VISOKA ŠKOLA ELEKTROTEHNIKE I RAČUNARSTVA
STRUKOVNIH STUDIJA
AKADEMIJA TEHNIČKO-UMETNIČKIH STRUKOVNIH
STUDIJA

Zoran Ćirović

INTEGRACIJA SOFTVERSKIH TEHNOLOGIJA

Beograd, 2020.

Naslov:	Integracija softverskih tehnologija
	1. izdanje
Autor:	dr Zoran Ćirović
Recenzenti:	dr Slobodanka Đenić
	mr Jelena Mitić
Izdavači:	Visoka škola elektrotehnike i računarstva strukovnih studija u Beogradu
	Akademija tehničko-umetničkih strukovnih studija Beograd
Tehnička obrada:	Zoran Ćirović
Korice:	Ana Miletić
Štampa:	Razvojno-istraživački centar grafičkog inženjerstva TMF, Beograd
Tiraž:	30
ISBN:	978-86-7982-330-4

CIP - Каталогизација у публикацији- Народна библиотека Србије, Београд
004.415(075.8)
ЋИРОВИЋ, Зоран, 1970-
Integracija softverskih tehnologija / Zoran Ćirović. - 1. izd. - Beograd : Visoka škola elektrotehnike i računarstva strukovnih studija : Akademija tehničko-umetničkih strukovnih studija, 2020 (Beograd : Razvojno-istraživački centar grafičkog inženjerstva TMF). - 328 str. : ilustr. ; 24 cm
Tiraž 30.- Bibliografija: str. 320-321.- Registri.
ISBN 978-86-7982-330-4 (VŠERSS)
a) Софтвер- Интеграција

COBISS.SR-ID 25460745

Predgovor

Knjiga „Integracija softverskih tehnologija" najvećim delom pokriva gradivo istoimenog predmeta koji se sluša na trećoj godini osnovnih strukovnih studija odseka Visoke škole elektrotehnike i računarstva Akademije tehničko-umetničkih strukovnih studija u Beogradu.

Knjiga je podeljena u 13 poglavlja koja se bave aktuelnim tehnologijama. Novo gradivo uvodi se postepeno počev od definicija i osnovnih objašnjenja, preko instalacije, zatim elementarnih primera pa sve do praktično primenljivih kodova.

U prvom poglavlju prezentuju se osnovni formati za prenos podataka u internet okruženju: XML i JSON. Kreće se sa osnovama XML formata, pregledom strukture dokumenata, slede sintaksna pravila, parsiranje praznina, novih redova i uvođenje pojma imenskih prostora. Nakon toga radi se validacija pomoću DTD odnosno XSD šema. Za svaku od validacija radi se način definisanja elemenata, atributa, kao i povezivanje sa XML dokumentom. Na kraju, daje se pregled i osobine JSON formata i poređenje sa XML formatom.

Drugo poglavlje je posvećeno sistemima za verzioniranje. U okviru ovog poglavlja se definiše pojam i namena ovih sistema. Zatim se prezentuju osnovne funkcije i svaka od njih se posebno objašnjava. Uvodi se grafički prikaz i prikazuju se tipični slučajevi. Takođe se prikazuje postupak kreiranja nove verzije kao i opis sadržaja jednog repozitorijuma.

Treće poglavlje posvećeno je jednom konkretnom VCS sistemu - Git. Čitaocu se najpre daju osnovni podaci o nastanku Git-a, a zatim instrukcije

za instalaciju i konfigurisanje. Ovo poglavlje obuhvata najveći broj komandi potrebnih za rad. To su pre svega komande za praćenje statusa, istorije, kreiranje nove verzije. Naročita pažnja se posvećuje delu koji se tiče brisanja fajlova odnosno ispravci eventualnih grešaka pri kreiranju verzija.

Četvrto poglavlje bavi se radom sa više verzija istovremeno tj. grananjem. Objašnjen je prelaz sa jedne na drugu granu, preuzimanje koda iz druge grane. Posebna pažnja posvećena je spajanju dve grane. Pošto se prilikom spajanja mogu dogoditi i konflikti, opisuje se postupak razrešavanja konflikata. Na kraju objašnjeno je i brisanje grana.

Pojam i rad sa mrežnim repozitorijumima prezentuje se u petom poglavlju. Kreće se od kreiranja mrežnog repozitorijuma, preko dodavanja nove verzije kao i preuzimanja odgovarajućih verzija. Posebno je obrađeno spajanje mrežne sa tekućom verzijom i obrnuto, kloniranje kao i rad sa tipičnim sekvencama u razvoju.

Šesto poglavlje je posvećeno osnovama Node.js platforme. Prvo se prezentuju karakteristike platforme sa posebnom naglaskom na asinhroni princip rada. Zatim se daju instrukcije za instalaciju, provere tekuće verzije kao i osnovni način upotrebe. Slede objašnjenja korišćenja osnovnih komandi, prezentuje se kreiranje prve aplikacije, a zatim i komande za rad u konzolnom prozorom. Posebna pažnja posvećena je radu sa paketima i alatki npm. Prikazano uključivanje novih modula, izvoz promenljivih i funkcija kao i povezivanje lokalnih modula. Opisan je i prikazan u potpunosti, praktičan primer kreiranja sopstvenih lokalnih i mrežnih modula, njihovo postavljanje na repozitorijum odnosno registar modula. Na kraju, prezentovano je kako se koriste sopstveni lokalni odnosno mrežni moduli.

U okviru sedmog poglavlja opisane su tehnike programiranja koristeći Node.js platformu. Najpre se prikazuje način na koji se obrađuju događaji u Node-u, a zatim se uvodi pojam i prezentuju karakteristike blokirajućih odnosno neblokirajućih funkcija. Uvodi se pojam i rad sa tajmerima. U

nastavku, prezentuju se sopstveni događaji, kao i osluškivači. Na kraju, prezentuju se ulazno/izlazne operacije kao i rad sa fajlovima.

Realizacija HTTP servisa je osnovna tema osmog poglavlja. Prvo se uvode moduli `url` odnosno `querystring`, a zatim i `http`. Nakon osnovnih objašnjenja kreira se prvi server i vrši njegovo testiranje. Na tom primeru radi se parsiranje podataka iz zahteva klijenta. Posebno je obrađeno opsluživanje statičkih stranica, a posebno REST zahteva koji koriste GET/POST/PUT metode.

U devetom poglavlju bavimo se modulom `express`. Prikazujemo postupak instalacije, zatim i kreiranje veb aplikacije. Uvodi se i generator veb aplikacija zasnovan na ovom modulu. Posebna pažnja posvećuje se tehnici rutiranja, prenosu podataka preko parametara rute, ali i preko tela poruke. U nastavku se implementiraju GET/POST servisi koristeći ovaj modul, uvodi se pojam MVC arhitekture. Na kraju se radi jedna realizacija veb aplikacije zasnovana na MVC arhitekturi.

Deseto poglavlje bavi se tehnikama integracija Node.js aplikacija i aktuelnih baza podataka. Najpre je dat prikaz instalacije i upotrebe tri baze. Zatim je prikazana primena *mongo* baze u našim aplikacijama. Nakon instalacije baze odnosno Node modula, pomoću primera pokazano je: kreiranje baze, kreiranje i brisanje kolekcija, snimanje u kolekciju, pronalaženje podata, ažuriranje podataka i na kraju brisanje.

U jedanaestom poglavlju radi se jedan objektno-orijentisan pristup za rad sa podacima - primenom Mongoose paketa. Najpre se prikazuju osnovne karakteristike modula, njegove prednosti i nedostaci, a zatim posebne karakteristike najvažnijih objekata modula. Nakon instrukcija instalacije, u nastavku se prezentuje praktičan rad sa objektima kroz rad na kreiranju prvog modela. Najpre se radi definisanje modela tj. šeme uz obavezno uvođenje tipova i ograničenja, zatim se daju osnove o virtuelnim svojstvima kao i metode modela, kreiranju konekcije i na samom kraju pokazuje se kreiranje svih CRUD metode modela.

U dvanaestom poglavlju prezentuje se jedan programski model za kreiranje SOAP servisa. Nakon uvoda sledi objašnjenje pojma krajnja tačka kao i objašnjene elemenata krajnje tačke Definišu se vrste servisnih operacija i načini hostovanja servisa. Za opis i lakši rad na servisima uvodi se pojam i način upotrebe WSDL standarda. U nastavku poglavlja, radi se konfigurisanje servisnih projekata, daje se značenje i način korišćenja najbitnijih sekcija za podešavanja, zatim sledi pregled tipova servisnih operacija, detaljno se rade servisni ugovori i realizacija zasnovana na interfejsima, uvode se atributi metoda i klasa koji su od značaja za rad servisa. Za prenos podataka definišu se tipovi koji se serijalizuju, a zatim i ugovori o podacima za definisanje struktura u prenosu. Sva objašnjenja su praćena praktičnim primerima tako da se na kraju dobija kompletan servis sa željenim karakteristikama.

Poslednje, trinaesto poglavlje posvećeno je Web Api programskom modelu zasnovanom na tehnologiji Asp .Net Core i MVC šablonu. Prezentuju se osnovne karakteristike ove tehnologije otvorenog koda. Najpre se prezentuje način kreiranja projekta, a zatim uvođenje modela u projekat. Nakon modela, dodaju se kontroleri kao i tehnike rutiranja zahteva. Asinhroni princip u ovom programskom modelu se objašnjava kroz objašnjavanje primenjenih klasa i intefejsa. Posebno se opisuje GET odnosno POST metode servisa. Zatim, za potrebe testiranja servisa uvodimo JavaScript odnosno jQuery metode za pozive, poseban je osvrt na POST zahteve. Prikazuje se testiranje i pomoću *Postman* aplikacije.

Nakon svakog poglavlja data su pitanja namenjena proveri i boljem razumevanju izložene materije.

Cilj knjige je da posluži kao udžbenik za sticanje osnovnih znanja i veština u integraciji softverskih tehnologija. Zato knjiga obiluje primerima praćenim slikama, odgovarajućim kodom i dodatnim objašnjenjem.

Beograd, 2020. Autor

Sadržaj

Predgovor **3**
Sadržaj **7**
1. XML i JSON **18**
XML dokument 18
Definicija 18
Osobine 19
Struktura 20
Koren dokumenta 20
Mešovit sadržaj 21
Atributi 22
XML imena 22
Reference 23
Vrednost atributa 24
Element ili atribut? 24
Komentari 25
Odeljak CDATA 26
Instrukcije obrade 27
XML deklaracija 29
Atribut `version` 29
Atribut `enconding` 29
Atribut `standalone` 30

Praznine 30
Novi red 31
Prostor imena 32
Kvalifikovano ime 32
Korišćenje prostora imena 32
Nekvalifikovani elementi 33
Atributi i prostor imena 34
Dobro formiran dokument 35
DTD validacija 36
Povezivanje 37
Spoljašnje povezivanje 37
Javni identifikatori 38
Ugrađeni DTD 39
Kombinacija spoljašnji-ugrađeni DTD 39
DTD blokovi 40
Deklarisanje elemenata 40
EMPTY 41
ANY 42
Definisanje strukture 42
Deklarisanje atributa 45
Nabrajanje kao način definisanja tipa atributa 47
Tip atributa: ID 47
Tip atributa: IDREF 47
Tip atributa: ENTITY, ENTITIES 48
Dodatni opis vrednosti atributa 49
Specijalni karakteri – ENTITY 50
Neke mane DTD validacije 50
XSD validacija 51
Osnovna struktura XSD dokumenta 51
Pridruživanje šeme XML dokumentu 53
Ograničenja i realizacija 55
Deklaracije atributa 56
Ograničenja 57

Složeni tipovi ... 61
Proširivanje/nasleđivanje ... 62
Referisanje na deklaracije ... 63
Ograničenja broja pojavljivanja ... 64
Mešoviti sadržaj elemenata ... 65
Zadavanje redosleda elemenata ... 66
xs:sequence ... 66
xs:choice ... 66
xs:all ... 67
JSON ... 68
Tipovi podataka ... 69
JSON i XML ... 70
JavaScript ... 72
Šema ... 73
JSON i Ajax ... 74
Pitanja i zadaci ... 74
2. Sistemi za verzioniranje ... 77
Osnove ... 77
Čuvanje istorije ... 78
Rad u timu ... 79
Grananje ... 79
Rad sa spoljnjim učesnicima ... 79
Skaliranje ... 80
Grafički prikaz i tipični slučajevi ... 80
Organizacija verzioniranja ... 83
Postupak kreiranja nove verzije ... 84
Sadržaj repozitorijuma ... 85
Pitanja i zadaci ... 86
3. Osnove Git-a ... 87
Istorijat ... 88
Instalacija ... 89

Pomoć 90
Konfigurisanje 92
Sistemski/globalni/lokalni nivo 92
.gitignore 93
Kreiranje repozitorijuma 94
Status 95
Dodavanje fajla 96
Snimanje nove verzije 98
Tajno skladištenje 100
Brisanje 102
Čišćenje radnog prostora 103
Razlike 105
Referenca HEAD 108
Istorija promena 110
Blame 113
Tag 114
Naknadno tagovanje 114
Reset 115
Tipovi reseta 116
--hard 116
--mixed 116
--soft 117
Checkout 117
Revert 118
Ispravka 119
Pitanja i zadaci 120
4. Grananje 122
Uvod 122
Grananje 123

Prelazi 125
Preuzimanje fajlova 125
Spajanje grana 126
Konflikti 127
Rešavanje konflikata 128
Brisanje grana 132
Pitanja i zadaci 133
5. Mrežni repozitorijumi 134
Git hosting 134
Timski rad 135
Rad sa udaljenim repozitorijumima 138
Dodavanje udaljenog repozitorijuma 138
Snimanje 140
Preuzimanje izmena 141
Spajanje 143
Kloniranje 145
Tipične sekvence 146
Mrežni folder kao mrežni repozitorijum 148
Pitanja i zadaci 149
6. Node.js 150
Uvod 150
Karakteristike 151
Asinhroni princip 152
Upotreba 153
Instalacija 154
Lokacija 154
REPL 155

Komande ... 156
Prva aplikacija ... 158
Konzolni prozor ... 159
Instalacija paketa ... 161
Instalacija razvojnih alata ... 162
Pogled/provera paketa ... 163
Učitavanje modula ... 163
npm komande ... 164
Exports ... 165
Povezivanje lokalnih modula ... 167
Upotreba lokalnih fajlova i modula ... 167
Paketski moduli i aplikacije ... 168
Package.json ... 169
Kreiranje ... 170
Uključivanje paketa u projekat ... 171
Ispitivanje zavisnost modula ... 172
Dodavanje zavisnosti ... 173
Kreiranje deljenih modula ... 174
Repozitorijum za modul ... 174
Registar npmjs ... 176
Brisanje registracije ... 178
Upotreba deljenog modula ... 178
Paketi i moduli ... 179
Scouped paketi ... 180
Instalacija ... 180
Primena ... 181
Objava ... 181
Javni Paketi ... 182
Privatni paketi ... 182
Publikovanje ... 183
Pitanja i zadaci ... 183

7. Node.js programiranje 185
Model obrade događaja 185
Blokirajuće funkcije 187
Tajmeri 188
Tajmer sa istekom vremena 188
Neprekidni tajmeri 189
Trenutni tajmeri 190
Sopstveni događaji 191
Osluškivači 192
Dodavanje događaja objektu 193
I/O operacije sa fajlovima 194
Pitanja i zadaci 196
8. HTTP servisi 197
Osnove 197
Objekat *url* 198
Objekat *querystring* 199
Kreiranje servera 202
Testiranje 203
Parsiranje podataka iz zahteva 204
Node.js klijentski modul 205
Opsluživanje statičkih stranica 207
Testiranje 208
POST/PUT servisi 210
Testiranje 211
Elementi poruke 214
Zaglavlje 214
Tip poruke 215
Status poruke 215
Pitanja i zadaci 216

9. Express modul 218
Uvod 218
Prva aplikacija 219
Generator 220
Rutiranje 221
Primeri rutiranja 222
Primeri servisa 224
Get 224
Post 225
Request objekat 226
Response objekat 226
MVC arhitektura 229
Pitanja i zadaci 231
10. Pristup podacima 233
Uvod 233
Integracija baza podataka 234
MongoDB 234
MySql 235
Sql Server 235
MongoDB 237
Instalacija 237
Kreiranje baze 238
Kreiranje/brisanje kolekcija 239
Snimanje u kolekcije 240
Pronalaženje podataka 241
Ažuriranje dokumenta 243
Brisanje dokumenta 245
Združivanje kolekcija 246
Limit 247

Pitanja i zadaci 247
11. Uvod u *mongoose* 249
Karakteristike 249
Objekti 250
Instalacija 251
Kreiranje modela 251
Definisanje modela 251
Tipovi podataka 252
Virtuelna svojstva 255
Sopstvene metode modela 255
Konekcije 257
CRUD metode 258
Pretraga 258
Insert 259
Delete 260
Izmena 261
Pitanja i zadaci 262
12. SOAP servisi 264
Uvod 265
Osnovni pojmovi 265
Krajnja tačka 265
Vrste operacija 266
Kontejner servisa 266
WSDL 267
Osnove programskog modela 267
Dodavanje reference 268
Implementacija interfejsa 270
Hostovanje 271
Prva klijentska aplikacija 272
Upotreba konfiguracionih fajlova 274
Mex krajnja tačka 276
WCF klijent i konfiguracioni fajl 278

Ugovori 282
Vrste ugovora 282
Servisni ugovor 283
Poruke: zahtev-odgovor 283
Asinhroni zahtev-odgovor 284
Rad sa *task-based* operacijama 286
Jednosmerne i dupleks operacije 286
Jednosmerne operacije 286
Dupleks Operacije 287
Više ugovora i krajnjih tačaka 288
Ugovor o podacima 290
Serijalizacija u WSDL-u 291
Pitanja i zadaci 292
13. Veb Api .Net 295
Uvod 295
Kreiranje projekta 296
Dodavanje Modela 298
Dodavanje kontrolera 300
Get metode 303
POST metode 305
JavaScript / jQuery pozivi 309
Testiranje Get metoda 311
Primena u HTML stranicama 312
Lista 312
Jedan podatak 313
Testiranje Post metoda 313
Slanje kompleksnih tipova 314
Slanje prostih tipova 315
Testiranje aplikacije 316
HTTP Request / Response 317

Pitanja i zadaci 319
Literatura 320
Indeks pojmova 322
Indeks slika 325

1. XML i JSON

U ovom poglavlju prezentuju se osnovni formati za prenos podataka, a koji će biti korišećeni kasnije pri realizaciji programskih komponenata. Nakon uvodnih informacija o nazivu, verziji i istoriji nastanka, prezentuju se osnovne informacije o XML formatu, daje se pregled strukture dokumenata, pravila dobro formiranih dokumenta. Uvodi se pojam elementa i atributa, njihove osobine, a zatim i pojam i način upotrebe instrukcija obrade. Takođe, obrađuju se i pravila pisanja, npr: komentara, parsiranje novog reda odnosno praznina kao i pojam imenskih prostora.

Nakon toga radi se validacija dokumenata pomoću DTD odnosno XSD šema. Za svaku od validacija prezentuje se način definisanja elemenata, atributa, kao i povezivanje sa XML dokumentom.

Na kraju, daje se pregled i osobine JSON formata i poređenje sa XML formatom.

XML dokument

Definicija

XML dokument predstavlja podatke koji su tekstualno formatirani u skladu sa strogim XML pravilima. XML dokument mora u potpunosti da zadovoljava sva ova pravila tj. da bude dobro formiran (eng. *well formed*). Dokument tj. tekst sa podacima može da bude:

- smešten u datoteku na disku,
- kao poruka koja se šalje HTTP protokolom,
- kao niz znakova, tj. string u programskom jeziku,
- kao objekat u bazi podataka,
- na bilo koji drugi način koji omogućava korišćenje tekstualnih podataka.

Zbog svog formata, omogućava razmenu podataka između nekompatibilnih sistema. Nezavisan je od korišćenih hardverskih i softverskih platformi.

Osobine

XML je danas postao *de-facto* **standard za opis sadržaja** i strukture (tekstualnih i multimedijalnih) dokumenata i razmenu dokumenata na vebu. Dve važne osobine proizilaze iz njegovog naziva.

Markup označava:
- posebno značenje podataka,
- koristi se tag za predstavljanje osnovne jedinice podataka, slično kao kod HTML-a.

Extensible:

- prošiv jezik, tj. dozvoljava definisanje novih tagova,
- meta jezik koji omogućava definisanje drugih markup jezika.

Uz ovo, XML je:

- samoopisujući,
- platformski nezavisan,
- u tekstualnom formatu.

Sa ovim osobinama XML je **format prilagođen za razmenu podataka između heterogenih aplikacija na vebu.** Omogućava **razdvajanje struktuiranog sadržaja dokumenta od njegovog prikaza.**

XML je **projektovan za distribuirano okruženje.** Kaže se da je to format **dovoljno formalan za mašinsko procesiranje i dovoljno razumljiv za korisnike.**

Struktura

XML dokument se sastoji iz teksta organizovanog uz pomoć tagova u elemente. **Elementi** su osnovni blokovi XML-a. Na primer:

```
<pozdrav> Zdravo za XML! </pozdrav>
```

```
<pozdrav></pozdrav>
```

Postoje dve vrste XML elemenata:

- ***Složeni*** ili ***kontejner*** element. Čini par tagova, početni i krajnji tag, sa sadržajem između, kao u prvom primeru elementa `pozdrav`.
  Prost ili ***prazan*** element. Nema sadržaj. Obično se za krajnji tag koristi skraćenica **/>**, kao u drugom primeru. Prazan element može da se zapiše i na sledeće načine, na primer:
  ```
  <poruka/>
  <pozdrav tekst = 'Hello XML' />
  ```

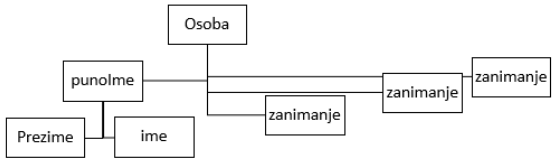
Koren dokumenta

XML dokument obavezno sadrži tačno jedan element koji nema roditeljske elemente. Uvek je to prvi element u dokumentu. Sadrži sve druge elemente.

To su sve karakteristike elementa koji nazivamo **korenskim** (eng. *root*). Grafički, struktura elemenata može da se prikaže u obliku stabla, gde je korenski element prvi od koga kreće grananje svih drugih elemenata u dokumentu.

U primeru koji sledi dat je grafički prikaz i ceo dokument čije je korenski element `osoba`:

Grafički prikaz:



XML dokument:

```
<osoba>
  <punoIme>
    <ime>Perica</ime>
    <prezime>P</prezime>
  </punoIme>
  <zanimanje>analitičar</zanimanje>
  <zanimanje>dizajner</zanimanje>
  <zanimanje>...</zanimanje>
</osoba>
```

Slika 1.1. XML stablo dokumenta

Mešovit sadržaj

XML se može koristiti i za narativne dokumente slobodnijeg oblika, kao što su: poslovni izveštaji, članci, eseji, priče... U ovim slučajevima sadržaj elementa može biti tekst, ali i drugi elementi. Za takve sadržaje kažemo da su **mešoviti**. Na primer:

```
<biografija>    <punoIme><ime>Jovan</ime>
<prezime>Petrovic</prezime> </punoIme> jedan je od istaknutih
<istaknuto>naučnih saradnika</istaknuto>našeg instituta.
```

```
Zaposlen je <datum><dan>17</dan><mesec>juna</mesec>
<godina>1994</godina></datum> …. </biografija>
```

Atributi

Elementima se mogu pridružiti atributi koji pružaju dodatne informacije o podacima. Vrednosti atributa se uvek pišu unutar znakova navoda. Na primer:

```
<poruka datum="12.5.08." sala="201">
   <tekst>Sastanak Katedre sutra u 10</tekst>
</poruka>
```

U gornjem primeru atributi su `datum` i `sala`.

XML imena

Imena koja se koriste u XML dokumentima (eng. *XML names*) **mogu** da sadrže:

- Sve alfa-numeričke znakove;
- Neengleska slova, brojeve i ideograme (ö, ç,..);
- Donja crta, crtica, tačka.

XML imena **ne mogu** sadržati:

- " - Navodnike;
- ' - Polunavodnike;
- $ - Znak za dolar;
- ^ - Kapicu;
- % - Znak za procenat;
- ; - Tačka-zarez.

Dvotačka je dozvoljena, ali je rezervisana za prostore imena.

Sva imena koja počinu sa **xml** su rezervisana za standardizaciju.

Posebno se definišu pravila za **početak imena:**

- XML imena **mogu** da počnu samo slovom i donjom crtom.
- **Ne mogu** početi: cifrom, crticom, tačkom.

Na kraju, recimo i to da **dužina** imena nije ograničena.

Reference

Znakovni **podaci unutar elementa** ne smeju da sadrže znak manje - **<**. Ovaj karakter se uvek tumači kao početak taga.

Ako ipak, u nekom XML dokumentu, treba da se koristi baš ovaj karakter kao deo sadržaja, onda se mora koristiti referenca **<**. Reference su posebne oznake koje programi za tumačenje XML dokumenata tzv. XML parseri umeju da protumače, odnosno da formiraju parsirani sadržaj na izlazu, tako da izvrše zamenu **<** reference sa znakom **<** . Na primer:

```
<SCRIPT LANGUAGE="JavaScript">
   if (location.host.toLowerCase().indexOf("kafa") &lt; 0)
   {
       location.href="http://www.cafeconleche.org/";
   }
</SCRIPT>
```

Takođe, XML elementi ne smeju sadržati znak &, jer se ovaj znak uvek interpretira kao početak reference. Zato se za prikaz znaka & koristi referenca **&** , na primer:

```
<publisher>O'Reilly &amp; Associates</publisher>
```

U sledećoj tabeli prikazano je nekoliko često korišćenih referenci:

Tabela 1.1. Česte reference

<	<
>	>
&	&
'	‘

"	“

Vrednost atributa

Vrednosti atributa moraju uvek biti unutar znaka navoda, bez obzira koji tip podatka koristi atribut. Moguće je koristiti **jednostruke ili dvostruke** znake navoda:

`<ime="Zmaj">` ili `<ime = 'Zmaj'>`

Dupli znaci navoda su češći. Nekada je neophodno koristiti jednostruke i dvostruke zajedno, kao na primer kada vrednost atributa sadrži određene navode, na primer:

```
<ime='Jovan Jovanović "Zmaj"'>
```

Element ili atribut?

Podaci se mogu čuvati u elementima ili u atributima. Pitanje izbora se definiše na osnovu: potreba, osobina podataka koji se definiše, ali i na osnovu iskustva odnosno umeća dobrog projektovanja. Pogledajmo sledeće primere:

Primer 1.

```
<ime>Zmaj</ime>
<nešto ime="Zmaj">
```

Primer 2a.

```
<partner tip="nabavljač">
   <ime>Pera</ime>
   <prezime>Perić</prezime>
</partner>
```

Primer 2b.

```
<partner>nabavljač</partner>
<ime>Pera</ime>
<prezime>Perić</prezime>
```

U primeru 2a. `tip` je atribut. U primeru 2b. `tip` je element. Oba primera sadrže iste informacije. Ne postoje pravila koja govore kada koristiti podatak kao atribut, a kada kao element. Jedna načelna, dakle neobavezujuća, preporuka jeste da se elementi koriste kada je u pitanju podatak koji je sam po sebi jedna celovita informacija, a ne neki njen pomoćni deo.

Postoje i konkretni razlozi za izbor jedne ili druge opcije. Potencijalni problemi tj. ograničenja prilikom korišćenja atributa su:

1. Atributi ne mogu sadržati višestruke vrednosti, elementi mogu;
2. Atributi nisu lako proširivi;
3. Atributi ne opisuju strukturu;
4. Atributima se teže manipuliše u programskom kodu;
5. Vrednosti atributa se teško testiraju koristeći validaciju zasnovanu na DTD dokumentima – definicija tipa dokumenta.

Komentari

Komentari se koriste da pruže dodatne informacije u dokumentu, a da pri tome ne utiču ni na koji način na informacije koje XML parser obrađuje. XML parseri mogu da izdvoje komentare, ali su komentari bez uticaja na podake koje XML čuva. Sadržaj komentara nije bez ograničenja. Sintaksa za komentar je:

```
<!- komentarisani text -->
```

Aplikacije za XML parsiranje mogu, a ne moraju da prosleđuju informacije o komentarima. Ovo treba imati u vidu kada se pišu komentari u XML dokumentima na strani servera/servisa. Eventualno zapisivanje bezbedonosno nesigurnih podataka može da dovede do kompromitovanja istih.

Primeri:

```
<!– Ovo je komentar u vazi otvaranja ( <![CDATA[ ) odnosno
zatvaranja ( ]]> ) CDATA sekcije -->
<!– Drugi primer upotrebe taga <test> u primerima -->
<!– Komentar koji sadrži specijalne karaktere koji uključuju &,
<, >, ' odnosno". -->
<!– Ako se entiteti koriste unutar komentar ( &lt; na primer ).
-->
```

Sadržaj u komentaru ne može imati – ili druge komentare. Primeri ne ispravnih komentara:

```
<!-- Komentar ne može da sadrži -- sekvencu -->
<!-- Komentar se ne sme završiti crticom --->
<!-- Komentari se ne mogu <!-- ugnježdavati --> -->
```

Odeljak CDATA

XML dokumenti imaju strogo definisana pravila koja isključuju pisanje nedozvoljenih karaktera. Ukoliko dokument sadrži sirovi tekst, na primer jedan novinski članak ili ceo roman, u tom tekstu mogu postojati i nedozvoljeni karakteri, pa se mora izvršiti izmena istih na svim mestima gde se pojavljuju. Takav dokument je težak za kreiranje, težak za izmene i postaje teško razumljiv.

U takvim slučajevima, kada je potrebno zapisati sirovi tekst, preporučuje se upotreba odeljka **CDATA** koji obezbeđuje da se njen sadržaj ne parsira tj. može da se čuva u sirovom obliku. Na primer.

```
<p>ovde ide neki tekst </p>
<pre>
<![CDATA[
    <svg xmlns="http://www.w3.org/2000/svg"
         width="22cm" height="20cm">
    <ellipse rx="210" ry="130" />
    <rect x="4cm" y="1cm" width="3cm" height="6cm" />
]]>
</pre>
```

Instrukcije obrade

Instrukcije obrade koriste se kako bi pružile dodatne informacije aplikacijama za obradu XML dokumenata.

Mogu sadržati informacije kako dokument obraditi, prikazati i slično. Sastoje se iz dva dela:

- Ime instrukcije **target** ;
- Podatak za instrukciju **data**.

Sintaksa je: `<?target data?>`

`Target` mora da zadovolji ista pravila za imene kao elementi i atributi, osim završnog karaktera za sekvencu (**?>**).

Mada zapis instrukcija obrade odstupa od pravila XML-a, XML parseri ih prepoznaju i odvajaju ih od ostatka dokumenta. Po preporukama W3C konzorcijuma, ova sekvenca **ne sme počinjati sa xml sekvencom**.

Imenski prostori se ne primenjuju na instrukcije. Zato je teško garantovati jedinstvenost instrukcija i to predstavlja potencijalni problem. Na primer:

```
<?robots index="yes" follow="no"?>
<?display table-view?>
<?sort alpha-ascending?>
<?textinfo whitespace is allowed ?>
<?elementnames <fred>, <bert>, <harry> ?>
```

Instrukcije obrade mogu imati potpuno drugačiju sintaksu i semantiku od xml-a. Na primer, instrukcije obrade mogu imati efektivno neograničenu količinu teksta. PHP smešta velike programe u instrukcije za obradu. Na primer:

```
<?php
   mysql_connect("database.unc.edu", "clerk", "password");
   $result = mysql("HR", "SELECT LastName, FirstName FROM
Employees ORDER BY LastName, FirstName");
   $i = 0;
   while ($i < mysql_numrows ($result))
   {
        $fields = mysql_fetch_row($result);
        echo "<person>$fields[1] $fields[0]
        </person>\r\n"; $i++;
   }
   mysql_close( );
?>
```

Instrukcije obrade spadaju u označavanja dokumenta, tj. nisu elementi koji čuvaju sadržaj dokumenta. Mogu se pisati na bilo kom mestu u dokumentu osim unutar tagova. Dakle, pišu se kao i komentari što se pišu. Mogu se pisati i pre korenskog elementa i posle.

Primer primene stilova:

```
<?xml-stylesheet href="osoba.css" type="text/css"?>
<osoba>
```

```
        Jovan Jovanović
</osoba>
```

XML deklaracija

Svaki XML dokument mora da sadrži **XML deklaraciju**, tj. ***instrukciju obrade*** kojom se dokument identifikuje kao XML dokument. Ovo je prva linija u dokumentu. Pogledajmo nekoliko primera.

Minimalni oblik XML deklaracije:

```
<?xml version = "1.0"?>
```

Prošireni olik XML deklaracije uz definisanje kodovanja:

```
<?xml version = "1.0" encoding = "UTF-8"?>
<?xml version = "1.0" encoding = "UTF-16"?>
```

Opšti oblik XML deklaracije:

```
<?xml  version='1.0' encoding='char.encoding'
standalone='yes|no'?>
```

Atribut `version`

Atribut `version` treba da ima vrednost `1.0`. U veoma retkim slučajevima može da dobije vrednost `1.1`. Pošto zadavanje verzije `1.1` ograničava upotrebu na manji broj analizatora, ne bi trebalo postavljati ovu verziju bez istinskog razloga.

Ako ne govorite burmanski, mongolski, kambodžanski, amharski ili diverhi, ako ne koristite netekstualne C0 kontrolne znakove (vertikalni tabulator, nova stranica, zvonce) nemate razloga da koristite verziju 1.1.

Atribut `enconding`

XML je dokument čistog tekstualnog formata. Međutim tekst se zapisuje na različite načine. Ovaj atribut definiše kako su kodovani karakteri teksta

u konkretnom dokumentu. Neke vrste kodovanja su: ASCII, Latin-1, Unicode.

Svi XML dokumenti su podrazumevano kodovani sa **UTF-8** kodovima promenljive dužine u skupu Unicode.

Pri analizi XML fajla, analizator čita iz metapodataka vrstu kodovanja tog fajla i primenjuje ga. Međutim ne pružaju svi dokumenti takve podatke, pa je neophodno iste navesti u samom dokumentu koristeći gore naveden atribut.

Ako podaci o kodovima ne postoje u samoj datoteci, a ne navedu se u dokumentu, analizator pretpostavlja **Unicode**. Ukoliko postoje oba podatka, a u suprotnosti su, analizator veruje sistemu tj. koristi kodovanje koje je zapisano u sistemu.

Atribut `standalone`

Ako ovaj atribut ima vrednost *"no"* onda aplikacija može učitati spoljni DTD dokument preko koga se utvrđuju vrednosti i značenje delova dokumenta. U narednim poglavljima biće više informacija o DTD standardu.

Dokument koji nema DTD može imati ovaj atribut postavljen na vrednost *"yes"*. Ako je izostavljen, pretpostavlja se da ima vrednost *"no"*

Praznine

Podrazumevano, **u XML dokumentu prazan prostor je od značaja**. Korišćenjem XML-a prazan prostor je deo sadržaja i u parsiranom dokumentu. Na primer:

Pre parsera:

```
<body>Puno          pozdrava iz Beograda</body>
```

A nakon parsera:

```
Puno          pozdrava iz Beograda
```

Osim kao deo sadržaja elementa, praznine mogu da budu pisane u okviru tagova tako da predstavljaju grešku u formatu ili da doprinose njegovoj čitljivosti.

Primer ispravno korišćenih praznina:

```
<pre:vozilo xmlns:pre='urn:example-org:Transport'
type='auto' >
   <sedista> 4 </ sedista >
   <boja> Bela </boja>
  <motor>
    <benzin />
    <zapremina jedinica='cc' >1998</zapremina>
  </motor >
</pre:vozilo >
```

Primer neispravno korišćenih praznina:

```
<pre :vozilo xmlns:pre='urn:example-org:Transport'
Type>='auto'>
< sedista>4</ sedista >
</pre: vozilo>
```

Novi red

U XML-u, nov red u tekstu je uvek sačuvan kao LF (eng. *Line Feed*). U Windows aplikacijama nov red je par CR (eng. *Carriage Return*) i LF karaktera. Kod UNIX sistema karakter nov red je LF, mada neke aplikacije koriste samo CR. Ova razlika među operativnim sistemima često za

posledicu ima da se podaci vraćaju u obliku toka (eng. *stream*), a ne u željenom formatu.

U XML-u, CR / LF karakteri se pretvaraju u LF karakter.

Prostor imena

Uloga prostora imena je:

1. Da omogući razlikovanje istoimenih elemenata;
2. Da grupiše srodne elemente.

Dizajneri XML dokumenata mogu da odabere sopstvene nazive elemenata. Zato postoji velika verovatnoća da se u različitim namenama koriste istoimeni elementi.

Na primer, neki XHTML dokument može da sadrži i SVG slike i MathMl jednačine. Dokumenti koji sadrži elemente iz različitih aplikacija, moraju obezbediti jedinstveno ime za sve elemente. To se postiže koristeći imenski prostor.

Kvalifikovano ime

XML prostor imena omogućava razdvajanje XML elemenata koji imaju isto lokalno (kratko) ime, koristeći **jedinstveni identifikator resursa – URI** (eng. ***U****niform* ***R****esource* ***I****dentifier*).

URI predstavlja jedinstveni string za definisanje punih imena na osnovu lokalnih imena i URI-a. Dakle, **prostor imena i lokalno ime zajedno čine globalno jedinstveno ime**.

Korišćenje prostora imena

Deklaracija prostora imena se vrši unutar početnog taga.

Mapiranje prostora imena se vrši u jedan drugi string, obično kraći, poznat kao *prefiks* prostora imena. Na primer:

`xmlns:prefix='URI'`

Ako se izostavi prefiks, na primer, **xmlns='URI'** , onda se mapiranje vrši u **podrazumevani prostor imena**.

Opseg važenja (eng. *scope*) prostora imena je skup elemenata na koje se odnosi taj prostor imena. **Prostor imena važi za elemenat u kome je deklarisan i u svim elementima koji su hijerarhijski ispod tog elementa.**

Nekvalifikovani elementi

Ako element nema prefiks, onda pripada podrazumevanom prostoru imena, ako postoji, ako ne onda je **nekvalifikovan**.

Dakle, elementi koji nisu u poznatom prostoru imena nazivaju se nekvalifikovanim *elementima.* Prostor imena takvih elemenata je **prazan string - ""**.

Ako postoji podrazumevani prostor imena za neki elemenat, a element ne pripada tom prostoru imena tj. elemenat treba da bude nekvalifikovan, tada se taj, podrazumevani, prostor imena može maskirati koristeći deklaraciju oblika **xmlns=""** u tom elementu.

Primer 1.

```
<pre:Osoba xmlns:pre='tmp:primeri-ist' >
   <ime>Marko</ime>
   <godine>33</godine>
</pre:Osoba>
```

Element **Osoba** ima prefiks **pre** koji je mapiran u prostor imena **tmp:primeri-ist**. Ovaj elemenat sadrži podelemente sa lokalnim imenima **ime**, **godine**. Oba podelementa su nekvalifikovana; zato što nisu u nekom prostoru imena. Treba razlikovati oblast važenja prostora imena od njegove primene na neki element.

Primer 2.

```
<Osoba xmlns='tmp:primeri-ist' >
   <ime xmlns=''>Marko</ime>
   <godine xmlns=''>33</godine>
</Osoba>
```

Element `Osoba` nema prefiks. Ujedno isti elemenat definiše podrazumevani prostor imena `tmp:primeri-ist`. Ovaj elemenat sadrži podelemente sa imenima `ime` odnosno `godine`. Međutim, oba podelementa preklapaju podrazumevani prostor imena sa prostorom ''. Dakle, oba podelementa su nekvalifikovana, zato što nisu u nijednom prostoru imena tj. pripadaju prostoru imena koji je prazan string. Ovaj primer je ekvivalentan primeru 1.

Primer 3.

```
<pre:Osoba xmlns:pre='tmp:primeri-ist ' >
   <pre:ime>Marko</pre:ime>
   <pre:godine''>33</pre:godine>
</pre:Person>
```

Element `Osoba` ima prefiks, kao i podelementi. Elemenat definiše `pre` prostor imena `tmp:primeri-ist`. Dakle, svi elementi su kvalifikovani.

Primer 4.

```
<Osoba xmlns='tmp:primeri-ist ' >
   <ime>Martin</ime>
   <godine>33</godine>
</Osoba>
```

Element `Osoba` definiše podrazumevani prostor imena `tmp:primeri-ist`. Taj prostor imena koriste podelementi. Ovaj primer je ekvivalent primeru 3.

Atributi i prostor imena

Atributi bez prefiksa ne pripadaju nijednom imenskom prostoru, čak i kada su u oblasti važenja nekog podrazumevanog imenskog prostora.

Primer. Nekvalifikovani atributi

```
<Osoba xmlns='tmp:primeri-ist'>
   <ime>Marko</ime>
   <godine osnova='10' jedinica='god' >33</godine>
</Osoba>
```

Mada postoji podrazumevani imenski prostor, atributi su nekvalifikovani. Jedan primer kvalifikovanih atributa bio bi:

```
<Osoba xmlns='tmp:primeri-ist'
       xmlns:o='tmp:primeri-ist:osnove'
       xmlns:j='tmp:primeri-ist:jedinice'>
   <ime>Marko</ime>
   <godine o:osnova='10' j:jedinica='god' >33</godine>
</Osoba>
```

Dobro formiran dokument

Da bi XML dokument bio dobro formiran, dokument treba da bude kreiran po svim pravilima tj. da ima sledeće karakteristike:

- Postoji XML deklaracija.
- Dokument sadrži jedan i samo jedan korenski element u kome su ugnježđeni svi ostali elementi i njihovi sadržaji.
- Svi elementi i atributi u dokumentu moraju da budu sintaksno ispravni.
- Elementi moraju imati završni tag (<…> … </…>). Jedini izuzetak predstavlja *empty* tag koji nema ni sadržaj ni telo, a označava se <…/>.
- Elementi moraju biti ugnježđeni.
- Imena tagova u XML-u zavise od primene velikih i malih slova (eng. *case-sensitive*). Pri dodeljivanju imena moraju se poštovati pravila.
- Sve vrednosti atributa moraju biti u okviru navodnika.

DTD validacija

XML dokument mora biti dobro formiran. Iako, velika sloboda u kreiranju dokumenata pruža velike mogućnosti i raznovrsnost primena, da bi neka primena bila efikasna neophodna su dodatna pravila, a ona se uvode validacijom.

Svrha uvođenja validacije za XML dokument je:

1. Da se uvedu ograničenja tj. opis strukture podataka, a time se dobijaju nove mogućnosti. Opis strukture je javan i potpuno razumljiv tj. standardizovan.
2. Automatizacija provere ispravnosti dokumenta.
3. Višestruka upotreba definisanih delova dokumenta.

DTD je dodatni dokument koji može biti pridružen nekom XML dokumentu i pruža gore navedene osobine, dakle predstavlja dokument za validaciju XML dokumenta. Inače, DTD je skraćenica od engleskog naziva ***D****ocument* ***T****ype* ***D****efinition.*

Tako na primer, DTD definiše: elemente i pripadajuće atribute. Svaki element i atribut karakteriše se određenim tipom kao i strukturom podataka koje on sadrži. Takođe, atributi su definisani tipom atributa i opisom podataka koje atribut sadrži. Postoje i liste atributa koje omogućavaju grupisanje srodnih atributa elementa.

Sa druge strane, DTD ima svoje domete u pogledu definisanja ograničenja. Tako, DTD ne može da deklariše:

1. Koji je element korenski.
2. Broj primeraka elemenata pojedinih vrsta u dokumentu.
3. Kako izgledaju znakovni podaci unutar elemenata.
4. Semantiku, tj. značenje elemenata; na primer, da li određeni element sadrži datum ili ime neke osobe.

Primer jednog DTD dokumenta je:

```
<!ELEMENT osoba (ime, zanimanje*)>
<!ELEMENT punoIme (ime, prezime)>
<!ELEMENT ime (#PCDATA)>
<!ELEMENT prezime  (#PCDATA)>
<!ELEMENT zanimanje (#PCDATA)>
```

A jedan mogući XML dokument koji bi odgovarao ovom DTD dokumentu mogao bi biti:

```
<osoba>
  <punoIme>
    <ime>Jovan</ime>
    <prezime>Jovanović</prezime>
  </punoIme>
  <zanimanje>pesnik</ zanimanje >
  <zanimanje>lekar</zanimanje>
</osoba>
```

Povezivanje

Pridruživanje DTD dokumenta nekom XML dokumentu može da se uradi na više načina:

1. Kao poseban fajl u odnosu na XML. Na ovaj način se postiže ponovna upotrebljivost napisanih pravila, tj. DTD može biti naknadno više puta korišćen.
2. Kao deo XML dokumenta.
3. Kombinacijom prethodna dva slučaja, tj. deo DTD-a je u posebnom dokumentu, a deo u XML dokumentu.

Spoljašnje povezivanje

XML i DTD su zasebni dokumenti. Povezivanje se ostvaruje tako što se prosleđuje URI na kome je DTD fajl. Sintaksa za povezivanje je:

```
<!DOCTYPE root-element [element-declarations]>
```

Primeri.

hello.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE message SYSTEM
"http://www.viser.edu.rs/dtd/poruka.dtd">
<poruka>
     <tema>Opis DTD formata!</tema>
     <potpis>Perica P!</potpis>
</poruka>
```

http://www.viser.edu.rs/dtd/poruka.dtd

```
<!ELEMENT poruka (tema, potpis)>
<!ELEMENT tema (#PCDATA)>
<!ELEMENT potpis (#PCDATA)>
```

U gornjim primerima koristi se putanja do DTD fajla. Ako se dokument nalazi na istoj osnovnoj lokaciji kao i DTD, može se koristiti **relativna putanja** (adresa) umesto apsolutne, na primer:

```
<!DOCTYPE message SYSTEM "/dtd/poruka.dtd">
```

Ako se dokument nalazi u istom direktorijumu može se zadati samo ime datoteke:

```
<!DOCTYPE message SYSTEM "poruka.dtd">
```

Javni identifikatori

Ovo je posebna vrsta spoljašnjeg povezivanja XML dokumenta i DTD dokumenta. Zbog specifičnosti i značaja posebno je opisujemo. Sintaksa za povezivanje je:

```
<!DOCTYPE root_el. PUBLIC "publicID" "OpcioniURL">
```

Standardni DTD dokumenti mogu biti smešteni na više URL adresa. Ime javnog identifikatora, *public ID,* jednoznačno određuje XML aplikaciju za koju se koristi. U isto vreme se navodi i rezervna URL adresa, za slučaj da analizator validnosti ne prepozna javni identifikator.

Primeri nekih javnih DTD identifikatori:

Rich Site Summary (Netscape):

```
<!DOCTYPE rss PUBLIC “-//Netscape Communications//DTD RSS
0,91//EN” “http://my.netscape.com/publish/formats/rss-0.91.dtd">
```

XHTML:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtl1/DTD/xhtml1-transitional.dtd">
```

Ugrađeni DTD

Ovo je povezivanje XML-a i DTD-a, ali kada su u istom XML fajlu. Na primer

hello.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE poruka [
       <!ELEMENT poruka (tema, potpis)>
       <!ELEMENT tema (#PCDATA)>
       <!ELEMENT potpis (#PCDATA)>
]>
<poruka>
     <tema>Opis DTD formata!</tema>
     <potpis>Perica P!</potpis>
</poruka>
```

Kombinacija spoljašnji-ugrađeni DTD

Kombinacija je slučaj kada je DTD jednim delom ugrađen u XML dokument, a drugim delom je u posebnom fajlu. Na primer:

hello.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE poruka SYSTEM "poruka.dtd" [
   <!ELEMENT potpis (#PCDATA)>
   <!– preklapa potpis u message.dtd -->
]>
<poruka>
     <tema>Opis DTD formata!</tema>
     <potpis>Perica P!</potpis>
</poruka>
```

poruka.dtd

```
<!ELEMENT poruka(tema,farewell)>
<!ELEMENT tema (#PCDATA)>
<!ELEMENT potpis EMPTY>
```

DTD blokovi

DTD definiše pet različitih celina tj. blokova. To su:

- Elementi.
- Atributi.
- Entiteti. Oznake koje imaju specijalno značenje, a mogu zameniti i zabranjene karaktere (<,>,&..). Entiteti se ekspanduju kada se neki dokument parsira.
- PCDATA (eng. *Parsed Character Data).* Tekst koji će biti procesiran od parsera. Tekst će biti analiziran kao markap celine.
- CDATA (eng. *Character Data)*. Označava tekst koji neće biti parsiran. Tagovi u okviru ovog teksta neće biti tretirani kao markap.

Deklarisanje elemenata

Elementi u DTD dokumentu deklarišu se na sledeći način:

`<!ELEMENT ime (sadrzaj)>` gde je:

`ime` – naziv elementa,

`sadrzaj` – podaci koje element može da sadrži. Postoji više opisa sadržaja po DTD specifikaciji. O ovome nešto više kasnije.

Na primer, ako je DTD dokument:

```
<!ELEMENT pozdrav (#PCDATA)>
```

Onda su validni XML elementi:

```
<pozdrav>Zdravo svima!</pozdrav>
<pozdrav>
        <![CDATA[G'day!]]>
</pozdrav>
```

Sada ćemo da pogledamo kako se opisuje sadržaj elementa koji se definiše. Najpre definišimo vrste tj. tipovi sadržaja. To su:

- Prazan element;
- Bilo koji element;
- Tačno definisani elementi;
- Tekst (#pcdata).
- Mešavina.

Za svaki tip postoji odgovarajuća sintaksa tj. način kako se zapisuje. Slede opisi za svaki tip redom.

EMPTY

Element koji ima ovako definisan sadržaj ne sme da sadrži nikakve podatke. Može da ima samo atribute. Na primer:

DTD deklaracija: `<!ELEMENT NoviRed EMPTY>`

Ispravan XML kod: `< NoviRed ></ NoviRed >` ili `< NoviRed />`

ANY

Svi elementi koji su korektno definisani mogu da budu sadržani u elementu čiji je sadržaj podataka **ANY**. Elementi koji se koriste moraju biti deklarisani u DTD-u. Na primer:

DTD fajl:

```
<!ELEMENT grupa ANY>
<!ELEMENT student (#PCDATA)>
<!ELEMENT nastavnik (#PCDATA)>
```

XML kod:

```
<grupa/> ili <grupa></grupa>
<grupa>pera, luka, mika</grupa>
<grupa>
   <student>joca</student>
   <nastavnik>milan</nastavnik>
</grupa>
```

Definisanje strukture

DTD dokument opisuje strukturu elemenata koji se koriste u XML dokumentu, pri tome, koristeći određena pravila i sintakse. Sintaksa za definisanje elemenata je opšteg oblika:

```
<!ELEMENT ime (struktura)>
```

Element je definisan nazivom **ime** i strukturom elemenata koji su njegovi podelementi tj. članovi. U narednoj tabeli prikazane su strukture i prateći primeri:

Tabela 1.2. Opis strukture elementa:

Sekvenca	`<!ELEMENT ime (a,b)>`
Izbor	`<!ELEMENT ime (a|b)>`
Jedan	`<!ELEMENT ime (a)>`
Jedan ili više	`<!ELEMENT ime (a) +>`

Nula ili više	`<!ELEMENT ime (a) *>`
Nula ili jedan	`<!ELEMENT ime (a) ?>`

a i b mogu imati složeniju strukturu, kao na primer a = (c,d)

- Sekvenca `(a,b,c)`;
  - Navedeni elementi moraju se pojaviti **u zadatom redosledu**.
- Broj potomaka;
  - `?` Dozvoljeno **nula ili jedan** element.
  - `*` Dozvoljeno **nula ili više** elemenata.
  - `+` Dozvoljeno **jedan ili više** elemenata .
- Izbor `|`
  - Spisak elemenata razdvojenih vertikalnom crtom. Mora sadržati jedan od ponuđenih elemenata. Ne može sadržati više od jednog ponuđenog.

Primeri

Na primer, ako treba saopštiti da element *krug* sadrži element *centar* i element *prečnik* ili *poluprečnik*, ali ne oba.

```
<!ELEMENT krug (centar,(prečnik|poluprečnik)) (#PCDATA)>
```

DTD deklaracija:

```
<!ELEMENT osoba ((ime,prezime)|(prezime,ime))>
<!ELEMENT ime(#PCDATA)>
<!ELEMENT prezime (#PCDATA)>
```

Ispravan XML kod:

```
<osoba>
   <prezime>Jovanović</prezime>
   <ime>Jovan</ime>
</osoba>
```

ili

```
<osoba>
   <ime>Jovan</ime>
   <prezime>Jovanović</prezime>
</osoba>
```

Sledeći primer pokazuje DTD opis jednog telefonskog imenika. Imenik se sastoji od jedne ili više strana. Svaka strana ima zaglavlje, a zatim slede podaci unos ili napomena

DTD deklaracija:

```
<!ELEMENT imenik (strana)+>
<!ELEMENT strana (zaglavlje, (unos|napomena)+)>
<!ELEMENT zaglavlje (#PCDATA)>
<!ELEMENT unos (#PCDATA)>
<!ELEMENT napomena (#PCDATA)>
```

Ispravan XML:

```
<imenik>
   <strana>
      <zaglavlje>Komsije</zaglavlje>
      <unos>Jova Peric, 555-1212</unos>
      <napomena>Broj na poslu - 123-4567</napomena>
   </strana>
 </imenik>
```

Neispravan XML :

```
<imenik><strana><unos/><unos/></strana></imenik>
```

ili

```
<imenik><strana/></imenik>
```

Mešoviti sadržaj opisuje elemente koji sadrže tekstualne podatke i/ili druge elemente. Primer mešovitog sadržaja.

Primer DTD deklaracije:

```
<!ELEMENT proizvod (#PCDATA)>
<!ELEMENT pregled  (#PCDATA | proizvod)*>
<!ELEMENT pregled  (#PCDATA | naziv | zanimanje | napomena)*>
```

Ispravan XML kod:

```
<pregled>Pregledni tekst</pregled>
<pregled>
   Ovo je opis najprodavanijeg
   <proizvod>racunara</proizvod>
   koji je pracen tekstom.
</pregled>
```

Deklarisanje atributa

DTD koristi ključnu reč ATTLIST za deklarisanje atributa elementa. Može se navesti posebno za svaki atribut jednog elementa ili jednom za sve atribute. Dakle, moguće primene su:

```
<!ATTLIST element atribut tip opis>
```

odnosno

```
<!ATTLIST element_atribut tip opis...atribut tip opis>
```

gde je:

`tip` definisani tip atributa.

`opis` opis sadržaja / vrednosti atributa.

Na primer:

```
<!ATTLIST img source CDATA #REQUIRED>
```

Element `img` ima atribut `source`. Tip podataka atributa su znakovni podaci (CDATA). Svi primerci elementa `img` moraju imati neku vrednost za atribut `source`.

Kao što smo rekli, jedna deklaracija `ATTLIST` može deklarisati više atributa istog elementa primenom druge sintakse. Na primer:

```
<!ATTLIST img source CDATA #REQUIRED
             width CDATA #REQUIRED
             height CDATA #REQUIRED
             alt CDATA #IMPLIED >
```

Atribut `alt` je neobavezan i može se izostaviti u nekim elementima.

Ova deklaracija ima isti efekat i značenje kao 4 zasebne deklaracije `ATTLIST`, po jedna za svaki atribut.

Mogući tipovi atributa:

`CDATA` Vrednost je „character data“. Znači da kao vrednost može sadržati proizvoljan tekstualni niz prihvatljiv u dobro formiranom XML-u. Predstavlja osnovni tip atributa.

`(en1|en2|..)` Vrednost mora biti jedna iz liste.

`ID` Vrednost je jedinstvena – id.

`IDREF` Vrednost je id nekog drugog elementa.

`IDREFS` Vrednost je lista drugih id vrednosti.

`NMTOKEN` Vrednost je validno XML ime. Sme da sadrži iste znakove kao i XML ime, tj. alfa-numeričke i/ili ideografske znakove i znakove interpunkcije _,-,. i : Ne sme sadržati beline. Sme početi svim znakovima (za razliku od xml imena). Na primer 12 i .chcrs su validni tokeni.

`NMTOKENS` Vrednost je lista validnih XML imena.

`ENTITY` Vrednost je entitet.

`ENTITIES` Vrednost je lista entiteta.

`NOTATION` Vrednost je naziv notacije.

`xml`: Vrednost je neka predefinisana xml vrednost.

Nabrajanje kao način definisanja tipa atributa

U ovom tipu sadržaja, moguće tj. željene vrednosti se nabrajaju. Vrednosti su ujedno XML imenski tokeni. Ovaj način ne bi bio moguć, ako bi želeli da imena imaju praznine ili neki drugi znak interpunkcije osim donje crte, crtice, dvotačke i tačke.

```
<!ATTLIST datum mesec (januar | februar | mart |
april | maj | jun | jul | avgust | septembar |
oktobar | novembar | decembar) >
```

Tip atributa: ID

Atribut tipa ID mora sadržati XML ime (ne imenski token, već ime) jedinstveno unutar XML dokumenta. Nijedan drugi atribut tipa ID u dokumentu ne može imati istu vrednost.

Primer:

```
<!ATTLIST radnik jmbg ID #REQUIRED >
```

Tip atributa: IDREF

Upućuje na drugi atribut tipa ID nekog elementa u dokumentu.

Mora biti XML ime.

Pogledajmo sledeći XML dokument:

```
<projekat id="p1">
  <naziv>akreditacija</naziv>
  <clan osoba="p2" />
  <clan osoba ="a123451234" />
</projekat>
<projekat id="p2"> <naziv>samovrednovanje</naziv>
  <clan osoba ="a121212121" />
  <clan osoba ="a343434343" />
</projekat>
<radnik jmbg ="a343434343">
  <ime>Pera Peric</ime>
</radnik>
<radnik jmbg ="a123456789">
  <ime>Pera Peric</ime>
</radnik>
```

Neka je ovaj dokument praćen DTD fajlom sadržaja:

```
<!ATTLIST radnik jmbg ID #REQUIRED>
<!ATTLIST projekat id ID #REQUIRED>
<!ATTLIST clan osoba IDREF #REQUIRED>
```

Napomena. Ove deklaracije garantuju da atribut osoba sadrži vrednost koja odgovara nekom atributu u dokumentu koji je tipa ID. Formalno bi to mogao da bude i ID projekta, a ne samo osobe!

Tip atributa: ENTITY, ENTITIES

ENTITY sadrži ime neraščlanjenog (koji se ne parsira) entiteta deklarisanog na drugom mestu DTD-a. Na primer, element *film* može imati atribut ENTITY koji identifikuje MPEG ili QuickTime datoteku koju treba reprodukovati.

ENTITIES sadrži imena jednog ili više neraščlanjenih entiteta, razdvojenih belinama i deklarisanih na drugom mestu.

Na primer:

```
<!ATTLIST film izvor ENTITY #REQUIRED>
<film izvor="supermen 3"/>
<!ATTLIST projekcija slajdovi ENTITIES #REQUIRED>
<projekcija slajdovi="slajd1 slajd2 slajd3"/>
```

Dodatni opis vrednosti atributa

Na kraju deklaracije opisa atributa navodi se dodatni opis. Radi se nekoliko ključnih reči koje imaju posebno značenje u pogledu ograničenja koja se postavljaju za atribute. To su:

`#REQIRED` Atribut mora postojati.

`#IMPLIED` Atribut nije obavezan i nema podrazumevane vrednosti.

`value` Ako vrednost atributa ne postoji, ovaj parametar predstavlja podrazumevanu vrednost.

`#FIXED value` Ako atribut postoji, njegova vrednost mora da odgovara parametru *value*.

Na primer, DTD fajl:

```
<!ELEMENT knjiga (#PCDATA)>
<!ATTLIST knjiga
          id ID #IMPLIED
          isbn CDATA #REQIURED
          tip (ČvrskiPovez|MekiPovez) “MekiPovez”
          adresa CDATA “Beogradska”
          godina CDATA #FIXED “2020”
          komentar CDATA #IMPLIED>
```

Validan XML:

```
<knjiga isbn=“1-35267-742-4“ adresa=“Balkanska">
    XML za programere
</knjiga>
```

Specijalni karakteri – ENTITY

To su promenljive korišćene za definisanje skraćenica ka standardnom tekstu ili specijalnim karakterima.

Interna deklaracija

```
<!ENTITY entity-name "entity-value">
```

Primeri:

```
<!ENTITY pisac "Jovan Jovanović">
<!ENTITY prava "Beogradski izdavački zavod">

<author>&pisac;&prava;</author>
```

Važno: Entity ima 3 dela: ampersend (&), entity ime, i tačka-zarez (;).

Neke mane DTD validacije

Osnovni nedostaci DTD-a su:

- odsustvo tipizacije podataka (#PCDATA može biti bilo koji string),
- sintaksa DTD nije usklađena sa sintaksom XML-a,
- postoje ograničenja koja se ne mogu lako izraziti DTD-om (na pr. element x se može pojaviti od 4 do 17 puta).

Mnoga ograničenja DTD-a uspešno prevazilazi XML Schema (XML šema).

XSD validacija

XML Schema predstavlja XML dokument koji sadrži opis strukture i pravila za jedan XML dokument. Ako XML dokument zadovoljava sva ograničenja zadata šemom onda je **validan** za tu šemu.

XML Schema ili šema dokumenti imaju ekstenziju **.xsd** - pišu se u XSD (eng. ***X****ML* ***S****chema* ***D****efinition*) jeziku.

Osnovne karakteristike šema su:

- Omogućava validaciju XML dokumenta uz puno detalja;
- Podržava sintaksu XML-a;
- Omogućava tipizaciju podataka i prikaz ograničenja. To je svakako jedna od najvažnijih osobina. Na ovaj način XSD omogućava lakšu primenu za: opis dozvoljenog sadržaja tj. tipova, validnost podataka, rad sa podacima iz baza, definisanje ograničenja za podatke, definisanje složenih uzoraka podataka i konverzije između tipova podataka.
- Koristi prostor imena – eng. *namespace*;
    - Prezentuje veze koje postaje između elemenata.

Osnovna struktura XSD dokumenta

Osnovna struktura je data zapisom:

```
<?xml version="1.0"?>
<xs:schema xmlns:xs=http://www.w3.org/2001/XMLSchema>
...
</xs:schema>
```

Kao što se vidi, ključni element je `schema` element iz prostora imena http://www.w3.org/2001/XMLSchema. Pogledajmo sada primer jednog ograničenja u DTD odnosno šema dokumentu:

DTD deklaracija elementa `količina`:

```
<!ELEMENT količina (#PCDATA)>
```

Deklaracija elementa `količina` u šemi:

```
<xsd:schema xmlns:xsd='http://www.w3.org/2001/XMLSchema'>
  <xsd:element name='količina' type="xsd:nonNegativeInteger"/>
</xsd:schema>
```

Na prvi pogled, nešto kraći zapis je kod DTD fajla. Ipak, XSD nudi značajno bolje karakteristike. Na primer, na osnovu DTD deklaracije validan je XML kod:

```
<količina>3</količina>
```

```
<količina>-12</količina>
```

```
<količina>malo</količina>
```

Dok, na osnovu XML šeme validan bi bio samo prvi primer. Pogledajmo sada detaljnije element `schema` tj. njegove atribute. Na primer:

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
targetNamespace=" http://www.mytestschema.rs "
xmlns="http://www.mytestschema.rs"
elementFormDefault="qualified">
</xs:schema>
```

Ovde su iskorišćeni atributi čije je značenje sledeće:

- `xmlns:xs` - Označava da su elementi i tipovi podataka korišćeni u ovoj šemi iz prostora imena "http://www.w3.org/2001/XMLSchema", za koji će se koristiti prefiks **xs.**

- `targetNamespace` - Označava da su elementi definisani sa ovom šemom (note, za, od, zaglavlje, teloPoruke.) iz prostora imena "http://www.mytestschema.rs".
- `xmlns` - Označava da je podrazumevani prostor imena "http://www.mytestschema.rs".
- `elementFormDefault` - Označava da elementi korišćeni od XML dokumenata moraju biti kvalifikovani.

Pridruživanje šeme XML dokumentu

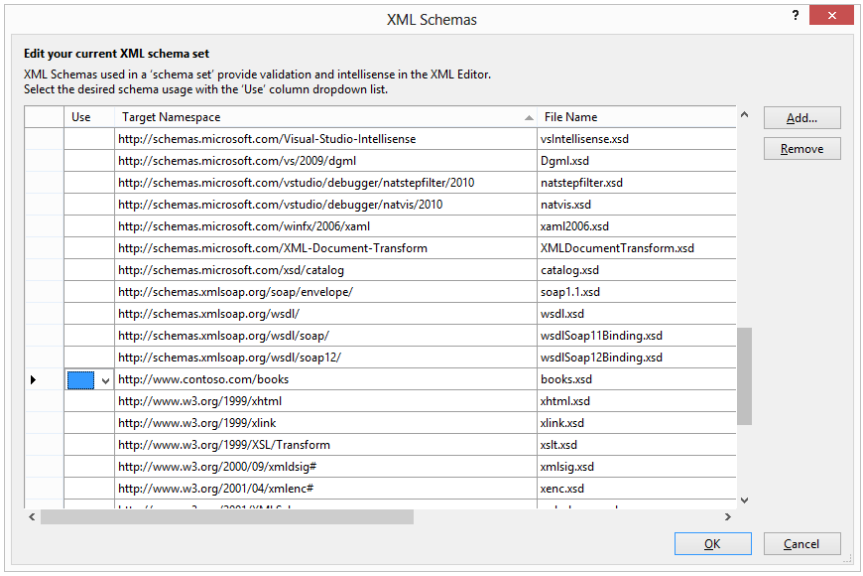
Postoje dva načina kao se jedna šema može pridružiti određenom XML dokumentu.

Prvi način je koristeći atribut `xsi:schemaLocation` koji sadrži dva tokena. Prvi token sadrži URI odredišnog prostora imena. Drugi je fizička lokacija.

```
<stylesheet xmlns="http://www.w3.org/1999/XSL/Transform"
xmlns:html="http://www.w3.org/1999/xhtml"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.w3.org/1999/xhtml
http://www.w3.org/1999/xhtml.xsd" >
```

Druga varijanta zadavanje imenskog prostora je korišćenjem atributa `xsi:noNamespaceSchemaLocation` koji sadrži adresu šeme koju treba upotrebiti za proveru validnosti elemenata a koji ne pripadaju nijednom prostoru imena.

Drugi način je **izdavanjem komande nekom procesoru** za proveru validnosti kojom se proverava validnost datog dokumenta u odnosu na eksplicitno zadatu šemu (a pri tome da zanemari sva podešavanja u samom dokumentu).



Slika 1.2. Šeme uključene u proveru validnosti Visual Studio procesora

Primer povezivanja:

XML dokument:

```
<?xml version="1.0"?>
<note xmlns="http://www.mojprimer.com"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.w3schools.com note.xsd">
<od>Pere</od>
<za>Janu</za>
<zaglavlje>Podsetnik</zaglavlje>
<teloPoruke>Ne zaboravi sastanak!</teloPoruke>
</note>
```

Nad elementom `note` vazi:

`xmlns=http://www.mojprimer.com` - Označava podrazumevani prostor imena.

`xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance` - Kada je xmlns:xsi na raspolaganju može se koristiti atribut *schemaLocation.*

`xsi:schemaLocation="http://www.w3schools.com note.xsd">` -

Prva vrednost ovog atributa je korišćeni prostor imena. Druga vrednost je lokacija XML scheme korišćene u tom prostoru imena. Nad svim podelementima i nadelementom koji su u naznačenom prostoru imena, primenjuje se navedena šema.

Primer za element koji ne pripada imenskom prostoru

```
<?xml version="1.0"?> <punoIme>Jova Ilić</punoIme>
```

Navedeni dokument je dobro formiran XML dokument. Element `punoIme` ne pripada imenskom prostoru. Ako `punoIme` može da sadrži jednostavan znakovni niz, šema će izgledati:

adr-schema.xsd

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="punoIme" type="xs:string"/>
</xs:schema>
```

Objašnjenje. `punoIme` ne pripada nijednom prostoru imena: atribut `noNamespaceSchemaLocation` se primenjuje:

```
<?xml version="1.0"?>
<punoIme xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="adr-schema.xsd">Jovan Jovanovic
</punoIme>
```

Ograničenja i realizacija

Deklaracije elemenata u šemi vrše se na sledeći način:

```
<xs:element name="ime" type="xs:tip">
```

U gornjem primeru, `xs` je prefiks imenskog prostora za XSD šeme. Sadržaj elementa može biti:

1. prost ili
2. složen.

Prost sadržaj se sastoji od teksta bez ugnježdenih elemenata. U tabeli su navedeni neki najčešće korišćeni prosti tipovi definisani u W3C specifikaciji:

Tabela 1.3. Najčešće korišćeni prosti tipovi

`anyURI`	URI identifikator resursa		`duration`	Vremenski period
`boolean`	`true` ili `false`		`integer`	Ceo broj
`dateTime`	datum i vreme zajedno		`string`	Unicode znakovni niz
`date time`	Datum Vreme		`decimal`	Decimalna vrednost

Ako je neki element prost, onda ima samo naziv i tip bez dodatnih atributa. Slede deklaracije atributa pa će biti jasnije o čemu se radi.

Deklaracije atributa

Atributi moraju biti deklarisani kao prosti tipovi. Atributi nemaju dodatnih atributa niti sadržaja. Međutim, elementi koji koriste atribute su složenog tipa. Atributi se deklarišu pomoću elementa:

```
<xs:attribute name="ime" type="xs:tip">
```

Primer:

`<xs:attribute name="id" type="xs:integer"/>` - definisanje atributa `id` koji je ceo broj. Za ovu definiciju jedan primer je:

```
<proizvod id="135">stolica</proizvod>
```

Atributi su podrazumevano opcioni. Ukoliko je atribut obavezan navodi se: `use="required"`

Primer:

```
<xs:attribute name="id" type="xs:integer" use="optional"/>
<xs:attribute name="id" type="xs:integer" use="required"/>
```

Atributi se deklarišu:

- Globalno – unutar elementa najvišeg nivoa i tada mogu biti referencirani na bilo kom mestu u šemi;
- Lokalno – kao deo definicije složenog tipa koja je pridružena određenom elementu.

Ograničenja

Ograničenja kontrolišu vrednosti elemenata odnosno atributa prostih tipova podataka. Koristi se sintaksa:

`xs:restriction`

Jedno ograničene se iskazuje kao podelement unutar ograničenja. Više ograničenja se može kombinovati da bi se dodatno ograničile moguće vrednosti prostog tipa. U nastavku dajemo određena ograničenja:

`xs:minInclusive`, `xs:minExclusive` - najmanje vrednosti stavki na koje se odnose.
`xs:maxInclusive`, `xs:maxExclusive` - najveće vrednosti stavki.

Navedene su po dve varijante restrikcija za min i max vrednosti. Razlika je da li se data vrednost smatra delom skupa dozvoljenih vrednosti ili ne. Na primer:

```
<xs:element name="starost">
     <xs:simpleType>
         <xs:restriction base="xs:integer">
               <xs:minInclusive value="0"/>
               <xs:maxInclusive value="100"/>
          </xs:restriction>
     </xs:simpleType>
</xs:element>
```

Primer: Ekvivalentne deklaracije:

```
<xs:maxInclusive value="0"/>
<xs:maxExclusive value="1"/>
```

`xs:enumeration` - ograničava moguće vrednosti na članove unapred definisane liste. Primer:

```
<xs:element name="smer">
     <xs:simpleType>
          <xs:restriction base="xs:string">
               <xs:enumeration value="NRT"/>
               <xs:enumeration value="RT"/>
               <xs:enumeration value="IS"/>
          </xs:restriction>
     </xs:simpleType>
</xs:element>
```

`xs:pattern` - Ograničenje uzorka zasniva se na primeni regularnih izraza za definisanje mogućih vrednosti.

Na primer, ako je potrebno da se definiše broj socijalnog osiguranja tako da sadrži tri cifre, zatim crticu, dve cifre, novu crticu i još četiri cifre.

```
<xs:simpleType name="bso">
     <xs:restriction base="xs:string">
          <xs:pattern value="\d\d\d-\d\d-\d\d\d\d"/>
     </xs:restriction>
</xs:simpleType>
```

Drugi primer pokazuje regularni izraz za jednu od mogućih vrednosti:

```
<xs:element name="pol">
<xs:simpleType>
     <xs:restriction base="xs:string">
          <xs:pattern value="muski|zenski">
     </xs:restriction>
</xs:simpleType>
</xs:element>
```

Zbog posebnog značaja primene regularnih izraza u nastavku dajemo kratak pregled značenja ovih izraza.

Tabela 1.4. Osnovni regularni izrazi

.	Odgovara jednom, bilo kom, karakteru. Ako se koristi u izrazima u zagradama, tačka odgovara literalnoj tački. Na primer, a.c odgovara "*abc*", etc., ali [a.c] odgovara samo "*a*", ".", ili "*c*".
[ ]	Odgovara karakteru koji je sadržan u zagradama. Na primer, [abc] odgovara "*a*", "*b*", ili "*c*". [a-z] specificira sve karaktere od "*a*" do "*z*". Ovi oblici se mogu mešati: [abcx-z] odgovara "*a*", "*b*", "*c*", "*x*", "*y*", ili "*z*", kao i [a-cx-z]. Karakter – se tretira kao literalni karakter ako je na poslednji ili prvi (posle ^) karakter unutar zagrada: [abc-], [-abc]. Zapazite da *backslash escapes* nisu dozvoljeni. Karakter] može biti uključen u izraz u zagradama ako je prvi karakter (posle ^): []abc].
[^]	Odgovara jednom karakteru koji nije sadržan unutar zagrada. Na primer, [^abc] odgovara bilo kom karakteru drugačije od "*a*", "*b*", ili "*c*". [^a-z] odgovara bilo kom karakteru koji nije malo slovo u opsegu od "*a*" do "*z*". Kao i gore, literalni karakteri i opsezi mogu se mešati.

^	Odgovara početnoj poziciji u stringu. U *line-based* alatkama, odgovara startnoj poziciji u svakoj liniji.
$	Odgovara krajnjoj poziciji stringa ili poziciji pre završetka nove linije. U *line-based* alatkama odgovara krajnjoj poziciji u bilo kojoj liniji.
\d	Odgovara jednoj cifri
*	Prethodni element se pojavljuje nula ili više puta. Na primer, ab*c odgovara "*ac*", "*abc*", "*abbbc*", itd. [xyz]* odgovara "", "*x*", "*y*", "*z*", "*zx*", "*zyx*", "*xyzzy*", i tako dalje. (ab)* odgovara "", "*ab*", "*abab*", "*ababab*", itd.
{m,n}	Odgovara pojavljivanju prethodnog elementa bar *m* puta i ne više od *n* puta. Na primer a{3,5} odgovara samo "*aaa*", "*aaaa*", i "*aaaaa*".

Evo još nekoliko primera regularnih izraza na osnovu gornje tabele:

- `.at` odgovara bilo kom stringu od tri karaktera koji se završava sa "at", na primer "hat", "cat", i "bat".
- `[hc]at` odgovara "hat" i "cat".
- `[^b]at` odgovara svim stringovima kao i .at sa izuzetkom "bat".
- `[^hc]at` odgovara svim stringovima kao .at ali drugačijim od "hat" i "cat".
- `^[hc]at` odgovara "hat" i "cat", ali samo na početku nekog stringa ili linije.
- `[hc]at$` odgovara "hat" i "cat", ali samo na kraju stringa ili linije.
- `\[.\]` odgovara jednom karakteru koji je okružen sa "[" i "]", na primer: "[a]" i "[b]".

Nakon dela o regularnim izrazima, nastavljamo sa objašenjenjima drugih ograničenja.

`xs:length`, `xs:minLength`, `xs:maxLength` Ograničavanje dužine, minimalne dužine, maksimalne dužine podataka.

Na primer:

```
<xs:element name="password">
  <xs:simleType>
    <xs:restriction base="xs:string">
      <xs:length value="8"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

Dužina je 8.

```
<xs:element name="password">
<xs:simleType>
<xs:restriction base="xs:string">
   <xs:minLength value="5"/>
   <xs:maxLength value="8"/>
</xs:restriction>
</xs:simpleType>
</xs:element>
```

Min. dužina je 5 a max. 8.

`xs:whiteSpace` - određuje način na koji se obrađuju beline unutar podataka. Npr. `<xs:whiteSpace value="preserve" />`

`xs:totalDigits` - ograničava ukupan broj cifara u decimalnom broju

`xs:fractionDigits` - zadaje obavezan broj cifara u broju desno od decimalne tačke

Složeni tipovi

Šema pridružuje tip svakom elementu i atributu koji deklariše.

Elementi koji definišu složene tipove mogu imati atribute i mogu sadržati ugnježdene elemente. Samo elementi mogu biti složenog tipa. Tipovi atributa su uvek prosti. Pogledajmo jedan elementarni primer XML dokumenta:

```
<zaposleni>
   <ime>Pera</ime>
   <prezime>Miric</prezime>
</zaposleni>
```

Odgovarajuća XSD šema bi bila.

```
<xs:element name="zaposleni">
   <xs:complexType>
      <xs:sequence>
         <xs:element name="ime" type="xs:string"/>
         <xs:element name="prezime" type="xs:string"/>
      </xs:sequence>
   </xs:complexType>
</xs:element>
```

Kompleksni tip može biti korišćen za više elemenata. U tom slučaju definicija elementa sadrži **referencu** na složeni tip. Pogledati sledeći primer:

```
<xs:element name="zaposleni" type="osobainfo"/>
<xs:element name="student" type="osobainfo"/>
<xs:element name="clan" type="osobainfo"/>

<xs:complexType name="osobainfo"/>
   <xs:sequence>
     <xs:element name="ime" type="xs:string"/>
     <xs:element name="prezime" type="xs:string"/>
   </xs:sequence>
</xs:complexType>
```

Proširivanje/nasleđivanje

Složeni element može proširiti neki postojeći složeni element novim elementima. Pogledajte primer, a dodatno objašnjenje sledi ispod koda.

```
<xs:element name="zaposleni” type="dopunjeninfo"/>
<xs:complexType name="osobainfo"/>
   <xs:sequence>
     <xs:element name="ime" type="xs:string"/>
     <xs:element name="prezime" type="xs:string"/>
   </xs:sequence>
</xs:complexType>

<xs:complexType name="dopunjeninfo"/>
   <xs:complexContent>
      <xs:extension base="osobainfo">
         <xs:sequence>
            <xs:element name="ulica" type="xs:string"/>
            <xs:element name="grad" type="xs:string"/>
            <xs:element name="drzava" type=“xs:string"/>
         </xs:sequence>
       </xs:extension>
   </xs:complexContent>
</xs:complexType>
```

Kreiran je osnovni tip `osobainfo`. Ovaj tip sadrži polja za `ime` i `prezime`. Iz ovog tipa kreira se novi tip `dopunjeninfo`, proširivanjem. Novi tip dodaje nova svojstva: `ulica`, `grad` i `drzava`. Na ovaj način obezbeđuje se bolje projektovanje šema i naravno bolje održavanje.

Referisanje na deklaracije

Jedan tip se može koristiti za proširivanje, ali može da učestvuje i kao celina u novim deklaracijama. Referisanje na već definisane tipove doprinosi boljoj organizaciji šema i boljem održavanju. Pogledajte sledeći primer:

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

<xs:element name="poruka">
  <xs:complexType>
    <xs:sequence>
    <xs:element ref="za"/>
    <xs:element ref="od"/>
    <xs:element ref="zaglavlje"/>
    <xs:element ref="teloPoruke"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="za" type="xs:string"/>
<xs:element name="od" type="xs:string"/>
<xs:element name="zaglavlje" type="xs:string"/>
<xs:element name="teloPoruke" type="xs:string"/>
</xs:schema>
```

U prethodnom primeru referisanje je korišćeno da pokaže način upotrebe referenci. Ovako pisanje doprinosi jasnoći i razumevanju, naravno, kasnije je lakše vršiti izmene.

Ograničenja broja pojavljivanja

Eksplicitno zadavanja najmanjeg i najvećeg broja pojavljivanja nekog elementa na nekom mestu u dokumentu, vrši se koristeći sledeće atribute elementa `xs:element`:

1. `minOccurs` – definiše minimalan broj pojavljivanja elementa,
2. `maxOccurs` – definiše maksimalan broj pojavljivanja elementa.

Podrazumevana vrednost za oba atributa je **1**. Na primer:

```
<xs:element name="student">
   <xs:complexType>
      <xs:sequence>
         <xs:element name= "brIndeks" type="xs:string" />
         <xs:element name= "telefon" type="xs:string"
         minOccurs= "1"  maxOccurs= "10" />
      </xs:sequence>
   </xs:complexType>
</xs:element>
```

Mešoviti sadržaj elemenata

Mešoviti sadržaj imaju elementi koji sadrže stringove, tj. tekst ali i druge elemente.

XML šema omogućava definisanje ove funkcionalnosti kao i precizno definisanje broja i redosleda elemenata unutar znakovnih podataka.

Atribut `mixed` elementa tipa `complexType` određuje da li se znakovni podaci smeju pojaviti unutar tela elementa. Na primer, za xml dokument:

```
<pismo>
   Dragi<ime> Pero</ime>
   Vasa narudzbina <narid>1432 </narid>
   ce stici<nardatum> 2007-06-18</nardatum>
</pismo>
```

Odgovarajuća xsd šema za validaciju bila bi:

```
<xs:element name="pismo" type="pismoTip"/>
<xs:complexType name="pismoTip" mixed="true">
   <xs:sequence>
      <xs:element name="ime" type="xs:string"/>
      <xs:element name="narid" type="xs:positiveInteger"/>
      <xs:element name="nardatum" type="xs:date"/>
   </xs:sequence>
</xs:complexType>
```

Zadavanje redosleda elemenata

Za definisanje redosleda elemenata, XML šema nudi:

1. xs:sequence
2. xs:choice
3. xs:all

Ove vrste rasporeda mogu da se ugnježdavaju i na taj način definišu složenije strukture. Prikazaćemo jednostavnije i složenije slučajeve kroz naredne primere.

xs:sequence

Ovaj element definiše koje elemente sadrži složeni element i tačan redosled kojim se ti elementi pojavljuju.

```
<xs:element name="pismo">
   <xs:complexType  mixed="true">
      <xs:sequence>
         <xs:element name="pozdrav" />
         <xs:element name="telo" />
         <xs:element name="zavrsetak" />
      </xs:sequence>
    </xs:complexType>
</xs:element>
```

Pismo mora sadržati element `pozdrav`, element `telo` i element `završetak`, tim redom.

xs:choice

Definiše pojavljivanje samo jedne vrednosti iz niza navedenih:

```
<xs:element name="pozdrav">
   <xs:complexType  mixed="true">
      <xs:choice>
         <xs:element name="zdravo" />
         <xs:element name="cao" />
         <xs:element name="draga" />
      </xs:choice>
    </xs:complexType>
 </xs:element>
```

Pozdrav mora da sadrži samo jednu iz liste dozvoljenih pozdravnih reči (`zdravo`, `cao` ili `draga`).

xs:all

Definiše da se svaki od elemenata mora pojaviti jednom. Redosled pojavljivanja nije važan. Pogledajmo primer jednog cirkularnog pisma koje se sastoji od šeme i xml dokumenta.

cirpismo.xsd

```
<xs:element name="telo">
   <xs:complexType  mixed="true">
      <xs:all>
         <xs:element name="stavka"/>
         <xs:element name="datumPrispeca"/>
         <xs:element name="cena"/>
      </xs:all>
    </xs:complexType>
 </xs:element>
```

cirpismo.xlm

```
<pismo xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:noNamespaceSchemaLocation="cirpismo.xsd">
   <pozdrav> <zdravo/>Bobo! </pozdrav>
   <telo>
         Hvala Vam sza se narucili <stavka/>
         (<cena/>). Trebalo bi da stigne do <datumPrispeca/>
   </telo>
   <zavrsetak/>
 </pismo>
```

Redosled pojavljivanja elemenata `stavka`, `cena` i `datumPrispeca` nije bitan, a sprečeno je i pojavljivanje više referenci iste vrednosti.

JSON

JSON (eng. *JavaScript Object Notation*) predstavlja format za razmenu podataka. Jednostavan i lak za razumevanje za programe i za čoveka. Jednostavan za dalju obradu u veb aplikacijama. Računari ga lako parsiraju. Bazira se na podskupu JavaScript programskog jezika, standard ECMA-262– decembar 1999.

JSON predstavlja tekstualni zapis podataka u obliku: `naziv:vrednost`.

Nezavistan je od korišćenog programskog okruženja i jezika. 2002. godine ozvaničava se JSON.org, a 2005. godine postaje ključni deo ideje AJAX-a.

Nije markap jezik! JSON se bazira na dve programske strukture:

- Složena vrednost, tj. objekat, zapisuje se kao skup parova naziv-vrednost, a ceo skup je oivičen vitičastim zagradama.
- Uređena lista. U najvećem broju jezika, ovo se realizuje kao niz, vektor, lista, ili sekvenca. U slučaju niza formira se uređena kolekcija vrednosti, npr: `["Pera","Mika","Zika"]`.

To su univerzalne strukture podataka. Svi moderni programski jezici podržavaju ih u nekoj formi. Upravo iz tog razloga i ima smisla da format za razmenu podataka bude baziran na tim strukturama.

Primeri:

```
{ "studenti": [
   { "ime":"Pera" , "prezime":"Peric" },
   { "ime ":"Mika" , "prezime":"Mikic" },
   { "ime ":"Zika" , "prezime":"Zikic" }
]}
```

ili

```
{
    "ime": "Jovan",
    "prezime": "Jovanovic",
    "godine": 35,
    "adresa": {
        "ulica": "Lipa 33",
        "grad": "Beograd"
 },
"telefoni": [
        {
            "tip": "kuca",
            "broj": "011 555 5555"
        },
        {
            "tip": "fax",
            "broj": "011 777 7777"
        }
    ]
}
```

Tipovi podataka

Tipovi koji se mogu koristiti u JSON zapisu podataka su:

1. `Number` (JavaScript format u pokretnom zarezu sa dvostrukom preciznošću, zavisi od implementacije)
2. `String` (Unicode format, sa dvostrukim navodnicima, kao izlazna sekvenca se koristi backslash)
3. `Boolean` (`true` ili `false`)
4. `Niz` (uređena sekvenca vrednosti, odvojena zarezima i uokvirena kockastim zagradama; vrednosti ne moraju biti istog tipa)
5. `Objekat` (neuređena kolekcija ključ:vrednost parova sa ':' karakterom koji razdvaja ključ i vrednost, razdvojeni zarezima i uokvireni vitičastim zagradama; ključevi moraju biti niske i različiti od ostalih ključeva)
6. `null` (prazno)
7. Bezznačajne beline se mogu slobodno dodati između strukturalnih karaktera (zagrada "{ } [ ]", dve tačke ":" i zareza ",").

   Potencijalni problem nastaje zbog slobode pisanja brojeva u JSON-a. Brojevi se mogu zapisati kao numerički literali ili nizovi pod navodnicima. Na primer poštanski kod počinje sa nulama (na primer 011 za Beograd). Ako jedan programer piše pod navodnicima a drugi ne, vodeća nula se može izgubiti prilikom razmene podataka ta 2 sistema.

JSON i XML

Najčešće se nudi kao alternativa XML-u. Sličnosti između ovih formata su:

- Oba su bliska čitanju i tumačenju čoveka;
- Oba imaju vrlo prostu sintaksu;
- Oba su hijerarhijski organizovana;
- Oba su nezavisna od jezika, oba se mogu koristiti za AJAX;

Razlike su:

- Sintaksa je različita JSON je manje opisan;
- JSON može biti parsiran sa JavaScript eval metodom;
- Name u JSON-u ne sme biti JavaScript rezervisana reč;

- XML može biti proverljiv.

Uporedimo formate:

JSON:

```
ime: Jovan
prezime: Jovanovic
godine: 33
address:
   ulica: Balkanska
   grad: Beograd
telefoni:
   -
   tip: kuća
   broj: 011 555 5555
   –
   type: fax
   broj: 011 777 7777
```

XML -1:

```
<osoba>
  <ime> Jovan </ime>
  <prezime> Jovanovic </prezime>
  <godine> 33 </godine>
  <adresa>
    <ulica>Balkanska</ulica>
    <grad>Beograd</grad>
  </adresa>
  <telefoni>
    <broj tip="kuća">011 555 5555</broj>
    <broj tip="fax">011 777 7777</broj>
  </telefoni>
</osoba>
```

XML -2:

```
<osoba ime="Jovan" prezime="Jovanovic" godine="33">
  <adresa ulica="Balkanska" grad="Beograd" />
  <telefoni>
    <telefon tip="kuća" broj="011 555 5555"/>
    <telefon tip="fax" broj="011 777 7777"/>
```

```
  </telefoni>
</osoba>
```

JavaScript

Jedna od glavnih prednosti JSON formata je njegova prirodna povezanost sa JavaScript jezikom. Objekat u JavaScriptu se definiše kao jedan JSON podatak. Takođe, svaki objekat ima svoju stringovsku reprezentaciju u vidu JSON podatka.

JavaScript ima i jaku podršku za rad sa XML podacima. Neki slučajevi zasnovani na razmeni XML podataka od velike su važnosti za komunikaciju na vebu. Na primer:

```
function myHandler(){
 if (req.readyState == 4 /*complete*/){
   var addrField = document.getElementById('addr');
   var root = req.responseXML;
   var addrsElem = root.getElementsByTagName('addresses')[0];
   var firstAddr = addrsElem.getElementsByTagName('address')[0];
   var addrText = fistAddr.firstChild;
   var addrValue = addrText.nodeValue;
   addrField.value = addrValue;
}}
```

Rad sa JSON podacima

```
function myHandler()
{
   if (req.readyState == 4 /*complete*/)
   {
      var addrField = document.getElementById('addr');
      var card = eval('(' + req.responseText + ')');
      addrField.value = card.addresses[0].value;
```

```
    }
}
```

Šema

JSON šema je specifikacija za formate zasnovane na JSON-u koja definiše strukturu podataka JSON dokumenta. JSON šema pruža garancije za prirodu i strukturu JSON podataka, odnosno garantuju formu podataka koja je potrebna u datoj aplikaciji i način kako se mogu menjati. JSON šema ima za cilj da obezbedi validaciju, dokumentaciju i kontrolu interakcija sa JSON podacima. Zasnovana je na konceptima XML šema. Njihova svrha je da se JSON podaci (podaci šeme) mogu koristiti za validaciju JSON dokumenata. Dakle, isti alati se mogu koristiti za serijalizaciju odnosno deserijalizaciju šema i podataka.

JSON šema je napisana kao Internet nacrt , trenutna verzija 4. Na raspolaganju postoji nekoliko validatora za različite programske jezike, svaki sa različitim nivoom prilagođavanja. Primer JSON šeme:

```
{
  "name": "Product",
  "properties": {
    "id": {
      "type": "number",
      "description":"opis..",
       "required": true
    },
    "name": {
      "type": "string",
      "description": "opis…",
       "required": true
     },
     "price": {
       "type": "number",
        "minimum": 0,
        "required": true
      },
}
```

```
"tags":
{
   "type": "array",
   "items": {
       "type": "string"
    }
},
"stock": {
   "type": "object",
   "properties": {
           "warehouse": {
              "type": "number"
            },
            "retail": {
                type": "number"
             }
           }
       }
    }
}
```

JSON i Ajax

Ajax predstavlja mogućnost veb stranice da traži nove podatke nakon učitavanja stranice pregledača, najčešće kao odgovor na akcije korisnika u okviru same stranice. JSON se često koristi uz Ajax. Kao deo Ajax modela, novi podaci se obično pripajaju u korisnički interfejs dinamično kako pristižu sa servera. Primer ovoga u praksi bi bio da korisnik kuca u polje za pretragu, klijentska strana šalje šta je otkucano do tada serveru koji odgovara sa listom mogućnosti koje se poklapaju sa pretragom u bazi podataka. Ovo se može predstaviti u padajućem meniju, te korisnik može zaustaviti svoje kucanje i izabrati nešto od ponuđenog. Kada se pojavio sredinom 2000-tih, Ajax se najčešće oslanjao na XML kao format za razmenu podataka, ali su mnogi programeri koristili upravo JSON za razmenu Ajax ažuriranja između servera i klijenta. Naredni JavaScript kod je primer klijentskog korišćenja XMLHttpRequest zahteva u JSON formatu od servera.

```
var my_JSON_object = {};
var http_request = new XMLHttpRequest();
http_request.open("GET", url, true);
http_request.onreadystatechange = function () {
 var done = 4, ok = 200;
 if (http_request.readyState == done && http_request.status ==
ok) {
    my_JSON_object = JSON.parse(http_request.responseText);
 }
};
http_request.send(null);
```

Pitanja i zadaci

1. Šta je XML i koja je namena?

2. Objasniti strukturu XML dokumenta.
3. Kako se kreira mešoviti sadržaj?
4. Šta su atributi i kako birati između atributa i elemenata ako kreiramo sopstveni XML dokument?
5. Kako se pišu komentari?
6. Šta je CDATA sekcija i kako se koristi?
7. Šta su instrukcije obrade i koje poznajete?
8. Kako se novi red tretira u XML dokumentima?
9. Šta je prostor imena i kako se koristi?
10. Šta je kvalifikovano ime?
11. Šta je podrazumevani prostor imena? Kako se postavljaju elementi u podrazumevani prostor imena?
12. Kako se postavlja neki element da bude nekvalifikovan?
13. Šta je DTD dokument i koja je njegova namena?
14. Kako se povezuje DTD dokument sa XML dokumentom?
15. Šta su javni identifikatori? Da li poznajete neki?
16. Koja je razlika u primeni unutrašnjeg od spoljašnjeg DTD dokumenta?
17. Kako se deklarišu elementi pomoću DTD dokumenta?
18. Objasniti kako se definiše struktura sadržaja elementa.
19. Kako se deklarišu atributi?
20. Objasniti tip atributa ID.
21. Objasniti tip atributa IDREF.
22. Objasniti tip atributa ENTITY.
23. Napisati primer artikala sa postojanjem identifikatora.
24. Napisati primer narudžbina sa korišćenjem artikala tj. IDREF.
25. Šta znači kada se kaže da je dokument dobro formiran?

26. Šta znači kada se kaže da je dokument validan?
27. Ako govorimo o XHTML onda se radi o posebnom HTML jeziku. Objasniti detalje.
28. Uporediti XSD validaciju sa DTD validacijom.
29. Kako se XSD šema pridružuje XML dokumentu?
30. Kako se postavljaju ograničenja u XSD šemi i koja ograničenja postoje?
31. Kako se deklarišu atributi u šemi?
32. Šta su složeni a šta su prosti tipovi?
33. Kako se vrši proširivanje tj. nasleđivanje tipova?
34. Objasniti kako se vrši referisanje na kreirane deklaracije?
35. Objasniti način ograničavanja broja pojavljivanja?
36. Kako se kreira mešoviti sadržaj?
37. Kako se opisuje redosled elemenata i koje definicije postoje?
38. Kako se definiše sekvenca?
39. Kako se definiše izbor jedne stavke iz skupa?
40. Kako se definiše sadržaj koji čine elementi koji su iz navedenog skupa, bez obzira na redosled?
41. Šta znači JSON?
42. Napiši JSON zapis podataka koji se tiču artikala i porudžbine.
43. Kako se JSON koristi u JavaScript jeziku? Napiši primer.
44. Kako se XML koristi u Java JavaScript jeziku? Napiši primer.
45. Da li JSON može da strogo definiše strukturu podataka?
46. Šta je šema za JSON?

2. Sistemi za verzioniranje

U okviru ovog poglavlja definiše se pojam i namena sistema za verzioniranje. Posebno se objašnjavaju osnovne funkcije ovakvih sistema: čuvanje istorije, timski rad, grananje, spajanje, skaliranje i rad sa spoljašnjim učesnicima. Uvodi se grafički prikaz i oznake, prikazuju se tipični slučajevi za razumevanje čuvanja verzija. Zatim se prikazuje postupak kreiranja nove verzije kao i opis sadržaja jednog repozitorijuma.

Osnove

Upravljanje promenama grupe dokumenata, na primer tokom razvoja nekog softverskog projekta, obuhvata kreiranje novih verzija, spajanje različitih ili povratak na neku prethodnu verziju. Jednim imenom ovo se naziva se **verzioniranje.** Tipično, potreba za verzioniranjem nastaje u toku razvojnog ciklusa jednog rešenja, na primer tokom razvoja sajta, ali i u drugim slučajevima kada postoji potreba za upravljanjem većim brojem dokumenata. Sistemi koji omogućavaju rad sa verzijama nazivaju se sistemi za verzioniranje - **VCS** sistemi (eng. **V**ersion **C**ontrol **S**ystems).

Promene tj. nova verzija se obično identifikuje određenim brojem ili slovom označenim kao broj izmene (eng. *revision number*). Na primer,

početni skup fajlova je „revision 1“. Kada se urade prve izmene, rezultujući skup promena daje „revision 2“ itd.

Svaka verzija je označena sa odgovarajućom vremenskom oznakom tj. pečatom (eng. *timestamp*), kao i korisničkim imenom osobe koja izvodi izmene.

Jedan od najpoznatijih VCS sistema je Git. Git će biti obrađivan detaljno u narednim poglavljima. Tvorac projekta Git smatra se Linus Torvards. Naravno, razvijen je kao projekat otvorenog koda sa ciljem lakšeg razvoja i vođenja projekta Linux. Zbog sličnih potreba, veoma brzo pokazuje se da je za njega zainteresovana brojna zajednica, tako da ubrzo postaje veoma popularan i masovno primenjivan. Tako, prateći zahteve svojih brojnih korisnika Git se vremenom razvijao zajedno sa drugim tehnikama i tehnologijama, pa da danas predstavlja najzastupljeniji VCS sistem.

Uloga jednog VCS sistema je višestruka. Sistem obezbeđuje:

- Čuvanje istorije dokumenata;
- Timski rad;
- Grananje;
- Rad sa spoljnim učesnicima;
- Skaliranje.

Pogledajmo redom svaku od ovih uloga.

Čuvanje istorije

Čuvanje istorije znači da se sve promene na dokumentima čuvaju od samog početka. One se mogu pratiti kroz verzije. Istorija sadrži podatke:

- Kada je verzija urađena?
- Ko je uradio izmene?
- Šta je izmenjeno?
- Zašto je menjano?
- U kom kontekstu se izmene događaju, tj. šta je bilo ispred i šta se događalo nakon toga.

Izuzetno je važno je da je u svakom trenutku moguće uraditi povratak na neku od prethodnih verzija.

Sav izbrisan sadržaj ostaje dostupan kroz istoriju.

Rad u timu

Svi veći projekti rade se timski. Timovi mogu biti od svega nekoliko ljudi pa sve do desetina i hiljada. Sam Git je nastao iz potrebe timskog rada i upravljanja verzija. U tom smislu VCS sistemi omogućavaju:

- Deljenju kolekcije fajlova sa ostalim učesnicima tj. članovima tima. Mogu se definisati verzije za pojedine timove, takođe moguće je izostaviti fajlove koje ne treba menjati i slično.
- Spajanje promena koje su nastale od drugih učesnika ili timova.
- Osiguravanje da se ništa ne može slučajno izgubiti ili preklopiti.

Grananje

Toko razvoja, sa jednim ili više učesnika, važno je da se omogući rad na više verzija istog istovremeno. Ovo se ostvaruje preko grananja. Na primer, pri razvoju jednog programa mogu se napraviti sledeće grane:

- Glavna grana.
- Grana za održavanje (grana koja omogućava ispravke grešaka u starijim izdanjima).
- Grana za razvoj programa.
- Grana za novo izdanje (gde se vrši zamrzavanje koda pre novog izdanja).

Rad sa spoljnjim učesnicima

Alati za verzioniranje pomažu u razvoju u kome učestvuju spoljnji saradnici tzv. saradnici trećih strana (eng. *third-party contributors*). Ovi alati omogućavaju:

- Uvid u ono što se događa u razvoju tj. u projektu;
- Pomaže im da urade i integrišu izmene (zakrpe);

- Grananje razvoja softvera i njegovo spajanje u glavnu liniju.

Skaliranje

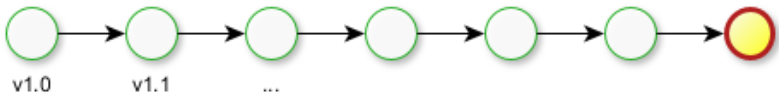
VCS sistemi omogućavaju rad na velikim projektima i tzv. skaliranje. Neki podaci (izvor: Linux Foundation) kazuju da Linux kernel, razvijan primenom GIT-a:

- ima oko 10000 promena u svakoj novoj verziji, na svaka 2-3 meseca
- 1000+ saradnika

Primenom verzioniranja omogućava se razvoj i skaliranje projekata čak i u slučaju tako velikog broja učesnika kao što je slučaj sa Linux razvojem.

Grafički prikaz i tipični slučajevi

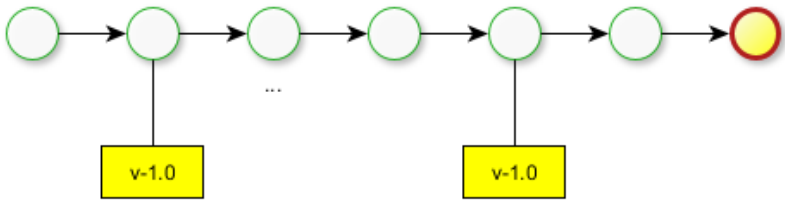
Verzioniranje se često prikazuje grafički. Razlog je što grafički prikaz daje jasno postupak promene tj. prelaz iz verzije u verzije, grananje i spajanje. Grane su tipično označene strelicama koje povezuju stanja koja su prikazana grafički kao kružići sa pripadajućom labelom koja označava naziv verzije.



Slika 2.1. Repozitorijum koji sadrži istoriju počev od verzije v1.0

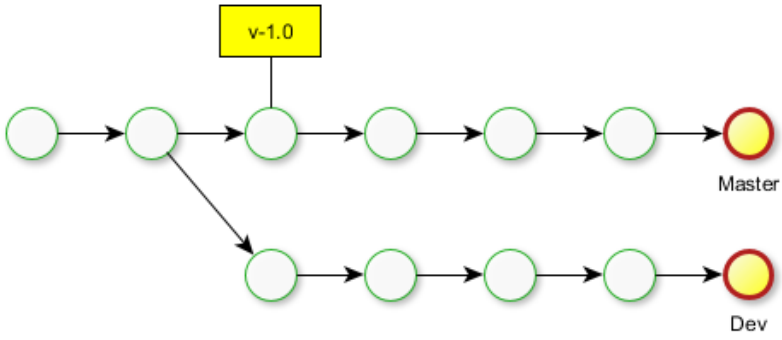
Tekuća verzija se posebno prikazuje i obično se označava kao „HEAD“.

Svaka verzija nosi prateći opis. Međutim, postoje posebne verzije koje su od posebnog značaja i koje se posebno označavaju. Obično su to verzije softvera koje se objavljuju (eng. *release version*).



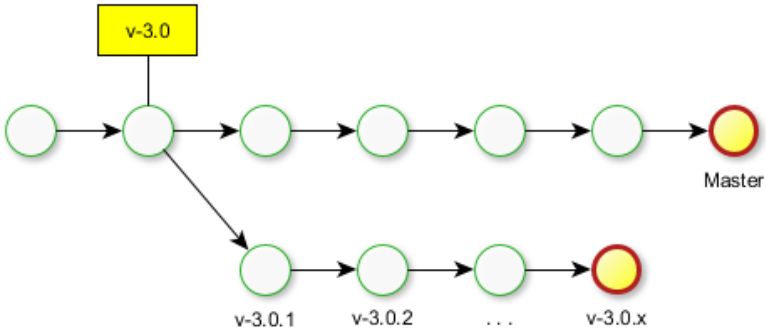
Slika 2.2. Niz verzija sa posebno označenim verzijama

Grananje predstavlja razdvajanje dokumenata u više verzija. Događa se tokom procesa razvoja projekta kada se istovremeno radi u više pravaca. Svaki od pravaca ima sopstvene verzije odnosno istoriju.



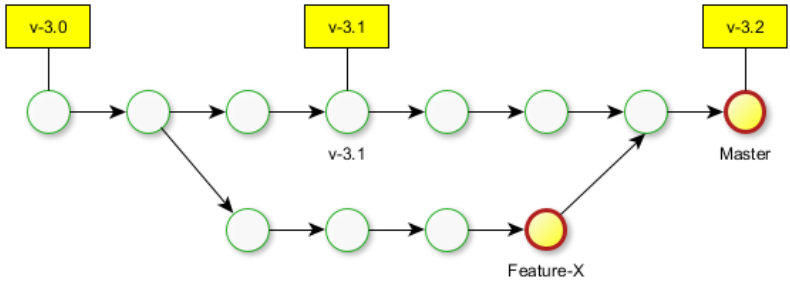
Slika 2.3. Grananje razvoja u dve grane: *Master* i *Dev*

Jedna tipična grana je glavna (Master) grana sa verzijama koje sadrže osnovne karakteristike tokom razvoja. Druga, koja se u praksi koristi, je grana koja se vezuje za objavljenu verziju softvera za koju se vrši ispravka grešaka.



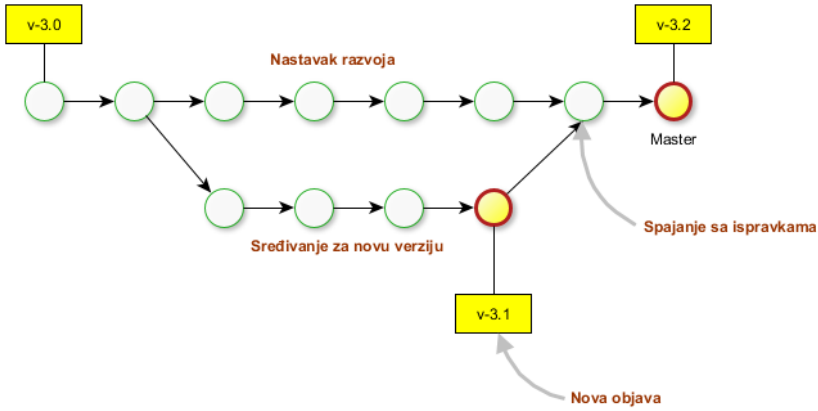
Slika 2.4. Grana za ispravljanje grešaka za objavljenu verziju v-3.0

Takođe, pri generisanju novih karakteristika za neku postojeću verziju vrši se izdvajanje posebne grane. U toj grani se vrši razvoj novih karakteristika, kao i eventualno ispravljanje grešaka u nekoj podgrani, a na kraju kada se karakteristika uradi, vrši se spajanje (eng. *merging*) sa glavnom granom.



Slika 2.5. Grana izdvojena za razvoj nove karakteristike *Feature-X* a zatim spajanje sa glavnom granom.

Kada se ostvari razvoj sa dovoljno karakteristika za novu objavu, nakon toga su prihvatljive jedino izmene koje se tiču ispravki grešaka. Zato se nakon dostizanja željenih karakteristika, a pre objave tj. pre ispravke grešaka vrši odvajanje u granu za objavu (eng. *release branch*). Istovremeno nastavlja se razvoj na glavnoj grani.



Slika 2.6. Grananje i spajanje grana

Kada se razvoj završi, vrši se spajanje sa glavnom granom na koju se prebacuju sve ispravke tj. promene koje su nastale u grani za objavu. Takođe, grana za objavu opet postaje grana za održavanje tj. ispravku grešaka.

Organizacija verzioniranja

U osnovi postoje dve vrste organizacije rada više korisnika sa repozitorijumom.

- **Centralizovana** - Podrazumeva da svaki korisnik radi sa istim repozitorijumom.
- **Decentralizovana** - Podrazumeva da svaki korisnik radi sa sopstvenim repozitorijumom.

Razlikuju se dva načina rešavanja konflikta istovremenog rada na istom projektu. Jednostavniji, ali istovremeno manje efikasan metod, jeste **zaključavanje pre izmena**. Drugi način je **spajanje promena**. Drugi način je efikasniji u pogledu neprekidnog razvoja softvera, ali se mora voditi računa o eventualnim konfliktima. Git verzioniranje podrazumeva

decentralizovan sistem sa spajanjem. Git omogućava pristup bilo kojoj grani tokom razvoja kao i praćenje svih izmena u svakoj fazi razvoja.

Svaki repozitorijum mora imati na raspolaganju odgovarajući prostor za skladištenje kako bi bilo moguće skladištiti izmene u svakom fajlu projekta.

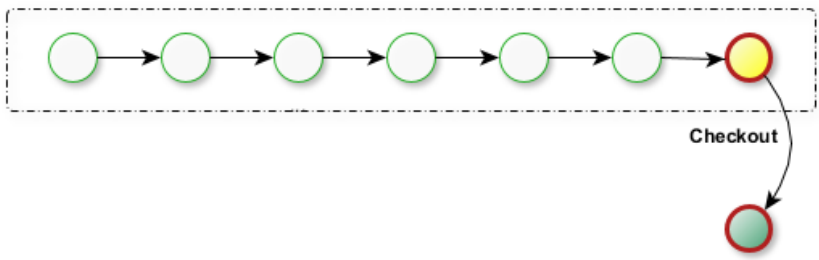
Većina alata za verzioniranje koristi delta kompresiju kako bi optimizovao prostor za skladištenje, osim Git-a koji koristi tzv. objektno pakovanje.

Svaki repozitorijum se identifikuje odgovarajućim URL-om. Alati za verzioniranje koriste više načina za interakciju sa udaljenim repozitorijumima. Obično se koriste standardni protokoli **http** ili **https**, ali i zaštićeni poput **ssh**. U nekim slučajevima moguća je upotreba specifičnih poput **svn** ili **git** protokola.

Postupak kreiranja nove verzije

Repozitorijum predstavlja jedinstvenu **celinu koja se ne menja direktno**. Promena, odnosno kreiranje nove verzije, vrši se u nekoliko koraka.

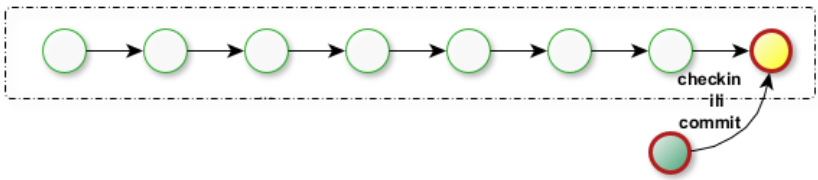
Korak 1. Preuzimanje (lokalne) kopije fajlova koji se koriste odnosno menjaju. Ova se postiže komandom **checkout**.



Slika 2.7. Grafički prikaz preuzimanja poslednje verzije

Korak 2. Sledi rad i izmene na preuzetim kopijama fajlova, tj. kreiranje verzija koje se oslanjaju na ovu preuzetu. Istovremeno radi se i na glavnoj grani.

Korak 3. Kada je radna kopija fajlova spremna za postavljanje tj. za evidentiranje nove verzije izvodi se komanda commit.



Slika 2.8. Urađene izmene se vraćaju na repozitorijum

Koraci od 1 do 3 se zatim više puta mogu ponoviti i tako formirati više verzija.

Sadržaj repozitorijuma

Na repozitorijumu se čuvaju svi fajlovi koji se ne generišu od strane alata koji se koriste u razvoju. Takvi fajlovi su:

- fajlovi izvornog koda (.c .cpp .java . . .),
- skripte za izradu projekta (project.sln makefile configure.in . . .),
- fajlovi koji su dokumenta koja prate projekat (.doc .txt . . .),
- fajlovi koji predstavljaju prateće resurse (ikone, slike, audio, . . .).

Fajlovi koje ne treba čuvati su fajlovi koji se generišu. Takvi su:

- .obj .exe .o .dll .class .jar
- fajlovi op. sistema ili skripti ali koje generišu alati.

Čuvanje ovakvih fajlova izazvaće pojavu nepotrebnih konflikata pri spajanju različitih verzija.

Pitanja i zadaci

1. Šta su sistemi za verzioniranje i koja je njihova namena?
2. Objasniti šta znači čuvanje istorije?
3. Kako VCS obezbeđuje timski rad?
4. Kako se grafički predstavlja i koja je namena grananja?
5. VCS obezbeđuje i rad spoljašnjih učesnika, kako?
6. Uloga VCS sistema je i u skaliranju, objasni.
7. Postoje dve osnovne organizacije VCS sistema. Koje su to?
8. Koji se protokoli koriste?
9. Objasni postupak kreiranja nove verzije.
10. Šta čini novu verziju odnosno sadržaj repozitorijuma?

3. Osnove Git-a

Treće poglavlje posvećeno je najpoznatijem VCS sistemu – Git. Čitaocu se najpre daju osnovni podaci o nastanku Git-a kao i istorija razvoja verzija ovog sistema. Sa praktičnom primenom kreće se opisom postupka instalacije Git-a kao i proverom instalirane verzije.

Početak rada sa Git-om opisan je komandom za pomoć, zatim sledi opis načina konfigurisanja kao i tipova konfiguracionih nivoa. Slede komande za prikaz statusa repozitorijuma, za dodavanje fajla na scenu, snimanje nove verzije.

Osim standardnog snimanja pokazano je i privremeno ili tajno skladištenje verzija. Na kraju je dat opis brisanja. Pošto je brisanje osetljiva operacija, tj. se pogrešnim brisanjem mogu se izgubiti podaci od značaja, detaljno se obrađuju komande brisanja i čišćenja indeksa.

Rad na Git-u nastavlja se opisom komande za upoređivanje fajlova odnosno spajanje grana u slučaju konflikata. Zatim se detaljno objašnjava pojam reference i rad sa HEAD referencom. Daje se prikaz i pretraga istorije promena. Uvodi se pojam označavanja tj. tagovanja.

Komanda reset se detaljno opisuje, imajući u vidu da se njenom primenom mogu izgubiti neki podaci odnosno da nastaju različite promene u zavisnosti od pratećih opcija. U nastavku, prikazuju se i druge komande za prevazilaženje grešaka tokom objave verzija.

Istorijat

Razvoj počinje u aprilu 2005. pošto je usledio opoziv od *free-use* licence Bitkeeper-a, na kome je počivao razvoj Linux-a. Nijedan alat nije bio dovoljno napredan da bi zadovoljio Linux-ov razvoj sa ograničenjima (distribuiran rad, integritet, performanse), pa Linus Torvald počinje sopstveni razvoj sistema za verzioniranje - Git. Ubrzo sledi i prva verzija. U junu 2005. sledi prvo izdanje Linux-a kojim je upravljao Git. Zaim, Torvald predaje održavanje Gita Juniu Hamanu, glavnom saradniku na projektu, a već u decembru 2005. sledi i zvanična Git 1.0 verzija. Sledi brz i uspešan razvoj Gita.

Danas, git ima brojnu zajednicu koja se brine o njemu. Svake godine zvanično se objavi nekoliko novih verzija, pogledajte tabelu sa listom verzija u par prethodnih godina.

Tabela 3.1. Verzije i datumi objave

Verzija	Datum objave	Poslednja verzija
2.16	2018-01-17	2.16.6
2.17	2018-04-02	2.17.5
2.18	2018-06-21	2.18.4
2.19	2018-09-10	2.19.5
2.20	2018-12-09	2.20.4
2.21	2019-02-24	2.21.3
2.22	2019-06-07	2.22.4
2.23	2019-08-16	2.23.3

2.24	2019-11-04	2.24.3
2.25	2020-01-13	2.25.4
2.26	2020-03-22	2.26.2
2.27	2020-06-01	2.27.0

Instalacija

Pre upotrebe, Git je potrebno instalirati. Zvanična lokacija za preuzimanje Gita, za Windows operativni sistem, je na adresi: http://git-scm.com/download/win . Pošto posetite navedenu lokaciju, preuzimanje će biti pokrenuto. Instalacija je moguća i koristeći instalacionu verziju sa lokacije **https://git-for-windows.github.io/**. Za ovu lokaciju, na zvaničnom sajtu stoji napomena da je projekat za Windows operativni sistem nezavisan i naziva se **Git for Windows**. Drugi način je putem instalacije GitHub-a. Ovu instalaciju potražite na adresi **https://desktop.github.com/.**

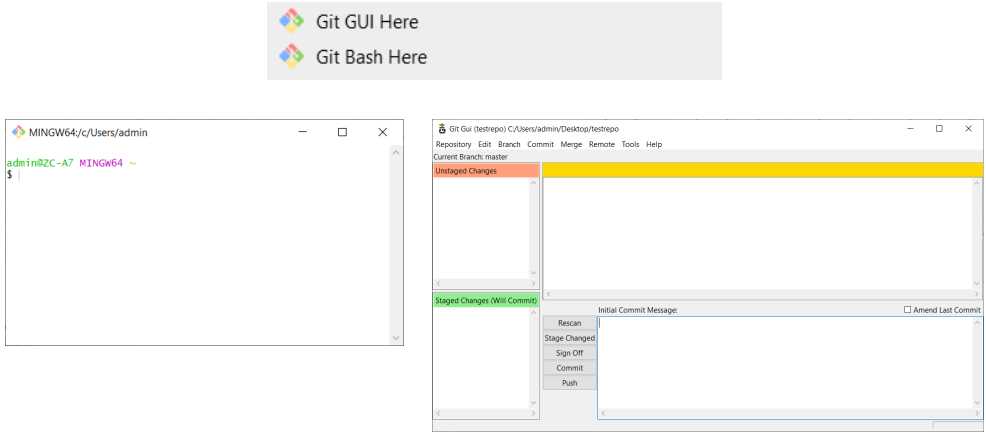
Nakon instalacije dobija se:

Git Bash – BASH emulator za git funkcije preko komandne linije. Ovaj način rada biće najviše korišćen u nastavku. Može se pokrenuti samostalno preko **Start** dugmeta ili iz konteksnog menija u **File Explorer** programu.

Druga alatka koja se dobija je **Git GUI**. Ovom alatkom se dobija vizuelno okruženje preko koga je moguće pratiti sva dešavanja i pokretati komande.

Napomena: Postoje mnogi popularni programi i alati koji pružaju podršku u radu sa Git-om, naročito zbog grafičkog UI, kao na primer *GitKraken* ili *Soursetree*. U Windows okruženju puno se koristi i *TortoiseGit*. Mi se ovde

zadržavamo na programima koji nude punu administratorsku podršku odnosno celokupnu Git sintaksu.



Slika 3.1. **Git Bash** i **Git GUI** nakon instalacije

Git je lokalizovan na veliki broj jezika, pa i za srpski. Svu dokumentaciju kao i uputstvo za instalaciju možete čitati na srpskom jeziku na **https://git-scm.com/book/sr/v2/Početak-Instaliranje-Gita**. Na ovom mestu pogledajte postupak instalacije za druge operativne sisteme.

Pomoć

Pre nego što praktično počnemo sa radom primenjujući i objašnjavajući razne komande koje možete koristiti, treba da znate prvu:

`git help` [*komanda*]

Ova komanda daje pomoć u vidu objašnjena i sintakse. Ukoliko je poznat naziv komande, ona se navodi na kraju. Na primer: **`git help status`**, **`git help commit`**.

Bez navedene komande, koja je opciona, dobijate kratko objašnjenje i listu mogućih komandi. Na primer:

```
$ git help
```

```
usage: git [--version] [--help] [-C <path>] [-c <name>=<value>]
           [--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]
           [-p | --paginate | -P | --no-pager] [--no-replace-objects] [--bare]
           [--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]
           <command> [<args>]
These are common Git commands used in various situations:
start a working area (see also: git help tutorial)
   clone      Clone a repository into a new directory
   init       Create an empty Git repository or reinitialize an existing one
work on the current change (see also: git help everyday)
   add        Add file contents to the index
   mv         Move or rename a file, a directory, or a symlink
   reset      Reset current HEAD to the specified state
   rm         Remove files from the working tree and from the index
examine the history and state (see also: git help revisions)
   bisect     Use binary search to find the commit that introduced a bug
   grep       Print lines matching a pattern
   log        Show commit logs
   show       Show various types of objects
   status     Show the working tree status
grow, mark and tweak your common history
   branch     List, create, or delete branches
   checkout   Switch branches or restore working tree files
   commit     Record changes to the repository
   diff       Show changes between commits, commit and working tree, etc
   merge      Join two or more development histories together
   rebase     Reapply commits on top of another base tip
   tag        Create, list, delete or verify a tag object signed with GPG
collaborate (see also: git help workflows)
   fetch      Download objects and refs from another repository
   pull       Fetch from and integrate with another repository or a local
branch
   push       Update remote refs along with associated objects
'git help -a' and 'git help -g' list available subcommands and some
concept guides. See 'git help <command>' or 'git help <concept>'
to read about a specific subcommand or concept.
```

Kao što se vidi, Git je jednostavan, tj. sadrži mali broj komandi. Jednostavnost i efikasnost čini ovaj alat popularnim. Naravno, osim upotrebe komandi moguće je korišćenje i specijalizovanih alata koji još više mogu olakšati upotrebu Git-a.

Konfigurisanje

Git poseduje mogućnost za podešavanje tj. konfigurisanje. Konfigurisanje se vrši postavljanjem određenih vrednosti za specifične parametre Git-a. Ovi parametri se koriste tokom rada. Osnovna komanda je:

```
git config [parametri]
```

Postoje 3 različita nivoa konfiguracionih parametara. To su:

- Sistemski;
- Globalni;
- Lokalni.

Vrednosti parametara jednog nivoa preklapa vrednosti iz prethodnog nivoa.

Sistemski/globalni/lokalni nivo

Osnovne karakteristike različitih nivoa kofiguracija su:

- Sistemski parametri važi za svakog korisnika na sistemu i za sve repozitorijume;
- Globalni se koristi za sve repozitorijume, ali jednog korisnika;
- Lokalni nivo se koristi za jedan repozitorijum.

Da bi se pregledali konfiguracioni parametri na određenom nivou koristi se komanda:

```
git config --list [--system][--global][--local]
```

Slično, može se pokrenuti podrazumevani editor za izmenu konfiguracionog fajla za određeni nivo.

```
git config --edit [--system][--global][--local]
```

Napomene: Ako se ne navede nivo onda se prikazuju parametri za sva tri nivoa istovremeno. Za izmene konfiguracionog fajla, neophodno je da postoji već podešen program za uređivanje sadržaja fajla - editor.

Slede tri primera koji ilustruju postavljanje specifičnog parametra određenog nivoa:

```
git config --system color.ui true
git config --global user.name pera
git config --local core.ignorecase true
```

Ako ne postoji postavljen editor u konfiguraciji ili se želi promeniti postojeći, tako da *notepad* bude postavljen, primenjuje se sledeća komanda:

```
$ git config --global core.editor notepad
```

Za promene na konfiguracionim fajlovoma dodaje se fleg `--edit`. Ova vrsta izmena omogućava korisniku uvid i u druga podešavanja:

```
git config --system --edit
```

.gitignore

U realnim slučajevima kada se Git koristi obično se radi verzioniranje velikog broja fajlova u okviru jednog ili više projekata. Pri tome veliki broj fajlova se generiše radom nekog programa, na primer kompajlera, ili korišćenjem određenog razvojnog - IDE okruženja (.class, .pyc, .o). Takvi fajlovi se obično ne koriste pri formiranju nove Git verzije. Oni se kreiraju automatizovano i usled njihovog poređenja nastaće nepotrebne verzije

ali i problemi u praćenju realnih izmena. Zato je važno da objasnimo mehanizam koji Git koristi u ovom slučaju.

Da bi se uradilo odvajanje fajlova, koji se postavljaju na repozitorijum od onih drugih, i to automatizovano a ne ručno, Git predviđa upotrebu fajla .**gitignore**, čiji sadržaj predstavljaju ekstenzije fajlova koje Git treba da ignoriše pri kreiranju verzija.

Na primer, za jedan projekata u Visual Studio IDE možete kreirati fajl .gitignore sledećeg sadržaja:

```
#Visual Studio files

*.[Oo]bj
*.user
*.aps
*.pch
*.vspscc
*.vssscc
*_i.c
*_p.c
*.ncb
*.suo
*.tlb
*.tlh
*.bak
*.[Cc]ache
*.ilk
*.log
*.lib
*.sbr
*.sdf
*.pyc
*.xml
ipch/
obj/
[Bb]in
[Dd]ebug*/
[Rr]elease*/
Ankh.NoLoad
```

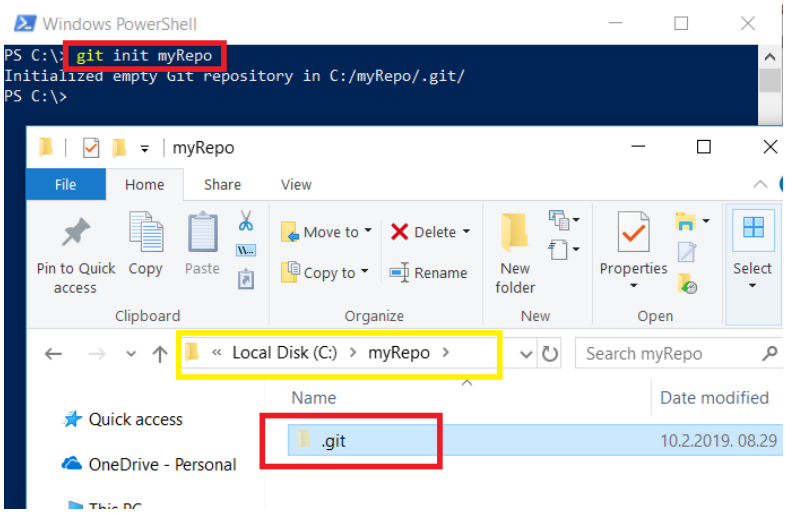
Kreiranje repozitorijuma

Komanda za kreiranje lokalnog repozitorijuma je:

`git init` *`repo`*

Ovom komandom se kreira direktorijum koji će biti repozitorijum. Ako se direktorijum ne navede repozitorijum se kreira u tekućem direktorijumu. Ovaj direktorijum / folder naziva se **radni** (eng. *working directory*).

Kreiranje repozitorijuma prouzrokuje kreiranje skrivenog foldera **`.git`** u repozitorijumu. Ovaj folder sadrži sve podatke koji se tiču promena na repozitorijumu. Brisanje ovog foldera znači brisanje celokupne istorije, odnosno brisanje svih podataka o repozitorijumu. Pogledajte primer:



Slika 3.2. Kreiranje prvog repozitorijuma

Na kraju, recimo i to da se lokalni repozitorijum može kreirati i kao klon udaljenog repozitorijuma i koristiti za lokalni razvoj uz povremenu sinhronizaciju sa udaljenim. Više o ovome u delu gde ćemo se baviti udaljenim tj. mrežnim repozitorijumima.

Status

Status repozitorijuma predstavlja stanje u repozitorijumu koje obuhvata podatke o svim fajlovima koji se prate (radni fajlovi) kao i podatke o fajlovima koji su već pripremljeni za snimanje sledeće verzije (indeksirani fajlovi). Komanda je:

`git status`

Na primer:

1. Ako nema fajlova za snimanje:

```
$ git status
```

```
On branch master
nothing to commit, working tree clean
```

2. Ako postoje brisanje/dodavanje/izmena fajlova:

```
$ git status
```

```
On branch master
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working
directory)

        modified:   dokument3.txt
        deleted:    dokument4.txt

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        New Text Document.txt
        dokument5.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

Kao što se vidi, ova komanda daje prikaz promena u odnosu na poslednju verziju dokumenata koja je snimljena (o snimanju nešto kasnije). Vide se fajlovi koji su u međuvremenu izbrisani, takođe i fajlovi koji su potpuno novi i za koje ne postoji praćenje, kao i promene na onim fajlovima za koje već postoji praćenje. Da bi promene bile deo buduće verzije promenjeni fajlovi se moraju dodati u indeks. Nekada se kaže da se ti fajlovi moraju postaviti na scenu.

Sada pogledajmo kako se vrši odavanje jednog fajla na scenu tj. indeks.

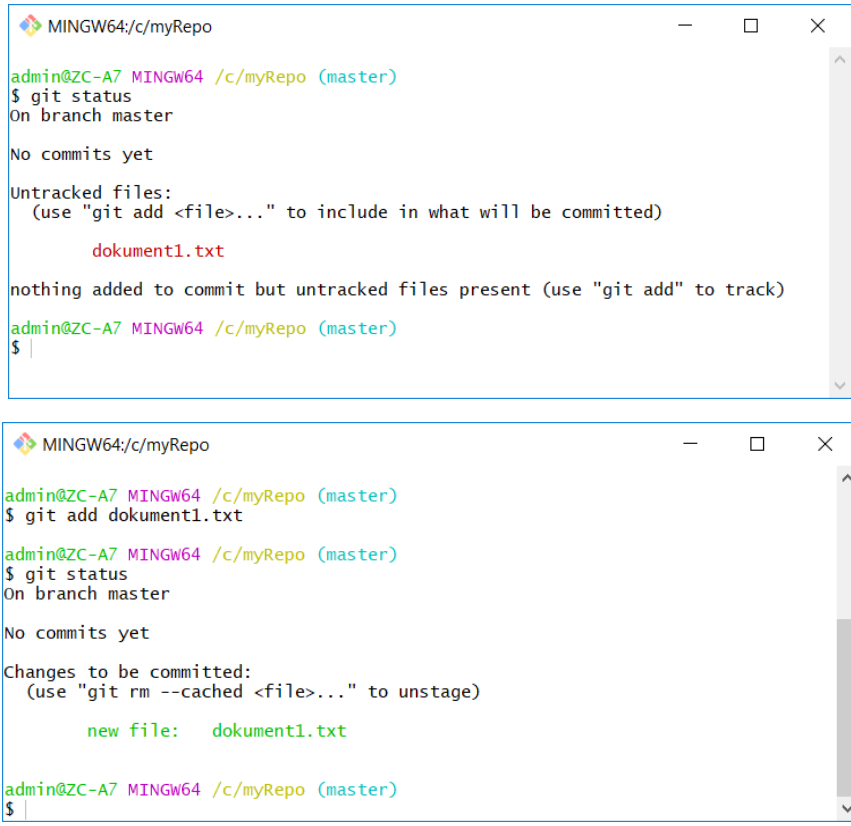
Dodavanje fajla

Ako je neki fajl u folderu koji predstavlja repozitorijum, to ne znači automatski da je fajl u repozitorijumu, odnosno ne znači da će postojati praćenje izmena u tom fajlu. Da bi Git čuvao verzije ovog fajla tj. pratio

promene na tom fajlu najpre taj fajl treba uključiti u praćenje tj. kaže se da je potrebno postaviti ga na **scenu** (eng. *stage*) koju obuhvata Git. Za fajlove koji su na sceni kaže se da su **indeksirani** (eng. *index*). Ovo se postiže komandom:

`git add imeFajla`

Pogledajmo prikaz Git statusa pre i posle dodavanja fajla *dokument1.txt*



Slika 3.3. Status pre i posle dodavanja jednog fajla

Skup fajlova koji Git prati naziva se **scena** ili indeks tj. a zbog dokumentacije, često se koriste i engleski termini ***index*** ili stejdžing (eng. *Staging area*). Obratite pažnju da se poruke odnose na granu **master**. Više

o grananju kasnije, za sada recimo samo da je ovo osnovna tj. glavna grana repozitorijuma.

Neke varijante ove komande su:

`git add`

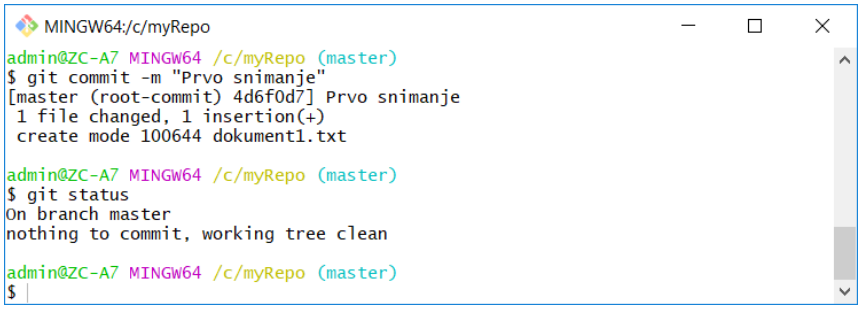
`--all(-A)` -dodavanje svih fajlova u folderu,
`--force(-f)` -dodavanje fajlova koji su definisani za ignorisanje,
`--dry-run(-n)` -pokazuje sve poruke ali bez dodavanja fajlova,
`--interactive(-i)` – uključena je interaktivnost sa korisnikom.

Snimanje nove verzije

Snimanje promena u repozitorijum, tj. formiranje nove verzije, obavlja se komandom commit na sledeći način:

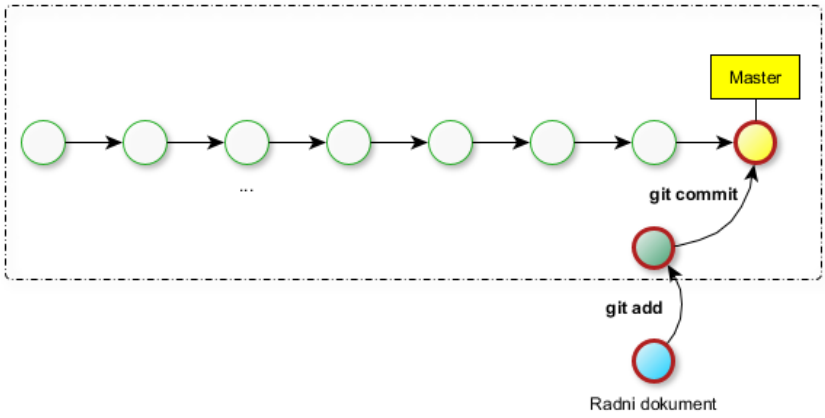
`git commit -m "poruka"`

Uz snimanje obavezno se navodi poruka koja se kasnije koristi u prikazu istorije. Poruka je obeležje verzije i treba je birati tako da daje neki podatak. Novi status bio bi:



Slika 3.4. Snimanje promena u repozitorijumu

Postupak dodavanja fajla u repozitorijum kao i snimanje promene na repozitorijumu može se prikazati na slici 3.5.



Slika 3.5. Grafički prikaz dodavanja i snimanja novog fajla

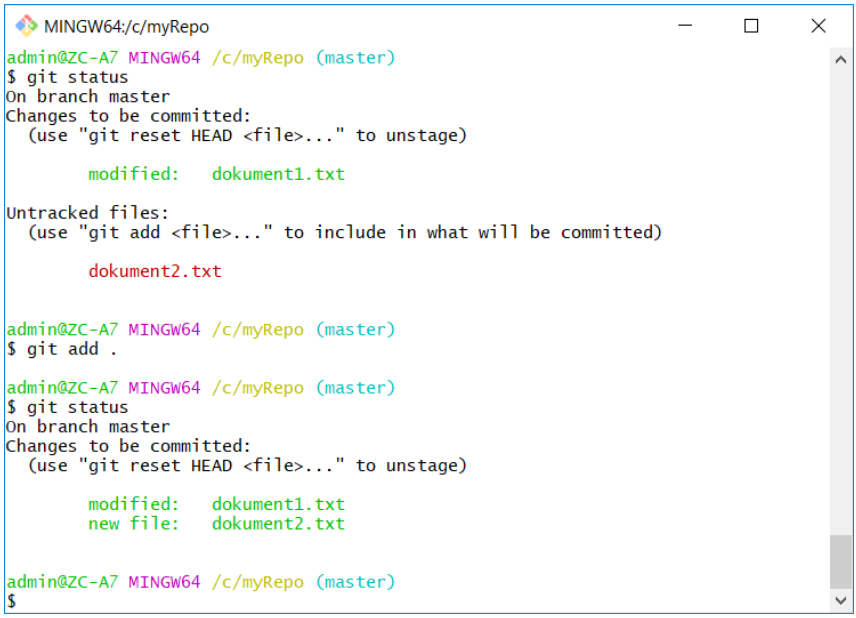
Važno. Naredba `commit` snima sve promene na fajlovima koji su indeksirani. Dakle, samo novi ili promenjeni fajlovi koji su stejdžovani tj. na sceni tj. indeksirani, primenom komande **`git add`**, biće deo buduće verzije.

Ako se u naredbi `commit` navede eksplicitno naziv fajla onda se taj fajl istovremeno indeksira i snima (kaže se *komituje* od eng. *commit*) u repozitorijum.

Moguće je izvesti delimično snimanje izmena. Ovo se izvodi eksplicitnim navođenjem fajlova koji treba snimiti. Češći slučaj je indeksiranje veće grupe fajlova. Ovo se postiže primenom specijalnih karaktera * ili . umesto eksplicitnog naziva fajla. Na primer:

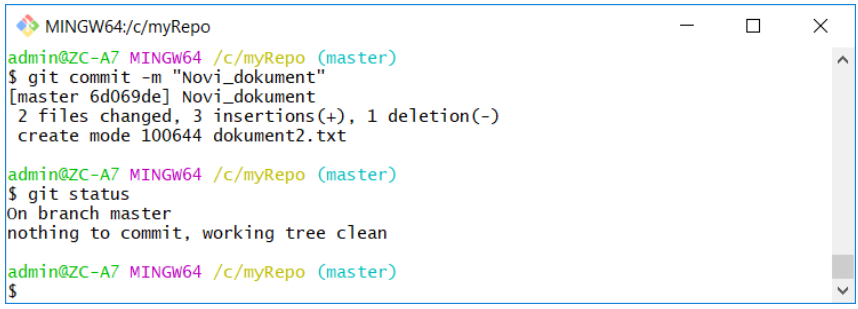
```
git add .
```

Ova komanda dodaje sav sadržaj tekućeg foldera u repozitorijum. Pod sadržajem se podrazumevaju fajlovi u ovom folderu kao i fajlovi u svim podfolderima.



Slika 3.6. Dodavanje svih dokumenata jednog foldera

Zatim se može izvesti novo snimanje.



Slika 3.7. Snimanje svih izmena

Tajno skladištenje

Tokom promena na nekom fajlu, po nekada se moramo prebaciti u drugu granu, a pri tome tekuće izmene nismo snimili. Takvu promenu Git neće dopustiti zbog tekućih promena na radnom direktorijumu. Zato se tekuće promene ostavljaju „po strani“. Ovo znači primenu posebne komande

koja će uraditi snimanje tekućeg stanja radnog direktorijuma - `git stach` ili `git stash push` pomoću koje je moguće privremeno sačuvati izmene. Kada se kasnije vratite na prvobitnu granu, prethodne izmene mogu se vratiti unazad. Na primer:

```
$ git status
```

```
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
        new file:   dokumen7.txt
        modified:   dokument1.txt
```

```
$ git stash
```

```
Saved working directory and index state WIP on master: 3d75a7a v-
1.6
admin@ZC-A7 MINGW64 /c/myRepo (master)
```

```
$ git status
```

```
On branch master
nothing to commit, working tree clean
```

Sada pogledajmo listu onoga što smo privremeno sačuvali.

```
 $ git stash list
stash@{0}: WIP on master: 3d75a7a v-1.6
```

Naravno, u jednom trenutku, potrebno je podatke vratiti na početne. Ovo se postiže primenom komande: `git stash apply [stash@{x}]`

```
$  git stash apply
```

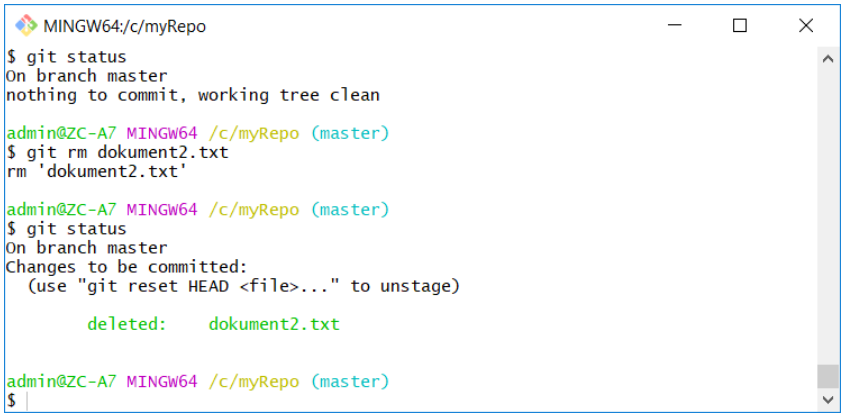
```
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)2
        new file:   dokumen7.txt
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working
directory)
        modified:   dokument1.txt
```

Brisanje

Brisanja fajlova spada u rizične operacije. Git nudi posebne opcije za brisanje iz repozitorijuma, a posebne da se fajl obriše i u radnom direktorijumu. Komanda kojom se briše neki fajl je:

`git rm [--cached] file`

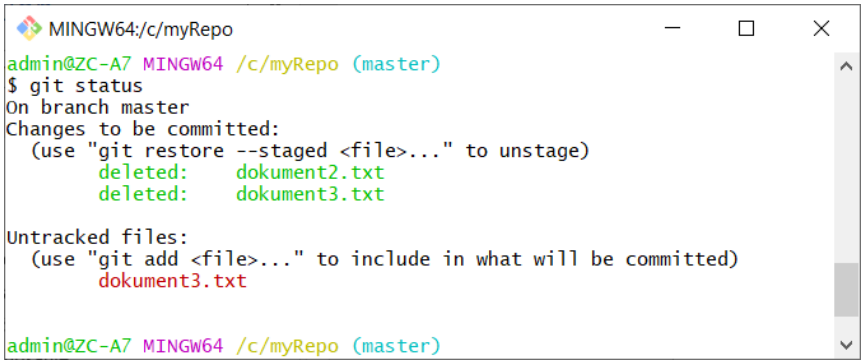
Pogledajmo primer primene ove komande sa pratećim statusima.



Slika 3.8. Brisanje fajla dokument2.txt iz repozitorijuma

Komanda brisanja **uklanja fajl iz radnog direktorijuma i istovremeno sa scene**. Izbrisani fajl neće biti deo budućih verzija. Naravno, moguće je uraditi vraćanje unazad i na taj način povratiti obrisani fajl.

Ova komanda ima više opcija. Primenom opcije **--cached** fajl se izbacuje sa scene tj. ostavlja trag kao da je obrisan i na dalje neće biti deo sledećih snimanja, ali fajl neće biti zaista obrisan iz radnog foldera. Pogledajmo šta se događa



Slika 3.9. Brisanje fajla dokument3.txt iz repozitorijuma opcijom --cached

Čišćenje radnog prostora

Git poseduje posebnu komandu za brisanje fajlova koje nisu deo scene. Na primer postavljen je novi fajl u radni folder, ali nije dodat na scenu za novu verziju (git add). To su oni fajlovi označeni crvenom bojom kada se vrši prikaz statusa. Ova komanda je neka vrsta „čišćenja“ foldera repozitorijuma.

`git clean [-n][-i][-q][-x][-X][-d][-f]<path>`

Brisanje se obavlja rekurzivno na svim fajlovima koji nisu uključeni u reviziju, počev od tekućeg foldera. Pošto Git može da ograniči rad odvajajući vrste fajlova sa kojima ne radi, moguće je preko opcije –x isključiti takvo ponašanje tj. obezbediti brisanje na svim fajlovima. Ovo je od koristi ako se inače isključi praćenje fajlova koje generiše sistem, a pri brisanju se želi takve fajlove izbrisati. Pogledajmo značenje gore navedenih opcija ove komande.

Ako se navede `<path>` putanja onda se akcija primenjuje samo na te fajlove.

`-n` (--dry-run)

Samo pokazuje koji fajlovi/folderi će biti uklonjeni bez stvarnog uklanjanja. Zbog osetljivosti ove komande, obično je važno proveriti šta će zaista biti obrisano.

`-i` (--interactive)

Interaktivno pokazuje šta će biti obrisano. Vrlo korisno.

`-q` (--quiet)

Tiho izvršavanje. Prikazaće samo greške ako postoje, ali i ne fajlove koje obriše.

`-x` (malo slovo x)

Ignoriše pravila navedena u fajlu .gitignore (po folderu) odnosno $GIT_DIR/info/exclude. Omogućava brisanje svih fajlova koji nisu indeksirani.

`-X` (veliko slovo X)

Uklanja samo fajlove koje Git ignoriše. Ovo može biti korisno za *rebuild*.

`-d`

Ova komanda omogućava rekurzivno prolaženje kroz direktorijume. Dakle, uklanja foldere koji se ne prate kao i fajlove. Ukoliko je neki folder deo nekog drugog repozitorijuma onda se ipak ne briše.

`-f` (--force)

Ukoliko je Git konfigurisan tako da je promenljiva clean.requireForce postavljena na true, onda se brisanje neće izvršiti bez da eksplicitno navedete ovu opciju.

Pogledajmo sledeći primer:

```
$ git status
```

```
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        dokument2_BACKUP_13472.txt
        dokument2_BASE_13472.txt
        dokument2_LOCAL_13472.txt
        dokument2_REMOTE_13472.txt
        sh.exe.stackdump
nothing added to commit but untracked files present (use "git add"
to track)
```

```
$ git clean -n
```

```
Would remove dokument2_BACKUP_13472.txt
Would remove dokument2_BASE_13472.txt
Would remove dokument2_LOCAL_13472.txt
Would remove dokument2_REMOTE_13472.txt
Would remove sh.exe.stackdump
```

```
$ git clean -f
```

```
Removing dokument2_BACKUP_13472.txt
Removing dokument2_BASE_13472.txt
Removing dokument2_LOCAL_13472.txt
Removing dokument2_REMOTE_13472.txt
Removing sh.exe.stackdump
```

```
$ git status
```

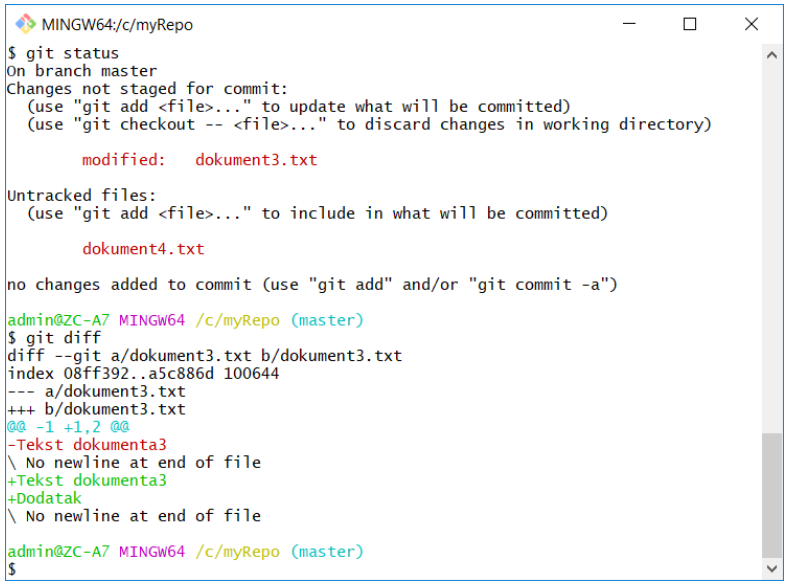
```
On branch master
nothing to commit, working tree clean
```

Razlike

Git poseduje komande za prikaz razlike dve verzije. Standardna komanda je:

`git diff`

Ovom komadom prikazuje se razlike radnih kopija svih indeksiranih fajlova. Razlika se odnosi na promenjene fajlove u odnosu na poslednje snimanje tzv. komit.



Slika 3.10. Prikaz komande: **`git diff`**

Obratite pažnju da ova komanda ne obuhvata fajlove koji nisu indeksirani bez obzira što pripadaju istom folderu.

Radni fajlovi se najpre postavljaju na scenu/indeks, a zatim se vrši snimanje/komit. Prebacivanjem na scenu, razlike se mogu tražiti sada između fajlova na sceni i poslednjeg snimanja. Opcija **`--staged`** odnosi se na razlike u odnosu na fajlove koji su na scenu.

Primenom komande

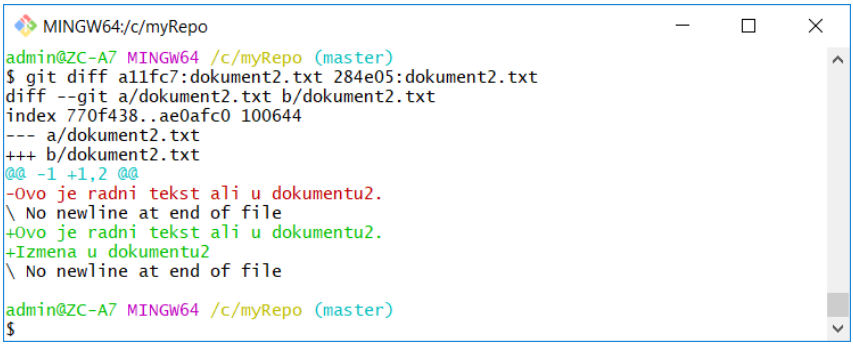
`git diff --staged` *`r1`*

prikazuju se razlike tekućih fajlova na sceni i neke revizije r1. (Pojam revizija se koristi povremeno kao zamena za verziju.) Ako se ne navede revizija onda se prikazuje razlika fajlova koji su na sceni od poslednjeg snimanja, na primer.

`git diff --staged HEAD^` (kasnije ćemo objasniti oznaku HEAD)

Komanda se može koristiti i za prikaz razlika između dve verzije. U tom slučaju komanda izgleda ovako:

`git diff r1:file1 r2:file2`



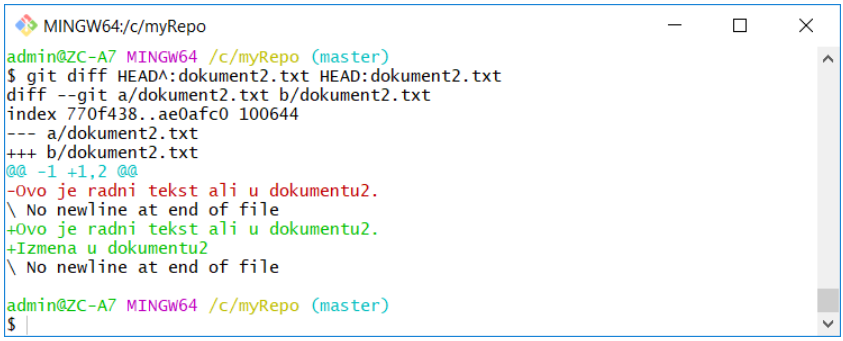
Slika 3.11. Prikaz razlika: git diff a11fc7:dokument2.txt 284e05:dokument2.txt

Napomena: U primeru su za oznake revizija korišćene vrednosti a11fc7 odnosno 284e05 koje predstavljaju jedinstvene oznake sačuvanih revizija. Ove vrednosti se mogu dobiti primenom **`git log`** komade koju ćemo kasnije opisati detaljno. Moguće je pogledati listu svih dosadašnjih revizija:

`git rev-list --all`

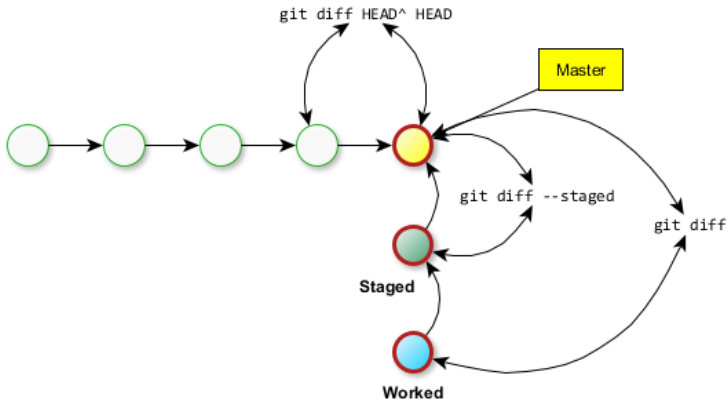
Sada pogledajmo još jedan način upoređivanja postojećih revizija.

Imajući u vidu da je **HEAD** univerzalna oznaka za tekuću verziju, onda se ova oznaka zajedno sa prefiksom ^ može koristiti za oznaku revizije. Tako gornji se primer može drugačije napisati kao:



Slika 3.12. Prikaz razlika: git diff HEAD^:dokument2.txt HEAD:dokument2.txt

Grafički prikaz nekih od prethodno objašnjenih komadi dat je na narednoj slici.



Slika 3.13. Prikaz više različitih diff komandi

Referenca HEAD

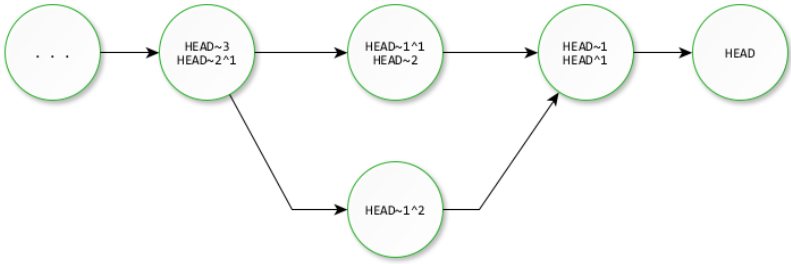
U radu sa Git-om često se koriste termini odnosno oznaka HEAD.

HEAD: Ukazuje na poslednji komit tj. važeće snimanje na repozitorijum. U standardnim operacijama, HEAD ukazuje na poslednji komit u trenutnoj

grani, ali to ne mora biti slučaj. HEAD znači referencu na ono što je trenutni repozitorijum.

Ukoliko referenca HEAD ukazuje na snimanje koje nije vrh grane tj. nije poslednje u grani onda se to naziva "odvojena glava" (eng. *detached head*).

Postoji niz skraćenih oznaka za pozicioniranje revizije počev pod poslednje. Ove oznake prikazane su u primeru na slici:



Slika 3.14. Prikaz oznaka revizija baziranih na HEAD referenci

Značenje simbola je sledeće:

~ označava prethodnu verziju u tekućoj grani,

^ označava roditeljsku verziju. Ako ima više roditeljskih verzija, onda se ova oznaka mora koristiti da bi označili određenog roditelja.

Tako na primer:

A~ je uvek isti kao A^

A~~ je uvek isti kao A^^, i tako dalje.

A~2 nije isto što i A^2. Oznaka ~2 je skraćenica za ~~ dok ^2 označava drugog roditelja po redu, naravno samo ako takav postoji, pogledajte prethodnu sliku.

Istorija promena

Jedna od osnovnih karakteristika VCS sistem je mogućnost uvida u sve izmene tj. istoriju repozitorijuma. Komande koje prikazuju istoriju snimanja su:

`git log` `[--grap=regExp][-p]`

`git whatchanged` `[--grap=regExp]`

Komanda `whatchanged` se razlikuje u odnosu na `log` komandu u tome što druga komanda uz prikaz verzija daje i kratak prikaz promena u svakoj od njih. Na primer:

`git log`

```
commit 284e05d50978f23ee7357976befef27434b64ed9

Author: zoran <zoran.cirovic@gmail.com>

Date:   Sun Feb 10 18:45:04 2019 +0100

    v-1.2

    Izm: dokument2

    Nov: dokument3


commit a11fc79d92cc8a98f7cf859db23a3489098768b0

Author: zoran <zoran.cirovic@gmail.com>

Date:   Sun Feb 10 10:16:05 2019 +0100

    v-1.1

    Nov: dokument2

    Izm: dokument1
```

`git whatchanged`

```
commit 284e05d50978f23ee7357976befef27434b64ed9

Author: zoran <zoran.cirovic@gmail.com>

Date:   Sun Feb 10 18:45:04 2019 +0100

    v-1.2

    Izm: dokument2

    Nov: dokument3

:100644 100644 770f438 ae0afc0 M       dokument2.txt

:000000 100644 0000000 08ff392 A       dokument3.txt


commit a11fc79d92cc8a98f7cf859db23a3489098768b0

Author: zoran <zoran.cirovic@gmail.com>

Date:   Sun Feb 10 10:16:05 2019 +0100
```

```
commit 4d6f0d78c030c2caccb5ad2cb0a004d42276767c

Author: zoran <zoran.cirovic@gmail.com>

Date:   Sun Feb 10 09:21:44 2019 +0100

    Prvo snimanje
```

```
    v-1.1

    Nov: dokument2

    Izm: dokument1

:100644 100644 760889e bd049fb M        dokument1.txt

:000000 100644 0000000 770f438 A        dokument2.txt


commit 4d6f0d78c030c2caccb5ad2cb0a004d42276767c

Author: zoran <zoran.cirovic@gmail.com>

Date:   Sun Feb 10 09:21:44 2019 +0100

    Prvo snimanje
```

Očigledno je da istorija omogućava da pronađemo izmene koje su urađene tokom razvoja projekta. U slučaju velikog broja revizija, od značaja je imati dodatni način pretrage željenih podataka. Zato obe komande mogu imati parametar za pretragu.

Opcija za pretragu ostavljenih poruka uz svaku reviziju je `--grep`. Vrednost koja se pridružuje ovoj opciji je regularni izraz, na primer:

```
$ git whatchanged --grep=$1.6
commit 3d75a7a0851b9a3663a2d79d9b37192e2db9946c (HEAD -> master, tag: ignore)
Author: zoran <zoran.cirovic@gmail.com>
Date:   Sat Feb 16 14:27:22 2019 +0100
    v-1.6
:000000 100644 0000000 36045b4 A        .gitignore
:000000 100644 0000000 a31b7e9 A        dokument5.txt
```

Drugi opcija pretrage namenjena je u slučaju kada se radi pretraga snimljenih verzija po nekoj vrednosti stringa ali u nekom fajlu (eng. *pickaxe option*). Sintaksa je:

```
git log S"string koji se pretražuje"
```

Na primer:

```
$ git log -STekst
commit 7f6fde2b899f92e9924fc5543af6062138a35d08 (tag: grananje)
Author: zoran <zoran.cirovic@gmail.com>
Date:   Mon Feb 11 19:20:55 2019 +0100
    v-1.3

commit 284e05d50978f23ee7357976befef27434b64ed9
Author: zoran <zoran.cirovic@gmail.com>
Date:   Sun Feb 10 18:45:04 2019 +0100
    v-1.2
    Izm: dokument2
    Nov: dokument3
```

Zapazite da ako string koji se pretražuje ima razmake onda ga moramo staviti između znaka navoda, u suprotnom ne.

Jedna često korišćenih opcija u praksi je: `-p (--patch)`, koja pokazuje razliku između svakog snimanja. Zakrpa je skup razlika između revizija kako bi se pokazalo šta se između njih razlikuje. Obično se generiše zakrpa samo da se pokaže šta je promenjeno. Primer kada se to može uraditi je kada pronađete i ispravite grešku u aplikaciji, a zatim objavite ispravku.

Zbog mogućeg velikog broja verzija, postoji opcija koja će prikazati određen broj komita u pretrazi.

```
$ git log -p -1 [>imefajla]
```

```
commit 8f2da4dc6d656a2b896feb1eef310b24f4e871d3 (HEAD -> master)
Author: zoran <zoran.cirovic@gmail.com>
Date:   Sun Feb 17 21:35:09 2019 +0100
    v-1.8
diff --git a/dokument2.txt b/dokument2.txt
index 10d5de9..fc06fab 100644
--- a/dokument2.txt
+++ b/dokument2.txt
@@ -1 +1 @@
-glavne izmene...jos neke izmene
\ No newline at end of file
+glavne izmene
\ No newline at end of file
admin@ZC-A7 MINGW64 /c/myRepo (master)
```

Postoji veliki broj opcija koje mogu biti od interesa. Pogledajmo neke od njih:

- `--since, --after` – filtrira komite počev od zadatog datuma,
- `--until,--before` – pre nekog datuma,
- `--author` – prikazuje komite od određenog autora.

Blame

Git nudi posebnu komandu samo za praćenje promena na određenoj datoteci. Sintaksa za primenu ove komande je:

`git blame` [*`fileName`*]

Rezultat je prikaz kada i u kom snimanju je napravljena izmena navedenom fajlu. Na primer:

```
$ git blame dokument1.txt
```

```
a11fc79d (zoran              2019-02-10 10:16:05 +0100 1) Ovo je
radni tekst.
00000000 (Not Committed Yet 2019-02-17 20:52:32 +0100 2) Ovo smo
izmenili.xx
admin@ZC-A7 MINGW64 /c/myRepo (master)
$ git blame dokument3.txt
7f6fde2b (zoran 2019-02-11 19:20:55 +0100 1) Tekst dokumenta3
7f6fde2b (zoran 2019-02-11 19:20:55 +0100 2) Dodatak
```

Tag

Svako snimanje pomoću Gita jedinstveno je označeno jednom SHA vrednošću. Na tu vrednost korisnik ne utiče. Ove vrednosni nisu čitljive pa ih je teže koristiti u radu. Umesto njih lakše je pratiti snimanja preko komentara koji se obavezno unose pri svakom komitu.

Međutim, komentar ne može da posluži kao jedinstvena oznaka odnosno za identifikaciju komita. **Tag** predstavlja čitljiv, jedinstven, korisnički definisan tekst za jedan komit. Moguće je pridružiti više tagova za jedan komit. Pogledajmo osnovnu sintaksu ove komande.

`git tag [naziv][-d naziv]`

Komanda **`git tag`** bez dodatnih argumenata daje listu tagova za tekući projekat. Lista može biti dodatno filtrirana primenom opcije –l uz string za pretragu, na primer: **`git tag -l "v1.*"`**.

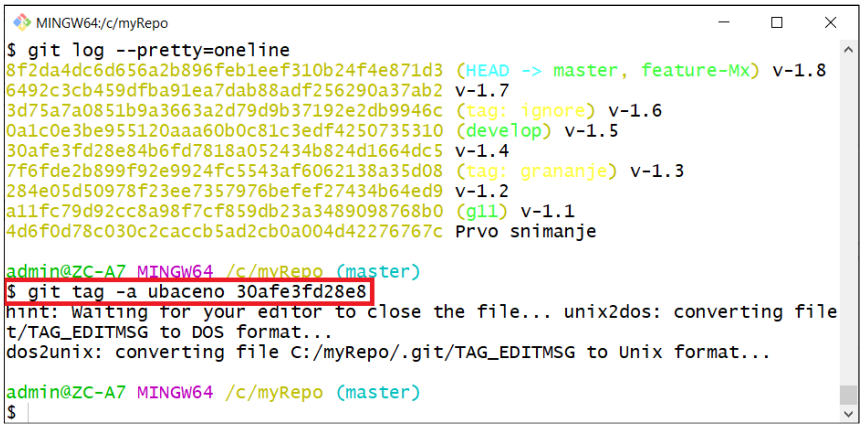
Preostale opcije uz ovu komadu su:

- `naziv` - dodaje se novi tag na poslednji komit. Naziv mora biti jedinstveni string, na primer: git tag oznaka-1.
- `-d naziv` - briše se postojeći tag naziv.

Naknadno tagovanje

Moguće je izvesti i naknadno tagovanje nekih revizija. Za ovu akciju potrebno je identifikovati željenu reviziju. Jedan način da sažeto prikažete

istoriju komita, tagova i pripadajućih SHA identifikatora je korišćenjem opcije `--pretty` formatiranja, https://www.git-scm.com/docs/ pretty-options.



Slika 3.15. Naknadno ubacivanje novog taga

Reset

Mada postoji komanda rm koju smo već radili, Git poseduje specijalizovane komande za brisanje odnosno resetovanje repozitorijuma odnosno odgovarajućeg indeksa. Ovo se izvodi primenom komande:

```
git reset [--hard|soft|mixed]
```

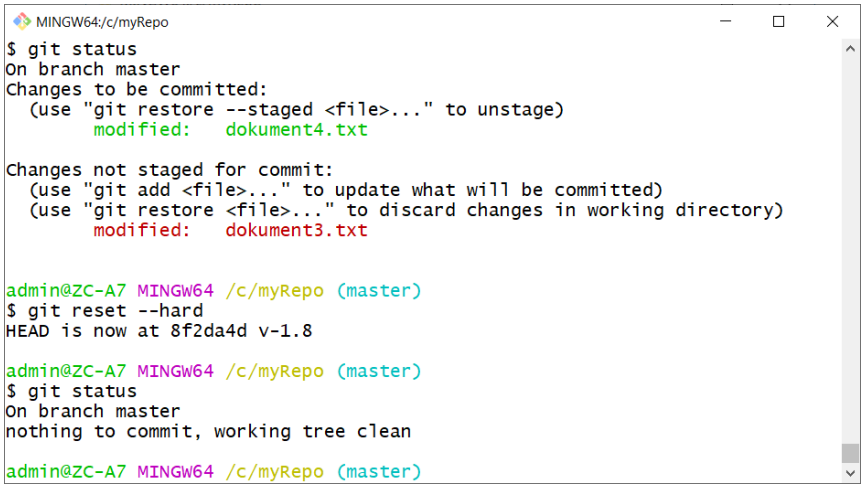
Postoje tri osnovna oblika ove komande. Ovi oblici se definišu argumentima: `--soft`, `--mixed`, `--hard`. Ova tri argumenta odgovaraju redom internim stanjima Git mehanizmima.

Tipovi reseta

Podrazumevano izvršavanje komande **`git reset`** praćeno je implicitnim argumentima **`--mixed`** i **HEAD**. Umesto reference HEAD može se koristiti bilo koja Git SHA-1 heš vrednost odnosno referenca na određeni komit.

--hard

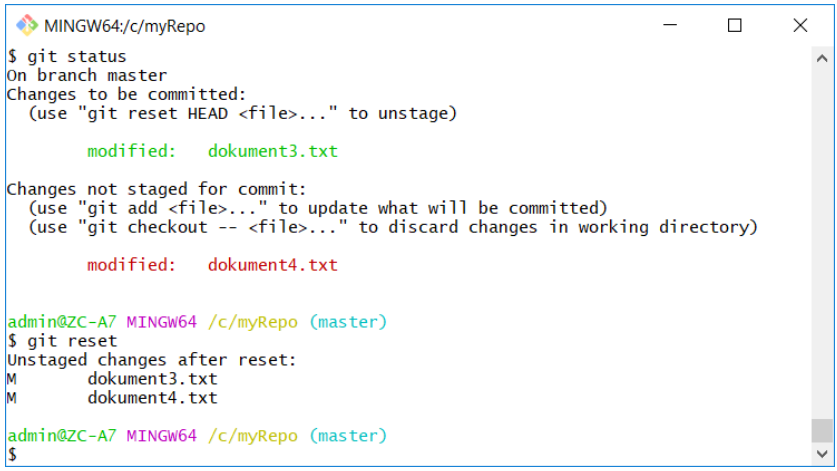
Ovo je najdirektnija i istovremeno najopasnija opcija, ali i često korišćena. Kada se uradi **`--hard`** komanda izvršava se komit, a zatim se sadržaj scene i radnih kopija resetuju tako da se poklapaju sa navedenim komitom. Sve promene koje postoje, takođe se resetuju da odgovaraju stanju poslednjeg komita. To znači da će svaki rad koji je bio na čekanju bilo na sceni ili radnom delu biti izgubljen.



Slika 3.16. Rezultat primene **`git reset --hard`**

--mixed

Ovo je podrazumevana opcija - mod. Referenca se ažurira. Promene na sceni se resetuju na stanje nekog specifičnog snimanja. Promene na sceni se prebacuju u radne promene.



Slika 3.17. Rezultat primene `git reset`

Ukoliko želimo da uklonimo baš određeni fajl, određene revizije, onda se ova komanda proširuje oznakom revizije kao i nazivom fajla:

```
git reset HEAD fileA fileB ...
```

--soft

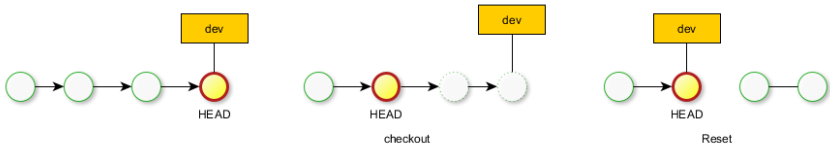
Primena ovog argumenta menjaju se samo reference. Sadržaj na sceni i u radnom folderu se ne menja.

Checkout

Prethodna komanda radi reset fajlova repozitorijuma na osnovu revizije na koju pokazuje HEAD referenca. Bitno drugačije radi komanda checkout.

`git checkout` *rev*

Ovom komandom vrši se poništavanje promena i preuzimanje neke prethodne revizije i njeno postavljanje na tekuću. Ovo se bolje vidi na primeru koji je prikazan:



Slika 3.18. Prikaz razlike primene checkout i reset komandi

Sumirano

U narednoj tabeli prikazaćemo sličnosti i razlike ovih komandi.

Tabela 3.1. Uporedni prikaz reset i checkout komande

	HEAD	Index	Radni dir.
Komit			
`reset --soft [commit]`	Ref	Ne	Ne
`reset [commit]`	Ref	Da	Ne
`reset --hard [commit]`	Ref	Da	Da
`checkout <commit>`	Head	Da	Da
File			
`reset [commit] <paths>`	Ne	Da	Ne
`checkout [commit] <paths>`	Ne	Da	Da

Revert

Svako novo snimanje utiče na istoriju verzija dodajući novi čvor sa svim podacima jedne verzije. Reset komanda može da promeni istoriju komita, a to može negativno da utiče na timski rad, odnosno reset nije dobra uvek opcija da bi se ispravila nega greška. Ukoliko greška postoji u poslednjem

komitu, a nije nam dozvoljeno da koristimo reset zbog takvih razloga, može se koristiti `revert` komanda. Sintaksa je:

`git revert` *`ref`*

Ova komanda dodaje novi komit na sam kraj te grane, koji je identičan referenci koja se navodi. Na primer:

```
$ git status
```

```
On branch master
nothing to commit, working tree clean
admin@ZC-A7 MINGW64 /c/myRepo (master)
$ git revert HEAD
hint: Waiting for your editor to close the file... 'C:/Program
Files
admin@ZC-A7 MINGW64 /c/myRepo (master)
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
        deleted:    dokumen7.txt
        modified:   dokument1.txt
```

Ispravka

U praksi se događa da se nakon snimanja zaključi da je neophodno nešto izmeniti, ali ne u novoj verziju već na postojećem komitu tj. verziji koja je već snimljena.

Neka je trenutno stanje:

```
$ git status
```

```
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
        new file:   dokument5.txt
```

Dakle imamo novi dokument na sceni koji hoćemo naknadno da snimimo u poslednji komit i naravno preklopimo poruku novom porukom. Komanda bi bila:

```
git commit --amend -m "v-1.6"
```

Za prethodni primer rezultat ove komande je:

```
$ git commit --amend -m "v-1.6"
[master 3d75a7a] v-1.6
 Date: Sat Feb 16 14:27:22 2019 +0100
 2 files changed, 47 insertions(+)
 create mode 100644 .gitignore
 create mode 100644 dokument5.txt
```

Pitanja i zadaci

1. Kreirajte novi repozitorijum
2. Kreirajte novi fajl. Dodajte ga u repozitorijum. Snimite.
3. Promenite fajl i uradite novo snimanje.
4. Proverite status.
5. Obrišite fajl i snimite.
6. Kreirajte dva nova fajla i snimite. Zatim modifikujte sadržaj a zatim pogledajte razliku radnih kopija.
7. Dodajte jedan fajl na scenu, a drugi neka ostane u radnom delu. Prikažite promene:
    a. Između fajla na sceni i radne kopije
    b. Poslednjeg snimljenog i indeksa
    c. Poslednjeg komita i radne kopije
8. Resetujte indeks.
9. Resetujte indeks i radne kopije.
10. Kako se proverava instalirana verzija Gita?
11. Koje vrste konfigurisanja postoje?
12. Kako se konfiguriše korisničko ime?
13. Koja je komanda za inicijalizaciju repozitorijuma?
14. Objasniti način dodavanja nove verzije.
15. Kako se prati istorija revizija?
16. Kako se identifikuje jedna revizija? Šta je tag i kako se koristi?
17. Tehnike brisanja fajlova u repozitorijumu.

18. Šta je *reset* koje vrste postoje?
19. Koje su razlike u vrstama *reseta*?
20. Objasniti *checkout* u odnosu na *reset*.
21. Kako se koristi i čemu služi *blame* komanda?

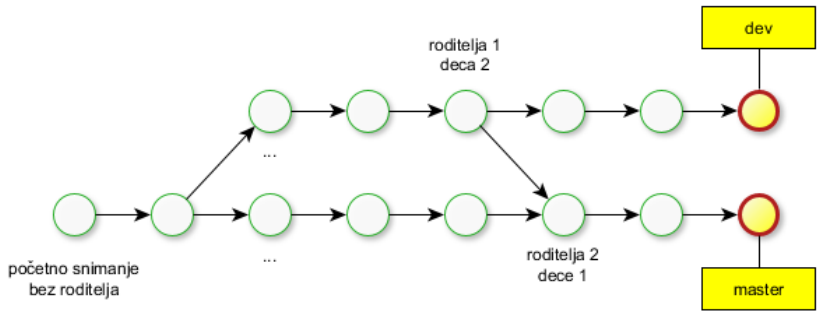
4. Grananje

Ovo poglavlje je u celosti posvećeno radu sa više verzija istovremeno, dakle grananju. Objašnjen je pojam grananja, način prelaza sa jedne na drugu granu, zatim preuzimanje sadržaja iz druge grane. Posebna pažnja posvećena je spajanju dve grane. Pošto se prilikom spajanja mogu dogoditi i konflikti opisuje se postupak razrešavanja konflikata. Na kraju objašnjeno je i brisanje grana.

Uvod

Svaka snimljena verzija dokumenata ima svoje mesto u istoriji revizija. Sve revizije su povezane kao objekti jedne liste. Samo početna revizija nema roditeljsku i samo poslednja nema decu objekte. Zapazite da revizija može imati više potomaka tj. dece pri grananju, odnosno dva roditelja pri spajanju.

Grananje nastaje kao odgovor na potrebu da se istovremeno radi na više verzija dokumenata.



Slika 4.1. Primer istorije koja sadrži grananje i spajanje grana

Na prethodnoj slici su prikazane dve grane: dev i master. Svaka od grana ima svoju oznaku i poslednje snimanje.

Pri komitu tj. snimanju jedne verzije formira se jedinstveni identifikator dužine 160 bita koristeći SHA-1 šifrovanje. Pri generisanju ovog identifikatora učestvuju:

- Fajlovi koji učestvuju u jednoj reviziji
- Prateći (meta) podaci (poruka pri snimanju, ime autora,...)
- Heš vrednost od roditeljskog komit-a.

Ovako dobijena vrednost obezbeđuje sigurnost i pouzdanost sadržaja revizije kao i njenu povezanost sa prethodnim revizijama.

Grananje

Grananje je postupak kojim se od jedne grane formira još jedna. Nova grana se kreira primenom komande:

```
git checkout -b naziv [start]
```

```
git branch naziv [start]
```

gde je:

naziv – naziv nove grane,

start – početna tačka tj. lokacija za novu granu. Može biti jedinstveni identifikator heš vrednost ili tag, ako postoji. Ako se ne navede koristi se tekuća lokacija.

Ovom komandama kreira sa nova grana, a istovremeno vrši se prebacivanje na novu kreiranu granu, tj. tekuća grana i referenca HEAD pokazuju na poslednji komit nove grane.

Na primer:

```
$ git checkout master
Switched to branch 'master'

$ git status
On branch master
nothing to commit, working tree clean

$ git checkout -b feature-Mx
Switched to a new branch 'feature-Mx'
```

Obratite pažnju na prateće poruke. Vidi se da se generiše nova grana i istovremeno se prelazi na tu novu granu. Uvek postoji samo jedna grana koja je trenutno aktivna tj. na kojoj se izvode operacije.

Kreiranje nove grane može da se izvede i primenom komande

git branch *naziv* [*start*]

Razlika u odnosu na prethodnu komandu je što se grana kreira, ali se ne vrši prebacivanje na novu granu.

Treba imati na umu da obično postoji veći broj grana. U nastavku dajemo listu nekoliko komandi koje mogu pomoći u sagledavanju svih grana.

`git branch` – lista lokalnih grana,

`git branch -r` – lista udaljenih grana,
`git branch --merged` - lista spojenih grana,
`git branch --no-merged` – lista nespojenih grana,
`git branch --contains [<commit>]` – lista grana koje sadrže određeni komit. Svaki put kada kreiramo novu granu ona kreće sa tekućim komitom grane sa koje kreće grananje.

Prelazi

Prelaz na određenu granu obavlja se primenom gotovo iste komande koja je korišćena pri kreiranju grane. Sintaksa je:

`git checkout` *`naziv`*

`naziv` – ime grane na koju se prelazi.

Ovu naredbu smo koristili i kada je bilo reči o resetu. Sada vidimo potpuno istu komandu koja omogućava prelaz na drugu granu, čisti scenu kao i prebacuje referencu na kraj nove grane.

```
$ git status
On branch feature-Mx
nothing to commit, working tree clean
admin@ZC-A7 MINGW64 /c/myRepo (feature-Mx)
$ git checkout dev
Switched to branch 'dev'
```

Preuzimanje fajlova

Naredba `git checkout` koristi se i za preuzimanje određenog fajla koji je na nekoj drugoj grani.

`git checkout nazivgrane -- file1 file2`

Na primer:

```
$ git status
```

```
On branch dev
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        aaa.xml
nothing added to commit but untracked files present (use "git add"
to track)
admin@ZC-A7 MINGW64 /c/myRepo (dev)
```

```
$ git checkout master -- dokument5.txt
```

```
admin@ZC-A7 MINGW64 /c/myRepo (dev)
```

```
$ git status
```

```
On branch dev
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
        new file:   dokument5.txt
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        aaa.xml
```

Spajanje grana

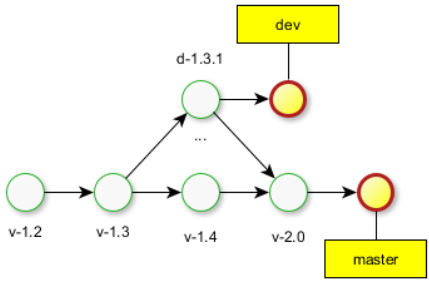
Neka u glavnoj grani postoje 4 fajla. Nakon grananja (git checkout –b dev) u novoj grani se nastavlja rad i neka se izvrše izmene na jednom fajlu (npr. dokument1.txt), a zatim se uradi novi komit. Istovremeno, u glavnoj grani, na drugom fajlu (dokument2.txt), izvrše se izmene i opet uradi komit. Zatim se radi spajanje te dve grane. Obično se vrši spajanje pomoćne grane sa glavnom. Dakle, ako smo na glavnoj grani, spajanje se postiže primenom komande:

```
git merge nazivGrane –m "message"
```

gde je:

`nazivGrane` - naziv grane sa koje se preuzimaju izmene pri spajanju, a

`-m "message"` - poruka koja se postavlja pri snimanju koje se obavlja automatizovano pri spajanju.




```
// v-1.3
git checkout dev
// rad na nekim dok.;
git commit –m "d - 1.3.1"
git checkout master
// rad na nekim dok.;
git commit –m "v - 1.4"
git merge dev –m "v - 2.0"
```

Slika 4.2. Grananje i spajanje

Konflikti

Konflikt nastaje ako se vrši izmena na istom fajlu u više grana koje se spajaju. Pri spajanju Git razdvaja dve varijante:

1. Tekstualni fajlovi. Spajaju se po linijama.
   - Ako je jedna linija promenjena samo u jednoj grani onda se može uraditi automatski spajanje.
   - Ako je ista linija promenjena u više grana onda Git prijavljuje konflikt. Zone konflikta su unutar oznaka <<<<<<< . . >>>>>>>. U ovom slučaju Git zahteva spajanje fajlova eksplicitnim učešćem korisnika.
2. Binarni fajlovi. Uvek se označava konflikt.

Rešavanje konflikata

Da bi Git uradio spajanje dve verzije potrebno je da se reše konflikti ukoliko oni postoje. Ako se neki fajlovi ne spoje ostaju u radnom delu i označeni su kao „unmerged“.

Ostali fajlovi kod kojih ne postoji konflikt, kao i preteći metapodaci se automatski dodaju u indeks (na scenu).

Postoje dva načina za rešavanje konflikata.

1. Ručno rešavanje. Ovo rešavanje podrazumeva editovanje fajlova, a zatim eksplicitno dodavanje/brisanje nekih od njih primenom komandi: `git add file` ili `git rm file`
2. Pokretanje nekog od alata za rešavanje konflikata tj. spajanje (xxdiff, kdiff3, beyond compare,...)

Pošto se konflikti fajlova u indeksu reše, vrši se snimanje nove verzije: `git commit`

Primer

Neka je sledeći redosled aktivnosti u repozitorijumu.

Korak 1. Repozitorijum inicijalizovan sa jednim dokumentom:
`git init, git add .`

Korak 2. Formiramo novu granu dev:
`git checkout –b dev`

Korak 3. Dodamo dva nova fajla i snimimo ih u novoj granu: dokument2.txt i dokument-Y.txt. Ovo možemo uraditi u dva koraka:
`git add .+ git commit –m “***“`

Korak 4. Prebacimo se u glavnu granu, dodamo dva nova fajla takođe, takođe u dva koraka, pri čemu se jedan od njih poklapa imenom sa jednim od fajlova u grani dev, ali sa svojim sadržajem.

Korak 5. Radimo spajanje. Dakle, u obe grane sve izmene su snimljene tj. Urađen je commit. Pokrećemo komandu:

```
$ git merge dev
Auto-merging dokument2.txt
CONFLICT (add/add): Merge conflict in dokument2.txt
Automatic merge failed; fix conflicts and then commit the
result.
```

Detektovan je konflikt u fajlu dokument2.txt.

Kako rešiti konflikt? Konflikt mora da se rešava učešćem tj. ručnim definisanjem šta će biti sačuvano u novoj verziji.

Najpre pogledajmo razliku verzija koje se spajaju. Ovo se postiže komandom:

git diff

Pogledajmo efekat primene ove komande, na slučaju dva dokumenta:

Master: dokument2.txt	dokument2 u glavnoj grani
dev: dokument2.txt	Neki tekst...

```
$ git diff
diff --cc dokument2.txt
index 3a88407,115b8f6..0000000
--- a/dokument2.txt

+++ b/dokument2.txt
@@@ -1,1 -1,1 +1,5 @@@
- dokument2 u glavnoj grani
 -Neki tekst...
++<<<<<<< HEAD
++dokument2 u glavnoj grani
++=======
++Neki tekst...
++>>>>>>> dev
```

Sada ćemo pokrenuti alat za spajanje dokumenata:

```
$ git mergetool dokument2.txt
```

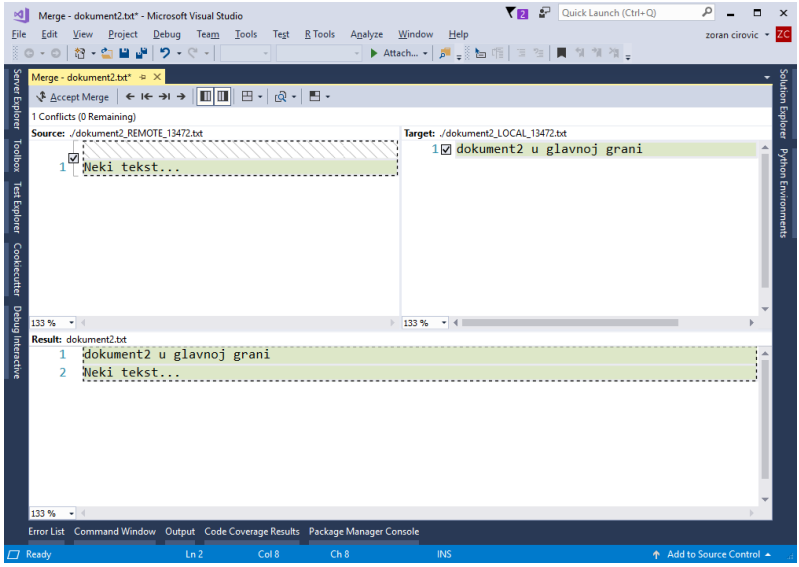
```
Merging:
dokument2.txt
Normal merge conflict for 'dokument2.txt':
  {local}: created file
  {remote}: created file
Hit return to start merge resolution tool (vsdiffmerge):
```

Obratite pažnju da će pritiskom na ENTER da bude otvoren neki alat za spajanje.

Git nema sopstveni ugrađeni alat za vizualni prikaz i rešavanje konflikata. Kad nađete program koji vam odgovara možete ga postaviti kao merge.tool alat:

```
git config --global merge.tool /putanja/do/programa
```

Na slici je prikazan jedan takav alat koji dolazi uz Visual Studio - vsdiffmerge.



Slika 4.3. Alat u okviru Visual Studio radnog okruženja za rešavanje konflikata pri spajanju

Nakon snimanja promena imamo sledeći status:

```
$ git status
```

```
On branch master
You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)
Changes to be committed:
        new file:   dokument-Y.txt
Unmerged paths:
  (use "git add <file>..." to mark resolution)
        both added:      dokument2.txt
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        dokument2_BACKUP_13472.txt
        dokument2_BASE_13472.txt
        dokument2_LOCAL_13472.txt
        dokument2_REMOTE_13472.txt
        sh.exe.stackdump
```

Sada eksplicitno dodajemo dokument koji smo dobili spajanjem u glavnoj grani:

```
$ git add dokument2.txt
```

```
admin@DESKTOP-KUT132H MINGW32 /d/myRepo (master|MERGING)
$ git commit -m "v-2.0"
[master fa5eb98] v-2.0
```

Obratite pažnju da je u tekućem folderu generisano nekoliko fajlova koje nisu deo verzije i koje možete obrisati ili uključiti u praćenje. Ako ste sigurni da ove fajlove nikada nećete dodavati u reviziju i želite da ih nakon spajanja automatski alat za spajanje briše, onda je moguće to postaviti u podešavanja opcijom:

```
git config --global mergetool.keepBackup false
```

U suprotnom potrebno je da obrišete ove fajlove.

Inače, komanda `git diff` bez pratećih argumenata pokazuje razliku u odnosu na indeks. Ako želimo da uočimo razlike u odnosu na poslednji komit treba koristiti `git diff HEAD`.

Brisanje grana

Komanda za brisanje grane:

```
git branch -d naziv
```

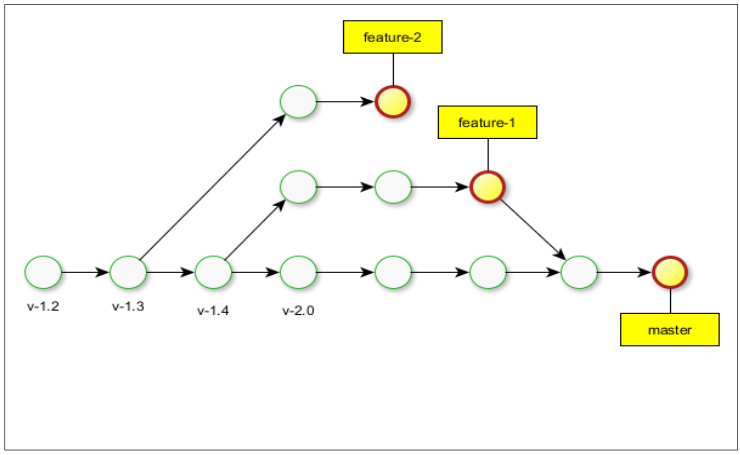
dde je:

`naziv` - naziv grane koja se briše.

Pri tome treba imati na umu da brisanje ne može da se izvrši u slučaju ako pokušamo da brišemo:

1. tekuću granu – HEAD
2. granu koja još nije spojena na tekuću.

Pogledajmo sledeći primer prikazan na slici.



Slika 4.4. Istorija revizija

Za gornju sliku akcije i prateći rezultat bio bi:

```
$ git branch -d feature-1
Deleted branch feature-1 (was 54149ea).
$ git branch -d feature-2
```

```
error: The branch 'feature-2' is not fully merged.
If you are sure you want to delete it, run 'git branch -D feature-
2'.
```

```
$ git branch -d master
```

```
error: Cannot delete the branch 'master' which you are currently
on.
```

Pitanja i zadaci

1. Pokrenite alatku "gitk - all" za prikaz svih grana. Ne zaboravite da pritisnete F5 posle svake komande da biste vizualizovali promene. Pokušajte i sa nekim drugim alatom.

2. Napravite novu granu pod nazivom "develop".

3. Napravite nekoliko verzija u ovoj grani.

4. Vratite se na granu "master". Napravite nove izmene i komitujte ih.

5. Kako se radi spajanje grana? Sada uradite spajanje "develop" u "master"?

6. Šta su konflikti i kako se razrešavaju pri spajanju grana?

7. Pokazati na primeru grana spajanje grana "develop" i "master" rešavanje konflikta pri spajanju verzija.

8. Kako se vrši brisanje jedne grane?

9. Nakon spajanja uradite na tekućem primeru brisanje grane "develop".

5. Mrežni repozitorijumi

Mrežni repozitorijum se koristi kada se organizuje timski rad ili radi dostupnosti drugim korisnicima. Članovima tima koji ga koriste za timski rad na zajedničkom projektu repozitorijum je dostupan radi:

- Preuzimanja promena sa mreže koje su nastale akcijama drugih učesnika.
- Postavljanje promena koje jedan učesnik uradi.

Ovo poglavlje posvećeno je radu sa mrežnim repozitorijumom, počev od njegovog kreiranja, preko dodavanja nove verzije kao i preuzimanja odgovarajućih verzija. Posebno je obrađeno spajanje mrežne sa tekućom verzijom i obrnuto, kloniranje kao i rad sa tipičnim sekvencama u razvoju.

Git hosting

Ukoliko se odlučimo za rad sa mrežnim repozitorijumom, bilo iz razloga timskog rada ili dostupnosti za druge korisnike moramo definisati da li taj repozitorijum formiramo na sopstvenom serveru ili ćemo ga formirati na nekom od javno dostupnih servera. Većina ozbiljnih hostova namenjenih

za verzioniranje odnosno timski rad podržava Git specifikaciju. Neki od najpopularnijih su:

- http://github.com – besplatan, popularan, verovatno najpoznatiji. Gotovo sve velike kompanije koriste ovaj server.
- http://bitbucket.org – takođe popularan, besplatan i za privatne repozitorijume.
- http://code.google.com
- http://sourceforge.net

Timski rad

Tipičan timski rad obuhvata preuzimanje aktuelne vezije dokumenata, rad na sopstvenoj verziji i na kraju spajanje sa novom aktuelnom verzijom na mreži. Dakle, obuhvata nekoliko koraka:

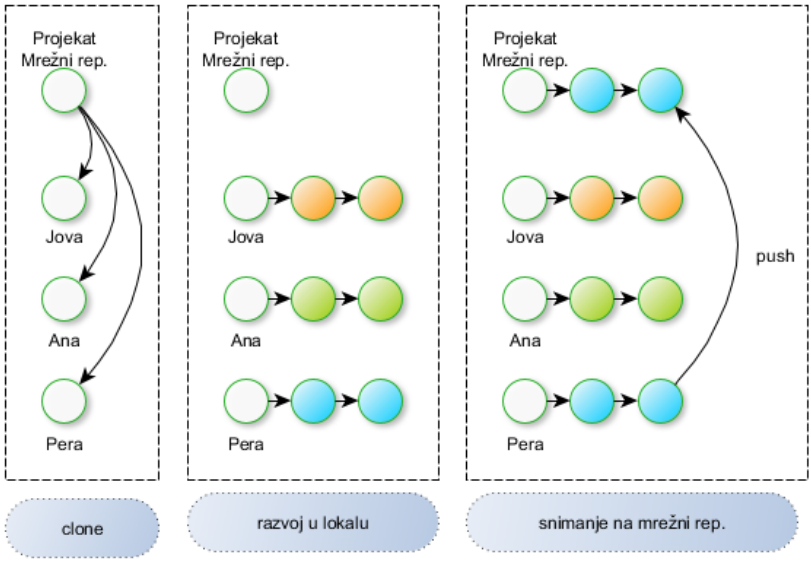
Korak 1. Kloniranje repozitorijuma.

Korak 2. Razvoj lokalnih verzija.

Korak 3. Prebacivanje lokalne verzije na mrežnu.

Korak 4. Eventualno rešavanje konflikta.

Razmotrimo detaljno navedene korake kroz jedan konkretan primer. Recimo da u timskom radu učestvuje više učesnika. Njihov prvi korak bio bi preuzimanje tekuće tj. poslednje verzije sa zajedničke lokacije. Ta lokacija može biti mrežni folder ili folder negde na nekom od hosting sajtova. Ovo je prikazano na narednoj slici u prvom koraku.



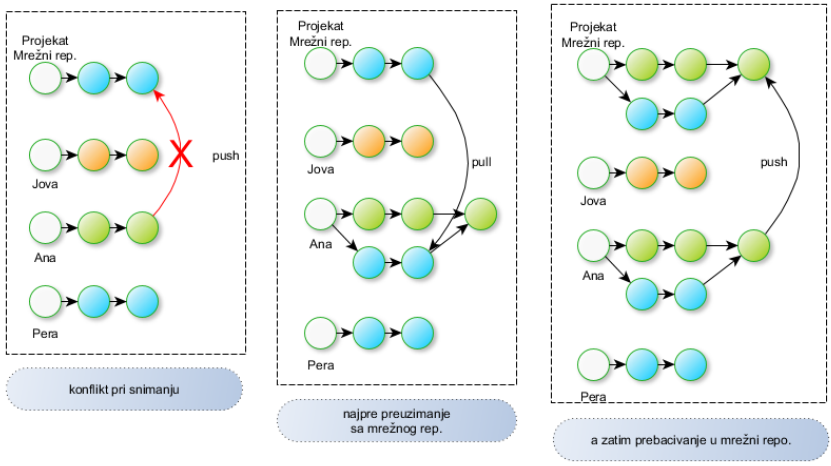
Slika 5.1. Preuzimanje zajedničke verzije, lokalne izmene i promene prvog korisnika

Sledeći korak, prikazan na istoj slici, je lokalni razvoj. Svaki od korisnika kreira svoj razvoj, naravno zasnovan na zajedničkom projektu koji je preuzet sa mrežnog repozitorijuma.

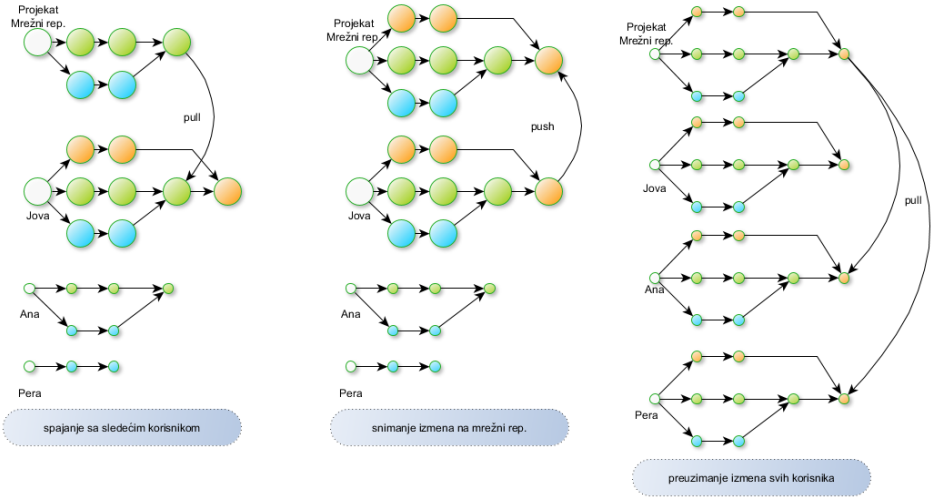
Sledeći korak je početak razmene podataka. Neko od učesnika, neko ko prvi odluči da snimi primene na mrežni repozitorijum, prebacuje svoju verziju. Njegova verzija je nastala od mrežne i promene koje je on učinio lako se prebacuju na mrežu, tako da se na njoj postavlja verzija koja odgovara promenama kod tog korisnika. Ovaj postupak je grafički prikazan na prvoj slici u trećem delu.

Sledi korak u kome drugi učesnik treba da prebaci svoju verziju. Njegove izmene lako nailaze na konflikt sa verzijom na mreži. Pogledajte narednu sliku. Razlog je promena koja je usledila u međuvremenu tj. novim snimanjem od strane prethodnog korisnika. Da bi se problem tj. konflikt rešio, drugi korisnik treba da preuzme promene sa mreže, izvrši spajanje

sa svojim izmenama, a zatim se tako spojena verzija ponovo šalje na mrežu. Takva verzija sadrži podatke od oba korisnika.



Slika 5.2. Postupak sinhronizacije



Slika 5.3. Sinhronizacija ostalih korisnika

Postupak se nastavlja na isti način sa ostalim korisnicima. Svaki od njih prvo preuzima sve sa mreže, vrši spajanje sa svojim promenama, a novu verziju koja sadrži sve spojeno šalje ponovo na mrežu.

Rad sa udaljenim repozitorijumima

Git radi sa udaljenim repozitorijumima tako što kreira njihovu lokalnu sliku. Pri tome se može raditi sa više repozitorijuma istovremeno. Svaki od njih je identifikovan nekim lokalnim jedinstvenim imenom – alijasom.

Kada se radi sa jednim udaljenim repozitorijumom, obično se takav naziva **origin**.

Na sličan način se vrši imenovanje grana: remote/name/branch. Na primer

- remote/origin/master predstavlja glavnu granu udaljenog repozitorijuma origin.
- master predstavlja glavnu lokalnu granu

Dodavanje udaljenog repozitorijuma

Dodavanje udaljenog repozitorijuma u tekući zahteva poznavanje URL lokacije istog. U tom slučaju vrši se komandom:

```
git remote add naziv url
```

gde je:

- `naziv` – lokalni alijas koji identifikuje udaljeni repozitorijum, pošto ih može biti više,
- `url` – lokacija udaljenog repozitorijuma.

Na primer:

```
git remote add origin https://github.com/zcirovic/
ist.primer1.git
```

```
git remote add origin https://zcirovic@bitbucket.org/
zcirovic/ist.primer1.git
```

Moguće je raditi sa više udaljenih repozitorijuma, naravno koristeći različite nazive. Pregled korišćenih repozitorijuma i prateće *url* lokacije možete dobiti komandom:

git remote *[-v]*

Opcija –v prikazuje URL adrese udaljenih repozitorijuma. Na primer:

```
$ git remote -v
```

```
netOrigin       //ZC-A7/netRepo (fetch)
netOrigin       //ZC-A7/netRepo (push)
origin  //ZC-A7/wwwRepo (fetch)
origin  //ZC-A7/wwwRepo (push)
```

Napomena: Ako jedan projekat počinje kopiranjem podataka sa mrežnog repozitorijuma, onda je opcija kloniranja prva koja se mora uraditi. Više o kloniranju u jednom od narednih poglavlja.

Ukoliko se predomislite i želite da uklonite udaljeni repozitorijum to se čini na sličan način:

git remote rm *naziv*

Dodavanje udaljenog repozitorijuma znači kreiranje posebne grane na lokalnom repozitorijumu koja je zadužena za rad sa udaljenim repozitorijumom i ima specifičan naziv. Naziv se sastoji od delova koji označavaju da se radi o grani za rad sa udaljenim repozitorijumom kao i naziv grane. Pogledajmo prikaz svih grana:

```
$ git branch -a
```

```
* master
  remotes/origin/HEAD -> origin/master
  remotes/origin/master
```

Snimanje

Snimanje lokalnih izmena na udaljeni repozitorijum vrši se komandom:

`git push` `[--tag]`

Ova komanda snima tekuću granu

- ako promene u grani prate rast grane, onda se lokalne promene šalju na udaljenu granu
- ako ne, onda se ništa ne snima
- u slučaju konflikta prvo treba da se vrši izvršavanje **`git pull`** naredbe.

Za svaku granu koja je ažurna za dodavanje *upstream* referencu za mrežni repozitorijum koju koristi git-pull i druge naredbe bez argumenta. Dakle, nakon što ste snimili sa *push -u* vašu lokalnu granu ova će se automatski povezati sa udaljenom granom, pa možete koristiti git pull bez ikakvih argumenata.

`git push -u` `remoteName branchName`

gde je:

`-u` – skraćenica od `--set-upstream` podešava informacije praćenja za **push**

`remoteName` – (npr. **`origin`**) naziv udaljenog rep.

`branchName` – (npr. **`master`**)naziv grane

Primer:

```
$ git push -u origin master
```

```
Enumerating objects: 18, done.
Counting objects: 100% (18/18), done.
Delta compression using up to 12 threads
Compressing objects: 100% (11/11), done.
Writing objects: 100% (18/18), 1.46 KiB | 748.00 KiB/s, done.
Total 18 (delta 3), reused 0 (delta 0)
remote: Resolving deltas: 100% (3/3), done.
To https://github.com/zcirovic/ist.primer1.git
 * [new branch]      master -> master
Branch 'master' set up to track remote branch 'master' from
'origin'.
```

Preuzimanje izmena

Preuzimanje verzija sa udaljenog repozitorijuma obavlja se pomoću komandi:

`git fetch`

`git pull`

Tokom preuzimanja vrši se sinhronizacija tj. ažuriranje podataka lokalnog repozitorijuma na osnovu udaljenog i to uz sledeće aktivnosti:

- Preuzimanje novih snimanja (komita) sa udaljenog repozitorijuma.
- Ažuriranje reference sa remote/name/* tako da odgovaraju novoj poziciji na korišćenoj grani.

Naredba `fetch` preuzima promene sa mrežnog skladišta i prebacije ih na lokalnu granu koja je zadužena za rad sa udaljenom (na primer: `origin/master`). Pogledajte sledeći primer:

```
$ git fetch
```

```
remote: Enumerating objects: 4, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From //ZC-A7/wwwRepo
   d2ae608..b104682  master     -> origin/master
```

```
$ git status
```

```
On branch master
Your branch is behind 'origin/master' by 1 commit, and can be
fast-forwarded.
  (use "git pull" to update your local branch)

nothing to commit, working tree clean
```

Ako pokušamo da se prebacimo na ovu granu, dobijamo sledeću poruku:

```
$ git checkout origin/master
```

```
Note: checking out 'origin/master'.

You are in 'detached HEAD' state. You can look around, make
experimental changes and commit them, and you can discard any
commits you make in this state without impacting any branches by
performing another checkout.

If you want to create a new branch to retain commits you create,
you may do so (now or later) by using -b with the checkout command
again. Example:

  git checkout -b <new-branch-name>

HEAD is now at b104682 v4
```

Očigledno da ova grana nije ista kao grane sa kojima inače radimo i da služi da pruži dodatnu funkcionalnost samom Git-u. Ono što ona ipak pruža je mogućnost spajanja.

```
$ git merge origin/master
```

```
Updating d2ae608..b104682
Fast-forward
 dokument4.txt | 0
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 dokument4.txt
```

Spajanje

Spajanje udaljenih izmena sa tekućom lokalnom granom vrši se pomoću eksplicitne komande:

`git merge`

Na primer:

```
git merge origin/master
```

U praksi obično se koristi `git pull`, koji je alijas za dve komande:

`git pull`	`git fetch` `git merge origin/master`

Ovo je jako važan momenat u razumevanju tehnike spajanja. Ranije je spominjan i koristi se i na ovom mestu. Dakle, da bi se spajanje uspešno obavilo najpre se podaci prebacuju u lokalnu granu, a zatim se spajaju sa tekućom.

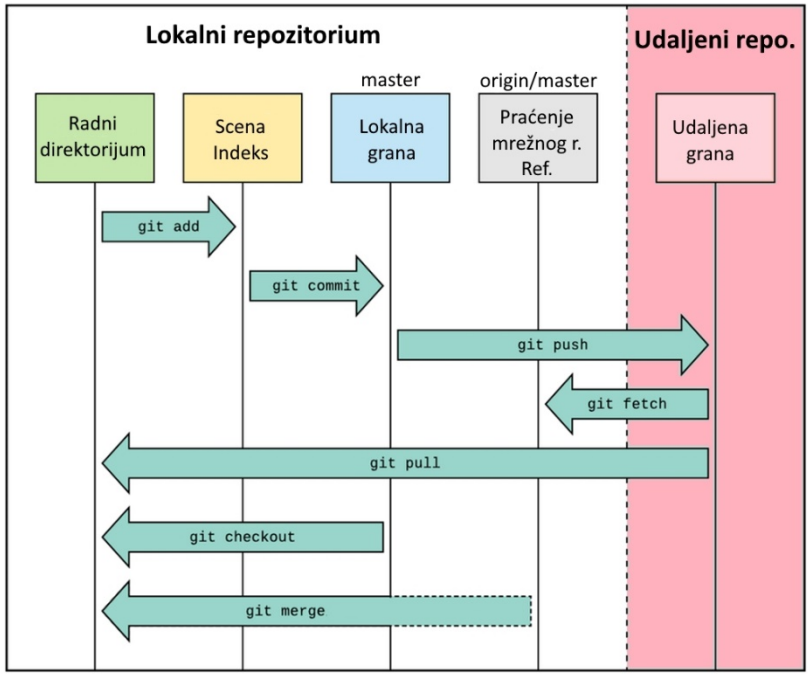
U opštem slučaju, spajanje podataka lokalnog i mrežnog repozitorijuma podrazumeva upotrebu push, pull i fetch komandi. Lokalna pomoćna grana za rad sa udaljenim repozitorijumom, obično `origin/master`, je sinhronizovana sa mrežnom u nekom stanju.

`Push` - vrši se prebacivanje lokalnih kopija na mrežni repo. Ne utiče na stanje na pomoćnu granu.

`Fetch` - vrši preuzimanje mrežnih verzija na lokalnu pomoćnu. Neophodno je uraditi spajanje da bi se te izmene koristile u lokalnoj verziji.

`Pull` - radi preuzimanje i spajanje mrežne verzije sa lokalnom.

Sve ovo je prikazano na narednoj slici.



Slika 5.4. Uporedni prikaz osnovnih funkcija u radu sa mrežnim repo.

Remote primer:

- `(git init –bare –shared // remote repo)`
- `git init // local repo`
- `git commit`
- `git remote add origin shared_url`
- `git push //nista`

- `git push –u origin master`
- `git commit`
- `git commit`
- `git push`
- Neki drugi korisnik snima svoje verzije
- `git commit`
- `git push // konfilkt`
- `git fetch // nove ver. na origin/master granu`
- Radi se spajanje u lokalu. `git merge origin/master` i resavanje konflikata u lokalu
- Zatim se radi `git pull,` tj ponovo se preuzimaju izmene, a sada
- `git push` – radi bez konflikta

Kloniranje

Kloniranje repozitorijuma predstavlja formiranje lokalne kopije udaljenog repozitorijuma na lokalu. Kloniranje omogućava dalji rad sa lokalnom granom preko koje će se obavljati sinhronizacija podataka sa udaljenim repozitorijumom. Komanda je:

```
git clone url [dir]
```

Zapravo, `git clone` je skraćena komanda tj. sekvenca nekoliko komandi:

- `git init dir`
- `cd dir`
- `git remote add origin url`
- `git fetch`
- `git checkout master`

U praktičnoj upotrebi retko se koristi ova sekvenca komandi već jednostavno komande:

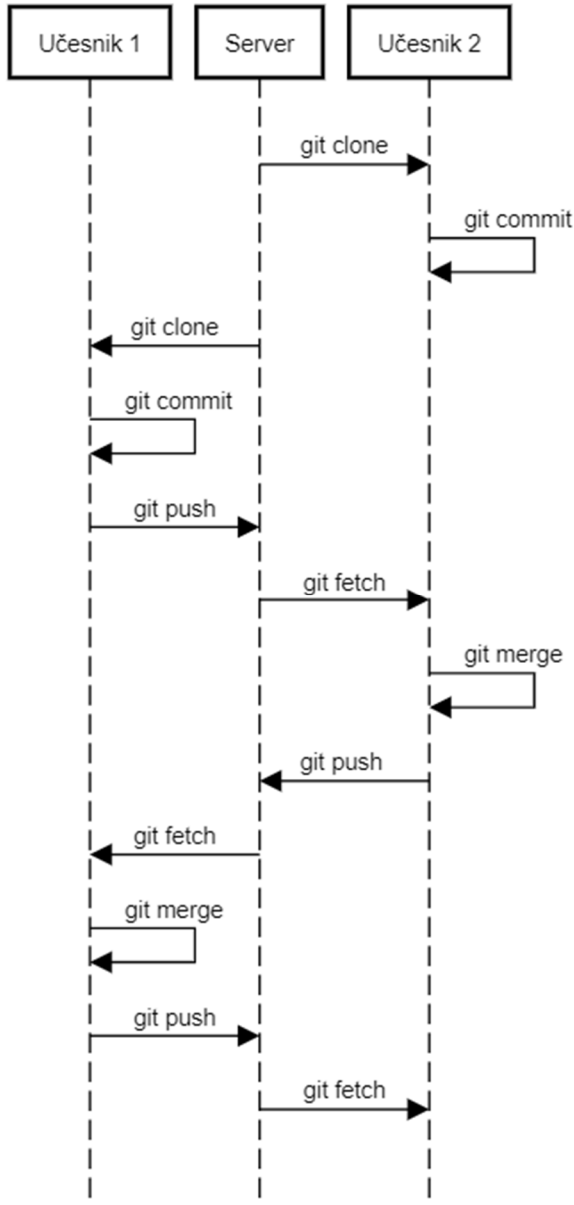
`git clone` odnosno `git pull` .

Tipične sekvence

Najčešće korišćena sekvenca komandi u radu sa mrežnim repozitorijumom koja obezbeđuje sinhronizaciju sastoji se od komandi:

`clone, commit, commit,…, fetch, merge, push`.

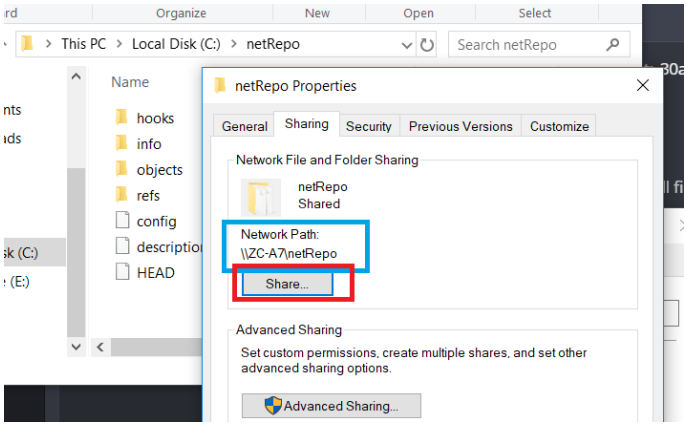
Vremenski dijagram ove sekvence za dva učesnika dat je na slici.



Slika 5.5. Vremenski dijagram

Mrežni folder kao mrežni repozitorijum

Postavljanje određenog foldera u ulogu mrežnog repozitorijuma vrši se u dve faze. Prva je da taj folder mora imati dostupnost do svih korisnika. Ako je u pitanju Windows operativni sistem ovo se postiže podešavanjem opcije deljenja.



Slika 5.6. Određivanje foldera sa privilegijama

Drugi korak je inicijalizacija repozitorijuma za mrežni rad

```
git init --bare --shared
```

Nakon ove fazi vrši se povezivanje lokalnog repozitorijuma sa mrežnim, kako je već objašnjeno.

```
git remote add origin \\ZC-A7\netRepo
```

ili ako je samo korišćen folder kao repozitorijum na nekoj lokaciji:

```
git remote add origin2 c:\\netRepo
```

Nakon ovih postavki može se proveriti spisak mrežnih repozitorijuma za lokalno skladište. Na primer:

```
$ git remote
```

```
origin
origin2
```

Pitanja i zadaci

1. Objasnite čemu služi mrežni repozitorijum i koje poznajete?
2. Kako se kreira jedan udaljeni repozitorijum?
3. Prikazati listu udaljenih repozitorijuma, dodati neki, i ponoviti prvi korak.
4. Kako se podaci preuzimaju sa mrežnog repozitorijuma?
5. Šta je konflikt i kada se događa?
6. Kako se vrši spajanje sa mrežnim repozitorijumom?
7. Kako izgleda tipična sekvenca Git komandi u timskom radu?

6. Node.js

Ovo poglavlje je posvećeno osnovama JavaScript platforme Node.js. U okviru ovog poglavlja prvo se prezentuju karakteristike Node.js platforme sa posebnom naglaskom na asinhroni princip rada. Zatim, prikazuje se instalacija Noda kao i način upotrebe. Nakon objašnjenja korišćenja osnovnih komandi, prezentuje se kreiranje prve aplikacije, a zatim i komande za rad sa konzolnom prozorom.

Posebna pažnja posvećena je radu sa paketima. Zatim je prikazano uključivanje novih modula, eksportovanje promenljivih i funkcija kao i povezivanje lokalnih modula. Opisan je i prikazan u potpunosti, praktičan primer kreiranja sopstvenih lokalnih i mrežnih modula, njihovo postavljanje na repozitorijum odnosno registar modula.

Na kraju, prezentovano je kako se koriste sopstveni lokalni odnosno mrežni moduli.

Uvod

JavaScript je programski jezik koji razumeju svi veb čitači. Njegova prvobitna namena je i bila da pruži posebne mogućnosti na strani klijenta. Danas je JavaScript, po svim analizama, prvi programski jezik u pogledu broja korisnika. Njegova jednostavnost, fleksibilnost kao i velika zajednica čini da ovaj jezik dobija vremenom i druge izmene. Node.js je platforma

bazirana na jeziku JavaScript sa ciljem da se primenjuje na serverskoj strani.

Platforma nastala 2009. godine, a razvio ga je Rajan Dal (eng. *Ryan Dahl*). Osnovni cilj nove platforme je da se primeni jedinstvena tehnologija u razvoju klijent-server aplikacija. Do tada je bilo drugačije. Za pristup bazama podataka i složenu obradu bilo je neophodno za serversku stranu koristiti C#, Javu, PHP ili neki drugi jezik, dok za obradu na klijentskoj strani koristiti JavaScript.

To znači da su bile potrebne bar dve sintakse, u okviru istog projekta.

Karakteristike

Node.js omogućava upotrebu JavaScript jezika za front-end i back-end. To znači da se kompletan klijent-server projekat može realizovati pomoću jednog programskog jezika.

Veoma je brz. Baziran na Guglovim V8 JavaScript mašini. Zapravo, nekoliko puta je brži od ostalih skriptnih jezika poput Ruby-a i Python-a.

Poseduje neblokirajuću arhitekturu. Takva arhitektura je idealna za rad veb aplikacija u realnom vremenu (eng. *real-time*). Kompletan Node.js ekosistem čine na hiljade paketa specifičnih imena, izgrađen je oko neblokirajuće arhitekture.

Ovako je nastala platforma tj. ceo ekosistem koji se savršeno uklapa sa zahtevima izrade modernih klijent-server aplikacija.

Asinhroni princip

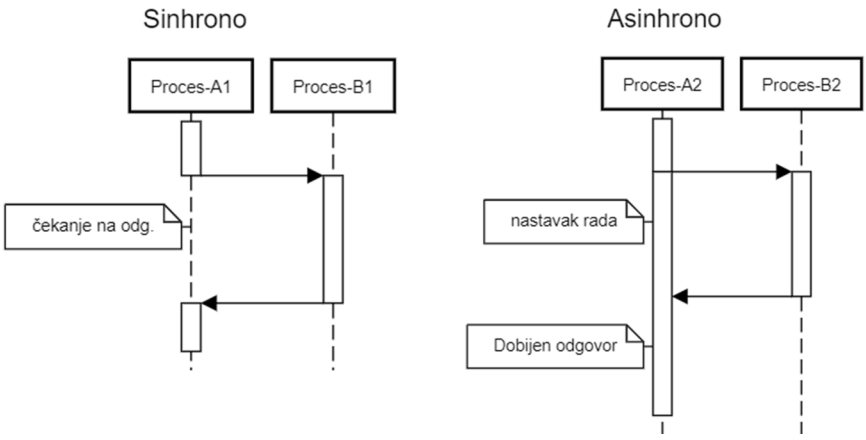
Sinhrona operacija podrazumeva takvu operaciju čiji se rezultat čeka na izvršavanje. Tradicionalno linije koda nekog programa se izvršavaju sinhrono. Ukoliko izvršavanje jedne linije koda ne zahteva više procesorskog vremena onda je to prihvatljiv koncept.

Međutim, sinhrono izvršavanje predstavlja veliki problem ukoliko se zahteva dugo čekanje; na primer čitanje podataka iz neke baze podataka, čekanje odziva od servera, čitanje fajlova i slično.

Jedno moguće rešenje, koje nude na primer programski jezici Java i C#, je uvođenje posebne niti za ovakve operacije.

Međutim, rad sa više niti može zadati dosta problema programerima, naročito ako više niti istovremeno pristupa istim resursima. Na primer: jedna nit menja neki podatak, druga nit ga čita, treća briše.

Node.js tj. JavaScript ima drugačiji pristup. Uvek se izvršava samo jedna Nit. Kada se izvršavaju spore operacije, kao što je čitanje podataka iz baze, program ne čeka, nego ide na izvršavanje drugih delova koda. Kada se operacija vrati, pokreće se povratna (eng. *callback*) funkcija i rezultat se procesira dalje. Node.js nudi jednostavan, brz, asinhroni model zasnovan na događajima (eng. *event-driven model*) za izradu modernih veb aplikacija. Na slici je prikazan uporedni vremenski dijagram ova dva načina rada.



Slika 6.1. Sinhroni / asinhroni model rada

Upotreba

Node.js može da se koristi u različite svrhe. Pošto je zasnovan na V8 mašini veoma je optimizovan za HTTP saobraćaj, tako da je najčešća imena u veb aplikacijama na serverskoj strani, tačnije za http saobraćaj. Međutim, Node.js se može koristiti i za različite druge aktivnosti veb usluge kao što su:

- Web servisi: kao što su REST ili API.
- Aplikacije u realnom vremenu, na primer: *Real-time multiplayer games.*
- Pozadinski servisi kao što je: *cross-domain*, *server-side requests*.
- Ostale aplikacije koje su zasnovane na veb-u: *Web-based applications*.

Instalacija

Za instalaciju možete koristiti zvanični sajt i preuzimanje sa lokacije: https://nodejs.org/download/. Pošto je Node.js platforma otvorenog koda, druga opcija je preuzimanjem preko repozitorijuma Github:

https://github.com/joyent/node/wiki/Installing-Node.js-via-package-manager. Nakon instalacije obavezno uradite i verifikaciju instalacije. Postupak je sledeći:

Nakon instalacije, u nekom konzolnom prozoru, *Command prompt* prozoru, unesemo:

`node -v`, ili

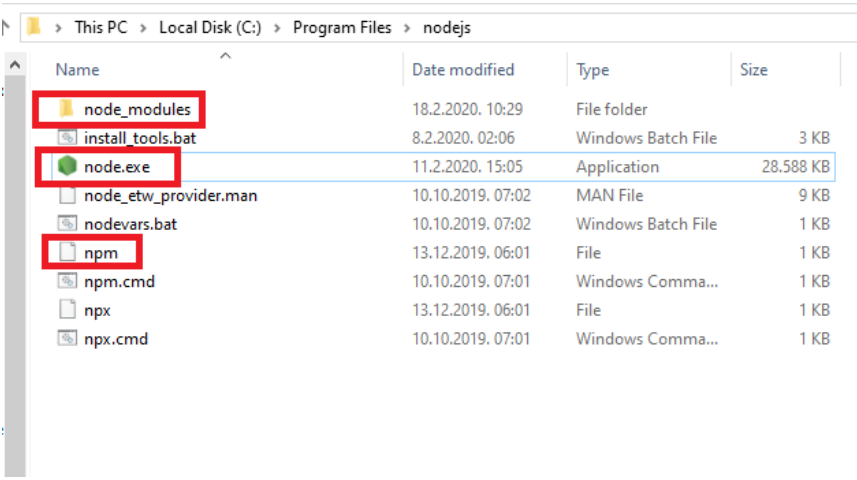
`node --version`.

Treba da se prikaže verzija instaliranog node-a.

Posle toga ćemo proveriti verziju programa *Node package manager* - skraćeno NPM-a, komandom `npm -v`.

Lokacija

Prilikom instalacije određuje se folder za instalaciju, a istovremeno se ta putanja postavlja da bude dostupna u kasnijem radu preko sistemskih promenljivih.



Slika 6.2 Putanja i sadržaj do instaliranih paketa

Ako pogledate ovaj folder videćete da se tu nalazi nekoliko fajlova koji su izvršni kao i **node_modules** folder.

- **node** - pokreće Node.js virtuelnu mašinu. Ako se ovom fajlu prosledi JavaScript fajl onda ga on pokreće, inače otvara se *Command prompt*.
- **npm** – upravlja Node.js paketima
- **node_modules** – sadrži instalirane pakete.

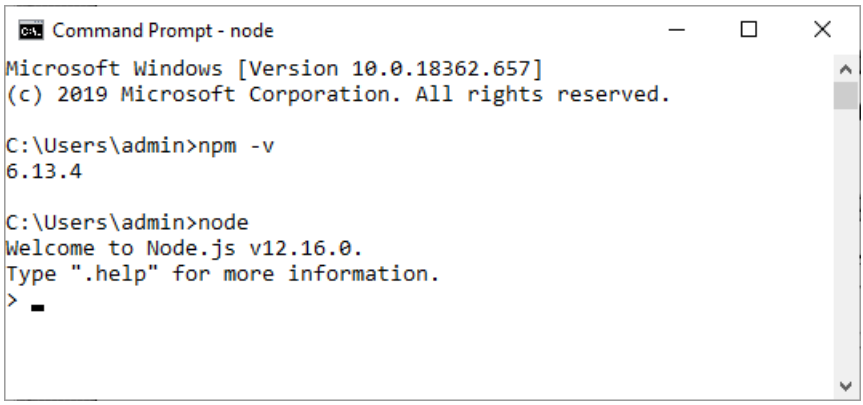
REPL

Interaktivna (eng. *shell*) konzola za Node.js naziva se REPL (eng. ***R**ead-**E**valuate-**P**rint **L**oop*).

Ova konzola je prostor u koji možete u realnom vremenu da pišete čist JavaScript i procenjujete svoj kod u prozoru terminala. U konzolnom prozoru Node.js čita vaš kod odnosno ispisuje poruke, a rezultati se

štampaju na vašoj konzoli. U ovom odeljku prikazaću nekoliko akcija koje možete primeniti u REPL-u.

Već ste koristili terminal da biste proverili da li je Node.js pravilno instaliran. Drugi način da vidite da li je instalacija uspela je da ukucate node i pritisnete taster *Enter*. Ova komanda prebacuje rad u interaktivan Node.js konzolni prozor. Dakle, naredba je uspešna kada vidite promenu terminala sa oznakom `>`. Da biste izašli iz ovog upita, otkucajte `.exit` ili pritisnite `Ctrl-C` dva puta.



Slika 6.3 Provera npm verzije i prelaz u REPL interaktivni mod

Komande

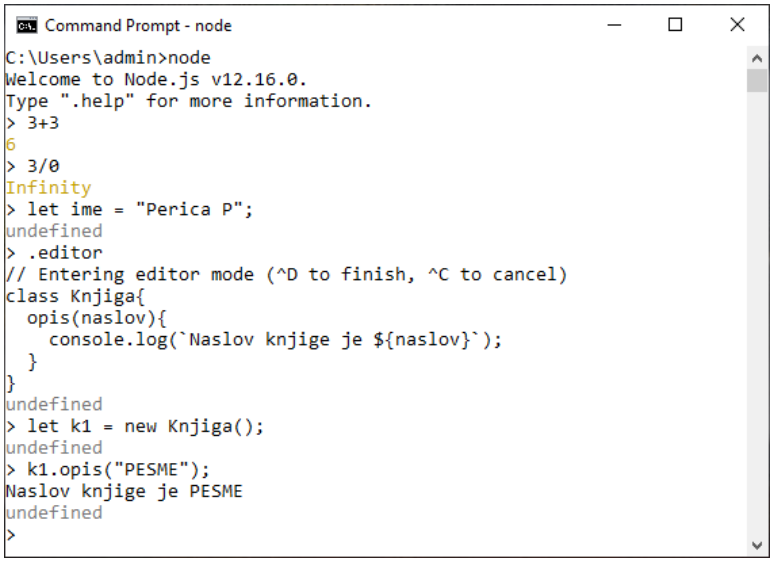
U interaktivnom okruženju dostupne je pisati standardni JavaScript kod ali i koristiti posebne naredbe. Veći deo ovih naredbi prikazan je u narednoj tabeli.

Tabela 5.1. REPL komande

Komanda	Opis
`.break` (`.clear`)	Napušta blok dok smo u REPL sesiji, korisno ukoliko se zaglavite u nekom bloku koda. clear dodatno briše lokalni kontekst.

.editor	Otvara se interni editor za unos više linija koda. Prečicom Ctrl-D čuva se napisano i napušta editor, tj. vraćamo se u REPL okruženje.
.exit	Napušta se REPL
.help	Lista komandi i kratak opis značenja.
.load	Iza komande se navodi ime fajla koji se učitava u REPL, npr: .load zdravoSvete.js
.save	Iza komande se navodi ime fajla u koji se snima tekuća REPL sesija. .save test.js
tab	Prikazuje se lista svih komandi.
Up/Down	Prikazuju se prethodne unete komande u REPL.
ctrl + c (dva puta)	Završava se tekuća komanda (Napušta se REPL okruženje).
ctrl + d	Napušta se REPL okruženje.

Na primer:



Slika 6.4. Primena REPL komandi

Prva aplikacija

Nakon instalacije Node.js okruženja sledeći korak je kreiranje prve aplikacije. Prvi primer napisaćemo koristeći isključivo node okruženje i preko terminala. Postupak je sledeći:

1. Otvoriti neki tekst editor
2. Uneti JS kod:

```
3. console.log("Zdravo svete!");
```

4. Sačuvati fajl pod nazivom zdravoSvete.js.

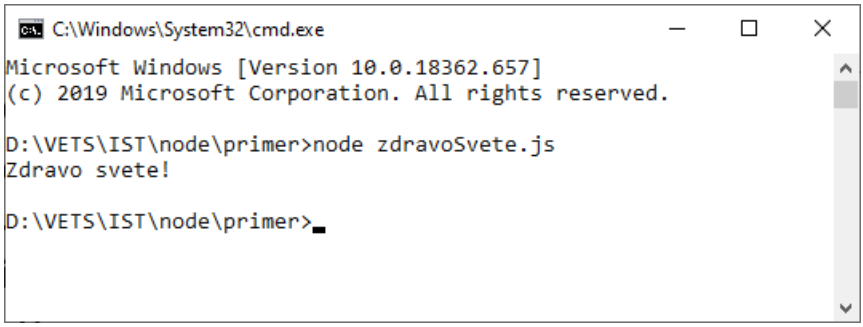
Zaista, početak je jednostavan. Kreirana je prva aplikacija. Pokretanje se obavlja preko konzolnog okruženja na sledeći način.

- Otvorite konzolni prozor i pozicionirajte se u folder u kome ste snimili dokument zdravoSvete.js

- U konzolni prozor unesite:

  `node zdravoSvete.js`

Na slici je prikazano pokretanje i odgovor sistema.



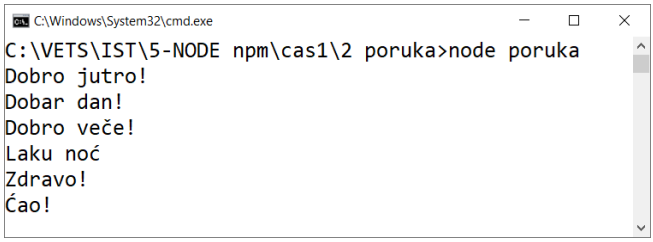
Slika 5.5. Pokretanje jedne node aplikacije

Konzolni prozor

Napišimo sada i našu drugu aplikaciju. Ona treba da prikaže upotrebu lokalnih JS promenljivih kao i ispisivanje njihovih vrednosti u konzolnom prozoru. Unesite sledeći kod:

```
let poruke1 = [
    "Dobro jutro!",
    "Dobar dan!",
    "Dobro veče!",
    "Laku noć"
  ];
  let poruke2 = [
    "Zdravo!",
    "Ćao!"
  ];
  poruke1.forEach(x=>console.log(x));
  poruke2.forEach(x=>console.log(x));
```

Pokretanjem ove aplikacije dobija se:



Slika 6.6. Izvršavanje druge aplikacije

Jedan od najkorisnijih modula u Node.js tokom razvoja je konzolni modul. Ovaj modul pruža funkcionalnosti potrebne tokom ispravki grešaka odnosno razvoja koda. Pošto je ovaj modul široko zastupljen, njega nije potrebno učitavati koristeći require() naredbu, koju ćemo inače kasnije opisati. Za primenu, jednostavno se poziva odgovarajuća funkcija konzolnog paketa pozivom:

```
console.<function> (<parametri>).
```

Jedan parametar može biti string ili objekat koji se može pretvoriti u string. Primeri funkcija i parametara:

- `log("Postoji %d knjiga",4)`
  `>>Postoji 4 knjiga`
- `info([data],[...])` Isto kao `console.log.`
- `error([data],[...])` Isto kao `console.log;` međutim, izlaz se istovremeno šalje na stderr. Obično je poruka praćena oznakom da je u pitanju greška.
- `warn([data],[...])` Slično kao `console.error.`
- `dir(obj)` Ispisuje string reprezentaciju JavaScript objekta na konzolu, na primer:
  `console.dir({ime:"Nemanja", role:"Admin"});`
  `>> { ime: 'Nemanja', role: 'Admin' }`
- `time(label)` Pridružuje tajmer na labelu.
- `timeEnd(label)` Kreira promenu između tekućeg vremena i *timestamp*-a pridruženog labeli i kreira izlaz rezultata. Na primer:
  - `console.time("FileWrite");`

```
- f.write(data); //traje oko 500ms
- console.timeEnd("FileWrite");
- >> FileWrite: 500ms
```

- `trace(label)` Ispisuje *stack trace* tekuce pozicije koda u stderr. Na primer:

```
module.trace("traceMark");
>>Trace: traceMark
  at Object.<anonymous> (C:\test.js:24:9)
  at Module._compile (module.js:456:26)
  at Object.Module._ext.js (module.js:474:10)
  at Module.load (module.js:356:32)
  at Function.Module._load (module.js:312:12)
  at Function.Module.runMain(module.js:497:10)
  at startup (node.js:119:16)
```

Instalacija paketa

Tipičan rad sa Node.js aplikacijama podrazumeva povremenu instalaciju paketa specijalizovanih imena. Zato odmah, na početku dajemo način instalacije paketa.

Instalacijom Node.js paketa, dobija se istovremeno i alat za rad sa paketima tzv. paket menadžer `npm`.

Listu npm komandi dobijate unosom u terminal:

```
npm -l
```

Instalacija novog paketa vrši se komandom:

```
npm install <package> [--global][-s]
```

Osim naziva paketa `<package>` važno je da se definiše način instalacije. Dodatak `[--global][-s]` omogućava globalnu instalaciju paketa tj.

snimanje paketa u folder koji je dostupan za sve aplikacije odnosno njegovu upotrebu bilo gde na računaru. Ako se instalira određena verzija, verzija se navodi iza @: npm install express4@4.2.0. Ukoliko se navede samo prvi broj verzije na primer: `npm install express@4`, instalira se poslednja verzija veća od 4. Poslednja verzija modula se može instalirati koristeći i ključnu reč `latest`.

`npm install` – komanda bez argumenata. Često je korišćena. Pokreće instalaciju svih biblioteka koje su navedene u fajlu package.json kao biblioteke koje su neophodne za aplikaciju (eng. *dependencies*).

Instalacija jednog modula znači zapravo preuzimanje istog sa mreže Node.js modula i njegovo snimanje u folder node_modules. Ako je instalacija globalna, onda se moduli smeštaju u zajednički folder na računaru gde je instaliran Node.js, inače se smeštaju u folder aplikacije. Modul je obično zavisan od drugih modula. Ovo je jako važno uočiti. Ova zavisnost je opisana posebnim odeljkom u package.json fajlu koji se naziva: *dependencies*.

Instalacija razvojnih alata

Prilikom instalacije modula mogu se postaviti i različiti flegovi. Ako je modul potreban samo u razvoju, za njegovu instalaciju se može koristiti fleg `-D`. Ovo je korisno za instaliranje programa za otkrivanje grešaka tzv. debagera i alata za praćenje rada i ispravke grešaka. Fleg `--production` prilikom instalacije zaobilazi module navedene pod svojstvom zavisnosti u razvoju. Ukoliko se postavi fleg `-g` to znači da će se modul instalirati globalno. Globalno instalirani moduli se kasnije mogu pozivati pomoću komandne linije dok se lokalno instalirani moduli (podrazumevana opcija prilikom instalacije) mogu pozivati samo u okviru aplikacije u kojoj su instalirani. Neki moduli, mada se koriste često, preporučuju se da se instaliraju lokalno, kako bi se uvek koristila najnovija verzija. Globalni se moraju povremeno ažurirati.

Pogled/provera paketa

Node.js poseduje komande za proveru podatka vezanih za neki paket. Na primer, takvi su podaci o zavisnosti tog paketa od drugih paketa. Komanda je **npm view**. Ovom komando se čitaju podaci o modulu sadržani u package.json fajlu. Na primer:

- `npm view` express
- **npm view** express **dependencies** (samo sekcija dependencies). Na sličan način se mogu dobiti podaci i iz drugih sekcija.

Slika 5.7. Prikaz podataka primenom view komande

Učitavanje modula

Pošto se neki modul instalira, njegovo učitavanje u tekući fajl u kom se modul namerava koristiti, vrši se komandom.

```
require("naziv_modula");
```

Na ovaj način tekućem projektu su dostupne promenljive i funkcije koje su definisane u navedenom modulu. Konkretan način primene videćemo u nastavku.

Node.js koristi CommonJS sistemski modul koji omogućava da se JavaScript izvršava van okruženja jednog veb čitača. CommonJS definiše **require** funkciju za učitavanje modula koji se pišu u odvojenim JavaScript fajlovima, izvršava taj kod i na kraju koristi prosleđene tj. eksportovane objekte iz tih modula.

npm komande

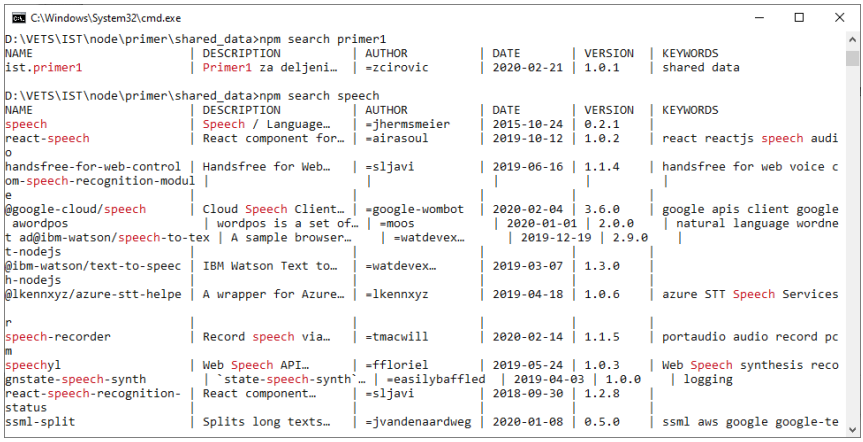
Pogledajmo sada neke osnovne komande vezane za rad sa Node.js paketima.

Tabela 6.2. npm komande

Komanda	Opis
`npm search <naziv>`	pronalazi paketske module na repozitorijumu • npm search primer1, npm search speech
`npm install`	već objašnjeno
`npm uninstall <naziv>`	uklanja navedeni modul • npm uninstall express
`npm pack`	vrši pakovanje modula definisanog u package.json fajlu u .tgz fajl
`npm view <ime>`	prikazuje detalje o modulu • npm view express
`npm publish <ime>`	objavljuje tekući modul definisan sa package.json na registar

`npm unpublish <ime>`	uklanja objavljeni modul sa registra.
`npm owner add/rm/ls`	dodaje/uklanja/prikazuje listu vlasnika paketa na repozitorijumu • npm owner ls ist.primer1

Na primer:



Slika 6.8. Primer primene komande search

Exports

Funkcije se mogu grupisati u okviru jednog fajla, radi bolje organizacije koda. To ima smisla čak i kada je reč o primeni na samo jednoj lokalnoj aplikaciji. Ovakvi fajlovi funkcija se obično smeštaju u zaseban folder, a kasnije se vrši referisanje iz drugih fajlova.

Na primer, ako hoćemo da koristimo podatke o srpskim nazivima dana u nedelji odnosno meseci u godini onda ove podatke postavimo u poseban

folder u jedan fajl srpkalendar.js. Na primer, neka su definisane konstante kao u primeru:

```
const _Dani =["Ponedeljak", "Utorak",. . .];
const _Meseci =["Januar", "Februar",. . .];
```

Za ove promenljive (**const** je jedan od 3 tipa promenljivih u JS) definišimo funkcije koje treba da budu dostupne iz drugih Node.js fajlova, tačnije da budu dostupne iz JavaScript koda drugih fajlova.

JavaScript fajlovi koji se pišu sa namerom da budu deo drugih fajlova koriste **exports** objekat koji izlaže objekte, metode, nizove, promenljive na upotrebu iz drugih JS fajlova koji će isti koristiti kao modul. Na primer:

```
exports.Dani = _Dani;
exports.Meseci = _Meseci;
```

Na ovaj način, konstante koje su definisane u fajlu srpKalendar.js postaju dostupne u drugim JS fajlovima. Eksportovana promenljiva odnosno funkcija može biti korišćena svojim novim nazivom koji se koristi za **export** modul.

Funkcija se eksportuje na veoma sličan naziv kao i promenljiva, s tim što se njoj dodeljuje promenljiva koja je funkcija. Funkcija koja se dodeljuje može da već postoji napisana u kodu, da se piše kada se eksportuje sa ili bez strelica notacije. Na primer:

```
exports.dan = nadjiDan;
exports.mesec = (i)=> { return nadjiMesec(i); }
exports.vikendDani = ()=>{
    return [_Dani[5],_Dani[6]];
}
```

Povezivanje lokalnih modula

Pogledajmo način upotrebe lokalnih modula. Neka je definisani fajl srpKalendar.js postavljen u folder **.\srpKalendar** u odnosu na lokalni **testKalendar.js** fajl.

Prva komanda u okviru **testKalendar.js** fajla je učitavanje lokalnog modula, dakle:

```
const kalendar = require("./srpKalendar/srpkalendar");
```

Ovako učitan modul smešta se u konstantnu varijablu `kalendar`, koja se koristi u nastavku koda. Na primer:

```
kalendar.Dani.forEach(element=>{console.log(element);});
kalendar.Meseci.forEach(element=>{console.log(element);});
console.log(kalendar.vikendDani());
let d=[0,3,6];
d.forEach(idx=>{ console.log(kalendar.dan(idx)); })
```

Upotreba lokalnih fajlova i modula

Pre upotrebe spoljnog fajla ili modula uvek stoji ključna funkcija koja označava obavezno učitavanje modula. Kada je reč o uključivanju lokalnog JavaScript fajla, važno je naglasiti da nije neophodno da postoji, ranije pominjani, package.json fajl. Učitavanje se vrši komandom:

```
const kalendar = require("./srpKalendar/srpkalendar");
```

Pri tome je očigledno da se navodi putanja do željenog fajla - srpkalendar.js, koji se u konkretnom slučaju nalazi na folderu ispod tekućeg pod nazivom srpKalendar. Na ovom mestu zapazite da nastavak u nazivu fajla .js treba izostaviti kao suvišan.

Međutim, ako je uz JavaScript fajl, koji se koristi u drugim JS fajlovima, kreiran i package.json, tada se kaže da je kreiran lokalni modul. Modul se ne mora direktno referencirati koristeći URL ili path putanju, već se koristiti tako što se najpre instalira u nekom folderu gde se koristi node aplikacija ili drugi modul (dakle i u tom folderu treba package.json fajl). Instalacija se izvodi komandom:

```
npm install path
```

Na primer, ako testiramo modul pre postavljanja u repozitorijum, otvorimo folder test, a zatim pozovimo komandu:

```
npm install "../modul kalendar"
```

U nastavku pogledaćemo kako da kreiramo paketski modul i da ga zatim instaliramo I koristimo u nekom kodu.

Paketski moduli i aplikacije

Jedan paketski modul je biblioteka koja se deli, ponovo koristi, instalira u različitim projektima. Postoji zaista veliki broj modula najrazličitijih imena. Na primer modul Express pruža HTTP funkcionalnosti i veoma je pogodan za izradu veb servera, modul Mongoose omogućava rad sa operacionim podacima – ODM (eng. ***O****perational* ***D****ata* ***M****odel*) za ModngoDB.

Jedna Node.js aplikacija sastoji se od velikog broja JavaScript fajlova. Da bi aplikacija bila efikasna i dobro organizovana, ona se obično organizuje i sama po paketima i modulima. Svaki JavaScript fajl ili folder koji sadrži biblioteku predstavlja celinu koja se može opisati i koristiti zasebno kao *modul*.

Node.js modul uključuje **package.json** fajl koji definiše paket. Ovaj paket uključuje meta podatke o paketskom modulu: naziv, verziju, zavisnost od drugih paketa, početni fajl za izvršavanje…

Package.json

Jedine obavezne vrednosti u package.json fajlu su ime i verzija. Ostale se uključuju po izboru. Značenja su sledeća:

- `name` – jedinstveno ime paketa. Npr: "name": "ist.primer1".
- `preferGlobal` – modul će se instalirati globalno ako bude moguće. Npr: "preferGlobal": "true".
- `version` – verzija modula. Npr. "version": "1.0.0".
- `author` – autor projekta.
- `description` – tekstualni opis modula.
- `contributors` – dodatni saradnici. Npr. "contributors": [{"name":"Zoran", "email":"Zoran.Cirovic@gmail.com"}].
- `main` – ulazna tačka aplikacije, glavni skript, binarni ili js fajl. Podrazumevano je `index.js`. Npr. "main":"main.js".
- `repository` – definiše tip i lokaciju repozitorijuma.
- `keywords` – ključne reči koje se koriste u pretrazi.
- `dependencies` – moduli i verzije koji učestvuju u tom modulu. Mogu se koristiti *jocker* znaci poput * i x. Npr. "dependencies":{"http":"latest", "express":"1.x.x", "cookies":"*"}
- `engines` – verzija Noda.js na kome paket radi.
- `devDependencies` - zavisnosti u razvoju - ovo svojstvo se koristi kada je aplikacija u razvojnom (eng. *development*) okruženju, tj. ako globalna promenljiva NODE_ENV ima vrednost development i neki modul se instalirao korišćenjem -D flega. Prilikom pokretanja npm install komande instaliraće se svi moduli koji su ovde navedeni, u suprotnom, ukoliko je aplikacija u produkcionom okruženju, ni jedan od ovih modula neće biti instaliran

Kreiranje

Svaki Node.js projekat ili modul sadrži package.json fajl. Ovaj fajl je u korenu projekta i predstavlja ključni podatak pomoću koga se aplikacija ili modul čuva kao jedinstveni paket u ogromnoj node zajednici programera.

Najpre kreirati folder u kom će biti svi fajovi. Zatim se pozicionirajte u taj folder i koristeći konzolni prozor unesite komadu:

```
npm init
```

Nakon ove komade bićete upitani za pojedine podatke koji ujedno čine sastavni deo package.json fajla koji se kreira.

Obratite pažnju da se prilikom inicijalizacije modula definiše lokacija ne nekom git repozitorijumu tako da bude dostupan preko mreže. U nastavku dat je prikaz dijaloga tokom inicijalizacije:

```
package ime: (3-proba povezivanja) srpkalendar
version: (1.0.0)
description: testiranje sopstvenog modula
entry point: (index.js) srpkalendar.js
test command:
git repository: https://github.com/zcirovic/srpkalendar.git
keywords: srpski kalendar dani meseci
author: zoran cirovic
license: (ISC)
About to write to C:\VETS\IST\5-NODE npm\cas1\4 lokalniModul\3
proba povezivanja\package.json:
{
  "ime": "srpkalendar",
  "version": "1.0.0",
  "description": "testiranje sopstvenog modula",
  "main": "srpkalendar.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "repository": {
    "type": "git",
    "url": "git+https://github.com/zcirovic/srpkalendar.git"
```

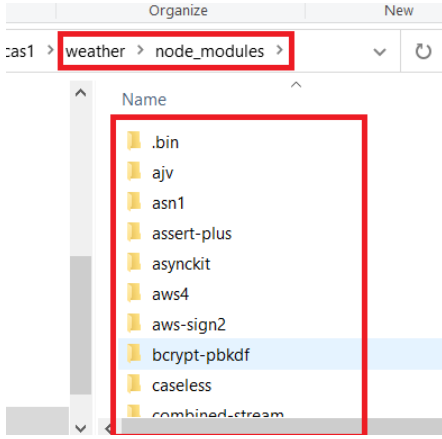
```
  },
  "keywords": [
    "srpski",
    "kalendar",
    "dani",
    "meseci"
  ],
  "author": "zoran cirovic",
  "license": "ISC",
  "bugs": {
    "url": "https://github.com/zcirovic/srpkalendar/issues"
  },
  "homepage": "https://github.com/zcirovic/srpkalendar#readme"
}
Is this OK? (yes)
```

Uključivanje paketa u projekat

Uključivanje jednog paketa u neki projekat najbolje ćemo prikazati na konkretnom slučaju nekog već postojećeg modula. Kasnije, vratićemo se na objavu i uključivanje sopstvenih modula.

Recimo da pišemo aplikaciju u kojoj želimo da prikažemo podatke o vremenskoj prognozi. Za tu svrhu odabrali smo paket **weather-js**. Postupak je sledeći:

- Korak 1. Najpre inicijalizujmo sopstveni projekat. Podsećamo da se to radi u folderu projekta i primenom komande `npm init`.
- Korak 2. Instalirajmo sada željeni paket weather-js. Komanda je:
  `npm install weather-js`
  Nakon ovog koraka kreira se folder `node_modules` sa ovim modulom i onim koji su potrebni za njegov rad, pogledati sliku.



Slika 6.9. Pogled na node_modules folder nakon instalacije paketa weather-js

- Korak 2. Sledi uključivanje novog modula u projekat:

```
var weather = require('weather-js');
   weather.find({search: 'Belgrade, SER', degreeType: 'C'},
   function(err, result) {
      if(err) console.log(err);
          console.log(JSON.stringify(result, null, 2));
});
```

- Korak 3. Testiranje.

```
>node nazivprojekta
```

Ispitivanje zavisnost modula

Ako imamo paket sa svim instaliranim modulima paketima, onda listu korišćenih modula paketa dobijamo komandom:

```
npm list
```

Za konkretni slučaj dobija se lista čije deo prikazujemo:

```
eather@1.0.0 C:\VETS\IST\5-NODE npm\cas1\5 weather
 -- weather-js@2.0.0
  +-- request@2.88.2
  | +-- aws-sign2@0.7.0
  | +-- aws4@1.9.1
  | +-- caseless@0.12.0
```

Ponovimo još jednom, ako želimo da ispitamo određeni modul koji još uvek nismo instalirali, onda možemo koristiti komandu:

`npm view nazivModula.`

Na primer:

`npm view express`,
ili sa detaljima zavisnosti:
`npm view express version dependencies`

Dodavanje zavisnosti

Dodavanje zavisnosti za jedan projekat ili modul, vrši se jednostavnim editovanjem package.json fajla. U opštem slučaju dovoljno je dodati samo pakete na nultom nivou zavisnosti, ostali će biti automatski prepoznati kao uslov i biće preuzeti takođe sa mreže.

Na primer, ako u modulu/projektu srpkalendar nameravamo da koristimo weather-js modul, dovoljno je ubaciti sekciju

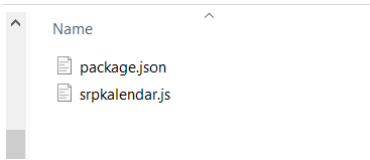
```
"dependencies": {
    "weather-js": "^2.0.0"
}
```

unutar package.json fajla. Nakon ovoga, svi potrebni moduli, dakle moduli navedeni u ovom fajlu biće instalirani komandom:

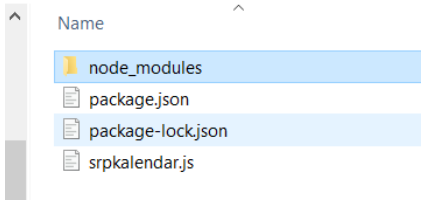
`npm install`

Evo pogleda na folder projekta pre i posle ove komande, nakon dodavanja ove zavisnosti.

pre



posle



Slika 6.10. Pogled pre i posle instalacije zavisnosti

Kreiranje deljenih modula

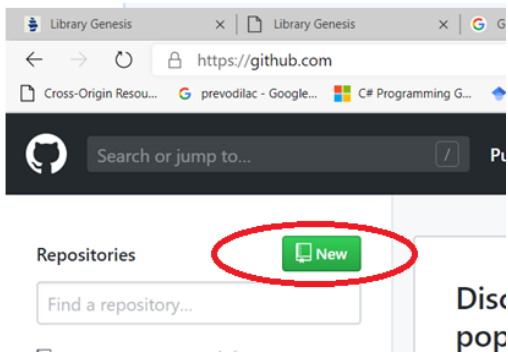
Deljenje npm modula je jako važna osobina modula. Deljenje se obavlja jednostavno i uz male razlike u odnosu na lokalne module.

Za deljenje modula potrebno je, osim modula, da posedujete:

- Mesto na mreži gde ćete postaviti modul koji će tako postati dostupan preko mreže, dakle uz odgovarajuće privilegije.
- Registracija modula u npm registru modula.

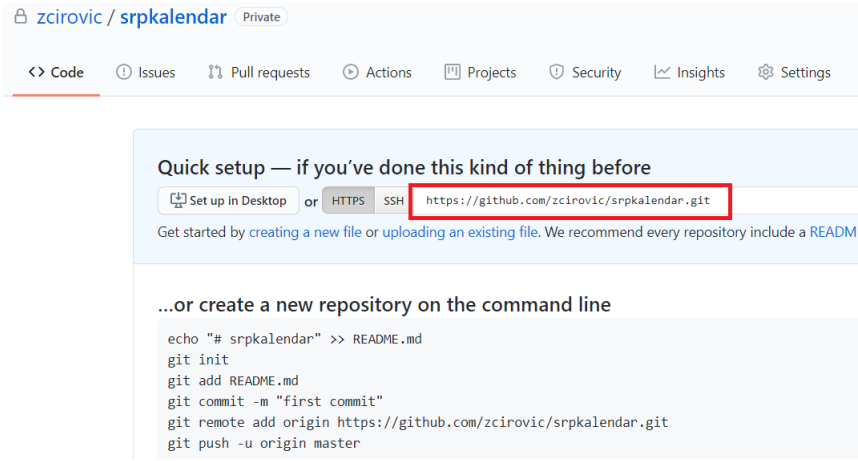
Repozitorijum za modul

Mesto za postavljanje modula može biti jedan od repozitorijuma. Pretpostavimo da za to koristimo Github. Nakon registracije, potrebno je da se odvoji deo za modul koji ćete kasnije postaviti na isti. Deo koji se odvaja za modul je zapravo jedan repozitorijum, pogledajte sliku.



Slika 6.11. Kreiranje posebnog repozitorijuma za modul

Nakon kreiranja repozitorijuma, važno je zapamtiti tj. kopirati url kreiranog repozitorijuma, pogledajte sliku:



Slika 6.12. Određivanje lokacije do mrežnog repozitorijuma

Ovaj repozitorijum treba da čuva npm modul u celosti. Međutim, za registraciju i rad sa modulima koristi se poseban registar `npmjs`.

Prebacivanje fajlova modula vrši se na standardan način, primenom git komandi. Ipak, komande su navedene i na samoj http stranici nakon kreiranja repozitorijuma, pogledati prethodnu sliku.

Najpre kreirajmo fajl .gitignore i dodajmo u njega: `node_modules/` da ne bi kopirali sve module u repozitorijum. Slede komande:

- `git init`
- `git add .`
- `git commit -m "***"`
- `git remote add origin https://github.com/zcirovic/srpkalendar.git`
-
```
$ git push -u origin master
Enumerating objects: 6, done.
Counting objects: 100% (6/6), done.
Delta compression using up to 8 threads
Compressing objects: 100% (5/5), done.
Writing objects: 100% (6/6), 5.17 KiB | 5.17 MiB/s, done.
Total 6 (delta 0), reused 0 (delta 0)
To https://github.com/zcirovic/srpkalendar.git
 * [new branch]      master -> master
Branch 'master' set up to track remote branch 'master' from
'origin'.
```

Nakon ovog koraka modul je na mreži i biće dostupan drugim učesnicima projekta. Sada sledi registracija modula.

Registar npmjs

Javni npm registar je baza JavaScript paketa. Svaki se sastoji od koda i meta podataka.

Razvojni programeri otvorenog koda (eng. *open source*) kao i programeri u kompanijama, koriste npm registar za dodavanje paketa koji su na raspolaganju celoj zajednici i njihovim organizacijama, a takođe i preuzimaju pakete za upotrebu u sopstvenim projektima.

Jednom kada se paket objavi, on je dostupan celoj zajednici i može se pretraživati koristeći `search` komandu. Takođe, primenom komande `npm install` paket se instalira u sopstveno okruženje.

Registar je namenjen samo registrovanim korisnicima. Registrovani korisnici mogu objavljivati a zatim i održavati podatke o modulima.

Registracija tj. kreiranje naloga obavlja se na: https://npmjs.org/signup.

Napomena. Modul ostaje na mrežnom repozitorijumu github. U ovom registru modul se registruje i preko njega koristi u budućim projektima.

Podatak o registru modula postavlja se tokom instalacije modula ili naknadno primenom komande:

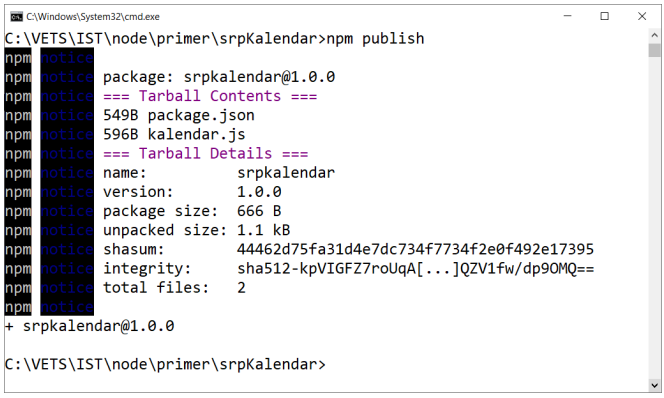
`npm adduser`

Ovom komandom pokreće se dijalog u kome se unose podaci o npm registru:

```
...>npm adduser
Userime: zcirovic
Password: ********
Email: (this IS public) zoran.cirovic@gmail.com
Logged in as zcirovic on https://registry.npmjs.org/.

...>
```

Ovim je postupak registracije mogula završen, sledi poslednji korak tj. sama objava modula. Ona se obavlja primenom komande:

`npm publish`



Slika 6.13. Objava modula

Inače, možete uraditi i testiranje naloga, odnosno proveru dali je nalog unet, koristeći npm cli:

- `npm login` - sledi nizu upita o korisničkom imenu, lozinci, E-mail adresi pre nego budete prijavljeni na npm,
- `npm whoami` - vraća korisničko ime prijavljenog korisnika.

Brisanje registracije

Svaka nova verzija objavljenog paketa se uredno evidentira. Međutim, ipak postoji mogućnost da se neki paket izbriše iz registra. Naravno, bitno je da korisnik koji ga briše ima prava za to. Korisnik takvih prava bora se najpre dodati, ako to već nije slučaj, komandom npm adduser, a zatim se izvršava sledeća komanda:

`npm unpublish modul [--force]` – važno je da vodite računa da će biti uklonjena ona verzija koja je package.json fajlu označene, a ne sve. Takođe, ako navedeni modul koristi drugi modul biranje neće biti moguće sve dok se ne obriše i takav modul.

Upotreba deljenog modula

Došli smo do kraja postupka rada sa mrežnim modulima. Ostaj još samo da se primeni udaljeni modul. Testiranje znači da ćemo udaljeni modul koristiti ravnopravno kao i sve ostale npm module, dakle najpre instalacija pa upotreba u kodu.

`npm init` – kreiranje projekta za testiranje

```
npm install srpkalendar
```

Zatim editujemo js ulazni fajl aplikacije, ako je podrazumevano ime onda je `index.js`. Možemo koristi isti kod koji smo koristili i ranije, na primer:

```
var kalendar = require("srpkalendar");
kalendar.Dani.forEach(element => {
  console.log(element);
});
console.log(kalendar.vikendDani());
let d=[0,3,6];
d.forEach(idx=>{
  console.log(kalendar.dan(idx));
})
. . .
```

Paketi i moduli

Pojam paketa isprepletan je sa pojmom modula i zaista se često mogu koristiti oba pojma. Zato dajemo detaljniji opis ovih pojmova.

Paketi

- Paket je fajl ili direktorijum koji je opisan fajlom package.json. Svaki paket mora sadržati jedan fajl package.json da bi bio objavljen na npm registru.
- Paketi mogu biti označeni da pripadaju jednom korisniku ili organizaciji. Ovakvi paketi, (eng. *scoped packages*), mogu biti privatni ili javni.

Moduli

- Jedan Node modul je fajl ili direktorijum koji se nalazi u folderu **node_modules** a koji može biti učitan koristeći Node.js **require()** funkciju.
- Da bi Node.js **require()** funkcija mogla da učita odgovarajući modul, treba da budu ispunjeno:
  - Folder sa fajlom package.json u kome postoji polje "main".
  - U folderu postoji fajl index.js ili drugi koji odgovara main polju.

Napomena: Ako neki modul ne zahteva fajl package.json onda je to samo modul a ne i paket. U kontekstu Node programa, modul je takođe nešto što se učitava iz fajla. Na primer:

`var req = require('request')` - kaže se da promenljiva `req` referiše zahtevani modul request.

Scouped paketi

Svi paketi imaju naziv. Ipak, samo neki paketi imaju definisan opseg, tj. posebno ime - tzv. *scope*. Kada se koristi u imenima paketa, *scope* se piše tako što se navodi znak `@` ispred naziva, zatim sledi slash `/` pa naziv paketa.

Na primer:

```
@scopeime/packageime
```

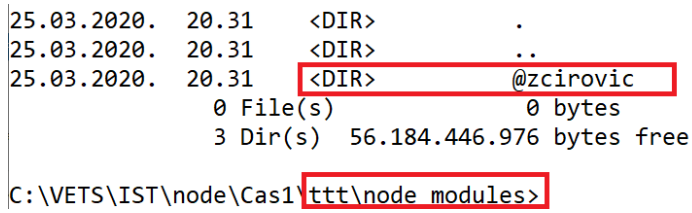
Primenom opsega paketi se grupišu, a istovremeno može da se obezbedi sigurni pristup do njih. Svaki npm user/organization ima sopstveni opseg imena, a samo vlasnik može dodati pakete u sopstveni opseg. Ovo znači da ne treba da brinete u vezi preuzimanja naziva paketa. Paketi koji nisu u opsegu mogu zavisiti od onih koji jesu i obrnuto.

Instalacija

Instalacija se izvodi na način kao i kod običnih modula, na primer

```
npm install @zcirovic/srpkalendar
```

Nakon instalacije u folderu node_modules formira se podfolder `@scopeime` u kome će biti smešteni moduli o kojima je reč.



Slika 6.14. Prikaz modula sa opsegom imena u folderu node_modules

U package.json takođe su posebno označeni ovi moduli.

```
"dependencies": {
   "@zcirovic/srpkalendar": { . . .
}
```

Primena

Ne postoji ništa specijalno za primenu i uključenje ovih modula. Jednostavno se navede:

```
require(@zcirovic/srpkalendar);
```

Ovim se zahteva modul srpkalendar u folderu @zcirovic

Objava

Da bi objavili javni paket u opsegu, mora se dodatno specificirati način pristupa ovim paketima:

```
--access public
```

Napomena: Da bi se obajvio privatni paket u opsegu, mora se u registru koristiti privatni nalog. Više na tu temu pogledajte na adresi: https://docs.npmjs.com/creating-and-publishing-private-packages

U naredna dva poglavlja opisaćemo osnovne karakteristike javnih i privatnih paketa.

Javni Paketi

Kao npm korisnik možete kreirati i objaviti javne pakete koje bilo ko može da preuzima i koristi u svojim projektima.

Unscoped javni paketi postoje u globalnom javnom registru imenskog prostora, a jedan takav paket može biti referenciran u bilo kom fajlu package.json samo sa njegovim imenom: package-ime.

Scoped javni paketi pripadaju jednom korisniku ili organizaciji i moraju biti korišćeni sa prefiksom korisnika kada se koristi kao jedna zavisnost u package.json fajlu:

```
@userime/package-ime
@org-ime/package-ime
```

Privatni paketi

Da bi koristili privatne pakete mora se koristiti npm verzija 2.7.0 ili novija. Ako eventualno posedujete stariju verziju npm-a, potrebno je da se ažurira, odnosno u komandnoj liniji pokrenuti komandu

```
npm install npm@latest -g
```

Koristeći privatne npm pakete, koristi se npm registar za hostovanje koda koji je vidljiv samo vlasniku naloga i odabranim saradnicima, tako omogućavajući upravljanje privatnim kodom, ali istovremeno koristeći i javni kod u projektima.

Privatni paketi uvek imaju neki *scope*, a *scoped* paketi su podrazumevano privatni. *User-scoped* privatni paketi su oni kojima vlasnik i saradnici mogu pristupiti sa pravima čitanja ili čitanja i pisanja. *Org-scoped* privatni paketi su oni kojima pristupaju članovi nekog tima sa pravima čitanja ili čitanja i pisanja.

Publikovanje

Privatni moduli se mogu publikovati samo pomoću plaćenih naloga. Podrazumevano privatni moduli su moduli u opsegu. Ispred naziva modula postavlja se @username, na primer:

```
@username/zcirovic
```

Ovo se, obično, postavlja pri inicijalizaciji projekta dodatkom uz init komandu:

```
npm init --scope=@zcirovic
```

Napomena: Ako je paket privatni, tj. ako u package.json stoji "private": "true“ onda se paket ne može objaviti bez definisanja prava i privilegija!

Za objavu neophodno je najpre korisnika prijaviti za registar. Komanda je već ranije korišćena: npm adduser.

Tek nakon prijave, moguće je uraditi objavu sa definisanim pravima. U našem slučaju to će biti javni pristup:

```
npm publish --access=public
```

Pitanja i zadaci

1. Objasnite svojim rečima šta je Node.js?
2. Koje su osnovne karakteristike Node.js platforme?.
3. Objasniti asinhroni princip?
4. Objasniti i uraditi instalaciju i proveru instalacije. Šta je REPL i koje komande poznajete?
5. Kako se kreira Node.js aplikacija?

6. Kako se dodaju određeni moduli? Pokazati to na konkretnom primeru.
7. Objasniti kako se izvoze tj. eksportuju promenljive i funkcije.
8. Objasniti kako se uvoze funkcije i promenljive iz drugih js fajlova.
9. Šta je definisano fajlom package.json?
10. Kako se definiše a kako se ispituje zavisnost nekom modula od drugih?
11. Objasniti postupak kreiranja deljenih modula?
12. Šta znači registracija modula i zbog čega je ona potrebna? Napraviti nalog na npm registru.
13. Kako se vrši upotreba sopstvenih deljenih modula?
14. Šta su to *scoped* moduli?
15. Kako se publikuju a kako se koriste *scoped* moduli?
16. Napraviti sopstvenu node aplikaciju koja nosi naziv zadatak1
17. Instalirati modul mathjs.
18. Koristeći pomoć sa strane https://www.npmjs.com/package/mathjs napisati js funkciju koja treba da izračuna rezultat 5 različitih matematičkih izraza po sopstvenom izboru. Izrazi mogu biti fiksno definisani.
19. Modul prebaciti na github i u registar npmjs
20. Testirati sopstveni modul.

7. Node.js programiranje

U okviru ovog poglavlja opisane su tehnike programiranja koristeći Node.js platformu. Najpre se prikazuje način na koji se obrađuju događaji u Node.js-u, a zatim se uvodi pojam i prezentuju karakteristike blokirajućih odnosno neblokirajućih funkcija. Uvodi se pojam i rad sa tajmerima i to: sa istekom vremena, neprekidnim i trenutnim.

U nastavku, prezentuju se sopstveni događaji. Primena istih zahteva i upoznavanje sa osluškivačima događaja, pa je osluškivačima posvećeno jedno podpoglavlju. Prikazano je i pridruživanje osluškivača objektu.

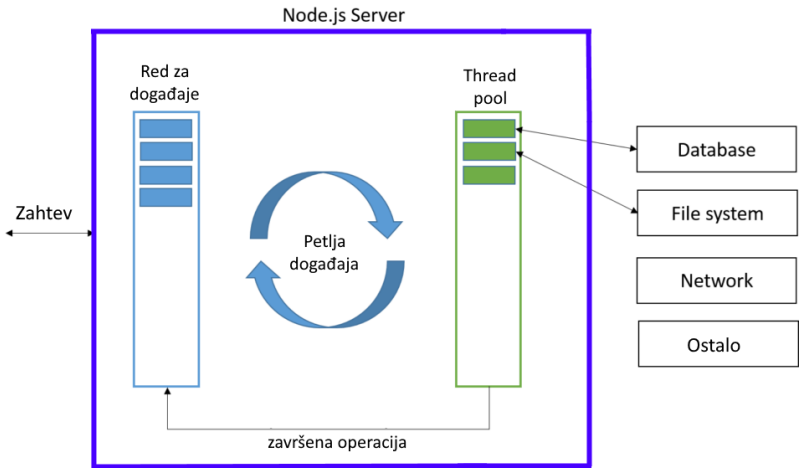
Na kraju, prezentuju se ulazno/izlazne operacije kao i rad sa fajlovima.

Model obrade događaja

Node.js aplikacije se izvršavaju u jednonitnom modelu zasnovanom na događajima. Mada Node.js koristi više niti u pozadini (eng. *thread pool*) sama aplikacija ne poseduje koncepte rada sa više niti. Bez obzira na to, skalabilnost i performanse Node.js aplikacija su ključna karakteristika upravo zahvaljujući modelu događaja i niti.

U standardnim veb serverima tj. veb modelima, zahtev koji se prihvata pridružuje se jednoj niti. Tako se celokupan rad odvija u niti sve dok se zahtev ne razreši i eventualno pošalje odgovor.

Node.js model obrade zahteva/događaja je konceptualno drugačiji. Na svaki zahtev ne događa se kreiranje niti već se zahtev smešta u red za čekanje. U redu za čekanje čeka svoj red na izvršavanje od strane glavne niti tj. njene petlje događaja. Petlja događaja preuzima jedan po jedan događaj i inicira izvršavanje, pogledajte narednu sliku.



Slika 7.1. Model obrade događaja

Ako se poziva funkcija koja može izazvati blokirajući rad, onda se, umesto poziva funkcije direktno, funkcija postavlja u red za čekanje takođe, sa povratnim pozivom koji se izvršava kada se izvršavanje funkcije završi. Kada se svi događaji u redu za čekanje završe Node.js aplikacija sama se završava. Ovo je jako važno za rad u Nodu pa je ovom detalju posvećeno još dodatnih objašnjenja.

Blokirajuće funkcije

Model događaja koji Node.js koristi kao povratne pozive je odlično rešenje sve dok ne se ne naiđe na problem izvršavanja funkcija koje su blokirajuće. Blokirajuće I/O funkcije zaustavljaju izvršavanje tekuće niti sve dok se ne dobije odgovor. Takvi slučajevi su na primer:

- I/O operacije nad fajlovima
- Upiti u bazu podataka
- Pristup veb servisima

Razlog zbog kojeg Node.js ne mora da čeka blokirajuće U/I operacije je što koristi postojanje povratnog poziva tj. *callback* funkcije. Kada događaj koji izaziva blokiranje bude preuzet iz reda događaja, Node.js preuzima nit iz baze raspoloživih niti i pokreće odgovarajuću operaciju u posebnoj niti umesto da njeno izvršavanje bude deo glavne niti. Ovo sprečava blokiranje odnosno zadržavanje preostalih događaja u redu za čekanje.

Model događaja treba imati u vidu kada se projektuju i pišu aplikacije u Nodu.js. To znači da kod treba podeliti na celine koje se mogu izvršavati tako da se postavljaju u red za čekanje, odnosno obradom tih celina iz nekog reda za čekanje.

Dakle, u Node.js aplikacijama kod treba organizovati tako da se celine postavljaju u redove čekanja odnosno primanjem odgovora tj. *callback* funkcija nakon završetka. Na primer, jedan poziv preko reda za čekanje bi bio operacija snimanja podataka u fajl. Drugi bi mogao da bude ostvarivanje konekcije sa serverom. Treći bi bio pokretanje osluškivača za zahteve…

Tajmeri

Node.js poseduje set funkcija za rad sa tajmerima odnosno za rad na odloženo vreme. Sve one koriste petlju događaja, s tim da se neke mogu izvršavati odmah, neke na odloženo vreme, a u nekim slučajevima se to može neprekidno ponavljati.

Tajmer sa istekom vremena

Ovaj tajmer kreira se funkcijom:

```
setTimeout(callback, vreme_ms, [args])
```

Ova komanda postavlja funkciju ***callback*** kao funkciju koja će se izvršiti kada prođe `vreme_ms` milisekundi. Poslednji argument je niz objekata koji se prosleđuje ***callback*** funkciji opciono. Na primer

```
timeoutId = setTimeout(PrikaziVreme, 100);
```

Funkcija `setTimeout()` vraća objekat koji je ID tajmera. Ovaj id se može proslediti funkciji `clearTimeout(timeoutId)` u bilo kom trenutku kako bi se tajmer prekinuo.

Na primer:

```
let brojac = 0;
let t1 = setTimeout(prikaziVreme1000, 1000, brojac);
console.log(t1);
function prikaziVreme1000(dobijenoOdTajmera) {
    console.log("primljeno: " + dobijenoOdTajmera);
    let vreme = new Date();
    vreme.getTime();
    console.log(vreme.getHours() + ":" + vreme.getMinutes() +
    ":" + vreme.getSeconds());
    if (brojac++ < 5)
        t1 = setTimeout(prikaziVreme1000, 1000, brojac);
    else
        clearTimeout(t1);
}
```

Neprekidni tajmeri

Umesto da se sa svakim završenim intervalom kreira novi tajmer, Node.js nudi i **neprekidne tajmere** koji se izvršavaju neprekidno u zadatom intervalu vremena. Ovi tajmeri se kreiraju primenom funkcije:

```
setInterval(callback, vreme_ms, [args])
```

Dakle, nakon ovog poziva, funkcija *callback* izvršava se nakon svakih vreme_ms.

Na primer, prethodni primer bi se jednostavno rešio primenom:

```
setInterval(prikaziVreme1000, 1000);
```

Takođe, funkcija `setInterval()` vraća tajmer objekat. Ovaj objekat se može proslediti funkciji `clearInterval(timerObject)` u bilo kom trenutku kako bi se prekinulo izvršavanje tajmera.

```
timerObject=  setInterval(myFunc, 100000);
```

…

```
clearInterval(myInterval);
```

Na primer:

```
let brojac = 0;
let t1 = setInterval(prikaziVreme1000, 1000, "Vreme:");
console.log(t1);
function prikaziVreme1000(poruka) {
  let vreme = new Date();
  vreme.getTime();

console.log(poruka+vreme.getHours()+":"+vreme.getMinutes()+":"+v
reme.getSeconds());
  if (brojac++ > 10)
     clearInterval(t1);
}
```

Trenutni tajmeri

Node.js poseduje jednu specifičnu vrstu tajmera tzv. **trenutni tajmer** (eng. *immediate timer*).

Ovi tajmeri se koriste da bi izvršilo izvršavanje jedne funkcije čim pre je to moguće, ali sigurno pre svih odloženih ili intervalnih tajmera. Ovaj tajmer omogućava planiranje izvršavanje funkcija nakon tekućih događaja koji su u petlji događaja. Recimo, ove tajmere treba koristiti da bi se funkcije koje se dugo izvršavaju pokrenuli nakon dugotrajnih I/O događaja.

Ovi tajmeri se kreiraju metodom:

`setImmediate(callback,[args])`

Kada se pozove ova metoda, onda se njeno izvršavanje smešta u red za čekanje i callback se izvršava čim pre je moguće, ali preko reda za čekanje, dakle u prvom sledećem ciklusu preuzimanja funkcija.

Ova metoda takođe vraća objekat tajmera. Ovaj objekat se može proslediti da bi se tajmer poništio primenom metode `clearImmediate(objectId).`

Na primer:

```
objectId =  setImmediate(test);
…
clearImmediate(objectId);
```

Sopstveni događaji

U nastavku pogledaćemo kako se kreiraju sopstveni događaji kao i odgovarajući osluškivači. Osluškivači su objekti koji očekuju poruke na događaje.

Događaji se emituju primenom objekta `EventEmitter`. Ovaj objekat je deo modula `events`. U ovom modulu, funkcija

```
emit(eventIme, [args])
```

emituje događaj, `eventIme` a događaju se mogu pridružiti argumenti `args`. Sledeći kod prikazuje primer emitovanja događaja:

```
var events = require('events');
var emitter = new events.EventEmitter();
emitter.emit("nazivSopstvenogDogadjaja");
```

Takođe, jedan događaj se može pridružiti određenom objektu. Na primer:

```
events.EventEmitter.call(this)
```

Ovaj kod dok se izvršava tokom kreiranja instance događaja, pridružuje događaj objektu **this** dodajući **events.EventEmitter.prototype** prototipu vašeg objekta.

Osluškivači

Osluškivače pridružujemo nekom objektu koji očekuje da prihvati emitovani događaj. Ovo se postiže koristeći funkcije emitera:

`.addListener(eventIme,callback)` ili

`.on(eventIme,callback)` ili

`.once(eventIme,callback)`, samo za prvo izvršavanje.

Kada se doda osluškivač, onda svaki put kada se emituje događaj eventIme, izvršavanje callback funkcije se prosleđuje u red za čekanje, a kada dođe na red biće i izvršena.

Osluškivači su važne komponente Node.js programiranja. Istovremeno, oni troše resurse i treba ih koristiti po potrebi. Sada ćemo pokazati i neke funkcije kojima možemo da upravljamo osluškivačima u ovom kontekstu. To su:

- `.listeners(eventIme)` - vraća listu osluškivača događaja eventIme.
- `.setMaxListeners(n)` – ograničava maksimalni broj osluškivača, a ako ih bude više onda izbacuje upozorenje. Podrazumevana vrednost je 10.
- `.removeListener(eventIme,callback)` - uklanja callback za događaj eventIme.

Pogledajmo sledeći primer različitih načina dodavanja za više događaja:

```
const Events = require('events');//pretpostavka da je instaliran
const em = new Events();
//Pretplata 1
em.addListener('event1', function (data) {
    console.log('Evidentiran event1, data=: ' + data);
});
//Pretplata 2
em.on('event2', function (data) {
    console.log('Evidentiran event1, data=: ' + data);
});
//Pretplata 3
em.once('event3', function (data) {
    console.log('Evidentiran event3, data=: ' + data);
});
```

A primena bi mogla da se testira na sledeći način:

```
//Emitovanje više događaja
for (let i = 0; i < 3; i++) {
    em.emit('event1', 'Emitujem prvi događaj.', i);
    em.emit('event2', 'Emitujem drugi događaj.', i);
    em.emit('event3', 'Emitujem treci događaj.', i);
}
```

Dodavanje događaja objektu

Dodavanje tj. pridruživanje događaja za neki objekat izvodi se koristeći prototip objekta. Na primer:

```
function MyObj(){
    // definisanje objekta MyObj
  }
  MyObj.prototype.__proto__ = events.EventEmitter.prototype;
```

Na osnovu ovog može se direktno emitovati iz instance objekta. Na primer:

```
var myObj = new MyObj();
myObj.emit("someEvent");
```

I/O operacije sa fajlovima

Node.js poseduje modul za interakciju sa *file* sistemom. Ovaj modul poseduje standardne API-je za pristup fajlovima koji su dostupni: za otvaranje, čitanje, pisanje.

Za sve pozive file sistema koje ćemo raditi u nastavku, morate da učitate **fs** modul, na primer:

```
var fs  = require('fs');
```

Modul fs omogućava da skoro sve funkcionalnosti budu dostupne u dve varijante: asinhronoj i sinhronoj. Na primer, postoji podrazumevani asinhroni oblik za upis u fajl: `write()` kao i sinhroni `writeSync()`.

Važno je razumeti razliku. Pozivi sinhronog fajl sistema blokiraju rad dok se poziv ne završi, a zatim se kontrola niti vraća. To ima prednosti, ali takođe može izazvati ozbiljne probleme sa performansama. Iz tog razloga, sinhrone pozive trebalo bi izbegavati kad god je to moguće.

Asinhroni pozivi se postavljaju u red događaja koji će se pokrenuti kasnije. Ovo dozvoljava da se pozivi uklope u model događaja Node.js. Osnovne funkcionalnosti i sinhronih i asinhronog fajl sistema su iste. Prihvataju iste parametri s izuzetkom što svi asinhroni pozivi zahtevaju dodatni parametar na kraju tj. funkciju koja se izvršava kada poziv file sistema dovršava.

```
fs.open(path, flags, [mode], callback)
fs.openSync(path, flags, [mode])
```

Kada se pristupa fajlu koristeći URL fajla, uvek se navodi protokol `file` na startu URL-a, na primer:

```
const fs = require('fs');
const fileUrl = new URL('file:///tmp/hello');
fs.readFileSync(fileUrl);
```

Gornji primer pokazuje kako bi se sinhrono pročitali podaci iz fajla. S druge strane, komanda:

```
fs.write(fd, string[, position[, encoding]],
callback)
```

upisuje string u fajl `fd`. Ostali parametri su:

- `position` definiše offset od početka fajla gde počinje upis. Ako se podatak nije naveden ili nije broj, upis se nastavlja od tekuće pozicije.
- `encoding` definiše način zapisa.
- Funkcija `callback` prima argumente `(err,written,string)` gde je `written` broj bajtova koji je upisan. Ovaj broj nije neophodno da bude isti kao broj koji se zahteva.

U okviru ovog modula postoji još veliki broj funkcija za pristup i rad sa fajlovima.

U nastavku dajemo jedan primer u kome se koriste i porede tehnike sinhronog i asinhronog rada sa fajlovima. Komentarima je opisan efekat.

```
//Sinhrono
const fs = require('fs');
const data = fs.readFileSync('./file.txt');
console.log(data);
ispis(); // ispisano posle console.log(data);
//Asinhrono
const fs = require('fs');
fs.readFile('file.txt', (err, data) => {
    if (err) throw err;
    console.log(data);
});
ispis(); // ispisano pre console.log(data);
```

Pitanja i zadaci

1. Objasnite i nacrtajte model obrade događaja u Node.js-u?
2. Šta su to blokirajuće funkcije?
3. Koji tipovi tajmera postoje?
4. Napravite sopstveni primer tajmera sa istekom vremena.
5. Napravite sopstveni primer neprekidnog tajmera.
6. Napravite sopstveni primer trenutnog tajmera.
7. Kako se kreiraju sopstveni događaji?
8. Šta su osluškivači događaja?
9. Koje operacije nad fajlovima poznajete?
10. Koristeći tajmere, napisati dve Node.js aplikacije, tako da jedna upisuje a druga čita iz nekog fajla podatke u određenim vremenskim intervalima. Da li se u ovom primeru mogu koristiti emiteri odnosno osluškivači i na koji način?

8. HTTP servisi

Realizacija HTTP servisa je osnovna tema ovog poglavlja. Naravno, primenom Node.js platforme, kreiranje servisa svodi se na primenu određenih modula namenjenih za to. Zato se u nastavku pre svega radi o http modulu. Ali pre same realizacije servisa, radi jednostavnije obrade podataka koji se razmenjuju, uvode se moduli url odnosno querystring. Nakon objašnjenja ovih modula kreira se prvi server i vrši njegovo testiranje. Na tom primeru radi se parsiranje podataka iz zahteva klijenta. Testiranju se posvećuje naročita pažnja. Urađeno je testiranje koristeći specijalizovane programe, ali i klijentsku aplikaciju kreiranu pomoću Node.js-a.

Posebno je obrađeno opsluživanje statičkih stranica, a posebno REST zahteva koji koriste GET/POST/PUT metode.

Osnove

Primenom odgovarajućih modula, Node.js može da jednostavno i brzo implementira HTTP/HTTPS servera. Ovo je veoma važna karakteristika ovog okruženja.

Zapravo, najpre ćemo prikazati implementaciju zasnovanu na primeni **http** modula, a kasnije, za potpuniju implementaciju veb servera, primenom **express** modula. Zapravo http je modul nižeg nivoa, pruža

veliki stepen fleksibilnosti, ali sav rad koji se tiče nekih tipičnih akcija na strani servera, na primer poput sesija, ostavlja se da programer implementira, dok express ima već pripremljene funkcije.

Imajući u vidu da je rad sa URL podacima neophodan u implementaciji servisa, najpre uvodimo modul koji olakšava rad sa URL podacima.

Objekat *url*

Modul za rad sa URL podacima je `url`, dakle:

```
var url = require('url');
```

Da bi se kreirao URL objekat od stringa, koristi se funkcija `parse`, kojoj se prosleđuje string kao parametar:

```
url.parse(urlStr, [parseQueryStr],
[slashesDenoteHost])
```

-Parametar `parseQueryStr` je tipa `Boolean`. Ako je njegova vrednost `true` onda se vrši parsiranje URL-a. Podrazumevana vrednost je `false`.
-Parametar `slashesDenoteHost` je takođe `Boolean`. Podrazumevana vrednost je `false`. Ako je `true` onda parsira posebno host na osnovu kose crte. Na primer, ako je vrednost string `//host/path`, parsira se u oblik `{host:'host', pathime:'/path'}`
umesto u `{pathime: '//host/path'}`.

Pogledajmo i naredni primer:

```
var url = require('url');
var urlStr =
'http://viser.edu.rs:80/resource/ist/path?query=#abc';
var urlObj = url.parse(urlStr, true, false);
urlString = url.format(urlObj);
```

Korisna karakteristika url modula je rešavanja segmenata URL-a na način kao što pretraživač radi. Ovo omogućava rad sa URL-om na strani servera čuvajući njegove funkcije. Na primer, moguće je promeniti URL pre obrade zahteva na primer zbog obrade resursa ili promene parametara.

Druga korisna funkcija ovog modula je razrešavanje URL-a u odnosu na baznu lokaciju. Osnovna sintaksa ove funkcije je:

`url.resolve(from, to)` gde je:
`from` – parametar koji definiše string osnovnog URL-a.
`to` – parametar koji određuje novi URL.

Sledeći kod prikazuje ilustraciju primera rešavanja URL adrese na novu lokaciju:

```
const url = require('url');
var testUrl1 = "/seg1/seg2/seg3/poslednji";
var testUrl2 = "http://viser.edu.rs:80";
var testUrl3 = "http://viser.edu.rs:80/resource/ist/path?query=#
abc";

console.log(url.resolve(testUrl1, 'seg4'));
//seg1/seg2/seg3/seg4

console.log(url.resolve(testUrl2, '/predmeti'));
//http://viser.edu.rs:80/predmeti

console.log(url.resolve(testUrl3, '/predmeti'));
//http://viser.edu.rs:80/predmeti
```

Objekat *querystring*

HTTP zahtevi često uključuju podatke koji su deo URL-a ili su u okviru same HTTP poruke, u telu ili zaglavlju. Niz parametara upita mogu se izdvojiti od definisanog URL objekta kao što je već prikazano. Podaci poslati preko

zahteva neke forme mogu se očitati preko tela **request** objekta klijenta, kao što će biti opisano kasnije.

Parametri koji su prosleđeni podaci, predstavljaju parove ključ - vrednost. Rad sa parametrima na Node.js veb serveru vrši pretvaranjem u JavaScript objekat pomoću metode `parse()`:

`querystring.parse(str, [sep], [eq], [options])`, gde je

- `str` je zadati upit,
- `sep` je znak separatora koji se koristi. Podrazumevano to je **&**.
- `eq` definiše znak koji se koristi za pridruživanje vrednosti u upitu. Podrazumevano to je `=`.
- `options` je objekat opcija. Na primer, maxKeys definiše maksimalni broj parametara. Podrazumevano je 1000. Ako je 0, nema ograničenja.

```
var qstring = require('querystring');
console.log('kreiranje objekta iz stringa')
var urlParametri="ime=Perica&Prezime=P&color=gray&color=red";
var params = qstring.parse(urlParametri);

console.log("url parametri");
console.log(urlParametri);
console.log("querystring objekat");
console.log(params);

console.log('\n','\n');
console.log('kreiranje stringa iz objekta')
params={
    ime:"Nemanja",
    Prezime:"Petrovic",
    color:["green","yellow"]
};
var qs_obj = qstring.stringify(params/*, '&', '='*/);
console.log("querystring objekat");
console.log(params);
console.log("url parametri");
console.log(qs_obj);
```

Pokretanjem dobija se odziv:

```
kreiranje objekta iz stringa
url parametri
ime=Perica&Prezime=P&color=gray&color=red
querystring objekat
[Object: null prototype] {
  ime: 'Perica',
  Prezime: 'P',
  color: [ 'gray', 'red' ]
}


kreiranje stringa iz objekta
querystring objekat
```

```
{ ime: 'Nemanja', Prezime: 'Petrovic', color: [ 'green',
'yellow' ] }
url parametri
ime=Nemanja&Prezime=Petrovic&color=green&color=yellow
```

Kreiranje servera

Opsluživanje klijentskih zahteva, odnosno prijem `request` zahteva i odgovor na tu poruku tj. slanje `response` poruke obavlja se kreiranjem serverskog objekta `http.Server`. Funkcija za kreiranje je:

```
var server = http.createServer([options][,reqList]);
```

Ova funkcija kreira objekat koji je server, istovremeno definiše rukovaoca događajem na primljeni zahtev od klijenta. U stvari, objekt server koji je vratila funkcija `createServer` je `EventEmitter`, a ono što ovde imamo je samo skraćenica za kreiranje serverskog objekta i kasnije dodavanje slušaoca. Osluškivanje se definiše pozivom metode `listen()` na server objektu. Parametri su:

- `options` – opcije (insecureHTTPParser, maxHeaderSize,...)
- `reqList` – funkcija koja se dodaje za request događaj.

Zahtevi se osluškuju na određenom portu, startovanjem osluškivača:

```
server.listen(port);
```

Evo i prvog primera u kome se kreira server i pokreće osluškivač na portu 80.

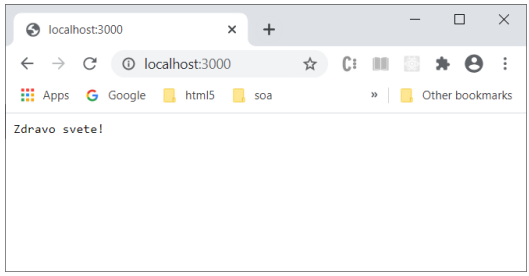
```
var http = require('http');
// kreiranje jednostavnog servera
var server = http.createServer(function (req, res) {
    console.log("primljeno");
    res.write('Zdravo svete!');
    res.end(); // kraj za odgovor - obavezno!
}).listen(3000); //osluskivanje na portu 80
```

Najpre proverimo šta radi ovaj kod, tj. startujmo server i tekućeg foldera preko konzolnog prozora unoseći komandu: **node server1**. Nakon pokretanja servera, osluškivač ostaje u aktivnom stanju, a konzolni prozor i dalje otvoren.

Testiranje

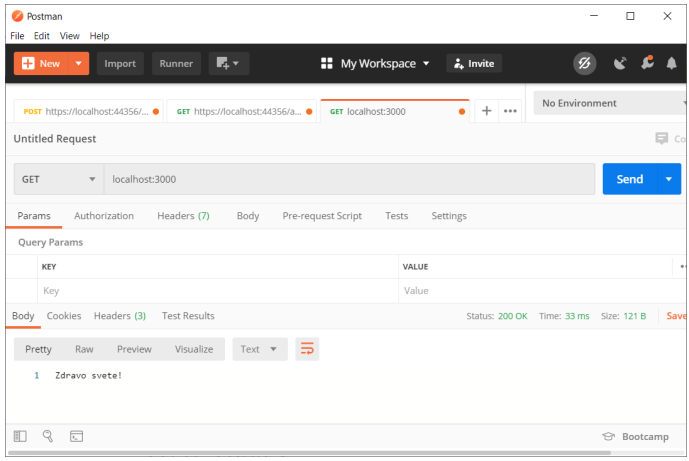
Sada se može uputiti neki zahtev serveru. Za početak pogledaćemo dva načina, a kasnije objasnićemo i način kreiranja zahteva preko samog Node.js-a.

- Prvi način je koristeći neki veb čitač, odnosno unosom adrese servera u adresbar veb čitača. Pogledajte sliku:



Slika 7.1. Slanje zahteva serveru preko veb čitača

- Drugi način je primenom nekog od alata koji mogu poslužiti za testiranje zahteva. Na primer: *Postman*, *Fiddler*, *Insomnia*,... Na narednoj slici je prikaza zahtev i odgovor u *Postman* programu.



Slika 7.2. Zahtev i odgovor koristeći *Postman*

- Treći način bio bi primenom nekih Node.js modula, kao što su request odnosno axios. O primeni ovih modula biće reči u nastavku.

Parsiranje podataka iz zahteva

Najpre ćemo pogledati više detalja o samom objektu zahteva. Obradom podataka koji se šalju putem ovog objekta, server može da razume zahtev klijenta i da odgovori na adekvatan način.

Najpre pogledajmo standardni način za izdvajanje delova zahteva na sledećem primeru:

```
const metod1 = req.method; // get
const url1 = req.url;  // '/?ime=Perica&prezime=P'
const headers1 = req.headers;

const { method, url } = req;
const { headers } = req;
const userAgent = headers['user-agent'];
```

Oba načina primene su identična i pokazuju da je req složen objekat i da koristeći svojstva ovog objekta možemo lako dobiti neke parametre kao što su: metod, parametre uz zahtev, zaglavlje,...

Međutim, za dalje parsiranje parametara url svojstva obično se koristi `url` modul, koji smo već objašnjavali ranije. Tako na primer, ako su prosleđeni parametri uz adresu još:

Primenom koda:

```
var url = require('url');
. . .
var q = url.parse(req.url, true).query;
var txt = "Parametri:"
if(q.ime == "Perica")
    txt += q.ime + " " + q.prezime;
```

Dobija se odziv:

```
Parametri:Perica P
```

Node.js klijentski modul

U nastavku prikazaćemo dva primera kreiranja http klijentskih aplikacija uz pomoć dva Node.js modula, koji mogu da upute zahteve serveru sa određenim parametrima.

Node.js klijentske aplikacije mogu da posluže kao deo aplikacija za testiranje servera ili kao samostalne aplikacije za rad i prikaz podataka.

Prvi način je primenom funkcije `request` ili ugrađenog omotača za `http` modul koji je `request`. Ovaj modul kreira objekat čiji parametri predstavljaju parametre http request zahteva. Dakle, prvi parametar je url adresa zahteva, drugi je definisanje tipa poruke, a treći je povratna funkcija odgovora koja sadrži kao parametar podatke.

```
const request = require('request');

request('http://localhost:3000', { json: true }, (err, res, data
) => {
  if (err) { return console.log(err); }
  console.log(data);
});
```

Drugi način je primenom modula `axios`. Recimo odmah da postoje i drugi modulu, takođe široko korišćeni koji bi mogli da budu primenjeni, ali da ćemo se zaustaviti sa nabrajanjem na primeni ovog modula. Radi se o jednom *Promise based* klijentu baš kao što je i sam Node.js. Evo načina primera primene ovog modula:

```
const axios = require('axios');

axios.get('http://localhost:3000')
   .then(response => {
      console.log(response.data);
   })
   .catch(error => {
      console.log(error);
});
```

Objašnjenje: axios objekat ima prilagođenu metodu **get** za čitanje get zahteva sa određene adrese. Asinhrono, bez blokiranja, ali nakon pročitane poruke vrši se dalja obrada metodom **then** ili hvatanje izuzetka –**catch**. U našem slučaju se samo vrši ispisivanje poruke u konzolu.

Nakon početnog kreiranja servera, pokretanja istog, zatim testiranja rada korišćenjem drugih programa ili samog Node.js-a, u nastavku ćemo detaljnije da pogledamo rad na strani servera i dalje opsluživanje zahteva.

Opsluživanje statičkih stranica

Najosnovniji tip servera je onaj koji opslužuje statičke fajlove. Server se pokreće i osluškuje određeni port. Na zahtev klijenta, server otvara lokalni fajl koristeći **fs** modul, a sadržaj fajla ispisuje kao odgovor klijentu.

Rad na serverskoj strani se sastoji od dva osnovna koraka. U okviru rukovaoca događajem koristi se `url.parse()` metod kako bi se izdvojila putanja koja je jedan od atributa, a zatim, koristeći tu putanju i `fs.readFile()`, čita se sadržaj fajla, koji se zatim vraća preko odgovora tj. objekta `res` koristeći metodu `res.end(data)`.

```
var fs = require('fs');
var http = require('http');
var url = require('url');
var PATH = "www/"; // lokacija do stat. fajlova na serv. strani
http.createServer(function (req, res) {
    var urlObj = url.parse(req.url, true, false);
    fs.readFile(PATH + urlObj.pathime, function (err, data) {
        if (err) {
            res.writeHead(404);
            res.end(JSON.stringify(err));
            return;
        }
        res.writeHead(200);
        res.end(data);
    });
}).listen(80);
```

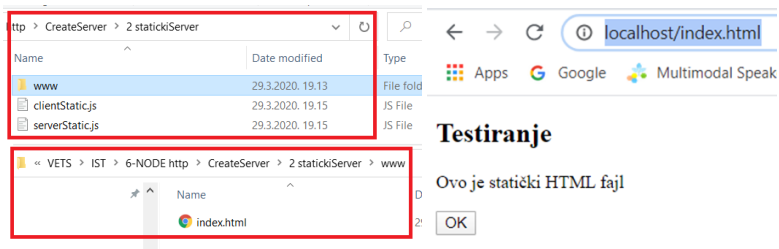
Objašnjenje: Iz zahteva klijenta izdvaja se url parametar - `req.url`, i on se parsira koristeći `url` modul. Među podacima koji se dobijaju ovim parsiranjem nalazi se `pathname` koji predstavlja deo url adrese zahteva koji sadrži putanju da statičkog fajla. Dakle, `pathname` svojstvo se koristi zajedno sa tekućom lokacijom servera `PATH` za čitanje statičkih fajlova.

Pročitani podaci se šalju preko komande `res.end(data)` uz poruku o uspešnom čitanju prosleđenoj kroz zaglavlje poruke, `res.writeHead(200)`.

U Node.js postoji `__dirime` promenljiva koja pokazuje na folder u kome se tekuća skripta izvršava. Tako na primer ako se `__dirime` koristi u fajlu `/scripts/booking/main.js` onda ima vrednost `/scripts/booking`. Suprotno ovome, ako se koristi samo tačka `.` dobija se uvek folder iz koda je pokrenuta aplikacija komandom `node`.

Testiranje

Nakon pokretanja ovog koda lako se može uraditi testiranje. Pretpostavimo da je na serverskoj strani fajl index.html na lokaciji u folderu www koji je na istom mestu gde i sam server, kao na narednoj slici:



Slika 7.3. Lokacije fajlova i testiranje iz veb čitača

Testiranje se vrši jednostavnim unosom odgovarajuće adrese tj. pozicije statičkog fajla na serveru, u našem slučaju to je: localhost/index.html

Ovaj poziv vraća stranicu veb čitaču i on je prikazuje klijentu.

Takođe, testiranje možemo uraditi i koristeći http request funkciju (ili ranije pominjani omotač nalik samostalnom modulu).

```
var http = require('http');
var options = {
    hostime: 'localhost',
    port: '80',
    path: '/index.html'
};
function handleResponse(response) {
    var serverData = '';
    response.on('data', function (chunk) {
        serverData += chunk;
    });
    response.on('end', function () {
        console.log(serverData);
    });
}
http.request(options, function (response) {
    handleResponse(response);
}).end();
```

Objašnjenje: Prvo zapazite da se koristi jedino **http** modul. Drugo, opcije za **request** upit definisane su posebno promenljivom **options**. Dalje, u ovom primeru pokazano je kako se piše kod tako da povratna funkcija ne bude u telu funkcije **request** – to je funkcija **handleResponse**, čiji je jedini argument objekat **response**. Ovaj objekat postavlja osluškivače na događaje 'data' odnosno 'end'. Događaj **data** se emituje kada se detektuje prijem novih podataka - **chunk**, a događaj **end** kada su primljeni poslednji podaci. Nakon primljenih svih podataka vrši se njihovo ispisivanje u konzoli. Dakle, pokretanjem ovog programa videćete na konzoli sadržaj index.html fajla.

Isti način „prikupljanja“ *chunk* podataka posredstvom događaja 'data' odnosno 'end', može da se realizuje i na serverskoj strani. U nastavku pogledajmo opšti slučaj prijema podatak, a zatim izdvajanje podatak po tipu zahteva GET/POST/PUT/….

POST/PUT servisi

Kada se prima POST ili PUT zahtev, HTTP telo poruke sadrži potrebne podatke. Na serverskoj strani se vrši izdvajanje podataka iz tela poruke. Objekat **request** implementira interfejs **ReadableStream**. Ovaj tok se može čitati na bilo kom mestu kao i bilo koji drugi tok. Podatke možemo izvući odmah iz toka slušajući događaje „**data**“ i „**end**“. Realizacija je slična realizaciji do sada. Zapravo, lako je napraviti server koji prihvata i POST i GET i druge vrste metoda. Za POST tip zahteve važno je da se implementira čitanje sadržaja poslatog tela poruke i obradu istog. Pošto se podaci obrade onda se dinamički vraćaju klijentu i zahtev se završava sa **end**(). Pogledajmo jednu jednostavnu realizaciju gde su funkcije izdvojene kako bi se lakše razumemo kod.

```
const http = require('http');
const server = http.createServer(); // kreiranje servera

// dodavanje osluškivača na request
server.on("request",(req,res)=>{
    let buffer = [];
    // za POST zahtev podakci se prikupljaju i obrađuju
    if(req.method == 'POST'){
        req.on('data',(chunk)=>{
            buffer.push(chunk);
        }).on('end',()=>{
            console.log("Primljeno:" + buffer.toString());
            res.writeHead(200,{'content-Type':'json'})
            res.end('{"status":"uspesno primljeno!"}')
        })
    }
})

const port = 3000;
const host = '127.0.0.1'; //127.0.0.1 ili localhost
server.listen(port, host);
console.log(`Server na adresi: http://${host}:${port}`);
```

Objašnjenje: Nakon kreiranog objekta **server** postavlja se osluškivač na **request** zahtev. U kodu samo osluškivača, dakle kada se obrađuje zahtev, vrši se najpre provera tipa zahteva (POST/PUT/GET/DELETE/...) a za određeni i obrada koja počinje dodavanje dva osluškivača za prijem poruka, 'data' i 'end'. Njih smo već objasnili.

Testiranje

Testiranje napisanog servisa vršimo na više načina. Najpre ćemo pogledati html klijenta koji se može otvoriti u nekom veb čitaču i preko njega poslati poruku. Za ovo smo kreirali HTML stranicu sa sledećim kodom:

```
<form action="http://localhost:3000" method="post">
    <label for="ime">
        Ime <br>
        <input type="text" ime="ime"> <br>
    </label>
    <label for="zanimanje">
        Profesija <br>
        <input type="text" ime="zanimanje"> <br>
    </label>
    <input type="submit" value="Submit" />
    </form>
</form>
```

Objašnjenje: HTML forma ima atribut action kojim se definiše url adresa servera koji prima podatke od forme. Način slanja podataka definisan je atributom method. Standarno, podaci od forme se šalju u vidu parova ime=vrednost razdvojenih znakom &. Tako, ako se za formu na narednoj slici unesu podaci u polje *ime* „Pera“, a u polje *zanimanje* vrednost „pekar“:

Slika 7.4. Forma za testiranje servisa

Pritiskom na dugme Submit podaci se primaju na serverskoj strani i dobija se odgovor: `Primljeno:ime=Pera&zanimanje=pekar`.Drugi način, koji se takođe radi preko HTML forme može biti primenom JavaScript koda. Razlika je što se pritiskom na dugme aktivira metoda koja pokupi podatke iz kontrola za unos koje su u formi i onda se slanje vrši u toj metodi. Pogledajmo ovaj kod:

```
<form . . .
      . . .
</form>
<script>
   function senddata(){
      // slanje JS post zahteva
      var xhr = new XMLHttpRequest();
      xhr.open("POST", 'http://localhost:3000', true);
      xhr.setRequestHeader("Content-Type", "text/html");
      var ime=document.getElementById("ime").value;
      var zanimanje=document.getElementById("zanimanje").value;
      var poruka = "ime="+ime+"&zanimanje="+zanimanje;
      console.log("saljem " + poruka);
      xhr.send(poruka);
   }

</script>
```

Napomena: U ovom slučaju, baš kao što je u primeru i urađeno, moguće je promeniti podatke i nazive podataka pre samog slanja. Dakle, fleksibilnost je nešto veća, ali sa druge strane postoji dodatno opterećenje većim kodom.

Drugi način je primenom Node.js-a. Na sličan način kao i ranije, registracijom odgovarajućih događaja objekta zahteva. Na primer:

```
var http = require('http');
var options = {
  host: '127.0.0.1',
  port: '3000',
  method: 'POST'
};
function readResponse (response) {
  var data = '';
  response.on('data', function (chunk) {
    data += chunk;
  });
  response.on('end', function () {
    console.log("Primljeno: " +data);
  });
}
var req = http.request(options, readResponse);
//slanje teksta
req.write('ime=Jova&zanimanje=pesnik');
req.end();
```

Elementi poruke

U ovom poglavlju razmotrićemo posebno zaglavlje poruke kao i tip. Do sada smo ih spominjali I bili su deo svih poruka u razmeni, tako da ćemo u ovom poglavlju reći nešto više o ovim delovima poruke.

Zaglavlje

Zaglavlje http poruke podešava se koristeći metodu **setHeader**

```
response.setHeader('Content-Type',
'application/json');
```

```
response.setHeader('X-Powered-By', 'bacon');
```

Kada se postavlja zaglavlje odgovora, treba imati u vidu da je sadržaj neosetljiv na veličinu slova. Takođe, ako se postavlja više puta, poslednja vrednost je vrednost koja se šalje.

Ako se heder defniše eksplicitno, onda se koristi metoda **writeHead**, preko koje se ispisuje **status** i **zaglavlje** toka.

```
response.writeHead(200, {
    'Content-Type': 'application/json',
    'X-Powered-By': 'bacon'
});
```

Tip poruke

Podatak u zaglavlju poruke, **Content-Type**, koji definiše tip sadržaja koristi se za označavanje vrste medija tj. resursa. Vrsta medija je niz koji se šalje zajedno sa datotekom koja označava format datoteke. Na primer, za fajl slike tip medija je **image / png** ili **image / jpg**, itd.

Kao deo odgovora, govori klijentu o vrsti vraćenog sadržaja. Veb čitač se upoznaje sa vrstom sadržaja koji mora da učita na uređaj. Svaki put kada veb čitač prihvati podatke prihvata i ovaj podatak koji čini deo sadržaja zaglavlja. Tada pregledač izvršava nešto pod nazivom MIME njuškanje, tj. pregledaće tok koji prima i potom shodno tome učitati podatke.

Za ulogu zadatih tipova važna su dva glavna MIME tipa:

- **text/plain** - je vrednost za tekstualne fajlove.
- **application/octet-stream** - je vrednost za sve ostale slučajeve. Nepoznati tip datoteke treba da koristiti ovaj tip.

Status poruke

Uz svaku poruku šalje se i status koji je bliže određuje. Statusi poruke su:

- 2XX - označavaju uspeh

- 3XX - označavaju redirekciju ili informacije o keširanju
- 4XX - označavaju greške na strani klijenta, gde je najpoznatija je upotreba kodova 401,403 i 404
- 5XX - označava greške koje su se desile na strani servera

Moguća je i upotreba posebnog modula za statuse poruke. Modul specijalizovan za rad sa statusima poruka je **http-status-codes**. Na primer:

```
httpStatus = require("http-status-codes");
response.writeHead(httpStatus.OK, {
    "Content-Type": "text/html"
});
let responseMessage = "<h1>Hello, Universe!</h1>";
response.write(responseMessage);
response.end();
```

Pitanja i zadaci

1. Objasni ulogu `url` modula.
2. Objasni ulogu `query` string modula.
3. Koja je imena `http` modula?
4. Napisati jedan jednostavan server `http` modula.
5. Kao se obavlja parsiranje prosleđenih zahteva pomoću `http` modula?
6. Kreirati server koji će raditi sa statičkim html stranicama. Koju metodu zahteva ovi serveri obrađuju?
7. Kako se opslužuju GET zahtevi pomoću `http` modula? Objasni na sopstvenom primeru.
8. Kako se opslužuju POST zahtevi pomoću `http` modula? Objasni na sopstvenom primeru.

9. Kako se opslužuju PUT zahtevi pomoću `http` modula? Objasni na sopstvenom primeru.
10. Opisati testiranja GET servisa na sopstvenom primeru.
11. Opisati testiranja POST servisa na sopstvenom primeru.
12. Opisati testiranja PUT servisa na sopstvenom primeru.
13. Šta je zaglavlje poruke i kako se može definisati, a kako ispitati?
14. Kako se definiše tip poruke odnosno `Content-Type`?
15. Koji status je predviđen za grešku a koji za uspešan prenos?

9. Express modul

Na osnovu statistike preuzimanja i upotrebe svih paketa, **express** modul je svakako u samom vrhu. Ovaj modul je sastavni deo poznatih veb platformi. Njegova jednostavnost, brzina, veliki broj dodataka ali i pouzdanost omogućili su da ovaj modul danas bude veoma popularan među programerima.

U ovom poglavlju prikazujemo postupak instalacije, zatim i kreiranje prve aplikacije. Uvodi se i generator veb aplikacija zasnovan na ovom modulu. Posebna pažnja posvećuje se tehnici rutiranja, prenosu podataka preko parametara rute ali i preko tela poruke.

U nastavku se implementiraju GET/POST servisi koristeći ovaj modul, a takođe i jedna realizacija veb aplikacije koristeći MVC arhitekturu.

Uvod

Rad baziran na http modulu predstavlja bazični rad na nižem nivou, koji pruža puno mogućnosti, fleksibilnosti, ali i poteškoća. Za potrebe razvoja servisa u realnim uslovima i na standardan način, pokazalo se pogodnijim korišćenje modula višeg nivoa. Razlog je izbegavanje ponavljanja pisanja koda.

`Express` modul je razvijen da bi olakšao pisanje veb orijentisanih aplikacija i servisa, pre svega na standardan način, ostavljajući mogućnost i za specifične realizacije.

Osnovna filozofija na kojoj se zasniva rad primenom express modula je definisanje **ruta** (url putanja) za određene resurse i operacija koje se mogu izvršiti nad tim resursom. Ovo se postiže postavljanjem posredničkih funkcija za određene resurs.

Prva aplikacija

Najpre se vrši instalacija modula. Preporučena je lokalna instalacija, dakle:

```
npm install express
```

Express modul se uključuje u aplikaciju:

```
var express = require('express');
```

A zatim se kreira aplikacija:

```
var app = express();
```

Pre upotrebe treba razmotriti potrebu za podešavanjem servera. Express objekat daje mogućnost podešavanja koje se izvodi metodama:

```
set(setting, value),
enable(setting),
disable(setting).
```

Na primer:

```
app.enabled('trust proxy');  // true
app.disabled('strict routing'); // true
app.get('view engine'); //  npr. pug
```

Generator

Kreiranje kostura početnog servera i aplikacije ubrzava se koristeći express generatora. Naravno, da bi se koristio express generator, potrebno je najpre uraditi instalaciju

`npm install -g express-generator.`

Više informacija o komandama express modula može se pogledati komandom `express –h`:

Slika. 10.1. Skraćeni prikaz express komandi

Nakon instalacije generatora lako se kreira početna Node.js serverska aplikacija na sledeći način:

`express testServer` – generiše folder i sve fajlove i foldere aplikacije. Zatim se instaliraju ostali paketi. Postupak je sledeći:

`cd testServer`

`npm install` – postoji package.json na osnovu koga se vrši instalacija ostalih paketa.

`npm start` – pokretanje servera.

Ako pogledate fajl package.json videćete da je definisana skripta „start“:
`"scripts": { "start": "node ./bin/www"}`
Dakle, pozivom ove skripte „`npm start`“ komandom, pokreće se skripta `"/bin/www.js"` koja pokreće server uključuje `app.js` koja sadrži svu logiku generatora.
Zapaža se nekoliko kreiranih foldera:

- public skladišti statička dokumenata: slike, js, css, koji se koriste,
- routes, se koristi za implementaciju REST API –ja,
- view se koristi za HTML šablone. Express.js ima podršku za veliki broj šabloniranih pogleda. Neki od njih su pug (nekada jade), haml.js, ejs, react, swig, … Šabloni olakšavaju kreiranje pogleda i njihovo povezivanje sa podacima.

Nešto više o nazivima ovih foldera, dato je u polgavlju koje se tiče MVC arhitekture.
Pokretanje klijenta i uvid u startovan server ostvaruje se pokretanjem nekog veb čitača na adresi http://localhost:3000/.

Rutiranje

Rutiranje definiše način kako aplikacija odgovara na zahteve klijenata ka određenim krajnjim tačkama tj. putanjama kao i specifične HTTP zahteve (GET, POST, itd.).
Svaka ruta može imati jednu ili više funkcija rukovaoca koje se izvršavaju kada se detektuje neki prijem na određenoj ruti. Rutiranje se definiše sledećim funkcijama objekta aplikacije app:
`app.`*`metoda`*`(putanja, [posrednik], callback)`, gde je:

- `app` – referenca `Express` objekta,
- `metoda` - jedna od REST metoda,

- `putanja` - URL ruta
- `callback` - funkcija sa `request` i `respond` objektima, slično kao kod `http` modula.

Uz aplikaciju se koriste određena podešavanja koja se izvode na sledeće načine:

- `app.use(ime,value)` – podešavanje promenljivih okruženja, ili
- `app.use([path],callback)` – definisanje posredničke funkcije koja će rukovati HTTP zahtevima upućenim serveru

Primeri rutiranja

Osnovni skup funkcija koje prihvataju zahteve na određenom url i određenog tipa date su u narednom primeru.

```
app.get('/', function (req, res) {
    res.send('Get zahtev');
})
app.post('/', function (req, res) {
    res.send('POST zahtev');
})
app.put('/korisnik', function (req, res) {
    res.send('PUT zahtev na putanji /knjige');
})
app.delete('/korisnik', function (req, res) {
    res.send('DELETE zahtev /knjige');
})
```

Objašnjenje: Kao što se vidi, svaka http metoda (get, post, put, delete) ima odgovarajuću metodu za app objekata, tako da se koristi metoda koja odgovara tipu metoda koji se obrađuje. Prvi argument svake od metoda je putanja tj. ruta koja je specifična za ove metode. Ostala obrada je zasnovana na povratnoj funkciji i argumentima zahteva i odgovora.

Dakle, u praksi se rutama dodaju informacije od značaja. Na primer, id objekta koji se traži u bazi može biti deo rute. Ova tehnika prosleđivanja parametara dobro je poznata, na primer:

- `?kljuc=vrednost` - a navodi se nakon URL-a.
- Upiti se automatski obraduju i smeštaju kao objekat u svojstvo **query** objekta **Request**. Dakle, koristi se **request.query** objekat za rad sa podacima iz upita. Podaci su deo ovog objekta, dakle, predstavljaju svojstva sa kojima se dalje radi. Na primer:

```
app.get('/korisnik', function(request, response){
  console.log(request.query.ime);
  console.log(request.query.zanimanje);
  response.send('ok');
});
```

Express.js omogućava da se rute navode sa parametrima. Ovi parametri predstavljaju još jedan način slanja podataka kroz zahtev. Parametri rute imaju dvotačku (:) ispred parametra i mogu se navesti bilo gde u putanji. Naredni kod prikazuje upotrebu rute sa parametrima.

```
// http://localhost:3000/korisnik/Jova/pesnik/33
app.get("/korisnik/:ime/:zanimanje/:id",(request , response )=>{
    let send=" Detalji stranica "+
    "<br > ime: "+ request.params.ime +
    "<br > zanimanje:"+ request.params.zanimanje +
    "<br > id: "+ request.params.id;
    response .send(send);
    });
```

Dakle, ruta u gornjem primeru očekuje zahtev praćen /**korisnik**/ uz nastavak koji ima više parametara ime, zanimanje i id, baš ovim redosledom. Na ovaj način se podaci o svim parametrima koji se prosleđuju, server automatizovano prepoznaje kao očekivano svojstvo **params** objekta.

Primeri servisa

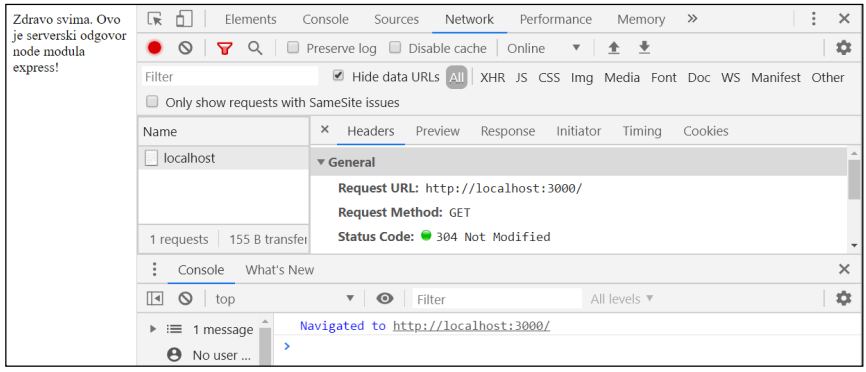
Na potpuno dosledan način kreiraju se različiti tipovi servisa. U nastavku dajemo nekoliko konkretnih realizacija različitih primera.

Get

```
var express = require('express');
var app = express();

app.get('/', function(request, response){
    response.send('Zdravo svima. Ovo je serverski odgovor node m
odula express!');
});

app.listen(3000);
```

Get zahtev se očekuje na osnovnoj adresi servera na portu 3000. Primljen Get zahtev obrađuje se i vraća tekstualna poruka.

Ako se testiranje uradi koristeći veb čitač, koristeći razvojne alatke uz veb čitač (F12) moguće je pogledati detalje u vezi http poruka, pogledati sliku.



Slika 8.1. Razvojni alat veb čitača za praćenje poruka

Post

Pošto se u razmeni post poruka podaci šalju kroz telo http poruke, za realizaciju post servisa obično se koristi dodatni modul za obradu tela poruke: **body-parser**.

Do verzije 4. ovaj modul je bio integrisan u Node.js, a sada je neophodno uraditi instalaciju ovog posredničkog programa. Ovaj modul parsira JSON, bafer, string i url podatke koji se šalju preko HTTP POST zahteva. Ovo se izvodi dodavanje posredničke funkcije za parsiranje:

```
//podržava parsiranje podataka tipa application/json
app.use(bodyParser.json());
//podržava parsiranje application/x-www-form-urlencoded
app.use(bodyParser.urlencoded({ extended: true }));
```

Evo konkretnog primera:

```
var express = require("express");
var bodyParser = require("body-parser");
var app = express();
app.use(bodyParser.urlencoded({ extended: false }));
app.use(bodyParser.json());
app.post('/data', function (req, res) {
    console.log(JSON.stringify(req.body));
    var ime = req.body.ime;
    var zanimanje = req.body.zanimanje;
    console.log("ime = " + ime + ", zanimanje " + zanimanje);
    res.end("ok");
});
app.listen(3000, function () {
    console.log("Server pokrenun na PORTU 3000");
})
```

Request objekat

Request objekat sadrži informacije koje se tiču HTTP zahteva od klijenta. Express nudi sledeća svojstva:

`req.query` – sadrži upit (eng. *query string*) prosleđen sa zahtevom. Parametrima zahteva se pristupa preko imena parametra, npr: req.query.ime, req.query.zanimanje,

`req.params` – sadrži parametre rutiranja koji su imenovani, na primer podaci jedinstvene id vrednosti,

`req.body` – parsirane podatke tela HTTP poruke,

`req.path`, `req.host`, i `req.ip` – dodatne informacije o klijentu,

`req.cookies` – parsiranje kolačića.

Response objekat

Response objekat služi kao objekat odgovora koji se upućuje klijentu od servera.

`res.status(code)` – postavlja statusni kod odgovora,

`res.set(field,[value])` – podešavanje HTTP polja zaglavlja poruke,

`res.cookie(ime,value,[options])` – postavlja kolačić,

`res.redirect([status],url)` – redirekcija,

`res.send([body|status],[body])` – vraća odgovor. Podešavajući automatizovano tip i dužinu odgovora (Content-Type, Content-Length),

`res.json([status|body],[body])` – isti kao i res.send() s tim što se forsira slanje objekata u JSON formatu

`res.render(view,[locals],callback)` – generiše pogled i vraća HTML.

Primer http zahteva

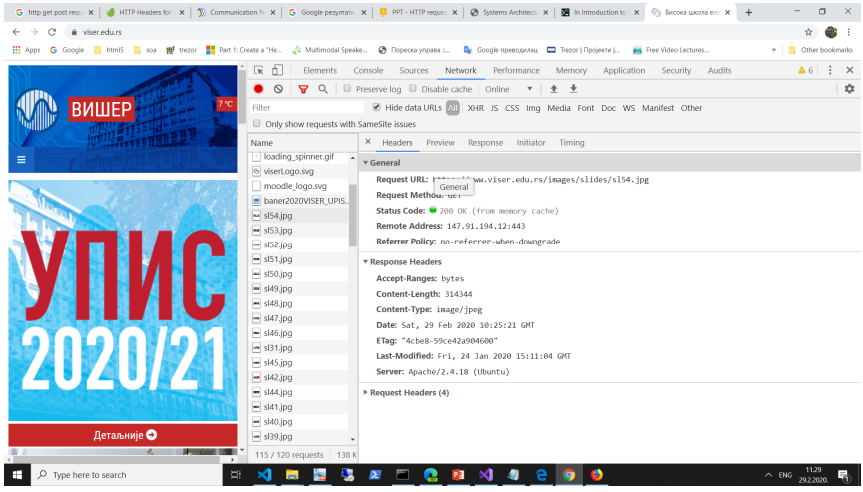
```
GET /tutorials/primeri/ HTTP/1.1
Host: net.istprimeri.com
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 6.1; en-US;
rv:1.9.1.5) Gecko/20091102 Firefox/3.5.5 (.NET CLR 3.5.30729)
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
Cookie: PHPSESSID=r2t5uvjq435r4q7ib3vtdjq120
Pragma: no-cache
Cache-Control: no-cache
```

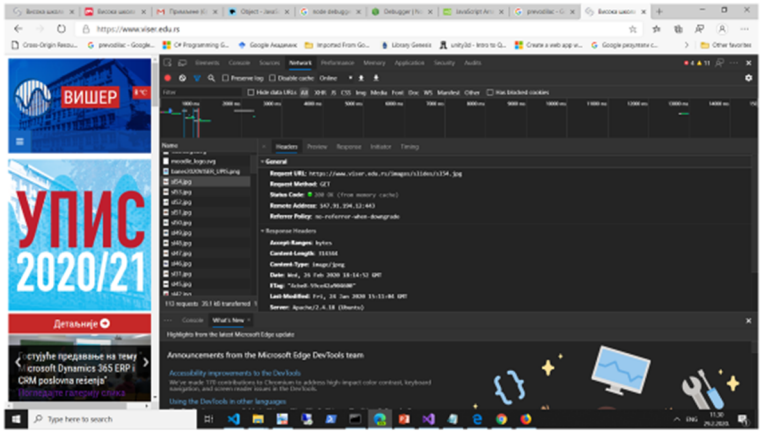
Primer http odgovora

```
HTTP/1.x 200 OK
Transfer-Encoding: chunked
Date: Sat, 28 Nov 2009 04:36:25 GMT
Server: LiteSpeed
```

```
Connection: close
X-Powered-By: W3 Total Cache/0.8
Pragma: public
Expires: Sat, 28 Nov 2009 05:36:25 GMT
Etag: "pub1259380237;gz"
Cache-Control: max-age=3600, public
Content-Type: text/html; charset=UTF-8
Last-Modified: Sat, 28 Nov 2009 03:50:37 GMT
X-Pingback: https://istprimeri.com/test.php
Content-Encoding: gzip
Vary: Accept-Encoding, Cookie, User-Agent

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-
8" />
<title>IST primeri…</title>
<!-- ... rest of the html ... -->
```

Kako pogledati HTTP zaglavlje - Chrome





Slika 8.2. Prikaz detalja HTTP poruka preko Chrome veb čitača

MVC arhitektura

Express.js omogućava rad jedne aplikacije zasnovan na odvojenim funkcijama tj. modulima. Jedna od najčešće primenjivanih arhitektura danas je MVC (eng. *Model View Controller*) arhitektura. Ona omogućava razvoj i održavanje velikih aplikacija.

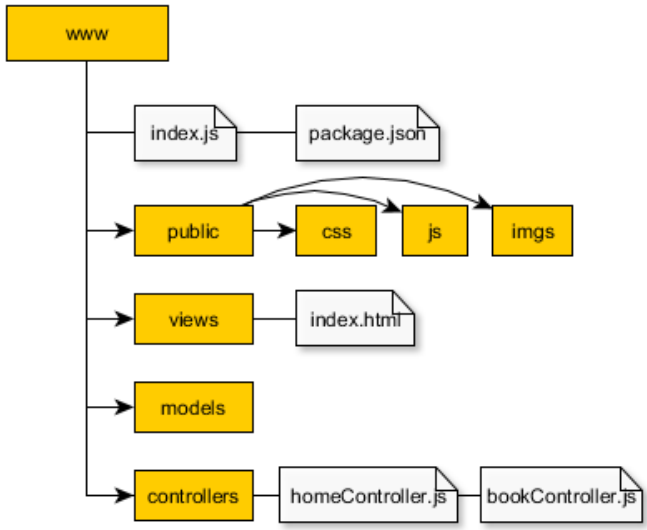
MCV arhitektura se fokusira na tri osnovna funkcionalna dela aplikacije: modele, poglede i kontrolere. Od ovih delova je formiran i naziv arhitekture.

Modeli – Klase koje predstavljaju podatke u aplikaciji odnosno bazi podataka. Objekte modela koriste kontroleri u radu sa podacima. Ovi podaci mogu biti prilagođeni pogledu ili mogu odgovarati više sirovoj logičkoj obradi podataka u bazi.

Pogledi – Neposredni prikaz podataka u aplikaciji, obično se generišu od strane kontrolera tj. na zahtev korisnika.

Kontroleri – Čine vezu modela i podataka. Kontroleri su zaduženi za izvršavanje logičkih operacija na zahtev klijenata. Osluškuju zahteve, parsiraju podatke, a zatim i formiraju odgovor, obično kao novi pogled formiran na osnovu modela i prosleđenih podataka.

Prateći MVC šablon, povratne funkcije se pomeraju na primenu određenih modula koji zaduženi za određene funkcije. Tako na primer, povratne funkcije za prijavu korisnika na sistem, brisanje, izmene smeštaju se u poseban fajl accountController.js, smešten u folder gde su i ostali kontroleri. Pogledajmo jedan kompleksniji primer:



Slika 8.3. Struktura servera zasnovana na MVC arhitekturi

Ovu arhitekturu automatizovano kreira komanda za skraćeno generisanje express aplikacija, naravno uz neke manje razlike.

Postojanje kontrolera obezbeđuje jasno razdvajanje funkcije za obradu zahteva po osnovu određene logike, pre svega na osnovu url parametara. Na primer, na strani servera dodajemo modul i obradu:

```
// var app = express();
const osobaController= require("./controllers/osobaController");
. . .
//http://localhost:3000/osoba?ime=Jova&zanimanje=pesnik&id=12
app.get("/:osoba",osobaController.sendReqParam);
```

A na strani kontrolera definiše se korišćena funkcija:

```
exports.sendReqParam = (req, res) => {
  let send= "<br > osoba: "+ req.params.osoba +
            "<br > ime: "+ req.query.ime +
            "<br > zanimanje:"+ req.query.zanimanje +
            "<br > id: "+ req.query.id;
   res.send(`Poslati parametri ${send}`);
}
```

Pitanja i zadaci

1. Objasni ulogu **express** modula.
2. Napisati elementarnu aplikaciju pomoću **express** modula i uporedite je sa aplikacijom napisanom pomoću **http** modula.
3. Šta je generator koji se povezuje sa **express** modulom? Testirati šta se kreira njegovom primenom.
4. Kako se kreira servis za Get zahteve?
5. Kako se kreira servis za Post zahteve?
6. Kako se pristupa elementima **request** odnosno **response** objekta? Koje elemente poseduje ovi objekti?
7. Šta je MVC arhitektura? Šta predstavlja model, pogled odnosno kontroler?

8. Kako se piše MVC aplikacija zasnovana na `express` modulu? Napisati jedan primer.

10. Pristup podacima

Ovo poglavlje bavi se tehnikama pristupa podataka, odnosno integracija Node.js aplikacija i aktuelnih baza podataka. Node.js poseduje module prilagođene za pristup podacima bez obzira o kojoj se bazi radi.

Imajući u vidu različitost korišćenih baza, najpre je dat prikaz instalacije i upotrebe tri baze. Veoma korisna u različitim namenama, brza i pouzdana, a takođe i najčešći izbor u Node.js aplikacijama jeste mongo baza. U nastavku poglavlja, bavimo se primenom ove baze u našim aplikacijama. Nakon instalacije baze odnosno Node modula, pomoću primera pokazano je: kreiranje baze, kreiranje i brisanje kolekcija, snimanje u kolekciju, pronalaženje podata, ažuriranje podataka i na kraju brisanje.

Uvod

Rad sa podacima jedna je od najvažnijih karakteristika servisa. Podaci se čuvaju u bazama podataka za koje su zaduženi posebni serveri. Na kraju prethodnog poglavlja videli smo da Node.js ima jaku podršku za sve aktuelne baze.

Rad sa podacima treba da obezbedi servisima brz, dakle bez zastoja, pristup do podataka, i u slučaju kada opslužuju istovremeno veliki broj zahteva. Takođe, podaci treba da budu zapisani u obliku koji najviše odgovara potrebama aplikacija koje servisi opslužuju.

U ranijem periodu, najčešće korišćena organizacija podataka u bazama zasnovana je na tabelama i vezama između njih. To su tzv. relacione baze podatka. Ipak, kao odgovor na sve veću potrebu za specifičnom organizacijom podataka a pre svega za brz odgovor servera baze čak i u slučajevima kada postoji veliki skup podata razvile su se tzv. nosql baze podataka.

Integracija baza podataka

Node.js poseduje ogroman broj modula raznih imena, pa tako i module koji predstavljaju drajvere za rad sa određenim bazama podataka. U ovom poglavlju biće predstavljen rad sa nekoliko baza podataka.

MongoDB

Modul koji se koristi je mongodb. Instalacija se izvodi komandom:

```
npm install mongodb
```

```
var Client = require('mongodb').MongoClient
Client.connect('mongodb://localhost:12345/osobe',
  { useNewUrlParser: true, useUnifiedTopology: true },
  function (err, db) {
     db.collection('zaposleni').find().toArray(
       function (err, result) { console.log(result) }
     )
})
```

Više reči o radu sa mongodb bazom biće kasnije.

MySql

Modul koji se koristi je mysql, Instalacija se izvodi komandom:

```
npm install mysql
```

```
var mysql = require('mysql')
var connection = mysql.createConnection({
  host: '127.16.2.1',  user: 'admin',
  password: '******',  database: 'dbTest'
})
connection.connect()
connection.query('* AS solution', function (err, rows, fields) {
  console.log(rows[0].solution)
})
// * sql upit
connection.end()
```

Sql Server

Modul koji se koristi je tedious, Instalacija se izvodi komandom:

```
npm install tedious
```

```
var Connection = require('tedious').Connection
var Request = require('tedious').Request

var config = { server: '127.16.1.1',
  authentication: {
    type: 'default',
    options: {
      userIme: 'your_userime', password: 'your_password'
    }
  }
}

var connection = new Connection(config)

connection.on('connect', function (err) {
  if (err) { console.log(err) } else {
    executeStatement()
  }
})

function executeStatement () {
  request = new Request("*", function (err, cnt) {
    if (err) { console.log(err) } else {
      console.log(cnt + ' rows')
    }
    connection.close()
  })
  // * sql upit
  request.on('row', function (columns) {
    columns.forEach(function (column) {
      if (column.value === null) { console.log('NULL')
      } else {
        console.log(column.value)
      }
    })
  })

  connection.execSql(request)
}
```

Osim navedenih drajvera baza, na raspolaganju stoje sve značajnije baze podataka, https://expressjs.com/en/guide/database-integration.html.

MongoDB

MongoDB je baza podataka otvorenog koda, koja organizuje podatke koristeći dokumenta. MongoDB dokumenta čuvaju podatke u vidu JSON strukture. To znači da su podaci u bazi oblika ključ-vrednost, odnosno predstavljeni su kao objekti sa svojstvima.

Prednost ovako čuvanih podataka je oblik koji je nalik podacima objekata u JavaScript-u. U pozadini, zbog efikasnosti rade baze, koristi se binarni zapis JSON podataka (BSON). Upotreba ove baze je vodeća među Node.js zajednicom.

Instalacija

Instalacija zavisi od korišćenog operativnog sistema. Za Windows instalacije se izvodi koristeći instalacioni fajl koji se preuzima sa zvaničnog mongodb sajta https://www.mongodb.com (.msi). Naravno, odabrati odgovarajuću verziju (npr. MongoDB Community Server).

Nakon preuzimanja fajla, isti se otvara, nakon čeka biva pokrenut *čarobnjak* kojim se ostvaruje interakcija tokom instalacije tj. postavljaju izvesna podešavanja.

Nakon instalacije kreira se konekcija za rad sa bazom definišući konekcijski string, na primer „mongodb://localhost:27017“

Ako nameravamo da koristim ovaj server za potrebe node.js aplikacija, mora se uraditi i instalacija odgovarajućeg modula koristeći npm:

```
npm install mongodb
```

A zatim se modul može koristi u aplikacijama:

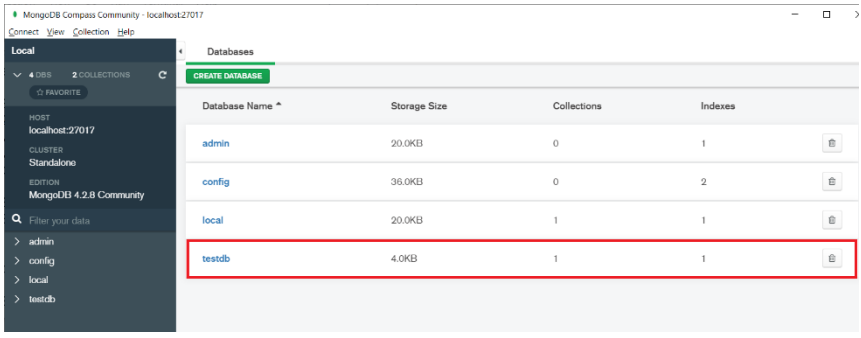
```
var mongo = require('mongodb')
```

Kreiranje baze

Da bi se kreirala baza potrebno je najpre kreirati MongoClient objekat. Nakon toga se baza kreira na određenoj IP adresi i portu koji na dalje predstavljaju mongodb server.

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/testdb";

MongoClient.connect(url, function(err, db) {
  if (err) throw err;
  db.close();
  console.log("Uspešno!");
});
```

Rezultat se može pratiti kroz neki od dodatnih alata za praćenje podataka u bazi, na primer MongoDB Compass Community, kao na slici:



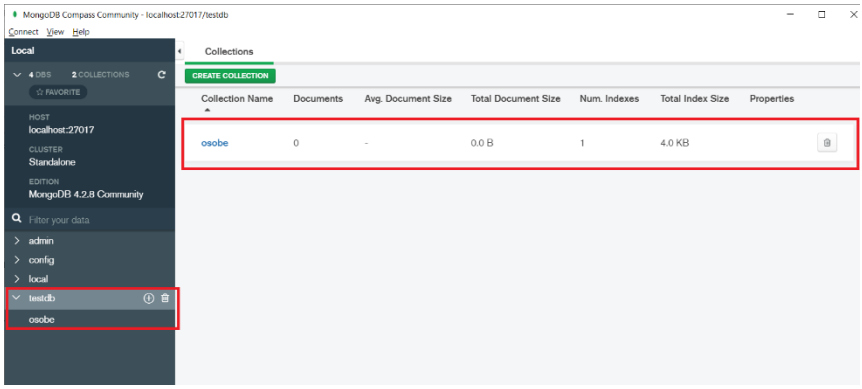
Slika 9.1. Pogled na kreiranu mongo bazu

Kreiranje/brisanje kolekcija

Slično tabelama u relacionim bazama, kolekcije su elementi baze koji su zaduženi za čuvanje podataka. Za kreiranje kolekcija koristi se metoda createCollection(). Pogledajmo primer kako se kreira kolekcija osoba:

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";

MongoClient.connect(url, function(err, db) {
  if (err) throw err;
  var dbo = db.db("testdb");
  dbo.createCollection("osobe", function(err, res) {
    if (err) throw err;
    db.close();
    console.log("Uspešno!");
  });
});
```

Izvršavanje prethodnog primera kreira se jedna kolekcija u mongo bazi. Na slici ispod je prikazan pogled na promene u bazi.



Slika 9.2. Pogled na kreiranu kolekciju baze

Kao što se tabela može izbrisati tako se mogu brisati I kolekcije u MongoDB bazi. Za brisanje kolekcija koristi se metoda `drop()`. Povratna funkcija sadrži objekat greške kao i parametar *result* koji je `true` ako je kolekcija uspešno obrisana, inače ima vrednost false.

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";

MongoClient.connect(url, function(err, db) {
  if (err) throw err;
  var dbo = db.db("testdb");
  dbo.collection("osobe").drop(function(err, result) {
    if (err) throw err;
    db.close();
    if (result) console.log("Uspesno");

  });
});
```

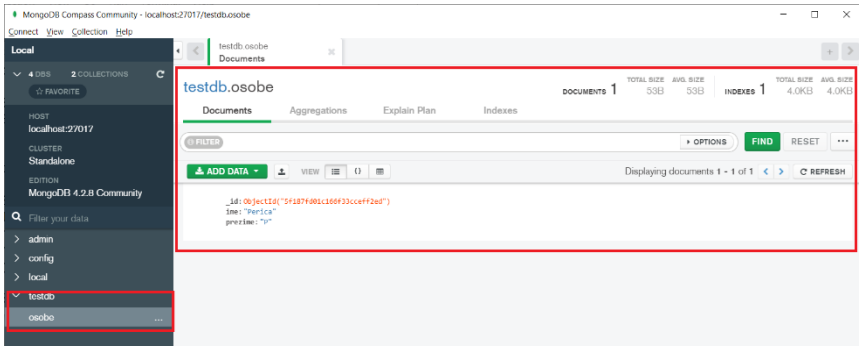
Snimanje u kolekcije

Da bi se neki zapis podataka, dakle neki objekat ili neki JSON zapis dodao u kolekciju koja postoji u MongoDB bazi koristi se metoda insertOne(). Ovaj zapis podataka naziva se dokument u MongoDB bazi a analogan je jednom zapisu u relacionim bazama.

Prvi parametar metode insertOne() je objekat koji sadrži podatke tipa ime-vrednost za svako svojstvo u dokumentu koji se snima.

Metoda ima I povratnu funkciju koju koristim da bi se obradili podaci eventualne greške ili podaci o samom snimanju. Na primer:

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";

MongoClient.connect(url, function(err, db) {
  if (err) throw err;
  var dbo = db.db("testdb");
  var osoba1 = { ime: "Perica", prezime: "P" };
  dbo.collection("osobe").insertOne(osoba1, function(err, res) {
    if (err) throw err;
    db.close();

    console.log("dodat je jedan dokument");
  });
});
```



Slika 9.3. Pogled na kreiran dokument u kolekciji

Napomena: Zapazite da pokušaj snimanja dokumenata u kolekciju koja ne postoji rezultuje kreiranje kolekcije automatski. Takođe, za razliku od sql baza podataka, snimanje podataka osoba sa drugim svojstvima biće moguće.

Pronalaženje podataka

Za pronalaženje podataka, tačnije određenog dokumenta u kolekciji, koriste se metode `find` odnosno `findOne`. Ove komande analogne su *select* naredbama u standardnom sql jeziku.

findOne

Ova metoda vraća prvo pojavljivanje dokumenta u kolekciji na osnovu navedenih uslova.

Prvi parametar je objekat upita, dakle sa vrednostima koje su od značaja. Ako se koristi prazan objekat onda se selektuju svi dokumenti u kolekciji, a u našem slučaju vraća se prvi

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";

MongoClient.connect(url, function(err, db) {
  if (err) throw err;
  var dbo = db.db("testdb");
  dbo.collection("osobe").findOne({'prezime':'J'}, function(err,
 result) {
    if (err) throw err;
    db.close();
    console.log(result.ime + ' ' + result.prezime);
  });
});
```

find

Kada se iz kolekcije izdvaja više dokumenata po osnovu nekog kriterijuma, tj. kada se vrši filtriranje koristi se metoda find().

Prvi argument je objekat upita, dakle objekat koji sadrži svojstva koja se ispituju. Objekat upita sadrži svojstva po kojima se vrši pretraga, a za vrednost može da se koristi regularni izraz.

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/";

MongoClient.connect(url, function(err, db) {
  if (err) throw err;
  var dbo = db.db("testdb");
  //var query = { ime: /^Jov/ }; // pretraga po početku stringa
  var query = { ime: /ca$/ }; // ime se završava sa “ca“
  dbo.collection("osobe").find(query).toArray(function(err, resu
lt) {
    if (err) throw err;
    db.close();
    console.log(result);
  });
});
```

Odgovor:

```
[
 { _id: 5f187fd01c166f33cceff2ed, ime: 'Perica', prezime: 'P' },
 { _id: 5f1882d6d98dff31101f852d, ime: 'Jovica', prezime: 'J' }
]
```

Ažuriranje dokumenta

Ažuriranje jednog dokumenta vrši se koristeći metodu updateOne().

Prvi parametar je upit kojim određuje dokument koji se ažurira. Ukoliko se upitom određuje više dokumenata, samo prvi dokument će biti ažuriran.

Drugi parametar je nova vrednost dokumenta.

Pri definisanju objekta koji se koristi za ažuriranje koristi se svojstvo $set. Ukoliko se ažurira samo neko od svojstava dokumenta u kolekciji, postavlja se vrednost samo za to svojstvo, ne i za ostala.

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://127.0.0.1:27017/";

// postavljanje novog dokumenta
MongoClient.connect(url, function(err, db) {
  if (err) throw err;
  var dbo = db.db("testdb");
  var q = { prezime: "P" };
  var novaOsoba = { $set: {ime: "Perica", prezime: "Petrović" }
};
  dbo.collection("osobe").updateOne(q, novaOsoba, function(err,
res) {
    if (err) throw err;
    db.close();
    console.log("Uspesno");
  });

  // promena jednog svojstva
  q = { prezime: /^P/ };
  var novaVrednost = { $set: { prezime: "Peric" } };
  dbo.collection("osobe").updateOne(q, novaVrednost, function(er
r, res) {
    if (err) throw err;
    db.close();
    console.log(JSON.stringify(res)); // res objekat

    console.log("Uspesno");
  });
});
```

Ukoliko je potrebno ažurirati sve dokumente koji zadovoljavaju kriterijum pretrage, da se koristi metoda updateMany().

Obe metode, updateOne() i updateMany(), vraćaju rezultujući objekat koji sadrži informacije u vezi ažuriranja u bazi. U gornjem slučaju taj objekat je:

```
{"result":{"n":1,"nModified":1,"ok":1},"connection":{"id":0,"hos
t":"127.0.0.1","port":27017},"modifiedCount":1,"upsertedId":null
,"upsertedCount":0,"matchedCount":1,"n":1,"nModified":1,"ok":1}
```

Objekat result povratne vrednosti res je:

```
{"n":1,"nModified":1,"ok":1}
```

Ovaj objekat sadrži broj izmena u bazi, na primer:

```
console.log(res.result.nModified);
```

Brisanje dokumenta

Da bi se izbrisao jedan dokument u nekoj kolekciji koristi se metod `deleteOne()`.

Prvi parametar ove metode je objekat upita po kome se definiše objekat koji se briše. Ukoliko upit referiše više od jednog dokumenta, samo jedan tj. prvi od njih biće izbrisan.

```
var myquery = { Prezime: 'P' };
dbo.collection("osobe").deleteOne(myquery, function(err, obj) {
    if (err) throw err;
    db.close();
    console.log("obrisan 1 dokument");
});
```

Za brisanje više zapisa koristi se metoda deleteMany(). Metoda je slična prethodnoj, prvi argument je objekat upita, samo se vrši brisanje svih dokumenata u kolekciji koji zadovoljavaju upit.

```
var q = { ime: /ca$/ };
dbo.collection("osobe").deleteMany(q, function(err, obj) {
if (err) throw err;
db.close();
console.log("obrisano: " + obj.result.n + " documenata.");
});
```

Združivanje kolekcija

Iako se radi o nerelacionoj bazi podataka, po nekada je bitno da se između kolekcija može uraditi združivanje podataka. Ovo združivanje odgovara levom spoljnjem *join-u* u sql-u. Pogledajmo sledeći kod:

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://127.0.0.1:27017/";

MongoClient.connect(url, function(err, db) {
  if (err) throw err;
  var dbo = db.db("testdb");
  dbo.collection('studenti').aggregate([
    { $lookup:
       {
         from: 'osobe',
         localField: 'id_osoba',
         foreignField: 'id',
         as: 'osobadetalji'
       }
     }
    ]).toArray(function(err, res) {
    if (err) throw err;
    db.close();
    console.log(JSON.stringify(res));

  });
});
```

U gornjem primeru, na tekuću kolekciju studenti, pridružuje se kolekcija osobe. Postupak se sastoji u tome da se svakom dokumentu u kolekciji

studenti dodaju dokumenti iz kolekcije **osobe** po osnovu kriterijuma koji je definisan funkcijom aggregate preko objekta $lookup.

Limit

MongoDB je nosql baza, veoma brza čak i kada radi sa veoma velikim kolekcijama. Ipak, potrebno ograničiti broj podataka koji se preuzima od baze, jer je postupak preuzimanja i eventualnog prikaza znatno sporiji i tiče se klijentske aplikacije.

Metoda za definisanje maksimalnog broja vraćenih dokumenata je **`limit()`**. Primena je jednostavna i podseća na sql naredbu top. Na primer:

```
dbo.collection("osobe").find().limit(2).toArray(function(err, re
sult) {
    if (err) throw err;
    db.close();
    console.log(result);
});
```

Pitanja i zadaci

1. Objasni postupak instalacije modula za baze podataka mysql, sql server i mongodb.
2. Instalirati Mongo bazu.
3. Koristeći Node.js modul mongodb kreirati bazu po vašem izboru.
4. Za prethodno kreiranu bazu kreirati Node.js aplikaciju koja kreira dve kolekcije.

5. Napisati Node.js aplikaciju koja će za dve kolekcije kreirati po tri dokumenta.
6. Kako se vrši pretraga a kako ažuriranje dokumenata u bazi?
7. Napisati Node.js aplikaciju koja će pronaći jedan zapis iz jedne kolekcije i uraditi izmenu zapisa.
8. Napisati Node.js aplikaciju za brisanje određenih dokumenata.
9. Obezbediti združivanje dve kolekcije, pa napisati Node.js aplikaciju za njihovo združivanje.

11. Uvod u *mongoose*

U ovom poglavlju prikazaćemo jedan objektno-orijentisan pristup za rad sa podacima u Node.js okruženju. Za tu svrhu koristićemo Mongoose modul. Ovaj modul se koristi kao jedan dodatni sloj između aplikacione logike i baze podataka. Najpre se prezentuju osnovne karakteristike modula, prednosti i nedostatke, a zatim posebne karakteristike najvažnijih objekata modula: Schema, Model, Document, Nakon instrukcija instalacije, u nastavku se prezentuje praktičan rad sa objektima kroz rad na kreiranju prvog modela. Najpre se radi definisanje modela tj. šeme uz obavezno uvođenje tipova i ograničenja, zatim se daju osnove o virtuelnim svojstvima kao i metode modela, kreiranju konekcije i na samom kraju pokazuje se kreiranje svih CRUD metode modela.

Karakteristike

Mongoose omogućava konverziju aplikacionih podataka u odnosu na strukturu modela. To je paket za objektno-dokumentno mapiranje -ODM. Ovaj paket omogućava pokretanje MongoDB komandi na način koji čuva objektno-orijentisan način pisanja koda.

Tako na primer, za razliku od čistog MongoDB pristupa, Mongoose može da prati i koristi šemu podataka koje čuva. Na ovaj način Mongoose pruža

mogućnost rada sa modelima u formi JavaScript klasa koje Mongoose koristi da organizuje upite.

Ključne prednosti su:

- Može da se kreira jedna šema za dokumenta. Dakle, model može da bude validiran pre korišćenja.
- Aplikacioni podaci mogu da se prilagode objektnom modelu.
- Poslovna logika može da se poveže sa srednjim slojem.
- Mongoose je lakši za upotrebu od MongoDB paketa.

Naravno, postoje i neki nedostaci, kao na primer:

- Zahteva se korišćenje šema podataka, što nije uvek najbolja opcija pošto MongoDB ne zahteva šeme.
- Ne poseduje neke operacije kao što su čuvanje podataka, za razliku od ugrađenih tj. nativnih drajvera.

Objekti

Mongoose proširuje MongoDB nativni drajver uvodeći dodatne objekte kojima nudi posebne funkcije. Takvi objekti su:

Schema – definiše struktuiranu šemu za dokumente u kolekciji. Ovaj objekat omogućava definisanje polja i tipova koje uključuje, jedinstvenost, indekse kao i validaciju. Koristeći šema objekat obezbeđuje se tačno definisana struktura dokumenata u kolekciji.

Model – predstavlja objektnu reprezentaciju dokumenata u kolekciji.

Document – omogućava funkcionalnost neophodnu za realizaciju ODM i validaciju.

Osim ovih objekata, mongoose kreira omotač za standardne funkcije korišćene za upite i spajanje dokumenata u nove objekte: Query i `Aggregate`.

Instalacija

Instalacija se izvodi pokretanjem komande u folderu projekta:

```
npm install mongoose
```

Inače, ako se koristi paket Mongoose nije neophodno koristiti **mongodb** paket.

Kreiranje modela

Modeli se definišu pomoću šema objekata. Šeme omogućavaju da se definišu polja koja su deo svakog dokumenta, i tako omogućavaju njihovu validaciju i podrazumevane vrednosti. Uz sve to, moguće je definisanje statičkih odnosno pomoćnih metoda koje olakšavaju rad sa tipovima podataka, kao i virtuelna svojstva koja se mogu koristiti kao i druga svojstva, ali koja se ne čuvaju u bazi podataka.

Šeme se prevode u modele koristeći metod **mongoose.model()**. Kada se jedan model kreira, pomoću njega možete kreirati, brisati, ažurirati i nalaziti objekte tog tipa.

Svaki model se mapira ka jednoj kolekciji dokumenata u MongoDB bazi. Ovi dokumenti sadrže polja šeme koja je definisana modelom Schema.

Definisanje modela

Definisanje modela vrši se na osnovu definisane scheme. Dakle, najpre se kreira šema pa model. U narednom primeru pokazujemo kako se to radi na jednom jednostavnom primeru:

```
var mongoose = require('mongoose');

var Schema = mongoose.Schema;

var radnikSchema = new Schema({
  ime: String,
  datumZaposlenja: Date
});

// prevodjenje scheme u model
var RadnikModel = mongoose.model('RadnikModel', radnikSchema );
```

U gornjem primeru kreira se kolekcija 'RadnikModel'. Dakle, prvi argument je naziv kolekcije, a drugi je šema po osnovu koje se to radi. Povratna vrednost je objekat same kolekcije.

Tipovi podataka

Za svako polje u jednoj šemi definiše se određeni tip podataka. Tipovi su:

- String;
- Number;
- Boolean ili Bool;
- Array;
- Buffer;
- Date;
- ObjectId;
- Mixed;
- Schema;
- Decimal128;
- Map.

Većina tipova je jasna. Evo objašnjenja za nekoliko novih:

ObjectId: Predstavlja identifikator za jednu instancu u bazi. Tačnije, predstavlja jedinstveni ID za specifičan objekat.

Mixed – proizvoljni šema tip.

[] – niz stavki. Nad ovim tipom se mogu koristiti operacije poput: push, pop,…

Pogledajmo jedan primer koji ilustruje navedene tipove, a još pri tome uključuje neka ograničenja:

```
var testSch = new Schema({
    tacno:  Boolean,
    naziv:    String,
    binarno:  Buffer,
    godineStarosti:{ type: Number, min: 18, max: 65 },
    kreirano: { type: Date, default: Date.now },
    nekiSadrzaj:   Schema.Types.Mixed,
    Id: Schema.Types.ObjectId,
    plata: Schema.Types.Decimal128,
    niz: [],
    nizStringova: [String],
    nizBrojeva: [Number],
    nizDatuma: [Date],
    nizNizova: [[]],
    nizNizovaBrojeva: [[Number]],
    mapa: Map,
    mapaStringova: {
      type: Map,
      of: String
    }
  })
```

Dodatna ograničenja su:

- Podrazumevana vrednost - **`default`**;
- Neki ugrađeni validator kao na primer min/max ili neki prilagođen potrebi;
- Oznaka da li je polje obavezno - **`required`**;
- Ako je tip **`string`**, oznaka koja govori da li se postavlja na velika - **`uppercase`**, mala slova - **`lowercase`** i da li se vrši odsecanje praznina, na početku ili kraju stringa - **`trim`**.
- Ako je tip **`Number`** ili **`Date`** može se koristiti **`min`** i **`max`**. Za tip **`string`** koriste se **`minlength`** i **`maxlength`** za dužine stringa.

- Za tip `string` i `Number` postoji validator `enum` kojim se definiše lista mogućih vrednosti, npr. `enum:['ponedeljak','utorak']`.

U nastavku dajemo jedan primera sa nekoliko primenjenih validatora. Pogledajte kod pažljivo, a objašnjenja slede nakon koda.

```
var studentSchema = new Schema({
    smer: {
      type: String,
      enum: ['NRT', 'RT', 'IS']
    },
    godine: {
      type: Number,
      min: [15, 'Možda je previše mlad'],
      max: 100,
      required: [true, 'Zašto godine skrivati?']
    },
    napomena: {
      type: String,
      default: 'student',
      lowercase: true,
    }
});
```

Gornja šema sastoji se od tri polja: `smer`, `godine` i `napomena`. Polje smer je tipa `String`, vrednost se postavlja na velika slova, a vrednost je uvek iz skupa navedenih `enum: ['NRT', 'RT', 'IS']`. Sledeće polje – `godine` je polje tipa `Number` u opsegu od `15` do `100`. Ako se unese vrednost ispod minimalne dobija se poruka koja je navedena. Polje je obavezno, pa ako se ne navede takođe se može pročitati odgovarajuća poruka. Poslednje polje je tipa `String` koje ima podrazumevanu vrednost `'student'`, a vrednost se uvek prebacuje u mala slova.

Virtuelna svojstva

Virtuelna svojstva nekog dokumenta su svojstva koja se ne snimaju u bazi, ali se mogu koristiti tj. imaju get i set svojstva. Get metode ovih svojstava tzv. geteri korisni su za dobijanje željenog formata ili za kombinovanje više drugih svojstava. Seteri, na primer, mogu da se koriste da bi jedna vrednost snimila kao više svojstava.

Tipičan primer je rad sa imenima. Ako su svojstva ime i prezime, onda virtuelno svojstvo može biti punoIme. Get metoda za puno ime vraća kombinaciju imena i prezimena a set metoda snima posebno ime a posebno prezime.

Sopstvene metode modela

Jedna šema ima metode za: celu šemu, za instance tj. dokumente i upitne metode. Statičke metode su slične metodama za instance s tim da se ove druge tiču posebnih instanci tj. dokumenata. Upitne metode omogućavaju proširenje i ulančavanje mongoose upita.

Mongoose ima mogućnost ugradnje sopstvenih **metoda po instancama**. Pogledajte sledeći primer:

```
var mongoose = require('mongoose')
var osobaSchema = new mongoose.Schema({
    ime: String,
    prezime: String
});

osobaSchema.methods.napomena = function(arg) {
    return console.log(arg);
}

// kompajliranje
var Osoba = mongoose.model('Osoba', osobaSchema);

var osobe = new Array(3);

// kreiranje dokumenata
osobe[0] = new Osoba({
     ime: 'Aca', prezime: 'Aleksic'
});
osobe[1] = new Osoba({
    ime: 'Bora', prezime: 'Brkic'
});
osobe[2] = new Osoba({
    ime: 'Ceca', prezime: 'Cukic'
});

// sada sve instance imaju metodu napomena, preklopimo prve dve
osobe[0].napomena = function(arg) {
    console.log("1. " + JSON.stringify(arg));
};
osobe[1].napomena = function(arg) {
    console.log("2. " + JSON.stringify(arg));
};

osobe[0].napomena("prva osoba");
osobe[1].napomena("druga osoba");
osobe[2].napomena("bez sopstvene funkcije");
```

U gorenjem primeru napisane su dve metode za prve dve instance koje preklapaju metodu na nivou cele kolekcije. Obe instance su deo iste kolekcije, ali sa dve posebne funkcije.

Osim gore navedenih, postoje i **statičke** metode - metode na nivou šeme. Dva načina za dodavanje ovih metoda su: dodavanje metode svojstvu `schema.statics` odnosno pozivom metode `Schema.static()`.

Pogledajte pažljivo naredni primer:

Konekcije

Povezivanje sa bazom podataka obavlja se pomoću konekcijskog stringa koji je isti koji se koristi i kod mongodb paketa. Komanda za povezivanja je:

```
connect(uri, options, [callback])
```

Ova komanda definiše parametre za vezu i pokreće otvaranje konekcije prema bazi. Otvaranje konekcije je dugotrajna operacija i ona se završava nakon nekog vremena. Zato se koristi povratna funkcija. Međutim, sam objekat konekcije koji se definiše može da se koristi za mongoose operacije, pa i za praćenje otvoren konekcije. Analogno, postoji i `disconnect()` metoda za raskidanje konekcije. Na primer:

```
var mongoose = require('mongoose');
mongoose.connect('mongodb://localhost/osobe', {useNewUrlParser:
true}, err=>{
    if(err!=null)
        console.log(err)
    else
        console.log("Uspesno - 1");

});
const conn = mongoose.connection;
conn.on('error', console.error.bind(console, 'Greska'));
conn.once('open', function() {
  console.log('Uspesno - 2');
});
mongoose.disconnect();
```

Napomena. U gornjem primeru, pri otvaranju konekcije korišćena su podešavanja {useNewUrlParser: true} bez kojih bi prevodilac javio upozorenje.

Primer samo pokazuje uspešno otvaranje konekcije do baze i zatim zatvaranje.

CRUD metode

Pogledajmo sada i osnovne operacije nad modelom koje se tiču kreiranja, brisanja promene i pretrage podataka, koristeći mongoose.

Pretraga

Pretraga se uz podršku objekta `Query`. Ovaj objekat je sastavni deo brojnih drugih metoda. Najjednostavniji način da se uradi kompletna pretraga neke kolekcije uz definisanje uslova pretrage, načina sortiranja rezultata odnosno oblika dobijenih rezultata bio bi:

```
model.find({value:{$lt:11}},{sort:{[['value',-1]]},
fields:{ime:1, title: 1, value: 1}, function(err,
results){});
```

Mongoose omogućava lakšu primenu iste funkcije, zadajući parametre deo po deo, na primer:

```
var query = model.find({});
query.where('value').lt(11);
query.sort('-value');
query.select('ime title value');
query.exec(function(err, results){});
```

Svojstva upita i opcije su podeljene na posebne podmetode jednog query objekta. Na ovaj način se izvršavanje upita, koje se izvodi exec metodom na kraju, razdvaja od prosleđivanja parametara.

Funkcije koje čine model.find() vraćaju Query objekat. Tako da se može isto uraditi i na drugi način:

```
model.find({}).where('value').lt(11).sort('-
value').select('ime title value').exec(function(err,
results){});
```

Insert

Kada se napravi model on ima pripadajuće metode za rad sa podacima. Jedna instanca modela može da se kreira i koristi za rad, a jednostavnom komandom `save` vrši se snimate te instance.

```
conn.once('open', function() {
  console.log('Uspesna konekcija');
  var Osoba = mongoose.model('Osoba', osobaSchema, 'osobe');
  var aca = new Osoba({ id:3,  ime: 'Aca', prezime: 'Aleksic'});
  aca.save(function(err,obj){
    if(err)return console.error(err)
    console.log("uspesno snimljen novi objekat " + obj.ime)
  })
});
```

Kreirani model je `Osoba`. Ovaj model se koristi za kreiranje pojedinih instance, dakle koristi se baš kao klasa. Tako je napravljena instanca ovog modela `aca`, koja je po svojoj prirodi dokument. Ovaj dokument poseduje metodu `save` kojom se vrši snimanje. Naravno, snimanje nije uvek uspešna operacija, a i vremenski traje, pa se rezultat snimanja dobija povratnom funkcijom.

Napomena. U prethodnom primeru kreiran je model za postojeću kolekciju `'osobe'`. Ukoliko se ne bi naveo ovaj argument pri kreiranju modela, model bi bio kreiran za novu kolekciju.

Delete

Brisanje je takođe metoda koja se može izvesti na konkretnom modelu, naravno željene objekte je potrebno izdvojiti. Na primer:

```
conn.once('open', function() {
    console.log('Uspesna konekcija');
    var Osoba = mongoose.model('Osoba', osobaSchema, 'osobe');

    Osoba.remove({id:3},function(err, result){
        if (err) {
            console.err(err);
          } else {
            console.log(result)
          }
    })
});
```

Dakle, model `Osoba` poziva metodu `remove` za brisanje zapisa po osnovu pretrage. Svi nađeni dokumenti koji zadovoljavaju `{id:3}` biće obrisani.

Izmena

Izmena se obavlja metodom updateOne ili updateMany. Postoji I metoda update koja se trenutno smatra zastarelom, pa se umesto nje koriste updateOne i updateMany. Argumenti te metode su: string po kome se vrši pretraga, opcije koje se koriste za metodu i na kraju povratna funkcija.

```
conn.once('open', function() {
    console.log('Uspesna konekcija');

    var Osoba = mongoose.model('Osoba', osobaSchema, 'osobe');

    var query = { ime: 'Jovica' }
    const res = Osoba.updateMany(query, { ime: 'JOVica' }, funct
ion(err, obj){
        if(err) console.error(err)
        console.log('Uspesno izmenjeno: ' + JSON.stringify(obj))
    });
});
```

Izmena se može uraditi i primenom set metoda kao u narednom primeru:

```
conn.once('open', function() {
    console.log('Uspesna konekcija');
    var Osoba = mongoose.model('Osoba', osobaSchema, 'osobe');
        var query = Osoba.findOne().where('ime', 'Jovica');
        query.exec(function(err, obj){
            console.log('Pre snimanja')
            console.log(obj.toJSON())
            obj.set('ime','JOVICA')
            obj.save(function(err,obj){
                console.log('Posle snimanja')
                console.log(obj.toJSON())
            })
        })
});
```

Pitanja i zadaci

1. Koja je osnovna karakterisnika paketa mongoose?
2. Koje su objekti od značaja za rad pomoću mongoose paketa?
3. Objasni objekata Schema.
4. Objasni objekata Document.
5. Objasni objekata Schema.
6. Koja je uloga objeka Query i navedi jedan primer njegove upotrebe?
7. Instalirati mongoose paket.
8. Objasni postupak kreiranja jednog modela pomoću ovog paketa.
9. Kako bi definisali model za rad sa knjigama?
10. Koji sve tipovi podataka postoje?
11. Objasni tipove:

- Array;
- Buffer;

- Date;
- ObjectId;
- Mixed;
- Schema;
- Decimal128;
- Map.

12. Šta su virtuelna svojstva? Napisati jedan primer za model knjige.
13. Kako se kreiraju funkcije modela? Koje vrste takvih funkcija postoje?
14. Objasni posebno svaku od CRUD operacija.
15. Napiši za svaku CRUD operaciju jedan primer za model knjige.

12. SOAP servisi

U ovom poglavlju prezentuje se jedan programski model za kreiranje komunikacije klijent server. Koristeći taj model, kroz praktične primere, objašnjavaju se osnovni objekti u komunikaciji kroz realizaciju zasnovanu na SOAP (eng. ***S**imple **O**bject **A**ccess **P**rotocol*) protokolu.

Nakon uvoda sledi objašnjenje pojma krajnja tačka kao i objašnjene elemenata krajnje tačke: adresa, povezivanje i ugovor. Definišu se vrste servisnih operacija i načini hostovanja servisa. Za opis i lakši rad na servisima uvodi se pojam i način upotrebe WSDL (eng. ***W**eb **S**ervice **D**escription **L**anguage*) standarda.

U nastavku poglavlja, definiše se konfigurisanje servisnih projekata, daje se značenje i način korišćenja najbitnijih sekcija za podešavanja, zatim sledi pregled tipova servisnih operacija, detaljno se bavimo servisnim ugovorima i realizaciji zasnovanoj na interfejsima, uvode se atributi metoda i klasa koji su od značaja za rad servisa. Za prenos podataka definišemo tipove koji se serijalizuju, a zatim uvodimo i ugovor o podacima za definisanje struktura u prenosu. Sva objašnjenja su praćena praktičnim primerima tako da se na kraju dobija kompletan servis sa željenim karakteristikama.

Uvod

U ovom poglavlju objasnićemo rad sa jednim opštim programskim modelom za realizaciju klijent-server komunikacije. Ovaj programski model zasniva se na WCF biblioteci (eng. ***W**indows **C**ommunication **F**oundation*). Neke osnovne karakteristike ove biblioteke su:

- Omogućava potpun pristup distribuiranom programiranju;
- Efikasan način da proizvedemo i upotrebljavamo servise.
- WCF podržava sledeće transportne šeme:
  - HTTP/HTTPS;
  - TCP;
  - IPC;
  - Peer network;
  - MSMQ;
  - Service bus.

Osnovni pojmovi

Mada su neki programski pojmovi koji se koriste u nastavku, sasvim logični, a moguće da su poznati od ranije, njihovo razumevanje je važno za razumevanje koncepata ali i konkretnih primera tj. koda.

Krajnja tačka

Osnovni elementi u programskoj realizaciji klijent-server komunikacije čine **krajnje tačke** (eng. *end point*): adresa (eng. ***A**ddress*), objekti povezivanja (eng. ***B**inding*) i pravila tj. ugovor (eng. ***C**ontract*).

- **Adresa** pristupne tačke na kojoj servis sluša dolazne upite. Predstavlja se u standardnom **Url** formatu.

- **Povezivanje.** Definiše osobine komunikacionog kanala kojim se pristupa servisu. Kanal može da se sastoji od niza povezanih elemenata. Najniži element tog komunikacionog kanala je transportni protokol, koji definiše način na koji se poruke razmenjuju između klijenta i servisa. Standardni format koji se koristi za opis zahteva koje servis postavlja za povezivanja je WS-Policy.
- **Ugovor.** Definiše funkcionalnosti koje pristupna tačka pruža i format poruka koje implementirane funkcije očekuju. Za opis ugovora koristi se standardni format - **WSDL** (eng. ***W****eb* ***S****ervice* ***D****escription* ***L****anguage*)

Vrste operacija

U jednoj klijent-server komunikaciji poruke se najčešće razmenjuju u oba pravca: od servera ka klijentu i obrnuto. Međutim, jedna servisna operacija je kreirana za jednu vrstu komunikacije. U tom smislu, **vrste operacija** za razmenu poruka su:

- Poruke u jednom smeru – eng. *one-way*;
- Poruke koje vraćaju podatak – eng. *request-reply*;
- Istovremena razmena poruka u oba pravca – eng. *duplex*.

Kontejner servisa

Komunikacija klijent-server podrazumeva postojanje krajnje tačke, odnosno procesa osluškivača na serverskoj strani. To znači da treba da postoji neki procesa koji će pokretati i hostovati servis. Program koji hostuje server tj. servis naziva se **kontejner servisa.** Da bi servis mogao da radi tj. da osluškuje zahteve odnosno pozive klijent, mora postojati kontejner za hostovanje. Kod standardnih WS to je veb server, na primer IIS server. Kontejner za WCF **može biti bilo koji proces koji se izvršava**.

WSDL

Servisne operacije mogu da se potpuno standardno opišu pomoću jedne sintakse, i na taj način automatizuje rad odnosno podešavanje na strani klijentskih aplikacija.

Klijentska aplikacija pristupa servisnoj krajnjoj tački kako bi prikupila ABC-du servisa preko jezika WSDL. Servis izlaže ovaj opis servisa, koristeći specijalno projektovanu krajnju tačku tzv. *MEX-end-point* (eng. ***M****etadata* ***Ex****change)*.

Klijent pošto prihvati WSDL kreira *proxy* klasu i *app.config*.

Proxy klasa preslikava potpise operacija krajnje tačke kako bi kod na klijentskoj strani mogao da pozove na odgovarajući način operacije na udaljenoj krajnjoj tački. Tačnije, proxy mora da obezbedi da poruke koje se šalju servisu budu po tačno zadatoj specifikaciji.

Osnove programskog modela

Realizacija jednog servisa podrazumeva realizaciju funkcija na određenoj krajnjoj tački. Jedna funkcija servisa definiše se svojim nazivom kao i parametrima servisa. Kako je komunikacija klijent-server u ovoj arhitekturi zasnovana na XML formatu, onda je jasno da se za opis servisa koristi XSD šema. Za uspešan rad, servis ne treba da klijentu opisuje rad servisne funkcije već naziv i parametre funkcije koja se može pridružiti jednoj krajnjoj tački servisa.

Da bi se kreirao jedan servis, potrebno je da eksplicitno tj. programski definišemo njegove funkcije. Za to se koristi **interfejs**. Interfejs daje opis operacija koje želimo da budu vidljive klijentima.

Primer jednog interfejsa:

```
namespace WCFprvi
{
    [ServiceContract]
    interface ICenovnik
    {
        [OperationContract]
        double cenaArtikla(string artikal);
    }
    . . .
}
```

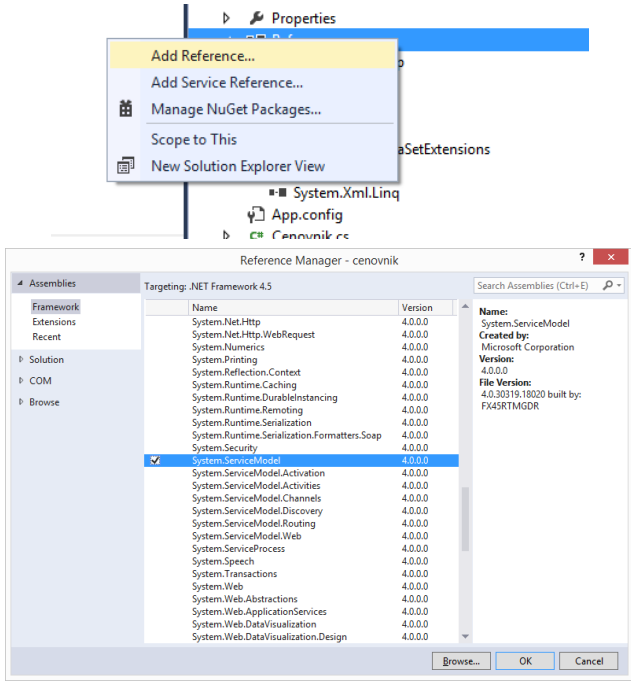
Napomena: Kada se neki interfejs primeni tj. ako neka klasa nasleđuje interfejs, to znači da treba da realizuje sve metode tog interfejsa. Osim navedene metode ovaj interfejs ima pridružene atribute (dekoratore) uz funkciju i uz sam interfejs, `OperationContract`, odnosno `ServiceContract`. Više o značenju ovih dekoratora biće kasnije kada budemo govorili o ugovorima. Za sada recimo da su neophodni pri realizaciji servisa.

Drugo, potrebno je definisati način komunikacije tzv. kanal za komunikaciju. Servis mora da osluškuje zahteve na određenoj lokaciji tj. adresi.

Međutim, u osnovim šablonima projekata, kompajler ne prepoznaje ove dekoratore, odnosno neophodno je uključivanje dopunskih biblioteka.

Dodavanje reference

Ukoliko tip projekta koji je napravljen nije za WCF servise (na primer Console Application) potrebno je uključiti dodatne biblioteke.



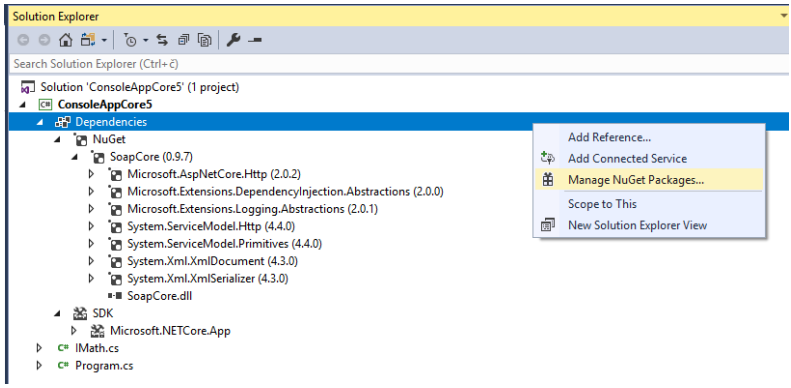
Slika 12.1. Dodavanje reference

Na slici je prikazano dodavanje `System.ServiceModel` biblioteke postojećem projektu.

Nakon, dodavanja biblioteke, greške prevodioca biće uklonjene, a u sekciji References mogu se naći nove biblioteke koje čine projekat.

Važna napomena:

Ako se referenca dodaje za .Net Core projekte (platforma koja nudi univerzalna rešenja tj. projekte za sve platforme), onda se takvim projektima, dodaje biblioteka `SoapCore` koristeći `NuGet` okruženje za dodavanje paketa postojećem projektu.



Slika 12.3. Dodavanje pakete pomoću NuGet-a

Moguće je kreirati samo klase servisa i bez upotrebe interfejsa, ali se takva praksa ne preporučuje. Postojanje interfejsa omogućava da u postupku prevođenja tj. kreiranja servisa, automatizovano kreiramo i mex krajnju tačku i izložimo opis servisa preko WSDL-a na validan način, pa tako olakšamo kasnije povezivanje sa istim. Sam servis se implementira kao klasa koja nasleđuje interfejs servisa (i ugovor koji se iz interfejsa generiše).

Implementacija interfejsa

Implementacija interfejsa servisa ne razlikuje se od implementacije interfejsa inače. Na primer:

```
class Cenovnik : ICenovnik
{
    public double cenaArtikla(string artikal)
    {
        switch (artikal.ToLower().Trim())
        {
            case "kafa": return 55.00;
            case "sok": return 105.00;
            case "voda": return 45.00;
            default: return 0;
        }
    }
}
```

Hostovanje

Svaki servis, da bi se izvršavao treba da funkcioniše kao programska komponenta. Tačnije, dovoljan je bilo kakav izvršni proces na računaru koji bi pokrenuo servis i bio domaćin za njegov rad. Za takav proces kažemo da hostuje servis.

Servis može biti

- samo-hostovan, u svom izvršnom fajlu, ili
- može biti hostovan u nekom drugom procesu koji je kontrolisan od stane nekih eksternih agenata, kao na primer:
    - IIS ili
    - Windows Authentication Service (WAS).
    - Izvršni fajl servisa može biti pokrenut i automatski kao Windows servis.

Pri pokretanju servisa mora da se definiše jedna ili više servisnih pristupnih tačaka sa svim informacijama potrebnim za komunikaciju sa klijentima. Dakle, adresa, povezivanje i ugovor.

Primer pokretanja servisa iz konzolne aplikacije.

```
Uri uriAdresa = new Uri("http://localhost:8000/Cenovnik");
ServiceHost sh = new ServiceHost(typeof(Cenovnik), uriAdresa);
sh.AddServiceEndpoint(
          typeof(ICenovnik),
          new BasicHttpBinding(),
          "");
sh.Open();
Console.WriteLine("Press <Enter> to terminate.\n\n");
Console.ReadLine();
sh.Close();
```

Objašnjenje: Pokretanje servisa se obavlja tako što se servis pridruži host objektu, `sh`, a zatim da se pokrene osluškivanje komandom `sh.Open()`. Pre pokretanja hosta, njemu se dodaju krajnje tačke metodom `AddServiceEndpoint`. Argumenti ove metode su osnovni elementi krajnje tačke: ugovor, povezivanje i adresa. Tačnije, kada je reč o adresi, u ovoj metodi se dodaje adresa krajnje tačke u odnosu na ceo host, tj. relativna adresa krajnje tačke na hostu. Osnovna adresa hosta definiše u konstruktoru objekta (`uriAdresa`). Nakon `Open` komande, izvršavanje konzolnog programa se zaustavlja komandom `Console.ReadLine()`, i na taj način servis ostaje pokrenut i u stanju osluškivanja poziva od klijenata.

Prva klijentska aplikacija

Kada je reč o pisanju klijentskih aplikacija, odmah treba naglasiti da je taj proces jednostavniji ukoliko se za to koristi **servisna referenca**. Kako se kreira servisna referenca pokazaćemo nešto kasnije. Na ovom mestu videćete kod sa pratećim objašnjenjem, napisan pomoću osnovnih klasa za komunikaciju.

```
1.  ChannelFactory<ICenovnik> cf = new ChannelFactory<ICenovnik>
2.  (new BasicHttpBinding(),
     new EndpointAddress("http://localhost:8000/Cenovnik"));
3.  ICenovnik o = cf.CreateChannel();
4.  double rez = o.cenaArtikla("kafa");
5.  Console.WriteLine("Cena kafe je: {0} \n\n", rez);
6.  cf.Close();
```

```
7.  //interfejs na strani klijenta koji definiše ponašanje
    servisa tj ugovor
8.  [ServiceContract]
9.  interface ICenovnik
10. {
11. [OperationContract]
12. int Add(int a, int b);
13. }
```

Objašnjenje koda: Ključni objekat koji se kreira za komunikaciju je `ChannelFactory<ICenovnik>`. Dakle, radi se o generičkoj klasi `ChannelFactory` kojoj se prosleđuje interfejs ugovora `ICenovnik`. Ovo je važan detalj. Klijentska aplikacija mora poznavati funkcije na servisu, a to se ostvaruje poznavanjem servisnog ugovora tj. odgovarajućeg interfejsa. S obzirom da smo sami pisali servis, potpuno je jasno da nam je interfejs poznat. Postavlja se pitanja, šta ako to nije slučaj? Odgovor će biti kasnije u poglavlju. Konstruktoru se prosleđuje povezivanje i adresa krajnje tačke. Zatim se kreira kanal: `ICenovnik o = cf.CreateChannel()`. Kreirani kanal vraća i realizovane metode interfejsa pa se može koristiti tip `ICenovnik`, pošto na taj način dobijamo mogućnost da objekat kreiranog kanala koristi metode interfejsa. Ostatak koda je jasan.

Upotreba konfiguracionih fajlova

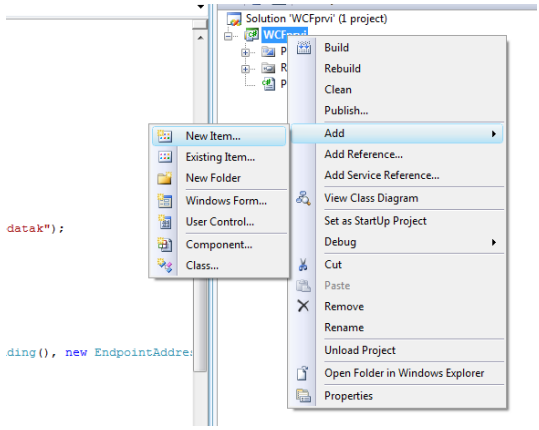
Osim same logike koje funkcije izvršavaju, gotovo sve ostalo na strani servisa može biti opisano posebnim parametrima. Ovi parametri se mogu čuvati kao deo podešavanja odnosno sadržaj konfiguracionog fajla.

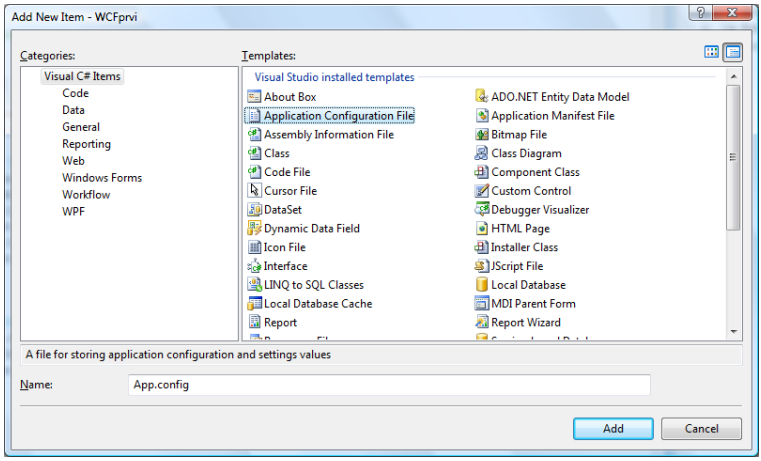
Dakle, može da se konfigurišu:

- Adresa;
- Osobine tj. vrsta prenosnog kanala;
- Ugovor za komunikaciju.

Tačnije, definišu se krajnje tačke na hostu, može ih biti više. Za svaku krajnju tačku posebno se definiše gore navedeni parametri. Pogledajmo sada postupak dodavanja jednog konfiguracionog fajla.

Desnim klikom na projekat u prozoru Solution Explorer, bira se opcija dodavanja nove stavke, kao na slici:





Slika 12.4. Dodavanje konfiguracionog fajla

Konfiguracioni fajl je xml dokument. Opis servisa smeša se u sekciju sysetm.serviceModel. U nastavku dajemo primer jednostavne konfiguracije:

```
<configuration>
  <system.serviceModel>
    <services>
      <service name="WCFprvi.Cenovnik">
        <host>
          <baseAddresses>
            <add baseAddress="http://localhost:8000/cenovnik"/>
          </baseAddresses>
        </host>
        <endpoint address="dodatak" binding="basicHttpBinding"
contract="WCFprvi.ICenovnik" />
      </service>
    </services>
  </system.serviceModel>
</configuration>
```

Ako servis koristi povezivanje bazirano na HTTP protokolu, adresa kojoj klijent šalje poruke je oblika http://www.myserver.com:8080/MyService/

Ako je adresa krajnje tačke specificirana (relativna adresa), ona dopunjuje baznu adresu u servisu (apsolutna adresa)

Mex krajnja tačka

U servisima metapodaci se odnose na informacije koje opisuju detalje konekcije odnosno sve ono što je potrebno klijentu da uspostavi vezu i koristi servis. Klijenti mogu tražiti metapodatke od pokrenutog servisa kako bi saznali sve informacije o njegovim krajnjim tačkama. U toku faze projektovanja, klijent šalje poruku sa zahtevom za definisanje standarda *WS-MetadataExchange* i prima WSDL kao povratne podatke. Nakon toga, WSDL se koristi za kreiranje *proxy* klase odnosno konfiguracionog fajla koji se koristi za komunikaciju sa servisom.

MEX krajnja tačka nije obavezna za rad servisa. To znači da klijent ne može zahtevati od servisa da izloži WSDL informacije. Bez znanja o adresi, povezivanju i ugovoru teško je kreirati odnosno koristiti servise. WCF to olakšava izlaganjem MEX krajnje tačke tako da klijenti mogu lako kreirati komunikaciju sa servisom. MEX krajnja tačka se izlaže: u kodu ili u konfiguraciji. Pokazaćemo oba načina, s tim da je drugi mnogo lakši i biće kasnije standardno korišćen.

MEX u kodu:

```
Uri adresa = new Uri("http://localhost:8000/Cenovnik");
ServiceHost serviceHost = new ServiceHost(typeof(Cenovnik), adre
sa);
serviceHost.AddServiceEndpoint(typeof(ICenovnik),new BasicHttpBi
nding(),"");
//mex
ServiceMetadataBehavior ponasanja = new ServiceMetadataBehavior(
);
ponasanja.HttpGetEnabled = true;
serviceHost.Description.Behaviors.Add(ponasanja);
serviceHost.AddServiceEndpoint(typeof(IMetadataExchange),Metadat
aExchangeBindings.CreateMexHttpBinding(), "mex");
serviceHost.Open();
Console.WriteLine("Press <Enter> to terminate.\n\n");
Console.ReadLine();
serviceHost.Close();
```

Mex krajnja tačka u konfiguracionom fajlu biće automatski kreirana kada se koriste standardni šabloni za projekte za izradu servisa. U slučaju kada koristimo konzolnu aplikaciju ovaj deo izmena potrebno je da uradimo sami. U nastavku je dat kod konfiguracionog fajla:

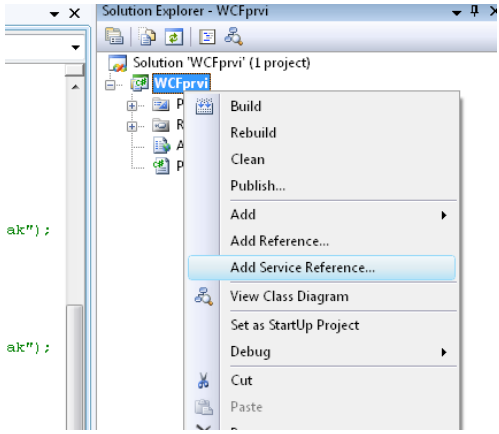
```
<services>
      <service name="WCFprvi.Cenovnik"
        behaviorConfiguration="myServiceBehavior">
        <host>
          <baseAddresses>
            <add baseAddress="http://localhost:8000/cenovnik"/>
          </baseAddresses>
        </host>
        <endpoint address="dodatak" binding="basicHttpBinding"
contract="WCFprvi.ICenovnik" />
        <endpoint address="mex" binding="mexHttpBinding"
contract="IMetadataExchange"/>
      </service>
    </services>
    <behaviors>
      <serviceBehaviors>
        <behavior name="myServiceBehavior">
          <serviceMetadata httpGetEnabled="true"/>
        </behavior>
      </serviceBehaviors>
    </behaviors>
```

WCF klijent i konfiguracioni fajl

Za klijentsku aplikaciju neophodno je da postoje ABC podaci o servisu. Ovi podaci i kod klijenta mogu biti u konfiguracionom fajlu.

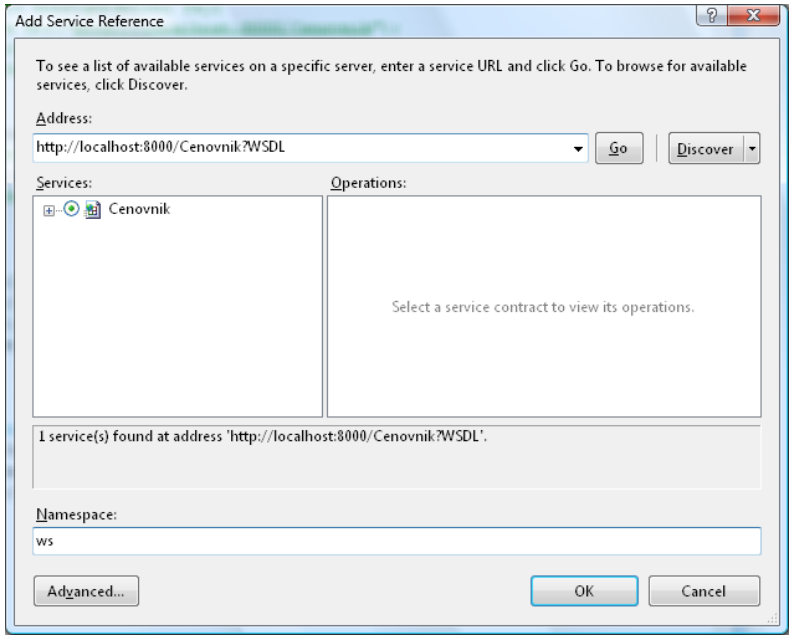
VS daje mogućnost automatskog kreiranja konfiguracionog fajla i *proxy* klase (koja kreira kanal na osnovu ABC podataka) za komunikaciju sa WCF servisom, dodavanjem servisne reference.

Upotreba servisna referenca nalik je upotrebi obične reference u projektu. U oba slučaja, nakon dodavanja, u projektu su dostupne metode odnosno funkcije na koje se referiše. Kreiranje servisne reference vrši se preko konteksnog menija projekta, tj. desnim klikom na projekat, pa izborom stavke Add Service Reference..., pogledati sliku:



Slika 12.4. Pokretanje dodavanja servisne reference

Zatim se otvara forma za nalaženje servisne reference. Servis na koji se povezujemo mora biti pokrenut ili deo rešenja u okviru Visual Studio okruženja. Pogledajte narednu sliku, pa ispod nje slede dodatna objašnjena.



Slika 12.5. Izbor, podešavanje i imenovanje servisne reference

U polje Address unosi se adresa servisa (do WSDL-a tj. mex krajnje tačke ili do samog servisa). Ako je servis deo VS IDE rešenja može se koristiti dugme Discover. Pošto se klikne na Go dugme, u delu Services biće prikazani nađeni servisi. Na kraju, dodaje se imenski prostor u kome će se formirati proxy klase. Na kraju, pritiskom na OK vrši se kreiranje klasa na osnovu WSDL-a servisa koje značajno olakšavaju rad, a istovremeno i vrše se promene u konfiguracionom fajlu. Dakle:

Dodato u conf fajl:

```
<bindings>
  <basicHttpBinding>
    <binding name="BasicHttpBinding_ICenovnik"
      closeTimeout="00:01:00" openTimeout="00:01:00"
      receiveTimeout="00:10:00" sendTimeout="00:01:00"
      allowCookies="false" bypassProxyOnLocal="false"
         hostNameComparisonMode="StrongWildcard" />
      <security mode="None">
        <transport clientCredentialType="None"
          proxyCredentialType="None"
          realm="" />
        <message clientCredentialType="UserName"
          algorithmSuite="Default" />
      </security>
    </binding>
  </basicHttpBinding>
</bindings>
<client>
   <endpoint address="http://localhost:8000/cenovnik/dodatak"
    binding="basicHttpBinding" bindingConfiguration=
    "BasicHttpBinding_ICenovnik" contract="ws.ICenovnik" name=
    "BasicHttpBinding_ICenovnik" />
</client>
```

Kreirane klase u Reference.cs:

```
[System.CodeDom.Compiler.GeneratedCodeAttribute("System.ServiceM
odel", "3.0.0.0")]
[System.ServiceModel.ServiceContractAttribute(ConfigurationName=
"ws.ICenovnik")]
    public interface ICenovnik {
            * * * * * *
        [System.CodeDom.Compiler.GeneratedCodeAttribute("System.
ServiceModel", "3.0.0.0")]
    public interface ICenovnikChannel : WCFprvi.ws.ICenovnik, Sy
stem.ServiceModel.IClientChannel {
    }

    [System.Diagnostics.DebuggerStepThroughAttribute()]
    [System.CodeDom.Compiler.GeneratedCodeAttribute("System.Serv
iceModel", "3.0.0.0")]
    public partial class CenovnikClient : System.ServiceModel.Cl
ientBase<WCFprvi.ws.ICenovnik>, WCFprvi.ws.ICenovnik {

        public CenovnikClient() {
        }
    * * * * * * *
    }
```

Zaista, nakon ovog podešavanja, programeru ostaje da kreirane klase i podešavanja jednostavno iskoristi za rad.

```
ws.CenovnikClient cf = new WCFprvi.ws.CenovnikClient();
double p = cf.cenaArtikla("kafa");
Console.WriteLine("Dobijeno: {0} \n\n", p);
cf.Close();
```

Ugovori

Vrste ugovora

Ugovor predstavlja **opis poruka odnosno funkcija_**koje mogu da se pošalju/prime preko krajnjih tačaka. U ovom programskom modelu razmatra se više vrsta ugovora:

- **ServiceContract**
  - Definiše implementirane operacije. ServiceContract mapira metode klasa u WSDL, tipove porta i operacije. **OperationContract** je deo ServiceContract-a i opisuje operacije/metode servisa.
- **DataContract**
  - Definiše strukture podataka koje se koriste. Mapira tipove u XSD i definiše na koji način će oni biti serijalizovani. *DataContract* opisuje sve podatke koje servis prima ili šalje.
- **MessageContract**
  - Mapira tipove u SOAP poruke, definiše format tih poruka i utiče na WSDL i XSD definicije tih poruka. MessageContract obezbeđuje direktnu kontrolu nad SOAP zaglavljem i telom poruke.

Ugovori su definisani preko WSDL dokumenta, odnosno XSD-a. Ugovori su kodirani koristeći CLR tipove (tipovi koje standardno koristimo u C#). Zato se uspostavlja mapiranje.

Pri pisanju koda servisa, deklarišemo klase sa WCF atributima (stoje ispred klase ili metode u uglastim zagradama): **[ServiceContract], [OperationContract], [FaultContract], [MessageContract]** odnosno **[DataContract].**

Pri pisanju klijentskog dela uspostavlja se upit ka servisu kako bi se saznali svi detalji ugovora na osnovu koga se generiše *proxy* klasa, koja izlaže servisni interfejs i koja nam omogućava korišćenje servisa.

Pri kompajliranju, kada klijent pozove metodu koja je na strani servera, WCF serijalizije CLR tipove i potpise i pravi se XML dokument koji se zatim šalje po definisanoj ABC-di.

Servisni ugovor

Opisuje ***interfejs*** sastavljen od operacija koje su implementirane na strani servisa tj. na strani jedne krajnje tačke. Ugovori se baziraju na formatu poruka i načinu na koji se one razmenjuju. Format poruka se dalje opisuje od strane ***DataContract-a*** i ***MessageContract-a***.

Servisni ugovor je obeležena sa ***[ServiceContract]*** i njegove metode obeležene su sa ***[OperationContract]*** i izložene preko ***WSDL***-a da bi klijent mogao da ih koristi. Klase su oivičene elementom ***wsdl:service,*** a operacije sa ***wsdl:operation***.

Poruke: zahtev-odgovor

Sinhrona poruka tipa zahtev-odgovor (eng. *request-response)* omogućava komunikaciju na jednostavan način: klijent inicira zahtev, servis vraća odgovor. Tokom dizajna servisa, koristeći “Add Service Reference” ili koristeći alatku svcutil.exe poziva se MEX endpoint i generiše se proxy

klasa na klijentskoj strani koja obezbeđuje rad sa udaljenim servisnim operacijama.

Proxy serijalizije ime metode i parametre u SOAP poruku, koju šalje servisu, a zatim ostaje u stanju osluškivanja povratne poruke. Po odgovoru servis kreira podatak odgovora koje je nekog .NET tipa.

U WCF-u, request-response sinhrone servisne operacije **prouzrokuju čekanje tj. blokadu klijenta** sve dok se ne odradi zahtevana funkcija. Tačnije, proxy koristi blokadu WCF kanala zaduženog za komunikaciju sa servisom, a ona prouzrokuje blokadu klijentske aplikacije za onoliko vremena koliko treba servisu da završi sa radom i pošalje odgovor klijentu.

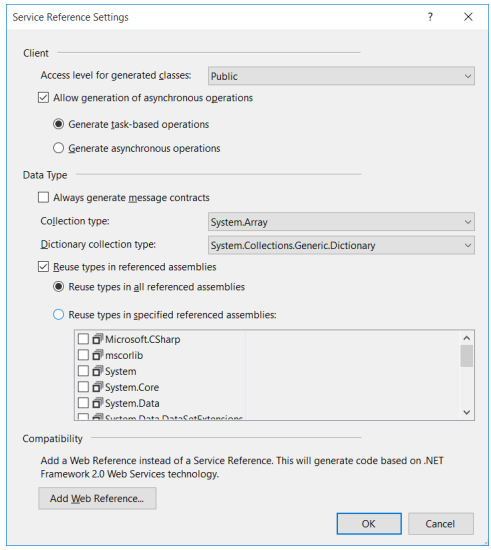
Asinhroni zahtev-odgovor

Kao odgovor na problem blokirajućih metoda, moguće je koristiti asinhroni način programiranja kako bi obezbedili asinhroni rad klijenta. To se postiže pomoću klase ***IAsyncResult*** i dve metode:

- *BeginOperationName*;
- *EndOperationName.*

Prvo se poziva *BeginOperationName,* zatim nastavlja sa radom u istoj niti, dok se asinhrone operacije obavljaju u drugoj niti. Za svaki poziv metode *BeginOperationName* klijent prosleđuje ***delegat*** koji se poziva kada i sama metoda i služi za skladištenje korisničkih informacija, kasnije se poziva i *EndOperationName* kako bi se pribavili rezultati.

Automatizovano generisanje asinhronih metoda vrši se tokom dodavanja servisne reference. Ovo se postiže klikom na *Advanced* dugme u "*Add Service Reference"* dijalogu selektujući potvrdno polje (check box) "*Generate Asynchronous Operations*".



Slika 12.6. Izbor generisanja asinhronih metoda

Napomena. Servis nema informaciju o tome da li je klijent u asinhronoj ili sinhronoj komunikaciji sa servisom.

U kodu:

```
// koristeci dogadjaj
proxy.AnalizaDugotrajnaCompleted += new EventHandler<System.Comp
onentModel.AsyncCompletedEventArgs>(proxy_AnalizaDugotrajnaCompl
eted);
Console.WriteLine("zapoceta analiza");
proxy.AnalizaDugotrajnaAsync();

// koristeci povratnu metodu
proxy.BeginAnalizaDugotrajna(new AsyncCallback(callback2), null)
;
Console.ReadLine();
```

Obratite pažnju da smo promenili standardnu opciju generisanja zasnovnu na task operacijama, pa sada pogledajmo i ovaj način upravljanja asinhronim metodama.

Rad sa *task-based* operacijama

Ukoliko se koristi generisanje zasnovano na *task* operacijama, onda se asinhrona pozivanja izvodi pomoću await-async funkcija.

```
async private void Form1_Load(object sender, EventArgs e)
{
    // sinhroni poziv
    ServiceReference1.Service1Client x =
       new ServiceReference1.Service1Client();
    string r = x.GetData(44);
    // asinhroni koristeći await
    string xx = await x.GetDataAsync(33);
}
```

Jednosmerne i dupleks operacije

Jednosmerne operacije

Jednosmerna ***(****eng. one way)* razmena poruka je korisna u slučaju da korisnik želi da pošalje informaciju servisu, ali ne želi od sistema nikakav odgovor.

Ovakvom metodom, klijentu samo treba potvrda da je servis primio njegovu poruku, ne i povratni podatak.

Ponekad ase ovakva metoda slanja poruka zove i *"Fire and forget".*

WCF podržava ovaj tip razmena poruka na servisnom operacionom nivou. Kada klijent pošalje poruku servisnoj krajnjoj tački čija je operacija označena sa *OneWay*, kontrola nad aplikacijom će se vratiti korisniku pre nego što server završi sa obradom podatka.

OneWay operacije su specifikovane u **[OperatonContract]** atributu, tako što se upotrebi modifikator *IsOneWay=true.*

```
[OperationContract(IsOneWay=true)]
void Clear();
```

Dupleks Operacije

WCF omogućuje dvosmernu komunikaciju kroz kanal koristeći dupleks servis ugovore.

Poruke se mogu slati iako nisu zatražene od strane klijenta ili servisa. Zato što poruke mogu da putuju u oba pravca, od servisa ka klijentu i od klijenta ka servisu, oba učesnika u dupleksu moraju da imaju adresu, povezivanje i ugovor. Kako bi omogućio protok podataka od klijenta ka servisu i obrnuto, WCF kreira dodatni kanal.

Operacije u dupleksu su ili *oneway* ili *request/response* operacije. Takođe se implementira uzorak poruka koje se mogu slati u bilo kom smeru. Pošto se poruke mogu slati u oba smera, oba dela moraju da sadrže ABCedu. Ako početni kanal ne podržava bidirekcionu komunikaciju WCF kreira i drugi kanal koristeći isti protokol.

Problem slanja poruka u oba smera može da se definiše preko dva modela, tj. koristeći dva *OneWay* ugovora, ili jedan dupleks ugovor. Sa parom od dva *OneWay* ugovora, i klijent i servis su nezavisni jedan od drugog. I jedan i drugi imaju krajnje tačke na koje se šalju poruke. Kada je reč o dupleks ugovoru, klijent ne postaje eksplicitno servisna stana, ne bira povezivanje niti otkriva svoje krajnje tačke. **Umesto toga, adresu, povezivanje, i ugovor klijentske krajnje tačke implementira objekat tipa ChanalFactory u trenutku kada je dupleks konekcija pokrenuta od strane klijenta**. Dupleks ugovor sadrži interfejs specifikacije za obe strane, i servisni i klijentsku krajnju tačku. U ovakvim tipovima ugovora, deo servisnog ugovora je implementiran u klijentski deo.

Zavisno od vrste povezivanja nastaje kreiranje dodatnog kanala ili ne. Ako je povezivanje *„named pipes TCP“* onda su oni dvosmeri, ako je u pitanju *http* WCF će kreirati dodatni kanal.

Više ugovora i krajnjih tačaka

Jedna krajnja tačka može imati samo jedan ugovor. Sa druge strane, jedan ugovor može biti korišćen od više krajnjih tačaka.

Više krajnjih tačaka, sa istim povezivanjem, ali različitim ugovorom može biti locirano na istoj adresi, dajući iluziju da je jedna krajnja tačka implementirala više ugovora.

Izlažući ugovor kroz više krajnjih tačaka servisa, ugovor se može učiniti dostupnim uz različita povezivanja.

Postoji i drugi način realizacije više ugovora na jednoj krajnjoj tački. Pogledajte sledeći kod:

```
[ServiceContract]
interface ICenovnik
{
    [OperationContract]
    string dajCenu(string vrsta);
}

[ServiceContract]
interface IAnaliza
{
    [OperationContract]
    void AnalizaDugotrajna();
    [OperationContract(IsOneWay = true)]
    void AnalizaKratkotrajna();
}

[ServiceContract]
interface IAnalizaCenovnika : IAnaliza, ICenovnik
{ }
```

Dakle, pošto je višestruko nasleđivanje kod interfejsa moguće, onda je lako napraviti novi interfejs koji nasleđuje dva ugovora, tačno čineći novi treći ugovor koji obuhvata metode iz oba. Taj novi ugovor se može postaviti na jednu krajnju tačku.

Primer:

```
<service name="WCFprvi.Cenovnik"
behaviorConfiguration="myServiceBehavior">
  <host>
    <baseAddresses>
       <add baseAddress="http://localhost:8000/cenovnik"/>
     </baseAddresses>
   </host>
   <endpoint address="dodatak" binding="basicHttpBinding"
contract="WCFprvi.ICenovnik" />
   <endpoint name="a" address="dodatak"
binding="basicHttpBinding" contract="WCFprvi.IAnaliza" />
```

```
   <endpoint name="b" address="dodatak" binding="wsHttpBinding"
contract="WCFprvi.IAnaliza" />
   <endpoint address="dodatak" binding="basicHttpBinding"
contract="WCFprvi.IAnalizaCenovnika" />
   <endpoint address="mex" binding="mexHttpBinding"
contract="IMetadataExchange"/>
</service>
```

Napomena. Ukoliko ima više krajnjih tačaka, potrebno je da budu imenovane kako bi mogli upravljati svakom po osnovu imena.

Servisne operacije se imenuju atributom [OperationContract]. Ostale metode mogu da posluže za funkcionisanje klase i rad sa podacima, ali samo one operacije koje imaju ovaj atribut biće izložene preko WSDL-a i biće dostupne klijentskim aplikacijama.

Ukoliko naziv operacije ne treba da bude identičan nazivu metode u klasi, naziv operacije se može promeniti. Za to se koriste pripadajući atribut uz operaciju.

Ugovor o podacima

Logika jednog servisa implementirana je preko funkcija. Van programa, sve o servisu opisuje se preko WSDL-a. Podaci koje koriste servisi definisani su prostim ili složenim tipovima podataka, a van servisa su predstavljeni preko XSD šeme.

Kao što postoji servisni ugovor postoje I ugovori o podacima (eng. *Data Contract*) koji mapiraju .NET CRL tipove u XSD šeme.

Naravno, koristeći biblioteku klasa za razvoj servisa, programeri su orijentisani ka programskoj jeziku, kodu i interfejsu, a ne ka XSD odnosno ka WSDL semantici.

Tokom dizajna, atribut **[DataContract]** se koristi za opis podataka koji treba da budu predstavljeni u XSD šemi i uključene u WSDL dokument a koji će biti izložen korisniku. **[DataMember]** atribut definiše koji članovi klase treba da budu uključeni u spoljnu reprezentaciju.

U toku izvršavanja, klasa **DataContractSerializer** serijalizuje objekte u XML koristeći pravila opisana u DataContract i DataMember atributima.

Šta se serijalizuje?

DataContractSerializer će serijalizovati promenljive sledećih tipova i izložiti ih u WSDL dokumentu, ako su zadovoljeni sledeći uslovi:

- Ako su označeni sa `[DataContract]` i `[DataMember]` atributima.
- Tipovi koji su opisani sa `[CollectionDataContract]`
- Tipovi izvedeni iz `XmlSerializable`
- Tipovi označeni sa `Serializable`, a čiji članovi nisu `NotSerializabled`
- Tipovi označeni sa `Serializable` koji implementiraju `ISerializable`
- CLR primitivni tipovi, kao što su `string`-ovi i `int`-ovi.
- `Bytes array`, `DateTime`, `DateSpan`, `GUID`, `Uri`, `XmlQualifiedName, XmlElement`, `XmlNode`
- Nizovi i kolekcije tipa `List<T>, Dictionary<K,V>` , `Hashtable`
- `Enumeratori`

Serijalizacija u WSDL-u

Podrazumevano, ime XML šeme je isto kao ime klase i ciljnog prostora imena, pri čemu se vrši nadovezivanje imena klase. Imena se mogu preklopiti nekim posebnim imenom.

Takođe, članovi klase koji imaju *DataMember* će biti izloženi preko WSDL-a, za razliku od onih članova klase koji nemaju taj atribut.

Napomena: Generisanje WSDL dokumenta je ugrađeno u projektne šablone i ne mora se posebno raditi. Serijalizacija podrazumeva uključivanje klasa za serijalizaciju odgovarajućeg imenskog prostora.

Sledi primer koda i generisanje pratećeg WSDL dokumenta.

```
using System.ServiceModel;
using System.Runtime.Serialization;

[DataContract]
public class clsCenonvnik{
    [DataMember(Name="CurrentPrice", IsRequired=true, Order=0)]
    public double tekucaCena;
    [DataMember(Name="CurrentTime", IsRequired=true, Order=1)]
    public DateTime tekuceVreme;
}
[ServiceContract]
public class Cenovnik{
    [OperationContract]
    private clsCenonvnik cenovnik(){
        clsCenonvnik r = new clsCenonvnik();
        r.tekuceVreme = DateTime.Now;
        r.tekucaCena = 49.99;
        return r;
    }
}
```

Pitanja i zadaci

1. Šta znači skraćenica SOAP i kakvi su to SOAP servisi?
2. Šta je krajnja tačka?
3. Šta predstavlja ugovor za jedan servis?
4. Kako se definiše adresa jednog servisa, a kako jedne krajnje tačke?

5. Šta je bitno tj. koji objekti se definišu pri definisanju jedne krajnje tačke?
6. Koje vrste servisnih operacija postoje?
7. Šta je to kontejner servisa?
8. Kako se jedan WCF servis može hostovati?
9. Objasni značenje i ulogu WSDL jezika.
10. Kako se kreira jedna servisna referenca u projektu?
11. Zašto se koristi interfejs pri kreiranju servisa?
12. Čemu služe konfiguracioni fajlovi i kako se dodaju jednom projektu?
13. Šta je mex krajnja tačka?
14. Koje sve vrste ugovora poznajete?
15. Napisati jedan DataContract za rad sa knjigama.
16. Da li jedan servis može da poseduje više krajnjih tačaka?
17. Da li jedan servis može da poseduje više ugovora?
18. Da li može da se više ugovora postavi na jednu krajnju tačku? Kako?
19. Da li može da se postavi više krajnjih tačaka za jedan ugovor?
20. Koje vrste poruka možete da kreirate?
21. Objasniti kako se vrši serijalizacija tipova podataka. Navesti primere.
22. Kreirati servis sa i bez mex krajnje tačke.
23. Kreirati klijetsku aplikaciju koja će se povezati sa servisom koji ima i koji nema mex krajnju tačku.
24. Kakve su to dupleks operacije?
25. Kakve su operacije OneWay?
26. Šta je to asnihroni zahtev-odgovor?
27. Kako se kreiraju asinhrone metode na klijentskoj strani?

28. Prikazati na koje sve načine možete obezbediti asinhrono pristupanje podacima na klijentskoj strani.

13. Veb Api .Net

Ovo poglavlje posvećeno je Web Api programskom modelu zasnovanom na tehnologiji Asp .Net Core i MVC šablonu. Prezentuju se osnovne karakteristike ove tehnologije otvorenog koda. Najpre se prikazuje način kreiranja projekta, dodavanje modela, a zatim se dodaju kontroleri i kod za rutiranja zahteva. Asinhroni princip objašnjava se postepenim uvođenjem i tumačenjem klasa i interfejsa specifične namene. Posebna pažnja posvećuje se obradi http zahteva GET odnosno POST, preko definisanja odgovarajućih metoda servisa. Zatim, za potrebe testiranja servisa uvodimo JavaScript odnosno jQuery metode za pozive. U ovom delu akcenat je na POST zahtevima. Istovremeno, prikazujemo testiranje i pomoću *Postman* aplikacije.

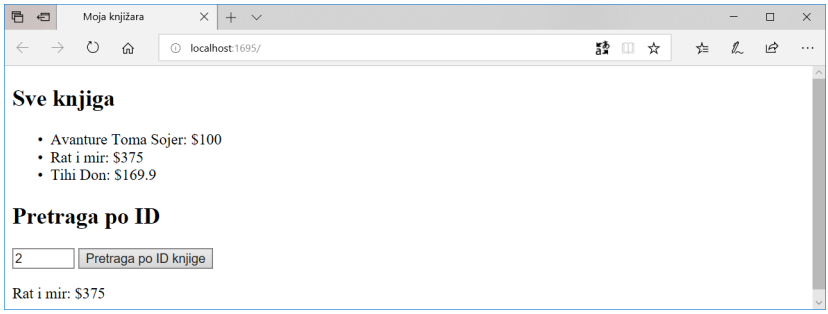
Uvod

HTTP je protokol namenjen ne samo za dostavljanje veb sadržaja u vidu stranica, to je moćna platforma i za izgradnju API-ja koji nude usluge i podatke. HTTP je jednostavan, fleksibilan i svuda prisutan. Gotovo svaka veb platforma ima biblioteke koje nude HTTP usluge. HTTP servisi mogu da ostvare širok spektar klijenata, uključujući pregledače, mobilne uređaje kao i tradicionalne desktop aplikacije.

ASP.NET Web API je radni okvir za izgradnju veb API-ja u .NET Framework-u. U nastavku pokazaćemo način kreiranja ASP.NET Web API koji formira listu knjiga.

Kreiranje projekta

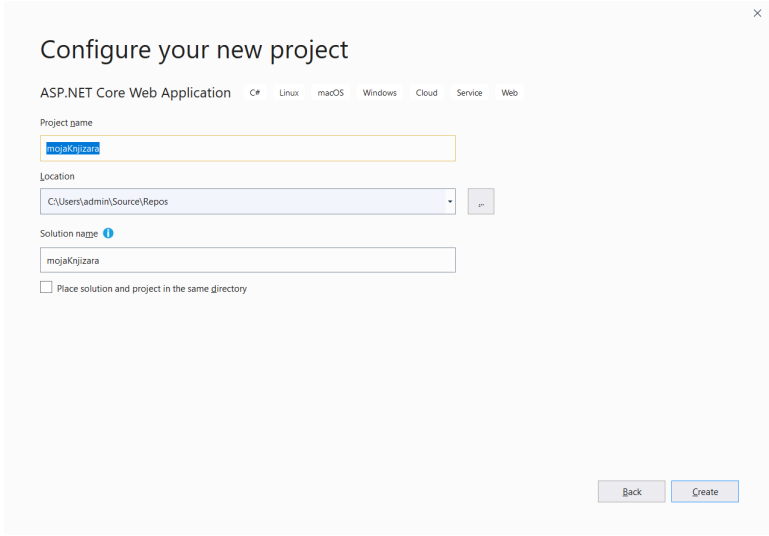
U ovom predavanju, koristićete ASP.NET Web API za kreiranje veb API koji daje listu knjiga. Veb stranica koristi jQuery pristupanje api funkcijama i vraćanje rezultata.



Slika 13.1. Prikaz stranice koju kreiramo primenom Web API

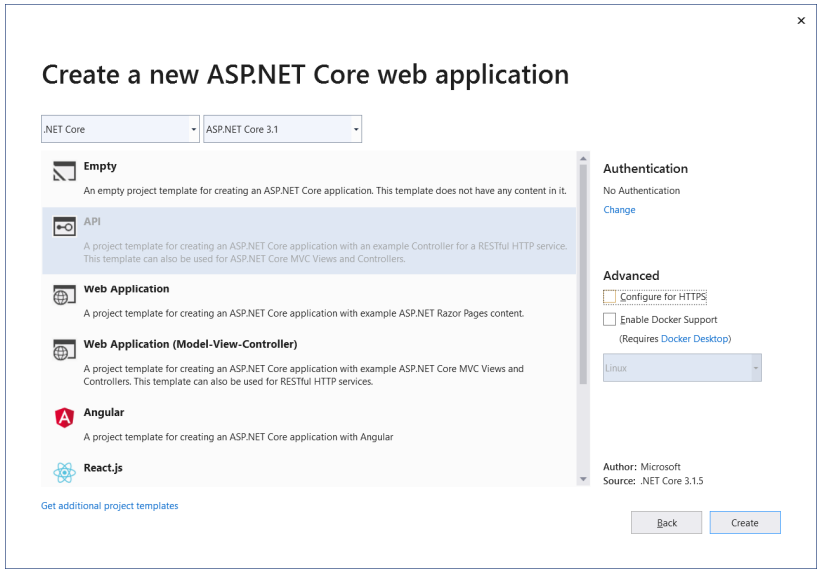
Najpre pokrenimo Visual Studio i odaberimo **Create New Project**.

U delu za pretragu šablona biramo veb šablon nezavisan od platforma. U listi nađenih šablona projekta odaberimo **ASP.NET Core Web Application**. Imenujmo projekat "mojaKnjizara" a zatim **Create**.



Slika 13.2. Formiranje novog projekta

U dijalogu **Create a new ASP.NET Core web application** odaberite šablon **API**, uz prethodno isključivanje opcije za konfigurisanje HTTPS.



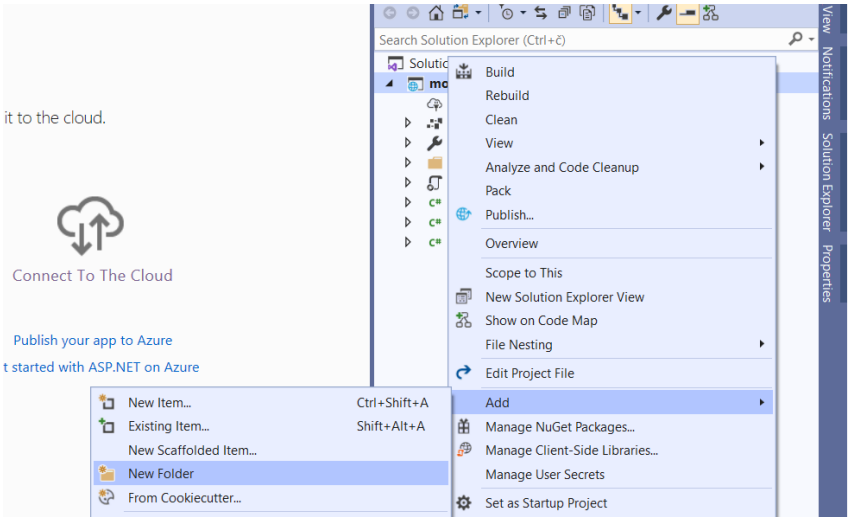
Slika 13.3. Izbor šablona za novi projekta

Dodavanje Modela

Model je objekat koji zadužen za podatke u aplikaciji. Modeli mogu odgovarati objektima u bazi ali ne obavezno, odnosno mogu biti prilagođeni za određeni pogled. ASP.NET Web API može automatski da serijalizije model u formate kao što je JSON odnosno XML. A zatim se takvi podaci mogu ubaciti u telo poruke koja je HTTP odgovor. Većina klijenata može da analizira ili XML ili JSON. Štaviše, možete naznačiti format koji želite da se prihvati kroz zaglavlje HTTP zahteva za poruke.

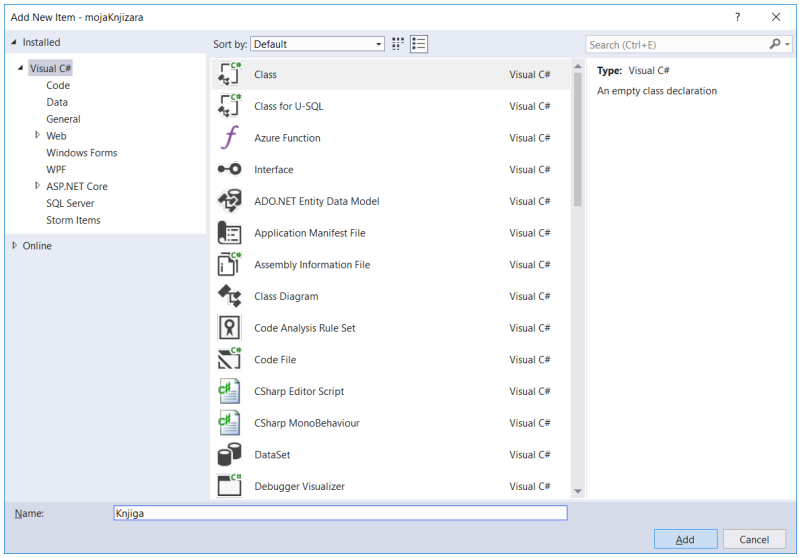
Kreirajmo sada jednostavan model.

Najpre kreirajmo folder Models u kom ćemo smeštati buduće klase pojedinih modela. Folder dodajemo desnim klikom na projekat u prozoru `Solution Explorer, kao na slici.`



Slika 13.4. Dodavanje foldera Models

Zatim, desni-klik na folder Models, a onda iz kontekstnog menija birati opciju **Add** a zatim odabrati **Class**.



Slika 13.5. Dodavanje nove klase za model

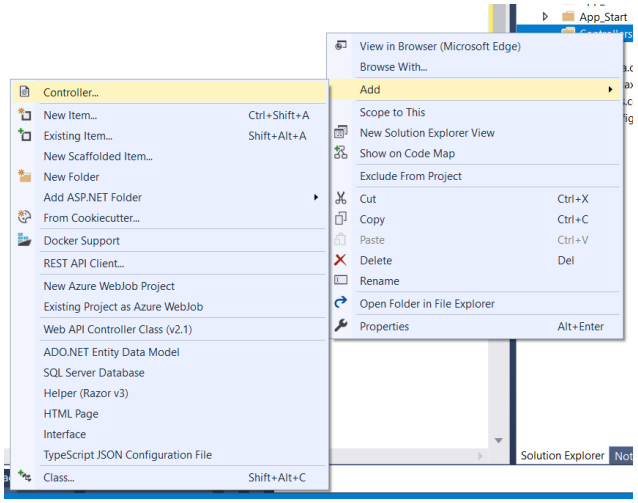
Imenujmo klasu "Knjiga". Zatim dodajmo sledeća svojstva ovoj klasi.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;

imespace mojaKnjizara.Models
{
    public class Knjiga
    {
        public int Id { get; set; }
        public string Naziv { get; set; }
        public string Kategorija { get; set; }
        public decimal Cena { get; set; }
    }

}
```

Dodavanje kontrolera

U Web API, kontroler je objekat koji rukuje HTTP zahtevom. Dodaćemo kontroler koji može da vrati listu proizvoda ili jedan proizvod a koji je zahtevan na osnovu prosleđenog Id podatka, a zatim promenu i brisanje određenog objekta.

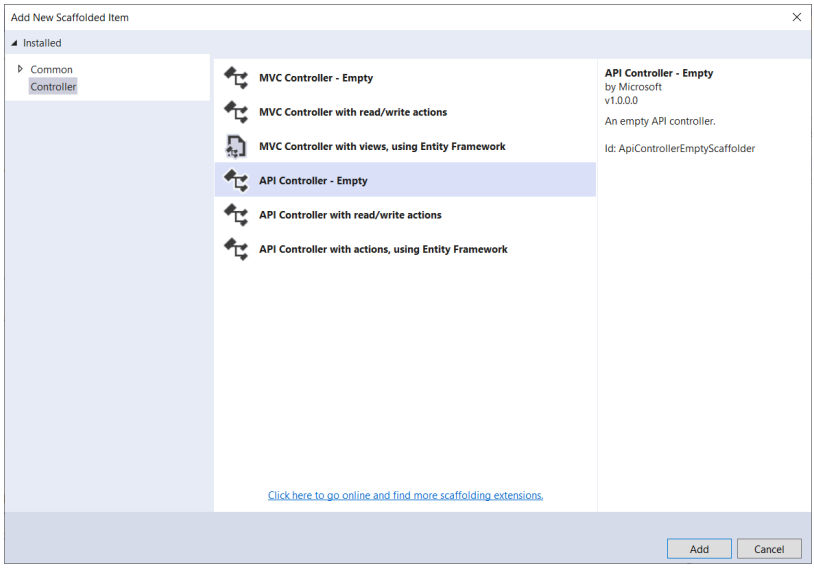
Korak 1. U prozoru `Solution Explorer`, desni-klik na `Controllers` folder, zatim odaberite **Add** a zatim odaberite `Controller`.



Slika 13.6. Dodavanje kontrolera

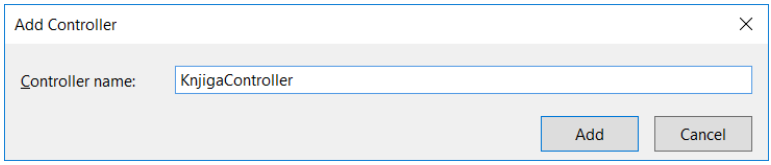
Napomena: Kreiranje WebApi kontrolera moguće je i izborom opcije Web Api Controller Class

Korak 2. Odaberite **Web API Controller - Empty**. a zatim **Add**.



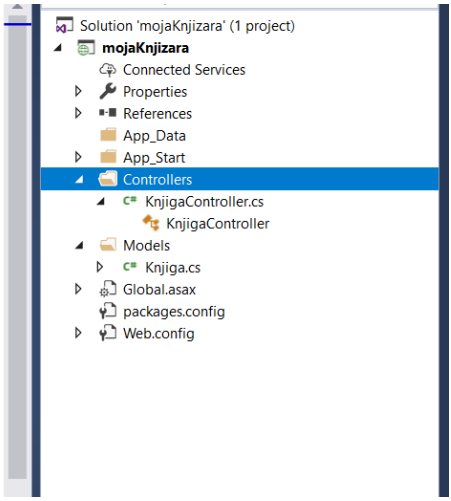
Slika 13.7. Izbor kontrolera

Korak 3. U dijalogu Add Controller imenujte kontroler KnjigaController. Odaberite Add.



Slika 13.8. Definisanje naziva kontrolera

Mehanizam automatskog kreiranja (eng. *scaffolding*) kreira jedan fajl naziva KnjigaController.cs u folderu Controllers.



Slika 13.9. Pogled na prozor `Solution Explorer`

Napomena: Nije neophodno da se kontroler postavi u folder koji se naziva Controllers. Izbor naziva foldera je pitanje konvencije i organizacije fajlova u projektu.

Drugo, da bi u kontroleru mogli da radite sa modelima obično se dodaje odgovarajući imenski prostor.

Ako ovaj fajl nije već automatski otvoren, dvostrukim-klikom na fajl on će se otvoriti za editovanje. Za početak definišite podatke modela koje ćete koristiti u kontroleru:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Net;
using System.Net.Http;
using System.Web.Http;
using mojaKnjizara.Models;

imespace mojaKnjizara.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class KnjigaController : ControllerBase
```

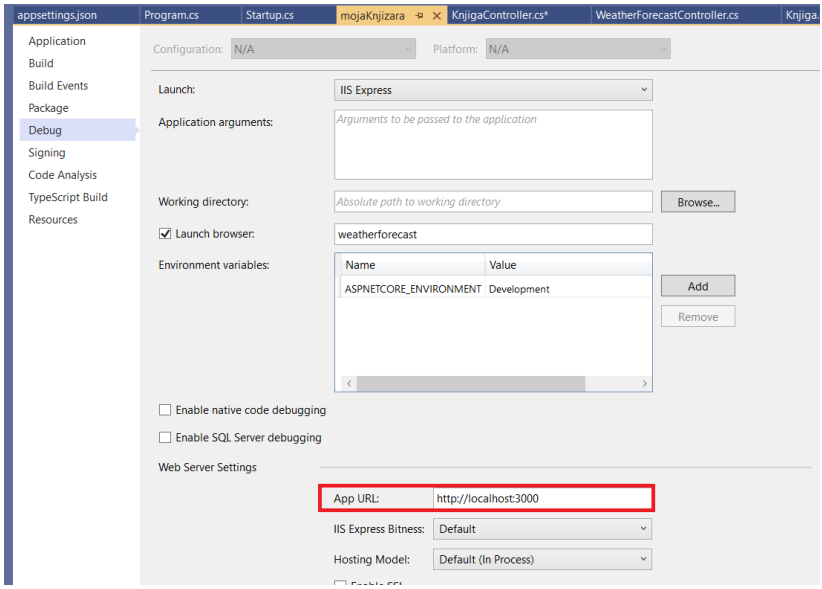
```
    {
        Knjiga[] knjige = new Knjiga[]
        {
                new Knjiga { Id = 1, Naziv = "Avanture Toma
Sojer",
                Kategorija = "Dečiji", Cena = 100 },
                new Knjiga { Id = 2, Naziv = "Rat i mir",
                Kategorija = "Klasika", Cena = 375 },
                new Knjiga { Id = 3, Naziv = "Tihi Don",
                Kategorija = "Klasika", Cena = 169.9M }
        };
    }
}
```

Get metode

Zadatak kontrolera je da prihvati zahteve sa određene URL adrese, i uz pomoć modela formira odgovor. URL mapiranje zahteva koji se obrađuju u kontroleru, definisano je dekoratorom uz klasu kontrolera

```
[Route("api/[controller]")]
```

Ovo znači da će kontroler osluškivati zahteve na adresi koja je sufiks **api/knjiga** osnove adrese. Osnovna adresa je definisana podešavanjima u aplikaciji. Dakle, desni klik na naziv projekat u prozoru **Solution Explorer**, pa onda **Properties** opcija. U sekciji **Debug** nalazi se definisana osnovna adresa koju se koristi za testiranje, pogledajte sliku.



Slika. 9.10. Podešavanje osnovne adrese aplikacije i porta za testiranje

Dakle, sve GET metode kontrolera nalaze se na istoj adresi: http://localhost:3000/api/knjiga

Metoda koja će se izvršavati za GET zahtev može imati proizvoljan naziv, ali mora imati definisan dekorator `[HttpGet]`, na primer:

```
[HttpGet] //http://localhost:3000/api/knjiga
public IEnumerable<Knjiga> Get()
{
    return knjige;
}
```

Ukoliko dekorator sadrži naziv, kao u narednom primeru, URL adresa se produžava za taj naziv, na primer:

```
[HttpGet("sve")] //http://localhost:3000/api/knjiga/sve
```

GET zahtevi mogu da budu praćeni podacima tzv. upitnim podacima. Kada su api metode u pitanju, obično se ovi parametri daju u produžetku adrese. Pogledajmo kako se piše metoda koja bi trebalo da prihvati parametar koji je **id** knjige.

```
[HttpGet("{id}")]//http://localhost:3000/api/knjiga/id=3
```

```
public Knjiga Get(int id)
{
    var knjiga = knjige.FirstOrDefault((p) => p.Id == id);
    return knjiga;
}
```

Ukoliko se podaci prihvataju sa HTML formi, onda bi trebalo da budu prihvaćeni u vidu **query** parametara, pošto se šalju tako što se URL dopunjuje parametrima na sledeći način: `?ime1=vrednost1&ime2=vrednost2`

Iako metode mogu da opišu parametre koji primaju preko izraza navedenih u vitičastim zagradama, slučaj sa upitnim parametrima se odvaja, tj. nije moguće navesti u izrazu znak pitanja pa parametri. Dakle, ako se navodi znak pitanja pa parametri tj. podaci i taj deo ne spada u osnovni URL, odnosno spada u podatke.

Ovi parametri preuzimaju se iz zahteva preko dekoratora koji se navode uz parametre metode `[FromQuery]`:

```
[HttpGet]//http://localhost:3000/api/knjiga?id=3
public Knjiga Get([FromQuery]int id)
{
    var knjiga = knjige.FirstOrDefault((p) => p.Id == id);
    return knjiga;
}
```

Napomena: Treba voditi računa da je URL ove metode onaj deo do znaka pitanja i da nije moguće imati više metoda koje će obrađivati GET zahtev za isti URL.

```
[HttpGet("{user}")]//…/api/knjiga/{user}?id=3
public Knjiga Get (string user, [FromQuery]int id)
```

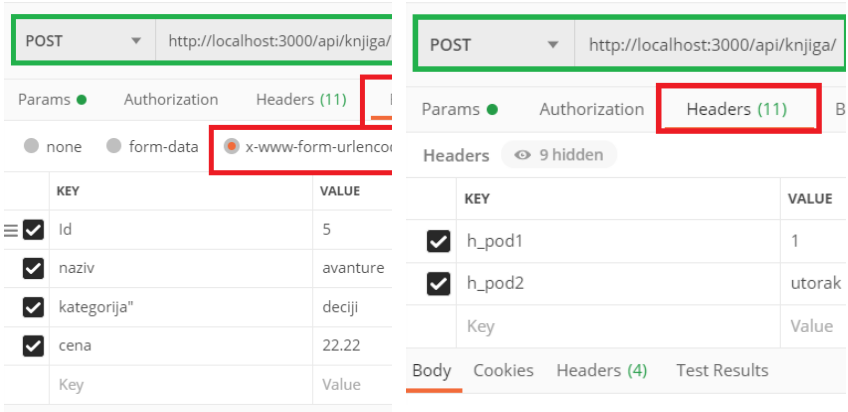
Za prethodni primer može se za parametar *user* koristiti dekorator `[FromRoute]`.

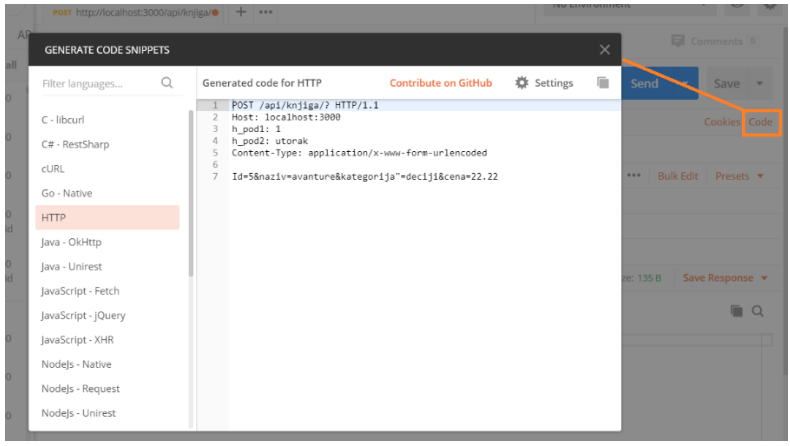
POST metode

Post metode kontrolera zadužene su za obradu POST zahteva na adresi kontrolera. Način zapisa metode koja obrađuje POST zahtev dosledan je u odnosu na GET. Na primer:

```
[HttpPost] // testirati preko Postman-a
public void Post()
{
}
```

Prosleđeni podaci servisu mogu se dobiti čitanjem toka podataka, koristeći klasu `StreamReader`. Na ovaj način čita se cela poruka i mogu se pročitati podaci iz tela ili zaglavlja poruke, zavisno od načina kako su poslati. Pretpostavimo da podatke šaljemo putem *Postman* programa koristeći i telo i zaglavlje poruke, kao na slici:



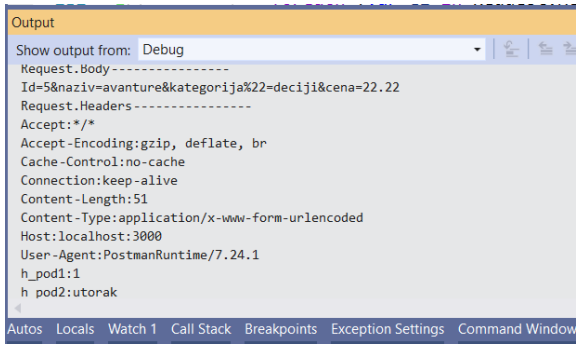


Slika 13.11. Testiranje POST zahteva slanjem podataka kroz telo i zaglavlje poruke

Za gornji primer, pogledajmo kako izgleda kod koji prihvata POST podatke i kako se može obezbediti čitanje svih podataka:

```
[HttpPost]
public async void Post()
{
    // podaci u telu poruke
    StreamReader reader = new StreamReader(Request.Body,
Encoding.UTF8);
    System.Diagnostics.Trace.WriteLine("Request.Body---------
-------");
    string bodyData = await reader.ReadToEndAsync();
    System.Diagnostics.Trace.WriteLine(bodyData);

    // podaci u zaglavlju
    System.Diagnostics.Trace.WriteLine("Request.Headers------
----------");
    foreach (var el in Request.Headers)
    {
        System.Diagnostics.Trace.WriteLine(el.Key + ":" +
el.Value);
    }
}
```

A kao odgovor u Output prozoru dobijamo:

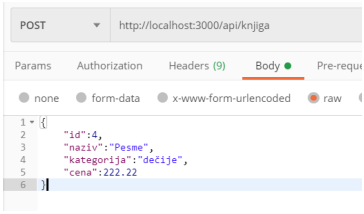


Slika 13.12. Podaci prihvaćeni i ispisani u Output prozoru IDE okruženja

Ukoliko se podaci šalju na način da se deserijalizacijom može formirati određeni objekat, onda se metoda može napisati tako da se u argumente metode navede taj objekat. Na primer:

```
[HttpPost]
public void Post([FromBody]Knjiga k)
{
    System.Diagnostics.Trace.WriteLine("Knjiga---------------
-");
    Type type = typeof(Knjiga);
    PropertyInfo[] properties = type.GetProperties();
    foreach (PropertyInfo property in properties)
    {
        System.Diagnostics.Trace.WriteLine("" + property.Ime
+ ": " + property.GetValue(k));
    }
}
```

Ovako napisana metoda je spremna za prihvat podataka iz *Postman* aplikacije ili JavaScript-a na načine kako je to prikazano na narednoj slici.




```
// JavaScript
var knjiga2 = {"Id":"22","Naziv":"Pesme",
    "Kategorija":"Poezija","Cena":222.22 }
$("#testPost2").click(()=>{
    $.post("…api/knjiga", knjiga2);
});
```

Slika 13.13. Primeri testiranja za navedenu POST metodu

Ukoliko je potrebno posebno naglasiti način formatiranja sadržaja POST poruka potrebno je koristiti *Consumes* dekorator u kome se navodi zahtevani format. Za prethodni slučaj, kada je korišćen json forma moglo je, a nije bilo obavezno, da se navede `[Consumes("application/json")]`. U slučaju da našu metodu hoćemo da prilagodimo za prijem podataka sa formi onda je potrebno navesti sledeći dekorator:

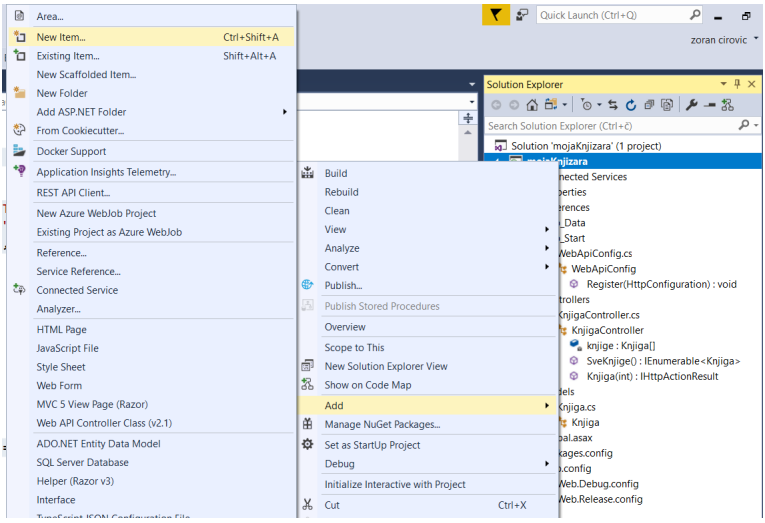
```
[HttpPost]
[Consumes("application/x-www-form-urlencoded")]
public void Post([FromForm]Knjiga value) //
{
}
```

Napomena: ASP.NET Core ne daje mogućnost obrade sirovih „row“ podataka ne drugi način osim čitanja toka podataka.

JavaScript / jQuery pozivi

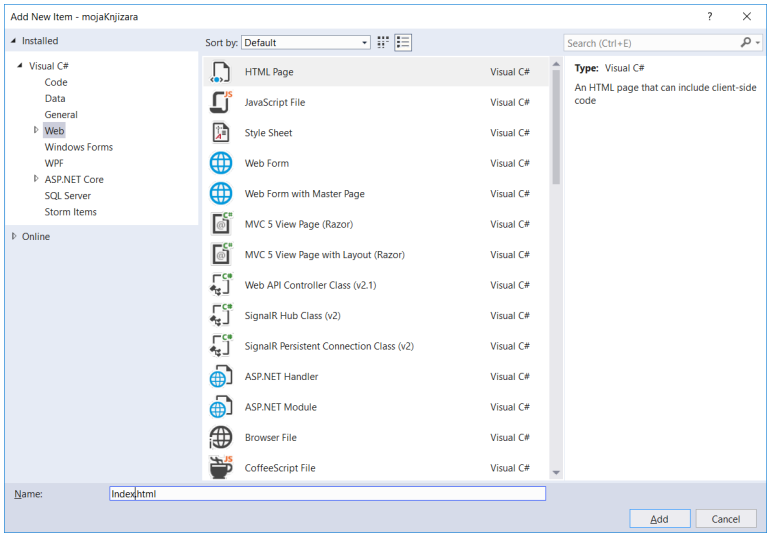
U ovom poglavlju pokazaćemo način kako da postojećem rešenju dodamo neku HTML stranicu koja koristi AJAX a koja treba da koristi kreirani veb API. Pri tome, koristićemo biblioteku jQuery kako bi napravili AJAX poziv, a zatim ažurirali stranicu sa rezultatom.

Korak 1. U prozoru `Solution Explorer`, desni-klik na projekat a zatim odaberite **Add**, zatim odaberite **New Item**.



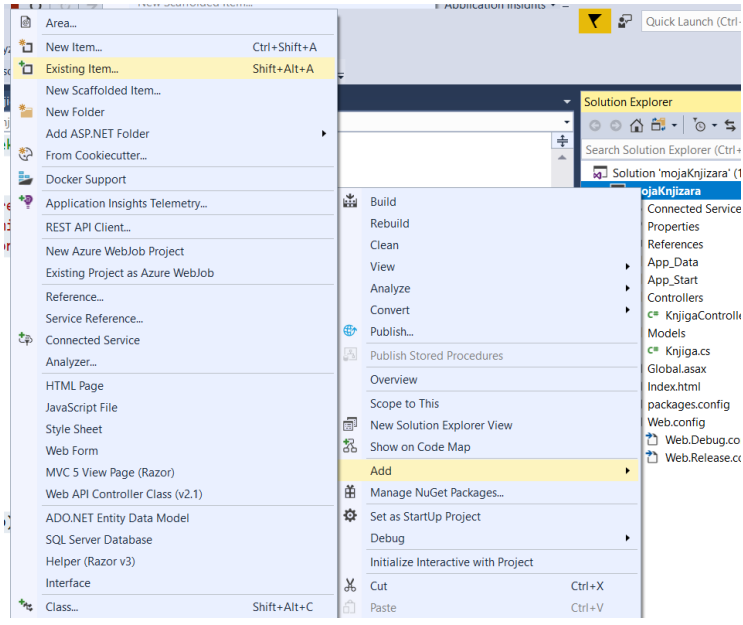
Slika 13.14. Dodavanje nove stavke

Korak 2. U dijalogu **Add New Item** odaberite čvor **Web** pod opcijom **Visual C#**, a zatim odaberite **HTML Page**. Nazovite stranicu Index.html.



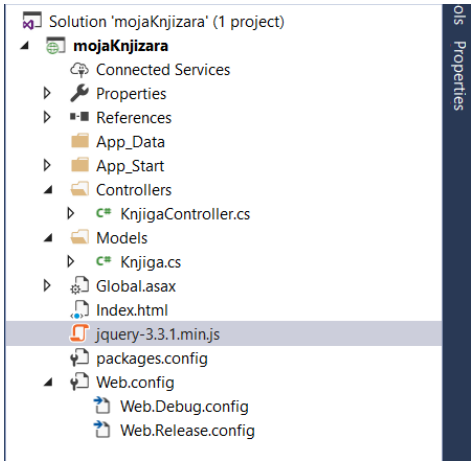
Slika 13.15. Izbor HTML stranice

Postoji više načina kako da se uključi biblioteka jQuery u veb stranicu. U ovom primeru korišćen je Microsoft Ajax CDN. Moguće je preuzeti celu biblioteku sa lokacije http://jquery.com/, a zatim je uključiti u projekat, kao na slici.



Slika 13.16. Uključivanje postojeće biblioteke projektu

Nakon dodavanja, u Project folderu pojavljuje se ubačeni js dokument, kao na slici:



Slika 13.17. `Solution Explorer` prikaz dodate biblioteke

A zatim isti možete koristiti i u aplikaciji:

```
<!--<script src="https://ajax.aspnetcdn.com/ajax/jQuery/jquery-3.3.1.min.js"></script>-->
<script src="jquery-3.3.1.min.js"></script>
```

Testiranje Get metoda

Get zahtev se može testirati neposredno preko web čitača unosom adrese metode u adres-bar čaitača. U stranicu index.html može se dodati kod za testiranje Get poziva.

Koristeći form element:

```
<form action="http://localhost:4713/api/knjiga" method="GET">
    <button id="sendGetSveKnjige1" type="submit">Get</button>
</form>
```

Koristeći Ajax – primer 1:

```
$("#btnGet").on('click', function () {
    alert("GET Request Sent");

    var request = $.ajax({
        url: "http://localhost:4713/api/knjiga",
        method: "GET",
        dataType: "json"
    }).done(function (msg) {
       console.log(msg);
   }).fail(function (jqXHR, textStatus) {
       console.log(jqXHR, textStatus);
   });
});
```

U prvom slučaju odgovor sistema je vidljiv na stranici, a u drugom se može ispitati u konzolnom prozoru (F12).

Primena u HTML stranicama

Lista

Dakle, da bi se dobila lista knjiga potrebno je da se pošalje HTTP GET zahtev na URI: "/api/knjiga". jQuery funkcija getJSON šalje AJAX zahtev. Za odgovor očekuje se niz JSON objekata. Funkcija ***done*** definiše povratnu funkciju tzv. callback koji se poziva kada je zahtev obavljen uspešno. U povratnoj metodi menjamo DOM elemente uključujući povratne informacije:

```
$(document).ready(function () {
    $.getJSON(uri)
        .done(function (data) {
            $.each(data, function (key, item) {
                $('<li>',    {    text:    formatItem(item)
}).appendTo($('#sveknjigeUL'));
            });
        });
```

```
});
```

Jedan podatak

Da bi se dobila knjiga po ID, potrebno je da korisnik učita ID I klikom na dugme pošalje zahtev tipa HTTP GET na adresu "/api/knjige/*id*", gde je *id* ID od knjige koja se traži:

```
function find() {
    var id = $('#knjigaId').val();
    $.getJSON(uri + '/' + id)
        .done(function (data) {
            $('#knjigaP').text(formatItem(data));
        })
        .fail(function (jqXHR, textStatus, err) {
            $('#knjigaP').text('Error: ' + err);
        });
}
```

Testiranje Post metoda

Post metodom klijent šalje podatke preko tela poruke. Serverska strana samo jednom čita podatke. Postoji nekoliko mogućih realizacija na serverskoj odnosno klijentskoj strani pogledajmo neke od njih.

Serverska metoda je bez argumenata. Ukoliko je naziv metode drugačiji od Post onda se dodaje dekorator `[HttpPost]` za ovu metodu.

```
public string Post()
{
   var naslov = HttpContext.Current.Request.Params["naslov"];
   var kateg  = HttpContext.Current.Request.Params["kateg"];

   return "OK";
}
```

Ukoliko se šalje jedan podatak drugi će na prijemu biti `null`. Pogledajmo tri primera zahteva.

```
<form action=". . ./api/knjiga" method="POST">
    <div><input type="text" ime="naziv"></div>
    <div><button type="submit">POST</button></div>
</form>
<form action=". . ./api/knjiga" method="POST">
    <div><input type="text" ime="naziv"></div>
    <div><input type="text" ime="kategorija"</div>
    <div><button type="submit">POST</button></div>
</form>
var knjigaZaSlanje = {
    "Id":"33",
    "Naziv":"Pesme",
    "Kategorija":"Poezija",
    "Cena":444.44
}
$("#post").on('click', function () {
    alert("POST Request Sent");
    var request = $.ajax({
        url: "http://localhost:4713/api/knjiga",
        method: "POST",
        data: knjigaZaSlanje,
        dataType: "xml"
    })
   .done(function (msg) {
       console.log(msg);
   })
   .fail(function (jqXHR, textStatus) {
       console.log(jqXHR, textStatus);
   });
});
```

Prvi se odnosi na slanje iz forme gde se šalje samo jedan podatak, drugi sa poslata dva podataka a treći preko Ajax-a. Sva tri slanja mogu se prihvatiti metodom koja je navedena.

Slanje kompleksnih tipova

Kompleksni objekti se mogu poslati preko forme ili koristeći Ajax. U oba slučaja možemo na serverskoj strani obezbediti prihvat takvih objekata u

celosti. Naravno, da bi se primljeni podaci na serveru mogli prebaciti u objekat željenog tipa neophodno je da postoji deserijalizacija za taj objekata. Dakle, na serverskoj strani dovoljno je samo da se navede objekat koji se očekuje u prihvatnoj metodi.

```
public HttpResponseMessage Post(Knjiga k)
{
    if (ModelState.IsValid && k != null)
    {
        // Create a 201 response.
        var response = new HttpResponseMessage
                       (HttpStatusCode.Created){
            Content = new StringContent("OK")
        };
        return response;
    }
    else
    {
        return Request.CreateResponse
               (HttpStatusCode.BadRequest);
    }
}
```

Ovakva metoda prihvata objekte koji imaju svojstva kojima je opisan objekat Knjiga. Na ovaj način se kreira očekivani objekat u celosti. Ukoliko nedostaju neke vrednosti objekat ima svojstva koja dobijaju podrazumevane vrednosti (npr. null ili 0)

Testiranja na klijentskoj strani koja su pokazana za slučaj metode rada sa metodom Post() mogu se koristiti i u ovom slučaju.

Slanje prostih tipova

Ako se šalje samo jedan podatak koristeći Post metodu i taj podatak mora biti pripremljen na strani klijenta u vidu objekta pri čemu se kao ključ ne navodi ništa: `{"": "Tom Sojer"}`. Na primer:

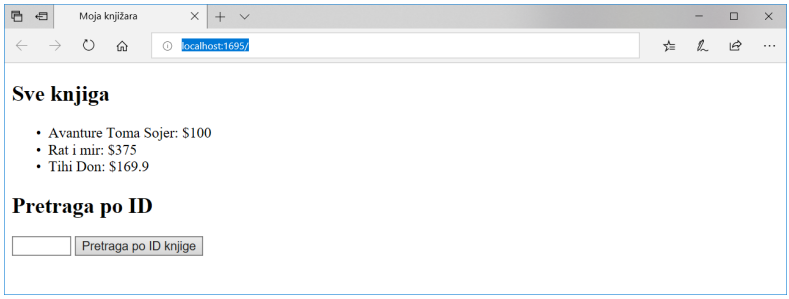
```
post2
.on('click', function () {
    alert("POST2 Request Sent");

    var request = $.ajax({
        url: "http://localhost:4713/api/values",
        method: "POST",
        data: samoTekst, //var samoTekst={"":"Tom Sojer"};
        dataType: "json"
    })
    .done(function (msg) {
        console.log(msg);
    })
    .fail(function (jqXHR, textStatus) {
        console.log(jqXHR, textStatus);
    });
});
```

Za ovako poslate podatke koji su prosti postoji mala modifikacija u kodu:

```
public void Post([FromBody]string value)
```

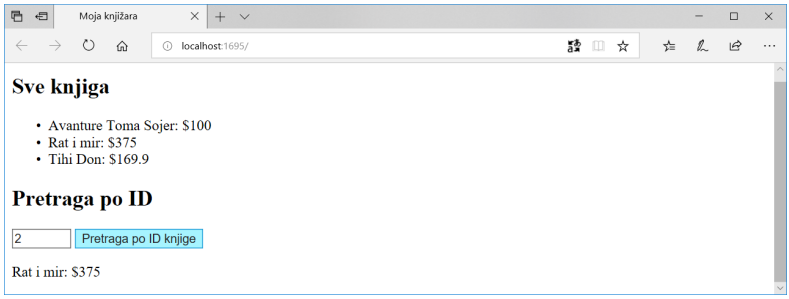
Testiranje aplikacije

Klikom na skraćenicu F5 pokreće se aplikacija sa mogućnošću debagovanja, *Debug mode*. Stranica izgleda na sledeći način:



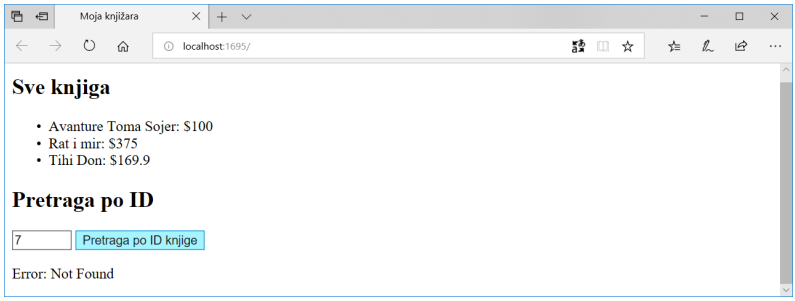
Slika 13.18. Prikaz početne stranice

Da bi se dobila knjiga po ID, unese se vrednost u polje klikne na dugme za pretragu:



Slika 13.19. Prikaz rezultata pretrage

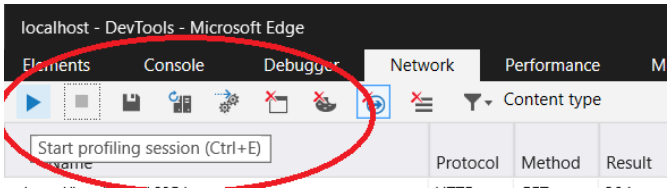
Ako se unese neispravan ID, server vraća HTTP grešku:



Slika 13.20. Prikaz pogrešnog zahteva za pretragu

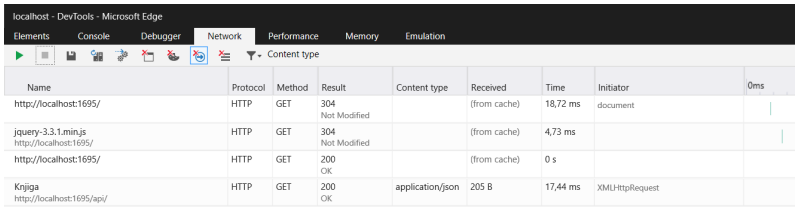
HTTP Request / Response

Kada radite sa HTTP servisima može biti veoma korisno da se vide HTTP zahtevi odnosno odgovori. To se može izvesti pomoću alatke za programere koja se pokreće tasterom **F12**. Odaberite karticu Network.



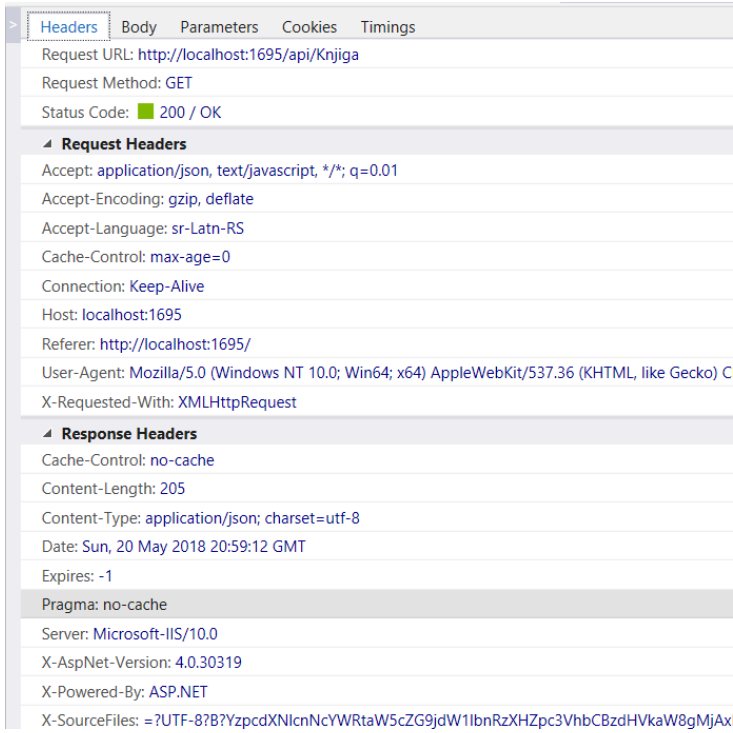
Slika 13.21. Kartice u veb čitaču korišćene za razvoj

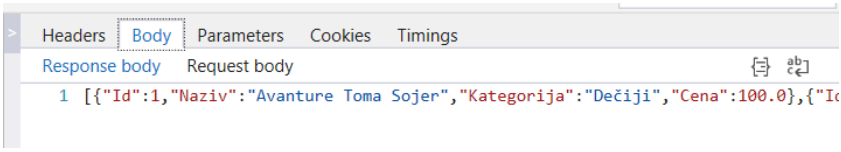
zatim pritisnite dugme za „hvatanje“ poruka. Sada se vratite se na veb stranicu i pritisnite taster F5 da ponovo učitati veb stranicu. Veb čitač će izvršiti hvatanje HTTP saobraćaja između veb čitača i veb servera. Slika pokazuje mrežni saobraćaj za stranicu:



Slika 13.22. Network kartica

Nađite poziv za URI api/Knjiga/. Odaberite tu stavku, a zatim pogledajte detalje u desnom delu prozora. Među detaljima postoje tabovi koji prikazuju request/response zaglavlja i telo poruke.





Slika 13.23. Praćenje sadržaja poruka

Pitanja i zadaci

1. Kreirati projekat za obradu HTTP zahteva, koristeći VS IDE okruženje. Obratiti pažnju na prateće opcije, kao i na dobijenu strukturu projekta.
2. Koja je uloga modela u API projektima? U postojeći projekat dodati dva modela, tako da budu međusobno povezana (npr. Knjiga i Autor, Student i Osoba….).
3. Koja je uloga kontrolera? U postojeći projekat dodati dva kontrolera koji će rukovati sa Get zahtevima.
4. Kako se definiše rutiranje zahteva?
5. Koja je razlika između POST i GET zahteva?
6. Proširiti kontrolere za obradu po jednog POST zahteva.
7. Koristeći aplikaciju *Postman* testirati napisane matode.
8. Koje se metode JavaScript-a, odnosno jQuery-a koriste za slanje http zahteva? Napisati po jedan primer za svaku.
9. Za prethodno napisane servisne metode, kreirati jednu http stranicu za testiranje.
10. Kako se može pratiti stanje promenljivih u veb čitaču?

Literatura

[1] C. Binstock, D. Peterson, M. Smith, "The XML Schema Complete Reference", Pearson 2002.

[2] A. B. Chaudhri, A. Rashid, R. Zicari, "XML Data Management: Native XML and XML-Enabled Database Systems", Addison-Wesley 2003.

[3] J. Friesen, "Java XML and JSON: Document Processing for Java SE", Apress, 2019.

[4] A. Grinberg, "XML and JSON Recipes for SQL Server", Apress, 2018.

[5] B. Benz, J. Durant, J. Durant, "XML Programming Bible", Wiley 2003.

[6] B.Marshal, " XML by example", QUE 2000.

[7] S. Patni, "Pro RESTful APIs: Design, Build and Integrate with REST, JSON, XML and JAX-RS", Apress, 2017.

[8] J. Hartwell, "C# and XML Primer", Apress, 2017.

[9] E. R. Haroldm, W. S. Means, "XML za programere", Mikroknjiga 2006., prevod 3. izdanja, orig. izd. O'Reilly

[10] https://www.w3schools.com/xml/, pristup jul.2020.

[11] https://www.tutorialspoint.com/xml/index.htm, pristup jul.2020.

[12] J. Wexler, "Get Programming with Node.js", Manning Publications 2019.

[13] B. Dayley, B. Dayley, C. Dayley, "Node.js, MongoDB and Angular Web Development", Pearson Addison-Wesley 2018.

[14] E. Wilson, I. Koroliova, "MERN Quick Start Guide", Packt Publishing 2018.

[15] A. Mead, "Learning Node.js Development", Packt Publishing 2018.

[16] F. Doglio, "Progress Your Personal Projects to Production-Ready, Apress 2018.

[17] https://www.w3schools.com/nodejs/, pristup jul 2020

[18] https://www.tutorialspoint.com/nodejs/index.htm, pristup jul 2020.

[19] J. Löwy, "Programming WCF Services", O'Reilly 2010.

[20] G. Hohpe, B. Woolf, "Enterprise Integration Patterns", Addison-Wesley 2012.

[21] https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-web-api?view=aspnetcore-3.1&tabs=visual-studio, pristup jul 2020.

[22] https://docs.microsoft.com/en-us/aspnet/core/web-api/?view=aspnetcore-3.1, pristup jul 2020.

[23] https://docs.microsoft.com/en-us/dotnet/framework/wcf/getting-started-tutorial, prisutp jul 2020.

Indeks pojmova

A

Adresa, 256
Aggregate, 242
all, 64, 65
ANY, 40
Api, 283
ATTLIST, 43

C

CDATA, 25, 26, 39, 40, 44, 47, 72
choice, 64
Content-Type, 207, 209, 210, 211, 220, 221
Controller, 287, 288
createServer, 197, 198, 202, 205
CRUD, 15, 250, 254

D

DataContract, 271, 272, 279, 280, 281
DataMember, 279, 280
Delete, 15, 251
dependencies, 158, 159, 165, 169, 177
Document, 35, 94, 242, 253, 307
DTD, 3, 18, 25, 29, 34, 35, 36, 37, 38, 39, 40, 42, 43, 46, 47, 48, 50, 72, 73, 221
Dupleks, 276
duplex, 257

E

Empty, 284, 288
EMPTY, 39, 40
enconding, 28
ENTITY, 45, 48
enumeration, 56
Exports, 12, 161
Extensible, 19

F

~~**First**~~, 5

G

Generator, 214
Get, 218
GET, 299, 300
getJSON, 299, 300
git blame, 111
git branch, 120, 121, 122, 129, 130, 136
git checkout, 94, 99, 115, 120, 121, 122, 123, 124, 125, 139, 142
git clean, 101
git clone, 142
git commit -m, 96, 128, 172

git diff, 103, 104, 105, 106, 126, 128, 312, 313
git fetch, 138, 140, 141, 142
git help, 88
git init, 92
git log, 105, 107, 108, 109, 110
`git merge`, 123, 124, 126, 128, 139, 140, 142
git pull, 137, 138, 139, 140, 142
git push, 137, 141, 142, 172
git remote add, 135, 136, 141, 142, 144, 172
git reset, 99, 113, 114, 115, 116, 117, 123, 313
git revert, 116
git rm, 100, 125
git status, 93
git whatchanged, 107, 108, 109
gitignore, 10, 91, 92, 102, 109, 117, 171

I

IDREF, 46
`IMPLIED`, 47
Insert, 15, 251
Izmena, 252

J

~~JavaScript~~, 6, 296
~~jQuery~~, 6, 283, 296, 297, 298, 299
JSON, 3, 18, 66, 67, 68, 70, 71, 72, 74, 168, 202, 219, 220, 229, 230, 233, 237, 239, 248, 252, 285, 299, 307

K

Konekcije, 249
Koren, 20
Kvalifikovano, 31

M

Markup, 19
maxExclusive, 55
maxInclusive, 55
maxOccurs, 62
Mešovit, 21
MessageContract, 271, 272
Microsoft, 297
minExclusive, 55
minInclusive, 55
minOccurs, 62
Mixed, 244
Model, 13, 164, 181, 182, 183, 223, 242, 285, 314
MongoDB, 14, 227, 229, 230, 231, 232, 233, 239, 241, 242, 243, 308
mongoose, 15, 241, 242, 243, 244, 247, 249, 250, 251, 252, 253
MVC, 5, 14, 212, 215, 222, 223, 224, 225, 314
MySql, 228

N

Neprekidni tajmeri, 185
`NMTOKEN`, 45
npm init, 166

O

ObjectId, 244
one way, 275
one-way, 257

P

Package.json, 165
`parse`, 72, 194, 196, 200, 202
parseQueryStr, 194
pattern, 56
Post, 218
POST, 205
postavlja, 256
Povezivanje, 256
Prostor, 31, 32
PUT, 205

Q

Query, 242, 250, 253
querystring, 5, 13, 193, 195, 196, 197

R

Reference, 22, 267, 269, 272, 273, 307
REPL, 151
repozitorijum, 4, 11, 82, 83, 90, 92, 93, 94, 96, 97, 106, 117, 131, 133, 135, 136, 137, 144, 145, 146, 164, 170, 171, 312
`REQIRED`, 47
Request, 304
Request objekat, 219
request-reply, 257
require, 159
Reset, 113
Response objekat, 220
restriction, 55
Rutiranje, 215

S

~~Scaffold~~, 6
Schema, 48, 49, 242, 243, 244, 245, 246, 247, 248, 253, 254, 307
Scouped, 176
sequence, 64
Serijalizacija, 16, 280
ServiceContract, 258, 263, 271, 272, 277, 278, 280
slashesDenoteHost, 194
SOAP, 15, 255, 271, 272, 281
Solution, 285, 287, 289, 296, 298
`Solution Explorer`, 287, 289, 296, 298
Sopstvene metode, 247
Sql Server, 228
standalone, 28, 29
Status poruke, 210

T

tag, 19, 20, 34, 89, 109, 112, 118, 121
Tajmer sa istekom vremena, 184
Testiranje, 303
Tip poruke, 209
Tipovi, 244
Trenutni tajmeri, 186

V

Virtuelna svojstva, 247
Visual Studio, 284

W

WSDL, 16, 256, 257, 260, 265, 266, 269, 271, 272, 278, 279, 280, 281

X

XML, 3, 18, 19, 20, 21, 22, 26, 28, 29, 30, 31, 34, 35, 36, 37, 38, 39, 40, 42, 43, 45, 46, 47, 48, 49, 50, 51, 52, 53, 59, 63, 64, 68, 69, 70, 71, 72, 73, 74, 258, 272, 279, 280, 285, 307
XSD, 3, 18, 49, 50, 53, 59, 73, 258, 271, 279

Indeks slika

Slika 1.1. XML stablo dokumenta 21
Slika 1.2. Šeme uključene u proveru validnosti Visual Studio procesora .. 54
Slika 2.1. Repozitorijum koji sadrži istoriju počev od verzije v1.0 80
Slika 2.2. Niz verzija sa posebno označenim verzijama 81
Slika 2.3. Grananje razvoja u dve grane: *Master* i *Dev* 81
Slika 2.4. Grana za ispravljanje grešaka za objavljenu verziju v-3.0 82
Slika 2.5. Grana izdvojena za razvoj nove karakteristike *Feature-X* a zatim spajanje sa glavnom granom. 82
Slika 2.6. Grananje i spajanje grana 83
Slika 2.7. Grafički prikaz preuzimanja poslednje verzije 84
Slika 2.8. Urađene izmene se vraćaju na repozitorijum 85
Slika 3.1. Git Bash i Git GUI nakon instalacije 90
Slika 3.2. Kreiranje prvog repozitorijuma 95
Slika 3.3. Status pre i posle dodavanja jednog fajla 97
Slika 3.4. Snimanje promena u repozitorijumu 98
Slika 3.5. Grafički prikaz dodavanja i snimanja novog fajla 99
Slika 3.6. Dodavanje svih dokumenata jednog foldera 100
Slika 3.7. Snimanje svih izmena 100
Slika 3.8. Brisanje fajla dokument2.txt iz repozitorijuma 102
Slika 3.9. Brisanje fajla dokument3.txt iz repozitorijuma opcijom --cached 103
Slika 3.10. Prikaz komande: **`git diff`** 106

Slika 3.11. Prikaz razlika: git diff a11fc7:dokument2.txt 284e05:dokument2.txt 107
Slika 3.12. Prikaz razlika: git diff HEAD^:dokument2.txt HEAD:dokument2.txt 108
Slika 3.13. Prikaz više različitih diff komandi 108
Slika 3.14. Prikaz oznaka revizija baziranih na HEAD referenci 109
Slika 3.15. Naknadno ubacivanje novog taga 115
Slika 3.16. Rezultat primene **`git reset --hard`** 116
Slika 3.17. Rezultat primene **`git reset`** 117
Slika 3.18. Prikaz razlike primene checkout i reset komandi 118
Slika 4.1. Primer istorije koja sadrži grananje i spajanje grana 123
Slika 4.2. Grananje i spajanje 127
Slika 4.3. Alat u okviru Visual Studio radnog okruženja za rešavanje konflikata pri spajanju 130
Slika 4.4. Istorija revizija 132
Slika 4.1. Preuzimanje zajedničke verzije, lokalne izmene i promene prvog korisnika 136
Slika 4.2. Postupak sinhronizacije 137
Slika 4.3. Sinhronizacija ostalih korisnika 137
Slika 4.4. Uporedni prikaz osnovnih funkcija u radu sa mrežnim repo... 144
Slika 4.5. Vremenski dijagram 147
Slika 4.6. Određivanje foldera sa privilegijama 148
Slika 6.1. Sinhroni / asinhroni model rada 153
Slika 6.2 Putanja i sadržaj do instaliranih paketa 155
Slika 6.3 Provera npm verzije i prelaz u REPL interaktivni mod 156
Slika 6.4. Primena REPL komandi 158
Slika 5.5. Pokretanje jedne node aplikacije 159
Slika 6.6. Izvršavanje druge aplikacije 160
Slika 5.7. Prikaz podataka primenom view komande 163
Slika 6.8. Primer primene komande search 165
Slika 6.9. Pogled na node_modules folder nakon instalacije paketa weather-js 172

Slika 6.10. Pogled pre i posle instalacije zavisnosti..... 174
Slika 6.11. Kreiranje posebnog repozitorijuma za modul..... 175
Slika 6.12. Određivanje lokacije do mrežnog repozitorijuma..... 175
Slika 6.13. Objava modula..... 177
Slika 6.14. Prikaz modula sa opsegom imena u folderu node_modules 181
Slika 7.1. Model obrade događaja 186
Slika 7.1. Slanje zahteva serveru preko veb čitača 203
Slika 7.3. Lokacije fajlova i testiranje iz veb čitača..... 208
Slika 7.4. Forma za testiranje servisa 212
Slika. 10.1. Skraćeni prikaz express komandi 220
Slika 8.1. Razvoji alat veb čitača za praćenje poruka 224
Slika 8.2. Prikaz detalja HTTP poruka preko Chrome veb čitača 229
Slika 8.3. Struktura servera zasnovana na MVC arhitekturi 230
Slika 9.1. Pogled na kreiranu mongo bazu..... 238
Slika 9.2. Pogled na kreiranu kolekciju baze 239
Slika 9.3. Pogled na kreiran dokument u kolekciji 241
Slika 12.3. Dodavanje pakete pomoću NuGet-a..... 270
Slika 12.4. Dodavanje konfiguracionog fajla 275
Slika 12.4. Pokretanje dodavanja servisne reference 279
Slika 12.5. Izbor, podešavanje i imenovanje servisne reference 279
Slika 12.6. Izbor generisanja asinhronih metoda 285
Slika 13.1. Prikaz stranice koju kreiramo primenom Web API..... 296
Slika 13.2. Formiranje novog projekta..... 297
Slika 13.3. Izbor šablona za novi projekta..... 297
Slika 13.4. Dodavanje foldera Models..... 298
Slika 13.5. Dodavanje nove klase za model 299
Slika 13.6. Dodavanje kontrolera..... 300
Slika 13.7. Izbor kontrolera 301
Slika 13.8. Definisanje naziva kontrolera..... 301
Slika 13.9. Pogled na prozor **`Solution Explorer`** 302
Slika. 9.10. Podešavanje osnovne adrese aplikacije i porta za testiranje 304

Slika 13.11. Testiranje POST zahteva slanjem podataka kroz telo i zaglavlje poruke 307
Slika 13.12. Podaci prihvaćeni i ispisani u Output prozoru IDE okruženja 307
Slika 13.13. Primeri testiranja za navedenu POST metodu 308
Slika 13.14. Dodavanje nove stavke 309
Slika 13.15. Izbor HTML stranice 310
Slika 13.16. Uključivanje postojeće biblioteke projektu 310
Slika 13.17. **`Solution Explorer`** prikaz dodate biblioteke 311
Slika 13.18. Prikaz početne stranice 316
Slika 13.19. Prikaz rezultata pretrage 317
Slika 13.20. Prikaz pogrešnog zahteva za pretragu 317
Slika 13.21. Kartice u veb čitaču korišćene za razvoj 318
Slika 13.22. Network kartica 318
Slika 13.23. Praćenje sadržaja poruka 319