ŠOLSKI CENTER VELENJE
GIMNAZIJA

Trg mladosti 3, 3320 Velenje

MLADI RAZISKOVALCI ZA RAZVOJ SAŠA REGIJE

RAZISKOVALNA NALOGA:

MOBILNA APLIKACIJA ZA MEDSEBOJNO POMOČ MED DIJAKI

Tematsko področje:
računalništvo

Avtorja:
Denis Balant, 4. letnik
Enej Hudobreznik, 4. letnik

Mentorja:
Mag. Ivan Jovan
Islam Mušić, prof.

Velenje, 2023

Raziskovalna naloga je bila opravljena na Gimnaziji ŠC Velenje.

Mentorja: mag. Ivan Jovan, Islam Mušić, prof.

Datum predstavitve: marec, 2023

KLJUČNA DOKUMENTACIJSKA INFORMACIJA

ŠD Gimnazija ŠC Velenje, šolsko leto 2022/23

KG Vrstniška pomoč / Mobilna aplikacija / Učna pomoč / Programski vmesnik / Vrstniško tutorstvo / Podatkovna baza

AV BALANT, Denis / HUDOBREZNIK, Enej

SA JOVAN, Ivan / MUŠIĆ, Islam

KZ 3320 Velenje, SLO, Trg mladosti 3

ZA Gimnazija ŠC Velenje

LI 2023

IN **MOBILNA APLIKACIJA ZA MEDSEBOJNO POMOČ MED DIJAKI**

TD Raziskovalna naloga

OP X, 87 str., 152 sl., 9 pril., 72 vir.

IJ sl

JI sl / en

AI Iskanje učne pomoči je že od nekdaj zelo aktualna tema med dijaki, ki potrebujejo pomoč pri razlagi in razumevanju učne snovi.

V raziskavi se je analizirala primernost trenutnih tovrstnih rešitev. Zaradi njihovih pomanjkljivosti se je izdelala razvila nova platforma, poimenovana OpenTutor. Ima obliko mobilne aplikacije, zgrajene z ogrodjem Flutter. Ta s podatkovno bazo PostgreSQL komunicira preko REST programskega vmesnika, zgrajenega v ogrodju ASP.NET Core, za prijavo pa uporablja šolske Arnes račune.

Glavni namen rešitve je olajšanje vzpostavitve prvega stika med iskalci dodatne razlage in uporabniki, ki so jo pripravljeni nuditi, ter spodbuditi dijake k iskanju pomoči, ko jo potrebujejo. Osnovne funkcije, ki so bile določene na podlagi primerjav različnih platform, so: prosto dostopen diskusijski forum, vmesnik za iskanje tutorjev in neposreden pogovor med uporabniki.

Po zaključenem razvoju je bila aplikacija objavljena na Google Play in Apple App Store. Povratne informacije so bile pridobljene preko intervjujev z najbolj aktivnimi tutorji ter preko ocen uporabnikov.

KEY WORDS DOCUMENTATION

ND Gimnazija ŠC Velenje, school year 2022/23

CX Peer assistance / Mobile application / Assisted learning / Application interface / Peer tutoring / Database

AU BALANT, Denis / HUDOBREZNIK, Enej

AA JOVAN, Ivan / MUŠIĆ, Islam

PP 3320 Velenje, SLO, Trg mladosti 3

PB Gimnazija ŠC Velenje

PY 2023

TI **MOBILE APPLICATION FOR PEER ASSISTANCE BETWEEN STUDENTS**

DT Research work

NO X, 87 p., 152 fig., 9 ann., 72 ref.

LA sl

AL sl / en

AB The search for learning assistance has always been a very current topic among students, who need aid in understanding the learning material.

This paper analyses the suitability of current solutions. Because of their shortcomings, a new platform called OpenTutor was developed. It takes the form of a mobile application built with the Flutter framework. It communicates with the PostgreSQL database via a REST programming interface built in the ASP.NET Core framework and uses school Arnes accounts for authentication.

The main purpose of the solution is to facilitate the establishment of the first contact between those looking for additional explanation and users who are ready to provide it, and to encourage students to seek help when they need it. The basic functions, which were determined by comparing different platforms, are: a freely accessible discussion forum, an interface for contacting tutors and a direct conversation between users.

After the development had been completed, the application was published on Google Play and the Apple App Store. Feedback was obtained through interviews with the most active tutors and through user reviews.

KAZALO VSEBINE

KAZALO SLIK

1 UVOD

Iskanje učne pomoči predstavlja večen problem učencev. Tudi nama so se kot dijakoma nižjih letnikov večkrat porajala vprašanja glede učne snovi in načina preverjanja znanja pri določenem profesorju. Zaradi pomanjkljivih stikov med dijaki različnih razredov se je bilo težko dokopati do ustreznih informacij, saj nisva vedela, na koga naj se sploh obrneva. Ko sva o tem povprašala vrstnike, sva ugotovila, da pri tem nisva sama, saj so soočali s tovrstnimi problemi.

Na naši šoli je že nekaj časa nazaj nastal sistem tutorstva, katerega cilj je ponuditi alternativo inštruktorski pomoči, ki bi bila dostopna vsem dijakom gimnazije, vendar program do sedaj še ni zares zaživel. Tako sva se odločila oblikovati aplikacijo, ki podpira zasnovan program in hkrati povezuje dijake različnih letnikov.

Aplikacija je namenjena vzpostavitvi prvega stika med tutorandom in tutorjem. Z uporabo aplikacije lahko dijaki najdejo tutorja in se z njim posvetujejo o dilemah glede učne snovi ali pa se z njim preko neposrednega pogovora dogovorijo za srečanje v živo. Prav tako je na voljo tudi forum za prosto postavljanje vprašanj, navezanih na težave pri razumevanju snovi.

1.1 Namen in cilj raziskave

Namen raziskovalne naloge je ponuditi in razviti enostavno najprimernejšo aplikacijsko rešitev za lažje iskanje učne pomoči med dijaki našega šolskega centra. Ta je posebej namenjena dijakom nižjih letnikov s ciljem olajšati prehod iz osnovne v srednjo šolo.

Cilj raziskovalne naloge je med dijaki spodbuditi povpraševanje po medsebojni pomoči in olajšati navezavo prvega stika med tutorandi in tutorji.

1.2 Hipoteze

Glede na namen najine raziskovalne naloge sva izoblikovala naslednje hipoteze:

1. Obstoječe rešitve za iskanje učne pomoči ne spodbujajo medvrstniške pomoči.

2. Relacijska podatkovna baza je za izdelavo diskusijskega foruma primernejša od nerelacijske.

3. Razvoj mobilne aplikacije z ogrodjem Flutter predstavlja učinkovito alternativo razvoju več ločenih rešitev brez vpliva na uporabniško izkušnjo.

4. Najina aplikacija je okrepila in uspešno digitalno podprla program tutorstva na Gimnaziji ŠC Velenje.

2 METODE DELA

V raziskovalni nalogi sva uporabljala različne metode, s katerimi sva raziskovala zastavljen problem ter tako spoznavala nove tehnologije in pridobila potrebne podatke ter veščine, s pomočjo katerih sva razvila primerno rešitev.

Pri raziskovalni nalogi sva uporabljala sledeče metode:

- analiza spletne dokumentacije,
- sklicevanje na uporabljeno izvorno kodo,
- testiranje med uporabniki,
- intervju s tutorjema,
- metoda analize podatkov.

2.1 Zbiranje informacij

Pri izdelavi aplikacije sva uporabljala in spoznavala mnoge nove tehnologije. Podatke, ki sva jih potrebovala za implementacijo ključnih funkcij, sva pridobila iz različnih internetnih in pisnih virov ter uradne dokumentacije razvijalcev uporabljenih tehnologij. Mnogokrat sva naletela na težave s pomanjkljivo opisanimi funkcijami. Za reševanje teh sva si pomagala z izvorno kodo različnih knjižnic in ogrodij, ki sva jo morala občasno tudi prilagoditi.

Zaradi nenehnih sprememb v svetu informacijske tehnologije sva tudi za zapis teoretičnih osnov uporabila predvsem spletne vire, ki pogosto vsebujejo veliko bolj aktualne podatke.

2.2 Analiza podatkov

Po enem mesecu delovanja aplikacije sva opravila intervju z najaktivnejšima tutorjema najine platforme. Postavila sva jim nekaj splošnih vprašanj o primernosti in delovanju aplikacije ter analizirala njune povratne informacije.

Uporabnike aplikacije sva prosila, naj nama sporočijo napake in morebitne izboljšave. Glede na njihova mnenja aplikacijo redno posodabljava in izboljšujeva ter tako skrbiva za čim boljšo uporabniško izkušnjo.

3 OBSTOJEČE REŠITVE

V spodnjem poglavju sva zbrala spletne rešitve za iskanje učne pomoči, ki sva jih poznala že sama, in druge, ki sva jih našla z iskanjem po spletu.

3.1.1 OpenProf

OpenProf je slovenski spletni izobraževalni portal, namenjen dijakom in osnovnošolcem kot pripomoček pri učenju, njegova uporaba pa je vedno pogostejša tudi kot učni pripomoček pri pouku. Portal obsega dve glavni funkciji:

- zbirka rešenih primerov z vključeno razlago in postopkom,
- ponudba inštruktorjev in tečajev. [55]

Glavna slabost platforme je omejenost brezplačnega paketa, ki poleg spletnih učbenikov zajema dostop do le manjšega števila rešenih vaj s postopki. Zahtevnejši primeri s podrobnejšimi obrazložitvami so na voljo le z naročniško storitvijo OpenProf+, ki ponuja še možnost mesečnega srečevanja z učitelji.

3.1.2 Astra.si

V okviru projekta Astra.si je ljubiteljski matematik Andrej P. Škraba ustanovil spletno učilnico, preko katere je prosto dostopnih več kot 2600 video razlag matematične snovi in okoli 100 ur različnih matematičnih izzivov, namenjenih osnovnošolcem, dijakom in deloma tudi študentom. Nudi pregled nad avtorjevim kanalom YouTube, kamor je objavil vse videoposnetke.

Gre za popolnoma brezplačno platformo, a je uporabniku kljub temu ponujena možnost donacije. [1]

3.1.3 Razturi na maturi

Slovenski portal Razturi na maturi dijakom gimnazij in drugih srednjih šol pomaga doseči boljše rezultate s pomočjo video razlag vaj, snovi in z zapiski iz vseh letnikov. Poleg razlage ponuja tudi dostop do interaktivnih nalog z rešitvami.

Učitelji, ki portal osvežujejo, stopajo v stik z uporabniki preko webinarjev in z odgovarjanjem na najpogosteje zastavljena vprašanja. Preko portala je mogoče tudi organizirati individualne ali skupinske inštrukcije prek spleta ali v živo.

Brezplačen račun omogoča dostop do webinarjev in razlage vaj ter snovi za samo eno poglavje. Za dostop do ostalih vsebin morajo dijaki plačati naročnino za eno- ali trimesečni oz. celoletni dostop. [2]

3.1.4 e-Matura

E-Matura je platforma, namenjena nudenju inštrukcij in učne pomoči dijakom zaključnih letnikov. Uporabniki lahko z enkratnim plačilom kupijo sklop video predavanj, zapiskov,

nalog in rešitev za določen predmet. Ponuja tudi brezplačen dostop do t. i. e-Knjižnice, ki vsebuje različne webinarje, članke, kvize in smernice pri vpisu na fakulteto. [3]

3.1.5 e-Inštruktor

V letu 2022 je ekipa e-Mature ustanovila novo platformo e-Inštruktor, ki je namenjena dijakom nižjih letnikov. Ta ponuja za razliko od prej omenjene pomoč le pri fiziki, matematiki in kemiji. Razlikuje se tudi v obliki plačila, saj gre v tem primeru za naročnino eno- ali trimesečnega dostopa do vsebin za prvi, drugi ali tretji letnik. Poleg tega ponuja individualno učno pomoč še pri več predmetih, kot so npr. slovenščina, angleščina, biologija itd. Inštruktor pripravi uporabniku prilagojen program, namenjen poglabljanju razumevanja učne snovi. Inštrukcije potekajo izključno preko spleta in trajajo 45 minut. [4]

3.1.6 Dijaški.net in Študentski.net

Izobraževalna portala Dijaški.net in Študentski.net predstavljata osrednjo slovensko platformo za brezplačno deljenje gradiv iz različnih predmetov med dijaki oz. študenti. Njuna glavna slabost je neverodostojnost informacij, saj gradivo, ki ga naložijo nepreverjeni uporabniki, pogosto vsebujejo napačne podatke, čeprav portal ponuja možnost ocenjevanja gradiva.

Platformi ponujata tudi zbirko vseh pomembnih informacij glede šolstva vključno s predstavitvijo fakultet, mature, tekmovanj ipd. Vgrajen je tudi diskusijski forum, ki pa v osnovi ni namenjen postavljanju vprašanj, povezanih z učno snovjo. [5]

3.1.7 Go-Inštrukcije

Platforma Go-Inštrukcije ponuja učno pomoč pri zelo obširnem naboru področij, ki vključuje tudi bolj praktične spretnosti, kot so programiranje, igranje različnih inštrumentov ipd. Inštruktorji se lahko platformi pridružijo po opravljenem intervjuju in krajšem preverjanju znanja. Sami navajajo svoje cene in določajo potek inštrukcij.

Uporabnikom je za vsakega inštruktorja na voljo tudi koledar prostih terminov, s pomočjo katerega si lažje organizirajo čas in najdejo ustrezno možnost za svoje potrebe. [6]

Poleg Go-Inštrukcij obstajajo tudi druge platforme, ki delujejo na podoben način (npr. Inštrukcije Horizont).

3.1.8 Primerjava z najino rešitvijo

Večina zgoraj opisanih rešitev se osredotoča na iskanje plačljive inštruktorske pomoči ali nudi pomoč pri samostojnem učenju v obliki gradiv, ki so pogosto plačljiva. Vse rešitve ponujajo brezplačne možnosti, ki so namenjene predvsem preizkusu in ne omogočajo dostopa do vseh potrebnih vsebin in pripadajočih razlag.

Najino aplikacijo sva namenila brezplačni medvrstniški pomoči, za katero na slovenskem trgu nisva našla ustrezne rešitve. Najprimerljivejša rešitev je portal Dijaški.net, ki se osredotoča na deljenje dijaških vsebin (predstavitev seminarskih nalog, poročil itd.) in ne

na medsebojno pomoč. Funkcija diskusijskega foruma tako ni organizirana za učinkovito postavljanje učnih vprašanj, prav tako pa nima ustrezne moderacije.

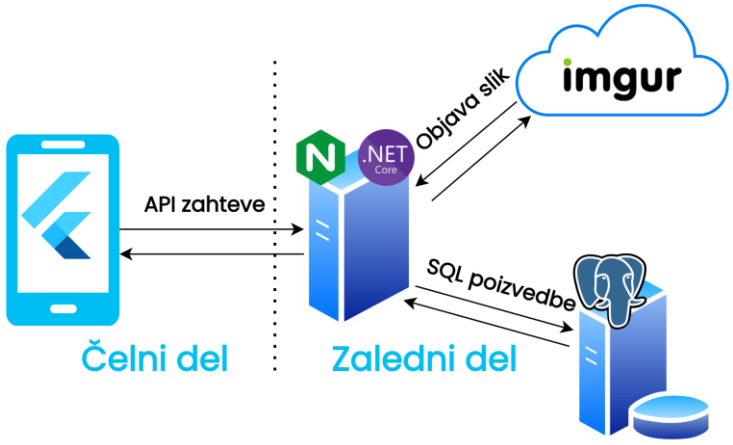
Zato sva najino aplikacijo namenila samo dijakom, ki pomoč iščejo, in tistim, ki so jo pripravljeni nuditi. Prednost pred ostalimi družbeni omrežji (npr. Facebook, Instagram in Snapchat) je, da je najina aplikacija zasnovala izključno za iskanje in nudenje dostopne učne pomoči, kar močno zmanjša ostale moteče dejavnike.

Hotela sva ponuditi brezplačno alternativo stranem za iskanje inštruktorjev, zato sva v aplikacijo vgradila funkcijo iskanja tutorjev, ki omogoča iskanje med dijaki, ki so pripravljeni poučevati določen predmet. Za hitro možnost kontaktiranja sva v aplikacijo vgradila tudi sprotni pogovor in povezavo z omrežjem Microsoft Teams ter e-poštnimi naslovi dijakov.

Pri številnih rešitvah sva pogrešala možnost, da uporabniki sami postavijo vprašanja glede nejasnosti, zato sva se odločila dodati še funkcijo diskusijskega foruma, ki poleg aktivnega vključevanja (postavljanja novih vprašanj in odgovarjanje) omogoča tudi pasivno (iskanje po že postavljenih vprašanjih in njihovih odgovorih).

4 ZASNOVA APLIKACIJE IN UPORABLJENE TEHNOLOGIJE

Mobilna aplikacija je kos programske opreme, namenjen in prilagojen uporabi na mobilnih napravah, kot so tablice in mobilni telefoni. Upošteva tako omejitve kot dodatne funkcije naprav. [7]



Slika 1: Shema arhitekture najine aplikacije (lasten vir)

Strukturo sodobne mobilne aplikacije lahko poenostavljeno razdelimo na dva dela:

- **Čelni del** (angl. frontend) predstavlja del aplikacije, s katerim uporabnik neposredno upravlja. Skrbi za prikaz podatkov.
- **Zaledni del** (angl. backend) predstavlja del aplikacije, ki uporabniku preko aplikacije ni neposredno dostopen. Ponavadi se izvaja na strežniku in ne na uporabnikovi napravi. Skrbi za obdelavo in hranjenje podatkov.

4.1 Čelni del

Čelni del aplikacije je sestavljen iz dveh med seboj tesno povezanih elementov: grafične oblike oz. izgleda in uporabniškega vmesnika, preko katerega komuniciramo z zalednim delom aplikacije. Oba sta razvita posamično, večina tehničnega dela pa je porabljena za razvoj slednjega s pomočjo različnih programskih jezikov, kot sta npr. JavaScript in Dart. [8]

4.1.1 Mobilna aplikacija

Za razvoj mobilne aplikacije so nam na voljo različne tehnologije in ogrodja, ki v osnovi omogočajo dva različna pristopa do razvoja. Med njima izbiramo glede na naš končni cilj. Če razvijamo aplikacijo za točno določen operacijski sistem iOS ali Android, si izberemo platformi lastno (angl. native) rešitev. Lastne Android aplikacije so zapisane v programskem jeziku Java ali Kotlin, iOS pa uporablja Objective-C ali Swift. Na drugi strani, pa nam medplatformski razvoj mobilne aplikacije omogoča izgradnjo za več različnih platform hkrati. [9]

Ker sva želela z najino aplikacijo doseči čim večje število dijakov, ki uporabljajo tako iOS kot Android, sva se odločila za medplatformsko rešitev, s čimer sva občutno skrajšala razvojni čas, saj ena kodna baza pokrije vsa potrebna področja. Izbirala sva med dvema najpopularnejšima ogrodjema za razvoj mobilnih aplikacij: React Native in Flutter. Raziskala sva njune prednosti in slabosti in se na podlagi na novo pridobljenega znanja tudi odločila, katero bi bilo za najine potrebe najustreznejše. [10]

4.1.1.1 Izbrana rešitev

Izbrala sva Googlovo odprtokodno ogrodje za izgradnjo mobilnih, spletnih in namiznih uporabniških vmesnikov iz ene kodne baze Flutter, ki temelji na objektno orientiranem programskem jeziku Dart.

Od že omenjenega ogrodja React Native se razlikuje po tem, da ne deluje na podlagi spletnih tehnologij in je posledično veliko hitrejši in učinkovitejši. Uporablja namreč lasten prikazovalni pogon (angl. rendering engine), ki je pri upodabljanju UI-komponent zelo učinkovit. Združuje različne tehnologije, npr. knjižnico za prikazovanje 2D-grafik Skia in virtualno okolje za programski jezik Dart, ki omogoča čiščenje spomina (angl. garbage collection), in jih izvaja v posebni lupini (angl. shell), ki je drugačna za vsako platformo, s čimer omogoča izvajanje na več platformah (imenovana embedder). [11]

Flutter se je kot dobra izbira izkazal tudi, ker je med ogrodjem, napisanim v programskem jeziku Dart, in omenjeno lupino le tanka plast C in C++ kode. Večina glavnih funkcionalnosti ogrodja je implementiranih v programskem jeziku Dart, kar razvijalcem omogoča enostavno branje izvirne kode ogrodja, ki jo lahko tudi svobodno spreminjajo glede na svoje potrebe. Razvijalcem tako ponuja velik nadzor nad celotnim sistemom, hkrati pa ga naredi dostopnejšega. [12]

4.2 Zaledni del

Zaledni del aplikacije omogoča obdelavo in spreminjanje podatkov ter komunikacijo med aplikacijo in podatkovno bazo.

4.2.1 Podatkovna baza

Za shranjevanje podatkov v računalniških aplikacijah se uporabljajo podatkovne baze – organizirane zbirke logično povezanih podatkov in njihovih opisov, ki so načrtovane tako, da zadovoljujejo informacijske potrebe uporabnikov. [13]

Za trajno shranjevanje podatkov sta danes najpogostejša dva tipa podatkovnih baz:

- **Relacijske** podatkovne baze so danes najpogosteje uporabljene. Za povezavo z njimi uporabljamo povpraševalni jezik SQL (angl. Syntax Query Language), ki omogoča tako poizvedbo podatkov kot njihovo manipulacijo. Podatki so predstavljeni kot vrstice v tabelah, ki so vnaprej določene v shemi podatkovne baze. Vsak vnos ima svoj primarni ključ (angl. primary key), ki je značilen samo zanj. S tujimi ključi (angl. foreign keys) se lahko sklicujemo na podatke, zapisane v drugi tabeli in tako med podatki vzpostavimo relacije. Najpogosteje uporabljeni so: Oracle, MySQL, SQL Server in PostgreSQL.

- **Nerelacijske** podatkovne baze so novejša alternativa relacijskim, ki za razliko od njih podatke najpogosteje (tip »document store«) shranjujejo v dinamični strukturi, sestavljeni iz kolekcij (angl. collections), ki vsebujejo več dokumentov, shranjenih v obliki polje–vrednost, ki spominja na standard JSON (JavaScript Object Notation). Za povezavo z njimi se uporablja nestrukturiran jezik NoSQL (Not Only SQL). Najpogosteje uporabljeni so: MongoDB, Azure Cosmos DB in OrientDB.

4.2.1.1 Izbrana rešitev

Sprva sva se odločila za najpogosteje uporabljeno nerelacijsko podatkovno bazo MongoDB, vendar sva pri razvoju zalednega dela naletela na težave.

Kot sva že omenila zgoraj, je ena glavnih prednosti nerelacijskih baz dinamična struktura podatkov, vendar sva ugotovila, da lahko tako hitro razvijemo neučinkovito in neskalabilno strukturo. Z relacijskimi podatkovnimi bazami te svobode nimamo, uporaba namreč temelji na določenih dobrih praksah, ki so za razliko od nerelacijskih v dokumentaciji dobro predstavljene na primerih, s pomočjo katerih lahko sistematično razvijemo urejeno shemo podatkovne baze.

Čeprav sva zasledila, da so nerelacijske podatkovne baze v primerjalnih testih pogosto hitrejše, je eden izmed glavnih razlogov za to izogibanje relacij. To nama je predstavljalo velike težave pri razvoju foruma, ki temelji na relacijah med predmeti, uporabniki, njihovimi odgovori in objavami. Ko sva težavo raziskala, sva ugotovila, da čeprav NoSQL-baze podpirajo reference, te niso pogosto uporabljene in so slabo podprte v mnogih knjižnicah.

Posebno težavo nama je predstavljalo shranjevanje objav pod različnimi predmeti. Dobra praksa pri oblikovanju sheme podatkov nerelacijske baze je gnezdenje (angl. nesting) povezanih podatkov, vendar je to pomenilo, da sva z vsako poizvedbo po predmetih iz podatkovne baze prebrala tudi vsa vprašanja, kar je upočasnilo aplikacijo.

Pripravo podatkov si pri nerelacijskih bazah pogosto olajšamo tako, da se namesto reference na drug podatek shrani kar njegova kopija. S takšnim shranjevanjem podatkov sva naletela na težavo z omejitvijo velikosti dokumenta, ki je pri MongoDB le 16 MB.

Kot začasno rešitev sva podatke razdelila na več kolekcij, vendar sva bila tako za pridobivanje relacijskih podatkov prisiljena izvesti več zaporednih klicev podatkovne baze, kar je močno zmanjšalo zmogljivost aplikacije.

Zaradi opisanih omejitev nerelacijskih podatkovnih baz pri relacijah med podatki sva se odločila zaledni del lanske aplikacije prenesti na relacijsko podatkovno bazo.

Želela sva izbrati dobro podprto odprtokodno rešitev. Ugotovila sva, da imamo na našem šolskem centru strežnik s PostgreSQL, ki je ustrezal najinim zahtevam, zato sva izbrala to podatkovno bazo.

PostgreSQL (ali Postgres) je odprtokodna podatkovna baza. Znana je po razširljivosti in ustreznosti standardu SQL. Napisana je v programskem jeziku C, prva različica pa je izšla že leta 1996. Podpira številne različne vrste tipov podatkov (angl. data types), dovoljuje

pa tudi definiranje svojih lastnih. Uporabljajo jo tudi večje organizacije in aplikacije, kot sta na primer socialni omrežji Reddit in Instagram. [14]

4.2.2 Aplikacijski programski vmesnik (API)

Aplikacijski programski vmesnik (angl. application programming interface – API) je niz definiranih pravil, ki pojasnjujejo, kako računalniki in aplikacije komunicirajo med seboj. Nahaja se med aplikacijo in spletnim strežnikom s podatkovno bazo in deluje kot vmesna plast, ki obdeluje prenos podatkov med sistemi. Ima obliko programa, ki se izvaja na zalednem strežniku naše aplikacije. [15]

Za izdelavo programskega vmesnika sva se odločala med pristopom REST in poizvedbenim jezikom GraphQL, ki sva ju primerjala v prilogi.

Po premisleku in obravnavi obeh možnosti sva ugotovila, da bi uporaba GraphQL le upočasnila najin razvojni čas brez večjih pridobitev.

Pri REST programskemu vmesniku namreč lahko sami določimo pričakovano obliko poizvedbe, ki je najučinkovitejša za naše potrebe, pri GraphQL pa mora biti strežnik sposoben izvesti poizvedbe različnih oblik, hkrati pa moramo paziti na varnost podatkov. Tako mora biti GraphQL vmesnik previdno zasnovan, zato je razvoj z REST hitrejši in lažji, sploh ko pride do kompleksnejših poizvedb.

Po najinem mnenju GraphQL tako najbolj zablesti pri večjih in kompleksnejših projektih, ki imajo več različnih odjemalskih aplikacij, ki za delovanje potrebujejo veliko podatkov iz različnih virov, kjer lahko neučinkovite ali ponovljene poizvedbe terjajo velik davek na delovanje aplikacije. Ker pa gre v najinem primeru za manjši projekt, ki ne zahteva pretirane robustnosti, sva se raje odločila za REST.

Ker sva imela predhodne izkušnje s programskim jezikom C# in Microsofotvim spletnim ogrodjem za izgradnjo spletnih aplikacij in storitev ASP.NET Core (Active Server Pages Network Enabled Technologies), sva se odločila zanj.

Ogrodje ASP (Active Server Pages) je bilo sprva namenjeno izdelavi spletnih strani, ki se izvajajo na strežniku, napisanih v skriptnem jeziku ASP. Pozneje je izšel ASP.NET, ki je omogočal izdelavo spletne strani v programskem jeziku C#. V naslednjih letih je podjetje izdalo še mnogo različnih ogrodij, namenjenih izdelavi različnih spletnih aplikacij.

Vsa različna ogrodja, ki so bila prej ločeni projekti, so leta 2016 združili v modularno ogrodje ASP.NET Core, ki omogoča preprosto izdelavo spletnih strani, HTTP programskih vmesnikov ipd. v jeziku C#. [16]

4.3 Sodelovanje pri izdelavi kode

Ker sva pri razvoju želela učinkovito in ustrezno sodelovati, sva uporabljala orodje Git, za shrambo najinih Git repozitorijev (angl. repository) pa sva uporabljala spletno platformo GitHub. Zanju sva se odločila, ker nama je potek dela z njima že precej znan, saj sva ju že večkrat uporabljala za druge osebne projekte, hkrati pa sta to tudi najpopularnejši rešitvi na področju sodelovanja v razvoju programske opreme in gostovanja kode.

4.3.1 Git

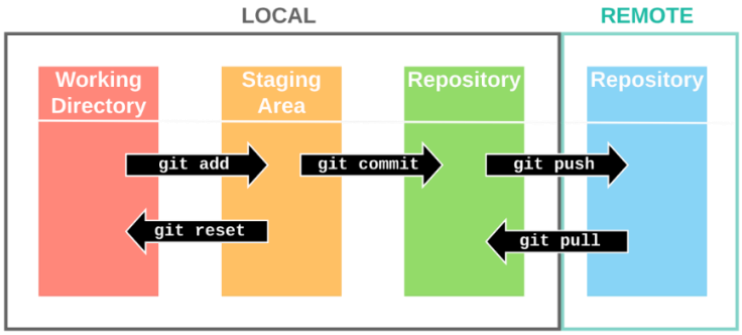
Git je brezplačen in odprtokodni sistem za nadzor različic (angl. Version Control System), s katerim lahko nadzorujemo spremembe dokumentov ali kode. Omogoča sodelovanje več razvijalcev na istem projektu, hkrati pa lahko kadarkoli obnovimo kodo na katerokoli preteklo različico. V primeru problemov v kodi lahko tako kadarkoli vidimo, kdo je katerikoli delček kode spremenil in objavil ter tako hitreje pridemo do rešitev. [17]

Zelo pomembna je tudi funkcija razvoja na različnih vejah (angl. branches), ki omogoča, da je koda projekta, namenjenega produkciji, ločena od kode, kjer razvijalci implementirajo nove funkcije. Ko je posodobljena in razširjena koda pripravljena za objavo, jo lahko enostavno združimo (angl. merge) s tisto, namenjeno objavi.

Spremembe v kodi dodajamo v treh fazah. V prvi fazi dodajamo v lokalni repozitorij vse datoteke (ukaz *git add*), v drugi fazi spremembe z ukazom *git commit* pošljemo (angl. commit) na naš lokalni repozitorij in tako naredimo posnetek trenutnih sprememb. [18]

V zadnji fazi spremembe z ukazom *git push* pošljemo repozitoriju na oddaljeni platformi. V najinem primeru sva uporabljala GitHub.

Obstajajo tudi razni grafični uporabniški vmesniki, ki nekatere Git ukaze in storitve olajšajo, a sva zaradi hitrosti skozi projekt uporabljala kar ukazno vrstico.



Slika 2: Najpogostejši ukazi pri delu z orodjem Git [19]

4.3.2 GitHub

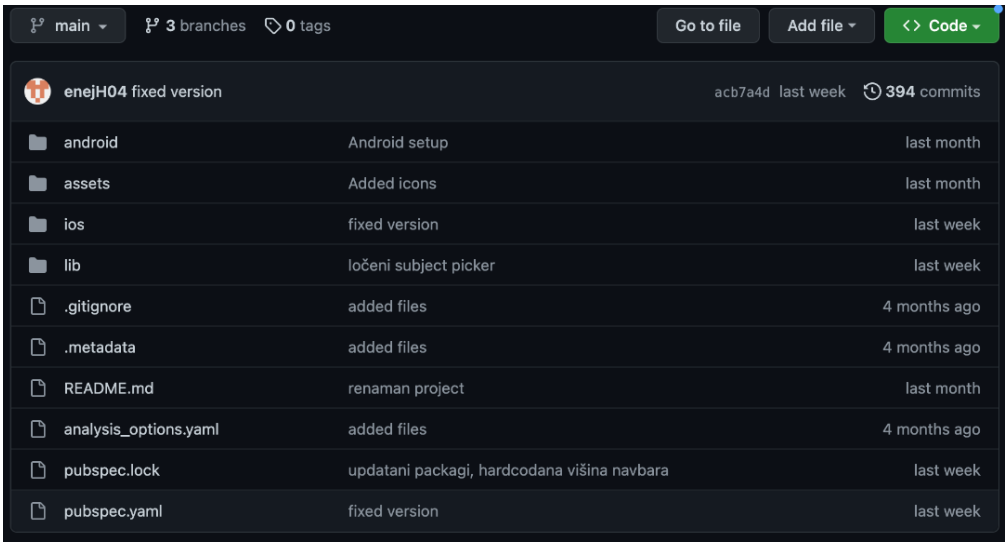
GitHub je spletna platforma v lasti podjetja Microsoft, ki omogoča gostovanje javnih ali zasebnih Git repozitorijev.

S svojim preprostim spletnim uporabniškim vmesnikom razvijalcem močno poenostavi delo z orodjem Git. Vizualizira spremembe kode, prikaže različne veje in močno olajša medsebojno sodelovanje. [20]

Razvijalci lahko na tej platformi poiščejo tudi nove odprtokodne projekte, pri katerih lahko sodelujejo in s tem hkrati tudi dopolnijo svoje znanje.

GitHub olajša tudi posodabljanje kodne baze s številnimi storitvami.

Takšen primer je GitHub Actions – storitev za zvezno integriranje in zvezno dostavo kode (CI/CD). Uporabnikom brezplačno ponuja omejeno število minut (2000 min/mesec za zasebne in neomejeno za javne repozitorije) na njihovih virtualnih napravah, na katerih lahko izvajamo željene procese. Z njo preprosto in avtomatsko objaviva novo verzijo programskega vmesnika na najin strežnik ob spremembi kode. [21]



Slika 3: Primer GitHub repozitorija (lasten vir)

5 IZDELAVA APLIKACIJE

5.1 Podatkovna baza

Po opredelitvi glavnih funkcij najine aplikacije, opisanih v začetku naloge, sva začela načrtovati shemo podatkovne baze. Odločila sva se za relacijsko podatkovno bazo, zato sva morala previdno določiti strukturo podatkov, ker jo je pozneje težje bistveno spreminjati.

Sama s poizvedbenim jezikom SQL in relacijskimi podatkovnimi bazami nimava veliko izkušenj, zato sva se odločila, da si bova delo olajšala z uporabo orodja za objektno-relacijsko mapiranje (angl. ORM – Object-relational mapping).

5.1.1 Objektno-relacijsko mapiranje

Objektno-relacijsko mapiranje (ORM) je programska metoda za pretvorbo podatkov, zapisanih v podatkovnih bazi, v objekte, lastne objektno orientiranim programskim jezikom. [22]

5.1.1.1 Entity Framework

Okolje .NET, ki sva ga izbrala za izdelavo zalednega dela aplikacije, ima odlično podporo za tovrstna orodja. Najbolj dopolnjeno je Microsoftovo lastno odprtokodno ogrodje Entity Framework Core (na kratko EF Core).

EF Core tako omogoča podatkovno usmerjen razvoj (t. i. pristop Code First). Razvijalci med seboj povezane podatke zberejo v razrede (angl. class), ogrodje pa med njimi samo ugotovi relacije in oblikuje shemo podatkovne baze, ki jo zapiše v lastnem formatu. Z vključenimi orodji lahko uporabnik shemo brez poznavanja SQL namesti na željeno podatkovno bazo. [23]

Ogrodje s podatkovnimi bazami komunicira preko vtičnih knjižnic (angl. plug-in libraries), imenovanih ponudniki podatkovnih baz (angl. database providers). Uporabnik knjižnice namesti preko upravljalca paketov (angl. package manager) NuGet za C# okolje. Uradne knjižnice so na voljo le za Microsoftove podatkovne rešitve SQL Server, Azure Cosmos DB in enodatotečni projekt SQLite, zaradi dela odprtokodne skupnosti pa ogrodje podpira praktično vse večje relacijske podatkovne baze, vključno s PostgreSQL preko knjižnice *NpgSql*, ki jo tudi uporabljava. [24, 56]

5.1.2 Strukturiranje podatkov

5.1.2.1 Definicija osnovnih entitet

Začela sva z definicijami devetih osnovnih entitet in njihovih atributov v obliki podatkovnih razredov (t. i. modelov) za najino aplikacijo:

- uporabnik (angl. User)
- tutor (angl. Tutor)
- predmet (angl. Subject)
- vprašanje (angl. Question)
- odgovor (angl. Reply)
- zapis glasovanja (angl. Vote Record)
- pogovor (angl. Chat)
- sporočilo (angl. Message)
- slika (angl. Image)

Za vsak razred sva definirala polja (angl. field). Vsak zapis v podatkovni bazi ima, kot že omenjeno, primarni ključ ali Id (edinstvena zaporedna števila), po katerem se lahko drugi sklicujejo nanj. Definicije atributov so enake kot za navaden razred v jeziku C#, zato bova postopek predstavila le na primeru vprašanja.

```
5 references
public class Question
{
    [Key]
    8 references
    public int Id { get; set; }
    7 references
    public string Title { get; set; }
    6 references
    public string Content { get; set; }
    6 references
    public List<Image> Images { get; set; }

    8 references
    public User UserCreated { get; set; }
    7 references
    public string UserCreatedId { get; set; }

    6 references
    public List<Reply> Replies { get; set; }
    5 references
    public int SubjectId { get; set; }
    2 references
    public Subject Subject { get; set; }
}
```

Slika 4: Definicija razreda za vprašanje (lasten vir)

Poleg primarnega ključa, označenega z atributom Key, sta definirani še polji naslov (Title) in vsebina (Content). Ta tri polja predstavljajo osnovne podatkovne tipe, ki jih podatkovne baze podpirajo neposredno, zato jih ogrodje prevede v stolpce ustvarjene tabele.

Sledita seznam slik, ki jih lahko vsebuje vprašanje, in seznam odgovorov na to vprašanje. Tukaj gre za referenco na več elementov (angl. one-to-many).

Dodala sva še referenco na avtorja in predmet, pod katerega spada vprašanje. Tukaj gre za referenco na en element, na katerega se sklicujemo večkrat (angl. many-to-one).

5.1.2.2 Povezava s podatkovno bazo

Dostop do podatkovne baze pri EF Core poteka preko razreda, ki ima kot polja definirane zbirke (*DbSet*) osnovnih razredov podatkov in deduje razred *DbContext*.

```
32 references
public class DataContext : DbContext
{
    0 references
    public DataContext(DbContextOptions<DataContext> options) : base(options) { }

    // protected override void OnModelCreating(ModelBuilder modelBuilder) {}

    9 references
    public DbSet<User> Users { get; set; }
    3 references
    public DbSet<Tutor> Tutors { get; set; }
    5 references
    public DbSet<Subject> Subjects { get; set; }
    8 references
    public DbSet<Reply> Replies { get; set; }
    4 references
    public DbSet<VoteRecord> VoteRecords { get; set; }
    8 references
    public DbSet<Question> Questions { get; set; }
    7 references
    public DbSet<Chat> Chats { get; set; }
    5 references
    public DbSet<Message> Messages { get; set; }
    3 references
    public DbSet<Image> Images { get; set; }
}
```

Slika 5: Definicija razreda za povezavo s podatkovno bazo (lasten vir)

Povezavo s podatkovno bazo ogrodje vzpostavi preko t. i. povezovalnega niza (angl. connection string), ki ima za bazo PostgreSQL sledečo obliko (argumenti so: naslov strežnika s podatkovno bazo, ime podatkovne baze, uporabniško ime in geslo):

Host=XX.XX.XX.XX; Database=XX; Username=XX; Password=XX

```
builder.Services.AddDbContext<DataContext>(db => db.UseNpgsql(connectionString));
```

Slika 6: Klic za dodajanje razreda za podatkovno bazo (lasten vir)

Povezovalni niz in našo implementacijo razreda *DbContext* navedemo v klicu metode *AddDbContext* razreda *WebApplicationBuilder* v izhodiščni datoteki aplikacije (Program.cs), s katerim projektu dodamo ogrodje EF Core.

5.1.3 Shema podatkovne baze in njena generacija

5.1.3.1 Generiranje sheme in migracije

Kot že omenjeno, EF Core omogoča avtomatsko generacijo sheme podatkov iz naše kode in uvedbo te sheme na izbrano podatkovno bazo preko vključenih orodij ukazne vrstice.

To funkcionalnost zagotavlja funkcija t. i. migracij (»selitev« podatkovne baze), s katero lahko postopno posodabljamo shemo podatkovne baze, da ostane sinhronizirana s podatkovnim modelom aplikacije, hkrati pa omogoča ohranitev obstoječih podatkov. Nove migracije so namreč ustvarjene s primerjavo trenutnega modela podatkov s trenutno shemo podatkovne baze (s stanjem zadnje migracije). Zgodovino nameščenih migracij ogrodje spremlja v posebni tabeli, imenovani *__EFMigrationsHistory*.

```
1 reference
public partial class AddedClassYear : Migration
{
    0 references
    protected override void Up(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.AddColumn<int>(
            name: "Class",
            table: "Users",
            type: "integer",
            nullable: false,
            defaultValue: 0);
    }

    0 references
    protected override void Down(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.DropColumn(
            name: "Class",
            table: "Users");
    }
}
```

Slika 7: Primer migracije (lasten vir)

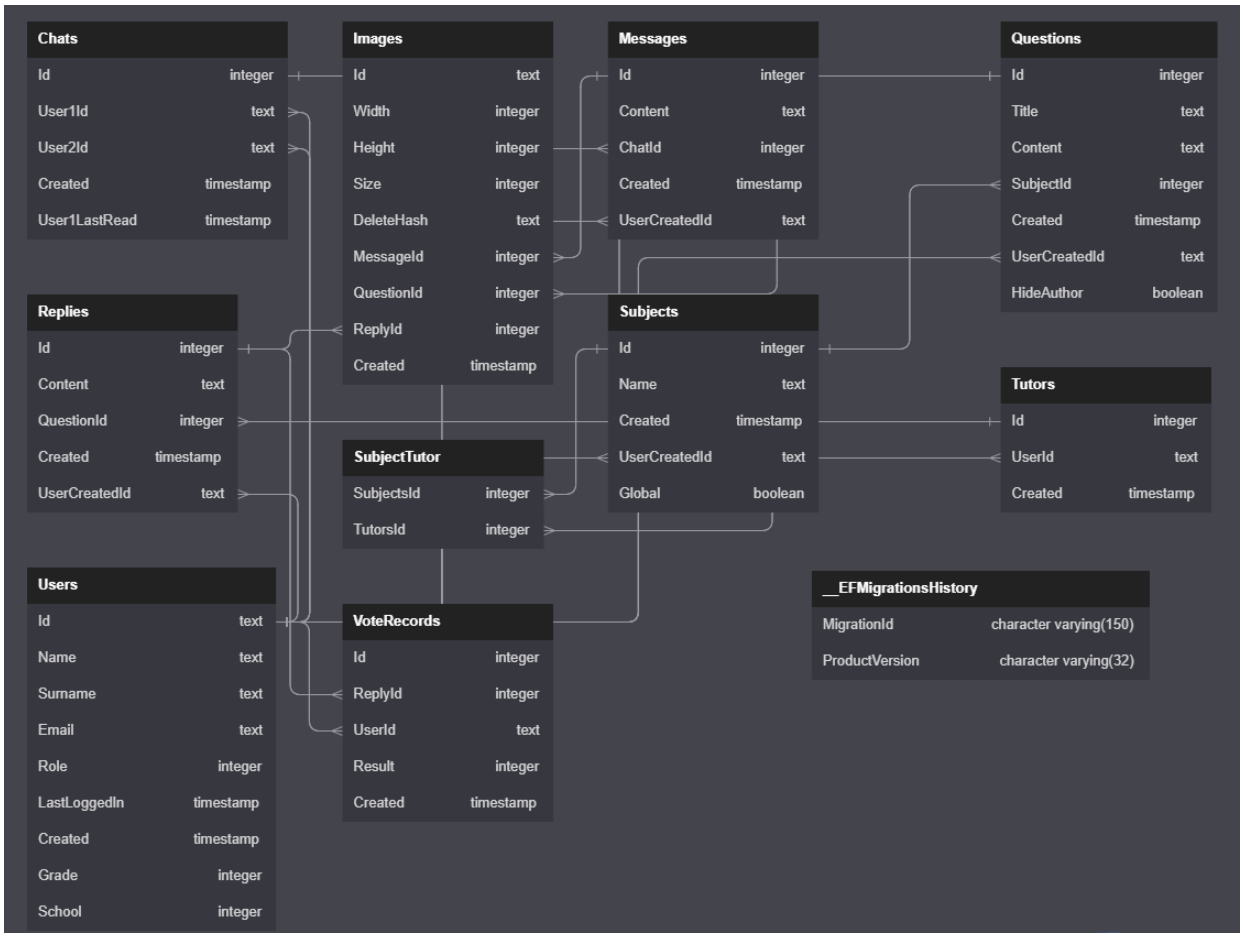
Migracijo ustvarimo z ukazom *Add-Migration <ImeMigracije>*. Generirane migracije se shranijo v mapo *Migrations* v obliki C# razreda, ki deduje razred *Migration*. Ima dve metodi: *Up*, s katero migracijo dodamo podatkovni bazi, in *Down*, s katero lahko migracijo odstranimo (ukaz *Remove-Migration*).

Čeprav je generirana koda v večini primerov ustrezna, jo je še vseeno potrebno preveriti ročno, saj se nama je nekajkrat zgodilo, da je neustrezno spremenila podatkovno bazo, zaradi česar sva shemo morala popraviti ročno. Ko smo z migracijo zadovoljni, jo namestimo z ukazom *Update-Database*.

5.1.3.2 Generirana shema

Na spodnji sliki je prikazan diagram generirane končne sheme podatkovne baze (model entitet in povezav med njimi, ERM – angl. entity-relationship model), ki sva ga ustvarila s pomočjo spletnega portala DbDiagrams.

Vsak pravokotnik predstavlja svojo tabelo (tip entitete), vrstice pa predstavljajo njene stolpce (atribute).



Slika 8: Shema podatkovne baze (lasten vir)

5.1.4 Poizvedbe podatkovne baze

Ko sva končala z oblikovanjem sheme podatkovne baze, sva se lotila pisanja poizvedb za potrebe najine aplikacije.

Kot že omenjeno, je glavna prednost uporabe orodja za objektno-relacijsko mapiranje, kot je EF Core, izogibanje pisanja SQL-poizvedb in oblikovanja podatkovnih razredov s seznama vrnjenih vrednosti.

Omenjeno ogrodje za pisanje poizvedb uporablja sintakso LINQ (angl. Language Integrated Query), ki predstavlja preprost in enoten način za pridobivanje in obdelovanje podatkov iz različnih virov (npr. nizi, razredi, datoteke XML, relacijske podatkovne baze).

Sintaksa je osnovana na verižnih klicih poizvedbenih metod, imenovanih standardni operatorji poizvedb (SOO – angl. Standard Query Operators). Te metode kličemo na zbirkah podatkov v razredu za povezavo s podatkovno bazo.

Takšno poizvedbo ogrodje (ponudnik podatkovne baze) prevede v SQL-poizvedbo, rezultate pa prevede nazaj v zahtevan format, ki je lahko seznam objektov, le en objekt ali pa samo en atribut entitete.

5.1.4.1 Primer poizvedbe podatkov

```
var subjects = await _dataContext.Subjects
    .Select(s => new SubjectDto
    {
        Name = s.Name,
        QuestionsCount = s.Questions.Count,
        Id = s.Id,
    })
    .ToListAsync();
```

Slika 9: Primer poizvedbe podatkov v jeziku C# (lasten vir)

```
SELECT s."Name", (
    SELECT count(*)::int
    FROM "Questions" AS q
    WHERE s."Id" = q."SubjectId") AS "QuestionsCount", s."Id"
FROM "Subjects" AS s
```

Slika 10: Primer generirane SQL-poizvedbe (lasten vir)

Zgornji sliki prikazujeta primer poizvedbe seznama vseh predmetov. Prva slika prikazuje najino kodo za poizvedbo, druga pa avtomatsko generirano SQL-poizvedbo.

_dataContext.Subjects predstavlja tabelo predmetov (gl. 5.1.2.2). Na njej kličeva operator *Select*, ki nam podobno kot stavek SELECT v jeziku SQL omogoča izbiro le tistih podatkov, ki jih potrebujemo. V tem primeru podatke shraniva v razred *SubjectDto*, kar bova bolje predstavila v naslednjem podpoglavju. Pomembno je omeniti, da razčlenjevalnik zahteve pri spremenljivki *QuestionsCount* sam prepozna, da gre za relacijo na drugo tabelo in jo tudi ustrezno preoblikuje v COUNT-funkcijo podatkovne baze.

5.1.4.2 Primer dodajanja podatkov

```
var newSubject = new Subject
{
    Name = subjectRes.Name,
    UserCreatedId = userCreatedId,
};

await _dataContext.Subjects.AddAsync(newSubject);
await _dataContext.SaveChangesAsync();
```

Slika 11: Primer zahteve za dodajanje podatkov v jeziku C# (lasten vir)

```
INSERT INTO "Subjects" ("Name", "UserCreatedId")
VALUES (@p0, @p1)
RETURNING "Id";
```

Slika 12: Primer generirane SQL-zahteve za dodajanje podatkov (lasten vir)

Zgornji sliki prikazujeta primer zahteve za dodajanje predmeta.

Najprej v spremenljivki *newSubject* definirava nov primerek predmeta, ki ga v naslednji vrstici preprosto dodava v zbirko predmetov z metodo *AddAsync*. Ogrodje spremlja spremembe podatkov, povezanih s podatkovno bazo. Tako lahko vse neshranjene spremembe preprosto shranimo s klicem metode *SaveChangesAsync*. V tem primeru se zahteva pretvori v INSERT stavek v jeziku SQL, ki v izbrano tabelo vstavi nov vnos (vrsto).

5.1.4.3 Primer spreminjanja podatkov

```
if (replyVoteRecord is not null) {
    replyVoteRecord.Result = isUpvote ? VoteResult.UPVOTE : VoteResult.DOWNVOTE;
}

await _dataContext.SaveChangesAsync();
```

Slika 13: Primer zahteve za spreminjanje podatkov v jeziku C# (lasten vir)

```
UPDATE "VoteRecords" SET "Result" = @p0
WHERE "Id" = @p1;
```

Slika 14: Primer generirane SQL-zahteve za spreminjanje podatkov (lasten vir)

Zgornji sliki prikazujeta primer zahteve za spreminjanje stanja glasovanja pri odgovoru, za katerega smo že glasovali.

V izseku kode najprej preveriva, ali tak zapis že obstaja (ni ničeln – nima vrednosti *null*). Vrednost glasovanja preprosto spremenimo v *UPVOTE* (všeček) ali *DOWNVOTE* (»nevšeček«) glede na poslano vrednost. Ogrodje spremlja vrednosti objektov, pridobljenih iz podatkovne baze, zato ob klicu metode *SaveChangesAsync* samodejno generira ustrezno zahtevo s stavkom UPDATE v jeziku SQL, ki spremeni vrednost ustreznega polja v podatkovni bazi.

5.2 Aplikacijski programski vmesnik (API)

Po oblikovanju sheme podatkov sva začela z razvojem programskega vmesnika (API), preko katerega bo mobilna aplikacija komunicirala s podatkovno bazo.

5.2.1 Razvojno okolje

Za delo v .NET okolju sva uporabljala brezplačno izdajo Microsoftovega integriranega razvojnega okolja (IDE – angl. Integrated Development Environment) Visual Studio 2022, ki je razvoj olajšal s številnimi funkcijami in olajšavami, kot so samodejna dopolnitev kode s predlogi (funkcija IntelliSense), bogat razhroščevalnik (angl. debugger), grafični vmesnik za upravljalec paketov NuGet in vgrajena podpora za sistem za upravljanje izvorne kode Git.

5.2.2 Arhitekturni vzorec MVC

Struktura projekta v ogrodju ASP.NET Core je osnovana na arhitekturnem vzorcu Model-View-Controller (MVC), katerega cilj je napisati kodo, ki jo je lažje vzdrževati in testirati. [25]

Vzorec sestavljajo trije osnovni deli:

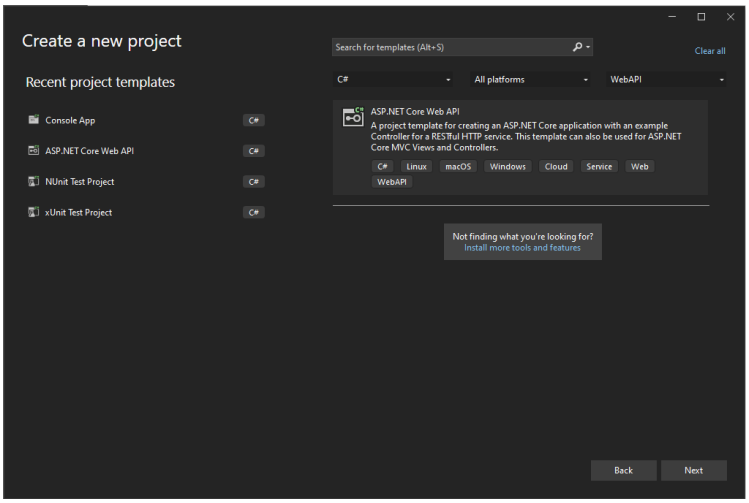
- **Modeli** (angl. models) so razredi, ki predstavljajo obliko podatkov v naši aplikaciji. Pogosto zraven štejemo tudi kodo, ki preveri njihovo pravilno obliko in jih shrani v podatkovno bazo.

- **Pogledi** (angl. views) so komponente aplikacije, ki skrbijo za prikaz uporabniškega vmesnika (UI). Ponavadi prikazujejo podatke modelov.

- **Krmilniki** (angl. controllers) so razredi, ki obdelujejo zahtevke brskalnikov, kličejo podatke modelov in določajo, kateri pogledi se bodo prikazali v aplikaciji.

Pri API nimamo uporabniškega vmesnika, zato tukaj ne govorimo o pogledih, preostala dva dela vzorca pa sta bistvena za njegovo delovanje.

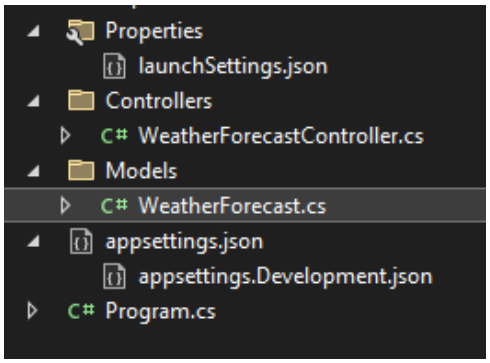
5.2.3 Generiranje začetnega projekta



Slika 15: Vmesnik okolja Visual Studio 2022 za generiranje novega projekta (lasten vir)

Ogrodje ASP.NET Core vsebuje več predlog (angl. templates) za različne tipe spletnih aplikacij. Te predloge lahko pregledno iščemo in nameščamo preko okolja Visual Studio. Ker sva izdelovala spletni API, sva izbrala temu ustrezno uradno predlogo.

5.2.3.1 Datotečna struktura



Slika 16: Datotečna struktura predloge za spletni vmesnik (lasten vir)

Na zgornji sliki je prikazana datotečna struktura predloge. Vlogo modelov ali osnovnih razredov sva že predstavila v prejšnjem podpoglavju, ostalo pa bova predstavila spodaj.

5.2.4 Vstopna točka (Program.cs)

```
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.

builder.Services.AddControllers();

var app = builder.Build();

// Configure the HTTP request pipeline.

app.UseAuthorization();

app.MapControllers();

app.Run();
```

Slika 17: Vstopna datoteka projekta (lasten vir)

Na zgornji sliki je prikazana začetna vsebina osnovna datoteke projekta *Program.cs*, ki predstavlja vstopno točko v program.

Spletno aplikacijo v okolju ASP.NET Core ustvarimo z dvema klicema. Najprej moramo poklicati metodo *CreateBuilder* razreda *WebApplication*. Tako dobimo primerek razreda *WebApplicationBuilder*, preko katerega lahko dodamo odvisnosti projekta, kar bova podrobneje opisala v naslednjem podpoglavju. [26]

Sledi klic metode *Build*, preko katere dobimo primerek razreda *WebApplication*, ki predstavlja temelj spletne aplikacije. Dodamo mu t. i. vmesno programje (angl. middleware), s katerim določimo, katere komponente in v katerem vrstnem redu bodo obdelale http-zahtevek, preden bo ta prišel do naše kode. To storimo s klicem metod *Use~*. Tako določimo t. i. cevovod obdelave zahtevkov (angl. HTTP request pipeline) naše aplikacije. [27]

S klicem metod *Map~* povežemo našo kodo z URI-potjo zahtevka in tako dokončamo definicijo cevovoda obdelave zahtevkov. S klicem *MapControllers* tako povežemo zahtevek s krmilniki naše aplikacije.

Spletno aplikacijo nato poženemo na trenutni niti s klicem metode *Run* na nastavljenem primerku razreda *WebApplication*.

5.2.5 Upravljanje odvisnosti projekta

Pri izdelavi programskega vmesnika sva se trudila izdelati čim več komponent za večkratno uporabo. Tako sva vse povezane funkcionalnosti združila v ločene razrede. Takšne ločene programske komponente strokovno imenujemo storitve.

Za klic funkcionalnosti storitev vsak razred potrebuje svojo instanco razreda storitve – pravimo, da storitev predstavlja t. i. odvisnost (angl. dependency) razreda – gre za odvisnost enega razreda od drugega.

5.2.5.1 Upravljanje odvisnosti (DI)

Uporabljeno ogrodje upravljanje odvisnosti močno poenostavi z vgrajenim zabojnikom storitev. Storitve aplikacije preprosto dodamo k zbirki storitev tipa *IServiceCollection* (*builder.Services)* s klici metod *Add~*, ki jih imajo definirane vnaprej narejene storitve. Tako ogrodje skrbi tudi za upravljanje s krmilniki (metoda *AddControllers*) in povezavo s podatkovno bazo (že omenjena metoda *AddDbContext*).

Pri dodajanju lastnih storitev v obliki razredov in njihovih vmesnikov moramo zraven določiti še njihovo življenjsko dobo (angl. lifetime) – določiti, kako pogosto se mora ustvariti nov primerek storitve. Klicu metode dodamo še definicijo tipa v obliki *<ImeVmesnikaStoritve, ImeRazredaStoritve>();*. [28]

Poznamo tri različne življenjske dobe storitev:

- Začasno (angl. transient) – storitev se generira na novo po vsakem klicu (namenjena je lahkim storitvam brez pomnjenja) – metoda *AddTransient*
- Okvirno (angl. scoped) – storitev se generira ob vsaki zahtevi odjemalca (HTTP klicu) – metoda *AddScoped*
- Edinec (angl. singleton) – storitev se generira samo ob prvem klicu – metoda *AddSingleton*

```
builder.Services.AddSignalR();
builder.Services.AddDbContext<DataContext>(db => db.UseNpgsql(connectionString));
builder.Services.AddSingleton<IPhotoApiService, PhotoApiService>();
builder.Services.AddSingleton<IGraphService, GraphService>();
builder.Services.AddSingleton<INotificationAPIService, NotificationAPIService>();
builder.Services.AddSingleton<IImageService, ImageService>();

builder.Services.AddTransient<IUserService, UserService>();
```

Slika 18: Klici za dodajanje storitev v najino aplikacijo (lasten vir)

V najino aplikacijo sva dodala več različnih odvisnosti. Kot sva že opisala v prejšnjem poglavju, sva najprej namestila ogrodje EF Core. Njegov razred za povezavo s podatkovno bazo *DataContext* predstavlja primer storitve z okvirno življenjsko dobo. Posledično vse storitve, ki so odvisne od njega, ne morejo biti edinci, zato sva tudi *UserService* definirala kot storitev z okvirno življenjsko dobo.

Ostale storitve kličejo zunanje vmesnike preko razreda *HttpClient*, torej shranjujejo stanje, zato sva se jih odločila dodati kot edince.

5.2.5.2 Primer storitve in uporaba vmesnikov

```
public class UserService : IUserService
{
    private readonly DataContext _dataContext;
    0 references
    public UserService(DataContext dataContext) => _dataContext = dataContext;

    4 references
    public async Task<User?> GetUserProfile(ClaimsPrincipal user)
        => await _dataContext.Users.FindAsync(UserUtils.GetUserIdFromClaims(user));

    11 references
    public async Task<bool> UserIsAdmin(ClaimsPrincipal user) =>
        await _dataContext.Users.AnyAsync(u => u.Id == UserUtils.GetUserIdFromClaims(user) && u.Role == UserRole.ADMIN);

    3 references
    public async Task<bool> UserIsAdminOrAuthor(string authorId, ClaimsPrincipal User)
    {
        var userId = UserUtils.GetUserIdFromClaims(User);

        if (userId.Equals(authorId))
            return true;

        return await UserIsAdmin(User);
    }
}
```

Slika 19: Primer storitve (lasten vir)

```
10 references
public interface IUserService
{
    4 references
    public Task<User?> GetUserProfile(ClaimsPrincipal user);
    11 references
    public Task<bool> UserIsAdmin(ClaimsPrincipal user);
    3 references
    public Task<bool> UserIsAdminOrAuthor(string authorId, ClaimsPrincipal User);
}
```

Slika 20: Primer vmesnika za storitev (lasten vir)

Na zgornjih dveh slikah je prikazana implementacija storitve za pridobivanje pogosto rabljenih podatkov o uporabnikih iz podatkovne baze, ki sva jo po konvenciji poimenovala s pripono *-Service*, torej *UserService*.

Prva slika prikazuje implementacijo razreda storitve. Metoda, ki ima enako ime kot razred, predstavlja metodo za njegovo kreacijo oz. konstruktor. Kot že omenjeno, pri vzorcu upravljanja odvisnosti te metode sami ne kličemo, ampak kot argumente samo preprosto navedemo odvisnosti razreda in ogrodje samo poskrbi za njihovo upravljanje.

V tem primeru je kot odvisnost naveden razred za povezavo s podatkovno bazo *DataContext*, primerek katerega v telesu konstruktorja shraniva v zasebno spremenljivko (*_dataContext*), ki jo lahko uporabiva v metodah razreda.

V razredu sva definirala metode, ki sva jih klicala večkrat na različnih mestih v kodi. V metodi *GetUserProfile* dobiva profil uporabnika iz podatkovne baze glede na njegov identifikator (metoda *FindAsync* vrne element glede na njegov primarni ključ). Metoda *UserIsAdmin* preveri, če v podatkovni bazi obstaja zapis uporabnika, ki ima določen identifikator in je administrator (metoda *AnyAsync* vrne, ali element z izpolnjenimi pogoji obstaja). Metoda *UserIsAdminOrAuthor* vrne, če je uporabnikov identifikator enak identifikatorju avtorja ali pa je ta administrator, kar ugotovi s klicem že omenjene metode.

Definicije metod brez njihovih implementacij zapišemo v t. i. vmesnike (angl. interfaces), ki predstavljajo nekakšen načrt razreda. Te ponavadi poimenujemo z dodatkom predpone *I-* imenu razreda (npr. *IUserService*). Z razredom, ki vsebuje vse metode v vmesniku, jih povežemo s sintakso, enako dedovanju (*:ImeVmesnika* po imenu razreda) – pravimo, da razred »implementira« ta vmesnik.

Vmesniki so za ločevanje kode (angl. decoupling) zelo pomembni, saj predstavljajo povezavo, prek katere storitve kličejo razredi, ki so od te storitve odvisni. Ravno zato je zelo pomembno ustvariti čim bolj splošne metode, saj lahko dejansko implementacijo razreda ob spreminjanju sistema ali testiranju spremenimo, vmesnik pa ostane enak.

5.2.6 Definicija končnih točk

Nato sva začela z načrtovanjem, katere podatke bova potrebovala kje v aplikaciji. Tako sva oblikovala shemo različnih končnih točk vmesnika (angl. endpoints), ki morajo biti čim bolj predvidljive.

Konvencija standarda REST narekuje uporabo samostalnikov in identifikatorjev v segmentih URI naslova (npr. *forum/subjects/{id_predmeta}*), ločenih s poševnico.

5.2.6.1 Objekti, namenjeni prenosu (DTO)

Oblika podatkov, razdeljenih na tabele podatkovne baze, ni najprimernejša za prikaz in obdelavo v mobilni aplikaciji.

Dobro izdelan spletni vmesnik mora biti tako sposoben sprejeti le pričakovane podatke ter jih preoblikovati v ustrezno obliko, ki vsebuje vse potrebne reference in ne vključuje nepotrebnih podatkov. Tudi oblika vrnjenih podatkov mora slediti dejanski rabi v aplikaciji (z enim klicem dobimo celotno strukturo željenih zapisov), s čimer zmanjšamo število klicev programskega vmesnika in posledično tudi podatkovne baze.

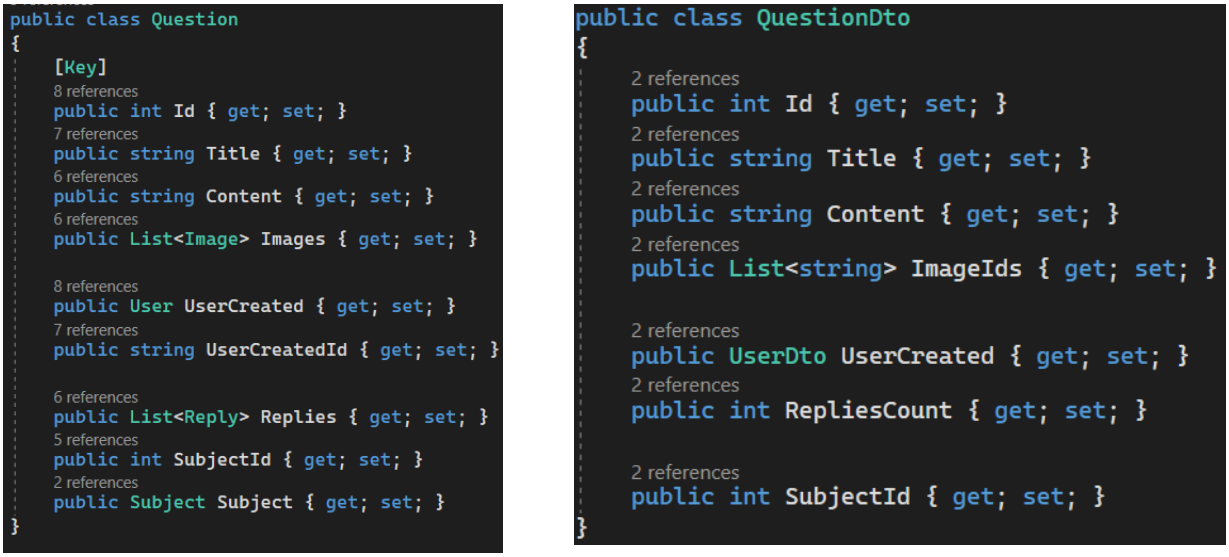
Najpogostejša rešitev opisane težave je uvedba t. i. objektov, namenjenih prenosu (angl. DTO – data transfer object). Gre za razrede, ki definirajo obliko prejetih in poslanih podatkov in tako predstavljajo dodatno stopnjo abstrakcije podatkovne baze in njenih modelov. [29]

Spodnji dve sliki prikazujeta definicijo modela za vprašanje in njegovega DTO-razreda, namenjenega prikazovanju v aplikaciji.

Določena polja so enaka (identifikator – *Id,* naslov – *Title* in vsebina – *Content*). Za prikaz slik v aplikaciji potrebujemo le njihove identifikatorje za dostop (*ImageIds*), samo

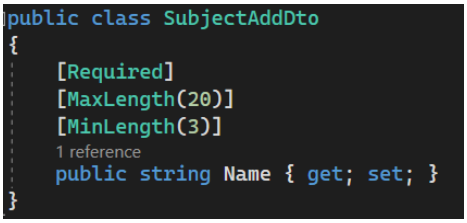
določene podatke o profilu avtorja (razred *UserDto*). Namesto seznama vseh odgovorov vključiva le njihovo število (*RepliesCount*). Ker je seznam predmetov ob prikazu vprašanj že prisoten v aplikaciji, pošljeva le identifikator predmeta (*SubjectId*).

Entitete podatkovne baze dokaj preprosto pretvorimo v razrede DTO z LINQ-metodo *Select*, ki omogoča izbiro le željenih podatkov (gl. 5.1.4.1).



Slika 21: Primer osnovnega razreda in pripadajočega DTO-razreda (lasten vir)

Podobno sva definirala takšne razrede tudi za sprejete podatke, ki obsegajo le podatke, ki jih je vnesel uporabnik. Tako nepooblaščenim uporabnikom preprečimo dodajanje in spreminjanje vsebin, do katerih nimajo dostopa.



Slika 22: Primer DTO-razreda za dodajanje podatkov (lasten vir)

Najpreprostejši takšen primer je razred za dodajanje novih predmetov (*SubjectAddDto*). Podatke, ki jih želimo sprejeti od odjemalcev, preprosto definiramo kot navadne lastnosti, lahko pa jim dodamo še dodatne atribute, s katerimi poudarimo njihovo obveznost (atribut *Required*) ali omejimo dolžino poslanega niza znakov (atributa *MinLength* in *MaxLength*).

Takšne razrede lahko neposredno uporabimo kot argumente metod za definicijo končnih točk v krmilnikih.

5.2.6.2 Implementacija krmilnikov

Pri strukturi MVC je obdelava zahtevkov (povezava URI-naslova in namena zahtevka z ustrezno metodo) naloga razredov, imenovanih krmilniki (angl. controllers). [30]

Krmilnik je v okolju ASP.NET Core predstavljen kot razred, ki deduje razred *ControllerBase* in ima nad svojo deklaracijo dodana atributa *ApiController* in *Route*. Slednji določa URI-naslov krmilnika. V oklepajih naveden niz *[controller]* pomeni ime razreda brez pripone *-Controller* (npr. *ForumController* → *Forum*).

Končne točke (endpoints) programskega vmesnika so predstavljene kot javne (*public*) metode krmilnika, označene z atributoma *Http + NamenZahtevka* (*HttpGet, HttpPost, HttpPut, HttpDelete* itd.) in *Route* (URI-naslov končne točke).

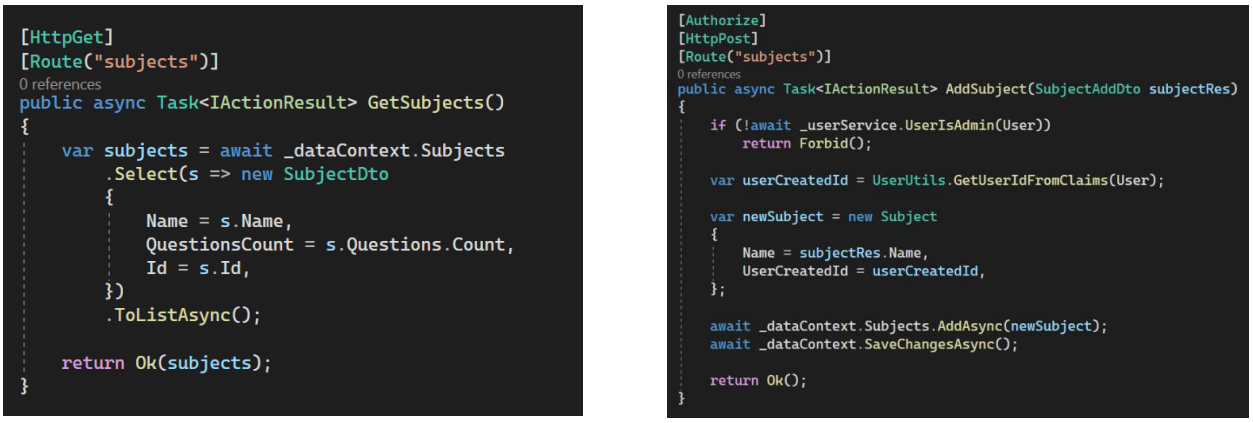
V metodah lahko dostopamo do vseh poslanih podatkov preko razreda *HttpContext*. Kot argumente omenjenih metod preprosto dodamo objekte, ki jih ogrodje samo razbere iz zahtevka (iz označenega dela URL ali JSON telesa zahtevka).

Metode vračajo podatkovni tip *IActionResult*, s katerim določimo status zahtevka (*Ok* – koda 200 ali uspešen zahtevek, *NotFound* – koda 404 ali ni najdeno, *BadRequest* – koda 400 ali neustrezen zahtevek itd.).

```
[ApiController]
[Route("[controller]")]
1 reference
public class ForumController : ControllerBase
{
    private readonly IUserService _userService;
    private readonly DataContext _dataContext;

    0 references
    public ForumController(DataContext dataContext,
                           IUserService userService)
    {
        _dataContext = dataContext;
        _userService = userService;
    }
```

Slika 23: Primer definicije krmilnika (lasten vir)

V krmilnik lahko preko konstruktorja preprosto dodamo tudi odvisnosti, saj zanje poskrbi upravljanje odvisnosti. Tako lahko neposredno kličemo podatkovno bazo z razredom *DataContext* prek vstavljene spremenljivke *_dataContext* in dostopamo do uporabniške storitve z vmesnikom *IUserService* prek vstavljene spremenljivke *_userService*.



```
[HttpGet]
[Route("subjects")]
0 references
public async Task<IActionResult> GetSubjects()
{
    var subjects = await _dataContext.Subjects
        .Select(s => new SubjectDto
        {
            Name = s.Name,
            QuestionsCount = s.Questions.Count,
            Id = s.Id,
        })
        .ToListAsync();

    return Ok(subjects);
}
```

```
[Authorize]
[HttpPost]
[Route("subjects")]
0 references
public async Task<IActionResult> AddSubject(SubjectAddDto subjectRes)
{
    if (!await _userService.UserIsAdmin(User))
        return Forbid();

    var userCreatedId = UserUtils.GetUserIdFromClaims(User);

    var newSubject = new Subject
    {
        Name = subjectRes.Name,
        UserCreatedId = userCreatedId,
    };

    await _dataContext.Subjects.AddAsync(newSubject);
    await _dataContext.SaveChangesAsync();

    return Ok();
}
```

Slika 24: Primer definicij končnih točk programskega vmesnika (lasten vir)

Leva slika prikazuje implementacijo metode krmilnika *GetSubjects*, ki predstavlja končno točko, s katero dobi uporabnik seznam vseh predmetov v podatkovni bazi. Do nje dostopamo s klicem na URI */forum/subjects* (ime krmilnika, ki mu sledi ime, definirano z atributom *Route*) z metodo GET.

V že opisanem postopku dobiva seznam vseh predmetov iz podatkovne baze in jih preoblikujeva v razred za prenos podatkov *SubjectDto*. Dobljen seznam vrneva uporabniku preko konstruktorja *Ok*, ki uporabniku vrne zahtevane podatke in mu sporoči, da je zahteva uspela (statusna koda 200).

Desna slika predstavlja prikazuje implementacijo metode krmilnika *AddSubject*, ki predstavlja končno točko, s katero lahko administrator doda nov predmet v podatkovno bazo. Do nje dostopamo s klicem na URI */forum/subjects* z metodo POST.

Kot argument metode dodamo objekt, za katerega želimo, da ga pošlje uporabnik. Ogrodje samo prepozna JSON iz telesa zahtevka in ga poskuša preoblikovati (»deserializirati«) v primerek željenega objekta. Ker sva v implementaciji razreda argumenta (slika 22) navedla atribut *Required*, ogrodje zavrne zahtevke, ki v telesu zahtevka nimajo omenjenega objekta, kot neustrezne (statusna koda 400).

V tej metodi najprej s storitvijo *UserService* preveriva, ali je uporabnik administrator. Če uporabnik ni administrator, vrneva rezultat *Forbid*, ki uporabnika obvesti, da nima dovolj pravic za dostop do te končne točke (statusna koda 403).

V nadaljevanju s pomočjo pomožnega razreda *UserUtils*, ki bere podatke iz prijavnega žetona v glavi zahtevka, prebereva identiteto uporabnika. Ime novega predmeta in identifikator (*Id*) uporabnika shraniva v nov primerek entitete *Subject*, ki ga po že opisanem postopku dodava v podatkovno bazo. Uporabniku sporočiva uspešnost zahtevka s klicem konstruktorja *Ok* (statusna koda 200).

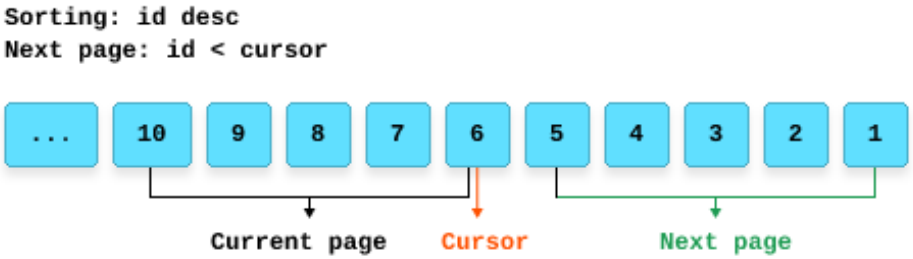
5.2.6.3 Paginacija

Pri poizvedbah vprašanj na forumu sva hitro ugotovila, da pošiljava veliko preveč podatkov, če ob vsakem zahtevku uporabniku pošljeva vse shranjene zapise. Tako se

delovanje programskega vmesnika močno upočasni, saj mora za vsak zahtevek obdelati veliko več podatkov, hkrati pa je na udaru tudi podatkovna baza. Posledično se upočasni tudi delovanje mobilne aplikacije, saj mora ta vse te podatke obdelati in prikazati.

Rešitvi te težave pravimo paginacija ali straničenje (angl. pagination). Gre za postopek, kjer odjemalec podatke zahteva v manjših skupinah oz. straneh. [31]

Sama sva se odločila za paginacijo, osnovano na kazalcu (angl. cursor-based pagination), saj je pristop iskanja po straneh pri mobilnih aplikacijah redkejši, hkrati pa sva želela izdelati odzivnejšo in bolj skalabilno rešitev. [32]



Slika 25: Shema delovanja paginacije, osnovane na kazalcu [72]

```
public class PageModel<T>
{
    1 reference
    public int TotalItems { get; set; }
    1 reference
    public int PageSize { get; set; }
    1 reference
    public int TotalPages { get; set; }
    1 reference
    public int Cursor { get; set; }
    1 reference
    public List<T> Data { get; set; }

    3 references
    public PageModel(int totalItems, int pageSize, int? cursor, List<T> data)
    {
        TotalItems = totalItems;
        Data = data;
        PageSize = pageSize;
        Cursor = (data.Count < totalItems && cursor != null) ? (int) cursor : 0;

        TotalPages = (int) Math.Ceiling((double) totalItems / pageSize);
    }
}
```

Slika 26: Definicija razreda za paginacijo (lasten vir)

Za implementacijo paginacije morava poslati še dodatne podatke in ne samo seznama elementov, zato sva se odločila definirati nov razred *PageModel*, ki ga bova vračala v zahtevah. Tukaj gre za posebno vrsto razreda, ki kot argument vzame tudi vrsto razreda (<*T*>) – t. i. generični razred (angl. generic class). Tako lahko uporabiva enak razred za vse tipe podatkov, ki jih želiva vrniti v aplikaciji.

Podatke vračava v spremenljivki *Data*, ki predstavlja seznam želenih tipov. Spremenljivka *Cursor* predstavlja kazalec, ki je lahko enak 0, kar predstavlja konec podatkov. Zraven pošljeva še omejitev oz. število elementov na eni strani v spremenljivki *PageSize*, število

vseh elementov, ki jih odjemalec lahko zahteva, v spremenljivki *TotalItems* in število strani, ki jih odjemalec lahko zahteva, v spremenljivki *TotalPages*.

```
var messages = await _dataContext.Messages
    .Where(c => c.ChatId == chatId)
    .OrderByDescending(m => m.Id)
    .WhereIf(cursor != 0, m => m.Id < cursor)
    .Take(pageSize)
    .Select(m => new MessageDto
    {
        Id = m.Id,
        Content = m.Content,
        UserId = m.UserCreatedId,
        Created = m.Created,
        PhotoIds = m.Images.Select(p => p.Id)
    })
    .ToListAsync();

var pageModel = new PageModel<MessageDto>(messagesCount, pageSize, cursor: messages.LastOrDefault()?.Id, messages);
```

Slika 27: Primer poizvedbe podatkovne baze s paginacijo (lasten vir)

Zgornja slika prikazuje poizvedbo za sporočila v pogovoru z identifikatorjem *ChatId* in predstavlja primer poizvedbe s paginacijo. V poizvedbi najprej filtriramo sporočila na samo tista, ki spadajo k iskanemu pogovoru (*Where*), nato jih razvrstimo padajoče po identifikatorju (*Id* – edinstvena zaporedna števila – *OrderByDescending*). Za paginacijo sta ključni poizvedbi, kjer vzamemo samo elemente, ki imajo *Id* manjši od kazalca in kjer vzamemo le zahtevano število elementov (*Take*).

Dobljen seznam elementov tako vstavimo v konstruktor že opisanega razreda *PageModel*, izpolnjen primerek katerega vrnemo uporabniku. Za vrednost kazalca vzamemo največji identifikator, ki ga dobimo z metodo *LastOrDefault*, ki v primeru praznega seznama vrne ničelno vrednost (*null*).

5.2.7 Funkcija sprotnega pogovora

Odjemalec lahko dobi z uporabo protokola HTTP osvežene podatke le, ko pošlje strežniku ponovno zahtevo. Ta pristop je primeren za večino funkcij spletnih aplikacij, a ima slabost, da uporabniku prikazujemo zastarele podatke, dokler jih ta ne osveži.

Ta omejitev nama je predstavljala še posebno težavo pri implementaciji pogovora, katerega ažurnost je za uporabniško izkušnjo zelo pomembna. Našla sva sledeče rešitve.

5.2.7.1 Izbrana rešitev

Pri izdelavi funkcije sprotnega pogovora (angl. realtime chat) sva si delo olajšala z Microsoftovo odprtokodno rešitvijo SignalR, ki je del ogrodja ASP.NET Core. [57] Knjižnica podpira vse pogosto uporabljene rešitve, opisane v prilogi, in sama izbere najustreznejše glede na odjemalca.

Delo s knjižnico je osnovano na posebnih t. i. vozliščih (angl. hubs), ki predstavljajo abstrakcijo za povezavo med odjemalcem in strežnikom. [33]

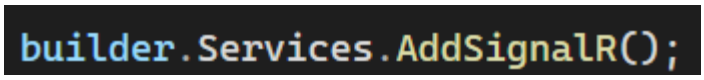
Slika 28: Definicija SignalR vozlišča za pogovor (lasten vir)

Vozlišče je v kodi predstavljeno kot razred, ki deduje vgrajen razred *Hub*. Vanj definiramo metode, ki jih lahko odjemalci kličejo na strežniku (angl. Remote Procedure Call – RPC). Definirala sva dve metodi: *DeleteMessage*, s katero lahko uporabnik izbriše svoje sporočilo, in *SendMessage*, s katero lahko uporabnik pošlje novo sporočilo.

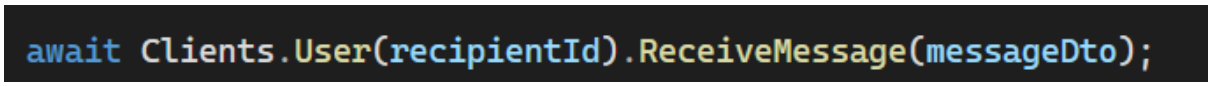
V definiciji dedovanja sva dodala še vmesnik *IChatHub*, s katerim definiramo imena metod, ki jih lahko iz strežnika sprožimo na odjemalcu (t. i. strongly-typed hub). Ob klicu metod se odjemalcu pošljejo tudi vsi argumenti metode.

Metodo *ReceiveMessage* kličeva, ko je uporabnik dobil novo sporočilo, metodo *DeleteMessage* pa, ko je sogovornik eno izmed sporočil izbrisal.

Vozlišče dodamo v zabojnik storitev s klicem metode *AddSignalR* v izhodiščni datoteki *Program.cs*.



Slika 29: Klic za dodajanje knjižnice SignalR v projekt (lasten vir)



Slika 30: Klic za pošiljanje sporočila določenemu uporabniku (lasten vir)

Zgornja slika prikazuje klic, s katerim sproživa metodo *ReceiveMessage*. Sporočilo pošljemo preko lastnosti *Clients*, ki vsebuje metode, s katerimi lahko dostopamo do povezanih odjemalcev. Primer takšne metode je *User*, ki kot argument vzame identifikator uporabnika, ki je v zgornjem primeru spremenljivka *recipientId* (identifikator prejemnika). Tako dobimo konfiguriran začasni (angl. transient) primerek vmesnika *IChatHub*, na katerem lahko neposredno kličemo metodo, ki jo želimo izvesti na povezanem odjemalcu.

Dogodke lahko sprožimo tudi iz krmilnikov preko vmesnika *IHubContext*, preko katerega lahko dostopamo do metod, definiranih v vmesniku *IChatHub*.

```
[Authorize]
[ApiController]
[Route("[controller]")]
1 reference
public class ChatsController : ControllerBase
{
    private readonly IHubContext<ChatHub, IChatHub> _chatHub;

    0 references
    public ChatsController(IHubContext<ChatHub, IChatHub> chatHub)
    {
        _chatHub = chatHub;
    }
```

Slika 31: Definicija krmilnika z vmesnikom IHubContext (lasten vir)

Slabost uporabe te knjižnice je, da dogodek odjemalcu pošljemo ne glede na to, ali je povezan z vozliščem. Tako lahko uporabniki nova sporočila spregledajo.

Knjižnica nima vgrajenega načina za preverjanje, ali je uporabnik povezan, vendar sva ugotovila, da je implementacija takšne funkcije dokaj preprosta, saj omogoča dodajanje kode vgrajenim funkcijam (angl. override), kot sta *OnConnectedAsync*, ki se izvede, ko se poveže nov uporabnik, in *OnDisconnectedAsync*, ki se izvede, ko uporabnik prekine povezavo.

Rešitev sva implementirala z definiranjem seznama tekstovnih spremenljivk, v katerem sva shranila identifikatorje povezanih uporabnikov (ob novi povezavi sva identifikator dodala, ob prekinitvi povezave pa odstranila). Dodala sva še statično metodo *UserIsOnline*, ki preveri, ali je uporabnik z navednenim identifikatorjem v seznamu povezanih uporabnikov.

Tako lahko preveriva, ali je uporabnik povezan, preden sproživa dogodek. Nepovezanemu uporabniku lahko tako namesto dogodka pošljeva potisno sporočilo.

```
if (UserIsOnline(recipientId))
{
    await Clients.User(recipientId).ReceiveMessage(messageDto);
} else
{
    var notification = new Notification
    {
        Content = $"Novo sporočilo od {senderFullName}",
        Heading = message.Content
    };

    await _notificationService.SendNotification(recipientId, notification);
}
```

Odjemalci se z vozliščem na strežniku povežejo preko URL-naslova, ki ga definiramo s klicem metode *MapHub* v datoteki *Program.cs*, s katerim določimo, kateri razred (npr. *ChatHub)* povežemo s katerim naslovom (npr. */chat_hub*).

```
app.MapHub<ChatHub>("/chat_hub");
```

Slika 32: Klic za določitev končne točke vozlišča za pogovor (lasten vir)

5.2.8 Slike

Pred izdelavo aplikacije sva ugotovila, da bo njena uporaba veliko lažja, če bo uporabnikom omogočena delitev slik za dodajanje konteksta vprašanjem in odgovorom.

Hitro sva ugotovila, da bo shranjevanje slik predstavljalo izziv, saj zanj obstaja več različnih pristopov, ki imajo vsak svoje prednosti in slabosti. Ugotovitve sva zbrala v prilogi.

5.2.8.1 Imgur

Sama sva po iskanju brezplačne alternative našla ameriško spletno storitev Imgur, ki je znana po gostovanju vsebin na socialnem omrežju Reddit. Za razliko od Amazonove rešitve ne omogoča popolnega nadzora nad vsemi naloženimi vsebinami, vendar je za najine potrebe povsem ustrezna, saj omogoča objavo slik preko njihovega REST programskega vmesnika, ki je popolnoma brezplačen za nekomercialne namene. Za njegovo uporabo je treba registrirati le aplikacijo, kar lahko storimo z brezplačnim računom. [34]

5.2.8.2 Implementacija

Za pošiljanje datotek v telesu HTTP-zahtevkov se v okolju ASP.NET Core uporablja vmesnik *IFormFile*, zato sva razredom za dodajanje sporočil, vprašanj in odgovorov dodala spremenljivko *Images* tipa *List<IFormFile>*. Tako lahko sprejmeva več slik v enem zahtevku.

```
2 references
public class ReplyAddDto
{
    [Required]
    5 references
    public string Content { get; set; }
    6 references
    public List<IFormFile>? Images { get; set; }
}
```

Slika 33: Razred z možnostjo sprejema slik (lasten vir)

Pri sprejemu razredov moramo dodati atribut *FromForm*, saj uporabljamo večdelne zahtevke (*multipart/form-data*), kar bova bolje predstavila pri predstavitvi nalaganja slik v poglavju o aplikaciji.

```
[HttpPost("replies/{questionId:int}")]
0 references
public async Task<ActionResult> AddReply(int questionId, [FromForm] ReplyAdd replyRes)
{
```

Slika 34: Definicije končne točke z možnostjo objave slik (lasten vir)

V končni točki za dodajanje objav preveriva, če je uporabnik poslal slike, in jih objaviva preko storitve *ImageService*.

Storitev *ImageService* sva definirala z vmesnikom *IImageService*, ki vsebuje eno metodo za nalaganje (*UploadImages*) in eno za brisanje slik (*DeleteImages*).

```
8 references
public interface IImageService
{
    6 references
    Task DeleteImages(List<Image> images);
    4 references
    Task<List<Image>> UploadImages(List<IFormFile> imageFiles);
}
```

Slika 35: Storitev za nalaganje in brisanje slik (lasten vir)

V implementaciji metode *UploadImages* vsako sliko naloživa v Imgur shrambo, nato pa podatke o sliki shraniva v seznam primerkov razreda *Images*, ki jih nato shraniva v podatkovno bazo (razred *Image*).

Najpomembnejši podatek o sliki je njen identifikator (*Id*), ki predstavlja končnico spletnega naslova gostovane slike (oblike *i.imgur.com/<ID slike>*). Uporabnikom lahko tako pošljeva le identifikatorje slik, s katerimi lahko generirajo spletni naslov slike in dostopajo do nje.

Imgur za objavo slik ne ponuja uradne knjižnice za okolje .NET, zato sva se odločila programski vmesnik klicati s HTTP klici preko razreda *HttpClient*. V funkciji, prikazani na spodnji sliki, opraviva POST klic na URL-naslov *https://api.imgur.com/3/image* s sliko v telesu in identifikatorjem aplikacije (*Client-Id*) v glavi zahtevka. Rezultat, vrnjen v JSON-obliki, deserializirava v primerek razreda *ImageUploadResult*, ki sva ga definirala po uradni dokumentaciji oblike odgovora (*https://apidocs.imgur.com/*).

```
1 reference
private async Task<ImageUploadResult> UploadImage(Stream imageStream)
{
    using var client = _httpClientFactory.CreateClient();
    client.DefaultRequestHeaders.Authorization = new AuthenticationHeaderValue("Client-ID", clientId);
    using var multipartFormContent = new MultipartFormDataContent();

    var fileStreamContent = new StreamContent(imageStream);
    fileStreamContent.Headers.ContentType = new MediaTypeHeaderValue("image/png");
    multipartFormContent.Add(fileStreamContent, name: "image");

    var response = await client.PostAsync("https://api.imgur.com/3/image", multipartFormContent);

    string message = await response.Content.ReadAsStringAsync();

    var imageResult = JsonConvert.DeserializeObject<ImageUploadResult>(message);

    if (imageResult is null || !imageResult.Success)
        throw new Exception();

    return imageResult;
}
```

Slika 36: Klic Imgur programskega vmesnika za objavo slike (lasten vir)

Za izbris slike Imgur uporablja t. i. izbrisno šifro (angl. delete hash), ki je edinstvena vrednost, potrebna za izbris slike iz njihove storitve, ki jo poleg drugih podatkov o sliki

shranjujeva v podatkovni bazi. S funkcijo, predstavljeno na spodnji sliki, opraviva delete klic (metoda *DeleteAsync*) na naslov *https://api.imgur.com/3/image/<vrednost izbrišne šifre>*, s katerim izbriševa sliko.

```
private async Task DeleteImage(string deleteHash)
{
    using var client = _httpClientFactory.CreateClient();
    client.DefaultRequestHeaders.Authorization = new AuthenticationHeaderValue("Client-ID", clientId);

    var response = await client.DeleteAsync("https://api.imgur.com/3/image/" + deleteHash);

    if (!response.IsSuccessStatusCode)
        throw new Exception();
}
```

Slika 37: Klic programskega vmesnika Imgur za izbris slike (lasten vir)

5.3 Avtentikacija

Najin cilj je bil narediti čim varnejšo aplikacijo, do katere bodo lahko dostopali le dijaki in učitelji, zato sva za prijavo želela uporabiti kar šolske račune, preko katerih bi lahko pridobila verodostojne informacije o uporabnikih. Takšen način filtriranja med uporabniki se je izkazal kot najzanesljivejši in za uporabnike najpreprostejši.

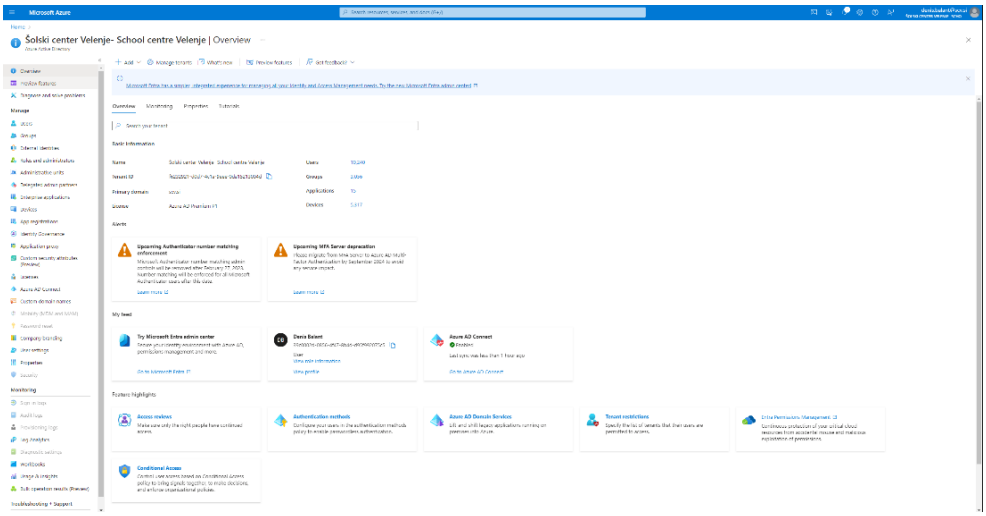
Slovenske izobraževalne ustanove za delo s spletnimi storitvami uporabljajo Arnesovo spletno infrastrukturo za enotno overjanje ArnesAAI (AAI – Authentication and Authorization Infrastructure), ki omogoča dostop do različnih aplikacij z enim uporabniškim imenom in geslom. Med drugim tako omogoča dostop tudi do Microsoftovih orodij vključno z zbirko pisarniških programov Office 365, zato so vse Arnes identitete povezane tudi z Microsoftom.

Čeprav Arnes omogoča registracijo lastnih aplikacij za njihovo storitev, sva se raje odločila za prijavo prek Microsofta, ker sva zanj našla veliko več dokumentacije in že izdelanih knjižnic, ki so nama močno olajšale delo.

Izbrala sva Microsoftovo storitev za oblačno prijavo in dostop do identitet uporabnikov Azure Active Directory (skrajšano Azure AD), ki je primer t. i. rešitve identiteta kot storitev (IDaaS – Identity as a Service). Ta razvijalcem aplikacij omogoča dodajanje t. i. enotne prijave (SSO – single sign-on), s čimer lahko preprosto omogočimo dostop obstoječim uporabnikom brez dodatne registracije. [35]

Prednost te izbire je tudi dostop do Microsoftovega združenega programskega vmesnika Microsoft Graph, s katerim lahko preprosto in hitro dostopava do identitet prijavljenih uporabnikov.

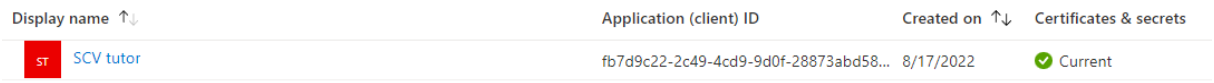
5.3.1.1 Nastavitev Azure AD



Slika 38: Domača stran portala Azure (lasten vir)

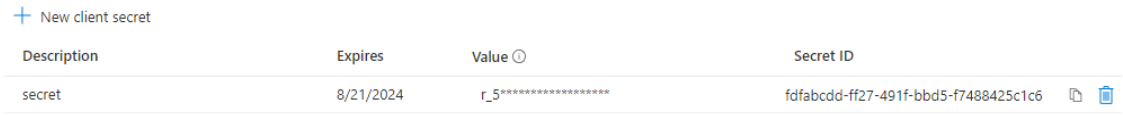
Za nastavitev storitve Azure AD moramo preko spletnega portala Azure konfigurirati njeno instanco, ki je lastna podjetju, ki se registrira zanjo, in je povezana z naročnino na to storitev. Imenujemo jo »najemnik« organizacije (angl. tenant). V najinem primeru je bil ta že nastavljen s strani spletnih administratorjev najinega šolskega centra.

Naslednji korak je registracija aplikacije, ki jo želimo povezati s platformo za prijavo (Microsoft identity platform). To lahko stori uporabnik z ustreznimi pravicami, zato sva za to prosila šolskega administratorja.



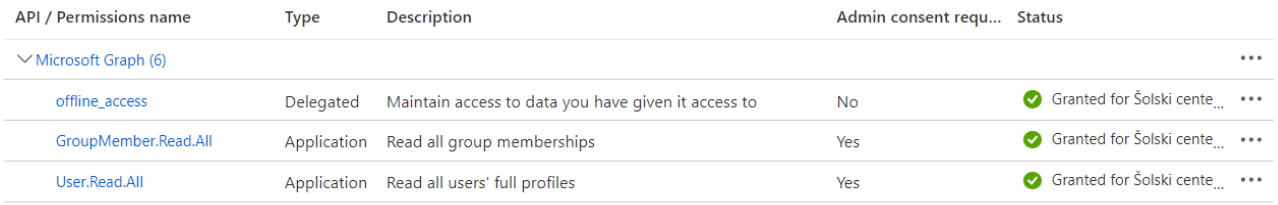
Slika 39: Registrirana aplikacija (lasten vir)

Za konfiguracijo zalednega dela sva ustvarila skrivnost odjemalca, saj je ta lahko varno shranjena na našem strežniku, kar pri odjemalcu (mobilni aplikaciji) ne bo mogoče.



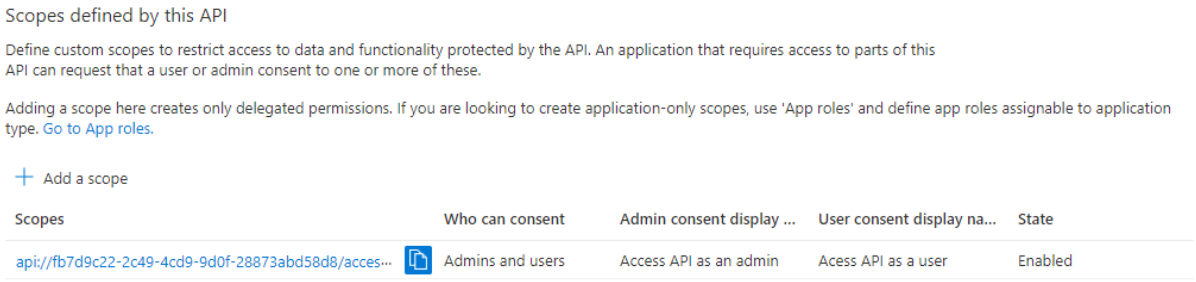
Slika 40: Generiranje vrednosti skrivnosti (lasten vir)

Nato sva nastavila pravice aplikacije za dostop do spletnih vmesnikov (angl. API permissions). Pri zalednih aplikacijah poznamo dve vrsti pravic: delegirane (angl. delegated), kjer aplikacija do podatkov dostopa z identiteto prijavljenega uporabnika, in aplikacijske (angl. application), kjer aplikacija do podatkov dostopa brez povezane identitete.



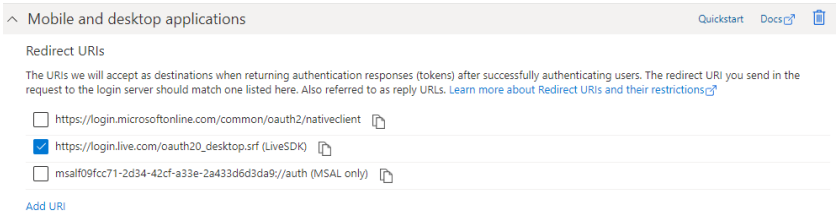
Slika 41: Pravice za aplikacijo (lasten vir)

Zgornja slika prikazuje pravice najine aplikacije. V zgornjem predelu (*Microsoft Graph*) sva določila pravici *GroupMember.Read.All*, s katero lahko bereva, člani katerih skupin (v najinem primeru razredov) so uporabniki, in *User.Read.All*, s katero lahko bereva dodatne podatke o uporabnikih.



Slika 42: Dodajanje okvirja programskega vmesnika (lasten vir)

Za dostop do programskega vmesnika morava ustvariti še okvir (angl. scope), ki ga morajo uporabniki navesti ob prijavi, da lahko dostopajo do najinega programskega vmesnika. Okvir ima obliko *api://<Id aplikacije>*.



Slika 43: Nastavitev preusmeritvenega naslova (lasten vir)

Za registraciji čelnega dela sva najprej konfigurirala preusmeritveni naslov. Sledila sva uradni dokumentaciji knjižnice za prijavo in ga nastavila na vrednost *https://login.live.com/oauth20_desktop.srf*, ki je standardna za nespletne aplikacije. [58]

5.3.1.2 API konfiguracija

Po registraciji obeh aplikacij sva morala dobljene konfiguracijske vrednosti (domeno (*Domain*) in identifikator najemnika (*TenantId*), identifikator (*ClientId*) in skrivnost odjemalca (*ClientSecret*)) dodati obema deloma aplikacije.

```
"AzureAd": {
  "Domain": "scv.si",
  "ClientId": "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX",
  "ClientSecret": "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX",
  "Instance": "https://login.microsoftonline.com/",
  "TenantId": "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"
},
```

Slika 44: Vrednosti za konfiguracijo avtentikacije v zalednem delu (lasten vir)

Vrednosti za zaledni del sva dodala h konfiguraciji v privzeti datoteki *appsettings.json*, do katere lahko v datoteki *Program.cs* preprosto dostopamo preko spremenljivke *builder.Configuration* v že razčlenjeni obliki.

```
builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
    .AddMicrosoftIdentityWebApi(builder.Configuration);
```

Slika 45: Klic za dodajanje avtentikacije prek Microsoft Identity (lasten vir)

Storitev za avtorizacijo sva projektu dodala s klicem metode *AddAuthentication*. Kot argument sva navedla avtentikacijsko shemo, ki je v najinem primeru privzeta Bearer avtentikacija z JWT-žetoni.

Klic sva verižila še z metodo *AddMicrosoftIdentityWebApi*, ki je na voljo v uradni knjižnici *Microsoft Identity Web*. Omenjena metoda omogoči avtentikacijo z žetonom, pridobljenim preko Microsoftove storitve za prijavo.

```
[Authorize]
[HttpPost("subjects")]
0 references
public async Task<ActionResult> AddSubject(SubjectAddDto subjectRes)
{
```

Slika 46: Atribut [Authorize] (lasten vir)

Z nastavljeno avtentikacijo lahko preprosto omejimo dostop do končnih točk z atributom *Authorize*, ki neprijavljenim uporabnikom vrne sporočilo, da niso prijavljeni – *Unauthorized* (statusna koda 401).

Takšen način dodajanja avtentikacije olajša tudi dostop do uporabnikovih podatkov, saj so vsi njegovi podatki, zapisani v tovoru žetona, v krmilnikih dostopni kot instanca razreda *ClaimsPrincipal* preko konteksta zahtevka (*HttpContext.User*).

```
27 references
public static class UserUtils
{
    23 references
    public static string GetUserIdFromClaims(ClaimsPrincipal user)
        => user.FindFirstValue(ClaimTypes.NameIdentifier);

    1 reference
    public static string GetUserEmailFromClaims(ClaimsPrincipal user)
        => user.FindFirstValue(ClaimTypes.Name);

    2 references
    public static string GetUserSurnameFromClaims(ClaimsPrincipal user)
        => user.FindFirstValue(ClaimTypes.Surname);

    2 references
    public static string GetUserNameFromClaims(ClaimsPrincipal user)
        => user.FindFirstValue(ClaimTypes.GivenName);
```

Slika 47: Pomožni razred UserUtils (lasten vir)

Za branje podatkov z žetonov sva ustvarila statični razred *UserUtils*, v katerem sva definirala metode, s katerimi sva prebrala vrednost trditve iz podanih podatkov o uporabniku s klicem metode *FindFirstValue*. Kot argumente funkcij sva navedla imena trditev (njen ključ), ki so shranjene v konstantah razreda *ClaimTypes*.

5.3.1.3 Klicanje vmesnika MS Graph

Kot že omenjeno, lahko s prijavo prek Microsofta dostopava do podatkov o uporabnikih prek programskega vmesnika Microsoft Graph. Klice v okolju .NET dodatno olajša Microsoftova uradna knjižnica *Microsoft Identity Web MicrosoftGraph*. [59]

```
var clientSecretCredential = new ClientSecretCredential(tenantId, clientId, clientSecret);
_graphServiceClient = new GraphServiceClient(clientSecretCredential);
```

Slika 48: Konfiguracija potrebnih razredov za povezavo z MS Graph (lasten vir)

Spodnja slika prikazuje primer klica vmesnika za pridobivanje profilne slike uporabnika glede na njegov e-poštni naslov. Prvi del poizvedbe je »grajenje« zahtevka, ki ga zaključimo s klicem metode *Request*. Zahtevek nato pošljemo z veriženo metodo *GetAsync*.

```
3 references
public async Task<Stream?> GetProfilePictureFromMicrosoft(string userEmail)
{
    try
    {
        var photo = await _graphServiceClient.Users[userEmail].Photo.Content.Request().GetAsync();
        return photo;
    } catch (ServiceException)
    {
        return null;
    }
}
```

Slika 49: Zahtevek za profilno sliko uporabnika (lasten vir)

Spodnja slika prikazuje klic vmesnika, s katerim izveva, kateremu oddelku in letniku pripada uporabnik glede na njegov šolski e-poštni naslov. Na našem šolskem centru ima

vsak razred ustvarjeno svojo skupino oblike *letnik.oddelek* (npr. 4. B), ki ima ustvarjen enakoimenski e-poštni naslov (npr. 4.B@scv.si).

S poizvedbo v spremenljivki *groupsData* dobiva vse skupine, katerih član je uporabnik.

Razred dijaka ugotoviva s primerjavo imena skupine z uporabo t. i. regularnega izraza (angl. regular expression, skrajšano regex) – niza znakov, s katerim definiramo iskalni vzorec v nizu znakov, ki sva ga shranila v konstanti *CLASS_REGEX*. Izraz preveri, če se ime začne s številom od 1 do 4, ki mu sledi pika, ki ji sledijo od 1 do 3 velike tiskane črke – ugotoviva, ali je v formatu imena razreda.

Niz znakov shraniva v primerku razreda *Regex*. Ta ima metodo *IsMatch*, s katero lahko preverimo ujemanje v poljubnem nizu. To metodo uporabiva v poizvedbi na seznamu vseh skupin z metodo *FirstOrDefault*, ki vrne prvi primerek, ki se ujema z danim pogojem ali privzeto vrednost razreda (*null*).

Z operatorjem *?.*, ki dopušča ničelno vrednost (*null*), dostopava do atributa *DisplayName*, ki predstavlja ime skupine oz. oddelka (npr. 4. B). Prvi znak imena tako predstavlja letnik, ki ga obiskuje uporabnik, del za piko pa njegov oddelek.

Imena oddelkov so različna za vsako šolo na šolskem centru, zato lahko z njimi ugotoviva, katero šolo obiskuje uporabnik.

```
2 references
public async Task<string?> GetClass(string userEmail)
{
    const string CLASS_REGEX = @"^[1-4].\s?([\p{L}]){1,3}$";
    var groupsData = await _graphServiceClient.Users[userEmail].MemberOf.Request().GetAsync();
    var groupsPage = groupsData.CurrentPage.Cast<Group>();

    var classNameRegex = new System.Text.RegularExpressions.Regex(CLASS_REGEX);
    var classGroup = groupsPage.FirstOrDefault(g => (g.MailEnabled ?? false) && classNameRegex.IsMatch(g.DisplayName));
    var className = classGroup?.DisplayName;

    return className;
}
```

Slika 50: Zahtevek za iskanje uporabnikovega oddelka (lasten vir)

```
private static readonly Dictionary<School, string[]> ClassSuffixes = new()
{
    { School.GIMNAZIJA, new[] { "A", "B", "C", "Š", "U"} },
    { School.STORITVENA, new[] { "EKT", "GA", "GH", "GTT", "P", "PBO" } },
    { School.STROJNA, new[] { "AS", "ASPT", "GR", "GT", "GTD", "ISI", "MHT", "OVT", "PT", "PTP", "S", "VHMT", "VOKO", "VSTM" } },
    { School.RAČUNALNIŠKA, new[]  { "EL", "ET", "PTI", "TM", "TRA", "TRB" }}
};

1 reference
public static School? GetUserSchool(string className)
{
    var suffix = className.Split(".")[1];
    return ClassSuffixes.FirstOrDefault(c => c.Value.Contains(suffix)).Key;
}
```

Slika 51: Pomožna funkcija za iskanje uporabnikove šole (lasten vir)

5.3.1.4 Flutter konfiguracija

Za prijavo v mobilni aplikaciji sva se odločila za najbolje vzdrževan tovrsten paket *aad_oauth* švicarskega podjetja EarlyByte, ker Microsoft za okolje Flutter ne ponuja uradne rešitve. [58]

Konfiguracijske vrednosti za čelni del sva zapisala v konstruktor razreda *Config*, ki je del omenjene knjižnice. Med argumente sva morala dodati tudi *navigatorKey*, ki sva ga prav tako navedla v konstruktorju glavnega gradnika aplikacije *MaterialApp*. Z njim namreč knjižnica opravi navigacijo na spletno stran za prijavo preko knjižnice *webview_flutter*. [60]

```
config = Config(
  tenant: 'xxxxxxxxxxxxxxxxxxxxxx',
  clientId: 'xxxxxxxxxxxxxxxxxxxxx',
  scope: 'api://xxxxxxxxxxxx offline_access',
  redirectUri: "https://login.live.com/oauth20_desktop.srf",
  navigatorKey: _navigatorKey,
);
```

Slika 52: Konfiguracija avtentikacije v mobilni aplikaciji (lasten vir)

Z avtentikacijo upravljamo preko razreda *AadOauth*, ki kot argument zahteva zgoraj opisano konfiguracijo. Razred ima definirane metode, kot sta *login*, ki poskrbi za prijavo, osveževanje žetonov in njihovo pridobivanje iz varne shrambe (preko paketa *flutter_secure_storage* [61]) ter vrne dostopni žeton in *logout*, ki izbriše vse podatke o uporabniku iz naprave in ga odjavi.

```
oauth = AadOAuth(config);
```

Slika 53: Kreacija razreda Oauth (lasten vir)

Do funkcij za avtentikacijo sva želela dostopati iz različnih gradnikov aplikacije, zato sva jih dodala v t. i. ponudnik (angl. provider), ki ga bova natančneje predstavila v podpoglavju 5.6.5.2.

```
await oauth.login();
```

Slika 54: Metoda za prijavo uporabnika (lasten vir)

```
await oauth.logout();
```

Slika 55: Metoda za odjavo uporabnika (lasten vir)

5.4 Potisna sporočila

Ker sva želela uporabnike obveščati o dogajanju v najini aplikaciji tudi izven nje, sva ji dodala možnost sprejemanja t. i. potisnih sporočil.

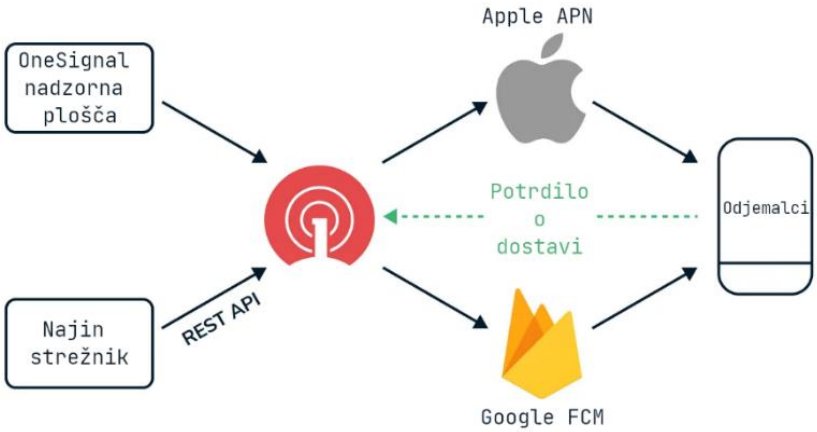
Potisna sporočila so primer pojavnega obvestila, ki ga zaledni strežnik »potisne« aplikaciji. Najpogosteje so uporabljena na mobilnih in namiznih, značilna pa so tudi za spletne aplikacije.

Vsaka platforma (iOS, Android, Windows ipd.) ima svoje smernice razvoja, standarde in storitev za dostavo sporočil (OSPNS – angl. Operating system push notifications service). [36]

Slika 56: Shema delovanja potisnih sporočil [36]

5.4.1 OneSignal

Ker nisva hotela pisati ločene kode za obe platformi, sva izbrala brezplačno večplatformno rešitev ameriškega podjetja OneSignal, ki ponuja paket za razvoj programske opreme (angl. SDK) za številne platforme, preko katerega lahko z ustrezno nastavitvijo pošljemo potisna sporočila na več platform hkrati iz našega strežnika preko njihovega spletnega vmesnika.



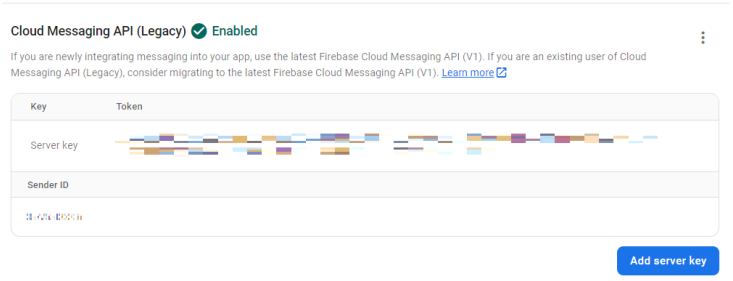
Slika 57: Shema delovanja platforme OneSignal [37]

5.4.1.1 Nastavitev za platformo Android

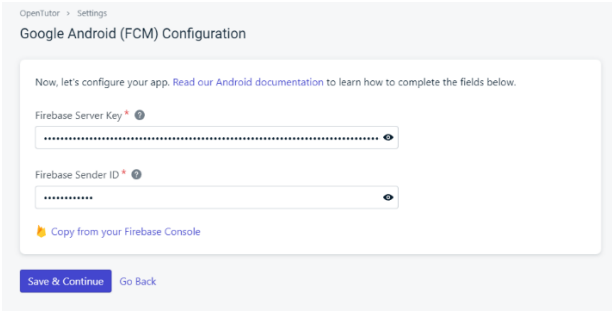
Potisna sporočila za platformo Android so dostavljena preko Googlove storitve Firebase Cloud Messaging (FCM), ki predstavlja del razvojne platforme Firebase.

Za avtentikacijo s storitvijo FCM-platforma OneSignal uporablja strežniški ključ (angl. server key) in identifikator pošiljatelja (angl. sender ID), ki ga lahko generiramo z brezplačnim Firebase projektom.

Omenjen ključ dodamo v nastavitvah za dodajanje platforme v administratorskem portalu OneSignal.

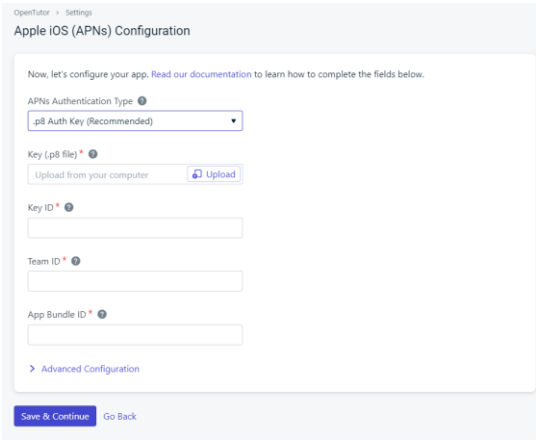


Slika 58: Konfiguracija žetona za potisna sporočila za platformo Android prek portala Firebase (lasten vir)



Slika 59: OneSignal konfiguracija za platformo Android (lasten vir)

5.4.1.2 Nastavitev za platformo iOS



Slika 60: OneSignal konfiguracija za platformo iOS (lasten vir)

Potisna sporočila so za platformo iOS dostavljena preko Applove storitve Apple Push Notification service (APNs), za dostop do katere potrebujemo Applov plačljiv račun za razvijalce (angl. Apple Developer Account).

Za avtentikacijo s storitvijo APNs platforma OneSignal uporablja t. i. ključ oz. žeton za avtentikacijo s storitvijo APNs (angl. APNs Auth Key), ki je za razliko od Androidovega strežniškega ključa v obliki datoteke s končnico p8. Za registracijo žetona moramo dodati še identifikatorje ključa (angl. Key ID), ekipe (edinstven identifikator uporabljenega računa za razvijalce, angl. Team ID) in skupka aplikacije (angl. App Bundle ID). Omenjene vrednosti dodamo v nastavitvah za dodajanje platforme.

5.4.1.3 Nastavitev mobilne aplikacije

Aplikacijo sva povezala s platformo OneSignal s knjižnico *onesignal_flutter*, ki jo je bila treba nastaviti. [62]

Vso kodo, povezano s potisnimi sporočili, sva zbrala v statične metode abstraktnega razreda *NotificationsService*.

Metoda *setup* nastavi stopnjo beleženja (angl. log level) s klicem metode *setLogLevel* (prvi argument predstavlja stopnjo za konzolo, drugi pa za prikazovanje opozoril v aplikaciji), ID aplikacije s klicem metode *setAppId* in prosi uporabnika za dovoljenje za prejemanje potisnih sporočil s klicem metode *promptUserForPushNotificationPermission*. Omenjeno metodo kličeva ob vsakem odprtju aplikacije.

Z metodo *setExternalUserId* lahko nastaviva identifikator uporabnika, s katerim je povezana naprava. Tako za pošiljanje potisnega sporočila ne potrebujeva identifkatorja naprave, ampak le identifikator uporabnika. Metodo kličeva ob novi prijavi uporabnika.

Z metodo *removeExternalUserId* izbriševa povezavo identifikatorja uporabnika z napravo. Metodo kličeva ob odjavi uporabnika.

```dart
abstract class NotificationsService {
  static void setup() {
    OneSignal.shared.setLogLevel(OSLogLevel.warn, OSLogLevel.none);
    OneSignal.shared.setAppId("XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX");
    OneSignal.shared
        .promptUserForPushNotificationPermission()
        .then((accepted) => debugPrint("Potrjena potisna sporočila"));
  }

  static Future<void> setExternalUserId(String userId) async {
    await OneSignal.shared.setExternalUserId(userId);
  }

  static Future<void> removeExternalUserId() async {
    await OneSignal.shared.removeExternalUserId();
  }
}
```

Slika 61: Metode, povezane z potisnimi sporočili (lasten vir)

5.4.1.4 Pošiljanje potisnega sporočila s strežnika

Za pošiljanje potisnega sporočila z našega strežnika k platformi OneSignal se uporablja njihov RESTful spletni vmesnik. Za olajšano integracijo obstaja veliko knjižnic, ki ponujajo že izdelane metode za klic končnih točk. Sama sva se odločila za knjižnico *OneSignal RestAPIv3 Client*. [63]

Kodo za pošiljanje potisnih sporočil sva združila v razred *NotificationApiService* s pripadajočim vmesnikom *INotificationApiService*. Klici vmesnika potekajo preko razreda *OneSignalClient*, primerek katerega ustvariva v konstruktorju razreda z navedbo API-ključa kot argument njegovega konstruktorja. Njegov primerek shranjujeva v spremenljivki *_signalClient*.

Razred vsebuje le metodo *SendNotification* z argumentoma *userId*, ki zahteva identifikator uporabnika, ki mu želimo poslati potisno sporočilom, in obvestilo, ki ga želimo poslati.

Obvestilo prikaževa z razredom *Notification*, ki vsebuje polji *Heading* (naslov) in *Content* (vsebina).

Klic konfiguriramo s primerkom razreda *NotificationCreateOptions*, v konstruktor katerega navedemo identifikator aplikacije in identifikatorje uporabnikom, ki jim želimo poslati sporočilo. Naslov in vsebino dodamo s klicema metod *Add* na ustreznih poljih konfiguracije. Sporočilo pošljemo s klicem metode *CreateAsync*.

```
4 references
public class NotificationApiService : INotificationApiService
{
    private readonly ILogger<NotificationApiService> _logger;
    private readonly OneSignalClient _signalClient;

    0 references
    public NotificationApiService(ILogger<NotificationApiService> logger)
    {
        _signalClient = new OneSignalClient("XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX");
        _logger = logger;
    }

    4 references
    public async Task SendNotification(string userId, Notification notification)
    {
        var options = new NotificationCreateOptions
        {
            AppId = new Guid("XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"),
            IncludeExternalUserIds = new List<string>
            {
                userId
            }
        };

        options.Headings.Add(LanguageCodes.English, notification.Heading);
        options.Contents.Add(LanguageCodes.English, notification.Content);

        _logger.LogInformation("Uporabniku {} pošiljam notification {}.", userId, notification.Heading);

        await _signalClient.Notifications.CreateAsync(options);
    }
}
```

Slika 62: Storitev za pošiljanje potisnih sporočil (lasten vir)

```
5 references
public record Notification(string Heading, string Content);
```

Slika 63: Definicija zapisa za potisno sporočilo (lasten vir)

5.5 Načrtovanje mobilne aplikacije

Po končanem razvoju programskega vmesnika in podatkovne baze sva začela z izdelavo mobilne aplikacije. Ker sta red in urejenost ključna pri razvoju na videz lepih uporabniških vmesnikov, ki so hkrati tudi uporabni, sva izgled najine aplikacije najprej načrtovala v priljubljenem brezplačnem spletnem oblikovalskem orodju Figma.

5.5.1 Načrtovanje mobilne aplikacije z urejevalnikom Figma

5.5.1.1 Figma

Figma je spletni urejevalnik vektorske grafike, namenjen načrtovanju in izdelavi prototipov spletnih strani, mobilnih in namiznih aplikacij ipd. Na voljo sta tudi namizna in mobilna aplikacija. Slednja izmed njih omogoča ogled in interakcijo s prototipom kar na telefonu.

Orodje podpira tudi zelo močna skupnost oblikovalcev uporabniških vmesnikov, ki redno objavlja že vnaprej načrtovane komponente za oblikovanje. Vse to občutno pospeši postopek dela.

5.5.1.2 Oblikovanje aplikacije

Preden sva začela izrisovati različne zaslone, ki bodo gradili najino aplikacijo, sva definirala osnovne stile in tipografijo. Inspiracijo za dizajn sva črpala iz oblikovalskega stila claymorphism, ki naju je prepričal zaradi svojega mehkega in vabljivega videza.

Ker sva na področju dizajna uporabniških vmesnikov začela praktično brez predznanja, sva si pomagala z različnimi internetnimi viri. Največ sva se naučila s pomočjo YouTube kanalov DesignCode in Malewicz.

Slika 64: Primer claymorphic dizajna [38]

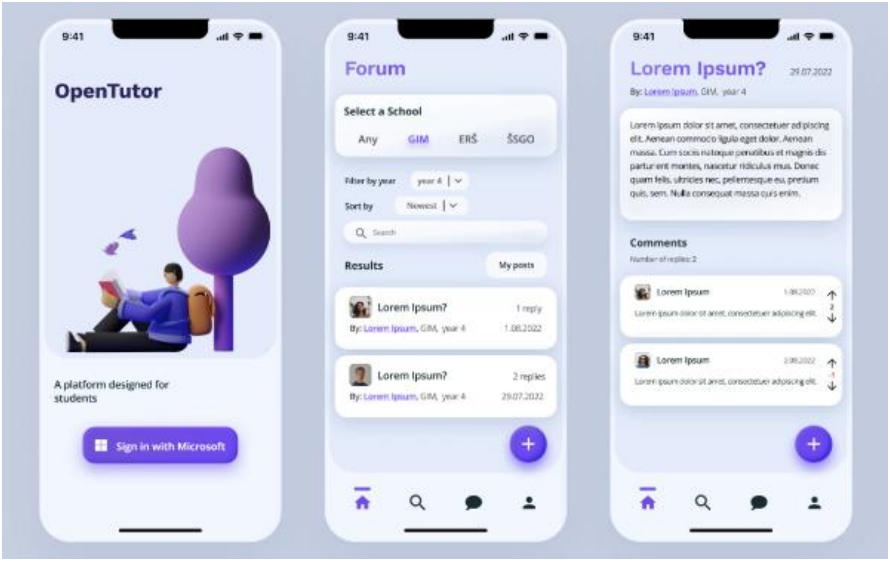
Izbrala sva šest osnovnih barv, ki se med seboj ujemajo, in dve različni pisavi (Work Sans za naslove in podnaslove in Open Sans za ostali tekstovni material). Da sva se lahko držala omejitev mobilnih naprav, sva uporabila tudi brezplačno dostopen skupek predlog, sestavljen iz že narejenih komponent, iOS 15 Toolkit, ki nama je nudil smernice pri dizajnu.

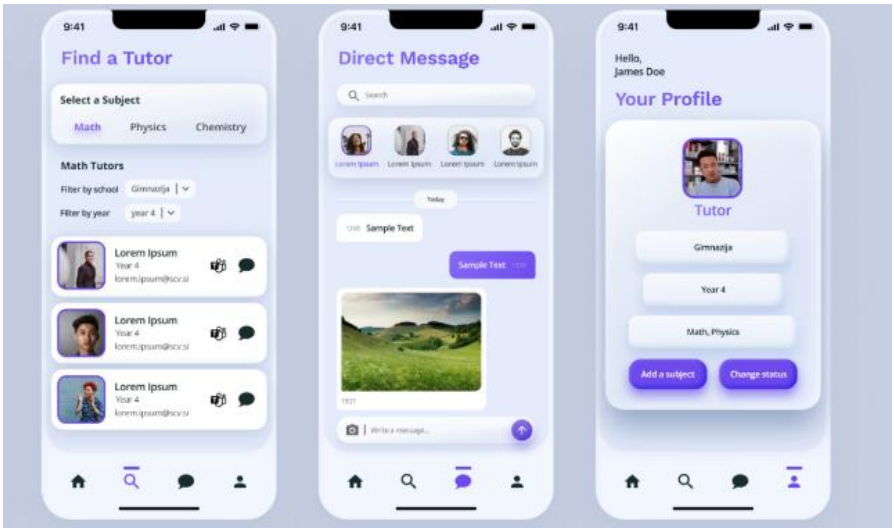
Slika 65: Izbira osnovnih barv in tipografije znotraj Figme (lasten vir)

Ko sva končala z definiranjem osnovnih stilov, sva začela izdelovati zaslone, ki bodo gradili najino mobilno aplikacijo. Pri oblikovanju sva se držala definiranih slogov in najinega oblikovalskega jezika, hkrati pa sva ohranjala konstantne marže (angl. margins) in oblazinjenja (angl. padding) skozi celotno aplikacijo.

Različna polja sva s pisnim in slikovnim materialom napolnila s pomočjo Figminih vtičnikov, kot sta vtičnik za slike Unsplash in vtičnik za tekst Lorem Ipsum, za vstavljanje brezplačnih in odprtih ikon pa sva uporabila vtičnik Material Design Icons.

Na koncu sva oblikovala sledeči dizajn aplikacije, ki nama je ustrezal.





Slika 66: Končni dizajn najine aplikacije (lasten vir)

5.6 Razvoj Flutter mobilne aplikacije

Po zaključenem oblikovanju aplikacije sva se lotila njenega razvoja. Da sva aplikacijo lahko testirala kar na najinih računalnikih, sva potrebovala ustrezne emulatorje, ki omogočajo dostop do virtualnih mobilnih naprav. Uporabo Android emulatorjev nama je omogočalo integrirano razvojno okolje, Android Studio, ki je na voljo za Windows, Linux in Mac naprave, uporabo Applovih emulatorjev pa Xcode, ki je na voljo le za Applove računalnike.

Kot tekstovni urejevalnik sva skozi celoten razvoj uporabljala Visual Studio Code, saj je dokaj lahek, hkrati pa nama je nudil mnogo vtičnikov, ki so razvoj Flutter mobilne aplikacije pospešili. Sploh uporaben je uradni Flutter vtičnik, ki olajša zaganjanje, ponovni zagon in oblikovanje ali preoblikovanje kode.

Ker nama je Flutter ponudil precej drugačen način izgradnje uporabniških vmesnikov, v katerem sva dokaj neizkušena, sva si med celotno izdelavo pomagala z uradno dokumentacijo in ostalimi spletnimi viri, sploh s platofrmo Stack Overflow in z raznimi YouTube kanali.

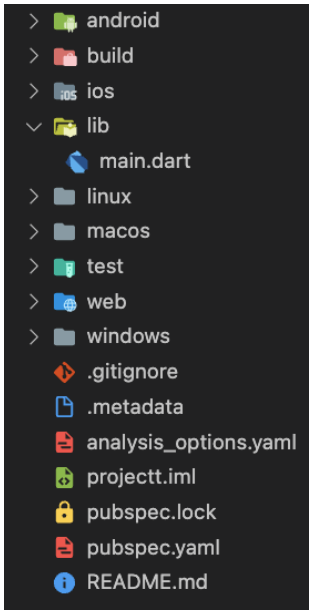
5.6.1 Namestitev okolja Flutter

Za razvoj Flutter aplikacij potrebujemo ustrezen paket za razvoj programske opreme (SDK – angl. software development kit), ki ga prenesemo iz uradne spletne strani projekta (https://docs.flutter.dev/development/tools/sdk/releases) v obliki ZIP-arhiva, ki ga ekstrahiramo v poljubno mapo.

Mapa *bin* v paketu vključuje vsa orodja ukazne vrstice, ki omogočajo generacijo novih projektov, posodabljanje razvojnega okolja in njegovo diagnosticiranje. Da lahko do teh orodij dostopamo iz katerekoli mape, mapo *bin* dodamo h globalni spremenljivki poti (angl. PATH).

5.6.2 Generacija novega Flutter projekta in osnovne mape ter datoteke

Najprej sva preko ukazne vrstice z ukazom *flutter create »ime aplikacije«* ustvarila osnovni Flutter projekt in inicializirala Git repozitorij.



Slika 67: Datotečna struktura osnovnega Flutter projekta (lasten vir)

Datoteka *.gitignore* vsebuje imena in končnice datotek, ki naj jih orodje Git ne spremlja. *Pubspec.lock* je pomožna datoteka, ki je ustvarjena na podlagi datoteke *pubspec.yaml*. V prvi so navedene specifične verzije vsake odvisnosti paketov, drugo pa uporabljamo za opravljanje z odvisnostmi aplikacije (paketi že pripravljene kode, ki jih upravljamo z upraviteljem paketov jezika Dart, Pub, slikovno gradivo in različne pisave), definiranje opisa, imena in verzije aplikacije itd. Datoteka *pubspec.yaml* je torej za razvoj in objavo aplikacije ključna.

V datoteko *README.md* navadno vključimo kratek opis našega projekta in ostale potrebne podatke, ki jih želimo prikazati ostalim ljudem. V odprtokodnih aplikacijah navadno služijo kot neke vrste dokumentacija za razvijalce, ki sodelujejo v njenem razvoju.

Mape *android, ios, web, linux, macos* in *windows* vsebujejo Flutter projekt za vsako od naštetih platform. Tega lahko po končanem razvoju prevedemo v platformi lastno aplikacijo. V te mape redko posegamo. Največkrat moramo vanje dodati nekatere konfiguracije za odvisnosti, ki jih naša aplikacija potrebuje za ustrezno delovanje na specifičnem operacijskem sistemu.

Mapa *build* je ustvarjena, ko so Flutterjeva orodja za izgradnjo (angl. Flutter build tools) prvič izvedena. Vsebuje generirane datoteke, potrebne za zagon aplikacije na različnih platformah. Vsaka platforma ima svojo podmapo.

Razvijalcem najpomembnejša mapa je mapa *lib* (*library*), saj se v njej nahajajo vse datoteke, ki vsebujejo kodo aplikacije. Flutter pri generiranju projekta sam ustvari osnovno datoteko *main.dart*, ki velja za izhodiščno točko vsakega Flutter projekta.

5.6.3 Osnove Flutterja in začetek projekta

Po inicializaciji projekta sva se lotila pisanja kode. Kot sva že omenila, pri razvoju vso Dart kodo pišemo v mapo *lib*, v kateri se nahaja izhodiščna datoteka vsake Flutter aplikacije *main.dart*, ki na začetku vsebuje predlogo, v kateri so predstavljeni nekateri ključni koncepti razvoja z ogrodjem Flutter.

Uporabniške vmesnike gradijo drevesa ponovno uporabnih gradnikov (angl. widgets), ki jih lahko napišemo sami ali pa so že na voljo znotraj ogrodja. Takšen pristop naredi UI-strukturo deklarativno in razširljivo. Najosnovnejši gradniki so nam na voljo preko paketa oz. datoteke *material.dart*, ki se nahaja znotraj generirane mape *flutter*. V datoteko jih pokličemo s ključno besedo *import*.

```
import 'package:flutter/material.dart';
```

Slika 68: Uvozni stavek v programskem jeziku Dart (lasten vir)

```
void main() {
  runApp(const MyApp());
}
```

Slika 69: Izhodiščna metoda vsake Flutter aplikacije (lasten vir)

```
class MyApp extends StatelessWidget {
  const MyApp({super.key});

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter Demo',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: const MyHomePage(title: 'Flutter Demo Home Page'),
    );
  }
}
```

Slika 70: Gradnik brez stanja in gradnik MaterialApp (lasten vir)

Začetna točka datoteke *main.dart,* ki je hkrati tako tudi začetna točka celotnega projekta, je funkcija *main*. Znotraj nje kličemo metodo *runApp*, ki kot argument vzame izhodiščni gradnik *MyApp*.

Gradniki so v ogrodju Flutter predstavljeni kot podrejeni razredi (angl. extended class), ki jim dodamo svojo implementacijo (pravimo, da jih »povozimo«, angl. override). Izgled gradnika določimo v metodi *build,* ki vrne kombinacijo gradnikov, ki jih želimo prikazati.

To dosežemo s klicem konstruktorjev obstoječih gradnikov in posredovanjem imenovanih argumentov (angl. named arguments – npr. *title, theme, home*). Ti so različnih definiranih tipov, vključno z drugimi gradniki, kar omogoča, da ustvarimo gnezden vmesnik, sestavljen iz več med seboj povezanih gradnikov.

Ogrodje to metodo pokliče, ko hoče izrisati gradnik. Prvič se to zgodi, ko ga vstavi v drevo, ponovno pa ga izriše, ko se njegove odvisnosti spremenijo, omogoča pa tudi, da uporabnik sam zahteva njegov ponovni izris (funkcija *setState*).

V osnovi lahko imajo gradniki stanje (angl. stateful – dedujejo razred *StatefulWidget*) ali pa so brez njega (angl. stateless – dedujejo razred *StatelessWidget*), kakršen je tudi gradnik *MyApp*. Gradniki brez stanja so nespremenljivi in ne nosijo dinamičnih podatkov, medtem ko lahko v gradnikih s stanjem prikazujemo spreminjajoče podatke.

Metoda *build* gradnika *MyApp* vrne vnaprej narejen gradnik *MaterialApp,* ki omogoča uporabo navigacijskih in tematskih gradnikov, potrebnih za razvoj aplikacij po načinu razvoja Material Design. Argumentu *home,* ki je odgovoren za prikazovanje osnovne strani aplikacije ob ustreznem delovanju, je pripisan gradnik s stanjem, imenovan *MyHomePage*. Ta je torej prvi gradnik, ki je prikazan uporabnikom aplikacije.

```dart
class MyHomePage extends StatefulWidget {
  const MyHomePage({super.key, required this.title});

  final String title;

  @override
  State<MyHomePage> createState() => _MyHomePageState();
}
```

Slika 71: Razširitev gradnika StatefulWidget (lasten vir)

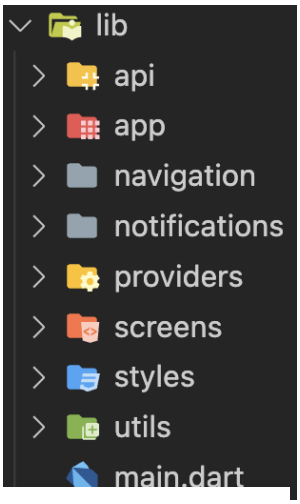
```dart
class _MyHomePageState extends State<MyHomePage> {
  int _counter = 0;

  void _incrementCounter() {
    setState(() {
      _counter++;
    });
  }

  @override
  Widget build(BuildContext context) {
```

Slika 72: Podrazred stanja _MyHomePageState (lasten vir)

V Flutterju so gradniki s stanjem implementirani z dvema razredoma. Prvi razširi nespremenljivi (angl. immutable) gradnik *StatefulWidget*, drugi pa je spremenljivi (angl. mutable) podrazred stanja (angl. *State*). V prvem funkcija *createState* vrne primerek objekta tipa *State*, katerega razširi gradnik *_MyHomePageState*, znotraj katerega je implementirana praktično vsa logika, odgovorna za obdelavo in prikazovanje dinamičnih podatkov vključno z metodo *build*.

V predstavljenem primeru je edini spremenljivi podatek celoštevilska spremenljivka *_counter*, vrednost katere spreminjamo z metodo *_incrementCounter*, ki kliče funkcijo *setState*, ki je ključna, saj ogrodju sporoči, da se je po klicu metode stanje v aplikaciji spremenilo. Tako ogrodje ve, da mora gradnik ponovno izrisati.

5.6.4 Nastavitev projekta za najine potrebe in pomožne funkcije



Slika 73: Struktura mape lib znotraj najine aplikacije (lasten vir)

Ker je pri obsežnejših projektih zelo pomembna organizacija kodne baze, sva najprej definirala osnovne stile aplikacije (mapa *styles*) s pomožnimi razredi Flutter knjižnice Material UI, uporabljene znotraj najine aplikacije. Takšen pristop nama je omogočal dosleden dizajn.

Znotraj abstraktnih razredov (angl. abstract class) sva definirala aplikacijske barve, barvne gradiente, osnovno oblazinjenje (angl. padding), delovanje drsenja in stile pisave, implementacije katerih je olajšala knjižnica storitve Google Fonts *google_fonts*. [64]

```dart
abstract class Containers {
  static final darkBackgroundContainer = BoxDecoration(
    borderRadius: BorderRadius.circular(36),
  );
}
```

```dart
abstract class FontStyles {
  static final h1Title = GoogleFonts.openSans(
    fontSize: 18,
    fontWeight: FontWeight.w700,
  );
}
```

```dart
abstract class FontStyles {
  static final h1Title = GoogleFonts.openSans(
    fontSize: 18,
    fontWeight: FontWeight.w700,
  );
}
```

Slika 74: Definicija stilov znotraj abstraktnih razredov (lasten vir)

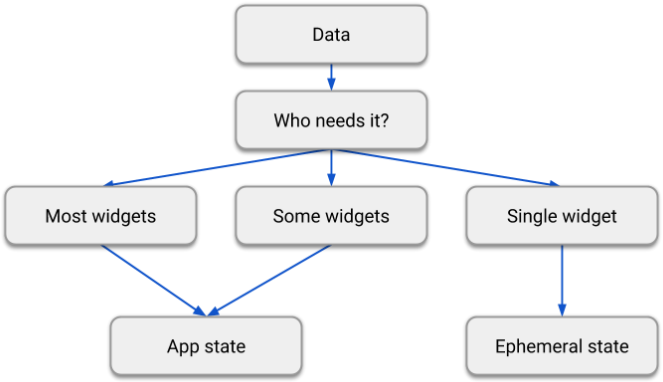
Poleg osnovnih stilov sva definirala tudi osnovne pomožne funkcije, zadolžene za najrazličnejše stvari, kot so na primer pridobivanje dimenzij naprave, preverjanje dostopa do interneta, prikazovanje pojavnega okna itd.

5.6.5 Upravljanje s stanjem

Stanje v aplikaciji lahko glede na mesto uporabe delimo na: [39]

- kratkotrajno (lokalno oz. stanje uporabniškega vmesnika, angl. ephemereal) – stanje, ki pripada samo enemu gradniku. Takšno stanje v ogrodju Flutter prikažemo z gradnikom tipa *StatefulWidget*. Primer so podatki, pridobljeni preko programskega vmesnika.

- aplikacijsko (deljeno, angl. app oz. shared state) – stanje, do katerega želimo dostopati iz več različnih gradnikov in pogosto želimo, da se ohrani med sejami uporabe. Primer so podatki o prijavi in uporabniške preference (npr. barvna tema).



Slika 75: Delitev na aplikacijsko in kratkotrajno stanje [39]

5.6.5.1 Aplikacijsko stanje med povezanimi gradniki (povratni klici)

Najpreprostejša rešitev za upravljanje aplikacijskega stanja med povezanimi gradniki je uvedba t. i. starševskega gradnika (angl. parent widget), v katerega umestimo stanje, vse odvisne gradnike pa navedemo kot otroke tega gradnika. Tako lahko stanje preprosto podamo v njihove konstruktorje. Težava se pojavi, ko otrok želi spremeniti stanje starša, saj metoda *setState* v otroku ne izriše ponovno tudi starševskega gradnika.

Rešitev te težave predstavljajo t. i. povratni klici (angl. callback function) oz. metode, ki jih podamo otrokom in ob klicu spremenijo stanje starševskega gradnika ter posledično ponovno izrišejo vse otroke.

Primer takšnega starševskega gradnika v najini aplikaciji je *ForumScreen*, ki prikazuje vprašanja glede na izbran predmet. Indeks izbranega predmeta shranjujeva v spremenljivki *subjectIndex*, ki jo uporabiva pri poizvedbi s programskim vmesnikom.

Gradniku otroku (angl. child widget) *ForumSubjects*, s katerim uporabnik izbere željen predmet dodava argument *callback* – funkcijo *setSubjectIndex*, s katero lahko gradnik

spremeni stanje (spremenljivko *subjectIndex*) znotraj starševskega gradnika *ForumScreen* in vseh njegovih otrok.

```
class ForumSubjects extends StatefulWidget {
  const ForumSubjects(this.subjects, this.callback, {Key? key})
      : super(key: key);
  final List<FullSubject> subjects;
  final IndexSetter callback;
```

```
void setSubjectIndex(int? indexValue) {
  subjectIndex = indexValue;
  fetchQuestionsAndSubjectsConsecutively();
}
```

Slika 76: Sprememba stanja s funkcijo povratnega klica (lasten vir)

5.6.5.2 Aplikacijsko stanje med večimi nepovezanimi gradniki (ponudniki)

Pri razvoju aplikacije sva večkrat prišla do težave z upravljanjem aplikacijskega stanja med več nepovezanimi gradniki, saj Flutter zanj ne ponuja preproste dopolnjene rešitve. Zgoraj opisana rešitev namreč zahteva številne neželene povezave med gradniki, ki močno otežijo razvoj in vzdrževanje aplikacije. Zaradi tega obstaja veliko zunanjih knjižnic, ki olajšajo reševanje tega problema.

Najbolj priljubljena tovrstna rešitev je knjižnica *Provider* (slov. ponudnik) francoskega razvijalca Remija Rousseleta, ki močno olajša uporabo Flutter mehanizma dedovanja gradnikov (*InheritedWidget*). [65]

Uporaba knjižnice temelji na razredih, imenovanih ponudniki (angl. providers), ki »ponujajo« stanje (v obliki razreda »edinca« – angl. singleton) vsem gradnikom potomcem preko razreda *BuildContext*, ki ima vlogo lokatorja, ki sledi gradnikom v drevesu, hkrati pa jih tudi poišče. Obstaja več tipov ponudnikov, najpreprostejši je *ChangeNotifierProvider*, ki omogoča deljenje primerka razreda *ChangeNotifier*. Ta omogoča posodobitev uporabniškega vmesnika z novimi podatki ob klicu metode *notifyListeners*.

V aplikaciji sva ustvarila dva ponudnika, enega za avtentikacijo (*AuthProvider*) in enega za upravljanje barvnega načina aplikacije (*ThemeProvider*). Slednji menja definirane stile med dvema barvnima načinom aplikacije.

Za dodajanje več ponudnikov naenkrat lahko uporabimo gradnik *MultiProvider*, z uporabo katerega se lahko izognemo nepotrebnemu gnezdenju kode. Kot argument *providers* navedemo seznam ponudnikov znotraj naše aplikacije, kot argument *child* pa gradnik, ki ga želimo prikazati.

Ker želimo stanje deliti z vsemi gradniki potomci aplikacije, ponudnike najpogosteje dodamo kar v izhodiščni metodi aplikacije *runApp*. Tako so vsi gradniki potomci ponudnika.

```
runApp(
  MultiProvider(
    providers: [
      ChangeNotifierProvider(create: (_) => AuthProvider(navigatorKey)),
      ChangeNotifierProvider(create: (_) => ThemeProvider()),
    ],
    child: const MyApp(),
  ), // MultiProvider
);
```

Slika 77: Dodajanje več ponudnikov znotraj metode runApp (lasten vir)

Najlažji način za branje podatkov iz ponudnikov so razširitvene metode *watch*, *read* in *select* razreda *BuildContext*. Metoda *watch* (*context.watch<T>()*) spremlja spremembe v ponudniku in ponovno izriše gradnik ob spremembi, metoda *read* (*context.read<T>()*) vrne primerek ponudnika, vendar ne spremlja sprememb, metoda *select* pa omogoča, da gradnik spremlja spremembe le v delu ponudnika.

```
themeMode: context.watch<ThemeProvider>().themeMode,
```

Slika 78: Uporaba metode watch razreda ThemeProvider (lasten vir)

5.6.6 Povezava med zalednim in čelnim delom

Klice programskega vmesnika opravljava preko pomožnih funkcij za opravljanje različnih HTTP-zahtevkov (*getData, postData, deleteData, patchData, multipartPostData*), ki sva jih definirala v datoteki *api.dart* znotraj mape *api*.

V metodah najprej kličeva zasebno metodo *_getDataForRequest*, s katero ustvariva primerek razreda *ApiRequest*, v katerega shraniva URL-naslov in glave zahtevka, med katere tukaj dodava dostopni žeton.

Nato opraviva HTTP-klic s funkcijami uradne knjižnice *http*, ki vrnejo odziv v obliki razreda *Response*. Najprej preveriva, če je odziv uspešen (ima statusno kodo 200). V primeru neuspešnega zahtevka sproži izjemo tipa *ApiException*. Če odziv vsebuje podatke v telesu (ni prazen), jih deserializirava z metodo *jsonDecode* standardne knjižnice *dart:convert*.

```
static Future<dynamic> getData(String urlSegment, BuildContext context,
    {QueryParams? queryParameters}) async {
  final requestData =
      await _getDataForRequest(urlSegment, queryParameters, context);

  final response = await http.get(
    requestData.url,
    headers: requestData.headers,
  );

  if (response.statusCode == 200) {
    if (response.body.isEmpty) {
      return;
    }

    final body = jsonDecode(response.body);
    return body;
  } else {
    throw ApiException(response);
  }
```

Slika 79: Funkcija za opravljanje GET-klica getData

Končne točke najinega programskega vmesnika sva zbrala v pomožne metode razredov, ki sva jih poimenovala storitve (angl. services).

Spodnja slika predstavlja del storitve *UserService* s klici končnih točk, povezanih z uporabniškimi računi. Najprej sva v konstanti *usersUrl* definirala del URI-naslova, ki je skupen vsem zahtevkom v tej storitvi.

Vsaka metoda sprejme gradnik *BuildContext*, ki ga posreduje razredu *API*, preko katerega ta dostopa do ponudnika za prijavo in posledično dostopnega žetona, in dele URI naslova za poizvedbo. V metodah kličeva pomožne funkcije razreda *API*.

Ker iz zalednega dela podatke pridobiva v formatu JSON, ki nima točno določene strukture, sva jih zapisala znotraj razredov (v mapi *types*), ki predstavljajo zgradbo poizvedenih podatkov.

Vrnjene odzive pretvoriva v primerke entitet s klicem tovarniških funkcij (angl. factory function) *fromJson*, ki kličejo konstruktorje svojih razredov z ustreznimi argumenti, ki jih preberejo iz dobljenega odziva.

```dart
abstract class UserService {
  static const usersUrl = "/users";

  static Future<Tutor> getLoggedInUser(BuildContext context) async {
    final response = await API.getData("$usersUrl/me", context);
    final user = Tutor.fromJson(response);

    return user;
  }
```

Slika 80: Storitev UserService (lasten vir)

```dart
class Subject {
  final int id;
  final String name;

  const Subject({required this.id, required this.name});

  factory Subject.fromJson(JsonData json) =>
      Subject(id: json['id'], name: json['name']);
}
```

Slika 81: Razred Subject s tovarniško funkcijo fromJson (lasten vir)

5.6.6.1 Klici s paginacijo

Kot sva že opisala v predstavitvi zalednega dela, sva se pri pošiljanju seznamov podatkov odločila za implementacijo straničenja oz. paginacije s pomočjo razreda *PageModel*, ki ga vračava ob zahtevkih.

Opisan razred sva implementirala tudi v mobilni aplikaciji. Dodala sva mu poimenovan konstruktor *fromJson*, ki kot argument vzame tudi tovarniško funkcijo *factoryFunction* željenega razreda, s katero pretvoriva JSON elemente dobljenih pod ključem *data*.

```dart
class PageModel<T> {
  final int totalItems;
  final int pageSize;
  final int totalPages;
  final int cursor;
  final List<T> data;

  const PageModel(
      {required this.totalItems,
      required this.pageSize,
      required this.cursor,
      required this.data,
      required this.totalPages});

  factory PageModel.fromJson(
    JsonData json,
    JsonFactoryFunction<T> factoryFunction,
  ) =>
      PageModel(
        totalItems: json["total_items"],
        pageSize: json["page_size"],
        cursor: json["cursor"],
        data: List<T>.from(json["data"].map((x) => factoryFunction(x))),
        totalPages: json["total_pages"],
      ); // PageModel
}
```

Slika 82: Definicija osnovnega razreda za paginacijo v aplikaciji (lasten vir)

Poizvedbe s paginacijo vključujejo argumenta kazalec (*cursor*) in velikost strani (*pageSize*), ki jih dodava na konec URL-naslova kot parametre (argument *queryParameters* – *?cursor=vrednost&pageSize=vrednost* na koncu naslova zahteve). Tako dobljene podatke deserializirava s klicem že opisane funkcije *fromJson* z dobljenim odzivom (spremenljivka *response*) in konstruktorjem entitete (*Message.fromJson*).

```dart
static Future<PageModel<Message>> getMessages(
    int chatId, BuildContext context, int cursor,
    {int pageSize = 15}) async {
  final response = await API.getData("$chatUrl/$chatId/messages", context,
      queryParameters: {
        'cursor': cursor.toString(),
        'pageSize': pageSize.toString()
      });

  final pageModel = PageModel.fromJson(response, Message.fromJson);
  return pageModel;
}
```

Slika 83: Primer poizvedbe s paginacijo (lasten vir)

V gradnikih pošljeva poizvedbo za naslednjo stran, ko uporabnik doseže konec seznama.

To doseževa z razredom *ScrollController*, ki omogoča spremljanje, kako daleč je uporabnik podrsal gradnik *ListView* s klicem »poslušalne« funkcije (angl. listener) ob spremembi lokacije drsenja. V njej preveriva, če je uporabnik dosegel dno seznama s primerjavo odmika z maksimalnim možnim (atribut *maxScrollExtent*). Ko uporabnik doseže konec seznama, ki ima na voljo še naslednjo stran (kazalec ni enak 0), sproži povratni klic (angl. callback), s katerim nastaviva novo vrednost kazalca (identifikator zadnjega elementa) in sproži klic za naslednjo stran.

V metodi za klic naslednje strani najprej pošljeva zahtevo programskemu vmesniku z novo vrednostjo kazalca. Nove podatke dodava na začetek prejšnjega seznama z metodo

insertAll. Nato s spremenljivko *mounted* preveriva, če je gradnik še vpet (prikazan na zaslonu, angl. mounted), saj lahko uporabnik gradnik v času klica zahtevka zapre. Če je gradnik še prikazan, s funkcijo *setState* zahtevava ponoven izris gradnikov z novimi podatki.

```
controller.addListener(() {
  if (controller.position.maxScrollExtent == controller.offset) {
    debugPrint("Dosežen konec seznama");

    if (cursor != 0) {
      widget.fetchNextPageCallback(cursor);
      debugPrint("Kazalec ni 0, kličem naslednjo stran");
    }
  }
});
```

Slika 84: Koda za preverjanje odmika drsenja (lasten vir)

```
Future<void> fetchNextPage() async {
  var messages =
      await ChatService.getMessages(widget.chatId, context, cursor);

  final data = await messagesFuture;
  messages.data.insertAll(0, data.data);

  if (mounted) {
    setState(() {
      messagesFuture = Future.value(messages);
    });
  }
}
```

Slika 85: Metoda za poizvedbo naslednje strani (lasten vir)

5.6.6.2 Objava slik

Pri pošiljanju datotek preko HTTP zahtevkov se uporablja poseben tip vsebine (angl. content type), imenovan *multipart/form-data*, ki sporočilo razdeli na več manjših delov, ki so strežniku poslani v zaporedju. S takim zahtevkom lahko pošljemo tudi več slik hkrati. [40]

Za objavo slik sva definirala metodo *multiPartPost*, s katero opraviva večdelni zahtevek preko razreda *MultiPartRequest*, s katerim lahko dodava slike z razredom *MultipartFile*.

5.6.6.3 Neposredni pogovor

Ker Microsoft ne ponuja uradne SignalR knjižnice za okolje Flutter, sva se odločila za najbolje vzdrževano rešitev, paket *signalr_netcore* avstralskega razvijalca Farshida Sefidgarana. [66]

Vso logiko za povezavo s SignalR končno točko programskega vmesnika sva zbrala v razred *ChatHub*. Kot argument konstruktorja sva navedla razred *BuildContext*, preko katerega dostopava do metode *getToken* ponudnika *authProvider*.

Z omenjeno metodo lahko povezavi s strežnikom preprosto dodava dostopni žeton, saj jo navedeva kot argument *accessTokenFactory* razreda *HttpConectionOptions*.

Povezava s strežnikom poteka preko razreda *HubConnection*, ki ga ustvarimo s pomožnim razredom *HubConnectionBuilder*, ki mu navedemo URL-naslov naše končne točke in možnosti povezave (metoda *withUrl*)

```
class ChatHub {
  static const chatHubUrl = "httpa://${API.apiURL}/chat_hub";

  static const receiveMessageEvent = "ReceiveMessage";
  static const sendMessageEvent = "SendMessage";
  static const deleteMessageEvent = "DeleteMessage";

  late HubConnection hubConnection;

  final logger = Logger("SIGNALR: ");

  ChatHub(BuildContext context) {
    final authProvider = Provider.of<AuthProvider>(context, listen: false);
    final httpConnectionOptions = HttpConnectionOptions(
      accessTokenFactory: () async => (await authProvider.getToken())!,
    );

    hubConnection = HubConnectionBuilder()
        .withUrl(chatHubUrl, options: httpConnectionOptions)
        .withAutomaticReconnect()
        .configureLogging(logger)
        .build();
  }
```

Slika 86: Razred za neposredni pogovor (lasten vir)

Za vzpostavitev povezave kličeva metodo *start*, če povezava ni že vzpostavljena, za prekinitev povezave pa kličeva metodo *stop*, če je povezava vzpostavljena.

Metode na strežniku kličemo preko pomožne metode *invoke*, ki kot argumente zahteva ime metode (v spodnjem primeru konstanta *sendMessageEvent*) in seznam argumentov, ki jih želimo poslati.

Ugotovila sva, da je slabost uporabe SignalR, da ta ne podpira definicij tipov (angl. type safety), zato Flutter knjižnica kot odgovor vrne primerek neobveznega objekta (*Object?*), ki ga je potrebno pretvorit v ustrezen tip (angl. type casting) s ključno besedo *as*. V najinem primeru odgovor pretvoriva v *JsonData* in ga podava konstruktorju želenega razreda (*Message*).

```
Future<Message> sendMessage(int chatId, MessageAdd message) async {
  final messageObject =
      await hubConnection.invoke(sendMessageEvent, args: [chatId, message]);
  final receivedMessage = Message.fromJson(messageObject as JsonData);

  return receivedMessage;
}
```

Slika 87: Pošiljanje sporočil z metodo sendMessage (lasten vir)

Klice metod s strežnika povežemo z definiranimi metodami v gradnikih preko metode *on*, ki kot argumenta zahteva ime metode (v spodnjem primeru konstanta *deleteMessageEvent*) in metodo, ki se bo izvedla.

Tudi tukaj je treba definirati tipe vrnjenih podatkov. Na spodnji sliki tako pred klice metode iz neobveznega seznama objektov (*List<Object?>?*) izluščiva številčni argument identifikatorja izbrisanega sporočila.

```
void registerDeleteEvent(Function(int) method) {
  hubConnection.on(deleteMessageEvent, (params) {
    final messageId = params![0] as int;

    method(messageId);
  });
}
```

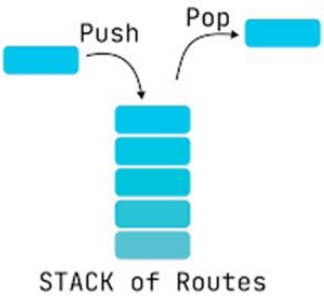
Slika 88: Pridobitev identifikatorja izbrisanega sporočila z metodo registerDeleteEvent (lasten vir)

5.6.7 Usmerjevanje, navigacija in prijava

Ker najino aplikacijo sestavlja več različnih zaslonov (angl. screens), sva morala implementirati način usmerjevanja in navigacijo. Flutter za te namene uporablja gradnik *Navigator*, ki zaslone prikazuje kot sklad (angl. stack).

```
GestureDetector(
  onTap: () {
    Navigator.pop(context);
  },
  child: const Icon(
    MdiIcons.arrowLeft,
    size: 32.0,
    color: CustomColors.mainPurple,
  ),
```

Slika 89: Uporaba osnovne metode pop gradnika Navigator (lasten vir)



Slika: Flutter aplikacije [41]

Do gradnika *Navigator* dostopamo skozi atribut smeri (angl. route) razreda *BuildContext*. Na njem kličemo metode, kot sta na primer *push*, ki na sklad doda zaslon, in *pop*, ki ga iz sklada odstrani.

V primeru na zgornji sliki gre za povratni gumb. Po kliku na gumb se izvede metoda *pop* gradnika *Navigator* iz sklada odstrani trenutni zaslon in nas vrne na tistega, ki se nahaja pod njim.

Gradnik *App*, definiran znotraj datoteke *main.dart*, predstavlja temelje navigacije najine aplikacije.

```dart
class App extends StatelessWidget {
  const App({super.key, required this.
  navigatorKey, required this.home});

  final GlobalKey<NavigatorState> navigatorKey;
  final Widget home;

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      themeMode: context.watch<ThemeProvider>().
      themeMode,
      theme: CustomThemes.lightTheme,
      darkTheme: CustomThemes.darkTheme,
      navigatorKey: navigatorKey,
      home: home,
      debugShowCheckedModeBanner: false,
    );
  }
}
```

Slika 90: Osnovni gradnik najine aplikacije (lasten vir)

Atributi gradnika *MaterialApp, themeMode, theme* in *darkTheme* so namenjeni spreminjanju barvne teme aplikacije. *NavigatorKey* omogoča dostop do navigatorja brez gradnika *BuildContext*. Atribut *home* določa, kateri gradnik se bo prikazal po zagonu aplikacije.

Ker je po zagonu aplikacije več možnih stanj uporabnika (preveriva, če je povezan z internetom in ali je prijavljen), je možnosti za osnovno stran, ki se uporabniku prikaže po zagonu, več.

```dart
if (!isConnected) {
  return App(
    navigatorKey: navigatorKey,
    home: const NoInternetScreen(),
  ); // App
}
```

```dart
@override
void initState() {
  super.initState();

  Connectivity().onConnectivityChanged.listen((event) {
    debugPrint("Sprememba povezave");
    setState(() {
      isConnected = event != ConnectivityResult.none;
    });
  });
}
```

Slika 91: Preverjanje stanja povezave z razredom Connectivity (lasten vir)

Če uporabnik ni povezan z internetom, mu aplikacija prikaže gradnik *NoInternetScreen*, ki uporabnika obvesti, da za uporabo aplikacije potrebuje povezavo z internetom. Za preverjanje stanja internetne povezave sva uporabila zunanjo knjižnico *connectivity_plus* razvijalca *fluttercommunity.dev*, ki omogoča spremljanje stanja povezave prek toka *onConnectivityChanged*. [67]

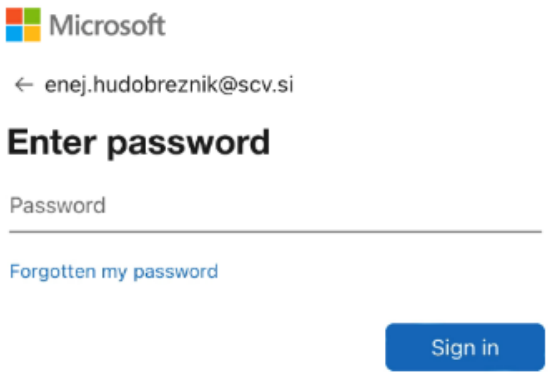
Ali je uporabnik prijavljen, preverimo s pomočjo ponudnika za avtentikacijo *AuthProvider*.

```
if (!context.watch<Auth>().isLoggedIn) {
  return App(
    navigatorKey: navigatorKey,
    home: const LoginScreen(),
  ); // App
}
```

Slika 92: Preverjanje stanja prijave uporabnika s ponudnikom AuthProvider (lasten vir)

V primeru, da nama metoda *getAccessToken* razreda *oauth* v ponudniku za avtentikacijo ne vrne dostopnega žetona (to pomeni, da uporabnik ni prijavljen), se izriše gradnik *LogInScreen*.

Slika 93: Zaslon za prijavo (lasten vir)



Slika 94: Prijava preko Microsofta (lasten vir)

```
onPressed: () => context.read<AuthProvider>().login(),
```

Slika 95: Prijava s ponudnikom AuthProvider (lasten vir)

Po pritisku na gumb za prijavo se izvede metoda *login* razreda *AuthProvider*, ki prek knjižnice za avtentikacijo uporabnika poskrbi za prijavo prek Microsoftovega strežnika s tem, da prikaže ustrezno spletno strani za prijavo, nato pa obdela dobljene podatke (žetone). Nato kličeva metodo *_executeLoginWithApi*, s katero še najin strežnik obvestiva o prijavi.

Po zagonu se morajo gradniki najprej izrisati, aplikacija pa mora od strežnika dobiti ustrezne podatke. Takrat pokaževa gradnik *LoadingIndicator*, ki je eden od mnogih gradnikov za večkratno uporabo oz. komponent (angl. components) znotraj mape *screens/components/general*.

```
58    @override
59    Widget build(BuildContext context) {
60      if (context.watch<Auth>().isLoading) {
61        return App(
62          navigatorKey: navigatorKey,
63          home: const Scaffold(body: LoadingIndicator()),
64        ); // App
65      }
```

Slika 96: Odziv na stanje nalaganja aplikacije (lasten vir)

Ko so izpolnjeni vsi pogoji (vzpostavljena povezava z internetom in uspešna prijava), aplikacija pa je pripravila vse potrebno, se uporabniku prikaže gradnik *ScaffoldNavigation*, ki opravlja vso nadaljnjo navigacijo znotraj najine aplikacije.

```
return App(navigatorKey: navigatorKey, home: const ScaffoldNavigation());
```

Slika 97: Odziv na doseženo ciljno stanje aplikacije (lasten vir)

Osnovala sva jo okoli spodnje navigacijske vrstice (angl. bottom navigation bar), ki omogoča dostop do štirih osnovnih zaslonov: foruma, iskanja tutorjev, pogovora in uporabniškega profila.

Slika 98: Spodnja navigacijska vrstica znotraj najine aplikacije (lasten vir)

```
return SizedBox(
  height: 100,
  child: BottomNavigationBar(
    items: widget.barItems,
    currentIndex: currIndex,
    onTap: (index) {
      setState(() => currIndex = index);
    },
```

Slika 99: Atributi gradnika BottomNavigationBar (lasten vir)

```
final List<BottomNavigationBarItem> barItems = const [
  BottomNavigationBarItem(
    icon: Icon(MdiIcons.homeOutline),
    activeIcon: Icon(MdiIcons.home),
    label: "Forum",
  ), // BottomNavigationBarItem
  BottomNavigationBarItem(
    icon: Icon(MdiIcons.magnify),
    label: "Iskanje",
  ), // BottomNavigationBarItem
  BottomNavigationBarItem(
    icon: Icon(MdiIcons.chatOutline),
    activeIcon: Icon(MdiIcons.chat),
    label: "Chat",
  ), // BottomNavigationBarItem
  BottomNavigationBarItem(
    icon: Icon(MdiIcons.accountOutline),
    activeIcon: Icon(MdiIcons.account),
    label: "Profil",
  ), // BottomNavigationBarItem
];
```

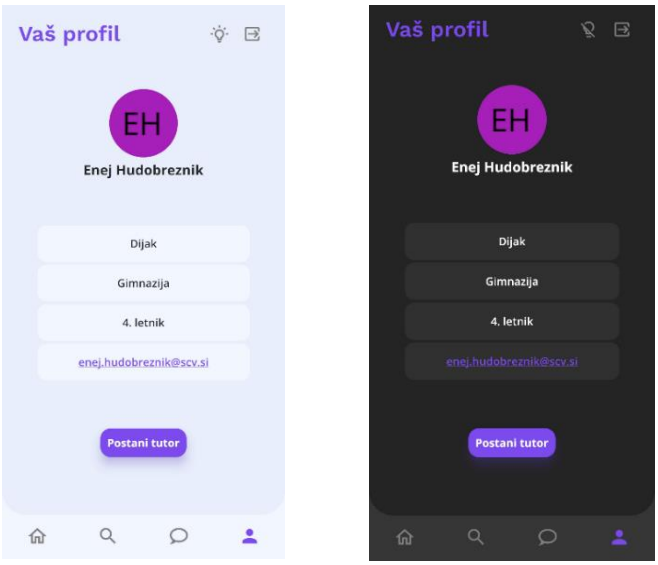
Slika 100: Seznam gradnikov BarItems (lasten vir)

Spodnjo navigacijsko vrstico prikaže gradnik *BottomNavigationBar*, preko katerega nastavljava izbrani zaslon (indeks seznama zaslonov *barItems* v spremenljivki *currIndex*, ki jo nastaviva v metodi *onTap*, ki se izvede ob kliku na element navigacijske vrstice) v seznamu.

Privzeti zaslon je forum, zato sva spremenljivko inicializirala z vrednostjo 0 (prvi element seznama).

5.6.8 Profil uporabnika

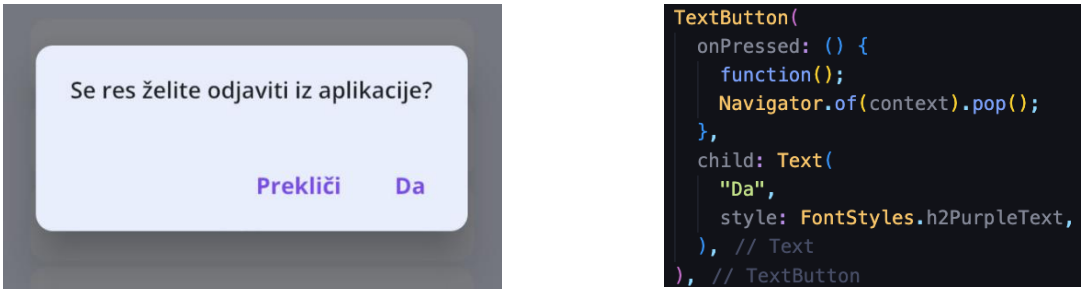
Ko sva nastavila vse potrebno za navigacijo znotraj najine aplikacije, sva se lotila razvoja zaslona *ProfileScreen*, ki nosi podatke o profilu uporabnika. V osnovi izgleda sledeče:



Slika 101: Zaslon s profilom uporabnika ter preklapljanje med svetlo in temno temo (lasten vir)

Uporabniki lahko s pomočjo funkcij ponudnika *ThemeProvider* preklapljajo med dvema temama (med svetlim in temnim načinom), ki sta uporabljeni skozi celotno aplikacijo, na voljo pa imajo tudi izbor sistemske teme, ki je izbrana glede na nastavitev naprave.

Desno od gumba za preklop teme sva postavila gumb za odjavo. Po kliku nanj se prikaže gradnik *AlertDialog*, ki ob kliku na gumb »Da« izvede željeno metodo (v tem primeru metoda *logout* ponudnika za avtentikacijo), nato pa gradnik odstrani iz sklada (metoda *Navigator.pop*).



Slika 102: Odjava iz aplikacije (lasten vir)

V tem delu aplikacije uporabljava razred *User*, ki sva ga razširila še z razredom *Tutor*, ki poleg osnovnih podatkov (identifikator, ime, priimek, vloga, letnik in šola) o uporabniku vključuje še podatke o premetih, ki jih ta poučuje (*List<Subject>*).

Za prikaz podatkov, pridobljenih prek spletnih poizvedb, se uporablja gradnik *FutureBuilder*, ki omogoča spremljanje stanja poizvedbe (razreda *Future*).

V primeru, da je poizvedba vrnila odgovor, se izvede navedeni konstruktor, če pa pride kjerkoli do napake, se uporabniku na zaslonu izriše opozorilo znotraj gradnika *ApiError*. Ker lahko med poizvedbo in odgovorom pride do zakasnitve, sva implementirala tudi indikator nalaganja (*LoadingIndicator*).

To logiko sva uporabila za vse poizvedbe, zato sva jo shranila v gradnik *ApiDataBuilder*.

```
return FutureBuilder<T>(
  future: future,
  builder: (context, snapshot) {
    if (snapshot.hasData) {
      final data = snapshot.data as T;
      return widgetConstructor(data, setter);
    } else if (snapshot.hasError) {
      // final error = snapshot.error!;
      return const ApiError();
    }
    return const LoadingIndicator();
  },
); // FutureBuilder
```

Slika 103: Gradnik ApiDataBuilder za prikaz podatkov, pridobljenih prek spletnih poizvedb

Gradnik *ProfileWidget* sva razdelila na več gradnikov (*NameAndProfilePic*, *ProfileFields* in *BottomProfileButtons*), ki so namenjeni prikazovanju različnih podatkov o uporabniku.

V gradniku *NameAndProfilePic* prikaževa ime, priimek in profilno sliko uporabnika.

Profilno sliko uporabnika, ki jo v zalednem delu pridobiva prek vmesnika Microsoft Graph, (5.3.1.4) prikaževa s pomočjo gradnika *NetworkImage*, ki prikaže omrežno sliko,

```
NetworkImage("$imageUrlPrefix/profile_pics/${user.id}"),
```

Slika 104: Izris profilne slike z gradnikom NetworkImage (lasten vir)

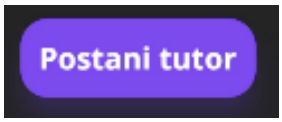
Gradnik *ProfileFields* je namenjen prikazovanju ostalih podatkov o uporabniku (vloga, šola, letnik, šolski e-poštni naslov in predmete, ki jih poučuje, če je uporabnik tutor). Zgrajen je iz več gradnikov *InfoAreaComponent*, ki kot argumente sprejmejo različne atribute razreda *Tutor*.

```dart
@override
Widget build(BuildContext context) {
  return Column(
    mainAxisAlignment: MainAxisAlignment.spaceEvenly,
    children: [
      NameAndProfilePic(user: user),
      ProfileFields(user),
      BottomProfileButtons(user, refreshUser)
    ],
  ); // Column
}
```

Slika 105: Gradnik ProfileWidget (lasten vir)

```dart
InfoAreaComponent(
  text: user.role.name.capitalize(),
), // InfoAreaComponent
InfoAreaComponent(
  text: user.school.name.capitalize(),
), // InfoAreaComponent
InfoAreaComponent(
  text: '${user.grade}. letnik',
), // InfoAreaComponent
InfoAreaComponent(
    text: user.email, url: generateMailToUrl(user.email)),
if (user.role == UserRole.tutor)
  InfoAreaComponent(
    text: user.subjects.map((s) => s.name).join(", "),
  ), // InfoAreaComponent
```

Slika 106: Gradnik ProfileFields (lasten vir)

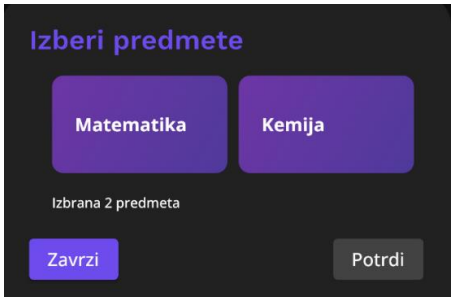



```dart
if (user.role == UserRole.dijak) {
  return PurpleButton(
    onPressed: () {
      showModalBottomSheet(
        context: context,
        builder: (context) => SubjectsSelectModalBottomSheet(
          refreshUser: refreshUser,
        ), // SubjectsSelectModalBottomSheet
      );
    },
    text: "Postani tutor",
  ); // PurpleButton
}
```

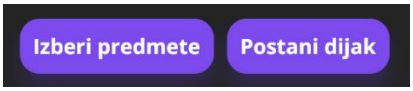
Slika 107: Povišanje v vlogo tutorja (lasten vir)

V osnovi je na začetku vsak uporabnik po vlogi dijak. Če želi postati tutor, mora klikniti na gumb »Postani tutor« v gradniku *BottomProfileButtons*, ki prikaže gradnik, ta pa omogoča izbiro predmetov, ki jih želi poučevati. Po vsaki potrditvi spremembo sporočiva programskemu vmesniku prek pomožnih metod razreda *UserService*.

Dijaku tutorju sta pod polji s podatki na voljo dva gumba, ki omogočata spremembo izbranih predmetov (prek enakega vmesnika, ki ima že izbrane predmete, ki jih že poučuje) in spremembo vloge nazaj v dijaka.



Slika 108: Izbira predmetov, ki jih tutor želi poučevati (lasten vir)



Slika 109: Gumb za izbiro predmetov in spreminjanje vloge (lasten vir)

Če uporabnik znotraj najine aplikacije klikne na profilno sliko drugih uporabnikov, se mu prikaže gradnik *OtherUserProfile*, ki za razliko od *UserProfile* nima dodatne

funkcionalnosti spremembe vloge in izbire predmetov, omogoča pa odpiranje pogovora z izbrano osebo, kar omogoča metoda *createOrGetChat* storitve *ChatService*, ki od programskega vmesnika vrne identifikator pogovora med uporabnikoma. Po uspešni poizvedbi se izvede še pomožna metoda *pushChatScreen*, ki uporabnika preusmeri na ustrezen zaslon s pogovorom.

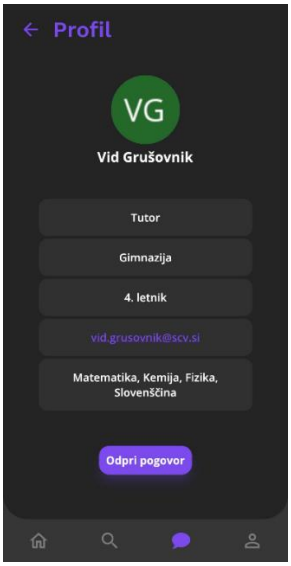
5.6.9 Iskanje tutorjev

Eden izmed ciljev najine aplikacije je zagotovo tudi olajšati iskanje vrstniških tutorjev.

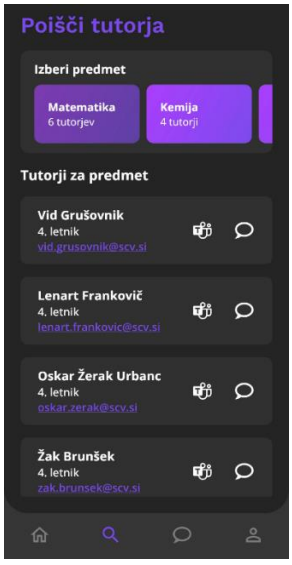
Vloga uporabnika je prikazana na njegovem profilu, ki je dostopen s klikom njegove profilne slike. Učinkovitejše iskanje sva implementirala znotraj posebnega zaslona, namenjenega točno tej funkciji.

Izhodišče tega zaslona je gradnik *TutorScreen*. Uporabniki imajo možnost filtriranja med predmeti, pri katerih potrebujejo pomoč. To funkcionalnost omogočava preko drsne vrstice znotraj gradnika *SubjectPicker*, ki prikaže seznam (gradnik *ListView*) predmetov *subjectsFuture*, pridobljenih prek programskega vmesnika, in izbiro prek povratnega klica *setSubjectIndex* sporoči nazaj starševskemu gradniku (5.6.5.1).

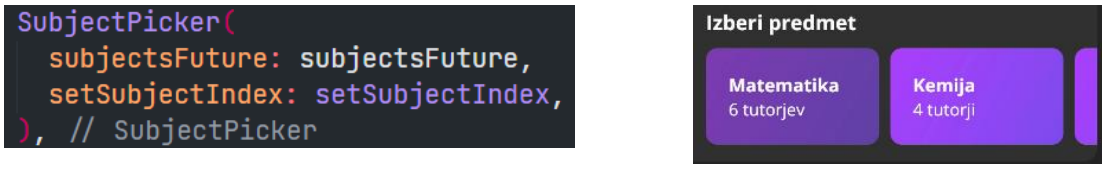
Po izbiri določenega predmeta posodobiva gradnik *TutorList*, ki nosi podatke o tutorjih v obliki seznama gradnikov *TutorWidget*, ki pokažejo osnovne podatke o vsakem tutorju (ime, priimek, letnik in šolski e-poštni naslov). Ob kliku nanj, ki ga zaznava z gradnikom *GestureDetector*, prikaževa profil tutorja (metoda *Navigator.push*).



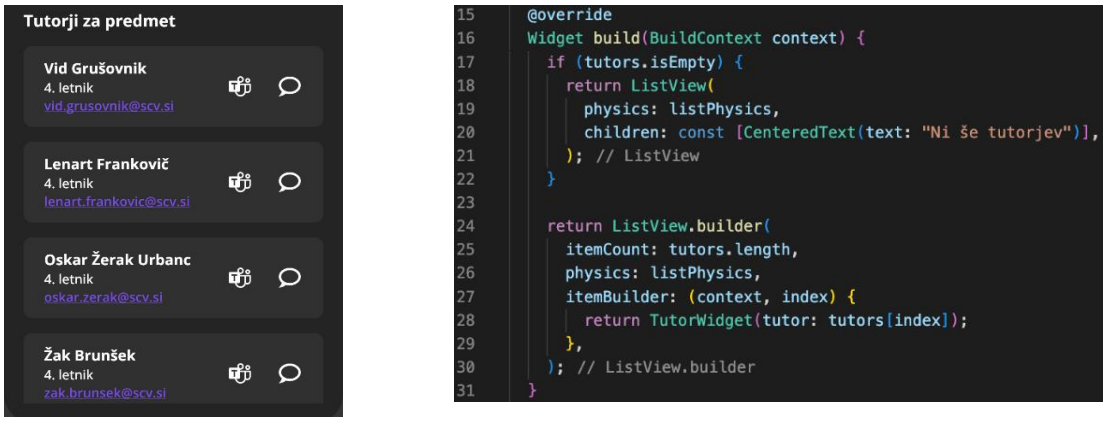
Slika 110: Profil tutorja (lasten vir)



Slika 111: Zaslon, namenjen iskanju tutorja (lasten vir)



Slika 112: Gradnik SubjectPicker (lasten vir)



Slika 113: Drsni seznam tutorjev znotraj gradnika ListView (lasten vir)

Uporabnik lahko s tutorjem stopi v kontakt tudi preko šolskega e-poštnega naslova kar znotraj aplikacije. Po kliku nanj (gradnik *GestureDetector*) se s klicem metode *launchUrl* uradne knjižnice *url_launcher* [68], ki odpre URL-naslov v obliki *mailto://<e-pošta uporabnika>*, uporabniku odpre privzeta aplikacija namenjena e-pošti, kjer je tutor že naslovljen kot prejemnik.

Desno od osnovnih podatkov se nahajata dve ikoni, ki predstavljata še ostali metodi kontaktiranja tutorjev znotraj najine aplikacije.

Po kliku na prvo ikono se uporabniku odpre pogovor s tutorjem znotraj aplikacije Microsoft Teams. To storiva s klicem URL-naslova *https://teams.microsoft.com/l/chat/0/0?users=<e-poštni naslov uporabnika>* prek že omenjene metode *launchUrl*.

Klik na drugo ikono uporabniku odpre neposredni pogovor s tutorjem znotraj najine aplikacije, kot sva že opisala v opisu profila neprijavljenega uporabnika.

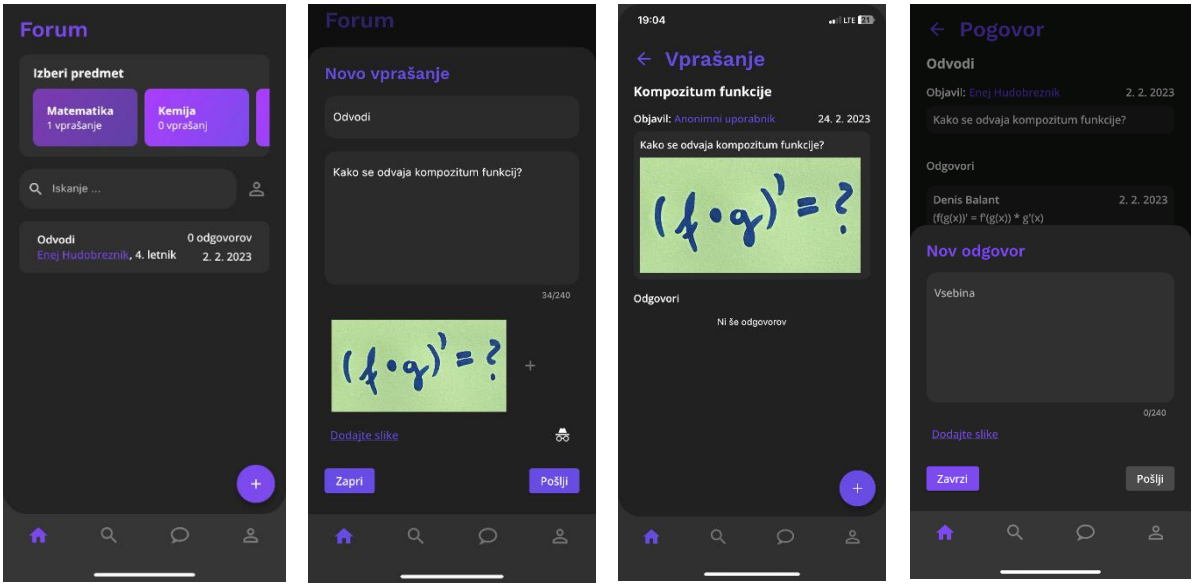
5.6.10 Forum

Znotraj foruma lahko vsi uporabniki med seboj komunicirajo tako, da postavljajo vprašanja in nanje odgovarjajo.

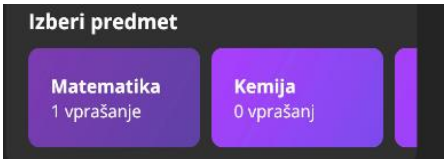
Forum je razdeljen na dva zaslona: zaslon, namenjen postavljanju vprašanj in brskanju med njimi, (prvi dve sliki) in zaslon za odgovarjanje nanje (zadnji dve sliki).

Izhodišče foruma predstavlja gradnik *ForumScreen*. Da lahko uporabniki filtrirajo med vprašanji za različne predmete, sva v zgornjem delu gradnika dodala drsno vrstico *SubjectPicker* (opisana v poglavju 5.6.10).

Z namenom, da bi uporabnikom nudila kar se da dober način brskanja med vprašanji, sva implementirala tudi iskalno vrstico znotraj komponente *SearchBar* (s povratnim klicem *setSearchTerm*) in gumb, ki prikaže le uporabnikova vprašanja za izbran predmet (s spremenljivko logičnega tipa *showOnlyMyPosts*). Po spremembi omenjenih gradnikov seznam vprašanj osveži s ponovnim klicem programskega vmesnika prek metode *fetchQuestions*.



Slika 116: Funkcionalnosti foruma (lasten vir)



Slika 117: Gradnik SubjectPicker (lasten vir)

Slika 118: Iskalna vrstica znotraj gradnika SearchBar (lasten vir)

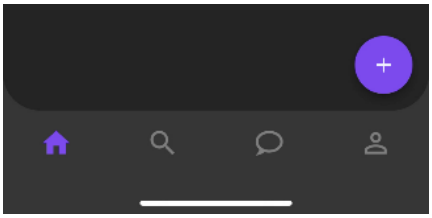
```
void setSearchTerm(String value) {
  searchTerm = value;
  fetchQuestions();
}
```

Slika 114: Metoda povratnega klica setSearchTerm (lasten vir)

```
onPressed: () {
  setState(() {
    showOnlyMyPosts = !showOnlyMyPosts;
  });

  fetchQuestions();
},
```

Slika 115: Osveževanje prikazanih vprašanj po vnosu v iskalno vrstico (lasten vir)

Po izbiri predmeta znotraj drsne vrstice se uporabnikom v spodnjem desnem kotu izriše gumb (gradnik *FloatingActionButton*), na katerega lahko kliknejo in preko gradnika *ForumModalBottomSheet* vnesejo svoje vprašanje.



```
floatingActionButton: subjectIndex != null
    ? AddButton(
        bottomSheetWidget: ForumModalBottomSheet(
          selectedSubjectId: subjectIndex!,
          refreshQuestions: fetchQuestionsAndSubjectsInSuccession,
          outsideContext: context,
        ), // ForumModalBottomSheet
      ) // AddButton
    : null,
```

Slika 119: Gumb za dodajanje vprašanj (lasten vir)

Znotraj prvega besedilnega polja vpiše uporabnik naslov vprašanja, v drugega pa njegovo vsebino, dolžino katere sva omejila na 240 znakov z atributom *maxLength* gradnika *TextField*.

Uporabniki lahko vprašanje postavijo tudi anonimno (namesto imena in priimka se pojavi napis *Anonimni uporabnik*) s klikom na spodnjo desno ikono.

Vprašanja lahko vsebujejo tudi slike. Po kliku na polje »Dodajte slike« se pod besedilnimi polji izriše vrstica (gradnik *ImageUploadBar*) za dodajanje slik, ki s povratno metodo *onImagesChanged* obvesti starševski gradnik o izbranih slikah.

Po kliku na plus-ikono se odpre uporabnikova galerija, kjer lahko izbere ustrezno slikovno gradivo s pomočjo metode *pickMultiImage* razreda *ImagePicker* uradne knjižnice *image_picker*. [69] Slike pred pošiljanjem skrčiva z metodo *compressAndGetFile* knjižnice *flutter_image_compress* [70] skupine kitajskih razvijalcev Flutter Candies.

```
ImageUploadBar(
  onImagesChanged: onImagesChanged,
), // ImageUploadBar
```

```
final result = await FlutterImageCompress
.compressAndGetFile(
  imagePath,
  compressedImageName,
  quality: 30,
);
```

Slika 120: Objavljanje slik preko gradnika ImageUploadBar (lasten vir)

Ko uporabnik izpolni vsa potrebna polja, lahko vprašanje zavrže (gradnik se zapre prek metode *pop*) ali objavi (pošljeva ga programskemu vmesniku prek razreda *ForumService*). Po novi objavi seznam vprašanj osveži s klicem povratne metode.

Vprašanje prikaževa z gradnikom *QuestionWidget*, ki prikaže oblikovan gradnik *ListTile*. Ob kliku nanj uporabnika preusmeriva na zaslon *QuestionScreen*, ki uporabnikom omogoča pogled posamičnega vprašanja in odgovarjanje nanj.

V zgornjem delu gradnika izriševa naslov vprašanja, takoj pod njim pa se nahajata datum objave in ime avtorja. Ob kliku (metoda *onTap* gradnika *GestureDetector*) na njegovo ime uporabnika preusmeriva na profil avtorja (gradnik *OtherUserProfile*).

Vsebino vprašanja prikaževa z gradnikom *QuestionContent*, ki kot argumente zahteva vsebino (*content*) in identifikatorje objavljenih slik (*imageIds*), ki jih prikaževa v seznamu z gradnikom *ListView.separated*, ki omogoča dodajanje oblazinjenja med elementi drsnega seznama.

```
QuestionContent(
  content: widget.question.content,
  imageIds: widget.question.imageIds,
), // QuestionContent
```

Slika 121: Gradnik QuestionContent (lasten vir)

```
child: ListView.separated(
  scrollDirection: Axis.horizontal,
  physics: listPhysics,
  itemBuilder: (context, index) =>
      GeneratedImagePreview(photoId: imageIds[index]),
  separatorBuilder: (context, index) => const Padding(
    padding: EdgeInsets.only(right: 8),
  ), // Padding
  itemCount: imageIds.length,
), // ListView.separated
```

Slika 122: Drsni seznam z prikazanimi slikami (lasten vir)

V omenjenem seznamu so gradniki *GeneratedImagePreview*, osnovani na gradniku *NetworkImage*, ki kot argument zahtevajo URL-naslove prikazanih slik, ki jih generirava s pomožno metodo *generateImageUrl*, ki identifikator slike doda naslovu storitve Imgur.

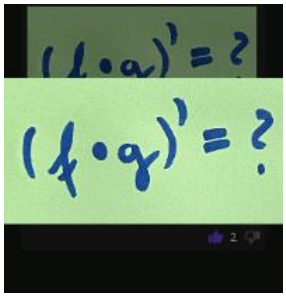
```
final imageUrl = generateImageUrl(photoId);

return ImagePreview(image: Image.network(imageUrl));
```

```
String generateImageUrl(String id) => "https://i.imgur.com/$id.jpeg";
```

Slika 123: Generiranje URL-naslovov slik (lasten vir)

Da bi bile slike znotraj najine aplikacije čimbolj uporabne, sva morala implementirati sistem interakcije z njimi. Za povečevanje slik sva uporabila gradnik *InteractiveViewer*, ki ga odpreva ob kliku na sliko. Kot argumente sprejme maksimalen faktor povečave (*maxScale*), obnašanje elementa, ko zasede prevelik prostor znotraj uporabniškega vmesnika (*clipBehavior*) in otroka (*child),* ki predstavlja izrisano sliko.

```
child: InteractiveViewer(
  clipBehavior: Clip.none,
  maxScale: 5.0,
  child: image,
), // InteractiveViewer
```



Slika 124: Gradnik InteractiveViewer (lasten vir)

Pod vsebino vprašanja se nahaja seznam gradnikov *ReplyWidget*, ki prikazujejo odgovore.

Znotraj tega gradnika prikaževa ime uporabnika, ki je na vprašanje odgovoril, datum odgovora in vsebino, lahko vsebuje tudi slike, ki jih prikaževa na enak način kot v gradniku

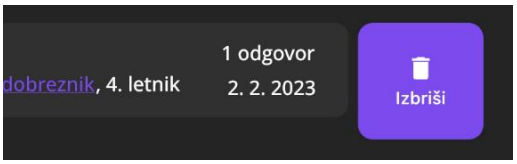
QuestionContent. Tudi tukaj lahko uporabniki na sliko kliknejo, da jo povečajo ter njeno velikost po želji spreminjajo.

Da bi bila uporabniška izkušnja znotraj foruma čim bogatejša, sva znotraj gradnika *ReplyUpvoteCounter* vgradila možnost glasovanja o ustreznosti odgovora v obliki »všečkov« in »nevšečkov«.

Uporabniku med ikonama (gradnika *IconButton*), ki sprožita ustrezne klice programskega vmesnika (metoda *sendVote*), prikaževa razliko med »všečki« in nevšečki«, ki služi kot ocena verodostojnosti odgovora.

Uporabniki znotraj tega zaslona odgovore dodajajo na enak način kot vprašanja v gradniku *ForumScreen*. Zelo podoben je tudi gradnik *ReplyModalBottomSheet*, ki se izriše po kliku na gumb v spodnjem desnem kotu, le da ta nima polja za vpis naslova.

Uporabnikom omogočava tudi brisanje njihovih lastnih vprašanj in odgovorov. To zmožnost imajo tudi administratorji, s to razliko, da lahko izbrišejo katero koli vprašanje oz. odgovor.



Slika 125: Brisanje lastnih objav na forumu (lasten vir)

Gradnik *QuestionWidget* je osnovan na gradnika *DeleteSlidable*, ki uporabniku omogoča, da dostopa do dodatnih možnosti z drsenjem na stran. Omenjen gradnik, ki sva ga uporabila tudi znotraj medsebojnega pogovora, je del paketa *flutter_slidable* francoskega razvijalca Romaina Rastela. [71] Omogočen je le, če je identifikator prijavljenega uporabnika enak identifikatorju uporabnika, ki je vprašanje ustvaril oz. če je uporabnik administrator (metoda *isAdminOrAuthor*). Po kliku na gumb “Izbriši” programskemu vmesniku pošljeva zahtevo za izbris (metoda *deleteQuestion*) in nato osveževa seznam vprašanj.

```
future<void> deleteQuestion(BuildContext context) async {
  await ForumService.deleteQuestion(question.id, context);
  await refreshCallback();
}
```

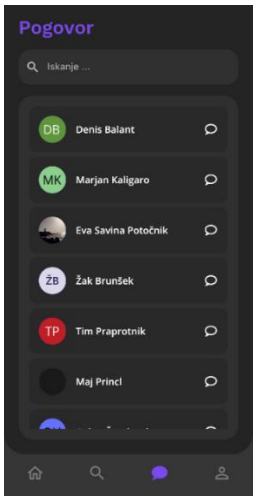
```
return DeleteSlidable(
  enabled: isAdminOrAuthor(question.userCreated.id, context),
  onPressed: (context) => deleteQuestion(context),
```

Slika 126: Možnost izbrisa katerega koli vprašanja, če je uporabnik administrator in gradnik DeleteSlidable (lasten vir)

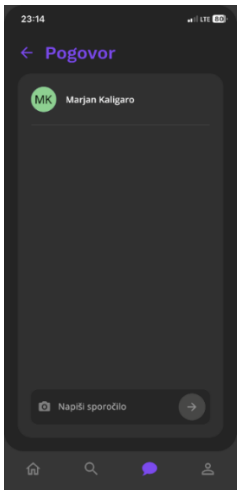
5.6.11 Neposredni pogovor

Poleg foruma imajo uporabniki na voljo tudi funkcijo neposrednega pogovora.

Pogovor je v osnovi razdeljen na dva zaslona. Preko prvega lahko dostopamo do pogovorov z uporabniki, preko drugega pa lahko beremo in pišemo sporočila.



Slika 127: Brskanje med pogovori (lasten vir)



Slika 128: Neposreden pogovor z uporabnikom (lasten vir)

Izhodišče pogovora predstavlja gradnik *ChatUsersList*, ki prikaže seznam gradnikov *ChatUser* (argumenta identifikator pogovora in uporabnik) z gradnikom *ListView.separated*.

Ob kliku na uporabnika se odpre gradnik *ChatMessagesScreen*, ki prikaže sporočila (gradnik *MessageWidget*) v seznamu z gradnikom *ChatDataListView*.

Najprej se iz strežnika prenese zgodovina sporočil s klicem metode *ChatService.getMessages*, istočasno pa se odpre tudi povezava s SignalR strežnikom (*chatHub.startConnection*) prek razreda *ChatHub*, kar omogoči sprotnost pogovora.

V *initState* metodi tako registrirava funkcijo, ki se izvede ob dobljenem sporočilu (*handleReceivedMessage*), in funkcijo, ki se izvede, ko sogovornik sporočilo (*handleDeletedMessage*) izbriše.

```
@override
void initState() {
  messagesFuture = ChatService.getMessages(widget.chatId, context, cursor);
  chatHub = ChatHub(context);
  chatHub.registerReceiveEvent(handleReceivedMessage);
  chatHub.registerDeleteEvent(handleDeletedMessage);
  chatHub.startConnection();

  super.initState();
}
```

Slika 129: Registracija metod, klicanih s strežnika (lasten vir)

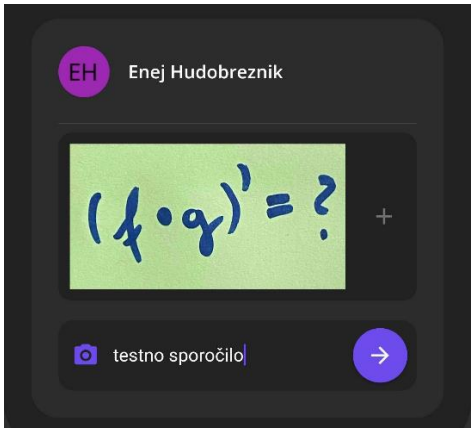
Uporabnik sporočilo pošlje prek gradnika *BottomChatBar*, ki ima dodan že opisan gradnik *ImageUploadBar*.

Ob kliku na gradnik *IconButton* s puščico se ustvari primerek razreda *MessageAdd*, ki ga podava kot argumenti povratnemu klicu *sendMessageCallback*, ki novo sporočilo doda v

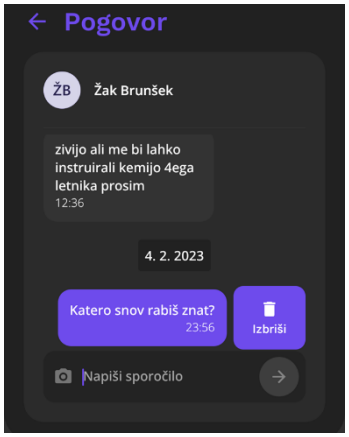
seznam in kliče metodo *ChatHub.sendMessage*, hkrati pa izprazniva vsebino tekstovnega polja in skrijeva vrstico za dodajanje slik.

```
Future<void> sendMessage() async {
  final messageAdd =
      MessageAdd(content: textController.text, imagePaths: imagePaths);
  textController.text = "";
  setState(() {
    showImageBar = false;
  });

  widget.sendMessageCallback(messageAdd);
}
```

Slika 130: Pošiljanje sporočila (lasten vir)



Slika 131: Dodajanje slik (lasten vir)



Slika 132: Brisanje lastnih sporočil (lasten vir)

Uporabniki lahko svoja sporočila izbrišejo na enak način kot vprašanja na forumu z že opisanim gradnikom *DeleteSlidable*, ki tokrat kliče metodo *_chatHub.deleteMessage*.

```
@override
Widget build(BuildContext context) {
  return DeleteSlidable(
    enabled: isSentByMe,
    onPressed: (context) => _chatHub.deleteMessage(chatId, message.id),
    child: Align(
```

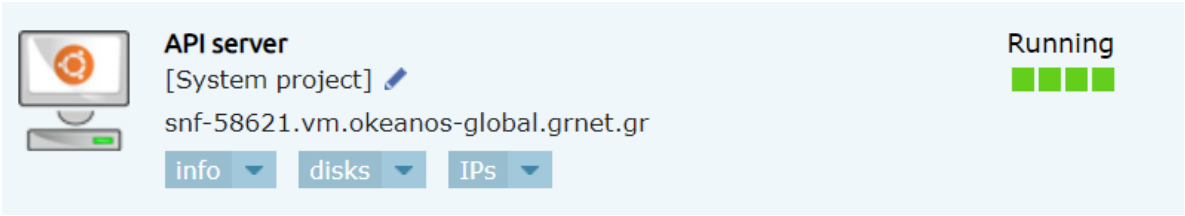
Slika 133: Gradnik DeleteSlidable (lasten vir)

6 OBJAVA APLIKACIJE IN GOSTOVANJE

6.1 Gostovanje zalednega dela

Aplikacija lahko deluje le, če je povezana s programskim vmesnikom, ki se izvaja na strežniku, do katerega lahko prosto dostopajo vsi uporabniki. Najpogosteje aplikacij ne izvajamo na lastnih strežnikih, ampak za ta namen uporabljamo storitve, ki jih ponujajo podjetja z ustrezno infrastrukturo. Take storitve imenujemo gostovanje. [42]

Sama sva se odločila za gostovanje na t. i. navideznih zasebnih strežnikih (angl. Virtual Private Server – VPS), ki so nam, dijakom, brezplačno dostopni preko grške oblačne storitve Okeanos, do katere lahko dostopamo z Arnes AAI računom.



Slika 134: Preko Okeanosa nastavljen strežnik za gostovanje zalednega dela (lasten vir)

Pri takem načinu gostovanja dobimo dostop do popolnoma lastnega dela fizičnega strežnika, na katerem se izvaja virtualna naprava (angl. virtual machine) z operacijskim sistemom po izbiri uporabnika.

Sama sva se zaradi predhodnih izkušenj in dobre uradne dokumentacije odločila za GNU/Linux distribucijo Ubuntu Server, ki ga upravljamo preko ukazne vrstice s protokolom SSH (Secure Shell Protocol). Pri konfiguraciji sva sledila uradnemu vodiču na uradni spletni platformi Microsoft Learn.

Čeprav ima okolje ASP.NET Core vgrajen spletni strežnik Kestrel, Microsoft priporoča uporabo bolje optimiziranega dodatnega strežnika kot vzvratni namestnik (angl. reverse proxy), preko katerega preusmerimo podatke do aplikacije. Sama sva sledila priporočilom in izbrala priljubljen odprtokodni strežnik *Nginx*.

Omenjen strežnik sva nastavila preko konfiguracijske datoteke */etc/nginx/sites-available/default*, v kateri sva navedla tudi domeno, povezano z naslovom strežnika, kot ime strežnika (*server_name*). Spodaj prikazana konfiguracija ves zunanji promet iz vrat 80 skupaj z glavami zahtevkov preusmeri na naslov *http://127.0.0.1:5000* (lokalna vrata 5000). Da aplikacija lahko sprejme posredovane glave zahtevkov (*XForwardedFor* za IP naslov odjemalca in *XForwardedProto* za protokol odjemalca), je bilo potrebno dodati vmesno programje *UseForwardedHeaders*. [43]

```
app.UseForwardedHeaders(new ForwardedHeadersOptions
{
    ForwardedHeaders = ForwardedHeaders.XForwardedFor | ForwardedHeaders.XForwardedProto
});
```

```
server {
    listen        80 default_server;
    listen        [::]:80 default_server;

    server_name   api.opentutor.si;

    location / {
        proxy_pass         http://127.0.0.1:5000;
        proxy_http_version 1.1;
        proxy_set_header   Upgrade $http_upgrade;
        proxy_set_header   Connection keep-alive;
        proxy_set_header   Host $host;
        proxy_cache_bypass $http_upgrade;
        proxy_set_header   X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header   X-Forwarded-Proto $scheme;
    }

    location /chat_hub {
        proxy_pass http://localhost:5000;

        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection $http_connection;

        proxy_buffering off;
        proxy_read_timeout 100s;

        proxy_set_header Host $host;
    }
}
```

Slika 135: Konfiguracija Nginx strežnika (lasten vir)

Za delovanje storitve SignalR je potrebno dodati še dodatno konfiguracijo za končne točke naših vozlišč (*location /chat_hub*), s katero omogočimo uporabo dodatnih protokolov (WebSockets, Server Sent Events in Long Polling).

Na zalednem strežniku sva nato preko uradnega repozitorija namestila .NET Runtime, ki je potreben za izvajanje C# programov.

Projekt zalednega vmesnika sva »zapakirala« v aplikacijo z ukazom *dotnet publish –configuration Release* in dobljen paket prenesla na strežnik s pomočjo protokola SFTP (SSH File Transfer Protocol).

Za avtomatski zagon in nadzor nad izvajajočo aplikacijo sva jo registrirala kot storitev, ki je pod nadzorom sistemskega upravitelja storitev *systemd*. Za registracijo sva morala dodati t. i. datoteko storitve (angl. service file) v formatu INI v mapo */etc/systemd/system* in jo zagnati z ukazom *sudo systemctl enable opentutor.service*.

Uporaba strežnika Nginx močno olajša tudi brezplačno dodajanje certifikata za TLS (Transport Layer Security) šifriranje preko odprtokodnega orodja *certbot*, ki omogoča povezavo z neprofitnim potrdilnim organom (angl. certificate authority – CA) Let's Encrypt. Orodje omogoča pridobivanje in namestitev certifikata skupaj s konfiguracijo strežnika z le enim ukazom *sudo certbot --nginx*. Pridobljen certifikat je predpogoj za uporabo protokola HTTPS, ki je za razliko od HTTP šifriran in posledično varnejši.

6.2 Objava mobilne aplikacije

Ko sva zaključila z razvojem najine aplikacije, sva jo objavila na platformah Apple App Store in Google Play. Postopek je za obe precej podoben. Morala sva oblikovati ikono in jo dodati Flutter projektu, narediti posnetke zaslonov na telefonih različnih dimenzij,

navesti splošne pogoje itd. Verzijo aplikacije sva določala preko spremenljivke *version* v Flutter datoteki *pubspec.yaml*, kar precej olajša konfiguracijo pred posodabljanjem aplikacije.

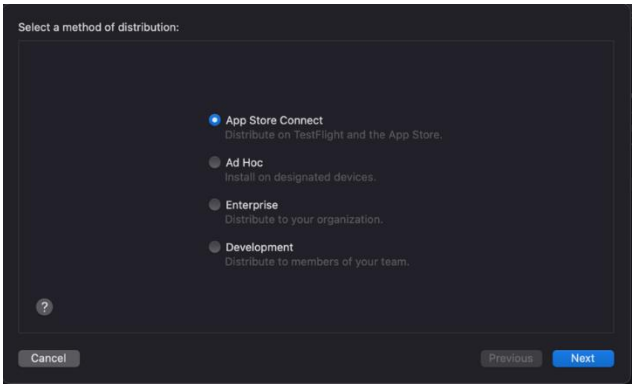
6.2.1 Apple

Aplikacije na App Store objavljamo in posodabljamo preko Apple platforme App Store Connect. Ker je za to potrebno orodje Xcode, je to mogoče le preko Applovih računalnikov.

Pred navedenimi koraki sva morala registrirati edinstven identifikator skupka (angl. bundle identifier), ki ima navadno obliko obratnega zapisa imena domene (t. i. reverse domain name notation) in imena aplikacije, ki sta ločena s pikami (v najinem primeru si.scv.opentutor).

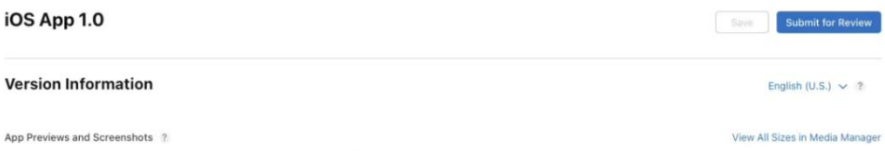
Nato sva znotraj App Store Connect ustvarila novo aplikacijo, ki sva jo morala tudi podpisati s posebnim certifikatom, ki zagotavlja pristnost aplikacije. Nekatere nastavitve so zahtevale tudi poseg v mapo *ios*.

Po uspešni nastavitvi sva aplikacijo združila v skupek z ukazom *flutter build ios*, tega pa sva z Xcode zgradila v t. i. arhiv (angl. app archive). Ko se proces izgradnje zaključi, arhiv kar znotraj integriranega razvojnega okolja objaviva z metodo App Store Connect (uporabnik mora biti vpisan z ustreznim Apple Developer računom).



Slika 136: Objava arhiva na App Store preko metode distribucije App Store Connect (lasten vir)

Naložen skupek na spletnem portalu še na kratko opišemo, na koncu pa aplikacijo oddamo v pregled s klikom na gumb “submit for review”.



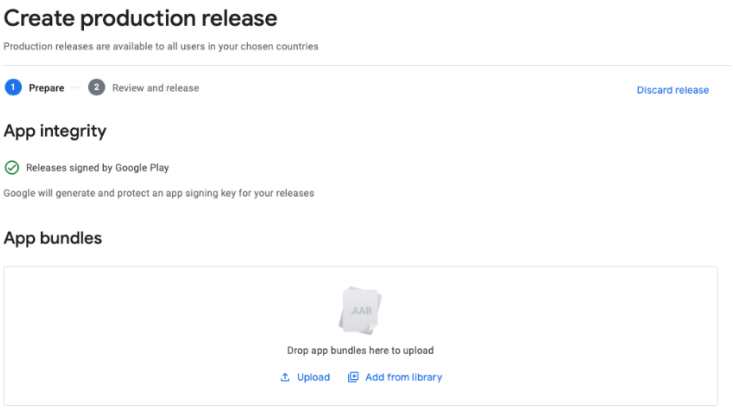
Slika 137: Oddaja aplikacije v pregled (lasten vir)

6.2.2 Google

Proces objave na platformi Google Play poteka preko Google Play Console in je zelo podoben kot pri Applu, le da je enostavnejši. Tudi tu sva morala registrirati identifikator

skupka, ustvariti novo aplikacijo in jo podpisati s posebnim ključem ter nastaviti vse potrebno znotraj mape *android*.

Na koncu sva aplikacijski skupek zgradila z ukazom *flutter build appbundle* in izhodno *.aab* datoteko oddala v pregled na portalu Google Play Console.

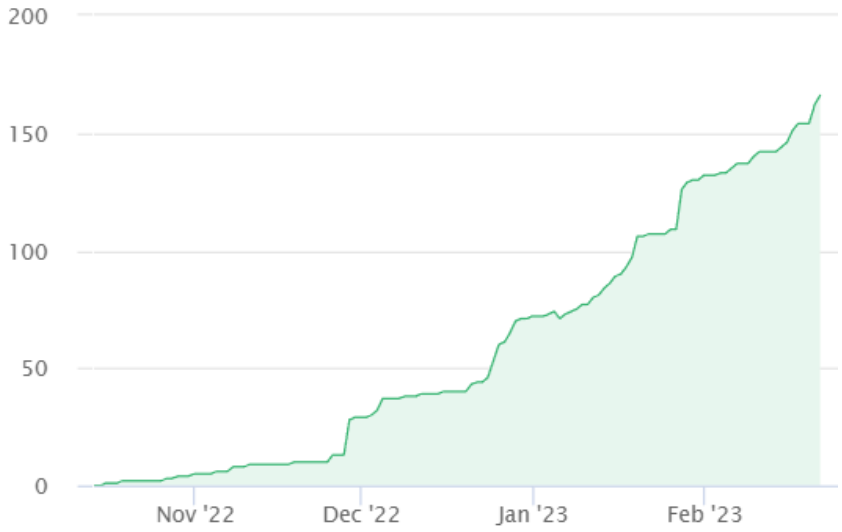


Slika 138: Dodajanje aplikacijskega skupka v pregled (lasten vir)

7 TESTIRANJE

Aplikacijo sva na obeh platformah objavila 28. novembra 2022. Idejo sva predstavila ravnateljici, ki je dijake naše gimnazije pozvala k njeni uporabi. Do sredine februarja je rešitev dosegla 50 dokaj aktivnih uporabnikov.

Mnogi izmed njih so nama preko ocen posredovali povratne informacije, s katerimi sva odpravila mnoge hrošče in dodelala obstoječe funkcije. Takšen primer je bil med drugim tudi predlog najinih mentorjev, naj dodava možnost anonimne objave vprašanj (s skritim imenom avtorja). Funkcija je med uporabniki hitro zaživela in opazno povečala aktivnost dijakov na forumu.



Slika 139: Število OneSignal naročnin (lasten vir)

Uporabnost aplikacije sva v začetku februarja testirala s pomočjo intervjuja z dvema najbolj aktivnima tutorjema najine platforme: Oskarjem Žerakom Urbancem, ki poučuje matematiko, in Rokom Hudournikom, ki poučuje matematiko, kemijo in fiziko. Intervjuja sta priložena v prilogi, odgovore pa sva analizirala v razpravi.

8 RAZPRAVA

Iskanje učne pomoči ostaja večni problem mladih. V sodobnem času se pojavljajo mnoge digitalne rešitve, ki se s številnimi funkcijami trudijo olajšati iskanje različnih informacij in znanj.

Po njihovi temeljiti analizi sva ugotovila, da na slovenskem trgu ne obstaja nobena platforma, ki bi podpirala medvrstniško pomoč, ki je po najinih izkušnjah pogosto najpreprostejši in najugodnejši način pridobivanja znanja, saj vrstniki pogosto ponudijo drug pogled na učno snov. Ta prednost je še večja med učenci z istim profesorjem.

Tako sva se odločila za izgradnjo mobilne aplikacije s funkcijami, ki sva jih pogrešala pri obstoječih platformah. Zasnovala sva diskusijski forum, ki vsem uporabnikom omogoča postavljanje vprašanj in odgovarjanje nanje. Dijakom, ki želijo pomagati pri določen predmetu, sva dodala možnost, da se registrirajo kot tutorji, ki jih lahko iskalci učne pomoči poiščejo preko specializiranega vmesnika. Za lažjo vzpostavitev prvega stika sva vgradila tudi neposredni pogovor med uporabniki.

Nadaljnja komunikacija lahko poteka preko povezanega omrežja Microsoft Teams ali elektronske pošte, ki je dostopna prek uporabnikovega profila. Za takšno integracijo in pridobivanje informacij o uporabnikih sva uporabila šolske Arnes račune, ki jih lahko uporabimo tudi za prijavo prek Microsofta s storitvijo Azure Active Directory, na kateri sva osnovala avtentikacijo v najini aplikaciji.

Pri razvoju sva se odločila za uporabo sodobnega Googlovega ogrodja Flutter, ki omogoča večplatformni razvoj z eno kodno bazo v programskem jeziku Dart. Tega prej še nisva poznala, vendar je njegova sintaksa dokaj podobna ostalim objektno-orientiranim jezikom (npr. Javi), zato je bil prehod dokaj preprost. Večje preglavice nama je povzročal način dela z ogrodjem Flutter, ki je kljub uporabi komponent (gradnikov) precej drugačen od ostalih platform za izgradnjo uporabniških vmesnikov, saj je osnovan na delu z velikim številom objektov.

Aplikacijo sva preko podatkovnega vmesnika, zgrajenega s C# ogrodjem ASP.NET Core, povezala z relacijsko podatkovno bazo PostgreSQL. Z omenjenim ogrodjem sva že imela izkušnje, zato nama delo z njim ni predstavljalo večje težave, a je bila to najina prva prava izkušnja z delom z relacijsko podatkovno bazo. Zaradi kompleksnih poizvedb sva si delo z njo močno olajšala z Microsoftovim orodjem za objektno-relacijsko mapiranje Entity Framework Core, ki omogoča delo s podatkovno bazo s poizvedbeno sintakso LINQ jezika C#, ki sva jo že poznala.

Proces razvoja aplikacije tolikšnega obsega ni bil enostaven, saj se mnogokrat problemi skrivajo v zelo specifičnih podrobnostih, ki jih je težko raziskati na internetu. Uporabniški vmesnik sva morala prilagoditi za delovanje na telefonih različnih dimenzij in poskrbeti, da so ključne funkcije implementirane čim bolj učinkovito. Tako sva hkrati poskrbela za privlačen videz in odzivno delovanje aplikacije.

Največ težav so nama predstavljale funkcionalnosti, ki jih je težko testirati in ugotoviti vzrok napak. Takšna primera sta potisna sporočila, saj nas sistem na napačno konfiguracijo ne opozori, in implementacija neposrednega pogovora, saj je bilo za povezavo prek

protokola WebSockets potrebno strežnik Nginx v vlogi vzvratnega namestnika dodatno konfigurirati, kar nama je vzelo kar nekaj poskusov.

8.1 Interpretacija intervjuja s tutorjema

Med testiranjem aplikacije sva izvedla dva intervjuja s tutorjema, ki sta opisala svojo uporabniško izkušnjo in ocenila celotno aplikacijo, hkrati pa sta podala tudi predloge za izboljšave. Tako sta nama ponudila vpogled v primernost novonastalega orodja.

Iz obeh izvedenih intervjujev je razvidno, da je najino orodje potrebno, pri čemer intervjuvanca poudarjata zlasti prvi korak navezave stika med dijaki in komunikacijo nasploh. Poudarjata tudi strah dijakov pri komuniciranju, kar je značilno zlasti za nižje letnike. Ravno ti namreč najpogosteje potrebujejo pomoč pri razumevanju učne snovi. Po oceni obeh intervjuvancih najina aplikacija oba problema učinkovito rešuje, hkrati pa omenjata, da najina rešitev služi kot odlična podpora programu tutorstva, ki do sedaj še ni zares zaživelo.

Funkcije najine aplikacije intervjuvana tutorja ocenjujeta kot zadostne oz. primerne, pri čemer sta pohvalila zlasti forum, ki po njunem mnenju ponuja najboljšo možnost za pasivno prejemanje znanja. Menita, da je okolje, ki sva ga oblikovala, zelo lepo zasnovano in dobro služi svojemu namenu. Oba sta pohvalila tudi njeno enostavnost uporabe, kar bi utegnila biti velika prednost, saj njun odgovor nakazuje na primernost uporabe tudi v osnovnošolske namene. Uporabniška izkušnja najinega orodja je po njunem mnenju zelo dobra in intuitivna, aplikacija pa ima velik potencial tudi v prihodnje.

Zelo spodbudna je tudi njuna ugotovitev glede odzivnosti, saj naj bi aplikacija delovala odzivno in brez večjih motenj. Rok je pri uporabi opozoril na manjšo težavo glede foruma, ki ni deloval pravilno, ki sva jo s povratnimi informacijami hitro odpravila.

Eno izmed zastavljenih vprašanj se je nanašalo tudi na izboljšave. Nasploh sta bila oba tutorja z aplikacijo zadovoljna, vendar sta kljub temu podala tudi par izboljšav, ki bi njuno uporabniško izkušnjo še izboljšale. Oskar kot možno izboljšavo navaja obvestilo ob zastavljenem vprašanje na forumu iz predmetov, ki jih sam poučuje, in možnost daljših odgovorov, Rok pa izpostavlja možnost spletnih učilnic ali pa videoklicev.

Vsi podani predlogi izboljšav so jasno dobrodošli, zato se bova kar se da potrudila, da jih pri modeliranju aplikacije v prihodnje tudi upoštevava. Ob izvedbi intervjuja sva nekoliko podvomila v mnenje o videoklicih, saj najino orodje v osnovi ni namenjeno učni pomoči na daljavo, vendar zgolj kot opora za navezavo prvega stika med dijaki.

8.2 Pregled hipotez

1. **Obstoječe rešitve za iskanje učne pomoči ne spodbujajo medvrstniške pomoči.**

   Po pregledu obstoječih rešitev sva prvo hipotezo potrdila, saj nisva našla rešitve, specializirane za iskanje medvrstniške učne pomoči. Večinoma gre za plačljive inštruktorske platforme ali pa za nabor nalog z rešitvami. Edina podobna rešitev je nemoderirani forum Dijaški.net, ki že vrsto let ni aktiven, rešitve za iskanje vrstniškega tutorja pa na slovenskem trgu nisva našla.

Sama sva pred leti iskala pomoč preko obstoječih socialnih omrežjih, ki v veliki meri niso namenjena izobraževanju, ampak zabavi. Posledično vključujejo številne moteče dejavnike, ki uporabnike pogosto odvrnejo od učenja. Prav tako močno otežijo iskanje ustrezne pomoči in verodostojnih podatkov, ker temu preprosto niso namenjena.

2. **Relacijska podatkovna baza je za izdelavo diskusijskega foruma primernejša od nerelacijske.**

   Po uporabi obeh tipov podatkovnih baz sva drugo hipotezo potrdila, saj sva naletela na težave s predstavljanjem različnih relacij med entitetami foruma pri nerelacijskih podatkovnih bazah. Relacijske podatkovne baze so osnovane okoli sistematičnega modeliranja relacij med povezanimi podatki, kar orodja, kot je objektno-relacijsko mapiranje, še dodatno olajšajo.

3. **Razvoj mobilne aplikacije z ogrodjem Flutter predstavlja učinkovito alternativo razvoju več ločenih rešitev brez vpliva na uporabniško izkušnjo.**

   To hipotezo sva potrdila le delno, saj so aplikacije, izdelane z ogrodjem Flutter, po učinkovitosti in zmogljivosti močno primerljive z lastnimi rešitvami. Razvijalce privlači predvsem preprostost večplatformnega razvoja z eno kodno bazo, ki ga ogrodje ponuja.

   Določene funkcionalnosti uporabniških vmesnikov ne delujejo na čisto enak način, zato je potrebno aplikacijo pred izdajo testirati na obeh platformah. Prav tako različni zunanji paketi pogosto zahtevajo dodatno konfiguracijo znotraj ločenih projektov, kar zahteva poznavanje njihove strukture. Takšna neustrezna konfiguracija vodi do hroščev v času izvajanja, ki jih je težko natančno določiti.

   Pogosto sva zasledila tudi, da številne uradne knjižnice nimajo podpore za omenjeno platformo (npr. prijava prek Microsofta, SignalR). Njihovo funkcionalnost nadomestijo odprtokodne knjižnice, ki so pogosto zelo slabo vzdrževane in zastarele.

4. **Najina aplikacija je okrepila in uspešno digitalno podprla program tutorstva na Gimnaziji ŠC Velenje.**

   Po izvedbi intervjujev sva to hipotezo potrdila, saj sta oba tutorja najino aplikacijo označila kot koristno. Zaradi prijazne uporabniške izkušnje in privlačnega uporabniškega vmesnika olajša proces komunikacije med dijaki ter tako po njunem mnenju občutno olajša prvi stik med tutorjem in iskalcem učne pomoči. Zelo sta bila navdušena tudi nad forumom, ki služi kot vsestranska oblika medvrstniške komunikacije in pomoči.

   Aplikacija lahko uspešno deluje le, če je ustrezno moderirana, saj nekateri uporabniki najdejo veselje v zbadanju drugih in širjenju dezinformacij, zato vodenje takšne aplikacije ni preprosto. Rešitev sva tako zasnovala z moderacijo v mislih, saj omogočava hitro brisanje neprimernih vsebin.

8.3 Možne izboljšave

Da bi še dodatno izboljšala uporabniško izkušnjo znotraj najine aplikacije, bi morala razviti učinkovitejši sistem moderacije. Ročno brisanje sporočil je zamudno, hkrati pa bi ob večjem številu uporabnikov filtriranje med vsebinami postalo praktično nemogoče brez velikega števila moderatorjev. Tega problema bi se lahko lotila na več načinov.

Najbolj učinkovito ga lahko rešila z integracijo umetne inteligence, ki bi bila sposobna zaznati neprimerne izraze in nezaželjene objave. Hkrati bi lahko implementirala tudi funkcijo avtomatskega brisanja neustreznih in slabo ocenjenih odgovorov na vprašanja. Vgraditi nameravava še možnost prijave odgovorov in vprašanj, kar bi v proces moderacije vključilo tudi ostale uporabnike in tako platformo v celoti prilagodilo njihovim potrebam.

Uporabnike, ki redno objavljajo neustrezne vsebine pa bi lahko tudi blokirala in jih tako onemogočila nadaljnjo uporabo aplikacije. Ta funkcija bi bila zelo učinkovita, saj za prijavo uporabljava šolske račune, katerih dijaki na naredijo sami. Vsak račun je tako povezan točno z enim dijakom, ustvarjanje novih računov pa ni mogoče.

Trenutno znotraj najine aplikacije zaradi manjšega števila tutorjev še ni funkcije njihovega ocenjevanja. To želiva vključiti v bližnji prihodnosti, saj bodo tako uporabniki lažje našli primernega tutorja, ki jim bo sposoben nuditi kvalitetno znanje.

Aplikacijo za zdaj testirava le na gimnaziji, po dodatnih izboljšavah pa jo nameravava razširiti na cel šolski center. Naslednji korak bi predstavljala uporaba aplikacije po drugih slovenskih šolah, a bi za to morala spremeniti proces prijave. Vse izobraževalne institucije v Sloveniji uporabljajo Arnesove AAI račune, s katerimi bi lahko pokrila celotno Slovenijo. O njihovi uporabi sva razmišljala že na začetku razvoja, a sva se zaradi pomanjkanja ustrezne dokumentacije omejila le na Šolski center Velenje.

Da bi pridobila čim več uporabnikov, bi morala razviti dober promocijski načrt, ki bi dijake spodbudil k uporabi najine aplikacije. Za pomoč bi lahko prosila ravnatelje in profesorje na srednjih šolah ali pa jo promovirala preko družbenih omrežij z znanimi slovenskimi vplivneži in oglasi.

Dostopnost in uporabnost najine platforme bi lahko dodatno povečala še z izdelavo spletne strani, ki bi vsebovala vse funkcionalnosti mobilne aplikacije. Njen razvoj bi bil dokaj hiter, saj bi morala na novo zgraditi le čelni del, s čimer že imava izkušnje.

9 POVZETEK

Iskanje učne pomoči je že od nekdaj zelo aktualna tema med dijaki, ki potrebujejo pomoč pri razlagi in razumevanju učne snovi. Mnogi dijaki ne vedo, pri kom lahko pomoč poiščejo ali pa si tutorjev ne upajo kontaktirati. Tako medvrstniška pomoč vse prepogosto ostaja neizkoriščena.

V raziskovalni nalogi sva raziskala obstoječe rešitve. Ker nisva našla nobene ustrezne, sva se lotila reševanja opisanega problema z izdelavo nove mobilne aplikacije, ki sva jo poimenovala OpenTutor. Njen glavni namen je olajšanje vzpostavitve prvega stika med iskalci dodatne razlage in uporabniki, ki so jo pripravljeni nuditi, ter spodbuditi dijake k iskanju pomoči, ko jo potrebujejo.

Prosto dostopen forum omogoča preprosto postavljanje novih vprašanj ter hitro iskanje med njimi, najbolj pa sva se osredotočila na oblikovanje vmesnika za iskanje tutorjev in neposredni pogovor med uporabniki. Celoten postopek razvoja od izbire tehnologij do implementacije posameznih funkcij skupaj z analizirano dokumentacijo temeljito opiševa, na koncu izdelek testirava med najinimi vrstniki in na podlagi povratnih informacij oceniva njegov potencial.

10 ZAKLJUČEK

Z raziskovalnim delom sva želela ponuditi nov način iskanja medvrstniške učne pomoči kot alternativo za ta namen neprimernim omrežjem ter tako digitalno podpreti sistem tutorstva na najini šoli, na koncu pa tudi oceniti njegovo ustreznost in potrebnost.

Po primerjavi obstoječih rešitev sva določila tri osnovne funkcije, ki sva jih pogrešala pri obstoječih rešitvah: diskusijski forum za postavljanje vprašanj in odgovarjanje nanja, vmesnik za iskanje tutorjev in neposredni pogovor med uporabniki. Omenjene funkcije sva povezala v lastno mobilno aplikacijo, ki sva jo tudi objavila na trgovine z aplikacijami.

Pridobljene povratne informacije kažejo na potencial tovrstne rešitve in prednosti takšnega pristopa k reševanju omenjenega problema. Intervjuvana tutorja sta bila nad najino aplikacijo zelo navdušena, hkrati pa sta izpostavila njeno uporabnost in pozitivne vidike tekoče uporabniške izkušnje. S potencialom najine platforme se strinjajo tudi številni dijaki, ki v vedno večjem številu sodelujejo v diskusijskemu forumu, zato meniva, da lahko dodelano aplikacijo razširiva tudi zunaj najinega šolskega centra.

Med izdelovanjem naloge sva se soočila z mnogimi izzivi, ki sva se jih naučila učinkovito reševati s pomočjo spletnih virov. Naučila sva se uporabe programske dokumentacije za izdelavo specifičnih rešitev, spoznala sva številne nove tehnologije, z uporabo katerih sva se soočila prvič. Najpomembneje pa sva se podučila v skupinskem delu, ki je nepogrešljiv del razvoja večjih razvojnih projektov.

11 ZAHVALA

Iskreno se zahvaljujeva mentorjema prof. Islamu Mušiću in mag. Ivanu Jovanu za vso podporo, pomoč in usmerjanje med izdelavo raziskovalnega dela.

Posebna zahvala gre prof. Polonci Glojek za lektoriranje besedila naloge. Zahvala sošolcu Vidu Grušovniku za pomoč pri izpeljavi intervjujev in Marjanu Kaligaru za strokovno svetovanje.

Zahvaljujeva se tudi obema intervjuvancema in vsem uporabnikom najine aplikacije.

12 VIRI IN LITERATURA

[1] „O projektu | Astra.si | Več kot 2500 matematičnih razlag“, *Astra.si*. https://astra.si/o-projektu/ (pridobljeno 15. november 2022).
[2] „Kdo smo? | Razturi na maturi“. https://www.pripravenamaturo.si/Kdo-smo (pridobljeno 15. november 2022).
[3] „O nas“, *e-Matura*. https://www.ematura.si/o-nas/ (pridobljeno 15. november 2022).
[4] „O nas“, *e-Inštruktor*. https://www.einstruktor.si/o-nas/ (pridobljeno 15. november 2022).
[5] „Izobraževalni portal Dijaški.net upihnil 18 svečk“, *rtvslo.si*. https://www.rtvslo.si/oglasno-sporocilo/izobrazevalni-portal-dijaski-net-upihnil-18-sveck/452421 (pridobljeno 15. november 2022).
[6] G. O. inštrukcije, „Pri GO inštrukcijah vam pomagamo najti inštuktorja v vaši bližini.“, *GO inštrukcije*. https://www.go-tel.si/instrukcije/ (pridobljeno 15. november 2022).
[7] „What is a mobile app (mobile application)? – TechTarget Definition“, *WhatIs.com*. https://www.techtarget.com/whatis/definition/mobile-app (pridobljeno 19. november 2022).
[8] „What is a Front End (In a Website) - Definition & Development“. https://airfocus.com/glossary/what-is-a-front-end/ (pridobljeno 19. november 2022).
[9] „Native vs cross-platform mobile app development“, *CircleCI*, 24. avgust 2022. https://circleci.com/blog/native-vs-cross-platform-mobile-dev/ (pridobljeno 19. november 2022).
[10] „8 Most Popular Mobile App Development Frameworks in 2022“. https://www.purchasely.com/blog/mobile-app-frameworks (pridobljeno 19. november 2022).
[11] „Flutter vs React Native: A Comparison“, *BrowserStack*. https://browserstack.wpengine.com/guide/flutter-vs-react-native/ (pridobljeno 19. november 2022).
[12] „FAQ“. https://docs.flutter.dev/resources/faq (pridobljeno 20. november 2022).
[13] „Opredelitev termina Podakovna baza“. http://colos.fri.uni-lj.si/eri/RACUNALNISTVO/PODATKOVNE_BAZE/opredelitev_termina_podakovna_baza.html (pridobljeno 3. december 2022).
[14] „PostgreSQL: About“. https://www.postgresql.org/about/ (pridobljeno 3. december 2022).
[15] „Programski vmesnik“, *iPROM*. https://iprom.si/slovar/api-programski-vmesnik/ (pridobljeno 6. december 2022).
[16] „ASP Tutorial“. https://www.w3schools.com/asp/default.asp (pridobljeno 6. december 2022).
[17] „Git“. https://git-scm.com/ (pridobljeno 8. december 2022).
[18] „Kaj je GIT in kako začeti?“, *Žabja reganja*, 11. maj 2016. https://blog.zabec.net/kaj-je-git-in-kako-zaceti/ (pridobljeno 8. december 2022).
[19] „Git: Reference Sheet“, *NeSI Support*, 23. januar 2022. https://support.nesi.org.nz/hc/en-gb/articles/360001508515-Git-Reference-Sheet (pridobljeno 8. december 2022).
[20] „What Is GitHub? A Beginner’s Introduction to GitHub“, *Kinsta®*, 13. oktober 2022. https://kinsta.com/knowledgebase/what-is-github/ (pridobljeno 8. december 2022).
[21] „Understanding GitHub Actions“, *GitHub Docs*. https://docs.github.com/en/actions/learn-github-actions/understanding-github-actions (pridobljeno 4. januar 2023).
[22] „Objektno-relacijsko mapiranje“, *Wikipedija, prosta enciklopedija*. 28. maj 2016. Pridobljeno: 15. december 2022. [Na spletu]. Dostopno na: https://sl.wikipedia.org/w/index.php?title=Objektno-relacijsko_mapiranje&oldid=4650594

[23] ajcvickers, „Overview of Entity Framework Core - EF Core“, 25. maj 2021. https://learn.microsoft.com/en-us/ef/core/ (pridobljeno 15. december 2022).
[24] ajcvickers, „Database Providers - EF Core“, 19. januar 2022. https://learn.microsoft.com/en-us/ef/core/providers/ (pridobljeno 15. december 2023).
[25] „Model–view–controller“, *Wikipedia*. 10. februar 2022. Pridobljeno: 18. december 2022. [Na spletu]. Dostopno na: https://en.wikipedia.org/w/index.php?title=Model%E2%80%93view%E2%80%93controller
[26] wadepickett, „Tutorial: Create a web API with ASP.NET Core“, 12. marec 2022. https://learn.microsoft.com/en-us/aspnet/core/tutorials/first-web-api (pridobljeno 18. december 2022).
[27] Rick-Anderson, „ASP.NET Core Middleware“, 5. januar 2023. https://learn.microsoft.com/en-us/aspnet/core/fundamentals/middleware/ (pridobljeno 9. januar 2023).
[28] Rick-Anderson, „Dependency injection in ASP.NET Core“, 26. januar 2022. https://learn.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection (pridobljeno 14. januar 2023).
[29] Rick-Anderson, „Create Data Transfer Objects (DTOs)“, 10. maj 2022. https://learn.microsoft.com/en-us/aspnet/web-api/overview/data/using-web-api-with-entity-framework/part-5 (pridobljeno 16. januar 2023).
[30] Rick-Anderson, „Create web APIs with ASP.NET Core“, 18. november 2022. https://learn.microsoft.com/en-us/aspnet/core/web-api/ (pridobljeno 18. januar 2023).
[31] roji, „Pagination - EF Core“, 26. julij 2022. https://learn.microsoft.com/en-us/ef/core/querying/pagination (pridobljeno 19. januar 2023).
[32] M. C, „Is offset pagination dead? Why cursor pagination is taking over“, *Medium*, 2. avgust 2020. https://uxdesign.cc/why-facebook-says-cursor-pagination-is-the-greatest-d6b98d86b6c0 (pridobljeno 19. januar 2023).
[33] bradygaster, „Use hubs in ASP.NET Core SignalR“, 18. december 2022. https://learn.microsoft.com/en-us/aspnet/core/signalr/hubs (pridobljeno 26. januar 2023).
[34] „Imgur API“, *Imgur API*. https://apidocs.imgur.com (pridobljeno 2. januar 2023).
[35] barclayn, „What is Azure Active Directory? - Microsoft Entra“, 10. januar 2023. https://learn.microsoft.com/en-us/azure/active-directory/fundamentals/active-directory-whatis (pridobljeno 1. februar 2023).
[36] „What is a Push Notification?“ https://www.twilio.com/docs/glossary/what-is-push-notification?utm_source=docs&utm_medium=social&utm_campaign=guides_tags (pridobljeno 3. februar 2023).
[37] „Push Notifications and In-App Messaging with OneSignal and Expo | by George Deglin | Exposition“. https://blog.expo.dev/push-notifications-and-in-app-messaging-with-onesignal-and-expo-3abf77b7f288 (pridobljeno 3. februar 2023).
[38] M. Malewicz, „Claymorphism in User Interfaces | Hype4Academy“, *Hype4 Academy*. https://hype4.academy/articles/design/claymorphism-in-user-interfaces (pridobljeno 18. december 2022).
[39] „Differentiate between ephemeral state and app state“. https://docs.flutter.dev/development/data-and-backend/state-mgmt/ephemeral-vs-app (pridobljeno 20. december 2022).
[40] „Forms in HTML documents“. https://www.w3.org/TR/html401/interact/forms.html#h-17.13.4 (pridobljeno 10. januar 2023).
[41] „Flutter Navigator 2.0 for Authentication and Bootstrapping — Part 2: User Interaction | by Cagatay Ulusoy | Level Up Coding“. https://levelup.gitconnected.com/flutter-navigator-2-

0-for-authentication-and-bootstrapping-part-2-user-interaction-5dc043e7e44a (pridobljeno 20. december 2022).
[42] „What is Hosting? - Definition from Techopedia“, *Techopedia.com*. http://www.techopedia.com/definition/29023/web-hosting (pridobljeno 23. december 2022).
[43] Rick-Anderson, „Host ASP.NET Core on Linux with Nginx“, 12. december 2022. https://learn.microsoft.com/en-us/aspnet/core/host-and-deploy/linux-nginx (pridobljeno 23. december 2022).
[44] „Representational state transfer“, *Wikipedia*. 5. marec 2022. Pridobljeno: 30. december 2022. Dostopno na: https://en.wikipedia.org/w/index.php?title=Representational_state_transfer&oldid=1140131803
[45] „openapi_generator | Dart Package“, *Dart packages*. https://pub.dev/packages/openapi_generator (pridobljeno 10. januar 2023).
[46] „GraphQL | A query language for your API“. https://graphql.org/ (pridobljeno 2. januar 2023).
[47] „Preslikava razmerja“. http://colos.fri.uni-lj.si/eri/RACUNALNISTVO/PODAT_PLAT_RAZ_PO/preslikava_razmerja.html (pridobljeno 15. januar 2023).
[48] „Real-Time Communication Techniques“, *Telerik Blogs*, 10. december 2019. https://www.telerik.com/blogs/real-time-communication-techniques (pridobljeno 28. december 2023).
[49] „Store images in the database | Learn Data Modeling“. https://www.datanamic.com/support/storeimagesinthedatabase.html (pridobljeno 2. januar 2023).
[50] jswymer, „Using OAuth to Authenticate Business Central Web Services (OData and SOAP) - Business Central“, 30. junij 2022. https://learn.microsoft.com/en-us/dynamics365/business-central/dev-itpro/webservices/authenticate-web-services-using-oauth (pridobljeno 20. januar 2023).
[51] „Get Started with JSON Web Tokens“, *Auth0*. https://auth0.com/learn/json-web-tokens (pridobljeno 20. januar 2023).
[52] auth0.com, „JWT.IO“. http://jwt.io/ (pridobljeno 20. januar 2023).
[53] „What is OAuth 2.0 and what does it do for you?“, *Auth0*. https://auth0.com/intro-to-iam/what-is-oauth-2 (pridobljeno 20. januar 2023).
[54] OwenRichards1, „Microsoft identity platform and OAuth 2.0 authorization code flow - Microsoft Entra“, 5. januar 2023. https://learn.microsoft.com/en-us/azure/active-directory/develop/v2-oauth2-auth-code-flow (pridobljeno 20. januar 2023).
[55] „O nas :: OpenProf.com“. https://si.openprof.com/o-nas/ (pridobljeno 7. februar 2023).
[56] „Npgsql.EntityFrameworkCore.PostgreSQL 7.0.3“. https://nuget.org/packages/Npgsql.EntityFrameworkCore.PostgreSQL/ (pridobljeno 7. november 2022).
[57] „Microsoft.AspNet.SignalR.Core 2.4.3“. https://nuget.org/packages/Microsoft.AspNet.SignalR.Core/ (pridobljeno 9. november 2022).
[58] „aad_oauth | Flutter Package“, *Dart packages*. https://pub.dev/packages/aad_oauth (pridobljeno 15. november 2022).
[59] „Microsoft.Identity.Web.MicrosoftGraph 2.0.8-preview“. https://nuget.org/packages/Microsoft.Identity.Web.MicrosoftGraph/ (pridobljeno 18. november 2022).

[60] „webview_flutter | Flutter Package“, *Dart packages*. https://pub.dev/packages/webview_flutter (pridobljeno 24. november 2022).
[61] „flutter_secure_storage | Flutter Package“, *Dart packages*. https://pub.dev/packages/flutter_secure_storage (pridobljeno 24. november 2022).
[62] „onesignal_flutter | Flutter Package“. https://pub.dev/packages/onesignal_flutter (pridobljeno 2. december 2022).
[63] „OneSignal.RestAPIv3.Client 1.2.0“. https://nuget.org/packages/OneSignal.RestAPIv3.Client/ (pridobljeno 2. december 2022).
[64] „google_fonts | Flutter Package“, *Dart packages*. https://pub.dev/packages/google_fonts (pridobljeno 6. december 2022).
[65] „provider | Flutter Package“, *Dart packages*. https://pub.dev/packages/provider (pridobljeno 7. december 2022).
[66] „signalr_netcore | Flutter Package“, *Dart packages*. https://pub.dev/packages/signalr_netcore (pridobljeno 16. december 2022).
[67] „connectivity_plus | Flutter Package“, *Dart packages*. https://pub.dev/packages/connectivity_plus (pridobljeno 18. december 2022).
[68] „url_launcher | Flutter Package“, *Dart packages*. https://pub.dev/packages/url_launcher (pridobljeno 26. december 2022).
[69] „image_picker | Flutter Package“, *Dart packages*. https://pub.dev/packages/image_picker (pridobljeno 26. december 2022).
[70] „flutter_image_compress | Flutter Package“. https://pub.dev/packages/flutter_image_compress (pridobljeno 4. januar 2023).
[71] „flutter_slidable | Flutter Package“. https://pub.dev/packages/flutter_slidable (pridobljeno 8. januar 2023).
[72] „A deep dive into cursor-based pagination in MongoDB“, *The Engage Blog*, 16. junij 2022. https://engage.so/blog/a-deep-dive-into-offset-and-cursor-based-pagination-in-mongodb/ (pridobljeno 15. januar 2023).

13 PRILOGA

13.1 Primerjava REST in GraphQL programskih vmesnikov

Pri načrtovanju aplikacijskih vmesnikov sta v zadnjih letih najpogosteje uporabljena dva pristopa:

- **REST** (angl. REpresentational State Transfer) je najpogosteje uporabljen skupek smernic za izdelavo programskih vmesnikov (angl. imenovani RESTful APIs). Največkrat je uporabljen pri arhitekturi spletnih storitev oz. spletnih aplikacijskih vmesnikov, osnovanih na HTTP zahtevah (GET, POST, PUT …). [44]

  Podatki so dostopni kot spletni viri (angl. web resources) na različnih URI (Uniform Resource Identifier) naslovih, ki jih definira razvijalec aplikacije (t. i. končne točke vmesnika, angl. API endpoints). Za pridobitev podatkov uporabnik potrebne parametre doda v URI naslov poizvedbe ali pa jih pošlje v telesu zahtevka. Odgovor je najpogosteje v formatu JSON, lahko pa je tudi v drugih formatih (npr. XML ali celo HTML).

  Ena izmed glavnih prednosti tega pristopa je uporaba standardnega protokola in zahtev (HTTP), ki omogoča, da takšen vmesnik deluje brez pomnjenja (angl. stateless). To pomeni, da takšen sistem na strežniku ne hrani nobenih podatkov o seji uporabnika (vključno z avtorizacijo ali avtentikacijo). Vsak zahtevek uporabnika mora namreč vsebovati vse podatke, potrebne za njegovo izvedbo. To pomeni, da ni pomembno, kateri strežnik obdeluje naše zahteve, kar omogoča odlično skalabilnost (stopnjevanje) sistema.

  Glavna slabost REST spletnih vmesnikov je po najinih izkušnjah potreba uporabnika, da točno pozna URI naslove za poizvedbe in obliko podatkov, ki jih dobi nazaj. V ta namen so bili razviti t. i. jeziki za opis REST vmesnikov (angl. RESTful API Description Languages), katerih cilj je avtomatsko izdelati tako dokumentacijo za ljudi kot kodo za prejemanje podatkov. Najpogostejši takšen jezik je specifikacija OpenAPI, znana tudi kot Swagger. Kljub vedno pogostejši uporabi niso povsem uveljavljeni na vseh različnih platformah.

  Za okolje Flutter sva našla le knjižnico *openapi_generator*, ki ni posebno priljubljena, zato sva se odločila, da je ne bova uporabila. Tako sva morala sama v kodo dodati URI naslove končnih točk in definicije razredov entitet. [45]

- **GraphQL** je odprtokodni poizvedbeni jezik in sistem izvajanja (angl. runtime) za aplikacijske vmesnike podjetja Meta. Uporablja se za zagotavljanje robustne strukture storitev, ki delujejo s kompleksnimi podatki. [46]

  Vsaka GraphQL storitev definira tipe, ki opišejo vse podatke, ki jih je mogoče zahtevati s poizvedbo te storitve. To omogoča, da lahko uporabnik zahteva le tiste podatke, ki jih potrebuje (s tem preprečimo »over-fetching«). Za zahtevo se uporablja t. i. jezik za definicijo GraphQL sheme (GraphQL Schema Definition Language), v katerem so podatki predstavljeni kot objekti, sestavljeni iz polj. Poizvedba je zelo podobna rezultatu, za katerega standard predvideva format JSON, zato lahko sklepamo, kaj bo vrnila, brez predhodnega poznavanja strežnika.

Standard podpira tako branje kot spreminjanje podatkov, omogoča pa tudi sprotno poslušanje sprememb podatkov (angl. subscribing), kar doseže z uporabo protokola WebSockets.

Glavna slabost standarda je kompleksnost izdelave dobrega sistema, primernega za pridobivanje različnih, med seboj povezanih podatkov. Zato je njegovo učinkovito uporabo standard treba zelo dobro poznati in razumeti.

Za razliko od standarda REST GraphQL ne uporablja standardnih poizvedb HTTP, zato je za poizvedbe potrebno uporabiti temu namenjene knjižnice.

Čeprav je standard odprtokoden, je vseeno veliko manj popularen kot REST. Posledično je razvijalcem na voljo veliko manj orodij in knjižnic, razvoj katerih je pogosto odvisen od spletne skupnosti prostovoljcev in ne od podjetij. Uradna strežniška knjižnica je namreč na voljo le za jezik JavaScript.

13.2 Glavni tipi relacij med podatki

Na diagramu najine podatkovne baze so jasno vidne tudi relacije med podatki, ki jih uporabljeno ogrodje EF Core avtomatsko upravlja. Predstavljene so s črtnimi povezavami s t. i. sračjo oz. Martinovo notacijo (angl. crow's foot notation). Z njo prikažemo kardinalnost oz. številčnost relacije. [47]

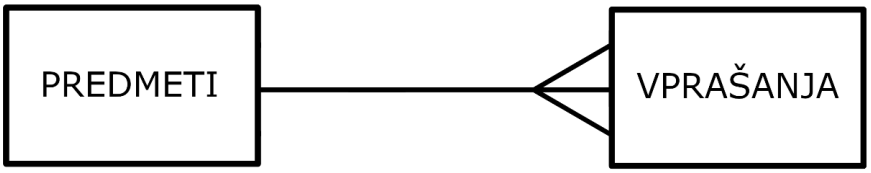
Pri shemi zasledimo tri glavne tipe relacij:

- **Razmerje s kardinalnostjo 1 : M (angl. one-to-many)** predstavlja relacijo med elementoma tipov A in B, pri čemer je element tipa A povezan z večimi elementi tipa B, ampak element tipa B je povezan samo z enim elementom tipa A.

  Takšno razmerje v relacijski podatkovni bazi predstavimo z vključitvijo primarnega ključa elementa tipa A v zapisu elementa tipa B. Takemu ključu pravimo tuji (angl. foreign) ključ.

  Primer takšne relacije je relacija med predmetom in vprašanji. En predmet obsega več vprašanj, a vsako vprašanje spada le pod en predmet. Tako je v tabeli *Questions* (vprašanja) dodan tuji ključ *SubjectId* (identifikator predmeta).

  Pomembno je dodati tudi, da je razmerje s kardinalnostjo 1 : M gledano z druge strani M : 1 (angl. many-to-one).

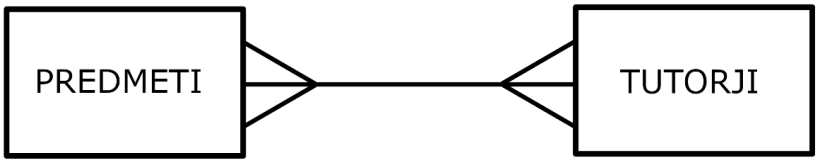


Slika 140: Razmerje s kardinalnostjo 1 : M (lasten vir)

- **Razmerje s kardinalnostjo M : N (angl. many-to-many)** predstavlja relacijo med elementoma tipov A in B, pri čemer sta tako A kot B povezana z več zapisi drugega tipa.

  Takšnega razmerja v relacijski podatkovni bazi ni mogoče neposredno predstaviti, zato relacije tega tipa ponavadi predstavimo s pomočjo t. i. asociativne tabele (angl. associative table, join table), ki je sestavljena iz primarnih ključev obeh entitet. Tako takšno razmerje razdelimo na dve razmerji 1 : M.

  Primer takšne relacije je relacija med tutorji in predmeti, ki jih ti poučujejo. Vsak tutor namreč lahko poučuje več predmetov, vsak predmet pa ima več tutorjev. Asociativna tabela *SubjectTutor* je tako sestavljena iz dveh tujih ključev: *SubjectsId* (primarni ključ predmeta) in *TutorsId* (primarni ključ tutorja).

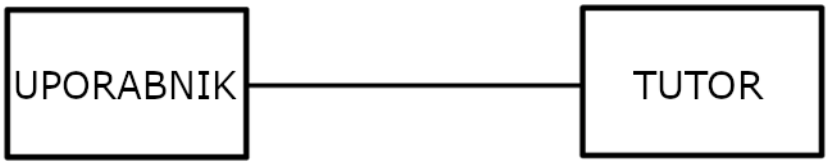


Slika 141: Razmerje s kardinalnostjo M : N (lasten vir)

- **Razmerje s kardinalnostjo 1 : 1 (angl. one-to-one)** predstavlja relacijo med elementoma tipov A in B, pri čemer je en element tipa A povezan s točno enim elementom tipa B.

  Takšno razmerje v relacijski podatkovni bazi predstavimo z vključitvijo primarnega ključa v entiteti v drugi tabeli.

  Primer takšne relacije v najini aplikaciji je relacija med profilom uporabnika in profilom tutorja. Podatki, katere predmete tutor poučuje, so shranjeni v tabeli *Tutors*, ostali podatki o profilu uporabnika pa v tabeli *Users*. Tabela *Tutors* vključuje primarni ključ tabele *Users* (*UserId*).



Slika 142: Razmerje s kardinalnostjo 1 : 1 (lasten vir)

13.3 Upravljanje odvisnosti (angl. dependency injection)

V ogrodju ASP.NET Core se odvisnosti upravljajo preko zasnovnega vzorca, imenovanega upravljanje odvisnosti (angl. dependency injection – DI). Cilj takšnega pristopa je rešiti tri glavne težave standardnega pristopa k odvisnostim:

- povezanost razredov s svojimi implementacijami,

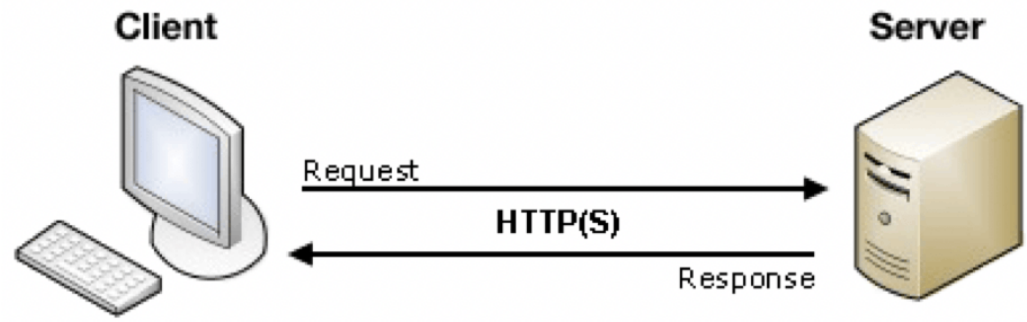
- težave z upravljanjem odvisnosti od odvisnosti in
- težavnost testiranja enot s konkretnimi implementacijami razredov.

Upravljanje odvisnosti te težave rešuje z:

- uporabo vmesnikov (angl. interface) in osnovnih razredov (angl. base class), ki služijo kot abstrakcija dejanske implementacije metod,
- uporabo t. i. zabojnika storitev (angl. service container) – programske rešitve, ki centralizirano poskrbi za ustvarjanje primerkov storitev in njihovo odstranitev po uporabi ter
- »injiciranjem« (angl. injection) odvisnosti v konstruktor razreda (metodo za njegovo ustvarjanje). [28]

13.4 HTTP-zahtevki in odzivi

Osnovo delovanja spletnih vmesnikov, ki temeljijo na arhitekturi REST, predstavlja protokol HTTP (angl. Hyper Text Transfer Protocol), preko katerega poteka izmenjava podatkov. Protokol je osnovan na arhitekturi odjemalec-strežnik, saj temelji na osnovi zahtevkov (angl. requests), ki jih odjemalec (angl. client) pošlje strežniku, ki nanje odgovori z odzivom (angl. response).



Slika 143: Delovanje HTTP zahtevkov in odzivov [48]

```
GET /forum/subjects HTTP/1.1
Host: api.opentutor.si
Accept: application/json
```

```
HTTP/1.1 200 OK
Server: nginx/1.14.0 (Ubuntu)
Date: Sun, 08 Jan 2023 19:08:40 GMT
Content-Type: application/json; charset=utf-8
Content-Length: 50
Connection: keep-alive

[{"questions_count":1,"name":"Matematika","id":1}]
```

Slika 144: Zgradba http-sporočila (lasten vir)

Vsako HTTP-sporočilo (zahtevek ali odgovor) je šifrirano kot besedilo in sestavljeno iz glave in telesa.

Zahtevek je sestavljen iz:

- **Vrstice za zahtevo** (angl. request line) z argumenti: HTTP-metoda (predstavlja namen zahtevka – npr. GET za pridobitev podatkov, POST za ustvarjanje novih podatkov, PUT za posodabljanje obstoječih podatkov in DELETE za odstranitev obstoječih podatkov), URI zahtevanega vira in verzija HTTP-protokola, ločenimi s presledki

- **Glav zahtevka** (angl. request headers) – neobvezni parametri, s katerimi zahtevku dodamo podatke. Zapisujejo se kot pari imen in vrednosti (angl. key-value pairs), ločeni z vejico.

- **Prazne vrstice –** ločuje glavo zahtevka od telesa

- **Telesa zahtevka** (angl. request body) – neobvezni dodatni podatki

Odziv je sestavljen iz:

- **Statusne vrstice** (angl. status line) z argumenti: verzija HTTP-protokola, trimestna statusna koda zahtevka (angl. status code) in njena obrazložitev v angleščini (angl. reason phrase, npr. »Not Found«, »OK«)

- **Glav odziva** (angl. response headers) – njihova struktura je enaka kot pri zahtevku, opisujejo podatke, poslane v telesu odziva (njihova dolžina, vrsta ipd.)

- **Prazne vrstice –** ločuje glavo odziva od telesa

- **Telesa odziva** (angl. response body) – zahtevani podatki v opisanem formatu (pri najinem spletnem vmesniku je to JSON)

Naloga izdelovalca programskega spletnega vmesnika je tako določiti končne točke, ki se med sabo razlikujejo po URI-naslovu in HTTP-metodi, in jih povezati s kodo, ki odjemalcu vrne ustrezne podatke.

13.5 Pristopi do shranjevanja slik

Najpreprostejša rešitev je, da slike shranjujemo kot zbirko binarnih podatkov (angl. Binary Large OBject, blob) v podatkovni bazi (kot stolpec oz. atribut v tabeli), saj imamo tako slike vedno na voljo. Takšen pristop ima vrsto slabosti [49]:

- podatkovna baza je (za razliko od datotečnega sistema) optimizirana za shranjevanje veliko krajših nizov podatkov,

- slike je treba po branju iz podatkovne baze pretvoriti iz niza bitov, kar zahteva dodatno kodo in predstavlja relativno zahtevno opravilo,

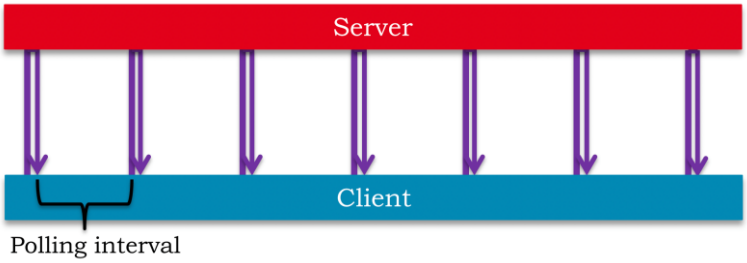
- občutno poveča količino podatkov, shranjenih v podatkovni bazi, kar lahko upočasni njeno delovanje, indeksiranje in oteži možnost varnostnega kopiranja.

Pogosta rešitev, ki sva jo zasledila, je shranjevanje slik na datotečni sistem strežnika. Tako v podatkovno bazo shranimo le pot do datoteke s sliko in se posledično izognemo opisanim slabostim. Glavna slabost tega pristopa pa je težavnost skalabilnosti, saj so vse datoteke shranjene na disku enega strežnika, do katerega drugi nimajo dostopa.

Najpogosteje uporabljen pristop je tako danes gostovanje slik na zunanjem strežniku, do katerega lahko preprosto dostopamo prek spleta. Priljubljena takšna rešitev je Amazonov AWS S3 (Amazon Web Services Simple Storage Service), ki omogoča nizkocenovno oblačno shrambo datotek v t. i. vedrih (angl. S3 buckets). Prednost takšne rešitve je tudi njena izjemna skalabilnost, saj uporabnikom omogoča shranjevanje poljubne količine podatkov, pri čemer uporabnik plača le za porabljeno.

13.6 Primerjava tehnologij za sprotni pogovor

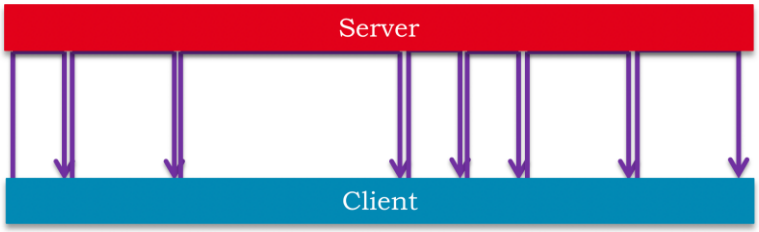
13.6.1.1 Redno izpraševanje (angl. regular polling)



Slika 145: Redno izpraševanje [48]

Najpreprostejša rešitev je ponavljajoče se pošiljanje zahtevkov strežniku za nove podatke vsak izpraševalni interval (angl. polling interval). Glavna slabost takšnega pristopa je visoko število nepotrebnih in ponovljenih zahtev, ki jih mora obdelati strežnik.

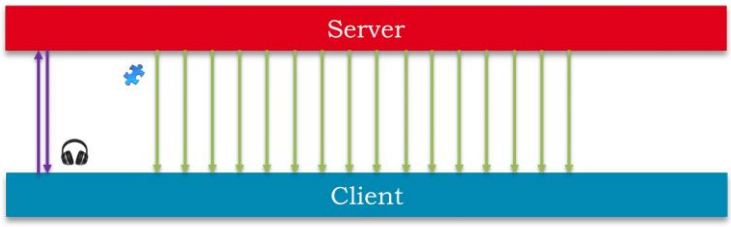
13.6.1.2 Dolgotrajno izpraševanje (angl. long polling)



Slika 146: Dolgotrajno izpraševanja [48]

Izboljšana rešitev je pošiljanje zahtevkov, na katere strežnik ne odgovori, dokler ne pride do novih podatkov (npr. uporabnik pošlje novo sporočilo). Slabost takšnega pristopa je časovna omejitev odjemalca, po kateri ta smatra, da je prišlo do težave na strežniku. Pravimo, da je zahteva potekla. Posledično je glavna težava potreba po posebni nastavitvi tako odjemalca kot strežnika.

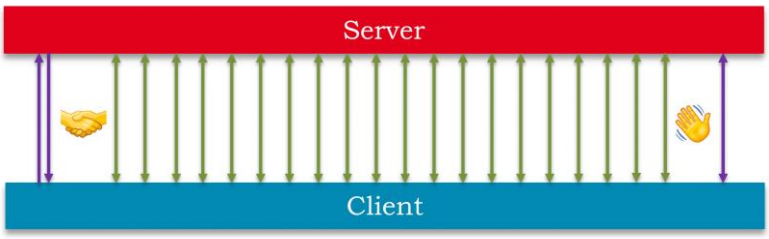
13.6.1.3 Dogodki, poslani s strežnika (angl. Server-Sent Events)



Slika 147: Protokol Server-Sent Events [48]

Protokol, imenovan Server-Sent Events, olajša sprotno komunikacijo, vendar omogoča le enostransko povezavo iz strani strežnika k odjemalcu. Rešitev je vedno pogosteje uporabljena, saj je del spletnega standarda HTML5.

13.6.1.4 »Spletne vtičnice« (angl. WebSockets)



Slika 148: Spletni vtičniki [48]

Del standarda HTML5 je tudi protokol WebSockets, ki omogoča pravo obojestransko sprotno komunikacijo med odjemalcem in strežnikom. Za vzpostavitev povezave je potreben proces, imenovan rokovanje (angl. handshake), v katerem se odjemalec in strežnik pogajata o uporabljenih standardih. Po uspešnem rokovanju se odpre trajna povezava z majhno zakasnitvijo. Rešitev tako predstavlja odličen način za sprotno povezavo med najrazličnejšimi spletnimi napravami, saj je podprta na številnih platformah.

13.7 Primerjava glavnih pristopov do paginacije

Poznamo dva glavna pristopa do paginacije:

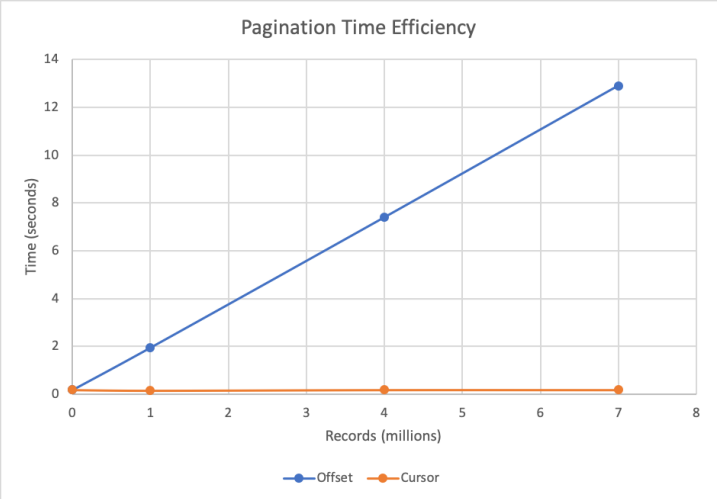
- **paginacija, osnovana na odmiku** (angl. offset-based pagination) – Odjemalec pošlje dva argumenta: omejitev (angl. limit), ki predstavlja največje število elementov na vsaki strani, in odmik (angl. offset), ki predstavlja mesto elementa, od koder želi podatke. Vmesnik tako v tabeli, ki je urejena po dogovorjeni vrednosti, preskoči toliko prvih elementov, kot je odmik, in pošlje toliko elementov, kot je omejitev.

  Prednost takšnega pristopa je relativna preprostost implementacije (ukaza LIMIT in OFFSET v jeziku SQL) in možnost, da lahko preskočimo na katerokoli stran podatkov (odmik = št. strani * omejitev), slabost pa so nezanesljivi rezultati, saj se

lahko med zahtevkoma za sosednji strani dodajo novi podatki, ki lahko vodijo v izpuščene ali podvojene vnose. Da lahko podatkovna baza elemente preskoči, mora prebrati vse elemente zamika, zato je ta pristop primernejši za manjšo količino podatkov.

- **paginacija, osnovana na »kazalcu«** (angl. cursor-based pagination) – Je pristop, ki rešuje težave prejšnjega. Odjemalec v prvem zahtevku pošlje le omejitev (angl. limit) in vmesnik mu vrne zahtevano število elementov iz urejene tabele, hkrati pa mu pošlje tudi edinstveno vrednost zadnjega neprebranega elementa, t. i. kazalec (angl. cursor, continuation token). To vrednost, ki predstavlja točko nadaljnjega branja podatkov, pošlje odjemalec v naslednjih zahtevkih.

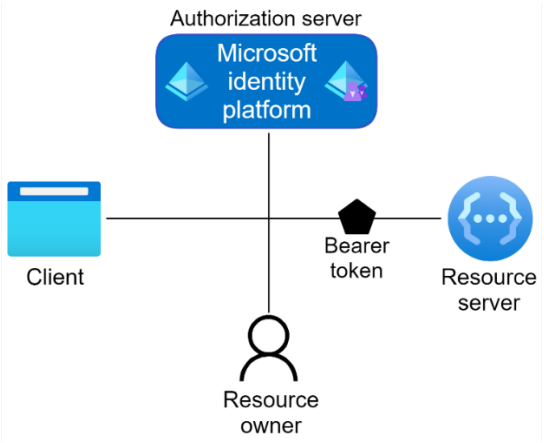
  Tako se izognemo večkratnemu branju enakih podatkov in dosežemo enakomerne rezultate, ki jih lahko dodatno pospešimo z indeksiranjem vrednosti kazalca v podatkovni bazi. Glavna slabost tega pristopa je, da je uporabniku onemogočeno, da kakšno stran preskoči, saj tako ne dobi vmesnih vrednosti kazalca. Težavo predstavlja tudi težja implementacija rešitve, saj mora biti temu prilagojena tako struktura podatkovne baze (vrednosti moramo dodajati zaporedno) kot odjemalec, ki mora za pravilno delovanje ustrezno upravljati z vrednostjo kazalca.



Slika 149: Primerjava hitrosti paginacije, osnovane na odmiku, s hitrostjo paginacije, osnovane na kazalcu [32]

13.8 OAuth 2.0 in JWT žetoni

Prijava z Microsoftovim računom poteka preko protokola OAuth 2.0 (Open Authorization). Gre za metodo avtorizacije, kjer je določeni aplikaciji dodeljen omejen dostop do zaščitenih podatkov preko protokola HTTPS z uporabo niza znakov v glavi zahtevka (t. i. Bearer Authentication – *Authorization: Bearer <vrednost žetona>*), imenovanega dostopni žeton (angl. access token), namesto prijavnih podatkov uporabnika (angl. login credentials).



Slika 150: Shema prijave z Microsoftovim računom [50]

Za delovanje protokola so potrebne sledeče osnovne komponente:

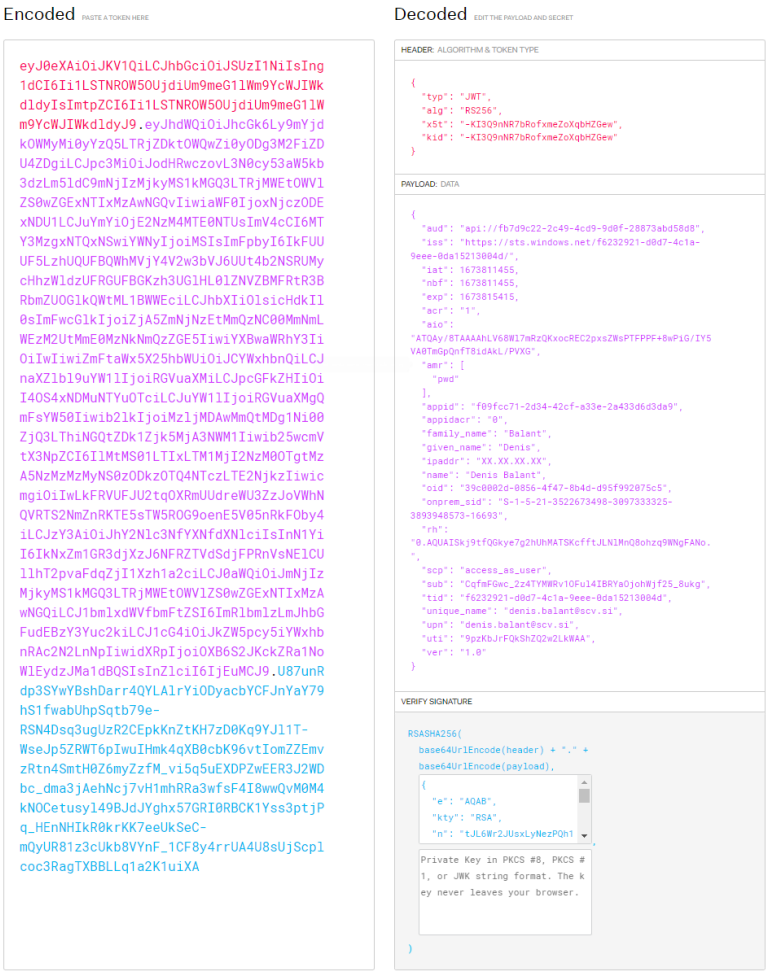
- lastnik vira oz. podatkov (angl. resource owner) – uporabnik, ki ima dostop do podatkov oz. je njihov lastnik,
- odjemalec (angl. client) – aplikacija, ki zahteva dostop do zaščitenega vira (zanj potrebuje dostopni žeton),
- avtorizacijski strežnik (angl. authorization server) – strežnik, ki izdaja dostopne žetone odjemalcem in
- strežnik z viri (angl. resource server) – strežnik, ki ima v lasti uporabnikove podatke in odjemalcu omogoči dostop do njih z ustreznim dostopnim žetonom.

Čeprav standard ne definira formata za dostopni žeton, je zanj najpogosteje uporabljen standard JSON Web Token (skrajšano JWT), ki omogoča vključitev digitalno podpisanih podatkov v formatu JSON (pari ključ-vrednost).

Najpomembnejši tak podatek je rok izteka, s čimer lahko omejimo trajanje žetonove veljavnosti in posledično izboljšamo varnost naše aplikacije. [51]

Žeton JWT je sestavljen iz treh delov, ločenih s pikami [51]:

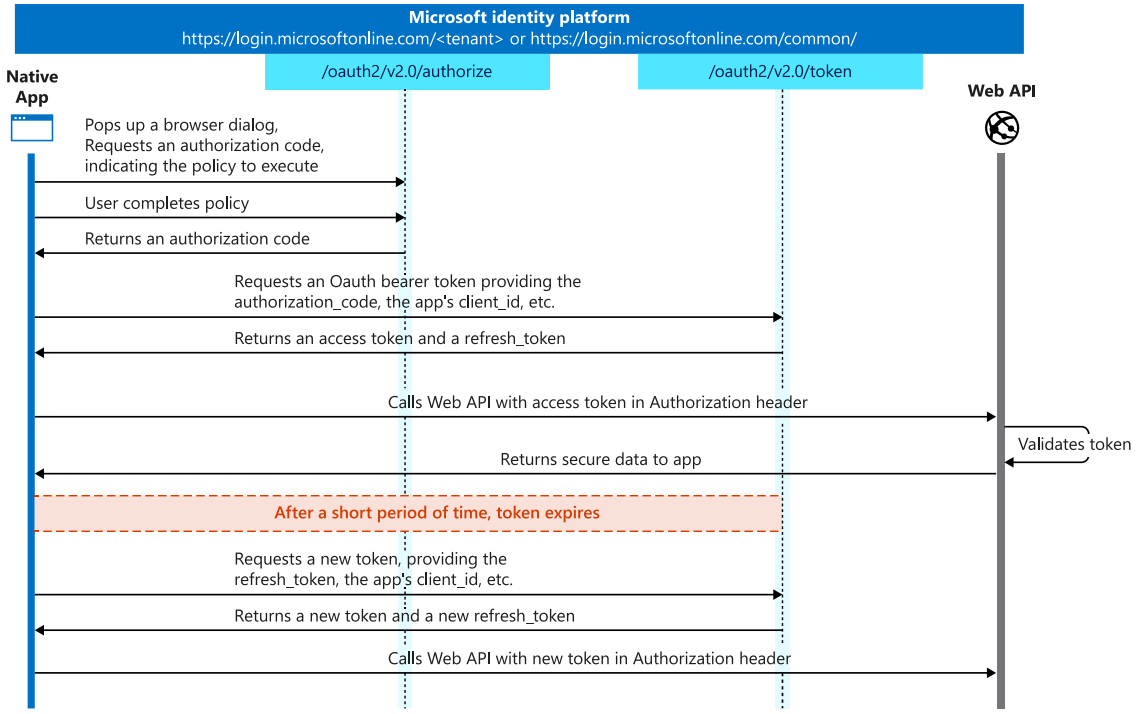
- Glava (angl. header) – V njej sta shranjena vrsta žetona in algoritem, uporabljen za šifriranje (najpogosteje HMAC SHA256 ali RSA).
- Tovor (angl. payload) – Vsebuje podatke o uporabniku (angl. claims), kodirane z algoritmom Base64Url. Najpogostejši so: iss (issuer – izdajatelj), exp (expiration date – rok izteka v obliki UNIX timestamp), sub (subject – osebek – edinstven identifikator uporabnika), aud (audience – občinstvo – komur je žeton namenjen).
- Digitalni podpis (angl. signature) – Vsebuje digitalni podpis združene glave, tovora in skritega niza podatkov (t. i. skrivnosti – angl. secret). Z njim lahko zagotovimo verodostojnost žetona in njegovega izdajatelja.



Slika 151: Primer dekodiranega JWT dostopnega žetona [52]

Pomemben pojem OAuth so t. i. okvirji (angl. scopes), s katerimi ob prijavi navajamo natančen razlog za dostop do zaščitenih podatkov (pravice). Imajo obliko besednih vrednosti, ki so popolnoma odvisne od implementacije. [53]

Da lahko odjemalec prejme žeton, mora izvesti določene korake, ki jih imenujemo odobritev (angl. grant). Poznamo več vrst odobritev, najpogosteje uporabljena je t. i. odobritev z avtorizacijsko kodo (angl. authorization code grant oz. flow), ki jo tudi uporabljava v najini aplikaciji.



Slika 152: Potek avtorizacijske kode [54]

Potek se začne s klicem končne točke */authorize* Microsoftovega programskega vmesnika na naslovu *https://login.microsoftonline.com/<Id najemnika>/oauth/v2.0*. V zahtevku odjemalec zahteva željene pravice (okvirje) od uporabnika (npr. *openid*, *offline_access*).

Uspešen odziv vrne avtorizacijsko kodo, s katero lahko odjemalec do njenega izteka (ponavadi 10 min) zahteva dostopni žeton (access token) s klicem na končno točko */token*. Za izboljšano varnost se za dostop do žetona pogosto uporabljajo tudi t. i. »skrivnosti« (skriven niz znakov odjemalca) ali digitalni certifikati, ko so ti lahko varno shranjeni.

Po uspešni pridobitvi žetona nas aplikacija sama preusmeri na nastavljen preusmeritveni naslov (angl. redirect URI).

Kot že omenjeno, imajo dostopni žetoni omejen rok veljavnosti. Nemoten dostop do zaščitenih vsebin brez ponovne prijave omogočajo t. i. osvežilni žetoni (angl. refresh token), za uporabo katerih moramo navesti okvir (scope) *offline_access*. Z osvežilnim žetonom, ki ima veliko daljši rok veljavnosti, lahko zahtevamo nov dostopni žeton s ponovnim klicem na končno točko */token*. Temu postopku pravimo osveževanje žetona in se ponavadi izvaja avtomatsko v ozadju brez vednosti uporabnika. [54]

13.9 Intervjuja s tutorjema (testiranje)

13.9.1 Intervju z Oskarjem Žerakom Urbancem

Misliš, da je bila najina aplikacija potrebna? Je tutorstvo premalo izkoriščeno?
Tutorstvo je na naši šoli aktivno že nekaj časa, vendar v tem času ni zares zaživelo. Po mojem mnenju je aplikacija OpenTutor zelo dober pripomoček za medsebojno komunikacijo na ravni dijakov, saj olajša komunikacijo med tutorjem in dijakom z učnimi težavami. Aplikacija ima zelo velik potencial, hkrati pa predstavlja tutorstvo v najboljši luči.

Se ti funkcije, ponujene v aplikaciji, zdijo primerne za njen namen?
Ponujene funkcije se mi zdijo zelo primerne. Možnost klepeta z dijakom in pošiljanja slik je koristna za hitre odgovore na vprašanja, vprašanja, ki jih lahko zastaviš v javni forum, pa so lahko zelo uporabna, saj lahko nanje odgovori več tutorjev hkrati, kar nudi različne poglede na problem. Aplikacija nudi tudi enostavno iskanje tutorjev, kar je zelo dobrodošlo pri vzpostavitvi prvega stka med dijaki.

Ti je aplikacija olajšala vzpostavitev prvega stika z iskalci učne pomoči?
Aplikacija mi je zagotovo olajšala vzpostavitev prvega stika. Dijaki, ki iščejo pomoč, lahko tutorja kontaktirajo preko direktnih sporočil ali pa vprašanje napišejo na forum. Tudi sam sem preko foruma večkrat sodeloval in pomagal dijakom nižjih letnikov pri vprašanjih.

Meniš, da je aplikacija spodbudila dijake k iskanju učne pomoči?
Aplikacija močno olajša iskanje učne pomoči, saj je pomoč zagotovljena tudi tistim, ki ne želijo direktno kontaktirati tutorja ali pa se z njim nimajo časa sestati v živo. Moje mnenje je, da forum, kjer je mogoča tudi pasivna uporaba, močno pripomore k angažiranosti uporabnikov ter odkriti komunikaciji in nadejam se, da bo aplikacija spodbujala dijake k bolj odprti komunikaciji tudi v prihodnje.

Kakšen se ti zdi dizajn aplikacije? Je intuitiven? Je aplikacija preprosta za uporabo in primerna tudi za mlajše uporabnike (npr. osnovnošolce)?
Na splošno aplikacija izgleda zelo dobro, dizajn ni pretiran oz. »kičast« in služi svojemu namenu. Uporabniška izkušnja se mi zdi precej dobra, ker je aplikacija za uporabo dokaj preprosta in menim, da bi jo zato znali uporabljati tudi mlajši.

So funkcije znotraj aplikacije dovolj odzivne?
Pri uporabi aplikacije še nisem naletel na težave in nasploh ocenjujem, da aplikacija deluje dobro in nemoteno. Funkcije so zelo odzivne in nimam večjih pripomb.

Imaš kakšne predloge za izboljšave? Kaj bi spremenil?
Aplikacija je dobro zasnovana, vendar vedno je prostor za napredek. Všeč bi mi bilo, da bi v odgovoru na vprašanje lahko napisal več besed in tako bolj temeljito odgovoril na zastavljena vprašanja. Koristno bi bilo dodati tudi obvestilo ob zastavljenem vprašanje iz predmetov, ki jih poučujem.

13.9.2 Intervju z Rokom Hudournikom

Misliš, da je bila najina aplikacija potrebna? Je tutorstvo premalo izkoriščeno?
Menim, da je aplikacija še kako potrebna, saj veliko ljudi potrebuje pomoč pri razumevanju učne snovi. Pomoč navadno potrebujejo zlasti nižji letniki, ki se velikokrat bojijo poiskati pomoč, novonastala aplikacija pa ponuja anonimen pristop k težavi in nasploh se mi zdi, da so ljudje preko ekrana veliko bolj sproščeni in komunikativni. Kar se tiče tutorstva menim, da še ni zares zaživelo in je bistveno premalo izkoriščeno.

Se ti funkcije, ponujene v aplikaciji, zdijo primerne za njen namen?
Funkcije se mi zdijo primerne in za aplikacijo takšnega tipa tudi potrebne. Aplikacija ponuja več različnih orodij, od foruma, iskanja tutorjev pa vse do neposrednih sporočil. Oblikovano

okolje je zelo lepo zasnovano in nudi kar se da primerne funkcije za efektivno nudenje in prejemanje učne pomoči.

Ti je aplikacija olajšala vzpostavitev prvega stika z iskalci učne pomoči?
Stik je definitivno lažje vzpostaviti preko aplikacije kot v živo, saj lahko kontaktiraš praktično kateregakoli tutorja v vsakem trenutku. Sam na aplikaciji sodelujem pretežno kot tutor in imam zato z vzpostavitvijo prvega stika preko nje veliko manj izkušenj.

Meniš, da je aplikacija spodbudila dijake k iskanju učne pomoči?
Aplikacija je še dokaj sveža, zato konkretno ne morem odgovoriti na to vprašanje, menim pa, da so odzivi po par tednih uporabe zelo dobri. Med drugim sem se pogovarjal tudi s sošolci, ki jim je novo orodje za iskanje učne pomoči zelo pomagalo pri razumevanju učne pomoči, zato menim, da je potencial v prihodnje zelo velik.

Kakšen se ti zdi dizajn aplikacije? Je intuitiven? Je aplikacija preprosta za uporabo in primerna tudi za mlajše uporabnike (npr. osnovnošolce)?
Dizajn aplikacije mi je zelo všeč. Okolje ponuja zelo dobro preglednost in je zelo intuitivno. Aplikacija je zelo preprosta in vsakdo jo lahko uporablja. Funkcije so jasno zastavljene in predstavljene, predvsem pa niso dvoumne, zato menim, da je aplikacija primerna tudi za osnovnošolsko rabo.

So funkcije znotraj aplikacije dovolj odzivne?
Pri uporabi aplikacije sem naletel na problem s forumom, ko mi je kljub izbiri matematike prikazalo vprašanja za druge predmete. Drugače aplikacija deluje brezhibno in brez večjih motenj. Z uporabniško izkušnjo sem zelo zadovoljen in bi uporabo toplo priporočil vsem tistim, ki bi želeli nuditi učno pomoč, ali pa tistim, ki jo potrebujejo.

Imaš kakšne predloge za izboljšave? Kaj bi spremenil?
V aplikaciji ni veliko prostora za izboljšave. Sam bi predlagal tudi možnost spletnih učilnic ali pa video klicev, da bi bil stik med uporabniki lahko še bolj pristen tudi v popoldanskih urah, ko se dijaka ne moreta srečati v šoli.