APLIKACIJA ZA UPRAVLJANJE PODACIMA O KLIJENTIMA I UGOVORIMA

Kasalo, Ivana

VISOKO UČILIŠTE ALGEBRA

ZAVRŠNI RAD

Aplikacija za upravljanje podacima o klijentima i ugovorima

Ivana Kasalo

Zagreb, veljača 2020.

„Pod punom odgovornošću pismeno potvrđujem da je ovo moj autorski rad čiji niti jedan dio nije nastao kopiranjem ili plagiranjem tuđeg sadržaja. Prilikom izrade rada koristila sam tuđe materijale navedene u popisu literature, ali nisam kopirala niti jedan njihov dio, osim citata za koje sam navela autora i izvor, te ih jasno označila znakovima navodnika. U slučaju da se u bilo kojem trenutku dokaže suprotno, spremna sam snositi sve posljedice uključivo i poništenje javne isprave stečene dijelom i na temelju ovoga rada“.

U Zagrebu, 18.02.2020.

Ivana Kasalo

Predgovor

Prvenstveno zahvaljujem profesoru Aleksanderu Radovanu koji mi je svojim mentorstvom pomogao u ostvarenju cilja te svojim savjetima omogućio podizanje kvalitete rada. Također zahvaljujem svim profesorima Visokog učilišta Algebra na prenesenom znanju i izučavanju kompetencija koje su mi uvelike pomogle ne samo u akademskom okruženju, nego i u začetku poslovne karijere.

Posebno se zahvaljujem svojoj obitelji na iskazanoj potpori i podršci tijekom studiranja i života.

Prilikom uvezivanja rada, Umjesto ove stranice ne zaboravite umetnuti original potvrde o prihvaćanju teme završnog rada kojeg ste preuzeli u studentskoj referadi

Sažetak

Ubrzani način života doveo je do ubrzanog vođenja poslovanja te se većina administrativnih poslova pokušava automatizirati. U ovoj aplikaciji je prikazan mali dio te automatizacije, odnosno kreiranje dokumenata kroz predloške i organiziranje dokumenata, što ubrzava proces pretrage. Brzim razvojem tehnologija, potreba za interoperabilnošću sustava je sve veća, a prelazak na novu verziju treba biti brz i jednostavan. U tom vidu aplikacija je podijeljena na serversku i klijentsku aplikaciju koje su neovisne jedna o drugoj i komuniciraju putem HTTP protokola. Na ovaj način je pojednostavljena mogućnost prelaska na poslovanje u oblaku jer se klijentska aplikacija može koristiti samo na Windows operacijskom sustavu. U cilju izrade aplikacije, koja ima sve što korisniku treba na jednom mjestu, aplikacija se povezuje s raznim API-jima kao što su Google Drive, Google Calendar, One Drive, Outlook Calendar, HNB tečajna lista i popis poduzetnika Sudskog registra.

Ključne riječi: WPF, Spring Boot, MongoDB, interoperabilnost, API, Sudski registar API, HNB API, Google API, Microsoft API, REST

Abstract

Busy way of life has led to a hastily business organization which made the business world try to automate as many everyday jobs as possible. In this application a bit of automatization is shown through an automated document creation via templates and documents management. Lately, system operability has been more important than ever and process of implementing a new version has to be fast and simple. With that in mind this application has been divided in two applications; server and client. They are independent from each other as they communicate through HTTP protocol. Client application is a desktop application available only for Windows OS and independency makes it easier to migrate to cloud solutions (if needed). The application is trying to give users everything they use and need at one place as it connects with multiple APIs like Google Drive, Google Calendar, One Drive, Outlook Calendar, HNB exchange rate list and list of Croatian companies from Sudski registar.

Key words: WPF, Spring Boot, MongoDB, interoperability, API, Sudski registar API, HNB API, Google API, Microsoft API

Sadržaj

1. Uvod

Ovaj rad je zamišljen kao aplikacija za olakšavanje poslovnih procesa, odnosno automatiziranje izrade ugovora i drugih dokumenata te upravljanje dokumentima. Mnoga poduzeća i dalje vode liste klijenata kroz Excel tablice te je za te korisnike ova aplikacija i namijenjena. Aplikacija omogućuje manjim poduzećima, koja ne koriste naprednije sustave npr. CRM, da organiziraju svoje podatke i dokumente te da kreiraju dokumente pomoću predložaka.

Kroz ovaj rad se želi pokazati interoperabilnost kod višeslojnih arhitektura. Fokus je na interoperabilnosti poslovnih slojeva, odnosno izvedbi klijentske aplikacije kroz WPF grafički podsustav i izvedbi serverske aplikacije u programskom okviru Spring kao REST aplikacija.

Kôd je pisan na engleskom jeziku u cilju stvaranja uniformnog kôda, lakše čitljivosti i poštivanja konvencija nazivlja.

Rad je podijeljen u sedam cjelina: uvod, tehnologije, opis aplikacije, aplikacija, testiranje, implementacija i zaključak. U uvodnom dijelu je opisana svrha i tema rada. U cjelini tehnologije su opisane tehnologije koje su korištene u radu, odnosno WPF za klijentski dio, Spring za serverski dio te MongoDB baza. U poglavlju interoperabilnost je opisano kako se serverska i klijentska aplikacija povezuju i razmjenjuju podatke. U opisu aplikacije su navedeni propisi kojih se aplikacija treba pridržavati, funkcionalnost aplikacije i usporedba s postojećim rješenjima na tržištu. U cjelini aplikacija opisana je arhitektura aplikacije s primjerima kôda. U testiranju je opisan način testiranja i rezultati. Poglavlje implementacija opisuje kreiranje instalacijskih paketa i instalaciju. Zaključak označava završni dojam i najvažnije stavke. Na kraju dokumenta se nalazi popis kratica, slika, tablica, kôdova i literatura.

Rad je napravljen u skladu sa službenom dokumentacijom i literaturom.

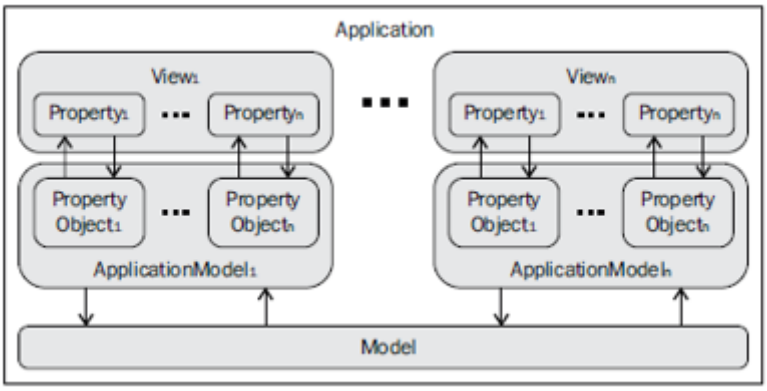
2. Tehnologije

2.1. WPF

WPF je grafički podsustav za *desktop* aplikacije za Windows operacijski sustav. WPF je dio .NET-a programskog okvira, koji je razvojna platforma otvorenog kôda (engl. *open source*). [1]

Koristi se za razvoj aplikacijskih obilježja koja uključuje XAML, kontrole, povezivanje podataka (engl. *data binding*), razmještaj kontrola (engl. *layout*), 2D i 3D grafiku, animaciju, stilove, predloške, dokumente, slike, tekst i tipografiju. Programski jezici koji se koriste za razvoj su C# i Visual Basic. U ovom radu koristit će se C# programski jezik. C# je objektivno orijentirani jezik čija je sintaksa vrlo slična C, C++ i Java jeziku. Trenutno aktualna verzija C#-a je 8. Neke od posebnosti C# jezika su definiranje svojstva (engl. *properties*) koja služe za dohvat privatnih varijabli, odnosno kraći način zapisivanja *get* i *set* metoda i korištenje ugrađenih upita (LINQ) za iteraciju. [2]

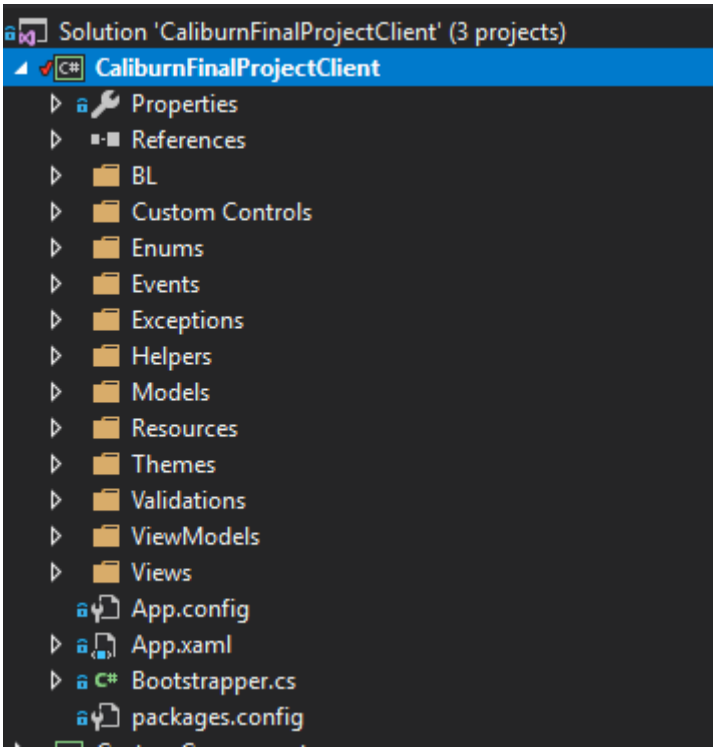
WPF koristi MVVM strukturu (engl. *pattern*) za izvedbu korisničkog sučelja (engl. *UI*) čiji je cilj odvojiti objekte, odnosno podatke koji se žele prikazati od prezentacijskog dijela. U modelu se nalaze informacije izražene kroz svojstva, ali ne i logika dohvaćanja podataka ili dio dizajna. *View* je pisan u XAML jeziku i sastoji se od komponenti koje se prikazuju korisniku (npr. forma, gumb). *ViewModel* služi kao poveznica između podataka i dizajna. Model se instancira podacima iz servisa koji dohvaća podatke iz baze ili se podaci šalju u bazu te se sinkroniziraju s podacima koji se prikazuju korisniku. Na ovaj način se pojednostavljuju izmjene u kôdu te održavanje kôda. Na sljedećoj slici je prikazan odnos *view*-a, *viewmodel*-a i *modela*-a. *View* ima n *property*-ja na komponentama koji se povezuju s *property* objektima u *viewmodel*-u. *Model* je povezan s *viewmodel*-om preko *property* objekata. Svi *property*-ji su povezani sa strelicama u oba smjera, što označava obostrano povezivanje objekata odnosno `Binding`, koji je detaljnije objašnjen u poglavlju 4. [3]



Slika 2.1. Odnos *view* – *viewmodel* – *model* koji omogućuje obostrano povezivanje objekata

Svaki *view* ima *.xaml* datoteku i parcijalnu klasu, odnosno C# ili Visual Basic kôd (engl. *code behind*). *Code behind* bi se trebao što manje koristiti kako bi se ostvarila MVVM struktura. U tu svrhu će se u ovom radu koristiti **Caliburn Micro** programski okvir koji automatski povezuju *view* i *viewmodel*. Caliburn Micro se zasniva na konvencijama, odnosno vrijednosti iz *viewmodel*-a se povezuju s vrijednošću koja je definirana kao **x:Name** ili se poziva „glavni *event*“ (npr. *click* na gumbu) u *view*-u. [4]

Na slici niže je prikazana struktura WPF aplikacije u kojoj su glavni direktoriji za korištenje Caliburn Micro-a: *Views, ViewModels, Models* te ostali direktoriji koji nisu obavezni, ali se koriste za bolju organizaciju aplikacije. Struktura je detaljnije objašnjena u cjelini 4.



Slika 2.2. MVVM struktura WPF aplikacije

2.2. Spring

Java je objektno orijentirani programski jezik, razvijen 1995. godine, kako bi se koristio na svim platformama kroz JVM. Trenutno aktualna verzija je Java 13, ali sve novije verzije Jave su kompatibilne sa starijim verzijama. Budući da se korištenje verzije Java 11 (ili više) u produkciji naplaćuje, u ovom radu se koristi Java 8. Java 8 uvodi mnogo inovacija od kojih je jedna i funkcionalno programiranje kroz lambda izraze. [5]

Spring je Java web programski okvir koji se temelji na Java web tehnologiji, odnosno na servletima. Trenutno aktualna verzija je Spring 5 i dijeli se na reaktivni stog koji koristi *Spring WebFlux* i servletski stog koji koristi *Spring MVC*. [6]

Spring projekt inicijalizira se pomoću **Spring Initializr** sučelja kojim se definira osnova konfiguracija i ovisnosti (engl. *dependencies*). Spring Initializr se može koristiti kroz web sučelje, Netbeans *plugin*, Eclipse *plugin* ili u IntelliJ komercijalnoj verziji. Spring Boot je platforma koja predstavlja početak Spring aplikacije i pojednostavljuje postupak pokretanja i konfiguracije, a označava se pomoću **@SpringBootApplication** anotacije. Spring Boot koristi ugrađene servere: Tomcat, Jetty ili Undertow. [6]

Spring aplikacije se pokreću pomoću Maven ili Gradle alata, a u ovom radu se koristi Maven. **Apache Maven** je alat za pokretanje otvorenog kôda koji preuzima JAR datoteke s javnog Maven repozitorija. Korišteni JAR-ovi se definiraju kroz ovisnosti u pom.xml datoteci.

U modernom Java razvoju se koriste anotacije koje pojednostavljuju kôd i koriste već definiranu konfiguraciju. Spring kroz anotacije koristi princip inverzije kontrole (IOC), odnosno umetanje ovisnosti (DI) (npr. `@Autowired` anotacija). [7]

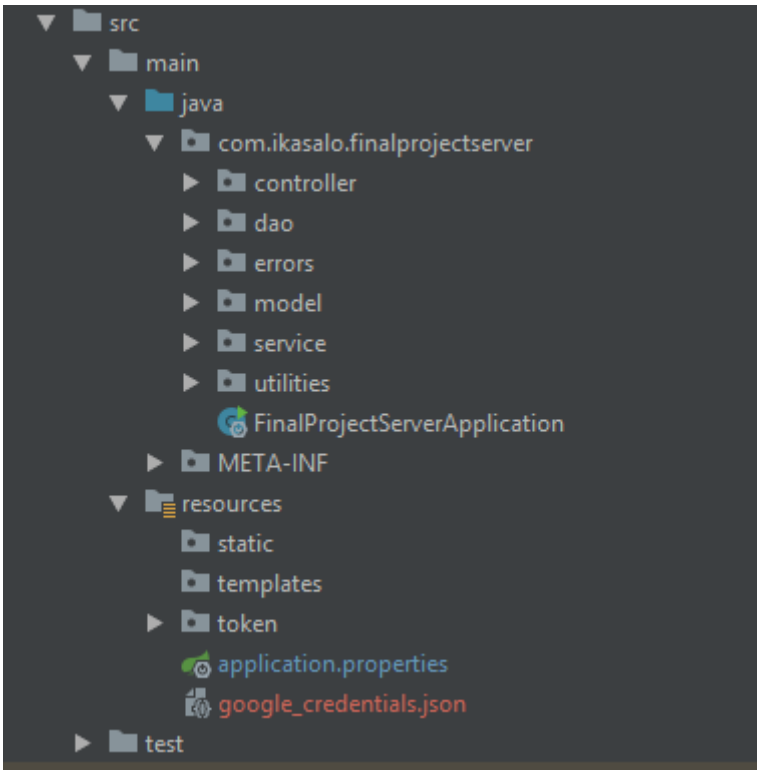
U tabličnom prikazu su navedene anotacije koje se koriste u radu te njihovo objašnjenje.

Tablica 2.1. Stereotipi koji omogućuju skeniranje i automatsko povezivanje

Komponenta	**Opis**
@Component	osnovni stereotip
@Service	označava klase u sloju poslovne logike
@Repository	označava repozitorij baze i baca neprovjerene pogreške (engl. *unchecked exceptions*)
@Controller	povezuje REST metode s *view*-om
@RestController	koristi se za REST servise. Kraći način zapisivanja kombinacije `@Controller` i `@ResponseBody`

@Bean	objekti koje koristi Spring razvojni okvir

Na sljedećoj slici je prikazana struktura Spring aplikacije čija se podjela bazira na gore navedenim anotacijama. Glavni paketi su: *controller, dao, model i service*. *Error* paket sadrži pogreške koje aplikacija baca, a *utilities* statičke klase s metodama koje se koriste kroz cijelu aplikaciju. Struktura je detaljnije objašnjena u cjelini 4.



Slika 2.3. Struktura Spring aplikacije

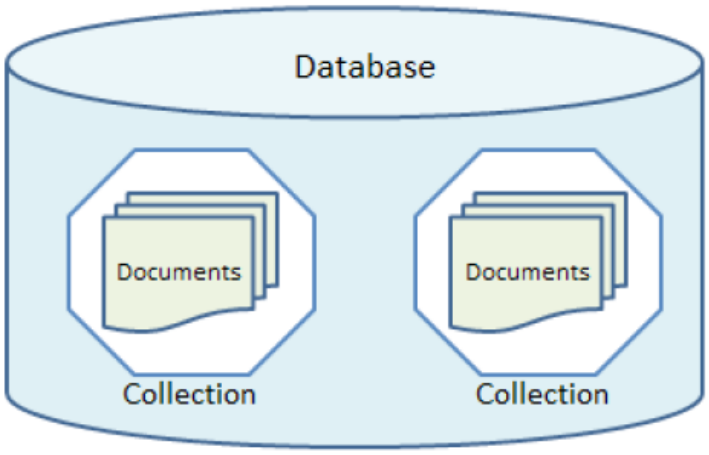
2.3. MongoDB

„Baza podataka je skup međusobno povezanih podataka pohranjenih na vanjskoj memoriji koji su istovremeno dostupni raznim korisnicima i programima.“ [9] Baze podataka se dijele na relacijske i nerelacijske. Relacijske baze koriste tablice koje su definirane atributima i tipovima podataka. Svaki atribut tablice definira primarni ključ (eng. *primary key*) koji je obavezno ograničenje (engl. *constraint*), a s podacima druge tablice se povezuje pomoću stranih ključeva (engl. *foreign key*). Osim navedenih ograničenja, ostala ograničenja mogu biti *null, not null, unique, check* i/ili *default*. Shema tablica i upiti prema bazi (engl. *query*) definirani su u SQL jeziku. Neke od popularnih relacijskih baza su: MySQL, MSSQL, SQLite, Oracle, PostgresSQL, itd. Nerelacijske baze, odnosno NoSQL baze se dijele u

nekoliko kategorija: baza ključ/vrijednost (engl. *key-value stores*), baza širokih kolona (engl. *wide-column stores*), dokumentne baze (engl. *document databases*), baze grafova (engl. *graph databases*). Najpoznatije baze su: Redis, Amazon DynamoDB, Cassandra, Scylla, MongoDB, CouchBase, Neo4j, itd. [8]

Najveća razlika između relacijskih i NoSQL baza podataka je fleksibilnost shema. Relacijske baze ovise o predefiniranim strukturama, a u NoSQL bazama svaka shema ne mora bit ista. NoSQL su brže od relacijskih baza, ali nisu dobar izbor ukoliko postoji puno podataka koji se referenciraju iz drugih kolekcija, što dovodi do kompleksnijih upita prema bazi. SQL baze imaju vrlo sličnu sintaksu, što omogućuje jednostavnu prenosivost, dok NoSQL baze nemaju određenu standardizaciju te je prelazak na drugu bazu dosta kompleksan i skup. [8]

MongoDB je najpopularnija NoSQL baza i besplatna je za korištenje. Kategorizira se kao dokumentna baza jer koristi kolekcije i dokumente. Jedna kolekcija može imat više različito definiranih dokumenata. Podaci se spremaju u BSON obliku koji omogućuje prikaza podataka u JSON obliku te brže enkodiranje i pohranu podataka. Uz osnovne JSON tipove podataka, BSON podržava pohranu binarnih podataka, podataka tipa datum (engl. *Date*) u Unix formatu koji vraća trenutni datum. Sljedeća slika prikazuje osnovnu strukturu baze, odnosno odnos kolekcija i dokumenata. Iz slike je vidljivo kako svaka kolekcija može imati više dokumenata, tj. različitu shemu. [8]



Slika 2.4. Osnovni elementi Mongo DB baze

Svaki objekt u dokumentu je definiran 12-bitnim tekstualnim „primarnim ključem“ (ObjectId, _id) koji Mongo automatski generira. ID je indeksiran i koristi se za brzu pretragu. Ostali podaci po kojima će se vršiti pretraga se također mogu indeksirati. Indeks može biti

na jednoj ili više vrijednosti, ali također i na polju pri čemu se automatski postavlja indeks na svaku vrijednost iz polja. [8]

2.4. Interoperabilnost

Jedna od mogućih definicija interoperabilnost je „stupanj u kojem se dva proizvoda, programa i sl. mogu zajedno koristiti, ili kvaliteta mogućnosti da se koriste zajedno“, odnosno povezivanje različitih tehnologija u jedinstvenu cjelinu. [10]

U ovom radu su razvijene dvije aplikacije, klijent u C# i poslužitelj u Java programskom jeziku. Aplikacije su prikazane kroz višeslojnu arhitekturu te su podijeljene na slojeve. Klijent je podijeljen na prezentacijski sloj (MVVM struktura) i sloj poslovne logike koji služi za dohvat podataka s poslužitelja. Poslužitelj je REST servis te se sastoji od sloja pristupa podacima i sloja poslovne logike. Podatkovni sloj predstavlja bazu podataka.

REST je usluga Weba, koja koristi URI za dohvaćanje i izmjenu resursa, što omogućuje neovisnost korisnika usluge. REST funkcionira na principu zahtjev-odgovor koji se odvija putem HTTP metoda. Četiri osnovne HTTP metode su: dohvat (**GET**), kreiranje (**POST**), ažuriranje (**PUT**) i brisanje (**DELETE**). [10]

Osim interoperabilnosti prikazane kroz višeslojnu arhitekturu, u ovom radu je prikazana i integracija s javnim servisima, tj. API-jima. Aplikacija se povezuje s tečajnom listom HNB-a, Sudskim registrom, Google Drive-om, Google kalendarom, One Drive-om i Microsoft kalendarom.

3. Opis aplikacije

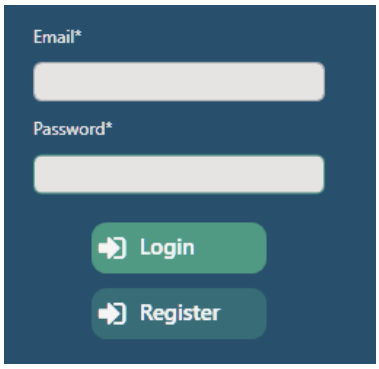
3.1. Propisi

Uredba o zaštiti osobnih podataka (**GDPR**) stupila je na snagu sa Zakonom o provedbi Opće uredbe o zaštiti podataka dana 25.05.2018. godine te vrijedi za građane Europske Unije.
Članak 4.1. definira osobne podatke: „“osobni podaci“ znači svi podatci koje se odnose na pojedinca čiji je identitet utvrđen ili se može utvrditi „ispitanik“; pojedinac čiji se identitet može utvrditi jest osoba koja se može identificirati izravno ili neizravno, osobito uz pomoć jednog ili više identifikatora kao što su ime, identifikacijski broj, podaci o lokaciji, mrežni identifikator, ili uz pomoć jednog ili više čimbenika svojstvenih za fizički, fiziološki, genetski, mentalni, ekonomski, kulturni ili socijalni identitet tog pojedinca. Članak 6.1.(a) navodi kako je obrada zakonita ako je „ispitanik dao privolu za obradu svojih osobni podataka u jednu ili više posebnih svrha“ na što aplikacija nema utjecaj. [11]
U svrhu automatiziranih podataka aplikacija neće sakupljati posebno kategorizirane osobne podatke koji se navode u članku 9.1., odnosno one koji otkrivaju rasno ili etničko podrijetlo, politička mišljenja, vjerska ili filozofska uvjerenja, članstvo u sindikatu, genetičke podatke, podatke o zdravlju, spolnom životu, ekonomskom stanju, predviđanje poslovnog učinka ili kaznene osude. [11]
U svrhu transparentnosti iz članka 14.2.(f) aplikacija navodi izvor osobnih podataka te da li podatci dolaze iz javno dostupnih izvora. U skladu s člankom 13.2.(b) aplikacija omogućuje brisanje i ispravak podataka. [11]
U svrhu sigurnosti navedene u članku 32.(c), aplikacija neće biti odgovorna za sposobnost pravodobne ponovne uspostave dostupnosti osobnih podataka. Kako bi se osigurala pseudonimizacija (članak 4.5. navodi kako su to osobni podaci koji „se više ne mogu pripisati određenom ispitaniku bez uporabe dodatnih informacija“), korisnik može pristupiti aplikaciji isključivo autentifikacijom i sve izmjene se prate putem zapisanih podataka (engl. *log*). [11]

3.2. Funkcionalnost aplikacije

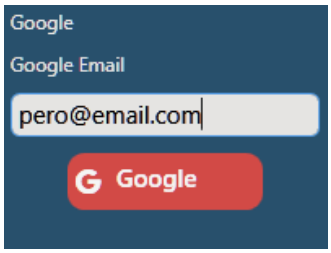
Aplikaciji mogu pristupiti isključivo prijavljeni korisnici koji se prije prijave moraju registrirati. Na sljedećoj slici je prikazan ekran za prijavu na kojem korisnik unosi email i

lozinku te se prijavljuje u aplikaciju. *Click*-om na gumb „Register“ se otvara ekran za registraciju.



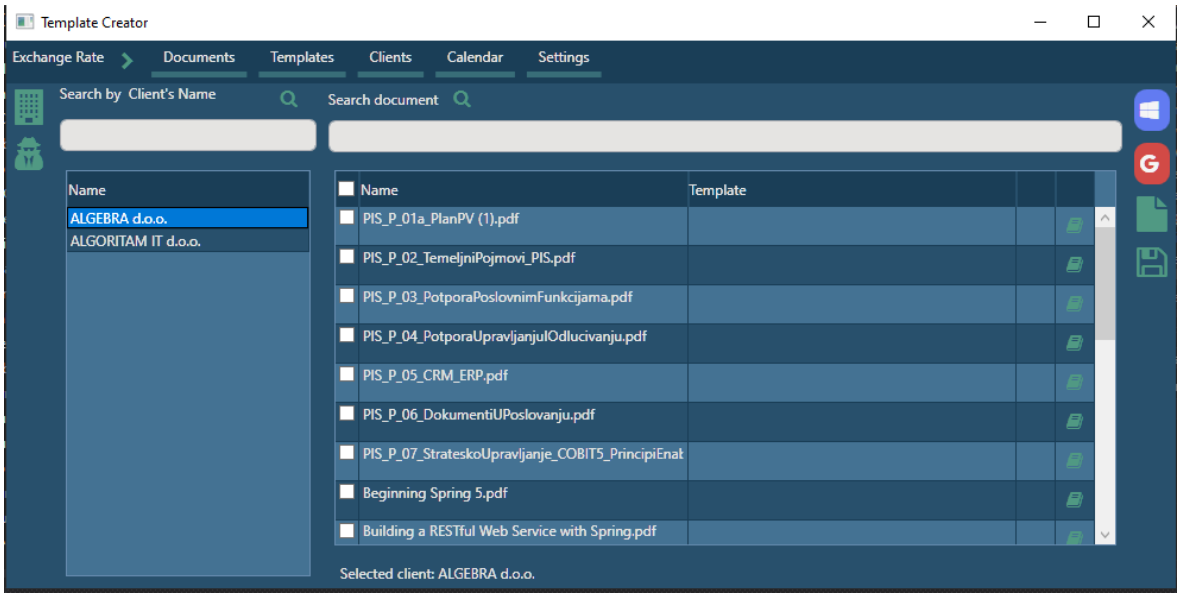
Slika 3.1 Ekran za prijavu

Nakon uspješne registracije korisniku se prikazuje ekran „Settings“ na kojem može povezati svoj račun s Google i/ili Microsoft računima. Kako bi se računi povezali, korisnik unosi email adresu koju želi povezati te se otvara zadani preglednik preko kojeg se prijavljuje na svoj Google ili Microsoft račun i daje privolu. Nakon prijave korisnik može nastaviti koristiti aplikaciju. Ukoliko se korisnik prijavljuje, otvara se ekran „Documents“. Unos emaila za povezivanje s Google računom prikazan je na sljedećoj slici.



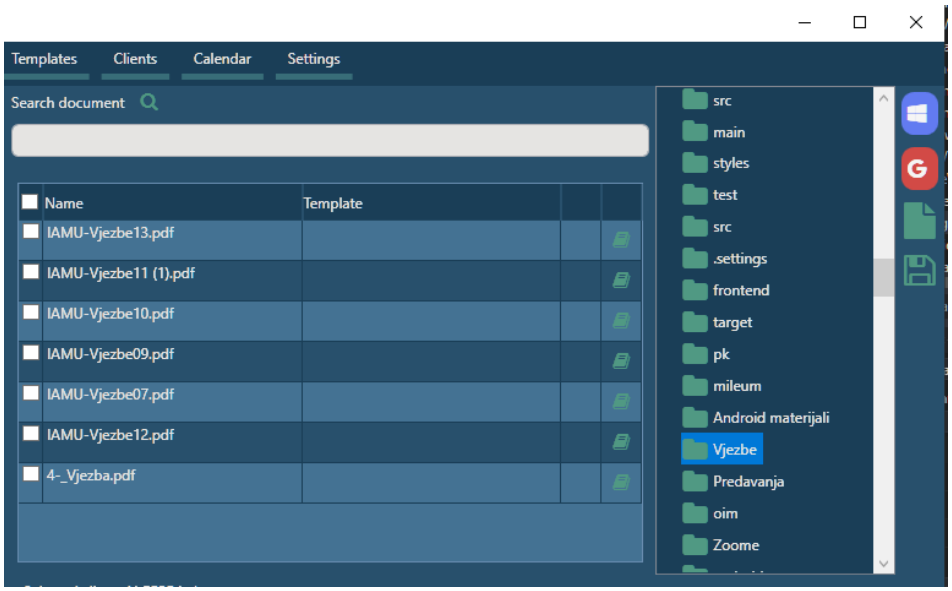
Slika 3.2 Povezivanje s Google računom

Na ekranu za prikaz dokumenata, korisnik pretražuje dokumente koji su povezani s poduzećima ili fizičkim osobama ili pretražuje dokumente po nazivu. Novi dokumenti se dodaju tako što korisnik odabire direktorij iz kojeg želi uvesti *.pdf*, *.doc* i *.docx* dokumente. Za rješenja u oblaku, direktoriji se pokazuju s desne strane te se odabirom direktorija dokumenti prikazuju u tabličnom obliku. Korisnik može odabrati koje dokumente želi povezati s poduzećem ili fizičkom osobom. Sami dokumenti se ne pohranjuju u bazu, već se sprema lokacija ili ID dokumenta ukoliko se radi o rješenju u oblaku. Na sljedećoj slici je prikazan „Documents“ ekran (za poduzeća) na kojem su prikazani dokumenti koji su povezani s poduzećem ALGEBRA d.o.o.



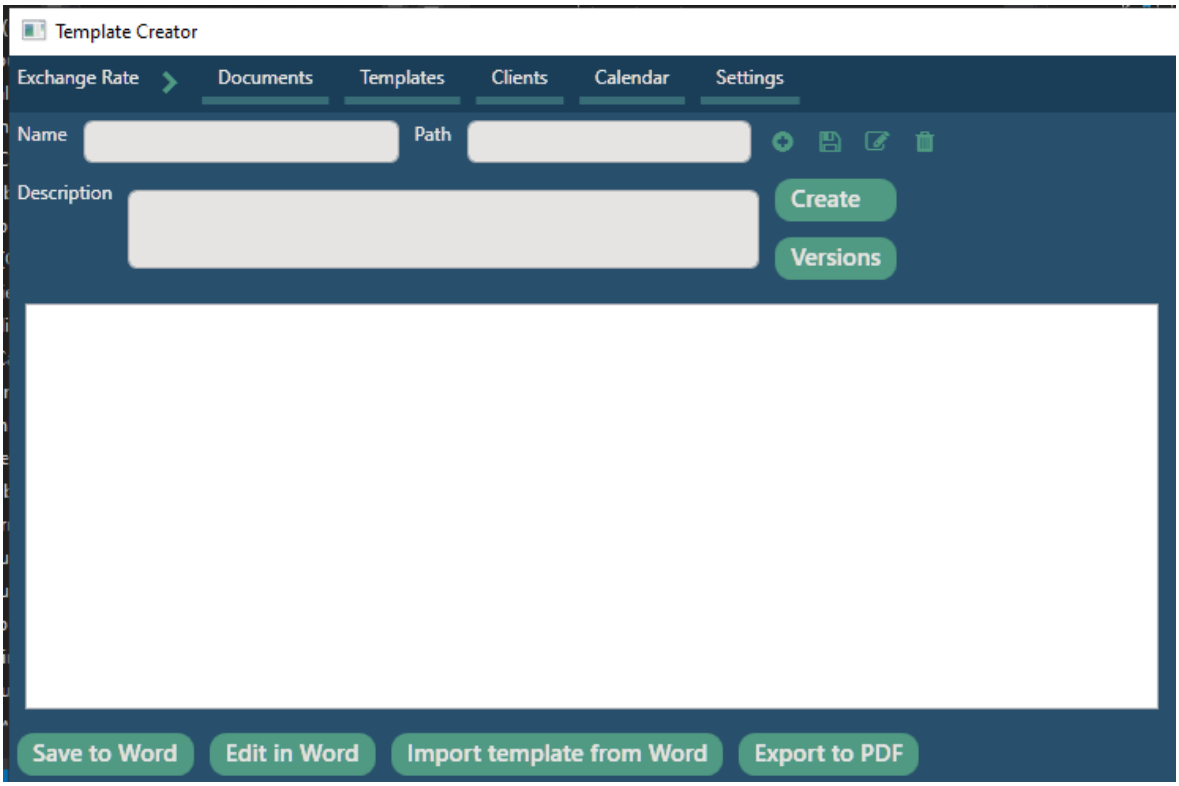
Slika 3.3. Ekran za prikaz dokumenata

Na sljedećoj slici, s desne strane prikazani Google Drive direktoriji te dokumenti iz direktorija „Vjezbe“.



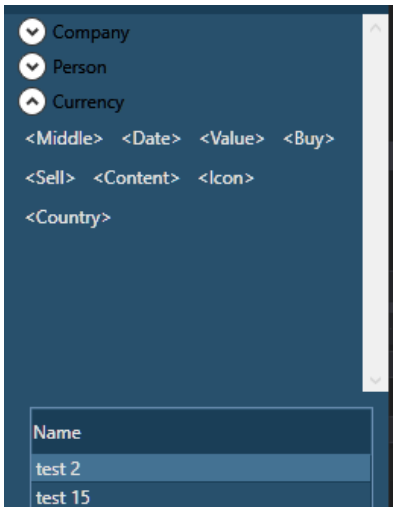
Slika 3.4 Ekran s Google Drive direktorijima

Izrada dokumenata s predloškom je moguća samo za klijente koji su pohranjeni u bazu. Sve promjene na predlošcima i dokumentima su vidljivi iz programa te se na taj način jednostavno mogu pratiti verzije dokumenata, koji predložak je korišten te tko je napravio zadnju verziju. Verzije dokumenata se pohranjuju u bazu. Korisnici mogu izrađivati predloške u *.docx* formatu, a kasnije se mogu izvesti u *.pdf* dokument. Sljedeća slika prikazuje ekran za izradu predložaka. Korisnik unosi naziv, opis i tekst predloška te može pohraniti predložak lokalno kako bi se popunilo polje za putanju dokumenta.



Slika 3.5 Ekran za izradu predložaka

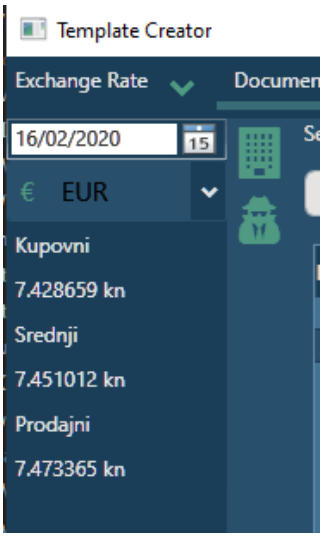
Na sljedećoj slici je prikazan isječak *tag*-ova koji se mogu dodati u predložak te tablični prikaz unesenih predložaka. Vrijednost *tag*-ova su svi *property*-ji za određeni objekt, odnosno datum, valuta, država te kupovna, srednja i prodajna vrijednost deviza objekta `Currency`. *Tag*-ovi se u predložak dodaju metodom povuci-ispusti (engl. *drag and drop*).



Slika 3.6 *Placeholder*-i za izradu predložaka

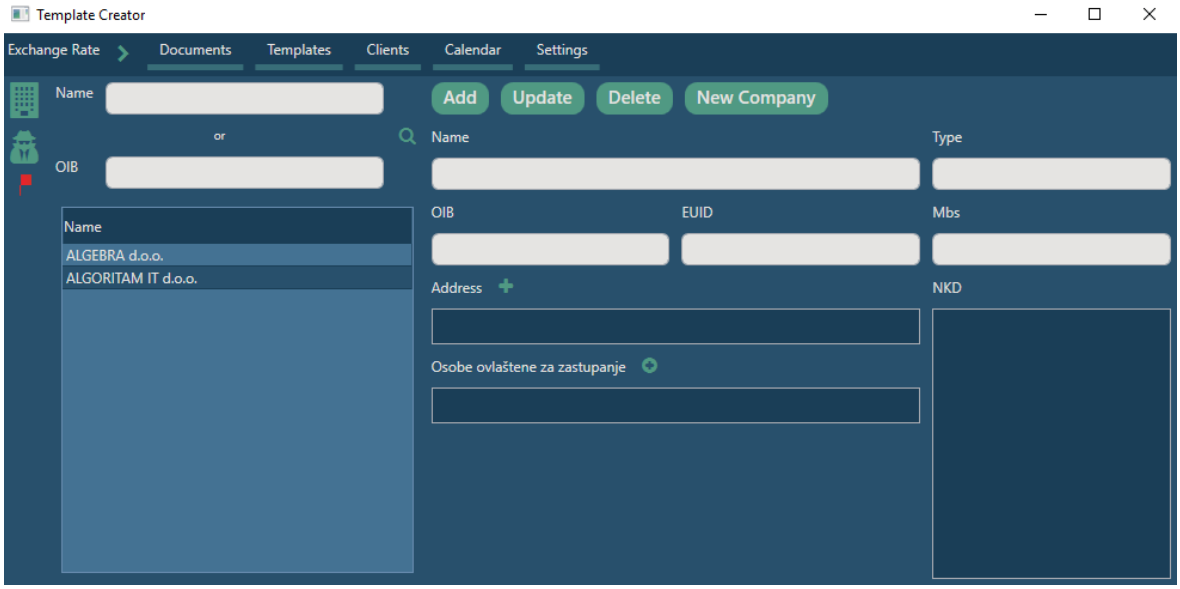
Osim podataka o klijentima, u predloške se mogu dodati iznosi valuta s tečajne liste HNB-a, čime se olakšava izrada računa ili ugovora. Ti podaci su vidljivi na lijevoj strani ekrana.

Sljedeća slika prikazuje isječak ekrana s HNB tečajnom listom. Prikazan je tečaj za euro na dan 16.02.2020.



Slika 3.7 Ekran tečajne liste

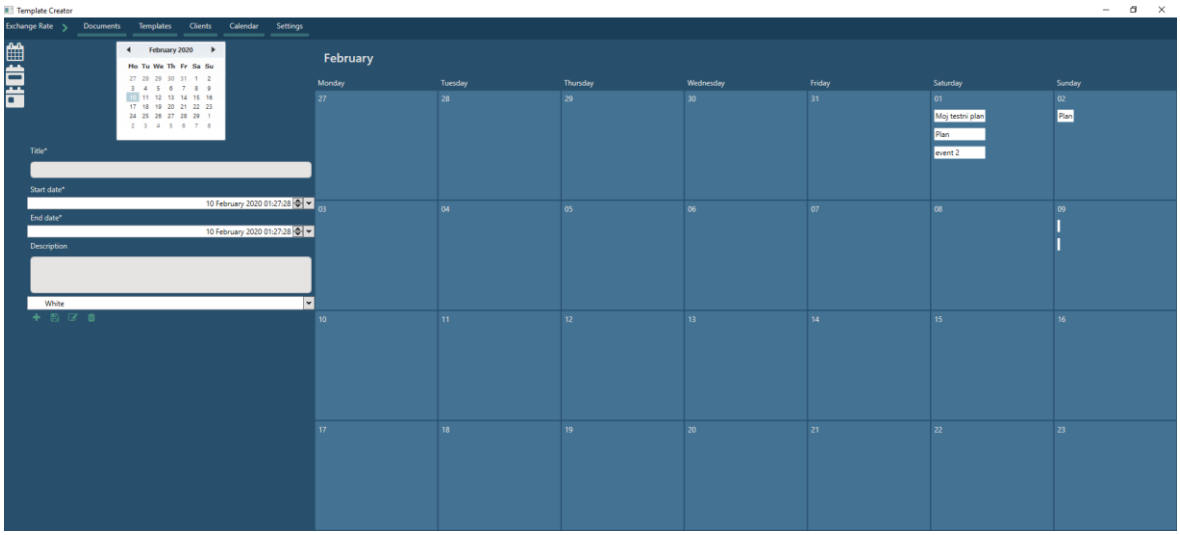
Novi klijenti se dodaju „ručno“ ili pretragom baze hrvatskih poduzetnika (Sudski registar). Za poduzeća se pohranjuju podaci: MBS, OIB, naziv, EUID, datum osnivanja, datum brisanja, adresa, kapital, klasifikacija po NKD-u, tip društva, ovlaštene osobe i kontakt. Za fizičke osobe se unose: ime, prezime, OIB, adresa i kontakt. Ekran za prikaz i uređivanje klijenata prikazan je niže te se mogu vidjet polja za pretraživanje po nazivu i OIB-u, tablični prikaz poduzeća i polja s podacima o poduzeću.



Slika 3.8. Ekran za prikaz i uređivanje klijenata

Planer i kalendar omogućuju jednostavno praćenje zadataka i sastanaka. Kalendar je također povezan s Google ili Microsoft kalendarom. Unosi se vrijeme, datum, naslov i opis te se

zadatci kategoriziraju u različitim bojama koje korisnik odabire sukladno svojim preferencama. Pregled kalendara je omogućen u dnevnom, tjednom ili mjesečnom obliku. Na donjoj slici prikazan je izgled mjesečnog kalendara i polja za uređivanje ili unos plana.



Slika 3.9. Prikaz mjesečnog kalendara

3.3. Usporedba s postojećim rješenjima

Na tržištu postoji dosta aplikacija za upravljanje dokumentima. Jedna od najpoznatijih aplikacija je M-Files. M-Files kao glavne prednosti navodi inteligentan sustav, neutralan repozitorij i baziranje na meta podacima. Dokumente pretražuje ovisno o tipu dokumenta i meta podacima, a ne na temelju putanje. Budući da ne ovise o platformi, M-Files se može instalirati u oblak, na poslužitelj ili se može koristiti kao hibridno rješenje. Također, jedna od prednosti je uniformno sučelje na svim uređajima. Osim samog upravljanja dokumentima M-Files nudi integraciju s raznim rješenjima: Salesforce CRM, Google G Suite, SharePoint Online (za Office 365) te rješenje za timove. [15]

DocuWare navodi kako njihov sustav ima sve komponente DMS-a, odnosno pohranjivanje skeniranih dokumenata, sigurnost i restrikciju pristupa, indeksiranje, dijeljenje i istovremeni rad s dokumentima, integraciju s različitim sustavima (SAP, Microsoft SharePoint, Microsoft Dynamics i QuickBooks), upravljanje informacijama kroz BI te rješenje u oblaku. [16]

Toscana ERP & CRM je hrvatsko rješenje u oblaku dostupno na 3 jezika (hrvatski, engleski i njemački). U osnovnom rješenju nudi centralnu bazu koja pristupa svim poslovnim subjektima, povezivanje s predefiniranim email klijentom, arhiviranje dokumenata po

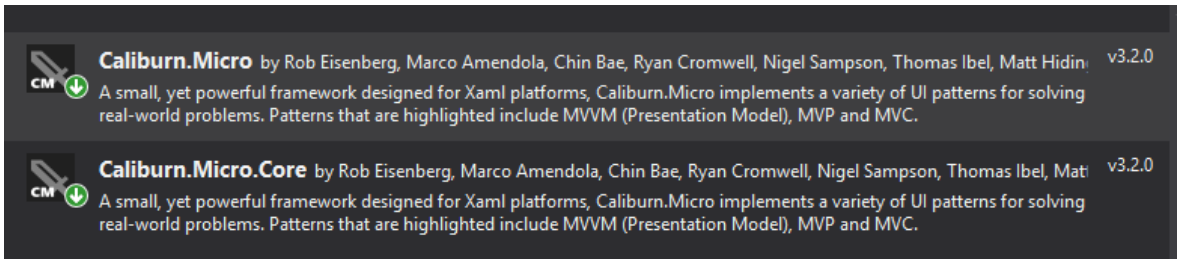
klijentima, kalendar s radnim zadacima, kreiranje GDPR privola, evidenciju radnog vremena po dodijeljenim zadacima. [17]

Ova aplikacija se spaja s bazom hrvatskih poduzetnika (Sudski registar) čime je prilagođena hrvatskom tržištu. Ostale mogućnosti za kreiranje i uređivanje dokumenata su vrlo slične M-Files-a i DocuWare-a rješenjima. Ta rješenja koriste napredan sustav pretraživanja koji nije u sklopu ovog projekta. Budući da je klijentska aplikacija vezana uz Windows operacijski sustav, nije omogućeno pregledavanje na više uređaja bez prethodne instalacije. Gore navedena rješenja imaju tu mogućnost s obzirom da su dostupna u oblaku. No za razliku od gore navedenih rješenja, ova aplikacija je besplatna za korištenje te samim time dostupnija svim budućim korisnicima koji nemaju potrebe za kompleksnijim alatima.

4. Aplikacija

4.1. Prezentacijski sloj

Kako bi se Caliburn Micro mogao koristiti u projektu, dodaju se reference kroz *NuGet* paket. Caliburn Micro koristi `Caliburn Micro i Caliburn Micro Core` referencu. Na sljedećoj slici prikazano je dodavanje navedenih referenci preko *NuGet Manager*-a.



Slika 4.1. Caliburn Micro reference

Početni prozor (engl. *window*) definira se kao `ShellViewModel.cs` i `ShellView.xaml`, a Caliburn se konfigurira u početnoj klasi nasljeđivanjem `BootstrapperBase` klase i definiranjem početnog prozora, što je vidljivo u sljedećem isječku kôda. `Initialize` metoda inicijalizira Caliburn Micro programski okvir. U `Configure` metodi se navode dodatna konfiguracija za *viewmodel*. U `OnStartup` metodi definira se početni ekran, odnosno `ShellViewModel`. [4]

```
public class Bootstrapper : BootstrapperBase
{
      public Bootstrapper()
      {
            Initialize();
      }
      protected override void Configure()
      {
            SimpleContainer container = new
            SimpleContainer();
      }
      protected override void OnStartup(object
      sender, StartupEventArgs e)
      {
            DisplayRootViewFor<ShellViewModel>();
      }
```

```
}
```

Kôd 4.1. Caliburn Micro inicijalizacija u početnoj klasi i postavljanje prozora

Kako bi se `Bootstrapper` klasa aktivirala, trebaju ju definirati u `App.xaml` klasi kao rječnik. Osim `Bootstrapper` klase referencira se i projekt `ThemeLibrary` u kojem je definirana tema, što je opisano niže u ovom poglavlju. `ToastNotifications` je referenca na *library* dodan putem *NuGet Package Manager-a* koji šalje notifikacije o uspješno odrađenim zadacima ili o pogreškama koje su se dogodile. Detaljna konfiguracija navedenih rječnika prikazana je u sljedećem kôdu.

```
<Application x:Class="CaliburnFinalProjectClient.App"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml
/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
xmlns:local="clr-namespace:CaliburnFinalProjectClient">
      <Application.Resources>
            <ResourceDictionary>
                  <ResourceDictionary.MergedDictionaries>
                        <ResourceDictionary>
                              <local:Bootstrapper x:Key=
                          "Bootstrapper"/>
                        </ResourceDictionary>

                        <ResourceDictionary Source="/
                        ThemeLibrary;
                        component/Theme.xaml"/>

                        <ResourceDictionary
                        Source="pack://application:,,,/
                        ToastNotifications.Messages;component
                        /Themes/Default.xaml" />
                  </ResourceDictionary.MergedDictionaries>
            </ResourceDictionary>
      </Application.Resources>
</Application>
```

Kôd 4.2. Konfiguracija `App.xaml` klase

Budući da Caliburn Micro ovisi o konvenciji imenovanja, *view* i *viewmodel* direktoriji koji se koriste, moraju se zvati **Views** i **ViewModels** kako bi ih Caliburn uspješno pronašao. Povezivanje *view*-a i *viewmodel*-a radi se na način da se iz punog naziva

CaliburnFinalProjectClient.ViewModels.ShellViewModel makne riječ **Model** i traži *CaliburnFinalProjectClient.Views.ShellView*. [4]

Aplikacija se izvršava u glavnom prozoru (`ShellView`) u kojem je definirana i navigacija. Sadržaj unutar prozora se mijenja kroz korisničke kontrole (engl. *User Control*) koje su predstavljene kao *view* s odgovarajućim *viewmodel*-om. `Window` i `UserControl` su komponente pisane u XML jeziku u kojima se kao atributi definiraju postavke za cijelu komponentu (npr. veličina slova, boja pozadine, klasa, reference, itd.). U njima se postavlja sadržaj (engl. *content*), koji se može postaviti samo jednom, kroz panel, u kojem se definiraju druge kontrole. Popis i opis panela koji se koriste u aplikaciji prikazan je u tabličnom obliku.

Tablica 4.1. Opis WPF panela korištenih u projektu

Naziv	**Opis**
WrapPanel	Kontrole se pozicioniraju horizontalno ili vertikalno ovisno o dostupnom prostoru te ukoliko nema prostora, kontrole prelaze u novi red.
StackPanel	Sličan `WrapPanel`-u, ali kontrole ne prelaze u novi red ukoliko nema mjesta.
Grid	Tablični prikaz, definira se kroz retke i kolone kojim se određuje fiksna ili automatska veličina. Automatska veličina može biti predstavljena kao auto ili *. Auto označava povećavanje ovisno o veličini kontrole, a * širenje ovisno o dostupnom prostoru.
DockPanel	Pozicioniranje kontrola gore, dolje, lijevo ili desno. Zadnja dodana kontrola kojoj nije definirana pozicija preuzima slobodan prostor.

Kontrole se povezuju s *viewmodel*-om u jednom ili u oba smjera tako da se postavi Mode koji može biti `Default`, `OneTime`, `OneWay`, `OneWayToSource` i `TwoWay`. Glavno povezivanje je u oba smjera. Ukoliko je definira `DependecyObject` u kontroli, Caliburn automatski povezuje `x:Name` svojstvo s *viewmodel*-om. Tako će npr.za `Label x:Name` napraviti povezivanje s `Content` svojstvom te je tako `<Label x:Name="Name"/>` istovjetno s `<Label Content="{Binding Path=Name, Mode=TwoWay"/>`. U *viewmodel*-u se definiraju svojstva koja će se povezivati s *view*-om, koji prima podatke o promjeni, ukoliko se pozove `NotifyOfPropertyChange` koji dolazi iz `Screen` klase. `Screen` je osnova konstrukcija s životnim ciklusom za aktivacijsku i deaktivacijsku logiku. `Conductor` je konstrukcija koja aktivira i deaktivira

Screen-ove za koje je zadužena pozivom metode ActivateItem. ActivateItem kao parametar prima *viewmodel* te aktivira ContentControl kontrolu s nazivom ActiveItem.U sljedećem primjeru je prikazan CompanyViewModel koji nasljeđuje Screen te potpuni *property* Company s *get* i *set* metodama. U *set* metodi se poziva metoda NotifyOfPropertyChange, koja obavještava o promjenama na javnom Company objektu.

```
class CompanyViewModel : Screen {
private Company;
public Company
    {
        get { return company; }
        set { company = value;
              NotifyOfPropertyChange(() => Company);
            }
     }
}
```

Kôd 4.3. Primjer potpunog svojstva u CompanyViewModel -u

Click event na gumbu se može automatski definirati kroz x:Name ili kao akcija. U UserControl ili Window komponenti definira se atribut xmlns:cal=http://www.caliburnproject.org i na kontroli se poziva akcija putem Attach svojstva. U Attach svojstvu se navodi Event kontrole koji se poziva i Action koji definira metodu u *viewmodel*-u. U sljedećem primjeru je prikazano definiranje PasswordChanged *event-a* i referenciranje na Change metodu iz *viewmodel*-a. Metoda se poziva svaki put kad se promijeni vrijednost unesene lozinke i postavlja vrijednost Password *property*-ja. PasswordBox komponenta pohranjuje lozinku kao SecureString kako se vrijednost lozinke ne bi mogla iščitati iz memorije. U metodi Change se iščitava prava vrijednost lozinke.

```
<PasswordBox cal:Message.Attach=
"[Event PasswordChanged] = [Action Change($source)]"
Width="200" Margin="5,5,5,15"/>
```

Kôd 4.4. PasswordChanged *event*

U Action definiranu metodu se mogu proslijediti parametri kao što su tekst ili predefinirani parametri koji daju više informacija o kontroli koja aktivira *event*. U tabličnom prikazu su

navedeni predefinirani parametri koji se mogu proslijediti prilikom aktiviranja *event*-a te objašnjenje parametara.

Tablica 4.2. Predefinirani parametri

Predefinirani parametri	**Opis**
$eventArgs	detalji o pokrenutom *event*-u
$dataContext	vrijednost objekta koji se veže uz kontrolu
$source	kontrola koja je aktivirala *event*
$view	`UserControl`-a ili `Window` koji se veže uz *viewmodel*
$executionContext	sadrži sve gore navedene informacije
$this	vrijednost predodređenog svojstva

Tema se nalazi u projektu **ThemeLibrary** koji je tipa *WPF User Control Library*. ThemeLibrary se sastoji od rječnika definiranih u *.xaml* datotekama u kojem su definirani stilovi i boje. Svi rječnici su navedeni u glavnom rječniku `Theme.xaml` kojeg referencira `App.xaml` glavnog projekta.

Stilovi se definiraju ovisno o tipu (`TargetType`) kontrole te se identificiraju prema zadanim ključevima. Ukoliko ključ nije naveden, stil se odnosi na sve kontrole tog tipa koje nemaju navedeno `Style` svojstvo. `Setter`-i definiraju izgled kontrole. `TemplateBinding` omogućuje mijenjanje izgleda kontrole iz aplikacije. Na sljedećem primjeru je prikazano stiliziranje gumba koji nasljeđuje stil `BtnTransparent` u kojem su `Background` i `BorderBrush` bez boje (prozirni), `BorderThickness` je 0 i sadržaj je centriran. Na `BtnIcon` je postavljena zadana širina i veličina fonta jer `FontAwesome` veličina ikone ovisi o veličini fonta. Boja ikone se referencira iz rječnika `ThemeColours.xaml` u kojem su navedene boje se koriste u aplikaciji. Na taj način samo izmjenom boja u jednoj datoteci mijenja se boja cijele aplikacije i komponenti. Budući da je `FontAwesome` ikona ugniježđena komponenta koristi se `TemplatedParent` koji označava da se treba referencirati na `Tag` koji je definiran na gumbu, odnosno na roditelju.

```
<Style TargetType="Button" BasedOn="{StaticResource
 BtnTransparent}" x:Key="BtnIcon">
    <Setter Property="Width" Value="20"/>
    <Setter Property="FontSize" Value="15"/>
    <Setter Property="Foreground" Value="{StaticResource
     Accent}"/>
```

```
        <Setter Property="Margin" Value="5
        <Setter Property="Template">
              <Setter.Value>
                    <ControlTemplate>
                          <fa:FontAwesome
                          x:Name="icon"
                          Icon="{Binding Path=Tag,
                    RelativeSource={RelativeSource
                           TemplatedParent}}"
                    Foreground="{TemplateBinding Foreground}"
                    FontSize="{TemplateBinding FontSize}"
              </Setter.Value>
        </Setter>
  </Style>
```

Kôd 4.5. Primjer stila za gumb koji se prikazuje kao FontAwesome ikona

Niže je prikazan primjer korištenja stila iz rječnika na `Button` kontroli. Gumb se prikazuje kao `FontAwesome` ikona koja se mijenja pozivom *click eventa* `ViewRate` koji je definiran u *viewmodel*-u. Ikone se izmjenjuju i povezane su s `IconExpand` *property*-jem u *viewmodel*-u te imaju vrijednost `ChevronRight` ili `ChevronDown`.

```
<Button x:Name="ViewRate" Style="{StaticResource
 BtnIcon}" Tag="{Binding Path=IconExpand}"/>
```

Kôd 4.6. Gumb sa stilom `BtnIcon`

UI je zadužen za **validaciju** unesenih vrijednosti. Za validacije je korišten `FluentValidation` *library* koji je dodan u projekt preko *NuGet Manager-a*. `FluentValidator` omogućuje postavljanja pravila putem metode `RuleFor` koji prima lambda izraz u kojem se definira na koji *property* se odnosi. Metoda `RuleFor` i ostale validacijske metode (`Validate`, `ValidateAndThrow`, itd.) su dostupne nasljeđivanjem apstraktne klase `AbstractValidator<T>` koja prima tip objekta na koji se validator odnosi. U sljedećem primjeru je prikazano pravilo za `Name` *property* modela `Company` koje označava kako je naziv tvrtke obavezno polje. Ukoliko postoje dodatna pravila, `Cascade` svojstvo omogućuje prikaz pogreške samo za prvo pravilo koje nije zadovoljeno.

```
RuleFor(c => c.Name)
      .Cascade(CascadeMode.StopOnFirstFailure)
```

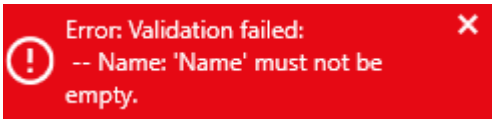
```
.NotEmpty();
```

Kôd 4.7. Pravilo za `Name` *property* modela `Company`

Validacije se pozivaju preko pomoćne metode u `Toast` klasi. U sljedećem primjeru je prikazana `ValidateData` generička metoda, za objekt tipa `T`, koja prima validator klasu i objekt koji klasa treba validirati. Metoda validira objekt i ukoliko svi kriteriji nisu zadovoljeni, baca pogrešku koja se ulovi u `try catch` bloku te se korisniku prikazuje notifikacija o nastaloj pogrešci. Primjer notifikacije za `Company` model i *property* `Name` je prikazan je na slici niže. Na notifikaciji je prikazano kako vrijednost za *property* `Name` ne smije biti prazna.

```
public void ValidateData<T>(AbstractValidator<T>
 validator, T data)
{
     if (validator == null)
           return;
     validator.ValidateAndThrow(data);
}
```

Kôd 4.8. Generička metoda za validaciju



Slika 4.2. Notifikacija o nastaloj pogrešci

Osim obaveznih polja, validiraju se polja OIB, MBS i EUID. OIB mora sadržavati 11 numeričkih znakova i u bazu se sprema kao tekst tako da ne može doći do neispravnosti ukoliko započinje s znamenkom 0. MBS mora sadržavati 9 znamenaka te se kao i OIB sprema kao tekst. EUID se sastoji od 3 obavezna dijela: oznake države, identifikatora registra i registracijskog broja subjekta te od neobavezne verifikacijske znamenke. Format za Hrvatsku je HRSR.<MBS broj>. [22]

4.2. Sloj poslovne logike

Sloj poslovne logike prisutan je i u serverskoj i u klijentskoj aplikaciji. Klijentske klase su definirane u direktoriju *BL* u kojem su klase koje šalju zahtjeve prema serveru. Serverske klase su definirane u paketima *controller* i *service*.

Svi zahtjevi s klijentske strane pozivaju se preko globalnog `HttpClient` objekta. `HttpClient` objekt definiran je pod nazivom `ServerRest`, u statičkoj klasi `Server`, kao što je prikazano u sljedećem primjeru. `ServerRest` se inicijalizira u metodi `InitServerConnection` koja se poziva samo jednom, odnosno kad se aplikacija pokrene te se prikaže `LoginView`. URL se postavlja kroz URI *property* `BaseAddress`, a vrijednost URL-a je `http://localhost:8080`. Osim adrese, navedeno je i zaglavlje, koje označava u kojem obliku aplikacija može primiti odgovor što je u ovom slučaju samo `"application/json"`.

```
internal static void InitServerConnection(string serverUrl)
{
        ServerRest = new HttpClient();
        ServerRest.BaseAddress=new Uri(serverUrl);

        ServerRest.DefaultRequestHeaders.Accept.Clear();
        ServerRest.DefaultRequestHeaders.Accept.Add( new
        MediaTypeWithQualityHeaderValue(JSON));
}
```

Kôd 4.9. Postavljanje globalnog `HttpClient` objekta

CRUD operacije su implementirane u apstraktnoj klasi `AbstractService` s generičkim tipom `T`. Sve metode su označene oznakom `virtual` koja omogućuje *override* metoda, ukoliko je potrebno. Kroz konstruktor se prosljeđuje naziv rute koji je definiran u serverskom kontroleru (npr. `UserController`). Svi zahtjevi prema serveru šalju se asinkrono i vraćaju `Task` s generičkom tipom. Asinkrone metode su označene s ključnom riječi `async`, a takve metode moraju vraćati `void`, `Task` ili `Task<T>`. Kako bi se asinkrona metoda sinkronizirala, koristi se ključna riječ `await` koja označava da se mora pričekati do kraja izvršavanja operacije. Ukoliko je odgovor sa servera vratio uspješan statusni kôd 2xx, čita se `Content` iz odgovora s definiranim POCO objektom koji očekuje. Ukoliko odgovor nije bio uspješan, čita `ErrorModel` POCO i baca `Exception` u koji se prosljeđuje razlog (`Message`) neuspješnog zahtjeva. U sljedećem primjeru se može vidjeti implementacija asinkrone `Create` metode koja vraća `Task<T>`. Metoda `PostAsJsonAsync` vraća odgovor tipa `T`, ukoliko je zahtjev bio uspješan, odnosno baca pogrešku. Metoda se u `using` bloku koji zatvara konekciju nakon izvršavanja.

```
internal virtual async Task<T> Create(T obj)
{
```

```
        using (HttpResponseMessage res = await
        Server.ServerRest.PostAsJsonAsync<T>(url, obj))
        {
              if (res.IsSuccessStatusCode)
              {
                    return await res.Content.ReadAsAsync<T>();
              }
              else
              {
                    throw new Exception((await
                    res.Content.ReadAsAsync<ErrorModel>())
                    .Message);
              }
        }
  }
```

Kôd 4.10. Asinkrona metoda `Create`

Osim povezivanja sa serverom, poslovni sloj klijenta implementira i logiku za kreiranje predložaka. Kako bi se Word mogao koristiti kroz aplikaciju, dodana je referenca **Microsoft.Office.Inrerop.Word** koja omogućuje otvaranje, uređivanje i brisanje Word datoteka. Word je implementiran u klasi `WordTemplate` koja implementira *interface* `IDisposable`. `IDisposable` omogućuje zatvaranje datoteke pozivom `Dispose` metode u `using` bloku ili po potrebi. Word se koristi tako što se kreira `Application` objekt koji predstavlja Word aplikaciju i može imati više `Document` objekata koji predstavljaju *.doc* i *.docx* dokumente. Za njih treba biti definirana putanja (engl. *path*) kako bi se dokumenti otvorili u pozadini prije početka korištenja. Kreiranje predložaka se izvodi po principu „pronađi i zamjeni“, odnosno definirani su *placeholder*-i koji odgovaraju nazivu *property*-ja s dodanim prefiksom „<“ i sufiksom „>“. Niže je prikazana implementacija metode `FindAndReplace` koja pronalazi riječ, razlikuje velika i mala slova, pri čemu ne mijenja format dokumenta te zamjenjuje sve slučajeve s novom vrijednošću.

```
public void FindAndReplace(object findText,
object replaceWithText)
{
       word.Selection.Find.Execute(FindText: findText,
        MatchCase: true, MatchWholeWord: true,
        Replace: Word.WdReplace.wdReplaceAll,
        ReplaceWith: replaceWithText, Format:false);
```

```
}
```

Kôd 4.11. Metoda za zamjenu *placeholder*-a u tekstu

Naziv i vrijednost *placeholder*-a se dohvaćaju pomoću refleksije, što je vidljivo u sljedećem primjeru kôda. Metoda `ReplaceData` prolazi `foreach` petljom kroz sve *property*-je u `data` objektu i poziva metodu `FindAndReplace`. `Id` je opcionalni parametar koji kao zadanu vrijednost ima prazan `string`. `GetType` metoda vraća tip objekta (npr. `User`), a `GetProperties` vraća listu `PropertyInfo` objekata koji sadrže sve informacije o *property*-jima. Ukoliko je *property* primitivan tip, zamjena je vrlo jednostavna, odnosno vrijednost se pretvara u `string` (ukoliko je nekog drugog tipa) i zamjenjuje. Ukoliko je tip `object`, izvršava se provjera da li je objekt kolekcija ili objekt. Kad se utvrdi tip objekta, metoda se rekurzivno zove s proslijeđenim objektom, dok se ne dohvate i zamjene sve vrijednosti. Metode za provjeru tipa su implementirane u statičkoj klasi `ReflectionUtils`.

```
public void ReplaceData(object data, string id="")
{
      try
      {
            foreach (PropertyInfo prop in data.GetType()
            .GetProperties())
            {
                  if ( ReflectionUtils.IsSimpleType(
                  prop.PropertyType))
                  {
                        FindAndReplace(ReflectionUtils
                        .GetReplacerText(prop.Name, id),
                        prop.GetValue(data, null)
                        .ToString());
                        continue;
                  }
                  else if (ReflectionUtils
                  .IsCollection(prop.PropertyType))
                  {
                        ReplaceData(ReflectionUtils
                        .GetListType(prop.PropertyType),
                         id);
                  }
                  else
```

```
                {
                        ReplaceData(prop.PropertyType,
                         id);
                }
            }
        }
        catch (Exception)
        {
        throw;
        }
    }
```

Kôd 4.12. Metoda za dohvat naziva i vrijednosti *placeholder*-a

Budući da se nad svim dokumentima baze vrše CRUD operacije, servis i kontroler su definirani kroz apstraktne klase u kojima je definirana zajednička logika, a u svrhu reduciranja dupliciranog kôda, odnosno kako bi se kôd lakše održavao. Kao što je navedeno u poglavlju 2, anotacija `@Service` samo označava klasu poslovne logike. Apstraktna klasa servisa označena je s generičkim tipom `T`. Repozitorij se sa servisom povezuje kroz *dependency injection* u konstruktoru. Niže je prikazana metoda za dohvat svih podataka iz baze kroz repozitorij.

```
public abstract class AbstractService<T>
{
      private MongoRepository<T, String> repository;

      public AbstractService(MongoRepository repository)
      {
            this.repository = repository;
      }
      public List<T> getAll()
      {
            return repository.findAll();
      }
}
```

Kôd 4.13. Isječak apstraktne klase servisa s generičkim tipom

U sljedećem isječku prikazan je `UserService` koji nasljeđuje gore navedenu apstraktnu klasu i prosljeđuje `UserRepository` kroz konstruktor. `UserService` je anotiran kao `@Service` kako bi se mogao povezati s repozitorijem pomoću `@Autowired` anotacije.

```
@Service
public class UserService extends AbstractService<User> {

    @Autowired
    public UserService(UserRepository userRepository) {
        super(userRepository);
    }
}
```

Kôd 4.14. `UserService` servis

Budući da se radi o REST servisu, sve kontroler klase su označene kao `@RestController`. Ukoliko bi kontroler bio označen samo kao `@Controller`, prilikom mapiranja metoda bi bilo potrebno dodati `@ResponseBody` anotaciju. U sljedećem primjeru je prikazano mapiranje GET metoda. Vrijednost u zagradi predstavlja `value`, odnosno dio koji se prikazuje u URL-u. Zbog jednostavnosti, ostale metode nisu prikazane, ali funkcioniraju na vrlo sličan način. Kontroler ima apstraktnu metodu `getService` koju definira servis koji nasljeđuje apstraktnu klasu i postavlja servis s kojim se kontroler povezuje. Prikazane su dvije GET metode koje vraćaju objekt u JSON obliku i statusni kôd 200 (OK).

```
public abstract class AbstractController<T extends
AbstractId> {

    protected abstract AbstractService<T> getService();

    @GetMapping
    public ResponseEntity<List<T>> getAll() {
        ResponseEntity<List<T>> res = new
        ResponseEntity<>(getService().getAll(),
        HttpStatus.OK);

        return res;
        }
   @GetMapping("/{id}")
```

```
    public ResponseEntity<T> getById(@PathVariable
    String id) {
        return new ResponseEntity<>(getService()
        .getById(id), HttpStatus.OK);
    }
}
```

Kôd 4.15. Apstraktni generički kontroler

Prilikom implementacije kontrolera navodi se `@RestController` i `@RequestMapping` na razini klase te se definira servis koji se koristi za taj kontroler. Koriste se samo 2 URL-a *http:localhost:8080/users* i *http:localhost:8080/{id}* te se ispravne metode pozivaju ovisno o definiranoj HTTP metodi kao što je prikazano na kontroleru za `User` model.

```
@RestController
@RequestMapping("/users")
public class UserController extends
AbstractController<User> {
    @Autowired
    UserService userService;

    @Override
    protected AbstractService<User> getService() {
        return userService;
    }
}
```

Kôd 4.16 `UserController`

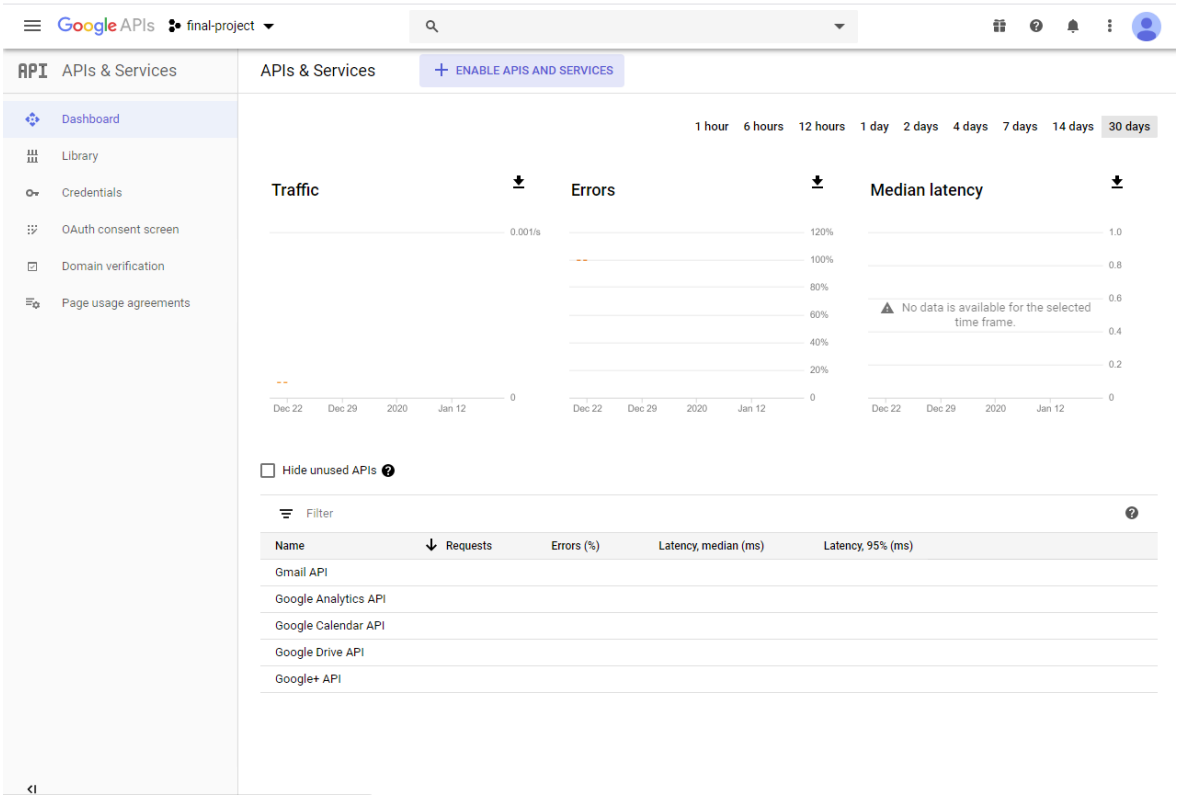
REST servis može vraćati objekt ili `ResponseEntity`. `ResponseEntity` daje veću fleksibilnost jer se osim samih podataka definira statusni kôd, a mogu se definirati i dodatna zaglavlja. [7] U slučaju uspješnog zahtjeva poslužitelj vraća statusni kôd 2xx. Ukoliko dođe do pogreške, poslužitelj baca neprovjerenu iznimku, odnosno `RuntimeException`. Pogreške se definiraju pomoću `@ExceptionHandler` i mogu se definirati u svakom kontroleru zasebno ili u globalnom kontroleru `@RestControllerAdvice`. [14] Kod globalnog kontrolera definira se iznimka (ili polje iznimki) koja vraća rezultat koji je u ovoj aplikaciji tipa `ErrorModel`. Niže je prikazana uporaba `NoEntityException` iznimke u klasi `ExceptionHandling` koja *extend*-a `RuntimeException` i vraća statusni kôd 204 (*No Content*)

```
@ExceptionHandler(NoEntityException.class)
public ResponseEntity<ErrorModel>
handleNoEntityException(NoEntityException ex, WebRequest req)
{
    return createResponse(HttpStatus.NO_CONTENT, ex, req);
}
```

Kôd 4.17. Hvatanje `NoEntityException` iznimke

4.2.1. Google

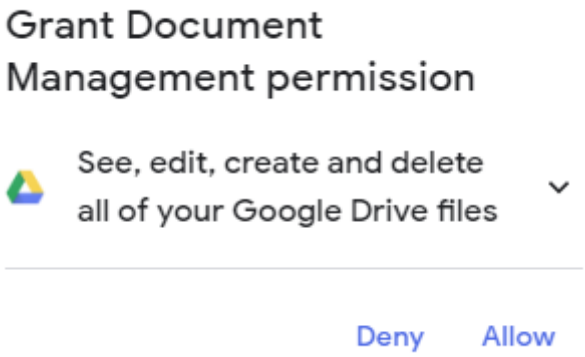
Za korištenje Google API-ja, aplikacija mora biti registrirana na **Google API Console** portalu i mora se dati dopuštenje svim API-jima koje koristi kao što je vidljivo na sljedećoj slici. Preko portala se može pratiti promet aplikacije, odnosno prelazi li zahtjevi zadanu kvotu. Također se kreiraju *credential*-i, postavlja se zasebni ekran za privole i URL-ovi za preusmjeravanje nakon uspješne ili neuspješne registracije. Za rad s osjetljivim podacima Google treba odobriti aplikaciju, ali budući da aplikacija radi na *localhost*-u, smatra se da je u razvojnom procesu te kasnije korisnik mora odobriti pristupanje „nesigurnoj aplikaciji“.



Slika 4.3. Početna stranica registrirane aplikacije

Google API koristi OAuth 2.0 autorizaciju i Java Web Token-e. Podaci za pristup aplikaciji (engl. *credentials*) se generiraju kroz Google API Console portal i preuzimaju u JSON

obliku. Pristupni podaci sadrže informacije o tipu akreditacije (web, Android, iOS, Chrome aplikacija ili ostalo), `client_id`, `project_id`, `auth_uri`, `token_uri`, `auth_provider`, `client_secret`, `redirect_uris` i `javascript_origins`. JSON datoteka se pohranjuje na server i učitava kad se korisnik želi prijaviti. Akreditacijski podaci se učitavaju kao `GoogleClientSecrets` objekt koji se prosljeđuje u `GoogleAuthorizationCodeFlow` koji generira URL za Google prijavu. Nakon što se korisnik prijavi i da privolu za rad s njegovim podacima, vraća se odgovor s tokenom za pristup u JSON formatu. Odgovor sadrži: `access_token`, `refresh_token`, `expires_in`, `scope` i `token_type` koji se kasnije učitava kao `Credential` POJO. `Refresh_token` služi za ponovni dohvat `access_token`-a kad istekne. Ukoliko `refresh_token` istekne, korisnik mora ponovno dati privolu za rad. `Refresh_token` može isteći ukoliko je korisnik maknuo dozvolu za rad s aplikacijom, token nije korišten zadnjih šest mjeseci, korisnik je promijenio lozinku a dozvoljen je rad s Gmail-om ili korisnik ima 50 aktivnih tokena. U zadnjem slučaju najstariji token se briše i zamjenjuje novim bez upozorenja. `Refresh_token` se pohranjuje u bazu kao `googleToken`. Ekran za davanje privole, za pristup Google Drive-u, prikazan je na idućoj slici.



Slika 4.4. Korisnik mora dati privolu za pristup podacima koji su definirani kroz *scopes*

Pristup bilo kojem Google API-ju mora biti autoriziran kroz `GoogleAuthorizationCodeFlow` u koji se pohranjuju podaci o svim aktivnim korisnicima. Metoda `createAndStoreCredential` prima `GoogleResponseToken` i identifikator korisnika. Identifikator korisnika je definiran kao `accessToken` koji se generira kad se korisnik prijavi, što je detaljnije objašnjeno u poglavlju 4.5. U sljedećem primjeru je prikazana metoda `getCredentials` za dohvat

credential-a kroz `flow`. `flow` omogućuje dohvat podataka kroz metodu `loadCredentials` koja prima ključ pod kojim je korisnik spremljen u mapu. U ovom slučaju je to `accessToken`. Kad se korisnik odjavljuje, dohvaća se mapa kroz metodu `getCredentialDataStore` i podaci se uklanjaju iz mape. Ukoliko je `credential` `null`, korisnik nije prijavljen i baca se `GoogleAuthorizationException` s porukom o iznimci. Ukoliko korisnik postoji, vraća se objekt `Credential` s podacima o tokenu i tipu autentifikacije.

```
public Credential getCredentials(String accessToken)
 throws IOException {
      Credential credential =
      flow.loadCredential(accessToken);

      if(credential == null){
            throw new GoogleAuthorizationException(
            "Cannot get credentials for user: %s
            ",accessToken);
      }
      return credential;
}
```

Kôd 4.18. Metoda za dohvat pohranjenih podataka o tokenu koja vraća `Credential`

HTTP pozivi prema Google Drive-u se izvršavaju putem `Drive` klase koja kreira URI za pozivanje *drive* API-ja. `Builder` prima tip konekcije, `JsonFactory` za parsiranje JSON-a, `credential` i naziv registrirane aplikacije s Google API Console portala. Metoda `build` kreira zahtjev s navedenim parametrima i vraća `Drive` instancu.

```
HttpTransport httpTransport = GoogleNetHttpTransport
.newTrustedTransport();

Credential credential = authService.
getCredentials(accessToken);

drive = new Drive.Builder(httpTransport,
GoogleConstants.JSON_FACTORY, credential)
.setApplicationName(GoogleConstants.APPLICATION_NAME).build()
;
```

Kôd 4.19. Inicijalizacija `Drive` klase

Dohvaćeni podaci se serijaliziraju u `File` i `FileList` POJO. U `builder`-u se zasebno definiraju polja kroz metodu `setFields` od kojih aplikacija prihvaća: `id`, `name`, `mimeType`, `createdTime`, `modifiedTime`, `ownedByMe`, `shared` i `driveId`. Aplikacija omogućuje dohvat direktorija, dohvat i brisanje datoteka te dohvat i pohranu prijašnjih verzija dokumenata. Sve verzije se pohranjuju kao BLOB. Za filtriranje direktorija i tipova datoteka postavljen je `mimeType` filter, odnosno dohvaćaju se samo *.pdf, .doc* i *.docx* datoteke ili direktoriji. Tip se definira kroz `mimeType` te se tako *.pdf* definira kao `mimeType = 'application/pdf'`. Može se dodavati više `mimeType`-ova koji se grupiraju unutar zagrada i odvojeni su logičkim operatorom `and` (i) ili `or` (ili). U aplikaciji je napravljen `enum` za preglednije filtriranje koji sadrži metode `is` za MIME tipove tog tipa, odnosno `not` za MIME tipove koji nisu tog tipa. U sljedećem primjeru je prikazana implementacija metode `is`, `enuma`-a `MimeType`, koja formatira `String`, odnosno postavlja vrijednost naziv i vrijednost MIME tipa.

```
public String is() {
    return String.format("%s = '%s'", MIME_TYPE, type);
}
```

Kôd 4.20. `is` metoda `enum`-a `MimeTypes`

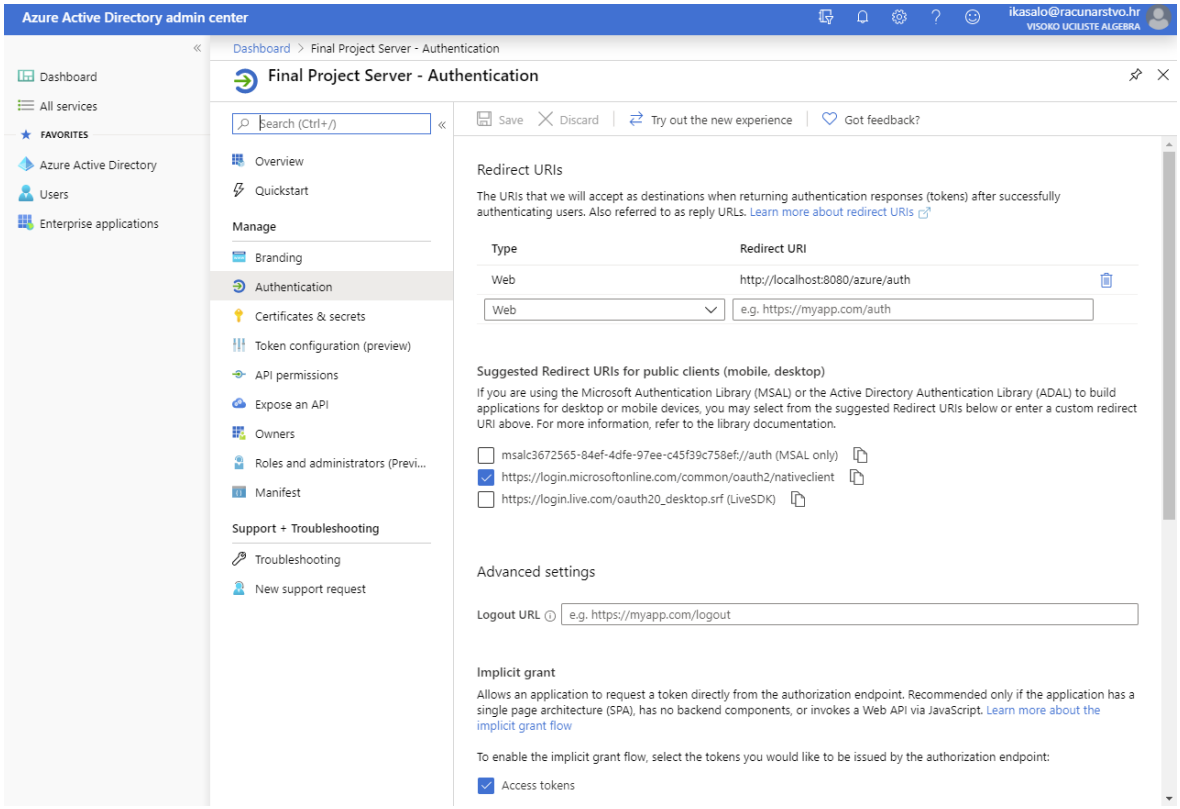
Google Calendar se inicijalizira slično kao i *drive* putem pomoćne klase `Calendar` za kreiranje HTTP poziva. Korisnik ima uvid u *evente* u svom kalendaru koji se označava kao *primary*. Korisnik može kreirati, ažurirati, brisati i pregledavati *evente*, a oni su predstavljeni kao `Event` POJO. Lista *evenata* se dohvaća putem filtriranja od-do određenog datuma. `Calendar` datumi su označeni kao `DateTime` iz Google API *library*-ja. U aplikaciji se koristi `LocalDateTime` te je potrebno napraviti pretvorbu u `java.util.Date` tako što se `LocalDateTime` pretvara u `ZonedDateTime` sa zonom koja je definirana kao zadana zona na računalu. Nakon pretvorbe se zove metoda `from` u koju se prosljeđuje `ZonedDateTime`.

Ostale metode u servisu se pozivaju prosljeđivanjem kalendar ID-ja, odnosno oznake *primary*, *event* ID-ja i/ili `Event` objekta.

4.2.2. Microsoft

Za korištenje Microsoft (Azure) API-ja, aplikacija mora biti registrirana na **Azure Active Directory** portalu. Kao što je vidljivo na sljedećoj slici, na portalu se definira naziv

aplikacije koji će se prikazivati korisniku, *client ID* i *tenant ID* koji služi kao obilježje grupacije (npr. racunarstvo.hr). Na slici niže je prikazano postavljanje URL za preusmjeravanje na serverski URL nakon uspješne ili neuspješne prijave, odabir tipa tokena (*access tokens, ID tokens)* te mogu li aplikaciji pristupiti samo korisnici koji su u istoj organizaciji (*single tenant*) ili iz svih organizacija (*multi tenant*). Također se daju dozvole za pristupanje Azure API-jima. U ovom slučaju aplikacija ima pristup samo na **Microsoft Graph** koji uključuje sve dozvole za kalendar i *drive* te čitanje podataka o prijavljenom korisniku.

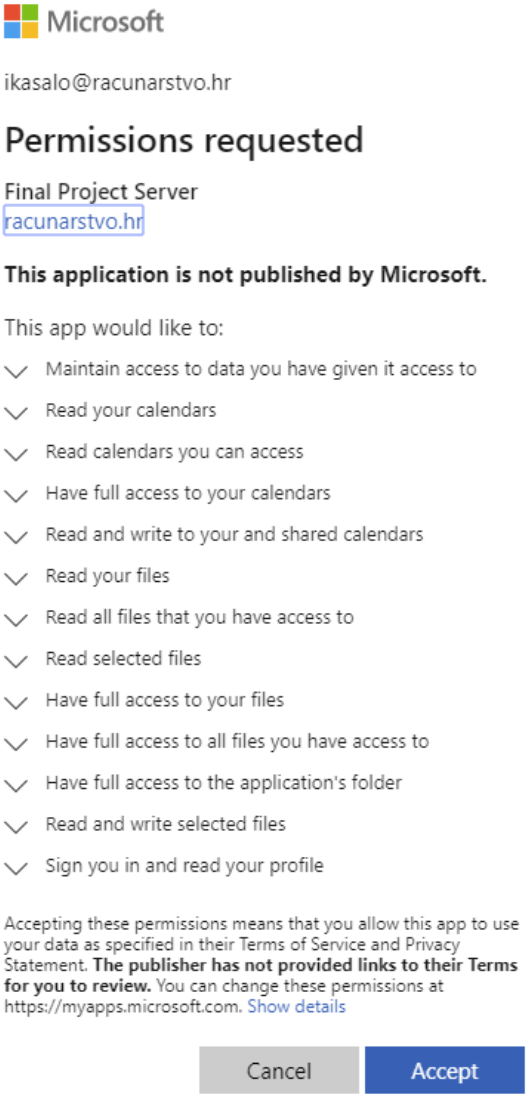


Slika 4.5. Registrirana aplikacija na Azure portalu

Kao i Google, Azure koristi OAuth2 za autorizaciju te je postupak prijave vrlo sličan. Kako bi korisnik pristupio aplikaciji u `AzureAuthService` servisu, kroz metodu `login` se generira URL putem kojeg se korisnik prijavljuje na Microsoft portal. Nakon uspješne prijave se šalje zahtjev na server s generiranim *access tokenom*. *Library* vraća `IAuthenticationResult` koji sadrži *access token* i *refresh token*. *Refresh token* je prisutan u svakom zahtjevu, ali je skriven od korisnika.

Kako bi korisnik povezao aplikaciju s Microsoft računom mora dati privolu za čitanje, kreiranje, ažuriranje i brisanje *evenata* u kalendaru i datoteka na One Drive-u. Također, aplikacija ima pristup korisnikovom profilu iz kojeg čita email adresu s kojom se korisnik

prijavljuje na portal. Zahtjev za davanje privole s navedenim opcijama prikazan je na slici niže.



Slika 4.6. Davanje privole za pristup aplikaciji

4.2.3. HNB

HNB API nema zaseban *library* za pozivanje metoda te se koristi `RestTemplate`. `RestTemplate` omogućuje pristupanje drugim API-jima te automatski mapira JSON u POJO. Svi zahtjevi se pozivaju GET metodom nad URL-om **http://api.hnb.hr/tecajn/v1**, a podaci se vraćaju u JSON ili XML obliku. Specifikacija HNB API-ja navodi kako je tečaj valuta iskazan u kunama, datumi su formatirani u obliku **dd.MM.yyyy.** i valute se upotrebljavaju prema standardu ISO 4217.

U sljedećem primjeru prikazana je metoda `getCurrencyList` koja se koristiti u svim zahtjevima prema HNB API-ju. Metoda kreira `RestTemplate` objekt koji kroz metodu `exchange` šalje zahtjev na zadani URL. Određuje se tip poziva (u ovom slučaju GET), dodaju se zaglavlja (u ovom slučaju `null`) i definira se tip objekta koji se očekuje u odgovoru. Budući da su valute odvojene decimalnim zarezom, a ne decimalnom točkom, valute se pretvaraju iz `String`-a u `double`. Budući da datum nije po ISO 8601 standardu koje je u formatu **yyyy-MM-dd'T'HH:mm:ssXXX,** `String` vrijednost datuma se pretvara u `LocalDate`.

```
private Optional<List<Currency>> getCurrencyList(String url) {
      RestTemplate restTemplate = new RestTemplate();

      ResponseEntity<List<Currency>> res = restTemplate
      .exchange(url, HttpMethod.GET, null, new
      ParameterizedTypeReference<List<Currency>>() {
      });

      List<Currency> list = res.getBody();
      Optional<List<Currency>> opt = Optional.ofNullable(list);
      opt.ifPresent((l -> l.forEach(c -> {
            formatRateNumber(c);
      LocalDate date = DateUtils.
      getDateFromString(c.getDatumPrimjene(),
       DateUtils.FORMAT_HR);
      c.setDate(date.atStartOfDay());
      })));
      return opt;
}
```

Kôd 4.21. Metoda za poziv API-ja koja vraća `Optional` listu valuta

4.2.4. SudReg

Za korištenje API-ja Sudskog registra potrebna je registracija i pretplata na API. Prilikom pretplate se kreira pretplatnički ključ koji se dodaje u zaglavlje **Ocp-Apim-Subscription-Key** prilikom slanja zahtjeva. Kao i HNB API, SudReg API nema zaseban *library* te se svi pozivi odvijaju putem `RestTemplate`-a. Aplikacija koristi samo jednu metodu GET `subjekt_detalji` na URL-u **https://sudreg-api.pravosudje.hr/javni/.** API vraća podatke u JSON, XML ili HTML obliku. Zbog kompleksnosti rezultata podaci se „ručno“

deserijaliziraju u POJO. U sljedećem primjeru je prikazana deserijalizacija za čvor OIB koji je u JSON shemi definiran kao broj. Budući da je prikazan kao broj, vodeća 0 se gubi (ukoliko postoji) te vraća rezultat koji ima 10 znamenaka umjesto 11. Zbog toga se rezultat pretvara u `String` te se 0 i dobiveni rezultat spajaju metodom `concat`. `getNode` je metoda koja vraća `Optional<JsonNode>` te ukoliko čvor postoji nastavlja s deserijalizacijom vrijednosti i postavljanjem u POJO.

```
getNode(jsonNode, "oib").ifPresent(e -> {
    String oib = e.asText();
    oib = oib.length() != 11 ? addZero.concat(oib) : oib;
    company.setOib(oib);
});
```

Kôd 4.22. OIB čvor

4.3. Sloj pristupa podacima

Za korištenje MongoDB *driver*-a i Spring Data ODM-a dodan je sljedeći *dependency* koji sadrži *mongodb-driver, spring-boot-starter* i *spring-data-mongodb library*-je.

```
<dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>
            spring-boot-starter-data-mongodb
      </artifactId>
</dependency>
```

Kôd 4.23. Referenca na Mongo ODM JAR u pom.xml datoteci

Podaci o bazi (naziv baze, korisnik, lozinka, *host* i *port*) se u Spring Boot-u definiraju u **application.properties** datoteci. Od Mongo verzije 3 na više, *port* i *host* više nije potrebno zasebno definirati već se definiraju kao URI. Niže su prikazana dva načina spajanja na bazu, lokalno i u oblaku (Atlas). Atlas koristi SRV koji identificira klaster (engl. *cluster*), tj. *host* i *port*. SRV omogućuje korištenje liste klastera, odnosno omogućuje korisniku spajanje na bilo koji klaster, ukoliko je jedan (ili više) nedostupan. [12] URI svojstva se definiraju kako slijedi: korisnik, lozinka i klaster. `database` svojstvo definira naziv baze, ali naziv može biti i definiran kroz `uri` svojstvo kao što je prikazano u primjeru lokalne konekcije.

```
spring.data.mongodb.uri = mongodb+srv://Ivana:<lozinka>@final-
project-n7fnv.azure.mongodb.net/test?retryWrites=true&w=majority
```

```
spring.data.mongodb.database = final-project-test
```

Kôd 4.24. Konekcija na Atlas (baza u oblaku)

```
spring.data.mongodb.uri=mongodb://myUser:user@localhost:27017/final-project
```

Kôd 4.25. Lokalna konekcija

Shema baze se definira kroz modele, odnosno objekte koji se koriste u aplikaciji. Dokumenti se kreiraju i mapiraju anotacijom `@Document`, na razini klase, gdje se može dodatno definirati i kolekcija te bi puni oblik bio npr. `@Document(collection="Users")`. Svaki dokument je definiran jedinstvenim identifikatorom koji se označava kao `@Id` i Mongo ga automatski generira prilikom kreiranja. Mongo generira JSON nazive kako su definirani u modelu, osim ako nisu anotirani kao `@Field` (npr. `@Field("name")`). `@DBRef` predstavlja referencu na drugu kolekciju (strani ključ kod relacijskih baza).

U svrhu čišćeg kôda korištene su **Lombok** anotacije. U primjeru niže `@Getter` i `@Setter` anotacije su korištene na razini klase te generiraju *get* i *set* metodu za svaku varijablu. Koristi se i zadani konstruktor (engl. *constructor*), `@NoArgsConstructor`, odnosno konstruktor bez argumenata. Za sve argumente klase bi se koristio `@AllArgsConstructor`. Niže je prikazan Lombok *dependency* koji ima `scope provided` što znači da će JDK osigurati korištenje *dependency*-ja za vrijeme *runtime*-a. Korištena je trenutno zadnja verzija, 1.18.10. [13]

```
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <version>1.18.10</version>
    <scope>provided</scope>
</dependency>
```

Kôd 4.26. Lombok *dependency*

Niže se nalazi primjer modela i dokumenta `User` iz kolekcije `Users` te Lombok anotacije. Lombok anotacije generiraju *get* i *set* metode, `toString` metodu i zadani konstruktor odnosno konstruktor bez argumenata. Odgovor u JSON obliku prikazuje samo *property*-je koji imaju vrijednost, odnosno nisu `null`. Osim navedenih Lombok anotacija, korištene su i Jackson anotacije. `@JsonInclude` omogućuje prikaz samo objekata koji imaju vrijednost, a `@JsonIgnore` označava kako se taj objekt ne treba serijalizirati u JSON. `@NotEmpty` je validacija koja označava da je vrijednost za taj objekt obavezna.

```
@Getter
@Setter
@NoArgsConstructor
@ToString
@Document(collection = "Users")
@JsonInclude(JsonInclude.Include.NON_NULL)
public class User extends AbstractId {

    @NotEmpty
    private String name;
    @NotEmpty
    private String surname;
    private String email;
    @NotEmpty
    private String password;
    private boolean isAdmin;
    private boolean isUserVerified;
    private String accessToken;

    @JsonIgnore
    public String getFullName(){
        return  name + " " + surname;
    }
}
```

Kôd 4.27. `User` model

Repozitorij je *interface* za upite nad bazom i označava se anotacijom `@Repository`. Repozitorij ima implementirane CRUD operacije, a ostale metode se trebaju definirati. Svaki **Spring Data** repozitorij je definiran modelom i ključem (npr. `<User, String>`). U sljedećem primjeru, prikazan je `UserRepository` koji nasljeđuje `MongoRepository` te definira metodu `findByEmail` koja pretražuje korisnike po zadanom emailu i vraća `User` objekt.

```
@Repository
public interface UserRepository extends
MongoRepository<User, String> {
    User findByEmail(String email);
}
```

Kôd 4.28. Primjer `User` repozitorija

Predefinirane CRUD metode u `MongoRepository`-ju su prikazane u tabličnom obliku s pojašnjenjem.

Tablica 4.3. Predefinirane CRUD metode koje pruža `MongoRepository`

Metoda	**Opis**
insert(T obj) **insert(Iterable<T> iterable)**	dodavanje jednog objekta ili više kroz kolekciju
save(T obj) **saveAll(Iterable<T> iterable)**	ažuriranje objekta ukoliko postoji `_id` za objekt ili dodavanje objekta ukoliko `_id` ne postoji
findAll()	dohvat svih objekata kolekcije
findById(String id)	dohvat objekta s određenim `_id` -jem
deleteById()	brisanje objekta s određenim `_id` -jem
deleteAll()	brisanje svih objekata u kolekciji

Ostali upiti nad bazom se definiraju u repozitoriju kolekcije kroz upite ili ključne riječi u metodi, kao što se može vidjeti u metodi `findByEmail`. Sljedeći primjeri kôda prikazuju istovjetan kôd u SQL i Mongo sintaksi.

```
SELECT * from Users WHERE email='pero@email.com'
```

Kôd 4.29. SQL upit za metodu `findByEmail("pero@email.com")`

```
{„email" : „pero@email.com"}
```

Kôd 4.30. Mongo upit za metodu `findByEmail("pero@email.com")`

Upit se može definirati i iznad metode kao `@Query` u kojoj se definira Mongo upit. Tako naprimjer `@Query("{ 'name' : {$regex : ?0, $options: 'i'}}")` pretražuje sve objekte čiji `name` sadrži zadani parametar ignorirajući velika i mala slova (definirano kroz `$options`). Sljedeća slika, preuzeta sa službene stranice, prikazuje neke ključne riječi te primjer kako se koriste na primjeru *property*-ja *age* i *birthdate*. „Sample" kolona prikazuje skraćeni zapis koji se piše u repozitoriju, a kolona logički rezultat prikazuje istovjetan Mongo upit. [6]

Tablica 4.4 Ključne riječi za izvođenje upita, primjer i rezultat Mongo upita

Ključna riječ	Primjer	Logički rezultat
After	`findByBirthdateAfter(Date date)`	{"birthdate" : {"$gt" : date}}
GreaterThan	`findByAgeGreaterThan(int age)`	{"age" : {"$gt" : age}}
GreaterThanEqual	`findByAgeGreaterThanEqual(int age)`	{"age" : {"$gte" : age}}
LessThan	`findByAgeLessThan(int age)`	{"age" : {"$lt" : age}}
Between	`findByAgeBetween(int from, int to)` `findByAgeBetween( Range<Integer> range)`	{"age" : {"$gt" : from, "$lt" : to}}
Like, StartingWith, EndingWith	`findByFirstnameLike(String name)`	{"firstname" : name} (name as regex)
Exists	`findByLocationExists(boolean exists)`	{"location" : {"$exists" : exists }}

4.4. Podatkovni sloj

U ovom radu se pristupa sigurnoj lokalnoj bazi, odnosno kreiran je `superAdmin` korisnik u *admin* bazi. Sve komande se izvršavaju kroz konzolu, odnosno **Mongo Shell**, a prvo treba kreirati korisnika u *admin* bazi. Prema službenoj Mongo dokumentaciji postupak je prikazan u sljedećem primjeru. [12]

```
use admin
db.createUser({
user: "superAdmin",
pwd: "admin",
roles: [ { role: "userAdminAnyDatabase", db: "admin" }]})
```

Kôd 4.31. Kreiranje `superAdmin` korisnika koji ima pristup svim bazama

Svi korisnici se mogu vidjeti pomoću komande `show users`. Admin korisnik se treba prijaviti kako bi kreirao korisnike za *final-project* bazu. Upisuje se sljedeća komanda: `mongo --port 27017 --authenticationDatabase "admin" -u "superAdmin" -p` te se unosi lozinka. Nakon uspješne prijave pokreće se Mongo Shell

i ponavlja se postupak kreiranja korisnika, ali za *final-project* bazu. Nakon što je korisnik kreiran, bazi se može pristupiti putem Spring Boot aplikacije kao što je navedeno u prethodnom poglavlju.
Popis kolekcija:

- Companies – poduzeća
- People – fizičke osobe
- Users – korisnici programa
- Plans – kalendar i planovi
- Templates – predlošci za kreiranje dokumenta
- Documents – detalji o kreiranim i postojećim dokumentima

Detaljna JSON shema se nalazi u dodatnoj dokumentaciji.

4.5. Osiguravanje aplikacije

Sigurnost je uvijek bila bitan dio razvoja aplikacija, a dolaskom GDPR-a je dobila na još većem značaju. Zbog toga aplikaciji mogu pristupiti samo prijavljeni korisnici, a osjetljivi podaci kao što su lozinke se kriptiraju.
U ovom projektu se koristi **Spring Security** koji dolazi u *spring-boot-starter-security dependency*-ju i sadrži *spring-aop, spring-boot-starter, spring-security-config* i *spring-security-web library*-je. Primjer *dependecy*-ja prikazan je na primjeru niže.

```
<dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>
            spring-boot-starter-security
      </artifactId>
</dependency>
```

Kôd 4.32. Spring Security *dependency*

Klijentskoj aplikaciji se pristupa prijavom putem emaila i lozinke. Ukoliko su podaci ispravni, server generira `accessToken` koji vrijedi, dok se korisnik ne odjavi ili ugasi aplikaciju. U svim dodatnim zahtjevima prema serveru, `accessToken` se prosljeđuje u zaglavlje zahtjeva te se provjerava postoji li korisnik s tim `accessTokenom`. `AcessToken` se generira pomoću BCrypt *library*-ja metodom `genSalt` koja generira tekst nasumičnih znakova.

BCrypt je algoritam sažimanja (engl. *hash*) čija sigurnost ovisi o zadanoj kompleksnosti generiranja (engl. *salt rounds*). Zadana vrijednost za *salt round* je 10 što znači da se funkcija za generiranje poziva 2^{10} puta. Povećavanjem *salt rounds* za 1, kompleksnost se duplo povećava. BCrypt generira *hash* duljine 60 znakova te se tako npr. za lozinku „123“ generira „$2a$12$4m0wBRJ4sSMCl/fc2CWAtOnsu239n68CBXDSslwd.b37myvkKFkle“. [6]

$2a označava kako se radi o BCrypt *hash*-u i predstavlja verziju koja se koristi. **$12** označava *salt rounds* te slijedi *hash*-irana lozinka i *salt*. *Salt* je nasumično kreiran podatak koji se dodaje na originalnu vrijednost. Na taj način *salt* daje dodatnu kompleksnost i svaki put prilikom promjene lozinke se ponovno generira.

Spring Security funkcionira pomoću filtera, odnosno na svim zahtjevima koji idu prema serveru se provjerava da li je korisnik prijavljen. Konfiguracijska klasa se postavlja pomoću anotacija `@Configuration` koja označava da se radi o konfiguracijskoj klasi i `@EnableWebSecurity` koja označava da se koristi Spring Security. `GlobalSecurity` klasa nasljeđuje apstraktnu klasu `WebSecurityConfigurerAdapter` te se samo treba napraviti *override* metoda `configure` kao što je prikazano u idućem primjeru. Sama konfiguracija se gradi na `HttpSecurity` objektu. Većina metoda u `http` objektu može pokrenuti `build` metodu, a one koje ne mogu, koriste se za kreiranje lanca. Primjer takve metode je `and` koja vraća `SecurityBuilder` objekt i očekuje nastavak niza s drugim `HttpSecurity` objektom. CSRF služi kako bi spriječio lažirane *cross-site* zahtjeve u JavaScript-u te nije potreban za REST aplikaciju. REST aplikacija ne koristi sesije tako da je `sessionManagement` također nepotreban. Metoda `authorizeRequests` ograničava pristup ovisno o `HttpServletRequest`-u, nakon čega se veže metoda `antMatchers` kojom se definiraju URL-ovi na koje se autorizacija odnosi. Metoda `permitAll` dopušta pristupanje svim korisnicima (prijavljenim i anonimnim). Metoda `anyRequest` se odnosi na sve URL-ove koji nisu definirani u prijašnjim `antMatchers` metodama. [6]

```
@Override
protected void configure(HttpSecurity http)
throws Exception {
      TokenFilter filter = new TokenFilter(userService);
      filter.setAuthenticationManager(filter
            .getAuthenticationManager());

      http
            .csrf().disable()
            .sessionManagement().disable()
            .authorizeRequests()
            .antMatchers("/auth/**", "/google/auth/**",
            "/azure/auth/**").permitAll()
            .anyRequest().authenticated()
            .and()
            .addFilter(filter)
            .addFilterAfter(new ExceptionTranslationFilter(
            new Http403ForbiddenEntryPoint()),
            filter.getClass());
}
```

Kôd 4.33. Metoda za provjeru prijavljenih korisnika i pristupanje API-ju

Filter je kreiran u zasebnoj klasi `TokenFilter` za kojeg se definira na koje zaglavlje se odnosi. Aplikacija treba provjeravati da li za svaki zahtjev postoji `access-token`. Zaglavlje se definira u `getPreAuthenticatedPrincipal` metodi koja prima zahtjev i vraća vrijednost `access-token`-a. U sljedećem primjeru prikazana je metoda `getAuthenticationManager` koja vraća `AuthenticationManager`. U `AuthenticationManager`-u se uspoređuje vrijednost `access-token`-a s tokenima koji su pohranjeni u bazi. Ukoliko postoji korisnik koji ima taj `access-token`, autentifikacija se postavlja na `true` vrijednost. U `GlobalSecurity` klasi filter se postavlja metodom `addFilter`. Ukoliko ne postoji token, vraća se odgovor sa statusnim kôdom *403 – Forbidden* koji je definiran u metodi `addFilterAfter` koja dodaje filter nakon prvog filtera.

```
public AuthenticationManager getAuthenticationManager(){
      return authentication -> {
            String accessTokenValue = (String)
            authentication.getPrincipal();

            if(userService.isUserLoggedIn(
            accessTokenValue)){
                  authentication.setAuthenticated(true);
            }

            returnauthentication;
      };
}
```

Kôd 4.34. Kreiranje `AuthenticationManager`-a u `TokenFilter` klasi

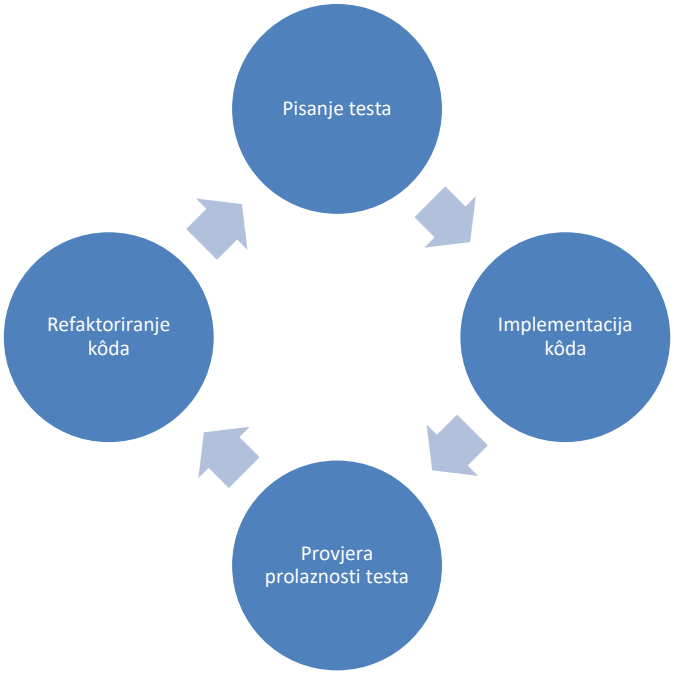
5. Testiranje

U tradicionalnom razvoju prvo se piše specifikacija funkcionalnosti te se implementiraju zahtjevi i logika nakon čega slijede faze testiranja: programersko, testersko i korisničko. Testiranje se dijeli na 3 tipa: testiranje kao bijela, siva i crna kutija. Programersko testiranje je testiranje bijele kutije gdje se testira stanje baze, što se pohranjuje u bazu te kako funkcionira sam kôd. Suprotno testiranje je testiranje crne kutije gdje krajnji korisnik testira da li se softver ponaša predviđeno. Testiranje sive kutije izvodi testni tim. Ovo testiranje je kombinacija programerskog i korisničkog testiranja jer se istovremeno provjerava da li se softver ponaša predviđeno te da li se podaci pravilno pohranjuju. [19]

TDD je tehnika i metodologija po kojoj se prvo pišu testovi, a zatim produkcijski kôd, kako bi se test izvršio i prošao. Nakon testiranja slijedi refaktoriranje, odnosno poboljšavanje kôda. Testovi omogućuju poboljšavanje kôda tako što osiguravaju da će kôd i dalje biti ispravan. Najbitnije je prvo proći kroz zahtjeve te onda pisati kôd, odnosno najbitniji je rezultat. TDD je nastao kako bi se pojednostavio razvoj i stavio fokus na stvarne zahtjeve korisnika te smanjio nepotreban kôd zbog krivih procjena o pogreškama. Pogreške čine kôd nestabilnim, nepredvidljivim ili potpuno neupotrebljivim. Loš kôd se rješava kroz pozitivno i negativno testiranje. Prvo se piše kôd koji se izvršava (pozitivno testiranje), a zatim kôd koji baca iznimku, tj. ne zadovoljava uvjete (negativno testiranje). Na taj se način sprječava pripajanje lošeg kôda s glavnim kôdom koji u najgorem slučaju može dovesti do potpuno nefunkcionalnog softvera. [18]

TDD je bitan dio agilnog razvoja čiji je cilj brzo dostavljanje novih funkcionalnosti. TDD osigurava dostavljanje funkcionalnog softvera iako možda nisu napravljene sve funkcionalnosti predviđene za tu iteraciju. [19]

Bitan dio testiranja je sposobnost pisanja testova, odnosno predviđanje programerskih i korisničkih grešaka. Budući da se prvo pišu testovi, programeri se postavljaju kao korisnik i fokusiraju se na krajnji rezultat, a ne na samu implementaciju. TDD smanjuje vrijeme popravljanja pogrešaka što smanjuje cijenu razvoja i potrebu korištenja *debugger*-a. [19] Na sljedećoj slici prikazan je razvojni ciklus testova. Prvo se piše test i logika koja je dovoljna kako bi se aplikacija kompilirala te slijedi implementacija kôda i provjera prolaznosti testova. Ukoliko test ne prođe, jednostavno je uočiti koji dio kôda je neispravan te se ispravlja logika. Ukoliko test prođe, kôd se refaktorira i postupak se ponavlja.

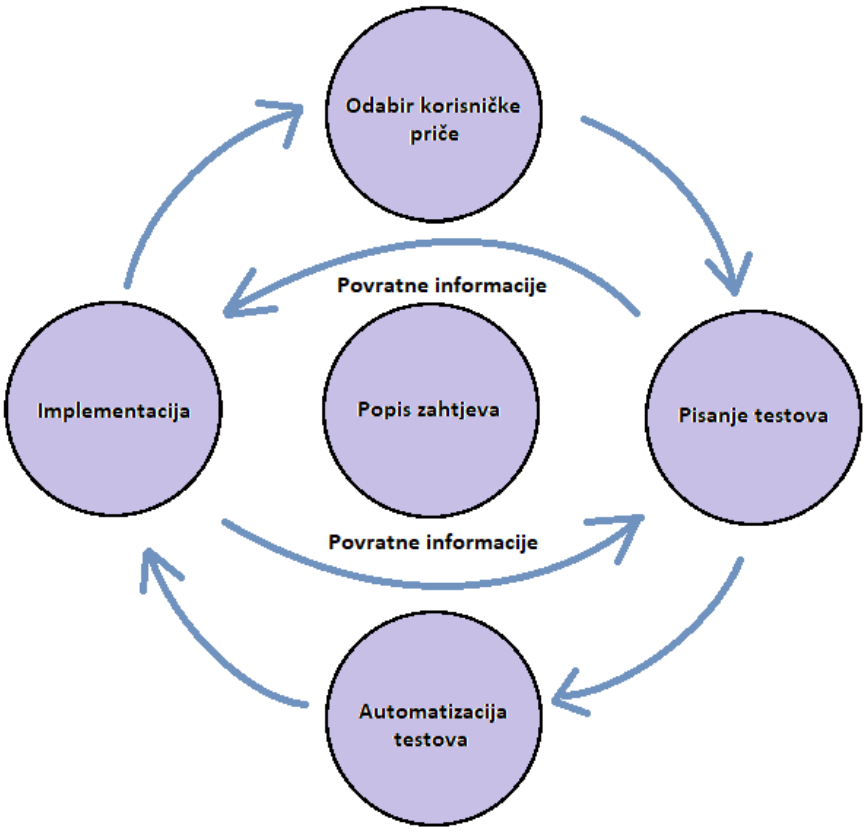


Slika 5.1. Razvojni ciklus testova

5.1. Testovi prihvaćanja

TDD omogućuje poboljšanje samog kôda i logike, a ATDD omogućuje održavanje fokusa na stvarnim zahtjevima. ATDD su pisani jezikom koji je razumljiv korisnicima i programerima, odnosno kroz korisničke priče (engl. *user story*). Korisničke priče se postavljaju kao jedna rečenica u formatu: <*uloga*> <*funkcionalnost*> <*mogućnosti*>. Korisničke priče ne objašnjavaju kakav će biti dizajn aplikacije ili kako će funkcionalnost biti implementirana. Najbitniji je prikaz krajnjeg cilja funkcionalnosti te da korisničku priču razumiju istovjetno i korisnici i programeri. Zbog toga je bitno da formu testova prihvaćanja pišu korisnici, programeri i testeri.

Na sljedećoj slici prikazan je agilni razvoj funkcionalnosti. Prvo se formiraju korisničke priče iz specifikacije funkcionalnosti i određuju se prioriteti. Nakon određivanja zahtjeva, programer odabire korisničku priču s najvećim prioritetom i započinje s pisanjem testova i implementacijom. Pisanje testova i implementacija se nastavlja sve dok se ne postigne željeni rezultat nakon kojeg programer prelazi na drugu korisničku priču.



Slika 5.2. Agilni razvoj funkcionalnosti

Testovi prihvaćanja se mogu koristiti za testiranje korisničkog sučelja i poslovne logike. Testiranje korisničkog sučelja ovisi o situaciji i zahtjevima korisnika. I dalje je bitno imati u vidu da se testovi izvršavaju brzo.

Razvojni okviri koje testovi prihvaćanja koriste, dijele se na okvire bazirane na tablicama, testove bazirane na tekstu (slično pisanom jeziku) i okvire bazirane na skriptnim jezicima (npr. Ruby). U ovom radu se koristi programski okvir baziran na tablicama, **Cucumber**, jedan od najpopularnijih alata za BDD. Kombinacija TDD-a i ATDD-a je dovela do razvoja BDD-a koji se bazira na kontinuiranoj interakciji razvojnog tima i korisnika putem automatizirane dokumentacije kroz korisničke priče i **Gherkin** sintaksu. Na sljedećoj slici je prikazan odnos TDD-a, ATDD-a i BDD-a.

Slika 5.3. TDD i ATDD su tehnike koje se koriste u BDD-u

Cucumber nudi integraciju s mnogim popularnim alatima kao što su Slack, Jira, Git i dr. Za korištenje je potrebno dodati *dependency* i instalirati *plugin* u IDE (IntelliJ ili Eclipse). Cucumber *dependecy* ima definiran `<scope> test` što označava da će biti korišten samo za vrijeme testiranja. Cucumber *dependency* je prikazan na slici niže. Koristi se *artifact* `cucumber-java8` što omogućuje korištenje lambdi i trenutno aktualna verzija 4.7.4.

```
<dependency>
    <groupId>io.cucumber</groupId>
    <artifactId>cucumber-java8</artifactId>
    <version>4.7.4</version>
    <scope>test</scope>
</dependency>
```

Kôd 5.1. Cucumber *dependecy*

Cucumber koristi Gherkin sintaksu koja koristi lokalizaciju, odnosno testovi se mogu pisati na više jezika što dodatno pojednostavljuje komunikaciju s korisnikom. U daljnjim primjerima se koristi engleski jezik. U tablici koja je preuzeta sa Cucumber službene stranice navedeni su prijevodi ključnih riječi s engleskog jezika na hrvatski jezik.

Tablica 5.1. Gherkin sintaska

Engleski	**Hrvatski**
Feature	Osobina Mogućnost Mogucnost
Background	Pozadina
Scenario	Scenarij
Scenario Outline	Skica Koncept
Examples	Primjeri Scenariji
Given	* Zadan Zadani Zadano

When	* Kada Kad
Then	* Onda
And	* I
But	* Ali

Spring Boot *dependency* za testiranje sadrži *JUnit, Spring Test, AssertJ, Hamcrest, Mockito, JSONassert i JsonPath. Dependency* je prikazan niže, a koristi se samo u `test` *scope*-u, odnosno samo za testiranje.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
</dependency>
```

Kôd 5.2. Spring *dependency*

Za korištenje Unit testova i testiranje baze ne koristi se direktna konekcija, već lažni objekti (eng. *mock*), čime se ubrzavaju testovi i ne koriste se pravi podaci.

U sljedećem primjeru je prikazan test za prijavu korisnika. Test se sastoji od 2 dijela: *.feature* datoteke u kojoj je navedena funkcionalnost u Gherkin sintaksi i klase u kojoj su navedeni koraci (engl. *steps*) izvršavanja testa. U oznaci `Feature`, naveden je naziv i opis funkcionalnosti. U oznaci `Scenario Outline` je navedena radnja koji korisnik želi izvršiti, odnosno u ovom primjeru korisnik unosi podatke za prijavu u aplikaciju. `Scenario Outline` označava kako će test pokriti više mogućnosti, tj. ispravnu i neispravnu prijavu. `Give, When` i `Then` su ključne riječi koje označavaju korake koji korisnik mora poduzeti kako bi izvršio prijavu. U testnoj klasi u `Given` koraku se postavljaju vrijednosti za email i lozinku. U `When` koraku se poziva metoda login *mock* servisa `UserMockService` s proslijeđenim emailom i lozinkom. `Then` korak označava očekivani rezultat koji se provjerava metodom `assertEquals`. Vrijednosti koje se

prosljeđuju u test su definirane kroz `Example` te je test uspješan, ukoliko rezultantna vrijednost odgovara `answer` vrijednosti.

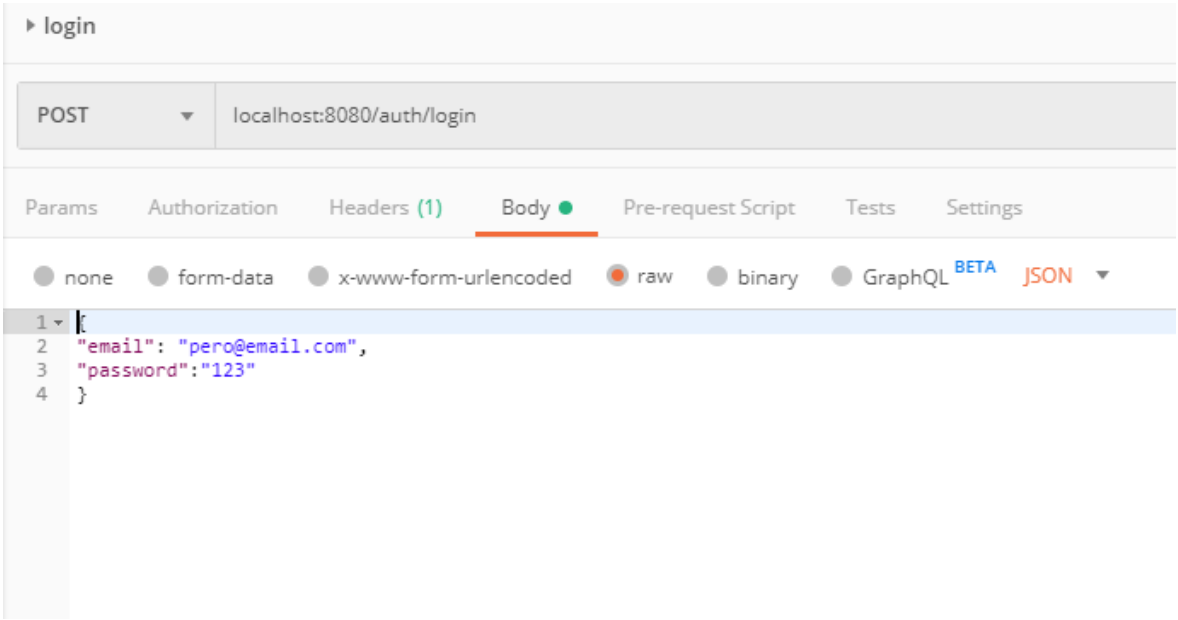
```
Feature: User authorization
  Users who can access the application

  Scenario Outline: User enters wrong or correct credentials
    Given user enters "<email>" and "<password>"
    When they login
    Then I should be told "<answer>"

    Examples:
      | email           | password | answer                   |
      | pero@email.com  | mypass   | correct                  |
      | pero@email.com  | wrong    | Password is not correct  |
      | pero@email      | mypass   | pero@email is not valid
```

Kôd 5.3. Test za autorizaciju

Osim kroz testove prihvaćanja, serverska aplikacija je testirana u **Postman** aplikaciji. Postman je aplikacija koja imitira preglednik i omogućuje slanje svih tipova zahtjeva i postavljanje parametara. Na sljedećoj slici je prikazano slanje POST zahtjeva za prijavu korisnika. Postavljen je `ContentType` `application/json` u zaglavlju te je navedeno tijelo u JSON obliku.



Slika 5.4. Postman `login` primjer

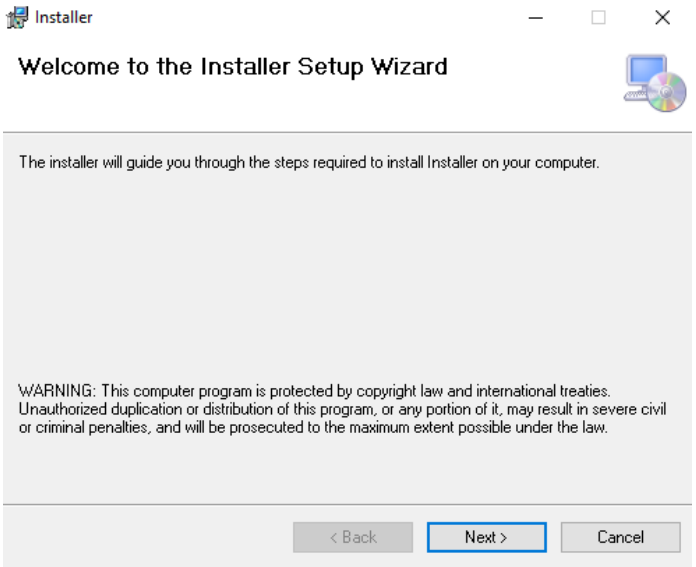
Rezultati testova su u dodatnim materijalima.

6. Implementacija

Za pripremu WPF instalacijskog paketa korišten je *Microsoft Visual Studio Installer Projects* [21] koji se dodaje u Visual Studio kao zaseban tip projekta u istom *solution*-u. **Project Setup** je prikazan kroz interaktivno sučelje koje je prikazano na slici niže. U *Application Folder* su dodani svi projekti (**CaliburnFinalProject** i **ThemeProject**) i svi korišteni resursi. U *User's Desktop* i *User's Program Menu* je dodana ikona u *.ico* formatu koja će nakon instalacije biti prikazana na radnoj površini korisnika. Za **Installer** projekt su podešena sljedeća svojstva: ikona za deinstalaciju, autor, opis i naziv aplikacije. Nakon uspješnog *build*-a cijelog *solution*-a u *Debug* direktoriju Installer projekta se nalaze *.msi* i *.exe* datoteke. Aplikacija se može instalirati samo na Windows operacijskom sustavu.

Slika 6.1. Setup Project sučelje

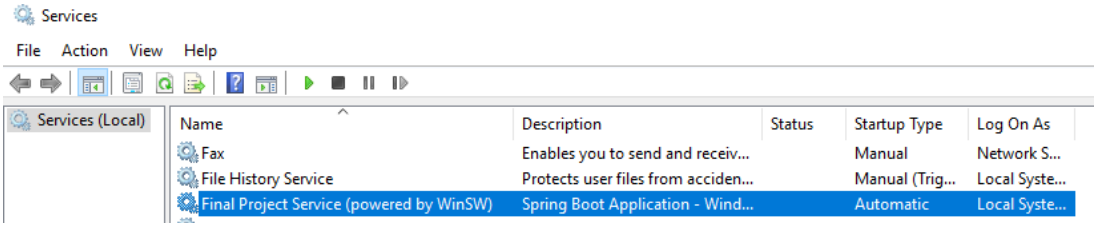
Prilikom instalacije se prikazuje sljedeći ekran, na kojem korisnik može uz nekoliko „Next" klikova instalirati aplikaciju. Korisnik može odabrati destinacijski direktorij te želi li instalirati aplikaciju za sve korisnike računala ili samo za sebe (prijavljenog korisnika).



Slika 6.2. Početni ekran za instalaciju

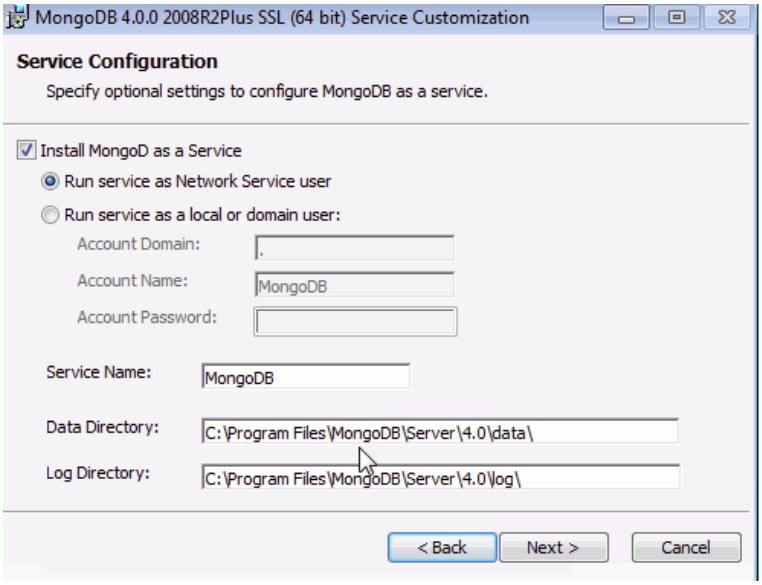
Spring aplikacija koristi ugrađeni Tomcat server te se može koristiti kao samostalna aplikacija (Windows servis) ili se može *deploy*-ati kao WAR datoteka.

U ovom projektu se koristi kao samostalna aplikacija te je potrebno pripremiti izvršni (engl. *executable*) JAR. U pom.xml datoteci je postavljena konfiguracija za Maven *plugin* `executable true` te se JAR datoteka generira pomoću naredbe `package`. Izvršni JAR se generira u *target* direktoriju i pokreće se kao bilo koja druga aplikacija, a aplikaciji se može pristupiti putem preglednika na *http://localhost:8080* URL-u. Ovakvim pristupom korisnik mora ponovno pokretati aplikaciju prilikom svakog pokretanja računala. Kako bi se to izbjeglo, potrebno ju je instalirati kao Windows servis za što se koristi **WinSW** *wrapper*. Ovim pristupom se može podesiti automatsko pokretanje servisa prilikom pokretanja računala te se servis može zaustaviti ili ponovno pokrenuti. Također je servis vidljiv u lokalnim servisima kao što je prikazano na slici niže. Detaljne upute za korištenje i instalaciju WinSW se nalaze u instalacijskom direktoriju i dodatnim materijalima. [20]



Slika 6.3. Instalirani servis

MongoDB se instalira pomoću instalacijskog paketa i potrebno je odabrati opciju servis te instalacijski direktorij. Za pokretanje servisa je dovoljno dva puta kliknuti na „mongo.exe“ datoteku. Bazi se može pristupati samo preko *localhost*-a te server i baza moraju biti na istom računalu. Na sljedećoj slici prikazan je instalacijski ekran na kojem korisnik bira tip instalacije, naziv servisa te direktorije za pohranu. [12]



Slika 6.4. Instalacija Mongo servisa

Zaključak

WPF grafički podsustav je jedan od najpopularnijih razvojnih rješenja za razvoj *desktop* aplikacija. Jednostavnim kreiranje responzivnog dizajna i povezivanjem podataka u *view*-u i *viewmodel*-u se omogućuje kreiranje kompleksnog i interaktivnog sučelja. Koristi se C# programski jezik čija je sintaksa vrlo slična Java jeziku što je pojednostavilo razvoj u različitim jezicima. Spring kao vrlo zreo razvojni okvir omogućuje brz razvoj REST servisa koji je dodatno poboljšan sa Spring Boot automatiziranom konfiguracijom. Mongo baza omogućuje prikaz i spremanje podataka u JSON obliku te kreiranje različitih shema u jednoj kolekciji čime će sigurno u budućnosti dobiti na još većoj popularnosti.

Dolaskom GDPR-a aplikacije moraju poštivati više sigurnosnih mjera i omogućiti retroaktivno otkrivanje u slučaju nedopuštenih postupaka.

Iako postoji dosta sličnih aplikacija na tržištu, često nisu dostupne svim korisnicima u ovakvom obliku, nego su uglavnom dio većih sustava kao što je CRM ili ERP.

Automatizacija procesa je bitan dio svakodnevnog poslovanja te korisnici koriste razne alate kako bi to postigli. Ovom aplikacijom korisniku se omogućava povezivanje s javnom dostupnim podacima o poduzećima registriranim u Republici Hrvatskoj te automatsko kreiranje dokumenata pomoću predložaka.

Testiranje prikazuje zanimljiv pristup razvoju u kojem je naglasak na korisničkim zahtjevima (što je oduvijek i trebao biti prioritet), a ne na samoj izvedbi. Automatizirano kontinuirano testiranje smanjuje pojavljivanje pogrešaka, odnosno brzo otkrivanje i popravljanje istih.

Kroz implementaciju je prikazan postupak u kojem se prelazi iz razvojne faze u produkcijsku, odnosno kako se isporučuje krajnjim korisnicima.

Na odabir ove teme potaknula su me iskustva iz prakse gdje je vidljiv nedostatak ovakvih aplikacija za manja poduzeća. Cilj je bio prikazati važnost interoperabilnosti u praksi, čime se u konačnici može dovesti do uštede vremena i sredstava u svim sferama gospodarstva.

Popis kratica

API	*Application Programming Interface*	sučelje za programiranje aplikacija
ATTD	*Acceptance Test Driven Development*	programiranje upravljano testovima prihvaćanja
BDD	*Behavior Driven Development*	programiranje upravljano ponašenjem
BI	*Business Intelligence*	poslovna inteligencija
BSON	*Binary JSON*	binarni JSON
CRM	*Customer Relationship Management*	upravljanje kupcima
CRUD	*Create Read Update Delete*	kreiranje, čitanje, ažuriranje, brisanje
DB	*Database*	baza podataka
DI	*Dependency injection*	umetanje ovisnosti
DMS	*Document Management System*	upravljanje dokumentima
ERP	*Enterprise Resource Planning*	planiranje resursa poduzeća
EUID	*European Unique Identifier*	europski jedinstveni identifikator
HNB	*Hrvatska narodna banka*	Hrvatska narodna banka
HTML	*Hypertext Markup Language*	
HTTP	*Hypertext Transfer Protocol*	
ID	*Identifier*	identifikator
IOC	*Inversion of Control*	inverzija kontrole
GDPR	*General Data Protection Regulation*	Opća uredba o zaštiti podataka
JAR	*Java ARchive*	Java arhiva
JSON	*JavaScript Object Notation*	notacijski objekt JavaScript jezika
JVM	*Java Virtual Machine*	Java virtualna mašina
LINQ	*Language-Integrated Query*	ugrađeni upiti u jezik
MBS	*Matični broj subjekta*	matični broj subjekta
MIME	*Multipurpose Internet Mail Extensions*	višenamjesnka ekstenzija
MVVM	*Model View ViewModel*	*model view viewmodel*
NKD	*Nacionalna klasifikacija djelatnosti*	Nacionalna klasifikacija djelatnosti
OIB	*Osobni identifikacijski broj*	Osobni identifikacijski broj
ODM	*Object Document Mapper*	objektno mapiranje dokumenata
POCO	*Plain Old C# Objects*	obični C# objekt
POJO	*Plain Old Java Object*	obični Java objekt
REST	*Representational State Transfer*	reprezentativno stanje prijenosa
SQL	*Structured Query Language*	strukturirani jezik upita
SRV	*Service Record*	oznaka servisa
TDD	*Test Driven Development*	programiranje upravljano testiranjem
UI	*User interface*	korisničko sučelje
URI	*Uniform Resource Identifier*	ujednačeni identifikator resursa
URL	*Uniform Resource Locator*	ujednačeni lokator sadržaja
WAR	*Web Application ARchive*	arhiva za web aplikacije
WPF	*Windows Presentation Foundation*	prezentacijski temelj za Windows-e
XAML	*Extensible Application Markup Language*	prezentacijski jezik za proširivanje aplikacija
XML	*Extensible Markup Language*	jezik za proširivanje

Popis slika

Popis tablica

Popis kôdova

Literatura

[1] https://docs.microsoft.com/en-us/dotnet/framework/get-started/overview, 15.12.2019.

[2] https://docs.microsoft.com/en-us/dotnet/csharp/getting-started/introduction-to-the-csharp-language-and-the-net-framework, 15.12.2019.

[3] VICE R., SIDDIQI M., MVVM Survival Guide for Enterprise Architectures in Silverlight and WPF, Packt Publishing, Birmingham (2012.), 978-1-84968-342-5

[4] https://caliburnmicro.com/, 15.12.2019.

[5] https://java.com/en/download/faq/whatis_java.xml, 15.12.2019.

[6] https://spring.io/, 15.12.2019.

[7] OTTINGER J., LOMABARDI A., Beginning Spring 5, Apress, New York City (2019.), 978-1-4842-4486-9

[8] MARCHIONI F., MongoDB for Java Developers, Packt Publishing, Birmingham (2015.), 978-1-78528-027-6.

[9] KAŠTELAN T., Uvod u baze podataka, Algebra, Zagreb (2010.), 978-953-7390-90-7

[10] LJUBI I., MIHALJEVIĆ B, Interoperabilnost informacijskih sustava, Algebra, Zagreb, (2013.), 987-953-322-158-8

[11] Uredba EU 2016/679, Opća uredba o zaštiti podataka

[12] https://www.mongodb.com/, 26.12.2019

[13] https://projectlombok.org/features/all, 26.12.2019.

[14] https://www.baeldung.com/exception-handling-for-rest-with-spring, 26.12.2019.

[15] https://www.m-files.com/en, 26.12.2019.

[16] https://start.docuware.com/, 26.12.2019.

[17] https://www.toscana-systems.eu/, 26.12.2019.

[18] KOSKELA L., Practical TDD and Acceptance TDD for Java Developers, Manning Publications, Greenwich (2008.), 978-1932394856

[19] KRAJCAR M., Projektni razvoj aplikacija, Algebra, Zagreb (2011.), 978-953-322-060-4

[20] https://docs.spring.io/spring-boot/docs/current/reference/html/deployment.html#deployment-windows, 01.02.2020.

[21] https://marketplace.visualstudio.com/items?itemName=VisualStudioClient.MicrosoftVisualStudio2017InstallerProjects, 01.02.2020.

[22] https://eur-lex.europa.eu/legal-content/HR/TXT/HTML/?uri=CELEX:32015R0884&from=FI, 25.01.2020.

Prilog

- Upute za instalaciju
- Upute za korištenje
- Shema baze u JSON obliku
- Prototip softvera (klijent i server)

ALGEBRA VISOKO UČILIŠTE

NASLOV ZAVRŠNOG RADA

Pristupnik: Ivana Kasalo, 0321006212

Mentor: prof. Aleksander Radovan

Datum: 18. 02. 2020.