

ELEKTROTEHNIČKI
FAKULTET
BEOGRAD

LABORATORIJSKI SISTEM ZASNOVAN NA AJAX JSF TEHNOLOGIJI

mentor: dr Nikolić Boško

student: Šerić Ivan 01/011

Beograd Oktobar, 2008.

Sadržaj		strana
1	<u>Uvod</u>	3
2	<u>Uvod u Ajax i JSF tehnologije</u>	4
	2.1 <u>Ajax</u>	4
	2.1.1 <u>Istorija AJAX-a</u>	4
	2.1.2 <u>Razlike između tradicionalnih i AJAX internet aplikacija</u>	5
	2.1.3 <u>Načini upotrebe AJAX tehnologije</u>	6
	2.2 <u>JSF</u>	7
	2.2.1 <u>Istorija JSF-a</u>	7
	2.2.2 <u>Šta je JSF ?</u>	8
	2.2.3 <u>Navigacioni model</u>	9
	2.2.4 <u>JSF arhitektura</u>	9
	2.2.4.1 <u>Model</u>	10
	2.2.4.2 <u>View</u>	11
	2.2.4.3 <u>Controller</u>	12
	2.2.5 <u>JSF životni ciklus</u>	12
	2.2.6 <u>Dodatne klase(konverteri,validatori listeneri)</u>	13
3	<u>Opis sistema sa korisničke strane</u>	15
	3.1 <u>Početni meni</u>	15
	3.2 <u>Korišćenje Ajax-a za prikaz datuma i vremena</u>	15
	3.3 <u>Korišćenje Ajax-a za popunjavanje forme</u>	16
	3.4 <u>Korišćenje Ajax-a i JSF-a za realtime validaciju</u>	17
	3.5 <u>JSF-Rico komponenta</u>	18
	3.6 <u>Korišćenje Ajax4JSF framework-a za popunjavanje forme</u>	20
	3.7 <u>Korišćenje Ajax4JSF framework-a za realtime validaciju</u>	20
	3.8 <u>Ajax4JSF Eho</u>	21
4	<u>Implementacija sistema</u>	21
	4.1 <u>Radno okruženje</u>	21
	4.2 <u>Primeri</u>	24
	4.2.1 <u>Korišćenje Ajax-a za prikaz datuma i vremena</u>	24
	4.2.2 <u>Korišćenje Ajax-a za popunjavanje forme</u>	28
	4.2.3 <u>Korišćenje Ajax-a i JSF-a za realtime validaciju</u>	31
	4.2.4 <u>JSF-Rico komponenta</u>	36
	4.2.4.1 <u>Spinner</u>	36
	4.2.4.2 <u>Kreiranje Tag klase za spinner</u>	37
	4.2.4.3 <u>Kreiranje UISpinner klase</u>	41
	4.2.4.4 <u>Kreiranje Accordion Komponente</u>	44
	4.2.5 <u>Korišćenje Ajax4JSF framework-a za popunjavanje forme</u>	50
	4.2.6 <u>Korišćenje Ajax4JSF framework-a za realtime validaciju</u>	52
	4.2.7 <u>Ajax4JSF Eho</u>	55
	4.2.8 <u>Početni meni</u>	57
5	<u>Zaključak</u>	58
	<u>Literatura</u>	59

1 Uvod <<

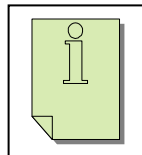
U ovom radu kroz set labaratorijskih vežbi prikazaćemo i objasniti neke od mogućnosti koje nam nude JSF i AJAX tehnologije. Cilj je da se kroz osam labaratorijskih vežbi uključujući i meni kao jednu vežbu studenti postepeno uvedu u ove tehnologije i njihove mogućnosti. Svaki primer je detaljno objašnjen, kako vizuelno (treće poglavlje) tako i kroz kod (četvrto poglavlje). Primeri treba da pokažu studentima upotrebu čistog AJAX-a, upotrebu AJAX-a uz pomoć JavaScript biblioteka i dodavanje AJAX funkcionalnosti JSF komponentama uz pomoć AJAX4JSF framework-a. Takođe pokazaćemo kako se prave custom JSF komponente i kako ostvariti različite interakcije između AJAX-a i JSF-a.

Prvo ćemo u poglavlju dva ukratko objasniti same JSF i AJAX tehnologije i kako njihovim korišćenjem možemo da napravimo jednu bogatu aplikaciju koja omuagaćava veliku interakciju korisnika sa aplikacijom kao i bogat GUI. Cilj je da korišćenjem naprednih tehnologija napravimo aplikaciju koja se po izgledu i performansama približava standardnim desktop aplikacijama.

U trećem poglavlju dat je prikaz sistema sa korisničke strane. Kroz niz slika prikazujemo funkcionalnost sistema-objašnjavamo koje će korisničke akcije generisati AJAX zahtev i određene promene na ekranu. Takođe pokazujemo i kako izgledaju custom JSF komponente.

Četvrto poglavlje nudi objašnjenje o načinu na koji je sistem implementiran. Prikazani su i objašnjeni najvažniji delovi koda koji su sistemu omogućili specifične funkcionalnosti.

U poslednjem poglavlju izloženi su zaključci o korišćenim tehnologijama kao i predlozi za unapređenje sistema. Napomene smo stavljali na mestima gde su se u toku izra de diplomskog rada javljali problemi ili na mestima gde je neki kod mogao da se napiše na više načina. Za napomene koristimo sliku



Ovaj rad je prilagođen elektronskom načinu korišćenja u kojem na veoma lak način preko linkova skaćemo sa teme na temu i nazad. Znak << uvek vraća za jedan nivo više u hijerarhiji poglavlja i paragrafa.

2 Uvod u Ajax i JSF tehnologije <<

2.1 Ajax

2.2 JSF

Početkom 90-ih godina prošlog veka dolazi do masovnog širenja interneta. Npr. 1989 –te broj korisnika interneta širom sveta je bio oko 100 000 a početkom 21-og veka taj broj je dorastao do 150 000 000. U ovom razdoblju menja se i ideja o tome kakva internet aplikacija treba da bude. Prve internet aplikacije prosleđivale su klijentu statičan sadržaj dok se sva logika obavljala na serveru. Aplikacije su podržavale standardan HTML grafički korisnički interfejs(GUI). Najveći nedostatak aplikacija je bio u tome što je svaka korisnička akcija morala da prođe kroz server što je zahtevalo da se podaci često šalju na server i da server generiše odgovor koji će vratiti klijentu. Ovakav koncept je bio adekvatan sve dok je postojao mali broj oglašavača i veliki broj pasivnih korisnika. Mogućnosti i izgled aplikacija bile su daleko od standardnih desktop aplikacija. Sa povećanjem broja korisnika njihova uloga postaje sve značajnija i dolazi do potrebe za novijim, kvalitativno drugačijim oblicima interakcije korisnika sa aplikacijom. Razvoj internet tehnologija omogućio je da aplikacije kako po izgledu tako i po funkcionalnosti počnu sve više da podsećaju na desktop aplikacije. Tako i dolazimo do pojma bogate internet aplikacije (Rich Internet Application-RIA). U mnoštvu tehnologija koje su se tokom godina pojavile u ovom radu ćemo se baviti sa mogućnostima koje nam nude dve AJAX i JSF.

2.1 Ajax <<

2.1.1. Istorija AJAX-a

2.1.2 Razlike između tradicionalnih i AJAX internet aplikacija

2.1.3 Načini upotrebe AJAX tehnologije

I pored ogromnog tehnološkog razvoja interneta tokom prošle decenije, mogućnosti internet aplikacija zaostaju za standardnim desktop aplikacijama. Svaka interakcija sa web aplikacijom uzrokuje čekanje dok se ne obavi komunikacija između aplikacije i servera preko interneta. Bogate internet aplikacije imaju dve značajne osobine koje ih približavaju desktop aplikacijama. To su performanse i bogat GUI. Performanse se u mnogome poboljšavaju korišćenjem AJAX tehnologije. AJAX je skraćenica od Asynchronous JavaScript And XML. AJAX koristi mogućnosti JavaScript tehnologije koja se izvršava na klijentskoj strani kako bi komunicirao sa serverskom stranom i učinio aplikaciju osetljivijom na akcije klijenata. Glavna prednost AJAX-a je u tome što omogućava asinhrono osvežavanje delova strana.

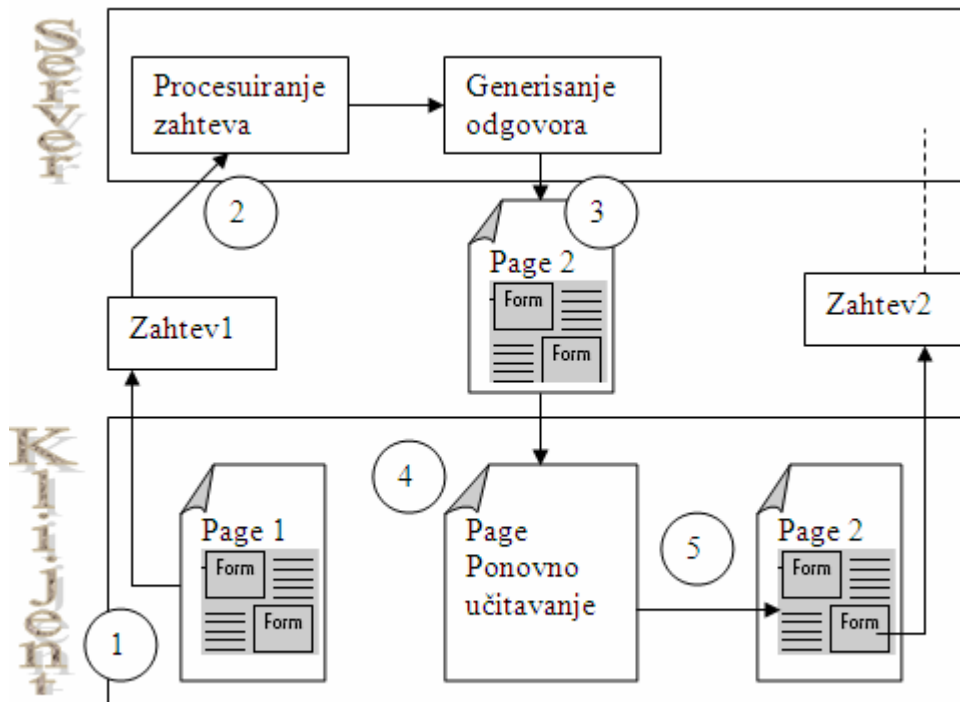
2.1.1 Istorija AJAX-a <<

Termin AJAX uveo je Jesse James Garrett 2005. godine kada je prezentovao prethodno neimenovanu tehnologiju klijentima. Tehnologije koje Ajax koristi (XHTML, JavaScript, CSS, DOM...) postojale su već godinama unazad. 1990. godine Netscape je razvio LiveScript tehnologiju koja je omogućila da se script uključi u web

stranu i da se izvršava na klijentskom pretraživaču. Kasnije od LiveScript-a nastaje JavaScript. 1998. Microsoft razvija XMLHttpRequest objekat koji omogućava da se kreiraju i obrađuju asinhroni zahtevi i odgovori. Ovime se omogućava da korisnik interaguje sa ostatkom strane i pre nego što server obradi asinhroni zahtev. Danas je AJAX jedna od najkorišćenijih internet tehnologija

2.1.2 Razlike između tradicionalnih i AJAX internet aplikacija ≤

U ovom poglavlju razmatramo ključne razlike između AJAX-a i tradicionalnih internet aplikacija. Slika 2.1 prikazuje tipičnu interakciju između klijenta i servera u tradicionalnoj web aplikaciji u kojoj korisnik popunjava formu za registraciju. Prvo korisnik popuni formu i pritisne submit dugme (korak 1). Browser generiše zahtev koji server prihvata i obrađuje (korak 2). Server generiše i šalje odgovor koji sadrži stranicu koju će browser izrendati klijentu (korak 3). Browser mora da učitava novu stranu (korak 4) i zbog toga privremeno prozor browser-a ostaje prazan. Za to vreme korisnik ne može da interaguje sa aplikacijom. Nakon interakcije korisnika sa dobijenom formom ceo proces se ponavlja.



Sl. 2.1 Tradicionalna Web Aplikacija

Ajax aplikacija dodaje sloj između klijenta i servera koji upravlja komunikacijom između njih. Kada korisnik interaguje sa delom strane, klijent (browser) kreira XMLHttpRequest objekat (korak 1) koji šalje asinhroni zahtev (korak 2) i čeka od odgovor od strane servera. Pošto je zahtev asinhron korisnik može da nastavi da interaguje sa aplikacijom na klijentskoj strani dok paralelno server obrađuje raniji zahtev. Svaka korisnička akcija može da prouzrokuje slanje novog zahteva. (kreiranje novog XMLHttpRequest objekta (koraci 3-4). Kada server pošalje odgovor na prvi zahtev

The diagram illustrates the asynchronous request-response cycle in a web browser. It shows the flow of data and control between the browser, the server, and the XMLHttpRequest object.

- Browser (Korisnik):** The top part of the diagram shows the user's interaction with the browser. The browser sends requests to the server and receives responses. The browser also updates the page and re-renders it.
- Server (Korisnik):** The bottom part of the diagram shows the server's interaction with the browser. The server processes requests and sends responses back to the browser.
- XMLHttpRequest Object:** The central part of the diagram shows the XMLHttpRequest object, which is responsible for sending requests and receiving responses. It also handles the callback function.
- Sequence of Events:**
 - Korisnička akcija kreira asinhroni zahtev:** The user triggers an asynchronous request.
 - Procesuiranje zahteva 1:** The browser sends the request to the server.
 - Procesuiranje odgovora:** The server processes the request and sends a response back to the browser.
 - Procesuiranje zahteva 2:** The browser sends the response back to the server.
 - Osveženje dela stranice:** The browser updates the page.
 - Page 1:** The page is re-rendered.
 - Procesuiranje odgovora:** The browser sends the response back to the server.
 - Osveženje dela stranice:** The browser updates the page again.

2.1.3 Načini upotrebe AJAX tehnologije <<

6

Zato je bolje koristi gotove JavaScript biblioteke koje sakrivaju brojne detalje koje je potrebno poznavati kada se piše čist AJAX. Neke od biblioteka su Prototype, Dojo, Rico itd. U primerima za laboratorijske vežbe biće opisana upotreba kako čistog tako i AJAX-a uz pomoć biblioteka.

2.2 JSF <<

2.2.1 [Istorija JSF-a](#)

2.2.2 [Šta je JSF ?](#)

2.2.3 [Navigacioni model](#)

2.2.4 [JSF arhitektura](#)

2.2.4.1 [Model](#)

2.2.4.2 [View](#)

2.2.4.3 [Controller](#)

2.2.5 [JSF životni ciklus](#)

2.2.6 [Dodatne klase\(konverteri,validatori listeneri\)](#)

2.2.1 Istorija JSF-a <<

Krajem 70-ih i početkom 80-ih godina prošlog veka dolazi do razvoja PC računara za razliku od velikih mejnfrejmova koji su postojali pre toga. Aplikacije nastale u to vreme bile su moćne i mogle su da obavljaju brojne zadatke ali postojao je jedan glavni nedostatak, bile su namenjene isključivo jednom korisniku. E-mail kao i centralna baza podataka na serveru nisu postojale. Ovakve aplikacije koje su se instalirale i koristile na jednom računaru objedinjavale su tri aplikaciona sloja (prezentacija, biznis logika,podaci)u jedan. Aplikacije su bile teške za održavanje a razmena podataka je bila gotovo nemoguća.

Sa razvojem interneta i web browser-a nastaju višeslojne aplikacije koje dele aplikaciju u tri sloja i omogućavaju web programerima da se fokusiraju na razvoj odgovarajućeg sloja.Ovi slojevi su model (pristup podacima), view (prezentacija) i controller(logika). Tako dolazimo do programske paradigme poznate kao MVC arhitektura.

Kako bi došli do toga zašto je potreban framework kao što je JSF opisaćemo ukratko tehnologije koje su se koristile za razvoj internet aplikacija u proteklih dvadesetak godina.

Sredinom 90-ih kada je razvoj web aplikacija bio u pvoju glavna tehnologija je bila CGI (Common GateWay Interface) koji je služio za kreiranje i prikaz dinamičkog sadržaja. CGI je tehnika koja omogućava da se sa web stranice pozove proces na serverskoj strani koji generiše dinamički sadržaj (npr. koliko je puta posećena strana). Program koji je proizvodio dinamički sadržaj je bio deo operativnog sistema.To je bio glavni nedostatak ove tehnologije jer je svaki zahtev klijenta da vidi stranu sa dinamičkim sadržajem rezultovao u stvaranju novog procesa što je skupa operacija.

Sledeći korak u evoluciji web aplikacija bio je pojava Java Servlet API-ja. Do tada Java se nije koristila kao tehnologija za pisanje serverske strane već uglavnom za pravljenje apleta. Java servlet API je pored boljih performansi u odnosu na CGI imao još

par ključnih prednosti. Servleti su pisani u Javi što omogućava objektno orijentisani pristup i što je još važnije servleti mogu da se izvršavaju na bilo kojoj platformi koja podržava Javu. Servleti su imali i svoje nedostatke. Generisanje HTML koda je bio mučan i posao podložan greškama. Kao primer navodimo kako servlet generiše HTML tag za tabelu.

```
out.println("<table width=\"75%\" border=\"0\" align=\"center\">")
```

JSP (Java Server Pages) tehnologija je sledeći važan korak u razvoju internet aplikacija. JSP je omogućio jednostavniju stranično orijentisanu mogućnost da se generišu velike količine dinamičkog HTML sadržaja. Upotreba JSP-a omogućila je jednostavniju izgradnju korisničkih interfejsa. JSP stranice proširuju klasične HTML strane ubacivanjem specijalnih JSP tagova. JSP mehanizam funkcioniše tako što je na serveru instaliran specijalan servlet poznat kao JSP kontejner. Ovaj servlet obrađuje sve zahteve za JSP stranicama. JSP kontejner prevodi traženu JSP stranu u servlet kod koji se kompajlira i izvršava. Naknadni zahtevi za istom stranom prouzrokuju pozivanje kompajliranog servleta. Promena JSP strane uzrokuje samo ponovno kompajliranje servleta.

U poslednjih deset godina zahtevi klijenata za sve bogatijim i interaktivnijim aplikacijama rasli su iz dana u dan. Korisnici zahtevaju aplikacije koje će imati ugrađenu bezbednost, internacionalizaciju, prenosivost itd. Višeslojne aplikacije moraju uspešno da obrade sve tražene zahteve a njihova složenost stalno raste. Zbog toga su potrebna rešenja koja će omogućiti jednostavniju izgradnju složenih aplikacija. JSP i servlet tehnologije ne omogućavaju dovoljnu apstrakciju i sakrivanje posla koji je potreban da bi se implementirala višeslojna aplikacija. Takođe programeri su u JSP strane često stavljali Java kod što je dovelo do mešanja prezentacionog i logičkog sloja i napravilo aplikacije težim za održavanje. Zbog toga su se mnoge softverske kompanije okrenule razvoju framework-a koji treba da omoguće web programerima lakšu izgradnju višeslojnih aplikacija.

Jedan od najpopularnijih framework-a nastalih u proteklih par godina je Jakarta Struts. Ovaj framework elegantno razdvaja kod za serversku stranu od prezentacionog koda i u potpunosti poštuje MVC paradigmu. Ipak razvoj korisničkog interfejsa je i dalje vrlo komplikovan i zahteva dosta posla.

2.2.2 Šta je JSF ? <<

JSF (Java Server Faces) je standardni java komponentno baziran framework koji služi za izgradnju korisničkog interfejsa u web aplikaciji. Njegova ključna prednost je što pojednostavljuje razvoj korisničkog interfejsa što često predstavlja najteži deo posla u izgradnji web aplikacije ako se koriste standardne Java web tehnologije kao što su servleti i JSP strane.

Za razliku od drugih tradicionalnih MVC web framework-a JSF koristi komponentno baziran pristup. JSF programeri koriste skup gotovih UI komponenti koje sakrivaju veliki deo posla koji je potreban da bi se unela veća funkcionalnost u web aplikacije. Tako programeri mogu da se koncentrišu na prezentacioni (view) sloj bez znanja da komponente koje koriste kriju script ili neki drugi kod.

2.2.3 Navigacioni model <<

JSF je opremljen deklarativnim navigacionim modelom koji omogućava programerima da navigaciona pravila definišu na jednom mestu. Navigaciona pravila pišu se u faces-config.xml fajlu. Sledeći kod prikazuje jedno navigaciono pravilo.

Uzorak koda: UINavigationo pravilo konfigurisano u faces-config.xml fajlu

```
<navigation-rule>
  <from-view-id>/login.jsp</from-view-id>
  <navigation-case>
    <from-outcome>success</from-outcome>
    <to-view-id>/result.jsp</to-view-id>
  </navigation-case>
</navigation-rule>
```

Ovo navigaciono pravilo nam govori da se sa strane login.jsp ako je ishod success odlazi na stranu result.jsp. Ishod je rezultat (String) akcije koja je izvršena kada je korisnik pritisnuo dugme na strani i time poslao formu. Akcije u JSF se povezuju za specifične UI komponente.

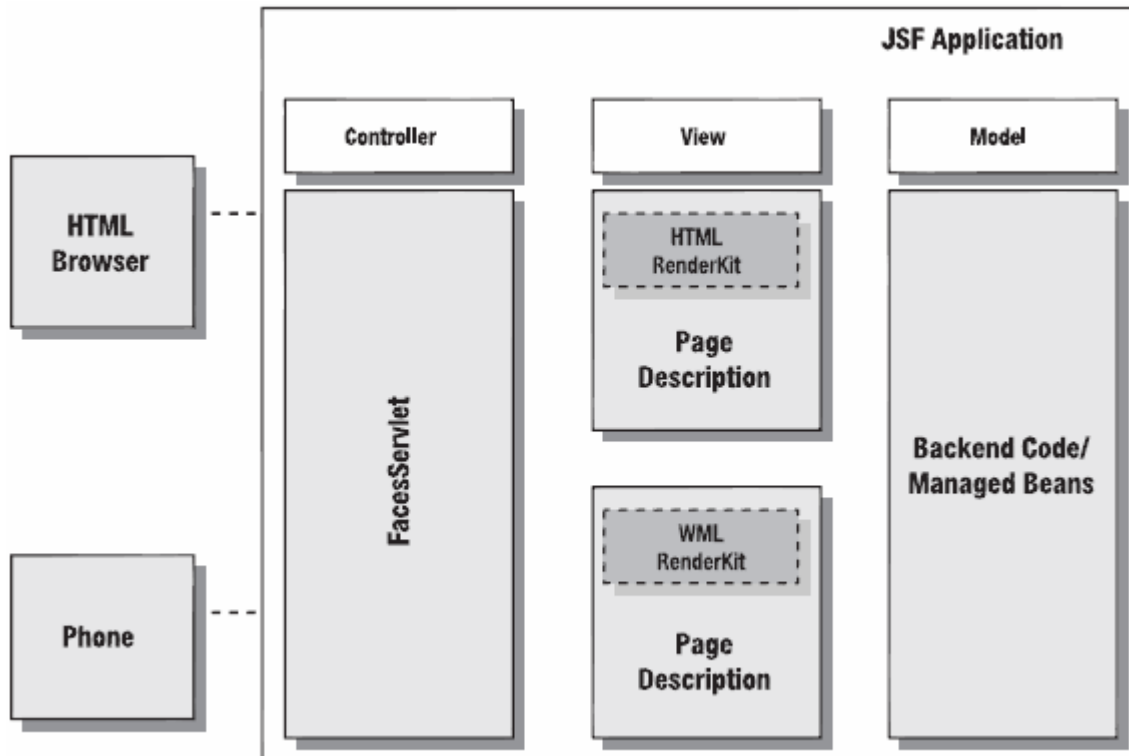
2.2.4 JSF arhitektura <<

2.2.4.1 Model

2.2.4.2 View

2.2.4.3 Controller

JSF kao i struts prati MVC dizajn paradigmu. Na slici 2.3 prikazana je MVC arhitektura sa JSF-om.



Sl 2.3

2.2.4.1 Model <<

Za model sloj tj. pristup podacima koriste se managed bean-ovi. Managed bean-ovi sadrže biznis kod za pristup podacima i mogu se povezivati sa JSF komponentama. Managed bean-ovi se registruju u faces-config.xml fajlu. Primer kako se managed bean registruje dat je u sledećem uzorku koda.

```
<managed-bean>
  <managed-bean-name>user</managed-bean-name>
  <managed-bean-class>beans.UserBean</managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
</managed-bean>
```

Managed beanovi mogu da se koriste na JSP strani, npr.

```
<h:inputText value="#{user.name}"/>
```

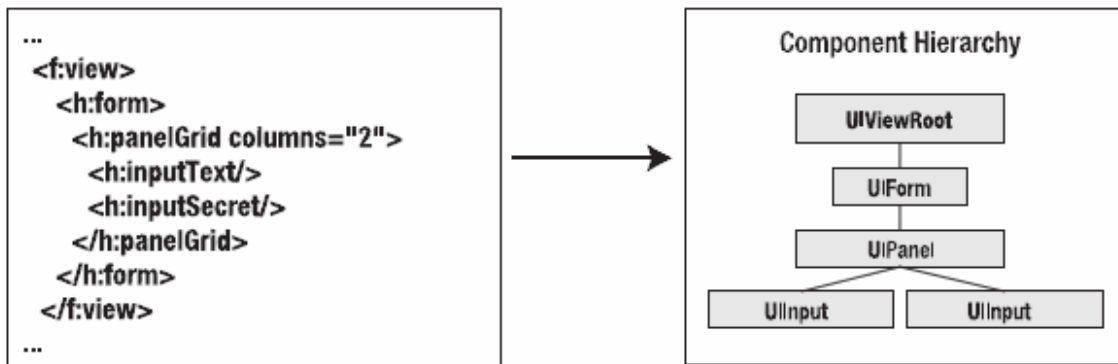
U ovom kodu definišemo bean sa imenom user koji pripada klasi beans.UserBean i koji će biti živ dok traje sesija. Jednom definisan bin može da se koristi na JSP stranama. U ovom slučaju inputText polje smo povezali sa poljem bina name.

2.2.4.2 View <<

Prezentacioni sloj je sloj sa svim kodom potrebnim za prezentaciju korisničkog interfejsa. Ovaj sloj čine JSP strane sa podrškom za JSF tagove(JSF strane). JSF tagovi povezani su sa odgovarajućim JSF komponentama. UI komponente su temelj prezentacionog sloja. UI komponente definišu ponašanje komponenti, način na koji će se one prikazati klijentu itd. UI komponente mogu same da se rendaju ili da budu povezane sa posebnom Renderer klasom zaduženom za rendanje tj. prikaz komponente. Da bi na JSP strani mogli da koristimo JSF tagove potrebno je prvo napisati dve taglib direktive.

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

Sve što se prikazuje na klijentskoj strani ima svoje ogledalo u vidu stabla UI komponenti na serverskoj strani. Na vrhu ovog stabla uvek se nalazi instanca `UIViewRoot` klase. Zato pre nego što napišemo bilo koji JSF tag moramo da stavimo `f:view` tag. Kao deo ovog taga stavlja se `h:form` tag a kao njegova deca ostali elementi forme. Na slici 2.4 je prikazan primer koda koji se piše na JSP strani i stablo UI komponenti koje se generiše u memoriji servera.



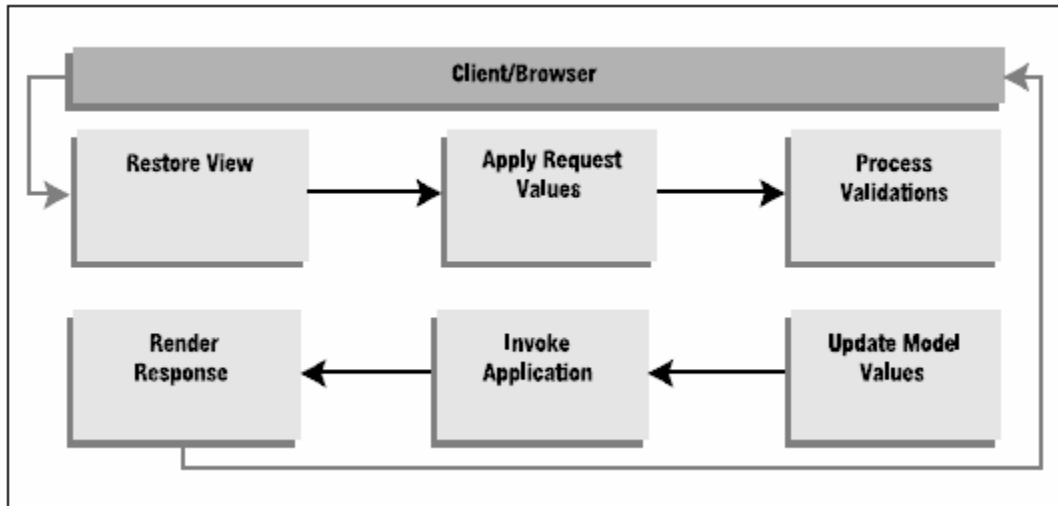
Sl. 2.4

Rekli smo da je ono što razlikuje JSF framework od drugih framework-a to što je on komponentno baziran. JSF donosi skup komponenti koje se mogu koristiti ali i daje mogućnost izgradnje korisničkih (custom) komponenti. To je jedan od razloga zbog kojih je JSF idealno kombinovati sa AJAX tehnologijom. Vešti JSF programeri mogu da napišu korisničke komponente koje će u sebi sakriti AJAX kod a zatim ih ostali JSF programeri mogu koristiti bez potrebe da sami pišu script kod. U četvrtom poglavlju u primeru kombinovanje komponenti ćemo detaljnije opisati kako se prave JSF komponente.

Dakle još jednom ukratko. JSF programeri na JSP strane ubacuju JSF tagove. Za svaki JSF tag kao što je npr. `h:inputText` tag ili `h:commandButtonTag` na serverskoj strani biće instancirana odgovarajuća UI komponenta koja ima dužnost da klijentu izrenda odgovarajući prezentacioni kod. Komponenta može sama da rendera kod ili da bude povezana sa klasom za renderanje. Upotreba klase za renderanje je bolja jer se time omogućava upotreba komponenti na različitim browser-ima.

- **Invoke Application faza:** faza u kojoj se izvršavaju akcije potrebne da se opsluži zahtev klijenta i u kojoj se vrši navigacija na određenu stranu.
- **Render response faza:** faza u kojoj se klijentu rendo odgovor. Za renderanje HTML ili drugog koda zadužene su komponente ili njihove Renderer klase.

Na slici 2.6 je prikazan JSF životni ciklus.



Sl. 2.6

2.2.6 Dodatne klase (konverteri,validatori listeneri) <<

JSF omogućava pridruživanje raznih pomoćnih klasa UI komponentama. Ove pomoćne klase dele se u konvertere,validatore i osluškivače(listener-e). Konverteri omogućavaju konverziju stringova koji se kupe sa forme u određeni tip podataka. Validatori proveravaju ispravnost tačno određenog tipa podataka (npr da li je broj u dozvoljenom opsegu ili da li je datum koji je poslat iz prošlosti).Sledeći kod prikazuje kako se jednom input polju pridružuje konverter.

```

<h:inputText value="#{sample.date}" >
  <f:convertDateTime pattern="yyyy-MMM-dd" />
</h:inputText>

```

U datom kodu tekst polju je pridružen jedan od unapred definisanih konvertera DateTimeConverter. Atribut pattern definiše uzorak po kome datum mora da bude napisan da bi konverzija bila uspešna. JSF definiše određeni broj standardnih konvertera i validatora. Postoji mogućnost definisanja korisničkih (custom) validatora i konvertera.

JSF omogućava i da se komponentama pridruže osluškivači događaja slično kao što se radi u AWT-u i Swing-u.

```

<h:commandButton value="Login"
  action="success"
  actionListener="#{sample.onLogin}" />

```

U ovom primeru komandno dugme ima dva atributa: action i actionListener. Oba atributa omogućavaju povezivanje sa metodom managed bean-a. Razlika je što action atribut zahteva metodu koja vraća string i nema argumente a actionListener zahteva void metodu sa ActionEvent argumentom. I jedna i druga metoda izvršavaće se u petoj fazi životnog ciklusa. String koji vraća action metoda koristi se za navigaciju tj odluku koja će se strana izrendati klijentu. U slučaju kada se koristi samo actionListener klijentu se vraća ista strana. Ako je vrednost atributa action string kao u našem primeru, onda se ne izvršava nikakva akcija već se samo string koristi za navigaciju.

JSF implementacija ispaljuje događaje koji se nazivaju phase events, pre i posle svake faze životnog ciklusa. Ovi događaji se obrađuju pomoću posebnih osluškivača (Phase Listener-a) . Da bi koristili Phase Listener moramo da ga registrujemo u faces-config.xml fajlu na sledeći način.

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

Ovaj kod registruje jedan osluškivač mada je ovaj broj neograničen. Osluškivači će se pozivati u onom redosledu u kome su registrovani u faces-config.xml fajlu.

Phase Listener mora da implementira PhaseListener interfejs koji definiše tri metode.

- PhaseId getId()
- void afterPhase(PhaseEvent)
- void beforePhase(PhaseEvent)

Prvi metod govori JSF implementaciji kada da prosledi događaj osluškivaču. Npr. ako ovaj metod vrati PhaseId.APPLY_REQUEST_VALUES instancu klase PhaseId metode beforePhase i afterPhase biće pozvane tačno jednom u toku životnog ciklusa, pre i posle Apply Request Values faze.

Komponentama izvedenim iz UIInput klase mogu se dodavati valueChangeListener-i.

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

U ovom primeru ako se promeni vrednost menija nakon validacije u trećoj fazi životnog ciklusa pozvaće se metoda bina countryChanged. Tag h:selectOneMenu će se prikazati kao select element sa atributom size="1".

Brojne mogućnosti koje donosi JSF ne mogu da se opišu na ovako malom prostoru. Zato ćemo u četvrtom poglavlju kada budemo opisivali implementaciju koda da se bavimo još nekim dodatnim detaljima vezanim za JSF.

Ajax i JSF predstavljaju idealan par. Mnoge teškoće nastaju kada programeri pišu AJAX kod. Mogućnost pravljenja custom JSF komponenti i enkapsulacije script koda u komponentu omogućava da određeni broj programera može da napiše AJAX-JSF

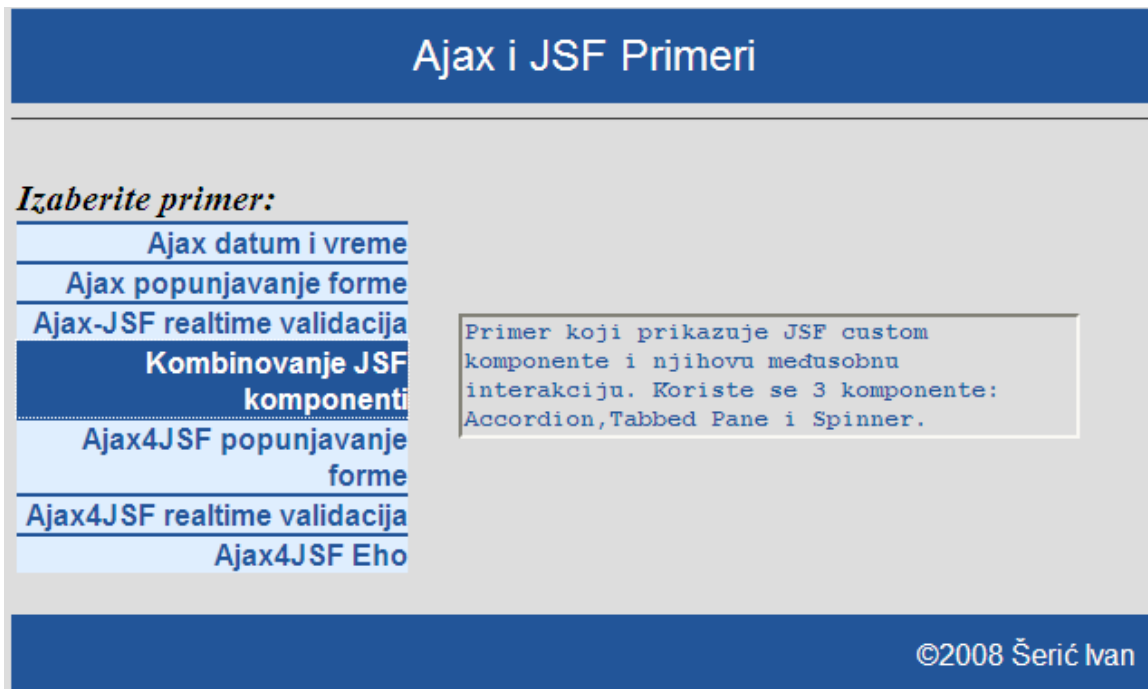
komponente koje drugi programeri posle mogu da koriste na način kao što koriste svaku JSF komponentu.

3 Opis sistema sa korisničke strane <<

Cilj ovog poglavlja je da opiše funkcionalnosti urađenih labaratorijskih vežbi. Labaratorijske vežbe sastoje se od sedam primera. Za razliku od četvrtog poglavlja u kojem ćemo da objasnimo kako je sistem implementiran, ovo poglavlje će dati vizuelni opis primenjenih tehnologija, onako kako to vidi korisnik.

3.1 Početni meni <<

Kada se startuje aplikacija otvara se meni u kome može da se izabere jedan od sedam primera (slika 3.1). Kada se mišem pređe preko nekog od linkova, tada se u polju za tekst prikazuje kratak opis primera. Treba obratiti pažnju da prilikom prelaska preko komandnih linkova dolazi do ponovnog renderanja samo polja za tekst. Ovaj efekat je omogućen zato što smo svakom komandnom linku dodali AJAX funkcionalnost korišćenjem AJAX4JSF framework-a [Vidi detalje](#).

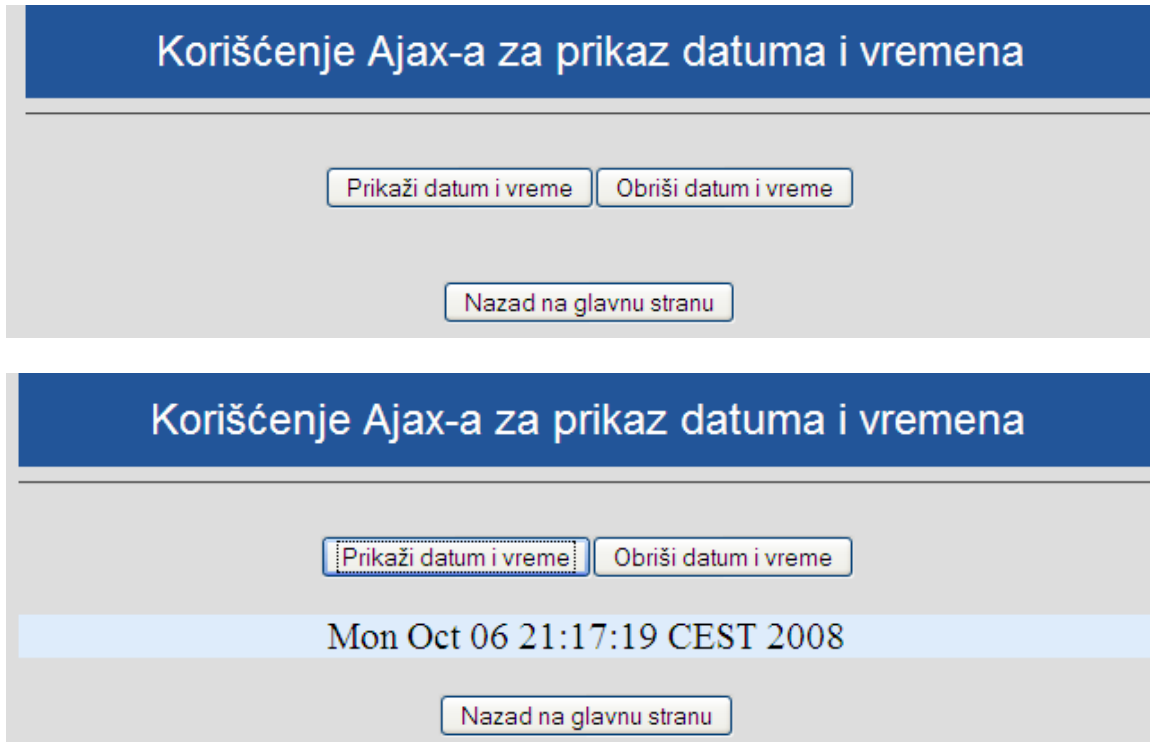


Sl 3.1

3.2 Korišćenje Ajax-a za prikaz datuma i vremena <<

Ovaj primer takođe pokazuje mogućnosti AJAX tehnologije. Kada se pritisne dugme *Prikaži datum i vreme* poziva se script funkcija u kojoj se generiše XMLHttpRequest objekat koji generiše AJAX zahtev i prosleđuje ga serveru. Poziva se poseban servlet tj u ovom slučaju ne pokreće se JSF životni ciklus. Servlet vraća trenutni

datum i vreme kao odgovor. Kada se dobije odgovor od servleta poziva se script funkcija u kojoj se u DOM (document object model) web strane upisuje trenutni datum i vreme. Tako se neće osvežiti cela strana već samo jedan deo. Ponovnim pritiskom na dugme ponovo će se prikazati trenutno vreme. U ovom primeru korišćen je čist AJAX. [Vidi detalje.](#)



Sl 3.2

3.3 Korišćenje Ajax-a za popunjavanje forme <<

Ovaj primer prikazuje jednu od standardnih primena AJAX-a u kojoj klijent popuni deo forme a to uzrokuje ažuriranje drugih delova forme. Konkretno kada klijent upiše zip kod u tekst polje na osnovu upisanog sadržaja ažuriraće se polja grad i država. U trenutku kada zip tekst polje izgubi fokus šalje se AJAX zahtev serveru i zatim se osvežavaju polja grad i država. U ovom primeru nismo koristili čist AJAX već smo pokazali kako se AJAX zahtevi mogu ostvariti pomoću Prototype JavaScript biblioteke. [Vidi detalje.](#)

Korišćenje Ajax-a za popunjavanje forme

Molimo vas recite nam gde živite:

Zip kod:

Grad:

Država:

[Nazad na glavnu stranu](#)

Korišćenje Ajax-a za popunjavanje forme

Molimo vas recite nam gde živite:

Zip kod:

Grad:

Država:

[Nazad na glavnu stranu](#)

Sl 3.3

3.4 Korišćenje Ajax-a i JSF-a za realtime validaciju <<

Ovaj primer je proširenje prethodnog jer uvodi i realtime validaciju. Kada korisnik upiše peti znak u zip kod polje trenutno dobija odgovor da li je zip kod ispravan pre nego što polje izgubi fokus. Takođe u ovom primeru postoji složenija interakcija između JSF frameworka i AJAX-a jer se za validaciju koristi JSF validator. U prethodnim primerima korišćenje JSF-a se svodi uglavnom na upotrebu određenih JSF komponenti kao što je komandno dugme ili komandni link i komponenti za uređivanje. Pošto se koristi JSF validator koji je pridružen zip kod tekst polju ne može da se napiše servlet koji će obraditi AJAX zahtev jer on ne zna ništa o JSF-u i view state-u pa ne može da pristupi instanci validatora. Umesto toga zahtev obrađujemo pomoću Phase Listener-a koji će reagovati nakon Restore View faze kada je view state obnovljen u memoriji servera. Phase Listener obrađuje zahtev vrši validaciju i generiše odgovor. Ajax zahtev biće prosleđen JSF servletu nakon što se otkuca peti znak u zip kod polje. Tako će klijent čim otkuca peti znak dobiti informaciju o tome da li je uneo ispravan zip kod.

[Vidi detalje.](#)

Korišćenje Ajax-a i JSF-a za realtime validaciju

Molimo vas recite nam gde živite:

Zip kod: 12344

✖ Niste uneli pravilan zip kod

Grad:

Država:

Nazad na glavnu stranu

Korišćenje Ajax-a i JSF-a za realtime validaciju

Molimo vas recite nam gde živite:

Zip kod: 11000

✔

Grad:

Država:

Nazad na glavnu stranu

Sl 3.4

3.5 JSF-Rico komponenta <<


Korisnik klikom na heder panela može da učini vidljivim njegov sadržaj. Po default-u se prikazuje panel u kome mogu da se vide slike i informacije o američkim predsednicima. Ukoliko je veličina panela suviše mala može da se otvori panel sa spinner komponentom i da se visina panela poveća. (Slika 3.5)

Ovaj primer pokazuje upotrebu više custom JSF komponenti i njihovu međusobnu interakciju. Komponenta Accordion(harmonika) je hibridna komponenta koja enkapsulira JavaScript komponentu Accordion za čiju se izradu korisiti JavaScript biblioteka Rico. Sve što programer treba da uradi je da na JSF strani ubaci tag za Accordion komponentu. Ova komponenta prilikom renderanja prosleđuje HTML generisani kod Rico biblioteci kako bi se dobio efekat harmonike. U Accordion komponentu može da se stavi bilo šta, između ostalog druge JSF komponente. To smo iskoristili da u nju smestimo druge custom komponente (Spinner i Tabbed Pane). Spinner se koristi za povećanje visine panela Accordion komponente čime je ostvarena interakcija komponenti. [Vidi detalje.](#)

JSF-Rico Komponenta

Tabbed Pane

[Jefferson](#) [Roosevelt](#) [Lincoln](#) [Washington](#)



Thomas Jefferson, the 3rd US president, was born in 1743 in Virginia. Jefferson was tall and awkward, and was not known as a great public speaker. Jefferson became minister to France in 1785, after Benjamin Franklin held that post. In 1796, Jefferson was a reluctant presidential candidate, and missed winning the election by a mere three votes. He

Spinner za podešavanje visine panela

[Nazad na glavnu stranu](#)

JSF-Rico Komponenta

Tabbed Pane

Spinner za podešavanje visine panela

245

[Nazad na glavnu stranu](#)



Sl 3.5

3.6 i 3.7

Korišćenje Ajax4JSF framework-a za popunjavanje forme i realtime validaciju <<

Ovi primeri imaju iste funkcionalnosti kao i primeri 2 i 3 ali umesto da sami pišemo kod za AJAX koristimo AJAX4JSF framework. Ovaj framework omogućava da JSF komponentama na jednostavan način dodamo Ajax funkcionalnost. Sve što programer treba da uradi je da na JSF stranu u odgovaraće JSF tagove za komponente ubaci AJAX4JSF tagove. Takođe AJAX4JSF tagovi omogućavaju i neke dodatne efekte kao što je animacija dok se obrađuje AJAX zahtev (slika 3.6).

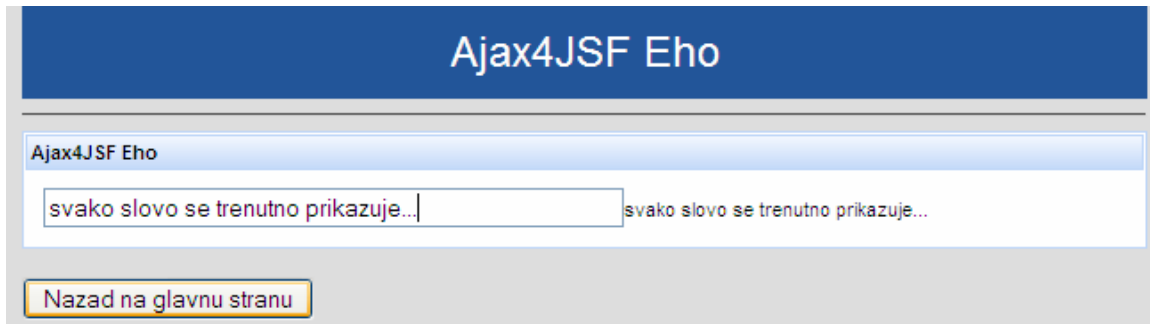
[Vidi detalje za popunjavanje forme.](#)

[Vidi detalje za realtime validaciju.](#)

Sl 3.6

3.8 Ajax4JSF Eho <<

U ovom primeru takođe se koristi AJAX4JSF framework. Svako pojedinačno slovo koje korisnik otkuca u tekst polju kome je pridružena AJAX funkcionalnost se trenutno prikazuje u izlaznom (output polju). [Vidi detalje.](#)



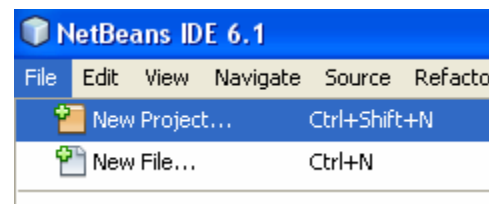
Sl 3.8

4 Implementacija sistema <<

U ovom poglavlju detaljnije ćemo se baviti izradom samog sistema i prikazaćemo najvažnije delove koda. Prvo ćemo reći par rečenica o okruženju u kome je laboratorijski sistem rađen a zatim prelazimo na implementaciju samog sistema.

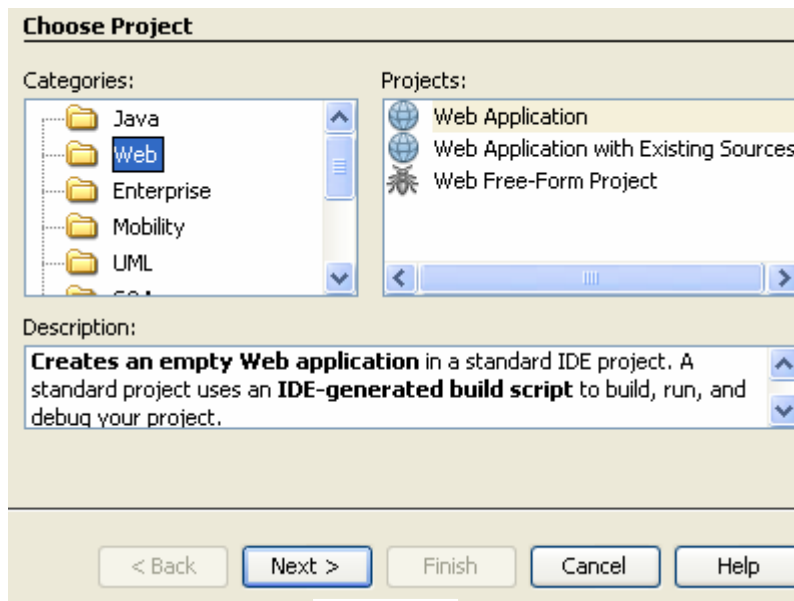
4.1 Radno okruženje <<

Zbog jednostavnosti rada izabrali smo okruženje NetBeans IDE 6.1. Ova verzija NetBeans-a dolazi sa ugrađenim serverima Apache Tomcat 6.0.16 i GlassFishV2. Kada pokrenemo web projekat koji smo napisali izabrani server će se pokrenuti automatski. Ukratko ćemo objasniti kako da pripremimo novi web projekat. Novi projekat otvaramo tako što u padajućem meniju izaberemo opciju File->New Project (Slika 4.1).



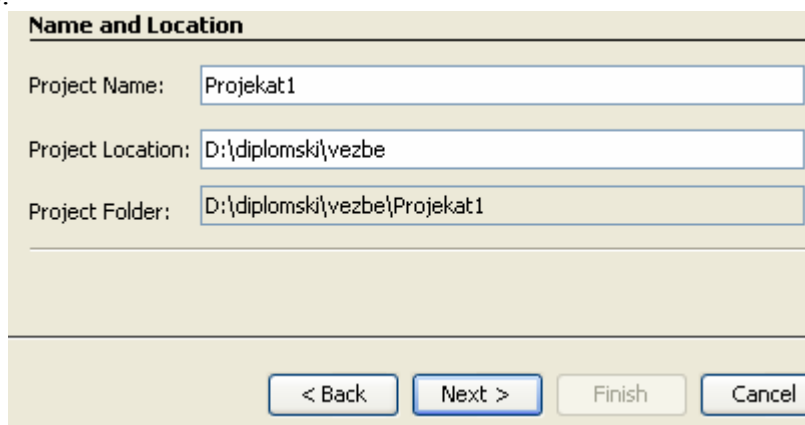
Sl 4.1

Nakon toga dobijamo meni u kome biramo vrstu projekta. U levom prozoru biramo kategoriju web, a zatim u desnom prozoru opciju Web Application. (slika 4.2)



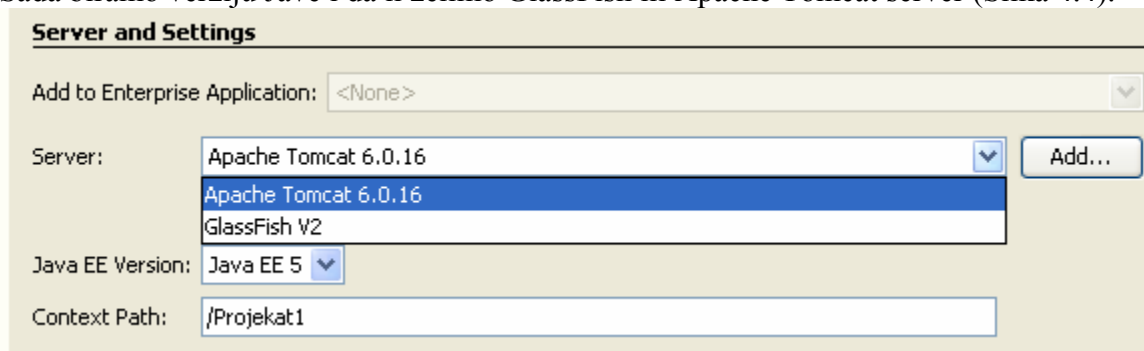
Sl 4.2

U sledećem koraku se bira ime projekta i folder u koji želimo da smestimo projekat (slika 4.3).



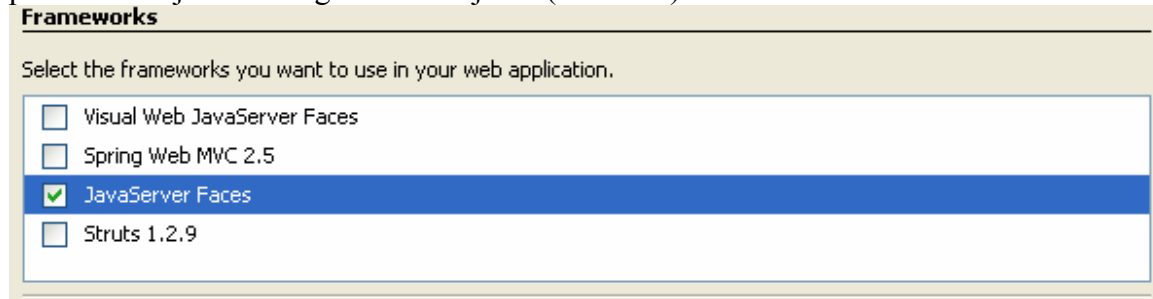
Sl 4.3

Sada biramo verziju Jave i da li želimo GlassFish ili Apache Tomcat server (Slika 4.4).



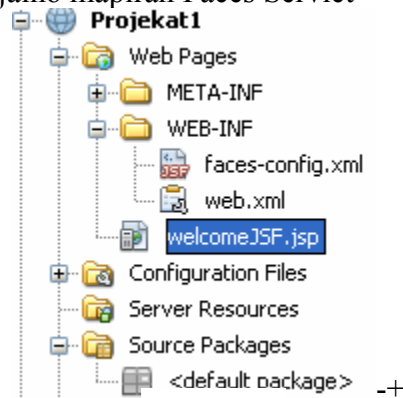
Sl 4.4

U poslednjem koraku biramo da li želimo da uključimo framework. Biramo Java Server Faces i NetBeans će automatski uključiti u projekat biblioteke potrebne za rad i podesiti inicijalno konfiguracione fajlove (Slika 4.5).



SI 4.5

Kada izaberemo JSF framework i pritisnemo dugme finish otvara se novi projekat. U folder Web Pages možemo da smeštamo nove jsp strane. NetBeans kreira sam welcomeJSF.jsp stranu (Slika 4.6). U Source Packages folder smeštamo pakete i pišemo java kod. Ovde ćemo stavljati naše servlete, binove itd. U folderu WEB-INF automatski se smeštaju konfiguracioni fajlovi web.xml i faces-config.xml. Primere upotrebe faces-config.xml fajla smo videli i ranije a i u daljem tekstu u njega ćemo dodavati sadržaj. U web.xml fajlu je inicijalno mapiran Faces Servlet



SI 4.6

koji opslužuje sve faces zahteve. NetBeans vrši automatsko mapiranje tako da će svi zahtevi koji dođu sa url prefiksom /faces/* biti opsluženi od strane Faces Servlet-a. Ovakvo mapiranje se uvodi da bi se na jednostavan način razlikovali faces zahtevi od zahteva za drugim servletima. Prikazujemo kod jednog inicijalnog web.xml fajla.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5" xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation=
"http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
    <context-param>
        <param-name>com.sun.faces.verifyObjects</param-name>
        <param-value>false</param-value>
    </context-param>
```

```

<context-param>
    <param-name>javax.faces.STATE_SAVING_METHOD</param-name>
    <param-value>client</param-value>
</context-param>
<servlet>
    <servlet-name>Faces Servlet</servlet-name>
    <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>/faces/*</url-pattern>
</servlet-mapping>
<session-config>
    <session-timeout>
        30
    </session-timeout>
</session-config>
<welcome-file-list>
    <welcome-file>faces/welcomeJSF.jsp</welcome-file>
</welcome-file-list>
</web-app>

```

Bitno je napomenuti da je NetBeans automatski stavio u listu početnih JSP strana koje se traže kada se pokreće aplikacija welcomeJSF.jsp stranu. Takođe automatski je podešeno da se view state čuva na klijentu a ne u memoriji servera tako da će prilikom svakog klijentskog zahteva sa forme doći i vrednost hidden polja u kome se čuva view state. Na osnovu vrednosti ovog polja restaurira se hijerarhija UI komponenti na serverskoj strani.

Sada smo spremni da pišemo JSF web aplikacije. U daljem delu poglavlja opisaćemo najvažnije delove koda potrebne za realizaciju labaratorijskog sistema zasnovanog na JSF AJAX tehnologiji.

4.2 Primeri [<<](#)

4.2.1 Korišćenje Ajax-a za prikaz datuma i vremena [<<](#) [Vidi opis.](#)

U ovom primeru na pritisak dugmeta formira se asinhroni AJAX zahtev i prosleđuje serveru. Zahtev se prosleđuje posebnom servletu koji smo nazvali AjaxServlet tako da se ne pokreće JSF životni ciklus. AjaxServlet se nalazi u paketu servlets. JSP strana je primerDatumVreme.jsp. Pogledajmo najpre deo forme sa ove strane.


```

<div class="center">
    <h:commandButton type="button" value="#{msgs.button1Ex1}" onclick="getContent()" />
    <h:commandButton type="button" value="#{msgs.button2Ex1}" onclick="clearContent()" />
</div>
<br/>
<div id="output" class="watch"></div> <br/>
<h:form>
    <div class="center"><h:commandButton value="#{msgs.button3Ex1}" action="nazad"/></div>

```

Poslednje dugme ima action atribut string nazad. Ovo dugme služi da se vratimo na početni meni naše aplikacije. Kako bi se to ostvarilo u faces-config.xml fajlu registrovali smo navigaciono pravilo.

```

<navigation-rule>
    <navigation-case>
        <from-outcome>nazad</from-outcome>
        <to-view-id>/welcomeJSF.jsp</to-view-id>
    </navigation-case>
</navigation-rule>

```

Ovakvo pravilo kaže da ako je ishod nazad bez obzira na kojoj se JSP strani nalazili vraćamo se na welcomeJSF.jsp stranu.

Prva dva dugmeta su od ključnog značaja. Vidimo najpre da atribut dugmeta value nije zadat kao string već je definisan kao value expression. Fajl messages.properties definisali smo u paketu com.corejsf. U ovom fajlu nalaze se poruke koje koristimo. Da bi mogli da koristimo poruke definisane u messages.properties fajlu potrebno je i njega registrovati u faces-config.xml fajlu.

```

<application>
    <resource-bundle>
        <base-name>com.corejsf.messages</base-name>
        <var>msgs</var>
    </resource-bundle>
</application>

```

Sada na JSP stranama možemo da pomoću value expression-a pristupimo poruci koja se nalazi u properties fajlu. Uvođenje properties fajlova je dobro zbog internacionalizacije.

Druga važna osobina je da oba dugmeta imaju atribut type="button" što znači da ona neće poslati formu i pokrenuti JSF životni ciklus. Umesto toga pritisak na bilo koje od ovih dugmadi poziva odgovarajuću script funkciju. Kodovi script funkcija su sledeći.

```

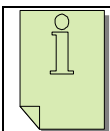
<script type="text/javascript">
    var asyncRequest;
    function getContent() {
        try{
            var url="AjaxServlet";
            init();
            asyncRequest.onreadystatechange=stateChange;
            asyncRequest.open('GET', url, true);
            asyncRequest.send(null);

        }catch(exception){alert("Zahtev nije uspeo!");}
    }
    function init(){
        if(window.XMLHttpRequest) //za mozilla
            asyncRequest=new XMLHttpRequest();
        else if(window.ActiveXObject) //starije verzije explorer
            asyncRequest=new ActiveXObject("Microsoft.XMLHTTP");
    }
    function stateChange() {

        if(asyncRequest.readyState==4 //request je završen
            && asyncRequest.status==200 ){ // uspešno!
            clearContent();
            var div=document.getElementById("output");
            document.getElementById("output").innerHTML=asyncRequest.responseText;
            Fat.fade_element("output", 60, 4000, "#dfeeff", "#dddddd");
        }
    }
    function clearContent() {
        document.getElementById("output").innerHTML="";
    }
}
</script>

```

Funkcija `getContent` priprema i šalje asinhroni AJAX zahtev. Najpre smo pripremili url servleta koji se poziva. Funkcija `init` služi za instanciranje `XMLHttpRequest` objekta. U zavisnosti od tipa browser-a ovaj objekat instanciramo na različite načine. `asyncRequest` je globalna script promenljiva.



Mogli smo samo da napišemo `asyncRequest=new XMLHttpRequest()` i sve bi radilo i u Mozilla Firefox-u i u Internet Explorer-u(verzija 7).

Funkcija `stateChange` poziva se svaki put kada se promeni stanje zahteva. Ova funkcija je callback funkcija i registrovana je tako da se pozove svaki put kada se promeni stanje zahteva tj ona obrađuje događaje koji se ispaljuju svaki put kada se stanje promeni. Stanje zahteva se čuva u posebnom property-u `XMLHttpRequest` objekta koji se zove `readyState` i može da ima vrednosti od 0 do 4 pri čemu 4 znači da je zahtev obrađen.

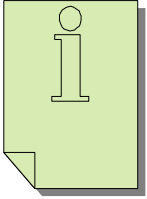
Metoda `open` služi za pripremanje asinhronog zahteva. Prvi argument govori da li će zahtev biti GET ili POST. Drugi argument je url servleta kome se šalje zahtev. Treći je logički i govori da je zahtev asinhron ako je postavljen na `true`. Funkcija `send` šalje asinhroni zahtev serveru.

Već smo rekli da se funkcija `stateChange` poziva svaki put kada se promeni stanje zahteva. Mi želimo da se funkcija izvrši samo ako je zahtev obrađen i to uspešno. Ako je vrednost `status=200` onda je zahtev uspešno obrađen. Prvo se poziva funkcija `clearContent` da obriše sadržaj `div` elementa. Zatim u isto polje upisujemo tekst koji je stigao kao odgovor od strane servera. Ovaj tekst se nalazi u `property-u` `responseText`. Poslednji red `stateChange` metode poziva metodu `fade_element` iz JavaScript biblioteke `Fat` i ima čisto vizuelni efekat. Kratko objašnjenje je da će on obojiti pozadinu `div` elementa i zatim je menjati. Drugi argument je broj frejmova u sekundi. Treći argument kaže da će efekat trajati četiri sekunde, a naredna dva argumenta su početna i krajnja boja.

Ostaje nam još da vidimo deo koda `AjaxServlet-a`.

```
protected void processRequest(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    response.setContentType("text/html;charset=UTF-8");
    response.setHeader("Cache-Control", "no-cache");
    PrintWriter out = response.getWriter();
    try {
        out.write(new Date().toString());
    } finally {
        out.close();
    }
}
```

Ovaj metod dohvata objekat klase `PrintWriter` i upisuje trenutni datum i vreme.

	Browser može da kešira odgovor koji dobije od servera. To je vrsta optimizacije jer kada se javi isti zahtev browser može da izvuče odgovor iz keša. To bi imalo za posledicu da ako dva puta šaljemo zahtev serveru za datumom i vremenom prvi put bi dobili odgovor od servera a drugi put iz keš memorije browser-a. Tako bi dva puta dobili isto vreme. Da bi sprečili da browser pamti sadržaj koji dobije od servera koristimo metod <code>response.setHeader("Cache-Control", "no-cache")</code> .
---	---

Ovaj servlet je mapiran u `web.xml` fajlu na sledeći način.

```

<servlet>
    <servlet-name>AjaxServlet</servlet-name>
    <servlet-class>servlets.AjaxServlet</servlet-class>
</servlet>
<servlet>
<servlet-mapping>
    <servlet-name>AjaxServlet</servlet-name>
    <url-pattern>/AjaxServlet</url-pattern>
</servlet-mapping>
<servlet-mapping>

```

4.2.2 Korišćenje Ajax-a za popunjavanje forme <<

[Vidi opis](#)

U ovom primeru se pomoću AJAX tehnologije na osnovu sadržaja koji se upiše u zip polje automatski ažuriraju polja za grad i državu. Korstili smo Prototype JavaScript biblioteku za slanje asinhronog zahteva serveru. Takođe u ovom primeru uvodimo i JSON (JavaScript Object Notation) notaciju koju je pogodno koristiti u kombinaciji sa AJAX-om.

Ukratko JSON string prati tačno određenu formu. Oblik ovog stringa je

{ "propertyName1" : value1, "propertyName2": value2 }.

Dakle to je lista ime-vrednost property-a. Vrednost može da bude string, niz drugi JSON objekat itd. JSON stringovi mogu se konvertovati u JavaScript objekte pomoću JavaScript funkcije eval().

Primer:

```

contact={ "firstName": "John",
  "lastName": "Smith",
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": 10021
  },
  "phoneNumbers": [
    "212 555-1234",
    "646 555-4567"
  ]
}
var p = eval('(' + contact + ')');
```

U ovom primeru property firstName ima string vrednost, property adress ima za vrednost drugi JSON objekat a property phoneNumbers niz stringova. Nakon primene funkcije eval možemo vrlo lako pristupati property-ima kao p.firstName, p.adress.city, p.phoneNumbers[0].

Pogledajmo sada deo strane primerZip.jsp

```

<h:panelGrid columns="3">
    <h:outputText value="#{msgs.ex2Zip}"/>
    <h:inputText id="zip" size="5" maxlength="5" onblur="getContent()"/>
    <div id="output" class="divError"></div>

    <h:outputText value="#{msgs.ex2City}"/>
    <h:inputText id="grad"/>
    <div></div>

    <h:outputText value="#{msgs.ex2Country}"/>
    <h:inputText id="drzava"/>
    <div></div>

</h:panelGrid>

```

Za uređivanje strane koristimo `h:panelGrid` tag. Ustvari `UIPanel` komponenta će izrendati HTML tabelu na strani. Atribut `columns` govori da imamo tri kolone. Dakle nakon tri elementa četvrti će ići u novi red. Script koji generiše `XMLHttpRequest` objekat biće pozvan kada se izađe iz fokusa `zip` tekst polja. Funkcija `getContent` razlikuje se od funkcije iz prethodnog primera jer koristimo `Prototype` biblioteku za slanje asinhronog zahteva.

```

<script type="text/javascript">
    var uzorakZip=/\d\d\d\d\d/;
    function getContent(){
        clearContent();
        var zip=document.getElementById("forma:zip").value;
        var sadrzi=zip.search(uzorakZip);
        if( sadrzi==-1 )
            document.getElementById("output").innerHTML=
                "Niste uneli pravilan zip code.Unesite 5 cifara!";
        else{

            new Ajax.Request("zipChanged.ajax", //url
            {
                method: "get",
                parameters: "zip=" + zip, //parametri
                onComplete: stateChange //callback fja
            });

        }
    }
}

```

Ajax zahtev se kreira samo ako je korisnik uneo tačno 5 cifara. Koristimo objekat `Prototype` biblioteke `Ajax.Request` koji će poslati asinhroni zahtev servletu `AjaxServlet2`. Prvi argument je url korišćenog servleta. Vidimo da se mapiranje u ovom slučaju razlikuje od prvoj primera .Da bi pozvali `AjaxServlet2` koristimo url `zipChanged.ajax`. Ovo će biti jasnije ako pogledamo kako je servlet mapiran u `web.xml` fajlu.

```

<servlet>
    <servlet-name>AjaxServlet2</servlet-name>
    <servlet-class>servlets.AjaxServlet2</servlet-class>
</servlet>
<servlet>
<servlet-mapping>
    <servlet-name>AjaxServlet2</servlet-name>
    <url-pattern>*.ajax</url-pattern>
</servlet-mapping>

```

Dakle bilo kakav poziv servleta koji se završava sa .ajax pozvaće AjaxServlet2. Drugi argument govori da je metod get a treći argument su parametri koje prosleđujemo kao deo zahteva. U našem primeru prosleđujemo vrednost koju smo upisali u zip polje. Pri obrađenom zahtevu poziva se funkcija stateChange.

Funkcija stateChange uzima tekst koji je stigao kao odgovor i pošto je taj tekst pisan u JSON notaciji pomoću funkcije eval on se pretvara u JavaScript objekat. Zatim se polja ovog objekta koriste da se ažuriraju tekst polja za grad i državu.

```

function stateChange (asyncRequest){ //rezultat je vracen prema json notaciji
    var grad=document.getElementById("forma:grad");
    var drzava= document.getElementById("forma:drzava");
    var resp=asyncRequest.responseText;
    var resp2=resp.substring(0,resp.length-7);
    // grad.value=resp2;

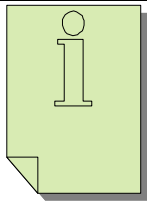
    var jsonGradDrzava=eval('(' +resp2+')');

    grad.value=jsonGradDrzava.city;
    drzava.value=jsonGradDrzava.state;
    Fat.fade_element("forma:grad",60,4000,"#dfeeff","#ffffff");
    Fat.fade_element("forma:drzava",60,4000,"#dfeeff","#ffffff");

}
}

```

Obratimo pažnju na to da je id tekst polja grad i država u obliku forma:grad i forma:država. Ovo je posledica toga kako se inputText polja rendaju klijentu (tj šta rendaju UIInputText klasa). Npr <h:inputText id="tekst"> biće izrendan kao <input type="text" name="idforme:tekst">. Dakle id forme se dodaje id-u komponente. Ukoliko želimo da sprečimo dodavanje id-a forme elementima forme možemo da upotrebimo atribut prependID(<h:form id="forma" prependID="false">).



Ako se obrati pažnja na kod funkcije stateChange primećuje se da je od teksta koji je stigao kao odgovor odsečeno sedam znakova a tek onda pozvana funkcija eval. Ovo je moralo da se uvede jer je uvođenjem podrške za AJAX4JSF iz neobjašnjivih razloga na kraj JSON stringa dodato 7 karaktera. Pre uvođenja podrške nije bilo potrebe za odsecanjem dela stringa.

Ostaje nam još da pogledamo kod za AjaxServlet2.

```
protected void processRequest(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
    response.setContentType("text/html;charset=UTF-8");
    response.setHeader("Cache-Control", "no-cache");
    response.setStatus(HttpServletResponse.SC_OK);
    PrintWriter out = response.getWriter();
    try {
        int zip=Integer.parseInt(request.getParameter("zip"));
        String jsonGradDrzava=GradDrzava.uzmiIzBaze(zip);// baza u ovom slucaju u memoriji
        out.write(jsonGradDrzava);
    } finally {
        out.close();
    }
}
```

Dakle dohvatamo parametar zahteva zip i na osnovu njega iz baze podataka dohvatamo odgovarajući grad i državu. Baza podataka je jednostavna i nalazi se u memoriji (videti Source Packages/Beans/GradDrzava.java).

```
//vracamo u json notaciji {"property": "value"}
public static String uzmiIzBaze(int zip) {
    switch (zip) {
        case 11000:
            return "{\"city\":\"Beograd\",\"state\":\"Srbija\"}";
        case 21000:
            return "{\"city\":\"Novi Sad\",\"state\":\"Srbija\"}";
        default:
            return "{\"city\":\"Nema podataka\",\"state\":\"Nema podataka\"}";
    }
}
```

4.2.3 Korišćenje Ajax-a i JSF-a za realtime validaciju <<

[Vidi opis.](#)

U ovom primeru pokazaćemo značajniju interakciju AJAX-a i JSF-a. U primeru se vrši realtime validacija i pri tome se koristi JSF validator. AJAX zahtev se šalje kada unesemo peti znak u zip polje. Takođe AJAX zahtev se šalje i kada se napusti fokus zip polja ali taj kod je gotovo identičan kodu iz primera dva tako da ga nećemo analizirati. Pogledajmo deo strane primerZipRTV.jsp koji se odnosi na zip inputText polje.

```
<h:inputText id="zip" size="5" maxlength="5"
             onkeyup="getContent()"
             onblur="getContent2()"
             value="#{zipBean.zip}">
    <f:validator validatorId="zipValidator"/>
</h:inputText>
```

Dakle na događaj onkeyup pozivamo script funkciju getContent(). Zip polju smo pridružili validator sa imenom zipValidator. Kada se prosledi zahtev za validacijom potrebno je dohvatiti validator koji je pridružen UIInputText komponenti. Zbog toga naš servlet mora da bude svestan view state-a što znači da u ovom slučaju zahtev mora da bude prosleđen faces servletu.

```
function getContent() {
    clearContent();
    var jsfState=getJSFState();
    var zip=document.getElementById("forma:zip").value;
    if(zip.length!=5) return;
    init();
    var url=window.document.forms[0].action;
    asyncRequest.onreadystatechange=stateChange;
    asyncRequest.open('POST', url+"?ajax=true&zip="+zip+
        "&javax.faces.ViewState="+jsfState, true);
    asyncRequest.send(null);
}
```

Da bi zahtev bio prosleđen faces servletu kao url navodimo

```
window.document.forms[0].action;
```

Ovaj url predstavlja akciju usamljene forme na strani. U stvari na ovaj način prevarili smo JSF jer se ovakav url shvata kao da je klijent poslao formu. Kao parametre zahteva šaljem zip kod, view state i jedan logički parametar ajax=true. Ovaj parametar ćemo ubrzo objasniti. Takođe zahtev mora da bude POST jer je samo tada dostupan view state koji nam je potreban. Pošto se view state čuva na klijentskoj strani funkcija getJSFState ga dohvata.

```
function getJSFState() {
    var state = window.document
        .getElementsByName("javax.faces.ViewState");
    var value = null;

    if(null != state && 0 < state.length) {
        value = state[0].value;
        var encodedValue = encodeURI(value);
        var re = new RegExp("\\+", "g");
        return encodedValue.replace(re, "%2B");
    }
}
```

Objasnimo sada detaljnije sam postupak validacije. Kada se generiše AJAX zahtev ulazimo u JSF životni ciklus. Nama u ovom slučaju nije potrebno da se prođe kroz svih šest faza ciklusa jer korisnik nije poslao formu. Sve što želimo je da proverimo da li je zip kod ispravan. Ovo može da se proveriti čim se završi restore view faza jer je tada obnovljena hijerarhija komponenti u memoriji. Zbog toga smo definisali naš

PhaseListener koji reaguje na PhaseEvent događaj nakon što se završi prva faza životnog ciklusa.

PhaseListener smo registrovali u faces-config.xml fajlu. Kod je sledeći.

```
<lifecycle>
    <phase-listener>listeners.MyPhaseListener</phase-listener>
</lifecycle>
```

Naredni kod je implementacija PhaseListenera.

```
public class MyPhaseListener implements PhaseListener {
    public void beforePhase(PhaseEvent arg0) {
    }
    public PhaseId getPhaseId() {
        return PhaseId.RESTORE_VIEW;
    }
    public void afterPhase(PhaseEvent pe) {
        FacesContext context = FacesContext.getCurrentInstance();
        //da li je ajax poziv pokrenuo jsf zivotni ciklus
        String ajaxParam =
            (String) context.getExternalContext().getRequestParameterMap().get("ajax");

        if ("true".equals(ajaxParam)) {
        ...
    }
}
```

Nakon Restore View faze poziva se metoda afterPhase našeg oslušivača. Prvo se dohvata ajax parametar zahteva i proverava da li je jednak true. Ovo je potrebno uraditi jer ne želimo da se metoda poziva pri svakom već samo pri AJAX zahtevu.

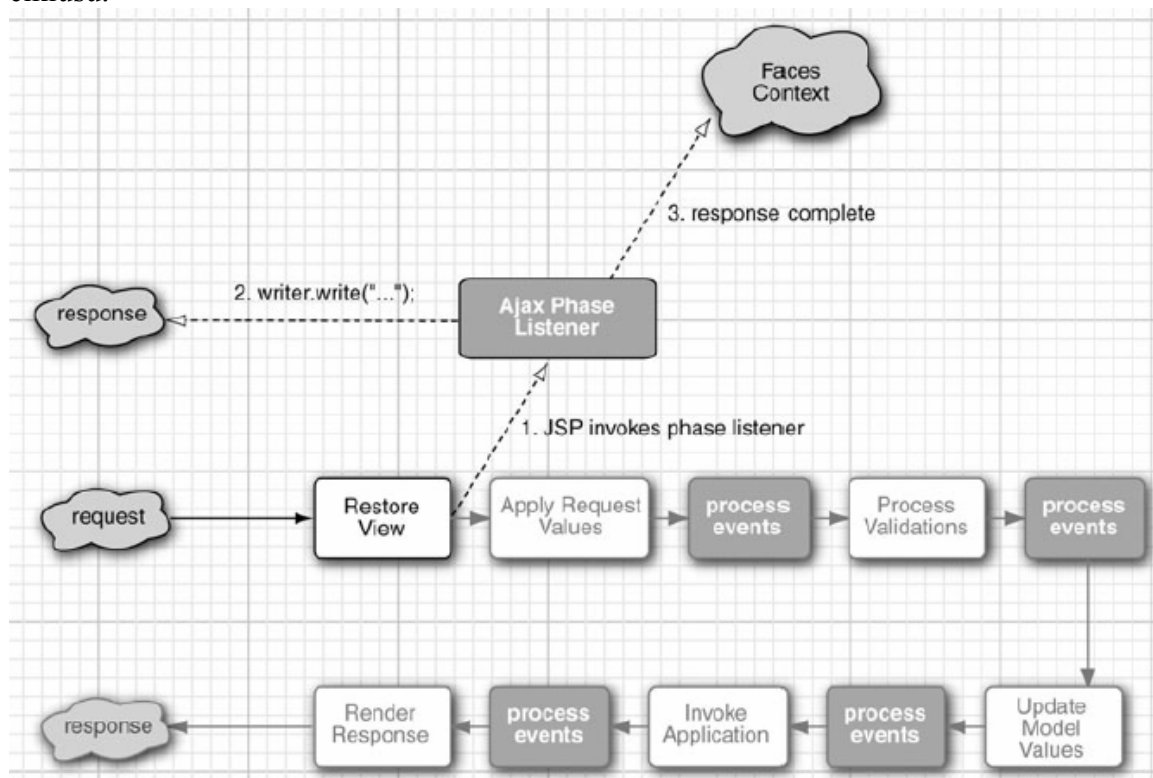
```
...
boolean valid = true;
//dohvatamo prosledjeni element forme zip
String zipParam =
    (String) context.getExternalContext().getRequestParameterMap().get("zip");
//dohvatamo UIInput komponentu kako bi dosli do validatora
UIViewRoot view = context.getViewRoot();
UIInput zip = (UIInput) view.findComponent("forma:zip");
if (zip != null) {
    HttpServletResponse response =
        (HttpServletResponse) context.getExternalContext().getResponse();
    Validator[] validators = zip.getValidators();
    PrintWriter writer=null;
    try {
        writer = response.getWriter();
    } catch (IOException ie) {
        ie.printStackTrace();
    }
}
```

Ako je AJAX zahtev dohvatamo zip parametar. Pozivamo metodu getExternalContext klase FacesContext a zatim dohvatamo mapu parametara i uzimamo parametar sa ključem zip. Pomoću metode klase FacesContext getViewRoot dobijamo koreni čvor stabla UIKomponenti. Pomoću metode findComponent klase UIViewRoot dohvatamo traženu komponentu. To je UIInputText polje sa zip kodom. Zatim pomoću

metode `getValidators()` dohvatamo sve validatore ove komponente. Ovaj metod imaju sve UI komponente koje implementiraju `EditableValueHolder` interfejs a klasa `UIInputText` ga implementira.

```
//validacija zip koda
for (int i = 0; i < validators.length; i++) {
    try {
        validators[i].validate(context, zip, zipParam);
    } catch (ValidatorException ve) {
        writer.write(ve.getMessage());
        valid=false;
        break;
    }
}
if(valid==true){ writer.write("okay"); }
} //kraj zip!=null
//prekidamo dalji zivotni ciklus i odlazimo u 6 fazu render response
context.responseComplete();
} //kraj ajax zahteva
```

Sada vršimo validaciju našeg zip polja. Prolazimo kroz sve validatore (u ovom slučaju samo jedan). Validatori bacaju `ValidatorException` ukoliko dođe do greške prilikom validacije. Ako dođe do greške kao odgovor se šalje poruka o grešci a ako je validacija prošla šaljemo string "okay". Nakon toga prekidamo životni ciklus i vraćamo se na istu stranu. Na slici 4.7 je prikazana uloga AJAX PhaseListener-a u životnom ciklusu.



Sl 4.7

Implementacija validatora je jednostavna. U `faces-config.xml` fajlu registrujemo validator.

```

<validator>
  <validator-id>zipValidator</validator-id>
  <validator-class>validators.ZipCodeValidator</validator-class>
</validator>

```

Metoda validate ZipCodeValidator-a proverava da li je zip 11000. U protivnom zip je neispravan i baca se ValidatorException. Poruka u grešci čuva se u messages.properties fajlu. Da bi došli do ove poruke koristimo statičku metodu getMessage klase Messages iz paketa UTIL čiju implementaciju nećemo objašnjavati. Ovo je urađeno kao podrška za internacionalizaciju a više detalja se može videti u knjizi "Prentice Hall-CoreJSF".

```

public class ZipCodeValidator implements Validator {

    public void validate(FacesContext context, UIComponent component, Object value)
        throws ValidatorException {
        if(!"11000".equals(value.toString())){
            throw new ValidatorException(Messages.getMessage("com.corejsf.messages", "badZip"));
        }
    }
}

```

Na kraju evo izgleda script funkcije stateChange.

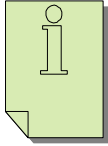
```

function stateChange(){
    if(asyncRequest.readyState==4 && asyncRequest.status==200){
        clearContent();
        if(asyncRequest.responseText!="okay"){
            var div=document.getElementById("output");

            div.innerHTML=
                "<img src='images/cancel_48.png' alt='slika neuspeh' \n\
                height='20px' width='20px' />" + asyncRequest.responseText;

        }
        else {
            var div=document.getElementById("output");
            div.innerHTML=
                "<img src='images/accepted_48.png' alt='slika uspeh' \n\
                height='20px' width='20px' />";
        }
        Fat.fade_element("output");
    }
}

```



Ovaj primer radi ispravno u Mozilla FireFox browser-u. U Internet Exploreru dolazi do greške ako reagujemo na oba događaja. Primer radi ako se generiše samo AJAX zahtev na peti unešeni znak ili na izlaz iz fokusa. Kada se AJAX zahtev generiše pomoću Prototype biblioteke primer i dalje ne radi lepo u Internet Explorer-u iako ova biblioteka garantuje da je kompatibilna sa svim browserima.

4.2.4 JSF-Rico komponenta << [Vidi opis.](#)

[4.2.4.1 Spinner](#)

[4.2.4.2 Kreiranje Tag klase za spinner](#)

[4.2.4.3 Kreiranje UISpinner klase](#)

[4.2.4.4 Kreiranje Accordion Komponente](#)

Ovo je najsloženiji primer jer se u njemu koriste custom komponente. Da bi se napravila custom komponenta potrebno je znati mnoge detalje o JSF klasama i samom JSF životnom ciklusu. JSF komponente je teško praviti ali kada se jednom naprave one postaju lake za korišćenje drugim programerima. U ovom primeru detaljnije ćemo objasniti kako je napravljena UI komponenta spinner i hibridna komponenta Accordion (ova komponenta enkapsulira rico.Accordion JavaScript komponentu). Komponentu TabbedPane nećemo objašnjavati jer bi to uzelo previše prostora. Ona je uvedena čisto radi vizuelnog efekta. Više detalja može se videti u Prentice Hall-Core JSF knjizi.

4.2.4.1 Spinner <<

Prvo ćemo detaljno objasniti kako je napravljena custom komponenta Spinner. Pre nego što kažemo koje su klase neophodne da bi napravili komponentu pokažimo upotrebu spinner komponente na JSP strani.

```
<%@ taglib prefix="corejsf" uri="/WEB-INF/biblioteka"%>
<corejsf:spinner id="spin" value="#{backingBean.panelHeight}"
    minimum="100" maximum="300" size="7" step="10"
    readonly="true"/>
```

Prvo smo napisali taglib direktivu sa uri-jem koji nas povezuje sa taglib bibliotekom. Sada na JSF stranu možemo da ubacimo komponentu spinner sa prefiksom corejsf. Spinner komponenta ima određen broj atributa kao što su id,value minimum itd. koji joj obezbeđuju određene funkcionalnosti. Ubacivanjem spinner taga na JSF stranu korisniku će se izrendati sledeća komponenta.

Sl 4.8

Dakle korisniku će se prikazati jedno tekst polje i dva dugmeta. Pritiskom na odgovarajuće dugme povećava se ili smanjuje vrednost u tekst polju. Naglasimo da je script koji to radi sakriven. Atribut step pokazuje za koliko će se promeniti vrednost.

Glavna prednost pravljenja komponenti je što programeri koji ih koriste ne moraju da znaju kako su komponente implementirane već samo treba da poznaju njihovu funkcionalnost i skup atributa koje komponenta poseduje.

Sada ćemo objasniti šta je sve potrebno da bi se napravila spinner komponenta.

Minimalno tag za JSF komponentu zahteva dve klase. Prva klasa je klasa koja procesira atribut taga. Ova klasa se po konvenciji završava sufiksom Tag. Mi smo je nazvali SpinnerTag. Druga klasa je sama komponenta koja ima dve glavne funkcije: da rendera komponentu i da procesira podatke koji dođu sa forme. Po konvenciji ova klasa ima UI prefiks. Mi smo je nazvali UISpinner.

Pomoću Tag klase se instancira komponenta i postavlja joj se atributi. Implementacija Tag klase je mehanička i uvek se radi na gotovo isti način. Komponenta može sama da rendera markup klijentu ili može da bude povezana sa klasom izvedenom iz klase Renderer koja će onda biti zadužena za renderanje komponente. Komponenta mora da bude izvedena iz klase UIComponent koja ima više od 40 apstraktnih metoda. Zato se obično komponente izvode iz klasa koje su već redefinisale ove metode. Komponente se najčešće izvode iz jedne od tri klase:

- UIInput
- UIOutput
- UICommand.

Naš spinner biće izveden iz UIInput klase.

4.2.4.2 Kreiranje Tag klase za spinner <<

Da bi definisali tag klasu prvo moramo da napišemo tld (tag library descriptor) fajl koji može da definiše više tagova i njihove attribute. Ovaj fajl se smešta u WEB-INF direktorijum. Prikazujemo deo ovog fajla.

```
<?xml version="1.0" encoding="UTF-8"?>
<taglib xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-jsptaglibrary_2_1.xsd"
  version="2.1">
  <description>A library containing a tabbed pane and spinner</description>
  <tlib-version>1.1</tlib-version>
  <short-name>biblioteka</short-name>
  <uri>/WEB-INF/biblioteka</uri>
```

...

Ime našeg tld fajla je biblioteka. Bitan tag je uri koji identifikuje našu taglib biblioteku. Vrednost ovog taga će se koristiti na JSP stranama u taglib direktivi. U našoj biblioteci definišemo tag spinner koji ćemo koristiti na JSP stranama i njemu odgovarajuću tag klasu.

...

```
<tag>
  <name>spinner</name>
  <tag-class>com.corejsf.SpinnerTag</tag-class>
```

...

Većina atributa imaće sledeći oblik:

```
<attribute>
  <description>The spinner minimum value</description>
  <name>minimum</name>
  <deferred-value>
    <type>int</type>
  </deferred-value>
</attribute>
```

Definisali smo atribut minimum. Tag deferred-value je uveden u JSF 1.2 i označava da će vrednost ovog atributa moći da bude value expression. Vrednost atributa moći će da bude konstantan string ili string oblika #{...}. To znači da ćemo atribut minimum moći da povežujemo sa proprietijama bina koji moraju da budu tipa int.

Pored value expression-a neki atributi povezuju se sa metodama(method expression). Npr spinner-u može da se pridruži valueChangeListener.

```
<attribute>
  <name>valueChangeListener</name>
  <deferred-method>
    <method-signature>void valueChange(javax.faces.event.ValueChangeEvent)
  </method-signature>
  </deferred-method>
</attribute>
```

Postoje četiri atributa koji se mogu povezivati sa metodama. Pošto je spinner UIInput komponenta on može da ima attribute validator i valueChangeListener. Pored toga postoje i atributi action i actionListener koji se mogu pridruživati npr. UICommand klasama. Ubacivanje ovih atributa u tld je šablonsko i jedino se razlikuje potpis metoda. Naš atribut value changeListener može da se povezuje sa void funkcijama koje kao argument imaju ValueChangeEvent.

Sada ćemo opisati SpinnerTag klasu. SpinnerTag klasa se izvodi iz klase UIComponentELTag.

```
public class SpinnerTag extends UIComponentELTag {
...
}
```

Ova klasa ima pet odgovornosti:

- Da identifikuje tip komponente (kako je komponenta registrovana u faces-config.xml)
- Da identifikuje tip renderera za komponentu (kako je renderer registrovan u faces-config.xml)
- Da obezbedi setter metode za sve tag attribute
- Da upiše vrednosti tag atributa u komponentu (u njenu mapu atributa)
- Da oslobodi resurse

Za identifikaciju tipa komponente i renderera koriste se dve metode.

```
...
public String getComponentType() {
    return "spinner";
}

public String getRendererType() {
    return null;
}
...
```

Metoda getComponentType vraća string koji predstavlja tip komponente. Tip komponente je definisan u faces-config.xml fajlu.

```
<component>
    <component-type>spinner</component-type>
    <component-class>com.corejsf.UISpinner</component-class>
</component>
```

Pošto spinner klasa rendra samu sebe, metoda getRendererType vraća vrednost null.

Za svaki atribut spinner-a u SpinnerTag klasi moramo da imamo polje tipa ValueExpression. Izuzetak su atributi id, binding i rendered koje obrađuje superklasa. Polje valueChangeListener je tipa MethodExpression.

```
private ValueExpression minimum;
private ValueExpression maximum;
private ValueExpression size;
private ValueExpression value;
private ValueExpression readonly;
private ValueExpression step;
private MethodExpression valueChangeListener = null;
```

Za svaki od atributa bitno je da se napiše setter metod. Get metode nisu potrebne.

```
...
public void setMinimum(ValueExpression minimum) {
    this.minimum = minimum;
}
...
```

Kada se tag procesira na jsp strani onda se vrednost svakog atributa konvertuje u ValueExpression i poziva se setter metoda.

Za upis vrednosti atributa u komponentu koristi se metoda `setProperties`.

```
public void setProperties(UIComponent component) {
...
    public void setProperties(UIComponent component) {
        super.setProperties(component);
        component.setValueExpression("size", size);
        component.setValueExpression("minimum", minimum);
        component.setValueExpression("maximum", maximum);
        component.setValueExpression("value", value);
        component.setValueExpression("readonly", readonly);
        component.setValueExpression("step", step);
        if (valueChangeListener != null) {
            ((EditableValueHolder) component).addValueChangeListener
                (new MethodExpressionValueChangeListener(valueChangeListener));
        }
    }
...
}
```

Prvo se poziva `setProperties` metoda super klase kako bi se setovali atributi `id`, `binding` i `rendered`. Metoda `setValueExpression` upisuje vrednost atributa u komponentu. Prvi argument metode je string koji predstavlja ime atributa. Drugi argument je tipa `ValueExpression` i može da bude običan string ili string oblika `#{...}`. U zavisnosti od oblika ovog stringa atribut se upisuje u jednu od dve mape komponente (mapa atributa ili mapa `ValueExpression`-a). Ovaj postupak je uveden u JSF 1.2 i automatizuje setovanje atributa komponente. U verziji JSF 1.1 svi atributi taga su bili stringovi a ne `ValueExpression` objekti. U metodi `setProperties` programer je sam morao da proveri da li je ovaj string konstanta ili oblika `#{...}`. U zavisnosti od vrednosti stringa programer je morao da smesti atribut u odgovarajuću mapu.

Za svaku od četiri vrste atributa koji mogu da se povežu sa metodama programer sledi šablon prikazan u tabeli.

Attribute Name	Method-Signature Element in TLD	Code in setProperties Method
valueChangeListener	void valueChange(javax.faces.event.ValueChangeEvent)	((EditableValueHolder) component).addValueChangeListener(new MethodExpressionValueChangeListener(expr));
validator	void validate(javax.faces.context.FacesContext, javax.faces.component.UIComponent, java.lang.Object)	((EditableValueHolder) component).addValidator(new MethodExpressionValidator(expr));
actionListener	void actionListener(javax.faces.event.ActionEvent)	((ActionSource) component).addActionListener(new MethodExpressionActionListener(expr));
action	java.lang.Object action()	((ActionSource2) component).addAction(expr);

Metoda za oslobađanje resursa je `release` i ima sledeći oblik:

```
public void release() {  
    // always call the superclass method  
    super.release();  
    minimum = null;  
    maximum = null;  
    size = null;  
    value = null;  
}
```

4.2.4.3 Kreiranje `UISpinner` klase <<

Klasa `UISpinner` izvedena je iz klase `UIInput`. Komponente generišu markup (u našem slučaju HTML) u 3 metode:

- `encodeBegin()`
- `encodeChildren()`
- `encodeEnd()`

Ove metode se pozivaju u šestoj fazi životnog ciklusa kada treba izrendati komponentu. Uvek se prvo pozove metoda `encodeBegin`. Metoda `encodeChildren` se poziva samo ako metoda `getRendersChildren` vrati vrednost `true`. Ova metoda podrazumevano vraća vrednost `false`. Posle `encodeChildren` metode poziva se `encodeEnd` metoda. Pošto spinner nema decu sav kod za rendanje je smešten u `encodeBegin` metodu.

Sada ćemo opisati metodu `encodeBegin`.

```
public void encodeBegin(FacesContext context) throws IOException {  
    //dohvatamo atribut spinera  
    ResponseWriter writer = context.getResponseWriter();  
    Integer min = (Integer) getAttributes().get("minimum");  
    Integer max = (Integer) getAttributes().get("maximum");  
    Integer size = (Integer) getAttributes().get("size");  
    Boolean readonly = (Boolean) getAttributes().get("readonly");  
    Integer step = (Integer) getAttributes().get("step");  
    if (step == null) {  
        step = new Integer(10);  
    }  
    String clientId = getClientId(context);  
    String id = getId();  
    ...  
}
```

Prvo smo preko metode klase `FacesContext` `getResponseWriter` dohvatili `writer` koji ćemo koristiti za rendanje HTML-a. Zatim dohvatamo atribut koji će nam biti potrebni prilikom rendanja spinera. Ovi atributi možda ne postoje u mapi ako se nisu koristili u tagu na JSP strani. U tom slučaju odgovarajuće promenljive imaju vrednost `null`. Metod `getAttributes` naše komponente vraća mapu atributa. Ako je atribut `step` `null` dajemo mu podrazumevanu vrednost 10. Pomoću metode `getClientId` dobijamo id naše

spinner komponente oblika id_forme:id_komponente. Metod getId vraća id komponente bez id-a forme.

Spinner komponenta rendi i script funkciju.

```
writer.write("<script type=\"text/javascript\">");
writer.write("function promena" + id + "(increment)");
if (min != null) {
    writer.write("var min=" + min.intValue() + ";");
} else {
    writer.write("var min=-100;");
}
if (max != null) {
    writer.write("var max=" + max.intValue() + ";");
} else {
    writer.write("var max=100;");
}
writer.write("var spinner=document.getElementById(\"" + clientId + "\");");
writer.write("var vr=parseInt(spinner.value);");
writer.write("if(isNaN(vr))return;");
writer.write("vr+=increment;");
writer.write("if(vr<min) return;");
writer.write("if(vr>max) return;");
writer.write("spinner.value=vr; ");
writer.write("</script>");
```

Pošto na jednoj JSP strani može da se nađe više spinner komponenti svaka od njih rendi funkciju koja se zove promena ali joj dodaje sufiks id. Ovo je urađeno da se na istoj strani ne pojavi više script funkcija sa istim imenom. Funkcija promena biće pozvana svaki put kada se klikne na jedno od dva dugmeta spinnera. Ova funkcija služi da uveća ili umanja vrednost tekst polja spinnera za vrednost atributa step. Ako su atributi minimum ili maximum definisani definišemo promenljive funkcije min i max sa vrednostima atributa. Ako nisu definisani dajemo im podrazumevane vrednosti. Zatim dohvatamo vrednost koja se nalazi u tekst polju spinner komponente. Vidimo da je ime te komponente isto kao clientId našeg spinner-a. Videćemo ubrzo da je tekst polje izrendano tako da dobije baš ime koje je jednako clientId-u. Nakon provere da li je vrednost tekst polja broj, uvećavamo ili umanjujemo njegovu vrednost za increment ali samo pod uslovom da se ne pređu dozvoljene granice. Argument funkcije increment odgovaraće atributu step spinner komponente.

Sada ćemo pokazati kako se rendi tekst polje i jedno od dva dugmeta.

...

```

writer.startElement("input", this);
writer.writeAttribute("type", "text", null);
writer.writeAttribute("name", clientId, null);
writer.writeAttribute("id", clientId, null);
Object v = getValue();
if (v != null) {
    writer.writeAttribute("value", v.toString(), null);
}
if (size != null) {
    writer.writeAttribute("size", size, null);
}
if (readonly != null) {
    writer.writeAttribute("readonly", readonly.booleanValue(), null);
}
writer.endElement("input");

```

...

Da ne bi direktno u metodi write našeg writer-a pisali HTML kod koristimo pomoćne metode. Sa metodom startElement otvaramo input element (<input>). Metoda writeAttribute služi za definisanje parova ime i vrednost atributa. Vidimo da je tip input objekta tekst polje a ime odgovara clientId-u naše komponente. Tekst polje je dobilo i atribut id zbog metode koju koristimo u script funkciji. (getElementById). Ako su definisani atributi size i readOnly spinner-a koristimo ih da definišemo veličinu tekst polja i da li je ono samo za čitanje. Ako naša spinner komponenta ima vrednost upisujemo je u tekst polje. Metoda getValue vraća Object. Ona prvo proverava da li komponenta ima lokalnu vrednost (tipa Integer). Ako je ne nađe gleda se da li je atribut value spinner komponente vezan za property bina. Sa metodom endElement završavamo input element.

Evo i koda za jedno od dva dugmeta spinera:

...

```

writer.startElement("input", this);
writer.writeAttribute("type", "button", null);
writer.writeAttribute("value", "<", null);
writer.writeAttribute("onclick", "promena" + id + "(-" + step.intValue() + ")", null);
writer.endElement("input");

```

...

Sada ostaje da objasnimo drugu važnu funkciju svake komponente. Kada se u drugoj fazi životnog ciklusa uzimaju vrednosti koje su stigle sa forme svaka komponenta je odgovorna da pokupi vrednost koja joj odgovara. Sa forme samo stižu parovi ime i vrednost elementa forme. Komponenta ima pamet da zaključi koja joj vrednost pripada.

...

```

public void decode(FacesContext context) {
    Map requestMap = context.getExternalContext().getRequestParameterMap();
    String clientId = getClientId(context);
    setSubmittedValue((String) requestMap.get(clientId));
    setValid(true);
}

```

...

Konverzija vrednosti koju smo upisali u spinner vrši se u trećoj fazi životnog ciklusa. Treba obratiti pažnju da se životni ciklus neće pokrenuti pritiskom na dugmiće spinner-a jer oni pozivaju script funkciju. Kada klijent pošalje formu on može da upiše u polje spinner-a i vrednost koja nije broj. Konverzija se vrši automatski u trećoj fazi tako što se proverí da li spinner ima pridružen konverter. Ako nema pridružen konverter pokušaće se na osnovu tipa bina za koji je vezan value spinner-a kreiranje konvertera. Mi smo spinner-u dodelili IntegerConverter u njegovom konstruktoru.

```
...
public UISpinner() {
    setConverter(new IntegerConverter());
    setRendererType(null);
}
...
```

4.2.4.4 Kreiranje Accordion komponente <<

Sada smo detaljno objasnili kako se pravi spinner komponenta. Da bi objasnili kako smo napravili hibridnu Rico-Accordion komponentu pokažimo kako bi napravili Accordion komponentu na jednoj JSP strani. Zatim ćemo objasniti kako da napravimo JSF komponentu koja enkapsulira Rico-Accordion JS komponentu.

```
<html>
<head>
<link href="styles.css" rel="stylesheet" type="text/css"/>
<script type='text/javascript' src='prototype.js'></script>
<script type='text/javascript' src='rico-1.1.2.js'></script>
<script type='text/javascript'>
function createAccordion() {
new Rico.Accordion($("#theDiv")); //od spoljašnjeg div pravimo accordion
}
</script>
</head>
<body onload="createAccordion();">
//kada se učita JSP stran poziva se script funkca za pravljenje accordiona
<div id="theDiv" class="accordion">
//Spolašnji div u koji se smešta drugi div
<div class="accordionPanel">
// unutrašnji div Panel!
<div class="accordionPanelHeader">// heder panela
Fruits
</div>
<div class="accordionPanelContent">// sadržaj panela
<ul>
<li>Oranges</li>
<li>Apples</li>
<li>Watermelon</li>
<li>Kiwi</li>
</ul>
</div>
</div>
```

```

<div class="accordionPanel">//novi panel
<div class="accordionPanelHeader">//heder panela
Vegetables
</div>
<div class="accordionPanelContent">//sadržaj panela
<ul>
<li>Radishes</li>
<li>Carrots</li>
<li>Spinach</li>
<li>Celery</li>
</ul>
</div>
</div>
</div>
</body>
</html>

```

Dakle da bi napravili Rico.Accordion definišemo jedan div element kome je obavezno dati id. Ovaj div element će u script funkciji biti prosleđen Rico biblioteci da instancira Accordion. Na ovaj način naš div će dobiti specijalne efekte. U div zatim može da se stavi jedan ili više div elemenata koji predstavljaju jedan panel Accordion komponente. Ovaj unutrašnji div sadrži dva div elementa. Div za heder panela i div za sadržaj panela. Sadržaj panela može da bude bilo šta. To smo u ovom primeru iskoristili da u panele stavljamo druge komponente kao što je npr. spinner.

Sada ćemo objasniti kako smo napravili našu komponentu koja enkapsulira Rico.Accordion. Detalje oko izrade tag klasa i tld biblioteke nećemo objašnjavati jer one prate isti šablon kao kod spinner komponente. Reći ćemo samo koje atribute ima naša komponenta.

Da bi napravili Accordion komponentu mi smo u stvari definisali dve korisničke komponente. Jedna je Accordion a druga AccordionPanel komponenta. Ove komponente imaju svoje renderer klase tako da implementacija nije potrebna. Sve što treba je da navedemo da su naše komponente izvedene iz odgovarajućih UI komponenti. Evo kako u faces-config.xml fajlu definišemo naše komponente i njihove renderer klase.

```

<component>
  <component-type>com.corejsf.Accordion</component-type>
  <component-class>javax.faces.component.UINamingContainer</component-class>
</component>
<component>
  <component-type>com.corejsf.AccordionPanel</component-type>
  <component-class>javax.faces.component.UIPanel</component-class>
</component>

```

```

<render-kit>
  <renderer>
    <component-family>javax.faces.NamingContainer</component-family>
    <renderer-type>com.corejsf.AccordionRenderer</renderer-type>
    <renderer-class>com.corejsf.AccordionRenderer</renderer-class>
  </renderer>
</render-kit>
<render-kit>
  <renderer>
    <component-family>javax.faces.Panel</component-family>
    <renderer-type>com.corejsf.AccordionPanelRenderer</renderer-type>
    <renderer-class>com.corejsf.AccordionPanelRenderer</renderer-class>
  </renderer>
</render-kit>

```

Tag `component-family` definiše familiju komponenti za koju se definiše renderer. Renderer nikad ne definišemo za individualnu komponentu već za čitavu familiju. Familije komponenti su određene prema standardnim klasama. Npr za klasu `UICommand` definišemo `Command` familiju renderera a za `UIInput` klasu `Input` familiju renderera. Renderer za klasu `Accordion` treba da izrenda spoljašnji div. U njega stavljamo na jsp strani tagove za `AccordionPanel`. Renderer ove klase treba da izrenda unutrašnji div za panel. U ovaj tag na JSP strani stavljamo proizvoljan sadržaj. Sada ćemo prikazati deo strane `kombinovanjeKomponenti.jsp`

```

<@ taglib prefix="rico" uri="/WEB-INF/rico" %>
<@ taglib prefix="corejsf" uri="/WEB-INF/biblioteka"%>
...
<rico:accordion name="bookAccordionTwo" binding="#{backingBean.accordion}"
  contentClass="accordionPanelContent"
  headerClass="accordionPanelHeader"
  panelClass="accordionPanel"
  styleClass="accordion"
  panelHeight="175"
  >
...

  <rico:accordionPanel heading="Spinner za podešavanje visine panela">
    <h:form>
      <corejsf:spinner id="spin" value="#{backingBean.panelHeight}"
        minimum="100" maximum="300" size="7" step="10"
        readonly="true"/>

      <br/>
      <h:message for="spin"/>
      <h:commandButton value="#{msgs.newHeight}"
        actionListener="#{backingBean.changeAccordionHeight}"/>

    </h:form>
  </rico:accordionPanel>
</rico:accordion>
...

```

U accordion komponentu stavili smo accordionPanel komponentu. Vidimo i jedan broj atributa accordion komponente. Atribut name je obavezan atribut jer on mora da se prosledi Rico biblioteci. Evo kako je u rico tld biblioteci definisan atribut name.

```
<attribute>
  <name>name</name>
  <required>true</required>
  <deferred-value>
    <type>java.lang.String</type>
  </deferred-value>
</attribute>
```

Atribut binding služi da komponentu poveže sa property-em bina backingBean.accordion. Ovaj property je referenca na klasu UINamingContainer kojoj pripada accordion. Atributi styleClass, panelClass, contentClass i headerClass su css stilovi koje ćemo koristiti kada rendamo odgovarajuće div elemente. panelHeight atribut služi za definisanje visine panela. accordionPanel komponenta ima obavezan atribut heading što će biti naslov koji će se pojaviti u heder div elementu. U accordionPanel stavili smo spinner komponentu koja će se prikazati kao deo panela.

Sada ćemo objasniti kako se rendaa accordion a kako accordionPanel. Rekli smo da ove klase imaju svoje klase za rendanje. Metode za rendanje sada se pišu u ovim klasama kao i decode metoda ako je potrebna. Razlika je samo u tome što metode imaju i dodatni argument a to je referenca na komponentu.

Pogledajmo kod za AccordionRenderer.

```
public class AccordionRenderer extends Renderer {

    public void encodeBegin(FacesContext context, UIComponent component)
        throws IOException {

        ResponseWriter writer = context.getResponseWriter();
        processIncludeScripts(writer);
        writeEnclosingDiv(component, writer);
    }
}
```

...

Metoda processIncludeScripts služi da uključimo odgovarajuće script biblioteke na našu JSP stranu. Time u potpunosti od programera sakrivamo da upotreba accordion komponente u pozadini poziva script za kreiranje Rico.Accordion-a.

...

```
//ubacujemo script biblioteke
public void processIncludeScripts(ResponseWriter writer) throws IOException {

    writer.write("<script type=\"text/javascript\" src=\"prototype.js\"></script>");
    writer.write("<script type=\"text/javascript\" src=\"rico-1.1.2.js\"></script>");
    writer.write("<script type=\"text/javascript\" src=\"scriptaculous.js\"></script>");

}

...
```


Metoda `writeEnclosingDiv` služi da se izrenda spoljašnji div element.

```
...
public void writeEnclosingDiv(UIComponent component, ResponseWriter writer)
    throws IOException {
    String id = (String) component.getAttributes().get("name");
    String styleClass = (String) component.getAttributes().get("styleClass");
    //id je obavezan koristi se da bi se rico biblioteci prosledio kao argument!
    writer.write("<div id='" + id + "' ");
    if (styleClass != null) {
        writer.write("class='" + styleClass + "'>");
    }
}
```

Spoljašni div dobija atribut `name` accordion komponente. Accordion komponenta ima deca. To su accordion paneli. Ova deca se rendaju sama nakon što se završi `encodeBegin` metoda a onda se poziva `encodeEnd` metoda renderera za accordion.

```
...
public boolean getRendersChildren() {
    return false;
}

public void encodeEnd(FacesContext context, UIComponent component) throws IOException {
    ResponseWriter writer = context.getResponseWriter();
    writer.write("</div>");
    String divName = (String) component.getAttributes().get("name");
    writer.write("<script type='text/javascript'>");
    writer.write("new Rico.Accordion($(' " + divName + " ' ),");
    writer.write("{");
    String panelHeight = (String) component.getAttributes().get("panelHeight");
    if (panelHeight == null) panelHeight = "100";
    writer.write("    panelHeight: " + panelHeight);
    writer.write(" } ");
    writer.write(");");
    writer.write("</script>");
}
```

Kada se završi rendanje dece accordion komponente poziva se `encodeEnd` metoda. Prvo se zatvori div za accordion a zatim moramo da definišemo script u kome Rico biblioteci prosleđujemo ime našeg div elementa i visinu panela.

Sada ćemo pokazati `encodeBegin` i `encodeEnd` metodu `AccordionPanelRenderer` klase. Pošto se u ovu komponentu smešta bilo kakav HTML kod prvo se poziva `encodeBegin` metoda a tek kada se izrendaju deca komponente poziva se `encodeEnd` metoda.

...


```

public void encodeBegin(FacesContext context, UIComponent component) throws IOException {

    String panelClass = (String) component.getParent().getAttributes().get("panelClass");
    String headerClass = (String) component.getParent().getAttributes().get("headerClass");
    String contentClass = (String) component.getParent().getAttributes().get("contentClass");
    String heading = (String) component.getAttributes().get("heading");
    ResponseWriter writer = context.getResponseWriter();
    //div za panel
    writer.write("<div ");
    if (panelClass != null) {
        writer.write("class='" + panelClass + "'");
    }
    writer.write(">");
    //div heder panela
    writer.write("<div ");
    if (headerClass != null) {
        writer.write("class='" + headerClass + "'");
    }
    writer.write(">");
    writer.write(heading);
    writer.write("</div>");
    //kraj div heder panela pocetak div content panela
    writer.write("<div ");
    if (contentClass != null) {
        writer.write("class='" + contentClass + "'");
    }
    writer.write(">");
}
...

```

Preko getParent metode komponente dolazimo prvo do roditelja naše komponente a to ja accordion komponenta. Ona nam je potrebna da bi uzeli odgovarajuće css atribute. Zatim se rend a heder div panel element i u njemu heder div i div za sadržaj. Pošto u div za sadržaj idu deca ne zatvaramo panel div i div za sadržaj. Metoda encodeEnd samo zatvara ova dva div elementa i nećemo je navoditi.

Obratimo sada pažnju na još jedan detalj sa JSP strane kombinovanjeKomponenti.jsp. U accordionPanel stavili smo formu u kojoj se nalazi spinner i jedno komandno dugme. Atribut value spinner-a vezan je za property backinBean.panelHeight a ActionListener atribut komandnog dugmeta za property backingBean.changeAccordionHeight. Ovaj metod služi da se na osnovu vrednosti koja se nalazi u tekst polju spinner-a promeni visina panela našeg accordion panela.

Evo i dela koda BackingBean klase.

```

public class BackingBean {

    private int panelHeight=175;
    private UINamingContainer accordion;
    public void changeAccordionHeight(ActionEvent ae){

        accordion.getAttributes().put("panelHeight", panelHeight+"");
    }
    ...




```

Ne zaboravimo da smo našu accordion komponentu vezali za property bina accordion.

4.2.5 Korišćenje Ajax4JSF framework-a za popunjavanje forme << [Vidi opis.](#)

Ovaj primer je sa funkcionalne strane isti kao primer 3. Na osnovu vrednosti koja se stavi u zip tekst polje u poljima za grad i državu se upisuje sadržaj bez osvežavanja strane. Razlika je u tome kako je primer implementiran tj. kako se kreira AJAX zahtev. Cilj ovog i naredna dva primera je da se upoznamo sa AJAX4JSF framework-om. AJAX4JSF framework nudi 18 tagova koji se mogu ubacivati na JSP strane. Pomoću ovih tagova JSF komponentama se lako dodaje AJAX funkcionalnost. U ovom primeru videćemo upotrebu a4j:support taga.

Da bi koristili AJAX4JSF framework potrebno je u projekat ubaciti odgovarajuće biblioteke. U WEB-INF folderu naše aplikacije treba napraviti lib folder i u njega staviti tri jar fajla. Ti fajlovi su:

-  richfaces-api.jar
-  richfaces-impl.jar
-  richfaces-ui.jar.

Ove jar fajlove je potrebno uključiti u projekat. U NetBeans okruženju desnim klikom na ime projekta otvara se meni u kome se izabere opcija properties. Zatim se ode na kategoriju libraries i izabere opcija Add Jar/Folder. Sada samo treba dodati jar fajlove iz lib foldera naše aplikacije.

Da bi koristili AJAX4JSF tagove na jsp strani moramo da dodamo sledeći sadržaj u web.xml fajl.

```
<filter>
  <display-name>RichFaces Filter</display-name>
  <filter-name>richfaces</filter-name>
  <filter-class>org.ajax4jsf.Filter</filter-class>
</filter>
<filter-mapping>
  <filter-name>richfaces</filter-name>
  <servlet-name>Faces Servlet</servlet-name>
  <dispatcher>REQUEST</dispatcher>
  <dispatcher>FORWARD</dispatcher>
  <dispatcher>INCLUDE</dispatcher>
</filter-mapping>
```

Na JSP strani uvodimo sledeću taglib direktivu:

```
<%@ taglib uri="http://richfaces.org/a4j" prefix="a4j" %>
```

Sada se AJAX4JSF tagovi mogu koristiti na JSP strani sa prefiksom a4j.

Deo stranice A4JPrimerZip.jsp dat je u daljem tekstu.

...

```

<h:inputText id="zip" size="5" value="#{zipBean.zip}" maxlength="5">
    <a4j:support event="onfocus"
        actionListener="#{zipBean.clearCityAndState}"
        reRender="city, state"/>

    <a4j:support event="onblur"
        actionListener="#{zipBean.setCityAndStateForZip}"
        reRender="city, state"

    />
</h:inputText>

<h:outputText value="#{msgs.ex2City}"/>
<h:inputText id="city" value="#{zipBean.city}"/>

<h:outputText value="#{msgs.ex2Country}"/>
<h:inputText id="state" value="#{zipBean.state}"/>

```

...

Tekst polju za zip kod dodali smo dva a4j:support taga. Prvi a4j:support tag dodaje AJAX funkcionalnost tako što pošalje zahtev serveru kada polje dobije fokus. Poziva se faces servlet tako da se ulazi u JSF životni ciklus. AJAX4JSF ugrađuje i actionListener u životni ciklus. U petoj fazi životnog ciklusa pozvaće se metoda zipBean.clearCityAndState. Atribut reRender pokazuje da treba ponovo izrendati ova dva tekst polja. Drugi a4j:support tag radi gotovo identičnu stvar. Kada se izađe iz fokusa polja generiše se AJAX zahtev i ulazi u JSF životni ciklus. U petoj fazi ciklusa pozvaće se metoda zipBean.setCityAndState.

Evo i izgleda ove dve metode klase ZipBean.

```

public void clearCityAndState(ActionEvent e) {
    setCity("");
    setState("");
}

public void setCityAndStateForZip(ActionEvent e) {
    if (zip.equals("11000")) {
        setCity("Beograd");
        setState("Srbija");
    }
    else if (zip.equals("21000")) {
        setCity("Novi Sad");
        setState("Srbija");
    }
    else {
        setCity("Nema podataka");
        setState("Nema podataka");
    }
}

```

4.2.6 Korišćenje Ajax4JSF framework-a za realtime validaciju << [Vidi opis.](#)

Ovaj primer pokazuje kako smo uz pomoć AJAX4JSF framework-a implementirali realtime validaciju. Pogledajmo najpre deo JSP strane

```
<h:panelGrid columns="2" styleClass="heading">
    <body>
...
<a4j:status id="status">
    <f:facet name="start">
        <h:graphicImage value="images/indicator_radar.gif"/>
    </f:facet>
</a4j:status>
...
</h:panelGrid>
```

Uveli smo a4j:support tag. Kada se desi AJAX zahtev iniciran od strane AJAX4JSF-a prikazaće se sadržaj faceta-a sa imenom start a kada se zahtev završi facet-a sa imenom stop. Mi smo uveli samo start facet da bi se kada se generiše zahtev na ekranu videla animacija.

```
<h:form id="form">
    <h:panelGrid columns="2">
        <h:outputLabel for="zip"
            value="#{msgs.ex2Zip}"/>
        <h:panelGroup>
            <h:inputText id="zip"
                size="5"
                value="#{bb.zip}"
                required="true" maxLength="5">
                <f:validateLength minimum="5"
                    maximum="5"/>
                <f:validator
                    validatorId="ZipcodeValidator2"/>

            <a4j:support event="onblur"
                immediate="true"
                actionListener="#{bb.validateZip}"
                reRender="city, state, errorMessage"/>
        </h:inputText>
    </h:panelGrid>
</h:form>
```

Polju za zip kod dodali smo AJAX podršku. Kada polje izgubi fokus pozvaće se metoda bb.validateZip. Samo tekst polje ima dodata tri validatora. Prvo atribut required postavljen na true govori da je neophodno upisati neku vrednost u polje. Dalje dodali smo JSF validator validateLength (LengthValidator) koji proverava da li je uneto tačno pet znakova u zip polje. Treći validator je naš ZipcodeValidator2.čija je implementacija jednostavna i izostavićemo je. Ovaj validator proverava da li je uneti zip kod u bazi podataka i ako nije baca ValidatorException. Ono što se osvežava su polja grad,država i poruka o grešci.

Ostao je da se objasni atribut immediate AJAX4JSF taga. Bez ovog atributa primer ne bi radio. Iako je naizgled dodavanje AJAX podrške jednostavno programer

ipak mora da bude svestan JSF životnog ciklusa. Metoda `bb.validateZip` će se izvršiti u petoj fazi životnog ciklusa. Međutim validacija se izvršava u trećoj fazi životnog ciklusa. Ako validacija ne uspe nakon treće faze prekida se životni ciklus i odmah se vraćamo na stranu na kojoj se desila greška. To znači da se naša metoda ne bi ni izvršila. Atribut `immediate` govori da naša metoda treba da se izvrši što je pre moguće a to znači u drugoj fazi životnog ciklusa kada se pokupi vrednost zip koda sa forme. U ovoj metodi se proverava da li je zip kod ispravan a onda se preskaču treća, četvrta i peta faza životnog ciklusa i zatim se rendera ista strana.

Prikazan je bitan ostatak jsp strane. Tag `h:messages` služi da prikaže sve greške, ako se dese, kada klijent pritisne submit dugme kako bi poslao formu. Pošto se tada pokreće životni ciklus, može da dođe do greške u validaciji i o tome želimo da obavestimo korisnika. Takođe primetimo da su i tekst polje za grad i tekst polje za državu povezani binding atributom sa binom.

```
...
        <h:outputText id="errorMessage"
            value="#{bb.errorMessage}"
            style="color: red; font-style: italic;"/>
    </h:panelGroup>

    <h:outputLabel for="city"
        value="#{msgs.ex2City}"/>
    <h:inputText id="city" binding="#{bb.cityInput}"
        size="25"
        value="#{bb.city}"/>

    <h:outputLabel for="state"
        value="#{msgs.ex2Country}"/>
    <h:inputText id="state" binding="#{bb.stateInput}"
        size="25"
        value="#{bb.state}"/>

    <h:commandButton id="submit"
        value="#{msgs.submitButtonText}"
        action="showBackingBean"/>
    <h:commandButton value="#{msgs.button1Ex2}"
        action="nazad" immediate="true"/>

    </h:panelGrid>

    <h:messages layout="list" style="color: red; font-style: italic;"/>
</h:form>
...
```

Sada ćemo prikazati deo koda koji se odnosi na sam bin.

```

public class BackingBean2 {

    private String zip;
    private String city;
    private String state;
    private String errorMessage;
    private UIInput cityInput = null;
    private UIInput stateInput = null;
    ...
    public void validateZip(ActionEvent ae) {

        try {
            Thread.sleep(300); //kako bi se videla animacija

        } catch (Exception e) {
            e.printStackTrace();
        }

        UIInput input = (UIInput) ae.getComponent() //ajax4jsf komponenta!!!
            .getParent(); //nasa zip input komponenta

        if (input != null) {

            String zip = (String) input.getSubmittedValue();
            if (zip != null) {

                setCityAndState(zip);
                FacesContext context = FacesContext.getCurrentInstance();
                input.validate(context);
                if (!input.isValid()) {
                    setErrorMessage(context, input);
                }
                cityInput.setSubmittedValue(null);
                stateInput.setSubmittedValue(null);

            }
        }
    }
    ...
}

```

U drugoj fazi životnog ciklusa pozvaće se validateZip metoda. Metoda Thread.sleep se koristi da bi uspavali nit kako bi se videla animacija dok traje AJAX zahtev. Kada je ne bi uspavali zahtev bi se obradio suviše brzo i ne bi videli animaciju. Prvo moramo da dodemo do naše input komponente. Pošto je događaj ispalila AJAX4JSF komponenta metod getComponent ActionEvent klase vraća AJAX4JSF objekat čiji je roditelj naša input komponenta. Zatim uzimamo vrednost koja je upisana u zip polje i koja se nakon Apply Request Values faze može dohvatiti preko getSubmittedValue metode naše komponente. Metoda setCityAndState će kontaktirati bazu podataka i na osnovu zip koda setovati city i state polja bina. Nakon toga pozivamo metod validate

naše komponente. Ovde će se pozivati validacija za svaki od validatora pridružen našoj komponenti. Ako validacija uspe metoda isValid komponente vratiće vrednost true. Ako metoda vrati vrednost false pozivamo metodu bina setErrorMessage da setuje polje errorMessage. Metoda setErrorMessage ima sledeći oblik.

```
private void setErrorMessage(FacesContext context, UIInput input) {  
    Iterator it = context.getMessages(input.getClientId(context));  
    if (it.hasNext()) {  
        FacesMessage facesMessage = (FacesMessage) it.next();  
        errorMessage = facesMessage.getSummary();  
    }  
}
```

Prvo preko metode getMessages klase FacesContext vraćamo iterator koji iterira po svim objektima FacesMessage koji se odnose na našu komponentu. Uzimamo prvu od tih poruka i njenu vrednost zapisujemo u errorMessage. Metoda getSummary klase FacesMessage vraća poruku o grešci.

Objasnimo još poslednja dva reda u metodi validateZip. Vidimo da smo i polju za grad i polju za državu setovali vrednost koju smo dohvatili u drugoj fazi životnog ciklusa na null. Kada se životni ciklus odvija normalno u svih šest faza tada imamo sledeći scenario. Nakon validacije, ako je vrednost ispravna JSF (faces servlet) svim UIInput komponentama setuje vrednost koja je prihvaćena u drugoj fazi životnog ciklusa na null. JSF proverava u šestoj fazi kada se rendera strana klijentu da li su svim UIInput komponentama setovane vrednosti na null. Ako nisu JSF ne rendera ponovo stranu već samo vraća staru stranu sa svim pogrešnim podacima. Za JSF je to znak da validacija nije uspeła. U našem slučaju mi preskačemo tri faze životnog ciklusa i JSF sam neće setovati vrednosti UIInput komponenti na null. Sa druge strane bilo da je zip ispravan ili ne mi želimo da osvežimo stranu i dobijemo poruku o grešci ili ispravne podatke. Zato smo dužni da sami setujemo UIInput komponentama vrednost null.

Dugme koje nas vraća na glavnu stranu takođe ima immediate atribut postavljen na vrednost true. Ovo dugme korisnik pritiska ako sa primera želi da se vrati na stranu sa glavnim menijem. Da nismo stavili immediate atribut pritiskom na dugme pokrenuo bi se JSF životni ciklus i ako u trećoj fazi validacija ne prođe bili bi vraćeni na istu stranu. To je efekat koji želimo da izbegnemo jer nam u tom slučaju validacija nije potrebna. Postavljanjem atributa immediate na true odmah nakon druge faze izvršava se akcija i odlazi na fazu renderanja strane klijentu.

4.2.7 Ajax4JSF Eho <<

[Vidi opis.](#)

Ovo je jednostavan primer upotrebe AJAX4JSF framework-a. Sve što unesemo u tekst polje prikazuje se odmah u jednom izlaznom polju. Pogledajmo kod strane eho.jsp

```

<%% taglib uri="http://java.sun.com/jsf/core" prefix="f"%>
<%% taglib uri="http://java.sun.com/jsf/html" prefix="h"%>
<%% taglib uri="http://richfaces.org/a4j" prefix="a4j" %>
<%% taglib uri="http://richfaces.org/rich" prefix="rich" %>
<%%page contentType="text/html" pageEncoding="UTF-8"%>
<f:view>
    <html>
        <head>
            <link type="text/css" rel="stylesheet" href="styles.css"/>
            <title><h:outputText value="#{msgs.title}"/></title>
        </head>
        <body>
            <div class="container">
                <h:form>
                    <div class="banner">
                        <h:outputText value="#{msgs.echo}"
                            styleClass="bannerText"/>
                    </div>
                    <hr/>
                    <rich:panel header="#{msgs.echo}">
                        <h:inputText size="50" value="#{backingEcho.txt}" >
                            <a4j:support event="onkeyup" reRender="eho"/>
                        </h:inputText>
                        <h:outputText value="#{backingEcho.txt}" id="eho" />
                    </rich:panel>

```

...

Tag rich:panel smo iskoristili čisto radi vizuelnog efekta. Ovu biblioteku dobili smo sa jar fajlovima koji donose podršku za AJAX4JSF. Tekst polju smo dodali AJAX funkcionalnost. Svaki put kada se pritisne i pusti dugme na tastaturi generiše se AJAX zahtev i ulazi u životni ciklus. Kada se završi životni ciklus osvežava se output polje koje ima id eho. Kako su i input i output polje povezani sa istim poljem bina scenario je sledeći.

Kada se pritisne dugme pozove se faces servlet. U četvrtoj fazi životnog ciklusa se string koji je upisan u input polje upiše u polje bina. Ta vrednost se koristi u šestoj fazi za renderanje output polja. Tako svaki put kada ukucamo znak u tekst polje on se odmah prikaže u output polju.

4.2.8 Početni meni << [Vidi opis.](#)

Na kraju još par reči o početnom meniju. Pogledajmo deo koda strane welcomeJSF.jsp.

```
<h:panelGrid columns="2">
    <h:panelGroup>
        <div class="menu"><h:outputText value="#{msgs.menu}"
                                         styleClass="heading"/>

        <h:commandLink action="primer1" >
            <a4j:support event="onmouseover" id="a1"
                        actionListener="#{backingWelcome.setOutput}"
                        reRender="ta"/>

            <h:outputText value="#{msgs.link1}"/>
        </h:commandLink>

        <h:commandLink action="primer2" >
            <a4j:support event="onmouseover" id="a2"
                        actionListener="#{backingWelcome.setOutput}"
                        reRender="ta"/>
            <h:outputText value="#{msgs.link2}"/>
        </h:commandLink>
    ...
</h:panelGroup>

<h:inputTextarea id="ta" value="#{backingWelcome.txt}" rows="3" cols="40"
                 styleClass="textAreaStyle" readOnly="true"/>

</h:panelGrid>
...
```

Svakom komandnom linku dali smo AJAX4JSF podršku. Kada se pređe mišem preko komandnog linka generisaće se AJAX zahtev i ulazi se u JSF životni ciklus. U petoj fazi će se pozvati metoda `backingWelcome.setOutput`. Jedino što se osvežava je tekst area koja ima id "ta". U nju će biti upisana vrednost `backingWelcome.txt`. Metoda `backingWelcome.setOutput` treba na osnovu zaključka koji je AJAX4JSF objekat generisao AJAX zahtev da setuje drugu poruku u polje `txt`. Deo metode `setOutput` prikazan je u daljem tekstu.

```
public void setOutput(ActionEvent ae) {
    if (ae.getComponent().getId().equals("a1")) {
        txt = Messages.getMessage("com.corejsf.messages", "exp1").getSummary();
    } ...
}
```




Odgovarajuća poruka se uzima iz `messages.properties` fajla. Svaka poruka predstavlja kratko objašnjenje korisniku šta koji primer radi i biće prikazana kada se mišem pređe preko linka.

5 Zaključak [<<](#)

U ovom poglavlju ćemo izložiti neke zaključke o korišćenim tehnologijama kao i kako bi sistem mogao da se unapredi.

Pre svega je potrebno ekstrahovati custom JSF komponente iz postojećeg projekta u poseban projekat. Drugim rečima, potrebno je napraviti reupotrebljiv artifakt, odnosno jar arhivu koja se posle može koristiti na više drugih projekata. Ovo je osnovni princip projektovanja softvera.

U ovom projektu su ilustrovana sva tri načina integracije JSF framework-a i AJAX tehnologije:

-  Čist AJAX (JavaScript)
-  Ajax uz upotrebu JavaScript biblioteka
-  Gotove AJAX JSF komponente

U nastavku je data diskusija na ovu temu kao i komentari za svako od ovih rešenja.

U realnom projektu je potrebno odlučiti se za jedan od ovih pristupa radi uniformnosti.

Čist AJAX: Ovo rešenje nije prihvatljivo za realan projekat iz prostog razloga što je potrebno previše vremena za kodiranje uz pomoć čistog java skripta. Takođe, uvek se ide logikom „zašto pisati nešto, ako je neko drugi to već napisao“. Pored toga, postoji veliki problem nekompatibilnosti browser-a, koji JS biblioteke uglavnom rešavaju dok u čistom JS-u to moramo sami da obrađujemo. Naročito ako se uzme u obzir da je JSF komponentno orijentisani framework, postaje jasno da se želi maksimalna enkapsulacija funkcionalnosti, a čist JS ne ide u prilog tome.

Upotreba biblioteka: Ovo je naprednije rešenje i ono pruža najbolji odnos brzine izrade i kontrole. Na ovaj način je moguće postići bilo kakvu funkcionalnost koliko god da je specifična, jer se JavaScript biblioteke koriste eksterno od JSF komponente. Sa druge strane, potrebno je da programer poznaje korišćenu JS biblioteku i da sa njom svakodnevno radi. Treba se odlučiti se za jednu od JS biblioteka kao sto su: JQuery, Prototype, Rico, DOJO itd.

Gotove komponente: Ovo rešenje je najviše u duhu JSF framework-a zato što se Ajax funkcionalnost potpuno enkapsulira u JSF komponentu tako da programer nema potrebe da poznaje interni rad. Na ovaj način je omogućen RAD (rapid application development), ali po cenu smanjene kontrole. Naime, čim je potrebna neka specifična funkcionalnost, programer je u problemu. U ovoj varijanti rešenja, potrebno je odlučiti se za jednu od JSF AJAX biblioteka kao sto su: AJAX4JSF, richfaces (Jboss), ADF (Oracle) itd.

Dakle, u naprednijoj verziji projekta trebalo bi da se odabere jedan od ova 3 načina rada, a potom se odlučiti za konkretnu implementaciju odabranog rešenja i držati se toga. Po mom mišljenju najbolje je hibridno rešenje 2 i 3, što znači da bih napravio svoju custom AJAX JSF biblioteku koja interno koristi neku JS biblioteku. Upravo u četvrtom primeru laboratorijskih vežbi je napravljena custom komponenta koja je enkapsulirala gotovu JavaScript komponentu iz Rico.js biblioteke.

Na kraju, možemo navesti nekoliko uočenih mana JSF frameworka za koje se nadamo da će biti ispravljene u najavljenoj 2.0 verziji JSF-a:

- Nepotpuna kontrola nad generisanim HTML kodom. Na primer, neke RichFaces komponente rendaju nekompatibilan XHTML kod.
- Nepotpuna kontrola samih custom komponenti: dok je potrebno bazično funkcionisanje komponente sve lepo radi. Promena funkcionalnosti komponente je komplikovana. Ukoliko je potrebno da se komponenta modifikuje najbolje rešenje je da se napiše nova.
- Razvoj custom komponenti je suviše komplikovan.

Literatura: <<

- [1] *Chris Schalk, Ed Burns.* The Complete Reference Java Server Faces. McGrawHill, 2007
- [2] *Jonas Jacobi, John R. Fallows.* Pro JSF and Ajax. Building Rich Internet Components. Apress, 2006
- [3] *David Geary, Cay Horstmann.* Core Java Server Faces Second Edition. Prentice Hall, 2007
- [4] *David Johnson, Alexei White, Andre Charland.* Enterprise Ajax . Prentice Hall, 2007
- [5] *Paul J. Deitel, Harvey M. Deitel.* Ajax, Rich Internet Applications and Web Development for Programmers. Deitel, 2008
- [6] Ajax4Jsf Developer Guide. Exadel 2006.