Kaj je Blazor in kako se primerja z JavaScript ogrodji?

Matjaž Prtenjak
Matjaž Prtenjak s.p., Laško, Slovenija
matjaz@mnet.si

Sinopsis Nekako smo se že navadili, da se spletne aplikacije razvijajo v programskem jeziku JavaScript. Tukaj govorim o programski kodi, ki teče v brskalniku, kajti sodobna spletna aplikacija potrebuje tudi zaledje, podatkovne baze in podobno. Vsa ta zaledna koda je pisana v različnih programskih jezikih. Koda, ki teče v brskalniku in torej uporabnik z njo neposredno kontaktira, pa je v večini primerov klasična JavaScript koda.
Kaj pa, če bi želeli izdelati spletno aplikacijo v kakšnem drugem programskem jeziku? Kaj pa, če sicer uporabljate Microsoftova razvojna okolja in bi želeli spletno aplikacijo, ki teče v poljubnem brskalniku, napisati v programskem jeziku C#?

Leta 2018 je bil sprejet standard poimenovan WebAssembly [1], ki ga podpirajo vsi novejši brskalniki. Kot že ime implicira gre za neke vrste strojni jezik (assembly) brskalnika. In če naredimo prevajalnik za ta strojni jezik, se potem lahko ta koda izvaja v brskalniku.
To je Blazor [2] in to je tematika tega prispevka.

Ključne besede:
Blazor
WASM
.NET
spletne aplikacije
JavaScript
VUE
C#

Univerzitetna založba Univerze v Mariboru
ISBN 978-961-286-639-6
DOI https://doi.org/10.18690/um.feri.10.2022.7

1 Uvod

Leta 2018 je bila sprejeta prva verzija standarda WebAssembly [1], ki so ga podprli vsi izdelovalci brskalnikov.

WebAssembly (WASM) je binarni format navodil za virtualni stroj. WASM je zasnovan tako, da omogoča izvajanje programske kode (v peskovniku) tako na odjemalcih, kot tudi na strežnikih.

Microsoft je vzpon JavaScript-a zamudil. Pred leti je poskušal razviti okolje, ki bi bilo bližje povprečnemu MS razvijalcu/razvijalki. Naredili so Silverlight [3], ki pa je v brskalniku potreboval vtičnik in ni nikoli zares zaživel.

Hkrati pa MS ima močno orožje, ki nenehno tekmuje z Java svetom in to je seveda .NET, ki ga je v zadnjih letih povsem prenovil in tudi odprl svetu, saj je ustanovil .NET Foundation [4] in je zatorej odprtokodno orodje.

Vendar to ni tematika tega prispevka. To je samo osnova, ki nam pomaga razumeti, kaj se je zgodilo, ko sta se združili ideji odprtega .NET razvojnega okolja in odprtega WASM. MS je dobil možnost, da veliko količino programske kode prevede v WASM ali povedano drugače, dobil je možnost razvoja .NET programov, ki tečejo v brskalniku.

Blazor je odprtokodno razvojno okolje podjetja Microsoft, kjer lahko program razvijamo v C# programskem jeziku, prevajalnik pa ga prevede v WASM. Takšen program se torej lahko izvaja v poljubnem brskalniku (vsi novejši).

2 Blazor

2.1 Kratka zgodovina

Definicija imena Blazor, ki je zapisana v prejšnjem odstavku, nam ne pove kaj dosti, zato si na kratko poglejmo razvojno pot Blazor-ja in posledično razvojno pot podjetja Microsoft v spletne aplikacije.

Podobno kot siceršnji razvoj, lahko tudi spletni razvoj razdelimo na dva dela. Na strežniški del (angl. back-end) in na uporabniški del (angl. front-end).

2.1.1 Strežniški del

Seveda se je najprej razvijal strežniški dal, saj moramo najprej nekaj imeti, da to lahko pokažemo uporabniku. Uporabnik oz. brskalnik je torej na začetku pridobival statične HTML strani, ki pa so kmalu postale dinamične v smislu, da jih je strežnik pred-obdelal, preden jih je poslal uporabniku.

Tipičen primer je recimo spletna stran za prikaz detajla artikla. Če imamo 100 artiklov, ni potrebno napisati 100 HTML strani, temveč napišemo eno HTML stran, ki ima dinamične komponente in preden brskalnik dobi takšno stran, strežnik vanjo vrne podatke konkretnega artikla. Programska jezika PHP in RUBY sta bila med pionirji takšnega načina izdelave spletnih strani.

Ta vlak je MS malce zamudil, toda navkljub vsemu je dokaj hitro izdal programski jezik ASP (Active Server Pages). Njegova težava nikoli ni bila neuporabnost ali zapletenost ali kaj podobnega, temveč precej bolj banalna - denar. Vse ostale rešitve so lahko tekle na zastonjskih (ali pa zelo poceni) Linux strežnikih, za ASP (in kasneje ASP.NET) rešitve, pa je bilo potrebno v ozadju imeti MS infrastrukturo, ki pa je seveda plačljiva.

Kakorkoli, MS je imel rešen strežniški del, manjkalo pa jim je orodje na uporabniški strani.

2.1.2 Uporabniški del

Prvi resni vstop v svet uporabnikov je MS naredil leta 2007, ko je izdal orodje Silverlight. Tedaj je na brskalnikih kraljeval Adobe Flash [5] in Silverlight je bil Microsoftov poskus vstopa v ta segment. Oba sta bila vtičnika in oba je bilo potrebno najprej instalirati v brskalnik. V nekem trenutku so vtičniki postali nekaj slabega (upravičeno ali ne) in uporabniki jih niso več želeli, zato so se poslovili.

Javascript se je pojavil seveda še pred Flash-om in je bil vseskozi prisoten, največja težava pa je bila v neusklajenosti brskalnikov pri podpori JavaScript. Tako so/smo morali razvijalci razvijati različne funkcije za različne brskalnike. Toda, ker to ni prispevek o JavaScript, bom ta del preskočil in preprosto skočil v čas po Flash-u in po JQuery knjižnici [6].

JQuery je sicer knjižnica, ki je prva poenotila način delovanja JavaScript v različnih brskalnikih. Razvijalci smo imeli možnost razvoja JavaScript programske kode, ki je v 95% primerov enako tekla na vseh brskalnikih.

Spet je MS ostal zadaj in spet je zamudil vlak. Na njegovo srečo pa so se v WWW konzorciju uspeli dogovoriti in vzpostaviti standard WebAssembly, ki predstavlja neke vrste strojni jezik brskalnika. Nek program lahko torej prevedemo v WebAssembly (popularno imenovan WASM) in tekel bo v vseh novejših brskalnikih (torej v brskalnikih izdanih v zadnjih nekaj letih).

Po drugi strani pa je Microsoft začel odpirati njegovo paradno okolje .NET in naredil .NET Core. Ker to spet ni prispevek o MS, bom preskočil tudi ta del in preprosto prišel na ključni del:

1. WWW konzorcij je standardiziral WebAssembly.
2. MS je odprl .NET ter naredil .NET Core, ki ni več vezan na MS, temveč odprtokoden.
3. Točki 1 in 2 sta omogočili, da je MS prevedel .NET Core knjižnice v WebAssembly in sedaj lahko vsaka koda, ki sicer teče v .NET Core teče tudi na brskalniku, saj se brskalnik obnaša kot operacijski sistem z .NET knjižnicami.

Torej sedaj imamo produkt, ki lahko .NET program prevede v WebAssembly in stvar teče v brskalniku. In to je Blazor.

Pravzaprav je to del Blazor-ja. Blazor se namreč ravno tako deli na strežniški in uporabniški del. Medsebojno sta neodvisna in seveda lahko sodelujeta. Pomembno pa je, da lahko uporabniški del Blazor aplikacije teče povsem neodvisno od strežniškega dela.

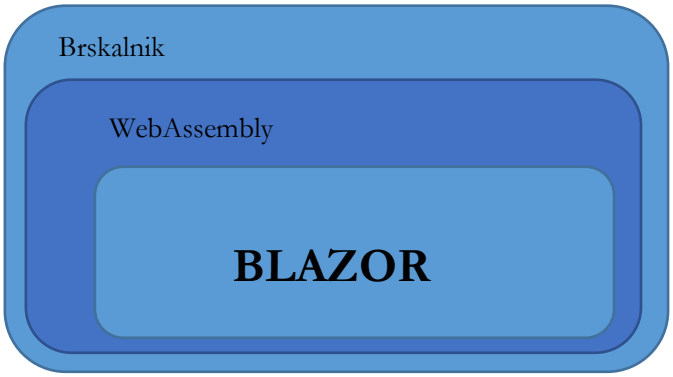
To slednje pa pomeni, da za delovanje ne potrebujemo MS infrastrukture!

In to je bistvo tega prispevka. V njem bom predstavil uporabniški del Blazor aplikacije, ki teče v WebAssembly-u in ne potrebuje MS infrastrukture, temveč v celoti teče v brskalniku, aplikacijo pa lahko streže poljuben strežnik - recimo Apache.

2.2 Dva tipa Blazor aplikacij

2.2.1 Blazor Web Assembly

Ta v celoti teče na brskalniku in to je tudi model, ki ga bom opisoval.



Slika 1: Blazor WebAssembly.
Vir: lasten.

Prednosti:

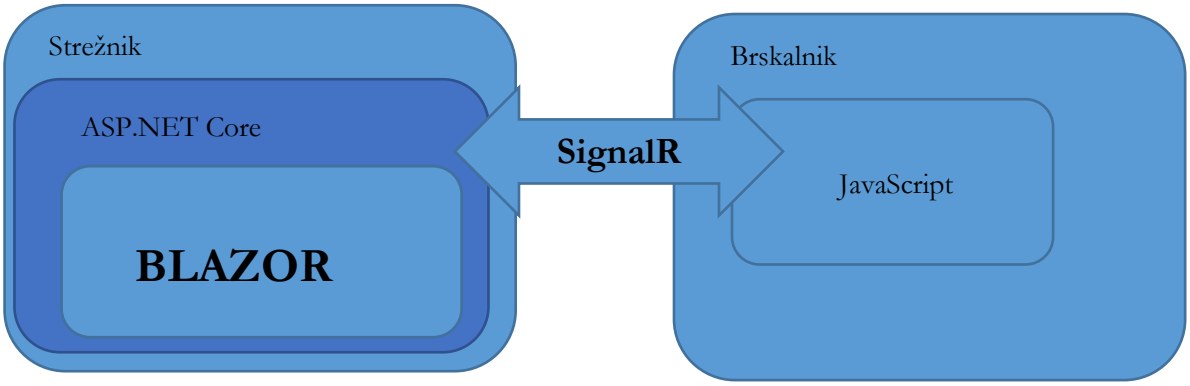
- Teče v brskalniku, neodvisno od strežnika (razen seveda, da potrebujete spletni strežnik za osnovni zagon).
- Lahko pa razvijete tudi t.i. PWA (Progressive Web App), ki pa je nisem preizkusil.

Slabosti:

- Osnovna slabost tega načina je daljši zagonski čas, saj mora strežnik najprej na uporabnika poslati datoteko blazor.webassembly.js, ki potem posledično s strežnika pretoči vse potrebne .DLL datoteke, kar lahko traja dlje časa, v kolikor je povezava slabša.

2.2.2 Blazor Server Side

Tega nisem preizkušal in tukaj opisujem samo v toliko, da podam celotno sliko o Blazor tehnologiji. Razlog neukvarjanja s tovrstno rešitvijo pa je zopet v dejstvu, da zanjo potrebuješ MS infrastrukturo, medtem ko mene zanimajo odprtokodne rešitve, ki lahko tečejo na različnih spletnih strežnikih.



Slika 2: Blazor Server Side.
Vir: lasten.

Prednosti:

- Ni dolgega zagonskega časa.
- Na uporabnika se prenaša samo HTML in JavaScript.
- Ker ni potrebe po WASM, lahko rešitev deluje tudi v starejših brskalnikih.

Slabosti:

- Aplikacija za komunikacijo med uporabniki in strežnikom uporablja SignalR, kar pomeni, da se mora strežnik zavedati VSEH trenutnih uporabnikov in to lahko povzroči veliko porabo sistemskih sredstev na strežniku.
- Zelo veliko komunikacije med uporabnikom in strežnikom, saj mora uporabnik vsak dogodek (preko SignalR) poslati na strežnik, ki ga (v .NET okolju) obdela in rezultat (delni HTML/CSS) pošlje nazaj uporabniku.
- Potrebujemo MS strežniško infrastrukturo.

2.3 Kako razviti Blazor WASM aplikacijo?

Najprej moramo seveda instalirati .NET Core razvojna orodja, ki so brezplačno dosegljiva na MS spletnih straneh.

Nadalje moramo izbrati nek urejevalnik, ki je seveda lahko poljuben, čeravno se s .NET (seveda!) najbolje razumeta MS urejevalnika Visual Studio Code (ki je brezplačen) in Visual Studio (ki obstaja v več različicah, tudi brezplačni).

Ko torej imamo razvojna orodja in urejevalnik:

- Odpremo nov BLAZOR WASM projekt.
- Izdelamo program.
- Ga prevedemo.
- Prepišemo na spletni strežnik.
- Ko spletna aplikacija teče, spletni strežnik v celotnem procesu deluje zelo malo, saj gre za statično podajanje spletnih strani in se aplikacija prenese na brskalnik, kjer se tudi izvaja.

Ker ta prispevek nikakor ni namenjen učenju programiranja ali čemu podobnemu, postopka ne bom nadalje in podrobneje opisoval, saj so postopki lepo opisani na mnogih spletnih straneh. Najlažje pa je začeti recimo na spletni strani: https://blazor-university.com/.

2.4 Primerjava med JavaScript knjižnico VUE in Blazor WASM

VUE je JavaScript knjižnica in lahko jo vključite v vsako HTML stran ter na njej uporabite poljubne VUE konstrukte. Torej brez kakršnekoli instalacije, brez prevajanja, brez "aplikacij" in podobnega. Preprosta JavaScript koda, ki jo vključite v HTML in uporabite.

Blazor WASM tega ne omogoča. V tem primeru imate vedno aplikacijo, ki jo je potrebno razviti, prevesti, prepisati na strežnik in uporabiti.

Primerjamo lahko torej samo VUE CLI aplikacije in Blazor WASM aplikacije. VUE CLI aplikacija je standardna VUE aplikacija, ki je v splošnem sestavljena iz več datotek izvorne kode in zato se najprej prevede v standardno JavaScript kodo in nato prepiše na spletni strežnik.

Kot že rečeno, to ni tečaj programiranja, zato bom razlike med okoljema prikazal tako, da bom vzel par primerov iz VUE tečaja (Intruduction), na VUE spletni strani [7].

V razlagi bom uporabljal besedo RAZOR, ki je zelo podobna besedi BLAZOR in to namenoma. RAZOR je namreč MS programski jezik za »nadzor« HTML kode in BLAZOR za obdelavo HTML uporablja RAZOR. Od tod v resnici tudi izhaja njegovo ime. RAZOR je namreč MS že imel in potem si je nad njim pač izmislil BLAZOR.

2.4.1 Ustvarjanje aplikacije

VUE (potrebujemo node)	*Blazor (potrebujemo .NET CORE)*

`npm install -g @VUE/cli`
`VUE create pozdrav`
`cd pozdrav`
`npm run serve`
	`dotnet new blazorwasm -n pozdrav`
	`cd pozdrav`
	`dotnet run`

Kot je vidno tukaj, razlike ni. Oba ukaza pripravita vse potrebno za razvoj aplikacije v VUE oz. Blazor okolju.

2.4.2 Prikaz podatkov

VUE	*Blazor*

`<template>`
`  <div>`
`  {{ message }}`
`  </div>`
`</template>`
 	
`<script>`
`  export default {`
`  data () {`
`  return {`
`  message: 'Hello VUE!'`
`  }`
`  }`
`  }`
`</script>`
	`<div>`
	`  @message`
	`</div>`
	
	`@code {`
	`  private string message = "Hello Blazor!";`
	`}`

Kot je vidno iz primera je programska koda pri Blazor aplikaciji manjša, predvsem pa je pomembno, da BLAZOR uporablja RAZOR programski jezik, kar pomeni, da spremenljivke izpisuje z uporabo afne (@message), VUE pa uporablja Mustashe, kar pomeni, da spremenljivke izpisuje z uporabo dvojnih zavitih oklepajev ({{ message }})

2.4.3 Prikaz podatkov - uporaba znotraj HTML atributov

VUE	*Blazor*

`<template>`
`  <div>`
`  <span v-bind:title="message">`
`  Hover your mouse over me for a few seconds`
`  to see my dynamically bound title!`
`  </span>`
`  </div>`
`</template>`
 	
`<script>`
`  export default {`
`  data () {`
`  return {`
`  message: 'You loaded this page on '`
	`<div>`
	`  <span title="@message">`
	`  Hover your mouse over me for a few seconds`
	`  to see my dynamically bound title!`
	`  </span>`
	`</div>`
	
	`@code {`
	`  private string message =`
	`  $"You loaded this page on`
	`  {DateTime.Now.ToLongDateString()}`
	
	`{DateTime.Now.ToLongTimeString()}";`
	`}`
	`]]>`

```
          + new Date().toLocaleString()
      }
    }
  }
</script>
```

VUE za spremembo lastnosti HTML elementov uporablja v-bind, medtem ko Blazor še vedno ohranja poenoten način z uporabo afne. Gornji primer uporabniku izpiše trenutni datum in čas, v kolikor se z miško dovolj dolgo zadrži na DIV elementu.

2.4.4 Pogojni prikaz

VUE

```
<template>
  <div @click="ToogleMessage">
    <span v-if="seen">Now you see
me</span>
    <span v-else>Click Me!</span>
  </div>
</template>

<script>
  var app = new VUE({
    el: "#app",
    data: {
      seen: true
    },
    methods: {
      ToogleMessage() {
        this.seen = !this.seen;
      }
    }
  });
</script>
```

Blazor

```
<div @onclick="ToogleMessage">
  @if (seen)
  {
    <span>Now you see me</span>
  }
  else
  {
    <span>Click Me!</span>
  }
</div>

@code {
  private bool seen = false;

  private void ToogleMessage() =>
            seen = !seen;
}
```

Tukaj pa se vidi prva slabost RAZOR programskega jezika, saj ne pozna posebnih HTML lastnosti, s pomočjo katerih bi lahko nadzirali posamezne elemente. VUE tako pozna ukaz v-if (tudi v-show), kar je super orodje za pogojno prikazovanje elementov. Po drugi strani pa RAZOR ohranja koncept programskega jezika, zato je pač potrebno kodo zapreti v pogojni stavek. Resnici na ljubo lahko prikaz elementa kontroliramo tudi z uporabo CSS lastnosti, vendar tukaj ni moj namen prikazati različne opcije, temveč preprosto razliko med obravnavo vejitev v HTML kodi.

2.4.5 Zanke

VUE

```
<template>
  <div>
    <ol>
      <li v-for="todo in todos">
        {{ todo.text }}
      </li>
    </ol>
  </div>
</template>

<script>
  export default {
    data () {
```

Blazor

```
<div>
  <ol>
    @foreach(var item in todos)
    {
      <li>@item</li>
    }
  </ol>
</div>

@code {
  private string[] todos = {
    "Learn C#",
    "Learn .NET Core",
```

```
      return {
        todos: [
          { text: 'Learn JavaScript' },
          { text: 'Learn VUE' },
          { text: 'Build something
awesome' }
        ]
      }
    }
  }
</script>
```

```
    "Build something awesome"
  };
}
```

Tudi pri zankah je (meni osebno) VUE lepši, saj zopet uporabi vgrajeni konstrukt v-for, medtem ko Blazor ohranja klasično zanko, ki se grdo zareže v sicer lepo HTML kodo.

2.4.6 JavaScript in Blazor

Iz Blazor aplikacije lahko seveda uporabljate in kličete JavaScript funkcije in knjižnice. V primeru VUE je to seveda samoumevno, saj je VUE napisan v JavaScript. V primeru Blazor aplikacije pa to ni tako samoumevno.

Kličete lahko tako JavaScript kodo in Blazor aplikacije, kot tudi Blazor kodo iz JavaScript aplikacije. Povezljivost je torej 100%.

Na žalost pa zadeva ne deluje tako, da bi lahko programer znotraj Blazor aplikacije preprosto pisal v JavaScript, temveč je potrebno napisati JavaScript funkcijo v neki .js datoteki, ki se naloži in potem jo lahko Blazor pokliče preko vgrajenega objekta IJSRuntime.

Recimo primer iz testne aplikacije:

```
js.InvokeAsync<string>("highlightCode", code, language);
```

V tem primeru je js objekt tipa JSRuntime, funkcija highlightCode, pa je JavaScript funkcija, ki sprejme dva parametra in vrača string:

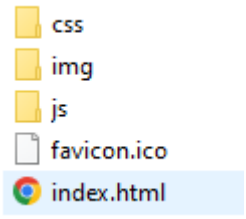
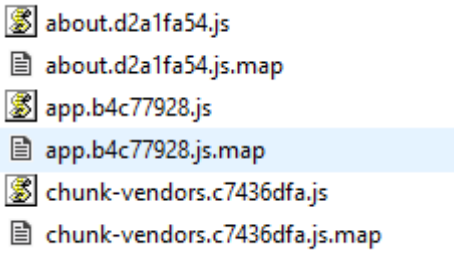
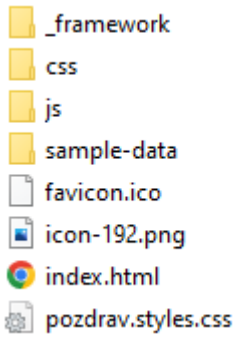
```
function highlightCode(code, language)
{
  let result = hljs.highlight(code, { language, ignoreIllegals: true })
  return result.value
}
```

V tem prispevku bom s programskimi primeri končal, saj sem jih napisal samo v toliko, da dobite občutek in vidite, da je zadeva zares zelo podobna. Bi pa v nadaljevanju pokazal, kaj dobimo, ko projekt prevedemo.

2.5 Prevod projekta

Ko smo s programiranjem zaključili in želimo naš produkt postaviti na spletno stran, ga je potrebno seveda najprej prevesti. VUE v ta namen uporablja klasičen nabor Node orodij in programsko kodo prevede s pomočjo orodja Webpack, Blazor pa uporabi .NET Core prevajalnik, ki ga razvijalec dobi z instalacijo .NET Core razvojnih orodij.

Toda ne glede na prevajalnik, slednji pač pripravi neko prevedeno kodo, ki jo je potrebno prepisati na strežnik. In vprašanje je, kaj pripravi kateri prevajalnik.

VUE	*Blazor*
VUE pripravi **dist** podmapo, ki ima lahko več podmap (odvisno od velikosti in kompleksnosti projekta), zagotovo pa je med podmapami tudi **js** podmapa, kamor **VUE** prevede program:  **js** podmapa torej vsebuje več ali manj **.js** datotek, spet seveda odvisno od velikosti in kompleksnosti projekta:  	Blazor lahko projekt prevede tako, da je direktno pripravljen za določene platforme, kot sta recimo Azure ali IIS. Ima pa tudi možnost prevoda v mapo na disku, kar lahko potem prepišemo na spletni strežnik. To je enakovredno VUE in zato prikazujem to opcijo. Blazor pripravi mapo **wwwroot**, ki ima, podobno kot v primeru VUE, več ali manj podmap, zagotovo pa ima podmapo **_framework**  in podmapa **_framework** je pravzaprav največja težava Blazor aplikacije. V njej se namreč nahajajo vse .NET osnovne knjižnice (DLL-i), katerih število je odvisno od kompleksnosti projekta, jih je pa v vsakem primeru veliko

Slika 3: Blazor Server Side.
Vir: lasten.

Blazor pripravi knjižnice v nestisnjeni obliki, s končnico **.dll**, kakor tudi stisnjene različice **.br** in **.gz.**

2.5.1 Uporaba Blazor WASM aplikacije na Apache spletnem strežniku

Na žalost ni dovolj, da na Apache spletni strežnik prepišemo samo wwwroot podmapo, temveč je seveda potrebno nastaviti tudi slavno Apache **.htaccess** datoteko. Osnovni primer takšne .htaccess datoteke je preprosto najti, je pa smiselno Apache prepričati, naj uporabi **.br** oz **.gz** datoteke (stisnjene datoteke), v kolikor jih uporabnik sprejme in večina uporabnikov jih sprejme. Na ta način lahko potrebno število prenesenih kB zmanjšamo kar za kakšnih 60%!

Tukaj prilagam **.htaccess** datoteko, ki jo sam uporabljam na Apache strežniku. V njej sem z odebeljeno pisavo označil element **podmapa-programa,** saj je to neka poljubna podmapa aplikacije na Apache strežniku. Smiselno je namreč, da Apache gosti več kot eno aplikacijo:

```
<IfModule mod_headers.c>
    # V kolikor je možno, uporabi brotli datoteke
    RewriteCond "%{HTTP:Accept-encoding}" "br"
    RewriteCond "%{REQUEST_FILENAME}\.br" "-s"
    RewriteRule "^(.*)\.(js|json|css|dll|dat|blat|wasm)$" "$1\.$2\.br" [QSA]

    # Pošlji ustrezen tip in prepreci dvojno stiskanje
    RewriteRule "\.css\.br$" "-" [T=text/css,E=no-brotli:1]
```

```
    RewriteRule "\.js\.br$"  "-" [T=text/javascript,E=no-brotli:1]
    RewriteRule "\.json\.br$"  "-" [T=application/json,E=no-brotli:1]
    RewriteRule "\.dll\.br$"  "-" [T=application/x-msdownload,E=no-brotli:1]
    RewriteRule "\.dat\.br$"  "-" [E=no-brotli:1]
    RewriteRule "\.blat\.br$"  "-" [E=no-brotli:1]
    RewriteRule "\.wasm\.br$"  "-" [E=no-brotli:1]

    <FilesMatch
"(\.js\.br|\.css\.br|\.json\.br|\.dll\.br|\.dat\.br|\.blat\.br|\.wasm\.br)$">
      Header append Content-Encoding br
      Header append Vary Accept-Encoding
    </FilesMatch>
</IfModule>

<IfModule mod_rewrite.c>
  RewriteEngine On
  RewriteBase /podmapa-programa/
  RewriteRule ^index\.html$ - [L]
  RewriteCond %{REQUEST_FILENAME} !-f
  RewriteCond %{REQUEST_FILENAME} !-d
  RewriteRule . /podmapa-programa/index.html [L]
</IfModule>
```

2.5.2 Verzioniranje

S sodobnimi brskalniki naletimo še na eno težavo in to je predpomnilnik. Brskalniki si namreč ob obisku spletne strani zapomnijo naložene datoteke in v kolikor je lokacija datoteke enaka neki prejšnji, je pač ne zahtevajo s strežnika, temveč jo vzamejo iz predpomnilnika.

Kje je težava? Težava se pojavi, ko mi programsko kodo popravimo, Blazor pa posamezne datoteke še vedno enako poimenuje. To pa pomeni, da uporabnik, čeravno smo mi prepisali novo verzijo, lahko še vedno pridobiva stare datoteke iz lastnega predpomnilnika. Ali pa se zgodi še hujša nočna mora in uporabnik dobi nekaj starih datotek iz predpomnilnika in nekaj novih s strežnika.

VUE to lepo rešuje z dodajanjem številk na konec datotek. Na prejšnji strani se lepo vidi, da je VUE eno izmed JavaScript datotek poimenoval app.b4c77928.js, kjer je b4c77928 pač »hash vrednost«, ki se bo ob naslednjem prevodu spremenila. V originalu se datoteka namreč imenuje app.js. Tega, po mojem vedenju, v tem trenutku, Blazor še nima zadovoljivo rešenega.

V dosedanjih projektih je zadostovalo, da sem napisal majhen pomožni programček, ki datoteko index.html popravi za prevajalnikom. Prevajalnik torej pripravi datoteko index.html, jaz pa nadalje popravim tako, da povezavam dodam še edinstven atribut.

V index.html tako recimo piše:

```
<link href="css/main.css" rel="stylesheet">
```

pomožni programček pa to popravi v:

```
<link href="css/main.css?cache_version=1.1.44" rel="stylesheet">
```

kjer je številka 1.1.44 nekaj, kar sam kontroliram. Na ta način dosežem, da strežnik vedno posreduje datoteke uporabniku, v kolikor se številka pač spremeni. Programsko kodo in primere lahko najdete na **GitHub** spletišču: https://github.com/MPrtenjak/OTS2022-Blazor

3 Zaključek

S programiranjem se ukvarjam profesionalno 25 let, sicer pa seveda še dobrih 10 let več. Moji začetki segajo v čase vzpona objektnega programiranja in jezika C++. Osebno se torej dobro počutim v statičnih jezikih z znano strukturo in čim manj dinamičnosti.

Preveč dinamičnosti mi ne diši, saj dnevno vzdržujem starejšo programsko kodo (tudi več kot 20 let staro!) in ne predstavljam si, kako bi to počel, če bi bila koda dinamična in bi me prevajalnik ne mogel opozoriti na določeno vrsto napak.

Ta uvod je pomemben, saj je to tudi razlog, da mi je Blazor tehnologija resnično všeč in se v njej počutim bolje, kot ob razvoju z JavaScript orodji. Vsaka izmed opcij seveda ima določene prednosti in slabosti, izmed katerih sem jih nekaj omenil v tem prispevku, nekaj pa v samem predavanju na konferenci.

Toda če nekako rešimo največjo težavo BLAZOR WASM aplikacije in to je dolgi začetni zagon (veliko DLL-ov) in posledično tudi veliko kB prenesenih podatkov, potem zame sploh ni nikakršne dileme in vedno bi raje izbral Blazor WASM pred neko JavaScript knjižnico.

Sam osebno vidim rešitev v tem, da bi se .NET WebAssembly knjižnice instalirale v brskalnik podobno kot v računalnik in potem bi lahko VSE aplikacije uporabljale tiste skupne knjižnice. Upam, da bodo sčasoma torej obstajali neki nabori standardnih knjižnic, ki jih bo lahko uporabnik dodal brskalnikom, WASM programi pa jih bodo lahko uporabili, ne da bi jih morali sami vleči s spleta.

Dokler pa bo torej Blazor imel težavo z veliko količino prenesenih podatkov, pa bom pri rešitvah uporabljal tudi VUE oz. bom med njima preklapljal glede na konkretne zahteve in potrebe.

Literatura

[1] WabAssembly, https://webassembly.org/, obiskano 15.07.2022

[2] Blazor, https://dotnet.microsoft.com/en-us/apps/aspnet/web-apps/blazor, obiskano 15.07.2022

[3] Microsoft Silverlight, https://www.microsoft.com/Silverlight/, obiskano 15.07.2022

[4] .NET Foundation, https://dotnetfoundation.org/, obiskano 15.07.2022

[5] Adobe Flash Wiki, https://en.wikipedia.org/wiki/Adobe_Flash, obiskano 15.07.2022

[6] JQuery knjižnica, https://jquery.com/, obiskano 15.07.2022

[7] VUE Introduction, https://v2.VUEjs.org/v2/guide/index.html, obiskano 16.07.2022