Razvoj web aplikacije za dohvaćanje i pohranu informacija o molekulama korištenjem okvira ASP.NET

Ladić, Mihael

Undergraduate thesis / Završni rad

2023

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Rijeka / Sveučilište u Rijeci**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:195:861011

Sveučilište u Rijeci
Fakultet informatike i digitalnih tehnologija
Sveučilišni prijediplomski studij Informatika

Mihael Ladić

Razvoj web aplikacije za dohvaćanje i pohranu informacija o molekulama korištenjem okvira ASP.NET

Završni rad

Mentor: prof. dr. sc. Vedran Miletić

Rijeka, 17. srpnja 2023.

Sažetak

Ovaj završni rad obuhvaća pregled razvoja moderne web aplikacije u razvojnom okviru ASP.NET na primjeru web aplikacije ElementSearch koja omogućava spremanje i upravljanje podatcima o kemijskim spojevima, fokusirane na mogućnosti međusobne komunikacije i dijeljenja znanja između različitih instanci aplikacije. Također se dotiče procesa postavljanja testnog okruženja s više instanci ElementSearch aplikacije u Docker containerima.

Ključne riječi: web; aplikacija; ASP.NET; kemija; molekule; suradnja;

Sadržaj

Poglavlje 1

Uvod

Web je postao nezamjenjiv aspekt svakodnevnog života svih stanovnika srednje razvijenih i razvijenih država modernog svijeta. Samim time porasla je i uloga web aplikacija.

Najjednostavnije rečeno, web aplikacija je interaktivni program koji se izvodi na poslužitelju udaljenom od korisničkog računala (kojeg u kontekstu web aplikacija zovemo klijent), i prikazuje se na njegovom web pregledniku.

Glavna prednost web aplikacija u odnosu na najpopularnije alternative (mobilne i desktop aplikacije) je činjenica da se web aplikacije ne trebaju preuzimati niti instalirati prije korištenja. Dovoljno je poslati zahtjev (http) na poslužitelj za sadržajem koji tamo stvara, obrađuje ili pohranjuje aplikacija te interpretirati odgovor koji dolazi u obliku statičkog sadržaja (html, xml i css) i pripadnih skripti (JavaScript). To sve je najčešće zadatak web preglednika. Dodatna interaktivnost, koja mora biti prisutna kako bi se web sadržaj mogao nazvati aplikacijom, ostvaruje se daljnjom komunikacijom klijenta i poslužitelja preko sličnih zahtjeva i odgovora.

Iz pogleda korisnika, zahtjevi koji definiraju dobru web aplikaciju ne razlikuju se od onih kod mobilnih i desktop aplikacija - intuitivnost, jednostavnost korištenja, brzina izvođenja, dizajn, korisnost, zabava itd.

Ovaj rad proučava proces razvoja moderne web aplikacije unutar razvojnog okvira ASP.NET na primjeru aplikacije za spremanje i upravljanje podatcima o kemijskim spojevima, fokusirane na dijeljenje sadržaja i suradnju, zvane ElementSearch.

Poglavlje 2

Pregled razvoja aplikacije na modernom webu

2.1 Komponente moderne web aplikacije

Sve moderne web aplikacije se sastoje od dva osnovna dijela: klijentski (*front-end*) i poslužiteljski (*back-end*) dio.

Kako samo ime kaže, poslužiteljski dio se izvodi na poslužitelju web aplikacije. Služi za provođenje zahtjeva koje poslužitelj zaprimi od klijenata, zajedno sa svim sporednim zadatcima koje provođenje zahtjeva podrazumijeva, poput autentikacije i autorizacije korisnika, te validacije unosa, stvaranja zapisa i sl. Za kodiranje *back-end* dijela aplikacije najčešće se koriste objektno orijentirani i skriptni jezici kao na primjer Python, PHP, C#, Java, JavaScript itd. Neki popularni framework-ovi su Ruby on Rails, Django, Cake i ASP.NET.

Klijentski dio se izvodi u web pregledniku klijentskog računala. Služi za obavljanje jednostavnijih, manje zahtjevnih zadataka u svrhu poboljšanja korisničkog iskustva. Neki od primjera su sortiranje, filtriranje, sakrivanje i prikazivanje sadržaja. U osnovi se za kodiranje *front-end*-a koriste HTML, CSS i JavaScript, ali kao nadogradnja na to su dostupne mnoge biblioteke i razvojni okviri. Popularni razvojni okviri su Angular, Vue i React.

Kada je to potrebno, *front-end* ostvaruje komunikaciju s *back-end*-om i traži ili šalje podatke. Najčešći format tih podataka je JSON - *JavaScript Object Notation*, tekstualni format prikladan za prikaz objekata i lista objekata. Osim JSON-a, mogući su i drugi formati komunikacije, poput datoteke - *Upload* na *server* i *Download* sa *servera*.

Jedan od čestih primjera primjene, na kojem se može konkretno objasniti poboljšanje korisničkog iskustva, je validacija na strani klijenta (engl. “client-side validation”). Validacija je provjera ispravnosti korisničkog unosa neophodna na strani *servera*, i poželjna na strani klijenta. Na primjer, možemo uzeti formu za registraciju korisnika, gdje je neophodno upisati korisničko ime. U slučaju bez validacije na strani klijenta, ukoliko korisnik nije unio koris-

ničko ime, klijent šalje zahtjev na poslužitelja, koji obavi validaciju i vraća odgovor klijentu da je registracija neuspješna i zašto, nakon čega klijentska strana prikazuje grešku korisniku. U toj komunikaciji između poslužitelja i klijenta se može izgubiti osjetna količina vremena. No ako validacija postoji i na strani klijenta, greška će se odmah prikazati korisniku, bez potrebe za komunikacijom s poslužiteljem. Naravno, validacija i dalje mora postojati na *serveru*, jer zlonamjerni korisnici validaciju na klijentskoj strani mogu lako zaobići (gašenjem ili izmjenom *JavaScript* koda u *browseru*), i u najgorem slučaju oštetiti podatke i sustav.

Osim *front-end* i *back-end* dijela aplikacije, dodatne komponente čine ostale opcionalne usluge i moduli koji komuniciraju s *back-end* dijelom aplikacije. Najčešća treća komponenta, koju skoro bez iznimke koriste sve ozbiljnije web aplikacije, je relacijska baza podataka. Ostale komponente koje se mogu pokazati korisnima su nerelacijske baze podataka, Email servisi, programi za periodičku sinkronizaciju podataka, i sl.

2.2 Zahtjevi prilikom razvoja web aplikacije

Prije svega, kada se dio aplikacije izvodi na poslužitelju negdje daleko od korisnika, javlja se varijabla dostupnosti. Najčešće je potrebno imati poslužitelj dostupan svaki dan, u svako doba dana, dok se u primjerima najveće skale javlja i potreba za dostupnošću iz bilo kojeg mjesta na svijetu. Razlozi nedostupnosti mogu biti hardverski kvarovi, prirodne katastrofe, greške u sustavu, administraciji, softveru itd. Također, zbog jednostavnog pristupanja uslugama aplikacije moguće su velike i nagle promjene u trenutnom broju korisnika, što u nekim slučajevima stvara hitnu potrebu za nadogradnjom poslužitelja ili skalabilnim sustavom.

Iako je sve to velikim dijelom i pitanje načina postavljanja poslužitelja i aplikacija na njemu, također je potrebno poduzeti preporučene korake u razvoju kako bi se minimizirao broj grešaka i problema koji promaknu razvojnom timu i dospiju u produkcijsko okruženje, te optimizirao proces ispravka istih te uvođenja potrebnih promjena. Kako bi to bilo moguće, potrebno je voditi brigu o mnogim elementima razvojnog procesa poput načina uvođenja nadogradnji, testiranja, verzioniranja izvornog koda, upravljanja zavisnostima, konfiguracije, zapisa i dr.

2.3 Dvanaesto-faktorska aplikacija

Iako pristup razvoju web aplikacija može ovisiti o osobnom učenju i metodologijama, grupa programera koja je sudjelovala u razvoju stotina web aplikacija uvidjela je probleme od kojih barem neke svatko tko se upusti u razvoj web aplikacija primijeti prije ili kasnije [1]. Zato su napisali općenite smjernice kojih bi se svaki web developer trebao držati, poznatije kao Dvanaesto-faktorska aplikacija. Dvanaest faktora je napisano imajući na umu web aplikaciju namijenjenu za izvedbu u oblaku (na *software-as-service* uslugama kao Azure i AWS), ali mnoge smjernice mogu se primijeniti na razvoj web aplikacija općenito.

Konkretno, prvi faktor govori kako je potrebno držati bazu koda jedne web aplikacije u jednom sustavu za verzioniranje. Time se olakšava proces postavljanja web aplikacije (i koda općenito) jer ne dolazi do komplikacija prilikom rada na različitim aplikacijama u jednom Git repozitoriju ili do problema prilikom usklađivanja dijelova aplikacija spremljenih u različitim repozitorijima.

Sljedeći faktori su također sve prakse koje bi prilikom razvoja web aplikacija trebale biti opće znanje, no vrijedi izdvojiti nekoliko praksi koje manje iskusni developeri češće krše.

Na primjer, treći faktor koji govori o potrebi odvajanja konfiguracije aplikacije od koda. Mnogim programerima je prvi instinkt spremiti sve potrebne parametre za rad u lokalne varijable unutar koda. Naravno, takav pristup je neispravan zbog toga što konfiguracija definira uvjete pod kojima se kod izvodi. Ti uvjeti se u pravilu mijenjaju puno češće nego sam kod. Kada bi konfiguracija bila spremljena u kodu, za svaku izmjenu u konfiguraciji bilo bi potrebno ponovo kompajlirati kod, spremiti promjene u sustavu za verzioniranje, i postaviti novu "verziju" u radno okruženje, iako u stvarnosti nema potrebe za takvim postupcima, već je samo potrebno odvojiti promjenjivu konfiguraciju od verzioniranog koda.

Slični problemi s nepotrebno kompliciranim promjenama u kodu mogu nastati ako aplikacija nije kreirana imajući na umu četvrti faktor. Radi se o načinu na koji se aplikacija ponaša u komunikaciji sa vanjskim resursima i uslugama. Najčešći primjer takve usluge je baza podataka, no česte usluge su i mail usluga, repozitorij dokumenata, treće-partijski *API* i sl. Sukladno sa smjernicama objektno orijentiranog programiranja, naglašava se potreba za korištenjem generaliziranih sučelja odvojenih od aplikacijskog koda za pristup vanjskim uslugama. Cilj tome, a ujedno i provjera je li četvrti faktor zadovoljen, je mogućnost promjene vanjske usluge (npr. promjena baze podataka sa MySQL-a na PostgreSQL) bez velikih promjena u aplikacijskom kodu - to je u idealnom slučaju izmjena jedne linije u kojoj se specificira korišten sustav za upravljanje bazama podataka.

Nadalje, jedna od bitnijih stvari koje developer web aplikacija treba imati na umu je činjenica da će njegovoj aplikaciji istovremeno pristupati deseci, stotine, ili tisuće korisnika. Zato je bitno programirati procese imajući na umu da će se mnogo instanci izvoditi odjednom. U takvoj situaciji potrebno je izbjegavati slučajeve kada si te instance međusobno onemogućuju, blokiraju ili narušavaju rad.

Poglavlje 3

Pregled razvoja aplikacije korištenjem razvojnog okvira ASP.NET

ASP.NET je razvojni okvir otvorenog koda namijenjen razvoju web aplikacija i servisa, stvoren i održavan od strane Microsofta [2]. Temelji se na platformi .NET, koja u osnovi sadrži alate i biblioteke koje se mogu primijeniti na sve vrste aplikacija općenito.

ASP.NET sadrži sve elemente koji su potrebni za razvoj moderne web aplikacije. Neki od njih su okvir za obradu zahtjeva u programskim jezicima C# i F#, sintaksa za izgradnju web stranica i stvaranje njihovih predložaka zvana Razor, te podrška i biblioteke za *Model-view-controller* format Web aplikacije, koji će dalje biti detaljnije opisan [3]. Svi korisnici .NET platforme imaju na raspolaganju veliki ekosustav paketa i biblioteka za različite slučajeve korištenja. Na primjer, vrijedi spomenuti predložak za autentikaciju koji podržava dvofaktorsku autentikaciju i opcije prijave s vanjskim uslugama (Google, Facebook itd.).

Model-view-controller (MVC) je naziv za popularni uzorak razvoja softwarea koji odvaja aplikacijsku logiku u tri međusobno povezana sloja. Prvi sloj je *model*, koji predstavlja strukturu podataka te metode za njihovo dohvaćanje i izmjenu. *View* predstavlja prezentacijski sloj, točnije onaj dio aplikacije koji korisnik vidi. *Controller* je dio aplikacije koji obrađuje zahtjeve i komunicira s modelom, kako bi ili dohvatio podatke te ih formatirao za prikaz u prezentacijskom sloju, ili obradio ulazne podatke te izmijenio spremljene podatke na osnovu ulaznih.

3.1 Visual Studio u kontekstu ASP.NET-a

Iako se, ako netko baš želi, ASP.NET aplikacije mogu razvijati i u drugim razvojnim okvirima, Microsoftov Visual Studio je najbolji odabir zbog integriranih alata i ekstenzija za .NET platformu poput automatskih prijedloga najboljih praksi i refaktorizacije koda te upravitelja paketima i zavisnostima unutar projekta. Kada govorimo o upravljanju paketima, dostupan je NuGet package manager s integriranim grafičkim sučeljem, koje se može koristiti za instalaciju, deinstalaciju i nadogradnje potrebnih paketa, koji predstavljaju jednostavan način za dodavanje

biblioteka u projekt. Pokretanje aplikacije je unaprijed konfigurirano za lightweight poslužitelj IIS Express, te je debugiranje standardno i dovoljno je dodavanje točki prekida (engl. "breakpoint") kako bi se iskoristilo. U suštini, sve potrebno za rad na ASP.NET aplikaciji je sakupljeno u jedno integrirano okruženje. Jedan od primjera zadataka koji se mogu obaviti bez izlaska iz tog okruženja je dodavanje migracija, s kojim ćemo se susresti u primjeru koji slijedi.

Poglavlje 4

Tijek izvedbe ASP.NET web aplikacije

U ovom poglavlju promatrat ćemo tijek izvedbe ASP.NET MVC aplikacije na primjeru stranice za popis kemijskih elemenata u Web aplikaciji napravljenoj u sklopu ovog rada.

4.1 Model

Model u kontekstu ASP.NET aplikacije u pravilu podrazumijeva skup objekata koji definiraju strukturu baze podataka povezane s aplikacijom, te sučelje koje se koristi za komunikaciju s navedenom bazom. Općenito, moderne Web aplikacije za komunikaciju s relacijskom bazom podataka koriste objektno-relacijske mapere (ORM). To su biblioteke ili skupovi biblioteka koji se koriste za pretvorbu podataka iz relacijske baze u objekte unutar koda Web aplikacije i obrnuto. Konkretno, jedan objekt neke klase koja definira model predstavlja jedan redak u relacijskoj bazi podataka, a polja unutar tog objekta se podudaraju sa stupcima unutar tablice. Samim time, ako želimo iskoristiti ORM kako bismo dohvatili sve podatke iz neke tablice, dobiti ćemo listu objekata neke klase za koju smo definirali povezanost s dohvaćenom tablicom.

ORM-ovi se, u slučaju da baza podataka nije prethodno stvorena, mogu koristiti i za stvaranje baze podataka na osnovu modela definiranog u MVC aplikaciji, što nazivamo Code-First pristupom razvoja - prvo se razvije kod, a tek onda podatci na osnovu koda. Nakon definiranja modela, ORM može generirati datoteku koja sadrži upute za stvaranje nove baze podataka. Ta datoteka se dogovoreno naziva migracijska datoteka ili samo migracija. Isto tako, ako naknadno postoji potreba za izmjenom modela, stvara se nova migracija koja definira što se mora izmijeniti u relacijskom modelu spojene baze podataka. Migracije sadrže podatke o aktualnom relacijskom modelu, koje ORM općenito koristi prilikom pristupa bazi podataka.

ASP.NET aplikacije koriste moćni Microsoftov objektno-relacijski maper naziva Entity Framework (EF). U nastavku promatramo kako se tablica kemijskih elemenata (ChemElements) stvara i mapira koristeći Entity Framework.

Prije svega, moramo definirati našu konekciju prema bazi podataka. Postoji mnogo načina

za spremiti konfiguracijske podatke, ali u ovom slučaju koristiti ćemo appsettings.json, koji se stvara pri inicijalizaciji projekta. *Connection string* spremamo unutar liste "ConnectionStrings" kao u sljedećem primjeru:

```
{
  "ConnectionStrings": {
    "DefaultConnection": "user id=postgres;Password=123456;Host=localhost;
    Database=ElementSearchDev;Pooling=true;Maximum Pool Size=1024;"
  },
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*"
}
```

Naravno, ovdje definirana baza podataka ElementSearchDev ne mora prethodno postojati, jer će ju u tom slučaju Entity Framework stvoriti za nas.

Zatim možemo stvoriti klasu za entitet kemijskog elementa. Radi organizacije, u praksi se klase modela definiraju u mapi "Models":

```
using System.ComponentModel.DataAnnotations;

namespace ElementSearch.Models
{
    public class ChemElement
    {
        public int Id { get; set; }
        public string Formula{ get; set; }
        [Display(Name="Ime")]
        public string Name { get; set; }
    }
}
```

`[Display(Name="Ime")]` je primjer anotacije, koje se koriste za pobliže opisivanje i definiciju c# koda. Važnost ove anotacije doći će do izražaja u prezentacijskom dijelu aplikacije. Sljedeće je potrebno stvoriti klasu konteksta baze podataka koja služi kao sučelje prema bazi podataka. Ovdje se definiraju sve tablice koje povezana baza podataka sadrži/će sadržati:

```
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore;
```

```
namespace ElementSearch.Data
{
    public class ApplicationDbContext : IdentityDbContext
    {
        public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
            : base(options)
        {
        }

        public DbSet<ElementSearch.Models.ChemElement> ChemElements { get; set; }
    }
}
```

Naš `ApplicationDbContext` nasljeđuje od `Microsoft.AspNetCore.Identity. EntityFrameworkCore.IdentityDbContext`, koji sa sobom uključuje i tablice vezane za autentikaciju (korisnici, grupe, pripadnost grupama itd.). Polje ChemElements je DbSet (skup) podataka tipa ChemElement kojeg smo prethodno definirali. Definicijom ovog polja unutar našeg konteksta Entity Framework-u dajemo do znanja da povezana baza podataka sadrži tablicu Kemijskih elemenata, te ćemo za pristup toj tablici u našem kodu koristiti upravo ovaj DbSet.

Sljedeći korak bio bi stvaranje migracije i na osnovu te migracije stvaranje baze podataka. Unutar Visual Studia potrebno je otvoriti "Package Manager Console". Ako već nije otvoren na donjoj strani sučelja, moguće ga je otvoriti u "Tools > NuGet Package Manager > Package Manager Console". U ovoj konzoli možemo pokrenuti naredbu `Add-Migration ime_migracije` kako bismo stvorili novu migraciju.

Migracije su ovdje C# datoteke, koje sadrže podatke o tome što se promijenilo u modelu baze podataka od posljednje migracije. Sadrže metode "Up" i "Down". Up opisuje naredbe koje se moraju izvesti prilikom promjene modela, a Down naredbe koje se moraju izvesti prilikom poništavanja promjena migracije.

Ako promatramo tablicu ChemElements iz prethodnog primjera, Migracija će u metodi Up sadržavati:

```
migrationBuilder.CreateTable(
    name: "ChemElements",
    columns: table => new
    {
        Id = table.Column<int>(type: "integer", nullable: false)
            .Annotation("Npgsql:ValueGenerationStrategy",
            NpgsqlValueGenerationStrategy.IdentityByDefaultColumn),
        Formula = table.Column<string>(type: "text", nullable: false),
```

```
            Name = table.Column<string>(type: "text", nullable: false)
        },
        constraints: table =>
        {
            table.PrimaryKey("PK_ChemElements", x => x.Id);
        });
```

A u metodi Down:

```
migrationBuilder.DropTable(
    name: "ChemElements");
```

Naš primjer, osim za tablicu elemenata, sadrži takve naredbe i za svaku Identity tablicu koja dolazi s *IdentityDbContext*-om.

Ako naknadno želimo promijeniti strukturu tablice, samo ćemo izmijeniti klasu ChemElement:

```
public class ChemElement
{
    public int Id { get; set; }
    public string Formula{ get; set; }
    [Display(Name="Ime")]
    public string Name { get; set; }

    [Display(Name="Molekulska masa")]
    public float? MolecularWeight { get; set; }

    [Display(Name = "Polar Area")]
    public float? PolarSurfaceArea { get; set; }

    [Display(Name = "Broj teških atoma")]
    public int? HeavyAtomCount { get; set; }
}
```

I zatim ponovo pozvati `Add-Migration ime_migracije`. Nova migracija sada neće sadržavati naredbu `CreateTable`, već `AddColumn`:

```
protected override void Up(MigrationBuilder migrationBuilder)
{
    migrationBuilder.AddColumn<int>(
        name: "HeavyAtomCount",
        table: "ChemElements",
        type: "integer",
        nullable: true);
    migrationBuilder.AddColumn<float>(
```

```
            name: "MolecularWeight",
            table: "ChemElements",
            type: "real",
            nullable: true);
        migrationBuilder.AddColumn<float>(
            name: "PolarSurfaceArea",
            table: "ChemElements",
            type: "real",
            nullable: true);
}
```

Za stvaranje baze podataka ili provođenje izmjena potrebno je samo provesti naredbu `Update-Database` u Package Manager konzoli. Za ovaj korak je posebno bitno da je *Connection string* u appsettings.json datoteci pravilno definiran jer ova naredba automatski čita tamo spremljenu vrijednost. Nakon toga naš kontekst je spreman za korištenje za pristup našoj bazi podataka.

4.2 Controller

Kontroleri u ASP.NET-u su klase koje nasljeđuju od Microsoft.AspNetCore.Mvc.Controller. Često se razdvajaju prema području primjene, što često podrazumijeva podjelu prema tipovima entiteta. Na primjer, u aplikaciji ElementSearch imamo ChemElementsController. Kontroler se sastoji od akcija, koje su zapravo metode do kojih klijent dolazi preko različitih putanja, koje obavljaju različite zadatke, točnije obrađuju različite zahtjeve.

Za pristup uslugama poput prethodno definiranog ORM-a u kontrolerima se koristi Dependency injection [4]. Dependency injection je metoda kodiranja pri kojoj se uklanja potreba za direktnom zavisnošću koda nad klasom na nižoj razini apstrakcije. Konkretnije, bez Dependency injection-a kontroler bi našem kontekstu pristupao otprilike na sljedeći način:

```
public class ChemElementsController : Controller
{
    private readonly ApplicationDbContext _context;

    public ChemElementsController(string connectionString)
    {
        _context = new ApplicationDbContext(connectionString);
    }
}
```

Takva implementacija ima više problema. Ako se implementacija konstruktora `ApplicationDbContext`-a mijenja, biti će potrebno mijenjati i konstruktor kontrolera, točnije svih kontrolera koji koriste `ApplicationDbContext`. Također, svakim instanciranjem kontrolera, mo-

rat će se instancirati novi kontekst koji će se ispočetka morati spojiti na bazu podataka. Alternativa je prepustiti ASP.NET-u da upravlja uslugama u dostupnom upravitelju uslugama IServiceProvider-u. Tada programer ne mora brinuti o stvaranju novih instanci usluga niti njihovom pravilnom završavanju. Tako se, na primjer usluga za komunikaciju s bazom podataka `ApplicationDbContext` može registrirati prilikom pokretanja aplikacije u datoteci Program.cs. Pored postojećih linija, u metodu main, pri samom vrhu je potrebno dodati sljedeće linije:

```
var connectionString = builder.Configuration.GetConnectionString("DefaultConnection")
    ?? throw new InvalidOperationException("Connection string
            'DefaultConnection' not found.");
builder.Services.AddDbContext<ApplicationDbContext>(options =>
    options.UseNpgsql(connectionString));
```

Ovime prvo dohvaćamo "connectionString" definiran u appettings.json, i zatim registriramo uslugu tipa `ApplicationDbContext` te u postavkama definiramo da je to PostgreSQL baza podataka ("UseNpgsql") s pripadnim *connection string*-om.

Sada tu uslugu možemo "injectati" u naš kontroler jednostavno definiranjem parametra tipa `ApplicationDbContext` i postavljanjem unutarnjeg polja na instancu dobivenu parametrom.

```
public class ChemElementsController : Controller
{
    private readonly ApplicationDbContext _context;

    public ChemElementsController(ApplicationDbContext context)
    {
        _context = context;
    }
}
```

Kada smo uspješnjo injectali naš *DbContext* objekt u kontroler, možemo ga pozivati unutar kontrolerovih akcija. Akcije su u praksi metode unutar kontrolera, a jedna akcija odgovara jednom zahtjevu koji server može obraditi. Na primjer, stranica za kreaciju kemijskog elementa će koristiti dva osnovna zahtjeva - GET za prikaz stranice za kreaciju i POST za samu obradu zadatka kreacije - provjera unosa i dodavanje novog elementa u bazu podataka.

```
// GET: ChemElements/Create
public IActionResult Create()
{
    return View();
}

// POST: ChemElements/Create
[HttpPost]
```

```
[ValidateAntiForgeryToken]
public async Task<IActionResult> Create(
    [Bind("Id,Formula,Name,MolecularWeight,PolarSurfaceArea,HeavyAtomCount")]
     ChemElement chemElement)
{
    if (ModelState.IsValid)
    {
        _context.Add(chemElement);
        await _context.SaveChangesAsync();
        return RedirectToAction(nameof(Index));
    }
    return View(chemElement);
}
```

Prva metoda obrađuje GET zahtjeve za rutu ChemElements/Create - ruta koju ASP.NET sam definira na osnovu uzorka "ime_kontrolera/ime_akcije". Za prikaz stranice za unos novog elementa *front-end*-u nisu potrebne dodatne informacije, pa se samo prikazuje stranica. Pošto nije drugačije specificirano, ASP.NET traži prikladni *view* (.cshtml datoteka) u projektnom direktoriju "Views/ime_kontrolera/ime_akcije" - u ovom slučaju je to "Views/ChemElements/Create.cshtml".

Druga metoda odnosi se na samu obradu zahtjeva s validacijom. Potrebno je anotacijom `[HttpPost]` neposredno prije definicije metode naglasiti da se radi o POST zahtjevu (ako nije ništa definirano, pretpostavlja se GET), i iako opcionalno, uobičajeno je koristiti `[ValidateAntiForgeryToken]`, kako bi se za dodatnu sigurnost provjerio klijentov anty-forgery token. Definicijom parametra govorimo metodi da pri dolaznom zahtjevu očekuje podatke oblika ChemElement, a anotacijom `[Bind("Id,Formula,Name,MolecularWeight, PolarSurfaceArea,HeavyAtomCount")]` definiramo koja polja unutar objekta će nam trebati pri obradi zahtjeva. ModelState je polje unutar bazne klase kontroler-a koje sadrži podatke o trenutnom stanju podataka s kojima kontroler radi. Najčešće se koristi za validaciju kao u trenutnom primjeru. Ukoliko su zaprimljeni podaci ispravni, ModelState.IsValid će biti "true" i primjenom injectanog *contexta* dodati će se novi objekt u bazu i zatim spremiti promjene, te preusmjeriti korisnika na Index stranicu u istom kontroleru. Ukoliko zaprimljeni podatci nisu ispravni, korisnika se vraća na stranicu "Create", gdje će mu se prikazati nastale greške kod unosa, o čemu također brine ModelState.

Zbog prikaza podataka iz baze na stranici, upravo je index stranica prikladna za objasniti primjenu Razor-a u izgradnji *view*-ova.

4.3 View

Najprije vrijedi pokazati kako izgleda endpoint za GET Index stranice u našem `ChemElementsController`-u:

```
// GET: ChemElements
public async Task<IActionResult> Index()
{
    return View(await _context.ChemElements.ToListAsync());
}
```

Navedena akcija vraća *view*, ali ovaj puta se kao parametar prosljeđuju podatci koje će pripadni *view* koristiti. U ovom slučaju to je lista objekata tipa ChemElement, povučena direktno iz baze podataka koristeći već poznati kontekst baze podataka.

Slijedi sadržaj Index.cshtml datoteke, koji daje tablični prikaz kemijskih elemenata. To je html datoteka s dijelovima c# koda koji se pozivaju koristeći znak '@'

```
@model IEnumerable<ElementSearch.Models.ChemElement>

@{
    ViewData["Title"] = "Index";
}

<h1>Index</h1>

<p>
    <a asp-action="Create">Create New</a>
</p>

<table class="table" id="elementsTable">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Formula)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Name)
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
```

```
@foreach (var item in Model) {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.Formula)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Name)
            </td>
            <td>
                <a asp-action="Edit" asp-route-id="@item.Id">Edit</a> |
                <a asp-action="Details" asp-route-id="@item.Id">Details</a> |
                <a asp-action="Delete" asp-route-id="@item.Id">Delete</a>
            </td>
        </tr>
}
    </tbody>
</table>

@section scripts {
    <script src="~/lib/simple-datatables/simple-datatables.js"></script>
    <link rel="stylesheet" href="~/lib/simple-datatables/simple-datatables.css" />

    <script>
        new window.simpleDatatables.DataTable("#elementsTable")
    </script>
}
```

U prvoj liniji definiramo tip podataka koji *view* koristi - taj tip podataka se podudara s podatcima kojime smo *view*-u proslijedili prilikom pozivanja `View()` metode u kontroleru. Ienumerable je interface koji između ostalog implementira i tip podataka `List<T>` u c#-u.

Slijedi definicija naslova koji se prikazuje na kartici u pregledniku, i definicija tablice kemijskih elemenata. U tablici se prvo definira zaglavlje, i koriste se metode DisplayNameFor kako bi se prikazalo ime definirano u modelu, unutar spomenute `[Display(Name="...")]` anotacije. Ukoliko polje nema definirano ime za prikaz, prikazati će se sam naziv polja. U tijelu tablice se koristi c#-ov `foreach` kako bi se iterativno generirali redovi u tablici, za svaki kemijski element. Objekt `Model` je onaj objekt koji *view* dobije od *back-end*-a, definiran u prvoj liniji *view*-a. Redovi u tablici se popune podatcima, a u posljednjem stupcu se generiraju linkovi za stranice detalja, uređivanja i brisanja dotičnog elementa.

Na samom kraju definirana je sekcija “scripts”, čime samo ASP.NET-u dajemo smjernicu gdje

na html stranici generirati sadržaj unutar tog bloka koda. Ova sekcija se koristi za referenciranje skripti i pisanje vlastitih. U ovom slučaju koristi se za instanciranje tablice tipa “DataTable”, koristeći biblioteku simple-datatables. Time u prethodno stvorenu tablicu uvodimo mogućnost front-end sortiranja, pretražvanja i paginacije, što su nerijetko potrebne i tražene funkcionalnosti. Biblioteka simple-datatables je bazirana na malo poznatijem JQuery Datatables, a glavna prednost simple-datatables-a je uklonjena zavisnost o moćnoj, ali zastarjeloj biblioteci JQuery.

Poglavlje 5

Web aplikacija ElementSearch

ElementSearch je Web aplikacija namijenjena za prikaz, pretragu, spremanje i izmjenu podataka o kemijskim spojevima, fokusirana na međusobnu suradnju istraživačkih ustanova. Time uz osnovnu pretragu lokalne baze podataka, omogućuje se i pretraga baza podataka suradničkih ustanova u dubinu, koja se ostvaruje komunikacijom s njihovom instancom ElementSearch aplikacije.

5.1 Pregled funkcionalnosti aplikacije

Funkcionalnosti aplikacije ElementSearch se mogu podijeliti u dvije glavne skupine - pregled i upravljanje lokalnim podacima, i pretraživanje vanjskih podataka. Prva skupina se sastoji od stranica za pregled, izmjenu, dodavanje i brisanje kemijskih spojeva te stranice za skupni import koristeći CSV datoteku. Druga skupina se sastoji od pregleda vanjskih podataka za specifični kemijski spoj, i globalnog pretraživanja kemijskih spojeva.

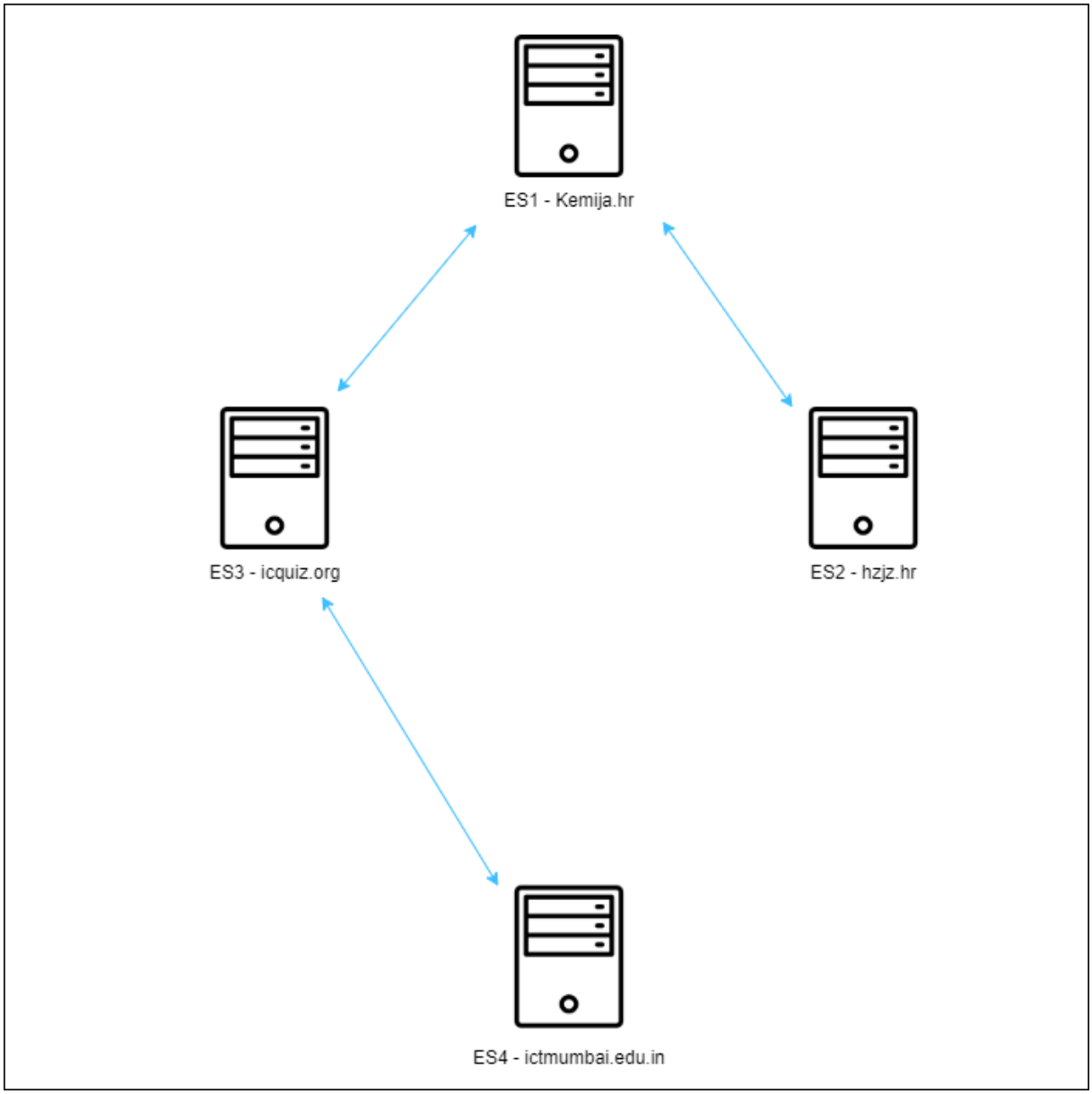
5.2 Stranice za autentikaciju i početna stranica

Stranice za autentikaciju koriste standardni i često viđeni model. Ukoliko korisnik nije prijavljen pristup bilo kojoj od stranica na web aplikaciji preusmjerit će ga na stranicu za prijavu. Ovdje se može prijaviti unosom svog emaila i password-a te klikom na gumb “Prijava”. Prije toga je moguće i označiti “Upamti prijavu”, kako se sljedeći put kada posjeti stranicu s istog računala, korisnik ne bi morao ponovo prijavljivati.

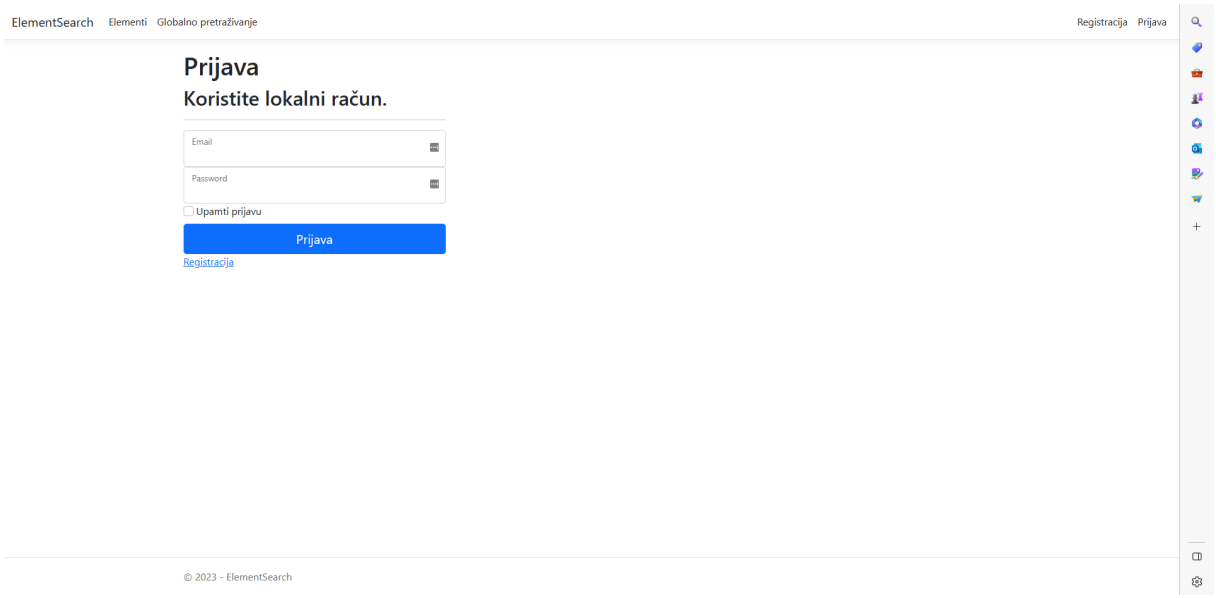
Na početnoj stranici aplikacije nalazi se ime specifične instance (u ovom primjeru Kemija.hr) spremljene u konfiguraciji, i linkovi na glavne funkcionalnosti aplikacije, koji su također vidljivi i u navigacijskoj traci na svakoj stranici.

Sekcija konfiguracije u kojoj se spremaju podatci o web aplikaciji izgleda ovako:

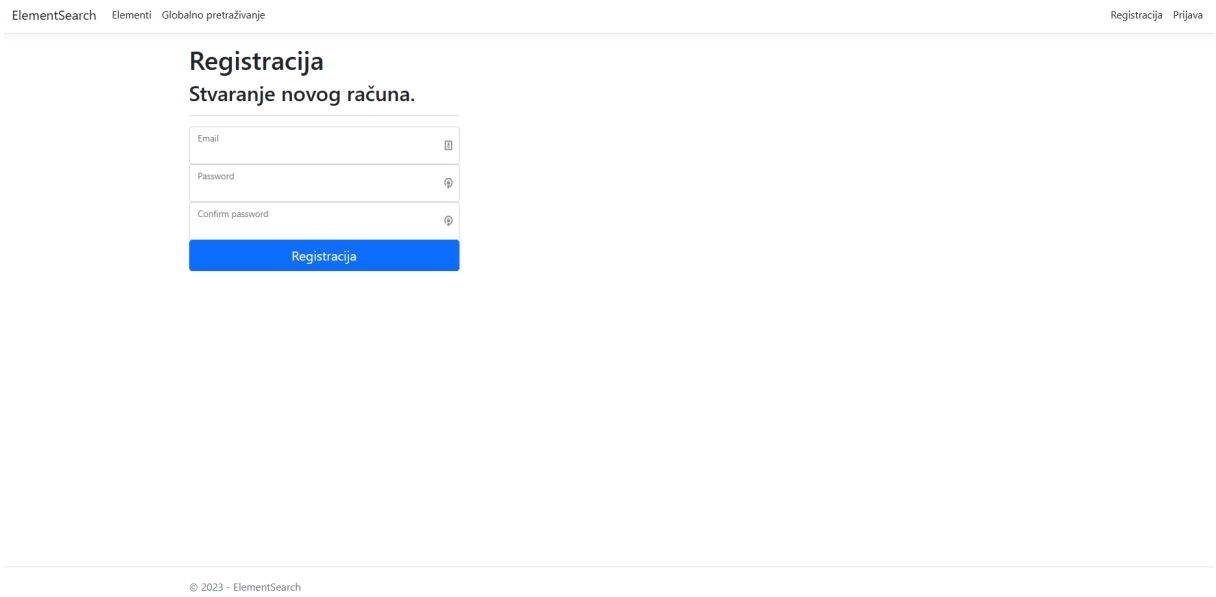
```
"Site": {
```



Slika 5.1: Komunikacija između ElementSearch instanci



Slika 5.2: Login stranica



Slika 5.3: Register stranica

```
    "Id": "6195a367f753",
    "Name": "Kemija.hr",
    "Depth":  10
}
```

Vrijednosti “Id” i “Depth” biti će objašnjene prilikom objašnjavanja funkcionalnosti globalnog pretraživanja.

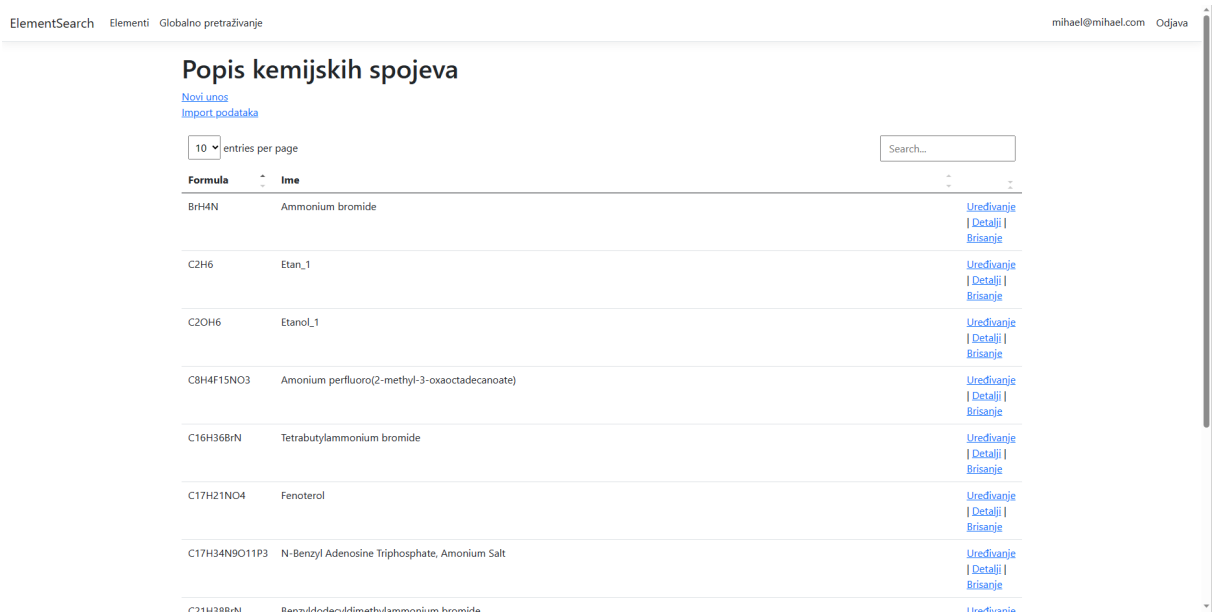
Slika 5.4: Homepage

5.3 Upravljanje lokalnim podacima

Klikom na “Elementi” u navigacijskoj traci ili na “Upravljanje lokalnim podatcima o kemijskim spojevima” na početnoj stranici, dolazimo do stranice na kojoj se prikazuje tablica s formulama i imenima kemijskih spojeva. Pozadinsko funkcioniranje upravo ove stranice je prethodno objašnjeno u sekciji “View” i rezultat je poziva GET na kontroler “ChemElements”, na akciju “Index”.

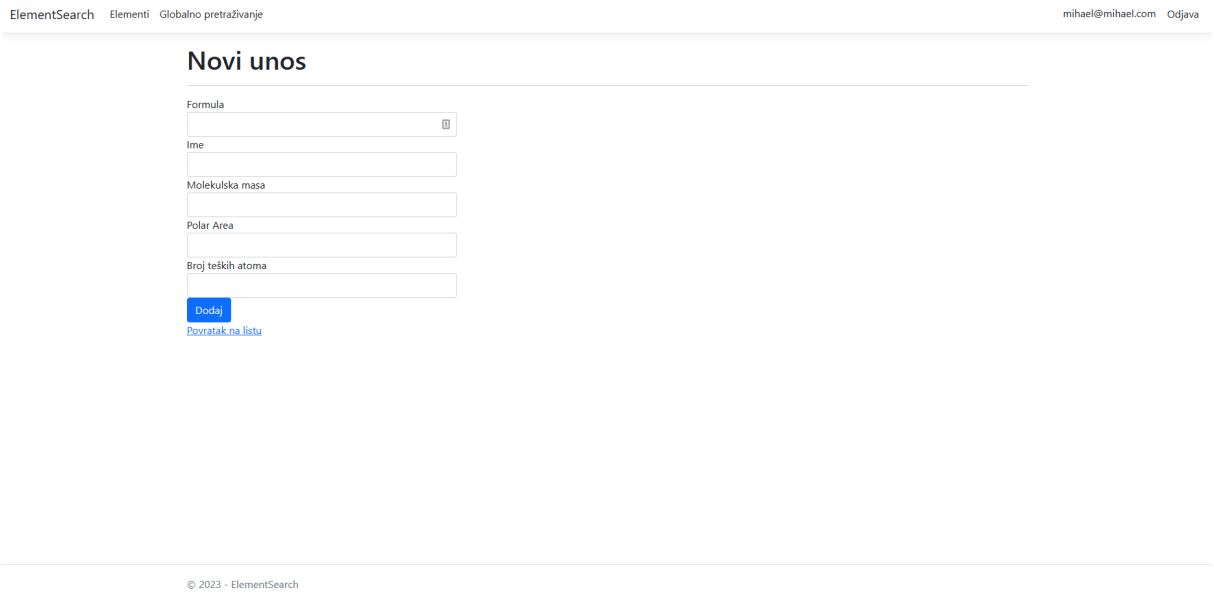
U tablici možemo redove sortirati po stupcu, pretraživati, i birati broj redova koji se prikazuje na jednoj stranici paginiranog (razdvojenog na više listova) sadržaja. Sve to omogućuje nam prethodno spomenuti simple-datatables. Ispod samog naslova nude se linkovi na stranicu za dodavanje vlastitog unosa kemijskog spoja, i na stranicu za import podataka koristeći SCV datoteku. U posljednjem stupcu svakog retka tablice nalaze se linkovi na stranice detalja, uređivanja i brisanja prikladnog reda, točnije kemijskog spoja.

Stranice za novi unos i uređivanje su vizualno praktički identične. Glavna razlika je u tekstu naslova i linkova na stranicama, te u ponašanju nakon klika gumba “Dodaj”, odnosno “Spremi”. Klikom na te gumbove, korisnika se nakon dodavanja/spremanja promjena preusmjerava nazad



Slika 5.5: Chem Elements Index

na stranicu s popisom kemijskih spojeva.

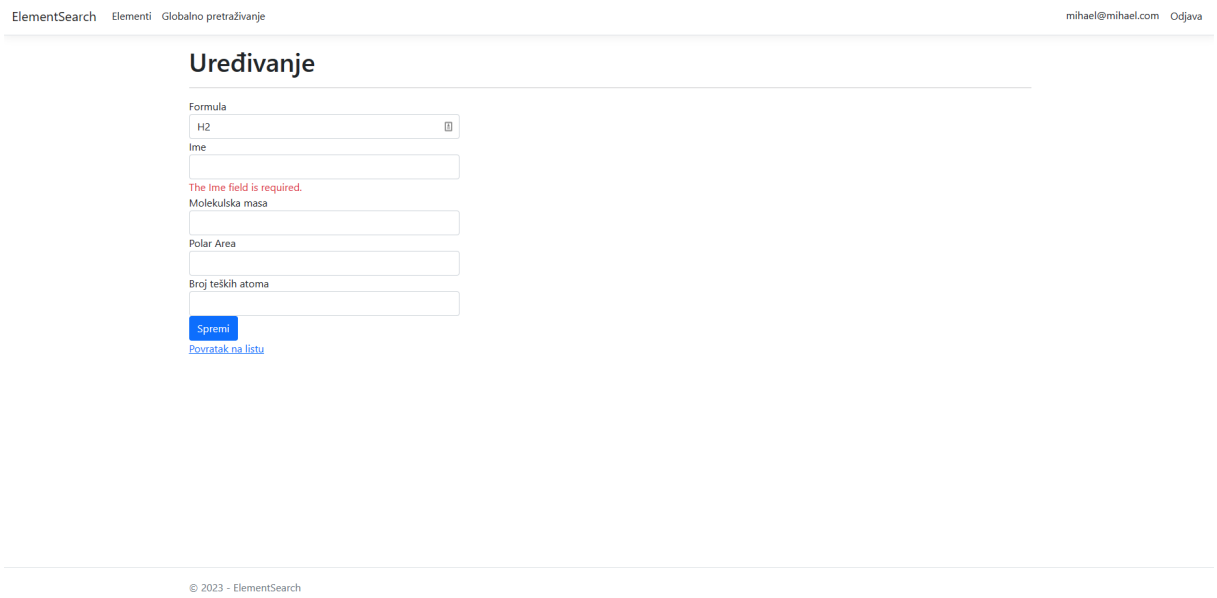


Slika 5.6: Chem Elements Create

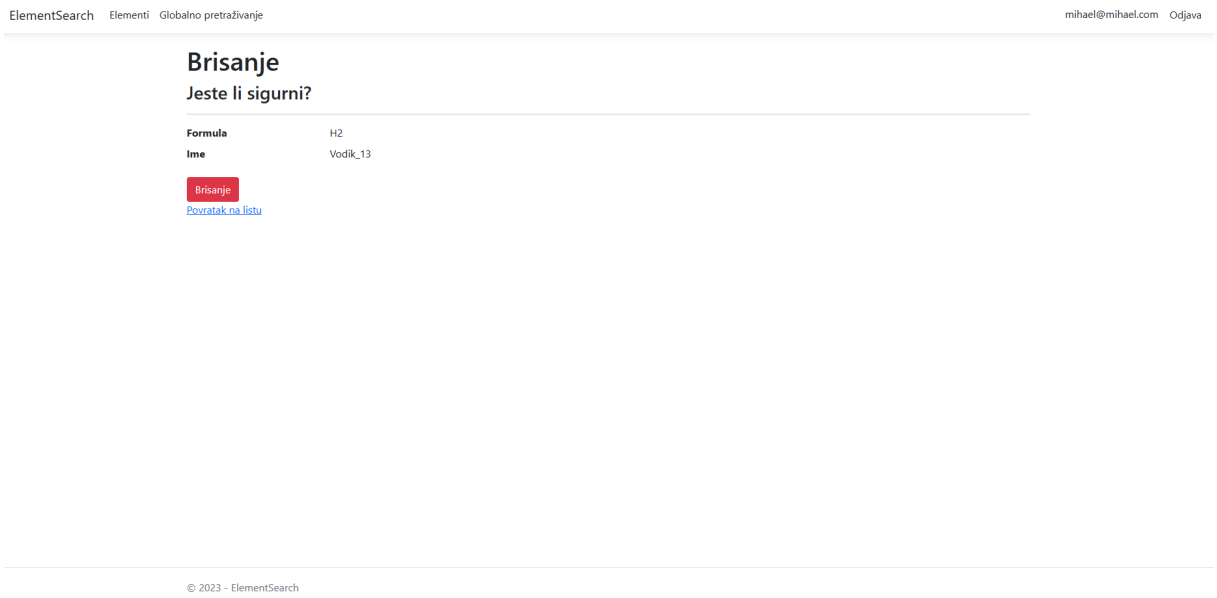
Osim toga, prilikom učitavanja stranice za uređivanje kemijskog spoja, polja za unos će već bit popunjena trenutnim vrijednostima tog polja u bazi podataka. Na sljedećoj slici je također pokazano i kako izgleda stranica nakon pokušaja spremanja promjena ako neka validacijska provjera nije zadovoljena - u ovom slučaju za polje “Ime”.

Stranica za brisanje kemijskog spoja služi kao potvrdni dijalog prije nepovratnog brisanja unosa iz baze podataka:

Kao što je već spomenuto u sekciji “Controller” za stranicu “Create”, tako i stranice “Edit” i



Slika 5.7: Chem Elements Edit

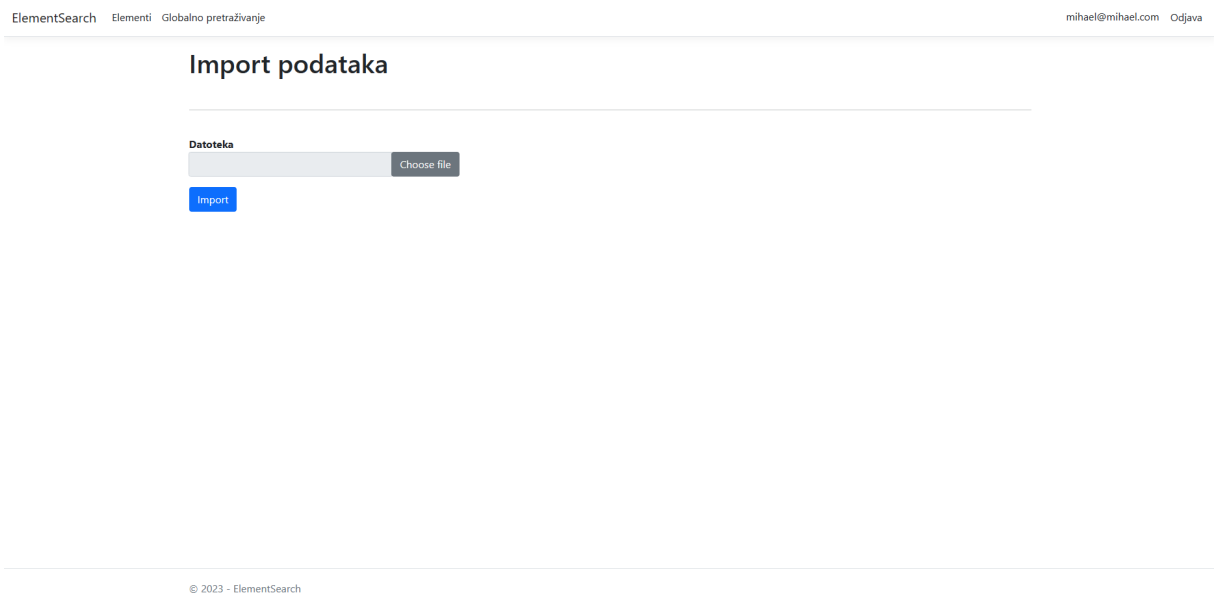


Slika 5.8: Chem Elements Delete

"Delete" u pozadini koriste dvije akcije - "Get" koja prikazuje stranicu, koja se poziva klikom na gumbove "Uređivanje" i "Brisanje", te akcije "Post" koje se pozivaju klikom na gumbove "Spremi", odnosno "Brisanje".

5.4 Import podataka

Na isti način funkcionira i stranica za import podataka, kojoj se pristupa sa ChemElements Index stranice, točnije stranice s popisom kemijskih spojeva, klikom na "Import podataka". Akcija koja poziva tu stranicu je radi organizacije koda odvojena u novi kontroler - `ImportDataController`, no iste akcije je moguće, i čak prihvatljivo zadržati u ChemElementsController-u jer se također odnosi na upravljanje entitetima tipa ChemElement.



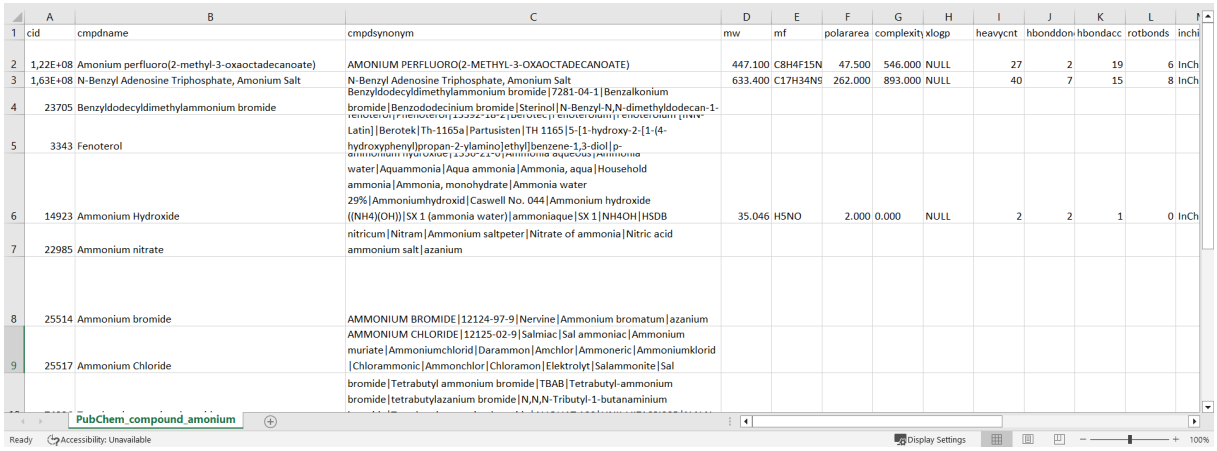
Slika 5.9: Chem Elements Import Data

Klikom na "Choose file" otvara se lokalni file explorer koji korisniku omogućuje odabir prikladne .csv datoteke za import podataka, a mogući je I Drag-and-Drop u polje na stranici. Klikom na "Import", serveru se šalje datoteka koju on dalje obrađuje - svaki redak u tablici, osim naravno zaglavlja, pretvara se u novi entitet tipa ChemElement i sprema u bazu. Naravno, server uvijek u istim stupcima očekuje iste podatke, zato je bitno da prilikom importa datoteka zadovoljava određeni format.

Trenutno se za import podataka koriste .csv datoteke preuzete s javne baze podataka o kemijskim spojevima PubChem [5]. Slijedeća slika prikazuje dio datoteke koju ElementSearch prihvaća za Import, vizualizirane u Excelu:

Sam import je ostvaren uz pomoć biblioteke CsvHelper, autora Josha Closea, na slijedeći način:

Prvo je potrebno definirati Post akciju koja prima csv file. to je ostvareno definiranjem ViewModela (VM) koji sadrži polje tipa IFormFile i bindanjem tog polja prilikom pristupa navedenoj



	A	B	C	D	E	F	G	H	I	J	K	L	M
1	cid	cmpdname	cmpdsynonym	mw	mf	polararea	complexity	xlogp	heavycnt	hbonddon	hbondacc	rotbonds	inchi
2	1,22E+08	Amonium perfluoro(2-methyl-3-oxaoctadecanoate)	AMONIUM PERFLUORO(2-METHYL-3-OXAOCTADECANOATE)	447.100	C8H4F15N	47.500	546.000	NULL	27	2	19	6	InCh
3	1,63E+08	N-Benzyl Adenosine Triphosphate, Amonium Salt	N-Benzyl Adenosine Triphosphate, Amonium Salt	633.400	C17H34N9	262.000	893.000	NULL	40	7	15	8	InCh
4	23705	Benzyldodecyldimethylammonium bromide	Benzyldodecyldimethylammonium bromide|7281-04-1|Benzalkonium bromide|Benzododecinium bromide|Sterinol|N-Benzyl-N,N-dimethyldodecan-1-										
5	3343	Fenoterol	Latin)|Berotek|Th-1165a|Partusisten|TH 1165|5-[1-hydroxy-2-[1-(4-hydroxyphenyl)propan-2-ylamino]ethyl]benzene-1,3-diol|p-										
6	14923	Ammonium Hydroxide	water|Aquammonia|Aqua ammonia|Ammonia, aqua|Household ammonia|Ammonia, monohydrate|Ammonia water 29%|Ammoniumhydroxid|Caswell No. 044|Ammonium hydroxide ((NH4)(OH))|SX 1 (ammonia water)|ammoniaque|SX 1|NH4OH|HSDB	35.046	H5NO	2.000	0.000	NULL	2	2	1	0	InCh
7	22985	Ammonium nitrate	nitricum|Nitram|Ammonium saltpeter|Nitrate of ammonia|Nitric acid ammonium salt|azanium										
8	25514	Ammonium bromide	AMMONIUM BROMIDE|12124-97-9|Nervine|Ammonium bromatum|azanium										
9	25517	Ammonium Chloride	AMMONIUM CHLORIDE|12125-02-9|Salmiac|Sal ammoniac|Ammonium muriate|Ammoniumchlorid|Darammon|Amchlor|Ammoneric|Ammoniumklorid |Chlorammonic|Ammonchlor|Chloramon|Elektrolyt|Salammonite|Sal										
			bromide|Tetrabutyl ammonium bromide|TBAB|Tetrabutyl-ammonium bromide|tetrabutylazanium bromide|N,N,N-Tributyl-1-butanaminium										

PubChem_compound_amonium

Ready | Accessibility: Unavailable | Display Settings | 100%

Slika 5.10: Sadržaj datoteke za import

Post akciji.

```
public class ImportDataVM
{
    public IFormFile File { get; set; }
}
```

```
[HttpPost]
public async Task<IActionResult> Import([Bind(nameof(ImportDataVM.File))]
                                        ImportDataVMvm)
{
}
```

IFormFile se u ASP.NET-u koristi za slanje i primanje datoteka općenito. Sada kada imamo tu datoteku u kontroleru, potrebno ju je pročitati kao .csv datoteku. Iz spomenute biblioteke CsvHelper koriste se tri klase. `CsvReader` čita csv datoteku imajući u obziru konfiguraciju koja mu je proslijeđena u obliku klase `CsvConfiguration`. Treća korištena klasa je `ClassMap` za tip ChemElement. Definiramo klasu koja nasljeđuje od klase `ClassMap<ChemElement>` kako bismo tu mapu mogli koristiti za mapiranje stupaca u datoteci na odgovarajuće polje u objektu tipa ChemElement.

```
public class ChemElementCSVMap : ClassMap<ChemElement>
{
    public ChemElementCSVMap()
    {
        Map(p => p.Name).Index(1);
        Map(p => p.MolecularWeight).Index(3);
        Map(p => p.Formula).Index(4);
        Map(p => p.PolarSurfaceArea).Index(5);
        Map(p => p.HeavyAtomCount).Index(8);
```

```
    }
}
```

Definiramo konstruktor u kojem za svako polje u klasi ChemElement definiramo indeks stupca koji u kojem će se tražiti podatak.

Sada u Import akciji sve navedeno povezujemo na slijedeći način:

```
var config = new CsvConfiguration(CultureInfo.InvariantCulture);
config.Delimiter = ",";
config.HasHeaderRecord = true;

var csv = new CsvReader(textReader, config);
csv.Context.RegisterClassMap<ChemElementCSVMap>();
```

Prvo definiramo konfiguraciju u kojoj specificiramo da želimo Invariant Culture. To nam je bitno jer će u datotekama za import decimalni separator biti točka. Zatim definiramo da je očekivani delimiter zarez i da će ulazna datoteka imati header. Zatim definiramo CsvReader na osnovu konfiguracije i sadržaja datoteke. Varijablu textReader ovdje možemo promatrati kao sadržaj zaprimljene datoteke koji je prethodno obrađen da bi ga konstruktor klase CsvReader mogao pročitati. Prije čitanja same datoteke potrebno je u readeru registrirati definiranu mapu kako bi znao na koji način pročitati datoteku.

Sada datoteku čitamo linijom:

```
var records = csv.GetRecords<ChemElement>().ToList();
```

Nakon koje u varijabli records imamo sve redke iz datoteke u obliku objekata tipa ChemElement. Iz te liste se zatim uklone duplikati na osnovu formule i imena te se ubace u bazu podataka.

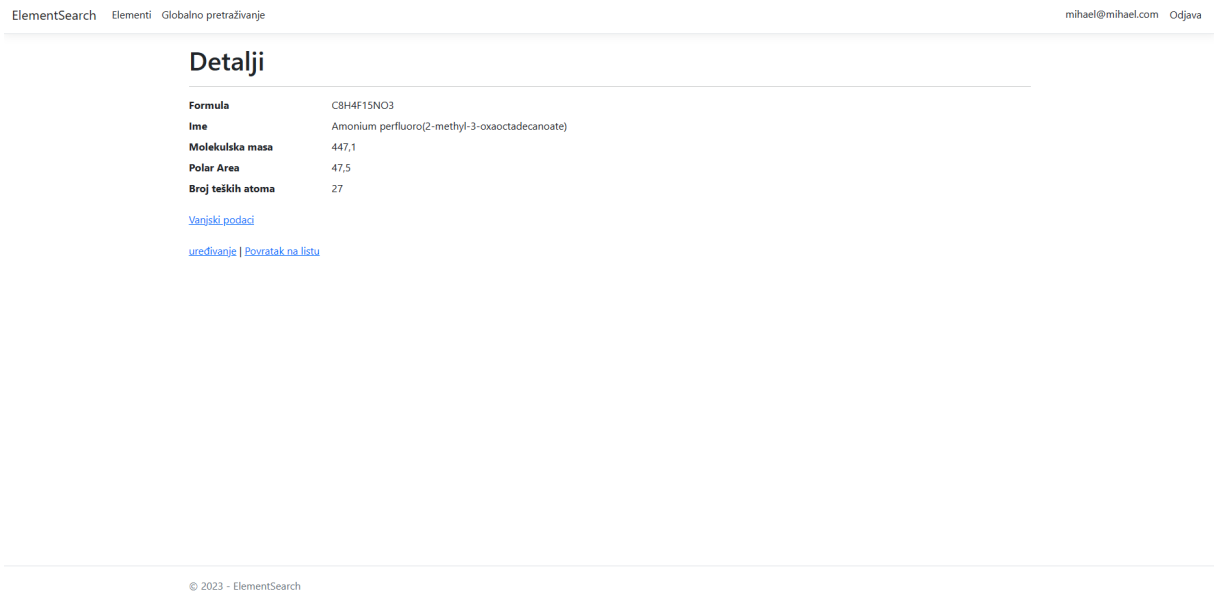
5.5 Pretraživanje suradničkih podataka za određeni element

Jednostavnija od dviju funkcionalnosti pretraživanja suradničkih podataka pretražuje podatke vezane za određeni element. Ovoj funkcionalnosti se pristupa preko stranice detalja za određeni kemijski element, koja izgleda ovako:

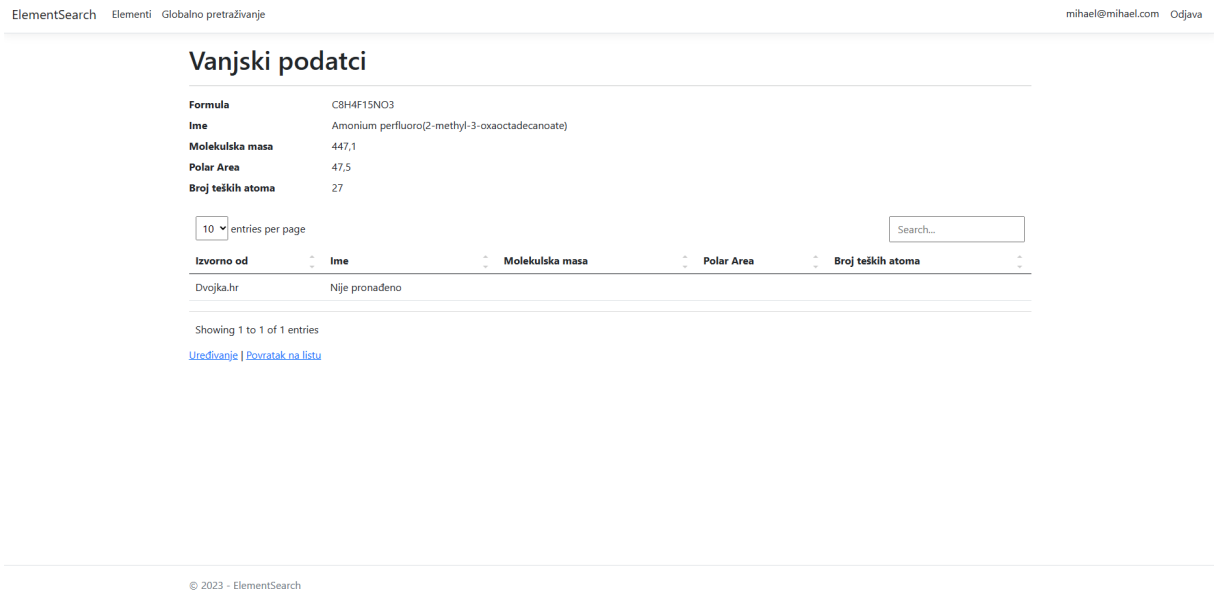
Klikom na “Vanjski podaci” prikazati će se istoimena stranica, koja je identična stranici “Detalji”, osim što se na njoj dodatno i prikazuje tablica podataka dohvaćenih od suradničkih stranica koje su spremljene u konfiguraciji aplikacije (appsettings.json)

U ovoj tablici se nalazi 1 redak jer je trenutno u konfiguraciji spremljena samo jedna adresa, koja je u kontekstu razvojnog okruženja localhost adresa na Docker kontejner čije postavljanje ćemo detaljnije promotriti kasnije u radu.

Sekcija u *appsettings.json* datoteci koja definira adrese suradničkih ElementSearch instanci izgleda ovako:



Slika 5.11: ChemElement Details



Slika 5.12: Vanjski podatci elementa

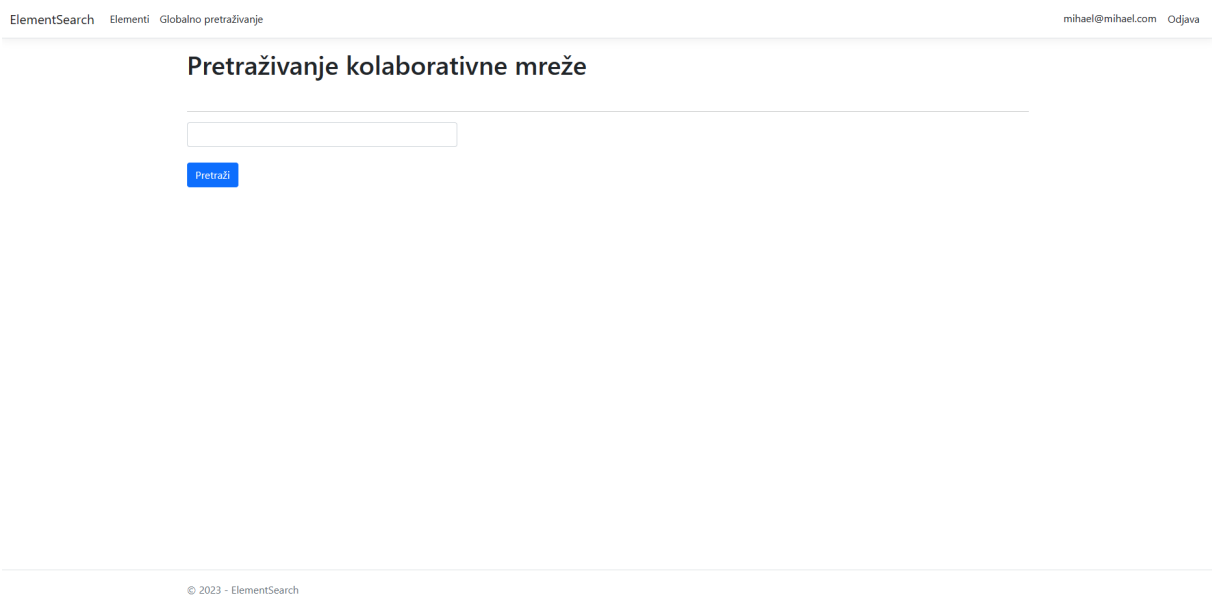
```
"ExternalSources": [
  "http://localhost:32781"
],
```

U stupcu “Izvorno od” prikazuju se imena web stranica koje su vratile podatke za taj redak - isto ime koje se sprema u konfiguraciji i prikazuje na početnoj stranici web aplikacije - u našem primjeru to je “Kemija.hr”.

Zahtjev koji se šalje na suradničke instance sadrži formulu kemijskog spoja. Suradničke stranice na osnovu zaprimljene formule pretražuju bazu podataka i, ukoliko nađu podatke, šalju odgovor s vlastitim imenom stranice i podatcima, ili, ukoliko ne pronađu odgovarajući kemijski spoj, šalju odgovor s imenom stranice i “Nije pronađeno”, kao stranica “Dvojka.hr” iz primjera iznad.

5.6 Globalno pretraživanje suradničkih podataka

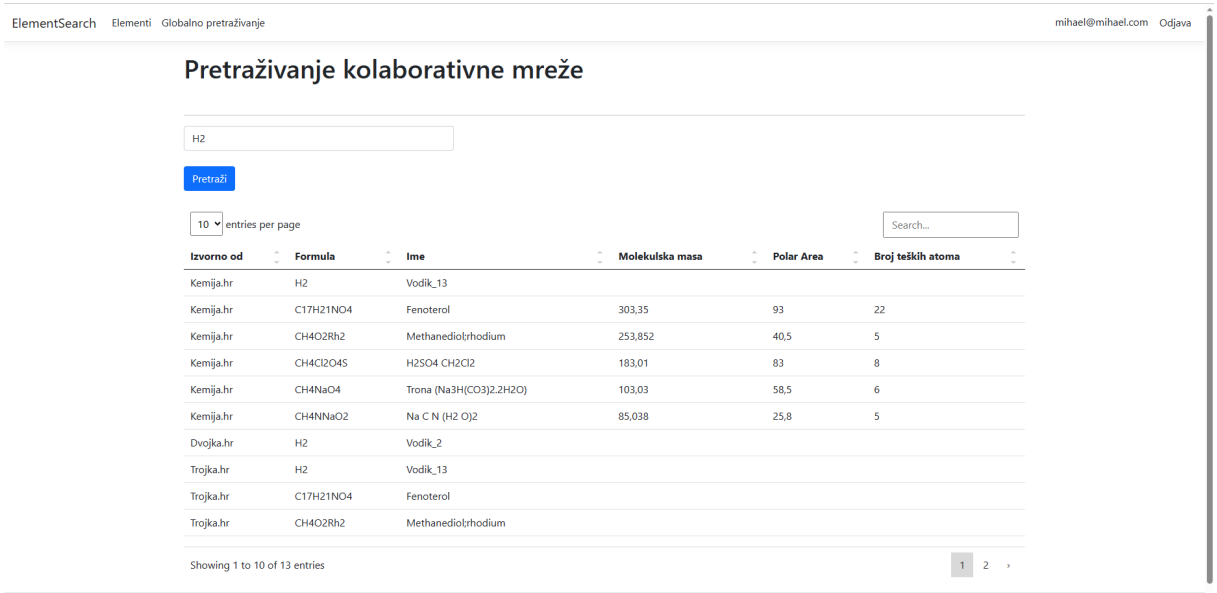
Globalno pretraživanje funkcionira na način da se suradničke stranice zatraži rezultat pretrage za uneseni *string*. Rezultati se vraćaju ne samo na osnovu formule, već na osnovu svih atributa spremljenih u tablici kemijskih spojeva. Dodatno, kada suradničke datoteke zaprime upit za globalno pretraživanje, osim što će vratiti lokalno dobivene rezultate, zatražiti će i vlastite suradničke instance ElementSearch aplikacije za njihove rezultate. Takvim pretraživanjem u dubinu se uistinu postiže globalno pretraživanje kolaborativne mreže.



Slika 5.13: Global Search interface

Sada, ako želimo pronaći sve elemente koji u formuli sadrže “H2”, možemo upisati “H2” u tražilicu i pretražiti mrežu suradničkih ElementSearch instanci.

Ovdje možemo vidjeti da, iako “Dvojka.hr” nema puno podataka o traženim kemijskim spojevima, dobivamo i rezultate od lokacije “Trojka.hr”, iako ju nemamo spremljenu u konfiguraciji



Slika 5.14: Global Search interface

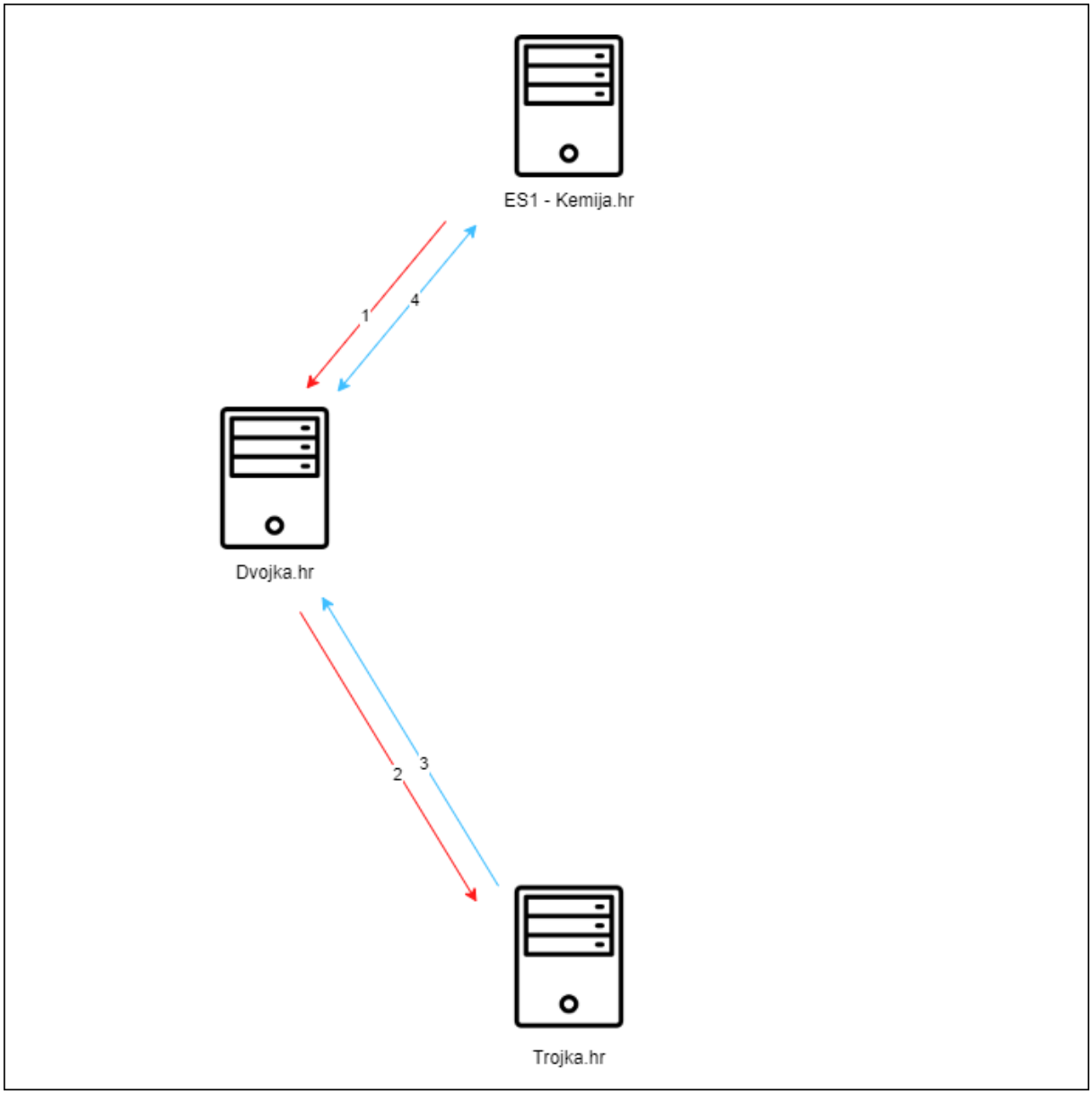
naše ElementSearch instance. Te podatke dobili smo posredno preko Dvojke, koja je Trojki poslala isti upit koji je Dvojka zaprimila od Kemije.

No, što ako Trojka.hr ima Kemiju ili Dvojku u konfiguraciji kao suradničku lokaciju? Nastaje beskonačni krug zahtjeva koji nikada korisniku neće vratiti njegov zahtjev. Za izbjegavanje takvog slučaja implementirane su dvije funkcionalnosti - praćenje ID-jeva stranica i praćenje dubine zahtjeva.

Obje vrijednosti se nalaze u sekciji *Site* u *appsettings.json*, kako je spomenuto u sekciji "Stranice za autentikaciju i početna stranica". Uz *string* za pretraživanje, u zahtjevu za globalnim pretraživanjem stranica šalje kolaborativnoj stranici i te dvije vrijednosti.

Zadržimo se na primjeru sa sjedištima Kemija, Dvojka i Trojka, ali uzimo da Trojka u konfiguraciji ima zapisanu Kemiju. Kemija će Dvojci poslati zahtjev za pretraživanjem i svoj *ID* unutar liste. Dvojka će prvo provjeriti nalazi li se njen *ID* među zaprimljenim *ID*-jevima. Ukoliko se nalazi, bez daljnjih upita će vratiti praznu listu. Pošto se u ovom slučaju u listi *ID*-jeva nalazi samo *ID* Kemije, Dvojka će prikupiti relevantne podatke koje ima u svojoj bazi podataka, dodati svoj *ID* u listu *ID*-jeva i proslijediti upit Trojci. Trojka također prikuplja svoje podatke pošto ne pronalazi svoj *ID* u dobivenoj listi. Sada Trojka dodaje svoj *ID* u listu i šalje upit Kemiji, koja pronalazi svoj *ID* na listi i, umjesto nastavljanja beskonačnog kruga Kemija-Dvojka-Trojka, Trojci odmah vraća praznu listu kemijskih spojeva. Trojka nakon zaprimljenog odgovora Dvojci vraća svoje podatke, koja Kemiji vraća spojene podatke zajedno s informacijom otkud dolaze podatci.

Implementacijom provjere *ID*-jeva zaustavljaju se kružni upiti. No u praksi mreža povezanih sjedišta može sezati u velike lance poziva, što korisniku može stvoriti veliko vrijeme čekanja. Kako bi se to izbjeglo, u zahtjevu između sjedišta se šalje i parametar dubine ("Depth"), koji



Slika 5.15: Prikaz komunikacije na primjeru tri sjedišta

definira maksimalan broj zahtjeva u dubinu. Koristi se na način da sjedište od kojeg originalno seže zahtjev pošalje svim svojim suradničkim sjedištima vrijednost dubine koje sadrži u dokumentaciji. Kada prosljeđuju zahtjev, ta sjedišta smanje vrijednost dubine za 1. Ako se nastavi izvođenje zahtjeva do definirane dubine, neko sjedište će zaprimiti zahtjev s vrijednošću dubine 0. Ukoliko je zaprimljena vrijednost dubine 0, sjedište vraća praznu listu kemijskih spojeva, bez daljnjeg proslijeđivanja zahtjeva.

Na primjer, kada bi vrijednost "Depth" na sjedištu Kemija.hr bila konfigurirana na 0, Trojka.hr bi vratila praznu listu jer pretraživanje seže samo jedan zahtjev u dubinu. Dobiveni rezultati bi bili samo od Dvojke. U praksi, možemo pretpostaviti da bi vrijednost "Depth" najčešće bila postavljena između 10 i 20.

Poglavlje 6

Postavljanje lokalne mreže ElementSearch instanci Koristeći Docker Desktop

Docker Container je standardna jedinica softvera koja sprema aplikacijski kod i sve njegove zavisnosti i potrebne datoteke na jedno mjesto, kako bi se olakšalo postavljanje i pokretanje usluga na računalnom sustavu [6]. Docker image je skup softvera i ostalih datoteka koje neki container treba za rad. Svaki container ima svoj image, a jedan image može biti korišten za pokretanje više containera. Najjednostavnije rečeno image postaje container prilikom pokretanja.

Pokretanje lokalnih containera znatno je pojednostavljeno korištenjem Docker Desktop grafičkog sučelja. Koje je prethodno potrebno instalirati. Također, lokalno na računalu (alternativno može i na udaljenom serveru) je potrebno imati PostgreSQL instaliran, jer ElementSearch instance koriste PostgreSQL baze podataka.

Prvi korak je u naredbenom retku otvoriti direktorij projekta. To se može napraviti bez izlaska iz Visual Studia, klikom na View > Terminal. U otvorenom Terminalu, prvo je potrebno izgraditi image. To radimo naredbom:

```
docker build -t elementsearch .
```

Ukoliko je izgradnja uspješna, možemo nastaviti s postavljanjem containera. Pošto će nastali containeri međusobno komunicirati, potrebno ih je prilikom stvaranja smjestiti u istu mrežu. Prvo stvaramo novu mrežu naredbom:

```
docker network create elementSearch-network
```

Zatim možemo pokrenuti naše containere naredbom `docker run`. parametrom `-p` definiramo kojim vanjskim portom pristupamo containeru i na koji unutarnji port se zahtjev preslikava. Parametrom `--network` definiramo kojoj će mreži novi container pripadati. Na samom kraju Dockeru dajemo ime imagea na kojem će se zasnivati novi container.

Za tri sjedišta iz prethodnih primjera, naredbe mogu izgledati ovako:

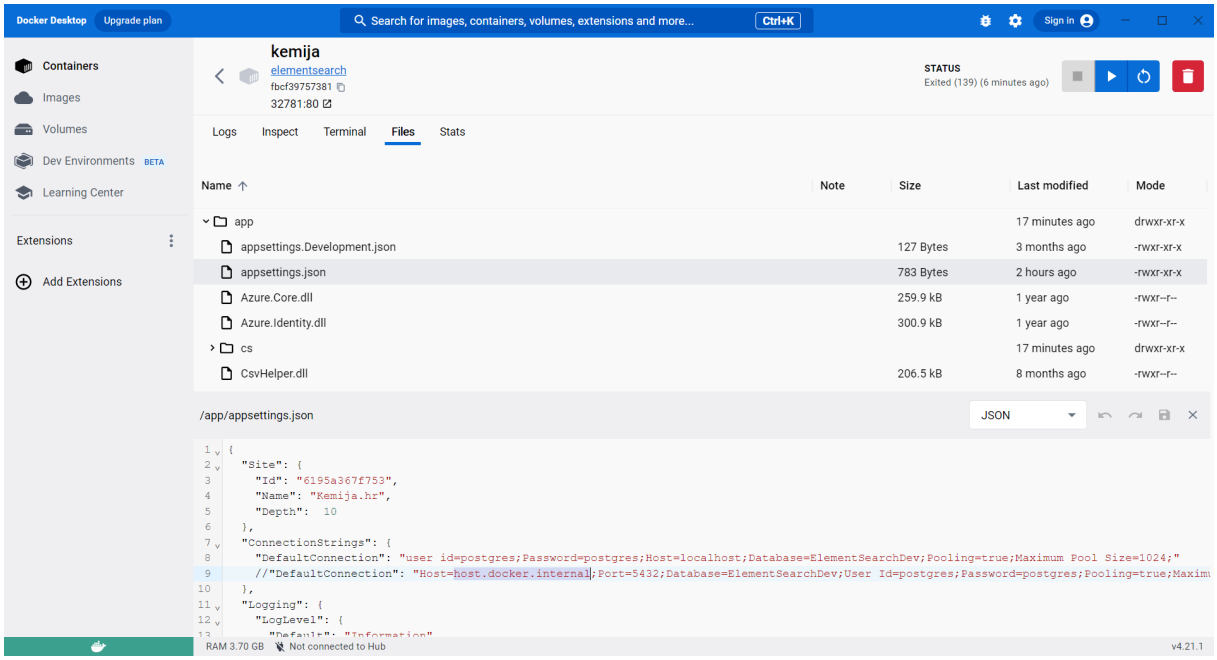
```
PS C:\Users\mladi\Documents\Faks\Zavrsni\ElementSearch\ElementSearch>
   docker run -d -p 32781:80 --name kemija
```

```
    --network elementSearch-network elementsearch

PS C:\Users\mladi\Documents\Faks\Zavrsni\ElementSearch\ElementSearch>
    docker run -d -p 32782:80 --name dvojka
    --network elementSearch-network elementsearch

PS C:\Users\mladi\Documents\Faks\Zavrsni\ElementSearch\ElementSearch>
    docker run -d -p 32783:80 --name trojka
    --network elementSearch-network elementsearch
```

Naravno, potrebno je za svaki container definirati različite brojeve portova, inače će Docker javljati grešku.

Ako odemo u Docker Desktop sučelje, vidjet ćemo da novi containeri nisu pokrenuti. To je zato što se nisu uspješno podigli zbog neispravne konfiguracije. Containeri ne prepoznaju localhost kao mašinu domaćina, već kao lokaciju unutar samog containera. Kako bi se ispravno referencirale baze podataka potrebno je umjesto "localhost" upisati "host.docker.internal". Što se tiče konfiguracije vanjskih izvora, containeri unutar iste mreže se prepoznaju po nazivu containera. Potrebno je specificirati i ispravni port (u našem slučaju 80).

Podatke unutar containera se u Docker Desktop sučelju uređuju u sekciji "Files". Datoteka koju je potrebno izmijeniti je appsettings.json, koji se nalazi u folderu "app".



Slika 6.1: Docker Desktop

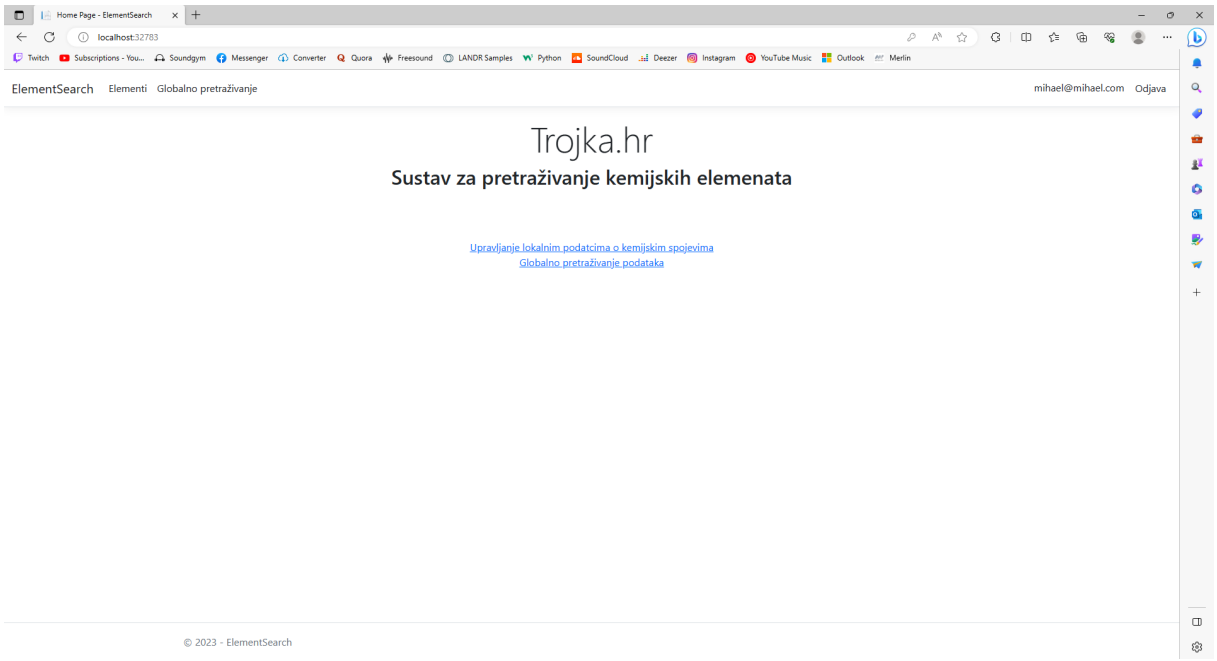
Slijede dijelovi koji su u ovom primjeru izmijenjeni:

```
  "Site": {
    "Id": "kemija",
```

```
    "Name": "Kemija.hr",
    "Depth":  10
  },
  "ConnectionStrings": {PS
  "ExternalSources": [
    "http://dvojka:80"
  ],
  "Site": {
    "Id": "dvojka",
    "Name": "Dvojka.hr",
    "Depth":  10
  },
  "ConnectionStrings": {
    "DefaultConnection": "Host=host.docker.internal;Port=5432;Database=dvojka;
    User Id=postgres;Password=postgres;Pooling=true;Maximum Pool Size=1024;"
  },

  "ExternalSources": [
    "http://trojka:80"
  ],
  "Site": {
    "Id": "trojka",
    "Name": "Trojka.hr",
    "Depth":  10
  },
  "ConnectionStrings": {
    "DefaultConnection": "Host=host.docker.internal;Port=5432;Database=trojka;
    User Id=postgres;Password=postgres;Pooling=true;Maximum Pool Size=1024;"
  },

  "ExternalSources": [
    "http://kemija:80"
  ],
```

Ono što vrijedi napomenuti je da će u praksi Site ID-jevi biti malo kompleksnije šifre, poput GUID-a, a ovdje su radi jednostavnosti postavljene na imena sjedišta. Također, ElementSearch je napravljen da prilikom pokretanja primijeni sve migracije koje dotad nisu bile primijenjene metodom `Migrate()`. Zato, ako je server dostupan, ne moramo brinuti o prethodnom postojanju baza podataka.

Sada možemo sjedištima pristupiti preko definiranih portova (32781, 32782, 32783).



Slika 6.2: Trojka.hr

Poglavlje 7

Zaključak

U sklopu ovog završnog rada razvijena je Web aplikacija formata MVC u razvojnom okviru ASP.NET. Predstavlja jednostavnu platformu na kojoj znanstvene institucije mogu spremati i dijeliti podatke o kemijskim spojevima, zvanu ElementSearch. Iako sam zadovoljan trenutnom verzijom ElementSearcha, vidljivo je da postoji mnogo smjerova u kojima se aplikacija može dalje razvijati. Neki od njih su dijeljenje i pretraživanje znanstvenih radova, implementacija administracije korisnika, proširivanje kemijskih spojeva s novim detaljima, itd. U praksi ElementSearch je namijenjen znanstvenim institucijama, gdje se općenito cijene vrijednosti međusobne suradnje.

Literatura

[1] A. Wiggins, "The Twelve-Factor App." 2017. Available: https://12factor.net/. [Accessed: Jul. 15, 2023]

[2] "ASP.NET | Open-source web framework for .NET," *Microsoft*. Available: https://dotnet.microsoft.com/en-us/apps/aspnet. [Accessed: Jul. 15, 2023]

[3] "What is ASP.NET? | .NET," *Microsoft*. Available: https://dotnet.microsoft.com/en-us/learn/aspnet/what-is-aspnet. [Accessed: Jul. 15, 2023]

[4] Rick-Anderson, "Dependency injection in ASP.NET Core." May 2023. Available: https://learn.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-7.0. [Accessed: Jul. 15, 2023]

[5] PubChem, "PubChem." Available: https://pubchem.ncbi.nlm.nih.gov/. [Accessed: Jul. 16, 2023]

[6] "What is a Container? | Docker." Nov. 2021. Available: https://www.docker.com/resources/what-container/. [Accessed: Jul. 16, 2023]