroa na jedinstven način. Popularni portali su MSN, Yahoo, Na dlanu i AOL. Pored standardnih mašina za pretragu, Web portali nude i druge servise, kao što su novosti, stanje berze, e-mail i druge mogućnosti. Portali omogućavaju konzistantan pristup i izgled sa mogućnošću kontrole pristupa i procedura za više aplikacija, čime se podržava zajednički rad više različitih entiteta. Poslednjih godina pojavljuje se i pojam personalnog portala, Internet sajta koji pruža personalizovane mogućnosti svojim posetiocima. Dizaniran je da koristi distribuirane aplikacije, sa većim brojem i različitim tipovima softvera i hardvera, omogućava servise iz različitih izvora. Budući zahtevi portala koji su zasnovani na poslovanju biće mogućnost da se njihov sadržaj prikazuje na različitim platformama, kao što su personalni računari, pomoćni PDA uređaji i mobilni telefoni. Trenutno, postoji i veliki broj alata koji pomaže pri boljem i jednostavnijem organizovanju i dizajniranju ovakve vrste sajtova. Blogovi (ime dobijeno od web log) su jedna vrsta on-line dnevnika, gde korisnici postavljaju svoja razmišljanja i daju komentare na određene događaje u hronološkom redosledu. Mogućnost da i čitaoci ostavljaju svoje komentare u interaktivnom formatu je takođe važna odlika bloga.

Iz svih navedenih razloga može se zaključiti da je znanje programiranja Internet aplikacija veoma bitno za nekog ko želi da se bavi razvojem softvera u bilo kakvom vidu. Tada se može postaviti pitanje šta koristiti i od čega početi.

Današnji trend koji je prisutan kod velikog broja uspešnih Internet aplikacija je inovativna upotreba i integracija velikog broja tehnologija. Ovaj trend je poznat pod jednim još uvek kontraverznim i neprihvaćenim pojmom Web

2.0. Ovaj pojam je za većinu ljudi koji se bave Internetom mogućnost da veliki broj trenutnih tehnologija nazovu jednim imenom. Tako da ne postoje jedna Web 2.0 tehnologija i to za ljude koji se bave profesionalnim razvojem ove vrste softvera predstavlja izazov. Prethodnih godina je postojala tendencija da se specijalizuje razvoj Internet aplikacija. Postojao je veliki broj eksperata iz pojedinih tehnologija kao što su HTML, CSS, HTTP, JavaScript, XML, serverska tehnologije za razvoj, ... koji su posedovali limitirano celokupno znanje. Iako je ponekad potrebno usko specijalizovano znanje, ovakav pristup se pokazao kao nepraktičan, već je za nekog ko želi da se ozbiljnije bavi razvojem ove vrste aplikacija, potrebno da ima veće znanje iz svih tehnologija koje se trenutno uspešno primenjuju.

S druge strane trenutno teško je definisati šta predstavlja pojam „uspešno primenjuju“. Dok je za klijentsku stranu, skoro od samog početka, definisan standard u vidu HTML, CSS, JavaScript, za razvoj sreverske strane postoji više dilema. Ipak, mogu se izdvojiti tri trenutno vodeće tehnologije. Microsoft, kao jedna od vodećih firmi u razvoju softvera, ima svoju podršku za Internet aplikacije. Ranije je to bila ASP tehnologija, danas je razvoj Internet aplikacija moguć pomoću više Microsoft alata. Za sve njih važi da su vezane za Microsoft operativne sisteme i za Internet Service Provider (ISP) kao Web servere. Za razvoj manjih aplikacija trenutno je najpopularnija PHP tehnologija. Hypertext Preprocessor (PHP) je serverski skript jezik koji se koristi za realizaciju interaktivnih Web sajtova. To je open-source tehnologija, jednostavna za učenje, koja koristi mogućnosti objektno-orijentisanog programiranja i podržava mnoge baze podataka (MySQL, Oracle, Sybase, sve koje su ODBC-bazirane). Pored niza dobrih i uspešnih osobina, PHP ima i bezbedonosne probleme, čime otežava razvoj kompleksnijih komercijalnih Internet aplikacija.

Tehnologija koja će se razmatrati u ovoj knjizi je programski jezik Java,. Autor smatra da je to, danas, najefikasnija i najmoćnija tehnologija za razvoj ovakve vrste aplikacija, sa velikim mogućnostima razvoja i napredovanja.

Ako se pogleda istorijski razvoj, priča o Javi kao programerskom alatu na serverskoj strani aplikacija je počela 1997 kada je Sun Microsystems realizovao "Java Web Server" beta i Java Servlet Developers Kit. Servleti su specifične Java klase koje imaju mogućnost da se izvršavaju na strani servera. Servleti dinamički prihvataju i obrađuju mrežne zahteve i odgovore, najčešće koristeći HTTP protokol. U junu 1999, Sun je predstavio Java Server Pages (JSP), koje su omogućile realizaciju Java koda zajedno sa dizajniranim JavaScript i HTML stranicama.

JSP stranice, koje su kasnije još razvijane (sa JSP Verzijom 2.0), su dizajnirane da izvrše enkapsulaciju domenske logike u okviru standardnih i dodatih tagova, i tako izvrše odvajanje ovog sloja od komponenti prezentacione logike na JSP strani. Kasniji koncept je nazvan "the stuff that people see" kada se programeru omogućava da radi sa gotovim HTML-baziranim šablonima. Idealno, JSP koriste tagove da bi pristupile bazama podataka i zaštitile pravila domena, i od statičkih ili dinamički generisanih tekstualnih šablona, kao što su XML ili XHTML, da bi se generisale krajnje stranice za korisnika. Pojava Jave kao serverske komponente je omogućila samim programerima upotebu alata koji je apsolutno objektno orijentisan i modularan, i na taj način jednostavnije i efikasnije organizovanje projekata različitih veličina. Pored toga, dobila se mogućnost korišćenja i velikog skupa pomoćnih Java klasa, kao što su klase za rad sa Stringovima, sa ulazno-izlaznim fajlovima, matematičke kalkulacije, ...

Sledeća velika prednost ove tehnologije je mogućnost korišćenja u okviru više platformi. Moguće je realizovati Internet aplikaciju, zapakovati je u specijalne JAR fajlove za Web komponente, koji se nazivaju Web Application Archive (WAR) fajlovi, pa postaviti dobijene WAR fajlove na različite servere, pod različitim operativnim sistemima. Java Web komponente nisu vezane za određene operativne sisteme, ili za određeni serverski softver, kao neke druge softverske tehnologije za razvoj ove vrste aplikacija.

Od kraja 2003, Java je već dobila status predvodnika razvoja serverske strane. Servleti i JSP stranice su uključene u Java 2 Enterprise Edition (J2EE), veoma prihvaćenu „enterprise“ tehnologiju za mrežno bazirano i distribuirano programiranje. Tako su stotine hiljada programera širom sveta počeli da razvijaju "web sloj" u okviru J2EE-baziranih tehnologija, koristeći servlete i JSP stranice, kao i specijalne web framework-e, na primer Struts.

Tako su programeri došli u situaciju da biraju između dve mogućnosti. Jedna je bila da koriste „rapid devolpment“ stil, u okviru koga su koristili vizuelna okruženja za razvoj, kako što je Microsoft ASP.NET. Drugi način je bio da koriste „hard-core coding“ stil, koji je podrazumevao pisanje dosta koda sa dobijenim visokim perfomansama, ponajpre serverske strane. Vodeća tehnologija ovakvog stila je bila J2EE. Na taj način, programeri su bili pred nedoumicom. Da li koristiti jednu tehnologiju koja je veoma skalabilna, portabilna na mnogim platformama, a podržava je veliki broj proizvođača. S druge strane, ASP.NET je pružao mogućnost kreiranja atraktivnog korisničkog interfejsa bez dugog programiranja. Postavilo se pitanje da li je moguće napraviti tehnologiju, koja ima oba svojstva.

Tako se došlo do JavaServer Faces (JSF), tehnologije koja nudi brz razvoj korisničkog interfejsa, uz sve prethodno navedene osobine dobre osobine Java programskog jezika, uključujući i sve dotadašnje mogućnosti razvoja serverske strane.

JavaServer Faces (JSF) je jedan od poslednjih Java tehnologija za razvoj Internet aplikacija, razvijena na osnovu iskustva dobijenog korišćenjem Java Servleta, JSP stranica, i velikog broja komercijalnih i open source framework-a za Internet aplikacije. JSF definišu događajima vođen, baziran na komponentama model za razvoj Internet aplikacija, sličan modelu koji je godinama korišćen za razvoj standalone vizuelnih aplikacija. Ovakav pristup je privukao veliku pažnju u sferi Internet softvera. Java Community Process (JCP) je objavio specifikaciju i omogućio proizvođačima da se fokusiraju na implementaciju alata za razvoj i komponenti korisničkog interfejsa baziranih na JSF specifikaciji.

Cilj ove knjige je da čitaocu približi osnove programiranja Internet aplikacija pomoću Java tehnologija. Opisani su osnovni koncepti, kao što su, već pomenuti, Java servleti, JSP stranice i Java framework JSF. Pored teorijskih postavki, postoji i veliki broj primera koje čitalac može da testira i pokreće uz bilo koji trenutno popularnih alata. Autor je koristio Eclipse i NetBeans okruženja.

Za razumevanje primenjenih tehnologija potrebno je da čitalac bude upoznat sa osnovama HTML-a i JavaScript-a Da bi se dati primeri sa serverskim komponentama pokretali i izvršavali, potrebno je primer-aplikacije postaviti na određeni Web server. Primeri dati u knjizi bazirani su na Jakarta Tomcat Web serveru i u Prilogu je dat opis osnovnih operacija koje se mogu izvršiti sa ovim serverom. Rad sa drugim Web serverima koji imaju podršku za Javu je sličan i čitalac neće imati nikakvih problema da date primere prilagodi.

2. OSNOVE DIZAJNA INTERNET APLIKACIJE

Dizajn Internet stranica je karakteristika o kojoj treba dosta voditi računa pri realizaciji ove vrste programa. Za ovaj deo softvera ne mogu da budu odgovorni samo programeri, ali ni samo dizajneri, zbog nepoznavanja okruženja u kome se softver izvršava. Zato je potrebno da i jedni i drugi poznaju osnove razvoja svih delova ovakvog softverskog sistema. Iz tih razloga u nastavku će se definisati osnovni pojmovi i koncepti vezani za Web dizajn.

2.1. Osnove Web dizajna

Iako sa Web dizajnom može da se poveže na desetine različitih disciplina, njegova četiri glavna aspekta su sadržaj, tehnologija, vizuelni elementi i ekonomičnost. Uloga tehnologije na Web sajtu ogleda se u obezbeđivanju funkcija sajta. Vizuelni elementi obezbeđuju potrebnu formu. Konačno, za većinu sajtova potrebno je razmotriti i ekonomske implikacije pravljenja sajta. Ukoliko nije sasvim jasno zbog čega treba napraviti sajt ili od njega nema neke koristi, on se obično i ne pravi. Naravno, značaj svakog pojedinačnog aspekta razlikuje se od jednog do drugog sajta. Obično se neće razmatrati ekonomski pokazatelji početne strane, ali će se obratiti pažnju na taj aspekt na sajtu na kome se nešto prodaje. Neophodan broj različitih disciplina koje je potrebno razmotriti pri izradi sajta varira, pa je stoga nemoguće da se o Web dizajnu govori na jedinstven način.

Ako Web sajt zamislimo kao piramidu, sadržaj predstavlja blokove − gradivne elemente piramide, a njen temelj počiva na vizuelnim elementima i tehnologiji uz čvrstu vezu sa ekonomijom kao osnovnim motivom koji opravdava čitav projekat. Dizajneri ili njihovi poslodavci često provode više vremena razmatrajući sopstvene potrebe i želje nego potrebe i želje posetilaca Web lokacija što često dovodi do neuspeha. Zato, pri izradi sajta, prvo treba znati:

- šta uraditi
- koja je svrha sajta
- šta korisnici sajta žele
- koji je željeni stil i značenje sajta

Tek onda treba osmisliti osnovne opcije i izgled sajta tako da privuče korisnikovu pažnju. Pre početka dizajniranja sistema mora se definisati sopstveni cilj:

- Da li je cilj impresionirati druge Web dizajnere?
- Da li je cilj zadovoljan klijent?
- Da li je cilj dobar profit?

Takođe, treba definisati šta predstavlja uspeh i kako će se proceniti da li je uspeh postignut.

Nakon arhitekture sajta, sledeći korak je osmišljavanje vizuelnog izgleda i prikaza svih opcija sajta. U ovoj fazi se određuje sadržaj, deo za navigaciju, logo i osnovne linije. Može se ponovo krenuti od olovke i papira dok se sve ne osmisli.

Pri osmišljavanju vizuelnog izgleda sajta trebalo je voditi računa o principima grafičkog dizajna. Postoji 5 osnovnih principa grafičkog dizajna i to su:

- jednostavnost
- kontrast
- prazan prostor
- balans
- poravnanje

Princip jednostavnosti ili „Keep it simple“ u suprotnosti je sa ostalim fazama dizajna kada dizajner dodaje nove i nove opcije misleći da neće škoditi ako ih bude više. Osnov ovog principa je stav da se mora imati dobar razlog za svaki elemenat koji se dodaje. To se postiže sledećim tehnikama:

- Redukcija – Redukcijom se eliminiše sve što nije neophodno u četiri koraka:
    1. odlučiti šta je osnovna potreba samog dizajna
    2. kritički ispitati svaki element
    3. ukloniti sve što nije potrebno
    4. testirati dizajn i onda eventualno ukloniti i neki element koji je procenjen kao neophodan
- Kombinovanje elemenata – npr. isto dugme ima više funkcija
- Regularnost – Minimizovati razlike između elemenata koji su ostali, koristiti iste fontove, boje, debljine linija, dimenzije i sl.

Kontrast daje dodatnu vizuelnu dimenziju, kao što su boja i/ili veličina. Kontrast mora biti tako izabran da je prihvatljiv u samom zadatku i da postoje jasne razlike između vizuelnih dimenzija. Ovde razlikujemo više vizuelnih promenljivih kao što su: jačina, obojenost, tekstura, pozicija, oblik, orijentacija i veličina. Obojenost predstavlja čistu boju dok je prva

vrednost jačina izabrane boje. Vizuelne promenljive se koriste za komunikaciju, kodiranje podataka i crtanje razlika između vizuelnih elemenata.

Postoje dve karakteristike vizuelnih promenljivih: skala i dužina skale. Dužina skale je broj različitih vrednosti koje se mogu definisati nad njom.

Skala može biti :

- nominalna – predstavlja listu kategorija pomoću koje se vrše poređenja jednakosti, gde se naravno različite vrednosti ne mogu porediti, npr. nominalna veličina je obojenost
- uređena – definiše poredak vrednosti promenljivih, npr. pozicija i veličina su uređene vrednosti
- kvantitativna – kod ove skale se može primetiti mera razlike u okviru poređenja, npr. pozicija je kvantitativna veličina, jer se za svake dve vrednosti na grafiku može reći koja je iznad

Vrednost promenljive ima sledeće osobine:

- selektivnost – stepen kojim se pojedinačna vrednost može selektovati iz ulaznog vizuelnog polja, npr. raspoznavanje zelenog objekta
- asocijativnost – mogućnost korisnika da ignoriše pojedine promenljive u trenutku kada se usredsredi na druge, npr. veličina i jačina nisu asocijativne što se može zaključiti ako se posmatraju predmeti male veličine i slabe boje.

Za kontrast je izuzetno važno izabrati pravu vizuelnu promenljivu. Kada se promenljiva izabere, potrebno je koristiti što više dužinu promenljive. Prvo se prepozna maksimum i minimum promenljive, a onda istražuje ceo opseg. Takođe, poželjna je primena multiplikativnog skaliranja gde je svaka sledeća veličina 1.5 do 2 puta veća od prethodne. Dalje, naslov dokumenta je često veći od ostalog teksta (veličina), centriran je (pozicija), boldovan (jačina), a i može biti prikazan drugom bojom. Kod slika se koriste pojačane razlike, znači objektu se daju uveličane proporcije da bi se lakše uočio na ekranu. Svi primenjeni efekti kontrasta mogu se lako testirati ako se zatvori jedno oko i gleda na drugo. Tako se poremeti uobičajeni fokus posmatranja i mogu da se primete razne pojedinosti koje će uočiti i korisnik pri površnom posmatranju interfejsa.

Prazan prostor ima najbitniju ulogu u kompoziciji. Dobar dizajn znači balans između potrebe za praznim prostorom i želje da se prikaže što više informacija. Nedostatak praznog prostora dovodi do otežanog posmatranja ekrana. Takođe, labele i kontrole koje su preblizu krajevima ekrana sporije se „skroluju“. Kada je objekat okružen praznim prostorom potrebno je

zadržati proporciju između objekata i okruženja, što će biti lakše ako se postave margine oko samog sadržaja. U slučaju grupisanja kontrola ili objekata zarad većeg praznog prostora i organizovanijeg ekrana, treba voditi računa o Gastelt principima grupisanja.

Gastelt principi grupisanja obuhvataju sledeće:

- Bliskost – elementi koji su bliži jednostavnije se grupišu
- Jednostavnost – elementi koji imaju slična svojstva jednostavnije se grupišu
- Kontinuitet – oko očekuje da kontura bude neprekidan objekat, pa tako vidimo presek dve linije, a ne da su se 4 linije susrele u jednoj tački
- Kompletnost – oko vidi kompletnu, zatvorenu figuru pa čak i onda kad mu ivice objekta nisu kompletne
- Prostor – kada se dva elementa preklapaju, manji će biti shvaćen kao figura ispred veće površine
- Simetrija – oko daje prednost objektima koji su simetrični

Sledeći princip grafičkog dizajna je *balans*. Balans i simetrija su važni alati u realizaciji dizajna. U grafičkom dizajnu simetrija retko znači istu količinu elemenata na svakoj strani simetričnih površina. Kako se pod pojmom elemenata misli i na bele i na ne-bele piksele, trebalo bi da ove površine budu balansirane. Jednostavan način da se postigne balans jeste da se centriraju elementi u okviru prozora i automatski se postiže balans oko vertikalne ose. Dobar primer balansa je početna strana sajta Google.

Poravnanje, horizontalno i vertikalno, čini osnovu dizajna i doprinosi jednostavnosti dizajna. Postoje dve škole mišljenja o poravnanju labela. Jedna škola smatra da leve ivice labela treba da budu poravnate, a druga da poravnate treba da budu desne ivice. Brojna istraživanja i eksperimenti nisu pronašli značajnu razliku u rezultatima ova dva pristupa. Najbolji rezultati će se postići ako su labele iste veličine i ako su dugačke labele podeljene u više linija. Što se tiče kontrola, one bi trebalo da budu poravnate i po desnoj i po levoj ivici. Nekada ovo zvuči nerazumno, kao npr. u slučaju datuma gde polje za datum treba da bude iste veličine kao polje za ime i prezime. U tom i sličnim slučajevima treba voditi računa o aranžiranju elemenata. Takođe, labele i kontrole u istom redu treba da dele istu srednju liniju.

2.2. Dizajniranje sa aspekta korisnika

Kao što je već rečeno, uobičajen propust Web dizajnera jeste to što sajtove prave više prema svojim potrebama i željama nego prema potrebama korisnika sajta. Ne može se garantovati izrada dobrog sajta ukoliko dizajner ne vodi računa o potrebama korisnika. Dizajner detaljno poznaje Web sajt, zna gde se informacije nalaze, podrazumeva se da mu je na ekranu optimalna rezolucija, da je Web browser ispravno podešen itd. Mnogi korisnici nemaju toliko znanja o samom sajtu, oni čak ne moraju deliti ni isto zanimanje za sajt kao dizajner. Dolazi se do termina „Dizajniranje sa aspekta korisnika“ koji se odnosi na proces dizajniranja koji uvek u prvom planu ima korisnika. Ovde treba voditi računa o tome da su korisnici među sobom prilično različiti i ne postoji tzv. „tipičan“ korisnik. Ono što je jednome od njih lako, drugome može da bude teško. Sajtovi namenjeni „običnom“ korisniku ne moraju da zadovolje svakog korisnika. Korisnici su skup različitih pojedinaca koje povezuju izvesne zajedničke sposobnosti i osobine. Dobar sajt mora da vodi računa o razlikama između pojedinaca, usredsređujući se pri tome na ono što je među njima zajedničko.

Dobri Web sajtovi zaista koriste posetiocima. Njihova osobina da budu korisni posetiocima može se posmatrati kao kombinacija korisnosti i upotrebljivosti. Korisnost je, u stvari, ona funkcionalnost sajta koja izlazi u susret potrebama korisnika. Upotrebljivost se odnosi na sposobnost posetioca da koristi sajt kako bi postigao određeni cilj. Dakle, ako neki sajt ne funkcioniše ispravno, bilo zbog grešaka dizajnera, bilo zbog lošeg izbora dizajna zbog koga se teško koristi, to nije dobar sajt. Upotrebljivi sajtovi su efikasni, snalaženje na njima je lako i oni korisnicima omogućavaju da zadovoljavajuće i bez grešaka postignu svoj cilj.

Web sajt se može smatrati uspešnim samo ako je koristan, upotrebljiv, ispravan i dopadljiv. Svaki od ovih epiteta može se tumačiti i subjektivno, izuzev ispravnosti. Dobro dizajniran sajt mora najpre da funkcioniše bez greške. To znači da sajt ni pod kojim okolnostima ne sme da se zaglavi. HTML kod mora da bude besprekoran. Svi interaktivni elementi moraju da rade ispravno i bez poruka o greškama. Navigacija na sajtu uvek mora da funkcioniše nepogrešivo. Raskinute veze, propraćene uobičajenom porukom „404: Not Found“ nisu obeležje dobrog sajta.

Između ostalih, razlozi zbog kojih nastaju problemi sa izvršavanjem su: promene u tehnologiji, neiskustvo autora, tesni vremenski rokovi, nepostojanje metodologije i zanemarivanje ograničenja samog medija.

Lokacija napravljena jednom tehnologijom često za nekoliko meseci mora potpuno da se preradi. Treba uzeti u obzir da je u poslednjih nekoliko godina HTML pretrpeo tri velike revizije, a da su ga u istom periodu i proizvođači Web browser-a dopunili brojnim vlastitim komponentama. Tehnologije kao što je JavaScript menjaju se još brže. Web dizajner mora da upozna ključne Web tehnologije (HTML, kaskadne stilove (CSS) i dr.) pre nego što počne sa izgradnjom lokacije. Dizajner mora da zna da i sama mreža, serveri i protokoli mogu da utiču na njegovo delo.
Znači, cilj svakog dizjanera je da sajt funkcioniše pravilno i da pored klijenta koji je naručio sajt, i korisnik bude zadovoljan onim što vidi. Kada se definišu ciljevi sajta i ciljevi korisnika, kao i način na koji se to postiže, sledeća faza je planiranje strukture sajta. Prvi korak je pomoću olovke i papira prikazati najvažnije interakcije između korisnika i sajta. Potrebno je naznačiti koje su ulazne tačke sistema, moguće akcije, kretanje korisnika po sajtu i dr. Zatim se pravi jednostavna i ekonomična struktura sajta. Treba isplanirati koje stranice treba da se nalaze u okviru sajta, kako su grupisane i kako ih korisnik može posetiti.

Trenutno se razlikuje više arhitektura sajta:

- Sve-u-jednom – najjednostavniji model gde se celokupan sadržaj stavlja na jednu Home stranicu
- Flat – jednostavan model gde se sve stranice postavljaju kao čvorovi, i svakoj stranici se može pristupiti iz bilo koje druge
- Indeksirana – gde postoji posebna stranica koja predstavlja listu sadržaja datog sajta-indeks i organizovana je tako da omogući jednostavniju pretragu samog sajta
- Ulančana – ovaj model je koristan za razgranate linearne algoritme, primer e-mail aplikacija
- Stroga hijerarhija – gde se stranica može posetiti samo preko stranice prethodnika
- Višedimenzionalna hijerarhija – gde postoji više pretraživanja istog sadržaja, a struktura sadržaja se prikazuje različito u zavisnosti od moda u kome se korisnik nalazi, primer Amazon.com
- Pretraživanje – u kome se korisniku omogućava da automatski pređe na traženu infomaciju, bez prethodnog pretraživanja po hijerarhiji ili indeksima

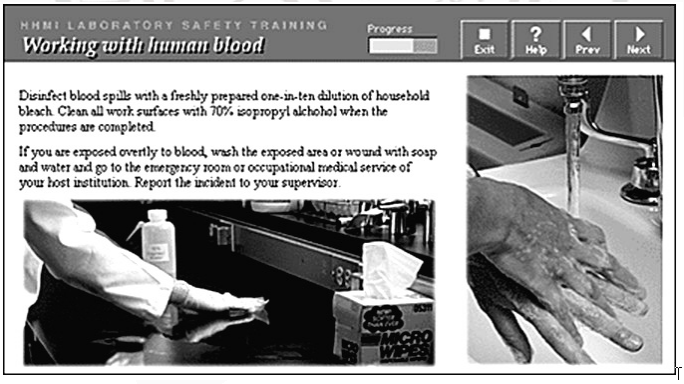
Izbor pravilne prezentacije zavisi od velikog broja faktora, među kojima su korisnikovi ciljevi, tehničke arhitekture, kompatibilnosti i dr. Nekada je jednostavna arhitektura sajta i najpodesnija.

Način prezentacije informacija na stranicama je određen parametrima koje definišu zadati ciljevi, praktična realizacija u izbaranoj tehnologiji i krajnji korisnici. Naredni dijagram prikazuje glavne oblasti obrađene u okviru Web stranica posmatrane u funkciji dve promenljive – linearnosti strukture prezentacije i dužine korišćnja sistema od strane tipičnog korisnika.
Generalno posmatrano, neke načini korišćenja Interneta su više strukturirani i zavise od korisnika koji ih posećuje i njihovih ciljeva, bez udubljivanja u motivaciju samog sajta. Intranet mreže, sajtovi za obuku, edukacioni sajtovi, sajtovi sa referencama, kao i novinski ili informativni sajtovi zavise od korisnika koji tačno znaju koju vrstu informacija žele i posećuju navedene sajtova sa unapred definisanom idejom šta od sajta očekuju. E-commerce i opšti sajtovi imaju kompleksnu dvostruku ulogu: da motivišu običnog korisnika da provede određeno vreme na njjihovom sajtu, ali i da obezbede efikasan pristup određenim proizvodima i informacijama za stalne posetioce i kupce.
Neiskusni dizajneri i klijenti pokušavaju da urade sve što je moguće da bi privukli pažnju na svoj Web sajt, praveći pri tome osnovnu grešku da maksimiziraju ulogu grafike u odnosu na ostale moguće atribute dizajna. Ovakav pristup dovodi do toga da korisnik koji je izložen napadnoj grafici ili Flash animaciji, veoma brzo gubi pažnju i ne primećuje osnovnu poruku samog sajta. Istraživanja pokazuju da ovakav pristup dovodi da već posle 10-15 sekundi korisnik ne primećuje nove detalje, a Web sajt se ne realizuje zbog trenutne korisnikove pažnje. Većina korisnika je zasićena od bleštećih animiranih slika i dugog učitavanja, i trudi se da izbegne takve sajtove. Uspešan sajt zahteva realan i stalan napredak, i jedino ravnoteža između sofisticiranog vizuelnog prikaza i strukturiranog sajta omogućava zadovoljne korisnike.
Naveden je ogroman broj Internet aplikacija, koje se međusobnp veoma razlikuju. Zato eje teško izvršiti njihovu preciznu klasifikaciju. Ipak mogu se izdvojiti sledeće vrste Internet aplikacija:

- Trening
- Obučavanje
- Kontinuirano obrazovanje
- Reference
- Opšti i magazinski sajtovi
- Sajotvi sa vestima
- Elektronsko poslovanje

Trening

Web-bazirane aplikacije za trening bi trebale da budu linearne i uobičajeno nude nekoliko opcija odstupanja od centralnog toka prezentacije. Ne treba zbunjivati korisnika ponudom previše linkova i odstupanjem od centralne poruke. Ograničenje linkova na "Sledeca tema" i "Prosla tema" garantuje da će svi korisnici videti istu osnovu prezentacije i omogućava da se preciznije predvidi korisnikovo ponašanje i provedeno vreme. Mnoge prezentacije ove vrste pretpostavljaju određeno vreme korišćenja, pa ukidaju korisnikovu sesiju nakon tog vremenskog intervala. U takvim slučajevima obavezno treba navesti korisniku koliko traje sesija, koliko je preostalo vremena do njenog kraja i upozoriti ga da ne bude preopširan u radu sa ponuđenim materijalima, posebno u slučajevima ako na određeni način korisnik i plaća korišćenje sesije. Uobičajeno za ove aplikacije imaju korisničku stranu za logovanje i da nude pitalice u formatu tačno-netačno ili izbora između ponuđenih odgovora. Registracija korisnika i rezultati na pitalicama se zapisuju u baze podataka koje su povezane sa datim sajtom.



info.med.yale.edu/caim/hhmi

Obučavanje

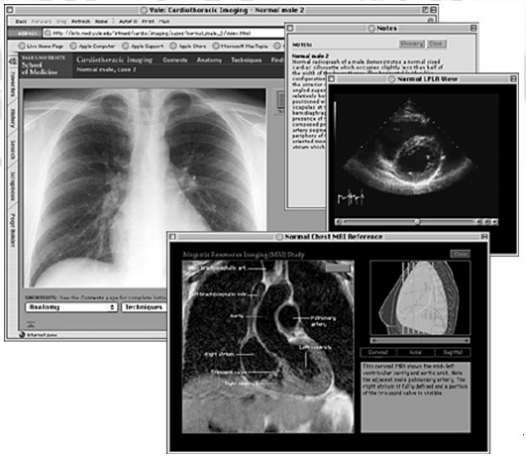
Kvalitetne aplikacije za obučavanje su takođe oslonjene na strogu centralnu kontrolu, ali nude i više mogućnosti za interesantne digresije u odnosu na glavnu temu sajta. Informacije koje se prikazuju su zahtevnije i traže više predznanja nego što je slučaj kod aplikacija za trening. Linkovi su najmoćniji aspekt samog Web-a, ali mogu skrenuti korisnika sa same prezentacije. Ako postoje linkovi na Web-bazirane resurse izvan datog sajta, trebalo bi takve linkove grupisati na odvojene stranice, izvan samog materijala koji se prezentuje. Korisnici često žele i da odštampaju ponuđeni materijal, pa da ga kasnije proučavaju sa papira. Zato bi trebalo pojednostaviti ovu opciju, ponudom "printing" verzije koja bi grupisala više stranica sa jednom temom u jednu dugačku stranicu.



www.dartmouth.edu/~chemlab

Kontinuirano obrazovanje

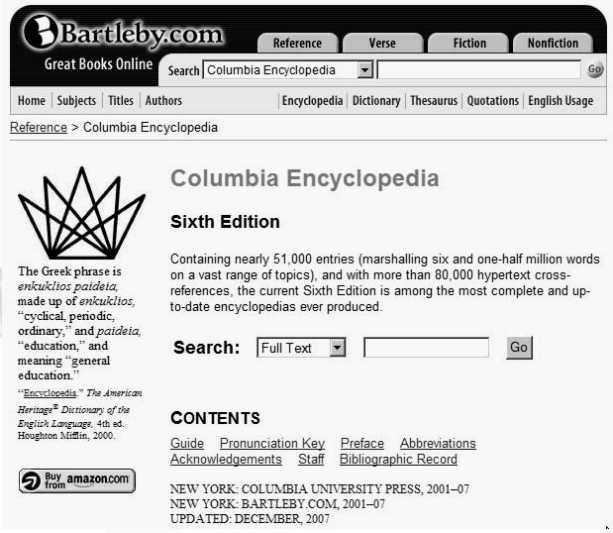
Posetioci koji žele obrazovne informacije o tačno određenoj temi ili pojmu mogu smatrati da su strategije dizajna prethodne dve vrste aplikacija suviše restriktivne i linearne. Tipičan akademski korisnik ovakvih sajtova uobičajeno ima veliko predznanje u datoj temi. S druge strane Internet je idelana medijum za "just in time" trening, gde se korisnik usredsredi na određenu specifičnu temu koja ga zanima. Fleksibilni, interaktivni, nelinerni dizajn je idealan za ovakve korisnike, zato što je teško predvideti koja će ih tema u određenom trenutku najviše zanimati. Dizajn mora omogućiti brz pristup velikom opsegu tema i obično mora obilovati linkovima na povezane materijale, kako na lokalnom sajtu, tako i na celom Web-u. Tekst bazirana lista linkova u ovom slučaju je efikasna za pregled sadržaja i osnovnih pojmova, jer se učitava brzo i prikazuje dosta informacija, ali su potrebne i dobro dizajnirane slike i ilustracije da bi se zainteresovali sami korisnici. Vreme pristupa korisnika se ne može predvideti, ali je uobičajeno da je kraće od vremena za prethodne dve grupe aplikacija, jer su ovi korisnici bez vremenskih rokova i pritiska. I u ovom slučaju je opcija za jednostavnu štampu skoro obavezujuća.



info.med.yale.edu/intmed/cardio/imaging

Reference

Najbolje dizajnirani sajtovi sa referencama dozvoljavaju korisnicima da posete sajt, pronađu šta žele i jednostavo odštampaju ili učitaju to što su pronašli. U ovakvim slučajevima nema glavne obrađene teme i njenih pojedniosti, pa je korišćenje nelinearno. Nevažne slike i ilustracije bi trebale da budu minimalne i odvojene od samog konteksta. Kontekst stranice i meni struktura bi trebalo da bude pažljivo strukturirana da omogući brzu pretragu, jednostavan download dokumenata i opciju za štampu. Dobro osmišljena opcija za pretragu je obavezna za sajtova sa više od trideset stranica ili za sajtove sa stranicama koje obuhvataju dugačke tekstualne dokumente u pojedinačnoj stranici. Trebalo bi razmisliti i o realizaciji efikasnije više parametarske pretrage umesto pretrage zasnovane na pojedinačnoj reči. Kod ovakvih sajtova vreme zadržavanja korisnika je po principu „the shorter the better“.



www.bartleby.com

Opšti i magazinski sajtovi

Za razliku od prethodnih grupa, opšti sajtovi su okrenuti ka korisnicima čija je motivacija mnogo manje fokusirana. Ovi korisnici moraju biti odmah privučeni kompletnom grafičkom i tekstualnom prezentacijom, ili će otići na neki drugi sličan sajt. Kao i drugim vrstama dizajna šta će privući korisnika zavisi od njegovih očekivanja. Na primer uticaj političkih komentara i kritike socijalnog stanja umnogome zavise od izdvojenih naslova i glavnih teza tekstova. Dobro dizajnirana naslovna stranica prezentacije je u stilu političkih magazina, jer obuhvata istu vrstu korisnika.



www.slate.com

Na žalost, sam sadržaj prezentacije je konstantno opterećen banerima koji se pomeraju, nestaju, sijaju i otežavaju čitanje sa samog ekrana.

Sajtovi sa vestima

Novinski sajtovi su u velikoj meri prilagodili svoj dizajn sa papira na manje formate na Internetu. Tako su na primer na sajtu *New York Times* iskorišćene sve osobine papirnog izdanja, ali uz korišćenje više boja i manje konzervativnih shvatanja i principa.

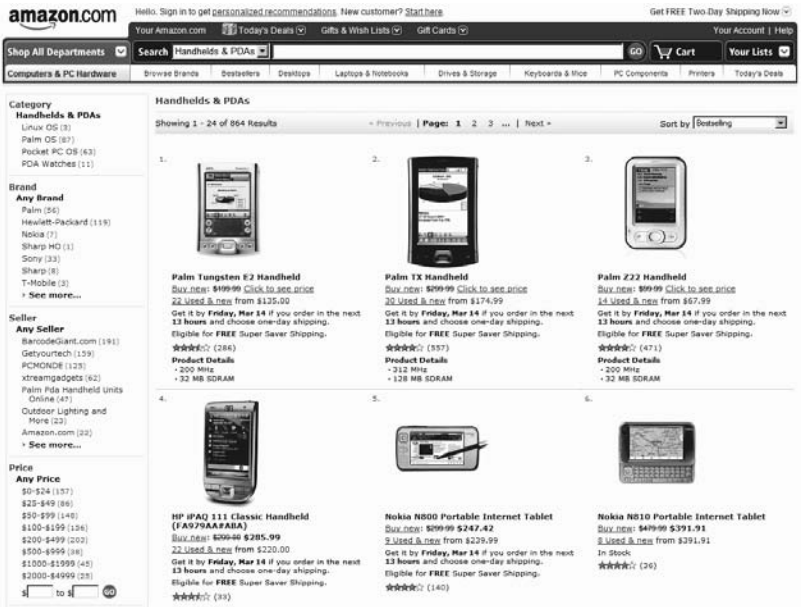


www.politika.co.yu

Elektronsko poslovanje

Kod sajtova sa elektronskim poslovanjem najvažniji parametri dizajna su efikasna navigacija i pretraga, i brzina kojom korisnik stiže do dugmeta za naručivanje. Tokom naglog razvoja ovog tržišta mnogi sajtovi za elektronsko poslovanje su potrošili ogromne količine para za razvoj Macromedia Flash ili digitalnih video prezentacija i pogrešili — mnogi su doživeli i bankrot pre nego što su sajtovi i počeli da se koriste. S druge strane, najuspešniji Internet komercijalni sajtovi su koristili i koriste tehnički osnovne i jednostavne stvari. Amazon, eBay, Yahoo!, i ostali uspešni sajtovi koriste upadljive jednostavne šeme za dizajn stranice i jednostavne tekst ili tab bazirane navigacione sisteme. Još jedna oblast gde su se sajtovi za elektronsko poslovanje pokazali kao neuspešni je realizacija alata za pretragu koji su dovoljno inteligentni da ponude odgovarajuće odgovore kada za neki upit nema odgovarajućih rezultata. Na primer, ako korisnik otkuca "PDA" neodgovarajući pristup je da se prikažu svi proizvodi

koji imaju ovu reč u svom imenu ili opisu, već je efikasno da se prikaže lista sortirana po odgovarajućim kriterijumima.



www.amazon.com

Amazon je eksperimentisao sa različitim tab sistemima kako je sajt rastao, ali se pokazalo da je izbor tabova kao navigacionog sistema mudar — oni predstavljaju jedan od nekoliko grafičko navigacionih metafora koje se preslikavaju na ekran. Ali tabovi funkcionišu ako postoji oko 8 ili manje opcija. Veći broj dovodi do konfuzije.

Amazon stranice za naručivanje predstavljaju takođe specijalan navigacioni dizajn za sajtove sa elektronskim poslovanjem. Većina dobro dizajniranih ekrana za naručivanje su kratki i obrađuju jednu temu (Mogućnosti u okviru opcije "shopping cart," su adresa korisnika, informacije o kreditnoj kartici, itd) na prozorima koji ne zahtevaju scroll bar da bi se pregledao sadržaj. Ali ovakav pristup dovodi do toga da se sam proces naručivanja odvija pomoću više ekrana, što može postati zamorno, ako ne postoje obaveštenja o trenutnom statusu (na primer ikonice na vrhu stranice tipa "you are here"). Pravila su jasna: svaki proces elektronskog poslovanja je jednoličan za samog kupca. Zato informacija o trenutnoj poziciji korisnika je korisna, jer serija ekrana postaje manji problem za korisnika koji zna šta ga dalje očekuje.

2.3 Elementi sajta

Web sajtovi se veoma razlikuju u primenjenom stilu, sadržaju, organizaciji i upotrebi, ali se može pronaći nekoliko zajedničkih karakteristika za sve sajtove čiji je cilj prikaz određenih informacija:

- Home page
- Meniji
- Lista resursa,"other related sites" stranice
- Uputstvo
- "What's new?" stranice
- Opcije pretrage
- Informacije o kontaktu i korisnički utisci
- Bibliografija i dodaci
- Strane sa najčešće postavljenim pitanjima
- Obrada korisničkih grešaka

Home page

Svi Web sajtovi su organizovani oko početne stranice, koja predstavlja i logički ulaz u sistem stranica. U hijerarhijskoj organizaciji, home page se nalazi na vrhu hijerarhije i sve stranice u okviru sajta sadrže direktan link povratka na ovu stranicu. Smatra se da je površina pravougaonika širine oko 30-ak inča na vrhu početne stranice najviše primećena površina od strane korisnika. Zato se u dizajnu stranice efikasnom navigacijom smatraju linkovi na vrhu stranice – ne postoji bolji način korisniku ponuditi opcije koje su dostupne na preostalom delu sajta.

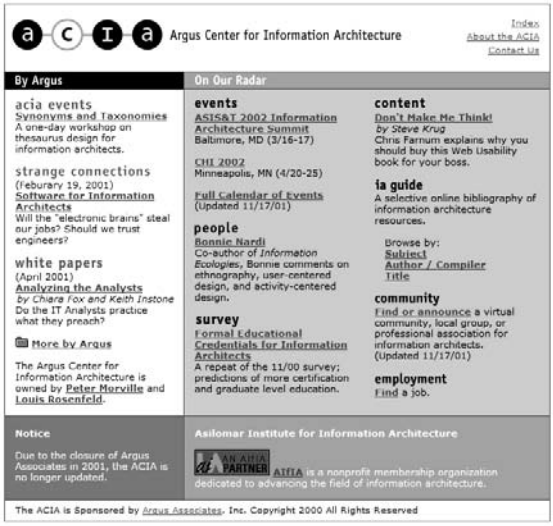


www.singidunum.ac.yu

Početna stranica može imati više funkcija. Neki dizajneri koriste prednost ove stranice kao najposećenije stranice sajta, pa je smatraju idealnom za postavljanje novosti i novih informacija. Takođe, velika posećenost ove stranice je stvara idealnim mestom za linkove ili smeštanje opisa sadržaja sajta. Ostale mogućnosti je da se ova stranica iskoristi kao prva mogućnost da se privuče pažnja korisnikana neku određenu podtemu ili specijalnu oblast prikazanu na sajtu. Najčešće korišćene strategije realizacije početne stranice su:

Početna stranica u formi menija

Lista linkova u formi menija je dominirala u dizajnu početnih stranica prvih nekoliko godina razvoja Web-a, i trenutno je najčešće korišćeni tip početnih stranica. Ovakva organizacija stranica zahteva korišćenje ne samo obične liste tekst baziranih HTML linkova, već korišćenje i grafičkih slika, koje su i efikasnije, jer se dati prostor može maksimalno iskoristiti. Efektan dizajn kombinuje slike i blokove tekst baziranih linkova. Tekst linkovi su manje vizuelno uočljivi, ali je jednostanije održavanje samog koda.



argus-acia.com

Početne stranice bazirane na vestima

Početne stranice ovakve organizacije, kao što su sajtovi New York Times i CNN su tipični primeri, ali mnoge organizacije koriste prednosti njihove velike posećenosti radi obaveštavanja kako samih svojih zaposlenih, tako i šireg kruga korisnika. Informacije uživo povećavaju atraktivnost sajta i ponovnu posetu korisnika. Mnogi dizajneri rezervišu jednu ili više površina za poslednje vesti, naredne događaje ili upozoravajuće poruke. Ako se izabere ovakav pristup potrebno je standardizovati lokaciju i prirodu površine za vesti sa generalnim okvirom dizajna stranice koji se neće menjati neko vreme. Korisnici će biti dezorijentisani, ako se početna stranica menja od nedelje do nedelje.



www.salon.com

Početne stranice bazirane na korisnicima

Veliki Web sajtovi pružaju mnogo informacija širokom opsegu korisnika, pa je nemoguće opisati širinu i dubinu takvog sajta na jednoj početnoj stranici. Takođe, i sami korisnici često posećuju Web sajt sa specifičnim interesom ili ciljom. U takvim slučajevima dobro je iskoristiti početnu stranicu da bi se odmah izvršila podela korisnika u interesne grupe i onda svakoj grupi ponuditi specifične informacije u okviru sledećih stranica.



www.iastate.edu

Splash stranice

Splash stranice su jedan od najosporavanijih elemenata sajta. Za mnoge korisnike one predstavljaju dodatak i nepoteban klik miša do sadržaja koji ih zanima. Takvi korisnici bi radije želeli da im se odmah prikaže struktura sajta i da krenu od te tačke, a ne od aplash stranica sa lepom grafikom ili interesantnim animacijama. Zato je ključna stvar upoznavanje sa krajnjim korisnicima i tek onda najprikladniji izbor ulaska na sajt.

Treba prepoznati funkciju sajta. Da li će tipični korisnici jednom posetiti sajt ili više puta? Tako da alati kao što su kalendar ili alat za pretragu nemaju samo estetsku ulogu, jer neki korisnici mogu posetiti sajt nekoliko puta dnevno. Elegantni, ali nefunkcionalni elementi tako mogu vrlo brzo postati zamorni. Naravno, korisnici koji ne žele da pristupaju sajtu preko slash stranice, mogu zapamtiti u okviru čitača narednu stranicu sa sadržajem i pristupati sajtu preko te stranice. Ali ako dizajner koristi ovakav razlog za korišćenje splash stranice, možda je bolje da je ukloni sa prezentacije.

Uspeh splash stranice umnogome zavisi od očekivanja korisnika sajta. Postoje i korisnici koje zanimaju misteriozni, enigmatski ili estetski ugođaji, koji će uživati u efektno dizajniranim splash stranicama.



www.loc.gov/thomas

Pri realizaciji početne stranice mogu se kombinovati različiti dizajni početne stranice i na taj način napraviti mešavinu sve 4 strategije. Prethodni primer The Library of Congress's THOMAS kombinuje slike, listu meni opcija i specijalne delove sajta

Slika ili tekst?

Osnovna layout odluka pri realizaciji početne stranice je da li koristiti vizuelne elemente. Mnoge kompanije, institucije i edukacione organizacije koriste samo male grafičke banere na vrhu početne stranice, dok je kod komercijalnih sajtova trend porasta upotrebe kompleksnih kombinacija linkova ugrađenih u vizuelne elemente. Iako efektni vizuelni elementi mogu biti efikasni u privlačenju korisnikove pažnje, veliki grafički meniji podrazumevaju i veće vreme učitavanja stranice, posebno za korisnike koji su povezani putem modema ili mreže slabe propusne moći. Čak, iako su korisnici povezani pomoću bržih veza, grafički meniji i dalje se prikazuju sporije nego tekst bazirana lista linkova.
Ova nesimetrija između sporog učitavanja, ali atraktivnog prikaza stranica baziranih na vizuelnim elementima i jednostavnih tekst baziranih stranica se reflektuje i na različite potrebe različite vrste korisnika. Ciljevi većine Web sajtova je da prenesu neke interne informacije (studentima, zaposlenima, ili klijentima) i da komuniciraju sa potencijalnim klijentima i proizvoljnim Web korisnicima. Prime The Guggenheim Museum je izbor grafičkog dizajna početne stranice, ali je layout pažljivo dizajniran da odgovara dimenzijama prosečnog monitora. Zbog ove odluke stranica se učitava relativno brzo za primer grafičkog menija



www.guggenheim.org

Relativno jednostavna, većinom tekst bazirana početna stranica za World Wide Web Consortium korisnicima pruža efikasan odnos linkova po KB veličine stranice, ali je dobijen siromašan vizuelni izgled. Stranica se brzo učitava i dobro je dizajnirana za Web specijaliste, ali nije atraktivna za običnog korisnika



www.w3.org

Najbolji način da se obuhvate i efektan izgled i potrebe običnog korisnika je da se pruže alternativni pogledi na ist sajt. Jedan pristup je da se realizuje vizuelno atraktivna početna stranica sa ciljem da se privuče pažnja većine korisnika, ali i da se realizuje više tekst orijentisana alternativna stranica koja bi omogućila brz pristup podacima putem detaljnih tekstualnih menija. Drugi pristup je da se koristi grafički baner na vrhu stranice, praćen skupom tekst baziranih linkova.

Neki korisnici koji pristupaju Internetu preko modema biraju da ne učitavaju slike, pa se može desiti da ne vide stavke menija koje su realizovane putem slika. Ako su u okviru stranice realizovana ovakva vrsta menija potrebno je obezbediti i alternativne tekst bazirane linkove koji će biti vidljivi i ovoj vrsti korisnika. Mnogi sajtovi realizuju tekst bazirane linkove smanjene veličine iznad footer-a, gde ne utiču mnogo na ostali deo dizajna



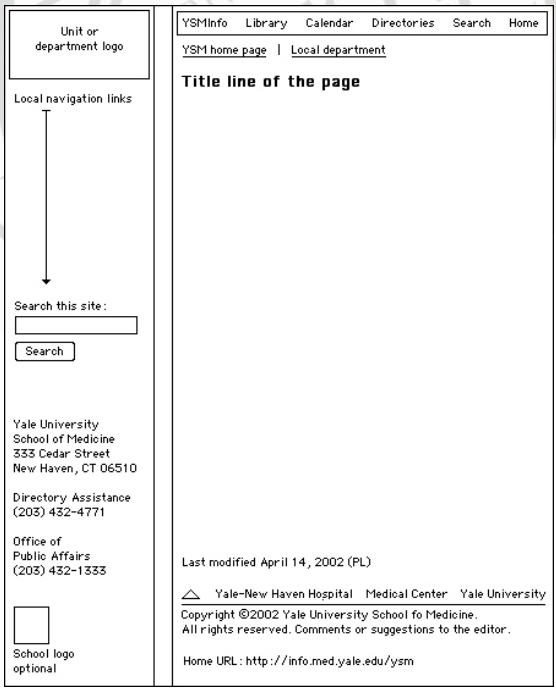
www.ynhh.org

Obrazac stranica

Uobičajeno je da se ka početnoj stranici usmeri najviše pažnje u početku dizajniranja Web sajta. Ova stranica jeste bitna, ali to je samo jedna stranica sajta, i ne treba dopustiti da njen dizajn dominira nad strategijom celog sajta. Kada se realizuje veliki kompleksan Web sajt, mnogo je bitnije koncetrisati se na standardan layout ostalih unutrašnjih stranica u okviru sajta, jer početna stranica je samo jedna, a preostalih može biti i na stotine. Ako se napravi greška na početnoj stranici, to je greška samo na jednoj stranici, ali greška na obrascu ostalih strancia, je greška na stotinama unutrašnjih stranica. Zato uspeh samog sajta zavisi od preciznog logičkog obrasca koji se primenjuje.

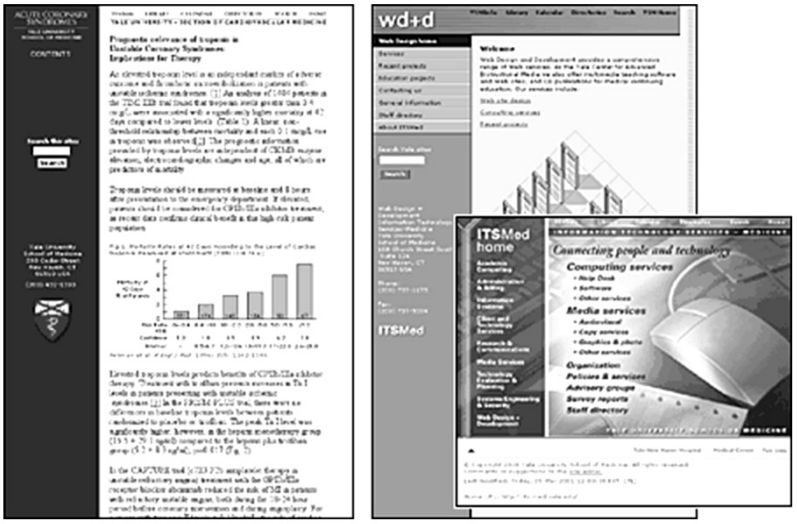
Realizacija layout-a obrasca obično počinje analiziranjem strukture sadržaja i odlukom šta je potrebno za relaizaciju osnovne navigacije. Odlučuje se koji će se linkovi prikazati i kako će se koristiti na svakoj stranici sajta. Ako se realizuje sajt veće kompanije ili dela vlade ili neke neprofitabilne organizacije mora se proučiti kako će se sajt uklopiti u širi kontekst samih klijenata. Da li klijent već ima postavljen i uspešan format Web dizajna u koji se treba uklopiti? Ako je odgovor negativan, treba pažljivo odabrati koji linkovi, logo oznake, i drugi grafički i funkcionalni elementi određuju mesto u većoj hijerarhiji. Cilj je ustanoviti logički i konzistentan pristup gde i kako će se osnovni grafički elementi identiteta, navigacioni linkovi i ostale najbitnije informacije, pojaviti na svim stranicama u okviru sajtova. Takođe, terminologija je veoma bitna: treba izabrati odgovarajuće reči za linkove i naslove. Dvosmislene ili konfuzne labele i fraze mogu dovesti do pogrešnog rada.

Obrazac stranice utvrđuje broj, lokaciju i terminologiju svih glavnih linkova i grafičkih elemenata na stranici:



info.med.yale.edu/ysm

Obrazac je način i da se različiti vizuelni pristupi primene na identičnom navigacionom mehanizmu. U primer, sve tri stranice, iako koriste različite boje i grafičke šeme, koriste isti obrazac koji definiše poziciju i termine svih glavnih navigacionih linkova i ostalih elemenata stranice:



info.med.yale.edu, its.med.yale.edu/wdd, its.med.yale.edu

Nakon definisanja obrasca stranica može se realizovati i početna stranica na osnovu istog obrasca, tako da se korisnici koji pristupaju sajtu odmah upoznaju sa primenjenim obrascem, vizuelnim identitom, i navigacionom šemom koja je upotrebljena na kompletnom sajtu. Treba naglasiti da napravljeni obrazac ne treba da bude konačan i za početnu stranicu, ona treba da poseduje veći broj navigacionih elemenata i samog sadržaja nego obične unutrašnje stranice. Iz ovog razloga početne stranice su obično šire i sa mnogo više grafike od ostalih stranica:

Meniji

Često je potrebno realizovati veći broj podmeni stranica kojima će se pristupati iz generalne liste kategorija sa početne stranice. Kod kompleksnijih sajtova sa više glavnih tema nije praktično zatrpati početnu stranicu sa desetinama linkova – povećava se vreme učitavanja stranice, kao i sama kompleksnost što može odbiti same korisnike. Realizacija podmeni stranica za svaku temu dovodi do kreiranja mini-home page za svaku temu sajta. Za specijalizovane, detaljne podmenije, česti korisnici se mogu uputiti da im automatski pristupaju (www.b92.net/sport). Na ovaj način podmeniji postaju alternativne početne stranice orijentisani ka specifičnim korisnicima. Svakako treba uključiti osnovni skup linkova na ostale delove sajta na svakoj stranici podmenija, i uvek uključiti i link ka glavnoj početnoj stranici.

Lista resursa,"other related sites" stranice

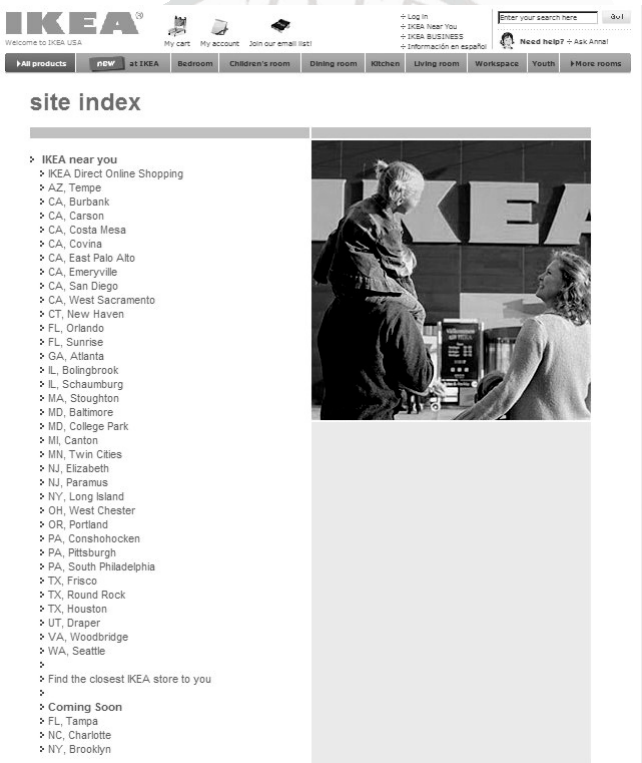
Razvoj Interneta je ogroman, da čak i veliki komercijalni Web indeks servisi, kao što su Yahoo! i Google ne mogu skupiti sve informacije koje postoje na Web-u. Kada se počne realizacija sajta, početna stranica je obično kolekcija najvažnijih linkova ka sajtovima povezanim preko profesije, industrije ili nekog drugog interesa. Kod sajtova organizacija ili institucija, dobro organizovana stranica tipa "Other useful sites" može biti najvredniji i najčešće korišćeni resurs od strane korisnika.

Uputstvo

Za razliku od štampanih medija, gde se na primer prema veličini knjige ili magazina može oceniti koja količina informacija se sadrži unutra, Web sajtovi daju nekoliko eksplicitnih informacija o dubini prezentovanja obrađenog sadržaja. Ove informacije su posebno bitne, ako početna stranica ne sadrži detaljnu listu unutrašnjih linkova. Takođe i ako opcije pretrage omogućavaju korisniku brz pristup traženim informacijama, one ne mogu zameniti definisan, dobro organizovan opis sadržaja samog sajta. Razlog tome je što je i mehanizam najboljih opcija za pretragu relativno neinteligentan i samo preko nekoliko određenih parametara pruža informaciju o prioritetu, relevantnosti i međusobnim relacijama, informacionih resursa na sajtu.

Sadržaj ili indeksi sajta

Sadržaj i indeksi najbitnijih informacija na sajtu su jednostavan i efikasan način da se korisniku predstave namere, organizacija i sam sadržaja sajta. Sadržaj i indeks pojmova su konvencije iz štampanih medija, korisnici ih razumeju i podržavaju preglede, perspektive i efikasnu navigaciju koju nude. Glavna razlika između Web-baziranih indeksa i štampanih je da Web indeksi zahtevaju da ne budu složeni i detaljni kao oni iz knjiga, jer korisnik pomoću alata za pretragu može da nađe referencu za svaku nepoznatu reč. Indeksi na Web sajtu treba da ukažu na najvažnija i najkorisnija pojavljivanja definisane reči i da zanemare nevažne reference.



www.ikea-usa.com

Mapa sajta

Mapa sajta pruža korisniku pregled sadržaja sajta, i mogu se organizovati na dva načina: grafički dijagrami koji opisuju sadržaj ili organizovana lista linkova na glavne delove sajta.
Grafička forma mape sajta se razlikuje od hijerarhijskih dijagrama sa grananjem do geografskih metafora, ali imaju iste limite:

- Grafičke mape sadržaja velikih i kompleksnih sajtova su u najboljem slučaju metafore koje opisuju samo gflavne teme sadržaja sajta. Sam ekran je limitiranog prostora, pa grafička prezentacija teži da pojednostavi i zanemari hijerarhiju informacija. Rezultat je potrošeno vreme na realizaciju grafičkih elemenata i prikaz samo osnove sajta. Sve dok se na sajtu koriste informacije koje nisu vizuelne (skup mapa na primer), tekst-bazirani sadržaji ili indeksi su efikasniji i sa više informacija.
- Grafičke mape sajta su u većini slučajeva kompleksni grafički fajlovi, koji manje trpe eventualne izmene.
- Fajlovi za grafičke mape sajta su veliki i sporije se učitavaju.

Mape sajta zasnovane na brižljivo organizovanim tekst linkovima glavnih meni i podmeni stranica sa naslovima glavnih stranica su sa mnogo više informacija i sa jednostavnijim načinima promene sadržaja. I najefikasniji tekstualne mape sajta su samo proširene verzije sadržaja. Takve elemente, korisnici bolje razumeju i koriste.



www.bitstream.com

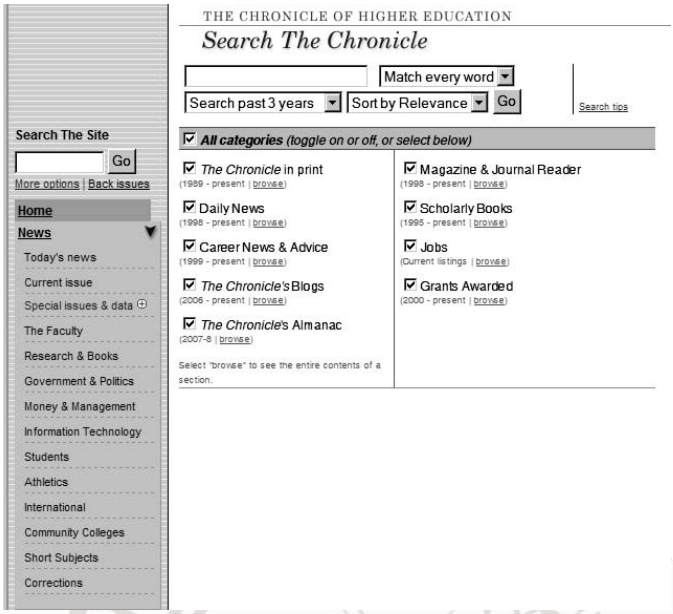
"What's new?" stranice

Mnogi sajtovi treba da se menjaju često i informacije koje se prikazuju nisu stalne. Ali prisustvo novih informacija korisnik ne mora da uoči, pa mu sistematski treba usmeravati pažnju na njih. Na primer pored novih informacija treba postaviti grafički element "new". Ali ako je sajt veoma kompleksan sa nekoliko desetina ili stotina stranica, treba dizajnirati posebnu "What's New" stranicu koja će informisati korisnika o novim informacijama postavljenim na sajt.
Svaka stranica u okviru organizacije ili institucije treba da ima i datum poslednje promene informacija na stranici, pa korisnik može uporediti svoj poslednji pristup sa datumom postavljanja informacija. Mnogi korisnici štampaju stranice sa Weba. Bez datuma promene, korisnici ne mogu znati da li je odštampani materijal trenutni ili zastareli.

Opcije pretrage

Opcije pretrage su neophodne za velike sajtove i male sajtove sa dugačkim dokumentima. Sajtovi koji se menjaju često takođe zahtevaju dobar alat za pretragu, jer je verovatno da meniji i indeksi sajta neće sadržavati svaku promenu na samom sadržaju sajta. Ali alati za pretragu nisu zamena za pažljivo organizovanu strukturu menija i podmenija. Dva sistema, rad sa menijem i pretraga po ključnoj reči dopunjuju jedan drugog – nijedan sistem sam za sebe nije dovoljan. Pretraga po reči daje korisniku linkove koje treba da prati, ali korisnik nema osećaj kako su organizovane informacije. Meniji i sadržaji su odlični za generalni pregled, ali korisnik koji traži specifičnu informaciju ne zanima se za celokupan sadržaj.

Softver koji realizuje pretragu definiše i relaizaciju dizajna korisničkog interfejsa za ovu opciju. Ako se sadržaj menja često treba omogućiti da se pretraga vrši i po poslednjim dodatim informacijama. Takođe, korisnik treba da razume koji se sadržaj pretražuje: ceo sajt ili određeni deo? Ako je sajt kompleksan podesno je ponudtiti glavne oblasti sa sajta, pa da korisnik ima mogućnost da limitira proces pretraga. Takođe, stranica sa rezultatom treb da bude u skladu sa grafičkim dizajnom sajta.



www.chronicle.com

Informacije o kontaktu i korisnički utisci

Internet je bidirekcionalni medijum – korisnici očekuju da mogu da pošalju svoje komentare, pitanja, ili sugestije. Uvek treba definisati najmanje jedan link sa email adresom na određenu lokaciju u okviru sajta. Takođe, mogu se koristiti stranice u okviru formi za dobijanje korisničkih informacija i utisaka, ove informacije se mogu zapisivati u bazu podataka i kasnije analizirati.

Logistička podrška ovoj opciji može biti veoma otežana, pa čak imati težinu stvaranja novog dela samog sajta. Ko će odgovarati na pitanja o organizaciji, proizvodima i servisu? Ko će prikupljati i analizirati dobijene informacije? Ko će se zahvaljivati korisnicima na sugestijama? Pre nego što se doda ova funkcionalnost, treba stvoriti infrastrukturu koja ć je i podržati.

Adresa, brojevi telefona, brojevi faksa

Veliki je broj slučajeva da dizajneri sajta zaborave da se ne obavlja sva komunikacija sa korisnikom samo preko sajta. Čak i ako je dizajn sajta odličan, korisnik i dalje želi da okrene telefon, pošalje pismo ili paket, pošalje dokumente faksom. Početna stranica treba da obuhvati iste informacije za kontakt koje su date i u reklamnim materijalima ili karticama. Ako na početnoj stranici nema dovoljno prostora za ove informacije, treba realizovati posebnu stranicu koja će sadržavati informacije oko kontakta

Mape, opisi putovanja, informacije o parkingu

Web sajt je idealno mesto za pružanje informacija oko putovanja korisnicima, posetiocima, sponzorima – svakome ko hoće da poseti klijenta. Vizuelne mape, tekst bazitrana uputstva, informacije o lokalnim hotelima, i čak plan zgrade može mnogo pomoći samom korisniku.



www.montereybayaquarium.org

Bibliografija i dodaci

Koncept dokumenata u elektronskomokruženju kao što je Web, je fleksibilno, pa ekonomski i logistički aspekti digitalnog izdavanja omogućavaju pružanje informacija bez troškova vezanih za štampanje samih dokumenata. Izveštaj za kolege na papiru obično podrazumeva štampanje po jedne kopije za svakog, i praktično dovodi do toga da papirni izveštaji budu koncizni i sa limitiranim materijalom za podršku. Bibliografija, dodaci i apendiksi se zato umesto u osnovni dokument, mogu postaviti na Web sajt i na taj način dostupni svakome ko želi da ih pogleda.

Strane sa najčešće postavljenim pitanjima

Web i ostali Internet bazirani sistemi imaju posebno relaizovanu opciju - FAQ ili "frequently asked questions" stranicu, gde su prikazana najčešće postavljena pitanja od strane korisnika sa datim odgovorima. FAQ strane su idealne za sajtove koji su dizajnirani za potrebe podrške ili informacija određenoj radnoj grupi korisnika. Mnoga pitanja od novih korisnika su u prošlosti često postavljena i na njih su davani odgovori. Dobro dizajnirana FAQ stranica može povećati korisnikovo razumevanje informacija i ponuđenog servisa, pa smanjiti pritisak na tim za podršku.



www.palm.com

Obrada korisničkih grešaka

Mnogi korisnici su upoznati sa stranicom tipa "404 error, file not found" koja se pojavljuje kada Web server ne može da pronađe traženu stranicu. Dokument može da nedostaje, jer ga je autor obrisao ili pomerio, ili je korisnik ukucao pogrešnu adresu. Odlika dobro osmišljenih sajtova je korisnički dizajnirana i korisna stranica serverske poruke o greškama. Većina standardnih ekrana za obradu greške je ružno, generičko i neinformativno. Dobro dizajnirana ova vrsta stranica mora da bubude konzistentna sa ostatkom sajta. Stranica treba da ponudi objašnjenja za grešku, ponudi alternative i omogući link na lokalnu početnu stranicu, indeks sajta ili stranicu za pretragu

Na kraju može se zaključiti da tehnologija i vizuelni elementi predstavljaju osnovu Web dizajna; obe komponente su neophodne i moraju da budu u direktnoj vezi sa svrhom sajta. Prvo treba razmisliti o nameni sajta da bi se lakše utvrdili ciljevi i zadaci sajta i kako da se oni ostvare. Pre nego što se započne sa gradnjom potrebno je da se pažljivo definiše problem koji sajt treba da reši ili cilj koji treba da postigne. Kada se jasno definiše namena sajta, dobija se odgovor na pitanje da li je sajt uopšte vredno praviti. Pošto se utvrde ciljevi ili zadaci koje sajt treba da ispuni, treba izraditi specifikaciju. U nju treba uključiti sve pojedinačne zahteve koje sajt treba da ispuni, kao i pažljivo razmotrene potrebe korisnika. Posle toga dolazi na red dizajn sajta, što podrazumeva izradu tehničkog i/ili vizuelnog prototipa. Kada se to obavi, treba realizovati i isprobati sajt. Posle toga treba objediniti sve delove sajta, uključujući i sadržaj, i ponovo proveriti da li sve pravilno funkcioniše. Tek tada se sajt može objaviti, pri čemu uvek treba da ostane „otvoren“ kako bi mogao da se popravi i dopuni na osnovu mišljenja korisnika.

2.4 Pravila efektnog dizajna

Postoji više načina da se definišu principi razvoja efektnog dizajna. Najpoznatija su Nielsen-ova pravila. Jacob Nielsen, „usability guru“ kako ga neki oslovljavaju, jedan je od najzaslužnijih za Web kakav danas postoji. On je 1996. godine definisao deset osnovnih pravila ili se može reći grešaka Web dizajna .

1. Korišćenje okvira. Korisnicima deluje vrlo zbunjujuće podela stranice na okvire (engl. frame), budući da okviri razbijaju osnovni korisnički model Web stranice – onemogućeno je pohranjivanje adrese tekuće stranice u obliku „bookmarka“, kao i ponovno njezino učitavanje, URL adrese prestaju da rade, a ispisivanje postaje teško.
2. Bezrazložno korišćenje najnovije tehnologije. Nikada ne privlačiti korisnike na lokaciju hvalom upotrebe poslednje Web tehnologije. Većina korisnika je ipak više zainteresovana za koristan sadržaj stranice, kao i za sposobnost pružanja dobrog korisničkog servisa od strane dizajnera.
3. „Skroliranje“ teksta, „marquees“ i kontiunuirane animacije. Nikada ne koristiti/ne uključivati elemente stranice koji se neprekidno pomeraju. Pokretne slike imaju previše snažan efekat na ljudsku percepciju. Takođe treba istaknuti da je treperenje određenog teksta izrazito loše.
4. Složene URL adrese. Adresiranje na nivou mašine, kao kod adresa mrežnih resursa, URL adresa, nikada ne bi smelo biti vidljivo korisniku. Međutim, s obzirom na to da se ipak nalazi, treba ga učiniti što suvislijim. Pokazalo se da korisnici nastoje dekodirati URL adrese u cilju shvatanja strukture same Web lokacije. Stoga adresa mrežnog resursa treba sadržavati imena čoveku čitljivih foldera i datoteka, koje odražavaju prirodu informacionog prostora. Takođe treba voditi računa o korišćenju kratkih imena napisanih malim slovima, po mogućnosti bez korišćenja specijalnih znakova i slično.
5. Stranice „siročići“. Treba voditi računa o tome da sve stranice imaju jasnu indikaciju kojoj Web stranici pripadaju, budući da korisnici mogu konkretnoj stranici pristupiti i direktno, bez prolaska kroz glavnu / inicijalnu / početnu stranicu (engl. home page). Najbolje je da svaka stranica ima vezu sa glavnom stranicom, kao i određenu

indikaciju/informaciju koja upućuje na mesto stranice unutar strukture informacionog prostora.

6. Stranice koje zahtevaju dugo „skroliranje“. Istraživanja sprovedena sredinom 90-tih godina pokazuju da samo oko 10% korisnika „skrolira“ ispod informacije/sadržaja koji je vidljiv na stranici u trenutku kada se ona učita. Zbog toga sav ključan sadržaj, kao i postojeće opcije za navigaciju moraju biti u gornjem delu stranice. Kasnija istraživanja pokazuju da korisnici ipak učestalije pribegavaju „skroliranju“ nego prvih godina Web-a.
7. Nedostatak podrške navigaciji. Ne treba se zanositi činjenicom da korisnici o određenoj Web lokaciji znaju koliko i sam njegov dizajner. Oni uvek imaju poteškoća prilikom pronalaženja informacija, stoga im je uvek neophodna podrška pružena na način koji će im omogućiti dobro poimanje strukture stranice i njenog mesta unutar informacionog prostora. Budući skoro uvek čak i najbolja podrška navigaciji naprosto nije dovoljna, osigurati i dobru mogućnost pretraživanja.
8. Nestandardne boje veza. Veze prema stranicama koje korisnik još nije posetio plave su boje, dok su veze prema prethodno viđenim stranicama crvene ili ljubičaste boje. Nikako ne mešati navedene boje, budući je sposobnost shvatanja veza koje se slede jedno od malobrojnih navigacionih pomagala koje predstavlja standard u većini Web browser-a.
9. Neažurirane informacije. Većina dizajnera više voli trošiti vreme na stvaranje novog sadržaja, nego na održavanje postojećeg. Međutim, praksa pokazuje da je održavanje postojećeg/starog sadržaja jeftin način osnaživanja sadržaja Web lokacije, budući je većina starih stranica i dalje relevantna, te zbog toga naprosto treba biti povezana s novim stranicama. Naravno, i tu postoje iznimke.
10. Doskočiti dugom vremenu učitavanja stranice. Kao najduže vreme učitavanja pre nego što korisnici izgube interes, kada je reč o Web-u, navodi se 15 sekundi potrebnih za učitavanje sadržaja nove stranice.

Ovih 10 najvećih grešaka Web dizajna i dalje umanjuju upotrebu Web-a, i danas se mogu pronaći na velikom broju Web lokacija. Međutim, nova Web tehnologija kao i nove Web aplikacije uvode jednu potpuno novu klasu grešaka koju je 1999. godine takođe identifikovao J. Nielsen.

Nefunkcionisanje BACK dugmeta. Back dugme predstavlja „liniju života“ Web korisnika, te drugo po redu najčešće korišćeno navigaciono sredstvo (posle aktiviranja link veza). Korisnici nesmetano mogu pokušati bilo što

unutar Web-a, ali se uvek mogu spasiti izborom Back dugmeta koje će im omogućiti povratak na poznato područje.

Otvaranje novih prozora u browser-u. Nikako ne otvarati nove prozore na novi zahtev. Ako korisnik želi novi prozor, on će ga sam otvoriti! Dizajneri pravdaju otvaranje novih prozora teorijom da time zadržavaju korisnike na svojoj lokaciji. Ipak, na taj način se onemogućava funkcionisanje back dugmeta koje predstavlja uobičajeni/normalni način na koji se korisnici vraćaju na prethodne lokacije. Činjenica je da sami korisnici često ne uoče otvaranje novog prozora, te zbog toga ostaju zbunjeni.

Nestandardno korišćenje grafičkog korisničkog interfejsa. Konzistentnost je jedan od najsnažnijih principa upotrebljivosti – kada se stvari uvek ponašaju na isti način, korisnici se ne zamaraju time što će se dogoditi. Umesto toga, znaju šta će se dogoditi na temelju prethodnog iskustva. Npr. uobičajeni rezultat biranja određene link veze je naprosto zamena sadržaja originalne stranice sadržajem odredišne stranice, i to u istom prozoru browser-a. Ili npr., pravila za korišćenje radio dugmadi govore da se ona koriste za odabir jedne od postojećih opcija, koja će se aktivirati tek kada korisnik potvrdi svoj izbor biranjem „OK“ dugmeta. Na žalost, dosta Web lokacija koristi radio dugmad kao dugmad za akciju, koja imaju neposredni učinak nakon njihovog aktiviranja.

Nedostatak biografija. Korisnici vole znati ko se krije iza informacija na Web-u. Generalno, biografije i fotografije autora daju Web-u osobnost, te povećavaju korisnikovo poverenje. Lokacije treba da imaju vezu prema autorovoj biografiji, kao što takođe treba da osiguraju i način kako korisnici mogu pronaći na primer, i ostale članke istog autora. Posebno je neprikladno kada veza, umesto prema autorovoj biografiji, sadrži vezu za slanje e-mail-a autoru.

Nedostatak arhive. Stara informacija je često ujedno i dobra informacija koja može biti korisna čitaocima. Procenjuje se da postojanje arhive povećava troškove vođenja lokacije za oko 10%, ali povećava njegovu korisnost za oko 50%.

Pomeranje stranica na nove URL-ove. Uvek kada se stranica pomera na novu URL adresu, kidaju se dolazeće veze sa drugih lokacija.

Naslovi koji su besmisleni izvan konteksta. Natpisi/naslovi kao i drugi mikrosadržaj za Web mora biti napisan sasvim drugačije nego za stare medije – radi se o stavkama koje predstavljaju elemente korisničkog interfejsa, te stoga moraju pomoći korisnicima pri navigaciji. Natpisi/naslovi obično nisu sastavni deo konteksta celokupne stranice, a koriste se u sadržaju, kao i u rezultatima pretraživanja. Stoga naslovi moraju

biti jednostavni i da jasno govore šta se nalazi s druge strane veze, te ujedno da zaštite korisnika od prelaska na stranicu, za čiji sadržaj uopšte nisu zainteresovani.
Usvajanje novog Internet „buzzword“-a. Većina Internet „buzzwords“-a (npr. chat, e-mail, aukcije i sl.) ipak ima neku svoju vrednost, te mogu doneti dobit nekim Web lokacijama koje ih znaju prikladno koristiti. Treba računati da će već sledećeg meseca na dnevni red doći neki novi „buzzword“.
Spor odziv browser-a. Nielsen-ova studija donosi podatak o čak 84% ispitanih lokacija koja krše pravilo vremena odziva. Korisnike uopšte nije briga zbog čega je vreme odziva sporo. Oni samo znaju da ta lokacija ne pruža dobru uslugu, a sporo vreme odziva direktno utiče na smanjenje nivoa poverenja i uvek kao rezultat ima smanjenje prometa, budući korisnici svoj posao naprosto odnose na neku drugu lokaciju. Očigledno treba investirati u brzi browser, preispitati arhitekturu samog sistema, te optimizovati vreme odziva.
Sve što iole liči na reklamu. Selektivna pažnja je vrlo moćna, a Web korisnici su naučili ne obraćati pažnju na sve ono što im stane na put prilikom njihove ciljne navigacije. Činjenica je da reklamiranje preko Web-a ipak ne funkcioniše. Stoga treba izbegavati svaki onaj dizajn koji liči na reklamu.
Kako Web raste, tako svaka godina donosi neke nove greške, a u 2002. godini po J. Nielsen-u neke od najgorih grešaka u Web dizajnu vezane su za lošu integraciju elektronske pošte. Korisnici su se počeli ponašati vrlo zaštitnički prema svojim e-mail adresama. Uvek kada se u okviru korisničkih testiranja na Web lokaciji od korisnika zatraži e-mail adresa, oni negativno reaguju. Dalje, korisnici mrze „skrolirati“ s leva na desno. S druge strane, vertikano skroliranje je u redu možda i zbog toga što je uobičajeno. Upotreba Web sajta znatno se smanjuje i nefleksibilnom mogućnošću pretraživanja, budući se ne podržavaju različite varijante upita, npr. množina, upisivanje crtice i sl.. Takođe, dugački URL-ovi nepovoljno utiču na društvenu navigaciju Web-a. Na primer, ukoliko je URL adresa predugačka da bi se prikazala u adresnom polju browser-a, mnogi korisnici neće znati kako da je selektuju tj. obeleže.
I ovde se može zaključiti da dizajn prezentacije treba udovoljiti korisnikovim očekivanjima. Ako se stvari uobičajeno ponašaju, to sami korisnici bolje razumeju mogućnosti koje im stoje na raspolaganju. Odgovornost Web dizajnera leži u davanju prioriteta određenim delovima

informacionog prostora, te isticanju vrlo malog broja objekata u cilju njihove preporuke korisnicima. Opsežne studije otkrile su da:

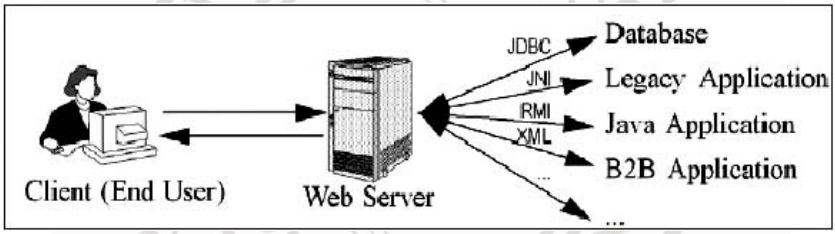
1. Korisnici imaju vrlo malo strpljenja za siromašno dizajnirane sajtove; dobar primer je izjava jednog od korisnika: „Što je stranica bolje organizovana, to ću imati više poverenja u informacije“,
2. Korisnici ne žele „skrolirati“: Informaciju koja se nalazi na dnu stranice u trenutku kada se stranica otvori, će pročitati isključivo vrlo zainteresovani korisnici
3. Korisnici ne žele čitati: Brzina čitanja sa monitora računara 25% je manja nego sa papira, ali to ipak ne znači da na stranicu treba staviti 25% manje teksta nego što bi bilo napisano na papiru; treba napisati 50% manje

UNIVERZITET PRIVATNI AKREDITOVANI UNIVERZITET SINGIDUNUM

3. JAVA SERVLETI

Servleti su Java komponente za rad sa serverima koji su orijentisani na komunikaciju tipa request/response, kao što su web serveri sa podrškom za Javu [slika 1]. Ove Java klase mogu da se izvršavaju na mnogim različitim serverima, jer servlet API, koji se koristi za pisanje servleta, ne koristi ništa iz serverskog okruženja ili protokola. Servleti su postali sastavni deo skoro svih HTTP servera koji na taj način podržavaju servlet tehnologiju. Servleti se koriste kod:

- Komunikacije između korisnika − Servlet može da obradi više zahteva konkurentno i da ih ujedno sinhronizuje. Na ovaj način servleti podržavaju sisteme kao što su on-line konferencije
- Prosleđivanja zahteva − Servleti mogu da proslede zahteve ostalim serverima i servletima. Na taj način se odražava bolji balans opterećenja nekoliko servera koji imaju istu funkciju.



Slika 3 Izvršavanje Internet aplikacije pomoću Web servera

Njihov osnovni zadatak je:

- Prihvatanje i čitanje eksplicitnih podataka poslatih od strane klijenta, u suštini podaci dobijeni sa forme klijenta
- Prihvatanje i čitanje implicitnih podataka poslatih od strane klijenta, podaci dobijeni na osnovu hedera zahteva
- Generisanje rezultata u zavisnosti od primenjene poslovne logike
- Slanje eksplicitnih podataka klijentu, u formi HTML stranica
- Slanje implicitnih podataka klijentu u formi kodova statusa i samog heder-a odgovora

Osnovni primeri

Naglašeno je da su servleti Java klase koje moraju da imaju određeni format. Primer jednog najosnovnijeg servleta bi bio

```
import java.io.*;
import java.servlet.*;
import java.servlet.http.*;

public class Zdravo extends HttpServlet{
public voig doGet(HttpServletRequest request,
HttpServletResponse response) throws ServletException,
IOException {
PrintWriter out = response.getWriter();
out.println("Zdravo!");
}
}
```

U ovom osnovnom primeru, u okviru servleta se generiše tekst "Hello World" koji se prikazuje na strani klijenta, u trenutku kada se zatraži definisani servlet. Ovako definisanu klasu nije moguće kompajlirati samo sa osnovnim Java okruženjem, već je potrebno imati dodatne pakete. I nakon uspešnog kompajliranja datu klasu nije moguće samostalno pokretati, već određenim pozivom preko Web servera. O načinima instaliranja i poziva aplikacije preko web servera biće reči u kasnijim odeljcima.
Današnja popularna vizuelna okruženja ovaj postupak u mnogome olakšavaju. Potrebno je samo u okviru novog Web projekta izabrati opciju Insert – New – Servlet i dobija se automatski generisani kod u koji treba ubaciti kod iz prethodnog primera.
Ako je aplikacija pravilno instalirana, sam poziv servleta se može obaviti na više načina. Servleti se mogu pozivati direktno upisom njihove URL putanje unutar browser-a. Format URL putanje generalno zavisi od servera koji se koristi. Primer za Jakarta Tomcat Web server kod koga je je u direktorijumu „TOMCAT_HOME/webapps" instalirana aplikacija izgleda ovako:

```
http://machine-
name:port/Direktorijum_Aplikacije/Ime_Servleta
http://localhost:8080/Proba/ObradaPostMetodaServlet
```

Ako se želi simulirati GET metoda poziva servleta:

```
http://localhost:8080/Proba/ObradaGetMetodaServlet?par1=
88
```

Servlet se može pozivati i sa klijentske HTML stranice. U okviru HTML stranice se pojavljuje sledeći tag:

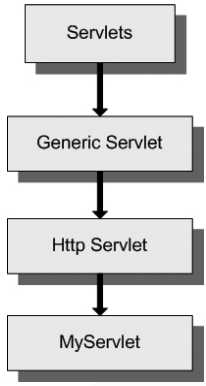
```
    <a href="/Proba/ObradaPostMetodaServlet"> Poziv
servleta </a>
```

Tada se u okviru servleta generiše HTML kod koji izgleda slično:

```
public class PozivHTMLServlet extends HttpServlet {
        public void doGet (HttpServletRequest request,
                           HttpServletResponse response)
                           throws ServletException,
IOException {
                ...
               out.println(... +
        "<a href=\"" +
response.encodeURL("/Proba/ObradaPostMetodaServlet") +
                            "\">Poziv servleta</a>
" +
                            ...);
                ...
}
        ...
}
```

Čitalac je mogao da primeti da servleti koriste paket javax.servlet, koji sadrži interfejse i klase za njihovo pisanje. Svi servleti implementuju interfejs Servlets, ili nasleđuju klasu koja je već implementirala ovaj interface. Najkorišćenija klasa ove vrste je klasa HttpServlet. Kada servlet dobije poziv od strane klijenta, on prihvata dva objekta:

- ServletRequest, koji održava komunikaciju od klijenta do servera.
- ServletResponse, koji održava komunikaciju od servera ponovo ka klijentu.



Slika 4 Hijerahija klasa servleta

ServletRequest and ServletResponse su takođe interfejsi definisani u navedenom javax.servlet paketu. ServletRequest interfejs dozvoljava servletu da priđe:

- informacijama kao što su imena parametara koje šalje klijent, protokol koji koristi klijent, i ime udaljenog računara koji je poslao zahtev koji je server primio.
- ServletInputStream Servleti dobijaju podatke od klijenta koji su poslati preko protokola kao što su HTTP POST ili GET metodi.

ServletResponse interfejs omogućava servletima metode za generisanje odgovora klijentima:

- Dozvoljava servletu da definiše sadržaj odgovora i tip podataka (MIME).
- Obezbeđuje izlazni stream, ServletOutputStream i Writer preko kojih servlet šalje podatke.

U okviru klase koja predstavlja servlet i koja nasleđuje klasu HttpServlet klasu postoje metode koji mogu obrađivati zahteve klijenta. Da bi se olakšala sama komunikacija sa klijentom postoje dva osnovna argumenta koja sadrže ovi metodi:

1. HttpServletRequest objekat, koji obuhvata podatke *od* klijenta
2. HttpServletResponse objekat, koji obuhvata podatke koji se šalju *ka* klijentu

HttpServletRequest objekat omogućava pristup HTTP header podacima. Takođe, omogućava primanje argumenata koje klijent šalje kao deo svog zahteva.

U zavisnosti od dobijenih podataka i tražene poslovne logike, potrebno je klijentu generisati odgovarajući odgovor. Najjednostavniji način, koji će biti primenjen u sledećim primerima ovog poglavlja je da se dati odgovor generiše u formi HTML stranice. Tada je potrebno obavestiti browser da se šalju podaci koji su tipa HTML i to na sledeći način:

```
response.setContentType("text/html");
```

Da bi se dobila korektna Web stranica potrebno je primeniti određeni broj puta naredbu println, tako što se kao argumenti u vidu Stringova pojavljuju HTML tagovi. Primer:

```
import java.io.*;
import java.servlet.*;
import java.servlet.http.*;

public class Zdravo extends HttpServlet {
public void doGet(HttpServletRequest request,
HttpServletResponse response)
throws ServletException, IOException {
response.setContentType("text/html");
PrintWriter out = response.getWriter();
String docType =
"<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 "+
"Transitional//EN\">\n";
out.println(docType +
"<HTML>\n" +
"<HEAD><TITLE>Zdravo</TITLE></HEAD>\n"+
"<BODY BGCOLOR=\"#FDF5E6\">\n" +
"<H1> Zdravo </H1>\n" +
"</BODY></HTML>");
}
}
```

U datom primeru je u okviru String promenljive docType definisan header HTML stranice i kasnije izvršena konkatenacija sa samim sadržajem, koji je takođe definisan u obliku Stringa.

Još jedan sličan primer je :

```
public class PrviServlet extends HttpServlet
    {
public void doGet (HttpServletRequest zahtev,
                    HttpServletResponse odgovor)
        throws ServletException, IOException
        {
          PrintWriter         out;
         String title = "Jednostavan odgovor Servera";
         odgovor.setContentType("text/html");
         out = odgovor.getWriter();
          out.println("<HTML><HEAD><TITLE>");
          out.println(title);
          out.println("</TITLE></HEAD><BODY>");
          out.println("<H1>" + title + "</H1>");
          out.println("<P>Ova strana je generisana pomocu
Servleta.");
          out.println("</BODY></HTML>");
          out.close();
        }
    }
```

U datom primeru klasa PrviServlet nasleđuje klasu HttpServlet, koja je implementirala Servlet interfejs. PrviServlet definiše svoj doGet metod. Ovaj metod se poziva kada klijent generiše GET zahtev sa svoje forme i kao rezultat se jednostavna HTML stranica vraća klijentu. Zahtevi od strane klijenta su sadržani u HttpServletRequest objektu, a odgovor klijentu je u objektu tipa HttpServletResponse. Zato što se u ovom primeru tekstualni podaci vraćaju klijentu, odgovor je poslat pomoću objekta Writer koji je sadržan u okviru HttpServletResponse objekta.

Naglašeno je da HttpServletRequest objekat omogućava pristup podacima koji su sadržani u HTTP header-u zahteva. Ovaj objekat omogućava prihvatanje argumenata koje klijent šalje kao deo svog zahteva. Podacima koje šalje klijent može se pristupiti pomoću getParameter() metoda, koji vraća vrednost imenovanog parametra. Ako parametar ima više od jedne vrednosti, koristi se metod getParameterValues(). Ovaj metod kao rezultat vraća niz vrednosti za imenovani parametar. Postoji i metod getParameterNames() koji vraća imena svih parametera. Objekat HttpServletResponse daje dve mogućnosti za vraćanje podataka klijentu, upotrebom metoda getWriter(), kada se dobija objekat tipa Writer, ili

upotrebom metoda getOutputStream(), kada se dobija objekat tipa ServletOutputStream. Metoda getWriter() se koristi kada su podaci koji se vraćaju klijentu tekstualnog tipa, a metoda getOutputStream() za binarne podatke. Pozivanje metoda close() u okviru dobijenih objekata nakon slanja odgovora omogućava serveru da zna kada je odgovor komplentiran.
Sa strane klijenta zahtevi mogu doći ili metodom GET ili POST. Metoda kojom se šalju podaci je definisana u okviru taga forme na stranici klijenta i njegovog atributa METHOD, kome se definiše jedna od datih vrednosti.
GET zahtev se obrađuje preklapanjem metoda doGet().

```
public class ObradaGetMetodaServlet extends HttpServlet {
public void doGet (HttpServletRequest zahtev,
HttpServletResponse odgovor)throws ServletException,
IOException
    {        ...
odgovor.setContentType("text/html");
        PrintWriter out = odgovor.getWriter();
out.println("<html>" +                  "<head><title>Primer
citanja vrednosti parametra</title></head>" +        ...);
String servletPar1 = zahtev.getParameter("par1");
        if (servletPar1 != null) {
out.println("<body>" + servletPar1);
        }
        out.println("</body></html>");
        out.close(); }
    ...}
```

Unutar metoda doGet(), pomoću metoda getParameter() objekta koji predstavlja `HttpServletRequest` se dobijaju vrednosti očekivanih parametara. Za generisanje odgovora klijentu doGet() metod koristi objekat Writer iz objekta HttpServletResponse da bi poslao tekstualne podatke klijentu. Pre pristupa ovom objektu definiše se tip sadržaja. Na kraju doGet() metoda, kada je odgovor poslat, Writer objekat se zatvara.
POST zahtev se obrađuje na identičan način preklapanjem metoda doPost().

```
public class ObradaPostMetodaServlet extends HttpServlet {
    public void doPost(HttpServletRequest request,
                       HttpServletResponse response)
      throws ServletException, IOException
    {
        ...
         response.setContentType("text/html");
        PrintWriter out = response.getWriter();
```

```
        out.println("<html>" + "<head><title>Primer citanja
vrednosti parametra</title></head>" +
      ...);
String servletPar1 = request.getParameter("par1");
        if (servletPar1 != null) {
            // prikazivanje procitane vrednosti
            out.println("<body>" + servletPar1);
        }
        out.println("</body></html>");
        out.close();
}
    ...
}
```

Kao i kod metoda doGet(), unutar metoda doPost(), pomoću metoda getParameter() objekta koji predstavlja `HttpServletRequest` se dobijaju vrednosti očekivanih parametara. Za generisanje odgovora klijentu doGet() metod koristi Writer iz objekta HttpServletResponse da bi poslao tekstualne podatke klijentu. Pre pristupa ovom objektu definiše se tip sadržaja. Na kraju doPost() metoda, kada je odgovor poslat, Writer objekat se zatvara.
Upotreba metoda doGet() ili metoda doPost() je potpuno identična. Svi ugrađeni objekti i njihovi metodi se mogu koristiti na potpuno identičan način. Kao što je moglo da se primeti iz prethodnih primera jedna od najbitnijih osobina servleta je mogućnost da se priđe parametrima koji su definisani i poslati na strani klijenta.
Najvažniji metodi u okviru objekta HttpServletRequest koji se mogu koristiti su:

- getParameter(“ime"). Dobija se URL-dekodirana vrednost prvog elementa koji se zove ime. Rezultat je null ako ne postoji takav parametar u elementima forme
- getParameterValues(“ime") Dobija se niz URL-dekodiranih vrednosti za sve elemente koji se zovu ime. U slučaju da se ime pojavljuje samo jednom dobija se niz sa jednim elementom. Rezultat je null ako ne postoji takav parametar u elementima forme
- getParameterNames() ili request.getParameterMap(). Dobija se Enumeration ili Map objekti od poslatih elemenata Uobičajeno je da je koristi u procesu debagovanja.

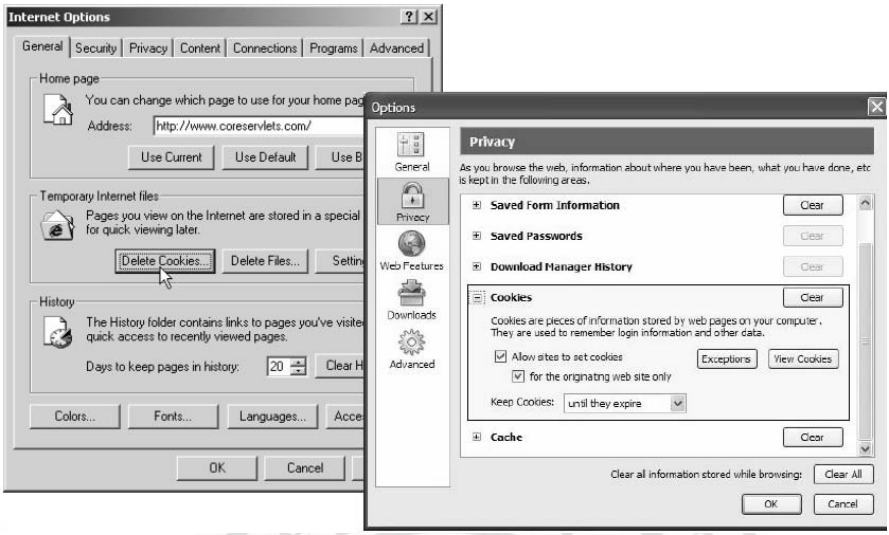
Životni ciklus servleta

Pored opisanih metoda doGet() i doPost() čijim preklapanjem se definišu odgovori nakon dobijenog zahteva od strane klijenta, postoji još metoda koji su definisani u osnovnoj klasi servleta i definišu načine izvršavanja i upotrebe ovih Javinih klasa. Osnovni metodi su:

- **init()** Izvršava se jednom kada se servlet prvi put učitava, pa se poziva tačno jednom. Ne postoje prepreke za postojanje konstruktora u samoj servlet klasi u kome će se izvršiti određene inicijalizacije, ali je to upravo zadatak ove metode *Ne* poziva se za svaki zahtev.
- **service()** Poziva se kod novog thread-a od strane servera za svaki zahtev. Tipičan scenario poziva je već opisan: postavi se tip sadržaja HTTP odgovora, definše se objekat PrintWriter, pomoću koga se šalje dinamički kreiran HTML Ne preporučuje se da se u izvedenoj klasi preklapa ovaj metod.
- **doGet, doPost, doXxx** Obrađuju GET, POST, ... zahteve. Upravo preklapanje ovih metoda omogućava željeno ponašanje servleta.
- **destroy()** Poziva se kada server briše instancu servleta. Namenjen je zadatku brisanja nepotrebnih podataka neposredno pre uništenja servleta, izvršava se oslobađanje resursa koje je servlet zauzimao, kao što su otvorene datoteke, konekcija sa bazom podataka, ... Obično se izvršava prilikom zaustavljanja Web servera *Ne* poziva se posle svakog zahteva.

Rad sa cookie-ijima

Cookie-iji omogućavaju da se na klijentski računar zapiše određena informacija koja se može iskoristiti prilikom sledećeg pristupa istog klijenta. Na primer servlet generiše jedinstveno ime i vrednost koje dodeli datom klijentu. Klijent vraća isto ime i vrednost kada se ponovo konektuje na isti sajt, ili isti domen u zavisnosti od podešavanja vrednosti cookie-ija. Ovakva mogućnost se može iskoristiti na više načina, pa su tipične upotrebe cookie-ija: mogućnost identifikacije korisnika tokom sesije elektronskog poslovanja(treba napomenuti da sami servleti imaju svoj API na višem nivou za ovaj zadatak i o njemu više reči u narednom odeljku), mogućnost izbegavanja upotrebe korisničkog imena i šifre, prilagođavanje sajta korisniku, ciljane reklame, ...



Slika 5 Brisanje vrednosti cookie-ija od strane korisnika

Sami cookie-iji su tekstualne informacije, pa samim tim i dostupne boljim poznavaocima računarskih tehnologija. Takođe, prilikom upotrebe cookie-ija problem predstavlja sama privatnost klijenta, a ne njegova bezbednost, jer na ovaj način serveri mogu da pamte korisnikove ranije akcije. Ako korisnik daje personalne informacije, serveri mogu da povezuju te informacije sa prethodnim akcijama. Takođe, serveri mogu deliti cookie informacije sa trećom stranom. Najgora opcija je ona koju koriste loše dizajnirani sajtovi tako što smeštaju poverljive informacije, kao što su brojevi kreditnih kartica, direktno u okviru cookie-ija. Osnovna preporuka pri upotrebi cookie-ija je da ako cookie-iji nisu kritični za izvršavanje zadatka, treba izbeći servlete koji u potpunosti prestaju sa radom ako se cookie-iji zabrane. Takođe nikako ne treba smeštati poverljive informacije u cookie-ije.

Treba napomenuti i da se cookie-iji mogu obrisati od strane korisnika na veoma jednostavan način (slika 5):

Sa strane servleta cookie se veoma jednostavna stvara i prosleđuje klijentu. Prvo je potrebno kreirati objekat koji predstavlja cookie. Treba pozivati Cookie konstruktor sa željenim imenom i vrednosti, pri čemu su oba argumenta tipa String:

```
Cookie c = new Cookie("userID", "a1234");
```

Preporuka je i da se postavi maksimalno vreme života cookie-ija, i na taj način se naglasiti čitaču da smesti cookie na disk umesto samo u memoriju. Data operacija se postiže pomoću metoda setMaxAge() i argumenta koji predstavlja broj sekundi.

```
c.setMaxAge(60*60*24*7); // Jedna nedelja
```

Kada se kreira željeni Cookie objekat, tada ga je potrebno smestiti u objekat koji predstavlja HTTP odgovor, koristeći response.addCookie. Tek izvršavanjem ove naredbe, cookie se šalje na stranu klijenta

```
response.addCookie(c);
```

Druga operacija koja se može izvršiti je čitanje podataka koji su upisani na strani klijenta u obliku cookie-ija. Da bi se izvršila ova operacija potrebno je pozvati metod request.getCookies(). Kao rezultat izvršavanja ovog metoda dobija se niz objekata tipa Cookie. Da bi se pronašla određene vrednot potrebno je kretati se po nizu objekata, pozvati metod getName() za svaki element niza sve dok se ne pronađe željeni cookie. Nakon toga pomoću getValue() može se koristiti upisana vrednost.

```
String cookieName = "userID";
Cookie[] cookies = request.getCookies();
if (cookies != null) {
for(int i=0; i<cookies.length; i++) {
Cookie cookie = cookies[i];
if (cookieName.equals(cookie.getName())) {
doSomethingWith(cookie.getValue());
}
}
}
```

Cookie se može iskoristiti i za proveru da li je korisnik već pristupao datoj aplikaciji. Sledi jedan takav primer:

```
public class RepeatVisitor extends HttpServlet {
public void doGet(HttpServletRequest request,
HttpServletResponse response)
throws ServletException, IOException {
boolean newbie = true;
Cookie[] cookies = request.getCookies();
if (cookies != null) {
for(int i=0; i<cookies.length; i++) {
```

```
Cookie c = cookies[i];
if((c.getName().equals("repeatVisitor"))&&(c.getValue().equal
s("yes"))) {
      newbie = false;
      break;
}
}
}
String title;
if (newbie) {
      Cookie returnVisitorCookie = new
Cookie("repeatVisitor", "yes");
      returnVisitorCookie.setMaxAge(60*60*24*365);
      response.addCookie(returnVisitorCookie);
      title = "Dobrodosli prvi put";
} else {
      title = "Dobrodosli ponovo";
}
response.setContentType("text/html");
PrintWriter out = response.getWriter();
… // (Izlaz sa razlicitim naslovom)
```

Pored navedenih, objekat Cookie ima i sledeće metode

- **getDomain()/setDomain()** – Specificira se domen koji obrađuje cookie. Trenutni host mora biti deo specificiranog domena.
- **getMaxAge()/setMaxAge()** – Čita/postavlja vreme života cookie-ija u sekundama. Ako se ova vrednost ne postavi, podrazumeva se da je vreme života cookie-ija samo trenutna sesija.
- **getName()** – Dobija se ime cookie-ija. Ne postoji setName metod, jer se ime definiše u okviru konstruktora. U okviru niza cookie-ija koji se dobija od strane klijenta, metod getName se koristi da bi se pronašao željeni cookie.
- **getPath()/setPath()** – Čita/postavlja putanju koja obrađuje cookie. Ako se ne navede, cookie pripada URLu koji je u okviru ili direktorijum iznad trenutne stranice.
- **getSecure()/setSecure()** – Čita/postavlja flag koji definiše da li se cookie izvršava samo pomoću SSL konkecije ili bilo koje konekcije.
- **getValue()/setValue()** – Čita/postavlja vrednost koja se želi pamtiti u okviru cookie-ija. Za nove cookie-ije, ova vrednost se postavlja u konstruktoru, a ne pomoću setValue. getValue se koristi da bi se dobila upisana vrednost. Ako se postojećem cookie-iju menja vrednost, potrebno je poslati tu novu vrednost sa response.addCookie.

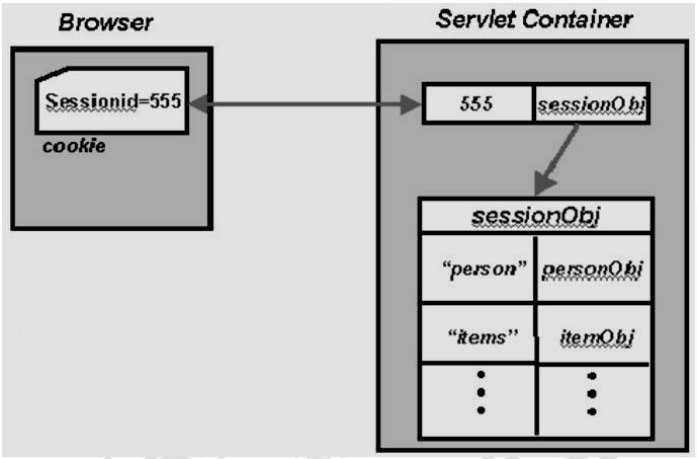
Još jedan primer koji prikazuje pročitane cookie-ije sa strane klijenta:

```
public class CookieTest extends HttpServlet {
public void doGet(HttpServletRequest request,
HttpServletResponse response)
throws ServletException, IOException {
for(int i=0; i<3; i++) {
Cookie cookie =
new Cookie("Session-Cookie-" + i,
"Cookie-Value-S" + i);
// No maxAge (ie maxAge = -1)
response.addCookie(cookie);
cookie = new Cookie("Persistent-Cookie-" + i,
"Cookie-Value-P" + i);
cookie.setMaxAge(3600);
response.addCookie(cookie);
}
… // Start an HTML table
Cookie[] cookies = request.getCookies();
if (cookies == null) {
out.println("<TR><TH COLSPAN=2>No cookies");
} else {
for(Cookie cookie: cookies) {
out.println
("<TR>\n" +
" <TD>" + cookie.getName() + "\n" +
" <TD>" + cookie.getValue());
}
}
out.println("</TABLE></BODY></HTML>");
}
}
```

Rad sa sesijom

Tokom rada određenog korisnika sa aplikacijom, često je potrebno čuvati određene podatke tokom jednog ili više zahteva. U okviru Java Internet aplikacije moguće je organizovati čuvanje podataka na 4 nivoa. Prvi nivo je kada se podaci čuvaju na nivou stranice. Tada ovi podaci nisu deljeni i ne može im se pristupati van date stranice. Drugi nivo je na nivou zahteva, kada su podaci vezani za pojedinačni zahtev korisnika. Tada su podaci vidljivi samo komponentama koje su direktno vezane za dati zahtev. Treći nivo je na nivou sesije. U ovom slučaju podaci su deljeni tokom sesije jednog korisnika i vidjivi su svim stranicama datog korisnika, bilo da su u

pitanju serverske komponente ili klijentske stranice. Četvrti nivo je nivo aplikacije i podaci su tada dostupni svim stranicama svih korisnika koji pripadaju datoj aplikaciji. U ovom odeljku će se detaljnije razmotriti čuvanje podataka na nivou sesije, a o ostalim tehnikama biće reči kasnije.
Sama sesija je bitan pojam i definiše ponašanje korisnika. Simulaciju sesije moguće je izvesti alternativnim metodama, kao što su čuvanje informacija u cookie-iju i provera informacija pri svakom pristupu korisnika. Sesiju je moguće simulirati i pomoću HIDDEN promenljive u okviru forme korisnika u koju se upisuje vrednost koja se kasnije proverava



Slika 6. Realizacija sesije

. Dalje moguće je u okviru pozivnog URL-a sa strane klijenta dodavati određene vrednosti za proveru. Zajedničko ovim pristupima je da zahtevaju dosta programerskog znanja i vremena. Da bi se olakšao ovaj veoma bitan zahtev, razvijen je poseban deo klasa i objekata koji se mogu koristiti i umnogome olakšati ovaj proces. Odlučeno je da se sami objekti sesije svakog klijenta čuvaju na serveru. S druge strane, sesije su automatski povezane sa klijentom pomoću cookie-ija ili promene URLa (slika).
Programerima je jednostavno pomoću metoda request.getSession() omogućeno da se dobije objekat sesije datog klijenta. U pozadini, sistem pregleda cookie ili URL dodatne informacije i pretražuje da li se ključ poklapa sa nekim od prethodno smeštenih objekata sesije. Ako pronađe poklapanje, kao rezultat vraća pronađeni objekat. Ako ne pronađe, kreira novi, povezuje cookie ili URL informacije sa njegovim ključem i vraća kao rezultat novi objekat sesije. Mehanizmi poput Hashtable dozvoljavaju da se

smeštaju neophodni objekti unutar sesije. Koriste se metodi objekta koji predstavlja sesiju i to setAttribute() smešta vrednost u sesiju, odnosno getAttribute() čita datu vrednost iz sesije. Primer korišćenja sesije

```
HttpSession session = request.getSession();
SomeClass value =
(SomeClass)session.getAttribute("someID");
if (value == null) {
value = new SomeClass(...);
session.setAttribute("someID", value);
}
doSomethingWith(value);
```

Treba napomenuti da nije potrebno pozivati ponovo metod setAttribute() (nakon promene vrednosti) ako je promenjena vrednost isti objekat. Ali ako je vrednost nepromenljiva, tada će promenjena vrednost biti novi objekat i mora se ponovo pozvati setAttribute().
Objekat koji predstavlja sesiju i koji je HttpSession tipa ima sledeće metode:

- **getAttribute()** – Vraća prethodno smeštenu vrednost iz objekta sesije. Rezultat je null ako nijedna vrednost nije povezana sa datim imenom.
- **setAttribute()** – povezuje vrednost sa navedenim imenom. Vrši se nadgledanje promena: vrednosti implementiraju HttpSessionBindingListener.
- **removeAttribute()** – uklanja vrednosti koje su povezane sa navedenim imenom.
- **getAttributeNames()** – vraća imena svih atributa u okviru sesije.
- **getId()** – vraća jedinstveni identifikator sesije.
- **isNew()** – prepoznaje da li je sesija nova za klijenta (ne za *stranicu*)
- **getCreationTime()** – vraća vreme kada je sesija prvi put kreirana
- **getLastAccessedTime()** – vraća kao rezultat vreme kada je sesija poslednji put poslata klijentu
- **getMaxInactiveInterval(), setMaxInactiveInterval()** – pročita ili postavi vremenski interval kada je sesije bez pristupa i postavlja se kao nevalidna
- **invalidate()** – postavi trenutnu sesiju kao nevalidnu

Jedan primer upotrebe sesije kada se ćita vrednost postavljena u okviru ovog objekta:

```
public void doGet(HttpServletRequest req,
HttpServletResponse res) {
  HttpSession session = request.getSession(true);
  userInfo = new UserInfo();
  session.putValue(session.getId(), userInfo);
  res.setContentType("text/html");
  PrintWriter out = res.getWriter();
  out.println("<HTML>");
  out.println("<HEAD><TITLE>Test</TITLE></HEAD>");
  out.println("<BODY>");
  ...
  userInfo =
(UserInfo)session.getValue(session.getId());
  out.close();
}
```

UNIVERZITET PRIVATNI AKREDITOVANI UNIVERZITET SINGIDUNUM

Primer međusobne komunikacije dva servleta

U sledećem primeru prikazaće na koji se način može ostvariti komunikacija između dva servleta. Pozivom prvog servleta korisniku će se prikazati ekran sa opcijom logovanja, pri kome korisnik treba da unese korisničko ime i lozinku. Kada se podaci unesu i pozove sledeći servlet, u njemu će se proveriti da li u bazi podataka postoji dati korisnik i prikazati korisniku odgovarajuću poruku.

Prvi servlet ima sledeći kod

```
package servleti;

import java.io.*;
import java.net.*;

import javax.servlet.*;
import javax.servlet.http.*;
public class Pocetak extends HttpServlet {

    protected void doGet(HttpServletRequest request,
HttpServletResponse response)
    throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-
8");
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<head>");
        out.println("<title>Pocetak Aplikacije
</title>");
        out.println("</head>");
        out.println("<body>");
        out.println("<h1>Unesite Vase podatke</h1>");
        out.println("<form name='forma' method='POST'
ACTION='ProveraPodatakaServlet'>");
        out.println("Korisnicko ime  <input
type='text' name='korisnickoIme'>");
        out.println("Lozinka  <input type=text'
name='lozinka'>");
        out.println("<input type='submit' name='dugme'
value='Posalji'>");
       out.println("</form>");
        out.println("</body>");
        out.println("</html>");
        out.close();
    }
}
```

U okviru ovog servleta se generiše opisana početnaforma sa korisničkim imenom i lozinkom. Kada se aktivira dugme serveru se upućuje poziv servleta koji je mapiran sa „ProveraPodatakaServlet“. Izgled ekrana nakon poziva prvog servleta je



Slika 7. Izgled ekrana nakon poziva servleta

U drugom servletu se prihvataju podaci poslati od strane korisnika i servlet treba da proveri u bazi podataka da li postoji traženi korisnik. U bazi podataka postoji tabela Student sa poljima ID (tipa AutoNumber), korisnickoIme (Text) i lozinka (Text), u kojoj se pamte podaci o korisnicima aplikacije. Ako korsnik postoji prikazuje mu se poruka „Uspesno ste se ulogovali“, a ako ne postoji dobija poruku „Poslati podaci nisu korektni!“.

Izgled ekrana nakon uspešnog logovanja je



Slika 8. Izgled ekrana nakon uspešnog logovanja

Izgled ekrana nakon neuspešnog logovanja je



Slika 9. Izgled ekrana nakon neuspešnog logovanja

Prilikom realizacije date aplikacije korišćen je i sledeći konfiguracioni fajl web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.4"
xmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
    <servlet>
        <servlet-name>Pocetak</servlet-name>
        <servlet-class>servleti.Pocetak</servlet-class>
    </servlet>
    <servlet>
        <servlet-name>ProveraPodataka</servlet-name>
        <servlet-
class>servleti.ProveraPodataka</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>Pocetak</servlet-name>
        <url-pattern>/PocetakAplikacija</url-pattern>
    </servlet-mapping>
    <servlet-mapping>
        <servlet-name>ProveraPodataka</servlet-name>
        <url-pattern>/ProveraPodatakaServlet</url-
pattern>
    </servlet-mapping>
    <session-config>
        <session-timeout>
            30
        </session-timeout>
    </session-config>
    <welcome-file-list>
  <welcome-file>
            index.jsp
        </welcome-file>
    </welcome-file-list>
</web-app>
```

4. JAVASERVER PAGES

Servleti su predstavljali početak razvoja Java Internet aplikacija. Ako se izvrši njihova kritička analiza može se zaključiti da je pomoću servleta jednostavno čitati podatke sa forme, čitati header-e HTTP zahteva, postavljati HTTP status kod i header odgovora, koristiti cookie-ije i rad sa sesijom, deliti podatke između servleta, zapamtiti podatke između različitih zahteva. S druge strane veoma je naporno koristiti println naredbe da bi se generisao HTML kod, i održavati i menjati generisani HTML. Zato se krenulo sa novom tehnologijom JSP, čije su osnovne prednosti:

- Odvojen statički od dinamičkog sadržaja: Kada se klijentski sadržaj generiše u okviru servleta, logika stvaranja dinamičkog sadržaja je sastavni deo koda servleta. Takođe statički deo klijentske strane je „sakriven“ u okviru servleta. Pored otežanog praćenja samog koda, nakon svake, pa i najmanje promene, servlet se mora ponovo kompajlirati. Sve ovo prouzrokuje stvaranje aplikacije čija se logika veoma teško razume, a svaka promena zahteva određeno vreme. Sa JSP stranicama statički deo se odvaja od dinamičkog korišćenjem specijalnih tagova i scriptleta. Kada dizajner strane napravi bilo koju promenu, JSP stranica se automatski rekompajlira i ponovo učita na web server pomoću JSP mašine.

- Piši jednom izvršavaj bilo gde: JSP stranice se mogu prenositi sa jedne platforme na drugu, ili usled promene Web server, ponašanje aplikacije će biti potpuno isto.

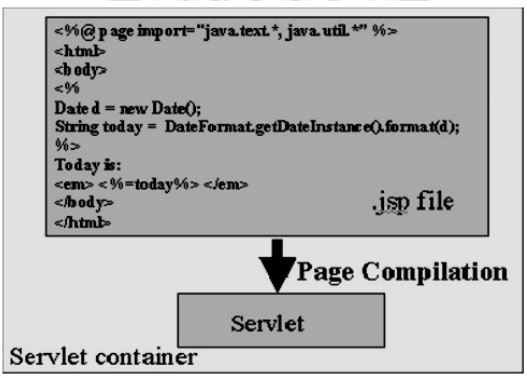
- Dinamički sadržaj se može predstaviti u različitim formatima: Ne postoji pravilo u kom formatu mora da bude statički (template) deo JSP stranice. Tako, JSP stranica može koristiti standardni HTML/DHTML kod koji se može izvršavati pomoću browser-a, ili WML kod koji se izvršava na ,,hanheld“ bežičnim uređajima, dok novije aplikacije koriste XML kod.

- Dobijanje Web pristupnog nivoa kod aplikacija n-slojne arhitekture: Standardna Sun-ova dokumentacija J2EE preporučuje korišćenje JSP stranica u odnosu na servlete za generisanje klijentske strane aplikacije.

- Potpuna usklađenost sa servlet tehnologijom: Za svakog servlet programera, potrebno je veoma malo truda i vremena za razumevanje JSP tehnolgije. U stvari, servlet programeri su u velikoj prednosti, jer su JSP stranice u stvari servleti pisani na velikom nivou apstrakcije. Sve što se može uraditi sa servletima, može i pomoću JSP stranica, ali mnogo jednostavnije!

Svrha postojanja JSP stranica je definisanje deklarativnog, prezentacijskog metoda za pisanje servleta. Kao što je već navedeno, JSP specifikacija je definisana kao standardna tehnologija na vrhu Servlet API klasa. Uobičajeno je da JSP stranica prolazi kroz dve faze: translacionu i fazu odgovora na zahtev. Translaciona faza se izvršava samo jednom, sve dok se u kod stranice ne unesu neke izmene, u kom slučaju se prevođenje ponavlja. Ako u kodu na stranici ne postoji greška, rezultat prevođenja je fajl koji predstavlja klasu koja implementira Servlet interface, kao što je prikazano na slici .

Ovu fazu izvršava JSP mašina na web serveru sama, automatski, kada primi prvi put zahtev za stranicom. Takođe, JSP stranica se može iskompajlirati u „class“ fajl pre instalacije aplikacije na server čime se izbegava kašnjenje pri prvom pozivu stranice od strane korisnika. Gde se postavlja kod stranice ili iskompajlirana klasa zavisi od korišćenog Web servera. Dobijena klasa od JSP stranice nasleđuje osnovnu klasu HttpJspBase, čime se u stvari vrši implementaciju Servlet interfejsa. Kada se jednom klasa učita u okviru Servlet container, _jspService() metod je odgovoran za odgovor na klijentske zahteve. Ovaj metod se ne može preklapati i izvršava se u posebnoj niti u okviru Servlet container-a.



Slika10. Translaciona faza JSP stranice

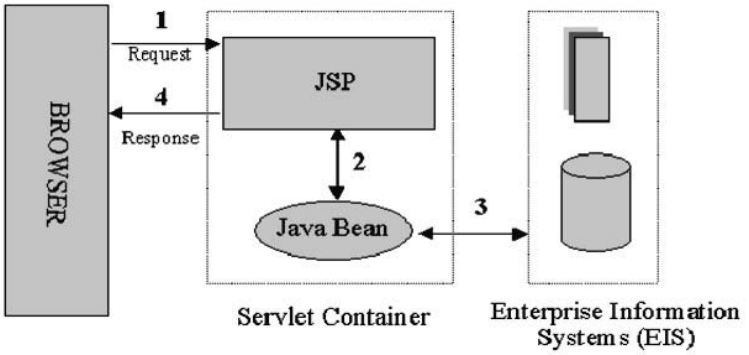
Kada se izvrši prevođenje koda JSP strane navedenog na prethodnoj slici dobija se sledeći složen i veoma nepregledni servlet, koji je je s druge strane prilagođen Web serverima i njihovom izvršavanju:

```
package jsp;
import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.PrintWriter;
import java.io.IOException;
import java.io.FileInputStream;
import java.io.ObjectInputStream;
import java.util.Vector;
import org.apache.jasper.runtime.*;
import java.beans.*;
import org.apache.jasper.JasperException;
import java.text.*;
import java.util.*;
public class
_0005cjsp_0005cjsptest_0002ejspjsptest_jsp_0
extends HttpJspBase {
static {
}
public _0005cjsp_0005cjsptest_0002ejspjsptest_jsp_0( ) {
}
private static boolean _jspx_inited = false;
public final void _jspx_init() throws JasperException {
}
public void _jspService(HttpServletRequest request,
HttpServletResponse response)
throws IOException, ServletException {
JspFactory _jspxFactory = null;
PageContext pageContext = null;
HttpSession session = null;
ServletContext application = null;
ServletConfig config = null;
JspWriter out = null;
Object page = this;
String _value = null;
try {
if (_jspx_inited == false) {
_jspx_init();
_jspx_inited = true;
}
_jspxFactory = JspFactory.getDefaultFactory();
```

```
response.setContentType("text/html");
pageContext = _jspxFactory.getPageContext(this,
request,response, "", true, 8192, true);
application = pageContext.getServletContext();
config = pageContext.getServletConfig();
session = pageContext.getSession();
out = pageContext.getOut();
// begin
out.write("\r\n<html>\r\n<body>\r\n");
// end
// begin
[file="E:\\jsp\\jsptest.jsp";from=(3,2);to=(5,0)]
Date d = new Date();
String today = DateFormat.getDateInstance().format(d);
// end
// begin
out.write("\r\nToday is: \r\n<em> ");
// end
// begin
[file="E:\\jsp\\jsptest.jsp";from=(7,8);to=(7,13)]
out.print(today);</b>
// end
// begin
out.write(" </em>\r\n</body>\r\n</html>\r\n");
// end
} catch (Exception ex) {
if (out.getBufferSize() != 0)
out.clear();
pageContext.handlePageException(ex);
} finally {
out.flush();
_jspxFactory.releasePageContext(pageContext);
}
}
}
```

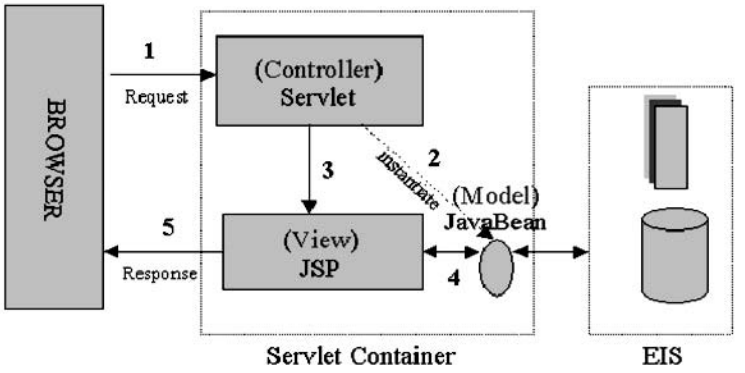
Rana JSP specifikacija je preporučivala dva metoda pristupa JSP stranicama, koji su nazvani Model 1 i Model 2 arhitekture. Ova dva pristupa su se razlikovala po mestu gde je vršena obrada zahteva. Kod Model 1 arhitekture zahtev dobijen od strane Web browsera se šalje direktno JSP stranici, koja je odgovorna za procesiranje zahteva i generisanje odgovora klijentu. Još uvek postoji razdvajanje klijentskog od serverskog dela aplikacije, jer se svi pristupi podacima odvijaju preko Java Beans. Model 1 arhitektura se može koristiti za jednostavnije aplikacije, dok je za kompleksne implementacije ovaj model nepogodan. Korišćenje ove

arhitekture obično dovodi do veoma velike količine scriptleta Java koda u okviru JSP stranice, posebno u slučajevima kada postoji složenija obrada klijentskih zahteva. Na ovaj način se gubi mogućnost da i dizajneri učestvuju u razvoju logike strane, jer je za ovu vrstu implementacije potrebno veće Java iskustvo. Još jedan nedostatak ovakve arhitekture je da svaka JSP stranica stranica mora individualno da vodi računa o stanju aplikacije.



Slika 11. Model 1 arhitekture pristupa JSP stranicama

S druge strane Model 2 arhitektura je podrazumevala da se zahtev šalje servletu, koji je imao ulogu kontrolera i pozicije gde je smeštena celokupna poslovna logika. Dobijeni rezultat se slao JSP stranici koja u ovom slučaju predstavlja prezentacioni nivo, t.j. određeni pogled na postojeće podatke.



Slika 12. Model 2 arhitekture pristupa JSP stranicama

JSP tehnologija je tokom vremena omogućila više pristupa pomoću kojih se mogla razviti određena Internet aplikacija.

Najvažniji su sledeći

- U okviru stranica scripting elementi pozivaju kod servleta direktno
- U okviru stranica scripting elementi pozivaju kod servleta indirektno, korišćenjem pomoćnih klasa
- Upotreba JavaBeans-a
- Servlet/JSP aplikacija (MVCpristup)
- MVC pristupa (Servlet-JSP aplikacija) uz upotrebu JSP expression jezika
- Posebni tagovi
- MVC pristup (Servlet-JSP aplikacija) upotrebom bean-ova, pomoćnih tagova, i framework-a, kao što su Struts ili JSF

Tehnologije su poređane po složenosti, ali i po korisnosti. Prve tehnologije su najjednostavnije, ali se najmanje upotrebljavaju, zbog nemogućnosti razvijanja kompleksnih aplikacija. S druge strane određeni framework, kao što je Struts ili JSF, je veoma kompleksna tehnologija i zahteva doređeno vreme da se sam framework savlada, ali s druge strane razvoj složenih aplikacija je pojednostavljen.

U nastavku teksta objasniće se osnove svih pomenutih tehnologija, ali će se pažnja usredsrediti na najsloženije koje se i najviše koriste.

4.1 Osnovna JSP sintaksa

JSP stranice se upotrebljavaju da bi se na jednostavniji način prikazao klijentski interfejs. Kako je HTML bio i ostao standard za prikaz Internet stranice, osnovna prednost JSP tehnologije je mogućnost kombinovanja HTML teksta sa Java kodom. Na ovaj način se omogućava i dizajnerima i programerima podela poslova. Tako se u okviru JSP stranica HTML tagovi mogu koristiti, kao i u običnim HTML stranicama. Na primer kod

```
<H1>Neki tekst</H1>
```

je dozvoljen i može se poslati dalje klijentu. Pri tome on se prevodi u kod servleta koji je sličan sledećem

```
out.print("<H1> Neki tekst </H1>");
```

Dozvoljeno je koristiti i HTML komentare koji se isto kao u HTML-u šalju dalje klijentu

```
<!-- Komentar -->
```

Moguće je koristiti i posebne JSP komentare, koji se za razliku od prethodnih, ne šalju samom klijentu

```
<%-- Komentar --%>
```

U oviru JSP stranica moguće je koristiti i dinamičke elemente, kao što su Java izrazi, skripleti , deklaracije i direktive. Pomoću posebne sintakse moguće je koristiti regularne Java izraze za prikaz određenih rezultata. Opšti tip sintakse je

```
<%= java_izraz %>
```

Primer

```
<%= new java.util.Date() %>
```

I još jedan, gde se može primetiti kominovane običnog HTML teksta sa mogućnostima Java programskog jezika

```
<html>
...
<h4>Dobrodošli, <%= username %></h4>
Danas je <%= new java.util.Date() %>.
...
</html>
```

Pored samih izraza moguće je koristiti i ostale osobine i mogućnosti Jave. Deo JSP stranice koji sadrži Java kod se naziva skriptlet (scriptlet) i ima sledeću sintaksu:

```
<% java_kod %>
```

Primer

```
<% for (int i = 0; i < 10; i++) ... %>
```

Ili

```
<html>
...

<% if (Math.random() < 0.5) { %>
Dobar dan!
<% } else { %>
Dobro veče!
<% } %>
...
</html>
```

```
<html>
...

<table border=1>
<tr>
  <td>R.br.</td>
  <td>Ime</td>
</tr>
<%
String names[] = {"Marko", "Nikola", "Igor", "Vladimir",
"Dejan"};
for (int i = 0; i < names.length; i++) {
%>
<tr>
  <td><%= i %></td>
  <td><%= names[i] %></td>
</tr>
<% } %>
</table>


...
</html>
```

Moguće je koristiti i Java deklaracije (declarations):

```
<%! java_deklaracija %>
<%! int a; %>
```

Kao i posebne direktive (directives) koje samoj JSP stranici daju dodatne mogućnosti, kao što je upotreba dodatnih fajlova, kodiranje stranice, reagovanje na grešku, ...

```
<%@ direktiva attr="..."%>
<%@ page contentType="text/plain" %>
```

U okviru JSP stranice moguće je koristiti i neke predefinisane promenljive. Neke od ovih promenljivih su definisane u sledećoj Tabeli

ime	Tip	Opis
request	HttpServletRequest	Zahtev klijentu
response	HttpServletResponse	Odgovor klijentu
out	JspWriter	Izlaz ka klijentu
session	HttpSession	Sesija
application	ServletContext	Aplikacija
page	(this)	Sama stranica

Objekat tipa HttpServletResponse je povezan sa odgovorom klijentu. Dozvoljeno je postavljanje HTTP statusnih kodova i zaglavlja odgovora (response headers). PrintWriter se koristi za slanje odgovora klijentu, pri čemu je out baferovana verzija PrintWriter pod nazivom JspWriter. Moguće je podešavanje veličine bafera, kao i njegovo potpuno isključivanje pomoću buffer atributa page direktive. out se koristi skoro isključivo u skriptletima, jer se JSP izrazi automatski smeštaju u izlazni tok. HttpSession objekat je povezan sa samim zahtevom. Sesije se kreiraju automatski, tako da je ova varijabla već instancirana čak iako nema ulazne reference na sesiju. Izuzetak je ako se koristi session atribut u okviru page direktive sa vrednošću false kako bi se isključilo praćenje sesija. U tom slučaju pokušaj pristupanja session promenljivoj rezultuje generisanjem poruke o grešci od strane servera u momentu prevođenja JSP strane u servlet. Objekat application je

objekat klase ServletContext koji služi za komunikaciju servleta i aplikacionog servera. Uobičajena upotreba je za smeštanje globalnih promenljivih uz pomoć metoda setAttribute()/getAttribute(). Objekat page je sinonim za ključnu reč this. Primer upotrebe predefinisanih promenljivih

```
<html>
...
<% if (request.getParameter("username") != null) { %>
Vrednost parametra: <%= request.getParameter("username") %>
<% } %>
...
</html>
```

Kod JSP stranica, kao i kod regularnih servleta, ponekad postoji potreba za korišćenjem init i destroy metoda. Problem može nastati kada servlet koji napravljen od JSP stranice možda već koristi svoje init i destroy metode. Tada njihovo preklapanje može da prouzrokuje probleme, pa je nelegalno koristiti JSP deklaracije da bi se deklarisali init ili destroy metodi. I u ovom slučaju rešenje postoji – treba koristiti metode jspInit() i jspDestroy(). Samom specifikacijom se garantuje da će auto-generisani servlet pozivati ove metode u okviru svojih init i destroy metoda, ali je standardna verzija jspInit i jspDestroy metoda prazna, sa mogućnošću preklapanja.

4.2 Direktive

U prethodnom delu teksta navedeno je da se u okviru JSP stranica mogu koristiti direktive kao posebne naredbe koje samoj stranici daju dodatne mogućnosti, na primer upotreba dodatnih fajlova, kodiranje stranice, reagovanje na grešku, ...
Prva direktiva koja se prezentira je **page** direktiva. Ona pruža informacije na visokom nivou o samom servletu koji se izvršava nakon JSP stranice. Pomoću ove direktive može se kontrolisati koje se klase importuju u okviru generisanog servleta, koju klasu servlet nasleđuje, koji se dodatni MIME tipovi generišu, kako se obrađuje multithread zahtevi, da li generisani servlet pripada sesiji, veličinu i ponašanje izlaznog bafera, definisanje koja stranica obrađuje neočekivane greške. U narednom delu teksta se prezentuje osnovna sintaksa upotrebe ove direktive da bi se postigli navedeni rezultati.

```
- <%@ page import="package.class" %>
- <%@ page import="package.class1,...,package.classN" %>
```

Pomoću atributa import dobija se generisanje import naredbi na vrhu definicije autogenerisanog servleta. Treba napomenuti da se JSP stranice mogu nalaziti bilo gde na serveru, ali klase koje se koriste u okviru JSP stranica moraju biti u uobičajenim servlet direktorijumima …/WEB-INF/classes ili …/WEB-INF/classes/direktorijumPaketa. Preporuka je da se uvek koriste paketi za klase koje se koriste u okviru JSP stranice!

```
- <%@ page contentType="MIME-Type" %>
- <%@ page contentType="MIME-Type; charset=Character-Set" %>
- <%@ page pageEncoding="Character-Set" %>
```

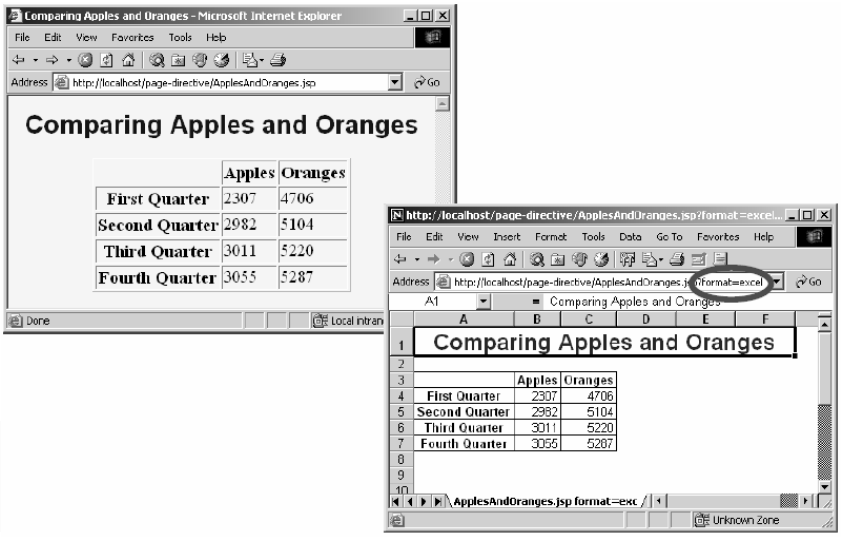
Pomoću atributa contentType i pageEncoding moguće je specificirati MIME tip stranice generisane od strane servleta koji je nastao od JSP stranice. Vrednost upotrebljenog atributa se ne može izračunati u vreme zahteva.
Jedan od efektnih primera korišćenja ovih atributa je i generisanje Excel stranice u okviru Web čitača. Za ovaj problem se može koristiti atribut contentType, ali treba napomenuti da ovaj atribut ne može da bude uslovljen. Sledećim kodom se uvek dobija Excel MIME tip, jer se navedena page direktiva uvek izvršava

```
<% boolean usingExcel = checkUserRequest(request); %>
<% if (usingExcel) { %>
<%@ page contentType="application/vnd.ms-excel" %>
<% } %>
```

Ako se želi da se Excel stranica generiše uslovno (na primer samo u slučajevima kada je određeni uslov ispunjen) rešenje je koristiti regularne JSP skriptlete sa response.setContentType:

```
…
<BODY>
<CENTER>
<H2>Prodaja jabuka i pomorandzi</H2>
<%
String format = request.getParameter("format");
if ((format != null) && (format.equals("excel"))) {
response.setContentType("application/vnd.ms-excel");
}
%>
<TABLE BORDER=1>
<TR><TH></TH> <TH>Jabuke<TH>Pomorandze
```

```
<TR><TH>Januar <TD>2307 <TD>4706
<TR><TH>Februar<TD>2982 <TD>5104
<TR><TH>Mart <TD>3011 <TD>5220
<TR><TH>April<TD>3055 <TD>5287
</TABLE>
</CENTER></BODY></HTML>
```



Slika 13. Različiti rezultati aplikacije u HTML formatu i u EXCEL formatu

- `<%@ page session="true" %> <%-- Default --%>`
- `<%@ page session="false" %>`

Pomoću atributa session moguće je da se označi da neka stranica ne pripada sesiji. Ako se navedeni atribut ne navede, stranica je deo sesije. Zašto koristiti ovaj atribut i kada je potrebno da stranica ne pripada sesiji? Dobra strana upotrebe ovog atributa je da se se smanjuje količina potrebne memorije na serveru, a to je veoma bitno, posebno ako je sajt visokog inteziteta.

- `<%@ page isELIgnored="false" %>`
- `<%@ page isELIgnored="true" %>`

Pomoću atributa isELIgnored se postiže kontrola da li se deo JSP 2.0 specifikacije koji se naziva Expression Language (EL) ignoriše (true) ili

normalno obrađuje (false). Treba napomenuti da će se sam EL detaljno obrađivati kao važan deo JSP specifikacije u daljem delu teksta.
Ako web.xml specificira verziju servleta 2.3 (JSP 1.2) ili raniju, default vrednost je true, ali je i dalje moguće menjati default vrednosti. Ako web.xml specificira verziju servleta 2.4 (JSP 2.0) ili raniju, default je false.

- `<%@ page errorPage="Relative URL" %>`

Pomoću atributa errorPage se specificira JSP stranica koja obrađuje bilo koji izuzetak koji se dogodio na tekućoj stranici. Izuzetak koji se dogodio je automatski dostupan dizajniranoj error stranici u obliku "exception“ promenljive. web.xml fajl dozvoljava definisanje error stranica na nivou aplikacija.

- `<%@ page isErrorPage="true" %>`
- `<%@ page isErrorPage="false" %> <%-- Default --%>`

Pomoću atributa isErrorPage se specificira da li se trenutna stranica može izvršavati kao error stranica za neku drugu JSP stranicu. I u ovom slučaju nova predefinisana promenljiva exception se kreira i dostupna je u okviru error stranica. Treba napomenuti da se ovakav način rada preporučuje samo za hitan backup, a da bi trebalo eksplicitno obraditi što je moguće više izuzetaka, na primer uvek treba proveriti unete podatke.
Primer stranice koja generiše grešku:

```
…<BODY>
<%@ page errorPage="/WEB-INF/SpeedErrors.jsp" %>
<TABLE BORDER=5 ALIGN="CENTER">
<TR><TH CLASS="TITLE">Computing Speed</TABLE>
<%!
private double toDouble(String value) {
return(Double.parseDouble(value));
}
%>
<%
double furlongs = toDouble(request.getParameter("furlongs"));
double fortnights = toDouble(request.getParameter("fortnights
"));
double speed = furlongs/fortnights;
%>
<UL>
<LI>Distance: <%= furlongs %> furlongs.
<LI>Time: <%= fortnights %> fortnights.
<LI>Speed: <%= speed %> furlongs per fortnight.
</UL>
```

```
</BODY>
</HTML>
```

Stranica koja obrađuje grešku sa prethodne stranice

```
…<BODY>
<%@ page isErrorPage="true" %>
<TABLE BORDER=5 ALIGN="CENTER">
<TR><TH CLASS="TITLE">
Error Computing Speed</TABLE>
<P>
ComputeSpeed.jsp reported the following error:
<I><%= exception %></I>. This problem occurred in the
following place:
<PRE>
<%@ page import="java.io.*" %>
<% exception.printStackTrace(new PrintWriter(out)); %>
</PRE>
</BODY></HTML>
```

– <%@ page isThreadSafe="true" %> <%-- Default --%>
– <%@ page isThreadSafe="false" %>

Pomoću atributa isThreadSafe se naglašava sistemu da kod nije siguran pri simultanom izvršavanju, tako da sistem može da spreči konkurentne pristupe kodu. U stvari, servletu se prenosi da implementira SingleThreadModel. Ovakav pristup može prouzrokovati lošije prefomanse u nekim situacijama, kao i nekorektan rezultat u drugim. Primer kada korisnik treba da dobije jedinstveni ID – znači postoji resurs kome treba da priđu i da ga dele svi korisnici:

```
<%! private int idNum = 0; %>
<%
String userID = "userID" + idNum;
out.println("Your ID is " + userID + ".");
idNum = idNum + 1;
%>
```

Navedeni atribut nije potrebno koristiti u svim slučajevima kada se želi siguran kod. Moguće je upotrebiti i standardnu Java sinhronizaciju:

```
<%! private int idNum = 0; %>
<%
synchronized(this) {
String userID = "userID" + idNum;
out.println("Your ID is " + userID + ".");
idNum = idNum + 1;
}
%>
```

Sa ovakvim pristupom dobijaju se bolje perfomanse u okruženju sa velikim saobraćajem. Takođe, naredba isThreadSafe="false" se neće korektno izvršiti, ako server koristi pristup sa redom instanci.

– <%@ page extends="package.class" %>

Pomoću atributa extends specificira se klasa roditelja servleta koji se dobija od JSP stranice. Ovaj atribut treba koristiti sa ekstremnom pažnjom. Uobičajena upotreba je da bi se nasledile klase koje prezentuje proizvođač servera (različite vrste podrške), a ne da bi se nasleđivale sopstvene klase.
Navedena page direktiva nije jedina koja se može koristiti u JSP okruženju. U narednom delu teksta će se razmatrati direktive koje omogućavaju jednostavniji rad sa spoljašnjim fajlovima, **include** i **plugin**.

– <jsp:include page="Relative URL" />

Ova direktiva se koristi kada se želi da se više puta koriste iste JSP, HTML, ili obični tekst dokumenti. Na ovaj način se dozvoljavalju promene uključenog sadržaja bez promene osnovnih JSP strana. Takođe, osnovna JSP stranica ne može da menja osnovne stranice: koristi se samo izlaz uključenih JSP stranica. Pri upotrebi ove direktive ne treba zaboraviti simbol / na kraju taga. Relativne URL adrese koje počinju sa znakom / se intrerpretiraju relativno u odnosu na Web aplikaciju, a ne relativno u odnosu na root servera i moguće je uključiti fajlove iz WEB-INF direktorijuma.

Primer upotrebe include direktive:

```
…
<BODY>
<TABLE BORDER=5 ALIGN="CENTER">
<TR><TH CLASS="TITLE">
Nove vesti na nasem sajtu</TABLE>
<P>
Poslednje tri price sa naseg sajta su:
<OL>
<LI><jsp:include page="/WEB-INF/Item1.html" />
<LI><jsp:include page="/WEB-INF/Item2.html" />
<LI><jsp:include page="/WEB-INF/Item3.html" />
</OL>
</BODY></HTML>
```

Jedna od stranica (na primer Item1.html) koje se pozivaju iz navedenog fajla

```
<B>Odlicni rezultati studenata naseg fakulteta</B> Na
poslednjim takmicenjima iz programiranja, nasi studeni su
pokazali odlicne rezultate Detaljnije na
<A        HREF="http://www.microsoft.com/Sometimes.html">Vise
detalja</A>
```

Može se primetiti da stranice koja se poziva nije celokupan HTML dokument, već postoje samo tagovi koji treba da se pojave na osnovnoj strani.
U okviru ove direktive mogu se koristiti i određeni elementi. Jedan od njih je i param element. Primer upotrebe ovog elementa:

```
<jsp:include page="/fragments/StandardHeading.jsp">
<jsp:param name="bgColor" value="YELLOW" />
</jsp:include>
```

Dobija se sledeći URL

```
http://host/path/MainPage.jsp?fgColor=RED
```

Na ovaj način osnovna stranica ima osobine fgColor: RED i bgColor: null, dok uključena stranica fgColor: RED i bgColor: YELLOW.

Pored include direktive, može se koristiti i include naredba:

```
– <%@ include file="Relative URL" %>
```

Pomoću nje se takođe dobija upotreba više puta istog JSP sadržaja, ali postoje izvesne razlike između include direktive i naredbe. Kod naredbe serveri ne proveravaju da li postoje promene uključenih fajlova. Zato je potrebno promeniti i osnovne JSP fajlove svaki put kada su i uključeni fajlovi promenjeni, kada se mogu koristiti i specifični mehanizmi operativnih sistema
Postavlja se pitanje u kojim slučajevima treba primeniti jedan ili drugi pristup. Preporuka je da se koristi jsp:include uvek kada je moguće, jer promene uključenih stranica ne zahtevaju nikakvu ručnu obradu. Takođe, postoji i razlika u brzini primene ovih naredbi. Ipak i include naredba (<%@ include …%>) ima i dodatna svojstva kao što ilustruje sledeći primer

– Osnovna stranica

```
 <%! int accessCount = 0; %>
```

– Uključena stranica

```
<%@ include file="snippet.jsp" %>
<%= accessCount++%>
```

Primer

```
<%@ page import="java.util.Date" %>
<%-- The following become fields in each servlet that
results from a JSP page that includes this file. --%>
<%!
private int accessCount = 0;
private Date accessDate = new Date();
private String accessHost = "<I>No previous access</I>";
%>
<P>
<HR>
This page &copy; 2006
<A HREF="http//www.my-company.com/">my-company.com</A>.
This page has been accessed <%= ++accessCount %>
times since server reboot. It was most recently
```

```
accessed from
<%= accessHost %> at <%= accessDate %>.
<% accessHost = request.getRemoteHost(); %>
<% accessDate = new Date(); %>
<BODY>
<TABLE BORDER=5 ALIGN="CENTER">
<TR><TH CLASS="TITLE">
Some Random Page</TABLE>
<P>
Information about our products and services.
<P>
Blah, blah, blah.
<P>
Yadda, yadda, yadda.
<%@ include file="/WEB-INF/ContactSection.jsp" %>
</BODY></HTML>
```

U prethodnom primeru footer definiše polje accessCount kao promenljivu instance. Ako osnovna stranica koristi accessCount, trebalo bi da se koristi naredba @include. U drugim slučajevima accessCount će biti undefined
Ipak u ovom primeru osnovna stranica ne koristi accessCount, ali se ipak koristi naredba @include.

4.3 Bean-ovi i njihova upotreba u Internet aplikacijama

To su Java klase koje moraju ispunjavati nekoliko pravila

- Moraju imati zero-argument (prazan) konstruktor
  - Ovaj zahtev se može ispuniti eksplicitnim definisanjem takvog konstruktora ili izostavljanjem svih konstruktora
- Ne bi trebalo da ima public promenljive instanci (polja)
- Neposredne vrednosti bi trebale da se dobijaju pomoću metoda nazvanih getXxx i setXxx
  - Ako klasa ima metod getTitle koji kao rezultat vraća String, kaže se da klasa poseduje String property title
  - Boolean property-ije koriste Xxx umesto getXxx
- Više informacije na http://java.sun.com/beans/docs

Da bi se napisao bean, ne smeju postojati public polja. Znači sledeći kod

```
public double speed;
```

treba zameniti sa

```
private double speed;
public double getSpeed() {
return(speed);
}
public void setSpeed(double newSpeed) {
speed = newSpeed;
}
```

I ovo treba uraditi u celokupnom Java kodu na svakom mestu.
U okviru JSP stranice Bean se koristi pomoću jsp:useBean. U najjednostavnijem slučaju, ovaj element pravi novi bean.

```
<jsp:useBean id="beanName" class="package.Class" />
```

U okviru stranice se inicijalizovani bean može dalje koristiti pomoću jsp:getProperty i jsp:getProperty tagova. Prvi navedeni tag čita i prikazuje vrednosti definisanih bean property-ija.

```
<jsp:getProperty name="beanName" property="propertyName"
/>
```

S druge strane jsp:setProperty menja vrednost bean property-ija

```
<jsp:setProperty name="beanName" property="propertyName"
value="propertyValue" />
```

U daljem tekstu će se razmatrati navedeni elementi.

```
- <jsp:useBean id="name" class="package.Class" />
```

Ovaj element dozvoljava instanciranje Java klasa bez eksplicitnog Java programiranja (XML-kompatibilna sintaksa). Kod

```
<jsp:useBean id="book1" class="coreservlets.Book" />
```

se može dobiti i pomoću skriptleta

```
<% coreservlets.Book book1 = new coreservlets.Book(); %>
```

ali jsp:useBean ima dve prednosti: jednostavnije je dobijati vrednosti objekata iz zahtevanih parametera i jednostavnije je deliti objekte između stranica ili servleta.

```
- <jsp:getProperty name="name" property="property" />
```

Ovaj element dozvoljava pristup bean property-ijima bez eksplicitnog Java programiranja. Opet

```
<jsp:getProperty name="book1" property="title" />
```

je ekvivalentno sa sledećim JSP izrazom

```
  <%= book1.getTitle() %>

- <jsp:setProperty name="name" property="property"
value="value" />
```

Ovaj element dozvoljava postavljanje bean property-ija bez eksplicitnog Java programiranja.
I

```
<jsp:setProperty name="book1" property="title"
value="PIA" />
```

je ekvivalentno sa sledećem skriptletu

```
<% book1.setTitle("PIA"); %>
```

Kada se želi da se upotrebi Bean u okviru aplikacije, potrebno je prvo implementirati klasu koja će preedstavljati dati Bean:

```
package primeri;

public class StringBean{
  private String poruka = "Nije specificirana poruka!"
```

```
public String getMessage(){
      return message;
}

public void setMessage(String message){
      this.message = message;
}
```

Bean-ovi se postavljaju u uobičajeni Java direktorijum

```
…/WEB-INF/classes/imePaketa
```

Treba napomenuti da se svakako preporučuje da beanovi (i pomoćne klase) moraju uvek biti u okviru paketa!
Pomoću prethodno navedenih elemenata u okviru JSP stranice koristi se napravljeni Bean:

```
<jsp:useBean id="stringBean" class="coreservlets.StringBean"
/>
<OL>
<LI>Initial value (from jsp:getProperty):
<I><jsp:getProperty name="stringBean" property="message"
/></I>
<LI>Initial value (from JSP expression):
<I><%= stringBean.getMessage() %></I>
<LI><jsp:setProperty name="stringBean" property="message"
value="Best string bean: Fortex" />
Value after setting property with jsp:setProperty: <I>
<jsp:getProperty name="stringBean" property="message" /></I>
<LI><% stringBean.setMessage ("My favorite: Kentucky
Wonder"); %>
Value after setting property with scriptlet:
<I><%= stringBean.getMessage() %></I>
</OL>
```

Vrednost Bean-a se može i postavljati

```
<!DOCTYPE ...>
...
<jsp:useBean id="entry"
class="coreservlets.SaleEntry" />
<%-- setItemID expects a String --%>
<jsp:setProperty
name="entry"
```

```
property="itemID"
value='<%= request.getParameter("itemID") %>' />
<%
int numItemsOrdered = 1;
try {
numItemsOrdered =
Integer.parseInt(request.getParameter("numItems"));
} catch(NumberFormatException nfe) {}
%>
<%-- setNumItems expects an int --%>
<jsp:setProperty
name="entry"
property="numItems"
value="<%= numItemsOrdered %>" />
<%
double discountCode = 1.0;
try {
String discountString =
request.getParameter("discountCode");
discountCode =
Double.parseDouble(discountString);
} catch(NumberFormatException nfe) {}
%>
<%-- setDiscountCode expects a double --%>
<jsp:setProperty
name="entry"
property="discountCode"
value="<%= discountCode %>" />
```

Može se primetiti da se pomoću atributa jsp:setProperty može postići da se vrednosti postavljaju pomoću specificiranog parametra zahteva. Ipak, potrebno je izvršiti jednostavnu automatsku konverziju tipova za property-ije koji očekuju vrednosti standardnih vrednosti boolean, Boolean, byte, Byte, char, Character, double, Double, int, Integer, float, Float, long, ili Long.

Radi jednostavnijeg pristupa može se upotrebiti simbol "*" za vrednost property-ija atributa jsp:setProperty da bi se postiglo da se vrednost dobija preko parametara zahteva koji ima isto ime kao i ime property-ija, ali se izvršava i jednostavna automatska konverzija tipova.

```
<jsp:useBean id="entry" class="coreservlets.SaleEntry" />
<jsp:setProperty name="entry" property="*" />
```

Ovo je ekstremni primer za realizaciju "form beans" – objekata čiji se property-iji popunjavaju prilikom slanja forme. Radi jednostavnije organizacije može se dati proces podeliti između više formi, tako da svaka forma popunjava određeni deo objekta.
Takođe, moguće je dozvoliti i uslovno kreiranje bean-a, t.j. kreirati novi objekat samo ako se ne može pronaći postojeći. Tada rezultat jsp:useBean je novi bean čija će se instanca formirati samo ako se ne može pronaći bean sa istim id i oblasti važenja. Ako bean sa istim id i oblasti važenja se pronađe, postojeći bean se jednostavno poveže sa promenljivom pomoću id. Uslovno postavljanje Bean property-ija se dobija tako što se

```
<jsp:useBean ... />
```

zamenjuje sa

```
<jsp:useBean ...>inicijalizacija</jsp:useBean>
```

Navedene naredbe (jsp:setProperty elementi) se izvršavaju samo ako se kreira novi bean, a ne izvršavaju se kada se pronađe postojeći. Primer uslovnog kreiranja Bean-a

```
public class AccessCountBean {
private String firstPage;
private int accessCount = 1;
public String getFirstPage() {
return(firstPage);
}
public void setFirstPage(String firstPage) {
this.firstPage = firstPage;
}
public int getAccessCount() {
return(accessCount);
}
public void setAccessCountIncrement(int increment) {
accessCount = accessCount + increment;
}
}
```

Stranica koja koristi definisani Bean:

```
<jsp:useBean id="counter"
class="coreservlets.AccessCountBean"scope="application">
<jsp:setProperty name="counter"property="firstPage"
```

```
value="SharedCounts1.jsp" />
</jsp:useBean>
Of SharedCounts1.jsp (this page),
<A HREF="SharedCounts2.jsp">SharedCounts2.jsp</A>, and
<A HREF="SharedCounts3.jsp">SharedCounts3.jsp</A>,
<jsp:getProperty name="counter" property="firstPage" />
was the first page accessed.
<P>
Collectively, the three pages have been accessed
<jsp:getProperty name="counter" property="accessCount" />
times.
<jsp:setProperty                                    name="counter"pr
operty="accessCountIncrement" value="1" />
```

Jedna od glavnih prednosti upotrebe Bean-ova je i mogućnost deljenja određene instance Bean-a između više stranica, bilo klijentskih, bilo serverskih. U okviru deklaracije Bean-a na JSP stranici može se koristiti atribut scope da bi se definisale dodatne lokacije gde se može smestiti bean.

```
<jsp:useBean id="…" class="…‟ scope="…" />
```

Na ovaj način može se definisati da više servleta ili JSP stranica dele podatke. Atribut scope može imati sledeće vrednosti

page - <jsp:useBean … scope="page"/> ili <jsp:useBean…>
Ovo predstavlja default vrednost. Bean objekat treba da bude smešten u okviru PageContext objekta za vreme trajanja zahteva. Metodi u okviru istog servleta mogu pristupaju bean-u.

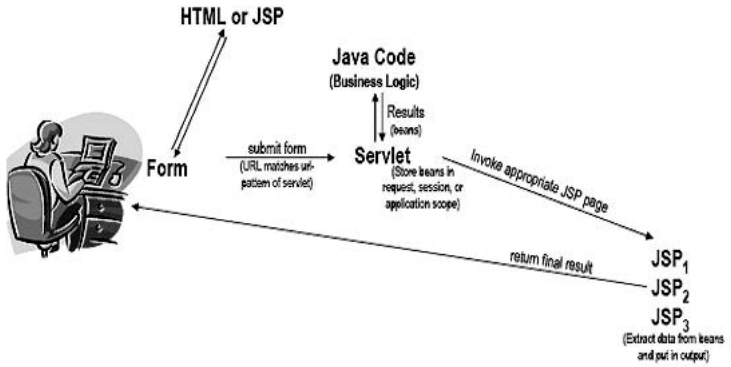
application - <jsp:useBean … scope="application"/>
Pomoću ove vrednosti Bean će biti smešten u ServletContext (dostupan preko promenljivih aplikacije ili pozvan pomoću getServletContext()). ServletContext je deljen između svih servleta iste Web aplikacije (ili svih servleta na serveru, ako eksplicitno nijedna Web aplikacija nije definisana).

session - <jsp:useBean … scope="session"/>
Pomoću ove vrednosti Bean će biti smešten u HttpSession objektu povezanom sa trenutnim zahtevom, gde mu se može pristupiti iz regularnog servlet koda sa metodama getAttribute() i setAttribute(), kao bilo kom drugom objektu sesije

request - <jsp:useBean … scope="request"/>
Pomoću ove vrednosti Bean objekat treba da bude smešten u ServletRequest objektu za vreme trajanja trenutnog zahteva, kada mu se može pristupiti pomoću getAttribute().

4.4 MVC pristup – kombinacija JSP strana i servleta

U okviru prethodnog teksta objašnjene su neke od tehnologija i njihove mogućnosti da se realizuju Internet aplikacije. Može se primetiti da se aplikacija može realizovati korišćenjem samo JSP stranica, kada se one koristite za jednostavniji razvoj i održavanje HTML sadržaja. U tom slučaju za jednostavniji dinamički kod, moguće je pozvati servlet kod pomoću scripting elemente. Za nešto kompleksnije aplikacije mogu se koristiti pomoćne klase koje se pozivaju iz scripting elemenata. Za srednje kompleksne aplikacije, koriste se bean-ovi i pomoćni tagovi. Ali se lako može uočiti da to nije dovoljno. Za kompleksnija procesiranja, rad samo sa JSP stranama nije najprihvatljivije rešenje. I pored odvajanja Java koda u odvojene klase, bean-ove, i pomoćne tagove, važi da pojedinačna stranica pruža pojedinačni osnovni izgled.
Znači za pojedinačni zahtev koji dolazi sa strane klijenta moguće je koristiti samo servlete. Ovaj pristup funkcioniše dobro kada je izlaz binarnog tipa (na primer slika) ili ne postoji izlaz (na primer obavlja se prosleđivanje ili redirekcija) ili se format/layout stranice drastično menja (portali). Takođe moguće je koristiti samo JSP stranice. Ovaj pristup funkcioniše dobro kada je izlaz je većim delom tipa karaktera (HTML) i kada je format/layout statičan bez promena.
Ipak najbolje je koristiti kombinaciju JSP strana i servleta (MVC arhitektura). Ona je neophodna kada je rezultat pojedinačnog zahteva više neposrednih stranica različitog izgleda. Ovaj pristip se preporučuje i u slučajevima kada postoji veliki razvojni tim sa različitim podtimovima koji realizuju Web razvoj i poslovnu logiku, i kada se izvršava komplikovana obrada podataka, ali je rezultat relativno statični layout. Vizuelni prikaz izvršavanja Internet aplikacije pomoću navedenog pristupa je dat na sledećoj slici



Slika 14. Životni tok aplikacije

Ovakav pristup je moguće unaprediti, tako što se određeni framework (Struts, JavaServer Faces) može koristiti. Ponekada je upotreba frameworka korisna, ali se ne zahteva njihovo korišćenje, jer je moguće implementirati MVC pristup pomoću objekta tipa RequestDispatcher i dobiti sasvim zadovoljavajuće rezultate za jednostavne i srednje kompleksne aplikacije. Treba naglasiti da MVC u potpunosti menja dizajn sistema, i ranije je navedeno da se još naziva i Model 2 pristup.

Da bi se realizovao navedeni pristup potrebno je sprovesti sledeće korake

1. Definisati bean-ove da bi prezentovali podatke
2. Koristiti servlet da bi se prihvatio podatak, pri čemu servlet čita parametre zahteva, proverava unete podatke, itd.
3. Postaviti bean-ove. Servlet realizuje poslovnu logiku (kod specifičan za aplikaciju) ili kod za pristup podacima da bi došao do rezultata. Resultati se smeštaju u bean-ove koji su definisani u okviru koraka 1.
4. Postaviti bean u okviru zahteva, sesije, ili sadržaja servleta. Servlet poziva setAttribute() u okviru objekata zahteva, sesije, ili sadržaja servleta da bi smestio referencu bean-ova koji predstavljaju rezultat zahteva.
5. Proslediti zahtev JSP stranici. Servlet prepoznaje koja JSP stranica odgovara datoj situaciji i koristi metod forward () od RequestDispatcher objekta da bi prebacio kontrolu datoj stranici.
6. Preuzeti podatke iz bean-a. JSP stranica pristupa beanu pomoću jsp:useBean i odgovarajuće oblasti važenja iz koraka 4. Stranica tada koristi jsp:getProperty da bi pristupila bean-ovim property-ijima.

U ovom slučaju JSP stranica ne kreira ili modifikuje bean; ona samo prihvata i pokazuje podatke koje servlet kreira.

Na strani servlete koristi se objekat RequestDispatcher da bi se definisala stranica koja se naredna izvršava. Uoptreba ovog objekta je definisana u sledećem primeru:

```
public void doGet(HttpServletRequest request,
HttpServletResponse response)
throws ServletException, IOException {
String operation = request.getParameter("operation");
if (operation == null) {
operation = "unknown";
}
String address;
if (operation.equals("order")) {
address = "/WEB-INF/Order.jsp";
} else if (operation.equals("cancel")) {
address = "/WEB-INF/Cancel.jsp";
} else {
address = "/WEB-INF/UnknownOperation.jsp";
}
RequestDispatcher dispatcher =
request.getRequestDispatcher(address);
dispatcher.forward(request, response);
}
```

Ovde treba napomenuti još jednu mogućnost objekta tipa RequestDispatcher. Pomoću metoda forward se dalje izvršavanje prosleđuje određenoj klijentskoj stranici i to tako da se sam URL stranice ne vidi, već korisnik vidi samo URL servleta. Ovakav pristup ima i određene nedostatke, na primer korisnik ne može da izvrši bookmark dobijene JSP stranice, jer se njen URL i ne vidi.

Navedeni objekat ima i mogućnost upotrebe metoda response.sendRedirect umesto RequestDispatcher.forward. Tada i sam korisnik vidi URL dobijene JSP stranice.

Prednosti upotrebe metoda sendRedirect su da

- Korisnik može pristupiti JSP stranici odvojeno
- Korisnik može da zapamti adresu JSP stranice

Nedostaci upotrebe metoda sendRedirect

- Korisnik može da poseti JSP stranici bez pristupanja servletu, pa JSP podaci mogu da budu nedostupni
- U okviru JSP stranice trebalo bi detektovati ovu situaciju
- Ne može se pristupati objektima koji su smešteni u okviru zahteva, jer se pomoću sendRedirect pravi nova instanca ovog objekta.

Ako se pogledaju preporučeni koraci može se zaključiti da JSP stranice ne bi trebale da kreiraju objekte. Sada servlet, a ne JSP stranica, bi trebalo da kreira sve objekte koji predstavljaju podatke. Da bi se garantovalo da se u okviru JSP stranice neće kreirati objekti, trebalo bi koristiti

```
<jsp:useBean ... type="package.Class" />
```

Umesto dosadašnjeg

```
<jsp:useBean ... class="package.Class" />
```

Takođe JSP stranica ne bi trebalo ni da modifikuje objekte, znači trebalo bi koristiti samo tag jsp:getProperty, ali ne i jsp:setProperty.

Ako se razmatra navedena MVC arhitektura i preporuka da se podaci smeštaju u okviru Bean-ova, ostaje pitanja gde i koliko čuvati napravljene objekte. Kao i u ranijim razmatranjima postoje 3 (mada teorijski 4) mogućnosti:

- na nivou zahteva (request <jsp:useBean id="..." type="..." scope="request" />),
- na nivou sesije (session <jsp:useBean id="..." type="..." scope="session" />),
- na nivou aplikacije (application <jsp:useBean id="..." type="..." scope="application" />)
- i pristup koji se ne koristi u okviru MVC arhitekture – na nivou stranice (page <jsp:useBean id="..." type="..." scope="page" /> ili samo <jsp:useBean id="..." type="..." />).

Za svaki od navedenih nivoa deljenje podataka važi veoma sličan pristup. U sledećim primerima demonstriraće se čitanje parametara na osnovu onog što je do sada navedeno. U kasnijem delu ovog poglavlja obradiće se poseban deo ove tehnologije (EL) koji omogućava efikasniji i razumljiviji pristup podacima smeštenim u okviru određene oblasti važenja.

U slučaju rada sa deljenjem podataka na nivou zahteva kod na strani servleta izgleda:

```
ValueObject value = new ValueObject(...);
request.setAttribute("kljuc", value);
RequestDispatcher dis = request.getRequestDispatcher
("/WEB-INF/SledecaStrana.jsp");
dispatcher.forward(request, response);
```

Kao što se može primetiti željena vrednost je smeštena u okviru objekta tipa request pod određenim imenom (kljuc) i dalje izvršavanje je prosleđeno stranici SledecaStrana.jsp. U okviru te stranice sačuvanom podatku se može pristupiti na sledeći način

```
<jsp:useBean id="kljuc" type="somePackage.ValueObject"
scope="request" />
<jsp:getProperty name="kljuc" property="someProperty" />
```

Sledeći primer ilustruje primenu ovakvog načina deljenja podataka. Neka je cilj prikazati slučajan broj korisniku. Na taj način svaki zahtev dobija novi broj, tako da je odgovarajuće deljenje podataka na nivou zahteva (request based).

Rezultat će se pamtiti u okviru sledećeg bean-a

```
package primer;
public class NumberBean {
      private double num = 0;
      public NumberBean(double number) {
            setNumber(number);
      }
      public double getNumber() {
            return(num);
      }
      public void setNumber(double number) {
            num = number;
      }
}
```

U okviru JSP stranice podatak se koristi na sledeći način

```
<HTML>
<BODY>
<jsp:useBean id="randomNum" type="primer.NumberBean"
scope="request" />
<H2>Random Number:
<jsp:getProperty name="randomNum" property="number" />
</H2>
</BODY>
</HTML>
```

- Sličan postupak je i kod deljenja podataka na nivou sesije. Kod na strani servleta izgleda:

```
ValueObject value = new ValueObject(...);
HttpSession session = request.getSession();
session.setAttribute("kljuc", value);
RequestDispatcher dis =
request.getRequestDispatcher("/WEB-INF/
SledecaStrana.jsp");
dispatcher.forward(request, response);
```

Kao što se može primetiti željena vrednost je smeštena u okviru objekta tipa HttpSession, koji predstavlja trenutnu sesiju samog korisnika, pod određenim imenom (kljuc) i dalje izvršavanje je prosleđeno stranici SledecaStrana.jsp. U okviru te stranice sačuvanom podatku se može pristupiti na sledeći način

```
<jsp:useBean id="kljuc" type="somePackage.ValueObject"
scope="session" />
<jsp:getProperty name="kljuc" property="someProperty" />
```

Kao primer ovakvog pristupa može poslužiti aplikacija čiji je cilj prikazati korisnikovo ime i prezime. Ako korisnik ne definiše svoje ime, treba da mu se prikaže ime koje je prethodno definisao, a ako korisnik ne definiše svoje ime i ne može da se pronađe nijedno prethodno definisano ime, potrebno je prikazati upozorenje. Sada je slučaj kada se podatak smešta za svakog klijenta, tako da je odgovarajuće deljenje podataka na nivou sesije (session-based).

Opet se podaci smeštaju u okviru bean-a

```
package primer;
public class NameBean {
private String firstName = "Missing first name";
private String lastName = "Missing last name";
public NameBean() {}
public NameBean(String firstName, String lastName) {
setFirstName(firstName);
setLastName(lastName);
}
public String getFirstName() {
return(firstName);
}
…
}
```

Kod servleta koji realizuje samu poslovnu logiku i formira dati bean je

```
public class RegistrationServlet extends HttpServlet {
public void doGet(HttpServletRequest request,
HttpServletResponse response)
throws ServletException, IOException {
HttpSession session = request.getSession();
NameBean nameBean =
(NameBean)session.getAttribute("nameBean");
if (nameBean == null) {
nameBean = new NameBean();
session.setAttribute("nameBean", nameBean);
}
String firstName =
request.getParameter("firstName");
if ((firstName != null) &&
(!firstName.trim().equals(""))) {
nameBean.setFirstName(firstName);
}
String lastName =
request.getParameter("lastName");
if ((lastName != null) &&
(!lastName.trim().equals(""))) {
nameBean.setLastName(lastName);
}
String address =
"/WEB-INF/mvc-sharing/ShowName.jsp";
RequestDispatcher dispatcher =
request.getRequestDispatcher(address);
```

```
dispatcher.forward(request, response);
}
}
```

Klijentska strana koristi smešteni bean na sledeći način

```
<BODY>
<H1>Thanks for Registering</H1>
<jsp:useBean id="nameBean" type="primer.NameBean"
scope="session" />
<H2>First Name:
<jsp:getProperty name="nameBean" property="firstName" />
</H2>
<H2>Last Name:
<jsp:getProperty name="nameBean" property="lastName" />
</H2>
</BODY>
</HTML>
```

- I rad sa podacima smeštenim na nivou aplikacije je skoro identičan:

```
      synchronized(this) {
            ValueObject value = new ValueObject(...);
            getServletContext().setAttribute("kljuc", value);
RequestDispatcher dis =request.getRequestDispatcher("/WEB-
INF/SledecaStrana.jsp");
            dispatcher.forward(request, response);
             }
```

Kao što se može primetiti željena vrednost je smeštena u okviru objekta koji predstavlja kontekst same aplikacije i dobijen je pomoću metoda getServletContext(), pod određenim imenom (kljuc) i dalje izvršavanje je prosleđeno stranici SledecaStrana.jsp. U okviru te stranice sačuvanom podatku se može pristupiti na sledeći način

```
<jsp:useBean id="kljuc" type="somePackage.ValueObj"
scope="application" />
<jsp:getProperty name="kljuc" property="someProperty" />
```

Kao primer ovakvog pristupa može poslužiti aplikacija čiji je cilj prikazati prost broj određene dužine. Ako korisnik ne defniše željenu dužinu, trebalo bi prikazati bilo koji prost broj dužine koja se koristi za bilo kog korisnika.

U datom primeru podatak se deli između više klijenata, tako da je deljenje podataka na nivou aplikacije (application-based) odgovarajuće.
Podatak koji predstavlja prost broj smešta se u okviru bean-a

```
package primer;
import java.math.BigInteger;
public class PrimeBean {
private BigInteger prime;
public PrimeBean(String lengthString) {
int length = 150;
try {
      length = Integer.parseInt(lengthString);
} catch(NumberFormatException nfe) {}
      setPrime(Primes.nextPrime(Primes.random(length)));
}
public BigInteger getPrime() {
      return(prime);
}…
}
```

Servlet koji realizuje traženu operaciju je

```
public class PrimeServlet extends HttpServlet {
public void doGet(HttpServletRequest request,
HttpServletResponse response)
throws ServletException, IOException {
String length = request.getParameter("primeLength");
ServletContext context = getServletContext();
synchronized(this) {
if ((context.getAttribute("primeBean") == null) ||(length !=
null)) {
      PrimeBean primeBean = new PrimeBean(length);
      context.setAttribute("primeBean", primeBean);
}
String address ="/WEB-INF/mvc-sharing/ShowPrime.jsp";
RequestDispatcher dispatcher
=request.getRequestDispatcher(address);
dispatcher.forward(request, response);
}
}
}
```

I na kraju podatak se prikazuje u okviru JSP stranice

```
<HTML>
<BODY>
<H1>A Prime Number</H1>
<jsp:useBean id="primeBean" type="primer.PrimeBean"
scope="application" />
<jsp:getProperty name="primeBean" property="prime" />
</BODY>
</HTML>
```

4.5 Expression jezik (EL)

Ako se analizira opisani pristup, pored većeg broja korisnih osobina, mogu se uočiti i određeni nedostataci. Glavni nedostatak je u završnom koraku, način na koji se realizuje prezentovanje rezultata u okviru JSP stranice. U datim primerima koriste se tagovi jsp:useBean i jsp:getProperty. Njihova primena može biti nespretna i preopširna, ne može se pristupiti bean podproperty-ijima. Na kraju dobija se kod koji se teško otežava i umanjuje se uticaj MVC arhitekture. Zato je odlučeno da se izvrše određena proširenja realizovane tehnologije. Cilj je bio upotrebiti koncizniji pristup, omogućiti pristup bean podproperty-ijima i jednostavniju sintaksu pristupa za Web developere. Trenutno najbitnije proširenje JSP specifikacije je expression jezik (EL).

Glavne prednosti novih opcija su

- Koncizniji pristup smeštenim objektima - da bi se pristupilo "smeštenim promenljivama" (objekti smešteni metodom setAttribute u okviru PageContext, HttpServletRequest, HttpSession, ili ServletContext), na primer nekoj promenljivoj primer, koristi se ${primer}.
- Skraćena notacija za bean property-ije - da bi se pristupilo companyName property-iju (na primer, rezultat dobijen od metoda getCompanyName) smeštene promenljive pod imenom company, koristi se ${company.companyName}. Da bi se pristupilo property-iju firstName od property-ija president smeštene promenljive company, koristi se ${company.president.firstName}.
- Jednostavniji pristup elementima kolekcije – pristup elementima tipa Array, List, ili Map, koristi se ${variable[indexOrKey]}. Treba primetiti da je potrebno realizovati prikaz indeksa ili ključa u formi legalnoj za imena Java promenljivih.

- Jednostavan pristup parametrima zahteva, cookie-ija, i drugih podataka zahteva – da bi se pristupilo standardnim tipovima podataka zahteva, moguće je koristiti jedan od nekoliko predefinisanih implicitnih objekata.
- Mali, ali koristan skup jednostavnih operatora – za rad sa objektima u okviru EL izraza, može se koristiti bilo koji od nekoliko aritmetičkih, relacionih, logičkih, ili empty-testing operatora.
- Uslovni rezultati – da bi se izabrao prikaz između nekoliko opcija, nije potrebno koristiti Java skriplete. Moguće je koristiti ${test ? option1 : option2}.
- Automatska konverzija tipova – EL uklanja potrebu za korišćenjem konverzije tipova i omogućava ovaj proces automatskim.
- Ne prikazuje se nikakva vrednosti, umesto poruke o greškama – u većini slučajeva, nedefinisana promenljiva ili NullPointerExceptions prikazaće se kao prazni stringovi, a neće se dogoditi izuzetak.

Kako ovaj deo JSP tehnologije nije deo početnih verzija, nije ga moguće koristiti u svim slučajevima. EL je moguće upotrebljavati samo u okviru servera koji imaju podršku za verziju JSP 2.0 i kasnije (odnosno verzije servleta 2.4 i kasnije). Na primer Tomcat 5 ima podršku za EL, a Tomcat 4 nema.
Takođe, potrebno je naznačiti u okviru web.xml fajla da se koriste navedene verzije

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation= "http://java.sun.com/xml/ns/j2ee web-
app_2_4.xsd"
version="2.4">
…
</web-app>
```

Osnovna forma koja se koristi je sledeća definica EL elementa

```
${expression}
```

Ovako definisani EL elementi se mogu pojaviti kako u običnom HTML tekstu tako i u okviru JSP tag atributa, i omogućiti izvršavanje regularnih JSP izraza.

Na primer:

```
<UL>
<LI>Ime: ${expression1}
<LI>Adresa: ${expression2}
</UL>
<jsp:include page="${expression3}" />
```

Sam EL je veom fleksibilan, pa se može koristiti više izraza u okviru jednog atributa (moguće ih je kombinovati i sa statičkim tekstom), tako da se na kraju konvertuju u stringove i izvrši njihova konkatenacija. Na primer:

```
<jsp:include page="${expr1}nesto${expr2}" />
```

Treba naglasiti da ako se navede ${varName}, tada se izvršava pretraga objekata tipa PageContext, HttpServletRequest, HttpSession, ServletContext, u navedenom redosledu i prikaz objekta koji sadrži navedeno ime. Opet treba naglasiti da upotreba PageContext ne odgovara MVC pristupu. Datom pozivu EL elementa ekvivalentne forme su

```
<jsp:useBean id="name‟ type="paket.klasa" scope="...">
```

ili

```
<%= name %>
```

Sledeći primer ilustruje redosled pretragu imena navedenog property-ija

```
public class ScopedVars extends HttpServlet {
      public void doGet(HttpServletRequest request,
      HttpServletResponse response) throws ServletException,
      IOException {
      request.setAttribute("attribute1", "First Value");
      HttpSession session = request.getSession();
      session.setAttribute("attribute2", "Second Value");
      ServletContext application = getServletContext();
      application.setAttribute("attribute3", new
java.util.Date());
      request.setAttribute("repeated", "Request");
      session.setAttribute("repeated", "Session");
      application.setAttribute("repeated", "ServletContext");
      RequestDispatcher dispatcher =
      request.getRequestDispatcher("/el/redosled.jsp");
      dispatcher.forward(request, response);
      }
}
```

Ako se na navedenoj klijentskoj strani (redosled.jsp) pristupi definisanim parametrima:

```
<!DOCTYPE …>
…
<TABLE BORDER=5 ALIGN="CENTER">
<TR><TH CLASS="TITLE">
Pristup parametrima
</TABLE>
<P>
<UL>
<LI><B>attribute1:</B> ${attribute1}
<LI><B>attribute2:</B> ${attribute2}
<LI><B>attribute3:</B> ${attribute3}
<LI><B>Gde se nalazi "repeated" atribut:</B> ${repeated}
</UL>
</BODY></HTML>
```

Do sada su predstavljeni primeri kada je pristupano samim parametrima. Pomoću EL može se na veoma koncizan i efikasan način pristupiti i složenijim podacima, kao što su property-iji u okviru bean-a. Sintaksa je sledeća

```
${varName.propertyName}
```

Okruženje traži promenljivu pod imenom varName i to u okviru objekata tipa PageContext, HttpServletRequest, HttpSession, ServletContext, u navedenom redosledu. Kada je pronađe, uzima vrednost njenog property-ija propertyName i dalje obrađuje. Na osnovu dosadašnjeg pregleda može se zaključiti da su ekvivalentne forme

```
${customer.firstName}
```

i

```
<%@ page import="coreservlets.NameBean" %>
<% NameBean person =
(NameBean)pageContext.findAttribute("customer");%>
<%= person.getFirstName() %>
```

i

```
<jsp:useBean id="customer" type="coreservlets.NameBean"
scope="request, session, or application" />
<jsp:getProperty name="customer" property="firstName" />
```

Jedna od prednosti EL je što ima mogućnost da prikaže članove određene kolekcije (niza, liste, heš mape, …). Sintaksa je

```
${attributeName[entryName]}
```

Može se upotrebiti kod

- Nizova. Ekvivalentno je sa theArray[index]
- Lista. Ekvivalentno je sa theList.get(index)
- Mapa. Ekvivalentno je sa theMap.get(keyName)

4.6 JSTL (JSP Standard Tag Library)

JSTL (JSP Standard Tag Library) je baziran na Struts tagovima za kontrolu petlji i logičke operacije . Nije bio deo JSP 1.2 ili 2.0 specifikacije, već je zahtevao posebnu specifikaciju i podešavanje. JSTL EL je sada deo JSP 2.0 specifikacije. Kompletna JSTL specifikacija se nalazi na

http://jcp.org/aboutJava/communityprocess/final/jsr052

Pored ostalih mogućnosti JSTL nudi tagove pomoću kojih se može na jednostavan način prikazati (obraditi) članovi određene kolekcije i to u okviru klijentske strane. Treba napomenuti da sam JSTL obuhvata i čitav niz dodatnih opcija klijentskoj JSP strani, kao što su pristup bazi podataka, tagovi za uslovne realizacije, za rad sa XML fajlovima, itd. Jedan od razloga zašto se ovi tagovi neće obrađivati u ovoj knjizi je i taj jer se njihova upotreba ne uklapa u MVC pristup, koji se želi realizovati.
Da bi se JSTL tagovi mogli koristiti potrebno je na početku JSP strane uključiti JSTL biblioteku pomoću sledećeg taga

```
<%@ taglib prefix="c"uri="http://java.sun.com/jstl/core"
%>
```

Tag koji ima mogućnost obrade kolekcije podataka je c:forEach i njegova sintaksa je

```
<c:forEach var="element" items="${kolekcija}">
  <c:out value="${element}"/>
</c:forEach>
```

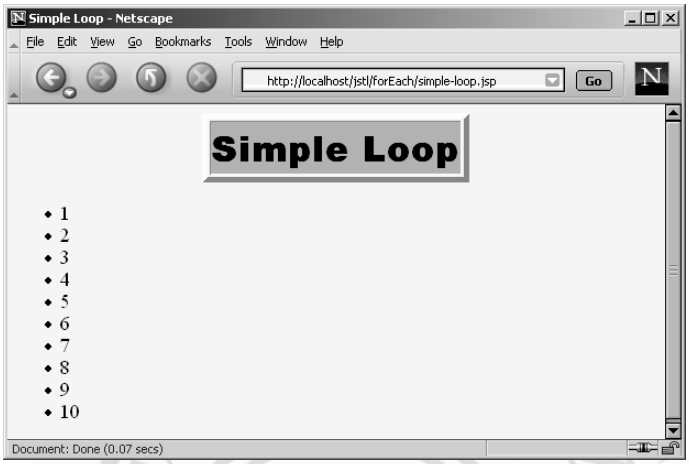
U navedenoj strukturi lokalna promenljiva element u svakoj iteraciji dobija vrednost određenog člana promenljive kolekcija, i u okviru c:forEach taga se obrađuje na određeni način, pri čemu se može koristiti, kao u navedenom primeru, tag c:out koji prikazuje trenutnu vrednost promenljive element.
Ako se pretpostavi da je promenljiva kolekcija niz prethodni primer se može uraditi i pomoću skripleta, ali treba opet napomenuti da ovakav pristup ne odgovara MVC arhitekturi

```
<UL>
<%
  for(int i=0; i<kolekcija.length; i++) {
        String element = kolekcija[i];
%>
<LI><%= element %>
<% } %>
</UL>
```

Pored odstupanja od MVC arhitekture može se primetiti da upotreba c:forEach taga dovodi do konciznijeg i jednostavnijeg koda, a izbegava se i eksplicitna upotreba samog Java programiranja. Navedeni tag c:forEach se može koristiti i za prikaz statičkih kolekcija kao što je slučaj u sledećem primeru

```
<%@ taglib prefix="c"uri="http://java.sun.com/jstl/core"
%>
<UL>
<c:forEach var="i" begin="1" end="10">
  <LI><c:out value="${i}"/>
</c:forEach>
</UL>
```

Nakon izvršavanja datog koda dobija se sledeća stranica



4.7 Primer JSP-servlet MVC aplikacije

U narednom delu teksta prikazaće se kod jednostavne Web aplikacije koja primenjuje tehnologiju koja je razmatrana u prethodnim poglavljima. To znači da će se rezultati na klijentskoj strani prezentovati pomoću JSP stranica, koje će koristi EL i JSTL EL tagove za prikaz rezultata. Poslovna logika, pristup bazi podataka, smeštanje rezultata biće obrađeni u okviru servleta, u formi serverskih komponenti. Na serverskoj strani rezultati će biti zapakovani u okviru odgovarajućeg bean-a, smešteni u određenu oblast važenja i na taj način dostupni klijentskoj strani. Radi jednostavnosti koristiće se Microsoft Access baza podataka koja je povezana sa ODBC drajverom pod imenom obuka.

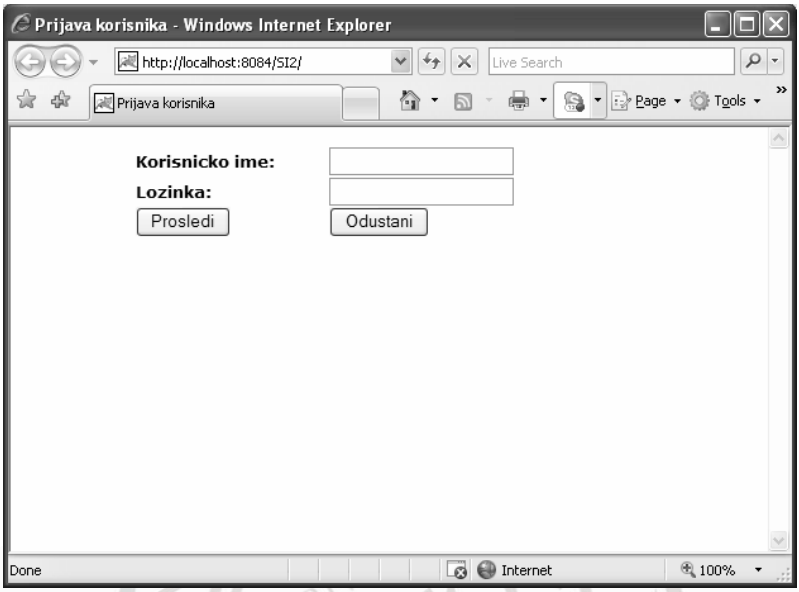
Početna stranica aplikacije je index.html koja sadrži početnu formu za logovanje korisnika:

```
<html>
<head>
<title>Prijava korisnika</title>
<meta http-equiv="Content-Type" content="text/html;
charset=windows-1250">
</head>

<body bgcolor="#FFFFFF" text="#000000">

    <form name="Login" method="post"
action="LogovanjeServlet">
  <table width="400" border="0" cellspacing="0"
cellpadding="0" align="center">
    <tr>
      <td width="153"><b>Korisnicko  ime:</b></td>
      <td width="247">
        <input type="text" name="korisnickoIme">
      </td>
    </tr>
    <tr>
      <td width="153"><b>Lozinka:</b></td>
      <td width="247">
        <input type="password" name="lozinka">
      </td>
    </tr>
    <tr>
      <td width="153">
        <input type="submit" name="prosledi"
value="Prosledi">
      </td>
      <td width="247">
        <input type="reset" name="odustani"
value="Odustani">
      </td>
    </tr>
  </table>
</form>
</body>
</html>
```

Nakon pokretanja date stranice dobija se sledeći prozor



Slika 15. Početni prozor aplikacije

U formi na datoj stranici je definisan poziv akcije LogovanjeServlet. Ova akcija kao i sve ostale akcije su mapirani u web.xml fajlu u odgovarajuće servlete:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.4"
xmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
    <servlet>
        <servlet-name>Logovanje</servlet-name>
        <servlet-class>Logovanje</servlet-class>
    </servlet>
    <servlet>
        <servlet-name>SpisakPredmeta</servlet-name>
        <servlet-class>SpisakPredmeta</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>Logovanje</servlet-name>
        <url-pattern>/LogovanjeServlet</url-pattern>
```

```
    </servlet-mapping>
    <servlet-mapping>
        <servlet-name>SpisakPredmeta</servlet-name>
        <url-pattern>/SpisakPredmeta</url-pattern>
    </servlet-mapping>
    <session-config>
        <session-timeout>
            30
        </session-timeout>
    </session-config>
    <welcome-file-list>
  <welcome-file>
            index.jsp
        </welcome-file>
    </welcome-file-list>
</web-app>
```

Kao što se može videti akcija LogovanjeServlet je mapirana u servlet pod imenom Logovanje. Ovaj servlet treba da prihvati podatke koje je uneo korisnik za svoje korisničko ime i lozinku i da proveri u bazi podataka da li postoji traženi korisnik. U bazi podataka postoji tabela Student sa poljima ID (tipa AutoNumber), korisnickoIme (Text) i lozinka (Text), u kojoj se pamte podaci o korisnicima aplikacije. Ako korsnik postoji njegovi podaci će biti smešteni u bean, a bean će se čuvati u okviru sesije i na taj način biti dostupan svim klijentskim i serverskim stranicama. Bean koji se koristi je nazvan KorisnikBean.java i definisan je u paketu podaci:

```
package podaci;

public class KorisnikBean {
  private String korisnickoIme = "";
  private String lozinka = "";
  public KorisnikBean(){

  }
  public String getKorisnickoIme() {
        return korisnickoIme;
  }
  public void setKorisnickoIme(String korisnickoIme) {
        this.korisnickoIme = korisnickoIme;
  }
  public String getLozinka() {
        return lozinka;
  }
  public void setLozinka(String lozinka) {
```

```
        this.lozinka = lozinka;
    }


}
```

Ovo je jednostavna Java klasa koja predstavlja bean sa dva definisana property-ija (korisnickoIme i lozinka) i njihovim odgovrajućim getter-setter metodama.
Servlet Logovanje.java je:

```
import java.io.*;
import java.util.*;
import java.sql.*;
import javax.servlet.*;
import javax.servlet.http.*;
import podaci.*;

public class Logovanje extends HttpServlet {

String upit="";
String greska="";
String korisnickoIme="";
String lozinka="";


Connection con=null;
Statement stmt=null;
ResultSet RS=null;

boolean prazno=false;

ServletContext sc =null;
RequestDispatcher rd=null;
HttpSession  session=null;

protected void doPost(HttpServletRequest req,
HttpServletResponse res)
    throws ServletException, IOException {
    session=req.getSession(true);
    sc=getServletContext();

if(!req.getParameter("korisnickoIme").equals("")) {

korisnickoIme=req.getParameter("korisnickoIme");
       }
```

```
        else{
        prazno=true;
              greska=greska+"Korisnicko ime nije
prosledjeno!<br>";
  }
   if(!req.getParameter("lozinka").equals("")) {

lozinka=req.getParameter("lozinka");
             }
   else{
             prazno=true;
               greska=greska+"Lozinka nije
prosledjena!<br>";
        }
    try{
        poveziSaBazom(req,res);
        zatvoriKonekciju();
        pozoviJSP(req,res);
    }catch (Exception e) {
        greska="Doslo je do greske! Pokusajte ponovo!";
        prosledi Gresku(greska,req,res);
        greska="";
    }
}
// metoda za povezivanje sa bazom podataka
public void poveziMe(HttpServletRequest req,
HttpServletResponse res){
    try
    {
       Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
       String url="jdbc:odbc:obuka";
        con = DriverManager.getConnection(url);
        stmt=con.createStatement();
  String query1 = "select * from Student where
(korisnickoIme='"+korisnickoIme+"' and
lozinka='"+lozinka+"')";
  RS=stmt.executeQuery(query1);
        if(!RS.next()) {
            greska="Ne postoji korisnik sa unetim
korisnickim imenom i sifrom!";
            prosledi Gresku(greska,req,res);
        }else{
             KorisnikBean kB= new KorisnikBean();
             kB.setKorisnickoIme(korisnickoIme);
             kB.setLozinka(lozinka);
             session.setAttribute("korisnik",kB);
        }
```

```
    }
   catch (Exception e)
    {
  e.printStackTrace();
  greska="Konekcija sa bazom podataka nije
uspostavljena!";
  prosledi Gresku(greska,req,res);
  greska="";
  //prosledjuje se greska jsp stranici koja vraca
gresku...
  }
}

//metoda za zatvaranje konekcije sa bazom podataka
public void zatvoriKonekciju(){
    if (con!=null){
            try {
               con.close();
            }
            catch (SQLException e) {
              e.printStackTrace();
             greska="Greska prilikom zatvaranja
konekcije sa bazom podataka!";
             prosledi Gresku(greska,req,res);
            }
}
}
//pocinje metoda koja prosledjuje odgovarajucu gresku
korisniku
public void prosledi Gresku(String g, HttpServletRequest
req, HttpServletResponse res) {
    HttpSession  session= req.getSession(true);
    session.setAttribute("greska",greska);
        rd = sc.getRequestDispatcher("/Greska.jsp");
        try{
            rd.forward(req,res);
            greska="";
        }
        catch(Exception e){
        }
}
//pocinje metoda koja poziva  JSP stranicu "Login"
public void pozoviJSP(HttpServletRequest req,
HttpServletResponse res){
       try{
        rd =
sc.getRequestDispatcher("/OpstiPodaci.jsp");
```

```
            rd.forward(req,res);
            }
        catch(Exception e){
               greska="nije uspelo "+e;
               prosledi Gresku(greska,req,res);
        }                                                    }
    }
```

doPost metod servleta se sastoji iz tri dela: poveziSaBazom(req,res) – deo u kome se uspostavlja konekcija sa bazom podataka i ispituje se da li postoji uneti korisnik, zatvoriKonekciju() – deo u kome se zatvara konekcija otvorena u prethodnom delu i pozoviJSP(req,res) – deo u kome se poziva sledeća JSP stranica (OpstiPodaci.jsp). U okviru servleta proveravaju se moguće greške i u slučaju da se neka ustanovi dalje izvršavanje aplikacije se prosleđuje stranici Greska.jsp, i to u okviru dela prosledi Gresku(greska,req,res). U ovoj funkciji se lokalna promenljiva greska smešta u sesiju pod istim imenom i na taj način je dostupna sledećoj JSP stranici.

U okviru funkcije poveziSaBazom(req,res) koristi se definisana MS Access baza i njen ODBC drajver i izvršava željeni SQL SELECT upit. Kada se dobije odgovor da korisnik postoji (ako postoji zapis u okviru promenljive tipa ResultSet) formira se instanca klase KorisnikBean pod imenom kB, popuni se dobijenim korisničkim imenom i lozinkom i smesti se u sesiju pod imenom korisnik. Sesija je izabrana za smeštanje ovog podatka, jer se želi da se imenu i šifri korisnika pristupa iz više narednih stranica

Znači, rezultat izvršavanja servleta je jedna od dve JSP strane – OpstiPodaci.jsp, ako je sve u redu, odnosno Greska.jsp u slučaju bilo koje greške.

Kod stranice OpstiPodaci.jsp je sledeći:

```
<html>
<head><title>Opsti podaci</title></head>
<body>
<FORM name='opstiPodaci' method="post"
action="SpisakPredmeta">
<h1>Dobrodosli!</h1> <br />
Vase korisnicko ime je  ${korisnik.korisnickoIme}.
<br>
A vasa lozinka je  ${korisnik.lozinka}.
<br>
Pritisnite dugme da  dobijete spisak predmeta.
<br>
```

```
 <Input type=submit name="predmeti" value="Spisak
predmeta">
</form>
</body>
</html>
```

Na datoj stranici se koristi EL da bi se pristupilo podacima koji su smešteni u sesiji pod imenom korisnik. Taj podatak predstavlja bean sa dva svoja property-ija – korisnickoIme i lozinka i upravo se pristupa ovim property-ijima i prikazuje njihova vrednost.
Forma na datoj stranici ima jednu akciju (SpisakPredmeta) i u prethodno definisanom fajlu web.xml može se primetiti da je ova akcija mapirana u servlet SpisakPredmeta.java. U slučaju logovanja korisnika nbosko sa lozinkom nb, koji postoji u nazi, dobiće se sledeći ekran:



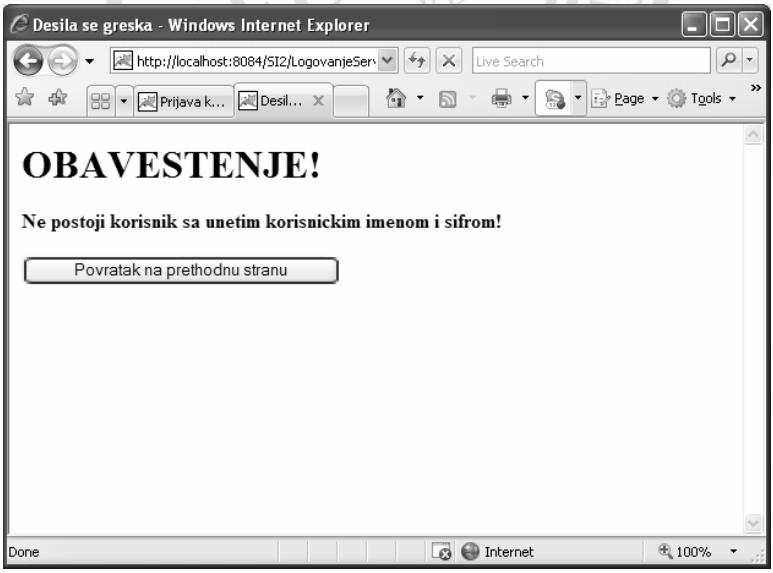
Slika 16. Prozor koji se dobija nakon uspešnog logovanja

Kod stranice Greska.jsp je sledeći:

```
<html>
    <head>
        <meta http-equiv="Content-Type"
content="text/html; charset=UTF-8">
        <title>Desila se greska</title>
    </head>
    <body>

    <h1>OBAVESTENJE!</h1>
    <b>
        ${greska}
    </b><br><br>
    <input type="button" onclick="history.back()"
value="Povratak na prethodnu stranu">
    </body>
</html>
```

Opet se koristi EL da bi se prišlo određenoj promenljivoj, u ovom slučaju to je promenljiva greska, koja sadrži opis greške i koja je takođe smeštena u okviru sesije. Ekran koji se dobije ako korisnik pokuša da pristupi sa neodgovarajućim korisničkim imenom ili šifrom je:



Slika 17. Prozor koji se dobija nakon neuspešnog logovanja

Navedeno je da je u slučaju ispravnog rada sledeća komponenta koja se izvršava servlet SpisakPredmeta.java. Ovaj servlet dobijene rezultate smešta u okviru instance bean-a PredmetiBean.java, koji se takođe nalazi u okviru paketa podaci:

```
package podaci;
import java.util.*;

public class PredmetiBean {
private Vector spisakPredmeta;
private int brojPredmeta = 0;

public Vector getSpisakPredmeta() {
  return spisakPredmeta;
}

public void setSpisakPredmeta(Vector spisakPredmeta) {
  this.spisakPredmeta = spisakPredmeta;
}

public int getBrojPredmeta() {
  return brojPredmeta;
}

public void setBrojPredmeta(int brojPredmeta) {
  this.brojPredmeta = brojPredmeta;
}

}
```

Ovaj bean ima 2 property-ija – spisakPredmeta, koji je kolekcija tipa Vector i sadržaće sve predmete koji pripadaju određenom korisniku, i brojPredmeta, koji je tipa int i predstavlja broj predmeta koji pripadaju određenom korisniku. Treba napomenuti da drugi property nije neophodan, ali je uveden radi jednostavnijeg prikaza na JSP stranici koja će prikazivati rezultate.
Kod servleta SpisakPredmeta.java je:

```
import java.io.IOException;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
```

```
import java.util.*;
import javax.servlet.RequestDispatcher;
import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;

import podaci.*;

 public class SpisakPredmeta extends
javax.servlet.http.HttpServlet implements
javax.servlet.Servlet {
   String upit="";
   String greska="";
   String korisnickoIme="";
   String lozinka="";


   Connection con=null;
   Statement stmt=null;
   ResultSet RS=null;

   boolean prazno=false;

   ServletContext sc =null;
   RequestDispatcher rd=null;
   HttpSession  session=null;
  protected void doGet(HttpServletRequest request,
HttpServletResponse response) throws ServletException,
IOException {
        doPost(request,response);
  }

  protected void doPost(HttpServletRequest request,
HttpServletResponse response) throws ServletException,
IOException {
     session=request.getSession(true);
      sc=getServletContext();
        KorisnikBean kB= new KorisnikBean();
        kB = (KorisnikBean)
session.getAttribute("korisnik");
        korisnickoIme =kB.getKorisnickoIme();
        lozinka = kB.getLozinka();

         try
         {
```

```
        PredmetiBean pB= new PredmetiBean();
         Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
        String url="jdbc:odbc:obuka";
        con = DriverManager.getConnection(url);
             stmt=con.createStatement();
        String query1 = "select * from Predmeti where
(korisnickoIme='"+korisnickoIme+"' and
lozinka='"+lozinka+"')";

        RS=stmt.executeQuery(query1);
        Vector predmeti = new Vector();
         String opis="";
        while(RS.next()) {
              String s = RS.getString("Opis");
                 predmeti.addElement(s);
             }
        pB.setSpisakPredmeta(predmeti);
        pB.setBrojPredmeta(predmeti.size());
        request.setAttribute("predmeti",pB);
         rd = sc.getRequestDispatcher("/Prikaz.jsp");
         try{
             rd.forward(request,response);
             greska="";
         }
         catch(Exception e){
              System.out.println(e);
         }

         }
        catch (Exception e)
         {
         e.printStackTrace();
         greska="Konekcija sa bazom podataka nije
uspostavljena!";
        prosledi Gresku(greska,request,response);
        greska="";
        //prosledjuje se greska jsp stranici koja vraca
gresku...
       }

  }
  public void zatvoriKonekciju(){
      if (con!=null){
              try {
                 con.close();
              }
              catch (SQLException e) {
```

```
                e.printStackTrace();
                   greska="Greska prilikom zatvaranja
konekcije sa bazom podataka!";
                   prosledi Gresku(greska,req,res);
             }
  }
  }

//      metoda koja prosledjuje odgovarajucu gresku
korisniku
  public void prosledi Gresku(String g, HttpServletRequest
req, HttpServletResponse res) {

  HttpSession  session=null;

  session=req.getSession(true);

      session.setAttribute("greska",greska);
          rd = sc.getRequestDispatcher("/Greska.jsp");
          try{
              rd.forward(req,res);
              greska="";
          }
          catch(Exception e){
          }
  }

}
```

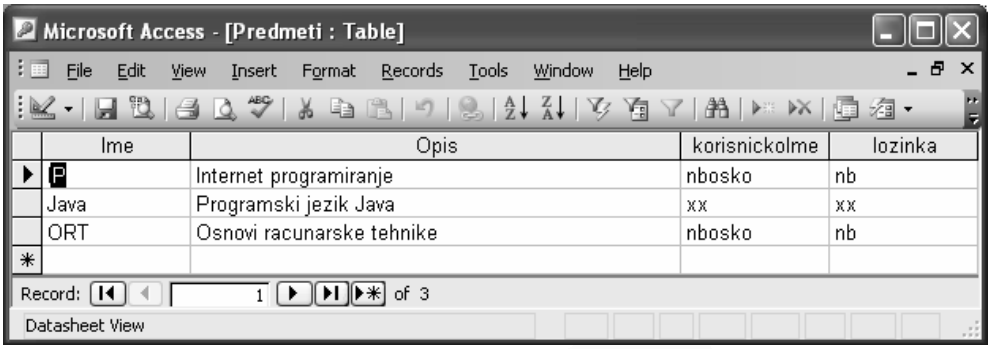
U okviru doPost metoda se prvo iz sesije uzimaju podaci korisnika (korisničko ime i lozinka) koji su tu smešteni kada se ustanovilo da se definisani korisnik nalazi u bazi podataka. Nakon toga se izvršava SQL SELECT upit koji u promenljivu tipa ResultSet smešta sve zapise iz tabele Predmeti zadovoljavaju traženi kriterijum. Tabela Predmeti ima četiri polja koja su tipa Text – Ime, Opis, korisnickoIme i lozinka. Treba napomenuti da je ovakva struktura tabele izabrana radi jednostavnosti, i da bi se čitaoci usredsredili na Internet tehnologije, a ne na sam rad sa bazom podataka. Kada se upit izvrši tada se formira nova instanca bean-a PredmetiBean. Navedeno je da u okviru ovog bean-a postoji property koji je kolekcija i on se koristi da bi se sačuvala imena predmeta. Zato se prolazi kroz dobijene zapise i vrednost kolone Opis smešta u lokalnu promenljivu predmeti koja je tipa Vector. Kada se obrade svi zapisi, lokalna promenljiva se dodeli property-iju napravljenog bean-a, kao i property koji označava broj

predmeta. Na kraju se dobijeni bean zapamti u okviru zahteva pod imenom predmeti.
Kao i u slučaju prethodnog servleta, i servlet SpisakPredmeta izvršavanje može nastaviti na dva načina, prosleđivanjem daljeg rada stranici Prikaz.jsp ako je sve u redu, odnosno stranici Greska.jsp u slučaju da je došlo do neke greške. Stranica Greska.jsp je razmatrana u prethodnom delu teksta, dok je kod stranice Prikaz.jsp:

```
<%@taglib uri="http://java.sun.com/jsp/jstl/core"
prefix="c"%>

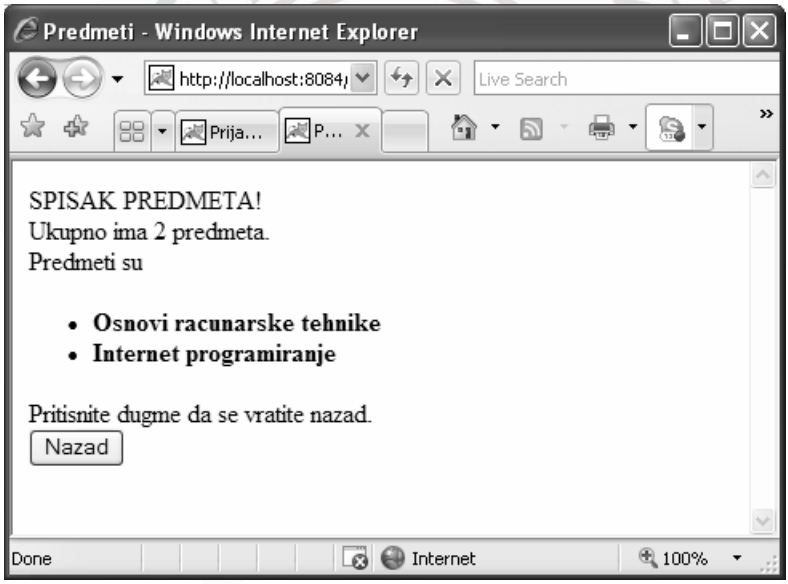
<html>
<head><title>Predmeti</title></head>
<body>
    SPISAK PREDMETA!
<br>
Ukupno ima ${predmeti.brojPredmeta} predmeta. <br>
Predmeti su <ul>
<c:forEach var="opis"
items="${predmeti.spisakPredmeta}">
    <li><b> <c:out value="${opis}"/> </b></li>
</c:forEach>
</ul>
Pritisnite dugme da  se vratite nazad.
<br>
 <Input type=button name="predmeti" value="Nazad"
onClick="history.back()">
</form>
</body>
</html>
```

Na datoj stranici kao i u prethodnoj JSP stranici koristi se EL da bi se pročitala vrednost property-ija brojPredmeta koji se nalazi uz okviru bean-a predmeti. Ali u okviru ove stranice potrebno je pristupiti spisku imena predmeta, t.j. složenom property-iju koji predstavlja kolekciju. Za pristup ovoj kolekciji koristi se JSTL EL tag c:forEach koji prolazi kroz kolekciju i u svakoj iteraciji pojedinačne članove smešta u lokalnu promenljivu opis, koju pomoću c:out prikazuje na ekranu.
Neka je dat sledeći sadržaj tabele Predmeti u okviru korišćene baze podataka



Slika 18. Izgled tabele Predmeti

Za korisnika nbosko dobija se sledeći ekran



Slika 19. Prikaz predmete određenog korisnika

5. JAVASERVER FACES

JavaServer Faces (JSF) je poslednjih godina najdominantnija tehnologija Java Internet programiranja. Cilj ovog alata je automatizacija i jednostavnost izvršavanja najkorišćenijih operacija izrade ove vrste aplikacije. Glavna odlika ove tehnologije je da se sastoji od sledećih delova:

- skup ugrađenih ulazno-izlaznih komponenti
- događajima vođen (event driven) programski model, sa opcijama za obradu i reagovanje na događaje
- model komponenti koji omogućava programerima serverske strane razvoj dodatnih komponenti

Neke od JSF komponenti su jednostavne, kao na primer ulazna polja ili dugmad. Druge su dosta sofisticiranije, na primer tabele podataka i stabla. JSF sadrži sav potreban kod za obradu događaja i organizaciju komponenti. Aplikativni programeri mogu da zanemare sve nepotrebne detalje i da se usredsrede na sam razvoj aplikacione logike.

Takođe, treba napomenuti da je JSF danas deo Java EE standarda, što znači da je uključena u svaki Java EE aplikacioni server, i da se može jednostavno dodati na Web servere, kao što je Jakarta Tomcat. Tada je potrebno koristiti i dodatne JSF biblioteke (videti na http://javaserverfaces.dev.java.net).

5.1 Struktura JSF aplikacije

JSF aplikacija se instaliraju u obliku WAR fajla – zapakovane aplikacije sa ekstenzijom war i strukturom direktorijuma sa sledećim standardizovanim rasporedom:

```
project-name
      HTML i JSP stranice
      /WEB-INF
            konfiguracioni fajlovi
            /classes
                  Javine klase
            /lib
                  Dodatne biblioteke
```

U okviru predložene Javine BluePrints dokumentacije (videti na http://java.sun.com/blueprints/code/projectconvertions.com) se preporučuje malo promenjena struktura. Preporučuje se da se i sam kod čuva u

direktorijumu src/java , a da se JSF stranice i konfiguracioni fajlovi čuvju u web direktorijumu.

```
project-name
   /source
  /java
        /com
              /mojPaket
                    /korisnikBean.java
   /web
  Html, jsp, …
       /WEB-INF
          /classes
          /lib
             XML fajlovi
```

Detaljan opis date strukture je dat u sledećoj tabeli.

Direktorijum ili fajl	Objašnjenje
project-name	Direktorijum projekta koji se zove kao i projekat
/source	Ovde se smeštaju java source klase i .properties fajlovi
/web	Ovde se nalaze fajlovi web aplikacije koje koristi server ili servlet container.
/WEB-INF	Ovaj folder sadrži fajlove web aplikacije koji se koriste u runtime-u ali su sakriveni od browser-a
/classes	Ovde se nalaze kompajlirane java klase kao i .proprties fajlovi iskopirani iz JavaSource-a
/lib	Ovde se nalaze biblioteke potrebne za rad aplikacije

5.2 Servisi JSF framework-a

Ako se razmatra JSF arhitekture na visokom nivou može se primetiti da je ovaj framework odgovoran za interakciju sa klijentskim mašinama i sadrži alate koji omogućavaju zajedničko izvršavanje vizuelne prezentacije, aplikacione logike i poslovne logike za određenu Internet aplikaciju. Ipak, oblast JSF-a je ograničena na prezentacioni nivo. Perzistencija baze podataka, Web servisi i ostale backend konekcije su izvan oblasti kojom se bavi JSF.

Najvažniji servisi koje JSF obezbeđuje su:

- MVC arhitektura – sve softverske aplikacije omogućavaju korisnicima da manipulišu sa određenim podacima, kao što su karte za kupovinu, informacije o putovanju, ili bilo koji drugi potrebni podaci za određeni problem. Ovi podaci se nazivaju model (Model). Onaj koji razvija softver proizvodi poglede (Views) na dati model podataka. Kod Internet aplikacija, HTML se koristi da bi se dobili traženi pogledi. JSF omogućava konekciju između pogleda i modela. Na primer, komponenta pogleda se može povezati sa property-ijem Bean-a na sledeći način

```
<h:inputText value="#{korisnik.ime}" />
```

  Ovaj kod dovodi do toga da kada korisnik klikne na dato dugme i forma se pošalje na server, JSF implementacija poziva metod check() koji se nalazi u okviru Bean-a korisniik. Ovaj metod može da izvrši validacione akcije i da izvrši promene modela, kao i da vrati navigacioni ID koji definiše koja će se stranica sledeća prikazati.
- Konverzija podataka – korisnik unosi podatke u okviru forme u obliku teksta. Objektima povezanim sa poslovnom logikom potrebni su podaci u obliku brojeva, datuma, ili nekog drugog tipa podataka. JSF omogućava da se na jednostavan način specificiraju i primene potrebna pravila konverzije.
- Validacija i obrada grešaka – JSF omogućava jednostavno povezivanje validacionih pravila za sva ulazna polja. Na primer „vrednost u ovo polje je neophodno uneti“, „vrednost ovog polja mora biti broj“,... Takođe. Kada korisnik unese neodgovarajući podatak, potrebno je prikazati odgovarajuću poruku. JSF umnogome olakšava ovu potrebu.
- Internacionalizacija – JSF sadrži razne opcije za kodiranje posebnih karaktera i selekciju po

- moćnih resursa koje se koriste.
- Dodatne komponente – onaj deo tima koji razvija potrebne komponente može veoma kompleksne i složene komponente da pruži na raspolaganje dizajnerima, koji mogu da ih na jednostavan način prikažu na stranici. Na primer može da se razvije komponenta kalendar sa uobičajenom zvučnom podrškom. Tada se može kodirati na stranici sa

```
<acme:kalendar value="#{let.dolazak}" poecetak="Pon"
/>
```

- Alternativni prikazi – definisano ponašanje JSF je da generiše tagove za HTML stranice. Ali, veoma je jednostavno nadograditi JSF framework da proizvodi oznake za neki drugi opisni jezik, kao što je WML, XUL.

Iako JSF prirodno koristi JSP kao prezentacionu tehnologiju - nije ograničen samo na JSP. Kako se JSF API sloj nalazi direktno iznad Servlet API sloja to omogućava da se neka druga prezentaciona tehnologija koristi umesto JSP-a .

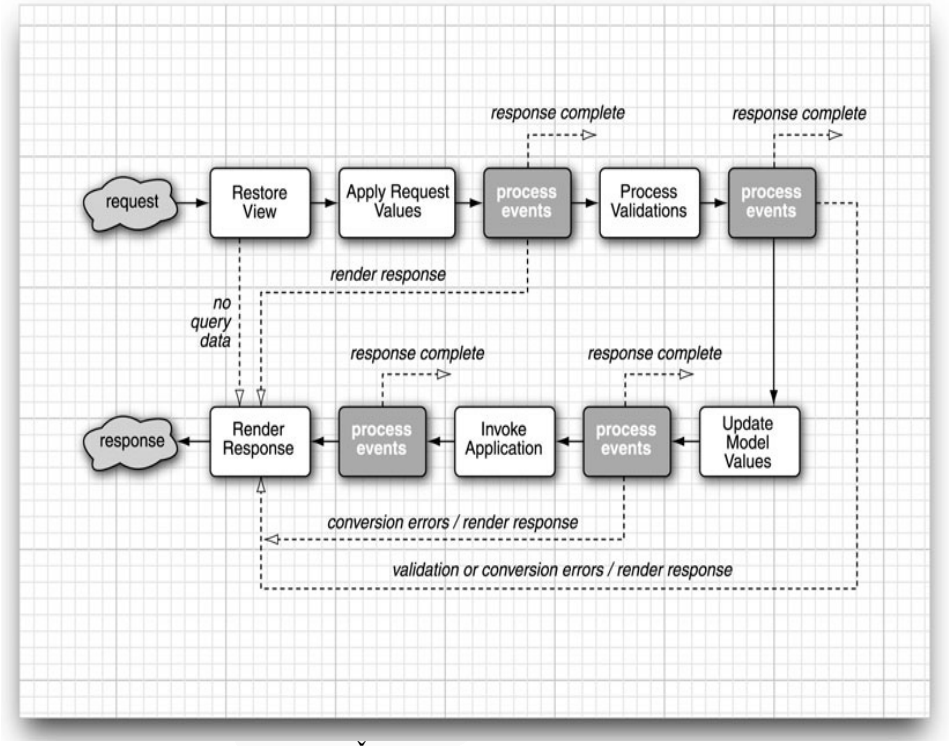
Da bi mogli da se koriste JSF tagovi svaka JSP strana na vrhu mora imati sledeće dve taglib direktive:

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h"
%>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f"
%>
```

Prva uključuje html_basic, a druga jsf_core tagove. html_basic tag-ovi predstavljaju kontrole HTML forme i druge osnovne HTML elemente i prikazuju podatke ili prihvataju podatke od korisnika (column, commandButton, commandLink, form, message, outputText, itd.). jsf_core tagovi se koriste za ključne akcije koje su nezavisne od određenog render-a (na primer tagovi za rad sa događajima (actionListener), tagovi za konverziju podataka (converter, convertDateTime, convertNumber), tagovi za validaciju (validator, validateLength), loadBundle, param, subview, view itd.).

5.3 Životni ciklus

JSF specifikacija definiše 6 različitih faza prilikom izvršavanja određenog poziva. Uobičajeni tok izvršavanja je prikazan sa punim linijama, alternativni su prikazani isprekidanim.



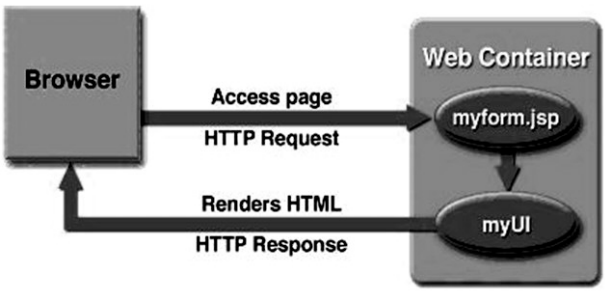
Slika 20. Životni ciklus JSF aplikacija

Faza *Restore View* vraća stablo komponenti, za stranicu koja je ranije prikazivana, odnosno pravi novo stablo komponenti ako se stranica prikazuje prvi put. Ako je stranica ranije prikazivana, sve komponente se postavljaju na početno stanje. To znači da JSF automatski postavlja informacije o traženoj formi. Na primer kada korisnik unese i pošalje određene nelegalne podatke, koji se neće propustiti na dalju obradu tokom njihove provere, ulazni podaci se korisniku ponovo prikazuju, tako da

korisnik može da ih ispravi. Ako zahtev ne sadrži ulazne podatke, JSF implementacija prelazi na *Render Response* fazu. Ovo se događa kada se stranica prikazuje prvi put.
Sledeća faza je *Apply Request Values.* U okviru ove faze JSF implementacija prolazi kroz objekte komponenti u okviru stabla komponenti. Za svaki objekat komponenti se proverava koja mu vrednost pripada i ona mu se dodeljuje. U ovoj fazi se dodaju događaji vezani za dugme ili link u red događaja.
U okviru *Process Validation* poslati stringovi se prvo prebacuju u „lokalne" vrednosti, koji mogu biti objekti bilo kog tipa. Kada se diznira JSF stranica, mogu se za klijentsku stranicu vezati validatori koji izvršavaju proveru korektnosti lokalnih vrednosti. Ako je validacija uspešna, JSF ciklus se nastavlja normalno. Kada se desi greška JSF implementacija poziva *Render Response* fazu direktno, prikazujući ponovo traženu stranicu, tako da se korisniku omogući korektan unos. Može se prikazati i odgoavarjuća poruka, koja korisniku objašnjava zašto ponovo vidi iste podatke.
Kada se izvrše konverzije i validacije, pretpostavlja se da je bezbedno promeniti model podataka. Tokom *Update Model Values* faze, lokalne vrednosti se koriste da bi se promenile vrednosti u Bean-ovima povezanim sa komponentama.
U okviru *Invoke Application* faze izvršava se metod action() dugmeta ili linka koji je doveo do slanja forme. U ovom metodu se mogu izvršiti dodatne aplikacione obrade. Takođe, tu se generiše izlazni string, koji se šalje delovima odgovornim za navigaciju, gde se izvršava poziv nove stranice.
Na kraju, *Render Response* faza dekodira odgovor i šalje ga klijentu. Kada korisnik sa nove stranice pošalje novu formu, klikne na link ili na drugi način generiše novi zahtev, startuje se novi ciklus.
Naglašeno je da JSF pojednostavljuje razvojni proces obezbeđujući komponentno-centričan pristup razvoju java web user intreface-a. Glavni delovi ovog framework-a su:

- API koji omogućava predstavljanje UI komponenti i manipulisanje njihovim stanjima, upravljanje događajima, validacije na strani servera, konverziju podataka, definisanje navigacije između stranica, internacionalizaciju.
- Dve JSP tag biblioteke za predstavljanje UI komponenti u okviru JSP strane i za povezivanje komponenti sa serverskim objektima.

Kao što je prikazano na slici korisnički interfejs koji je kreiran pomoću JSF tehnologije (*myUI* na slici) se izvršava na serveru i renderuje nazad klijentu.

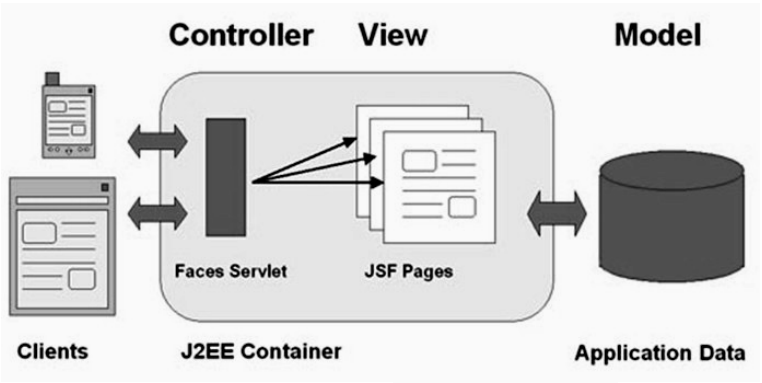


Slika 21. Ulaz-Izlaz koji se izvršava na serveru

myform.jsp je JSF stranica tj. JSP stranica koja u sebi sadrži JSF tagove. Ona prikazuje UI komponente koristeći tagove definisane JavaServer Faces tehnologijom. UI za web aplikaciju (*myUI*) manipuliše objektima koje koristi JSP strana. Ovi objekti uključuju:

- UI component objekte koji se mapiraju na tag-ove na JSP strani
- Event lintener-e, validator-e, i converter-e koji su registrovani na komponente
- Objekte koji enkapsuliraju podatke i funkcionalnosti komponente specifične za aplikaciju

JSF framework prati Model-View-Controller (MVC) arhitekturu što JSF aplikacije čini dosta prilagodljivijm i pogodnijim za rad pošto je user-interface kod (*View*) razdvojen od aplikacionih podataka i logike (*Model*). JSF servlet (*Controller*) upravlja svom korisničkom interakcijom sa aplikacijom. On priprema JSF kontekst koji omogućava stranicama pristup aplikacionim podacima kao i zaštitu od neovlašćenog i neadekvatnog pristupa stranicama.



Slika 22 JavaServer Faces implementacija MVC-a

5.4 Navigacija

Navigacija je jedna od najbitnijih osobina JSF paketa. Navigacija za neku aplikaciju se definiše u okviru faces-config.xml konfiguracionog fajla:

```
<navigation-rule>
  <from-view-id>/pages/inputname.jsp</from-view-id>
  <navigation-case>
    <from-outcome>greeting</from-outcome>
    <to-view-id>/pages/greeting.jsp</to-view-id>
  </navigation-case>
</navigation-rule>
```

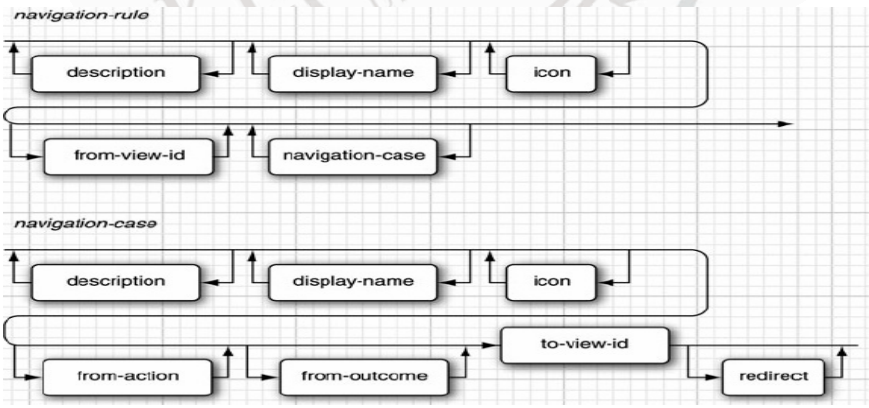
Definisano je kako će se sa strane inputname.jsp (definisane u okviru from-view-id elementa) preći na stranu greeting.jsp (definisane u okviru to-view-id elementa). navigation-rule može imati proizvoljan broj navigation-case-ova od kojih svaki definiše koja se strana sledeća učitava bazirano na logičkom ishodu (definisanom u okviru from-outcome).
Ishod može biti definisan od strane action atributa UICommand komponente koja vrši submit forme:

```
<h:commandButton action="greeting"
value="#{msg.button_text}" />
```

Ishod se takođe može dobiti kao povratna vrednost akcione metode pratećeg bean-a. Ovaj metod vrši neki proces da bi utvrdio ishod. Na primer metod može da proveri da li je lozinka koju je korisnik uneo ispravna. Ako jeste metod vraća rezultat success, a u suprotnom slučaju failure.
U prvom slučaju korisnik bi bio usmeren na novu stranicu u okviru aplikacije, a u drugom bi se ponovo učitala login strana sa porukom o grešci.
Ako se pažljivo biraju stringovi, moguće je skupiti višestruka pravila navigacije na jedno mesto. Ako se želi da se akcija prikaz nalazi u više različitih JSF stranica i da iz svake poziva stranicu prikaz.jsp, tada se unosi sledeće pravilo navigacije:

```
<navigation-rule>
<navigation-case>
<from-outcome>logout</from-outcome>
<to-view-id>/logout.jsp</to-view-id>
</navigation-case>
</navigation-rule>
```

Ovo pravilo se odnosi na sve stranice, zato što nijedan from-view-id element nije definisan. Tehnike koje su prikazane su dovoljne za većinu aplikacija. Ali mogu se primeniti i druga pravila za navigacione elemente u okviru faces-config.xml fajla.



Slika 22. Pravila navigacije u JSF aplikacijama

Ako se unese redirect element posle to-view-id elementa, JSF prekida trenutni zahtev i šalje HTTP redirekt zahtev klijentu, što govori klijentu koji URL da koristi za sledeću stranicu.

Preusmerenje stranice je sporije u odnosu na prosleđivanje, jer se aktivira još jedan ciklus sa browser-om, ali dopušta browser-u da izmeni vrednost svog adresnog polja.

```
<navigation-case>
<from-outcome>success</from-outcome>
<to-view-id>/success.jsp</to-view-id>
<redirect/>
</navigation-case>
```

Bez redirekcije, originalni URL (localhost:8080/javaquiz/index.faces) ostaje nepromenjen kada se sa /index.jsp pređe na stranicu /success.jsp. Sa redirekcijom, browser prikazuje novi URL (localhost:8080/javaquiz/success.faces).
Primer sa osnovnom statičkom navigacijom je sledeći
Početna forma je u sledećem formatu:

```
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f"
%>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h"
%>
<f:view>
HTML markup
<h:form>
HTML markup and h:blah tags
</h:form>
HTML markup
</f:view>
```

Pri kreiranju početne forme se koriste h:form elementi. Kod njih je atribut ACTION označabva automatski isti fajl (trenutni URL), a METHOD je automatski POST.
Unutar h:form se mogu koristiti specijalni tagovi da bi se prikazali ulazni elementi:

- h:inputText odgovara <INPUT TYPE="TEXT">
- h:inputSecret odgovara <INPUT TYPE="PASSWORD">
- h:commandButton odgovara <INPUT TYPE="SUBMIT">

U kasnijim naprednijim primerima ulazni elementi će se povezivati sa propertie-ijima bean-a. U ovom primeru za jednostavnu statičku navigaciju, specificira se String kao akcija za h:commandButton, pri čemu String mora da odgovara pravilima navigacije iz faces-config.xml. Primer ove stranice je:

```
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f"
%>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h"
%>
<f:view>
<!DOCTYPE …>
<HTML>
<HEAD>…</HEAD>
<BODY>
<CENTER>
<TABLE BORDER=5>
<TR><TH CLASS="TITLE">New Account Registration</TH></TR>
</TABLE>
<P>
<h:form>
Email address: <h:inputText/><BR>
Password: <h:inputSecret/><BR>
<h:commandButton value="Sign Me Up!" action="register"/>
</h:form>
</CENTER></BODY></HTML>
</f:view>
```

U datom primeru koristi se action atribut u okviru h:commandButton elementa. U većini realnih slučajeva, specificira se metod koji se poziva ka da se forma prosledi. Metod vraća različite string-ove, i faces-config.xml mapira stringove u izlazne stranice
Za statičku navigaciju, specificira se jednostavan string, a faces-config.xml mapira taj string u izlaznu stranicu. U navedenom primeru to je deo koda

```
<h:form>
Email address: <h:inputText/><BR>
Password: <h:inputSecret/><BR>
<h:commandButton value="Sign Me Up!" action="register"/>
</h:form>
```

Deklarišu se mapiranja u faces-config.xml

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE faces-config PUBLIC …>
<faces-config>
<navigation-rule>
<from-view-id>/blah.jsp</from-view-id>
<navigation-case>
<from-outcome>some string</from-outcome>
<to-view-id>/WEB-INF/results/something.jsp</to-view-id>
</navigation-case>
</navigation-rule>
</faces-config>
```

U datom primeru ovaj fajl će izgledati

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE faces-config PUBLIC …>
<faces-config>
<navigation-rule>
<from-view-id>/register1.jsp</from-view-id>
<navigation-case>
<from-outcome>register</from-outcome>
<to-view-id>/WEB-INF/results/result1.jsp</to-view-id>
</navigation-case>
</navigation-rule>
</faces-config>
```

I na kraju izlazna stranica, koja se nalazi u …/jsf-test/WEB-INF/results/result1.jsp, izgleda

```
<!DOCTYPE …>
<HTML>
<HEAD>…</HEAD>
<BODY>
<CENTER>
<TABLE BORDER=5>
<TR><TH CLASS="TITLE">Uspesno</TH></TR>
</TABLE>
<H2>Uspesno ste se registrovali.<BR>
(Ver 1)
</H2>
</CENTER>
</BODY>
</HTML>
```

U sledećem primeru se prikazuje dinamička navigacija. U ovom slučaju kada se početna forma pošalje, u zavisnosti od realizovane poslovne logike i unetih podataka. jedan od dva rezultat se prikazuje ili je unos korisnika prihvaćen ili se korisnik odbija. Sada postoje određeni koraci koji se nisu primenjivali u prethodnom primeru.
Prvi je da se specificira metod koji će se izvršiti na serverskoj strani (controller) pomoću

```
#{controllerName.methodName}
```

Na primer

```
<h:commandButton value="Sign Me Up!"
action="#{healthPlanController.register}"/>
```

Navedeni metod kao rezultat generisaće String čije vrednosti odgovaraju odgovaraju uslovima, koji su definisani u okviru faces-cofig.xml fajlova. U slučaju da se kao rezultat ne generiše konkretna vrednost, već se prosledi vrednost null, početna forma se ponovo prikazuje. Ovde treba napomenuti da sam controller, t.j. klasa koja se izvršava na serverskoj strani i čiji metod se poziva ne zahteva da nasledi neku posebnu klasu. Kod Struts framework-a princpi je drugačiji i tamo komponenta koja je ekvivalent controller-u mora danasledi tačno određenu klasu (Action) iz Struts paketa.
I ovde je potrebno u okviru faces-config.xml fajla deklarisati pomenutu serversku komponentu koja će se pozvati sa klijentske strane

```
<faces-config>
<managed-bean>
<managed-bean-name>controller name</managed-bean-name>
<managed-bean-class>controller class</managed-bean-
class>
<managed-bean-scope>request</managed-bean-scope>
</managed-bean>
</faces-config>
```

Sledeći korak je da se definiše više ulaza pravila navigacije u okviru faces-config.xml fajla, i to po jedan za svaki mogući String koji vraća controller. I u ovom slučaju ako se ne pronađe odgovarajuće poklapanje, forma se ponovo prikazuje.

Sada početna klijentska strana ima sledeći izgled:

```
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f"
%>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h"
%>
<f:view>
…
<h:form>
First name: <h:inputText/><BR>
Last name: <h:inputText/><BR>
...
<h:commandButton value="Sign Me Up!"
action="#{healthPlanController.signup}"/>
</h:form>…
</f:view>
```

Navedeno je da pri realizaciji serverske komponente (controller-a) treba dizajnirati odgovarajući metod, koji je naveden na klijentskoj strani, ne prihvata nijedan argument i vraća String kao rezultat, po ranije definisanim pravilima. U praksi većina controller-a pristupa parametrima klijentskog zahteva preko bean-ova, ali će se taj pristup primeniti u kasnijim primerima. Za sada, poslovna logika će biti nezavisna od ulaznih podataka, t.j. u ovom konkretnom primeru na osnovu slučajnog broja će se vraćati rezultat da je logovanje uspelo ili nije. Kako je u prethodnim stranicama definisano da se poziva objekat healthPlanController i njegov metod signup, potrebno je realizovati navedenu klasu i metod. U ovom slučaju metod vraća dve vrednosti “accepted” i “rejected”:

```
package coreservlets;
public class HealthPlanController {
public String signup() {
  if (Math.random() < 0.2) {
        return("accepted");
  } else {
        return("rejected");
  }
}
}
```

U okviru faces-config.xml fajla potrebno je mapirati klasu coreservlets.HealthPlanController u objekat healthPlanController:

```
…
<faces-config>
<managed-bean>
<managed-bean-name>
healthPlanController
</managed-bean-name>
<managed-bean-class>
coreservlets.HealthPlanController
</managed-bean-class>
<managed-bean-scope>request</managed-bean-scope>
</managed-bean>
…
</faces-config>
```

Pored mapiranja controller-a u ovom fajlu potrebno je mapirati i dobijene rezultate u izlazne JSP stranice, t.j. potrebno je specificirati pravila navigacije koja se moraju se poklapati sa vrednostima koje šalje controller u ovom slučaju sa vrednostima “accepted” i “rejected”

```
…
<faces-config>
…
<navigation-rule>
<from-view-id>/signup.jsp</from-view-id>
<navigation-case>
<from-outcome>accepted</from-outcome>
<to-view-id>/WEB-INF/results/accepted.jsp</to-view-id>
</navigation-case>
<navigation-case>
<from-outcome>rejected</from-outcome>
<to-view-id>/WEB-INF/results/rejected.jsp</to-view-id>
</navigation-case>
</navigation-rule>
…
</faces-config>
```

Poslednji korak je realizacija izlaznih strana, u ovom slučaju postoje dve takve stranice - …/jsf-test/WEB-INF/results/accepted.jsp:

```
<!DOCTYPE …>
<HTML>
<HEAD>…</HEAD>
<BODY>
<CENTER>
<TABLE BORDER=5>
<TR><TH CLASS="TITLE">Accepted!</TH></TR>
</TABLE>
<H2>You are accepted into our health plan.</H2>
Congratulations.
</CENTER>
</BODY>
</HTML>
```

I …/jsf-test/WEB-INF/results/rejected.jsp:

```
<!DOCTYPE …>
<HTML>
<HEAD>…</HEAD>
<BODY>
<CENTER>
<TABLE BORDER=5>
<TR><TH CLASS="TITLE">Rejected!</TH></TR>
</TABLE>
<H2>You are rejected from our health plan.</H2>
Get lost.
</CENTER>
</BODY>
</HTML>
```

U dosadašnjim primerima na serverskoj strani nije se pristupalo podacima koji se šalju sa klijentske strane. To je moguće realizovati na više načina. Jedna od mogućnosti je prihvatanje podataka iz request objekta koji predstavlja zahtev. Da bi se realizovala navedena operacija potrebno je pristupiti posebnom objektu ExternalContext-u, koji omogućava pristup standardnim servlet i JSP objektima (request, response, ServletContext, session, ...).

Ovaj objekat se može koristiti i u konkretnim JSF stranicama za:

- čitanje request header-a
- čitanje cookie-ija
- postavljanje response header-a ili status kodova
- pristup promenljivama koje nisu povezane sa form bean-om

Za čitanje parametra poslatih sa klijentske strane potrebno je navesti sledeće naredbe

```
ExternalContext context =
FacesContext.getCurrentInstance().getExternalContext();
HttpServletRequest request =
(HttpServletRequest)context.getRequest();
String param1 = request.getParameter("imeParametra");
```

Početna forma se realizuje kao u prethodnom primeru, ali je potrebno na neki način definisati elemente kojima se želi pristupiti sa serverske strane. Zato u novom primeru element h:form ima id atribut, kao i elementi h:inputText i h:inputSecret. Sada se trenutnom request parametru pristupa sa formID:elementID. Ipak treba naglasiti da je ovakav pristup potreban samo u slučaju kada se ručno čitaju parametri zahteva! Osnovni izgled stranice je

```
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f"
%>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h"
%>
<f:view>
…
<h:form id="form2">
Email address: <h:inputText id="email"/><BR>
Password: <h:inputSecret id="password"/><BR>
<h:commandButton value="Sign Me Up!"
action="#{registrationController.register}"/>
</h:form>…
</f:view>
```

Kao što se može primetiti, akcija u okviru komandnog dugmeta je definisana kao u prethodnim primerima.
Pozvana akcija se realizuje kao serverska komponenta, prihvata podatke koji su poslati u okviru zahteva na već navedeni način i realizuje određenu logiku. U ovom konkretnom slučaju vrše se dve provere. Za email adresu proverava se da li uopšte postoji podatak, da li ima više od 3 karaktera i da li

sadrži karakter. Što se tiče polja za šifru, proverava se da li postoji i da li ima bar 6 karaktera. U zavisnosti od rezultata provera mogući izlazi su bad-address – ako email adresa ne zadovoljava postavljene uslove, bad-password – ako šifra ne zadovoljava postavljene uslove, i success – ako je sve u redu.

Kompletan kod date klase je

```
import javax.servlet.*;
import javax.servlet.http.*;
import javax.faces.context.*;
public class RegistrationController {
public String register() {
ExternalContext context =
FacesContext.getCurrentInstance().getExternalContext();
HttpServletRequest request =
(HttpServletRequest)context.getRequest();
String email = request.getParameter("form2:email");
String password =
request.getParameter("form2:password");
if ((email == null) || (email.trim().length() < 3) ||
                    (email.indexOf("@") == -1)) {
        return("bad-address");
  } else if ((password == null) ||
(password.trim().length() < 6)) {
return("bad-password");
} else { return("success");
}
}
}
```

Definisanu klasu treba definisati u faces-config.xml fajlu i to u okviru objekta zahteva:

```
…
<faces-config>
<managed-bean>
<managed-bean-name>
registrationController
</managed-bean-name>
<managed-bean-class>
coreservlets.RegistrationController
</managed-bean-class>
<managed-bean-scope>request</managed-bean-scope>
</managed-bean>
…
</faces-config>
```

Takođe treba definisati i pravila navigacije, t.j. u oovom primeru, tri navedena stringa (bad-address, bad-password, success) mapirati u odgovarajuće klijentske stranice:

```
…
<faces-config>
…
<navigation-rule>
<from-view-id>/register2.jsp</from-view-id>
<navigation-case>
<from-outcome>bad-address</from-outcome>
<to-view-id>/WEB-INF/results/bad-address2.jsp</to-view-
id>
</navigation-case>
<navigation-case>
<from-outcome>bad-password</from-outcome>
<to-view-id>/WEB-INF/results/bad-password2.jsp</to-view-
id>
</navigation-case>
<navigation-case>
<from-outcome>success</from-outcome>
<to-view-id>/WEB-INF/results/result2.jsp</to-view-id>
</navigation-case>
</navigation-rule>
</faces-config>
```

I na kraju se definišu navedene tri klijentske stranice …/jsf-test/WEB-INF/results/bad-address2.jsp

```
<!DOCTYPE …>
<HTML>
<HEAD>…</HEAD>
<BODY>
<CENTER>
<TABLE BORDER=5>
<TR><TH CLASS="TITLE">Nelegalna Email Adresa</TH></TR>
</TABLE>
<P>
Adresa mora da bude u formatu username@host.
Molimo <A HREF="register2.faces">probajte ponovo </A>.
</CENTER>
</BODY>
</HTML>
```

Ako šifra nije u definisanom formatu - …/jsf-test/WEB-INF/results/badpassword2.jsp

```
<!DOCTYPE …>
<HTML>
<HEAD>…</HEAD>
<BODY>
<CENTER>
<TABLE BORDER=5>
<TR><TH CLASS="TITLE">Nelegalna Sig+fra</TH></TR>
</TABLE>
<P>
Sifra mora da ima najmanje 6 karaktera.
Molimo <A HREF="register2.faces">probajte ponovo</A>.
</CENTER>
</BODY>
</HTML>
```

Poslednja stranica se prikazuje u slučaju da je sve korektno …/jsf-test/WEB-INF/results/result2.jsp

```
<!DOCTYPE …>
<HTML>
<HEAD>…</HEAD>
<BODY>
<CENTER>
<TABLE BORDER=5>
<TR><TH CLASS="TITLE">Uspesno</TH></TR>
</TABLE>
<H2>Uspesno ste se registrovali<BR>
</H2>
</CENTER>
</BODY>
</HTML>
```

5.5 Bean-ovi

Sledeći korak u složenosti aplikacije bi bio upotreba bean-ova. Tipična JSF aplikacija uparuje bean u pozadini sa svakom stranicom u aplikaciji. Bean definiše sadržaj i metode koje su pridružene svakoj UI komponenti koja se koristi na stranici. Sadržaj samog bean-a može biti povezan ili sa instancom komponente ili sa njenom vrednošću.
Metode backing bean-a izvode određene funkcije vezane za komponentu kao što su: validacija podataka komponente, obrada događaja koje komponenta okida i izvođenje akcija vezanih za navigaciju kada se aktivira komponenta. Atribut value tag-a komponente se koristi da poveže sadržaj bean-a sa vrednošću komponente. Atribut binding tag-a komponente se koristi da poveže sadržaj bean-a sa instancom komponente.

```
public class PersonBean {
   String personName;
   /**
   * @return Vraca ime
   */
   public String getPersonName() {
      return personName;
   }
   /**
   * @param Ime
   */
   public void setPersonName(String name) {
      personName = name;
   }
}
```

Vrednost komponente je povezana sa atributom bean-a na JSP stranici:

```
<h:inputText value="#{personBean.personName}"
required="true">
    <f:validateLength minimum="2" maximum="15"/>
</h:inputText>
```

Posle implementacije bean-ova koji će se koristiti u aplikaciji potrebno je konfigurisati ih u faces-config.xml konfiguracionom fajlu. To je potrebno da bi JSF mogao da automatski kreira nove instance bean-ova kad god se ukaže potreba:

```
<managed-bean>
  <managed-bean-name>personBean</managed-bean-name>
  <managed-bean-class>jsfks.PersonBean</managed-bean-
class>
  <managed-bean-scope>request</managed-bean-scope>
  <managed-property>
    <property-name>personName</property-name>
    <property-class>java.lang.String</property-class>
    <value>Bosko Nikolic</value>
  </managed-property>
</managed-bean>
```

Za prezentovanej upotrebe bean-ova u ovoj vrsti Internet aplikacija koristiće se sličan primer kao i u delu teksta neposredno pre ovog. Opet korisnik definiše dva parametra, koja se prosleđuju serverskoj strani, izvršava se njihova provera i generiše oddređeni rezultat.
Prvi korak će biti realizacija bean-a koji će služiti za enkapsulaciju podataka koji dolaze od strane klijenta ka serveru. Treba napomenuti da ova vrsta bean-ova nije korišćena u okviru JSP-servlet tehnologije, a da u svakom slučaju umnogome doprinosi modularnosti celog sistema. Kod bean-a za dati preimer je

```
package coreservlets;
public class RegistrationBean {
private String email = "user@host";
private String password = "";
public String getEmail() {
return(email);
}
public void setEmail(String email) {
this.email = email;
}
public String getPassword() {
return(password);
}
public void setPassword(String password) {
this.password = password;
}
}
```

Što se tiče komunikacije u smeru server-klijent, već ranije su korišćeni bean-ovi za smeštanje rezultata na serverskoj strani, i kasniju prezentaciju na klijentskoj.

Realizovan je bean i za ovu namenu

```
package coreservlets;
public class SuggestionBean {
private String email;
private String password;
public SuggestionBean(String email, String password) {
this.email = email;
this.password = password;
}
public String getEmail() {
return(email);
}
public String getPassword() {
return(password);
}
}
```

I

```
package coreservlets;
public class SuggestionUtils {
private static String[] suggestedAddresses =
{ "president@whitehouse.gov",
"gates@microsoft.com",
"palmisano@ibm.com",
"ellison@oracle.com" };
private static String chars =
"abcdefghijklmnopqrstuvwxyz0123456789#@$%^&*?!";
public static SuggestionBean getSuggestionBean() {
String address = randomString(suggestedAddresses);
String password = randomString(chars, 8);
return(new SuggestionBean(address, password));
}
...
```

U ovom bean-u će se smestiti predlozi korektnih vrednosti za email i šifru u slučaju da nisu ranije uneti na pravilan način.
Bean koji predstavlja objekat za komunikaciju od klijenta ka serveru, može imati i dodatne efekte. Pri prikazu ove stranice moguće je odmah uspostaviti vezu sa odgovarajućim property-ijima bean i na taj način izvršiti određene inicijalizacije. Da bi se to uspleo potrebni je koristiti tagove tipa h:inputNesto koji pomoću vrednosti atributa definišu odgovarajući bean property, pa i početnu vrednost tekst polja. Treba obratiti pažnju da u ovom slučaju nije potrebno definisati id atribut.

```
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f"
%>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h"
%>
<f:view>…
<h:form>
Email address:
<h:inputText value="#{registrationBean.email}"/><BR>
Password:
<h:inputSecret
value="#{registrationBean.password}"/><BR>
<h:commandButton value="Sign Me Up!"
action="#{registrationBean.register}"/>
</h:form>…
</f:view>
```

Sada je na serverskoj strani u okviru definisane akcije moguće pristupiti property-ijima bean-a direktno. Sistem ih automatski popunjava pre poziva action metoda. Za razliku od prethodnog primera u ovom slučaju nije potrebno pristupati objektu ExternalContext ili pozivati metod request.getParameter. Slično JSP-servlet pristupu rezultati se smeštaju u property-ije rezultujućeg bean-a, a i dalje se rezultujući stringovi poklapaju sa mogućim izlazima

```
public class RegistrationBean {
public String register() {
if ((email == null) ||
(email.trim().length() < 3) ||
(email.indexOf("@") == -1)) {
suggestion = SuggestionUtils.getSuggestionBean();
return("bad-address");
} else if ((password == null) ||
(password.trim().length() < 6)) {
suggestion = SuggestionUtils.getSuggestionBean();
return("bad-password");
} else {
return("success");
}
}
}
```

U ovom primeru postupak definisanja faces-config fajla je potpuno isti kao u prethodnim primerima

```
…
<faces-config>
<managed-bean>
<managed-bean-name>
registrationBean
</managed-bean-name>
<managed-bean-class>
coreservlets.RegistrationBean
</managed-bean-class>
<managed-bean-scope>request</managed-bean-scope>
</managed-bean>
…
</faces-config>
```

Kao i pravila navigacije

```
…
<faces-config>
…
<navigation-rule>
<from-view-id>/register3.jsp</from-view-id>
<navigation-case>
<from-outcome>bad-address</from-outcome>
<to-view-id>/WEB-INF/results/bad-address3.jsp</to-view-
id>
</navigation-case>
<navigation-case>
<from-outcome>bad-password</from-outcome>
<to-view-id>/WEB-INF/results/bad-password3.jsp</to-view-
id>
</navigation-case>
<navigation-case>
<from-outcome>success</from-outcome>
<to-view-id>/WEB-INF/results/result3.jsp</to-view-id>
</navigation-case>
</navigation-rule>
</faces-config>
```

Što se tiče prezentacionih klijentskih strana, rezultat je sada smešten u okviru bean-a, pa se može koristiti h:outputText da bi se pristupilo property-ijima bean-a

```
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f"
%>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h"
%>
<f:view>
<!DOCTYPE …>
<HTML>
…
<h:outputText value="#{beanName.propertyName}"/>
…
</HTML>
</f:view>
```

Za nelegalnu email adresu …/jsf-test/WEB-INF/results/bad-address3.jsp

```
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f"
%>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h"
%>
<f:view>
<!DOCTYPE …>
<HTML>
…
<TABLE BORDER=5>
<TR><TH CLASS="TITLE">Nelegalna Email Adresa</TH></TR>
</TABLE>
<P>
Adresa
"<h:outputText value="#{registrationBean.email}"/>"
Nije u formi username@hostname (n.p.,
<h:outputText
value="#{registrationBean.suggestion.email}"/>).
<P>
Molimo <A HREF="register3.faces">probajte ponovo</A>.
…
</HTML>
</f:view>
```

Treba primetiti da se u ovoj stranici pristupa property-ijima i bean-a koji je formiran i povezan sa početnom klijentskom stranicom (registrationBean.email}), i sa bean-om u kome je smešten rezultat (registrationBean.suggestion.email), na potpuno isto način.
I za nekorektnu šifru …/jsf-test/WEB-INF/results/bad-password3.jsp

```
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f"
%>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h"
%>
<f:view>
<!DOCTYPE …>
<HTML>
…
<TABLE BORDER=5>
<TR><TH CLASS="TITLE">Nelegalna šifra</TH></TR>
</TABLE>
<P>
Šifra
"<h:outputText value="#{registrationBean.password}"/>"
Je prekratka; mora da sadrzi najmanje 6 karaktera.
Moguća sifra je:
<h:outputText
value="#{registrationBean.suggestion.password}"/>.
<P>
Molimo <A HREF="register3.faces">probajte ponovo</A>.
…
</HTML>
</f:view>
```

Ako je sve uspešno …/jsf-test/WEB-INF/results/result3.jsp

```
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f"
%>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h"
%>
<f:view>
<!DOCTYPE …>
<HTML>
…
<TABLE BORDER=5>
<TR><TH CLASS="TITLE">Uspesno</TH></TR>
</TABLE>
<H2>Registrovali ste se uspesno.</H2>
<UL>
```

```
<LI>Email Adresa:
<h:outputText value="#{registrationBean.email}"/>
<LI>Sifra:
<h:outputText value="#{registrationBean.password}"/>
</UL>
…
</HTML>
</f:view>
```

Date stranice je moguće realizovati i pomoću EL pristupa. U tom slučaju se izbegava taglib deklaracija i f:view tagovi i dobijaju se kraći izrazi koji prikazuju property-ije bean-a

```
<TR><TH CLASS="TITLE">Uspesno</TH></TR>
</TABLE>
<H2>Registrovali ste se uspesno.</H2>
<UL>
<LI>Email Adresa:${registrationBean.email}
<LI>Sifra: ${registrationBean.password}
</UL>
…
</HTML>
```

Upotreba EL u okviru JSF stranica se u određenim stvarima razlikuje od EL u okviru JSP stranica. Može se koristiti samo u okviru atributa JSF tagova, zahtevaju taglib deklaraciju i mogu se koristiti samo na serverima koji podržavaju JSP 1.2+. Da bi se pristupilo određenim objektima koristi se #{blah}. Mogu da prikažu poslate podatke i izlazne vrednosti i pristupaju bean-ovima definisanim u okviru zahteva, sesije, aplikacije ili posebno definisanim (mogućnost koju običan EL nema).

5.6 Rad sa pomoćnim fajlovima

Često je potrebno u okviru aplikacije menjati određene poruke ili natpise na ekranima. Do sada je data operacija morala da se obavi tako što bi se menjao kod određene klijentske stranice ili serverske komponente. Za više poruka, mora se proučavati i menjati kod više stranica. Posebane teškoće bi se desile ako je potrebno aplikaciju prebaciti na potpuno drugi jezik, t.j učiniti internacionalizaciju.
Zato se došlo do ideje da se tekstovi poruka i natpisi čuvaju u jednom posebnom, u suštini tekstualnom fajlu. Promenom ovog fajla, automatski bi se promenili i tekstualni sadržaji cele aplikacije. Pomoćni fajl koji se koristi naziva se properties fajl i pomoću nekoliko koraka može se koristiti u okviru JSF aplikacije.
Prvi korak je kreirati .properties fajl u okviru direktorijuma WEB-INF/classes. Ovaj fal će sadržiti jednostavne parove imeKljuca=vrednost. Tako napravljeni fajl je moguće koristiti u okviru aplikacije pomoću f:loadBundle taga, kod koga atribut basename definiše osnovno ime fajla, a atribut var definiše promenljivu (koja je tipa Map) gde se sadrži rezultat. Pri pozivu fajla treba obratiti pažnju o fizičkoj lokaciji fajla, jer se pozicioniranje obavlja relativno u odnosu na WEB-INF/classes, a i ekstenzija .properties se podrazumeva. Znači, na primer ako se želi pristupiti fajlu WEB-INF/classes/messages.properties treba na klijentskoj strani pozvati

```
<f:loadBundle basename="messages" var="msgs"/>
```

Takođe za WEB-INF/classes/package1/test.properties potrebno je pozvati

```
<f:loadBundle basename="package1.test" var="msgs"/>
```

Kada se na ovaj način ostvari konekcija sa properties fajlom, njegove vrednosti definisane pomoću imena ključa koriste izlazne poruke sa uobičajenjim EL konstrukcijama

```
#{msgs.keyName}
```

Jednostavan primer ovakve upotrebe pomoćnih faajlova bi bio sledeći. Neka su u okviru fajla WEB-INF/classes/messages1.properties definišu vrednosti par ključeva, kao što je naslov stranice, osnovni tekst, poruke za pitanje imena, przimena, adrese i natpis na dugmetu:

```
title=Registration
text=Please enter your first name, last name,
and email address.
firstNamePrompt=Enter first name
lastNamePrompt=Enter last name
emailAddressPrompt=Enter email address
buttonLabel=Register Me
```

Sada se na klijentskoj strani ovi parametri mogu prikazati:

```
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f"
%>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h"
%>
<f:view>
<f:loadBundle basename="messages1" var="msgs"/>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0
Transitional//EN">
<HTML>
<HEAD><TITLE>
<h:outputText value="#{msgs.title}"/>
</TITLE>
<LINK REL="STYLESHEET"
HREF="./css/styles.css" TYPE="text/css">
</HEAD>
<BODY>
<CENTER>
<TABLE BORDER=5>
<TR><TH CLASS="TITLE">
<h:outputText value="#{msgs.title}"/></TH></TR>
</TABLE>
<BR>
<h:outputText value="#{msgs.text}"/>
<P>
<h:form>
<h:outputText value="#{msgs.firstNamePrompt}"/>:
<h:inputText value="#{person.firstName}"/>
<BR>
<h:outputText value="#{msgs.lastNamePrompt}"/>:
<h:inputText value="#{person.lastName}"/>
```

```
<BR>
<h:outputText value="#{msgs.emailAddressPrompt}"/>:
<h:inputText value="#{person.emailAddress}"/>
<BR>
<h:commandButton
value="#{msgs.buttonLabel}"
action="#{person.doRegistration}"/>
</h:form>
</CENTER>
</BODY>
</HTML>
```

Radi opštosti definisanih poruka moguće je u okviru njih definisati i parametre koji će dobiti vrednosti tokom samog pristupa određenoj poruci iz properties fajla. I u ovomslučaju kreira se .properties fajl u okviru WEB-INF/classes. Vrednosti poruka sadržer parametre {0}, {1}, {2}, .., n.p., nekoIme=blah {0} blah {1}. Zatim jr potrebno učitati fajl pomoću taga f:loadBundle kao i ranije, i pomoću atributa basename definisati osnovno ime fajla, a sa var definisati promenljivu koja će sadržati rezultat. Izlazne poruke na klijetskoj strani koriste h:outputFormat i vrednost definiše osnovnu poruku, dok ugnežđeni elementi f:param daju vrednosti za zamenu. Na primer:

```
<h:outputFormat value="#{msgs.nekoIme}">
      <f:param value="vrednost za 0ti ulaz"/>
      <f:param value="vrednost za1vi ulaz/>
</h:outputFormat>
```

Prethodni primer sada ima par izmena. Properties fajl (WEB-INF/classes/messages2.properties) ima sledeći izgled

```
title=Registration
firstName=first name
lastName=last name
emailAddress=email address
text=Please enter your {0}, {1}, and {2}.
prompt=Enter {0}
buttonLabel=Register Me
```

I u okviru klijentske stranice pristupa se porukama, i definišu se vrednosti parametara

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h"
%>
<f:view
locale="#{facesContext.externalContext.request.locale}">
<f:loadBundle basename="messages2" var="msgs"/>
...
<TABLE BORDER=5>
<TR><TH CLASS="TITLE">
<h:outputText value="#{msgs.title}"/></TH></TR>
</TABLE>
<BR>
<h:outputFormat value="#{msgs.text}">
<f:param value="#{msgs.firstName}"/>
<f:param value="#{msgs.lastName}"/>
<f:param value="#{msgs.emailAddress}"/>
</h:outputFormat>
<P>
<h:form>
<h:outputFormat value="#{msgs.prompt}">
<f:param value="#{msgs.firstName}"/>
</h:outputFormat>:
<h:inputText value="#{person.firstName}"/>
<BR>
<h:outputFormat value="#{msgs.prompt}">
<f:param value="#{msgs.lastName}"/>
</h:outputFormat>:
<h:inputText value="#{person.lastName}"/>
<BR>
<h:outputFormat value="#{msgs.prompt}">
<f:param value="#{msgs.emailAddress}"/>
</h:outputFormat>:
<h:inputText value="#{person.emailAddress}"/>
<BR>
<h:commandButton
value="#{msgs.buttonLabel}"
action="#{person.doRegistration}"/>
</h:form>
</CENTER></BODY></HTML>
</f:view>
```

5.7 Obrada događaja

U okviru Web aplikacija je često potrebno da se odgovori na korisničke događaje, koji su selektovani iz menija ili kliknuti na dugme. Najčešće je potrebno da se u okviru komponente registruju metodi koji će oobrađivati određeni događaj (event handlers kod), na primer:

```
<h:selectOneMenu
valueChangeListener="#{form.countryChanged}"...>
...
</h:selectOneMenu>
```

U prethodnom kodu, metod povezuje metod countryChanged koji je definisan u okviru Bean-a form. Sam poziv metoda će se desiti nakon korisnikove akcije selektovanja opcije iz menija.
JSF tehnologija podržava tri vrste događaja :

- Događaj izazvan promenom vrednosti (Value change events)
- Događaj izazvan akcijom (Action events)
- Fazni događaji (Phase events)

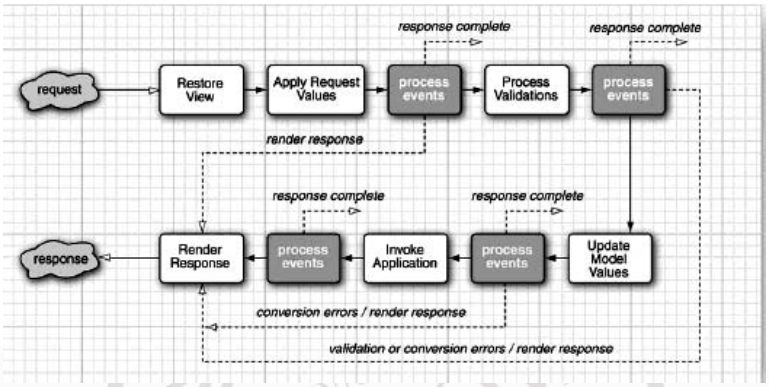
Događaji koji su izazvani usled promene vrednosti su vezani za određene HTML elemente, kao što su h:inputText, h:selectOneRadio, i h:selectManyMenu, i to u slučajevima kada se vrednost komponente menja.
Događaji koji su izazvani usled akcije javljaju se nakon akitivranja komandnih komponenti, kao što su elementi h:commandButton i h:commandLink, slučajevi kada je dugme ili link aktiviran.

Fazni događaji se aktiviraju u toku JSF životnig ciklusa.

5.8 Životni ciklus događaja

Zahtevi u JSF aplikacijama su procesirani od strane JSF okruženja pomoću kontrolnog servleta, koji izvršava JSF životni ciklus .

JSF životni ciklus je dat na sledećoj slici:



Slika 23 JSF životni ciklus

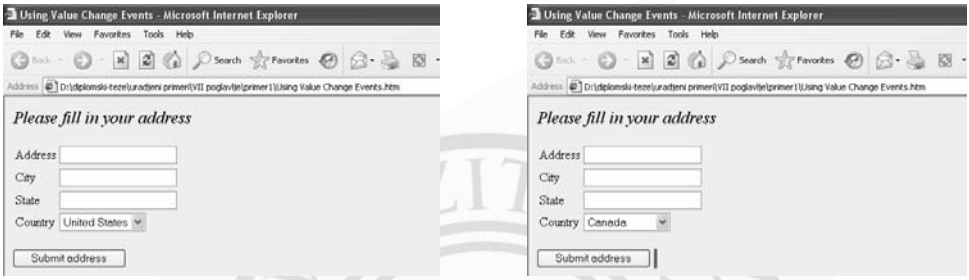
Počevši od Apply Request Values faze, JSF implementacija može da kreira događaje i da ih doda u red događaja u toku svake faze životnog ciklusa. Posle ovih faza, JSF implementacija šalje događaje iz redova regitrovanim osluškivačima.

Treba napomenuti da osluškivači događaja mogu da utiču na JSF životni ciklus na jedan od sledećih načina:

- Pustiti da se životni ciklus izvršava normalno.
- Pozvati FacesContext.renderResponse() da bi se preskočio ostatak životnog ciklusa sve do Render Response.
- Pozvati FacesContext.responseComplete() da bi se preskočio ostatak životnog ciklusa, ceo.

5.9 Događaji koji su izazvani usled promene vrednosti

Komponente u Web aplikaciju često zavise jedna od druge. Može se zadržati zavisnost komponenti u sinhronizaciji sa događajima koji su izazvani usled promene vrednosti, koji su se desili iz ulaznih komponenti nakon novih vrednosti koja je unešena.



Slika 24. Izgled aplikacije

Aplikacija na slici vezuje vrednost osluškivača promena za vrednost opcije "Country" i koristi onchange atribut kako bi forma prihatila vrednost nakon što je se izvršila promena iz menija.

```
<h:selectOneMenu value="#{form.country}"
onchange="submit()"
valueChangeListener="#{form.countryChanged}">
<f:selectItems value="#{form.countryNames}"/>
</h:selectOneMenu>
```

Kada korisnik selektuje zemlju iz menija, JavaScript prihvatna funkcija se poziva kako bi prihvatila formu iz menija koja zatim pokreće JSF životni ciklus. Nakon faze procesa validacije, JSF okruženje poziva metodu countryChanged koja je definisana u okviru Bean-a form.

```
private static String US = "United States";
...
public void countryChanged(ValueChangeEvent event) {
FacesContext context =
FacesContext.getCurrentInstance();
if (US.equals((String) event.getNewValue()))
context.getViewRoot().setLocale(Locale.US);
else
context.getViewRoot().setLocale(Locale.CANADA);
}
}
```

Kao svi događaji koji su izazvani usled promene vrednosti, prethodni osluškivač prosleđuje vrednost događaju. Osluškivači koriste taj događaj kako bi pristupili vrednosti nove komponente. Klasa ValueChangeEvent nasleđuje klasu FacesEvent, a obe ove klase se nalaze u paketu javax.faces.event package.

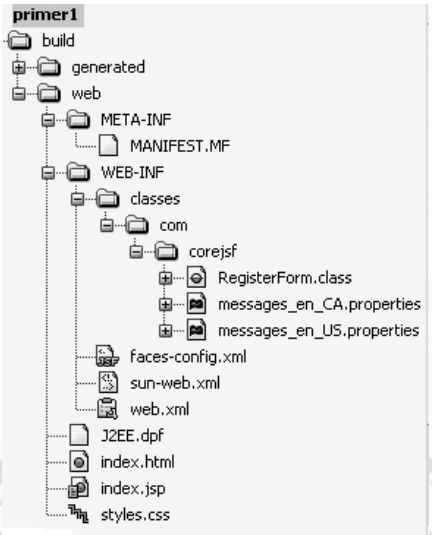
Metodi klase javax.faces.event.ValueChangeEvent koji mogu korisni su

- UIComponent getComponent() - Vraća ulaznu komponentu koju je okinuo događaj
- Object getNewValue() - Vraća novu vrednost komponente, nakon što je vrednost konvertovana i potvrđena.
- Object getOldValue() - Vraća prethodnu vrednost komponente.

I klasa javax.faces.event.FacesEvent ima sledeće metode

- void queue() - Stavlja događaje u red, na kraju tekuće faze životnog siklusa.
- PhaseId getPhaseId() - Vraća fazni indentifikator u skladu sa fazom tokom koje je događaj isporučen.
- void setPhaseId(PhaseId)- Postavlja fazni indentifikator u skladu sa fazom tokom koje je događaj isporučen.

Sledeći primer koji ilustruje obradu događaja ima strukturu



Slika 25. Struktura fajlova u sledećem primeru

U index.jsp (valuechange/web/index.jsp) stranici čitaju se vrednosti se iz properties fajla za parametar windowTitle i ispisuje se kao naslov u okviru title taga, zatim se ispisuje vrednost pageTitle u okviru tela same stranice.
Zatim se se pravi panelGrid sa 2 kolone u koje se prvo u upisuju vrednosti iz properties fajla, a potom u drugu kolonu i sama vrednost, koja se unosi u tekst polje i čita vrednost iz Bean-a, za odgovarajući property.
Ko što je ranije naglašeno kada korisnik selektuje zemlju iz menija, JavaScript prihvatna funkcija se poziva kako bi prihvatila formu iz menija koja zatim pokreće JSF životni ciklus. Nakon faze procesa validacije, JSF okruženje poziva metodu countryChanged koja je definisana u okviru Bean-a form.
Na kraju strane definiše se dugme, koje izaziva akcione događaje.

```
1. <html>
2. <%@ taglib uri="http://java.sun.com/jsf/core"
prefix="f" %>
3. <%@ taglib uri="http://java.sun.com/jsf/html"
prefix="h" %>
4. <f:view>
5. <head>
6. <link href="styles.css" rel="stylesheet"
type="text/css"/>
```

```
7. <title>
8. <h:outputText value="#{msgs.windowTitle}"/>
9. </title>
10. </head>
11.
12. <body>
13. <h:outputText value="#{msgs.pageTitle}"
styleClass="emphasis"/>
14. <p/>
15. <h:form>
16. <h:panelGrid columns="2">
17. <h:outputText value="#{msgs.streetAddressPrompt}"/>
18. <h:inputText id="streetAddress"
value="#{form.streetAddress}"
19. required="true"/>
20.
21. <h:outputText id="cityPrompt"
value="#{msgs.cityPrompt}"/>
22. <h:inputText value="#{form.city}"/>
23.
24. <h:outputText id="statePrompt"
value="#{msgs.statePrompt}"/>
25. <h:inputText value="#{form.state}"/>
26.
27. <h:outputText value="#{msgs.countryPrompt}"/>
28.
29. <h:selectOneMenu value="#{form.country}"
30. onchange="submit()"
31. valueChangeListener="#{form.countryChanged}">
32. <f:selectItems value="#{form.countryNames}"/>
33. </h:selectOneMenu>
34. </h:panelGrid>
35. <p/>
36. <h:commandButton value="#{msgs.submit}"/>
37. </h:form>
38. </body>
39. </f:view>
40. </html>
```

U serverskoj klasi RegisterForm.java

(valuechange/src/java/com/corejsf/RegisterForm.java)

pravi se instance Bean-a RegisterForm koji sadrži privatna polja i javne metode za dohvatanje vrednosti tih polja, kao i metodu koja zahteva vrednost od događaja i koja postavlja parametre konteksta na odgovarajuću vrednost,u zavisnosti od vrednosti koja je dobijena od događaja.

```
1. package com.corejsf;
2.
3. import java.util.ArrayList;
4. import java.util.Collection;
5. import java.util.Locale;
6. import javax.faces.context.FacesContext;
7. import javax.faces.event.ValueChangeEvent;
8. import javax.faces.model.SelectItem;
9.
10. public class RegisterForm {
11. private String streetAddress;
12. private String city;
13. private String state;
14. private String country;
15.
16. private static final String US = "United States";
17. private static final String CANADA = "Canada";
18. private static final String[] COUNTRY_NAMES = { US,
CANADA };
19. private static ArrayList<SelectItem> countryItems =
null;
20.
21. // PROPERTY: countryNames
22. public Collection getCountryNames() {
23. if (countryItems == null) {
24. countryItems = new ArrayList<SelectItem>();
25. for (int i = 0; i < COUNTRY_NAMES.length; i++) {
26. countryItems.add(new SelectItem(COUNTRY_NAMES[i]));
27. }
28. }
29. return countryItems;
30. }
31.
32. // PROPERTY: streetAddress
33. public void setStreetAddress(String newValue) {
streetAddress = newValue; }
34. public String getStreetAddress() { return
streetAddress; }
35.
36. // PROPERTY: city
37. public void setCity(String newValue) { city =
newValue; }
38. public String getCity() { return city; }
39.
40. // PROPERTY: state
```

```
41. public void setState(String newValue) { state =
newValue; }
42. public String getState() { return state; }
43.
44. // PROPERTY: country
45. public void setCountry(String newValue) { country =
newValue; }
46. public String getCountry() { return country; }
47.
48. public void countryChanged(ValueChangeEvent event) {
49. FacesContext context =
FacesContext.getCurrentInstance();
50.
51. if (US.equals((String) event.getNewValue()))
52. context.getViewRoot().setLocale(Locale.US);
53. else
54. context.getViewRoot().setLocale(Locale.CANADA);
55. }
56.
```

Preostao je konfiguracioni fajl cele aplikaci (valuechange/web/WEB-INF/faces-config.xml). U okviru njega definiše se properties fajl u okviru taga application , gde se navodi paket u kome se nalazi taj fajl kao i oznaka promenljive koja će se koristiti u index.jsp stranici. Zatim se definiše ime Bean-a, putanja Bean-a i opseg Bean-a u okviru taga managed-bean.

```
1. <?xml version="1.0"?>
2. <faces-config
xmlns="http://java.sun.com/xml/ns/javaee"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
5. http://java.sun.com/xml/ns/javaee/web-
facesconfig_1_2.xsd"
6. version="1.2">
7. <application>
8. <resource-bundle>
9. <base-name>com.corejsf.messages</base-name>
10. <var>msgs</var>
11. </resource-bundle>
12. </application>
13.
14. <managed-bean>
15. <managed-bean-name>form</managed-bean-name>
16. <managed-bean-
class>com.corejsf.RegisterForm</managed-bean-class>
17. <managed-bean-scope>session</managed-bean-scope>
18. </managed-bean>
19. </faces-config>
```

Treba definisati i odgovarajuću properties fajl koji sadrži vrednosti koje se koriste u okviru samog prikaza. Jedan mogući ovakav properties fajl je valuechange/src/java/com/corejsf/messages_en_US.properties, i sadrži podatke za državu SAD.

```
1. windowTitle=Using Value Change Events
2. pageTitle=Please fill in your address
3.
4. streetAddressPrompt=Address
5. cityPrompt=City
6. statePrompt=State
7. countryPrompt=Country
8. submit=Submit address
```

Sledeći properties fajl je identične strukture (valuechange/src/java/com/corejsf/messages_en_CA.properties) i sadrži podatke za državu Kanadu.

```
1. windowTitle=Using Value Change Events
2. pageTitle=Please fill in your address
3.
4. streetAddressPrompt=Address
5. cityPrompt=City
6. statePrompt=Province
7. countryPrompt=Country
8. submit=Submit address
```

5.10 Akcioni događaji

Događaji koji su izazvani usled akcije se javljaju nakon akitiviranja komandnih komponenti, kao što su h:commandButton i h:commandLink, kada je dugme ili link aktiviran. Oni se aktiviraju u toku faze buđenja aplikacije (Invoke Application faza), negde pri kraju životnog ciklusa. Obično se vezuju osluškivači akcija za komandne komponente u JSP stranici. Na primer, osluškivač koji se dodaje na link:

```
<h:commandLink actionListener="#{bean.linkActivated}">
...
</h:commandLink>
```

Kada se aktivira komanda ili link, okružujuća forma je prihvaćena i zatim JSF okruženje generiše akcioni događaj. Neophodno je razlikovati pojmove osluškivači akcija (action listeners) i same akcije(actions). U suštini, akcije su dizajnirane za logiku poslova i učestvuju u navigaciji upravljanja, dok osluškivači događaja izvode interfejsnu logiku i ne učestvuju u navigaciji upravljanja. Osluškivači događaja ponekad rade zajedno sa akcijama kada je akcijama potrebna informacija o korisničkom interfejsu.
Treba napomenuti da akcija ne može da implementira ponašanje, ali ona može da upravlja sa odgovarajućom stranicom, i ne može da određuje odgovarajuću stranicu, zato što ona ne zna ništa o slikovnim dugmadima (Image button) iz korisničkog intefejsa ili o kliku miša.
Novi primer se pokreće od sledeće slike



Slika 26. Početni ekran primera

koristi dugme sa slikom, u ovom slučaju:

```
<h:commandButton image="mountrushmore.jpg"
actionListener="#{rushmore.listen}"
action="#{rushmore.act}"/>
```

Kada se klikne mišem na odgovarajući lik, osluškivač,koji je pristupio koordinatama na koje ste pozicionirali miš, određuje koji je lik selektovan.

```
public class Rushmore {
private String outcome;
private Rectangle washingtonRect = new
Rectangle(70,30,40,40);
private Rectangle jeffersonRect = new
Rectangle(115,45,40,40);
private Rectangle rooseveltRect = new
Rectangle(135,65,40,40);
private Rectangle lincolnRect = new
Rectangle(175,62,40,40);
public void listen(ActionEvent e) {
FacesContext context =
FacesContext.getCurrentInstance();
String clientId = e.getComponent().getClientId(context);
Map requestParams = context.getExternalContext().
getRequestParameterMap();
int x = new Integer((String) requestParams.get(clientId
+ ".x")).intValue();
int y = new Integer((String) requestParams.get(clientId
+ ".y")).intValue();
outcome = null;
if (washingtonRect.contains(new Point(x,y)))
outcome = "washington";
if (jeffersonRect.contains(new Point(x,y)))
outcome = "jefferson";
if (rooseveltRect.contains(new Point(x,y)))
outcome = "roosevelt";
if (lincolnRect.contains(new Point(x,y)))
outcome = "lincoln";
}
}
```

Akcija koja je pridružena dugmetu koristi rezultat da bi uticala na upravljanje

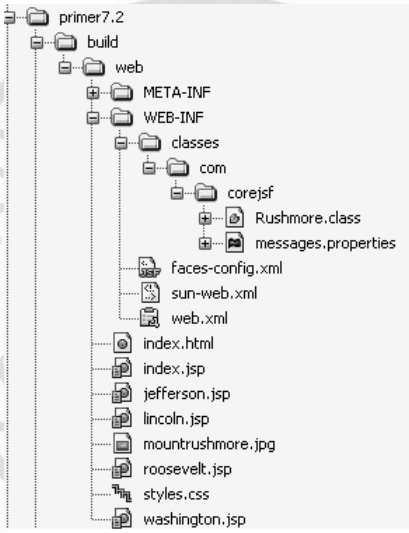
```
public String act() {
return outcome;
}
```

Mogući rezultati su:



Slika 27. Rezultati nakon određenih akcija

Treba posebno obratiti pažnju na JSF okruženje koje uvek pokreće osluškivače akcija pre same akcije. Takođe, JSF insistira da se razdvoji korisnikčka i poslovna logika. U prethodnom primeru akcija ne može da pristupi ID-u klijenta, koji je pročitan iz komponente koja je ispalila događaj, zatim ne može da pristupi informacijama koje su neophodne za izvlačenje koordinata miša iz parametara zahteva. Sve se to dešava zbog toga što, akcija ne zna ništa o korisničkom interfejsu, tako da mi moramo da dodamo akciju osluškivaču kako bi on implementirao zahtevano ponašanje. Sledeća slika pokazuje strukturu direktorijuma koji se koriste u navedenom primeru:



Slika 28. Struktura direktorijuma u datom primeru

U stranici rushmore/web/index.jsp se u okviru body taga ispisuje instructions properties fajla, pravi se referenca na odgovarajuće komandno dugme i definišu se osluškivač događaja i akcija.

```
1. <html>
2. <%@ taglib uri="http://java.sun.com/jsf/core"
prefix="f" %>
3. <%@ taglib uri="http://java.sun.com/jsf/html"
prefix="h" %>
4. <f:view>
5. <head>
6. <link href="styles.css" rel="stylesheet"
type="text/css"/>
7. <title>
8. <h:outputText value="#{msgs.indexWindowTitle}"/>
9. </title>
10. </head>
11.
12. <body>
13. <h:outputText value="#{msgs.instructions}"
14. styleClass="instructions"/>
15. <h:form>
16. <h:commandButton image="mountrushmore.jpg"
17. styleClass="imageButton"
18. actionListener="#{rushmore.handleMouseClick}"
19. action="#{rushmore.navigate}"/>
20. </h:form>
21. </body>
22. </f:view>
23. </html>
```

Ostale stranice u ovom primeru su stranice koje prikazuju detaljnije informacije o određenom liku. Na primer stranica rushmore/web/lincoln.jsp ispisuje parametre lincolnPageTitle, lincolnDiscussion iz properties fajla i pravi se referenca na odgovrajući komandni link.

```
1. <<html>
2. <%@ taglib uri="http://java.sun.com/jsf/core"
prefix="f" %>
3. <%@ taglib uri="http://java.sun.com/jsf/html"
prefix="h" %>
4. <f:view>
5. <head>
6. <link href="styles.css" rel="stylesheet"
type="text/css"/>
7. <title>
8. <h:outputText value="#{msgs.lincolnWindowTitle}"/>
9. </title>
10. </head>
11.
```

```
12. <body>
13. <h:form>
14. <h:outputText value="#{msgs.lincolnPageTitle}"
15. styleClass="presidentPageTitle"/>
16. <p/>
17. <h:outputText value="#{msgs.lincolnDiscussion}"
18. styleClass="presidentDiscussion"/>
19. <p>
20. <h:commandLink action="index"
21. styleClass="backLink">
22. ${msgs.indexLinkText}
23. </h:commandLink>
24. </p>
25. </h:form>
26. </body>
27. </f:view>
28. </html>
```

Slično stranica

rushmore/web/jefferson.jsp

ispisuje parametre jeffersonPageTitle, jeffersonDiscussion iz properties fajla i pravi se referenca na odgovrajući komandni link.

```
1. <html>
2. <%@ taglib uri="http://java.sun.com/jsf/core"
prefix="f" %>
3. <%@ taglib uri="http://java.sun.com/jsf/html"
prefix="h" %>
4. <f:view>
5. <head>
6. <link href="styles.css" rel="stylesheet"
type="text/css"/>
7. <title>${msgs.jeffersonWindowTitle}</title>
8. </head>
9.
10. <body>
11. <h:form>
12. <h:outputText value="#{msgs.jeffersonPageTitle}"
13. styleClass="presidentPageTitle"/>
14. <p/>
15. <h:outputText value="#{msgs.jeffersonDiscussion}"
16. styleClass="presidentDiscussion"/>
17. <p>
18. <h:commandLink action="index"
19. styleClass="backLink">
20. ${msgs.indexLinkText}
21. </h:commandLink>
```

```
22. </p>
23. </h:form>
24. </body>
25. </f:view>
26. </html>
```

U okviru stranice

rushmore/web/roosevelt.jsp

ispisuje parametre rooseveltPageTitle, rooseveltDiscussion iz properties fajla i pravi se referenca na odgovrajući komandni link

```
1. <html>
2. <%@ taglib uri="http://java.sun.com/jsf/core"
prefix="f" %>
3. <%@ taglib uri="http://java.sun.com/jsf/html"
prefix="h" %>
4. <f:view>
5. <head>
6. <link href="styles.css" rel="stylesheet"
type="text/css"/>
7. <title>
8. <h:outputText value="#{msgs.rooseveltWindowTitle}"/>
9. </title>
10. </head>
11.
12. <body>
13. <h:form>
14. <h:outputText value="#{msgs.rooseveltPageTitle}"
15. styleClass="presidentPageTitle"/>
16. <p/>
17. <h:outputText value="#{msgs.rooseveltDiscussion}"
18. styleClass="presidentDiscussion"/>
19. <p>
20. <h:commandLink action="index"
21. styleClass="backLink">
22. ${msgs.indexLinkText}
23. </h:commandLink>
24. </p>
25. </h:form>
26. </body>
27. </f:view>
28. </html>
```

I na kraju u stranici

rushmore/web/washington.jsp

ispisuje parametre washingtonPageTitle, washingtonDiscussion iz properties fajla i pravi se referenca na odgovrajući komandni link

```
1. <html>
2. <%@ taglib uri="http://java.sun.com/jsf/core"
prefix="f" %>
3. <%@ taglib uri="http://java.sun.com/jsf/html"
prefix="h" %>
4. <f:view>
5. <head>
6. <link href="styles.css" rel="stylesheet"
type="text/css"/>
7. <title>
8. <h:outputText value="#{msgs.washingtonWindowTitle}"/>
9. </title>
10. </head>
11.
12. <body>
13. <h:form>
14. <h:outputText value="#{msgs.washingtonPageTitle}"
15. styleClass="presidentPageTitle"/>
16. <p/>
17. <h:outputText value="#{msgs.washingtonDiscussion}"
18. styleClass="presidentDiscussion"/>
19. <p>
20. <h:commandLink action="index"
21. styleClass="backLink">
22. ${msgs.indexLinkText}
23. </h:commandLink>
24. </p>
25. </h:form>
26. </body>
27. </f:view>
28. </html>
```

Što se tiče serverske komponente koja je definisana u okviru klase rushmore/src/java/com/corejsf/Rushmore.java definiše se Bean, njegova polja i metoda koja radi sa akcionim događajem. Ova metoda ispituje coordinate izabranog lika, pa na osnovu njih generiše rezultat, pomoću koga se prikazuje odgovarajuća stranica.

```
1. package com.corejsf;
2.
3. import java.awt.Point;
4. import java.awt.Rectangle;
5. import java.util.Map;
6. import javax.faces.context.FacesContext;
7. import javax.faces.event.ActionEvent;
8.
9. public class Rushmore {
10. private String outcome = null;
11. private Rectangle washingtonRect = new Rectangle(70,
30, 40, 40);
12. private Rectangle jeffersonRect = new Rectangle(115,
45, 40, 40);
13. private Rectangle rooseveltRect = new Rectangle(135,
65, 40, 40);
14. private Rectangle lincolnRect = new Rectangle(175,
62, 40, 40);
15.
16. public void handleMouseClick(ActionEvent e) {
17. FacesContext context =
FacesContext.getCurrentInstance();
18. String clientId =
e.getComponent().getClientId(context);
19. Map requestParams =
context.getExternalContext().getRequestParameterMap();
20.
21. int x = new Integer((String)
requestParams.get(clientId + ".x"))
22. .intValue();
23. int y = new Integer((String)
requestParams.get(clientId + ".y"))
24. .intValue();
25.
26. outcome = null;
27.
28. if (washingtonRect.contains(new Point(x, y)))
29. outcome = "washington";
30.
31. if (jeffersonRect.contains(new Point(x, y)))
32. outcome = "jefferson";
33.
34. if (rooseveltRect.contains(new Point(x, y)))
35. outcome = "roosevelt";
36.
37. if (lincolnRect.contains(new Point(x, y)))
```

```
38. outcome = "lincoln";
39. }
40.
41. public String navigate() {
42. return outcome;
43. }
44. }
```

U konfiguracionom fajlu (rushmore/web/WEB-INF/faces-config.xml) kao i prethodnom primeru definiše se properties fajl i promenljiva koja se koristi u okviru index.jsp stranice. Zatim se definšu stranice koje će biti izvršene u zavisnosti od pozicije miša prilikom aktiviranja. Na kraju je data definicija Bean-a koji se koristi.

```
1. <?xml version="1.0"?>
2. <faces-config
xmlns="http://java.sun.com/xml/ns/javaee"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
5. http://java.sun.com/xml/ns/javaee/web-
facesconfig_1_2.xsd"
6. version="1.2">
7. <application>
8. <resource-bundle>
9. <base-name>com.corejsf.messages</base-name>
10. <var>msgs</var>
11. </resource-bundle>
12. </application>
13.
14. <navigation-rule>
15. <navigation-case>
16. <from-outcome>index</from-outcome>
17. <to-view-id>/index.jsp</to-view-id>
18. </navigation-case>
19. </navigation-rule>
20.
21. <navigation-rule>
22. <from-view-id>/index.jsp</from-view-id>
23. <navigation-case>
24. <from-outcome>washington</from-outcome>
25. <to-view-id>/washington.jsp</to-view-id>
26. </navigation-case>
27. <navigation-case>
28. <from-outcome>jefferson</from-outcome>
29. <to-view-id>/jefferson.jsp</to-view-id>
30. </navigation-case>
```

```
31. <navigation-case>
32. <from-outcome>roosevelt</from-outcome>
33. <to-view-id>/roosevelt.jsp</to-view-id>
34. </navigation-case>
35. <navigation-case>
36. <from-outcome>lincoln</from-outcome>
37. <to-view-id>/lincoln.jsp</to-view-id>
38. </navigation-case>
39. </navigation-rule>
40.
41. <managed-bean>
42. <managed-bean-name>rushmore</managed-bean-name>
43.    <managed-bean-class>com.corejsf.Rushmore</managed-
bean-class>
44. <managed-bean-scope>session</managed-bean-scope>
45. </managed-bean>
46.
47. </faces-config>
```

U properties fajlu (rushmore/src/java/com/corejsf/messages.properties) se definišu parameteri koji se prikazuju u okviru pomoćnih stranica.

```
1. instructions=For information about a president, click
on the president's face.
2.
3. indexWindowTitle=Actions and Action Listeners
4. indexLinkText=Back...
5. jeffersonWindowTitle=President Jefferson
6. rooseveltWindowTitle=President Roosevelt
7. lincolnWindowTitle=President Lincoln
8. washingtonWindowTitle=President Washington
9.
10. jeffersonPageTitle=Thomas Jefferson
11. rooseveltPageTitle=Theodore Roosevelt
12. lincolnPageTitle=Abraham Lincoln
13. washingtonPageTitle=George Washington
14.
15. jeffersonTabText=Jefferson
16. rooseveltTabText=Roosevelt
17. lincolnTabText=Lincoln
18. washingtonTabText=Washington
19.
20. lincolnDiscussion=President Lincoln was known as the
Great Emancipator because \
21. he was instrumental in abolishing slavery in the
United States. He was born \
```

```
22. into a poor family in Kentucky in 1809, elected
president in 1860, and \
23. assassinated by John Wilkes Booth in 1865.
24.
25. washingtonDiscussion=George Washington was the first
president of the United \
26. States. He was born in 1732 in Virginia, was elected
Commander in Chief of \
27. the Continental Army in 1775, and forced the
surrender of Cornwallis at Yorktown \
28. in 1781. He was inaugurated on April 30, 1789.
29.
30. rooseveltDiscussion=Theodore Roosevelt was the 26th
president of the United \
31. States. In 1901 he became president after the
assassination of President \
32. McKinley. At only 42, he was the youngest president
in U.S. history.
33.
34. jeffersonDiscussion=Thomas Jefferson, the 3rd U.S.
president, was born in \
35. 1743 in Virginia. Jefferson was tall and awkward,
and was not known as a \
36. great public speaker. Jefferson became minister to
France in 1785, after \
37. Benjamin Franklin held that post. In 1796, Jefferson
was a reluctant \
38. presidential candiate, and missed winning the
election by a mere three votes. \
39. He served as president from 1801 to 1809.
```

5.11 Tagovi osluškivača događaja

U prethodnom delu ove glave demonstrirana je primena akcija i osluškivača koji su reagovali na promene vrednosti komponenti i označavali ih sa actionListener and valueChangeListener atributima .
U osnovnom JSF okruženju, pored njih, postoje i sledeći tagovi:

- f:actionListener
- f:valueChangeListener

Ovi tagovi su analogni actionListener i valueChangeListener atributima. Na primer:

```
<h:selectOneMenu value="#{form.country}"
onchange="submit()"
valueChangeListener="#{form.countryChanged}">
<f:selectItems value="#{form.countryNames}"/>
</h:selectOneMenu>
```

Moguće je koristiti i tag f:valueChangeListener:

```
<h:selectOneMenu value="#{form.country}"
onchange="submit()">
<f:valueChangeListener
type="com.corejsf.CountryListener"/>
<f:selectItems value="#{form.countryNames}"/>
</h:selectOneMenu>
```

Prednost tagova u odnosu na atribute je to što oni dozvoljavaju da se vezuje više osluškivača na jednu kompponentu. U prethodnom kodu CountryListener je java klasa, koja implementira ValueChangeListener interfejs i definiše metodu void processValueChange(ValueChangeEvent event).

```
public class CountryListener implements
ValueChangeListener {
private static final String US = "United States";
public void processValueChange(ValueChangeEvent event) {
FacesContext context =
FacesContext.getCurrentInstance();
if (US.equals((String) event.getNewValue()))
context.getViewRoot().setLocale(Locale.US);
else
context.getViewRoot().setLocale(Locale.CANADA);
}
}
```

f:actionListener tag je analogan f:valueChangeListener, s tom razlikom, što implementira interfejs ActionListener. Na primer:

```
<h:commandButton image="mountrushmore.jpg"
styleClass="imageButton"
actionListener="#{rushmore.handleMouseClick}"
action="#{rushmore.navigate}"/>
```

Može i da se koristi f:actionListener:

```
<h:commandButton image="mountrushmore.jpg"
action="#{rushmore.navigate}">
<f:actionListener type="com.corejsf.RushmoreListener"/>
</h:commandButton>
```

Akcioni osluškivač mora da implementira ActionListener interfejs, koji definiše processAction metod, tako da u prethodnom kodu, JSF će pozvati RushmoreListener.processAction nakon što je dugme aktivirano. Takođe, mogu se dodati i višestruki osluškivači po komponenti. Na primer:

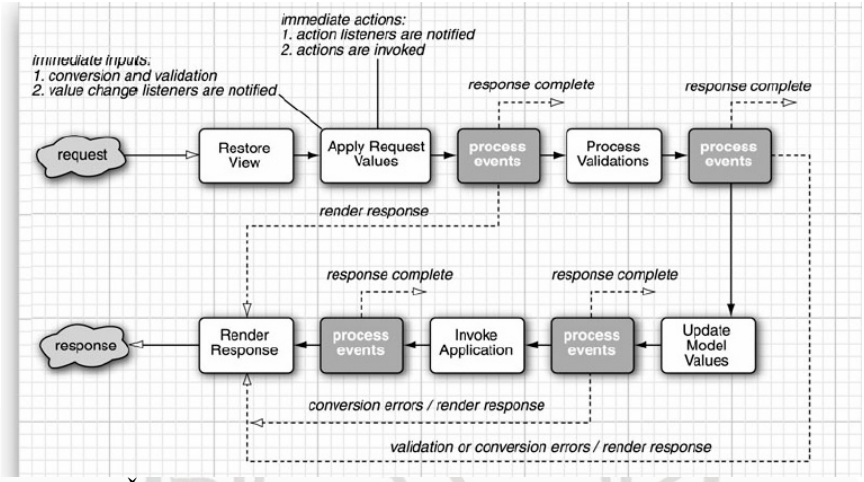
```
<h:commandButton image="mountrushmore.jpg"
action="#{rushmore.navigate}">
<f:actionListener type="com.corejsf.RushmoreListener"/>
<f:actionListener type="com.corejsf.ActionLogger"/>
</h:commandButton>
```

U prethodnom kodu ActionLogger klasa je jednostavan osluškivač događaja, koji loguje akcione događaje. Ako se specificiraju višestruki osluškivači po jednoj komponenti, kao što je urađeno u prethodnom kodu, redosled buđenja osluškivača je sledeći:

1. Osluškivači specificirani atributima.
2. Osluškivači specificirani tagovima.

5.12 Korišćenje neposrednih ulaznih komponente

Ako se želi da se akcioni događaji i događaji koji reaguju na promenu vrednosti komponenti ispale na početku životnog ciklusa tada se primenjuju neposredene komponente.



Slika 29. Životni ciklus aplikacije i neposredne ulazne komponente

Neposredni događaji se generišu nakon faze Apply Request Values.
Problem koji može da se javi je sledeći: U odnosu na prethodne primere ako se želi da se izvrši validacija kada korisnik klikne na dugme, ali kada nije Country izabrana. Kako da se specificira validaciju za jednu zemlju ali ne i za ostale?
Rešenje je da se realizuje meni za zemlje tako da bude neposredna komponenta Neposredna komponenta izvršava konvertovanje i validaciju, zatim dostavlja događaje koji reaguju na promenu vrednosti na početak životnog JSF ciklusa, nakon Apply Request Values faze, umesto posle Process Validations faze.
Potrebno je specificirati neposredne komponente sa neposrednim atributom, koji je slobodan svim ulazima i komandnim komponentama.

```
<h:selectOneMenu value="#{form.country}"
onchange="submit()"immediate="true"
valueChangeListener="#{form.countryChanged}">
<f:selectItems value="#{form.countryNames}"/>
</h:selectOneMenu>
```

Sa neposrednim atributom koji je postavljen na vredost true, posmatrani meni generiše događaje nakon Apply Request Values, pre nego što su druge ulazne komponente potvrđene. Zbog same validacije drugih komponenti u formi, poziva se metoda renderResponse konteksta JSF, na kraju osluškivača promene vrednosti.
Konkretno to izgleda ovako:

```
private static final String US = "United States";
...
public void countryChanged(ValueChangeEvent event) {
FacesContext context =
FacesContext.getCurrentInstance();
if (US.equals((String) event.getNewValue()))
context.getViewRoot().setLocale(Locale.US);
else
context.getViewRoot().setLocale(Locale.CANADA);
context.renderResponse();
}
```

Pri pozivu metode renderResponse() preskače se ostatak životnog ciklusa - uključujući validaciju ostalih ulaznih komponenti iz forme, pa sve do faze Render Response. Ako se želi da se se izbegne validacija kod događaja koji reaguju na promenu vrednosti, moževda se primeni postupak:

- dodavanja atribut immediate na ulazni tag ili
- poziva metoda FacesContext.renderResponse() na kraju osluškivača

U slučaju kada se želi da se aktivira link, pojavljuje se validaciona greška.Tada se, umesto neposredne ulazne komponente koristi neposredna komanda, odnosno dodaje se atribut immediate tagu h:commandLink. Na primer:

```
<h:commandLink action="#{localeChanger.germanAction}"
immediate="true">
<h:graphicImage value="/german_flag.gif" style="border:
0px"/>
</h:commandLink>
```

5.13 Prosleđivanje podataka iz korisničkog interfejsa serveru

U prethodnom kodu je postojao odgovarajući link koji se odnosio na prikaz nemačke zastave. Za prikaz britanske zastave je:

```
<h:commandLink action="#{localeChanger.englishAction}"
immediate="true">
<h:graphicImage value="/british_flag.gif" style="border:
0px"/>
</h:commandLink>
```

Može se primetiti da svaki od navedenih linkova predstavlja različitu akciju:

localeChanger.englishAction,

Odnosno

localeChanger.germanAction.

Njihove implementacije su:

```
public class ChangeLocaleBean {
public String germanAction() {
FacesContext context =
FacesContext.getCurrentInstance();
context.getViewRoot().setLocale(Locale.GERMAN);
return null;
}
public String englishAction() {
FacesContext context =
FacesContext.getCurrentInstance();
context.getViewRoot().setLocale(Locale.ENGLISH);
return null;
}
}
```

Navedeni deo klase sadrži dosta redudantnog koda, pa je moguće definisati jedinstvena akcija ili osluškivač akcija kako bi se promenio pogled na koren. JSF pruža tri mehanizma za prosleđivanje podataka iz korisničkog interfejsa ka serveru. Ti mehanizmi su u okviru JSP taga:

- f:param
- f:setPropertyActionListener
- f:attribute

U nastavku teksta se opisuju navedeni tagovi.

f:param tag

f:param tag dozvoljava da se parametar poveže sa određenom komponentom. Kada se izvrši povezivanje ovog taga sa komandnom komponentom, kao što su dugme ili link, JSF okruženje prosleđuje vrednosti parametara serveru i to kao zahtevane parametre.Na primer:

```
<h:commandLink immediate="true"
action="#{localeChanger.changeLocale}">
<f:param name="languageCode" value="de"/>
<h:graphicImage value="/german_flag.gif" style="border:
0px"/>
</h:commandLink>
...
<h:commandLink immediate="true"
action="#{localeChanger.changeLocale}">
<f:param name="languageCode" value="en"/>
<h:graphicImage value="/british_flag.gif" style="border:
0px"/>
</h:commandLink>
```

Na serverskoj strani se pristupa zahtevanom parametru languageCode kako bi se postavila lokalna podešavanja:

```
public class ChangeLocaleBean {
public String changeLocale() {
FacesContext context =
FacesContext.getCurrentInstance();
String languageCode = getLanguageCode(context);
context.getViewRoot().setLocale(new
Locale(languageCode));
return null;
}
private String getLanguageCode(FacesContext context) {
Map<String, String> params =
context.getExternalContext().
getRequestParameterMap();
return params.get("languageCode");
}
}
```

f:attribute tag

Drugi način da se prosledi informacija od korisničkog interfejsa ka serveru jeste da se postavi atribut komponente sa f:attribute tagom.

```
<h:commandLink immediate="true"
actionListener="#{localeChanger.changeLocale}">
<f:attribute name="languageCode" value="de"/>
<h:graphicImage value="/german_flag.gif" style="border:
0px"/>
</h:commandLink>
...
<h:commandLink immediate="true"
actionListener="#{localeChanger.changeLocale}">
<f:attribute name="languageCode" value="en"/>
<h:graphicImage value="/british_flag.gif" style="border:
0px"/>
</h:commandLink>
```

U datom primeru se pomoću f:attribute postavlja atribut linka, i to atribut je languageCode i on ima ili vrednost de ili vrednost en. Takođe, koristi se i osluškivač akcija umesto same akcije. Razlog je što osluškivač akcija prosleđuje objekat događaja koji nam omogućava da pristupimo komponenti koja je ispalila događaj.

```
public class ChangeLocaleBean {
public void changeLocale(ActionEvent event) {
UIComponent component = event.getComponent();
String languageCode = getLanguageCode(component);
FacesContext.getCurrentInstance()
.getViewRoot().setLocale(new Locale(languageCode));
}
private String getLanguageCode(UIComponent component) {
Map<String, Object> attrs = component.getAttributes();
return (String) attrs.get("languageCode");
}
}
```

Sada se postavlja kod u atribut komponente, a ne u zahtevane parametre.

f: setPropertyActionListener tag

Pomoću f:setPropertyActionListener taga, JSF postavlja propertyi Bean-a. Na primer:

```
<h:commandLink immediate="true"
action="#{localeChanger.changeLocale}">
<f:setPropertyActionListener
target="#{localeChanger.languageCode}"
value="de"/>
<h:graphicImage value="/german_flag.gif" style="border:
0px"/>
</h:commandLink>
...
<h:commandLink immediate="true"
action="#{localeChanger.changeLocale}">
<f:setPropertyActionListener
target="#{localeChanger.languageCode}"
value="en"/>
<h:graphicImage value="/british_flag.gif" style="border:
0px"/>
</h:commandLink>
```

U prethodnom kodu realizovano je da se property languageCode Bean-a localeChanger postavi sa de ili en. Bean localeChanger je:

```
public class ChangeLocaleBean {
private String languageCode;
public String changeLocale() {
FacesContext context =
FacesContext.getCurrentInstance();
context.getViewRoot().setLocale(new
Locale(languageCode));
return null;
}
public void setLanguageCode(String newValue) {
languageCode = newValue;
}
}
```

Na osnonovu primera, može se zaključiti da je najbolji izbor za podešavanje property-ija languageCode Bean-a localeChanger, zato što rezultuje u najjednostavniju implementaciju klase ChangeLocaleBean.

5.14 Fazni događaji

JSF generiše događaje, koji se zovu fazni događaji (phase events), pre i posle svake faze životnog ciklusa. Ovim događajima upravljaju fazni osluškivači. Fazni događaji, za razliku od prethodna dva događaja, se specificiraju u konfiguracinom fajlu aplikacije, na sledeći način:

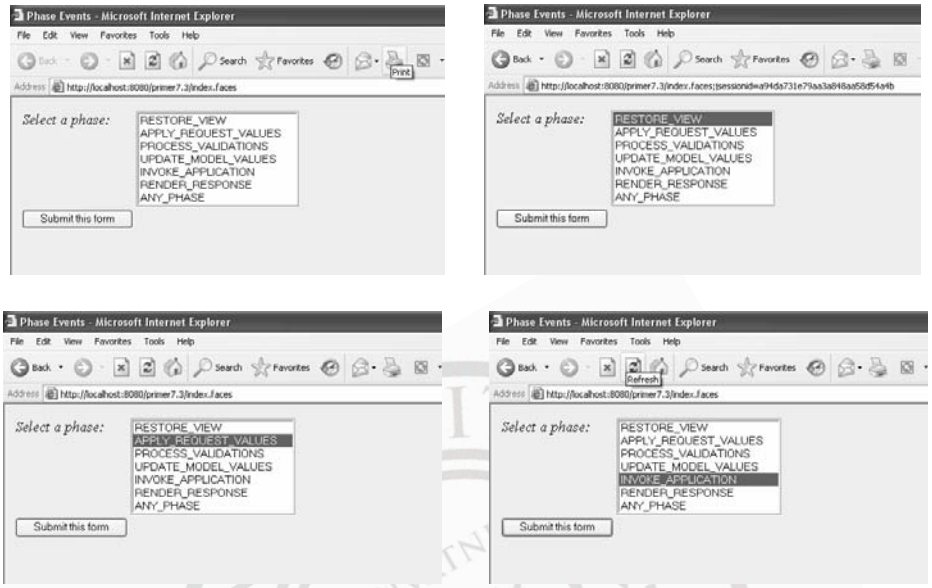
```
<faces-config>
<lifecycle>
<phase-listener>com.corejsf.PhaseTracker</phase-
listener>
</lifecycle>
</faces-config>
```

U navedenom fajlu se može dodati i više osluškivača, ali se oni bude onim redosledom kojim su navedeni u konfiguracionom fajlu.
Fazni osluškivači implementiraju interfejs PhaseListener koji se nalazi u paketu
javax.faces.event. Taj interfejs ima sledeće tri metode:

- PhaseId getPhaseId() – govori implementaciji kako da dostavi fazne događaje osluškivaču; može da vrati vrednost PhaseId.APPLY_REQUEST_VALUES ;
- void afterPhase(PhaseEvent) - poziva se jednom u životnom ciklusu, pre faze Apply Request Values
- void beforePhase(PhaseEvent) - poziva se jednom u životnom ciklusu, posle faze Apply Request Values

Fazni osluškivači su pogodni za debagovanje i specijalno ponašanje.

Sledeća aplikacija ilustruje korišćenje faznih osluškivača.



Slika 30. Primer aplikacije sa faznim osluškivačina

Navedena aplikacija ima jedan osluškivač faze koji loguje poruke sa logerom (logger).

```
public class PhaseTracker implements PhaseListener {
...
private static final Logger logger =
Logger.getLogger("com.corejsf.phases");

...
public void beforePhase(PhaseEvent e) {
logger.info("BEFORE " + e.getPhaseId());
}
public void afterPhase(PhaseEvent e) {
logger.info("AFTER " + e.getPhaseId());
}
}
```

Fazni ID za osluškivače se postavlja sa listbox koji je definisan na sledeći način:

```
<h:selectOneListbox onchange="submit()"
valueChangeListener="#{form.phaseChange}">
<f:selectItems value="#{form.phases}"/>
</h:selectOneListbox>
```

Kada se selektuje faza i aktivira potvrdno dugme, fazni ID faznog osluškivača se setuje od strane osluškivača koji reaguje na promene vrednosti.

```
public class FormBean {
...
// VALUE CHANGE LISTENER: phaseChange
public void phaseChange(ValueChangeEvent e) {
// get a reference to the current lifecycle
...
PhaseListener[]                listeners                =
lifecycle.getPhaseListeners();
for (int i=0; i < listeners.length; i++) {
PhaseListener listener = listeners[i];
if (listener instanceof com.corejsf.PhaseTracker)
((com.corejsf.PhaseTracker) listener).setPhase((String)
e.getNewValue());
}
}
}
```

Izlaz osluškivača (taj izlaz se upisuje u servletov log fajl pri startovanju aplikacije):

```
INFO: BEFORE RESTORE_VIEW 1
INFO: AFTER RESTORE_VIEW 1
INFO: BEFORE RENDER_RESPONSE 6
INFO: AFTER RENDER_RESPONSE 6
```

Ukoliko se aktivira dugme potvrde, bez selektovanja faze, izlaz je sledeći:

```
INFO: BEFORE RESTORE_VIEW 1
INFO: AFTER RESTORE_VIEW 1
INFO: BEFORE APPLY_REQUEST_VALUES 2
INFO: AFTER APPLY_REQUEST_VALUES 2
INFO: BEFORE PROCESS_VALIDATIONS 3
INFO: AFTER PROCESS_VALIDATIONS 3
```

```
INFO: BEFORE UPDATE_MODEL_VALUES 4
INFO: AFTER UPDATE_MODEL_VALUES 4
INFO: BEFORE INVOKE_APPLICATION 5
INFO: AFTER INVOKE_APPLICATION 5
INFO: BEFORE RENDER_RESPONSE 6
INFO: AFTER RENDER_RESPONSE 6
```

Ukoliko se selektuje Apply Request Values iz liste i aktivira dugme potvrde dobija se rezultat:

```
INFO: BEFORE RESTORE_VIEW 1
INFO: AFTER RESTORE_VIEW 1
INFO: BEFORE APPLY_REQUEST_VALUES 2
INFO: AFTER APPLY_REQUEST_VALUES 2
INFO: BEFORE PROCESS_VALIDATIONS 3
```

Ukoliko se postavi fazni ID osluškivača na Apply Request Values pre početka životnog ciklusa, onda je rezultat:

```
INFO: BEFORE APPLY_REQUEST_VALUES 2
INFO: AFTER APPLY_REQUEST_VALUES 2
```

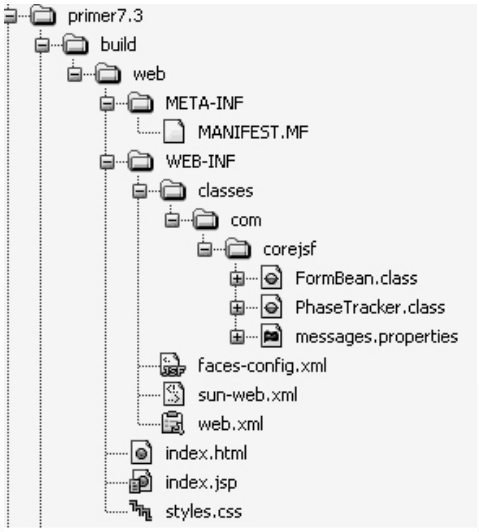
Ukoliko se selektuje Invoke Application i aktivira dugme potvrde, rezultat je:

```
INFO: BEFORE APPLY_REQUEST_VALUES 2
INFO: AFTER APPLY_REQUEST_VALUES 2
INFO: BEFORE INVOKE_APPLICATION 5
INFO: AFTER INVOKE_APPLICATION 5
```

Ukoliko se još jednom aktivira dugme potvrde izlaz je:

```
INFO: BEFORE INVOKE_APPLICATION 5
INFO: AFTER INVOKE_APPLICATION 5
```

Na sledećoj slici prikazana je strukturu direktorijuma koji se koriste u sledećem primeru sa faznim osluškivačima



Slika 31. Struktura direktorijuma za primer sa faznim osluškivačima

U okviru stranice phase-tracker/web/index.jsp koristi se title tag da bi se prikazala odgovarajuća vrednost iz properties fajla, zatim u okviru taga body pravi tabela sa dve kolone u kojoj se ispisuju vrednost iz properties fajla i Listboxa, u prvoj vrsti, odnosno komandno dugme u drugoj vrsti.U okviru Listbox-a definiše se fazni osluškivač.

```
1. <html>
2. <%@ taglib uri="http://java.sun.com/jsf/core"
prefix="f" %>
3. <%@ taglib uri="http://java.sun.com/jsf/html"
prefix="h" %>
4. <f:view beforePhase="#{form.beforePhase}"
5. afterPhase="#{form.afterPhase}">
6. <head>
7. <link href="styles.css" rel="stylesheet"
type="text/css"/>
8. <title>
9. <h:outputText value="#{msgs.indexWindowTitle}"/>
10. </title>
11. </head>
12.
13. <body>
14. <h:form>
15. <h:panelGrid columns="2"
columnClasses="phaseFormColumns">
16. <h:outputText value="#{msgs.phasePrompt}"/>
```

```
17.
18. <h:selectOneListbox
valueChangeListener="#{form.phaseChange}">
19. <f:selectItems value="#{form.phases}"/>
20. </h:selectOneListbox>
21.
22. <h:commandButton value="#{msgs.submitPrompt}"/>
23. </h:panelGrid>
24. </h:form>
25. </body>
26. </f:view>
27. </html>
```

U klasi phase-tracker/src/java/com/corejsf/FormBean.java definiše se fazni osluškivač, odnosno definišu se faze u formi SelectItem[], zatim se definišu metode phaseChange, afterPhase, beforePhase i showEvent.
U okviru metode phaseChange definiše se fabrika životnog ciklusa iz koje se prilazi odgovarajućem životnom ciklusu, potom se za dati ciklus određuje fazni osluškivač i ispituje da li je on instanca klase PhaseTracker i ako jeste postavlja se faza na odgovarajuću vrednost.

```
1. package com.corejsf;
2.
3. import javax.faces.FactoryFinder;
4. import javax.faces.event.PhaseEvent;
5. import javax.faces.event.PhaseListener;
6. import javax.faces.event.ValueChangeEvent;
7. import javax.faces.lifecycle.Lifecycle;
8. import javax.faces.lifecycle.LifecycleFactory;
9. import javax.faces.model.SelectItem;
10.
11. public class FormBean {
12. private SelectItem[] phases = {
13. new SelectItem("RESTORE_VIEW"),
14. new SelectItem("APPLY_REQUEST_VALUES"),
15. new SelectItem("PROCESS_VALIDATIONS"),
16. new SelectItem("UPDATE_MODEL_VALUES"),
17. new SelectItem("INVOKE_APPLICATION"),
18. new SelectItem("RENDER_RESPONSE"),
19. new SelectItem("ANY_PHASE"),
20. };
21.
22. public SelectItem[] getPhases() { return phases; }
23.
24. public void phaseChange(ValueChangeEvent e) {
```

```
25. LifecycleFactory factory = (LifecycleFactory)
FactoryFinder.getFactory(
26. FactoryFinder.LIFECYCLE_FACTORY);
27. Lifecycle lifecycle =
factory.getLifecycle(LifecycleFactory.
28. DEFAULT_LIFECYCLE);
29.
30. PhaseListener[] listeners =
lifecycle.getPhaseListeners();
31. for (int i = 0; i < listeners.length; i++) {
32. PhaseListener listener = listeners[i];
33. if (listener instanceof com.corejsf.PhaseTracker)
34. ((com.corejsf.PhaseTracker) listener).setPhase(
35. (String) e.getNewValue());
36. }
37. }
38. public void afterPhase(PhaseEvent event) {
39. System.out.println("AFTER PHASE " +
showEvent(event));
40. }
41. public void beforePhase(PhaseEvent event) {
42. System.out.println("BEFORE PHASE " +
showEvent(event));
43. }
44. private String showEvent(PhaseEvent event) {
45. return "Phase Event: " + event.getPhaseId();
46. }
47. }
```

Klasa phase-tracker/src/java/com/corejsf/PhaseTracker.java implementira interfejs PhaseListener. U okviru nje se definišu metode za dohvatanje i postavljanje vrednosti faze, kao i metode beforePhase, afterPhase.

```
1. package com.corejsf;
2.
3. import java.util.logging.Logger;
4. import javax.faces.context.FacesContext;
5. import javax.faces.event.PhaseEvent;
6. import javax.faces.event.PhaseListener;
7. import javax.faces.event.PhaseId;
8.
9. public class PhaseTracker implements PhaseListener {
10 private static final String
PHASE_PARAMETER="com.corejsf.phaseTracker.phase";
11. private static final Logger logger =
Logger.getLogger("com.corejsf.phases");
```

```
12. private static String phase = null;
13.
14. public void setPhase(String newValue) { phase =
newValue; }
15.
16. public PhaseId getPhaseId() {
17. if (phase == null) {
18. FacesContext context =
FacesContext.getCurrentInstance();
19. phase = (String)
context.getExternalContext().getInitParameter(
20. PHASE_PARAMETER);
21. }
22. PhaseId phaseId = PhaseId.ANY_PHASE;
23.
24. if (phase != null) {
25. if ("RESTORE_VIEW".equals(phase))
26. phaseId = PhaseId.RESTORE_VIEW;
27. else if ("APPLY_REQUEST_VALUES".equals(phase))
28. phaseId = PhaseId.APPLY_REQUEST_VALUES;
29. else if ("PROCESS_VALIDATIONS".equals(phase))
30. phaseId = PhaseId.PROCESS_VALIDATIONS;
31. else if ("UPDATE_MODEL_VALUES".equals(phase))
32. phaseId = PhaseId.UPDATE_MODEL_VALUES;
33. else if ("INVOKE_APPLICATION".equals(phase))
34. phaseId = PhaseId.INVOKE_APPLICATION;
35. else if ("RENDER_RESPONSE".equals(phase))
36. phaseId = PhaseId.RENDER_RESPONSE;
37. else if ("ANY_PHASE".equals(phase))
38. phaseId = PhaseId.ANY_PHASE;
39. }
40. return phaseId;
41. }
42. public void beforePhase(PhaseEvent e) {
43. logger.info("BEFORE " + e.getPhaseId());
44. }
45. public void afterPhase(PhaseEvent e) {
46. logger.info("AFTER " + e.getPhaseId());
47. }
48. }
```

U okviru properties fajla phase-tracker/web/WEB-INF/faces-config.xml kojise koristi, definišu se ime, klasa i opseg Bean-a. Pored njih definiše se i fazni osluškivač u okviru taga lifecycle.

```
1. <?xml version="1.0"?>
2. <faces-config
xmlns="http://java.sun.com/xml/ns/javaee"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
5. http://java.sun.com/xml/ns/javaee/web-
facesconfig_1_2.xsd"
6. version="1.2">
7. <application>
8. <resource-bundle>
9. <base-name>com.corejsf.messages</base-name>
10. <var>msgs</var>
11. </resource-bundle>
12. </application>
13.
14. <managed-bean>
15. <managed-bean-name>form</managed-bean-name>
16. <managed-bean-class>com.corejsf.FormBean</managed-
bean-class>
17. <managed-bean-scope>session</managed-bean-scope>
18. </managed-bean>
19.
20. <lifecycle>
21. <phase-listener>com.corejsf.PhaseTracker</phase-
listener>
22. </lifecycle>
23. </faces-config>
```

Definiše se i properties fajl phase-tracker/src/java/com/corejsf/messages.properties.

```
1. indexWindowTitle=Phase Events
2. phasePrompt=Select a phase:
3. submitPrompt=Submit this form
```

Klijentske strane korsite css fajl phase-tracker/web/styles.css

```
1. body {
2. background: #eee;
3. }
4. .phaseFormColumns {
```

```
5. vertical-align: top;
6. font-style: italic;
7. font-size: 1.1em;
8. }
9. .columns {
10. vertical-align: top;
11. }
```

Na kraju ovog poglavlja će se prikazati primer koji koristi prozore sa tabovima.U ovom primeru se razmatra obrada događaja i proširenі aspekti JSF HTML tagova.Ti proširenі aspekti uključuju sledeće

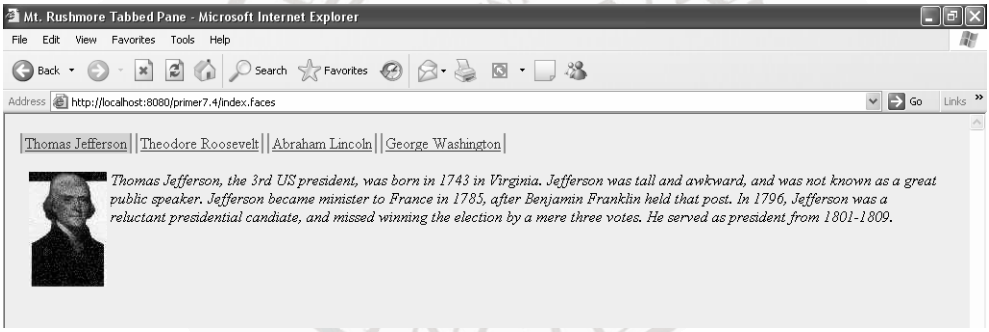
- umrežene h:panelGrid tagove
- korišćenje facets modula
- specifično tab indeksiranje
- dodavanje opisa alatki komponentama sa atributom title
- dinamičko određivanje stilskih klasa
- korišćenje akcionih događaja
- opciono izvođenje
- statičko uključivanje JSF stranica

JSF 1.2 verzija ne podržava prozore sa tabovima, tako da ako se želi korišćenje tab mehanizma u okviru aplikacija, postoje dva izbora: implementacija klasičnih komponenti sa postojećim tagovima i mogućnost izbora sa binom podrške, i na taj način bi se kreirao tab.
JSF stranica koja koristi tabove izgleda :

```
<h:form>
<%-- Tabs --%>
<h:panelGrid                          styleClass="tabbedPane"
columnClasses="displayPanel">
<f:facet name="header">
<h:panelGrid columns="4" styleClass="tabbedPaneHeader">
<h:commandLink tabindex="1"
title="#{tp.jeffersonTooltip}"
styleClass="#{tp.jeffersonStyle}"
actionListener="#{tp.jeffersonAction}">
<h:outputText value="#{msgs.jeffersonTab}"/>
</h:commandLink>
...
</h:panelGrid>
</f:facet>
```

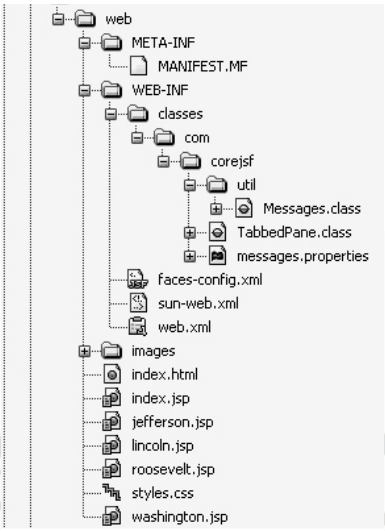
```
<%-- Main panel --%>
<%@ include file="jefferson.jsp" %>
<%@ include file="roosevelt.jsp" %>
<%@ include file="lincoln.jsp" %>
<%@ include file="washington.jsp" %>
</h:panelGrid>
</h:form>
```

Prozor sa tabom se implementira pomoću h:panelGrid objektaa. S obzirom da se ne specificira columns atribut, panel ima samo jednu kolonu.Heder panela, koji je definisan sa f:facet tagom. On sadrži tabove koji su implementirani sa drugim h:panelGrid tagom koji sadrži h:commandLink tag za svaki tab. Jedini red u panelu sadrži sadržaj za selektovanje taba.
Kada korisnik selektuje tab, pridruženi akcioni osluškivač za komandni link se budi i modifikuje podatke koji se nalaze u Bean-u podrške. U navedenom primeru koriste se različiti CSS stilovi za selektovani tab, pa zato styleClass atribut za svaki tag je praćen odgovarajućim Bean-om podrške.



Slika 32. Izgled aplikacije u datom primeru

Sledeća slika pokazuje strukturu direktorijuma koji se koristi u datom primeru.



Slika 33. Izgled direktorijuma u navedenom primeru

Početna stranica aplikacije je tabbedpane/web/index.jsp, gde se okviru taga title ispisuje vrednost iz properties fajla. Zatim se u okviru taga body definiše tabela sa pet kolona. U prvoj vrsti se ispisuje komandni link za odgovarajućeg predsednika, i za svaki link se vezuje odgovarajući osluškivač događaja. Zatim se ubacuju sadržaji odgovarajućih tabova, u zavisnosti koji je link aktiviran , odnosno koji je tab izabran.

```
1. <html>
2. <%@ taglib uri="http://java.sun.com/jsf/core"
prefix="f" %>
3. <%@ taglib uri="http://java.sun.com/jsf/html"
prefix="h" %>
4.
5. <f:view>
6. <head>
7. <link href="styles.css" rel="stylesheet"
type="text/css"/>
8. <title>
9. <h:outputText value="#{msgs.windowTitle}"/>
10. </title>
11. </head>
12.
13. <body>
```

```
14. <h:form>
15. <h:panelGrid styleClass="tabbedPane"
columnClasses="displayPanel">
16. <%-- Tabs --%>
17.
18. <f:facet name="header">
19. <h:panelGrid columns="5"
styleClass="tabbedPaneHeader">
20.
21. <h:commandLink tabindex="1"
22. title="#{tp.jeffersonTooltip}"
23. styleClass="#{tp.jeffersonStyle}"
24. actionListener="#{tp.jeffersonAction}">
25.
26. <h:outputText value="#{msgs.jeffersonTabText}"/>
27. </h:commandLink>
28.
29. <h:commandLink tabindex="2"
30. title="#{tp.rooseveltTooltip}"
31. styleClass="#{tp.rooseveltStyle}"
32. actionListener="#{tp.rooseveltAction}">
33.
34. <h:outputText value="#{msgs.rooseveltTabText}"/>
35. </h:commandLink>
36.
37. <h:commandLink tabindex="3"
38. title="#{tp.lincolnTooltip}"
39. styleClass="#{tp.lincolnStyle}"
40. actionListener="#{tp.lincolnAction}">
41.
42. <h:outputText value="#{msgs.lincolnTabText}"/>
43. </h:commandLink>
44.
45. <h:commandLink tabindex="4"
46. title="#{tp.washingtonTooltip}"
47. styleClass="#{tp.washingtonStyle}"
48. actionListener="#{tp.washingtonAction}">
49.
50. <h:outputText value="#{msgs.washingtonTabText}"/>
51. </h:commandLink>
52. </h:panelGrid>
53. </f:facet>
54.
55. <%-- Tabbed pane content --%>
56.
57. <%@ include file="washington.jsp" %>
58. <%@ include file="roosevelt.jsp" %>
```

```
59. <%@ include file="lincoln.jsp" %>
60. <%@ include file="jefferson.jsp" %>
61. </h:panelGrid>
62. </h:form>
63. </body>
64. </f:view>
65. </html>
```

Zatim se kreiraju odgovarajuće strane .
U prvoj (tabbedpane/web/jefferson.jsp) se definiše odgovrajuća stranica koja se ubacuje u stranicu index.jsp, kada se selektuje odgovarajući tab.

```
1. <h:panelGrid columns="2"
columnClasses="presidentDiscussionColumn"
2. rendered="#{tp.jeffersonCurrent}">
3.
4. <h:graphicImage value="/images/jefferson.jpg"/>
5. <h:outputText value="#{msgs.jeffersonDiscussion}"
6. styleClass="tabbedPaneContent"/>
7.
8. </h:panelGrid>
```

Sledeća stranica je tabbedpane/web/roosevelt.jsp:

```
1. <h:panelGrid columns="2"
columnClasses="presidentDiscussionColumn"
2. rendered="#{tp.rooseveltCurrent}">
3.
4. <h:graphicImage value="/images/roosevelt.jpg"/>
5. <h:outputText value="#{msgs.rooseveltDiscussion}"
6. styleClass="tabbedPaneContent"/>
7.
8. </h:panelGrid>
```

Postoji i tabbedpane/web/lincoln.jsp:

```
1. <h:panelGrid columns="2"
columnClasses="presidentDiscussionColumn"
2. rendered="#{tp.lincolnCurrent}">
3.
4. <h:graphicImage value="/images/lincoln.jpg"/>
5. <h:outputText value="#{msgs.lincolnDiscussion}"
6. styleClass="tabbedPaneContent"/>
7.
8. </h:panelGrid>
```

Poslednja je tabbedpane/web/washington.jsp

```
1. <h:panelGrid columns="2"
columnClasses="presidentDiscussionColumn"
2. rendered="#{tp.washingtonCurrent}">
3.
4. <h:graphicImage value="/images/washington.jpg"/>
5. <h:outputText value="#{msgs.washingtonDiscussion}"
6. styleClass="tabbedPaneContent"/>
7.
8. </h:panelGrid>
```

U fajlu tabbedpane/src/java/com/corejsf/messages.properties su definisani tekstualni delovi aplikacije

```
1. windowTitle=Mt. Rushmore Tabbed Pane
2. lincolnTooltip=Abraham Lincoln
3. lincolnTabText=Abraham Lincoln
4. lincolnDiscussion=President Lincoln was known as the
Great Emancipator because \
5. he was instrumental in abolishing slavery in the
United States. He was born \
6. into a poor family in Kentucky in 1809, elected
president in 1860, and \
7. assassinated by John Wilkes Booth in 1865.
8.
9. washingtonTooltip=George Washington
10. washingtonTabText=George Washington
11. washingtonDiscussion=George Washington was the first
president of the United \
12. States. He was born in 1732 in Virginia, was elected
Commander in Chief of \
13. the Continental Army in 1775, and forced the
surrender of Cornwallis at Yorktown \
14. in 1781. He was inaugurated on April 30, 1789.
15.
16. rooseveltTooltip=Theodore Roosevelt
17. rooseveltTabText=Theodore Roosevelt
18. rooseveltDiscussion=Theodore Roosevelt was the 26th
president of the United \
19. States. In 1901 he became president after the
assassination of President \
20. McKinley. At only 42, he was the youngest president
in U.S. history.
21.
22. jeffersonTooltip=Thomas Jefferson
```

```
23. jeffersonTabText=Thomas Jefferson
24. jeffersonDiscussion=Thomas Jefferson, the 3rd U.S.
president, was born in \
25. 1743 in Virginia. Jefferson was tall and awkward,
and was not known as a \
26. great public speaker. Jefferson became minister to
France in 1785, after \
27. Benjamin Franklin held that post. In 1796, Jefferson
was a reluctant \
28. presidential candiate, and missed winning the
election by a mere three votes. \
29. He served as president from 1801 to 1809.
```

U aplikaciji se koristi i CSS koji je definisan u okviru fajla tabbedpane/web/styles.css

```
1. body {
2. background: #eee;
3. }
4. .tabbedPaneHeader {
5. vertical-align: top;
6. text-align: left;
7. padding: 2px 2px 0px 2px;
8. }
9. .tabbedPaneText {
10. font-size: 1.0em;
11. font-style: regular;
12. padding: 3px;
13. border: thin solid CornflowerBlue;
14. }
15. .tabbedPaneTextSelected {
16. font-size: 1.0em;
17. font-style: regular;
18. padding: 3px;
19. background: PowderBlue;
20. border: thin solid CornflowerBlue;
21. }
22. .tabbedPane {
23. vertical-align: top;
24. text-align: left;
25. padding: 10px;
26. }
27. .displayPanel {
28. vertical-align: top;
29. text-align: left;
30. padding: 10px;
```

```
31. }
32. .tabbedPaneContent {
33. width: 100%;
34. height: 100%;
35. font-style: italic;
36. vertical-align: top;
37. text-align: left;
38. font-size: 1.2m;
39. }
40. .presidentDiscussionColumn {
41. vertical-align: top;
42. text-align: left;
43. }
```

Kao i u ostalim JSF aplikacijama konfiguracija je definisana u okviru tabbedpane/web/WEB-INF/faces-config.xml

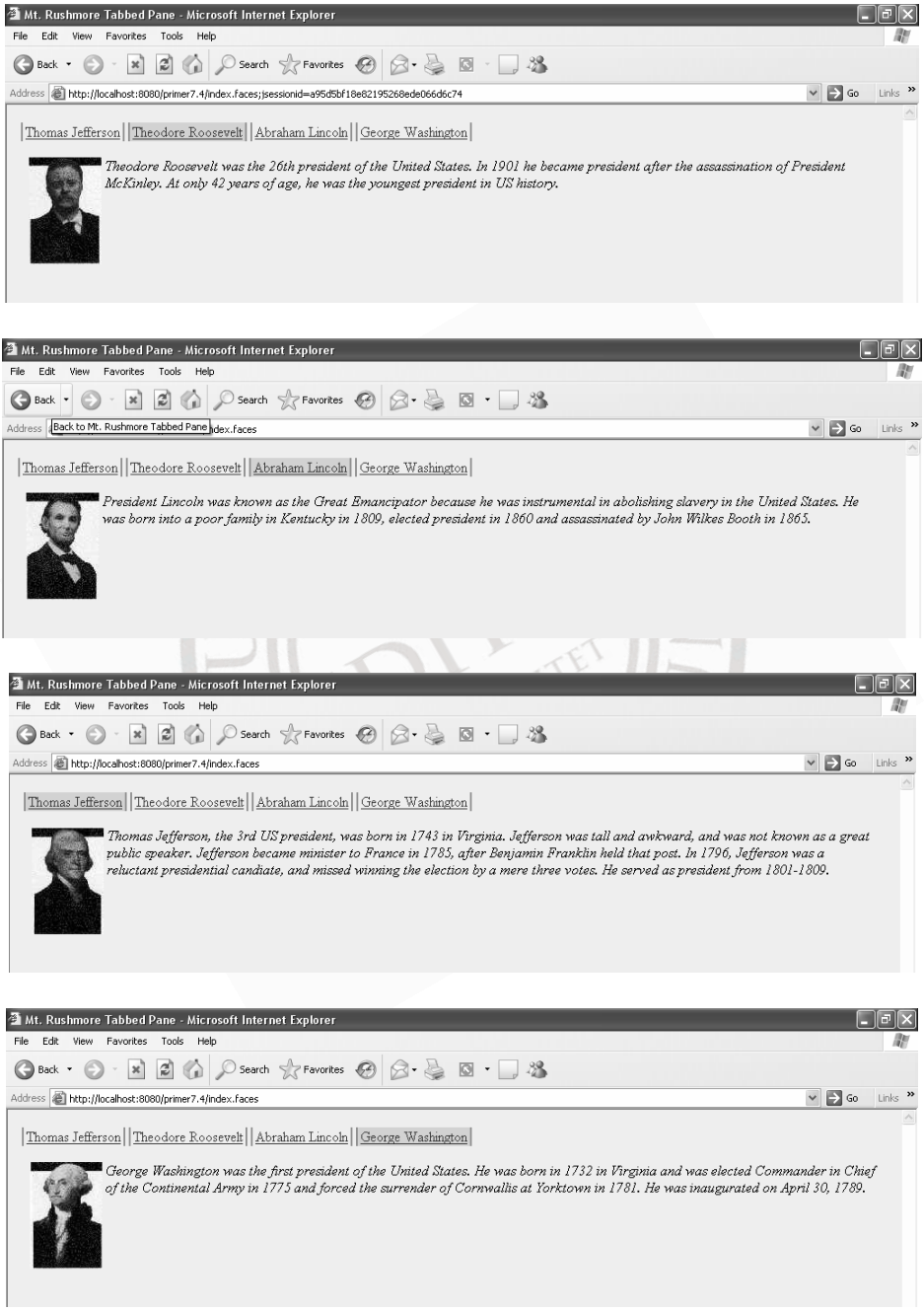
```
1. <?xml version="1.0"?>
2. <faces-config
xmlns="http://java.sun.com/xml/ns/javaee"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
5. http://java.sun.com/xml/ns/javaee/web-
facesconfig_1_2.xsd"
6. version="1.2">
7. <application>
8. <resource-bundle>
9. <base-name>com.corejsf.messages</base-name>
10. <var>msgs</var>
11. </resource-bundle>
12. </application>
13.
14. <managed-bean>
15. <managed-bean-name>tp</managed-bean-name>
16. <managed-bean-class>com.corejsf.TabbedPane</managed-
bean-class>
17. <managed-bean-scope>session</managed-bean-scope>
18. </managed-bean>
19. </faces-config>
```

U okviru klase tabbedpane/src/java/com/corejsf/TabbedPane.java definišu se konstante koje se odnose na predsednike, zatim se definišu i tabovi, potom se definišu metode za dohvatanje CSS stila, za određivanje tekućeg taba, metode za postavljanje tekućeg taba, kao i metode za dohvatanje odgovarjućeg opisa alatki (tabTooltips).

```
1. package com.corejsf;
2.
3. import javax.faces.event.ActionEvent;
4.
5. public class TabbedPane {
6. private int index;
7. private static final int JEFFERSON_INDEX = 0;
8. private static final int ROOSEVELT_INDEX = 1;
9. private static final int LINCOLN_INDEX = 2;
10. private static final int WASHINGTON_INDEX = 3;
11.
12. private String[] tabs = { "jeffersonTabText",
"rooseveltTabText",
13. "lincolnTabText", "washingtonTabText", };
14.
15. private String[] tabTooltips = { "jeffersonTooltip",
"rooseveltTooltip",
16. "lincolnTooltip", "washingtonTooltip" };
17.
18. public TabbedPane() {
19. index = JEFFERSON_INDEX;
20. }
21.
22. // action listeners that set the current tab
23.
24. public void jeffersonAction(ActionEvent e) { index =
JEFFERSON_INDEX; }
25. public void rooseveltAction(ActionEvent e) { index =
ROOSEVELT_INDEX; }
26. public void lincolnAction(ActionEvent e) { index =
LINCOLN_INDEX; }
27. public void washingtonAction(ActionEvent e) { index
= WASHINGTON_INDEX; }
28.
29. // CSS styles
30.
31. public String getJeffersonStyle() { return
getCSS(JEFFERSON_INDEX); }
32. public String getRooseveltStyle() { return
getCSS(ROOSEVELT_INDEX); }
33. public String getLincolnStyle() { return
getCSS(LINCOLN_INDEX); }
34. public String getWashingtonStyle() { return
getCSS(WASHINGTON_INDEX); }
35.
36. private String getCSS(int forIndex) {
```

```
37. return forIndex == index ? "tabbedPaneTextSelected"
: "tabbedPaneText";
38. }
39.
40. // methods for determining the current tab
41.
42. public boolean isJeffersonCurrent() { return index
== JEFFERSON_INDEX; }
43. public boolean isRooseveltCurrent() { return index
== ROOSEVELT_INDEX; }
44. public boolean isLincolnCurrent() { return index ==
LINCOLN_INDEX; }
45. public boolean isWashingtonCurrent() { return index
== WASHINGTON_INDEX; }
46.
47. // methods that get tooltips for titles
48.
49. public String getJeffersonTooltip() {
50. return com.corejsf.util.Messages.getString(
51. "com.corejsf.messages",
tabTooltips[JEFFERSON_INDEX], null);
52. }
53. public String getRooseveltTooltip() {
54. return com.corejsf.util.Messages.getString(
55. "com.corejsf.messages",
tabTooltips[ROOSEVELT_INDEX], null);
56. }
57. public String getLincolnTooltip() {
58. return com.corejsf.util.Messages.getString(
59. "com.corejsf.messages", tabTooltips[LINCOLN_INDEX],
null);
60. }
61. public String getWashingtonTooltip() {
62. return com.corejsf.util.Messages.getString(
63. "com.corejsf.messages",
tabTooltips[WASHINGTON_INDEX], null);
64. }
65. }
```

Rezultati su:

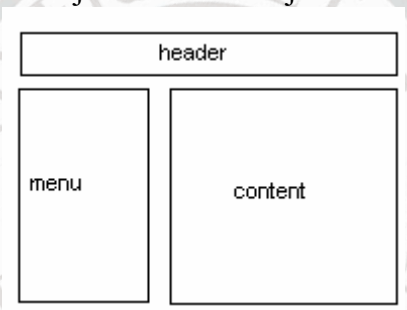


Slika 34. Rezultati rada aplikacije

5.15 Isti izgled svih stranica aplikacije

U prvim poglavljima knjige naglašeno je da je korisnički interfejs je obično deo aplikacije koji se najviše menja, pa je uvek upravo kritični deo aplikacije da se kreira fleksibilan i proširiv interfejs. U ovom poglavlju se govori prvo standardnim JSP mehanizmima koji omogućavaju da se uključi zajednički sadržaj u JSF aplikaciji. Zatim se govori o Apache Tiles paketu, koji se koristit za zajednički layout stranica aplikacija i takoše koristi se još i u drugim framework-cima kao što je STRUTS.
Rečeno je da mnogi popularni sajtovi, kao što su nytimes.com, java.sun.com, ili amazon.com, koriste isti uobičajeni layout za svoje web stranice.

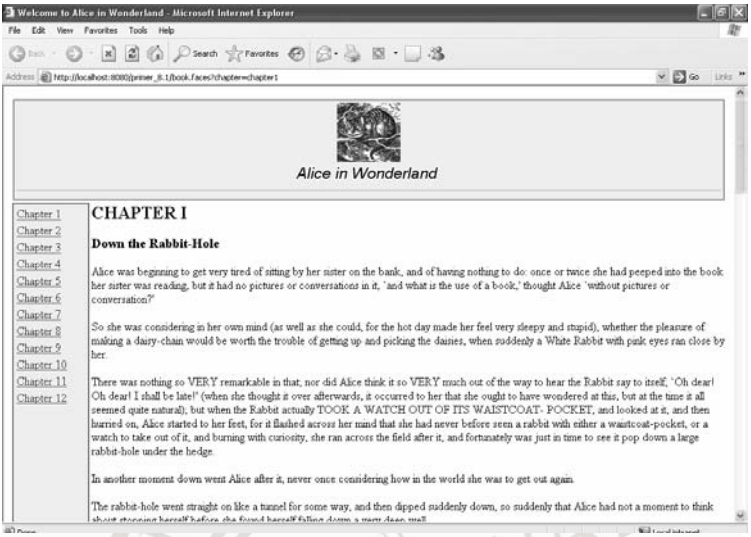
Tipičan layout web stranica je dat na sledećoj slici:



Slika 35. Tipičan layout web stranica

Kako bi se ilustrovali upotreba layout i zajedničkih komponenti uz korišćenje

Tiles paketa, realizovaće se dve aplikacije.



Slika 36. Primer prve aplikacije

Prva aplikacija za pregled knjiga je intuitivna. Ako se klikne na link za poglavlje, prikazaće se dato poglavlje u onom delu stranice koja je predviđena za to. Sledeća aplikacija će biti njen nastavak, i ona će omogućiti da se vidi više od jedne knjige. Knjige se mogu selektovati iz menija koji se nalazi na vrhu stranice.

Takođe, prva aplikacija je ograničena i po pitanju opsega. Ona podržava samo jednu knjigu, realizovanu kao Bean koji je definisan u okviru JSF konfiguracije. Ovaj Bean ima sledeće property-ije:

- titleKey
- image
- numChapters
- chapterKeys

Property titleKey reprezentuje ključ u okviru properties fajla sa spoljašnjim resursima i na klijentskoj strani se prikazuje sa:

```
<h:outputText value="#{msgs[book.titleKey]}"/>
```

gde je msgs objekat koji predstavlja definisani properties fajla sa spoljašnjim resursima.

Property image je takođe String. Aplikacija interpretira string kao URL i učitava je u header stranice aplikacije kao:

```
<h:graphicImage url="#{book.image}"/>
```

Property chapterKeys sadrži read-only listu ključeva, za svako poglavlje i koristi se na sledeći način

```
<h:dataTable value="#{book.chapterKeys}"
var="chapterKey">
<h:commandLink>
<h:outputText value="#{msgs[chapterKey]}"/>
...
</h:commandLink>
</h:dataTable>
```

Klasa Book koristi property numChapters kako bi izračunala ključ poglavlja. Sledeći kod pokazuje kako je definisana Book klasa u faces-config.xml:

```
<faces-config>
<!-- The book -->
<managed-bean>
<managed-bean-name>book</managed-bean-name>
<managed-bean-class>com.corejsf.Book</managed-bean-
class>
<managed-bean-scope>request</managed-bean-scope>
<managed-property>
<property-name>titleKey</property-name>
<value>aliceInWonderland</value>
</managed-property>
<managed-property>
<property-name>image</property-name>
<value>cheshire.jpg</value>
</managed-property>
320
<managed-property>
<property-name>numChapters</property-name>
<property-class>java.lang.Integer</property-class>
<value>12</value>
</managed-property>
</managed-bean>
</faces-config>
```

U ovom primeru su JSF stranice implementirane pomoću iste strukture.

```
<!-- A panel grid, which resides in a form, for the
entire page --%>
<h:panelGrid columns="2" styleClass="book"
columnClasses="menuColumn, chapterColumn">

<!-- The header, containing an image, title, and
horizontal rule --%>

<f:facet name="header">
<h:panelGrid columns="1" styleClass="bookHeader">
<h:graphicImage value="#{book.image}"/>
<h:outputText value="#{msgs[book.titleKey]}"
styleClass='bookTitle'/>
<hr>
</h:panelGrid>
</f:facet>

<!-- Column 1 of the panel grid: The menu, which
consists of chapter links --%>

<h:dataTable value="#{book.chapterKeys}"
var="chapterKey"
styleClass="links" columnClasses="linksColumn">
<h:column>
<h:commandLink>
<h:outputText value="#{msgs[chapterKey]}"/>
<f:param name="chapter" value="#{chapterKey}"/>
</h:commandLink>
</h:column>
</h:dataTable>

<!-- Column 2 of the panel grid: The chapter content --
%>

<c:import url="${param.chapter}.html"/>
</h:panelGrid>
```

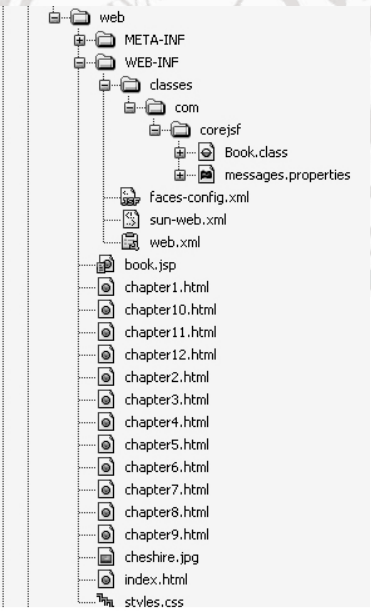
Kao što se iz prethodnog koda može videti ova stranica je implementirana pomoću panel grid-a koji ima 2 kolone. Region header-a sadrži sliku, tekst, and HTML horizontalnu liniju. Panel grid ima samo jedan red, pri čemu je meni zauzeo levu kolonu i tekuće poglavlje se nalazi u desnoj koloni. Kako je definisano i ranije naglašeno Book.getChapterKeys() kao rezultat vraća listu stringova, koja izgleda kao:

```
chapter1
chapter2
...
chapterN
```

ChapterN predstavlja poslednje poglavlje knjige. U fajlu sa spoljašnjim resursima definišu se vrednosti za te ključeve:

```
chapter1=Chapter 1
chapter2=Chapter 2
...
```

Da bi se kreirao link za poglavlje, koristi se tag h:dataTable koji izvršava iteracije kroz book's chapter ključeve. Za svako poglavlje, kreira se link čiji tekst odgovara vrednosti ključa, pomoću izraza: #{msgs[chapterKey]}.
Desna kolona je rezervisana za sadržaj poglavlja. Sadržaj se uključuje pomoću JSTL c:import taga.
Struktura direktorijuma data na sledećoj slici se koristi u primeru.



Slika 37. Struktura direktorijuma iz primera

U okviru stranice book-viewer/web/book.jsp se definiše tabela sa dve kolone koja služi za definisanje header-a. U okviru nje postoji sliku za datu knjigu i ispisivanje naziva knjige, za dati Bean. Zatim se definišu tabela podataka, koja služi za kreiranje linkova za poglavlja. U okviru nje je definisan i komandni link. Na kraju se definiše da se za dato selektovano poglavlje, ubaci odgovarajuća stranica.

```
1. <html>
2. <%@ taglib uri="http://java.sun.com/jsp/jstl/core"
prefix="c" %>
3. <%@ taglib uri="http://java.sun.com/jsf/core"
prefix="f" %>
4. <%@ taglib uri="http://java.sun.com/jsf/html"
prefix="h" %>
5.
6. <f:view>
7. <f:loadBundle basename="com.corejsf.messages"
var="msgs"/>
8. <head>
9. <link href="styles.css" rel="stylesheet"
type="text/css"/>
10. <title><h:outputText
value="#{msgs.bookWindowTitle}"/></title>
11. </head>
12.
13. <body>
14. <h:form>323
15. <h:panelGrid columns="2" styleClass="book"
16. columnClasses="menuColumn, chapterColumn">
17. <f:facet name="header">
18. <h:panelGrid columns="1" styleClass="bookHeader">
19. <h:graphicImage value="#{book.image}"/>
20. <h:outputText value="#{msgs[book.titleKey]}"
21. styleClass='bookTitle'/>
22. <hr/>
23. </h:panelGrid>
24. </f:facet>
25.
26. <h:dataTable value="#{book.chapterKeys}"
var="chapterKey"
27. styleClass="links" columnClasses="linksColumn">
28. <h:column>
29. <h:commandLink>
30. <h:outputText value="#{msgs[chapterKey]}"/>
31. <f:param name="chapter" value="#{chapterKey}"/>
32. </h:commandLink>
```

```
33. </h:column>
34. </h:dataTable>
35.
36. <c:import url="${param.chapter}.html"/>
37. </h:panelGrid>
38. </h:form>
39. </body>
40. </f:view>
41. </html>
```

U book-viewer/web/WEB-INF/faces-config.xml se definiše Bean i njegovi property-iji.

```
1. <?xml version="1.0"?>
2. <faces-config
xmlns="http://java.sun.com/xml/ns/javaee"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
5. http://java.sun.com/xml/ns/javaee/web-
facesconfig_1_2.xsd"
6. version="1.2">
7. <managed-bean>
8. <managed-bean-name>book</managed-bean-name>
9. <managed-bean-class>com.corejsf.Book</managed-bean-
class>
10. <managed-bean-scope>request</managed-bean-scope>
11.
12. <managed-property>
13. <property-name>titleKey</property-name>
14. <value>aliceInWonderland</value>
15. </managed-property>
16.
17. <managed-property>
18. <property-name>image</property-name>
19. <value>cheshire.jpg</value>
20. </managed-property>
21.
22. <managed-property>
23. <property-name>numChapters</property-name>
24. <property-class>java.lang.Integer</property-class>
25. <value>12</value>
26. </managed-property>
27. </managed-bean>
28. </faces-config>
```

book-viewer/src/java/com/corejsf/Book.java

U navedenoj klasi Book.java definisane su standardne metode za dohvatanje i postavljanje vrednosti odgovarajućih polja.

```
1. package com.corejsf;
2.
3. import java.util.LinkedList;
4. import java.util.List;
5.
6. public class Book {
7. private String titleKey;
8. private String image;
9. private int numChapters;
10. private List<String> chapterKeys = null;
11.
12. // PROPERTY: titleKey
13. public void setTitleKey(String titleKey) {
this.titleKey = titleKey; }
14. public String getTitleKey() { return titleKey; }
15.
16. // PROPERTY: image
17. public void setImage(String image) { this.image =
image; }
18. public String getImage() { return image; }
19.
20. // PROPERTY: numChapters
21. public void setNumChapters(int numChapters) {
this.numChapters = numChapters;}
22. public int getNumChapters() { return numChapters; }
23.
24. // PROPERTY: chapterKeys
25. public List<String> getChapterKeys() {
26. if(chapterKeys == null) {
27. chapterKeys = new LinkedList<String>();
28. for(int i=1; i <= numChapters; ++i)
29. chapterKeys.add("chapter" + i);
30. }
31. return chapterKeys;
32. }
33. }
```

Fajl book-viewer/src/java/com/corejsf/messages.properties sa spoljašnjim resursima

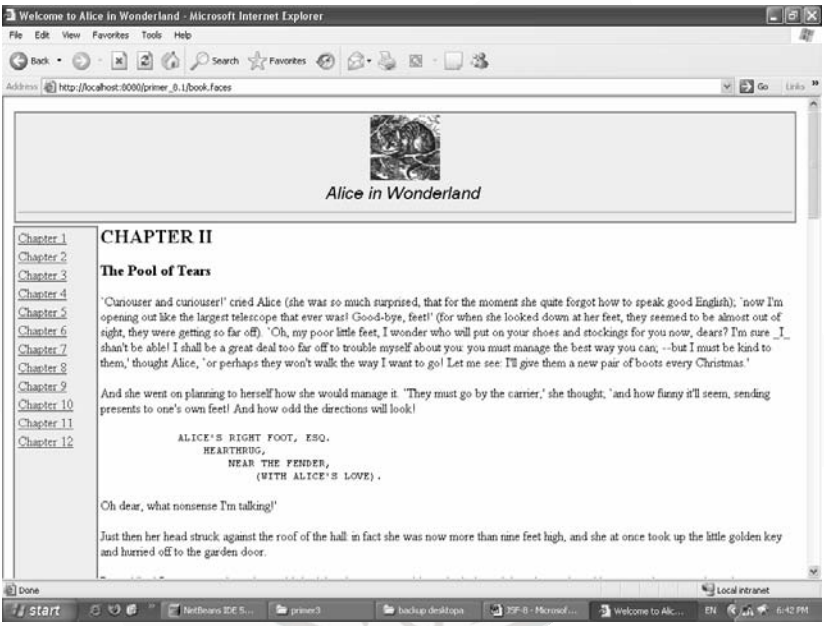
```
1. bookWindowTitle=Welcome to Alice in Wonderland
2. aliceInWonderland=Alice in Wonderland
3.
4. chapter1=Chapter 1
5. chapter2=Chapter 2
6. chapter3=Chapter 3
7. chapter4=Chapter 4
8. chapter5=Chapter 5
9. chapter6=Chapter 6
10. chapter7=Chapter 7
11. chapter8=Chapter 8
12. chapter9=Chapter 9
13. chapter10=Chapter 10
14. chapter11=Chapter 11
15. chapter12=Chapter 12
16. chapter13=Chapter 13
17. chapter14=Chapter 14
18. chapter15=Chapter 15
```

U okviru klijentskih stranica koristi se i CSS book-viewer/web/styles.css

```
1. .bookHeader {
2. width: 100%;
3. text-align: center;
4. background-color: #eee;
5. padding: 0 px;
6. border: thin solid CornflowerBlue;
7. }
8. .bookTitle {
9. text-align: center;
10. font-style: italic;
11. font-size: 1.3em;
12. font-family: Helvetica;
13. }
14. .book {
15. vertical-align: top;
16. width: 100%;
17. height: 100%;
18. }
19. .menuColumn {
20. vertical-align: top;
21. background-color: #eee;
```

```
22. width: 100px;
23. border: thin solid #777;
24. }
25. .chapterColumn {
26. vertical-align: top;
27. text-align: left;
28. width: *;
29. }
```

Dobijeni rezultati su:



Slika 38. Izgled prethodnog primer

Dati problem smo mogli rešiti i na jednostavniji način. Sama JSF stranica reprezentuje dve stvari: layout i sadržaj. Kao što je korišćeno loayout se implementira pomoću taga h:panelGrid, dok je sadržaj reprezentovan različitim JSF tagovima, kao što su h:graphicImage, h:outputText, h:commandLink, i property book chapters. Ono što ovde predstavlja problem je što realizovani layout ne može da se ponovo koristi.
Zato je bolje rešenje da se definiše zajednički sadržaj kako bi mogao da se ponovo koristi u drugim JSF stranicama. U okviru JSP stranica postoje tri načina za uključivanje sadržaja:

- `<%@ include file="header.jsp"% >`
- `<jsp:include page="header.jsp"/>`
- `<c:import url="header.jsp"/>`

Data rešenja su već razmatrana, zato će sada biti navedene samo glavne karakteristike. Prva opcija, JSP include direktiva uključuje specificirani fajl JSF stranice koja se kompajlira na serveru i nakon toga se ta stranica zatvara. Postoje i neka ograničenja za datu direktivu, ako se uključeni sadržaj fajla nakon zatvaranja stranice procesira, te promene neće biti prikazane u osnovnoj stranici. To znači da mora ručno da se menja osnovna stranica - bez obzira da li je sadržaj menjan ili ne.
Druge dve direktive uključuju sadržaj stranice u trenutku izvršavanja i spajaju uključeni sadržaj sa JSF stranicom. Zbog toga što se uključivanje stranica radi u vreme izvršavanja, sve promene će se odmah videti na osnovnoj stranici. Iz navedenih razloga jsp:include i c:import se više koriste od include direktive. Takođe, c:import tag ima slično ponašanje kao i jsp:include, ali ima dosta prednosti i to c:import može da uključi resurse iz drugih fajlova, a jsp:include ne može. U svakom slučaju mora da se uključi JSTL core tag biblioteka kako bi mogao da se koristi c:import.
Bez obzira koja se direktiva koristi, jsp:include ili c:import, mora se uzeti u obzir sledeća razmatranja pri uključenju spolajšenjeg sadržaja u JavaServer Faces aplikaciju:

- Moraju se uključiti JSF tagovi u okviru f:subview taga.
- Uključeni JSF tag ne može da sadrži f:view tagove.

Na primer, book viewer bi trebao da enkapsulira sadržaj hedera u svoju JSF stranicu kako bi mogao ponovo da ga koristi:

```
<%-- this is header.jsp --%>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f"
%>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h"
%>
<h:panelGrid columns="1" styleClass="header">
<h:graphicImage value="books/book/cheshire.jpg"/>
<h:outputText value="#{msgs.bookTitle}"
styleClass="bookTitle"/>
...
</h:panelGrid>
```

Sada može da se uključi sadražaj iz originalne JSF stranice.

```
<%-- This is from the original JSF page --%>

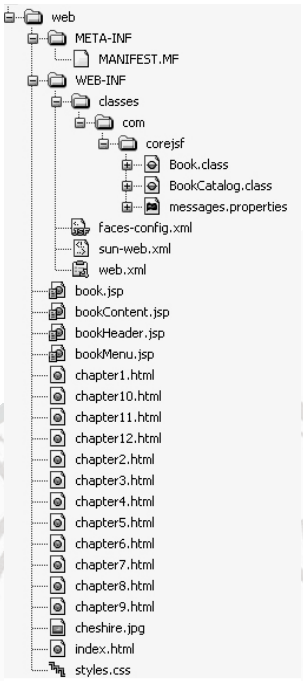
<f:view>
...
<f:subview id="header">
<c:import url="header.jsp"/>
</f:subview>
...
</f:view>
```

Takođe, moraju se dodeliti ID svakom podpogledu. JSF pogledi, koje su obično web stranice, mogu da sadrže veliki broj podpogleda (Subviews). Ali postoji samo jedan pogled koji se koristi u okviru stranice.
Sledeći korak u posmatranom primeru je da se jedinstvena JSF stranicu podeli na četiri fajla: originalnu JSF stranicu, /header.jsp, /menu.jsp, i /content.jsp. Navedeni delovi header, meni, i sadržaj se uključuju u originalnu JSF stranicu:

```
<h:panelGrid columns="2" styleClass="book"
columnClasses="menuColumn, contentColumn">
<f:facet name="header">
<f:subview id="header">
<c:import url="header.jsp"/>
</f:subview>
</f:facet>
<f:subview id="menu">

<c:import url="menu.jsp"/>
</f:subview>
<c:import url="content.jsp"/>
</h:panelGrid>
.....
```

Struktura direktorijuma data na sledećoj slici se koristi u ovom primeru



Slika 39. Izgled direktorijuma koji se koriste u primeru

U okviru stranice book-viewer-include/web/book.jsp
definišu se 2 podpogleda, jedan se odnosi na heder, a drugi na meni.

```
1. <html>
2. <%@ taglib uri="http://java.sun.com/jsp/jstl/core"
prefix="c" %>
3. <%@ taglib uri="http://java.sun.com/jsf/core"
prefix="f" %>
4. <%@ taglib uri="http://java.sun.com/jsf/html"
prefix="h" %>
5.
6. <f:view>
7. <f:loadBundle basename="com.corejsf.messages"
var="msgs"/>
8. <head>
9. <link href="styles.css" rel="stylesheet"
type="text/css"/>
10. <title><h:outputText
value="#{msgs.bookWindowTitle}"/></title>
```

```
11. </head>
12.
13. <body>
14. <h:form>
15. <h:panelGrid columns="2" styleClass="book"
16. columnClasses="menuColumn, chapterColumn">
17. <f:facet name="header">
18. <f:subview id="header">
19. <c:import url="/bookHeader.jsp"/>
20. </f:subview>
21. </f:facet>
22.
23. <f:subview id="menu">
24. <c:import url="/bookMenu.jsp"/>
25. </f:subview>
26.
27. <c:import url="/bookContent.jsp"/>
28. </h:panelGrid>
29. </h:form>
30. </body>
31. </f:view>
32. </html>
```

Zatim se definiše stranica book-viewer-include/web/bookHeader.jsp koja se ubacuje kao heder u okviru stranice book.jsp.

```
1. <%@ taglib uri="http://java.sun.com/jsf/core"
prefix="f" %>
2. <%@ taglib uri="http://java.sun.com/jsf/html"
prefix="h" %>
3.
4. <h:panelGrid columns="1" styleClass="bookHeader">
5. <h:graphicImage value="#{book.image}"/>
6. <h:outputText value="#{msgs[book.titleKey]}"
styleClass="bookTitle"/>
7. <hr>
8. </h:panelGrid>
```

Stranica book-viewer-include/web/bookMenu.jsp predstavlja stranicu koja se ubacuje kao meni u book.jsp.

```
1. <%@ taglib uri="http://java.sun.com/jsf/core"
prefix="f" %>
2. <%@ taglib uri="http://java.sun.com/jsf/html"
prefix="h" %>
3.
4. <h:dataTable value="#{book.chapterKeys}"
var="chapterKey"
5. styleClass="links" columnClasses="linksColumn">
6. <h:column>
7. <h:commandLink>
8. <h:outputText value="#{msgs[chapterKey]}"/>
9. <f:param name="chapter" value="#{chapterKey}"/>
10. </h:commandLink>
11. </h:column>
12. </h:dataTable>
```

Stranica book-viewer-include/web/bookContent.jsp se ubacuje kao sadržaj u book.jsp.

```
1. <%@ taglib uri="http://java.sun.com/jsp/jstl/core"
prefix="c" %>
2.
3. <c:import url="${param.chapter}.html"/
```

Dobijeni rezultat je





Slika 40. Izgled rada aplikacije

5.16 Tiles mogućnosti

Pomoću tiles tehnologije definiše se jedan layout koji se može više puta koristiti u okviru aplikacije. Ovaj paket dozvoljava da se implementira layout kod samo jednom, a da se koristi više puta. Dodatne mogućnosti su i ugnežđeni layout-i, mogućnost proširenja definicije tiles-a, definisanje tiles-a u zavisnosti od uloge korisnika, dodeljivanje kontrolera (Java objekata) tiles-u koji se pozivaju pre nego što se tiles prikažu.
Da bi se koristio ovaj pakete mora se u okviru aplikacije ubaciti njegov JAR file. Potrebno je uraditi sledeće:

- Kopirajte Tiles JAR fajl u aplikacioni WEB-INF/lib direktorijum.
- Dodati Tiles servlet deskriptor (web.xml). Koristiti load-on-startup element kako bi se uverili da je Tiles servlet pokrenut kada je aplikacija startovana.

Deployment deskriptor bi trebalo da izgleda :

```
<?xml version="1.0"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
version="2.4">
...
<servlet>
<servlet-name>Faces Servlet</servlet-name>
<servlet-class>javax.faces.webapp.FacesServlet</servlet-
class>
<load-on-startup>1</load-on-startup>
</servlet>
<servlet>
<servlet-name>Tiles Servlet</servlet-name>
<servlet-
class>org.apache.tiles.servlets.TilesServlet</servlet-
class>
<load-on-startup>2</load-on-startup>
</servlet>
...
</web-app>
```

Tiles se koriste u okviru JSF stranica pomoću sledećih koraka:

1. Koristi se tiles:insert kako bi se ubacila tiles definicija u JSF stranicu.
2. Definiši se tiles u okviru Tiles konfiguracionog fajla.
3. Implementira se tiles layout.

U posmatranom primeru, u okviru stranice book.jsp ubacuje se tiles koji se zove book:

```
...
<%@ taglib uri="http://jakarta.apache.org/tiles"
prefix="tiles" %>
...
<h:form>
<tiles:insert definition="book" flush="false"/>
</h:form>
...
```

Sledeći korak je definicija book tile u /WEB-INF/tiles.xml:

```
<definition name="book"
path="/headerMenuContentLayout.jsp">
<put name="header" value="/bookHeader.jsp"/>
<put name="menu" value="/bookMenu.jsp"/>
<put name="content" value="/bookContent.jsp"/>
</definition>
```

I poziv u okviru stranice

```
<%-- this is /headerMenuContentLayout.jsp --%>
<%@ taglib uri="http://java.sun.com/jsf/html"
prefix="h"%>
<%@ taglib uri="http://java.sun.com/jsf/core"
prefix="f"%>
<%@ taglib uri="http://jakarta.apache.org/tiles"
prefix="tiles" %>

<h:panelGrid columns="2" styleClass="gridClass"
headerClass="headerClass"
columnClasses="menuClass, contentClass">

<f:facet name="header">
<f:subview id="header">
<tiles:insert attribute="header" flush="false"/>
</f:subview>
```

```
</f:facet>
<f:subview id="menu">
<tiles:insert attribute="menu" flush="false"/>
</f:subview>

<f:subview id="content">
<tiles:insert attribute="content" flush="false"/>
</f:subview>
</h:panelGrid>
```

Tag tiles:insert dinamički uključuje sadržaj. Taj sadržaj je vrednost atributa taga iz tiles:insert. Prethodni kod ubacuje heder atribut. Vrednost atributa je /bookHeader.jsp, tako da tiles:insert dinamički uključuje taj fajl.
Treba primetiti da je specificiran atribut flush="false" za tiles:insert tag. To je neophodno za sve moderne servlete, jer servleti mogu da ne dopuste brisanje bafera unutar uobičajenog taga. Ovaj atribut je po definiciji postavljen na početnu vrednost true.
U sledećem primeru koda je prikazano kako se vrši parametrizovanje CSS imena klasa.

```
<definition name="book"
path="/headerMenuContentLayout.jsp">
<put name="headerClass" value="headerClass"/>
<put name="menuClass" value="menuClass"/>
<put name="contentClass" value="contentClass"/>
<put name="header" value="/bookHeader.jsp"/>
<put name="menu" value="/bookMenu.jsp"/>
<put name="content" value="/bookContent.jsp"/>
</definition>
```

Zatim se koriste ti atributi u okviru layout-a:

```
<%-- this is an excerpt of /headerMenuContentLayout.jsp
--%>

...
<tiles:importAttribute scope="request"/>
<h:panelGrid columns="2" styleClass="#{gridClass}"
headerClass="#{headerClass}"
columnClasses="#{menuClass}, #{contentClass}">
...
</h:panelGrid>
```

Tiles atributi kao što su headerClass, menuClass, postoje u tiles opsegu, koji nije bio pristupačan JSF-u. Kako bi postavili da ovi atributi budu pristupačni layout JSF stranice koristi se tiles:importAttribute tag. Taj tag importuje sve atribute onom opsegu koji je specificiran atributom scope.

```
<definition name="anotherTile"
path="/headerMenuContentLayout.jsp">
<put name="headerClass" value="aDifferentHeaderClass"/>
...
</definition>
```

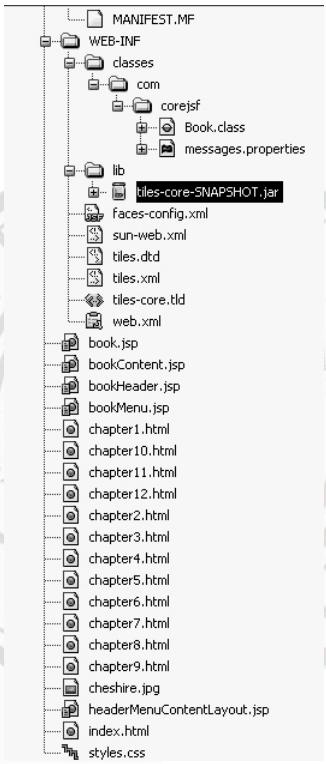
U parametrizaciji Tiles-a, definisan je sledeći izgled:

```
<definition name="book"
path="/headerMenuContentLayout.jsp">
<put name="headerClass" value="headerClass"/>
<put name="menuClass" value="menuClass"/>
<put name="contentClass" value="contentClass"/>
<put name="header" value="/bookHeader.jsp"/>
<put name="menu" value="/bookMenu.jsp"/>
<put name="content" value="/bookContent.jsp"/>
</definition>
```

Ovde postoje dva različita tipa atributa u okviru tiles-a CSS klase i uključeni sadržaj.

```
<definition name="header-menu-content"
path="/headerMenuContentLayout.jsp">
<put name="headerClass" value="headerClass"/>
<put name="menuClass" value="menuClass"/>
<put name="contentClass" value="contentClass"/>
</definition
<definition name="book" extends="header-menu-content">
<put name="header" value="/bookHeader.jsp"/>
<put name="menu" value="/bookMenu.jsp"/>
<put name="content" value="/bookContent.jsp"/>
</definition>
```

Sada sam book tiles proširuje header-menu-content tile. Kada se izvršava proširivanje tiles-a, nasleđuje se njegov layout i atributi, slično kao što u Javi podklase nasleđuju metode i promenljive iz njihove osnovne klase
Kako je originalni tiles podeljen na dva dela, atributi CSS klase su slobodni za ponovno korišćenje kod drugih tiles koji proširuju header-menu-content tile.
Struktura direktorijuma data na sledećoj slici se koristi u ovom primeru.



Slika 41. Struktura direktorijuma koja se koristi u primeru

U okviru stranice

book-viewer-tiles/web/WEB-INF/tiles.xml

definišu se tiles koji se koristi, njegovo ime i putanja, kao i drugi elementi koji se ubacuju na odgovarajući način.

```
1. <!DOCTYPE tiles-definitions PUBLIC
2. "-//Apache Software Foundation//DTD Tiles
Configuration//EN"
3. "http://jakarta.apache.org/struts/dtds/tiles-
config.dtd">
4.
5. <tiles-definitions>
6. <definition name="book"
path="/headerMenuContentLayout.jsp">
7. <put name="gridClass" value="headerMenuContent"/>
8. <put name="headerClass" value="header"/>
9. <put name="menuColumnClass" value="menuColumn"/>
10. <put name="contentColumnClass"
value="contentColumn"/>
11.
12. <put name="header" value="/bookHeader.jsp"/>
13. <put name="menu" value="/bookMenu.jsp"/>
14. <put name="content" value="/bookContent.jsp"/>
15. </definition>
16. </tiles-definitions>
```

U stranici book-viewer-tiles/web/headerMenuContentLayout.jsp definiše se tabela ,u okviru nje delovi tiles-a za heder, meni i sadržaj.

```
1. <%@ taglib uri="http://java.sun.com/jsf/core"
prefix="f" %>
2. <%@ taglib uri="http://java.sun.com/jsf/html"
prefix="h" %>
3. <%@ taglib uri="http://jakarta.apache.org/tiles"
prefix="tiles" %>
4.
5. <tiles:importAttribute scope="request"/>
6.
7. <h:panelGrid columns="2" styleClass="#{gridClass}"
8. headerClass="#{headerClass}"
9. columnClasses="#{menuColumnClass},
#{contentColumnClass}">
10. <f:facet name="header">
11. <f:subview id="header">
12. <tiles:insert attribute="header" flush="false"/>
```

```
13. </f:subview>
14. </f:facet>
15.
16. <f:subview id="menu">
17. <tiles:insert attribute="menu" flush="false"/>
18. </f:subview>
19.
20. <f:subview id="content">
21. <tiles:insert attribute="content" flush="false"/>
22. </f:subview>
23. </h:panelGrid>
```

Početna stranica je book-viewer-tiles/web/book.jsp koja koristi sam tiles.

```
1. <html>
2. <%@ taglib uri="http://java.sun.com/jsp/jstl/core"
prefix="c" %>
3. <%@ taglib uri="http://java.sun.com/jsf/core"
prefix="f" %>
4. <%@ taglib uri="http://java.sun.com/jsf/html"
prefix="h" %>
5. <%@ taglib uri="http://jakarta.apache.org/tiles"
prefix="tiles" %>
6.
7. <f:view>
8. <f:loadBundle basename="com.corejsf.messages"
var="msgs"/>
9. <head>
10. <link href="styles.css" rel="stylesheet"
type="text/css"/>
11. <title><h:outputText
value="#{msgs.bookWindowTitle}"/></title>
12. </head>
13.
14. <body>
15. <f:subview id="book">
16. <h:form>
17. <tiles:insert definition="book" flush="false"/>
18. </h:form>
19. </f:subview>
20. </body>
21. </f:view>
22. </html>
```

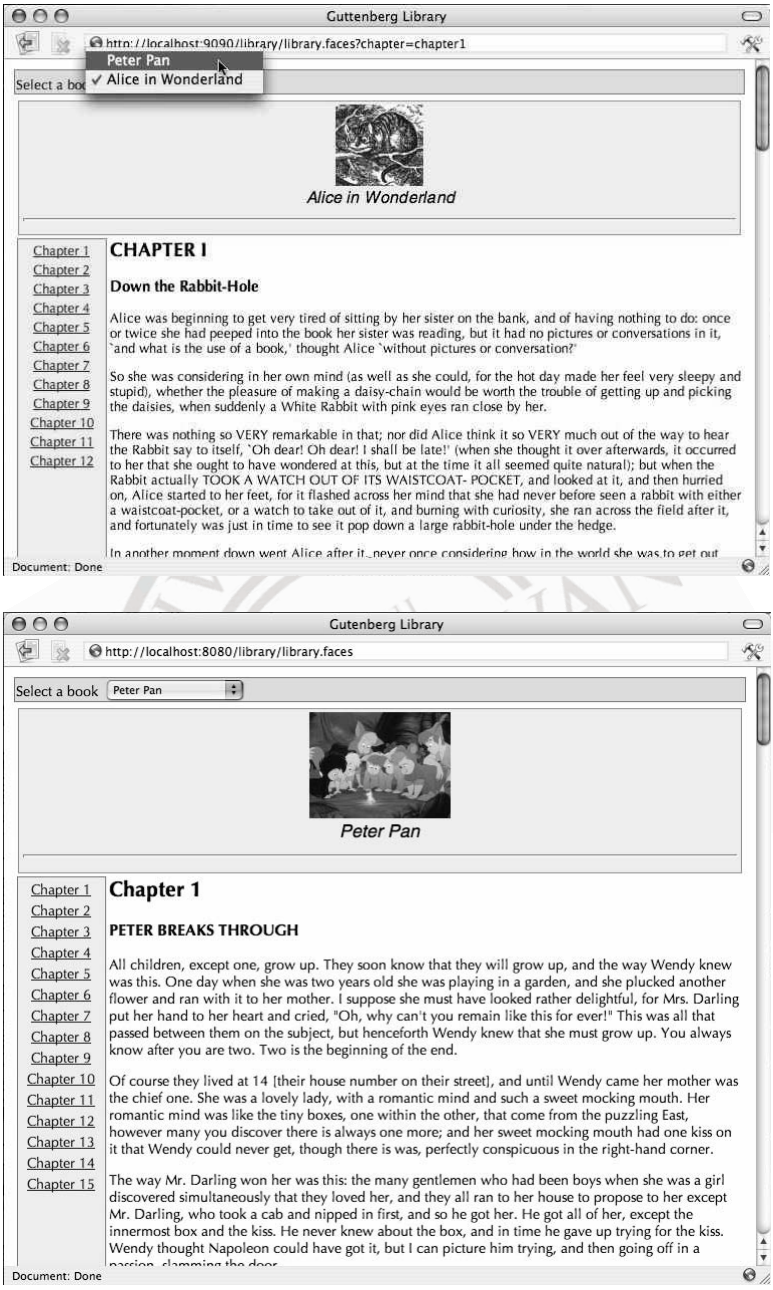
U sledećem primeru će se razmotriti dve posebne tiles tehnike: ugnježđeni tiles i korišćenje kontrolera.

```
<definition name="book">
...
</definition>
<definition name="library" path="/libraryLayout.jsp"
controllerClass="com.corejsf.LibraryTileController">

<put name="header" value="/bookSelector.jsp"/>
<put name="book" value="book"/>
</definition>
```

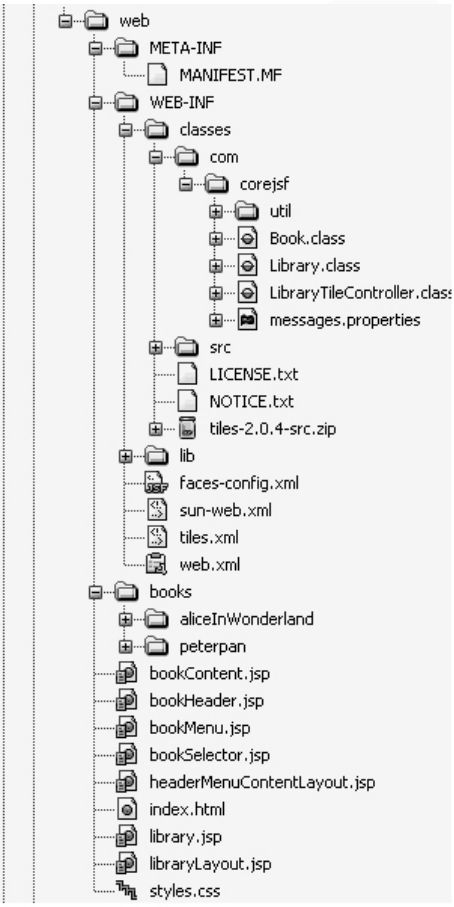
Treba primetiti da je vrednost za book atribut tile, a ne JSP stranica. Korišćenje imena tile umesto JSP stranice dozvoljava da se ugnježde tiles. Tiles dozvoljava da se veže poseban Java objekat, koji se zove kontroler. Objekat klase mora da implementira interfejs org.apache.struts.tiles. Kontroler definiše jednistvenu metodu. Tiles poziva tu metodu pre nego što se učita kontroler pridružen tiles. Tiles kontroleri prisupaji svom tile konektstu i atributima. Takođe tiles kontroleri mogu da kreiraju nove atribute.





Slika 42. Rezultat narednog primera

U prethodnom primeru pridužen je kontroler library tile. Kontroler posmatra library atribut u opsegu sesije. Ako ga tamo ne pronađe, kontroler kreira library i stavlja ga u opseg sesije. Potom kontroler konsultuje property selectedBook u okviru library kako bi video da li je knjiga selektovna. Ako jeste onda kontroler postavlja vrednost atributa knjige, koji je u opsegu sesije, na selektovanu knjigu. U suptrotnom, postaviće na vrednost Peter Pan.
Struktura direktorijuma data na sledećoj slici se koristi u ovom primeru.



Slika 43. Struktura direktorijuma primera

Početna stranica library/web/library.jsp koja koristi sam tiles.

```
1. <html>
2. <%@ taglib uri="http://java.sun.com/jsf/core"
prefix="f" %>
3. <%@ taglib uri="http://java.sun.com/jsf/html"
prefix="h" %>
4. <%@ taglib uri="http://jakarta.apache.org/tiles"
prefix="tiles" %>
5.
6. <f:view>
7. <f:loadBundle basename="com.corejsf.messages"
var="msgs"/>
8. <head>
9. <link href="styles.css" rel="stylesheet"
type="text/css"/>
10. <title><h:outputText
value="#{msgs.libraryWindowTitle}"/></title>
11. </head>
12.
13. <body>
14. <f:subview id="library">
15. <h:form>
16. <tiles:insert definition="library" flush="false"/>
17. </h:form>
18. </f:subview>
19. </body>
20. </f:view>
21. </html>
```

U okviru strane library/web/WEB-INF/tiles.xml definiše se tiles koji se koristi, njegovo ime i putanja do njega.U datoj realizaciji postoje 3 tipa tiles atributa.

```
1. <?xml version="1.0" encoding="ISO-8859-1" ?>
2.
3. <!DOCTYPE tiles-definitions PUBLIC
4. "-//Apache Software Foundation//DTD Tiles
Configuration 1.1//EN"
5. "http://struts.apache.org/dtds/tiles-config_1_1.dtd">
6.
7. <tiles-definitions>
8. <definition name="menu-header-content"
path="/headerMenuContentLayout.jsp">
9. <put name="gridClass" value="headerMenuContent"/>
10. <put name="headerClass" value="header"/>
```

```
11. <put name="menuColumnClass" value="menuColumn"/>
12. <put name="contentColumnClass"
value="contentColumn"/>
13. </definition>
14.
15. <definition name="book" extends="menu-header-
content">
16. <put name="header" value="/bookHeader.jsp"/>
17. <put name="menu" value="/bookMenu.jsp"/>
18. <put name="content" value="/bookContent.jsp"/>
19. </definition>
20.
21. <definition name="library" path="/libraryLayout.jsp"
22. controllerClass="com.corejsf.LibraryTileController">
23. <put name="header" value="/bookSelector.jsp"/>
24. <put name="book" value="book"/>
25. </definition>
26. </tiles-definitions>
```

U okviru stranice library/web/libraryLayout.jsp koristi se definisani tiles.

```
1. <%@ taglib uri="http://java.sun.com/jsf/core"
prefix="f" %>
2. <%@ taglib uri="http://java.sun.com/jsf/html"
prefix="h" %>
3. <%@ taglib uri="http://jakarta.apache.org/tiles"
prefix="tiles" %>
4.
5. <h:panelGrid columns="1" styleClass="book"
headerClass="libraryHeader">
6. <f:facet name="header">
7. <f:subview id="header">
8. <tiles:insert attribute="header" flush="false"/>
9. </f:subview>
10. </f:facet>
11.
12. <f:subview id="book">
13. <tiles:insert attribute="book" flush="false"/>
14. </f:subview>
15. </h:panelGrid>
```

U stranici library/web/bookSelector.jsp se definiše osluškivač događaja koji reaguje na promenu vrednosti atributa kada se selektuje odgovarajuća stavka menija.

```
1. <%@ taglib uri="http://java.sun.com/jsf/core"
prefix="f" %>
2. <%@ taglib uri="http://java.sun.com/jsf/html"
prefix="h" %>
3.
4. <h:outputText value="#{msgs.selectABookPrompt}"/>
5.
6.   
7.
8. <h:selectOneMenu onchange="submit()"
value="#{library.book}"
9. valueChangeListener="#{library.bookSelected}">
10. <f:selectItems value="#{library.bookItems}"/>
11. </h:selectOneMenu>
```

I u stranici library/web/bookSelector.jsp definiše se osluškivač događaja koji reaguje na promenu vrednosti atributa kada se selektuje odgovarajuća stavka menija.

```
1. <%@ taglib uri="http://java.sun.com/jsf/core"
prefix="f" %>
2. <%@ taglib uri="http://java.sun.com/jsf/html"
prefix="h" %>
3.
4. <h:outputText value="#{msgs.selectABookPrompt}"/>
5.
6.   
7.
8. <h:selectOneMenu onchange="submit()"
value="#{library.book}"
9. valueChangeListener="#{library.bookSelected}">
10. <f:selectItems value="#{library.bookItems}"/>
11. </h:selectOneMenu>
```

Servereska komponenta je library/src/java/com/corejsf/Library.java

```
1. package com.corejsf;
2.
3. import java.util.*;
4. import javax.faces.model.SelectItem;
```

```
5. import javax.faces.event.ValueChangeEvent;
6.
7. public class Library {
8. private Map<String,Book> bookMap = new
HashMap<String,Book>();
9. private Book initialBook = null;
10. private List bookItems = null;
11. private String book = null;
12. private String selectedBook = null;
13.
14. public Library() {
15. Book peterpan = new Book();
16. Book aliceInWonderland = new Book();
17.
18. initialBook = peterpan;
19.
20.
aliceInWonderland.setDirectory("books/aliceInWonderland"
);
21. aliceInWonderland.setTitleKey("aliceInWonderland");
22.
aliceInWonderland.setImage("books/aliceInWonderland/ches
hire.jpg");
23. aliceInWonderland.setNumChapters(12);
24.
25. peterpan.setDirectory("books/peterpan");
26. peterpan.setTitleKey("peterpan");
27. peterpan.setImage("books/peterpan/peterpan.jpg");
28. peterpan.setNumChapters(15);
29.
30. bookMap.put("aliceInWonderland", aliceInWonderland);
31. bookMap.put("peterpan", peterpan);
32. }
33. public void setBook(String book) { this.book = book;
}
34. public String getBook() { return book; }
35.
36. public Map<String,Book> getBooks() {
37. return bookMap;
38. }
39. public void bookSelected(ValueChangeEvent e) {
40. selectedBook = (String) e.getNewValue();
41. }
42. public Book getSelectedBook() {
43. return selectedBook != null ?
bookMap.get(selectedBook) : initialBook;
44. }
```

```
45. public List getBookItems() {
46. if(bookItems == null) {
47. bookItems = new LinkedList();
48. Iterator<Book> it = bookMap.values().iterator();
49. while(it.hasNext()) {
50. Book book = it.next();
51. bookItems.add(new SelectItem(book.getTitleKey(),
52. getBookTitle(book.getTitleKey())));
53. }
54. }
55. return bookItems;
56. }
57. private String getBookTitle(String key) {
58. return com.corejsf.util.Messages.
59. getString("com.corejsf.messages", key, null);
60. }
61. }
```

I klasa koja predstavlja kontroler

library/src/java/com/corejsf/LibraryTileController.java

```
1. package com.corejsf;
2.
3. import java.io.IOException;
4. import javax.servlet.ServletContext;
5. import javax.servlet.ServletException;
6. import javax.servlet.http.HttpServletRequest;
7. import javax.servlet.http.HttpServletResponse;
8. import javax.servlet.http.HttpSession;
9. import org.apache.tiles.ComponentContext;
10. import org.apache.tiles.Controller;
11.
12. public class LibraryTileController implements
Controller {
13. public void execute(ComponentContext tilesContext,
14. HttpServletRequest request,
15. HttpServletResponse response,
16. ServletContext context)
17. throws IOException, ServletException {
18. HttpSession session = request.getSession();
19.
20. String chapter = (String)
request.getParameter("chapter");
21. session.setAttribute("chapter", chapter == null ||
"".equals(chapter) ?
22. "chapter1" : chapter);
```

```
23.
24. Library library = (Library)
session.getAttribute("library");
25.
26. if(library == null) {
27. library = new Library();
28. session.setAttribute("library", library);
29. }
30.
31. Book selectedBook = library.getSelectedBook();
32. if(selectedBook != null) {
33. session.setAttribute("book", selectedBook);
34. }
35. }
36. public void perform(ComponentContext tilesContext,
37. HttpServletRequest request,
38. HttpServletResponse response,
39. ServletContext context)
40. throws IOException, ServletException {
41. HttpSession session = request.getSession();
42. execute(tilesContext, request, response, context);
43. }
44. }
```

U serverskom radu koristi se i Bean Book klase
library/src/java/com/corejsf/Book.java.
U okviru ove klase postoje metode za dohvatanje i postavljanje vrednosti property-ija.

```
1. package com.corejsf;
2.
3. import java.util.LinkedList;
4. import java.util.List;
5.
6. public class Book {
7. private String titleKey;
8. private String image;
9. private String directory;
10. private int numChapters;
11. private List<String> chapterKeys = null;
12.
13. // PROPERTY: titleKey
14. public void setTitleKey(String titleKey) {
this.titleKey = titleKey; }
15. public String getTitleKey() { return titleKey; }
16.
```

```
17. // PROPERTY: image
18. public void setImage(String image) { this.image =
image; }
19. public String getImage() { return image; }
20.
21. // PROPERTY: numChapters
22. public void setNumChapters(int numChapters) {
this.numChapters = numChapters;}
23. public int getNumChapters() { return numChapters; }
24.
25. // PROPERTY: directory
26. public void setDirectory(String directory) {
this.directory = directory;}
27. public String getDirectory() { return directory; }
28.
29. // PROPERTY: chapterKeys
30. public List<String> getChapterKeys() {
31. if(chapterKeys == null) {
32. chapterKeys = new LinkedList<String>();
33. for(int i=1; i <= numChapters; ++i)
34. chapterKeys.add("chapter" + i);
35. }
36. return chapterKeys;
37. }
38. }
```

Postoji više stranica koje se koriste i pozivaju iz tiles-a. Jedna je library/web/bookHeader.jsp koja definiše heder stranica

```
1. <%@ taglib uri="http://java.sun.com/jsf/core"
prefix="f" %>
2. <%@ taglib uri="http://java.sun.com/jsf/html"
prefix="h" %>
3.
4. <h:panelGrid columns="1" styleClass="bookHeader">
5. <h:graphicImage value="#{book.image}"/>
6. <h:outputText value="#{msgs[book.titleKey]}"
styleClass="bookTitle"/>
7. <hr>
8. </h:panelGrid>
```

U okviru stranice library/web/bookMenu.jsp definisan je meni

```
1. <%@ taglib uri="http://java.sun.com/jsf/core"
prefix="f" %>
2. <%@ taglib uri="http://java.sun.com/jsf/html"
prefix="h" %>
3.
4. <h:dataTable value="#{book.chapterKeys}"
var="chapterKey"
5. styleClass="links" columnClasses="linksColumn">
6. <h:column>
7. <h:commandLink>
8. <h:outputText value="#{msgs[chapterKey]}"/>
9. <f:param name="chapter" value="#{chapterKey}"/>
10. </h:commandLink>
11. </h:column>
12. </h:dataTable>
```

U okviru stranice library/web/bookContent.jsp definisan je sadržaj

```
1. <%@ taglib uri="http://java.sun.com/jsp/jstl/core"
prefix="c" %>
2.
3. <c:import url="${book.directory}/${chapter}.html"/>
```

U fajlu library/web/styles.css se definiše se CSS.

```
1. .library {
2. vertical-align: top;
3. width: 100%;
4. height: 100%;
5. }
6. .libraryHeader {
7. width: 100%;
8. text-align: left;
9. vertical-align: top;
10. background-color: #ddd;
11. font-weight: lighter;
12. border: thin solid #777;
13. }
14. .bookHeader {
15. width: 100%;
16. text-align: center;
17. background-color: #eee;
18. border: thin solid CornflowerBlue;
```

```
19. }
20. .bookTitle {
21. text-align: center;
22. font-style: italic;
23. font-size: 1.3em;
24. font-family: Helvetica;
25. }
26. .menuColumn {
27. vertical-align: top;
28. background-color: #eee;
29. border: thin solid #777;
30. }
31. .chapterColumn {
32. vertical-align: top;
33. text-align: left;
34. width: *;
35. padding: 3px;
36. }
37. .contentColumn {
38. vertical-align: top;
39. text-align: left;
40. width: *;
41. }
42. .links {
43. width: 85px;
44. vertical-align: top;
45. text-align: center;
46. }
47. .linksColumn {
48. vertical-align: top;
49. text-align: center;
50. }
```

Knfiguracioni fajl library/web/WEB-INF/faces-config.xml je.

```
51. <?xml version="1.0"?>
52. <faces-config
xmlns="http://java.sun.com/xml/ns/javaee"
53. xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
54.
xsi:schemaLocation="”http://java.sun.com/xml/ns/javaee
55. http://java.sun.com/xml/ns/javaee/web-
facesconfig_1_2.xsd"
56. version="1.2">
57. </faces-config>
```

U okviru fajla library/web/WEB-INF/web.xml definiše se Tiles Servlet i putanja do tog servleta, kao i putanja do definicija tilesa. Zatim se definiše Faces Servlet i njegova putanja. Na kraju se definiše koja stranica je početna stranica aplikacije.

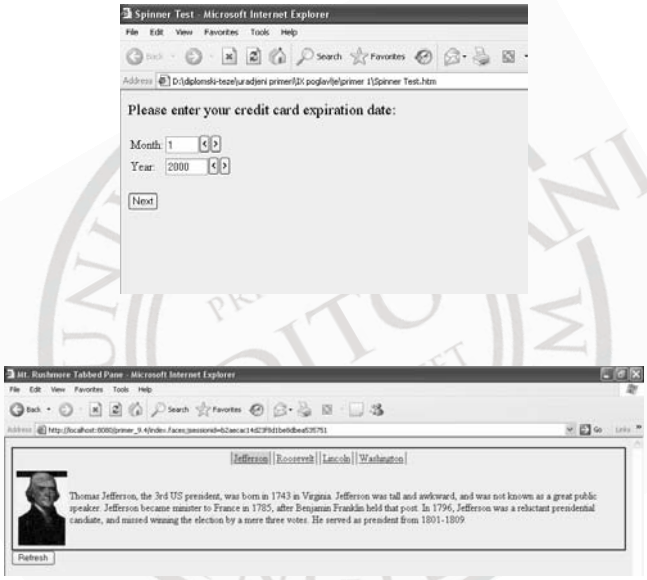
```
1. <?xml version="1.0"?>
2. <web-app xmlns="http://java.sun.com/xml/ns/javaee"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
5. http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
6. version="2.5">
7. <servlet>
8. <servlet-name>Tiles Servlet</servlet-name>
9. <servlet-
class>org.apache.tiles.servlets.TilesServlet</servlet-
class>
10. <init-param>
11. <param-name>definitions-config</param-name>
12. <param-value>/WEB-INF/tiles.xml</param-value>
13. </init-param>
14. <load-on-startup>2</load-on-startup>
15. </servlet>
16.
17. <servlet>
18. <servlet-name>Faces Servlet</servlet-name>
19. <servlet-
class>javax.faces.webapp.FacesServlet</servlet-class>
20. <load-on-startup>1</load-on-startup>
21. </servlet>
22.
23. <servlet-mapping>
24. <servlet-name>Faces Servlet</servlet-name>
25. <url-pattern>*.faces</url-pattern>
26. </servlet-mapping>
27.
28. <welcome-file-list>
29. <welcome-file>index.html</welcome-file>
30. </welcome-file-list>
31. </web-app>
```

I na kraju fajl sa spoljašnjim resursima
library/src/java/com/corejsf/messages.properties

```
1. libraryWindowTitle=Gutenberg Library
2. aliceInWonderland=Alice in Wonderland
3. peterpan=Peter Pan
4. selectABookPrompt=Select a book
5.
6. chapter1=Chapter 1
7. chapter2=Chapter 2
8. chapter3=Chapter 3
9. chapter4=Chapter 4
10. chapter5=Chapter 5
11. chapter6=Chapter 6
12. chapter7=Chapter 7
13. chapter8=Chapter 8
14. chapter9=Chapter 9
15. chapter10=Chapter 10
16. chapter11=Chapter 11
17. chapter12=Chapter 12
18. chapter13=Chapter 13
19. chapter14=Chapter 14
20. chapter15=Chapter 15
```

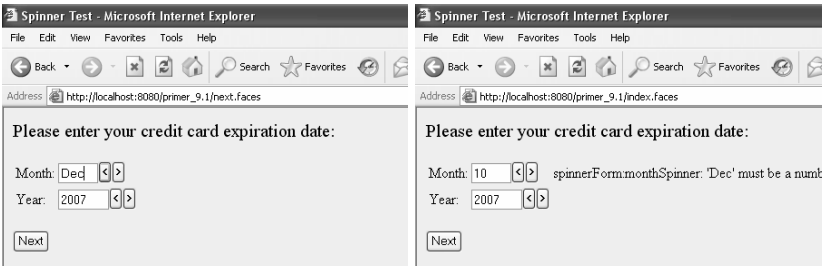
5.17 Napredne komponente i validacija

Ponekad je potrebno da se pored standardnih HTML komponenti, kao što su: tekst polja, polja za potvrdu, dugme, ... da se koriste i naprednije komponente, kao što su: kalendar, površine sa tabovima, navigaciona stabal, ... Na žalost ove komponente nisu deo standarnih JSF komponenti.
U ovom poglavlju će biti reči o tome kako se implementiraju napredne komponente. Posebno će se obraditi dve komponente: spinner i the tabbed pane.



Slika 44. Izgled naprednih komponenti spinner i the tabbed pane

Spinner komponenta dozvoljava da se unese broj u tekst polje, bilo da se unosi direktno u polje ili aktiviranjem dugmeta za inkrementiranje ili dekrementiranje. Na prethodnoj slici je dat prikaz ekrana na kome su navedena 2 objekta za unos datuma kada ističe kreditna kartica, jedan prihvata podataka za mesec, a drugi za godinu.
Ova komponenta zahteva celobrojne vrednosti. U slučaju da se unese netačan podatak standardni konvertor celobronih vrednosti će obraditi nastalu grešku.



Slika 45. Greška prilikom upotrebe spinner komponente

Sledeći deo koda ilustruje kako se koristi ova komponenta:

```
<%@ taglib uri="http://corejsf.com/spinner"
prefix="corejsf" %>
...
<corejsf:spinner value="#{cardExpirationDate.month}"
id="monthSpinner" minimum="1" maximum="12" size="3"/>
<h:message for="monthSpinner"/>
...
<corejsf:spinner value="#{cardExpirationDate.year}"
id="yearSpinner" minimum="1900" maximum="2100"
size="5"/>
<h:message for="yearSpinner"/>
```

Iz navedenog koda može se primetiti da se koristi tag corejsf:spinner. Ovaj tag podržava sledeće atribute:

- binding
- id
- minimum
- maximum
- rendered
- size
- value

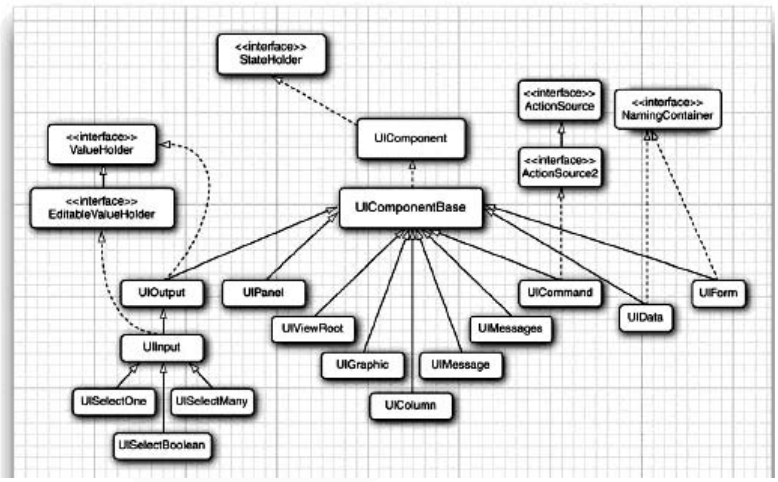
Potrebno je definisati bar jedan od navedenih atributa. Atributi minimum i maximum označavaju opseg validnih vrednosti. Atribut value prihvata string literal. Tag klasa za ove atribute nasleđuje klasu

javax.faces.webapp.UIComponentELTag.

Prilikom konverzije gređke će se prikazivati pomoću h:message.
Kod navednih JSF naprednih komponenti koriste se dve klase:

- Klasa koja procesira tag atribute. Po konvenciji, klasa ima Tag sufiks.
- Klasa komponente održava stanje, korisnički interfejs i procesira ulaz. Po konvenciji, klasa ima UI prefiks.

UI klasa radi veoma važan posao.Ona ima dve odvojene odgovornosti: da izražava korisnički interfejs tako što kodira oznake i da procesira korisnčki ulaz tako što koduje tekući HTTP zahtev.
Klase komponenti mogu da deligiraju modeliranje i procesiranje ulaza na dva odvojena modula. Inicijalno spiner komponenta će se sama modelirati. Komponenta UI klasa mora da proširi osnovnu klasu UIComponent. Ova klasa definiše preko 40 apstraktinh metoda, tako da ako se želi da se proširi postojeća klasa moguće je implemtirati ove metode.



Slika 46. Hijerarhija klasa kod naprednih komponenti

Navedena UISpinner klasa će proširiti UIInput, koja proširuje UIOutput i implementira interfejs EditableValueHolder. S druge strane razvijena UITabbedPane klasa će proširiti UICommand, koja implementira interfejs ActionSource2 .

Prilikom implementiranja naprednih komponenti sledeće JSF klase se koriste:

- javax.faces.component.UIComponent
- javax.faces.webapp.UIComponentELTag
- javax.faces.context.FacesContext
- javax.faces.application.Application
- javax.faces.context.ResponseWriter

UIComponent je osnovna apstraktna klasa koja definiše šta znači biti komponenta. Svaka komponenta upravlja sa nekoliko važnih kategorija podataka, kao što su lista dece komponenti, mapa facet komponenti (facets su slični deci komponent, ali svaki facet ima ključ, a ne poziciju u listi), mapu atributa (ova mapa je opšte namene koja može da se koristi za stavljanje ključa /vrednosti), mapa vrednosnih izraza (ovo je još jedna mapa opšte namene koja može da služi za stavljanje vrednosti izraza), kolekciju osluškivača (ovu kolekciju podržava JSF framework).

Kada se definiše JSF komponenta, uobičajeno je da se koriste sledeće klase, od kojih se prave podklase:

- UICommand, ako komponenta proizvodi akcija slične komandama, linku ili dugmetu.
- UIOutput, ako komponenta prikazue vrednost koji ne može da menja korisnik.
- UIInput, ako komponenta čita vrednost od korisnika, na primer od spinner-a.

Na slici koja predstavlja hijererhiju klasa koje se koriste u okviru naprednih komponenti može se primetiti da:

- ActionSource intefejs definiše metode za upravljanje osluškivačima događaja.
- ActionSource2 interfejs definiše metode za upravljanjem akcijama.
- ValueHolder interfejs definiše metode za upravljanje vrednosnim komponentama, lokalnim vrednostima i konvertorima.
- EditableValueHolder interfejs proširuje ValueHolder i dodaje metode za upravljanjem validatora i osluškivača događaja koji reaguju na promenu vrednosti.

FacesContext klasa sadrži JSF-zahtevane informacije. Između ostalog moguće je pristupiti parametrima kroz FacesContext, da bi se prihvatila referenca na objekat Application, da se prihvati tekuća komponenta view root-a , ili da se dohvati refernca na odgovarajući response writer, koji služi za kodiranje oznaka.
Application klasa čuva osnovne objekata koji se čuvaju pri svakoj aplikaciji. Ona, takođe, služi kao fabrika, sa fabričkim metodama za komponente, konvertore i validatore. Response writers klasa ima metode za startovanje i zaustavljanje HTML elemenata, i metode za pisanje atributa elemenata.
JSF komponente generišu oznake za svoj korisnički interfejs. Po definiciji, standardne JSF komponente generišu HTML. Komponente mogu da vrše sopstveno kodiranje, ili da delegiraju kodiranje na odvojene module.
Komponente koje kodiraju oznake imaju sledeće metode:

- encodeBegin()
- encodeChildren()
- encodeEnd()

JSF budi encodeChildren samo ako komponenta generiše true iz svoje getRendersChildren metode. Po definiciji, getRendersChildren metod vraća false za većinu komponenti.
Za proste komponente, kao što je spinner, koji nema objekte potomke, nije potrebno da se implementira metoda encodeChildren.
Spiner generiše HTML za tekst polje i dva dugmeta, i dobijeni HTML kod izgleda kao:

```
<input type="text" name="..." value="current value"/>
<input type="submit" name="..." value="<"/>
<input type="submit" name="..." value=">"/>
```

Sledeći kod prikazuje kako je HTML kodiran u UISpinner-u:

```
public class UISpinner extends UIInput {
private static final String MORE = ".more";
private static final String LESS = ".less";
...
public void encodeBegin(FacesContext context) throws
IOException {
ResponseWriter writer = context.getResponseWriter();
String clientId = getClientId(context);
encodeInputField(writer, clientId);
encodeDecrementButton(writer, clientId);
```

```
encodeIncrementButton(writer, clientId);
}

private void encodeInputField(ResponseWriter writer,
String clientId)
throws IOException {
writer.startElement("input", this);
writer.writeAttribute("name", clientId, "clientId");
Object v = getValue();
if (v != null)
writer.writeAttribute("value", v.toString(), "value");
Integer size = (Integer) getAttributes().get("size");
if (size != null) writer.writeAttribute("size", size,
"size");
writer.endElement("input");
}

private void encodeDecrementButton(ResponseWriter
writer, String clientId)
throws IOException {
writer.startElement("input", this);
writer.writeAttribute("type", "submit", null);
writer.writeAttribute("name", clientId + LESS, null);
writer.writeAttribute("value", "<", "value");
writer.endElement("input");
}

private void encodeIncrementButton(ResponseWriter
writer, String clientId)
throws IOException {
writer.startElement("input", this);
writer.writeAttribute("type", "submit", null);
writer.writeAttribute("name", clientId + MORE, null);
writer.writeAttribute("value", ">", "value");
writer.endElement("input");
}
...
}
```

ResponseWriter klasa ima metode za pisanje identifikatora.Metode startElement i endElement proizvode granične elemente. Metoda writeAttribute ispisuje par atributa name/value sa odgovarajućim escape karakterima.Poslednji parametar u metodama startElement i writeAttribute je namenjen za podršku alatki, ali on se trenutno ne koristi.

UISpinner.encodeBegin faces mora da uzime tekuće stanje objekta spinner-a. Numeričku vrednost jednostavno dobija i to pomoću metode getValue koju spinner nasleđuje od klase UIInput.
Pored toga, UISpinner.encodeBegin faces mora da pozove metodu getClientId kako bi saznao klijentski ID komponente.
Inkrement i dekrement dugmad počinju sa klijentskim ID i završavaju se sa .more i .less, respektivno.
Ovde je primer HTML koji je generisao spiner:

```
<input type="text" name="_id1:monthSpinner" value="1"
size="3"/>
<input type="submit" name="_id1:monthSpinner.less"
value="<"/>
<input type="submit" name="_id1:monthSpinner.more"
value=">"/>
```

Spinner metode dekodiranja za javax.faces.component.UIComponent su

```
void encodeBegin(FacesContext context) throws
IOException
```

Ova metoda se poziva u Render Response fazi JSF životnog ciklusa, samo ako je komponentni renderer tip jednak null vrednosti.

```
String getClientId(FacesContext context)
```

Vraća ID komponente.JSF framework, kreira klijentski ID iz ID-a kodirajuće forme i iz ID ove komponente.

```
Map getAttributes()
```

Vraća mapu atributa komponente i property-ije. Ova metoda se koristi da bi se videli, dodali, promenili i uklonili atributi iz komponente.
get i put metode mape proveravaju da li je ključ uparen sa property-ijem komponente. Ako jeste onda se pozivaju getter ili setter metode odgovarajućeg property-ija.

Za klasu javax.faces.context.FacesContext metode dekodiranja su

```
ResponseWriter getResponseWriter()
```

Vraća referencu na response writer. JSF koristi response writer kako bi generisao HTML tagove.

Za klasu javax.faces.context.ResponseWritersu

```
void startElement(String elementName, UIComponent
component)
```

Ispisuje početni tag za specifični element.

```
void endElement(String elementName)
```

Ispisuje završni tag za specifični element.

```
void writeAttribute(String attributeName, String
attributeValue, String componentProperty)
```

Ispisuje atribut i njegovu vrednost. Ova metoda može da bude pozvana samo između poziva metoda startElement() i endElement(). componentProperty je ime property-ija komponente koja odgovara atributu. Sam proces dekodiranja obuhvata procesiranje zahtevanih vrednosti. Server šalje HTML formu browser-u, dok browser šalje nazad POST zahtev, koji sadrži par name/value. POST zahtev je jedini podatak koji server može da koristi za interpretiranje korisnikovih akcija unutar browser-a.
Ako korisnik klikne dugme inkrement ili dekrement, POST zahtev uključuje imena i vrednosti svih tekst polja, ali samo ime i vrednost kliknutog dugmeta. Na primer, ako korisnik izabere spinner inkrement dugme u aplikaciji prikazanoj na slici iz ovog poglavlja, sledeći zahtevani parametri su prebačeni sa browser-a na server:

Ime	Vrednost
_id1:monthSpinner	1
_id1:yearSpinner	12
_id1:monthSpinner.more	>

Sledeća metoda spinner-a za dekodiranje je data u sledećem kodu:

```
public void decode(FacesContext context) {
Map requestMap =
context.getExternalContext().getRequestParameterMap();
String clientId = getClientId(context);
int increment;
if (requestMap.containsKey(clientId + MORE)) increment =
1;
else if (requestMap.containsKey(clientId + LESS))
increment = -1;
else increment = 0;
try {
int submittedValue
= Integer.parseInt((String) requestMap.get(clientId));

int newValue = getIncrementedValue(submittedValue,
increment);
setSubmittedValue("" + newValue);
setValid(true);
}
catch(NumberFormatException ex) {
// let the converter take care of bad input, but we
still have
// to set the submitted value or the converter won't
have
// any input to deal with
setSubmittedValue((String) requestMap.get(clientId));
}
}
```

Metoda dekodiranja posmatra zahtevane parametre kako bi odredila koje je dugme u okviru objekta spinner aktvirano. Ako je zahtevani parametar clientId.less, gde je clientId ID klijenta onog spinner-a koji treba da se dekoduje, onda se može ustanoviti da je dugme dekrement aktivirano. Ukoliko je zahtevani parametar clientId.more, onda se može ustanoviti da je dugme inkrement aktivirano.

Ako ne postoji nijedan parametar, onda se može ustanoviti da zahtev nije inicijalizovan od strane spinner-a i postavlja se inkrement na nulu. Ipak, još uvek je potrebno da se promeni vrednost, jer je možda uneta vrednost u tekst polje i aktivirano dugme “Next”.

Ukoliko postoje više spinner objekata na jednoj stranici onda svaka ova komponenta dekoduje svoj zahtev.

Kada jednom metoda dekodiranja odredi da je jedno dugme aktivirano, ona postavlja inkrementalnu vrednost na 1 ili –1, u zavisnosti koje je dugme aktivirano.
Tu inkrementalnu vrednost izračunava metoda getIncrementedValue :

```
private int getIncrementedValue(int submittedValue, int
increment) {
Integer minimum = (Integer)
getAttributes().get("minimum");
Integer maximum = (Integer)
getAttributes().get("maximum");
int newValue = submittedValue + increment;
if ((minimum == null || newValue >= minimum.intValue())
&&
(maximum == null || newValue <= maximum.intValue()))
return newValue;
else
return submittedValue;
}
```

Ova metoda proverava vrednost koji je korisnik uneo u spinner da li je u opsegu atributa minimum i maximum. Nakon što inkrementira vrednost, metoda dekodiranja poziva setSubmittedValue. Ta metoda postavlja prihvaćenu vrednost u komponentu.

Klasa javax.faces.component.UIComponent

I ova klasa ima metod

```
void decode(FacesContext context)
```

Ova metoda je pozvana od strane JSF okruženja na početku JSF životnog ciklusa, i to u slučaju ako je komponentni renderer tip jednak null vrednosti. Ova metoda dekodiranja dekoduje zahtevane parametre.

Klasa javax.faces.context.FacesContext

Metod ExternalContext getExternalContext() vraća referencu na context proxy. Tipično, pravi konktekst je servlet ili portlet context. Ukoliko se koristi eksterni konktekst umesto realnog, onda Java aplikacija može da radi sa servletima i portletima.

Klasa javax.faces.context.ExternalContext

Metod Map getRequestParameterMap() vraća element mape zahtevanih parametara. Uobičajene komponenta obično zovu ovu metodu u okviru metode decode() kako bi videli da li ima komponenti koje su ispalile zahtev.

Klasa javax.faces.component.EditableValueHolder

Metod void setSubmittedValue (Object submittedValue) postavlja komponentinu prihvaćenu vrednost. Ulazne komponente imaju vrednosti koje mogu da se menjaju, tako da UIInput implementira interfejs EditableValueHolder.
Metod void setValid(boolean valid) koriste napredne komponente kako bi utvrdili validnost svojih vrednosti. Ako komponenta ne može da konvertuje svoju vrednost, onda postavlja property valid na vrednost false.

Klasa javax.faces.component.ValueHolder

Metod void setConverter(Converter converter) se koristi, jer ulazne i izlazne komponente koriste određene vrednosti i zbog toga obe implementiraju interfejs ValueHolder. Njihove vrednosti moraju da budu konvertovane, tako da ValueHolder interfejs definiše metodu za postavljanje konvertora.

5.18 KORIŠĆENjE KONVERTORA

Spinner komponenta koristi standardni JSF konvertor celih brojeva kako bi konvertovao stringove u celobrojne objekte i obrnuto.
UISpinner konstruktor poziva setConverter, na sledeći način:

```
public class UISpinner extends UIInput {
...
public UISpinner() {
setConverter(new IntegerConverter()); // za konverziju
poslatih vrednosti
setRendererType(null); // this component renders itself
}
```

Ukoliko se unese nepravilan formata ulazne vrednosti, metoda dekodiranja će da proizvesti izuzetak tipa NumberFormatException, u okviru catch klauzulie. U nastavku poglavlja razmotriće se primeri koji koriste navedene komponente. U okviru prvog primera koristi se klasa

spinner/src/java/com/corejsf/UISpinner.java

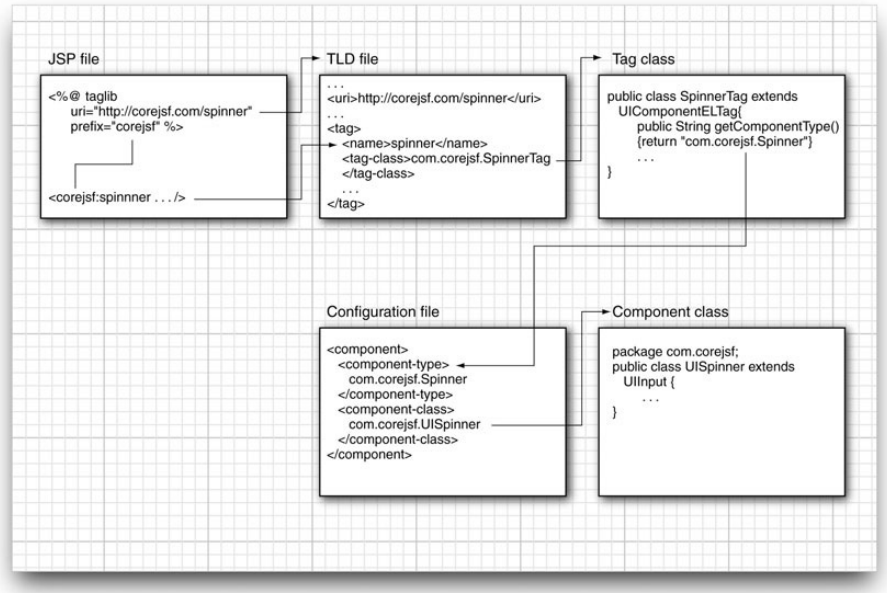
```
1. package com.corejsf;
2.
3. import java.io.IOException;
4. import java.util.Map;
5. import javax.faces.component.UIInput;
6. import javax.faces.context.FacesContext;
7. import javax.faces.context.ResponseWriter;
8. import javax.faces.convert.IntegerConverter;
9.
10. public class UISpinner extends UIInput {
11. private static final String MORE = ".more";
12. private static final String LESS = ".less";
13.
14. public UISpinner() {
15. setConverter(new IntegerConverter()); // to convert
the submitted value
16. setRendererType(null); // this component renders
itself
17. }370
18.
19. public void encodeBegin(FacesContext context) throws
IOException {
20. ResponseWriter writer = context.getResponseWriter();
21. String clientId = getClientId(context);
22.
```

```
23. encodeInputField(writer, clientId);
24. encodeDecrementButton(writer, clientId);
25. encodeIncrementButton(writer, clientId);
26. }
27.
28. public void decode(FacesContext context) {
29. Map<String, String> requestMap
30. =
context.getExternalContext().getRequestParameterMap();
31. String clientId = getClientId(context);
32.
33. int increment;
34. if (requestMap.containsKey(clientId + MORE))
increment = 1;
35. else if(requestMap.containsKey(clientId + LESS))
increment = -1;
36. else increment = 0;
37.
38. try {
39. int submittedValue
40. = Integer.parseInt((String)
requestMap.get(clientId));
41.
42. int newValue = getIncrementedValue(submittedValue,
increment);
43. setSubmittedValue("" + newValue);
44. setValid(true);
45. }
46. catch(NumberFormatException ex) {
47. // let the converter take care of bad input, but we
still have
48. // to set the submitted value, or the converter
won't have
49. // any input to deal with
50. setSubmittedValue((String)
requestMap.get(clientId));
51. }
52. }
53.
54. private void encodeInputField(ResponseWriter writer,
String clientId)
55. throws IOException {
56. writer.startElement("input", this);
57. writer.writeAttribute("name", clientId, "clientId");
58.
59. Object v = getValue();
60. if (v != null)
```

```
61. writer.writeAttribute("value", v.toString(),
"value");
62.
63. Integer size = (Integer)getAttributes().get("size");
64. if(size != null)
65. writer.writeAttribute("size", size, "size");
66.
67. writer.endElement("input");
68. }
69.
70. private void encodeDecrementButton(ResponseWriter
writer, String clientId)
71. throws IOException {
72. writer.startElement("input", this);
73. writer.writeAttribute("type", "submit", null);
74. writer.writeAttribute("name", clientId + LESS,
null);
75. writer.writeAttribute("value", "<", "value");
76. writer.endElement("input");
77. }
78. private void encodeIncrementButton(ResponseWriter
writer, String clientId)
79. throws IOException {
80. writer.startElement("input", this);
81. writer.writeAttribute("type", "submit", null);
82. writer.writeAttribute("name", clientId + MORE,
null);
83. writer.writeAttribute("value", ">", "value");
84. writer.endElement("input");
85. }
86.
87. private int getIncrementedValue(int submittedValue,
int increment) {
88. Integer minimum = (Integer)
getAttributes().get("minimum");
89. Integer maximum = (Integer)
getAttributes().get("maximum");
90. int newValue = submittedValue + increment;
91.
92. if ((minimum == null || newValue >=
minimum.intValue()) &&
93. (maximum == null || newValue <= maximum.intValue()))
94. return newValue;
95. else
96. return submittedValue;
97. }
98. }
```

TLD dokument

TLD (tag library descriptor) fajl opisuje jedan tag ili više tagova, kao njihove atribute. Ovaj fajl bi trebao da se nalazi u WEB-INF direktorijumu. Na slici je data šema za TLD fajl koji opisuje navedeni Spinner tag.



Slika 47. TLD fajl za Spinner tag

Namena ovog fajla je da specificira ime klase koja obrađuje tagove (com.corejsf.
SpinnerTag) i da se dozvole atributi za tagove (id, rendered, minimum, maximum, size, i value).Treba zapamtiti da uri tag definiše tag biblioteku:

```
<uri>http://corejsf.com/spinner</uri>
```

Ova URI adresa se referencira u okviru taglib direktive JSF stranice, na sledeći način:

```
<%@ taglib uri="http://corejsf.com/spinner"
prefix="corejsf" %>
```

Navedena taglib direktiva je analogna direktivama koje su definisanane standadnim prefiksom f i h , koje se koriste na svakoj JSF stranici.

JSF okruženje traži TLD fajlove u okviru sledećih lokacija:

- WEB-INF direktorijumu ili nekom njegovom poddirektorijumu.
- META-INF direktorijumu ili bilo kom JAR fajlu u WEB-INF/lib direktorijumu.

Većina atributa definisana u TLD fajlu sadrži deferred-value child element, kao što je ovaj:

```
<attribute>
<description>The spinner minimum value</description>
<name>minimum</name>
<deferred-value>
<type>int</type>
</deferred-value>
</attribute>
```

U primeru je minimum definisan kao vrednosni izraz tipa int. Za definisanje akcionog osluškivača koristi se sledeći tag:

```
<attribute>
<name>actionListener</name>
<deferred-method>
<method-signature>
void actionListener(javax.faces.event.ActionEvent)
</method-signature>
</deferred-method>
</attribute>
```

Atribut definisan kao runtime expression value je JSP izraz, ali ne JSF vrednosni izraz.

```
<attribute>
<description>The client id of this
component</description>
<name>id</name>
<rtexprvalue>true</rtexprvalue>
</attribute>
```

Sada se može u okviru razmatranog primera realizovati i TLD fajl spinner/web/WEB-INF/spinner.tld:

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <taglib xmlns="http://java.sun.com/xml/ns/javaee"
```

```
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
5. http://java.sun.com/xml/ns/javaee/web-
jsptaglibrary_2_1.xsd"
6. version="2.1">
7. <tlib-version>1.1</tlib-version>
8. <short-name>spinner</short-name>
9. <uri>http://corejsf.com/spinner</uri>
10.
11. <tag>
12. <name>spinner</name>
13. <tag-class>com.corejsf.SpinnerTag</tag-class>
14. <body-content>empty</body-content>
15. <attribute>
16. <description>A value binding that points to a bean
property</description>
17. <name>binding</name>
18. <deferred-value>
19. <type>javax.faces.component.UIComponent</type>
20. </deferred-value>
21. </attribute>
22.
23. <attribute>
24. <description>The client id of this
component</description>
25. <name>id</name>
26. <rtexprvalue>true</rtexprvalue>
27. </attribute>
28.
29. <attribute>
30. <description>Is this component
rendered?</description>
31. <name>rendered</name>
32. <deferred-value>
33. <type>boolean</type>
34. </deferred-value>
35. </attribute>
36.
37. <attribute>
38. <description>The spinner minimum value</description>
39. <name>minimum</name>
40. <deferred-value>
41. <type>int</type>
42. </deferred-value>
43. </attribute>
44.
45. <attribute>
```

```
46. <description>The spinner maximum value</description>
47. <name>maximum</name>
48. <deferred-value>
49. <type>int</type>
50. </deferred-value>
51. </attribute>
52.
53. <attribute>
54. <description>The size of the input
field</description>
55. <name>size</name>
56. <deferred-value>
57. <type>int</type>
58. </deferred-value>
59. </attribute>
60.
61. <attribute>
62. <description>The value of the spinner</description>
63. <name>value</name>
64. <required>true</required>
65. <deferred-value>
66. <type>int</type>
67. </deferred-value>
68. </attribute>
69. </tag>
70. </taglib>
```

Zajedno sa TLD fajlom, neophodno je da se podrži klasa tag handler za svaku naprednu komponentu. Za tag komponente, tag hendler klasa bi trebalo da bude podklasa klase UIComponentELTag. Za uobičajene konvertore neophodno je da tag hendler bude podklasa klase ComponentELTag, a za validator da njegov hendler bude podklasa klase ValidatorELTag.

Klase taga komponenti imaju pet odgovornosti:

- da identifikuje tip komponente
- da identifikuje tip modula
- da omogući setter metode za tag atribute.
- da postavlja vrednosti atributa taga u okviru tag komponente
- da oslobodi resurse.

Implementacija SpinnerTag klase je:

```
public class SpinnerTag extends UIComponentELTag {
private ValueExpression minimum;
private ValueExpression maximum;
private ValueExpression size;
private ValueExpression value;
...
}
```

Spinner tag klasa ima instancu polja za svaki atribut. Tag klasa bi trebalo da čuva sve atribute kao ValueExpression objekte.Takođe navedena klasa SpinnerTag identifikuje svoje komponente kao com.corejsf.Spinner i renderer tip kao null vrednost.

```
public String getComponentType() { return
"com.corejsf.Spinner"; }
public String getRendererType() { return null; }
```

SpinnerTag obezbeđuje setter metode za atribute koje podržava: minimum, maximum,
value i size. Metode su prikazane sledećim kodom:

```
public void setMinimum(ValueExpression newValue) {
minimum = newValue; }
public void setMaximum(ValueExpression newValue) {
maximum = newValue; }
public void setSize(ValueExpression newValue) { size =
newValue; }
public void setValue(ValueExpression newValue) { value =
newValue; }
```

Kada se procesira tag u JSF stranici, tag atribut se konveruje u ValueExpression objekat i poziva se setter metoda. Getter metode nisu potrebne.
Tag hendler mora da preklopu metodu setProperties kako bi kopirao vrednost atributa taga. Sledeći kod predstavlja implementaciju metode setProperties:

```
public void setProperties(UIComponent component) {
// pozvati metod osnovne klase
super.setProperties(component);
```

```
component.setValueExpression("size", size);
component.setValueExpression("minimum", minimum);
component.setValueExpression("maximum", maximum);
component.setValueExpression("value", value);
}
```

Kasnije, mogu se razvijati vrednosni izrazi korisšćenjem mape atributa. Na primer:

```
Integer minimum = (Integer)
component.getAttributes().get("minimum");
```

Na kraju se definiše metoda release koja resetuje sva polja instance na njihovu default vrednost:

```
public void release() {
// pozvati metod osnovne klase
super.release();
minimum = null;
maximum = null;
size = null;
value = null;
}
```

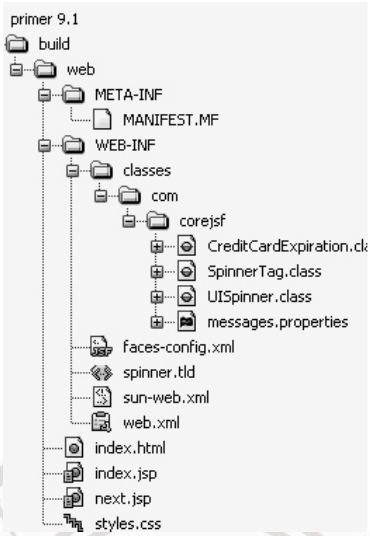
Ova metoda je neophodna jer JSF implementacija može da kešira tag hendler objekte i da ih ponovo koristi za parsiranje tagova.
U primeru kod klase spinner/src/java/com/corejsf/SpinnerTag.java je

```
1. package com.corejsf;
2.
3. import javax.el.ValueExpression;
4. import javax.faces.component.UIComponent;
5. import javax.faces.webapp.UIComponentELTag;
6.
7. public class SpinnerTag extends UIComponentELTag {
8. private ValueExpression minimum = null;
9. private ValueExpression maximum = null;
10. private ValueExpression size = null;
11. private ValueExpression value = null;
12.
13. public String getRendererType() { return null; }
14. public String getComponentType() { return
"com.corejsf.Spinner"; }
```

```
15.
16. public void setMinimum(ValueExpression newValue) {
minimum = newValue; }
17. public void setMaximum(ValueExpression newValue) {
maximum = newValue; }
18. public void setSize(ValueExpression newValue) { size
= newValue; }
19. public void setValue(ValueExpression newValue) {
value = newValue; }
20.
21. public void setProperties(UIComponent component) {
22. // always call the superclass method
23. super.setProperties(component);
24.
25. component.setValueExpression("size", size);
26. component.setValueExpression("minimum", minimum);
27. component.setValueExpression("maximum", maximum);
28. component.setValueExpression("value", value);
29. }
30.
31. public void release() {
32. // always call the superclass method
33. super.release();
34.
35. minimum = null;
36. maximum = null;
37. size = null;
38. value = null;
39. }
40. }
```

Klasa javax.faces.webapp.UIComponentELTag poseduje metod void setProperties(UIComponent component), koji prebacuje vrednost atributa taga do propertiy-ija komponente ili atributa komponente ili do oba. Napredne komponente moraju da pozivaju metodu setProperties svoje nadklase, kako bi se uverili da je property setovan na atribute tipa UIComponentELTag(binding, id, and rendered).
Metod void release() briše stanje taga kako bi on mogao ponovo da se koristi.
Klasa javax.faces.component.UIComponent ima metod
void setValueExpression (String name, ValueExpression expr).
Ukoliko je izraz konstanta bez #{...} izraza, onda se on razvija u onda se par (name, value) ubacuje u mapu atributa komponenti. U suprotnom će biti par (name, expr) ubačen u mapu vrednosnih izraza.

Struktura aplikacije je



Slika 48. Struktura aplikacije

Početna klijentska stranica je spinner/web/index.jsp

```
1. <html>
2. <%@ taglib uri="http://java.sun.com/jsf/core"
prefix="f" %>
3. <%@ taglib uri="http://java.sun.com/jsf/html"
prefix="h" %>
4. <%@ taglib uri="http://corejsf.com/spinner"
prefix="corejsf" %>
5. <f:view>
6. <head>
7. <link href="styles.css" rel="stylesheet"
type="text/css"/>
8. <title><h:outputText
value="#{msgs.windowTitle}"/></title>
9. </head>
10.
11. <body>
12. <h:form id="spinnerForm">
13. <h:outputText
value="#{msgs.creditCardExpirationPrompt}"
14. styleClass="pageTitle"/>
15. <p/>
16. <h:panelGrid columns="3">
17. <h:outputText value="#{msgs.monthPrompt}"/>
```

```
18. <corejsf:spinner value="#{cardExpirationDate.month}"
19. id="monthSpinner" minimum="1" maximum="12"
size="3"/>
20. <h:message for="monthSpinner"/>
21. <h:outputText value="#{msgs.yearPrompt}"/>
22. <corejsf:spinner value="#{cardExpirationDate.year}"
23. id="yearSpinner" minimum="1900" maximum="2100"
size="5"/>
24. <h:message for="yearSpinner"/>
25. </h:panelGrid>
26. <p/>
27. <h:commandButton value="#{msgs.nextButtonPrompt}"
action="next"/>
28. </h:form>
29. </body>
30. </f:view>
31. </html>
```

Sledeća klijentska stranica je spinner/web/next.jsp

```
1. <html>
2. <%@ taglib uri="http://java.sun.com/jsf/core"
prefix="f" %>
3. <%@ taglib uri="http://java.sun.com/jsf/html"
prefix="h" %>
4.
5. <f:view>
6. <head>
7. <link href="styles.css" rel="stylesheet"
type="text/css"/>
8. <title><h:outputText
value="#{msgs.windowTitle}"/></title>
9. </head>
10. <body>
11. <h:form>
12. <h:outputText value="#{msgs.youEnteredPrompt}"
styleClass="pageTitle"/>
13. <p>
14. <h:outputText value="#{msgs.expirationDatePrompt}"/>
15. <h:outputText value="#{cardExpirationDate.month}"/>
/
16. <h:outputText value="#{cardExpirationDate.year}"/>
17. <p>
18. <h:commandButton value="Try again" action="again"/>
19. </h:form>
20. </body>
21. </f:view>
22. </html>
```

U slučaju drada sa kreditnom karticom koristi se spinner/src/java/com/corejsf/CreditCardExpiration.java

```
1. package com.corejsf;
2.
3. public class CreditCardExpiration {
4. private int month = 1;
5. private int year = 2000;
6.
7. // PROPERTY: month
8. public int getMonth() { return month; }
9. public void setMonth(int newValue) { month =
newValue; }
10.
11. // PROPERTY: year
12. public int getYear() { return year; }
13. public void setYear(int newValue) { year = newValue;
}
14. }
```

Konfiguracioni fajl spinner/web/WEB-INF/faces-config.xml je

```
1. <?xml version="1.0"?>
2. <faces-config
xmlns="http://java.sun.com/xml/ns/javaee"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
5. http://java.sun.com/xml/ns/javaee/web-
facesconfig_1_2.xsd"
6. version="1.2">
7. <navigation-rule>
8. <from-view-id>/index.jsp</from-view-id>
9. <navigation-case>382
10. <from-outcome>next</from-outcome>
11. <to-view-id>/next.jsp</to-view-id>
12. </navigation-case>
13. </navigation-rule>
14.
15. <navigation-rule>
16. <from-view-id>/next.jsp</from-view-id>
17. <navigation-case>
18. <from-outcome>again</from-outcome>
19. <to-view-id>/index.jsp</to-view-id>
20. </navigation-case>
21. </navigation-rule>
22.
```

```
23. <component>
24. <component-type>com.corejsf.Spinner</component-type>
25. <component-class>com.corejsf.UISpinner</component-
class>
26. </component>
27.
28. <managed-bean>
29. <managed-bean-name>cardExpirationDate</managed-bean-
name>
30. <managed-bean-
class>com.corejsf.CreditCardExpiration</managed-bean-
class>
31. <managed-bean-scope>session</managed-bean-scope>
32. </managed-bean>
33.
34. <application>
35. <resource-bundle>
36. <base-name>com.corejsf.messages</base-name>
37. <var>msgs</var>
38. </resource-bundle>
39. </application>
40. </faces-config>
```

Koristi se i pomoćni fajl spinner/src/java/com/corejsf/messages.properties

```
1. windowTitle=Spinner Test
2. creditCardExpirationPrompt=Please enter your credit
card expiration date:
3. monthPrompt=Month:
4. yearPrompt=Year:
5. nextButtonPrompt=Next
6. youEnteredPrompt=You entered:
7. expirationDatePrompt=Expiration Date
8. changes=Changes:
```

Koristi se i CSS fajl spinner/web/styles.css

```
1. body {
2. background: #eee;
3. }
4. .pageTitle {
5. font-size: 1.25em;
6. }
```

Izvršavanjem navedenih stranica dobijaju se rezultati:



Slika 49. Rezultati aplikacije

U ranijoj verziji JSF 1.1 obezbeđuje dve odvojene tag nadklase UIComponentTag i UIComponent- BodyTag.
Samo četiri standardna taga u JSF 1.1 proširuju UIComponentBodyTag: f:view, f:verbatim, h:commandLink i h:outputLink. Spiner komponenta ne procesira svoj body, tako da bi mogla da proširuje UIComponentTag.

Pre JSF 1.2, TLD fajlovi su imali DOCTYPE deklaraciju umesto schema deklaracije, kao što je ova:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib PUBLIC "-//Sun Microsystems, Inc.//DTD
JSP Tag Library 1.2//EN"
"http://java.sun.com/dtd/web-jsptaglibrary_1_2.dtd">
<taglib>
<tlib-version>0.03</tlib-version>
<jsp-version>1.2</jsp-version>
...
</taglib>
```

Napomena je da se više ne mogu da se koriste deferred-value ili deferred-method elementi kao atributi. Umesto, da tag hendler klasa uzima prosleđene string izraze, on mora da konvertuje objekte u tip ValueBinding ili MethodBinding. Ove klase su analogne klasama ValueExpression i MethodExpression, koje se koriste u JSF 1.1. Takođe tag hendler mora da definiše setters metode za stringove. Na primer:

```
public class SpinnerTag extends UIComponentTag {
private String minimum;
private String maximum;
...
public void setMinimum(String newValue) { minimum =
newValue; }
public void setMaximum(String newValue) { maximum =
newValue; }
...
}
```

U metodi setProperties, u stvari se proverava da li je string vezivajuća vrednost. Njena implementacija je:

```
public void setProperties(UIComponent component) {
super.setProperties(component);
setInteger(component, "minimum", minimum);
setInteger(component, "maximum", maximum);
}
```

U okviru metode setProperties poziva se pomoćna metoda setInteger, čija je implementacija data sledećim kodom :

```
public void setInteger(UIComponent component, String
name, String expr) {
if (expr == null) return null;
else if (UIComponentTag.isValueReference(expr)) {
FacesContext context =
FacesContext.getCurrentInstance();
Application app = context.getApplication();
ValueBinding binding = app.createValueBinding(expr);
component.setValueBinding(name, binding);
}
else
component.getAttributes().put(name, new Integer(expr));
}
```

Za druge tipove podataka potrebne su pomoćne metode setString, setBoolean... Sledeći kod kreira vezivanje metode za osluškivača događaja koji reaguje na promenu vrednosti.

```
FacesContext context =
FacesContext.getCurrentInstance();
Application app = context.getApplication();
Class[] paramTypes = new Class[] {
ValueChangeListener.class };
MethodBinding mb =
app.createMethodBinding(attributeValue, paramTypes);
```

Potom se postavlja objekat MethodBinding sa sledećom komponentom:

```
((EditableValueHolder)
component).setValueChangeListener(mb);
```

Akcioni osluškivači i validatori su po sličnom šablonu. Međutim, akcije su nešto složenije. Akcija može da bude vezujuća metoda ili fiksni string. Na primer:

```
<h:commandButton value="Login"
action="#{loginController.verifyUser}"/>
```

ili

```
<h:commandButton value="Login" action="login"/>
```

Metoda setAction u interfejsu ActionSource zahteva MethodBinding u svim slučajevima.
Ukoliko je akcija fiksni string onda mora da se konstruiše objekat MethodBinding koji sadži metodu getExpressionString, koja vraća string:

```
if (UIComponentTag.isValueReference(attributeValue))
((ActionSource) component).setMethodBinding(component,
"action", attributeValue,
new Class[] {});
else {
FacesContext context =
FacesContext.getCurrentInstance();
Application app = context.getApplication();
MethodBinding mb = new
ActionMethodBinding(attributeValue);
component.getAttributes().put("action", mb);
}
```

Sledeći kod predstavlja klasu ActionMethodBinding :

```
public class ActionMethodBinding extends MethodBinding
implements Serializable {
private String result;
public ActionMethodBinding(String result) { this.result
= result; }
public Object invoke(FacesContext context, Object
params[]) { return result; }
public String getExpressionString() { return result; }
public Class getType(FacesContext context) { return
String.class; }
}
```

Klasa

javax.faces.component.ActionSource

ima metod void setActionListenerm (MethodBinding m), koji postavlja vezivajuću metodu osluškivača akcionih događaja za datu komponentu. Povr atna vrednost mora da bude tipa void i prosleđuje se ActionEvent. Njegov metod void setAction(MethodBinding m) postavlja vezivajuću metodu akcije za datu komponentu. Metoda može da vrati objekat bilo kog tipa i ona nema parametre.
Spinner koji je razmatran do sada ima dve mane. Prva mana je to što se spinner sam definiše, tako da ne može da veže odvojene module za sebe, na primer kada se prelazi sa aplikacije na mobilni telefon. Druga je to što

spinner zahteva da se svaki put obrati serveru kada korisnik klikne inkrement ili dekrement dugme.
Upravo prednost ove komponente je to što on može da veže osluškivače događaja koji reaguju na promenu vrednosti.
U prethodnom kodu, UISpinner klasa bila je zadužena za svoje sopstveno modelovanje. Međutim, većina UI klasa delegiraju modelovanje na odvojene klase. Korišćenje odvojenih klasa je dobra ideja. Lako je zameniti module, pa ih adaptirati na različite UI alate.
Korišćenje spoljnih modula (external renderer) zahteva sledeće korake:

1. Definisanje ID string za sopstveni modul
2. Deklarisanje modula u JSF konfiguracionom fajlu.
3. Modifikacija taga klase, odnosno metode getRendererType, tako da ona vrati ID modula.
4. Implementiranje renderer klase

Indetifikator, u razmatranom primeru, com.corejsf.Spinner mora da bude definisan u okviru JSF konfiguracionog fajla na sledeći način:

```
<faces-config>

   <component>
        <component-type>com.corejsf.Spinner</component-
type>
        <component-
class>com.corejsf.UISpinner</component-class>
   </component>

 <render-kit>
   <renderer>
      <component-family>javax.faces.Input</component-
family>
      <renderer-type>com.corejsf.Spinner</renderer-type>
      <renderer-
class>com.corejsf.SpinnerRenderer</renderer-class>
   </renderer>
 </render-kit>
</faces-config>
```

Moduli su određeni tipom renderer i komponentom familije. getRendererType tag klase treba da vrati renderer ID.

```
public class SpinnerTag extends UIComponentTag {
...
public String getComponentType() { return
"com.corejsf.Spinner"; }
public String getRendererType() { return
"com.corejsf.Spinner"; }

}
```

U sledećoj tabeli su definisane klase komponenti i njihove familije.

Klasa komponenti	Familija komponenti
UICommand	javax.faces.Command
UIData	javax.faces.Data
UIForm	javax.faces.Form
UIGraphic	javax.faces.Graphic
UIInput	javax.faces.Input
UIMessage	javax.faces.Message
UIMessages	javax.faces.Messages
UIOutput	javax.faces.Output
UIPanel	javax.faces.Panel
UISelectBoolean	javax.faces.SelectBoolean
UISelectMany	javax.faces.SelectMany
UISelectOne	javax.faces.SelectOne

Dobra je ideja da se renderer tip postavi u konstruktor komponente:

```
public class UISpinner extends UIInput {
public UISpinner() {
setConverter(new IntegerConverter()); // to convert the
submitted value
setRendererType("com.corejsf.Spinner"); // this
component has a renderer
}
}
```

Renderer proširuje klasu javax.faces.render.Renderer, koja ima sedam metoda koje su dosta slične:

- void encodeBegin(FacesContext context, UIComponent component)
- void encodeChildren(FacesContext context, UIComponent component)
- void encodeEnd(FacesContext context, UIComponent component)
- void decode(FacesContext context, UIComponent component)
- Object getConvertedValue(FacesContext context, UIComponent component, Object submittedValue)
- boolean getRendersChildren()
- String convertClientId(FacesContext context, String clientId)
- getConvertedValue metoda konvertuje vrednost koja je prihvaćena iz stringa u objekat.
- getRendersChildren metoda specificira da li je renderer odgovoran za modeliranje dece svojih komponenti. Ako metoda vraća true, JSF će pozvati metodu encodeChildren;
- a ako vrati false (definisano ponašanje), JSF neće pozvati tu metodu i deca će biti kodirana odvojeno.
- convertClientId metoda konveruje ID string tako da on može da bude korišćen od strane klijenta.

U sledećem primeru se opet koristi klasa
spinner2/src/java/com/corejsf/UISpinner.java

```
1. package com.corejsf;
2.
3. import javax.faces.component.UIInput;
4. import javax.faces.convert.IntegerConverter;
5.
6. public class UISpinner extends UIInput {
7. public UISpinner() {
8. setConverter(new IntegerConverter()); // to convert
the submitted value
9. }
10. }
```

Sada postoji i klasa spinner2/src/java/com/corejsf/SpinnerRenderer.java

```
1. package com.corejsf;
2.
3. import java.io.IOException;
4. import java.util.Map;
```

```
5. import javax.faces.component.UIComponent;
6. import javax.faces.component.EditableValueHolder;
7. import javax.faces.component.UIInput;
8. import javax.faces.context.FacesContext;
9. import javax.faces.context.ResponseWriter;
10. import javax.faces.convert.ConverterException;
11. import javax.faces.render.Renderer;
12.
13. public class SpinnerRenderer extends Renderer {
14. private static final String MORE = ".more";
15. private static final String LESS = ".less";
16.
17. public Object getConvertedValue(FacesContext
context, UIComponent component,
18. Object submittedValue) throws ConverterException {
19. return
com.corejsf.util.Renderers.getConvertedValue(context,
component,
20. submittedValue);
21. }
22.
23. public void encodeBegin(FacesContext context,
UIComponent spinner)
24. throws IOException {
25. ResponseWriter writer = context.getResponseWriter();
26. String clientId = spinner.getClientId(context);
27.
28. encodeInputField(spinner, writer, clientId);
29. encodeDecrementButton(spinner, writer, clientId);
30. encodeIncrementButton(spinner, writer, clientId);
31. }
32.
33. public void decode(FacesContext context, UIComponent
component) {
34. EditableValueHolder spinner = (EditableValueHolder)
component;
35. Map<String, String> requestMap
36. =
context.getExternalContext().getRequestParameterMap();
37. String clientId = component.getClientId(context);
38.
39. int increment;
40. if (requestMap.containsKey(clientId + MORE))
increment = 1;
41. else if (requestMap.containsKey(clientId + LESS))
increment = -1;
42. else increment = 0;
```

```
43.
44. try {
45. int submittedValue
46. = Integer.parseInt((String)
requestMap.get(clientId));
47.
48. int newValue = getIncrementedValue(component,
submittedValue,
49. increment);
50. spinner.setSubmittedValue("" + newValue);
51. spinner.setValid(true);
52. }
53. catch(NumberFormatException ex) {
54. // let the converter take care of bad input, but we
still have
55. // to set the submitted value, or the converter
won't have
56. // any input to deal with
57. spinner.setSubmittedValue((String)
requestMap.get(clientId));
58. }
59. }
60.
61. private void encodeInputField(UIComponent spinner,
ResponseWriter writer,
62. String clientId) throws IOException {
63. writer.startElement("input", spinner);
64. writer.writeAttribute("name", clientId, "clientId");
65.
66. Object v = ((UIInput) spinner).getValue();
67. if(v != null)
68. writer.writeAttribute("value", v.toString(),
"value");
69.
70. Integer size = (Integer)
spinner.getAttributes().get("size");
71. if(size != null)
72. writer.writeAttribute("size", size, "size");
73.
74. writer.endElement("input");
75. }
76.
77. private void encodeDecrementButton(UIComponent
spinner,
78. ResponseWriter writer, String clientId) throws
IOException {
79. writer.startElement("input", spinner);
```

```
80. writer.writeAttribute("type", "submit", null);
81. writer.writeAttribute("name", clientId + LESS,
null);
82. writer.writeAttribute("value", "<", "value");
83. writer.endElement("input");
84. }
85.
86. private void encodeIncrementButton(UIComponent
spinner,
87. ResponseWriter writer, String clientId) throws
IOException {
88. writer.startElement("input", spinner);
89. writer.writeAttribute("type", "submit", null);
90. writer.writeAttribute("name", clientId + MORE,
null);
91. writer.writeAttribute("value", ">", "value");
92. writer.endElement("input");
93. }
94.
95. private int getIncrementedValue(UIComponent spinner,
int submittedValue,
96. int increment) {
97. Integer minimum = (Integer)
spinner.getAttributes().get("minimum");
98. Integer maximum = (Integer)
spinner.getAttributes().get("maximum");
99. int newValue = submittedValue + increment;
100.
101. if ((minimum == null || newValue >=
minimum.intValue()) &&
102. (maximum == null || newValue <=
maximum.intValue()))
103. return newValue;
104. else
105. return submittedValue;
106. }
107. }
```

U prethodnom delu primera vidi se da je spinner obrađivao konverzije buđenjem metode setConverter(), u svom konstruktoru. Kako je spinner ulazna komponenta onda ona proširuje klasu UIInput. Kada spinner delegira svom modulu, dužnost je modula da konveruje vrednosti spinnera, tako što proširuje metodu Renderer.getConvertedValue().

U okviru klase javax.faces.component.UIInput postoji kod:

```
public void getConvertedValue(FacesContext context,
Object newSubmittedValue)
throws ConverterException {
Object newValue = newSubmittedValue;
if (renderer != null) {
newValue = renderer.getConvertedValue(context, this,
newSubmittedValue);
} else if (newSubmittedValue instanceof String) {
Converter converter = getConverterWithType(context); //
a private method
if (converter != null)
newValue = converter.getAsObject(
context, this, (String) newSubmittedValue);
}
return newValue;
}
```

Privatna metoda getConverterWithType traži odgovarajući konvertor za vrednost komponente.
Ako je korišćena napredna komponentra u stvari ulazna komponenta, može da se generiše događaj koji reaguje na promenu vrednosti, kako bi se zainteresovao osluškivač. Podrška za osluškivače događaja koji reaguju na promenu vrednosti je vrlo jednostavna. UIInput klasa automatski generiše događaje koji reaguju na promenu vrednost kad god se unese ulazna vrednost.
Postoje dva načina da se veže osluškivač koji reaguje na promenu vrednosti.Prvi je: dodavanje jednog ili više osluškivača sa f:valueChangeListener, kao na primer:

```
<corejsf:spinner ...>
<f:valueChangeListener
type="com.corejsf.SpinnerListener"/>
...
</corejsf:spinner>
```

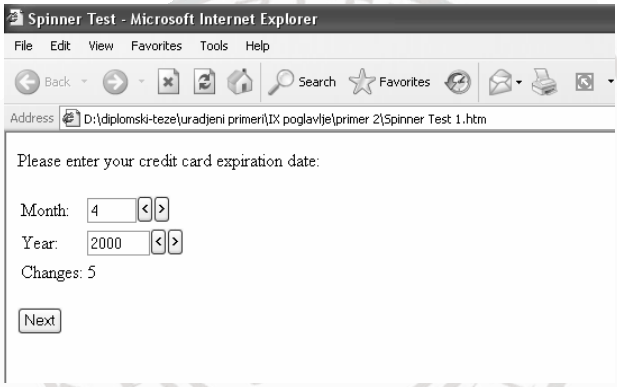
ili sa valueChangeListener atributom:

```
<corejsf:spinner value="#{cardExpirationDate.month}"
id="monthSpinner" minimum="1" maximum="12" size="3"
valueChangeListener="#{cardExpirationDate.changeListener
}"/>
```

Prvi način ne zahteva bilo kakvu podršku implementora komponente. Drugi zahteva da tag hendler podrži atribut valueChangeListener.
Sada klasa koja radi sa kreditnim karticama izgleda

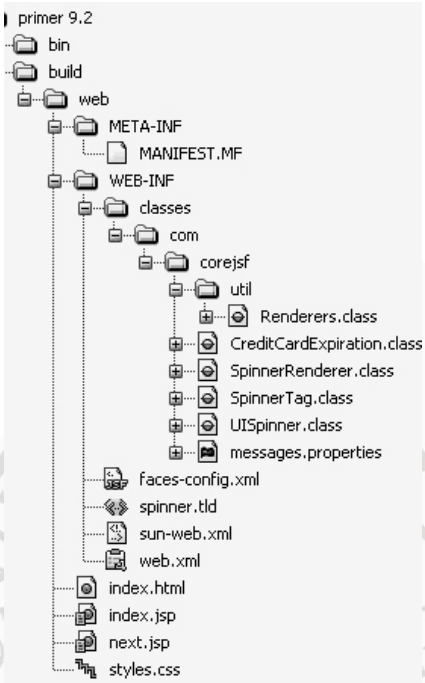
```
public class CreditCardExpiration {
private int changes = 0;
// to demonstrate the value change listener
public void changeListener(ValueChangeEvent e) {
changes++;
}
}
```

Rezultat je



Slika 50. Izgled aplikacije

Struktura ove aplikacije je



Slika 51. Struktura aplikacije

Klasa Spinner2/src/java/com/corejsf/util/Renderers.java

```
1. package com.corejsf.util;
2.
3. import java.util.ArrayList;
4. import java.util.Arrays;
5. import java.util.Collection;
6. import java.util.List;
7. import java.util.Map;
8.
9. import javax.el.ValueExpression;
10. import javax.faces.application.Application;
11. import javax.faces.component.UIComponent;
12. import javax.faces.component.UIForm;
13. import javax.faces.component.UISelectItem;
14. import javax.faces.component.UISelectItems;
15. import javax.faces.component.ValueHolder;
16. import javax.faces.context.FacesContext;
17. import javax.faces.convert.Converter;
```

```
18. import javax.faces.convert.ConverterException;
19. import javax.faces.model.SelectItem;
20.
21. public class Renderers {
22. public static Object getConvertedValue(FacesContext
context,
23. UIComponent component, Object submittedValue)
24. throws ConverterException {
25. if (submittedValue instanceof String) {
26. Converter converter = getConverter(context,
component);
27. if (converter != null) {
28. return converter.getAsObject(context, component,
29. (String) submittedValue);
30. }
31. }
32. return submittedValue;
33. }
34.
35. public static Converter getConverter(FacesContext
context,
36. UIComponent component) {
37. if (!(component instanceof ValueHolder)) return
null;
38. ValueHolder holder = (ValueHolder) component;
39.
40. Converter converter = holder.getConverter();
41. if (converter != null)
42. return converter;
43.
44. ValueExpression expr =
component.getValueExpression("value");
45. if (expr == null) return null;
46.
47. Class targetType =
expr.getType(context.getELContext());
48. if (targetType == null) return null;
49. // Version 1.0 of the reference implementation will
not apply a converter
50. // if the target type is String or Object, but that
is a bug.
51.
52. Application app = context.getApplication();
53. return app.createConverter(targetType);
54. }
55.
```

```
56. public static String getFormId(FacesContext context,
UIComponent component) {
57. UIComponent parent = component;
58. while (!(parent instanceof UIForm))
59. parent = parent.getParent();
60. return parent.getClientId(context);
61. }
62.
63. @SuppressWarnings("unchecked")
64. public static List<SelectItem>
getSelectItems(UIComponent component) {
65. ArrayList<SelectItem> list = new
ArrayList<SelectItem>();
66. for (UIComponent child : component.getChildren()) {
67. if (child instanceof UISelectItem) {
68. Object value = ((UISelectItem) child).getValue();
69. if (value == null) {
70. UISelectItem item = (UISelectItem) child;
71. list.add(new SelectItem(item.getItemValue(),
72. item.getItemLabel(),
73. item.getItemDescription(),
74. item.isItemDisabled()));
75. } else if (value instanceof SelectItem) {
76. list.add((SelectItem) value);
77. }
78. } else if (child instanceof UISelectItems) {
79. Object value = ((UISelectItems) child).getValue();
80. if (value instanceof SelectItem)
81. list.add((SelectItem) value);
82. else if (value instanceof SelectItem[])
83. list.addAll(Arrays.asList((SelectItem[]) value));400
84. else if (value instanceof Collection)
85. list.addAll((Collection<SelectItem>) value); //
unavoidable
86. // warning
87. else if (value instanceof Map) {
88. for (Map.Entry<?, ?> entry : ((Map<?, ?>)
value).entrySet())
89. list.add(new SelectItem(entry.getKey(),
90. "" + entry.getValue()));
91. }
92. }
93. }
94. return list;
95. }
96. }
```

Klasa koja definiše Spinner tag

spinner2/src/java/com/corejsf/SpinnerTag.java

```
1. package com.corejsf;
2.
3. import javax.el.MethodExpression;
4. import javax.el.ValueExpression;
5. import javax.faces.component.EditableValueHolder;
6. import javax.faces.component.UIComponent;
7. import
javax.faces.event.MethodExpressionValueChangeListener;
8. import javax.faces.webapp.UIComponentELTag;
9.
10. public class SpinnerTag extends UIComponentELTag {
11. private ValueExpression minimum = null;
12. private ValueExpression maximum = null;
13. private ValueExpression size = null;
14. private ValueExpression value = null;
15. private MethodExpression valueChangeListener = null;
16.
17. public String getRendererType() { return
"com.corejsf.Spinner"; }
18. public String getComponentType() { return
"com.corejsf.Spinner"; }
19.
20. public void setMinimum(ValueExpression newValue) {
minimum = newValue; }
21. public void setMaximum(ValueExpression newValue) {
maximum = newValue; }
22. public void setSize(ValueExpression newValue) { size
= newValue; }
23. public void setValue(ValueExpression newValue) {
value = newValue; }
24. public void setValueChangeListener(MethodExpression
newValue) {
25. valueChangeListener = newValue;
26. }
27.
28. public void setProperties(UIComponent component) {
29. // always call the superclass method
30. super.setProperties(component);
31.
32. component.setValueExpression("size", size);
33. component.setValueExpression("minimum", minimum);
34. component.setValueExpression("maximum", maximum);
35. component.setValueExpression("value", value);
```

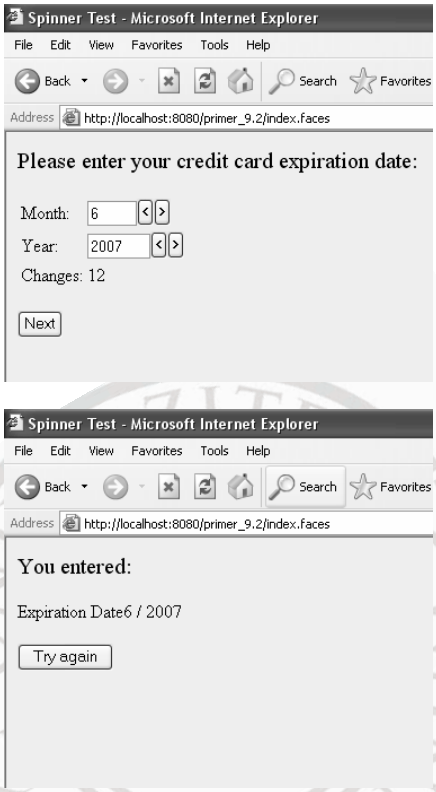
```
36. if (valueChangeListener != null)
37. ((EditableValueHolder)
component).addValueChangeListener(
38. new
MethodExpressionValueChangeListener(valueChangeListener)
);
39. }
40.
41. public void release() {
42. // always call the superclass method
43. super.release();
44.
45. minimum = null;
46. maximum = null;
47. size = null;
48. value = null;
49. valueChangeListener = null;
50. }
51. }
```

Konfiguracioni spinner2/web/WEB-INF/faces-config.xml fajl

```
1. <?xml version="1.0"?>
2.
3. <faces-config
xmlns="http://java.sun.com/xml/ns/javaee"
4. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5. xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
6. http://java.sun.com/xml/ns/javaee/web-
facesconfig_1_2.xsd"
7. version="1.2">
8.
9. <navigation-rule>
10. <from-view-id>/index.jsp</from-view-id>
11. <navigation-case>
12. <from-outcome>next</from-outcome>
13. <to-view-id>/next.jsp</to-view-id>
14. </navigation-case>
15. </navigation-rule>
16.
17. <navigation-rule>
18. <from-view-id>/next.jsp</from-view-id>
19. <navigation-case>
20. <from-outcome>again</from-outcome>
21. <to-view-id>/index.jsp</to-view-id>
22. </navigation-case>
```

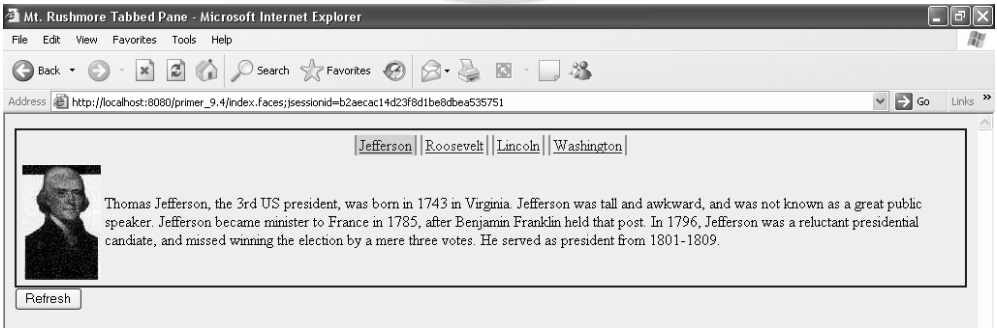
```
23. </navigation-rule>
24.
25. <managed-bean>
26. <managed-bean-name>cardExpirationDate</managed-bean-
name>
27. <managed-bean-
class>com.corejsf.CreditCardExpiration</managed-bean-
class>
28. <managed-bean-scope>session</managed-bean-scope>
29. </managed-bean>
30.
31. <component>
32. <component-type>com.corejsf.Spinner</component-type>
33. <component-class>com.corejsf.UISpinner</component-
class>
34. </component>
35.
36. <render-kit>
37. <renderer>
38. <component-family>javax.faces.Input</component-
family>
39. <renderer-type>com.corejsf.Spinner</renderer-type>
40. <renderer-
class>com.corejsf.SpinnerRenderer</renderer-class>
41. </renderer>
42. </render-kit>
43.
44. <application>
45. <resource-bundle>
46. <base-name>com.corejsf.messages</base-name>
47. <var>msgs</var>
48. </resource-bundle>
49. </application>
50. </faces-config>
```

Dobijeni rezultati su:



Slika 52. Rezultat aplikacije

U ovom odeljku se razmatraju tabovi(tabbed pane).



Slika 53. Primer tabova

Tabbed pane komponenta se razlikuje od tabbed pane implementacije koja je rađena u prethonom poglavlju. U toj implementaciji su se koristile h:graphicImage i h:commandLink. Sada će se kreirati upotrebljive komponente koje se mogu jednostavno ubaciti u stranicu. Tabbed pane komponenta ima sledeće osobine:

- Može da se koristi CSS klase za tabbed pane za selektovane i neselektovane tabove.
- Može da se specifira sa f:selectItem tagom (ili f:selectItems), na isti način kao i standardni JSF menu i listbox tagovi specificiraju menu ili listbox items.
- Može da se specificira tabbed pane sadržaj sa facet (koji je renderer renders).
- Može da se doda osuškivač akcija tabbed pane-u. Osluškivač se obaveštava kad god je ta selektovan.
- Može da se lokalizuje tab text by specificiranjem ključa iz resource bundle umesto postojećeg teksta koji je prikazan u tabu.
- tabbed pane koristi sakrivena polja kako bi emitovala selektovani tab i njegov sadržaj, od klijenta ka serveru.

Postoje više načina u kojima se koristi tabbed pane. Jedan od njih je i:

```
<corejsf:tabbedPane >
<f:selectItem itemLabel="Jefferson"
itemValue="jefferson"/>
<f:selectItem itemLabel="Roosevelt"
itemValue="roosevelt"/>
<f:selectItem itemLabel="Lincoln" itemValue="lincoln"/>
<f:selectItem itemLabel="Washington"
itemValue="washington"/>
<f:facet name="jefferson">
<h:panelGrid columns="2">
<h:graphicImage value="/images/jefferson.jpg"/>
<h:outputText value="#{msgs.jeffersonDiscussion}"/>
</h:panelGrid>
</f:facet>
<!-- three more facets -->
...
</corejsf:tabbedPane>
```

Može da se koristi i CSS stil, kao:

```
<corejsf:tabbedPane styleClass="tabbedPane"
tabClass="tab" selectedTabClass="selectedTab">
```

Može da se koristi i jedinstveni f:selectItems tag koji je multiple f:selectitem tagova:

```
<corejsf:tabbedPane styleClass="tabbedPane"
tabClass="tab" selectedTabClass="selectedTab">
<f:selectItems value="#{myBean.tabs}"/>
...
</corejsf:tabbedPane>
```

Specificira se resource bundle sa resourceBundle atributom,kao što je:

```
<corejsf:tabbedPane
resourceBundle="com.corejsf.messages">
<f:selectItem itemLabel="jeffersonTabText"
itemValue="jefferson"/>
<f:selectItem itemLabel="rooseveltTabText"
itemValue="roosevelt"/>
<f:selectItem itemLabel="lincolnTabText"
itemValue="lincoln"/>
<f:selectItem itemLabel="washingtonTabText"
itemValue="washington"/>
...
</corejsf:tabbedPane>
```

U datom primeru je item oznaka—ključ u messages resource bundle:

```
jeffersonTabText=Jefferson
rooseveltTabText=Roosevelt
lincolnTabText=Lincoln
washingtonTabText=Washington
```

Tabbed pane komponenta ispaljuje akcioni događaj kada korisnik selektuje tab. Može da se koristi f:actionListener tag da bi se dodao jedan ili više osluškivača, ili da se specificira metoda koja obrađuje akcione događaje sa tabbed pane-ovim actionListener atributom:

```
<corejsf:tabbedPane                                          ...
actionListener="#{tabbedPaneBean.presidentSelected}">
<f:selectItems value="#{tabbedPaneBean.tabs}"/>
</corejsf:tabbedPane>
```

Tabbed pane dozvoljava da se specificraju tabovi sa f:selectItem ili f:selectItems. Te tagove kreira UISelectItem komponenta i dodaje ih kao decu.
Tabbed pane renderer ima decu i njegovi definiše tu decu, zbog toga on proširuje metode rendersChildren() i encodeChildren():

```
public boolean rendersChildren() {
return true;
}

public void encodeChildren(FacesContext context,
UIComponent component)
throws java.io.IOException {
// if the tabbedpane component has no children, this
method is still called
if (component.getChildCount() == 0) {
return;
}
...
List items =
com.corejsf.util.Renderers.getSelectItems(context,
component);
Iterator it = items.iterator();
while (it.hasNext())
encodeTab(context, writer, (SelectItem) it.next(),
component);
...
}
...
}
```

Generalno, komponenta koja procesira svoju decu sadrži sledeći kod:

```
Iterator children = component.getChildren().iterator();
while (children.hasNext()) {
UIComponent child = (UIComponent) children.next();
processChild(context, writer, child, component);
}
```

Tabbed pane koristi facet imena za sadržaj koji je pridružen određenom tagu. Metoda encodeEnd je odgovorna za modelovanje selektovanog facet:

```
public void encodeEnd(FacesContext context, UIComponent
component)
throws java.io.IOException {
ResponseWriter writer = context.getResponseWriter();
UITabbedPane tabbedPane = (UITabbedPane) component;
String content = tabbedPane.getContent();
...
if (content != null) {
UIComponent facet = component.getFacet(content);
if (facet != null) {
if (facet.isRendered()) {
facet.encodeBegin(context);
if (facet.getRendersChildren())
facet.encodeChildren(context);
facet.encodeEnd(context);
}
}
}
}
...
}
```

Klasa UITabbedPane ima polje content, koje postavlja ime facet-a ili URL tekućeg taba. Metoda encodeEnd proverava da li je sadržaj tekućeg selektovanog taba facet-a date komponente. Ako jeste,onda ona kodira facet, tako što budi svoje encodeBegin, encodeChildren i encodeEnd metode.
Takođe, klasa

javax.faces.component.UIComponent

ima metod UIComponent getFacet(String facetName) koji vraća referencu na facet, ako postoji. Ukoliko facet ne postoji, onda metoda vraća null vrednost. Metod boolean getRendersChildren() vraća vrednost true ako komponenta modeluje naslednike, u suprotnom vraća vrednost false. Komponentina encodeChildren metoda neće biti pozvana ako ova metoda ne vraća true. Treba naglasiti po definiciji getRendersChildren vraća false. Metod boolean isRendered() vraća rendered property. Komponenta se sama definiše, ako je rendered property postavljen na true.

Upotreba CSS stilova

JSF aplikacije mogu da podrže CSS stil, i sama realizacija je u dva koraka:

1. da se doda atribut na tag library descriptor.
2. da se kodira komponentin atribut u okviru renderer's encode metode.

Prvo se dodaju styleClass, tabClass i selectedTabClass u okviru TLD fajla:

```
<taglib>
...
<tag>
...
<attribute>
<name>styleClass</name>
<description> CSS stil za ovu komponentu</description>
</attribute>
...
</tag>
</taglib>
```

Zatim se definiše atribut za CSS klase:

```
public class TabbedPaneRenderer extends Renderer {
...
public void encodeBegin(FacesContext context,
UIComponent component)
throws java.io.IOException {
ResponseWriter writer = context.getResponseWriter();
writer.startElement("table", component);
String styleClass = (String)
component.getAttributes().get("styleClass");
if (styleClass != null)
writer.writeAttribute("class", styleClass,
"styleClass");
writer.write("\n"); // to make generated HTML easier to
read
}
public void encodeChildren(FacesContext context,
UIComponent component)
throws java.io.IOException {
...
encodeTab(context, responseWriter, selectItem,
component);
...
}
...
```

```
private void encodeTab(FacesContext context,
ResponseWriter writer,
SelectItem item, UIComponent component) throws
java.io.IOException {
...
String tabText = getLocalizedTabText(component,
item.getLabel());
...
String tabClass = null;
if (content.equals(selectedContent))
tabClass = (String)
component.getAttributes().get("selectedTabClass");
else
tabClass = (String)
component.getAttributes().get("tabClass");
if (tabClass != null)
writer.writeAttribute("class", tabClass, "tabClass");
...
}
...
}
```

U sledećem primeru se kodira styleClass atribut za tabbed pane-ovu spoljnu tabelu i kodira ae tabClass i selectedTabClass atributi za svaki poseban tag.

Upotreba sakrivenih polja

Svaki tab je kodiran kao hiperlink, kao:

```
<a href="#"
onclick="document.forms[formId][clientId].value=content;
document.forms[formId].submit();"/>
```

Kada korisnik klikne na određeni hiperlink, prihvata se forma. Naravno, server mora da zna koji je tab selektovan. Informacija se nalazi u skrivenoj promenljivoj koja se smešta nakon svih tabova:

```
<input type="hidden" name="clientId"/>
```

Kada se forma prihvati, ime i vrednost sakrivenog polja su poslati serveru, dozvoljavajući da se dekodira aktviranu selektovanu tab. Renderer-ova encodeTab metoda pravi hiperlink tagove. Metoda encodeEnd pozive encodeHiddenFields(), koja kodira sakrivena polja. Kada tabbed pane

renderer dekodira dolazeći zahtev, on koristi zahtevane parametre, koji su dodeljeni sakrivenom polju, kako bi postavio sadržaj tabbed pane komponente:

```
public void decode(FacesContext context, UIComponent
component) {
Map requestParams =
context.getExternalContext().getRequestParameterMap();
String clientId = component.getClientId(context);
String content = (String) (requestParams.get(clientId));
if (content != null && !content.equals("")) {
UITabbedPane tabbedPane = (UITabbedPane) component;
tabbedPane.setContent(content);
}
...
}
...
}
```

Snimanje i postavljanje starog stanja

JSF okruženje čuva i uzima sve objekte u tekućem pogledu između zahteva, čime uključuje komponente, konvertore, validatore i osluškivače događaja. Neophodno je da se implementira stanje koje se snima, za sve napredne komponente. Kada aplikacija snimi svoje stanje na serveru, tada se objekti pogleda nalaze u memoriju, a kada se stanje snimi na klijentskoj strani, onda se objekti pogleda kodiraju u sakrivenom polju, u okviru stringa, koji izgleda slično:

```
<input type="hidden" name="javax.faces.ViewState"
id="javax.faces.ViewState"
value="rO0ABXNyACBjb20uc3VuLmZhY2VzLnV0aWwuVHJlZVN0cnVjd
HVyZRRmG0QclWAgAgAETAAI...
...4ANXBwcHBwcHBwcHBwcHBxAH4ANXEAfgA1cHBwcHQABnN1Ym1
pdHVxAH4ALAAAAAA=" />
```

Snimanje stanja na klijentskoj strani podrazumeva da korisnik isključi cookie-ije i da se poboljšava skalabilnost web aplikacije. Kada komponenta ima instance polja i kada je potrebno da se u okviru web aplikacije snimi stanje na klijentskoj strani, neophodno je da se implementiraju metode saveState i restoreState interfejsa StateHolder.

Implementacije tih metoda su:

```
public Object saveState(FacesContext context) {
Object values[] = new Object[n];
values[0] = super.saveState(context);
values[1] = instance field #1;
values[2] = instance field #2;
...
return values;
}
public void restoreState(FacesContext context, Object
state) {
Object values[] = (Object[]) state;
super.restoreState(context, values[0]);
instance field #1 = (Type) values[1];
instance field #2 = (Type) values[2];
...
}
```

Kako bi testirali zašto je potrebno snimanje stanja, izvršiće se sledeći koraci iskomentariše se saveState i restoreState metode i aktivira se klijentska strana čuvanjem stanja pri dodavanju linija u fajl web.xml:

```
<context-param>
<param-name>javax.faces.STATE_SAVING_METHOD</param-name>
<param-value>client</param-value>
</context-param>
```

Zatim se doda dugme <h:commandButton value="Test State Saving"/> u index.jsp. Pokrene se aplikacija i pritisne tab, klikne se“Test State Saving” dugme. Tekuća stranica je ponovo prikazana, ali nijedan tab nije selektovan. Ovaj problem se pojavio, jer je stranica snimnjena na klijentskoj strani i kodirana je vrednost sakrivenog polja. Kada se stranica ponovo prikazuje, novi UITabbedPane objekat je konstruisan i pozvana je njegova metoda restoreState. Ukoliko klasa UITabbedPane ne proširuje metodu restoreState, sadržaj polja neće biti učitano.
U sledećem primeru koristi se klasa

tabbedpane/src/java/com/corejsf/UITabbedPane.java

```
1. package com.corejsf;
2.
3. import javax.faces.component.UICommand;
4. import javax.faces.context.FacesContext;
```

```
5.
6. public class UITabbedPane extends UICommand {
7. private String content;
8.
9. public String getContent() { return content; }
10. public void setContent(String newValue) { content =
newValue; }
11.
12. // Comment out these two methods to see what happens
13. // when a component does not properly save its
state.
14. public Object saveState(FacesContext context) {
15. Object values[] = new Object[3];
16. values[0] = super.saveState(context);
17. values[1] = content;
18. return values;
19. }
20.
21. public void restoreState(FacesContext context,
Object state) {
22. Object values[] = (Object[]) state;
23. super.restoreState(context, values[0]);
24. content = (String) values[1];
25. }
26. }
```

Okidanje događaja akcije

Kada komponenta obrađuje akcione događaje ili akcije, onda su potrebni sledeći koraci:

- komponenta bi trebalo da proširuje klasu UICommmand.
- Neophodno je da se stavi u red ActionEvent, u okviru metode dekodiranja modula

Tabbed pane komponenta ispaljuje akcioni događaj kada god korisnik selektuje jedan od svojih tabova.

Akcija se postavlja u red TabbedPaneRenderer, u metodi dekodiranja.

```
public void decode(FacesContext context, UIComponent
component) {
...
UITabbedPane tabbedPane = (UITabbedPane) component;
...
component.queueEvent(new ActionEvent(tabbedPane));
}
```

U sledećem delu primera prikazana je implementacija klase
tabbedpane/src/java/com/corejsf/TabbedPaneRenderer.java

```
1. package com.corejsf;
2.
3. import java.io.IOException;
4. import java.util.Map;
5. import java.util.logging.Level;
6. import java.util.logging.Logger;
7. import javax.faces.component.UIComponent;
8. import javax.faces.context.ExternalContext;
9. import javax.faces.context.FacesContext;
10. import javax.faces.context.ResponseWriter;
11. import javax.faces.event.ActionEvent;
12. import javax.faces.model.SelectItem;
13. import javax.faces.render.Renderer;
14. import javax.servlet.ServletContext;
15. import javax.servlet.ServletException;
16. import javax.servlet.ServletRequest;
17. import javax.servlet.ServletResponse;
18.
19. // Renderer for the UITabbedPane component
20.
21. public class TabbedPaneRenderer extends Renderer {
22. private static Logger logger =
Logger.getLogger("com.corejsf.util");
23.
24. // By default, getRendersChildren() returns false,
so encodeChildren()
25. // won't be invoked unless we override
getRendersChildren() to return true
26.
27. public boolean getRendersChildren() {
28. return true;
29. }
30.
31. // The decode method gets the value of the request
parameter whose name
32. // is the client Id of the tabbedpane component. The
request parameter
33. // is encoded as a hidden field by
encodeHiddenField, which is called by
34. // encodeEnd. The value for the parameter is set by
JavaScript generated
35. // by the encodeTab method. It is the name of a
facet or a JSP page.
```

```
36.
37. // The decode method uses the request parameter
value to set the
38. // tabbedpane component's content attribute.
39. // Finally, decode() queues an action event that's
fired to registered
40. // listeners in the Invoke Application phase of the
JSF lifecycle. Action
41. // listeners can be specified with the
<corejsf:tabbedpane>'s actionListener
42. // attribute or with <f:actionListener> tags in the
body of the
43. // <corejsf:tabbedpane> tag.
44.
45. public void decode(FacesContext context, UIComponent
component) {
46. Map<String, String> requestParams
47. =
context.getExternalContext().getRequestParameterMap();
48. String clientId = component.getClientId(context);
49.
50. String content = (String)
(requestParams.get(clientId));
51. if (content != null && !content.equals("")) {
52. UITabbedPane tabbedPane = (UITabbedPane) component;
53. tabbedPane.setContent(content);
54. }
55.
56. component.queueEvent(new ActionEvent(component));
57. }
58.
59. // The encodeBegin method writes the starting
<table> HTML element
60. // with the CSS class specified by the
<corejsf:tabbedpane>'s styleClass
61. // attribute (if supplied)
62.
63. public void encodeBegin(FacesContext context,
UIComponent component)
64. throws java.io.IOException {
65. ResponseWriter writer = context.getResponseWriter();
66. writer.startElement("table", component);
67.
68. String styleClass = (String)
component.getAttributes().get("styleClass");
69. if (styleClass != null)
70. writer.writeAttribute("class", styleClass, null);
```

```
71.
72. writer.write("\n"); // to make generated HTML easier
to read
73. }
74.
75. // encodeChildren() is invoked by the JSF
implementation after encodeBegin().
76. // The children of the <corejsf:tabbedpane>
component are UISelectItem
77. // components, set with one or more <f:selectItem>
tags or a single
78. // <f:selectItems> tag in the body of
<corejsf:tabbedpane>
79.
80. public void encodeChildren(FacesContext context,
UIComponent component)
81. throws java.io.IOException {
82. // if the tabbedpane component has no children, this
method is still
83. // called
84. if (component.getChildCount() == 0) {
85. return;
86. }
87.
88. ResponseWriter writer = context.getResponseWriter();
89. writer.startElement("thead", component);
90. writer.startElement("tr", component);
91. writer.startElement("th", component);
92.
93. writer.startElement("table", component);
94. writer.startElement("tbody", component);
95. writer.startElement("tr", component);
96.
97. for (SelectItem item :
com.corejsf.util.Renderers.getSelectItems(component))
98. encodeTab(context, writer, item, component);
99.
100. writer.endElement("tr");
101. writer.endElement("tbody");
102. writer.endElement("table");
103.
104. writer.endElement("th");
105. writer.endElement("tr");
106. writer.endElement("thead");
107. writer.write("\n"); // to make generated HTML
easier to read
108. }
```

```
109.
110. // encodeEnd() is invoked by the JSF implementation
after encodeChildren().
111. // encodeEnd() writes the table body and encodes
the tabbedpane's content
112. // in a single table row.
113.
114. // The content for the tabbed pane can be specified
as either a URL for
115. // a JSP page or a facet name, so encodeEnd()
checks to see if it's a facet;
116. // if so, it encodes it; if not, it includes the
JSP page
117.
118. public void encodeEnd(FacesContext context,
UIComponent component)
119. throws java.io.IOException {
120. ResponseWriter writer =
context.getResponseWriter();
121. UITabbedPane tabbedPane = (UITabbedPane) component;
122. String content = tabbedPane.getContent();422
123.
124. writer.startElement("tbody", component);
125. writer.startElement("tr", component);
126. writer.startElement("td", component);
127.
128. if (content != null) {
129. UIComponent facet = component.getFacet(content);
130. if (facet != null) {
131. if (facet.isRendered()) {
132. facet.encodeBegin(context);
133. if (facet.getRendersChildren())
134. facet.encodeChildren(context);
135. facet.encodeEnd(context);
136. }
137. } else
138. includePage(context, component);
139. }
140.
141. writer.endElement("td");
142. writer.endElement("tr");
143. writer.endElement("tbody");
144.
145. // Close off the column, row, and table elements
146. writer.endElement("table");
147.
148. encodeHiddenField(context, writer, component);
```

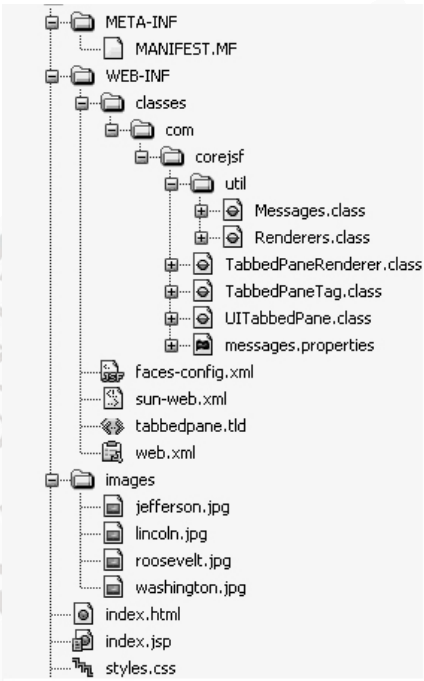
```
149. }
150.
151. // The encodeHiddenField method is called at the
end of encodeEnd().
152. // See the decode method for an explanation of the
field and its value.
153.
154. private void encodeHiddenField(FacesContext
context, ResponseWriter writer,
155. UIComponent component) throws java.io.IOException {
156. // write hidden field whose name is the
tabbedpane's client Id
157. writer.startElement("input", component);
158. writer.writeAttribute("type", "hidden", null);
159. writer.writeAttribute("name",
component.getClientId(context), null);
160. writer.endElement("input");
161. }
162.
163. // encodeTab, which is called by encodeChildren,
encodes an HTML anchor
164. // element with an onclick attribute which sets the
value of the hidden
165. // field encoded by encodeHiddenField and submits
the tabbedpane's enclosing
166. // form. See the decode method for more information
about the hidden field.
167. // encodeTab also writes out a class attribute for
each tab corresponding
168. // to either the tabClass attribute (for unselected
tabs) or the
169. // selectedTabClass attribute (for the selected
tab).
170.
171. private void encodeTab(FacesContext context,
ResponseWriter writer,
172. SelectItem item, UIComponent component) throws
java.io.IOException {
173. String tabText = getLocalizedTabText(component,
item.getLabel());
174. String content = (String) item.getValue();
175.
176. writer.startElement("td", component);
177. writer.startElement("a", component);
178. writer.writeAttribute("href", "#", "href");
179.
180. String clientId = component.getClientId(context);
```

```
181. String formId =
com.corejsf.util.Renderers.getFormId(context,
component);
182.
183. writer.writeAttribute("onclick",
184. // write value for hidden field whose name is the
tabbedpane's client Id
185.
186. "document.forms['" + formId + "'][" + clientId +
"'].value='"
187. + content + "'; " +
188.
189. // submit form in which the tabbedpane resides
190. "document.forms['" + formId + "'].submit(); ",
null);
191.
192. UITabbedPane tabbedPane = (UITabbedPane) component;
193. String selectedContent = tabbedPane.getContent();
194.
195. String tabClass = null;
196. if (content.equals(selectedContent))
197. tabClass = (String)
component.getAttributes().get("selectedTabClass");
198. else
199. tabClass = (String)
component.getAttributes().get("tabClass");
200.
201. if (tabClass != null)
202. writer.writeAttribute("class", tabClass, null);
203.
204. writer.write(tabText);
205.
206. writer.endElement("a");
207. writer.endElement("td");
208. writer.write("\n"); // to make generated HTML
easier to read
209. }
210. 424
211. // Text for the tabs in the tabbedpane component
can be specified as
212. // a key in a resource bundle, or as the actual
text that's displayed
213. // in the tab. Given that text, the
getLocalizedTabText method tries to
214. // retrieve a value from the resource bundle
specified with the
```

```
215. // <corejsf:tabbedpane>'s resourceBundle attribute.
If no value is found,
216. // getLocalizedTabText just returns the string it
was passed.
217.
218. private String getLocalizedTabText(UIComponent
tabbedPane, String key) {
219. String bundle = (String)
tabbedPane.getAttributes().get("resourceBundle");
220. String localizedText = null;
221.
222. if (bundle != null) {
223. localizedText =
com.corejsf.util.Messages.getString(bundle, key, null);
224. }
225. if (localizedText == null)
226. localizedText = key;
227. // The key parameter was not really a key in the
resource bundle,
228. // so just return the string as is
229. return localizedText;
230. }
231.
232. // includePage uses the servlet request dispatcher
to include the page
233. // corresponding to the selected tab.
234.
235. private void includePage(FacesContext fc,
UIComponent component) {
236. ExternalContext ec = fc.getExternalContext();
237. ServletContext sc = (ServletContext)
ec.getContext();
238. UITabbedPane tabbedPane = (UITabbedPane) component;
239. String content = tabbedPane.getContent();
240.
241. ServletRequest request = (ServletRequest)
ec.getRequest();
242. ServletResponse response = (ServletResponse)
ec.getResponse();
243. try {
244. sc.getRequestDispatcher(content).include(request,
response);
245. } catch (ServletException ex) {
246. logger.log(Level.WARNING, "Couldn't load page: " +
content, ex);
247. } catch (IOException ex) {
```

```
248. logger.log(Level.WARNING, "Couldn't load page: " +
content, ex);
249. }
250. }
251. }
```

U okviru datog primera će se demnostrirati upotreba tabova. Sledeća struktura direktorijuma se koristi.



Slika 54. Struktura direktorijuma

Početna stranica aplikacije je tabbedpane/web/index.jsp:

```
1. <html>
2. <%@ taglib uri="http://java.sun.com/jsf/core"
prefix="f" %>
3. <%@ taglib uri="http://java.sun.com/jsf/html"
prefix="h" %>
4. <%@ taglib uri="http://corejsf.com/tabbedpane"
prefix="corejsf" %>
5. <f:view>
6. <head>
```

```
7. <link href="styles.css" rel="stylesheet"
type="text/css"/>
8. <title><h:outputText
value="#{msgs.windowTitle}"/></title>
9. </head>
10. <body>
11. <h:form>426
12. <corejsf:tabbedPane styleClass="tabbedPane"
13. tabClass="tab"
14. selectedTabClass="selectedTab">
15. <f:facet name="jefferson">
16. <h:panelGrid columns="2">
17. <h:graphicImage value="/images/jefferson.jpg"/>
18. <h:outputText value="#{msgs.jeffersonDiscussion}"
19. styleClass="tabbedPaneContent"/>
20. </h:panelGrid>
21. </f:facet>
22. <f:facet name="roosevelt">
23. <h:panelGrid columns="2">
24. <h:graphicImage value="/images/roosevelt.jpg"/>
25. <h:outputText value="#{msgs.rooseveltDiscussion}"
26. styleClass="tabbedPaneContent"/>
27. </h:panelGrid>
28. </f:facet>
29. <f:facet name="lincoln">
30. <h:panelGrid columns="2">
31. <h:graphicImage value="/images/lincoln.jpg"/>
32. <h:outputText value="#{msgs.lincolnDiscussion}"
33. styleClass="tabbedPaneContent"/>
34. </h:panelGrid>
35. </f:facet>
36. <f:facet name="washington">
37. <h:panelGrid columns="2">
38. <h:graphicImage value="/images/washington.jpg"/>
39. <h:outputText value="#{msgs.washingtonDiscussion}"
40. styleClass="tabbedPaneContent"/>
41. </h:panelGrid>
42. </f:facet>
43.
44. <f:selectItem itemLabel="#{msgs.jeffersonTabText}"
45. itemValue="jefferson"/>
46. <f:selectItem itemLabel="#{msgs.rooseveltTabText}"
47. itemValue="roosevelt"/>
48. <f:selectItem itemLabel="#{msgs.lincolnTabText}"
49. itemValue="lincoln"/>
50. <f:selectItem itemLabel="#{msgs.washingtonTabText}"
51. itemValue="washington"/>
```

```
52. </corejsf:tabbedPane>
53. <h:commandButton value="Refresh"/>
54. </h:form>
55. </body>
56. </f:view>
57. </html>
```

Koristi se pomoćni TLD fajl tabbedpane/web/WEB-INF/tabbedpane.tld:

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <taglib xmlns="http://java.sun.com/xml/ns/javaee"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
5. http://java.sun.com/xml/ns/javaee/web-
jsptaglibrary_2_1.xsd"
6. version="2.1">
7. <description>A library containing a tabbed
pane</description>
8. <tlib-version>1.1</tlib-version>
9. <short-name>tabbedpane</short-name>
10. <uri>http://corejsf.com/tabbedpane</uri>
11. <tag>
12. <description>A tag for a tabbed pane
component</description>
13. <name>tabbedPane</name>
14. <tag-class>com.corejsf.TabbedPaneTag</tag-class>
15. <body-content>JSP</body-content>
16. <attribute>
17. <description>Component id of this
component</description>
18. <name>id</name>
19. <rtexprvalue>true</rtexprvalue>
20. </attribute>
21. <attribute>
22. <description>
23. Component reference expression for this component
24. </description>
25. <name>binding</name>
26. <deferred-value>
27. <type>javax.faces.component.UIComponent</type>
28. </deferred-value>
29. </attribute>
30. <attribute>
31. <description>
32. A flag indicating whether or not this component
should
```

```
33. be rendered. If not specified, the default value is
true.
34. </description>
35. <name>rendered</name>
36. <deferred-value>
37. <type>boolean</type>
38. </deferred-value>
39. </attribute>
40. <attribute>
41. <description>The CSS style for this
component</description>
42. <name>style</name>
43. <deferred-value>
44. <type>java.lang.String</type>
45. </deferred-value>
46. </attribute>
47. <attribute>
48. <description>The CSS class for this
component</description>
49. <name>styleClass</name>
50. <deferred-value>
51. <type>java.lang.String</type>
52. </deferred-value>
53. </attribute>
54. <attribute>
55. <description>The CSS class for unselected
tabs</description>
56. <name>tabClass</name>
57. <deferred-value>
58. <type>java.lang.String</type>
59. </deferred-value>
60. </attribute>
61. <attribute>
62. <description>The CSS class for the selected
tab</description>
63. <name>selectedTabClass</name>
64. <deferred-value>
65. <type>java.lang.String</type>
66. </deferred-value>
67. </attribute>
68. <attribute>
69. <description>
70. The resource bundle used to localize select item
labels
71. </description>
72. <name>resourceBundle</name>
73. <deferred-value>
```

```
74. <type>java.lang.String</type>
75. </deferred-value>
76. </attribute>
77. <attribute>
78. <description>
79. A method expression that's called when a tab is
selected
80. </description>
81. <name>actionListener</name>
82. <deferred-method>
83. <method-signature>
84. void actionListener(javax.faces.event.ActionEvent)
85. </method-signature>
86. </deferred-method>
87. </attribute>
88. </tag>
89. </taglib>
```

Serverska komponenta je tabbedpane/src/java/com/corejsf/TabbedPaneTag.java:

```
1. package com.corejsf;
2.
3. import javax.el.MethodExpression;
4. import javax.el.ValueExpression;
5. import javax.faces.component.ActionSource;
6. import javax.faces.component.UIComponent;
7. import
javax.faces.event.MethodExpressionActionListener;
8. import javax.faces.webapp.UIComponentELTag;
9.
10. // This tag supports the following attributes
11. //
12. // binding (supported by UIComponentELTag)
13. // id (supported by UIComponentELTag)
14. // rendered (supported by UIComponentELTag)
15. // style
16. // styleClass
17. // tabClass
18. // selectedTabClass
19. // resourceBundle
20. // actionListener
21.
22. public class TabbedPaneTag extends UIComponentELTag
{
23. private ValueExpression style;
```

```
24. private ValueExpression styleClass;
25. private ValueExpression tabClass;
26. private ValueExpression selectedTabClass;
27. private ValueExpression resourceBundle;
28. private MethodExpression actionListener;
29.
30. public String getRendererType() { return
"com.corejsf.TabbedPane"; }
31. public String getComponentType() { return
"com.corejsf.TabbedPane"; }
32.
33. public void setTabClass(ValueExpression newValue) {
tabClass = newValue; }
34. public void setSelectedTabClass(ValueExpression
newValue) {
35. selectedTabClass = newValue;
36. }
37. public void setStyle(ValueExpression newValue) {
style = newValue; }
38. public void setStyleClass(ValueExpression newValue)
{
39. styleClass = newValue;
40. }
41. public void setResourceBundle(ValueExpression
newValue) {
42. resourceBundle = newValue;
43. }
44. public void setActionListener(MethodExpression
newValue) {
45. actionListener = newValue;
46. }
47.
48. protected void setProperties(UIComponent component)
{
49. // make sure you always call the superclass
50. super.setProperties(component);
51.
52. component.setValueExpression("style", style);
53. component.setValueExpression("styleClass",
styleClass);
54. component.setValueExpression("tabClass", tabClass);
55. component.setValueExpression("selectedTabClass",
selectedTabClass);
56. component.setValueExpression("resourceBundle",
resourceBundle);
57. if (actionListener != null)
58. ((ActionSource) component).addActionListener(
```

```
59. new MethodExpressionActionListener(actionListener));
60. }
61.
62. public void release() {
63. // always call the superclass method
64. super.release();
65.
66. style = null;
67. styleClass = null;
68. tabClass = null;
69. selectedTabClass = null;
70. resourceBundle = null;
71. actionListener = null;
72. }
73. }
```

Konfiguracioni fajl je tabbedpane/web/WEB-INF/faces-config.xml

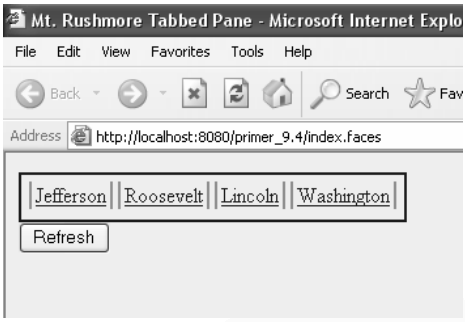
```
1. <?xml version="1.0"?>
2. <faces-config
xmlns="http://java.sun.com/xml/ns/javaee"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
5. http://java.sun.com/xml/ns/javaee/web-
facesconfig_1_2.xsd"
6. version="1.2">
7. <navigation-rule>
8. <from-view-id>/index.jsp</from-view-id>
9. <navigation-case>
10. <to-view-id>/welcome.jsp</to-view-id>
11. </navigation-case>
12. </navigation-rule>
13.
14. <component>
15. <description>A tabbed pane</description>
16. <component-type>com.corejsf.TabbedPane</component-
type>
17. <component-
class>com.corejsf.UITabbedPane</component-class>
18. </component>
19.
20. <render-kit>
21. <renderer>
22. <component-family>javax.faces.Command</component-
family>
```

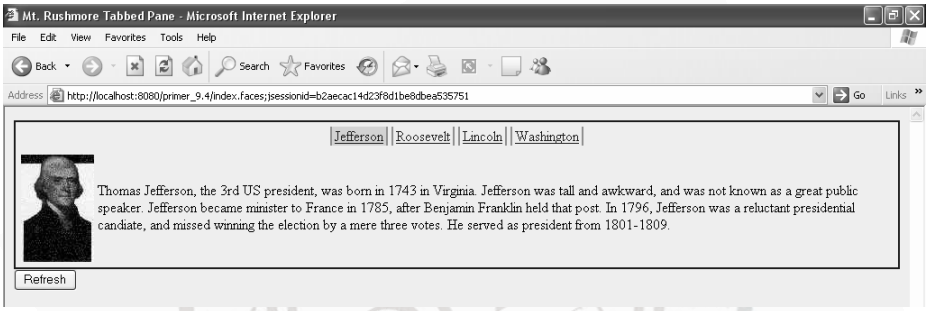
```
23. <renderer-type>com.corejsf.TabbedPane</renderer-
type>
24. <renderer-
class>com.corejsf.TabbedPaneRenderer</renderer-class>
25. </renderer>
26. </render-kit>
27.
28. <application>
29. <resource-bundle>
30. <base-name>com.corejsf.messages</base-name>
31. <var>msgs</var>
32. </resource-bundle>
33. </application>
34. </faces-config>
```

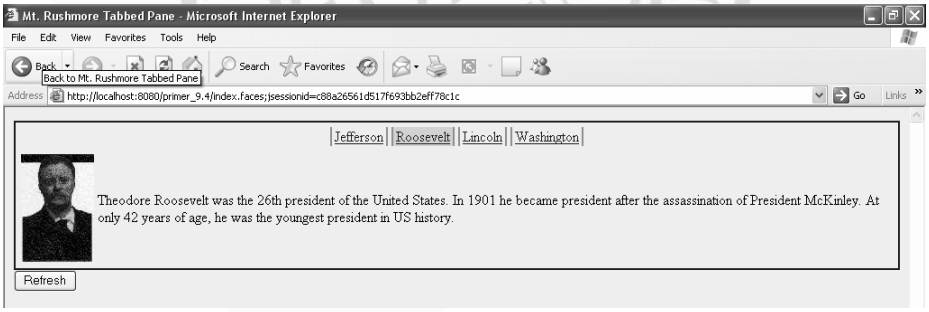
Za obrau klijentskih strana koristi se tabbedpane/web/styles.css

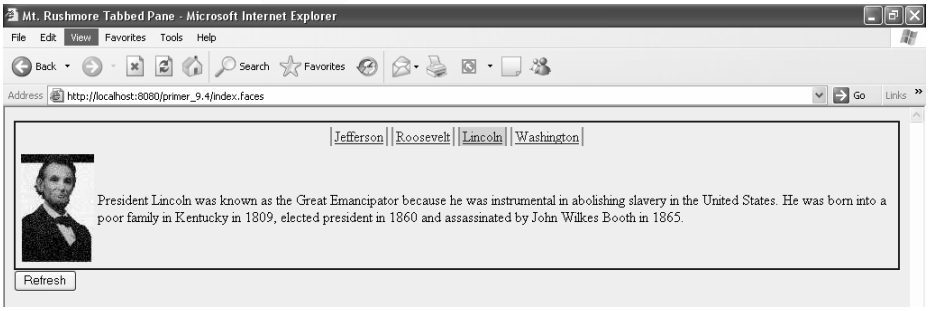
```
1. body {
2. background: #eee;
3. }
4. .tabbedPane {
5. vertical-align: top;
6. border: thin solid Blue;
7. }
8. .tab {
9. padding: 3px;
10. border: thin solid CornflowerBlue;
11. color: Blue;
12. }
13. .selectedTab {
14. padding: 3px;
15. border: thin solid CornflowerBlue;
16. color: Blue;
17. background: PowderBlue;
18. }
```

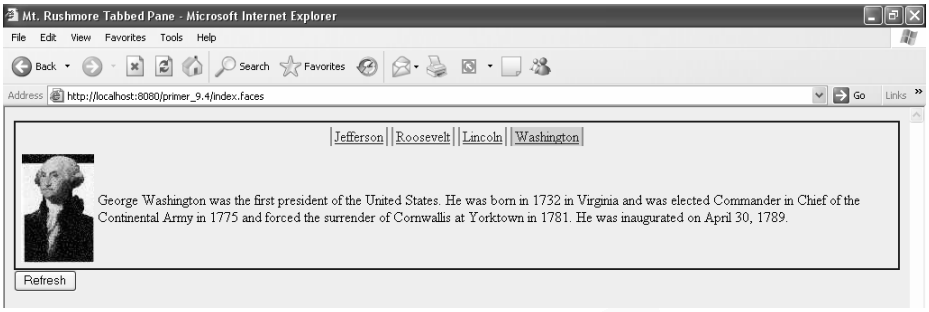
Izvršavanjem aplikacije dobijaju se sledeći ekrani:











Slika 55. Rezultati rada aplikacije

Implementacija konvertora i validatora

Do sada razmatrani konvertori i validatori nisu imali mogućnost definisanja parametara.S druge strane konvertori bi trebalo da imaju iste mogućnosti kao i stadardni f:convertNumber i f:convertDateTime tagovi.

```
<h:outputText value="#{payment.card}">
<corejsf:convertCreditcard separator="-"/>
</h:outputText>
```

Da bi pristupili navedenom tagu, potrebno je da se implementira konvertor tag. Slično postupku sa naprednim komponentama, konverter tag zahteva značajno programiranje, ali rezultat je taj da je tag upotrebiv za druge stranice.
Prvi korak je da se postave opisi ovog taga u TLD fajl. Zatim se premesti taj fajl u WEB-INF direktorijum. Sledeći kod prikazuje kako TLD fajl opisuje convertCreditcard custom tag. Fajl je nazvan custom-converter/web/WEB-INF/converter.tld

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <taglib xmlns="http://java.sun.com/xml/ns/javaee"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
5.                 http://java.sun.com/xml/ns/javaee/web-
jsptaglibrary_2_1.xsd"
6. version="2.1">
7. <tlib-version>1.1</tlib-version>
8. <tlib-version>1.1</tlib-version>
9. <short-name>converter</short-name>
```

```
10. <uri>http://corejsf.com/converter</uri>
11.
12. <tag>
13. <name>convertCreditcard</name>
14.   <tag-class>com.corejsf.CreditCardConverterTag</tag-
class>
15. <body-content>empty</body-content>
16. <attribute>
17. <name>separator</name>
18. <deferred-value>
19. <type>java.lang.String</type>
20. </deferred-value>
21. </attribute>
22. </tag>
23. </taglib>
```

Namena fajla je da se specificira ime klase za tag hendler(com.corejsf.CreditCardConverterTag) i da se dozvole atributi za tag(u našem slučaju separator). Može se primetiti da uri tag indetifikuje tag library.
Vrednost elementa sledbenika unutar definicije separator atributa označava da je atribut definisan sa vrednosnim izrazom, koji bi trebao da popuni string. Vrednost atributa mora da bude konstantan string ili string koji sadrži izraz #{...} .
TLD identifikator se referncira u taglib direktivi JSF stranice:

```
<%@ taglib uri="http://corejsf.com/converter"
prefix="corejsf" %>
```

Kada se implementira klasa tag hendlera neophodno je da se ispune sledeće stvari:

1. da se specificira konverter atributa klasa
2. da se dobiju tag atributi
3. da se konfiguriše konverter objekat, korišćenjem dobijenih atributa.

Za konverter, tag hendler klasa bi trebalo da bude podklasa klase ConverterELTag. Takođe tag hendler klasa bi trebalo da specificra setter metode za svaki atribut.

```
public class ConvertCreditCardTag extends ConverterELTag
{
private ValueExpression separator;
public void setSeparator(ValueExpression newValue) {
separator = newValue; }
...
}
```

Da bi se konfigurisala instanca konvertora sa tag atributom, potrebno je da se proširi metoda create-Converter.

```
public Converter createConverter() throws JspException {
CreditCardConverter converter = new
CreditCardConverter();
ELContext elContext =
FacesContext.getCurrentInstance().getELContext();
converter.setSeparator((String)
separator.getValue(elContext));
return converter;
}
```

Ova metoda postavlja separator property CreditCardConverter-a. Pored ove metode, potrebna je i metoda za resetovanje polja instanci, na njihove predefinisane vrednosti:

```
public void release() {
separator = null;
}
```

Kod klase
custom-converter/src/java/com/corejsf/CreditCardConverterTag.java

```
1. package com.corejsf;
2.
3. import javax.el.ELContext;
4. import javax.el.ValueExpression;
5. import javax.faces.context.FacesContext;
6. import javax.faces.convert.Converter;
7. import javax.faces.webapp.ConverterELTag;
```

```
8. import javax.servlet.jsp.JspException;
9.
10. public class CreditCardConverterTag extends
ConverterELTag {
11. private ValueExpression separator;
12.
13. public void setSeparator(ValueExpression newValue) {
14. separator = newValue;
15. }
16.
17. public Converter createConverter() throws
JspException {
18. CreditCardConverter converter = new
CreditCardConverter();
19. ELContext elContext =
FacesContext.getCurrentInstance().getELContext();
20. converter.setSeparator((String)
separator.getValue(elContext));
21. return converter;
22. }
23.
24. public void release() {
25. separator = null;
26. }
27. }
```

U primerima je korišćena klasa javax.faces.webapp.ConverterELTag, koja ima metod protected void createConverter(). Ovaj metod treba da se preklopi kako bi se kreirali konverteri i kako bi postavili se property-iji koji su specificirani tag atributima. Takođe postoji i metod void release() koji briše stanje ovog taga kako bi mogao ponovo da se koristi.
Klasa javax.el.ValueExpression sadrži metod Object getValue(ELContext context) koji Uzima tekuće vrednosti izraza vrednosti.
Klasa javax.faces.context.FacesContext ima metod ELContext getELContext() koji uzima kontekst razvijenog izraza u okviru expression language-a.
Postoje dva načina za implementiranje validatora i konvertora. Prvi, jednostavniji, je implementiranje klase validatora i konvertora od interfejsa Serializable. Tako na primer klasa koja radi sa kreditnim karticama, implementira pored interfejsa Converter i interfejs Serializable.

```
public class CreditCardConverter implements Converter,
Serializable { ... }
```

Drugi način je da se podrži default konstruktor i implementira interfejs StateHolder. Ovaj pristup je korišćen u sledećem primeru u okviru klase DateTimeConverter. U metodi saveState interfejsa StateHolder, konstruiše se serijalizovani objekat koji opisuje instance polja. U metodi restoreState, uzimaju se instance polja iz tog objekta.

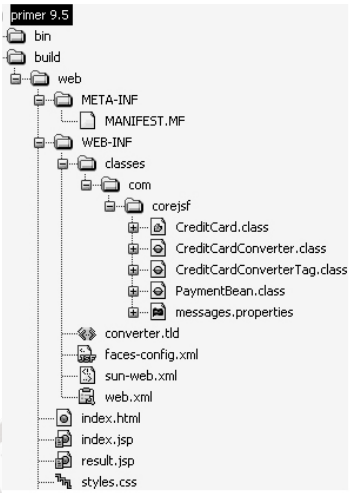
```
public class DateTimeConverter implements Converter,
StateHolder {
public Object saveState(FacesContext context) {
Object[] values = new Object[6];
values[0] = dateStyle;
values[1] = locale;
values[2] = pattern;
values[3] = timeStyle;
values[4] = timeZone;
values[5] = type;
return values;
}

public void restoreState(FacesContext context, Object
state) {
Object[] values = (Object[]) state;
dateStyle = (String) values[0];
locale = (Locale) values[1];
pattern = (String) values[2];
timeStyle = (String) values[3];
timeZone = (TimeZone) values[4];
type = (String) values[5];
}
...
}
```

StateHolder interfejs zahteva da se doda transient property. Ako je property setovan,onda objekat neće biti snimljen. Properti se koristi na analogan način kao reč transient koja je korišćena u Java serijalizaciji.

```
public class DateTimeConverter implements Converter,
StateHolder {
private boolean transientFlag; // "transient" is a
reserved word
public boolean isTransient() { return transientFlag; }
public void setTransient(boolean newValue) {
transientFlag = newValue; }
...
}
```

Za realizaciju ovog primera koristi se sledeća struktura direkotrijuma



Slika 56. Struktura direktorijuma aplikacije

U nastavku primera koji se razmatra realizovana je klasa
custom-converter/web/result.jsp

```
1. <html>
2. <%@ taglib uri="http://java.sun.com/jsf/core"
prefix="f" %>
3. <%@ taglib uri="http://java.sun.com/jsf/html"
prefix="h" %>
4. <%@ taglib uri="http://corejsf.com/converter"
prefix="corejsf" %>
5. <f:view>
6. <head>
7. <link href="styles.css" rel="stylesheet"
type="text/css"/>
8. <title><h:outputText value="#{msgs.title}"/></title>
9. </head>
10. <body>
11. <h:form>
12. <h1><h:outputText
value="#{msgs.paymentInformation}"/></h1>
13. <h:panelGrid columns="2">
14. <h:outputText value="#{msgs.amount}"/>
15. <h:outputText value="#{payment.amount}">
16. <f:convertNumber type="currency"/>
17. </h:outputText>
18.
```

```
19. <h:outputText value="#{msgs.creditCard}"/>
20. <h:outputText value="#{payment.card}">
21. <corejsf:convertCreditcard separator="-"/>
22. </h:outputText>
23.
24. <h:outputText value="#{msgs.expirationDate}"/>
25. <h:outputText value="#{payment.date}">
26. <f:convertDateTime pattern="MM/yyyy"/>
27. </h:outputText>
28. </h:panelGrid>
29. <h:commandButton value="#{msgs.back}"
action="back"/>
30. </h:form>
31. </body>
32. </f:view>
33. </html>
```

Sereverska komponenta je
custom-converter/src/java/com/corejsf/CreditCardConverter.java

```
1. package com.corejsf;
2.
3. import java.io.Serializable;
4.
5. import javax.faces.component.UIComponent;
6. import javax.faces.context.FacesContext;
7. import javax.faces.convert.Converter;
8. import javax.faces.convert.ConverterException;
9.
10. public class CreditCardConverter implements
Converter, Serializable {
11. private String separator;
12.
13. // PROPERTY: separator
14. public void setSeparator(String newValue) {
separator = newValue; }
15.
16. public Object getAsObject(
17. FacesContext context,
18. UIComponent component,
19. String newValue)
20. throws ConverterException {
21. StringBuilder builder = new StringBuilder(newValue);
22. int i = 0;
23. while (i < builder.length()) {
24. if (Character.isDigit(builder.charAt(i)))
25. i++;
```

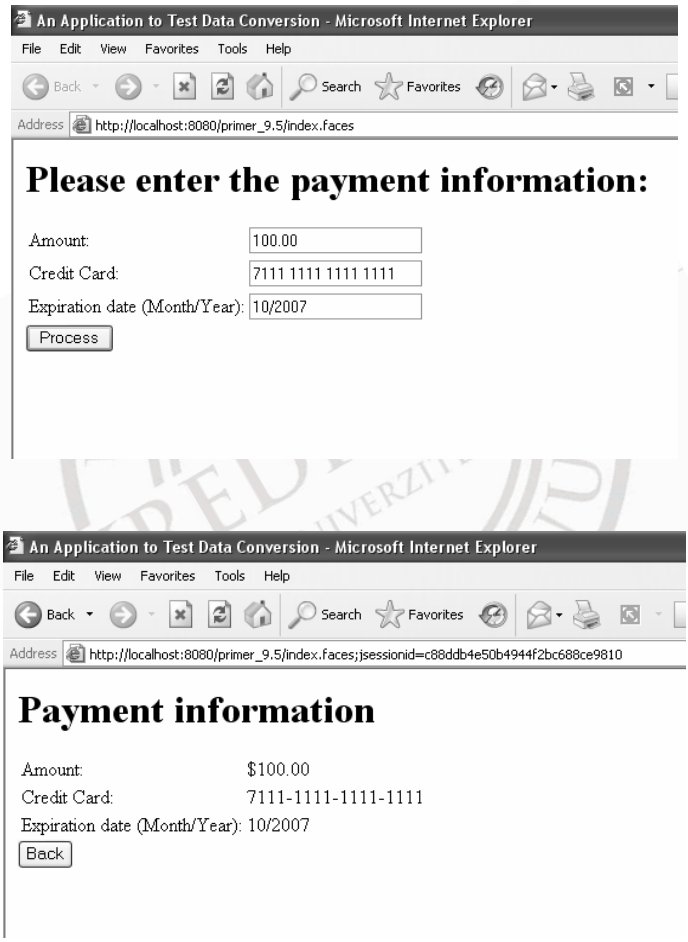
```
26. else
27. builder.deleteCharAt(i);
28. }
29. return new CreditCard(builder.toString());
30. }
31.
32. public String getAsString(
33. FacesContext context,
34. UIComponent component,
35. Object value)
36. throws ConverterException {
37. // length 13: xxxx xxx xxx xxx
38. // length 14: xxxxx xxxx xxxxx
39. // length 15: xxxx xxxxxx xxxxx
40. // length 16: xxxx xxxx xxxx xxxx
41. // length 22: xxxxxx xxxxxxxx xxxxxxxx
42. if (!(value instanceof CreditCard))
43. throw new ConverterException();
44. String v = ((CreditCard) value).toString();
45. String sep = separator;
46. if (sep == null) sep = " ";
47. int[] boundaries = null;
48. int length = v.length();
49. if (length == 13)
50. boundaries = new int[] { 4, 7, 10 };
51. else if (length == 14)
52. boundaries = new int[] { 5, 9 };
53. else if (length == 15)
54. boundaries = new int[] { 4, 10 };
55. else if (length == 16)
56. boundaries = new int[] { 4, 8, 12 };
57. else if (length == 22)
58. boundaries = new int[] { 6, 14 };
59. else
60. return v;
61. StringBuilder result = new StringBuilder();
62. int start = 0;
63. for (int i = 0; i < boundaries.length; i++) {
64. int end = boundaries[i];
65. result.append(v.substring(start, end));
66. result.append(sep);
67. start = end;
68. }
69. result.append(v.substring(start));
70. return result.toString();
71. }
72. }
```

Konfiguracioni fajl je custom-converter/web/WEB-INF/faces-config.xml

```
1. <?xml version="1.0"?>
2. <faces-config
xmlns="http://java.sun.com/xml/ns/javaee"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
5. http://java.sun.com/xml/ns/javaee/web-
facesconfig_1_2.xsd"
6. version="1.2">
7. <navigation-rule>
8. <from-view-id>/index.jsp</from-view-id>
9. <navigation-case>
10. <from-outcome>process</from-outcome>
11. <to-view-id>/result.jsp</to-view-id>
12. </navigation-case>
13. </navigation-rule>
14.
15. <navigation-rule>
16. <from-view-id>/result.jsp</from-view-id>
17. <navigation-case>
18. <from-outcome>back</from-outcome>
19. <to-view-id>/index.jsp</to-view-id>
20. </navigation-case>
21. </navigation-rule>
22.
23. <converter>
24. <converter-id>com.corejsf.CreditCard</converter-id>
25. <converter-
class>com.corejsf.CreditCardConverter</converter-class>
26. </converter>
27.
28. <converter>
29. <converter-for-
class>com.corejsf.CreditCard</converter-for-class>
30. <converter-
class>com.corejsf.CreditCardConverter</converter-class>
31. </converter>
32.
33. <managed-bean>
34. <managed-bean-name>payment</managed-bean-name>
35. <managed-bean-
class>com.corejsf.PaymentBean</managed-bean-class>
36. <managed-bean-scope>session</managed-bean-scope>
37. </managed-bean>
38.
39. <application>
```

```
40. <resource-bundle>
41. <base-name>com.corejsf.messages</base-name>
42. <var>msgs</var>
43. </resource-bundle>
44. </application>
45. </faces-config>
```

Izvršavanjem datog primera dobijaju se razultati:



Slika 57. Rezultati rada aplikacije

Tagovi validatora

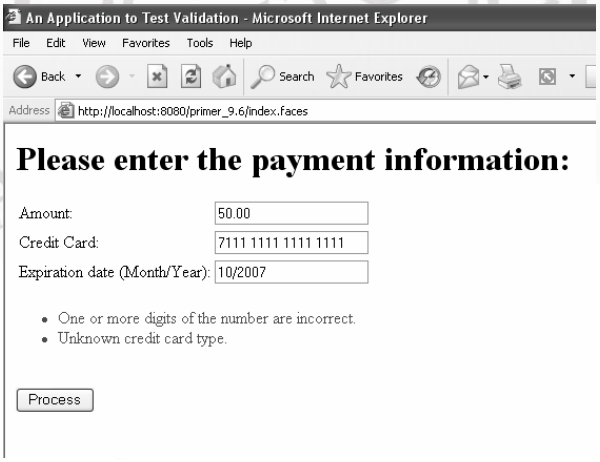
Koraci za obezbeđivanje uobičajenih validatora su skoro isti koracima za opisane konvertere.

1. Proizvodi se TLD fajl i referncira u JSF stranici.
2. Implementira tag hendler klasa koja proširuje klasu ValidatorELTag, dohvata atribute koje je oglasio TLD fajl i šalje ih objektu validatora.
3. Implementira validator klasu koja implementira inteferfejs Validator.

Podržava se validate metod na uobičajen način, kreirajući izuzetak ValidatorException, ukoliko se greška detektuje. Implementira se interfejs Serializable ili StateHolder, kako bi se snimilo i uzelo stanje validator objekta.

Kao primer može poslužiti validacija broja kreditne kartice. U tom slučaju je neophodno da se urade sledeće provere:

1. da li je korisnik uneo vrednost.
2. da li brojevi odgovaraju Luhn-ovoj formuli.
3. da li brojevi počinju sa odgovarajućim prefiksom.



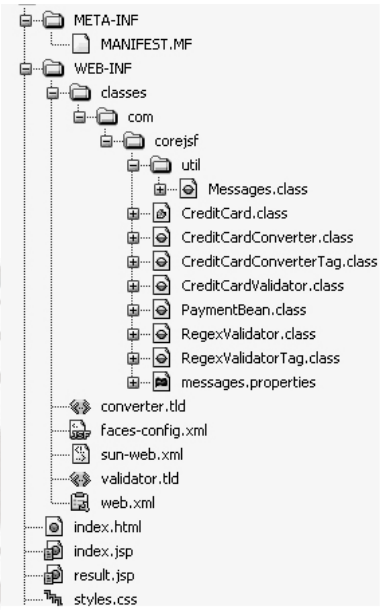
Slika 58. Prikaz ekrana primera validacije

Prefiks kreditene karitce određuje tip kartice, na primer, ako je prefiks između 51 i 55 kartica je rezervisana za MasterCard, ako je prefiks 4 onda je tipa Visa. Validator ima ulogu da izvrši validaciju regularnih izraza.

U datom primeru se koristi na sledeći način:

```
<corejsf:validateRegex expression="[3-6].*"
errorDetail="#{msgs.unknownType}"/>
```

Struktura direktorijuma ovog primera je



Slika 59. Struktura direktorijuma aplikacije

U primeru se koristi klasa javax.faces.webapp.ValidatorTag koja ima metod void setValidatorId(String id), koji postavlja ID tog validatora. ID se koristi kako bi se pretražila validator klasa. Metodom protected void createValidator() ove klase prilagođava se validator tako što se setuje propery koji je specificiran tag atributom.

Serverska komponenta je klasa
custom-validator/src/java/com/corejsf/RegexValidator.java

```
1. package com.corejsf;
2.
3. import java.io.Serializable;
4. import java.text.MessageFormat;
5. import java.util.Locale;
6. import java.util.regex.Pattern;
```

```
7. import javax.faces.application.FacesMessage;
8. import javax.faces.component.UIComponent;
9. import javax.faces.context.FacesContext;
10. import javax.faces.validator.Validator;
11. import javax.faces.validator.ValidatorException;
12.
13. public class RegexValidator implements Validator,
Serializable {
14. private String expression;
15. private Pattern pattern;
16. private String errorSummary;
17. private String errorDetail;
18.
19. public void validate(FacesContext context,
UIComponent component,
20. Object value) {
21. if (value == null) return;
22. if (pattern == null) return;
23. if(!pattern.matcher(value.toString()).matches()) {
24. Object[] params = new Object[] { expression, value
};
25. Locale locale = context.getViewRoot().getLocale();
26. FacesMessage message =
com.corejsf.util.Messages.getMessage(
27. "com.corejsf.messages", "badRegex", params);
28. if (errorSummary != null)
29. message.setSummary(
30. new MessageFormat(errorSummary,
locale).format(params));
31. if (errorDetail != null)
32. message.setDetail(
33. new MessageFormat(errorDetail,
locale).format(params));
34. throw new ValidatorException(message);
35. }
36. }
37.
38. // PROPERTY: expression
39. public void setExpression(String newValue) {
40. expression = newValue;
41. pattern = Pattern.compile(expression);
42. }445
43.
44. // PROPERTY: errorSummary
45. public void setErrorSummary(String newValue) {
46. errorSummary = newValue;
47. }
```

```
48.
49. // PROPERTY: errorDetail
50. public void setErrorDetail(String newValue) {
51. errorDetail = newValue;
52. }
53. }
```

Druga serverska komponenta je
custom-validator/src/java/com/corejsf/RegexValidatorTag.java

```
1. package com.corejsf;
2.
3. import javax.el.ELContext;
4. import javax.el.ValueExpression;
5. import javax.faces.context.FacesContext;
6. import javax.faces.validator.Validator;
7. import javax.faces.webapp.ValidatorELTag;
8. import javax.servlet.jsp.JspException;
9.
10. public class RegexValidatorTag extends
ValidatorELTag {
11. private ValueExpression expression;
12. private ValueExpression errorSummary;
13. private ValueExpression errorDetail;
14.
15. public void setExpression(ValueExpression newValue)
{
16. expression = newValue;
17. }
18.
19. public void setErrorSummary(ValueExpression
newValue) {
20. errorSummary = newValue;
21. }
22.
23. public void setErrorDetail(ValueExpression newValue)
{
24. errorDetail = newValue;
25. }
26.
27. public Validator createValidator() throws
JspException {
28. RegexValidator validator = new RegexValidator();
29. ELContext elContext =
FacesContext.getCurrentInstance().getELContext();
30.
```

```
31. validator.setExpression((String)
expression.getValue(elContext));
32. if (errorSummary != null)
33. validator.setErrorSummary((String)
errorSummary.getValue(elContext));
34. if (errorDetail != null)
35. validator.setErrorDetail((String)
errorDetail.getValue(elContext));
36.
37. return validator;
38. }
39.
40. public void release() {
41. expression = null;
42. errorSummary = null;
43. errorDetail = null;
44. }
45. }
```

Konfiguracioni fajl custom-validator/web/WEB-INF/faces-config.xml je

```
1. <?xml version="1.0"?>
2.
3. <faces-config
xmlns="http://java.sun.com/xml/ns/javaee"
4. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5. xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
6. http://java.sun.com/xml/ns/javaee/web-
facesconfig_1_2.xsd"
7. version="1.2">
8. <navigation-rule>
9. <from-view-id>/index.jsp</from-view-id>
10. <navigation-case>
11. <from-outcome>process</from-outcome>
12. <to-view-id>/result.jsp</to-view-id>
13. </navigation-case>
14. </navigation-rule>
15.
16. <navigation-rule>
17. <from-view-id>/result.jsp</from-view-id>
18. <navigation-case>
19. <from-outcome>back</from-outcome>
20. <to-view-id>/index.jsp</to-view-id>
21. </navigation-case>
22. </navigation-rule>
23.447
```

```
24. <converter>
25. <converter-id>com.corejsf.CreditCard</converter-id>
26. <converter-
class>com.corejsf.CreditCardConverter</converter-class>
27. </converter>
28.
29. <converter>
30. <converter-for-
class>com.corejsf.CreditCard</converter-for-class>
31. <converter-
class>com.corejsf.CreditCardConverter</converter-class>
32. </converter>
33.
34. <validator>
35. <validator-id>com.corejsf.CreditCard</validator-id>
36. <validator-
class>com.corejsf.CreditCardValidator</validator-class>
37. </validator>
38.
39. <validator>
40. <validator-id>com.corejsf.Regex</validator-id>
41. <validator-
class>com.corejsf.RegexValidator</validator-class>
42. </validator>
43.
44. <managed-bean>
45. <managed-bean-name>payment</managed-bean-name>
46. <managed-bean-
class>com.corejsf.PaymentBean</managed-bean-class>
47. <managed-bean-scope>session</managed-bean-scope>
48. </managed-bean>
49.
50. <application>
51. <resource-bundle>
52. <base-name>com.corejsf.messages</base-name>
53. <var>msgs</var>
54. </resource-bundle>
55. </application>
56. </faces-config>
```

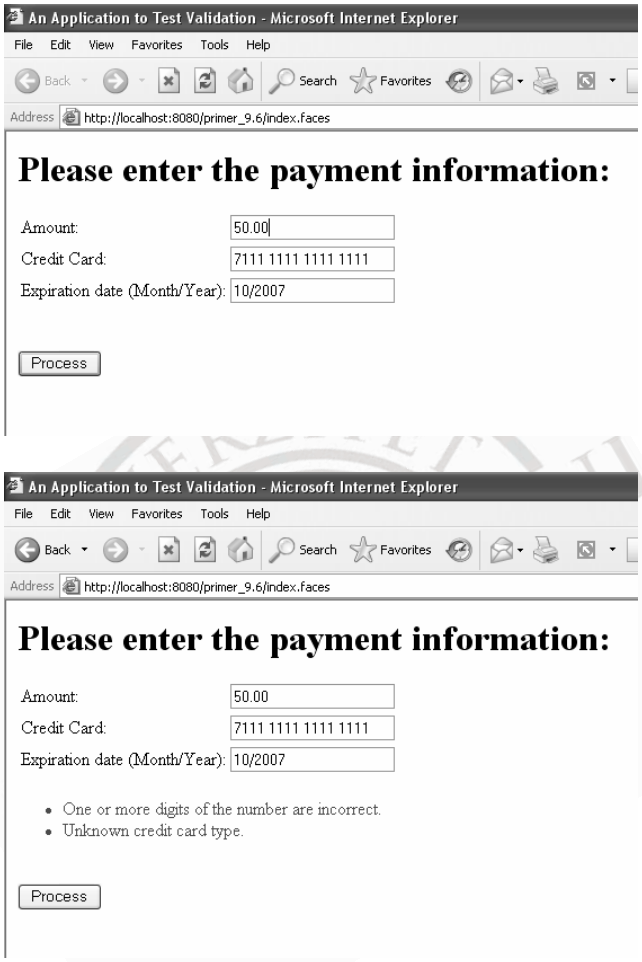
U primeru se koristi i TLD fajl
custom-validator/web/WEB-INF/validator.tld

```
1. <?xml version="1.0" encoding="UTF-8"?>
2. <taglib xmlns="http://java.sun.com/xml/ns/javaee"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
5.                 http://java.sun.com/xml/ns/javaee/web-
jsptaglibrary_2_1.xsd"
6. version="2.1">448
7. <tlib-version>1.1</tlib-version>
8. <short-name>validator</short-name>
9. <uri>http://corejsf.com/validator</uri>
10. <tag>
11. <name>validateRegex</name>
12. <tag-class>com.corejsf.RegexValidatorTag</tag-class>
13. <body-content>empty</body-content>
14. <attribute>
15. <name>expression</name>
16. <deferred-value>
17. <type>java.lang.String</type>
18. </deferred-value>
19. </attribute>
20. <attribute>
21. <name>errorSummary</name>
22. <deferred-value>
23. <type>java.lang.String</type>
24. </deferred-value>
25. </attribute>
26. <attribute>
27. <name>errorDetail</name>
28. <deferred-value>
29. <type>java.lang.String</type>
30. </deferred-value>
31. </attribute>
32. </tag>
33. </taglib>
```

Početna strana aplikacije je custom-validator/web/index.jsp

```
1. <html>
2. <%@ taglib uri="http://java.sun.com/jsf/core"
prefix="f" %>
3. <%@ taglib uri="http://java.sun.com/jsf/html"
prefix="h" %>
4. <%@ taglib uri="http://corejsf.com/validator"
prefix="corejsf" %>
5. <f:view>
6. <head>
7. <link href="styles.css" rel="stylesheet"
type="text/css"/>
8. <title><h:outputText value="#{msgs.title}"/></title>
9. </head>
10. <body>
11. <h:form>
12. <h1><h:outputText
value="#{msgs.enterPayment}"/></h1>
13. <h:panelGrid columns="2">
14. <h:outputText value="#{msgs.amount}"/>
15. <h:inputText id="amount" value="#{payment.amount}">
16. <f:convertNumber minFractionDigits="2"/>
17. </h:inputText>
18.
19. <h:outputText value="#{msgs.creditCard}"/>
20. <h:inputText id="card" value="#{payment.card}"
required="true">
21. <f:validator validatorId="com.corejsf.CreditCard"/>
22. <corejsf:validateRegex expression="[3-6].*"
23. errorDetail="#{msgs.unknownType}"/>
24. </h:inputText>
25.
26. <h:outputText value="#{msgs.expirationDate}"/>
27. <h:inputText id="date" value="#{payment.date}">
28. <f:convertDateTime pattern="MM/yyyy"/>
29. </h:inputText>
30. </h:panelGrid>
31. <h:messages styleClass="errorMessage"
32. showSummary="false" showDetail="true"/>
33. <br/>
34. <h:commandButton value="Process" action="process"/>
35. </h:form>
36. </body>
37. </f:view>
38. </html>
```

Aplikacije generišesledeće rezultate:



Slika 60. Rezultati rada aplikacije

5.19 RAD SA BAZAMA PODATAKA

U ovom poglavlju će se razmatrati kako da se ostvari konekcija na određenu bazu podataka pomoću JDBC (Java Database Connectivity) Java paketa.
Da bi mogle da se koriste SQL naredbe nad određenom bazom podataka, potrebno je prvo instancirati konekcioni objekat. Postoje različite metode konekcije. Najelegantnije rešenje je korišćenjedata source.

```
DataSource source = . . .
Connection conn = source.getConnection();
```

U narednim sekcijama opisivaće se data source u okviru GlassFish i Tomcat Web server kontejnera. Kada se jednom kreira Connection objekat, kreira se Statement objekat koji služi za slanje SQL naredbi bazi. U okviru ovog objekta koristi se executeUpdate metoda za SQL naredbe koje update-ju sadržaj baze i executeQuery metoda za upite koji vraćaju skup rezultata (u okviru objekta tipa ResultSet).

```
Statement stat = conn.createStatement();
stat.executeUpdate("INSERT INTO Users VALUES
('troosevelt', 'jabberwock')");
ResultSet result = stat.executeQuery("SELECT * FROM
Users");
```

Klasa ResultSet ima neuobičajen protokol iteracija. Prvo se poziva metoda next(), kako bi se pristupilo prvom zapisu. Zatim se poziva metoda getString() (getXXX()) kako bi se dohvatila polja koja imaju vrednost tipa String (XXX). Na primer:

```
while (result.next()) {
username = result.getString("username");
password = result.getString("password");
. . .
}
```

Kada se završi sa korišćenjem baze, obavezno mora da se zatvori konekcija.

Ceo kod, koji se odnosi na zatvranje konekcije, se stavlja u try/finally blok, kao što je:

```
Connection conn = source.getConnection();
try {
. . .
}
finally {
conn.close();
}
```

Pri otvaranju konekcije sa bazom, protekne neko vreme. Nekoliko sekundi može da protekne samo za procesiranje konekcije, autentifikaciju i zahtevanje resursa. Zbot toga, ne može se jednostavno otvoriti nova konekcija za svaki zahtev stranice. Sa druge strane, ne mogu da se drže otvorene veliki broj konekcija sa bazom. Konekcije koriste resurse, i na klijentskoj strani i na strani servera baze podataka. Baza podataka dozvoljava onoliko konekcija koliko po definiciji može da ih podrži.
Zbot toga, aplikacija ne sme da otvara konekciju prilikom logovanja korisnika i da se ostavi otvorena dok se on ne izloguje. Postoji situacija kada korisnik napusti aplikaciju, ali se nikad ne izloguje.
Jedan dobar mehanizam za rešavanje ovih problema jeste da se postave konekcije sa bazom u jedan red (pool). Konekcioni red čuva samo one konekcije koje su otvorene. Aplikacioni programi zahtevaju konekcije iz reda. Kada konekcija nije više potrebna, ona se vraća u red, ali one nisu zatvorene. Zbog toga, red minimalizuje vreme koje je potrebno za uspostavljanje konekcije sa bazom. Implementiranje reda koji sadrži konekcije sa bazom, nije jednostavna ali nije ni odgovornost aplikacionog programera. Od verzije 2.0, JDBC podržava tehniku rada sa redom na jednostavan način. Kada se primi Connection objekat iz reda, realizovan je mehanizma da se prilikom upotrebe metode close objekat u stvari vraća redu. Na aplikacionom serveru je da postavi red i da prosledi data source čija getConnection metoda postavlja polja konekcije koja radi sa redom.

```
DataSource source = ...
Connection conn = source.getConnection();
Statement stat = conn.createStatement();
String command = "INSERT INTO Users VALUES
('troosevelt', 'jabberwock')";
stat.executeUpdate(command);
conn.close();
```

U ovom kodu postoji velika greška. Ako metoda generiše izuzetak, metoda close nikada neće biti pozvana! Zbog toga, postavlja se metoda close u finally blok:

```
DataSource source = ...
Connection conn = source.getConnection();
try {
Statement stat = conn.createStatement();
String command = "INSERT INTO Users VALUES
('troosevelt', 'jabberwock')";
stat.executeUpdate(command);
}
finally {
conn.close();
}
```

Ne preporučuje se da se u try/finally konstrukciju stavi i kod koji obrađuje izuzetke.
I u sledećem kodu postoje greške. Prva, ako se pozove metoda getConnection i ona generiše izuzetak, tada je conn još uvek null vrednost i ne može da se pozove metoda close. Pozivanje metoda close može da izazove generisanje izuzetka SQLException.

```
Connection conn = null;
try {
conn = source.getConnection();
Statement stat = conn.createStatement();
String command = "INSERT INTO Users VALUES
('troosevelt', 'jabberwock')";
stat.executeUpdate(command);
}
catch (SQLException) {
// obrada greske
}
finally {
conn.close(); // GRESKA
}
```

Rešenje je da se prave dva odvojena try bloka:

```
try {
Connection conn = source.getConnection();
                   try {
                  Statement stat =
conn.createStatement();
                  String command = "INSERT INTO Users
VALUES ('troosevelt', 'jabberwock')";
                   stat.executeUpdate(command);
                           }

                  finally {
                   conn.close();
                                }

      }
catch (SQLException) {
// obrada greske
     }
```

Unutrašnji try blok osigurava da je konekcija zatvorena. Spoljašnji try blok osigurava da je izuzetak logovan.
Uobičajena optimizaciona tehnika za JDBC programe je korišćenje klase PreparedStatement. Na ovaj način se ubrzavaju operacije sa bazom, ako kod daje isti tip upita biše puta. Na primer

```
SELECT password FROM Users WHERE username=...
```

Prepared statement pita bazu podataka da prekompajlira upit, ova operacija odrazumeva parsiranje SQL naredbe i izračuavanje strategije upita. prepared statement se kreira sa metodom prepareStatement klase Connection, pri čemu se karakter ? koristi za svaki parametar.

```
PreparedStatement stat = conn.prepareStatement(
"SELECT password FROM Users WHERE username=?");
```

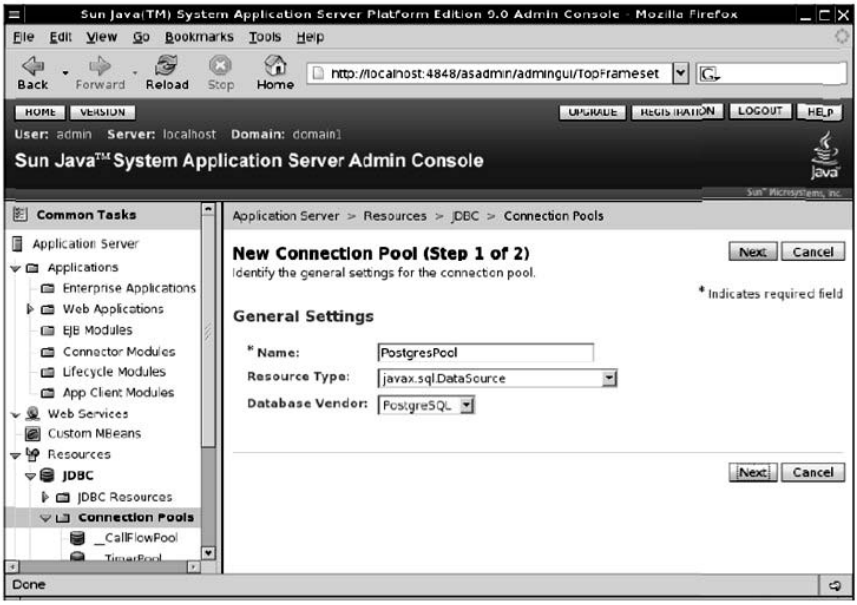
Kada se želi da se koristi prepared statement, prvo mora da se podesi vrednost parametra.

```
stat.setString(1, name);
```

Zatim se koristi prihvatanje rezultata na standardan način:

```
ResultSet result = stat.executeQuery();
```

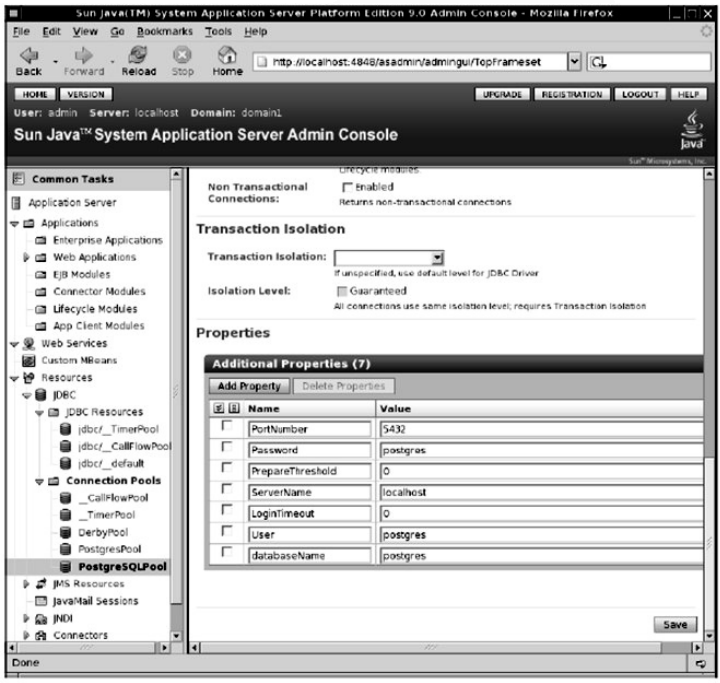
Ako se fizička konekcija sa bazom nalazi u redu, tada je dobra šansa da će se i dalje prepared statement koristiti za preuzimanje konekcije. Mnoge implementacije reda sa konekcijama će keširati prepared statement-e.
Kada se pozove prepareStatement, red će prvo da tražiti unutar keša naredbi, koristeći string upita kao ključ. Ako je prepared statement pronađena, tada će se ponovo koristiti. U suprotnom, novi prepared statement će se kreirati i dodati u keš.
U narednom delu teksta će se razmatrati kako da se konfiguriše data source u aplikacionom serveru, kao što su GlassFish i Tomcat, i kako da se pristupi data source iz web aplikacije.



Slika 61. Postavljanje parametara u okviru servera

Konfigurisanje parametara baze podataka u okviru servera GlassFish

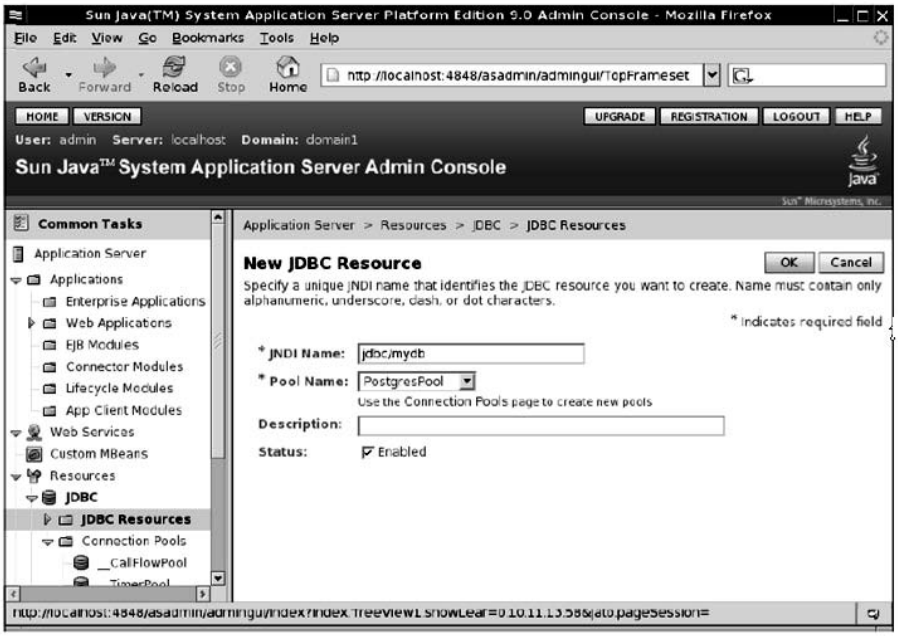
GlassFish ima web-baziran administratorski interfejs koji može da se koristi za konfiguraciju data source. Potrebno je da se unese u browser http://localhost:4848 i izvrši logovanje (po definiciji korisničko ime je admin i šifra je adminadmin). Prvo, se konfiguriše red podataka. Izabere se "Connection Pools" i postavi novi red. Imenuje se red, selektuje se tip resursa (javax.sql.DataSource) i izabere odgovarajuća baza podataka.
Na slici je prikazano kako se konfiguriše konekcija sa bazom i njene opcije kao što su username i password.



Slika 62. Postavljanje parametara u okviru servera

Na sledećoj slici je prikazano kako se konfiguriše novi data source. Treba definisati ime jdbc/mydb i selektuje se bazen koji je upravo definisan. Na kraju, dodaje se fajl koji predstavlja drajver baze podataka (kao što je postgresql-8.2.jdbc3.jar za PostgreSQL bazu podataka) unutar

domains/domain1/lib/ext poddirektorijuma GlassFish instalacije.



Slika 63. Podešavanje drajvera

Konfigurisanje parametara baze podataka u okviru servera Tomcat

U ovom odeljku će se razmatrati konfigurisanje resursa baze podataka u okviru Tomcat 5 kontejnera. Potrebno je locirajti conf/server.xml fajl i pretražiti elementi koji opisuji host koji će sadržati web aplikaciju, kao što je:

```
<!-- Define the default virtual host -->
<Host name="localhost" debug="0" appBase="webapps"
unpackWARs="false" autoDeploy="true">
...
</Host>
```

Unutar ovog elementa treba postaviti DefaultContext element koji specificira detalje baze (driver, URL, username i password) i željene karakteristike reda. U sledećem kodu prikazano je kako se specificira konekcija bazena na PostgreSQL bazu podataka.

```
<DefaultContext>
<Resource name="jdbc/mydb" auth="Container"
type="javax.sql.DataSource"/>
<ResourceParams name="jdbc/mydb">
<parameter>
<name>factory</name>
<value>org.apache.commons.dbcp.BasicDataSourceFactory</v
alue>
</parameter>
<parameter>
<name>driverClassName</name>
<value>org.postgresql.Driver</value>
</parameter>
<parameter>
<name>url</name>
<value>jdbc:postgresql://127.0.0.1:5432/postgres</value>
</parameter>
<parameter>
<name>username</name>
<value>dbuser</value>
</parameter>
<parameter>
<name>password</name>
<value>dbpassword</value>
</parameter>
<parameter>
<name>maxActive</name>
```

```
<value>20</value>
</parameter>
<parameter>
<name>maxIdle</name>
<value>10</value>
</parameter>
<parameter>
<name>poolPreparedStatements</name>
<value>true</value>
</parameter>
</ResourceParams>
</DefaultContext>
```

Da bi konfigurisali red,potrebno je da se specificira sekvenca parametara.Kompletan opis svih validnih parametara mogu se naći na sledećoj stranici:

http://jakarta.apache.org/commons/dbcp/configuration.html

Ime parametra	Opis
driverClassName	Ime JDBC drajvera, na primer org.postgresql.Driver
url	URL baze podataka, na primer jdbc:postgresql:mydb
Username	Korisničko ime baze podataka
Password	Šifra korisnika baze podataka
maxActive	Maksimalan broj paralelnih aktivnih konekcija, ili nula ako nema limita
maxIdle	Maksimalan broj aktivnih konekcija, koje mogu da se pauziraju u redu, bez dodavanja novih konekcija
poolPreparedStatements	true ako su prepared statements u redu (po definiciji: false)

Java EE specifikacija zahteva da resursi budu deklarisani u okviru web.xml fajla:

```
<resource-ref>
<res-ref-name>jdbc/mydb</res-ref-name>
<res-type>javax.sql.DataSource</res-type>
<res-auth>Container</res-auth>
</resource-ref>
```

Navedeno je da je ime resursa jdbc/mydb. To ime se koristi kako bi se pretražio data source. Postoje dva načina za pretraživanje, elegantniji je resource injection. Deklariše se polje u okviru a managed bean i označi se kao komentar:

```
@Resource(name="jdbc/mydb")
private DataSource source;
```

Kada aplikacioni server učita managed bean, polje je automatski inicijalizovano. Ova osobina se izvršava samo kod aplikacionog servera koji podržava standard Java EE 5. Ukoliko se koristi JSF 1.1 ili neki drugi Tomcat server, mora da se napravi JNDI (Java Naming and Directory Interface):

```
try {
InitialContext ctx = new InitialContext();
source = (DataSource)
ctx.lookup("java:comp/env/jdbc/mydb");
}
catch (NamingException ex) {
. . .
}
```

U sledećem primeru se pokazuje kako se proverava username/password kombinacija. Prvi ekran je kreiran za logovanje.



Slika 64. Ekran za logovanje

Ako je username/password kombinacija korektna, onda će se korisniku prikazati pozdravni ekran.



Slika 65. Ekran nakon uspešnog logovanja

U suprotnom se koristi novi pokušaj.



Slika 66. Ekran nakon neuspešnog logovanja

Konačno, ako se u bazi pojavila greška prikazaće se korisniku



Slika 67. Ekran nakon greške

Navigaciona pravila u okviru fajla faces-config.xml, koriste loginAction i logoutAction property-ije UserBean klase. U primeru kod za bazu je direktno dodat u UserBean klasu. Moguće je da se obezbedi dva sloja objekta: binovi za komunikaciju sa JSF stranicama i podaci koji pristupaju objektima koji predstavljaju entitete u bazama podataka.

Kod za pristup bazi je postavljen u odvojenoj metodi:

```
public void doLogin() throws SQLException
```

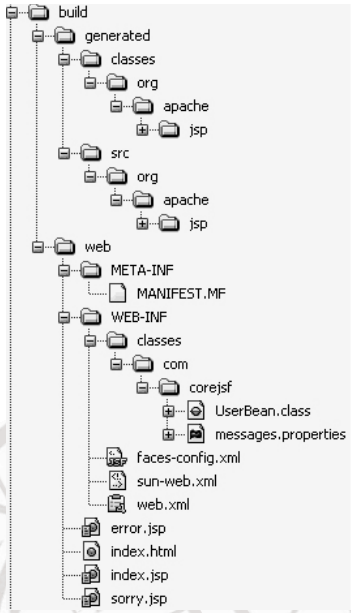
Navedena metoda formira upite ka bazi podataka za username/password kombinaciju i postavlja polje loggedIn na true ako su username i password korektni. Dugme na index.jsp stranici poziva login metodu UserBean-a. Upravo u okviru u ove metode poziva se doLogin metodu i vraća se rezultat u formi Stringa za obradu navigacije. Metoda login sarađuje sa izuzecima koje doLogin metoda obaveštava. Podrazumeva se da je doLogin metoda fokusirana na bazu podataka, a ne na korisnički interfejs. Ako se desi izuzetak, doLogin bi trebalo da obavesti i da ne preduzima nikakve dalje akcije. Metoda login, sa druge strane, privata izuzetke i vraća kao rezultat String "internalError" delu za realizaciju navigacije.

```
public String login() {
try {
doLogin();
}
catch (SQLException ex) {
logger.log(Level.SEVERE, "loginAction", ex);
return "internalError";
}
if (loggedIn)
return "loginSuccess";
else
return "loginFailure";
}
```

Pre pokretanja primera, neophodno je da se startuje baza i kreira tabela koja se zove Users i doda jedan ili više username/password ulaza:

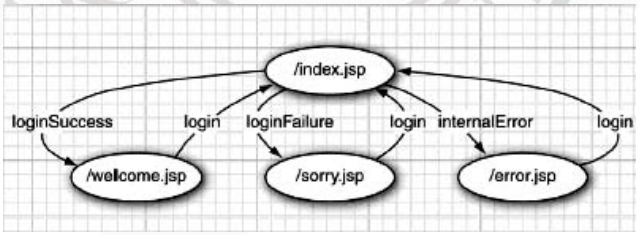
```
CREATE TABLE Users (username CHAR(20), password
CHAR(20))
INSERT INTO Users VALUES ('troosevelt', 'jabberwock')
```

Sledeća struktura direktorijuma se koristi u primeru.



Slika 68. Struktura direktorijuma aplikacije

Na sledećoj slici se prikazuje i mapa navigacije ove aplikacije.



Slika 69. Mapa navigacije aplikacije

Kao što se sa slike može videti početna stranica cele aplikacije je db/web/index.jsp

```
1. <html>
2. <%@ taglib uri="http://java.sun.com/jsf/core"
prefix="f" %>
3. <%@ taglib uri="http://java.sun.com/jsf/html"
prefix="h" %>
4. <f:view>
5. <head>
6. <title><h:outputText value="#{msgs.title}"/></title>
7. </head>
8. <body>
9. <h:form>
10. <h1><h:outputText
value="#{msgs.enterNameAndPassword}"/></h1>
11. <h:panelGrid columns="2">
12. <h:outputText value="#{msgs.name}"/>
13. <h:inputText value="#{user.name}"/>
14.
15. <h:outputText value="#{msgs.password}"/>
16. <h:inputSecret value="#{user.password}"/>
17. </h:panelGrid>
18. <h:commandButton value="#{msgs.login}"
action="#{user.login}"/>
19. </h:form>
20. </body>
21. </f:view>
22. </html>
```

U slučaju uspešnog logovanja korisniku se prikazuje stranica db/web/welcome.jsp

```
1. <html>
2. <%@ taglib uri="http://java.sun.com/jsf/core"
prefix="f" %>
3. <%@ taglib uri="http://java.sun.com/jsf/html"
prefix="h" %>
4. <f:view>
5. <head>
6. <title><h:outputText value="#{msgs.title}"/></title>
7. </head>
8. <body>
9. <h:form>
10. <p>
11. <h:outputText value="#{msgs.welcome}"/>
```

```
12. <h:outputText value="#{user.name}"/>!
13. </p>
14. <p>
15. <h:commandButton value="#{msgs.logout}"
action="#{user.logout}"/>
16. </p>
17. </h:form>
18. </body>
19. </f:view>
20. </html>
```

U slučaju neuspešnog logovanja korisniku se prikazuje stranica db/web/sorry.jsp

```
1. <html>
2. <%@ taglib uri="http://java.sun.com/jsf/core"
prefix="f" %>
3. <%@ taglib uri="http://java.sun.com/jsf/html"
prefix="h" %>
4. <f:view>
5. <head>
6. <title><h:outputText value="#{msgs.title}"/></title>
7. </head>
8. <body>
9. <h:form>
10. <h1><h:outputText value="#{msgs.authError}"/></h1>
11. <p>
12. <h:outputText value="#{msgs.authError_detail}"/>!
13. </p>
14. <p>
15. <h:commandButton value="#{msgs.continue}"
action="login"/>
16. </p>
17. </h:form>
18. </body>
19. </f:view>
20. </html>
```

U slučaju interne greške (na primer nemogućnost povezivanja sa bazom podataka) korisniku se prikazuje stranica db/web/error.jsp

```
1. <html>
2. <%@ taglib uri="http://java.sun.com/jsf/core"
prefix="f" %>
3. <%@ taglib uri="http://java.sun.com/jsf/html"
prefix="h" %>
4. <f:view>
5. <head>
6. <title><h:outputText value="#{msgs.title}"/></title>
7. </head>
8. <body>
9. <h:form>
10. <h1><h:outputText
value="#{msgs.internalError}"/></h1>
11. <p><h:outputText
value="#{msgs.internalError_detail}"/></p>
12. <p>
13. <h:commandButton value="#{msgs.continue}"
action="login"/>
14. </p>
15. </h:form>
16. </body>
17. </f:view>
18. </html>
```

Konfiguracioni fajl je db/web/WEB-INF/faces-config.xml

```
1. <?xml version="1.0"?>
2. <faces-config
xmlns="http://java.sun.com/xml/ns/javaee"
3. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4. xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
5. http://java.sun.com/xml/ns/javaee/web-
facesconfig_1_2.xsd"
6. version="1.2">
7. <navigation-rule>
8. <from-view-id>/index.jsp</from-view-id>
9. <navigation-case>
10. <from-outcome>loginSuccess</from-outcome>
11. <to-view-id>/welcome.jsp</to-view-id>
12. </navigation-case>
13. <navigation-case>
14. <from-outcome>loginFailure</from-outcome>
```

```
15. <to-view-id>/sorry.jsp</to-view-id>
16. </navigation-case>
17. <navigation-case>
18. <from-outcome>internalError</from-outcome>
19. <to-view-id>/error.jsp</to-view-id>
20. </navigation-case>
21. </navigation-rule>
22. <navigation-rule>
23. <from-view-id>/welcome.jsp</from-view-id>
24. <navigation-case>
25. <from-outcome>login</from-outcome>
26. <to-view-id>/index.jsp</to-view-id>
27. </navigation-case>
28. </navigation-rule>
29. <navigation-rule>
30. <from-view-id>/sorry.jsp</from-view-id>
31. <navigation-case>
32. <from-outcome>login</from-outcome>
33. <to-view-id>/index.jsp</to-view-id>
34. </navigation-case>
35. </navigation-rule>
36. <navigation-rule>
37. <from-view-id>/error.jsp</from-view-id>
38. <navigation-case>
39. <from-outcome>login</from-outcome>
40. <to-view-id>/index.jsp</to-view-id>
41. </navigation-case>
42. </navigation-rule>
43.
44. <managed-bean>
45. <managed-bean-name>user</managed-bean-name>
46. <managed-bean-class>com.corejsf.UserBean</managed-
bean-class>
47. <managed-bean-scope>session</managed-bean-scope>
48. </managed-bean>
49.
50. <application>
51. <resource-bundle>
52. <base-name>com.corejsf.messages</base-name>
53. <var>msgs</var>
54. </resource-bundle>
55. </application>
56. </faces-config>
```

Konekcija sa bazom podataka i provera unetih podataka se obavlja u okviru klase db/src/java/com/corejsf/UserBean.java

```
1. package com.corejsf;
2.
3. import java.sql.Connection;
4. import java.sql.PreparedStatement;
5. import java.sql.ResultSet;
6. import java.sql.SQLException;
7. import java.util.logging.Level;
8. import java.util.logging.Logger;
9.
10. import javax.annotation.Resource;
11. import javax.sql.DataSource;
12.
13. public class UserBean {
14. private String name;
15. private String password;
16. private boolean loggedIn;
17. private Logger logger =
Logger.getLogger("com.corejsf");
18.
19. @Resource(name="jdbc/mydb")
20. private DataSource ds;
21.
22. /*
23. Ako se koristi Tomcat ili JSF 1.1, treba ukloniti
@Resource liniju i dodati ovaj konstruktor:
24. public UserBean()
25. {
26. try {
27. Context ctx = new InitialContext();
28. ds = (DataSource)
ctx.lookup("java:comp/env/jdbc/mydb");
29. } catch (NamingException ex) {
30. logger.log(Level.SEVERE, "DataSource lookup failed",
ex);
31. }
32. }
33. */
34.
35. public String getName() { return name; }
36. public void setName(String newValue) { name =
newValue; }
37.
38. public String getPassword() { return password; }
39. public void setPassword(String newValue) { password
= newValue; }
```

```
40.
41. public String login() {
42. try {
43. doLogin();
44. }
45. catch (SQLException ex) {
46. logger.log(Level.SEVERE, "login failed", ex);
47. return "internalError";
48. }
49. if (loggedIn)
50. return "loginSuccess";
51. else
52. return "loginFailure";
53. }
54.
55. public String logout() {
56. loggedIn = false;
57. return "login";
58. }
59.
60. public void doLogin() throws SQLException {
61. if (ds == null) throw new SQLException("No data
source");
62. Connection conn = ds.getConnection();
63. if (conn == null) throw new SQLException("No
connection");
64.
65. try {
66. PreparedStatement passwordQuery =
conn.prepareStatement(
67. "SELECT password from Users WHERE username = ?");
68.
69. passwordQuery.setString(1, name);
70.
71. ResultSet result = passwordQuery.executeQuery();
72.
73. if (!result.next()) return;
74. String storedPassword =
result.getString("password");
75. loggedIn = password.equals(storedPassword.trim());
76. }
77. finally {
78. conn.close();
79. }
80. }
81. }
```

LITERATURA

1. http://www.ecommerce.gov/emerging.htm
2. „Globalni građani“, Centar za proučavanje informacionih tehnologija (CEPIT) Beogradske otvorene škole, Beograd, 2003.
3. „Core JavaServer Faces“, Second Edition, David Geary, Cay Horstmann, Prentice Hall, USA, 2007
4. „Programiranje grafičkih aplikacija“, Boško Nikolić, Univerzitet Singidunuma, 2005
5. „Core J2EE Patterns: Best Practices and Design Strategies“, Second Edition, Deepak Alur, John Crupi, Dan Malks, Prentice Hall, USA, 2003
6. „Core JSP“, D. Hougland, A. Tavistock, Prentice Hall, USA, 2000
7. „Professional Web 2.0 Programming“, E. Vlist, A. Vernet, E. Bruchez, J. Fawcett, D. Ayers, Wiley Publishing, USA, 2007
8. „JSP: A Beginner’s Guide“, G. Bollinger, B. Natarajan, McGrow Hill, USA, 2001
9. „Pro JSF and AJAX“, J. Jacobi, J. Fallows, Apress, USA, 2006
10. „Core Servlets and JavaServer Pages Volume 1: Core Technologies“, Second Edition, M. Hall, L. Brown, Prentice Hall, USA, 2003
11. „More Servlets and JavaServer Pages„ M. Hall, Pearson Education; USA, 2001
12. „Java Servlet Programming“, J. Hunter, O'Reilly, 2001
13. „Core JSTL: Mastering the JSP Standard Tag Library“, D. Geary, Prentice Hall, USA, 2002
14. „Web Development with JavaServer Pages“, D. Fields, M. Kolb, S. Bayern, Manning Publications, 2001