Универзитет у Београду

Факултет организационих наука

Лабораторија за софтверско инжењерство

ЗАВРШНИ РАД

Развој софтверског система за војну обуку паса употребом .NET Core технологија

Ментор

проф. др Саша Лазаревић

Студент

Катарина Симић 2016/0020

Београд, јул 2020. године

УНИВЕРЗИТЕТ У БЕОГРАДУ
ФАКУЛТЕТ ОРГАНИЗАЦИОНИХ НАУКА

Лабораторија за софтверско инжењерство

ЗАВРШНИ РАД

Развој софтверског система за војну обуку паса употребом .NET Core технологија

Ментор

проф. др Саша Лазаревић

Студент

Катарина Симић 2016/0020

Београд, јул 2020. године

Развој софтверског система за војну обуку паса употребом .NET Core технологија

Развојем интернет и веб технологија, веб апликације постају озбиљан конкурент уобичајним десктоп апликацијама када је у питању избор платформе на чијим основама ће се имплементирати идеја неког софтверског решења. Све већа потреба за комуникацијом у пословном свету и убрзани начини пословања траже свеобухватна решења која ће уштедети време и новац док ће истовремено корисницима омогућити све опције као и традиционални начини комуницирања физичким присуством. Рад пружа увид у развој софтверског система, веб апликације за праћење рада логистичке службе за обуку војних паса употребом ASP .NET Core развојног оквира, MVC оквира и Entity Framework Core објектно - релационог мапера на серверској страни и JavaScript, Bootstrap, AJAX и jQuery технологија на клијентској страни. Описан је процес развоја софтверског система помоћу упрошћене Ларманове методе, што је омогућило да развој апликације буде испраћен одговарајућом документацијом кроз све фазе развоја: прикупљање захтева, анализа, пројектовање, имплементација и тестирање.

кључне речи: софтверски систем, веб апликација, Ларманова метода, .NET Core, ASP .NET Core MVC, Entity Framework Core

Садржај

Списак слика:

Списак табела:

1. Увод

У склопу Мајкрософтове .NET Core платформе важно место заузимају веб апликације којима је намењен развојни оквир ASP.NET Core. Уз помоћ овог оквира могуће је изградити скалабилне апликације користећи патерне попут *Model-View-Controller* унутар .NET окружења.

Са убрзаним развојем интернета и повећаним растом броја корисник интернета, веб апликације постају све више значајне. Корисник од једне веб апликације очекује да она буде доступна у сваком тренутку без обзира где се корисник налази и који паметнан уређај користи. Захтеви који се постављају веб аплиакцијама јесу да оне морају бити сигурне, флексибилне и скалабилне како би се испунили циљеви потражње.

Окружење .NET представља једну од најпопуларнијих платформи за развој апликација на мрежи. ASP.NET Core је оптимизован за савремене веб апликације, а његов модуларни дизајн омогућава апликацијама да зависе само од оних функција које заиста користе. Према популарној веб страници *Builtwith.com*, ASP.NET заузима готово 40% тржишта (https://trends.builtwith.com/framework/ASP.NET). Технологија коју подржава *Microsoft* доминира на тржишту веб апликација. Мноштво различитих алата и оквира који се налазе у софтверу базираном на .NET-у пружа разноврсне могућности на различитим нивоима сложености. Предузећа и велике корпорације прелазе са старих система на нове, динамичне и робусне веб апликације. На тржишту засићеном различитим технологијама, ASP.NET Core се показао популарним избором који нуди разноврсне могућности за развој *business-to-business* (B2B) и *business-to-consumer* (B2C) апликација, чиме постаје преферирани избор програмера [1].

У овом раду биће приказан преглед радног оквира ASP.NET Core као и опис процедуре изградње веб апликације помоћу овог радног оквира. Циљ овог рада је развој софтверског система за праћење ангажовања службених паса на задацима у Војсци Србије, развијен коришћењем Ларманове методе развоја софтвера.

Рад је организан у десет поглавља и подељен је на теоријски и практични део.

У другом поглављу описан је теоријски део који почиње описом коришћених технологија. Биће описане технологије коришћене на серверској страни: платформа ASP .NET Core, ASP.NET MVC архитектура и *Entity Framework Core* оквир. Описане су карактеристике и основни концепти C# објектно-оријентисаног програмског језика. У оквиру ове секције дати су описи основних коцепата овог језика - класе и методе, апстрактне класе, интерфејси, наслеђивање, полиморфизам и енкапсулација. Затим су описане техологије коришћене за потребе имплементације система на клијентској страни: *Bootstrap, HTML, CSS, AJAX, JavaScript* и *jQuery*.

У трећем поглављу почиње опис практичног дела рада. Описана је Ларманова метода развоја софтвера, након чега је кроз низ поглавља детаљно објашњен процес развоја софтверског система коришћењем исте. У четвртом поглављу описан је развој система кроз фазу прикупљања захтева. Затим је у петом поглављу дат приказ развоја софтверског система кроз фазу анализе, док је у шестом поглављу приказан развој помоћу фазе пројектовања. Седмо поглавље описује процес развоја система кроз фазу имплементације, а у осмом је приказана последња фаза развоја софтверског система - фаза тестирања. У деветом поглављу дат је закључак. У последњем, десетом поглављу приказана је коришћена литература.

Завршни рад је заснован на веб апликацији, написаној у програмском језику C#, у окружењу *MS Visual Studio 2019* при коришћењу локалне *SQL* базе података.

2. Преглед коришћене технологије

Веб технологије можемо поделити у две групе:

- Клијентске (енг. *Client side*) - извршавају се у корисниковом веб претраживачу и њихов код је могуће видети
- Серверске (енг. *Server side*) – извршавају се на удаљеном, дељеном рачунару – серверу и за разлику од клијентских технологија могуће је видети само резултат извршења скрипте.

У овом поглављу биће детаљно описане све технологије које су коришћене приликом израде студијског примера. На самом почетку поглавља описана је .NET платформа и њене основне компоненте, основни концепти C# програмског језика и примена ASP .NET MVC оквира. У наставку је описан оквир *Entity Framework Core* и техологије коришћене на клијентској страни – *Bootstrap, JavaScript, jQuery и AJAX*.

2.1 Разумевање .NET платформе

Мајкрософтова .NET *Framework* платформа представља развојну платформу која укључује *Common Language Runtime* (CLR) чија је намена управљање извршењем кода и обезбеђивање библиотеке класа за изградњу апликација. *Microsoft* је дизајнирао .NET *Framework* тако да ова платформа најбоље функционише на *Windows* оперативном систему. Наиме, платформа .NET *Framework* је намењена само за *Windows* оперативне системе, док је .NET Core вишеплатформски [2].

У току развоја ове платформе, независан део програмера је радио на развоју.NET имплементације под називом *Mono* пројекат. *Mono* представља међуплатформски пројекат отвореног кода (engl. *open-source*) и сет алата који су компатибилни са .NET оквиром чија је намена развој мулти-платформских апликација. Написан је од стране компаније *Xamarin* коју је *Microsoft* купио 2016. Године, а библиотека се појавила 2004. године и од тада прати трендове у оквиру оргиналног .NET оквира. Појавом ове библиотеке појављују се .NET апликацијаме на *Linux* и *Mac* оперативним системима. Компанија *Microsoft* је одучила да развије оквир отвореног кода за мулти-платформски развој, чиме настаје .NET Core оквир отвореног кода за развој .NET апликација на другим оперативним системима као што су *Linux* и *MacOS* [1].

Куповином *Xamarin*-а од стране *Microsoft* компаније, *Xamarin* постаје алат за развој апликација за мобилне уређаје и изградњу *cloud* сервиса за подршку апликацијама за мобилне уређаје. Модерни развој мобилних апликација и *cloud* платформи је учинио да *Windows* постане много мање важан оперативни систем. На самом почетку креирања апликације, једна од важнијих ставки на које треба обратити пажњу јесте оперативни систем на којем би се покретала апликација. На ову одлуку утиче тржиште, крајњи корисници и у неким случајевима, развојни тим програмера. Један од главних разлога за коришћење .NET Core окружења је могућност развоја апликација које се могу покренути на *Windows*, али и *на Linux, MacOS* и на различитим архитектурама попут *x86 и ARM* архитектуре (енг. *cross-platform*), што је погодно за пуно сценарија, укључујући десктоп апликације. На овај начин, Microsoft је раздвојио уску везу .NET-а са Windows оперативним системом [1]. Апликације развијене помоћу .NET-а могуће је написати у C#, F# или *Visual Basic* језику. C# је једноставан, модеран и објектно оријентисан програмски језик. F# је отвореног кода, функционални програмски језик који такође укључује објектно-оријентисано и императивно програмирање. *Visual Basic* је приступачан језик са једноставном синтаксом за изградњу објектно оријентисаних апликација. Без обзира у ком од наведених језика је написан, код ће се покренути на било ком компатибилном оперативном систему [1].

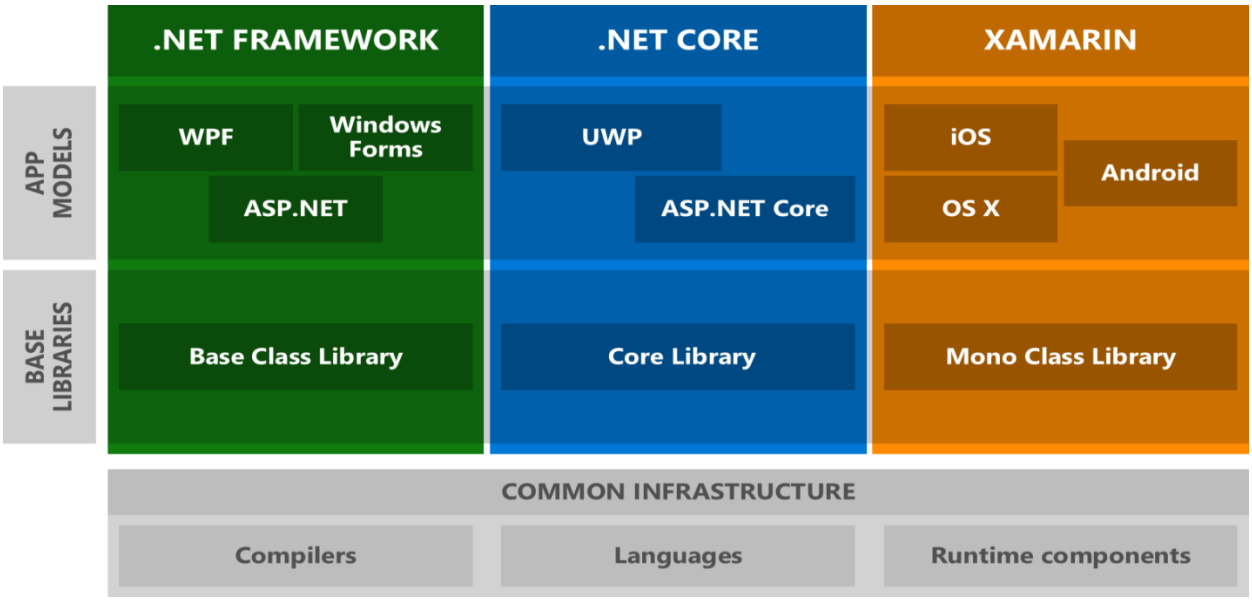
Док је мењао .NET, тако да он буде међуплатформски, *Microsoft* је искористио могућности да преради .NET и задржи главне делове, а оно што је било уско повезано са *Windows* оперативним системом је изостављено. Нови

производ, назван *.NET Core*, укључује међуплатформску имплементацију *CLR*-а (енг. *Common Language Runtime*), који је познатији као *CoreCLR* и модернизовану библиотеку класа познату као *CoreFX*. Платформа .NET Core је скуп компоненти за извршавање, библиотеке и компоненте компајлера које се могу користити у различитим конфигурацијама за радне задатке на уређајима. Кроз различите платформе и отворени код, платформа .NET Core обезбеђује једноставан модел развоја и флексибилност у раду при развоју апликација. Изворни код платформе је доступан на *GitHub*-u под *MIT* (Масачусетски технолошки институт) лиценцом. Платформа .NET Core се односи на неколико технологија, укључујући *.NET Core, ASP.NET Core u Entity Framework Core* [1].

Постоје три различите платформе намењене за развој апликација које нуди *Microsoft* компанија [1]:

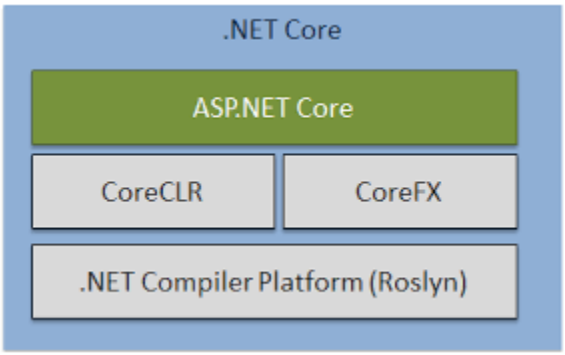
- .NET Framework
- .NET Core
- Xamarin

Microsoft је дефинисао .NET стандард, основни скуп *API*-ја (енг. Application programming interface) који свака .NET платформа мора да имплементира Свака имплементација такође може изложити додатне *API*-је који су специфични за оперативне системе на којима ради, као што је на пример, платформа *.NET Framework* само за *Windows .NET* имплементацију која укључује *API*-је за приступ *Windows* регистру. Како би се функционалности на једноставан начин проширили, *Microsoft* је развио *NuGet,* менаџер пакета, изграђен посебно за .NET и садржи преко 90.000 пакета. Свака од ових платформи има своје врлине и мане, будући да су дизајниране за различите сценарије. Да би се употребио .NET стандард потребно је да се инсталира .NET платформа која имплементира .NET стандард спецификацију. На следећем дијаграму приказано је како три варијанте .NET-а (*App Models*) деле заједнички .NET стандард и инфраструктуру [1].



Слика 1 - *Приказ .NET платформе.* (2020, May 31). [Digital image]. https://content.altexsoft.com/media/2018/07/word-image.jpeg

2.2 Компоненте .NET Core окружења



Слика 2 - *Компоненте .NET Core окружења*. (2018, November 24). [Digital image]. https://i0.wp.com/www.tutorialsteacher.com/Content/images/core/dotnet-core.png

Као што се може видети на слици, платформа .NET Core укључује *.NET Compiler* платформу Рослин, затим *.NET Core runtime CoreCLR*, *CoreFX* и *ASP.NET Core* који је део *.NET Core SDK* (енг. *Software development kit*) тако да није потребно засебно инсталирати *ASP.NET Core* [1].

- На најнижем нивоу налази се *.NET Compiler Platform* - скуп отворених извора компајлера и API-ја за анализу кода за C# и Visual Basic .NET језике. Пројекат нарочито укључује верзије C-а и *VB NET*-а компајлера - компајлере написане на самим језицима [3].
- *CoreCLR* - основна и виртуелна машина компоненте .NET Core-а. То је радно окружење у .NET-у које покреће кодове и помаже у олакшавању процеса развоја пружајући разне услуге као што су управљање нитима, безбедност типа, управљање меморијом, робусност и слично. У време извршења *CoreCLR* учитава *IL* (енг. *Intermediate language*) код из програмског склопа, затим наступа *JIT* (енг. *Just-In-Time compilation*) који исти код потом компајлира у инструкције процесора. Процесор извршава овај код на машини. Предност овог процеса компајлирања, који се састоји из два корака, јесте у могућности креирања компоненти *CLR* за *Linux* и *macOS* као и за *Windows*. Исти *IL* код се покреће било где, због другог процеса компајлирања који генерише код за изворни оперативни систем и скуп инструкција за *CPU (енг. central processing unit)* [3].
- *CoreFX* – основна библиотека класе за .NET Core. Садржи типове за колекције, датотечне системе, конзолу, *JSON, XML, async* и многе друге [3].
- *ASP.NET Core* - популаран оквир за развој веб апликације на .NET платформи. Перформансе су кључни фокус ASP.NET Core - а. Бржи је од осталих популарних веб оквира према независним мерењима. Дизајниран је тако да омогућава *runtime* компонентама, API-ијима, компајлерима и програмским језицима брзу еволуцију, поуздану стабилност и подршку за одржавање апликације. Пружа разне могућности подршке животног циклуса како би се задовољиле потребе различитих апликација [1].

2.3 C# програмски језик

C# је објектно-оријентисан програмски језик опште намене који служи за грађење апликација у оквиру .NET окружења. Омогућава креирање, развој и имплементацију напредних десктоп и веб апликација [4].

Објеткно-оријентисано програмирање има за циљ да представи све што је потребно испрограмирати што ближе моделу реалног света. Код објектно-оријентисаног програмирања, све се своди на објекте. Објекат је увек представљен својом класом. Сама класа није објекат, већ представља опис по коме ће објекат бити направљен.

Класа унутар себе садржи различите елементе, а најбитнији су атрибути и методе. Атрибути су својства која ће будући објекат имати, а методе су функције које ће моћи да се примењују над објектом.Објекат се креира на основу класе и тек тада се може користити. За објекат се каже и да је инстанца, односно примерак класе, а то значи да се на основу класе може направити неограничен број објеката који деле заједничке особине дефинисане у класи [4].

Подражава генеричке методе и типове чиме се обезбеђује већа безбедност типа и боље перформансе. Изрази са интегрисаним упитима језика чине изразито типизован упит првокласном конструкцијом језика. Као објектно оријентисани језик подржава концепте енкапсулације, наслеђивања и полиморфизма [4].

2.3.1 Класе и методе

Класа представља шаблон за креирање објеката који имају исте особине. Класама се дефинише како ће се генерисати објекти и које особине и функционалности ти објекти поседују, путем атрибута и метода. Методама се дефинише понашање класа, а атрибутима стање. Класом се врши генерализација групе објеката односно, занемарују се карактеристике које нису јединствене за групу објеката посматраних у једном контексту. Објекат представља једно конкретно појављивање своје класе [4].

Свака метода садржи [4]:

1) Повратни тип (*void* уколико не враћа никакав тип)
2) Назив
3) Листу параметара
4) Тело методе

Кључна реч *return* користи се да означи вредност коју враћа метода уколико она постоји. Да би се креирао нови објекат класе, позива се конструктор који представља мехнизам помоћу којег се креира објекат. Конструктор је дефинисан у класи и може да буде без аргумената или са аргументима. Постоје две врсте конструктора, то су параметарски и непараметарски конструктор. Основне особине конструктора дате су у наставку [4]:

1) Конструктор поседује исти назив као што је и назив класе
2) Конструктор се покреће приликом декларације објеката
3) Конструктори немају повратни тип података

C# језик подржава преклапање (енг. *overload*) метода у оквиру једне класе, што значи да класа може имати две или више истоимених метода, при чему се све такве методе морају разликовати по потпису. Преклапање, односно *overload* методе се остварује тако што се свака метода разликује у броју или типу (или оба) параметара које метода добија као улазне параметре. На исти начин је то могуће и са конструкторима [4].

2.3.2 Наслеђивање

Класа може да наслеђује другу класу како би се оригинална класа проширила или прилагодила. Наслеђивање класе омогућава да се поново искористи функционалност те класе. У програмском језику C# једна класа може да наследи само једну класу. У C#-у је могуће забранити наслеђивање класа користећи резервисану реч sealed, која се такође може користити и при писању метода или својстава (енг. property) и на тај начин онемогућити наслеђивање класе, али и преписивање (енг. override). Приликом дефинисања класа могуће је користити кључну реч *final*. То означава да се ова класа не може наслеђивати [4]. У овом примеру дефинисана је класа по имену *Asset*:

```
public class Asset { public string Name; }
```

Након тога, дефинисане су класе *Stock* и *House*, које ће наслеђивати класу *Asset*. Класе *Stock и House* добијају све што има *Asset* и додатне чланове које саме дефинишу:

```
public class Stock : Asset // наслеђује класу  Asset
{
public long SharesOwned;
 }
public class House : Asset // наслеђује класу  Asset
{
public decimal Mortgage;
}
```

Потклассе, *Stock* и *House*, наслеђују својство *Name* од основне класе (енг. base class), Asset. Потклассе се зову још и изведене класе (енг. *derived classes*) [4] .

Правила наслеђивања у C# су следећа [4]:

- Интерфејс може да наследи више интерфејса.
- Класа може да наследи више интерфејса.
- Класа не може да наследи више класа, може наследити само једну класу.
- Интерфејс не може да наследи класу.
- Класа може у исто време да наследи класу и да имплементира један или више интерфејса.

2.3.3 Енкапсулација

Енкапсулација или учауривање је концепт по којем је информација у класи заштићена од директног приступа и једини начин да се промени је кроз дефинисане методе (на пример приликом управљања неким уређајем ми позивамо одређене функције без потребе да знамо како се и над којим подацима оне извршавају). Овакав концепт програмирања назива се „црна кутија“ - познат је улаз и излаз, али не и процес који се одвија унутар функције. Учауривање се остварује поделом чланова класе на јавне и приватне. Јавни чланови могу слободно да се користе у било ком делу програма (ван тела саме класе), док су приватни доступни само у оквиру класе у којој су дефинисани. Поља су најчешће приватна, док су методе већином јавне. Учауривањем се избегава додела нерегуларних података, јер се не додељују директно пољима, већ коришћењем метода које имају уграђен механизам за проверу њихове исправности . Без обзира на ниво заштите, у оквиру тела класе сви чланови су међусобно доступни [4].

Постоје два битна аспекта енкапсулације [4]:

- обједињавање података и функција у јединствен ентитет (класа)
- контрола могућности приступа члановима ентитета (модификатори приступа)

Обједињавање података и функција у јединствен ентитет остварује се помоћу класа и одређују се границе ентитета. Заштита се врши помоћу модификатора приступа који се наводе пре дефиниције поља или метода. Модификатори за одређивање права приступа члану су [4]:

- *public* - јавни приступ без ограничења
- *private* - члановима се може приступити само у оквиру класе; не могу се наследити
- *protected* - члановима се може приступити у оквиру изворне или изведене класе, тј. могу се наслеђивати

Постоје два разлога зашто се користи енкапсулација [4]:

- омогућава се контрола коришћења - објекат се може користити искључиво преко јавних метода
- омогућава се смањивање утицаја промена - уколико су детаљи имплементације објекта приватни могу се променити, а да те промене не утичу директно на корисничке објекте (које једино могу да приступе јавним методама)

2.3.4 Полиморфизам

Полиморфизам је фундаментални концепт објектно оријентисаног програмирања. Способност промењиве да референцира објекте различитих типова и да аутоматски позива одговарајућу методу објекта који се референцира се назива полиморфизам. Полиморфизам омогућава да основна класа дефинише функције које ће бити заједничке за све изведене класе, с тим да те изведене класе на свој начин могу да имплементирају те функције. То значи да објекти неког специфичног типа могу да се посматрају као објекти основног типа. Међутим, полиморфно понашање позива омогућава да изведени објекат изрази своју различитост. Важно је знати и да основна класа и оне изведене формирају хијерархију која се поставља од уже генерализације ка широј. У том случају, основна класа поседује све што изведене класе могу да користе. Изведене класе на овај начин имају могућност да самостално имплементирају функције [4]. У наставку следи пример методе који показује примену полиморфизма.

```
public static void Display (Asset asset)
{
System.Console.WriteLine (asset.Name);
}
```

Ова метода може да прикаже својство *Name* објекта класе и *Stock* и *House,* пошто обе наслеђују класу *Asset*. Полиморфизам функционише на основу тога што потклассе (*Stock u House*) имају све особине своје основне класе (*Asset)*. Међутим, обрнуто не важи.

Полиморфизам се реализује помоћу виртуелних метода. Приликом дефинисања, виртуелне методе се морају имплементирати. Приватне и статичке методе се не могу дефинисати као виртуелне. Процес имплементације виртуелне методе у изведеној класи се назива реимплементација и приликом тог процеса се користи кључна реч *override*. Функција означена резервисаном речју *virtual* може бити редефинисана у потклассама које треба да обезбеде специјализовану имплементацију. Методе, својства, индексери и догађаји могу се декларисати као виртуелни [4].

```
public class Asset
{
 public string Name;
 public virtual decimal Liability { get { return 0; } }
}
```

Потклаcа редефинише виртуелну методу применом модификатора *override.*

```
public class House : Asset
{
 public decimal Mortgage;
 public override decimal Liability
 {
 get {  return Mortgage; }
 }
}
```

Својство *Liability* објекта *Asset* подразумевано има вредност 0. Класа *Stock* не мора да то изричито задаје. Међутим, *House* задаје својство *Liability* тако да враћа вредност *Mortgage*:

```
House mansion = new House { Name=”Mansion”, Mortgage=250000 };
Asset a = mansion;
```

```
Console.WriteLine (mansion.Liability); // 250000
Console.WriteLine (a.Liability); // 250000
```

Потписи, повратни типови и доступност виртуелних и редефинисаних метода морају бити идентични. Редефинисана метода може да позове имплементацију своје основне класе помоћу резервисане речи base, која има две основне сврхе: приступање редефинисаној функцији чланици из поткласе и позивање конструктора основне класе [4].

Рано повезивање [4]:

- статичко повезивање
- позив методе се разрешава у току компајлирања.

Касно повезивање [4]:

- динамичко повезивање
- позив методе се разрешава у току извршавања.

Касно повезивање са једне стране повећава флексибилност, док са друге стране утиче на перформансе:

- потребно је више времена за повезивање
- потребно је пронаћи класу чија је метода позвана.

Методе нису имплицитно виртуелне из следецих разлога [5]:

- Перформансе - потребно је време да би се одредило (пронашло) коју од реимплементираних метода треба позвати, што најчешће не утиче у великој мери на перформансе. Већи проблем је немогућност оптимизације кода приликом компајлирања. За невиртуелне методе ова информација је доступна у време компајлирања, што значи да се приликом компајлирања могу спровести одређене оптимизације.

- Дизајн - методе класе које су намењене за интерну употребу и које се односе искључиво на дизајн дате класе не треба да се реимплементирају у изведеним класама, па самим тим не треба ни да буду виртуелне.

Поред ових основних објектно оријентисаних принципа, C# олакшава развој софтверских компоненти кроз неколико иновативних језичких конструкција, укључујући следеће [5]:

- Енкапсулирани потписи метода названи делегати, који омогућавају обавештења о догађајима који су сигурни за тип.
- Атрибути који обезбеђују декларативне метаподатке о типовима у време извођења.
- *Language-Integrated Query (LINQ)* који пружа уграђене могућности упита у различитим изворима података.

2.3.5 Апстрактне класе

Апстактне класе се дефинишу на исти начин као и обичне класе. Разлика је у томе да се оне не могу појављивати. Сврха њиховог постојања је када постоји разлог да класа има једну или неколико апстрактних метода. Што се апстрактних метода тиче, оне се разликују од обичних метода. Немају тело методе. Реализација апстрактних метода се преноси на методе класе које наслеђу апстрактну класу. Кључна реч која се користи за апстрактне класе и апстрактне методе је *abstract*. Апстрактне класе немају своје инстанце и користе се за

извођење других класа, конструктор апстрактне класе не може бити апстрактан и апстрактне методе из основне класе морају се прегазити методама у изведеним класама [5]. Следећи пример показује креирање конструктора апстрактне класе *Abc* и уз помоћ ланчаног конструктора позивање базног конструктора апстрактне класе из изведене класе *Pqr*.

```
public abstract class Abc
{
     public int a, b;
     public Abc(int a1, int b1)
    {
        a = a1;
        b = b1;
    }
}
public class Pqr : Abc
{
    public Pqr() : base(3, 4)
    {
    }
}
```

2.3.6 Интерфејси

Интерфејс је концепт који раздваја спецификацију методе од њихове имплементације. Све методе у интерфејсу су апстрактне методе. У интерфејсу се даје спецификација методе, док се имплементација метода из интерфејса врши у класама које тај интерфејс имплементирају [4].

Поређење интерфејса и апстрактне класе [4]:

- Код интерфејса се могу дефинисати само константе. Апстрактна класа може имати атрибуте.
- Свака метода у интерфејсу има само потпис без имплементације. Апстрактна класа може да има конкретну методу.
- Пошто су све методе дефинисане у интерфејсу апстрактне методе, испред имена методе се не ставља кључна реч *abstract,* док се код апстрактних метода и апстрактних класа ставља кључна реч abstract, јер оне могу да имају и конкретне методе.
- Метода класе која реализује мотоду интерфејса мора бити јавна (*public*).
- Ни интерфејс ни апстрактна класа не могу да имају своја појављивања.

2.4 ASP.NET MVC архитектура

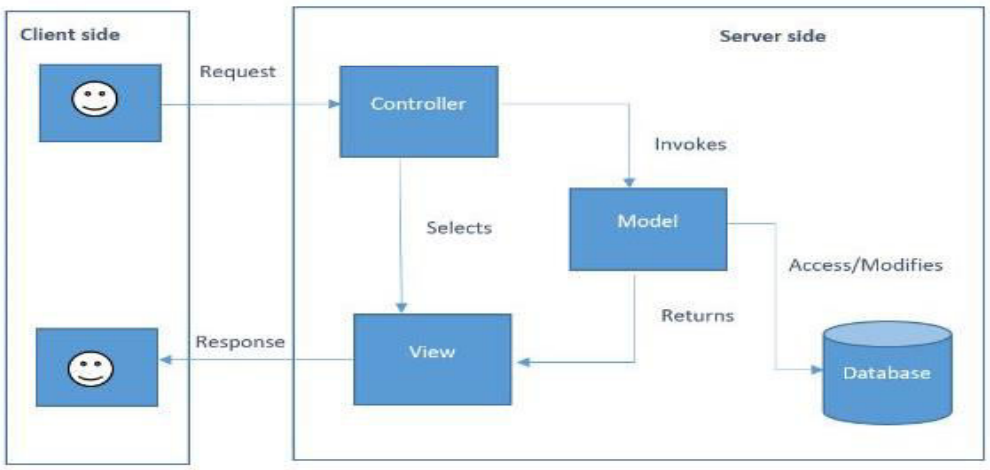
ASP.NET странице се извршавају на страни сервера и генеришу *HTTP* који се шаље претраживачима. ASP.NET користи „*event-driven*“ модел програмирања који побољшава перформансе и омогућава сепарацију корисничког интерфејса од логике апликације. ASP.NET ради на врху *HTTP* протокола користећи *HTTP* команде и правила како би омогућио обострану комуникацију између клијента и сервера. Кôд је могуће писати у *C#, VisualBasic* или *JavaScript* језику [1]. Оквир ASP.NET омогућава три методе развоја:

- *Web Forms*
- *Web Pages*
- *MVC (Model View Controller)*

Microsoft нуди две архитектуре за реализацију веб апликација, прва је *ASP .NET Web API* чија је улога да олакша развој веб апликација које имају богатог клијента и засноване су на *HTTP* сервисима, док је њена алтернатива *ASP .NET MVC* архитектура која је изузетно погодна за реализацију изузетно комплексних веб апликација са

мноштвом пословне логике у себи. *ASP .NET MVC* је имплементација *Model-View-Controller* архитектуре и функционише на принципу доделе задатака овим компонентама. Поред доделе задатака овај патерн такође дефинише начин на који ће компоненте комуницирати. Сваки од наведених објеката је одвојен од објекта другог типа апстрактних интерфејса, преко којих успостављају међусобну комуникацију [6]. Архитектуре се, независно од имплеметације, састоји од следећих слојева:

- *Model* - интерна репрезентација података и пословне логике. Он обухвата објекте и податке домена. Не зна ништа о самом корисничком интерфејсу. Захтеве за промене прима од конторлера и обавештава поглед о променама, а захтеве за читање прима од погледа [6].
- *View* - поглед, представља имплементацију корисничког интерфејса. Обухвата све елементе интерфејса и може непосредно да чита податке модела [6].
- *Controller* – контролер, логичка компонента корисничког интерфејса. Он прима обавештења о активностима корисника од погледа и иницира како активности на нивоу модела, тако и промене на поглед [6].



Слика 3 – *MVC архитектура*. (n.d.). [Digital image]. https://code-maze.com/wp-content/uploads/2018/12/mvc-architecture.jpg

Корисник преко веб претраживача где се налази интерфејс наше апликације, уноси захтев за подацима који су му потребни. Подаци се затим прослеђују до контролера који сада прослеђује захтев до модела унутар којих се налазе подаци које корисник жели. Модел зато путем низа порука враћа одговор у виду информација до контролера који сада има задатак да проследи ове информације до одговарајућег погледа. У овом процесу од наведена три елемента једино је поглед пасиван и нема никакву самосталну улогу, него приказује само оно што му је проследио контролер [6].

Као и свака архитектура тако и претходно описана MVC архитектура има своје предности и мане које су дате у наставку текста.

Предности [6]:

- модел се може приказати на више начина
- лакше је додати нови приказ података (на пример нову интернет страницу која приказује постојеће податке или део њих)
- лакша се мења интеракција са корисником
- више програмера може радити истовремено и паралелно на различитим компонентама
- могућност поновна коришћења кôда

- појединачни делови се могу лако тестирати, мењати и побољшавати
- приказ података је одвојен од логике обраде података

Мане [6]:

- превише комплексна за примену код развоја мањих апликација, што доводи до погоршања како њеног дизајна, тако и њених карактеристика
- услед честих измена модела приказ података може остати преплављен захтевима за измену, што може довести до кашњења у одговорима на захтеве

2.5 Entity Framework Core

Развој модерних информационих система у великој мери се базира на манипулацији великом количином података смештених у релационим базама података. Овакви системи морају омогућити интеграцију података из различитих извора,њихову трансформацију, смештање у базу, као и прибављање података у циљу доношења пословних одлука. Имплементација слоја података треба да омогући прихватање података са пословног слоја и њихово смештање у базу, као и прибављање података из базе и њихово прослеђиваје вишем слоју. Класична стратегија за имплементирање слоја података се заснива на директној комуникацији са базом података. Ова стратегија показала је извесне недостатке који се огледају у немогућности објектно-оријентисаних прогамских језика да у потпуности пресликају податке и релације из базе података [7]:

• Промене у структури базе, веома често проузрокују грешке у апликацији услед неслагања са базом и овакве грешке се не могу уочити након компилације програма, већ једино у току детаљног тестирања.

• Ниво програмибилности података који се захтева од савремених апликација условљава програмирање комплексних оквира, сложених за одржавање. Пример овога је имплементација механизама за опслуживање конкурентног приступа бази.

• Комплексност овог слоја је проузрокована и потребом писања стандардних упита над базом (*insert,update, delete*) који захтевају додатно време развоја.

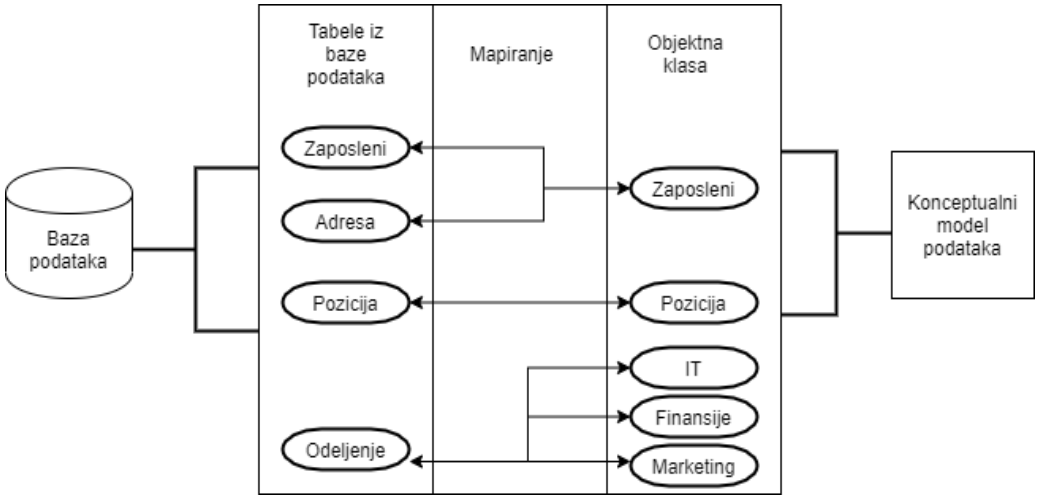
Делимично решење претходних проблема нуде алати за Објектно Релационо Мапирање (*ORM*). *Entity Framework Core* је технологија која подржава развој објектно-оријентисаних апликација. Временом постаје једна од водећих технологија за приступ подацима, а од верзије 6 доступан је изворни код (енг. *open source*) под *Apache v.2* лиценцом. Entity Framework *Core* омогућава потпуну апстракцију релационих база, креирањем објектног слоја између апликације и самих података [7].

Предности коришћења *Entity Framework Core* оквира су [7]:

- Основни разлог велике популарности лежи у његовој спостобности да велики део кода генерише аутоматски и на тај начин програмерима штеди драгоцено време и труд.
- Аутоматско ажурирање при променама објеката и лако освежавање шеме након промене структуре базе.
- Ослобођење зависности од фиксирања кода при раду са одређеним подацима или при складиштењу шема.
- Пресликавање између концептуалног модела и шеме за складиштење се може променити без промене кôда саме апликације.

Entity Framework Core је технологија са снажним акцентом на моделовање. Језгро *Entity Framework Core*-а је модел података (енг. *Entity Data Model-EDM*). Овај модел омогућава прилагођавање пресликавања између објекта класе и конкретне табеле из базе података. Уводи се са намером да се рад са подацима олакша и учини

још ефикаснијим. *EDM* омогућава програмеру да креира објекат класе који ће што прецизније одговарати домену проблема [7]. На слици је приказан пример *EDM*-а.



Слика 4 - ***Приказ пресликавања објекта класе и табеле из базе података***

На наведеној слици можемо приметити да табеле из базе података нису директно пресликане на објекте класе. Приметимо да се информације о запосленима налазе смештене у три различите табеле у бази што је, са становишта базе података, сасвим оправдан распоред података, али са становишта класе, ове информације нема потребе раздвајати у засебне класе. На пример, оправдано је да све информације о запосленом буду смештене у једну класу *Zaposleni*. Супротно овом примеру, имамо пример када се из једне табеле у бази праве три различите класе које представљају појединачне објекте. Постојаће три засебне класе објеката (*IT, Marketing, Finansije*), али приликом складиштења података информације о све три класе ће се уписивати у једну табелу *Odeljenje* у бази података. Такође, на слици се може видети пресликавање табеле *Pozicija* у једну класу *Pozicija*. Пресликавање једне табеле из базе у једну класу се је подразумевано понашање *Entity Framework Core*-а. Захваљујући пресликавању, омогућен је рад са објектима класе без размишљања о табелама из базе [7].

Entity Framework Core користи комбинацију конвенција, атрибута анотације и *Fluent API* исказа за изградњу модела ентитета у време извршавања, тако да све акције извршене на класама могу касније аутоматски да буду преведене у акције које су извршене у актуелној бази података. Класа ентитета представља ред у табели [7].

- Конвенција – за назив табеле поставља се да се поклапа са називом својства *DbSet<T>* у класи *DbContext.*
- Анотација – Одређена колона за коју није пожељно да се мапира у бази имаће анотацију [*NotMapped*] изнад одговарајућег атрибута у класи који представља овако описану колону.
- *EF Core Fluent API* – Атрибути могу да буду уклоњени из класе (атрибути анотације) и замењени Fluent API исказом у методи *OnModelBuilding* одговарајуће класе [7] :

```
modelBuilder.Entity<Proizvod>()
.Property(p => p.Naziv)
.IsRequired()
.HasMaxLength(40);
```

уместо:

```
[Required]
[StringLength(40)]
public string Naziv {get; set; }
```

Да би се употребио *EF Core*, класа којом се то омогућава јесте класа која наслеђује *DbContext*. Таква класа разуме како може да комуницира са базама података и да динамички генерише *SQL* исказе за слање упита и манипулацију подацима. Унутар ове класе дефинишу се својства која представљају табеле и њихова својства. Трансакције се имплицитно покрећу помоћу методе *SaveChanges*. Ова метода обезбеђује да се трансакција или у потпуности изврши над базом података, или трансакција неће имати никаквог ефекта и аутоматски ће се поништити све промене. Трансакције задржавају интегритет базе података применом блокада за спречавање читања и писања, док се дешава секвенца операција и подржавају *ACID* особине (атомност, конзистентност, изолација, трајност) [7].

2.5.1 LINQ језик

За рад са овим оквиром, често се користи *LINQ* (енг. *Language Intergrated Query*), језик структуираних упита за претраживање локалних колекција објекта али и удаљених извора података на такав начин да не нарушава безбедност података. *LINQ* пружа предности и проверавања исправне употребе типова у време превођења и динамичког састављања упита. Сви основни типови су дефинисани у именским просторима (енг. *namespace*) *System.Linq* и *System.Linq.Expressions* [8].

Основни концепти LINQ језика

Основне јединице података са којима *LINQ* ради јесу секвенце и елементи. Секвенца је сваки објекат који имплементира интерфејс *IEnumerable<T>,* док је елемент свака ставка секвенце [8]. У примеру који следи, низ *names* је секвенца, а *"Mike","Anna" и "Harry"* су њени елементи:

```
string[] names = { "Mike", "Anna", "Harry" };
```

Овакву секвенцу зовемо локална секвенца зато што представља локалну колекцију објеката у меморији. Оператор за упит је метода која трансформише секвенцу. Типичан оператор за упит прихвата улазну секвенцу коју трансформише у излазну секвенцу. Упити који претражују локалне секвенце зову се локални упити или *LINQ-to-object* упити. *LINQ* подржава и секвенце које се могу динамички попуњавати из одређеног удаљеног извора података као што је SQL база података. Те секвенце додатно имплементирају интерфејс *IQueryable<T>* и подржане су помоћу одговарајућег стандардног скупа оператора за упите у класи *Queryable*. Упит је израз који трансформише секвенце помоћу оператора за упите. Већина оператора за упите прихвата ламбда израз као аргумент. Ламбда израз олакшава разумевање и формирање упита [8].

Најједноставнији упит се састоји од једне улазне секвенце и једног оператора, што је приказано на следећем примеру [8]:

```
using System;
using System.Collections.Generic;
using System.Linq;
class LinqDemo
{
  static void Main()
  {
    string[] names = { "Mike", "Anna", "Harry" };
IEnumerable<string> filteredNames = names.Where (n => n.Length >= 5);
    foreach (string name in filteredNames) Console.WriteLine (name);
  }
}
//Резултат:
Harry
```

Fluent API

Стратегија састављања упита помоћу проширених метода и ламбда израза пружа велике могућности комбиновања јер омогућава уланчавање оператора за упите. Течна синтакса је флексибилна и фундаментална. Оператори за упите се могу уланчавати да би се формирали сложенији упити, апроширене методе су веома важне за тај поступак. *Where, OrderBy и Select* су стандардни оператори за упите који се преводе у проширене методе класе Enumerable. Оператор *Where* захтева да ламбда израз враћа вредност типа *bool*, која ако је *true*, значи да тај елемент треба укључити у излазну секвенцу. Оператор *OrderBy* враћа сортирану верзију своје улазне секвенце, а метода *Select* враћа излазну секвенцу у којој је сваки улазни елемент трансформисан или пројектован помоћу задатог ламбда израза. Подаци теку ланцем оператора слева надесно, тако да се подаци прво филтрирају, затим сортирају, а онда пројектују. Овако изгледају потписи поменутих проширених метода [8]:

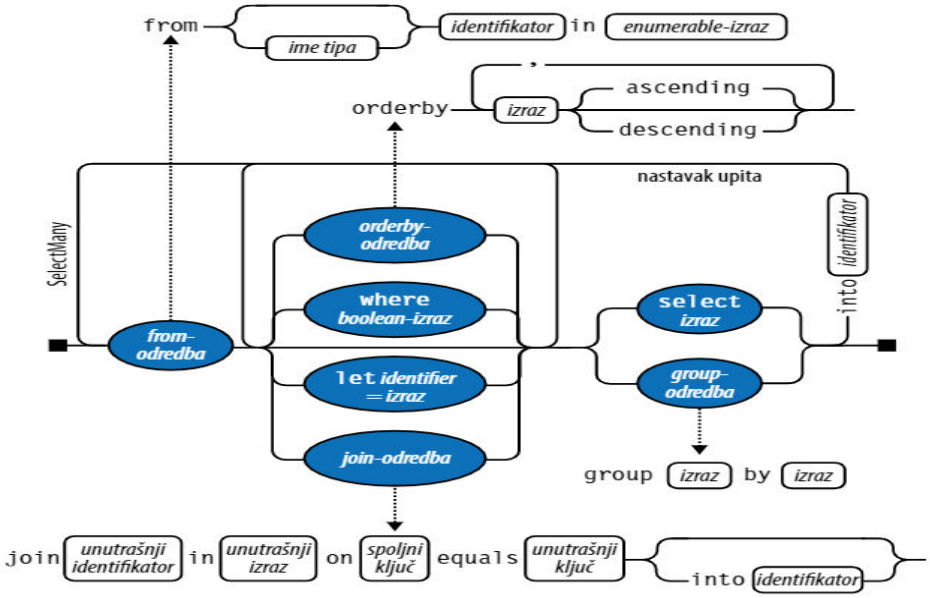
```
public static IEnumerable<TSource> Where<TSource>
  (this IEnumerable<TSource> source, Func<TSource,bool> predicate)

public static IEnumerable<TSource> OrderBy<TSource,TKey>
  (this IEnumerable<TSource> source, Func<TSource,TKey> keySelector)

public static IEnumerable<TResult> Select<TSource,TResult>
  (this IEnumerable<TSource> source, Func<TSource,TResult> selector)
```

Када се оператори за упите уланчавају као у овом примеру, излазна секвенца једног оператора јесте улазна секвенца оператора који му следи. Сврха ламбда израза зависи од конкретног оператора за упите. У случају оператора *Where*, ламбда израз одређује да ли елемент треба укључити у излазну секвенцу. У случају оператора *OrderBy*, ламбда израз пресликава сваки елемент у улазној секвенци на његово место по кључу за сортирање. Уз оператор *Select*, ламбда израз одређује како се сваки елемент улазне секвенце трансформише пре него што се уметне у излазну секвенцу [8].

Изрази упита

Компајлер обрађује израз упита тако што га преводи у течну синтаксу. Све што се може написати у облику израза упита, може се написати и у облику течне синтаксе. Наведени пример упита компајлер преводи прво у следећи облик [8]:

```
IEnumerable<string> query = names.Where   (n => n.Contains (“a”))
                                 .OrderBy (n => n.Length)
                                 .Select  (n => n.ToUpper());
```



Слика 5 - *Синтакса за упите*. (n.d.). [Digitlal image]. https://docplayer.rs/142364469-8-linq-upiti-linq-language-integrated-query-upit-integrisan-u-jezik-jeste-skup-mogu%C4%87nosti-koje-jezik-c-i-framework-pru%C5%BEaju-za-pisanje-strukturirani.html

Операторе *Where, OrderBy* и *Select* затим разрешава према истим правилима која би важила када би упит био написан у течној синтакси. У овом случају, они се претварају у позиве одговарајућих проширених метода класе *Enumerable* [8].

Упити над базом података

Упити над базом података *Entity Framework Core* - у реализују се преко својства *DbSet<T>* класе *DbContext.*

First и FirstOrDefault

First и *FirstOrDefault* методе су намењене за потребе враћања једног резултата. Уколико се очекује бар један запис који одговара критеријуму којих може бити више, користи се метода *First*. Уколико постоји могућност да не постоји запис који одговара критеријуму користи се *FirstOrDefault* метода, која ће вратити *null* као подразумевану вредност у случају да није пронађен запис по одређеном критеријуму или критеријумима. Обе методе резултирају тренутним извршавањем упита, што значи да се *SQL* упит генерише и извршава над базом података чим дође до позива методе. У наставку је дат пример коришћења поменуте методе и одговарајући *SQL* упит који се у позадини окида [8]:

```
        var author = context.Authors.First();
SQL упит:

        SELECT TOP(1) [a].[AuthorId], [a].[FirstName], [a].[LastName]
        FROM [Authors] AS [a]
```

Single и SingleOrDefault

Single и *SingleOrDefaul*t методе се користе за враћање једног записа где само један запис треба да одговара критеријумима који су наведени. Ова метода у позадини генерише *SELECT TOP(2*) упит и уколико се упитом врати више резултата, генерише се *InvalidOperationException* изузетак са поруком: *Sequence contains more than one element*. У наставку је дат пример коришћења поменуте методе и одговарајући *SQL* упит који се у позадини окида [8]:

```
        var author = context.Authors.Single(a => a.AuthorId == 1);
SQL упит:
        SELECT TOP(2) [a].[AuthorId], [a].[FirstName], [a].[LastName]
        FROM [Authors] AS [a]
        WHERE [a].[AuthorId] = 1
```

Препорука је користи *SingleOrDefault* методу која ће вратити null уколико постоји могућност да се упитом не пронађе ниједан запис који одговара критеријумима. *DbSet.Find* метода узима кључ (кључеве) ентитета. Ова метода представља скраћеницу (*syntactic sugar*) методе *SingleOrDefault* и могуће је да ова метода врати *null* вредност [8].

```
var author = context.Authors.Find(1);
```

Include

Include метода се користи за жељено учитавање повезаних података. Користи се преко навигационог пропертија приказаног на следећем примеру:

```
var authors = context.Authors.Include(a => a.Books).ToList();
```

На овај начин из базе добијају се подаци о аутору као и о свим књигама из колекције које ентитет поседује.

За учитавање података користи се такође метода *ThenInclude*. Ова метода је корисна када постоји сценарио вишеструких нивоа повезаности ентитета, на пример уколико одређена класа садржи колекцију, а потом сваки члан те колекције такође поседује колекцију . У овом примеру из базе се добија резултат који укључује податке о студенту, његовим оценама и наставницима за сваку оцену. Упит који се генерише над базом података је *SELECT* упит који обавезно користи *JOIN* клаузулу [8].

```
        var student = context.Students.Where(s => s.FirstName == "Bill")
                              .Include(s => s.Grade)
                                  .ThenInclude(g => g.Teachers)
                              .FirstOrDefault();
SQL упит:

        SELECT TOP(1) [s].[StudentId], [s].[DoB], [s].[FirstName], [s].[GradeId], [s].[LastName],
                 [s].[MiddleName], [s.Grade].[GradeId], [s.Grade].[GradeName], [s.Grade].[Section]
        FROM [Students] AS [s]
        LEFT JOIN [Grades] AS [s.Grade] ON [s].[GradeId] = [s.Grade].[GradeId]
        WHERE [s].[FirstName] = N'Bill'
        ORDER BY [s.Grade].[GradeId]
        Go

        SELECT [s.Grade.Teachers].[TeacherId], [s.Grade.Teachers].[GradeId], [s.Grade.Teachers].[Name]
        FROM [Teachers] AS [s.Grade.Teachers]
        INNER JOIN (
            SELECT DISTINCT [t].*
            FROM (
                SELECT TOP(1) [s.Grade0].[GradeId]
                FROM [Students] AS [s0]
                LEFT JOIN [Grades] AS [s.Grade0] ON [s0].[GradeId] = [s.Grade0].[GradeId]
                WHERE [s0].[FirstName] = N'Bill'
                ORDER BY [s.Grade0].[GradeId]
            ) AS [t]
        ) AS [t0] ON [s.Grade.Teachers].[GradeId] = [t0].[GradeId]
        ORDER BY [t0].[GradeId]
        Go
```

NoTracking упити

Било који ентитет који се враћа помоћу упита се аутоматски прати од стране *context* класе која представља сесију са базом. У случају када је податке потребно користити само за сврху читања (енг. *read-only*) или само за

приказ, није неопходно да овакви ентитети буду праћени јер њихов садржај не желимо да променимо током захтева. Због тога се користи метода *AsNoTracking* [8]:

```
var cars = context.Cars.AsNoTracking().ToList();
```

Овом методом се спречавају непожељна сценарија и побољшавају перформансе апликације.

Додавање новог ентитета

Помоћу методе *Add* додаје се нови запис у базу. Најчешће се користи следећи запис за додавање новог ентитета:

```
var author = new Author{ FirstName = "William", LastName = "Shakespeare" };
context.Authors.Add(author);
```

Context класа започиње праћење ентитета који је прослешен методи Add и указује на то да тренутно стање овог ентитета, тј. *EntityState* треба да буде постављено на *Added.* Позивом методе *SaveChanges*, сви ентитети са стањем *Added* се преко *SQL Insert* упита уносе у базу [8].

Измена ентитета

Помоћу методе *Update* врши се измена одговарајућег ентитета из базе. Најчешће се користи следећи запис за измену ентитета:

```
        public void Save(Author author)
        {
        var author = context.Authors.Find(1);
        author.FirstName = "Bill";
        context.Authors.Update(author);
        context.SaveChanges();
        }
```

SQL упит:

```
        exec sp_executesql N'SET NOCOUNT ON;
        UPDATE [Authors] SET [FirstName] = @p0
        WHERE [AuthorId] = @p1;
         SELECT @@ROWCOUNT;
        ',N'@p1 int,@p0 nvarchar(4000)',@p1=1,@p0=N'Bill'
```

Наведена метода резултира тиме да context класа која представља сесију са базом сада прати стање овог ентитета које се поставља на *Modified*. *SQL* упит који се генерише извршиће *UPDATE* операцију над сваким атрибутом ентитета. Сви повезани ентитети са ентитетом над којим се врши ова операција, као што су колекције које садржи ентитет, постају праћене и имају *Modified* стање, што значи да ће се *UPDATE* операција одразити и на ове записе у бази података. Уколико неком повезаном ентитету није већ додељен кључ, његово стање биће постављено на *Added* и извршиће се операција *INSERT* уместо *UPDATE* [8].

Брисање ентитета

Помоћу методе *Remove* врши се брисање одговарајућег ентитета из базе. Најчешће се користи следећи запис за брисање ентитета:

```
context.Authors.Remove(context.Authors.Find(1));
context.SaveChanges();
```

У овом примеру, наведеном методом се стање ентитета поставља на *Deleted*. Када се позове метода *SaveChanges*, генерише се операција *DELETE*. На овај начин извршавају се два упита над базом података, један за добијање ресурса и други за брисање истог.

```
exec sp_executesql N'SET NOCOUNT ON;
DELETE FROM [Authors]
WHERE [AuthorId] = @p0;
SELECT @@ROWCOUNT;
',N'@p0 int',@p0=1
```

На сличан начин као и код измене ентитета, уколико је ентитет који се брише из базе повезан путем спољног кључа са другим ентитетима, операција брисања ће се одразити и на те ентитете зависно од тога како је ова веза конфигурисана. Ова конфигурација односи се на референцијални интегритет и могуће је подесити је у самој бази података директно или путем *Fluent API - ja*. Подразумевана акција јесте *NoAction* и тада је потребно ручно се побринути за повезане ентитете [8]. Следећи пример показује класу *Author* која има колекцију књига и класу *Book* којој није постављен проперти као спољни кључ:

```
public class Author
{
   public int AuthorId { get; set; }
   public string FirstName { get; set; }
   public string LastName { get; set; }
   public ICollection<Book> Books { get; set; }
}

public class Book
{
   public int BookId { get; set; }
   public string Title { get; set; }
}
```

EF Core уводи својство названо *shadow property* за представљање спољног кључа [8]. У примеру то је *AuthorId* и примењује се на ентитет *Book*. Уколико постоји намера за брисањем ентитета који представља класу *Author*, прво је потребно обрисати везу са сваким од ентитета из колекције *Books* жељеног аутора. То се постиже на следећи начин:

```
var author = context.Authors.Find(1);
var books = context.Books.Where(b => EF.Property<int>(b, "AuthorId") == 1);
foreach (var book in books)
{
    author.Books.Remove(book);
}
context.Authors.Remove(author);
context.SaveChanges();
```

Упит који се извршава над базом за сваки ентитет класе *Book* поставља *AuthorId* на null.

```
exec sp_executesql N'SET NOCOUNT ON;

UPDATE [Books] SET [AuthorId] = @p0
WHERE [BookId] = @p1;
SELECT @@ROWCOUNT;
UPDATE [Books] SET [AuthorId] = @p2
WHERE [BookId] = @p3;
SELECT @@ROWCOUNT;
UPDATE [Books] SET [AuthorId] = @p4
WHERE [BookId] = @p5;
SELECT @@ROWCOUNT;
',N'@p1 int,@p0 int,@p3 int,@p2 int,@p5 int,@p4 int',@p1=1,@p0=NULL,@p3=2,@p2=NULL,@p5=3,@p4=NULL
```

```
exec sp_executesql N'SET NOCOUNT ON;
DELETE FROM [Authors]
WHERE [AuthorId] = @p6;
SELECT @@ROWCOUNT;
',N'@p6 int',@p6=1
```

У овом примеру направљена су четири позива ка бази података: *SELECT* за добијање аутора*; SELECT* за добијање књига; *UPDATE* за измену ентитета књига и *DELETE* за брисање аутора.

2.6 *Bootstrap*, *JavaScript*, *jQuery* и *AJAX*

Начин преношења садржаја и жељене поруке је битан аспекат успешног маркетинга производа или услуге. У савременом глобалном пословном окружењу, купци производа и корисници услуга су све захтевнији и њихову пажњу и оданост временом постаје све теже придобити. Убрзан и динамички развој веб-а у правцу све бржих и оптимизованијих сајтова, прилагођених екранима мобилних телефона и осталих паметних уређаја, допринео је да управо JavaScript, Bootstrap, jQuery и Ajax технологије доживе невероватну експанзију.

2.6.1 *Bootstrap*

Bootstrap је веб оквир који значајно поједностављује развој респонзивних веб страница. Главна сврха додавања *Bootstrap*-а у пројекат је могућност коришћења великог броја шаблона за типографију, форме, навигацију и друге елементе графичког корисничког интерфејса. Крајњи резултат је уједначен изглед прозора, табела и форми на различитим веб претраживачима. Поред тога, програмери имају могућност додатног модификовања *Bootstrap*-ових *CSS* класа, како би прилагодили изглед садржаја. *Bootstrap* такође има неколико *JavaScript* компоненти у облику *jQuery* додатака, који нуде могућност додавања додатних функционалности на елементе корисничког интерфејса. Свака компонента *Bootstrap*-а се састоји од *HTML* структуре, *CSS* декларација и, у неким случајевима, *JavaScript* кода. Најважније компоненте су оне које се односе на распоред елемената на веб страници. За контролисање распореда *Bootstrap* користи грид систем који се састоји од дванаест колона. Три основне компоненте распореда су *Container, Row и Col. Container* компонента се користи као омотач за све друге компоненте на веб страни. *Row* се користи као омотач за колоне. Изглед апликације се гради као низ редова и колона. Све остале *HTML* компоненте се смештају унутар ових, а *Bootstrap* се сам брине за њихову респонзивност. Прекомпајлирана верзија *Bootstrap*-а је доступна у облику *CSS* и *JavaScript* фајлова које је једноставно додати било ком пројекту [9].

2.6.2 *JavaScript*, *jQuery* и *AJAX*

Основну примену JavaScript програмски језик има у области веб програмирања, пре свега, због једне од својих главних особина која JavaScript-у омогућава функционисање на свим познатијим веб претраживачима, као што су Internet Explorer, Mozilla, Chrome, Firefox, Netscape и Opera претрачивачи.

Са друге стране, AJAX представља групу технологија за веб развој која је намењена искључиво креирању интерактивних апликација за веб. Комбинацијом JavaScript и AJAX-а уз коришћење једне од најпопуларнијих JavaScript библиотека - jQuery, у потпуности су искоришћене све предности JavaScript језика уз постизање респонзивности веб апликација.

JavaScript

JavaScript је објектно базиран, платформски неутралан, вишекорисnички језик који омогућава већу функционлност на клијентској страни. Он се доминантно користи при изради веб страница уз *HTML* и *CSS*. Реч

је о програмском језику који омогућава прављење интерфејса који омогућавају корисницима високу функционалност без потребе за новим учитавањем странице. Брзина и мала меморијска потрошња *JavaScript*-а у поређењу са осталим програмским језицима доприноси томе да он има све више различитих примена. Овај програм служи за програмирање задатака у апликацијама, али и за развој *Internet of Things* апликација [10].

JavaScript је настао 1995. године, инспирисан Јава програмским језиком. У то време није постојао данас најпопуларнији претраживач *Google Chrome*. Шта више, компанија *Google* ће бити основана тек три године касније. Тада су интернет конекције биле знатно скромнијих карактеристика. То значи да слање упита серверској страни за било који клијентски унос уопште није било практично. Због тога је појава *JavaScript* донела нове могућности обраде података на клијентској страни, што је знатно побољшало перформансе тадашњих интернет сајтова [10].

Једна од најважнијих могућности јесте чување података на уређају корисника. То значи да ће скрипта радити чак и без интернета јер су сви подаци смештени на локални рачунар. То је значајна разлика у који је серверски оријентисане језике. Ипак, за покретање кôда потребна је клијентска апликација. То су претраживачи попут Опере. У том случају од самог клијента зависи да ли ће се код уопште извршити. Наиме, сви модерни претраживачи имају ту специфичност онемогућавања ових скрипти [10].

Због својих специфичних особина, *JavaScript* се у великој мери користи при креирању веб страница. Он нуди неколико кључних предности у односу на неке друге програмске језике [10]:

- Висок ниво интеракције - могуће је креирати садржај који се активира када корисник пређе мишем преко њега или притсне дугме на тастатури;
- Могућност рада у локалу - с обзиром на то да је језик оријентисан ка клијенту, комуникација са сервером сведена је на минимум;
- Богатији интерфејс - омогућава комплекснију структуру странице, као и извршавање специфичних наредби попут превлачења садржаја;
- Тренутне повратне информације - корисник не мора поново да учитава страну да би видео промену коју је унео.

Ово су само неке од могућности које нуди *JavaScript*, али треба нагласити да неке операције намерно нису могуће због безбедности.

Синтакса је скуп основних правила која дефинишу на који се начин пишу програми у оквиру неког програмског језика. По синтакси, језгро овог програмског језика слично је језицима C++, C, C# и Јава језику, јер садржи конструкције као што су петље и оператори. Међутим, синтакса је једина сличност коју је могуће уочити. *JavaScript* је слабо типизиран језик, што значи да се за променљиве не мора дефинисати тип. Објекти у *JavaScript* програмском језику пресликавају (мапирају) имена својстава у произвољне вредности својстава. Због овога, типизирани језици су сличнији хеш табелама него објектима (у C# језику) [10].

JavaScript користи неколико основних података који омогућавају рад. То су [10]:

- Бројеви - бројеви као тип података;
- Логичке вредности - постоје само две могућности: тачно или нетачно; ове вредности се обично користе приликом неке контроле и тестирања;
- Знаковни низови - служи за представљање текста, тј. за складиштење слова, бројева и других знакова;
- Нул (енг. *null*) специјална вредност - указује на непостојање било какве друге конкретне вредности; то значи да вредност постоји, али да она није ни број, ни текст, али ни логичка вредност;
- Специјална вредност *undefined* - користи се при покушају приступања вредности неке променљиве која је дефинисана, али јој није додељена никаква вредност.

Осим наведених, овај језик има уграђену подршку за низове, датуме и регуларне изразе. Функције *JavaScript*-а се пишу унутар *<script></script>* ознака или у посебном документу са *.js* екстензијом. Свака функција почиње резервисаном речју *function* иза које следи име функције, заграде за аргумене и на крају код. Позивање ових функција се може остварити на више начина. Неки од најчешће коришћених су кликом на неки објекат, приликом учитавања, приликом промене неког параметра или неком другом акцијом. *JavaScript* је програмски језик који се извршава на страни клијента. Најчешће се користи у претраживачима, па се језгро опште намене проширује објектима који омогућавају скриптама интеракцију са корисником, управљање претраживачем и измену садржаја документа који се појављује унутар прозора претраживача. Ова уграђена (енг. *embedded*) верзија *JavaScript* језика изршава скрипте које су придружене *HTML* документима. Неки делови *JavaScript* програмског језика званично су стандардизовани, а остали представљају проширења која зависе од претраживача. Компатибилност између различитих претаживача значајна је за програмере који користе клијентски *JavaScript* [10].

jQuery

jQuery је *JavaScript* библиотека чија је сврха да олакша учитавање *JavaScript* кода на веб страницама. Наиме, велики број сајтова користе ову *JavaScript* библиотеку како би олакшали рад. Уместо да се пишу хиљаде линија кода, довољно је написати једну линију уз *jQuery*. За ову библиотеку креирано је доста додатака (енг. *plug - in*) опција који олакшавају процес рада. Писан у *JavaScript* језику и налази се у облику једног.js фајла који је могуће линковати са веб странице након чега *JavaScript* код приступа библиотеци позивајући разне *jQuery* функције. Ова библиотека долази у два облика [10]:

• некомпресовани *.js* фајл, класично снимање на локални диск - прост за читање и модификацију, али има око 160 килобајта величине.

• смањени .js фајл, односно *CDN* (енг. *Content Delivery Network*)- сви коментари и други непотребни карактери су уклоњени из фајла, чиме је целокупна библиотека смањена на 20-ак килобајта. Иако је тешка за читање, ово је верзија коју треба поставити на сајт ради што бржег учитавања стране.

jQuery је бесплатан за скидање и употребу. Осим што је доступна *jQuery* библиотека, доступно је и на стотине бесплатних *jQuery* додатака. Помоћу библиотеке и додатака *jQuery* омогућава [10]:

• Додавање ефеката тексту

• Прављање *xml* (*AJAX*) упита за тражење додатних информација од веб сервера без потребе за освежавањем странице

• Једноставно додавање, уклањање и премештање садржаја веб странице помоћу пар редова кода

• Прављење менија са подменијем и моћнијих формулара

• Креирање интерфејса тако да на страници елементи могу да се премештају једноставним превлачењем

jQuery је компатибилан са бројним претраживачима и представља библиотеку која апстрахује понашања различитих претраживача у кратку и концизну синтаксу.

AJAX

AJAX (енг. *Asynchronous JavaScript And XML*) је заснован на *JavaScript* и *HTTP* захтевима и његова главна особина је да омогући да се на неком сајту шаље и прима садржај без поновног учитавања странице. *AJAX* није

нови језик програмирања, већ техника за креирање бољих, бржих и интерактивнијих веб апликација. Са *AJAX*-ом, *JavaScript* код може да комуницира директно са сервером, користећи *JavaScript XMLHTTPRequest* објекат. Помоћу овог објекта, могуће је размењивати податке са веб сервером, без поновног учитавања странице. *AJAX* користи асинхрони трансфер података (*HTTP* захтеви) између претраживача и сервера, чиме даје могућност веб апликацији да обрађује само оне податке који су заиста потребни за ту сврху, уместо читавих страница. Веб стандарди који се користе у *AJAX*-у су добро дефинисани и подржани од свих великих претраживача [10].

AJAX је заснован на следећим веб стандардима [10]:

• *JavaScript*
• XML
• *HTML*
• *CSS*

Веб апликације имају много предности над десктоп апликацијама, а основна је да оне лако могу да допру до више корисника, лакше су за инсталирање и обезбеђивање подршке, и лакше за развијање. Са *AJAX*-ом, интернет апликације могу да буду богатије и брже за коришћење него што су биле до сада. У "класичном" програмирању, ако да би добили било какву информацију из базе података или неког фајла који се налази на серверу, треба направити *HTML* форму и преко *GET* или *POST* методе добити или послати податке серверу, што би значило да након што се пошаље/добије информација од сервера , учита опет цела страна како би резултати били видљиви. Сервер враћа нову страницу сваки пут када корисник пошаље неке податке, што традиционалне веб апликације чини спорим. Помоћу *AJAX*-а, *JavaScript* комуницира директно са сервером, а помоћу *HTTP* захтева, веб страница шаље захтев серверу и добије одговор од њега без поновног учитавања веб странице. Корисник готово да и неће приметити да се у позадини послати подаци серверу [10].

3. Студијски пример

У овом поглављу биће представљен студијски пример развоја дела софтверског система у .NET окружењу. Пројекат је реализован коришћењем Ларманове методе. Софтверски систем се састоји од атрибута и системских операција. Атрибути описују структуру система, док системске операције описују понашање система. Атрибути представљају концепте реалног система, док системске операције представљају основне функције система [11].

3.1 Ларманова метода

Ларманова метода [11] за развој софтвера се заснива на итеративно-инкременталном моделу животног циклуса софтвера. Упрошћена Ларманова метода се састоји од следећих фаза, и то:

1. Прикупљање захтева
2. Анализа
3. Пројектовање
4. Имплементација
5. Тестирање

У фази прикупљања захтева се дефинишу захтеви које пројекат треба да задовољи и ова фаза се заснива на блиској сарадњи корисника и програмера. Захтеви се описују помоћу *UML* модела случаја коришћења. Модел случаја коришћења се састоји од скупа случаја коришћења, актора и веза између случаја коришћења и актора. Случај коришћења описује скуп сценарија, односно скуп жељених случајева коришћења система од стране актора. Случај коришћења има један основни и више алтернативних сценарија. Сценарио представља секвенцу акције које описују интеракцију актера и система. Једну акцију сценарија може извршити или актер или систем, па их у зависности од тога можемо поделити на следећи начин:

Акције које изводи актер су [3]:

- АПУСО - Актер припрема улазне аргументе за системску операцију
- АПСО - Актер позива систем да изврши системску операцију
- АНСО - Актер извршава несистемску операцију Акције које изводи систем су:
- СО - Систем извршава системску операцију
- ИА - Излазни аргументи који представљају резултате извршења системске операције

Након прикупљања захтева следи фаза анализе у којој се описује пословна логика софтверског система, односно његова структура и понашање. Структура софтверског система се описује помоћу концептуалног и релационог модела, док се понашање софтверског система описује помоћу системских дијаграма секвенци и уговора о системским операцијама. Системски дијаграм секвенци се прави за сваки од претходно утврђених случајева коришћења и он приказује догађаје у одређеном редоследу. Као резултат анализе системских дијаграма секвенци идентификују се системске операције које треба пројектовати. За сваку од уочених системских операција праве се уговори. Један уговор везан је за једну системску операцију. Уговори се састоје од следећег [11]:

- Операција - име операције и њени улазни и излазни аргументи
- Веза са СК - имена СК у којима се позива системска операција
- Предуслов - пре извршења СО морају бити задовољени одређени предуслови, односно систем мора бити у одговарајућем стању
- Постуслов - после извршења СО у систему морају бити задовољени одређени постуслови, што значи да систем мора бити у одговарајућем подстању или се поништава резултат операције

Структура софтверског система се описује помоћу концептуалног модела који описује концептуалне класе домена проблема. Концептуални модел садржи концептуалне класе, које се називају доменски објекти, и асоцијације између концептуалних класа [11].

Фаза пројектовања описује архитектуру софтверског система. Обухвата пројектовање апликационе логике, складишта података и корисничког интерфејса. Након тога следи фаза имплементације која подразумева имлпементирање свих компоненти добијених пројектовањем архитектуре система [11].

Софтверски систем који је предмет овог завршног рада развијен је у C# програмском језику и коришћен је *VisualStudio 2019* као развојно окружење. Последња фаза Ларманове методе је фаза тестирања. Срж ове фазе јесте да се испита и провери да ли добијени систем задовољава почетне захтеве. Како би се валидно истестирао систем неопходно је унети како тачне, тако и погрешне податке. Пословна логика која је део софтверског система овог завршног рада тестирана је коришћењем Mock фрејмворка. На тај начин се проверава како систем реагује на грешке и да ли је његово понашање у складу са очекиваним.

4. Кориснички захтеви

Прикупљање захтева од корисника је први, можда и најважнији, корак у процесу развоја софтвера Ларамановом методом, јер ако се корисникове потребе не идентификују на прави начин може доћи до проблема у каснијим фазама развоја. Већ у овој фази потребно је уочити до каквих све проблема може доћи у каснијем развоју и одредити прави модел података који би задовољио потребе система [11].

4.1 Вербални опис модела

Софтверски систем за праћење рада центра за обуку паса представља пословни систем преко кога инструктори ангажују и оцењују псе на одређеним задацима за потребе Војске Србије.

У војсци се пси обучавају за једну од пет обука коју воде професионални војници, инструктори. Заступљене су обуке за следеће службе: трагачка, заштитна, чуварска, проналажење опојних средстава и проналажење експлозива. Пси се за потребе војске шаљу на различите терене како би обављали различите задатке. Након завршетка задатка, ови пси добијају оцене за дато ангажовање на задатку.

У систему постоје постоје две улоге, админ и инструктор. Админ управља овим системом и мора бити пријављен на систем како би му био омогућен приступ бази података и осталим функционалностима система. Његови задаци обухватавају администраторске задатаке попут регистрације инструктора, измене података о њима, управљање улогама и привилегијама у систему и унос новог задатка. Унос новог задатка представља привилегију која може бити додељена неком инструктору. Админ има све привилегије у овом систему и може вршити унос, измену и брисање било ког пса у систему и било ког ангажовања у систему.

Инструктору, као и админу је дозвољено уношење новог пса када се за то укаже прилика. Инструктор може изменити податке о свом профилу и вршити манипулацију подацима само за оне псе који су на обуци коју спроводи инструктор који је тренутно пријављен на систем. Такође, овај принцип важи и за оцену ангажовања паса. Посебно, уколико има привилегију која му омогућава унос задатка, инструктор може за задатак ангажовати и псе који су и на другим обукама.

Приликом креирања задатка врши се избор паса који ће бити ангажовани на овом задатку. За један задатак могуће је ангажовати више паса. Након што је задатак завршен, може се приступити оцењивању ангажовања. У систему се прати статистика о успешности обављених задатака која показује однос између укупног броја ангажованих паса и укупног броја паса који су успешно обавили задатак.

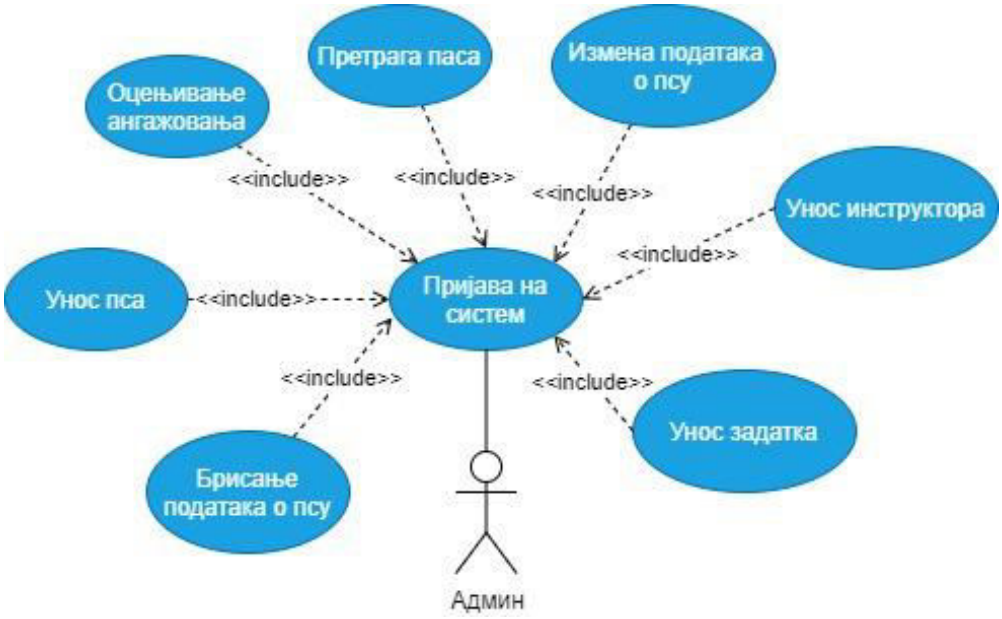
4.2. Спецификација захтева помоћу модела случајева коришћења

Админу, као кориснику система, треба омогућити следеће функционалности:

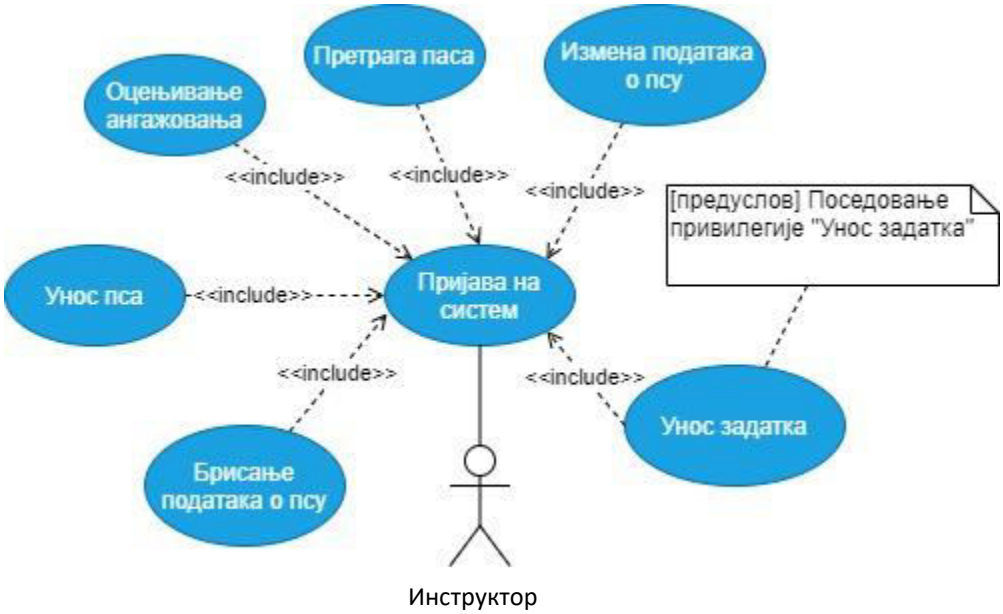
- пријаву на систем,
- унос новог пса,
- претрагу паса,
- брисање података о псу,
- измену података о псу
- унос новог инструктора,
- унос новог задатка,
- оцењивање ангажовања завршеног задатка

Инструктор користи софтверски систем за оцењивање ангажовања и потребно му је обезбедити следеће функционалности:

- пријаву на систем,
- унос новог пса,
- претрагу паса,
- брисање пса,
- оцењивање ангажовања завршеног задатка



Слика 6 - ***Дијаграм случајева коришћења, админ рола***



Слика 7 - ***Дијаграм случајева коришћења, инструктор рола***

СК1: Случај коришћења – Пријављивање на систем

Назив СК
Пријављивање на систем

Актори СК
Корисник (администратор или инструктор)

Учесници СК
Корисник и систем (програм)

Предуслов: Систем је укључен. Систем приказује форму за пријављивање на систем.

Основни сценарио СК
1. Корисник **уноси** корисничко име и лозинку. (АПУСО)
2. Корисник **контролише** да ли је коректно унео корисничко име и лозинку. (АНСО)
3. Корисник **позива** систем да се улогује (провери податке). (АПСО)
4. Систем **проверава** податке о кориснику. (СО)
5. Систем **приказује** кориснику почетну страну и поруку: "Успешно пријављивање на систем! ". (ИА)

Алтернативна сценарија
5.1. Уколико систем не може да нађе корисника, он приказује кориснику поруку: " Неуспешно пријављивање на систем!". (ИА)

СК2: Случај коришћења – Унос новог пса

Назив СК
Унос новог пса

Актори СК
Корисник (администратор или инструктор)

Учесници СК
Администратор и систем (програм)

Предуслов: Систем је укључен и корисник је пријављен на систем под својом шифром. Систем приказује форму за рад са псом. Учитана је листа свих раса и листа обука.

Основни сценарио СК

1. Корисник **уноси** податке о псу. (АПУСО)
2. Корисник **контролише** да ли је коректно унео податке о псу. (АНСО)
3. Корисник **позива** систем да запамти податке о псу. (АПСО)
4. Систем **памти** податке о псу. (СО)
5. Систем **приказује** кориснику поруку: "Пас је сачуван!". (ИА)

Алтернативна сценарија
5.1. Уколико систем не може да запамти податке о псу, он приказује кориснику поруку: "Грешка приликом уноса података о псу!". (ИА)

СК3: Случај коришћења – Претраживање паса

Назив СК
Претраживање паса

Актори СК
Корисник (админ или инструктор)

Учесници СК
Корисник и систем (програм)

Предуслов: Систем је укључен и корисник је пријављен на систем под својом шифром. Систем приказује форму за рад са псима. Учитана је листа обука.

Основни сценарио СК
1. Корисник **уноси** вредност по којој претражује псе. (АПУСО)
2. Корисник **позива** систем да нађе псе по задатој вредности. (АПСО)
3. Систем **тражи** псе по задатој вредности. (СО)
4. Систем **приказује** кориснику листу паса. (ИА)

Алтернативна сценарија
4.1. Уколико систем не може да нађе псе, систем приказује кориснику поруку: “Ниједан пронађен ниједан пас по унетом критеријуму претраге”. (ИА)

СК4: Случај коришћења – Брисање података о псу

Назив СК
Брисање података о псу

Актори СК
Корисник (админ или инструктор)

Учесници СК
Корисник и систем (програм)

Предуслов: Систем је укључен и корисник је пријављен на систем под својом шифром. Систем приказује форму за рад са псима. Учитана је листа обука.

Основни сценарио СК
1. Корисник **уноси** вредност по којој претражује псе. (АПУСО)
2. Корисник **позива** систем да нађе псе по задатој вредности. (АПСО)
3. Систем **тражи** псе по задатој вредности. (СО)
4. Систем **приказује** кориснику листу паса. (ИА)
5. Корисник **бира** пса којег жели да обрише. (АПУСО)
6. Корисник **позива** систем да учита податке о одабраном псу. (АПСО)
7. Систем **учитава** податке о одабраном псу. (СО)
8. Систем **приказује** кориснику податке о псу. (ИА)

9. Корисник **позива** систем да обрише пса. (АПСО)
10. Систем **брише** пса. (СО)
11. Систем **приказује** кориснику поруку: “Успешно обрисани подаци о псу! ”. (ИА)

Алтернативна сценарија
4.1. Уколико систем не може да нађе псе, он приказује кориснику поруку: “ Није пронађен ниједан пас по унетом критеријуму претраге!”. Прекида се извршење сценарија. (ИА)

8.1. Уколико систем не може да учита пса, он приказује кориснику поруку: “Пас чије податке захтевате није пронађен!”. Прекида се извршење сценарија. (ИА)

11.1. Уколико систем не може да обрише пса, он приказује кориснику поруку: “Грешка приликом брисања пса!”. (ИА)

СК5: Случај коришћења – Измена података о псу

Назив СК
Измена података о псу

Актори СК
Корисник (администратор или инструктор)

Учесници СК
Корисник и систем (програм)

Предуслов: Систем је укључен и корисник је пријављен на систем под својом шифром. Систем приказује форму за рад са псима. Учитана је листа обука.

Основни сценарио СК
1. Корисник **уноси** вредност по којој претражује псе. (АПУСО)
2. Корисник **позива** систем да нађе псе по задатој вредности. (АПСО)
3. Систем **тражи** псе по задатој вредности. (СО)
4. Систем **приказује** кориснику листу паса. (ИА)
5. Корисник **бира** пса чије податке жели да измени. (АПУСО)
6. Корисник **позива** систем да учита податке о одабраном псу. (АПСО)
7. Систем **учитава** податке о одабраном псу. (СО)
8. Систем **приказује** кориснику податке о псу. (ИА)
9. Корисник **уноси (мења)** податке о псу. (АПУСО)
10. Корисник **контролише** да ли је коректно унео податке о псу. (АНСО)
11. Корисник **позива** систем да запамти податке о псу. (АПСО)
12. Систем **памти** податке о псу. (СО)
13. Систем **приказује** кориснику поруку: “Подаци о псу успешно измењени!”. (ИА)

Алтернативна сценарија
4.1. Уколико систем не може да нађе псе, он приказује кориснику поруку: “ Није пронађен ниједан пас по унетом критеријуму претраге!”. Прекида се извршење сценарија. (ИА)

8.1. Уколико систем не може да учита пса, он приказује кориснику поруку: “Пас чије податке захтевате није пронађен!”. Прекида се извршење сценарија. (ИА)

11.1. Уколико систем не може да сачува податке о псу, он приказује кориснику поруку: “Грешка приликом чувања података о псу!”. (ИА)

СК6: Случај коришћења – Унос новог инструктора

Назив СК
Унос новог инструктора

Актори СК
Админ

Учесници СК
Админ и систем (програм)

Предуслов: Систем је укључен и админ је пријављен на систем под својом шифром. Систем приказује форму за рад са инструктором. Учитана је листа обука и листа чинова.

Основни сценарио СК
1. Админ **уноси** податке о инструктору. (АПУСО)
2. Админ **контролише** да ли је коректно унео податке о инструктору. (АНСО)
3. Админ **позива** систем да запамти податке о инструктору. (АПСО)
4. Систем **памти** податке о инструктору. (СО)
5. Систем **приказује** админу поруку: “Регистрација инструктора успешна!“. (ИА)

Алтернативна сценарија
5.1. Уколико систем не може да запамти податке о инструктору, он приказује админу поруку: “Дошло је до грешке приликом обрађивања захтева!” .(ИА)

СК7: Случај коришћења – Унос новог задатка

Назив СК
Унос новог задатка

Актори СК
Корисник

Учесници СК
Админ или инструкор и систем (програм)

Предуслов: Систем је укључен и корисник је пријављен на систем под својом шифром. Систем приказује форму за рад са задатком. Учитана је листа свих паса.

Основни сценарио СК
1. Корисник **уноси** податке о задатку. (АПУСО)
2. Корисник **контролише** да ли је коректно унео податке о задатку. (АПСО)

3. Корисник **позива** систем да запамти податке о задатку. (АПСО)
4. Систем **памти** податке о задатку. (СО)
5. Систем **приказује** кориснику поруку: “Успешно сачувани подаци о задатку!“. (ИА)

Алтернативна сценарија
5.1. Уколико систем не може да запамти податке о задатку, он приказује кориснику поруку: “Дошло је до грешке приликом уноса задатка!”. (ИА)

СК8: Случај коришћења – Оцењивање ангажовања завршеног задатка

Назив СК
Оцењивање ангажовања завршеног задатка

Актори СК
Корисник (админ или инструктор)

Учесници СК
Корисник и систем (програм)

Предуслов: Систем је укључен и корисник је пријављен на систем под својом шифром. Систем приказује форму за рад са задацима. Учитана је листа задатка.

Основни сценарио СК

1. Админ **уноси** вредност по којој претражује задатке. (АПУСО)
2. Админ **позива** систем да нађе задатке по задатој вредности. (АПСО)
3. Систем **тражи** задатке по задатој вредности. (СО)
4. Систем **приказује** кориснику листу задатка. (ИА)
5. Корисник **позива** систем да прикаже ангажовања на одабраном задатаку. (АПСО)
6. Систем **приказује** податке о ангажовањима на одабраном задатку. (ИА)
7. Корисник **уноси** оцене. (АПУСО)
8. Корисник **контролише** да ли је коректно унео податке о оценама. (АНСО)
9. Корисник **позива** систем да запамти оцене. (АПСО)
10. Систем **памти** оцене.(СО)
11. Систем **приказује** кориснику поруку: “ Ангажовања успешно оцењена!“. (ИА)

Алтернативна сценарија
4.1. Уколико систем не може да нађе задатке, систем приказује админу поруку: “Није пронађен ниједан задатак по унетом критеријуму претраге”. Прекида се извршење сценарија. (ИА)

6.1. Уколико систем не може да прикаже податке о ангажовањима на изабраном задатку, он приказује кориснику поруку: “Није пронађено ниједно ангажовање по унетом критерујуму претаге. “ . Прекида се извршење сценарија. (ИА)

7.1. Уколико систем не може да запамти податке о оцењеним ангажовањима, он приказује кориснику поруку: “Дошло је до грешке приликом оцењивања ангажовања!”. (ИА)

5. Фаза анализе

Логичком структуром софтверског система и *пословном логиком софтверског система* бавимо се у фази анализе. Фаза анализе обухвата приказ *системских дијаграма секвенци* и *системских операција*, чиме се даје опис понашања система, док је структура представљена преко *концептуалног* и *релационог модела* [11].

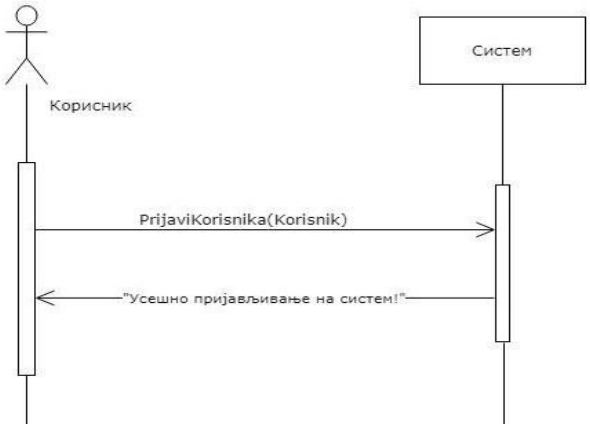
5.1 Понашање софтверског система – Системски дијаграми секвенци

Да би се на најбољи начин представило понашање софтверског система, за сваки случај коришћења уочен у *фази прикупљања захтева* приказује се одговарајући системски дијаграм секвенци, путем кога се моделује интеракција између актера и система путем активности у одређеном редоследу [11].

ДС1. дијаграми секвенци случаја коришћења - Пријављивање на систем

Основни сценарио

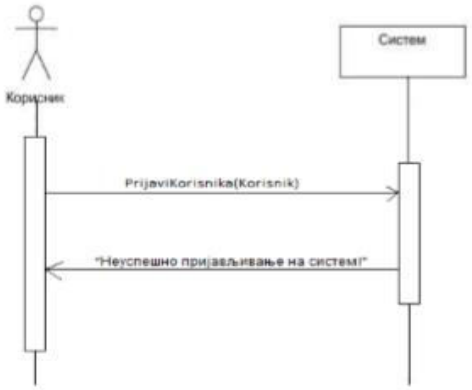
1. Корисник **позива** систем да се улогује (провери податке). (АПСО)
2. Систем **приказује** кориснику поруку: “Успешно пријављивање на систем! “. (ИА)



Слика 8 - ДС Пријављивање на систем

Алтернативна сценарија

2.1. Уколико систем не може да нађе корисника, он приказује кориснику поруку: “Неуспешно пријављивање на систем!”. (ИА)



Слика 9 - *ДС Неуспешно пријављивање на систем*

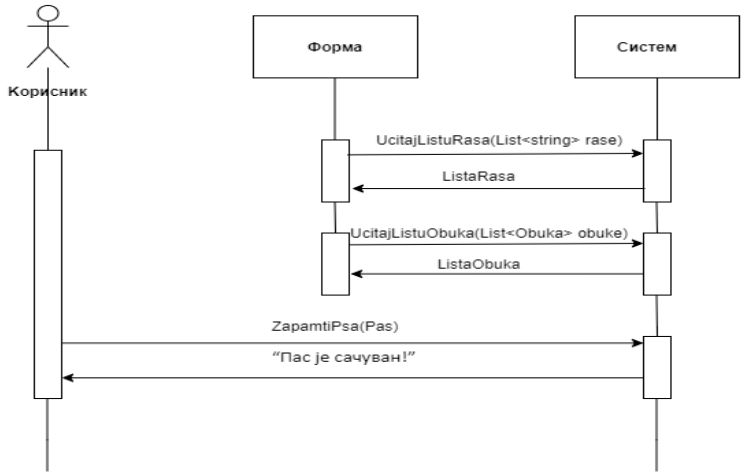
Идентификована системска операција:

- signal **PrijaviKorisnika(Korisnik)**

ДС2. дијаграми секвенци случаја коришћења - Унос новог пса

Основни сценарио

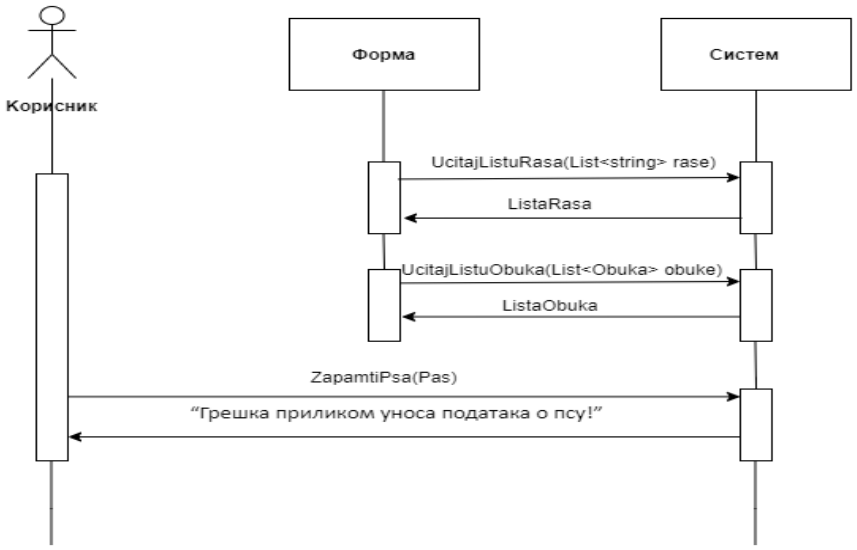
1. Форма **позива** систем да прикаже листу раса.(АПСО)
2. Систем **приказује** на форми листу раса.(ИА)
3. Форма **позива** систем да прикаже листу обука.(АПСО)
4. Систем **приказује** на форми листу обука.(ИА)
5. Корисник **позива** систем да запамти податке о псу. (АПСО)
6. Систем **приказује** кориснику поруку: "Пас је сачуван!" (ИА)



Слика 10 - *ДС Унос новог пса*

Алтернативна сценарија

2.1. Уколико систем не може да запамти податке о псу, он приказује кориснику поруку: "Грешка приликом уноса података о псу!". (ИА)



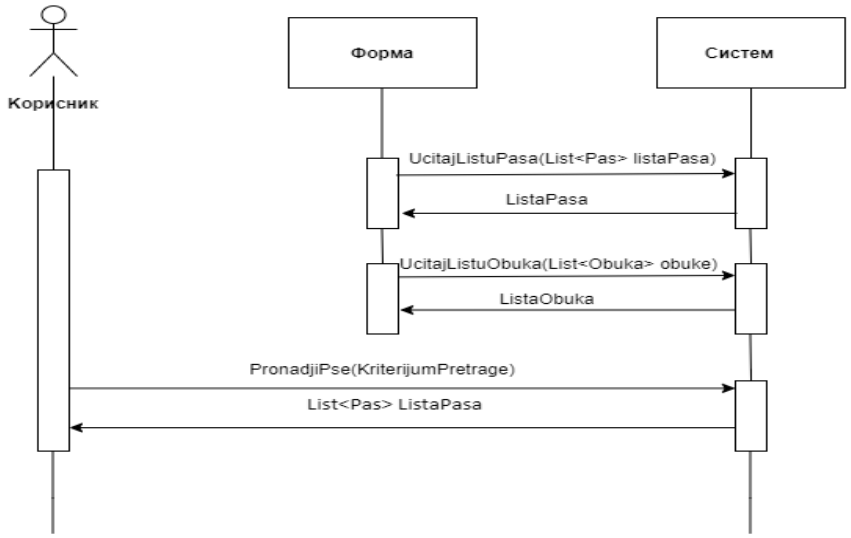
Слика 11 - *ДС Грешка приликом уноса подата о псу*

Идентификоване системске операције:

- signal **UcitajListuRasa(List<string> rase)**
- signal **UcitajListuObuka(List<Obuka> obuke)**
- signal **ZapamtiPsa(Pas)**

ДС3. дијаграми секвенци случаја коришћења - Претраживање паса

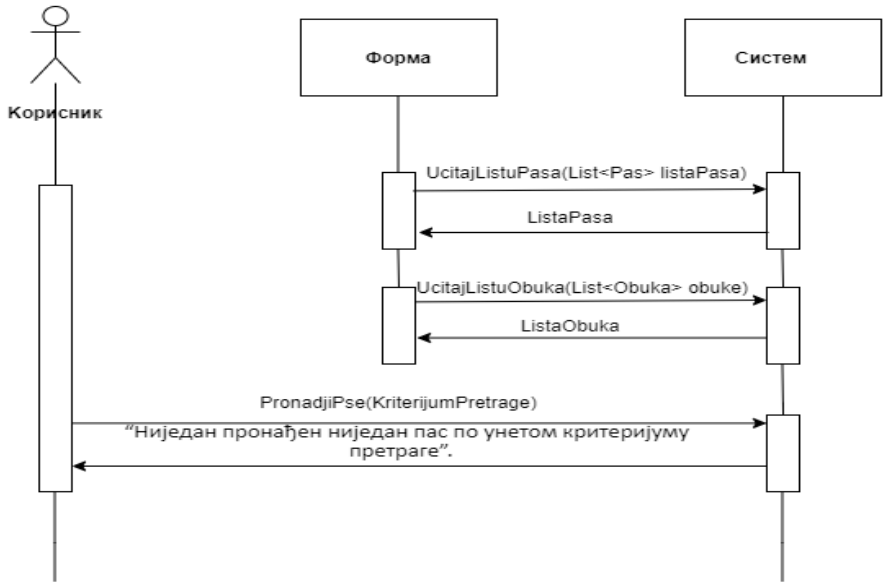
1. Форма **позива** систем да прикаже листу обука.(АПСО)
2. Систем **приказује** на форми листу обука.(ИА)
3. Форма **позива** систем да прикаже листу паса. (АПСО)
4. Систем **приказује** на форми листу паса. (ИА)
5. Корисник **позива** систем да нађе псе по задатој вредности. (АПСО)
6. Систем **приказује** кориснику листу паса. (ИА)



Слика 12 - *ДС Претраживање паса*

Алтернативна сценарија

6.1. Уколико систем не може да нађе псе, систем приказује кориснику поруку: “Ниједан пронађен ниједан пас по унетом критеријуму претраге”.



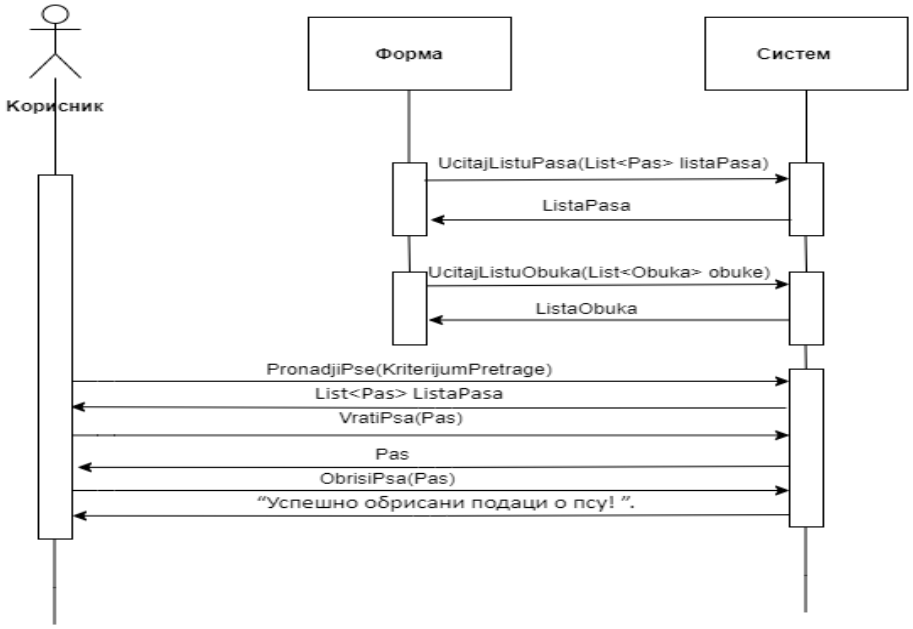
Слика 13 - *ДС Ниједан пронађен ниједан пас по унетом критеријуму претраге*

Идентификоване системске операције:

- signal **UcitajListuPasa(List<Pas> listaPasa)**
- signal **UcitajListuObuka(List<Obuka> obuke)**
- signal **PronadjiPse(KriterijumPretrage)**

ДС4. дијаграми секвенци случаја коришћења - Брисање пса

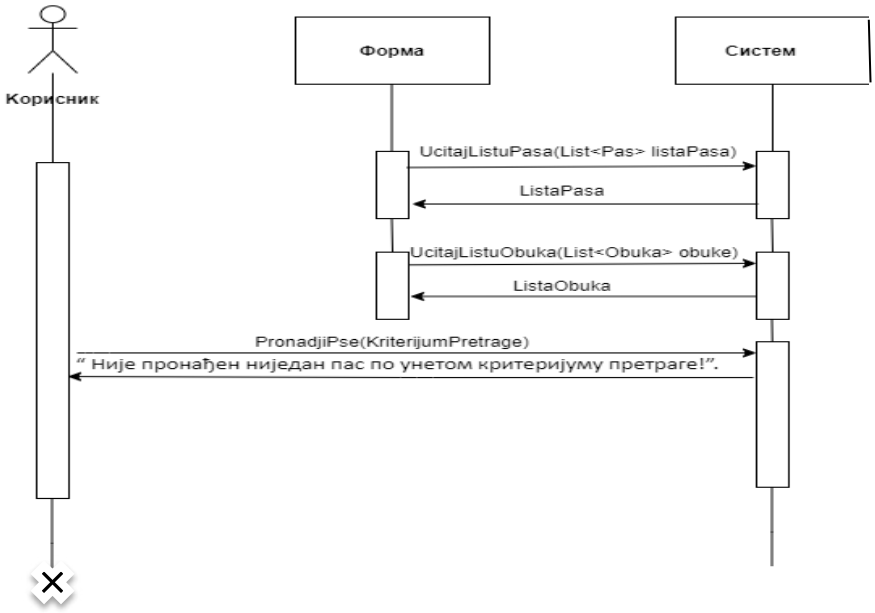
1. Форма **позива** систем да прикаже листу обука.(АПСО)
2. Систем **приказује** на форми листу обука.(ИА)
3. Форма **позива** систем да прикаже листу паса. (АПСО)
4. Систем **приказује** на форми листу паса. (ИА)
5. Корисник **позива** систем да нађе псе по задатој вредности. (АПСО)
6. Систем **приказује** кориснику листу паса. (ИА)
7. Корисник **позива** систем да учита податке о одабраном псу. (АПСО)
8. Систем **приказује** кориснику податке о псу. (ИА)
9. Корисник **позива** систем да обрише пса. (АПСО)
10. Систем **приказује** кориснику поруку: “Успешно обрисани подаци о псу! ”. (ИА)



Слика 14 - *ДС Успешно обрисани подаци о псу*

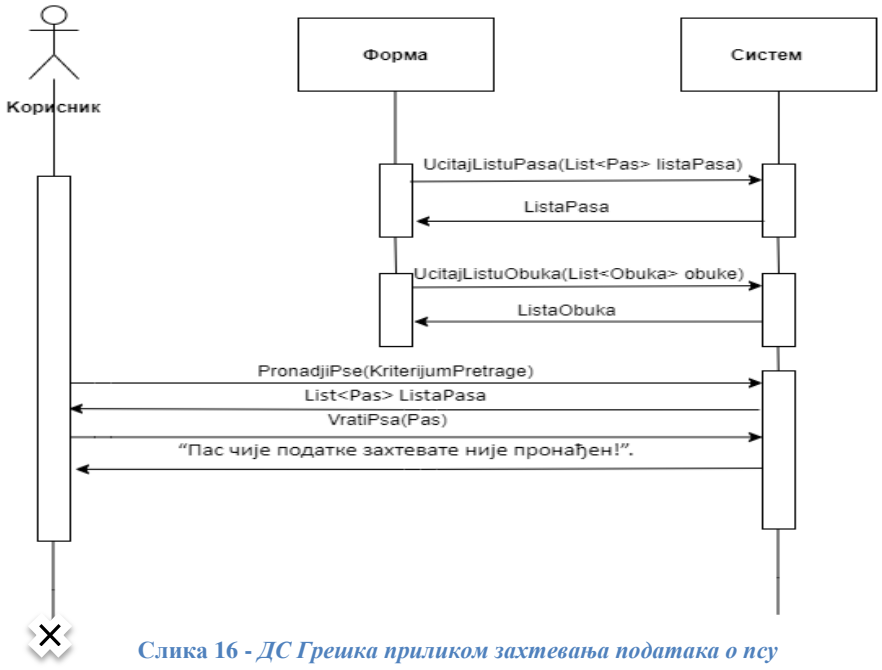
Алтернативна сценарија

4.1. Уколико систем не може да нађе псе, он приказује кориснику поруку: “Није пронађен ниједан пас по унетом критеријуму претраге!”. Прекида се извршење сценарија. (ИА)



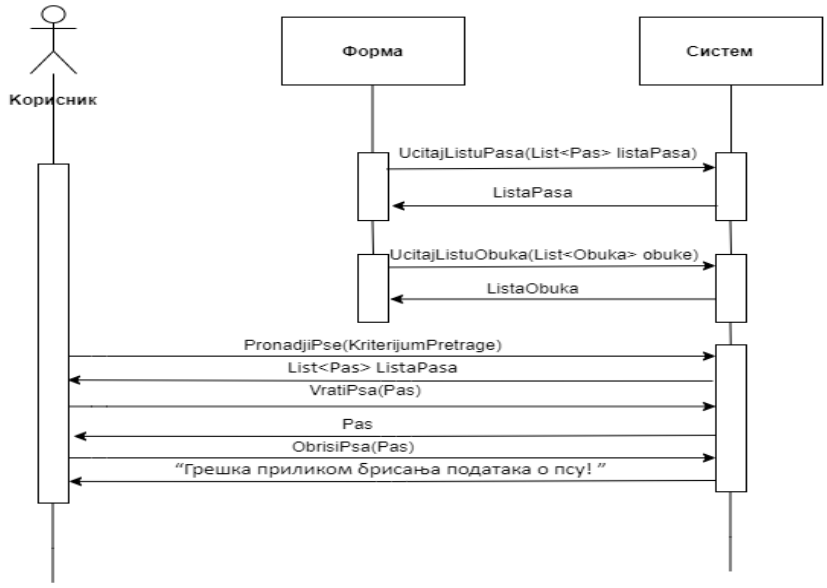
Слика 15 - *ДС Није пронађен ниједан пас по унетом критеријуму претраге*

6.1. Уколико систем не може да учита пса, он приказује кориснику поруку: “Пас чије податке захтевате није пронађен!”. Прекида се извршење сценарија. (ИА)



Слика 16 - *ДС Грешка приликом захтевања података о псу*

10.1. Уколико систем не може да обрише пса, он приказује кориснику поруку: “Грешка приликом брисања пса!”. (ИА)



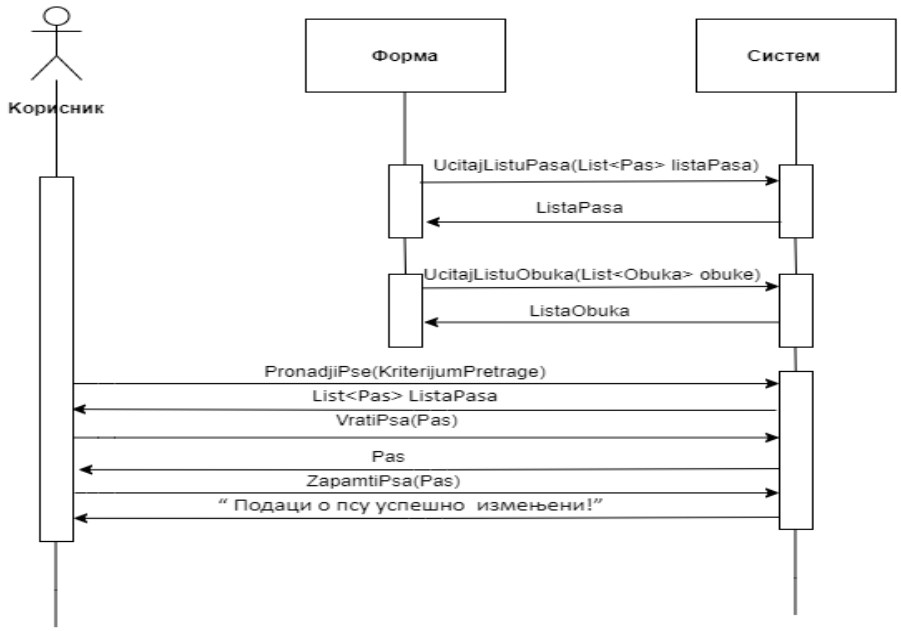
Слика 17 - *ДС Грешка приликом брисања пса*

Идентификоване системске операције:

- signal **UcitajListuPasa(List<Pas> listaPasa)**
- signal **UcitajListuObuka(List<Obuka> obuke)**
- signal **PronadjiPse(KriterijumPretrage)**
- signal **VratiPsa(Pas)**
- signal **ObrisiPsa(Pas)**

ДС5. дијаграми секвенци случаја коришћења - Измена података о псу

1. Форма **позива** систем да прикаже листу обука.(АПСО)
2. Систем **приказује** на форми листу обука.(ИА)
3. Форма **позива** систем да прикаже листу паса. (АПСО)
4. Систем **приказује** на форми листу паса. (ИА)
5. Корисник **позива** систем да нађе псе по задатој вредности. (АПСО)
6. Систем **приказује** кориснику листу паса. (ИА)
7. Корисник **позива** систем да учита податке о одабраном псу. (АПСО)
8. Систем **приказује** кориснику податке о псу. (ИА)
9. Корисник **позива** систем да запамти податке о псу. (АПСО)
10. Систем **приказује** кориснику поруку: “ Подаци о псу успешно измењени!” (ИА)



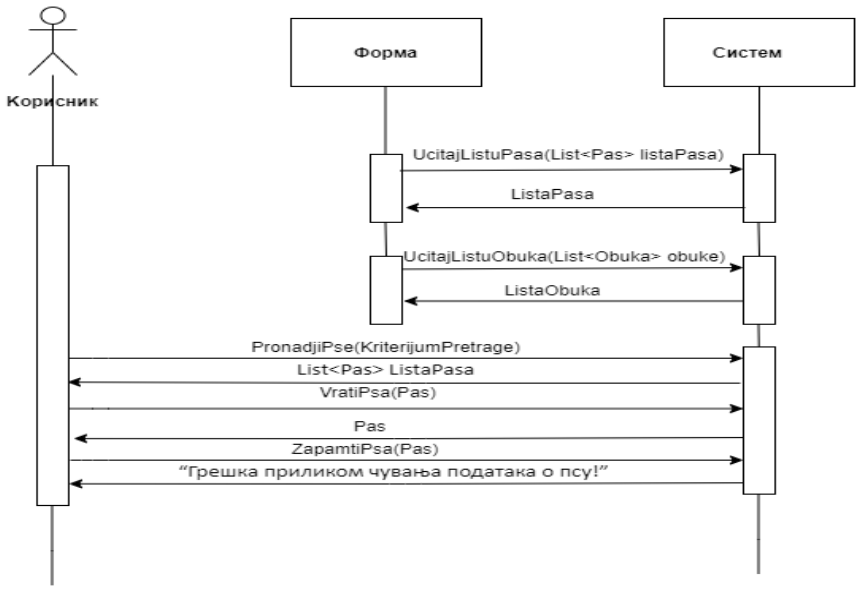
Слика 18 - *ДС Подаци о псу успешно измењени*

Алтернативна сценарија

4.1. Уколико систем не може да нађе псе, он приказује кориснику поруку: " Није пронађен ниједан пас по унетом критеријуму претраге!". Прекида се извршење сценарија. (ИА)

6.1. Уколико систем не може да учита пса, он приказује кориснику поруку: "Пас чије податке захтевате није пронађен!". Прекида се извршење сценарија. (ИА)

10.1. Уколико систем не може да сачува податке о псу, он приказује кориснику поруку: "Грешка приликом чувања података о псу!". (ИА)



Слика 19 - ДС Грешка приликом чувања података о псу

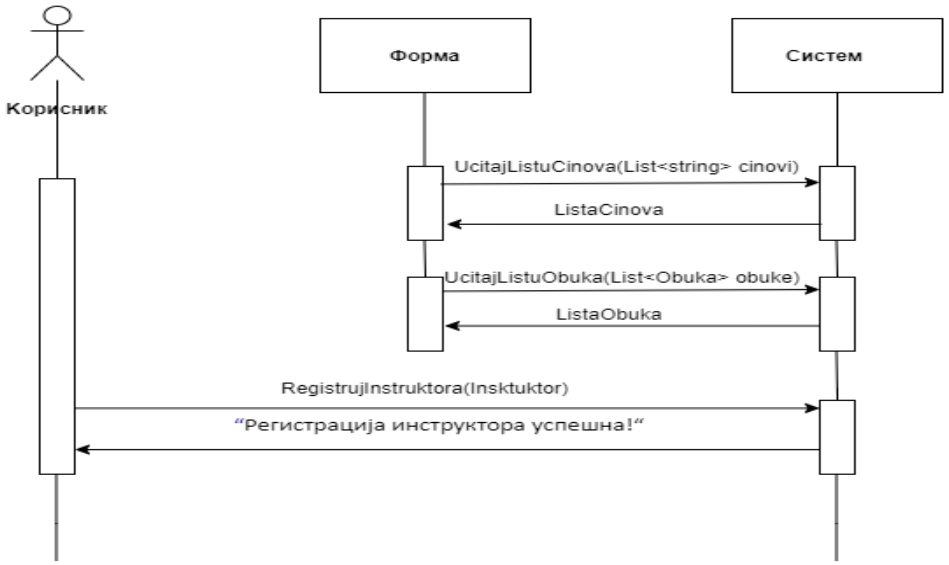
Идентификоване системске операције:

- signal **UcitajListuPasa(List<Pas> listaPasa)**
- signal **UcitajListuObuka(List<Obuka> obuke)**
- signal **PronadjiPse(KriterijumPretrage)**
- signal **VratiPsa(Pas)**
- signal **IzmeniPsa(Pas)**

ДС6. дијаграми секвенци случаја коришћења - Унос новог инструктора

Основни сценарио

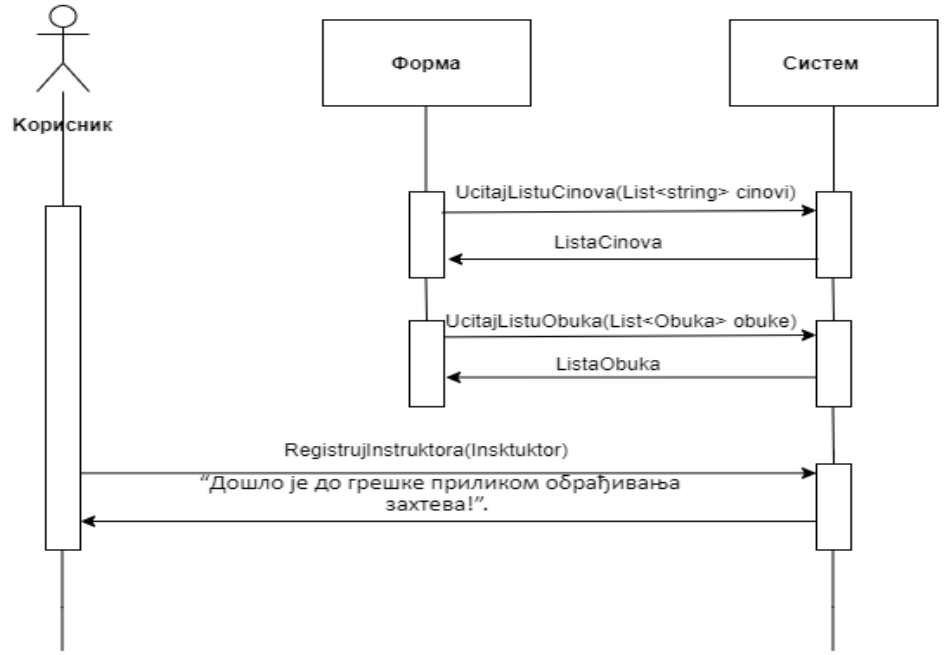
1. Форма **позива** систем да прикаже листу обука.(АПСО)
2. Систем **приказује** на форми листу обука.(ИА)
3. Форма **позива** систем да прикаже листу чинова. (АПСО)
4. Систем **приказује** на форми листу чинова. (ИА)
5. Админ **позива** систем да запамти податке о инструктору. (АПСО)
6. Систем **приказује** админу поруку: "Регистрација инструктора успешна!" (ИА)



Слика 20 - *ДС Унос новог инструктора*

Алтернативна сценарија

6.1. Уколико систем не може да запамти податке о инструктору, он приказује админу поруку: “Дошло је до грешке приликом обрађивања захтева!” (ИА)



Слика 21 - *ДС Дошло је до грешке приликом обрађивања захтева*

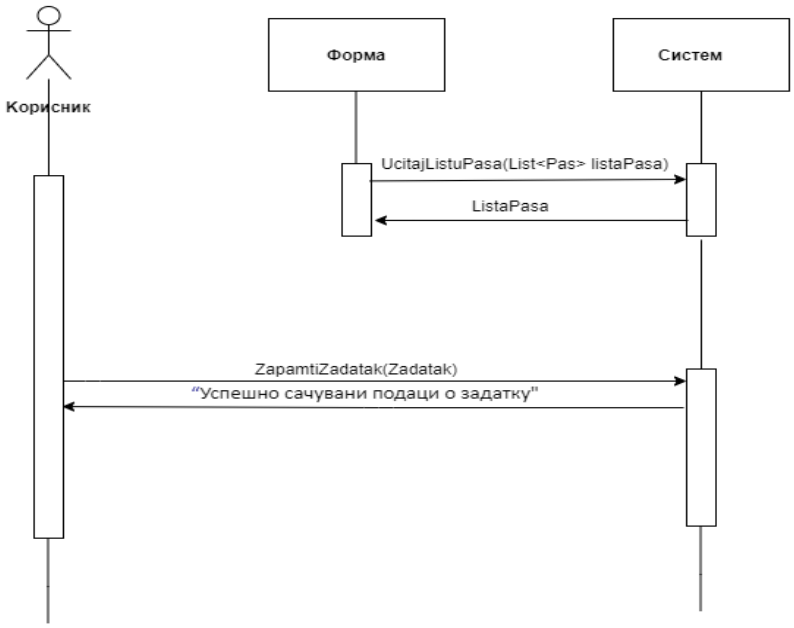
Идентификоване системске операције:

- signal **UcitajListuCinova(List<string> cinovi)**
- signal **UcitajListuObuka(List<Obuka> obuke)**
- signal **RegistrujInstruktora(Instruktor)**

ДС7. дијаграми секвенци случаја коришћења - Унос новог задатка

Основни сценарио

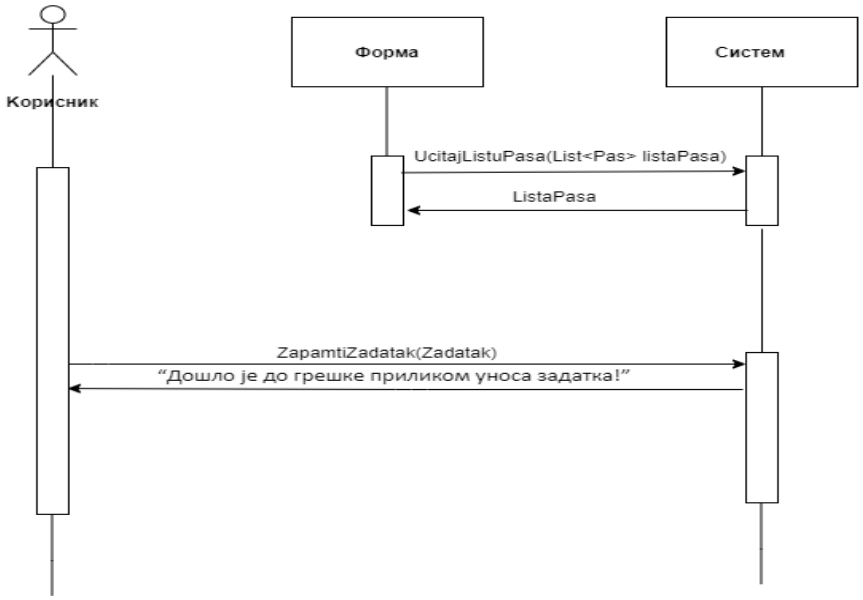
1. Форма **позива** систем да прикаже листу раса.(АПСО)
2. Систем **приказује** на форми листу раса.(ИА)
3. Корисник **позива** систем да запамти податке о задатку. (АПСО)
4. Систем **приказује** кориснику поруку: “Успешно сачувани подаци о задатку!“. (ИА)



Слика 22 - ДС Унос новог задатка

Алтернативна сценарија

4.1. Уколико систем не може да запамти податке о задатку, он приказује кориснику поруку: “Дошло је до грешке приликом уноса задатка!”. (ИА)



Слика 23 - ДС Грешка приликом уноса задатка

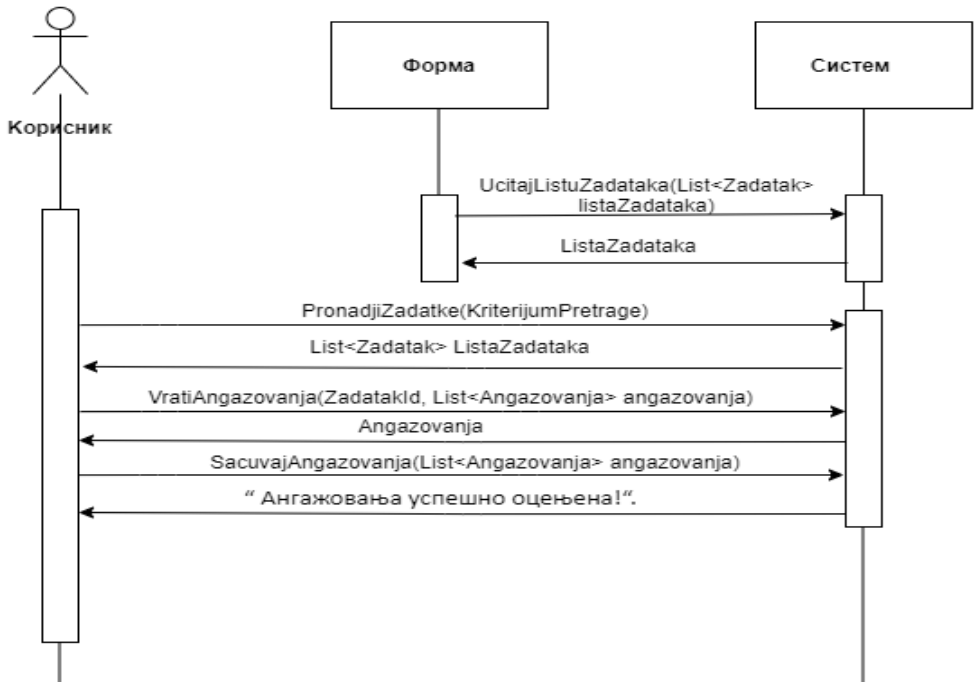
Идентификоване системске операције:

- signal **UcitajListuPasa(List<Pas> listaPasa)**
- signal **ZapamtiZadatak(Zadatak)**

ДС8. дијаграми секвенци случаја коришћења - Оцењивање ангажовања завршеног задатка

Основни сценарио

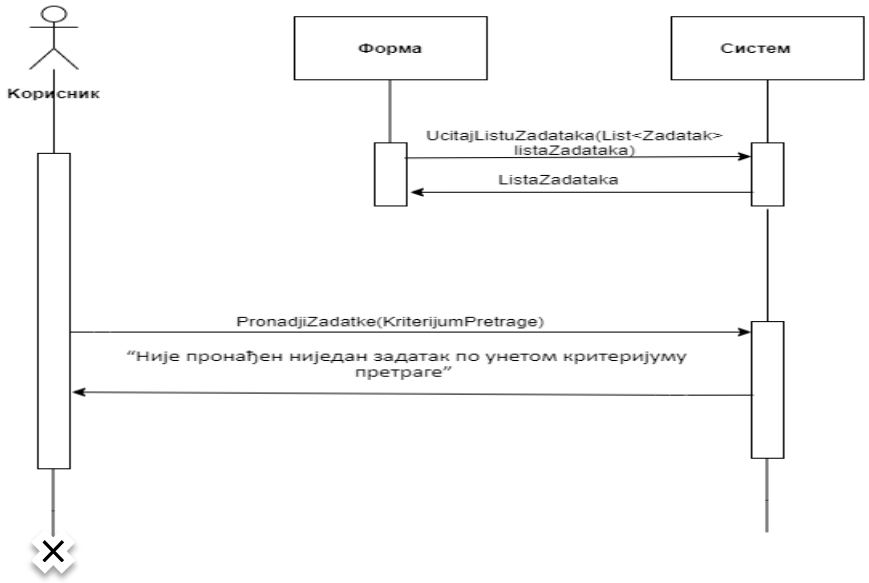
1. Форма **позива** систем да прикаже листу задатака.(АПСО)
2. Систем **приказује** на форми листу задатака.(ИА)
3. Корисник **позива** систем да нађе задатке по задатој вредности. (АПСО)
4. Систем **приказује** кориснику листу задатака. (ИА)
5. Корисник **позива** систем да учита ангажовања одабраног задатка. (АПСО)
6. Систем **приказује** кориснику ангажовања на задатку.(ИА)
7. Корисник **позива** систем да запамти оценења ангажовања. (АПСО)
8. Систем **приказује** кориснику поруку: “ Ангажовања успешно оцењена!“. (ИА)



Слика 24 - ДС Оцењивање ангажовања завршеног задатка

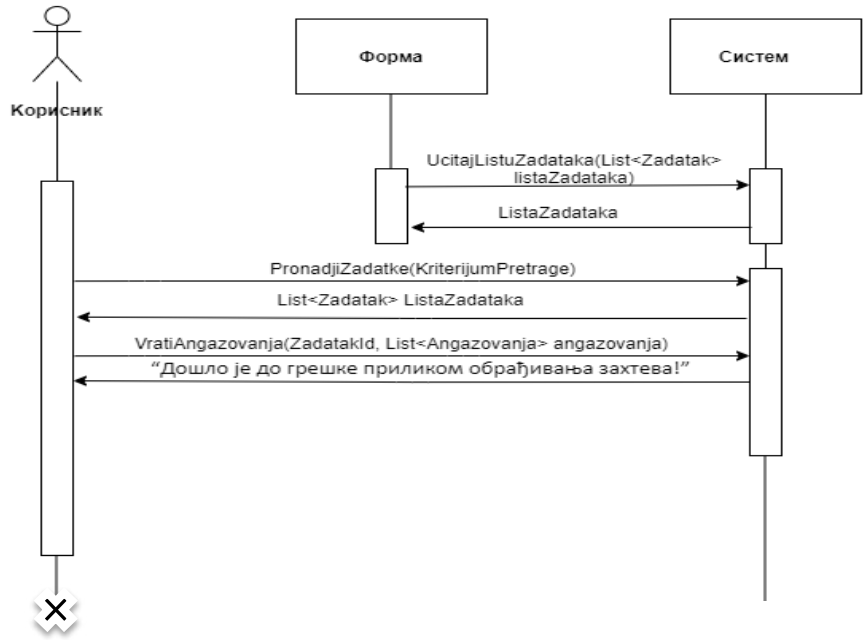
Алтернативна сценарија

4.1. Уколико систем не може да нађе задатке, систем приказује кориснику поруку: “Није пронађен ниједан задатак по унетом критеријуму претраге”. Прекида се извршење сценарија. (ИА)



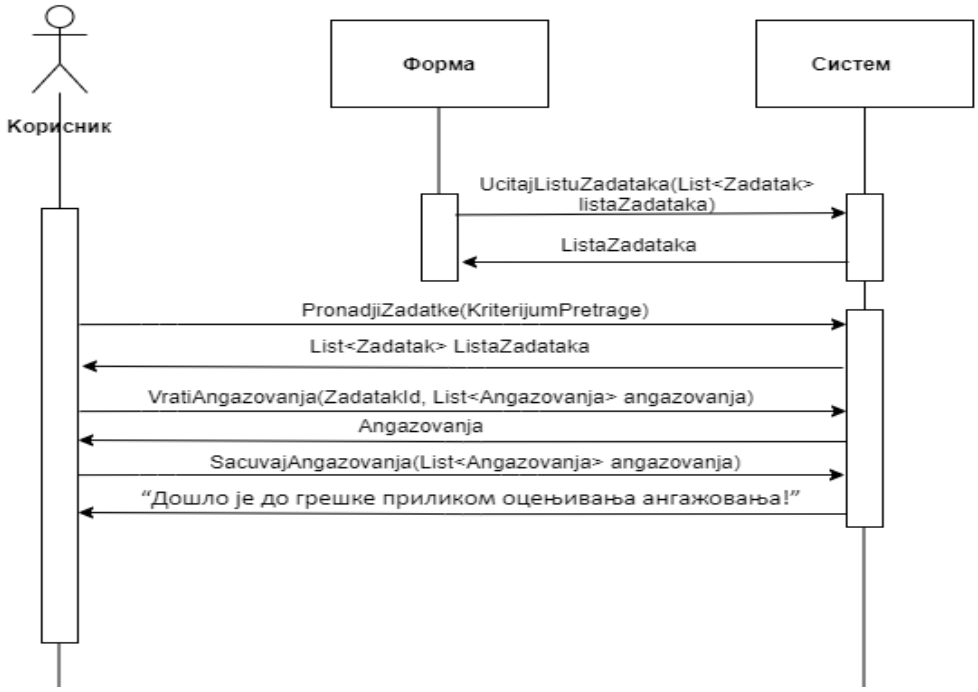
Слика 25 - ДС Није пронађен ниједан задатак по унетом критеријуму претраге

6.2. Уколико систем не може да учита ангажовања, систем приказује кориснику поруку: “Дошло је до грешке приликом обрађивања захтева!”. Прекида се извршење сценарија. (ИА)



Слика 26 - *ДС Дошло је до грешке приликом обрађивања захтева*

8.1. Уколико систем не може да запамти податке о оцењеним ангажовањима, он приказује кориснику поруку: "Дошло је до грешке приликом оцењивања ангажовања!". (ИА)



Слика 27 - *ДС Дошло је до грешке приликом оцењивања ангажовања*

Идентификоване системске операције:

- signal **UcitajListuZadataka(List<Zadatak> listaZadataka)**

- signal **PronadjiZadatake(KriterijumPretrage)**
- signal **VratiAngazovanja(ZadatakId, List<Angazovanja> angazovanja)**
- signal **SacuvajAngazovanja(List<Angazovanja> angazovanja)**

Резултат анализе системских дијаграма секвенци:

Као резултат анализе диграма секвенци, уочава се 16 системских операција које треба пројектовати:

- signal **PrijaviKorisnika(Korisnik)**
- signal **UcitajListuRasa(List<string> rase)**
- signal **UcitajListuObuka(List<Obuka> obuke)**
- signal **ZapamtiPsa(Pas)**
- signal **UcitajListuPasa(List<Pas> listaPasa)**
- signal **PronadjiPse(KriterijumPretrage)**
- signal **VratiPsa(Pas)**
- signal **ObrisiPsa(Pas)**
- signal **UcitajListuCinova(List<string> cinovi)**
- signal **RegistrujInstruktora(Instruktor)**
- signal **ZapamtiZadatak(Zadatak)**
- signal **UcitajListuZadataka(List<Zadatak> listaZadataka)**
- signal **PronadjiZadatake(KriterijumPretrage)**
- signal **VratiAngazovanja(ZadatakId, List<Angazovanja> angazovanja)**
- signal **SacuvajAngazovanja(List<Angazovanja> angazovanja)**
- signal **IzmeniPsa(Pas)**

5.2 Понашање софтверског система - Дефинисање уговора о системским операцијама

Понашање софтверског система дефинисано је системским операцијама за које се праве *уговори. Уговори* описују понашање саме операције, оно што операција треба да одради, али не описују начин на који ће се то одрадити. Један уговор се везује за једну системску операцију [11].

Уговор UG1: PrijaviKorisnika
Операција: PrijaviSe(Korisnik): signal;
Веза са СК: **СК1**
Предуслови: Вредносна и структурна ограничења над објектом ApplicationUser морају бити задовољена.
Постуслови: Korisnik је пријављен на систем.

Уговор UG2: UcitajListuRasa
Операција: UcitajListuRasa(List<string> rase): signal;
Веза са СК: **СК2**
Предуслови: /
Постуслови: /

Уговор UG3: UcitajListuObuka
Операција: UcitajListuObuka(List<Obuka> obuke): signal;
Веза са СК: **СК2, СК3, СК4, СК5, СК6**
Предуслови: /

Постуслови: /

Уговор UG4: ZapamtiPsa
Операција: ZapamtiPsa(Pas): signal;
Веза са СК: **СК2**
Предуслови: Вредносна и структурна ограничења над објектом Пас морају бити задовољена.
Постуслови: Подаци о псу су запамћени.

Уговор UG5: UcitajListuPasa
Операција: UcitajListuPasa(List<Pas> listaPasa): signal;
Веза са СК: **СК3, СК4, СК5, СК7**
Предуслови: /
Постуслови: /

Уговор UG6: PronadjiPse
Операција: PronadjiPse(KriterijumPretrage): signal;
Веза са СК: **СК3, СК4, СК5**
Предуслови: /
Постуслови: /

Уговор UG7: VratiPsa
Операција: VratiPsa(Pas): signal;
Веза са СК: **СК4, СК5**
Предуслови: /
Постуслови: /

Уговор UG8: ObrisiPsa
Операција: ObrisiPsa(Pas): signal;
Веза са СК: **СК4**
Предуслови: Структурна ограничења над објектом Пас морају бити задовољена.
Постуслови: Подаци о псу су обрисани.

Уговор UG9: UcitajListuCinova
Операција: UcitajListuCinova(List<string> cinovi) : signal;
Веза са СК: **СК6**
Предуслови: /
Постуслови: /

Уговор UG10: RegistrujInstruktora
Операција: RegistrujInstruktora (Instruktor): signal;
Веза са СК: **СК6**
Предуслови: Вредносна и структурна ограничења над објектом Инструктор морају бити задовољена.
Постуслови: Инструктор је регистрован.

Уговор UG11: ZapamtiZadatak
Операција: UnosNovogZadatka(Zadatak): signal;
Веза са СК: **СК7**
Предуслови: Вредносна и структурна ограничења над објектом Задатак морају бити задовољена.
Постуслови: Подаци о задатку су запамћени.

Уговор UG12: UcitajListuZadataka

Операција: UcitajListuZadataka(List<Zadatak> listaZadataka): signal;
Веза са СК: **СК8**
Предуслови: /
Постуслови: /

Уговор UG13: PronadjiZadatake
Операција: PronadjiZadatake(KriterijumPretrage): signal;
Веза са СК: **СК8**
Предуслови: /
Постуслови: /

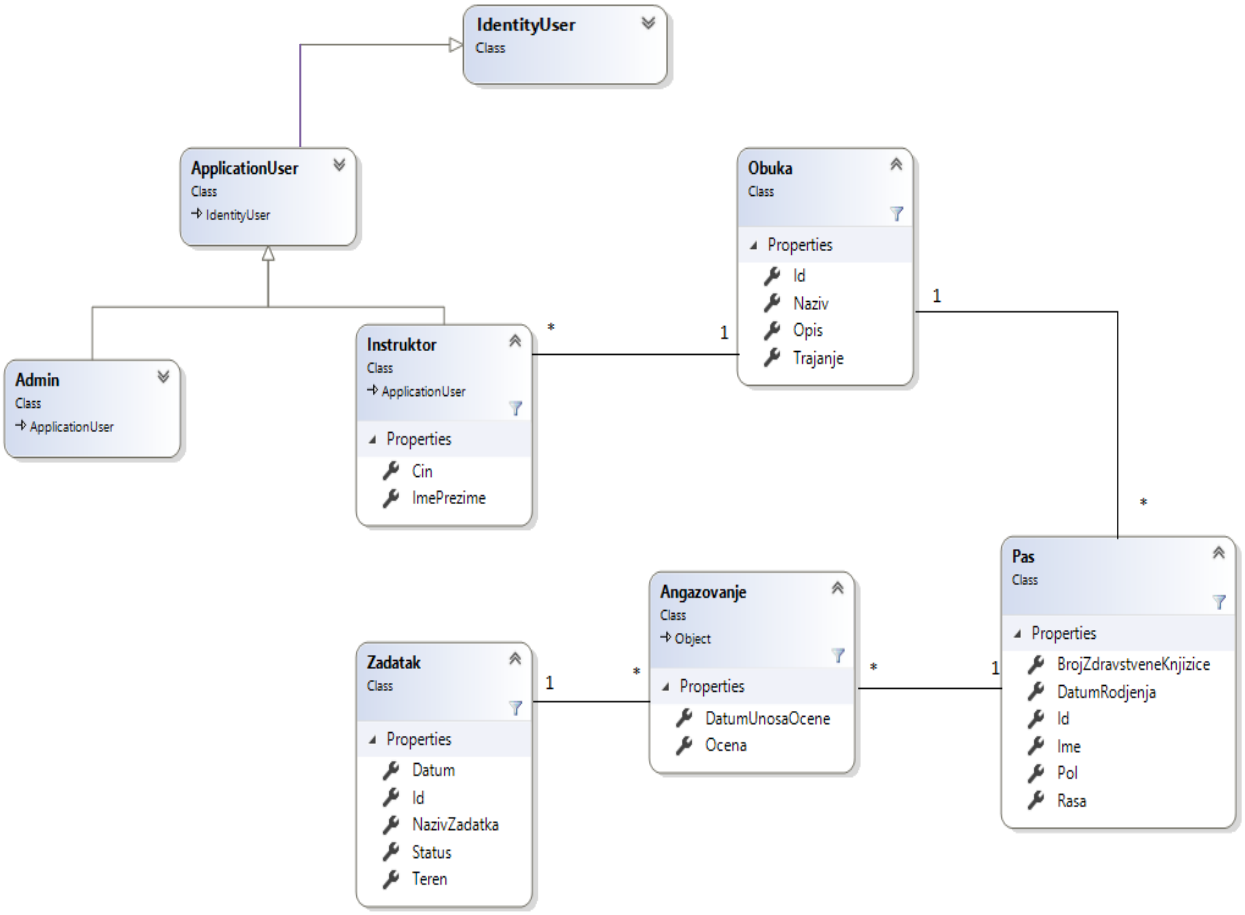
Уговор UG14: VratiAngazovanja
Операција: VratiAngazovanja(ZadatakId): signal;
Веза са СК: **СК8**
Предуслови: /
Постуслови: /

Уговор UG15: SacuvajAngazovanja
Операција: SacuvajAngazovanja(List<Angazovanja> angazovanja): signal;
Веза са СК: **СК8**
Предуслови: Вредносна и структурна ограничења над објектом Ангажовање морају бити задовољена.
Постуслови: Ангажовања су замапћена.

Уговор UG16: IzmeniPsa
Операција: IzmeniPsa(Pas): signal;
Веза са СК: **СК5**
Предуслови: Вредносна и структурна ограничења над објектом Пас морају бити задовољена.
Постуслови: Измењени подаци о псу су запамћени.

5.3 Структура софтверског система – Концептуални (доменски) модел

Структура софтверског система се описује помоћу *концептуалног модела*. На концептуалном моделу приказане су концептуалне класе доменског модела као и њихове међусобне везе, односно асоцијације између њих. Називају *доменским моделима* или *моделима објектне анализе* [11].

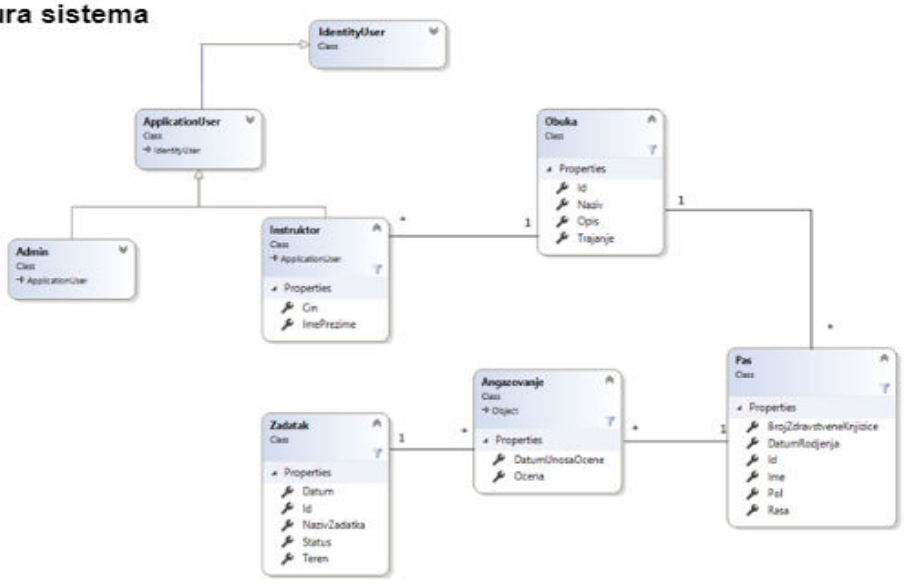


Слика 28 - ***Дијаграм класа***

Као резултат анализе случајева коришћења и прављења концептуалног модела добија се *логичка структура и понашање система [11].*

Softverski sistem

Struktura sistema



Ponašanje sistema

SISTEMSKE OPERACIJE (SO)
PrijaviKorisnika(Korisnik):signal
UcitajListuRasa(List<string> rase):signal
UcitajListuObuka(List<Obuka>obuke):signal
ZapamtiPsa(Pas):signal
UcitajListuPasa(List<Pas> listaPasa):signal
PronadjiPse(KriterijumPretrage):signal
VratiPsa(Pas):signal
ObrisiPsa(Pas):signal
UcitajListuCinova(List<string> cinovi):signal
RegistrujInstruktora(Instruktor):signal
ZapamtiZadatak(Zadatak):signal
UcitajListuZadataka(List<Zadatak>listaZadataka):signal
PronadjiZadatake(KriterijumPretrage):signal
VratiAngazovanja(ZadatakId,List<Angazovanja>angazovanja):signal
SacuvajAngazovanja(List<Angazovanja>angazovanja):signal
IzmeniPsa(Pas):signal

Слика 29 - *Софтверски систем*

5.4 Структура софтверског система - Релациони модел

На основу концептуалног модела добијен је *релациони модел,* којиће да представља основу за пројектовање релационе базе података [11].

Класе *Instruktor* и *Admin* представљају наслеђене класе чија је базна класа *ApplicationUser*. Класа *ApplicationUser* наслеђује класу *IdentityUser* коју чине неколико важних атрибута за корисника апликације као што су јединствени идентификатор, електронска пошта, лозинка заштићена криптографском *hash* функицојом и слично.

- Pas(Id, Ime, Pol, DatumRodjenja, BrojZdravstveneKnjizice, Rasa, *ObukaId*)
- Instruktor(*IdentityUserId* ,Cin,ImePrezime, *ObukaId*)
- Obuka(Id, Naziv, Opis, Trajanje)
- Zadatak(Id,Datum,NazivZadatka,Status,Teren)
- Angazovanje(*PasId,ZadatakId*,DatumUnosaOcene,Ocena)
- Admin(*IdentityUserId*)
- ApplicationUser(*IdentityUserId*)
- IdentityUser(Id, UserName, PasswordHash, Email, EmailConfirmed)

Табела *Pas*		**Просто вредносно ограничење**		**Сложено вредносно ограничење**		**Структурно ограничење**
Атрибути	**Име**	**Тип атрибута**	**Вредност атрибута**	**Међузависност атрибута једне табеле**	**Међузависност атрибута више табела**	INSERT RESTRICTED *Obuka* UPDATE RESTRICTED *Obuka* CASCADES *Angazovanje* DELETE RESTRICTED *Angazovanje*
	Id	Integer	Not null and > 0			
	Ime	String	Not null			
	ObukaId	Integer	Not null and >0			
	Rasa	String	Not null			
	Pol	String	Not null			
	DatumRodjenja	DateTime	Not null			
	BrojZdravstveneKnjizice	String	Not null and > 0			

Табела 1 - Табела Pas

Табела *Obuka*		**Просто вредносно ограничење**		**Сложено вредносно ограничење**		**Структурно ограничење**
Атрибути	**Име**	**Тип атрибута**	**Вредност атрибута**	**Међузависност атрибута једне табеле**	**Међузависност атрибута више табела**	INSERT / UPDATE CASCADES *Pas, IdentityUser* DELETE RESTRICTED *Pas, IdentityUser*
	Id	Integer	Not null and > 0			
	NazivObuke	String	Not null			
	Trajanje	Double	Not null and > 0			
	Opis	String	Not null			

Табела 2 - Табела Obuka

Табела *Zadatak*		**Просто вредносно ограничење**		**Сложено вредносно ограничење**		**Структурно ограничење**
Атрибути	**Име**	**Тип атрибута**	**Вредност атрибута**	**Међузависност атрибута једне табеле**	**Међузависност атрибута више табела**	INSERT / UPDATE CASCADES *Angazovanje* DELETE RESTRICTED *Angazovanje*
	Id	Integer	Not null and > 0			
	NazivZadatka	String	Not null			
	Datum	Date	Not null			
	Status	String	Not null			
	Teren	String	Not null			

Табела 3 - Табела Zadatak

Табела *Angazovanje*		**Просто вредносно ограничење**		**Сложено вредносно ограничење**		**Структурно ограничење**
Атрибути	**Име**	**Тип атрибута**	**Вредност атрибута**	**Међузависност атрибута једне табеле**	**Међузависност атрибута више табела**	INSERT RESTRICTED *Zadatak,Pas* UPDATE RESTRICTED *Zadatak,Pas* DELETE /
	ZadatakId	Integer	Not null and > 0			
	PasId	Integer	Not null and > 0			
	DatumUnosaOcene	Date				
	Ocena	Integer				

Табела 4 – Табела Angazovanje

Табела *AspNetUsers*		**Просто вредносно ограничење**		**Сложено вредносно ограничење**		**Структурно ограничење**
Атрибути	**Име**	**Тип атрибута**	**Вредност атрибута**	**Међузависност атрибута једне табеле**	**Међузависност атрибута више табела**	INSERT RESTRICTED *Obuka* UPDATE RESTRICTED *Obuka* DELETE /
	Id	Integer	Not null and > 0			
	UserName	Integer	Not null and > 0			
	Email	String				
	EmailConfirmed	Boolean	Not null			
	PasswordHash	String				
	ImePrezime	String				
	Cin	String				
	Obuka	Integer				

Табела 5 – Табела AspNetUsers

6. Фаза пројектовања

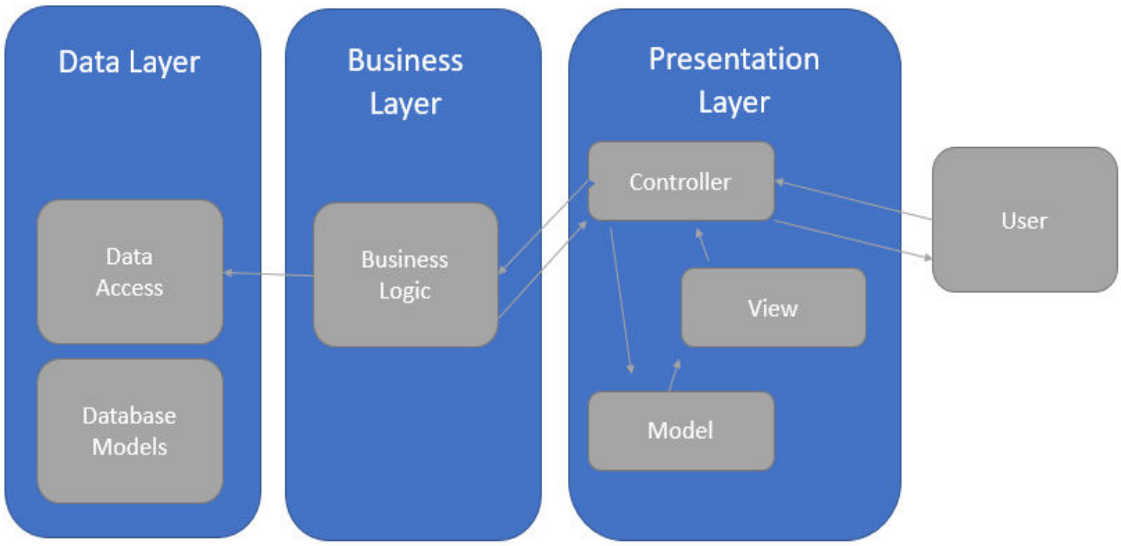
Фаза пројектовања описује *физичку структуру* и *понашање софтверског система* (архитектуру софтверског система). Најчешће коришћена архитектура је тронивојска архитектура која се састоји из следећих нивоа [11]:

- корисничког интерфејса односно екранских форми
- апликационе логике
- складишта података

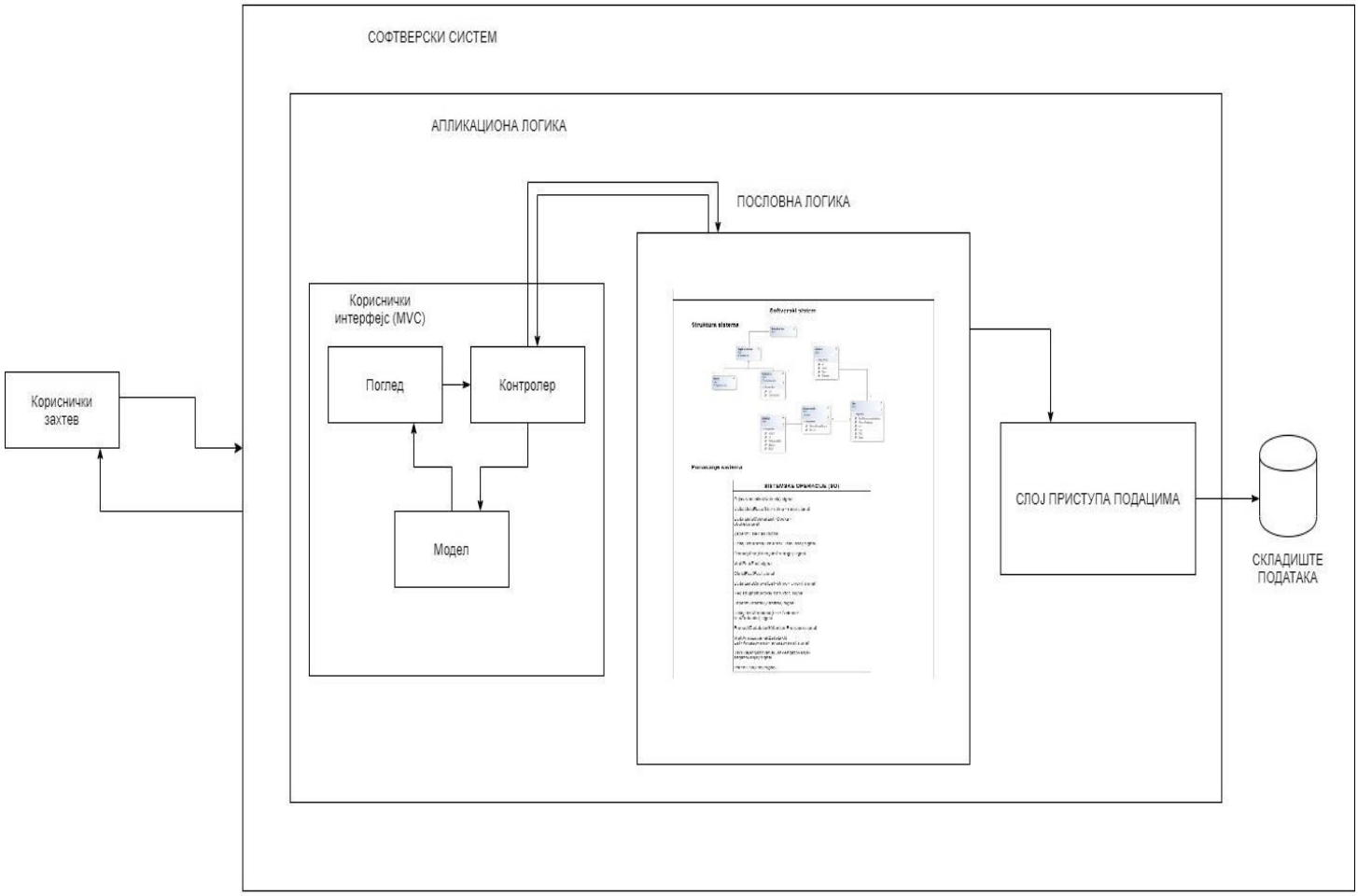
Пројектовање архитектуре софтверског система обухвата пројектовање наведена три слоја. Ниво корисничког интерфејса смештен је на страни клијента, док се апликациона логика и складиште података налазе на страни сервера.

6.1 Архитектура софтверског система

Приликом пројектовања ASP .NET CORE MVC веб апликације, кориснички интерфејс представља део презентационог слоја у оквиру MVC патерна, апликациону логику чини пословна логика и слој приступа подацима остварен преко *Entity Framework Core*-а. Складиште података представља у овом случају *SQL* базу података.



Слика 30 - *Архитектура ASP .NET Core MVC апликације*. (n.d.). [Digital image]. https://qph.fs.quoracdn.net/main-qimg-c1bf4bd17f11d65cdfba817587ca52ea

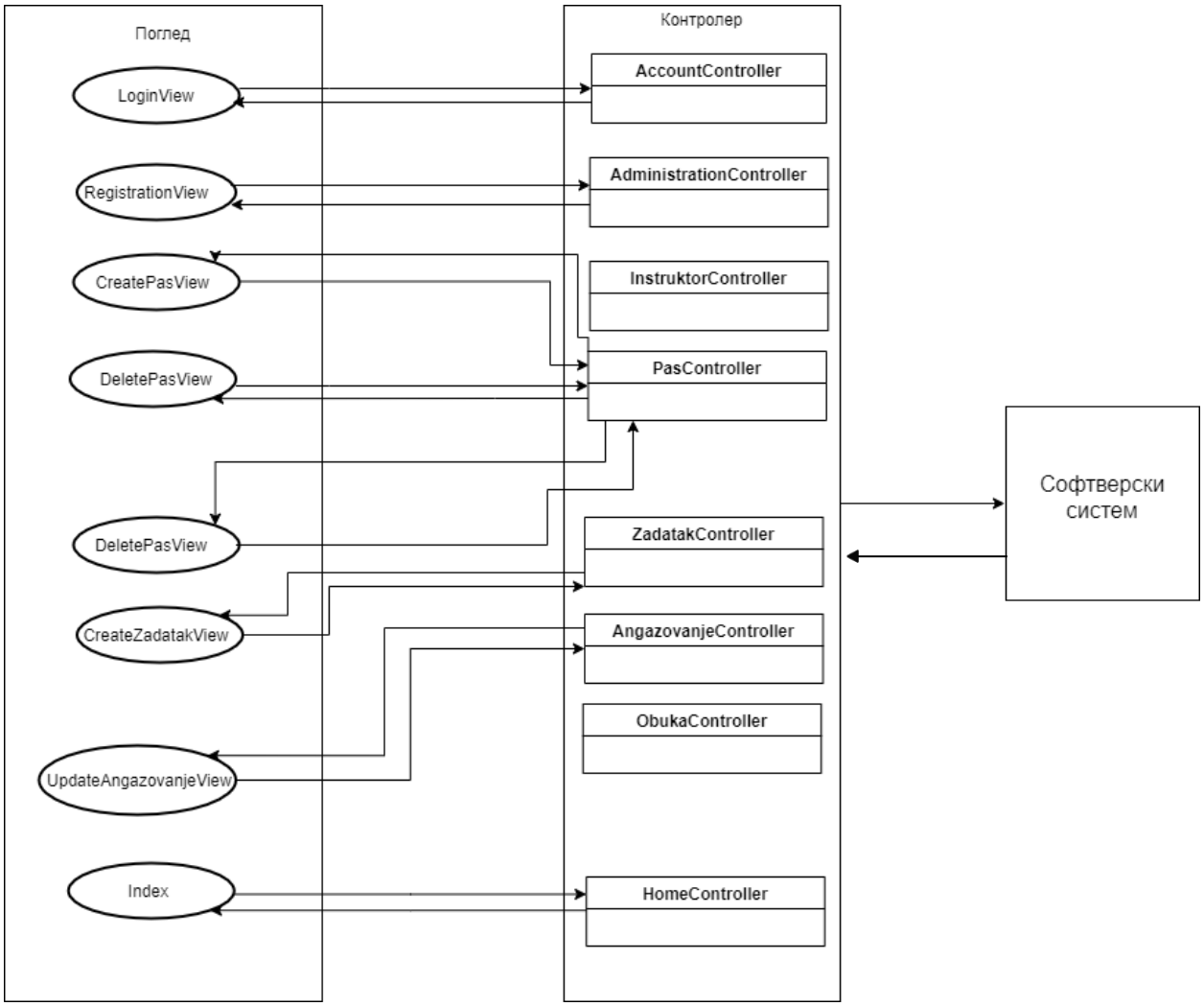


Слика 31 - ***Архитектура софтверског система***

До почетка фазе пројектовања, већ је дефинисана пословна логика софтверског система, односно његова логичка структура и понашање. У наставку ћемо пројектовати сваки од наведених елемената тронивојске архитектуре.

6.2 Пројектовање екранских форми

Кориснички интерфејс представља реализацију улаза и/или излаза софтверског система [11]. У овом раду структуру корисничког интерфејса чине веб странице које прихватају податке које корисник уноси, прослеђују их до одговарајућег контролера, а након што систем обради податке, приказује их кориснику.



Слика 32 – Повезаност контролера са корисничким интерфејсом у архитектури

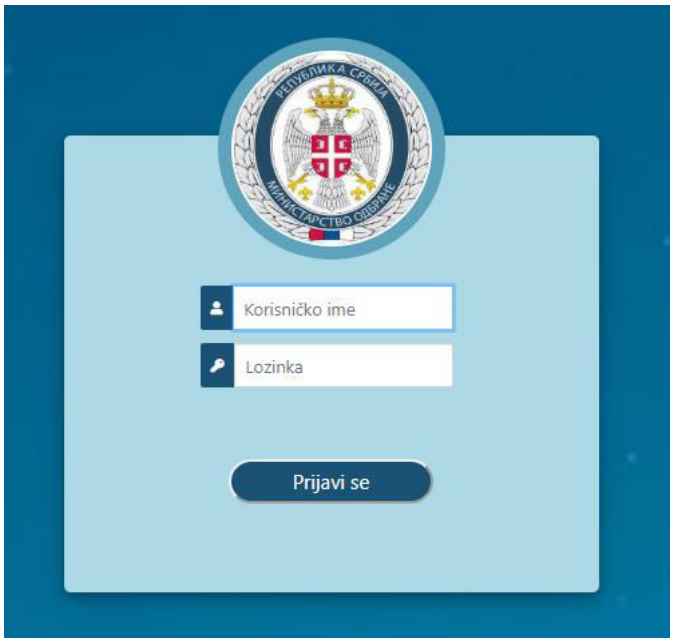
СК1: Случај коришћења – Пријављивање на систем

Назив СК
Пријављивање на систем

Актори СК
Корисник (админ или инструктор)

Учесници СК
Корисник и систем (програм)

Предуслов: Систем је укључен. Систем приказује форму за пријављивање на систем.



Слика 33 - *Форма за пријаву на систем*

Основни сценарио СК

1. Корисник **уноси** корисничко име и лозинку. (АПУСО)
2. Корисник **контролише** да ли је коректно унео корисничко име и лозинку. (АНСО)
3. Корисник **позива** систем да се улогује (провери податке). (АПСО)

Опис акције: Корисник кликом на дугме „Prijavi se“ позива системску операцију PrijaviKorisnika(Korisnik)

4. Систем **проверава** податке о кориснику. (СО)
5. Систем **приказује** кориснику почетну страну и поруку: “Успешно пријављивање на систем! “. (ИА)

Слика 34 - *Успешно пријављивање*

Алтернативна сценарија

5.1. Уколико систем не може да нађе корисника, он приказује кориснику поруку: “ Неуспешно пријављивање на систем!”. (ИА)

Слика 35 - *Неуспешно пријављивање*

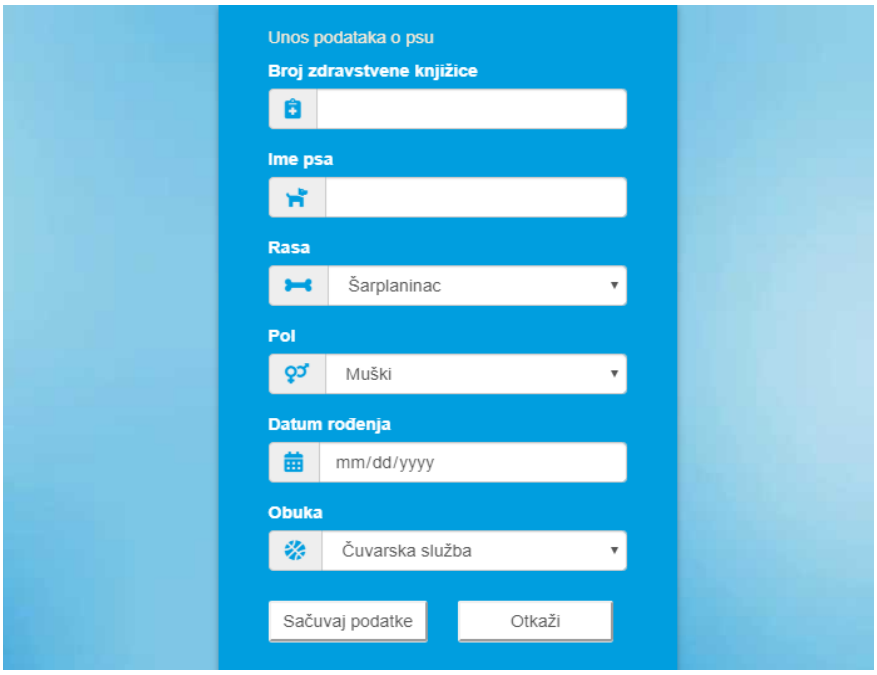
СК2: Случај коришћења – Унос новог пса

Назив СК
Унос новог пса

Актори СК
Корисник (администратор или инструктор)

Учесници СК
Администратор и систем (програм)

Предуслов: Систем је укључен и корисник је пријављен на систем под својом шифром. Систем приказује форму за рад са псом. Учитана је листа свих раса и листа обука.



Слика 36 - *Форма за унос података о псу*

Основни сценарио СК

1. Корисник **уноси** податке о псу. (АПУСО)
2. Корисник **контролише** да ли је коректно унео податке о псу. (АНСО)
3. Корисник **позива** систем да запамти податке о псу. (АПСО)

Опис акције: Кликом на дугме „Sačuvaj podatke“ радник позива системску операцију ZapamtiPsa(Pas)

4. Систем **памти** податке о псу. (СО)
5. Систем **приказује** кориснику поруку: “Пас је сачуван!”. (ИА)

Слика 37 - *Успешно сачувани подаци о псу*

Алтернативна сценарија

5.1. Уколико систем не може да запамти податке о псу, он приказује кориснику поруку: “Грешка приликом уноса података о псу!”. (ИА)

Слика 38 - *Грешка приликом уноса података о псу*

СК3: Случај коришћења – Претраживање паса

Назив СК
Претраживање паса

Актори СК
Корисник (администратор)

Учесници СК
Корисник и систем (програм)

Предуслов: Систем је укључен и корисник је пријављен на систем под својом шифром. Систем приказује форму за рад са псима. Учитана је листа обука.

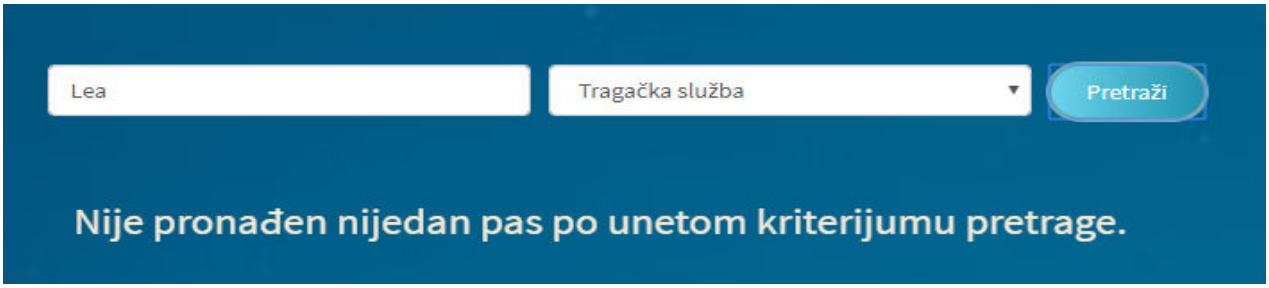
Слика 39 - *Форма за претрагу паса*

Основни сценарио СК
1. Корисник **уноси** вредност по којој претражује псе. (АПУСО)
2. Корисник **позива** систем да нађе псе по задатој вредности. (АПСО)

Опис акције: Корисник кликом на дугме „Pretraži“ позива системску операцију PronadjiPse(KriterijumPretrage)

3. Систем **тражи** псе по задатој вредности. (СО)
4. Систем **приказује** кориснику листу паса. (ИА)

Алтернативна сценарија
4.1. Уколико систем не може да нађе псе, систем приказује кориснику поруку: “Ниједан пронађен ниједан пас по унетом критеријуму претраге”. (ИА)



Слика 40 - *Ниједан пронађен ниједан пас по унетом критеријуму претраге*

СК4: Случај коришћења – Брисање података о псу

Назив СК
Брисање података о псу

Актори СК
Корисник (администратор или инструктор)

Учесници СК
Корисник и систем (програм)

Предуслов: Систем је укључен и корисник је пријављен на систем под својом шифром. Систем приказује форму за рад са псима. Учитана је листа обука.

Основни сценарио СК
1. Корисник **уноси** вредност по којој претражује псе. (АПУСО)
2. Корисник **позива** систем да нађе псе по задатој вредности. (АПСО)

Опис акције: Корисник кликом на дугме „Pretraži“ позива системску операцију PronadjiPse(KriterijumPretrage)

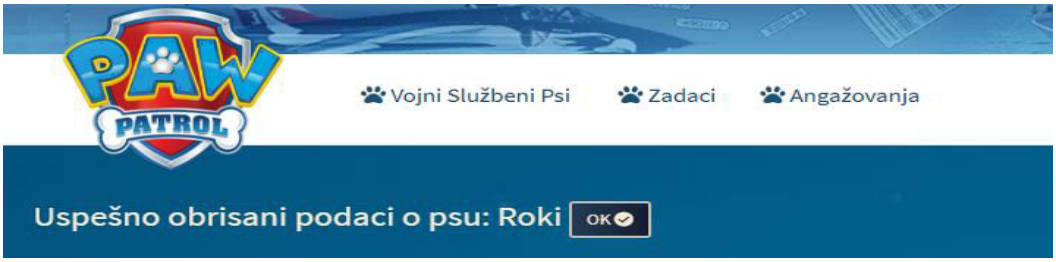
3. Систем **тражи** псе по задатој вредности. (СО)
4. Систем **приказује** кориснику листу паса. (ИА)
5. Корисник **бира** пса којег жели да обрише. (АПУСО)
6. Корисник **позива** систем да учита податке о одабраном псу. (АПСО)
7. Систем **учитава** податке о одабраном псу. (СО)
8. Систем **приказује** кориснику податке о псу. (ИА)

Слика 30 - ***Форма за брисање података о псу***

9. Корисник **позива** систем да обрише пса. (АПСО)

Опис акције: Корисник кликом на дугме „Potvrdi brisanje“ позива системску операцију ObrisiPsa(Pas)

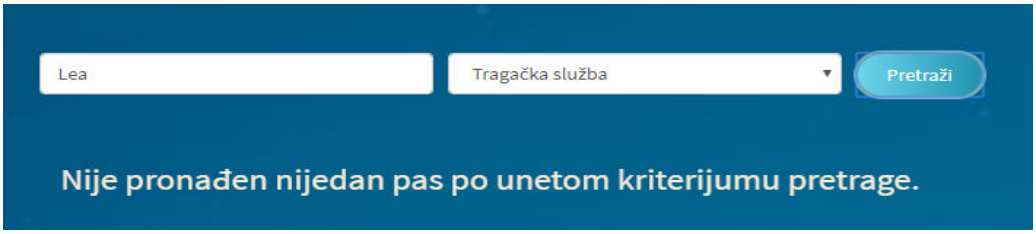
10. Систем **брише** пса. (СО)
11. Систем **приказује** кориснику поруку: “Успешно обрисани подаци о псу! ”. (ИА)



Слика 31 - ***Успешно обрисани подаци о псу***

Алтернативна сценарија

4.1. Уколико систем не може да нађе псе, он приказује кориснику поруку: “ Није пронађен ниједан пас по унетом критеријуму претраге!”. Прекида се извршење сценарија. (ИА)



Слика 32 - ***Није пронађен ниједан пас по унетом критеријуму претраге***

8.1. Уколико систем не може да учита пса, он приказује кориснику поруку: “Пас чије податке захтевате није пронађен!”. Прекида се извршење сценарија. (ИА)

Слика 33 - *Пас чији су подаци захтевани није пронађен*

11.1. Уколико систем не може да обрише пса, он приказује кориснику поруку: “Грешка приликом брисања пса!”. (ИА)

Слика 34 - *Грешка приликом брисања пса*

СК5: Случај коришћења – Измена података о псу

Назив СК
Измена података о псу

Актори СК
Корисник (администратор или инструктор)

Учесници СК
Корисник и систем (програм)

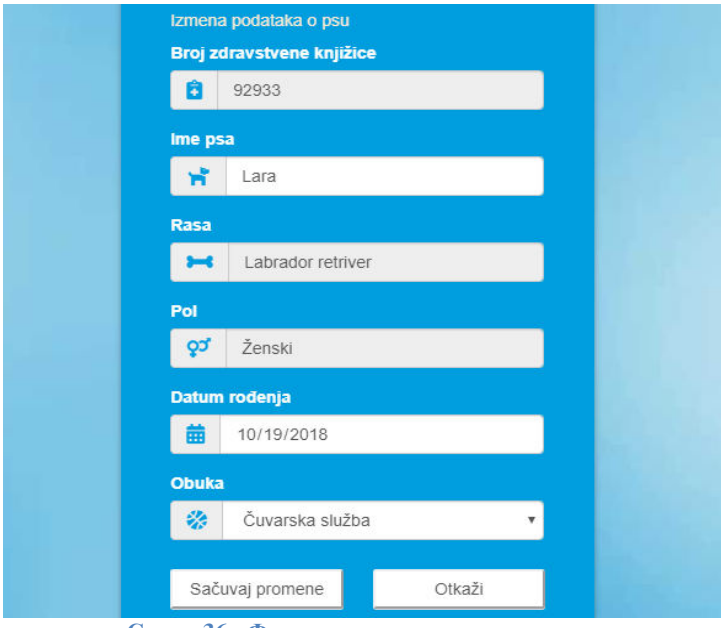
Предуслов: Систем је укључен и корисник је пријављен на систем под својом шифром. Систем приказује форму за рад са псима. Учитана је листа обука.

Основни сценарио СК

1. Корисник **уноси** вредност по којој претражује псе. (АПУСО)
2. Корисник **позива** систем да нађе псе по задатој вредности. (АПСО)
3. Систем **тражи** псе по задатој вредности. (СО)
4. Систем **приказује** кориснику листу паса. (ИА)

Слика 35 - *Листа паса*

5. Корисник **бира** пса чије податке жели да измени. (АПУСО)
6. Корисник **позива** систем да учита податке о одабраном псу. (АПСО)
7. Систем **учитава** податке о одабраном псу. (СО)
8. Систем **приказује** кориснику податке о псу. (ИА)



Слика 36 - Форма за измену података о псу

9. Корисник **уноси (мења)** податке о псу. (АПУСО)
10. Корисник **контролише** да ли је коректно унео податке о псу. (АНСО)

11. Корисник **позива** систем да запамти податке о псу. (АПСО)

Опис акције: Корисник кликом на дугме „Sačuvaj promene“ позива системску операцију IzmeniPsa(Pas)

12. Систем **памти** податке о псу. (СО)
13. Систем **приказује** кориснику поруку: “Подаци о псу успешно измењени!”. (ИА)

Слика 37 - *Подаци о псу успешно измењени*

Алтернативна сценарија

4.1. Уколико систем не може да нађе псе, он приказује кориснику поруку: “ Није пронађен ниједан пас по унетом критеријуму претраге!”. Прекида се извршење сценарија. (ИА)

8.1. Уколико систем не може да учита пса, он приказује кориснику поруку: “Пас чије податке захтевате није пронађен!”. Прекида се извршење сценарија. (ИА)

11.1. Уколико систем не може да сачува податке о псу, он приказује кориснику поруку: “Грешка приликом чувања података о псу!”. (ИА)

Слика 38 - *Грешка приликом чувања података о псу*

СК6: Случај коришћења – Унос новог инструктора

Назив СК
Унос новог инструктора

Актори СК
Админ

Учесници СК
Админ и систем (програм)

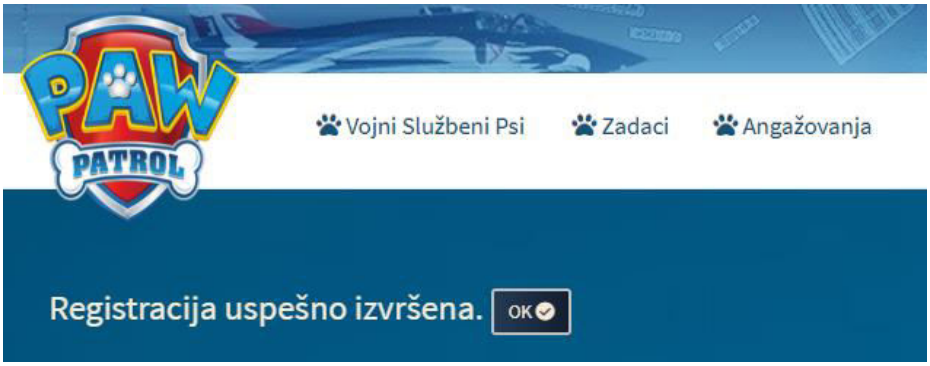
Предуслов: Систем је укључен и админ је пријављен на систем под својом шифром. Систем приказује форму за рад са инструктором. Учитана је листа обука и листа чинова.

Слика 39 - *Форма за регистрацију*

Основни сценарио СК
1. Админ **уноси** податке о инструктору. (АПУСО)
2. Админ **контролише** да ли је коректно унео податке о инструктору. (АНСО)
3. Админ **позива** систем да запамти податке о инструктору. (АПСО)

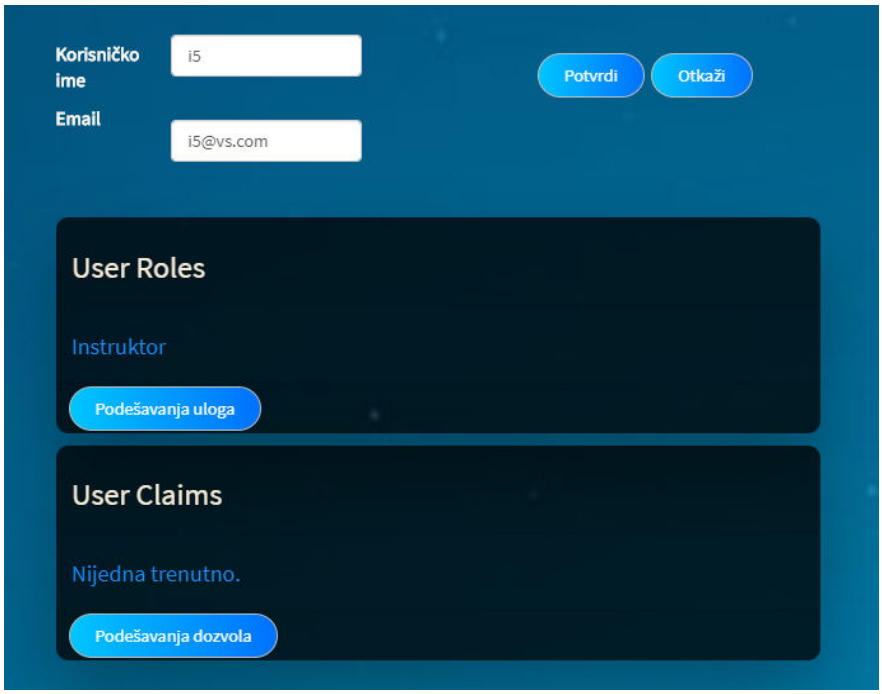
Опис акције: Корисник кликом на дугме „Registruj instruktora“ позива системску операцију RegistrujInstruktora(Instruktor)

4. Систем **памти** податке о инструктору. (СО)
5. Систем **приказује** админу поруку: “Регистрација инструктора успешна!“. (ИА)



Слика 40 - *Регистрација успешна*

Приликом регистрације кориснику се додељује улога „Инструктор“.



Слика 41 - *Додела улоге*

Алтернативна сценарија

5.1. Уколико систем не може да запамти податке о инструктору, он приказује админу поруку: “Дошло је до грешке приликом обрађивања захтева!” .(ИА)

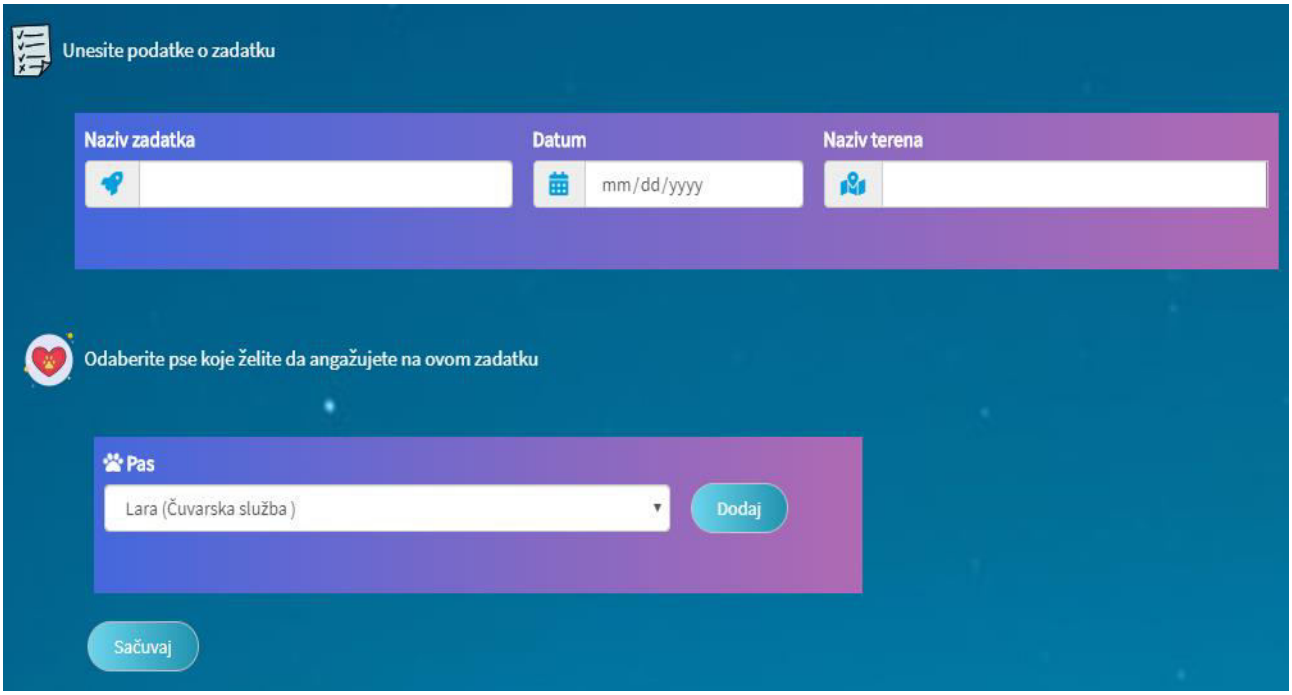
СК7: Случај коришћења – Унос новог задатка

Назив СК
Унос новог задатка

Актори СК
Админ
Учесници СК

Админ и систем (програм)

Предуслов: Систем је укључен и админ је пријављен на систем под својом шифром. Систем приказује форму за рад са задатком. Учитана је листа свих паса.

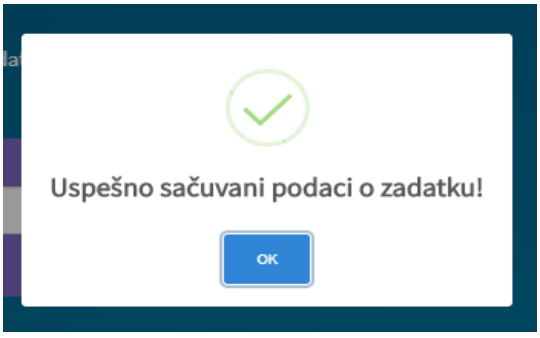


Слика 42 - *Форма за унос задатка*

Основни сценарио СК
1. Админ **уноси** податке о задатку. (АПУСО)
2. Админ **контролише** да ли је коректно унео податке о задатку. (АНСО)
3. Админ **позива** систем да запамти податке о задатку. (АПСО)

Опис акције: Корисник кликом на дугме „Sačuvaj“ позива системску операцију ZapamtiZadatak(Zadatak)

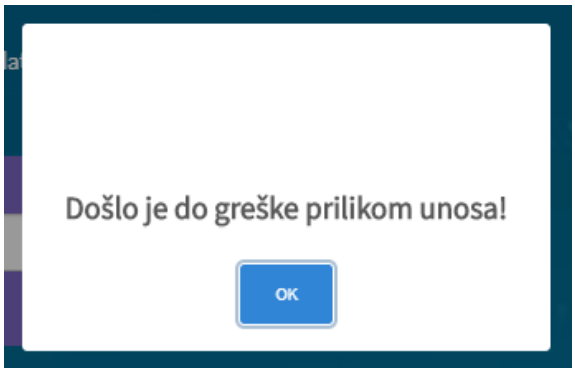
4. Систем **памти** податке о задатку. (СО)
5. Систем **приказује** админу поруку: “Успешно сачувани подаци о задатку!“. (ИА)



Слика 43 - *Успешно сачувани подаци о задатку*

Алтернативна сценарија

5.1. Уколико систем не може да запамти податке о задатку, он приказује админу поруку: “Дошло је до грешке приликом уноса!”. (ИА)



Слика 44 - *Дошло је до грешке приликом уноса*

СК8: Случај коришћења – Оцењивање ангажовања завршеног задатка

Назив СК
Оцењивање ангажовања завршеног задатка

Актори СК
Корисник (админ или инструктор)

Учесници СК
Корисник и систем (програм)

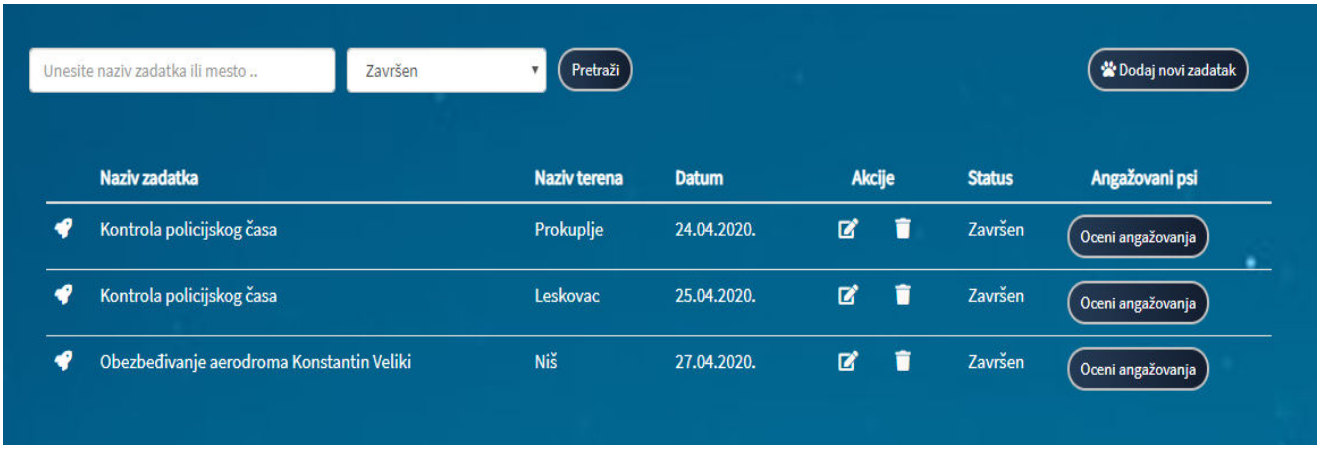
Предуслов: Систем је укључен и корисник је пријављен на систем под својом шифром. Систем приказује форму за рад са задацима. Учитана је листа задатка.

Слика 45 – *Форма за претрагу задатака*

Основни сценарио СК

1. Админ **уноси** вредност по којој претражује задатке. (АПУСО)
2. Админ **позива** систем да нађе задатке по задатој вредности. (АПСО)

3. Систем **тражи** задатке по задатој вредности. (СО)
4. Систем **приказује** кориснику листу задатка. (ИА)

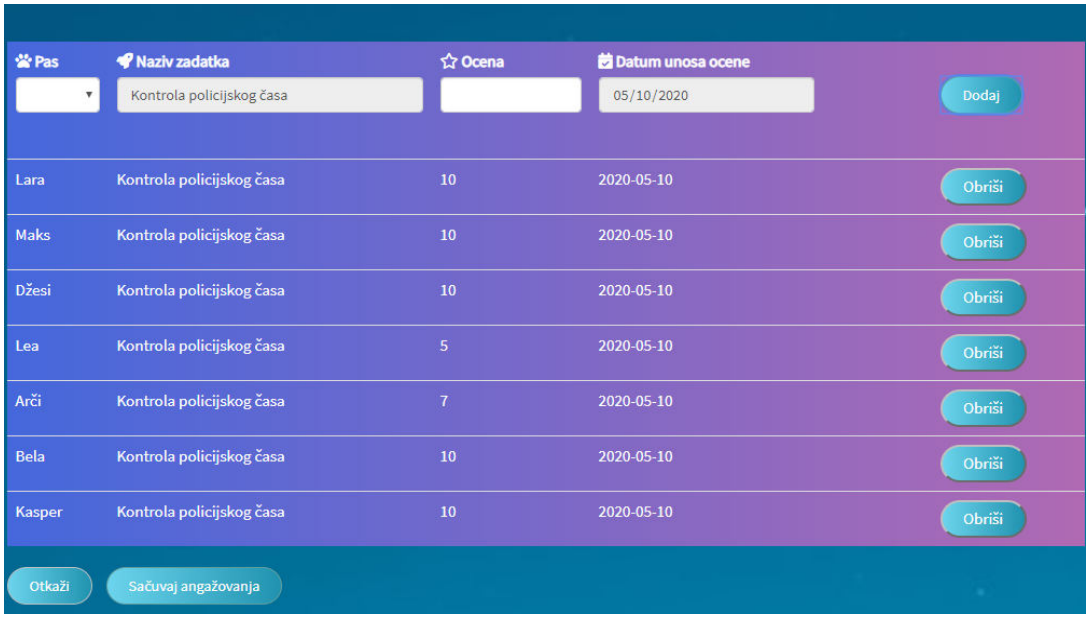


Слика 46 - ***Листа задатака***

5. Корисник позива систем да прикаже ангажовања на одабраном задатаку. (АПСО)

Опис акције: Корисник кликом на дугме „Oceni angažovanja“ позива системску операцију VratiAngazovanja(ZadatakId)

6. Систем приказује податке о ангажовањима на одабраном задатку. (ИА)
7. Корисник уноси оцене. (АПУСО)
8. Корисник контролише да ли је коректно унео податке о оцене. (АНСО)

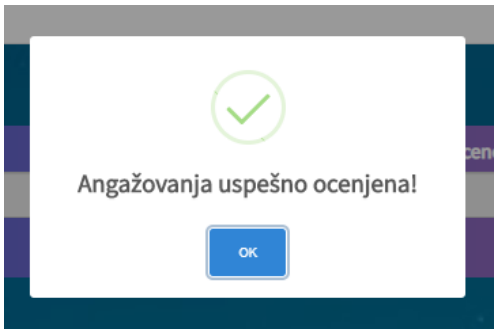


Слика 47 - ***Форма за оцењивање ангажовања на задатку***

9. Корисник позива систем да запамти оцене. (АПСО)

Опис акције: Корисник кликом на дугме „Sačuvaj angažovanja“ позива системску операцију SacuvajAngazovanja(Angazovanja)

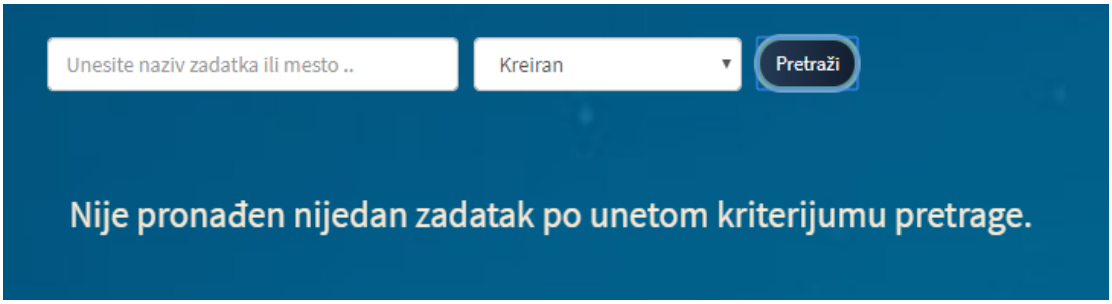
10. Систем памти оцене.(СО)
11. Систем приказује кориснику поруку: “ Ангажовања успешно оцењена!“. (ИА)



Слика 48 - *Ангажовања успешно оцењена*

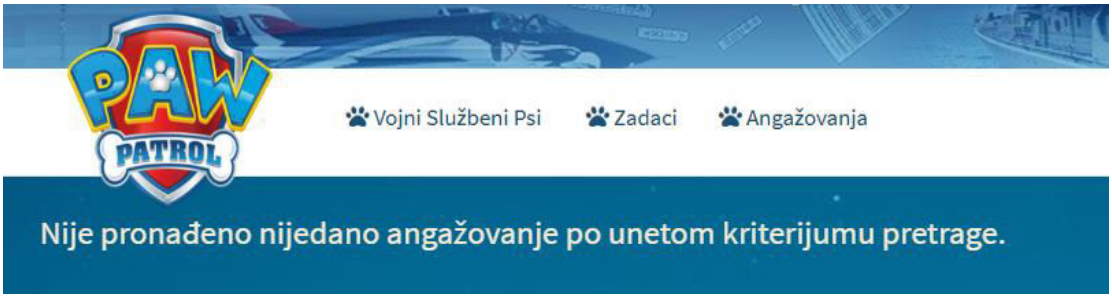
Алтернативна сценарија

4.1. Уколико систем не може да нађе задатке, систем приказује админу поруку: “Није пронађен ниједан задатак по унетом критеријуму претраге”. Прекида се извршење сценарија. (ИА)



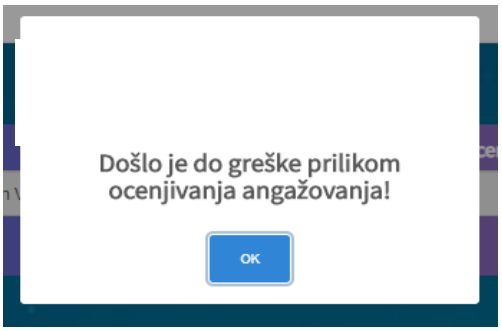
Слика 49 - *Није пронађен ниједан задатак по унетом критеријуму претраге*

6.1. Уколико систем не може да прикаже податке о ангажовањима на изабраном задатку, он приказује кориснику поруку: “Није пронађено ниједно ангажовање по унетом критерујуму претаге. “ Прекида се извршење сценарија. (ИА)



Слика 50 - *Није пронађено ниједно ангажовање по унетом критерујуму претаге*

7.1. Уколико систем не може да запамти податке о оцењеним ангажовањима, он приказује кориснику поруку: “Дошло је до грешке приликом оцењивања ангажовања!”. (ИА)



Слика 51 - *Дошло је до грешке приликом оцењивања ангажовања*

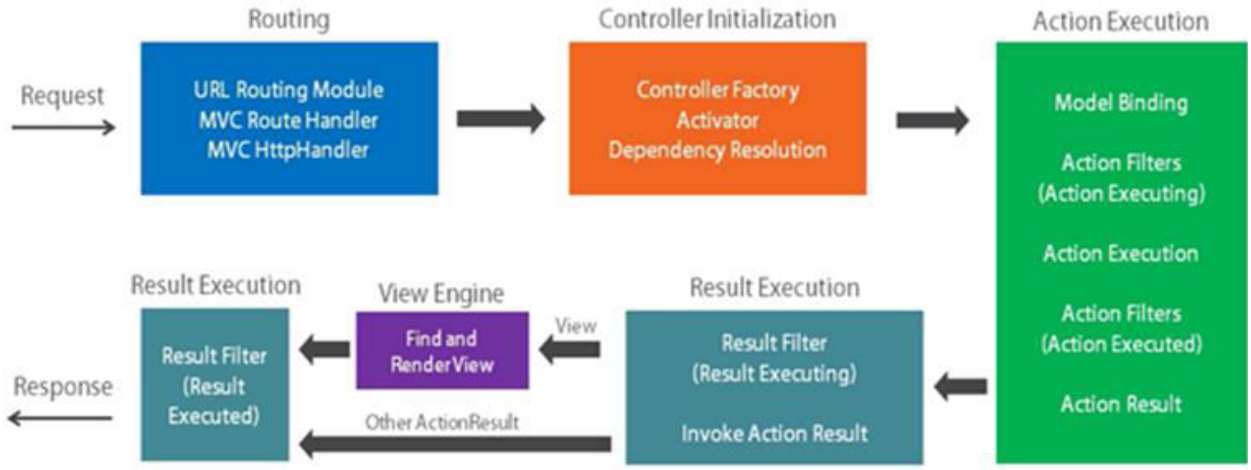
6.3 Пројектовање апликационе логике

Апликациона логика садржи класе које су неопходне за имплементацију пословне логике, а то су:

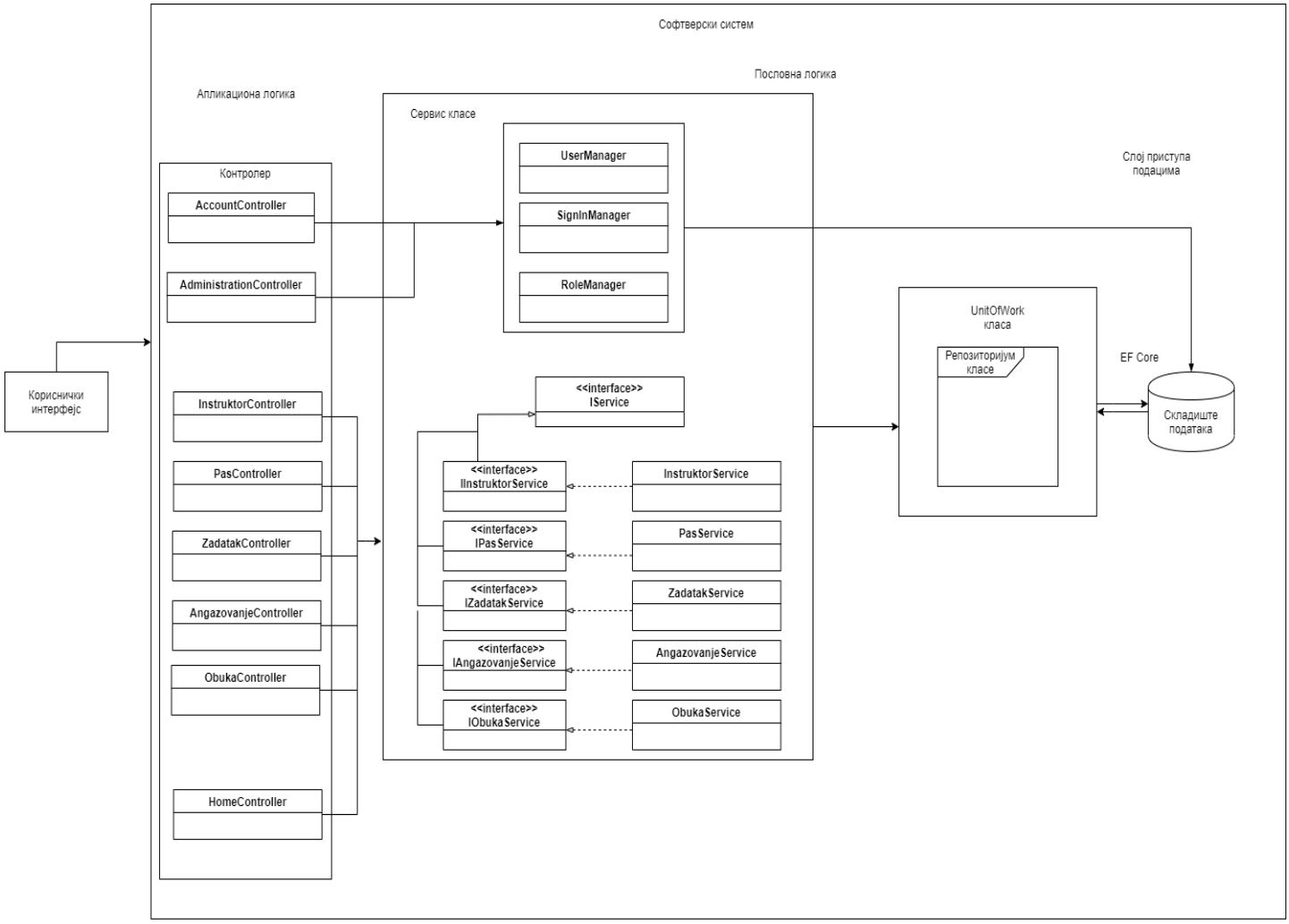
- Контролери – прихватају захтеве и прослеђују их сервисима на обраду
- Сервиси – обрађују захтеве који су пристигли, трансформише пристигле објекте у облик потребан за комуникацију са базом
- *UnitOfWork* класа – осигурава да када се користе више *Repository* инстанца, оне деле један објекат класе *DatabaseContext* како не би дошло до конфилкта са базом, координира складиштење измена и решавање проблема када је у питању конкурентност
- *Repository* класе – служе за комуникацију са базом
- *Entity* објекти – представљају табеле у бази података, резултат објектно-релационог мапирања

6.3.1 Контролер апликационе логике

Контролер прихвата захтеве који пристижу са клијентске стране и прослеђује их сервисима на обраду. Након обраде, контролер прихвата одговор и враћа назад клијенту одгвор.



Слика 52 - *Ток корисничког захтева*. (n.d.). [Digital image]. https://gwb.blob.core.windows.net/chetan/Windows-Live-Writer/1f4f73242d05_F000/image_thumb.png



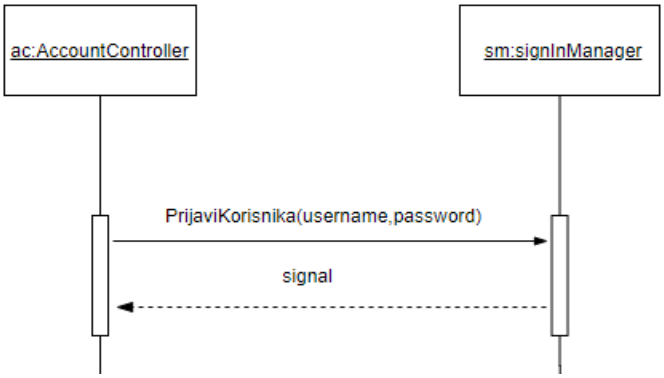
Слика 55 - ***Архитектура софтверског система након пројектовања контролера и класа које чине апликациону логику***

6.3.2 Пословна логика

Пословна логика је описана *структуром (доменским класама)* и *понашањем (системским операцијама)*. За сваки од уговора системских операција дефинисаних у фази анализе пројектује се концептуално решење [11].

Уговор UG1: PrijaviKorisnika

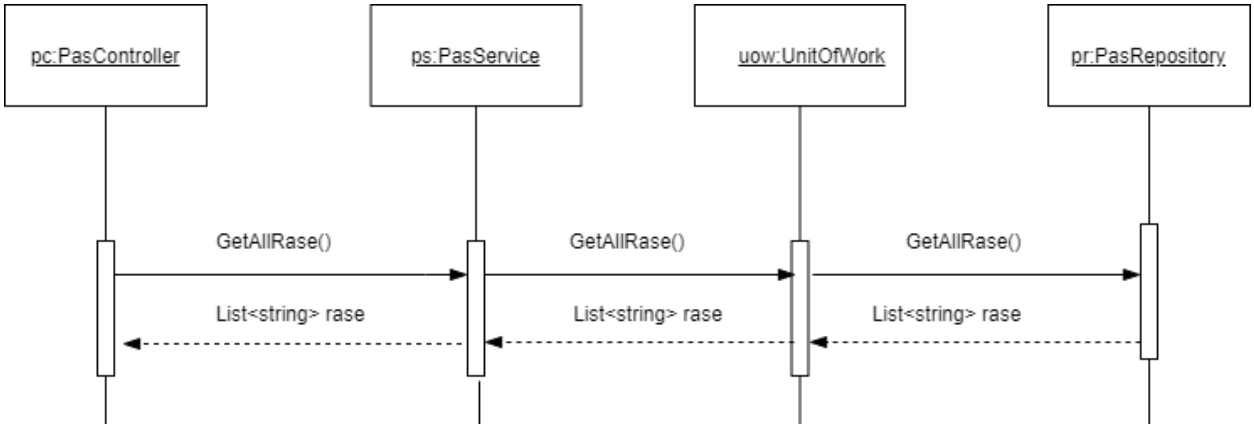
Операција: PrijaviKorisnika(Korisnik): signal;
Веза са СК: **СК1**
Предуслови: Вредносна и структурна ограничења над објектом ApplicationUser морају бити задовољена.
Постуслови: Korisnik је пријављен на систем.



Слика 53 - *Дијаграм секвенци: Уговор - PrijaviKorisnika*

Уговор UG2: UcitajListuRasa

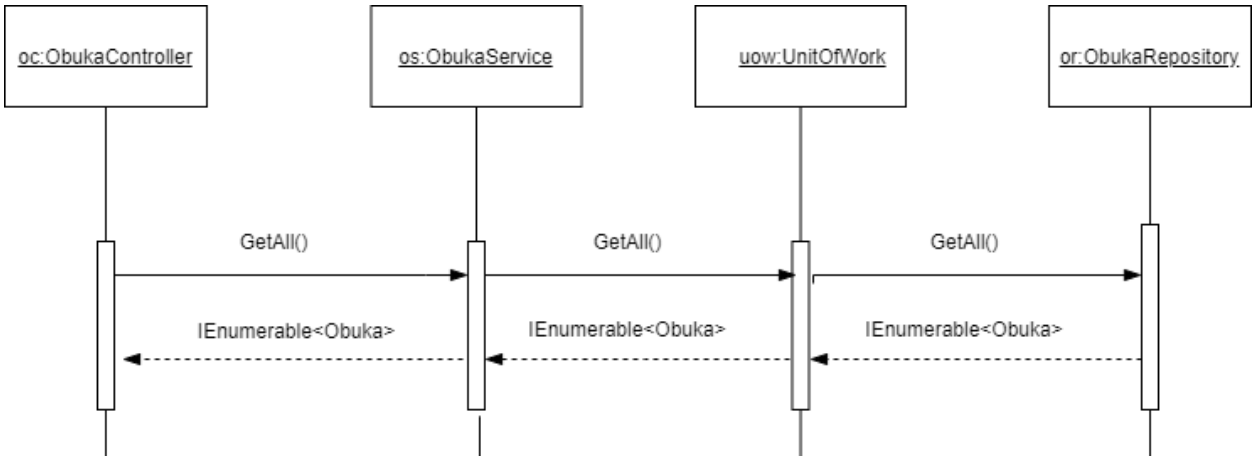
Операција: UcitajListuRasa(List<string> rase): signal;
Веза са СК: **СК2**
Предуслови: /
Постуслови: /



Слика 54 - *Дијаграм секвенци: Уговор - UcitajListuRasa*

Уговор UG3: UcitajListuObuka

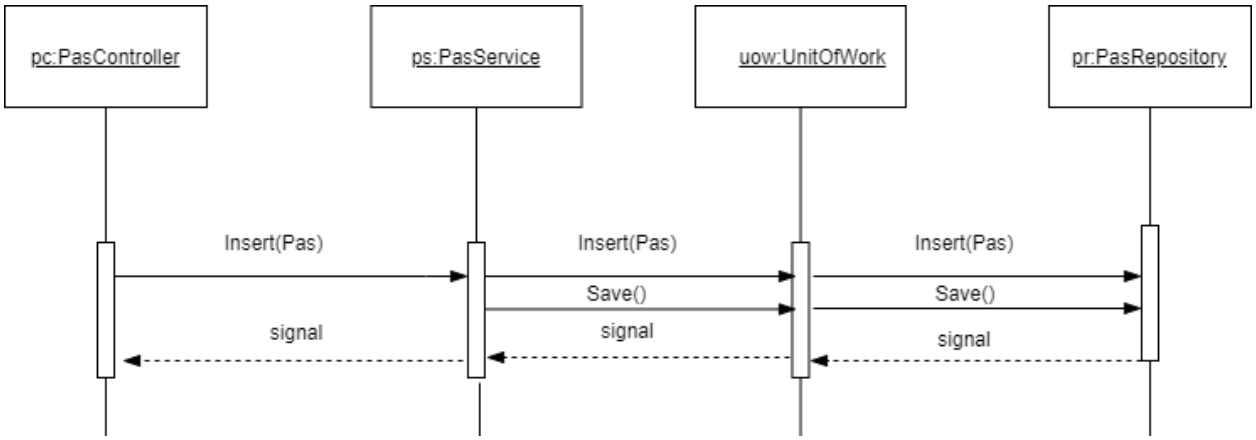
Операција: UcitajListuObuka(List<Obuka> obuke): signal;
Веза са СК: **СК2, СК3, СК4, СК5, СК6**
Предуслови: /
Постуслови: /



Слика 55 - *Дијаграм секвенци: Уговор - UcitajListuObuka*

Уговор UG4: ZapamtiPsa

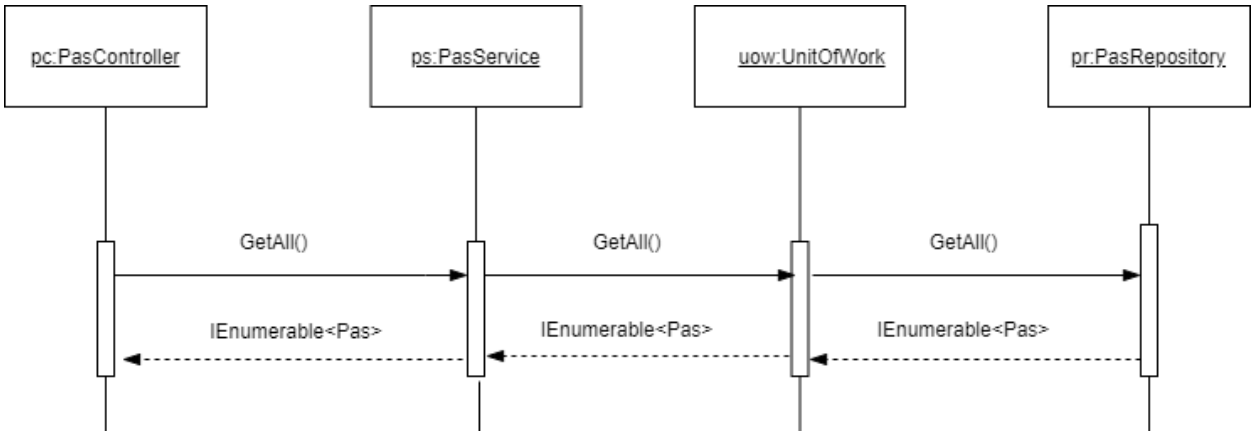
Операција: ZapamtiPsa(Pas): signal;
Веза са СК: **СК2**
Предуслови: Вредносна и структурна ограничења над објектом Пас морају бити задовољена.
Постуслови: Подаци о псу су запамћени.



Слика 56 - *Дијаграм секвенци: Уговор - ZapamtiPsa*

Уговор UG5: UcitajListuPasa

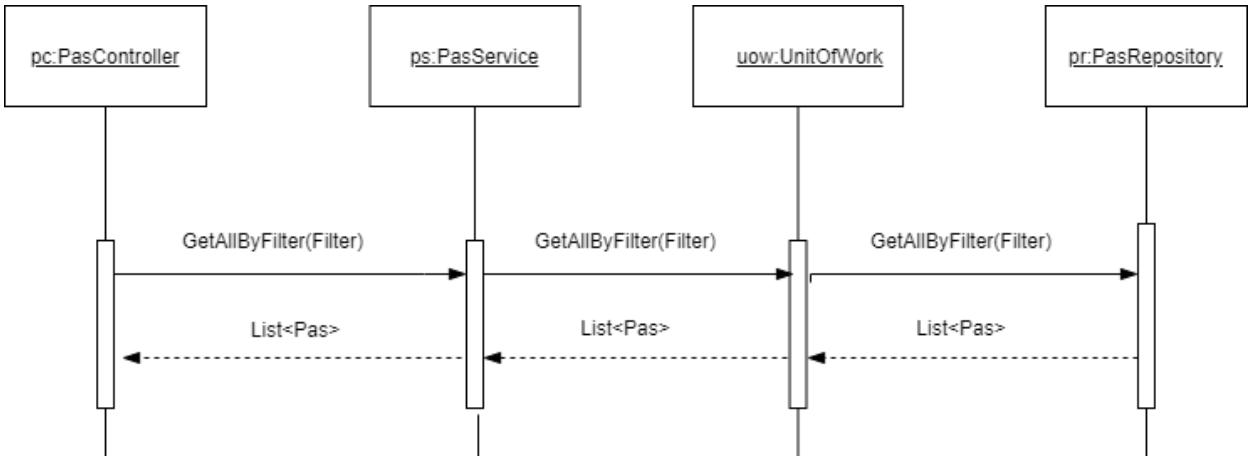
Операција: UcitajListuPasa(List<Pas> listaPasa): signal;
Веза са СК: **СК3, СК4, СК5, СК7**
Предуслови: /
Постуслови: /



Слика 57 - *Дијаграм секвенци: Уговор - UcitajListuPasa*

Уговор UG6: PronadjiPse

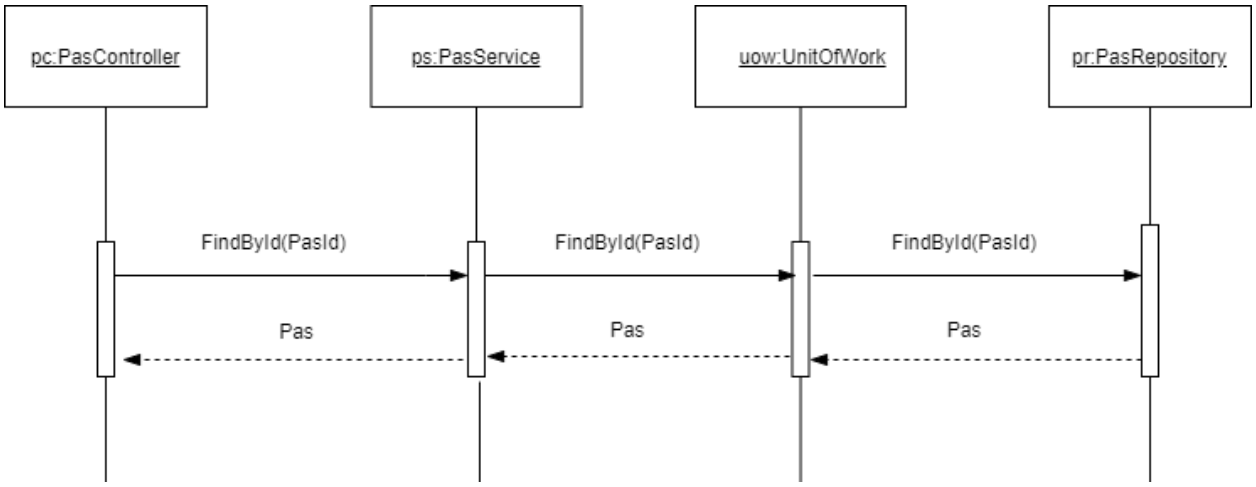
Операција: PronadjiPse(Kriterijum, List<Pas>): signal;
Веза са СК: **СК3, СК4, СК5**
Предуслови: /
Постуслови: /



Слика 58 - *Дијаграм секвенци: Уговор - PronadjiPse*

Уговор UG7: VratiPsa

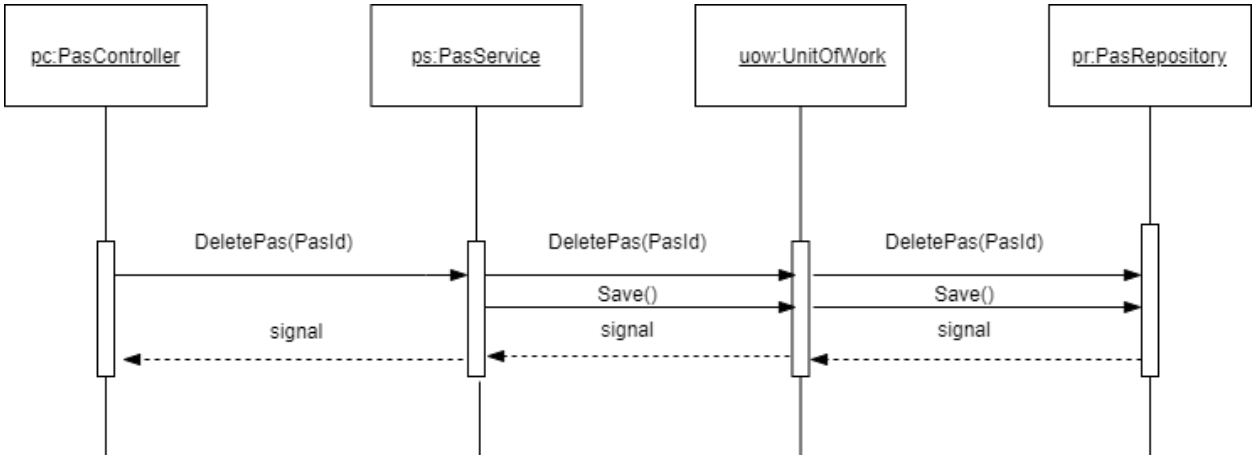
Операција: VratiPsa(PasId): signal;
Веза са СК: **СК4, СК5**
Предуслови: /
Постуслови: /



Слика 59 - *Дијаграм секвенци: Уговор - VratiPsa*

Уговор UG8: ObrisiPsa

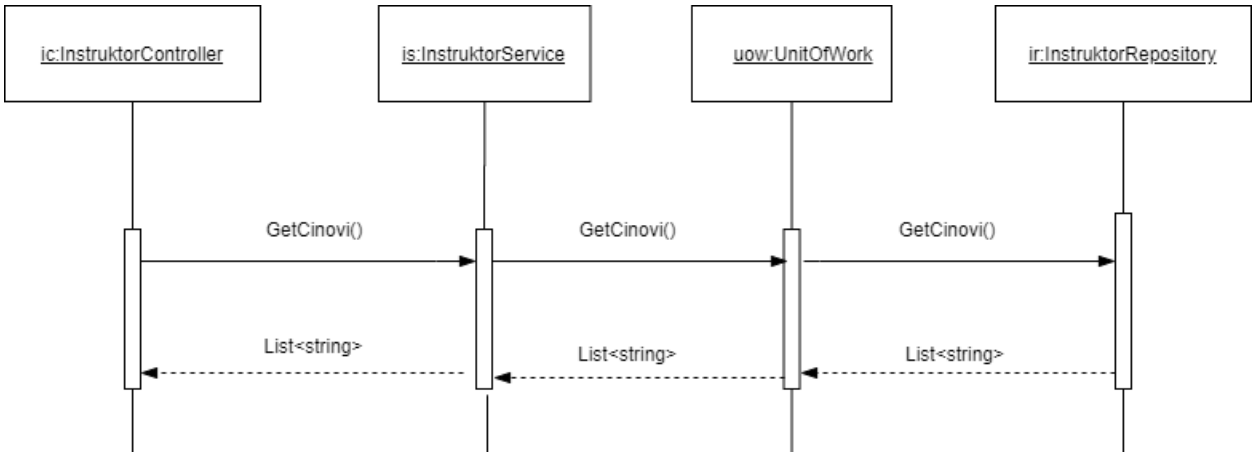
Операција: ObrisiPsa(PasId): signal;
Веза са СК: **СК4**
Предуслови: Структурна ограничења над објектом Пас морају бити задовољена.
Постуслови: Подаци о псу су обрисани.



Слика 60 - *Дијаграм секвенци: Уговор - ObrisiPsa*

Уговор UG9: UcitajListuCinova

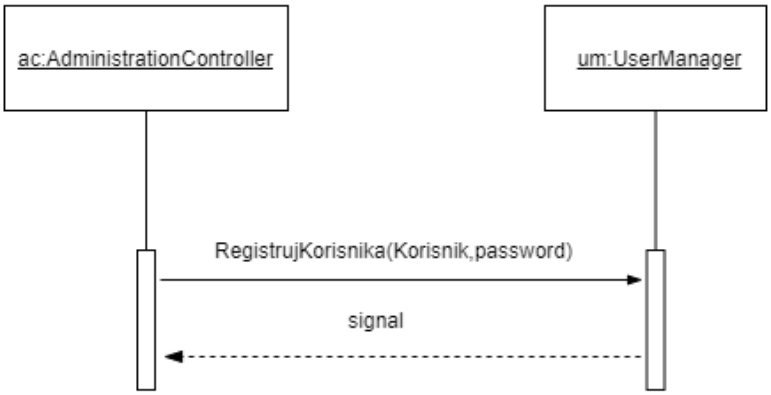
Операција: UcitajListuCinova(List<string> cinovi) : signal;
Веза са СК: **СК6**
Предуслови: /
Постуслови: /



Слика 61 - *Дијаграм секвенци: Уговор - UcitajListuCinova*

Уговор UG10: RegistrujInstruktora

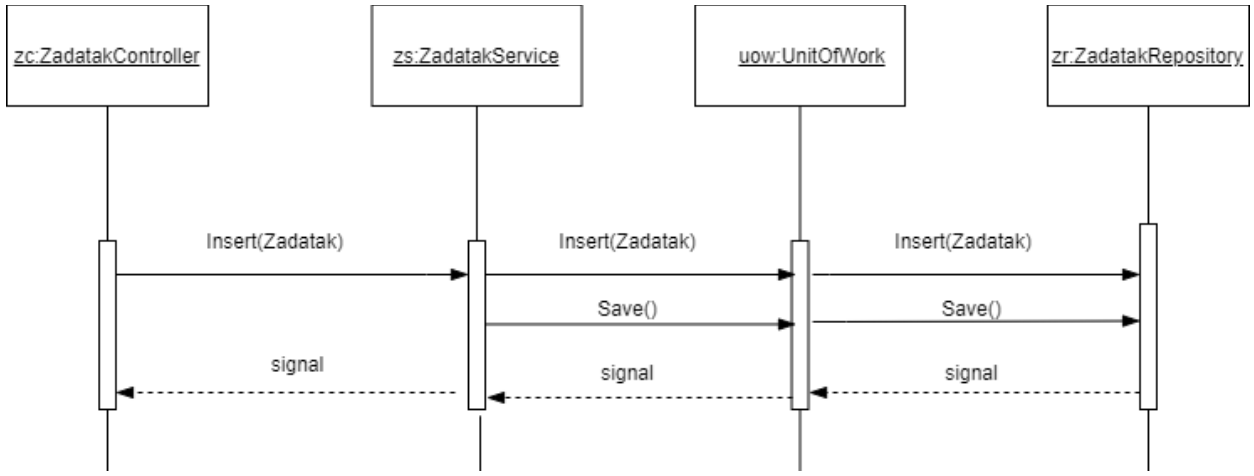
Операција: RegistrujInstruktora (Instruktor): signal;
Веза са СК: **СК6**
Предуслови: Вредносна и структурна ограничења над објектом Инструктор морају бити задовољена.
Постуслови: Инструктор је регистрован.



Слика 62 - *Дијаграм секвенци: Уговор - RegistrujInstruktora*

Уговор UG11: ZapamtiZadatak

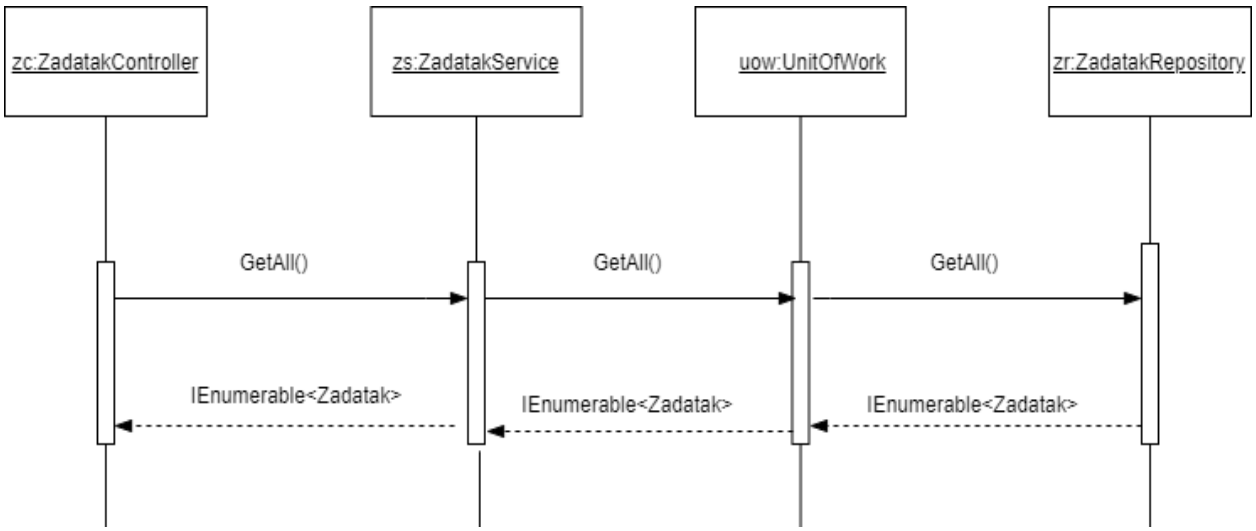
Операција: UnosNovogZadatka(Zadatak): signal;
Веза са СК: **СК7**
Предуслови: Вредносна и структурна ограничења над објектом Задатак морају бити задовољена.
Постуслови: Подаци о задатку су запамћени.



Слика 63 - *Дијаграм секвенци: Уговор - ZapamtiZadatak*

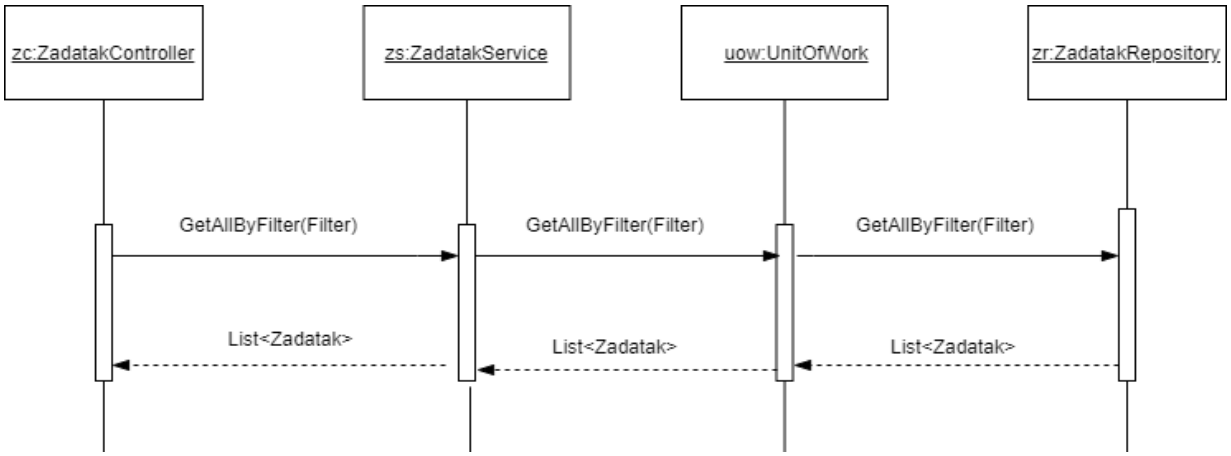
Уговор UG12: UcitajListuZadataka

Операција: UcitajListuZadataka(List<Zadatak> listaZadataka): signal;
Веза са СК: **СК8**
Предуслови: /
Постуслови: /



Слика 64 - *Дијаграм секвенци: Уговор - UcitajListuZadataka*

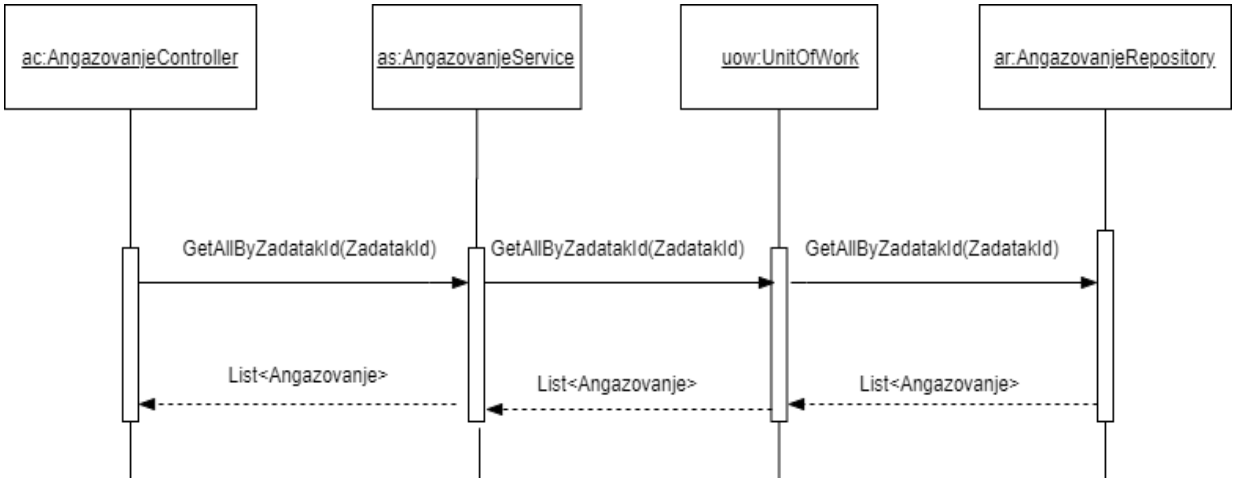
Уговор UG13: PronadjiZadatake
Операција: PronadjiZadatake(KriterijumPretrage, List<Zadatak>): signal;
Веза са СК: **СК8**
Предуслови: /
Постуслови: /



Слика 65 - *Дијаграм секвенци: Уговор - PronadjiZadatake*

Уговор UG14: VratiAngazovanja

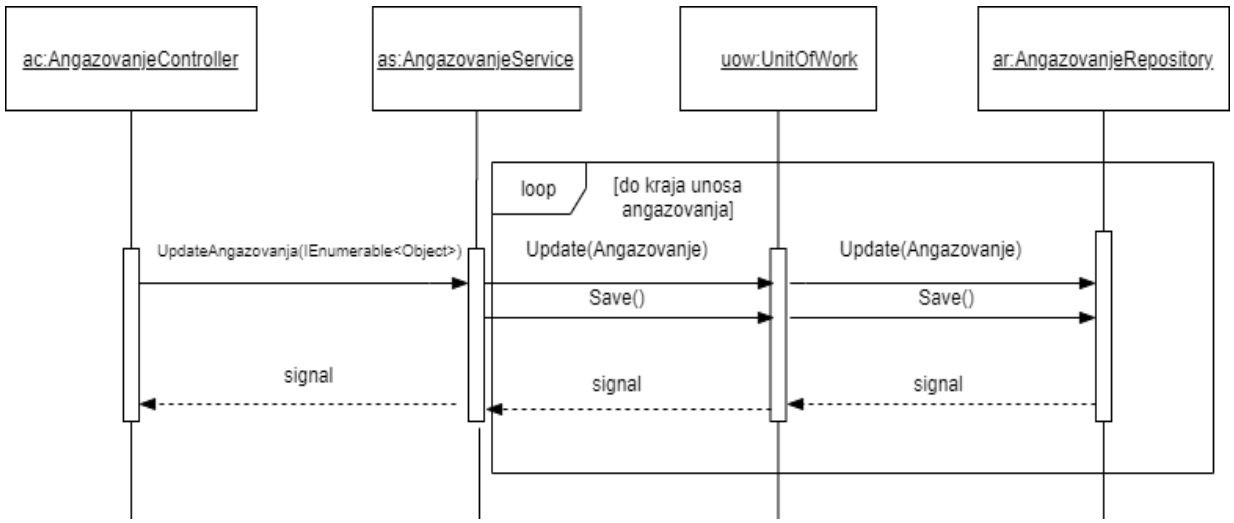
Операција: VratiAngazovanja(ZadatakId): signal;
Веза са СК: **СК8**
Предуслови: /
Постуслови: /



Слика 66 - *Дијаграм секвенци: Уговор - VratiAngazovanja*

Уговор UG15: SacuvajAngazovanja

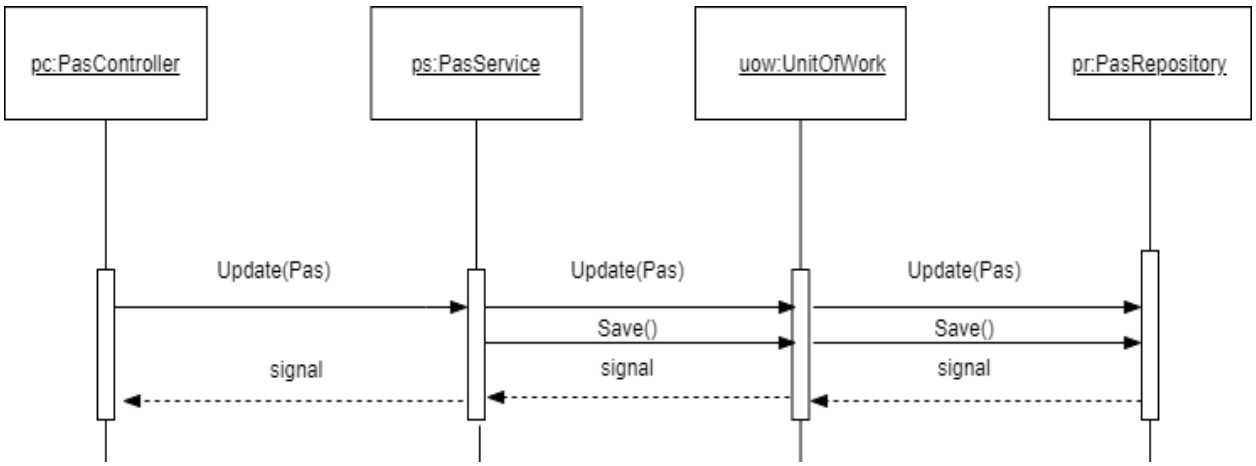
Операција: SacuvajAngazovanja(List<Angazovanja> angazovanja): signal;
Веза са СК: **СК8**
Предуслови: Вредносна и структурна ограничења над објектом Ангажовање морају бити задовољена.
Постуслови: Ангажовања су замапћена.



Слика 67 - *Дијаграм секвенци: Уговор - SacuvajAngazovanja*

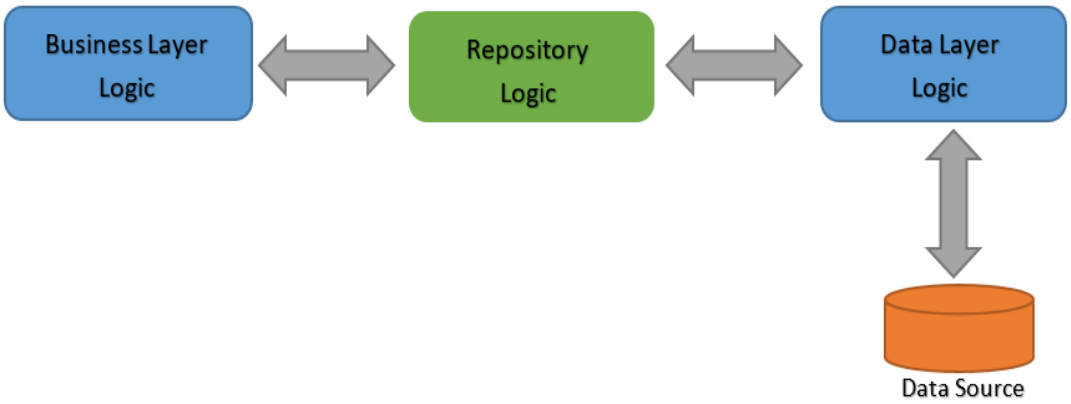
Уговор UG16: IzmeniPsa

Операција: IzmeniPsa(Pas): signal;
Веза са СК: **СК5**
Предуслови: Вредносна и структурна ограничења над објектом Пас морају бити задовољена.
Постуслови: Измењени подаци о псу су запамћени.



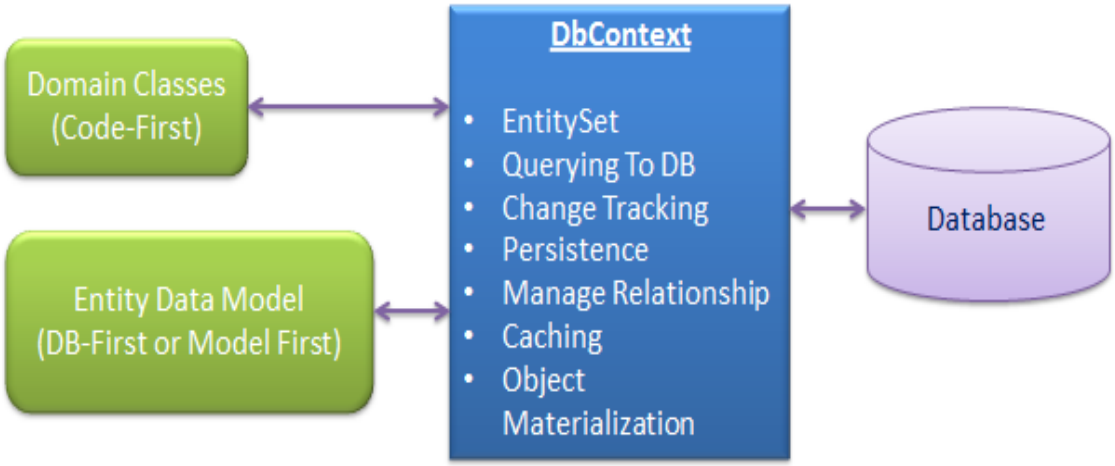
Слика 68- *Дијаграм секвенци: Уговор - IzmeniPsa*

6.4 Комуникација између пословне логике и складишта података

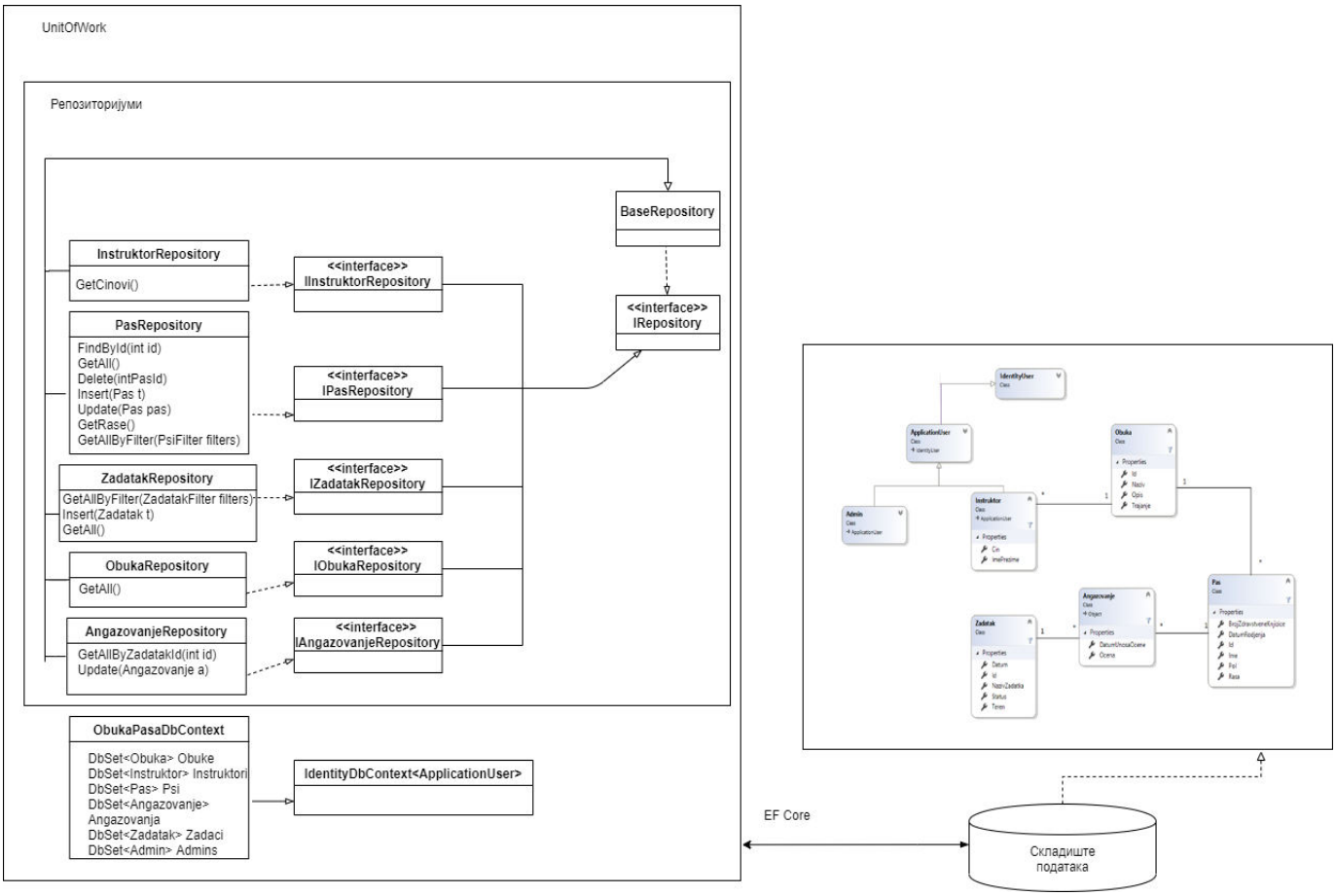


Слика 69 - *Комуникација између пословне логике и складишта података*. (n.d.). [Digital image]. https://miro.medium.com/max/1400/1*toUMfjUegcj9Ix7BDXdk5A.png

За комуникацију између пословне логике и складишта података одговорне су *Repository* класе. Оне представљају апстрактни слој између слоја пословне логике система и слоја података. Ове класе преко *DBContext* класе остварују комуникацију са базом. Свака класа кроз конструктор инстацира објекат класе *DatabaseContext* и садржи методе преко којих се уз помоћ *Entity Framework Core*-а врши приступ и чување података и остварују основне *CRUD* операције. Ова класа изведена од *DatabaseContext* класе је главна одговорна за интеракцију са базом. Свака инстанца класе контекста представља сесију са базом.



Слика 70 - *структура Database Context класе*. (n.d.). [Digital image]. https://i.pinimg.com/564x/4c/c3/61/4cc36120401d11d1002bd0afeafdc26f.jpg



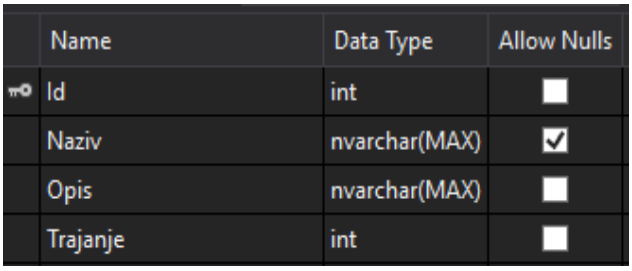
Слика 71 - *Пројектовање апстрактног слоја између пословне логике и складишта података*

6.5 Пројектовање складишта података

На основу доменских класа, пројектоване су табеле релационог система за управљање базом података.

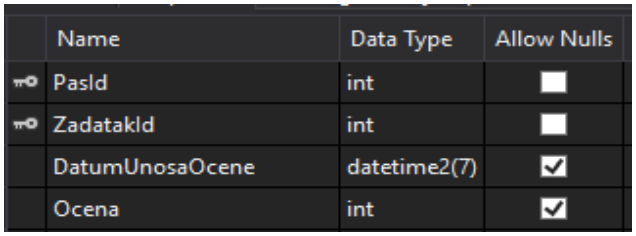
Name	Data Type	Allow Nulls
Id	int	☐
BrojZdravstveneKnjizice	nvarchar(MAX)	☐
Ime	nvarchar(MAX)	☐
DatumRodjenja	datetime2(7)	☐
Rasa	nvarchar(MAX)	☐
Pol	nvarchar(MAX)	☐
ObukaId	int	☐

Слика 72 - *Табела Pas*



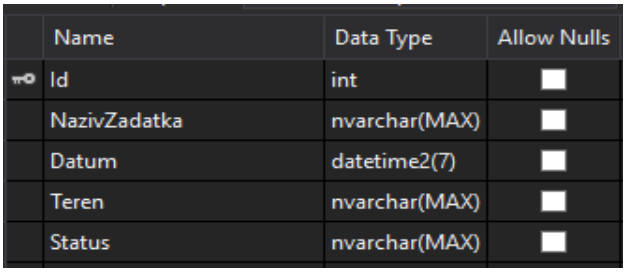
	Name	Data Type	Allow Nulls
🔑	Id	int	☐
	Naziv	nvarchar(MAX)	☑
	Opis	nvarchar(MAX)	☐
	Trajanje	int	☐

Слика 73 - *Табела Obuka*



	Name	Data Type	Allow Nulls
🔑	PasId	int	☐
🔑	ZadatakId	int	☐
	DatumUnosaOcene	datetime2(7)	☑
	Ocena	int	☑

Слика 74 - *Табела Angazovanje*



	Name	Data Type	Allow Nulls
🔑	Id	int	☐
	NazivZadatka	nvarchar(MAX)	☐
	Datum	datetime2(7)	☐
	Teren	nvarchar(MAX)	☐
	Status	nvarchar(MAX)	☐

Слика 75 - *Табела Zadatak*

Коришћењем *ASP.NET Identity API* – ја, који се користи за управљање корисцнима, лозинкама, подацима корисничког профила, улогама, захтевима, токенима, потврдом електронске поште и слично, аутоматски се генеришу следеће табеле које су коришћене за потребе овог система:

Update | Script File: dbo.AspNetUsers.sql

Name	Data Type	Allow Nulls
Id	nvarchar(450)	☐
UserName	nvarchar(256)	☑
NormalizedUserName	nvarchar(256)	☑
Email	nvarchar(256)	☑
NormalizedEmail	nvarchar(256)	☑
EmailConfirmed	bit	☐
PasswordHash	nvarchar(MAX)	☑
SecurityStamp	nvarchar(MAX)	☑
ConcurrencyStamp	nvarchar(MAX)	☑
PhoneNumber	nvarchar(MAX)	☑
PhoneNumberConfirmed	bit	☐
TwoFactorEnabled	bit	☐
LockoutEnd	datetimeoffset(7)	☑
LockoutEnabled	bit	☐
AccessFailedCount	int	☐
Discriminator	nvarchar(MAX)	☐
ImePrezime	nvarchar(MAX)	☑
Cin	nvarchar(MAX)	☑
Obukald	int	☑

Слика 76 - *Табела AspNetUsers*

Name	Data Type	Allow Nulls
Id	nvarchar(450)	☐
Name	nvarchar(256)	☑
NormalizedName	nvarchar(256)	☑
ConcurrencyStamp	nvarchar(MAX)	☑

Слика 77 - *Табела AspNetRoles*

Update | Script File: dbo.AspNetUserRoles.sql

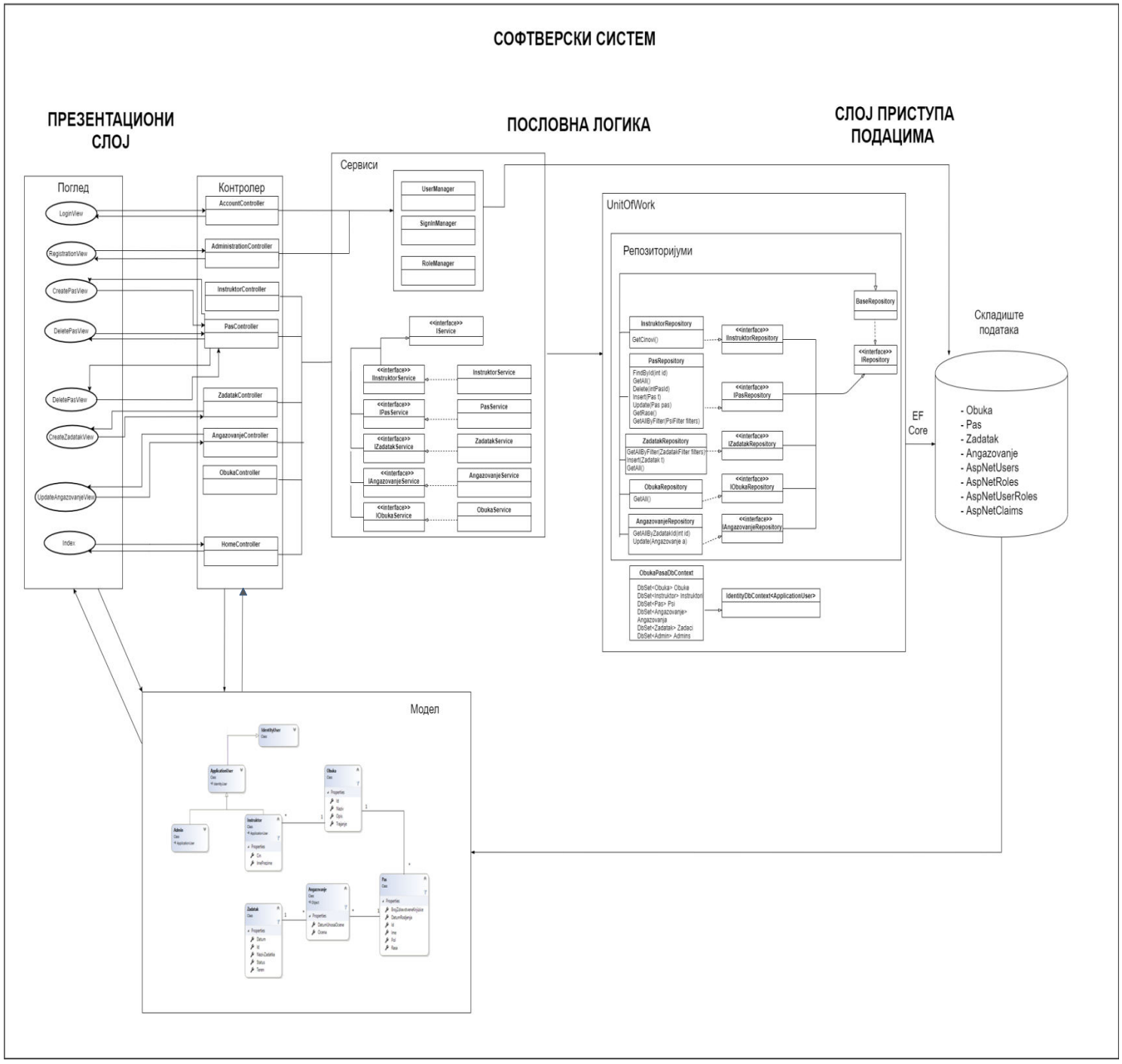
Name	Data Type	Allow Nulls
UserId	nvarchar(450)	☐
RoleId	nvarchar(450)	☐

Слика 78 - *Табела AspNetUserRoles*

6.6 Коначан изглед архитектуре софтверског система

Као резултат фазе пројектовања добија се коначна архитектура софтверског система. Погледи су одговорни за представљање садржаја путем корисничког интерфејса, контролери управљају интеракцијом корисника, раде с моделом и на крају бирају поглед који ће се приказивати. Све захтеве који стижу од корисника са корисничког интерфејса обрађује одговарајући контролер.

Контролери имају референце на потребне сервисе. Примљене захтеве контролери шаљу на обраду одговарајућим сервисима. Сервиси имају референце на *UnitOfWork* класу која представља јединицу рада и која има референце на *Repository* класе. Repository класе служе за комуникацију са базом података. Табеле у бази података су креиране на основу концептуалних класа, као резултат објектно-релационог мапирања.



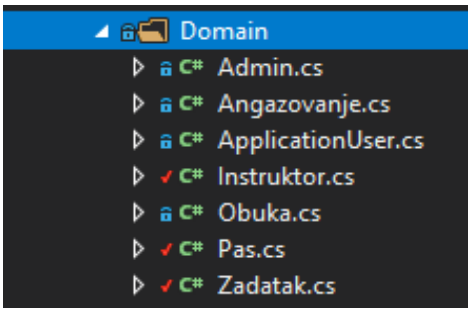
Слика 79- ***Коначна архитектура софтверског система***

7. Фаза имплементација

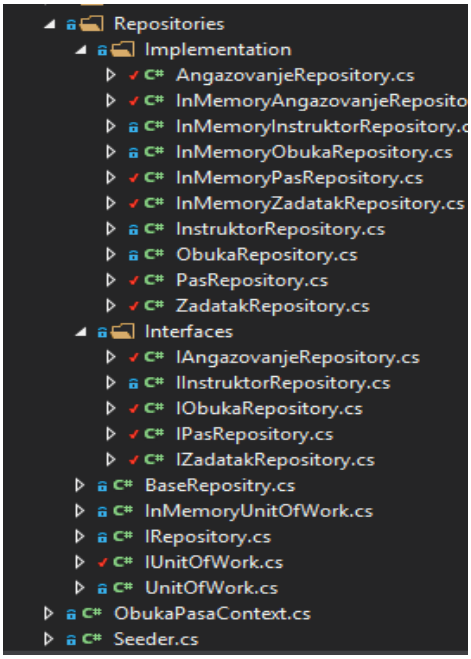
Веб апликација која је настала као резултат овог рада развијена је у програмском језику у C#, у .NET окружењу уз помоћ *MVC* оквира. Као развојно окружење коришћен је *Microsoft Visual Studio* 2019 и апликација се извршава на *IIS* серверу. Систем за управљање базом података је *MySQL*, док је клијентска страна изграђена уз помоћ *JavaScript, jQuery, AJAX u Bootstrap* технологија. За тестирање коришћен је *Moq* оквир.

7.1 Структура софтверског решења

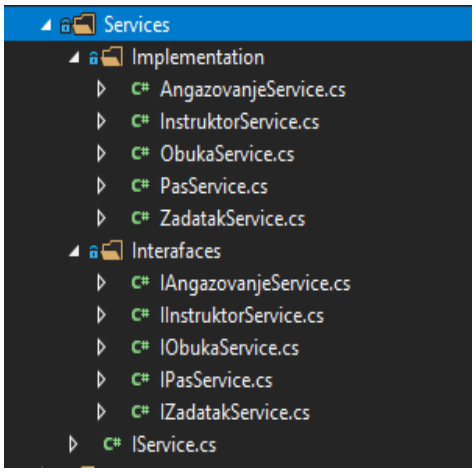
На серверској страни имплементиране су следеће класе:



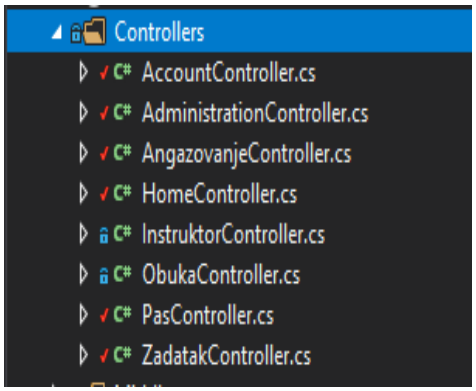
Слика 80- фолдер Domain



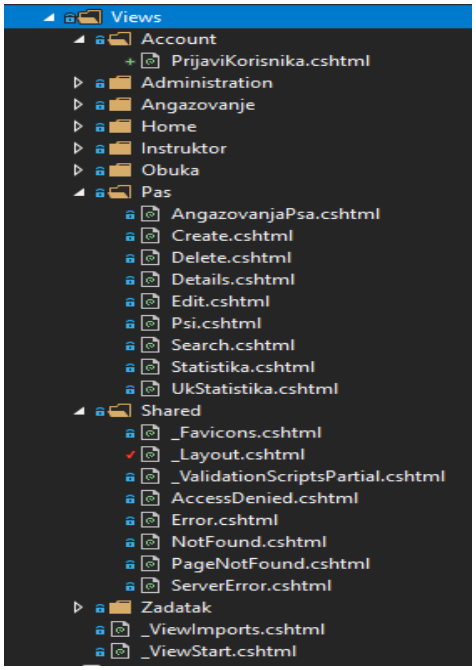
Слика 81 - фолдер Repositories



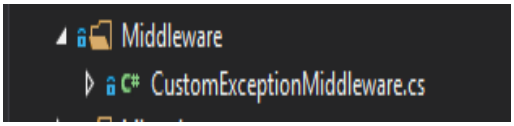
Слика 82 – фолдер Services



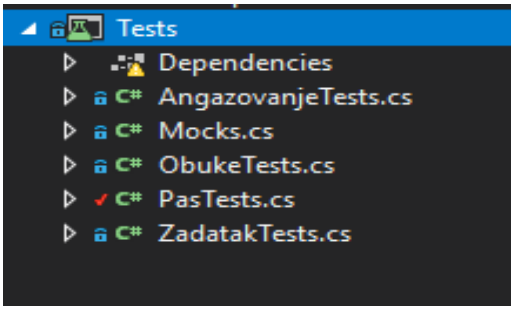
Слика 83 – фолдер Controllers



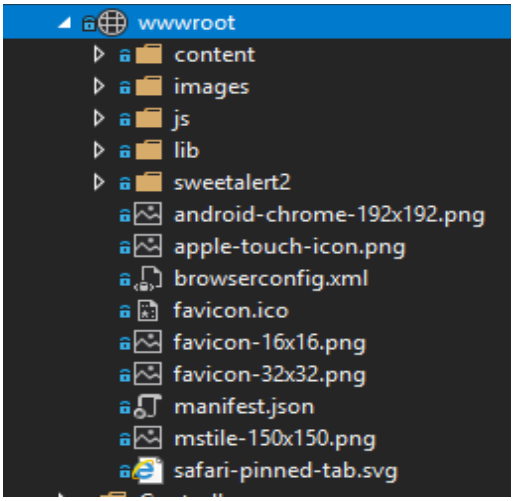
Слика 84 – фолдер Views



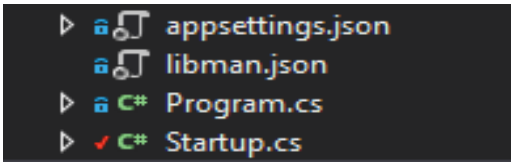
Слика 85 – фолдер Middleware



Слика 86 – фолдер Test



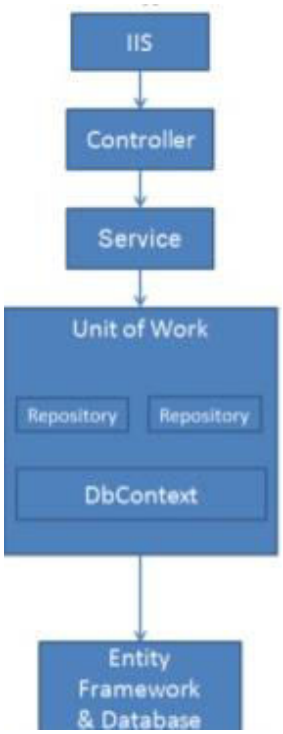
Слика 87 - фолдер root



Слика 88 - остале класе

7.2 Имплеметација апликационе логике

У овом поглављу описана је имплементација архитектуре приказане на слици:



Слика 89 - *Архитектура апликације*. (n.d.). [Digital image]. https://www.asp.net/media/2578149/Windows-Live-Writer_8c4963ba1fa3_CE3B_Repository_pattern_diagram_1df790d3-bdf2-4c11-9098-946ddd9cd884.png

7.2.1 Комуникација са клијентима

Класа Program.cs обезбеђује имплементацију *Main* методе која представља улазну тачку апликације. Ова метода позива *BuildWebHost* методу, која је одговорна за конфигурацију ASP.NET Core-а. Затим се *UseStartup* метода такође позива у овој класи, како би идентификовала класу која ће пружити конфигурацију специфичну за ову апликацију. *Build* метода обрађује сва подешавања конфигурације и креира објекат који имплементира интерфејс *IHostBuilder*, путем чега започиње обрађивање *HTTP* захтева [1]. У наставку је приказана имплементација ове класе.

```
public class Program
    {
        public static void Main(string[] args)
        {  CreateHostBuilder(args).Build().Run();  }
        public static IHostBuilder CreateHostBuilder(string[] args) =>
            Host.CreateDefaultBuilder(args)
                .ConfigureWebHostDefaults(webBuilder =>
                { webBuilder.UseStartup<Startup>()
                  .ConfigureLogging((hostingContext, logging) =>
                  {
                      logging.ClearProviders();
                      logging.AddConfiguration(hostingContext.Configuration.GetSection("Logging"));
                      logging.AddNLog();
                  });
                });
 }
```

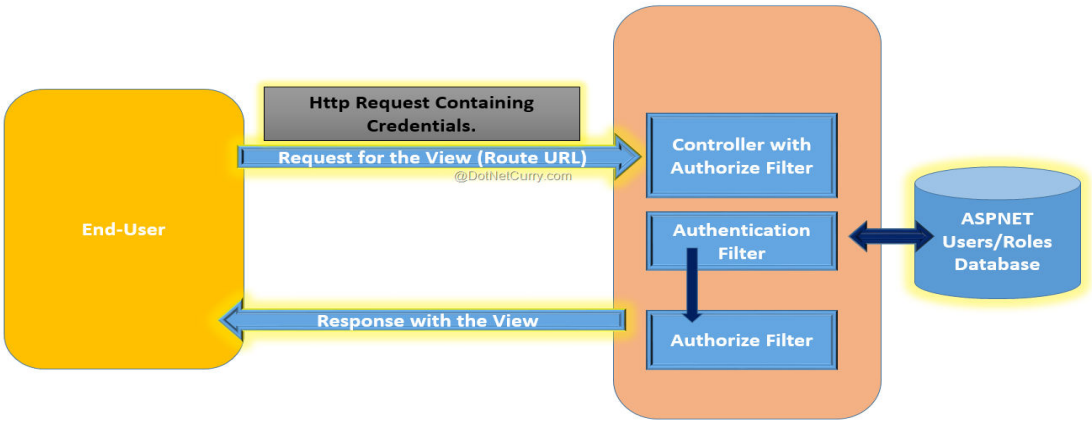
Upravljanje HTTP zahtevima

Интеракција са клијентима остварује се путем *HTTP* захтева. Приликом покретања апликације креира се инстанца *Startup* класе и прво се позива метода *ConfigureServices*, у којој се креирају објекти који обезбеђују разне функционалности које је неопходно креирати. Потом се позива метода *Configure* у којој се смештају *middleware* компоненте. Оне представљају ланац којим се заправо контролише извршавање захтева. Сваки захтев који стигне са *pipeline* – на обрађује се редом како онако како су *middleware* компоненте наведене. *Middleware* може да проследи захтев следећем у низу или да га обради и врати одговор. *Middleware* (у оквиру својих позива) може да користи сервисе који су наведени у методи *ConfigureServices* [1]. За потребе овог рада, имплементиран је сопствени *middleware* за обраду грешака. У наставку је приказан код имплементације ове компоненте:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env, Seeder seeder)
        {
            if (env.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();
            }
            app.UseRouting();
            app.UseCusMikeExceptionMiddleware();
                       ...

    }

namespace Obuka_Vojnih_Pasa.Middleware
{
    public class CusMikeExceptionMiddleware
    {
        private readonly RequestDelegate _next;

private readonly ILogger<CusMikeExceptionMiddleware> _logger;
        public CusMikeExceptionMiddleware(RequestDelegate next, ILogger<CusMikeExceptionMiddleware> loggerFactory)
        {
            _next = next;
            _logger = loggerFactory;
            _logger.LogDebug("NLog injected into CusMikeExceptionMiddleware");
        }
        public async Task Invoke(HTTPContext context)
        {
            try
            {
                await _next.Invoke(context);
                switch (context.Response.StatusCode)
                {
                    case (int)HTTPStatusCode.NotFound:

                        HandlePageNotFound(context); break;
                    case (int)HTTPStatusCode.InternalServerError: HandleInternalServerError(context); break;
                    default: break;
      }
            }
            catch (Exception ex)
            {
                HandleException(context, ex);
            }

        }

        private void HandleInternalServerError(HTTPContext context)
        {
            StringBuilder sb = new StringBuilder($"An error has occurred on {context.Request.Host}. \r\n \r\n");
            sb.Append($"Path = {context.Request.Path} \r\n \r\n");
            _logger.LogWarning($"500 error occured. Path = " +
           sb);
            context.Response.Redirect("/Home/ServerError");
        }
```

```
        private void HandlePageNotFound(HTTPContext context)
        {
            StringBuilder sb = new StringBuilder($"An error has occurred on {context.Request.Host}. \r\n \r\n");
            sb.Append($"Path = {context.Request.Path} \r\n \r\n");
            _logger.LogWarning(
           sb.ToString());

            string pageNotFound = context.Request.Host.ToString() + context.Request.Path.ToString();
            CookieOptions cookieOptions = new CookieOptions();
            cookieOptions.Expires = DateTime.Now.AddSeconds(20);
            cookieOptions.IsEssential = true;
            context.Response.Cookies.Append("PageNotFound", pageNotFound, cookieOptions);
            context.Response.Redirect("/PageNotFound");
        }

        private void HandleException(HTTPContext context, Exception ex)
        {
            StringBuilder sb = new StringBuilder($"An error has occurred on {context.Request.Host}. \r\n \r\n");
            sb.Append($"Path = {context.Request.Path} \r\n \r\n");
            sb.Append($"Error Message = {ex.Message} \r\n");
            sb.Append($"Error Source = {ex.Source} \r\n");
            if (ex.InnerException != null)
                sb.Append($"Inner Exception = {ex.InnerException.ToString()} \r\n");
            else
                sb.Append("Inner Exception = null \r\n");
            sb.Append($"Error StackTrace = {ex.StackTrace} \r\n");
            _logger.LogError(sb.ToString());
            context.Response.Redirect("/Error");
        }
    }
    public static class CusMikeExceptionMiddlewareExtensions
    {
        public static IApplicationBuilder UseCusMikeExceptionMiddleware(this IApplicationBuilder builder)
        {
            return builder.UseMiddleware<CusMikeExceptionMiddleware>();
        }
    }
}
```

Middleware контролише како апликација реагује на *HTTP* захтеве, који ће одговор дати, како се поступа у случају грешке, да ли је корисник аутентификован, да ли је кориснику дозвољено да користи одређени ресурс и слично. За потребе импелемнтације система коришћени су механизми аутентификације и ауторизације, путем којих се обезбеђује да се за приступ систему и одређеним страницама захтева да корисник буде пријављен, у супротном биће аутоматски преусмерен на страницу за пријаву. Захтев који стиже од корисника прослеђује се *Middleware* - u, а затим одговарајућој *Controller* класи. Такође, уколико корисник захтева страницу за коју није ауторизован, приступ ће му бити одбијен.



Слика 90 - *Ток корисничког захтева*. (n.d.-b). [Digital image]. https://www.dotnetcurry.com/images/mvc/Implementing-User-Authenticati.NET-MVC-6_E4AD/aspnet-mvc-user-auth.png

У *Startup* класи у методи *ConfigureServices* додати су следећи сервиси који омогућавају коришћење претходно описаних коцепата, *Authentication Middleware* и *Authorization Middleware* класе:

```
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddDbContext<ObukaPasaContext>(options                                                                    =>
options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));
            services.AddScoped<IInstruktorService, InstruktorService>();
            services.AddScoped<IObukaService, ObukaService>();
            services.AddScoped<IPasService, PasService>();
            services.AddScoped<IZadatakService, ZadatakService>();
            services.AddScoped<IInstruktorService, InstruktorService>();
            services.AddScoped<IAngazovanjeService, AngazovanjeService>();
            services.AddScoped<IUnitOfWork, UnitOfWork>();
            services.AddScoped<Seeder>();
            services.AddControllersWithViews(
            ).AddJsonOptions(x => x.JsonSerializerOptions.MaxDepth = Int32.MaxValue);
           services.AddAuMikeapper(this.GetType().Assembly);
            services.AddSwaggerGen(c =>
            {
                c.SwaggerDoc("v1", new OpenApiInfo() { Title = "PawPatrolAPI", Version = "v1" });
            });
            services.AddSingleton<IHTTPContextAccessor, HTTPContextAccessor>();
            services.AddSession();
            services.AddIdentity<ApplicationUser, IdentityRole>()
               .AddRoles<IdentityRole>()
              .AddDefaultTokenProviders()
              .AddEntityFrameworkStores<ObukaPasaContext>();
            services.AddMvc(options =>
            {
                var policy = new AuthorizationPolicyBuilder()
                .RequireAuthenticatedUser()
                .Build();
                options.Filters.Add(new AuthorizeFilter(policy));
            }).AddXmlSerializerFormatters();
            services.AddAuthentication();
            services.AddAuthorization(options =>
            {
                options.AddPolicy("ZadatakPolicy",
                 policy => policy.RequireClaim("Unos zadatka", "true").RequireRole("Korisnik", "Admin"));
                options.AddPolicy("AngazovanjePolicy",
                 policy => policy.RequireClaim("Brisanje angažovanja", "true").RequireRole("Korisnik", "Admin"));
            });
        }
```

Механизми аутентификације захтевају од корисника да буде пријављен на систем како би приступио функционалностима система. Након уноса креденцијала у поља форме за пријаву, контролер обрађује захтев и враћа кориснику одговор. У наставку је дат је део кода која приказује обраду захтева који се шаље контролеру када корисник кликне на дугме за пријаву:

```
        [HTTPPost]
        [AllowAnonymous]

      public async Task<IActionResult> PrijaviKorisnika(LoginViewModel model, string returnUrl)
        {
            if (ModelState.IsValid)
            {
                var result = await signInManager.PasswordSignInAsync(model.Username, model.Password,
                       false, false);

                if (result.Succeeded)
                {
                    if (!string.IsNullOrEmpty(returnUrl))
                    {
                        return LocalRedirect(returnUrl);
                    }
                    else
                    {
                        return RedirectToAction("Index", "Home", new { message = "Uspešno prijavljivanje na sistem! "});
                    }
                }
                else
                {
                    ViewBag.Message = "Neuspešno prijavljivanje na sistem!";
                }
            }
            return View(model);
        }
```

Механизми ауторизације имплементрани су кроз атрибуте:

```
   [Authorize(Roles ="Admin")]
    public class AdministrationController : Controller
    {
        private readonly RoleManager<IdentityRole> roleManager;
        private readonly IInstruktorService service;
        private readonly IObukaService serviceObuka;
        private readonly ObukaPasaContext context;
        private readonly UserManager<ApplicationUser> userManager;
        private readonly SignInManager<ApplicationUser> signInManager;
        private readonly ILogger<AdministrationController> logger;
        private readonly IConfiguration configuration

                                  ...
}
```

Слика 91 - *Неауторизован корисник*

Након обраде захтева, контролер који има референцу или више на њих на сервис класе, даље шаље захтев сервисима на обраду.

7.2.2 Пословна логика

Пословну логику апликације чине пословни процеси и пословне компоненте. У овом слоју се такође имплементирају и пословна правила добијена у процесу анализе. Пословна логика је описана структуром коју чине доменске класе и понашањем које се састоји из системских операција.

Пословна логика – доменске класе

На основу концептуалних класа праве се софтверске класе структуре система. Свака класа садржи приватна поља атрибута, модификаторе приступа за те атрибуте и конструкторе.

У наставку је дат код софтверске класе *Angazovanje*. Аналогно овом примеру, написан је и код за преостале софтверске класе.

```
public class Angazovanje
{
    public int PasId { get; set; }
    public Pas Pas { get; set; }
    public int ZadatakId { get; set; }
    public Zadatak Zadatak { get; set; }

    [DataType(DataType.Date)]
    [DisplayFormat(DataFormatString = "{0:dd.MM.yyyy.}", NullDisplayText = "Datum još uvek nije unet")]
    public DateTime? DatumUnosaOcene { get; set; }
    [Range(5,10)]
    [DisplayFormat(NullDisplayText = "Ocena još uvek nije uneta!")]
    public int? Ocena { get; set; }
}
```

Пословна логика – системске операције

Уговор UG2: UcitajListuRasa

Операција: UcitajListuRasa(List<string> rase): signal;
Веза са СК: **СК2**
Предуслови: /
Постуслови: /

```
public IEnumerable<Pas> GetAll()
        {
            return unitOfWork.PasRepository.GetAll();
        }
```

Уговор UG3: UcitajListuObuka

Операција: UcitajListuObuka(List<Obuka> obuke): signal;
Веза са СК: **СК2, СК3, СК4, СК5, СК6**
Предуслови: /
Постуслови: /

```
public IEnumerable<Obuka> GetAll()
        {
            return unitOfWork.ObukaRepository.GetAll();
        }
```

Уговор UG4: ZapamtiPsa

Операција: ZapamtiPsa(Pas): signal;
Веза са СК: **СК2**
Предуслови: Вредносна и структурна ограничења над објектом Пас морају бити задовољена.
Постуслови: Подаци о псу су запамћени.

```
public void Insert(Pas t)
        {
            if(isValid(t)==false || unitOfWork.PasRepository.GetAll().SingleOrDefault(p => p.BrojZdravstveneKnjizice ==
t.BrojZdravstveneKnjizice) != null) throw new Exception("Greška prilikom unosa podataka o psu!");
            unitOfWork.PasRepository.Insert(t);
            unitOfWork.Save();
        }

        private bool isValid(Pas t)
        {
            bool valid = true;
            if (t == null) return false;
            if (string.IsNullOrEmpty(t.BrojZdravstveneKnjizice) ||
                string.IsNullOrEmpty(t.Pol) || string.IsNullOrEmpty(t.Ime)
                || string.IsNullOrEmpty(t.Rasa) || t.Obuka == null) valid = false;
            if (t.Pol == "Muški" || t.Pol == "Ženski") valid = true;
            else valid = false;
            if(unitOfWork.ObukaRepository.FindById(t.ObukaId) == null) valid = false;
            if (!RaseStore.rase.Contains(t.Rasa)) valid = false;
            if (t.DatumRodjenja == null) valid = false;
          if(t.DatumRodjenja !=null && t.DatumRodjenja > DateTime.Now) valid = false;
            return valid;
        }
```

Уговор UG5: UcitajListuPasa

Операција: UcitajListuPasa(List<Pas> listaPasa): signal;
Веза са СК: **СК3, СК4, СК5, СК7**

Предуслови: /
Постуслови: /

```
public IEnumerable<Pas> GetAll()
        {
            return unitOfWork.PasRepository.GetAll();
 }
```

Уговор UG6: PronadjiPse

Операција: PronadjiPse(KriterijumPretrage): signal;
Веза са СК: **СК3, СК4, СК5**
Предуслови: /
Постуслови: /

```
  public List<Pas> GetAllByFilter(PsiFilter filters)
        {
           return unitOfWork.PasRepository.GetAllByFilter(filters);
        }
```

Уговор UG7: VratiPsa

Операција: VratiPsa(Pas): signal;
Веза са СК: **СК4, СК5**
Предуслови: /
Постуслови: /

```
 public Pas FindById(int? id)
        {
                return unitOfWork.PasRepository.FindById(id);
        }
```

Уговор UG8: ObrisiPsa

Операција: ObrisiPsa(Pas): signal;
Веза са СК: **СК4**
Предуслови: Структурна ограничења над објектом Пас морају бити задовољена.
Постуслови: Подаци о псу су обрисани.

```
      public void Delete(int pasIzBazeId)
        {
            Pas pas = unitOfWork.PasRepository.FindById(pasIzBazeId);
            if (pas == null) throw new Exception("Greška prilikom brisanja psa!");
            unitOfWork.PasRepository.Delete(pasIzBazeId);
            unitOfWork.Save();
        }
```

Уговор UG9: UcitajListuCinova

Операција: UcitajListuCinova(List<string> cinovi) : signal;
Веза са СК: **СК6**
Предуслови: /
Постуслови: /

```
  public IEnumerable<Instruktor> GetAllCinovi()
        {
            return unitOfWork.InstruktorRepository.GetAllCinovi();
```

```
        }
```

Уговор UG11: ZapamtiZadatak

Операција: UnosNovogZadatka(Zadatak): signal;
Веза са СК: **СК7**
Предуслови: Вредносна и структурна ограничења над објектом Задатак морају бити задовољена.
Постуслови: Подаци о задатку су запамћени.

```
public void Insert(Zadatak t)
        {
            if (isValid(t) == false ||
            t.Status != Enumerations.Status.Kreiran.ToString()
            || postojiUBazi(t)==true
            ) throw new ArgumentOutOfRangeException("Nevalidan unos!");

            unitOfWork.ZadatakRepository.Insert(t);
            unitOfWork.Save();
        }
        private bool postojiUBazi(Zadatak t)
        {
            bool postoji = false;
            if (t != null)
            {
               foreach(Zadatak z in GetAll().ToList())
                {
                    if (z.Datum.Equals(t.Datum) && z.NazivZadatka == t.NazivZadatka
                        && z.Teren == t.Teren) postoji = true;
                }
            }
            return postoji;
        }
        private bool isValid(Zadatak t)
        {
            bool valid = true;
            if (t == null) return false;
            if (string.IsNullOrEmpty(t.NazivZadatka) || string.IsNullOrEmpty(t.Teren)
            || string.IsNullOrEmpty(t.Status) || t.Datum == null || t.Angazovanja == null) valid = false;
            if (t.Status == Enumerations.Status.Kreiran.ToString() && t.Datum <= DateTime.Now) valid = false;
            if (t.Angazovanja != null && t.Angazovanja.Count() == 0) valid = false;
            return valid;
        }
```

Уговор UG12: UcitajListuZadataka

Операција: UcitajListuZadataka(List<Zadatak> listaZadataka): signal;
Веза са СК: **СК8**
Предуслови: /
Постуслови: /

```
  public IEnumerable<Zadatak> GetAll()
        {
            return unitOfWork.ZadatakRepository.GetAll();
        }
```

Уговор UG13: PronadjiZadatake

Операција: PronadjiZadatake(KriterijumPretrage): signal;
Веза са СК: **СК8**
Предуслови: /
Постуслови: /

```
public List<Zadatak> GetAllByFilter(ZadatakFilter filters)
      {
          return unitOfWork.ZadatakRepository.GetAllByFilter(filters);
      }
```

Уговор UG14: VratiAngazovanja

Операција: VratiAngazovanja(ZadatakId): signal;
Веза са СК: **СК8**
Предуслови: /
Постуслови: /

```
public List<Angazovanje> GetAllByZadatakId(int id)
      {
          return unitOfWork.AngazovanjeRepository.GetAllByZadatakId(id);
      }
```

Уговор UG15: SacuvajAngazovanja

Операција: SacuvajAngazovanja(List<Angazovanja> angazovanja): signal;
Веза са СК: **СК8**
Предуслови: Вредносна и структурна ограничења над објектом Ангажовање морају бити задовољена.
Постуслови: Ангажовања су замапћена.

```
public void Update(Angazovanje angazovanje)
      {
          if (isValid(angazovanje) == false) throw new ArgumentOutOfRangeException();
          unitOfWork.AngazovanjeRepository.Update(angazovanje);
          unitOfWork.Save();
      }
```

Уговор UG16: IzmeniPsa

Операција: IzmeniPsa(Pas): signal;
Веза са СК: **СК5**
Предуслови: Вредносна и структурна ограничења над објектом Пас морају бити задовољена.
Постуслови: Измењени подаци о псу су запамћени.

```
public void Update(Pas pas)
      {
          if (isValid(pas) == false) throw new ArgumentOutOfRangeException("Greška prilikom čuvanja podataka o psu!");
          unitOfWork.PasRepository.Update(pas);
          unitOfWork.Save();
      }
```

Имплементација сервиса

Сервис класе чине слој пословне логике и за сваку доменску класу имплементиран је сервис и одговарајући интерфејс у коме се налазе системске операције. Сервис користи UnitOfWork класу у којој се налазе референце ка Repository класама.

Основни интерфејс класе која представља сервис је генерички. Конкретни интерфејси сервис класа који су имплементирани за сваки од ентитета наслеђују базни интерфејс и допуњују га својим методама. На крају, конкретна имплементациона сервис класа имплементира конкретан интерфејс сервиса.

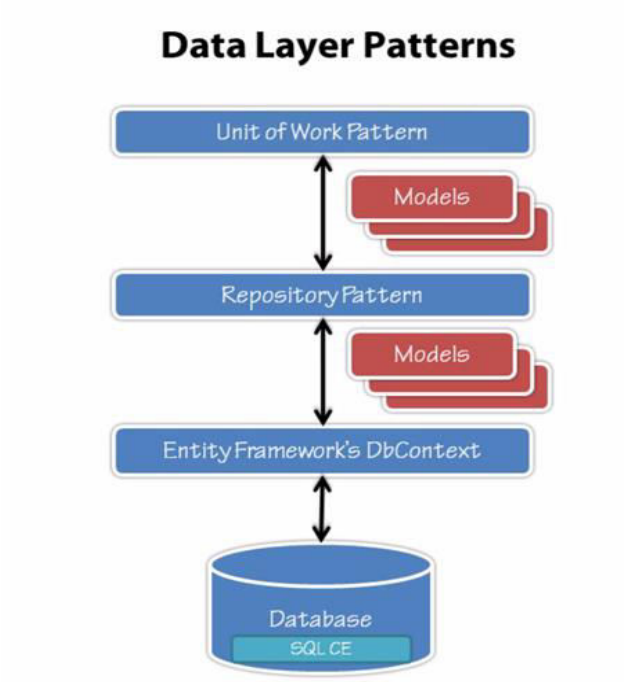
У наставку је дат код класе *ZadatakService* са одговарајућим интерфејсима као и *IService* интерфејс класа. Аналогно, имплементиране су сервис класе са одговарајућим интерфејсима и за остале доменске класе.

```
public class ZadatakService : IZadatakService
{
    private readonly IUnitOfWork unitOfWork;

    public ZadatakService(IUnitOfWork unitOfWork)
    {
        this.unitOfWork = unitOfWork;
    }
    public void Delete(int zadatakIzBazeId)
    {
        Zadatak zadatak = unitOfWork.ZadatakRepository.FindById(zadatakIzBazeId);
        if (zadatak == null) throw new Exception();
        unitOfWork.ZadatakRepository.Delete(zadatak.Id);
        unitOfWork.Save();
    }
    public Zadatak FindById(int? id)
    {
        return unitOfWork.ZadatakRepository.FindById(id);
    }
    public IEnumerable<Zadatak> GetAll()
    {
        return unitOfWork.ZadatakRepository.GetAll();
    }
    public void Insert(Zadatak t)
    {
        if (isValid(t) == false ||
        t.Status != Enumerations.Status.Kreiran.ToString()
        || postojiUBazi(t)==true
        ) throw new ArgumentOutOfRangeException("Nevalidan unos!");

        unitOfWork.ZadatakRepository.Insert(t);
        unitOfWork.Save();
    }
    private bool postojiUBazi(Zadatak t)
    {
        bool postoji = false;

        if (t != null)
        {
           foreach(Zadatak z in GetAll().ToList())
            {
                if (z.Datum.Equals(t.Datum) && z.NazivZadatka == t.NazivZadatka
                    && z.Teren == t.Teren) postoji = true;
            }
        }
        return postoji;
    }
    private bool isValid(Zadatak t)
    {
        bool valid = true;
        if (t == null) return false;
        if (string.IsNullOrEmpty(t.NazivZadatka) || string.IsNullOrEmpty(t.Teren)
        || string.IsNullOrEmpty(t.Status) || t.Datum == null || t.Angazovanja == null) valid = false;
        if (t.Status == Enumerations.Status.Kreiran.ToString() && t.Datum <= DateTime.Now) valid = false;
        if (t.Angazovanja != null && t.Angazovanja.Count() == 0) valid = false;

        return valid;
    }
    public void Update(Zadatak zadatak)
    {
        if(isValid(zadatak) == false) throw new ArgumentOutOfRangeException("Nevalidan unos!");
        unitOfWork.ZadatakRepository.Update(zadatak);
        unitOfWork.Save();
    }
```

```
        public List<int> VratiBrojAngazovanjaPoZadatku()
        {
            return unitOfWork.ZadatakRepository.VratiBrojAngazovanjaPoZadatku();
        }

        public List<int> VratiBrojPozitivihOcenaPoZadatku()
        {
            return unitOfWork.ZadatakRepository.VratiBrojPozitivihOcenaPoZadatku();
        }

        public List<string> VratiListuNazivaZadataka()
        {
            return unitOfWork.ZadatakRepository.VratiListuNazivaZadataka();
        }

        public List<Zadatak> GetAllByFilter(ZadatakFilter filters)
        {
            return unitOfWork.ZadatakRepository.GetAllByFilter(filters);
        }
    }


public interface IZadatakService:IService<Zadatak>
    {
        public void Update(Zadatak zadatak);
        public void Delete(int zadatakIzBazeId);
        public Zadatak FindById(int? id);
        public List<string> VratiListuNazivaZadataka();
        public List<int> VratiBrojAngazovanjaPoZadatku();
        public List<int> VratiBrojPozitivihOcenaPoZadatku();
        public void Insert(Zadatak zadatak);
        public List<Zadatak> GetAllByFilter(ZadatakFilter filters);}
   public interface IService<T> where T : class
    {
        IEnumerable<T> GetAll();
    }
```

7.2.3 Слој приступа подацима

Unit Of Work патерн и Repository патерн



Слика 92 - *Коришћени патерни*. (n.d.). [Digital image]. https://grekai.files.wordpress.com/2013/03/image_thumb15.png?w=479&h=544

Главна намена *Unit Of Work* и *Repository* патерна је стварање слоја апстракције између пословне логике и слоја приступа подацима. Имплементација ових образаца изолује апликацију од промена у складишту података и олакшава аутоматско *Unit* тестирање [12].

Unit Of Work патерн

У слоју за приступ бази није пожељно обраћати се бази за сваку промену у подацима апликације и одговарајућем објектом моделу јер би такав приступ допринео стварању веома великог саобраћаја ка бази података и око сваког позива базе као што је отварање конекције, припрема пакета и слично. *UnitOfWork* као јединица рада представља логичку трансакцију која групише већи број позива ка бази. Када *UnitOfWork* јединица рада заврши са својим радом, може се позвати метода *Save()* којом се потврђују промене у бази. Интерфејс *IUnitOfWork* имплементира интерфејс *IDisposable* који пружа могућност ослобађања ресурса чим се извршење заврши [12].

На овај начин повећава се ниво апстракције и одваја пословна логика од директног приступа подацима. *UnitOfWork* класа садржи референце ка *Repository* класама. У наставку је дат код којим се приказује имплементација овог патерна.

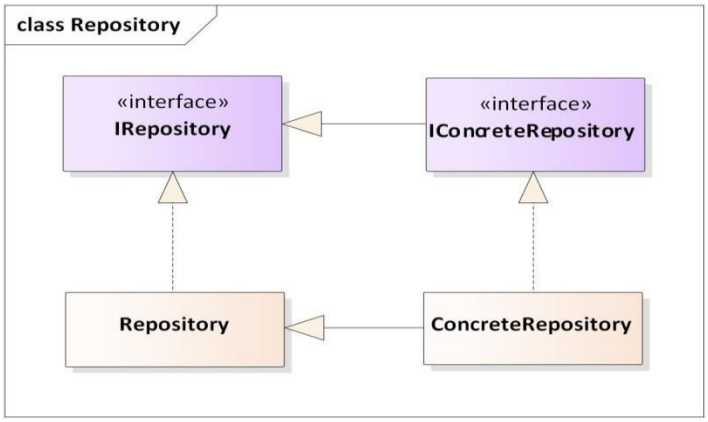
```
public class UnitOfWork : IUnitOfWork
 {
     private readonly ObukaPasaContext context;
     public UnitOfWork(ObukaPasaContext context)
     {
         this.context = context;
         InstruktorRepository = new InstruktorRepository(context);
         ObukaRepository = new ObukaRepository(context);
         PasRepository = new PasRepository(context);
         ZadatakRepository = new ZadatakRepository(context);
         AngazovanjeRepository = new AngazovanjeRepository(context);
     }
     public IInstruktorRepository InstruktorRepository { get; set; }
     public IObukaRepository ObukaRepository { get; set; }
```

```
        public IPasRepository PasRepository { get; set; }
        public IZadatakRepository ZadatakRepository { get; set; }
        public IAngazovanjeRepository AngazovanjeRepository { get; set; }
        public void Save()
        { context.SaveChanges();}
        public void Dispose()
        {
            context.Dispose();
        }}


public  interface IUnitOfWork:IDisposable
    {
        IInstruktorRepository InstruktorRepository { get; set; }
        IObukaRepository ObukaRepository { get; set; }
        IZadatakRepository ZadatakRepository { get; set; }
        IPasRepository PasRepository { get; set; }
        IAngazovanjeRepository AngazovanjeRepository  { get; set; }
        void Save();
    }
```

Repository патерн

За сваки тип у објектом моделу креирана је мапирајућа класа која имплементира интерфејс који приказује све операције над базом података које се могу извршити над типом. Овакве класе зову се репозиторијуми. Репозиторијум посредује између пословне логике и слоја приступа подацима, енкапсулира скуп објеката сачуваних на неком медијуму и дозвољене операције над њима, на објектно-оријентисани начин, такође одвајајући пословну логику од директног приступа подацима. На пример, пословна логика захтева методу *GetAll()* и очекује да добије све доступне објекте. Са друге стране, за више слојеве апликације није неопходно да имају информацију о томе да ли су ови објекти учитани из базе података или веб сервиса [12].

Основни интерфејс репозиторијума је генерички, чиме је омогућена једноставна и лака употреба у осталим деловима апликације. Конкретни интерфејси репозиторијума наслеђују базни интерфејс и допуњују га својим методама. Пракса је да се креира генеричка класа базна класа која имплементира заједничку функционалност за све конкретне репозиторијуме. На крају, конкретна имплементациона класа наслеђује базну класу имплементације и имплементира конкретни интерфејс репозиторијума. Следећи дијаграм приказује описану интеракцију која настаје између репозиторијума:



Слика 93 - *Интеракција између репозиторијума*

У наставку је дат код путем кога је имплементиран интерфејс и класа базног репозиторијума:

```
   public interface IRepository<T> where T : class
        {
            IEnumerable<T> GetAll();
        }

   public class BaseRepository<T> : IRepository<T> where T : class
    {
        private readonly ObukaPasaContext context;
        public BaseRepository(ObukaPasaContext context)
        {
            this.context = context;
        }
        public virtual IEnumerable<T> GetAll()
        {
            return context.Set<T>();
        }
    }
```

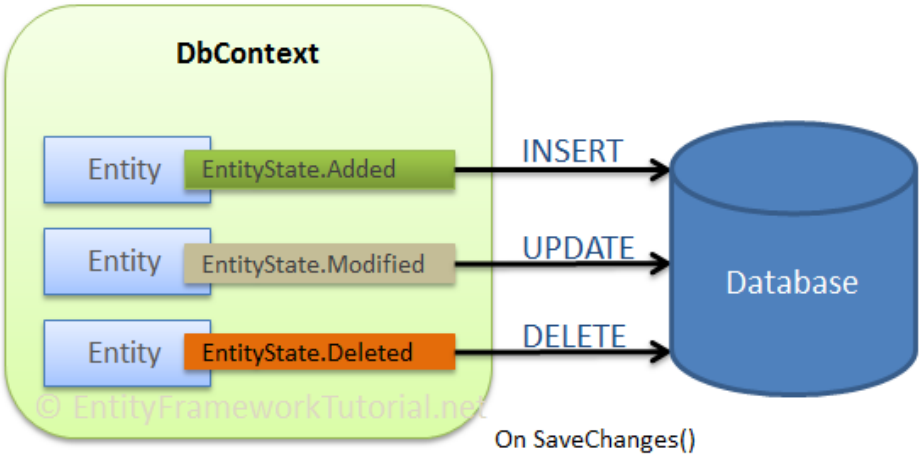
Интерфејс пружа стандардне методе за рад са ентитема користећи предности које доноси LINQ (Language Integrated Query). Главни задатак базне класе репозиторијума је да имплементира заједничку функционалност за све класе репозиторијума. За сваки ентитет имплементирана је класа репозиторијума. У наставку дат је код путем кога су имплементиране репозиторијум класе и одговарајући интерфејс за доменску класу *Angazovanje*. Аналогно су имплементиране и за све остале доменске класе. Класа *AngazovanjeRepository* имплементира интерфејс *IAngazovanjeRepository*. Поред имплементираних метода, конструктор класе прима контекст класу као параметар. Инстанца ове класе омогућава основне операције над жељеним ентитетом.

```
  public interface IAngazovanjeRepository:IRepository<Angazovanje>
    {
        public void Save();
        public void Update(Angazovanje a);
        public Angazovanje GetById(int PasId, int ZadatakId);
        public void Delete(Angazovanje angazovanje);
        public  void Insert(Angazovanje a);
        public List<Angazovanje> GetAllByZadatakId(int id);
    }

 public class AngazovanjeRepository : IAngazovanjeRepository

    {
        private readonly ObukaPasaContext context;
        public AngazovanjeRepository(ObukaPasaContext context)
        {
            this.context = context;
        }
        public void Delete(Angazovanje angazovanje)
        {
                context.Entry(angazovanje).State = EntityState.Deleted;
        }
        public IEnumerable<Angazovanje> GetAll()
        {
                return context.Angazovanja.AsNoTracking().Include(a => a.Zadatak).Include(a => a.Pas).ThenInclude(p =>
p.Obuka);
        }
        public List<Angazovanje> GetAllByZadatakId(int id)
        {
         return GetAll().Where(a => a.ZadatakId == id).ToList();
        }
        public Angazovanje GetById(int PasId, int ZadatakId)
        {

                return    context.Angazovanja.Include(a    =>    a.Zadatak).Include(a    =>    a.Pas).ThenInclude(p    =>
p.Obuka).ToList().Find(a => a.PasId == PasId && a.ZadatakId == ZadatakId);
        }
        public void Insert(Angazovanje t)
        {
                context.Add(t);
```

```
        }
        public void Save()
        {
                context.SaveChanges();
        }
        public void Update(Angazovanje a)
        {
        context.Entry(a).State = EntityState.Modified;
        }
}
```

DbContext u Entity Framework Core

Entity Framework Core омогућава претраживање, додавање, ажурирање и брисање података. Овај оквир мапира ентитете и везе који су дефинирани у нашем моделу директно у базу података.



Слика 94 - *Интеракција између DbContext класе и базе података* . (n.d.). [Digital image]. https://www.entityframeworktutorial.net/images/efcore/save-data-in-connected-scenario.png

Примарна класа која је одговорна за интеракцију са подацима као објектима је *System.Data.Entity.DbContext* класа. *DbContext*, односно контекст класа је најважнија класа током рада са *Entity Framework Core* - ом. Представља сесију са основном базом података помоћу које се изводе *CRUD* операције. Ова класа се користи за чување података и релизацију упита везаних за базу података. Такође се користи за конфигурисање класа домена, мапирања у вези са базом података, промене поставки праћења, кеширање, трансакције и слично. Имплементација ове класе остварује се путем дефинисања класе која је изведена из *DbContext* класе и излаже *DbSet* својства која представљају колекције одређених ентитета у контексту. Већ споменут *ASP.NET Core Identity* механизам утиче и на *DbContext* класу. Како би се искористиле све функционалности које овај механизам пружа, потребно да је класа која представља *DbContext* класу наследи класу *IdentityDbContext* уместо класе *DbContext*. Ово је потребно јер *IdentityDbContext* пружа сва својства потребна за управљање табелама идентитета на страни *SQL* сервера. У бази података могу се видети све табеле које *ASP.NET Core Identity* генерише. Уколико се погледа ланац хијерархије класе *IdentityDbContext*, уочава се да је она изведена из класе *DbContext* [8]. Следећи код представља пример који показује креирање ове класе за потребе имплементације датог софтверског система:

```
  public class ObukaPasaContext : IdentityDbContext<ApplicationUser>
    {
        public ObukaPasaContext(DbContextOptions<ObukaPasaContext> options) : base(options){}
        public DbSet<Obuka> Obuke { get; set; }
        public DbSet<Instruktor> Instruktori { get; set; }
        public DbSet<Pas> Psi { get; set; }
        public DbSet<Angazovanje> Angazovanja { get; set; }
        public DbSet<Zadatak> Zadaci { get; set; }
        public DbSet<Admin> Admins { get; set; }
        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            base.OnModelCreating(modelBuilder);
            //Obuka
            modelBuilder.Entity<Obuka>().ToTable("Obuka");
            modelBuilder.Entity<Obuka>().HasKey(o => o.Id);
            modelBuilder.Entity<Obuka>().Property(o => o.Id).IsRequired();
            //Instruktor
            modelBuilder.Entity<Instruktor>().HasOne(e  =>  e.Obuka).WithMany(c  =>  c.Instruktori).HasForeignKey(c  =>
c.ObukaId);
            //Pas
            modelBuilder.Entity<Pas>().ToTable("Pas");
            modelBuilder.Entity<Pas>().HasKey(p => p.Id);
            modelBuilder.Entity<Pas>().Property(p => p.BrojZdravstveneKnjizice).IsRequired();
            modelBuilder.Entity<Pas>().HasOne(p => p.Obuka).WithMany(o => o.Psi).HasForeignKey(p=>p.ObukaId);
            //Angazovanje
            modelBuilder.Entity<Angazovanje>().ToTable("Angazovanje");
            modelBuilder.Entity<Angazovanje>().HasKey(a => new { a.PasId, a.ZadatakId });
            modelBuilder.Entity<Angazovanje>().HasOne(a  =>  a.Pas).WithMany(p  =>  p.Angazovanja).HasForeignKey(a  =>
a.PasId);
            modelBuilder.Entity<Angazovanje>().HasOne(a  =>  a.Zadatak).WithMany(z  =>  z.Angazovanja).HasForeignKey(a  =>
a.ZadatakId);
            //Zadatak
            modelBuilder.Entity<Zadatak>().ToTable("Zadatak");
            modelBuilder.Entity<Zadatak>().HasKey(z => z.Id);
         }}
```

DbContextOptions који представља контенјер који укључује сву конфигурацију за рад са базом нам омогућава да поставимо конекциони стринг путем кога се апликација повезује са базом подтака. *DbContextOptions* се примењује на *ObukaPasaContext* класу и аутоматски ће позвати конекциони стринг, креирати базу података и омогућити нам коришћење миграција.

У фолдеру appsettings.json додаје се конекциони стринг:

```
{
  "ConnectionStrings": {

"DefaultConnection":"Server=(localdb)\\MSSQLLocalDB;Database=DatabaseVojniPsiDB;IntegratedSecurity=True;Trusted_Connecti
on=True;MultipleActiveResultSets=true"
  },
                                    ...
}
```

У стартап класи је такође потребно извршити следећа подешавања:

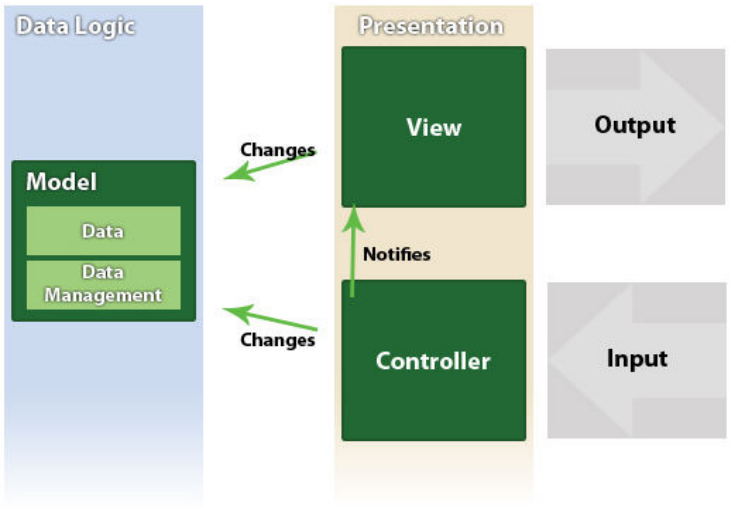
```
public void ConfigureServices(IServiceCollection services)
        {
            services.AddDbContext<ObukaPasaContext>(options=>
options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));

                                    ...

}
```

Као и што се може уочити, помоћу *DbSet*-а заправо дефинишу се табеле у бази. Креира се *DbSet* за сваку ентитеску класу која је претходно направљена. Метода *OnModelCreating* обезбеђује дефинисање имена

генерисаних табела, омогућава да се поставе примарни и спољни кључеви, дефинишу везе између табела и постављају подешавања за пропертије.

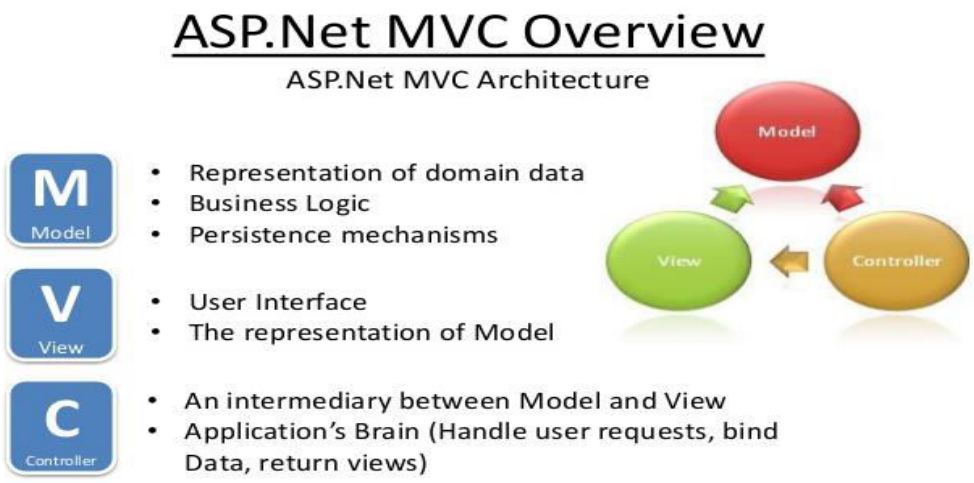
7.3 Имплементација презентационог слоја

Без обзира на квалитет кода који се налази у средишњем слоју, он се не може користити уколико не постоји начин да се он представи корисницима. Кориснички интерфејс *(UI - User Interface*), пословна логика и код за приступ подацима су подједнако неопходни систему било ког нивоа комплексности. На крају резултат мора бити такав да сваки слој у потпуности испуњава очекивања. Могућност да се постојећи презентациони слој који се састоји од две главне компоненте: корисничког интерфејса и презентационе логике, уклони са постојеће апликације и да се замени новим представља главни захтев који је пожељно испунити. Презентациони слој представља интерфејс између корисника и система. Са једне стране, он пружа алате који су корисницима апликације потребни да би је користили. Са друге стране, он садржи логику која је неопходна како би се извршила координација акција система и корисника са циљем приказивања и уношења података. Најтежи део презентационе логике јесте креирање апстракције погледа, за сваки поглед у презентационом слоју. Апстракција погледа дефинише који подаци су укључени у улазе и излазе погледа. Циљ презентационе логике јесте да се осигура да подаци коректно улазе и излазе из погледа. На следећој слици приказан је ток комуникације у презентационом слоју [1].



Слика 95 - *Комуникација у презентационом слоју*. (n.d.). [Digital image]. https://www.codeproject.com/KB/architecture/605786/MVC.jpg

У *MVC* патерну, одговарајући контролер бира *View* који ће се приказати кориснику и пружа му све податке модела који су му потребни. *Views*, односно погледи су одговорни за представљање садржаја путем корисничког интерфејса. У погледима би требало да постоји минимална логика која се односи на представљање садржаја.



Слика 96 - ***MVC архитектура.*** **(n.d.-b). [Digital image]. https://image.slidesharecdn.com/asp-150219223228-conversion-gate01/95/aspnet-mvc-presentation-by-nitin-sawant-8-638.jpg?cb=1424385538**

Као што је приказано на претходној слици, поглед чини кориснички интерфејс и репрезентацију модела, а контролер обрађује инпуте и управља погледима који ће се приказати кориснику.

Дат је пример *MVC* апликације у наставку текста који приказује ток захтева за приказ статистике ангажовања изабраног пса кроз одговарајући модел, поглед и контролер. Приказан је модел – класа *Pas* која садржи одговарајуће пропертије за јединствени идентификатор и друге релевантне атрибуте.

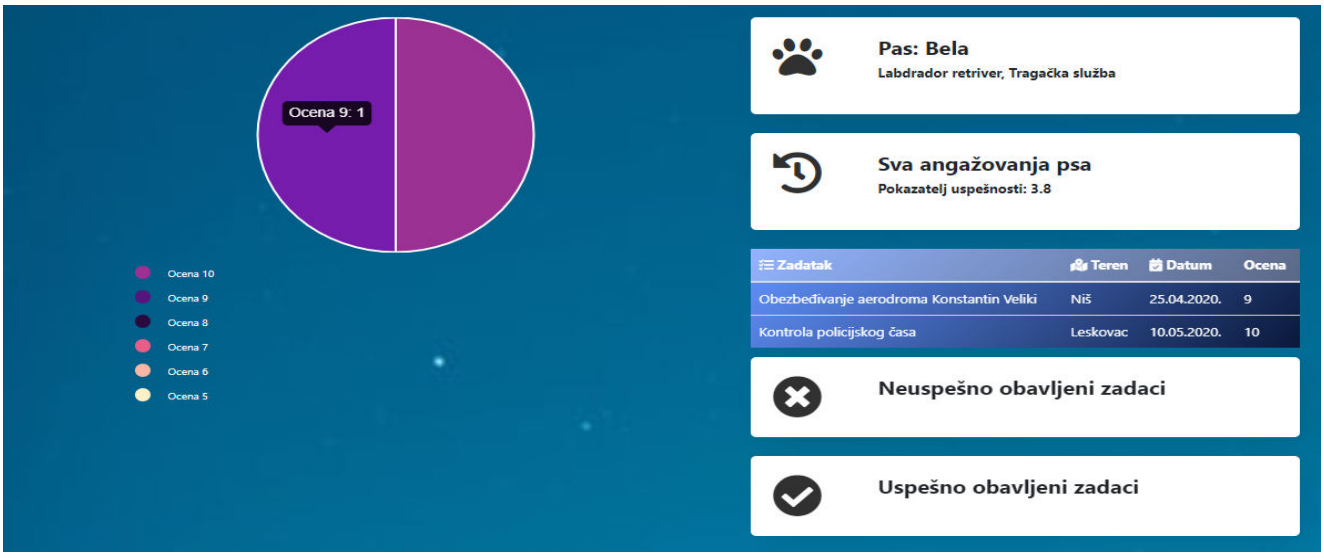
```
   public class Pas
    {
        public int Id { get; set; }

        [DisplayName("Zdravstvena knjizica")]
        [Required(ErrorMessage = "Ovo polje je obavezno!")]
        public string BrojZdravstveneKnjizice { get; set; }

        [Required(ErrorMessage = "Ovo polje je obavezno!")]
        public string Ime { get; set; }

        [DataType(DataType.Date)]
        [DisplayFormat(DataFormatString = "{0:dd.MM.yyyy.}")]
        [DisplayName("Datum rodjenja")]
        [Required(ErrorMessage = "Ovo polje je obavezno!")]
        public DateTime DatumRodjenja { get; set; }
        public string Rasa { get; set; }
        public string Pol { get; set; }
        public int ObukaId { get; set; }
        public Obuka Obuka { get; set; }
        public ICollection<Angazovanje> Angazovanja { get; set; }
    }
}
```

Затим је креирана контролер класа која је изведена из класе *Controller* и садржи јавне методе. Приказана је метода *Statistika* којој се приступа кликом на одељак *Statistika* жељеног пса у оквиру листе паса на одговарајућој страници страници. У оквиру акционе методе позива се сервис класа уз помоћ које се из базе добијају подаци о жељеном псу. Затим се постављају одговарајући подаци који се прослеђују погледу на основу којих се позива екстерни сервис помоћу кога се креира графикон статистике ангажовања.

```
public class PasController : Controller
```

```
    {
                            . . .
  [Authorize(Roles = "Korisnik,Admin")]
        public IActionResult Statistika(int? id)
        {
            try
            {
                    if (id == null)  throw new Exception("Pas čije podatke zahtevate ne postoji!");
                    Pas pasIzBaze = service.FindById(id);
                    if (pasIzBaze == null) throw new Exception("Pas čije podatke zahtevate ne postoji!");
                if (pasIzBaze.Angazovanja.Count() != 0)
                {
                    ViewBag.Desetke = service.VratiBrojOcena(pasIzBaze, 10);
                    ViewBag.Devetke = service.VratiBrojOcena(pasIzBaze, 9);
                    ViewBag.Osmice = service.VratiBrojOcena(pasIzBaze, 8);
                    ViewBag.Sedmice = service.VratiBrojOcena(pasIzBaze, 7);
                    ViewBag.Sestice = service.VratiBrojOcena(pasIzBaze, 6);
                    ViewBag.Petice = service.VratiBrojOcena(pasIzBaze, 5);
                    ViewBag.Uspeh = service.UspesnostPsaNaObavljenimZadacima(pasIzBaze);
                    ViewBag.UkupnoAng = pasIzBaze.Angazovanja.Count();
                }else
                {
                    ViewBag.UkupnoAng = 0;
                    ViewBag.Uspeh = 0;
                }
                ViewBag.PasId = pasIzBaze.Id;
                return View(pasIzBaze);
            }
                                                                    . . .
}
```

На крају контролер генерише поглед који се приказује кориснику. То је уједно последњи корак – додавање погледа који ће приказати податке. Поглед укључује класу *Pas* као свој модел, чиме се говори да је поглед намењен за приказивање података из датог дела модела. На наредној слици се може видети шта се приказује кориснику као одговор на његов захтев.



Слика 97 - *Крајњи одговор на захтев корисника*

8. Фаза тестирања

Приликом израде софтверског система тестирање представља покушај да се пронађу грешке у софтверу који је направљен. Софтвер је имплементиран према корисничким захтевима којима се решава неки реални проблем или се креира нека корисна функционалност која представља нешто што је потребно крајњим корисницима. Када се имплементира, софтвер може у већој или мањој мери да одговара оригиналним захтевима према којима је и направљен. Свако понашање софтвера које се не слаже са оригиналним захтевима представља грешку коју је потребно идентификовати и отклонити. Тестирање представља проверу да ли је одређени софтвер у потпуности имплементиран према оригиналним корисничким захтевима.

8.1 Unit тестови

Имплементиран систем тестиран је помоћу *unit* тестова. *Unit* тестови су скупови кода који тестирају појединачне модуле система и дизајнирани су тако да утврде да ли су модули који се тестирају прикладни за употребу у производним системима. То значи да ови тестови тестирају мале делове система, независно од понашања дефинисаног у другим деловима. За потребе тестирања овог система коришћени су *XUnit* тестови. *XUnit* је оквир за тестирање који пружа једноставан начин за обележавање, покретање и тестирање [13].

Израз "*Arrange, Act, Assert*" указује из чега се састоји један тест [13]:

- *Arrange* значи постављање потребног тестног окружења и зависности. То може значити стварање скупа података добро познатих тестова или стварање испитне зависности коју тренутно не тестирамо. Укратко, то значи "створити оно што тест треба да спроведе."
- *Act* значи стварно извршити код који се тестира, с обзиром на подешавања у претходном кораку.
- *Assert* значи проверити стварне резултате и очекиване резултате и потврдити да су они онакви какве и очекујемо.

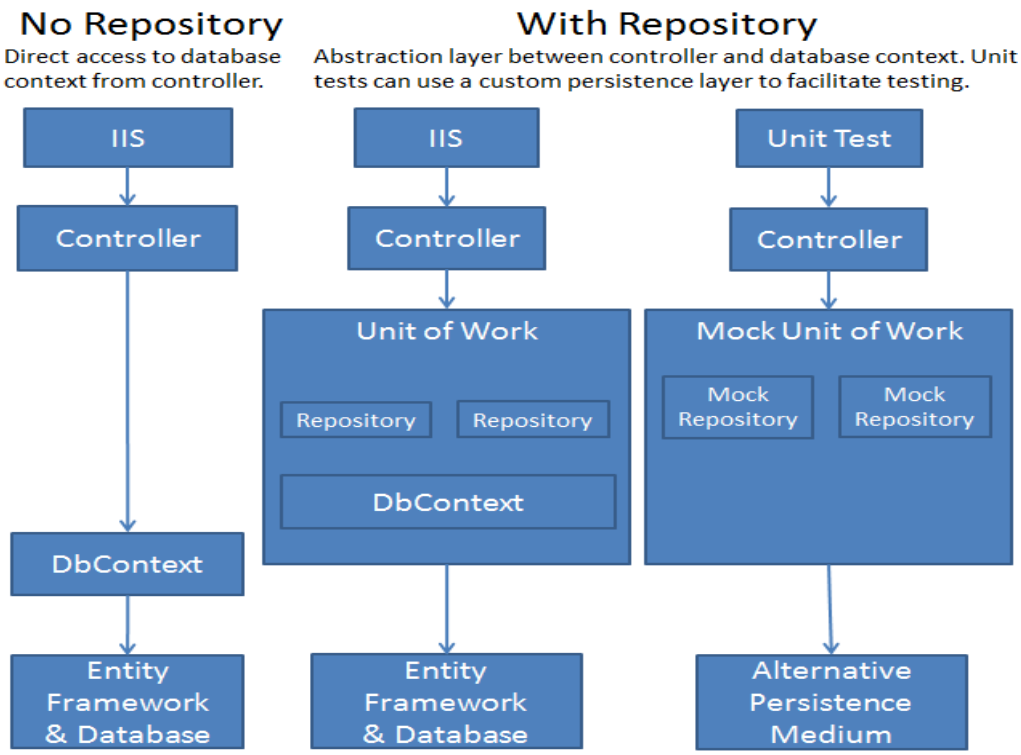
Unit тестирање представља најосновнију врсту тестирања одређеног софтверског система и осмишљен је да детектује промене у примени датог система. *XUnit* је један од многих пакета за тестирање и пружа једноставан начин да се напишу тестови помоћу атрибута *[Fact]* [13]. У наставку је дат пример теста путем кога се тестира системска операција која додаје нови задатак у базу података:

```
[Fact]
    public void TestZadatakServiceInsertZadatak()
    {
        //Arrange
        var newZadatak = new Zadatak()
        {
            Id = 3,
            Datum = new DateTime(2020, 04, 29),
            Status = Enumerations.Status.Kreiran.ToString(),
            Teren = "Niš",
            NazivZadatka = "Kontrola policijskog časa"
        };
        newZadatak.Angazovanja = new List<Angazovanje>()
        { new Angazovanje()
        {
                DatumUnosaOcene = null,
                        Ocena = null,
                        ZadatakId = 1,
                        Pas = unitOfWork.Object.PasRepository.FindById(1),
                        PasId = 1,
                        Zadatak = newZadatak
        }, new Angazovanje()
```

```
        {
                DatumUnosaOcene = null,
                        Ocena = null,
                        ZadatakId = 1,
                        Pas =  unitOfWork.Object.PasRepository.FindById(2),
                        PasId = 2,
                        Zadatak = newZadatak
        },
         new Angazovanje()
        {
                DatumUnosaOcene = null,
                        Ocena = null,
                        ZadatakId = 1,
                        Pas =  unitOfWork.Object.PasRepository.FindById(3),
                        PasId = 3,
                        Zadatak = newZadatak
        }
        };
        var service = new ZadatakService(unitOfWork.Object);
        //Act
        service.Insert(newZadatak);
        var result = service.GetAll();
        var zadaci = unitOfWork.Object.ZadatakRepository.GetAll();
        Zadatak zadatak = service.FindById(newZadatak.Id);
        //Assert
        Assert.Equal(newZadatak.Id, zadatak.Id);
        Assert.Equal(zadaci.Count(), result.Count());
        unitOfWork.Verify(x => x.Save(), Times.Once);
        unitOfWork.Verify(x => x.ZadatakRepository.Insert(It.Is<Zadatak>(p=>p.Id==3)),Times.Once);
    }
```

8.2 Moq Framework

Један од циљева које тест треба да има за тестирање апликације попут ове је тестирање сваког слоја независно од осталих. То значи да, на пример, да бисмо тестирали пословну логику у којој се налазе сервис класе , морамо да покренемо тестове који не зависе од функционалности у доњем слоју, односно слоју где се налазе репозиторијум класе и класа јединице рада, од којих сервисне класе зависе. Да би се то постигло потребно је *mock* – овати класу јединице рада, што значи да смо поставили лажну класу која враћа познату вредност за одређене позиве. *Moq* је најпопуларнија библиотека за извођење таквих активности у *ASP .NET Core* – u. Стога ћемо ту библиотеку користити у овом раду. Овде долази до изражаја већ споменуто коришћење *Unit of work* и *Repostiory* патерна. Следећа слика приказује које су предности коришћена ових патерна у погледу тестирања помоћу *Moq* - а [13].



Слика 98 - ***Предности коришћења Моq оквира за тестирање.*** **(n.d.). [Digital image]. https://www.asp.net/media/2578149/Windows-Live-Writer_8c4963ba1fa3_CE3B_Repository_pattern_diagram_1df790d3-bdf2-4c11-9098-946ddd9cd884.png**

Репозиторијуми омогућавају једноставније тестирање кода помоћу unit тестова на које не утиче стање нивоа података. У наставку рада приказан је део кода имплементиране класе *Mocks.cs* у којој су имплементиране *mock*-овани репозиторијуми *Angazovanje* и *Zadatak* и класа *UnitOfWork.cs:*

```
public class Mocks
    {

public static Mock<IZadatakRepository> GetMockZadatakRepository()
        {
            #region Initialize
            var zadaci = new List<Zadatak>();
            Zadatak z1 = new Zadatak()
            {
                Id = 1,
                Datum = new DateTime(2020, 04, 22),
                Status = "Završen",
                Teren = "Beograd",
                NazivZadatka = "Kontrola policijskog časa"
            };
            z1.Angazovanja = new List<Angazovanje>()
                                                    …
            zadaci.Add(z1);

            #endregion
            var mockZadatakRepository = new Mock<IZadatakRepository>();
            mockZadatakRepository.Setup(repo => repo.GetAll()).Returns(zadaci);
            mockZadatakRepository.Setup(repo => repo.FindById(It.IsAny<int>())).Returns((int? i) => {
                if (i != null)
                    return zadaci.SingleOrDefault(x => x.Id == i);
                else return null;
            });
            mockZadatakRepository.Setup(i => i.Insert(It.IsAny<Zadatak>())).Callback((Zadatak item) =>
            {
```

```
            zadaci.Add(item);
            foreach (Angazovanje a in item.Angazovanja)
            {
                GetMockAngazovanjeRepository().Object.Insert(a);
            }
        });
        mockZadatakRepository.Setup(m => m.Update(It.IsAny<Zadatak>())).Callback((Zadatak target) =>
        {
            var original = zadaci.FirstOrDefault(
                q => q.Id == target.Id);
            original.NazivZadatka = target.NazivZadatka;
            original.Datum = target.Datum;
            original.Status = target.Status;
            original.Teren = target.Teren;
            original.Angazovanja = target.Angazovanja;
        }).Verifiable();
        mockZadatakRepository.Setup(m => m.Delete(It.IsAny<int>())).Callback((int? i) =>
        {
            var original = zadaci.FirstOrDefault(
                q => q.Id == i);
            if (original != null)
            {
               zadaci.Remove(original);
                foreach(Angazovanje a in GetMockAngazovanjeRepository().Object.GetAll())
                {
                    if(a.ZadatakId == original.Id)
                    {
                        GetMockAngazovanjeRepository().Object.Delete(a);
                    }
                }

        }).Verifiable();
        return mockZadatakRepository;
    }

                                            …

    }public static Mock<IUnitOfWork> GetMockUnitOfWork()

    {
        var unitOfWork = new Mock<IUnitOfWork>();
        unitOfWork.Setup(x => x.ObukaRepository).Returns(GetMockObukaRepository().Object);
        unitOfWork.Setup(x => x.PasRepository).Returns(GetMockPasRepository().Object);
        unitOfWork.Setup(x => x.ZadatakRepository).Returns(GetMockZadatakRepository().Object);
        unitOfWork.Setup(x => x.AngazovanjeRepository).Returns(GetMockAngazovanjeRepository().Object);
        unitOfWork.Setup(x => x.Save()).Verifiable();
        return unitOfWork;}
```

Позив *.Setup()* обезбеђује да, када се одређена метода позове, врати оно што је позивом одређено, у овом случају, објекат класе конкретног репозиторијума или јединице рада [13].

Позив *It.IsAny<int>()* обезбеђује да ће метода вратити наведену ставку за било који валидан *int* [13].

Moq такође омогућава коришћење функције *„Verify*“ путем које можемо да осигурамо да је одређена *mock* -ована метода позвана одређени број пута [13].

За сваку класу за коју је имлементирана сервис класа креирана је посебна класа у којој су се писали одговарајући тестови у циљу тестирања пословне логике апликације. У наставаку рада приказане су имплементиране класе.

Класа *AngazovanjeTests*

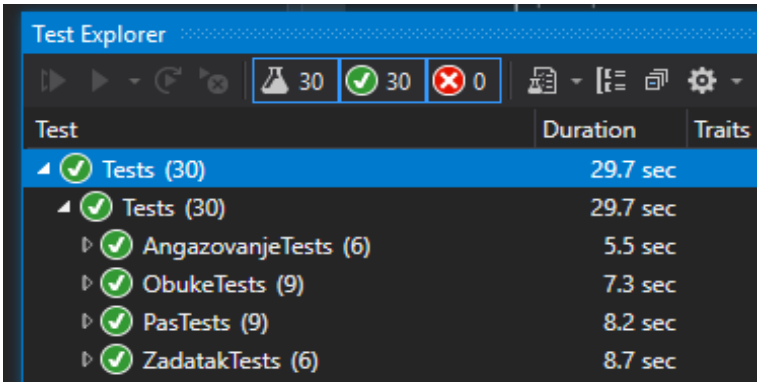
```
public class AngazovanjeTests
{
```

```
Mock<IUnitOfWork> unitOfWork = Mocks.GetMockUnitOfWork();

[Fact]
public void TestAngazovanjeServiceFindById()
{
    var service = new AngazovanjeService(unitOfWork.Object);
    var result = service.GetById(1,1);
    var resultAngazovanje = Assert.IsType<Angazovanje>(result);
    var expected = unitOfWork.Object.AngazovanjeRepository.GetById(1,1);
    Assert.Equal(expected.PasId, resultAngazovanje.PasId);
    Assert.Equal(expected.ZadatakId, resultAngazovanje.ZadatakId);
}
[Fact]
public void TestAngazovanjeServiceFindByIdInvalid()
{
    var service = new AngazovanjeService(unitOfWork.Object);
    var result = service.GetById(-9, 1);
   Assert.Null(result);
}
[Fact]
public void TestAngazovanjeServiceOceniAngazovanje()
{
    var service = new AngazovanjeService(unitOfWork.Object);
    var angazovanje = new Angazovanje
    {
        ZadatakId = 1,
        Pas = unitOfWork.Object.PasRepository.FindById(1),
        PasId = 1,
        Zadatak = unitOfWork.Object.ZadatakRepository.FindById(1),
        DatumUnosaOcene = DateTime.Now,
        Ocena = 10
    };
    service.Update(angazovanje);
    var angazovanjeOcenjeno = unitOfWork.Object.AngazovanjeRepository.GetById(1,1);
    Assert.NotNull(angazovanjeOcenjeno.Ocena);
    Assert.NotNull(angazovanjeOcenjeno.DatumUnosaOcene);
    unitOfWork.Verify(s => s.AngazovanjeRepository.Update(It.IsAny<Angazovanje>()), Times.Once);
    unitOfWork.Verify(s => s.Save(), Times.Once);
}
[Fact]
public void TestAngazovanjeServiceOceniAngazovanjeInvalid()
{
    var service = new AngazovanjeService(unitOfWork.Object);
    //ocena je nevalidna -> -2
    var angazovanje = new Angazovanje
    {
        ZadatakId = 1,
        Pas = unitOfWork.Object.PasRepository.FindById(1),
        PasId = 1,
        Zadatak = unitOfWork.Object.ZadatakRepository.FindById(1),
        DatumUnosaOcene = DateTime.Now,
       Ocena = -2
    };
    Assert.Throws<ArgumentOutOfRangeException>(() => service.Update(angazovanje));
    unitOfWork.Verify(s => s.AngazovanjeRepository.Update(It.IsAny<Angazovanje>()), Times.Never);
    unitOfWork.Verify(s => s.Save(), Times.Never);
}
[Fact]
public void TestAngazovanjeServiceDelete()
{
    var service = new AngazovanjeService(unitOfWork.Object);
    var toDelete = unitOfWork.Object.AngazovanjeRepository.GetById(1, 1);
    service.Delete(toDelete);
    var result = unitOfWork.Object.AngazovanjeRepository.GetById(1, 1);
    var lista = unitOfWork.Object.AngazovanjeRepository.GetAll();
    Assert.Null(result);
    Assert.DoesNotContain(lista, x => x.PasId == toDelete.PasId && x.ZadatakId == toDelete.ZadatakId);
    unitOfWork.Verify(s => s.AngazovanjeRepository.Delete(toDelete), Times.Once);
    unitOfWork.Verify(s => s.Save(), Times.Once);
```

```
        }
        [Fact]
        public void TestAngazovanjeServiceDeleteInvalid()
        {
            var service = new AngazovanjeService(unitOfWork.Object);
            // pasId is invalid id
            var toDelete = unitOfWork.Object.AngazovanjeRepository.GetById(100, 1);
            Assert.Throws<Exception>(() => service.Delete(toDelete));
            unitOfWork.Verify(s => s.AngazovanjeRepository.Delete(toDelete), Times.Never);
            unitOfWork.Verify(s => s.Save(), Times.Never);
    }
}
```

Аналогно овој класи, креиране су класе за тестирање пословне логике које су повезане са осталим ентитема: *Obuka, Pas, Zadatak*.

Резултати тестирања:



Слика 99 - *Резултати тестирања*

Овако написане тестове лако је извршити у паралели. Сваки тест тестира један сценарио. Прецизно су одређени подаци са којима ће се радити и користили су се mock - ови ради остваривања независности од других делова система.

9. Закључак

У раду *Развој софтверског система за војну обуку паса употребом .NET Core технологија* описана је архитектура комплексних апликација кроз комплетан поступак за креирање софтверских система.

Приказани су основни концепти .NET платформе, развојног оквира ASP .NET Core и програмског језика C#. Приликом развоја коришћен је MVC оквир због велике флексибилности, доста брзог и једноставног програмирања што је на крају значајно утицало на повећање продуктивности. За дизајн веб апликације коришћен је *Bootstrap* развојни оквир због великог броја већ готових стилова и *JavaScript* уз своју библиотеку *jQuery*. Уз помоћ *Bootstrap* и *JavaScript*-а се једноставно постиже респонзивност апликације и креира интуитивно корисничко искуство што је данас једна од најбитнијих ставки које веб апликације треба да задовоље. За складиштење података коришћена је *SQL* база података која је потпуно одговарала потребама апликације. Уз *Entity Framework Core* оквир и ОРМ библиотеке за аутоматизовање трансфера података између релационих табела и објеката модела у коду апликације, додатно је олакшан развој.

Завршни рад се састоји из две целине. Подељен је на теоријски и практични део. У теоријском делу су описани основни концепти и технологије које су коришћене за израду апликације. Практични део представља студијски пример развоја софтверског система који је реализован имплементацијом поменутих технологија.

Резултат практичног дела је веб апликација за војну обуку паса развијена на .NET платформи. Апликација је развијана кроз фазе упрошћене Ларманове методе обрађене на предмету Пројектовање Софтвера. Коришћење Ларманове методе омогућило је да развој апликације буде испраћен одговарајућом документацијом кроз све фазе развоја - прикупљање захтева, анализу, пројектовање, имплементацију и тестирање. Неки од најзахтевнијих задатака били су дефинисање захтева апликације који ће бити блиски реалним потребама корисника и омогућавање лаког сналажења приликом коришћења апликације.

Фаза тестирања је веома значајна. Након израде пројекта тестирана је пословна логика *UNIT* тестовима коришћењем *Moq* оквира, а остале фукнционалности тестиране су маунелно.

Иако развијена апликација чини једну заокружену целину која представља могући приступ решењу за праћење активности обука и евиденцију ангажовања војних службених паса, треба истаћи да постоји простор за напредак. Неке од функционалности које би додатно побољшале апликацију су: додавање нових улога у систему, омогућавање администраторима да креирају и шаљу емаил-ове запосленима и омогућавање запосленима да генеришу извештаје о задацима.

Уз архитектуру креирану у овој апликацији могуће је искористи слој сервиса и репозиторијума за креирање *ASP .NET Core WEB API* апликације: веб сервиса, што би даље омогућило коришћење оквира као што је на пример *AngularJS* на клијентској страни, ради веће брзине и стабилности учитавања страница. Процес израде рада представљао је изазован задатак будући да је на располагању велики број технологија за креирање веб апликација. Одлучила сам се за управо за ове технологије чије су основе обрађене на предмету *Напредне .NET технологије* на коме сам стекла неопходно основно знање за даљи самосталан рад. Област софтверског инжењерства: .NET и веб технологије и представљају област у којој желим даље да се усавршавам.

10. Литература

[1] Price, M. J. (2019). *C# 8.0 and .NET Core 3.0 - Modern Cross-Platform Development* (4th ed.). Van Haren Publishing.

[2] *Introduction to .NET Framework*. (2019, December 13). GeeksforGeeks. https://www.geeksforgeeks.org/introduction-to-net-framework/ [Последњи приступ 05.06.2020.]

[3] Pine, D. (2019, September 5). *Navigating the .NET Ecosystem*. InfoQ. https://www.infoq.com/articles/navigating-dotnet-ecosystem/ [Последњи приступ 28.05.2020.]

[4] B. (n.d.). *C# docs - get started, tutorials, reference.* Microsoft Docs. https://docs.microsoft.com/en-us/dotnet/csharp/ [Последњи приступ 05.06.2020.]

[5] Nakov, S., & Kolev, V. (2013). *Fundamentals of Computer Programming with C#* [E-book]. https://www.introprogramming.info/wp-content/uploads/2013/07/Books/CSharpEn/Fundamentals-of-Computer-Programming-with-CSharp-Nakov-eBook-v2013.pdf [Последњи приступ 05.06.2020.]

[6] R. (2013, February 18). *ASP.NET MVC 4 Fundamentals*. Microsoft Docs. https://docs.microsoft.com/en-us/aspnet/mvc/overview/older-versions/hands-on-labs/aspnet-mvc-4-fundamentals [Последњи приступ 07.06.2020.]

[7] Lerman, J. (n.d.). *Entity Framework Core 3.0: A Foundation for the Future*. Code Magazine. https://www.codemag.com/Article/1911062/Entity-Framework-Core-3.0-A-Foundation-for-the-Future [Последњи приступ 07.06.2020.]

[8] Albahari, J., & Albahari, B. (2014). *C# 5.0 Mali referentni priručnik* (1st ed.). Mikro knjiga. https://docplayer.net/53344411-C-5-0-mali-referentni-prirucnik.html

[9] Westhuizen, P., & Van der Westhuizen, P. (2014). *Bootstrap for ASP.NET MVC*. Van Haren Publishing.

[10] Kantor, I. (2020). *The Modern JavaScript Tutorial*. JavaScript Info. https://javascript.info/ [Последњи приступ 10.06.2020.]

[11] Vlajić, S. (2015). *Projektovanje softvera (skripta)*. Beograd, Srbija: FON.

[12] Aroraa, G., & Chilberto, J. (2019). *Hands-On Design Patterns with C# and .NET Core*. Van Haren Publishing.

[13] R. (2017, January 31). *Get Started with Unit Testing – using xUnit and Moq*. CodeProject. https://www.codeproject.com/articles/1168172/get-started-with-unit-testing-using-xunit-and-moq [Последњи приступ 10.06.2020.]