Zoran Ćirović

PROGRAMIRANJE
APLIKACIJA BAZA PODATAKA

Akademija tehničko-umetničkih strukovnih studija
Odsek Visoka škola elektrotehnike i računarstva

Naslov:	Programiranje aplikacija baza podataka 1. izdanje
Autor:	dr Zoran Ćirović
Recenzenti:	dr Slobodanka Đenić dr Goran Šimić
Izdavač:	Akademija tehničko-umetničkih strukovnih studija Beograd, Starine Novaka 24, Beograd, 2026
Za izdavača:	dr Ana Savić, predsednik
Korice:	mast. inž. Ana Miletić
Štampa:	BIROGRAF COMP D.O.O. BEOGRAD
Tiraž:	50
ISBN:	978-86-6090-189-9

CIP - Каталогизација у публикацији Народна библиотека Србије, Београд
004.65:004.42(075.8)
ЋИРОВИЋ, Зоран, 1970-
Programiranje aplikacija baza podataka / Zoran Ćirović. - 1. изд. - Beograd
: Akademija tehničko-umetničkih strukovnih studija, 2026 (Beograd : Birograf Comp). - 312 str. : ilustr. ; 24 cm
Tiraž 50. - Na nasl. str.: Odsek Visoka škola elektrotehnike i računarstva. -
Bibliografija: str. 302-304. - Registri.
ISBN 978-86-6090-189-9
а) Базе података -- Програмирање
COBISS.SR-ID 187327497

Predgovor

Savremeni razvoj softvera zahteva od programera da istovremeno vladaju različitim tehnologijama – od *frontend* okvira za izgradnju interaktivnih korisničkih interfejsa, preko *backend* sistema za obradu podataka, pa sve do jezika i alata koji povezuju te svetove u jednu celinu. Ova knjiga je zamišljena kao praktičan vodič kroz neke od najvažnijih tehnologija koje se danas koriste u razvoju modernih veb aplikacija: Vue.js, .Net, LINQ, Web API i TypeScript.

Knjiga je podeljena u poglavlja koja uvode postepeno čitaoca u gradivo. Najpre se uvode novi pojmovi počev od osnovnih i instalacije razvojnog alata, preko elementarnih primera primene funkcija i klasa, pa sve do praktičnih projektnih šablona veb aplikacija. U okviru knjige obrađene su sledeće celine:

- Vue.js – obrađuje teme kreiranja reaktivnog korisničkog interfejsa, rad sa komponentama, direktivama, životnim ciklusom aplikacije i povezivanje sa podacima.

- .Net i Entity Framework Core –modelovanje baze podataka, upotreba ORM, kreiranje relacija i implementacija CRUD operacija u realnim projektima.

- LINQ –pisanje integrisanih upita nad kolekcijama i bazama podataka, koristeći lambda izraze, projekcije, filtriranja i agregacije.

- Web API – izrada RESTful servisa (eng. REpresentational State Transfer), povezivanje modela i kontrolera, obrada HTTP zahteva i implementacija filtera i autorizacija.

- TypeScript – opisuje se proširenje JavaScript-a tipovima, klasama, interfejsima, modulima i generičkim tipovima, čime se obezbeđuje sigurniji i održiviji kod.

Svaka celina sadrži:

- **teorijsko objašnjenje** pojmova,
- **praktične primere koda**,
- **pitanja i zadatke** za proveru znanja.

Na taj način, čitalac ne samo da stiče teorijsko razumevanje, već odmah primenjuje znanje u praksi.

Cilj ove knjige je da čitaoca osposobi da samostalno razvija moderne veb aplikacije koristeći kombinaciju **Vue.js** za frontend, **.Net** i **Web API** za backend, uz **LINQ** za rad sa podacima i **TypeScript** kao tipizirani jezik koji povezuje sve slojeve. Drugim rečima – nakon čitanja i rada kroz primere, bićete spremni da kreirate aplikacije koje su **moderne, skalabilne i održive**.

Beograd, 2026. Autor

Sadržaj

1. Vue.js 13

1.1 Podešavanja okruženja 15

Editor 15

Eksterno povezivanje 16

Sintakse 19

1.2 Druga okruženja 21

Veb alatke 21

1.3 Instance aplikacije 22

Jedna instanca 22

Više instanci 23

1.4 Komponente 25

Šablon 25

Primena 26

Korenska komponenta 28

Životni ciklus i zakačaljke 29

1.5 Deklarativno renderovanje 33

1.6 Direktive 34

Reaktivnost 36

Pitanja i zadaci 39

1.7 Projektni rad 40
Komponente 40
Kreiranje projekta 41
Dodavanje komponenata 44
1.8 Pitanja i zadaci 46
1.9 API sintakse 47
Options API 47
Compositions API 48
Pitanja i zadaci 49
1.10 Osnovni koncepti 49
Nazivi komponenti 50
Interpolacija 51
Dinamički atributi 54
Izrazi 56
Renderovanje kolekcije 62
Svojstava - *props* 63
Dinamički prikaz kolekcija 65
Tok podataka 67
Pitanja i zadaci 68
1.11 Računajuća svojstva i posmatrači 69
Računajuća svojstva i funkcija *computed* 69
Posmatrači i funkcija *watch* 72
Pitanja i zadaci 75
1.12 Stilizacija 75
Klase 76
Stilovi 78
Pitanja i zadaci 80
1.13 Reaktivnost i renderovanje 80
Reaktivnost – detaljno 81

Uslovno renderovanje 84
Renderovanje liste 86
Primer 88
Povezivanje sa kontrolama formi: *v-model* 89
Modifikatori 93
Pitanja i zadaci 94
1.14 Pristup podacima 94
Obrada grešaka 99
Primer: rad sa bazom 101
Pitanja i zadaci 105
1.15 Globalne komponente 106
1.16 Sopstveni događaji 107
1.17 Slotovi 110
Imenovani slotovi 112
Pristup podacima 114
1.18 Ubrizgavanje podataka 118
Funkcija `Provide` 119
Funkcija `Inject` 120
1.19 Prosleđivanje atributa 122
1.20 Pitanja i zadaci 126
Dodatak: Korišćenje TypeScript-a 127
Sekcija *template* 128
Svojstvo *props* 128
Makro *defineEmits* 130
Funkcije *ref* i *reactive* 130
Funkcija *computed* 131
2. .NET - EF 132
2.1 Objektno-relaciono mapiranje 133

Konceptni model ... 134
Tip podataka *EntityType* ... 136
Asocijacije - veze između entiteta ... 137
Navigaciona svojstva ... 138
Klase modela ... 138
U ovom primeru: ... 139
Dobavljanje podataka ... 139
2.2 Radno okruženje ... 140
Instalacija ... 140
2.3 Šabloni projekata ... 142
2.4 Integrisani jezik za upite *LINQ* ... 146
Uvod ... 146
Osnovna sintaksa upita ... 146
Funkcionalnost ... 147
Primer primene *LINQ to Objects* ... 148
Metode proširivanja ... 149
Lambda izrazi ... 150
Anonimni tipovi ... 151
Operatori ... 152
Filtriranje ... 152
Sortiranje ... 153
Skupovni operatori ... 154
Kvantifikatori ... 154
Projekcije ... 155
Izdvajanje rezultata ... 156
Združivanja ... 157
Grupisanje ... 159
Opšti operatori ... 160
Operator jednakosti sekvence ... 160
Element operatori ... 160
Konverzije ... 161
Spajanje ... 162
Agregacija ... 162
2.5 Razvoj EF projekta ... 163
Modelovanje entiteta ... 165
Alat *Server Explorer* ... 167

Kreiranje baze ... 169
Migracije ... 171
Postavka početnih podataka ... 176
Osnove upita za podatke ... 177
Alat *Scaffolding* ... 178
Relacije ... 180
Osnovne karakteristike relacija ... 181
2.7 Pristup podacima ... 186
Dva tipa upita ... 187
Rad sa povezanim podacima ... 187
2.8 CRUD operacije ... 193
Stanje ... 193
Dodavanje novog zapisa u tabeli ... 194
Izmena zapisa ... 196
Brisanje ... 197
Meko brisanje ... 197
Brisanje sa relacijama ... 198
Čitanje iz baze podataka ... 199
Klasa AutoMapper ... 203
2.9 Pitanja i zadaci ... 205
3. *Web Api* interfejs ... 207
3.1 Kreiranje projekta ... 207
Dodavanje modela ... 209
3.2 Rad sa kontrolerima ... 212
Automatsko kreiranje akcija kontrolera ... 217
3.3 JavaScript pozivi ... 221
Testiranje metoda *Get* i rešenje problema *CORS* ... 223
Primena u HTML stranicama ... 225
Lista ... 225

Jedan podatak ... 225
Post metoda ... 226
Slanje složenih podataka ... 226
Slanje prostih tipova ... 228
Testiranje ... 229
3.4 HTTP zahtev i odgovor ... 230
3.5 Filteri ... 233
Autorizacioni filteri ... 233
Sopstveni filteri ... 234
Rezultujući filteri ... 235
Filteri izuzetaka ... 235
Promena URL adrese ... 235
3.6 Pitanja i zadaci ... 237
4. TypeScript ... 238
Poređenje sa JavaScript jezikom ... 239
4.1 Instalacija i pokretanje ... 240
Instalacija ... 241
Prevođenje koda ... 241
Verzije JS koda ... 242
4.2 Promenljive ... 243
Promenljive deklarisane pomoću **var** ... 243
Promenljive deklarisane pomoću ***let*** i ***const*** ... 244
4.3 Tipovi ... 245
Tip *number* ... 245
Tip *string* ... 246
Tip *boolean* ... 247
Tip *any* ... 247
Tip *void* ... 248
Tipovi za funkcije ... 248
Unija tipova ... 249

Opcija ***strictNullChecks*** 251
Tip *Array* 253
Implicitno određivanje tipa 254
4.4 Petlje 255
Rad sa nizovima 257
4.5 Klase 260
Konstruktor 261
Instanciranje 262
Prevođenje 262
Nasleđivanje 264
Modifikatori pristupa 265
Ključna reč *Readonly* 267
Metode *get/set* 267
Statičke funkcije i svojstva 268
Apstraktne klase 269
4.6 Interfejsi 270
Sintaksa 270
Implementacija 273
4.7 Funkcije 274
Preklapanje funkcija 275
Opcioni parametri i svojstva 276
Podrazumevane vrednosti 277
Sintaksta ostatka i proširenja – tri tačke 278
Notacija funkcija strelicom 281
Povratne funkcije 281
Objekat ***this*** 282
Nabrojive liste 284
Vrednosti u nabrojivoj listi 285
Deklarisanje elemenata **enum** liste string vrednostima 286

Tip *tuple* 286
Objekat `Math` 287
Generički tipovi 289
Posebni generički tipovi 292
4.8 Moduli 295
Primena modula 296
Učitavači 299
Imenski prostori 299
Primena 301
Literatura **302**
Rečnik skraćenica **305**
Indeks pojmova **307**
Indeks slika **310**

1. Vue.js

Vue je radni okvir (eng. framework) zasnovan na programskom jeziku JavaScript, namenjen prvenstveno razvoju frontend aplikacija – interaktivnih korisničkih interfejsa - UI (eng. User Interface) i jednostraničnih aplikacija (eng. Single Page Applications). Njegova osnovna ideja je da otkloni nedostatke klasičnog razvoja u JavaScript-u i ponudi jednostavniji, fleksibilniji i moderniji pristup.

Povezivanje sa podacima u Vue-u zasniva se na MVVM obrascu (eng. Model–View–ViewModel), koji omogućava jasnu separaciju logike, prikaza i modela podataka.

Prva verzija objavljena je 2014. godine. Autor je želeo da preuzme najbolje ideje iz postojećih okvira, pre svega Angular-a, ali i da izbegne njihove nedostatke. Za razliku od React-a i Angular-a, koji se razvijaju pod okriljem velikih kompanija (Facebook i Google), Vue je projekat koji održava i unapređuje zajednica programera širom sveta.

Danas je Vue jedan od najpopularnijih okvira za razvoj korisničkih interfejsa na vebu. Mali je, lagan i prilagodljiv, a uz osnovne biblioteke postoji i veliki broj dodatnih koje proširuju njegove mogućnosti i prilagođavaju ga specifičnim potrebama.

Vue je zasnovan na komponentnom pristupu – svaka aplikacija se gradi od manjih delova (komponenti), koje se mogu kombinovati i ugnježđivati. Ovakav način rada omogućava efikasno razvijanje aplikacija oslobođenih ograničenja starih tehnologija. Dok su tradicionalne tehnologije oslanjale na DOM model (Document Object Model), Vue koristi virtuelni DOM (Virtual DOM), što donosi značajna poboljšanja u performansama i brzini renderovanja.

Ispravan rad garantovan je u veb pregledačima koji podržavaju standard ECMAScript 5 (ES5), objavljen u decembru 2009. godine. Starije verzije pregledača, poput Internet Explorer-a 8 i ranijih, nemaju podršku za ES5 i samim tim ne mogu pravilno da izvršavaju Vue aplikacije.

Vue se neprekidno menja. Izmene se prave kako bi se odgovorilo na zahteve korisnika odnosno da bi se prilagodilo drugim biblitekama u radu. Nakon izvesnog vremena odnosno nakon usvojenih izmenama događa se objava nove verzije Vue okvira. Razvoj je neprekidan tako da postoji razvojna verzija i verzija za produkciju. Preporučuje se upotreba poslednje stabilne verzije.

Tabela 1.1. – Nekoliko poslednjih značajnih verzija

Ver.	Vreme objave
3.5	September 1, 2024
3.4	December 28, 2023
3.3	May 11, 2023
3.2	Avgust, 2021
3.1	Juni, 2021
3.0	September 18, 2020
2.6	February 4, 2019
2.5	October 13, 2017

Vue se sastoji od više međusobno povezanih JavaScript (JS) fajlova, a u projekat se može uključiti na različite načine:

- Uvoz preko CDN-a (eng. *Content Delivery Network*) – jednostavno dodavanje Vue biblioteke u HTML stranicu pomoću `<script>` taga.
- Preuzimanjem i lokalnim hostovanjem fajlova – Vue se može skinuti i čuvati lokalno, pa se fajlovi uključuju direktno iz projekta.

- **Korišćenjem npm alata** (eng. *Node Package Manager*) – instalacija Vue-a kao paketa u okviru projekta.
- **Primena zvaničnog komandnog interfejsa (Vue CLI** – eng. Command Line Interface**)** – kreiranje i upravljanje projektima kroz komandnu liniju.

Pored samog Vue okvira, razvoj aplikacija često uključuje i dodatne alatke koje olakšavaju rad. Na primer, za veb pregledače zasnovane na **Chromium** tehnologiji postoji zvanično proširenje za Vue, koje omogućava lakše testiranje i debugovanje aplikacija.

Ako se Vue uključuje pomoću **CDN-a**, postupak je isti kao i kod drugih biblioteka – dovoljno je navesti odgovarajući `<script>` tag u HTML dokumentu. Nakon toga, Vue objekat postaje dostupan kao **globalni objekat**, spreman za upotrebu.

Kada se Vue instalira lokalno pomoću **npm-a**, potrebno je kreirati odgovarajuću strukturu fajlova i projekat. U tom slučaju koristi se Vue **CLI**, koji automatski generiše struktuiran projekat spreman za dalji razvoj.

Prednost ovog pristupa je što se projekat može u svakom trenutku proširiti instalacijom dodatnih npm paketa. Naravno, ovakav način rada podrazumeva i osnovno poznavanje rada sa npm alatkom.

Ukratko: Vue se može uključiti brzo i jednostavno preko CDN-a, ali za ozbiljniji razvoj preporučuje se rad sa **npm-om i Vue CLI-jem**, jer oni obezbeđuju jasnu strukturu i mogućnost daljeg proširivanja projekta.

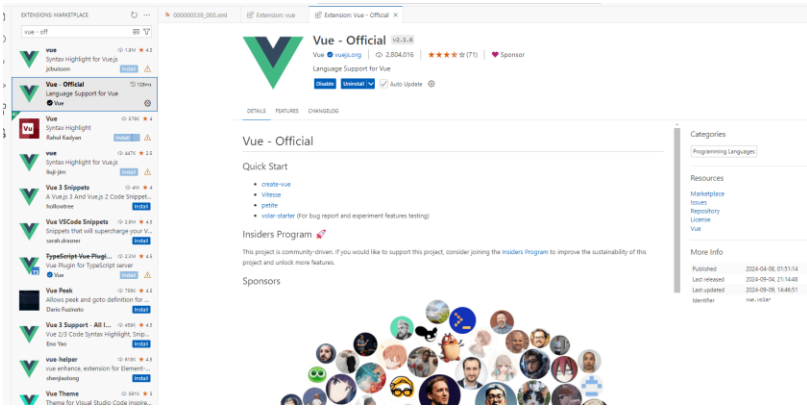
1.1 Podešavanja okruženja

Editor

Danas postoji veliki izbor dobrih editora koje možete koristiti za pisanje koda. Ovi editori nude podršku u pogledu: pravilne sintakse, lakše ispravke grešaka i

jednostavnijeg i bržeg pisanja koda. Ukratko, može se koristi editor po sopstvenom izboru, a neke preporuke su: **Visual Studio Code**, Atom ili Sublime Text.

U nastavku ove knjige, koristeće se okruženje VS Code sa instaliranom ekstenzijom Vue– *Official*. Na narednoj slici je prikazan izbor ove ekstenzije.



Slika 1.1. Pogled na ekstenziju „*Vue – Official*“ pre instalacije u VS Code

Eksterno povezivanje

Vue omogućava jednostavno povezivanje direktno iz HTML dokumenta. Na samom početku možemo pokazati kako se Vue objekat, odnosno Vue aplikacija, može kreirati i testirati na vrlo jednostavan način. Prvi korak je povezivanje postojeće HTML stranice sa Vue bibliotekom korišćenjem **CDN-a**. Nakon ovakvih primera, prelazimo na preuzimanje i instalaciju Vue radnog okvira u lokalnom okruženju.

Eksterno povezivanje se vrši pomoću `<script>` taga sa definisanim izvorom. Na taj način Vue postaje dostupan kao globalni objekat. Isti fajl je moguće preuzeti i koristiti lokalno, bez oslanjanja na CDN.

```
<script src="https://unpkg.com/vue@3/dist/vue.global.js">
</script>
```

Naravno, moguće je preuzeti isti fajl i koristiti ga lokalno. Gore navedeni link preuzima se kao **globalni Vue *build***. Evo jednog primera u celosti:

```
<script src="https://unpkg.com/vue@3/dist/vue.global.js"> </script>
<div id="app">{{poruka}}</div>
<script>
```

```
    const {createApp} = Vue;
    createApp({
        data(){
            return{
                poruka:"zdravo svete"
            }
        }
    }).mount('#app');
</script>
</body>
```

U prikazanom kodu koristi se funkcija `createApp`, koja se uvozi iz Vue objekta. Ova funkcija kreira novu Vue aplikaciju. Kao argument prima objekat koji opisuje aplikaciju – u njemu se nalazi metoda `data`(), koja vraća podatke dostupne aplikaciji.

U datom primeru koristi se samo jedna promenljiva, `poruka`, koja se ubacuje u HTML pomoću Vue sintakse sa dvostrukim vitičastim zagradama (`{{ }}`). Ovaj način rada pokazuje osnovne specifičnosti Vue sintakse, koje ćemo detaljnije obraditi kasnije.

U praksi se češće koristi modularni pristup. Pošto svi moderni veb pregledači podržavaju ES module, Vue se može uključiti kao modularni build preko CDN-a.

Umesto da se paket importuje u svakom modulu pojedinačno, bolja opcija je da se koristi `importmap`. Na taj način se izvor definiše jednom, a zatim se u ostatku koda Vue uvozi skraćenim zapisom, npr. `import { createApp } from 'vue'`.

```
<div id="app">{{ poruka }}</div>
<script type="module">
  import { createApp } from
 'https://unpkg.com/vue@3/dist/vue.esm-browser.js'
  const app = createApp({
    data() {
      return {
        poruka: 'Zdravo svete'
      }
    }
  })
```

```
  app.mount('#app')
</script>
```

U slučaju upotrebe modula, umesto da se importuje paket u svakom modulu, bolja opcija je da se koristi importovanje na jednom mestu, koristeći script tag za mapiranje - type="importmap". Tako mapiran izvor, na svim ostalim mestima, gde se Vue koristi, samo se importuje skraćenim zapisom:

```
import { createApp } from 'vue'
```

U ovom slučaju kod bi bio sledeći:

```
<head>
 <script type="importmap">
  {
   "imports": {
     "vue":"https://unpkg.com/vue@3/dist/vue.esm-browser.js"
   }
  }
  </script>
</head>
<body>
  <div id="app">{{ poruka }}</div>

  <script type="module">
    import { createApp } from 'vue'
    const app = createApp({
      data() {
        return {
          poruka: 'Zdravo svete'
        }
      }
    })
    app.mount('#app')
  </script>
</body>
```

Zaključak. Vue aplikacija se kreira u <script> elementu, gde se definišu opcije koje opisuju instancu u celosti. Aplikacija se oslanja na korensku komponentu, koja obično sadrži druge komponente. Iako su primeri jednostavni, oni jasno pokazuju osnovne principe Vue radnog okvira.

Napomena: U ovim primerima oslanjamo se na objekte, njihova svojstva i metode, a ne na ugrađene funkcije. Prikazani način pisanja zasniva se na sintaksi Options API.

Sintakse

Kada govorimo o sintaksi u Vue aplikaciji, možemo je posmatrati iz dva ugla:

- kao programski jezik koji koristimo za pisanje koda,
- ili kao skup API funkcija koje Vue nudi.

Što se jezika tiče, pored JavaScript-a, u razvoju obimnijih aplikacija često se koristi i TypeScript, zbog brojnih praktičnih prednosti koje donosi (tipizacija, bolja organizacija koda, sigurnost pri razvoju).

Kada je reč o API-ju, postoje dva pristupa:

- Options API (opciona sintaksa),
- Composition API (kompozitna sintaksa).

Do verzije 3 Vue-a postojao je samo Options API. Od verzije 3 uveden je i novi Composition API, dok je stari Options API i dalje dostupan i može se koristiti.

Mada **oba vrste API-a nude punu funkcionalnost**, ipak novi API poseduje neke osobine koje mogu da pomognu u rešavanju određenih ograničenja koja su postojala. Na primer, pomoću opcionog API sintakse teže je organizovati kod neke komponente u izvesnim slučajevima, jer zahteva da se podeli logika u više delova da bi funkcionisali, kao na primer: podaci, metode, računanja,…

Primenom kompozitne API sintakse moguće je da se organizacija koda uradi na mnogo više načina, što može da učini kod čistijim i lakšim za održavanje. Ovo je ključni razlog zbog koga većina programera prelazi na noviju sintaksu. Inače, **moguće je koristiti obe sintakse istovremeno u jednoj aplikaciji.**

Pogledajmo osnovne razlike ove dve sintakse.

Pre svega, opcioni API konfiguriše komponentu sa svojstvima i metodama. U kompozitnoj API sintaksi koriste se zakačaljke (eng. *hooks*) za iste stvari. Na ovaj način kompozitni API rešava dva glavna ograničenja koje ime opcioni API:

- grupiše relevantne delove koda koristeći zakačaljke,

- pomaže za ponovnu upotrebu koda.

Kompozitni API koristi zakačaljku `setup()` odnosno u SFC (eng. Single-File Components) varijanti rada dodatni atribut `<script setup>` koji omogućava jednu vrstu sintaktičkog šećera pri pisanju koda. Osnovne prednosti u odnosu na `<script>` sintaksu su:

- Sažetiji kod i manje šablona;
- Mogućnost deklarisanja svojstava i događaja koristeći TypeScript;
- Bolje performanse tokom izvršavanja;
- Bolji rad na tipizaciji od strane IDE.

Napomena za čitaoce koji su upoznati sa bibliiotekom React, kompozitni API pruža sličan nivo mogućnosti kao u slučaju rada sa sintaksom *React Hooks*.

U nastavku dajemo kod prvog primera („zdravo svete“) koristeći kompozitni Api.

```
<head>
     <script src="https://unpkg.com/vue@3/dist/vue.global.js">
     </script>
</head>
<body>
    <div id="app">{{poruka}}</div>
    <script>
        const {createApp, ref} = Vue;
        createApp({
            setup(){
                const poruka = ref("zdravo svete")
                return {poruka};
            }
        }).mount('#app');
    </script>
</body>
```

Kao što se vidi, kod jeste sličan, ali se više koriste ugrađene funkcije kao što je `ref` u ovom slučaju. Za čitaoca je bitno da na ovom mestu zapazi razliku u sintaksama, a svi elementi koda koji koristimo biće uvođeni postepeno u nastavku.

Zadatak. Napisati samostalno preostala dva primera koristeći kompozitni Api.

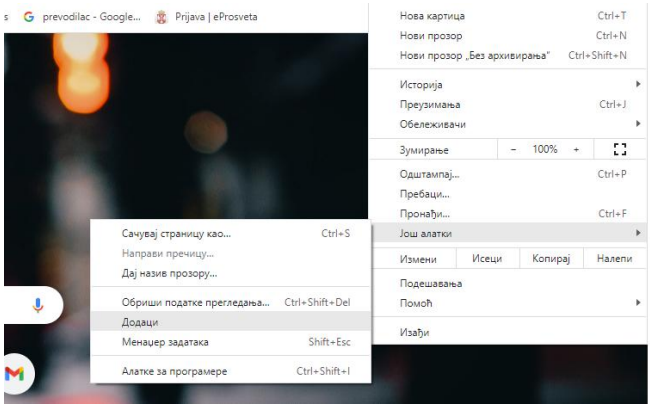
1.2 Druga okruženja

Dobar način da se počne sa učenjem odnosno testiranjem koda je upotreba veb stranica koje imaju integrisane alatke i biblioteke za brze testove. Jedna takva stranica se nalazi u okviru zvaničnog sajta, https://vuejs.org/, pod opcijom ***Playground***. Takođe, možete koristiti i ***JSFiddle*** kao opciju. Ako već koristite Node.js i koncept prevođenih aplikacija, možete se osloniti i na online razvojno okruženje **Stackblitz** - https://stackblitz.com/.

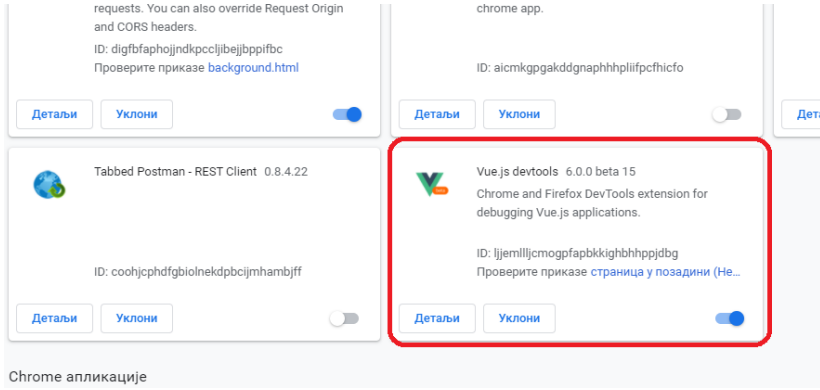
Veb alatke

Za dalje učenje u ovoj knjizi koristimo veb čitač zasnovan na *Chomium*-u, kao što je *Google Chrome*, a u okviru njega povremeno ćemo imati potrebu da koristimo razvoji alat *Chrome develper toolkit*. Osim te alatke, Vue nudi upotrebu korisnog proširenja (ekstenzije) ***Vue devtools***. Ova ekstenzija pojednostavljuje debagovanje Vue aplikacija. Na narednoj slici je prikazan način dodavanja ove alatke.

(i)



(ii)



Slika 1.2. Postavljanje „Vue.js devtools“ dodatka: (i)izbor iz menija (ii)pogled u instaliranim alatkama

1.3 Instance aplikacije

Jedna instanca

Svaka Vue aplikacija zasniva se na postojanju **bar jedne instance aplikacije**. Instanca aplikacije se kreira primenom globalne funkcije `createApp`.

```
var app = Vue.createApp({
  /* opcije root komponente */
})
```

Argument koji se prosleđuje ovoj funkciji jeste korenska komponenta. Ona predstavlja osnovu aplikacije i u sebi sadrži sve ostale komponente.
U slučaju da se koristi SFC (eng. Single-File Components) način organizovanja, korenska komponenta se obično uvozi kako bi mogla da se koristi:

```
import { createApp } from 'vue'
import App from './App.vue'
const app = createApp(App)
```

Jedan primer organizacije komponenti u aplikaciji može izgledati ovako:

```
App (root component)
├─ MyList
```

```
│   └─ MyButtons
│      ├─ DeleteButton
│      └─ EditButton
└─ MyFooter
   ├─ Notifications
   └─ Statistics
```

Više instanci

Vue ne ograničava upotrebu samo jedne instance aplikacije na stranici. Moguće je više puta pozvati `createApp` i kreirati više instanci, koje se koriste paralelno na istoj stranici. Svaka instanca ima sopstveni životni ciklus i svoju oblast važenja

```
const app1 = createApp({
  /* ... */
})
app1.mount('#container-1')
```

```
const app2 = createApp({
  /* ... */
})
app2.mount('#container-2')
```

Jedna instance aplikacije neće prikazati ništa (nekada se koristi izraz „renderovati“) sve dok se ne pozove metoda `.mount()`. Ova metoda očekuje **argument koji predstavlja kontejner aplikacije**. To može biti neki DOM (eng. Document Object Model) element odnosno CSS selector, na primer.

```
<head>
  <script type="importmap">
  {
      "imports":
       {
          "vue": "https://unpkg.com/vue@3/dist/vue.esm-browser.js"
```

```
        }
    }  </script>
    <style>
      .naslov{
        color: red;
        font-weight: bold;
      }
    </style>
</head>
<body>
    <div id="div1" class="naslov">{{ poruka }}</div>
    <div id="div2" >{{ poruka }}</div>

    <script type="module">
      import { createApp, ref } from 'vue'

      const app1 = createApp({
        setup(){
          const poruka = ref('Zdravo');
          return{
            poruka
          }
        }
      })
      app1.mount('.naslov')

      const app2 = createApp({
        setup(){
          const poruka = ref('svete!');
          return{
            poruka
          }
        }
      })
      app2.mount('#div2')
    </script>
</body>
```

Svaka korenska komponenta aplikacije poseduje sopstvena svojstva. Već smo pomenuli `data` svojstva komponente. Ova svojstva se izlažu preko instance komponente. Pogledajte sledeći primer koji bi mogao da se nadoveže na prethodni:

```
const korenskaKomponenta1 = app1.mount('.naslov')
console.log(korenskaKomponenta1.poruka)
const korenskaKomponenta2 = app2.mount('#div2')
console.log(korenskaKomponenta2.poruka)
```

Postoje različite opcije svake komponente kojima se definišu određena korisnička svojstva, na primer: `methods`, `props`, `computed`, `inject` i `setup`. Svim svojstvima jedne komponente moguće je pristupiti u šablonu komponente.

1.4 Komponente

Razvoj aplikacija u Vue-u zasniva se na **komponentnom principu**. Jedna Vue komponenta može da sadrži drugu, druga komponenta treću,... Komponente međusobno mogu da razmenjuju podatke i šalju događaje. Komponente mogu biti jednostavne ili složene. Jedna komponenta se može koristiti više puta. Sve ove osobine čine rad sa komponentama jednom od najvažnijih karakteristika Vue-a.

Šablon

Svaka Vue komponenta obično sadrži tri dela:

- **JS kod** – logika komponente,
- **HTML šablon** (eng. template) – struktura prikaza,
- **CSS stilovi** – prilagođavanje izgleda.

HTML deo komponente nazivamo šablonom ili template-om. Vue omogućava kombinovanje sopstvenog sadržaja sa DOM elementima kontejnera u kojem se aplikacija prikazuje:

```
<div id="app">
  <button @click="brojac++"> {{brojac}} </button>
</div>
```

```
<script type="module">
  import { createApp, ref } from 'vue'
  const app = createApp({
    setup(){
      const brojac = ref(0);
      return{
        brojac
      }
    }
  })
  app.mount('#app')
</script>
```

Ovakvo umetanje u spoljašnji DOM se koristi u slučaju kada se Vue primenjuje bez koraka prevođenja.

Primena

Pre nego što jedna komponenta bude primenjena u aplikaciji, ona mora biti **registrovana**. Registracija može biti **lokalna** i **globala**:

```
app.component(
  'zaglavlje', {
     template: '<div>Ovo je odeljak <b>zaglavlje</b></div>'
  })
```

U prethodnom kodu urađena je registracija komponente `'zaglavlje'`. Komponenta je veoma jednostavna i sadrži samo šablon pomoću koga se definiše prikaz.

Jednom registrovana komponenta može biti primenjena kao HTML element, naravno na više mesta, na primer:

```
<zaglavlje></zaglavlje>
<sadrzaj></sadrzaj>
<futer></futer>
<sadrzaj></sadrzaj>
```

U nastavku dat je ceo primer koji uključuje definisanje i primenu komponenti u jednoj Vue instanci.

```
<script type="module">
  import { createApp, ref } from 'vue'
  const app = createApp({
    setup(){
      const poruka = ref("Rad sa komponentama");
      return{ poruka }
    }
  })

  app.component( 'zaglavlje', {
       template: '<div>Odeljak <b>zaglavlje</b></div>'
  })
  app.component( 'sadrzaj', {
       template: '<div>Odeljak <b>sadrzaja</b></div>',
  })
  app.component( 'futer', {
       template: '<div>Ovo je odeljak <b>futer</b></div>'
  })

  app.mount('#app')
</script>
```

Komponenta može da bude definisana na takav način da se vrednost korišćenih podataka ne definiše unapred, tj. fiksno. Na primer, komponenta može da sadrži opciju `props` koja sadrži svojstva koja se definišu u trenutku definisanja komponente. Dakle, `props` su podaci koje se prosleđuju komponenti od roditeljske komponente.

Definisanje `props` sekcije podrazumeva obavezno definisanje naziva svojstava koje komponenta može da prihvati. Na primer:

```
app.component(
    'sadrzaj', {
        template: '...',
```

```
        props: ['id', 'ime', 'prezime', 'datum']
})
```

Još češće, svojstva se definišu da budu određenog tipa. Na taj način se obezbeđuje provera tipa podatak koji se prosleđuje. Na primer:

```
props:{
    id:Number,
    ime:String,
    prezime:String,
    datum:Date,
    myObj:Object
}
```

Ovako definisana komponenta daje značajno više opcija za višestruku upotrebu. Evo primera koji ilustruje upotrebu prethodno definisanih svojstava u komponenti.

```
app.component('sadrzaj', {
       template:   '<div>Ovo   je   odeljak   <b>sadrzaja</b><br>
  Svojstva su:<br>id:{{id}}<br>ime:{{ime}}<br>prezime:
  {{prezime}}<br>datum:{{datum}}<br>myobj:{{myobj}}.</div>',
  props:{
    id:Number,
    ime:String,
    prezime:String,
    datum:Date,
    myobj:Object
  }
})

------------------------------
<sadrzaj id=11 ime='Milica' prezime="Ilić" myobj="{poseduje:true,
naziv:'B'}"> </sadrzaj>
```

Korenska komponenta

Metoda `createApp` koristi jednu komponentu kao argument. Ta komponenta je osnovna tj. korenska komponenta aplikacije, a opcije koje se prosleđuju konfigurišu

aplikaciju. U aplikaciji ova komponenta je početna tj. startna tačka generisanja prikaza kada se pozove funkcija `mount` instance aplikacije.

Aplikacija se ugrađuje (često se koristi "odomaćen" izraz – mauntuje) u jedan DOM element identifikovan prosleđenim id. Na primer:

```
const KorenskaKomponenta = {
  /* opcije */
}
const app = Vue.createApp(KorenskaKomponenta)
const vm = app.mount('#app')
```

Napomena. Za razliku od većine aplikacionih metoda, metoda, `mount` ne vraća aplikaciju. Ova metoda vraća korensku komponentu instance aplikacije.

Većina realnih aplikacija se sastoji od većeg broja komponenata koje se koriste na više mesta i koje su hijerarhijski organizovane. Na primer:

```
Root Component
└─ FakturaList
   ├─ FakturaItem
   │  ├─ DeleteFakturaItem
   │  └─ EditFakturaItem
   └─ FakturaListFooter
      ├─ OsnovniPodaciFakture
      └─ OsnovniPodaciOperatera
```

Svaka komponenta ima sopstvenu instancu. Nekada se neke komponente instanciraju više puta. Sve instance komponenata jedne aplikacije dele instancu aplikacije. Metoda `mount()` treba da bude pozvana nakon svih postavki aplikacije.

Životni ciklus i zakačaljke

Životni ciklus jedne Vue komponete predstavlja niz funkcija i događaja koji se izvršavaju nad njenom instancom, (eng. lifecycle hooks). Tako na primer, životni ciklus jedne Vue komponente čine:

- inicijalizacija,
- kompajliranje šablona,
- postavljanje instace u DOM,
- ažuriranje DOM sa promenom podataka.

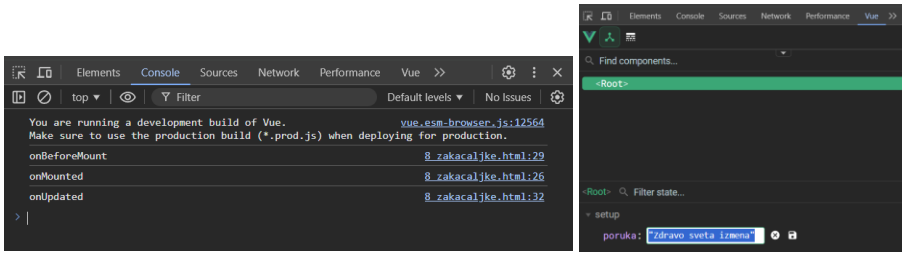
Faze životnog ciklusa omogućavaju bolje razumevanje rada komponenti i sagledavanje načina izmene komponenti od strane korisnika, odnosno umetanja sopstvenog koda u određene faze ciklusa.

Na primer, ako želimo da uradimo nešto sa tek pokrenutom komponentom, onda se obrađuje *mounted* zakačaljka komponente (eng. **mounted hook**) za postavljanje takve funkcije. Najčešće korišćene zakačaljke su:

- **mounted** - nakon pokretanja,
- **updated** – posle ažuriranja,
- **destroyed** – nakon uništavanja instance.

Na primer:

```
<script type="module">
 import {createApp,ref,onMounted,onBeforeMount,onUpdated} from 'vue'
 const app = createApp({
    setup(){
      onMounted(()=>
        console.log('onMounted')
      )
      onBeforeMount(()=>
        console.log('onBeforeMount')
      )
      onUpdated(()=>
        console.log('onUpdated')
      )
      const poruka = ref('Zdravo svete');
      return{
        poruka
      }
    }
  })
  app.mount('#app')
</script>
```



Slika 1.3. Primer sa kodom i tekstom u konzolnom prozoru

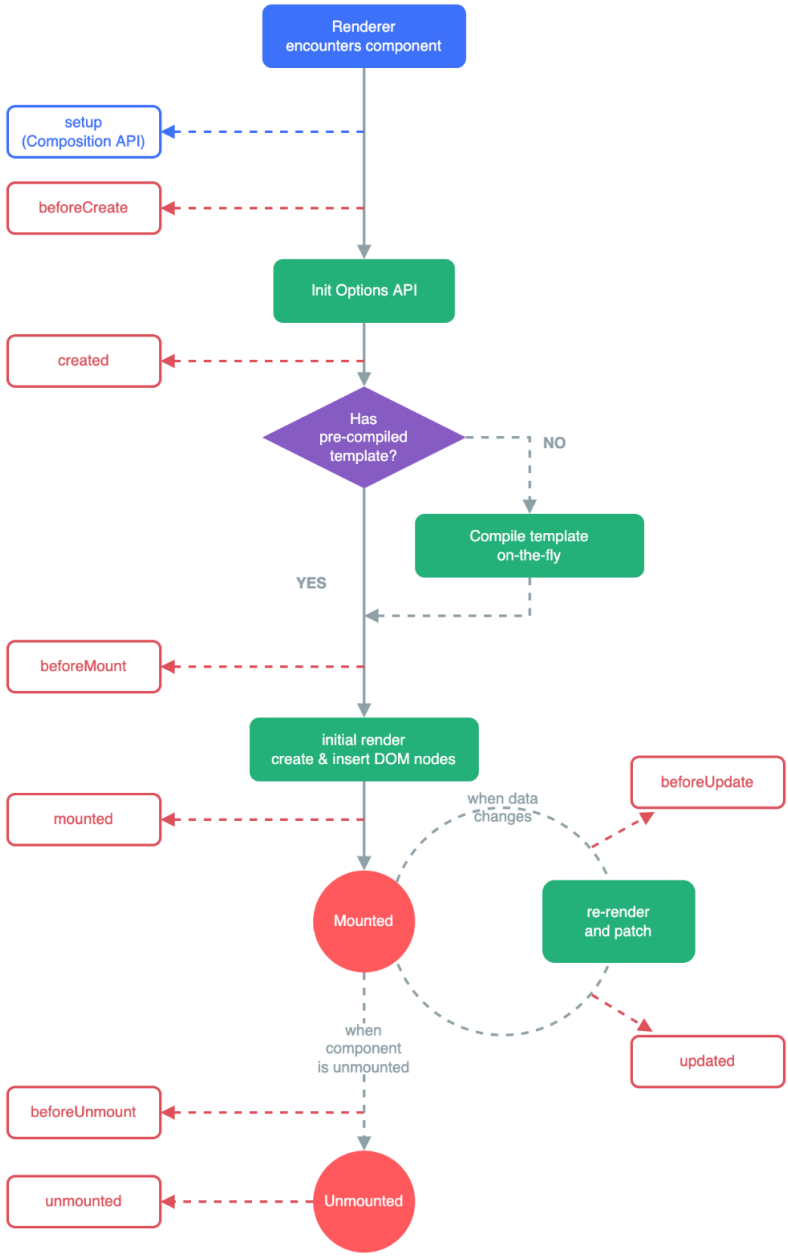
U slučaju Compositon Api funkcija u odnosu na Optional Api, postoje sledeće razlike.

1. Zakačaljke Created i BeforeCreate nemaju značaja. Umesto njih koristi se jedna **setup**. Ova funkcija predstavlja ulaznu tačku za Composition API u slučajevima kada se koristi bez prevođenja.

2. Umesto svojstava koriste se odgovarajuće funkcije:

    a. function `onMounted(callback:()=>void):void`
    b. function `onUpdated(callback:()=>void):void`
    c. function `onUnmounted(callback:()=>void):void`
    d. function `onBeforeMount(callback:()=>void):void`
    e. function `onBeforeUpdate(callback:()=>void):void`
    f. function `onBeforeUnmount(callback:()=>void):void`
    g. function
    ```
    onErrorCaptured(callback:ErrorCapturedHook):void
    type ErrorCapturedHook=(
      err: unknown,
      instance: ComponentPublicInstance | null,
      info: string
    ) => boolean | void
    ```
    h. function `onActivated(callback: () => void): void`
    i. function `onDeactivated(callback: () => void): void`

Dijagram životnog ciklusa je dat na narednoj slici, a preuzet je sa zvanične stranice Vue.org. Može se koristiti za bolje razumevanje na koji način kontrolisati i koristiti rad komponenti. Prikazujemo ga u originalnom obliku.



Slika 1.4. Životni ciklus –https://v3.Vue.org/guide/instance.html#lifecycle-diagram

1.5 Deklarativno renderovanje

U svetu frontend razvoja, postoje dva osnovna pristupa u manipulaciji prikaza korisničkog interfejsa: **imperativno** i **deklarativno renderovanje**. Razumevanje razlike između ova dva pristupa ključno je za efikasno korišćenje modernih JavaScript okvira poput Vue-a.

Imperativno renderovanje – direktna kontrola DOM-a

Imperativni pristup podrazumeva da programer eksplicitno definiše korake koje pregledač treba da izvrši kako bi prikazao određeni sadržaj. U ovom slučaju, DOM (eng. Document Object Model) se menja direktno, korak po korak. Na primer, pomoću JavaScript-a ili jQuery-a može se dodati paragraf u telo dokumenta:

```
var txt = document.createElement('p');
txt.innerHTML = 'Zdravo svete';
$('body').append(txt);
```

Ovaj način rada zahteva da programer vodi računa o svakom detalju prikaza – šta se dodaje, gde se dodaje i kada se menja. Iako daje potpunu kontrolu, može brzo postati neodrživ u složenijim aplikacijama.

Deklarativno renderovanje – opis stanja, ne koraka

Nasuprot tome, deklarativno renderovanje omogućava da se korisnički interfejs definiše kao funkcija stanja aplikacije. Umesto da direktno menjamo DOM, mi opisujemo kako interfejs treba da izgleda na osnovu podataka, a okvir (kao što je Vue) se brine o tome kako da te promene efikasno primeni.

U Vue-u, deklarativno renderovanje se oslanja na **virtuelni DOM** i **reaktivni sistem** koji automatski ažurira prikaz kada se podaci promene. Pogledajmo jednostavan primer:

```
<body>
    <div id="app">
        <zaglavlje naslov="primer" />
```

```
            {{ poruka }}
        <podnozje datum="primer" />
    </div>
    <script>
        const {createApp, ref} = Vue;
        createApp({
            setup(){
                const poruka = ref("zdravo svete")
                return {poruka};
            }
        }).mount('#app');
    </script>
</body>
```

U ovom primeru, komponenta `<zaglavlje>` prikazuje naslov, interpolacija `{{ poruka }}` prikazuje tekstualnu poruku, dok `<podnozje>` prikazuje datum. Sve ove vrednosti dolaze iz Vue instance koja se definiše u `<script>` sekciji:

Reaktivnost – ključna osobina Vue aplikacija

Jedna od najvažnijih karakteristika Vue-a je **reaktivnost**. Kada se promeni vrednost `poruka`, Vue automatski ažurira prikaz bez potrebe za dodatnim kodom. Ova automatska sinhronizacija između podataka i prikaza čini razvoj aplikacija bržim, jednostavnijim i manje sklonim greškama.

Vue koristi **tekstualnu interpolaciju** (`{{ ... }}`) za jednostavno povezivanje podataka sa prikazom, ali nudi i naprednije mehanizme kao što su **direktive** (`v-if`, `v-for`, `v-bind`, itd.) koje omogućavaju dinamičko ponašanje elemenata u DOM-u.

1.6 Direktive

U Vue radnom okviru, direktive predstavljaju specijalizovane HTML atribute sa prefiksom **v-**, koji omogućavaju dinamičko i reaktivno ponašanje elemenata u DOM stablu. One su ključni mehanizam pomoću kojeg se podaci iz Vue instance povezuju sa prikazom, bez potrebe za direktnim manipulisanjem DOM-a.

Osnovna sintaksa i ponašanje

Direktive se pišu kao atributi u HTML tagovima, sa prefiksom v-, i obično primaju JavaScript izraze kao vrednosti. Iznuzeci od ovog pravila su direktive kao što su **v-for**, **v-on** i **v-slot**, koje imaju specifičnu sintaksu i ponašanje.

Jedan od najjednostavnijih primera je **v-if**, koji omogućava uslovno renderovanje:

```
<p v-if="vidljivo">vidljivo je true ako se tekst prikazuje</p>
```

U ovom primeru, paragraf će biti prikazan samo ako je vrednost vidljivo istinita tj. `true`.

Dinamičko povezivanje atributa – v-bind*

Direktiva **v-bind** omogućava da se vrednosti HTML atributa dinamički povežu sa podacima iz Vue instance. Sintaksa koristi dvotačku (**:**) kao skraćeni zapis:

```
<p v-bind:style='stilParagrafa'>{{ poruka }}</p>
<a v-bind:href="url"> ... </a>
<div v-bind:id="'div-' + id"></div>
```

Ova direktiva je osnova za reaktivnost atributa – kada se promeni vrednost url, stilParagrafa ili id, DOM se automatski ažurira.

Umetanje sadržaja – v-text i v-html

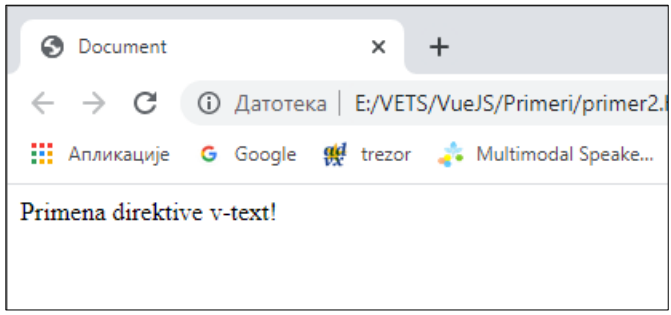
Vue štiti aplikaciju od XSS napada tako što ne dozvoljava direktno umetanje HTML sadržaja. To se obezbeđuje preko interpolacije vrednosti promenljivih (`{{ ... }}`). Takođe, direktiva **v-text** automatski preskače HTML sadržaj tj. svi potencijalno opasni tagovi se prikazuju kao običan tekst, a ne izvršavaju se kao HTML ili Java Script:

```
<div id="app">
    <p v-text="poruka"></p>
    <!-- identicno sa -->
    <!-- <p >{{poruka}}</p> -->
```

Ako je ipak potrebno umetnuti HTML sadržaj, to mora biti eksplicitno naznačeno pomoću direktive **v-html**. Ova direktiva koristi innerHTML, što znači da će se tagovi interpretirati kao pravi HTML :

```
    <p v-html="poruka"></p>
```

Ova direktiva zahteva dodatnu pažnju pošto omogućava direktno umetanje HTML-a u DOM, što može biti bezbednosni rizik ako se sadržaj ne kontroliše.



Slika 1.5. Primer povezivanja primenom direktiva

Reaktivnost

Reaktivnost je srž Vue filozofije – svaka promena podataka automatski se odražava u prikazu. Međutim, u nekim slučajevima može biti korisno prikazati podatke samo jednom, bez daljih ažuriranja. Tada se koristi direktiva `v-once`:

```
<div id="app">
   <p v-html="poruka"></p>
   <p v-once>{{ poruka }}</p>
<button onclick="Promena()">promeni</button>
```

Prvi paragraf će se ažurirati pri svakoj promeni poruka, dok će drugi ostati nepromenjen nakon inicijalnog renderovanja.

Obrada korisničkih akcija – v-on i @

Za povezivanje događaja sa metodama koristi se direktiva **v-on**.

```
<button v-on:click="onClick"></button>
```
Ili skraćenom sintaksom koristeći simbol @:

```
<button @click="onClick"></button>
```

```
<div id="app">
    <p>{{ poruka }}</p>
</div>
```

```
<button onclick="Promena()">promeni</button>
```

Skraćeni zapis koristi simbol `@`:

Izuzetno, ali je po nekada važno, ograničiti promene samo na jednom tj. prvi put. Ukoliko se u reaktivno prikazu doda atirbut v-once:

```
<p v-once>{{ poruka }}</p>
```

sada će se paragraf ažurirati samo jednom, tj. ostaće nepromenjen nakon inicijalnog renderovanja.

Nekada je potrebo pristupiti originalnom DOM događaju u linijskom rukovaocu događajem. Ovaj originialni događaj možete proslediti kroz metod koristeći specijalnu promenljivu **$event** ili upotrebiti umetnutu funkciju strelice:

```
<button v-on:click="onClick ('zdravo', $event)"></button>
```

Dinamički događaji

Vue omogućava dinamičko vezivanje događaja pomoću promenljivih u uglastim zagradama:

```
<a v-on:[eventname]="JS funkcija ili kod"> ... </a>
```

Važno je napomenuti da u uglastim zagradama **ne smeju** biti izrazi, navodnici ili razmaci. Takođe, naziv promenljive se automatski pretvara u mala slova, pa je preporučljivo koristiti **samo mala slova u imenovanju**.

Primer:

```
<script setup>
  let brojac = 0;
  let eventName = "mouseleave"; //mouseenter
  const test = ()=>{
    brojac++;
    console.log(brojac)
  }
</script>

<template>
  {{ eventName }}
  <div v-on:[eventName]="test" class="kvadrat"></div>
</template>

<style scoped>
```

```
  .kvadrat{
    width:200px;
    height: 200px;
    background-color: gold;
  }
</style>
```

Ovaj primer pokazuje kako se broj događaja može pratiti u konzoli, a naziv događaja se prikazuje direktno u interfejsu.

Događaji koji se obrađuju definisani su promenljivom `eventName`, a obrađuju se samo jednom funkcijom `test` koja obavlja uvećanje brojača:

```
  const test = ()=>{
    brojac++;
    console.log(brojac)
  }
```

Promene koje se događaju prate se izrazom:

```
{{ eventName }}
```

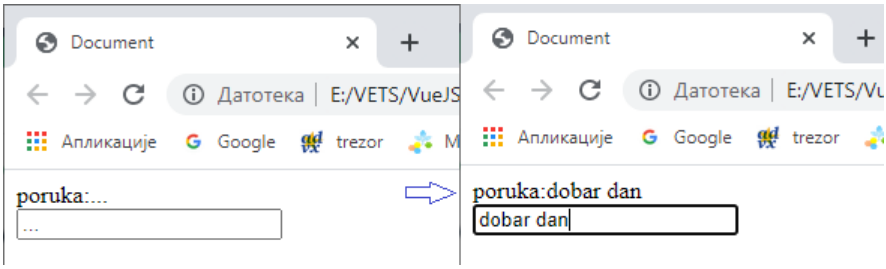
Bidirekciono povezivanje

Direktiva `v-model` omogućava **dvosmernu vezu** između podataka i prikaza. Kada korisnik unese tekst u polje, vrednost u Vue instanci se automatski ažurira:

```
<div id="app">
  poruka: {{ poruka }} <br>
  <input type="text" v-model="poruka">
</div>
```

Ova direktiva je posebno korisna za forme, jer eliminiše potrebu za ručnim osluškivanjem i ažuriranjem vrednosti.

Dakle, fokus u razvoju se postavlja na logiku ne na prikaz. Vue poseduje direktivu **v-modal** koja obezbeđuje povezivanje u oba pravca (bidirekciono), odnosno ne samo što se prikaz ažurira automatizovano, već se i podaci u modelu ažuriraju u skladu sa promenama koje se dešavaju u prikazu. (primer3b)



Slika 1.6. Primena v-model direktive u bidirekcionom povezivanju

Pitanja i zadaci

1. Šta je Vue? Objasniti osnovnu namenu i načine primene.
2. Koje se Vue sintakse?
3. Objasniti pojam korenske komponente odnosno Vue aplikacije.
4. Šta je virtuelni DOM i kako se koriste komponente u Vue aplikacijama?
5. Povezati proizvoljnu stranicu sa CDN paketima za Vue. Testirati primer aplikacije koja sadrži i prikazuje dva podatka: ime i prezime.
    a. Instalirati Vue Devtools u veb čitač.
    b. Testirati rad aplikacije.
6. Instalirati Node.js. Instalirati Vue. Verifikovati verzije.
7. Šta je životni ciklus Vue komponente?
8. Napisati primer u kome možete prikazati neke faze životnog ciklusa.
9. Koje direktive se koriste za umetanje sadržaja? Objasniti razliku.
10. Šta podrazumeva pojam „deklarativno renderovanje“?
11. Koja se direktiva koristi i kako za obradu korisničkih događaja?
12. Napiši sopstveni primer bidirekcionog povezivanja.

1.7 Projektni rad

Komponente

Već smo videli da Vue radni okvir primenjuje koncept rada sa virtuelnim DOM-om, a osnovna jedinica u tom razvoju je **komponenta**. Pri tome, jedna Vue komponenta se sastoji od nekoliko delova: HTML šablona, JS koda i CSS stilova.

Vue praktikuje način rada u kome se u jednom fajlu održavaju svi delovi jedne komponente, SFC (eng. Single-File Components). Komponente su zapisane u fajlovima koji imaju ekstenziju `.vue`. Na primer:

```
<script>
  export default {
    data() {
      return {
       pozdrav: 'Zdravo, ovo je PABP!'
      }
    }
  }
</script>
```

```
<template>
  <p class="pozdrav">{{ pozdrav }}</p>
</template>
```

```
<style>
  .pozdrav {
    color: green;
  }
</style>
```

Dakle, Vue komponenta je jedna vrsta nadogradnje kombinacije HTML, CSS i JavaScript-a koji su označeni označeni tagovima: `template`, `style` i `script`. Smešteni su u jedan fajl. Da bi primenio ovako napisan fajl tj. jedna SFC komponenta zahteva se da kod bude preveden. Neke prednosti ovakve organizacije projekta su:

- Organizacija koda po modulima i sintaksi koja je zasnovana na HTML, CSS i JavaScript.
- Povezivanje i organizacija kompnenata na način koji odgovara logici aplikacije.
- Tokom prevođenja vrši se optimizacija.
- Kreiranje jednostraničnih aplikacija.
- Upotreba unapred kompajliranih komponenata.
- IDE podrška, provera tipova, šablonskih izraza.
- Izmena modula i lako praćenje promena u kodu (eng. Hot-Module Replacement-HMR)

Kreiranje projekta

Nove verzije Vue okruženja oslanjaju se na alatku Vite. Ova alatka je lagana, kreirana od strane Evan You tj. kreatora samog Vue-a. Da bi započeli upotrebu Vite+Vue pokrenite komandu:

```
npm create vue@latest
```

Ovom komandom se instalira i izvršava create-vue, zvanični alat za kreiranje projekata (koristi se i engleski izraz *scaffolding tool*).

Kreiranje projekta predstavlja formiranje više foldera i preuzimanje neophodnih fajlova i biblioteka. S obzirom da se radi o šablonskom poslu, to se prepušta automatizovanom alatu koji pristiže uz instaliranu Vue biblioteku.

Napomena. Ovaj je zvaničan alat za kreiranje projekta i podrazumeva upotrebu Vite alatke za brzu izgranju jednokomponente podrške. Nakon pokretanja komande dobije, alat kreće u interakciju sa programerom, postavljajući mu niz pitanja u vezi projekta:

```
┌  Vue.js - The Progressive JavaScript Framework
│
◆  Project name (target directory):
```

```
│  primer
│
◆  Select features to include in your project: (↑/↓ to navigate,
space to select, a to toggle all, enter to confirm)
│  ◻ TypeScript
│  ◻ JSX Support
│  ◻ Router (SPA development)
│  ◻ Pinia (state management)
│  ◻ Vitest (unit testing)
│  ◻ End-to-End Testing
│  ◻ ESLint (error prevention)
│  ◻ Prettier (code formatting)
◆  Select experimental features to include in your project: (↑/↓
to navigate, space to select, a to toggle all, enter to
confirm)
│  ◻ Oxlint
│  ◻ rolldown-vite (experimental)
◆  Skip all example code and start with a blank Vue project?
│  ○ Yes / ● No
Scaffolding   project   in   D:\VETS\PABP\Predavanja\2025\1-8
VueJS\2\castest\primer...
│
└  Done. Now run:

   cd primer
   npm install
   npm run dev

| Optional: Initialize Git in your project directory with:

   git init && git add -A && git commit -m "initial commit"
```

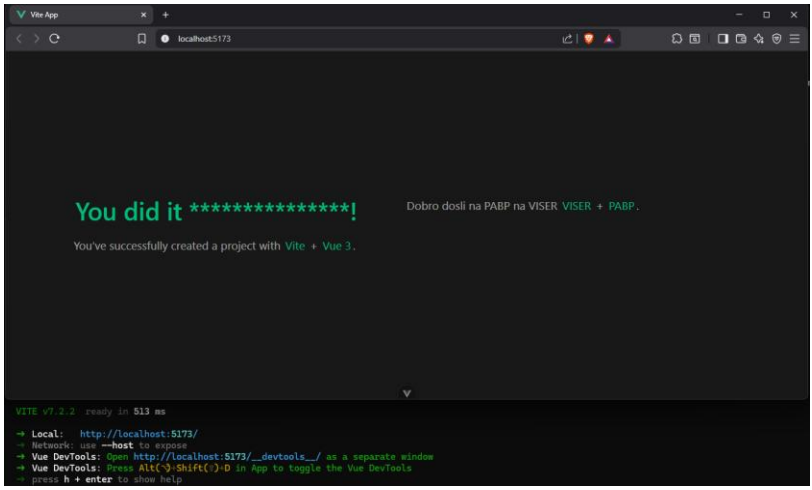
Napomena. Što se tiče ponuđenih opcija preporučuje se da korisnik ne koristi opcije za koje nije siguran šta znače i da li mu trebaju.

Pošto se završi kreiranje fajlova u folderu projekta, pratimo poslednje tri komande. Komanda `npm install` instalira potrebne pakete za projekat u folder node_modules:

Name	Date modified	Type	Size
.git	17.9.2021. 13:08	File folder	
node_modules	17.9.2021. 13:08	File folder	
public	17.9.2021. 13:08	File folder	
src	17.9.2021. 13:08	File folder	
.gitignore	17.9.2021. 13:08	Text Document	1 KB
babel.config.js	17.9.2021. 13:08	JS File	1 KB
package.json	17.9.2021. 13:08	JSON File	1 KB
package-lock.json	17.9.2021. 13:08	JSON File	472 KB
README.md	17.9.2021. 13:08	MD File	1 KB

Slika 1.7. Struktura generisanih foldera

Na kraju, pokrenemo izvršavanje komandom `npm run dev`, otvarajući neki već čitač na lokaciji koja je navedena, u našem slučaju localhost:5173, dobija se:



Slika 1.8. Pogled na pokrenuti projekat

Važno. Generisani projekat tj. pripadajuće komponente koriste sintaksu *Compositon API* odnosno `<script setup>`.

Napomena. Preporučena podrška za IDE Visual Studio Code, kao i za TypeScript je zvanična *Vue-Official* alatka. Osim sintaksne provere, ovaj dodatak nudi opcije koje identifikuju primenjeni jezik. Više informacija na:

https://Vue.org/guide/scaling-up/tooling.html#try-it-online

Dodavanje komponenata

Pogledajmo kako novu aplikaciju prilagoditi za rad sa primerima koji slede, a pre svega kako dodati nove komponente i njih uključiti u konačan izgled aplikacije.

Za početak, treba da se upoznamo sa već generisanom aplikacijom i načinom organizacije koda. Zato pokrenimo VS Code iz komandne linije sa pozicije aplikacije, dakle:

```
..\..\app>code .
```

Početna stranica koja se učitava i za koju se vezuje a zatim i menja Vue, je stranica index.html smeštena u folderu `..\app\index.html`

U okviru ove stranice nalazi se odeljak `<div id="app"></div>` u kome se podiže Vue instanca. Fajl u kome se nalazi JavaScript kod gde se podiže korenska komponenta aplikacije u **src\main.js**, čiji kod je krajnje jednostavan:

```
import { createApp } from 'vue'
import App from './App.vue'
createApp(App).mount('#app')
```

U ovom fajlu se koristi korenska komponenta **App** aplikacije koja je definisana na istoj lokaciji kao i main.js tj. nalazi se u fajlu `App.vue`. Najpre pogledajmo ceo kod pa ćemo zatim dati komentar odnosno opise.

```
<script setup>
import HelloWorld from './components/HelloWorld.vue'
import TheWelcome from './components/TheWelcome.vue'
</script>

<template>
  <header>
    <img alt="logo" class="logo" src="./assets/logo.svg"... />
    <div class="wrapper">
      <HelloWorld msg="You did it!" />
    </div>
  </header>
  <main>
    <TheWelcome />
  </main>
</template>
```

```
<style scoped>
   header { line-height: 1.5; }
   .logo {
     display: block;
     margin: 0 auto 2rem;
   }
</style>
```

Ovako generisani kod poštuje Compositon API sintaksu uz primenu razvoja SFC (eng. Single-File Components) tj. jedne komponente po fajlu.

Prva sekcija u ovoj komponenti je sekcija kojom se definiše šablon prikaza. Dakle, to je sekcija koja uključuje HTML kod sa ugrađenom Vue sintaksom, o kojoj će biti više reči kasnije. Od verzije 3, ova sekcija može imati više korenskih elemenata, ne samo sekciju `template`.

Druga sekcija je sekcija oivičena tagom `script`. U ovoj sekciji se definiše logika komponenete. Ona sadrži `export`, naredbu kojom se navedeni objekat može importovati u drugi fajl ili komponentu. Takođe, u ovom slučaju, komponeta App koristi komponentu `HelloWorld` iz fajla `'./components/HelloWorld.vue'`, što je definisano naredbom `import`.

Treća sekcija je stilska sekcija `style`. U njoj se definišu CSS stilovi koje treba primeniti u šablonu tj. pogledu za ovu komponentu.

Naravno, sve ove sekcije postoje i u primerima koji su korišćeni do sada, s tim da je ceo kod bio u jednom HTML dokumentu zajedno sa ugrađenom komponentom.

Dakle, dodavanje novih komponenata vršimo dodavanjem novih fajlova u folderu `\app\src\components.` Po uzoru na postojeći možemo dodati dva koja će samo prikazivati proizvoljan sadržaj u ovom slučaju, dakle bez posebne logike.

`Pitanje.vue`

`Zaglavlje.vue`

Zatim iste komponente treba uključiti u komponentu koja će ih koristiti. U ovom slučaju to je komponenta same aplikacije, dakle:

```
import Zaglavlje from './components/Zaglavlje.vue'
```

```
import Pitanje from './components/Pitanje.vue'
```

Istovremeno, uključene komponente treba dodati u šablon prikaza u pogledu:

```
<template>
  <zaglavlje></zaglavlje>
  <pitanje></pitanje>
</template>
```

U slučaju da ne koristimo `<script setup>` treba da te komponente postavimo u listu komponenata koje koristi komponenta `App`:

```
export default {
  name: 'App',
  components: {
    Zaglavlje,
    Pitanje
  }
}
```

Napomena. Zapazite da nećete morati ponovo da pokrećete aplikaciju da bi se kod preveo i aplikacija pravilno ažurirala, naravno ako nemate grešaka.

1.8 Pitanja i zadaci

1. Šta je znači pojam jednostranična aplikacija?
2. Opišite pojam komponenete uz konkretan primer.
3. Koje celine čine jednu komponentu?
4. Koje su prednosti komponentne organizacije?
5. Koja se komanda koristi za kreiranje početnog šablona projekta? Šta se dobija nakon pokretanja komande za instalaciju paketa?
6. Koja je korenska komponenta inicijalnog projekta?
7. Napravite projekat za prikaz stranice sa dve komponente: zaglavlje i sadrzaj.

8. Kako se uključuju komponente u korensku komponentu, a kako u druge komponente?

1.9 API sintakse

Reaktivni objekti se ponašaju se kao normalni objekti, s tom razlikom što Vue može da prati rad odnosno promene ovog objekta. Reaktivni podaci su podaci jedne komponente koji se koriste za definisanje stanja komponente, ono prati koncept reaktivnosti.

Options API i Compostions API sintakse imaju nešto drugačije pristupe po pitanju reaktivnosti.

Options API

Ovaj API za definisanje stanja koristi deklaraciju metode `data()` u glavnom objektu. Ova metoda uvek vraća objekat koji se koristi u radu komponente kao stanje komponente. Ovo je **jedino mesto** u komponenti gde se mogu dodati bilo koji tipovi podataka koji su potrebni.

```
data() {
    return {
        ime: 'Perica P',
        godine: 30,
        predmeti: ['UIT', 'ICR', 'IP'],
    }
},
computed: {
    details() {
        return `${this.ime} ima ${this.godine} godina`;
    },
    listaPredmeta() {
        return this.predmeti.join(', ');
    }
}
```

U gornjem primeru, metod data() vraća objekat koji sadrži: `ime`, `godine`, `predmeti`. Kada se ove vrednosti prikazuju u html šablonu, onda se automatski prikazuju promenjene vrednosti.

Ako je potrebno koristiti svojstva koja treba da se automatski računaju, dodaje se svojstvo `computed` unutar koga se dodaju računajuća svojstva.

Napomena: Obratiti pažnju da se podacima iz data metode pristupa obavezno uz upotrebu reference `this`.

Compositions API

U slučaju upotrebe *Composition API* funkcija, reaktivni podaci se dodaju koristeći funkcije `ref()` odnosno `reactive()`.

Ako se kao podatak komponente koristi neka prosta vrednost i to samostalno, ne kao deo većeg objekta, a pri tome, ta vrednost treba da poseduje reaktivnost, neophodno je da se koristi ugrađena funkcija `ref`. Kada se koriste ove funkcije mora se omogućiti postojanje početnih vrednosti.

```
<template>
    godine:<p>{{godine}}</p>
    osoba.godine:<p>{{osoba.godine}}</p>
    <button @click="Promena">Promena</button>
</template>
<script setup>
    import { computed, ref } from 'vue';

    const ime = ref('Perica P');
    const godine = ref(30);
    const osoba = reactive({
     ime:'Perica P',
     godine : 30
    });
    const details = computed(()=>`${ime.value}
          ima ${godine.value} godina`);
    const details2 = computed(()=>`${osoba.ime}
          ima ${osoba.godine} godina`);
    //. . .
    const predmeti = ref(['UIT', 'ICR', 'IP']);
    const listaPredmeta = computed(() => predmeti.value.join(', '));
```

```
    function Promena(){
      godine.value  = 33;
      osoba.godine = 34;
    }
</script>
```

U ovom primeru urađeno je slično kao u prethodnom, ali koristeći API sintaksu Compositon API. Vrednosti koje se dobijaju iz ovih funkcija će biti reaktivne.

Razlika je mala između korišćenih funkcija: `ref()` i `reactive()`. Funkcija `ref()` se koristi za vrednost osnovnog (primitivnog) tipa, dok se funkcija `reactive()` koristi samo za objekte.

Dodavanje računajućih svojstava ovde se postiže pomoću funkcije `computed()` koja prima kao argument tj. ulazni parametar drugu funkciju. Ta prosleđena funkcija mora da vraća neku vrednost. Više o ovoj vrsti funkcija biće kasnije reči.

Pitanja i zadaci

1. Da li se u Vue aplikacijama koristi JavaScript ili TypeScript?
2. Koje dve sintakse postoje u Vue aplikacijama? Koja je sintaksa novija?
3. Koja od dve sintakse može da uradi više ili su obe ravnopravne?
4. Navedi kako se organizuje kod u Option API i ulogu ključne reči this?
5. Navedi osnovne funkcije za kreiranje reaktivnih svojstava komponente u Composition API sintaksi.

1.10 Osnovni koncepti

Vue koristi *template* sekciju za definisanje pogleda koji koristi Vue instanca za prikaz podataka. Sekcija template sadrži kod koji podržava HTML sintaksu uz upotrebu

specifične sintakse kojom se povezuju podaci u procesu kompajliranja. Koristeći reaktivnost Vue omogućava prikaz trenutnih podataka koji su na raspolaganju komponenti.

Nazivi komponenti

VueJS koristi komponentni pristup u projektovanju UI. To znači da se aplikacija sastoji od VueJS komponenata koje se sastoje takođe od VueJS komponenta. Dakle, jedna komponenta može biti korišćena više puta i sa istom ili sličnom svrhom.

Komponenta koja se koristi mora biti registrovana koristeći **jedinstveno ime**. Jedna globalna registracija se obavlja na sledeći način:

Naziv komponente treba pažljivo birati, pogotovo ako se koristi direktno u HTML stranici. U slučaju koji mi koristimo tj. pri SFC razvoju, takođe je potrebno razumeti par osnovnih principa koji se tiču naziva komponenti i njihove upotrebe.

Komponenta se može imenovati koristeći alfanumeričke karaktere uz upotrebu velikih i malih slova. Ono što se ne može koristiti su specijalni znakovi kao na primer: belina ili crtica.

```
import my btn from '...'
import my@btn from '...'
import my-btn from '...'
import my"btn from '...'
```

Dakle, imenovanje se može izvesti primenom camelCase ili PascalCase stilom. Na primer:

```
i)   import myBtn from '...'
     <myBtn></myBtn>
     <my-btn></my-btn>
     <my-Btn></my-Btn>

ii)  import MyBtn from '...'
     <myBtn></myBtn>
     <MyBtn></MyBtn>
     <my-btn></my-btn>
     <my-Btn></my-Btn>
     <My-Btn></My-Btn>
```

Uglavnom se koristi PascalCase (npr. MyBtn) kada se registruje Vue komponenta, i to iz sledećih razloga:

- PascalCase je validan JavaScript identifikator, što olakšava uvoz i registraciju komponenti u kodu, a takođe omogućava bolju automatsku dopunu u editorima.
- Kada u šablonu pišemo `<MyBtn />`, jasno je da se radi o Vue komponenti, a ne o običnom HTML elementu ili web komponenti.
- Ovo je preporučeni stil kada radimo sa SFC ili string šablonima.

Napomena: U in-DOM šablonima (npr. kada pišemo direktno u HTML fajlu), PascalCase tagovi ne funkcionišu zbog ograničenja HTML parsiranja.

Srećom, Vue automatski prepoznaje i kebab-case (npr. `<my-btn>`) kao referencu na komponentu registrovanu kao MyBtn. To znači da možemo koristiti PascalCase u JavaScript kodu, a u šablonima birati između `<MyBtn>` i `<my-btn>`, u zavisnosti od konteksta.

Što se tiče imenovanja svojstava komponente, kada definišemo duža imena, koristimo camelCase (npr. brojStudenata) jer:

- Ne moramo da ih stavljamo pod navodnike u JavaScript objektima.
- Možemo ih direktno koristiti u šablonima (`{{ brojStudenata }}`), jer su validni JavaScript identifikatori.

Kada prosleđujemo prop-ove komponentama u šablonu, tehnički možemo koristiti camelCase, ali je preporučeno koristiti kebab-case (npr. `broj-studenata`) da bismo se uskladili sa stilom HTML atributa:

```
<MyBtn broj-studenata="248" />
```

Zaključak: koristimo camelCase u JavaScript kodu, kebab-case u HTML šablonima za svojstva, odnosno PascalCase za nazive komponenti — sve u skladu sa konvencijama jezika i boljom čitljivošću.

Interpolacija

Osnovni oblik povezivanja podataka je tekstualna **interpolacija**. Ona se koristi primenom dvostrukih vitičastih zagrada tzv. "*mustache*" sintakse, na primer:

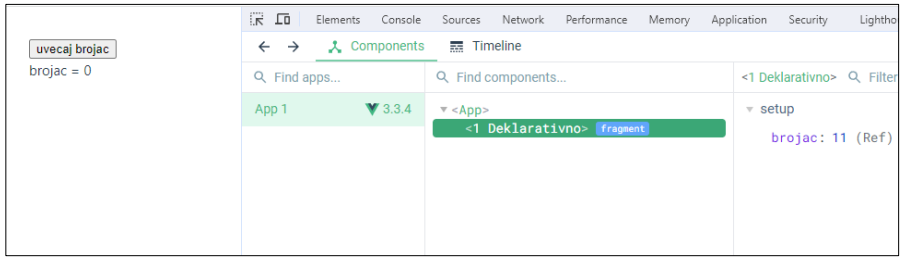
```
<div>
    {{poruka}}
</div>
```

Interpolacijom se vrši zamena celine oivičene dvostrukim vitičastim zagradama - `{{poruka}}`, sa vrednošću neke promenljive. Ukoliko se koristi neko od svojstava koje pruža osobine reaktivnosti, onda promene u prikazu prate promene vrednosti te promenljive.

Moguće je uraditi interpolaciju samo jednom (eng. *one-time interpolations*) i na taj način onemogućiti naknadne promene podataka u data sekciji koristeći direktivu `v-once`, na sledeći način:

```
<div v-once>
  brojac = {{ brojac }}
</div>
```

Nakon ovakvog povezivanja neće biti prikazana promena vrednosti brojača, mada će se njegova vrednost u pozadini menjati.



Slika 1.9. Prikaz povezivanja samo jednom kroz alat za debagovanje

Interpolacija teksta umeće čist tekstualni podatak na zadatu poziciju. Nekada je potrebno imati mogućnost umetanja HTML sadržaja umesto teksta. Za ovu potrebu ne koriste se dvostruke vitičaste zagrade već zasebna Vue direktiva `v-html`. Njena upotreba je slična upotrebi atributa u nekom HTML elementu, na primer:

```
<element v-html="poruka"></element>
```

Pošto je sadržaj `element`-a preklopljen vrednosti promenljive `poruka`, nema smisla da bude naveden. Inače, DOM sadržaj kreiran sa v-html ne stilizuju *scoped* stilovi komponente.

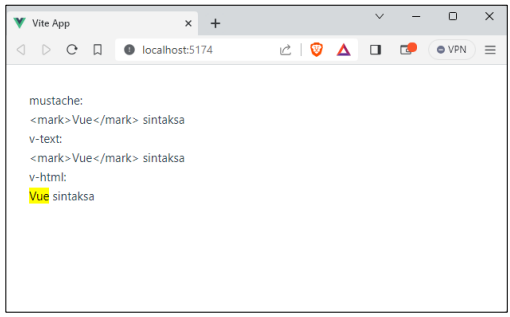
U narednom primeru prikazani su svi slučajevi primene tekstualne interpolacije. Nakon klika na dugme sadržaj se menja, a u zavisnosti od primenjene direktive razlikuje se i rezultat izmene:

```
<script setup>
const poruka = "<mark>Vue</mark> sintaksa";
</script>

<template>
  mustache:<p>{{poruka}}</p>
  v-text:<p v-text="poruka"> </p>
  v-once:<p v-once>{{poruka}}</p>
  v-html:<p v-html="poruka"></p>
</template>

<style scoped>
</style>
```

Ukoliko ovo želimo da probamo u HTML dokumentu povezujući se sa već bildovanom Vue bibliotekom i Options API:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-
scale=1.0">
  <title>Document</title>

  <script src="https://unpkg.com/vue@3/dist/vue.global.js"></script>
</head>
<body>
  <div id="app">
    <p>{{ poruka }}</p>
    <p v-once>{{ poruka }}</p>
    <p v-html="poruka"></p>
  </div>
  <script>
    var app = Vue.createApp({
```

```
      data() {
        return{
          poruka: '<mark>Vue</mark> sintaksa'
        }
      }
    }).mount("#app")
  </script>
</body>
</html>
```



Slika 1.10. Rezultat tekstualne interpolacije

Važno. Dinamičko prikazivanje proizvoljnog HTML-a predstavlja sigurnosni rizik pošto može dovesti do umetanja neželjenog sadržaja tzv. XSS ranjivosti (eng. Cross-site scripting). HTML interpolaciju treba koristiti samo na pouzdanom sadržaju, ne na sadržaju od korisnika.

Napomena: Za prethodna dva primera komponente se razlikuju dodatno, ne samo po sintaksi i načinu realizacije, već i po prirodi čuvanja podataka.

Dinamički atributi

U slučaju da želimo da koristimo dinamičke atribute na osnovu vrednosti podataka u Vue komponenti koristi se direktiva **v-bind**. Dvostruke vitičaste zagrade se ne mogu koristiti u HTML elementima na pozicijama atributa, na primer:

```
<div v-bind:id="divid"></div>
```

Ukoliko se v-bind koristi u kombinaciji sa postojećim atributima da označi povezivanje sa dinamičkim podatkom, moguće je koristiti i **skraćeni zapis** samo sa dvotačkom, na primer:

```
<div :id="divid"></div>
```

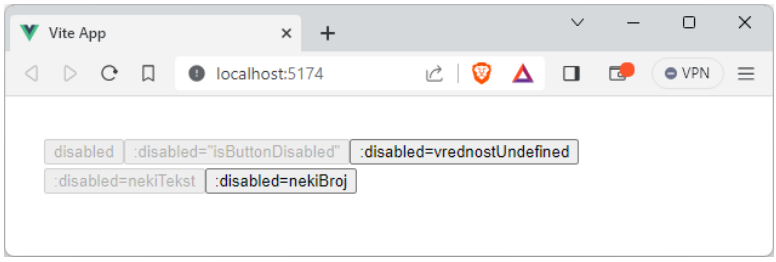
Na ovaj način `id` atribut elementa `div` je povezan dinamički sa promenljivom `divid`. Skraćeni zapis je opcioni i u praksi se često koristi.

Treba razlikovati dva slučaja povezivanja atributa.

- Prvi slučaj su **atributi koji imaju tekstualnu ili numeričku vrednost** koja se povezuje sa nekom vrednosti u komponenti.
- Drugi slučaj su **atrbuti koji imaju logičku tj. *boolean* vrednost** koji, ako su navedeni, imaju vrednost true. Ako se takvom atributu pridruži promenljiva čija je vrednost: null, undefined ili false onda atribut nije uključen u prikaz. U ostalim slučajevima je uključen.

U narednom primeru testiramo rad povezivanja sa logičkom promenljivom uz podrazumevano prevođenje različitih tipova. U ovom slučaju se vrši prevođenje tipa tzv. kastovanje u boolean promenljivu na način i po pravilima JavaScript-a.

```
<script setup>
  const isButtonDisabled = true;
  const vrednostUndefined = undefined
  const nekiTekst = "ASDF"
  const nekiBroj = 0; //ako nije nula onda je true
</script>

<template>
  <button disabled>disabled</button>
  <button :disabled="isButtonDisabled">isButtonDisabled</button>
  <button :disabled=vrednostUndefined>vrednostUndefined</button>
  <button :disabled=nekiTekst>nekiTekst</button>
  <button :disabled=nekiBroj>nekiBroj</button>
</template>
```



Slika 1.11. Povezivanje atributa logičkog tipa

Najčešći slučaj povezivanja u praksi je povezivanje sa više atributa istovremeno. Na primer ako imamo jedno dugme nad kojim se primenjuje određeni stil, tačnije klasa, a da isto dugme treba da bude onemogućeno dinamički. Evo kako bi to izgledalo na osnovu dosadašnjeg gradiva:

```
<script setup>
  const onemoguceno = false;
  const boja = 'crveno';
</script>
<template>
  <button :disabled=onemoguceno :class="boja">
     vise svojstava
  </button>
</template>
<style scoped>
  .crveno{ color:red; }
</style>
```

Nekada je jednostavnije grupisati svojstva koja se koriste. Dakle, umesto da se koristi promenljiva za svako svojstvo, možemo koristiti jedan zajednički objekat, npr objAttr. Ukoliko u tom objektu imamo svojstva istog imena kao atributi za koje treba da se uradi povezivanje, onda dobijamo veoma skraćenu i lepu sintaksu.

```
<script setup>
  const objAttr={
    disabled : false,
    class : 'crveno'
  }
</script>
<template>
  <button v-bind="objAttr">vise svojstava</button>
</template>
<style scoped>
  .crveno{
    color:red;
  }
</style>
```

U kodu je primenjen princip odmotavanja u JavaScript-u. Zapazite da je ovde neophodno korišćenje same v-bind deklaracije.

Izrazi

JavaScript izrazi mogu se koristiti na svim mestima gde se vrši povezivanje podataka Vue komponente. Evo nekoliko primera primene izraza u slučaju interpolacije:

```
{{ brojac + 1 }}
{{ poruka.toUpperCase() }}
{{ promenjeno ? 'DA' : 'NE' }}
```

Osim za sadržaj elementa, moguće je ubacivati JS izraze i u slučaju povezivanja atributa sa podacima. Evo par primera:

```
<p v-bind:style=
"promenaOmogucena ? 'color:red':'color:blue'">{{poruka}}</p>
<div v-bind:id="'div-'+id"></div>
```

Treba obratiti pažnju da se u navedenim primerima koriste delovi JavaScript koda u vidu jednog izraza, dakle ne i bilo koji JavaScript kod. Tako na primer NIJE dozvoljeno koristiti sledeći kod:

```
<!-- ovo je naredba, nije izraz: -->
{{ var x = 33 }}
<!—kontrola toka nije dozvoljena, ali jeste upotreba ternarnih operatora -->
{{ if (prikaz) { return poruka } }}
```

Osim JavaScript izraza, moguća je i upotreba postojećih funkcija na ovim mestima, na primer:

```
<span>{{puniNaziv()}}</span>
```

Napomena: Izrazi su siguran kod, omogućavaju pristup samo određenoj listi globalnih promenljivih, npr: `Math`, `Date`. Ne bi trebalo pristupati korisnički globalno definisanim promenljivama.

Vue direktive su slične atributima HTML elemenata. Imaju prefiks `v-`. Direktiva dobija neku vrednost ili JS izraz, osim `v-for` direktive koja ima nešto složeniji način upotrebe.

Neke direktive imaju dodatni argument koji se odvaja dvotačkom od same direktive. Dobar primer je često korišćena direktiva `v-bind` kojom se definiše povezivanje odgovarajućeg HTML atributa i Vue svojstva. Na primer:

```
<p v-bind:style='stilParagrafa'>{{ poruka }}</p>
<a v-bind:href="url"> ... </a>
<div v-bind:id="'div-' + id"></div>
```

Slično je i pri upotrebi direktive `v-on` kojom se definiše događaj koji se povezuje sa određenom metodom, na primer:

```
<a href="#" v-on:click="brojac ++">uvecavanje brojaca</a>
<button v-on:click="PrikaziPoruku('Zdravo')">Pozdrav</button>
```

Povezivanje se može definisati tako da i atributi budu dinamički definisani vrednostima svojstava u Vue kompoeneti. **Dinamički atributi** se definišu u uglastim zagradama. Na isti način može se definisati dinamički događaj na koji se vrši povezivanje.

Počev od Vue verzije 2.6.0 **može se koristiti JavaScript izrazi u direktivama**. U tom slučaju radi se o dinamički povezanim direktivama koje moraju biti oivičene **uglasnim zagradama**:

```
<a v-bind:[attributename]="attributevalue"> ... </a>
```

Gde je `attributename` podatka komponente koji dobija dinamičke vrednosti, a mogu se koristiti i JavaScript izrazi pri čemu se dobijena vrednost koristi kao konačna. Dakle, ako je vrednost `attributename`, `"href"`, onda je ovo povezivanje identično sa povezivanjem `v-bind:href`.

Na sličan način se može definisati i dinamički događaj, samo se ovde radi o deklaraciji `v-on`

```
<a v-on:[eventname]="JS funkcija ili kod"> ... </a>
```

Napomene:

U definisanju dinamičkih argumenata za atribute postoje ograničenja. Naime, u uglastim zagradama ne mogu se koristiti: izrazi, navodnici, razmaknice. Osim toga, naziv promenljive u uglastim zagradama bice tretirana bez velikih slova, zato bi bilo važno da takve promenljive imenujete samo **sa malim slovima**!

Sledi primer koji iliustuje ove karakteristike. Nakon prikazanog koda najpre u Options API, a zatim u Composition API, sledi detaljno objašnjenje:

```
<div id="app">
```

```
    {{inputevent}} {{brojac}}<br>
    {{buttonattribute}}:{{buttonAttributeValue}}<br>
    <input v-on:[inputevent]="inputEventHandler">
    <button
       v-bind:[buttonattribute]="buttonAttributeValue"
       v-on:click="promena">
          Promeni
    </button>
</div>

<script>
    var app = Vue.createApp({
        data() {
            return{
                brojac : 0,
                inputevent : 'click',
                buttonattribute : 'style', // samo mala slova
                buttonAttributeValue : 'width:30em'
            }
        },
        methods:{
            inputEventHandler:function(){
                this.brojac++;
            },
            promena:function(){
                console.log('promena');
                this.inputevent = 'keyup';
                this.buttonattribute = 'style',
                this.buttonAttributeValue = 'height:5em'
            }
        },
    })
    app.mount("#app");
</script>
```

Dakle, napisna je jedna komponenta koja ima sadrži dve korisničke kontrole: input i button.

```
<div id="app">
  {{inputevent}} {{brojac}}<br>
  {{buttonattribute}}:{{buttonAttributeValue}}<br>
  <input v-on:[inputevent]="inputEventHandler">
    <button v-bind:[buttonattribute]="buttonAttributeValue"
            v-on:click="Promena">Promeni</button>
```

```
</div>
```

Događaji koji se obrađuju za kontrolu input su definisani promenljivom `inputevent` dok se svi događaji obrađuju jednom funkcijom `inputEventHandler` koja obavlja samo uvećanje brojača događaja:

```
inputEventHandler:function(){
  this.brojac++;
}
```

što će korisnik moći da prati, kao i naziv obrađivanog događaja.

```
{{inputevent}} {{brojac}}
```

Događaj koji se obrađuje inicijalno je postavljen da bude `click`, a nakon klika na dugme menja se u `keyup`. Osim ove promene, klikom na dugme događaju se i druge izmene definisane metodom Promena. Klikom na dugme se definiše novi atribut `style` sa vrednosti `'height:5em'`.

```
promena:function(){
    console.log('promena');
    this.inputevent = 'keyup';
    this.buttonattribute = 'style',
    this.buttonAttributeValue = 'height:5em'
}
```

Isti kod u Compositon API sintaksi bio bi:

```
<script setup>
import {ref, reactive, computed} from 'vue'
const state = reactive({
  brojac:0,
  inputevent:'click',
  buttonattribute:'style',
  buttonattributevalue:'width:30em'
});
function Promena(){
  alert("P");
  state.inputevent = 'keyup'
  state.buttonattribute = 'style';
```

```
  state.buttonattributevalue = 'height:5em';
}
function inputEventHandler(){
  alert("P");
  state.brojac++;
}

</script>
<template>
  {{state.inputevent}} {{state.brojac}} <br>
  {{state.buttonattribute}} {{state.buttonattributevalue}} <br>

  <input type="text" @[state.inputevent] = "inputEventHandler">
  <button
   v-bind:[state.buttonattribute]="state.buttonAttributevalue"
   v-on:click="Promena">
    Promena
  </button>
</template>
<style scoped>
</style>
```

Kao što već znamo, Vue direktive počinju prefiksom `v-`. Ova oznaka zapravo identifikuje Vue specifične atribute. Zbog česte upotrebe direktiva `v-bind` ondnosno `v-on`, Vue uvodi **skraćen način upotrebe ovih direktiva**.

U slučaju `v-bind` direktive, ova direktiva se može izostaviti, tako da ostaje samo dvotačka iza koje sledi svojstvo na koje se povezivanje odnosi, na primer:

```
<div v-bind:id="id"> ... </div>
<!-- identično  -->
<div :id="id"> ... </div>
```

Slično, u slučaju obrade događa može se izostaviti direktiva `v-on:` (zajedno sa dvotačkom), umesto koje se ubacuje obrnuto a - `@`, na primer:

```
<button v-on:click="fjaObrade"> ... </button>
```

```
<!-- identično  -->
<button @click="fjaObrade"> ... </button>
```

Renderovanje kolekcije

v-for je jedna od ugrađenih Vue direktiva. Često se koristi zbog svoje jednostavnosti i efikasnosti. Namenjena je za upotrebu u **template** sekciji komponenti, gde omogućava rad sa listama stavki – odnosno, realizaciju petlje direktno u šablonu.

Kada budemo obrađivali temu uslovnog renderovanja, o ovoj direktivi će biti reči i naknadno.

Dakle, direktiva se koristi na sličan način kao i ostale direktive, u početnom tagu. Omogućava iteraciju kroz niz objekata i dinamičko renderovanje elemenata na osnovu sadržaja tog niza. Na primer, ako postoji niz objekata koji predstavljaju dane u nedelji, a opisani su u kodu:

```
var dani=[
  {
    id:1,
    naziv:'pon'
  },
  . . .]
```

onda se jedna uređena lista može kreirati dinamički koristeći `v-for`, na sledeći način:

```
<ul>
    <li v-for="dan of dani" :key="dan.id">
        {{ dan.id }}. {{ dan.naziv }}
    </li>
</ul>
```

U ovom slučaju se stavke liste formiraju u okviru petlje, a tekući objekat je dobijen kao jedan od objekata iz niza `dani`.

Treba obratiti pažnju na uvođenje dinamičke direktive `:key`. Ova direktiva omogućava jednoznačnost u prikazu i primenu naknadnih transformacija na listu objekata.

Svojstava - *props*

Važna osobina u radu komponenata jeste mogućnost da se podaci mogu prosleđivati od jedne ka drugoj komponenti. Ako je jedna komponenta dete komponenta, onda roditeljska komponenta može da prosledi podatke podkomponenti koristeći **props** direktivu.

Kada se koristi `props` direktiva, Vue komponenta automatski formira sopstveni atribut sa zadatim imenom, preko koga prima vrednosti od roditeljske komponente.

Na primer, kreirana je komponent sa direktivom props koja uključuje nekoliko svojstva: `'title'`, `'start'`, `'stop'`, `'step'` koja se koriste od strane glavne aplikacije za prosleđivanje vrednost:

```
<primer title="Test" start='2' stop='10' step='3'></primer>
```

U samoj komponenti, `props` se definiše pomoću metode (makroa) `defineProps`, **u formi niza ili objekta**. Ceo kod komponente za prethodni primer dat jeu nastavku:

```
<script setup>
  import {ref} from 'vue'
  const props = defineProps(['title','start','stop','step'])
  var brojac = ref(0);

  function uvecajBrojac(){
    //console.log(typeof(props.step));
    if(brojac.value + props.step <= props.stop)
       brojac.value += props.step;
  }
</script>

<template>
    <h2> {{ title }} </h2>
    <button @click="uvecajBrojac()">
      brojac {{brojac}}
    </button>
    <hr>
</template>
```

Podatak `'title'` tj. prosleđeni naslov je tipa String i on se ispravno prikazuje. Međutim, kada je u pitanju sabiranje prosleđenih vrednosti pokazuje se da se one nadovezuju jedna na drugu, umesto da se vrši sabiranje. **Razlog je u načinu definisanja**

svojstava kao statičkih, koja kada su tako definisana, prihvataju stringove, bez obzira na to da li izgledaju kao brojevi. Dakle, u svim dole navedenim slučajevima:

```
<primer step='1' />
<primer step=1 />
<primer step=true />
```

dobijeni tip podataka je **string**, a način njegove konverzije biće definisan pravilima JavaScript jezika u zavisnosti od načina primene i operacija.

Ukoliko se svojstvo povezuje na dinamički način, na primer:

```
<primer :step=1/>    <primer :step='1'/>    //number
<primer :step=[]/>   <primer :step='[]'/>   //object
<primer :step=true/> <primer :step='true'/> //boolean
```

Onda se prosleđena vrednost konvertuje u vrednost odgovarajućeg tipa podataka, odnosno: number, object, boolean.

Ukoliko se dinamičkim povezivanjem pridruži promenljiva, onda se vrednost te promenljive koristi. Takođe, ako se pridružuje string koji predstavlja naziv promenljive, takođe se koristi vrednost te promenljive. Na primer:

```
<script>
  var korak = 9
  var test = 'test'
</script>
<template>
<primer :step=korak/> <primer :step='korak'/> //number
<primer :step=test/>  <primer :step='test'/>  //string
<primer :step=xyz/>   <primer :step='xyz'/>   //undefined
```

Višestruka svojstva kao objekat

Višestruka svojstva se mogu definisati zajedno **kao jedan objekat**, što je isto kao da je istovremeno definisano više pojedinačnih svojstava. Prosleđivanje vrednosti se može vršiti pojedinačno ili se mogu prosleđivati sve vrednosti odjednom pomoću objekta. Pri definisanju se navodi:

- **tip** podataka koji odgovara svojstu,
- vrednost `null` ili `undefined` ako ga ne navodimo, ili se navodi
- **podrazumevana vrednost**, koja automatski sadrži i tip podataka.

Na primer:

```
const props = defineProps({
  title : String,
  start: Number,
  stop: Number,
  step: 1
})
```

Ovaj način definisanja svojstava sadrži validaciju po tipu podatka koji je definisan. U slučaju nepodudaranja tipova podataka, u konzolnom prozoru javlja se tekst upozorenja.

Dinamički prikaz kolekcija

Jedan od čestih slučajeva u radu sa Vue komponentama jeste dinamičko renderovanje liste podataka. Kada su podaci struktuirani kao niz objekata sa ključevima, recimo da sadrže svojstvo `id` i prateće vrednosti, oni se mogu proslediti komponentama pomoću `props` direktive i prikazati pomoću `v-for`.

Na primer, ako su podaci struktuirani kao niz objekata:

```
podaci:[{
   id:1,
   naslov:'Uvecaj',
   pocetnavr:0
},...]
```

Takođe, pretpostavimo da već imamo kreiranu komponentu `MyBtn.vue` koja sadrži svojstava: `id`, `naslov` i `pocetnavr`. U tom slučaju, za odgovaraći niz `podaci` može se kreirati dinamička lista kontrola `myBtn`.

Evo kako bi izgledao ceo kod zajedno sa jednom konkretnom listom od tri objekta:

```
import MyBtn from './MyBtn.vue'
let podaci = [{ id:1, naslov:'Uvecaj', pocetnavr:0 },
       { id:2, naslov:'brojac', pocetnavr:10 },
       { id:3, naslov:'Uvecaj brojac', pocetnavr:100 }]
</script>
```

```
<template>
  <div id="app">
    <MyBtn v-for="p in podaci"
          :key="p.id"
          :naslov="p.naslov"
          :pocetnavr="p.pocetnavr">
    </MyBtn>
  </div>
</template>
```

Kao što se vidi u ovom primeru, osim direktive `v-for`, pri dinamičkom povezivanju koristili smo dinamička svojstva za prosledjivanje vrednosti. Ovo će biti neophodno u slučajveima kada se unapred ne poznaje broj objekata koje kreiramo, odnosno kada ne poznajemo unapred sadržaj koji se prikazuje.

Ukoliko je svojstvo komponente tipa **boolean**, vrednost koja se prosleđuje se eksplicitno navodi ili se samo navodi atribut ako se prosleđuje vrednost `true`. Na primer: `<myBtn is-visible />`

Ako se navodi niz ili objekat, onda je standardno zadavanje:

```
<myBtn test-ids="[2,4,6]" />
<myBtn test-obj="{ime:'Perica', godine:44}" />
```

Važno. Povezivanje više vrednosti pomoću objekata može da se skrati ako su svojstva objekta istog imena kao i svojstva koja se prenose:

```
const props = defineProps({
    'btnId':undefined,
    'naslov' : undefined,
    'pocetnaVrednost': undefined
})
```

a podaci su:

```
let podaci = [{
        btnId:1,
        naslov:'Uvecaj',
        pocetnaVrednost:0
    },...]
```

Tada se skraćeno može pistati:

```
<myBtn v-for="podatak in podaci" v-bind="podatak"></myBtn>
```

Dakle, ceo kod bi bio značajno kraći i jasniji:

```
import myBtn from './myBtn.vue'
let podaci = [
    {id:1, naslov:'Uvecaj', pocetnavr:0},
    {id:2, naslov:'brojac', pocetnavr:10},
    {id:3, naslov:'Uvecaj brojac', pocetnavr:100},
  ]
</script>

<template>
  <div id="app">
    <myBtn v-for="podatak in podaci"
          :key="podatak.id"
          :naslov="podatak.naslov"
          :pocetnavr="podatak.pocetnavr">
    </myBtn>
  </div>
</template>
```

Tok podataka

Sva svojstva se koriste u prenosu podataka samo u jednom smeru. Dakle, samo kada se podatak promeni u roditeljskoj komponenti biće promenjen i u komponenti koja je dete tj. koja prihvata određeno svojstvo.

Zbog toga, svaki pokušaj promene tj. dodele podataka svojstvu biće praćeno upozorenjem od strane prevodioca. Ipak postoje dva slučaja koji su izuzeci i koji su prihvatljivi:

- Postavljanje početne vrednosti. U ovom slučaju se vrednost koristi tj. kopira kao inicijalna vrednost.

  ```
  const props = defineProps(['initialCounter'])
  const counter = ref(props.initialCounter)
  ```

- Promena objekata/nizova svojstava.

  Ako se objekti ili nizovi koji se prosleđuju kao props, iako podređena komponenta ne može da mutira vezivanje propa, moći će da **mutira**

ugnežđena svojstva objekta ili niza. To je zato što se u JavaScript-u objekti i nizovi prosleđuju po referenci pa je i za Vue nerazumno skupo rešenje kojim bi sprečio takve promene.

Glavni nedostatak ovakvih promena je što omogućava podređenoj komponenti da utiče na roditeljsko stanje na način koji nije očigledan za roditeljsku komponentu, a to potencijalno otežava praćenje toka podataka odnosno u daljem održavanju te komponente. Dobra praksa je, dakle, izbegavati takve promene osim ako su roditelj i dete čvrsto povezani dizajnom. U većini slučajeva, dete bi trebalo da emituje događaj koji će dozvoliti roditelju da izvrši mutaciju.

Pitanja i zadaci

1. Objasniti pojam direktiva.
2. Šta je to tekstualna interpolacija i kako se koristi?
3. Kada se koristi direktiva v-once?
4. Kada i kako se koristi direktiva v-html? Koja je potencijalna opasost u primeni ove direktive?
5. Za koje atribute kažemo da su dinamički? Koja direktiva se koristi u tom slučaju?
6. Navedite primer upotrebe dinamičkih atributa.
7. Kako delimo dinamičke atribute?
8. Napišite komponentu Student koja obezbeđuje prikaz: broja indeksa, imena i prezimena. Zatim, napisati komponentu ListaStudenata koja će, koristeći komponentu Student, prikazati sve studente u listi po izboru.

1.11 Računajuća svojstva i posmatrači

Vue komponente koriste šablonske izraze pri generisanju vrednosti za prikaz. Takvi izrazi mogu postati složeni i time otežati razvoj, odnosno onemogućiti laku upotrebu komponenata. Kako se u izrazima koriste vrednosti koje se izračunavaju u trenutku upotrebe, ceo sistem možemo optimizovati korišćenjem računajućih svojstava (eng. computed properties) ili posmatrača (eng. watch).

Računajuća svojstva i funkcija *computed*

Prikazaćemo tehniku primene računajućih svojstava u obe sintakse kada radimo sa SFC komponentama. Iako ćemo u nastavku koristiti Composition API, važno je sagledati razlike između dve sintakse radi lakšeg usvajanja novih koncepata.

- **Options API**: koristi se posebna sekcija komponente `computed`. U njoj se definišu svojstva koja se ponašaju kao promenljive, ali se vrednosti ažuriraju automatski kada se promene zavisnosti.

```
data() {
    return{
        brojac : 0,
        naziv1 : 'Test samoracu. svojstava',
}},
methods:{
    onClick:function(){
        this.brojac++;
    },
    naziv3:function(){
        return this.naziv1 + "; Br= " + this.brojac;
    },
    now:function(){
        return Date.now();
    }
```

```
  },
  computed:{
      naziv2:function(){
          return this.naziv1 + "; Brc= " + this.brojac;
      },
      now2:function(){
          return Date.now();
      }
  }
```

- **Composition API**: koristi se ugrađena funkcija `computed`, koja vraća reaktivnu vrednost zasnovanu na funkciji.

```
<script setup>
  import { computed, reactive } from 'vue';

  const state = reactive({
    brojac:0,
    naziv1 : 'Test samoračunajucih svojstava'
  });
  const naziv2 =
    computed(() => state.naziv1 + "; Brojač= " + state.brojac);
  const now2 =
    computed(() => Date.now());

  function onClick(){
    state.brojac++;
  }
  function naziv3(){
    return state.naziv1 + "; Brojač= " + state.brojac;
  }
  function now(){
    return Date.now();
  }
</script>

<template>
    {{state.naziv1}}<br>
    {{naziv2}}<br>
    {{naziv3()}}<br>
    {{now2}}<br>
    {{now()}}<br>
<button @click="onClick" >Promeni</button></template>
```

Važno. Može se zapaziti da je primena računajućih svojstava slična primeni metoda, odnosno da bi se isti rezultat dobio i ako bi imali identičnu metodu, međutim važna razlika postoji.

Računajuća svojstva se keširaju na osnovu reaktivne zavisnosti. Dakle, računajuća svojstva će biti ponovo računata samo ako je neka od reaktivnih zavisnosti promenjena. Ukoliko to nije slučaj, bez obzira na višestruki pristup računajućoj komponenti, ona se neće računati. Za razliku od računajuće komponente, svaki poziv funkcije znači i njeno izvršavanje.

Međutim, treba obratiti pažnju kada se koristi nereaktiva zavisnost, na primer:

```
now:function(){
    return Date.now();
}
```

U ovom slučaju računajuće svojstvo neće inicirati ponovni poziv funkcije i dobijanje novog vremena u odnosu na standardni poziv funkcije.

S druge strane, keširanje je izuzetno značajno u efikasnom radu komponenti. Ukoliko se neko računanje izvršava dugotrajno, svakako da je od ključnog značaja izbeći nepotrebno računanje ovakvih vrednosti. Bez keširanja, računanje bi bilo nepotrebno izvršavano i usporenje bi bilo neminovno.

Računajuća svojstva sa *getter* i *setter* funkcijama

Računajuća svojstva dobijaju vrednost na osnovu promene zavisnih podataka. Dakle, mogli bismo da kažemo da su pri podrazumevanoj primeni tipa ***getter*** svojstava. Osim takvog ponašanja, računajuća svojstva mogu imati i ***setter*** funkciju, koja se izvršava kada se svojstvu dodeli nova vrednost.

Pogledajmo primer koji pokazuje upotrebu oba tipa računajućih svojstava. Ako su svojstva Vue komponente: *ime* i *prezime*, a računajuće svojstvo *punoIme* se formira spajanjem imena i prezimena uz razmak, kod bi izgledao ovako:

```
<script setup>
  import { computed, ref } from 'vue';

  const data = ref({
    ime:'',
    prezime : ''
  });
  const punoIme = computed({
    get(){
```

```
      return data.value.ime + ' ' + data.value.prezime;
    },
    set(novoPunoIme){
      [data.value.ime, data.value.prezime] = novoPunoIme.split(' ');
    }
  });
</script>

<template>
  <div>
    <input v-model="data.ime"/>
    <input v-model="data.prezime"/>
    {{punoIme}}

    <br><br>
    <input v-model="punoIme"/><br>
    {{data?.ime}} {{data?.prezime}}
  </div>

</template>
<style scoped>
div {
  background-color: blue;
}
</style>
```

Dakle, metode jedne komponente treba koristiti kada se menjaju podaci komponente ili za druge akcije. Međutim, computed svojstva svakakko ne treba da se koriste za izmene podataka, već isključivo za formiranje željenih pogleda na podatke odnosno za različite vrste računa koji se zasnivaju na podacima komponente.

Napomena

- Metode komponente treba koristiti kada menjamo podatke ili pokrećemo akcije.
- Računajuća svojstva se koriste isključivo za formiranje izvedenih vrednosti na osnovu postojećih podataka.
- Dodavanje setter funkcije u računajuće svojstvo omogućava dvosmernu vezu, ali uz napomenu da to treba koristiti samo kada je zaista potrebno.

Posmatrači i funkcija *watch*

Vue nudi još jedan način za praćenje podataka, koji se izvodi primenom **posmatrača** tj. ugrađene **`watch`** funkcije. Iako je ovaj mehanizam sličan računajućim svojstvima,

princip posmatrača je zasnovan na monitoringu određenih podataka, tj. praćenju promene vrednosti zavisnih podataka.

Posmatrači se najčešće koriste zajedno sa asinhronim ili vremenski zahtevnim operacijama kada se očekuje da se sa odgovorom pokrene odgovarajuća logika. Dakle, ako imamo posmatrača tj. reaktivnu funkciju definisanu sa `watch`, tada svaka promena zavisnih podataka koji se nadgledaju, izaziva izvršavanje pridružene funkcije.

U sledećem primeru prate se promene dve promenljive (brojac1 i brojac2) i ažuriramo njihov zbir. Promene se pokreću sa dva dugmeta tj. klikom na svako od njih vrši se promena odgovarajuće promenljive u kodu.

Dodato je i treće dugme kojim se aktivira asinhrono učitavanje podataka sa API-ja, a dobijeni podaci se takođe prate koristeći `watch` mehanizam.

```
<script setup>
  import { watch, reactive } from 'vue';

  const state = reactive({
        brojac1 : 0,
        brojac2 : 0,
        brojac: 0,
        joke:"................."
  });
  const state2 = reactive({
        joke:"................."
  });

  const onClick1 = ()=>{
    state.brojac1++
  }
  const onClick2 = ()=>{
    state.brojac2++
  }

  watch(state, async(novoStanje, staroStanje)=>{
    state.brojac = state.brojac1 + state.brojac2
      try {
          const res =
          await fetch('https://api.chucknorris.io/jokes/random')
                 .then((response) => response.json());
          //state.joke =  res.value; //GRESKA! Petlja...
          state2.joke = res.value;
      } catch (error) {
```

```
            //state.joke = 'Error!' + error //GRESKA! Petlja...
            state2.joke = 'Error!' + error
        }
        console.log("promenjeno stanje")

    })

</script>

<template>
    <div>
      brojac: {{ state.brojac }} <br>
      brojacViceva: {{ state.brojacViceva }} <br>
      <button @click="onClick1">state.brojac1++</button> <br>
      <button @click="onClick2">state.brojac2++</button> <br>
      <!-- {{state.joke}} -->
       {{ state2.joke }}
       <br>

    </div>
</template>
<style scoped></style>
```

Primenom `watch` mehanizma postižemo da kod radi baš ono što je navedeno kao zahtev: na svaku promenu vrednosti `brojac1` ili `brojac2` vrši se ažuriranje ukupne vrednosti `brojac`.

Umesto više `watch` funkcija odnosno pojedinašnog praćenja više promenljivih, može se pratiti jedan objekat sa svojstvima koja se pojedinačno prate – rezultat je isti, a dobija se na preglednosti. U našem primeru pratimo stanje komponente, bez teksta (API vrednost) koji se preuzima sa url-a:

```
watch(state, async(novoStanje, staroStanje)=>{
```

Ukoliko bi dobijena API vrednost bila deo objekta praćenja, onda bi to izazvalo beskonačnu petlju tj. nakon svake nove vrednosti funkcija `watch` detektuje da se promena dogodila pa bi se to ponavljalo nadalje u nedogled. Ovaj problem možemo rešiti izmeštanjem svojstva `joke` iz stanja koje nadgledamo.

`watch` može nadgledati vrednosti koje su promenljive definisane kao: `ref`, `reactive`, `objekat`, `getter` ili niz. Na primer:

```
watch(()=>aaa.value+bbb.value, (sum)=>{
    console.log(`promenjena suma: ${sum}` );
```

```
})
watch(()=>state.brojac1, (brojac1)=>{
    console.log(`promenjen state.brojac1: ${brojac1}` );
})
```

Važno. Zapazite da se ne može posebno nadgledati neko svojstvo reaktivnog objekta. To možete uraditi preko getter funkcije.

Pitanja i zadaci

1. Šta su to računajuća svojstva?
2. Koja je razlika između računajućih svojstava i funkcija koje računaju iste vrednosti?
3. Napisati primer računajućeg svojstva *punoIme* koristeći reaktivne podatke jedne komponente: *ime* i *prezime*.
4. Napisati primer računajućeg svojstva *tekućeVreme* koje prikazuje sadašnje vreme.
5. Šta su *getteri* i *setteri* i kako se računajuća svojstva mogu iskoristiti za kreiranje ovih funkcija?
6. Objasni ulogu funkcije *watch*?
7. Kako se koristi ova funkcija?
8. Navedi primer primene *watch* funkcije.

1.12 Stilizacija

U prethodnim poglavljima pokazali smo kako Vue povezuje dinamičke podatke sa atributima HTML elemenata. Na sličan način moguće je upravljati atributima koji se

tiču prikaza: `class` odnosno `style`: **`v-bind:class`** odnosno **`v-bind:style`**. Ovim atributima se prosleđuju objekti zaduženi za klase i stilove.

Klase

Vue atribut v-bind:class, odnosno skraćeno :class, namenjen za dinamički rad sa CSS klasama. U opštem slučaju, ovaj atribut prihvata objekat koji odgovara CSS klasi, a svojstva tog objekta odgovaraju CSS svojstvima. Primena statičke CSS klase ostaje ista.

Uz to, Vue ubacuje opciju aktivacije klase. Da bi se jedna klasa uključila/isključila koristi se dinamička logička vrednost, kao u narednom primeru:

```
.myCls{
    ...
}
.clsTextColor{
    ...
}
----------
<div id="radni" :class="{myCls:aktivno}"
                class="clsTextColor">.....
</div>
```

U zavisnosti od vrednosti dinamičke promenljive `aktivno`, klasa `myCls` je aktivna ili ne. Obratite pažnju da se atributu `:class` prosleđuje objekat koji ima svojstvo istog naziva kao i klasa koja je definisana, a vrednost tog svojstva je logička vrednost.

Pogledajmo sledeći primer u kome je dodata logika kojom se naizmenično primenjuje CSS klasa, npr. klasa koja jednostavno menja pozadinsku boju. Reaktivna promenljiva `aktivno` se koristi zajedno sa klasom `clsZutaPozadina`. Inicijalno je vrednost te promenljive `false`, dok se svakim klikom na dugme vrši promena logičke vrednosti.

```
<template>
  <div id="radni" :class="{clsZutaPozadina:aktivno}"
                  class="clsPlaviText" >...
  </div>
  <button @click="onClick1">test1</button> <br>
</template>
<script setup>
 import {ref} from 'vue'
 const aktivno = ref(false);
```

```
 function onClick1(){ aktivno.value = !aktivno.value; }
</script>
<style scoped>
  .clsZutaPozadina { background-color: yellow; }
  .clsPlaviText{ color: blue; }
</style>
```

Napomena. Pri imenovanju klasa postoji pravilo koje se mora poštovati. Ukoliko naziv klase sadrži znak minus - , na primer:

```
.cls-CrveniText{ color: red; }
```

onda u kodu, pri primene takve klase, njen naziv mora stajati pod znacima navoda, tj:

```
<div id="radni" :class="{'cls-CrveniText':obojeno}">..</div>
```

Često je potrebno dinamički povezati više klasa tako da zajednički funkcionišu. Recimo da imamo dve klase koje bi trebalo da budu u istovremeno primenjene. Ceo objekat koji se pridružuje deklaraciji `:class` mora se oivičiti znacima navoda, dakle, koristi se jedan objekat sa više definisanih klasa:

```
<div id="radni" :class="{clsZutaPozadina:cls1, 'cls-
CrveniText':cls2}"> Tekst koji se prikazuje </div>
```

Slično, moguće je ceo objekat vezati za aktivnost više definisanih klasa preko jednog objekta postavljanjem stilskih klasa kao svojstava objekta, na primer:

```
    cls12: {
        clsZutaPozadina: true,
        'cls-CrveniText': false
    }
```

Svako svojstvo objekta `cls12` odgovara jednoj CSS klasi, a pripadajuća vrednost je logičkog tipa i ona određuje da li je klasa aktivna ili ne. Umesto konstantnih vrednosti `true/false` u primeru, može da stoji neka dinamička promenljiva. Tada je primena još jednostavnija i vrši se pridruživanjem samo ovog objekta:

```
<div id="radni" :class=cls12 >. . .</div>
```

a akcija promene stila bi izgledala kao akcija kojom se menjaju vrednosti svojstava objekta:

```
cls12.clsZutaPozadina = !cls12.clsZutaPozadina;
cls12['cls-CrveniText'] = !cls12['cls-CrveniText'];
```

Takođe, umesto da stoji uz svaku klasu `true`, više definisanih klasa se može napisati i kao niz, u slučaju da se više njih istovremeno primenjuje, na primer:

```
<div id="radni" :class="['clsZutaPozadina','cls-CrveniText']">. </div>
```

Ili kao niz svojstava kojima pripadaju klase:

```
<div id="radni" :class="[c.c1, c.c2]" >...</div>
```

Stilovi

Stilovi se slično primenjuju kao klase. Za definisanje dinamičkih stilova koristi se deklaracija `:style`, koja prihvata JavaScript objekat čija sintaksa odgovara CSS sintaksi stilova. To znači da se vrednost stila može formirati pomoću JS izraza i primenjujući reaktivna svojstva. Na primer:

```
<div id="div1"
     :style="{'background-color':boja,
              'font-size':velFontaUem +'em'}"
>
   Tekst koji se stilizuje
</div>
<div id="div2"
     :style="{backgroundColor:boja,
              fontSize:velFontaUem + 'em'}"
>
   Tekst koji se stilizuje
</div>

<script setup>
   const boja = ref('yellow');
   const velFontaUem = ref(2);
</script>
```

Treba zapaziti nekoliko važnih detalja. Prvo, u primeru je korišćen JS izraz pri formiranju vrednosti fonta u em. Drugo, svojstva koja se navode, iako se navode u okviru jednog stringa, moraju da poštuju dogovorenu sintaksu, a to znači da ukoliko u nazivu svojstva stoji znak minus (-), svojstvo se mora oivičiti dodatno znacima

navoda (eng. kebab-case), ili se koriste nazivi koji imaju prvo veliko slovo umesto znaka minus i prvog slova nove reči (eng. camelCase).

U slučaju da stilizaciju čuvamo u jednom objektu, onda bi to izgledalo ovako:

```
<div id="radni3" :style="stil" >Tekst koji se stilizuje</div>


<script setup>
        const stil=reactive({
            backgroundColor:'yellow',
            fontSize:'2em'
        });
```

Dakle, nije moguće u ovoj sekciji napraviti zavisne objekte od drugih. Ukoliko to želimo, onda bi bilo logično da ovaj objekat smestimo u računajuća svojstva, na sledeći način:

```
const velFontaUem = ref(2);
const stil1=reactive({
    backgroundColor:'yellow',
    fontSize:'2em'
});
const stil2=computed(()=>{
    return{
        backgroundColor:stil.backgroundColor,
        fontSize:2+velFontaUem.value+'em'
    }
});

<div :style="stil2">Tekst...</div>
```

U slučaju primene više stilova, odnosno više objekata koji su po svojoj prirodi stilovi, može se koristiti niz objekata pri definisanju. Na primer:

```
<div id="d5" :style="[stil1, stil2]">Tekst...</div>
```

Napomena. Primena stilova koji imaju vendor specifičnost za različite veb čitače, automatizovano se generišu. To je pod kontrolom Vue radnog okvira.

Pitanja i zadaci

1. Koje direktive koristi Vue za stilizaciju komponente?
2. Da li su *class* i *style* dinamičke direktive?
3. Kako se koristi postojeća klasa na statički a kako na dinamički način?
4. Ukoliko se koristi više dinamičkih klasa, kako se u tom slučaju vrši istovremena primena takvih klasa na jednom elementu?
5. Objasni sledeće kodove `:class="{cls1:c1, 'cls2':c2}"` `:class="['cls1','cls2']"`
6. Objasni sledeći kod: `<div id="radni" :class=cls12 >. . .</div>`

```
cls12: {
        clsZutaPozadina: true,
        'cls-CrveniText': false
    }
```

7. Kako se primenjuju stilovi u elementima jedne Vue komponenete? Uporediti sa primenom klasa.
8. Navedite primer upotrebe direktive :*style*.
9. Kako se izvodi dinamička promena veličine fonta preko dinamičkih stilova?
10. Da li se mogu koristiti računajuća svojstva pri formiranju dinamičkih stilova? Ako mogu, navedite primer.

1.13 Reaktivnost i renderovanje

U ovom poglavlju objasnićemo detaljnije koncept reaktivnosti i prikazati primenu dopunskog skupa ugrađenih funkcija koje omogućavaju dodatnu kontrolu nad njom. Takođe ćemo opisati kako se koristi uslovni prikaz (uslovno renderovanje), pomoću koga se postiže željena dinamika u prikazu.

Reaktivnost – detaljno

Do sada smo se susretali sa funkcijom `ref()`. Promenljive koje su deklarisane pomoću ove funkcije korišćene su u sekciji šablona, a njihova promena automatski je izazivala promene u prikazu.

Ovo je omogućeno pomoću sistema reaktivnosti zasnovanog na praćenju zavisnosti. Kada se komponenta renderuje po prvi put, Vue prati svaku referencu koja je korišćena tokom renderovanja. Kasnije, kada se neka od referenci promeni, pokreće se ponovno prikazivanje.

Promenljive koje se referencijaru sa `ref` mogu da sadrže bilo koji tip vrednosti – od prostih tipova, preko nizova, objekata pa sve do ugrađenih JavaScript struktura podataka, kao što je na primer `Map`. Funkcija `ref` će učiniti svoju vrednost **duboko reaktivnom**. To znači da možete očekivati da će promene biti otkrivene čak i kada se menjaju unutrašnji elementi objekata ili nizova. Pogledajmo sledeći primer:

```
<script setup>
import { ref, reactive } from 'vue'
const obj = ref({
  brojnoStanje: { brojac: 0 },
  studenti: ['1/22', '25/21', {ime:'jova', indeks:'88/22'}]
})
function Izmene() {
  obj.value.brojnoStanje.brojac++
  obj.value.studenti.push('11/19')
  obj.value.studenti[2].indeks = '99/22';
}
</script>
<template>
  <p>brojac: {{obj.brojnoStanje.brojac}}</p>
  <p>studenti: {{obj.studenti}}</p>
  <button @click="Izmene">Izmeni</button>
</template>
```

Promenljive neprimitivnog tipa (objekat `brojnoStanje` i niz `studenti`) koji se prosleđuju funkciji `ref` postaju reaktivni proksiji. To znači da Vue prati i unutrašnje izmene, pa se prikaz automatski osvežava kada se menja objekat `brojnoStanje` brojača ili sadržaj niza `studenti`.

shallowRef

Ponekad nije potrebno da Vue prati svaku unutrašnju promenu objekta. U takvim slučajevima možemo koristiti `shallowRef()`, koji **isključuje duboku reaktivnost**. Kod ovih referenci Vue prati samo promene na nivou `.value`, dok se unutrašnje izmene ignorišu. Na primer:

```
const state = shallowRef({ brojac: 1 })
// ne detektuje se promena stanja
state.value.brojac = 2
// detektuje se promena stanja
state.value = { brojac: 2 }
```

Vue nudi opciju forsiranog trigerovanja u slučaju primene `shallowRef` reference. To se postiže funkcijom `triggerRef` koja izaziva reakciju tj. trigerovanje. Na primer:

```
const state = shallowRef({
  brojac: 2
})
watchEffect(() => {
  console.log(state.value.brojac)
})
state.value.brojac = 33
triggerRef(state)
```

nextTick

Kada se promeni reaktivno stanje, rezultujuća ažuriranja DOM-a se ne primenjuju istovremeno. Vue ih baferuje do „sledećeg tika“ kako bi se osiguralo da se svaka komponenta ažurira samo jednom bez obzira na to koliko promena stanja se dogodilo u okviru jednog ciklusa.

Ako želimo da sačekamo da se DOM ažurira pre nego što izvršimo neku operaciju, koristimo funkciju `nextTick()`. Ona omogućava sinhronizaciju između promene stanja i ažuriranja prikaza.

```
<script setup>
import { ref, nextTick } from 'vue'
const brojac = ref(0)
async function uvecaj() {
  brojac.value++
  console.log(document.getElementById('brojac').textContent)//0

  await nextTick()
```

```
  console.log(document.getElementById('brojac').textContent)//1
}
</script>

<template>
  <button id="brojac" @click="uvecaj">{{ brojac }}</button>
</template>
```

reactive

Drugi način da se deklariše reaktivno stanje je primenom funkcije **reactive()**. Za razliku od funkcije **ref** koja pravi omotač za unutrašnju vrednost specifičnog objekta, funkcija **reactive()** čini ceo objekat reaktivnim.

Reaktivni objekti se ponašaju kao obični JavaScript objekti. Razlika je u tome što Vue može da presretne pristup i promenu svih svojstava reaktivnog objekta radi praćenja i pokretanja reaktivnosti. Na primer:

```
const x = {}
const proxy = reactive(x)
console.log(proxy === x) // false

console.log(reactive(x) === proxy) // true
// reactive(proxy) vraća sam proxy
console.log(reactive(proxy) === proxy) // true
```

Dakle:

- **reactive()** vraća novi proksi objekat, različit od originalnog.
- Ako se **reactive()** pozove nad već postojećim proksijem, vraća se isti objekat (ne pravi se novi).

Takođe, važi:

```
const proxy = reactive({})
const x = {}
proxy.x = x
console.log(proxy.x === x) // false
```

Ugnežđeni objekti su takođe reaktivni kada im se pristupi, bez obzira koliko nivoa imaju. Zbog toga tj. povećanja efikasnosti, slično **shallowRef** referencama, postoji i **shallowReactive** API za isključivanje duboke reaktivnosti.

Nedostatak

Može se primeniti samo na objektne tipove (`object`, `arrays`, `Map`, `Set`…). Dakle, ne mogu čuvati promenljive primitivnih tipova (`string`, `number`, `boolean`,…).

Referenca na reaktivni objekat se ne može izmeniti, može se kreirati nova. Na primer:

```
let state = reactive({ count: 0 })
// gornja referenca ({ count: 0 })  neće biti u nastavku praćena tj. reaktivnost je izgubljena
state = reactive({ count: 1 })
```

Prilikom razmotavanja (destrukcije) reactive objekata dolazi do gubitaka osobine praćenja reaktivnosti.

Uslovno renderovanje

Direktiva **`v-if`** se koristi za uslovno renderovanje. Dakle, ako se koristi ova direktiva u nekom elementu, onda će se taj element renderovati (tj. obrađivati njegov prikaz), samo ako je vrednost ovog izraza **`true`**.

Uz **`v-if`** može se koristiti i direktiva **`v-else`**, koja se izvršava kada uslov nije ispunjen. Primer:

```
<template>
    <div id="div1" v-if=prikaz >prikaz == true</div>
    <div id="div2" v-else >prikaz == false</div>
    <button @click="onClick">toggle</button>
</template>

<script setup>
 import {ref} from 'vue'
 const prikaz = ref(false);
 function onClick(){ prikaz.value = !prikaz.value; }
```

Osim za jedan element, uslovno renderovanje se može primeniti za grupu elemenata. U tom slučaju elementi se mogu grupisati u jedan element naziva **`template`**. Tada se uslovno renderovanje primenjuje na elementu template, odnosno na sve elemente u tom elementu. Pogledajmo primer koji ilustruje ovo kao i primenu direktive **`v-else-if`**:

```
<template>
   <button @click="onClick">random - {{ vrednost }}</button>
   <template v-if='vrednost>0.6'>
       <h2>h2</h2>
   </template>
   <template  v-else-if='vrednost>0.3'>
       <h5>h6</h5>
   </template>
   <template  v-else>
       <h6>h6</h6>
   </template>
</template>

<script setup>
  import {ref} from 'vue'
  const vrednost = ref(0);
  function onClick(){
    vrednost.value = Math.random();
  }
</script>
```

Osim direktive `v-if`, za uslovno renderovanje koristi se i direktiva `v-show`. Za razliku od direktive `v-if`, u slučaju primene direktive `v-show`, vrši se izračunavanje prikaza tj. renderovanje. Dakle, nakon renderovanja, direktivom se vrši „toglovanje“ CSS svojstva `display`, na taj način se kontroliše da li se element prikazuje ili ne.

Postoje još neke bitne razlike. Prvo, `v-show` ne podržava rad sa grupisanjem tj. ne može se primeniti na element `template`. Takođe, `v-show` ne podržava `v-else` direktivu.

Dakle, `v-show` definiše samo da li se element vidi ili ne, ali se proces renderovanja uvek obavlja. Za razliku od njega, `v-if` vrši renderovanje samo ako je uslov `true`, inače se ne vrši nikakav proračun prikaza. Možemo reći da je `v-show` inicijalno „skupa“ operacija, dok je `v-if` „skupa“ pri svakoj promeni prikaza. Iz ovoga sledi da

bi trebalo da `v-show` koristimo za učestale promene, a da `v-if` koristimo za povremene promene.

Renderovanje liste

Za renderovanje liste stavki u sekciji `template` Vue koristi direktivu `v-for`. Ova direktiva zahteva primenu odgovarajuće `in` petlje, na primer:

`v-for="item in items"`.

Odmah da naglasimo da upotreba direktive `v-if` u petlji `v-for`, nije preporučljiva. Razlog je činjenica da `v-if` ima veći prioritet od `v-for`. Zbog toga, `v-if` uslov neće imati pristup promenljivama iz oblasti važenja `v-for`. Dakle, sledeći primer nije dobro napisan kod:

```
<!-- nije dobro -->
<ul>
 <li v-for="item in items" v-if="item.isVisible" :key="item.id">
  {{ item.part1 }}
 </li>
</ul>
```

Kako ovo rešiti?

Prvo rešenje je da se umesto uslova u petlji unapred odredi, na primer primenom računajucih svojstava, lista stavki za prikaz, a onda se ta lista koristiti u petlji, dakle:

```
<!-- dobro -->
<ul>
    <li v-for="item in visibleItems" :key="item.id" >
    {{ item.part1 }}
    </li>
</ul>
```

Slično, ako bi uslov za prikaz bio zajednički za sve stavke, tj. ako `v-if` ide zajedno sa `v-for` u istom elementu, to ne bi bilo dobro:

```
<!-- nije dobro -->
<ul>
  <li v-for="item in items" v-if="isItemsVisible" :key="item.id">
     {{ item.part1 }}
    </li>
</ul>
```

Drugo rešenje je da se uslov za zajednički prikaz, ako je to moguće, izmestiti u roditeljsku kontrolu, ili dete kontrolu na primer:

```
<!--  dobro -->
<ul v-if="isItemsVisible">
    <li v-for="item in items" :key="item.id" >
        {{ item.part1 }}
    </li>
</ul>
```

ili

```
<ul>
    <template v-for="item in items" :key="item.id">
        <li v-if="item.isVisible">
            {{ item.part1 }}
        </li>
    </template>
</ul>
```

Napomena. JavaScript nudi dve slične petlje: `in` odnosno `of` petlju. Dok prva petlja formira iteracije preko ključeva kolekcije, druga koristi vrednosti kolekcije. S druge strane, Vue je radni okvir koji ima svoja pravila i sintaksu.

Ukoliko nam je u petlji potreban podatak o pojedinim svojstvima objekta, moguće je koristiti proširenu `in` petlju sa dodatnim objektima za iteraciju, na primer:

```
<div v-for="(value, name, index) in object">
    {{ index }}. {{ name }}: {{ value }}
</div>
```

Kada se ažurira lista elemenata prikazanih pomoću `v-for`, podrazumevano se koristi zamene sadržaja na definisanom mestu. Dakle, ako se redosled stavki promeni, umesto premeštanja DOM elemenata, vrši se zamena svakog elementa na osnovu određenog indeksa.

Ovo je efikasno, ali kada se ispis vaše liste ne oslanja na stanje podređene komponente ili privremeno stanje DOM. Da bi se mogao da prati identitet svakog čvora potrebno je da za svaku stavku postoji jedinstveni **ključni** atribut:

```
<div v-for="item in items" v-bind:key="item.id">
    <!-- sadržaj -->
</div>
```

Preporučuje se upotreba ključnog atributa sa `v-for` kad god je to moguće, osim ako je ponovljeni DOM sadržaj jednostavan. Pošto je Vue generički mehanizam za identifikaciju čvorova, ključ ima i druge namene koje nisu posebno vezane za `v-for`, kao što ćemo videti kasnije.

Važno. Ne koristite ne-primitivne vrednosti kao što su objekti i nizovi za `v-for` ključeve. Umesto toga koristite string ili numeričke vrednosti.

Primer

Prethodno objašnjeno renderovanje liste objekata sada ćemo iskoristiti za primer prikaza objekata koji predstavljaju dane u nedelji. Naredni kod sadrži veći broj kontrola `button` kojima su pridružene tipične funkcije za rad sa nizovima.

Primer:

```
<template>
     <ul>
        <li v-for="dan in dani" :key="dan.id">
            {{ dan.id }}. {{ dan.naziv }}
        </li>
    </ul>

    <button @click="onClickPush">Push</button>
    <button @click="onClickPop">Pop</button>
    <button @click="onClickShift">Shift</button>
    <button @click="onClickUnshift">Unshift</button>
    <button @click="onClickSplice">Splice</button>
    <button @click="onClickSort">Sort</button>
    <button @click="onClickReverse">Reverse</button>
</template>

<script setup>
import { reactive } from 'vue'
const
dani=reactive([{id:1,naziv:'pon'},{id:2,naziv:'ut'},{id:3,naziv:'sre
'},{id:4,naziv:'čet'},{id:5,naziv:'pet'},{id:6,naziv:'sub'},{id:7,na
ziv:'ned'}]);

function onClickPush(){
    dani.push({id:8,naziv:'abc'});
}
function onClickPop(){
```

```
    var dan = dani.pop();
    console.log(dan);
}
function onClickShift(){
    var dan = dani.shift();
    console.log(dan);
}
function onClickUnshift(){
    var dan = dani.unshift({id:8,naziv:'aaa'},{id:9,naziv:'bbb'});
    console.log(dan);
}
function onClickSplice(){
    var dan = dani.splice(1,0,{id:44,naziv:'aaa'});//splice(start,
DeleteCount, item1)
    console.log(dan);
}
function onClickSort(){
    dani.sort(function (a, b) {
        if(a.naziv > b.naziv) return 1;
        else if(a.naziv < b.naziv) return -1;
        else return 0;
    });
}
function onClickReverse(){
    dani.reverse();
}
</script>
```

Klikom na neko dugme, izvršava se jedna od funkcija nad nizovima. Izvršavanjem funkcije menja se reaktivni objekat i istovremeno se dešava odgovarajuća promenu u prikazu.

Dodatni zadatak. Dodati dugme kojim se filtriraju svi dani čiji naziv počinje sa „p“.

Povezivanje sa kontrolama formi: *v-model*

Vue poseduje specijalnu direktivu **v-model**, koja omogućava **bidirekciono povezivanje** (eng. two-way binding) između podataka i kontrola forme. Ova direktiva može se koristiti sa elementima:

- **input**,
- **textarea** ili
- **select**.

Bidirekciono povezivanje obezbeđuje:

- Sve promene na modelu, koje se događaju u pozadini, automatski se prikazuju u interfejsu.
- Sve pomene koje korisnik napravi u formi automatski se preslikavaju nazad na model.

Zbog ovakve prirode povezivanja sa podacima, **v-model ignoriše inicijalna podešavanja** postavljena preko atributa: **value**, **checked** i **selected**. Umesto toga treba koristiti JavaScript odnosno tehnike inicijalizacije ili data sekciju.

Pri povezivanju podataka **v-model** direktiva se oslanja na određena svojstva i događaje u zavisnosti od tipa input elemenata. Na primer, kontrole tipa:

- **text** odnosno **textarea** elementi koriste svojstvo **value** odnosno događaj **input**,
- **checkbox** odnosno **radiobutton** koriste svojstvo **checked** odnosno događaj **change**,
- **select** koriste svojstvo **value** odnosno događaj **change**.

U nastavku je prikazan primer forme koja koristi više tipova input elemenata. Svaka kontrola je povezana sa jednim podatkom u modelu, a rezultat povezivanja se odmah vidi u prikazu.

```
<script setup>
import {ref, reactive, computed} from 'vue'
const data = reactive({
    poruka:"početno stanje", //poruka:undefined,
    odabranaMuzika:'zabavna',
    odabraneBoje:[],
    language:"Ruski",

    odabranJezikVrednost: 1,
    jezici: [
        { naziv: 'Srpski', vrednost: 0 },
        { naziv: 'Engleski', vrednost: 1 },
        { naziv: 'Ruski', vrednost: 2 }
    ]
})
</script>

<template>
 <form>
```

```
    <input v-model="data.poruka" placeholder="unesi poruku" />
    <p>Poruka: {{ data.poruka }}</p> <hr>

    <textarea v-model="data.poruka"
              placeholder="unesi tekst sa vise linija">
    </textarea>
    <p style="white-space: pre-line;">{{data.poruka}}</p> <hr>

    <input type="radio" id="rbNarodna" value="narodna"
           v-model="data.odabranaMuzika" />
    <label for="rbNarodna">narodna</label>

    <input type="radio" id="rbZabavna" value="zabavna"
           v-model="data.odabranaMuzika" />
    <label for="rbZabavna">zabavna</label>
    <p>Muzika:{{ data.odabranaMuzika }}</p> <hr>

    <input type="checkbox" id="chkCrveno" value="crveno"
           v-model.lazy="data.odabraneBoje" />
    <label for="chkCrveno">crveno</label>

    <input type="checkbox" id="chkZeleno" value="zeleno"
           v-model="data.odabraneBoje" />
    <label for="chkZeleno">zeleno</label>

    <input type="checkbox" id="chkPlavo" value="plavo"
           v-model="data.odabraneBoje" />
    <label for="chkPlavo">plavo</label>
    <p>Boja:{{ data.odabraneBoje }}</p> <hr>

    <select v-model=" data.language">
      <option disabled value="">Odaberi jedan</option>
      <option>Srpski</option>
      <option>Engleski</option>
      <option>Ruski</option>
    </select>
    <p>Jezik: {{ data.language }}</p> <hr>

    <select v-model="data.odabranJezikVrednost" multiple>
      <option v-for="jezik in data.jezici" :value="jezik.vrednost">
          {{jezik.naziv}}
      </option>
    </select>
    <p>Odabran jezik: {{data.odabranJezikVrednost}}</p> <hr>

  </form>
</template>

<style scoped>
</style>
```

Ako se koristi `select` polje za unos, a vrednost koja je povezana preko v-model ne odgovara ni jednoj od ponuđenih opcija, tada neće biti selektovano ni jedno polje, tj. prikaz će ostati prazan dok korisnik ne izabere neku vrednost. Ovo je očekivano ponašanje, jer Vue ne može da pronađe odgovarajuću stavku u listi.

HTML kontrole: `radio`, `select` (sa elementima `option`) odnosno `checkbox`, koriste direktivu `v-model` za povezivanje vrednosti koje su tipa `string`, odnosno za `checkbox` tipa `boolean`. Ukoliko je potrebno izvesti dinamičko povezivanje drugih svojstava, koristi se direktiva `v-bind`, i u tom slučaju se za povezivanje mogu koristiti nestringovske vrednosti.

Kada je reč o prikazu liste stavki, element `select` sadrži opcije koje se navode kao podelementi. Ukoliko su opcije dinamičke, onda se i u prikazu mora omogućiti dinamički prikaz istih. Na primer, ako je definisana lista opcija `jezici` sa ključevima i vrednostima, onda se u prikazu `select` kontrola povezuje za jednu vrednost, na primer za `odabranJezikVrednost`:

```
odabranJezikVrednost: 1,
jezici: [
    { naziv: 'Srpski', vrednost: 0 },
    { naziv: 'Engleski', vrednost: 1 },
    { naziv: 'Ruski', vrednost: 2 }
]
```

Kao što se vidi, lista jezika je niz objekata sa atributima `naziv` i `vrednost`, dok se odabrani jezik opisuje samo kao vrednost, koja inače ne mora biti broj.

Dakle, u prikazu se povezivanje vrši na sledeći način:

```
<select v-model="odabranJezikVrednost">
   <option v-for="jezik in jezici" :value="jezik.vrednost">
      {{jezik.naziv}}
   </option>
</select>
<p>Vrednost za odabran jezik: {{ odabranJezikVrednost }}</p>
```

Ukoliko je potrebno omogućiti **višestruko selektovanje**, dovoljno je dodati atribut `multiple` u element `select`, automatski sve će funkcionisati, s tim da `odabranJezikVrednost` postaje niz.

Modifikatori

Modifikatori se dodaju na kraj direktive. U slučaju `v-model` direktive sledeći modifikatori su od značaja:

`v-model.number`

Kada se povezana vrednost automatizovano prebacuje u brojčanu vrednost:

```
<input v-model.number="kolicina" type="number"/>
```

Primenom modifikatora dobijena vrednost se automatski konvertuje u broj. Ako se koristi `type="number"` modifikator se primenjuje automatizovano. Samo u slučaju da parseFloat() ne uspe da izdvoji broj, vraća se originalna vrednost.

`v-model.trim`

Modifikator kojim se izbacuju nepotrebne beline.

```
<input v-model.trim="beline"/> <br>
```

`v-model.lazy`

Ovim modifikatorom Vue obezbeđuje sinhronizaciju podataka sa modelom tek pošto bude završen unos podataka. Dakle, dok je fokus na istoj kontroli i sadržaj se menja, ne vrši se sinhronizacija i promena podataka u modelu. Tek sa okončanjem unosa tj. promenom fokusa podaci postaju sinhronizovani.

Evo primera koji objedinjava prethodne slučajeve:

```
<div id="app">
  <form>
    v-model <input v-model="kolicina" type="number"/>
    <br>
    v-model.number
    <input v-model.number ="kolicina" type="number"/>
    <p>{{typeof(kolicina)}} {{kolicina}}</p>
    <hr>
    v-model.trim <input v-model.trim="beline"/> <br>
    v-model.lazy <input v-model.lazy="beline"/> <br>
    <p>{{beline}}</p>
    <hr>
  </form>
</div>
```

```
<script>
  var app = new Vue({
    el: '#app',
    data: {
        kolicina:10,
        beline:"proba  proba"
    },
  })
</script>
```

Pitanja i zadaci

1. Šta znači pojam reaktivnosti?
2. Kojim funkcijama se ostvaruje reaktivnost?
3. Kada se koristi funkcija shallowRef?
4. Objasni funkciju nextTick()?
5. Uporedi funkcije reactive i ref.
6. Napiši primer uslovnog renderovanja teksta u prikazu, u zavisnosti od vrednosti brojača. Brojač se menja sa klikom na dugme.
7. Direktiva v-for se koristi zajedno sa v-if? Kako se realizuje uslovno renderovanju pomoću direktivi v-for?
8. Objasni direktivu za bidirekciono povezivanje uz sopstveni primer.
9. Čemu služe modifikatori: .number i .trim?

1.14 Pristup podacima

Pristup do podataka iz Vue aplikacije može biti realizovan na više načina. Ovde će biti opisan način koji je u praksi načešći, a to je da se podacima pristupa posredstvom nekog API-ja. U jednom od ranijih primera već smo koristili metodu `fetch` koja je ugrađena u JavaScript jezgro. Ovu metodu smo koristili na sledeći način:

```
fetch('http://****.json')
```

```
  .then((response) => response.json())
  .then((data) => console.log(data));
```

Dakle, na ovaj način smo pristupali JSON fajlu kroz mrežu, a dobijeni podaci su prikazani u konzoli. Ovo je najjednostavnija primena kada se kroz jedan argument navodi samo putanja do resursa koji se preuzima. Naravno, resurs se ne dobija direktno već se vraća *obećanje* (eng. Promise - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise) koje se razrešava preko `response` objekta. Objekt `response`, ne sadrži samo JSON odgovor, već predstavlja ceo HTTP odgovor. Dakle, da bismo izdvojili sadržaj tela JSON podatka iz objekta `response`, koristimo metod `json()` koji vraća drugo obećanje koje se rešava kao JSON podatak `data`, koji se zatim koristi za prikaz.

Metoda `fetch` može da prosledi, kao drugi argument, objekat za podešavanja. To bi izgledalo ovako:

```
async function postData(url = '', data = {}) {
  // Default options: *
  const response = await fetch(url, {
    method: 'POST', // *GET, POST, PUT, DELETE, etc.
    mode: 'cors', // no-cors, *cors, same-origin
    cache: 'no-cache',
    // *default, no-cache, reload,
    // force-cache, only-if-cached
    credentials: 'same-origin',
    // include, *same-origin, omit

    headers: {
      'Content-Type': 'application/json'
      // 'application/x-www-form-urlencoded',
    },
    redirect: 'follow', // manual, *follow, error
    referrerPolicy: 'no-referrer',
    // no-referrer, *no-referrer-when-downgrade,
    // origin, origin-when-cross-origin, same-origin,
    // strict-origin, strict-origin-when-cross-origin, unsafe-url

    body: JSON.stringify(data)
    // body data mora da odgovara "Content-Type" header
  });
  return response.json(); // parses JSON response into native
JavaScript objects
}
```

```
postData('https://example.com/answer', { answer: 42 })
  .then((data) => {
    console.log(data); // JSON data parsed by `data.json()`
});
```

Fetch API predstavlja moćan mehanizam za asinhrono preuzimanje podataka. Detaljnije informacije su dostupne na adresi: https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch.

Pored Fetch API-ja, moguće je koristiti dodatni Node modul axios, koji takođe obezbeđuje HTTP klijenta zasnovanog na obećanjima. Prednost axios-a je što radi identično u Node okruženju i u veb čitačima, pa se može koristiti izomorfno.

U ovoj poglavlju koristićemo jedan javni otvoreni API za praćenje cene kriptovaluta u realnom vremenu pošto se vrednosti ažuriraju svakog minuta, kao i jedan postojeći API za rad sa Northwind bazom.

Da bi koristili axios biblioteku možemo instalirati odgovarajući modul koristeći npm/yarn alata.

```
npm install axios /  yarn add axios
```

Ili, u slučaju da koristimo CDN-a za povezivanje:

```
<script
src="https://cdn.jsdelivr.net/npm/axios/dist/axios.min.js"></script>
```

ili

```
<script src="https://unpkg.com/axios/dist/axios.min.js"></script>
```

Važno je na početku odrediti kako izgleda forma svih podataka. Na osnovu toga definišemo šta treba da koristimo za prikaz. Da bi se to uradilo, treba uputiti poziv tj. zahtev krajnjoj tački API-ja i izvesti to na način da se podaci prikažu. U dokumentaciji CoinGecko API-ja može se videti da će ovaj poziv biti upućen na https://api.coindesk.com/v1/bpi/currentprice.json.

```
<script setup>
import {onMounted} from 'vue'
import axios from 'axios';
onMounted(()=>{
```

```
  axios
  .get('https://api.coingecko.com/api/v3/simple/price?ids=bitcoin&vs
_currencies=usd,eur')
  .then(response=>{
    console.log(response.data.bitcoin);
  })
})
</script>
<template> </template>
<style scoped> </style>
```

Dobija se:

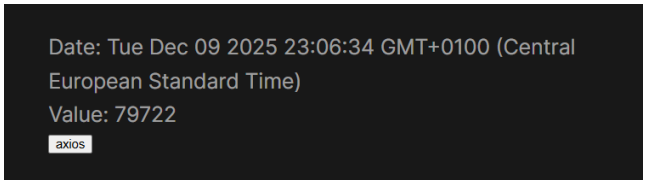
Slika 1.12. Pogled na prozor Console nakon izvršavanja

Pogledajmo sada jedan primer kako možemo da oblikujemo dobijene podatke na željeni način. Ovo radimo iz dva dela. Najpre, preuzimamo dobijene podatke, na primer `response.data` odnosno `response.data.bitcoin.eur`, a zatim ove podatke prikazujemo na željeni način:

```
<script setup>
  import {ref, reactive, onMounted} from 'vue'
  import axios from 'axios'

  const valuta = ref('eur');
  const bitcoinValue = ref(undefined);
  const bitcoinDate = ref(undefined);
  const greska = ref(undefined);
  const preuzimanjePodataka = ref(false)

  const testCurrentPrice = ()=>{
    fetch('https://api.coingecko.com/api/v3/simple/price?ids=bitcoin
&vs_currencies=eur')
```

```
      .then((response)=>response.json())
      .then(
        (data)=>{
          console.log(data)
          console.log(data.bitcoin.eur);
        }
      )
  }
  const CurrentPrice = () => {
    preuzimanjePodataka.value = true;
    axios
      .get('https://api.coingecko.com/api/v3/simple/price?ids=bitcoi
n&vs_currencies=usd,eur')
      .then(response=>{
        console.log(response.data.bitcoin);
        bitcoinValue.value = response.data.bitcoin.eur;
      })
      .catch((error)=>{
        greska.value = error;
        console.error('Error:' + error);
      })
      .finally(()=>preuzimanjePodataka.value = false)

      console.log(bitcoinValue.value);
  }

  onMounted(()=>{
    testCurrentPrice();
  })
  const onClick = () => {
    CurrentPrice();
  }
</script>

<template>
  <div v-if="greska">{{ greska }}</div>
  <div v-if="preuzimanjePodataka">Loading....</div>

  <h2>Date: {{ new Date() }}</h2>
  <h2 >Value: {{ bitcoinValue }} </h2>
  <button @click="onClick()">axios</button>
</template>

<style scoped> </style>
```

Dobija se:



Slika 1.13. Pogled za ispravno dobavljanje vrednosti kriptovalute

Obrada grešaka

Obrada grešaka je važan deo u okviru rada sa podacima, jer omogućava da se pravilno reaguje na neočekivane situacije tokom izvršavanja koda. Na primer, postoje slučajevi kada nećemo dobiti podatke koji su očekivani. Nekoliko razloga postoji zbog kojih bi axios poziv mogao da ne uspe u dobavljanju podataka su:

- API nije dostupan ili trenutno nije aktivan;
- Zahtev je pogrešno formiran;
- API nam ne vraća informacije u formatu koji je očekivan.

Kada se uputi zahtev, treba proveriti da li su se dogodile neke okolnosti zbog kojih nastaje greška, odnosno trebalo bi imati uvid u informacije o grešci kako bismo znali kako da rešimo problem. Ovo se rešava dodavanjem funkcije koja obrađuje odbijanje u slučaju *obećanja*, a to je `catch` funkcija.

```
fetch('http://****')
    .then((response) => response.json())
    .then((data) => {
      console.log('Success:', data);
    })
    .catch((error) => {
      console.error('Error:', error);
    });
```

Slično se postupa i u slučaju korišćenja axios-a, dakle korišćenjem `catch` metode. Obećanja poseduju i treću metodu `finally`. Ova metoda se poziva pošto obećanje bude razrešeno, bilo kao odobreno ili odbijeno. Dakle, pomoću ove metode se

obavljaju operacije koje se mogu označiti kao zajedničke i kada zahtev uspe i kada ne uspe.

U našem primeru kod se može proširiti sa poljima koja označavaju grešku odnsno fazu čitanja podataka:

```
<script setup>
  const greska = ref(undefined);
  const preuzimanjePodataka = ref(false);
  function getData() {
    preuzimanjePodataka.value = true;
    axios
    .get . . .
    .catch(error=>{
        greska.value = error;
        console.log(greska.value);
    })
    .finally(()=>preuzimanjePodataka.value=false)
  }
  onMounted(() => {
    getData();
  })
</script>
<template>
  <div v-if="greska">{{greska}}</div>
  <div v-if="preuzimanjePodataka">Loading...</div>
 . . .
</template>

<style scoped>
  .greska{
    color: red;
  }
  .loading{
    color: green;
  }
</style>
```

Način provere: Možete osvežiti stranicu kako bi ponovili čitanje podataka nakratko i eventualno videli status učitavanja dok se obavlja prikupljanje podataka preko API-ja. Ukoliko je učitavanje brzo, možete ga simulirati primenom nekog tajmera. Što se tiče greške, može se preko namenskog koda izazvati izuzetak za više slučajeva, recimo ubacivanjem greške u url adresu.

Primer: rad sa bazom

Pogledajmo sada pristup do podataka baze **Northwind** koristeći postojeće API funkcije. Ukoliko posedujete mogućnost da pokrenete sopstvene servise ili da koristite one koje su postavljeni za sam kurs, možete koristiti slične sa adrese: http://northwind.now.sh/api.

Prvi deo ovog primera biće posvećen dobijanju jedne, odnosno više kategorija.

Dakle, najpre ćemo dodati odgovarajuće podatke našoj komponenti za rad sa jednom kategorijom, odnosno za niz od više kategorija.

```
const data = reactive({
    kategorija: {
    description:"...",
    categoryName:"...",
    categoryId:0
      },

    kategorije:[],
})
```

Preko ovih podataka ostvarićemo slanje željenih zahteva kao i skladištenje onih koje se prihvataju. Primeri koji slede sadrže i delove ispisa u konzolu koji se koriste radi testiranja i detaljnije provere podataka u toku rada.

Pođimo redom od funkcije kojom ćemo ostvariti dohvatanje podataka za jednu kategoriju. Definisanje željene kategorije vrši se pomoću podatka `categoryId`. Koristićemo metodu tipa GET. Kod u template sekciji piše se tako da se zadaje željena kategorija po vrednosti id polja, a ono što se pročita ispisujemo nakon prijema podataka na dnu ove sekcije:

```
<section>
    <h2>GET: Dobijanje jedne kategorije</h2>
    <input v-model="data.kategorija.categoryId" /><br>
    <button @click="Get_Citanje1Kategorije()">
       Čitanje jedne kategorije
    </button> <br>
    <p>Procitano:</p>
    {{data.kategorija.categoryId}}
```

```
    {{data.kategorija.categoryName}}
    {{data.kategorija.description}}
</section>
```

Slika 1.14. Izgled za deo dobijanja jedne kategorije

Dok se kod za slanje i obradu primljenog podatka piše u sekciji methods:

```
function Get_Citanje1Kategorije(){
 axios
  .get('http://localhost:51047/api/categories/'
       + data.kategorija.categoryId)
  .then((response)=>{
    console.log('response.data:'+JSON.stringify(response.data));
    data.kategorija = response.data;
  })
  .catch(function (greska) {
    console.log(greska);
  })
}
```

Dobijeni podaci se prihvataju preko objekta `response.data` i mogu se, ako polja odgovaraju, postaviti u odgovarajući objekat komponente, kao što je to slučaj kod nas. Međutim, moguće je primeniti i različite programerske tehnike za to. Na primer:

- `Object.assign(kategorija.id, response.data)`
- `kategorija = {...kategorija, ...response.data}`

i slične, u zavisnosti od konkretnog slučaja.

Primljeni podaci se prikazuju u kontrolama za koje su povezani. Slično je i kada se radi sa skupom kategorija. Dakle, lista dobijenih podataka smešta se u dogovarajući niz u komponenti, a zatim se vrši prikaz koristeći v-for direktivu.

```
<section>
    <h2>GET: Dobijanje liste kategorija</h2>
    <button @click="Get_CitanjeKategorija()">
        Čitanje kategorija
    </button>
    <table>
    <thead>
```

```
    <tr>
        <th>categoryId</th>
        <th>categoryName</th>
        <th>description</th>
    </tr>
    </thead>
    <tbody>
    <tr v-for="k in data.kategorije" :key="k.categoryId">
        <td>{{ k.categoryId }}</td>
        <td>{{ k.categoryName }}</td>
        <td>{{ k.description }}</td>
    </tr>
    </tbody>
    </table>
</section>
```

Dok se čitanje liste podataka vrši u funkciji:

```
function Get_CitanjeKategorija(){
  axios.get('http://localhost:51047/api/categories')
  .then((response)=>{
      data.kategorije = response.data;
      console.log('kategorije:'+JSON.stringify(data.kategorije));
  })
  .catch(function (params) {
    debugger;
    console.log(params);
  })
}
```

Prilikom dodavanja nove kategorije treba voditi računa o zahtevima servera odnosno api funkcije koja to obavlja. Najčešće se dodavanje vrši POST metodom. U ovom konkretnom slučaju, id vrednost nove kategorije formira se automatski i ne zadaje. U primeru koji sledi dajemo primer slanja post zahteva drugi preko axios objekta. Evo odgovarajućih kodova.

```
<h2>POST: Nova kategorija - fja post</h2>
<div>
    <form>
    <label for="CategoryName">CategoryName</label>
    <input
        id="CategoryName"
        type="text"
        v-model='data.kategorija.categoryName' > <br>
    <label for="description">description</label>
    <input id="description" type="text"  v-
model='data.kategorija.description'> <br>
```

```
    <button @click="Post_NovaKategorija()">nova</button>
    </form>
</div>
</section>
<section>
<h2>PUT: Promena odredjene kategorije</h2>
  <div>
    <form>
      <label for="categoryId">id</label>
      <input
       id="categoryId"
       type="text"
       v-model = 'data.kategorija.categoryId'> <br>
      <label for="categoryName">categoryName</label>
      <input
       id="categoryName"
       type="text"
       v-model = 'data.kategorija.categoryName' > <br>
      <label for="description">description</label>
      <input
       id="description"
       type="text"
       v-model = 'data.kategorija.description'> <br>
      <button @click="Put_PromenaKategorije()">izmena</button>
    </form>
  </div>
</section>
<section>
<h2>DELETE: Brisanje odredjene kategorije</h2>
  <div>
    <form>
      <label for="categoryId">id</label>
      <input
       id="categoryId"
       type="text"
       v-model = 'data.kategorija.categoryId'> <br>
      <button
       @click="Delete_BrisanjeKategorije()">
          Brisanje
      </button>
    </form>
  </div>
</section>

function Post_NovaKategorija(){
  data.kategorija.categoryId = 0;
  alert("novakta:"+JSON.stringify(data.kategorija));
  axios.post('http://localhost:51047/api/Categories',data.kategorija
)
  .then(()=>{
```

```
      alert("ok");
  })
  .catch(function (greska) {
    alert(greska);
  })
}

function Put_PromenaKategorije(){
  console.log(JSON.stringify(data.kategorija));
  alert(JSON.stringify(data.kategorija));

  axios.put('http://localhost:51047/api/Categories/'+data.kategorija
.categoryId,data.kategorija)
  .then(()=>{
    alert("ok");
  })
  .catch(function (greska) {
      alert(greska);
  })
}
function Delete_BrisanjeKategorije(){
  alert("Delete:  "+JSON.stringify(data.kategorija.categoryId));
  axios.Delete('http://localhost:51047/api/categories/'+data.kategor
ija.categoryId)
  .then(()=>{
    alert("ok");
  })
  .catch(function (greska) {
      alert(greska);
  })
  .finally(()=>alert('finally'));
}
```

Pitanja i zadaci

1. Pomoću koje funkcije JavaScript-a je moguće dobavljanje podataka sa udaljenog API-ja?
2. Napiši primer gde ćeš pokazati sintaksu funkcije fetch? Koja podešavanja znaš i kako se ona ostvaruju?
3. Kako se vrši obrada grešaka kada se koristi fetch funkcija?

4. Koja biblioteka je korišćena za dobavljanje podataka preko udaljenog API-ja?
5. Napiši primer kojim se dobavljaju podaci iz Northwind baze preko API-ja.
6. Uporedi primenu axios i fetch pristupa za dobavljanje podataka.

1.15 Globalne komponente

Postoje dva načina da se neka komponeneta koristi u Vue aplikaciji, u zavisnosti od načina na koji je registrovana: **lokalno** i **globalno**. U prethodnim primerima korišćena je **lokalna** registracija komponente. Lokalno registrovana komponenta može se upotrebljavati samo u okviru fajla u kome je uvezena.

Sa druge strane, globalno registrovane komponente se samo jednom uvoze u aplikaciju, a dostupne su u svim ostalim komponentama bez dodatnog uvoza.

Lokalna registracija ima jednu veliku prednost u odnosu na globalnu. Ukoliko komponenta koja je lokalno registrovana nije korišćena u aplikaciji, aplikacija neće ugrađivati istu u završnu verziju tzv. završni *build*. Na taj način smanjuje se veličina aplikacije i poboljšava efikasnost. Svaka komponenta ima ključ koji je naziv komponente i koji se može koristiti kao lokalni element.

Definisaje globalnih komponenti vrši se u samoj instanci aplikacije koristeći metode za dodavanje komponenti:

```
app.component('btn', myBtn)
```

Na primer, ako je globalna komponenta definisana na lokaciji `'./components/18 Global.vue'`, onda se ceo kod smešta u fajl main.js, tamo gde se i komponenta aplikacije kreira:

```
import './assets/main.css'

import { createApp } from 'vue'
import App from './App.vue'
import btn from './components/18 Global.vue'

const app = createApp(App);
app.component('btn', btn)
//app.component('drugaKomp', drugaKomp)
```

```
//...

app.mount('#app')
```

Nakon ove registracije upotreba komponente `btn` je moguća u bilo kojoj drugog komponenti aplikacije. Na primer

```
<script setup>
  import {ref } from 'vue'
  var poruka = ref('zdravo');
</script>

<template>
    {{ poruka }}
    <btn></btn>
    <btn></btn>
</template>
```

1.16 Sopstveni događaji

Osim što se neki podatak može proslediti podkomponenti putem *props*, roditeljska komponenta može da osluškuje događaje koje podkomponenta emituje. Emitovani događaji mogu sadržati dodatnu vrednost (argument), što omogućava prenos informacija iz podkomponente ka roditelju.

Ukoliko se događaj definiše direktno u sekciji šablona koristi se ugrađena metoda `$emit`, a ako se koristi u kodu onda funkcija `emit`, koja mora biti prethodno deklarisana pomoću makroa `defineEmits`.

Napisaćemo jednu komponentu, sličnu prethodnim primerima brojača, koja sa svakim klikom na dugme **šalje notifikaciju** roditeljskoj komponenti informaciju.

```
<button @click="$emit('klikNaDugme')"> test </button>
```

Roditeljska komponenta osluškuje događaj koji je definisan na standardan način koristeći direktivu **v-on** (ili skraćeni zapis sa **@**), na primer:

```
<brojac @klik-na-dugme="()=>console.log('klik od podkomponenete')">
</brojac>
```

Odnosno sa odvojenom funkcijom za obradu događaja:

```
<brojac @klik-na-dugme="roditeljskaFunkcija()"> </brojac>, gde je u kodu:
const roditeljskaFunkcija = () =>{ console.log(klik od
podkomponenete'); }
```

Ukolilko se, osim događaja, prosleđuje i neki podatak, onda se mora predvideti prihvat podatka kroz argument u deklaraciji funkcije roditeljske komponene. Argumet se prosleđuje kao drugi arugment funkcije `emit`. Na primer, ako je u podkomponenti:

```
<button @click=" $emit('klikNaDugme', brojac)" >brojac
{{brojac}}</button>
```

Onda je u roditeljkoj komponenti:

```
<brojac @klik-na-dugme ="(n)=>console.log('n='+n)"> </brojac>,  ili:
<brojac @klik-na-dugme ="akcijaKlik"> </brojac>,
gde je u kodu:
const akcijaKlik = (n)=>{
  console.log("klik na brojac " + n);
}
```

Ovde se može primeniti modifikator `.once`.

```
<brojac @klik-na-dugme.once="akcijaKlik"> </brojac>
```

U ovom slučaju, događaj se evidentira u podkomponenti svaki put, ali se prihvata i obrađuje samo jednom u roditeljskoj komponenti.

Obratite pažnju na urađenu transformaciju slova iz camelCase u cebab-case varijantu.

Ukoliko želimo da podkomponenta, osim slanja događaja, radi brojanje, onda je potrebno da se na pozovu dve akcije: funkcija koja će uraditi uvećanje brojača, a zatim slanje događaja sa podatkom. Na primer:

```
<button @click="uvecajBrojac(); $emit('klikNaDugme', brojac)" >
  brojac {{brojac}}
</button>
```

Druga opcija je da se u istoj funkciji gde se radi uvećanje brojača uradi i slanje događaja. Za ovo je potrebno da uradimo jednu pripremu, a to je da definišemo sopstvene događaje koristeći makro `defineEmits`. Važno je da se definisanje uradi na početku komponente, ne u nekoj od funkcija. Na primer:

```
<script setup>
    import {ref} from 'vue'
    const emit = defineEmits(['klikNaDugme'])
    var brojac = ref(0);

    function uvecajBrojac(){
        brojac.value = brojac.value+1;
        emit('klikNaDugme', brojac.value);
    }
</script>
<template>
   <button @click="uvecajBrojac()" >brojac {{brojac}}</button>
</template>
```

Sopstveni događaji mogu da preklapaju postojeće događaje. Na primer, ako komponenta definiše sopstveni događaj pod imenom `click` ili `prijava` taj događaj postaje važeći i ima prednost nad podrazumevanim. Zbog toga se često preporučuje da se svi događaji komponente eksplicitno definišu na jednom mestu, kako bi bili jasno dokumentovani za tu komponentu.

Slično kao u slučaju `props` validacije (kada je `props` definisan objektnom notacijom, a ne kao niz), i emitovani događaj se može validirati. Da bi dodali validaciju, događaj mora biti pridružen funkciji koja prima argumente koji se šalju pri aktiviranju događaja tj. `$emit`. Ta funkcija vraća `boolean` vrednost koja pokazuje da li je emitovani događaj validan ili ne.

Na primer, ako imamo dva događaja: `click` i `prijava`, prvi bez, a drugi sa validacijom, onda se formira objekat sa svojstvima koja odgovaraju događajima, a kojima se pridružuje funkcija za validaciju. Na primer:

```
<script setup>
const emit = defineEmits({
  // bez validacije
  click: null,

  // Validacija prijave
  prijava: ({ email, password }) => {
```

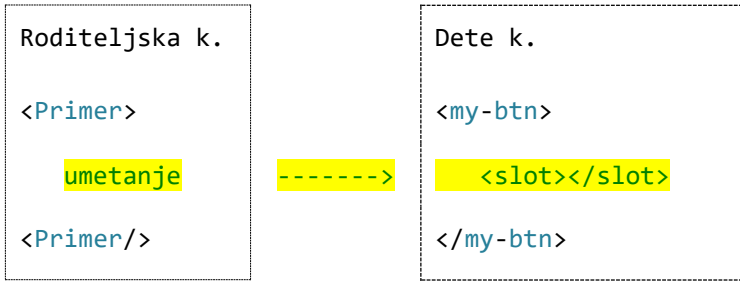
```
    if (email && password) {
      return true
    } else {
      console.warn('Nedostaju kredencijali!')
      return false
    }
  }
})

function submitForm(email, password) {
  emit('prijava', { email, password })
}
</script>
```

1.17 Slotovi

Slotovi u Vue-u predstavljaju jedan od ključnih mehanizama za kreiranje fleksibilnih i ponovo upotrebljivih komponenti. Njihova osnovna ideja je da komponenta može da definiše „prostor“ u svom šablonu u koji roditeljska komponenta ubacuje sadržaj. Na taj način se postiže razdvajanje strukture i sadržaja: komponenta obezbeđuje okvir (markup, stil, logiku), dok roditeljska komponenta definiše šta će se prikazati unutar tog okvira.

U najjednostavnijem obliku, slot je samo marker u šablonu komponente (`<slot></slot>`), ali iza tog jednostavnog taga krije se moćan koncept. Slotovi omogućavaju da se jedna komponenta koristi u različitim kontekstima, bez potrebe da se menja njen unutrašnji kod. To znači da se ista komponenta može ponašati kao generički kontejner, a da roditelj upravlja sadržajem.

Ovo se grafički može prikazati na sledeći način:



Na primer, roditeljska komponenta:

```
<script setup>
import myBtnSlot from './myBtnSlot.vue'
</script>
<template>
    <myBtnSlot class="zeleno">aaa</myBtnSlot>
    <myBtnSlot class="crveno">bbb</myBtnSlot>
</template>
<style scoped>
    .zeleno{ color:green; }
    .crveno{ color: red; }
</style>
```

koristi dete komponentu. A dete komponenta `myBtnSlot.vue` ima `<slot>` tag:

```
<template>
    <button>
      <slot></slot>
    </button>
</template>
```

Važno: Kada se sadržaj prosleđuje kroz slot, on se uvek renderuje u kontekstu roditeljske komponente. To znači da slot ima pristup podacima, reaktivnim promenljivama i metodama roditelja, a ne same komponente koja slot definiše. Ovo je važna razlika: **komponenta koja definiše slot samo obezbeđuje „prozor“, ali ne kontroliše podatke koji kroz njega prolaze.**

Vue omogućava definisanje **podrazumevanog** (rezervnog) sadržaja. Ako roditelj ne prosledi ništa, slot će prikazati ono što je definisano unutar njega. Ovo je korisno za kreiranje komponenti koje imaju smisleno ponašanje i bez dodatnog sadržaja. Na primer, dugme može imati podrazumevani tekst „Noname“, ali ako roditelj prosledi „Cancel“, taj tekst će biti prikazan umesto podrazumevanog.

```
<template>
    <button>
      <slot>Noname</slot>
    </button>
</template>
```

Na primer, pri upotrebi komponente

```
<myBtnSlot></myBtnSlot>
```

biće kreirano dugme sa podrazumevanim tekstom „Noname“, dok u slučaju kada se navede sadržaj, na primer „Cancel“ onda će ovaj sadržaj biti prosleđen i korišćen za prikaz.

Imenovani slotovi

Jedna važna karakteristika su imenovani slotovi. Oni omogućavaju da jedna komponenta ima više različitih slotova, svaki sa svojom ulogom. Na primer, komponenta stranice može imati slot za: zaglavlje, glavni sadržaj i podnožje. Roditeljska komponenta tada može precizno da odredi šta ide u koji deo. Identifikacija slotova se vrši pomoću atributa `name`, a sadržaj se prosleđuje pomoću direktive `v-slot` ili skraćeno `#`.

Sadržaj

```
<div>
  <header>
    <!-- zaglavlje -->
  </header>
  <main>
    <!--glavni sadržaj -->
  </main>
  <footer>
    <!-- podnožje -->
  </footer>
</div>
```

Komponenta

```
<template>
 <div>
  <div>
    <slot name="zaglavlje"></slot>
    <!-- zaglavlje -->
  </div>
  <div>
    <slot name="default"></slot>
    <!--glavni sadržaj, podrazumevani -->
  </div>
  <div>
    <slot name="podnozje"></slot>
    <!-- podnožje -->
  </div>
 </div>
</template>
```

Pogledajmo konkretan primer jedne komponente koja koristi tri slota: `n1`, `n2` odnosno `n3`:

```
<mySlot>
  <template v-slot:n1> aaa </template>
  <template v-slot:n2> sadrzaj </template>
  <template v-slot:n3> ccc </template>
```

```
</mySlot>
```

Pri upotrebi ovih slotova mogu se primeniti različiti scenariji. Dva osnova su: upotreba samo jednog slota, odnosno upotreba više slotova istovremeno.

- Primer upotrebe jednog slota u komponenti koja ima više, imenovanih, slotova:

```
<mySlot v-slot:n3>moji podaci</mySlot>
<mySlot>podrazumevani</mySlot>
```

- Druga opcija je da se koristi više slotova, ali uz obavezno imenovanje istih, na primer:

```
<mySlot>
  <template v-slot:n1> aaa </template>
  <template v-slot:n2> sadrzaj </template>
  <template v-slot:n3> ccc </template>
</mySlot>
```

Podrazumevani naziv slota

Ukoliko jedan `<slot>` nema atribut `name` njegovo ime je implicitno `default`, na primer:

```
<template>
 <div>
  <slot name="n1"></slot>
  <slot></slot>
  <slot name="n3"></slot>
 </div>
</template>
```

```
<mySlot>
  <template v-slot:n1>
    <h3 class="zeleno">Zaglavlje</h3>
  </template>
  <template v-slot:default>
    <h3>Glavni sadržaj</h3>
    <p>lorem...</p>
  </template>
  <template #n3>
    <p class="crveno">Podnožje</p>
  </template>
</mySlot>
```

Važno! Ako se radi imenovanje onda se koristi tag `template`. Atribut `v-slot` može da se doda jedino u tom tagu.

Slotovi se mogu referisati i dinamički upotrebom uglastih zagrada, slično atributima. Na primer:

```
#[state.slot]  ili
v-slot:[state.slot]
```

Pristup podacima

Videli smo da pristup do podataka unutar samog slotu nije podrazumevano moguć. Ipak, stoji da bi takav pristup mogao da se iskoristi u nekim slučajevima. Dakle, hajde da sada vidimo kako jedan slot ipak može da pristupi podacima.

Podaci koji se prosleđuju slotu prosleđuju se u trenutku kada se vrši renderovanje tj. slično kao što se prosleđuju svojstva (props) jednoj komponenti. Uporedimo sve slučajeve.

1. Prvi slučaj tj. **klasično prosleđivanje podataka**.
   U ovom slučaju roditeljska komponenta sama definiše šta će se ubaciti u slot:

Roditeljska komp.

```
<template> <script setup lang="ts">
  import {ref} from 'vue'
  import myBtn2 from  './myBtn2.vue'
  const poruka = ref("zdravo")
  const brojac = ref(3);
</script>
    <my-btn-2 >
        {{poruka}} - {{brojac}}
    </my-btn-2>
</template>
```

myBtn2.vue

```
<template>
    <div>
        <slot></slot>
    </div>
</template>
```

   - Dakle, roditelj kontroliše podatke (`poruka`, `brojac`).
   - **Analogija**: slot je definisan prostor u koju roditeljska komponenta ubacuje sadržaj.

2. Prosleđivanje podataka **od slot-komponente ka roditelju** (`slot props`).
   Sama komponenta koja sadrži tag `slot` tj. dete-komponenta može da ima kontrolu nad podacima tj. da ih vrati roditeljskoj kontroli. Pogledajmo sledeći primer:

Roditeljska komp.

```
<script setup lang="ts">
import myBtn2 from  './myBtn2.vue'
</script>

<template>
  <my-btn-2 v-slot="slotProps">
     {{slotProps.brojac}}
     {{slotProps.poruka}}
  </my-btn-2>
</template>

<style scoped>
</style>
```

myBtn2.vue

```
<script setup lang="ts">
</script>

<template>
<div>
  <slot
     poruka="zdravo"
     brojac="101">
  </slot>
</div>
</template>

<style scoped>
</style>
```

- Dete komponenta definiše props za slot (`poruka`, `brojac`).
- Roditeljska komponenta ih prima kroz `v-slot="slotProps"`
- Roditelj može da koristi podatke koje dete-komponenta generiše.
- **Analogija**: dete daje roditelju podatke, a roditelj odlučuje kako će ih prikazati. Ovaj pristup omogućava da roditelj prilagodi prikaz dok dete kontroliše logiku.

U narednom poglavlju uporedićemo standardnu primenu komponente koja sadrži celu logiku sa primenom slotova sa svojstvom props.

Primer – lista studenata

Standardna komponenta

Prvo ćemo pokazati slučaj koji je zasnovan na primeni standardne komponente myList za prikaz liste studenata. Ova komponenta enkapsulira celu logiku za prikaz, dok roditeljska komponenta samo definiše poziciju gde se myList primenjuje:

Roditeljska komponenta:

```
<script setup>
  import myList from  './myList.vue'
</script>
<template>
  <myList> </myList>
</template>
<style scoped> </style>
```

Komponenta myList.vue

```
<script setup>
import {reactive} from 'vue'
const studenti = reactive([
    { ime: 'Jovica J', indeks: 'NRT-11/11' },
    { ime: 'Perica P', indeks: 'RIN-1/2' }
]);
</script>
```

```
<template>
  <ul>
    <li
      v-for="student in studenti"
      :key="student.indeks">
       {{student.ime}}
    </li>
  </ul>
</template>
```

```
<style scoped>
  ul {
    list-style-type: none;
    padding: 5px 10px;
    background: linear-gradient(315deg, #514380 25%, #c8d1ff);
  }
  li {
    padding: 5px 20px;
    margin: 10px 20px;
    background: rgb(221, 243, 180);
  }
</style>
```

Fleksibilan prikaz sa slot props-om

Roditeljska komp.

```
<script setup lang="ts">
  import myList from  './myListNew.vue'
</script>
```

```
<template>
  <myList #std="{ime, indeks}"  class="c1">
      <p>{{ ime }}</p>
      <h4>
        {{ indeks }}
      </h4>
  </myList>
</template>
```

```
<style scoped>
  .c1 {
```

```
    list-style-type: none;
    padding: 10px 10px;
    background: linear-gradient(315deg, #514380 25%, #c8d1ff);
  }
</style>
```

U ovom slučaju roditeljska komponenta primenjuje dete-komponentu myList koja sadrži slot. Dete komponenta koristi slot u petlji i vraća podatke roditeljskoj komponenti u svakom prolazu.

Sadržaj elementa myList se prenosi slotu. U ovom slučaju taj sadržaj je dinamički. Ono što je ovde značajno da se taj sadržaj preuzima iz slota dete-komponente, oblikuje za prikaz u roditeljskoj komponenti, a na kraju se vraća za prikaz u dete-komponenti tj. samom slotu.

U nastavku dajmo i kod nove dete-komponente myList.vue.

```
<script setup lang="ts">
   import {reactive} from 'vue'
   const studenti = reactive([
     { ime: 'Jovica J', indeks: 'NRT-11/11' },
     { ime: 'Perica P', indeks: 'RIN-1/2' }
]);
</script>
```

```
<template>
<ul>
  <li class="cls1" v-for="student in studenti"
    :key="student.indeks">
    <slot name="std" v-bind="student"></slot>
  </li>
</ul>
</template>
```

```
<style scoped>
  .cls1 {
    list-style-type: none;
    padding: 5px 20px;
    margin: 10px 20px;
    background: rgb(221, 243, 180);
}
</style>
```

1.18 Ubrizgavanje podataka

Kada treba da prosledimo podatke sa roditeljske na podređenu komponentu, najčešće koristimo svojstva komponenata - `props`. To je osnovni mehanizam Vue-a za komunikaciju između komponenti.

Međutim, zamislite situaciju u kojoj postoji duboka hijerarhija komponenti. Ako podatak sa vrha treba da stigne do komponente na dnu, mora se prosleđivati kroz ceo lanac, čak i kroz komponente koje ga same ne koriste.

Na primer, ako imamo hijerarhiju komponenti:

```
Root
└─ Lista
   ├─ Stavka
   └─ Zaglavlje
      ├─ OK_Button
      └─ Stat_Data
```

Pri tome u komponenti `Lista` postoje statistički podaci (`proc` i `type`) koje treba da prosledimo komponenti `Stat_Data` odnosno komponenti `OK_Button`. Ovo je moguće uraditi koristeći `props` deklaracije kroz lanac komponenti: `Lista->Stavka->Zaglavlje->Stat_Data/OK_Button`. Pri tome, komponenta `Zaglavlje` samo prosleđuje podatak. Ovo dovodi do problema da međukomponente koje ne koriste podatke moraju da ih deklarišu odnosno da ih prosleđuju (*eng. prop drilling*).

Vue nudi par komandi ***`provide/inject`***, kao jedan elegantan način da se prethodni problem prevaziđe. Nadređene komponente mogu poslužiti kao dobavljač zavisnosti za sve pod-komponente, bez obzira na to koliko je duboka hijerarhija komponenti. Ova funkcionalnost je podeljena na dva dela:

- roditeljska komponenta ima opciju pružanja podataka ***`provide`***, a
- podređena komponenta opciju ubrizgavanja ***`inject`*** da bi mogla da koristi podatke.

Ovaj mehanizam može da posluži i za **kreiranje globalnih podataka** u aplikaciji.

Funkcija *Provide*

Obezbeđivanje podataka omogućen je funkcijom **provide**. Funkcija ima dva argumenta, prvi je **ključ** a drugi **vrednost** koja se prenosi. Vrednost može biti prostog ili objektnog tipa. Na primer:

```
import {provide} from 'vue'
provide("ključ","vrednost");
```

Najpre pogledajmo korensku komponentu kao i komponentu koja će obrađivati podatke odnosno komponente App.vue i Lista.vue pre promena:

App.vue – početna varijanta

```
<script setup lang="ts">
 import list from  './Lista.vue'
</script>

<template>
  <list></list>
</template>

<style scoped></style>
```

Lista.vue

```
<script setup lang="ts">
import OkButton from './OkButton.vue'
import StatData from './StatData.vue'
</script>

<template>
  <ok-button></ok-button> <br>
  <stat-data></stat-data>
</template>

<style scoped></style>
```

Dakle, na osnovu prethodno obrazloženog, potrebno je da podatke postavimo na raspolaganje svim komponentama počev od korenske komponente. U našem slučaju to će biti jedna poruka (message) i jedan objekat (stat).

Uključujemo funkciju **provide** i primenjujemo je, obavezno navodeći kao prvi argument ključ za prosleđene podatke.

App.vue – sa primenom funkcije `provide`

```
<script setup>
   import {ref, reactive, provide} from 'vue'
   import Primer from './components/PrimerD.vue'
   const poruka = ref("poruka za sve");
   const stat = reactive({count:101, proc:33, type:'comp'});
   provide("message", poruka );
   provide("stat", stat);
```

```
</script>
<template>
    <Primer />
</template>
<style scoped></style>
```

Funkcija *Inject*

Drugi deo prenosa padataka ovim načinom je ubrizgavanje podataka (eng. *inject*). Komponente koje primenjuju `inject` nemaju informaciju od koje komponente podaci dolaze. Podaci koji se prihvataju mogu biti: prosti kao što je **message**, odnosno objekti kao što je **stat**. U naredne dve komponente je urađena primena **inject** podataka, posebno za **message** odnosno **stat** podatak.

OkButton.vue

```
<script setup lang="ts">
import {inject} from 'vue'
const poruka = inject('message')
</script>

<template>
  <button type=""> {{poruka}}
  </button>
</template>

<style scoped></style>
```

StatData.vue

```
<script setup lang="ts">
import {inject} from 'vue'
const stat = inject('stat')
</script>

<template>
    <div>
      <h4>{{stat.type}}</h4>
      {{stat.proc}}
    </div>
</template>
<style scoped>
</style>
```

Na ovaj način omogućeno je **direktno ubrizgavanje** (eng. dependency injection) kojim se postiže da roditeljska komponenta ne mora da brine koja komponenta i na koji način koristi podatke koji se izlažu. Istovremeno deca komponente ne brinu u kom trenutku i od koje roditeljske kontrole stižu podaci.

Funkcija za ubrizgavanje podataka može da sadrži i podrazumevanu vrednost. Vrednost će biti korišćena samo ako realna vrednost nije dostupna od neke roditeljske komponente. Podrazumevana vrednost se navodi kao drugi argument funkcije inject, na primer:

```
const poruka = inject('message', 'poruka nije definisana')
```

Ako je potrebno da se postigne **reaktivnost** u ovom konceptu, preporuka je da se sve promene rade unutar provajdera kada god je to moguće. Ipak, postoje slučajevi kada je neophodno omogućiti promenu podataka iz dete komponente, tj. komponente koja sadrži `inject`. U ovom slučaju **preporuka je da se promena izvodi preko funkcija koje su odgovorne za izmene stanja**.

U nastavku prikazujemo ovaj način upravljanja sadržajem iz dete-komponente. Roditeljska komponenta treba da se proširi funkcijama za promenu stanja komponente. Ove funkcije se prosleđuju zajedno sa podacima. Zbog toga funkcija `provide` daje podrazumevani imenovani pristup dok se isto mora uraditi sa `inject` funkcijom.

Na primer, u našem slučaju za izmenu poruke, to bi izgledalo ovako:

App.vue – promena iz dete-komponente

```
<script setup>
   import {ref, reactive, provide} from 'vue'
   import Primer from './components/Primer.vue'
   const poruka = ref("poruka za sve");
   const stat = reactive({count:101, proc:33, type:'comp'});
   provide("message", {
      poruka,
      promenaPoruke
   });
   function promenaPoruke(newMessage:string){
      poruka.value = newMessage;
   }
   provide("stat", stat);
</script>
<template>
    <Primer />
</template>
<style scoped></style>
```

Primer.vue

```
<script setup>
import list from  './Lista.vue'
</script>
```

```
<template>
  <list></list>
</template>
<style scoped></style>
```

Lista.vue

```
<script setup>
import {inject} from 'vue'
import OkButton from './OkButton.vue'
import StatData from './StatData.vue'

const {poruka, promenaPoruke }  = inject('message')
promenaPoruke('Ubrizgano iz komp.Lista')
</script>

<template>
  <ok-button></ok-button> <br>
  <stat-data></stat-data>
</template>
<style scoped></style>
```

1.19 Prosleđivanje atributa

Komponente mogu da primaju atribute i događaje, čak i kada oni nisu eksplicitno definisani preko `props` (za atribute) ili `emits` (za događaje). Tipični primeri takvih atributa su: class, style i id.

Ako komponenta sadrži jedan korenski element (jedan element u template sekciji), svi prosleđeni atributi automatski se dodaju tom elementu.

U slučaju da komponenta ima više korenskih elemenata, prosleđivanje atributa mora biti eksplicitno definisano

U nastavku je data dete-komponenta `mybutton` koja ima samo jedan korenski element, tj. `button`:

```
<template>
```

```
    <button class="fntClr"> my button1 </button>
</template>
<style scoped>
    .fntClr{ color: blueviolet; }
</style>
```

Ovaj element (`button`) primiće atribute od roditeljske komponente. Ako su to atributi `class` i `style` onda se prosleđene vrednosti dodaju sopstvenim klasama i stilovima same komponente, na primer:

```
<template>
    <mybutton style="background-color:yellow"> </mybutton>
    <mybutton class="cyan" ></mybutton>
    <hr>
</template>

<style scoped>
  .cyan{ background-color: lightcyan; }
</style>
```

Dobija se:

Isto pravilo važi i za `v-on` osluškivače događaja. Navedeni događaj biće prenet na korenski element podkomponente:

```
<MyButton @click="onClick" />
```

Funkcija za obradu događaja `click` biće dodata događajima za korenski element komponente `<MyButton>`, tj. na sam HTML `<button>` element. Kada se klikne na HTML `<button>`, događaj trigeruje `onClick` metod roditeljske komponente. Ako HTML element `<button>` već ima osluškivač takođe vezan direktivom `v-on`, oba osluškivača se trigeruju.

Atributi i događaji se prosleđuju korenskom elementu ili korenskoj pod-komponenti. Prosleđeni atributi ne uključuju atribute i događaje koji su deklarisani sa `props` ili `v-on`. Drugim rečima, deklarisani `props` i slušaoci su "potrošeni" od strane `<MyButton>`.

Ako ne želite da jedna komponenta automatski prosleđuje atribute, može se postaviti `inheritAttr:false` u opcijama komponente.

```
<script setup>
  defineOptions({
    inheritAttrs: false
  })
// ...setup logic
</script>
```

Naravno, postavlja se pitanje zašto bi se uopšte prosleđivao neki atribut ako ga nije potrebno obraditi.

Najčešći razlog je potreba da pod-komponente koje su dublje u lancu nasleđivanja prihvate podatke od roditeljske. U tom slučaju, prvi element u lancu podelemenata koji treba da prihvati poslate atribute, kao što su: `class`, `style`, `v-on` osluškivače, ne uključujući `props` i `emits` svojstva, treba da označi klasu koju prihvata i da ima povezano svojstvo `$attrs` koristeći `v-bind`, na primer.

```
<div class="btn-wrapper">

  <button class="btn" v-bind="$attrs">click me</button>
</div>
```

Evo i konkretnog primera iz koga je ovo očigledno:

App.vue

```
.Clr{ color: blue; }
.bckClr{ background-color: yellow;}

<template>
   <MyBtn1 class="Clr bckClr"></MyBtn1>
</template>
```

MyBtn1

```
defineOptions({
   inheritAttrs: false
})
<template>
   <div class="bckClr Clr">lorem...
      <MyBtn2></MyBtn2>
   </div>
</template>
```

MyBtn2

```
<template>
  <div class="bckClr" v-bind="$attrs">
    ...
```

```
  </div>
</template>
```

Klase `Clr`, `bckClr` prosleđuju se počev od korenske komponente App.vue. Zatim komponenta `MyBtn1` **blokira prihvat atributa, ali se oni dalje prenose** na komponentu `MyBtn2` pri čemu se koristi jedna od navedenih klasa `bckClr`.

Napomena. Počev od verzije Vue3. Vue komponente mogu da imaju više korenskih elemenata, na primer:

```
<template>
   <MyBtn1>...</MyBtn1>
   <MyBtn2>...</MyBtn2>
</template>
```

U ovom slučaju, kada pokušate da koristite prosleđivanje atributa na prethodni način javlja se upozorenje (eng. *warning*), osim ako se definiše **koji korenski element prihvata atribute roditelja**. Dakle, ovde se takođe mora definisati koji elementi/komponente prihvataju atribute, na primer:

```
<template>
   <MyBtn2>...</MyBtn2>
   <MyBtn2 v-bind="$attrs">...</MyBtn2>
</template>
```

Na kraju, upotreba atributa u određenoj komponenti se može definisati i preko JavaScript sekcije koristeću ugrađenu funkciju `useAttrs()`:

```
<script setup>
import { useAttrs } from 'vue'

const attrs = useAttrs()
</script>
```

1.20 Pitanja i zadaci

1. Kako se kreiraju sopstveni događaji neke komponente, odnosno kako se vrši obrada takvih događaja od strane roditeljske komponente?
2. Objasniti pojam globalnih komponenti.
3. Šta su slotovi? Kako se koriste?
4. Koja je sintaksa za imenovane slotove? Da li se može koristiti slot bez naziva zajedno sa imenovanim slotovima i kako?
5. Kako se vrši povezivanje podataka u slučaju slotova? Da li dete-komponenta može da prenese podatke roditeljskoj komponenti i na koji način?
6. Šta znači ubrizgavanje podataka? Koje su prednosti ove tehnike?
7. U kontekstu prenosa podataka, koja komponenta definiše *provide*, a koja *inject* funkcije?
8. Da li se može obezbediti reaktivnost podataka koji se ubrizgavaju?
9. Koja je preporuka za upravljanje reaktivnošću podataka?
10. Da li se osobina prosleđivanja svojstava na isti način odnosi na i na događaje?
11. Ako roditeljska komponenta ima više od jedne pod-komponente, šta se događa sa prosleđivanjem?
12. Na koji način se može sprečiti preuzimanje prosleđenih svojstava?
13. Na koji način se može odrediti kojoj pod-komponenti se vrši prosleđivanje svojstava?
14. Napišite primer kojim možete pokazati sve osobine prosleđivanja događaja.

Dodatak: Korišćenje TypeScript-a

Jedno celo poglavlje ove knjige posvećeno je TypeScript jeziku. Ako ga već ne poznajete, preporuka je da se najpre upoznate sa osnovama jezika, jer će vam to značajno olakšati razumevanje narednih primera i koncepata. TypeScript je superset JavaScript-a, što znači da sve što važi za JS važi i za TS, ali uz dodatnu prednost statičke tipizacije i bolje podrške u razvoju.

Za pokretanje novog Vue projekta koristi se zvaničan scaffolding alat, koji se oslanja na Vite – modernu alatku za brzo kreiranje aplikacija. Vite generiše osnovnu strukturu foldera i nekoliko početnih komponenti, što vam omogućava da odmah krenete sa radom.

Pokretanje projekta izgleda ovako:

npm create vue@latest

Nakon pokretanja komande, dobićete interaktivni meni sa opcijama. Tu je važno da uključite TypeScript podršku:

```
┌  Vue.js - The Progressive JavaScript Framework
│
◇  Project name (target directory):
│  vue-project
│
◆  Select features to include in your project: (↑/↓ to navigate,
space to select, a to toggle all, enter to confirm)
│  ■ TypeScript
│  □ JSX Support
│  □ Router (SPA development)
│  □ Pinia (state management)
│  □ Vitest (unit testing)
│  □ End-to-End Testing
│  □ ESLint (error prevention)
│  □ Prettier (code formatting)
```

```
◇  Select experimental features to include in your project:
(↑/↓ to navigate, space to select, a to toggle all, enter to
confirm)
│  none
│
◇  Skip all example code and start with a blank Vue project?
│  No

Scaffolding project in D:\VETS\PABP\Predavanja\2025\1-8
VueJS\2\castest\vue-project...
│
└  Done. Now run:

   cd vue-project
   npm install
   npm run dev

| Optional: Initialize Git in your project directory with:

   git init && git add -A && git commit -m "initial commit"
```

Napomena. I dalje preporučujemo upotrebu IDE VisualStudio Code. Uz VS Code treba dodati aktuelne alatke tj. ekstenzije. Ove alatke nude pomoć u vidu sintaksne provere i dopune koda.

Za početak, zapazite da u sekciji koda stoji:

```
<script setup lang="ts">
```

Sekcija *template*

Osim u sekciji za script, TypeScript se može koristiti i u sekciji šablona. Pri tome se navodi

```
<script setup lang="ts">
```

Imajući u vidu da je TS superset od JS, može se koristiti ista sintaksa kao u slučaju JS. Na primer, ako se definiše `brojac` koji je tipa `number`:

```
var brojac:number = ref(props.pocetnaVrednost);
```

Svojstvo *props*

Pogledajmo zatim uvođenje TS u pisanju svojstava - **props**. Najpre samo zamenimo tipove:

```
const props = defineProps({
    btnId: Number,
    naslov : String,
    pocetnaVr: Number
})
```

Ili koristimo tzv „runtime“ tipiziranje, npr:

```
const props = defineProps(['btnId','naslov','pocetnaVr'])
```

Uvodjenjem objektnih tipova `Number`, `String` možemo prihvatiti sve vrednosti pa i `undefined` što se u našem slučaju i događa pri podizanju komponente.

Ipak TS nudi makro koji se oslanja na generički tip koji odgovara svojstvima, dakle:

```
const props = defineProps<{
    btnId?: number,
    naslov? : string,
    pocetnaVrednost?: number
}>()
```

Čest slučaj je i primena interno definisanog interfejsa, na primer:

```
interface Props{
    btnId?: number,
    naslov? : string,
    pocetnaVrednost?: number
}
const props = defineProps<Props>()
```

Ovde treba da naglasimo da generička primena interfejsa, koja bi bila prosleđena svojstvu defineProps ne može biti importovana, na primer

```
import { Props } from './other-file'
~~defineProps<Props>()~~
```

Primena podrazumevanih vrednosti zahteva primenu posebne metode `withDefaults`:

```
const props = withDefaults(defineProps<Props>(), {
    naslov: "BTN"
    //naslov: ()=>{ "BTN" }
})
```

Makro *defineEmits*

Pogledajmo sada kako se radi tipizacija sopsvenih događaja tj. upotreba **emit** makroa. U toku izvršavanja dovoljno je samo navesti imena, kao u JS, na primer:

```
const emit = defineEmits(['specijalniKlik', 'dogadjaj2']);
```

U slučaju tipizacije događaja, kod bi bio sledeći:

```
const emit = defineEmits<{
  (e:'specijalniKlik',id:number):void //naziv sv. nije bitan
  (e:'dogadjaj2'):void
}>();
```

Funkcije *ref* i *reactive*

Na osnovu inicijalne vrednosti definiše se tip koji se koristi kao ref promenljiva, na primer:

```
// podešava se imlicitno Ref<number>
const starost = ref(22)
starost.value = '22' // problem
```

Na sličan način tipizira se i promenljiva koristeći reactive.

```
// implicitno definisan tip: { indeks: string }
const student = reactive({ indeks: 'NRT-88/88' })
```

Pošto se u reactive definišu objekti, onda se za eksplicitno definisanje tipa koristi određeni interfejs, na primer:

```
interface Student{
    indeks: 'NRT-88/88',
    godina?:number
}
const student:Student = reactive({ indeks: 'NRT-88/88' })
```

Nije preporučljivo koristiti generički argument jer je vraćeni tip drugačiji od generičkog argument tipa.

Ako je tip poznat može se definisati unapred, koristeći type Ref i standardna TS pravila:

```
import type {Ref} from 'vue'

const starost: Ref<string | number> = ref('22')
starost.value = 22
starost.value = '22'
```

Ili se mo\e koristiti generički argument za zadavanje tipa, na primer:

```
const starost = ref<string | number>('22')
starost.value = 22
starost.value = '22'
```

Podrazumevano, svaki tip uključuje i undefined tip kao mogućnost, tj ref<number> odgovara Ref<number|undefined>.

Funkcija *computed*

Ova funkcija dobija tip od povratne vrednosti koja se računa. Moguće je eksplicitno definisanje tipa povratne vrednosti.

```
interface Student{
    indeks: 'NRT-88/88',
    godina:number
}
const student:Student = reactive({ indeks: 'NRT-88/88', godina:2 })
const semestar = computed(() => student.godina * 2)
const result = semestar.split('') //greska
```

ili

```
const semestar2 = computed<number>(() => student.godina * 2)
```

2. .NET - EF

U ovom poglavlju predstavljeni su osnovni pojmovi vezani za platformu **.Net**, koja će biti temelj za sve primere i projekte u nastavku udžbenika. Fokus je na razumevanju: ključnih alata, razvojnih okruženja, šablona projekata i postupka izgradnje aplikacija.

Pored toga, upoznaćemo se sa LINQ (eng. Language Integrated Query) – univerzalnim jezikom za upite zasnovanim na C# sintaksi, kao i sa razvojem prvog projekta korišćenjem biblioteke **Entity Framework** – skr. **EF**. Poseban akcenat biće stavljen na **CRUD operacije** (eng. Create, Read, Update, Delete), rad sa povezanim podacima i specifičnosti brisanja podataka.

.NET je razvojna platforma koju je kreirala kompanija **Microsoft**, a koristi se kao osnova za savremene i buduće softverske sisteme – kako unutar Microsoft ekosistema, tako i kod drugih multinacionalnih kompanija.

Platforma omogućava razvoj:

- univerzalnih Windows aplikacija,
- mobilnih aplikacija za **iOS**, **Android**,
- visoko pouzdanih sistema, uključujući i sisteme za kontrolu leta.

Savremeni razvoj .NET platforme odvija se u dva pravca, od kojih je jedan **otvorenog koda** – poznat kao **.Net**. .Net se može koristiti na operativnim sistemima koji nisu Windows, uključujući Linux i macOS. .Net je posebno pogodan za:

- razvoj veb aplikacija,
- komponenti i servisno orijentisanih aplikacija,
- rad u cloud okruženju.

EF (eng. – Entity Framework) je moderna ORM (eng. Object-Relational Mapping) biblioteka za rad sa bazama podataka u okviru .Net platforme. Omogućava rad sa podacima kroz objektno-orijentisane paradigme, bez potrebe za direktnim pisanjem SQL (eng. Structured Query Language) upita.

Ključne prednosti EF:

- Efikasniji razvoj aplikacija korišćenjem C# jezika umesto SQL-a;
- Platformska nezavisnost – podrška za Windows, Linux i macOS;
- Pogodnost za razvoj desktop, veb, cloud, mobilnih, gejming, IoT i AI aplikacija.

Za rad sa .Net i EF preporučuje se korišćenje sledećih razvojnih alata:

Tabela 2.1. Pregled alata za razvoj

Alat	*Operativni sistem*	*Napomena*
Visual Studio	Windows, macOS	Puna integracija sa .NET
Visual Studio Code	Windows, Linux, macOS	Lagan, fleksibilan editor

2.1 Objektno-relaciono mapiranje

Objektno-relaciono mapiranje - ORM (eng. Object-Relational Mapping) predstavlja savremeni pristup u radu sa podacima, zasnovan na objektno-orijentisanom programiranju. Umesto direktnog rada sa SQL upitima i tabelama, programeri koriste objekte i klase koji se automatski mapiraju na elemente u bazi podataka.

Na strani aplikacije, podaci se definišu kroz modele – klase koje predstavljaju entitete. Na strani baze podataka, ti podaci se čuvaju u tabelama, kolonama i redovima, u skladu sa pravilima konkretne baze.

ORM omogućava:

- apstrakciju baze podataka kroz objektni kod,
- lakšu manipulaciju podacima bez direktnog pisanja SQL-a,
- platformsku fleksibilnost, uključujući podršku za relacione i nerelacione baze.

U ovom udžbeniku fokusiraćemo se prvenstveno na relacione baze podataka, iako EF Core podržava i NoSQL (eng. Not Only SQL) sisteme.

ORM vrši **mapiranje objektnih koncepata** u koncepte skladišta podataka. Sledeća tabela prikazuje osnovne paralele između elemenata .NET aplikacije i struktura u relacionoj bazi podataka:

Tabela 2.2. Mapiranje objektnih elemenata na relacionu bazu

Relaciona baza	*.Net*
Tabela	Klasa
Kolona tabele	Svojstvo/polje klase
Redovi	Elementi .net kolekcije
Primarni ključ, jednistveni red	Jednistvena instanca klase
Strani ključ	Referenca ka drugoj klasi
SQL upit	.NET LINQ

Korišćenjem ORM-a, razvoj aplikacija postaje:

- brži – jer se eliminiše potreba za ručnim pisanjem SQL upita,
- bezbedniji – kroz automatsku zaštitu od SQL injekcija,
- održiviji – jer se promena baze ne mora nužno odraziti na aplikacioni kod.

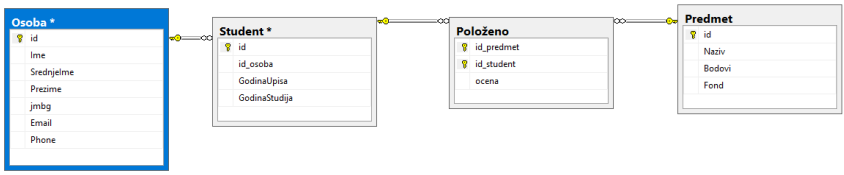
ORM biblioteke poput Entity Framework omogućavaju da se sve operacije nad bazom – uključujući kreiranje, čitanje, izmenu i brisanje (CRUD) – obavljaju kroz C# kod, uz punu kontrolu nad ponašanjem aplikacije.

Konceptni model

Svaka aplikacija koja radi sa podacima poseduje konceptualni model podataka, bilo da je on eksplicitno definisan kroz klase i strukture, ili implicitno kroz način

na koji se podaci obrađuju. Ovaj model predstavlja način na koji aplikacija vidi i koristi podatke, nezavisno od njihove fizičke organizacije u bazi.

Pretpostavimo sledeću šemu baze podataka, kao na slici, u kojoj se entiteti kao što su: Student, Osoba, Predmet i Položeno, nalaze u odvojenim tabelama.



Slika 2.1. Primer šeme podataka

Pogledajmo kako bi izgledao jedan SQL upit za gornji primer. Neka se upitom generiše prikaz studenta treće koji su upisani posle 2015. godine, a upit treba da prikaže imena i godinu studija studenta.

```
SELECT o.Ime, s.GodinaStudija
FROM Student s
INNER JOIN Osoba o ON o.id = s.id
WHERE s.GodinaUpisa >= '2015-01-01'
AND s.GodinaStudija = 3
```

U ovom primeru koristi se jedan JOIN kako bi se povezale informacije iz dve tabele.

Problemi tradicionalnog pristupa

Ako proširimo zahtev uslovom da se prikažu položeni ispiti sa ocenom 10, upit bi imao tri JOIN-a:

```
SELECT o.ime, o.prezime, s.indeks, pr.naziv
FROM  Polozeno po
INNER JOIN Predmet pr
       ON po.id_predmet = pr.id
INNER JOIN Student s
       ON s.id = po.id_student
INNER JOIN Osoba o
       ON o.id = s.id
```

```
WHERE po.ocena = 10
AND   s.GodinaStudija = 3
```

Očigledno da upit postaje komplikovan za primenu. Možemo reći da je Db šema podataka u bazi izdeljena i da nije prilagođena aplikaciji. Osim što je potrebno razumeti model baze podataka, teško je pratiti relacije između tabela u bazi. Dakle, bilo bi od koristi:

- Iskoristiti aplikativne zahteve za drugačije modelovanje, iako postoje već kreirane Db šeme.
- Izbeći da se Db šema provlače kroz aplikativni kod.

Tradicionalni model rada sa podacima oslanja se na strukturu baze: tabele, pogledi, uskladištene procedure i relacije. Ovakav pristup zahteva:

- detaljno poznavanje šeme baze podataka,
- ručno praćenje relacija između entiteta,
- direktno uključivanje strukture baze u aplikativni kod.

Zbog toga se preporučuje:

- **modelovanje podataka prema aplikativnim zahtevima**, čak i kada već postoji definisana šema baze,
- **izbegavanje direktne zavisnosti aplikacije od strukture baze**.

Primena **Entity Framework-a** omogućava korišćenje **objektno-orijentisanog modela**, koji je znatno pogodniji za razvoj aplikacija. Ovaj model uključuje:

- objekte i njihova ponašanja,
- svojstva i nasleđivanje,
- kompleksne tipove i navigaciju među entitetima.

Model podataka koji koristimo zasniva se na **entitet-relacija** pristupu. Osnovni pojmovi su:

- **Tip entiteta (eng. Entity Type)** – struktuirani zapis sa ključem;
- **Entitet** – konkretna instanca tipa entiteta;
- **Skup entiteta (eng. Entity Set)** – kolekcija entiteta istog tipa;
- **Nasleđivanje** – jedan tip entiteta može naslediti drugi.

Tip podataka *EntityType*

Tip podataka u modelu definiše se kao **EntityType**. Na primer, u aplikaciji možemo imati `Customers` i `Employees` kao skupove entiteta. Tipovi entiteta mogu biti opisani i kroz XSD šeme.

EntityType se sastoji od tri komponente:

- **Key** – svojstvo ključa;
- **Property** – obična svojstva entiteta;
- **Navigation Property** – veze ka drugim entitetima.

Svojstvo ključa *Key*

Definiše koja svojstva formiraju identitet tj. jedinstvenost entiteta.

Može biti sačinjen od više svojstava tj. može biti složen.

Osnovna svojstva entiteta *Property*

U okviru Entity Framework modela, svojstva (eng. Properties) predstavljaju osnovne elemente entiteta. Svako svojstvo je definisano:

- imenom,
- tipom podataka.

Tipovi podataka koji se koriste za definisanje svojstava nazivaju se prostim tipovima (eng. simple types). To su osnovni tipovi koji najviše odgovaraju tipovima iz .NET Framework-a, kao što su: int, string, DateTime, bool, itd.

Važno je napomenuti da se ovi tipovi koriste isključivo za definisanje svojstava entiteta, a ne za navigaciju ili relacije. Osnovna svojstva entiteta mogu se pregledati i kroz Properties prozor u razvojnom okruženju, što olakšava vizuelnu inspekciju modela.

Asocijacije - veze između entiteta

Asocijacije definišu **logičke veze između entitetskih tipova**. Iako su slične relacijama između tabela u relacionim bazama podataka, asocijacije se odnose na objektni model i imaju dodatne semantičke karakteristike.

Svaka asocijacija obuhvata:

- **krajnje tačke** – entitete koji učestvuju u vezi,
- **multiplikaciju** – broj instanci koje učestvuju u vezi (npr. jedan-na-više, više-na-više).

U automatski generisanim modelima, asocijacije se formiraju na osnovu relacija u bazi podataka. Na primer, u **Northwind** modelu postoji asocijacija između

entiteta `Products` i `Categories`, koja ukazuje na postojanje veze između proizvoda i kategorije kojoj pripadaju.

Navigaciona svojstva

Navigaciona svojstva (eng. Navigation Properties) su direktno povezana sa asocijacijama. Ona omogućavaju **navigaciju između povezanih entiteta** i predstavljaju objektne veze u okviru modela.

Kroz navigaciona svojstva, entitet može da pristupi drugim entitetima sa kojima je povezan. Na primer:

- Entitet Categories ima navigaciono svojstvo Products, koje omogućava pristup svim proizvodima u toj kategoriji.
- Entitet Products poseduje navigaciona svojstva kao što su Categories, Order_Details, Suppliers.

Za razliku od skalarnog svojstva koje predstavlja jednu vrednost (string, int, itd.), navigaciono svojstvo opisuje putanju ka drugom entitetu.

Svako navigaciono svojstvo je povezano sa odgovarajućom asocijacijom u modelu. Svojstvo Association sadrži informacije o:

- tipu veze,
- entitetima koji učestvuju,
- načinu navigacije.

Ove informacije su ključne za pravilno mapiranje relacija i omogućavaju EF da automatski generiše potrebne JOIN operacije prilikom izvršavanja LINQ upita.

Klase modela

Kada se generiše model iz baze podataka, EF automatski kreira klase koje predstavljaju entitete. Svaka tabela u bazi dobija odgovarajuću klasu, a kolone se mapiraju na svojstva te klase.

```
using (var context = new NorthwindEntities())
{
    var employees = context.Employees;
    foreach (var employee in employees)
    {
```

```
        string ime = employee.FirstName;
        string prezime = employee.LastName;
    }
}
```

U ovom primeru:

- `NorthwindEntities` je kontekst baze podataka,
- `Employees` je skup entiteta (EntitySet),
- `FirstName` i `LastName` su svojstva klase `Employees`.

Klase se mogu automatski generisati iz baze, ali se po potrebi mogu i ručno modifikovati radi dodavanja logike, validacije ili anotacija

Dobavljanje podataka

Jedna od ključnih prednosti EF -a je mogućnost dobavljanja podataka pomoću LINQ izraza, koji omogućavaju pisanje upita u C# sintaksi, bez direktnog korišćenja SQL-a.

Na primer, ako je potrebno izvršiti preuzimanje liste objekata tipa `Emplyees` za koje je postavljen uslov da je `Country == "USA"`, kod, koji uključuje odgovarajući LINQ upit je:

```
using (NorthwindEntities db = new NorthwindEntities())
{
    var q = from z in db.Employees
            where z.Country == "USA"
            select z;
}
```

Rezultat upita `q` je kolekcija objekata tipa `Employee` koji zadovoljavaju uslov `Country == "USA"`. Ova kolekcija implementira interfejs `IEnumerable`, što omogućava:

- iteraciju kroz rezultate,
- primenu dodatnih LINQ operacija (npr. `ToList()`, `OrderBy()`, `GroupBy()`).

Ukratko, mapiranje u EF -u omogućava:

- transparentnu integraciju baze i aplikacije,
- rad sa objektima umesto sa SQL strukturama,
- lakše testiranje, održavanje i proširivanje aplikacije.

Kroz model, aplikacija dobija **objektni prikaz baze**, čime se pojednostavljuje razvoj i omogućava bolja organizacija koda.

2.2 Radno okruženje

Za razvoj aplikacija zasnovanih na Entity Framework , koristićemo integrisano razvojno okruženje Visual Studio. Ovaj alat omogućava:

- efikasno upravljanje projektima,
- integraciju sa bazama podataka,
- rad sa C# i .Net tehnologijama.

Microsoft nudi besplatnu verziju Visual Studio Community, koja uključuje i SQL Server Express, što je idealno za edukativne i razvojne svrhe. Ovu verziju možete preuzeti sa veb adrese:

https://www.visualstudio.com/downloads/.

Instalacija

Korak 1. Provera sistemskih zahteva.

Pre instalacije, preporučuje se da proverite:

- minimalne hardverske zahteve,
- kompatibilnost operativnog sistema.

Pogledati detalje na linku:

https://learn.microsoft.com/en-us/visualstudio/releases/2022/system-requirements.

Korak 2. Preuzimanje instalacionog paketa.

Preuzmite odgovarajuću verziju Visual Studio-a sa zvaničnog sajta. Za edukativne potrebe, mnoge obrazovne ustanove poseduju licence, pa se informišite o dostupnim opcijama.

Dostupne besplatne verzije:

- Visual Studio Community – za individualne korisnike i obrazovne ustanove,
- Visual Studio Professional – uz licencu,
- Visual Studio Enterprise – za napredne potrebe i timove.

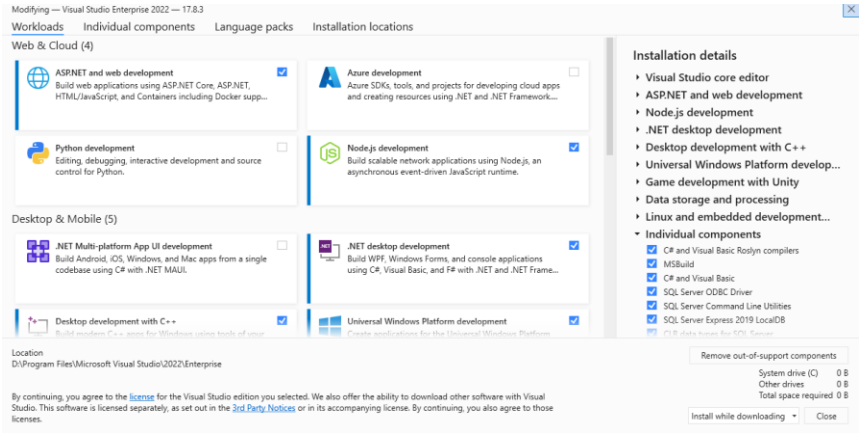
Korak 3. Pokretanje instalacije

U folderu "Downloads" dvostrukim klikom pokrenite odgovarajući instalacioni fajl:

- vs_enterprise.exe za Visual Studio Enterprise
- vs_professional.exe za Visual Studio Professional
- vs_communiti.exe za Visual Studio Community

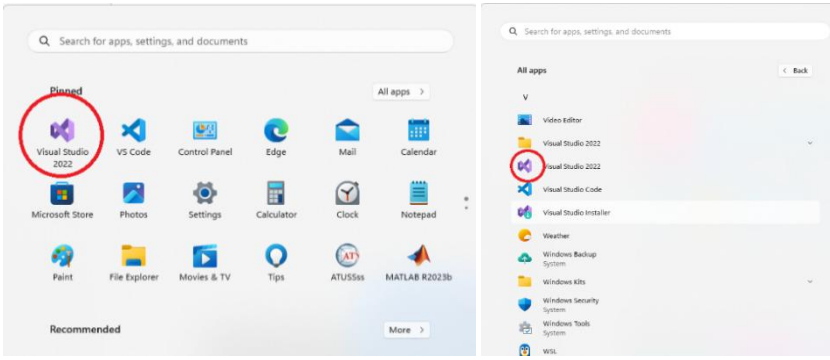
Korak 4. Izbor komponenti za instalaciju

Visual Studio je kompleksno i modularno okruženje. Tokom instalacije, preporučuje se da:
- pažljivo odaberete samo potrebne komponente,
- izbegnete instalaciju nepotrebnih modula kako biste sačuvali resurse sistema.



Slika 2.2. Izbor opcija pri instalaciji razvojnog okruženja Visual Studio

Nakon urađene instalacije pokrenite aplikaciju preko menija ili skraćene komande iz okruženja.



Slika 2.3. Pokretanje Visual Studio IDE okruženja za razvoj aplikacija

2.3 Šabloni projekata

U razvoju softverskih rešenja, projekat predstavlja skup međusobno povezanih datoteka koje zajedno čine funkcionalnu celinu. Njihovim prevođenjem (kompajliranjem) i povezivanjem nastaje aplikacija spremna za izvršavanje. Upravo razvojno okruženje Visual Studio omogućava da se sve izmene u projektu obavljaju sistematski i pregledno.

Datoteke unutar projekta mogu sadržati različite vrste sadržaja:

- Izvorni kod (npr. C#, JavaScript);
- HTML stranice i XML dokumenta;
- Multimedijalni sadržaji (slike, video zapisi);
- Konfiguracione datoteke i resursi.

Zahvaljujući fleksibilnosti Visual Studio-a, dostupni su brojni šabloni za kreiranje projekata, prilagođeni različitim vrstama aplikacija — od veb rešenja do desktop i mobilnih aplikacija.

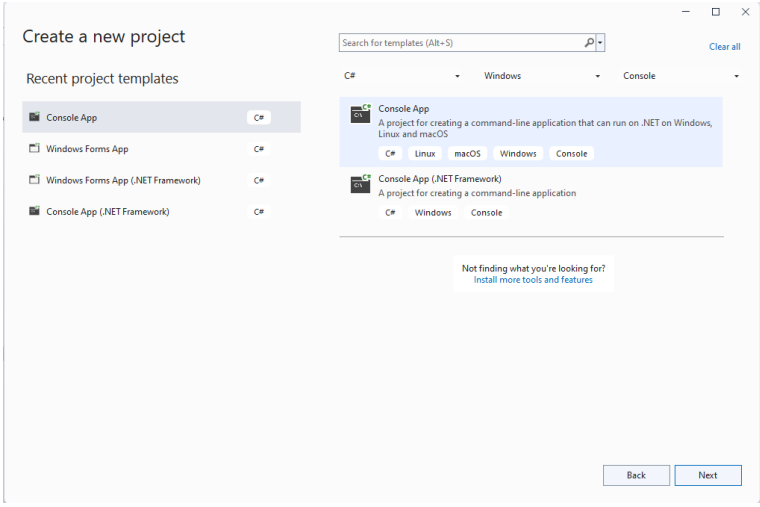
Kreiranje novog projekta:

Korak 1. Pokretanje čarobnjaka za novi projekat

Otvorite Visual Studio i izaberite: File -> New -> Project

Korak 2. Inicijalizacija projekta

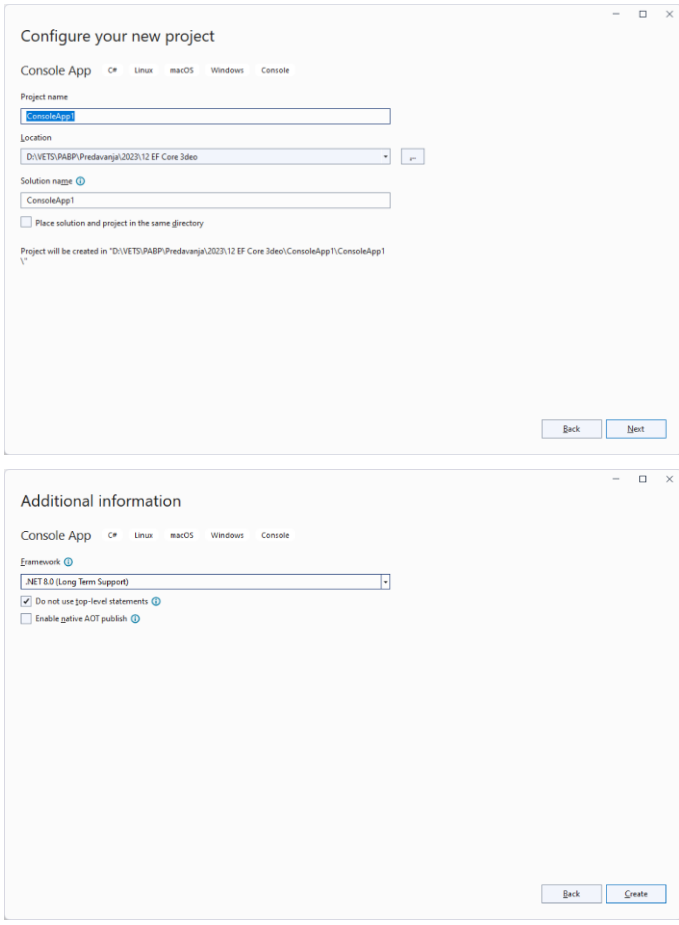
Nakon završetka Koraka 1. otvara se prozor za inicijalizaciju projekta tokom koje se definišu biblioteke i osnovni dizajn sa kojim se kreće u razvoj. Sve što se u ovoj fazi definiše moguće je naknadno promeniti, ali je svakako jednostavnije napraviti pravilan izbor, pogotovo za početak.



Slika 2.4. Izbor šablona novog projekta

Iako se u ovom poglavlju fokusiramo na veb aplikacije, za prvi susret sa Visual Studio okruženjem koristićemo najjednostavniji tip — konzolnu aplikaciju. Konzolne aplikacije su idealne za:

- Testiranje osnovne logike;
- Administrativne zadatke;
- Učenje sintakse programskog jezika.



Slika 2.5. Podaci konzolne aplikacije

Nakon odabira šablona Console App, potrebno je popuniti sledeća polja:

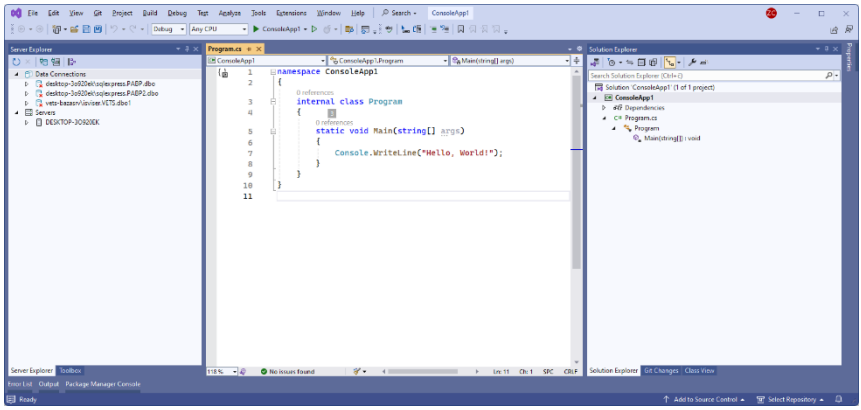
- Location – putanja na disku gde će projekat biti smešten;
- Project Name – naziv projekta;
- Solution Name – automatski se preuzima naziv projekta, ali može se izmeniti.

Rešenje (Solution) može sadržati jedan ili više projekata. U ovom koraku birate da li kreirate novo rešenje ili dodajete projekat postojećem.

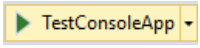
U polju Solution se bira opcija za kreiranje novog projekta ili se projekat dodaje nekom od drugih Solution-a.

Korak 3. Pokretanje aplikacije

Nakon što se aplikacija kreira, spremna je za dalje izmene i testiranje.



Slika 2.6. VS nakon kreiranja aplikacije

Pritiskom na taster **F5** ili na kombinaciju tastera **Ctrl + F5** ili preko menija na ikonu 

pokreće se prevođenje fajlova, njihovo povezivanje u aplikaciju i izvršavanje.

Pažnja. F5 odnosno Ctrl+F5 predstavljaju dva načina pokretanja aplikacije: sa debagovanjem ili bez debagovanja. Debagovanje je termin koji je već duže vreme odomaćen u našem jeziku, a koristi se da opiše za izvršavanje sa mogućnošću zaustavljanja i provere svih parametara programa tokom izvršavanja, a koristi se pri otkrivanju grešaka i učenju.

2.4 Integrisani jezik za upite *LINQ*

Cilj ovog poglavlja je da studentima pruži temeljno razumevanje jezika za upite zasnovanog na kolekcijama podataka — LINQ (eng. *Language-Integrated Query*). Stečena znanja omogućiće kreiranje efikasnih upita u različitim kontekstima:

- Rad sa bazama podataka;
- Manipulacija kolekcijama i listama;
- Upiti nad objektima tipa `DataSet`;
- Integracija sa ORM okruženjem kao što je Entity Framework.

Poglavlje obuhvata ključne koncepte kao što su **lambda izrazi**, **anonimni tipovi**, kao i standardni LINQ **operatori**: filtriranje, sortiranje, grupisanje i agregacija.

Uvod

LINQ predstavlja skup tehnologija koje omogućavaju integraciju upitnih izraza direktno u C# jezik. Za razliku od tradicionalnih upita koji se pišu kao stringovi (npr. SQL), LINQ upiti su deo sintakse jezika i mogu se proveriti tokom kompajliranja, uz podršku IntelliSense-a u Visual Studio okruženju.

Umesto da programer mora da poznaje različite upitne jezike (SQL, NoSQL, XPath, itd.), LINQ omogućava jedinstven pristup podacima kroz C# sintaksu — koristeći klase, metode i događaje.

LINQ je posebno koristan u ASP.NET MVC (eng. Model–View–Controller) aplikacijama, gde se koristi za efikasno i dosledno upravljanje podacima putem Entity Framework-a.

Osnovna sintaksa upita

Pogledajmo jednostavan primer LINQ izraza:

```
var query = from e in employees
```

```
            where e.id == 1
            select e.name;
```

Ova sintaksa podseća na SQL, ali je potpuno validna C# sintaksa. Ključne reči: `from`, `where`, `select` deo su LINQ jezika, dok se `var`, `in`, == oslanjaju na standardni C#.

LINQ podržava napredne operacije kao što su:

- Agregatne funkcije (Sum, Average, Count);
- Združivanje (join);
- Grupisanje (group by);
- Sortiranje (orderby);
- Projekcija i transformacija podataka.

Funkcionalnost

LINQ poseduje jedinstven model koji se može koristiti u različitim kontekstima:

- LINQ ***to Objects*** je aplikacioni interfejs koji prikazuje standardne operatore upita) za dobijanje podataka iz nekog objekta čija klasa implementira interfejs **IEnumerable<T>.** Ovi upiti se izvršavaju na podacima u memoriji.
- LINQ ***to ADO.NET*** (eng. ActiveX Data Objects for .NET – Nova Microsoft tehnologija sa pristup podacima) proširuje standardne operatore na rad sa relacionim podacima. Sastoji se iz 3 dela:
  - LINQ **to SQL** se koristi za upite ka relacionim bazama kao MSSQL.
  - LINQ **to DataSet** se koristi za upite koji koriste ADO.NET skupove podataka.
  - LINQ **to Entities** je Microsoft ORM (eng. Object-Relational Mapping) rešenje za upotrebu Entities objekata.
  - LINQ ***to XML***

Primer primene *LINQ to Objects*

Ovo je najopštiji prikaz rada LINQ jezika. U ovom slučaju koristimo LINQ za rad sa objektima, tačnije nad kolekcijama objekata. Za ove primere možete koristiti postojeću bilo koju postojeću aplikaciju ili možete kreirati novu, na primer konzolnog tipa.

Najpre, vašem projektu dodajte novu klasu `Person`:

```
class Person
{
    public int ID;
    public int IDRole;
    public string LastName;
    public string FirstName;
}
```

Zatim napravite listu objekata `Person`. Inicijalizacija objekata u listi se može uraditi tokom kreiranja liste:

```
List<Person> people = new List<Person>
{
    new Person()
    {
        ID = 1, IDRole = 1,
        LastName = "Petar", FirstName = "Ilić"
    },
    new Person() {
        ID = 2, IDRole = 2,
        LastName = "Jovan", FirstName = "Jović"
    },
    new Person() {
        ID = 3, IDRole = 1,
        LastName = "Ana", FirstName = "Milić"
    }
};
```

Zatim napišite prvi konkretan upit. Na primer:

```
var query = from p in people
            where p.ID == 1
            select p;
```

Na prvi pogled uočavaju se ključne reči iz C#: `var`, `in`, kao i operatori poređenja i dodele vrednosti. Osim toga, postoje i ključne reči za LINQ upite: `from`, `where`, `select`.

Logično je da ovakav upit daje rezultat koji predstavlja neku C# kolekciju objekata. Ove kolekcije koje LINQ koristi su uvek kolekcije tipa (tačnije interfejsa) `IEnumerable`. Ako su objekti koji se selektuju unapred definisane klase, kao u ovom primeru, onda je rezultat generički tip `IEnumerable<Tip>`. Dakle, u ovom slučaju, rezultat je kolekcija objekata koji su tipa `Person`.

Bez obzira što u izrazu koristimo tip `var` za rezultat upita, ovaj tip ne ograničava prihvatanje stvarnog tipa od upita. Postavlja se pitanje kako koristiti ove podatke? Postoji više načina, a ovde ću pokazati dva:

- Koristeći osobine kolelekcije. `IEnumerable` interfejs omogućava prolaz kroz sve elemente kolekcije, pa bi mogući prihvat rezultata mogao da bude:

```
List<Person> lst = new List<Person>();
if (query != null){
    foreach (object item in query)    {
        lst.Add((Person)item);
    }
}
```

- Direktnom konverzijom u željenu kolekciju, na primer u listu:
- `lst = query.ToList<Person>();`

Metode proširivanja

LINQ sintaksa koristi ključne reči, na primer: `where` i `select`, koji se transformišu u C# metode kao što se ceo LINQ izraz transformiše u sekvencu metoda koje se pozivaju na nekoj kolekciji objekata. Ove ključne reči se transformišu u dve metode: `Where<T>()` i `Select<T>().`

Važna osobina ovih metoda je da se se nadovezuju, tj. da se rezultati `Where` metode mogu dalje izdvajati novim pozivom `Where` metode ili pomoću `Select` metode. Tip podataka ostaje i dalje `IEnumerable<T>`. Ove metode su vezane za tip podataka na koji se odnose, odnosno vrše proširenje funkcionalnosti

postojećih klasa, primenom metoda proširivanja (eng. *extension methods*). Takve metode su još: `Join`, `OrderBy`, `GroupBy`, …

Prema tome, standardni LINQ izraz:

```
var query = from p in people
            where p.ID == 1
            select p;
```

može da se napiše drugačije, primenom metoda proširenja.

```
var queryCopy = people.Where(x => x.ID == 1).Select(x=>x);
```

Argumenti ovih metoda proširenja, na primer: `x => x.ID == 1`, predstavljaju specifične izraze tzv. **Lambda izraze**. U standardnom LINQ izrazu koristi se skraćena verzija Lambda izraza, `p.ID == 1`.

Lambda izrazi

Metoda `Where` se primenjuje na kolekciji objekata. Metoda vrši odabir određenih objekata na osnovu neke proizvoljne funkcije koja treba da se prosledi ovoj metodi. Ta funkcija obavlja filtriranje tako što vraća `true` za objekte koji zadovoljavaju uslov filtriranja, odnosno `false` za one koji to ne zadovoljavaju.

Funkcija, tj. druga metoda, koja se prosleđuje kao argument nekoj metodi mora biti tipa `delegate`. Dakle, ono što `Where` metoda prima je zapravo objekat tipa `delegate`.

Druga osobina primene funkcija za filtriranje koje se prosleđuju metodi `Where` jeste da su to obično kratke metode i koje se koriste samo na tom mestu. Zbog toga se ta funkcija piše neposredno kada se i koristi na način kako se pišu funkcije bez imena koje se odmah i koriste. Ovo su tzv. **anonimne** funkcije.

Dakle, `x => x.ID == 1`, predstavlja lambda izraz koji opisuje generisanje delegata za funkciju filtriranja. Izraz vraća tip `bool` u zavisnosti da li je argument `x.ID == 1`. Ova funkcija se izvršava na celoj kolekciji objekata na koju se metoda `Where` odnosi.

Anonimni tipovi

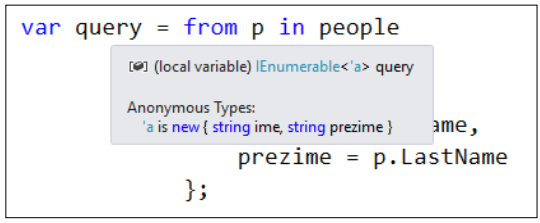
Anonimni tipovi, slično anonimnim funkcijama, označavaju tipove objekata koji nisu prethodno imenovani. Na primer:

```
var v = new { ime = "Jovan", prezime = "Jović" };
```

Dakle, promenljiva **v** je tipa koji nije definisan pre same upotrebe. U LINQ izrazima anonimni tipovi dozvoljavaju da se radi sa rezultatima upita bez eksplicitne definicije klase koja ih predstavlja.

```
var query = from p in people
            where p.ID == 1
            select new {
                ime = p.FirstName,
                prezime = p.LastName
            };
```

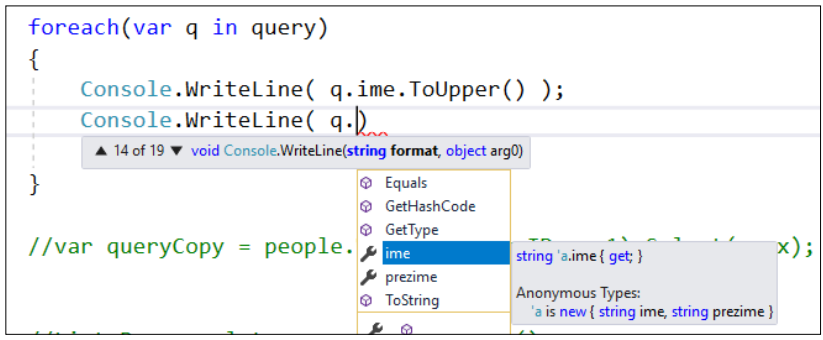
Pogledajmo koji tip pokazuje IDE tokom formiranja LINQ izraza:



Slika 2.7. Prikaz anonimnih tipovima u LINQ izrazima kroz VS

Nakon kreiranja LINQ izraza, rad pomoću *IntellySense* alatke omogućava laku upotrebu sa prepoznavanjem osobina rezultujućeg tipa u izrazu.

Pogledajmo primer kada je rezultat LINQ poznati tip iako je promenljiva deklarisana kao var.



Slika 2.8. Rad sa tipiziranim objektom u VS

Operatori

Radi lakšeg učenja, LINQ operatori su grupisani prema svojoj nameni i ilustrovani primerima. Grupisanje omogućava sistematično razumevanje i lakše povezivanje sa realnim scenarijima.

Filtriranje

Operatori za filtriranje omogućavaju izdvajanje elemenata iz kolekcije na osnovu zadatih kriterijuma:

- `Where`
  Filtrira elemente kolekcije na osnovu prosleđene funkcije (delegata) koja vraća `true` ili `false`. Samo oni elementi za koje funkcija vrati `true` bivaju uključeni u rezultat.
- `OfType`
  Filtrira elemente kolekcije po tipu. Koristan je kada radimo sa negeneričkim kolekcijama kao što je `ArrayList`, ili kada želimo da izdvojimo samo objekte određenog tipa iz heterogene kolekcije.

Primer:

```
System.Collections.ArrayList list = new
System.Collections.ArrayList();

list.Add("Petar");
list.Add("Jova");
list.Add(new object());
list.Add(32.22);
```

```
list.Add(new object());

var query = from x
in list.OfType<string>()
where x == "Jova"
select x;
```

Napomena: Pošto `ArrayList` ne implementira `IEnumerable<T>`, već samo `IEnumerable`, `OfType()` je jedini LINQ operator koji se može direktno primeniti. Ovaj operator je naročito koristan kada radimo sa klasama koje nasleđuju zajednički interfejs ili baznu klasu, a želimo da izdvojimo samo instance određenog tipa.

Sortiranje

Operatori za sortiranje omogućavaju raspoređivanje elemenata kolekcije po zadatim kriterijumima:

- `OrderBy`, `OrderByDescending`
  Sortiraju kolekciju po rastućem, odnosno opadajućem redosledu na osnovu zadate funkcije.
- `ThenBy`, `ThenByDescending`
  Omogućavaju dodatno sortiranje već sortirane kolekcije po sekundarnom kriterijumu.
- `Reverse`
  Sortiranje u obrnutom redosledu.

Primer:

```
var q = from p in people
orderby p.LastName, p.FirstName
select p;
```

Povratna vrednost operatora `OrderBy` je `IOrderedEnumerable<T>`, koji nasleđuje `IEnumerable<T>` i omogućava primenu dodatnih operatora kao što su `ThenBy` i `ThenByDescending`.

Skupovni operatori

Skupovni operatori omogućavaju rad sa kolekcijama na način koji podseća na skupovnu algebru:

- `Distinct`
  Uklanja duplikate iz kolekcije.
- `Except`
  Vraća elemente koji se nalaze u prvoj kolekciji, ali ne i u drugoj.
- `Intersect`
  Vraća zajedničke elemente iz dve kolekcije.
- `Union`
  Vraća sve jedinstvene elemente iz obe kolekcije.

Primer:

```
int[] niz2 = { 2, 4, 6, 8, 10 };
int[] niz3 = { 3, 6, 9, 12, 15 };
//6
var intersection = niz2.Intersect(niz3);
// 2, 4, 8, 10
var except = niz2.Except(niz3);
// 2, 4, 6, 8, 10, 3, 9, 12, 15
var union = niz2.Union(niz3);
```

Kvantifikatori

Kvantifikatori su LINQ operatori koji vraćaju logičku vrednost (true ili false) na osnovu zadatog uslova:

- `All`
  Vraća `true` samo ako svi elementi kolekcije zadovoljavaju navedeni uslov.
- `Any`
  Vraća `true` samo ako bar jedan element kolekcije zadovoljava navedeni uslov.
- `Contains`
  Vraća `true` ako kolekcija sadrži navedeni element.

Primeri:

```
int[] niz2 = { 2, 4, 6, 8, 10 };
// true
bool areAllevenNumbers = niz2.All(i => i % 2 == 0);
// true
bool containsMultipleOfThree = niz2.Any(i => i % 3 == 0);
```

```
// false
bool hasSeven = niz2.Contains(7);
```

Projekcije

Projekcioni operatori transformišu ulazne podatke u izlazne, najčešće menjajući oblik rezultata:

- `Select`
  Vraća jedan rezultat za svaki ulazni element. Može se koristiti za formiranje novog tipa podataka.
- `SelectMany`
  Koristi se kada radimo sa kolekcijom kolekcija (sekvencama sekvenci). Kombinuje sve unutrašnje kolekcije u jednu spoljnu kolekciju.
  SelectMany se koristi kada postoji višestruki **from.**

Primeri

```
string[] tekst = new string[]{
"Ovo je samo primer",
"Ovo je druga recenica",
"Treci primer" };

var query = (
  from recenice in tekst
  from reci in recenice.Split(' ')
  select reci
).Distinct();
```

U ovom primeru:

- `tekst` je niz stringova (rečenica).
- Prvi `from` izdvaja rečenice.
- Drugi `from` koristi `Split(' ')` da razdvoji rečenice na reči.
- `Distinct()` uklanja duplikate.

Rezultat: Kolekcija jedinstvenih reči:

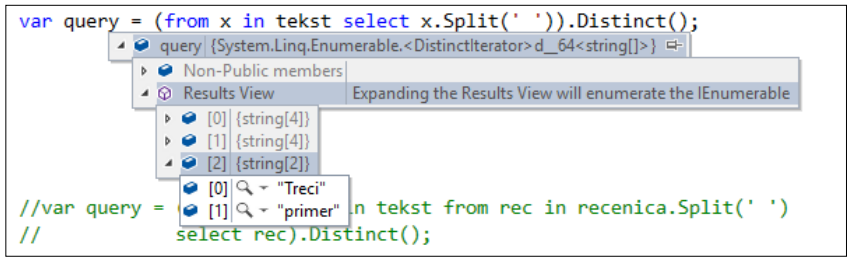
`Ovo, je, samo, primer, druga, recenica, Treci.`

Ako se koristi samo `Select`, bez ugnježdenog `from`, rezultat je kolekcija kolekcija:

```
var query = tekst.Select(x=> x.Split(' ')).Distinct();
```

```
// identicno sa
// var query = (from x in tekst select x.Split(' ')).Distinct();
```

dobija se rezultat koji je kolekcija od tri kolekcije. Svaka od tri kolekcije sadrži stringove tj. reči unutar rečenice na koju se odnosi, pogledajte sliku, a naredba `Distinct` je primenjena nad kolekcijama, ne na rečima.

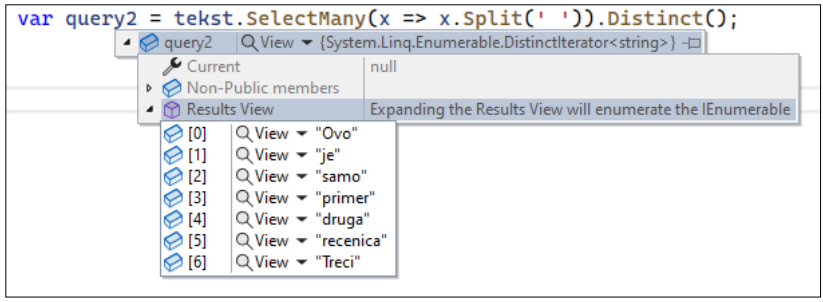


Slika 2.9. Prikaz rezultata LINQ izraza u IDE okruženju

Drugi operator projekcije, `SelectMany`, sve rezultate vraća u jednu kolekciju. Na ovaj način upit:

```
var query = tekst.SelectMany(x=>x.Split(' ')).Distinct();
```

vraća identičan rezultat kao više ugnježdenih upita.



Slika 2.10. Višestruki select u *debug* režimu

Izdvajanje rezultata

Ovi operatori omogućavaju selektivno izdvajanje delova kolekcije, što je korisno za straničenje i prikaz delimičnih rezultata:

- `Skip`

Preskače određeni broj elemenata.

- `Take`
  Uzimanje određenog broja elemenata.
- `SkipWhile, TakeWhile`
  Preskakanje/uzimanje elemenata dok je uslov zadovoljen.

U slučaju veb straničenja, da bi dobili treću stranicu rezultata, a uzima se 10 zapisa po stranici, može se koristiti `Skip(20)`, a zatim `Take(10).`

Takođe, postoje i operatori `SkipUntil`, `TakeUntil`.

Primer:

```
int[] numbers = { 1, 3, 5, 7, 9 };
var query =
 numbers
.SkipWhile(n => n < 5)
.TakeWhile(n => n < 10); // 5, 7, 9
```

Združivanja

Kao i u standardnim SQL upitima, nekada je važno uključiti više tabela u upit, naravno po osnovu određenih kriterijuma. Slično važi i za LINQ upite. Osnovni operatori združivanja su:

- `Join`
  Sličan SQL komandama INNER JOIN.
- `GroupJoin`
  Sličan SQL komandama LEFT OUTER JOIN.

Na primer, dodajmo novu klasu `Roles`, kao u kodu:

```
class Role
{
    public int ID;
    public string name;
}
```

Zatim uradimo inicijalizaciju podataka:

```
List<Role> roles = new List<Role> {
    new Role{
        ID=1,
        name = "Student"
```

```
    },
    new Role{
        ID=2,
        name = "Teacher"
    }
};
```

Sada možemo uraditi prvo povezivanje i testiranje rezultata primenom `Join` operatora.

```
var query = from p in people
            join r in roles
            on p.IDRole
            equals r.ID
            select new
            {
                osoba = p.LastName + " " + p.FirstName,
                uloga = r.name
            };
```

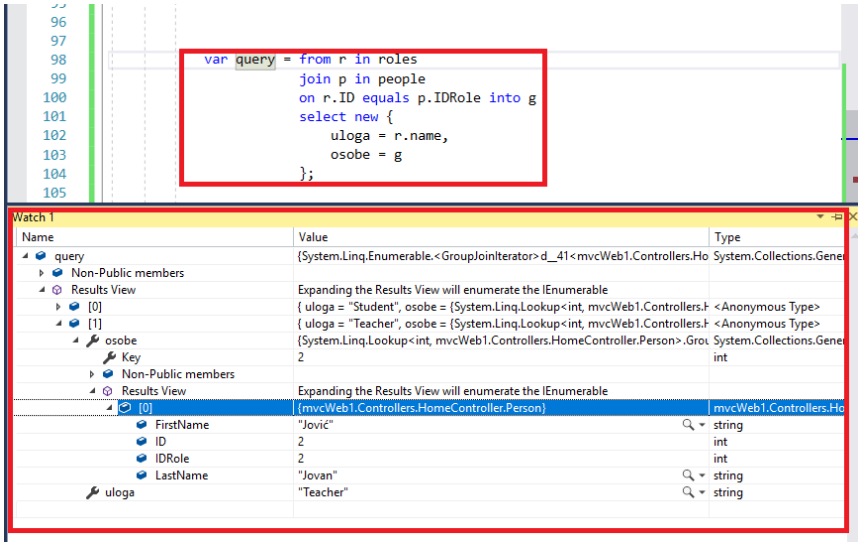
Kod je potpuno analogan unutrašnjem SQL združivanju. U našem slučaju dobija se lista osoba sa ulogom koja je preuzeta iz druge tabele.

Višestruko povezivanje

Osim operatora `join` grupisanje grupisanje se može uraditi i operatorom `group join`. Pogledajmo početni primer:

```
var query = from r in roles
            join p in people
            on r.ID equals p.IDRole into g
            select new {
                uloga = r.name,
                osobe = g
            };
```

Upit `join` daje **parove** elemenata za svako podudaranje: `(r1,p1), (r1,p2), (r2,p2), (r2,p3), (r2,p4)`. U slučaju upita `group join` dobija se samo jedan rezultujući objekat za svaki element prve kolekcije, `roles`, a odgovarajući elementi druge kolekcije, u ovom primeru `people`, grupisani su u jednu kolekciju:`(r1,[p1,p2]), (r2,[p2,p3,p4])`. Kao rezultat upita dobija se anonimni tip za svako podudaranje, a sastoji se od jedne uloge i kolekcije osoba.



Slika 2.11. Prikaz rezultata LINQ izraza preko *Watch* prozora u toku izvršavanja

Grupisanje

- `GroupBy`
  Standardno grupisanje po zadatoj funkciji. Izvršavanje se odlaže dok se rezultati ne zatraže (eng. lazy evaluation).
- `ToLookup`
  Grupisanje koje se izvršava odmah (eng. eager evaluation).

Oba operatora vraćaju sekvencu tipa `IGrouping<TKey, TElement>`, gde interfejs `IGrouping` izlaže svojstvo `Key` koje identifikuje grupu.

Primer:

```
int[] niz = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
var query = niz.ToLookup(i => i % 2);
foreach (IGrouping<int, int> group in query)
{
   Console.WriteLine("Key: {0}", group.Key);
   foreach (int element in group) { Console.WriteLine(element); }
}
```

Vraćaju se dve grupe sa **`Key`** vrednostima **`0, 1`**. U svakoj grupi je lista objekata originalnog niza koji joj pripadaju.

Osnovna razlika između `GroupBy` i `ToLookup` operatora je što operator `GroupBy` obavlja izvršavanje sa kašnjenjem. `ToLookup` obavlja izvršavanje odmah.

Opšti operatori

- Empty
  Kreira praznu sekvencu `IEnumerable<T>.` Na primer:
  ```
  var empty = Enumerable.Empty<Employee>();
  ```
- Range
  Operator koji generiše sekvencu brojeva. Na primer:
  ```
  IEnumerable<int> numbers = Enumerable.Range(1, 10);
  ```
- Repeat
  Generiše sekvencu bilo kojih vrednosti. `var petP = Enumerable.Repeat(new Employee { Name = "Perica" }, 10);`
- DefaultIfEmpty
  Generiše praznu kolekciju sa podrazumevanom vrednošću koja pripada tipu kada se primeni.

Operator jednakosti sekvence

- SequenceEquals
  Poredi dve sekvence po redosledu i sadržaju.

  Primer:

```
Person e1 = new Person() { ID = 1 };
Person e2 = new Person() { ID = 2 };
Person e3 = new Person() { ID = 3 };
var employees1 = new List<Person>() { e1, e2, e3 };
var employees2 = new List<Person>() { e3, e2, e1 };
//false
bool result = employees1.SequenceEqual(employees2);
```

Element operatori

- ElementAt
  Izdvaja se određeni element iz sekvence.
- First
  Prvi element iz sekvence. Ako ne postoji, emituje se izuzetak.
- Last
  Poslednji element iz sekvence. Ako ne postoji, emituje se izuzetak.

- Single
  Jedan element iz sekvence. Ako postoji više elemenata ili ne postoji ni jedan, emituje se izuzetak.

Za svaki operator postoji odgovarajući `OrDefault` operator koji se može koristiti da se izbegne izuzetak kada element ne postoji: `ElementAtOrDefault, FirstOrDefault, LastOrDefault, SingleOrDefault`.

Primer:

```
string[] empty = { };
string[] notEmpty = { "Zdravo", "Programeri" };
var result = empty.FirstOrDefault(); // null
result = notEmpty.Last(); // Programeri
result = notEmpty.ElementAt(1); // Programeri
result = empty.First(); // InvalidOperationException
result = notEmpty.Single(); // InvalidOperationException
result = notEmpty.First(s => s.StartsWith("Z"));
```

Osnovna razlika između operacija `First` i `Single` je što `Single` operator šalje izuzetak ako sekvenca ne sadrži jedan element, dok `First` vraća rezultat koji je prvi element. `First` šalje izuzetak samo ako ne postoji ni jedan element.

Konverzije

- OfType
  Filtrira elemente po tipu. Vraća samo one koje može bezbedno kastovati.
- Cast
  Pokušava da kastuje sve elemente u zadati tip. Baca izuzetak ako neki element ne može biti kastovan.

`OfType` operator je i operator filtriranja – vraća samo objekte koje može kastovati tj. pretvoriti u neki tip, dok će operator `Cast` vršiti konverziju i pri tome će slati izuzetak ako ne može da kastuje sve objekte u neki tip.

```
object[] podatak = { "Pera", 3, "Aca" };
// kreira sekvencu od 2 stringa
var query1 = podatak.OfType<string>();
// izaziva izuzetak
var query2 = podatak.Cast<string>();
```

Spajanje

- `Concat`
  Operator koji spaja dve sekvence bez uklanjanja duplikata.
- `Union`
  Spaja dve sekvence i uklanja duplikate.

Primer:

```
string[] ime = { "Joca", "Joca" };
string[] prezime = { "Ilic", "Mijic"};

//"Ilic", "Joca", "Joca", "Mijic"
var nadovezivanje = ime.Concat(prezime).OrderBy(s => s);
//"Ilic", "Joca", "Mijic"
var unija = ime.Union(prezime).OrderBy(s => s);
```

Agregacija

Agregatni operatori vraćaju jedan rezultat (skalar) iz kolekcije:

- `Average`, `Count`, `LongCount` (za velike rezultate), `Max`, `Min`, `Sum`.
  Agregatni operatori, slični agregatnim SQL funkcijama. Daju jedan rezultat – skalar, iz kolekcije na koju se primenjuju.

Agregatnim operacijama dobijaju se statistički podaci za kolekcije.

Primer:

```
int[] niz = { 2, 4, 6, 8, 10 };
var summary = new
{
    ProcessCount = niz.Count(),
    TotalThreads = niz.Sum(),
    MinThreads = niz.Min(),
    MaxThreads = niz.Max(),
    AvgThreads = niz.Average()
};
```

2.5 Razvoj EF projekta

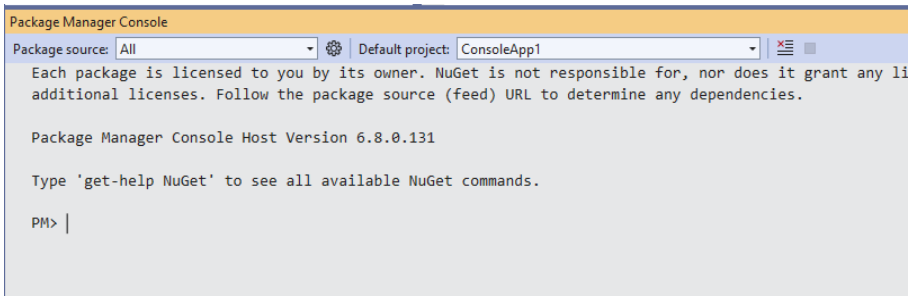
Nakon upoznavanja sa kreiranjem .Net projekata, a zatim i sa LINQ bibliotekom, možemo preći na razvoj EF projekta. Prvi projekat je zasnovan na šablonu konzolne aplikacije, koji je već kreiran u okviru poglavlja „Šabloni projekta“.

Da bi projekat imao podršku za rad sa bazama podataka odnosno ORM (eng. Object-Relational Mapping), potrebno je uključiti dodatne biblioteke. Moderna razvojna okruženja omogućavaju jednostavno dodavanje paketa putem alata. Za ovu svrhu koristićemo alata koji se naziva **NuGet** – menadžer paketa za .NET projekte.

Dodavanje EF paketa

Rad sa NuGet paketima se može ostvariti na dva načina.

1. Korišćenjem prozora **Package Manager Console**.



Slika 2.12. Package Manager Console

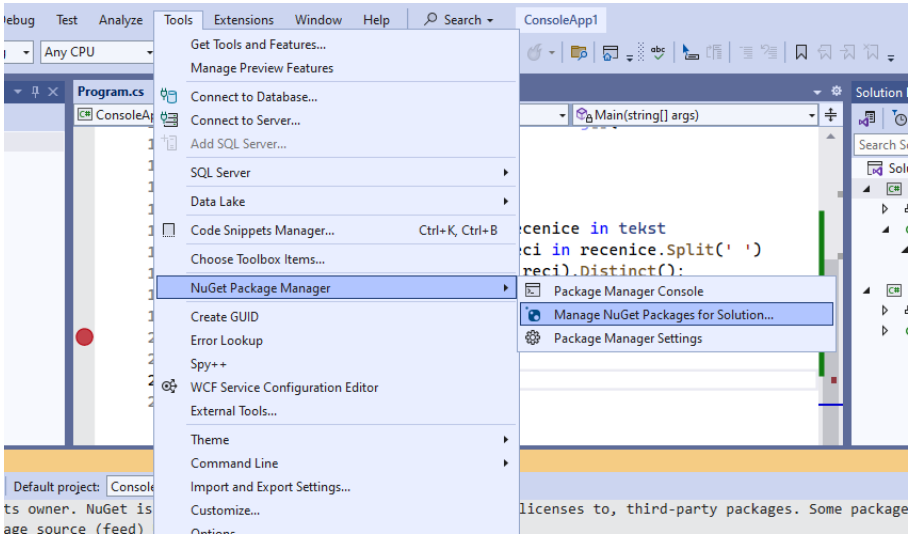
Ovaj prozor je podrazumevano smešten na dnu IDE okruženja. Oznaka **PM>** označava početak komandne linije gde se mogu uneti direktno komande za rad sa paketima. Unosom komandi može se dodati ili oduzeti neki paket, generalno može se uraditi modifikacija projekta.

```
PM>Install-Package Microsoft.EntityFrameworkCore.SqlServer
```

2. Korišćenjem grafičkog interfejsa

Ovo se postiže komandom iz menija: **`Tools->NuGet Package Manager>Manage NuGetPackages for Solutions...`**

Moguće je pretražiti raspoložive pakete po različitim kriterijumima: naziv, ključne reči i slično, a zatim iz liste odabrati željeni paket.



Slika 2.13. Otvaranje prozora za pretragu paketa

Za rad sa podacima koji su u bazi MS SQL Servera, potrebno je dodati sledeće pakete:

- `Microsoft.EntityFrameworkCore`

Paket za bazični rad sa EF bibliotekom.

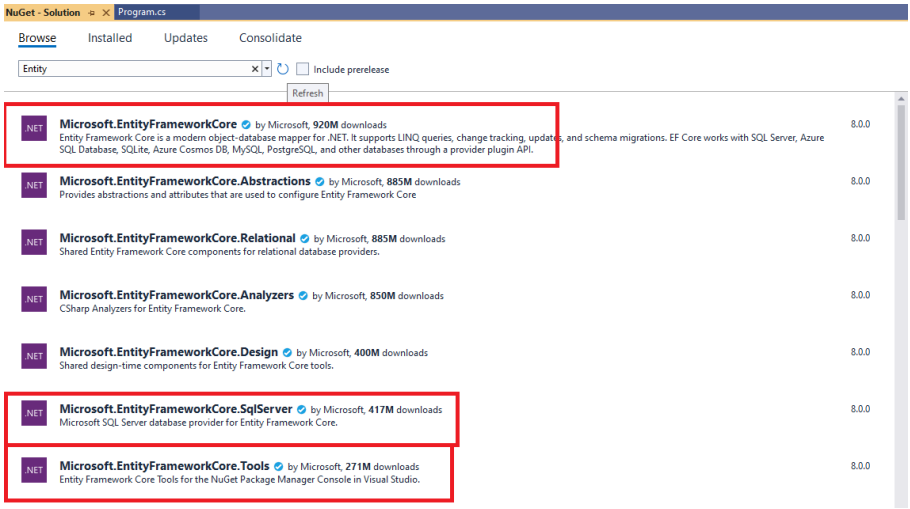
- `Microsoft.EntityFrameworkCore.SqlServer`

EF biblioteka za podršku u radu sa MS SQL Server bazom podataka.

- `Microsoft.EntityFrameworkCore.Tools`

EF biblioteka za ORM podršku. Omogućava migracije i scaffolding modela.

Ovi paketi su prikazani na slici u vidu liste koja je rezultat pretrage.



Slika 2.14. Izbor paketa za uključivanje u EF projekat

Modelovanje entiteta

Nakon dodavanja paketa, projekat možemo proširiti prvim modelima. Na početku rada sa EF bibliotekom, naveli smo da ćemo koristiti mapiranje relacionih baza na odgovarajuće objektne koncepte. Pri tome smo naveli da se tabela mapira na klasu.

Kao prvi primer, jako jednostavan, pretpostavićemo postojanje dve klase `Knjiga` odnosno `Komentar`. Klase koje se koriste za mapiranje tabela nazivaju se POCO klase – *Plain Old CLR Objects*, odnosno *entity klase* jer ih EF mapira na tabele u bazi. Svojstva klase odgovaraju kolonama tabele. Postoje i druga svojstva klase koja ukazuju na druge osobine, zato ćemo pogledati ceo model pa dati objašnjenje.

Klase postavljamo u posebno kreirani folder `Models` i definišemo ih na način kako je to prikazano u nastavku.

```
internal class Knjiga
{
    public int KnjigaId {  get; set; }
```

```
    public string Naslov { get; set; }
    public DateTime DatumIzdanja { get; set; }
}
internal class Komentar
{
    public int KomentarId { get; set; }
    public string Tekst { get; set; } = null!;
    public int? Ocena {  get; set; }
}
```

Konfiguracija DbContext klase

Da bi klase bile mapirane na odgovarajuće tabele, aplikacija se mora proširiti uvođenjem `DbContext` klase. Zapravo, u aplikaciji se kreira klasa koja nasleđuje EF `DbContext` klasu. Na primer:

```
internal class PabpContext : DbContext
{
    public DbSet<Knjiga> Knjigas { get; set; }
    public DbSet<Komentar> Komentars { get; set;}

    protected override void
    OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    => optionsBuilder.UseSqlServer(
        "Data Source=localhost\\sqlexpress;Initial
         Catalog=PABP;Integrated Security=True;Encrypt=False"
       );
}
```

Ova klasa je važan deo EF aplikacija jer čuva konfiguraciju konekcije i definiciju kolekcija entiteta.

Zapazite da se klase `Knjiga` i `Komentar` koriste za kreiranje kolekcija `Knjigas` odnosno `Komentars` koristeći EF generičku klasu `DbSet`. DbSet<T> definiše kolekciju entiteta koja se mapira na tabelu u bazi.

Konekcija sa bazom

Kako bi mogli aplikaciju povezati sa nekim serverom baze podataka, potrebno je imati pristupne parametre do baze. Pristupni parametri uključuju podatke o serveru:

- ip adresa (nekada je umesto ip adrese moguće koristiti ime računara na mreži),

- ime servera (pošto je moguće da postoji više servera na istom računaru),
- korisničko ime,
- lozinka.

Virtuelna metoda `OnConfiguring` koristi se za konfigurisanje konekcije do odgovarajućeg servera odnosno baze. U ovom slučaju radi se o MS Sql Serveru baza podataka, odnosno o konkretnoj bazi `PABP`, pa se koristi funkcija `UseSqlServer`. Kao argument ove funkcije koristi se string vrednost koja se naziva konekcijski string.

Ovaj konekcijiski string se može uneti samostalno ili se za njenu vrednost može iskoristiti pomoćni alat, Server Explorer, koji se nalazi u okviru Visual Studio okruženja.

Nakon podešavanja parametara konekcije, konekciju treba obavezno proveriti koristeći dugme „Test Connecton“. Ako ovaj test prođe uspešno, onda je i konekcija dobra, pa se nastavak zasniva na ispravnoj vezi do servera podataka.

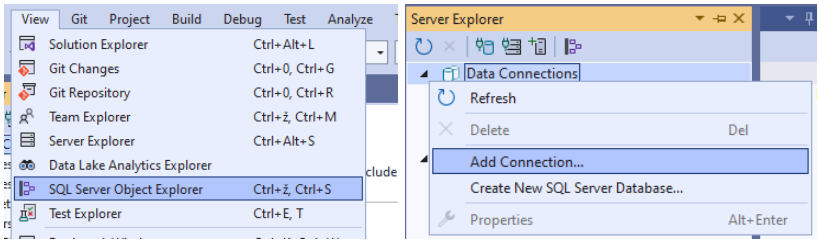
Server Explorer predstavlja alat u okviru VS koji nam daje mogućnost lakog i intuitivnog pristupa do baze kao i uvid u podatke mimo naše aplikacije koju razvijamo, pa ćemo videti kako sa njim možete početi sa radom.

Alat *Server Explorer*

Server Explorer je alat u okviru Visual Studio okruženja koji omogućava vizuelni pristup bazama podataka. Kroz njega se može:

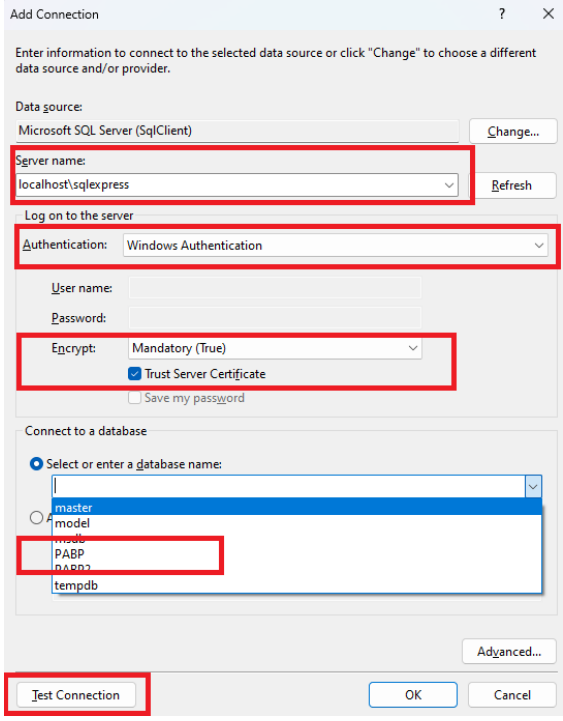
- Dodavati nove konekcije ka bazama;
- Pregledati strukturu tabela;
- Pristupiti podacima u tabelama;
- Analizirati relacije i veze između entiteta.

Na narednoj slici su prikazani koraci za prikaz prozora Server Explorer koji ćemo koristiti , između ostalog, za dodavanje i određivanje konekcijskog stringa do baze.

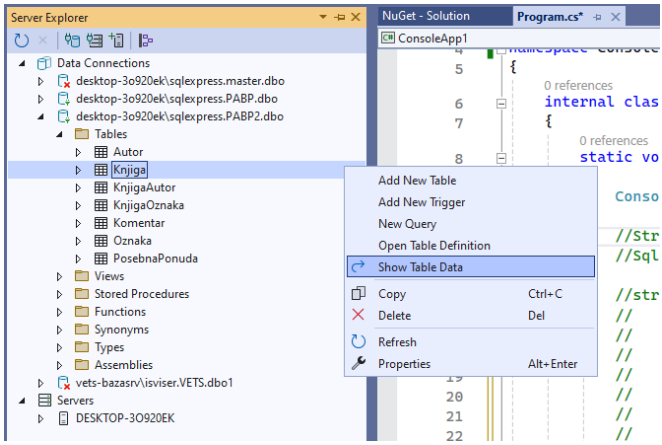


Slika 2.15. Otvaranje *Server Explorer* prozora početak dodavanja nove konekcije

Zatim se vrši podešavanje konekcije kao na sledećoj slici, naravno ako već imate kreiranu bazu na serveru koju ćete koristiti za dodavanje novih tabela.



Slika 2.16. Podešavanje konekcije

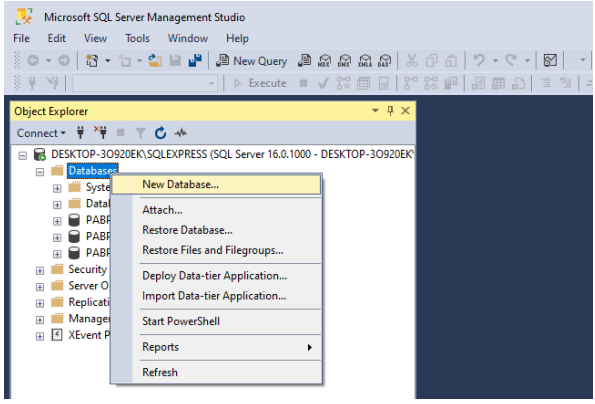


Slika 2.17. Pogled na Server Explorer i neke njegove funkcije

Kreiranje baze

Ukoliko još uvek nemate kreiranu bazu, makar i potpuno praznu tj. bez tabela, krajnje je vreme da je kreirate. Kreiranje prazne baze na server se može uraditi na više načina.

Način 1. Koristeći programskog alata, **Sql Management Studio**, kao na slici.



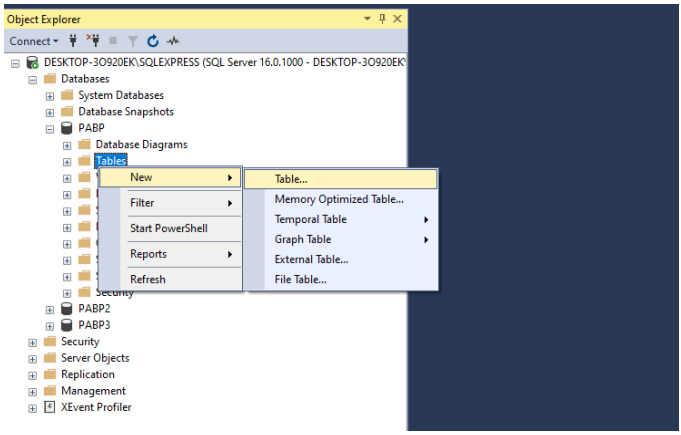
Slika 2.18. Kreiranje baze - MS SQL Server Management Studio

Način 2. Programski, tj. izvršavanjem sledećeg koda na nekom mestu ove ili neke druge aplikacije.

```
using Microsoft.Data.SqlClient;
using System.Data;
String str;
SqlConnection myConn = new SqlConnection("Data
Source=localhost\\sqlexpress;Initial Catalog=;Integrated
Security=True;Trust Server Certificate=True");

str = "CREATE DATABASE PABP ON PRIMARY " +
    "(NAME = PABP_data, " +
    "FILENAME = 'D:\\Program Files\\Microsoft SQL
Server\\MSSQL16.SQLEXPRESS\\MSSQL\\DATA\\PABP_data.mdf', " +
    "SIZE = 2MB, MAXSIZE = 10MB, FILEGROWTH = 10%)" +
    "LOG ON (NAME = PABP_log, " +
    "FILENAME = 'D:\\Program Files\\Microsoft SQL
Server\\MSSQL16.SQLEXPRESS\\MSSQL\\DATA\\PABP_log.ldf', " +
    "SIZE = 1MB, " +
    "MAXSIZE = 5MB, " +
    "FILEGROWTH = 10%)";

SqlCommand myCommand = new SqlCommand(str, myConn);
myConn.Open();
myCommand.ExecuteNonQuery();
myConn.Close();
```

U ovom kodu je korišćeno nekoliko predefinisanih string vrednosti za definisanje lokacije fajlova baze.

Nakon kreiranja baze podataka možemo kreirati tabele i podatke. Ovo se takođe može izvesti na dva načina: koristeći zasebnu alatku *Sql Management Studio* ili iz koda. Prvi način je svakako jednostavniji za one koji često ne koriste sql komande i radi se na sličan način kao i kreiranje baze. Dakle, desni klik na bazu pa izbor opcije, kao na slici:



Slika 2.19. Kreiranje novih tabela - MS SQL Server Management Studio

Iz koda bi postupak bio sledeći.

```
SqlConnection myConn = new SqlConnection("Data
Source=localhost\\sqlexpress;Initial Catalog=PABP;Integrated
Security=True;Trust Server Certificate=True");

myConn.Open();
string sql = "CREATE TABLE Knjiga" +
"(KnjigaId INTEGER CONSTRAINT PKeyKnjigaId PRIMARY KEY," +
"Naslov NVARCHAR(max), DatumIzdanja DateTime)";
SqlCommand cmd = new SqlCommand(sql, myConn);
cmd.ExecuteNonQuery();
```

Slično bi trebalo uraditi i sa drugom tabelom. Zatim u obe tabele treba dodati nekoliko redova podataka.

Način 3. Koristeći migracije.

Zbog svog značaja, ovaj način će biti posebno obrađen u narednom poglavlju.

Migracije

Pri razvoju aplikacija koje koriste baze podataka, gotovo nikada nije moguće sa sigurnošću predvideti sve potrebe i detalje na samom početku. Tokom razvoja,

zahtevi korisnika se menjaju, modeli se dopunjuju, a struktura baze mora da prati te izmene. Zbog toga su koncepti efikasnog prilagođavanja aplikacija i baza podataka postali sastavni deo modernih tehnika razvoja softvera. Jedan od ključnih mehanizama u tom procesu jesu migracije.

Šta su migracije?

Migracije predstavljaju sistematski način da se izmene u bazi podataka formiraju na osnovu izmena u modelima aplikacije. One obuhvataju:

- Početno kreiranje baze podataka (na osnovu definisanih modela).
- Praćenje i čuvanje prethodnog stanja baze pre svake izmene.
- Generisanje koda migracije koji opisuje promene u tabelama, kolonama i ograničenjima.

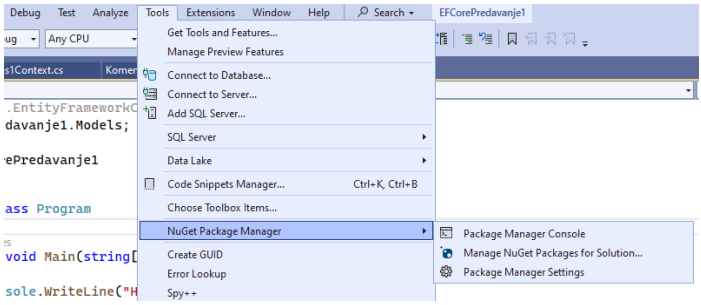
Na ovaj način, baza se razvija zajedno sa aplikacijom, a svaka promena ostaje zabeležena i može se pratiti hronološki.

EF i alat za migracije

Entity Framework (EF) poseduje ugrađen koncept migracija, koji se koristi putem paketa:

`Microsoft.EntityFrameworkCore.Tools`

Pokretanje migracija se obavlja u prozoru **`Package Manager Console`**, čije otvaranje iz glavnog menija se obavlja kao na slici:



Slika 2.20. Otvaranje konzole za rad sa migracijama

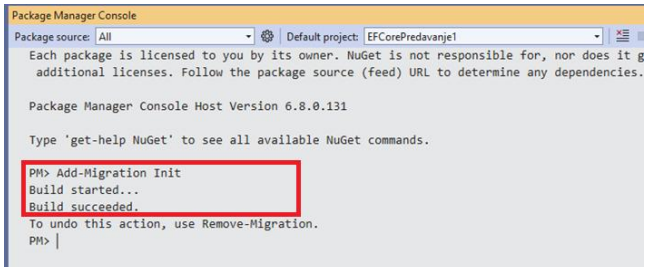
Kreiranje početne migracije

Prvi korak je kreiranje početne migracije, koja formira bazu i tabele na osnovu postojećih modela. Za to se koristi konekcioni string iz klase **`DbContext`**. Komanda u PM prozoru izgleda ovako:

```
Add-Migration NazivMigracije -Project NazivProjekta
```

U našem slučaju, ime projekta je podrazumevano i stoji u padajućoj listi pri vrhu konzolnog prozora.

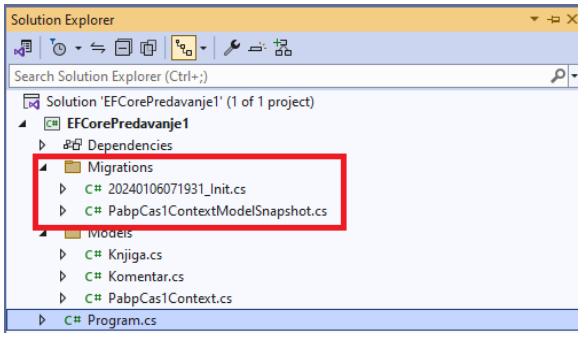
```
Add-Migration Init
```



Slika 2.21. Pokretanje migracije

Struktura migracije

Tokom izvršavanja migracije, kod mora biti ispravan i kompajliran. Ako postoje eventualne izmene u kodu biće automatski snimljene i kod preveden tj. kompajliran. Zatim sledi kreiranje koda tj. klasa migracije. Nakon završetka formira se poseban folder **`Migrations`** u okviru projekta.



Slika 2.22. Kreirani folder Migrations u okviru projekta

Uočavaju se dva fajla. Prvi je onaj koji sadrži naziv migracije u sufiksu, a na početku fajla stoji string formiran na osnovu datuma i vremena kreiranja migracije. Buduće migracije će biti smeštene hronološki jedna ispod druge.

Svaka klasa migracije je podeljena u dva fajla (parcijalne klase). Jedan fajl definiše model u aplikaciji, a drugi bazu tj. tabele i ograničenja u bazi.

Pogledajmo deo kod klase migracije koji definiše bazu.

```
public partial class Init : Migration
{
  protected override void Up(MigrationBuilder migrationBuilder){
    migrationBuilder.CreateTable(
      name: "Knjigas",
      columns: table => new
      {
         //. . .
      },
      constraints: table =>
      {
         table.PrimaryKey("PK_Knjigas", x => x.KnjigaId);
      });

      migrationBuilder.CreateTable(
         name: "Komentars",
         //. . .
  }

  /// <inheritdoc />
  protected override void Down(MigrationBuilder migrationBuilder)
  {
     migrationBuilder.DropTable(name: "Knjigas");
     migrationBuilder.DropTable(name: "Komentars");
   }
}
```

Metoda Up definiše kreiranje novih tabela i struktura, dok metoda Down omogućava vraćanje baze u prethodno stanje. Na ovaj način se obezbeđuje sinhronizacija baze podataka sa važećom migracijom.

Ažuriranje baze

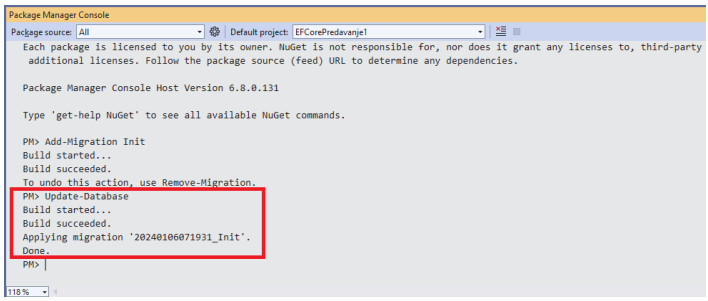
Nakon kreiranja migracije, baza se ažurira komandom:

```
Update-Database
```

Ova komanda sinhronizuje bazu sa poslednjom migracijom. Ako želimo da se baza postavi na određenu migraciju, navodimo njen naziv:

```
Update-Database -Migration 20211127165452_obrisanoPoljeX
```

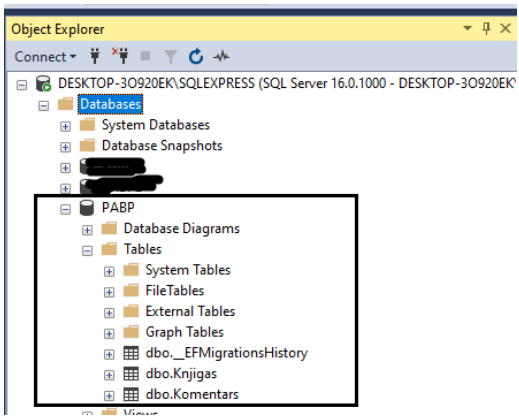
U konzolnom prozoru, izvršavanje komande izgleda kao na slici. Ovim je postupak završen.



Slika 2.23. **`Update-Database`** u PM prozoru

Provera rezultata

Nakon izvršene sinhronizacije, cela baza, odnosno sve tabele i podaci mogu se proveriti na strani servera pomoću dodatnih alata, kao na slici. Na ovaj način se potvrđuje da su migracije uspešno primenjene i da baza odgovara aktuelnom modelu aplikacije.



Slika 2.24. Kreirana baza i tabele nakon sinhronizacije

Postavka početnih podataka

EF vodi računa o mnogobrojnim aspektima upravljanja podacima. Jedan od tih aspekata je **inicijalno postavljanje podataka**. Nakon kreiranja baze, tabela i ograničenja, moguće je definisati inicijalni skup podataka koji će se automatski upisati u bazu. Ovaj proces se naziva inicijalizacija podataka.

Inicijalizacija se izvodi pomoću objekta **`modelBuilder`** i metode **`HasData`**. Na primer:

```
protected override void OnModelCreating(ModelBuilder
modelBuilder)
{
  modelBuilder.Entity<Knjiga>().HasData(
    new Knjiga { KnjigaId = 1, Naslov = "Price 1",
          DatumIzdanja = new DateTime(2024, 1, 1) },
    new Knjiga { KnjigaId = 2, Naslov = "Price 2",
          DatumIzdanja = new DateTime(2023, 2, 2) }
  );

  modelBuilder.Entity<Komentar>().HasData(
    new Komentar { KomentarId = 1, Ocena = 5, Tekst="Odlično" },
    new Komentar { KomentarId = 2, Tekst = "Nemam komentara..."
});
```

Ovim se obezbeđuje da se nakon kreiranja baze automatski dodaju početni podaci u tabele.

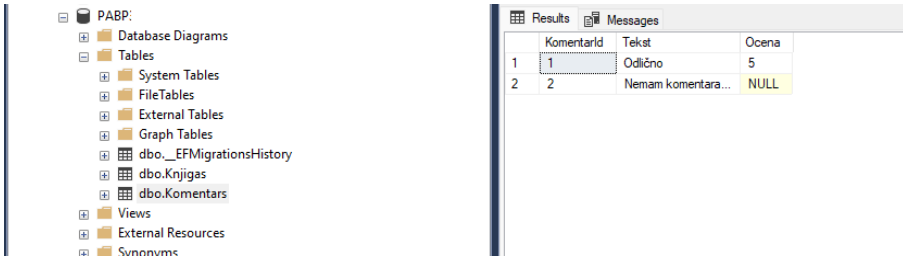
Generisani kod migracije

Nakon dodavanja migracije, EF generiše kod koji ubacuje podatke u bazu:

```
protected override void Up(MigrationBuilder migrationBuilder)
{
  migrationBuilder.InsertData(
    table: "Knjigas",
    columns: new[] { "KnjigaId", "DatumIzdanja", "Naslov" },
    values: new object[,]{
      { 1, new DateTime(2024, 1, 1, 0, 0, 0, 0,
         DateTimeKind.Unspecified), "Price 1" },
      { 2, new DateTime(2023, 2, 2, 0, 0, 0, 0,
         DateTimeKind.Unspecified), "Price 2" }
    });

  migrationBuilder.InsertData(
```

```
    table: "Komentars",
    columns: new[]{ "KomentarId", "Ocena", "Tekst" },
    values: new object[,]{
      { 1, 5, "Odlično" },
      { 2, null, "Nemam komentara..." }
    });
}
```

A nakon ažuriranja baze (**`PM>Update-Database`**) podaci se postavljaju u bazu, što je i prikazano na narednoj slici.



Slika 2.25. Kreirana baza sa inicijalnim podacima

Osnove upita za podatke

Kada baza i podaci postoje, sledeći korak je rad sa njima. Za dobavljanje i izmenu podataka koristimo LINQ upite u okviru EF aplikacije. U nastavku slede jednostavni primeri:

Primer 1. Dobavljanje podataka iz tabela **`Knjiga`**:

```
using (var db = new PabpContext())
{
    foreach (var k in db.Knjigas)
    {
        Console.WriteLine($"- {k.Naslov} izdata:
       {k.DatumIzdanja.ToString("dd.MM.yyyy.")}");
    }
}
```

Primer 2. Promena podataka u tabeli **`Komentar`**:

```
using (var db = new PabpContext())
{
```

```
    foreach (var kom in db.Komentars)
    {
        kom.Tekst = "Novi komentar koji dodajemo 13.12.";
    }
    db.SaveChanges();
}
```

Ovde smo pokazali da možemo pristupiti podacima i raditi sa njima preko LINQ odnosno koristeći EF biblioteku. Ipak, za više realnog rada neophodno je koristiti nešto složenije baze sa više tipova relacija.

Iz tog razloga, naredni naš korak biće da koristeći postojeću bazu sa više tabela i relacijskih veza, kreiramo potrebne modele za rad u našoj aplikaciji.

Alat *Scaffolding*

Za složenije baze sa više tabela i relacija, ručno kreiranje modela može biti zahtevno. Zbog toga .Net nudi alat Scaffolding, koji omogućava:

Automatsko generisanje modela na osnovu postojeće baze.

U ASP.NET aplikacijama, automatsko kreiranje CRUD operacija (`Create`, `Read`, `Update`, `Delete`) i pratećih pogleda.

Neophodni paketi

Da bi se Scaffolding koristio, potrebno je uključiti sledeće pakete:

- `Microsoft.EntityFrameworkCore.Tools`
- `Microsoft.EntityFrameworkCore.SqlServer`

Komanda za automatizovano generisanje modela

`Scaffold-DbContext`

Komandu prate parametri odnosno flegovi. Neki parametri ove komande su:

- `Connection` - Konekcijski string. Može se koristiti naziv name=<konek. string>. U ovom slučaju ime se preuzima iz konfiguracionog fajla.
- `Provider` - Provajder koji se koristi za pristup podacima. Tipično ovo je naziv NuGet paketa kao dobavljača podataka, na primer: `Microsoft.EntityFrameworkCore.SqlServer`.
- `OutputDir` - Folder gde se smeštaju generisani fajlovi. Putanja je relativna u odnosu na direktorijum projekta.

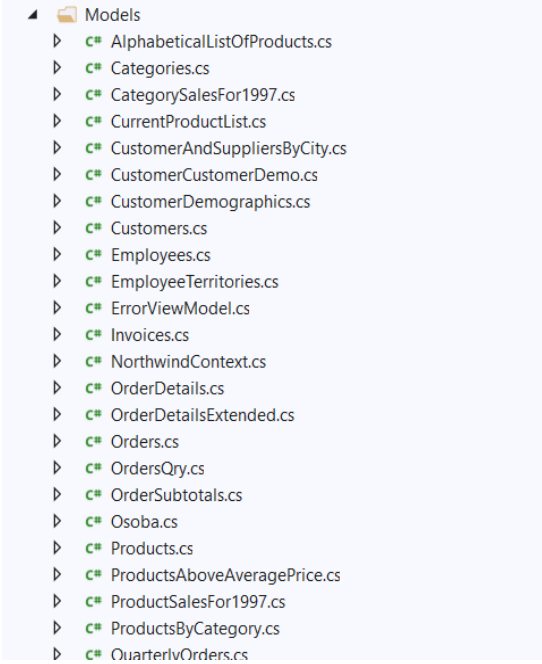
- `ContextDir` - Folder gde se smešta fajl DbContext klase. Putanja je takođe relativna.
- `Context` - Naziv DbContext klase koja se generiše.
- `Schemas` - Šeme tabela za koje se generišu entity tipovi. Ako se izostavi kreiraju se sve šeme.
- `Tables` - Tabele za koje se generišu entity tipovi. Ako se izostavi kreiraju se sve tabela.
- `DataAnnotations` – Označava da će se koristiti atributi za konfigurisanje modela. Ako se izostavi onda se podrazumevano koristi tzv. *Fluent API*, tačnije, svo konfigurisanje se obavlja kodom u metodi `OnModelCreating` u DbContext klasi.
- `Force` – Preklapanje postojećih fajlova.

Na primer:

```
PM>Scaffold-DbContext "Data Source=ZC-A7\SQLEXPRESS;Initial
Catalog=Northwind; Integrated Security=True"
Microsoft.EntityFrameworkCore.SqlServer -OutputDir Models
```

Nakon izvršavanja ove komande biće završeno automatizovano kreiranje modela. Na slici je prikazan deo prozora `Solution Explorer` nakon ove operacije.

Napomena. Baza Northwind dostupna je na adresi :
https://github.com/Microsoft/sql-server-samples/tree/master/samples/Databases/northwind-pubs



Slika 2.26. Pogled na generisanje modele

Zapaziti da je konekcijski string, inače korišćen u prethodnoj komandi, sačuvan u DbContext klasi, a koristi se pri konfigurisanju u metodi `OnConfiguring`.

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder){
 if (!optionsBuilder.IsConfigured){
  optionsBuilder.UseSqlServer("
Data Source=ZC-A7\\SQLEXPRESS;Initial Catalog=Northwind;Integrated Security=True
  ");
 }}
```

Relacije

Relacije koje povezuju tabele u bazi podataka mapiraju se u vidu određenih svojstava i ograničenja u kodu aplikacije. EF omogućava da se ove veze jasno definišu kroz modele i konfiguraciju, tako da aplikacija uvek zna kako su podaci međusobno povezani.

Osnovne karakteristike relacija

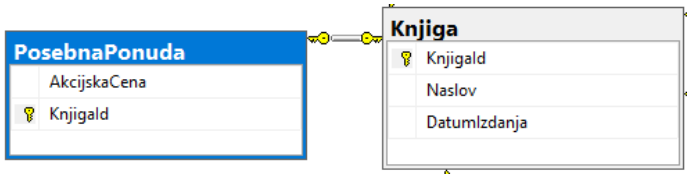
Relacije se najčešće dele na tri osnovna tipa:

- Jedan na jedan: *1* na *0..1*

  Relacija **jedan na nula ili jedan** označava situaciju u kojoj jedan objekat prvog tipa može biti povezan sa jednim ili nijednim objektom drugog tipa. Drugim rečima, veza je opciona – postoji mogućnost da objekat nema odgovarajući zapis u povezanoj tabeli.

  Na primer, jedna knjiga u tabeli Knjiga može imati promocionu cenu preko opcionog reda u tabeli PosebnaPonuda. Značenje tabele PosebnaPonuda se koristi ako se računa cena knjige. Tada se mora proveriti da li postoji podatak za tu knjigu u tabeli PosebnaPonuda koji je povezan sa tekućom knjigom.

  U tabeli PosebnaPonuda može postojati nezavisan primarni ključ, a veza će se ostvarivati preko stranog ključa koji odgovara knjizi. Međutim, moguće je da taj strani ključ bude primarni tj. KnjigaId može biti ključ za obe tabele i polje preko kojeg se vrši povezivanje.



Slika 2.27. Relacija 1 na 0..1

U kodu su kreirane klase za rad sa entitetima sa poljima kojima se definišu kolone u tabeli I sa referencama na povezane objekte relacijom 1 na 1.

Tabela 2.2. Povezivanje enetita u kodu za relaciju 1 na 1

Entiteti klasa
public partial class Knjiga {   //..definisanje atibuta/polja Knjiga

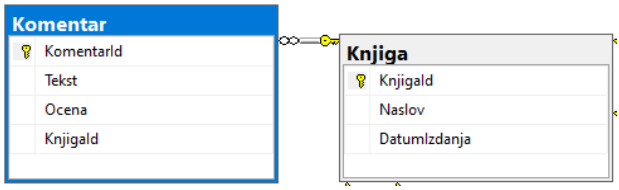
```
    public virtual PosebnaPonudum? PosebnaPonudum { get; set; }
}
public partial class PosebnaPonudum {
    //..definisanje atibuta/polja PosebnaPonuda
    public virtual Knjiga Knjiga { get; set; } = null!;
}
```

Klasa *DbContext, Metoda OnModelCreating

```
modelBuilder.Entity<Knjiga>(entity => {
    entity.ToTable("Knjiga");
    //..definisanje ostalih ograničenja
});
modelBuilder.Entity<PosebnaPonudum>(entity => {
    entity.HasOne(d => d.Knjiga).WithOne(p => p.PosebnaPonudum)
        .HasForeignKey<PosebnaPonudum>(d => d.KnjigaId)
        .OnDelete(DeleteBehavior.ClientSetNull)
        .HasConstraintName("FK_PosebnaPonuda_Knjiga");
    //..definisanje ostalih ograničenja
});
```

- Jedan na više: 1 na 0..*

Ako želite da omogućite kupcu da ostavi jedan ili više komentara na neku knjigu, može se formirati tabela Komentar koja će imati svoj primarni ključ, ali i strani ključ KnjigaId. Tako za jednu knjigu može biti nula ili više komentara, pogledajte sliku.



Slika 2.28. Relacija 1 na 0..*

Tabela 2.3. Povezivanje enetita u kodu za relaciju 1 na više

Entiteti klasa

```
public partial class Knjiga {
```

```
  //..definisanje atibuta/polja Knjiga
  public virtual ICollection<Komentar> Komentars {get; set;}
                                        = new List<Komentar>();
}
public partial class Komentar {
    //..definisanje atibuta/polja Komentar
    public virtual Knjiga Knjiga { get; set; } = null!;
}
```

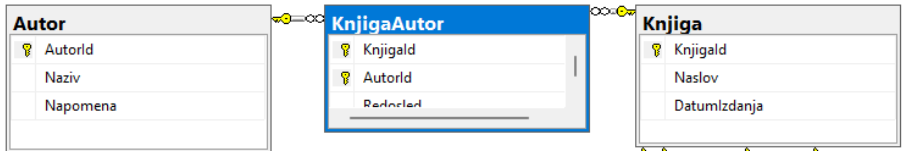
Klasa *DbContext, Metoda OnModelCreating

```
modelBuilder.Entity<Knjiga>(entity => {
    entity.ToTable("Knjiga");
    //..definisanje ostalih ograničenja
});
modelBuilder.Entity<Komentar>(entity => {
    entity.ToTable("Komentar");
    entity.HasOne(d => d.Knjiga)
          .WithMany(p => p.Komentars)
          .HasForeignKey(d => d.KnjigaId);
});
```

- Više na više: 1 na 0..* odnosno 0..* na 1

  Veza između tabele Autor i Knjiga može biti ostvarena posredstvom treće tabele koja sadrži reference ka obe tabele. Na ovaj način omogućava se da jedna knjiga ima vezu sa više autora i istovremeno da jedan autor može biti na više knjiga, što jeste očekivano kada je reč o prirodi ovih enetiteta. Postoji mogućnost da osim ova dva polja, tabela za povezivanje sadrži i druga polja.

  Počev od verzije EF Core 5, tokom sinhronizacije entiteta sa bazom, vrši se automatizovano dodavanje među-tabele u slučaju da postoji veza više na više. Zaseban među-entitet biće kreiran samo ako postoje dodatna polja osim polja koja su ključevi povezanih tabela.



Slika 2.29. Veza više na više

Tabela 2.3. Povezivanje enetita u kodu za relaciju 1 na više

Entiteti klasa

```
public partial class Autor
{
    //..definisanje atibuta/polja Autor
    public virtual ICollection<KnjigaAutor> KnjigaAutors {
       get; set; } = new List<KnjigaAutor>();
}
public partial class Knjiga {
    //..definisanje atibuta/polja Knjiga
    public virtual ICollection<KnjigaAutor> KnjigaAutors {
       get; set; } = new List<KnjigaAutor>();
}
public partial class KnjigaAutor
{
    //..definisanje atibuta/polja Komentar
    public virtual Autor Autor { get; set; } = null!;
    public virtual Knjiga Knjiga { get; set; } = null!;
}
```

Klasa *DbContext, Metoda OnModelCreating

```
modelBuilder.Entity<Autor>(entity => {
    entity.ToTable("Autor");
});
modelBuilder.Entity<Knjiga>(entity => {
    entity.ToTable("Knjiga");
    //..definisanje ostalih ograničenja
});
modelBuilder.Entity<KnjigaAutor>(entity =>
{
    entity.ToTable("KnjigaAutor");
    entity.HasKey(e => new { e.KnjigaId, e.AutorId });
    entity.HasOne(d => d.Autor)
          .WithMany(p => p.KnjigaAutors)
          .HasForeignKey(d => d.AutorId);
    entity.HasOne(d => d.Knjiga)
          .WithMany(p => p.KnjigaAutors)
          .HasForeignKey(d => d.KnjigaId);
});
```

U slučaju da u veznoj tabeli KnjigaAutor ne postoji treće polje osim ključeva: KnjigaId odnosno AutorId, ne generiše se zaseban enetit KnjigaAutor, drugim rečima između entiteta se ostvaruje veza više na više direktno, ali posredstvom koda koji opisuje tu vezu u metodi OnModelCreating. Generisani kod bi bio:

Tabela 2.4. Povezivanje enetita u kodu za relaciju 1 na više

Entiteti klasa

```
public partial class Autor
{
    //..definisanje atibuta/polja Autor
    public virtual ICollection<Knjiga> Knjigas { get; set; }
                    = new List<Knjiga>();
}
public partial class Knjiga {
    //..definisanje atibuta/polja Knjiga
   public virtual ICollection<Autor> Autors { get; set; }
                    = new List<Autor>();
}
```

Klasa *DbContext, Metoda OnModelCreating

```
modelBuilder.Entity<Autor>(entity => {
    entity.ToTable("Autor");
});
modelBuilder.Entity<Knjiga>(entity => {
    entity.ToTable("Knjiga");
    entity.HasMany(d => d.Autors)
          .WithMany(p => p.Knjigas)
          .UsingEntity<Dictionary<string, object>>(
             "KnjigaAutor", r =>
                 r.HasOne<Autor>()
                  .WithMany()
                  .HasForeignKey("AutorId"),
                   l => l.HasOne<Knjiga>().WithMany()
                        .HasForeignKey("KnjigaId")
```

```
.OnDelete(DeleteBehavior.ClientSetNull),
                    j =>
                    {
                        j.HasKey("KnjigaId", "AutorId");
                        j.ToTable("KnjigaAutor");
                    });
    //..definisanje ostalih ograničenja
});
modelBuilder.Entity<KnjigaAutor>(entity =>
{
    entity.ToTable("KnjigaAutor");
    entity.HasKey(e => new { e.KnjigaId, e.AutorId });
    entity.HasOne(d => d.Autor)
          .WithMany(p => p.KnjigaAutors)
          .HasForeignKey(d => d.AutorId);
    entity.HasOne(d => d.Knjiga)
          .WithMany(p => p.KnjigaAutors)
          .HasForeignKey(d => d.KnjigaId);
});
```

2.7 Pristup podacima

U ovakvim slučajevima za pristup podacima koristi se LINQ. Na primer, da bismo dobavili sve knjige čiji naslov počinje sa **Pesme**, upit izgleda ovako:

```
var db = new PabpContext()

db.Knjiga.Where(k=>k.Naslov.StartWith("Pesme")).ToList();
```

Ovakva struktura upita naziva se **fluent interface**.

Prvi deo, `db.Knjiga`, predstavlja DbContext pristup bazi podataka, najčešće konkretnoj tabeli.

Drugi deo, `Where(k => k.Naslov.StartsWith("Pesme"))`, koristi LINQ metode za formiranje upita.

Treći deo, u našem primeru `ToList()`, predstavlja zahtev za izvršavanje komande. Do trenutka izvršenja LINQ upita, čuva se niz komandi u obliku stabla izraza (eng. *expression tree*), koje još nisu realizovane. EF zatim prevodi ovo stablo izraza u konkretne SQL komande prema bazi podataka.

Izvršavanje komandi u EF -u se pokreće u sledećim slučajevima:

- kada se koristi `foreach` naredba,
- pozivom metoda `ToArray`, `ToDictionary`, `ToList`, `ToListAsync`,
- ili primenom LINQ operatora kao što su `First` i `Any`, koji se definišu na krajnjim delovima upita.

Dva tipa upita

Upit baze podataka iz prethodnog primera može se nazvati **standardnim upitom**, odnosno upitom tipa čitanje–upis. Takav upit čita podatke iz baze na način da ih možete ažurirati ili koristiti kao postojeći prikaz za novi unos, na primer prilikom kreiranja nove knjige sa već postojećim autorom.

Drugi tip upita je **`AsNoTracking`**, poznat i kao upit samo za čitanje. Ovaj tip upita koristi metod `AsNoTracking` dodat u LINQ upit i poboljšava performanse čitanja tako što isključuje određene EF funkcionalnosti.

```
db.Knjiga.AsNoTracking()
  .Where(k => k.Naslov.StartsWith("Pesme")).ToList();
```

Rad sa povezanim podacima

U našem slučaju tabela Knjiga je povezana sa nekoliko tabela. Pogledajmo kako se može pristupiti podacima u slučajevima kada su povezani. Podaci se mogu učitati na 4 načina: nestrpljivim učitavanje (eng. *Eager loading*), eksplicitnim učitavanjem (eng. *Explicit loading*), selektivno učitavanje (eng. *Select loading*), odnosno odloženo učitavanje (eng. *Lazy loading*).

Nestrpljivo učitavanje (eng. Eager loading)

U EF -u je moguće učitati relacije zajedno sa primarnom klasom entiteta u okviru istog upita. To se postiže korišćenjem metoda **Include** i **ThenInclude** u toku kreiranja upita.

Sledeći primer prikazuje učitavanje prvog reda iz tabele Knjiga kao instance entiteta Knjiga, uz nestrpljivo učitavanje povezane relacije Komentar:

```
var prvaKnjigaSaKomentarom = db.Knjiga.Include(k =>
k.Komentar).FirstOrDefault();
if(prvaKnjigaSaKomentarom != null)
{
    Console.WriteLine(prvaKnjigaSaKomentarom.Naslov + ": ");
    foreach (var kom in prvaKnjigaSaKomentarom.Komentar) {
        . . .
```

Ukoliko postoji više povezanih entiteta postupak se dosledno proširuje na ostale entitete. U našem slučaju, bilo bi sledeće:

```
var k1 = db.Knjiga
    .Include(k => k.Komentar)
    .Include(k => k.Oznaka)
    .Include(k => k.PosebnaPonuda)
    .FirstOrDefault();
Console.WriteLine(k1.Naslov + ": ");
foreach (var kom in k1.Komentar) Console.WriteLine(kom.Tekst);
foreach (var oz in k1.Oznaka) Console.WriteLine(oz.Opis);
Console.WriteLine("Akcija: " + k1.PosebnaPonuda.AkcijskaCena);
```

U slučaju postojanja među tabele u relaciji više na više, sa dodatnim poljima može se primeniti učitavanje u dva koraka, dakle

```
var k1 = db.Knjiga
.Include(k => k.KnjigaAutor)
        .ThenInclude(ka=>ka.Autor)
```

Od verzije EF Core 5 više **nije neophodno koristiti međutabelu za dobijanje podataka** u vezi više-na-više.

Prednost ovakvog načina učitavanja je što se svi podaci definisani pomoću Include i ThenInclude učitavaju efikasno, uz minimalni broj pristupa bazi. Nedostatak je što se učitavaju svi povezani podaci i u slučaju kada oni nisu potrebni.

EF omogućava **sortiranje** ili **filtriranje** relacijskih entiteta kada se koristi **Include** i **ThenInclude**. Ovo je korisno ako se učitava samo podskup podataka. Dakle, može se koristiti: `Where`, `OrderBy`, `ThenBy`, `Skip`,`Take`...

Na primer:

```
var k1 = db.Knjiga
    .Include(k => k.Komentar
        .Where(kom=>kom.Ocena>4))
    .Include(k => k.KnjigaAutor
        .OrderBy(a=>a.Redosled))
        .ThenInclude(ka=>ka.Autor)
    .Include(k => k.Oznaka)
    .Include(k => k.PosebnaPonuda)
    .FirstOrDefault();
```

Eksplicitno učitavanje (eng. Explicit loading)

Predstavlja učitavanje relacijskih podataka nakon učitavanja osnovne klase entiteta. Dakle, koristimo komande:

First – za čitanje prvog podatka

Load – eksplicitno učitavanje kolekcije ili reference podataka

Pimer explicitnog učitavanja:

```
var k1 = db.Knjiga.First();
Console.WriteLine(k1.Naslov + ": ");
db.Entry(k1).Collection(k => k.KnjigaAutor).Load();
foreach (var ka in k1.KnjigaAutor)
{
    db.Entry(ka).Reference(ka => ka.Autor).Load();
    Console.WriteLine("Autor,naziv:" + ka.Autor.Naziv);
}
    db.Entry(k1).Collection(k => k.Oznaka).Load();
    foreach(var ko in k1.Oznaka)
        Console.WriteLine("Oznaka,opis:"+ko.Opis);
    db.Entry(k1).Reference(k => k.PosebnaPonuda).Load();
    Console.WriteLine("PosebnaPonuda,AkcijskaCena:" +
                        k1.PosebnaPonuda.AkcijskaCena);
```

Alternativno, eksplicitno učitavanje može se izvršiti primenom posebnog upita umesto učitavanja relacija. U tom slučaju koristi se metoda `Query`. Iza ove metode može se koristiti standardni LINQ upit sa primenom `Where` i `OrderBy`:

```
var k1 = db.Knjiga.First();
Console.WriteLine(k1.Naslov + ": ");
var cnt=db.Entry(k1).Collection(k=>k.Komentar).Query().Count();
Console.WriteLine("Broj komentara:"+cnt);
```

Prednost ovakvog učitavanja je u tome što se relacijski podaci mogu učitati naknadno, što u slučaju dinamičkih zahteva predstavlja pogodnost. Takođe, korisno je kada je potrebno učitati samo jedan relacijski podatak na osnovu osnovnog entiteta.

Nedostatak je potreba za praćenjem podataka kroz organizaciju veza između tabela u bazi, što može predstavljati problem. Nestrpljivo učitavanje može biti efikasnije u situacijama kada su potrebni podaci unapred poznati.

Selektivno učitavanje – Učitavanje specifičnog dela primarnih entiteta i njegovih relacionih podataka

Ovo učitavanje se postiže primenom **`Select`** metode. Ovo je čest način dobavljanja željenih podataka. Obično je praćen željenim filtriranjem tj. metodom **`Where`**. Na primer:

```
var k1 = db.Knjiga.Where(k=>k.KnjigaId < 3).Select(k => new {
  k.Naslov,
  k.DatumIzdanja
});
foreach (var k in k1)
{
  Console.WriteLine("DatumIzdanja,naslov:" +
  k.DatumIzdanja.ToString("dd.mm.yyyy.") + ", " + k.Naslov);
}
```

Prednost ovog pristupa je da se samo potrebni podaci učitavaju, što može biti značajno efikasnije, naravno kada svi podaci nisu potrebni.

Nedostatak je što je potrebno napisati kod za svako svojstvo ili izračunavanje koje je potrebno.

Odloženo učitavanje (eng. Lazy loading)

Ovo je čest način pisanja složenih upita čije se izvršavanje odlaže do trenutka kada se podaci iz baze zaista zatraže. Iako logika može izgledati komplikovano, ovakvi upiti se pišu jednostavno. Uglavnom su dovoljno efikasni, ali ovaj tip učitavanja može negativno uticati na performanse. Generalno, izvršavanje se pokreće tek kada se zahtevaju podaci koji nisu već učitani, pa se optimizacija prepušta EF sistemu da je obavi u poslednjem trenutku.

Postavljanje odloženog učitavanja može se uraditi na dva načina:

- Dodavanje biblioteke Microsoft.EntityFrameworkCore.Proxies kada se konfiguriše sopstveni DbContext.
- Ubrizgavanje metode odloženog učitavanja u entity klasu preko njegovog konstruktora.

Pogledaćemo jedan primer ove prve opcije odloženog učitavanja.

Da bi se konfigurisao jednostavno odloženo učitavanje potrebno je:

- Instalirati paket Microsoft.EntityFrameworkCore.Proxies, ako već nije instaliran.
- Dodati ključnu reč *virtual* ispred svih svojstava koja su referencirana tj. koja su u relacijama.
- Dodati metodu UseLazyLoadingProxies pri podešavanju DbContext-a.

Primer:

```
public partial class Knjiga
{
  // svojstva kolona
  public int KnjigaId { get; set; }
  public string Naslov { get; set; } = null!;
  public DateTime DatumIzdanja { get; set; }
  // svojstva povezivanja
  public virtual PosebnaPonuda PosebnaPonuda { get; set; }
  public virtual ICollection<Komentar> Komentar { get; set; }
  public virtual ICollection<Oznaka> Oznaka { get; set; }
  public virtual ICollection<KnjigaOznaka> KnjigaOznaka {get;set;}
  public virtual ICollection<KnjigaAutor> KnjigaAutor {get;set;}
}
```

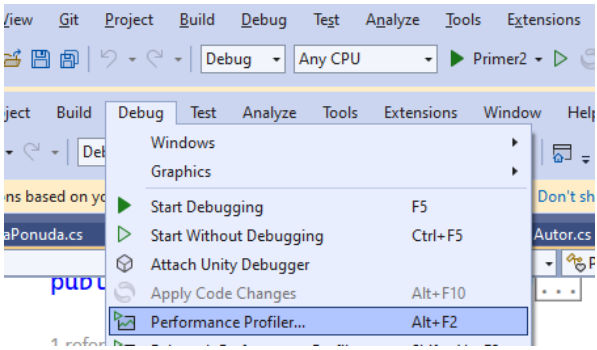
U ovom slučaju, čitanje relacionih podataka je jednostavno tj. nije potrebno koristiti dodatni Include metod u upitima, pošto se podaci iz baze učitavaju u relaciona svojstva. Na primer:

```
var optionsBuilder = new DbContextOptionsBuilder<PabpContext>();
optionsBuilder.UseLazyLoadingProxies();
using (var db = new PabpContext(optionsBuilder.Options))
{
    //var k1 = db.Knjiga.First();
    var k1 = db.Knjiga.OrderBy(x=>x.KnjigaId).Last();
    var komentari = k1.Komentar?.ToList();
    foreach (var k in komentari)
    {
        Console.WriteLine("Komentar: Tekst,Ocena:" + k.Tekst +
", " + k.Ocena);
    }
}
```

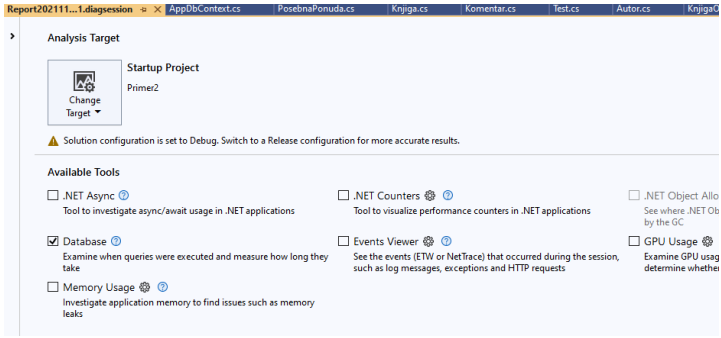
U prethodnom primeru formiraju se dva pristupa do baze. Prvi je kada se dobija podatak o jednoj knjizi, a drugi kada se za tu knjigu dobijaju komentari i ocene.

Nedostatak je eventualni pristup do podataka koji se sastoji od dva pristupa, što nije optimalno, tj. obično je cilj smanjiti broj pristupa. Na ovaj način se vremenom može kreirati jako puno upita što svakako dovodi do usporavanja i nepotrebnog opterećivanja servera.

(i)



(ii)



(iii)

Slika 2.30. Prikaz sql upita u Profiler prozoru. (i) Izbor Proviler-a iz menija, (ii)Podešavanje opcija (ii)Prikaz dobijenih rezultata primenom LINQ izraza

EF se karakteriše da se dohvatanje podataka, primenom LINQ upita, radi onda kada se dohvatanje podataka ne može transformisati u sql upit u finalnom select delu upita (eng. client vs. server evaluation). Treba obratiti pažnju da dobijeni upiti često imaju loše performanse.

```
using (var db = new PabpContext(optionsBuilder.Options)){
  var kn = db.Knjiga.Select(k => new{
    k.KnjigaId,
    k.Naslov,
    Autori = k.KnjigaAutor == null ? "" : string.Join(", ",
     k.KnjigaAutor.OrderBy(ka => ka.Redosled).Select(ka =>
      ka.Autor.Naziv))
    }).First();
  Console.WriteLine("Naslov,Autori:"+kn.Naslov+","+kn.Autori );
}
```

2.8 CRUD operacije

Pojam CRUD operacija predstavlja skraćenicu za 4 operacije nad bazama podataka: dodavanje, čitanje, izmenu i brizsanje (eng. `Create`, `Read`, `Update` `Delete`). U slučaju kada se te operacije povezuju sa klasama entiteta, CRUD operacije su povezane direktno sa stanjem pojedine instance EF klase.

Stanje

Svaka instanca *entity* klase ima jedno Stanje definisano svojstvom:

`db.Entry(instanca klase).State`

Stanje ujedno definiše šta će biti urađeno kada se primeni SaveChanges. Moguća stanja su:

- Added—podatak treba da se kreira u bazi, dakle SaveChanges dodaje novi zapis.
- Unchanged—podatak postoji u bazi i nije modifikovan. SaveChanges ne radi ništa tj. ignoriše ga.

- **Modified**—podatak postoji i promenjen je, dakle SaveChanges vrši ažuriranje.
- **Deleted**—podatak postoji u bazi ali je izbrisan, dakle SaveChanges ga briše.
- **Detached**—podatak se ne prati i SaveChanges ga ignoriše.

Obično se svojstvo State ne menja direktno već preko metoda. Ove metode i stanja omogućavaju da instanca bude praćena. Generalno, sve instance koje su dobijene čitanjem iz jedne baze podataka koje ne koriste **AsNoTracking** metodu se prate.

Dodavanje novog zapisa u tabeli

Dodavanje entiteta koji nema navigaciona svojstva sastoji se iz dva koraka:

- Dodavanje entiteta klasi DbContext.
- Izvršavanje SaveChanges metode.

```
using (var db = new PabpContext()){
    Knjiga0 k2 = new Knjiga0();
    k2.Naslov = "Naslov 2";
    k2.DatumIzdanja = DateTime.Now;
    db.Knjiga0.Add(k2);
    Knjiga0 k3 = new Knjiga0
    {
        Naslov = "Naslov 3",
        DatumIzdanja = DateTime.Now
    };
    db.Knjiga0.Add(k3);
    db.SaveChanges();
}
```

Analogno, dodavanje novog zapisa praćeno je sa dve SQL komande.

- Prva vrši dodavanje novog zapisa u tabelu (INSERT).
- Druga vrši vraćanje primarni ključ novog reda (SELECT).

Pogledamo sada dodavanje Knjige i novog kometara istovremeno:

```
using (var db = new PabpContext())
{
    Knjiga k = new Knjiga();
    k.Naslov = "Naslov 1";
    k.DatumIzdanja = DateTime.Now;
    Komentar kom = new Komentar();
    kom.Ocena = 5;
    kom.Tekst = "Odlična";
```

```
    k.Komentar = new Komentar[] { kom };
    db.Knjiga.Add(k);
    Knjiga k2 = new Knjiga{
        Naslov = "Naslov 2",
        DatumIzdanja = DateTime.Now,
        Komentar = new Komentar[]{
            new Komentar{
                Ocena = 5,
                Tekst = "Odlična"
            }
        }
    };
    db.Knjiga.Add(k);
    db.Knjiga.Add(k2);
    db.SaveChanges();
}
```

Dodati entitet na dalje biva praćen preko EF okruženja. Taj entitet dobija stanje Unchanged. Zbog toga što se kreiranjem tabele formira primarni ključ koji ima autoinkrementalno svojstvo, nakon ubacivanja novog podatka vrši se čitanje primarnog ključa novog reda u tabeli i to se vraća okruženju.

NoSql baza Cosmos DB nema ekvivalent autoinkrementu, pa se zato dodaje ključ koji je jedinstven kao na primer GUID. GUIDs se koriste kao primarni ključ u relacionim bazama.

SaveChanges obično se izvršava nakon više promena. Sve promene zajedno biće sačuvane, ako su ispravne ili odbačene ako bar jedna nije ispravna. Ovo ponašanje se javlja u transakcijama.

Na sličan način, dakle prateći logiku povezanih podataka, dodaju se i zapisi u slučaju kada neka referenca već postoji. U slučaju da Autor već postoji a radi se o vezi više na više, to bi bilo na sledeći način.

```
using (var db = new PabpContext()){
   var autor = db.Autor.SingleOrDefault(a => a.Naziv == "Pera");
   if (autor == null)
       throw new Exception("Autor ne postoji!");
   Knjiga k = new Knjiga();
   k.Naslov = "Naslov A";
   k.DatumIzdanja = DateTime.Now;
   KnjigaAutor ka = new KnjigaAutor();
   ka.Autor = autor;
   ka.Knjiga = k;
   k.KnjigaAutor = new KnjigaAutor[] { ka };
```

```
    Knjiga k2 = new Knjiga(); // da bi koristili kao referencu
    k2 = new Knjiga{
        Naslov = "Naslov B",
        DatumIzdanja = DateTime.Now,
        KnjigaAutor = new KnjigaAutor[] {
            new KnjigaAutor
            {
                Autor=autor,
                Knjiga=k2 //?? new Knjiga()
            }
        }
    };
    db.Knjiga.Add(k);
    db.Knjiga.Add(k2);
    db.SaveChanges();
}
```

Izmena zapisa

Izmena se sastoji od nekoliko koraka:

- Čitanje podataka.
- Promena jednog ili više svojstava.
- Snimanje novih podataka.

Praćenje se nastavlja sa izmenjenim podatkom.

```
var knjiga=db.Knjiga.FirstOrDefault(x=>x.Naslov=="Naslov A");
if (knjiga == null)
    throw new Exception("Knjiga ne postoji");
knjiga.DatumIzdanja = new DateTime(2019, 10, 1);
db.SaveChanges();
```

Na ovom mestu treba da spomenom i standardni način rada u veb aplikacijma. Dakle, u veb aplikacijama prikaz i promena podataka obavlja se pomoću dve nezavisne instance DbContext klase. Jedan od razloga je priroda veb aplikacija gde svaki HTTP zahtev predstavlja posebnu obradu u zasebnoj metodi, pa i sa novim DbContext objektom. Ovaj pristup naziva se diskonektovan. Tipično, jedno ažuriranje se sastoji od dva koraka:

- Početno čitanje podataka, koristeći jedan DbContext.
- Izvođenje promena, koristeći drugi DbContext.

Brisanje

Brisanje predstavlja najkritičniju operaciju koja može dovesti do stalnog gubitka podataka. Osim toga, brisanje je povezano i sa čuvanjem integriteta podataka u relacionim podacima. Organizacija i značaj podataka utiče na načine njihovog brisanja.

Meko brisanje

Predstavlja promenu statusa tj. ne radi se stvarno brisanje zapisa u bazi. Realizacija se sastoji od dodavanja nove kolone u tabeli koja će sadržati status zapisa u tabeli. Na primer:

```
public partial class Knjiga
{
  // ostala svojstva ostaju ista
  public bool obrisano {  get; set; }
  // odnosno početno filtriranje neobrisanih podataka u kodu:
  protected override void OnModelCreating(ModelBuilder
    modelBuilder){
    // ostatak koda ostaje
    modelBuilder.Entity<Knjiga>().HasQueryFilter(p =>
      !p.obrisano);
}
```

U slučaju da postoji više Context klasa, pri dodavanju nove migracije mora se definisati kontekst klasa, npr:

Add-Migration obrisano -Context Cas2EFCoreSca.Models.PabpContext

Testiranje

```
using (var db = new PabpContext())
{
  db.Knjiga.First().obrisano = true;
  db.SaveChanges();
  foreach (var k in db.Knjiga)
    Console.WriteLine("{0},{1}", k.KnjigaId, k.Naslov);
}
```

Brisanje sa relacijama

U ovom delu se bavimo zaista brisanjem zapisa u relacionoj bazi. Brisanje se obavlja primenom metode **Remove**, uz obaveznu brigu o povezanim podacima.

Na primer:

```
using (var db = new PabpContext())
{
    Knjiga0 k2 = db.Knjiga0.First();
    db.Remove(k2);
    db.SaveChanges();
}
```

Međutim, ovo je moguće samo ako entitet nije povezan sa drugim entitetima nekim relacijama. U ovom slučaju mora se voditi računa o njima tj. o referencijalnom integritetu.

Načini brisanja

Možemo prepoznati tri načina za brisanje povezanih podataka i čuvanje referencijalnog integriteta.

- Brisanje povezanih podataka može da se izvede preko servera baze ako je relacija napravljena da obezbedi kaskadno brisanje.
- Može se uraditi postavljanje stranog ključa kod zavisnih entiteta na `null`, ako pravilo za podatke za te kolone to dozvoljava.
- Ako se nijedno pravilo ne može realizovati server daje grešku.

U našem slučaju pri brisanju jedne knjige postoje entiteti koji ne mogu postojati ako su povezani sa tim entitetom Knjiga koji se briše. To su: PosebnaPonuda, Komentari, KnjigaAutor i KnjigaOznaka.

Podrazumevno, EF Core koristi kaskadno brisanje zavisnih povezanih stranih ključeva koje nisu `null`, koji su uključeni u objekat koji se briše.

```
using (var db = new PabpContext())
{
    Knjiga k = db.Knjiga
        .Include(x=>x.PosebnaPonuda)
        .Include(x=>x.KnjigaAutor)
        .Include(x=>x.KnjigaOznaka)
        .Include(x=>x.Komentar)
        .FirstOrDefault();

    db.Remove(k);
    db.SaveChanges();
}
```

Nakon brisanja, obično se pokreće migracija komandom:

```
>Update-Database
```

U slučaju problema, moguće je vraćanje baze unazan na neku prethodnu migraciju, takođe primenom komande ***Update***. Na primer:

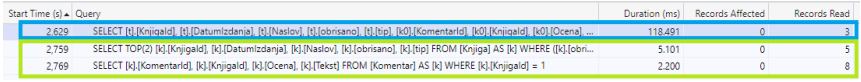
```
>Update-Database -Migration 20211127165452_obrisano_dodato
```

Čitanje iz baze podataka

Čitanje podataka koji su povezani može biti realizovano na više načina. Razmotrimo ovo u dva slučaja kroz naš primer kada se radi čitanje podataka iz tabele Knjiga odnosno pratećih povezanih tabela: Komentar, Autor,...

```
int knjigaId = 1;
var k1 = db.Knjiga.Include(x => x.Komentar).Single(x =>
x.KnjigaId == knjigaId);
int brKomentara1 = k1.Komentar?.Count ?? 0;

var k2 = db.Knjiga.Single(x => x.KnjigaId == knjigaId);
var kom = db.Set<Komentar>().Where(x => x.KnjigaId ==
1).ToList();
int brKomentara2 = k2.Komentar?.Count ?? 0;
```

Prvi upit učitava knjige sa povezanim komentarima, koristeći metod **Include**. Drugi upit učitava knjige bez komentara; onda se izvršava drugi upit koji posebno učitava Kometar podatke. Obe verzije daju isti rezultat.



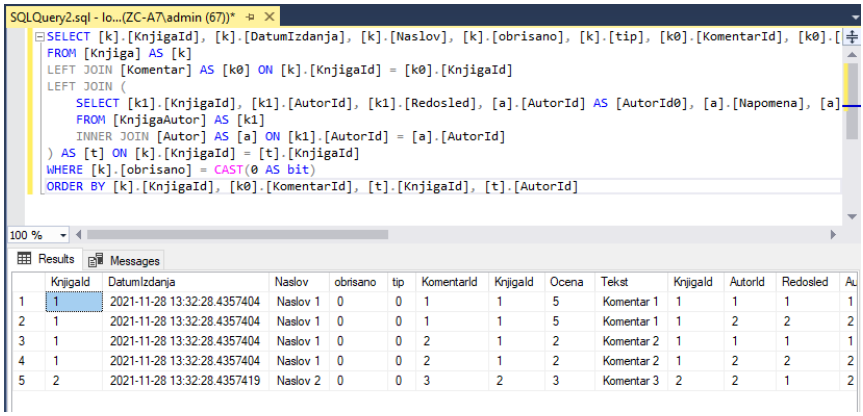
Slika 2.31. Prikaz konkretnog sql upita

Na ovom mestu je važno razumeti kako Include funkcioniše. Pogledajmo sledeći primer koji se oslanja na vezu više na više:

```
var k = db.Knjiga.Include(x => x.Komentar)
    .Include(x => x.KnjigaAutor)
        .ThenInclude(x=>x.Autor)
    .ToList();
```

Za svaki red tabele Knjiga učitavaju se podaci KnjigaAutor tabele, a onda, za svaki red u KnjigaAutor učitava se odgovarajući Autor. Dakle, ukupan broj redova je proizvod redova u Knjiga, KnjigaAutor i Autor.

Napomena: Ovo važi od EF Core 3.0. Pre ove verzije dobavljan je deo po deo.



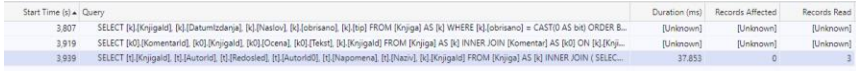
	KnjigaId	DatumIzdanja	Naslov	obrisano	tip	KomentarId	KnjigaId	Ocena	Tekst	KnjigaId	AutorId	Redosled	Au
1	1	2021-11-28 13:32:28.4357404	Naslov 1	0	0	1	1	5	Komentar 1	1	1	1	1
2	1	2021-11-28 13:32:28.4357404	Naslov 1	0	0	1	1	5	Komentar 1	1	2	2	2
3	1	2021-11-28 13:32:28.4357404	Naslov 1	0	0	2	1	2	Komentar 2	1	1	1	1
4	1	2021-11-28 13:32:28.4357404	Naslov 1	0	0	2	1	2	Komentar 2	1	2	2	2
5	2	2021-11-28 13:32:28.4357419	Naslov 2	0	0	3	2	3	Komentar 3	2	2	1	2

Slika 2.32. Priakaz konkretnog sql upita sa rezultatima

Počev od verzije EF Core 5 uvedena je metoda **`AsSplitQuery`** koja svaki Include posebno izvršava. Primenom ove naredbe na sledeći način

```
var k = db.Knjiga
    .AsSplitQuery()
    .Include(x => x.Komentar)
    .Include(x => x.KnjigaAutor)
        .ThenInclude(x => x.Autor)
    .ToList();
```

Dobija se niz sledećih upita koje prikazujemo kroz Profiler:



Start Time (s)	Query	Duration (ms)	Records Affected	Records Read
3,807	SELECT [k].[KnjigaId], [k].[DatumIzdanja], [k].[Naslov], [k].[obrisano], [k].[tip] FROM [Knjiga] AS [k] WHERE [k].[obrisano] = CAST(0 AS bit) ORDER B...	[Unknown]	[Unknown]	[Unknown]
3,919	SELECT [k0].[KomentarId], [k0].[KnjigaId], [k0].[Ocena], [k0].[Tekst], [k].[KnjigaId] FROM [Knjiga] AS [k] INNER JOIN [Komentar] AS [k0] ON [k].[Knji...	[Unknown]	[Unknown]	[Unknown]
3,939	SELECT [t].[KnjigaId], [t].[AutorId], [t].[Redosled], [t].[AutorId0], [t].[Napomena], [t].[Naziv], [k].[KnjigaId] FROM [Knjiga] AS [k] INNER JOIN (SELEC...	37,853	0	3

Slika 2.33 Prikaz konkretnog sql upita

A dobijeni upiti su:

```
SELECT [k].[KnjigaId], [k].[DatumIzdanja], [k].[Naslov], [k].[obrisano],
[k].[tip]
FROM [Knjiga] AS [k]
WHERE [k].[obrisano] = CAST(0 AS bit)
ORDER BY [k].[KnjigaId]
```

```
SELECT [k0].[KomentarId], [k0].[KnjigaId], [k0].[Ocena], [k0].[Tekst],
[k].[KnjigaId]
FROM [Knjiga] AS [k]
INNER JOIN [Komentar] AS [k0] ON [k].[KnjigaId] = [k0].[KnjigaId]
WHERE [k].[obrisano] = CAST(0 AS bit)
ORDER BY [k].[KnjigaId]
SELECT [t].[KnjigaId], [t].[AutorId], [t].[Redosled], [t].[AutorId0],
[t].[Napomena], [t].[Naziv], [k].[KnjigaId]
FROM [Knjiga] AS [k]
INNER JOIN (
    SELECT [k0].[KnjigaId], [k0].[AutorId], [k0].[Redosled],
[a].[AutorId] AS [AutorId0], [a].[Napomena], [a].[Naziv]
    FROM [KnjigaAutor] AS [k0]
    INNER JOIN [Autor] AS [a] ON [k0].[AutorId] = [a].[AutorId]
) AS [t] ON [k].[KnjigaId] = [t].[KnjigaId]
WHERE [k].[obrisano] = CAST(0 AS bit)
ORDER BY [k].[KnjigaId]
```

AsNoTracking, AsNoTrackingWithIdentityResolution

Prilikom učitavanja podataka u objekat DbContext, podaci se na dalje prate tako da ukoliko se izvode promene, brisanje ili dodavanje novih, svi objekti se mogu preslikati u odgovarajuću bazu podataka. Ovo jeste korisno, ako se koristi, ali utiče na performanse negativno, ako se ne koristi.

U praksi često se koristi samo čitanje podataka bez potrebe da se nad istim radi praćenje. U tom slučaju se ta činjenica može iskoristiti za značajno poboljšanje performansi tj. ubrzanje rada.

Metoda AsNoTracking ne rezrešava entitete na osnovu primarnog ključa. Ukoliko se dobavlja isti entitet više puta, EF će tretirati takve objekte kao različite. Ovo može biti problem ako je potrebno raditi sa relacijama ili održavati referentni integritet.

Pogledajmo primer čitanja podataka Knjiga sa referenciranim Komentar podacima:

```
var k1 = db.Knjiga.Include(x => x.Oznaka).ToList();
var k2 = db.Knjiga.AsNoTracking().Include(x =>
x.Oznaka).ToList();
```

```
var k3 =
db.Knjiga.AsNoTrackingWithIdentityResolution().Include(x =>
x.Oznaka).ToList();
```

Ako nema praćenja svakoj Knjizi pridruženi su podaci o Oznaci, što u praksi nije slučaj jer neke Oznake se dele između više Knjiga. O ovome treba voditi računa.

AsNoTracking variants	Time (ms)	Percentage difference
- no AsNoTracking (normal query)	95	100%
AsNoTracking	40	42%
AsNoTrackingWithIdentityResolution	85	90%

Slika 2.34. AsNoTracking vs AsNoTrackingWithIdentityResolution

Bezbedno rešavanje izuzetaka

U nastavku ovog poglavlja, bavimo se analizom pristupa podacima koji imaju referencirane podatke. U ovom delu se bavimo problemom čuvanja integriteta tj. sigurnim rešavanjem grešaka.

Izdvajaju se dva principa. Upotreba izuzetaka ili ne. Ako se koristi puno relacija često dolazi do izostavljanja upotrebe komande Include. U tom slučaju je bolje koristiti izuzetke, jer greška može biti teška za detekciju. Sa druge strane, često se koristi inicijalizacija praznom kolekcijom. Na ovaj način se lako piše kod za dodavanje novih instanci zajedno novim relacionim podacima. Međutim, ako se napiše kod

```
var k = db.Knjiga    //.Include(x=>x.Komentar)
        .First(x=>x.KnjigaId==1);
k.Komentar = new List<Komentar>{new Komentar{Ocena=3}};
db.SaveChanges();
```

a pri tome, **ukoliko se izostavi Include**, stari podaci iz baze neće biti izbrisani tj. pravi se kombinacija novih i starih podataka, što svakako može biti pogrešno. Drugi razlog zbog koga ovo nije dobro su performanse u slučaju da su već neki podaci dostupni a neki se dobavljaju.

Nekim LINQ komandama je potreban dodatni kod da bi se prilagodile načinu na koji baza podataka funkcioniše. Funkcije: **Average**, **Sum**, **Max** i druge agregatne komande vraćaju **null**. Jedina agregatna funkcija koji neće vratiti null je Count.

Neke LINQ komande mogu da rade sa bazom podataka, ali samo unutar čvrstih granica jer baza podataka ne podržava sve mogućnosti komande. Primer je komanda GroupBy; baza podataka može imati samo jednostavan ključ, a postoje značajna ograničenja u delu IGrouping.

https://docs.microsoft.com/en-us/ef/core/querying/complex-query-operators

Klasa AutoMapper

Klasa `AutoMapper` zajedno sa metodom `ProjectTo`, podrazumevano se koristi za konfigurisanje mapiranja, tj. kako se svojstva osnovne klase mapiraju u svojstva nove klase, koja odgovaraju po nazivu i tipu. AutoMapper može da mapira i određene relacije.

Za primenu je potrebno instalirati odgovarajući paket:

```
PM>Install-Package AutoMapper
```

Osim instalacije, treba definisati samo jednu instancu MapperConfiguration po aplikaciji i koja treba da se kreira pri pokretanju aplikacije.

```
var dto = db.Knjigas
.Select(p => new ChangePubDateDto{
       KnjigaId = p.KnjigaId,
       Naslov = p.Naslov,
       DatumIzdavanja = p.DatumIzdavanja
}).Single(k => k.KnjigaId == lastKnjiga.KnjigaId);
var dto = db.Knjiga
.ProjectTo<KnjigaDto>(config)
.Single(x => x.KnjigaId == lastKnjiga.KnjigaId);
```

Primer klasa

```
public class Knjiga{
       public int KnjigaId {get;set;}
       public string Naslov {get;set;}
       public decimal Cena {get;set;}
       public string Description
       {get;set;}
       public DateTime PublishedOn
       {get;set;}
```

```
        public string Publisher {get;set;}
        public string ImageUrl {get;set;}
        public PosebnaPonuda PP {get;set;}
        public ICollection<Komentar>Komentari {get;set;}
        ...
}
public class KnjigaDto{
        public int KnjigaId {get;set;}
        public string Naslov {get;set;}
        public decimal Cena {get;set;}
        public decimal? PPNovaCena {get;set;}
        public string PPOpis {get;set;}
        public ICollection<KomentarDto> Komentari {get;set;}
}
```

Obratite pažnju na upotrebu povezanih svojstava u mapiranom objektu. Pri tome se koristi sintaksa *Flattening relationships*. U toj sintaksi naziv svojstva u DTO klasi je kombinacija naziva navigacionog svojstva i svojstva u tipu navigacionog svojstva. Na primer, referenca entiteta knjige `PosebnaPonuda PP.NovaCena`, mapirana je u svojstvo `PPNovaCena` DTO-a.

Ova konfiguracija omogućava da mapirate kolekcije iz klase entiteta u DTO klasu, tako da možete da kopirate određena svojstva iz klase entiteta u svojstvu navigacione kolekcije.

U slučaju kompleksnog mapiranja koriste se dodatne opcije za podešavanje mapiranja. Za jednostavno mapiranje koristi se atribut AutoMap, uz već predstavljenu metodu ProjectTo.

```
[AutoMap(typeof(Knjiga))]
public class KnjigaDto
{
    public int KnjigaId { get; set; }
    public string Title { get; set; }
    public DateTime PublishedOn { get; set; }
}
```

Ovo mapiranje zahteva primenu specijalne klase, na primer Mapping Configruation. Tipično se koristi AutoMapper Profile klasa. Na primer:

```
public class KnjigaListDtoProfile : Profile
{
  public KnjigaListDtoProfile(){
    CreateMap<Knjiga, KnjigaListDto>()
    .ForMember(p => p.ActualPrice, m => m.MapFrom(s =>
     s.Promotion == null ? s.Price : s.Promotion.NewPrice))
```

```
    .ForMember(p => p.AuthorsOrdered, m => m.MapFrom(s =>
     string.Join(",       ",       s.AuthorsLink.Select(x       =>
x.Author.Name))))
    .ForMember(p => p.ReviewsAverageVotes, m => m.MapFrom(s =>
     s.Reviews.Select(y =>(double?)y.NumStars).Average()));
   }
}
```

Poslednja faza je registrovanje svih mapiranja sa injekcijom zavisnosti. Na sreću, AutoMapper ima NuGet paket koji se zove AutoMapper.Extensions.Microsoft .DependencyInjection koji sadrži metod AddAutoMapper, koji skenira sklopove koje obezbedite i registruje IMApper interfejs kao uslugu. Koristite IMapper interfejs za ubacivanje konfiguracije za sve vaše klase koje imaju atribut [AutoMap] i sve klase koje nasleđuju klasu AutoMapper-a Profile. U ASP.Net aplikaciji, sledeći isečak koda bi bio dodat metodu Configure klase Startup:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllersWithViews();
    // …
    services.AddAutoMapper( MyAssemblyToScan1,
MyAssemblyToScan2…);
}
```

2.9 Pitanja i zadaci

1. Koje su osnovne karakteristike .Net platforme? Za koje aplikacije se najčešće koristi?
2. Uporedi Visual Studio i Visual Studio Code.
3. Šta je to ORM? Koje su prednosti primene ORM-a?
4. Kako se mapiraju tabele, redovi i kolone baze u EF aplikacijama?
5. Koji je najbolji način za pisanje upita u EF aplikacijama?
6. Objasni tip podataka EntityType.
7. Šta su to asocijacije? Uporedi asocijacije i relacije.

8. Objasni navigaciona svojstva.
9. Pri upotrebi VS IDE postoji više šablona projekata. Objasni razlike.
10. Objasni način upotrebe i LINQ sintaksu?
11. Kako se koriste anonimni tipovi u LINQ izrazima?
12. Koji su operatori za filtriranje, a koji za sortiranje u LINQ?
13. Koji su skupovni operatori u LINQ? Kako se koriste agregatne funkcije?
14. Napiši upit koji prikazuje listu Knjiga sortirano po ceni i naslovu. Lista treba da prikaže samo knjige na pozicijama od 11-20 u redosledu.
15. Objasni razliku operatora: Join, GroupJoin, GroupBy, ToLookup.
16. Objasni namenu alata NuGet.
17. Dodajte paket Microsoft.EntityFrameworkCore.SqlServer projektu na dva načina.
18. Koje pakete koristimo u projektima za rad sa MS SQL Server bazom podataka?
19. Objani ulogu DbContext klase.
20. Na konkretnom primeru objasniti ulogu Server Explorer prozora.
21. Kreirajte sopstvenu bazu i 4 tabele. Uradite ovo koristeći SQL komande.
22. Objasnite pojam migracija. Kada se koriste i zašto?
23. Kako se kreira početna migracija?
24. Kako se kreira nova migracija? Objasni način vraćanja na prethodu migraciju.
25. Objasni ulogu virtuelnih metoda: `OnModelCreating`, Up, Down.
26. Koji alat se koristi za automatizovano generisanje modela?
27. U radu sa povezanim podacima postji više tipova učitavanja. Koji su to?
28. Objasni *nestrpljivo učitavanje*.
29. Objasni *eksplicitno učitavanje*.
30. Objasni *odloženo učitavanje.*
31. Svaka instanca jedne entity klase može imati više različitih stanja. O kojim stanjima je reč?
32. Objasni na primeru postupak dodavanja novog reda ako je tabela povezana sa drugom tabelom.
33. Objasni ulogu metode `AsNoTracking`.
34. Objasni na primeru postupak promene reda ako je tabela povezana sa drugom tabelom.
35. Šta je to meko brisanje?
36. Kako se radi brisanje sa relacijama?
37. Na konkretnom primeru objasni način čitanja povezanih podataka iz baze. Koja je uloga metode `AsSplitQuery`?

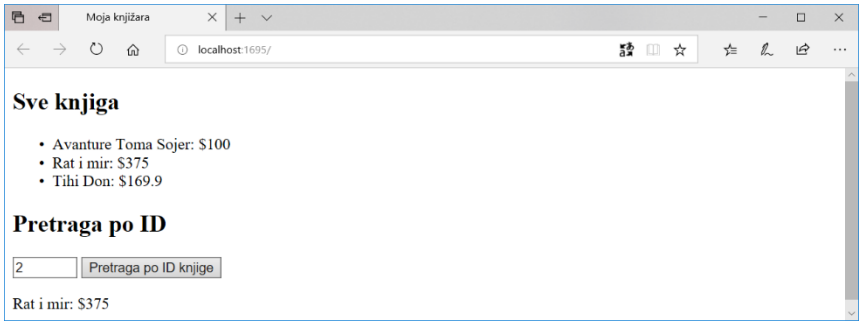
3. *Web Api* interfejs

Savremena upotreba HTTP protokola prevazilazi klasično prikazivanje veb stranica. HTTP protokol danas predstavlja temelj za izgradnju interfejsa za aplikacije tj. API-ja (eng. Application Programming Interface) koji omogućavaju razmenu podataka i pružanje različitih usluga. Zahvaljujući svojoj jednostavnosti, fleksibilnosti i univerzalnoj dostupnosti, HTTP je postao standardni komunikacioni protokol u gotovo svim softverskim okruženjima. Skoro svaka platforma poseduje biblioteke za rad sa HTTP-om, što omogućava kreiranje servisa dostupnih širokom spektru klijenata: od veb pregledača, preko mobilnih uređaja, pa sve do desktop aplikacija.

ASP.NET Web API je moćan radni okvir unutar .NET platforme, namenjen razvoju RESTful veb servisa. U nastavku ćemo prikazati kako se kreira jednostavan Web API koristeći ASP.NET, na primeru hipotetičke baze podataka knjiga.

3.1 Kreiranje projekta

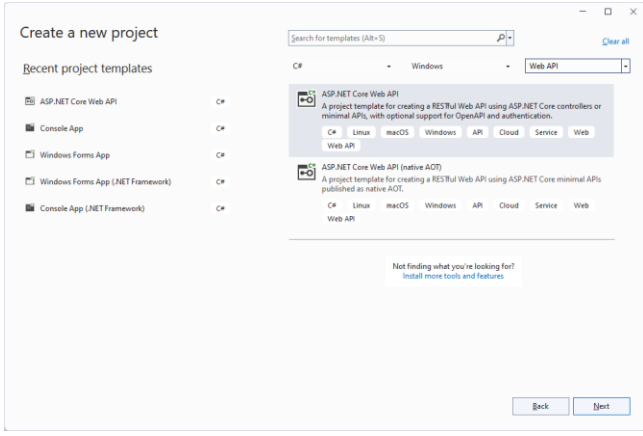
Kao što je već pomenuto, u okviru ovog kursa koristićemo `ASP.NET Web API` za izradu veb aplikacionog interfejsa koji upravlja bazom podataka `knjiga`. Veb stranica za prikaz i uređivanje podataka koristi VueJS, JavaScript ili jQuery za komunikaciju sa API funkcijama i prikaz rezultata. Na sledećoj slici prikazan je jednostavan primer takve stranice, bez dodatnih stilova tj. fokus je isključivo na funkcionalnosti.



Slika 3.1. Prikaz jedne veb stranice za prikaz liste knjiga

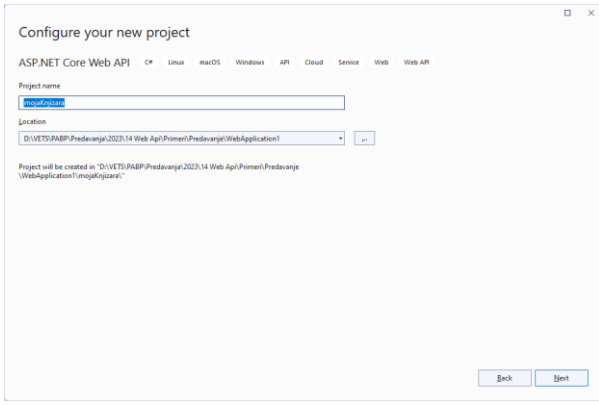
Oslanjajući se na pomenuti API interfejs, u nastavku sledi opis svih radnji u postupku kreiranje jednog Web API projekta:

Korak 1. Otvorite Visual Studio i kliknite na **Create New Project**. U okviru filtera za šablone odaberite opciju **Web API**, a zatim izaberite šablon **ASP.Net Web API**, kao što je prikazano na slici.



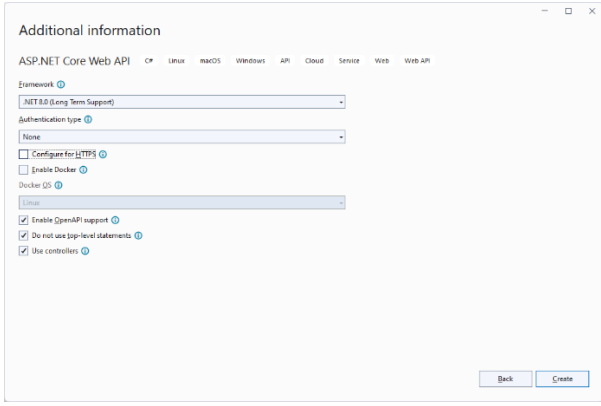
Slika 3.2. Filtriranje šablona u izbora tipa projekta: ASP.Net Core Web API

Korak 2. Imenujmo projekat. Unesite naziv projekta — u ovom primeru koristićemo ime **mojaKnjizara**.



Slika 3.3. Imenovanje projekta

Korak 3. Izbor radnog okvira. Ukoliko nije instaliran, potrebno je izvršiti instalaciju i ponovo pokrenuti kreiranje projekta.



Slika 3.4. Dodatne informacije o novom projektu

Nakon završetka ovih koraka, biće generisan početni projekat, spreman za dalji razvoj — počevši od definisanja modela.

Dodavanje modela

U arhitekturi ASP.NET Web API aplikacija, model predstavlja centralni objekat koji definiše strukturu podataka sa kojima aplikacija upravlja. Modeli se koriste za

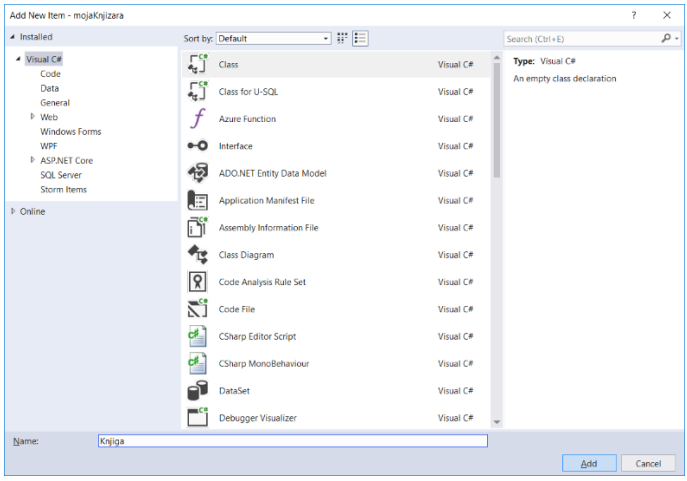
razmenu informacija između servera i klijenta, a ASP.NET Web API omogućava automatsku serijalizaciju modela u formate kao što su JSON ili XML (eng. Extensible Markup Language). Ovi formati se zatim ubacuju u telo HTTP odgovora, čime se omogućava jednostavna i standardizovana komunikacija sa klijentima.

Većina klijentskih aplikacija — uključujući veb pregledače, mobilne aplikacije i desktop softver — podržava obradu JSON ili XML formata. Štaviše, putem HTTP zaglavlja moguće je eksplicitno naznačiti koji format klijent očekuje, koristeći Accept header.

Kreiranje modela

Da bismo definisali model u okviru našeg Web API projekta, potrebno je da dodamo novu klasu u folder Models:

1. U prozoru **`Solution Explorer`**, kliknite desnim tasterom miša na folder Models.
2. Iz kontekstnog menija izaberite **Add**, zatim **Class**.



Slika 3.5. Dodavanje nove klase za model

Imenujte klasu **Knjiga**. Zatim definišite sledeća svojstva:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
```

```
namespace 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; }
    }
}
```

Ova klasa predstavlja osnovnu strukturu podataka o knjigama, uključujući: identifikator, naziv, kategoriju i cenu.

Dodavanje kontekstne klase

Da bi model bio povezan sa bazom podataka, potrebno je definisati kontekstnu klasu koja nasleđuje DbContext iz Entity Framework biblioteke. Ova klasa omogućava komunikaciju sa bazom i upravljanje entitetima:

```
using Microsoft.EntityFrameworkCore;
public class KnjizaraContext : DbContext{
    public DbSet<Knjiga> Knjigas { get; set; }
    protected override void OnConfiguring(
                            DbContextOptionsBuilder optionsBuilder)
    => optionsBuilder.UseSqlServer("Data Source = . . .
                                    Security = True; Encrypt=False");
    protected override void OnModelCreating(ModelBuilder modelBuilder){
        modelBuilder.Entity<Knjiga>(e=>e.ToTable("Knjiga"));
    }
}
```

U ovoj klasi:

- `DbSet<Knjiga>` predstavlja kolekciju knjiga u bazi podataka.
- `OnConfiguring` definiše konekcioni string za pristup SQL Server bazi.
- `OnModelCreating` mapira klasu `Knjiga` na tabelu istog imena u bazi.

Na ovaj način, model `Knjiga` zajedno sa kontekstnom klasom `KnjizaraContext` čini osnovu za upravljanje podacima u aplikaciji. Sledeći korak je omogućavanje interakcije sa ovim podacima putem HTTP zahteva — što

se postiže implementacijom kontrolera, koji obrađuju korisničke akcije i povezuju model sa API funkcionalnostima.

3.2 Rad sa kontrolerima

U ASP.NET Web API aplikacijama, **kontroler** predstavlja centralnu tačku kroz koju prolaze svi dolazni HTTP zahtevi. On je zadužen da primi zahtev, obradi ga i vrati odgovarajući odgovor klijentu. Zahtev može poticati od različitih izvora: krajnjeg korisnika koji koristi web ili mobilnu aplikaciju, druge klijentske aplikacije koja komunicira putem REST servisa, ili čak od drugog servera u okviru distribuiranog sistema. Bez obzira na izvor, kontroler je odgovoran za izvršavanje poslovne logike, pozivanje modela i interakciju sa bazom podataka ili drugim servisima.

Ako ste ranije radili sa **ASP.NET MVC**, koncept kontrolera će vam biti poznat. MVC kontroleri i Web API kontroleri dele sličnu filozofiju – oba služe kao posrednici između korisničkog zahteva i aplikacione logike. Međutim, postoji ključna razlika u njihovoj osnovnoj nameni i nasleđivanju. **MVC kontroleri** nasleđuju klasu `Controller` i fokusirani su na generisanje HTML sadržaja i prikazivanje pogleda (eng. views). **Web API kontroleri**, sa druge strane, nasleđuju `ControllerBase` (ili `ApiController` u starijim verzijama) i dizajnirani su da vraćaju podatke u formatu pogodnom za mašinsku obradu – najčešće JSON ili XML.

Ova razlika proizilazi iz same svrhe Web API-ja: dok MVC aplikacije prvenstveno ciljaju na krajnje korisnike kroz web interfejs, Web API aplikacije su namenjene razmeni podataka između sistema, integraciji sa mobilnim aplikacijama, IoT uređajima ili drugim servisima u okviru mikroservisne arhitekture.

Kontroler u Web API-ju se može posmatrati kao **ulazna tačka za RESTful operacije**. Svaka metoda unutar kontrolera obično odgovara određenoj HTTP akciji:

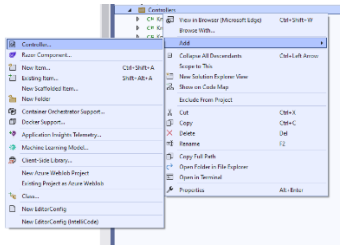
- `GET` metode za dobijanje podataka,
- `POST` metode za kreiranje novih resursa,
- `PUT` ili `PATCH` metode za ažuriranje postojećih resursa,
- `DELETE` metode za brisanje.

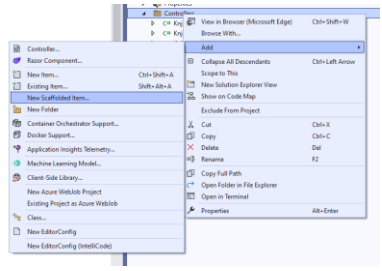
Na ovaj način, kontroler obezbeđuje jasan i standardizovan interfejs za komunikaciju sa aplikacijom.

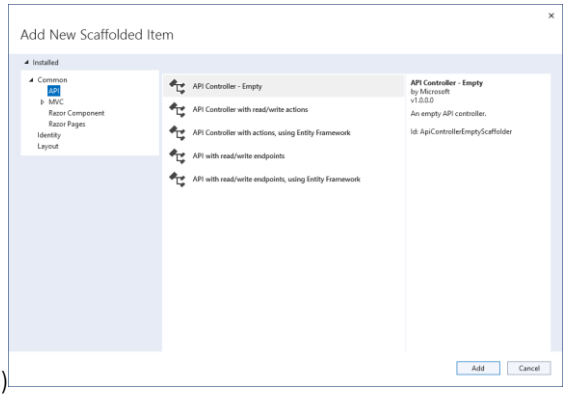
Pored osnovne funkcionalnosti, kontroleri u Web API-ju podržavaju i dodatne mehanizme kao što su **model binding** (automatsko mapiranje podataka iz HTTP zahteva na C# objekte), **validacija podataka**, **filteri** (za autentifikaciju, autorizaciju ili logovanje), kao i **direktno ubrizgavanje** za jednostavno povezivanje sa servisima i repozitorijumima. Sve ovo čini kontroler ne samo posrednikom, već i ključnim elementom u arhitekturi aplikacije.

Kreiranje kontrolera

Korak 1. U prozoru `Solution Explorer`, kliknite desnim tasterom miša na folder Controllers, zatim izaberite Add → Controller, ili Add → New Scaffolded Item. Zatim se bira jedna od stavki: API Controller – Empty, …

(i) 

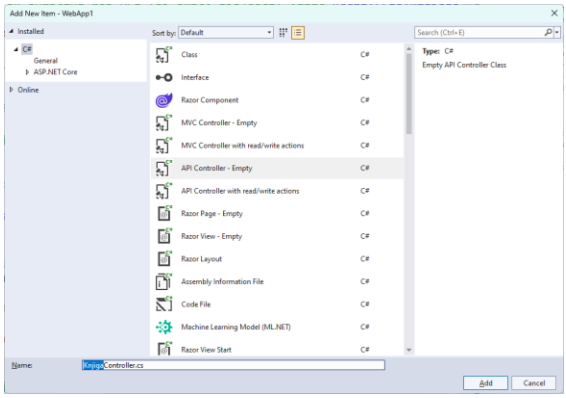
(ii) 

(iii) 

Slika 3.6. Dodavanje kontrolera: (i)direktno (ii)preko alatke *Scaffolding* (iii)određivanje stavke

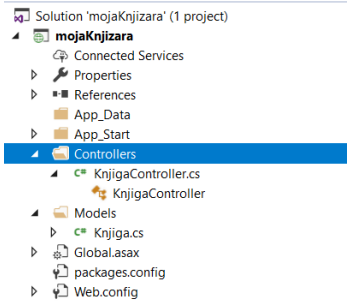
Korak 2. Izbor šablona. U dijalogu za dodavanje kontrolera, odaberite opciju **Web API Controller - Empty**, zatim kliknite na **Add**.

Korak 3. Imenovanje kontrolera. U dijalogu **Add Controller** imenujte kontroler **KnjigaController**, zatim odaberite **Add**.



Slika 3.7. Definisanje naziva kontrolera

Nakon toga, Visual Studio automatski kreira fajl **KnjigaController.cs** u folderu **Controllers**.



Slika 3.8. Pogled na prozor **Solution Explorer**

Napomena: Iako je uobičajeno da se kontroleri nalaze u folderu *Controllers*, to nije obavezno — raspored fajlova je stvar konvencije i organizacije projekta.

Implementacija kontrolera

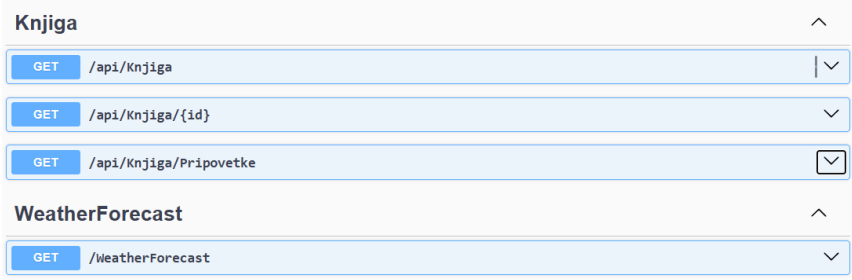
Ako fajl nije automatski otvoren, dvostrukim klikom ga otvorite i dopunite postojeći kod na sledeći način:

```
using Microsoft.AspNetCore.Mvc;
using WebApp1.Models;

namespace WebApp1.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class KnjigaController : ControllerBase
    {
        [HttpGet]
        public IEnumerable<Knjiga> Get(){
            using (var context = new TMPContext()){
                return context.Knjigas.ToList();
            }
        }
        [HttpGet("{id}")]
        public Knjiga JednaKnjiga(int id) {
            using (var context = new TMPContext()){
                return context.Knjigas.Where(
                    x => x.Id == id).FirstOrDefault();
            }
        }
        [HttpGet("Pripovetke")]
        public IEnumerable<Knjiga> GetPripovetke(){
            using (var context = new TMPContext()){
                return context.Knjigas.Where(x =>
                    x.Kategorija == "Pripovetke").ToList();
            }
        }
    }
}
```

Sada su u kontroleru definisane tri metode:

- **`Get()`** — vraća kompletnu listu knjiga kao IEnumerable<Knjiga>.
- **`JednaKnjiga(int id)`** — vraća jednu knjigu na osnovu prosleđenog identifikatora.
- **`GetPripovetke()`** — vraća listu pripovetki kao IEnumerable<Knjiga>.

Važno. Zapazite da svaka metoda mora imati sopstvenu rutu. Bez definisane rute može postojati samo jedna metoda za koju se koristi podrazumevana ruta kontrolera. U našem slučaju to je samo prva metoda.



Slika 3.9. Pogled na swagger stranicu nakon kreiranja metoda

Rutiranje i URL adrese

Svaka metoda u Web API kontroleru koristi jedinstvenu URL (eng. Uniform Resource Locator) adresu kako bi bila dostupna klijentskoj aplikaciji. Nije moguće da postoje dve metode, istog tipa (npr. Get ili Post) na istoj URL adresi.

URL adresa metode je pridružena metodi *SveKnjige*. Ona je definisana dekoratorom ispred same klase kontrolera: `[Route("api/[controller]")]`, što je u konkretnom slučaju: http://localhost:5266/api/Knjiga

Druga metoda koristi nastavak rute za adresu, a ujedno se taj nastavak može koristiti i kao podatak, na primer kao podatak za pronalaženje određenog objekta: http://localhost:5266/ api/Knjiga/2. U ovom slučaju je prosleđen id=2 za ovu metodu.

Veb API ima u kreiranom projektu podrazumevano definisan način rutiranja URI ka metodama. Ovaj način je dovoljan za gradivo ovog kursa. Ipak, za više informacija pogledati: https://learn.microsoft.com/en-us/aspnet/ web-api/overview/web-api-routing-and-actions/routing-in-aspnet-web-api.

Na sličan način se definišu i metode za POST zahteve, koje omogućavaju slanje podataka ka serveru. Podaci se mogu proslediti kroz telo HTTP poruke (body) ili

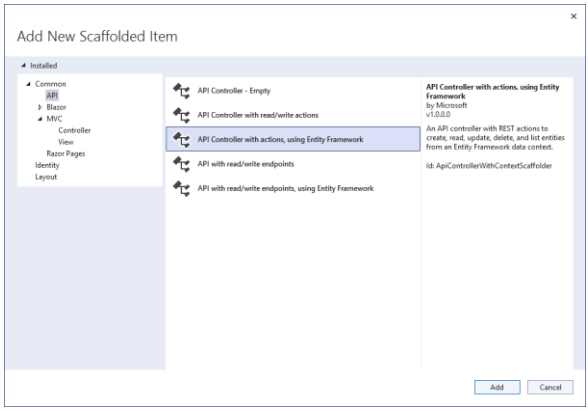
kao parametri u URL adresi. U sledećem poglavlju prikazaćemo primere kako se implementiraju POST metode za dodavanje novih knjiga.

Automatsko kreiranje akcija kontrolera

Ukoliko su modeli podataka pravilno definisani, moguće je koristiti automatsko kreiranje kontrolera. Ispravan model mora u potpunosti odgovarati bazi podataka sa kojom je povezan – to znači da je usklađen sa tabelama i kontekstnom klasom.

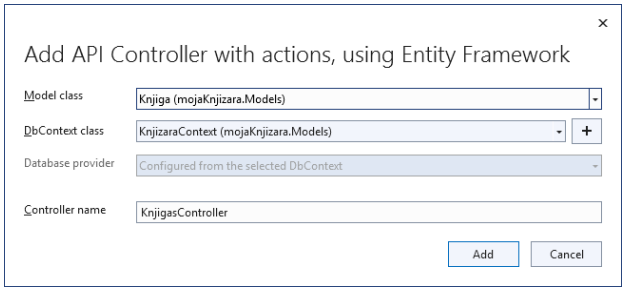
Jedan od najboljih načina da se dobiju ispravni modeli jeste korišćenje **scaffolding** alata. Provera ispravnosti modela može se izvršiti kreiranjem migracija i primenom komande **Update-Database**, čime se potvrđuje da baza može biti ažurirana na osnovu definisanih modela.

Korak 1. U prozoru `Solution Explorer`, kliknite desnim tasterom miša na folder Controllers, zatim izaberite Add → Controller. Ovoga puta odaberite opciju za dodavanje kontrolera sa akcijama koristeći Entity Framework.



Slika 3.10. Automatsko dodavanje akcija kontrolera

Korak 2. Otvara se pomoćni prozor **`Add API Controller with actions, using Entity Framework`**. Potrebno je da iz padajuće liste odaberete jedan model i odgovarajuću kontekstnu klasu, kao što je prikazano na sledećoj slici.



Slika 3.11. Izbor modela u postupku automatskog dodavanja akcija

Klikom na dugme Add (korišćenjem Scaffolding mehanizma), automatski se generišu akcije kontrolera. Dobijeni kod izgleda ovako:.

```
[Route("api/[controller]")]
[ApiController]
public class KnjigasController : ControllerBase{
  private readonly KnjizaraContext _context;
  public KnjigasController(KnjizaraContext context){
    _context = context;
  }

  [HttpGet]
  public async Task<ActionResult<IEnumerable<Knjiga>>> GetKnjigas(){
    return await _context.Knjigas.ToListAsync();
  }

  [HttpGet("{id}")]
  public async Task<ActionResult<Knjiga>> GetKnjiga(int id){
    var knjiga = await _context.Knjigas.FindAsync(id);
    if (knjiga == null){
       return NotFound();
    }
    return knjiga;
  }

  [HttpPut("{id}")]
  public async Task<IActionResult> PutKnjiga(int id, Knjiga knjiga){
    if (id != knjiga.Id){
      return BadRequest();
    }

    _context.Entry(knjiga).State = EntityState.Modified;
```

```
    try{
      await _context.SaveChangesAsync();
    }
    catch (DbUpdateConcurrencyException) {
      if (!KnjigaExists(id)) {
        return NotFound();
      }
      else{
        throw;
      }
    }

      return NoContent();
  }

  [HttpPost]
  public async Task<ActionResult<Knjiga>> PostKnjiga(Knjiga knjiga){
    _context.Knjigas.Add(knjiga);
      await _context.SaveChangesAsync();
      return CreatedAtAction("GetKnjiga", new { id = knjiga.Id }, knjiga);
  }

  [HttpDelete("{id}")]
  public async Task<IActionResult> DeleteKnjiga(int id) {
    var knjiga = await _context.Knjigas.FindAsync(id);
    if (knjiga == null) {
      return NotFound();
    }
    _context.Knjigas.Remove(knjiga);
    await _context.SaveChangesAsync();
    return NoContent();
  }

  private bool KnjigaExists(int id) {
    return _context.Knjigas.Any(e => e.Id == id);
  }
}
```

Postoji **nekoliko važnih razlika** u ovako generisanom kodu u odnosu na klasičan pristup:

1. Pristup bazi podataka koristi direktno ubrizgavanje. Umesto da ručno kreiramo instancu **_context** objekta (operatorom new) svaki put kada nam zatreba, mi je sada samo "zahtevamo" kroz konstruktor. Aplikacija je ta koja priprema objekat i prosleđuje nam ga. To pripremanje (registracija servisa) se dešava na samom početku, u main metodi (fajl Program.cs):

```
...
builder.Services.AddSwaggerGen();
builder.Services.AddDbContext<PabpContext>();
...
```

2. Asinhrone metode Sve metode kojima su pridružene rute napisane su kao asinhrone.

- Povratni tip je "omotan" klasom `Task<...>`. Na primer: `Task<ActionResult<Knjiga>>`
- Koriste se ključne reči `async` i `await`.

Ovo je ključno za performanse: omogućava serveru da obrađuje druge zahteve dok čeka odgovor od baze podataka (ne blokira rad), čime se aplikacija čini bržom i skalabilnijom.

Konfiguracija konekcijskog stringa

Dakle, izdvajaju se tri načina na koji aplikacija definiše gde se baza nalazi tj. kako koristi konekcijski string pri definisanju konteksnog objekta tj. objekta DbContext klase.

- **Ugrađen string**

  U inicijalno generisanom kodu, konekcijski string se često ugrađuje direktno u **OnConfiguring** metodu kontekstne klase. U tom slučaju, pri definisanju servisa u main metodi nije potrebno navoditi opcije.

  ```
  protected override void
  OnConfiguring(DbContextOptionsBuilder optionsBuilder) =>
  optionsBuilder.UseSqlServer("kon.string");
  ```

- **Spoljašnje zadavanje (preporučeno)**

  Fleksibilniji pristup je da se konekcijski string zadaje pri kreiranju servisa u **main** metodi. Da bi ovo radilo, vaša kontekstna klasa (**PabpContext**) mora imati javni konstruktor koji prima **DbContextOptions**. Pri automatskom kreiranju kontrolera, takav konstruktor postoji.

```
builder.Services.AddDbContext<PabpContext>(options=>
options.UseSqlServer("kon.string"));
```

- Korišćenje konfiguracionog fajla (najbolja praksa)

  Najbolja praksa je da se konekcijski podaci čuvaju odvojeno, u fajlu appsettings.json:

```
{
  ...
  "AllowedHosts": "*",
  "ConnectionStrings": {
    "PABPConnection": "kon.string"
  }
  ...
}
```

  Tada se u main metodi string učitava direktno iz konfiguracije:

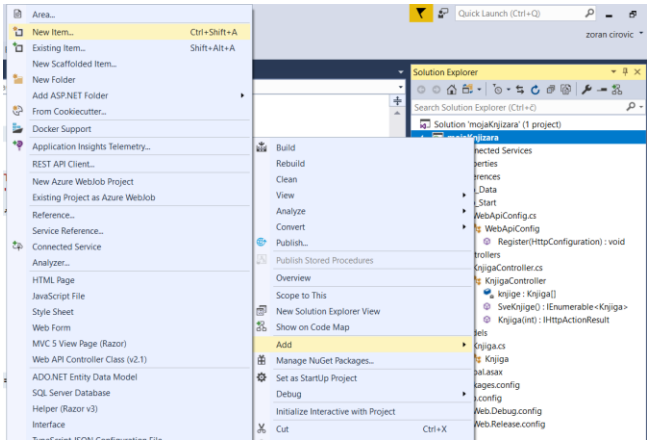
```
builder.Services.AddDbContext<PabpContext>(options =>
options.UseSqlServer(builder.Configuration.GetConnection
String("PABPConnection")));
```

3.3 JavaScript pozivi

U ovom poglavlju prikazaćemo kako da kreiramo jednostavnu HTML stranicu koja koristi JavaScript i AJAX (eng. Asynchronous JavaScript and XML) za slanje HTTP zahteva ka Web API interfejsu. Za realizaciju AJAX poziva koristićemo biblioteku jQuery, koja omogućava jednostavnu komunikaciju sa serverom i dinamičko ažuriranje sadržaja stranice — u ovom slučaju, prikaz knjiga.

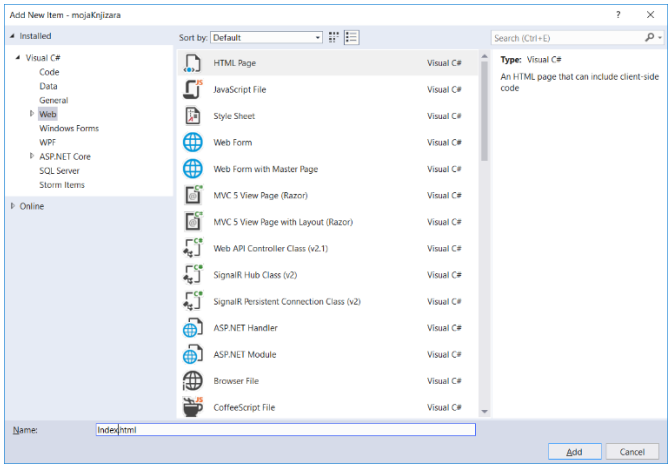
Kreiranje HTML stranice

Korak 1. U okviru Visual Studio okruženja, kliknite desnim tasterom miša na projekat u **Solution Explorer** panelu, zatim izaberite Add → New Item.



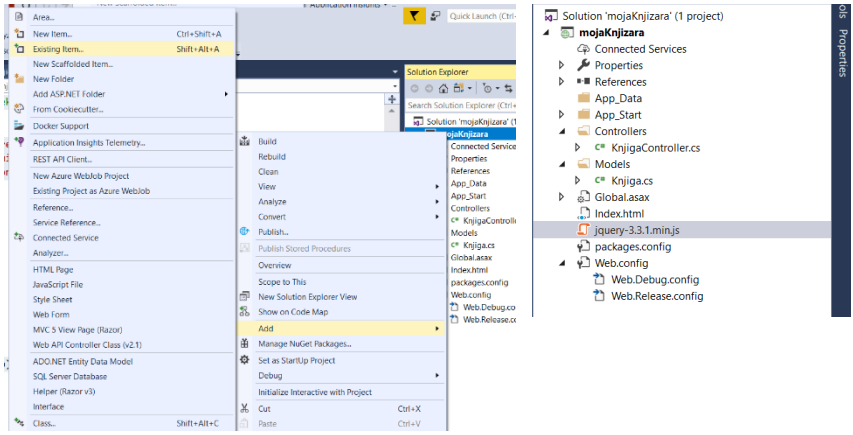
Slika 3.12. Dodavanje nove stavke

Korak 2. U dijalogu **Add New Item**, pod sekcijom Web u okviru Visual C#, odaberite HTML Page. Imenujte stranicu kao **Index.html**.



Slika 3.13. 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. Nakon dodavanja, u Project folderu pojavljuje se ubačeni `js` document, kao na slici:



Slika 3.14. Uključivanje postojeće biblioteke projektu

Nakon dodavanja, js kod se može koristiti i u aplikaciji, naravno povezivanjem u `script` elementu:

```
<!--<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 metoda *Get* i rešenje problema *CORS*

Get zahtev se može testirati neposredno preko web čitača unosom URL putanje do get metode u polje za adresu tj. u adres-bar čaitača. U stranici index.html može se dodati sledeći 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:

```
$("#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 primeru, odgovor se prikazuje direktno na stranici, dok se u drugom može videti u konzoli pregledača (F12).

Napomena: U novim verzijama veb čitača postoji podrazumevana zabrana pristupa API-ju sa drugih origin-a (npr. kada se frontend pokreće sa localhost), iz bezbednosnih razloga. Ovaj mehanizam je poznat kao CORS problem. Zbog toga se, pri kreiranju aplikacije, dodaje odgovarajuće CORS pravilo (policy).

```
public static void Main(string[] args){
    var builder = WebApplication.CreateBuilder(args);
    builder.Services.AddControllers();
    builder.Services.AddEndpointsApiExplorer();
    builder.Services.AddSwaggerGen();

    // dodato za CORS
    builder.Services.AddCors( x=>          //
        x.AddPolicy("AllowLocalhost",      //
            p=>p.AllowAnyOrigin()          //
                .AllowAnyMethod()          //
                .AllowAnyHeader()          //
        )                                  //
    );                                     //

    var app = builder.Build();
    if (app.Environment.IsDevelopment()){
        app.UseSwagger();
        app.UseSwaggerUI();
    }

    app.UseAuthorization();
    app.UseCors("AllowLocalhost");         // CORS
    app.MapControllers();
    app.Run();
}
```

Primena u HTML stranicama

Lista

Dakle, da bi se prikazala lista knjiga potrebno je poslati HTTP GET zahtev na URL: "/api/knjiga". jQuery funkcija `getJSON` šalje AJAX zahtev. Za odgovor očekuje se JSON. Funkcija ***done*** definiše povratnu funkciju tzv. `callback` koji se poziva kada je zahtev obavljen uspešno. U povratnoj funkciji (eng. callback) ažurira se DOM:

```
$(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, a zatim klikom na dugme pošalje zahtev tipa HTTP GET na adresu "/api/knjige/*id*", gde je deo *id* namenjen vrednosti 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);
    });
}
```

Post metoda

Post metodom se šalju podaci ka serveru unutar tela (body) HTTP poruke. Na serverskoj strani, podaci se čitaju i obrađuju u jednom prolazu.

Važno je napomenuti da se očekuje precizna struktura podataka; ukoliko se šalje samo jedan od predviđenih parametara, ostale vrednosti će na prijemnoj strani biti tretirane kao `null`.

U nastavku su prikazana tri primera standardnih korisničkih akcija koje rezultuju slanjem POST zahteva:

- Slanje kontakt forme – Korisnik unosi ime i poruku koji se pakuju u telo zahteva.
- Registracija korisnika – Slanje osetljivih podataka poput lozinke (koji ne bi trebali biti vidljivi u URL-u).
- Otpremanje datoteka (Upload) – Slanje slika ili dokumenata na server.

```
<form action=". . ./api/knjiga" method="POST">
    <div><input type="text" name="naziv"></div>
    <div><button type="submit">POST</button></div>
</form>
<form action=". . ./api/knjiga" method="POST">
    <div><input type="text" name="naziv"></div>
    <div><input type="text" name="kategorija"</div>
    <div><button type="submit">POST</button></div>
</form>
```

Prvi primer se odnosi na slanje iz forme samo jednog podatka, drugi sa dva podataka, a treći koristi Ajax. Nezavisno od realizacije, u svakom slučaju se mogu prihvatiti podaci metodom koja je navedena.

Slanje složenih podataka

Videli smo da u ASP.NET Web API aplikacijama, klijentska strana može slati podatke ka serveru putem HTTP POST zahteva preko formi. Generalno, podaci koji se šalju mogu biti jednostavni (npr. tekstualna vrednost) ili kompleksni (npr. objekat sa više svojstava). U ovom poglavlju prikazaćemo kako se složeni podaci šalju odnosno obrađuju na klijentskoj odnosno serverskoj strani, putem objekata.

Kompleksni objekti, kao što je klasa **Knjiga**, mogu se poslati putem HTML forme ili AJAX poziva. Na serverskoj strani, dovoljno je da metoda koja prima podatke ima parametar tipa tog objekta. ASP.NET Web API automatski vrši deserijalizaciju primljenih podataka u odgovarajući objekat, pod uslovom da je format ispravan (najčešće JSON).

Primer serverske metode:

```
[HttpPost]
public ActionResult<Knjiga> PostKnjiga(Knjiga knjiga)
{
    knjige.Add(knjiga);
    return CreatedAtAction("JednaKnjiga",
           new { id = knjiga.Id }, knjiga);
}
```

U ovom primeru:

- Metoda `Post` prima objekat `Knjiga` kao parametar.
- Ako je model validan i objekat nije `null`, vraća se HTTP 201 odgovor (**`Created`**).
- U suprotnom, vraća se HTTP 400 (Bad Request).

Važno je napomenuti da, ako neki podaci nedostaju, svojstva objekta dobijaju podrazumevane vrednosti (`null`, `0`, itd.).

Klijentska strana

Na klijentskoj strani, slanje kompleksnog objekta se realizuje kao JSON struktura:

```
var knjigaZaSlanje = {
    "Id":"33", //ne šalje se ako je autoincrement ID!!
    "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);
    });
});
```

U ovom primeru:

- Podaci se šalju kao JSON objekat.
- contentType mora biti postavljen na "application/json" vrednost da bi deserijalizacija na serveru bila uspešna.

Slanje prostih tipova

U nekim slučajevima, potrebno je poslati samo jednu vrednost — na primer, tekstualni podatak. I tada se koristi POST metod, ali sa drugačijim pristupom.

Klijentski primer:

{"": "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);
    });
});
```

Ovde se šalje objekat sa praznim ključem, što omogućava da se vrednost mapira direktno na string.

Serverska metoda:

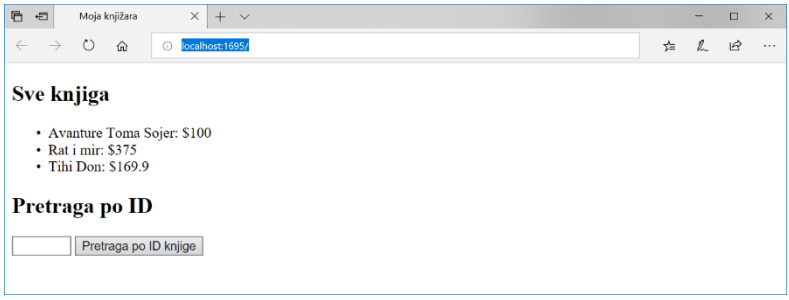
```
public void Post([FromBody]string value)
{
    //Obrada primljenog stringa
}
```

Testiranje

Klikom na taster **F5** u Visual Studio okruženju pokreće se aplikacija u tzv. **Debug modu**, koji omogućava testiranje i praćenje izvršavanja koda u realnom vremenu. Ovaj režim je izuzetno koristan za otkrivanje grešaka, proveru toka podataka i praćenje ponašanja aplikacije tokom rada.

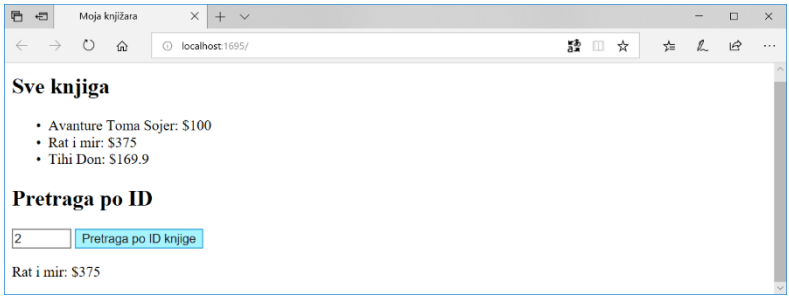
Nakon pokretanja, aplikacija se automatski otvara u podrazumevanom veb pregledaču, prikazujući početnu HTML stranicu koju smo prethodno definisali. Ova faza omogućava proveru da li su API rute ispravno definisane, da li se podaci pravilno prenose između klijenta i servera, kao i da li se greške pravilno hvataju i prikazuju.

Napomena: Ukoliko se aplikacija ne pokrene kako treba, proverite da li je izabrani projekat postavljen kao **StartUp Project**, da li su sve zavisnosti ispravno instalirane i da li je konekcioni string validan.



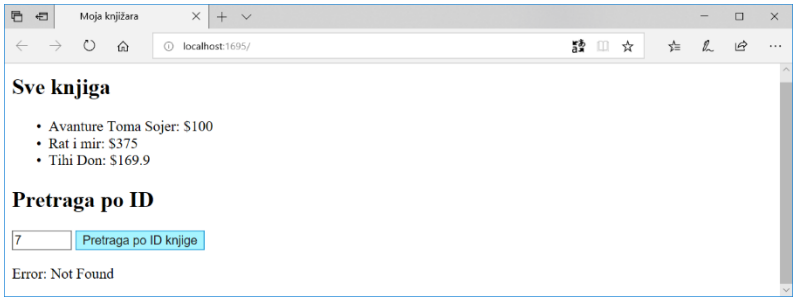
Slika 3.15. Prikaz početne stranice

Da bi se dobila knjiga po ID, unese se vrednost u polje klikne na dugme za pretragu:



Slika 3.16. Prikaz rezultata pretrage

Ako se unese neispravan ID, server vraća HTTP grešku:



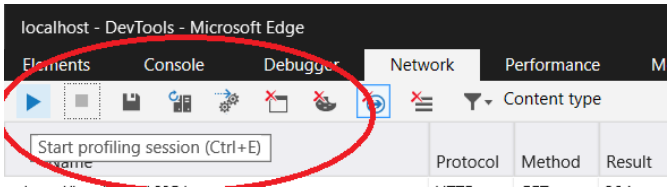
Slika 3.17. Prikaz pogrešnog zahteva za pretragu

3.4 HTTP zahtev i odgovor

Prilikom rada sa HTTP servisima, kao što je ASP.NET Web API, izuzetno je korisno imati uvid u tok komunikacije između klijenta (veb stranice) i servera. To uključuje pregled HTTP zahteva koje klijent šalje, kao i odgovora koje server vraća. Ovaj proces se može lako pratiti pomoću **alatke za programere** dostupne u svim modernim veb pregledačima.

Pokretanje alatke za razvoj

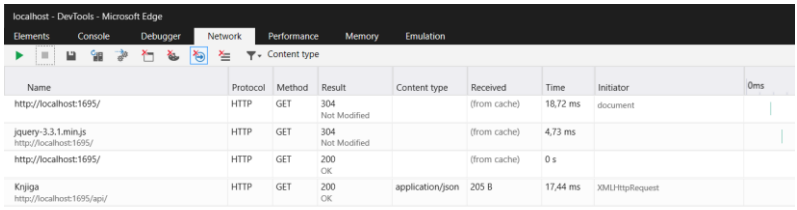
Alatka se pokreće pritiskom na taster **F12**, čime se otvara panel sa različitim karticama za razvoj i debagovanje. Za praćenje mrežnog saobraćaja, potrebno je odabrati karticu **Network**.



Slika 3.18. Kartice u veb čitaču korišćene za razvoj

Analiza zahteva ka Web API-ju

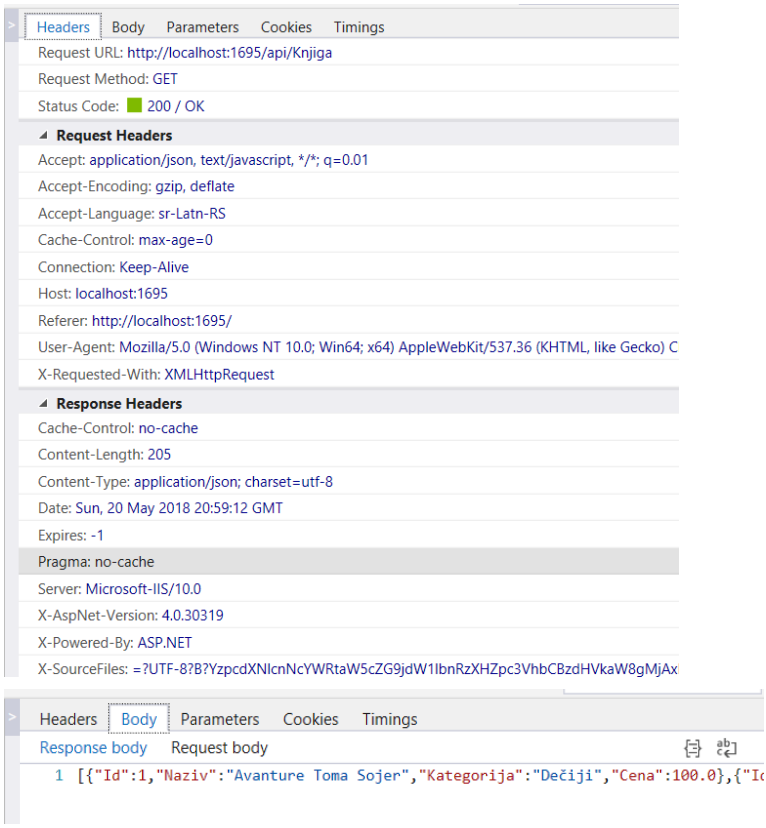
Sada se vratite se na veb stranicu i pritisnite taster F5 da ponovo učitati veb stranicu. Veb čitač će izvršiti hvatanje HTTP saobraćaja između veb čitača i veb servera. Slika pokazuje mrežni saobraćaj za stranicu:



Slika 3.19. Network kartica

U listi zahteva pronađite onaj koji se odnosi na poziv ka URL adresi **`api/Knjiga/`**. Kliknite na tu stavku da biste otvorili detalje o zahtevu i odgovoru.

U desnom delu prozora prikazuju se različiti tabovi sa tehničkim informacijama:



Slika 3.20. Praćenje sadržaja poruka

- **Headers** — prikazuje zaglavlja HTTP zahteva i odgovora (npr. `Content-Type`, `Status Code`, `Accept`, itd.);
- **Payload / Request Body** — prikazuje podatke koji su poslati serveru (kod POST zahteva);
- **Response** — prikazuje telo odgovora koje je server vratio (npr. JSON objekti sa podacima o knjigama);
- **Preview** — omogućava pregled odgovora u strukturiranom formatu.

3.5 Filteri

Filteri u ASP.NET aplikacijama predstavljaju moćan mehanizam za proširenje i kontrolu ponašanja metoda i kontrolera. Oni omogućavaju izvršavanje dodatne logike pre ili nakon poziva akcije, kao i upravljanje pristupom, keširanjem, greškama i drugim aspektima aplikacije. Filteri se primenjuju pomoću atributa (anotacija) koji se postavljaju iznad metoda ili klasa, u obliku uglastih zagrada.

Postoji više vrsta filtera, a najčešće korišćeni su:

Autorizacioni filteri

Filter `[Authorize]` koristi se za kontrolu pristupa metodama ili kontrolerima na osnovu korisničkih uloga i identiteta (`Role` i `Users`).

Primer:

```
[Authorize(Roles = "administrator", Users = "kondor")]
[HttpPost]
public ActionResult Contact(string inputMail, string inputText){...}
```

U ovom primeru, pristup metodi ima isključivo korisnik *kondor* koji je *administrator.* Ako se koristi samo `[Authorize]` pristup ima svaki ulogovani korisnik.

Filter može biti pridružen metodi, kao što je prikazano, ili svim metodama jedne klase, ako se postavi iznad deklaracije klase, na primer:

```
[Authorize]
public class HomeController : Controller{     }
```

U ovom drugom slučaju, moguće je napraviti izuzetak preklapajući autorizacioni filter cele klase posebnim filerima za pojedine metode, na primer:

```
[Authorize]
public class HomeController : Controller{
```

```
    public ActionResult Index(){
        return View();
    }
    [AllowAnonymous]
    public ActionResult About(){
        ViewBag.Message = "Your application....";
        return View("About");
    }
```

Sopstveni filteri

Možete definisati **vlastite filtere** nasleđivanjem klase `ActionFilterAttribute` i preklapanjem metoda kao što su `OnActionExecuting` ili `OnActionExecuted`.

Primer definicije:

```
public class myFilterAttr : ActionFilterAttribute
{
  public override void OnActionExecuted (ActionExecutedContext
filterContext){
    var request = filterContext.HttpContext.Request;
    //Logger.logRequest(request.UserHostAddress);
    base.OnActionExecuted(filterContext);
  }
}
```

Ovako napisana klasa može se kasnije koristiti kao novi filter:

```
[myFilterAttr]
public ActionResult Contact(){}
```

Da bi se novi filter koristio globalno potrebno je uraditi registraciju u `FilterConfig` klasi u folderu `App_Start`.

```
public static void RegisterGlobalFilters(GlobalFilterCollection
filters){
    filters.Add(new HandleErrorAttribute());
    filters.Add(new myFilterAttr());
}
```

Rezultujući filteri

Rezultujući filteri se primenjuju na metode koje vraćaju podatke, sa ciljem optimizacije performansi — najčešće kroz **keširanje**.

Primer sa `OutputCache`:

```
[OutputCache(Duration = 3600, VaryByParam = "none")]
public ActionResult Index(){   }

[OutputCache(Duration = 60, VaryByParam = "id")]
public ActionResult Details(int? id){   }
```

Filteri izuzetaka

Filteri izuzetaka omogućavaju definisanje ponašanja aplikacije u slučaju grešaka. Atribut [HandleError] se koristi za prikaz odgovarajućeg pogleda kada se dogodi određeni tip izuzetka.

Primer:

```
[HandleError(View="MathError", ExceptionType =
typeof(DivideByZeroException))]
[HandleError(View = "StackOfError", ExceptionType =
typeof(StackOverflowException))]
public ActionResult Create(){…
```

Promena URL adrese

U ASP.NET MVC aplikacijama, podrazumevana URL adresa ka određenom pogledu (view) formira se na osnovu imena kontrolera i metode unutar njega. Na primer, metoda `About()` u kontroleru `HomeController` automatski se mapira na URL adresu:

`/Home/About`

Međutim, postoje situacije kada želimo da:

- Više različitih metoda koristi isti pogled
- URL adresa sadrži specifične znakove (npr. crtica -)

- Prilagodimo URL radi SEO optimizacije ili boljeg korisničkog iskustva

U takvim slučajevima koristimo atribut `[ActionName]`, koji omogućava da se metodi dodeli alternativno ime koje se koristi u URL adresi.

Primer 1: Promena imena metode u URL-u

```
[ActionName ("about")]
public ActionResult Aboutttt()
{
    ViewBag.Message = "Your application....";
    return View();
}
```

U ovom primeru, iako se metoda zove `Aboutttt`, ona se poziva putem URL-a:

`/Home/about`.

Primer 2: URL sa specijalnim znakom (crtica)

```
[ActionName ("about-asp-mvc")]
public ActionResult About()
{
    ViewBag.Message = "Your application....";
    return View("About");
}
```

Ovde se metoda `About()` mapira na URL adresu:

`/Home/about-asp-mvc`

Ova tehnika je korisna kada želimo da URL bude deskriptivan, čitljiv i optimizovan za pretraživače.

Napomena o testiranju

Metode sa atributom `ActionName` mogu se testirati direktnim unosom URL adrese u adresnu traku veb pregledača. Na primer:

`http://localhost:xxxx/Home/about-asp-mvc`

Gde `xxxx` predstavlja broj porta koji je dodeljen aplikaciji prilikom pokretanja.

Dodatne napomene

Atribut `ActionName` ne menja ime metode u kodu, već samo način na koji se ona poziva putem URL-a.
Ako se koristi `ActionName`, metoda mora biti jedinstvena u okviru kontrolera — ne sme postojati druga metoda sa istim URL imenom.
Ova tehnika se može kombinovati sa drugim atributima kao što su [`HttpGet`], [HttpPost], [`Authorize`], itd.

3.6 Pitanja i zadaci

1. Koji protokol se koristi kao osnova za izgradnju Web API servisa?
2. Koji radni okvir se koristi za izgradnju Web API-ja u .NET okruženju?
3. Koji atribut se koristi za definisanje rute u Web API kontroleru?
4. Koji format se najčešće koristi za razmenu podataka u Web API-ju?
5. Koja atribut omogućava pristup samo određenim korisnicima?
6. Napisati primer modela koji opisuje objekat Automobil.
7. Za prethodni primer dodati konekciju sa bazom podataka.
8. Kako se naziva klasa koja omogućava pristup bazi podataka?
9. Za model Automobil i omogućenu vezu do baze podataka, kreirati kontroler.
10. Napisati jednu stranicu i pokazati sve načine komunikacije sa serverskim delom aplikacije kroz primer rada sa modelom Automobil.
11. Koji alat u veb pregledaču omogućava praćenje HTTP saobraćaja?
12. Koji atribut se koristi za promenu URL imena metode?
13. Koji atribut omogućava da metoda bude dostupna bez autentifikacije?
14. Koja metoda se koristi za prikaz jedne knjige po ID-u?

4. TypeScript

JavaScript je dizajnirao Brendan Eich iz Netscape-a 1995. za nekoliko dana sa ciljem da bude pristupačan i lak za korišćenje za rad na web stranicama. Mada postoje nedostaci JavaScript jezika, ipak, JavaScript je značajno unapredjen od svog nastanka. Njegov upravni odbor, neprekidno objavljuje nove verzije jezika, dosledno svake godine od 2015. sa novim funkcijama koje ga dovode u red sa drugim modernim dinamičkim jezicima. Impresivno je da, čak i sa redovnim novim jezičkim verzijama, JavaScript je uspeo da održi **kompatibilnost** decenijama unazad u različitim okruženjima, uključujući pregledače, ugrađene aplikacije kao i izvršavanja na strani servera. Danas je JavaScript izuzetno fleksibilan jezik sa mnogo prednosti.

Programeri često koriste JavaScript bez značajnih jezičkih proširenja. Za takav pristup kaže se, da ga koriste kao „vanilu“ tj. originalnu verziju. Najveći nedostatak JavaScript jezika, za većinu programera koji navode nedostatke, je što JavaScript praktično ne pruža ograničenja tj. pravila u načinu na koji pišete kod. Sa jedne strane moguće je brzo napisati kod na više načina. Međutim, ako se radi o složenim projektima, onda ta osobina postaje nedostatak.

TypeScript je programski jezik i skup alatki za generisanje JavaScript koda. Tvorac je **Anders Hejlsberg**, dizajner C# u Microsoft-u. Nastao je početkom **2010**. godine, a prva verzija otvorenog koda objavljena je **2012**.

TypeScript je **nadskup** ili **superset** JavaScript jezika. Najbitnija karakteristika je da je TypeScript (skraćeno TS) **čist objektni** jezik. Osnovne osobine koje čine TS su:

- **Uključuje postojeću JavaScript sintaksu**, plus novu sintaksu specifičnu za TipeScript za definisanje i korišćenje tipova.

- Provera tipova: TS podrazumeva primenu programa koji preuzima skup datoteka napisanih u JavaScript-u i/ili TipeScript-u, „razume“ sve programske konstrukcije (promenljive, funkcije, klase,…) i vraća poruku o grešci ako je nešto pogrešno podešeno ili upozorenje ako postoji nešto što bi moglo bolje biti definisano.
- Kompajler: program koji pokreće proveru tipa, izveštava o svim problemima, a zatim kreira ekvivalentni JavaScript kod.
- Jezičke usluge: program koji koristi proveru tipa i tu proveru prenosi editorima koda, kao što je VS Code, kako da se obezbede korisne funkcije programerima

Dakle, TypeScript podržava rad sa: klasama, interfejsima itd. To je jezik otvorenog koda koji se kompajlira tj. prevodi u JavaScript kod. Lako se pokreće uz NodeJS okruženje, kao i na web čitačima.

Sve osobine poslednjih standarda JavaScript jezika, poznatih kao ECMAScript, potpuno su podržane u TypeScipt jeziku, a uz to TS ima sopstvene objektno-orijentisane karakteristike.

Poređenje sa JavaScript jezikom

JavaScript je dugi niz godina jedan od najpopularnijih programskih jezika. JS je lagan i često korišćen programski jezik za kreiranje interaktivnih dinamičkih veb stranica. Razvijen je 1995 od kompanije Netscape, prvobitno nazvan LiveScript, kasnije preimenovan u JavaScript.

TypeScirpt je nadskup tj. superset JavaScript jezika koji primarno omogućava: tipizaciju, klase i interfejse. Uz sve to, velika prednost je mogućnost da se preko IDE okruženja identifikuju česte greške u kodu.

Razlike između ova dva jezika su navede u sledećoj tabeli.

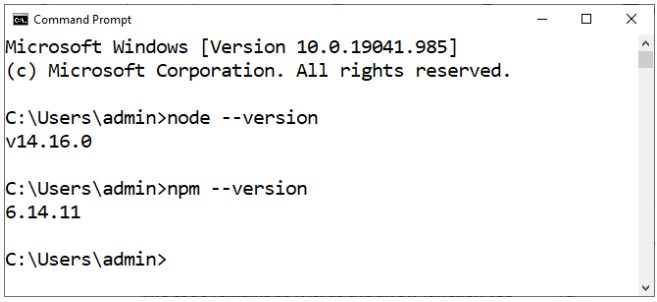
Tabela 4.1. Karakteristike jezika TS u odnosu na JS

TypeScript	JavaScirpt
Podržava objektno orijentisani koncept.	Skript jezik.
Podržava statičku tipizaciju.	Nema statičke tipizacije.
Podržava interfejse.	Ne podržava interfejse.
Ima podršku za opcione parametre.	Nema podršku za opcione parametre.
Podržava generičke tipove	Ne podržava generičke tipove.
Podržava module. Number, string,... su interfejsi.	Ne postoji podrška za module. Tipovi kao što su Number, string,... su objekti.

4.1 Instalacija i pokretanje

Instalacija se može uraditi na više načina. U nastavku navodimo jedan od najčešćih postupaka instalacije preko **npm** paket-menadžera. Dakle, najpre se instalira Node paket, preuzimajući poslednju verziju za korišćeni operativni sistem sa zvaniče lokacije: https://nodejs.org/en/download/. Detaljne instukcije, kako preuzeti i instalirati NodeJs nalaze se na veb stranici: https://www.guru99.com/download-install-node-js.html.

Nakon istalacije Node paketa radi se proveriti da li su zaista Node, odnosno npm instalirani. Za proveru možete koristiti sledeće komade u okviru konzolnog tj. *cmd* prozora:



Slika 4.1. Provera instalirane verzije paketa Node, odnosno alata npm

Instalacija

Instalacija paketa za podršku jezika TypeScript izvodi se: globalno ili lokalno. Lokalna instalacija se vrši samo u folderu gde fajlovi prilagođeni za rad projekta čiji fajlovi su u tom folderu.

Instalacija koja je globalna, obavlja se u folderu gde je instaliran Node i omogućava primenu na više projekata.

```
>npm -g install typescript
```

Alternativno, lokalna instalacija se izvodi bez flega **–g**.

Provera da li smo uradili instalaciju dobro ili već postoji instalcija TS, kao i koja verzija je u pitanju, vrši se komandom:

```
>tsc --version
```

```
Version 4.3.2
```

Prevođenje koda

Nakon instalacije dobijamo mogućnost primene prevodioca tj. komande `tsc` kojom se prevodi TS kod u JS kod.

Na primer, instalirajmo novi projekat startovanjem komande:

```
>npm init
```

Zatim, u folderu projekta kreirajmo `src` folder i u tom folderu kreirajte fajl proba.ts, sa sledećim kodom:

```
function saberi(x:number, y:number) {
    return x+y;
}
let rez = saberi(111,222);
console.log(rez);
```

Prevođenje znači kreiranje novog fajla sa JavaScript kodom, na osnovu TypeScirpt koda. Za prevođenje koristi se sledeća komanda:

```
>tsc proba.ts
```

Ukoliko je TypeScript instaliran globalno, komanda `tsc` je dostupna u svim folderima. Ukoliko je TS instaliran lokalno, potrebno je navesti putanju do tsc komade koristeći node_modules folder, na primer:

```
>node_modules/typescript/bin/tsc proba.ts
```

Nakon prevođenja dobija se JS fajl. U ovom slučaju dobija se fajl `proba.js`:

```
function saberi(x, y) {
    return x + y;
}
var rez = saberi(111, 222);
console.log(rez);
```

Nakon dobijenog JS fajla, može se uraditi njegovo testiranje. Izvršavanje se izvodi primenom komande:

```
C:\VETS\TypeScript\test1\src>node proba.js
```

Verzije JS koda

TypeScript podržava sve ECMAScript karakteristike, odnosno standarde JavaScript jezika, pa se o tom važnom detalju može voditi računa tokom kodovanja odnosno prevođenja.

Nisu sve nove karakterisike JavaScript-a podržane od svih, naročito ne od starijih veb čitača. Zato je ponekada potrebno vršiti prevođenje u specifične verzije ECMAScript tj. JavaScript-a.

Na primer, ako je naš TypeScript kod napisan koristeći noviju sintaksu, na primer sintaksu sa strelicama:

```
var saberi = (a, b) => {
    return a+b;
}

console.log(saberi(10, 20));
```

Prevođenjem u noviju JS verziju, na primer ES6, vrši se sledećom komandom:

```
tsc --target ES6  test.ts
```

ili

```
tsc -t ES6 test.ts
```

A dobija se JS fajl koji koristi sintaksu sa strelicama pošto je za tu verziju ona važeća:

```
var saberi = (a, b) => {
    return a + b;
};
console.log(saberi(10, 20));
```

Poredeći ovaj kod sa podrazumevanom ciljanim standardom, ES3, razlika je više nego očigledna. U podrazumevanom prevođenju dobija se:

```
var saberi = function (a, b) {
    return a + b;
};
console.log(saberi(10, 20));
```

4.2 Promenljive

Promenljive se koriste za skladištenje vrednosti. Skladištene vrednosti mogu biti tipa: string, number, boolean ili izraz. Promenljive tj. tipovi promenljivih u TS su slične sa JS. Da bi se jedna promenljiva koristila mora biti deklarisana. Tri načina deklarisanja promenljivih postoje: **var**, **let**, **const** .

Promenljive deklarisane pomoću var

Pogledajmo sledeći primer primene promenljivih deklarisanjem koristeći ključnu reč var.

```
var  g = 10; // promenljiva g je globalna

function proba() {
    var l = 1; // promenljiva l je lokalna
    return g++;
}

proba(); alert(g); // g = 11
proba(); alert(g); // g = 12
alert(l); // greška – l nije definisano
```

Promenljive deklarisane pomoću *let* i *const*

Razlika u promenljivama koje su deklarisane sa `let` u odnosu na var je značajna, mada na prvi pogled ne izgleda tako. Ta značajna razlika se ogleda **u oblasti definisanosti**. Na primer:

```
let g = 1;
function proba() {
    if (g > 0) {
      let x = 1;
    }
    return x;
}
proba(); // greška: x nije definisan.
```

Ovaj primer nije moguće prevesti. Sadrži grešku u korišćenju promenljive x. Ukoliko bi se promenljive deklarisale sa `var` greška ne bi postojala. Promenljive definisane sa `let` imaju oblast važenja (eng. scope) unutar bloka gde je promenljiva deklarisana. Tako je promenljiva x važeća samo u okviru bloka naredbe `if`, a ne u celoj funkciji.

Dakle, ako su promenljive deklarisane sa let unutar: neke funkcije, neke petlje (`for` ili `while`), `switch` bloku, ili drugom bloku koda, onda su one dostupne unutar istog bloka i ne postoji mogućnost njihove upotrebe van tog bloka. Ovo je ključna razlika promenljivih definisanih sa `var` u odnosu na `let`.

Ključnom reči `const` definišu se konstantne promenljive. Kada se kaže reč promenljiva, misli se na pojam koji smo definisali na početku poglavlja tj. imenovane podatke, a ne da su obavezno i izmenljivog sadržaja. Dakle, `const` promenljive su slične sa `let` promenljivama sa jednom ključnom razlikom. Jednom definisana vrednost se ne može menjati.

```
const pozdrav = "Zdravo";
pozdrav="Hallo"; // greška
```

4.3 Tipovi

TypeScript je programski jezik striktno definisanih tipova. To ga čini bitno drugačijim od JavaScript jezika. Na primer, u JS kodu, ako je definisana promenljiva čija vrednost je string, onda je bilo moguće promeniti ovu vrednost u drugi tip, na primer number. Međutim, ovo nije moguće u TypeScript-u. Naime, promenljiva kojoj se dodeli string se definiše kao string promenljiva od samog početka i pri pokušaju da joj se dodeli vrednost drugog tipa javlja se greška, tačnije izuzetak. Ovo se događa u fazi prevođenja, dakle ovakav kod nije moguće prevesti u JavaScript.

```
var pozdrav = "Zdravo";
pozdrav=22; // greška u TS, ne i u JS
```

Osnovi tipovu u TypeScript jeziku su:

- number,
- string,
- boolean,
- any,
- void.

Tip *number*

Ovaj tip se koristi za definisanje:

- celih brojeva,
- brojeva sa pokretnim zarezom, odnosno
- razlomaka.

Sintaksa je sledeća:

```
let a :number = 111;
let b :number = 22.2;
```

Neke važne metode koje se mogu koristiti za promenljive brojevog tipa su:

- `toFixed()` – konverzija broja u string sa zapisom sa fiksnom tačkom. Broj decimalnih mesta se može zadati kao argument.
- `toString()` – konvertuje broj u string.
- `valueOf()` – vraća vrednost promenljive.
- `toPrecision()` – formatira se broj sa specifičnom dužinom tj. ukupnim brojem cifara ispred decimalne tačke i iza, ako je dovoljno za prikaz.

Na primer:

```
console.log((8.8 + 2.5).toFixed(1));        11.3
console.log((8.8 + 2.5).toFixed(3));        11.300
console.log((8.8 + 2.5).toPrecision(1));    1e+1
console.log((8.8 + 2.5).toPrecision(3));    11.3
```

Tip *string*

Tip za definisanje i rad sa promenljivama tipa string. Na primer:

```
let str :string = "zdravo svete";
```

Uz ovaj tip sledi niz metoda za rad sa strigovima.
- `split`() – deli string na niz strinova koristeći separatore.
- `charAt`() – vraća karakter na određenoj poziciji.
- `indexOf`() – vraća poziciju prvog pojavljivanja zadatog stringa.
- `Replace`() – vrši nalaženje i zamenu navedenog stringa u stringu. Prvi argument je string u kome se vrši nalaženje i zamena, a drugi je string sa kojim se vrši zamena.
- `Trim`() –izbacuje praznine ispred i iza stringa.
- `substr`(), `substring`() – vraća podstring.
- `toUpperCase`() – konvertuje string u novi sa svim velikim slovima.

•`toLowerCase()` – konvertuje string u novi sa svim malim slovima.

Na primer:

```
let proba:string = "Novi Sad";
var a = proba.split(" ")
console.log(a[0]); //["Novi", "Sad"]
console.log(a[1]);
console.log(proba.charAt(5)); // S
console.log(proba.indexOf('S')); // 5

console.log(proba.replace("Sad","Beograd"));//"Novi Beograd"
let proba2:string = " Novi Sad   ";
console.log(proba2.trim()); //
console.log(proba.substr(5, proba.length)); //"Sad"
console.log(proba.substring(5, 8)); //"Sad"
console.log(proba.toUpperCase());//NOVI SAD
console.log(proba.toLowerCase());//novi sad
```

Tip *boolean*

Tip podataka za rad sa logičkim vrednostima: **true** odnosno **false**. Na primer:

```
let aa :boolean; // undefined
let bb :boolean = false;
if(!aa){
    console.log(aa); //undefined
    console.log(bb); //false
}
```

Tip *any*

Sintaksa:

```
let a :any = 222
a = "proba any tipa"; // promena tipa ne izaziva grešku.
```

Promenljive deklarisane tipom **any** mogu primati vrednosti različitih tipova, kao što su: `string`, `number`, `array`, `boolean` ili `void`. U prethodnom kodu,

TypeScript prevodilac neće prijaviti grešku ili izuzetak u navedenom slučaju, tj. ponašanje je slično kao u JavaScript-u.

Tip *void*

Tip `void` koristi se kao povratni tip funkcija koje ne vraćaju vrednost. Ove funkcije nemaju naredbu `return`, ili ako je imaju onda se koristi bez vrednosti: npr. `return;`. Sintaksa jedne funkcije je sledeća:

```
function prikaziVrednost(a:any, b:any, c:any):void{
    console.log("a="+a+" b="+b+" c="+c);
}
prikaziVrednost(22,"test", true);
```

Napomena: Zapazite da kod JavaScript funkcija, funkcija podrazumevano vraća `undefined`, čak i kada se vrednost ne vraća. Takođe, obratite pažnju da `void` nije isto što i `undefined`. `void` označava da funkcija ne vraća vrednost dok je `undefined` jedan literal odnosno vrednost koja može biti vraćena.

Tipovi za funkcije

JavaScript omogućava da se nekoj funkciji prosledi druga funkcija, kao što se prosleđuje neka vrednost. Kada je reč o strogo tipiziranom, objektnom jeziku TypeScript, moguće je definisanje tipova promenljivih koje se mogu koristi za prosleđivanje funkcija. Tipovi za funkcije su nalik potpisu same funkcije i pišu se na sledeći način:

```
let x: (arg1:tip1, arg2:tip2)=>tipPovratneVrednosti;
```

Tipovi koji su funkcije mogu se pisati bilo gde u kodu gde se mogu i koristiti. Ovi tipovi se prepoznaju po sintaksi koja sadrži male zagrade, ali je upotreba konzistentna sa ostalim tipovima.

```
let fja1: () => string[]; //fja vraca niz string-ova
let nizFja: (() => string)[]; //niz funk.koje vracaju string
```

Pogledajmo sledeći primer koji ilustruje prenos promenljive sa tipom funkcije.

```
const testNiz1 = [2, 2, 1, 3];
let val = sumaModifikovanihVrednosti(kvadrat,testNiz1)
console.log(val);

function sumaModifikovanihVrednosti(modifikator: (x: number) =>
number, niz:number[]) {
    let suma:number = 0;
    niz.forEach(element => {
        suma = suma + modifikator(element);
    });
    return suma;
}
function kvadrat(x: number) : number
{
    return x * x;
}
```

U prethodnom primeru fukcija `sumaModifikovanihVrednosti` prihvata kao prvi argument sve funkcije koje imaju definisani potpis, odnosno tip, tako da joj se može dinamički proslediti bilo koja funkcija definisanog potpisa.

Unija tipova

Pogledajmo sledeći izraz u JavaScript-u:

```
let x = Math.random() > 0.8 ? 2022 : "IST";
```

Postavlja se logično pitanje, kog tipa je promenljiva x?

To nije ni broj ni string, iako su to obe potencijalne vrednosti. Dakle, promenljiva x može biti broj ili string. Ova vrsta tipa naziva se „**unija**“. Tipovi unije su koncept koji nam omogućava da obrađujemo slučajeve koda u kojima ne znamo tačno koji je tip promenljive, ali znamo da je to jedna od dve ili više opcija.

TipeScript koristi sintaksu tj. karakter | (vertikalna crta) koja se umeće između mogućih vrednosti. Dakle, u prethodnom slučaju sintaksa bi bila

```
let x: string|number = Math.random() > 0.8 ? 32 : "A";
```

Dakle, to je tip koji se koristi da bi se obuhvatilo više određenih prostih tipova podataka, ali ne i bilo koji. To i jeste ključna razlika u odnosu na tip `any` . Stroga

tipizacija ostaje, samo se definisani tip menja u uniju dva ili više tipova. Na primer:

```
let skolskaGodina : number | string;
skolskaGodina = 2021; //ispravno
skolskaGodina = "2021/21"; // ispravno
skolskaGodina = { // greska
    g = 2021;
}
skolskaGodina = false; // greska
```

Dakle, promenljiva je i dalje tačno definisanog tipa, ali se taj tip definiše od dve, ili više, drugih tipova. Tip promenljive se dinamički određuje tokom izvršavanja koda, a u samom kodu se može odrediti primenom funkcije `typeof`, na primer:

```
typeof(skolskaGodina)
```

Ova funkcija bi vratila tip number odnosno tip string za prvi odnosno drugi slučaj ispravno dodeljene vrednosti promenljivoj.

TypeScript nudi i mogućnost definisanja **alijasa** za tipove. Alijas se definiše koristeći ključnu reč `type`. Alijas može biti upotrebljen i za unije, ali i za druge složene tipove, na primer one koji označavaju određenu vrstu tipova funkcija. Na primer:

```
type SkGod = number | string;
let skolskaGodina : SkGod;
skolskaGodina = 2021;
console.log(typeof(skolskaGodina)); // number
skolskaGodina = "2021/21";
console.log(typeof(skolskaGodina)); // string

type Potpis = (string) => void;
function potpis(text:string){
    console.log(`potpis: ${text}`);
}
function radSaPotpisom(arg1:string, arg2:Potpis){ // alijas kao
tip f.
    console.log("fja: radSaPotpisom");
    arg2(arg1);
};
radSaPotpisom("Petar Petrović", potpis);
```

Važno je da se za ovako definisane promenljive mogu koristiti funkcije koje su zajedničke za navedene tipove, dok nije moguća primena funkcija dostupnih samo za jedan tip. Na primer:

```
x.toString(); //ok
x.toLowerCase(); //error
x.toPrecision(2); //error
```

Takođe, u slučaju kada je definisana unija moguće je tzv. sužavanje tipova. Sužavanje se događa kada prevodilac na osnovu koda zaključi da je promenljiva specifičnijeg tipa od onoga kako je definisana. Jednom kada prevodilac odredi da je tip specifičniji nego što je u prethodnim linijama koda bilo, omogućava da se tretira kao taj specifični tip. Evo narednog primera:

```
let x: string | number;
x = "IST";
x.toString(); //ok
x.toLowerCase(); //ok
```

TypeScript može da odredi specifični tip podataka čak i u slučaju primene **`if`** naredbe. Ukoliko se u **`if`** naredbi vrši provera sa nekom promeljivom ili konstantom određenog tipa, uslov je zadovoljen, TypeScript nedvosmisleno „zna“ da je u pitanju promenljiva određenog tipa. Na primer:

```
let x: string | number;
if( x === "ist"){
    x.toLowerCase(); //ok
}
```

ili

```
typeof x == "number" ? x.toPrecision(2) : x.toLowerCase();
```

Opcija *`strictNullChecks`*

Za dobro razumevanje TypeScripta uvek je poseban izazov razumevanje tipova: **`null`** odnosno **`undefined`**. S tim u vezi važno je i razumevanje opcije **`strictNullChecks`**. Pogledajmo sledeći kod:

```
let x = Math.random() > 0.2 ? "IST" : undefined;
x.toLowerCase(); // ok
```

Kao što se vidi, slučajna vrednost može biti manja od 0.2 i u tom slučaju promenljiva x dobija vrednost `undefined`, pa bi poziv metode `toLowerCase()` izazvao grešku pri izvršavanju. Međutim, TypeScript ne označava kod kao pogrešan. Takođe, kod bi bio prepoznat kao korektan i kada bi umesto `undefined` stajalo `null`. Za one koji već imaju iskustva u C++, Java ili C# jeziku, znaju da je ovo moguće. TypeScript jezik nudi opciju u prevodiocu kojom se podešava da li je postavljanje `null` ili `undefined` vrednosti moguće ili ne. Postavljanjem ove opcije na `false` prethodni kod postaje neregularan. Pogledajte primer:

```
>tsc proba.ts --strictNullChecks
proba.ts:4:1 - error TS2531: Object is possibly 'null'.
4 x.toLowerCase();
```

Bez postavljanja ove opcije, svakom tipu podataka pridružuju se `undefined|null` tip. Sa druge strane, primenom ove opcije može se izbeći greška u izvršavanju zbog neočekivanih vrednosti tokom izvršavanja koje su najčešće `null`. U prethodnom kodu, ako bi slučajna vrednost bila ispod 0.2 odnosno ako bi x dobilo vrednost `undefined` onda bi nastupila greška:

```
x = undefined;
x.toLowerCase(); // error
```

Iz tog razloga mnogi programeri smatraju da je dobra praksa koristi opciju `strictNullChecks`.

Dakle, ako promenljiva nekog tipa može dobiti vrednost `null` ili `undefined`, u kodu se mora ispitati ta vrednost u toku izvršavanja tj. da li je moguće pozvati metode tog tipa. Evo i nekoliko predloga:

```
if(x){
    x.toUpperCase();  // opcija 1
}
x && x.toUpperCase(); // opcija 2
x?.toUpperCase();     // opcija 3
```

Naravno, ovo je efikasno samo u slučajevima kada neki tip podataka može biti `null` ili `undefined`. Međutim, ako se radi o dve mogućnosti koje ne obuhvataju `null` ili `undefined`, na primer `string` i `number`, onda ova provera nije korektna. Drugim rečima, morali bi da proverimo tip. Na primer:

```
if(typeof x === "string"){
    x.toUpperCase();
}
```

Dok druge dve opcije nisu moguće.

Tip *Array*

Nizovi predstavljaju promenljive koji sadrže više vrednosti istog tipa. Neke karakteristike nizova su:

- Deklaracijom niza vrši se alokacija uzastopnih memorijskih blokova. Svaki blok predstavlja jedan element niza.
- Jednom definisan niz ne može menjati svoju veličinu.
- Identifikacija elementa niza vrši se na osnovu jedinstvene celobrojne vrednosti koju nazivamo indeksom.
- Nizovi treba da se deklarišu pre upotrebe. Element niza se može promeniti, ali ne i brisati.

Sintaksa:

```
let array_name[:tipPodatka] = [val1,val2…valn]
```

Na primer:

```
var nizS:string[];
nizS = ["a","b","c","d"];
nizS.forEach(element => {
    console.log(element);
});
```

Jedan niz može da se kreira i primenom klase **Array**. U ovom slučaju, za definisanje niza koristi se konstrukor koji prima vrednost koja označava veličinu niza i listu elemenata niza. Elementi su razdvojeni zapetom.

Sintaksa je:

```
let niz : Array<type>;
```

Na primer:

```
//niz stringova
let dan11: Array<string> = ["Pon", "Uto", "Sre", "Čet", "Pet", "
Sub", "Ned"];
// niz elemenata dva mouguća tipa
let dan12: Array<string | number> = ["Pon", 1, "Uto", 2, "Sre",
3, "Čet", 4, "Pet", 5, "Sub", 6, "Ned", 7];
```

```
let dan21: Array<number> = [1,2,3,4,5,6,7];
let dan22: number[] = [1,2,3,4,5,6,7]; //isto kao dan21
let dan23: number[] = new Array<number>(7); //bez inicijal.

console.log(dan23);
console.log(typeof(dan23));
// greska u logici...
dan23.forEach((el, i)=>{
    el[i] = i*i;
    console.log(el[i])
})

for(let i=0; i<dan23.length; i++){
    dan23[i] = i*i;
    console.log(dan23[i])
}
```

Elementi u nizu su indeksirani počev od 0.

Implicitno određivanje tipa

Već smo se sreli sa implicitnim određivanjem tipa promenljive. Slično kao u JavaScript jeziku, pri prvoj dodeli vrednosti promenljiva dobija svoj tip, ukoliko nije ranije definisan. Tako dobijeni tip je ravnopravan sa unapred definisanim i ne može se vršiti promena drugim tipom. Na primer:

```
let a=1;
let b="test";
b = a; // greška
„Type 'number' is not assignable to type 'string'.“
```

Sada pogledajmo slučaj rada sa objektima koji se sastoje od nekoliko svojstava. Naročitu pažnju posvećujemo konverziji u tipove sa istim ili sličnim svojstvima.

```
var osoba1={
    ime:"Pera",
    prezime:"P"
}
var osoba2={
    prezime:"Ilic",
    ime:"Jova"
}
var student1={
    ime:"Pera",
    prezime:"P",
```

```
    indeks:"NRT-55/55"
}

osoba1 = osoba2; // ok
osoba2 = osoba1; // ok
student1 = osoba2; // Greška
osoba2 = student1; // ok
osoba1 = student1; // ok

//student1 = osoba2; // Greška
osoba1 = {   // ok
    ime:"Jova",
    prezime:"J"
};
// osoba1 = {   // Greška
//     ime:"Jova"
// };
```

Obratite pažnju da u poslednjem primeru nije bilo moguće napraviti novu ili promeniti postojeću osobu, **bez polja koja su učestvovala u definisanju prvog objekta tipa osoba**. Za ovakve tipove, ukoliko je potrebno obezbediti rad bez nekih polja, može se eksplicitno definisati definisati da su tipa **any**. Sedeći kod bi bio ispravno napisan:

```
let osoba:any = {   // ok
    ime:"Jova"
};
```

4.4 Petlje

TypeScript podržava većinu petlji koje su poznate u drugim jezicima. Njihova primena se lako povezuje u radu sa nizovima.

Pogledajmo najpre primenu petlje **for**.

```
// for
let dani: Array<string> = ["Pon", "Uto", "Sre", "Čet", "Pet", "S
ub", "Ned"];
for (let i=0;i<dani.length; i++) {
    console.log(dani[i]);
```

```
}
```

Druga petlja koja je podržana je **for-in** petlja. Ova petlja vraća listu ključeva objekta kroz koji se vrši iteracija. Na primer:

```
let dani2 =
{1:"Pon",2:"Uto",3:"Sre",4:"Čet",5:"Pet",6:"Sub",7:"Ned"}
for (let key in dani2) {
    console.log(`${key}: ${dani2[key]}`);
```

U slučaju primene petlje, iteracija se vrši preko ključeva niza. Na primer:

```
for (let i in dani) {
    console.log(`${i}: ${dani[i]}`);
}
```

Međutim, ova vrsta iteracije kroz niz ne garantuje sortiranost. Dakle, svi ključevi niza će biti obuhvaćeni, ali sortiranost nije definisana.

Sledeća petlja je **for-of**. Ova petlja iterira kroz objekte niza. Za verzije od es2015 vrši se prevođenje u for-of petlju, dok za ranije kao for petlja.

Verzija < es2015

```
for (var _i = 0, dani_1 = dani; _i
< dani_1.length; _i++) {
    var dan = dani_1[_i];
    console.log("" + dan);
}
```

Verzija >= es2015

```
for (let dan of dani) {
    console.log(`${dan}`);
}
```

Često primenjivana petlja je **forEach** petlja koja se primenjuje kao funkcija promenljive koja je tipa niz. Dakle, `niz.forEach()` prolazi kroz sve elemente niza, na primer:

```
dani.forEach(dan => {
    console.log(dan)
});
```

```
dani.forEach(function(dan){
    console.log(dan)
});
```

Petlje **while** odnosno **do...while** su slične istim petljama u drugim jezicima. Sledi primer ispisa po pet stringova koristeći ove dve petlje. Na primer:

TS

```
// while
var n:number = 5;
while (n > 0) {
  console.log("while");
   n--;
```

JS

```
// while
var n = 5;
while (n > 0) {
  console.log("while");
    n--;
```

```
}

// do...while
var m:number = 5;
do {
   console.log("do…while");
   m--;
} while (m > 0);
```

```
}

// do...while
var m = 5;
do {
    console.log("do…while");
    m--;
} while (m > 0);
```

Rad sa nizovima

Nizovi u TS poseduju veći broj metoda i svojstava koji se tiču rada sa nizovima. Većina tih metoda se sreće i u drugim programskim jezicima. Metode i svojstva se dobijaju navođenjem naziva metode sa zagradom odnosno naziva svojstva nakon tačke, npr: `objNiz.svojstvo` ili `objNiz.metoda()`. U nastavku slede važna svojstva i metode:

`length` – svojstvo koje sadrži broj elemenata niza.

`reverse` – metoda koja formira niz u obrnutom redosledu. Na primer:

```
let dani: Array<string> = ["Pon", "Uto", "Sre", "Čet", "Pet", "
Sub", "Ned"];
console.log(dani);
let daniR = dani.reverse();
console.log(daniR);
```

`sort` – metoda za sortiranje niza. Metoda **menja tekući niz**. Istovremeno taj sortiran niz se prihvata kao izlaz metode. Može se uraditi i specifično sortiranje primenom određene funkcije sortiranja.

```
let dani: Array<string> = ["Pon", "Uto", "Sre", "Čet", "Pet", "
Sub", "Ned"];
// standardno sortiranje
let daniS = dani.sort();
console.log(dani); //vazno!
console.log(daniS);
// sortiranje primenom specificne funkcije sortiranja
dani.sort(function(a:string, b:string){
```

```
    if (a[1] > b[1])
        return 1;
    else if (a[1] == b[1])
        return 0;
    else
        return -1;
})
console.log(dani);
```

pop – metoda za preuzimanje tj. izbacivanje **poslednjeg** elementa iz niza.

```
let dani: Array<string> = ["Pon", "Uto", "Sre", "Čet", "Pet", "Sub",
"Ned"];
let izbacen1:string = dani.pop();
let izbacen2:string = dani.pop();
console.log(izbacen1); //Ned
console.log(izbacen2); //Sub
console.log(dani);     //['Pon','Uto','Sre','Čet','Pet']
```

push – dodavanje novog elementa **na kraj** niza.

```
let dani: Array<string> = ["Pon", "Uto", "Sre", "Čet", "Pet", "
Sub", "Ned"];
let izbacen1:string = dani.pop();
console.log(dani);//['Pon','Uto','Sre','Čet','Pet','Sub']
dani.push(izbacen1);
console.log(dani); //['Pon','Uto','Sre','Čet','Pet','Sub','Ned']
```

shift – metoda za preuzimanje tj. izbacivanje prvog elementa unizu.

```
let dani: Array<string> = ["Pon", "Uto", "Sre", "Čet", "Pet", "
Sub", "Ned"];
let izbacen1:string = dani.shift();
console.log(izbacen1); // Pon
console.log(dani);//['Uto','Sre','Čet','Pet','Sub','Ned']
```

unshift – metoda za dodavanje prvog elementa unizu.

```
let dani: Array<string> = ["Pon", "Uto", "Sre", "Čet", "Pet", "
Sub", "Ned"];
let izbacen1:string = dani.shift();
dani.unshift();
```

concat – spajanje dva niza. Vraća novi niz. Na primer:

```
let radniDani: Array<string>=["Pon", "Uto", "Sre", "Čet", "Pet"];
let vikendDani: Array<string> = ["Sub", "Ned"];
let dani:Array<string> = radniDani.concat(vikendDani);
console.log(radniDani);
console.log(vikendDani);
console.log(dani);
```

Rezultat:

```
[ 'Pon', 'Uto', 'Sre', 'Čet', 'Pet' ]
[ 'Sub', 'Ned' ]
[ 'Pon', 'Uto', 'Sre', 'Čet', 'Pet', 'Sub', 'Ned' ]
```

4.5 Klase

Klasa predstavlja definiciju objekta, odnosno predstavlja tip podataka koji se koristi za instanciranje jednog ili više objekata. Klase postoje u JavaScript jeziku počev od standarda ES6. Pre tog standarda, klase su implementirane koristeći funkcije bazirane na prototipovima. Uvođenjem klasa omogućeno je pisanje koda nalik kodu u programskim jezicima: Java, C#, Python,... TypeScript je nadogradnja na JavaScript i poseduje mehanizam rada sa klasama, što ga čini veoma moćnim.

Osnovna sintaksa jedne klase je sledeća:

```
class nazivKlase {
    // svojstva klase

   constructor() {
    // telo konstruktora tj. inicijalizacija
   }

  // metode klase
}
```

Dakle, jednu klasu opisuju svojstva klase, a akcije su definisane metodama klase. Posebno izdvojena akcija je konstruktor koja postoji i u slučaju da se eksplicitno ne definiše, a tiče se kreiranja instanci klase.

Na primer, jedna konkretna klasa bila bi sledeća:

```
class Osoba {
    ime:string;
    prezime:string;
    id:number;

   constructor(ime:string, prezime:string, id:number) {
        this.ime = ime;
        this.prezime = prezime;
        this.id = id;
```

```
    }

    getName():string{
        return this.prezime + " " + this.ime;
    }
    getId():number{
        return this.id;
    }
}
```

Klasa se naziva `Osoba`. Ima svojstva `ime`, `prezime` i `id`. Takođe, klasa ima metode `getName()` odnosno `getId()`.

Važno. **Da bi se pristupilo svojstvima ili metodama jedne klase mora se upotrebiti ključna reč** `this`.

Konstruktor

Konstruktor je metoda kojom se vrši inicijalizacija svojstava prilikom kreiranje objekta klase, odnosno prilikom pozivanja klase. U prethodm primeru, konstruktor prima argumente pomoću kojih postavlja početne ili nepromenljive vrednosti jedne klase:

```
    constructor(ime:string, prezime:string, id:number) {
        this.ime = ime;
        this.prezime = prezime;
        this.id = id;
    }
```

Važno. Zapazite da se pristup do svojstava klase naglašava upotrebom ključne reči `this`. U našem slučaju, konstruktor prima promenljive istih imena kao polja klase. Upotreba samo naziva polja, bez ključne reči `this`, izaziva nejasnoću tj. grešku u logici mada ne i grešku u prevođenju.

U TypeScript-u postoji i skraćena verzija konstruktora. Ona omogućava da se specificiraju parametri upotrebom modifikatora pristupa direktno u konstruktoru. Modifikatori pristupa su obrađeni u nastavku ovog poglavlja . Prethodni primer se može iskoristiti da pokaže ovu osobinu tako što izbacimo polja klase, a uvedemo modifikatore pristupa u konstruktor. Na primer:

```
class Osoba {
    constructor(private ime:string, public prezime:string, protec
ted id:number) {
        this.ime = ime;
        this.prezime = prezime;
        this.id = id;
    }

    getName():string{
        return this.prezime + " " + this.ime;
    }
    getId():number{
        return this.id;
    }
}
```

Instanciranje

Kreiranje instaci klase tj. instanciranje, vrši se primenom operatora new. I ovo je nalik programskim jezicima C# ili Java. Nakon kreiranja instace klase moguće je pristupiti metodama i svojstvima objekta. Na primer:

```
let jova = new Osoba("Jovan", "Jovanović", 12345);
let neca = new Osoba("Nenad", "Nenadić", 54321);
```

Prevođenje

Uradićemo prevođenje bez dodatnog navođenja verzije, dakle sa starijim standardom za JavaScript i sa novim.

Nakon prevođenja, primenom komande ***tsc primer.ts*** dobija se:

```
var Osoba  = /** @class */ (function () {
    function Osoba(ime, prezime, id) {
        this.ime = ime;
```

```
        this.prezime = prezime;
        this.id = id;
    }
    Osoba.prototype.getName = function () {
        return this.prezime + " " + this.ime;
    };
    Osoba.prototype.getId = function () {
        return this.id;
    };
    return Osoba ;
}());
var jova = new Osoba("Jovan", "Jovanović", 12345);
var neca = new Osoba("Nenad", "Nenadić", 54321);
```

Što je drugačije u odnosu na prevođenje za ES6, ***tsc --target ES6 primer.ts***. U ovom slučaju dobija se:

```
class Osoba {
    constructor(ime, prezime, id) {
        this.ime = ime;
        this.prezime = prezime;
        this.id = id;
    }
    getName() {
        return this.prezime + " " + this.ime;
    }
    getId() {
        return this.id;
    }
}
let jova = new Osoba("Jovan", "Jovanović", 12345);
let neca = new Osoba("Nenad", "Nenadić", 54321);
```

Nasleđivanje

Osobina nasleđivanja je jedna od ključnih osobina objektno-orijentisanih jezika. TypeScript koristi ključnu reč `extend` za nasleđivanje. Na primer, ako klasa `B` nasleđuje klasu `A` piše se:

```
class B extends A {
    // svojstva klase B

       constructor() {
        // inicijalizacija B
       }

    // metode klase B
}
```

Na ovaj način klasa `B` sadrži sve osobine klase `A`, tj. nasleđuje ih. Pogledajmo pojednostavljem primer dve klase, u kome ne postoji konstruktor izvedene klase:

```
class A {
    constructor(public ime: string) {}
}

class B extends A {}

const instanca = new B("Zoran");
console.log(instanca.ime); // Ispisuje se: Zoran
```

Zapaža se da se u ovom primeru koristi konstruktor roditeljske klase. Dakle, ovo je **slučaj korišćenja podrazumevanog konstruktora**.

Napravimo još jedan konkretan primer. Kreirajmo klasu `Student` izvedenu iz klase `Osoba`. Jedan student ima svojstva: `ime`, `prezime`, `id` i `studentId`. Prva tri svojstva su zajednička sa svim osobama tj. pripadaju klasi `Osoba`. Poslednje svojstvo `studentId`, je dodatno svojstvo koje je specifično samo za studente. Kod izgdeda ovako:

```
class Osoba {
   constructor(public ime:string, public prezime:string, private
id:number) {
        this.ime = ime;
        this.prezime = prezime;
        this.id = id;
   }
}
class Student extends Osoba {
    studentId:string;
```

```
    constructor(ime:string, prezime:string,
                id:number, studentId:string) {
        super(ime, prezime, id);
        this.studentId = studentId;
    }

    getStudentId():string{
        return this.studentId;
    }
}
```

Takođe, i u ovom slučaju, inicijalizacija objekta klase `Student` mora istovremeno da obuhvati inicijalizaciju objekta `Osoba`. Konkretno radi se o inicijalizaciji svojstava. Dakle, konstruktor klase `Student` mora da poziva konstruktor klase `Osoba` i to radi direktno preko metode `super(ime, prezime, id)`.

TypeScript ne podržava koncept višestrukog nasleđivanja. Višestruko nasleđivanja bi podrazumevalo da jedna klasa može istovremeno da nasledi više drugih klasa. Ono što je moguće je višestruko nasleđivanje interfejsa, što bi mogli drugačije da nazovemo implementacijom više interfejsa.

Modifikatori pristupa

TypeScript podržava sledeće modifikatore pristupa: `public`, `private` odnosno `protected`. Podrazumevano, ako modifikator pristupa nije naveden smatra se da je `public`.

Svojstva i metode koja su označena kao `public` dostupni su preko instance objekta.

Svojstva i metode koje su `private`, dostupne su samo u klasi u kojoj se koriste. Naglašavam, da dostupnost ne znači i postojanje istih već samo mogućnost da se sa njima pristupa van klase. Drugim rečima, objekti izvedenih klasa sadrže sve osobine roditeljskih, pa samim tim i sve što je unutar roditeljskih klasa definisano, poput privatnih metoda i svojstava, ali nije moguće upravljati sa njima preko objekata ili izvedenih klasa.
Pogledajmo ponovo klasu `Osoba` koja koja je proširena sa dve metode. Za dobro razumevanje narednog primera treba dodati da je podrazumevani modifikator

pristupa `public`. U slučaju primene modifikatora `protected`, moguć je pristup u izvedenim klasama, ali ne i preko objekata klasa.

```
class Osoba {
    ime:string;
    prezime:string;
    private id:number;

   constructor(ime:string, prezime:string, id:number) {
        this.ime = ime;
        this.prezime = prezime;
        this.id = id;
   }

   getName():string{
       return this.prezime + " " + this.ime;
   }
   protected getId():number{
       return this.id;
   }
}
```

Što se tiče klase `Student`, ona nasleđuje klasu `Osoba` i dodaje svojstvo `studentId`. Dodate su i metode koje koriste svojstva definisana u roditeljskoj klasi – `print`, odnosno svojstva definisana u sopstvenoj klasi – `getStudentId`.

```
class Student extends Osoba {
    studentId:string;

    constructor(ime:string, prezime:string, id:number, studentId
:string) {
        super(ime, prezime, id);
        this.studentId = studentId;
    }

    getStudentId():string{
        return this.studentId;
    }

    print():void{
        // this.id - nije moguće koristiti
        console.log(this.getId() + ": " + this.getName());
        // ali je moguće javna svojstva
        console.log(this.getId() + ": " + this.prezime + " " + t
his.ime);

    }
}
```

Ključna reč *Readonly*

Polja jedne klase se mogu označiti ključnom reči `Readonly`. Ova ključna reč označava da se vrednost ne može menjati pošto je podešena. Dakle, vrednost se može podešavati samo na jednom mestu, a to je metoda `constructor`. Na primer:

```
class Osoba {
    Readonly sifra:string;
        constructor(private  ime:string,  public  prezime:string,
            protected id:number) {
        this.ime = ime;
        this.prezime = prezime;
        this.id = id;
         this.sifra = "2388-2222-..."; // samo u konstruktoru
}
```

Metode *get/set*

TypeScript nudi rad sa parom get/set metoda. Radi se o konceptu gde se pristup do jednog polja klase zamenjuje sa dve metode. Jedna metoda se koristi kada se čita polje klase, a druga kada se vrednost zapisuje.

Ovakav koncept iziskuje poštovanje određene nomenklature. Najpre, polje koje čuva podatak obično ima privatni modifikator pristupa, a ispred naziva stoji donja crta. U primeri koji se navodi u nastavku, to je polje `_name`. Na primer:

```
class Osoba {
    private _name:string;

    get name(){
        return this._name.toUpperCase();
    }
    set name(value:string){
        this._name = value;
    }
}

let o1:Osoba = new Osoba();
o1.name = "Jova";
console.log(o1.name);
```

Upotreba polja `_name` je kontrolisana parom metoda: `get name()` odnosno `set name(value:string)`. Na ovaj način se obezbeđuje kontrolisani pristup koristeći samo naziv `name` kao što bi bio slučaj kada se radi sa bilo kojim svojstvom. Međutim, u pozadini se izvršava `get` metoda ako se vrednost čita, odnosno `set` metoda ako se vrednost menja.

Prevođenje zahteva primenju ciljane platforme od ES5 inače se dobija greška

„Accessors are only available when targeting ECMAScript 5 and higher.“

Dakle, nakon prevođenja: `> tsc proba.ts --target ES5`

Odnosno pokretanja`:> node .\proba.js`

Dobija se: JOVA

Statičke funkcije i svojstva

Statičke funkcije klase se mogu koristiti bez kreiranja instanci klase. Dakle, radi se o jednoj vrsti globalnih funkcija. Po svojoj prirodi odnose se na akcije koje koriste objekat tj. stanje objekta.

Statička svojstva su svojstva koja sadrže istu vrednost za sve instance klase u kome su statička. Pogledajte naredni primer.

```
class Osoba {
    static count = 0;
    name:string;

    constructor(name:string){
        this.name = name;
        Osoba.count ++;
    }

    static line():void{
        console.log('------------------------');
    }
    static printCount():void{
        this.line();
        console.log(Osoba.count);
    }
}

let o1:Osoba = new Osoba('Jova');
let o2:Osoba = new Osoba('Pera');
```

```
Osoba.printCount();
Osoba.line();
```

Kao što se vidi, definisano je statičko polje `count` koje je namenjeno brojanju instanci klase `Osoba`. Takođe, postoji i statička metoda `line()` čija je namena da ispisuje samo liniju, nezavisnu od stanja objekta klase. Jasno je da jedna statička metoda može koristiti drugu kao i svojstva.

Apstraktne klase

Apstraktna klasa se može koristiti kao roditeljska klasa, ali se na osnovu nje ne mogu kreirati instance. Apstraktna klasa se definiše koristeći ključnu reč `abstract`. Dakle, apstraktne klase su namenjene za nasleđivanje, tj. omogućuju da druge klase budu izvedene iz njih.

Apstraktna klasa obično uključuje jednu ili više apstraktnih metoda ili deklaracija svojstava. Klasa koja nasleđuje tj. proširuje apstraktnu klasu mora da definiše, tj. da konkretizuje, sve apstraktne metode.

Na primer, apstraktna klasa `Osoba` deklariše: jednu apstraktnu metodu `test(string)`, jedno apstraktno svojstvo `jmbg` kao i standardnu metodu `ispis()` odnosno standardno svojstvo `naziv`.

```
abstract class Osoba {
    naziv: string;
    abstract jmbg: string;

    constructor(naziv: string) {
        this.naziv = naziv;
    }
    ispis(): void{
        console.log(this.naziv);
    }
    abstract test(string): void;
}

class Student extends Osoba {
   indeks: string;

   constructor(naziv: string, indeks: string, jmbg: string){
     super(naziv);
```

```
        this.indeks = indeks;
        this.jmbg = jmbg;
    }

    test(opis:string): void {
        console.log("*** " + opis);
        this.ispis();
    }
}

let student1: Osoba = new Student("Pera Perica", "nrt-1/1",
"1234512345123");
student1.ispis();
student1.test("proba");
```

Klasa `Student` koja nasleđuje apstraktnu klasu mora da uradi inicijalizaciju pozivom metode `super(...)` u konstruktoru. Takođe, zapazite da **apstraktna klasa može da uključi i apstraktna svojstva**.

4.6 Interfejsi

Interfejsi su jedan od najvažnijih koncepata tipičan za objektno programiranje, a ima veliku primenu u TypeScript jeziku. Interfejsi definišu **metode i svojstva** koja se moraju implementirati. Implementacija se odnosi na klase koje implementiraju, u nekim jezicima koriste takođe i reč nasleđuju, interfejse. Dakle, interfejsi definišu pravila koja se mogu generalizovati za klase, funkcije odnosno promenljive. Jedina metoda koju interfejs ne može imati je **konstruktor.**

Jedan interfejs se može kreirati od svojstava i metoda. Uz to, navedena svojstva ili metode mogu biti **opcioni**. Ako su opcioni, tada se navode koristeći oznaku „**?**“. Istovremeno, primena interfesa obezbeđuje strogu proveru tipova funkcija, promenljivih ili klasa koje implementiraju interfejs.

Sintaksa

Sintaksa je usklađena sa definisanjem tipova i liči na interfejse u drugim jezicima. Pogledajmo definiciju interfejsa `ILine`:

```
interface ILine{
    x1:number;
    y1:number;
    x2:number;
    y2:number;
}
```

Definisan interfejs `ILine` ima svojstva `x1`, `y1`, `x2`, `y2`. Sva navedena svojstva su obavezno tipa `number`.

Ovako definisan interfejs može biti implementiran:

- u nekoj promenljivoj,
- funkciji ili
- klasi.

Na primer, ako ovaj interfejs implemeniramo u promenljivoj `line`, kod bi bio sledeći:

```
let line : ILine = {
    x1:0,
    y1:0,
    x2:10,
    y2:10
}
```

Zapaziti da su u promenljivoj `line` sva svojstva obavezna i moraju odgovarati nazivom i tipom navedenom u interfejsu `ILine`.

Prevođenje ovog koda dobija se jednostavno definisan objekat sa svojstvima interfejsa, dakle:

```
var line = {
    x1: 0,
    y1: 0,
    x2: 10,
    y2: 10
};
```

Drugi slučaj implementacije interfejsa je u funkciji. Pogledajmo sledeći primer gde se interfejsom definiše povratni tip:

```
function getLine(a1:number,a2:number,a3:number,a4:number) : ILine
{
    return {
```

```
        x1:a1,
        y1:a2,
        x2:a3,
        y2:a4
    }
}
```

Interfejs `Iline` se koristi za definisanje povratne vrednosti funkcije `getLine`(). Dakle povratna vrednost mora sadržati sva svojstva i odgovarajuće tipove kao i interfejs. Slično se može izvesti preko definisanje lokalne promenljive, na primer:

```
function getLine(a1:number,a2:number,a3:number,a4:number) : ILine
{
    let rez:ILine;
    rez.x1;
    // ...
    return rez;
}
```

Prevođenjem u JavaScript kod dobija se sledeće:

```
function getLine(a1, a2, a3, a4) {
    return {
        x1: a1,
        y1: a2,
        x2: a3,
        y2: a4
    };
}
```

U slučaju da povratni tip podataka ne odgovara interfejsu biće detektovana greška odnosno generisan izuzetak.

Pogledajmo i poslednji primer kojim se pokazuje da interfejs može biti iskorišćen i za opis ulaznih vrednosti neke funkcije.

```
let line : ILine = {
    x1:0,
    y1:0,
    x2:10,
    y2:10
}
function lineLength(line:ILine) : number{
 return Math.sqrt((line.x1-line.x2)*(line.x1-
   line.x2) + (line.y1-line.y2)*(line.y1-line.y2));
 }
console.log(lineLength(line))
```

Dakle, u prethodnom primeru interfesom smo definisali ulaznu vrednost za funkciju `lineLength(line:ILine)`. Funkcija na taj način prihvata tačno definisan tip podataka i generiše brojčanu vrednost koja se računa koristeći svojstva ulazne promenljive tipa `ILine`. Konkretno, funkcija računa dužinu linije, a ulazna vrednost je jedna definisana linija.

Prevođenjem, dobija se sledeći kod:

```
var line = {
    x1: 0,
    y1: 0,
    x2: 10,
    y2: 10
};
function lineLength(line) {
    return Math.sqrt((line.x1-line.x2)*(line.x1-line.x2) +
        (line.y1-line.y2)*(line.y1-line.y2));
}
console.log(lineLength(line));
```

Implementacija

Kada je reč o klasi, interfejs se implementira tj. realizuje uz korišćenje ključne reči `implements`. Dakle, sinataksa je sledeća:

```
class NazivKlase implements NazivInterfejsa {
       // ...
}
```

Napisaćemo klasu na osnovu nešto modifikovanog interfejsa iz prethodnog primera `ILine`. Interfejs ima svojstva i metode. Klasa koja implementira interfejs mora da sadrži navedena svojstva i da ima realizovanu metodu koja je navedena u interfejsu, `getLength()`. Na primer:

```
interface ILine{
  x1:number;
  y1:number;
  x2:number;
  y2:number;
  getLength():number
}
class Line implements ILine {
```

```
  x1:number;
  y1:number;
  x2:number;
  y2:number;

  constructor(a1:number, a2:number, a3:number, a4:number){
        this.x1 = a1;
        this.y1 = a2;
        this.x2 = a3;
        this.y2 = a4;
  }
  getLength():number{
    return Math.sqrt((this.x1-this.x2) * (this.x1-
     this.x2) + (this.y1-this.y2) * (this.y1-this.y2));
    }
}
```

Kompajliranjem dobija se sledeći JavaScript kod:

```
var Line = /** @class */ (function () {
    function Line(a1, a2, a3, a4) {
        this.x1 = a1;
        this.y1 = a2;
        this.x2 = a3;
        this.y2 = a4;
    }
    Line.prototype.getLength = function () {
        return Math.sqrt(
              (this.x1 - this.x2) * (this.x1 - this.x2) +
              (this.y1 - this.y2) * (this.y1 - this.y2));
    };
    return Line;
}());
```

4.7 Funkcije

Funkcije objedinjuju skup instrukcija koje obavljaju jedan zadatak. Aplikacije koje se pišu u JavaScript jeziku pišu se najvećim delom primenom funkcija. U jeziku

TypeScript postoje: klase, interfejsi, moduli, imenski prostori, ali su funkcije i u TS izuzetno važne. Za razliku od JavaScript funkcija, funkcije pisane u TypeScript jeziku poštuju pravila tipizacije odnosno objektno orijntisanih jezika. U tom smislu funkcije u TypeScript-u definišu tipove za ulazne argumente, odnosno za tip za vrednost koju vraćaju.

Dakle, TypeScript funkcija:

```
function saberi(x: number, y:number) : number{
    return x + y;
}
```

Kompajliranjem postaje JavaScript funkcija:

```
function saberi(x, y) {
    return x + y;
}
```

Parametri funkcije su `x`, odnosno `y`. Takođe, definisano je da su ovi parametri tipa `number`. Dakle, regularni poziv ove funkcije bio bi:

```
console.log(saberi(44,55));
```

Naravno, ovo bi bilo ispravno i u JavaScript jeziku kao i u TypeScript-u.

Razlika se javlja ukoliko pokušamo da pozovemo istu funkciju, ali prosleđujući dva stringa umesto dva broja, na primer:

```
console.log(saberi("Zdravo ","svete"));
```

U slučaju JavaScripta, funkcija će biti regularno izvršena. Pri tome, funkciji će biti prosleđene dve vrednosti tipa string, a u funkciji će doći do „sabiranja“ tj. spajanja te dve vrednosti. U slučaju TypeScript-a, detektuje se greška i ovakav kod neće biti moguće kompajlirati.

Preklapanje funkcija

JavaScript je dinamički jezik koji omogućava da se pišu funkcije istog imena a različitog broja argumenata i tipa. Međutim ista logika ne važi kod TypeScript-a.

Na primer, dve funkcije istog imena, sa različitim brojem i tipom argumenata, ne mogu se preklapati:

```
// // nije korektno
// function saberi(a:number, b:number) : number{
//     return a + b;
// }
// function saberi(a:string, b:string, c:string) : string{
//     return a + b;
// }
```

Ipak, TypeScript nudi opciju preklapanja funkcija, ali realizovanu na drugačiji način. Naime, moguće je definisati jednu funkciju koja može biti pozivana na različite načine poštujući preklopljene potpise navedene funkcije.

U ovom primeru, implementacija fukcija koja prihvata različite tipove se piše pomoću tipa `any`, dok se ostale funkcije navode bez tela, odnosno samo uz različit potpis. Dakle, kod bi bio sledeći:

```
// korektno
function saberi(a:number, b:number) : number;
function saberi(a:string, b:string) : string;
function saberi(a:any, b:any) : any{
    return a + b;
}
```

Ovako napisana funkcija prihvata argumente koji su tipa `string`, odnosno `number`, ali ne i bilo koji drugi tip.

Dakle, implementacija više različitih potpisa mora biti komatibilna sa potpisima na koje se odnosi.

Opcioni parametri i svojstva

U funkcijama JavaScript-a svi parametri su opcioni. Ukoliko se vrednost ne prosledi, parametar funkcije dobija vrednost `undefined`. Međutim, u slučaju TypeScript-a parametri funkcije su obavezni osim ako se eksplicitno ne navede drugačije. Sintaksa za opcione parametre je dodavanje znaka pitanja iza naziva parametra, na primer:

```
function saberi(x1: number, x2:number, x3?:number, x4?:number):n
umber{
    let zbir = x1 + x2;
    if (x3 != undefined)
```

```
        zbir = zbir + x3;

    if (x4 != undefined)
        zbir = zbir + x4;

    return zbir;
}
```

Važno. Opcioni parametri moraju biti definisani kao **poslednji parametri** u funkciji. Dakle, nije moguće da funkcija ima prvi parametar koji je opcioni, a zatim neki koji je obavezan.

Interfejsi mogu na identičan način imati **opciona svojstva**. Pogledajmo primer jednog takvog interfejsa. Neka je dat interfejs `IOsoba`:

```
interface IOsoba{
    ime:string,
    srednjeIme?:string,
    preziime:string
}
```

koji ima jedno opciono svojstvo: `srednjeIme.` Zatim, koristeći ovako definisan interfejs definišu se dva objekta: `osoba1` odnosno `osoba2`. Prvi objekat nema opciono polje, a drugi ga ima. Dakle:

```
let osoba1:IOsoba={
    ime:'Jova',
    preziime:'Ilic'
}
let osoba2:IOsoba={
    ime:'Jovan',
    preziime:'Peric',
    srednjeIme:'Dragan'
}
```

Podrazumevane vrednosti

U gornjem primeru, upotreba opcionih parametara komplikuje kod u funkciji. Zato se opcioni parametri nekada koriste zajedno sa podrazumevanim vrednostima. To znači, ako se vrednost ne prosledi, onda se parametru dodeljuje

podrazumevana vrednost, a ne `undefined`. U ovom slučaju sintaksa podrazumeva izbacivanje znaka pitanja i tipa, jer je namera jedinstveno iskazana definisanjem podrazumevane vrednosti. Dakle, pojednostavljen kod bio bi sledeći:

```
function saberi(x1:number,x2:number,x3=0,x4=0):number{
    return x1 + x2 + x3 + x4;
}
console.log(saberi(1,2));
console.log(saberi(1,2,3));
console.log(saberi(1,2,3,4));
```

Kompajlirani kod izgleda:

```
function saberi(x1, x2, x3, x4) {
    if (x3 === void 0) { x3 = 0; }
    if (x4 === void 0) { x4 = 0; }
    return x1 + x2 + x3 + x4;
}
```

Vidi se da je u kodu izvršena provera definisanosti parametara i dodela vrednosti u tom slučaju. Generisani kod se uvek minimizuje i zato se koristi `void 0`, umesto `undefined` mada je značenje isto.

Sintaksta ostatka i proširenja – tri tačke

Počev od standarda ES7 JavaScript jezik nudi mogućnost skraćenog označavanja kopiranja između objekata koju nazivamo sintaksom ostatka i proširenja, nekada i sintaksom tri tačke (eng. *rest and sp**Read***).

```
let osoba1 = {
    ime:'Jova',
    prezime:'Ilić',
    id:123
}
let osoba2 = {...osoba1}
// prikaz kopiranih svojstava
for(const k in osoba2)
    console.log(osoba2[k])
```

```
Jova
Ilić
123
```

Objekat **osoba2** nastao je kopiranjem svih svojstava iz objekta **osoba1**, a to smo verifikovali prikazom u konzoli. Takođe, moguće je vršiti povezivanje različitih svojstava iz više objekata.

```
let osoba1 = {
    ime:'Jova',
    prezime:'Ilić',
    id:123
}
let skola = {
    nazivSkole : 'Petar Petrović'
}
let osoba2 = {...osoba1, ... skola}
```

```
Jova
Ilić
123
Petar Petrović
```

Međutim, ukoliko postoje istoimena svojtva, a vrši se spajanje, kao na primer u ovom slučaju:

```
let osoba1 = {
    ime:'Jova',
    prezime:'Ilić',
    id:123,
    naziv: "Jova Ilić"
}
let skola = {
    id:345,
    naziv : 'Petar Petrović'
}
let osoba2 = {...osoba1, ... skola}
```

```
Jova
Ilić
345
Petar Petrović
```

onda dolazi do preklapanja, odnosno istoimena svojstva kasnije navedenih objekata, preklapaju ona koja su već postavljena.

Kao i sa objektima, može se izvršiti i spajanje nizova primenom tri tačke. Neka su dva niza:

```
let nizA = ['a','b'];
```

```
let nizB = ['1','2'];
```

Lako se vrši spajanje primenom operatora ... :

```
nizA.push(...nizB);
```

Ukoliko jedna funkcija ima više opcionih parametara istog tipa, oni se mogu zameniti sa nizom pojedninačnih elemenata istog tipa i to upotrebom operatora tri tačke ispred naziva promenljive. Na ovaj način dobija se mogućnost da se pri upotrebi takve funkcije koriste pojedinačni elementi niza posebno.

Na primer:

```
function saberi(...args: number[]):number{
    let rez:number = 0;
    for(let a of args)
        rez = rez + a;
    return rez;
}
console.log(saberi(1,2));
console.log(saberi(1,2,3));
console.log(saberi(1,2,3,4));
```

Tri tačke se naziva i **parametrom destrukcije** pošto se može koristiti da se iz jednog niza izdvoji drugi niz ili da se niz ulaznih vrednosti zameni pojedinačnim elementima. Pogledajmo oba slučaja posebno.

```
let a:number, b:number;
let ostatakNiza;
[a, b, ...ostatakNiza] = [10, 20, "Jova", "Ilic", 45];
console.log(a); // 10
console.log(b); // 20
console.log(ostatakNiza); // ["Jova", "Ilic", 45]

let osoba;
({a, b, ...osoba} = {a: 10, b: 20, ime: "Jova", prezime: "Ilic",
 godine:45});
console.log(a); // 10
console.log(b); // 20
console.log(osoba); //{ime: "Jova", prezime: "Ilic", godine:45}
```

Notacija funkcija strelicom

Ova notacija je jedna od važnih karakteristika standarda ES6 i na raspolaganju je u TypeScript-u takođe. Notacija obezbeđuje skraćeni zapis funkcija i ujedno upotrebu funkcija bez naziva, tzv. anonimnih funkcija. Sintaksa je sledeća:

```
var nazivFunkcije = (argumentiFunkcije) => {

    // telo funkcije tj. kod

}
```

Pogledajmo primer upotrebe ovih funkcija u odnosu na standardnu notaciju.

Standardna notacija:

```
function start(pocetnaVrednost:number):void{
    setInterval(brojackaFunkcija, 1000);

    function brojackaFunkcija(){
        console.log(pocetnaVrednost);
        pocetnaVrednost = pocetnaVrednost + 1;
    }
}
start(200);
```

Notacija strelicom:

```
((pocetnaVrednost:number):void=>{
    setInterval(()=>{
        console.log(pocetnaVrednost);
        pocetnaVrednost = pocetnaVrednost + 1;
    },1000);
})(200)
```

Povratne funkcije

Povratne funkcije predstavljaju jednu od osnovnih karakteristika jezika JavaScript. TypeScript preuzima ovu karakteristiku u celosti, naravno pridružujući joj dodatne karakteristika tipizacije. Pogledajmo primer funkcije u JS

```
//JavaScript: povratna funkcija
function potpis(text){
    console.log(`potpis: ${text}`);
}

function radSaPotpisom(arg1, arg2){
    console.log("fja: radSaPotpisom");
    //if(typeof(arg2) === "function")
    arg2(arg1);
};
radSaPotpisom("Petar Petrović", potpis);
radSaPotpisom("Petar Petrović", "potpis"); //greška osim ako se
uključi uslov
```

Prebacivanje u TypeScript obezbeđuje ne samo proveru tipova prostih promenljivih, već i povratnih funkcija. Zato prethodna greška biva otkrivena još u fazi prevođenja, a kod bi izgledao ovako:

```
// povratna funkcija
function potpis(text:string){
    console.log(`potpis: ${text}`);
}

function radSaPotpisom(arg1:string, arg2:(inarg:string)=>void){
    console.log("fja: radSaPotpisom");
    arg2(arg1);
};
radSaPotpisom("Petar Petrović", potpis);
//radSaPotpisom("Petar Petrović", "potpis"); // greška pri prevo
đenju
```

Objekat *this*

Ključna reč **this** koristi se da označi tekući (nadređeni) objekat. Dakle, u zavisnosti gde se koristi ima adekvatno značenje. Ako se koristi u samom objektu, na primer:

```
let p = {
    ime: "Perica",
    toString : function() {
       return "ime: " + this.ime;
    }
};
console.log(p.toString());
```

onda označava sam objekat. Ako se koristi u funkciji označava nadređeni objekat. Na primer, ako smo napisali sledeći kod:

```
function knjiga(naslov, autor) {
    this.naslov = naslov;
    this.autor = autor;
    console.log(this);
    return this;
}
let k1 = knjiga("Pesme1", "Ršumović");
let k2 = knjiga("Pesme2", "Ršumović");
if(k1 === k2 && k2.naslov === 'Pesme2')
    console.log("striktno jednaki");
```

Funkcija `knjiga()` se poziva dva puta i oba puta se koristi objekat `this`, koji označava nadređeni objekat u kome se nalazi i sama funkcija. Zbog toga dolazi do preklapanja vrednosti i striktne jednaksti u proveri.

Na kraju, dolazimo do ključne razlike kada je reč o funkcijama napisanim u notaciji strelica u odnosu na standardnu. Pogledajte kod:

```
let kocka = true;
this.kvadrat = true;

console.log(this);          //{kvadrat: true}
console.log(kocka);         //true
console.log(this.kocka);    //undefined (upozorenje)
//console.log(kvadrat);     //greska
console.log(this.kvadrat);  //true

(()=>{
    console.log(this);          //{{kvadrat: true}}
    console.log(kocka);         //true
    console.log(this.kocka);    //undefined (upozorenje)
    //console.log(kvadrat);     //greska
    console.log(this.kvadrat);  //true
})();
function test(){
    console.log(this);          //[global]{...}
    console.log(kocka);         //true
    console.log(this.kocka);    //undefined
    //console.log(kvadrat);     //greska
    console.log(this.kvadrat);  //undefined
}
test();
```

Funkcije definisane sa strelicom nemaju sopstveni definisani this objekat, već ga dele sa roditeljskim objektom `this`.

Nabrojive liste

TypeScript ima podršku za primenu objekata koji su kolekcije povezanih vrednosti – tzv. nabrojive liste. Ovi objekti se označavaju sa `enum`. Treba reći da Javascript ne podržava nabrojive liste, dok većina programskih jezika, poput C#, Java, ili C++ imaju podršku za nabrojive liste.

Sintaksa:

```
enum NazivEnumListe {
    value1,
    value2,
    ..
 }
```

Na primer:

```
enum Dani {
    Ponedeljak, Utorak, Sreda,
    Četvrtak, Petak, Subota, Nedelja
}
```

U prethodnom primeru, definisana je nabrojiva lista tj. `enum` naziva `Dani`.
Vrednosti ove liste su:

`Ponedeljak`, `Utorak`, `Sreda`, `Četvrtak`, `Petak`, `Subota`, `Nedelja`.
Istovremeno, vrednosti liste su praćene numeričkim vrednostima počev od 0, sa inkrementom od 1.
Kompajliranje u JavaScript prevodi nabrojive liste u funkcije, na način da se obezbedi pristup po indeksu odnosno vrednosti. Ipak, jasnoća i elegantnost u kodu pokazuje prednost primene TypeScript-a. Preveden kod je:

```
var Dani;
(function (Dani) {
    Dani[Dani["Ponedeljak"] = 0] = "Ponedeljak";
    Dani[Dani["Utorak"] = 1] = "Utorak";
. . .
})(Dani || (Dani = {}));
```

Pogledajmo primer koji ilustruje vrednosti nabrojive liste koje se formiraju implicitno, kao i pristup pojedinim vrednostima liste.

```
enum Dani {
    Ponedeljak, Utorak, Sreda,
    Četvrtak, Petak, Subota, Nedelja
}

let d:Dani;
console.log("Dani.Ponedeljak:", Dani.Ponedeljak);// 0
console.log("Dani['Ponedeljak']:", Dani['Ponedeljak']);// 0
console.log("Dani[0]:",Dani[0]);//Ponedeljak

console.log('typeof(Dani):',typeof(Dani));//object
console.log('typeof(Dani.Ponedeljak):',typeof(Dani.Ponedeljak));
//number
console.log('typeof(Dani[0]):',typeof(Dani[0])); //string
```

Vrednosti u nabrojivoj listi

Ukoliko se eksplicitno ne definišu vrednosti u nabrojivoj listi, onda podrazumevano, vrednosti liste se dodeljuju počev od vrednosti 0, pa redom: 1,2… Postoji i opcija da se ove vrednosti eksplicitno dodele. Sledeći primer definiše način eksplicitnog definisanja vrednosti enum liste i zatim verifikaciju ručno definisanih vrednosti.

```
enum Dani {
    Ponedeljak=1, Utorak=2, Sreda=3,
    Četvrtak=4, Petak=5, Subota=6, Nedelja=7
}
console.log("Dani.Ponedeljak:",Dani.Ponedeljak);// 0
console.log("Dani['Ponedeljak']:",Dani['Ponedeljak']);// 0
console.log("Dani[0]:",Dani[0]);// undefined
console.log("Dani[1]:",Dani[1]);// Ponedeljak
```

Vrednosti koji se navode ne moraju ići redom. Ako idu redom, onda se mogu izostaviti, tj. redosled će biti automatski generisan. Ako ne idu redom, onda se moraju eksplicitno navesti.

```
enum Dani {
    Ponedeljak=1, Utorak=2, Sreda=3,
    Četvrtak=4, Petak=5, Subota=-6, Nedelja=-7
}
enum Dani2 {
```

```
    Ponedeljak=1, Utorak, Sreda,
    Četvrtak, Petak, Subota, Nedelja
}

console.log("Dani.Nedelja:",Dani.Nedelja);// -7
console.log("Dani['Ponedeljak']:",Dani['Ponedeljak']);// 1
console.log("Dani2.Utorak:",Dani2.Utorak);// 2
```

Deklarisanje elemenata **enum** liste string vrednostima

Moguće je definisati nabrojivu listu eksplicitnim navođenjem vrednosti koje su stringovi. Na primer, ako modifikujemo prethodni primer na sledeći način:

```
enum Dani {
    Ponedeljak="Pon", Utorak="Uto",
    // Sreda, // ako se ne navede broj ili  vr. biće greška
    Sreda=3,
    Četvrtak, Petak, Subota, Nedelja
}
console.log("Dani.Ponedeljak:",Dani.Ponedeljak);// Pon
console.log("Dani['Ponedeljak']:",Dani['Ponedeljak']);// Pon
//console.log(Dani.Pon);// Greska
console.log(Dani['Pon']);// undefined
```

Tip *tuple*

Sličan tip podataka javlja u drugim jezicima novijeg datuma. Radi se o *n-torci* više tipova ili o jednoj vrsti niza u kome se tačno zna koliko postoji elemenata i tačni tipovi na određenim pozicijama. Tuple tip je nepromenljiv niz. Na primer:

```
type t_rezultat = [string, number];
function testf(a: number, b:string):t_rezultat {
    return [b, a];
}
```

Ovaj tip podržava rest elemente kao i **Read**only, na primer:

```
type code1 = [string, boolean, ...number[]];
function testf(args: Readonly [string, number]) {
    // ...
}
```

Objekat *Math*

Math objekat omogućava rad sa matematičkim konstantama i metodama. Ovaj objekat se koristi kao statički, tj. ne kreira se objekat već se primenjuju metode i svojstva iz statičkog objekta Math.

Tabela 5.1. – Svojstva objekta Math

svojstvo	opis
`E`	Ojler-ov broj ili osnova prirodnog algoritma, 2.718.
`PI`	Broj Pi, 3.14159.
`LN2`	Vrednost prirodnog logaritma za broj 2 (0.693), odnosno za broj 10 (2.302).
`LOG2E, LOG10E`	Logaritam broja e, za logaritam sa osnovom 2, odnosno osnovom 10.
`SQRT2, SQRT1_2`	Kvadratni koren broja 2 (1.414), odnosno broja ½ (0.707);

Tabela 5.2. – Svojstva metoda Math

metoda	opis
`abs()`	Apsolutna vrednost.
`sin(),cos(),tan()`	Sinus, kosinus i tanges
`asin(),acos(),atan()`	Arkus sinus, arkus kosinus i arkus tanges.

`ceil(),round(),floor()`	Formira ceo koji je: manji, zaokružen kao bliži ili veći, za broj sa decimalnom tačkom.
`exp(),log()`	Eksponent sa osnovom Ojlerove konstante. Prirodni logaritam.
`min(),max()`	Vraća se: minimalan odnosno maksimalan broj od navedenih.
`pow()`	It returns base to the exponent power, that is, base exponent.
`random()`	Vraća pseudoslučajan broj između 0 odnosno 1.
`sqrt()`	Kvadratni koren nekog broja.

Primeri:

```
// ceil, round, floor
console.log("ceil(1.1)=" +Math.ceil(1.1) + " round(1.501)=" + Ma
th.round(1.501) + " floor(1.9)=" + Math.floor(1.9));
// ceil(1.1)=2 round(1.501)=2 floor(1.9)=1

// exp, log
console.log("exp(1)=" + Math.exp(1) + " exp(2)=" + Math.exp(2));
// exp(1)=2.718281828459045 exp(2)=7.38905609893065
console.log("log(exp(1))=" + Math.log(Math.exp(1)));
// log(exp(1))=1

// min, max
console.log(Math.min(2,3,-2,5)); // -2
console.log(Math.max(2,3,-2,5)); // 5

// pow, sqrt
console.log(Math.pow(2,10));  // 1024
console.log(Math.sqrt(1024)); // 32

// random
for(let i=1; i<10; i++){
    console.log(Math.random());
}
```

Napomena. Obratiti pažnju da se u funkcijama matematičke biblioteke često koriste parametri destrukcije. Pogledajmo sledeći primer:

```
let r1 = Math.max(1,2,1,3,2);
let niz = [1,2,1,3,2];
let r2 = Math.max(...niz);
```

Generički tipovi

Generički tipovi postoje da bi omogućili pisanje univerzalnog, ponovo upotrebljivog i tipiziranog (bezbednog) koda. Generičke tipove imaju i mnogi drugi jezici: C++, Java, C#, Go, Swift i drugi – svaki sa svojim specifičnostima.

Postojanje ovih tipova daje sledeće mogućnosti:

Ponovna upotreba koda – umesto da piše ista funkcija ili klasa više puta za različite tipove (`string`, `number`, `boolean`…), piše se jednom a koristi se sa bilo kojim tipom.

Tipiziranost – kompajler zna koji se tip koristi i sprečava greške (npr. nije moguće ubaciti `string` u listu brojeva).

Fleksibilnost – kod postaje šablon koji se prilagođava kontekstu.

Performanse – za razliku od rada sa „opštim tipom“ (`any` u TypeScript-u ili `Object` u Javi), generički tipovi ne zahtevaju kastovanje i proveru tokom izvršavanja, pa su brži i sigurniji.

TypeScript takođe ima opciju upotrebe generičkih tipova. Na osnovu šablonskog koda vrši se naknadno generisanje koda. Generisanje se precizira preko ulazne vrednosti koja je tip koji se prosleđuje pri generisanju.

Generički mogu biti:

- interfejsi,
- klase,
- apstraktne klase i
- funkcije.

Pogledajmo najpre način pisanja generičkih funkcija analizirajući sledeći:

```
function ispis<T,U>(d1:T, d2:U) {
    console.log(`d1:T (${d1}:${typeof(d1)}) ; d2:U (${d2}:${type
of(d2)})`)
}
```

Generička funkcija `ispis` poseduje dodatak o tipovima podataka koji se koriste u funkciji: `<T,U>`. Slično kao što je korišćeno u sintaksi definisanja nizova preko klase Array. Oznake `T,U` su **simboličke** oznake tipova podataka koji se koriste u generičkoj funkciji. **Sva pravila stroge tipizacije važe!**

Primena ovako napisane funkcije obavlja se analogno standardnim funkcijama uz obavezno navođenje konkretno korišćenih tipova u toj funkciji. Dakle:

```
let daniS: string[] = ["Pon", "Uto", "Sre", "Čet", "Pet", "Sub",
 "Ned"];
let daniN: number[] = [1,2,3,4,5,6,7];

for(let i=0; i<7; i++)
    ispis<number,string>(daniN[i], daniS[i])
```

Ukoliko pokušamo da promenimo funkciju `ispis` tako da primenimo neke operacije koje nisu moguće za neke tipove, prevodilac prijavljuje grešku. Ovo se događa s razlogom jer je generički tip obuhvatio sve tipove pa nije moguće ponuditi ono što bi moglo biti korisno samo u nekim situacijama. Nešto kasnije videćemo kako se rešava ovaj problem. Pre toga da pogledajmo rad sa drugim generičkim oblicima koda.

Slično generičkim funkcijama, pri definisanju interfejsa, navodi se jedan ili više tipova uz naziv interfejsa. Na primer, generički tipovi `T`, `U` u interfejsu `IDanUnedelji`:

```
interface IDanUnedelji<T,U>
{
    rbr: T;
    naziv: U;
}
```

Slično se kreiraju i klase:

```
class DanUnedelji<T,U>
{
    private rbr: T;
    private naziv: U;

    constructor(rbr: T, naziv: U){
        this.rbr = rbr;
```

```
        this.naziv = naziv;
    }

    display():void {
      console.log(`rbr=${this.rbr}, naziv=${this.naziv}`);
    }
}
```

Upotreba ovako definisane klase dosledno prati pravila upotrebe drugih klasa, naravno uz definisanje konkretnih tipova. Na primer, pri definisanju niz konkretnih objekata ove klase bilo bi:

```
let dani:DanUnedelji<number, string>[] = [
       new DanUnedelji(1,"Pon"),
       new DanUnedelji<number, string>(2,"Uto")
]
```

```
[LOG]: "rbr=1, naziv=Pon"
[LOG]: "rbr=1, naziv=Uto"
```

Pošto klase i interfejsi mogu naslediti druge klase odnosno interfejse, a takođe klase implementiraju interfejse, moguće su različite varijante primene generičkih interfejsa odnosno klasa.

Pre svega jedan interfejs može da nasledi drugi interfejs, pa se tip podataka kojim se implementira novi interfejs može vezati za neki prethodni. Ovo je korisna opcija kojom se dobija na efikasnosti primene i postepenom povećanju funkcionalnosti. Na primer, interfejs `IDanUnedeljiSaPrikazom` je generički, oslonjen na tip `T` koji je tipa drugog interfejsa tj. tipa koji nasleđuje interfejs `IDan`:

```
interface IDan{
    rbr:number;
    naziv:string;
}
interface IDanUnedeljiSaPrikazom<T extends IDan>
{
    dan:T;
    display():void;
}
```

Takođe, generički interfejs može da proširi drugi generički interfejs novim karakteristikama:

```
interface IDanUnedelji<T,U>
{
    rbr: T;
    naziv: U;
}
interface IDanUnedeljiSaPrikazom<T,U>
  extends IDanUnedelji<T,U>
{
    display(t: T, u: U):void;
}
```

Posebni generički tipovi

TypeScript ima na raspolaganju nekoliko predefinisanih korisnih generičkih tipova koji imaju specijalne namene.

1. **Partial<Type>**. Parcijalni generički tip omogućava kreiranje tipa, ali za koji važi da su sva svojstva opciona. Na primer:
```
interface IOsoba{
    ime:string;
    prezime:string;
    id:number;
}
function akcija(osoba1:IOsoba, osoba2:Partial<IOsoba>) {
    return{...osoba1, ...osoba2}
}
let o1 = {
    ime:"Perica",
    prezime:"P",
    id:666
}
console.log(akcija(o1,{id:777}))
```

2. **Required<Type>**. Obavezni generički tip omogućava kreiranje objekata kod kojih su sva svojstva obavezna. Po toj logici je suprotan od parcijalnog. Na primer:
```
interface IStudent{
    ime?: string;
    prezime?: string;
}
let o1: IStudent = {ime: "Perica"};
let o2: Required<IStudent> = {ime: "Perica"}; // Greska!
```

3. **Readonly<Type>**. Kreiranje objekata sa **Read**only svojstvima tj. svojstvima koja se mogu samo čitati.
```
interface IKnjiga{
    naslov:string;
}
```

```
let k1:Readonly<IKnjiga> = {
    naslov:"Pesme"
};
k1.naslov = "Priče"; // greska
```

4. **Record<Keys, Type>.** Kreira objekat čija su svojstva tipa **Keys**, a vrednosti tipa **Type**.

```
interface IOdsek {
  rbr: number;
  naziv: string;
  adresa?: string;
}

type odsekSkraceno = "viser" | "ict" | "vtts";

let odseci: Record<odsekSkraceno, IOdsek> = {
  viser: {rbr:1, naziv: "Visoka..."  },
  ict: {rbr: 2, naziv: "Inf...." },
  vtts: {rbr: 5, naziv: "Dizajnerska..."}
};

console.log(odseci.viser)
```

5. **Pick<Type, Keys>**. Konstruisanje novog tipa na osnovu navedenih svojstava **Keys**, obično kroz uniju stringova. Na primer:

```
interface IFigura{
    naziv:string;
    a:number;
    b:number;
    r:number;
    p():number;
}
let p1:Pick<IFigura,"a"|"b"|"p"|"naziv">={
    naziv:"Pravougaonik",
    a:10,
    b:20,
    p(){return this.a*this.b;}
}
console.log(p1.p())
```

6. **Omit<Type, Keys>**. Konstruisanje objekta na osnovu svih svojstava tipa Type izuzimajući navedena u Keys.
7. **Exclude<Type, ExT>.** Type je unija, a ExT su oni koji se isključuju. Na primer.

```
type T1 = Exclude< "id"|"ime"|"prezime"|"nadimak"|boolean,
"id"|"nadimak" >; // T1 = ime | prezime | boolean
```

8. **`Extract<Type, Union>`**. Rezultat je tip koji se presek Type i Union tipa.
9. **`NonNullable<Type>`**. Kreira tip koji isključuje **`null`** i **`undefined`**.
10. **`Parameters<Type>`**. Kreira se tuple tip podataka na osnovu tipa funkcije koji se prosleđuje. Na primer:

```
declare function f1(arg: { a: number; b: boolean }): void;
type tip1 = Parameters<() => number>;  // []
type tip2 = Parameters<(s: string) => boolean>; //
[s:string]
type tip3 = Parameters<typeof f1>; // [arg:{a: number; b:
boolean}]
```

11. **`ReturnType<Type>`**. Slično kao u prethodnom slučaju, samo što se vraća tip koji funkcija vraća.
12. **`InstanceType<Type>`**. Kreira se tip podataka na osnovu konstruktorske funkcije tipa. Na primer:

```
class Osoba {
    ime = "Jova";
    id = 1;
}
type tip1 = InstanceType<typeof Osoba>; // Osoba
type tip2 = InstanceType<any>;   // any
//type tip3 = InstanceType<string>; // greska
```

Generički tipovi predstavljaju jedan od najvažnijih mehanizama u TypeScript-u jer omogućavaju da se piše univerzalan, fleksibilan i bezbedan kod bez žrtvovanja performansi. Njihova snaga je u tome što kombinuju ponovnu upotrebu sa strogo definisanom tipiziranošću, čime se postiže balans između praktičnosti i sigurnosti.

- Ponovna upotreba – jednom napisana funkcija ili klasa može da radi sa različitim tipovima podataka.
- Tipiziranost – kompajler sprečava greške i garantuje konzistentnost.
- Fleksibilnost – kod se prilagođava kontekstu i postaje šablon.
- Performanse – nema dodatnog kastovanja niti proveravanja tipova u runtime-u.

U praksi, generički tipovi čine kod čitljivijim, održivijim i skalabilnijim, jer se isti koncept može primeniti na različite strukture podataka bez ponavljanja. Time se TypeScript približava jezicima poput C++ ili Jave, ali zadržava svoju jednostavnost i integraciju sa JavaScript ekosistemom.

Ukratko: Generici nisu samo alat za uštedu vremena, već i ključni mehanizam za pisanje modernog, robusnog i profesionalnog TypeScript koda.

4.8 Moduli

Moduli u TypeScript-u služe da se kod organizuje u manje, nezavisne celine koje imaju sopstveni prostor imena i ne utiču na globalni prostor. Samo ono što se eksplicitno eksportuje iz modula može da se koristi van njega, dok se ostatak čuva kao interna logika.

Standard ES2015 (ES6) je doneo koncept modula u JavaScript-u. Pre toga, programeri su koristili različite biblioteke i improvizacije.

TypeScript prati isti koncept – svaki fajl koji sadrži `import` ili `export` tretira se kao modul. Ako fajl nema te ključne reči, smatra se običnim skriptom i sve njegove promenljive završavaju u globalnom prostoru.

Ako se kreira kod u jednom fajlu, sve promenljive koje su definisane van metoda pripadaju globalnom prostoru. Ukoliko imamo više fajlova koji zajedno učestvuju u nekom projektu, dolazi do preklapanja i konflikata zbog zajedničkog globalnog prostora koji dele promenljive iz više fajlova.

Moduli se izvršavaju u sopstvenoj oblasti definisanosti, ne u globalnoj. To su celine koda zapisane u fajlovima. U njima se može eksplicitno navesti šta se od promenljivih ili funkcija eksportuje, odnosno importuje. Samo ono što se eksportuje je vidljivo van modula.

Moduli mogu da importuju druge module. Za importovanje se koristi modul za učitavanje (eng. Module loader). Najčešće se koriste *CommonJS* modul odnosno *RequireJS*.

Ukratko: moduli su temelj modernog JavaScript/TypeScript razvoja. Oni omogućavaju da se piše čist, organizovan i skalabilan kod, gde svaka datoteka ima jasno definisane granice i eksplicitno izlaže samo ono što je potrebno.

Primena modula

Moduli predstavljaju način da se kod organizuje u **samostalne celine** koje imaju sopstveni prostor imena. To znači da promenljive, klase ili funkcije definisane u jednom modulu ne „cure“ u globalni prostor i ne izazivaju konflikte sa kodom iz drugih fajlova.

U praksi, svaki fajl može da bude modul. Ako u njemu definišemo klasu ili promenljivu, one su dostupne samo unutar tog fajla – osim ako ih eksplicitno označimo ključnom reči `export`. Tek tada mogu da se koriste u drugim fajlovima pomoću ključne reči `import`.

Pogledajmo sledeći primer bez modula. Kod koji je lociran u jednom fajlu, modul1.ts, sa definisanom klasom i sa kreiranim konkretnim objektom te klase.

modul1.ts

```
class Osoba{
    name:string;
    godine:number;
}
let jova : Osoba;
jova = new Osoba();
jova.godine = 33;
jova.name = "Jovan Jovanović";
```

Kao što vidite, u fajlu modul1.ts definisana je klasa `Osoba` i objekat `jova` koji je tipa `Osoba`. Definisana svojstva u klasi su: `name` i `godine`. Ako pokušamo da ih koristimo u drugom fajlu, na primer u fajlu primer1.ts, dobićemo grešku jer podaci eksportovani tj. ne koristi se kao modul.

Ukoliko je napisani kod u fajlu modul, tada se napisani kod može koristiti tj. dobiti pristup do pojedinih promenljivih, funkcija ili cele klase, iz drugog fajla. Dakle, kocept modula omogućava da možemo u drugom fajlu imati funkcionalan, čitljiv kod. Pogledajmo kako se to radi u fajlu primer1.ts:

primer1.ts

```
jova.godine = 22; //upotreba postojeće promenljive
let pera : Osoba; //upotreba postojeće klase
pera = new Osoba();
pera.godine = 33;
pera.name = "Petar Perić";
```

Primenom modula, kod koji je napisan u modulu, ostaje lokalni za svaki fajl i ne može se koristiti izvan fajla. Da bi se omogućio pristup izvan fajla jedna promenljiva ili klasa mora da se eksplicitno označi ključnom reči **export**. Tada kod u fajlu postaje modul.

U fajlu gde je potrebno pristupiti eksportovanim klasama, interfejsima, funkcijama ili promenljivama navodi se ključna reč **import** uz fajl koji ih sadrži. Na ovaj način, kod koji iz drugog fajla ostaje vezana za fajl, čak i kada se navede promenljiva istog imena, nije moguća nejasnoća o kojoj promenljivoj je reč.

Za fajl ili fajlove iz kojih se vrši izvoz koda sintaksa je sledeća:

```
export nazivKlase;
export nazivInterfejsa;
export nazivFunkcije;
export nazivPromenljive;
```

Dok za fajl ili fajlove gde se vrši uvoz koda:

```
import {nazivKlase} from "nazivFajlaKlase.ts";
import {nazivInterfejsa} from "nazivFajlaInterfejs.ts";
import {nazivFunkcije,nazivPromenljive} from "nazivFajla.ts";
```

Primenjujući ovaj pristup za rad sa modulima, modifikovaćemo prethodni primer, tako da kod bude ponovo korišćen u drugom fajlu. Takođe, uradićemo i preimenovanje importovane promenljive:

modul1.ts

```
export class Osoba{
    name:string;
```

```
    godine:number;
}
export let jova : Osoba;
jova = new Osoba();
jova.godine = 33;
jova.name = "Jovan Jovanović";
```

primer1.ts

```
import {Osoba as Osoba} from "./modul1";
import {jova} from "./modul1"

jova.godine = 22; //upotreba postojeće promenljive

let pera : Osoba; //upotreba postojeće klase
pera = new Osoba();
pera.godine = 33;
pera.name = "Petar Perić";
```

Prilikom izvoza može se upotrebiti ključna reč **`default`**. Ovakav izvoz se koristi kada želimo da iz fajla izvezemo samo jednu glavnu vrednost, klasu ili funkciju.

Na sličan način obezbeđuje se podrazumevani **`import`** za takvu promenljivu, funkciju, interfejs ili klasu. Na primer:

primer2.ts

```
const pi:number = 3.14159;
export default pi;
```

primer1.ts

```
import PI from "./primer2"
```

Promenljiva **`pi`** je izvezena kao podrazumevana. Zato se pri uvozu, osim fajla koji se koristi, dovoljno navesti samo naziv preimenovane promenljive, u našem slučaju **`PI`**.

Učitavači

Moduli ne funkcionišu izolovano – da bi bili deo celokupnog rešenja, moraju se povezati sa drugim fajlovima koji čine projekat. Svaki fajl koji se koristi kao modul mora biti preveden zajedno sa ostalima, kako bi se dobila konačna aplikacija.

Da bi se obezbedilo pravilno povezivanje, koristi se **učitavač modula** (eng. module loader). On je zadužen da pronađe i učita zavisnosti koje se importuju iz drugih fajlova.

- U **Node.js** okruženju najčešće se koristi **CommonJS** loader.
- U veb pregledačima se često koristi **RequireJS**, koji je zasnovan na AMD (eng. Asynchronous Module Definition) sistemu.

Kada kompajliramo TypeScript kod, potrebno je da navedemo koji sistem modula koristimo. To se radi pomoću opcije `--module` u komandi `tsc`.

Na primer, ako se koristi *CommonJS*:

```
tsc --module commonjs primer1.ts
```

Za prevođenje koda koristeći RequireJs module koristi se komanda amd (eng. Asynchronous Module Definition). Glavna karakteristika amd sistema je da omogućava asinhrono učitavanje modula, što je korisno za veb okruženja gde treba minimizovati vreme učitavanja:

```
tsc --module amd primer1.ts
```

Imenski prostori

Imenski prostor (*namespace*) u TypeScript-u predstavlja način da se više klasa, interfejsa, funkcija i promenljivih grupiše u jednu logičku celinu unutar istog fajla. Na taj način se izbegava „zagađenje“ globalnog prostora imena i obezbeđuje jasna organizacija koda.

Kada se elementi definišu unutar imenskog prostora, oni se mogu učiniti dostupnim spolja pomoću ključne reči `export`. Sintaksa izgleda ovako:

```
namespace nazivImenskogProstora{
    export class nazivKlase{
    }
    export interface nazivInterfejsa{
    }
    export var nazivPromenljive;
}
```

U sledećem primeru, u okviru imenskog prostora Studenti definišu se i izvoze:

- interfejs `IOsoba`,
- interfejs `IStudent`,
- funkcija `sviPodaciStudenta`,
- klasa `Student`.

Time se svi ovi elementi grupišu u jednu celinu i jasno se razlikuju od drugih delova koda.

```
namespace Studenti {

    export interface IOsoba {
        ime: string;
        jmbg: number;
    }
    export interface IStudent extends IOsoba{
        indeks:string;
    }

    export function sviPodaciStudenta (s:Student) {
        return s.prikazStudenta() + " " + s.jmbg;
    }

    export class Student implements IStudent {
        ime: string;
        jmbg: number;
        indeks: string;

        constructor(stdata: IStudent) {
            this.ime = stdata.ime;
            this.jmbg = stdata.jmbg;
            this.indeks = stdata.indeks;
        }

        prikazStudenta(): string {
            return this.indeks + " " + this.ime;
        }
    }
}
```

Primena

Kada se koristi imenski prostor, svi njegovi elementi se pozivaju uz prefiks koji je naziv tog prostora. Nakon naziva imenskog prostora stavlja se tačka, pa naziv klase, interfejsa, funkcije ili promenljive.

Na primer, u prethodnom slučaju:

```
let pera:Studenti.Student = new Studenti.Student(
{ime:'Pera',jmbg:12345,indeks:'RT-5/55'});
let detalji:string = Studenti.sviPodaciStudenta(pera);
```

Literatura

[1] David Flanagan, JAVASCRIPT SVEOBUHVATAN VODIČ 7.izdanje, MIKRO KNJIGA, 2021

[2] Addy Osmani, JAVASCRIPT PROJEKTNI OBRASCI, KOMPJUTER BIBLIOTEKA, 2023

[3] Marijn Haverbeke, JavaScript ELOKVENTNO, MIKRO KNJIGA, 2019

[4] Laurence Lars Svekis, JAVASCRIPT OD POČETNIKA DO PROFESIONALCA, KOMPJUTER BIBLIOTEKA, 2022

[5] Ved Antani, Stojan Stefanov, OBJEKTNO ORIJENTISAN JAVASCRIPT III izdanje, KOMPJUTER BIBLIOTEKA, 2017

[6] Federico Kereki, JAVASCRIPT Funkcionalno programiranje, prevod drugog izdanja, KOMPJUTER BIBLIOTEKA, 2020

[7] David Herron, NODE.JS VEB RAZVOJ - prevod V izdanja, KOMPJUTER BIBLIOTEKA, 2020

[8] Joran Quinten, Building Real-World Web Applications with Vue.js 3, Pack 2024.

[9] Victor Jayden, Level Up Your JavaScript, A Practical Guide to TypeScript, ASIN B0DPVKM2V6

[10] Natan Rozentals, NAUČITE TYPESCRIPT prevod II izdanja, KOMPJUTER BIBLIOTEKA 2017

[11] https://vuejs.org/, pristupljeno: decembar 2025.

[12] https://www.vuemastery.com/, pristupljeno: decembar 2025.

[13] https://www.youtube.com/watch?v=zwvUAh91itA, pristupljeno: decembar 2025.

[14] https://developer.mozilla.org/en-US/docs/Learn_web_development/Core/Frameworks_libraries/Vue_getting_started, pristupljeno: decembar 2025.

[15] https://www.geeksforgeeks.org/javascript/vue-js/, pristupljeno: decembar 2025.

[16] https://www.tutorialspoint.com/vuejs/index.htm, pristupljeno: decembar 2025.

[17] Freeman, Adam. *Pro ASP.NET Core 7* Manning 2023.

[18] https://learn.microsoft.com/en-us/aspnet/core/?view=aspnetcore-9.0, pristupljeno: decembar 2025.

[19] Mark J. Price, C# 14 and .NET 10 – Modern Cross-Platform Development Fundamentals - Tenth Edition, 2025.

[20] Main Razor Syntax Rules for C#, w3schools.com/asp/razor_syntax.asp, pristupano: decembar 2025.

[21] Steve Smith, Build beautiful, responsive sites with Bootstrap and ASP.NET Core, https://aspnetcore.readthedocs.io/en/stable/client-side/bootstrap.html, pristupljeno: decembar 2025.

[22] Dino Esposito , Clean Architecture with .NET, Microsoft Press 2024.

[23] Trevoir Williams, Complete ASP.NET Core and Entity Framework Development, Packt 2024.

[24] Rick Anderson, Examining how ASP.NET MVC scaffolds the DropDownList Helper, https://learn.microsoft.com/en-us/aspnet/mvc/overview/older-versions/working-with-the-dropdownlist-box-and-jquery/using-the-dropdownlist-helper-with-aspnet-mvc pristupljeno: decembar 2025.

[25] Joseph Albahari, Ben Albahari, C# 12 in a Nutshell, O'Reilly 2023.

[26] Mirza Leka, Getting Started with Linq in C#, https://dev.to/mirzaleka/getting-started-with-linq-in-c-13-210o, pristupljeno: decembar 2025.

[27] Language Integrated Query, https://learn.microsoft.com/en-us/dotnet/csharp/linq/ , pristupljeno: decembar 2025

[28] Boris Cherny, TypeScript programiranje – unapredite vaše JavaScript aplikacije, CET 2021.

[29] Natan Rozentals, Naučite TypeScript, prevod II izdanja, Kompjuter biblioteka 2017.

[30] Martin Krause, The Complete Developer, Master The Full Stack With TypeScript, React, Next.Js, MongoDb, And Docker, Published by No Strach Press, 2024.

[31] Dan Vanderkam, Effective TypeScript, 83 Specific Ways to Improve Your TypeScript, O'Reilly 2024.

[32] TypeScript Tutorial, https://www.typescripttutorial.net/, pristupljeno: decembar 2025.

[33] TypeScript Documentation, https://www.typescriptlang.org/docs/, pristupljeno: decembar 2025.

[34] TypeScript tutorial in Visual Studio Code, https://code.visualstudio.com/docs/typescript/typescript-tutorial, pristupljeno: decembar 2025.

Rečnik skraćenica

ADO.NET – ActiveX Data Objects for .NET (Tehnologija za pristup bazama podataka u .NET okviru).

AJAX – Asynchronous JavaScript and XML (Tehnika za asinhrono osvežavanje delova web stranice bez ponovnog učitavanja cele stranice).

API – Application Programming Interface (Programski interfejs aplikacije koji omogućava komunikaciju između dva softvera).

CDN – Content Delivery Network (Mreža servera raspoređenih širom sveta koji ubrzavaju isporuku web sadržaja korisnicima).

CLI – Command Line Interface (Interfejs komandne linije koji omogućava korisnicima interakciju sa programom putem teksta/komandi).

CRUD – Create, Read, Update, Delete (Četiri osnovne operacije za upravljanje podacima u bazama).

CSS – Cascading Style Sheets (Jezik koji se koristi za opisivanje izgleda i formatiranja web stranica).

DOM – Document Object Model (Objektni model dokumenta koji predstavlja strukturu HTML stranice kao drvo objekata).

EDM – Entity Data Model (Konceptualni model podataka koji Entity Framework koristi za mapiranje baza podataka u objekte).

EF – Entity Framework (ORM radni okvir za .NET koji omogućava rad sa bazama podataka koristeći .NET objekte).

HTML – HyperText Markup Language (Standardni jezik za označavanje koji se koristi za kreiranje strukture web stranica).

JS – JavaScript (Programski jezik koji se koristi za kreiranje interaktivnog sadržaja na webu).

JSON – JavaScript Object Notation (Lagan format za razmenu podataka koji je lako čitljiv i ljudima i mašinama).

LINQ – Language Integrated Query (Deo .NET-a koji omogućava pisanje upita nad podacima direktno unutar C# ili VB.NET jezika).

MVC – Model–View–Controller (Arhitektonski obrazac koji deli aplikaciju na tri logička dela: podatke, prikaz i kontrolu).

MVVM – Model–View–ViewModel (Arhitektonski obrazac popularan u WPF-u i Vue.js koji odvaja logiku od korisničkog interfejsa).

NoSQL – Not Only SQL (Tip baza podataka koje ne koriste isključivo relacioni model, pogodne za velike količine nestruktuiranih podataka).

ORM – Object-Relational Mapping (Tehnika koja omogućava povezivanje nekompatibilnih sistema: objektno-orijentisanog koda i relacionih baza).

POCO klase– Plain Old CLR Objects (Jednostavne C# klase koje ne zavise od specifičnih radnih okvira/framework-a).

REST – Representational State Transfer (Arhitektonski stil za dizajniranje mrežnih aplikacija koji koristi HTTP protokol).

SFC – Single-File Components (Fajlovi sa .vue ekstenzijom koji u jednom fajlu sadrže HTML, logiku i stil komponente).

SQL – Structured Query Language (Standardni jezik za upravljanje i komunikaciju sa relacionim bazama podataka).

TS – TypeScript (Nadskup JavaScript-a koji dodaje statičku tipizaciju podataka).

UI – User Interface (Korisnički interfejs – sve ono sa čime korisnik direktno interaguje u aplikaciji).

URL – Uniform Resource Locator (Adresa resursa na internetu, npr. web adresa stranice).

VS – Visual Studio (Integrisano razvojno okruženje – IDE – kompanije Microsoft).

XML – Extensible Markup Language (Jezik za označavanje dizajniran da čuva i prenosi podatke na struktuiran način).

XPath – XML Path Language (Upitni jezik koji se koristi za pronalaženje informacija unutar XML ili HTML dokumenata).

Indeks pojmova

A

ADO.NET, 147
AJAX, 221, 225, 227
All, 154
Anonimni, 151
Any, 154
Api, 207
API, 3, 4, 19, 20, 31, 43, 45, 47, 48, 49, 53, 58, 60, 83, 94, 96, 99, 100, 101, 179, 207, 208, 209, 210, 212, 214, 216, 221, 226, 227, 229, 230, 231, 237
appsettings.json, 221
Asocijacije, 137
Authorize, 233
Average, 162

B

Bootstrap, 303

C

Cast, 161
Compositions, 48
`computed`, 25, 47, 48, 49, 60, 70, 71, 72, 79, 90, 131
`Concat`, 162
Contains, 154, 155
`Controller`, 213, 214, 233
Count, 162
`createApp`, 17, 18, 20, 22, 23, 24, 26, 27, 28, 29, 30, 34, 44, 53, 59, 106
CSS, 23, 25, 40, 41, 45, 76, 77, 78, 85

D

DataSet, 147
DbContext, 166, 173, 178, 179, 180, 182, 183, 184, 185, 186, 191, 194, 196, 201, 206, 211
`defineEmits`, 109, 130
`delegate`, 150
`Distinct`, 154, 155, 156
DOM, 13, 23, 25, 26, 29, 33, 34, 35, 36, 37, 40, 51, 53, 82, 87, 88, 225

E

ElementAt, 160, 161
Empty, 160
EntityType, 136
`enum`, 284
`Except`, 154

F

`fetch`, 73, 94, 95, 99
filteri, 233, 234, 235
Filtriranje, 152
First, 160, 161

G

GET, 225
getJSON, 225
group join, 158
GroupBy, 150, 159, 160
GroupJoin, 157

I

Inject, 120
Instalacija, 140, 141
Interpolacija, 51, 52
Intersect, 154

J

JavaScript, 3, 13, 14, 19, 33, 35, 40, 41, 44, 49, 51, 55, 56, 57, 58, 64, 68, 78, 81, 83, 87, 90, 94, 95, 105, 125, 127, 142, 207, 221, 238, 239, 241, 242, 245, 248, 249, 254, 260, 262, 272, 274, 275, 276, 278, 281, 282, 284, 295, 296, 302, 304
Join, 150, 157, 158
jQuery, 222, 223, 225
JSON, 225

K

Kvantifikatori, 154

L

Lambda, 150
Last, 160, 161
LINQ, 147, 148, 304

M

Math, 287
Max, 162
Microsoft, 147, 222
Migracija, 171
Min, 162
MVC, 303
MVVM, 13

N

Navigation Property, 138
Northwind, 96, 101, 137, 179, 180

O

OfType, 152, 153, 161
Options, 19, 47, 53, 58, 191, 193
OrderBy, 150, 153, 162
ORM, 3, 133, 134, 146, 147, 163, 164, 205, 305, 306
OutputCache, 235

P

pogled, 135
Projekcije, 155
Provide, 119

R

Request, 234

S

Single, 161
Single-File Components, 20, 22, 40
Skip, 156, 157
Skupovni operatori, 154
Solution, 145, 213, 214
Solution Explorer, 213, 214
Sortiranje, 153
Sum, 162

T

Take, 157
template, 128
Testiranje, 229
ThenBy, 153
ThenByDescending, 153

this, 283
Tuple, 286
TypeScript, 3, 4, 19, 20, 42, 43, 49, 105, 126, 127, 128, 238, 239, 240, 241, 242, 245, 248, 250, 251, 252, 255, 260, 262, 264, 265, 267, 270, 275, 276, 281, 282, 284, 289, 292, 295, 296, 299, 302, 304

U

Union, 154, 162

V

v-bind, 34, 35, 54, 56, 57, 58, 59, 61, 66, 76, 88, 92, 117, 124, 125
v-html, 35, 36, 52, 53, 68
ViewBag, 234, 236
virtuelni DOM, 13, 33, 39
Visual Studio, 141, 142
v-model, 38, 39, 72, 89, 90, 91, 92, 93, 101, 103, 104
v-on, 35, 36, 37, 58, 59, 61, 107, 123, 124
v-slot, 35, 112, 113, 114, 115
v-text, 35, 53

W

watch, 72, 73, 74, 75

X

XML, 147

Z

Združivanja, 157
Životni ciklus, 29, 32

Indeks slika

Slika 1.1. Pogled na ekstenziju „*Vue – Official*“ pre instalacije u VS Code 16
Slika 1.2. Postavljanje „Vue.js devtools“ dodatka: (i)izbor iz menija (ii)pogled u instaliranim alatkama 22
Slika 1.3. Primer sa kodom i tekstom u konzolnom prozoru 31
Slika 1.4. Životni ciklus –https://v3.Vue.org/guide/instance.html#lifecycle-diagram 32
Slika 1.5. Primer povezivanja primenom direktiva 36
Slika 1.6. Primena v-model direktive u bidirekcionom povezivanju 39
Slika 1.7. Struktura generisanih foldera 43
Slika 1.8. Pogled na pokrenuti projekat 43
Slika 1.9. Prikaz povezivanja samo jednom kroz alat za debagovanje 52
Slika 1.10. Rezultat tekstualne interpolacije 54
Slika 1.11. Povezivanje atributa logičkog tipa 55
Slika 1.12. Pogled na prozor Console nakon izvršavanja 97
Slika 1.13. Pogled za ispravno dobavljanje vrednosti kriptovalute 99
Slika 1.14. Izgled za deo dobijanja jedne kategorije 102
Slika 2.1. Primer šeme podataka 135
Slika 2.2. Izbor opcija pri instalaciji razvojnog okruženja Visual Studio 141
Slika 2.3. Pokretanje Visual Studio IDE okruženja za razvoj aplikacija 142
Slika 2.4. Izbor šablona novog projekta 143
Slika 2.5. Podaci konzolne aplikacije 144
Slika 2.6. VS nakon kreiranja aplikacije 145
Slika 2.7. Prikaz anonimnih tipovima u LINQ izrazima kroz VS 151
Slika 2.8. Rad sa tipiziranim objektom u VS 152
Slika 2.9. Prikaz rezultata LINQ izraza u IDE okruženju 156

Slika 2.10. Višestruki select u *debug* režimu 156
Slika 2.11. Prikaz rezultata LINQ izraza preko *Watch* prozora u toku izvršavanja 159
Slika 2.12. Package Manager Console 163
Slika 2.13. Otvaranje prozora za pretragu paketa 164
Slika 2.14. Izbor paketa za uključivanje u EF projekat 165
Slika 2.15. Otvaranje *Server Explorer* prozora početak dodavanja nove konekcije 168
Slika 2.16. Podešavanje konekcije 168
Slika 2.17. Pogled na Server Explorer i neke njegove funkcije 169
Slika 2.18. Kreiranje baze - MS SQL Server Management Studio 169
Slika 2.19. Kreiranje novih tabela - MS SQL Server Management Studio 171
Slika 2.20. Otvaranje konzole za rad sa migracijama 172
Slika 2.21. Pokretanje migracije 173
Slika 2.22. Kreirani folder Migrations u okviru projekta 174
Slika 2.23. `Update-Database` u PM prozoru 175
Slika 2.24. Kreirana baza i tabele nakon sinhronizacije 175
Slika 2.25. Kreirana baza sa inicijalnim podacima 177
Slika 2.26. Pogled na generisanje modele 180
Slika 2.27. Relacija 1 na 0..1 181
Slika 2.28. Relacija 1 na 0..* 182
Slika 2.29. Veza više na više 184
Slika 2.30. Prikaz sql upita u Profiler prozoru. (i) Izbor Proviler-a iz menija, (ii)Podešavanje opcija (ii)Prikaz dobijenih rezultata primenom LINQ izraza 192
Slika 2.31. Prikaz konkretnog sql upita 199
Slika 2.32. Priakaz konkretnog sql upita sa rezultatima 200
Slika 2.33 Prikaz konkretnog sql upita 200
Slika 2.34. AsNoTracking vs AsNoTrackingWithIdentityResolution 202
Slika 3.1. Prikaz jedne veb stranice za prikaz liste knjiga 208
Slika 3.2. Filtriranje šablona u izbora tipa projekta: ASP.Net Core Web API 208
Slika 3.3. Imenovanje projekta 209
Slika 3.4. Dodatne informacije o novom projektu 209
Slika 3.5. Dodavanje nove klase za model 210

Slika 3.6. Dodavanje kontrolera: (i)direktno (ii)preko alatke *Scaffolding* (iii)određivanje stavke....213
Slika 3.7. Definisanje naziva kontrolera....214
Slika 3.8. Pogled na prozor **`Solution Explorer`**....214
Slika 3.9. Pogled na swagger stranicu nakon kreiranja metoda....216
Slika 3.10. Automatsko dodavanje akcija kontrolera....217
Slika 3.11. Izbor modela u postupku automatskog dodavanja akcija....218
Slika 3.12. Dodavanje nove stavke....222
Slika 3.13. Izbor HTML stranice....222
Slika 3.14. Uključivanje postojeće biblioteke projektu....223
Slika 3.15. Prikaz početne stranice....229
Slika 3.16. Prikaz rezultata pretrage....230
Slika 3.17. Prikaz pogrešnog zahteva za pretragu....230
Slika 3.18. Kartice u veb čitaču korišćene za razvoj....231
Slika 3.19. Network kartica....231
Slika 3.20. Praćenje sadržaja poruka....232
Slika 4.1. Provera instalirane verzije paketa Node, odnosno alata npm....240