UDC: 004.42 Info M str. 39-45

RAZVOJ SOFTVERSKOG SISTEMA ZA GENERISANJE CROSS-PLATFORM KORISNIČKOG INTERFEJSA
DEVELOPMENT OF SOFTWARE SYSTEM FOR GENERATING CROSS-PLATFORM USER INTERFACES

Dimitrije Jevtić, Prof. dr Saša D. Lazarević, Tatjana Stojanović

REZIME: U vremenu kada broj platformi i operativnih sistema raste, nailazi se na teškoće pri razvoju softvera i korisničkih interfejsa koji podržavaju iste. Većina rešenja koji se trenutno nude znače eksplicitno ili implicitno odvajanje kôda i postavljanje aplikacija na klijentske uređaje, što donosi teškoće pri održavanju, konfiguraciji pa i korišćenju takvih softverskih rešenja. U ovom radu je dat predlog centralizovanog softverskog rešenja, koji se oslanja na postojeći HTTP protokol i postojeće pretraživače kako bi eliminisali potrebu za kreiranje više izvršnih verzija iste aplikacije.

KLJUČNE REČI: korisnički interfejs, kros-platform, strane generisane na serveru, SSR

ABSTRACT: During the expansion of platforms and operating systems, difficulties are emerging during the process of development of software and user interfaces that support them. Most of current solutions mean that there is explicit or implicit differentiation of code and deployment of applications on client devices, that further implies difficulties during maintenance period, configuration and even use of such software solutions. Suggestion of centralized software solution, is given in this paper that relies on current HTTP protocol and browsers and eliminates the need for creating multiple executable versions of the same application.

KEY WORDS: user-interface, cross-platform, server-side-rendering, SSR

1. UVOD

U aplikacijama koje se svakodnevno koriste, korisnički interfejsi predstavljaju veoma bitan deo softverskog sistema. U zavisnosti od prirode aplikacije, u nekim slučajevima se može reći i da predstavljaju bitniju celinu od skupa funkcionalnosti koje softverski sistemi objavljaju. Cilj korisničkog interfejsa jeste da korisniku pruži sve potrebne informacije u pravom trenutku, bez preopterećivanja korisnika nepotrebnim informacijama. Takođe, samom korisniku, interfejs bi trebao biti smislen, tj. korisnik ne bi trebao imati dvoumljenja pri korišćenju softverskog sistema između željene akcije i očekivanog rezultata te akcije. Prava razmera količine informacija na korisničkom interfejsu se nalazi u preseku ova dva slučaja.

Dva koncepta, koje je u svojim radovima uveo D. Norman [1], se mogu primeni i na današnje korisničke interfejse. Reč je o *zalivu izvršenja* i *zalivu ocenjivanja*. Pod *zalivom izvršenja* se misli na stepen podudaranja mogućnosti sistemskog *predmeta* sa željama i očekivanjima korisnika i šta on očekuje od akcije nad predmetom. Drugim rečima, *zaliv izvršenja* predstavlja odnos između namere korisnika i mogućnosti sistema koje on može da izvrši. Drugi koncept, *zaliv ocenjivanja*, se odnosi na stepen reprezentovanja predmeta sistema koje se direktno mogu zaključiti iz njega, u kontekstu korisnikovih očekivanja i namera. Drugačije rečeno, *zaliv ocenjivanja* je težina ocenjivanja stanja sistema i koliko dobro predmeti iz sistema omogućavaju pronalaženje i interpretaciju svog stanja. Na oba koncepta se može primeniti ista analogija, zalivi su mali ako je „razdaljina“ od korisnikovih očekivanja i akcija potrebnih za izvršenje akcije mala u prvom slučaju, odnosno, ako se iz predmeta sistema lako mogu uočiti njegova stanja.

Kod modernih softverskih sistema, problem izrade korisničkog interfejsa predstavlja izbor platforme na kojoj će se softverski sistem koristiti, kao i izbor tehnologija u kome će se korisnički interfejs razvijati. Popularizacijom pretraživača, fokus sa desktop aplikacija se preneo na aplikacije u pretraživaču. Takođe, popularizacijom prenosivih uređaja, sada je potrebno da isti softver podržava i više platformi. U prilog ovome govori da je udeo korisnika mobilnih uređaja premašio udeo korisnika desktop uređaja u maju 2019. godine [2]. Kada govorimo o razvoju softvera u ovim slučajevima, najčešće će ovo značiti odvajanje kôda za svaku platformu koju softverski sistem podržava, a često može i značiti drugačije tehnike pri postavljanju i dostavljanju softvera do različitih klijentskih platformi. Ova segmentacija nije samo segmentacija u tehnologiji i tehnikama u izradi softverskog sistema, već to predstavlja i dodatni vremenski i finansijski trošak softverskih kuća koje razvijaju ovakve softvere.

2. CENTRALIZACIJA

Pri rešavanju ovog problema, glavnu ideju predstavlja centralizacija, odnosno, kreiranje jednog mesta u softverskom sistemu koji generiše korisničke interfejse. Ova tehnika je poznatija i pod imenom **SSR** – *Server-side-rendering* (generisanje interfejsa na serveru). Softverski sistem, opisan u ovom radu, će, dakle, generisati korisničke interfejse za više različitih platformi i pri tome će se oslanjati na nekoliko postojećih tehnologija koje su već ustaljene u praksi.

Prva je HTTP protokol, odnosno, softverski sistem je implementiran kao *http veb server*. Svaki pretraživač ili drugi *klijent* (entitet koji ima mogućnost slanja HTTP zahteva) na HTTP zahtev dodaje zaglavlja koje jedinstveno identifikuju operativni sistem klijenta, tačnije operativni sistem uređaja, verziju istog, pretraživač sa koga se zahtev šalje i slično. Veb server koristi ovu informaciju kako bi napravio diferencijaciju platformi i tu informaciju prosleđuje na dalje korišćenje u softverskom sistemu.

Druga tehnologija, ili skup tehnologija, na koju se softverski sistem oslanja jeste HTML, CSS i JavaScript, kao tehnologije koje velika većina uređaja danas podržava, a predstavlja rezultat generisanja korisničkog interfejsa samog softverskog sistema. Razlog za izbor ove tehnologije je već napomenuta rasprostranjenost ovih tehnologija kod današnjih uređaja, pa čak i kod uređaja koji su stariji. Većina operativnih sistema danas ima već integrisane aplikacije koje podržavaju prikazivanje sadržaja napisanih koristeći ove tri tehnologije. Za manjinu operativnih sistema koji ne podržavaju pregledanje ovakvog sadržaja, dovoljno je instalirati *oguljen* pretraživač (npr. Chromium) koji će onda moći prikazivati generisane interfejse.

Imajući ovo u vidu, centralizacija se odnosi na kreiranje jedinstvenog mesta, softverskog sistema, gde će se korisnički interfejsi za različite platforme konfigurisati kroz softverski sistem, a ne razvijati, a takođe i mesto gde će se isti korisnički interfejs generisati različito, u zavisnosti od platforme sa koje se zahtev šalje.

3. MODEL PODATAKA

Model podataka softverskog sistema logički je odvojen na dve vrste podataka, specifikacije (*Specification*) i instance (engl. Instances). Specifikacije se mogu razumeti kao tip ili vrsta određenog entiteta (npr. specifikacija validacije za obavezno polje – *RequiredValidationSpecification*), dok se instanca može zamisliti kao pojavljivanje objekta određenog tipa (npr. validacija elementa obaveznog polja – *RequiredElementValidation*). Po ovom principu su logički postavljeni tipovi grafičkih elemenata i pojavljivanje istih, validacije i pojavljivanje validacija, atributi i pojavljivanje atributa. Dodatno, postoji veza između specifikacije i instance određenog entiteta (npr. tipa validacije i instance validacije) tako da je pojavljivanje instance uvek slab objekat od specifikacije.

Da bi se definisao jedan takav softverski sistem, potrebno je ustanoviti definicije pojedinačnih komponenata na grafičkom interfejsu, definisati njihove mogućnosti i definisati njihov odnos sa drugim grafičkim elementima. Takođe je potrebno definisati njihov grafički izgled na različitim platformama. Olakšavajuća okolnost je što, kroz vreme dok grafički interfejsi postoje, su se već definisale određene grafičke komponente [3], kao i njihove mogućnosti i okvirni izgled, pa će ovaj softverski sistem pokrivati najčešće korišćene grafičke elemente, kao što su polja za unos, dugme, stranica, plutajući prozor, itd... Ovi grafički elementi će se prvo implementirati kao vrste grafičkih elemenata, odnosno grupa tipova pod nazivom *ElementSpecification*.

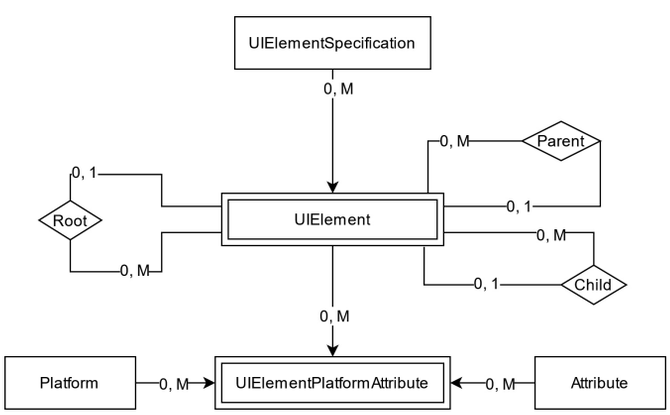
ElementSpecification predstavlja apstraktnu klasu koju će nasleđivati svi tipovi grafičkih elemenata. Kada ovakav entitet postoji u modelu, sa lakoćom se mogu dodati novi tipovi grafičkih elemenata i raditi sa postojećim tipovima, jednostavnim nasleđivanjem apstraktne klase.

Definisanje vrsta grafičkih elemenata nije dovoljno da bi se zahtevi softverskog sistema ispunili. Kako grafički interfejs predstavlja jedno *pojavljivanje* grafičkog elementa, ili skupa grafičkih elemenata određenog tipa, tako će i softverski sistem imati i pojavljivanja grafičkih elemenata, nazvanih *UIElements*. Drugim rečima, softverski sistem će imati i pojavljivanje (*instancu*) grafičkog elementa. Ova instanca (*UIElement*) se može i razumeti kao entitet koji će korisnici videti kao rezultat generisanja interfejsa. Analogno i specifikaciji (*ElementSpecification*), nasleđivanje *UIElement* abstraktne klase znači uvođenje novog tipa pojavljivanja grafičkog elementa u softverski sistem.

Kada imamo definisane pojedinačne grafičke elemente, potrebno je kreirati međuzavisnosti između samih grafičkih elemenata. Kako većina tehnologija, u kojima se danas kreiraju grafički interfejsi, samu strukturu interfejsa predstavlja kao *n*-arno stablo (HTML, XAML), tako će i grafički elementi u ovom softverskom sistemu imati identičnu strukturu. Ovo omogućava ugnježdavanje grafičkih elemenata, gde jedino ograničenje predstavlja količina informacija koja se na rezultujućem interfejsu generiše ili sama priroda grafičkog elementa na nekom čvoru u stablu (npr. element *Page* može imati podređene elemente, dok element *Button* ne može imati podređene).

Dodatno, definicije i instance grafičkih elemenata će nositi *svojstva* (Attributes), takođe po uzoru na HTML i CSS strukturu, gde će se svojstva, u domenu definicija elementa (*ElementSpecification*), tretirati kao podrazumevana svojstva grafičkih elemenata, a svojstva u domenu instanci (*UIElement*) će se tretirati kao svojstva specifična za određenu platformu nekog grafičkog elementa. Unija ova dva skupa atributa predstavlja skup svih atributa koji određeni grafički element nosi sa sobom, na određenoj platformi. U slučaju da i definicija grafičkog elementa i instanca grafičkog elementa nose isto svojstvo, primat vrednosti će uzeti svojstvo koje se nalazi na instanci grafičkog elementa, kako je ono specifičnije.

Na slici je prikazan uprošćeni dijagram modela koji opisuje odnos definicija i instanci grafičkih elemenata (Slika1).



Slika 1.

Veza pod nazivom *Parent* će imati podatak ka nadređenom elementu u *n*-arnom stablu, veza *Child* vezu ka podređenim elementima, dok će veza *Root* imati podatak ka korenom grafičkom elementu. Kada je element deo nekog stabla, vrednost

Root-a će postojati, a u slučaju da je element koreni element, vrednost će biti *null*. Ovako kreirane veze između grafičkih elemenata u potpunosti pokrivaju potrebne mogućnosti jednog modernog grafičkog interfejsa.

4. IMPLEMENTACIJA

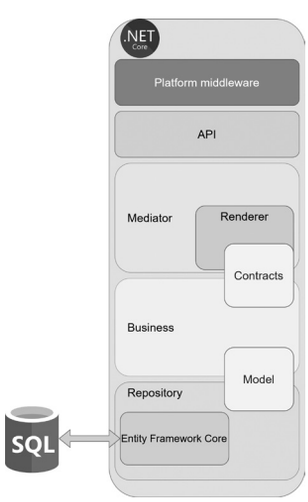
Kao tehnologija za implementaciju softverskog rešenja, odabran je programski jezik *C#*, tačnije, Microsoft-ov najnoviji okvir za razvoj veb aplikacija, ASP.NET Core. Microsoft svoje tehnologije vodi ka mogućnosti da se isti kôd može izvršavati na različitim operativnim sistemima, a ASP.NET Core predstavlja korak u napred u tom smislu, kako se on već sada može izvršavati na više operativnih sistema, pa u tom smislu daje dodatnu stavku zašto je izabran.

Dodatni argument za odabir ASP.NET Core-a je takođe i *Blazor renderer* koji predstavlja jednu veoma obećavajuću tehnologiju. Logički, Blazor model se takođe oslanja na komponente (grafičke elemente) i međusobno ugnježdavanje istih, pa odgovara modelu podataka ovog softverskog sistema. Mogućnost Blazor-a je to što on podržava i *WebAssembly*, novi koncept u programiranju veb aplikacija, gde se izvršni kôd, napisan u bilo kom jeziku, kompajlira u takozvani WebAssembly i kao takav izvršava na strani klijenta. Kako je WebAssembly skoro binarni kôd, njegovo izvršavanje je veoma brzo, skoro kao i izvršavanje *prirodnog* kôda napisanog za određeni operativni sistem [4].

Generatori interfejsa su implementirani na taj način da je moguće imati više različitih generatora interfejsa, koji se u zavisnosti od platforme pozivaju i generišu interfejs na svojstven način. Ovo je omogućeno korišćeći *DI* (Dependency injection) koji ASP.NET Core podržava. Na ovaj način, novi generatori interfejsa mogu lako biti implementirani u zasebnim projektima, nasleđujući samo odgovarajući interfejs.

Za sloj podataka, korišćena je takođe Microsoft tehnologija, *Entity Framework Core* kao i MS SQL baza podataka.

Na sledećem dijagramu je data arhitektura sistema koja ukratko opisuje slojeve u aplikaciji (Slika 2), a nakon toga i detaljniji opisi implementacija najbitnijih delova u softverskom sistemu:



Slika 2.

5. MIDLVER (MIDDLEWARE)

Middleware sloj softverskog sistema, na dijagramu nazvan *Platform middleware*, je baš kao što mu i ime sugeriše, sloj koji radi sa platformom korisnika. Kako je podrazumevano ponašanje *middleware*-a da se poziva na svakom zahtevu koji klijent uputi ka serveru, ovo je idealno mesto kako bi se odredila platforma korisnika. Kako svaki HTTP zahtev ima podrazumevane *header*-e, softverski sistem će se oslanjati na jedan od njih, tačnije *User-Agent* header i njegove vrednosti [5]. Svaki zahtev iz pretraživača će sadržati vrednost ovog header-a, na osnovu koje će softverski sistem odrediti ko, odnosno, koji operativni sistem, pretraživač i slično, šalje zahtev. Za samo razvrstavanje vrednosti u *header*-u, softverski sistem koristi biblioteku pod nazivom *DeviceDetector*, koja već pruža ove mogućnosti prepoznavanja vrednosti datog header-a. Na osnovu vraćenog rezultata biblioteke, softverski sistem onda proverava da li je, u bazi podataka, podržana data platforma jednostavim upitom po imenu operativnog sistema.

U sledećem delu kôda je prikazano korišćenje biblioteke pri detekciji platforme:

```
var device = new DeviceDetector(context.Request.Headers["User-Agent"].
ToString());
device.Parse();
```

Parsiran rezultat, ako je uspešan, se onda koristi za upoređivanje podržanih platformi u softverskom sistemu:

```
var dbPlatform = repository.GetQueryable<Model.Platform.Platform>().
AsNoTracking()
            .FirstOrDefault(x => x.OSName == device.GetOs().Match.
Name && x.PlatformName == device.GetOs().Match.Name);
          if (dbPlatform == null)
          {
             _platformProvider.SetPlatform(new PlatformContract()
             {
                OSName = osInfo.Match.Name,
                PlatformName = osInfo.Match.Platform,
                Name = deviceName
             });
          }
          else
          {
             _platformProvider.SetPlatform(_mapper.Map<PlatformContra-
ct>(dbPlatform));
          }
```

Ako je platforma podržana, podatak o platformi se čuva u servisu, koji je registrovan u *Dependency injection* kontejneru, kao servis koji se inicijalizuje svakim zahtevom (*scoped service*). Ovako upisana informacija o platformi je dostupna bilo kom servisu u softverskom sistemu, dok traje procesiranje datog zahteva u softverskom sistemu.

6. API I MEDIATOR SLOJ

U API sloju softverskog sistema su definisani *endpoint*-i do kojih klijent može da pošalje zahteve, koristeći mediator servis. Na kontroleru (API-u) su definisane dve metode, jedna za potrebe generisanja interfejsa i jedna za potrebe validacije podataka. Prva nosi naziv *View*, u kontroleru nazvanom *View*, a definisana je na taj način da kao parametre može da prihvati

ili ime, tipa *string*, ili id tipa *long.* Ovde treba napomenuti da elementi tipa *UIElement* nasleđuju *NamedObject*, koji u sebi nosi svojstvo *Name* i *Id.* Pri generisanju željenog interfejsa, moguće je generisati interfejs od korenog elementa, koristeći njegovo ime (*Name*) ili *Id*. Iz tog razloga, kontroler prihvata ili ime ili id. Softverski sistem je tako organizovan da klase koje implemetiraju sam model podataka nisu vidljive na API sloju. Iz tog razloga postoje određene DTO klase (u softverskom sistemu nazvani ugovori, ili *Contracts*) koji predstavljaju vezu sa konkretnim klasama u samom modelu podataka. Pored toga, kako jedna API metoda ima svoj ulaz i izlaz, analogno postoje DTO klase, ugovori, kao ulaz u endpoint i izlaz, odnosno rezultat iz endpoint-a.

Konkretna implementacija u View kontroleru za metodu koja generiše interfejse je:

```
[HttpGet]
[Route("{name}")]
 public async Task<IActionResult> View([FromQuery] DefaultContract contract, string name)
      {
         contract.Name = name;
   return await _mediator.SendQueryAsync<DefaultContract, DefaultResponse>(HttpContext.Request, contract);
      }
```

Još jedan razlog postojanja ugovora za ulaz i izlaz je ta što je *Mediator* sloj implementiran koristeći CQS (*command-query segregation*) patern. U ovoj implementaciji, mediator koristi tipove ugovora za ulaz i izlaz kako bi u biznis sloju pronašao odgovarajuću klasu i izvršio njenu logiku u zavisnosti od datih ulazno/izlaznih tipova. Mediator ove klase pronalazi koristeći *dependency injection*, jer su prethodno, pri inicijalizaciji softverskog sistema, date klase iz biznis sloja registrovane pod datim ulaz/izlaz tipovima. CQS patern je korišćen kako bi se napravila logička separacija delova koda čiji je zadatak samo čitanje podataka, i delova koda čiji je zadatak upisivanje i čuvanje podataka [6].

Sem što prosleđuje podatke do konkretne klase u biznis sloju, mediator može, nakon uspešnog generisanja strukture korisničkog interfejsa, pozvati i samo generisanje interfejsa u HTML jezik koristeći implementiran *renderer*. Na osnovu tipa prosleđenog odgovora i na osnovu podatka iz *ContentType* zaglavlja samog HTTP zahteva, rezultat se prosleđuje na rendering, ako je *ContentType* zahteva jednak „text/html".

7. BIZNIS SLOJ

U biznis sloju se nalaze implementirane klase koje izvršavaju samu logiku softverskog sistema, odnosno konstruišu zahtevanu strukturu korisničkog interfejsa, primenjuju podatak o platformi kako bi modifikovali samu strukturu i atribute unutar konstruisanog korisničkog interfejsa i validiraju podatke koje je korisnik uneo na generisanom korisničkom interfejsu.

Klase u biznis sloju, kao što je spomenuto, prate CQS patern, pa tako implementiraju ili *IQueryHandler* ili *ICommandHandler* (u softverskom sistemu su implementirane samo *IQueryHandler* klase kako sistem nema klasa koji obavljaju posao komandi). Ovaj interfejs u sebi definiše *Handle* metodu, koju mediator sloj poziva, a u njoj se nalazi sama logika koja se izvršava u konkretnoj klasi koja implementira dati interfejs u biznis sloju

Tako se u klasi koja je zadužena za generisanje strukture korisničkog interfejsa, nazvana „GetUIElementsByIdOrName", koja implementira *IQueryHandler*, u *Handle* metodi nalazi logika za čitanje postojećih *UIElemenata* definisanih u datom interfejsu, njihovo raspoređivanje u n-arno stablo koristeći podatak o nadređenom elementu i njihovo raspoređivanje u stablu na istom nivou, koristeći *Order* svojstvo elemenata. Nakon njihovog raspoređivanja, sledi mapiranje u DTO klase (podsetimo se, kao ulaz i izlaz, softverski sistem ima samo klase koji su ugovori, ne same klase domenskog modela) i vraćanje rezultata nazad, u mediator sloj. Primer kôda koji čita sve elemenate koji pripadaju jednom stablu korisničkog interfejsa je:

```
private List<UIElement> GetElementsBy(long? id, string name)
    {
       var elementQ = _repository.GetQueryable<UIElement>()
          .Include(x => x.ElementSpecification)
          .Include(x => x.Attributes).ThenInclude(x => x.Attribute).AsNoTracking();
       var elements = (from root in elementQ
                  where ((root.Id == id) || (root.Name == name)) && root.ElementSpecification.CanBeParent == true
                  select new
                  {
                     Root = root,
                     Childs = (from node in elementQ
                           where node.RootElementId == root.Id
                           select node).ToList()
                  }).FirstOrDefault();
       var result = new List<UIElement>(elements.Childs);
       result.Add(elements.Root);
       return result; }
```

Metoda koja raspoređuje elemente u jednom korisničkom interfejsu je data u sledećem primeru:

```
private UIElement ArrangeElements(UIElement root, List<UIElement> elements)
    {
       if (root == null)
          root = elements.Single(x => x.ParentId == null);
       var leafs = elements.Where(x => x.ParentId == root.Id).OrderBy(x => x.Order).ToList();
       for (int i = 0; i < leafs.Count(); i++)
       {
          leafs[i] = ArrangeElements(leafs[i], elements);
       }
       root.Childs = new Collection<UIElement>(leafs);
       return root;
    }
```

Klasa koja je zadužena za validaciju podataka i implementirana logika validacije nosi ime „ValidateFormQueryHandler". U modelu podataka, validacije su definisane na taj način da za jedan element korisničkog interfejsa mogu biti povezane više validacija, ako to vrsta grafičkog elementa dozvoljava. Takođe, model podataka validacija je sličan modelu podataka

grafičkih elemenata. On ima vrste validacija (*specification*) i same implementacije validacija sa konkretnim vrednostima i ograničenjima. Ove implementacije se nalaze u vezi sa samim grafičkim elementima i pomoću njih se određuje koji grafički element ima koje validacije.

Tako se, u listi elemenata jednog korisničkog interfejsa, pretražuju elementi sa validacijama, a onda se te validacije izvršavaju. Apstraktna klasa *ElementValidation* u sebi ima apstraktnu metodu *Validate* koja je implementirana u konkretnim tipovima validacije (*RequiredElementValidation* – validacija obaveznih polja, *RangeElementValidation* – validacija dozvoljenog opsega vrednosti, itd...). Sama implementacija validiranja pojedinačnih podataka nije komplikovana, ali način dolaska do podatka koji se validira je interesantniji i on će biti objašnjen u sledećem poglavlju.

8. VALIDACIJE

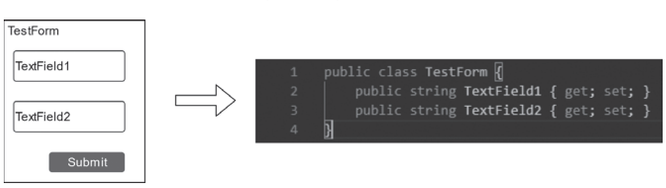
Problem koji proizilazi iz generisanih interfejsa jeste validacija podataka, kada korisnički interfejs sadrži polja za unos podataka. Ovakvi podaci se ne mogu validirati na konvencionalne načine (konvencionalni način podrazumeva klasu koja nosi podatke koji se validiraju i definisanje validacione logike nad poljima koji se validiraju u datoj klasi). Srećom, ovaj softverski sistem ima mogućnost i validacije podataka koji su generisani na interfejsu.

Kada posmatramo podatke koji treba da se validiraju, najčešće će oni biti u *JSON* formatu. Kako u definisanom modelu podataka ne postoji ni jedna klasa koja odgovara datom *JSON* modelu, biće potrebno definisati privremene klase u ovu svrhu.

Za ovu potrebu, softverski sistem implementira „dinamične tipove podataka“, koji se čuvaju u memoriji, tačnije keš-u (cache), na osnovu definisanih korisničkih interfejsa. Biblioteka koja se u ovu svrhu koristi je *System.Reflection* biblioteka, koja ima mogućnost kreiranja tipova u vreme izvršavanja kôda. Takođe, biblioteka, pored kreiranja tipova, može kreirati i atribute i svojstva u tim tipovima.

Za strukturu ovih dinamičnih tipova, upotrebiće se postojeća, definisana, struktura grafičkog interfejsa. Tačnije, kao zaglavlje, odnosno ime dinamičnog tipa, će se iskoristiti pojavljivanje *Form* grafičkog elementa, a za svojstva dinamičnog tipa će se iskoristiti elementi koji su direktni naslednici *Form* grafičkog elementa, u nekom konkretnom interfejsu.

Kreiranje ovakvog dinamičnog tipa podatka se može predstaviti sledećom slikom (Slika3):



Slika 3.

Pa tako, za konkretnu instancu *Form* grafičkog elementa, njegov tip kreiramo sledećom linijom:

TypeBuilder typeBuilder = _moduleBuilder.DefineType(typeName, System.Reflection.TypeAttributes.Public),

svojstvo u tipu za svako polje u Form elementu sledećom linijom:

typeBuilder.DefineField(typeInfo.Item1, typeInfo.Item2, System.Reflection.FieldAttributes.Public);

a na kraju i čitav tip, koji se inicijalizuje sledećom linijom:

typeBuilder.CreateType(),

Ovako kreirani tipovi se čuvaju u memorijskom kešu (cache), kako se instance objekata tipa *Type* ne mogu serijalizovati bez dodatnih modifikacija. Keširanje takođe doprinosi i performansama, kako se pri validaciji ne bi ponovo kreirali tipovi na osnovu definicije interfejsa.

Kada je potrebna validacija, zahtev koji sadrži *JSON* sa podacima unetih kroz grafički interfejs, odgovaraće strukturi datog dinamičnog tipa po imenima polja, a i po tipu podatka. Na ovaj način, kada sa jedne strane imamo *JSON* sa podacima, a sa druge strane tip (instanca klase *Type*), može se iskoristiti Newtonsoft.Json biblioteka za konverziju iz *JSON*-a u instancu date klase. Pa tako, podatke koje se trebaju validirati, dobijamo izvršavanjem sledeće linije kôda:

var objectValue = JsonConvert.DeserializeObject(queryMessage.QueryParams.ObjectData, objectType);

gde su *ObjectData* podaci u *JSON* formatu, dobijeni sa generisanog interfejsa, a *objectType* dinamični tip forme koja se validira.

Dalje se za svaku validaciju nad konkretnim poljem sa grafičkog interfejsa izvršavaju validacije, tako što se iz liste grafičkih elemenata čita ime elementa. Ime elementa se koristi kako bi se iz instance objekta (*objectValue*) dobila vrednost odgovarajućeg svojstva i sa tom vrednošću se poziva konkretna Validate metoda, koja validira sam podatak.

Nakon izvršenih validacija, lista validacionih grešaka se vraća klijentu, gde se validacione greške prikazuju ispod elemenata korisničkog interfejsa, kada su nevalidne, ili se ne vraća greška, ukoliko je validacija podataka zadovoljena.

9. RENDERER

U softverskom sistemu je implementiran i jedan primer *renderer*-a. U pitanju je HTML *renderer*, odnosno rezultat izvršavanja ovog kôda je HTML strana koju pretraživač može prikazati. Kako je spomenuto, kada zahtev za korisničkim interfejsom, u zaglavlju *ContentType* ima vrednost „text/html“, mediator će pozvati implementirani *renderer*. Kako bi *renderer* generisao individualne komponente, one moraju biti definisane. U softverskom sistemu, grafičke komponente su definisane kao *Razor* komponente, odnosno, implementirane u *Razor* jeziku, a *renderer*, analogno tome, koristi istu tehniku renderovanja komponenata kao i klasična ASP NET MVC aplikacija, prilagođena nameni softverskog sistema.

Princip iscrtavanja komponenti je takav da se one iscrtavaju koristeći *Html.RenderPartial* metodu, implementiranu u sa-

mom *AspNetCore.Mvc* okviru. Metoda, po prosleđenom imenu komponente, pronalazi parcijalni pregled (*Partial view*) same komponente i iscrtava je. Ime parcijalnih pregleda komponenti je usko povezano sa imenima u klasama ugovora koji se propagiraju do renderer-a. Tačnije, domenska klasa *Page*, će se nakon generisanja interfejsa u biznis sloju mapirati u klasu tipa *PageResponse*. *PageResponse* u sebi ima svojstvo *PartialName*, koje je već popunjeno prilikom mapiranja, koje se koristi kao ime samog parcijalnog pregleda.

Nakon iscrtavanja svih komponenti grafičkog interfejsa, koristeći strukturu konkretnog korisničkog interfejsa, rezultat iscrtavanja se prosleđuje mediatoru, odnosno, klijentu. Ovako generisan korisnički interfejs je spreman za prikazivanje u pretraživaču korisnika, bez dodatnih biblioteka, okvira ili slično.

Zbog ove mogućnosti, ovaj softverski sistem može svrstati u SSR (Server-Side-Rendering) softvere. Implementacija preko *Razor* jezika je odabrana jer noviji, *Blazor*, renderer je još uvek u razvojnoj fazi, pa kao takav, u trenutku implementacije još u potpunosti ne podržava SSR [7]. Još jedan razlog za odabir starije tehnologije je način na koji sam Blazor funkcioniše (Server-side Blazor). On podrazumeva održavanje konekcije između klijentske aplikacije i veb servera, pomoću WebSocket-a, preko koga sam okvir prati promene koje korisnik izvršava na samom korisničkom interfejsu. Promene se onda pomoću konekcije razmenjuju između klijenta i servera, a server, prilikom svake promene na interfejsu, generiše „razliku“ korisničkog interfejsa, pre korisnikove akcije i nakon akcije. Nakon toga klijentski deo Blazor-a primenjuje datu razliku na samom korisničkom interfejsu. Ovakav pristup bi značajno povećao opterećenje samog softverskog sistema, pogotovo zbog načina na koji su u softverskom sistemu definisani korisnički interfejsi.

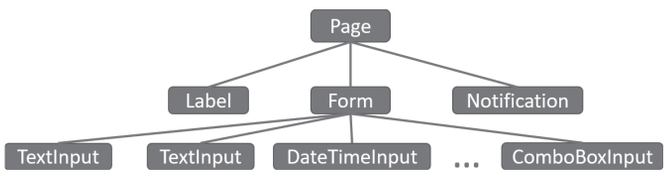
Po potrebi, komponente napisane u Razor jeziku se lako mogu prepisati u Blazor komponente, jer je logički model iza obe vrste komponenata isti.

Rezultat ovog softverskog sistema može biti i formatiran u JSON formatu, kako bi se mogao upotrebiti na bilo koji razvojni okvir koji ne podržava SSR tehnologiju ili JIT kompajliranje.

10. KORIŠĆENJE SOFTVERSKOG SISTEMA

Samo funkcionisanje softverskog sistema najbolje je prikazati na konkretnom primeru, pa je u ovu svrhu opisano korišćenje softverskog sistema u simuliranom sistemu banke, pri prijavi klijenta za zajam. Klijent će dolaziti na veb stranicu banke za prijavu za zajam. Na toj stranici, centralni deo interfejsa, gde će se nalaziti polja za unos podataka, će biti generisan od strane softverskog sistema za generisanje korisničkih interfejsa. Deo generisanog interfejsa će te podatke validirati, a nakon toga, slati nazad infrastrukturi banke na čuvanje.

Struktura korisničkog interfejsa koji će se koristiti u ovom primeru će biti konfigurisana kao jedno n-arno stablo, sa elementom *Page* na vrhu kao koreni element. Unutar *Page* elementa, nalaziće se komponenta *Label*, *Form* i *Notification*. Komponente Label i Notification dalje neće imati podređene elemente, ali komponenta *Form* će imati, i to komponente u kojima će korisnik unositi podatke na samom korisničkom interfejsu. Tako će u komponenti *Form* biti više komponenata vrste *TextInput*, *DateTime* input, *ComboBoxInput* itd... Ove komponente za unos će odgovarati podacima poput „Ime“, „Prezime“, „JMBG“, „Tip lica“ i slično, koji će se čuvati u softverskom sistemu banke. Struktura konfigurisanog korisničkog interfejsa može se prikazati i sledećom slikom (Slika4):

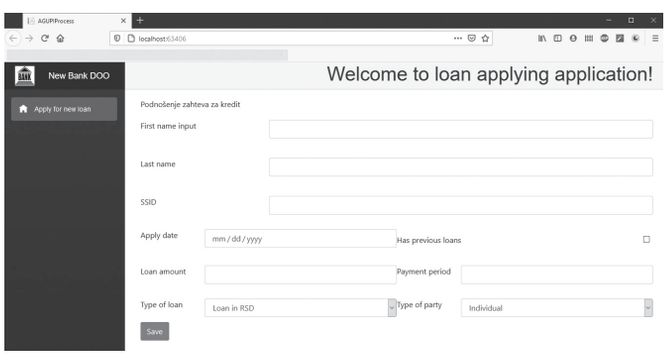


Slika 4.

Kao što je već napomenuto, kako bi dobili ovaj generisan interfejs, dovoljno je poslati zahtev softverskom sistemu sa imenom ili id-em korenog elementa, u ovom slučaju Page elementa, ovde nazvanog „ApplyForLoan“. Da bi ovaj generisani interfejs iskoristili na stranici koju koristi banka, iskoristićemo HTML element <iframe>, tačnije deo kôda kao u sledećem primeru:

```
<iframe src=”https://generatorinterfejsa/view/ApplyForLoan”></iframe>
```

Kada je sve konfigurisano i spremno za korišćenje, korisnik dolaskom na početnu stranicu banke, dobija interfejs u kome je deo kreirao softverski sistem banke, a deo softverski sistem za generisanje korisničkih interfejsa (Slika5). Korisnik onda može popuniti podatke i validirati ih. Nakon uspešne validacije podaci se čuvaju u softverski sistem banke. Treba napomenuti da je generisana komponenta *Form* konfigurisana tako da, nakon uspešne validacije i čuvanja podataka, pozove komonentu *Notification*, generisanu na istom interfejsu i aktivira njeno prikazivanje. Dakle, generisani korisnički interfejs ima i primitivne mogućnosti uvezivanja komponenti na samom generisanom korisničkom interfejsu.



Slika 5.

Ovaj konfigurisani interfejs je takođe konfigurisan za android uređaje, pa otvaranjem stranice sa nekog android uređaja, prikaz grafičkih komponenti bi se promenio, kako bi se prilagodio drugačijim dimenzijama samog uređaja i drugačijim konfiguracijama grafičkih elemenata.

11. ZAKLJUČAK

Ovakav softverski sistem može imati mnogo mesta primena. Od prostih prezentacionih veb strana, preko softvera za upitnike, sisteme za automatizaciju poslovnih procesa i slično [8]. Primene nisu ograničene na samo ove slučajeve. Model podataka na kome je postavljen ovaj softverski sistem je veoma prilagodiv i lako konfigurabilan u različite svrhe [9].

Glavni cilj ovog softverskog sistema je preseliti kreiranje grafičkih interfejsa sa programiranja na konfiguraciju istih, kao i olakšati njihovu integraciju sa već postojećim softverskim rešenjima.

U prilog softverskom sistemu ide i da se broj podržanih platformi, koje ovaj softverski sistem potržava, može po potrebi proširiti. Tako, kreiranjem neke nove platforme, ovaj softverski sistem se može, uz minimalne izmene, veoma brzo prilagoditi i konfigurisati.

Pored toga što se softverski sisetm može primeniti na veliki broj poslovnih slučajeva korišćenja, takođe se može i prilagoditi velikom broju biblioteka za razvoj veb aplikacija, kao što stu *Bootstrap*, *jQuery*, *Vue.js* [10], *React*, *Angular* [11] i slične biblioteke, koje podržavaju SSR. Imajući ovo u vidu, ovaj softverski sistem ima veoma veliku prednost u primeni u sistemima gde je korisnički interfejs podložan čestim promenama, što bi značilo da promene korisničkih interfejsa mogu da se dešavaju dok softverski sistem radi, bez potrebe objavljivanja nove verzije interfejsa, nove verzije biblioteki ili ažuriranja klijentskih aplikacija na raznim platformama ili slično.

LITERATURA

[1] D. Norman, The Design of Everyday Things, Basic Books, 1988.

[2] GlobalStats, “Desktop vs Mobile vs Tablet Market Share,” 2020. [Online]. Available: https://gs.statcounter.com/platform-market-share/desktop-mobile-tablet/worldwide/2019.

[3] Usability.gov, “User Interface Elements,” [Online]. Available: https://www.usability.gov/how-to-and-tools/methods/user-interface-elements.html.

[4] MDN Contributors, “WebAssembly Concepts,” 2019. [Online]. Available: https://developer.mozilla.org/en-US/docs/WebAssembly/Concepts.

[5] MDN contributors, “User-Agent,” Mozilla, 2020. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/User-Agent.

[6] K. Henney, “A tale of two patterns,” *Java Report 5.12,* pp. 84-88, 2000.

[7] D. Roth, “ASP.NET Core Blazor hosting models,” Microsoft, 2020. [Online]. Available: https://docs.microsoft.com/en-us/aspnet/core/blazor/hosting-models?view=aspnetcore-3.1.

[8] S. J. S. B. Jaap Kabbedijk, “A case study of the variability consequences of the CQRS pattern in online business software,” 2012. [Online].

[9] D. Jevtić, d. S. Lazarević and T. Stojanović, “Implementacija softverskog sistema za generisanje cross-platform korisničkog interfejsa,” in *Simpozijum o operacionim istraživanjima*, Beograd, 2020.

[10] Google, “Server-side rendering (SSR) with Angular Universal,” Google, 2020. [Online]. Available: https://angular.io/guide/universal.

[11] Vue.js, “Server-Side Rendering,” Vue.js, 2020. [Online]. Available: https://v3.vuejs.org/guide/ssr.html.

Dimitrije Jevtić
Kontakt: dimitrije.jevtic@ssoe.rs
Oblast interesovanja: softversko inženjerstvo, informacioni sistemi, .NET okruženje

Prof. Dr Saša D. Lazarević
Kontakt: lazarevic.sasa@fon.bg.ac.rs
Oblast interesovanja: softversko inženjerstvo, informacioni sistemi, baze podataka, sistemi za upravljanje dokumentacijom, .NET platforma

Tatjana Stojanović
Kontakt: tatjana.stojanovic@fon.bg.ac.rs
Oblast interesovanja: cross-platform development, .NET Core, distribuirane arhitekture, mikroservisi

CIP – Каталогизација у публикацији Народна библиотека Србије, Београд 659.25:004
INFO M : časopis za informacione tehnologije i multimedijalne sisteme = journal of Information technology and multimedia systems / glavni i odgovorni urednik Miroslav Minović. - [Štampano izd.]. - God. 1, br. 1 (2002)- . - Beograd : Fakultet organizacionih nauka, 2002- (Smederevo : Newpress). - 30 cm
Dva puta godišnje. - Je nastavak: Info Science = ISSN 1450-6254. - Drugo izdanje na drugom medijumu: Info M (Online) = ISSN 2683-3646
ISSN 1451-4397 = Info M (Štampano izd.)
COBISS.SR-ID 105690636