SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 262

MOBILNA APLIKACIJA ZA PROVOĐENJE INVENTURE NA TEMELJU DETEKCIJE BARKODA IZ SLIKE

Fran Kristijan Jelenčić

Zagreb, lipanj 2021.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 262

MOBILNA APLIKACIJA ZA PROVOĐENJE INVENTURE NA TEMELJU DETEKCIJE BARKODA IZ SLIKE

Fran Kristijan Jelenčić

Zagreb, lipanj 2021.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 12. ožujka 2021.

ZAVRŠNI ZADATAK br. 262

Pristupnik:	**Fran Kristijan Jelenčić (0036518901)**
Studij:	Elektrotehnika i informacijska tehnologija i Računarstvo
Modul:	Računarstvo
Mentor:	izv. prof. dr. sc. Alan Jović

Zadatak: **Mobilna aplikacija za provođenje inventure na temelju detekcije barkoda iz slike**

Opis zadatka:

U ovom završnom radu potrebno je ostvariti mobilnu aplikaciju za operacijski sustav Android kako bi se omogućilo skeniranje barkodova za provođenje inventure. Aplikacija treba omogućiti korištenje kamere za fotografiranje barkoda, ispravnu detekciju barkoda koristeći određeni model strojnog učenja, te vođenje stanja inventure i vođenje dnevnika ažuriranja stanja u bazi podataka. Detektirani barkodovi šalju se na poslužitelj radi evidencije u bazi podataka. Pritom je potrebno pretpostaviti postojanje više skladišta za koje se radi inventura, svakog sa svojim stanjem. Dodatno, aplikacija treba omogućiti izbor između nekoliko poslužitelja ovisno o skladištu za koje se radi inventura, ručni unos barkoda, pregled trenutnog stanja i pregled dnevnika ažuriranja stanja. Implementaciju je potrebno ostvariti u programskom jeziku po vlastitom izboru.

Rok za predaju rada: 11. lipnja 2021.

Sadržaj

Uvod

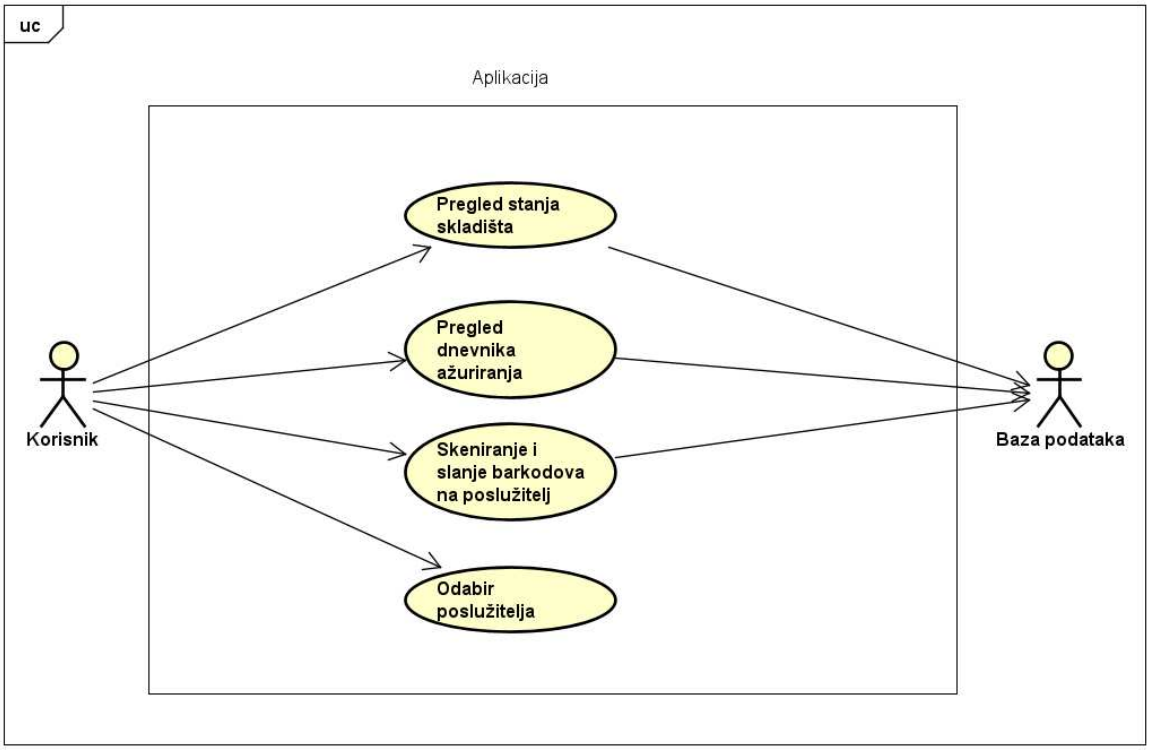
U ovom radu predstavit ću aplikaciju namijenjenu za rješavanje problema provođenja inventure koja koristi kameru za skeniranje barkodova te njihovo slanje na poslužitelj gdje se evidentiraju u bazu podataka. Potaknut iskustvom rada u skladištu te neefikasnošću procesa provođenja inventure odlučio sam za završni rad razviti mobilnu aplikaciju za provođenje inventure. Funkcionalnosti aplikacije bile su usmjerene rješavanju problema zapisivanja stanja na papir te popratnog unošenja stanja u sustav čime bi se proces inventure znatno ubrzao. Na servisu za distribuciju aplikacija Google Play postoje slične aplikacije, ali one nemaju mogućnosti slanja na poslužitelj već samo lokalna pohrana podataka.

U idućem poglavlju opisati ću korištenu arhitekturu, funkcionalne zahtjeve i način interakcije pojedinih dijelova sustava. Drugo i treće poglavlje analiziraju implementaciju rada te korištene tehnologije i alate i način na koji su korišteni.

1. Arhitektura sustava

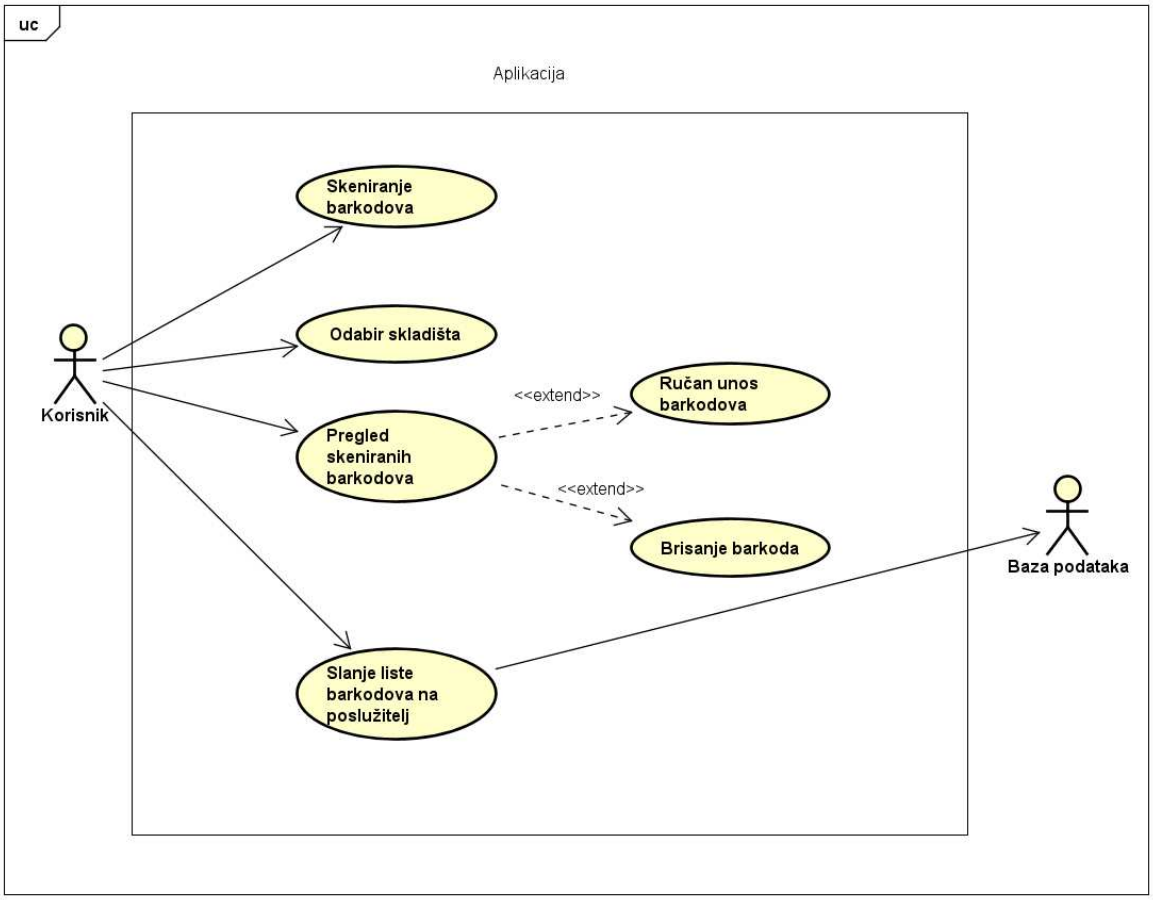
1.1. Funkcionalni zahtjevi

Aplikaciju koriste korisnici koji imaju mogućnost pregleda stanja skladišta gdje se u obliku tablice pokaže trenutno stanje odabranog skladišta ili ukupno stanje na svim skladištima, mogućnost pregleda dnevnika ažuriranja u kojem je moguće vidjeti kada i na kojem skladištu je skeniran koji barkod, mogućnost skeniranja i slanja barkodova na poslužitelj te odabira poslužitelja to jest promjena trenutnog poslužitelja. Navedeni obrasci uporabe vidljivi su na slici (Slika 1.1)



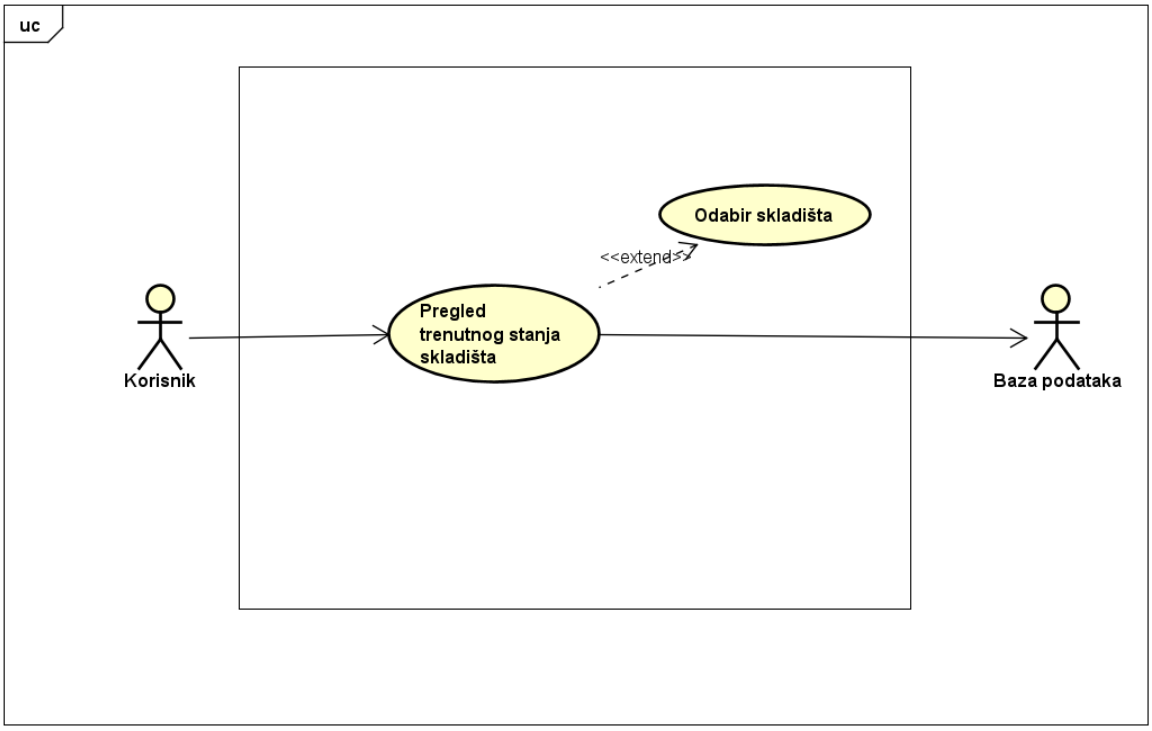
Slika 1.1 Obrasci uporabe – korištenje aplikacije

Prilikom skeniranja i slanja barkodova na poslužitelj korisnik ima opciju ručnog unosa barkoda, brisanja odabranog barkoda, odabira skladišta za koje se radi inventura te mogućnost slanja barkodova na poslužitelj. Ovi obrasci uporabe vidljivi su na slici (Slika 1.1).



Slika 1.2 Obrasci uporabe – skeniranje i slanje barkodova na poslužitelj

Kada korisnik pregledava stanje skladišta zadana vrijednost je pregled svih skladište te korisnik ima opciju ukoliko želi može pregledati stanje na određenom skladištu, što je vidljivo na slici (Slika 1.3)



Slika 1.3 Obrasci uporabe – pregled stanja skladišta

1.2. Komponentne sustava

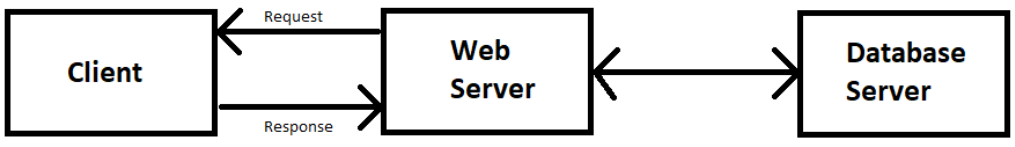
Sustav se temelji na arhitekturnom obrascu model-pogled-nadzornik (eng. *Model-View-Controller*, dalje: MVC) te se sastoji od dvije komponente, poslužitelja i aplikacije klijenta.

Arhitekturni obrazac MVC temelji se na podjeli na tri dijela: model (obrada), nadzornik (ulaz), pogled (izlaz) [9]. U ovom sustavu ulogu modela i nadzornika obavlja radni okvir Spring, a ulogu pogleda obavlja Android aplikacija. Oni će biti objašnjeni kasnije u radu.

Poslužitelj je udaljeno računalo na kojem se nalaze web aplikacija i baza podataka. Web aplikacija sluša na određenim vratima (eng. *port*) zahtjeve koje klijent šalje te ih obrađuje i šalje odgovor, u odgovoru se najčešće nalaze podatci koje web aplikacija dohvaća iz baze podataka.

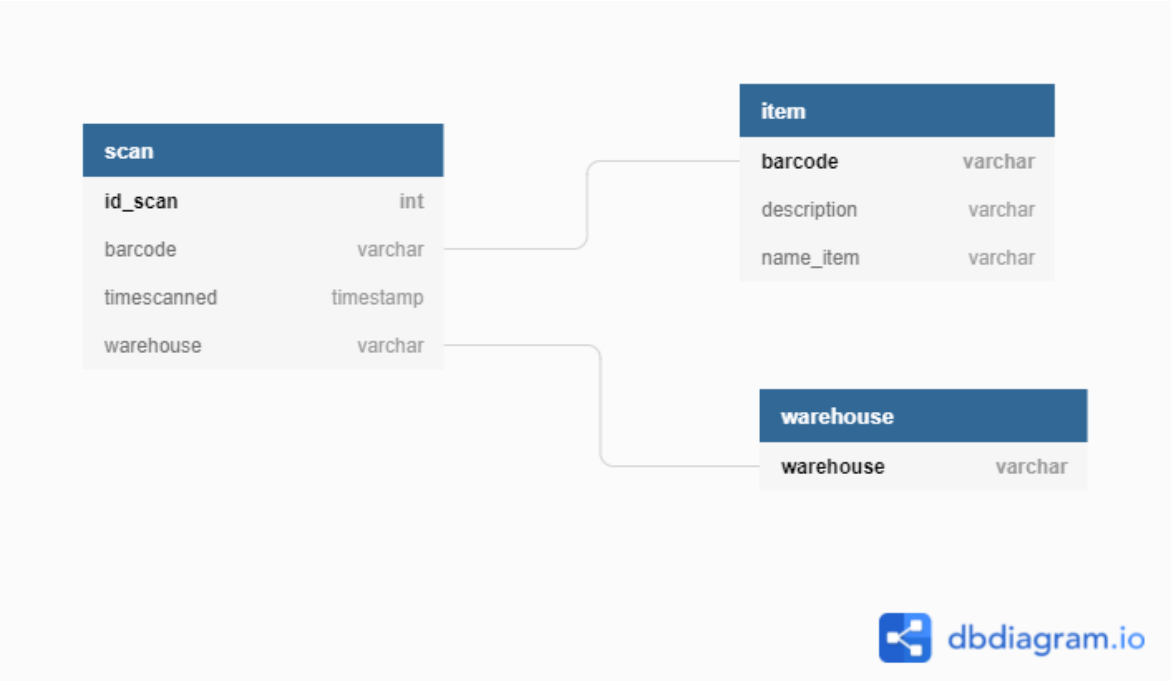
Klijent je mobilni telefon koji koristi operacijski sustav Android na kojem se nalazi instalirana aplikacija „Inventory App“. Prilikom korištenja aplikacije korisnik odabire razne opcije koje su opisane u prethodnom potpoglavlju i na temelju odabira aplikacija šalje potrebne zahtjeve na web poslužitelj.

Web poslužitelj i aplikacija komuniciraju preko HTTP protokola, gdje klijent šalje HTTP zahtjev (eng. *HTTP request*), a poslužitelj odgovara HTTP odgovorom (eng. *HTTP response*). Slika (Slika 1.4) ilustrira komponente i način na koji one komuniciraju.



Slika 1.4 Izgled sustava klijent poslužitelj

Baza podataka je strukturirani skup podataka te je za potrebe sustava korištena relacijska baza u kojoj su podaci podijeljeni u tablice, svaka sa svojim atributima. Slika (Slika 1.5) prikazuje izgled baze korištene u ovom radu.



Slika 1.5 Izgled baze podataka

Korištene su tri tablice: tablica **Scan**, **Item** i **Warehouse**.

U tablici **Item** nalaze se atributi **barcode**, **description**, **name_item.** Tablica predstavlja artikle koji mogu postojati u nekom skladištu. Atribut **barcode** je primarni ključ te predstavlja jedinstveni barkod tog artikla, njegov podatkovni tip je varijabilni niz znakova (eng. *varchar*), nakon njega je atribut **description** koji predstavlja kratki opis artikla također podatkovnog tipa (varijabilni niz znakova) i posljednji atribut **name_item** koji predstavlja ime artikla, isto podatkovnog tipa varijabilnog niza znakova.

U tablici **Warehouse** nalazi se samo jedan atribut i to je **warehouse** koji je ujedno i primarni ključ te predstavlja ime skladišta podatkovnog tipa (varijabilni niz znakova).

U tablici **Scan** postoje atributi **id_scan**, **barcode**, **timescanned** i **warehouse**. Atribut **id_scan** koji je primarni ključ i jednoznačno određuje jedan **Scan**, podatkovnog tipa broj (eng. *Integer*), atribut **barcode** koji predstavlja barkod kojeg je korisnik pomoću aplikacije skenirao te poslao na evidenciju u bazu podataka podatkovnog tipa varijabilnog niza znakova, atribut **timescanned** koji predstavlja vrijeme kada je korisnik skenirao taj barkod

podatkovnog tipa vremenske oznake (eng. *timestamp*) i atribut **warehouse** koji predstavlja skladište u kojem je skeniran barkod podatkovnog tipa varijabilnog niza znakova.

Dijagrami razreda

1.2.1. Model

Modeli su razredom opisani entiteti koji se nalaze u bazi podataka, tako postoje tri prava i dva pomoćna modela unutar ovog sustava, oni su: **Inventory**, **Item**, **KeyValuePair**, **Scan**, **Warehouse**.

Model **Inventory** modelira inventar skladišta te se sastoji od liste objekata razreda **KeyValuePair**.

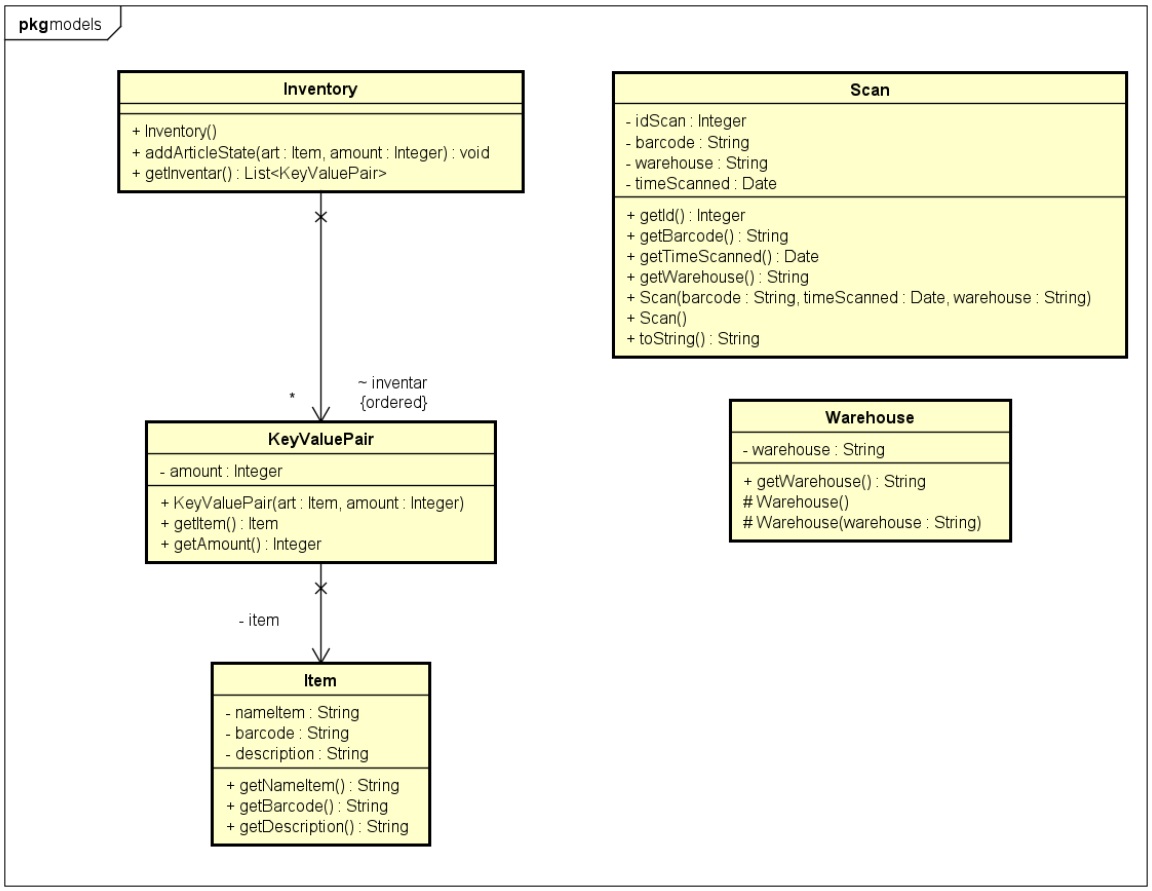
Model **Item** predstavlja jedan artikl koji ima atribute **nameItem** tipa String koji predstavlja ime, atribut **description** tipa String koji predstavlja opis i atribut **barcode** tipa String koji predstavlja barkod.

Model **KeyValuePair** nije entitet u bazi podataka već pomoćni model koji se koristi kako bi se mogao u razredu **Inventory** pamtiti artikl i njegova količina na skladištu, ima 2 atributa, referencu na jedan objekt razreda **Item** te broj (eng. *Integer*) koji predstavlja količinu tog artikla na skladištu.

Model **Scan** predstavlja jedan skenirani barkod, sadrži podatak **timeScanned** vrijeme kada je skenirano, **barcode** koji predstavlja skeniran barkod te **warehouse** na kojem skladištu je skenirano.

Model **Warehouse** predstavlja jedno skladište te ima samo jedan atribut **warehouse** koji predstavlja ime specifičnog skladišta.

Na Slika 1.6 prikazan je dijagram razreda za modele.



Slika 1.6. Dijagram razreda – modeli

1.2.2. Sloj nadglednika

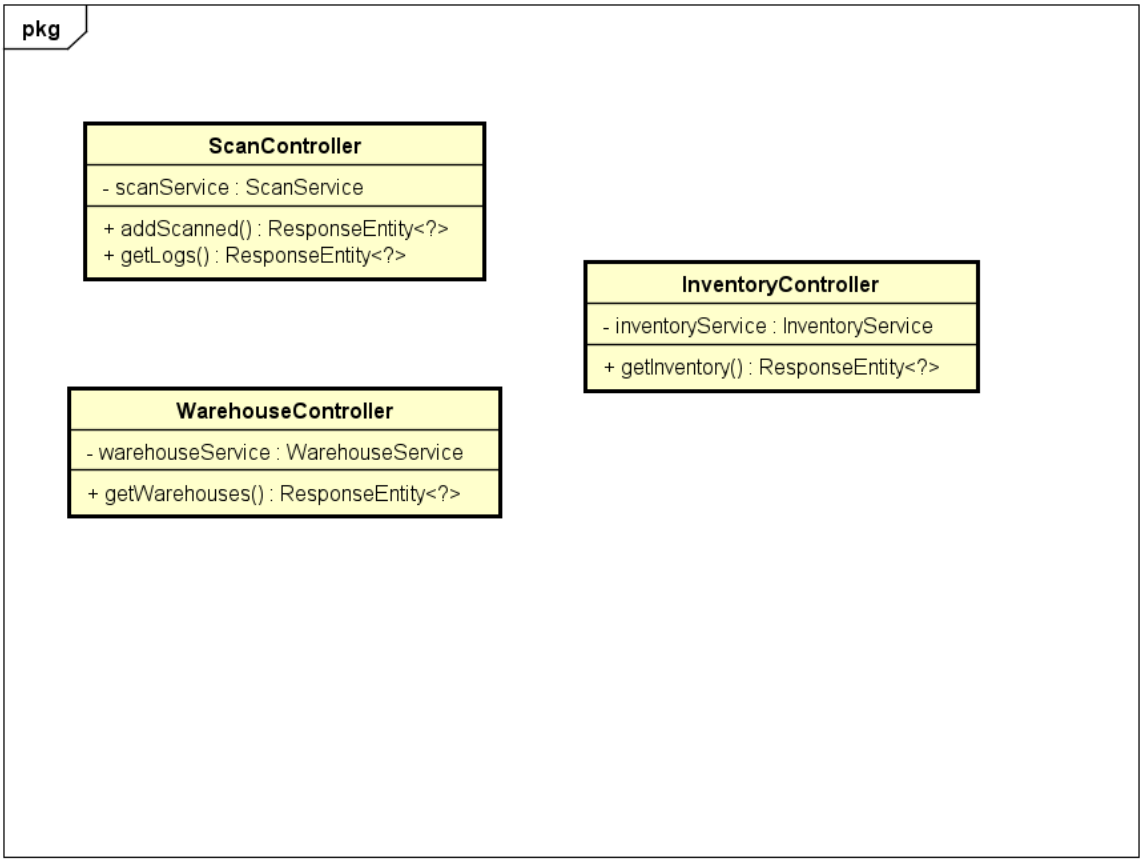
Sloj nadglednika korišten u ovom radu sastoji se od tri nadglednika: **InventoryController**, **ScanController** i **WarehouseController**.

InventoryController nalazi se na adresi „/inventory“, sadrži referencu na **InventoryService** servis te samo jednu metodu **getInventory(String warehouse)** koja vraća sve artikle koji se nalaze u skladištu koje je predano kao argument funkciji.

ScanController se nalazi na adresi „/scan“ te sadrži referencu na **ScanService** servis te dvije metode **addScanned(List<Scan> scanned)** i **getLogs()**, metoda **addScanned** prima listu objekata razreda **Scan** te ih dodaje u bazu podataka dok metoda **getLogs** vraća listu svih **Scan** objekata iz baze podataka.

WarehouseContoller nalazi se na adresi „/warehouse“, sadrži referencu na **WarehouseService** servis i jednu metodu **getWarehouses()** koja vraća listu imena skladišta.

Izgled sloja nadglednika vidljiv je na sljedećoj slici (Slika 1.7).



Slika 1.7. Dijagram razreda – nadglednici

1.2.3. Sloj usluge

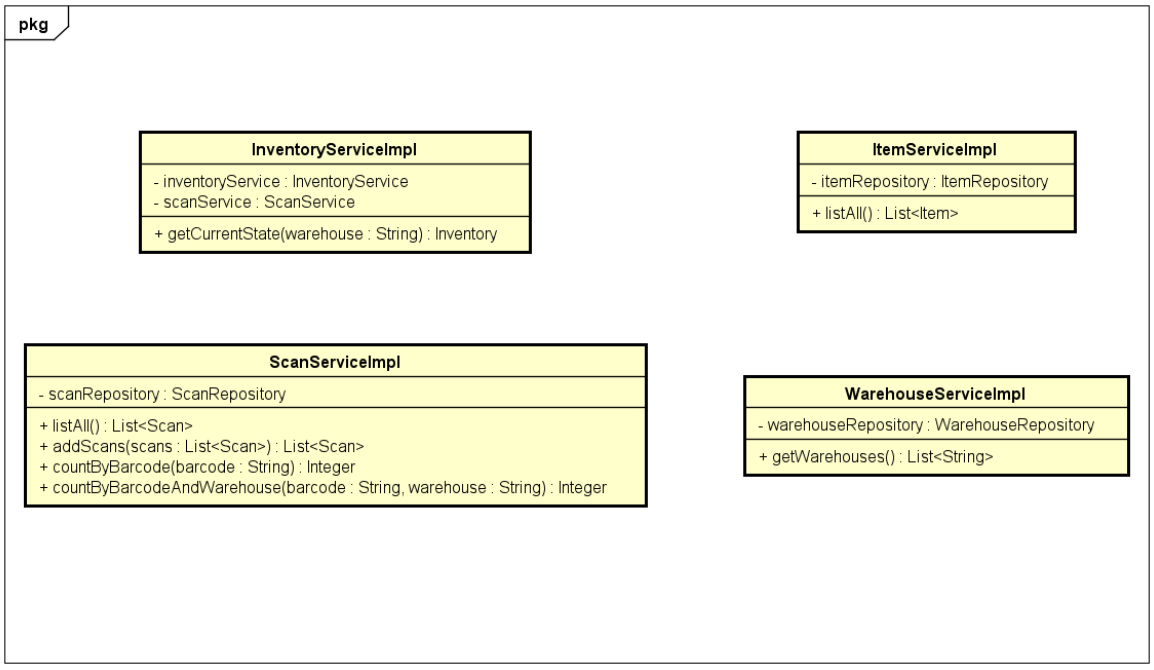
U sloju usluge nalaze se usluge koje obavljaju poslovnu logiku, pozivaju se od strane nadglednika te najčešće dohvaćaju podatke pomoću repozitorija te onda manipuliraju tim podatcima.

InventoryService sastoji se od referenca na **ScanService** i **ItemService** te implementira jednu metodu **getCurrentState(String warehouse)** koja vraća listu artikala koji se nalaze u skladištu sa danim imenom.

ItemService koji sadrži referencu na **ItemRepository** i implementira metodu **listAll()** koja vraća sve artikle u bazi.

ScanService koji ima referencu na **ScanRepository** i metode **listAll()** i **addScans(List<Scan> scans)**. Metoda **listAll** vraća sve objekte Scan koji se nalaze u bazi podataka, a metoda **addScans** dodaje sve objekte liste scans u bazu podataka.

WarehouseService koji sadrži referencu na **WarehouseRepository** i metodu **getWarehouses()** koja vraća listu svih skladišta.



Slika 1.8. Dijagram razreda – usluge

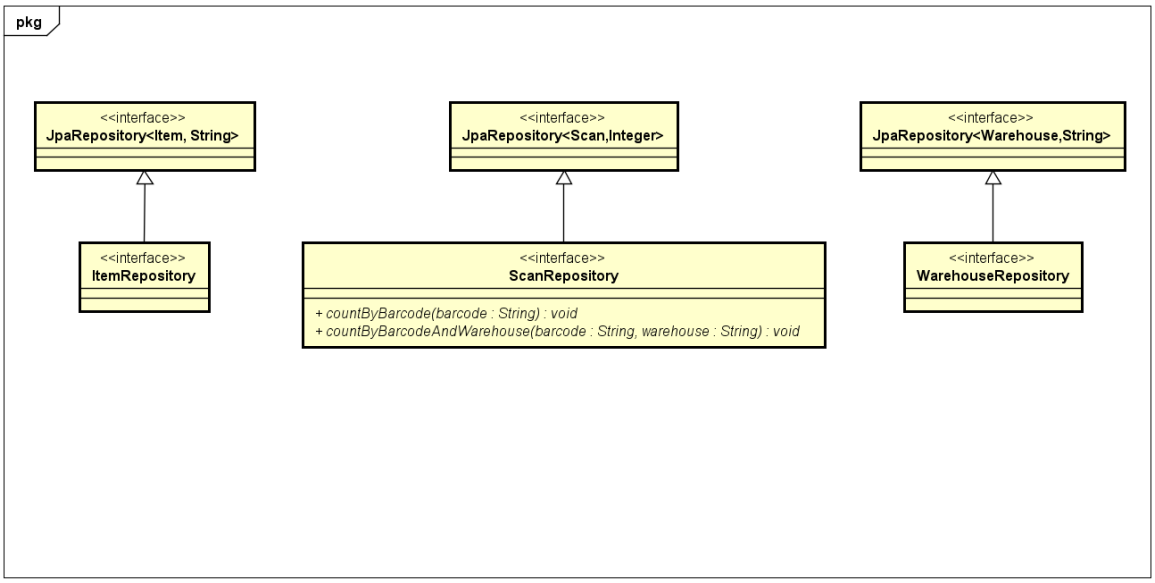
1.2.4. Sloj repozitorija

U sloju repozitorija nalaze se razredi koji su zaduženi za dohvaćanje podataka iz različitih izvora koji mogu biti npr. baza podataka, sa datotečnog sustava ili preko mrežnih servisa. Izvor koji ova implementacija koristi je baza podataka i sastoji se od tri sučelja koji implementiraju gotovo sučelje **JpaRepository** koje implementira već gotove metode za dohvaćanje, dodavanje, brojanje, sortiranje i ažuriranje podataka. JPA (eng. *Jakarta Persistence*) dio je radnog okvira Spring te mu je cilj omogućiti lakšu implementaciju pristupa podacima koji se nalaze u bazama podataka.

ItemRepository koristi gotove metode sučelja **JpaRepository** te ne implementira svoje metode.

ScanRepository koji osim gotovih metoda implementira i metode **countByBarcode(String barcode)** koja vraća broj zapisa sa danim barkodom i metoda **countByBarcodeAndWarehouse(String barcode, String warehouse)** koja vraća broj zapisa sa danim barkodom u danom skladištu.

WarehouseRepository koji također ne implementira svoje metode već se oslanja na gotove metode sučelja **JpaRepository**.



Slika 1.9. Dijagram razreda – repozitoriji

1.2.5. Android aplikacija

Android aplikacija zadužena je za sloj pogleda obrasca MVC. Ona se sastoji od velikog broja razreda, a zato će ovdje biti navedene samo najvažniji. Metode **onCreate(…)**, **onCreateView(…)** i **onViewCreated(…)** služe za inicijalizaciju te kreiranje izgleda i najčešće se u njima definira ponašanje i funkcionalnost razreda koji ih implementiraju.

MainActivity je razred koji implementira aktivnost (eng. *activity*) koja se pokreće prilikom pokretanja aplikacije te se preko nje pokreću sve ostale akcije koje korisnik želi obavljati, sastoji se od inicijalizacije korisničkog sučelja te njegove funkcionalnosti.

Svi razredi koji imaju varijablu **prefsName** koriste **SharedPreferences**, razred koji služi za pohranjivanje podataka u memoriju telefona kako bi adresa poslužitelja ostala zapamćena između različitih pokretanja aplikacije. **prefsName** je varijabla u kojoj se nalazi putanja za dohvaćanje adrese spremljenog poslužitelja

InventoryFragment je razred koji implementira metodu **loadWarehouseFromProvider()** i **load fromProvider()**. Metoda **loadWarehouseFromProvider()** sa zadanog poslužitelja dohvaća listu skladišta te zatim poziva **loadfromProvider()** koja dohvaća njihove inventare ovisno o korisnikovom odabiru skladišta i zatim inventar prikazuje u obliku tablice, prema zadanim postavkama aplikacija dohvaća sveukupno stanje svih skladišta.

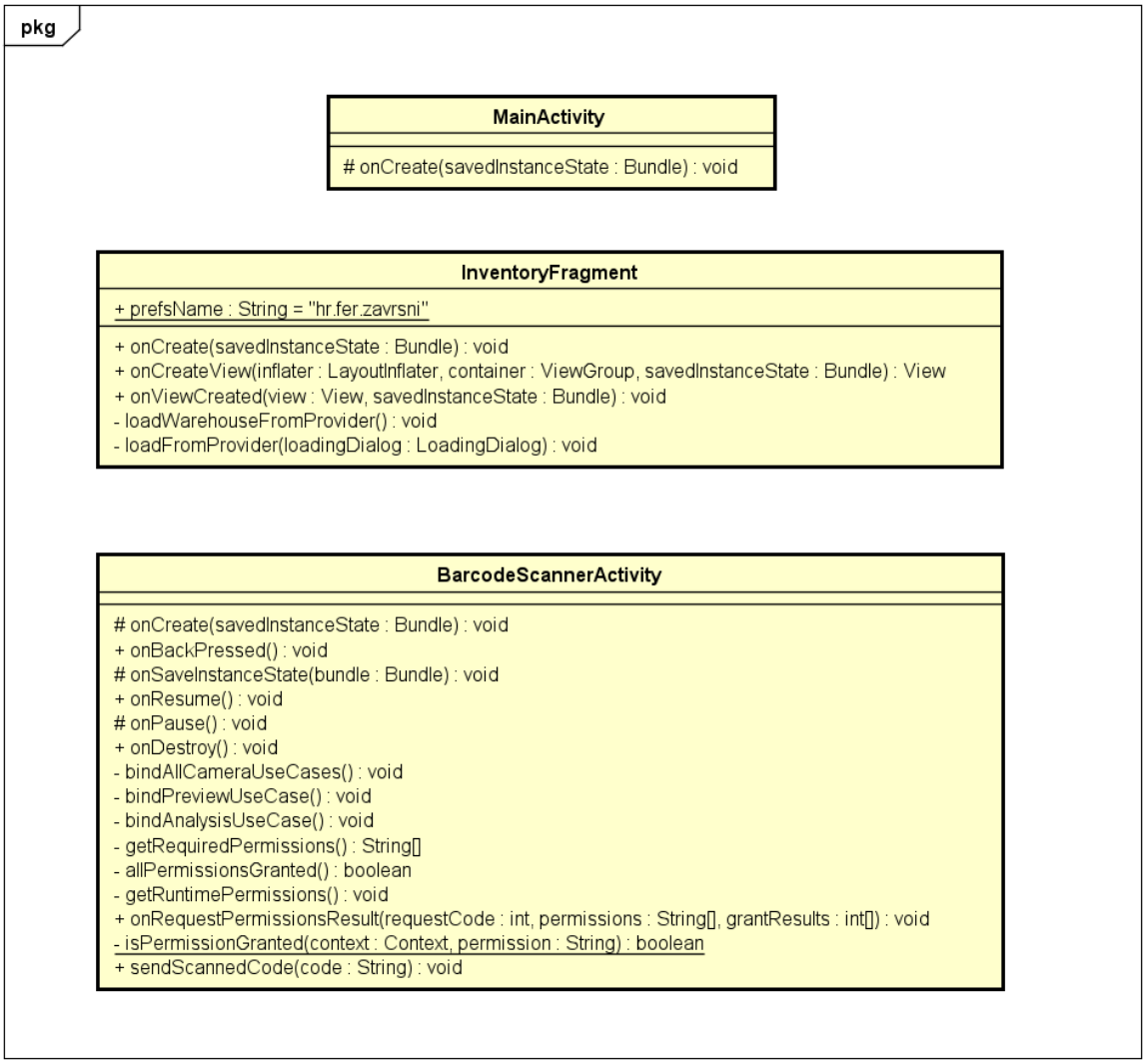
BarcodeScannerActivity je razred koji implementira metode **getRequiredPermissions(), allPermissionsGranted(), getRuntimePermissions(), bindAllCameraUseCases()**, **bindPreviewUseCase()**, **bindAnalysisUseCase()** i **sendScanned(String code)**. Metode **getRequiredPermissions()**, **allPermissionsGranted()** i **getRuntimePermissions()** služe kako bi se osigurao pristup kameri i njezinoj funckionalnosti automatskog fokusiranja. Metoda **bindPreviewUseCase()** služi kako bi se slika sa kamere prikazivala na korisnikovom zaslonu, metoda **bindAnalysisUseCase()** je metoda koja pokreće kod zadužen za detekciju barkodova i metoda **bindAllCameraUseCases()** koja samo poziva prve dvije metode. Za detekciju barkodova u slici metoda **bindAnalysisUseCase()** oslanja se na razred **BarcodeScannerProcessor** koji zapravo vrši analizu barkodova sa slike te rezultat tog procesiranja vraća pomoću metode **sendScanned(String code)**.

ScannedFragment koji se prikazuje nakon što **BarcodeScannerActivity** završi i vrati listu skeniranih barkodova koja se prikazuje u obliku tablice. **ScannedFragment**

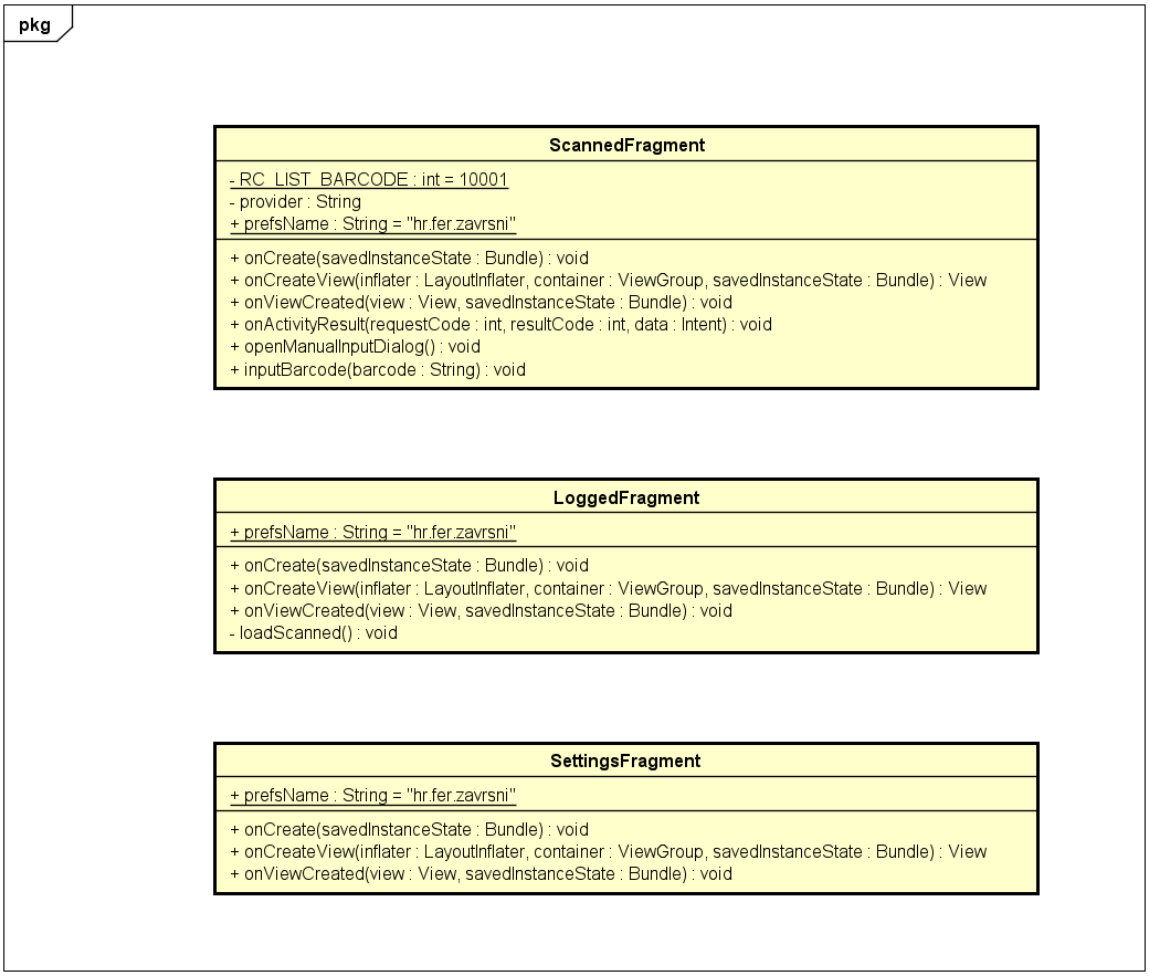
implementira i metode **openManualInputDialog()** i **inputBarcode()** za otvaranje prozora za ručan unos barkodova te njegovo dodavanje u listu skeniranih.

LoggedFragment implementira metodu **loadScanned()** koja sa poslužitelja dohvaća sve skenirane barkodove te ih prikazuje u obliku tablice.

SettingsFragment služi za promjenu poslužitelja na koji se šalju skenirani barkodovi te zahtjevi o trenutnom statusu skladišta.



Slika 1.10. Dijagram razreda – Android, prvi dio



Slika 1.11. Dijagram razreda – Android, drugi dio

2. Implementacija

U ovom poglavlju bit će opisana implementacija aplikacije i bit će prikazan programski kod zajedno s njegovim objašnjenjem. Poglavlje je podijeljeno na Springov i Androidov dio. Glavna komponenta jest Spring poslužitelj bez kojeg Android aplikacija ne može funkcionirati te on mora biti pokrenut i dostupan na mreži dok je Android aplikacija u upotrebi. Za razvoj Android aplikacije korišteni su vodiči sa Googleove stranice [4], dok su za razvoj Spring poslužitelja korišteni materijali tvrtke CROZ [8] te službena Spring dokumentacija [7]. Prilikom oblikovanja cijelog projekta korišteni su oblikovni obrasci kako bi se ubrzao proces razvoja softvera koristeći već testirane i dokazane strategije oblikovanja [6].

2.1. Spring

Razred **ScanController** sadrži anotaciju **@RestController** koja označava kako se radi o nadzorniku koji koristi arhitekturni stil **REST**. **REST** je arhitekturni stil koji propisuje standarde kojim se obavlja komunikacija između računala. **ScanController** korisnicima vraća objekte tipa **ResponseEntity** što je razred koji implementira **Spring** i omogućuje veću kontrolu i deskriptivnost programeru kod odgovaranja na HTTP zahtjeve. Spomenuti nadzornik implementira dvije metode koje koriste uslugu **ScanService** kako bi dohvatio listu objekata skeniranih barkodova iz baze podataka i vratio ju korisniku te metodu koja dodaje poslane objekte skeniranih barkodova u bazu podataka. Kod nadzornika vidljiv je na Slika 2.1.

```java
package hr.fer.zavrsni.controllers;

import java.util.Collections;
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import hr.fer.zavrsni.models.Scan;
import hr.fer.zavrsni.services.ScanService;

@RestController
@RequestMapping("/scan")
public class ScanController {
	@Autowired
	private ScanService scanService;

	@PostMapping("/add")
	public ResponseEntity<?> addScanned(@RequestBody List<Scan> scanned){

		return new ResponseEntity<>(scanService.addScans(scanned),HttpStatus.OK);
	}

	@GetMapping("/logs")
	public ResponseEntity<?> getLogs(){
		List<Scan> ret = scanService.listAll();
		Collections.reverse(ret);
		return new ResponseEntity<>(ret,HttpStatus.OK);
	}
}
```

Slika 2.1. Kod – ScanController

ScanService je servis koji implementira četiri metode koje su već spomenute na dijagramu razreda (Slika 2.2.). Metoda **listAll()** poziva metodu **findAll()** nad objektom tipa **ScanRepository** koji proslijeđuje upit u bazu podataka te vraća rezultat tog upita pozivatelju. Metoda **addScans(List<Scan> scans)** iterira po listi scans te jedan po jedan object **Scan** sprema u bazu podataka pozivajući metodu **save** razreda **ScanRepository**. Metoda **countByBarcode(String barcode)** oslanja se na istoimenu metodu razreda **ScanRepository** i broji koliko ima redaka u tablici **Scan** sa tim barkodom i posljednja metoda **countBarcodeAndWarehouse(String barcode, String warehouse)** koja poziva

metodu **countByBarcodeAndWarehouse(String barcode, String warehouse)** razreda **ScanRepository** i vraća korisniku koliko redaka u tablici **Scan** postoji s tim skladištem i tim barkodom.

```
1  package hr.fer.zavrsni.services.impl;
2
3  import java.util.ArrayList;
4  import java.util.List;
5  import org.springframework.beans.factory.annotation.Autowired;
6  import org.springframework.stereotype.Service;
7  import hr.fer.zavrsni.models.Scan;
8  import hr.fer.zavrsni.repository.ScanRepository;
9  import hr.fer.zavrsni.services.ScanService;
10
11 @Service
12 public class ScanServiceImpl implements ScanService{
13
14         @Autowired
15         private ScanRepository scanRepository;
16
17         @Override
18         public List<Scan> listAll() {
19                 return scanRepository.findAll();
20         }
21
22         @Override
23         public List<Scan> addScans(List<Scan> scans) {
24                 List<Scan> successfullyAdded = new ArrayList<Scan>();
25                  for(Scan sc: scans) {
26                         successfullyAdded.add(scanRepository.save(sc));
27                 }
28                 return successfullyAdded;
29         }
30
31         @Override
32         public Integer countByBarcode(String barcode) {
33                 return scanRepository.countByBarcode(barcode);
34         }
35
36         @Override
37         public Integer countByBarcodeAndWarehouse(String barcode, String warehouse) {
38                 return scanRepository.countByBarcodeAndWarehouse(barcode,warehouse);
39         }
40
41 }
```

Slika 2.2. Kod – ScanService

ScanRepository je sučelje koje proširuje sučelje **JpaRepository** i parametrizirano je po objektu **Scan** i broju koji predstavlja jedinstveni identifikator tog objekta u bazi podataka.

Sučelje nudi dvije metode: **countByBarcode(String barcode)** i **countByBarcodeAndWarehouse(String barcode, String warehouse),** čija je funkcionalnost objašnjena u odlomku iznad. Budući da sučelje **ScanRepository** proširuje **JpaRepository**, moguće je pisati metode poput **countByAtribut** bez potrebe za pisanjem SQL upita, jer je ta funkcionalnost već implementirana u sučelje **JpaRepository**. Kod sučelja vidljiv je na Slika 2.3.

```
1  public interface ScanRepository extends JpaRepository<Scan,Integer>{
2
3          Integer countByBarcode(String barcode);
4
5          Integer countByBarcodeAndWarehouse(String barcode,String warehouse);
6
7
8  }
```

Slika 2.3. Kod – ScanRepository

2.2. Android

Kod Android aplikacije najvažniji dio jest kod koji detektira i parsira barkodove u slici. Za detekciju i parsiranje barkodova korištena je Googleova tehnologija ML kit SDK. U stvari, korišten je jedan mali dio koji se odnosi na analizu slika, već istrenirani model s vrlo velikom točnošću za detekciju barkodova u slikama. Kako bi taj model mogao analizirati sliku, potrebno je prvo te slike nekako slikati, a za to je potrebno implementirati razred koji će pristupiti kameri i pokrenuti snimanje. Taj razred se u ovoj implementaciji zove **CameraXViewModel** i oslanja se na noviju verziju Googleovog API-a za rukovanje kamerom. Kod razreda pisan je pomoću članka [2] i koda u GitHub repozitoriju [3] te sam kod možemo vidjeti na Slika 2.4.

```java
package hr.fer.zavrsni.inventoryapp.barcode;

import android.app.Application;
import androidx.camera.lifecycle.ProcessCameraProvider;
import androidx.lifecycle.AndroidViewModel;
import androidx.lifecycle.LiveData;
import androidx.lifecycle.MutableLiveData;
import androidx.annotation.NonNull;
import androidx.core.content.ContextCompat;
import android.util.Log;
import com.google.common.util.concurrent.ListenableFuture;
import java.util.concurrent.ExecutionException;

public final class CameraXViewModel extends AndroidViewModel {

    private static final String TAG = "CameraXViewModel";
    private MutableLiveData<ProcessCameraProvider> cameraProviderLiveData;

    public CameraXViewModel(@NonNull Application application) {
        super(application);
    }

    public LiveData<ProcessCameraProvider> getProcessCameraProvider() {
        if (cameraProviderLiveData == null) {
            cameraProviderLiveData = new MutableLiveData<>();

            ListenableFuture<ProcessCameraProvider> cameraProviderFuture =
                    ProcessCameraProvider.getInstance(getApplication());
            cameraProviderFuture.addListener(
                    () -> {
                        try {
                            cameraProviderLiveData.setValue(cameraProviderFuture.get());
                        } catch (ExecutionException | InterruptedException e) {
                            // Handle any errors (including cancellation) here.
                            Log.e(TAG, "Unhandled exception", e);
                        }
                    },
                    ContextCompat.getMainExecutor(getApplication()));
        }

        return cameraProviderLiveData;
    }
}
```

Slika 2.4. Kod – CameraXViewModel

Sada kada se slike učitavaju, treba ih obraditi i provjeriti postoji li barkod na njima i, ako da, analizirati njegov sadržaj i spremiti ga negdje. Za tu svrhu korišten je dio koda koji je već implementiran od strane Googlea, jedino što je bilo potrebno implementirati bile su metode **detectInImage**, **onSuccess** i **onFailure**. Metoda **detectInImage** oslanjala se na

već spomenuti naučeni model strojnog učenja za detekciju barkodova na slici, dok su metode **onSuccess** i **onFailure** pozivane ovisno o rezultatima metode **detectInImage**. U implementaciji metode **onSuccess** provjerava se nalazi li se barkod u sredini ekrana i zatim, ukoliko se tamo nalazi, poziva se sučelje koje skenirani barkod sprema u listu za kasnije slanje na poslužitelj. Metoda **onFailure** se poziva ukoliko se dogodi pogreška prilikom detekcije i njezina implementacija samo zapisuje u dnevnik (eng. *log*) da se greška dogodila. Kod spomenutog razreda vidljiv je na slikama: Slika 2.5, Slika 2.6, Slika 2.7.

```java
package hr.fer.zavrsni.inventoryapp.barcode;

import android.animation.ValueAnimator;
import android.app.Activity;
import android.content.Context;
import android.graphics.RectF;
import android.util.Log;

import androidx.annotation.NonNull;

import com.google.android.gms.tasks.Task;
import com.google.mlkit.vision.barcode.Barcode;
import com.google.mlkit.vision.barcode.BarcodeScanner;
import com.google.mlkit.vision.barcode.BarcodeScannerOptions;
import com.google.mlkit.vision.barcode.BarcodeScanning;
import com.google.mlkit.vision.common.InputImage;

import java.util.List;

import hr.fer.zavrsni.inventoryapp.R;

/**
 * Barcode Detector.
 */
public class BarcodeScannerProcessor extends VisionProcessorBase<List<Barcode>> {

    private static final String TAG = "BarcodeProcessor";

    private static final String MANUAL_TESTING_LOG = "BarcodeProcessor_LOG";

    private final BarcodeScanner barcodeScanner;

    private ExchangeScannedData exchangeScannedData;

    private CameraReticleAnimator cameraReticleAnimator;

    private GraphicOverlay graphicOverlay;

    public BarcodeScannerProcessor(Context context, ExchangeScannedData exchangeScannedData) {
        super(context);

        BarcodeScannerOptions options = new BarcodeScannerOptions.Builder()
                .build();
```

Slika 2.5. Kod – BarcodeScannerProcessor, prvi dio

```
44
45          barcodeScanner = BarcodeScanning.getClient(options);
46          this.exchangeScannedData = exchangeScannedData;
47          graphicOverlay=((Activity)context).findViewById(R.id.graphic_overlay);
48          this.cameraReticleAnimator = new CameraReticleAnimator(graphicOverlay);
49      }
50
51      @Override
52      public void stop() {
53          super.stop();
54          barcodeScanner.close();
55      }
56
57      @Override
58      protected Task<List<Barcode>> detectInImage(InputImage image) {
59
60          return barcodeScanner.process(image);
61
62      }
63
64      @Override
65      protected void onSuccess(
66              @NonNull List<Barcode> barcodes, @NonNull GraphicOverlay graphicOverlay) {
67
68          if (barcodes.isEmpty()) {
69              Log.v(MANUAL_TESTING_LOG, "No barcode has been detected");
70          }
71          Barcode barcodeInCenter = null;
72          for (Barcode barcode : barcodes) {
73              RectF box = graphicOverlay.translateRect(barcode.getBoundingBox());
74              if (box.contains(graphicOverlay.getWidth() / 2f, graphicOverlay.getHeight() / 2f)) {
75                  barcodeInCenter = barcode;
76                  break;
77              }
78          }
79          graphicOverlay.clear();
80          if(barcodeInCenter==null){
81              graphicOverlay.add(new BarcodeReticleGraphic(graphicOverlay, cameraReticleAnimator))
82              cameraReticleAnimator.start();
83          }else{
84              exchangeScannedData.sendScannedCode(barcodeInCenter.getRawValue());
85
86          }
87          graphicOverlay.invalidate();
88
```

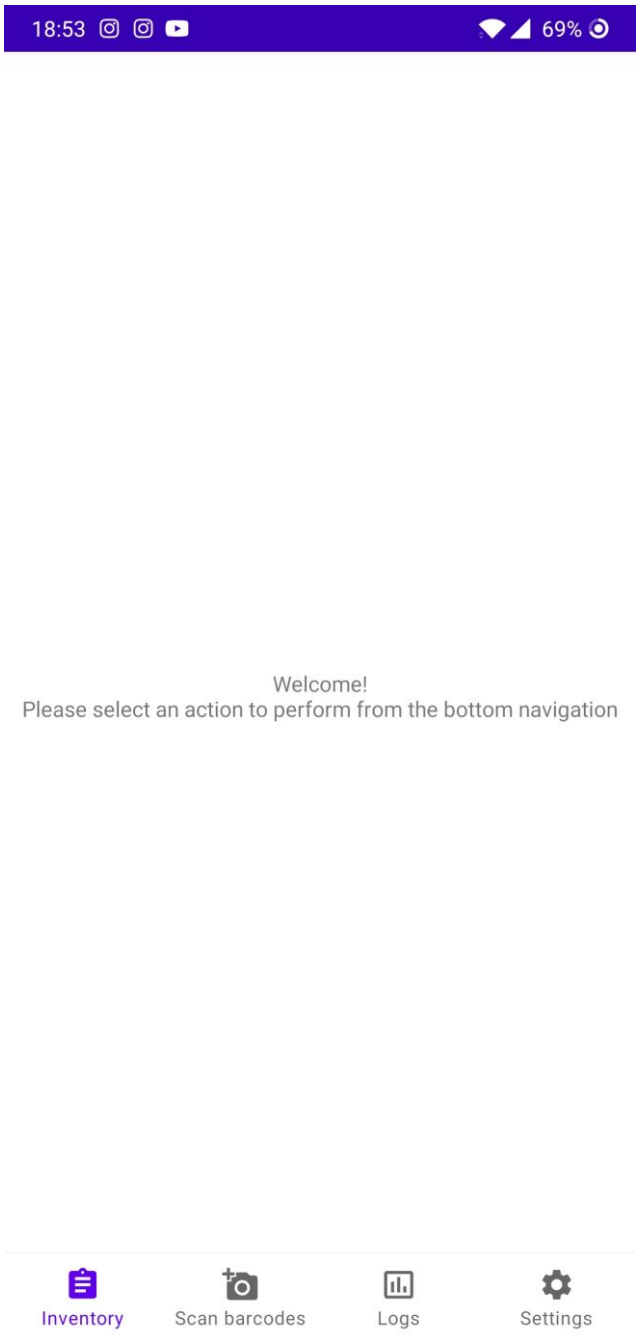
Slika 2.6. Kod – BarcodeScannerProcessor, drugi dio

```
89          }
90
91          @Override
92          protected void onFailure(@NonNull Exception e) {
93              Log.e(TAG, "Barcode detection failed " + e);
94          }
95
96      }
```

Slika 2.7. Kod – BarcodeScannerProcessor, treći dio

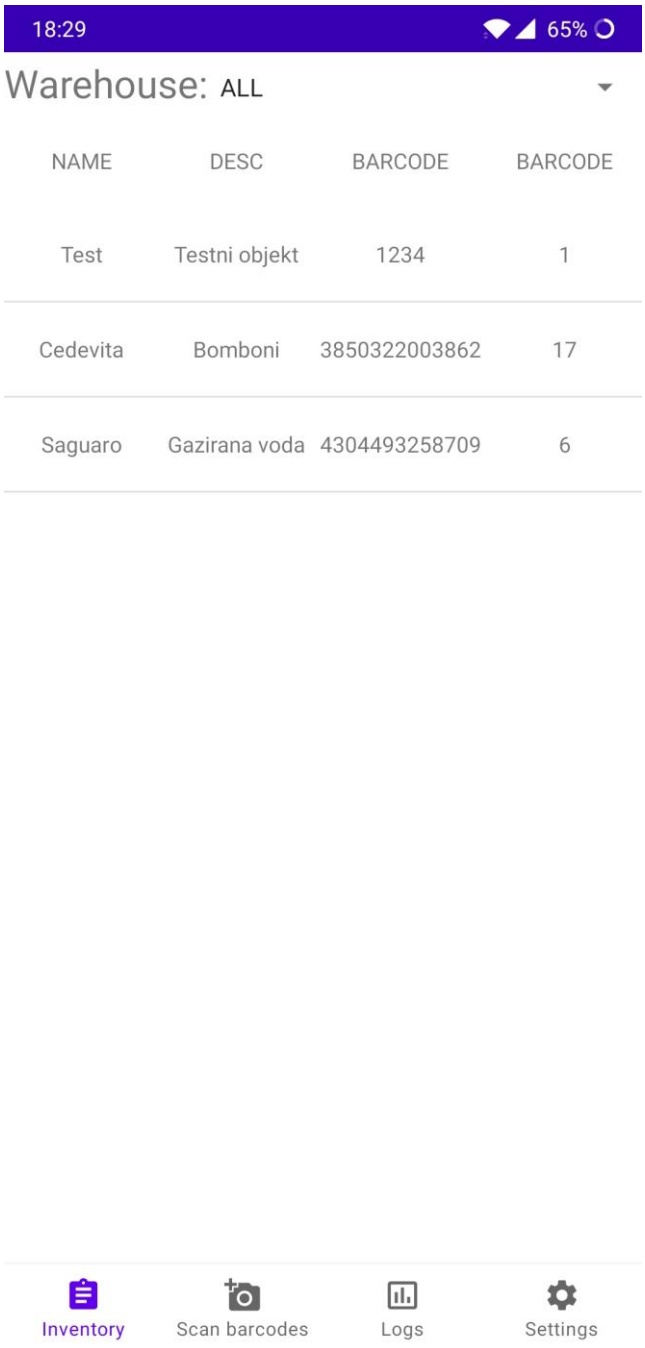
2.3. Izgled aplikacije

Kada se pokrene aplikacija, korisniku se najprije prikaže početni zaslon na kojem piše poruka dobrodošlice i sučelje za navigaciju s četiri opcije, Slika 2.8. Korisnik pritiskom na odgovarajuću stavku otvara aktivnost koja obavlja određenu funkcionalnost.



Slika 2.8. Aplikacija – početni zaslon

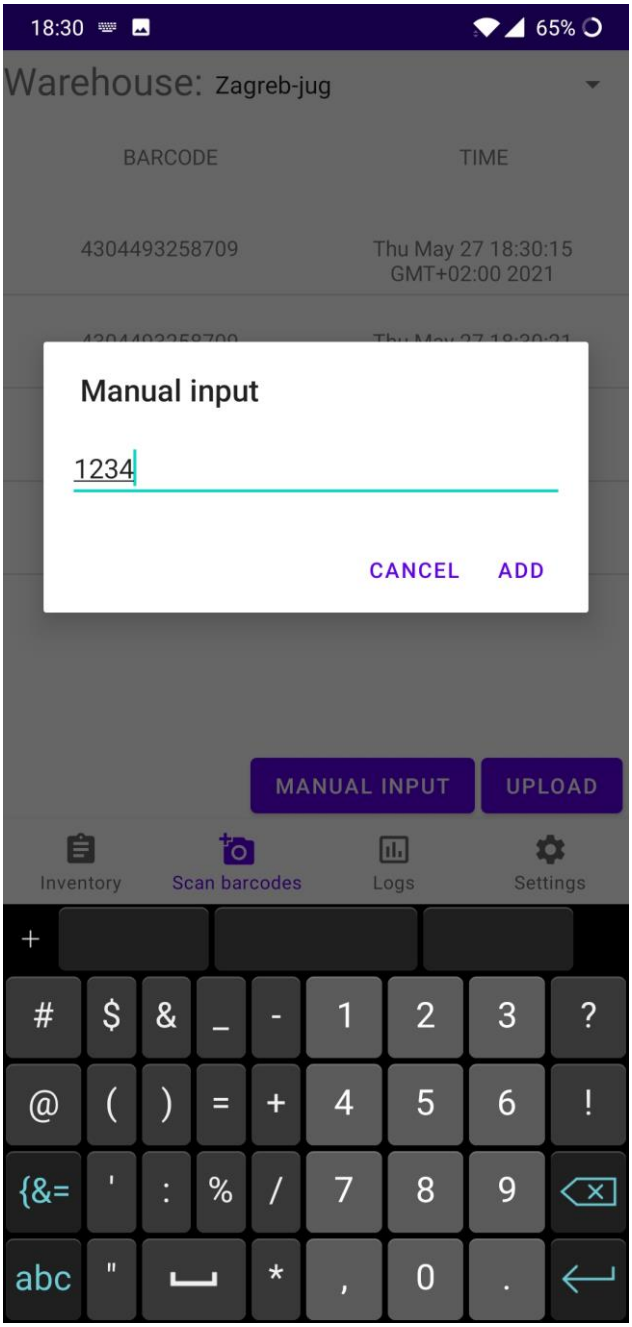
Prva opcija je pregled inventure, na tom ekranu pokazuje se tablica artikala te njihovih količina na odabranom skladištu, Slika 2.9.



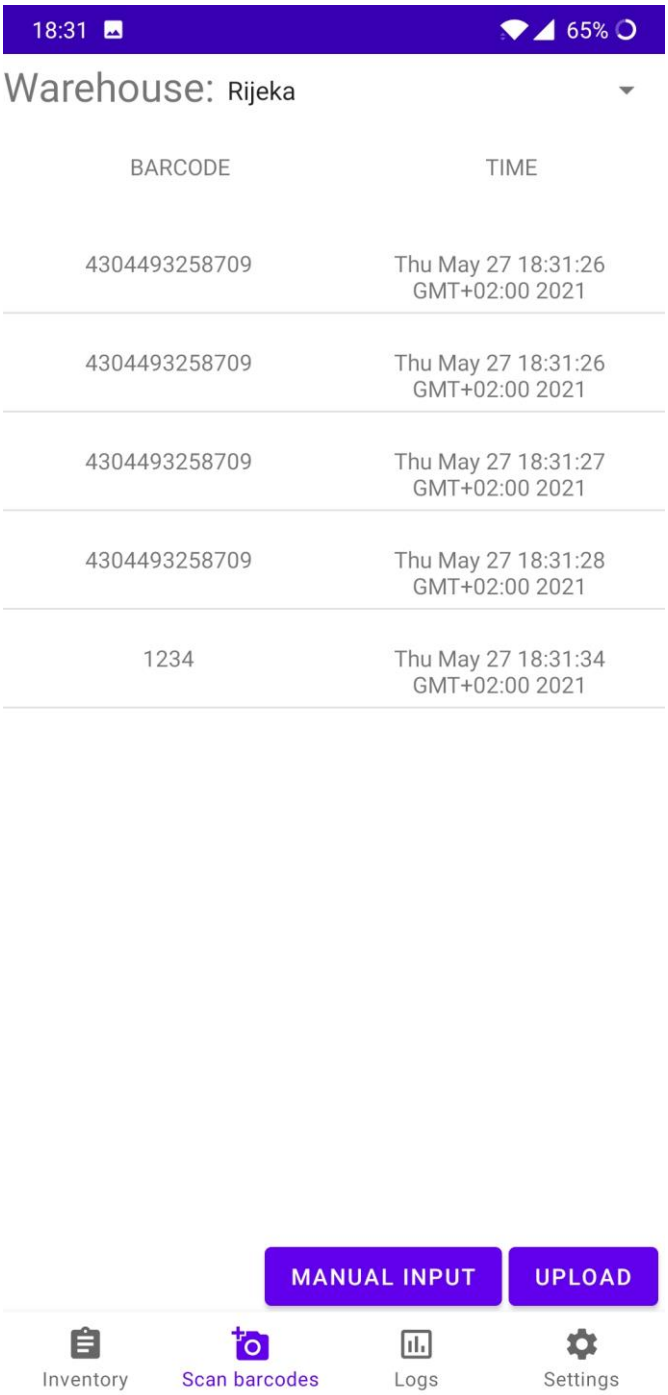
Slika 2.9. Aplikacija – pregled stanja skladišta

Druga opcija je skeniranje barkodova. Prilikom otvaranja ove aktivnosti odmah se od poslužitelja traži lista skladišta te se tek nakon što se dobije pozitivan odgovor od poslužitelja pali kamera i pokreće skeniranje barkodova, Slika 2.10. Nakon što korisnik završi skeniranje, pojavljuje se tablica sa skeniranim barkodovima i opcije za ručan unos (Slika 1.11) i opcije za slanje liste barkodova na poslužitelj (Slika 2.12).

Slika 2.10. Aplikacija – skeniranje barkodova



Slika 2.11. Aplikacija – ručan unos barkoda

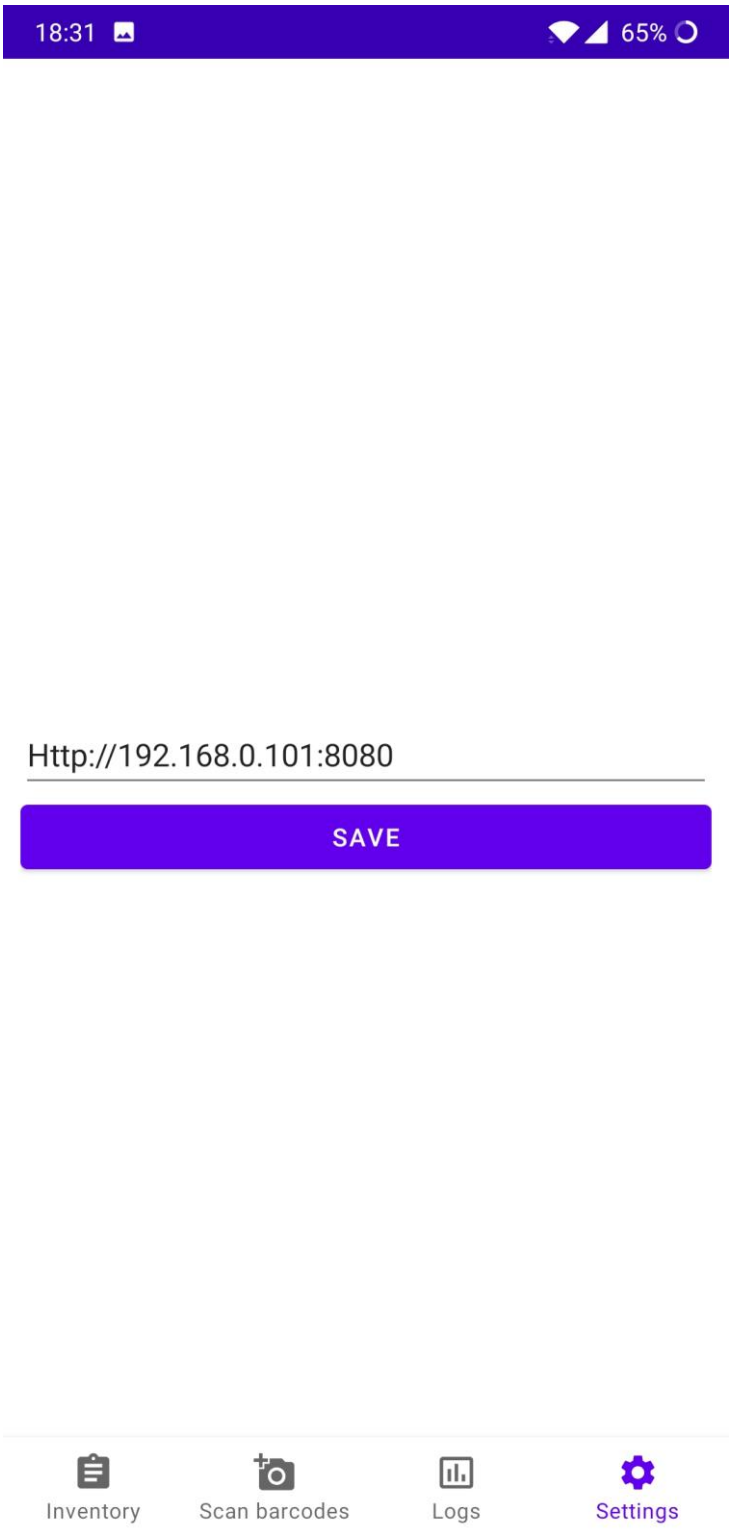


Slika 2.12. Aplikacija – pregled i prijenos na poslužitelj barkodova

Treća opcija jest pregled dnevnika ažuriranja, gdje se s poslužitelja dohvaća lista svih skeniranih barkodova i vremena skeniranja te se ti podaci prikazuju tablično, sortirani po vremenu skeniranja, Slika 2.13.

Slika 2.13. Aplikacija – pregled dnevnika ažuriranja

Posljednja, četvrta opcija je opcija postavljanja poslužitelja. Sastoji se od jedne forme gdje korisnik unosi adresu poslužitelja na koji se šalju zahtjevi, Slika 2.14.



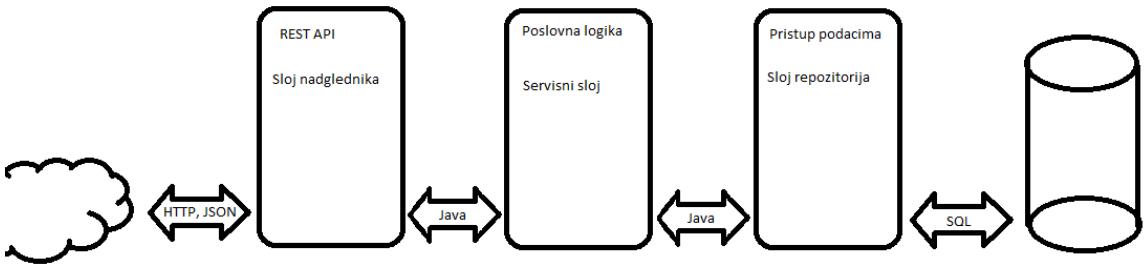
Slika 2.14. Aplikacija – postavljanje adrese poslužitelja

3. Korištene tehnologije

3.1. Spring Boot

Za potrebe poslužitelja korišten je razvojni okvir Spring. Razvojni okvir Spring pisan je u programskom jezika Java i namijenjen je za izgradnju i konfiguraciju modernih aplikacija na svim platformama. Spring se sastoji od mnogo manjih komponenti od kojih ovaj projekt koristi Spring Boot za izgradnju usluga putem REST API-ja.

Radni okvir Spring Boot temelji se na troslojnoj arhitekturi gdje su slojevi: sloj nadglednika (u nastavku teksta: kontroler), sloj usluge i sloj repozitorija. U sloju kontrolera nalazi se REST API (eng. *Representational state transfer*), njegova uloga je da funkcionalnost izlaže na web kao REST uslugu, sastoji se od više kontrolera gdje svaki kontroler prima i odgovara na HTTP zahtjeve, na temelju onoga što je primljeno u zahtjevu šalje podatke sloju usluge koji onda obrađuje taj zahtjev te vraća rezultat koji se zatim šalje kao odgovor na taj HTTP zahtjev. U sloju usluge nalazi se poslovna logika. Ovaj sloj sadrži logiku koja je potrebna za izvršavanje poslova. Posljednji sloj, sloj repozitorija ima svrhu pristupa podacima koji ne moraju nužno biti zapisani u bazi podataka. Prikaz troslojne arhitekture i načina komunikacije između komponenata dan je na sljedećoj slici (Slika 3.1).



Slika 3.1. Izgled troslojne arhitekture

3.2. Android

Za klijentski dio razvijena je aplikacija za operacijski sustav **Android**. Operacijski sustav **Android** temelji se na Linuxovoj jezgri te je dizajniran primarno za pametne telefone. Danas je jedan od najpopularnijih mobilnih operacijskih sustava. Razvijena aplikacija pisana je u programskog jeziku **Java** pomoću razvojne okoline **Android Studio**.

3.3. Ostale tehnologije

IntelliJ IDE je razvojna okolina za razne programske jezike koji uključuju i programski jezik Java u kojem je poslužiteljski dio pisan.

Android Studio je također razvojna okolina koja služi za razvoj Android aplikacija, bazirana je na razvojnoj okolini IntelliJ i oblikovana je specifično za razvoj Android aplikacija.

Java je objektno orijentirani programski jezik koji, zbog činjenice da se sav kod izvršava na virtualnom Javinom stroju, omogućuje izvođenje na bilo kojem uređaju. Razvija ga tvrtka Oracle i koristi ga čak 2 milijarde uređaja.

Google ML Kit je mobilni SDK (eng. *Software Development Kit*) koji koristi već naučene modele strojnog učenja [1]. U okviru ML Kita moguće je na slici detektirati lice, pozu, objekte i omogućiti njihovo praćenje, detektirati barkod i prepoznati tekst.

PostgreSQL je sustav za upravljanje bazom podataka otvorenog koda [5]. Izabran je zbog jednostavnosti postavljanja i korištenja.

Git je alat za upravljanje verzijama projekata te je korišten zajedno s besplatnom uslugom **GitHub**.

Astah je program koji je korišten za generiranje Use Case i Class dijagrama.

Zaključak

Izrađena aplikacija uspješno zadovoljava tražene funkcionalne zahtjeve i sastoji se od funkcionalnog korisničkog sučelja. Postoji mogućnost razvoja web aplikacije kojom bi se nadogradila funkcionalnost poslužitelja te time omogućio pregled, unos i ažuriranje na računalu. Aplikacija je razvijena s ciljem olakšavanja procesa inventure te ima potencijal biti korisna poslodavcima.

Radom je stečeno iskustvo u izradi aplikacija na operacijskom sustavu Android i korištenju radnog okvira Spring.

Literatura

[1] Google, *Barcode scanning*, https://developers.google.com/ml-kit/guides, pristupljeno 27.5.2021.

[2] E. Zeeshan, *How to use and configure CameraX in Android applications*, https://www.zeeshanelahi.com/2020/07/how-to-use-and-configure-camerax-in-android-applications/, pristupljeno 27.5.2021.

[3] E. Zeeshan, *Barcode Scanning with MLKit and CameraX Demo*, https://github.com/zeeshan-elahi/BarcodeScannerAndCameraXDemo, pristupljeno 27.5.2021.

[4] Google developers, *Developer guides,* https://developer.android.com/guide, pristupljeno 27.5.2021.

[5] The PostgreSQL Global Development Group, *PostgreSQL Documentation*, https://www.postgresql.org/docs/ , pristupljeno 27.5.2021.

[6] S. Šegvić, *Oblikovni obrasci u programiranju*, http://www.zemris.fer.hr/~ssegvic/ooup/pubs/ooup3MorePatterns.pdf, pristupljeno 27.5.2021.

[7] Pivotal, *Spring Framework documentation*, https://docs.spring.io/spring-framework/docs/current/reference/html/, pristupljeno 27.5.2021.

[8] H. Šimić, Croz, „*Izrada REST backenda u Spring Bootu*”, https://gitlab.com/hrvojesimic/progi-project-teams-backend/-/blob/master/education.md, pristupljeno 27.5.2021.

[9] A. Jović, N. Frid, D. Ivošević: Procesi programskog inženjerstva, e-skripta, FER, 2020. https://www.fer.unizg.hr/predmet/proinz/literatura, pristupljeno 27.5.2021.

Sažetak

Naslov: Mobilna aplikacija za provođenje inventure na temelju detekcije barkoda iz slike

Sažetak: U radu „Mobilna aplikacija za provođenje inventure na temelju detekcije barkoda iz slike“ prikazan je način izrade aplikacije i njezina implementacija. Prikazane su funkcionalnosti aplikacije, opisane tehnologije i alati koji su korišteni prilikom razvoja.

Ključne riječi: Spring Boot, Spring Framework, Android, Java, inventura, mobilna aplikacija

Summary

Title: Mobile Application for Stocktaking Based on Barcode Detection from Image

Summary: In this thesis the process of creation and implementation of a stocktaking app is explained and shown in detail, the applications functionality is shown as well as the tools and technologies used during its development.

Keywords: Spring Boot, Spring Framework, Android, Java, Stocktaking, mobile application