MOBILNE APLIKACIJE

Vežbe 4

GUI II

2024/2025

Sadržaj

1. ViewBinding

ViewBinding je funkcionalnost u Androidu koja olakšava interakciju s prikazima. Generiše binding klasu za svaki XML fajl sa rasporedom elemenata u vašoj aplikaciji. Ova binding klasa omogućava direktni pristup elementima prikaza (views) u fajlu rasporeda, bez potrebe za korišćenjem *findViewById* metode.

Da bismo koristili ViewBinding u Android aplikaciji, neophodno je da omogućimo ViewBinding u *build.gradle* fajlu.

```
34      buildFeatures {
35          viewBinding true
36      }
```

Slika 1. build.gradle - omogućavanje ViewBinding-a

Za svaki XML fajl sa rasporedom elemenata, generiše se odgovarajuća *ViewBinding* klasa. Na primer, ako je vaš fajl rasporeda nazvan *activity_home.xml,* generisana klasa će biti *ActivityHomeBinding*.

U vašoj Aktivnosti ili Fragmentu, napravite inflaciju rasporeda elemenata pomoću generisane *ViewBinding* klase:

```
56          binding = ActivityHomeBinding.inflate(getLayoutInflater());
57          setContentView(binding.getRoot());
58
```

Slika 2. Inflacija rasporeda elemenata pomoću generisane ViewBinding klase

```
59      binding.activityHomeBase.floatingActionButton.setOnClickListener(v -> {
60          Log.i( tag: "ShopApp",  msg: "Floating Action Button");
61          /*...*/
69          Intent intent = new Intent( packageContext: HomeActivity.this, CartActivity.class);
70          intent.putExtra( name: "title",  value: "Cart");
71          startActivity(intent);
72      });
73
74      drawer = binding.drawerLayout;
75      navigationView = binding.navView;
76      toolbar = binding.activityHomeBase.toolbar;
```

Slika 3. Pristupanje elementima korišćenjem binding objekta

2. Adapteri

Adapteri povezuju poglede i izvore podataka (podaci iz baze, sa interneta..).

Moguće je koristiti neke od predefinisanih adaptera (*BaseAdapter, ArrayAdapter, CursorAdapter*) ili napraviti *Custom Adapters,* adaptere koji povezuju proizvoljan pogled i izvor podataka.

ArrayAdapter povezuju *TextView* pogled (ili pogled koji sadrzi *TextView* pogled) i niz ili kolekciju. Automatski se poziva *toString()* metoda svakog objekta u nizu ili kolekciji i njena povratna vrednost se prikazuje u pogledu.

2.1 Pravljenje adaptera

Kreiramo novu klasu, *ProductListAdapter,* koja nasleđuje neki od postojećih adaptera. U našem primeru nasleđuje *ArrayAdapter* (slika 4), koji je sposoban da kao izvor podataka iskoristi listu ili niz. Dobijamo metode koje moramo da redefinišemo, da bi naš adapter ispravno radio.

```
34  public class ProductListAdapter extends ArrayAdapter<Product> {
35      private ArrayList<Product> aProducts;
36
37      public ProductListAdapter(Context context, ArrayList<Product> products){
38          super(context, R.layout.product_card, products);
39          aProducts = products;
40
41      }
```

Slika 4. Kreiranje adaptera

Metoda *getCount()* vraća ukupan broj elemenata u listi, koja treba da se prikaže.

```
45      @Override
46      public int getCount() {
47          return aProducts.size();
48      }
```

Slika 5. Metoda *getCount()*

Metoda *getItem(int position)* vraća pojedinačan element na osnovu njegove pozicije.

```
53      @Nullable
54      @Override
55      public Product getItem(int position) {
56          return aProducts.get(position);
57      }
```

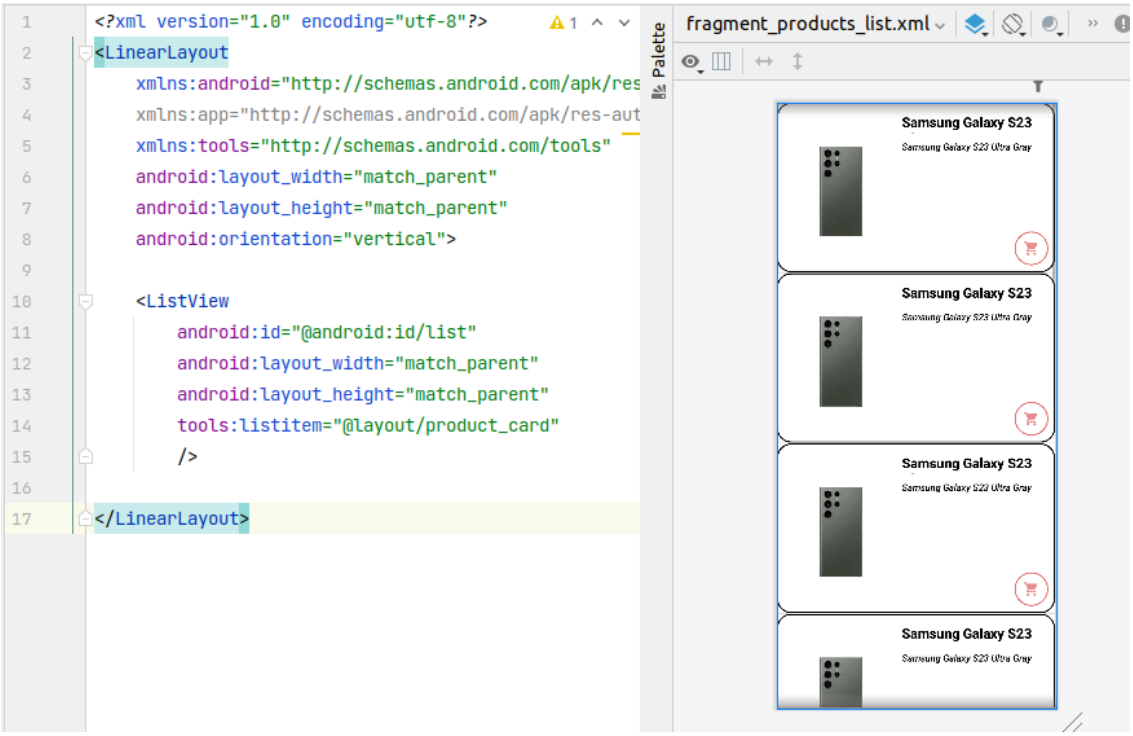
Slika 6. Metoda *getItem(int position)*

Metoda *getItemId(int position)* vraća jedinstveni identifikator.

```
64      @Override
65      public long getItemId(int position) {
66          return position;
67      }
```

Slika 7. Metoda *getItemId(int position)*

Metoda *getView* popunjava *ListView* sa podacima, tako što adapter, koji čuva listu elemenata, iterira kroz elemente i redom popunjava *ListView.* Adapter zna koliko iteracija treba da ima jer poseduje metodu *getCount().*



Slika 8. ListView komponenta

U metodi *getView position* je pozicija elementa u listi elemenata, koju čuva adapter, a *parent* je roditelj na kog će *view* biti postavljen. U prvom *if-u* vršimo inicijalizaciju *convertView*-a na kreirani layout *product_card,* koji će prikazivati sve elemente.

```
79      @NonNull
80      @Override
81      public View getView(int position, @Nullable View convertView, @NonNull ViewGroup parent) {
82          Product product = getItem(position);
83          if(convertView == null){
84              convertView = LayoutInflater.from(getContext()).inflate(R.layout.product_card,
85                      parent,  attachToRoot: false);
86          }
87          LinearLayout productCard = convertView.findViewById(R.id.product_card_item);
88          ImageView imageView = convertView.findViewById(R.id.product_image);
89          TextView productTitle = convertView.findViewById(R.id.product_title);
90          TextView productDescription = convertView.findViewById(R.id.product_description);
91
92          if(product != null){
93              imageView.setImageResource(product.getImage());
94              productTitle.setText(product.getTitle());
95              productDescription.setText(product.getDescription());
96              productCard.setOnClickListener(v -> {
97                  // Handle click on the item at 'position'
98                  Log.i( tag: "ShopApp",  msg: "Clicked: " + product.getTitle() + ", id: " +
99                          product.getId().toString());
100                 Toast.makeText(getContext(),  text: "Clicked: " + product.getTitle()  +
101                         ", id: " + product.getId().toString(), Toast.LENGTH_SHORT).show();
102             });
103         }
104
105         return convertView;
106     }
107 }
```

Slika 9. Metoda *getView*

Kao rezultat dobijamo prikazanu listu svih proizvoda (slika 10).

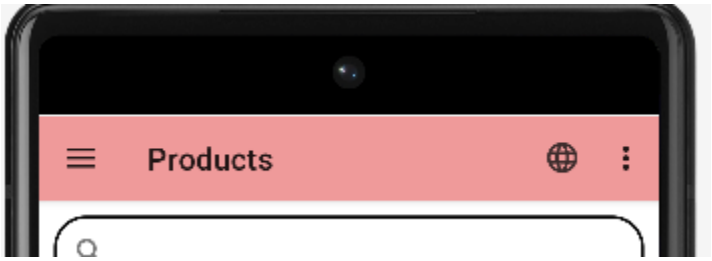
Slika 10. Lista proizvoda

3. Toolbar

Toolbar je element GUI-a koji se uglavnom nalazi na vrhu ekrana i obezbedjuje navigaciju, izvršavanje akcija, promenu pogleda, *branding* aplikacije.

3.1 Pravljenje toolbar-a

Na slici 11 se nalazi primer *toolbar*-a.



Slika 11. Primer *toolbar-a*

Toolbar se nalazi unutar *AppBarLayout*-a, tj. unutar elementa:

<com.google.android.material.appbar.AppBarLayout>.

Otvoriti *activity_home_base.xml* layout. *AppBarLayout* je vertikalni *LinearLayout* koji obezbeđuje mnoga svojstva *Material design-a. AppBarLayout layout* se koristi kao dete *CoodrinatorLayout*-a.

Material design je skup principa za vizuelni dizajn, dizajn pokreta i dizajn interakcija. Sve aplikacije koje su dizajnirane po ovim principima pružaju korisnicima konzistentno iskustvo i obezbeđuju da korišćenje aplikacija bude intuitivno.

Toolbar možemo ručno da definišemo ili u *Design* režimu da ga prevučemo iz palete (Containers > Toolbar) element *<androidx.appcompat.widget.Toolbar>* (slika 12).

```
10      <!-- AppBarLayout and Toolbar   -->
11      <com.google.android.material.appbar.AppBarLayout
12          android:id="@+id/app_bar"
13          android:layout_width="match_parent"
14          android:layout_height="wrap_content"
15          app:layout_constraintEnd_toEndOf="parent"
16          app:layout_constraintStart_toStartOf="parent"
17          app:layout_constraintTop_toTopOf="parent"
18          app:layout_constraintBottom_toBottomOf="parent"
19          app:layout_constraintVertical_bias="0.001"
20          android:theme="@style/AppTheme.AppBarOverlay">
21
22          <androidx.appcompat.widget.Toolbar
23              android:id="@+id/toolbar"
24              android:layout_width="match_parent"
25              android:layout_height="?attr/actionBarSize"
26              android:background="?attr/colorPrimary"
27              app:popupTheme="@style/AppTheme.PopupOverlay"
28              app:layout_scrollFlags="scroll|enterAlways">
29  <!--            zajedno s sadržajem (scroll) i
30              uvijek biti vidljiv (enterAlways)-->
31
32          </androidx.appcompat.widget.Toolbar>
33
34      </com.google.android.material.appbar.AppBarLayout>
```

Slika 12. Kreiranje *Toolbar*-a

Sledeći korak je da kreiramo klasu HomeActivity.java, koja nasleđuje *AppCompatActivity.* U metodi *onCreate()* dobavljamo *toolbar* (linija 76) i pozivamo metodu *setSupportActionBar()* kojoj prosleđujemo dobavljeni *toolbar* (slika 13). ActionBar je standardni element korisničkog interfejsa u Android aplikacijama koji prikazuje naslov aplikacije, ikone i druge korisničke akcije na vrhu ekrana.

```
76          toolbar = binding.activityHomeBase.toolbar;
77          // Postavljamo toolbar kao glavnu traku za ovu aktivnost
78          setSupportActionBar(toolbar);
79          // Dobavljamo referencu na glavnu traku za ovu aktivnost
80          actionBar = getSupportActionBar();
81          if(actionBar != null){
82              // postavlja prikazivanje "strelice prema nazad" (back arrow)
83              // kao indikatora navigacije na lijevoj strani Toolbar-a.
84              actionBar.setDisplayHomeAsUpEnabled(false);
85              // postavlja ikonu koja se prikazuje umjesto strelice prema nazad.
86              // U ovom slučaju, postavljena je ikona hamburger iz drawable resursa (ic_hamburger).
87              actionBar.setHomeAsUpIndicator(R.drawable.ic_hamburger);
88              //vo omogućuje da se klikom na gumb 'home' na Toolbar-u
89              // aktivira povratak na prethodni zaslon.
90              actionBar.setHomeButtonEnabled(true);
91          }
```

Slika 13. Postavljenje *Toolbar*-a

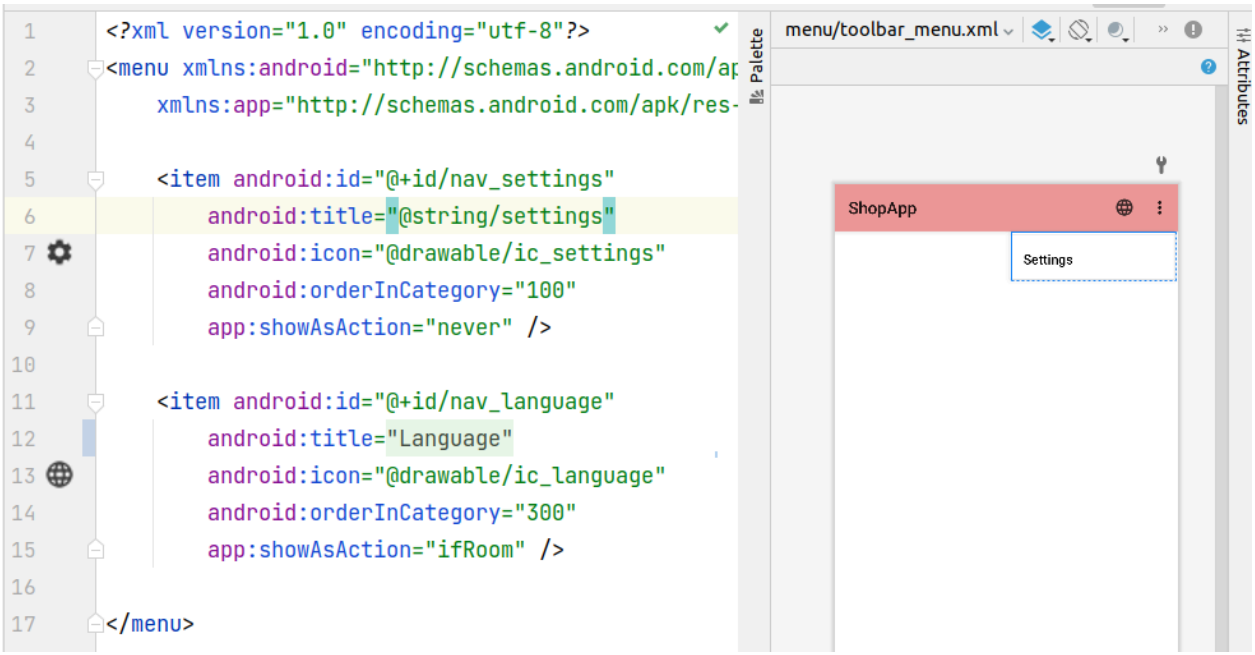
3.2 Postavljanje ikonica na toolbar i povezivanje sa akcijama

Da bismo mogli da izvršavamo akcije klikom na određene dugmiće iz *toolbar*-a, prvo kreiramo meni kao resurs. U ovom primeru kreirali smo meni sa 2 stavke*: settings* i *language* (slika 14) i to će biti ikonice koje će se prikazati na *toolbar*-u. Za svaku stavku smo definisali atribut *app:showAsAction*. Ovaj atribut govori kad i kako će stavka menija da bude prikazana.

Vrednosti koje ovaj atribut može da ima su:

- *ifRoom*
- *withText*
- *never*
- *always*
- *collapseActionView*

Ako atribut postavimo na *ifRoom* to znači da će ta stavka menija biti prikazana samo ako postoji dovoljno prostora u meniju. U suprotnom ova stavka će biti smeštena u padajući meni, kao što smo za stavku *settings* eksplicitno uradili sa postavljanjem atributa na vrednost *never*.




```xml
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/ap
    xmlns:app="http://schemas.android.com/apk/res-

    <item android:id="@+id/nav_settings"
        android:title="@string/settings"
        android:icon="@drawable/ic_settings"
        android:orderInCategory="100"
        app:showAsAction="never" />

    <item android:id="@+id/nav_language"
        android:title="Language"
        android:icon="@drawable/ic_language"
        android:orderInCategory="300"
        app:showAsAction="ifRoom" />

</menu>
```

Slika 14. *Menu* stavke

Da bismo kreirani meni postavili na *toolbar*, redefinišemo metodu *onCreateOptionsMenu* (slika 15) u *HomeActivity.java*.

```java
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    // menu.clear();
    // koristimo ako je nasa arhitekrura takva da imamo jednu aktivnost
    // i vise fragmentaa gde svaki od njih ima svoj menu unutar toolbar-a

    // Inflate the menu; this adds items to the action bar if it is present.
    getMenuInflater().inflate(R.menu.toolbar_menu, menu);
    return true;
}
```

Slika 15. Metoda *onCreateOptionsMenu*

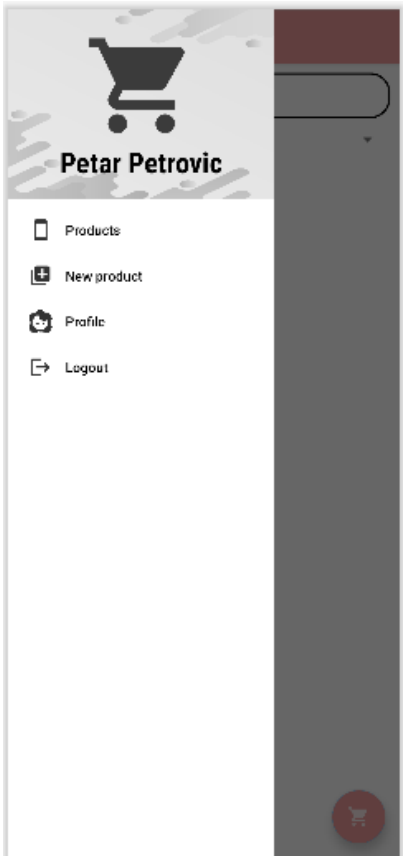
Za povezivanje dugmića iz *toolbar*-a sa akcijama, koristimo metodu *onOptionsItemSelected* (slika 16) u *HomeActivity.java*. Ova metoda vraća *MenuItem* na koji je korisnik kliknuo. Taj *MenuItem* u sebi sadrži i identifikator, uz pomoć god znamo tačno na koje dugme iz *toolbar*-a smo kliknuli.

```
204     @Override
205     public boolean onOptionsItemSelected(@NonNull MenuItem item) {
206         int id = item.getItemId();
207         switch (id) {
208             case R.id.nav_settings:
209                 Toast.makeText( context: HomeActivity.this,  text: "Settings", Toast.LENGTH_SHORT).show();
210                 break;
211             case R.id.nav_language:
212                 Toast.makeText( context: HomeActivity.this,  text: "Language", Toast.LENGTH_SHORT).show();
213                 break;
214         }
215         //...
218         navController = Navigation.findNavController( activity: this, R.id.fragment_nav_content_main);
219         //...
225         return NavigationUI.onNavDestinationSelected(item, navController) || super.onOptionsItemSelected(item);
226     }
```

Slika 16. Metoda *onOptionsItemSelected*

4. Navigation Drawer

Na slici 17 se nalazi primer *Navigation Drawer*-a koji ćemo kreirati.



Slika 17. Primer *Navigation Drawer*-a

Prvo deklarišemo *DrawerLayout* raspored (slika 18) i u njega smeštamo ostale poglede (*activity_home.xml*). Dodajemo jedan pogled koji sadrži glavni sadržaj aktivnosti *activity_home_base* i drugi pogled koji sadrži komponentu za navigaciju *NavigationView* (*nav_view)*.

NavigationView je deo Android korisničkog interfejsa koji se nalazi u *Android Design Support Library.* Obično se koristi unutar bočnog menija (navigation drawer) kako bi omogućio korisnicima navigaciju do različitih delova aplikacije.

Osnovna svrha *NavigationView*-a je da prikaže skup stavki unutar bočnog menija. Svaka stavka može biti povezana s određenom destinacijom unutar aplikacije, na primer, sa fragmentom ili drugom aktivnošću.

android:layout_gravity="start": Ovo postavlja gravitaciju NavigationView-a na start, što obično znači da će se bočni meni prikazivati sa leve strane ekrana.

android:fitsSystemWindows="true": Ovo se koristi da bi se osiguralo da se sadržaj prilagodi oko sistema prozora kako bi se izbegli prekidi ili isprekidani prikazi.

app:headerLayout="@layout/nav_header": Ovde se postavlja referenca na layout koji će se koristiti kao zaglavlje (header) unutar NavigationView-a. U ovom slučaju, @layout/nav_header se koristi za postavljanje zaglavlja koje se obično koristi za prikazivanje informacija o korisniku ili dodatnih opcija.

app:menu="@menu/nav_menu": Ovo postavlja referencu na menu resurs (nav_menu) koji sadrži stavke menija koje će biti prikazane unutar NavigationView-a. nav_menu obično definiše različite opcije menija koje korisnici mogu odabrati prilikom navigacije kroz aplikaciju.

```xml
<?xml version="1.0" encoding="utf-8"?>
<androidx.drawerlayout.widget.DrawerLayout
    xmlns:android="http://schemas.android.com/apk/res/android
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:fitsSystemWindows="true"
    tools:openDrawer="start"
    android:id="@+id/drawer_layout">

    <include
        android:id="@+id/activity_home_base"
        layout="@layout/activity_home_base"
        />

    <com.google.android.material.navigation.NavigationView
        android:id="@+id/nav_view"
        android:layout_width="wrap_content"
        android:layout_height="match_parent"
        android:layout_gravity="start"
        android:fitsSystemWindows="true"
        app:headerLayout="@layout/nav_header"
        app:menu="@menu/nav_menu" />
</androidx.drawerlayout.widget.DrawerLayout>
```

Slika 18. *DrawerLayout*

U *activity_home_base* layout-u kreiran je Toolbar ranije opisan i *activity_home_content* layout (slika 19).

```xml
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".activities.HomeActivity">

    <!-- AppBarLayout and Toolbar   -->
    <com.google.android.material.appbar.AppBarLayout...>

    <include layout="@layout/activity_home_content" />

    <com.google.android.material.floatingactionbutton.FloatingActionButton...>
</androidx.constraintlayout.widget.ConstraintLayout>
```

Slika 19. *Home Base layout*

Unutar *activity_home_content.xml* kreirana je komponenta androidx.navigation.fragment.NavHostFragment koja se koristi unutar Android aplikacija kao deo *Android Jetpack Navigation* komponente. Ova komponenta predstavlja kontejner za prikazivanje i upravljanje fragmentima (delovima korisničkog interfejsa) unutar koje se odvija navigacija između različitih ekrana (destinacija) unutar aplikacije.

Glavna svrha *NavHostFragment*-a je pružanje podrške za implementaciju navigacije kroz različite delove aplikacije, omogućavajući prelazak između različitih fragmenata ili destinacija koristeći Android Navigation komponentu.

app:defaultNavHost="true": Ova linija označava ovaj fragment kao "defaultni" NavHost. To znači da će ovaj fragment hvatati sve navigacijske događaje i destinacije.

app:navGraph="@navigation/base_navigation": Ovde se navodi koja XML datoteka sadrži grafički prikaz navigacije (NavGraph) unutar aplikacije koja se koristi za ovaj NavHostFragment (slika 20). *@navigation/base_navigation* referencira se na definiciju grafičke mape navigacije unutar resursa aplikacije (slika 21). Ova mapa definiše putanje i destinacije kroz koje se može navigirati unutar aplikacije.

```
15
16      <fragment
17          android:id="@+id/fragment_nav_content_main"
18          android:name="androidx.navigation.fragment.NavHostFragment"
19          android:layout_width="match_parent"
20          android:layout_height="match_parent"
21          app:layout_constrainedHeight="true"
22          app:layout_constraintBottom_toBottomOf="parent"
23          app:layout_constraintHorizontal_bias="0.0"
24          app:layout_constraintLeft_toLeftOf="parent"
25          app:layout_constraintRight_toRightOf="parent"
26          app:layout_constraintTop_toTopOf="parent"
27          app:layout_constraintVertical_bias="0.0"
28          app:defaultNavHost="true"
29          app:navGraph="@navigation/base_navigation" />
30
```

Slika 20. *NavHostFragment*

```
1   <?xml version="1.0" encoding="utf-8"?>
2   <navigation
3       xmlns:android="http://schemas.android.com/apk/res/android"
4       xmlns:app="http://schemas.android.com/apk/res-auto"
5       xmlns:tools="http://schemas.android.com/tools"
6       android:id="@+id/base_navigation"
7       app:startDestination="@+id/nav_products"
8       >
9       <fragment
10          android:id="@+id/nav_products"
11          android:name="com.example.shopapp.fragments.products.ProductsPageFragment"
12          android:label="Products"
13          tools:layout="@layout/fragment_products_page"
14          />
15
16      <fragment
17          android:id="@+id/nav_new"
18          android:name="com.example.shopapp.fragments.new_product.NewProductFragment'
19          android:label="New product"
20          tools:layout="@layout/fragment_new_product"
21          app:popUpTo="@+id/nav_products"
22          app:popUpToInclusive="true"
23          />
24
25      <fragment
26          android:id="@+id/nav_profile"
27          android:name="com.example.shopapp.fragments.profile.ProfileFragment"
28          android:label="Profile"
29          tools:layout="@layout/fragment_profile"
30          app:popUpTo="@+id/nav_products"
31          app:popUpToInclusive="true"
32          />
```

Slika 21. *Navigation*

ActionBarDrawerToggle se koristi za povezivanje i upravljanje navigation drawer-om unutar Android aplikacije. *ActionBarDrawerToggle* je klasa koja olakšava sinhronizaciju između navigation drawer-a i ActionBar-a (ili Toolbar-a) te omogućava otvaranje i zatvaranje navigation drawer-a putem ikone u ActionBar-u ili Toolbar-u.

NavigationController se koristi za upravljanje promenama destinacija unutar Android aplikacije korištenjem Android Navigation komponente. Pomoću *NavController* i *OnDestinationChangedListener*, prati se promena trenutne destinacije (screen-a/fragmenta) unutar aplikacije. Putem *navigate()* metode iz *NavigationController*-a moguće je navigirati se na specifičan fragment / screen.

Korišćenje Navigation Component-a nudi mnoge prednosti, uključujući automatsko upravljanje back stack-om, podršku za animacije i tranzicije, olakšano upravljanje argumentima i podacima između destinacija, kao i integraciju sa ostalim *Jetpack* komponentama poput *ViewModel*-a i *SafeArgs* za siguran prenos podataka.

AppBarConfiguration odnosi se na konfiguraciju ActionBar-a (ili Toolbar-a) u Android aplikaciji kako bi se omogućila navigacija koristeći Android Navigation komponentu. Takođe, postavlja se bočni meni (navigation drawer) u skladu sa konfiguracijom akcione trake i navigacije. Svaki ID menija prosleđuje se kao skup ID-ova jer svaki meni treba smatrati odredištima najvišeg nivoa.

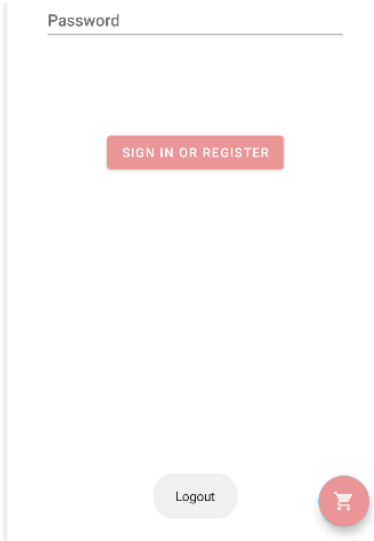
5. Toasts / Snackbars

Toast je *pop-up* poruka koja automatski nestaje posle određenog vremena. Ovakve poruke korisniku daju povratne informacije da je neka akcija izvršena ili eventualno da je došlo do neke greške.

Na slici se nalazi metoda koja dodeljuje akcije dugmićima iz menija (slika 22). Kada korisnik klikne na dugme *nav_new* u tom trenutku će se prikazati *toast* sa porukom *„New product“* (slika 23), a kada klikne na dugme *nav_logoout* prikazaće se poruka *„Logout“*.

```
122             if (!isTopLevelDestination) {
123                 if (id == R.id.nav_products) {
124                     Toast.makeText( context: HomeActivity.this,  text: "Products", Toast.LENGTH_SHORT).show();
125                     /*...*/
131                 } else if (id == R.id.nav_new) {
132                     Toast.makeText( context: HomeActivity.this,  text: "New product", Toast.LENGTH_SHORT).show();
133                 } else if (id == R.id.nav_profile) {
134                     Toast.makeText( context: HomeActivity.this,  text: "Profile", Toast.LENGTH_SHORT).show();
135                 } else if (id == R.id.nav_logout) {
136                     Toast.makeText( context: HomeActivity.this,  text: "Logout", Toast.LENGTH_SHORT).show();
137                 }
138                 // Close the drawer if the destination is not a top-level destination
139                 drawer.closeDrawers();
140             } else {
141                 if (id == R.id.nav_settings) {
142                     Toast.makeText( context: HomeActivity.this,  text: "Settings", Toast.LENGTH_SHORT).show();
143                 } else if (id == R.id.nav_language) {
144                     Toast.makeText( context: HomeActivity.this,  text: "Language", Toast.LENGTH_SHORT).show();
145                 }
146             }
147         });
```

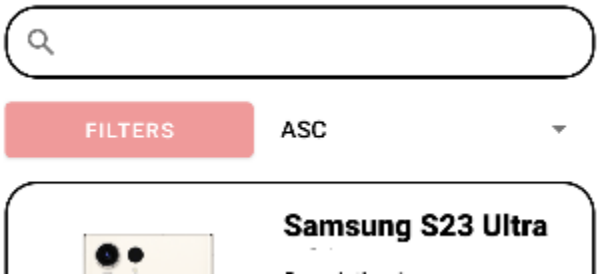
Slika 22. Kreiranje *toast*-a



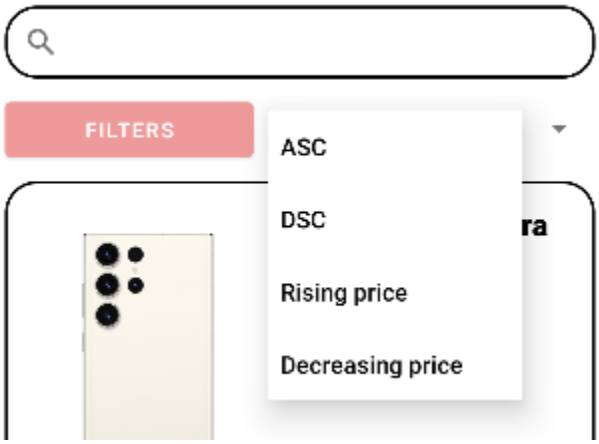
Slika 23. Primer *pop-up* poruke

6. Spinner

Spinner omogućava odabir jedne od više ponuđenih vrednosti. Na slikama 24 i 25 prikazan je izgled ovog elementa.



Slika 24. Izgled *spinner*-a sa izabranom opcijom



Slika 25. Izgled *spinner*-a sa ponuđenim opcijama

Za kreiranje elementa neophodno je definisati ga unutar XML datoteke kao na slici 26.

```
54        <Spinner
55            android:id="@+id/btnSort"
56            android:layout_width="wrap_content"
57            android:layout_height="wrap_content"
58            android:layout_marginStart="10dp"
59            android:layout_weight="0.3"
60            android:text="Sort"
61            android:tooltipText="Sort"
62            android:contentDescription="Sort"
63            />
```

Slika 26. Definisanje *spinner*-a

Ponuđene opcije mogu se kreirati unutar *values* foldera kao *sort-array* (slika 27).

```xml
<resources>
    <string-array name="sort_array">
        <item>ASC</item>
        <item>DSC</item>
        <item>Rising price</item>
        <item>Decreasing price</item>
    </string-array>
</resources>
```

Slika 27. Datoteka sa predefinisanim vrednostima *spinner*-a

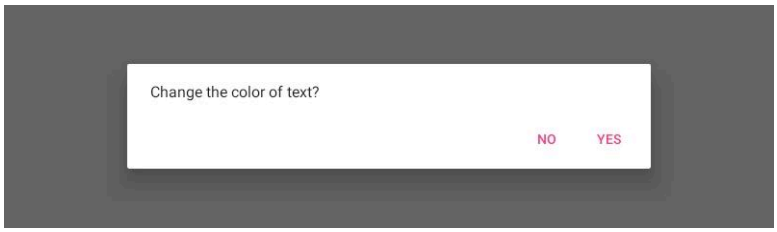
Za povezivanje prethodno prikazanih vrednosti, koriste se adapteri. Na slici 28. iskorišćen je *ArrayAdapter* kom se prosleđuje ugrađen *layout* - *simple_spinner_item* i element sa definisanim vrednostima - *spinner_option.* Otvoriti klasu *ProductsPageFragment.java*.

```
Spinner spinner = binding.btnSort;
// Create an ArrayAdapter using the string array and a default spinner layout
ArrayAdapter<String> arrayAdapter = new ArrayAdapter<>(getActivity(),
        android.R.layout.simple_spinner_item,
        getResources().getStringArray(R.array.sort_array));
// Specify the layout to use when the list of choices appears
arrayAdapter.setDropDownViewResource(android.R.layout.simple_spinner_dropdown_item);
// Apply the adapter to the spinner
spinner.setAdapter(arrayAdapter);
spinner.setOnItemSelectedListener(new AdapterView.OnItemSelectedListener() {...});
```

Slika 28. Povezivanje *spinner*-a sa predefinisanim vrednostima

7. Dijalozi

Dijalozi su komponente koje predstavljaju male prozore. Prikazuju se korisniku uglavnom kada se traži potvrda neke akcije ili unos nekih dodatnih podataka. Primer jednostavnog dijaloga dat je na slici 30.



Slika 30. Dijalog sa “da” i “ne” opcijom

Postoje izvedene klase poput *AlertDialog* i *DatePickerDialog* koje nasleđuju baznu klasu *Dialog.* Svaka od ovih klasa poseduje predefinisane *layout*-e i elemente koje je moguće prilagođavati. Na slici 31. prikazana je implementacija *AlertDialog*-a. Pomoću *setMessage* prosleđuje se tekst dijaloga, a *setPositiveButton* i *setNegativeButton* postavljaju dugmad za potvrđivanje i odustajanje. Da bi se dijalog prikazao portrebno je kreirati ga sa *create()*, a zatim pozvati i metodu *show().*

```
70      spinner.setOnItemSelectedListener(new AdapterView.OnItemSelectedListener() {
71          @Override
72          public void onItemSelected(AdapterView<?> parent, View view, int position, long id) {
73
74              AlertDialog.Builder dialog = new AlertDialog.Builder(getActivity());
75              dialog.setMessage("Change the sort option?")
76                      .setCancelable(false)
77                      .setPositiveButton( text: "Yes", new DialogInterface.OnClickListener() {
78                          public void onClick(DialogInterface dialog, int id) {
79                              Log.v( tag: "ShopApp", (String) parent.getItemAtPosition(position));
80                              ((TextView) parent.getChildAt( index: 0)).setTextColor(Color.MAGENTA);
81                          }
82                      })
83                      .setNegativeButton( text: "No", new DialogInterface.OnClickListener() {
84                          public void onClick(DialogInterface dialog, int id) { dialog.cancel(); }
87                      });
88              AlertDialog alert = dialog.create();
89              alert.show();
90          }
```

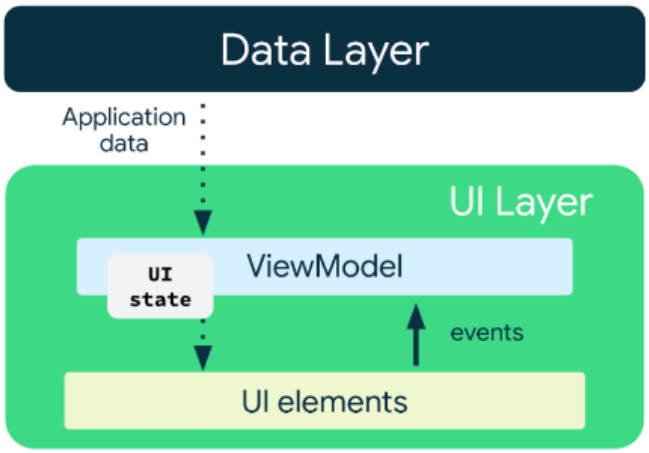
Slika 31. Implementacija *AlertDialog*-a

8. View Model

Upravljanje stanjem u Androidu odnosi se na proces upravljanja i čuvanja stanja aplikacije tokom različitih promena konfiguracije (kao što su rotacija ekrana) i između različitih sesija (kada korisnik napusti aplikaciju, a zatim se vrati). Cilj je pružiti besprekorno korisničko iskustvo čuvanjem stanja aplikacije uprkos promenama.

Klasa *ViewModel* (slika 4) je nosilac poslovne logike ili držač stanja na nivou ekrana. *ViewModel* klasa izlaže stanje korisničkom interfejsu i enkapsulira povezanu poslovnu logiku.

Glavna prednost je što kešira stanje i održava ga kroz promene konfiguracije. To znači da vaš korisnički interfejs ne mora ponovo da preuzima podatke prilikom navigacije između aktivnosti, ili prilikom promena konfiguracije, kao što je rotacija ekrana.



Slika 32. Prikaz UI Layer-a

Ključne prednosti klase ViewModel su suštinski dve:

1. Omogućava vam da održavate stanje korisničkog interfejsa.
2. Pruža pristup poslovnoj logici.

8.1 Perzistencija (Persistence)

ViewModel omogućava perzistenciju kako kroz stanje koje *ViewModel* drži, tako i kroz operacije koje *ViewModel* pokreće. Ovo keširanje znači da ne morate ponovo preuzimati podatke kroz uobičajene promene konfiguracije, kao što je rotacija ekrana.

8.2 SavedStateHandle

SavedStateHandle vam omogućava da sačuvate podatke ne samo kroz promene konfiguracije, već i kroz ponovno stvaranje procesa. To jest, omogućava vam da zadržite stanje korisničkog interfejsa čak i kada korisnik zatvori aplikaciju i otvori je u kasnijem trenutku.

Više na linku:
https://developer.android.com/topic/libraries/architecture/viewmodel/viewmodel-savedstate#java

8.3 Opseg (Scope)

Kada instancirate *ViewModel*, prosleđujete mu objekat koji implementira interfejs *ViewModelStoreOwner*. To može biti destinacija navigacije, grafikon navigacije, aktivnost, fragment ili bilo koji drugi tip koji implementira interfejs. Vaš ViewModel je onda ograničen na Lifecycle *ViewModelStoreOwner*-a. Ostaje u memoriji sve dok njegov *ViewModelStoreOwner* trajno ne nestane.

Kada se fragment ili aktivnost kojima je *ViewModel* ograničen uniš̌te, asinhroni rad se nastavlja u *ViewModelu* koji je njima ograničen. To je ključ za perzistenciju.

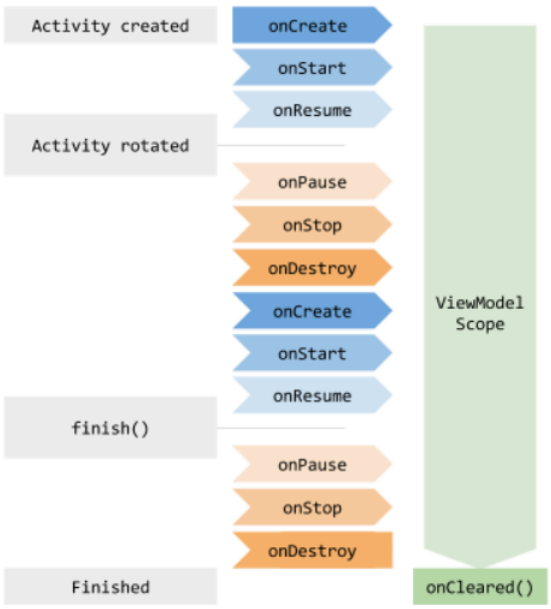
8.4 Životni ciklus ViewModel-a

Životni ciklus *ViewModela* je direktno vezan za njegov opseg. *VievModel* ostaje u memoriji sve dok *ViewModelStoreOwner* kome je obuhvaćen ne nestane. Ovo se može desiti u sledećim kontekstima:

- U slučaju aktivnosti, kada se završi.
- U slučaju fragmenta, kada se odvoji.
- U slučaju navigacionog unosa, kada je uklonjen iz zadnjeg steka.

Ovo čini *ViewModel* odličnim rešenjem za skladištenje podataka koji prežive promene konfiguracije.

Slika 33. ilustruje različita stanja životnog ciklusa aktivnosti dok prolazi kroz rotaciju i zatim završava. Ilustracija takođe prikazuje životni vek *ViewModel-a* pored povezanog životnog ciklusa aktivnosti. Ovaj poseban dijagram ilustruje stanja aktivnosti. Ista osnovna stanja važe za životni ciklus fragmenta.



Slika 33. Životni ciklus ViewModel-a

Obično zahtevate ViewModel prvi put kada sistem pozove *onCreate()* metod objekta aktivnosti. Sistem može pozvati *onCreate()* nekoliko puta tokom postojanja aktivnosti, na primer kada se ekran uređaja rotira. *ViewModel* postoji od trenutka kada prvi put zatražite *ViewModel* do završetka i uništenja aktivnosti.

8.5 Pristup poslovnoj logici

Iako je velika većina poslovne logike prisutna u sloju podataka, sloj korisničkog interfejsa takođe može sadržati poslovnu logiku. To može biti slučaj kada se kombinuju podaci iz više repozitorijuma za kreiranje stanja korisničkog interfejsa ekrana, ili kada određena vrsta podataka ne zahteva sloj podataka.

ViewModel je pravo mesto za rukovanje poslovnom logikom u sloju korisničkog interfejsa. ViewModel je takođe zadužen za rukovanje događajima i delegiranje istih drugim slojevima hijerarhije kada je potrebno primeniti poslovnu logiku za modifikaciju podataka aplikacije.

8.6 Implementacija

Sledi primer implementacije *ViewModela* za komponentu koja omogućava korisniku pretragu.

U ovom primeru, odgovornost za sticanje i držanje vrednosti teksta koji se pretražuje u searchView komponenti leži na *ViewModel-u*, a ne direktno na Aktivnosti ili Fragmentu.

ViewModel se obično koristi u kombinaciji sa LiveData ili nekom drugom observables klasom kako bi se osiguralo da UI reaguje na promene podataka. LiveData omogućava ViewModel-u da automatski ažurira UI komponente kada se podaci promene, čime se osigurava da je korisnički interfejs uvek sinhronizovan sa trenutnim stanjem podataka.

Na slici 34 prikazan je *ProductsPageViewModel*. Promenljiva searchText je instanca *MutableLiveData*, generičke klase koja drži neki tip podatka, u ovom slučaju String. *MutableLiveData* je lifecycle-aware observable klasa, što znači da UI komponente (kao što su Activity ili Fragment) mogu posmatrati (observe) promene u ovim podacima i biti automatski obaveštene kada do promene dođe. Posmatranje LiveData objekata omogućava UI komponentama da reaguju na promene podataka, ažurirajući se prema potrebi.

```
7   public class ProductsPageViewModel extends ViewModel {
8       /*...*/
.7      private final MutableLiveData<String> searchText;
.8      public ProductsPageViewModel(){
.9          searchText = new MutableLiveData<>();
:0          searchText.setValue("This is search help!");
:1      }
:2      public LiveData<String> getText() { return searchText; }
:5  }
:6
```

Slika 34. ProductsPageViewModel

ViewModel obično ne bi trebalo da upućuje na pogled, životni ciklus ili bilo koju klasu koja može da sadrži referencu na kontekst aktivnosti. Pošto je životni ciklus ViewModel-a veći od korisničkog interfejsa, držanje API-ja koji se odnosi na životni ciklus u ViewModel-u može izazvati curenje memorije.

Otvoriti ProductsPageFragment.java klasu, gde je prikazano kreiranje instance ViewModel-a (slika 35).

```
42  public View onCreateView(@NonNull LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState) {
43      // Instanciranje ViewModel-a
44      // ViewModelProvider osigurava da se ViewModel ne stvara svaki
45      // put prilikom promjene konfiguracije (npr. rotacije ekrana),
46      // već se ponovno koristi postojeća instanca, čime se očuvaju podaci.
47      productsViewModel = new ViewModelProvider( owner: this).get(ProductsPageViewModel.class);
48
49      binding = FragmentProductsPageBinding.inflate(inflater, container,  attachToParent: false);
50      View root = binding.getRoot();
51
```

Slika 35. Kreiranje instance ViewModel-a

Zatim možete pristupiti *ViewModel-u* iz fragmenta na sledeći način:

```
53
54      SearchView searchView = binding.searchText;
55      /*...*/
66      productsViewModel.getText().observe(getViewLifecycleOwner(), searchView::setQueryHint);
67
```

Slika 36. Pristupanje viewModelu iz aktivnosti/fragmenta

Posmatranje LiveData objekta i ažuriranje UI-a: Ovaj deo koda dodaje observer na LiveData<String> objekat unutar *ProductsPageViewModel*. Svaki put kada dođe do promene podatka unutar LiveData objekta (u ovom slučaju searchText), ta promena se automatski prosleđuje i aktivira se metoda *setQueryHint* na SearchView komponenti sa novom vrednošću. Funkcija *getViewLifecycleOwner()* garantuje da se observer povezuje sa životnim ciklusom vlasnika prikaza fragmenta, što znači da će se observer automatski ukloniti kada fragment više nije vidljiv ili je uništen, sprečavajući time moguće curenje memorije.

9. Domaći

Domaći se nalazi na *Canvas-u (canvas.ftn.uns.ac.rs)* na putanji *Вежбе/04 Задатак.pdf*.

Primer možete preuzeti na sledećem linku:
https://gitlab.com/mobilne-aplikacije/mobilne-aplikacije-siit-2024-25

Za dodatna pitanja možete se obratiti asistentima:

- Svetlana Antešević (svetlanaantesevic@uns.ac.rs)
- Jelena Matković (matkovic.jelena@uns.ac.rs)