Primjena slojevite arhitekture u razvoju stolne aplikacije za emitiranje zvučnih obavijesti

Koprek, Zvonimir

Undergraduate thesis / Završni rad

2022

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Zagreb, Faculty of Organization and Informatics / Sveučilište u Zagrebu, Fakultet organizacije i informatike**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:211:853573

Rights / Prava: Attribution-NonCommercial-ShareAlike 3.0 Unported/Imenovanje-Nekomercijalno-Dijeli pod istim uvjetima 3.0

Download date / Datum preuzimanja: **2024-03-29**

SVEUČILIŠTE U ZAGREBU
FAKULTET ORGANIZACIJE I INFORMATIKE
VARAŽDIN

Repository / Repozitorij:

Faculty of Organization and Informatics - Digital Repository

zir.nsk.hr

dabar
DIGITALNI AKADEMSKI ARHIVI I REPOZITORIJI

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Zvonimir Koprek

PRIMJENA SLOJEVITE ARHITEKTURE U RAZVOJU STOLNE APLIKACIJE ZA EMITIRANJE ZVUČNIH OBAVIJESTI

ZAVRŠNI RAD

Varaždin, 2022.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Zvonimir Koprek

JMBAG: 0016135996

Studij: Poslovni sustavi

PRIMJENA SLOJEVITE ARHITEKTURE U RAZVOJU STOLNE APLIKACIJE ZA EMITIRANJE ZVUČNIH OBAVIJESTI

ZAVRŠNI RAD

Mentor:

Dr. sc. Marko Mijač

Varaždin, rujan 2022.

Zvonimir Koprek

Izjava o izvornosti

Izjavljujem da je moj završni rad izvorni rezultat mojeg rada te da se u izradi istoga nisam koristio drugim izvorima osim onima koji su u njemu navedeni. Za izradu rada su korištene etički prikladne i prihvatljive metode i tehnike rada.

Autor potvrdio prihvaćanjem odredbi u sustavu FOI-radovi

Sažetak

U ovom završnom radu opisana je izrada aplikacije za stolna računala s Windows operativnim sustavom koja je namijenjena za reproduciranje zvučnih obavijesti u školskim ustanovama. Sastoji se od dva glavna dijela. U teorijskom su predstavljeni i objašnjeni svi korišteni alati za realizaciju projekta, a to su razvojno okruženje Microsoft Visual Studio 2022, softverski okvir .NET Framework i baza podataka Microsoft SQL Server 2019. Nakon toga slijedi detaljna analiza glavne tematike završnog rada, slojevite arhitekture, ali i kratko pojašnjavanje termina arhitekturnog uzorka općenito. Drugi dio je praktični dio koji se tiče pojašnjavanja i navođenja svega napravljenog u procesu izrade aplikacije i onoga što ona korisnicima nudi. Na kraju cjelokupni rad i priča oko njega bude zaključena i izneseni su dojmovi autora prilikom izrade istog.

Ključne riječi: C#, slojevita arhitektura, stolna aplikacija, programiranje, zvučne obavijesti**,** razvoj aplikacije, arhitekturni uzorci, NAudio, .NET Framework

Sadržaj

Sadržaj 1
1. Uvod 2
2. Metode i tehnike rada 3
2.1. Opis korištenih alata 4
2.1.1. Microsoft Visual Studio 2022 4
2.1.2. .NET Framework 6
2.1.3. Microsoft SQL Server 2019 7
3. Arhitekturni uzorci 9
3.1. Općenito o arhitekturnim uzorcima 9
3.2. Slojevita arhitektura 13
3.2.1. Karakteristike i koncepti slojevite arhitekture 13
3.2.2. Analiza slojevite arhitekture 19
4. Razvoj aplikacije 20
4.1. O aplikaciji 20
4.1.1. Ideja, primjena i preduvjeti aplikacije 20
4.1.2. Opis sučelja 21
4.1.3. Projektna struktura 27
4.2. Funkcionalnosti i tokovi procesa u aplikaciji 29
4.2.1. Registracija i prijava 29
4.2.2. Sinkronizacija s bazom podataka 32
4.2.3. Hitna evakuacija 34
4.2.4. Dodavanje zvučnog zapisa 35
4.2.4.1. Uvoz zvučnog zapisa 36
4.2.4.2. Snimanje zvučnog zapisa 38
4.2.5. Prijenos govora uživo 40
4.2.6. Postavljanje rasporeda sati 41
4.2.7. Upravljanje zapisima 43
4.2.8. Odjava 45
5. Zaključak 46
6. Popis literature 47
6.1. Popis literature korištene za teorijski dio 47
6.2. Popis literature korištene za programski kod 48
7. Popis slika 49
8. Popis tablica 50

1. Uvod

Isporuka aplikacije konačni je cilj svakog programera i da bi se on ostvario, potrebno je proći kroz proces izrade u kojem se mora, između ostalog, osmisliti način njezinog sastavljanja. Poučeni iskustvom, ljudi su shvatili da se, kao i u mnogim situacijama u životu, taj ponavljajući proces može olakšati i to na način da se osmisle uzorci koji bi se mogli slijediti ovisno o vrsti aplikacije koja se planira napraviti, njezinoj namjeni te o vrsti tehnologije koja se prilikom toga namjerava koristiti. To se radi u svrhu postizanja mnogobrojnih prednosti i ušteda, a u konačnici i boljih rezultata.

U kontekstu razvoja programskih proizvoda, arhitekturni uzorci predstavljaju kostur organizacije programskog koda projekta. Oni određuju kakvog će izgleda projektna struktura biti te raspodjelu uloga, ali i donose pravila kojih se programer mora držati kako ne bi otišao u krivi smjer. Drugim riječima, arhitekturni uzorak skup je putokaza kojima nastojimo cjelokupni programski kod zadržati unutar granica dobro osmišljenog načina rješavanja problema za konkretnu situaciju u praksi.

Danas je aktualno desetak takvih uzoraka, s time da uvijek postoji prostora za nastajanje novijih, modernijih i suvremenih. Prema jednom izvoru literature među svima isplivalo je njih 5 koji su diljem svijeta najprihvaćeniji i najčešće korišteni. Ta brojka će se u ovom radu svesti na samo jedan, nazvan slojevita arhitektura, koji će biti detaljno analiziran i po čijem ću vas principu provesti kroz shemu izrade jedne stolne aplikacije za emitiranje zvučnih obavijesti, namijenjenu za sve ovdašnje školske ustanove.

Ovaj rad sastoji se od dvije glavne cjeline, prva predstavlja teorijski dio koji započinje poglavljem u kojem ću objasniti korištene alate za izradu navedene stolne aplikacije, a to su *.NET Framework* u C# programskom jeziku, razvojno okruženje *Visual Studio 2022* i SQL Server 2019 američke tvrtke Microsoft. Potom slijedi kratak uvod u tematiku arhitekturnih uzoraka, a prije nego što se dođe do slojevite arhitekture, biti će predstavljeni ostali najpoznatiji uzorci radi pružanja šire slike čitateljima. Kulminaciju te cjeline čini detaljna analiza slojevitog arhitekturnog uzorka, navođenje i objašnjavanje njegovih karakteristika i čega se sve treba pridržavati ako se on koristi.

Cijeli drugi dio rada služi tome da se korak po korak prođe kroz sve dijelove izrađene stolne aplikacije u svrhu implementacije arhitekturnog uzorka koji čini glavnu tematiku rada. Navode se sve funkcionalnosti aplikacije po logičkom, ali i kronološkom redoslijedu odvijanja kada se ona upotrebljava. Pojašnjava se kome je i za koje radnje aplikacija namijenjena, ali i koji su preduvjeti potrebni da bi se mogla koristiti na zamišljen način. Prikazuje se njezin izgled sučelja, struktura projekta, a rad je naposlijetku zaokružen zaključkom.

2. Metode i tehnike rada

Za svrhu izrade ovog rada odlučio sam koristiti dobro poznati programski jezik C#, u *.NET Framework-u* i u razvojnom okruženju *Microsoft Visual Studio 2022*. Radi jednostavnosti, C# korišten u tom softverskom okviru se u različitim izvorima literature i člancima internetskih medija često naziva samo *Visual C#*.

Na fakultetu smo se u sklopu određenih kolegija već upoznali s ovim načinom izrade aplikacija pa mi je to predznanje naročito pomoglo odrediti što i kako raditi, a i tijekom same izrade imao sam prilike vratiti se na nastavne materijale kolegija Programsko inženjerstvo, kako bih se prisjetio određenih koncepata kod pisanja programskog koda.

Velik dio ove aplikacije oslanja se na već dostupne funkcionalnosti u biblioteci *NAudio*, koju sam dodao preko *NuGet Package Manager*-a u svoje razvojno okruženje, a kroz ovaj rad detaljno ću proći kroz sve njezine korištene klase i metode. Također, da bih ispravno koristio sve ono što se u njoj nudi, prije svakog novog značajnijeg koraka prilikom izrade, proučio sam službenu dokumentaciju kreatora ove biblioteke na stranici *GitHub*, gdje postoje poveznice na sve članke u kojima se opisuje svaka od tih mogućnosti. Valja napomenuti kako je korištenje biblioteke *NAudio* u potpunosti besplatno, a autor se u člancima uz priložene slike zaslona potrudio kratko i jasno objasniti svaki korak u implementaciji određene ponuđene funkcionalnosti.

Aplikaciju sam započeo raditi na način da sam prvo napravio četiri zasebna sloja, svaki od kojih je zapravo samostalan projekt, ali svi su u istom *solutionu* i međusobno se referenciraju kako bi mogli biti od pomoći jedni drugima i činiti jedinstveni sustav. Nakon toga je slijedila izrada prve ideje sučelja aplikacije pomoću *Windows Forms open-source* biblioteke s grafičkim klasama, a raspored, izgled i funkcionalnosti elemenata i korisničkih kontrola su se naravno mijenjali kako je razvoj aplikacije napredovao, u skladu s potrebama i željama. Da bi konačni izgled i kvaliteta usluge aplikacije bili na zadovoljavajućoj razini, poslužio sam se vodičima na *YouTube*-u, nastavnim materijalima, materijalima sa *Mendeleya* i *Google Scholara*, te člancima vezanim za ovu tematiku na internetu. Izvori iz literature pomogli su da se principi slojevite arhitekture pravilno pretoče iz teorije u praksu.

Smatrao sam da je dokumentaciju pametno početi pisati kad je većina zacrtanih funkcionalnosti već implementirana u aplikaciji, a primjenu željenog dizajna same aplikacije ostavio sam za kraj, što znači da sam većinu vremena radio u tvorničkom izgledu formi aplikacije kojeg nudi *Windows Forms*. Okvirnu strukturu poglavlja za dokumentaciju sastavio sam prije početka pisanja.

Konačnim ishodom aplikacije sam zadovoljan, iako sam mišljenja da se programerima u ovom razvojnom okruženju mogla dati ipak nešto veća sloboda oblikovanja dizajna prozora i elemenata korisničkog sučelja poput gumbova i tablica. Alat se unatoč tome pokazao kao vrlo intuitivan način za korištenje u ovu svrhu. Razvojno okruženje ima još podosta dodatnih funkcionalnosti koje su pridonijele podizanju kvalitete ovog razvojnog procesa i skraćivanju njegovog ukupnog trajanja.

Ovakvi složeniji projekti općenito zahtijevaju maksimalno ulaganje truda i iznimnu predanost poslu, a ništa drugačije nije bilo ni u mojem iskustvu. Ono što mi je izuzetno drago je da sam prvi puta dobio priliku samostalno izraditi jednu stolnu aplikaciju u konkretnu svrhu, a prilikom čega sam naučio mnogo novih i vrijednih stvari.

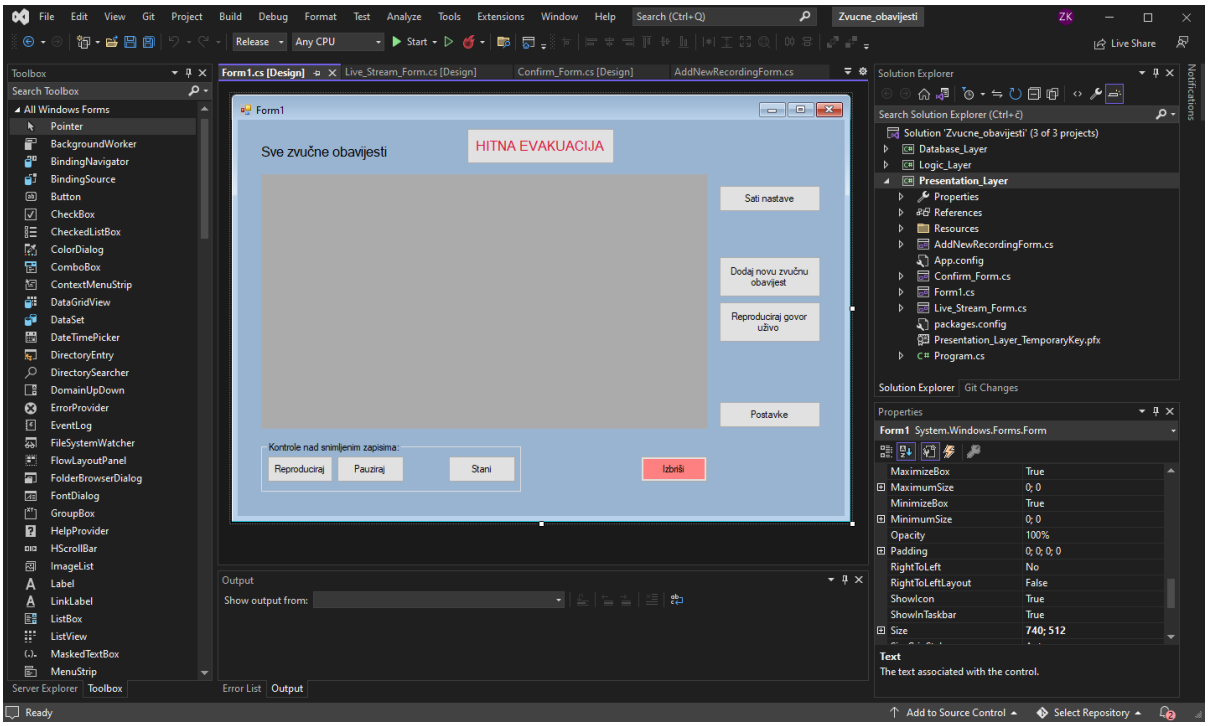
2.1. Opis korištenih alata

Ovo je cjelina u kojoj mi je namjera čitatelja upoznati s glavnim alatima koji su upotrebljavani pri izradi stolne aplikacije „Zvučne obavijesti“, *Microsoft Visual Studio 2022*, *.NET Framework* i *SQL Server 2019*.

2.1.1. Microsoft Visual Studio 2022

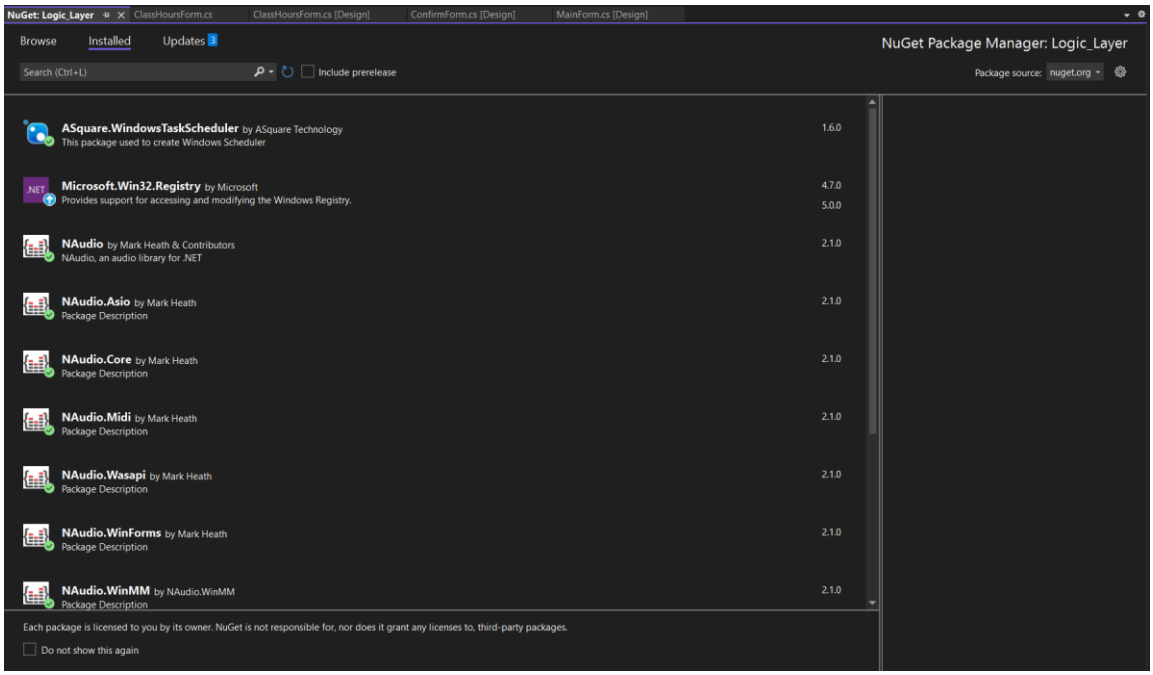
Službeno predstavljeno kao integrirano razvojno okruženje, Microsoft je ovaj alat napravio za programere kojima je želja izrađivati računalne programe, web stranice, web servise, ali i mobilne aplikacije. Okosnicu mu čine platforme za razvoj software-a istoimene tvrtke, primjerice *Windows API*, *Windows Forms (WinForms)*, *Windows Presentation Foundation (WPF)*, Windows trgovina i slične [1]. Također, važno je spomenuti da je to alat koji podržava takozvani *Intellisense*, generički izraz za brojne mogućnosti koje su nadodane da se fluidnije piše programski kod, kao što su nadopuna postojećeg koda, informacije parametara te brzi prikaz informacija [2].

Alat je to koji podržava korištenje 36 različitih programskih jezika i ima integrirani *debugger* koji funkcionira i kao *source-level debugger* i kao *machine-level debugger*. Jedan od unaprijed ugrađenih programskih jezika je i C# korišten ovom prilikom, a posljednja inačica je za godinu 2022 [1].



Slika 1: Snimka zaslona sučelja u programu Microsoft Visual Studio 2022 [autorski rad]

NAudio biblioteku koju ću kasnije često spominjati, dodao sam preko korisničkog sučelja *NuGet Package Manager*, kojem se može pristupiti na više načina, a jedan je ako se u *Solution Exploreru* klikne desnom stranom miša. Pritom se otvori izbornik, klikne se na *Manage NuGet Packages*, ide se na karticu *Browse*, pronađe se ta biblioteka i odmah se može instalirati [3].



Slika 2: Snimka zaslona dodanih biblioteka [autorski rad]

Projekte, odnosno u slučaju ove aplikacije - slojeve, međusobno sam povezao na način da sam kliknuo na ime sloja, nakon čega se desnim klikom otvori izbornik gdje se odabere *Add*, a potom *Reference* u novom izborniku. U dijaloškom okviru koji se zatim otvori se s lijeve strane odabere *Projects* i prikažu se dostupni projekti [4].

2.1.2. .NET Framework

Tijekom izrade ovog rada, uvelike je pomogla tehnologija koja je zadužena za stvaranje baš ovakvih vrsta aplikacija, na *Windows* operativnom sustavu, nazvana *.NET Framework*.

Prvog izdanja 13. veljače 2002. godine, tvrtka Microsoft razvila je kostur koji podržava stvaranje i pokretanje Windows aplikacija i web servisa, s time da su glavne namjere bile pružiti programerima:

- dosljedno objektno orijentirano okruženje bilo za objektni kod pohranjen i pokretan lokalno ili na daljinu
- okruženje za izvršavanje koda koje će minimizirati moguće konflikte kod verzioniranja i isporuke
- okruženje koje će promovirati sigurno izvršavanje koda, bez obzira je li sastavljen od skroz nepoznate ili nepotpuno povjerljive treće strane i izbjeći probleme u performansama pri interpretiranim okruženjima
- dosljedno, nepromjenjivo te kvalitetno iskustvo razvoja aplikacija različitih vrsti
- primijeniti sve industrijske standarde da se osigura glatka integracija koda temeljenog na ovom kosturu s bilo kojim drugim [5]

.NET Framework čini zapravo samo jedan dio *.NET platforme*, koja je skup tehnologija za razvoj aplikacija na operativnim sustavima i koji nisu *Windows*, poput *Linux-a*, *macOS-a*, *iOS*, *Androida* i ostalih. Posljednja inačica je *.NET Framework 4.8*, koja je korištena u ovom projektu.

Sastoji se od dvije glavne komponente. *Common Language Runtime (CLR)* ima zadatak da prilikom izvršavanja aplikacija upravlja dretvama i iznimkama, sklanja nepotrebna sredstva, osigura ispravnu dodjelu tipa određenim vrijednostima (ispravnost tipa) i *Class Library* koja je, kao što joj naziv sugerira, biblioteka za pružanje tipova za slovne nizove, datume, brojeve itd. Uz to, druga komponenta također sadrži i API-ove za čitanje i pisanje datoteka, spajanje na baze podataka, crtanje i još mnogo toga [6].

Class Library komponenta posjeduje tipove i članove koji su unaprijed testirani i korisnici mogu u vlastitim aplikacijama, ovisno o potrebi, pozvati njezine sadržaje da im se olakša implementacija njihovih ideja u različitim slučajevima. Korisna je dakle zbog toga jer bi

se inače za mnoge vrlo jednostavne i ponavljajuće operacije morale ručno pisati naredbe za računalo.

Iako postoje rijetke iznimke, aplikacije koje su nastale u određenoj inačici *.NET Framework-a* mogu se bez poteškoća pokretati i na novijim inačicama. Osim toga, na jednom računalu se podržava istovremeno korištenje više inačica *CLR-a*, što znači da više verzija iste aplikacije može paralelno postojati i sve se mogu pokretati bez konflikata. Usluga je to poznata pod nazivom *Side-by-side* izvršavanje, koja se može primijeniti na grupnim verzijama 1.0/1.1, 2.0/3.0/3.5 i 4/4.5.x/4.6.x/4.7.x/4.8.

Još jedna korisna stvar koju *.NET Framework* nudi jest sustav uobičajenog tipa podatka, zbog kojeg su tipovi korišteni u ovom softverskom okviru definirani tim sustavom i na taj način se mogu koristiti u bilo kojem programskom jeziku koji ga koristi. U tradicionalnim programskim jezicima to zna otežavati stvari jer su ondje tipovi definirani kompajlerima.

Upravljanje memorijom računala u ovom slučaju također nije briga programera. Za razliku od programskog jezika *C++*, *.NET Framework* čini automatsku alokaciju i dealokaciju računalne memorije i rukovanje životnim vijekom objekata. U ime aplikacije to sve preuzima *CLR* komponenta. Istu uslugu nude još primjerice *Java* i *Python*.

Nije potrebno nikakvo dodatno znanje za korištenje *.NET Framework* aplikacija jer je skoro u potpunosti transparentan za korisnike. Pri instalaciji, ako na računalu već ne postoji potrebna inačica, sustav za instalaciju programa će ga, iako ne u svim slučajevima, automatski instalirati [7]. Izvor koji je ujedno i službena dokumentacija, navodi da je od velike važnosti činjenica da su sve inačice poslije *.NET Framework 4* takozvana *in-place* ažuriranja, što znači da ne može više 4.x verzija biti prisutno istovremeno na računalu. Tako je slučaj i sa 3.5 koji je *in-place* inačica verzije 2.0.

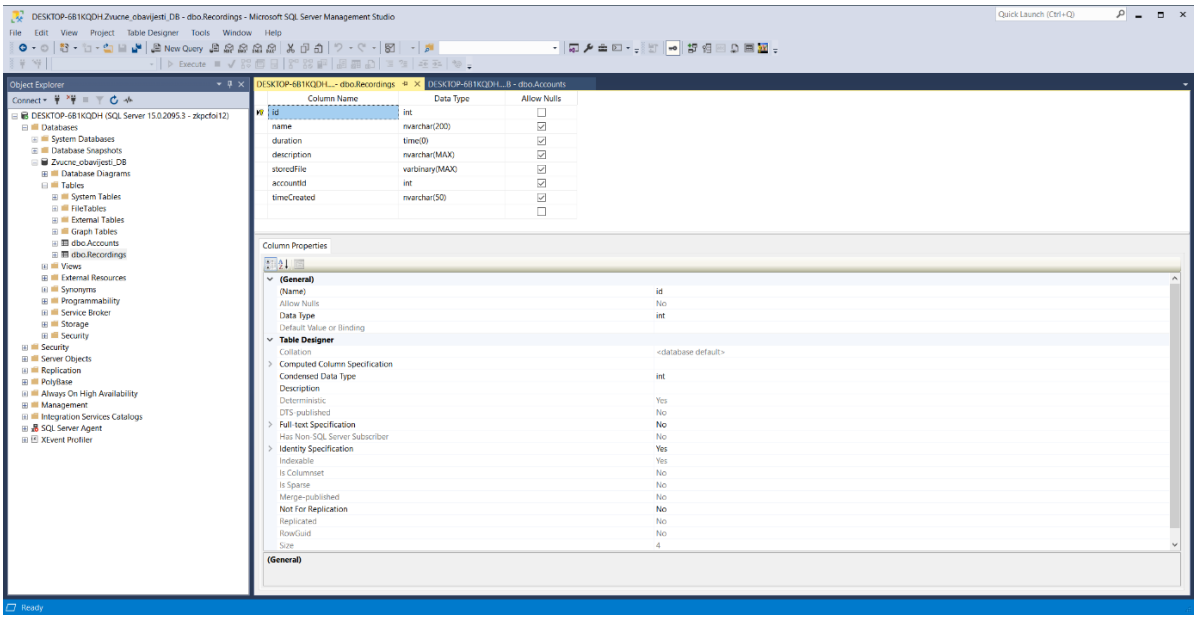
2.1.3. Microsoft SQL Server 2019

Kao bazu podataka odlučio sam koristiti poznati i pouzdani proizvod tvrtke *Microsoft*, *Microsoft SQL Server* verzije iz 2019. godine. Konkretno, odlučio sam se za *Developer* izdanje tog proizvoda, nakon kojeg je bilo potrebno preuzeti i instalirati i okruženje u kojem se može upravljati njime, nazvano *SQL Server Management Studio*.

Općenito gledano, baza podataka je softverski proizvod kojem su ciljevi pohranjivanje i dohvaćanje podataka iz aplikacija koje se njime koriste, a koje mogu biti pokrenute na *host* računalu ili na nekom udaljenom. Povijest počinje 1989. godine kada je objavljena prva verzija, 16 bitna i namijenjena za računala s OS/2 operativnim sustavima [7].

Po nazivu vidimo da koristi standardizirani SQL programski jezik (eng. *Structured Query Language*) razvijen da bi se njime moglo upravljati relacijskim bazama podataka i da bi

se mogle vršiti razne operacije nad podacima u njima. Podaci se slažu u tablice koje su najčešće korišteni objekti u ovu svrhu, sastoji se od stupaca s nazivima atributa i redaka u kojima se nalaze vrijednosti za svaki [8].



Slika 3: Snimka zaslona sučelja u Microsoft SQL Server Management Studio [autorski rad]

3. Arhitekturni uzorci

Da bismo došli do samog srca ovog završnog rada, potrebno je predstaviti i općenito objasniti što su to arhitekturni uzorci sustava u smislu izgradnje aplikacija. U ovom poglavlju ću stoga objasniti taj pojam, a potom ući u dublju analizu uzorka korištenog za ovaj rad. Ni ovdje, bez obzira na to koji pristup se odabere, nijedan nije besprijekoran i savršen, svaki ima svoje nedostatke i prednosti, stoga se itekako ima o čemu pisati.

3.1. Općenito o arhitekturnim uzorcima

Vrlo česta je pojava da programeri počnu programirati bez nekog znanja što se tiče službene arhitekture sustava. Izvor, točnije autor Mark Richards, navodi kako to vrlo često rezultira „*skupinom neraspoređenih modula izvornog koda kojima nedostaju točno određene uloge, odgovornosti i međusobna povezanost*“, slučaj nazvan *big ball of mud* ili u prijevodu kugletina blata. Nadalje, neke od karakteristika aplikacija na kojima se tako radi su: uska povezanost, složena izmjena, lomljivost i nedostatak točne vizije i smjera. Osobnosti takve arhitekture, ako se to uopće može tako nazvati, teško se razaznaju, a još teže se mogu odgovoriti sljedeća pitanja [9]:

- Može li se primijenjena arhitektura skalirati?
- Kakve su performanse aplikacije?
- Koliko neprimjetno aplikacija reagira na promjene?
- Kakve su karakteristike isporuke?
- Koliko je arhitektura responzivna?

Definicija općenitog pojma arhitekturnog uzorka prema Wikipediji glasi: „*generičko, višekratno upotrebljivo rješenje za učestalo ponavljajući problem u softverskoj arhitekturi unutar određenog konteksta*“. Neki uzorci su čak implementirani unutar softverskih okvira da bi se mogli lakše koristiti, a mogu se doticati problema poput ograničenosti računalnih performansi, velike dostupnosti i poslovnih rizika [10].

Potrebno je poznavati ponašanje aplikacije, a to se može samo ako je ona izgrađena na robusnom, čvrstom principu. Analogno, na pamet mi padaju gradski neboderi koji se mogu lako povezati s ovom temom pošto ih je uvijek potrebno smisleno i planski graditi jer moraju dugo trajati i jamčiti sigurnost ljudi koji u njemu borave i biti što otporniji na promjene i vanjske utjecaje poput potresa te ostalih prirodnih i neprirodnih nepogoda.

Da bi se neki uzorak mogao odabrati, važno je znati jačine i slabosti svakog. Izvor spominje da je pri pravilnom pristupu u razvoju neophodno trebati moći opravdati svaki potez koji se odluči napraviti i znati da je on ispravan, a pogotovo ako je odabran i ako se primjenjuje određeni uzorak [9].

Smatra se da je radi povećane složenosti općenito u današnjem softveru posebno mučno postalo razvijati, održavati i poboljšavati te sustave. Upravo zbog toga su nastali arhitekturni uzorci kako bi softverskim arhitektima, developerima i operatorima olakšali obavljati takve zadatke. Na njih se može gledati kao formalizirane najbolje prakse koje poduzeće može koristiti kako bi se što lakše prebrodile uobičajene poteškoće koje se javljaju prilikom izrade softverskog proizvoda [10]. U idućoj tablici navedene su prednosti i nedostaci korištenja arhitekturnih uzoraka.

Tablica 1: Analiza općenitih prednosti i nedostataka primjene arhitekturnih uzoraka

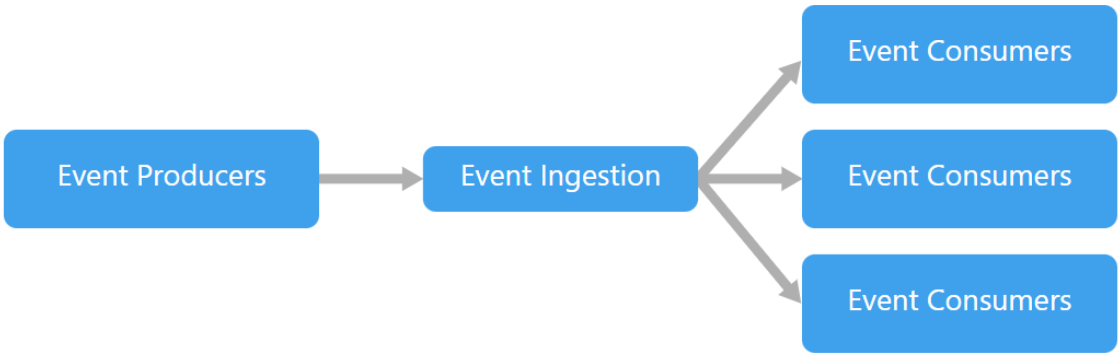
Prednosti	**Nedostaci**
Definira se model i način funkcioniranja softvera, što rezultira kvalitetnijim procjenama troškova i vremena potrebnog za izradu projekta	Ukoliko se odabere pogrešan uzorak za određeni slučaj, sveukupni troškovi i vrijeme se povećavaju zbog potrebe naknadne prilagodbe proizvoda tom uzorku
Upotreba dobro poznatog, višekratnog i testiranog načina rješavanja problema	Najmodernija rješenja možda neće odmah imati dostupne odgovarajuće uzorke spremne za korištenje
Prilika za smanjenje sveukupnih troškova i rizika	Pogrešan odabir također nepotrebno povećava složenost cijele aplikacije
Postizanje viših standarda pošto se njihovom primjenom dobiva višestruko provjerljiv i sljediv razvojni proces	

(Izvor: https://altitudeaccelerator.ca/software-architecture-design-patterns/)

Nakon pregleda ove tablice, jasno možemo zaključiti kako je s razlogom najveći strah kod primjene uzoraka pitanje hoće li odabrani uzorak biti adekvatan za aplikaciju koja se gradi. Ako se previše skrene s puta štete od njihove primjene biti će značajnije nego koristi, pa se zato prije svega jasno mora poznavati priroda i svrha te aplikacije.

Arhitekturnih uzoraka danas postoji mnogo, a spomenuti ću samo one najkorištenije. Na prvom mjestu je onaj iz naslova ovog rada, slojevita arhitektura, o kojemu će više riječi biti u idućoj cjelini.

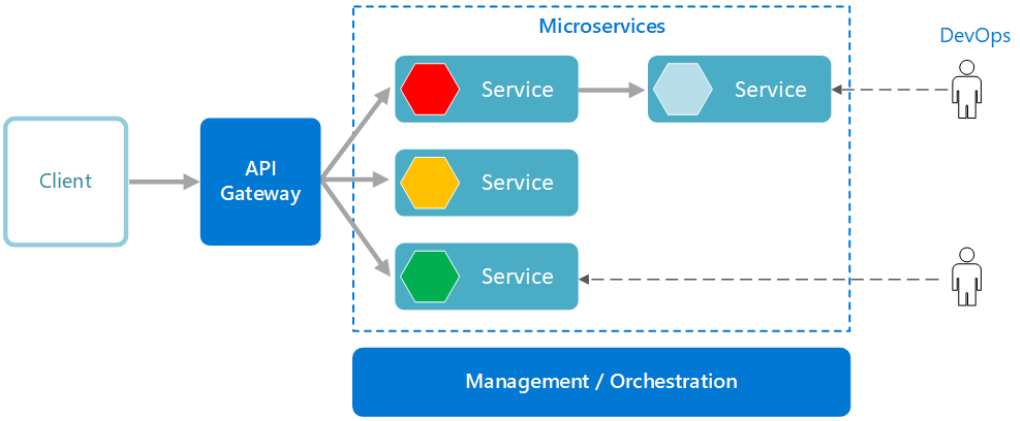
Event-based uzorak najbolje odgovara sustavima u kojima se većinu vremena provede čekajući da se nešto dogodi što će pokrenuti točno onaj proces koji je namijenjen za taj događaj. Tu postoji središnja jedinica koja prima podatke i preusmjerava ih prema odvojenim cjelinama koje su zadužene za konkretnu reakciju u toj situaciji. On se upotrebljava najučinkovitije na asinkronim sustavima koji su temeljeni na asinkronom toku podataka, korisničkim sučeljima i aplikacijama gdje zasebni blokovi koda vrše interakciju samo s nekim modulima.



Slika 4: Model event-based arhitekturnog uzorka (Izvor: https://docs.microsoft.com/en-us/azure/architecture/guide/architecture-styles/event-driven, pristupano 2022.)

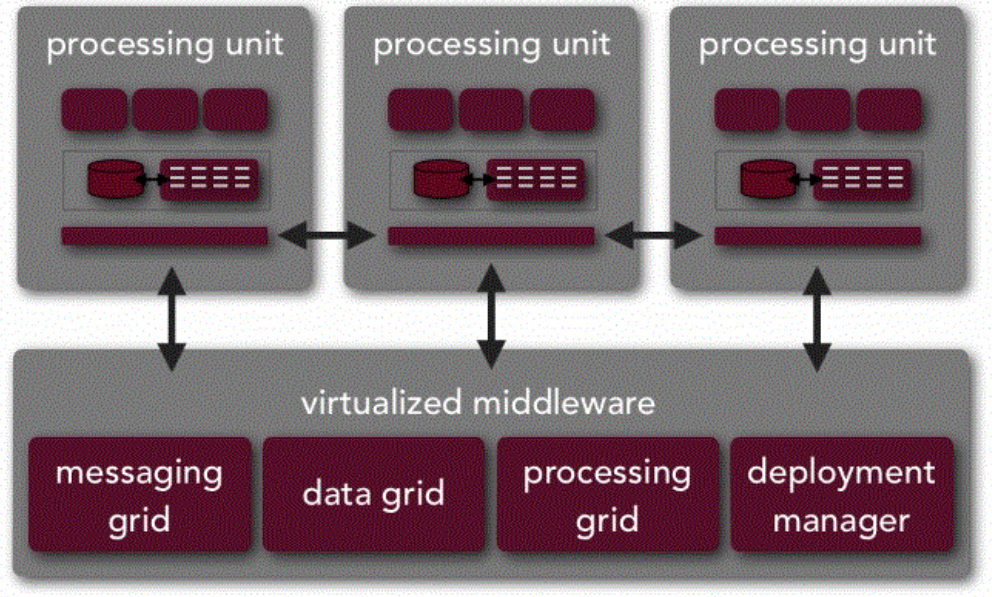
Microkernel uzorak s druge strane idealan je za one aplikacije gdje su točno definirane rutine višeg reda i gdje postoji dinamični broj pravila nižeg reda koja se moraju ažurirati redovito, a takve aplikacije koristi širok spektar ljudi.

Uzorak mikroservisa možda je i najlakše predstaviti jer se u principu radi o sustavu gdje se sve može podijeliti u komponente, svaka od kojih ima svoju zadaću i poziva se kada na nju dođe red. Takav uzorak je najbolje koristiti primjerice na web aplikacijama, gdje kada se želi dodati neka nova funkcionalnost, samo se napravi nova komponenta te se uvrsti u sustav. Poznata *Javascript* biblioteka *React* nudi takav pristup dizajnu, a komponente se slažu kao lego kockice na web stranici i jedna nadopunjuje drugu te pružaju vrlo kvalitetno korisničko iskustvo i dojam da je sve zapravo jedna cjelina.



Slika 5: Model arhitekturnog uzorka mikroservisa (Izvor: https://docs.microsoft.com/en-us/azure/architecture/guide/architecture-styles/microservices, pristupano 2022.)

Space-based uzorak je napravljen da svoju aplikaciju učini robusnom i izdržljivom kada baza podataka ne može pratiti zahtjeve koji joj se šalju, te kada pritom cijeli sustav teško ili uopće više ne funkcionira. U ovom slučaju se procesuiranje zadataka i upravljanje memorijom podijeli među više poslužitelja da bi se opterećenje lakše podnijelo pod velikim pritiskom nastalog korištenjem tog sustava. Ovaj princip koriste društvene mreže, koje imaju planski raspoređene poslužitelje diljem svijeta gdje god je predviđeno da će biti potrebne njihove usluge i doprinos sustavu. Zanimljivo je da takvim sustavima ne smeta sporadično gubljenje podataka, što predstavlja „nuspojavu“ kada se organizacija odluči osloniti na ovaj uzorak [10].



Slika 6: Model space-based arhitekturnog uzorka (Izvor: https://en.wikipedia.org/wiki/Space-based_architecture, pristupano 2022.)

Na kraju ovog poglavlja, zaključit ću da je neophodno bilo spomenuti ostale arhitekturne uzorke kako bismo razumjeli da svaki ima svoju idealnu primjenu i ako se ona u praksi pogodi, mogu se predviđeni zadaci zaista lakše i kvalitetnije izvesti. Tu ne treba ni spominjati kako će aplikacija kojoj se od samog začetka ideje pridaje dovoljno pažnje, dugoročno biti „zdravija“ i kako njezinim stvarateljima, tako i zajednici korisnika, biti zahvalnija i jednostavnije će se s njom moći baratati.

3.2. Slojevita arhitektura

Na engleskom *n-tier* ili slojevit arhitekturni uzorak, ili barem njegova struktura, mnogim početnicima u programiranju vjerojatno bi prvo pao na pamet kada bi ih se pitalo kako misle da bi im aplikacija trebala izgledati, a da to ima smisla i da je sve nekako logički povezano. Napraviti glavnu raspodjelu u slojeve, dodijeliti im zadatke i da se svaki put kada je potrebno vršiti interakciju s aplikacijom, svaki od njih koji je zadužen za to i koristi, bila bi prvotna zamisao.

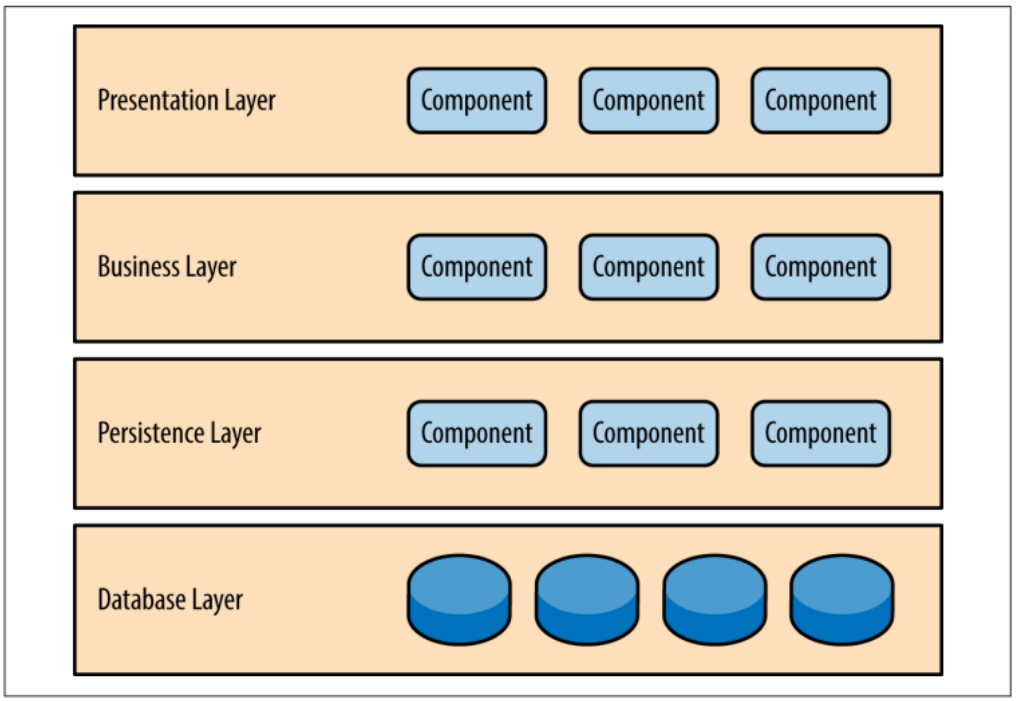
Tražeći informacije po internetu, često sam primijetio naizmjenično korištenje pojmova *n-tier* i *n-layer* u smislu karakteristika aplikacija. Potrebno ih je dodatno pojasniti s obzirom na to da nisu sinonimi. *N-tier* predstavlja fizičku raspodjelu zadataka, bilo po procesima, dretvama ili računalima na kojima se pokreću određeni dijelovi jednog sustava. Primjer *N-tier* sustava je web aplikacija koja se u dijelovima pokreće na korisničkom računalu, na poslužitelju i u bazi podataka. S druge strane, *n-layer* se odnosi na logičku raspodjelu koda i upravo ona je u središtu pažnje u ovom radu [11]. Iako glavni izvor literature spominje *n-tier* kao drugi naziv za slojevitu arhitekturu, aplikacija se u ovom slučaju odvija u samo dvjema fizičkim jedinicama, korisničkom računalu i bazi podataka.

Ovaj uzorak se poklapa s tradicionalnim organizacijskim strukturama u poduzećima, pa je zbog toga prirodan izbor za mnoge poslovne aplikacije [9].

3.2.1. Karakteristike i koncepti slojevite arhitekture

Pri korištenju ovog uzorka nema točno definiran broj slojeva koji se moraju koristiti, međutim najčešće ih bude četiri, a to su sloj prikaza (eng. *presentation layer*), poslovni (eng. *business layer*), sloj perzistencije (eng. *persistence layer*) te sloj baze podataka (eng. *database layer*). Horizontalno su raspodijeljeni, što znači da uvijek sve polazi od jednog koji je na vrhu te strukture, pa se ostalima prosljeđuje s obzirom na njihove uloge što trebaju učiniti. U nekim slučajevima se dva sloja koja su u sredini spajaju u jedinstveni. To bude obično kod manjih

aplikacija, pa ih sveukupno imaju samo tri, a kod većih i složenijih može postojati pet ili više slojeva. Unutar slojeva se nalaze komponente, kao što se može vidjeti na sljedećoj slici [9].



Slika 7: Model slojevite arhitekture (Izvor: Richards, 2015)

Od velikog značaja je činjenica da su slojevi izolirani, ono što se pojedinog tiče nalazi se samo u njegovoj domeni. Ipak, oni koji su na višoj razini apstrakcije ovise o neposredno nižem sloju. Programski kod koji je po nekoj logici član jedne skupine unutar tog sustava može se jasno odvojiti i zatvoriti unutar jedne lokacije u projektu. Kad bi se pokušavao mijenjati sloj baze podataka, radi primjerice promjene pružatelja usluge ili promjene načina pohranjivanja podataka, mijenjao bi se samo taj sloj i sloj perzistencije, pošto su jedan do drugoga. Sloj baze podataka je za sloj iznad njega vitalan jer ga poslužuje spremljenim podacima i ako se npr. način njihove pohrane promijeni, mijenja se i njihov način dohvaćanja u sloju perzistencije. Dakle, ako se jedna smislena cjelina uredno zaokruži, za buduće planove projekt temeljen na ovom uzorku može biti izuzetno fleksibilan i izazov će predstavljati mijenjanje samo onog sloja koji je obuhvaćen tom promjenom i veći ili manji dio susjednog s gornje strane [12].

U obliku tablice najlakše je predstaviti uloge i odgovornosti svakog od slojeva. Dakako, ovo predstavlja samo jedan način upotrebe ovog uzorka jer, kao što sam spomenuo ranije u poglavlju, broj slojeva i njihova nomenklatura nije ničime ograničena [9].

Tablica 2: Primjer mogućih slojeva u slojevitoj arhitekturi

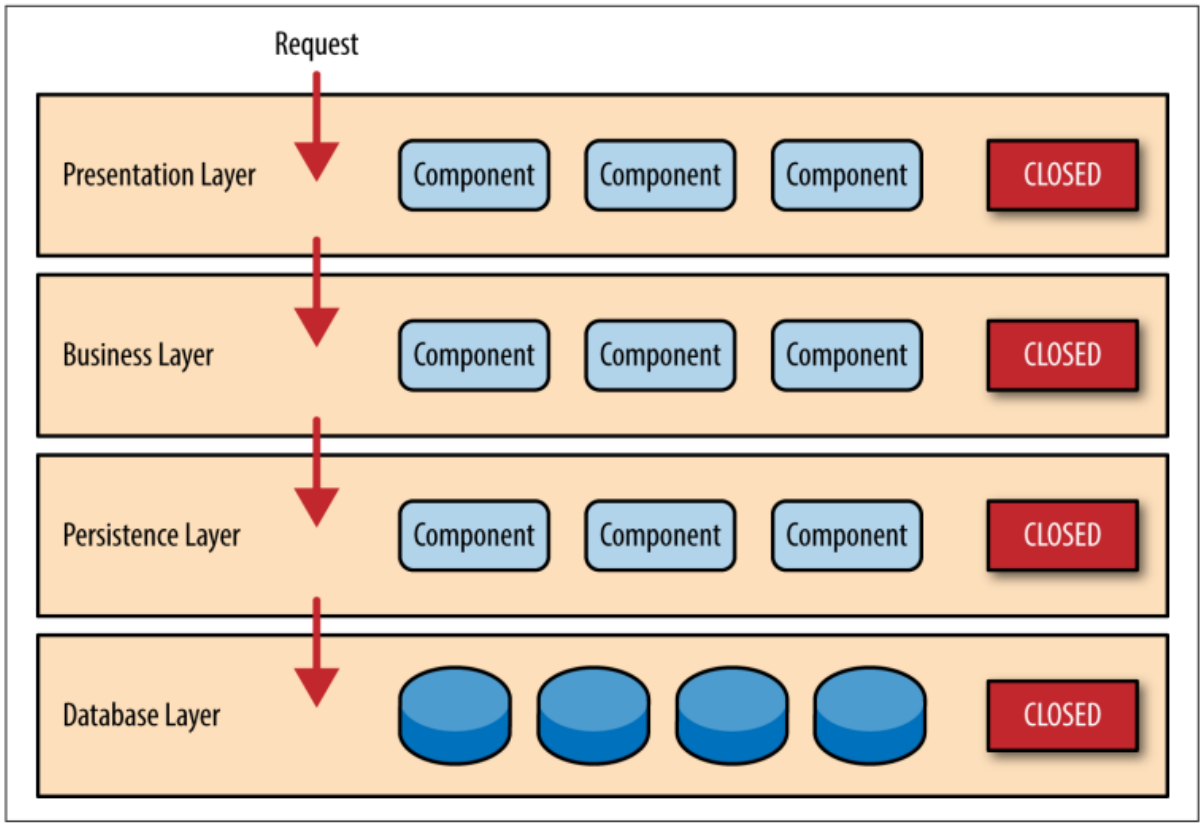
Ime sloja / opis sloja	**Odgovornosti i uloge**
Sloj prikaza (eng. *presentation*)	Predstavlja korisniku sve elemente u korisničkom sučelju aplikacije, pruža mu mogućnost interakcije i pokriva sve ono što se ispod njega nalazi, a to su ostali slojevi. Prosljeđuje ostalim slojevima korisnikove naredbe.
Poslovni (eng. *business*)	Posjeduje poslovnu logiku, mozak je aplikacije.
Sloj perzistencije (eng. *persistence*)	Služi za rukovanje funkcijama poput objektno relacijskog mapiranja.
Sloj baze podataka (eng. *database*)	Sama baza podataka, može ih pohranjivati u tablicama i na zahtjev ih ustupa sloju perzistencije i aplikaciji koje može ali i ne mora biti dio (unutarnja ili vanjska baza podataka).

(Izvor: Richards, 2015)

Ovako građene sustave puno je jednostavnije razvijati, testirati, kontrolirati i održavati zbog savršeno definiranog i ograničenog djelokruga komponenti [9].

U idućoj slici prikazan je još jednom model jednog sustava pod okriljem slojevite arhitekture, samo što su u ovom slučaju slojevi označeni s oznakom *closed*, što predstavlja zatvorenost pojedinog sloja. To je koncept kojim se prikazuje kako korisnički zahtjev polazi uvijek od sloja na vrhu te iako mu je ciljno odredište sloj koji se nalazi skroz na dnu ove ljestvice, mora proći i kroz sve ostale koji se nalaze na njegovom putu, a to su svi između prvoga i zadnjega. Stoga, u danom primjeru zahtjev se uvijek prima u sloju prikaza, a da bi došao do sloja perzistencije, potrebno je proći poslovni sloj.

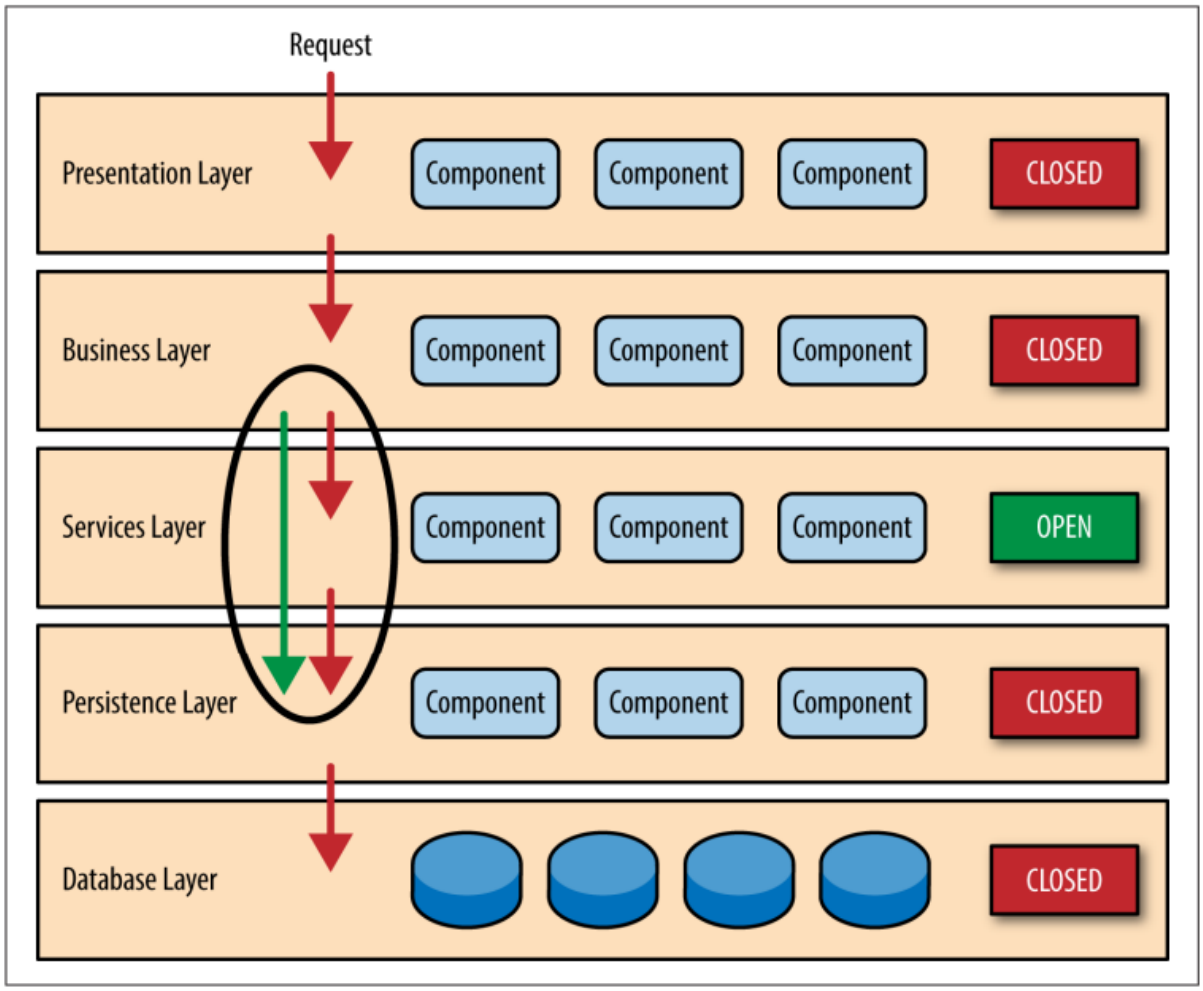
Naravno, pritom se postavlja pitanje: ne bi li onda bilo jednostavnije omogućiti svakom sloju izravan pristup svakog drugom, za slučaj kada to bude potrebno, zbog uštede vremena? Moglo bi se, ali onda bi se kršio koncept izoliranosti slojeva koji govori kako izmjena jednog sloja ne utječe na sve druge. Ako bi se navedena izravna veza omogućila, promjene napravljene unutar sloja perzistencije utjecale bi na sve one koji su s njime povezani. Takav pristup izazvao bi vrlo čvrsto i na kriv način povezanu aplikaciju čiji slojevi imaju mnogo međuovisnosti, a nastajali bi uz to i neki neželjeni rezultati koji nisu lako ispravljivi [9]. Ako se već mora cijeli sloj mijenjati, neka se mijenja samo taj i djelomično onaj koji je s njime povezani. Pridržavanjem koncepta izoliranosti postiže se urednost u strukturi sustava, zna se točno gdje je što i što je namijenjeno čemu.



Slika 8: Tok zahtjeva i svojstva slojeva (Izvor: Richards, 2015)

Unatoč tome što se pridržavanjem koncepta izoliranosti nastoji održati red unutar dijelova aplikacije, nekad ipak uistinu postoji scenarij u kojem se ne može naći bolje rješenje, nego da se kreira sloj koji je po svojim svojstvima i ulozi drukčiji od ostalih.

Takav primjer bio bi kada bismo htjeli napraviti jedan novi sloj koji bi sadržavao uslužne komponente koje bi koristile samo komponente u poslovnom sloju. Pošto želimo da samo poslovni sloj ima pristup tom novopečenom sloju, pozicionirali bismo ga upravo ispod njega. No, tu se stvara problem da bi zahtjev koji ide sve do posljednjeg sloja morao uvijek prelaziti taj uslužni sloj, što nije najbolje rješenje jer se neke stvari uslužnog sloja ne bi trebale ticati. Prema izvoru, to je već prastari i dobro poznat problem s ovim uzorkom, koji se rješava na način da se napravi sloj koji bi u odnosu na ostale bio otvorenog karaktera te kojeg bi se moglo preskakati kad je to potrebno, da bi se došlo do onih potrebnih. Navedeno je prikazano na slici broj 9 [9].



Slika 9: Dodavanje otvorenog sloja (Izvor: Richards, 2015)

Korištenjem otvorenih i zatvorenih slojeva stvaramo jasniju vezu između arhitekturnih slojeva i tokova zahtjeva kroz cjelokupni sustav, ali pružamo dizajnerima i programerima potreban uvid u različite ograničenosti pristupa slojevima unutar arhitekture. Kada se odlučimo upotrijebiti otvoreni sloj, važno ga je, kao i one zatvorene, propisno dokumentirati i jasno istaknuti te znati objasniti zašto je upravo on takav kakav jest. U suprotnome odmažemo stvaranju aplikacija koje su lagane za testirati, održavati i isporučiti [9].

Kao što sam u uvodu ove cjeline spomenuo, mnogi će programeri u početku razvoja prirodno težiti primjeni upravo ovog uzorka jer predstavlja način koji je općenito lako primjenjiv na mnoge aplikacije, pa će čak i ako ne znaju da se zove slojevit uzorak, odlučiti ga koristiti. Također, ako nismo sigurni koji uzorak najviše odgovara našoj ideji, velika je vjerojatnost da će to biti upravo ovaj, a čak i ako na kraju ustanovimo da to nije tako, svejedno može poslužiti kao odlična početna točka dok ne shvatimo koji je to drugi uzorak koji ima ono što nama treba, a da ga slojevita arhitektura ne može pružiti [9].

Kod slojevite arhitekture postoji također i jedna zamka na koju se mora pripaziti. Naime, zbog postojanja slojeva, postojat će uvijek slučajeva pri izgradnji aplikacije u kojima se s gornjeg sloja (sloj prikaza) moraju dohvatiti podaci nad kojima se neće vršiti apsolutno nikakve dodatne radnje poput raspoređivanja, mijenjanja ili nečeg što bi zahtijevalo intervenciju drugih slojeva kao što je primjerice logički. Proces dohvaćanja takvih podataka je takav da se on „provuče“ kroz sve one nepotrebne slojeve koji ga samo prosljeđuju dalje, a kada dođe do onog koji je zaista potreban, pokupe se potrebne „stvari“ i krene se natrag prema odredišnom sloju, bez nekog logičkog doticanja s ostalima.

Zanimljiva je činjenica da će većina, ako ne i svaka aplikacija imati nekolicinu ovakvih slučajeva, i to ne mora nužno biti nedostatak, ukoliko njihov broj ostane u razumnim granicama. Izvor navodi kako je potrebno broj takvih slučajeva pomno analizirati i usporediti ih s onima koji ne spadaju u tu kategoriju te se pri tome ne bi smio dobiti omjer koji nije ni blizu onog optimalnog, a to je 1:4. Dakle, u redu bi bilo kada bi otprilike svaki četvrti zahtjev koji se u takvom sustavu obrađuje bio samo *pass-through* ili propušteni. Ako takvih ima previše, moguće je da ovaj uzorak baš najbolje ne odgovara konkretnom slučaju ili bi se trebalo napraviti veći broj otvorenih slučajeva, ali to bi opet značilo da će se promjene napravljene u takvim aplikacijama mnogo teže moći pratiti i usklađivati radi nepridržavanja glavnom konceptu, izolaciji slojeva [9].

Iako i to ima svoje prednosti, činjenica da ovaj arhitekturni uzorak voli poprimati mnoga obilježja i naginjati prema monolitnoj arhitekturi mnoge će upućenije programere odvratiti od ideje njegovog primjenjivanja. To se ponajprije odnosi na fleksibilnost prema adaptaciji novijim tehnologijama, skalabilnost, lakoću isporuke i pouzdanost, osobine o kojima će više riječi biti u idućem poglavlju. Riječ „monolit“ dolazi od grčkih riječi *monos* (zaseban) i *lithos* (kamen), što u kontekstu softvera predstavlja izgrađenost u jednoj velikoj, teško izmjenjivoj cjelini [13]. Ovdje su moduli programa međusobno usko povezani i svi dijelovi aplikacije moraju biti prisutni kako bi se ona mogla pokrenuti.

Sve u svemu, vidimo kako je uistinu izuzetno važno poznavati arhitekturne uzorke do njihove srži i kako u praksi nije dovoljno dobro samo znati napraviti aplikaciju na temelju uzorka, već se ona mora napraviti na pravi način, koristeći sve dane savjete i upute koji se tiču pojedinog uzorka. Od početka se sve mora planski i savjesno slagati, kako trenutak u kojem shvatimo da smo mogli neke stvari drukčije izvesti ne bi došao prekasno.

3.2.2. Analiza slojevite arhitekture

Tablica 3: Opis svojstava arhitekturnih uzoraka u slučaju slojevite arhitekture

Svojstvo	Opis iskustva	Ocjena
Agilnost	lako se promjena unutar aplikacije može izolirati unutar slojeva, zbog težnje prema monolitnim svojstvima nije naročito jednostavno prilagođavati ju konstantno mijenjajućoj okolini.	Niska
Lakoća isporuke	Ako se napravi barem jedna promjena unutar samo jedne komponente, vrlo je lako moguće da će se aplikacija morati ponovno isporučiti u cjelini. Iz tog razloga potrebno je pomno planirati vrijeme isporuke, pogotovo kod složenijih aplikacija.	Niska
Provjerljivost	Ovo je najveća dobit od korištenja ovog uzorka. Slojevi omogućuju programeru da ih jednostavno može izolirano testirati, a ostali slojevi pritom ne moraju biti originalni, već ih se može prilagoditi konkretnoj potrebi.	Visoka
Učinak	Zbog opisanog načina izvođenja jednog procesa unutar sustava, pri čemu se potencijalno mora prolaziti kroz neke slojeve kroz koje se ne bi trebalo, a samo zato što nisu svi izravno povezani, u većini slučajeva performans se degradira.	Niska
Skalabilnost	Nije naročito pametno koristiti ovaj arhitekturni uzorak ukoliko se očekuje da bi tijekom životnog vijeka mogla postojati potreba za stabilnim funkcioniranjem pri smanjenim resursima, povećanoj količini procesnih zahtjeva te izvođenju promjena.	Niska
Lakoća razvoja	Zbog toga što bi se većini poslovnih aplikacija mogao nadjenuti ovaj uzorak i zato što ga je lagano razumjeti i primijeniti, implementacija programa s ovim uzorkom ne predstavlja preveliki izazov.	Visoka

(Izvor: Richards, 2015)

4. Razvoj aplikacije

Pošto se tematika ovog rada vrti oko primjene određenih koncepata programiranja na jednu korisnu aplikaciju u praksi, drugu polovicu ovog rada posvetit ću samo njoj. Ovdje će biti riječi o njezinoj strukturi, procesu razvoja, korištenim alatima, vremenskim okvirima izrade, namjeni aplikacije i njezinom načinu korištenja. Došao sam do zaključka da bi bilo dobro na brzinu proći kroz sadržaj svakog sloja, a potom, pošto se ovdje radi o slojevitom arhitekturnom uzorku, proći kroz sve funkcionalnosti koje aplikacija nudi kako bi se prikazao tok zahtjeva od gornjeg do donjeg sloja. Na taj način može se lako nadovezati na sve teorijski objašnjeno u poglavlju 2 i naročito poglavlju 3.

4.1. O aplikaciji

U ovom poglavlju dotaknut ćemo se svih općenitosti aplikacije Zvučne obavijesti. Ova cjelina je zamišljena da posluži kao uvod u dubinsku analizu koja slijedi nakon iste.

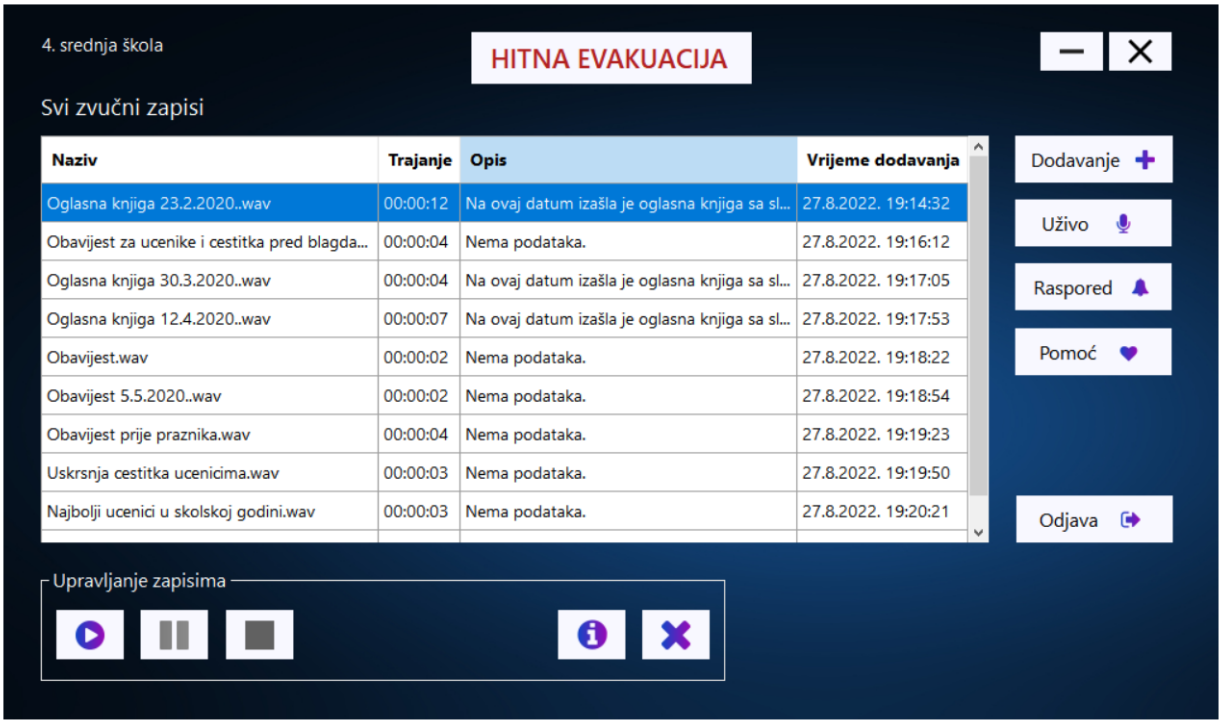
4.1.1. Ideja, primjena i preduvjeti aplikacije

Aplikacija „Zvučne obavijesti“ namijenjena je za školske ustanove i njihove potrebe u organizaciji nastave kada se ona izvodi. Zamišljeno je da ju koriste zaposlenici u školi kada će im biti potrebna, a primarno se to odnosi na satničare, ravnatelje ili čak profesore i učitelje te ustanove. Namijenjena je za Windows operativne sustave, bilo na stolnim računalima, bilo na prijenosnim računalima. Da bi se mogla efikasno koristiti, preduvjeti su da je škola opremljena zvučnicima koji se mogu žično ili bežično spojiti s računalom te na taj način reproducirati zvukove koji se na njemu izvode u željenom trenutku. Dvije funkcionalnosti aplikacije zahtijevaju opremljenost ulaznom jedinicom za primanje zvuka, točnije nekom vrstom mikrofona. Osim toga, bilo bi dobro da je korisnik ove aplikacije informatički pismen barem što se tiče nekih osnova kod upravljanja računalom.

Aplikacija može pomoći reprodukcijom za početak i završetak sata nastave, omogućavanjem govora uživo preko mikrofona, reprodukcijom zvuka za hitnu evakuaciju, računanjem svih termina satova nakon unosa potrebnih podataka te unošenjem i pospremanjem zvučnih zapisa poput obavijesti oglasne knjige školske ustanove i tome slično.

4.1.2. Opis sučelja

Jedan od glavnih ciljeva ovog završnog rada bio je izraditi aplikaciju koja će biti intuitivna i prijateljska za korištenje, bez obzira na životnu dob, zanimanje ili spol korisnika. Da bi se to ostvarilo, neophodno je da ona posjeduje glavno sučelje koje se sastoji od elemenata koji su razvrstani smisleno i koje je ugodno za oko. Na sljedećoj slici prikazano je sučelje na glavnom prozoru aplikacije, nakon registracije ili prijave korisnika.

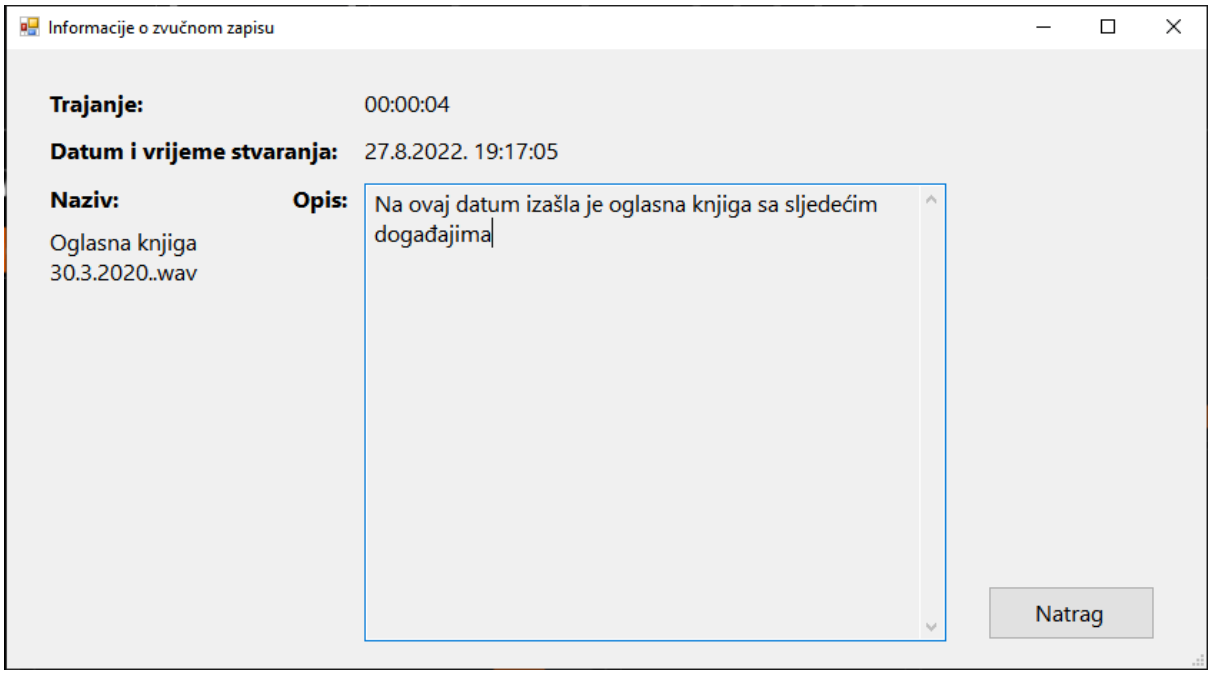


Slika 10: Snimka zaslona glavne forme aplikacije [autorski rad]

Krenimo od natpisa u gornjem lijevom uglu prozora gdje se može iščitati naziv škole koji je unesen prilikom prve registracije, desno od kojeg je jasno i velikim slovima označen gumb za reprodukciju zvuka za hitnu evakuaciju. Ispod njih je tablica u kojoj se nalaze svi uneseni zvučni zapisi za jednu školsku ustanovu, sa stupcima koji daju podatke o nazivu, trajanju, unesenom opisu i vremenu dodavanja tog zvučnog zapisa. S desne strane tablice mogu se pronaći gumbi koji redom omogućavaju sljedeće funkcionalnosti u novim prozorima:

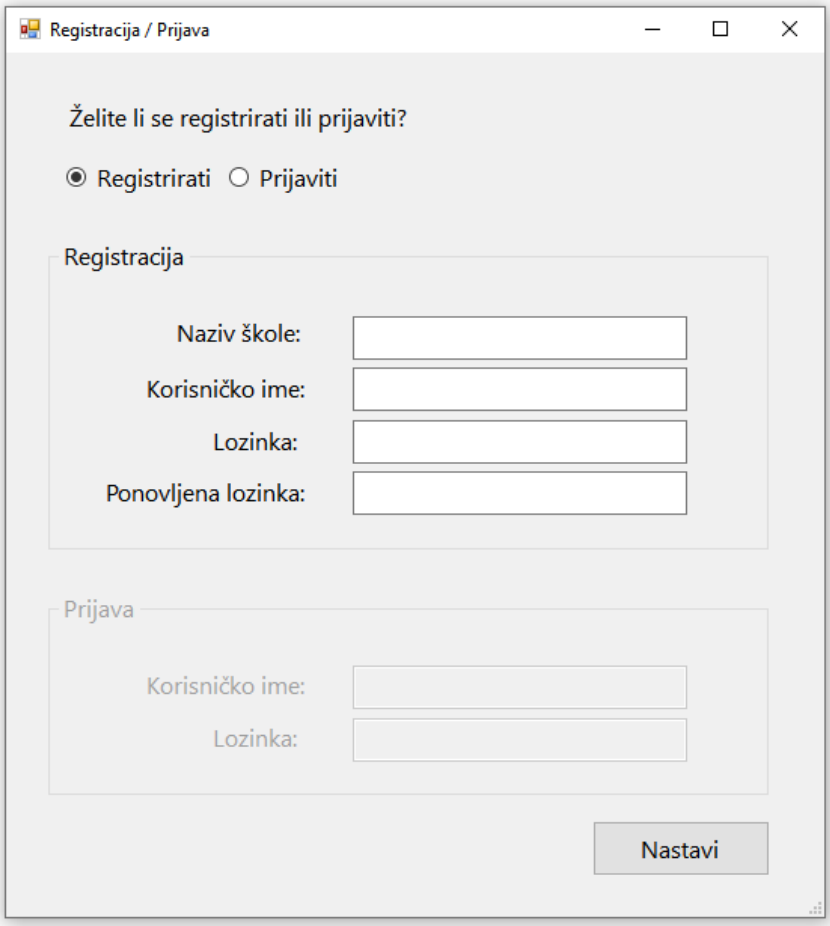
- Dodavanje novog zvučnog zapisa na način da se uveze postojeći ili kreira novi
- Prijenos i reprodukcija govora uživo
- Kreiranje rasporeda nastavnih sati
- Prikaz informacija koje mogu biti od pomoći ukoliko se korisnik ne snalazi
- Odjava trenutno prijavljenog korisnika iz sustava

Na samom dnu prozora vidljiva je grupirana skupina gumbi koji su zaduženi za upravljanje odabranim zvučnim zapisom iz tablice, a dozvoljava se njegova reprodukcija, pauziranje, zaustavljanje, prikaz svih informacija u novom prozoru, te brisanje iz baze podataka kao i sa lokalne pohrane. Glavna forma je ujedno jedina koja se može pomicati po zaslonu ekrana klikom na prazni prostor i držanjem lijeve tipke miša.



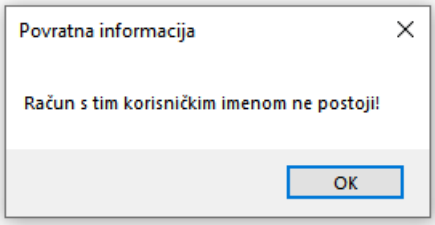
Slika 11: Snimka zaslona forme InfoForm.cs [autorski rad]

U gornjem desnom kutu glavnog prozora nalaze se prilagođeni gumbi za njegovo minimiziranje i zatvaranje. Ukoliko još nije dodan nijedan zapis, u donjem desnom kutu se pojavi žuti natpis „Molimo dodajte svoj prvi zvučni zapis“. Valja napomenuti kako su na dosta gumbi dodane ikonice preuzete s https://freeicons.io/ koje služe za lakše raspoznavanje njihovih funkcija. Iza svih tih elemenata se nalazi tamna, neupadljiva pozadina koja je u dovoljno velikom omjeru kontrasta s ostatkom prozora. Na svim elementima korisničkog sučelja ovog i svih ostalih prozora koristi se font *Segoe UI* različitih veličina, ovisno o potrebi.



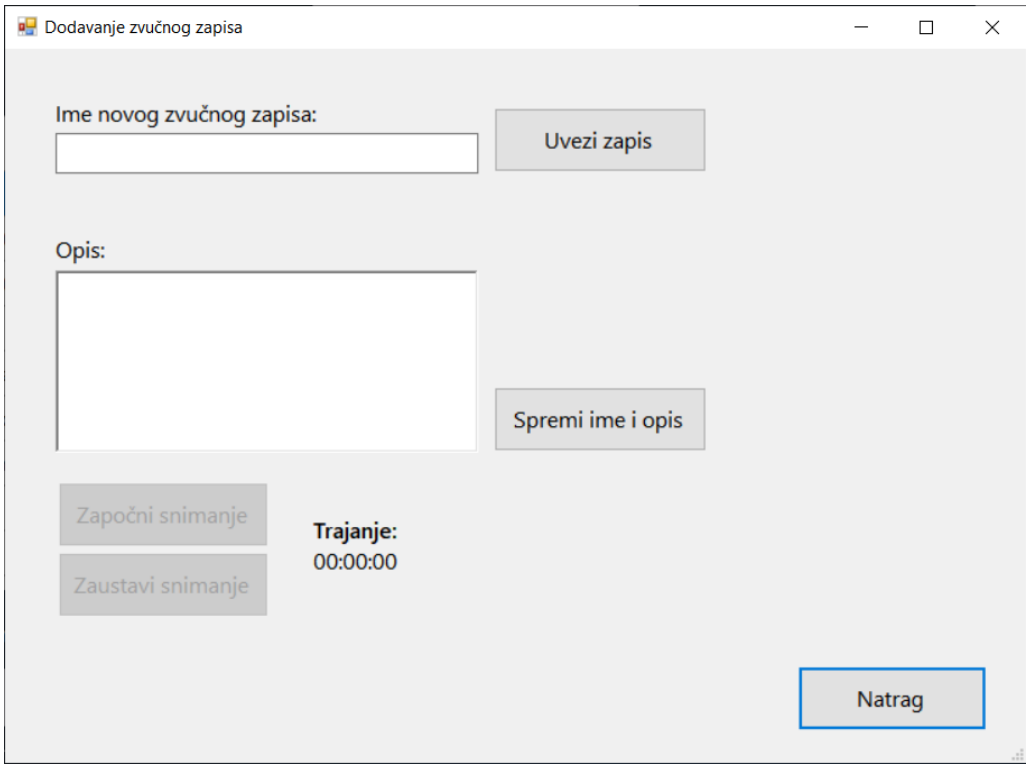
Slika 12: Snimka zaslona forme RegLoginForm.cs [autorski rad]

Na slici iznad vidljiva je forma gdje se korisnici registriraju ili prijavljuju. Odabire se jedno ili drugo, a ovisno o tome omogućuju se grupirani elementi (eng. *GroupBox*) registracije ili prijave u aplikaciju. Aplikacija je napravljena tako da je jedino glavni prozor iz kojeg se račvaju sve funkcionalnosti posebnog dizajna, dok su ostali koji uključuju zasebne funkcionalnosti i koji su jednostavniji ostavljeni pri zadanom dizajnu *Windows Forms* biblioteke. Kada korisnik unese odgovarajuće podatke i klikne na gumb „Nastavi“ slijedi provjera unesenih podataka o čemu će riječi biti kasnije i prosljeđivanje na glavnu formu. Naravno, kod sporednih formi postoji još mogućnost pojave popratnih prozorčića s porukama (eng. *MessageBox*) koji su izgleda kao na slici broj 13.



Slika 13: Slika zaslona prozorčića s povratnom informacijom [autorski rad]

Klikom na gumb „Hitna evakuacija“ na glavnom prozoru otvara se upitnik u kojem korisnik potvrđuje njegovu reprodukciju. Klikom na „Dodavanje“ otvara se novi prozor u kojem se nudi funkcionalnost dodavanja novog zvučnog zapisa u bazu podataka.

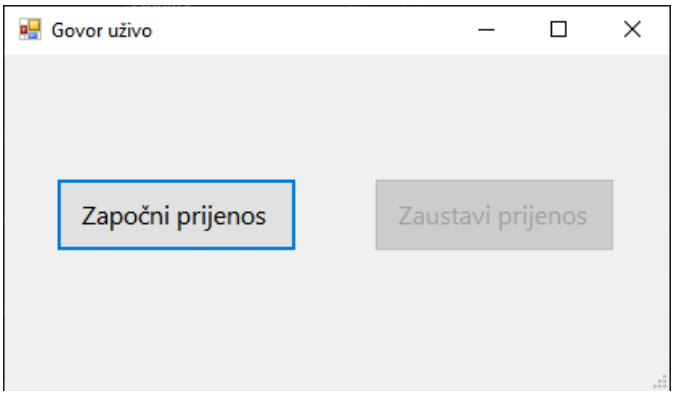


Slika 14: Snimka zaslona forme AddNewRecordingForm.cs [autorski rad]

Na tom prozoru moguće je unijeti novi zvučni zapis na dva načina, snimanjem vlastitog govora ili uvozom već postojećeg zvučnog zapisa, formata .mp3 ili .wav. Neće se dopustiti imenovanje zapisa posebnim hrvatskih slovima, niti bilo kakvim znakovima koji bi zadavali probleme prilikom stvaranja datoteke na trajnoj memoriji računala. Uz naziv zapisa dodati se može, ali i nema mora, pripadajući opis. Ta dva podatka, uz automatsku popunu datuma i vremena stvaranja i trajanja kasnije se prikažu u odgovarajućem retku na glavnom prozoru aplikacije. Kada se unese željeno ime i opis, klikne se na „Spremi ime i opis“, te se ta dva polja

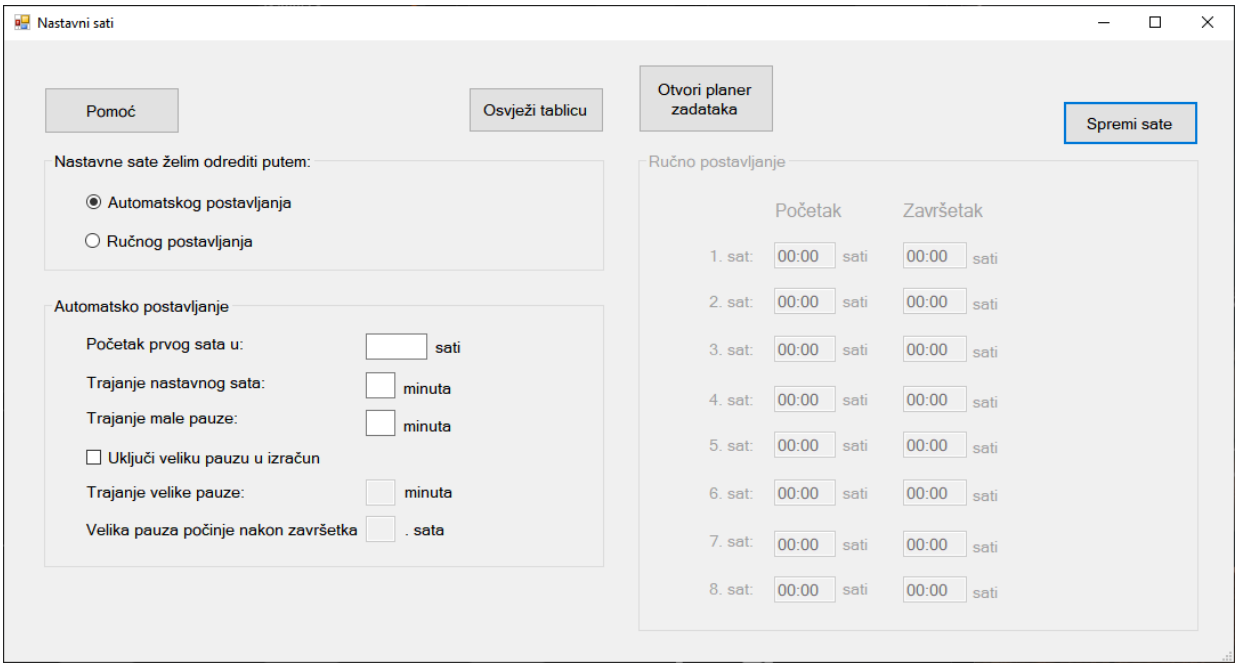
zajedno s gumbom „Uvezi zapis“ onemoguće, a omoguće se gumbi za početak i završetak snimanja. Kada se započne sa snimanjem, automatski započne i odbrojavanje na obližnjem natpisu, a najdulje se smije snimati do 10 minuta.

Prilikom snimanja, ako se pokuša izaći iz forme prije nego što se taj proces zaustavi, pojavit će se prozorčić s porukom da to nije moguće napraviti. Također, postoje upozorenja koja se aktiviraju kod neispravno upisanog naziva i kod snimanja dulje od 10 minuta. Povratna informacija o uspjehu ili neuspjehu spremanja se ukaže nakon klika na „Spremi ime i opis“ ukoliko se radi o uvozu, a nakon klika na „Zaustavi snimanje“ u slučaju novog snimanja.



Slika 15: Snimka zaslona forme LiveStreamForm.cs [autorski rad]

Pošto idemo redoslijedom odozgo prema dolje, idući je na redu prozor za snimanje i prijenos uživo namijenjen isključivo za to. Vjerojatno najjednostavniji prozor u aplikaciji se sastoji samo od dva gumba – za početak i završetak prijenosa vlastitog govora. Kako to funkcionira biti će objašnjeno u idućoj cjelini. U slučaju da se prozor pokuša zatvoriti prije završetka snimanja, pojavit će se poruka da to nije moguće.



Slika 16: Snimka zaslona forme ClassHoursForm.cs [autorski rad]

Na slici 16 prikazan je prozor za određivanje termina nastavnih sati. Oni se mogu izračunati automatski putem unesenih podataka ili ručno. Da bi se tablica na desnoj strani prozora automatski popunila, potrebno je unijeti minimalno 3 podatka – termin početka prvog nastavnog sata, trajanje jednog sata u minutama i trajanje kratkog odmora između tih sati u minutama. Kada se potom klikne na „Osvježi tablicu“, pojavit će se poruka da su podaci neispravno uneseni, ukoliko se pri unosi nisu poštivali formati u kojima sve mora biti uneseno. Za sate potrebno je unijeti vrijeme u obliku XX:YY, gdje je X jedinica za sat, a Y jedinica za minutu, dok za minute treba samo napisati broj minuta. Navedena pravila se mogu dobiti i klikom na gumb „Pomoć“ u gornjem lijevom kutu. Nakon ispravno unesenih podataka, tablica će se pritiskom na navedeni gumb osvježiti i sati nastave će biti uredno posloženi.

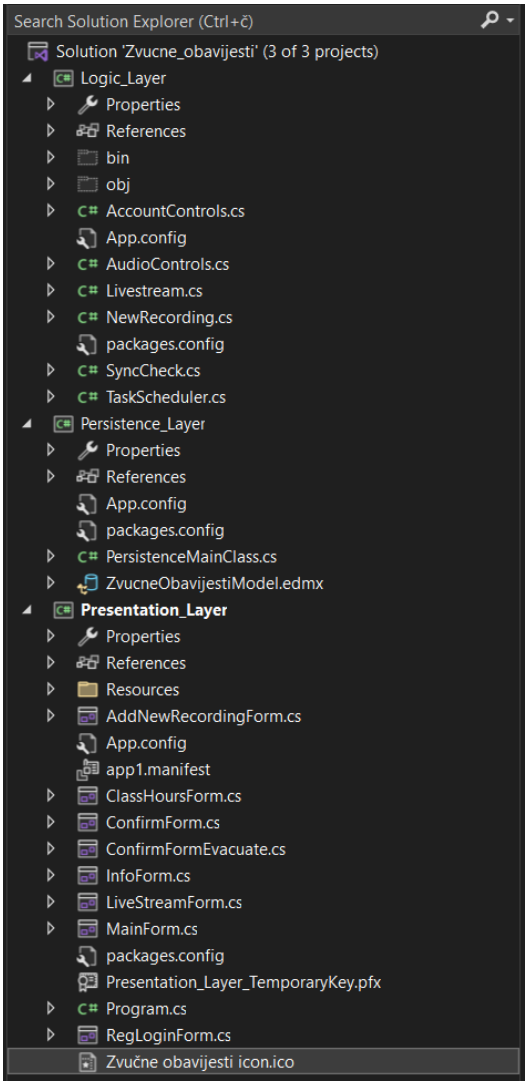
U tom slučaju jedini preostali korak je spremiti sate klikom na istoimeni gumb, a oni će biti stvoreni u obliku zadataka u *Windows Task Manageru.* Iz tog razloga postoji gumb koji služi za jednostavno pokretanje tog programa, a tamo se zadaci mogu pregledavati, brisati i dodatno mijenjati ukoliko je potrebno. Pojavit će se i prozorčić s porukom da je ta radnja uspješno izvršena.

Trenutno nije moguć unos više od 8 sati nastave, pošto sva nastava koja se izvodi poslije toga spada izvan okvira redovne nastave, a na vlastitom iskustvu znam da se zvona za početak i završetak nastave onda ni ne oglašavaju. Moguće je jedino unijeti manje, i to minimalno 1 sat, a svi termini poslije 23:00 također nisu dozvoljeni.

Pošto neke osnovne ili srednje škole imaju posebni oblik dugog odmora različitog trajanja jednom dnevno, postoji i mogućnost da se i on ubroji u taj izračun. Da bi se to postiglo, potrebno je označiti kvadratić „Uključi veliku pauzu u izračun“, pri čemu se omogućuje unos njegovog trajanja i rednog broja sata nakon kojeg on nastupa. Kad se to unese, na osvježenoj tablici pojavit će se plava linija koja vizualizira gdje je velika pauza smještena među ostalim satima u rasporedu nastave.

4.1.3. Projektna struktura

Kao što je već u nekoliko slučaja u radu bilo spomenuto, ovaj projekt je sastavljen od četiri sloja čiji nazivi su usko povezani s njihovim zadaćama, a to su sloj prikaza, logički sloj, sloj perzistencije i sloj baze podataka. Po svojem opsegu i sadržaju nisu istih veličina, a lančano su povezani referencama tim redom kojim su navedeni, bez preskakanja. Dakle, sloj prikaza „zna“ samo za logički, logički samo za sloj perzistencije, a potonji jedino za bazu podataka.



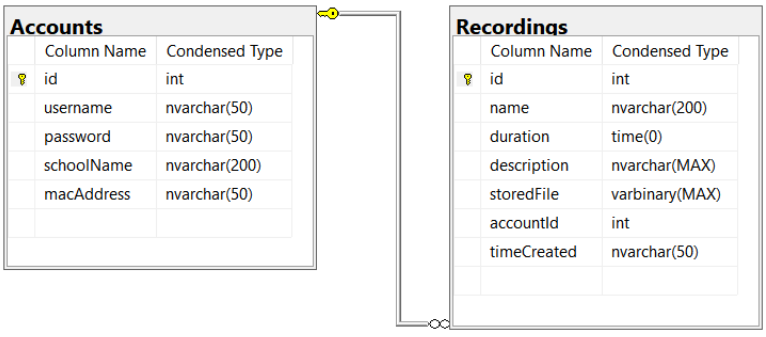
Slika 17: Snimka zaslona sadržaja slojeva [autorski rad]

U sloju baze podataka nalazi se jedna klasa u kojoj su smještene sve metode povezane s komuniciranjem s bazom podataka *SQL Server 2019.* Doduše te metode se radi praktičnosti na prethodnoj slici ne vide, ali će biti u nastavku rada prikazane, što vrijedi i za ostale slojeve. Osim toga, sloj je to na kojemu je izrađen model za bazu podataka, koji je zapravo set koncepata koji opisuju tamošnju podatkovnu strukturu, bez obzira na oblik u kojem su pohranjeni [14].

U logički sloj smještene su sve metode koje uključuju bilo kakvu logičku operaciju, bilo da se radi o radu s bibliotekom *NAudio*, radu s programom *Task Scheduler*, prilagođavanju i vraćanju podataka prema sloju prikaza, jednostavnom prijenosu zahtjeva na sloj baze podataka, provjeru sinkroniziranosti s bazom podataka i slično. Postoje iznimke gdje su neke logičke operacije unatoč tome smještene u sloj prikaza, što je bilo neophodno radi postizanja jednostavnosti i izbjegavanja redundancije u kodu.

Sloj prikaza je prema broju klasa i ostalog sadržaja najveći sloj u ovom projektu. Na njemu je smješteno sve ono što je izravno povezano s elementima korisničkog sučelja, tj. ono što korisnicima nudi interakciju s aplikacijom. Sakupljanje podataka iz tekstnih okvira, *CheckBoxeva*, gumbi, *RadioButtona* se obavlja u dotičnom sloju, a to se dalje prenosi na logički. Na prethodnoj slici vidljivo je također da tu postoji i jedna mapa, „Resources“, koja je namijenjena za pohranu 3 resursa *(*eng. *Embedded Resources)* koji se neće mijenjati, a dolaze zapakirani s ostatkom projekta kada se aplikacija instalira. To su zvučni zapisi za hitnu evakuaciju i za zvono nastavnog sata te pozadinska slika glavnog prozora aplikacije. Naziv tog sloja na slici 17 je, u odnosu na ostale slojeve, podebljan iz razloga što jedan sloj, odnosno projekt u ovom slučaju, mora biti definiran kao *Startup Project* od kojeg sve kreće. Pošto je sloj prikaza taj koji sadrži prozore s korisničkim sučeljem, a ujedno je i na vrhu hijerarhijske strukture ovog sustava, smisleno je da će on biti taj.

Baza podataka se sastoji od dvije tablice, „Accounts“ i „Recordings“. Jedina veza između njih dvije je ona koja govori da jedan račun može imati 0, 1 ili više zvučnih zapisa. Primarni ključ je u stupac „id“ u tablici za račune, dok je vanjski ključ stupac „accountId“ u tablici gdje su spremljeni podaci o zvučnim zapisima.



Slika 18: Prikaz tablica i njihove relacije [autorski rad]

4.2. Funkcionalnosti i tokovi procesa u aplikaciji

U ovoj cjelini biti će govora o konkretnom programskom kodu i uz konkretne isječke iz aplikacije biti će pokazan tok zahtjeva, tj. jednodretvenog procesa od njegovog začetka do svršetka iz svake funkcionalnosti koju ću obrađivati. Mnoge radnje koje obavlja ovaj sustav ne bi mogle biti odrađene bez korištenja biblioteke *NAudio*, koja se pokazala odličnom za primjenu u ovu svrhu, zbog čega se ona može smatrati njegovom žilom kucavicom.

4.2.1. Registracija i prijava

Kako je aplikacija namijenjena za korištenje u školama, logično je da svaka od njih treba moći pristupiti jedino svojim zvučnim zapisima. Autore tih zapisa ne bi bilo moguće razlikovati bez podataka o računima pomoću kojih su napravljeni, a ova funkcionalnost omogućuje upravo to, da se izradi račun i njime krenu obavljati sve ostale radnje.

Svaki puta kada se aplikacija pokrene, početna točka (eng. *Entry point*) je ista, *Main()* metoda u *Program.cs* klasi, odakle dalje sve ide preko glavne forme „MainForm.cs*“*

```
...
static void Main(string[] args)
{
        Application.EnableVisualStyles();
        Application.SetCompatibleTextRenderingDefault(false);

        if (args.Length != 0) Application.Run(new MainForm(args));
        else Application.Run(new MainForm());
}
...
```

Nakon toga se u *MainForm.cs* provjerava je li korisnik na tom računalu, s obzirom na njegovu *MAC (*eng. *Media Access Control)* adresu već prijavljen u aplikaciju, a ako nije, pokreće se forma „RegLoginForm.cs*“,* na kojoj se nakon korisnikovog unosa podaci dalje prosljeđuju logičkom sloju.

Kod registracije je važno da su unesena lozinka i ponovljena lozinka identične, da korisničko ime, lozinka i naziv škole nisu ostavljeni prazno jer u suprotnom se registracija obaviti ne može te se ispisuje odgovarajuća poruka korisniku, a nastavak procesa se onemogućuje.

```
...
if (txtPassword.Text != txtPassword2.Text)
```

```
{
        MessageBox.Show("Lozinke se ne podudaraju! Molimo ispravite.",
        MainForm.messageBoxTitle);
        return;
}
if (txtUserName.Text == "" || txtPassword.Text == "" || txtSchoolName.Text
== "")
{
        MessageBox.Show("Neispravan unos! Molimo ispravite.",
        MainForm.messageBoxTitle);
        return;
}
```

Osim toga, prekid radnje se događa i kada se, po primitku podataka iz logičkog sloja, kao rezultat dobije podatak tipa enumeracija „RegResult“ koji nije „RegSuccess“. To je u slučaju kada je unesen naziv škole koji već postoji ili već postoji uneseno korisničko ime ili se dogodila neka druga greška prilikom izvođenja tog zahtjeva.

```
public enum RegResult
{
        UsernameExists,
        SchoolExists,
        RegSuccess,
        Error
}
...
```

Rezultat tipa „RegResult“ se dobiva iz metode „RegisterUser()” u kojoj se iz baze podataka putem metode „GetData()” iz sloja baze podataka dohvaćaju sva korisnička imena i svi nazivi škola i pospremaju u dvije liste tipa *string* koje se potom koriste za provjeru postojanosti unesenih podataka u bazi podataka.

```
...
List<string> usernames;
List<string> schools;

usernames = PersistenceMainClass.GetData("username", "Accounts");
schools = PersistenceMainClass.GetData("schoolName", "Accounts");
...
```

Ukoliko sve provjere prođu uspješno, korisnikovi podaci se u pozivu metode „RegisterUser()“ iz sloja baze podataka prosljeđuju kao parametri, te se zapisuju u bazu podataka. Ujedno se putem metode „GetData()“ provjerava uspješnost izvođenja tog procesa i ako je sve u redu, dohvaća se korisnikova *MAC* adresa uz metodu „GetMACAddress()“ te se

i ona upisuje u odgovarajući redak i stupac tablice „Accounts“ baze podataka. Iako nije posve jedinstvena, *MAC* adresa je dobar i lako dohvatljiv identifikator za jedno računalo na internetu [15]. Sloju prikaza vraća se na kraju rezultat pokušaja registracije i u njemu se određuje koju povratnu informaciju treba u obliku *MessageBox* ispisati korisniku na ekran.

```
...
if (usernames.Contains(newUsername))
{
    currentUsername = newUsername;
    GetMACAddress();
    PersistenceMainClass.UpdateMACAddress(currentUsername,
currentMACAddress);
    return RegResult.RegSuccess;
} else
{
    return RegResult.Error;
}
...
```

Priča je slična i u slučaju prijave, a glavna razlika je što se tamo upisuju samo korisničko ime i lozinka jer se podrazumijeva da je registracija u neko vrijeme već bila obavljena. Ovdje se koristi metoda „LogUser()“ u logičkom sloju, koji kao rezultat vraća jednu vrijednost iz enumeracije „LogResult“. U odnosu na registraciju, tu je broj mogućih ishoda manji.

```
...
public enum LogResult
{
      UsernameDoesntExist,
      Incorrect,
      Correct
}
...
```

Nakon primitka rezultata, u sloju prikaza se s obzirom na njega ispisuje korisniku odgovarajuća poruka, a ujedno se vrijednost naziva škole pohranjuje u varijablu kako bi se kasnije mogla koristiti u glavnoj formi. Kao i u slučaju registracije, jedino kod uspješne prijave se prozor zatvara i proces se nastavlja na „MainForm.cs“.

```
...
switch (msg)
{
      case Logic_Layer.LogResult.UsernameDoesntExist:
```

```
            MessageBox.Show("Račun s tim korisničkim imenom ne postoji!",
            MainForm.messageBoxTitle);
            break;
      case Logic_Layer.LogResult.Incorrect:
            MessageBox.Show("Unesena lozinka nije točna!",
            MainForm.messageBoxTitle);
            break;
      case Logic_Layer.LogResult.Correct:
            pass = true;
            MessageBox.Show("Uspješno ste prijavljeni.",
            MainForm.messageBoxTitle);
            currentSchoolName =
            Logic_Layer.AccountControls.GetCurrentSchoolName(txtUserNameLog
            .Text);
            this.Close();
            break;
}
...
```

4.2.2. Sinkronizacija s bazom podataka

Funkcionalnost sinkronizacije s bazom podataka odvija se automatski neposredno nakon što se korisnik prijavi u aplikaciju, dok se kod registracije uvijek stvara novi račun s praznom listom zvučnih zapisa pa u tom kontekstu od nje koristi nema. Cilj ove funkcionalnosti je da se na vrijeme preuzmu svi zvučni zapisi postojani u bazi podataka za taj račun te da na taj način bude sve spremno kada se neki od njih želi reproducirati. To je najčešće slučaj kada se za isti račun promijeni računalo na kojem se aplikacija pokreće, na kojem ne postoje zapisi stvoreni na drugom računalu/ima. Oni koji nedostaju za taj račun, preuzmu se iz tipa *VARBINARY* i pretvore u niz bajtova koje je moguće koristiti. Korisnik pri tome ne sudjeluje, sve se odvija neprimjetno u pozadini.

Kao i obično, sve kreće od glavne forme koja poziva metodu „CheckSyncWithDB()“ u logičkom sloju, gdje se prvo dohvate podaci o prijavljenom računu, a najvažniji je „id“ računa uz pomoć kojeg se zatim u listu tipa *string* pohrane sva imena zvučnih zapisa iz tablice „Recordings“ u čijem stupcu „accountId“ postoji podatak točno te vrijednosti. Nakon toga slijedi dohvaćanje svih imena datoteka tipa WAV u jednu listu iz putanje "C:\Zvucne obavijesti" i kao što se može pretpostaviti, te dvije liste se međusobno usporede. Ono što ne postoji u listi datoteka sa diska „fileArray“, a postoji na listi sa baze podataka „recordingNames“, dodaje se u novu listu „missingRecordings“ u metodi „MakeMissingRecordingsList()“.

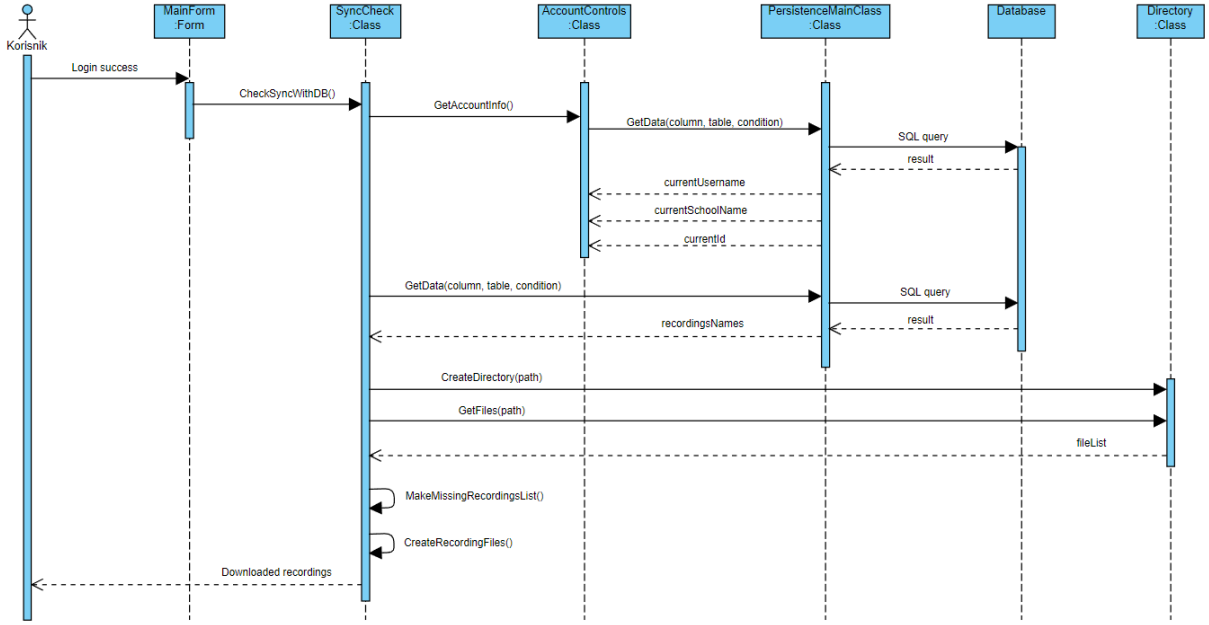
```
...
foreach (string uploadedRecording in recordingsNames)
{
      if (!fileArray.Contains(uploadedRecording))
      missingRecordings.Add(uploadedRecording);
}
...
```

Kako se nebi pretraživalo po imenima kojih može biti više identičnih za nazive zvučnih zapisa u bazi podataka, dohvaća se vrijednost stupca „id“ iz tablice „Recordings“ za svaki element u odgovarajućoj listi. Koristeći metodu „GetRecordingByteArray()“ iz sloja baze podataka dobiva se za svaki zvučni zapis njegov odgovarajući niz bajtova, prikazano u nastavku.

```
...
public static byte[] GetRecordingByteArray(int missingRecordingId)
{
      ClearConnectionPools();
      using (var cmd = new SqlCommand("SELECT storedFile FROM Recordings
      WHERE id=@ID", con))
      {
            con.Open();
            cmd.Parameters.AddWithValue("@ID", missingRecordingId);
            byte[] data = cmd.ExecuteScalar() as byte[];
            con.Close();
            return data;
      }
}
...
```

Kao zadnji korak se koristi metoda „DownloadMissingRecordings()“, u kojoj se u istoj petlji gdje se iz baze podataka dohvaća niz bajtova, za tu iteraciju napravi nova datoteka tipa WAV te se pospremi na C particiju u mapu „Zvucne obavijesti“.

```
...
foreach (string recordingId in missingRecordingsIds)
{
      byte[] result =
      PersistenceMainClass.GetRecordingByteArray(System.Int32.Parse(recordi
      ngId));
      File.WriteAllBytes(@"C:\Zvucne obavijesti\" + missingRecordings[y],
      result);
      y++;}
```



Slika 19: Dijagram slijeda za proces sinkronizacije s bazom podataka [autorski rad]

4.2.3. Hitna evakuacija

Ovo je vrlo jednostavan proces, a ipak se zbog njegove važnosti broji kao jedna funkcionalnost u aplikaciji. Iako se radi o zvučnom zapisu, ovom prilikom odlučio sam koristiti ugrađenu sistemsku biblioteku *SoundPlayer*, radi skraćivanja procesa. U njemu se prvo provjerava koji je tekst napisan na gumbu „Hitna evakuacija“. Ako to nije „STOP“, prikazati će se prozorčić na kojem korisnik mora potvrditi da je uistinu želio pokrenuti reprodukciju tog zvuka kako bi se izbjegle neugodnosti. Nakon potvrđivanja odabira, iz ugrađenih projektnih resursa se u varijablu učita „Alarm_1.wav“, a njega se uz pomoć *SoundPlayera* može početi i zaustaviti reproducirati. Pri tome se adekvatno mijenja tekst na tom gumbu, a cijeli proces se može prikazati u jednom isječku programskog koda, koji zapravo prikazuje *EventHandler* za klik na gumb „btnEvacuate“.

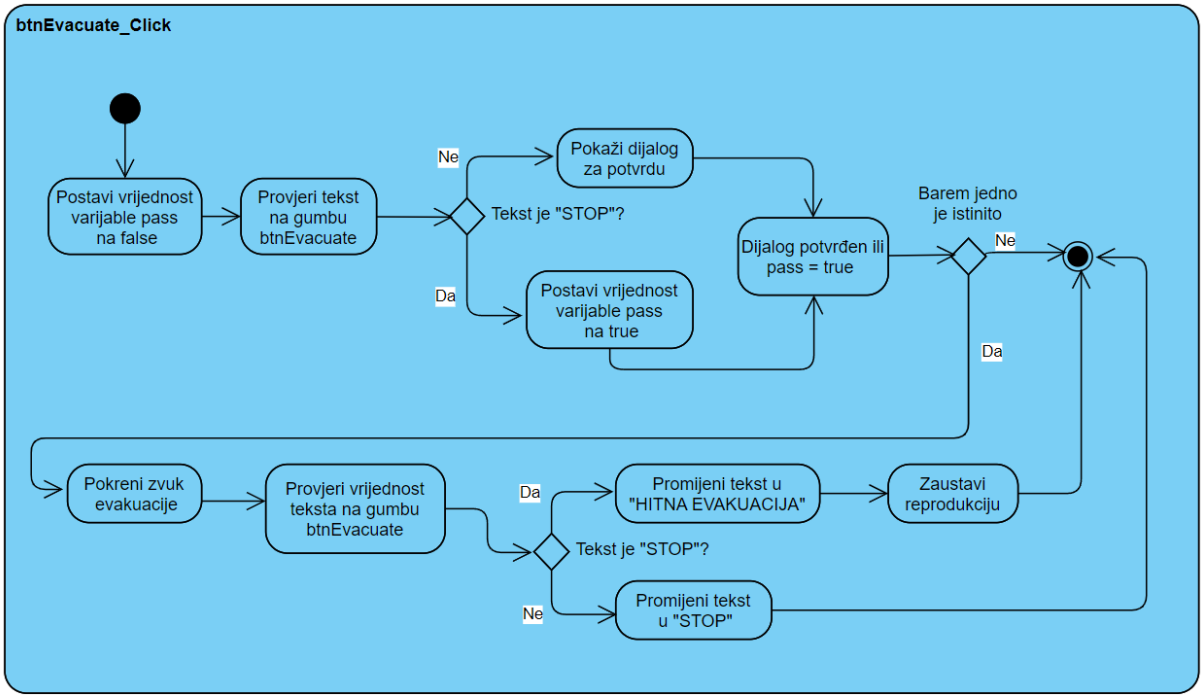
```
...
private void btnEvacuate_Click(object sender, EventArgs e)
{
	bool pass = false;

	if (btnEvacuate.Text != "STOP")
		confirmFormEvacuate.ShowDialog();
	else pass = true;

	if (confirmFormEvacuate.DialogResult == DialogResult.Yes || pass)
	{
```

```
            var evacSound = Zvucne_obavijesti.Properties.Resources.Alarm_1;
            SoundPlayer audio = new SoundPlayer(evacSound);
            audio.Play();

            if (btnEvacuate.Text == "STOP")
            {
                btnEvacuate.Text = "HITNA EVAKUACIJA";
                audio.Stop();
            }
            else
            {
                btnEvacuate.Text = "STOP";
            }
        }
}
...
```



Slika 20: Dijagram aktivnosti reprodukcije zvuka za hitnu evakuaciju [autorski rad]

4.2.4. Dodavanje zvučnog zapisa

Kao i u većini sličnih aplikacija gdje postoji opcija unošenja neke vrste medija, u ovoj korisnik može odabrati želi li u tom trenutku izraditi novi ili unijeti postojeći. Zbog toga, a i zbog njene složenosti ću ovu funkcionalnost objasniti u dvije cjeline. Oba načina se mogu izvesti unutar istog prozora, forme „AddNewRecordingForm.cs*“.*

4.2.4.1. Uvoz zvučnog zapisa

Da bi se uveo zvučni zapis, potrebno je kada se otvori *FileDialog* sa računala pronaći i odabrati datoteku tipa .wav ili .mp3. Svi drugi tipovi datoteka u tom dijaloškom okviru automatski neće biti vidljivi niti mogući za odabrati, a zapise je moguće uvesti jedan po jedan.

```
...
OpenFileDialog fileDialog = new OpenFileDialog();
fileDialog.Filter = "Wav, mp3 | *.wav;*.mp3";
fileDialog.FilterIndex = 1;
fileDialog.Multiselect = false;
...
```

Nakon odabira, u varijable tipa *var* i *string* se spremaju podaci o veličini, odnosno putanji datoteke, iz koje se izuzima zadnji dio koji predstavlja njezino ime. To ime će biti iskorišteno da se popuni odgovarajući tekstni okvir (eng. *TextBox),* ali samo u slučaju kad veličina u megabajtima nije veća od 50 za .mp3, odnosno 250 za .wav, u čijem slučaju se izbacuje povratna informacija u prozorčiću s porukom. Razlog tome je što je *WAV* format *lossless*, tj. ne radi kompresiju nad originalnim analognim zvučnim zapisom i zbog toga nudi mnogo kvalitetniji zvuk, ali zauzima i mnogo više prostora. MP3 je, s druge strane, primjer *lossy* formata, a upravo ta dva su najčešće korištena formata audiozapisa u svijetu [16].

Kad smo već kod toga, vrijedi spomenuti da se iz dijaloškog okvira može dobiti podatak o veličini datoteke samo u bajtovima, pa sam odlučio iskoristiti biblioteku *ByteSize*
koja uvelike olakšava posao i dovoljno je da se samo specificira iz koje jedinice se želi pretvoriti veličina datoteke u koju. U ovom scenariju je to iz bajtova u megabajte.

```
...
var sizeBytes = new FileInfo(fileDialog.FileName).Length;
var sizeConverted = ByteSize.FromBytes(sizeBytes);
...
```

Klikom na „Spremi ime i opis“ ovdje ova funkcionalnost poprima dva različita toka. U slučaju da se radi o uvozu, podaci o unesenom opisu i nazivu se prosljeđuju logičkom sloju i to metodi „ImportFile()“*,* u kojemu se radi konverzija iz MP3 u WAV ukoliko je to potrebno, a inače se preskače taj korak. Tu se po prvi puta, otkad smo u funkcionalnostima, koristi biblioteka *NAudio* i to u svrhu realizacije tog pretvaranja. Pri tome novopečena datoteka postaje višestruko veća u smislu prostora na disku računala i ručno joj se mijenja ekstenzija. Sprema se u mapu „Zvucne obavijesti“ gdje su smješteni svi ostali zvučni zapisi, na C particiji. Spomenuta biblioteka može reproducirati samo zvučne zapise formata .wav, zbog toga se mora činiti konverzija ukoliko se odabere .mp3 datoteka.

```
...
if (extension != "wav")
{
        fullNewName = Path.Combine(projectPath, nameWithoutExt + ".wav");

        using         (MediaFoundationReader         mp3         =         new
        MediaFoundationReader(sFileName))
        {
                using              (WaveStream              pcm              =
                WaveFormatConversionStream.CreatePcmStream(mp3))
                {
                        WaveFileWriter.CreateWaveFile(fullNewName, pcm);
                }
        }
} else
{
        File.Copy(sFileName, fullNewName, true);
}
...
```

Baza podataka se u ovom procesu također ažurira s novim retkom u tablici „Recordings“, a jedino još nedostaje da se odredi vrijeme trajanja, što se isto tako obavlja pomoću te biblioteke, a ona to omogućava besprijekorno i vrati podatak tipa *TimeSpan.* Sada na red dolazi metoda „InsertValue()“ u sloju baze podataka, gdje se jednostavnom SQL naredbom uz naknadno dodavanje parametara izbjegavaju *SQL injection* opasnosti, gdje može doći do gubitaka podataka kada se zlonamjernim korisnicima da prilika da takve rupe u sustavu iskorištavaju na vješt način u svoju korist [17].

Još jedna metoda iz posljednje spomenutog sloja ovdje sudjeluje, „UploadToDB()“. Kao što ime sugerira, ona je zadužena samo za to da se niz bajtova dobivenih iz dohvaćene novostvorene datoteke iz mape „Zvucne obavijesti“ pretvori u tip *VARBINARY*, koji je pogodan za spremanje u tablicu baze podataka. To se učini pomoću sistemskih biblioteka *FileStream* i *BinaryReader* i potom pospremi u stupac „storedFile“ tablice „Recordings“.

```
...
string filePath = @"C:\Zvucne obavijesti\" + fileName;
byte[] file;
using (var stream = new FileStream(filePath, FileMode.Open, FileAccess.Read))
{
      using (var reader = new BinaryReader(stream))
      {
```

```
        file = reader.ReadBytes((int)stream.Length);
        }
}
...
```

Na samom kraju se još metodom logičkog sloja „CheckIfSaved()“ provjeri je li uspješno spremljeno i ovisno o rezultatu u novom malom prozoru se prikaže povratna informacija.

4.2.4.2. Snimanje zvučnog zapisa

Da bi se posve novi zvučni zapis mogao započeti snimati, potrebno je ispravno unijeti ime datoteke, objašnjeno u poglavlju 4.1.2. i kliknuti na „Spremi ime i opis“. Kada se on započne snimati, započinje otkucavati i sat u natpisu (eng. *Label*) s desne strane gumbi, a zatvaranje prozora se onemogućuje sve dok se snimanje ne dovrši. Cijela logika zadužena za proces snimanja se nalazi u metodi logičkog sloja „ToggleRecording()“. „SaveNameAndDesc()“ se koristi samo u slučaju snimanja, a ne i uvoza. Metoda je to zadužena za dohvaćanje svih datoteka iz mape „Zvucne obavijesti“ i za davanje dozvole za nastavak operacije ukoliko nijedna nema isti naziv kao nova.

Prije svega, potrebno je odabrati gdje će se na računalo spremiti nastala snimka. Mapa za to je kao i u svim drugim slučajevima ona s putanjom „C:\Zvucne obavijesti“. Napravi se varijabla tipa *var*, koja predstavlja novi *WaveInEvent*, uz kojeg je potrebna varijabla tipa WaveFileWriter, a ta u je početku vrijednosti *null.* Započinjanje od završavanja snimanja se razlikuje putem parametara koje ta metoda prima, oba su tipa *bool*, prvi je istinit kada se započinje, a drugi je kada se završava [18].

```
...
var outputFilePath = Path.Combine(outputFolder, chosenFileName);

writer = new WaveFileWriter(outputFilePath, waveIn.WaveFormat);
waveIn.StartRecording();
...
```

Sada je potreban i *handler* za stvoreni *DataAvailable event* koji se pokreće periodično nakon što se započne snimati. Kako se snima, tako se redom podaci zapisuju u već spremnu novostvorenu datoteku [18].

```
...
waveIn.DataAvailable += (s, a) =>
{
        writer.Write(a.Buffer, 0, a.BytesRecorded);
        if (writer.Position > waveIn.WaveFormat.AverageBytesPerSecond * 30)
```

```
        {
                waveIn.StopRecording();
        }
};
...
```

Kada se klikne na gumb za završetak snimanja, na isti način kao i za početak pokrene se napravljeni *handler* za taj događaj. Stvorene varijable se uredno počiste uz pomoć biblioteke, a dodano je još i konačno zapisivanje u bazu podataka preko već spomenute metode „InsertValue()“ uz dodavanje svih potrebnih parametara. Posljednji po redu parametar predstavlja trenutni datum i vrijeme stvaranja zapisa, koji se dobiva pomoću sistemske *DateTime* biblioteke preko metode „Now()“.

```
...
waveIn.RecordingStopped += (s, a) =>
{
        writer?.Dispose();
        writer = null;
        if (closing)
        {
                waveIn.Dispose();
        }


        if (once)
        {
                once = false;

                PersistenceMainClass.InsertValue(chosenFileName,
                description,      timer,      AccountControls.currentUserId,
                DateTime.Now.ToString());
        }
};
...
```

Identično prethodnom načinu, i ovdje se koristi „CheckIfSaved()“ da bi se naposlijetku znalo koja se povratna informacija treba korisniku ispisati na ekran. Da bi korisnik imao koristi bilo od jednog, bilo od drugog načina dodavanja zvučnog zapisa, mora se zatvoriti prozor u kojem se to obavlja, da bi se vratilo na glavnu formu gdje će se prikazati osvježena tablica i gdje se posljedično može pokrenuti novopečeni zvučni zapis.

4.2.5. Prijenos govora uživo

Ovdje se radi o prozoru s vrlo jednostavnim sučeljem, a ni u pozadini nije mnogo drugačija priča. Prilikom ovog procesa zvuk kojeg korisnik napravi nigdje se trajno ne sprema, kao u slučaju funkcionalnosti snimanja zapisa, već se posprema na privremenu memoriju računala dovoljno dugo da se može povratno reproducirati [19]. Sve započinje pozivom metode „OnStartRecordingClick“, a gumbi budu naizmjence omogućeni.

```
...
Logic_Layer.Livestream.OnStartRecordingClick();
btnStartStreaming.Enabled = false;
btnStopStreaming.Enabled = true;
...
```

Kada se taj dio uspješno prođe, dalje se sve odvija u logičkom sloju u klasi „Livestream“. Napravi se nova *WaveIn()* varijabla i za nju se veže *event handler RecorderOnDataAvailable*, a sav zvuk kojeg mikrofon pokupi posprema se u varijablu tipa *BufferedWaveProvider.* Uređaj za automatsku reprodukciju tog zvuka mora postojati i on je nova *WaveOut()* varijabla koja se zatim inicijalizira i pokrene i na taj način korisnik svoj govor može čuti uživo. Za ulazni uređaj ključno je pritom pokrenuti snimanje, a to se radi uz metodu *StartRecording()* klase *WaveInEvent*, koju smo susreli u prethodno objašnjenim funkcionalnostima.

```
...
public static void OnStartRecordingClick()
{
      recorder = new WaveIn();
      recorder.DataAvailable += RecorderOnDataAvailable;

      bufferedWaveProvider = new BufferedWaveProvider(recorder.WaveFormat);

      player = new WaveOut();
      player.Init(bufferedWaveProvider);

      player.Play();
      recorder.StartRecording();
}
...
```

4.2.6. Postavljanje rasporeda sati

Aplikacija za zvučne obavijesti školskih ustanova ne bi bila od dovoljne koristi kad se njome ne bi mogla izvesti zvonjava nastavnih sati. U slučaju ove funkcionalnosti, pošto je izravno povezana s elementima kojima korisnik upravlja, jedan velik dio posla odradi se u sloju prikaza, koji prerađene podatke u potrebnim oblicima predaje logičkom sloju. Preostali sloj, sloj baze podataka, ovdje je oslobođen svih dužnosti. Nema potrebe da se išta zapisuje u bazu podataka pošto se ovdje radi s ugrađenom aplikacijom *Windows* operativnog sustava *Task Scheduler*, kojom se može sve postići. Bilo da korisnik unosi sate ručno, bilo automatski, princip funkcioniranja je isti. Preko tabličnog prikaza sastavljenog od tekstnih okvira u grupi za ručno podešavanje preuzmu se podaci o počecima i završecima nastavnih sati i oni se, ako su uneseni u pravilnom obliku, prenesu u logički sloj. Glavna razlika je da se kod automatskog to mnogo brže i pouzdanije može izvesti.

Ono što se prvo napravi kod učitavanja forme „ClassHoursForm.cs“, jest da se pozove metoda „UpdateTextboxArrays()“ koja pokupi sve tekstne okvire u grupi za ručno podešavanje i oni se postave na vrijednost „00:00“. Kod automatskog popunjavanja potrebno je kliknuti na „Osvježi tablicu“, čime se provjere svi unosi i ako su u redu, krene se u izračun.

Svaki tekstni okvir (eng. *Textbox*) ima naziv s obzirom na redak i stupac u kojem se nalazi. Slovom B započinju svi koji se nalaze u stupcu početka, a slovom E završetka sata. Također, na 5. mjestu u nazivu se nalazi redni broj retka, što je ako se gleda po indeksima, broj 4. Cijela logika računanja počiva na tome da se lančano zbrajaju termini, što znači da će idući imati vrijednost uvijek jednaku prošlome, uvećanu za unesenu vrijednost. Neizbježno je znati i odmah na početku računanja unijeti termin početka prvog sata kako bi se mogla postići ta lančana reakcija. Kada se dobije početak nekog sata, dodaje mu se trajanje jednog nastavnog sata, a tome se pak dodaje trajanje male pauze i tako sve do posljednjeg sata. Jedino se ne smije zaboraviti na opcionalni unos trajanja velike pauze, zbog koje se svaki puta u iteraciji provjerava treba li se vrijednosti tog tekstnog okvira koji je na redu dodati vrijednost velike pauze ili samo male.

Zbrajanje varijabli tipa *TimeSpan* se radi putem metode namijenjene za tu biblioteku zvane *Add(),* a da se dobije nova vrijednost nakon zbrajanja, dovoljno je da se vrijednosti koja se želi uvećati u toj metodi priloži novi objekt tipa *TimeSpan* sa definiranim satima, minutama ili sekundama. U nastavku je u programskom isječku prikazano računanje za stupac „Početak“.

```
...
if (textbox.Name[0].ToString() == "B")
{
        TimeSpan hrPlusPrevBB = previousStartHour.Add(new TimeSpan(0,
        hourDuration, 0));
```

```
        TimeSpan hrPlusPrevShortBB = hrPlusPrevBB.Add(new TimeSpan(0,
        shortBreak, 0));


        if (longBreakStart == Int32.Parse(textbox.Name[4].ToString()))
        {
                TimeSpan hoursAfterLongBreakBB = hrPlusPrevShortBB.Add(new
                TimeSpan(0, longBreakDuration - shortBreak, 0));
                hrPlusPrevShortBB = hoursAfterLongBreakBB;


                previousStartHour = hoursAfterLongBreakBB;
                textbox.Text = hoursAfterLongBreakBB.ToString().Substring(0,
                5);
        }
        else
        {
                previousStartHour = hrPlusPrevShortBB;
                textbox.Text = hrPlusPrevShortBB.ToString().Substring(0, 5);
        }
}
...
```

Tablični prikaz je u tom trenutku spreman i proces se dovršava kada se klikne na „Spremi sate“, a prvo se prikupljeni podaci sortiraju, pa se u zasebne liste odvoje minute početka i završetka, te sati početka i završetka svih termina. Tada se ujedno i provjerava jesu li brojevi početaka i završetaka jednaki i ako jesu, obrađeni podaci se predaju metodi „CreateScheduledTasks()“ u logičkom sloju.

U toj metodi se radi s bibliotekom *TaskScheduler* i potrebno je primljene nizove tipa *string* pretvoriti u tip *DateTime,* nakon što se sati i minute spoje i između njih se doda dvotočka.

```
...
DateTime parsedHourB = DateTime.Parse(unparsedHourB,
CultureInfo.GetCultureInfo("hr-HR"));
DateTime parsedHourE = DateTime.Parse(unparsedHourE,
CultureInfo.GetCultureInfo("hr-HR"));
...
```

Kasnije kada se ti zadaci spreme i budu vidljivi u aplikaciji *Task Scheduler*, koja se može otvoriti jednim klikom iz istog prozora, osim odgovarajućih naziva svaki zadatak će imati opis, argument za prosljeđivanje, svakodnevno ponavljanje te putanju .exe datoteke koja se mora pokrenuti, a to je već instalirana aplikacija „Zvučne obavijesti“.

Kada bude vrijeme za to, na temelju svakog zadatka će se automatski pokrenuti aplikacija, bez obzira je li već otvorena ili ne. To će biti vidljivo kao mali prazni prozorčić koji sam od sebe nestane nakon 8 sekundi i na kojem se ništa ne može izvesti, ali se zato u isto vrijeme reproducira zvuk zvonjave koji je ugrađen kao resurs u aplikaciju. Navedeno se izvede na način da se primi argument vrijednosti „scheduler“ i u konstruktoru glavne forme se sve obavi. Za ovaj slučaj nije korištena biblioteka *NAudio*, zbog jednostavnosti, a aplikacija se gasi naredbom „Environment.Exit()“, što je prikazano u nastavku. Dakako, preduvjeti da se sve to na vrijeme i pravilno izvrši su da je računalo pokrenuto i da zauzetost računalnih resursa nije toliko velika da ono bude „zamrznuto“ u tom trenutku.

```
...
public MainForm(string[] args)
{
        if (args[0] == "scheduler")
        {
                var bellSound = Properties.Resources.School_bell;
                SoundPlayer audio = new SoundPlayer(bellSound);
                audio.Play();

                Task.Delay(8000).ContinueWith(t => Environment.Exit(0));
        }
}
...
```

4.2.7. Upravljanje zapisima

Upravljanje zapisima moguće je na glavnoj formi pomoću gumbi koji na sebi ne sadržavaju tekst, već samo intuitivne ikonice. Za prve tri slijeva (reproduciraj, zaustavi, stani) slučaj je takav da se to radi preko *NAudio* biblioteke i u potpunosti u logičkom sloju. Za reprodukciju, napravi se novi *WaveOutEvent(),* doda mu se handler na event *PlaybackStopped* i stvori se novi *AudioFileReader* koji se u prethodno stvorenom izlaznom uređaju inicijalizira. Preostaje još jedino da se to skupa pokrene.

```
...
public static void OnButtonPlayClick(string recording)
{
        if (outputDevice == null)
        {
                outputDevice = new WaveOutEvent();
                outputDevice.PlaybackStopped += OnPlaybackStopped;
```

```
        }
        if (audioFile == null)
        {
                string audioFilePath = Path.Combine(projectPath, recording);

                audioFile = new AudioFileReader(audioFilePath);
                outputDevice.Init(audioFile);
        }
        outputDevice.Play();
}
...
```

Za slučajeve zaustavi i stani, sve što se radi je da se pozovu metode iz te biblioteke zvane *Pause()* i *Stop(),* na isti način kao za *Play().*

Idući po redu je gumb za prikazivanje informacija o odabranom zvučnom zapisu iz tablice. To se izvršava na jednostavan način preko forme „InfoForm.cs“, u kojoj se iz sloja baze podataka dohvate potrebni podaci s kojima se popunjavaju tekstni natpisi (eng. *Label*) i tekstni okviri (eng. *TextBox*) prisutni u prozoru prikazan na slici 11. Ovaj proces je izvrstan primjer za spominjani *pass-through* zahtjev iz poglavlja 3.2.1., u kojem bi se vrlo lako moglo i preskočiti logički sloj jer se u njemu ništa korisno ne radi, ali se ipak neće jer sloj prikaza može do sloja baze podataka samo preko sloja između njih, a to je upravo logički. Dakle, sve što se u logičkome radi jest da se primi zahtjev od sloja prikaza, pozove se metoda „GetData()“ iz donjeg sloja i primljeni podaci vrate se na početak. To je prikazano u isječku koji slijedi.

```
...
public static List<string> FillRecordingInfo (string recordingId)
{
        return    PersistenceMainClass.GetData(column:    "name,    duration,
        description, timeCreated", table: "Recordings", condition: "id=" +
        recordingId, rowsCount: 4);
}
...
```

Brisanje zapisa je posljednji gumb u nizu i on također ima proces koji se dotiče baze podataka. Naime, nije dovoljno samo obrisati datoteku sa trajne memorije računala, već se to mora evidentirati i u bazi. Tu logički sloj jedino provjerava postoji li odabrana datoteka tamo gdje bi trebala biti kako ne bi došlo do nekakve neželjene greške u procesu. Nakon toga na red dolazi sloj baze podataka i on odrađuje ostatak posla metodom „DeleteValue()“. Iako sakriven, iz sloja prikaza dolazi „id“ potreban da bi se ovo izvršilo. Svakim klikom na tablicu

poziva se *event handler* koji u vrijednost globalne varijable „recordingId“ u tom sloju stavlja identifikacijski broj odabranog zvučnog zapisa. On se na kraju uvrštava u *SQL* naredbu kojom se traženi redak iz tablice u bazi podataka briše, a ona u aplikaciji se osvježava najnovijim podacima.

```
...
public static void DeleteValue(int id)
{
        ClearConnectionPools();
        using (var context = new ZvucneObavijestiEntities())
        {
                string sql = "DELETE FROM Recordings WHERE (id) = (@id)";
                con.Open();
                SqlCommand cmd = new SqlCommand(sql, con);
                cmd.Parameters.AddWithValue("@id", id);
                cmd.ExecuteNonQuery();
                con.Close();
        }
}
...
```

4.2.8. Odjava

Za ovu funkcionalnost također u igri sudjeluje *pass-through* zahtjev, a u odnosu na onaj prošli kod popune elemenata podacima o zvučnom zapisu, ovdje se baš ništa sloju prikaza ne vraća. Za odjavu je potrebno donjem sloju proslijediti korisničko ime trenutno prijavljenog korisnika, a u vrijednost stupca „macAddress“ u bazi podataka za taj račun upisuje se vrijednost „Logged out“. Odmah po završetku tog izvršavanja, aplikacija se ponovno pokreće i dolazi se na konstruktor glavne forme koji ne prima parametre. Tamo se metodom *CheckSession()* najprije provjerava kakav je status računa na računalu, a on će biti „NotLogged“ u slučaju da je posljednje bila napravljena odjava iz aplikacije. Automatski se otvara prozor za registraciju i prijavu kojim dolazimo na početnu funkcionalnost u 4. poglavlju.

```
...
private void btnLogOut_Click(object sender, EventArgs e)
{
      Logic_Layer.AccountControls.LogOut();
      Application.Restart();
      Environment.Exit(0);
}
...
```

5. Zaključak

Zadatak razvoja ovakve aplikacije uz primjenu slojevite arhitekture pokazao se savladivim, ali uz podosta upornosti, truda i strpljenja. Pošto do sad još nisam radio aplikacije na ovaj način, na početku sastavljanja programskog koda bilo mi je neobično razvrstavati dijelove funkcionalnosti i dodjeljivati ih posebnim slojevima, ali jednom kada se uđe u tu shemu razmišljanja proces postaje jednostavniji.

Slojevita arhitektura uistinu jest najbolji izbor za izgradnju ovakvog sustava, a unatoč pojedinim nedostacima pokazala se učinkovitom i prije svega pomogla je da raspored programskog koda, metoda i klasa bude lako čitljiv i lako popravljiv. Uvijek se zna otkud se kreće s nekim procesom, a zna se i u kojem smjeru se ide.

Ono što je činilo možda najveći problem tijekom razvoja je činjenica da slojevi „ne znaju“ ni za jednog drugog, osim onog koji je direktno ispod njih u hijerarhijskoj strukturi. U nekim situacijama, pogotovo u slučaju kada se dotični arhitekturni uzorak primjenjuje na *Windows Forms* aplikaciji, potrebno je imati izravan pristup elementima sučelja. Ako neki proces obavlja neku logičku operaciju u sloju koji nije gornji, nije baš idealno da sloj koji predstavlja sve što korisnik vidi ispred sebe ne zna što se dešava „ispod“ njega, već dobiva samo rezultate tih operacija. Kružna ovisnost (eng. *circular dependency*) nije dozvoljena u korištenom okruženju, premda u nekim situacijama definitivno ne bi bila na odmet.

Dokument koji čitate ispunjen je tekstom kojim sam se potrudio približiti i pojednostaviti principe slojevitog arhitekturnog uzorka prvo u obliku teorije i detaljne analize istog, a u drugom dijelu rada i na praktičnom dijelu bez kojeg se cijela priča ne bi mogla zaokružiti na pravi način. Spomenuti su pri tome svi korišteni alati kojima se to ostvarilo. Čak ni smišljanje na koji način će cijeli odrađeni posao biti čitateljima predstavljen nije bio odviše jednostavan. Da se nije radilo o uzorku gdje je ključ svega razumjeti tok zahtjeva od početka do kraja, sigurno se ne bih odlučio za primjerice, objašnjavanje procesa u funkcionalnostima u okomitom smjeru, ako se gleda po hijerarhijskoj strukturi slojeva.

Smišljanje plana kako će projekt biti izveden mnogo je dulje trajao nego njegova implementacija. Bilo je dosta slučajeva pri izgradnji aplikacije kada sam morao stati s radom i iako je u tom trenutku to bilo teško, prihvatiti da je potrebna mnogo dublja analiza neke domene, potom krenuti u proučavanje iste i tek nakon toga nastaviti „borbu“ s, u tom trenutku, naizgled nesavladivim poteškoćama. Shvaćanje i razumijevanje onog što je opisano i navedeno u teorijskom dijelu ovog rada je samo vrh ledenog brijega puta koji vodi ka savladavanju cjelokupne problematike. Na kraju je ipak sve zacrtano, pa i više od toga, bilo ostvareno i na to sam izuzetno ponosan.

6. Popis literature

6.1. Popis literature korištene za teorijski dio

[1] Wikipedia, „Microsoft Visual Studio“, 2022. [Online] Dostupno na: https://en.wikipedia.org/wiki/Microsoft_Visual_Studio [Pristupano: 10.4.2022.]

[2] Visual Studio Code (bez dat.) „IntelliSense“, [Online] Dostupno na: https://code.visualstudio.com/docs/editor/intellisense [Pristupano: 10.4.2022.]

[3] Microsoft, „Install and manage packages in Visual Studio using the NuGet Package Manager“, 2021. [Online] Dostupno na: https://docs.microsoft.com/en-us/nuget/consume-packages/install-use-packages-visual-studio [Pristupano: 10.4.2022.]

[4] Microsoft, „Manage references in a project“, 2022. [Online] Dostupno na: https://docs.microsoft.com/en-us/visualstudio/ide/managing-references-in-a-project?view=vs-2022 [Pristupano: 10.4.2022.]

[5] Microsoft, „Overview of .NET Framework“, 2021. [Online] Dostupno na: https://docs.microsoft.com/en-us/dotnet/framework/get-started/overview [Pristupano: 25.4.2022.]

[6] Microsoft (bez dat.), „What is .NET Framework?“. [Online] Dostupno na: https://dotnet.microsoft.com/en-us/learn/dotnet/what-is-dotnet-framework [Pristupano: 25.4.2022.]

[7] Wikipedia, „Microsoft SQL Server“, 2022. [Online] Dostupno na: https://en.wikipedia.org/wiki/Microsoft_SQL_Server [Pristupano: 23.6.2022.]

[8] P. Loshin (bez dat.), „Structured Query Language (SQL)“. [Online] Dostupno na: https://www.techtarget.com/searchdatamanagement/definition/SQL#:~:text=Structured%20Query%20Language%20(SQL)%20is,on%20the%20data%20in%20them. [Pristupano: 24.6.2022.]

[9] M. Richards, (2015). Software architecture patterns (Vol. 4, p. 1005). 1005 Gravenstein Highway North, Sebastopol, CA 95472: O'Reilly Media, Incorporated.

[10] Altitudeaccelerator.ca (bez dat.), „Software Architecture & Software Design Patterns for Startups“. [Online] Dostupno na: https://altitudeaccelerator.ca/software-architecture-design-patterns/ [Pristupano: 25.6.2022.]

[11] S. Cusimano, „Difference Between Layers and Tiers“, 2022. [Online] Dostupno na: https://www.baeldung.com/cs/layers-vs-tiers [Pristupano: 25.6.2022.]

[12] Vocabulary (bez dat.), „Monolith“. [Online] Dostupno na: https://www.vocabulary.com/dictionary/monolith [Pristupano: 15.7.2022.]

[13] Techtarget (bez dat.), „Monolithic architecture“. [Online] Dostupno na: https://www.techtarget.com/whatis/definition/monolithic-architecture [Pristupano: 17.7.2022.]

[14] Microsoft, „Entity Data Model“, 2021. [Online] Dostupno na: https://docs.microsoft.com/en-us/dotnet/framework/data/adonet/entity-data-model [Pristupano: 17.7.2022.]

[15] J. Burke (bez dat.), „What's the difference between a MAC address and IP address?“. [Online] Dostupno na: https://www.techtarget.com/searchnetworking/answer/What-is-the-difference-between-an-IP-address-and-a-physical-address [Pristupano: 10.8.2022.]

[16] Masterclass, „A guide to audio file formats“, 2021. [Online] Dostupno na: https://www.masterclass.com/articles/a-guide-to-audio-file-formats [Pristupano: 10.8.2022.]

[17] P. Rubens, „What Is SQL Injection and How Can It Hurt You?“, 2018. [Online] Dostupno na: https://www.esecurityplanet.com/threats/what-is-sql-injection-and-how-can-it-hurt-you/#:~:text=SQL%20injection%20attacks%20pose%20a,individual%20machines%20or%20entire%20networks [Pristupano: 10.8.2022.]

6.2. Popis literature korištene za programski kod

[18] Github (bez dat.), „Recording a WAV file in a WinForms app with WaveIn“. [Online] Dostupno na: https://github.com/naudio/NAudio/blob/master/Docs/RecordWavFileWinFormsWaveIn.md [Pristupano: 29.8.2022.]

[19] M. Heath, „How to record and play audio at the same time with NAudio“, 2014. [Online] Dostupno na: https://markheath.net/post/how-to-record-and-play-audio-at-same [Pristupano: 29.8.2022.]

7. Popis slika

Slika 1: Snimka zaslona sučelja u programu Microsoft Visual Studio 2022 5

Slika 2: Snimka zaslona dodanih biblioteka 5

Slika 3: Snimka zaslona sučelja u Microsoft SQL Server Management Studio 8

Slika 4: Model event-based arhitekturnog uzorka 11

Slika 5: Model arhitekturnog uzorka mikroservisa.....12

Slika 6: Model space-based arhitekturnog uzorka 12

Slika 7: Model slojevite arhitekture 14

Slika 8: Tok zahtjeva i svojstva slojeva.....16

Slika 9: Dodavanje otvorenog sloja.....17

Slika 10: Snimka zaslona glavne forme aplikacije 21

Slika 11: Snimka zaslona forme InfoForm.cs.....22

Slika 12: Snimka zaslona forme RegLoginForm.cs 23

Slika 13: Slika zaslona prozorčića s povratnom informacijom 24

Slika 14: Snimka zaslona forme AddNewRecordingForm.cs 24

Slika 15: Snimka zaslona forme LiveStreamForm.cs 25

Slika 16: Snimka zaslona forme ClassHoursForm.cs 26

Slika 17: Snimka zaslona sadržaja slojeva 27

Slika 18: Prikaz tablica i njihove relacije 28

Slika 19: Dijagram slijeda za proces sinkronizacije s bazom podataka 34

Slika 20: Dijagram aktivnosti reprodukcije zvuka za hitnu evakuaciju 35

8. Popis tablica

Tablica 1: Analiza općenitih prednosti i nedostataka primjene arhitekturnih uzoraka 10

Tablica 2: Primjer mogućih slojeva u slojevitoj arhitekturi 15

Tablica 3: Opis svojstava arhitekturnih uzoraka u slučaju slojevite arhitekture 19