# Angular in .NET kot konkurenca namiznim aplikacijam

Matjaž Prtenjak

Endava d.o.o, Ljubljana, Slovenija
matjaz@matjazev.net

V kolikor ste spremljali moje prispevke v prejšnjih letih, boste presenečeni, saj bo ta prispevek zadeve obrnil na glavo. V kolikor pa ste z mano prvič, pa tudi ne skrbite, saj je prispevek povsem zaključena celota in neodvisen od prejšnjih prispevkov. V prispevkih prejšnjih let smo se spraševali, kako (starejše) razvijalce, ki so navajeni namiznih aplikacij, naučiti dela v spletnih aplikacijah, danes pa se sprašujemo, kako spletno aplikacijo napisati tako, da jo lahko damo na namizje. Vprašanje, ki se smiselno pojavlja je: »Zakaj bi to sploh počeli?«. Zakaj bi želeli spletno aplikacijo imeti kot namizno aplikacijo oz. jo gostiti sami? Odgovori so lahko različni, najpomembnejši pa je: zaupanje. Obstajajo uporabniki – tudi (ali predvsem) v poslovnem svetu, ki želijo imeti aplikacije pri sebi. Če želimo spletno aplikacijo prenesti k uporabniku, je zadeva dokaj preprosta. Uporabnik pri sebi postavi spletni strežnik, ki je dosegljiv samo v njegovem poslovnem okolju in aplikacija se prenese na ta strežnik in s tem je zadeva bolj ali manj zaključena. Kaj pa, če ne želimo postavljati spletnega strežnika? Ali bi ne bilo bolje, če lahko uporabnik naloži aplikacijo s spleta, jo požene in slednja teče? Ali pa jo celo kopira v mapo, požene in zadeva deluje?

**Ključne besede:**
.NET
Angular
Spletne aplikacije
Namizne aplikacije
Hibridne aplikacije

Univerzitetna založba Univerze v Mariboru
ISBN 978-961-286-894-9
DOI https://doi.org/10.18690/um.feri.4.2024.19

# 1 Uvod

V tem prispevku bomo razvili spletno/namizno aplikacijo z uporabo .NET 8.0 in Angular okolja. Seveda programska koda ne bo v članku, temveč jo lahko snamete s spleta, z GitHub-a, na naslovu: https://github.com/MPrtenjak/OTS2024-NET-Angular-WEB-And-Desktop.

## 1.1. Namen aplikacije

Razviti želimo aplikacijo, ki bo hranila tri posebnosti, za katere so uporabniki hvaležni v določenem dnevu. Dandanes je v svetu namreč vse več depresije in anksioznosti, zato nam lahko lepe stvar izboljšajo počutje. Če nam je namreč hudo, se lahko na hitro spomnimo, zakaj smo bili hvaležni včeraj, prejšnji mesec ali pa mogoče leto dni nazaj.

## 1.2. Zahteve

- Ker gre za osebna čustva, se morajo uporabniki v aplikacijo najprej prijaviti. Prijava je klasična, z imenom in geslom.
- Znotraj aplikacije lahko uporabniki brskajo po zgodovini in vidijo, česa so se veselili v preteklosti oz. čemu so bili hvaležni.
- Za vsak dan lahko vnesejo do tri posebnosti. *Tri je povsem dovolj, saj sicer posebnosti niso več posebnosti in gre za preprosto naštevanje.*
- Posebnosti lahko uporabnik vnaša za tekoči datum ali pa do največ tri dni v preteklosti; skupaj torej za največ štiri dni. *Tudi tukaj se strogo omejimo, saj je vnašanje za več kot tri dni v preteklosti že podleženo pozabljanju.*
- Aplikacija mora teči kot spletna aplikacija v npr. Azure, spletna aplikacija v Windows in Linux okolju ali pa kot navadna aplikacija znotraj Windows ali Linux operacijskega sistema.

## 1.3. O razvojnih okoljih

Opisane zahteve lahko rešimo na zelo različne načine in z zelo različnimi orodji.

Ena izmed možnosti je, da razvijemo spletno aplikacijo v poljubnem okolju, jo »zapečemo« v Docker kontejner in ponudimo kot »kontejnersko« aplikacijo. Ta zagotovo izpolnjuje zahteve, da teče »kjerkoli«. Toda v tem primeru mora uporabnik imeti Docker in podobno. Poskusimo biti torej čimbolj preprosti in prijazni do uporabnika.

Če torej ne razmišljamo o kontejnerskih aplikacijah, smo lahko še vedno agnostnični pri izbiti razvojnega okolja. Lahko razvijemo vse lepo v Python programskem jeziku in v ukazni vrstici (lahko sicer uporabimo tudi kakšno orodje, kot je PySimpleGUI za izdelavo grafičnega vmesnika). Lahko vse razvijemo v JavaScript in Node okolju. Lahko…

Povedati želim, da je glede na opisane zahteve rešitev mnogo, saj imamo različne programske jezike. Zato je smiselno pač pogledati, kaj znamo, v čem smo domači in potem aplikacijo razvijemo v tem okolju. Meni je blizu okolje .NET, zato bom izbral slednjega.

Vse lahko razvijemo v .NET okolju, saj imamo za grafični del aplikacije v .NET okolju res veliko izbiro. Ker želimo, da je aplikacija spletna, bi recimo izbrali Blazor, ki sem ga predstavljal v prispevkih prejšnja leta [3, 4].

Vendar spet poglejmo malo naokoli in ugotovili bomo, da je današnjim mladim programerjem/programerkam še najbližji spletni razvoj z uporabo različnih JavaScript razvojnih okolij, kot so React, Angular, Vue in podobni. Če sem pred leti torej predstavil Vue [5], pa dajmo danes zadevo razviti v Angular okolju.

### 1.4. Dodatne omejitve

Razvojno okolje smo torej izbrali. Imeli bomo kombinacijo .NET in Angular. Da pa bi programska koda ne bila prevelika in bi jo tisti, ki vas to zanima, lahko relativno hitro pogledali in preverili, se omejimo še malce bolj.

Našo aplikacijo bomo torej gostili na Azure in v tem primeru bomo govorili o spletni aplikaciji. Lahko pa bomo našo aplikacijo pognali v okolju Windows ali Linux in v tem primeru bomo govorili o namizni aplikaciji in/ali spletni aplikaciji.

- Razvili bomo aplikacijo, ki bo lahko tekla kot »navaden program« v okolju Windows ali kot storitev (service) v okolju Windows. Spletnega strežnika IIS se v Oknih ne bomo dotikali, ker uporabniki (navadni uporabniki) pač nimajo nameščenega IIS na računalnikih. Kadar bo aplikacija tekla kot »navaden program«, bo namizna aplikacija, kadar bo tekla kot service, bo spletna aplikacija.
- Lahko bo tekla v okolju Linux kot navaden program, ki ga požene konkretni uporabnik ali kot storitev (deamon), ki jo uporabljajo drugi uporabniki. Torej spet kot namizna ali spletna aplikacija.
- Lahko pa bo tekla kot spletna aplikacija v Azure.

In zdaj še dve posebni, a logični zahtevi:

- Programska koda bo enotna in poskušali jo bomo napisati tako, da bomo lahko program samo kopirali v neko mapo, pa bo že deloval.
- Če uporabnik/uporabnica poganja aplikacijo na svojem računalniku, potem ne potrebuje prijave, saj je že prijavljena v sistem. Tukaj imam v mislih opcije, ko program teče v Windows okolju (kot program ali kot storitev) ter kadar teče v Linux okolju kot program.

## 2 Razvoj

Kot že večkrat omenjeno, bomo aplikacijo razvili v dveh okoljih, ki bosta povezani z REST API komunikacijo. Razvoj lahko torej poteka vzporedno, dogovoriti se moramo samo za aplikacijski vmesnik (REST API), preko katerega bosta oba dela naše aplikacije komunicirala.

### 2.1. API

Pri aplikacijskem vmesniku se bomo spet, da bo aplikacija majhna in obvladljiva, držali minimalnih zahtev. Uporabili bomo torej sledeče REST zahteve:

- **POST**: `/users/sign-up`, ki služi za registracijo uporabnika
  - Vhod: `{ "userName", "password1", "password2" }`
  - Izhod: `{ "userId", "userName", "token" }`
  - Na vhodu torej pričakujemo uporabniško ime in dve gesli, ki se morata ujemati. Na izhodu pa vrnemo id in ime novega uporabnika v bazi ter JWT žeton.
- **POST**: `/users/try-login`, ki služi za prijavo uporabnika brez prijavnih podatkov

  - Vhod: nima
  - Izhod: `{ "userId", "userName", "token" }`
  - Ta funkcija služi specifičnemu namenu v naši aplikaciji. Zahteva je namreč, da se uporabnik v primeru namizne aplikacije ne prijavlja in uporabili bomo ta klic, da bomo poskušali uporabnika prijaviti brez prijavnih podatkov. Več o tem kasneje v prispevku.
- **POST**: `/users/login`, ki služi za prijavo uporabnika z uporabniškim imenom in geslom
  - Vhod: `{ "userName", "password" }`
  - Izhod: `{ "userId", "userName", "token" }`
- **GET**: `/admin/info`
  - Vrne JSON objekt z različnimi informacijami o trenutnem sistemu
- **POST**: `/admin/randomize-data`, ki bazo zapiše naključne hvaležnosti trenutnega uporabnika za zadnje tri mesece (*ta funkcija je tukaj samo za lažji prikaz delovanja aplikacije*).
  - VHOD: JWT žeton kot 'Authorization: Bearer'
- **POST**: `/gratitude`, ki v bazo vpiše čemu je uporabnik/uporabnica hvaležna v določenem dnevu
  - VHOD: `{ "date", "content": [] }`
  - VHOD: JWT žeton kot 'Authorization: Bearer'
- **GET**: `/gratitude/yyyy-mm`, ki vrne, čemu je bil/bila uporabnik/uporabnica hvaležna v izbranem mesecu
  - VHOD: JWT žeton kot 'Authorization: Bearer'

Tako, to je vse! To je sedem funkcij, s katerimi bomo komunicirali med strežnikom in uporabniškim vmesnikom.

## 2.2. .NET strežniški del

Pri strežniškem delu se bomo najdlje zadržali, saj bo uporabniški vmesnik, spisan v Angular, povsem klasična Angular aplikacija. V Angular aplikaciji ne bo popolnoma nobenih posebnosti in njen strežnik bo vedno naša .NET aplikacija, ki jo bo tudi gostila. Torej uporabniški vmesnik bo povsem enak in neodvisen od tega, ali aplikacija teče kot spletna, ali kot namizna aplikacija. **Ravno to je pravzaprav bistvo tega prispevka!**

> 😎 *V tem delu bomo torej govorili o .NET okolju in za razumevanje slednjega bo potrebno osnovno poznavanje .NET CORE okolja.*

### *2.2.1. Izvajanje programa v različnih operacijskih sistemih*

Ta del nam reši že .NET okolje samo in zanj nam ni potrebno skrbeti. Ko prevedemo .NET program, dobimo na izhodu že podmapo (`/runtimes`) s programskimi knjižnicami za različne operacijske sisteme.

Naša aplikacija, naš strežniški del, bo torej brez naše intervencije tekel v Windows, Linux in Azure okolju. Pravzaprav zna teči tudi še v nekaj drugih operacijskih sistemih, vendar to zaenkrat pustimo.

Ker želimo v .NET razviti strežniški del aplikacije, bomo za osnovo našega projekta izbrali `ASP.NET Core Web API`, ki ustvari natanko to, kar potrebujemo; REST strežnik.

Program se bo torej izvajal v različnih operacijskih sistemih, vendar bomo mi še vedno morali vedeti, kje konkretno se izvaja, da bomo lahko temu primerno – tam, kjer je potrebno, kodo zapisali drugače.

Za potrebe detekcije izvajalnega okolja si torej pripravimo naštevni tip `SupportedEnvironments`, v katerem bomo imeli našteta vsa okolja, ki jih podpiramo… Pravzaprav ne, obstaja boljša rešitev!

> 😎 *Namesto, da imamo po kodi vejitve in se sprašujemo, če je to okolje, potem naredi to, sicer to, je bolje, da naredimo vmesnik `ISuportedEnvironment` z vsemi razlikami, ki jih vsebuje določeno okolje. Na začetku aplikacije ugotovimo izvajalno okolje ter ustvarimo konkreten razred za to okolje, nadalje pa uporabljamo `ISuportedEnvironment` vmesnik. S tem se znebimo vseh vejitev in tudi na enem mestu imamo pregledno zapisane vse razlike med izvajalnimi okolji.*

Torej pripravimo raje vmesnik, ki bo združeval elemente, ki jih mora definirati posamezno izvajalno okolje.

> 😎 *V novejših različicah .NET okolja lahko imajo vmesniki že predefinirane elemente in zato lahko tukaj izberemo vrednosti, ki veljajo v največ primerih, razlike pa bomo definirali v posameznih razredih.*

```
public interface ISupportedEnvironment
{
  string Name { get; }
  bool RequireLogin { get => true; }
  bool OpenBrowser { get => false; }
  void ApplyDeamon(ConfigureHostBuilder configureHostBuilder) { }
}
```

Kot lahko vidite, je razlik v kodi, ki bi bile posledica različnih operacijskih sistemov, zelo malo. Za našo aplikacijo potrebujemo samo tri posebnosti.

Prvi dve sta evidentni že zaradi naših zahtev. V zahtevah piše, da namizne aplikacije ne zahtevajo prijave, spletne pa; to nam torej pove lastnost `RequireLogin`. Ko program teče v Windows okolju kot navaden program, pa je še dodatna zahteva, da se brskalnik avtomatično odpre, za to poskrbi `OpenBrowser`.

Preostali element (`ApplyDeamon`) pa ni takoj evidenten in potrebovali ga bomo kasneje, ko bomo inicializirali aplikacijo.

> 😎 *V kolikor kasneje v razvoju aplikacije naletite na razlike med okolji, jih morate reševati tako, da v tem vmesniku definirate lastnost, na katero se sklicujete, ali pa metodo, ki jo pokličete, da naredi tisto, kar mora na posameznem okolju.*

Kot primer si poglejmo še, kako je definirano okolje, kjer naša aplikacije teče v Linux OS kot storitev (*service, deamon*):

```
public class LinuxAsServiceEnvironment : ISupportedEnvironment
{
  public string Name => nameof(LinuxAsServiceEnvironment);
  public void ApplyDeamon(ConfigureHostBuilder configureHostBuilder)
    => configureHostBuilder.UseSystemd();
}
```

Večina elementov je zajeta že v vmesniku, spremenimo samo ime in aplikaciji povemo, da mora uporabljati `systemd`, ki v Linux sistemih služi za upravljanje s storitvami.

### 2.2.2. Inicializacija aplikacije

Ok, večino posebnosti v aplikaciji smo že rešili, zdaj se bolj ali manj lahko posvetimo sami aplikaciji kot takšni.

Ko ustvarimo prazno .NET WEB App aplikacijo, se kot prva vrstica našega programa izvede sledeče:

```
var builder = WebApplication.CreateBuilder(args);
```

To vrstico moramo popraviti, saj bo naša aplikacija lahko tekla tudi kot storitev in v tistih primerih teče kot sistemski uporabnik, predvsem pa je pomembno, da je njena delovna mapa nek sistemski imenik znotraj

Windows/Linus OS. Mi pa moramo aplikaciji dopovedati, da naj uporablja imenik z aplikacijo kot spletni strežnik, saj bomo znotraj naše aplikacije gostili Angular spletno aplikacijo.

> 😎 *Paziti je potrebno, kaj naša aplikacija vidi kot lastno mapo, oz. mapo s podatki.*

```
var options = new WebApplicationOptions {
  Args = args,
  ContentRootPath = AppContext.BaseDirectory
};

WebApplicationBuilder builder = WebApplication.CreateBuilder(options);
```

### *2.2.3. Detekcija izvajalnega okolja*

Dobro, napisali smo razrede, ki poskrbijo za razlike med izvajalnimi okolji, še vedno pa moramo nekako določiti izvajalno okolje. Napisati moramo torej kodo, ki bo nedvoumno ugotovila, na kakšnem sistemu teče naša aplikacija. Na podlagi tega testa bomo nato ustvarili spremenljivko ustreznega, prej razvitega, razreda:

```
public static ISupportedEnvironment Detect()
{
  if (IsRunningOnAzureWebApp())
    return new AzureEnvironment();

  if (OperatingSystem.IsWindows())
    return WindowsServiceHelpers.IsWindowsService()
      ? new WindowsAsServiceEnvironment()
      : new WindowsAsProgramEnvironment();

  if (OperatingSystem.IsLinux())
    return IsRunningAsLinuxDaemon()
      ? new LinuxAsServiceEnvironment()
      : new LinuxAsProgramEnvironment();

  throw new NotSupportedException("The current environment is not supported.");
}
```

Tole izgleda lepo in pravilno, vendar nam manjkata metodi `IsRunningOnAzureWebApp` in `IsRunningAsLinuxDaemon`. Kot vidite, lahko ostalo določimo že s pomočjo vgrajenih metod .NET okolja. No, resnici na ljubo lahko mogoče ugotovimo tudi še kaj več, vendar trenutno meni to ni uspelo!

Ker torej nimamo vgrajene metode za detekcijo Azure sistema ali za detekcijo, na kakšen način teče naša aplikacija znotraj Linux okolja, si lahko pomagamo s spremenljivkami okolja.

> 😎 *Razvijalci preradi pozabljamo, da nam operacijski sistemi nudijo spremenljivke okolja in da so določene postavljene že s strani OS ali s strani posameznih programov ali pa jih lahko nastavijo uporabniki ter s tem vplivajo na naš program.*

Za detekcijo Azure okolja bomo preverili obstoj dveh spremenljivk, ki jih postavi Azure okolje in za katere predvidevamo, da se drugje ne bodo pojavljale (`WEBSITE_INSTANCE_ID` in `WEBSITE_HOSTNAME`).

Za detekcijo demon-a znotraj Linux okolja pa bomo poskrbeli sami, tako da bomo nastavili spremenljivko (`DOTNET_RUNNING_IN_SERVICE`) v trenutku, ko bomo našo aplikacijo registrirali kot storitev znotraj Linux okolja.

### *2.2.4. Detekcija uporabnika*

Naslednja specifika je zahteva, da se uporabnik na lastnem računalniku ne prijavlja in v tem primeru pač uporabimo uporabniško ime trenutnega uporabnika.

Zaplete pa se tedaj, ko naša aplikacija teče kot storitev v Windows okolju.

V tem primeru imamo pri naši aplikaciji tudi neko specifiko. Seveda bi lahko v takšnem primeru našo aplikacijo klical/uporabil tudi kdo drug v nekem lokalnem omrežju povezanih Windows računalnikov in podobno. Toda

ustavimo konje in si pač razčistimo da je namen naše aplikacije, ki teče kot storitev v Windows okolju pač primer, ko isti računalnik uporabljajo različne osebe (recimo domači računalnik), vendar ne hkrati!

V takšnem primeru je torej potrebno določiti, kdo v resnici sedi za računalnikom oz. kdo se je prijavil, kdo je tisti, ki bo pognal brskalnik.

Slednje lahko ugotovimo s pomočjo vpogleda v sistem ter iskanja, kateri uporabnik poganja program `explorer.exe`.

> 😎 *Še enkrat, to ni »bulletproof« koda, vendar za naše potrebe in tudi potrebe marsikaterega drugega programa je povsem dovolj!*

Konkretno programsko kodo si lahko ogledate v GitHub-u v razredu `UserIdentifierServiceOnWindowsService`.

### 2.2.5. CORS

Zagotovo smo že velikokrat slišali o CORS, pa si zdaj malce podrobneje poglejmo, o čem je govora in kako to vpliva na nas.

CORS (Cross-Origin Resource Sharing) je varnostna funkcija v spletnem razvoju, ki spletni strani omogoča, da zahteva vire iz domene, ki ni tista, iz katere izvira. Če je naša spletna stran na moje-spletisce.si, CORS nadzoruje, ali lahko zahteva pride iz aplikacije na api.nekdo-drug.si. Spletni brskalniki zaradi varnostnih razlogov privzeto blokirajo te zahteve navzkrižnega izvora, vendar CORS strežniku omogoča, da določi, katerim domenam je dovoljen dostop do njegovih virov.

Ker bomo v času našega razvoja razvijali strežniški del aplikacije ločeno od uporabniškega vmesnika in bomo testirali strežniški del, ki bo tekel na npr. `localhost:1770` in uporabniški vmesnik, ki bo tekel na `localhost:1400`, moramo uporabiti CORS, da lahko uporabniški vmesnik kliče API zahteve strežniškega dela, ki teče na drugih vratih.

Kako izgleda vključitev CORS, si lahko ogledate na GitHub-u v razredu `program.cs`.

> 😎 *V fazi razvoja je CORS super in ga je dobro vklopiti, saj je tako lažje testirati. V produkciji pa ga je dobro izklopiti oz. ga omejiti.*

### 2.2.6. Usmerjanje (routing)

Naš uporabniški vmesnik bo tekel kot *enostranska aplikacija* (SPA – Single Page App) in kot vemo, slednje vse zahteve vodijo preko glavnega obrazca, torej preko `index.html`.

Če torej Angular neke podstrani ne bo našel, bo poskusil preko `index.html`. Ker bomo Angular aplikacijo gostili mi znotraj našega strežniškega dela, ji moramo zagotoviti to mehko pristajanje na `index.html` strani:

```
app.MapFallbackToFile("index.html");
```

> 😎 *Če vaša aplikacija gosti enostransko spletno aplikacijo (SPA), ne pozabite na obravnavo index.html datoteke.*

### 2.2.7. Preostali del strežniške aplikacije

V tem trenutku se lahko skupaj nasmehnemo, saj je za »preostali del aplikacije« namenjeno eno samo poglavje.

Seveda je jasno, da smo do sedaj pogledali samo pet odstotkov potrebne kode našega strežniškega dela, vendar pa ta prispevek ni namenjen prikazu razvoja REST API strežnika v .NET!

> 😎 *Celotna preostala aplikacija je povsem klasična .NET aplikacija in nima nobenih posebnosti, pač prevzem zahtev, komunikacija s podatkovno bazo in vrnitev rezultatov.*

Kot podatkovno bazo uporabljam kar `SQLite`, za lažji dostop pa mikro ORM, `Dapper`. Seveda lahko aplikacijo nadgradite z uporabo kakšnega `MSSql` strežnika, uporabo `Entity Framework` in podobnim, vendar kot rečeno, moja želja je bila, da je aplikacija majhna, obvladljiva, vseeno pa popolna.

> 😎 *V kolikor boste torej gledali programsko kodo, boste v njej videli tudi recimo lokalizacijo ali pa delo z JWT žetoni.*

### 2.3. Angular uporabniški vmesnik

Pri strežniškem delu sem predstavil posebnosti, ki nam bodo omogočile, da bo naša aplikacija brez sprememb ter samo s »kopiraj in prilepi« akcijo tekla v različnih okoljih.

S tem je imel strežniški del nekaj podpoglavij, pri uporabniškem vmesniku pa bo besedila še mnogo manj, saj tu ni absolutno nobenih posebnosti.

> 😎 *Uporabniški del v Angularju nima nobene zveze s sistemom v katerem teče. Vse, kar njega zanima je API, ki mu ga strežnik nudi in seveda, kje se strežnik nahaja; naslov strežnika torej. Ko se bo Angular povezal s strežnikom, bo tekel kot vsaka Angular/Vue/React/... aplikacija.*

Resnično gre za povsem navadno SPA aplikacijo, ki bi lahko bila razvita v kateremkoli JavaScript ogrodju in Angular sem izbral samo zato, da ne govorim vedno o VUE, ki sem ga do sedaj predstavljal [5].

#### *2.3.1. Kje je moj strežnik?*

Pri razvoju Angular aplikacije ni nobenih posebnosti, bi pa predstavil nekaj, kar lahko morda komu olajša razvoj.

Kot že omenjeno ob razlagi CORS, imamo v razvoju situacijo, ko nam strežniški del teče na npr. `localhost:1770`, naša Angular aplikacija pa teče na klasičnih Angular vratih `localhost:4200`. Kako torej dopovedati Angular-ju, naj tvori naslove, kot so `https://localhost:1770/user/login`, če pa sam teče na vratih `4200`?

Uporabili bomo neko JSON nastavitveno datoteko in notri napisali naslov strežnika in potem kar uporabljali ta naslov. Ok, do sem vse lepo in prav. Kaj pa, ko bo aplikacija tekla na produkciji?

Takrat pa mora naša aplikacija teči na istem naslovu! Poznamo torej naslov strežnika, saj je to isti naslov, na katerem teče Anguler sam. V tem primeru je torej nesmiselno še enkrat pisati ta isti naslov v namestitveno datoteko.

> 😎 *Lahko se znajdemo in rečemo: v kolikor namestitvena datoteka obstaja, potem uporabi podatek iz namestitvene datoteke, sicer uporabi lasten naslov.*

Torej, prebrati poskusimo nastavitve iz `/assets/config.json` in če nam to ne uspe, privzamemo naš naslov kot naslov strežnika:

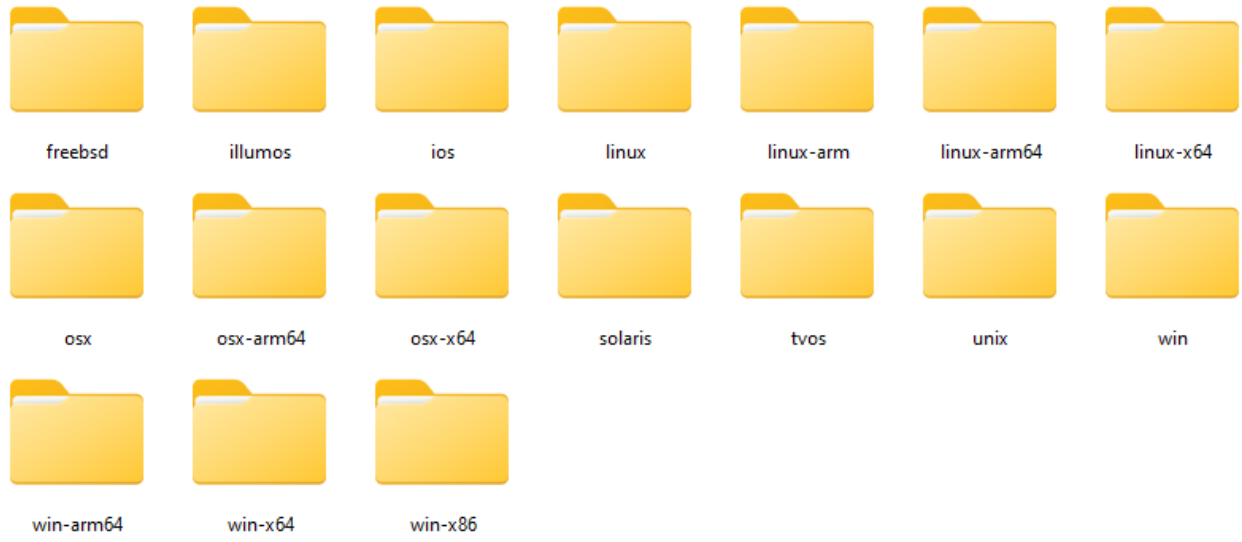
```
loadConfig(): Observable<any> {
  console.log('Loading configuration');
  return this.http.get('/assets/config.json').pipe(
    catchError(error => {
      return new Observable(observer => {
        observer.next({ apiUrl: window.location.origin });
        observer.complete();
      });
    })
  );
}
```

Ob izgradnji programa za produkcijo preprosto pobrišemo to namestitveno datoteko in vse deluje tako, kot mora.

## 2.4. Dajmo vse skupaj!

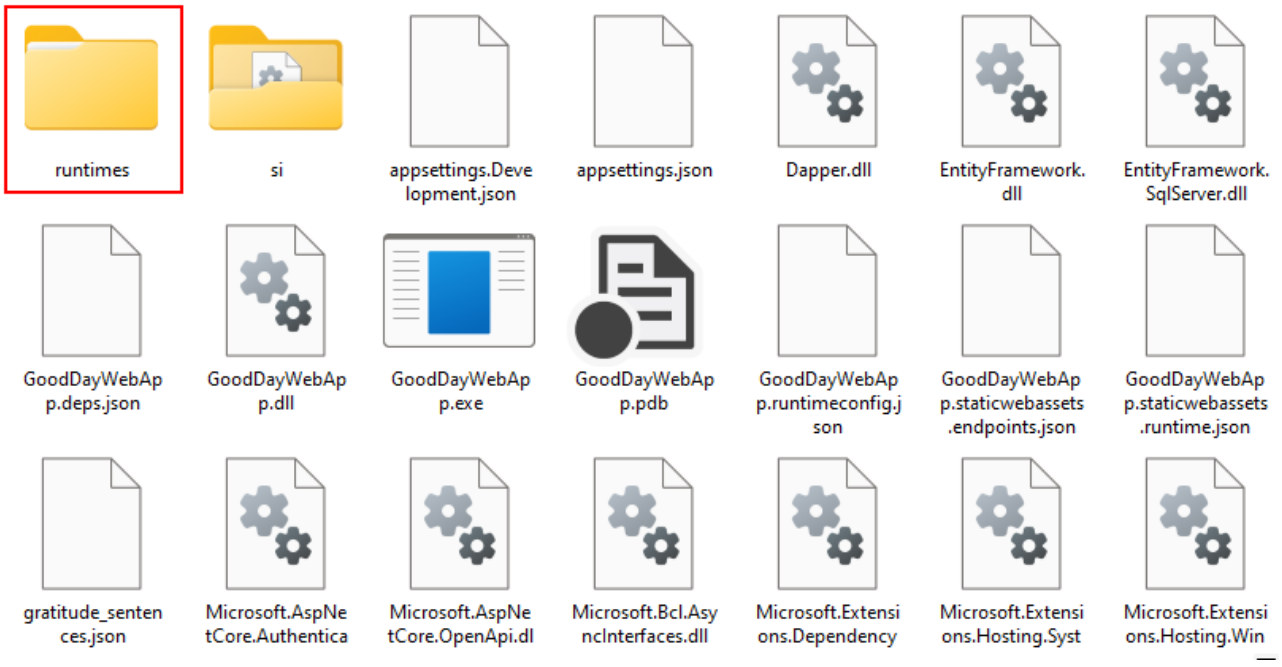
Ena izmed bolj znanih programerskih izjav »Na mojem računalniku deluje!« je posledica kompleksnosti izvajalnih okolij in izgradnje modernih programskih rešitev.

Vendar v našem primeru na srečo temu ni tako. Ko zgradimo naš strežniški del v .NET okolju, dobimo mapo, v kateri se nahaja naš program, predvsem pa se v tej mapi nahajajo tudi potrebne knjižnice (v podmapi `/runtimes`), da zadeve teče tudi na ostalih (podprtih) operacijskih sistemih




**Slika 1: Podprti operacijski sistemi, v katerih se lahko izvaja .NET 8.0 aplikacija.**

Če torej celotno izhodno mapo, v kateri je tudi podmapa `/runtimes`, kopiramo na ciljni računalnik, mora zadeva delovati.




**Slika 2: Izhodna mapa našega strežniškega programa s podmapo /runtimes.**

#### *2.4.1. Dobro, s tem smo rešili strežniški del, kaj pa naj naredimo z Angular aplikacijo?*

Najprej moramo Angular aplikacijo zgraditi. To naredimo s sledečim ukazom (ali kakšno njegovo izpeljanko):

```
ng build
```

V kolikor nimamo napak v kodi, se bo naša SPA aplikacija zgradila v podmapi `/dist`. Toda na žalost to še ni podmapa, ki nas zanima, saj je spodaj še nekaj podmap `/good-day-spa/browser`. Skupaj torej `dist/good-day-spa/browser`. Ime vmesne mape (`good-day-spa`) je seveda odvisno on nastavitev.

No, nas pa tako ali tako zanima samo tisto, kar je v podmapi `/browser` in to moramo kopirati v podmapo wwwroot našega strežniškega dela. To je vse 🙌.

> 😎 *Da bi naša aplikacija delovala kot celota, je potrebno prevedene Angular datoteke kopirati v podmapo /wwwroot naše strežniške aplikacije.*

Tole je izsek iz skripte, ki zgradi našo aplikacijo v celoti:

```
@REM Zgradimo .NET aplikacijo
@dotnet publish GoodDayWebApp.csproj  -c Release -f net8.0

@REM Rezultat našega prevoda bo v tejle podmapi
@SET BACKEND_SOURCE=GoodDayWebApp\bin\Release\net8.0\publish

@REM Zgradimo Angular SPA aplikacijo
@cmd /c ng build --configuration production

@REM Angular aplikacija bo v tej podmapi
@SET FRONTEND_SOURCE=GoodDaySPA\dist\good-day-spa

@REM Kopiramo Angular aplikacijo v wwwroot podmapo naše strežniške aplikacije
@xcopy %FRONTEND_SOURCE%\browser %BACKEND_SOURCE%\wwwroot /s /e /y
```

# 3 V produkciji

Pod predpostavko, da smo aplikacijo zgradili tako, kot je opisano v prejšnjem odstavku, torej da smo Angular preveden program prepisali v podmapo `wwwroot` naše .NET aplikacije, si poglejmo, kako se zadeva obnaša na posameznih okoljih.

## 3.1. Izvajanje aplikacije na drugem računalniku

Spet smo pred famozno razvijalsko izjavo: »Na mojem računalniku deluje!«. Kaj je torej potrebno narediti, da bo naša aplikacija tekla tudi na drugih računalnikih?

Dandanes to ni nobena posebnost več, pa vendar dajmo to zapisati še enkrat. Mnoga razvojna okolja, in med njimi je seveda tudi .NET, poznajo dve vrsti instalacije:

- **Razvojna instalacija** (SDK Install == Software Development Kit Instalation) je večja instalacija in s to instalacijo lahko na računalniku razvijamo programsko opremo in posledično jo lahko seveda tudi poganjamo.
- **Izvajalna instalacija** (Runtime instalation) pa uporabniku omogoča samo izvajanje že razvitih aplikacij in je posledično seveda manjša oz. zasede manj prostora na disku.

😎 *Da bi torej naš program tekel na kateremkoli računalniku, tam torej potrebuje izvajalno okolje, zato si mora uporabnik najprej instalirati izvajalno okolje.*

To je danes res preprosto in zahteva samo skok na splet in zagon instalacijske datoteke na poljubnem operacijskem sistemu

#### *3.1.1. Priprava aplikacije 'Vse V Enem'*

Lahko pa naš program prevedemo tudi tako, da v celoti vsebuje vso potrebno izvajalno okolje že v sebi. S tem dobimo nekaj pozitivnih in nekaj negativnih lastnosti:

- (+) Uporabnik ne potrebuje nobene posebne instalacije. Naloži naš program in vse deluje.
- (-) Ker naš program vsebuje tudi vso potrebno izvajalno okolje, je precej večji in zahteva več prostora.
- (-) Če ima uporabnik več programov in vsak prinese zraven izvajalno okolje, potem ima uporabnik na računalniku mnogo kopij izvajalnih okolij.

Odločitev pa je vedno na vas, razvijalcih, po kateri poti boste šli.

*Naši aplikaciji ne bomo prilagali izvajalnega okolja in bomo predvidevali, da slednje na računalniku obstaja.*

### 3.2. V okolju Windows kot program

Izvedemo EXE program brez kakršnihkoli sprememb in

- Program se zažene v ukazni vrstici
- Ker ga poganjamo v okolju Windows, se avtomatično zažene brskalnik in takoj smo v aplikaciji, ker prijava ni potrebna

### 3.3. V okolju Windows kot storitev

Naša aplikacije je v celoti pripravljena tako, da lahko teče kot storitev, vendar pa jo moramo kot storitev seveda najprej prijaviti.

V GitHub repozitoriju je pripravljenih nekaj kratkih skript, s pomočjo katerih lahko aplikacijo registriramo kot storitev, jo zaženemo, ustavimo, odstranimo… Ni pa to nič kompleksnega in tukaj prilagam primer skripte, ki aplikacijo registrira kot storitev, ter jo tudi požene:

```
# registracija

New-Service -Name GoodDayService `
            -BinaryPathName "GoodDayWebApp.exe --contentRoot GoodDayWebApp.exe" `
            -Description "Aplikacija dobrega počutja" `
            -DisplayName "GoodDayService"

# zagon

Start-Service -Name GoodDayService
```

S pomočjo takšne `PowerShell` skripte lahko našo aplikacijo, brez sprememb(!), poganjamo kot storitev v okolju Windows.

Ker sedaj aplikacija teče kot storitev, je vedno dosegljiva, zažene se ob zagonu oken in vsak uporabnik lahko odpre brskalnik, zažene ustrezen spletni naslov `http://localhost:5000`[24] in že lahko uporablja aplikacijo brez prijave.

Ko se bo na isti Windows računalnik prijavil drug uporabnik, bo aplikacijo brez prijave uporabljal z lastnim uporabniškim imenom in obiskom spletne strani `http://localhost:5000`.

### 3.4. V okolju Linux kot program

Podobno kot v okolju Windows, tudi tukaj program preprosto poženemo. Seveda pa Linux ne uporablja EXE datotek, zato moramo program zagnati kot `dotnet` program, torej:

```
/usr/bin/dotnet GoodDayWebApp.dll
```

In to je vse. Uporabnik lahko aplikacijo uporablja in ne potrebuje prijave, saj se prijava privzame iz uporabniškega računa. Mora pa ročno zagnati brskalnik ter odpreti spletno stran.

### 3.5. V okolju Linux kot storitev (deamon)

Tukaj je zadeva enakovredna Windows okolju, saj naš program ne potrebuje nobenih sprememb. Le zagnati ga moramo kot Linux deamon program.

Tudi tukaj si je najlažje pomagati s skriptami in tudi tukaj so skripte zelo enostavne, zato prilagam skripto za registracijo in zagon storitve v `BASH` izvajalnem okolju (kot vedno, so vse skripte v GitHub repozitoriju).

Za razliko od Windows okolja moramo v primeru Linux okolja najprej pripraviti datoteko, ki jo bo Linux razumel kot definicijo storitve »deamon«-a. Ustvarimo torej datoteko GoodDayWebApp.service, ki jo bomo potem morali skopirati v ustrezno mapo in Linux bo to razumel kot navodilo za izvajanje storitve:

```
[Unit]
Description=GoodDayService

[Service]
WorkingDirectory=/mnt/c/linux-dotnet
ExecStart=/usr/bin/dotnet GoodDayWebApp.dll
Restart=always
RestartSec=10
KillSignal=SIGINT
SyslogIdentifier=GoodDayWebApp
Environment=ASPNETCORE_ENVIRONMENT=Production
Environment=DOTNET_RUNNING_IN_SERVICE=1

[Install]
WantedBy=multi-user.target
```

Tukaj ne bom opisoval pomena in namena posameznih elementov te datoteke. Je pa bolj ali manj že iz angleških nazivov jasno, čemu je kateri del namenjen.

> 😎 *Opozoril bi samo na majhen detajl in sicer na vrstico 'Environment = DOTNET_RUNNING_IN_SERVICE=1'. S to vrstico postavimo spremenljivko okolja, ki je za nas zelo pomembna, saj z njeno pomočjo ugotovimo, da naša aplikacija teče kot Linux storitev!*

Dobro, sedaj smo Linux storitev definirali in moramo jo še registrirati ter zagnati:

```
#!/bin/bash
cp GoodDayWebApp.service /etc/systemd/system
```

[24] Vrata 5000 so privzeta in jih lahko z nastavitvami seveda spremenimo!

```
systemctl daemon-reload
systemctl start GoodDayWebApp
```

Definicijo storitve najprej kopiramo v ustrezno mapo, nato ponovno zaženemo upravitelja storitev, da slednji zazna našo novo storitev in na koncu aplikacijo (storitev, deamon) še zaženemo.

To je to, s tem bodo našo aplikacijo lahko uporabljali VSI, ki imajo dostop do tega Linux računalnika. Morali pa bodo ročno zagnati brskalnik ter se v aplikacijo prijaviti, saj teče kot večuporabniška aplikacija.

### 3.6. V okolju Azure

In seveda v letu 2024 ne moremo mimo tega, da bi ne omenili računalništva v oblaku, v našem primeru torej Azure.

> 😎 *Čeravno smo torej v letu 2024, mi je uspelo spisati celoten prispevek brez ene same omembe kratice, ki je v angleščini sestavljena iz dveh črk oziroma natančneje dveh samoglasnikov, ki ju lahko slišite, če poslušate osla… Pa s tem seveda ne mislim, da je vso to noro navdušenje nad to tehnologijo nesmiselno ali za osle; nikakor ne, včasih pa je vseeno dobro, če se ji izognemo in je ne omenjamo vedno v povezavi z računalništvom.*

Da bi nam karkoli delovalo v Azure, moramo najprej malce razvezati denarnico in dobiti dostop do teh oblakov.

Ko dostop imamo, pa je zadeva dokaj enostavna in razumljiva. Narediti moramo novo 'Web App' storitev, izbrati .NET 8.0, izvajalno okolje. Operacijski sistem pa je nepomemben, jaz sem izbral Linux in ne Windows, ker sem ga pač!

Ko v Azure definiramo novo 'Web App' in jo Azure inicializira, lahko do nje dostopimo preko FTP oz. preko SFTP (secure FTP).

Tam nas že čaka mapa wwwroot in vse, kar je potrebno storiti ,je našo celotno aplikacijo skopirati v mapo wwwroot naše Azure Web App storitve. Na Azure plošči pogledamo še naslov naše storitve in takoj jo lahko začnemo uporabljati, tako da preprosto odpremo ta naslov v poljubnem brskalniku kjerkoli na svetu!

> 😎 *Da bi naša aplikacija delovala v Azure, jo moramo samo v celoti kopirati v wwwroot podmapo!*
>
> 😎 *Da, res je, se tem imamo dve wwwroot mapi. Prva (glavna, višja) je Azurjeva in preko nje Azure izvaja našo aplikacijo! Naša aplikacija pa ima spodaj še eno wwwroot mapo in preko nje naša aplikacija izvaja Angular aplikacijo. Vau* 😱*!*

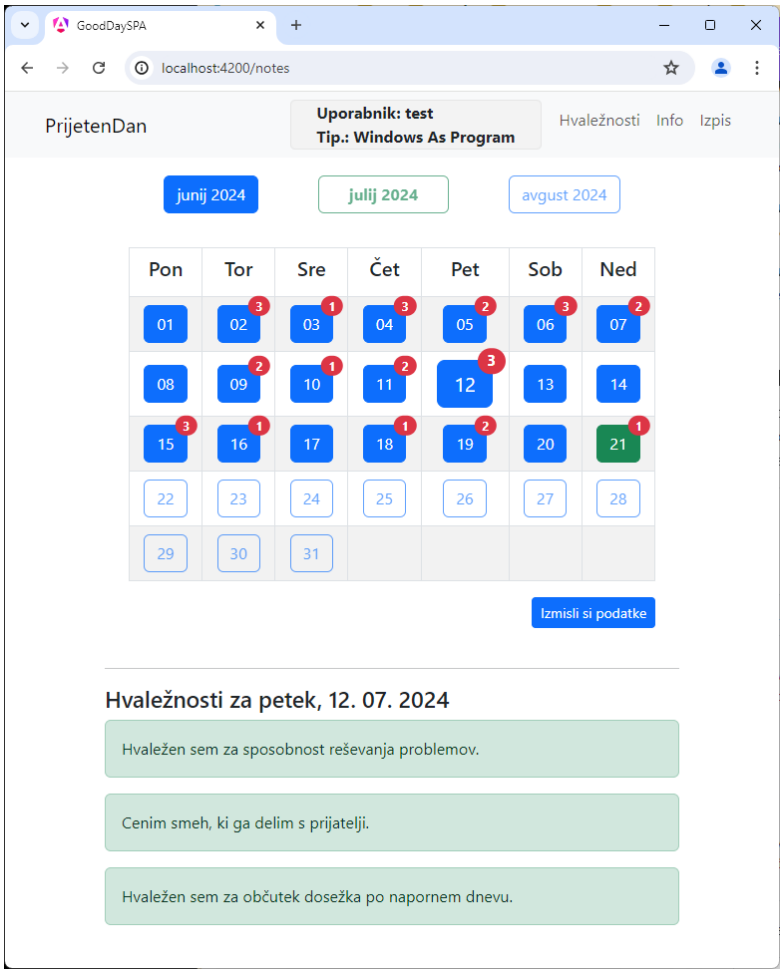
### 3.7. Slabosti

Ker je naša želja, da se aplikacija samo kopira drugam in tam kar »magično« deluje (*»Magično« bi delovala recimo tudi na Mac OS, vendar slednjega nimam in nisem poiskušal*), to pomeni, da zraven v `/runtimes` mapi vlečemo tudi izvajalne module za vse mogoče operacijske sisteme. V resnici je smiselno pobrisati zadeve, ki jih na določenem okolju ne potrebujemo.

Seveda ima sama aplikacija varnostne slabosti, slab uporabniški vmesnik in podobno. A izdelava popolne aplikacije nikoli niti ni bil namen tega prispevka!

# 4 Uporabniški vmesnik

Temu bi se res želel izogniti, kajti uporabniški vmesnik je pač nekaj, za kar razvijalec mora imeti žilico in čas. Jaz nimam ne enega ne drugega, vendar pa je vseeno dobro, da v prispevku vsaj na sliki prikažem, o kakšni aplikaciji sploh govorimo oz. kako zadeva izgleda.




**Slika 3: Glavni ekran aplikacije.**

Na sliki lahko vidite, da trenutno aplikacija teče kot program v okolju Windows in da jo uporablja uporabnik 'test'.

# 5 Zaključek

Namen prispevka je prikazati:

- Da je vsaka vaša .NET Web Api aplikacija pravzaprav spletni strežnik,
- Ker je spletni strežnik, lahko znotraj nje gostite spletne strani,
- Ker lahko gostite spletne strani, lahko gostite tudi SPA aplikacije,
- Ker lahko gostite SPA aplikacije, lahko imate vse v enem, tako REST strežnik kot uporabniški vmesnik,
- In, ker vse teče v odprtokodnem .NET okolju, posledično lahko takšna aplikacija teče v mnogih operacijskih sistemih.

## Literatura

[1] https://dotnet.microsoft.com

[2] https://angular.dev/

[3] OTS 2022, Sodobne informacijske tehnologije in storitve: Zbornik 25. konference, »Kaj je Blazor in kako se primerja z JavaScript ogrodji?«, Matjaž Prtenjak

[4] OTS 2023, Sodobne informacijske tehnologije in storitve, Zbornik 26. konference, »Kaj je Blazor Hibrid in kako nam lahko pomaga tudi pri nadgradnji programske opreme?«, Matjaž Prtenjak

[5] OTS 2022, Sodobne informacijske tehnologije in storitve: Zbornik 24. konference, »Kako razvijati v VUE/NUXT, če si navajen/a objektnih jezikov kot so C++, C# ali Java?«, Matjaž Prtenjak

Programska koda celotne aplikacije: https://github.com/MPrtenjak/OTS2024-NET-Angular-WEB-And-Desktop