Razvoj spletnih rešitev na osnovi ogrodja Angular z uporabo mikro čelnih zaledij

Alen Granda[1], Matic Strajnšak[2]

[1] Univerza v Mariboru, Fakulteta za elektrotehniko, računalništvo in informatiko, Maribor, Slovenija
alen.granda@student.um.si
[2] Alcad d.o.o., Zgornja Bistrica, Slovenija
matic.strajnsak@alcad.si

Sinopsis V današnjih časih so moderne spletne rešitve vedno bolj kompleksne. Posledično jih monolitno usmerjena arhitektura ne more več obvladovati, razvoj postane težaven, vzdrževanje pa nevzdržno. Prispevek predstavlja arhitekturni pristop mikro čelnih zaledij, ki rešuje opisano težavo. Pristop prinaša obetavne ugodnosti napram monolitni arhitekturi. Kljub temu se moramo vprašati, ali je naša aplikacija primerna za izbiro mikro čelnega zaledja. Ob nepravilni izbiri lahko koncept pripelje več negativnih posledic kot pozitivnih. Zato je v prispevku izpostavljeno, v katerih primerih je uporaba omenjene arhitekture primerna in opišemo njene pozitivne ter negativne lastnosti. Nato so opisane implementacijske podrobnosti v spletnem ogrodju Angular ter je predstavljen primer uporabe v praksi. Dodani so še koristni napotki ter možne izboljšave predstavljenega demonstracijskega primera v spletnem ogrodju Angular. Nazadnje smo izpostavili funkcionalnost samostojnih komponent spletnega ogrodja Angular, katerih namen je nadomestiti datoteke nalaganja modulov. Opisali smo razlike napram trenutni uporabi datotek nalaganja modulov ter prikazali primer izvedbe v praksi.

Ključne besede:

mikro čelna zaledja

Angular

Webpack 5

samostojna komponenta

Univerzitetna založba Univerze v Mariboru

ISBN 978-961-286-639-6
DOI https://doi.org/10.18690/um.feri.10.2022.11

1 Uvod

Razvoj spletnih aplikacij je skozi zgodovino prehodil mnogo različnih arhitekturnih konceptov in je dandanes eden izmed bolj priljubljenih pristopov za namen komunikacije s končnim uporabnikom. Tradicionalne rešitve iz preteklosti ponujajo monoliten razvoj spletnih rešitev zaradi njihove enostavnosti, nizkih realnočasovnih zahtev ter majhnega števila uporabnikov [11]. Kljub temu je uporaba omenjenega pristopa v današnjih časih, za katerega so značilni kompleksni sistemi z velikimi količinami podatkov in odjemalcev, neuporabna. Uporabnikov je vedno več, so vedno bolj zahtevni in uspešni v iskanju napak v aplikacijah. Zaradi tega so spletne aplikacije z vsakim popravkom ali dodano funkcionalnostjo večje. Posledično je vzdrževanje takšnih aplikacij zahtevno in izpostavljeno akumuliranju nepravilnosti v programski kodi. V ta namen se je že konec leta 2016 začelo govoriti o mikro čelnih zaledjih (angl. microfrontends) [1], ki razširijo koncept mikro storitev v svet čelnih zaledij spletnih aplikacij. Arhitektura razdeli bogato monolitno spletno aplikacijo na več manjših, obvladljivih aplikacij, katere lahko več razvojnih ekip neodvisno druga od druge razvijajo. Pristop mikro čelnih zaledij je postal uveljavljen način razvoja spletnih aplikacij med več podjetji, kot so: DAZN, Ikea, New Relic, SAP, Springer, Starbucks, Zalando ter mnogo ostalih [10].

V nadaljevanju bolj podrobno opišemo princip mikro čelnih zaledij, njihovih prednosti in slabosti ter predstavimo implementacijo arhitekture v spletnem ogrodju Angular. Za namen demonstracije smo ustvarili projekt, ki je sestavljen iz več mikro aplikacij ter lupine, ki je zadolžena za nalaganje mikro aplikacij na zahtevo. Mikro aplikacije smo namenoma razvili z različnimi verzijami, da smo lahko demonstrirali, kako lupina pouporabi enake verzije oziroma jih naloži ob prvem zahtevku. Obrazložili smo postopek konfiguracije celotnega projekta, njegove strukture ter zagona. Podali smo še nekaj ključnih informacij ob razvoju ter možnosti za izboljšave.

Prav tako smo predstavili vidik samostojnih komponent v spletnem ogrodju Angular. Samostojna komponenta služi nadomestitvi razredom nalaganja modulov, ki so okrašeni z dekoratorjem @NgModule. Ti razredi so namreč namenjeni le prevajalniku, ki z njihovo pomočjo poskrbi za uvoz, izvoz, deklaracijo in souporabo delov aplikacije. Samostojne komponente smo integrirali v demonstracijski projekt mikro čelnih zaledij, kjer se nam je to zdelo smiselno. Prav tako smo podali lastno mnenje glede integracije samostojnih komponent v obstoječe aplikacije čelnih zaledij.

2 Mikro čelna zaledja

2.1 Korelacija z mikro storitvami

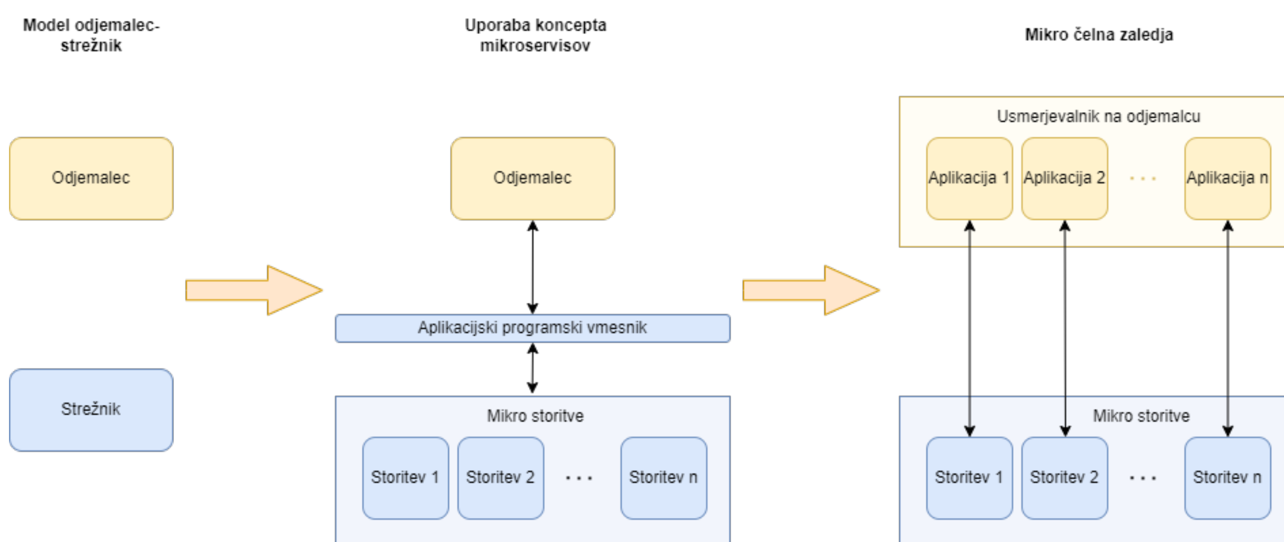
Pričetek arhitekture spletnih rešitev sega vse do modela odjemalec-strežnik [12]. Strežnik je namenjen ponujanju virov, ki jih odjemalec občasno zahteva čez internet. Koncept je zelo razširjen in je bil povod v pristop načrtovanja programske opreme, imenovan model-pogled-krmilnik (angl. Model-View-Controller - MVC) [12]. MVC razdeli aplikacijo na tri dele:

- Model: namenjen opisu podatkov, njihovemu procesiranju in implementaciji poslovne logike.
- Pogled: uporabniški vmesnik aplikacije.
- Krmilnik: skrbi za interakcijo med uporabnikom in izgledom ter posreduje akcije modelu.

Zaradi več uporabnih vrednosti se je MVC izkazal kot izredno uporaben arhitekturni načrtovalski vzorec ter je posledično še danes eden izmed bolj popularnih načinov razvoja spletnih aplikacij. Kljub temu je arhitektura zastavljena monolitno, kar pomeni, da je celotna aplikacija oblikovana kot skupek modulov, ki imajo medsebojne odvisnosti. S povečevanjem zahtev se moduli širijo in množijo. Posledično je meja med njimi vedno težje definirana, vzdrževanje zahtevno in možnost napak v programski kodi naenkrat začne rasti.

Na področju zalednih sistemov so omenjen zaplet pričeli reševati s pristopom mikro storitev okoli leta 2012 [12]. Mikro storitve temeljijo na storitveno usmerjenem arhitekturnem vzorcu (angl. Service-Oriented Architecture) in so predstavljene kot zbirka ohlapno povezanih storitev [11]. Razbijejo kompleksne zaledne aplikacije v manjše, obvladljive storitve, ki komunicirajo skozi zbirko jezikovno neodvisnih aplikacijskih programskih vmesnikov. Vsako storitev je mogoče neodvisno razvijati, testirati in namestiti, kar omogoča uporabo načela enotne odgovornosti (angl. Single-responsibility principle). Posledično je zaradi majhnosti razvoj lažji, razširljiv ter omogoča koncept neprekinjene dobave in namestitve (angl. Continuous delivery and deployment), saj s spremembo ene mikro storitve ne vplivamo na druge.

Prednosti uporabe mikro storitev na zalednem sistemu so očitne. Zato je pred kratkim v veljavo pristopil podoben koncept za čelni del spletnih rešitev, imenovan mikro čelna zaledja. Mikro čelna zaledja rešujejo težave monolitnih aplikacij tako, da razdelijo aplikacijo na več manjših, neodvisnih mikro aplikacij. Ponujajo torej prikaz spletne aplikacije kot kombinacijo funkcionalnosti, ki so pod nadzorom več različnih razvojnih ekip [11]. Glavna aplikacija čelnega zaledja deluje kot usmerjevalnik, ki lahko dinamično nalaga mikro čelne aplikacije. Slika 1 vizualno prikazuje razvoj arhitekture spletnih aplikacij skozi čas ter idejo mikro čelnih zaledij spletnih aplikacij.



Slika 1: Razvoj arhitektur spletnih aplikacij.
Vir: lasten.

Ključna ideja in doprinosi mikro čelnih zaledij spletnih aplikacij so zajeti v sledečih pogledih [10]:

- Neodvisnost ekip ob izbiri tehnologije za razvoj,
- Avtonomne in večnamenske ekipe. Vsaka ekipa ima namreč možnost razvoja dela spletne aplikacije, od mikro čelnega aplikacije do mikro storitve.
- Odpornost mikro čelnih aplikacij na izpade ostalih mikro čelnih aplikacij.
- Razširljivost. Majhne aplikacije je namreč lažje dodelati in razširiti.
- Takojšnje nameščanje novih verzij mikro čelnih aplikacij zaradi njihove majhnosti in samostojnosti.
- Enostavno testiranje in integracija novih spletnih ogrodij v arhitekturo mikro čelnih zaledij.

Kljub temu arhitekturni stil s seboj pripelje razne pomisleke:

- Zvišana količina podatkov, potrebnih za prenos. Čez omrežje moramo namreč prenesti več različnih spletnih ogrodij, kar lahko vpliva na čas nalaganja in končno uporabniško izkušnjo.
- Podvajanje programske kode zaradi neodvisnega razvoja.
- Upravljanje odvisnosti med programskimi knjižnicami mikro čelnih aplikacij ter zagotavljanje enkratnega nalaganja programske knjižnice, ki je zahtevana s strani več mikro čelnih aplikacij.

- Upravljanje med različnimi verzijami naloženih programskih knjižnic.
- Konsistentnost v izgledu aplikacije.
- Razhroščevanje in upravljanje aplikacije.
- Dodana kompleksnost.
- Časovna potratnost zaradi morebitnega podvajanja iskanja rešitve za problem, ki ga je v preteklosti že razrešila druga ekipa.
- Nevarnost objave napak v produkcijsko aplikacijo zaradi takojšnjega razvoja in namestitve.
- Orkestracija in obravnava nalaganja mikro čelnih aplikacij, kadar so zahtevane.
- Usmerjanje mikro čelnih aplikacij.
- Komunikacija med mikro čelnimi aplikacijami.

Z omenjenimi pozitivnimi in negativnimi vplivi uporabe pristopa mikro čelnih zaledij v mislih podajmo nekaj primerov, kdaj arhitekturni načrtovalski vzorec ponuja več neugodnih učinkov kot spodbudnih:

- Aplikacija čelnega zaledja vsebuje veliko komponent, ki so med seboj povezane. V tem primeru moramo poskrbeti za dodaten sistem komunikacije med komponentami mikro čelnih zaledij spletne rešitve, kar dodatno poveča zapletenost in oteži razhroščevanje celotne aplikacije. Zato je arhitekturni načrtovalski vzorec primeren zlasti za aplikacije z zelo malo povezanimi ali z nepovezanimi komponentami.

- Spletna programska rešitev je namenjena izvajanju manjšega števila opravil. Delitev programske rešitve na mikro čelna zaledja bi po nepotrebnem pripeljala kompleksnost ter posledično povišano možnost nabiranja nepravilnosti v programski kodi. Dodatno se podaljša čas nalaganja aplikacije in naraste zahteva po količini potrebnega pomnilnika v primeru nekompatibilnih verzij programskih knjižnic ali več različnih spletnih ogrodij.

- Potreba po avtorizaciji in avtentikaciji uporabnikov oziroma uporaba koncepta enotne prijave (angl. Single sign-on). Gre za poseben primer relacije med komponentami aplikacije, za katerega moramo pomisliti, kako zagotoviti zavedanje mikro čelnih zaledij spletnih aplikacij o uporabnikih, brez da se mora uporabnik prijaviti ob vsakem nalaganju mikro čelne aplikacije. Skupnost predlaga uporabo t.i. strategije mono repozitorijev (angl. monorepo) [6], ki s seboj prinesejo že omenjene posledice v kombinaciji z mikro čelnimi zaledji.

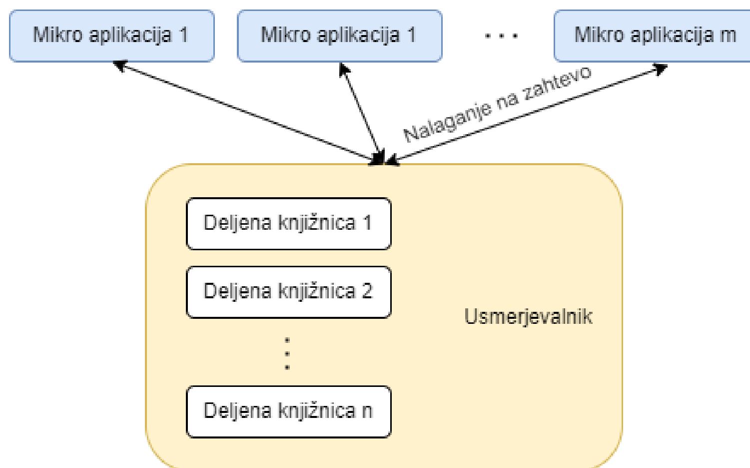
V nadaljevanju predstavimo predloge, kako zaobiti določenim slabostim mikro čelnih zaledij v spletnem ogrodju Angular. Pred tem poglejmo, kaj je implementacijska rešitev uporabe koncepta mikro čelnih zaledij v spletnem okolju ogrodij JavaScript.

2.2 Webpack 5

Opazili smo, da je pristop mikro čelnih zaledij spletnih rešitev močno povezan s konceptom mikro storitev. Mikro storitve imajo v svetu zalednih sistemov relativno naravno implementacijsko rešitev. Kljub temu je bilo v svetu čelnih zaledij spletnih aplikacij v preteklosti težavno implementirati arhitekturo mikro čelnih zaledij. Težava je bila v popularni tehnologiji prevajanja in povezovanja modulov JavaScript, imenovani Webpack [9]. Tehnologija Webpack je predvidevala, da je celotna programska koda na voljo med prevajanjem [3]

Leta 2020 nam je razvoj tovrstne arhitekture olajšal prihod vtičnika za povezovanje modulov (angl. Module federation) v tehnologiji Webpack 5 [2]. Vtičnik omogoča aplikaciji, ki deluje na osnovi programskega jezika JavaScript, dinamično nalaganje ločeno prevedenih delov programske kode. Posledično lahko dostopamo do delov programske kode, ki še niso znani v času prevajanja. Prav tako je omogočeno deljenje skupnih programskih knjižnic, kar onemogoči podvajanje. Na ta način lahko arhitekturo aplikacije zastavimo tako, da jo razdelimo na več ločenih aplikacij (Slika 2):

- Usmerjevalnik: služi dinamičnemu nalaganju mikro aplikacij, upravljanju deljenih programskih knjižnic, obravnavi nedostopnosti mikro aplikacij in podobno.
- Mikro aplikacija: del originalne aplikacije, delitev določena po smislu.



Slika 2: Shema mikro arhitekture čelnega zaledja spletne rešitve.
Vir: lasten.

Od prihoda tehnologije Webpack 5 je razvoj mikro čelnih zaledij v spletnih rešitvah močno pridobil v popularnosti. Koncept smo implementirali tudi mi, saj nas je zanimalo, ali se nam obrestuje. V nadaljevanju predstavimo implementacijske podrobnosti v spletnem ogrodju Angular.

3 Angular

3.1 Mikro čelna zaledja v spletnem ogrodju Angular

Angular je razvojno spletno ogrodje, zgrajeno na podlagi programskega jezika TypeScript. Omogoča nam:

- Multi platformnost: izdelava spletnih strani, mobilnih aplikacij z uporabo strategij iz ogrodja Ionic, aplikacij za namizne računalnike z uporabo ogrodja Electron.
- Učinkovitost: optimizacija prevajanja programske kode v virtualno napravo programskega jezika JavaScript.
- Produktivnost: uporaba predlog, komponent, orodij ukazne vrstice, podpora v integriranih razvojnih okoljih, ponuja bogat nabor knjižnic.
- Enostavna implementacija internacionalnih aplikacij.

Mi se osredotočimo predvsem na uporabo spletnega ogrodja Angular za namen implementacije mikro čelnih zaledij v spletnih rešitvah. V ta namen smo uporabili Angular verzije 14.0, ki vključuje podporo tehnologiji Webpack 5. Posledično se lahko poslužimo uporabe vtičnika povezovanja modulov. Implementacijo mikro čelnih zaledij predstavimo na enostavnem primeru aplikacije, sestavljene iz štirih strani.

3.2 Primer uporabe

Za namen demonstracije smo se odločili preoblikovati aplikacijo, ki je sestavljena iz štirih različnih strani (Slika 3). Odločili smo se, da je mikro čelno zaledje aplikacije sestavljeno iz štirih mikro aplikacij (za vsak vnos v meniju ena mikro aplikacija) ter iz usmerjevalnika, ki dinamično kliče mikro aplikacije (Slika 4).

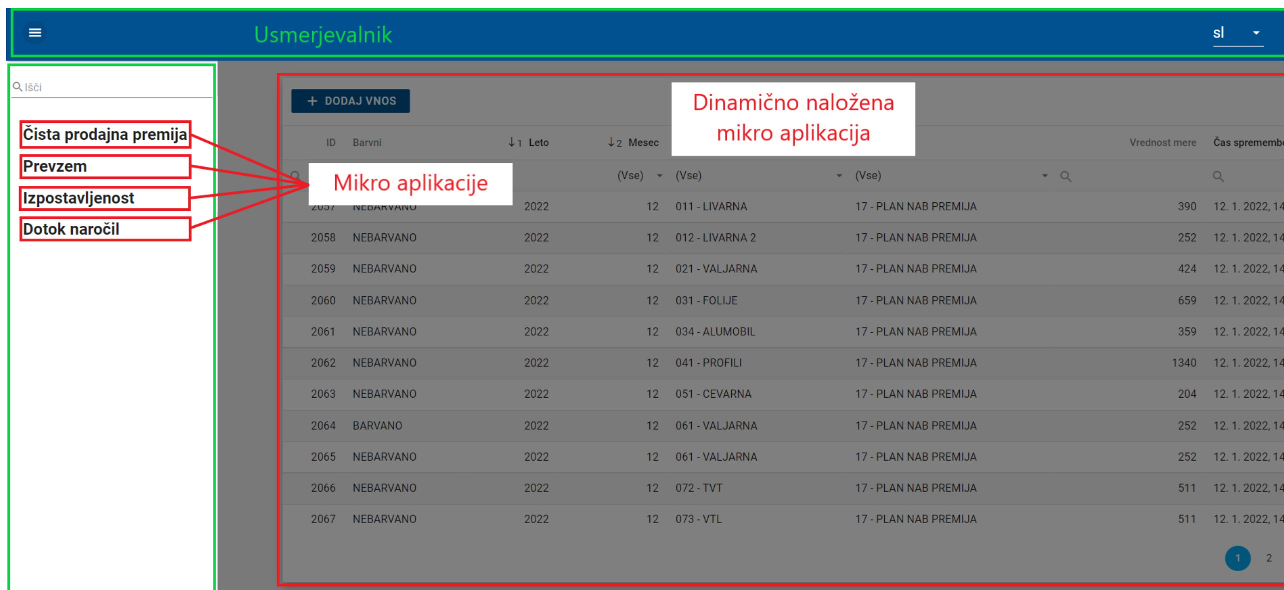
Najprej smo morali spremeniti strukturo aplikacije (Slika 5). Za namen demonstracije smo prestavili mikro aplikacijo Izpostavljenost izven strukture projekta, da lahko v nadaljevanju pokažemo, kako vtičnik povezovanja modulov razreši različne verzije ogrodja Angular. Po spremembi strukture smo ustrezno posodobili datoteko angular.json z dodanimi projekti. Nato je sledilo nalaganje knjižnice vtičnika za povezovanje modulov v spletnem ogrodju Angular [8], ki nam ponuja avtomatsko generiranje datotek, potrebnih za uporabo omenjenega vtičnika. V kolikor smo knjižnico naložili za aplikacijo Usmerjevalnik, smo v ukazni vrstici zagnali sledeč ukaz:

```
ng add @angular-architects/module-federation --project <ime_projekta_usmerjevalnik> --
type host --port 4200
```

Sicer pa smo zagnali sledeč ukaz:

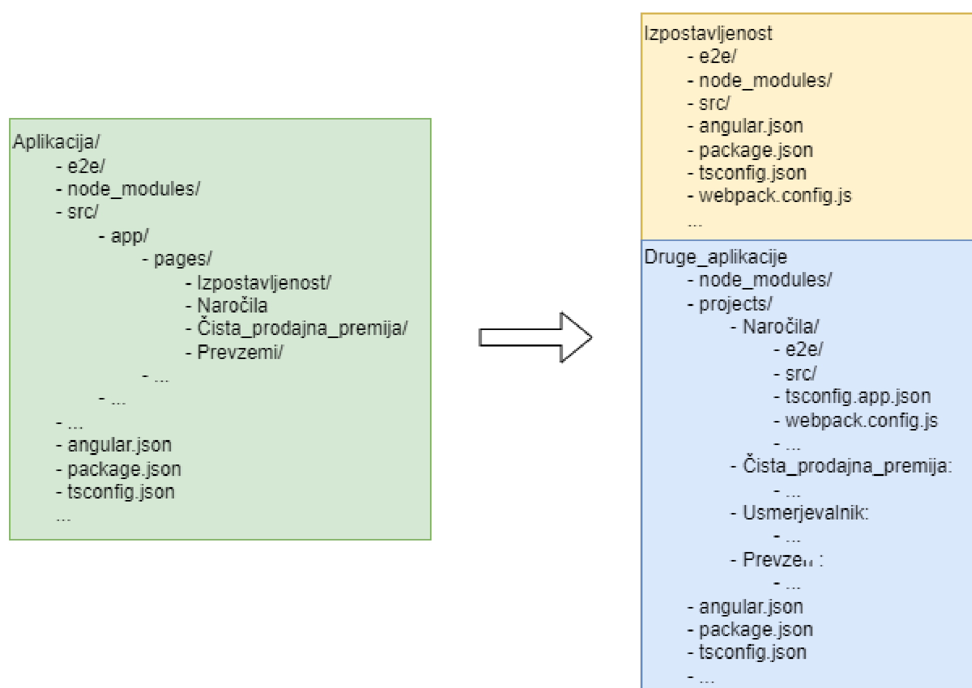
```
ng add @angular-architects/module-federation --project <ime_mikro_aplikacije> --type
remote --port 4201
```

Slika 3: Nespremenjena demonstracijska aplikacija.
Vir: lasten.



Slika 4: Delitev demonstracijske aplikacije na mikro čelne aplikacije.
Vir: lasten.

Opazimo, da je razlika le v tipu projekta ter v nastavljenih vratih, na katerih je aplikacija dostopna. Knjižnica nam generira datoteko webpack.config.js, ki služi konfiguraciji vtičnika povezovanja modulov, ter posodobi usmerjevalno datoteko app.routes.ts v aplikaciji Usmerjevalnik. Datoteko webpack.config.js moramo še ročno posodobiti tako, da definiramo, katere module mikro aplikacija izvaža in katere odvisnosti si deli z ostalimi mikro aplikacijami. Slika 6 nakazuje primer enostavnih nastavitev. Za mikro aplikacijo Usmerjevalnik je nastavitvena datoteka podobna, le da moramo določiti naslove oddaljenih mikro aplikacij, katere bo Usmerjevalnik klical (Slika 7).



Slika 5: Sprememba strukture demonstracijskega projekta.
Vir: lasten.

```
const { shareAll, withModuleFederationPlugin } = require('@angular-architects/module-federation/webpack');

module.exports = withModuleFederationPlugin({

  name: 'exposure',
  filename: 'remoteEntry.js',
  exposes: {
    './pages.module': './projects/exposure/src/app/pages/pages.module.ts',
  },

  shared: {
    ...shareAll({ singleton: true, strictVersion: true, requiredVersion: 'auto' }),
  },
});
```

Slika 6: Primer nastavitev datoteke webpack.config.js posamezne mikro aplikacije.
Vir: lasten.

```
const { shareAll, withModuleFederationPlugin } = require('@angular-architects/module-federation/webpack');

module.exports = withModuleFederationPlugin({

  remotes: {
    "exposure": "http://localhost:4201/remoteEntry.js"
  },

  shared: {
    ...shareAll({ singleton: true, strictVersion: true, requiredVersion: 'auto' }),
  },

});
```

Slika 7: Primer nastavitev datoteke webpack.config.js mikro aplikacije Usmerjevalnik.
Vir: lasten.

V obeh primerih smo za enostavnost demonstracije uporabili funkcijo programske knjižnice [8] shareAll za deljenje vseh odvisnosti, ki so določene v datoteki package.json. V kolikor imata mikro aplikaciji enake odvisnosti s kompatibilnimi verzijami, naša zahteva po deljenju odvisnosti povzroči enkratno nalaganje programskih knjižnic. S tem smo preprečili možnost večkratnega nameščanja odvisnosti.

Za prikaz delovanja nalaganja moramo le še prilagoditi usmerjanje v mikro aplikaciji Usmerjevalnik. V ta namen smo posodobili datoteko app.routes.ts (Slika 8). Ob tem omenimo, da pot 'exposure/pages.module' ne obstaja v aplikaciji Usmerjevalnik. Je le navidezna pot, ki nakazuje na oddaljeno mikro aplikacijo. Tega se prevajalnik ne zaveda, zato moramo ustvariti deklaracijsko datoteko decl.d.ts, v kateri deklariramo modul 'exposure/pages.module'.

V kolikor zaženemo obe aplikaciji v ločenih terminalih na vratih 4200 in 4201, opazimo, da Usmerjevalnik uspešno naloži oddaljeno mikro aplikacijo Izpostavljenost. Podoben postopek smo storili še za ostale mikro aplikacije in uspešno pretvorili začetno monolitno aplikacijo čelnega zaledja v več neodvisnih mikro aplikacij čelnega zaledja.

```
import { loadRemoteModule } from '@angular-architects/module-federation';
import { Routes } from '@angular/router';

export const APP_ROUTES: Routes = [
    {
      path: '',
      redirectTo: 'exposure',
      pathMatch: 'full'
    },

    {
      path: 'exposure',
      loadChildren: () => import('exposure/pages.module').then(m => m.PagesModule)
    },
];
```

Slika 8: Oddaljeno nalaganje mikro aplikacije.
Vir: lasten.

3.3 Koristni napotki in izboljšave

V kolikor pogledamo nastavitvene datoteke webpack.config.js, ki smo jih predstavili v primeru uporabe, vidimo, da smo za namen enostavnosti uporabili deljenje vseh paketov, ki jih mikro aplikacija koristi. Včasih to ni scenarij, ki bi si ga želeli. Še posebej, če mikro aplikacija vsebuje veliko takšnih modulov. V okolju soodvisnosti se namreč ne moremo poslužiti prednosti odstranitve tako imenovane mrtve kode (angl. Dead code), kar lahko povzroči večje količine nalaganja ločenih paketov programske kode. Kot optimizacijo nam programska knjižnica [8] omogoča uporabo funkcije share kot nadomestilo funkciji shareAll, v kateri ročno določimo, katere pakete delimo z ostalimi mikro aplikacijami.

Demonstracijski projekt je zastavljen tako, da se vse mikro aplikacije naložijo ob zagonu Usmerjevalnika. V primeru večjega števila mikro aplikacij je takšna implementacija neželena zaradi daljšega nalaganja. Prav tako nam aplikacija Usmerjevalnik prikaže prazno stran, v kolikor katera izmed mikro aplikacij ni dosegljiva, kar je v nasprotju s principi mikro čelnih zaledij spletnih rešitev. Z malo sprememb lahko omogočimo dinamično nalaganje oddaljenih mikro aplikacij na zahtevo, ki reši predstavljeni težavi. Vse kar moramo storiti je spremeniti 3 datoteke v projektu Usmerjevalnik (Slika 9):

- Potrebno je ustvariti datoteko /src/assets/mf.manifest.js, v kateri definiramo vse oddaljene mikro aplikacije v obliki formata JSON.
- Prilagoditi moramo datoteko main.ts tako, da najprej pokličemo funkcijo loadManifest, ki jo ponuja programska knjižnica [8]. S tem naložimo predhodno ustvarjeno datoteko, ki vsebuje vse naslove oddaljenih mikro aplikacij.
- Nadomestimo vsa mesta v datoteki app.routes.ts, kjer smo vključili vnaprej deklariran modul oddaljene mikro aplikacije, z metodo loadRemoteModule programske knjižnice [8]. Le-ta poskrbi za dinamično klicanje oddaljene mikro aplikacije ob zahtevi.

Kot smo že omenili, smo za namen demonstracije podpore različnih verzij spletnega ogrodja Angular mikro aplikacijo Izpostavljenost izolirali (Slika 5). Verzijo spletnega ogrodja Angular smo tej aplikaciji nastavili na 14.0.0 ter ustrezno posodobili pripadajočo datoteko webpack.config.js. Verzijo preostalih mikro aplikacij smo prepustili trenutno najnovejši verziji 14.0.3. Slika 7 prikazuje, da aplikacija Usmerjevalnik zahteva enkratno nalaganje deljenih knjižnic. V kolikor želimo naložiti mikro aplikacijo Izpostavljenost v Usmerjevalnik, nas Angular obvesti z napako: »Error: inject() must be called from an injection context«. S tem nam pove, da želimo naložiti več različnih verzij enake knjižnice, čeprav smo zahtevali enkratno nalaganje. Posledično se obe verziji naložita, prikaže pa se nam nič ne na zaslon. Ena izmed rešitev je sprememba datoteke webpack.config.js na strani mikro aplikacije Usmerjevalnik tako, da kot parameter deljene knjižnice podamo le možnost »requiredVersion: 'auto'«. Na ta način smo omogočili nalaganje več različnih verzij ene knjižnice, v kolikor med seboj niso kompatibilne. Kljub temu bi omenili, da nam

ta način prinese podaljšan čas nalaganja, kar je lahko v primeru več knjižnic uporabniku neprijazno. Prav tako je upravljanje nalaganja knjižnic, katerih verzije so ekstremno različne, težavno. Ob nepravilnih nastavitvah se lahko aplikacija zruši ali pa ne prikaže nič na zaslon. Zato priporočamo uporabo kompatibilnih verzij uporabljenih paketov. Z uporabo parametra »strictVersion« v nastavitvah deljenih knjižnic lahko priporočen način enostavno dosežemo, saj natančno zahtevamo verzijo deljene knjižnice. V kolikor dana verzija ni na voljo, lahko situacijo poljubno razrešimo s kakšnim obdelovalcem napak po meri.

1.) Dodajanje datoteke mf.manifest.js

```
// mf.manifest.json

{
  "sales-premium": "http://localhost:4201/remoteEntry.js",
  "exposure": "http://localhost:4202/remoteEntry.js",
  "takeover": "http://localhost:4203/remoteEntry.js",
  "orders": "http://localhost:4204/remoteEntry.js"
}
```

2.) Posodobitev datoteke main.ts

```
import('./bootstrap')
  .catch(err => console.error(err));
```

```
import { loadManifest } from '@angular-architects/module-federation';

loadManifest("/assets/mf.manifest.json")
  .catch(err => console.error(err))
  .then(_ => import('./bootstrap'))
  .catch(err => console.error(err));
```

3.) Dopolnitev datoteke app.routes.ts

```
import { loadRemoteModule } from '@angular-architects/module-federation';
import { Routes } from '@angular/router';

export const APP_ROUTES: Routes = [
    {
      path: '',
      redirectTo: 'exposure',
      pathMatch: 'full'
    },

    {
      path: 'exposure',
      loadChildren: () => import('exposure/pages.module').then(m => m.PagesModule)
    },
];
```

```
import { loadRemoteModule } from '@angular-architects/module-federation';
import { Routes } from '@angular/router';

export const APP_ROUTES: Routes = [
    {
      path: '',
      redirectTo: 'exposure',
      pathMatch: 'full'
    },

    {
      path: 'exposure',
      loadChildren: () => loadRemoteModule({
        type: 'manifest',
        remoteName: 'exposure',
        exposedModule: './pages.module'
      }).then(m => m.PagesModule)
    },
]
```

Slika 9: Potrebne posodobitve za dinamično nalaganje oddaljenih mikro aplikacij.
Vir: lasten.

Dodatno bi želeli izpostaviti, da moramo v datoteki app.module.ts aplikacije Usmerjevalnik vključiti globalne storitve (npr. storitev za omrežje - HttpClientModule), v kolikor te storitve oddaljena mikro čelna aplikacija pričakuje oziroma koristi. V kolikor tega ne storimo, oddaljena mikro čelna aplikacija ne bo vedela, kje naj dano globalno storitev poišče. Poleg tega omenimo, da oddaljene mikro čelne aplikacije ne bi smele izpostavljati svojih modulov aplikacije (angl. App module). Lahko se zgodi, da to niti ne bo mogoče zaradi napak v prevajanja modula usmerjanja (angl. Router module), ki zahteva enkratno klicanje metode forRoot. V kolikor te metode v oddaljeni mikro aplikaciji ne uporabimo, se nam lahko aplikacija celo zažene. Kljub temu je to slaba praksa, saj se s tem načinom podvojijo vse deljene storitve, ki so registrirane v obsegu mikro aplikacije [4].

4 Samostojna komponenta v spletnem ogrodju Angular

4.1 Uporabna vrednost

Pred kratkim je s prihodom verzije 14 spletnega ogrodja Angular prispela nova funkcionalnost, ki odpravi potrebo po stvaritvi nastavitvenih datotek nalaganja modulov (razredov z dekoratorjem @NgModule), ki so namenjene le prevajalniku. Funkcionalnost je poimenovana samostojna komponenta (angl. Standalone component), saj se potrebne informacije o vključevanju ostalih knjižnic prestavijo v metapodatke komponent. Na ta način zmanjšamo količino nepotrebne programske kode, olajšamo razvijalcu delo in natančno definiramo, katere programske

knjižnice potrebuje določena komponenta. Koncept samostojnih komponent ne velja le za komponente, ampak je razširjen tudi na cevi, direktive in storitve [5]. Opozoriti moramo, da je koncept samostojnih komponent zaenkrat le v predogled razvijalcem in se še lahko spremeni. Kljub temu pristop obljublja veliko pozitivnih lastnosti, zato smo želeli izkusiti, kako se obnese v praksi. V nadaljevanju opišemo postopek naše implementacije na predhodno demonstriranem primeru ter naše mnenje o doprinosu funkcionalnosti.

4.2 Integracija v demonstracijski primer uporabe mikro čelnih zaledij

Samostojne komponente si v bistvu lahko predstavljamo kot komponente z lastno nastavitveno datoteko nalaganja modulov. Integracija v demonstracijski primer je zelo enostavna. Najprej moramo v dekorator komponente dodati parameter »standalone« in mu nastaviti resnično vrednost. S tem povemo, da je komponenta samostojna. Nato v parametru »imports« določimo vse odvisnosti, ki jih komponenta potrebuje za delovanje. Kot odvisnost lahko vključimo programske knjižnice ali pa celo ostale samostojne komponente. Slika 10 prikazuje primer implementacije opisanega postopka. V kolikor po opisanih navodilih definiramo vse komponente v aplikaciji, lahko iz aplikacije brez skrbi odstranimo vse datoteke nalaganja modulov.

```
@Component({
  standalone: true,
  imports: [
    CommonModule,
    RouterModule,

    DxButtonModule,
    DxDataGridModule,
    DxDrawerModule,
    DxListModule,
    DxScrollViewModule,
    DxSelectBoxModule,
    DxToolbarModule,
    DxTreeViewModule,

    HttpClientModule,
    ToastrModule,
    BiManagementLibModule
  ],
  selector: 'app-exposure',
  templateUrl: './exposure.component.html',
  styleUrls: ['./exposure.component.scss']
})
export class ExposureComponent implements OnInit {
```

Slika 10: Primer nastavitve metapodatkov samostojne komponente.
Vir: lasten.

Preostane nam le še nastavitev začetnega nalaganja komponent in s tem posledično celotne aplikacije Angular. Do zdaj so za to poskrbele datoteke nalaganja modulov, ki smo jih izbrisali. Zato moramo na drug način razložiti spletnemu ogrodju Angular, kako naj naloži in zažene celotno aplikacijo. V ta namen nam verzija 14 spletnega ogrodja Angular ponuja funkcijo bootstrapApplication, ki smo jo uporabili v datoteki bootstrap.ts (Slika 11). Funkcija kot prvi parameter prejme glavno komponento aplikacije, kot drugi pa globalne ponudnike storitev. S tem smo opravili vse potrebne spremembe in lahko brez strahu na novo zaženemo našo aplikacijo.

Omeniti moramo, da je potrebno za delovanje posodobljene mikro čelne aplikacije posodobiti še nastavitveno datoteko webpack.config.js. Ker nimamo več razredov, ki so namenjeni nalaganju modulov, lahko oddaljenim mikro čelnim aplikacijam podamo na voljo datoteko poti (npr. app.routes.ts). Načeloma bi lahko izpostavili tudi samostojno komponento, kar pa ni dobra praksa mikro čelnih zaledij spletnih aplikacij [5]. Nato v aplikaciji Usmerjevalnik ustrezno spremenimo datoteko app.routes.ts, da naloži novo izpostavljeno usmerjevalno datoteko oddaljene mikro čelne aplikacije. Tako smo z uporabo principa samostojnih komponent uspešno nadomestili vse datoteke nalaganja modulov, ki so bile namenjene le prevajalniku.

```
if (environment.production) {
  enableProdMode();
}

bootstrapApplication(AppComponent, {
providers: [
  importProvidersFrom(RouterModule.forRoot(PAGES_ROUTES)),
  importProvidersFrom(HttpClientModule),
  importProvidersFrom(ToastrModule.forRoot({
    positionClass: 'toast-bottom-center',
    enableHtml: true
    })),
  importProvidersFrom(BiManagementLibModule.forRoot(environment))
  ]
});
```

Slika 11: Nalaganje spletne aplikacije Angular, sestavljene iz samostojnih komponent.
Vir: lasten.

5 Zaključek

Koncept mikro čelnih zaledij spletnih rešitev rešuje probleme, ki so posledica monolitnih aplikacij čelnega zaledja. Pristop izhaja iz arhitekturne rešitve mikro storitev na zalednem sistemu in prinaša mnogo pozitivnih vidikov. V spletnem ogrodju Angular se je s prihodom tehnologije Webpack 5 razvoj dane arhitekture razbesnel zaradi dodanega vtičnika povezovanja modulov in posledično enostavne implementacije v praksi. Pristop smo preizkusili in analizirali njegovo uporabnost ter dodano vrednost. Moramo se zavedati, ali je dana arhitektura primerna za naš problem. Z napačno izbranim projektom lahko z opisanim konceptom prinesemo več že opisanih negativnih posledic kot pozitivnih. Menimo, da je izbira mikro čelnih zaledij primerna predvsem za velike, dolgotrajne in kompleksne spletne rešitve z večjim številom razvijalcev.

V prispevku smo prav tako predstavili obetavno razširitev samostojnih komponent, ki jo je kot predogled prinesla verzija 14 spletnega ogrodja Angular. Funkcionalnost je implementacijsko zelo enostavna in prinaša mnogo pozitivnih pogledov, tako za razvijalca kot za arhitekturni pogled čelnih aplikacij spletnih rešitev. Koncept je preprost in enostaven za razumevanje, zato smo ga razširili v celoten ekosistem demonstracijske aplikacije. Verjamemo, da je dana razširitev eden izmed pomembnejših sprememb, ki jih ponuja verzija 14 spletnega ogrodja Angular. Menimo, da bo v prihodnosti vidik samostojnih komponent zaradi svoje praktičnosti zamenjal trenutno razširjen pristop uporabe datotek nalaganja modulov.

Literatura

[1] https://micro-frontends.org/, Micro Frontends, obiskano 22. 06. 2022.

[2] https://webpack.js.org/concepts/module-federation, Module federation, obiskano 23. 06. 2022.

[3] https://www.angulararchitects.io/en/aktuelles/the-microfrontend-revolution-module-federation-in-webpack-5/, STEYER, Manfred. The Microfrontend Revolution: Module Federation in Webpack 5, obiskano 23. 06. 2022.

[4] https://www.angulararchitects.io/en/aktuelles/pitfalls-with-module-federation-and-angular/, STEYER, Manfred. Pitfalls with Module Federation and Angular, obiskano 24. 06. 2022.

[5] https://www.angulararchitects.io/aktuelles/angulars-future-without-ngmodules-lightweight-solutions-on-top-of-standalone-components/, STEYER, Manfred. Module Federation with Angular's Standalone Components, obiskano 24. 06. 2022.

[6] https://www.angulararchitects.io/en/aktuelles/using-module-federation-with-monorepos-and-angular/, STEYER, Manfred. Using Module Federation with Nx Monorepos and Angular, obiskano 08. 07. 2022.

[7] https://www.angulararchitects.io/aktuelles/module-federation-with-angulars-standalone-components/, STEYER, Manfred. Angular's Future Without NgModules – Part 1: Lightweight Solutions Using Standalone Components, obiskano 27. 06. 2022.

[8] https://www.npmjs.com/package/@angular-architects/module-federation, @angular-architects/module-federation, obiskano 24. 06. 2022.

[9] https://webpack.js.org/, Webpack, obiskano 08. 07. 2022.

[10] PELTONEN, Severi; MEZZALIRA, Luca; TAIBI, Davide. Motivations, benefits, and issues for adopting micro-frontends: a multivocal literature review. *Information and Software Technology*, 2021, 136: 106571.

[11] YANG, Caifang; LIU, Chuanchang; SU, Zhiyuan. Research and application of micro frontends. In: *IOP conference series: materials science and engineering*. IOP Publishing, 2019. p. 062082.

[12] PAVLENKO, Andrey, et al. Micro-frontends: application of microservices to web front-ends. *J. Internet Serv. Inf. Secur.*, 2020, 10.2: 49-66.