

Izrada web sustava za nadzor prisutnosti i praćenje kvalitete zraka u zatvorenim prostorima

Kranjec, Ivan

Undergraduate thesis / Završni rad

2021

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **Bjelovar University of Applied Sciences / Veleučilište u Bjelovaru**

Permanent link / Trajna poveznica: <https://um.nsk.hr/um:nbn:hr:144:144134>

Rights / Prava: [In copyright](#) / [Zaštićeno autorskim pravom.](#)

Download date / Datum preuzimanja: **2025-01-03**



Repository / Repozitorij:

[Repository of Bjelovar University of Applied Sciences - Institutional Repository](#)



VELEUČILIŠTE U BJELOVARU
PREDDIPLOMSKI STRUČNI STUDIJ RAČUNARSTVO

**Izrada web sustava za nadzor prisutnosti i praćenje kvalitete
zraka u zatvorenim prostorima**

Završni rad br. 01/RAČ/2021

Ivan Kranjec

Bjelovar, srpanj 2021.



Veleučilište u Bjelovaru
Trg E. Kvaternika 4, Bjelovar

1. DEFINIRANJE TEME ZAVRŠNOG RADA I POVJERENSTVA

Kandidat: **Kranjec Ivan**

Datum: 25.05.2021.

Matični broj: 001975

JMBAG: 0314019280

Kolegij: **.NET PROGRAMIRANJE**

Naslov rada (tema): **Izrada web sustava za nadzor prisutnosti i praćenje kvalitete zraka u zatvorenim prostorima**

Područje: **Tehničke znanosti**

Polje: **Računarstvo**

Grana: **Programsko inženjerstvo**

Mentor: **Krunoslav Husak, dipl.ing.rač.**

zvanje: **predavač**

Članovi Povjerenstva za ocjenjivanje i obranu završnog rada:

1. Tomislav Adamović, mag.ing.el., predsjednik
2. Krunoslav Husak, dipl.ing.rač., mentor
3. Ivan Sekovanić, mag.ing.inf.et comm.techn., član

2. ZADATAK ZAVRŠNOG RADA BROJ: 01/RAČ/2021

U radu je potrebno izraditi i opisati web sustav koji će prikupljati podatke s uređaja koji u određenom zatvorenom prostoru nadziru prisutnost osoba i prate kvalitetu zraka. Sustav mora omogućiti prikupljanje podataka s više uređaja i sve prikupljene podatke skladištiti u bazu podataka. Web aplikacija ovog sustava mora omogućiti jednostavni pregled stanja u svim zatvorenim prostorima gdje se nalaze uređaji za nadzor, kao i pregled povijesti svih vrijednosti.

Zadatak uručen: 25.05.2021.

Mentor: **Krunoslav Husak, dipl.ing.rač.**



Sadržaj

1.	Uvod	1
2.	ASP.NET Core.....	2
2.1	<i>MVC arhitektura.....</i>	<i>3</i>
2.1.1	Kontroler	4
2.1.2	Model	5
2.1.3	Pogled.....	6
2.2	<i>Autentikacija.....</i>	<i>8</i>
2.2.1	ASP.NET Core Identity	8
2.3	<i>Baza podataka</i>	<i>11</i>
2.3.1	PostgreSQL	12
2.3.2	Komunikacija web aplikacije sa bazom podataka.....	14
2.3.3	Migracije	18
2.3.4	Provjera zdravlja baze podataka	20
2.4	<i>Vizualizacija podataka</i>	<i>24</i>
2.4.1	Prikaz podataka na aplikaciji.....	25
3.	Docker	28
3.1	<i>Kontejnerizacija.....</i>	<i>30</i>
3.1.1	Docker Compose	30
3.1.2	Dockerfile.....	31
4.	ZAKLJUČAK.....	33
5.	LITERATURA	35
6.	OZNAKE I KRATICE.....	36
7.	SAŽETAK.....	37
8.	ABSTRACT	38

9. PRILOZI.....	39
------------------------	-----------

1. Uvod

Ovaj završni rad obrađuje teme kontejnerizacije, razvoja web aplikacije u ASP.NET Core MVC tehnologiji, implementacije sustava autentikacije na web aplikaciji, te upravljanju i prikazu podataka.

Prvi dio rada je fokusiran na samu web aplikaciju, primjenu MVC oblikovnog obrasca, autentikaciju pomoću ASP.NET Core Identity rješenja, komunikaciju web aplikacije sa bazom podataka, provjeru ispravnog stanja baze podataka te vizualizaciju podataka na web aplikaciji.

Drugi dio rada je orijentiran na kontejnerizaciju web aplikacije u programskom okruženju Docker, te se objašnjavaju osnovni principi kontejnerizacije.

Krajnji cilj je ostvariti web sustav koji prikuplja informacije o kvaliteti zraka i prisutnosti u zatvorenoj prostoriji. Podaci se spremaju u bazu podataka, a web aplikacija prikazuje te podatke u obliku grafova.

2. ASP.NET Core

ASP.NET Core je programski okvir (engl. *framework*) otvorenog koda za realizaciju programskih rješenja koja su neovisna o operacijskom sustavu. Često se koristi za izradu web aplikacija ili samo web API¹-a. Omogućuje izradu web aplikacija sa fokusom na pisanje programskog koda u programskom jeziku C# uz podršku programskog jezika JavaScript.

Zbog toga što je ASP.NET Core programski okvir otvorenog koda, iza sebe ima znatnu zajednicu koja konstantno radi na novim poboljšanjima te aktivno sudjeluje u prijenosu znanja.

Ključna značajka ASP.NET Core programskog okvira su performanse što ga ujedno i svrstava u sam vrh u odnosu na ostale web programske okvire.



Slika 2.1: Logo ASP.NET Core programskog okvira

Razvojno okruženje Visual Studio uz odabir ASP.NET Core programskog okvira omogućuje automatsku integraciju raznih web programskih okvira i biblioteka poput React.js ili Angular.js, što pruža širi spektar tehnologija za razvoj web aplikacije. Uz to, omogućena je direktna podrška za kontejnerizaciju putem Docker platforme.

¹ Web API je sučelje koje ima izložene metode čiji je zadatak upravljanje skupinom entiteta nad nekom bazom podataka pomoću HTTP zahtjeva.

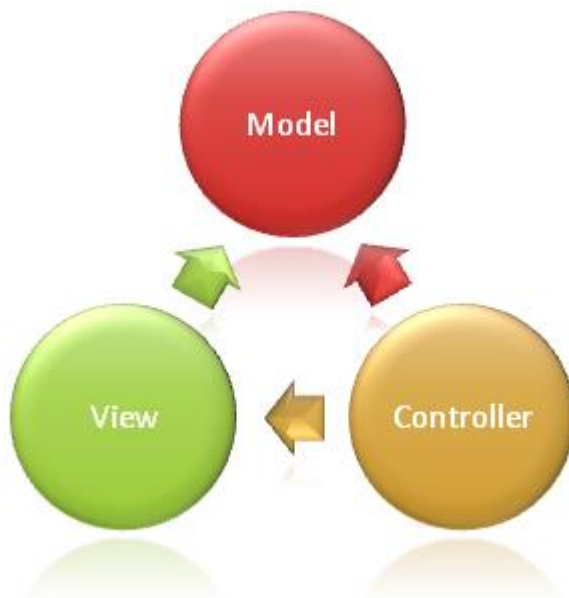
2.1 MVC arhitektura

Oblikovni obrasci predstavljaju načine rješavanja programskih izazova po predefiniranim koracima. Korištenjem oblikovnih obrazaca primjenjujemo pravila najbolje prakse što omogućuje lakše testiranje, ponovu uporabu i izbjegavanje grešaka u programskom kodu.

Web aplikacija opisana u ovome radu se temelji na Mode-Pogled-Kontroler (engl. *Model-View-Controller*) oblikovnom obrascu. MVC oblikovni obrasci služe kako bi se razdvojila ovisnost između korisničkog sučelja, podataka koji se prikazuju na korisničkom sučelju i poslovne logike.

Koristeći navedeni oblikovni obrazac, sve zahtjeve preusmjeravamo na pripadajući kontroler koji je odgovoran za vraćanje podataka na pogled. Na pogledu se podaci prikazuju pomoću odgovarajućeg modela.

Kontroler sam po sebi ne smije nositi cijelu poslovnu logiku, već to za njega radi odgovarajući servis². Ovisno o akciji, u kontroleru pozivamo servis, i na taj način pravilno razdvajamo poslovnu logiku pri korištenju MVC oblikovnog obrasca.



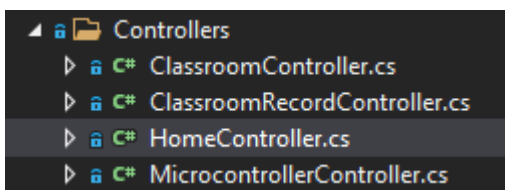
Slika 2.2: Ilustracija MVC oblikovnog obrasca

² Servis se odnosi na klasu u kojoj je napisana poslovna logika pripadajućeg kontrolera. Time ostvarujemo čitkost programskog koda u kontroleru, te odgovornost za komunikaciju sa bazom podataka prebacujemo na odgovarajući servis.

2.1.1 Kontroler

Kod MVC arhitekture, ključno je vratiti ispravne podatke na pogled. Za to nam služi kontroler. Kontroler se brine za dostavljanje ispravnih podataka na odgovarajući pogled. Također je zadužen za obradu podataka sa korisničke strane, te razne pozive prema bazi. Za svaku metodu u kontroleru, mora postojati pogled.

Po dogovoru, kontroler klase se nalaze u glavnom direktoriju projekta, u mapi Controllers. Naziv svake kontrolera mora završavati na „Controller“.



Slika 2.3: Prikaz pravilnog imenovanja kontroler klasa

Dobra praksa je da svaka kontroler klasa u sebi ima barem dva svojstva. DbContext³ klasu koja sadrži pripadajuće modele i služi za direktnu komunikaciju sa bazom podataka, te pripadajući servis.

```
public class HomeController : Controller {
    private readonly VaqWebDbContext _dbContext;
    private readonly ClassroomRecordService _classroomRecordService;

    0 references
    public HomeController(VaqWebDbContext dbContext) {
        _dbContext = dbContext;
        _classroomRecordService = new ClassroomRecordService(_dbContext);
    }

    [Authorize]
    0 references
    public IActionResult Index() {
        return View(_classroomRecordService.GetClassroomRecordsForCards());
    }

    [ResponseCache(Duration = 0, Location = ResponseCacheLocation.None, NoStore = true)]
    0 references
    public IActionResult Error() {
        return View(new ErrorViewModel { RequestId = Activity.Current?.Id ?? HttpContext.TraceIdentifier });
    }
}
```

Slika 2.4: Prikaz HomeController kontrolera

³ DbContext klasa omogućuje interakciju sa bazom podataka. Koristi se primjerice za dodavanje, brisanje, ažuriranje ili neku vrstu kompleksnijeg upita nad nekim modelom.

Programski kod na slici 2.4 predstavlja HomeController kontroler. Njegova zadaća je vraćanje podataka na pogled koji služi kao glavna stranica u web aplikaciji. U trenutku pokretanja aplikacije, u slučaju uspješne autentikacije, prikazuje nam se Index pogled.

2.1.2 Model

Glavna zadaća modela u MVC oblikovnom obrascu je prikaz podataka koji su uglavnom dobiveni iz nekog kontrolera. Model bi trebao vrlo dobro definirati podatke koje očekujemo vidjeti na pogledu, kada nam kontroler vrati podatke.

```
public class ClassroomRecord {
    [Key]
    0 references
    public int Id { get; set; }

    [Required]
    2 references
    public decimal Temperature { get; set; }

    [Required]
    2 references
    public decimal Humidity { get; set; }

    [Required]
    2 references
    public decimal AirQuality { get; set; }

    [Required]
    2 references
    public bool PresenceDetected { get; set; }

    9 references
    public DateTime CreatedAt { get; set; }

    // Properties that define relationship with Microcontroller model.
    0 references
    public int MicrocontrollerId { get; set; }

    9 references
    public Microcontroller Microcontroller { get; set; }
}
```

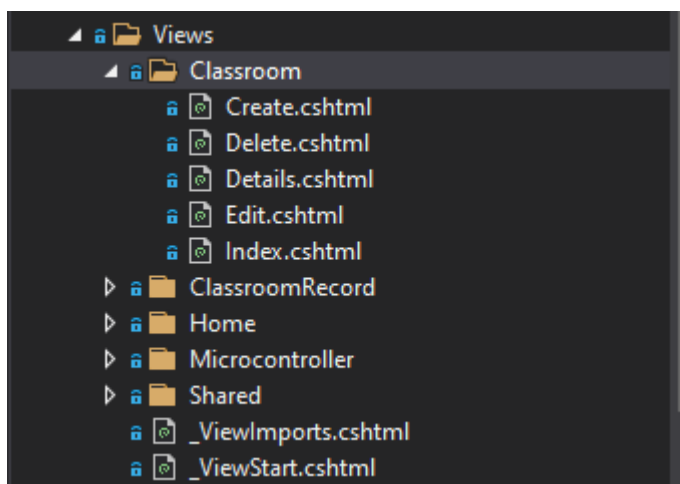
Slika 2.5: Prikaz ClassroomRecord modela

Na slici 2.5 nalazi se ClassroomRecord model koji sadrži informacije o trenutnim uvjetima u nekoj prostoriji. Imamo klasičnu numeričku jedinstvenu oznaku, podatke o temperaturi, vlažnosti, kvaliteti zraka, prisutnosti u prostoriji, vrijeme stvaranja zapisa, te vezu na Microcontroller model. U ovom slučaju zadnja dva svojstva definiraju vezu

ClassroomRecord i Microcontroller modela. Veza je definirana tako da jedan ClassroomRecord model mora biti povezan sa samo jednim Microcontroller modelom.

2.1.3 Pogled

Pogled je zadužen za prikaz podataka i interakciju korisnika. Zapravo se radi o HTML dokumentu sa ugrađenom Razor⁴ sintaksom.



Slika 2.6: Prikaz grupiranja pogleda u projektu

Za svaki kontroler postoji odgovarajuća mapa u kojoj se nalazi grupa pogleda. Primjerice, ClassroomController je predstavljen prema korisniku pomoću Classroom grupe pogleda.

Vrlo bitne komponente kod pogleda su Layout i Partial⁵ pogledi. Zadatak Layout pogleda je konzistentno prikazivanje sadržaja, dodavanje linkova na skripte i css datoteke kako bi se izbjeglo višestruko ponavljanje programskog koda u ostalim pogledima. Naime, Partial pogled ima ponešto drugačiji zadatak. Cilj korištenja Partial pogleda je iskorištavanje Partial pogleda poput predloška više puta, ovisno o potrebi.

Postoji više načina na koje možemo proslijediti podatke u pogled. Jedan od načina je navesti model na samom početku pogleda, drugi način je koristiti ViewData ili ViewBag objekte kojima se može pristupiti u pogledu. Preporučeni način je definirati model na početku pogleda jer u tom slučaju imamo potpunu podršku IntelliSense⁶ značajke u Visual

⁴ Razor sintaksa se koristi pri stvaranju dinamičnih web stranica koristeći programski jezik C#.

⁵ Partial pogled je pogled koji se može iskoristiti više puta, te se time izbjegava ponavljanje koda.

⁶ IntelliSense je značajka koja omogućuje inteligentno dovršavanje ili predlaganje programskog koda, te je dostupno u većini današnjih razvojnih okruženja.

Studio razvojnom okruženju, te možemo vidjeti sva svojstva nekog modela, te je samim tim vjerovatnost pogreške znatno manja.

```
@model IEnumerable<VaqWeb.Models.Classroom>
<div class="card">
  <div class="card-body">
    <h6 class="card-title">Dvorane</h6>
    <div class="table-responsive">
      <table class="table">
        <thead>
          <tr>
            <th>@Html.DisplayNameFor(model => model.Name)</th>
          </tr>
        </thead>
        <tbody>
          <foreach (var item in Model) {
            <tr>
              <td>
                @Html.DisplayFor(modelItem => item.Name)
              </td>
              <td class="text-right">
                <a class="mr-5" asp-action="Edit" asp-route-id="@item.Id"><i data-feather="edit"></i>Uredi</a>
                <a class="mr-5" asp-action="Details" asp-route-id="@item.Id"><i data-feather="info"></i>Detalji</a>
                <a class="mr-5" asp-action="Delete" asp-route-id="@item.Id"><i data-feather="delete"></i>Ukloni</a>
              </td>
            </tr>
          </foreach>
        </tbody>
      </table>
    </div>
  </div>
</div>
```

Slika 2.7: Prikaz partial pogleda u kojem prosljeđujemo podatke pomoću specifikiranja modela

```
<canvas id="airQualityChart" width="1200" height="200"></canvas>
<script type="text/javascript">
  var data = JSON.parse('@ViewBag.AirQualityData'.replace(/&quot;/g, ''));
  var ctx = document.getElementById('airQualityChart');

  var myChart = new Chart(ctx, {
    type: 'bar',
    data: {
      datasets: [{
        data: data,
        label: "Air quality [%]",
        borderColor: 'rgb(75, 192, 192)',
        borderWidth: 1
      }]
    },
    options: {
      parsing: {
        yAxisKey: 'AirQuality',
        xAxisKey: 'CreatedAt'
      },
      responsive: false,
      scales: {
        y: {
          beginAtZero: true
        }
      }
    }
  });
</script>
```

Slika 2.8: Primjer prosljeđivanja podataka na pogled pomoću ViewBag objekta

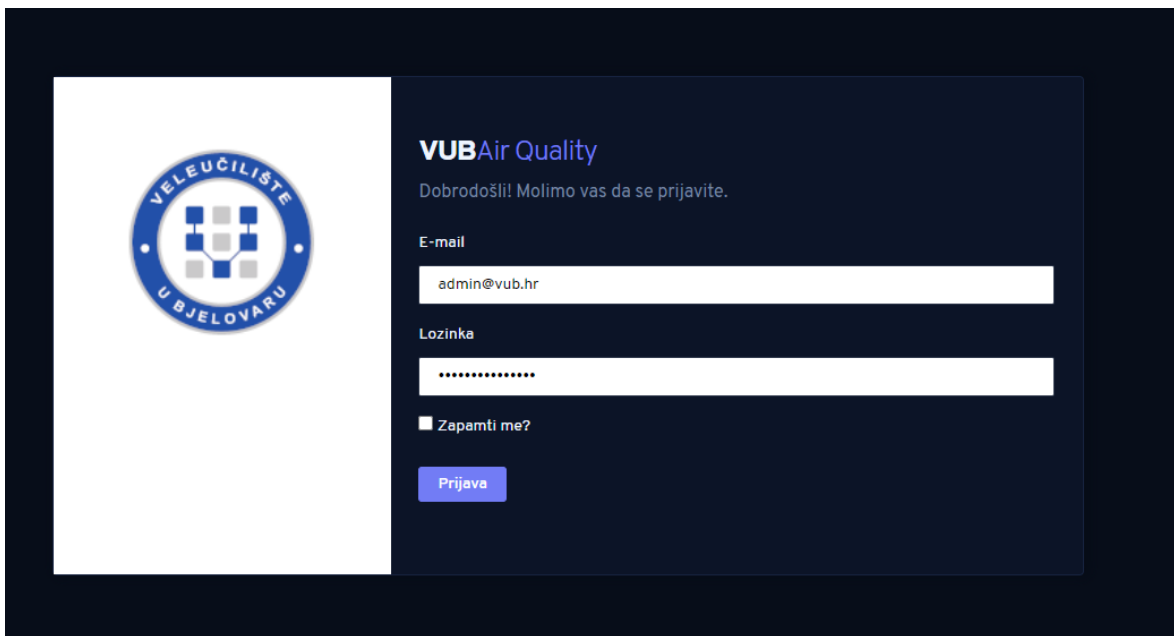
2.2 Autentikacija

2.2.1 ASP.NET Core Identity

Vrlo bitan aspekt svake web aplikacije je sigurnost. Da bi omogućili osnovne sigurnosne zahtjeve, potrebno je dodati sustav autentikacije na postojeću web aplikaciju. Jedno od mogućih rješenja je korištenje ASP.NET Core Identity platforma. Identity platforma pruža API koji podržava autentikaciju putem korisničkog sučelja. Omogućeno je upravljanje korisničkim podacima, dodavanje potvrde registracije putem email-a, korisničke uloge i još mnogo toga.

Korisnicima se može pružiti stvaranje korisničkih računa putem postojećih računa na socijalnim mrežama poput Facebook-a, Google-a, Twitter-a i Microsoft-a.

Identity je platforma temeljena na otvorenom kodu, što znači da je programski kod dostupan svima na GitHub ⁷platformi.



Slika 2.9: Prikaz forme za prijavu na web aplikaciju

Sama prijava na aplikaciju u ovom slučaju zahtjeva samo osnovne korisničke podatke poput email adrese i lozinke.

⁷ GitHub je web platforma koja omogućava verzioniranje programskog koda pomoću Git rješenja za verzioniranje.

```

@page
@model LoginModel

@{
    ViewData["Title"] = "Log in";
    Layout = "~/Views/Shared/Main/_Auth.cshtml";
}

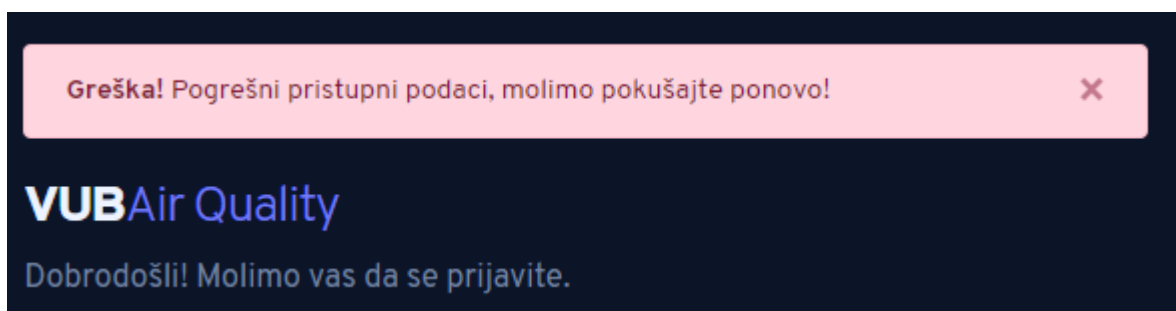
<div class="auth-form-wrapper px-4 py-5">
    <div class="alert alert-danger alert-dismissible fade show" role="alert">
        <strong>Error </strong> @ModelState.Root.Errors[0].ErrorMessage
        <button type="button" class="close" data-dismiss="alert" aria-label="Close">
            <span aria-hidden="true">&times;</span>
        </button>
    </div>
    <a asp-area="Identity" asp-page="/Login" class="noble-ui-logo logo-light d-block mb-2">VUB<span>Air Quality</span></a>
    <h5 class="text-muted font-weight-normal mb-4">Dobrodošli! Molimo vas da se prijavite.</h5>
    <form id="account" method="post">
        <div class="form-group">
            <label asp-for="Input.Email">E-mail</label>
            <input asp-for="Input.Email" class="form-control" />
            <span asp-validation-for="Input.Email" class="text-danger"></span>
        </div>
        <div class="form-group">
            <label asp-for="Input.Password">Lozinka</label>
            <input asp-for="Input.Password" class="form-control" />
            <span asp-validation-for="Input.Password" class="text-danger"></span>
        </div>
        <div class="form-group">
            <div class="checkbox">
                <label asp-for="Input.RememberMe">
                    <input asp-for="Input.RememberMe" />
                    @Html.DisplayNameFor(m => m.Input.RememberMe)
                </label>
            </div>
        </div>
        <div class="form-group">
            <button type="submit" class="btn btn-primary">Prijava</button>
        </div>
    </form>
</div>

@section Scripts {
    <partial name="_ValidationScriptsPartial" />
}

```

Slika 2.10: Prikaz programskog koda forme za prijavu na web aplikaciju

Slika 2.10 prikazuje programski kod pogleda za prijavu na web aplikaciju. Pogled se sastoji od forme koja je pisana kao Razor page. Za logiku prilikom prijave, odjave i registracije, zadužen je LoginModel. Korisnik je obaviješten u slučaju pokušaja prijave sa netočnim podacima vizualno, putem obavijesti.



Slika 2.11: Prikaz obavijesti u slučaju neuspjele prijave na web aplikaciju

```

0 references
public async Task<IActionResult> OnPostAsync(string returnUrl = null) {
    returnUrl ??= Url.Content("~/");

    ExternalLogins = (await _signInManager.GetExternalAuthenticationSchemesAsync()).ToList();

    if (ModelState.IsValid) {
        // This doesn't count login failures towards account lockout
        // To enable password failures to trigger account lockout, set lockoutOnFailure: true
        var result = await _signInManager.PasswordSignInAsync(Input.Email, Input.Password, Input.RememberMe, lockoutOnFailure: false);
        if (result.Succeeded) {
            _logger.LogInformation("User logged in.");
            return LocalRedirect(returnUrl);
        }
        if (result.RequiresTwoFactor) {
            return RedirectToPage("./LoginWith2fa", new { ReturnUrl = returnUrl, RememberMe = Input.RememberMe });
        }
        if (result.IsLockedOut) {
            _logger.LogWarning("User account locked out.");
            return RedirectToPage("./Lockout");
        }
        else {
            ModelState.AddModelError(string.Empty, "Invalid login attempt.");
            return Page();
        }
    }

    // If we got this far, something failed, redisplay form
    return Page();
}

```

Slika 2.12: Prikaz metode u LoginModel klasi koja se brine o prijavi na web aplikaciju

Prikaz programskog koda koji omogućuje prijavu vidljiv je na slici 2.12. Nakon što korisnik pritisne gumb za prijavu, kontaktira se baza podataka i ukoliko postoji korisnik sa unesenim pristupnim podacima, korisnik nastavlja normalno koristiti aplikaciju, te se radnja logira. U slučaju neuspješne autentikacije, prikazuje se poruka o grešci koja je prikazana na slici 2.11 i korisnik ostaje na formi za prijavu u web aplikaciju.

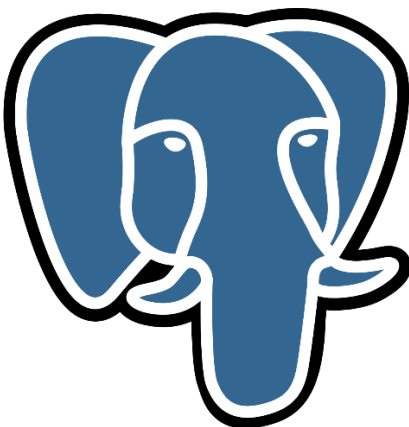
2.3 Baza podataka

Baza podataka je jedna od najbitnijih komponenti pri projektiranju web aplikacija. Glavna zadaća baze podataka u ovom slučaju je skladištenje podataka. Skladištenje i prikaz podataka u web aplikaciji odvijaju se pomoću DbContext klase, koja kao svojstva ima sve modele koji se koriste za prikaz podataka.

U ovom slučaju, poslovna logika se neće nalaziti u bazi podataka, što je moguće, recimo, kod Oracle baze podataka, koja nudi pisanje poslovne logike u obliku PL/SQL paketa.

Bitno je napomenuti da postoje relacijske i nerelacijske baze podataka. Relacijske baze podataka imaju strukturirane podatke u obliku tablica. Jedno polje u jednoj tablici, može, primjerice, imati vezu na neko drugo polje u drugoj tablici, te se na taj način može ostvariti veza. Kod realcijskih baza podataka, koristi se upitni SQL jezik. Nerelacijske baze podataka nemaju predefinirane tablice, te su pogodne za dinamičku pohranu nestrukturiranih podataka. Umjesto tablica imamo kolekcije podataka. Jedan zapis u nestrukturiranoj bazi podataka naziva se dokument.

Jedan od bitnih zahtjeva prilikom projektiranja ove web aplikacije predvidio je da se koristi relacijska baza podataka. Na kraju je donesena odluka da se koristi PostgreSQL baza podataka. PostgreSQL je baza podataka otvorenog koda, pisana u jeziku C. Trenutno broji tri desetljeća aktivnog razvoja zbog kojeg drži reputaciju za visoke performanse i pouzdanost.



Slika 2.13: Logo PostgreSQL baze podataka

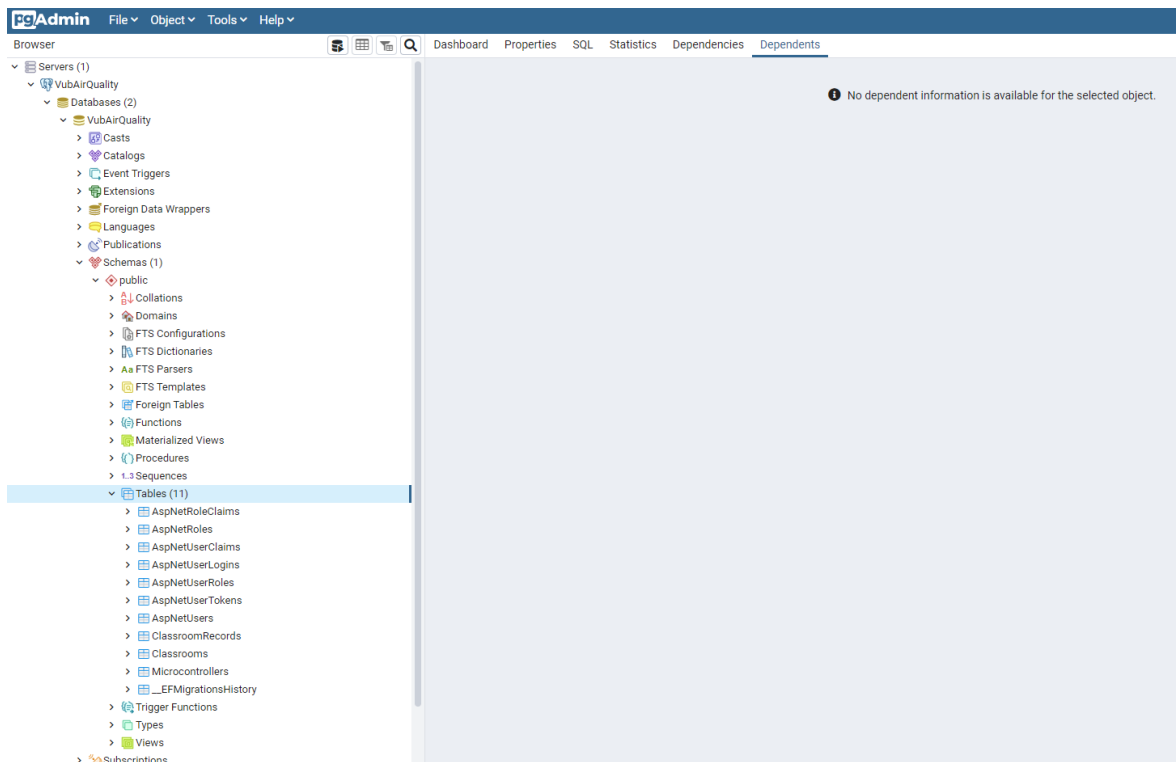
2.3.1 PostgreSQL

Uz PostgreSQL bazu podataka, potrebno je instalirati dodatan programski paket koji pomoću korisničkog sučelja omogućava interakciju sa bazom podataka. Naziv programskog paketa koji se koristi u ovoj web aplikaciji je pgAdmin.



Slika 2.14: Prijava na bazu podataka pomoću programskog paketa pgAdmin

MySQL baza podataka najčešće dolazi u kombinaciji sa phpMyAdmin programskim paketom koji pruža korisničko sučelje te olakšava interakciju sa bazom podataka, dok je to kod MongoDB baze podataka MongoExpress.



Slika 2.15: Prikaz korisničkog sučelja nakon uspješne prijave na bazu podataka

Korisničko sučelje pgAdmin omogućuje stvaranje procedura, funkcija, sekvenci, tablica, te različitih operacija nad tablicama. Primjerice, putem korisničkog sučelja, korisnik može izbrisati podatke iz tablice, izbrisati tablicu, koristiti funkcije izvoza i uvoza podataka, stvarati skripte nad tablicama i koristiti pretragu podataka.

The screenshot shows the pgAdmin interface with a SQL query editor and a data output table. The query is as follows:

```

1 SELECT "Id", "UserName", "NormalizedUserName", "Email", "NormalizedEmail", "EmailConfirmed", "PasswordHash", "SecurityStamp",
2 "ConcurrencyStamp", "PhoneNumber", "PhoneNumberConfirmed", "TwoFactorEnabled", "LockoutEnd", "LockoutEnabled", "AccessFailedCount"
3 FROM public."AspNetUsers";

```

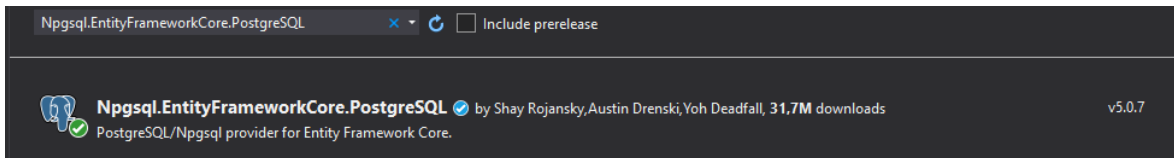
The data output table contains the following information:

Id	UserName	NormalizedUserName	Email	NormalizedEmail	EmailConfirmed	PasswordHash
1	admin@vub.hr	ADMIN@VUB.HR	admin@vub.hr	ADMIN@VUB.HR	true	AQAAAAEAACcQAAAAELPaXvtzAx9n7ysAEhKVvW4XtliqVCu+Kp

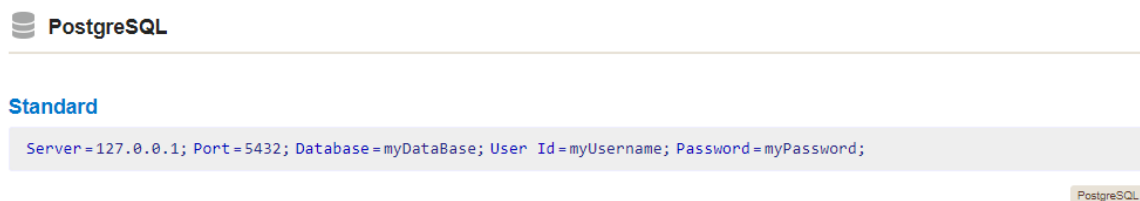
Slika 2.16: Izvršavanje SQL upita unutar pgAdmin programskog alata.

2.3.2 *Komunikacija web aplikacije sa bazom podataka*

Komunikacija web aplikacije sa bazom podataka omogućuje se u više koraka. Jedan od bitnih preuvjeta ostvarivanja veze sa bazom podataka je instalacija odgovarajućih NuGet paketa ⁸koji omogućuju povezivanje sa PostgreSQL bazom podataka.



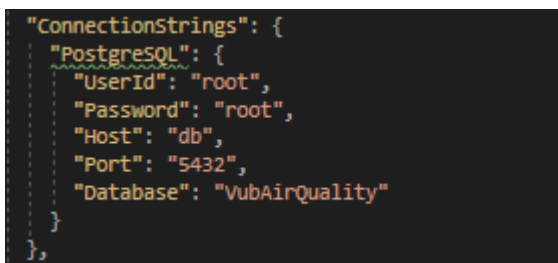
Slika 2.17: NuGet paket potreban za komunikaciju sa bazom podataka



Slika 2.18: Standardni konekcijski string za PostgreSQL bazu podataka

Nakon uspješne instalacije NuGet paketa prikazanog na slici 2.17, potrebno je promotriti od čega se sastoji konekcijski string za bazu podataka. Standardni konekcijski string za PostgreSQL bazu podataka vidljiv je na slici 2.18

Sljedeći korak je raspisati konekcijski string u appsettings.json⁹ datoteku kao na slici 2.19 u glavnoj mapi web aplikacije.



Slika 2.19: Izgled konekcijskog stringa u datoteci appsettings.json

⁸ NuGet paketi predstavljaju biblioteke koje se mogu preuzeti pomoću Visual Studio razvojnog okruženja ili pomoću komandnog retka.

⁹ appsettings.json je konfiguracijska datoteka koja primarno služi za pohranu konekcijskih stringova za servise poput baze podataka

Po završetku prethodnog koraka, potrebno je napisati klasu koja bi predstavljala konekcijski string. U ovom slučaju, klasa bi morala imati pet svojstava. Ideja je da se u toj klasi nadjača metoda ToString, te da ona vraća konekcijski string u odgovarajućem obliku. Navedena klasa je prikazana na slici 2.20.

```
public class DatabaseSettings {
    2 references
    public string UserId { get; set; }

    2 references
    public string Password { get; set; }

    2 references
    public string Host { get; set; }

    2 references
    public string Port { get; set; }

    2 references
    public string Database { get; set; }

    1 reference
    public DatabaseSettings() {
    }

    - references
    public override string ToString() {
        return $"User ID={UserId}; Password={Password}; Host={Host}; Port={Port}; Database={Database}";
    }
}
```

Slika 2.20: Klasa koja predstavlja konekcijski string za PostgreSQL bazu podataka

Naredni korak uključuje dodavanje DatabaseSettings klase kao svojstva u Startup¹⁰ klasu, instanciranje klase u konstruktoru Startup klase, što je vidljivo na slici 2.21., te zatim poziv metode koja će pokupiti vrijednosti iz appsettings.json konfiguracijske datoteke i postaviti svojstva DatabaseSettings klase, kako bi kasnije mogli pozvati metodu ToString koja bi morala vratiti ispravan konekcijski string. Sve ovo je bilo potrebno kako bi se izuzetno lako negdje u budućnosti mogao izmijeniti konekcijski string, bez potrebe za diranjem programskog koda, već samo izmjenom konfiguracijske datoteke appsetting.json.

```
0 references
public Startup(IConfiguration configuration) {
    Configuration = configuration;
    _databaseSettings = new DatabaseSettings();
    DatabaseConfig();
}

18 references
public IConfiguration Configuration { get; }

8 references
private DatabaseSettings _databaseSettings { get; set; }
```

Slika 2.21: Instanciranje klase DatabaseSettings u konstruktoru Startup klase

¹⁰ Startup klasa služi za konfiguraciju raznih servisa i zahtjeva na web aplikaciji.

```

1 reference
private void DatabaseConfig() {
    // Database config
    _databaseSettings.UserId = Configuration.GetSection("ConnectionStrings:PostgreSQL:UserId").Value;
    _databaseSettings.Password = Configuration.GetSection("ConnectionStrings:PostgreSQL:Password").Value;
    _databaseSettings.Host = Configuration.GetSection("ConnectionStrings:PostgreSQL:Host").Value;
    _databaseSettings.Port = Configuration.GetSection("ConnectionStrings:PostgreSQL:Port").Value;
    _databaseSettings.Database = Configuration.GetSection("ConnectionStrings:PostgreSQL:Database").Value;
}

```

Slika 2.22: Metoda u Startup klasi koja čita konekcijski string iz konfiguracijske datoteke

Nakon instanciranja objekta `_databaseSettings` u konstruktoru Startup klase, poziva se metoda `DatabaseConfig` koja je vidljiva na slici 2.22 čiji je cilj pročitati sve podatke vezane uz konekcijski string i postaviti ih u `_databaseSettings` objekt.

Posljednji korak koji je potreban prije konfiguracije konekcije u Startup klasi je dodavanje `DbContext` klase. `DbContext` klasa služi za direktnu interakciju sa bazom podataka. Omogućuje razne operacije, od jednostavnijih upita, stvaranje novih zapisa, ažuriranje ili brisanje zapisa. U slučaju jednostavnijih upita, ne moramo pisati klasičan SQL, već se koristi tehnologija upita koja se naziva LINQ, čiji primjer korištenja je vidljiv na slici 2.23.

```

2 references
public IEnumerable<object> GetTemperatureBasedChartData(int id) {
    var classroomRecords = GetClassroomRecordsById(id);

    var data = from c in classroomRecords
               orderby c.CreatedAt ascending
               select new { Temperature = c.Temperature, CreatedAt = c.CreatedAt.ToString("dddd, dd MMM yyyy HH:MM") };

    return data;
}

```

Slika 2.23: Primjer korištenja LINQ upita

```

2 references
public class VagWebDbContext : IdentityDbContext {
    0 references
    public VagWebDbContext(DbContextOptions<VagWebDbContext> options) : base(options) {
        ...
    }

    10 references
    public DbSet<Classroom> Classrooms { get; set; }

    7 references
    public DbSet<Microcontroller> Microcontrollers { get; set; }

    2 references
    public DbSet<ClassroomRecord> ClassroomRecords { get; set; }
}

```

Slika 2.24: VaqWebDbContext klasa

U ovom slučaju, VaqWebDbContext ima tri svojstva kojima može pristupiti radi interakcije sa bazom podataka. Ta tri svojstva su zapravo modeli koji služe za oblikovanje podataka pri prikazu na pogledu.

Posljednji korak je vidljiv na slici 2.25, gdje je potrebno dodati VaqWebDbContext klasu kao servis u Startup klasi u metodi ConfigureServices.

```
0 references
public void ConfigureServices(IServiceCollection services) {
    services.AddControllerswithViews();

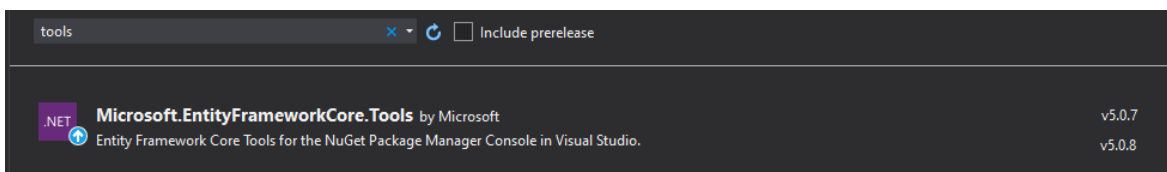
    services.AddDbContext<VaqWebDbContext>(opt => opt.UseNpgsql(_databaseSettings.ToString()));
}
```

Slika 2.25: Dodavanje DbContext klase kao servisa u ConfigureServices metodi u Startup klasi

2.3.3 Migracije

Stvaranje tablica na bazi podataka u ovom slučaju je moguće na dva načina. Jedan način je pisanje SQL skripti te potom izvršavanje na bazi podataka, a drugi način je iskorišavanje migracija u ASP.NET Core programskom okviru. Uz postojeće modele, konekciju na bazu podataka i DbContext klasu, lakše je iskoristiti značajku migracije koju nudi programski okvir ASP.NET Core.

Migracije se formiraju po uzoru na modele. Ideja je da svaki model predstavlja jednu tablicu u bazi.



Slika 2.26: NuGet paket koji omogućava generiranje migracije uz pomoć modela

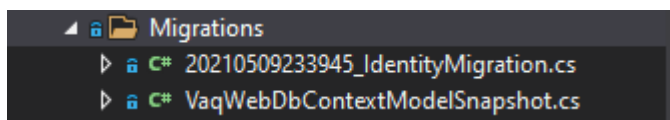
Da bi mogli pokrenuti proces koji će izgenerirati migracije, potreban je jedan NuGet paket koji je prikazan na slici 2.26. Nakon uspješne instalacije NuGet paketa, potrebno je pokrenuti Package Manager Console ¹¹u Visual Studio razvojnom okruženju i pokrenuti naredbu vidljivu na slici 2.27.

```
dotnet ef migrations add InitialMigration
```

Slika 2.27 Naredba za dodavanje migracija

¹¹ Package Manager Console je zapravo PowerShell konzola koja primarno služi za interakciju sa NuGet paketima.

Nakon uspješnog izvršenja naredbe na slici 2.27, u glavnom direktoriju projekta, stvoriti će se direktorij Migrations, prikazan na slici 2.28.



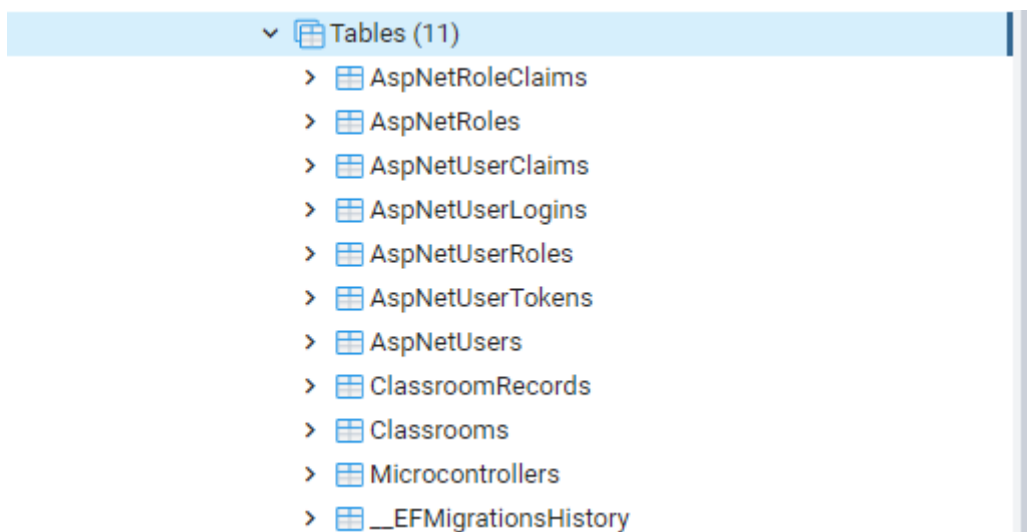
Slika 2.28 Novonastali direktorij Migrations

Kako bi izvršili migracije i stvorili tablice na bazi podataka, potrebno je izvršiti još jednu naredbu u Package Manager konzoli prikazanoj na slici 2.29.

```
dotnet ef database update
```

Slika 2.29 Naredba za izvršavanje migracija

Nakon uspješne migracije, poželjno je provjeriti jesu li sve tablice stvorene. To je moguće koristeći pgAdmin administracijski paket za PostgreSQL bazu podataka. Nakon što se spojimo na bazu podataka, odaberemo VubAirQuality bazu podataka, public shemu, te otvorimo padajući izbornik Tables koji je prikazan na slici 2.30.

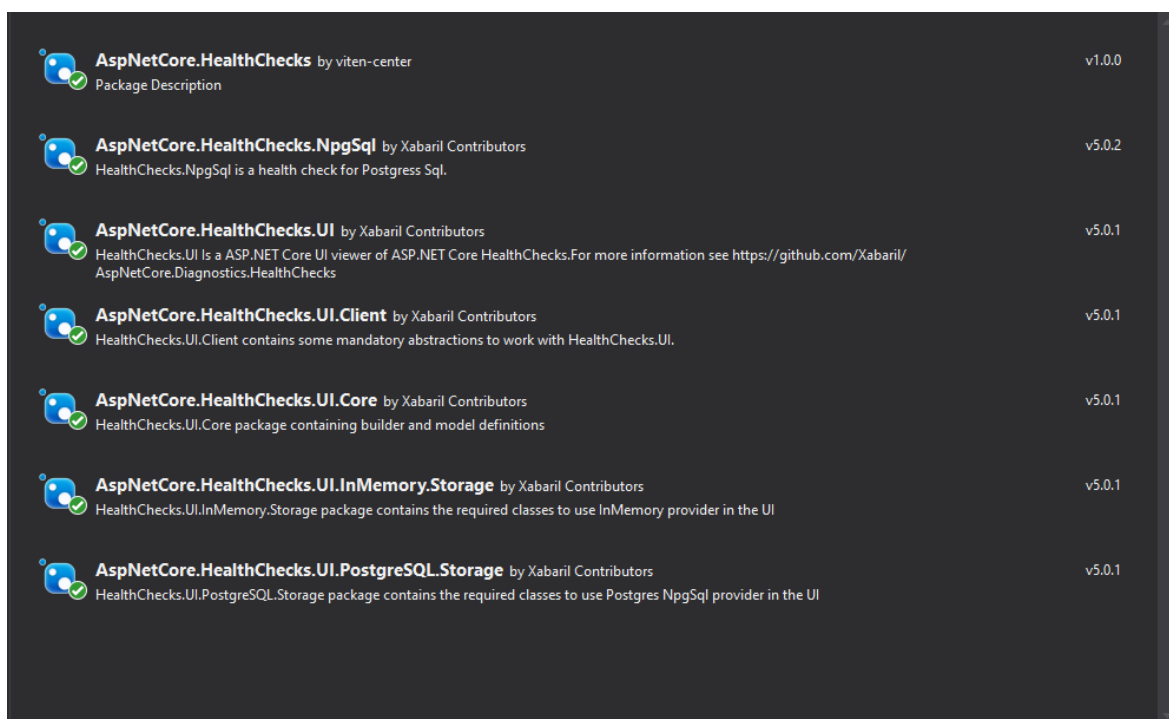


Slika 2.30 Prikaz tablica u administracijskom paketu pgAdmin nakon uspješne migracije

2.3.4 *Provjera zdravlja baze podataka*

Baza podataka je vitalni dio svake web aplikacije. Ukoliko dođe do pada baze podataka, web aplikacija prestaje raditi. U ovom slučaju, bilo bi nemoguće proći prvi korak, to jest autentikaciju. Čak i da nema sustava autentikacije, podaci o uvjetima unutar dvorana bili bi neuhvatljivi. Vrlo je zgodno imati programski alat koji bi na jednostavan način mogao poslati jednostavan upit na bazu, te na nekom korisničkom sučelju prikazati stanje baze podataka. Takav programski alat bi izvršavao funkciju provjere zdravlja, to jest, ispravnosti baze podataka (engl. *healthcheck*).

ASP.NET Core programski okvir nudi podršku za provjeru ispravnosti baze podataka. Sam upit na bazu traje vrlo kratko, potrebno je spojiti se na bazu podataka, izvršiti upit SELECT 1 te zatim vratiti rezultat.



Slika 2.31 Popis potrebnih NuGet paketa da bi se implementirala provjera ispravnosti baze podataka

Popis potrebnih NuGet paketa za implementaciju provjere zdravlja baze podataka vidljiv je na slici 2.31. Nakon instalacije navedenih NuGet paketa, potrebno je popuniti konfiguracijsku datoteku appsettings.json prema slici 2.32.

```
{
  "Healthcheck": {
    "Url": "/Healthcheck",
    "HealthcheckApiUrl": "/health-api",
    "CustomStylesheetPath": "./wwwroot/css/healthcheck_ui.css",
    "EvaluationTimeInSeconds": 15,
    "MaximumHistoryEntriesPerEndpoint": 60,
    "SetApiMaxActiveRequests": 1,
    "VagHealthcheck": {
      "Name": "VAQ Database",
      "HealthcheckSelectQuery": "SELECT 1;",
      "Tags": {
        "0": "PostgreSQL",
        "1": "VUB"
      }
    }
  }
},
```

Slika 2.32 Prikaz dodanih vrijednosti u appsettings.json datoteku

U konfiguracijsku datoteku su dodani detalji poput rute na kojoj će biti prikazano korisničko sučelje, SQL upita kojeg šaljem na bazu, oznaka za bazu podataka, naziva baze podataka, vrijeme koje mora proći između dvije provjere te prilagođena CSS¹² datoteka prema kojoj će se uskladiti stil korisničkog sučelja. Na slici 2.33 prikazan je sadržaj prilagođene CSS datoteke.

```
1 :root {
2   --primaryColor: #2250A9;
3   --secondaryColor: #F4F4F4;
4   --bgMenuActive: #002979;
5   --bgButton: #1F4CA5;
6   --logoImageUrl: url('https://vub.hr/images/uploads/vub-logo.png');
7   --bgAside: var(--primaryColor);
8   --successColor: #3AAA97;
9 }
10
```

Slika 2.33 Prikazuje sadržaj prilagođene CSS datoteke koja služi za izmjenu dogovoreno

¹² Cascaded Style Sheets ili CSS – markdown jezik koji služi za opisivanje i formatiranje HTML dokumenta.

Nakon uspješne konfiguracije, potrebno je još dodati programski kod za konfiguraciju provjere zdravlja u Startup klasu. Prvi dio koda koji je prikazan na slici 2.34 potrebno je dodati u metodu `ConfigureServices`. Navedeni dio programskog koda čita vrijednosti koje smo prethodno postavili u konfiguracijsku datoteku `appsettings.json`. Preostali dio programskog koda koji se nalazi na slici 2.35 potrebno je dodati u metodu `Configure` koja se nalazi u Startup klasi.

```
services.AddHealthChecks().AddNpgsql(
    npgsqlConnectionString: _databaseSettings.ToString(),
    healthQuery: Configuration.GetSection("Healthcheck:VagHealthcheck:HealthcheckSelectQuery").Value,
    name: Configuration.GetSection("Healthcheck:VagHealthcheck:Name").Value,
    tags: new string[] {
        Configuration.GetSection("Healthcheck:VagHealthcheck:Tags:0").Value,
        Configuration.GetSection("Healthcheck:VagHealthcheck:Tags:1").Value
    }
);
services.AddHealthChecksUI(opt => {
    opt.SetEvaluationTimeInSeconds(Int32.Parse(Configuration.GetSection("Healthcheck:EvaluationTimeInSeconds").Value)); // Time in seconds between checks.
    opt.MaximumHistoryEntriesPerEndpoint(Int32.Parse(Configuration.GetSection("Healthcheck:MaximumHistoryEntriesPerEndpoint").Value)); // Maximum history of checks.
    opt.SetApiMaxActiveRequests(Int32.Parse(Configuration.GetSection("Healthcheck:SetApiMaxActiveRequests").Value)); // API requests concurrency.
    opt.AddHealthCheckEndpoint( // Map healthcheck API.
        Configuration.GetSection("Healthcheck:VagHealthcheck:Name").Value,
        $"http://{Dns.GetHostName():80}{Configuration.GetSection("Healthcheck:HealthcheckApiUrl").Value}"
    );
});
.AddMemoryStorage();
```

Slika 2.34 Prikazuje programski kod za konfiguraciju provjere zdravlja baze podataka koji je potrebno dodati u metodu `ConfigureServices` koja se nalazi u Startup klasi.

```
app.UseHealthChecks("/health", new HealthCheckOptions() {
    Predicate = _ => true,
    ResponseWriter = UIResponseWriter.WriteHealthCheckUIResponse
});
app.UseHealthChecksUI(options => {
    options.UIPath = Configuration.GetSection("Healthcheck:Url").Value;
    options.AddCustomStyleSheet(Configuration.GetSection("Healthcheck:CustomStyleSheetPath").Value);
});
app.UseEndpoints(endpoints => {
    endpoints.MapControllerRoute(
        name: "default",
        pattern: "{controller=Home}/{action=Index}/{id?}");

    endpoints.MapHealthChecks(Configuration.GetSection("Healthcheck:HealthcheckApiUrl").Value, new HealthCheckOptions {
        Predicate = _ => true,
        ResponseWriter = UIResponseWriter.WriteHealthCheckUIResponse
    });

    endpoints.MapHealthChecksUI();
    endpoints.MapRazorPages();
});
```

Slika 2.35 Prikazuje programski kod za konfiguraciju provjere zdravlja baze podataka koji je potrebno dodati u metodu `Configure` koja se nalazi u Startup klasi.

Nakon uspješne konfiguracije, korisničko sučelje provjere zdravlja baze podataka dostupno je na ruti `/Healthcheck`. Prikaz korisničkog sučelja vidljiv je na slici 2.36

NAME	HEALTH	ON STATE FROM	LAST EXECUTION
VAQ Database	Healthy	2021-07-15T09:30:08.5114344+00:00	7/15/2021, 11:30:08 AM

Additional table details:

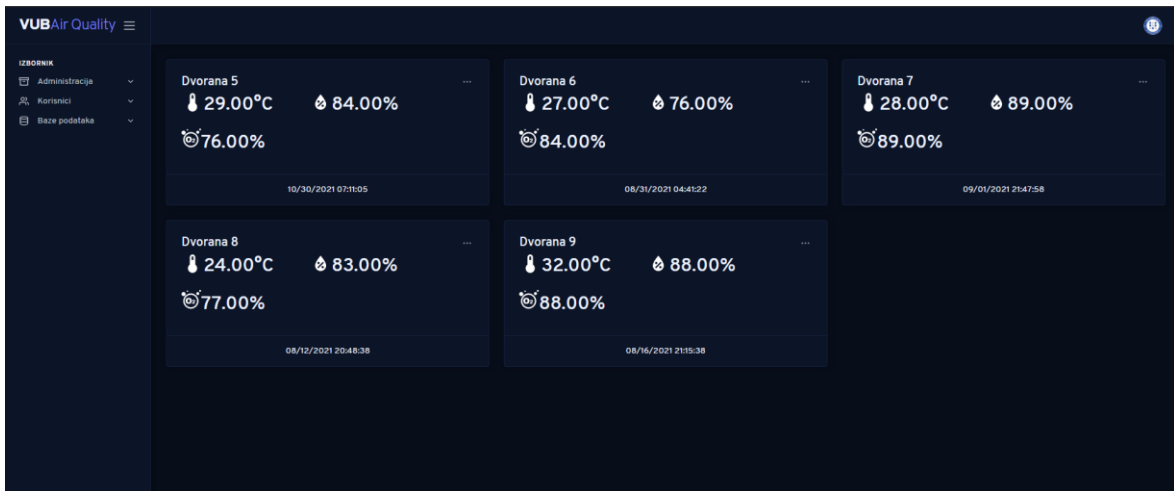
NAME	TAGS	HEALTH	DESCRIPTION	DURATION	DETAILS
VAQ Database	VAQ	Healthy		00:00:01.4186364	

Slika 2.36 Korisničko sučelje programskog paketa za provjeru ispravnost baze podataka.

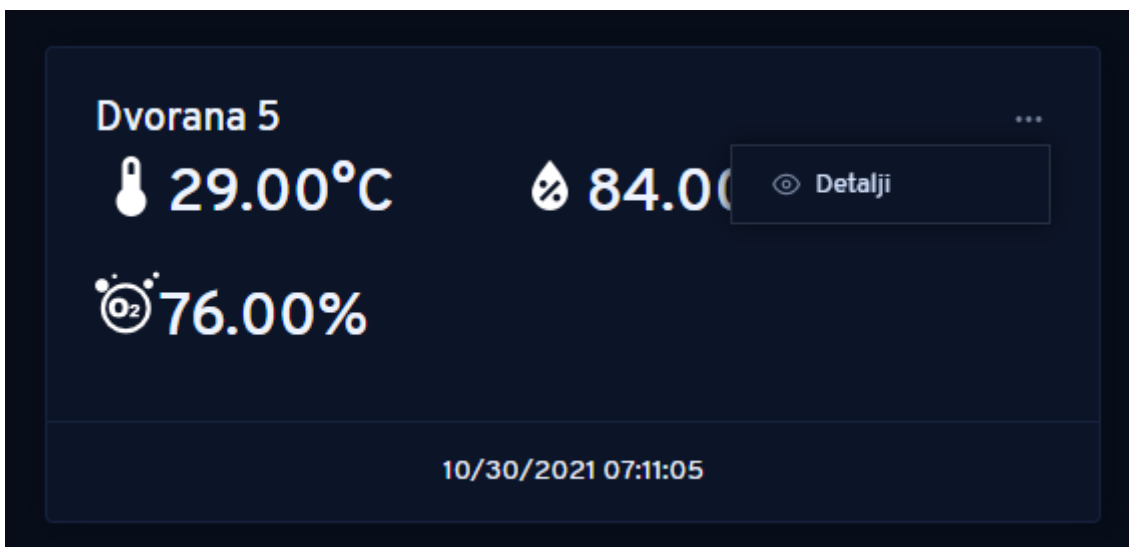
2.4 Vizualizacija podataka

Vizualizacija podataka je proces u kojem uzimamo skup podataka nad kojim radimo analizu, te ga prikažemo pomoću neke od postojećih biblioteka za vizualizaciju. Jedan od ključnih koraka je vraćanje odgovarajućih podataka u pogled.

Na glavnoj stranici koja je vidljiva na slici 2.37 prikazuju se sve dvorane koje imaju barem jedan zapis koji uključuje vrijednosti temperature, kvalitete zraka, vlažnosti i prisutnosti osoba u dvorani.



Slika 2.37 Prikaz najnovijih podataka na glavnoj stranici web aplikacije

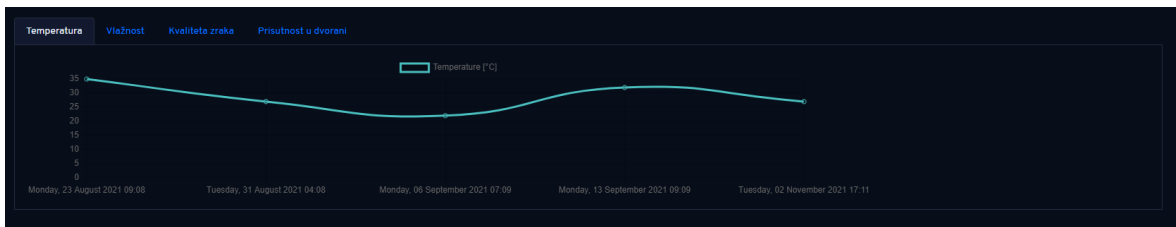


Slika 2.38 Prikaz skočnog izbornika koji omogućuje vizualizaciju podataka klikom na gumb „Detalji“

2.4.1 Prikaz podataka na aplikaciji

Korisnik može vidjeti detaljnu vizualizaciju podataka prilikom odabira jedne od dvorana. Odabir dvorana vrši se pomoću malog skočnog izbornika i pritiska na gumb „Detalji“, onako kako je prikazano na slici 2.38.

Prva opcija je pregled zapisa o temperaturi odabrane dvorane i nalazi se na slici 2.39.



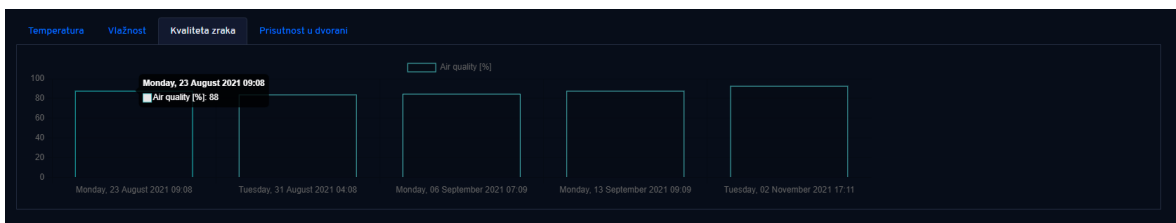
Slika 2.39 Vizualizacija temperature odabrane dvorane. Os ordinata prikazuje temperaturu u Celzijevim stupnjevima a na osi apscisa se nalaze vremenske oznake koje prikazuju vrijeme stvaranja zapisa

Druga opcija prikazuje zapise o vlažnosti u postocima, te se može vidjeti na slici 2.40.



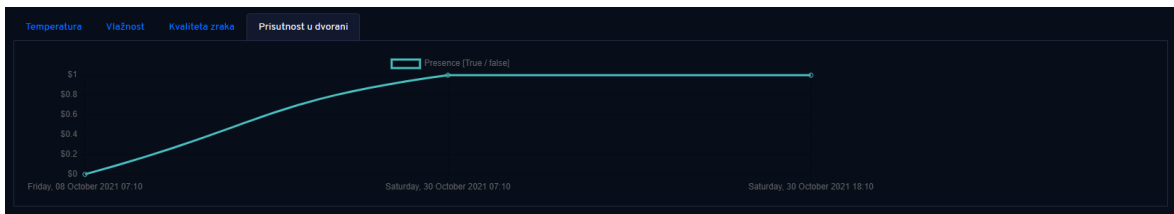
Slika 2.40 Vizualizacija vlažnosti odabrane dvorane. Os ordinata prikazuje postotak vlažnosti a na osi apscisa se nalaze vremenske oznake koje prikazuju vrijeme stvaranja zapisa.

Treća opcija služi za prikaz kvalitete zraka u odabranoj dvorani i to u postocima. Vizualizacija je prikazana na slici 2.41.



Slika 2.41 Vizualizacija kvalitete zraka odabrane dvorane. Os ordinata prikazuje postotak kvalitete zraka a na osi apscisa se nalaze vremenske oznake koje prikazuju vrijeme stvaranja zapisa

Zadnja vizualizacija predstavlja prikaz prisutnosti u odabranoj dvorani. Prikaz je vidljiv na slici 2.42.



Slika 2.42 Vizualizacija kvalitete zraka odabrane dvorane. Os ordinata prikazuje vrijednost između 0 i 1, to jest prisutnosti ili neprisutnosti u dvorani. Na osi apscisa su zabilježena vremena stvaranja zapisa.

Kontroler zadužen za glavnu stranicu web aplikacije je HomeController. U njemu se nalazi logika za dohvaćanje podataka o temperaturi, vlažnosti, kvaliteti zraka i prisutnosti u dvorani. Za vraćanje podataka na pogled (glavnu stranicu), zadužena je metoda Charts, prikazana na slici 2.43. Metoda prima jedan argument, id dvorane. Podaci se postavljaju u ViewBag u obliku JSON ¹³strukture.

```
0 references
public async Task<IActionResult> Charts(int? id) {
    if (id == null) {
        return NotFound();
    }

    ViewBag.TemperatureData = JsonConvert.SerializeObject(_classroomRecordService.GetTemperatureBasedChartData((int)id));
    ViewBag.HumidityData = JsonConvert.SerializeObject(_classroomRecordService.GetHumidityBasedChartData((int)id));
    ViewBag.AirQualityData = JsonConvert.SerializeObject(_classroomRecordService.GetAirQualityBasedChartData((int)id));
    ViewBag.PresenceData = JsonConvert.SerializeObject(_classroomRecordService.GetPresenceBasedChartData((int)id));

    return View();
}
```

Slika 2.43 Metoda Charts u HomeController kontroleru koja ovisno o jedinstvenoj oznaci dvorane vraća relevantne podatke

¹³ JSON (engl. JavaScript Object Notation) je struktura primarno zadužena za prijenos podataka.

Pogled koji se brine za vizualizaciju podataka naziva se Charts i prikazan je na slici 2.44. Na pogledu Charts posoji izbornik koji nudi pregled vizualizacije temperature, vlažnosti, kvalitete zraka i detekcije prisutnosti. Svaka vizualizacija je odvojena u vlastiti partial pogled.

```
<script type="text/javascript" src="/lib/chart.js/package/dist/chart.js"></script>
<ul class="nav nav-tabs" id="mytab" role="tablist">
  <li class="nav-item">
    <a class="nav-link active" id="temperature-tab" data-toggle="tab" href="#temperature" role="tab" aria-controls="temperature" aria-selected="true">Temperatura</a>
  </li>
  <li class="nav-item">
    <a class="nav-link" id="humidity-tab" data-toggle="tab" href="#humidity" role="tab" aria-controls="humidity" aria-selected="false">Vlažnost</a>
  </li>
  <li class="nav-item">
    <a class="nav-link" id="air-quality-tab" data-toggle="tab" href="#air-quality" role="tab" aria-controls="air-quality" aria-selected="false">Kvaliteta zraka</a>
  </li>
  <li class="nav-item">
    <a class="nav-link" id="presence-tab" data-toggle="tab" href="#presence" role="tab" aria-controls="presence" aria-selected="false">Prisutnost u dvorani</a>
  </li>
</ul>
<div class="tab-content border border-top-0 p-3" id="mytabContent">
  <div class="tab-pane fade show active" id="temperature" role="tabpanel" aria-labelledby="temperature-tab">
    <partial name="/Views/Shared/Charts/_TemperatureLineChart.cshtml" />
  </div>
  <div class="tab-pane fade" id="humidity" role="tabpanel" aria-labelledby="humidity-tab">
    <partial name="/Views/Shared/Charts/_HumidityBarChart.cshtml" />
  </div>
  <div class="tab-pane fade" id="air-quality" role="tabpanel" aria-labelledby="air-quality-tab">
    <partial name="/Views/Shared/Charts/_AirQualityBarChart.cshtml" />
  </div>
  <div class="tab-pane fade" id="presence" role="tabpanel" aria-labelledby="presence-tab">
    <partial name="/Views/Shared/Charts/_PresenceLineChart.cshtml" />
  </div>
</div>
```

Slika 2.44 Prikaz pogleda zaduženog za vizualizaciju podataka

Partial pogled za prikazivanje kvalitete zraka je zapravo JavaScript skripta koja uzima JSON iz ViewBag objekta te stvara vizualizaciju na temelju podataka iz navedenog objekta i dodatnih parametara. Primjer stvaranja vizualizacije za kvalitetu zraka nalazi se na slici 2.45

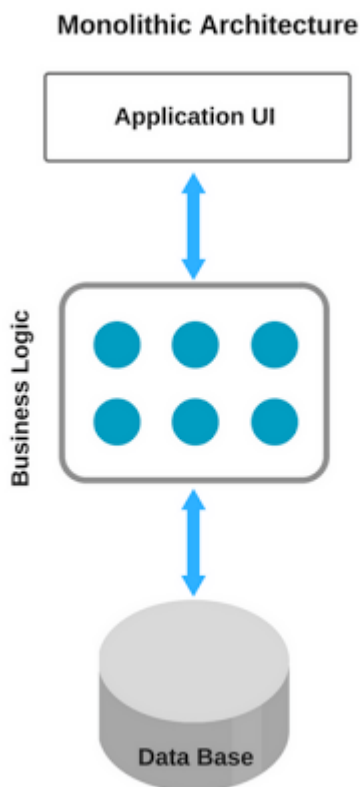
```
<canvas id="airQualityChart" width="1200" height="200"></canvas>
<script type="text/javascript">
  var data = JSON.parse('@ViewBag.AirQualityData'.replace(/&quot;/g, ''));
  var ctx = document.getElementById('airQualityChart');

  var myChart = new Chart(ctx, {
    type: 'bar',
    data: {
      datasets: [{
        data: data,
        label: "Air quality [%]",
        borderColor: 'rgb(75, 192, 192)',
        borderWidth: 1
      }]
    },
    options: {
      parsing: {
        yAxisKey: 'AirQuality',
        xAxisKey: 'CreatedAt'
      },
      responsive: false,
      scales: {
        y: {
          beginAtZero: true
        }
      }
    }
  });
</script>
```

Slika 2.45 Primjer stvaranja vizualizacije za pogled zadužen za kvalitetu zraka

3. Docker

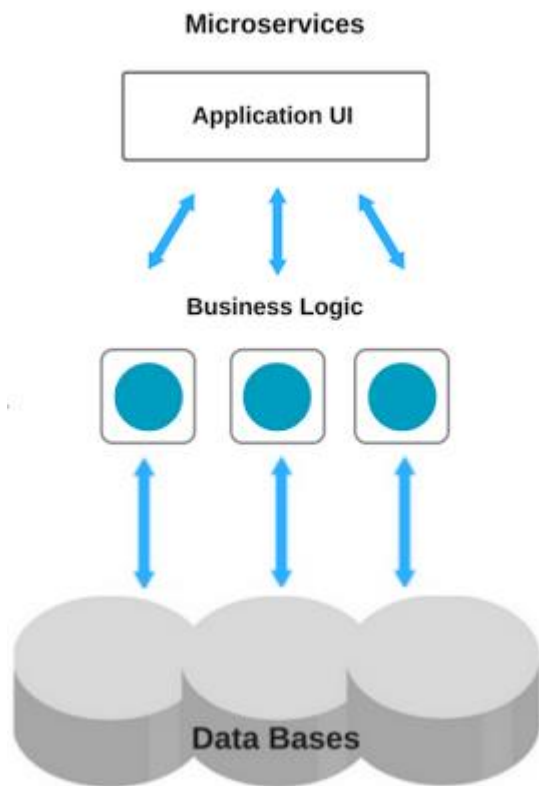
Monolitna arhitektura je klasičan oblik jedne web aplikacije. U tom slučaju, aplikacija je izgrađena kao jedna cjelina. Većinom se ta cjelina sastoji od tri sloja, klijentskog, baze podataka i serverske aplikacije. Uzmimo za primjer serversku aplikaciju. Sva poslovna logika se nalazi u jezgri serverske aplikacije. Kod takve arhitekture, nakon nekog vremena dodavanje novih značajki postane skupo. Nakon toga je vrlo teško postići idealnu strukturu projekta. Također je nemoguće skalirati samo segmente aplikacije u slučaju povećanog opterećenja, te je potrebno skalirati cijelu aplikaciju. Primjer vizualizacije monolitne arhitekture vidljiv je na slici 3.1.



Slika 3.1 Vizualizacija monolitne aplikacije

Mikroservisna arhitektura je moderniji oblik web aplikacije. U tom slučaju se web aplikacija sastoji od znatnog broja manjih servisa, što omogućava bolje upravljanje resursima. Svaki servis se u tom slučaju podiže kao posebna web aplikacija. Ovaj proces omogućuje nezavisnost i učinkovitu povezanost svih servisa. Komunikacija između servisa se najčešće izvodi pomoću alata poput Oracle Kafka ili RabbitMQ. Prednost korištenja

mikroservisa je to što više timova može raditi na jednoj aplikaciji, i to neovisno. Ne postoji centralna baza podataka, u pravilu bi svaki servis trebao imati svoju. Time se postiže decentraliziranost web aplikacije. Slika 3.2 jasno prikazuje interakciju između servisa u mikroservisnoj arhitekturi.



Slika 3.2 Vizualizacija mikroservisne arhitekture u web aplikaciji

Docker je platforma koja služi za izoliranje svakog servisa u posebnu okolinu. Potreba za instaliranjem programskih paketa na lokalni operacijski sustav više ne postoji, jer se sve odrađuje u izoliranim kontejnerima koji djeluju kao zasebni operacijski sustavi. Docker radi vrlo brzo zbog toga što se kontejneri nisu izolirani od lokalnog operacijskog sustava, već su pokrenuti na jezgri lokalnog računala što znatno štedi resurse, u usporedbi sa virtualnim mašinama koje su potpuno odvojene od ostatka lokalnog operacijskog sustava.

3.1 Kontejnerizacija

3.1.1 Docker Compose

Alat koji omogućuje definiranje i pokretanje aplikacija sa više kontejnera naziva se Docker Compose. Nakon ispunjavanja docker-compose.yaml datoteke sa željenom konfiguracijom, pomoću naredbe na slici 3.3.

```
docker-compose up
```

Slika 3.3 Prikaz naredbe koja pokreće aplikaciju sa više kontejnera

Docker Compose radi u svim okruženjima, od produkcije, razvoja, testiranja. Pisanje same konfiguracijske datoteke sastoji se od navođenja verzije Docker Compose alata, navođenja naziva servisa i dodavanja ostalih detalja koji su potrebni za podizanje servisa.

Na slici 3.4 prikazan je jednostavni oblik docker-compose.yml konfiguracijske datoteke. Rješenje sa slike se sastoji od više servisa. Prvi servis je automatski dodan pri dodavanju podrške za Docker. Sljedeći na redu je Jenkins¹⁴, nakon njega su definirana još dva servisa, db i pg-admin. Db servis se sastoji od PostgreSQL baze podataka i pgAdmin korisničkog sučelja.

¹⁴ Jenkins je besplatan automatizacijski server otvorenog koda.

```
version: '3.4'

services:
  vaqweb:
    image: ${DOCKER_REGISTRY-}vaqweb
    container_name: vaq-web
    build:
      context: .
      dockerfile: Dockerfile

  jenkins:
    image: jenkins/jenkins:lts
    container_name: jenkins
    privileged: true
    user: root
    ports:
      - 8081:8080
    volumes:
      - ~/jenkins:/var/jenkins_home
      - /var/run/docker.sock:/var/run/docker.sock
      - /usr/local/bin/docker:/usr/local/bin/docker

  db:
    image: postgres
    container_name: pg-db
    restart: always
    environment:
      POSTGRES_DB: postgres
      POSTGRES_USER: root
      POSTGRES_PASSWORD: root
      PG_DATA: /var/lib/postgresql/data
    volumes:
      - db-data:/var/lib/postgresql/data
    ports:
      - "5432:5432"

  pg-admin:
    image: dpage/pgadmin4
    container_name: pg-admin
    restart: always
    environment:
      PGADMIN_DEFAULT_EMAIL: ikranjec@vub.hr
      PGADMIN_DEFAULT_PASSWORD: root
      PGADMIN_LISTEN_PORT: 80
    ports:
      - "8080:80"
    volumes:
      - pg-admin-data:/var/lib/pgadmin
    links:
      - "db:pgsql-server"
```

Slika 3.4. Prikaz programskog koda za podizanje servisa

3.1.2 Dockerfile

Postoje slučajevi kada je potrebno dodati Dockerfile datoteku. Dockerfile datoteka služi kako bi se izvršile komplekse instrukcije prilikom izrade kontejnera. Primjerice, za vaqweb servis, potrebno je u kontejneru instalirati još dosta sitnih paketa kako bi na kraju migracije radile normalno. Detaljniji prikaz Dockerfile datoteke vidljiv je na slici 3.5 Sve naredbe koje se nalaze u Dockerfile datoteci se zapravo izvršavaju u vlastitom kontejneru. U slučaju vaqweb kontejnera, bilo je vrlo bitno dodati dotnet-ef alat koji služi prilikom migracije, pa da bi se u slučaju potrebe mogao pozvati.

```

FROM mcr.microsoft.com/dotnet/aspnet:5.0 AS base
RUN apt-get update
RUN apt-get install -y wget
RUN apt-get install -y apt-transport-https
RUN wget https://packages.microsoft.com/config/ubuntu/20.10/packages-microsoft-prod.deb -O packages-microsoft-prod.deb
RUN dpkg -i packages-microsoft-prod.deb
RUN apt-get update
RUN apt-get install -y dotnet-sdk-5.0
RUN dotnet tool install --global dotnet-ef
ENV PATH $PATH:/root/.dotnet/tools
WORKDIR /app
EXPOSE 80

FROM mcr.microsoft.com/dotnet/sdk:5.0 AS build
WORKDIR /src
COPY ["VaqWeb.csproj", "."]
RUN dotnet restore "VaqWeb.csproj"
COPY . .
WORKDIR "/src/"
RUN dotnet build "VaqWeb.csproj" -c Release -o /app/build

FROM build AS publish
RUN dotnet publish "VaqWeb.csproj" -c Release -o /app/publish

FROM base AS final
WORKDIR /app
COPY --from=publish /app/publish .
ENTRYPOINT ["dotnet", "VaqWeb.dll"]

```

Slika 3.5 Dockerfile korišten u radu

4. ZAKLJUČAK

Glavni cilj rada bio je projektirati web sustav za nadzor prisutnosti i praćenje kvalitete zraka u zatvorenim prostorijama. Takav web sustav omogućen je uz pomoć ASP.NET Core programskog okvira. Uz glavni cilj, bilo je potrebno savladati osnovna znanja pri korištenju Docker platforme. Trenutna web aplikacija napisana je kao monolit, jer se web aplikacija sastoji od samo jednog rješenja (engl. solution) u Visual Studio razvojnom okruženju. Idealna situacija bi bila kada bi se web aplikacija još dodatno razdvojila na web API za autentikaciju, web API koji bi služio pri upravljanju zapisima dvorana, web API za dvorane i web API za mikrokontrolere. Svaki od navedenih web API-a imao bi svoju bazu podataka.

Osim skupine web API-a kao servisa, morao bi se dodati i servis koji bi funkcionirao kao klijentska aplikacija. Opcije koje su podržane kroz Visual Studio su React.js i Angular.js.

Mane mikroservisnog pristupa u ovom slučaju bi bile znatno veći trošak vremena pri stvaranju predloška za aplikaciju i otežano testiranje.

Mane monolitnog sustava trenutno nisu toliko izražene jer se radi o relativno maloj web aplikaciji, ali u slučaju kada bi web aplikacija imala jednu stotinu kontrolera, servisa i modela, u tom slučaju postaje izuzetno teško pisati ispravan programski kod.

Docker kao platforma za virtualizaciju na razini operacijskog sustava izvrsno radi. Vrlo zgodna značajka je podizanje svih servisa samo sa jednom naredbom pomoću Docker Compose alata.

5. LITERATURA

- [1] Get started with ASP.NET Core MVC [Online]. 2021. Dostupno na:
<https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/start-mvc?view=aspnetcore-5.0&tabs=visual-studio> (01.06.2021.)
- [2] Views in ASP.NET Core MVC [Online]. 2019. Dostupno na:
<https://docs.microsoft.com/en-us/aspnet/core/mvc/views/overview?view=aspnetcore-5.0>
(02.06.2021.)
- [3] Partial views in ASP.NET Core [Online]. 2019. Dostupno na:
<https://docs.microsoft.com/en-us/aspnet/core/mvc/views/partial?view=aspnetcore-5.0>
(05.06.2021.)
- [4] Get Docker [Online]. 2020. Dostupno na:
<https://docs.docker.com/get-docker/> (21.06.2021.)
- [5] Overview of Docker Compose [Online]. 2020. Dostupno na:
<https://docs.docker.com/compose/> (25.06.2021.)
- [6] Health checks in ASP.NET Core [Online]. Dostupno na:
<https://docs.microsoft.com/en-us/aspnet/core/host-and-deploy/health-checks?view=aspnetcore-5.0>
- [7] Migrations Overview [Online]. Dostupno na:
<https://docs.microsoft.com/en-us/ef/core/managing-schemas/migrations/?tabs=dotnet-core-cli>

6. OZNAKE I KRATICE

API – Application Programming Interface

JSON – JavaScript Object Notation

MVC – Model-pogled-kontroler (engl. *Model-View-Controller*)

SQL – Structured Query Language

7. SAŽETAK

Naslov: Izrada web sustava za nadzor prisutnosti i praćenje kvalitete zraka u zatvorenim prostorima

Cilj rada je razvoj web aplikacije za nadzor prisutnosti i praćenje kvalitete zraka u zatvorenim prostorima. Kao web programski okvir korišten je ASP.NET Core u kombinaciji sa MVC oblikovnim obrascom. Web aplikacija omogućava prikupljanje podataka sa više uređaja i sve prikupljene podatke skladišti u bazu podataka. Zatim se na klijentskom segmentu aplikacije izvode jednostavne vizualizacije podataka po zatvorenoj prostoriji.

Ključne riječi: Docker, .NET, MVC

8. ABSTRACT

Title: Development of a web application for monitoring presence and air quality in enclosed spaces.


The main idea is to create a web application which will primarily be used to monitor presence, air quality, temperature and humidity in an enclosed space. ASP.NET Core was used as main framework with MVC design pattern. Web application collects data from multiple devices and stores all data in database. Then, on the client side of application, visualizations of data are performed.

Keywords: Docker, .NET, MVC

9. PRILOZI

IZJAVA O AUTORSTVU ZAVRŠNOG RADA

Pod punom odgovornošću izjavljujem da sam ovaj rad izradio/la samostalno, poštujući načela akademske čestitosti, pravila struke te pravila i norme standardnog hrvatskog jezika. Rad je moje autorsko djelo i svi su preuzeti citati i parafraze u njemu primjereno označeni.

Mjesto i datum	Ime i prezime studenta/ice	Potpis studenta/ice
U Bjelovaru, <u>16. srpnja 2021.</u>	Ivan Kranjec	

Prema Odluci Veleučilišta u Bjelovaru, a u skladu sa Zakonom o znanstvenoj djelatnosti i visokom obrazovanju, elektroničke inačice završnih radova studenata Veleučilišta u Bjelovaru bit će pohranjene i javno dostupne u internetskoj bazi Nacionalne i sveučilišne knjižnice u Zagrebu. Ukoliko ste suglasni da tekst Vašeg završnog rada u cijelosti bude javno objavljen, molimo Vas da to potvrdite potpisom.

Suglasnost za objavljivanje elektroničke inačice završnog rada u javno dostupnom nacionalnom repozitoriju

Ivan Kranjec

ime i prezime studenta/ice

Dajem suglasnost da se radi promicanja otvorenog i slobodnog pristupa znanju i informacijama cjeloviti tekst mojeg završnog rada pohrani u repozitorij Nacionalne i sveučilišne knjižnice u Zagrebu i time učini javno dostupnim.

Svojim potpisom potvrđujem istovjetnost tiskane i elektroničke inačice završnog rada.

U Bjelovaru, 16. srpnja 2021.



potpis studenta/ice