Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Žiga Kljun

Mobilna podpora skladiščnemu poslovanju

diplomsko delo

univerzitetni študijski program
prve stopnje
računalništvo in informatika

Mentor: doc. dr. Veljko Pejović

Ljubljana, 2018

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Mobilna podpora skladiščnemu poslovanju.

Ob tej priložnosti bi se rad zahvalil mentorju za nasvete in smernice pri izdelavi diplomskega dela. Zahvalil bi se podjetju E.R.S. Rokada, ki mi je omogočila okolje in dajala napotke za razvoj aplikacije. Zahvalil pa bi se tudi staršem, za podporo skozi celoten potek študija. Hvala.

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
SQL	Structured Query Language	Strukturirani povpraševalni jezik za delo s podatkovnimi bazami
API	Application Programming Interface	Vmesnik za namensko programiranje
URL	Uniform Resource Locator	Enolični krajevnik vira
VPN	Virtual Private Network	Virtualno zasebno omrežje
MVC	Model View Controller	Model Pogled Nadzornik
RAM	Random Access Memory	Bralno Pisalni Pomnilnik
CPU	Central Processing Unit	Centralna Procesna Enota
CRUD	Create, Read, Update, Delete	Ustvari, Beri, Posodobi, Briši
REST	Representational State Transfer	Predstavitveni prenos stanja
SOAP	Simple Object Access Protocol	Protokol preprostega dostopa do predmeta

Povzetek

Naslov: Mobilna podpora skladiščnemu poslovanju

Avtor: Žiga Kljun

Podjetja so včasih za nadzor nad materialom in izdelki v skladišču skrbela tako, da so si podatke zapisovala na list papirja. Sčasoma so se količine tako povečale, da so morali podatke s papirja zapisati še s pomočjo namiznega računalnika v bazo, saj je bilo podatkov preveč, da bi vse skupaj vodili ročno. Tukaj smo videli priložnost, da razvijemo mobilno podporo skladiščnemu poslovanju, ki bi privedla do boljše učinkovitosti in optimizirala časovno porabo. Uporabniku bi omogočala, da inventuro, izdajnico, povratnico in prevzemnico izdela kar preko mobilne Android naprave, kjer se podatki zapišejo neposredno v bazo brez dodatne uporabe namiznega računalnika. V diplomskem delu je opisan razvoj mobilne aplikacije, vključno s postavitvijo spletnega APIja, razvojem same Android aplikacije ter z delom na bazi.

Ključne besede: Android, skladišče, api, sql.

Abstract

Title: Mobile computing support for warehouses

Author: Žiga Kljun

Information about the inventory of a warehouse stocks and products used to be kept manually by means of paper forms. As the stocked items quantity and the frequency of transactions increased, the manually collected data was additionally kept in a computer database. In this work we use the opportunity to further improve the warehouse management process by implementing a mobile-based solution for inventory handling. Our solution relies on an Android application to enable direct interaction with the database without a need for a desktop PC. In this thesis we describe the application development, the related API design and development, the interaction with the database, and the extensive evaluation process we have conducted.

Keywords: Android, warehouse, api, sql.

Poglavje 1

Uvod

Podjetja so včasih nadzor skladišč opravljala ročno z uporabo evidenčnih papirnatih formularjev. Ko so dobili pošiljko, so podatke zabeležili na papir in enako storili, ko so iz skladišča kaj odposlali. Sčasoma so količine pošiljk in posledično podatkov tako narasle, da je stvar postala težko obvladljiva. Uveljavili so se računalniki in baze, kamor so podatke o inventuri shranjevali. Skladišča so imela oziroma imajo še vedno računalniške programe, kjer jim vmesnik omogoča, da podatke o operacijah (inventura, izdaja, prevzem...) vnašajo v sistem. Še vedno pa morajo podatke najprej nekam zapisati (papir, tablica) in jih kasneje pretipkati v sistem, saj računalnika nimajo na mestu operacije, na primer ob prevzemu. Tukaj nastopi naša mobilna aplikacija, ki skladiščnikom omogoča, da se znebijo vmesnega (odvečnega koraka) in podatke zapišejo kar preko Android tablice neposredno v bazo.

Konkretni prispevki diplomske naloge so naslednji:

- programiranje posrednika (API-ja) med Android aplikacijo in bazo. Diplomsko delo vsebje programiranje kode ter objavljanje API-ja na strežnik.

- programiranje Android aplikacije. Diplomsko delo vsebuje celotno implementacijo vizualnega in logičnega dela aplikacije ter njenih gradnikov.

- testiranje in evalvacija. Podrobno smo preverili pravilnost in kakovost implementiranih rešitev (API-ja in Android aplikacije). Uporabili smo različne metode.

V poglavju 2 predstavimo motivacijo za naše diplomsko delo. Sestavljeno je iz opisa problema, opisa funkcij, ki se izvajajo v skladiščih, ter analize obstoječih rešitev. V poglavju 3 predstavimo predlagano rešitev. Sem spadajo predstavitve naše ideje o začrtani aplikaciji, arhitekture aplikacije, razmerij med entitetami ter osrednjih orodij, uporabljenih pri izdelavi diplomskega dela. Poglavje 4 je namenjeno predstavitvi implementacije. Opisuje torej razvoj vseh komponent aplikacije in njeno delovanje. Poglavje 5 je namenjeno evalvaciji in testiranju. Vsebuje evalvaciji API-ja ter Android aplikacije. V tem poglavju je predstavljeno tudi testiranje v realnem okolju z metodo "Shadowing" ter microbenchmarking. V 6 poglavju povzamemo sklepne ugotovitve in podamo nekaj smernic za naprej.

Poglavje 2

Motivacija

2.1 Opis problema

Glavni izziv pri evidentiranju materialov in izdelkov v velikih skladiščih predstavlja količina podatkov o evidentiranih izdelkih, ter hitrost spreminjanja stanja. Zaradi tega tradicionalne metode evidentiranja, ki se zanašajo na papirne evidenčne formularje ali pregledne datoteke, ne zadostujejo več. O problemu govori tudi članek "Paperless warehouse management system"iz leta 1998, ki piše o potrebi po zanesljivem in logičnem skladiščnem sistemu, ki ne bo več uporabljal papirnatih formularjev. Sistem mora biti dobro organiziran in sistematičen in hkrati dovolj prilagodljiv za različne vrste izdelkov. V članku ima predlagana rešitev en server s podatki, na katerega je priključenih več računalnikov. Računalniki, ki se sicer med seboj razlikujejo (osebni računalniki, terminali z radijskimi frekvencami, prenosni računalniki) so namenjeni upravljanju skladiščnega poslovanja.

Skladišča so se posledično modernizirala in papirne evidenčne formularje zamenjala z računalniškimi poslovnimi informacijskimi sistemi. Kljub večjemu nadzoru in boljši organiziranosti informacij proces še vedno ni dobro časovno optimiziran. Časovno obdobje med opravljeno storitvijo (na primer prevzem) in zapisom v bazo je še vedno relativno veliko, saj mora skladiščnik podatke o evidentiranih izdelkih sprva še vedno zapisati na nek prenosni me-

dij in kasneje iz njega te prepisati še v računalniški poslovno informacijski sistem. S tem prihaja do podvajanja dela, kar pomeni slabšo časovno optimizacijo ter zaradi neizkoriščenosti časa posledično manjši dobiček.

V tem problemu smo videli poslovno priložnost za razvoj mobilne aplikacije, ki bi ta čas zmanjšala oziroma preprečila dvojni vnos evidenčnih podatkov. Aplikacija bo skladiščnikom omogočala, da podatke o evidentiranih izdelkih prenesejo neposredno v bazo brez posredovanja računalniškega poslovnega informacijskega sistema in dvojnega vnosa podatkov. Prav tako bo naša rešitev skladiščniku omogočila uporabo lastnosti mobilnih naprav, ki jih prej ni imel. Sem spadajo branje črtne kode, predlogi količin, prikazovanje lastnosti izdelka in podobne zadeve. Tudi to so funkcije, ki pripomorejo k zmanjšanju časovnega obdobja med opravljeno storitvijo in zapisom evidentiranih podatkov v bazo. V diplomskem delu v Android aplikacijo implementiramo naslednje operacije:

- inventuro (opisana v poglavju 2.2.1)
- izdajo (opisana v poglavju 2.2.2)
- prevzem (opisan v poglavju 2.2.3)
- povračilo (opisano v poglavju 2.2.4)

2.2 Aplikacijine funkcije

2.2.1 Inventura

Inventura oz. popis stanja je skladiščni proces, ki se običajno izvaja enkrat letno. V nekaterih primerih tudi pogosteje. Gre za popis zaloge dejanskega stanja (v skladiščih), ki se nato primerja z zabeleženo zalogo. Podatki o evidentiranih izdelkih se načeloma ujemajo, vendar občasno prihaja tudi do odstopanj, zaradi česar se proces inventure tudi izvaja. V primeru odstopanj je potrebno izvor le-tega tudi izslediti in preprečiti njegovo morebitno pojavitev. Naša aplikacija bo skladiščniku omogočala, da bo ob pregledu skladišča

z Android napravo poskeniral črtno kodo izdelka ter vnesel njegovo količino, ki se bo nato zapisala v bazo. Android naprava bo imela čitalec črtnih kod že vgrajen in nam bo ob uspešno prebrani kodi v polje za vnos zapisala ustrezno številko.

2.2.2 Izdajnica

Izdajnica je dokument, ki se ustvari ob procesu izvajanja izdaj. Dokument vsebuje evidentirane podatke o izdelkih ter njihovi zalogi, ki je bila izstavljena iz skladišča. Izdajnica se ustvari na podlagi prej ustvarjenega kupčevega naročila, s katerega se preneseta izdelek in količina. Pri izdaji se v bazi stanje zaloge zmanjšuje.

2.2.3 Prevzemnica

Prevzemnica je dokument, ki se ustvari ob procesu izvajanja prevzemov. Dokument vsebuje evidentirane podatke o izdelkih ter njihovi zalogi, ki so bili prevzeti v skladišče. Prevzemnica se ustvari na podlagi prej ustvarjene izdajnice, s katere se preneseta izdelek in količina. Pri prevzemnici se v bazi stanje zaloge povečuje. Skladiščnik bo v naši aplikaciji vnesel delovni nalog. Nato bo aplikacija v bazi poiskala ustrezno izdajnico ter ponudila skladiščniku ustrezne postavke, ki jim bo ta nastavil prevzeto količino.

2.2.4 Povratnica

Povratnica je dokument, ki se ustvari ob procesu vračanja slabih kosov oziroma izdelkov. Dokument vsebuje evidentirane podatke o izdelkih, ki so bili vrnjeni dobavitelju, saj je bilo z njimi nekaj narobe, ter o zalogi teh izdelkov. Povratnica se ustvari na podlagi prej ustvarjene prevzemnice, s katere se preneseta izdelek in količina. Pri povratnici se v bazi stanje zaloge zmanjšuje, saj količine izdelkov vračamo in niso več shranjene pri nas. Skladiščnik bo v naši aplikaciji vnesel vrsto in številko prevzemnice. Nato bo aplikacija

na zaslonu ponudila ustrezne postavke, katerim bo uporabnik nato nastavil vrnjeno količino.

2.3 Obstoječe rešitve

Pri analizi konkurenčnih izdelkov smo se osredotočil zgolj na slovenski trg, saj gre za trg, na katerega ciljamo. Na tem trgu imamo v primerjavi s tujejezičnimi aplikacijami veliko konkurenčno prednost, saj delavci v skladiščih pogosto niso vešči angleškega oziroma tujega jezika in zato močno preferirajo rabo slovenščine. V primerjavi s tujimi ponudniki mi ne ponujamo zgolj slovenske verzije programa, pač pa tudi podporo uporabnikom in navodila v slovenščini. Pri konkurenčnih izdelkih bomo analizirali, v kolikšni meri izpolnjujejo naše pogoje oziroma zahteve, ki smo jih definirali pred začetkom razvoja naše aplikacije. Te zahteve predstavlja implementacija naslednjih funkcij:

- pregled in ažuriranje inventurnih postavk,
- kreiranje izdajnic na podlagi kupčevega naročila,
- kreiranje prevzemnic na podlagi izdajnic,
- kreiranje povratnic na podlagi prevzemnice.

TronWareHouse [12]

TronWarehouse je razširitev drugega programa - TronInterCenter. Aplikacija od naših zahtev izpolnjuje kreiranje izdajnic in prejemnic. Manjkata ji funkciji za kreiranje povratnic ter pregled in ažuriranje inventurnih postavk.

Špica WMS [13]

WMS je samostojna aplikacija za skladiščno poslovanje, vendar pa omogoča povezljivost s poljubnim ERP sistemom preko XML datotek. Aplikacija ima

implementirane rešitve za pregled in ažuriranje inventurnih postavk, kreiranje prevzemnic in izdajnic. Ne izpolnjuje pa zahteve po kreiranju povratnic.

PerfTech Skladiščno poslovanje [14]

Pri PerfTech-u gre za samostojen produkt, ki tako kot WMS zgoraj nudi rešitve za inventuro, izdajnice in prevzemnice, medtem ko ne vsebuje implementacije kreacije povratnic.

mVasco [15]

Podobno kot pri TronWarehouse gre za razširitev namiznega programa - Vasco. Ponovno gre za produkt, ki nudi uporabniku rešitve za inventuro, izdajnice ter povratnice, manjka pa implementacija za kreacijo povratnic.

Minoa Logist [16]

Minoa Logist sicer nudi podporo skladiščnemu poslovanju, vendar se osredotoča na druge zadeve, kot vse ostale aplikacije do sedaj. Tudi v tem primeru gre za dodatek namizni aplikaciji. Aplikacija se v celoti sicer ukvarja predvsem z naprednim komisioniranjem in optimizacijo skladiščnih prostorov. Od naših zahtev izpolnjuje zgolj pregled in ažuriranje inventurnih postavk.

Povzetek

Razpredelnica 2.1 prikazuje funkcije, ki jih posamezne aplikacije nudijo. Po analizi konkurence ugotovimo, da bi z implementacijo omenjenih funkcij v Android aplikacijo prišli do konkurenčne prednosti v primerjavi s konkurenti, saj bi zgolj naša rešitev ustrezala vsem zahtevam. To bi lahko bil v velikem številu primerov kar odločilni faktor pri izbiri aplikacije za skladiščno poslovanje.

pod.\oper.	inventura	izdajnica	prevzemnica	povratnica
TronWH	x	x	x	
WMS	x	x	x	
PerfTech	x	x	x	
mVasco	x	x	x	
Logist	x			
naš izdelek	x	x	x	x

Tabela 2.1: Primerjava obstoječih rešitev. Tabela prikazuje, katere od zahtevanih funkcij konkurenčni izdelki ponujajo. Vidimo lahko, da je naš izdelek edini, ki vsebuje vse štiri funkcije.

Poglavje 3

Predlagana rešitev

Naša Android aplikacija bo temeljila na že uveljavljenem poslovnem informacijskem sistemu RISP-SQL, kateremu bo dodana kot mobilna razširitev. Ideja je, da si mobilna aplikacija s poslovnim informacijskim sistemom deli bazo (torej bo iste zapise mogoče urejati z Android aplikacijo in z RISP-om). Ostale veje razvoja so ločene. Prav tako bo naša aplikacija uporabljala RISP-SQL-ove podatke o naročilih kupca, naročilih materiala in podatkih drugih dokumentov in pregledov, na podlagi katerih se bodo izračunale in izpolnile informacije, potrebne za našo aplikacijo. S tem naša aplikacija pridobi na konkurenčni prednosti, ki jo predstavlja dobra povezljivost z že obstoječim računalniškim poslovnim informacijskim sistemom in predstavlja obstoječim uporabnikom namiznega programa edino logično izbiro pri iskanju mobilne podpore skladiščnemu poslovanju.

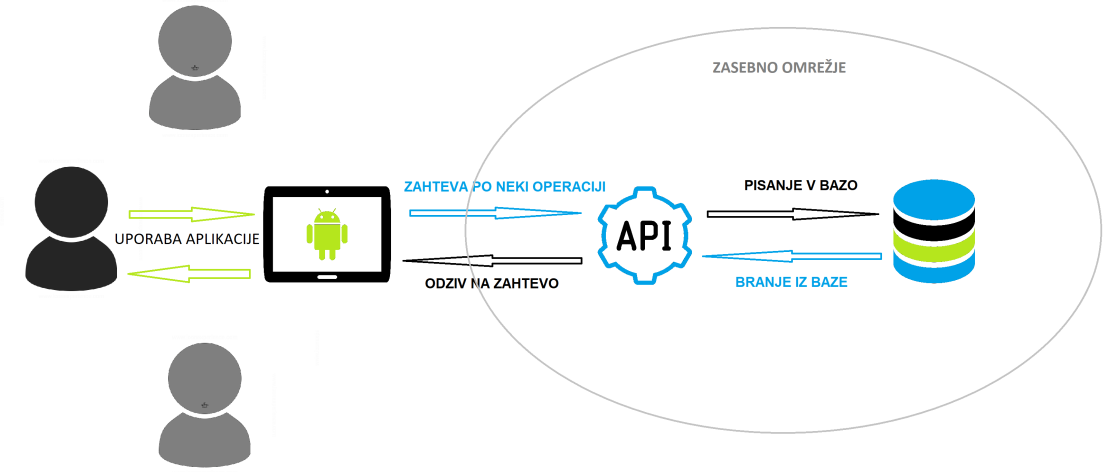
3.1 RISP-SQL

Sistem RISP-SQL ali na kratko RISP je široko zasnovan, integriran poslovni informacijski sistem, ki ga je razvilo podjetje E.R.S. Rokada [18]. Sestavlja ga večje število vsebinsko zaokroženih podsistemov, namenjenih računalniški podpori vsem poslovnim funkcijam, ki se izvajajo v podjetju. Jedro celotnega sistema sicer tudi v sistemu RISP-SQL predstavljajo programski moduli za

področja računovodstva in financ, prodaje, nabave, proizvodnje, spremljanja zalog in preskrbe z materialom. Dopolnjuje jih vrsta manjših tudi zelo uporabnih modulov, kot so evidenca in realizacija sklenjenih pogodb, evidenca danih ali prejetih zadolžitev, evidenca sestankov in sklepov in podobno.

Sistem vključuje tudi možnost uporabe črtne kode, elektronskega arhiva dokumentov in drugih najsodobnejših tehnologij. Modularna zasnova sistema RISP-SQL omogoča nakup posameznega modula ter možnost integracije v obstoječi informacijski sistem podjetja.

3.2 Arhitektura rešitve



Slika 3.1: Grafična ponazoritev arhitekture. V prvi fazi gre za interakcijo uporabnika (ali skupine uporabnikov – sive figure) z Android napravo, na kateri teče naša aplikacija. Za vse, kar uporabnik počne (poizveduje, vnaša, ureja), uporablja Android napravo. Ta je preko mobilnih podatkov povezana v zasebno omrežje, na katerem sta API in baza. Oba sta povezana z ožičeno povezavo. Do API-ja dostopa aplikacija s HTTP zahtevami, ta pa ji v obliki JSON zapisov vrača rezultate poizvedbe oziroma rezultate uspešnosti opravljenih operacij. API je neposredno povezan z bazo. Glede na parametre, ki mu jih Android aplikacija poda preko HTTP zahtev, izvede operacije na bazi (poizvedba, dodajanje oziroma urejanje zapisov) ter Android aplikaciji vrne rezultate.

Pred procesom programiranja je potrebno definirati arhitekturo rešitve. Naša iztočnica je mobilna aplikacija, ki bo preko Android naprave zajela vhodne podatke, jih posredovala bazi ter tam procesirala zahteve. Ker pa Android naprava ni neposredno povezljiva z bazo, je potrebno rešitvi dodati spletni

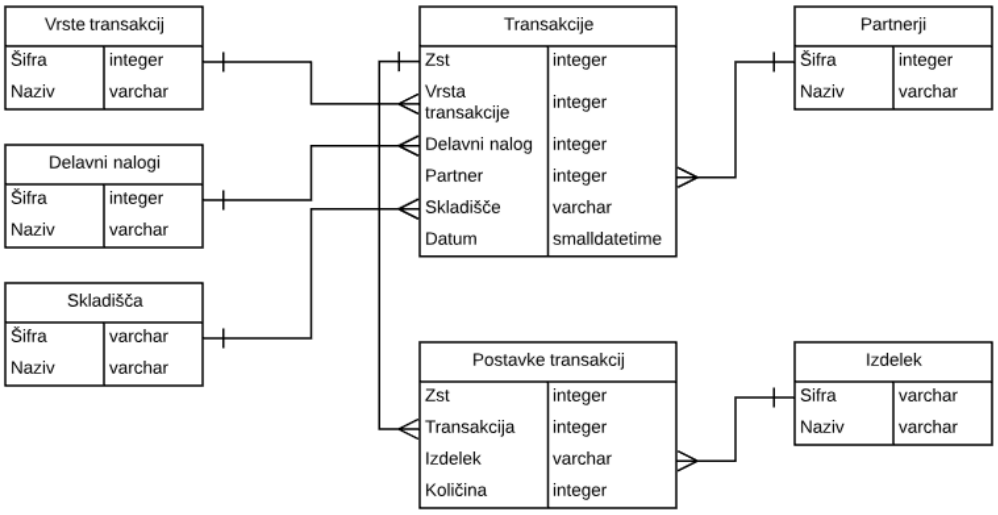
API (Application Programing Interface) [2], ki omogoči povezljivost z bazo.

API je programski vmesnik, ki uporabniku omogoča dostop do podatkov iz baze, operacijskega sistema in drugih podobnih storitev. Dostop do API-ja bo urejen preko navideznega zasebnega omrežja VPN (Virtual Private Network) ter protokola HTTP. VPN nam omogoča, da se preko VPN vmesnika ne glede na lokacijo nahajanja neposredno povežemo z zasebnim omrežjem, kjer se API nahaja. To nam precej olajša problem varnosti aplikacije, saj morebitni napadalec ne bo imel dostopa do podatkov, ne da bi se pred tem povezal z zasebnim omrežjem, ki pa je zaščiteno z drugimi protokoli. Za te protokole skrbi stranka, tako da se nam z njimi ni potrebno ukvarjati. Ob sprejemu podatkov API na podlagi zahtevanega krnilnika obravnava HTTP zahteve, jih procesira v bazi ter vrača zapise iz baze, če so ti zahtevani. Podrobnejša razlaga implementacije bo razložena v poglavju 4.

3.3 Organizacija entitet baze

Definicije tabel in odnosi med njimi so ključni za razumevanje nadaljnjega programiranja. V tem podpoglavju smo z entitetnimi diagrami predstavili tabele, potrebne za našo aplikacijo in povezave med njimi. Polja, ki niso povezana z drugimi polji se vnašajo skozi aplikacijo ali pa se dinamično izračunajo iz drugih tabel, ki pa jih zaradi boljše preglednosti rešitve nismo vključili v diagrame.

Slika 3.2: Diagram entitet in tabel, uporabljenih pri kreiranju inventure.



Slika 3.3: Diagram entitet in tabel, uporabljenih pri kreiranju izdajnice, prevzemnice in povratnice. Vse tri funkcije uporabljajo enake tabele. Glavna razlika je v načinu vnašanja ter različnosti podatkov. Glede na vrsto transakcije se polja na primer drugače obnašajo v nadaljevanju. Prevzem bo na primer vnesena polja spreminjal v pozitivno vrednost, izdaja pa v negativno. Tabela transakcij vsebuje zapise o glavah transakcij, kamor spadajo osnovni podatki o transakciji. Tabela postavk transakcij predstavlja izdelke, ki so vezani na transakcijo iz tabele transakcij ter njihovo zalogo.

3.4 Uporabljena orodja in tehnologije

Pri izdelavi naše aplikacije, smo uporabili spodaj navedena orodja in tehnologije.

3.4.1 Android

Android je mobilni operacijski sistem, ki ga je Google predstavil leta 2007. Zgrajen je na Linuxovem jedru in drugih odprtokodnih orodjih. Android

je primarno mišljen za mobilne telefone na dotik in tablice. Kljub temu se uporablja tudi drugje (ure, namizni računalniki). Prednosti Androida so predvsem v tem, da gre za odprtokodni operacijski sistem, kar omogoča razvijalcem cenejše in hitrejše razvijanje aplikacij. Izdelovalcem pametnih telefonov in tablic olajša delo, saj jim ni potrebno več razvijati lastnih operacijskih sistemov, pač pa se lahko osredotočijo zgolj na nekaj komponent operacijskega sistema.

Prva verzija Androida je izšla 9. februarja 2009. Od takrat so nove verzije izhajale približno na eno leto. Zadnja verzija je 8.1 Oreo, ki je izšla 5. decembra 2017.

3.4.2 Java

Java je objektno usmerjeni programski jezik, ki ga je leta 1991 razvil James Gosling v podjetju Suns Microsystems, ki si ga danes lasti podjetje Oracle Corporation. Razvit je bil kot zamenjava za C++, s katerim imata tudi podobno sintaktično strukturo. Programira se v načinu WORA (Write Once, Run Anywhere), kar pomeni, da se napisan program prevede enkrat, nato pa se izvaja na poljubnih napravah brez potrebe po ponovnem prevajanju programa. Java se je v zadnjih dvajsetih letih zelo razširila v računalniškem svetu in je opremljena z bogato standardno knjižnico struktur in funkcij [17].

Java je med drugim tudi glavni jezik pri razvoju aplikacij za Android. Hkrati je tudi večji del njegovega operacijskega sistema skupaj s programskimi vmesniki napisan v Javi.

3.4.3 Android Studio

Android Studio je uradno razvojno okolje za razvoj aplikacij v Androidu. Okolje je posebej za razvoj v Androidu 16. maja 2013 razvil Google na podlagi razvojnega okolja IntellJ, podjetja JetBrains. Tako kot za Android, tudi za Android Studio prihajajo redno nove verzije. Trenutna zadnja stabilna verzija je 3.0.1, ki je izšla 20. novembra 2017. Okolje je brezplačno in

uporabniku ponuja številne predloge in funkcije, ki olajšujejo razvoj aplikacije. Prav tako nam Android Studio ponuja emulator, ki simulira delovanje mobilne naprave in uporabniku omogoča, da svojo aplikacijo testira kar na namiznem računalniku brez potrebe po povezovanju naprave, ki jo poganja Androidov operacijski sistem.

3.4.4 SQL

SQL (Structured Query Language) je strukturirani jezik za povpraševanje za delo s podatkovnimi bazami. Uporablja se ga za iskanje, ustvarjanje in upravljanje podatkov, ki so običajno zbrani v organiziranih tabelah. SQL je določen z ANSI/ISO standardom. SQL standard se je začel razvijati že leta 1986, tako da imamo danes več verzij. Gre za relativno enostaven jezik za učenje. Njegovo znanje danes predstavlja nepogrešljivo vrlino vse več poklicev.

3.4.5 MS SQL Server

Microsoft SQL Server je sistem za upravljanje z relacijskimi bazami. Sistem je bil prvič predstavljen leta 1989. Od takrat se je zvrstilo več različnih verzij. Zadnja stabilna je bila izdana 2. oktobra 2017. Mi smo uporabljali verzijo 2014. Microsoft SQL Server deluje po sistemu odjemalec-strežnik, kar pomeni, da odjemalec strežniku pošlje zahtevo, nato pa mu ta vrne odgovor.

Jedro sistema je SQL Server Database Engine, ki nadzoruje shranjevanje podatkov, obdelavo in varnost. Ta teče na operacijskem sistemu SQL Server oz. SQLOS, ki skrbi za nižje-nivojske naloge, kot so upravljanje pomnilnika, nadzor vzhodno-izhodnih podatkov, razporejanje mest in zaklepanje podatkov tam, kjer do urejanja ne sme prihajati.

3.4.6 Visual Studio

Visual Studio je integrirano razvojno okolje (IDE) podjetja Microsoft. Okolje se uporablja za razvoj spletnih strani, iger, namiznih in mobilnih aplikacij

in številnih drugih programov. Visual Studio izvira iz leta 1997. V času je podobno kot ostali Microsoft programi doživel številne posodobitve in nove različice. Zadnja je bila 20. februarja 2018. Visual Studio omogoča urejanje in razhroščevanje kode za številne programske jezike. Nekateri so vgrajeni že v samo okolje, druge pa lahko uporabnik doda s pomočjo vtičnikov. Kljub temu sta najpogosteje uporabljena jezika C# in C++.

3.4.7 C#

C# je objektno orientiran programski jezik, ki ga je leta 2000 predstavil Microsoft. Jezik temelji na programskih jezikih C, C++ in Visual Basic. Tako kot Java, tudi C# uporablja način programiranja WORA. Ima pa C# v primerjavi s svojimi "predhodniki" kar nekaj prednosti. Tako recimo nudi dodatne funkcije, kot so vrste ničelnih vrednosti, enumeracije, delegati, izrazi lambda in neposredni dostop do pomnilnika. Prav tako C# podpira generične metode in vrste, ki omogočajo večjo varnost in učinkovitost. Poleg tega je C# dobro integriran z ostalimi Microsoftovimi orodji. C# je bil primarno sicer izdelan za razvoj .NET programov in delo z .NET platformo. Gre za knjižnico, ki je sestavni del operacijskega sistema Windows in predstavlja tudi ogrodje .NET orodjem ter številnim aplikacijam na različnih platformah.

3.4.8 Fiddler

Fiddler je brezplačni razhroščevalnik namestniškega strežnika HTTP. Leta 2003 ga je predstavil nekdanji programer na Microsoftu Eric Lawrence. Fiddler omogoča zajemanje vsega prometa med odjemalcem in strežnikom. Prav tako Fiddler omogoča urejanje HTTP prometa in s tem dovoljuje uporabniku, da lažje odpravlja napake. Tako kot pri številnih zgoraj omenjenih programih tudi Fiddler prejema posodobitve. Zadnja stabilna je bila narejena 23. decembra 2015.

3.4.9 IIS

IIS (Internet Information Services) je paleta storitev za strežnike z operacijskim sistemom Windows, ki jo je leta 1995 predstavil Microsoft. Trenutno podpira strežniške storitve s protokoli FTP, SMTP, NNTP ter HTTP/HTTPS. IIS, ki je napisan v C++, je bil najprej izdelan zgolj kot dodatek internetnim procesom za operacijski sistem Windows. Sčasoma je preko novejših verzij dobil še številne druge funkcionalnosti. Zadnja verzija je IIS 10.0 iz 29. julija 2015. IIS ima modularno arhitekturo. Uporablja core web server engine, kar pomeni, da se izvajajo zgolj procesi, ki jih zahtevamo. Prav tako omogoča tudi dodajanje funkcionalnosti preko uporabe različnih modulov. Ti so lahko napisani v kateremkoli .NET Frameworku jeziku. Trenutno so na voljo naslednji moduli: HTTP moduli, varnostni moduli, vsebinski moduli, pomnilniški moduli, dnevniški in diagnostični moduli. Ena od glavnih prednosti IIS-ja je tudi varnost, ki pa se je pojavila šele pri zadnjih verzijah (od 6 dalje).

3.4.10 JSON

JSON (Java Script Object Notation) je odprtokodni format datoteke za shranjevanje in izmenjavo podatkov, ki uporablja človeško berljivo besedilo. JSON se je pojavil kot podmnožica programskega jezika JavaScript leta 1999. JSON temelji na univerzalnih dveh strukturah. Prva je predstavljena kot par med imenom in vrednostjo, druga pa kot urejen seznam vrednosti [5]. Ena od prednosti je tudi ta, da je neodvisen od programskih jezikov, kar pomeni, da ga lahko preprosto izmenjujemo med dvema različnima jezikoma.

3.4.11 Photoshop

Photoshop je profesionalno orodje za obdelavo slik in drugih grafičnih elementov. Razvilo ga je podjetje Adobe Systems leta 1990. Photoshop temelji na bitnih slikah, kar ga loči od številnih drugih obdelovalcev slik, kot je npr. Illustrator, ki temelji na vektorjih. Photoshopov privzet format slik je sicer .PSD, vendar lahko sliko shranimo v večini obstoječih, razširjenih formatov

slik. Prav tako je zelo dobro integriran z ostalimi Adobeovimi izdelki, med katerimi omogoča enostavne izvoze ter uvoze. Photoshop nudi uporabniku širok nabor orodij za obdelavo grafik, zaradi česar je orodje postalo najbolj uporabljen program v svoji kategoriji. Adobe Systems tudi vsako leto izda novo verzijo programa. Zadnja različica se je na tržišču pojavila 14. februarja 2018 pod imenom CC 2018.

Poglavje 4

Implementacija

4.1 Implementacija API-ja

API je sestavljen iz različnih metod in protokolov ter orodij za gradnjo aplikacij. V svetu sta trenutno najbolj razširjeni metodi SOAP in REST. SOAP je starejša in bolj kompleksna izbira. Prednosti SOAP-a so standardiziranost ter neodvisnost od jezika in platforme. REST je novejša rešitev in je bil razvit z namenom, da odpravi pomanjkljivosti SOAP-a. REST je manj kompleksna izbira in je tudi lažje učljiv. Glavne prednosti REST-a so hitrost in učinkovitost, saj podpira večji nabor podprtih podatkovnih formatov. SOAP podpira zgolj XML.

Zaradi hitrejše odzivnosti in rabe manj časovne širine, smo se odločili za REST. V REST-ovem naboru je tudi format JSON, ki nudi boljšo podporo brskalnikom ter hitrejše razčlenjevanje podatkov. Zato smo se odločili, da omenjeni format uporabimo tudi pri implementaciji naše rešitve.

Do REST API-ja dostopamo s protokolom HTTP. To pomeni, da do njega lahko dostopa vsaka naprava z omogočenim brskanjem po spletu. Sem spada tudi Android telefon oziroma v našem primeru tablica. REST deluje po sistemu odjemalec in strežnik. Odjemalec pošlje strežniku zahtevo, strežnik pa mu vrne procesiran odgovor. Pri tem uporabljamo protokol HTTP po metodi CRUD – Create (ustvari), Read (beri), Update (posodobi) in Delete

(briši) [4]. HTTP metode imajo imena:

- GET - omogočal nam bo vračanje zapisov iz baze,
- POST - omogočal nam bo vnašanje zapisov v bazo,
- PUT - omogočal nam bo urejanje zapisov na bazi,
- DELETE - omogočal nam bo brisanje zapisov iz baze.

Pri programiranju so te metode vsebovane v datotekah, imenovanih krmilniki (v žargonu "kontrolerji"). API vsebuje več krmilnikov. Vsaka tabela v bazi, ki je namenjena pregledovanju oziroma ažuriranju, ima svoj pripadajoči krmilnik v API-ju, do katerega dostopamo s HTTP zahtevo.

Za programiranje API-ja smo izbrali orodje Visual Studio ter jezik C#, ki je z orodjem najbolje integriran. Visual Studio omogoča enostavno pisanje API-jev. Poleg tega je enostavno povezljiv z IIS-jem, ki ga bomo kasneje uporabili za objavo API-ja. Pri pisanju krmilnikov smo uporabili knjižnico CLR-Belgrade-SqlClient [20], ki omogoča programerju neposredno pisanje SQL stavkov v kodo krmilnika. Brez uporabe knjižnice to ni možno in bi rešitev implementirali s pomočjo procedur, spisanih v bazi. To nam omogoča, da jedra SQL operacij, ki jih bomo počeli nad bazo, "hardcodiramo"v krmilnikih. Vrednosti za filtriranje oziroma polja za vnos podamo kot parametre, ki nato dopolnijo "hardcodiran"del SQL stavka. Seveda so stavki vidni zgolj programerju v času pisanja kode. Ko se API objavi na strežnik, SQL stavkov ni možno več spreminjati in so možne izvedbe zgolj prej definiranih SQL ukazov. Programiranja smo se lotili na podlagi rešitve, predstavljene v članku "Building REST services with ASP.NET Core Web API and Azure SQL Database"[7].

Kljub neposrednemu pisanju SQL stavkov v API nam mora baza vračati rezultate v formatu JSON, saj bomo iz tega formata delali razčlembo podatkov. MS SQL 2016 in novejši imajo sicer poseben ukaz za to, ki pa nam ni na voljo, saj razpolagamo z MS SQL 2008, ki te funkcije ne podpira. SQL stavek

je potrebno torej prirediti tako, da bo vrnjen rezultat string, ki predstavlja zapis v formatu JSON (glej Sliko 4.1).

```
[HttpGet("{id}")]
public async Task Get(string id)
{

    await SqlPipe.Stream(" select '{\"GlavaIzdaje\":[,' + STUFF(( " +
                         " SELECT " +
                         "    '{\"stevilka\":\"' + cast(stevilka as varchar) + '\"' " +
                         "    + ',\"opis\":\"' + cast(opis as varchar) + '\"' " +
                         "    + ',\"kupec\":\"' + isnull(cast(kupec as varchar),'') + '\"' " +
                         "    + ',\"NAZIV1\":\"' + isnull(cast(NAZIV1 as varchar),'') + '\"' " +
                         "    + '}' " +
                         " from pronaln dn " +
                         " left join partztp kup on kup.MATST = dn.kupec " +
                         " where stevilka = '" + id+ "' " +
                         " for xml path(''), type " +
                         " ).value('.', 'varchar(max)'), 1, 1, '') +']}' ",
    Response.Body, "[]");

}
```

Slika 4.1: Delček kode krmilnika pri programiranju API-ja, kjer vračamo rezultate v JSON formatu

Pri metodah POST, PUT in DELETE ta prirejanja niso potrebna, saj ne vračamo nobenih rezultatov, pač pa zgolj izvedemo želeni proces na bazi. Potrebno je dodati objekte, preko katerih se bodo pošiljali dinamični parametri SQL stavka. Metoda DELETE tega objekta ne potrebuje, saj v večini primerov brisanje zapisa na bazi potrebuje zgolj en dinamičen parameter, ki je največkrat zaporedna identifikacijska številka zapisa. Tega lahko podamo kar preko URL naslova. Metodi POST in PUT potrebujeta več dinamičnih parametrov, saj pri dodajanju ter urejanju zapisov potrebujemo poleg identifikacijske številke še dejanske vrednosti atributov, ki jih bo nov zapis imel oziroma jih bo star zapis dobil na novo. Pošiljanje več parametrov preko URL naslova bi postalo zelo kaotično. Temu se izognemo tako, da API-ju dodamo objekte za želene postavke (glej Sliko 4.2). Te vsebujejo polja, ki jih bomo dinamično zapolnjevali preko klica HTTP ter se na ta polja sklicevali pri ustvarjanju SQL stavka.

```
using Newtonsoft.Json;

namespace RVK
{
    public class InventurnaPostavka
    {
        [JsonProperty("skl")]
        public string Skl { get; set; }

        [JsonProperty("vt")]
        public string Vt { get; set; }

        [JsonProperty("stdok")]
        public string Stdok { get; set; }

        [JsonProperty("sifart")]
        public string Sifart { get; set; }

        [JsonProperty("datum")]
        public string Datum { get; set; }

        [JsonProperty("invkoli")]
        public string Invkoli { get; set; }

        [JsonProperty("uporabnik")]
        public string Uporabnik { get; set; }
    }
}
```

Slika 4.2: Objekt v C#

Klic HTTP namreč omogoča, da mu podamo določene atribute kot request body. Request body uporabimo za prenos zapolnjenih objektov; vrednosti teh vstavljamo v nekonstantne dele SQL stavka (glej Sliko 4.3).

Opisovanja posameznih krmilnikov ne bomo obravnavali, saj gre pri vseh za identično strukturo. Vsak krmilnik ima pripadajočo tabelo. Vsebuje štiri metode (GET, POST, PUT, DELETE), ki se izvršujejo nad njo. Krmilnik tako vsebuje SQL ukaz za poizvedbo, ustvarjanje, popravljanje ter brisanje zapisov v bazi.

Ko so nastavitve na bazo narejene ter imamo dokončane krmilnike in

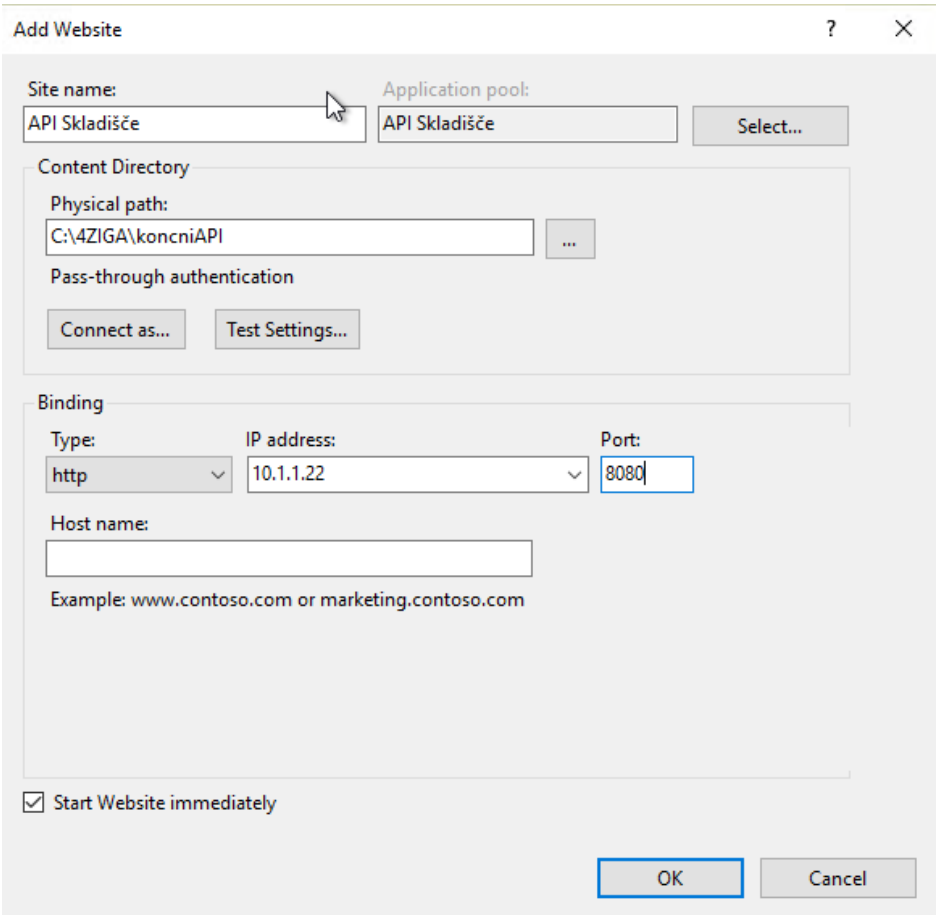
```
[HttpPut("{id}")]
public async Task Put(string id, [FromBody] InventurnaPostavka inventurnaPostavka)
{
    string skl = inventurnaPostavka.Skl;
    string vt = inventurnaPostavka.Vt;
    string stdok = inventurnaPostavka.Stdok;
    string sifart = inventurnaPostavka.Sifart;
    string datum = inventurnaPostavka.Datum;
    string invkoli = inventurnaPostavka.Invkoli;

    var cmd = new SqlCommand(@"UPDATE InventurDnevnik " +
                             "SET SKL = '" + skl + "', " +
                             "VT = '" + vt + "', " +
                             "STDOK = '" + stdok + "', " +
                             "SIFART = '" + sifart + "', " +
                             "DATUM = '" + datum + "', " +
                             "INVKOLI = '" + invkoli + "' " +
                             "WHERE ZST = @id");
    cmd.Parameters.AddWithValue("id", id);
    await SqlCommand.ExecuteNonQuery(cmd);
}
```

Slika 4.3: Delček kode krmilnika pri programiranju API-ja, kjer se podatki iz request bodya zapišejo v sql stavek

objekte, je potrebno API objaviti tako, da bo dostopen ves čas z različnih naprav. Pri tem je potrebno izvesti tudi veliko testiranja, ki je opisano v poglavju.

Kot smo omenili že zgoraj, za to uporabimo orodje Internet Information Services (IIS). Ker sta Visual Studio, v katerem smo razvijali API, in IIS produkta Microsofta, sta med seboj zelo dobro usklajena, kar močno poenostavi povezljivost. Visual Studio ima možnost, da API zapišemo v mapo, kjer so datoteke zapisane in prilagojene posebej za IIS. V IIS-ju nato odpremo novo stran in ji nastavimo fizično pot do mape, ki smo jo prej ustvarili preko Visual Studia. Nato je potrebno nastaviti še application pool (skupino aplikacij), kjer se bo API nahajal. Ta omogoča, da svoje aplikacije (v našem primeru API) ločimo od drugih ne glede na to, da te delujejo na istem strežniku. Prav nam pride v primeru večnivojske varnosti različnih aplikacij. Nastaviti je po-

trebno tudi ime aplikacije ter IP naslov ter port (vrata) za dostop do API-ja. To nam precej poenostavi dejstvo, da se bodo uporabniki k nam povezovali preko VPN-ja vmesnika. Implementacija dodatnih varnostnih ukrepov tako ni potrebna, saj bo uporabnik povezan neposredno v zasebno omrežje in bo overjanje opravljeno že v tem koraku. S tem se izognemo tudi potrebi odpiranja zunanjih portov, saj je za nas pomembno zgolj, da je API viden v zasebnem omrežju. Pri portu moramo paziti zgolj na to, da uporabimo takega, na katerega še ni vezana nobena druga aplikacija oziroma stran. V nasprotnem primeru namreč prihaja do konfliktov pri povezovanju z aplikacijo.



Slika 4.4: Nastavljanje strani v IIS-ju

Končani API na koncu prekopiramo in mu spremenimo povezavo s testno bazo, tako da imamo dva identična API-ja z različnimi povezavami z bazami. To nam bo prišlo prav v nadaljevanju pri testiranju Android aplikacije, ko se bo aplikacija najprej izvajala na testni bazi, da bi preprečili brisanja oziroma napačne vnose v produkcijsko bazo. Sedaj je vse pripravljeno, da začnemo programirati v Androidu.

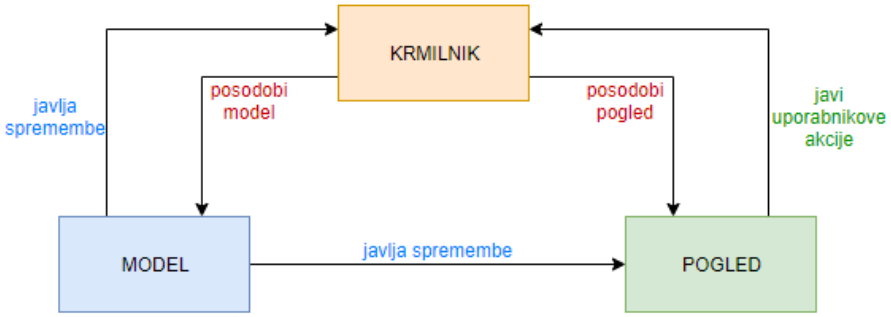
4.2 Android aplikacija

4.2.1 Iterativni razvoj

Pri naši Android aplikaciji smo uporabili iterativni razvoj. Pri razvoju smo sodelovali z naročnikom in na podlagi njegovega odziva in predlogov oblikovali uporabniški vmesnik ter prilagajali funkcionalnost. Tak način dela zahteva večjo prilagodljivost ter več pogovarjanja in usklajevanja, vendar pa pripomore k boljši uporabniški izkušnji in večjemu zadovoljstvu naročnika.

4.2.2 Organiziranost aplikacije

Prvi korak pri ustvarjanju Android aplikacije predstavlja določitev organizacije. Odločili smo se za model MVC (Model View Controller) [3]. Model v aplikaciji predstavlja objekt. Lahko tudi vsebuje logiko za posodobitev krmilnika, če se podatki spremenijo. View (pogled) predstavlja vizualizacijo vmesnika in prikazuje vidni del aplikacije. Controller (krmilnik) deluje z obema komponentama (pogledom in krmilnikom) ter usmerja tok podatkov. MVC smo pred ostalimi ureditvami izbrali zaradi njegove preprostosti, ki pa hkrati ustreza zahtevam naše aplikacije.



Slika 4.5: Arhitektura MVC [6]. Krmilnik skrbi za interakcijo z uporabnikom, kamor spadata branje podatkov s pogleda in pošiljanje podatkov modelu ter pogledu. Model skrbi za branje in shranjevanje podatkov v povezavi z bazo. Spremembe javlja krmilniku in pogledu. Pogled predstavlja uporabniku viden del aplikacije; prikazuje podatke iz modela.

4.2.3 Aktivnost [9]

Pri gradnji Android aplikacije imamo razrede, ki predstavljajo javanske datoteke. Razredi, ki jih pišemo mi, običajno že vključujejo obstoječe gradnike Android aplikacij. Mi jim dodajamo dodatno funkcionalnost. Najpogosteje bomo mi uporabljali razširitve gradnika Aktivnost oziroma Activity (na Sliki 4.6 lahko vidimo primer vključevanja oz razširitve že obstoječega razreda AppCompatActivity).

Razred Aktivnost (Activity) je osnovna izvedbena enota v Android aplikaciji in predstavlja del uporabniškega vmesnika (prikaz na zaslonu). Za Aktivnost je značilno, da ima svoj življenjski cikel. V življenjskem ciklu aktivnosti se kličejo tako imenovane "call-back" metode, med katerimi so najbolj pogoste: onCreate(), onDestroy(), onPause(), onResume(). Te metode so neposredno povezane z uporabo aplikacije. Ko se v aplikaciji pokliče poljubno aktivnost in se ta ustvari, izvede funkcijo onCreate. Ko se ta uniči,

```
import ...

public class MeniActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {...}

    public void init(){...}
}
```

Slika 4.6: Razširitev gradnika Activity

se izvede funkcija onDestroy(). Podobno velja tudi za onPause() in onResume(). Aktivnosti se lahko tudi kličejo med seboj (kar se bo v naši aplikacije zaradi večjega števila strani kar pogosto dogajalo).

Pomembna datoteka za nas je tudi AndroidManifest.xml, ki vsebuje vse deklaracije gradnikov Android aplikacije. Brez teh deklaracij Aktivnosti niso uporabne. AndroidManifest vsebuje tudi vse ključne informacije o aplikaciji, vključno s pravicami in zunanjimi knjižnicami.

4.2.4 Fragment[10]

Fragment si lahko predstavljamo kot neko "pod-aktivnost". Uporabljali ga bomo za zavihke znotraj strani. Fragment mora imeti vedno svojo gostiteljsko Aktivnost. V eni Aktivnosti se lahko pojavi več Fragmentov. Isti Fragment se lahko pojavi na več Aktivnostih. Predstavljamo si jih lahko kot Aktivnosti znotraj Aktivnosti. Fragment ima prav tako kot Aktivnost svoj življenjski cikel, vendar je ta pogojen z življenjskim ciklom gostiteljske Aktivnosti. Ko se konča Aktivnost, se končajo tudi vsi njeni Fragmenti.

4.2.5 AsyncTask[11]

Razred AsyncTask omogoča asihrono delovanje nalog v ozadju, ne da bi pri tem motili glavno nit programa. Namenjen je izvajanju krajših operacij, ki

trajajo največ nekaj sekund. AsyncTask ima štiri korake. To so:

- onPreExecute: sproži se ob klicu AsyncTaska. Običajno je namenjen nastavitvi naloge.
- doInBackground: sproži se ob zaključku onPreExecute. V njem se običajno izvaja glavna procedura razreda in tudi traja največ časa.
- onPostExecute: sproži se ob zaključku doInBackground. Rezultate glavne procedure dobi podane kot parameter.

V naši aplikaciji smo AsyncTask največkrat uporabili pri nalaganju podatkov v sezname postavk (ListView) [8] (in tudi pri posodabljanju in dodajanju novih vrednosti v bazo). Povezovanje preko API-ja do baze in nalaganje polj lahko namreč vzame nekaj časa in bi ob izvajanju znotraj metode v aktivnosti lahko slabo vplivalo na odzivnost aplikacije. AsyncTask nam omogoči, da se to zgodi v ozadju, tako da ne vpliva na odzivnost aplikacije. V prvem koraku smo zgolj zapisali sporočilo, da se je proces začel; v drugem smo izvedli klic API-ja in rezultate zapisali v javansko tabelo; v tretjem koraku smo vrednosti iz tabele shranili v naš ListView. Za to je bila potrebna uporaba Adapterja. Adapter predstavlja povezavo med podatki v tabeli in ListViewom.

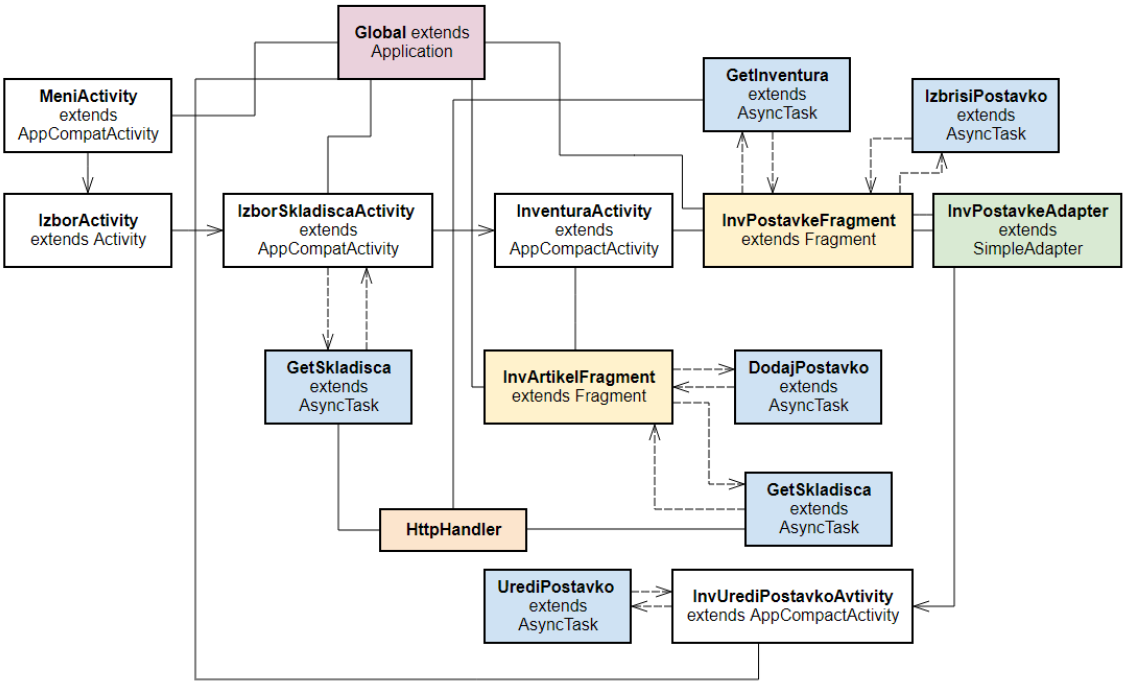
4.2.6 Postavitev

Medtem ko v Aktivnostih in Fragmentih pišemo kodo, kaj se bo zgodilo ob določenem vnosu uporabnika, potrebujemo tudi datoteke, kjer bo zapisano, kako naše strani sploh izgledajo. Zato imamo Postavitev oziroma Layout. Layout predstavljajo XML datoteke, v katerih so zapisane postavitve strani. Vsebuje komponente, kot so poravnave, gumbi, tekstovna polja, polja za vnos, slike in tako naprej. Vsaka Aktivnost oziroma Fragment ima svojo pripadajočo XML datoteko za postavitev strani. Lahko pa se XML datoteke ponovijo na več Aktivnostih oziroma Fragmentih. To se zgodi v primeru, če

gre za enako obliko strani. Pri nas se bo to zgodilo pri prikazovanju seznamov iz baz.

4.2.7 UML diagram aplikacije

Delovanje aplikacije iz programerskega vidika bomo predstavili s pomočjo UML diagrama. Ker pa je osnovni princip podoben pri vseh štirih osrednjih funkcijah programa (inventura, izdaja, prevzem in povračilo), smo se odločili, da predstavimo zgolj UML diagram razredov pri poteku izvajanja inventure.

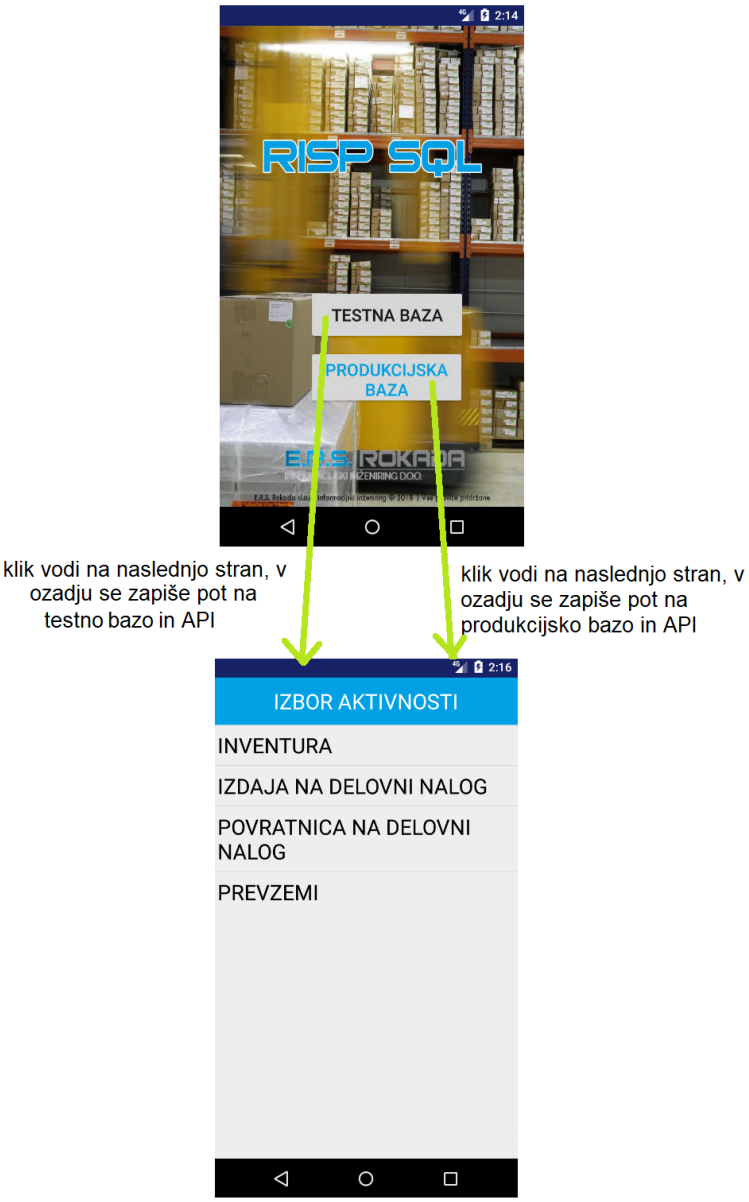


Slika 4.7: UML diagram pri izvajanju inventure. Vsak od razredov, ki predstavlja razširitev razredov Activity ali Fragment, ima tudi svojo pripadajočo oblikovno XML datoteko. Zaradi preglednosti te na diagram niso bile dodane. Global je razširitev razreda Application in sodeluje s skoraj vsemi aktivnostmi. Vsebuje namreč vse globalne spremenljivke ter metode za njihovo branje oziroma shranjevanje. MeniActivity predstavlja našo vhodno aktivnost. Ta kliče aktivnost IzborActivity, kjer lahko izberemo možnost za izvajanje inventure. Ta kliče aktivnost IzborSkladiscaActivity, ki v ozadju s klicem AsycTask-a GetSkladisca naredi HTTP zahtevo do API-ja in prejete vrednosti napolni v spustni meni za izbor skladišča. Vsi razredi, ki so

razširitev razreda AsyncTask in se uporabljajo za zajemanje JSON vrednosti iz API-ja, uporabljajo pri tem razred HttpHandler. Izbrana vrednost skladišča se ob prehodu na naslednjo aktivnost zapiše v razred Global. Naslednja aktivnost je InventuraActivity, ki vsebuje dva fragmenta. Prvi – InvArtikelFragment je namenjen vnašanju izdelkov in ob vnosu šifre izdelka v ozadju izvede AsyncTask GetPostavka. Ta naredi klic API-ja in dobljene vrednosti zapiše v pripadajočo XML datoteko razreda InvArtikelFragment. Ob kliku na gumb za dodajanje se nato izvrši AsycnTask DodajPostavko, ki naredi HTTP zahtevo na API, ki v bazo doda nov zapis. Drugi zavihek – InvPostavkeFragment predstavlja seznam že obstoječih postavk iz baze. Te ob metodi OnCreate() napolni AsyncTask GetPostavke. Pri tem uporablja tudi razred InvPostavkeAdapter, ki je razširitev razreda SimpleAdapter. Fragment InvPostavkeFragment omogoča tudi funkcijo brisanja postavke, za kar uporablja AsyncTask BrisiPostavko. Ta naredi HTTP zahtevo na API, ki v bazi postavko izbriše. Za urejanje postavke se ustvari nova aktivnost InvUrediPostavko, ki iz razreda Global prebere trenutne podatke za prikaz. Ob potrditvi se izvede AsyncTask UrediPostavko, ki naredi HTTP zahtevo na API, ki posodobi podatke v bazi.

4.2.8 Scenariji uporabe aplikacije

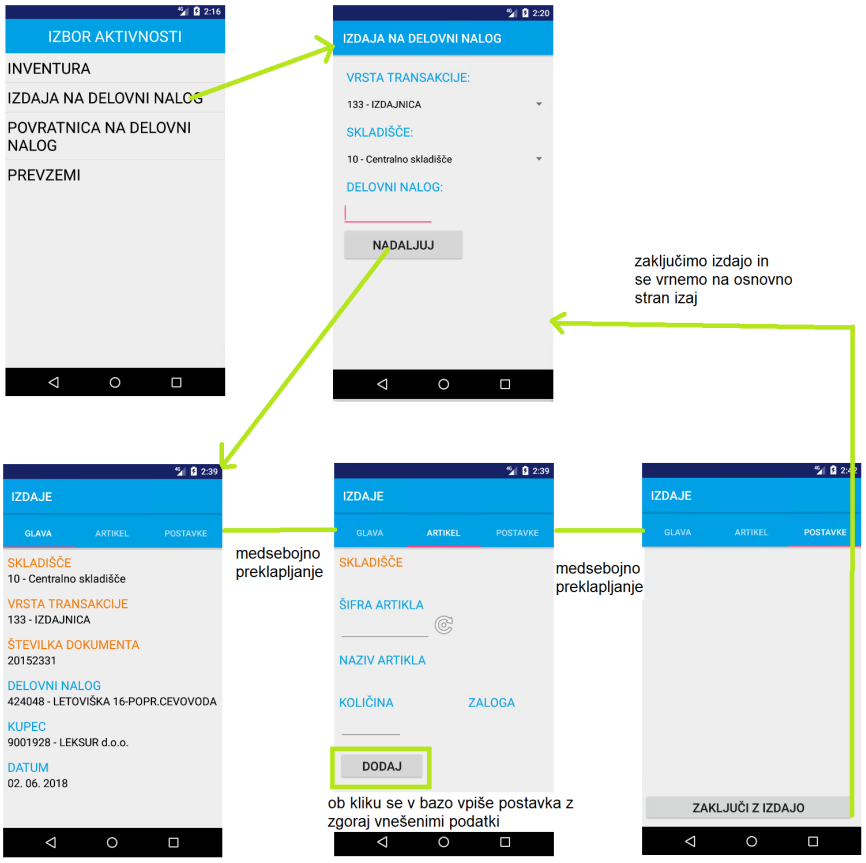
V tem podpoglavju z diagrami poteka ponazorimo delovanje aplikacije oziroma pomikanje po njenih straneh. Prikaz vsebuje pet diagramov. Prvi predstavlja vstopno stran in navigacijo do menija. Ostali štirje predstavljajo vsak po eno vrsto operacije – inventuro, izdajo, povračilo ter prevzem.



Slika 4.8: Vstopna stran in meni operacij. Na prvi strani imamo gumba za nastavitev baze. Da imamo testno in produkcijsko bazo, je pomembno, saj uporabnik želi aplikacijo nekaj časa testirati na testnih podatkih, da ob morebitnih napakah ne pride do brisanja oziroma napačnega vnašanja v produkcijsko bazo. Baza se namreč uporablja tudi v informacijskem poslovnem sistemu RISP. Funkcije aplikacije se v nadaljevanju glede na izbor baze ne spreminjajo. Razlika je zgolj v povezavi z API. Ob gumbu se poleg nastavitve baze odpre še menu, kjer lahko izbiramo med implementiranimi operacijami (inventura, izdaja, povratnica, prevzemi).

* - na zavihku VNOS se ob vnesu šifre avtomatsko naloži še naziv artikla
** - na zavihku PREGLED se ob podrsu navzdol podatki osvežijo. Upošteva se tudi filter za skladišče in šifro

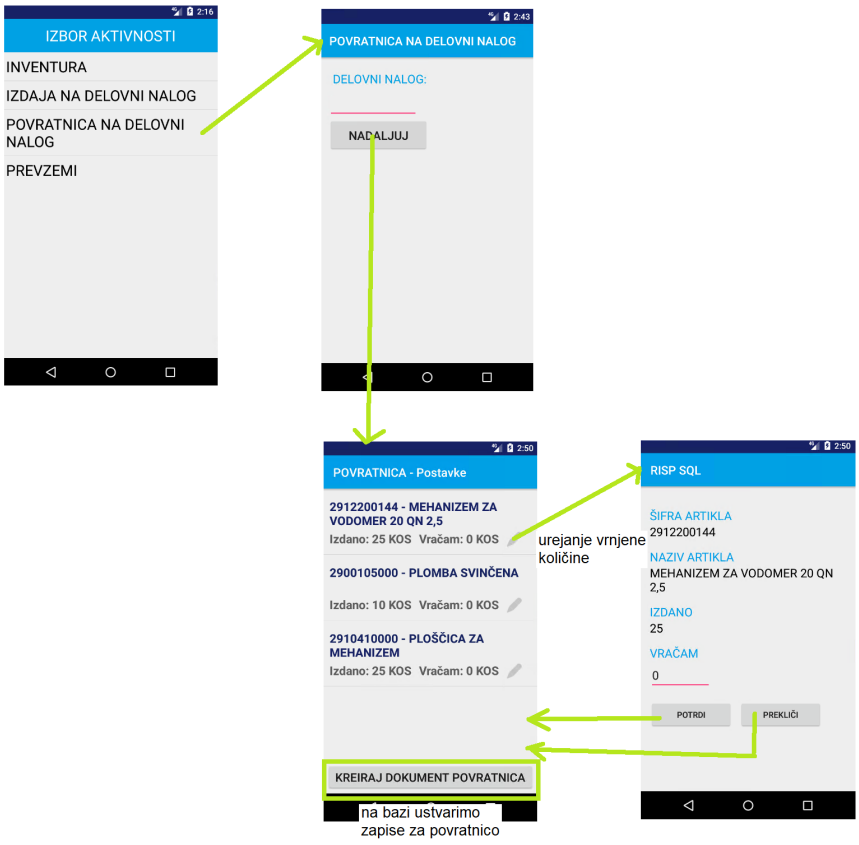
Slika 4.9: Diagram izvajanja inventure. Ob izboru inventure se odpre stran za vnos skladišča, v katerem bomo inventuro izvajali. Ob kreaciji strani se v ozadju izvede klic API-ja; vrnjeni rezultati se napolnijo v spustni seznam s skladišči iz baze. Ob kliku na gumb NADALJUJ se izbrano skladišče shrani v globalno spremenljivko. Odpre se stran z inventurami, ki vsebuje dva zavihka. Prvi služi vnašanju novih izdelkov, drugi pa za pregled le-teh. Pri vnosu uporabnik poskenira kodo izdelka, nato pa se v ozadju izvede poizvedba na bazo o nazivu izdelka, ki se nato zapiše na zaslon. Poleg polja šifra je tu še gumb za osveževanja, ki omogoča uporabniku, da sproži poizvedbo tudi ročno. Za tem vnese količino izbranega izdelka in ob gumbu vnos se podatki zapišejo v bazo. Pri zavihku "Pregled" imamo seznam vseh vnesenih postavk. Seznam se osvežuje ob podrsu navzdol. Nad seznamom ima uporabnik možnost filtriranja po skladišču oziroma šifri izdelka. Vsaka od postavk na seznamu ima še gumba za brisanja in urejanje le-te. Ob kliku na urejanje se odpre nova stran s podatki o postavki, kjer ima uporabnik možnost spreminjanja količine.



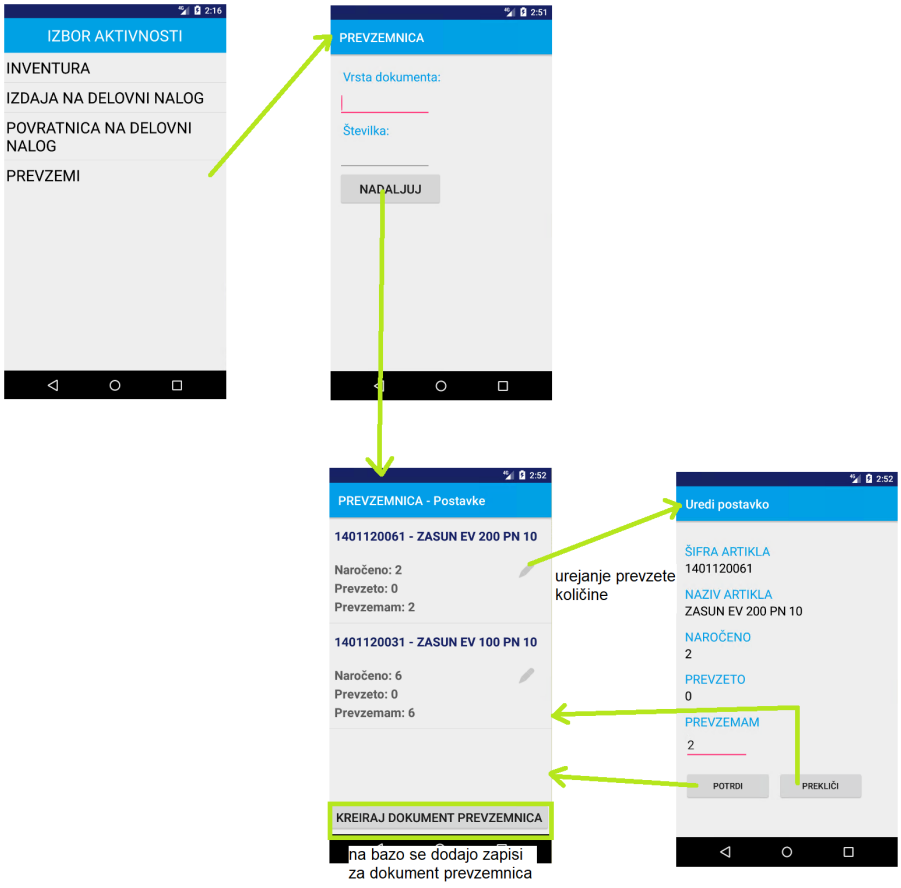
* - na zavihku ARTIKEL se ob vnesu šifre avtomatsko naloži še naziv artikla
** - na zavihku POSTAVKE se ob podrsu navzdol podatki osvežijo.

Slika 4.10: Diagram izvajanja izdaje. Ob izboru izdaje se odpre stran za vnos vrste transakcije, skladišča in delovnega naloga. Tako kot pri inventuri se tudi tokrat v ozadju izvede klic API-ja, s katerega vrednosti napolnijo izbore za vrsto transakcije (prikazovana vrednost je sicer po želji stranke nastavljena privzeto na šifro 133) in skladišča. Delovni nalog vnese uporabnik ročno. Ob kliku na gumb NADALJUJ se nato izvede še en klic API-ja, ki preveri pravilnost delovnega naloga. Če delovni nalog obstaja in je odprt, se na bazi ustvari nov zapis v tabeli z izdajnicami. Odpre se tudi stran z izdajami. Ta vsebuje tri zavihke. Prvi zavihek služi zgolj kot pregled osnovnih informacij

o pravkar kreiranem zapisu v tabeli izdajnic. Drugi zavihek je namenjen dodajanju postavke za izdajo in deluje podobno kot zavihek za dodajanje pri inventurah, le da se tokrat podatki zapišejo v drugo tabelo. Zavihek postavke je ponovno zelo podoben zavihku "Pregled" pri inventuri (vključno z podrsavanjem za osveževanje ter gumboma uredi in briši na posameznih postavkah), le da tukaj nimamo filtrov. Imamo gumb za zaključek izdaje. Ta nas zgolj preusmeri na začetno stran izdaj. Gumb je bil dodan na željo uporabnika, čeprav ima tukaj identično funkcijo kot sistemski gumb "nazaj".



Slika 4.11: Diagram izvajanja povračila. Ob izboru povratnice se odpre stran za vnos delovnega naloga, kjer uporabnik vnese delovni nalog. Ob kliku na gumb NADALJUJ se s klicem API-ja preveri pravilnost delovnega naloga. Nato se ob pravilnem delovnem nalogu odpre stran s seznamom postavk iz vnesenega delovnega naloga. Seznam se s podrsavanjem navzdol osveži. Vsaka postavka ima gumb za urejanje, ki odpre novo stran, kjer uporabnik vnese vrnjeno količino. Ta se zapisuje v vmesno tabelo v bazi. Pod seznamom postavk je gumb za kreiranje povratnice, ki podatke iz vmesne tabele zapiše v "pravo"tabelo, kjer so povratnice.



Slika 4.12: Diagram izvajanja prevzema. Ob izboru prevzema se odpre stran za vnos vrste dokumenta in njegove številke. Ob kliku na gumb NADALJUJ se preveri pravilnost vnesenih polj. Nato se odpre stran s seznamom postavk na vnesenem dokumentu. Gre za podoben seznam kot pri povratnicah, le da gre tokrat za drugačen tip dokumenta pri zapisovanju v bazo. Torej ima podrs za osveževanje ter gumb za urejanje prevzete količine.

4.2.9 Statistični podatki Android aplikacije

V spodnjih dveh tabelah so statistični podatki o celotnem Android projektu.

Število razredov	55
Število vseh datotek v projektu	3,613
Število map v projektu	633
Velikost projekta	59,9 MB
Velikost projekta na disku	66,7 MB

Tabela 4.1: Statistični pregled projekta. Tabela prikazuje število vseh razredov, ki so vsebovani v projektu, število vseh datotek in map ter velikosti projekta in njegovo zasedanje diska.

	št. vseh vrstic	št. vrstic kode	št. vrstic komentarjev	št. praznih vrstic
JAVA	21,601	9,691 (45%)	9,802 (45%)	2,108 (10%)
XML	21,775	19,516 (90%)	76 (0%)	2183 (10%)

Tabela 4.2: Statistični pregled števila vrstic. Tabela prikazuje, koliko vrstic imajo v projektu datoteke s končnico .java ter datoteke končnico .xml. Vrstice so nato razdeljene še na vrstice kode, vrstice komentarjev in prazne vrstice. Vrednosti v oklepajih predstavljajo število vrstic kode, izraženo v odstotkih.

Poglavje 5

Evalvacija

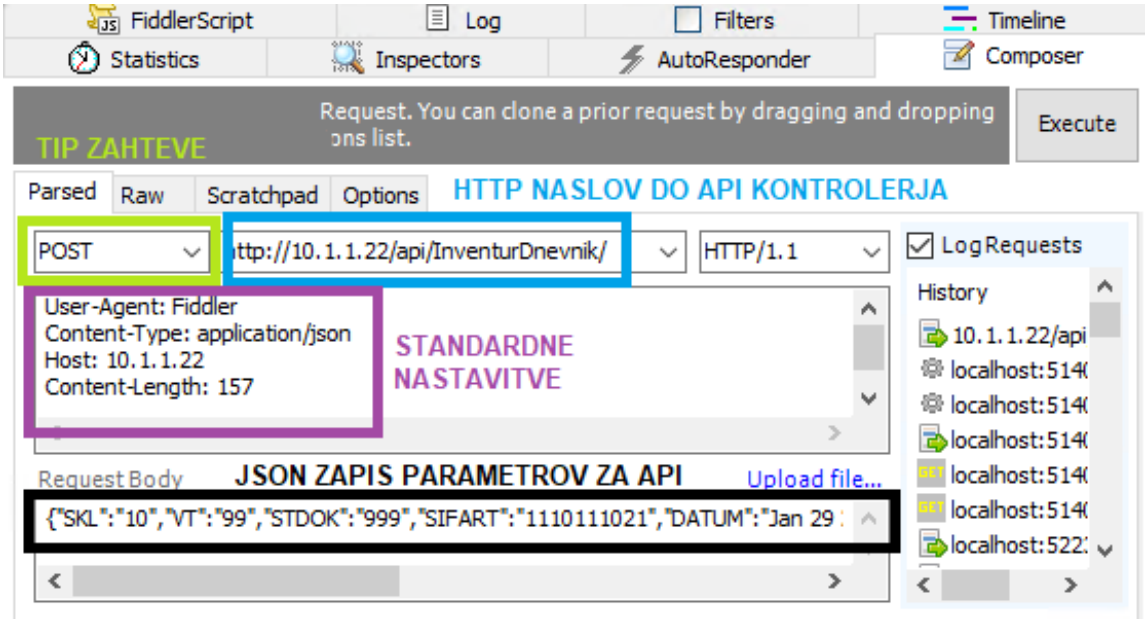
Evalvacijo sestavljata evalvacija API-ja in evalvacija Android aplikacije. Časovno se evalvacija API-ja načeloma izvaja prej, vendar se pogosto zgodi, da se obe evalvaciji izvedeta istočasno oziroma se med seboj izmenjujeta. Zaradi večje jasnosti bomo vsako obravnavali ločeno. Evalvaciji Android aplikacije smo dodali še dve obliki testiranja, ki ju bomo obravnavali na koncu poglavja. To sta testiranje v realnem okolju z metodo "Shadowing" ter microbenchmarking, kjer smo testirali, kako zahtevna je naša Android aplikacija in koliko sredstev porablja.

5.1 Evalvacija API-ja

Za evalvacijo API-ja smo uporabljali razhroščevalnik namestniškega strežnika HTTP Fiddler. Metodo GET lahko sicer testiramo v navadnem spletnem brskalniku, saj ob HTTP zahtevi, brskalnik avtomatsko izvede metodo GET. Spletni brskalnik sam po sebi ni dovolj pri metodah POST, PUT in DELETE. To privede do potrebe po razhroščevalniu proksi strežnika. API smo v prvi fazi testirali na localhost-u. Ko je ta deloval, smo ga objavili preko IIS-ja ter opravili evalvacijo tudi tam.

Testiranje v Fiddlerju poteka tako, da vnesemo URL naslov. Nato izberemo še želeno metodo (GET, POST, PUT, DELETE) ter podamo request

body, če je to potrebno. Request body smo podali v obliki zapisa JSON; ta se kasneje preko API-ja pretvori v spremenljivke, na podlagi katerih sestavljamo SQL stavke (glej poglavje 4). Fiddler nato izvede zahtevo. V primeru GET-a smo tako, kot je zapisano že zgoraj, uporabljali brskalnik Google Chrome ter nato preverili pravilnost vrnjenega JSON formata. Pravilnost v primeru izvedbe metod POST, PUT in DELETE smo preverjali na bazi, saj pri omenjenih metodah rezultat ni vrnjeno besedilo, pač pa sprememba zapisa v bazi. V primeru nepravilnega delovanje smo se vrnili nazaj v Visual Studio in API ter stvar popravili. Primer vnašanja v Fiddler vidimo na spodnji Sliki 5.1.



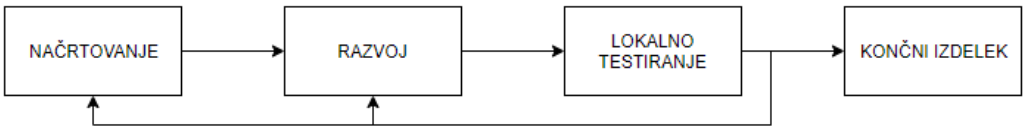
Slika 5.1: Slika prikazuje vnašanje podatkov pri testiranju API-ja s Fiddlerjem

V fazi evalvacije API-ja smo odkrili nekaj napak. Večina napak je izvirala iz napak v SQL stavku, zapisanem v krmilniku. Zaradi manj podrobnega testiranja posameznih SQL stavkov te napake prej niso bile zaznane. Na začetku evalvacije smo ugotovili tudi, da nastavitve na IIS-ju za naš API niso pravilno nastavljene, vendar smo to popravili. Napake te vrste se v prihodnje niso več pojavljale.

5.2 Evalvacija Android aplikacije

5.2.1 Lokalno testiranje

Gre za testiranje v fazi razvoja. Uporabljamo testno bazo na lokalnem strežniku. Poteka primarno na emulatorju, izdelanem znotraj Android Studia. Emulator simulira delovanje fizične naprave. Glede na Android tablico, na kateri se bo na koncu izvajala aplikacija, v Android Studiu izdelamo emulator (identične specifikacije) ter na njem testiramo aplikacijo. Testiranje in evalvacija potekata vzporedno z razvojem (glej Sliko 5.2). Za vsak manjši del razvite aplikacije na emulatorju testiramo, ali zadeva deluje, kot se pričakuje. Če zadeva deluje, nadaljujemo z razvojem, sicer pa z razhroščevalnikom v Android Studiu poskušamo ugotoviti, za kakšno napako gre. Testi v tej fazi niso tako podrobni, saj bodo ti prišli na vrsto kasneje. S tem se dopušča velika možnost manjših hroščev, za katere smo se odločili, da jih bomo odpravljali na koncu.



Slika 5.2: Diagram prikazuje, kam v fazo razvoja spada lokalno testiranje

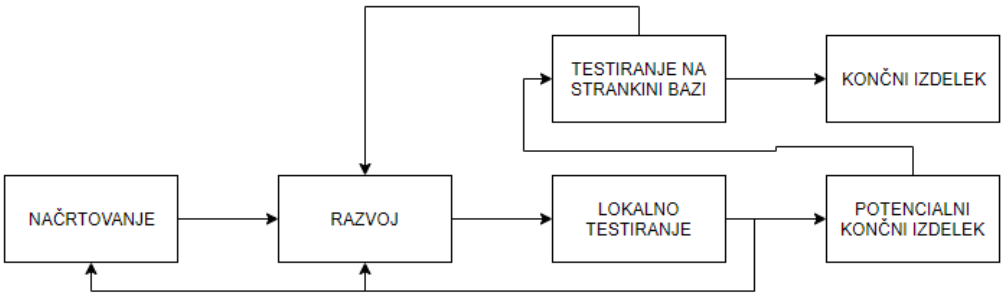
Ko je aplikacija končana oziroma je dokončan njen večji del, nastopi fizična Android tablica. Z njo se preko WiFi-ja povežemo z lokalnim omrežjem. Ponovno se lotimo procesa testiranja ter evalvacije delovanja aplikacije. Ob morebitnih nepravilnostih se vračamo v fazo razvoja. Četudi bi morala aplikacija na tablici delovati identično kot na emulatorju, smo se odločili dodati ta korak, saj se pogosto zgodi, da se v praktičnem okolju stvari zgodijo nekoliko drugače kot v teoriji.

Lokalno testiranje nam je prineslo največ ugotovitev, saj se uporablja v času razvoja in posledično največkrat izvede. V fazi lokalnega testiranja smo odkrili predvsem napake osnovne funkcionalnosti in ne toliko hroščev. Rezultat testiranja je delujoča aplikacija, preverjena na lokalni bazi.

5.2.2 Testiranje na strankini testni bazi

Stranka je podjetje, s katerim smo dogovorjeni za prodajo naše aplikacije, vendar zaradi varovanja podatkov naziv v tem diplomskem delu ne bo omenjen.

Ko smo pri nas testirali in so zadeve delovale pravilno, smo se prek VPN-ja povezali v strankino omrežje in podrobne teste in evalvacije izvedli na njeni testni bazi. Gre za podoben postopek kot pri lokalnem testiranju s tablico, le da smo tukaj preko VPN-ja povezani v dejanskim strankinim omrežjem, kjer se bo tudi kasneje uporabljala aplikacija. Preden se prvič lotimo tega testiranja, moramo stranki nastaviti API in vse nastavitve povezati z njim.



Slika 5.3: Diagram prikazuje, kam v fazo razvoja spada testiranje na strankini bazi

Na diagramu 5.3 vidimo, da se sedaj naš "končni"izdelek spremeni v "potencialnega", končni pa nastopi šele, ko se uspešno konča faza testiranja

pri stranki. Če pa naletimo na kakšno težavo, se ponovno vračamo v fazo razvoja.

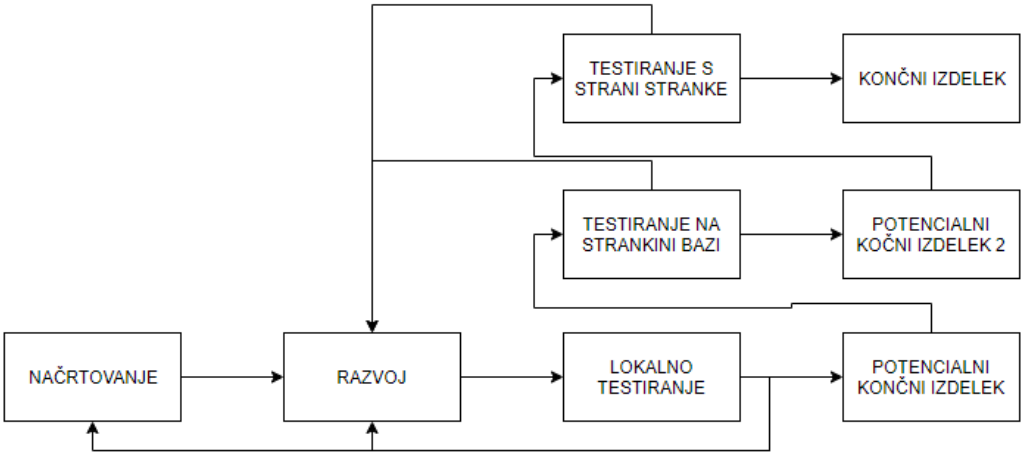
V fazi testiranja na strankini testni bazi smo odkrili neujemanje strankine baze z našo lokalno. Strankini bazi je bilo potrebno dodati vse potrebne tabele in polja, da je aplikacija lahko delovala normalno. Rezultat testiranja sta torej delujoča aplikacija ter delujoča baza stranke.

5.2.3 Testiranje s strani stranke

Ko preverimo, da zadeve tudi v strankinem omrežju delujejo tako, kot smo si zamislili, je na vrsti za testiranje stranka. Stranko najprej v skladišču naučimo uporabljati našo aplikacijo. Nato ji damo en mesec časa, da zadevo testira. Gre predvsem za testiranje všečnosti vmesnika, in ne toliko za dejanski test funkcionalnosti, saj je bil ta opravljen že z naše strani. Še vedno se je zgodilo, da je stranka našla kakšne nepravilnosti v zvezi z delovanjem, kot je na primer občasno nedelovanje posameznega gumba, vendar je bilo teh zelo malo. Testiranje je bolj vplivalo na vizualni izgled aplikacije. Čeprav smo se pri razvoju držali smernic, ki nam jih je stranka vnaprej podala, v živo še vedno prihaja do pripomb. To so lahko velikosti črk, premikanje gumbov, odvečni kliki, zapletena uporaba, slaba odzivnost in podobne zadeve. Tako kot pri prejšnjih dveh testiranjih, se tudi tukaj ob morebitnih nepravilnostih vrnemo v korak razvoja, kjer poskušamo uresničiti strankine želje; končni izdelek se pomakne še za en korak naprej (glej Diagram 5.4).

5.2.4 Testiranje v realnem okolju z metodo “Shadowing”

Pri testiranju v realnem okolju z metodo “Shadowing” gre za posebno obliko testiranja in evalvacije, predstavljene v knjigi Building Mobile Experiences [19]. Večji poudarek je na opazovanju in analiziranju strankine uporabe aplikacije. Izvedemo ga tako, da gremo k stranki (v našem primeru) v skladišče in tam nekaj časa opazujemo in si zapisujemo skladiščnikovo uporabo aplikacije. Na koncu z njim opravimo še krajšo anketo.



Slika 5.4: Diagram prikazuje, kam v fazo razvoja spada testiranje s strani stranke

Shadowing je pomembna oblika evalvacije, saj šele tam dobro vidimo, kako uporabnik uporablja aplikacijo in kakšne težave se mu pojavljajo. V teoriji bi nam sicer testiranje s strani stranke moralo prinesti identične rezultate, vendar se v praktičnem okolju pokaže, da določene stvari stranka spregleda oziroma se ji ne zdijo vredne omembe. Tako recimo vidimo, kako hitro se uporabnik z aplikacijo znajde, ali uporablja enake postopke, kot so bili zamišljeni pri kreiranju aplikacije, ali uporablja vse vgrajene funkcije ali ne, in podobno.

Ozadje

Testiranje v realnem okolju z metodo "Shadowing" smo izvedli približno en mesec po tem, ko smo stranki naložili aplikacijo na Android napravo in pokazali njihovemu skladiščniku, kako se ta uporablja. Testiranje smo opravili z omenjenim skladiščnikom, starim 51 let in s 27 leti delovnih izkušenj z delom v skladišču. Opazovanje je potekalo približno 30 minut. Skladiščnik je v tem času opravljal vse funkcije aplikacije: inventuro ter ustvarjanja izdajnic,

prevzemnic in povratnic. Na koncu smo opravili še krajši pogovor.

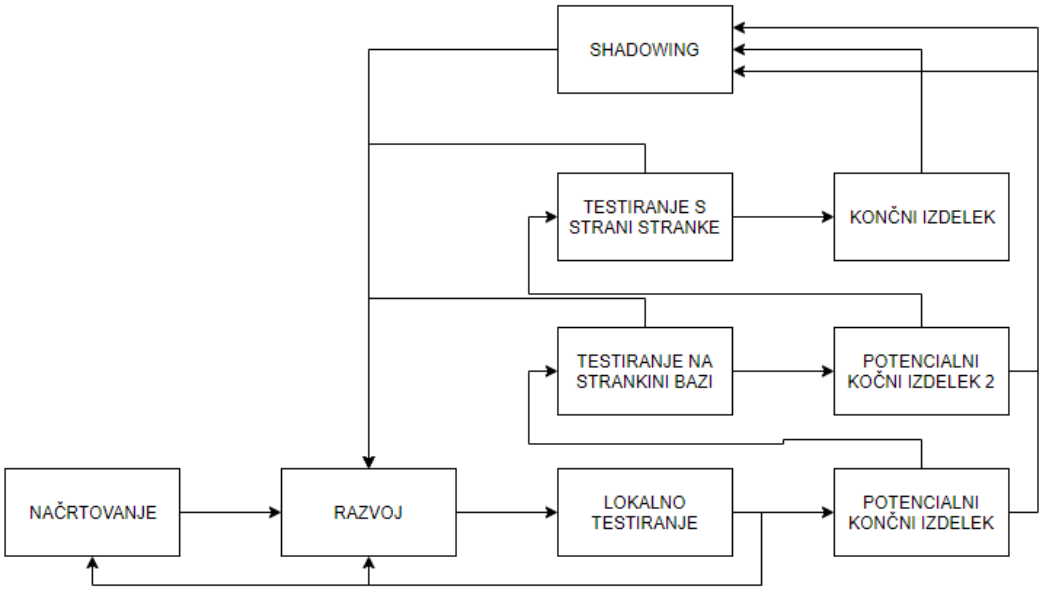
Rezultati in ugotovitve

Po opravljenem opazovanju smo ugotovili, da uporabnik upravlja aplikacijo brez posebnih težav. Smo pa opazili, da uporabnik izpisanih informativnih podatkov, kot je na primer izpis podatkov o glavi izdaje, ne spremlja. Tako bomo v nadaljevanju ob morebitnih hroščih bolj pozorni že na korak, preden se je pojavila težava, saj je morda bila napaka že tam, pa stranka tega ni opazila. Prav tako uporabnik ne uporablja možnosti filtriranja, čeprav si jih je sprva želel. To bomo gotovo v prihodnosti upoštevali in filtrom ne bomo dajali tolikšne pozornosti kot do sedaj.

Ko smo končali z opazovanjem, smo s skladiščnikom opravili še krajši pogovor oziroma anketo. Vprašanja smo imeli pripravljena vnaprej, a smo jih v času ankete prilagajali glede na odgovore. Povedal nam je, da je zelo zadovoljen z odzivnostjo programa in da se mu zdi, da se je porabljen čas za operacije bistveno zmanjšal. Težav s prilagajanjem na aplikacijo ni imel, saj je bila ta grajena na podoben način kot aplikacija, ki so jo uporabljali pred tem, vendar so tisto opustili zaradi počasnega delovanja. Dejal nam je še, da ga malce motijo majhna pisava ter majhni gumbi, saj se mora potruditi da jih zadane. Prav tako je opazil, da se občasno zgodi, da mu pri ustvarjanju izdajnice izračuna napačno številko dokumenta.

Popravki

Na koncu testiranja pridejo na vrsto še popravki oziroma dopolnitve aplikacije, do katerih smo prišli na podlagi testiranja v realnem okolju z metodo “Shadowing”. Po uporabnikovih željah smo povečali velikost pisave in povečali nekatere gumbe. Prav tako smo bolj podrobno analizirali omenjeno težavo z napačnim izračunom številke dokumenta in ugotovili, da do tega prihaja v primerih, ko poženemo dve SQL procedure in se druga konča hitreje kot prva. Da bi to preprečili, smo v aplikacijo med oba klica dodali umeten zamik, dolg pol sekunde, in s tem preprečili, da bi se drugi ukaz končal pred



Slika 5.5: Diagram prikazuje, kam v fazo razvoja spada shadowing

prvim.

Testiranje v realnem okolju z metodo "Shadowing" nam je zagotovo zelo koristilo. Dobili smo pozitiven odziv in s tem potrditev, da smo aplikacijo razvijali v pravi smeri in se bomo podobnih smernic držali tudi v prihodnje.

5.2.5 Microbenchmarking

API microbenchmarking

Opravili smo meritve odzivnosti API-ja – merili smo torej čas, ki preteče od trenutka, ko podamo HTTP zahtevo, do trenutka, ko dobimo odgovor. Seveda je to v veliki meri odvisno tudi od hitrosti internetne povezave, zmogljivosti strežnika, na katerem teče baza, ter fizični oddaljenosti strežnika, na katerem je API. Če bi imeli API na primer na Kitajskem, bi odgovor dobili kasneje, kot če ga imamo v isti pisarni. Ker pa bo naša aplikacija delovala v istem omrežju kot API in baza, smo v takem okolju testirali tudi mi. Za meritve smo uporabili Fiddler.

	GET	POST	PUT	DELETE
čas (ms)	15	187	16	93

Tabela 5.1: API microbenchmarking. V razpredelnici so podani povprečni časi, ki jih naš API potrebuje, da izvede posamezne HTTP metode. Test je potekal tako, da smo vsako metodo izvedli dvajsetkrat in izračunali povprečni čas, ki ga je API potreboval, da je metodo izvedel in nam vrnil rezultate. Metode smo izvajali na različnih krmilnikih. Metodo DELETE smo na primer uporabili za brisanje podatkov iz tabele inventur, iz tabele izdajnic in tabele povratnic. Rezultati so dobri, saj nam vse metode (z izjemo metode POST) izvede v manj kot desetinki sekunde. Tudi metoda POST se izvede relativno hitro. Potrebuje manj kot dve desetinke sekunde, kar je še vedno zanemarljivo majhen čas glede na to, da v bazo dodaja nov zapis.

Android aplikacija microbenchmarking

Meritve smo opravili na Android tablici Unitech PA720, na kateri se pri stranki izvaja aplikacija ter na emulatorju, ki ima nastavljene enake lastnosti kot omenjena tablica. Tablica vsebuje 1.3 GHz štirijedrni procesor, 3 GB RAM-a ter ima resolucijo 700x1280. Za meritve smo uporabljali Android Studio. Rezultati testiranja obremenitve bralno-pisalnega pomnilnika

(RAM) in procesorja (CPU) so vidni na razpredelnici. Stolpec "mirovanje"predstavlja čas, ko je aplikacija v mirovanju in imamo odprto poljubno stran. V procesu testiranja smo ugotovili, da je rezultat enak ne glede na to, katero operacijo imamo odprto. Stolpci "inventura", "izdaje", "povračila"ter "prevzemi"predstavljajo najvišje vrednosti oziroma obremenitve, do katerih smo prišli v času testiranja aplikacije. Rezultati so prikazani v spodnji razpredelnici.

	mirovanje	inventura	izdaje	povračila	prevzemi
RAM (MB)	25	25,2	25,9	24,6	24,7
CPU (%)	11	81	72	75	79

Tabela 5.2: Android aplikacija microbenchmarking na emulatorju. Obremenitev RAM-a je ves čas približno 25 MB, kar je zelo majhna številka in pomeni nizko obremenitev RAM-a. Številke pri obremenitvi procesorja pa v najvišji točki precej narastejo in se približajo 80 odstotkom. Vseeno ne gre za problematično stvar, saj je to zgolj kratek skok ob HTTP zahtevi do API-ja in polnitvi polj v aplikaciji. V naslednjih sekundah se stanje ponovno umiri na približno 10-15 odstotkov, kar je relativno nizko. Poleg tega so tablice namenjene zgolj uporabi naše aplikacije in posledično relativno nizki porabi virov s strani drugih procesov.

	mirovanje	inventura	izdaje	povračila	prevzemi
RAM (MB)	25	25,3	25,1	23,9	24,5
CPU (%)	9	75	60	62	71

Tabela 5.3: Android aplikacija microbenchmarking na napravi PA720. Rezultati so podobni rezultatom na emulatorju, kar je bilo pričakovati. Obremenitev RAM-a se tudi tukaj giblje okoli 25 MB, Obremenitve procesorja so v odstotku nekoliko nižje od obremenitev pri emulatorju, a razlike niso velike.

Microbenchmarking testiranje API-ja in Android aplikacije je pokazalo, da naša aplikacija deluje hitro in uporabniku ponuja odzivno uporabniško izkušnjo. Poleg tega aplikacija ne porablja veliko sredstev in s tem predstavlja relativno enostavne procese za mobilno napravo.

Poglavje 6

Sklepne ugotovitve in smernice

6.1 Ugotovitve

V tem diplomskem delu smo predstavili:

- ozadje trenutnega stanja na trgu mobilne podpore skladiščnemu poslovanju,
- implementacijo REST API-ja,
- implementacijo Android aplikacije,
- evalvacijo implementiranih rešitev na več načinov.

V vsem tem procesu je prihajalo tudi do ovir in težav, ki pa smo jih uspeli reševati. Veliko časa nam je vzelo analiziranje in načrtovanje aplikacije, predvsem njene podobe, saj igra ta v očeh uporabnika veliko vlogo. Za to je bilo potrebno precej skiciranja rešitev in pogovorov znotraj kolektiva E.R.S Rokada. Posvete smo imeli tudi z bodočo stranko.

Problem je predstavljal tudi slab WiFi signal v skladišču. Težavo smo rešili tako, da smo se začeli povezovati preko mobilnih podatkov namesto preko WiFi-ja.

6.2 Smernice za nadaljne delo

Čeprav smo uspeli implementirati začetne cilje, ima aplikacija še vedno prostor za izboljšave. V prvi vrsti je možnost napredka pri nadzoru uporabnikov. Trenutno aplikacija namreč ne vsebuje prijave, pač pa se vsak, ki požene aplikacijo, prijavlja kot isti uporabnik. V tem trenutku to sicer ni problem, saj je uporabnik zgolj eden, vendar bo lahko v prihodnosti uporabnikov več. Takrat bo prijava različnih uporabnikov potrebna. Prijavo bi implementirali tako, da bi v bazo dodali tabelo uporabnikov, ki bi vsebovala njihova uporabniška imena in gesla. Uporabnik bi v aplikaciji na zaslonu za prijavo vnesel svoje uporabniško ime ter geslo. Nato bi izvedli poizvedbo v bazo, kjer bi preverili njuno ujemanje. Če bi se uporabniško ime in geslo ujemalo, bi uporabnika spustil naprej, sicer pa ne.

Izboljšati se da tudi povezljivost, saj do API-ja sedaj dostopamo tako, da se povezujemo najprej v njegovo zasebno omrežje, nato pa od tam izvajamo klice do API-ja. Korak povezovanja v zasebno omrežje bi bilo dobro izpustiti, vendar bi za to bilo potrebno odpreti nekaj vrat v omrežje. Tukaj pa lahko ogrozimo varnost, zato bi bilo potrebno uvesti tudi dodatne varnostne ukrepe. Lahko bi uporabili protokol OAuth, kjer se klient najprej poveže z avtentikacijskim servisom, tam pa dobi žeton, s katerim se lahko nato povezuje na API.

Sicer se bo aplikacija izboljševala sproti glede na uporabnikove ideje in želje, kot smo to storili že vmes, po testiranju v realnem okolju z metodo "Shadowing". Popravki so sicer majhni, vendar močno pripomorejo k boljši uporabniški izkušnji.

Literatura

[1] Mark F. Livesay, et. al. Paperless warehouse management system. RETROTECH Inc, 1998. Dosegljivo 22.7.2018 na: https://patents.google.com/patent/US6339764B1/en

[2] Tutorials Teacher. What is Web API? Dosegljivo 22.7.2018 na: http://www.tutorialsteacher.com/webapi/what-is-web-api

[3] Realm. MVC vs. MVP vs. MVVM on Android. Dosegljivo 22.7.2018 na: https://academy.realm.io/posts/eric-maxwell-mvc-mvp-and-mvvm-on-android/

[4] Intrix. Programska integracija. Dosegljivo 22.7.2018 na: https://www.intrix.si/blog/it-aktualno/programska-integracija/

[5] Tomaž Žniderič. Povezovanje androidne aplikacije s spletnim ogrodjem Django. Diplomska naloga, Fakulteta za računalništvo in informatiko, Univerza v Ljubljani, 2015.

[6] Medium. Android Architecture Patterns Part 1: Model-View-Controller. Dosegljivo 22.7.2018 na: https://www.thoughtco.com/what-is-java-2034117

[7] Code Project. Building REST services with ASP.NET Core Web API and Azure SQL Database. Dosegljivo 22.7.2018 na: https://www.codeproject.com/Articles/1106622/Building-REST-services-with-ASP-NET-Core-Web-API-a

[8] Android Hive. Android JSON Parsing Tutorial. Dosegljivo 22.7.2018 na: https://www.androidhive.info/2012/01/android-json-parsing-tutorial/

[9] Developers. Activity. Dosegljivo 22.7.2018 na: https://developer.android.com/reference/android/app/Activity

[10] Developers. Fragment. Dosegljivo 22.7.2018 na: https://developer.android.com/reference/android/components/Fragment

[11] Developers. AsyncTask. Dosegljivo 22.7.2018 na: https://developer.android.com/reference/android/os/AsyncTask

[12] Comtron. Skladiščenje. Dosegljivo 22.7.2018 na: http://www.comtron.si/erp-tronintercenter/skladiscenje/

[13] Špica. Skladiščno poslovanje. Dosegljivo 22.7.2018 na: https://www.spica.si/resitve/skladiscno-poslovanje

[14] PerfTech. Celovita rešitev za mobilno skladiščno poslovanje. Dosegljivo 22.7.2018 na: http://www.perftech.si/resitve/strojna-oprema/racunalniska-strojna-oprema-za-skladisca/

[15] Vasco. Spletne in mobilne aplikacije. Dosegljivo 22.7.2018 na: https://www.vasco.si/produkti/web-storitve/

[16] Minoa. Minoa Logist. Dosegljivo 22.7.2018 na: http://www.minoa.si/logist

[17] ThoughtCo. What is Java? Dosegljivo 22.7.2018 na: https://www.thoughtco.com/what-is-java-2034117

[18] E.R.S. Rokada. Rešitve. Dosegljivo 22.7.2018 na: http://www.ers-rokada.si/rescaronitve.html

[19] Frank Bentley and Edward Barrett. Building Mobile Experiences. The MIT Press, Cambridge, Massachusetts, London, 2012.

[20] Belgrade-SqlClient. Dosegljivo 18.8.2018 na: https://github.com/JocaPC/Belgrade-SqlClient