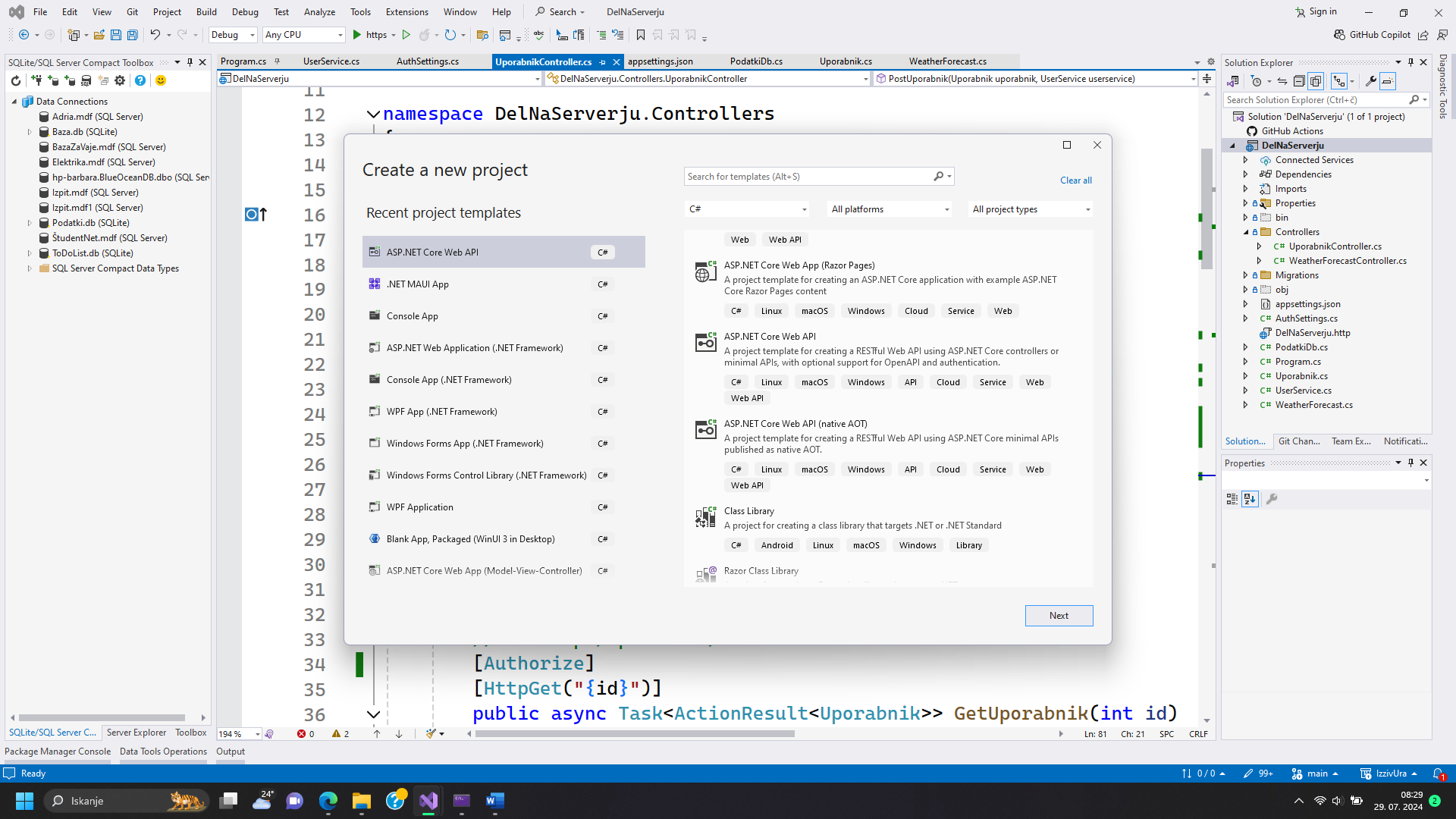
# Spletna aplikacija za delo z uporabniki

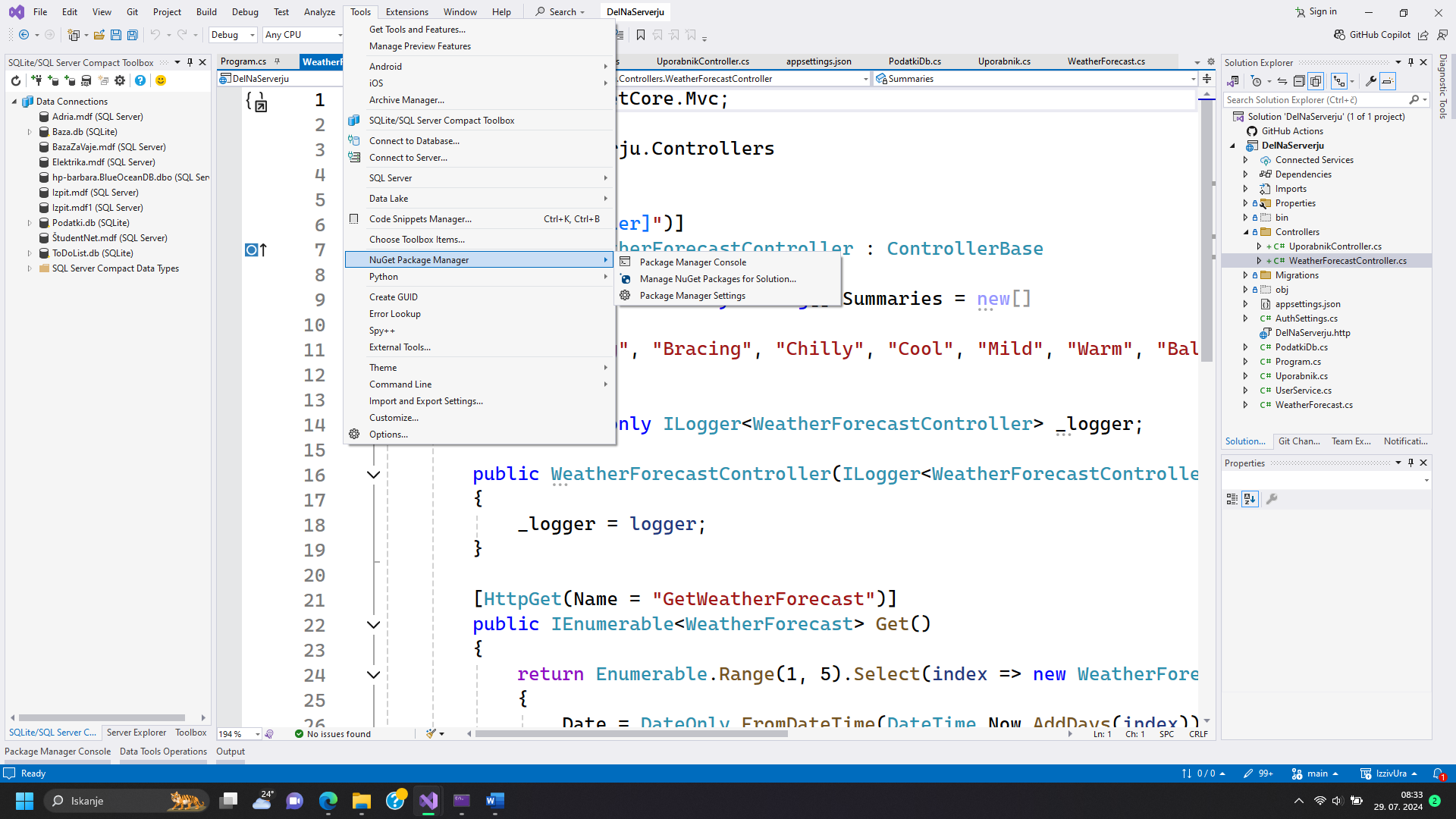
Napisati bi želeli spletno aplikacijo, ki vsebuje servise za delo z uporabniki. Uporabnik se bo **registriral** (ime – edinstveno, geslo, jeAktiven?). Po registraciji se bo njegovo ime **vpisalo v podatkovno bazo**. Ob prijavi z imenom in geslom, bo dobil **žeton**, ki ga bo lahko mobilna aplikacija uporabila za dostop do registracije delovnega časa. Pri tem si moramo odgovoriti na nekaj vprašanj. Kako poskrbeti za varnost podatkov, ki se prenašajo po omrežju? Kako zapisati v podatkovno bazo geslo? Možnosti je tukaj več. Uporabili bomo JWT (jason web token) za kreiranje žetonov in t.i. zgoščevanje gesla (iz niza v geslu izračunamo številko-»hashiranje«).

V Visual studiu izdelamo novo aplikacijo ASP. NET Core Web API



Izdela se nam ogrodje aplikacije, ki že vsebuje primer spletnega servisa za vreme.

Najprej si dodamo pakete, ki mi bodo omogočali delo s podatkovno bazo. V tem primeru bom uporabljali MSSQL podatkovno bazo. Odpremo si »Package Manager Console« (Tools, NuGetPackage Manager, Package Managager Console)



Install-Package Microsoft.EntityFrameworkCore

Install-Package Microsoft.EntityFrameworkCore.SqlServer

Install-Package Microsoft.EntityFrameworkCore.Tools

Install-Package Microsoft.EntityFrameworkCore.Design

Vsi paketi nam omogočajo lažje delo s podatkovno bazo.

Najprej si izdelamo razred, ki bo vseboval podatke o uporabniku:

public class Uporabnik

{

public int Id { get; set; }

public string Ime { get; set; }

public string KodiranoGeslo { get; set; }

public bool JeAktiven { get; set; }

}

Nato pa izdelamo še razred, ki vsebuje podatke o podatkovni bazi kot celoti.

public class PodatkiDb:DbContext

{

public PodatkiDb(DbContextOptions<PodatkiDb> options)

: base(options)

{

}

public virtual DbSet<Uporabnik> Uporabniki { get; set; }

protected override void OnModelCreating(ModelBuilder builder)

{

builder.Entity<Uporabnik>()

.HasIndex(u => u.Ime)

.IsUnique();

}

}

Kar nam preostane je, da naši aplikaciji povemo, da želimo uporabljati podatkovno bazo na MSSQL strežniku in dodamo še ime podatkovne baze. Najprej odpremo Program.cs in dodamo kodo (v rumenem)

builder.Services.AddControllers();

builder.Services.AddEndpointsApiExplorer();

var connectionString = builder.Configuration.GetConnectionString("PodatkiConn");

builder.Services.AddDbContext<PodatkiDb>(options => options.UseSqlServer(connectionString));

Naša podatkovna baza se bo imenovala Podatki. Tu smo samo zapisali, kje je kontekst podatkovne baze (slika naše baze v programu), dodali smo pa še povezovalni niz. (Če uporabljamo drugo bazo, se ta del lahko razlikuje).

Odpremo še datoteko **appsettings.json** in dodamo (kar je rumeno):

{

"Logging": {

"LogLevel": {

"Default": "Information",

"Microsoft.AspNetCore": "Warning"

}},

"ConnectionStrings": {"PodatkiConn":

"Data Source=(LocalDB)\\MSSQLLocalDB;AttachDbFilename="D:\\Challenger\\IzzivUra\\Podatki.mdf;Integrated Security=True;Connect Timeout=30; TrustServerCertificate=True;"},

"AllowedHosts": "\*"

}

Sedaj pa podatkovno bazo dejansko izdelamo.

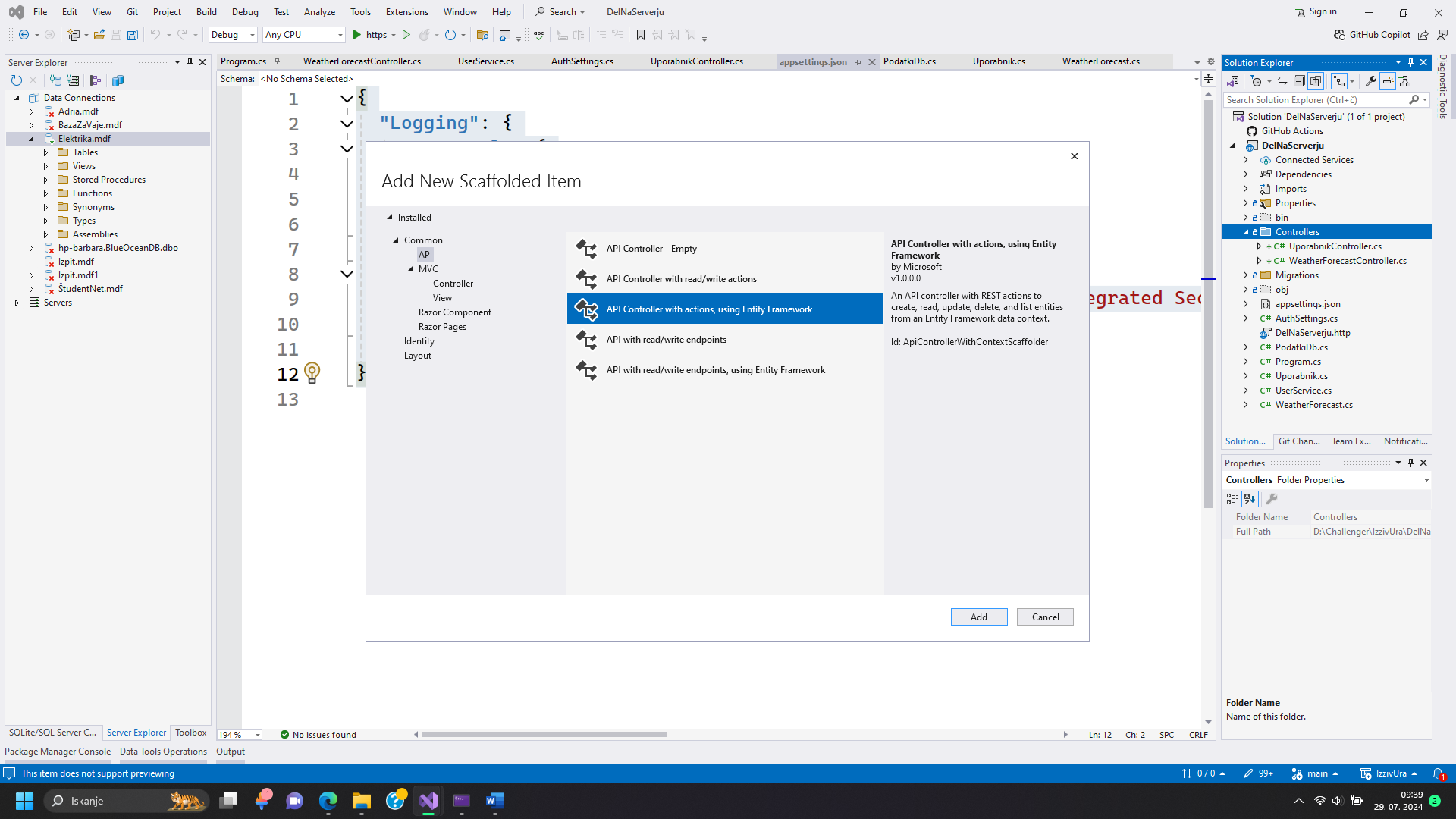
V »Package manager console« vtipkamo

Add-migration imeMigracije

Update-database

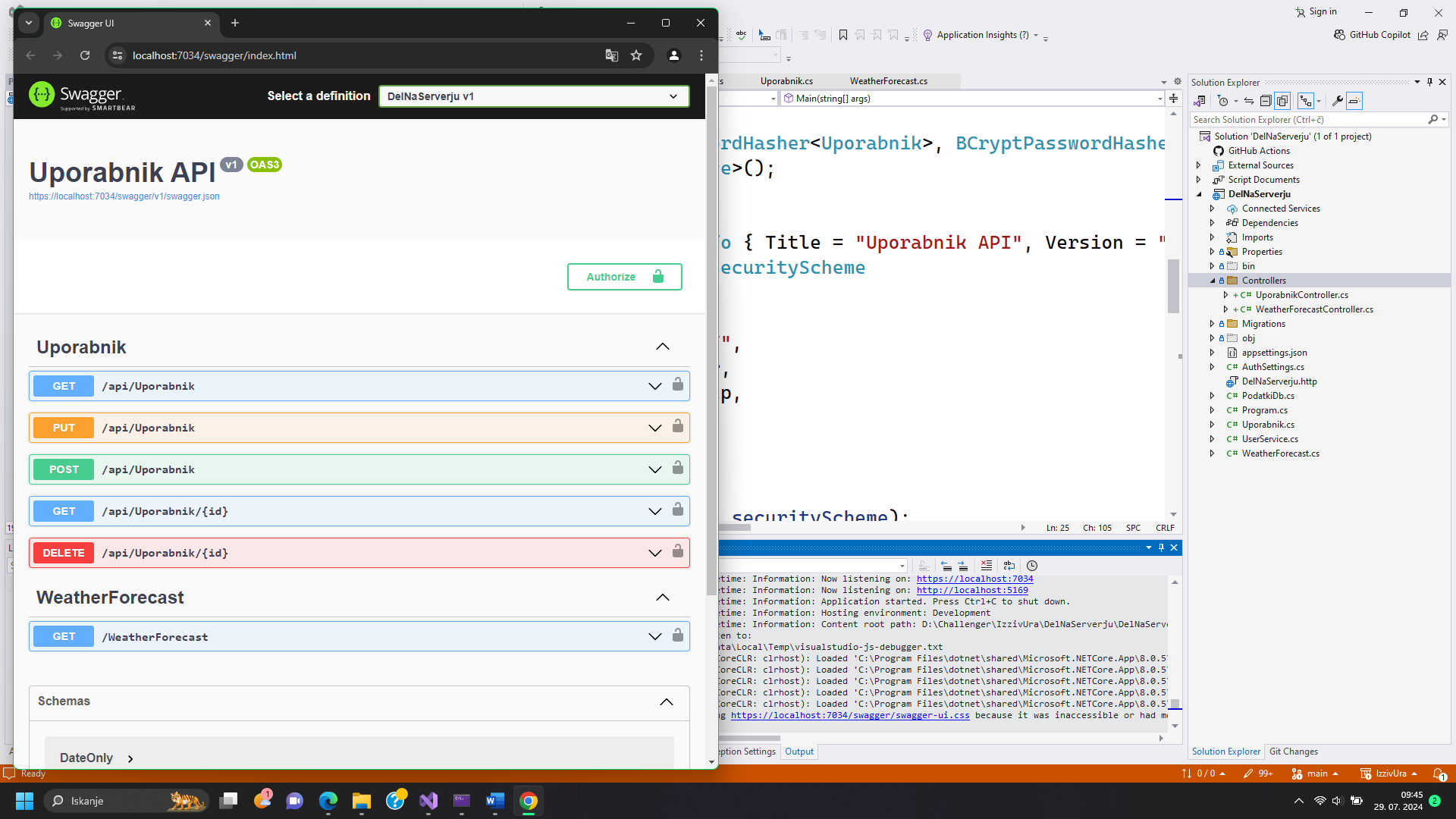
Če nam je šlo vse v redu, smo dobili podatkovno bazo z eno tabelo, ki se imenuje Uporabniki, vsak uporabnik ima lastnosti, ki smo jih našteli v razredu Uporabnik. Če pogledamo podatke v tabeli, vidimo, da je prazna.

Sedaj bi pa želeli napisati kodo, ki mi bo omogočala delo z uporabniki (vnos uporabnikov, popravljanje uporabnikov, pregled uporabnikov in brisanje uporabnikov). Na srečo nam kode ni potrebno pisati. V mapi »Controllers« dodamo nov kontroler.



V naslednjem koraku izberemo model (pri nas je to Uporabnik) in kontekst (pri nas je to PodatkiDb). Spremenimo lahko tudi ime in sicer v UporabnikController.

Sedaj lahko že preizkusimo delovanje. Ker so spletni servisu v bistvu programi brez uporabniškega vmesnika, uporabimo Swagger. Vse potrebno za uporabo Swaggerja se nam je že izdelalo ob izdelavi ogrodja aplikacije. Kar pomeni, da lahko program kar poženemo.



Vsakega od klicev lahko preskusimo. Ko kličemo POST (naslov: /api/Uporabnik) dodajamo novega uporabnika. Poskusite dodati enega uporabnika.

Do sedaj smo izdelali podatkovno bazo z eno tabelo. Sedaj pa poskrbimo za varnost. Odpremo UporabnikController in pregledamo kodo. Prej smo videli, da POST dodaja uporabnike. Le tu bi rada, da preveri ali morda uporabnik s takim imenom že obstaja, obenem bi geslo rada spremenila v neberljivo obliko. Za skrivanje gesla bomo uporabili že vnaprej pripravljen razred [scottbrady91/ScottBrady91.AspNetCore.Identity.BCryptPasswordHasher: ASP.NET Core Identity IPasswordHasher implementation using BCrypt.Net - next (github.com)](https://github.com/scottbrady91/ScottBrady91.AspNetCore.Identity.BCryptPasswordHasher)

(Podrobnosti računanja hasha tu ne bomo opisovali).

Za skrivanje našega gesla moramo dodati spodnjo kodo:

1. Najprej si bomo izdelali poseben razred, kjer bomo shranili vse metode potrebne za delo z gesli UserService.cs
2. Dodamo paket ScottBrady91.AspNetCore.Identity.BCryptPasswordHasher
3. V razred dodamo kodo:

public class UserService

{

private readonly PodatkiDb \_context;

private readonly IPasswordHasher<Uporabnik> \_passwordHasher;

private readonly IConfiguration \_configuration;

public UserService(PodatkiDb context, IPasswordHasher<Uporabnik> passwordHasher, IConfiguration configuration)

{

\_context = context;

\_passwordHasher = passwordHasher;

\_configuration = configuration;

}

public void CreateUser(Uporabnik user)

{

\_context.Uporabniki.Add(user);

\_context.SaveChanges();

}

public Uporabnik GetUserByUsername(string username)

{

return \_context.Uporabniki.SingleOrDefault(u => u.Ime == username);

}

public string HashPassword(string password)

{

return \_passwordHasher.HashPassword(null, password);

}

public bool VerifyPassword(string password, string hashedPassword)

{

return \_passwordHasher.VerifyHashedPassword(null, hashedPassword, password) == PasswordVerificationResult.Success;

}

}

1. Spremenimo metodo PostUporabnik v UporabnikControoler.cs

[HttpPost]

public async Task<ActionResult> PostUporabnik(Uporabnik uporabnik,UserService userservice)

{

var obstaja=await \_context.Uporabniki.Where(e=>e.Ime == uporabnik.Ime).FirstOrDefaultAsync(); //preverimo, da se uporabnik ne podvaja

if (obstaja != null)

{

return BadRequest("Uporabniško ime mora biti edinstveno");

}

var user = new Uporabnik

{

Ime = uporabnik.Ime,

KodiranoGeslo = userservice.HashPassword(uporabnik.KodiranoGeslo),

JeAktiven = uporabnik.JeAktiven

};

userservice.CreateUser(user);

return Ok();

}

1. Spremeniti moramo še Program.cs, da registriamo naš servis (UserService). Kodo dodamo **nad** builder.Services.AddSwaggerGen(…)

builder.Services.AddSingleton<IPasswordHasher<Uporabnik>, BCryptPasswordHasher<Uporabnik>>();

builder.Services.AddScoped<UserService>();

Preden poženemo program (ta del kode nam geslo skrije in skrito geslo shrani v podatkovno bazo), dodajmo še izdelavo žetonov.

Spletni žetoni JSON (JSON web token =JWT) so odprta, industrijska standardna metoda (dokumentirana v RFC 7519) za varno predstavljanje zahtevkov med dvema strankama. Sestavljeni so iz treh delov, ločenih s piko:

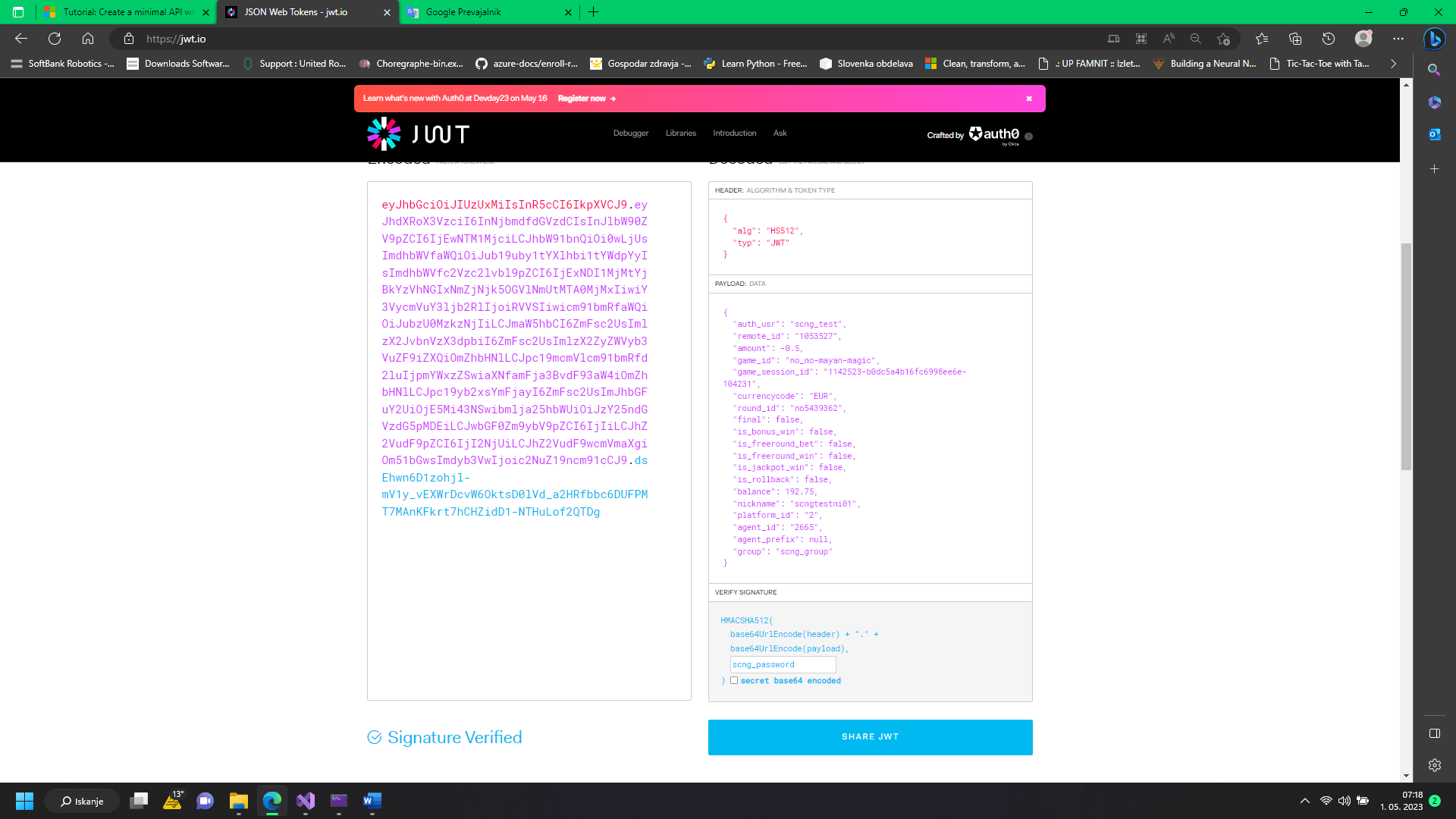
xxxx.yyyy.zzz

xxx=glava (header), navadno vsebuje tip podpisnega algoritma (HS256,HS512) in tip žetona (JWT)

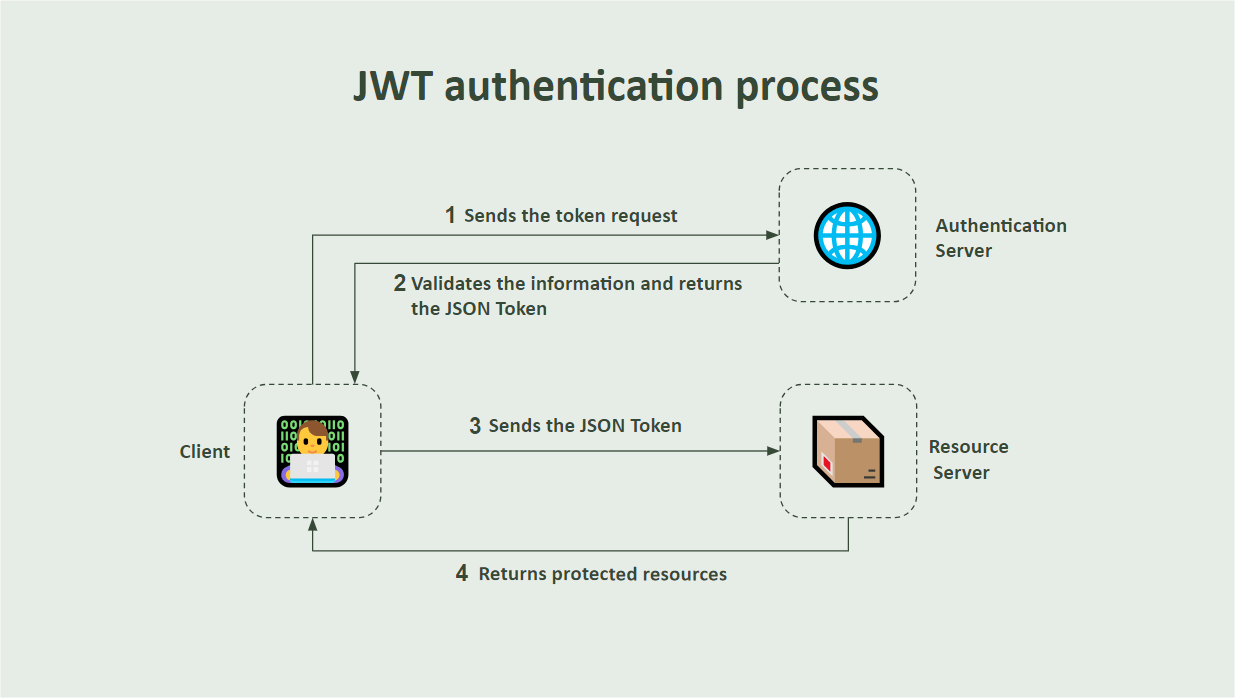
yyy=vsebina (payload) : struktura in podatki prenesenega

zzz=podpis (signature), kjer je zakodirana glava, vsebina in dodan skrivni ključ

Primer: [JSON Web Tokens - jwt.io](https://jwt.io/), dodala še skrivni ključ in dobila vsebino jwt-ja.

Kako

Kako poteka avtentikacija uporabnika z JWT-jem?



Tudi za izdelavo JWT-jev imamo vnaprej pripravljene razrede. Dodamo si z NugetPackage managerjem.

System.IdentityModel.Tokens.Jwt (to je verjetno že prisotno, ni treba nameščati?)

Microsoft.AspNetCore.Authentication.JwtBearer

(NE!! Microsoft.IdentutyModel.Tokens – dela ob 8. uri 30.7.2024)

Najprej si izdelamo razred, kjer hranimo samo podpis. Ta del bi morali skriti, ko bo aplikacija nameščena.

public class AuthSettings

{

public static string PrivateKey { get; set; } = "";

}

V UserService.cs dodamo kodo za generiranje JWT žetona

public string GenerateJwtToken(Uporabnik user)

{

var tokenHandler = new JwtSecurityTokenHandler();

var key = Encoding.ASCII.GetBytes(AuthSettings.PrivateKey);

var credentials = new SigningCredentials(

new SymmetricSecurityKey(key),

SecurityAlgorithms.HmacSha256Signature);

var tokenDescriptor = new SecurityTokenDescriptor

{

Subject = new ClaimsIdentity(new Claim[]

{

new Claim(ClaimTypes.Name, user.Ime)

}),

Expires = DateTime.UtcNow.AddHours(1),

SigningCredentials = credentials

};

var token = tokenHandler.CreateToken(tokenDescriptor);

return tokenHandler.WriteToken(token);

}

Nato moramo v programu dodati še avtentikacijo (rumeno) in nekoliko popraviti delovanje swaggerja (ta del je namenjen samo testiranju)

using Microsoft.AspNetCore.Authentication.JwtBearer;

using Microsoft.AspNetCore.Identity;

using Microsoft.EntityFrameworkCore;

using Microsoft.IdentityModel.Tokens;

using Microsoft.OpenApi.Models;

using ScottBrady91.AspNetCore.Identity;

using System.Text;

using System.IdentityModel.Tokens.Jwt;

namespace DelNaServerju

{

public class Program

{

public static void Main(string[] args)

{

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddControllers();

builder.Services.AddEndpointsApiExplorer();

var connectionString = builder.Configuration.GetConnectionString("PodatkiConn");

builder.Services.AddDbContext<PodatkiDb>(options => options.UseSqlServer(connectionString));

builder.Services.AddAuthentication(options =>

{

options.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;

options.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;

})

.AddJwtBearer(options =>

{

options.RequireHttpsMetadata = false;

options.SaveToken = true;

options.TokenValidationParameters = new TokenValidationParameters

{

ValidateIssuer = false,

ValidateAudience = false,

ValidateLifetime = true,

IssuerSigningKey = new SymmetricSecurityKey(Encoding.ASCII.GetBytes(AuthSettings.PrivateKey))

};

});

builder.Services.AddAuthorization();

builder.Services.AddSingleton<IPasswordHasher<Uporabnik>, BCryptPasswordHasher<Uporabnik>>();

builder.Services.AddScoped<UserService>();

builder.Services.AddSwaggerGen(c =>

{

c.SwaggerDoc("v1", new OpenApiInfo { Title = "Uporabnik API", Version = "v1" });

var securityScheme = new OpenApiSecurityScheme

{

Name = "JWT Avtentikacija",

Description = "Vnesi svoj JWT",

In = ParameterLocation.Header,

Type = SecuritySchemeType.Http,

Scheme = "bearer",

BearerFormat = "JWT"

};

c.AddSecurityDefinition("Bearer", securityScheme);

var securityRequirement = new OpenApiSecurityRequirement

{

{

new OpenApiSecurityScheme

{

Reference = new OpenApiReference

{

Type = ReferenceType.SecurityScheme,

Id = "Bearer"

}

},

new string[] {}

}

};

c.AddSecurityRequirement(securityRequirement);

});

var app = builder.Build();

// Configure the HTTP request pipeline.

if (app.Environment.IsDevelopment())

{

app.UseSwagger();

app.UseSwaggerUI();

}

app.UseAuthentication();

app.UseAuthorization();

app.UseHttpsRedirection();

app.MapControllers();

app.Run();

}

}

}

(dodala sem celotno kodo metode Main). Kar nam preostane je, da v kontrolerju popravimo metodo PutUporabnik, uporabnik bo vnesel ime in geslo, servis bo vrnil JWT žeton.

[HttpPut]

public async Task<IActionResult> PutUporabnik(Uporabnik uporabnik,UserService userService)

{

var user = userService.GetUserByUsername(uporabnik.Ime);

if (user != null && userService.VerifyPassword(uporabnik.KodiranoGeslo, user.KodiranoGeslo))

{

var token = userService.GenerateJwtToken(user);

return Ok(token);

}

return BadRequest("Ne morem ustvariti žetona");}

Kako sedaj vse skupaj testiramo. Uporabnik bo vnesel ime in geslo, tako da bo klical PutUporabnik, če bo uporabnik obstajal in bo geslo pravilno, bo metoda vrnila žeton. Kakšna je dejansko vsebina vašega žetona lahko preverite na [JSON Web Tokens - jwt.io](https://jwt.io/) . (videti morate uporabnika in njegovo skrito geslo)

Denimo, da bi radi nekaj izpisali za veljavno prijavljenega uporabnika. V ta namen samo metodi dodamo atribut [Authorize]

[Authorize]

[HttpGet]

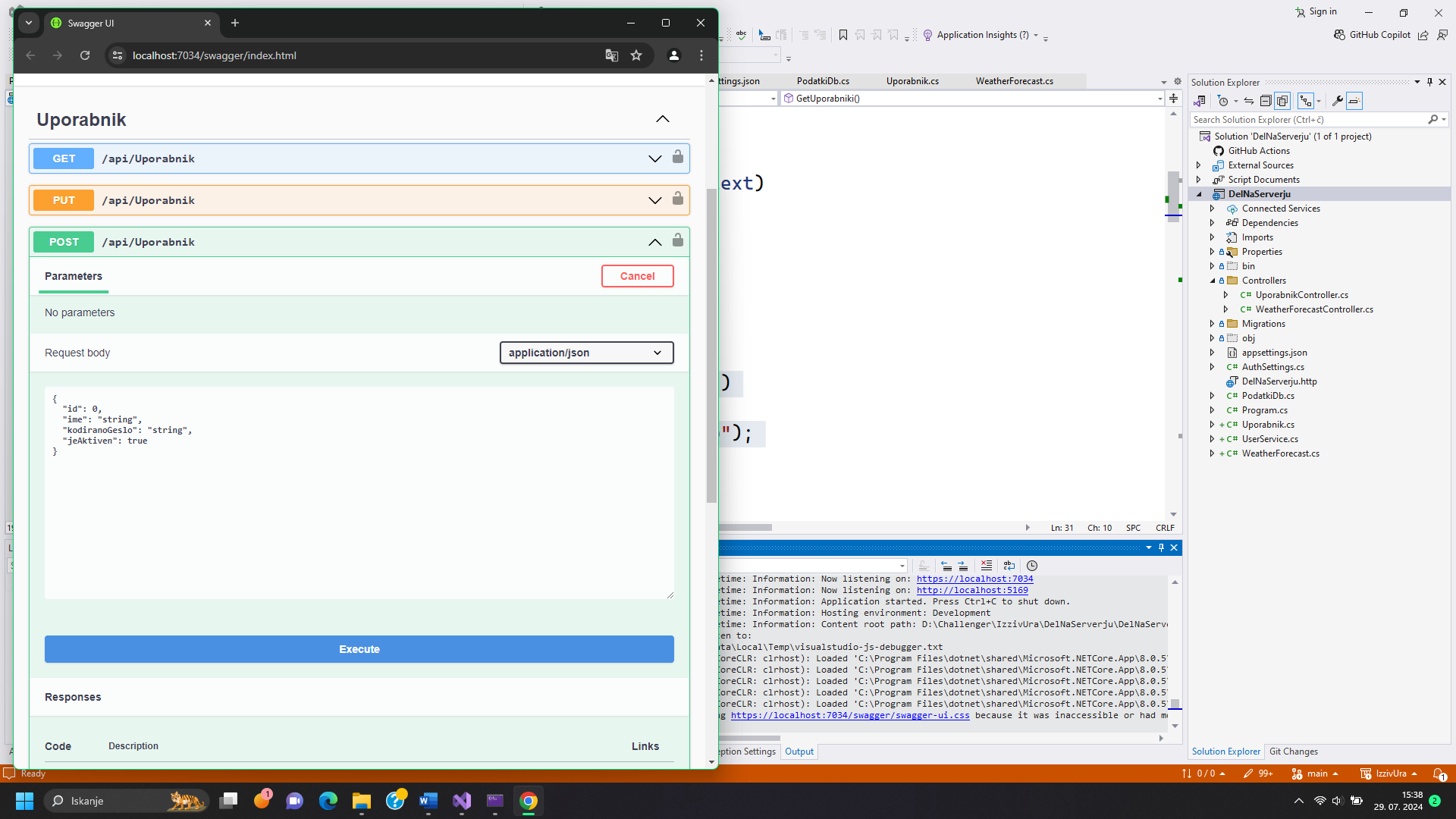
public async Task<IResult> GetUporabniki()

{

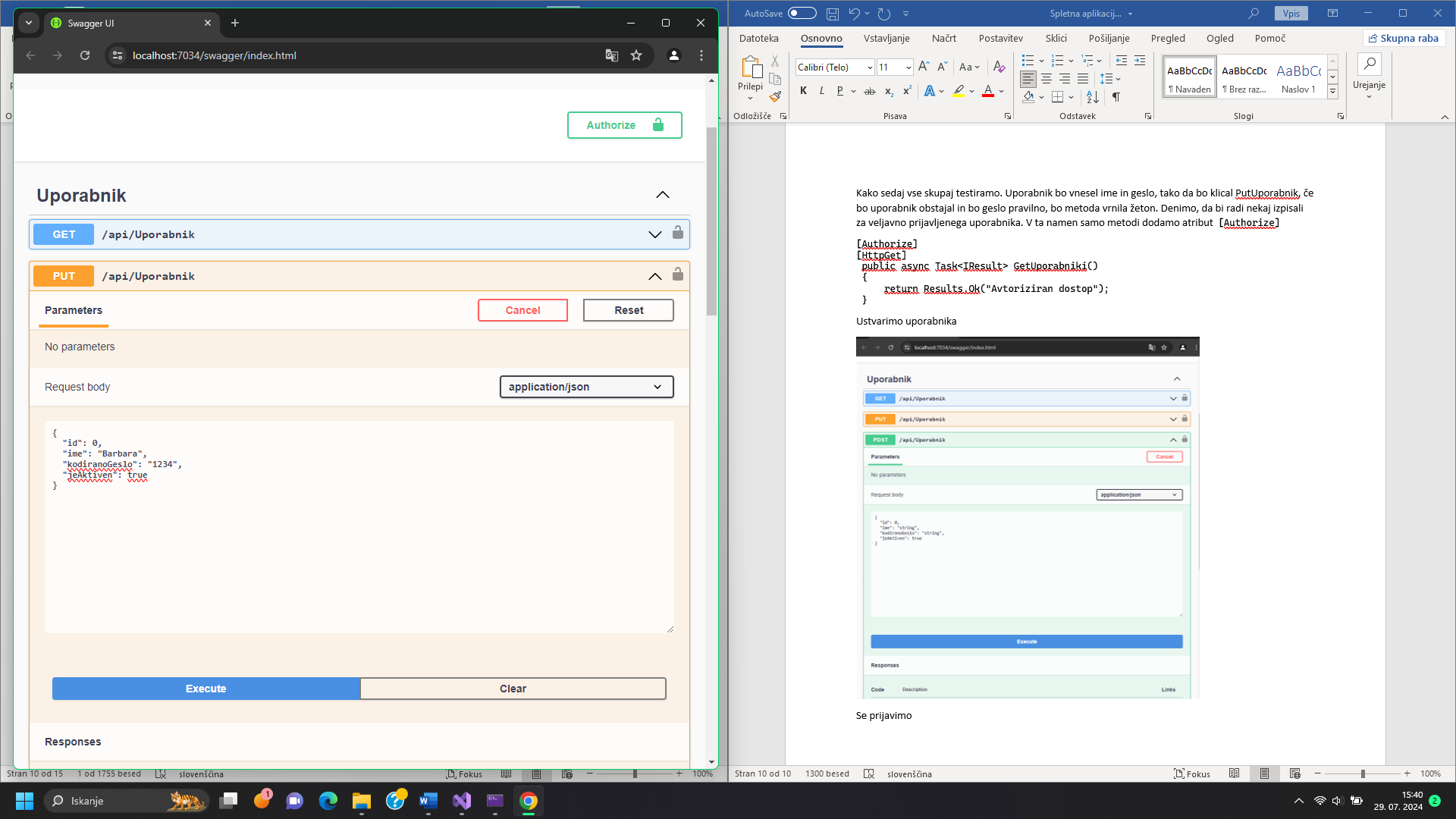
return Results.Ok("Avtoriziran dostop");

}

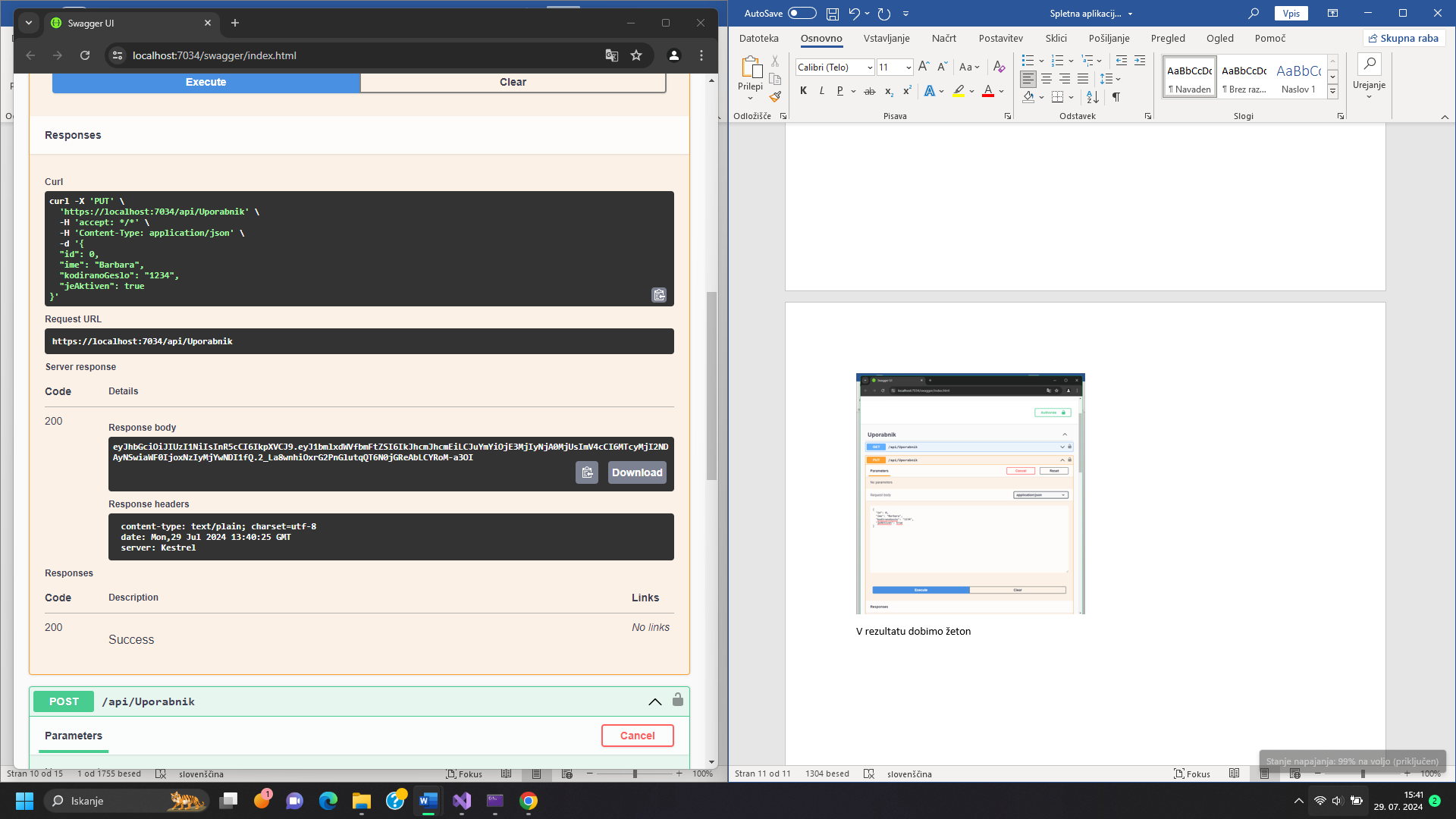
Ustvarimo uporabnika



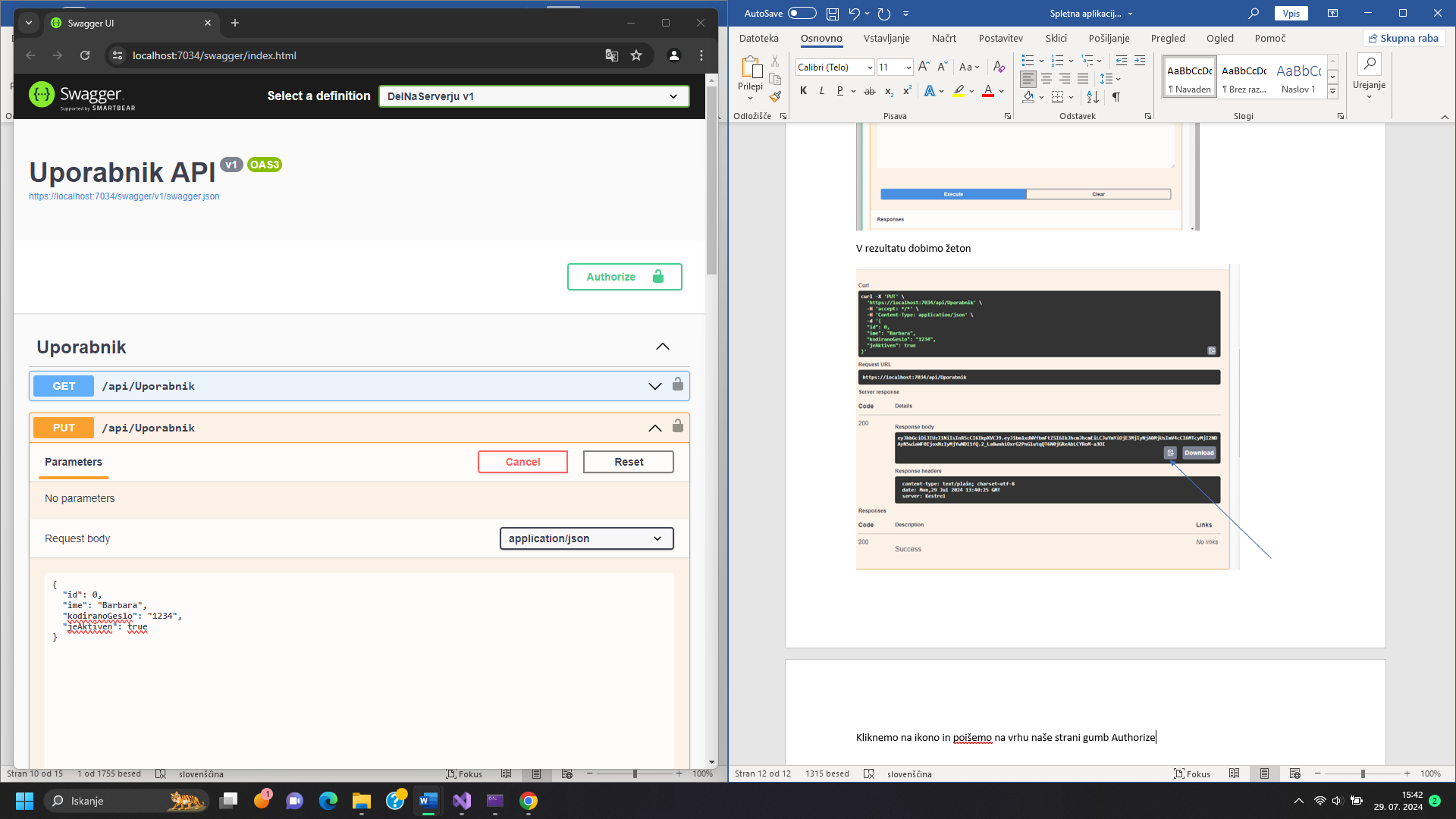
Se prijavimo



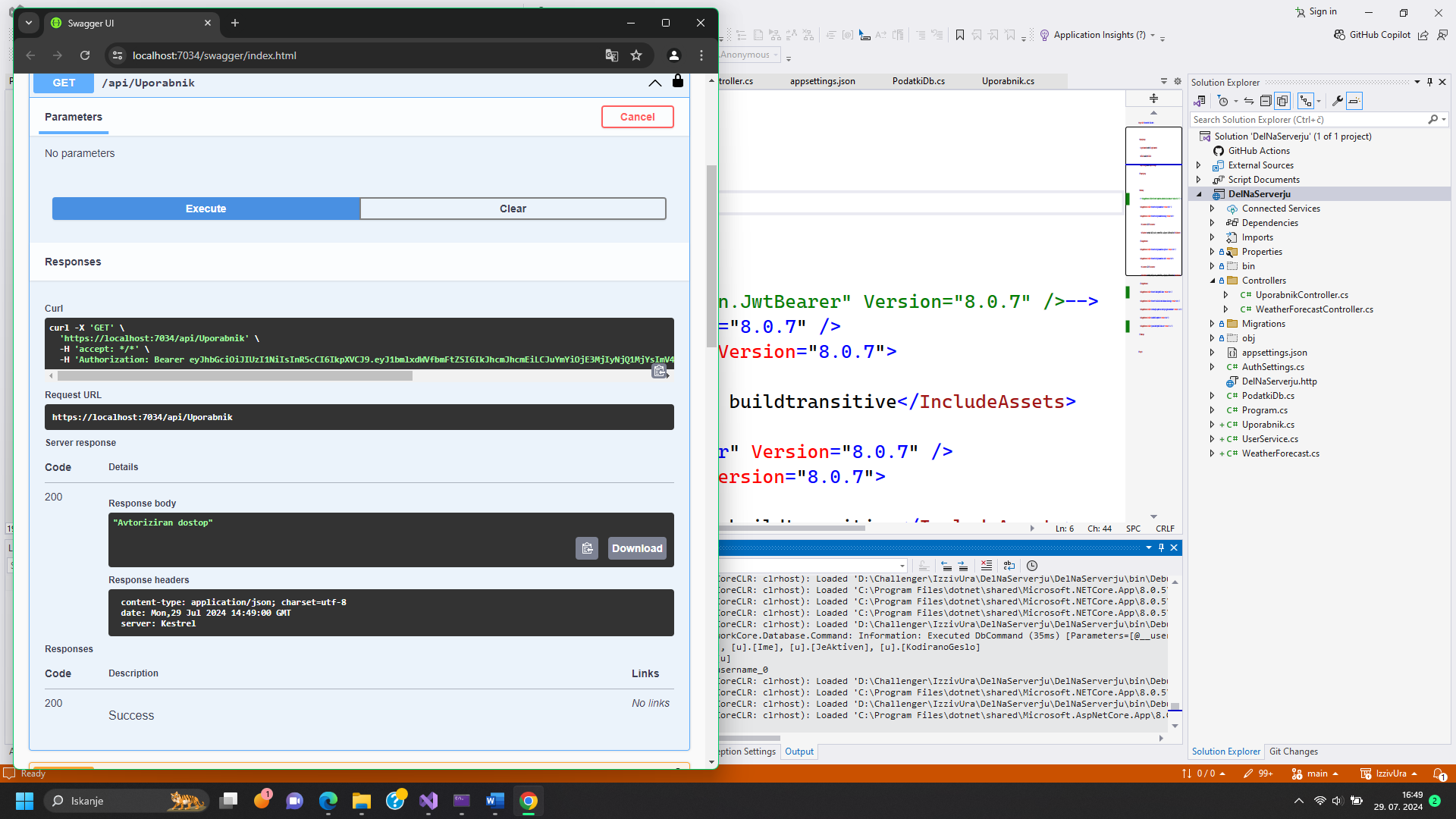
V rezultatu dobimo žeton



Kliknemo na ikono »Kopiraj« in poiščemo na vrhu naše strani gumb Authorize



Kliknemo na gumb in vnesemo žeton. Sedaj lahko poženemo metodo, ki zahteva avtorizacijo.



Poskusite še sami. Glede na vaš izziv dodajte še en razred, posodobite podatkovno bazo in dodajte API klice za vaš razred. Vsi klici naj uporabljajo avtorizacijo. (samo nekdo, ki ima žeton lahko kliče metode).

Primer: dodati hočemo Kolo.cs

public class Kolo

{

public int Id { get; set; }

public string Znamka { get; set; }

public string Slika { get; set; }

public virtual Uporabnik Lastnik { get; set; }//za FK

public decimal TrentnaLokacijaLongitude { get; set; }

public decimal TrentnaLokacijaLatitude { get; set; }

}

Naredimo migracijo in posodobimo bazo.

Naredimo nov kontroler.

Povsod dodamo [Authorize] atribut.

Testiramo delovanje.

Kako dodati sliko?

[How do I upload an image in ASP.NET Core WEB API with other form data? - Stack Overflow](https://stackoverflow.com/questions/73498624/how-do-i-upload-an-image-in-asp-net-core-web-api-with-other-form-data)