Komparativna analiza suvremenih web tehnologija

Ledinski, Robert

SVEUČILIŠTE U ZAGREBU

FAKULTET PROMETNIH ZNANOSTI

Robert Ledinski

KOMPARATIVNA ANALIZA SUVREMENIH WEB TEHNOLOGIJA

ZAVRŠNI RAD

Zagreb, 2017.

Sveučilište u Zagrebu

Fakultet prometnih znanosti

ZAVRŠNI RAD

KOMPARATIVNA ANALIZA SUVREMENIH WEB TEHNOLOGIJA

Mentor:

Doc dr. sc. Marko Periša

Student:

Robert Ledinski, 0135220932

Zagreb, 2017.

SADRŽAJ

1. UVOD

Razvojem tehnologije javljaju se inovacije u svim sektorima, osobito u IT (engl. *Information Technology*) sektoru. Takav razvoj utječe na informacijsko komunikacije sustave. Informacijsko komunikacijskim sustavima riješeni su problemi organizacije i komunikacije područja za koje su oni namijenjeni. Informacijsko komunikacijski sustavi se razlikuju po načinu rada, načinu izrade, korištenim metodologijama, funkciji za koju su namijenjeni, tehnologijama kojim su napravljeni te u kojem su vremenskom razdoblju rađeni.

Iz toga je vidljivo da je kompleksno a i nepoželjno reproducirati neki postojeći sustav zbog toga što ga je moguće kvalitetnije odraditi novim tehnologijama i novim metodologijama odnosno pristupima čime se povećavaju kako njegove mogućnosti tako i brzina rada iako nisu nužno vezani jedno za drugo.

Završni rad podijeljen je u 6 cjelina:

1. Uvod
2. Razvoj web tehnologija informacijsko-komunikacijskih sustava
3. Komparativna analiza suvremenih *web* tehnologija Student Projekt aplikacije
4. Arhitektura Student Projekt sustava primjenom razvojnih web tehnologija
5. Analiza funkcionalnosti korištenjem injekcija ovisnosti
6. Zaključak

U drugom poglavlju objašnjeni su informacijsko komunikacijski sustavi, razvoj web tehnologija informacijsko komunikacijskih sustava, povijest razvoja razvojnih tehnologije prije i nakon izuma Interneta.

U trećem poglavlju izvršiti će se komparativna analiza suvremenih web tehnologija koje su korištene u izradi Student Projekt sustava te njihova podjela na *frontend* i *backend*.

Četvrto poglavlje prikazuje i opisuje arhitekturu Student Projekt sustava te objašnjava važnost planiranja arhitekture.

Peto poglavlje opisuje injekciju ovisnosti i inverziju kontrole te prikazuje analizu rada aplikacije prije i poslije korištenja injekcije ovisnosti.

2. RAZVOJ WEB TEHNOLOGIJA INFORMACIJSKO-KOMUNIKACIJSKIH SUSTAVA

Informacijsko komunikacijski sustav jest skup tehnologija koji se sastoji od komunikacijske i mrežne opreme, hardvera i softvera zajedno povezanih u jednu cjelinu.

ICT jest naziv koji se koristi za informacijsko komunikacijsku tehnologiju. *ICT* je orijentiran na ujedinjenu komunikaciju između različitih komunikacijskih uređaja u jednu cjelinu čime se izgrađuje sustav. Sustav se sastoji od više podsustava. [10]

Prikaz povezane komunikacije vidljiv je na slici 1.

Slika 1: Cilj ICT-a [17]

Web (engl. *World Wide Web*) tehnologije informacijsko-komunikacijskih sustava mogu se gledati kroz dva dijela:

- Povijest razvojnih web tehnologija prije izuma Interneta i
- Suvremene web tehnologije.

U razvoju aplikacija postoje dva glavna dijela razvoja. Razvojni inženjeri raspravljaju od službenoj podijeli razvoja na dijela razvojna dijela: *frontend* i *backend*. [7]

Frontend predstavlja klijentski dio aplikacije koji omogućuje interakciju sa korisnikom, dok *backend* predstavlja serverski dio aplikacije koji komunicira sa bazom podataka na način da klijentsku akciju prosljeđuje na server.

Prvi informacijsko-komunikacijski sustavi bili su bazirani na *backendu* sve do izuma Interneta kada se počeo razvijati *frontend*.

Upravljani su sa softverom koji je izgledao poput terminala. Sustav se upravljao različitim naredbama od kojih je svaka imala svoje značenje. Primjer takvog izgleda vidljiv je na slici 2. Do razvoja Interneta koristilo se komandno sučelje te kao takvo nije imalo mogućnosti upravljanja se klijentske strane već se upravljalo izvršavanjem naredbi. Razvojem Interneta to se mijenja jer je tada izumljen HTML (engl *HyperText Markup Language*).

Razvojne web tehnologije koriste se za razvoj *web* stranica i *web* aplikacija. *Web* stranica je najčešće dokument na određenom serveru koji na sebi ima tvrdo zapisane podatke kojima se ne može dinamički upravljati. Web aplikacija može biti jednostavna aplikacija sa jednom stranicom koja se sastoji od tvrdo zapisanog teksta pa sve do kompleksnih poslovnih sustava poput bankovnih. Takvi kompleksni poslovni sustavni zovu se CMS-ovi(engl. *Content Managment System*) zbog toga što je na njima moguće upravljanje sadržajem. [8] Prilikom izrade aplikacija potrebno je koristiti različite metodologije ovisno o veličini i vrsti projekta u svrhu smanjenja vremena i cijene razvoja. Za ubrzan razvoj preporučeno je koristiti gotove dodatke (engl. *Libary*).

2.1 POVIJEST RAZVOJNIH TEHNOLOGIJA PRIJE IZUMA INTERNETA

Razvojne tehnologije su se počele javljati razvojem programskih jezika. Programski jezici se prikazuju kao set gramatičkih pravila i instrukcija računalu kojima ono izvršava određene zadatke. Programski se jezici razlikuju po sintaksi i setu gramatičkih pravila. Viši programski jezici izgledaju poput ljudskog govora dok se niži programski jezici poput računalnog jezika, koje koristi procesor (CPU engl. *Central Processing Unit*), sastoje se od niza bitova, jedinica i nula koji nisu čitljivi ljudskom oku. Različite vrste procesora imaju različite računalne jezike. Programski jezici koji se nalaze iznad računalnih i ispod viših programski jezika su asemblerski jezici. Oni su čitljiviji od računalnog jezika međutim nisu čitljivi kao viši programski jezici. U sebi sadrže skupove bitova, nula i jedinica međutim oni omogućuju razvojnom inženjeru da zamijeni imena za brojeve, a to je bitno jer se računalni jezik sastoji samo od brojeva. [14]

Takav prikaz vidljiv je na slici 2.

Slika 2: Arhitektura programskih jezika [3]

Iznad viših programskih jezika nalazi se četvrta generacija programskih jezika čiji je izgled najsličniji ljudskom jeziku i oni su najudaljeniji od računalnog jezika. Neovisno koji se programski jezik koristi, da bi se on mogao izvršiti potrebno je pretvoriti programski jezik u računalni jezik i to se radi na dva načina:

- Kompajliranje – omogućuje prevođenje viših programskih jezika u računalni jezik i obrnuto i
- Interpretiranje – daje značenje programskim jezicima. [14]

2.2 SUVREMENE RAZVOJE TEHNOLOGIJE

WWW je nastao početkom 1980-ih godina. Suvremene razvojne tehnologije započele su razvoj nastankom Interneta. Nastankom Interneta javila se potreba za uređivanjem *web* stranica te shodno tome i načini te tehnologije kojima se to uređivanje može vršiti.

Sastoji se od tri glavna protokola:

- HTML (engl. *Hypertext markup language*);
- HTTP (engl. *Hypertext Transfer Protocol*) i
- URL (engl. *Uniform resource locator*).

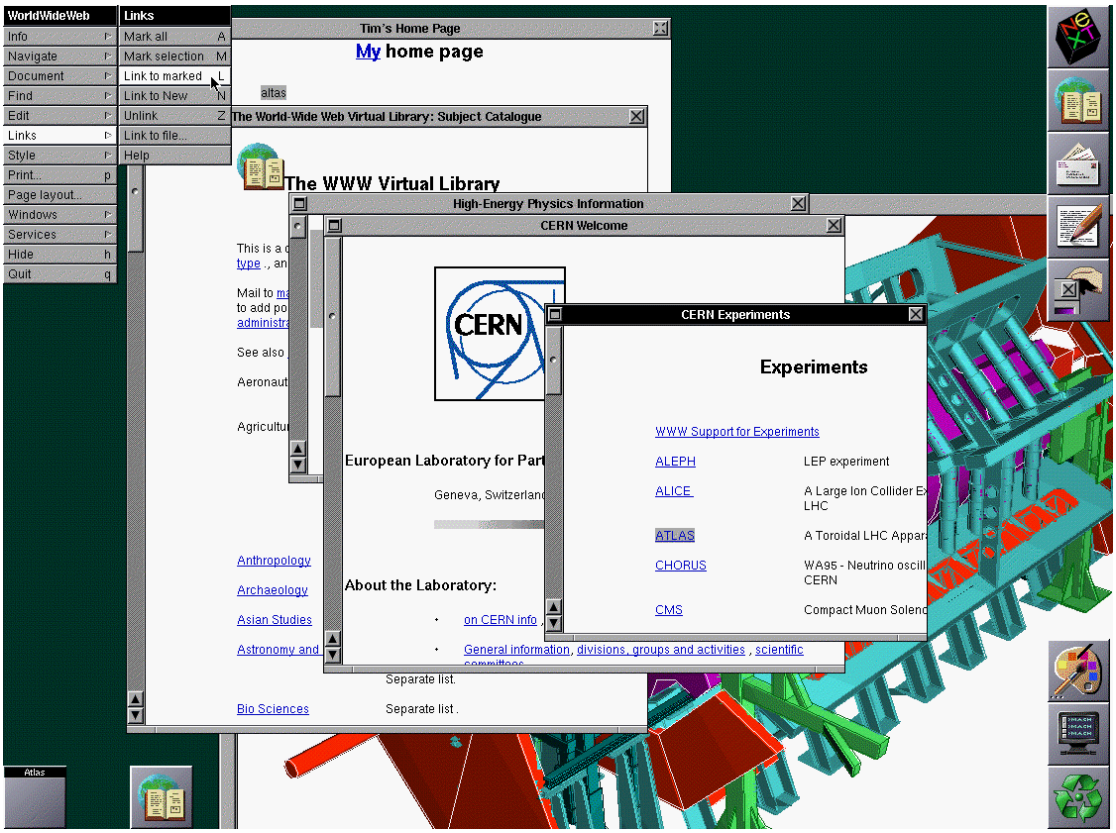
HTML protokol je omogućio uređivanje *web* stranica, HTTP protokol je omogućio komunikaciju sa stranicom dok je sve to prikazivao *web* pretraživač. Ime prvog *web* pretraživača bilo je „*the World Wide Web*“.

U idućih pet godina dogodio se nagli rast pretraživača, neki od njih su bili:

- *Line Mode* pretraživač – ujedno je bio i prvi pretraživač koji je podržavao više platformi.
- *Viola WWW* pretraživač - podržavao je uređivanje sadržaja i skriptni jezik 3 godine prije predstavljanja *JavaScripta* i 5 godina prije predstavljanja *CSS*-aa.
- *Mosaic* pretraživač – razvijen na sveučilištu u *Illinois*
- *Cello* pretraživač – prvi omogućio podršku za *Windows*
- *Netscape Navigator* 1.1 – prvi omogućio tablice u *HTML*-u
- *Opera* 1.0
- *Internet Explorer* 1.0 – pretraživač koji je radio samo na operativnom sustavu *Windows* 95.

Nastankom *WWW*-a sve web stranice bile su tekstualni dokumenti. Trenutne mogućnosti web stranica omogućio je *JavaScript*. *JavaScript* je objektno orijentirani programski jezik koji omogućuje interaktivne efekte na web stranicama. Pod te mogućnosti spadaju animacije te interaktivne izmjene elemenata. [6]

Prikaz prvog *web* pretraživača vidljiv je na slici 3.



Slika 3: *World Wide Web* pretraživač [3]

Razvoj Interneta omogućio je dijeljenje podataka preko mreže te komunikacije sa udaljenim serverima. Takav način komunikacije i dijeljenja resursa omogućio je korištenje i izmjenu pojedinih dijelova podataka ovisno o dozvolama. Isprva je svatko mogao mijenjati sadržaj svih *web* stranica. [6]

2.2.1 BACKEND

Prije izuma Interneta razvoje tehnologije su bile fokusirane na razvoj korištenjem komandne linije koja im je omogućavala izvršavanje komandi. Takva mogućnost izvršavanja operacija na računalu u IT svijetu nazvana je *backend* razvoj. *Backend* razvoj se razlikuje od *frontend* razvoja zbog toga što nema kontrolu nad interaktivnim elementima već barata podacima i prosljeđuje ih ovisno o načinu rada. Razvijenjem *backend*-a razvojni inženjer osigurava mogućnost toka podataka u oba smjera te praćenje izmjena podataka i komunikaciju sa bazom podataka.

Backend razvoj se još naziva i razvoj na serverskoj strani zbog toga što sve što se događa u aplikaciji, svi podaci koji su joj pruženi te kojima se upravlja odrađuje *backend*. Da bi se

nekim podacima moglo upravljati potrebno je imati *backend* zbog toga što on sprema sve promjene na određenu lokaciju. [9]

Prije razvoja Interneta *backend* je funkcionirao na način da su se sve izmjene na uređajima mijenjale u lokalnoj memoriji uređaja, dok se nakon razvoja to promijenilo na udaljene ili lokalne baze podataka.

Razvoj *backend* razvoja vrši se koristeći programske jezike. Neki od programskih jezika koji se koriste za komunikaciju sa serverom su:

- *Java*;
- *PHP*;
- *C#*;
- *VB*;
- *Ruby*;
- *Python*;
- *Pearl*;
- *Javascript (Node JS);*
- *Actionscript (Flash Media Server);*
- *CoffeeScript;*
- *C (CGI);*
- *Erlang* i
- *oh* i *SQL* za izvršavanje upita na bazu. [11]

U *backend* razvoj pripadaju sve akcije koje se događaju u sustavu nakon što su podaci s *frontend*-a poslani na server. Nakon slanja na server obavljaju se daljnje provjere te ukoliko je podatak ispravan zapisuje se u bazu podataka.

2.2.2 FRONTEND

Frontend razvoj započeo je razvojem *JavaScripta* 1995. godine. 1997. godine razvojem *CSS*-a *web* stranice su dobile mogućnost uređivanja boja, margina, postavljanje pozadinskih slika. [6]

Frontend razvoj omogućio je interakciju između korisnika i web stranica te uređivanje sadržaja.

Kao što je *backend* razvoj fokusiran na serverski dio aplikacije tako je *frontend* razvoj fokusiran na klijentski dio. *Frontend* dio nije fokusiran samo na uređivanje već se njime razvija *html* struktura stranice koja je potrebna za optimizaciju pretraživačkih algoritama. Prilikom pisanja neadekvatnog *frontend* dijela, nastaju sigurnosni propusti. Problemi s kojima se susreću *frontend* programeri prilikom izrade *frontend* dijela aplikacije su usklađivanje boje, fontova te responzivnosti na različitim pretraživačima. Responzivnost predstavlja izgled aplikacije na različitim rezolucijama. Razlika između začetka *frontenda* te istog u suvremeno vrijeme jest da su pretraživači postali operativni sustavi koji interpretiraju svaki na svoj način. Razlika u pretraživačima postala je ne samo u brzini rada i optimizaciji pretrage već i u vrsti i broju dodataka. Animacije su omogućene *frontend* jezicima.

Prikaz nekih od skriptnih jezika koji se koriste za razvoj *frontenda*:

- *HTML*;
- *JavaScript*;
- *CSS*;
- *Actionscript;*
- *CoffeeScript* (kompajliran u *JavaScript*);
- *XML*-bazirani jezici;
- *VBScript;*
- *Silverlight* i
- *Java*. [15]

Na slici 4. vidljive su jedne od aktualnih razvojnih *frontend web* tehnologija.

Slika 4: Aktualne razvojne *frontend* tehnologije [16]

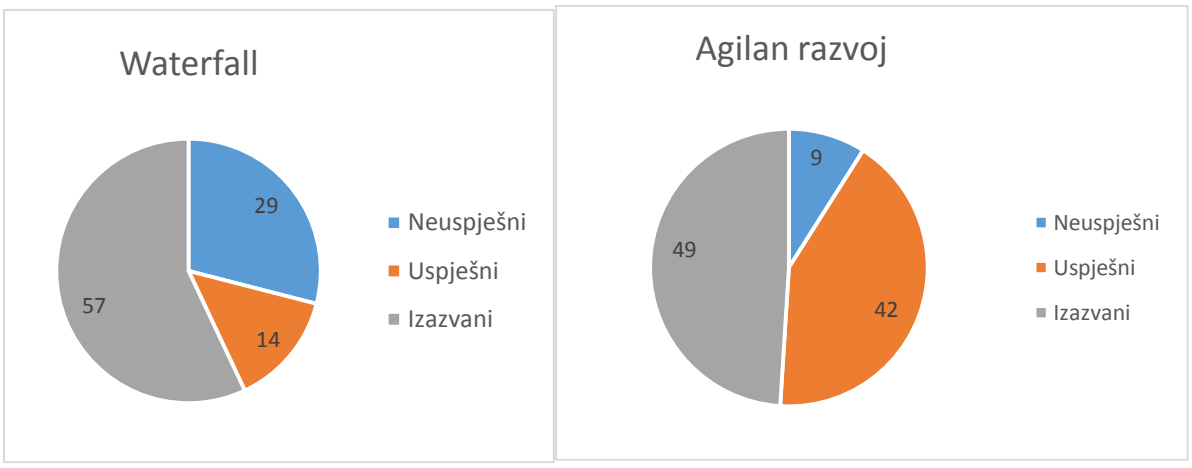
Trenutno aktualne *frontend* razvojne tehnologije su *Reactjs* i *Angular*. Razlog tomu su njihove brzine rada u određenim uvijetima. *Reactjs* se sastoji od dijela *javascripta* i *html* elementa te ima jedinstvenu sintaksu pisanja. Angular se također sastoji od jedinstvene sintakse međutim utemeljuje većinu razvojnih koncepata dok je *Reactjs* korisniji ukoliko je potrebno konstruirati manje elemenata čija je zadaća izvršavanje jednostavnijih operacija koje će se protezati kroz cijeli sustav tako da se mogu više puta koristiti.

2.2.3 AGILNE RAZVOJE METODOLOGIJE

Agilne razvojne metodologije predstavljaju načine razvoja kojima je cilj razvoj aplikacije organiziran na način da trajanje razvoja bude što kraće i uz što manje gubitke. Agilne metodologije su skupine postupaka koje su testirane i s time je dokazana njihova efikasnost.

Agilne razvoje metodologije su definirane po Manifestu o agilnom razvoju softvera. [1]

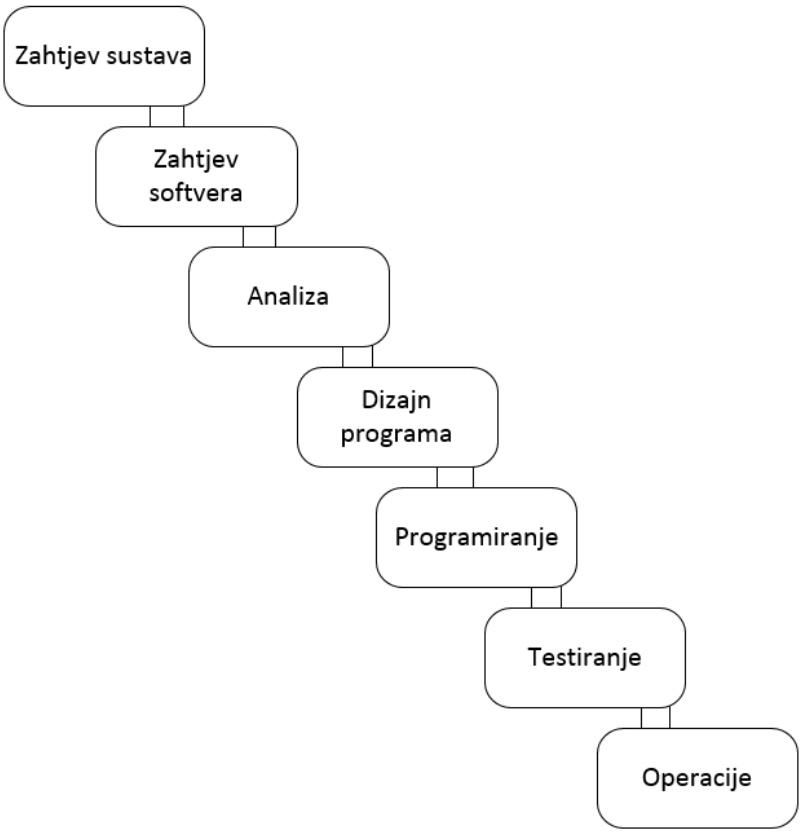
Razlika između agilnih razvojnih metodologije i razvoja bez određene metodologije zvane *Waterfall* vidljiv je na slici 5.



Slika 5: Usporedba dviju pristupa razvoju aplikacija [3]

Iz slike 1. vidljivo je da se postotak uspješnosti projekata znatno mijenja ovisno o pristupu. Korištenjem agilnih metodologija povećava se uspješnost projekata. Zbog toga se koriste agilne metode koje se prikazuju poput kišobrana jer sprječavaju organizacijske probleme i slijede određene procese.

Razlog imenu Vodopadni je taj što se proces razvoja kreće poput slapa kao što je prikazano na slici 6. Zbog cirkuliranja procesa u projektu povećava se prazni hod.



Slika 6: Razvoj korištenjem Vodopadnog pristupa

Prikaz aktualnih agilnih metodologija:

- Adaptivnog softverskog razvoja(engl. *ASD*);
- Agilnog modeliranja;
- Agilnog unificiranog procesa (engl. *AUP*) ;
- Kristalno jasne metode;
- Disciplinirane agilne dostave;
- Dinamičke sustavske razvoje metode (engl. *DSDM*) ;
- Ekstremno programiranje (engl. *XP*) ;
- Metodologija razvoja fokusirana na dodatke sustava (engl. *FDD*) ;
- Opuštena razvojna metoda;
- *Kanban*;

- *Scrum* i
- *Scrum* ban. [5]

Detaljnom analizom pojedinih agilnih metoda vidljivo jest da se sve agilne metode baziraju na uobičajenim vrijednostima i principima međutim razlikuju se po tome što se treba analizirati koji je princip kvalitetniji za određenu situaciju. [2]

3. KOMPARATIVNA ANALIZA SUVREMENIH WEB TEHNOLOGIJA STUDENT PROJEKT APLIKACIJE

Analizom student projekt aplikacije utvrđeno jest da se komparacija vrši usporedbom *frontend*-a i *backend*-a te da se prikažu njihove prednosti i nedostaci.

Tako se komparacija vrši na:

- *Frontend* dijelu student projekt aplikacije i
- *Backend* dijelu student projekt aplikacije.

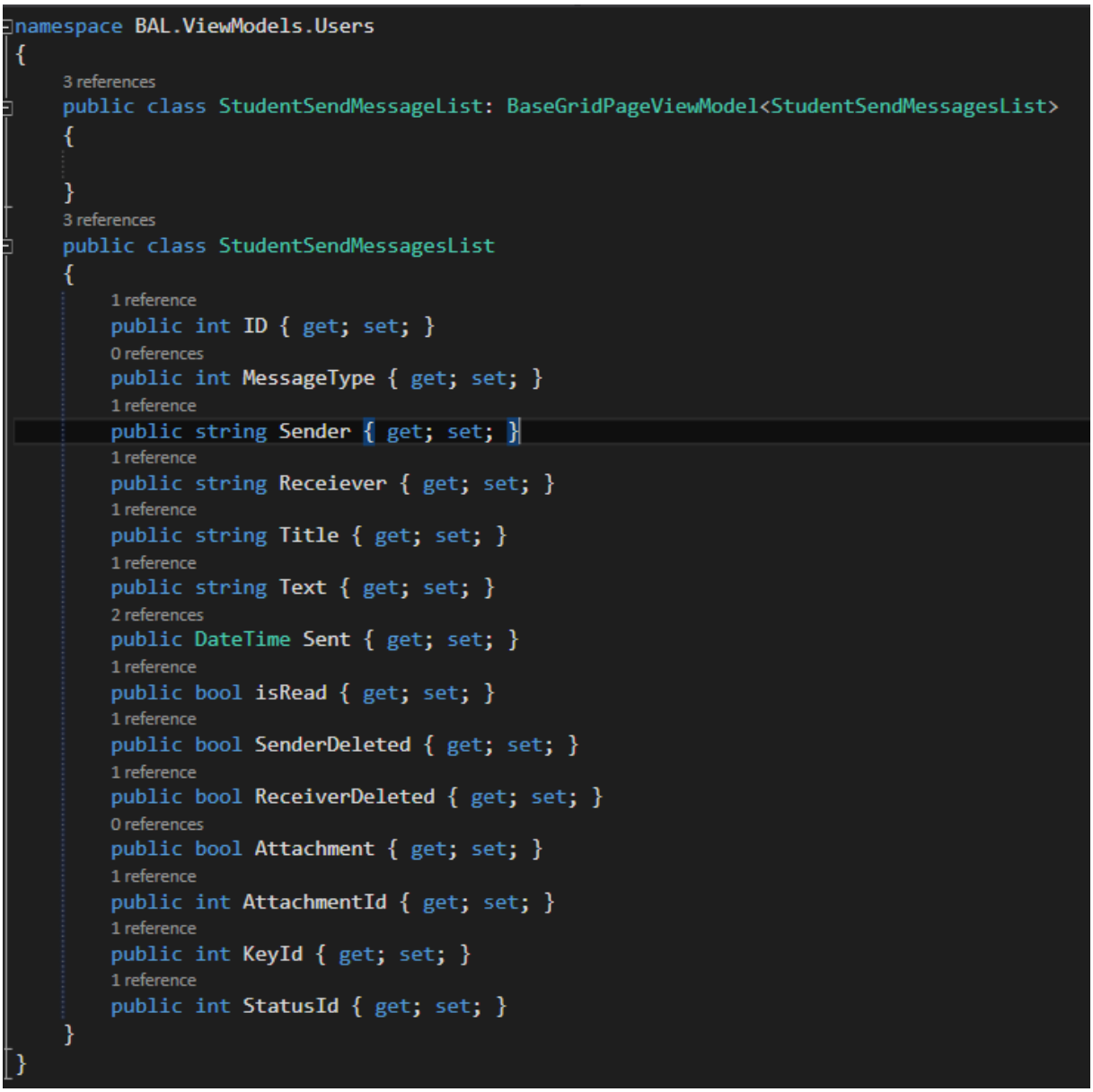
U komparativnoj analizi korištena je *asp.net MVC* aplikacija student projekt. Ona se sastoji od tri glavna dijela model, *view* i *controller*. Model predstavlja klasa objekta koja se sastoji od različitih tipova podataka. Ona se puni podacima i poziva na *controller*-ima. *View* predstavlja stranicu odnosno dio aplikacije vidljiv krajnjem korisniku gdje se prikazuju podaci. *Controller* predstavlja logički dio koji izvršava povezivanje između modela i *viewa*.

Slika 7. prikazuje model *asp.net mvc* aplikacije.

```
1 reference
public class StudentMessagesModel
{
    1 reference
    public StudentSendMessageList StudentSendMessages { get; set; }
    1 reference
    public StudentRecivedMessageList StudentReceivedMessages { get; set; }
}
```

Slika 7: Klasa(model) koja je dio *asp.net mvc* aplikacije

Iz slike je vidljivo da se unutar jedne klase *StudentMessagesModel*-a nalaze dva pod modela. Daljnjom analizom vidljivo je po imenu da se klasa *StudentSendMessageList* koristi za prikazivanje poruka poslanih od strane prijavljenih studenata, dok se klasa *StudentRecivedMessageList* koristi za prikazivanje poslanih poruka prijavljenog studenta studenta. Tipovi podataka koji se nalaze u tim klasama vidljivi su na slici 8.

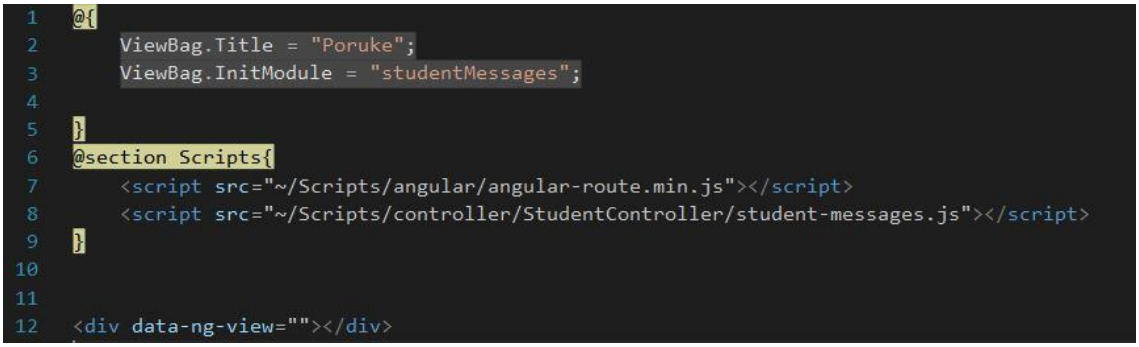


Slika 8: Prikaz podataka koji se šalju na klijentsku stranu (poslane poruke prijavljenog studenta)

U *asp.net MVC*-u moguće je na više načina povezati podatke sa *view*-om. U ovom slučaju korišten je *angular*. Za rad *angulara* potrebno je učitati *angular javascript file*-ove koji se sastoje od *angular-route-min.js* koji omogućuje rad ruta *angulara* te *student-messages.js* koji omogućuje rad Student *controller*-a. Prikaz učitavanja *javascript* datoteka vidljiv je na slici 9.

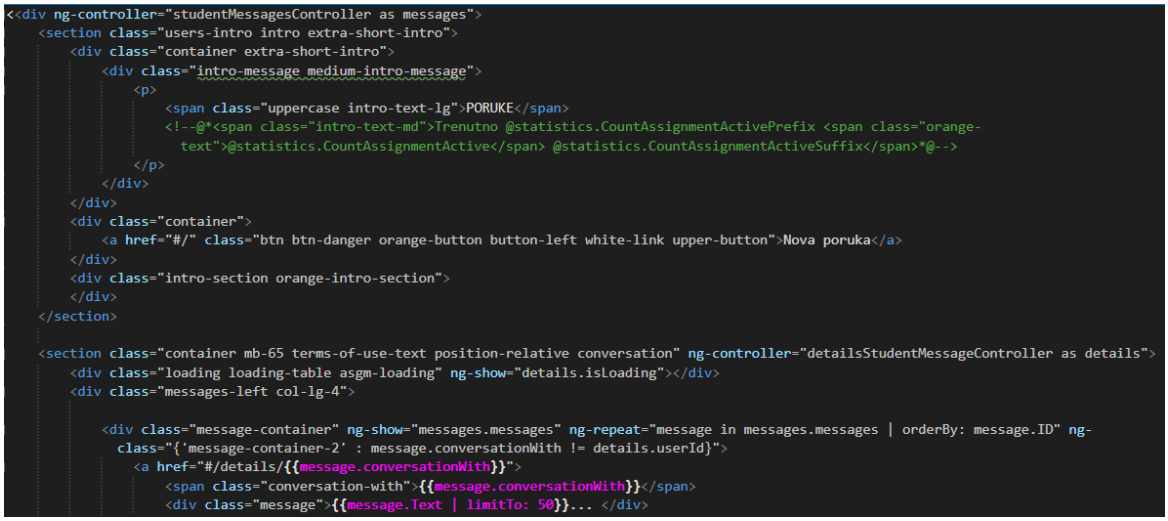
U Student *controller*-u nalazi se logika gdje se učitavaju podaci iz *backend*-a i to se vrši *ajax*-om. Student *controller* predstavlja klijentski način učitavanja podataka, međutim poziva metode na serverskoj strani.

Ajax predstavlja učitavanje *javascript*-a i *XML*-a unutar stranice bez potrebe da se ona osvježi. Takav se način učitavanja zove asinkrono učitavanje. [26]



Slika 9: *Html* prikaz učivanja *view*-a gdje će se prikazati korisničke poruke

Nadalje *angular* se u ovom slučaju koristi tako da se za svaku stranicu napravi *html* uzorak koji je jedinstven za stranicu. Prikaz takvog uzorka vidljiv je na slici 10. Taj se uzorak učita koristeći se *student-messages.js*-om koji na klijentskoj strani provjeri koja je vrsta *controller*-a i ovisno o njoj učita određeni *html* uzorak.



Slika 10: Prikaz statičnog djela *html* koda, kojemu se dodaje *angular* kod (ružičasti kod).

Iznad navedeni *ajax* pozivi se upućuju prema serveru te se njima asinkrono dohvaćaju podaci. Poziv sa klijentske strane na serversku poziva metodu *getAllMessages()* na slici 11. i prema njoj dohvaća podatke poslanih i primljenih poruka te ih sprema i model koji šalje nazad na klijentsku stranu. Dohvaćanje poruke se vrši preko serverskog servisa nazvanog *messagesService*.

U metodi *Messages()* dohvaća se *view* i vraća se stranica(njen *html* kod) dok se u metodi iznad *getAllMessages()* vraćaju samo *JSON* podaci.

```
/// <summary>
/// Gets the messages(inbox & outbox) and returns them at Messages view
/// </summary>
/// <returns></returns>
[AcceptVerbs(HttpVerbs.Get)]
1 reference
public virtual JsonResult getAllMessages()
{
    string UID = SPAuthentication.User.Username;
    var messageService = Service.Get<StudentService>();
    var model = new StudentMessagesModel();

    // Getting received messages
    model.StudentReceivedMessages = messageService.GetRecievedMessages(UID);
    // Getting sent messages
    model.StudentSendMessages = messageService.GetSendMessages(UID);

    return Json(model, JsonRequestBehavior.AllowGet);
}

/// <summary>
/// The view of messages
/// </summary>
[HttpGet]
1 reference
public virtual ActionResult Messages()
{
    return View(Views.ViewNames.Messages);
}
```

Slika 11: Prikaz dviju serverskih metoda koje se koriste za dohvaćanje podataka.

Iz slike iznad vidljivo je da se *view* vraća *ActionResult* metodu dok se podaci vraćaju kao *JsonResult*. Razlog tomu je da *angular* prima *JSON* objekt. Kroz *view* se poziva *angular* a pozivanjem *angulara javascript* podataka se poziva putanja na *getAllMessages()* metodu.

3.1 FRONTEND STUDENT PROJEKT APLIKACIJE

Klijentska strana student projekt aplikacije odnosno *frontend* izrađen je korištenjem *angular* razvojne tehnologije. *Angular* je vrsta *javascript* razvojnog okruženja koja omogućuje korisniku interaktivnost klijentu i ona je bazirana za rad na klijentskoj stani. *Angular* je radno okruženje koje se koristi za razvoj dinamičkih web aplikacija. [19]

Na slici 12. vidljiv je prikaz *javascript* datoteke koja dohvaća *html* uzorak ovisno o imenu *controller-a* koji se poziva na klijentskoj strani.

```
var app = angular.module("studentMessages", ['ngRoute']);

app.config(function ($routeProvider) {
    $routeProvider.when("/", {
        controller: "studentMessagesController",
        templateUrl: "/Templates/Student/Messages.html"
    });

    // Get message details
    $routeProvider.when("/details/:SenderName", {
        controller: "detailsStudentMessageController",
        templateUrl: "/Templates/Student/Messages.html"
    });

    // Get view for new message to some user
    $routeProvider.when("/newMessage", {
        controller: "newStudentMessageController",
        templateUrl: "/Templates/Student/newMessage.html"
    });

    // Send new message to selected user


    $routeProvider.otherwise({ redirectTo: "/" });
});
```

Slika 12: Prikaz rutiranja unutar *angular*-a

Iz koda na slici 12. vidljivo jest da ovisno o *controller*-u *angular* dohvaća drugi *html file* te učitava druge podatke. Na slici su prikazane rute vezane uz komunikaciju odnosno komunikacijski modul.

3.1.1 PREDNOSTI ANGULAR RAZVOJONG OKRUŽENJA IZ RAZVOJNE PERSPEKTIVE

Kako bi se programer odlučio na razvoj koristeći *angular* potrebno je provesti analizu koja će mu pokazati prednosti i nedostatke razvojne tehnologije kako bi mogao utvrditi mogućnost implementacije istog.

Angular je korišten u Student Projekt aplikaciji zbog svoje brzine rada te sustava koji je zamišljen kao SPA(engl. *Single Page Application*).

Za izradu Student Projekta korištena je 1.0 verzija *angulara*.

Angular omogućuje rutiranje na način da su sve stranice u aplikacije dio jedne velike pod nazivom SPA(engl. *Single Page Application*) te se zbog toga očitavanje sadržaja i navigacije po stranicama izvodi brže nego uobičajen način. Nedostatak je što inicijalno prvi učitavanje traje duže. Koristeći *angular* moguće je postaviti arhitekturu u kodu *frontend* dijela te mijenjati i ponovo koristiti ovisno o potrebi. Sa sigurnosnog aspekta *angular* smanjuje potrebu za pisanjem provjera jer u sebi ima ugrađene opcije i metode koje je potrebno aktivirati. *Angular* u sebi ima raznolike mogućnosti filtriranja podataka tako da osoba koja je zadužena za razvoj ne mora pisati svoje. [20]

3.1.2 NEDOSTACI ANGULAR RAZVOJNOG OKRUŽENJA IZ RAZVOJNE PERSPEKTIVE

Početkom 2016. godine izašla je beta verzija *angulara* 2.0 koja omogućuje dodatke među kojima je i razvoj orijentiran na mobilne uređaje međutim cijeli je *angular* biti prepisan i izmijenjen tako da ne izgleda poput aktualnog koda. U toj će se verziji promijeniti način navigacije stranicama unutar koda. [21]

Aplikacije i dalje rade na verziji *angulara* 1.0 međutim vrše se pomaci i sav je razvoj usmjeren u *angular* 2.0.

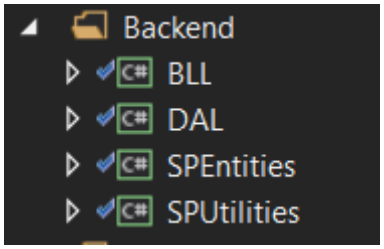
Krivulja brzine učenja u *angularu* u početku ima nizak uspon što prikazuje da je učenje i kompleksnost *angulara* u početku niska dok se ne nauče osnovni dijelovi međutim prilikom izrade velikih sustava te korištenja direktiva, injekcije ovisnosti, modula, servisa i *controller*-a kompleksnost znatno poraste čime se vrijeme razvoja povećava.

Nadalje veliki problem je testiranje kompleksnije aplikacije obzirom da se sastoji od velikog broja komponenti gledajući sa arhitekturne strane. [22]

3.2 BACKEND STUDENT PROJEKT APLIKACIJE

Backend aplikacije predstavljaju tri glavna dijela. Server, aplikacija i bazu podataka. Prilikom izrade aplikacije potrebno je koristiti programski jezik. [23]

Backend Student Projekt aplikacije pisan je *C#* programskim jezikom u *Asp.Net MVC* okolini. *Asp.net MVC* okolina se koristi za razvoj dinamičkih *web* aplikacija. Na server je potrebno postaviti Student Projekt bazu podataka. Zatim je potrebno napraviti izbacivanje aplikacije čime se ona kopira na server. *Backend* Student Projekt aplikacije se sastoji od 4 projekta. *BLL, DAL, SPEntities* i *SPUtilites*.



Slika 13. Prikaz pod projekata Student Projekt aplikacije

Backend metode koji se pozivaju u *controller*-ima usmjerene su prema *BLL*u(engl. *Bussiness Logic Layer*) gdje se nalazi sva poslovna logika. Ukoliko je u aplikaciji implementirana injekcija ovisnosti koriste se sučelja koja imaju putanje na metode, međutim svaki upit poziva metodu. Metode u aplikaciji vraćaju klase, listu klasa, ili druge varijable ovise o logici. U metodama se dohvaća ciljana tablica u bazi podataka pomoću *Entity Frameworka* te se kroz nju vrši navigacija na istu ili na neku drugu ukoliko je potrebno proći kroz više tablica. Nakon prolaska kroz tablice odredi se podatak koji je potreban i on se sprema u varijablu pa se vraća kroz klasu ili se vrati samo varijabla. Svaka konekcija sa bazom ostvaruje se kroz *DAL*(engl. *Data Access Layer*) sloj.

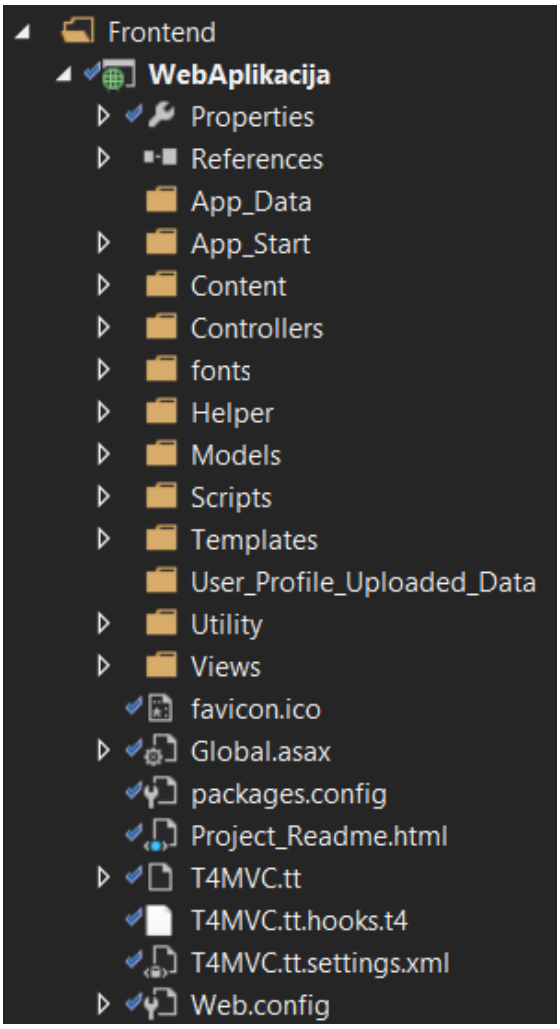
4. ARHITEKTURA STUDENT PROJEKT SUSTAVA PRIMJENOM RAZVOJNIH WEB TEHNOLOGIJA

Student Projekt aplikacija sastoji od jednog rješenja koji u sebi sadrži 6 projekata i jednu skriptu za bazu podataka kao što je vidljivo na slici 14.

Osim na razini rješenja Student Projekt aplikacija se sastoji od *frontenda* i *backenda*, a projekti koji su dio Student Projekta nalaze se u odgovarajućim folderima. Slika 14 prikazuje arhitekturu Student Projekt sustava koja se sastoji od 3 glavna sloja. Klijentski sloj koji komunicira sa korisnikom te vrši komunikaciju sa serverom. Serverski koji je posrednik između baze podataka i korisnika te baza podataka koja skladišti sve podatke. B*ackend* koji se sastoji od *BLL* (engl. *Business Logic Laye*r – sloj aplikacije gdje se nalazi poslovna logika), DAL (engl. *Data Access Layer* – sloj aplikacije koji povezuje logiku aplikacije sa bazom podataka), *SPEntities* i *SPUtilities*, dok *frontend* predstavlja WebAplikacija. *SPEntities* projekt je projekt u kojem se nalaze generirane tablice iz baze. *SPUtilities* projekt sadrži pomoćne metode koje se koriste na više mjesta unutar aplikacije. TestingProjekt je projekt koji se koristi za testiranje određenih metoda unutar aplikacije.

Slika 14. Prikaz Student Projekt strukture

WebAplikacija je *Asp.net MVC* projekt i u njoj se nalaze *controller*-i, pregledi te modeli. Detaljniji prikaz vidljiv na slici 15. Osim samih modela, pregleda i *controller*-a unutar *web* aplikacije se nalaze i druge datoteke poput *App_Starta* koji je zadužen za rute unutar aplikacije, sadržajne datoteke gdje se nalaze *CSS* stilovi, fontova, pomoćnih metoda i drugih datoteka.

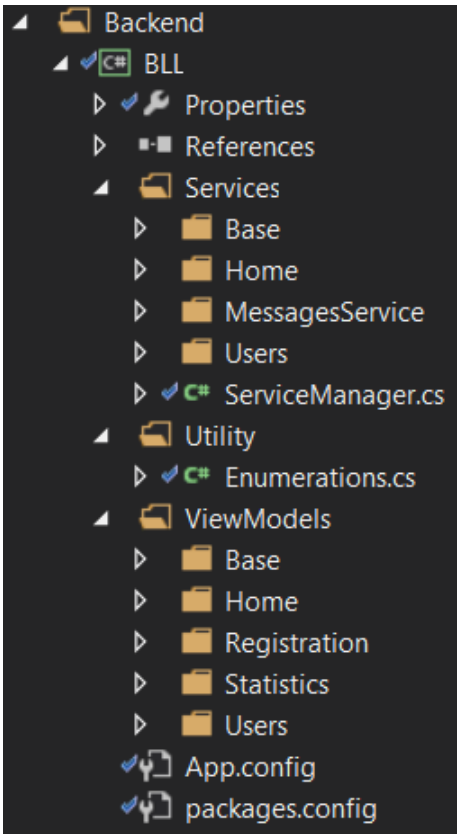


Slika 15. Prikaz projekta WebAplikacija i njegovih dijelova

Podaci kojima se puni model pune se na način da se pozivaju metode iz *BLL*-a. Izgled *BLL*-a vidljiv je na slici 16. Iz slike 16 vidljivo da se *BLL* sastoji od tri glavna dijela:

- Servisi – koriste se za dohvat metoda;
- *Utility* – pomoćne metode i
- *ViewModeli* – preslikavanje i konverzija modela koji komuniciraju sa bazom podataka.

WebAplikacije poziva sučelja *BLL* sloja koja implementiraju servise. Unutar servisa nalaze se metode za poslovnu logiku aplikacije. Cilj takve arhitekture jest da se svaki sloj može zamijeniti drugom tehnologijom bez da njegov rad izravno utječe na ostale slojeve unutar aplikacije. Takva se arhitektura naziva *DDD* (engl. *Domain Driven Design*).



Slika 16. Prikaz *BLL* projekta u kojemu se nalazi poslovna logika

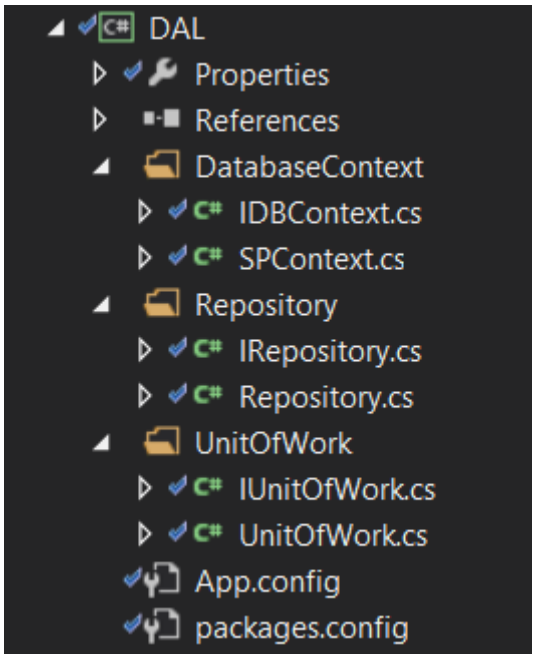
Na slici 17 vidljiv je *DAL*. *DAL* se sastoji od 3 dijela:

- *DatabaseContext*;
- *Repository* i
- *UnitOfWork*.

U folderu *DatabaseContext* nalazi se kontekst odnosno putanja prema bazi podataka, dok se u *IDBContextu* nalaze metode koje omogućuju praćenje izmjena prilikom komunikacije sa bazom, spremanje na bazu podataka, povezivanje storane procedure i po potrebi se proširuje.

U folderu *repository* nalazi se implementacija generičkog repozitorija te sučelje istog. Pomoću repozitorija pozivaju se metode koje omogućuju komunikaciju sa bazom od filtriranja podataka pa do *inserta* i *delete*-a.

Folder *UnitOfWork* isto se sastoji od implementacije i sučelja. Unutar implementacije nalaze se inicijalizacija konteksta u konstruktoru, spremanje, otpuštanje podataka te poziv na bazu koji dohvaća podatke iz određene tablice.



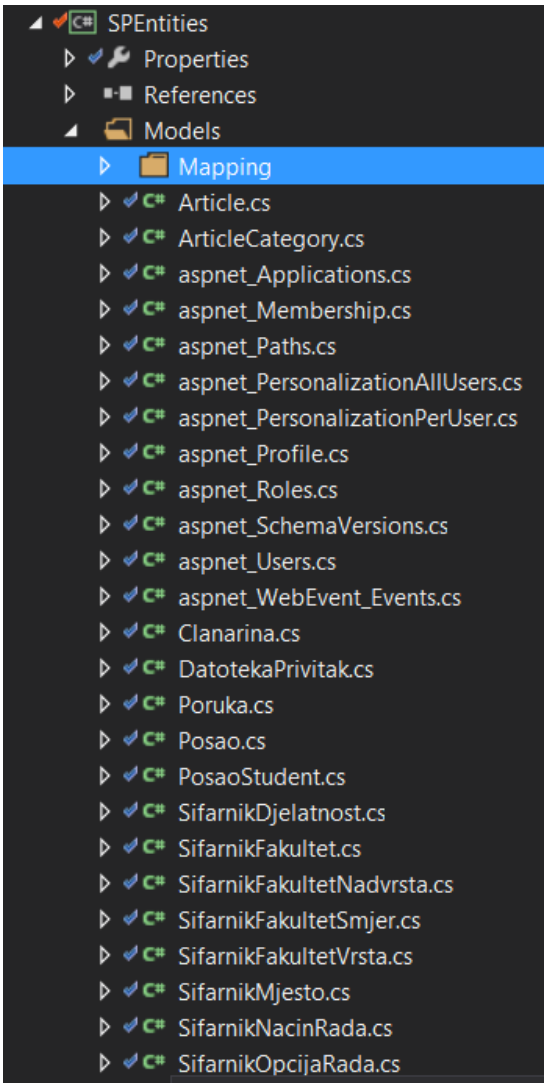
Slika 17. DAL projekt i prikaz njegovih sučelja koji vrše komunikaciju sa bazom podataka

Slika 18. prikazuje projekt *SPEntities*. Projekt *SPEntities* u sebi sadrži sve tablice te mapiranja na iste unutar baze podataka. Za komunikaciju sa bazom koristi se *entity framework*, a metoda kojom se generiraju tablice i njihova mapiranja naziva se *code first* metoda.

Ukidanjem ovisnosti između slojeva omogućuje se izmjena sloja drugom tehnologijom bez negativnog učinka na ostatak aplikacije. Svaki sloj može ovisiti samo o sloju iznad te ne smije postojati iznimka u tom toku. Kroz *DDD* postoje različiti načini prikazivanja arhitekture zbog toga što je to teorijski koncept, međutim arhitektura koja se najviše koristi sastoji se od četiri osnovna sloja:

- Korisničko sučelje – prezentacijski sloj;
- Aplikacijski sloj;
- Domenski – modelni sloj i
- Infrastrukturni sloj. [24]

Korisničko sučelje prikazuje i upravlja interakcijom sa korisnikom te se prilikom razvoja naziva i *frontend*, a u Student Projekt aplikaciji vidljiv je kao dio projekta WebAplikacija gdje se nalazi *html*, *angular*, *javascript* i *jquery*. Aplikacijski sloj prikazuje drugi dio projekta WebAplikacije *controller*-e koji komuniciraju sa *BLL*-om. Taj sloj ima za ulogu koordinaciju zadataka. Domenski sloj je u kojemu se nalazi sva aplikativna odnosno poslovna logika. Naziv toga sloja u Student Projekt aplikaciji jest *BLL*.



Slika 18. SPEntities projekt u kojem se nalaze generirani entiteti (tablice)

Infrastrukturni sloj omogućuje interakciju sa ostalim slojevima i njegova je uloga postavljanje arhitekture te prosljeđivanje komunikacije kroz njega između domenskog sloja te baze podataka. Na primjeru Student Projekt aplikacije taj je sloj prikazan kao *DAL*. Svaki sloj ovisi samo o sloju ispod. Slika 18 prikazuje entitete. Cilj korištenja *DDD-a* jest u tome da ukoliko je potrebno zamijeniti bazu podataka samo se taj sloj promijeni bez mnogo izmjena unutar aplikacije u sloju poslovne logike.

5. ANALIZA FUNKCIONALNOSTI KORIŠTENJEM INJEKCIJA OVISNOSTI

Injekcija ovisnosti prikazana prikazuje vezu između objekata unutar aplikacije. Svaka se radnja može opisati servisom. Kupovina knjige narudžbom, prijevoz taxi službom. Servis prikazuje na objekt odnosno implementaciju. Injekcija ovisnosti vidljiva je kroz pojmove servisa i klijenta.

Servis je definiran kao objekt koji izvršava akciju, dok je klijent korisnik servisa te se poziva na servise koji izvršavaju akcije. Veza između klijenta i servera naziva se ovisnost. Iz te veze vidljivo jest da klijent ne može funkcionirati bez servisa. [25]

Injekcija ovisnosti je način pisanja koda čime se smanjuje veličina koda te olakšava održavanje. Osim toga njom se omogućuje razdvajanje ovisnosti određenih dijelova sustava radi mogućnosti nadogradnje ili izmjene tehnologije. Ona predstavlja suvremen koncept pisanja koda. Problemi koji nastaju ukoliko se ne koristi taj koncept su: povećano vrijeme održavanja koda, nemogućnost optimizacije koda, stvara se ovisnost o tehnologiji čime se znatno smanjuje mogućnost optimizacije sustava u budućnosti, povećanje troškova održavanja zbog toga što razvojni inženjeri prelaze na korištenje novijih tehnologija te je teže naći osobu koja se bavi starijom tehnologijom i mnogi drugi.

Injekcija ovisnosti postala je standardan koncept u programiranju tako da je već podržavana od mnogih programskih jezika i razvojnih okruženja, npr. *C#, angualr* 2.0 i mnogi drugi.

5.1 IZGLED KODA BEZ UPOTREBE INJEKCIJE OVISNOSTI

Ukoliko se neki kod piše bez ograničenja i u zadatku toga koda je potrebno definirati objekt *Emailer* koja u sebi sadrži drugi objekt kod izgleda poput koda ispod.

```
public class Emailer {

    private SpellChecker spellChecker;

    public Emailer(){

    this.spellChecker = new SpellChecker();

    }
```

```
        public void send(String text) { .. }
}
```

Nakon pisanja koda potrebno ga je testirati pa se u svrhu testiranja koriste *unit* testovi kojima se testiraju svi dijelovi aplikacije. U ovom slučaju potrebno je testirati vrši li se provjera prije izvršavanja metode *send*. To se može napisati prema kodu ispod.

```
Public class MockSpellChecker : SpellChecker {
        Private bool didCheckSpelling = false;
        Public bool checkSpelling(String text) {
        didCheckSpelling = true;
        return true;
        }
        Public bool verifyDidCheckSpelling()
        { return didCheckSpelling; }
}
```

Ukoliko postoji takva implementacija te se želi implementirati objekt *Emailer* koji provjerava pogreške prilikom pisanja teksta na engleskom jeziku kod od objekta Email vidljiv je u nastavku.

```
Public class Emailer {
        Private SpellChecker spellChecker;
        Public Emailer() {
                This.spellChecker = new EnglishSpellChecker();
}
```

Ukoliko se napravi provjera teksta na engleskom jeziku, ona će se izvršavati međutim ukoliko je potrebno napraviti provjeru za francuski jezik to neće biti moguće. Problem je u tome što *Emailer* objekt mora biti fleksibilniji, a to se postiže na nekoliko načina:

- Može se koristiti ručna konstrukcija koda koja se ne preporuča zbog problema prilikom testiranja većeg broja servisa;
- Tvornički (engl. *The Factory pattern*) način pisanja koda i
- Servisni lokator (engl. *The Service Locator*).

Ukoliko se želi postojeći kod koristiti za različite dijelove aplikacije bez potrebe da ga se replicira koristi se injekcija ovisnosti.

5.1.1 RUČNA KONSTRUKCIJA KODA

Koristeći ručnu konstrukciju koda moguće je modificirati kod na način da se održi struktura i smanji teret koji se dobiva stvaranjem ovisnosti.

```
Public class Emailer {

Private SpellChecker spellChecker;

Public void setSpellChecker(SpellChecker spellChecker){

}

...

}
```

U kodu iznad izmjena se izvršila na način da se uklonio konstruktor koji je kreirao *SpellChecker* sa metodom koja prima *SpellChecker*. Test takvog koda prikazan je ispod.

```
Public void ensureEmailerChecksSpelling() {

MockSpellChecker mock = new MockSpellChecker();

Emailer emailer = new Emailer();

Emailer.SetSpellChecker(mock);

emailer.send(„Test“);

assert.IsNotNull(mock.verifyDidCheckSpelling());

}
```

Moguće je istu stvar napraviti za različite jezike. To je prikaz tehnike kojom se ručno odvija konstrukcija koda. Postoji mogućnost slanja ovisnosti putem konstruktora kao na kodu ispod.

```
Public class Emailer {
        Private SpellChecker spellChecker;
        Public Emaielr(SpellChecker spellChecker) {
                this.spellChecker = spellChecker;
        }
}
```

Na ovaj način moguće je injektirati servis putem konstruktora kao što je vidljivo na primjeru ispod.

```
Emailer servis = new Emailer(new EnglishSpellChecker());
```

Prikaz eksplicitnog definiranja ovisnosti omogućuje da se prilikom izrade objekta *Emailer* mora definirati ovisnosti ukoliko je ukoliko je zaboravljen poziv na *setSpellChecker()* metodu. Problem takvog načina pisanja koda vidljiv je u slučaju greške, jer za ispravak greške potrebno je mijenjati velik dio koda. Osim toga problem ovakvog pisanja koda vidljiv je prilikom većeg broja korisnika čime se vrši teret zbog velikog broja kreiranja ovisnosti. [25]

5.1.2 TVORNIČKI NAČIN PISANJA KODA

Ovakvim načinom pisanja koda smanjuje se teret koji se dobiva ručnom konstrukcijom koda. Prilikom izrade tvorničkog načina koda koristi se objekt zvan tvornica. Cilj tog objekta jest automatizirati proces ovisnosti. [25]

Razlika između tvorničkog načina pisanja koda te ručne konstrukcije koda vidljiva je na primjerima ispod.

Primjer kada se provjera na email servisu postavlja putem konstruktora.

```
Public class Emailer {
        Private SpellChecker spellChecker;
```

```
        Public Emailer(SpellChecker spellChecker){
        this.spellChecker = spellChecker;
        }
        …
}
```

Izrada ovisnosti se vrši putem tvornice što je vidljivo na primjeru ispod.

```
Public class EmailFactory {
        Public Emailer newEnglishEmailer(){
                Return new Emailer(new EnglishSpellChecker());
        }
}
```

Analizom koda pisanog tvorničkim načinom vidljivo jest da se eksplicitno definira kakvog će tipa biti *Emailer*, što je u ovom slučaju *newEnglishEmailer* i on se sastoji od engleskog *spellcheckera*. Svaki kod koji koristi engleski email servis piše se na način:

```
Emailer servis = new EmailerFactory().newEnglishEmailer();
```

Ono što je različito u primjeru koda ručno konstruiranog jest da klijentski kod nema referencu na dijelove *Emailer* objeka te se dodao nivo apstrakcije, odnosno odvojio se kod koji koristi *Emailer* sa kodom koji kreira *Emailer*. Prednost ovakvog načina pisanja vidljiva jest u kompleksnijim ovisnostima.

Ovakav način pisanja koda jedino mora znati koja se tvornica koristila da bi poznavao ovisnosti. Test ovakvog koda vidljiv ne u primjeru ispod.

```
Public void testEmailer() {
        // Napravi novi mock za svaku ovisnost
        MockSpellChecker spellChecker = new MockSpellChecker();
        …
        // Postavi sve mockane servise na emailer
```

```
        Emailer emailer = new Emailer();
        emailer.setSpellChecker(spellChecker);
        …
        Emailer.send(„test“);
        // Provjeri jel sve prošlo po planu
        assert …;
}
```

Testiranja klijenata vrši se na način prema kodu ispod.

```
Public class EmailClient {
        // Dohvati emailer iz tvornice
        Private Emailer emailer = new EmailerFactory().newEnglishEmailer();
        Public void run(){
        Emailer.send(message());
        Confirm(„Sent!“);
        }
}
```

Gledajući kod iznad vidljivo jest da klijent nema podataka o *Emailer* objektu već ovisi o tvornici.

Iako tvornice rješavaju većinu problema koji nastaju ručnom konstrukcijom koda, ne rješavaju sve. Osim problema testiranja, tvornice moraju popratiti svaki servis, te svaku varijaciju servisa. Takav primjer vidljiv je na kodu ispod.

```
Public class EmailerFactory {
        Public Emailer newJapaneseEmailer(){
                Emailer service = new Emailer();
                service.setSpellChecker(new JapaneseSpellChecker());
```

```
            service.setAddressBook(new EmailBook());
            service.setTextEditor(new SimpleJapaneseEditor());
            return service;
        }
        Public Emailer newFrenchEmailer() {
            Emailer service = new Emailer();
            service.setSpellChecker(new FrenchSpellChecker());
            service.setAddressBook(new EmailBook());
            service.setTextEditor(new SimpleFrenchEditor());
            return service;

        }
}
```

Problem nastaje ukoliko je potrebno zamijeniti *EmailBook* sa *PhoneAndEmailBook* i to na svakom servisu jer su sva tri identična. Da bi se taj problem izbjegao potrebno je raditi nove tvornice za svaku varijaciju. Testiranje koda na projektima sa tvornicama uzrokuje veliki gubitak vremena koji se znatno povećava sa veličinom projekta. [25]

5.1.3 SERVISNI LOKATOR

Servisni lokator je način pisanja koda koji u sebi sadrži značajke tvorničkog načina pisanja. Servisni lokator prima ključ koji govori aplikaciji da se traži njegova vrijednost, kao što je vidljivo na primjeru ispod.

```
Emailer emailer = (Emailer) new ServiceLocator().get(„Emailer“);
```

Prema primjeru iznad vidljivo jest da servisni lokator traži *Emailer*, što ujedno prikazuje razliku između tvorničkog načina pisanja te korištenja servisnog lokatora. Razlika je u tome što servisni lokator je također tvornica međutim on može primiti bilo koji tip servisa dok

tvornički način može primiti samo jedan tip i to taj koji očekuje. Servisni lokator radi na način da u početku registrira sve resurse pa ih samo kasnije poziva u dijelu gdje se koriste.

Nedostatak servisnog lokatora vidljiv je u tome što je on ujedno tip tvornice te zbog toga ima problema prilikom testiranja te dijeljenja koda. Osim toga ključevi koji se koriste za rad sa servisnim lokatorom su napisani na način da se prilikom ispisa krivog mapiranja referenca događa greška jer se kreira krivi objekt. Rješenje tih problema pronađeno je korištenjem injekcije ovisnosti. [25]

5.2 PRIKAZ KODA UPOTREBOM INJEKCIJE OVISNOSTI

Injekcija ovisnosti omogućuje korištenje prednosti svih iznad navedenih načina pisanja kod bez negativnih posljedica. Injekcija ovisnosti omogućuje klijentima da ne znaju za ovisnosti koje se nalaze u kodu ni kako se one kreiraju. Cilj injekcije ovisnosti jest da se implicitno poznaju ovisnosti. *Hollywood* princip prikazuje način injekcije ovisnosti koji omogućuje pružanje ovisnosti bez potrebe da je ona zatražena. [25]

Hollywood princip jest princip upotrebe injekcije ovisnosti na način da se pisanje koda vrši na način da se ne pozivaju svi dijelovi koda samo od sebe već postoji alat koji se aktivira te ukoliko je potrebno pozvati drugi dio koda on vrši taj poziv umjesto da se sav kod pozove.

5.2.1 HOLLYWOOD PRINCIP

Način rada *Hollywood* principa vidljiv je u ručnoj konstrukciji koda. Razlika je u tome što se svi zadaci oko izrade i mapiranja ovisnosti odvijaju preko vanjske okoline (engl. *Framework*). Takva se okolina naziva *DI*(engl. *Dependency Injector*). Hollywood princip omogućuje izmjenu ovisnosti koja se naziva *IoC*(engl. *Inversion Of Control*). [25]

DI okoline u sebi sadrže *IoC* kontejnere. Ti se *IoC* kontejneri razlikuju ovisno o programskom jeziku. Razlika između *IoC* kontejnera vidljiva jest u načina izrade što omogućuje različite brzine određenih metoda. Takav je primjer vidljiv na tablici 1. Brzina *IoC* kontejnere se razlikuje ovisno o situacijama u kojima se odvija injekcija. Sve vrijednosti unutar tablice prikazane su u milisekundama.

Najsporiji *IoC* kontejner je *Ninject*. *MEF*, *LinFU* i *Sprint.NET* rade brže nego *Ninject*. Od njih su brži *AutoFac*, *Catel* i *Windsor* te *StructureMap*, *Unity* i *LightCore*. Nedostatak *Sprint.Net*

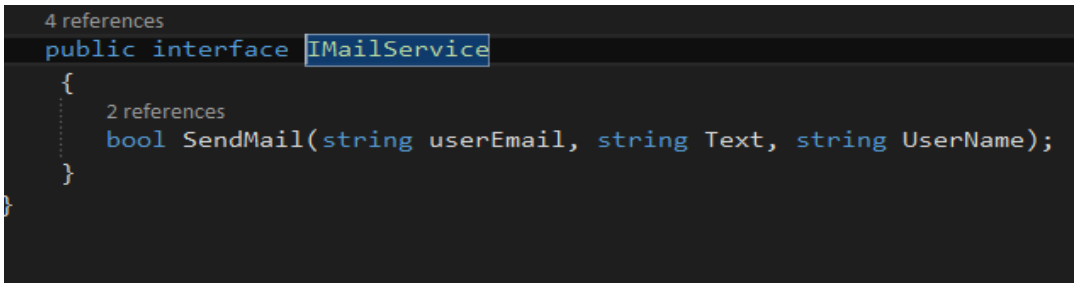
IoC kontejnera je taj da se može konfigurirati samo sa *XML*-om.(engl. *Extensible Markup Language*). [26]

Unutar Student Projekt aplikacije korišten je *Ninject IoC* kontejner. Prilikom instalacije *Ninjecta* u folderu *AppStart* pojavljuje se nova klasa *NinjectWebCommon* koja u sebi sadrži jezgru. Unutar *NinjectWebCommon* klase vrši se registracija jezgre, te je nakon iste potrebno registrirati klase te servise koji prikazuju na te njihove implementacije tako da se ovisnosti mogu riješiti prilikom pokretanja aplikacije kao što je vidljivo na slici 19.

```
38          /// <summary>
39          /// Creates the kernel that will manage your application.
40          /// </summary>
41          /// <returns>The created kernel.</returns>
            1 reference
42          private static IKernel CreateKernel()
43          {
44              var kernel = new StandardKernel();
45              try
46              {
47                  kernel.Bind<Func<IKernel>>().ToMethod(ctx => () => new Bootstrapper().Kernel);
48                  kernel.Bind<IHttpModule>().To<HttpApplicationInitializationHttpModule>();
49
50                  RegisterServices(kernel);
51                  return kernel;
52              }
53              catch
54              {
55                  kernel.Dispose();
56                  throw;
57              }
58          }
59
60          /// <summary>
61          /// Load your modules or register your services here!
62          /// </summary>
63          /// <param name="kernel">The kernel.</param>
            1 reference
64          private static void RegisterServices(IKernel kernel)
65          {
66              kernel.Bind<IMailService>().To<MailService>().InRequestScope();
67              kernel.Bind<IMainRepository>().To<MainRepository>().InRequestScope();
68          }
69      } NinjectWebCommon
70  }
```

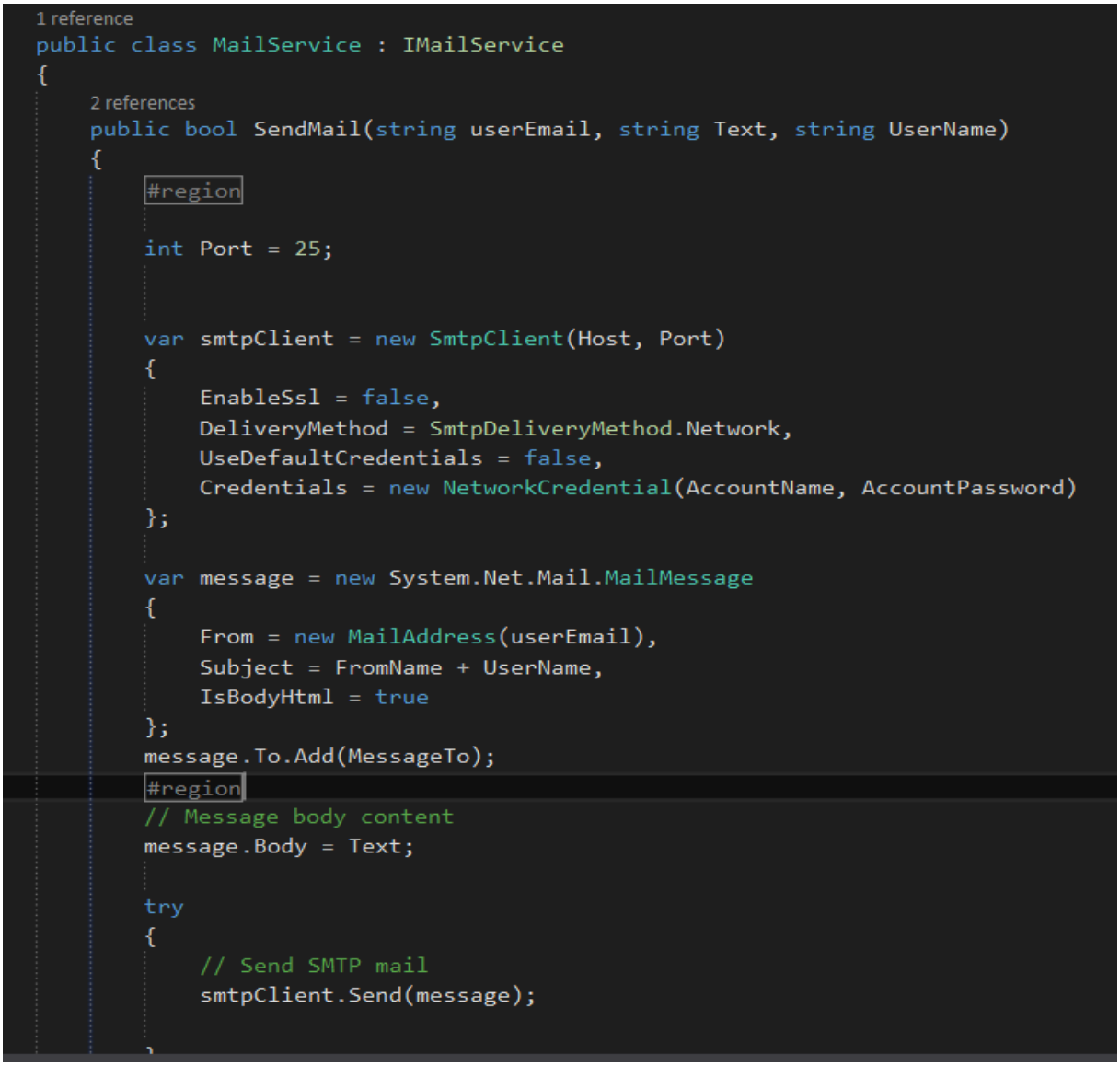
Slika 19: *Ninject IoC* kontejner unutar Student Projekt aplikacije

U implementaciji Student Projekta prema slici 19. vidljivo jest da su registrirana dva servisa koji prikazuju na implementacije istih. Na slici 20. vidljiv je prikaz *IMailService* servisa.



Slika 20: Prikaz implementacije *IMailService* servisa

Ukoliko se uđe dalje u metodu na koju pokazuje taj servis može se vidjeti da se u tu metodu šalju tri parametra. Sva tri parametra su tekstualni zapisi. Metoda *SendMail* omogućuje slanje email poruka korištenjem injekcije ovisnosti. Prikaz implementacije *SendMail* metode vidljiv je na slici 21.



Slika 21: Implementacije *MailService* servisa na koju pokazuje *IMailService* sučelje.

Na slici 22. vidljiv je prikaz injekcije ovisnosti unutar Student Projekt aplikacije. Injekcija ovisnosti se implementira na način da se izradi privatna varijabla *IMailService* sa imenom _mail te se unutar konstruktora *HomeControllera* injektira taj servis. Ukoliko *Ninject* nije instaliran projekt bi se mogao pokrenuti međutim ne bi se očitala web stranica već bi izbacila grešku. Implementacijom injekcije ovisnosti, ovisnosti se mapiraju i riješe na startu aplikacije.

```
7 references
public partial class HomeController : BaseController
{
    private IMailService _mail;
    0 references
    public HomeController(IMailService mail)
    {
        _mail = mail;
    }

    [AcceptVerbs(HttpVerbs.Post)]
    1 reference
    public virtual JsonResult SendContact(MessageModel viewModel)
    {
        if (_mail.SendMail(viewModel.userEmail, viewModel.Text, viewModel.NameAndSurname))
            ViewBag.MailSent = true;
        else
            ViewBag.MailSent = false;

        return Json(ViewBag.MailSent, JsonRequestBehavior.AllowGet);
    }
```

Slika 22. Implementacija injekcije ovisnosti

Container	Singleton	Transient	Combined	Complex	Property	Generics	IEnumerable	Conditional	Child Container	Interception With Proxy	Prepare And Register	Prepare And Register And Simple Resolve
No	108 78	126 116	147 168	222 206	348 176	107 106	275 177	214 180	1899 524	88 106	4	4
Autofac 3.5.2	893 723	2568 2571	6407 4071	18191 11244	32706 20892	5158 3346	17288 12199		108964 86101	51505 37712	487	643
Caliburn.Micro 1.5.2	538 353	670 405	1867 1085	7969 4632	10427 6064		7758 4514				63	70
Catel 4.1.0	456 460	5448 6074	14069 15657	32753 36704		13698 15615				5884 5804	9398	8824
DryIoc 1.4.1	**31** 48	**42** **57**	**56** 83	**92** **81**	**97** **87**	**64** **70**	**318** **225**	**68** 69			45	22822
Dynamo 3.0.2.0	105 80	134 103	234 162	823 493	855 519						19090	18509
fFastInjector 1.0.1	75 68	134 112	280 204	680 428							9324	8177
Funq 1.0.0.0	149 111	181 132	451 338	1327 852	1299 789						**12**	Error
Grace 2.4.2	188 127	303 294	907 926	2061 1291	2828 1589	707 467	2601 1645	805 536	17498 10605	8723 5580	122304	123571
Griffin 1.1.2	374 231	377 243	971 573	2813 1548							12609	12089
HaveBox 2.0.0	91 76	110 95	122 87	222 194	1119 697		2252 1373			**868** **538**	81766	83181
Hiro 1.0.4.41795	207 140	209 146	219 154	290 199	3104 1931						310074*	362562*
IfInjector 0.8.1	108 86	144 115	180 142	233 166	385 269	170 131					2101	2795
LightCore 1.5.1	203 171	3364 1998	34315 34496	193101* 205435*	2487 1843	23120 15653	52456 31250				233	247
LightInject 3.0.2.6	34 **44**	49 63	70 **74**	107 95	**97** 94	74 82	349 246	73 **67**		1660 1093	316	1079
LinFu 2.3.0.41559	4163 2443	24399 16041	64412 41898	170694 104578							140	538
Maestro 1.5.4	333 259	397 306	1115 728	3512 2556	3866 2367	783 534	3901 2707	1070 693		8955 5331	232	952
Mef 4.0.0.0	34967 19746	53521 31946	87598 65396	175864 169289	180329* 178984	198621* 151746	137126 140873				23	3054
Mef2 1.0.30.0	252 179	263 187	328 250	559 481	1388 1099	347 241	1759 1379				7330	11858
MicroSliver 2.1.6.0	566 303	818 543	2901 1802	8322 8170							18	**23**
Mugen 3.5.1	459 359	810 582	2380 1666	9042 6674	11883 7521	71914 76734	6944 7444	2060 1369	706941* OoM	5051527* Error	559	2378

Tablica 1: Prikaz brzina rada *IoC* kontejnera za različite situacije [26]

Prilikom korištenja inverzije ovisnosti, razvojni inženjer dužan je istražiti vrste *IoC* kontejnera i ovisno o slučaju upotrijebiti onaj koji mu najviše odgovara. Prikaz analize brzine rada različitih *IoC* kontejnera vidljiv je u tablici 1.

Iz tablice 1 vidljivo jest da različiti *IoC* kontejneri rade različitim brzinama u različitim situacijama ovisno kojim se komponentama programskog jezika korisnik koristi.

5.2.2 INVERZIJA KONTROLE

Inverzija kontrole se koristi zajedno sa injekcijom ovisnosti. Inverzija kontrole je način programiranja pri kojemu je cilj zajedno sa injekcijom ovisnosti maknuti ovisnosti unutar koda. Korištenjem inverzije kontrole događa se smanjenje ovisnosti (engl. *Loose coupling*) između klijenta i servisa na način da dijelovi servisa imaju minimalan ili nikakav utjecaj na

klijenta. Inverzija kontrole se koristi kada dio servisa ne može izvršavati zadatak zbog nedostatka informacija ili funkcionalnosti. [25]

6. ZAKLJUČAK

Napretkom razvojnih tehnologija razvijaju se i načini razvoja. Potrebno je stalno proučavati nove načine razvoja jer se u njima pronalaze rješenja za aktualne probleme prilikom razvoje ili optimizacije postojećih sustava. Razvojne tehnologije se granaju u dvije grane *frontend* i *backend* međutim zbog velike brzine razvoja događa se specijalizacija za jednu granu ponekad i područje unutar jedne grane. Razvojni postupci poput tvornica, servisnog lokatora te injekcije ovisnosti budućnost su razvojnih tehnologija i na njima je vidljivo koliko je znanje potrebno da bi se implementirala injekcija ovisnosti na većim sustavima. Potrebno je poznavanje različitih razvojnih postupaka te ukomponiravanje istih. Komparativna analiza vršena je na sustavu Student Projekt. Napravljena je sa ciljem analize aktualnih razvojnih web tehnologija i to prema dvama razvojnim dijelovima. *Frontend* i *backend*. Komparativnom analizom utvrđeno jest da se kompleksan sustav napravi potrebno je poznavanje razvojnih postupaka, arhitekture, injekcije ovisnosti, *frontenda*, *backenda*, baze podataka te komunikaciju između baze podataka i aplikacije. Cilj komparativne analize bio je usporediti brzine rada i funkcionalnosti razvojnih tehnologija i samog razvoja u vrijeme začetka razvojnih tehnologija pa sve do danas. Iz komparativne analize vidljivo je da se razvojne tehnologije razvijaju velikom brzinom čime se nude različite tehnologije ovisno o karakteristikama sustava dok u prošlosti nije postojao takav izbor. Zahvaljujući razvojnim tehnologijama te razvojnim metodologijama moguće je napraviti potreban sustav uz minimalno vrijeme izrade. Svake se godine sve više ubrzava razvoj razvojnih tehnologija što ujedno primorava razvojne inženjere da se specijaliziraju za jedno područje. Nadalje se očekuje daljnji rast aplikacija i njihovog tržišta.

POPIS KRATICA

IT - *Information Technology*

ICT - *Information Communication Technology*

WWW - *World Wide Web*

HTML - *HyperText Markup Language*

CMS - *Content Managment System*

CPU - *Central Processing Unit*

HTTP - *HyperText Transfer Protocol*

URL - *Uniform Resource Locator*

CSS - *Cascading Style Sheets*

XML - *Extended Markup Language*

BLL - *Bussiness Logic Layer*

DAL - *Data Access Layer*

DDD - *Domain Driven Design*

DI - *Dependency Injector*

IoC - *Inverison Of Control*

POPIS SLIKA

LITERATURA

Knjige, znanstvena i stručna djela:

[1] United States. Bureau of Labor Statistics, U.S. Department of Labor. "Information Security Analysts, Web Developers, and Computer Network Architects". Occupational Outlook Handbook, 2012-13 Edition. Retrieved 2013-01-17.

[2] Shore, J. The Art of Agile Development. [Internet]. O'Reilly Media, Inc.; 2008 Available from poestisty: https://poetiosity.files.wordpress.com/2011/04/art_of_agile_development.pdf

[12] Tm Berners-Lee, editor. The World Wide Web: Past, Present and Future. The World Wide Web Consortium(W3C). August 1996 Massachusettse Institute of Technology. Cambridge.

[13] Rojas, Raul Plankalkul: The First High-Level Programming Language and its Implementation". Institut frame Informatik, Freie Universitat Berlin, Technical Report B-3/2000

[24] Eric Evans, Domain Driven Design – Tackling Complexity in the Heart of Software April 2003.

[25] Dhanji R. Prasanna – Dependency Injection (2009.)

[26] Mozzila Developer Network (MDN) - https://developer.mozilla.org/en-US/docs/

Internet izvori:

[3] Monjurul Habib, Agile software development methodologies and how to apply them – Code project 30 Dec 2013, Available from: http://www.codeproject.com/Articles/604417/Agile-software-development-methodologies-and-how-t

[4] Manifesto for Agile Software Development, 2001, Available from: http://agilemanifesto.org/iso/en/

[5] The Definitive List of Software Development Methodologies, 18 July 2008, Available from: http://noop.nl/2008/07/the-definitive-list-of-software-development-methodologies.html

[6] Aaron Lumsden, A Brief History of the World Wide Web, 25 Sep 2012, Available from: http://webdesign.tutsplus.com/articles/a-brief-history-of-the-world-wide-web--webdesign-8710

[7] The case of separating front and back-end, 16 October 2014, Available from: https://blog.42.nl/articles/the-case-separating-front-back-end/

[8] Labyrinth Solutions, A (brief) history of CMS development, January 22, 2012, Available from: http://www.contegro.com/info-center/designers-blog/blog-article/_thread_/a-brief-history-of-cms-development

[9] Callie Kavourgias, What's the Difference Between the Front-End and Back-End, 2015. Available from: http://blog.digitaltutors.com/whats-difference-front-end-back-end/

[10] Margaret Rouse, ICT(Incormation and communications technology – or technologies). Available from: http://searchcio.techtarget.com/definition/ICT-information-and-communications-technology-or-technologies

[11] Quildreen Motta, Which programming languages are front-end and which ones are back-end, Sep, 18, 2014. Available from: http://www.quora.com/Which-programming-languages-are-front-end-and-which-ones-are-back-end

[14] Vangie Beal, programming language, available from: http://www.webopedia.com/TERM/P/programming_language.html

[15] Ivan Codesido, What is front-end development? 28, September, 2009, The guardian, awailable from: http://www.theguardian.com/help/insideguardian/2009/sep/28/blogpost

[19] Angular, available from: https://docs.angularjs.org/guide/introduction

[20] Stackoverflow, http://stackoverflow.com/questions/18414012/what-does-angularjs-do-better-than-jquery

[21] Preetish Panda, What's New in AngularJS 2.0, March 02, 2015, available from: http://www.sitepoint.com/whats-new-in-angularjs-2/

[22] Ruoyu Sun, This I Wish I Were Told About Angular.js, 25 May 2013, available from: http://ruoyusun.com/2013/05/25/things-i-wish-i-were-told-about-angular-js.html

[23] Josh Long, I don't speak your langauge: frontend vs backend, September, 25, 2012, available from: http://blog.teamtreehouse.com/i-dont-speak-your-language-frontend-vs-backend

[17] Mawlana Bhashani Science and technology University, [Image on Internet] avilable from: http://mbstu.ac.bd/depts/ict/courses.html

[18] Batch Man, [Image on Internet] available from: http://www.instructables.com/id/CMD-Tricks-101/

[3] [Image on Internet] http://www.webopedia.com/FIG/PROG-LAN.gif

[4] [Image on Internet] http://the.loading-info.net/2011/02/looking-back-world-wide-webs-first-web.html

[16] [Image on Internet] http://blog.ittestsonline.com/wp-content/uploads/2015/04/Blog-pic.png

[26] Daniel Palme, 30 August, IoC Container Benchmark – Performance comparison, [Image on Internet] http://www.palmmedia.de/blog/2011/8/30/ioc-container-benchmark-performance-comparison