KAUNAS UNIVERSITY OF TECHNOLOGY

FACULTY OF INFORMATICS

T120B169 App Development for Smart Mobile Systems

Shopping application

Group, Name and Surname:
IFZm-8, Rytis Lukaitis
IFZm-8, Karolis Baranauskas

Date: *2022.01.17*

Kaunas, 2022

Tables of Contents

Description of Your app

1. What type is your application/game? *It is an app that should help people with shopping.*
2. Description. *Some people sometimes forget what they want to buy, others take notes and carry a lot of them, which is uncomfortable. This app should help those people. You will be able to have separate checklists, create reminders, calculate how much everything costs. It would also be nice to save and categorize previously purchased items with their costs, which would save you less typing (you could just select them), have presets which you can instantly set up, and select app's language. In addition, having statistics of your purchases and setting up an allowance would help with your budget planning. Lastly, sorting, filtering, and organizing things will make your visits to the shop much easier.*

Team composition

Name, Surname	Role
Rytis Lukaitis	Lead Programmer, responsible for main function implementation
Karolis Baranauskas	Lead Designer, responsible for UI and additional function implementation

Project repository link

https://github.com/BroofTerr/ShopList

Functionality of your app

List of functions (adapt to your own app)

1. Have a custom class for products, in which you could specify the cost and add an image (either from gallery or on the go via camera);
2. Categorize products;
3. Make to-buy checklists where you either select products from the ones you previously entered yourself or creating on the go (which would automatically save them for later use)
    a. You could specify the amount, change the price if needed;
    b. It would automatically calculate the sum and display it to the user;
    c. You could check or cross-out an item that you picked up (could also have progress indication);
    d. Set importance of an item (is it a must to buy).
4. Make reminders (when you reach the specified location or at specified time, or in some amount of time;
5. Have stats to see what you buy the most (product, categories) or how much you spend either weekly or monthly;
6. Set up your budget/allowance and get warnings if your checklist exceeds the specified limit;
7. Save information about your products in a database;
8. Sort items in a list by importance/cost/name;
9. Filter items by letter or importance;
10. Make preset checklists to save time the next time;
11. Select a language.
12. Product code scanning

Solution

Task #1. Add navigation between to-buy lists, budget and statistics fragments

Firstly, in the main activity layout (*activity_main.xml*) *FrameLayout* and *BottomNavigationView* components were added (Figure 1). Using these components, I was able to swap between fragments.

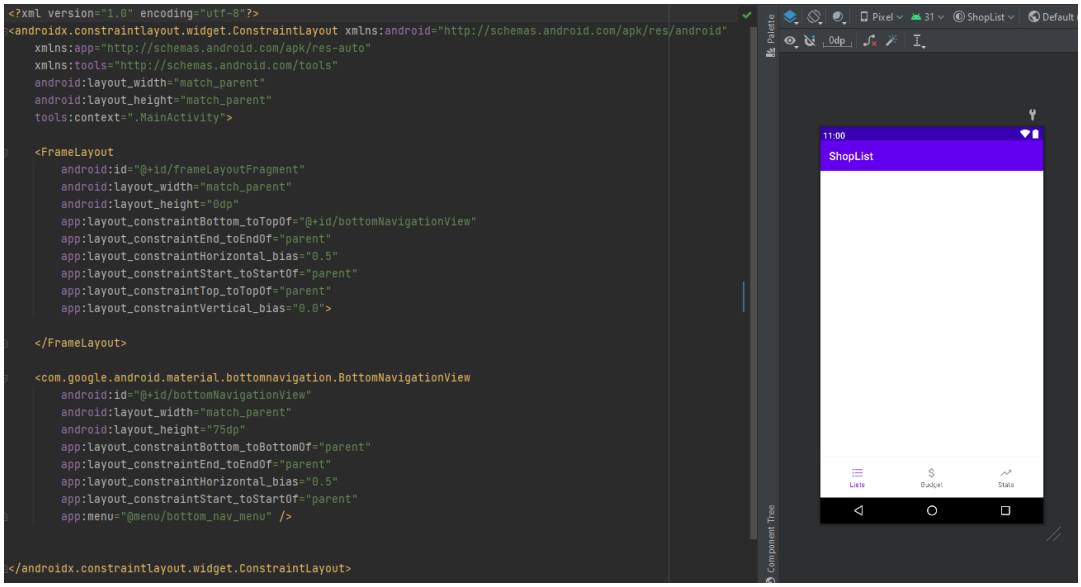


Figure 1. Main activity layout

For the menu, I created *bottom_nav_menu.xml* layout, with which I can display images along side the fragment titles (Figure 2).

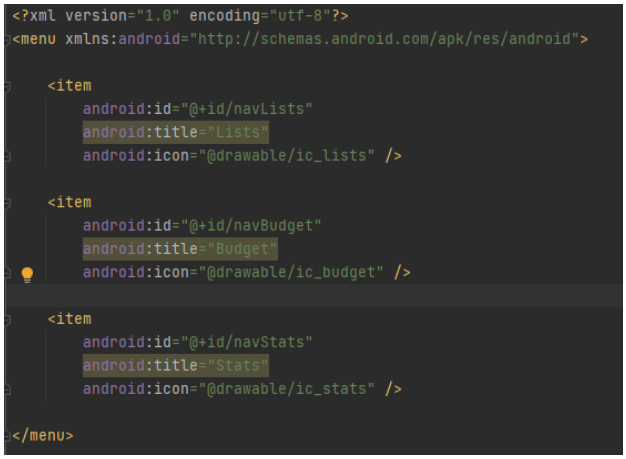


Figure 2. Bottom navigation menu layout

The fragment swapping happens in *MainActicity.java* class where I use implemented bottom navigation menu's *OnNavigationItemSelectedListener()* method (Figure 3).

```java
    // Swaps between activity fragments
    private void setCurrentFragment(Fragment frag)
    {
        getSupportFragmentManager().beginTransaction()
                .replace(R.id.frameLayoutFragment, frag)
                .commit();
    }

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        _listsFragment = new ListsFragment();
        _statsFragment = new StatsFragment();
        _budgetFragment = new BudgetFragment();
        setCurrentFragment(_listsFragment);

        _botNavMenu = findViewById(R.id.bottomNavigationView);
        _botNavMenu.setOnNavigationItemSelectedListener(new BottomNavigationView.OnNavigationItemSelectedListener() {
            @Override
            public boolean onNavigationItemSelected(@NonNull MenuItem item) {
                switch (item.getItemId()) {
                    case R.id.navLists:
                        setCurrentFragment(_listsFragment);
                        break;

                    case R.id.navBudget:
                        setCurrentFragment(_budgetFragment);
                        break;

                    case R.id.navStats:
                        setCurrentFragment(_statsFragment);
                        break;
                }
                return true;
            }
        });
    }

}
```

Figure 3. *MainActivity.java* class

The fragments can be seen in Figure 4.

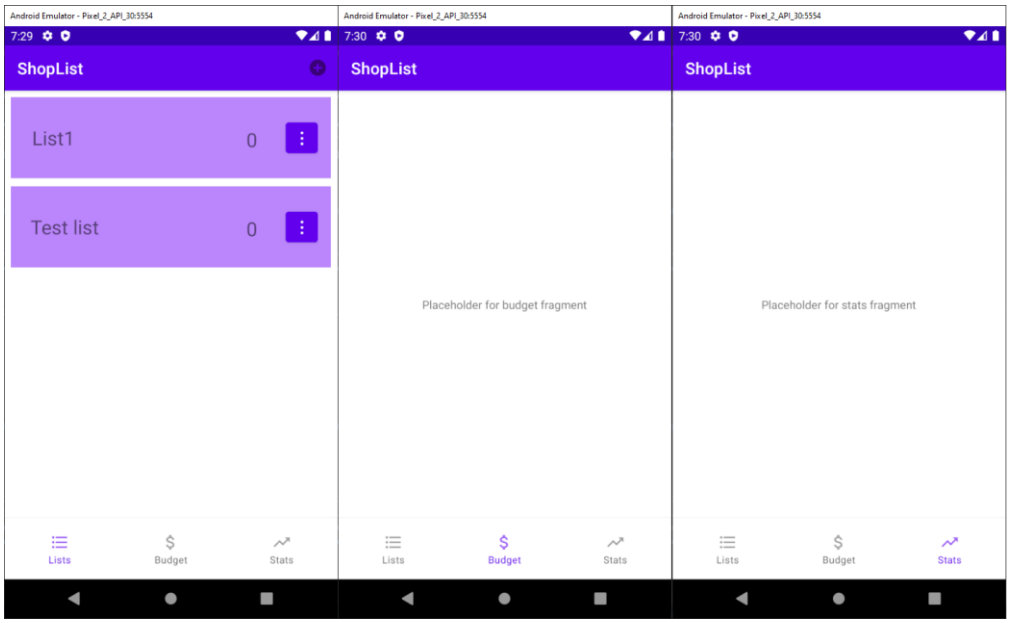


Figure 4. Preview of fragments

Task #2. Create to-buy list basic functionality

For this task I made so you could create new, rename or delete lists. A list example can be seen on the left-most fragment in Figure 4. To achieve this, I implemented *RecyclerView* widget in my *fragment_lists.xml* layout file (Figure 5).

```
<?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=".ListsFragment">

    <androidx.recyclerview.widget.RecyclerView
        android:id="@+id/recyclerViewShoppingLists"
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:layout_marginStart="8dp"
        android:layout_marginTop="8dp"
        android:layout_marginEnd="8dp"
        android:layout_marginBottom="8dp"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintVertical_bias="0.0" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

Figure 5. *fragment_lists.xml* fragment layout

I also needed to create a custom class for a single to-buy list entry, which would contain other information inside of it, to be able to display what I needed inside the *RecyclerView* widget. A part of the custom *ShoppingList.java* class can be seen in Figure 6 which contains its title, item count and another list for products, which was implemented during another task.

```
public class ShoppingList implements Parcelable {

    String title;
    int itemCount;
    List<ProductEntry> productList;

    public ShoppingList(String title)
    {
        this.title = title;
        this.itemCount = 0;
        productList = new ArrayList<ProductEntry>();
    }

    public ShoppingList(String title, List<ProductEntry> list)
    {
        this.title = title;
        productList = list;
        itemCount = list.size();
    }
```

Figure 6. Custom *ShoppingList.java* class

Main functionality of the *RecyclerView* widget is handled inside *ShoppingListAdapter.java* class. In short, it creates as many to-buy tabs as you have and this is done inside *onBindViewHolder* method, and the layout for each individual list is set inside *onCreateViewHolder* method (Figure 7). The list layout file, *item_shopping_list* can be seen in Figure 8.

```java
@NonNull
@Override
public ShoppingListViewHolder onCreateViewHolder(@NonNull ViewGroup parent, int viewType) {
    View view = LayoutInflater.from(parent.getContext()).inflate(R.layout.item_shopping_list, parent,  attachToRoot: false);
    return new ShoppingListViewHolder(view, mOnListListener);
}

@Override
public void onBindViewHolder(@NonNull ShoppingListViewHolder holder, int position) {
    ShoppingList list = shoppingList.get(position);

    holder.textViewList.setText(list.title);
    holder.textViewCount.setText(String.valueOf(list.itemCount));

    holder.bttnOptions.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View view) {

            //creating a popup menu
            PopupMenu popup = new PopupMenu(view.getContext(), holder.bttnOptions);
            popup.inflate(R.menu.list_options_menu);
            popup.setOnMenuItemClickListener(new PopupMenu.OnMenuItemClickListener() {
                @Override
                public boolean onMenuItemClick(MenuItem item) {
                    switch (item.getItemId()) {
                        case R.id.optionRename:
                            mOnListListener.onRenameClick(holder.getAdapterPosition());
                            return true;
                        case R.id.optionRemove:
                            removeListEntry(holder.getAdapterPosition());
                            return true;
                        default:
                            return false;
                    }
                }
            });
            popup.show();

        }
    });
}
```

Figure 7. *ShoppingListAdapter.java* class

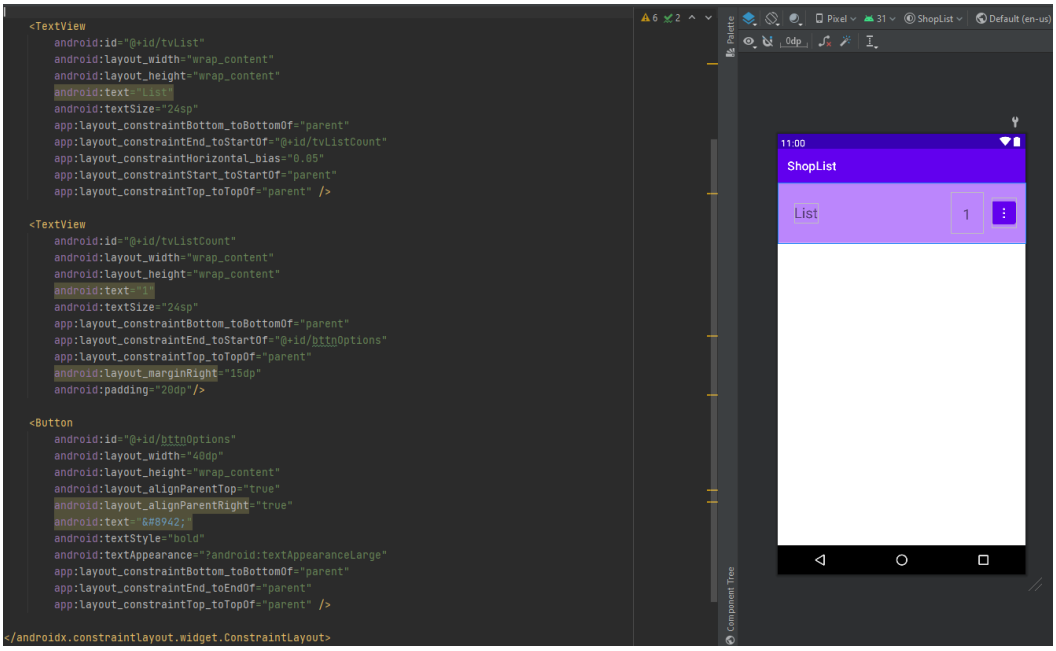


Figure 8. *item_shopping_list.xml* layout file

Lastly, example data and setup of the *RecyclerView* is done inside *ListsFragment.java* file. It creates the adapter and sends the list data inside it (Figure 9).

```java
public class ListsFragment extends Fragment implements NewListDialogue.NewListDialogueListener,
        ShoppingListAdapter.OnListListener,
        RenameListDialogue.RenameListDialogueListener {

    RecyclerView recyclerView;
    ShoppingListAdapter adapter;
    List<ShoppingList> shoppingList;

    LinearLayoutManager linearLayoutManager;
    int renameIndex;

    public ListsFragment() {
        // Required empty public constructor
        // Later load data
        shoppingList = new ArrayList<ShoppingList>();
        shoppingList.add(new ShoppingList( title: "List1"));
        shoppingList.add(new ShoppingList( title: "Test list"));

        adapter = new ShoppingListAdapter(shoppingList,  onListListener: this);
    }

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setHasOptionsMenu(true);

    }

    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
                             Bundle savedInstanceState) {

        View fragView = inflater.inflate(R.layout.fragment_lists, container,  attachToRoot: false);
        recyclerView = fragView.findViewById(R.id.recyclerViewShoppingLists);
        recyclerView.setHasFixedSize(true);
        recyclerView.setLayoutManager(new LinearLayoutManager(fragView.getContext()));
        adapter.setFragmentManager(getFragmentManager());
        recyclerView.setAdapter(adapter);

        return fragView;
    }
```

Figure 9. ***ListsFragment.java*** **class**

Task #3. Create basic display on what is inside the list

For this task I reused what I did in task 2 (the usage of *RecyclerView* and *Adapter*). But it was a little bit different, because instead of a fragment, list contents were displayed inside an activity. To begin with, I needed to create classes for products and list entries. *Product* class contains the name of the product and its price (Figure 10), while *ProductEntry* class contains the product object (which is inside the list), its quantity and the state of that product (if it has been picked up) (Figure 11).

```
public class Product {
    String name;
    //category;
    float price;
    //Image;

    public Product()
    {
        name = "";
        price = 0f;
    }

    public Product(String name, float price)
    {
        this.name = name;
        this.price = price;
    }
}
```

Figure 10. *Product.java* class

```
public class ProductEntry {
    Product product;
    int quantity;
    boolean isChecked;

    public ProductEntry()
    {
        product = new Product();
        quantity = 0;
        isChecked = false;
    }

    public ProductEntry(Product p, int q, boolean isChecked)
    {
        product = p;
        quantity = q;
        this.isChecked = isChecked;
    }

    public float getCost() { return product.price * quantity; }
}
```

Figure 11. *ProductEntry.java* class

The layout for the product inside the list was made in *item_product.xml* file, which can be seen in Figure 12.

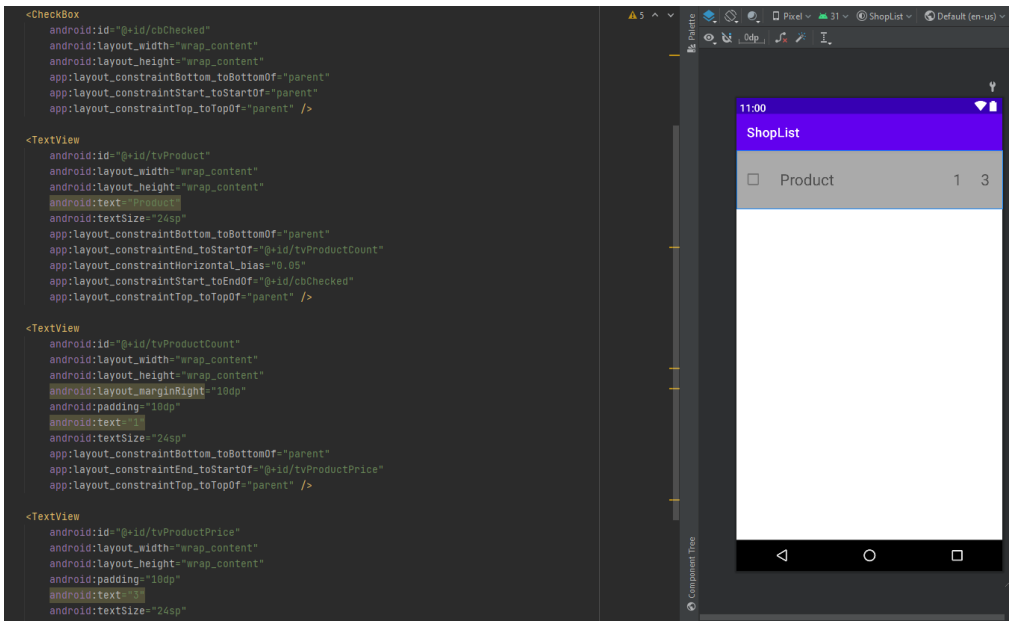


Figure 12. *item_product.xml* layout

Inside *ListActivity.java* class i set up the *RecyclerView* and *Adapter* components and prepare some example data. I also receive some data from the *ListsFragment* class when changing into another activity via intents. To display that, I change the app bar title to match the title of a list the user enters (Figure 13).

```
public ListActivity()
{
    Product p1 = new Product( name: "Apple", price: 3f);
    Product p2 = new Product( name: "Juice", price: 0.75f);
    ProductEntry e1 = new ProductEntry(p1, q: 2, isChecked: false);
    ProductEntry e2 = new ProductEntry(p2, q: 1, isChecked: true);
    productList = new ArrayList<>();
    productList.add(e1);
    productList.add(e2);

    adapter = new ProductListAdapter(productList, onProductListener: this);
}

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_list);
    Toolbar toolbar = findViewById(R.id.toolbarList);

    setSupportActionBar(toolbar);
    getSupportActionBar().setDisplayHomeAsUpEnabled(true);

    data = getIntent().getParcelableExtra( name: "list_object");

    if(data.productList != null && data.productList.size() != 0)
    {
        productList = data.productList;
    }

    setTitle(data.title);

    recyclerView = findViewById(R.id.recyclerViewProducts);
    recyclerView.setLayoutManager(new LinearLayoutManager( context: this));
    recyclerView.setAdapter(adapter);

    tvTotal = findViewById(R.id.tvTotalCost);
    tvChecked = findViewById(R.id.tvCheckedCost);

    CalculateCosts();
}
```

Figure 13. *ListActivity.java* class

Finally, in Figure 14 you can see example view of how it looks like when you enter a list. You can check and/or uncheck products and below the list it calculates the total cost and the current cost of your list.

Figure 14. Example view of what user can see inside a list entry

Defense Task #1. Implement product filtering inside a list

For this task I added *EditText* and *Button* components inside my *activity_list.xml* layout. In the *ListActivity.java* class, I added a new *ProductEntry* list, which will hold only the filtered products.

When a product is typed inside the *EditText* component and the button is clicked, via the button's *setOnClickListener* method, I get the inputed text and call *FilterList* method (Figure 15)

```
public void FilterList()
{
    filteredList = new ArrayList<>();
    for (ProductEntry e : productList)
    {
        if (e.product.name.equals(productFilterName))
        {
            filteredList.add(e);
        }
    }

    if (productFilterName.equals(""))
    {
        adapter.UpdateList(productList);
        isFiltered = false;
    }
    else
    {
        adapter.UpdateList(filteredList);
        isFiltered = true;
    }

}
```

Figure 15. *FilterList()* method

Inside the method, I iterate through all the products that are in the list and only add those products, whose name match with what was written in the text field. Later, if nothing was written, application will show the original list. The *RecyclerView* update happens in the custom *UpdateList* method where I send to the adapter what it should display (Figure 16).

```
public void UpdateList(List<ProductEntry> newList)
{
    productList = newList;
    notifyDataSetChanged();

}
```

Figure 16. *UpdateList() method*

Task #4. Update ProductEntry object with category field and include it with filter

To create a more flexible environment for users we have added a category field to all *ProductEntry* objects. This will help with filtering the products. The category is displayed alongside the items so when filtering it is clearer for the user (Figure 17).

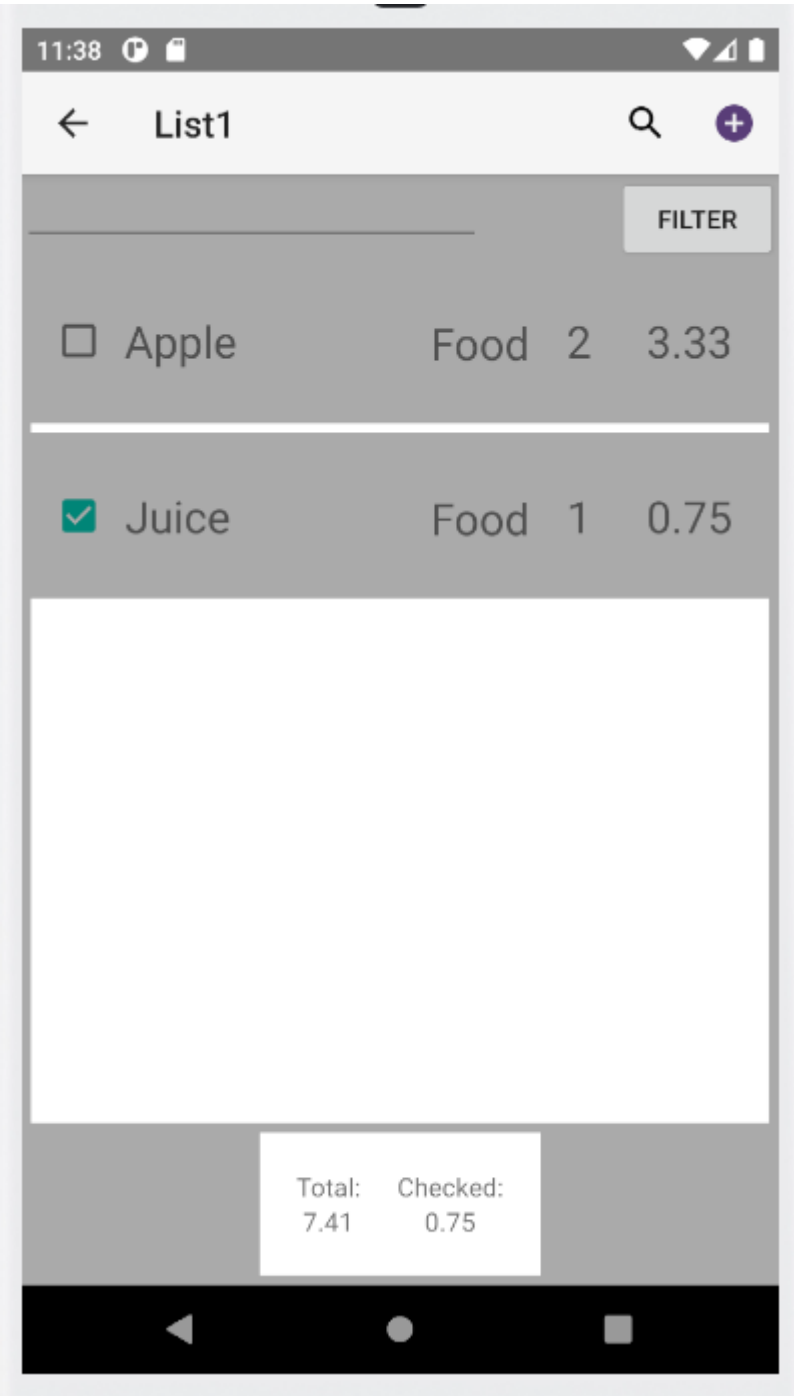


Figure 17. Product display in the list

The *FilterList()* method was updated to make sure the category is also included into the filter.

```
public void FilterList()
{
    filteredList = new ArrayList<>();
    for (ProductEntry e : productList)
    {
        if (e.product.name.equals(productFilterName) || e.category.equals(productFilterName))
        {
            filteredList.add(e);
        }
    }

    if (productFilterName.equals(""))
    {
        adapter.UpdateList(productList);
        isFiltered = false;
    }
    else
    {
        adapter.UpdateList(filteredList);
        isFiltered = true;
    }

}
```

Figure 18. *FilterList()* method after adding the category variable

Task #5. Implement an Add button for adding new products

In order for the user to use the shopping list application it is important that they would be able to add their own chosen products to the list. This is implemented via a *Dialogue box*, which is activated by clicking the + icon in the top right corner of the screen. In this dialogue box the user can input the name of the product, the quantity of the products (must be a number), the category of the products and the price of the products (must be a number) (Figure 19).

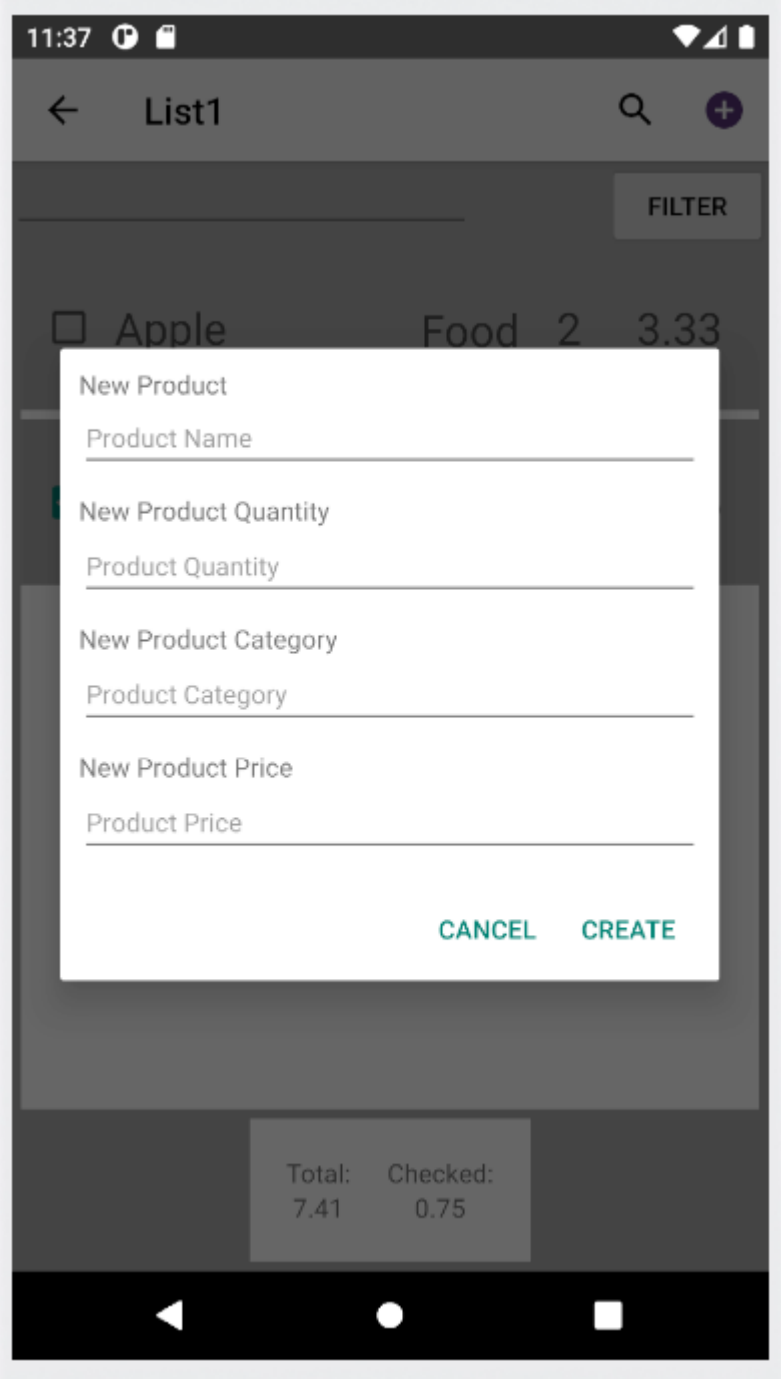


Figure 19. Dialogue box for adding a new product to the list

After the product information is added the user can click the *create* button, which will add the new product to the list of products. This is done via the *applyProduct()* method (Figure 20).

```java
@Override
public void applyProduct(String productName, String productCategory, float productPrice, int productQuantity)
{
    Product prod = new Product(productName, productPrice);
    ProductEntry prodEnt = new ProductEntry(prod, productCategory, productQuantity, isChecked: false);
    productList.add(prodEnt);
    adapter.notifyItemInserted( position: productList.size() - 1);
    Toast.makeText( context: this, text: "You created a new product: " + productName, Toast.LENGTH_SHORT).show();
    CalculateCosts(productList);
}
```

Figure 20. *applyProduct()* method added to the *ListActivity*

The method *applyProduct()* is being overriden from a new class *NewProductDialogue*. In this class we check the written values in the *EditTextFields* and apply them to the new *ProductEntry* object while also checking if the fields are empty or incorrect (Figure 21).

```java
@NonNull
@Override
public Dialog onCreateDialog(@Nullable Bundle savedInstanceState) {
    AlertDialog.Builder builder = new AlertDialog.Builder(getActivity());

    LayoutInflater inflater = getActivity().getLayoutInflater();
    View view = inflater.inflate(R.layout.layout_new_product, root: null);

    builder.setView(view)
            .setNegativeButton( text: "Cancel", new DialogInterface.OnClickListener() {
                @Override
                public void onClick(DialogInterface dialog, int which) { }
            })
            .setPositiveButton( text: "Create", new DialogInterface.OnClickListener() {
                @Override
                public void onClick(DialogInterface dialog, int which) {
                    productName = editTextProductName.getText().toString();
                    productCategory = editTextProductCategory.getText().toString();
                    productPrice = Float.parseFloat(editTextProductPrice.getText().toString());
                    productQuantity = Integer.parseInt(editTextProductQuantity.getText().toString());

                    if (productName.equals("") || productCategory.equals("") || productPrice <= 0f || productQuantity <= 0)
                    { Toast.makeText(getContext(), text: "Field is empty or incorrect", Toast.LENGTH_SHORT).show(); }
                    else
                    { listener.applyProduct(productName, productCategory, productPrice, productQuantity); }
                }
            });

    editTextProductName = view.findViewById(R.id.etNewProductName);
    editTextProductCategory = view.findViewById(R.id.etNewProductCategory);
    editTextProductPrice = view.findViewById(R.id.etNewProductPrice);
    editTextProductQuantity = view.findViewById(R.id.etNewProductQuantity);

    return builder.create();
}
```

Figure 21. Overriden Dialog method, which applies the variables from the *EditTextFields*

After having completed this the final product is added to the list and the method *CalculateCosts()* is called so that the Total cost values would be adjusted to the newly complete list (Figure 22).

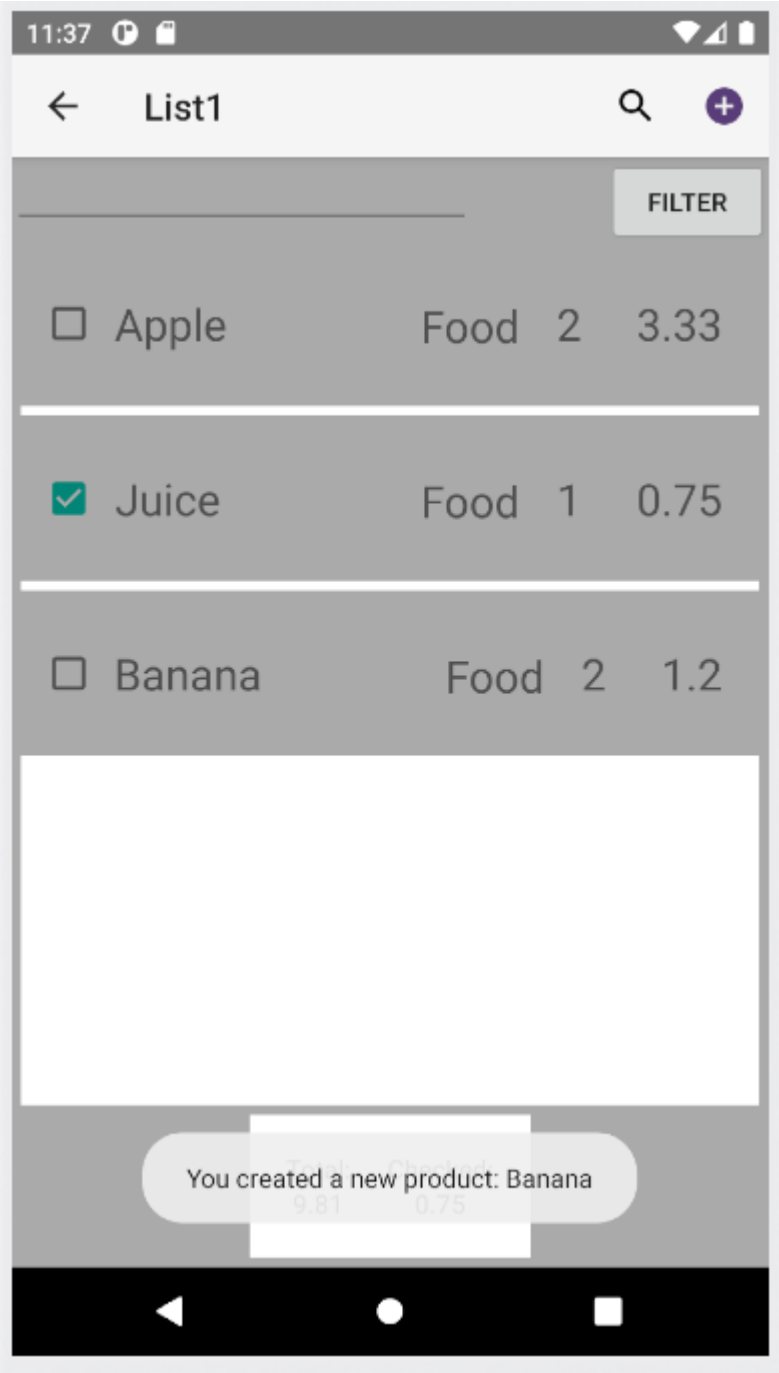


Figure 22. The list after having added a new product *Banana*

Task #6. Include all strings in the *values.xml* file and update UI

Hardcoding text into any field is a very bad practice. All text should be taken from the *values.xml* file where all strings are placed. All strings are called from this list by using *@string/chosenText* (Figure 23).

```
<resources>
    <string name="app_name">ShopList</string>
    <string name="total">Total:</string>
    <string name="checked">Checked:</string>
    <string name="filter">Filter</string>
    <string name="budgetFragment">Budget Fragment</string>
    <string name="statsFragment">Stats Fragment</string>
    <string name="product">Product</string>
    <string name="category">Category</string>
    <string name="list">List</string>
    <string name="editList">Edit List</string>
    <string name="listName">List Name</string>
    <string name="newList">New List</string>
    <string name="newProduct">New Product</string>
    <string name="productName">Product Name</string>
    <string name="newProductQuantity">New Product Quantity</string>
    <string name="productQuantity">Product Quantity</string>
    <string name="newProductCategory">New Product Category</string>
    <string name="productCategory">Product Category</string>
    <string name="newProductPrice">New Product Price</string>
    <string name="productPrice">Product Price</string>
    <string name="action_settings">Settings</string>
</resources>
```

Figure 23. Fixed *values.xml* file with updated strings

To ensure a better user experience we have also updated the UI to be more colorful and not as divided as it was before (Figure 24).

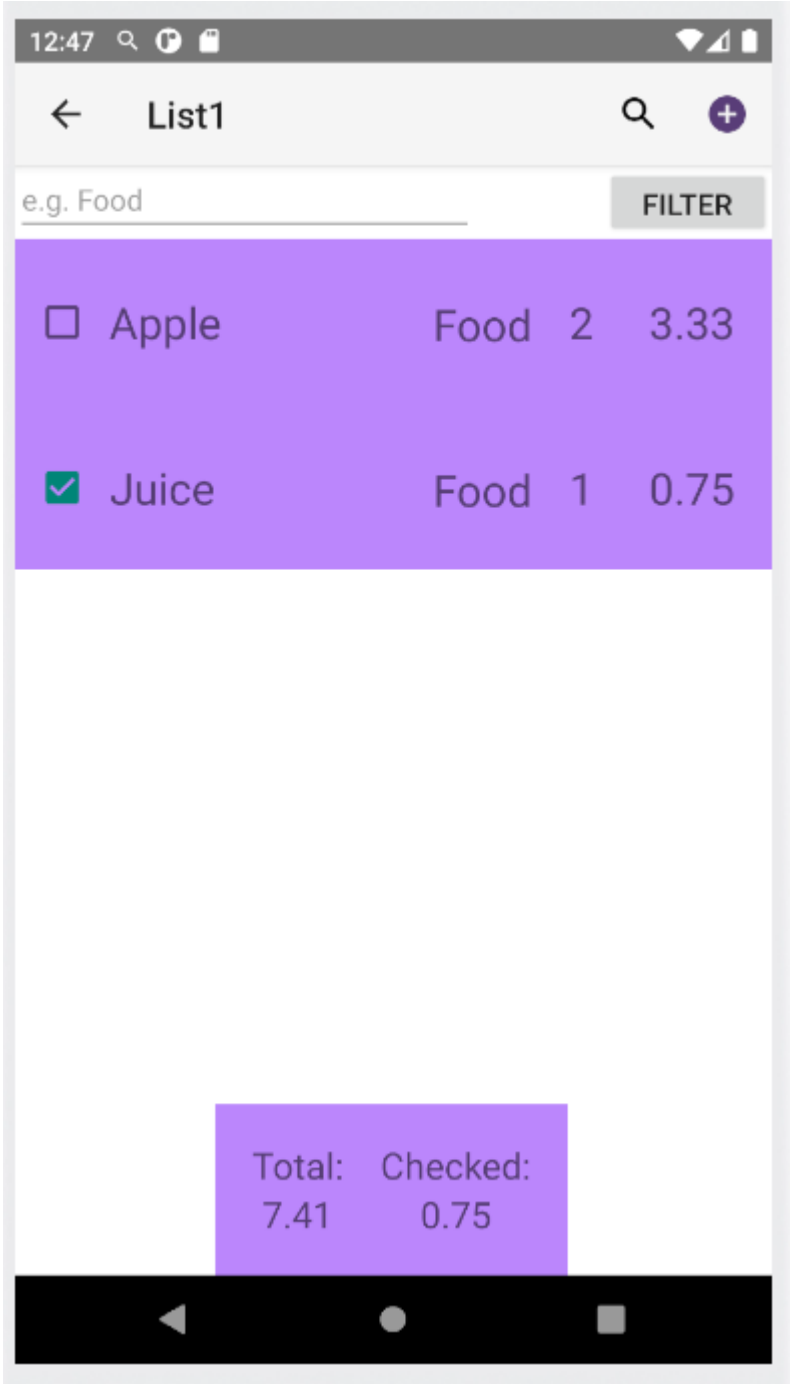


Figure 24. Updated UI with bigger text elements and more colors

Task #7. Add a reminder *Notification* element

A great shopping app will remind you to make sure you have got everything you need, to do this we have included a reminder notification. At this point the notification is simple as it appears when the app is launched, but in later iterations it will be timed, and location based (Figure 25).

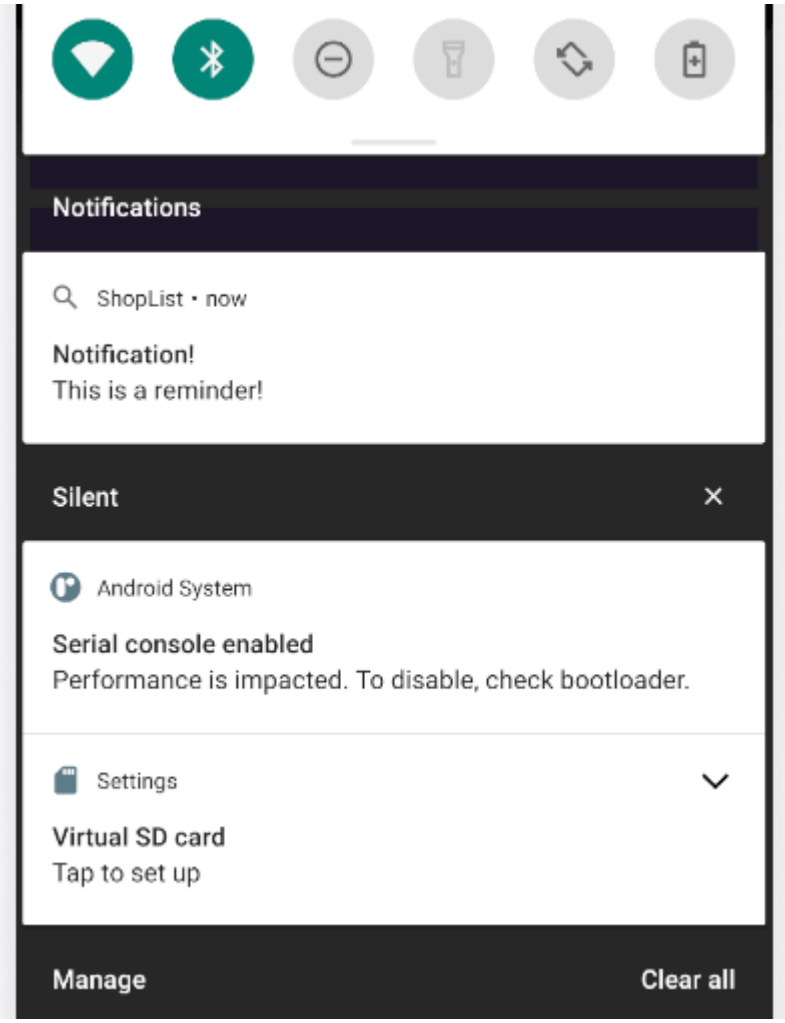


Figure 25. Notification reminder that appears on startup of the application

This method was realised by using the *NotificationCompat* class. With this class we were able to create a builder that has all of our needed attributes. We set the notification channel and call it using the *NotificationManager* to display the notification on the Android system.

```
if(Build.VERSION.SDK_INT >= Build.VERSION_CODES.O)
{
    NotificationChannel channel = new NotificationChannel( id: "My Notification",
            name: "My Notification", NotificationManager.IMPORTANCE_DEFAULT);
    NotificationManager manager = getSystemService(NotificationManager.class);
    manager.createNotificationChannel(channel);
}

NotificationCompat.Builder builder = new NotificationCompat.Builder( context: MainActivity.this,
        channelId: "My Notification")
        .setContentTitle("Notification!")
        .setContentText("This is a reminder!")
        .setSmallIcon(R.drawable.ic_search_black_24dp)
        .setAutoCancel(true)
        .setPriority(NotificationCompat.PRIORITY_DEFAULT);

NotificationManagerCompat notificationManager = NotificationManagerCompat.from(this);
notificationManager.notify( id: 1, builder.build());
```

Figure 26. Code fragment of the Notification function

Defense Task #2. Add a button that changes the name and category of the product to images instead.

The defense task was to add a button that would change the name of the product (*TextView*) and the name of the category (*TextView*) to *ImageView* components. I did this by adding the *ImageView* components behind the *TextView* components and making them invisible. After pressing the button, the program made the *TextView* components invisible and the *ImageView* components visible.

```
bttnChange.setOnClickListener(new View.OnClickListener()
{
    @Override
    public void onClick(View view) {
        for (ProductEntry e : productList)
        {
            productName = findViewById(R.id.tvProduct);
            productNameImage = findViewById(R.id.ProductImageView);
            productCategory = findViewById(R.id.tvCategory);
            productCategoryImage = findViewById(R.id.CategoryImageView);

            if (e.category.equals("Food") && e.product.name.equals("Apple"))
            {
                productName.setVisibility(view.INVISIBLE);
                productNameImage.setImageResource(R.drawable.apple);
                productNameImage.setVisibility(View.VISIBLE);
                productCategory.setVisibility(view.INVISIBLE);
                productCategoryImage.setImageResource(R.drawable.coupe_entree_plate_drk_brn_wht_o_ut);
                productCategoryImage.setVisibility(view.VISIBLE);
            }
        }
    }
});
```

Figure 27 *onClick()* method that changes the visibility of both *ImageView* and *TextView* components

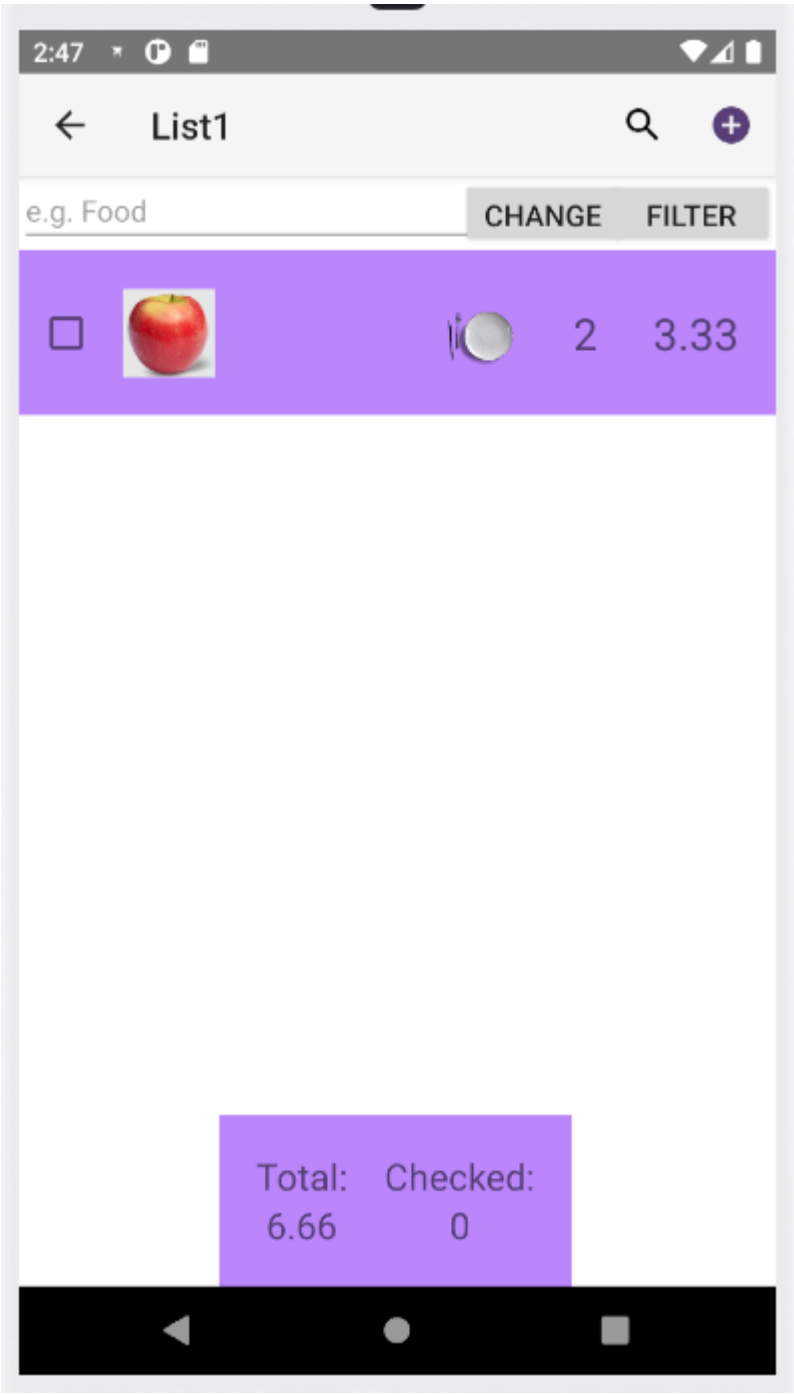


Figure 28. How the list looks after the button was pressed

Task #8. Added *Parcelable* implementation to custom data classes

For this task we needed to implement *Parcelable* inside *Product* (Figure 29) and *ProductEntry* (Figure 30) classes. This makes it possible to later send and receive custom data between activities.

```
19      public Product(String name, String category, float price)
20      {
21          this.name = name;
22          this.category = category;
23          this.price = price;
24      }
25
26      protected Product(Parcel in) {
27          name = in.readString();
28          category = in.readString();
29          price = in.readFloat();
30      }
31
32      @Override
33      public void writeToParcel(Parcel dest, int flags) {
34          dest.writeString(name);
35          dest.writeString(category);
36          dest.writeFloat(price);
37      }
38
39      @Override
40      public int describeContents() { return 0; }
43
44      public static final Creator<Product> CREATOR = new Creator<Product>() {
45          @Override
46          public Product createFromParcel(Parcel in) { return new Product(in); }
49
50          @Override
51          public Product[] newArray(int size) { return new Product[size]; }
54      };
```

Figure 29. *Parcelable* implementation in *Product* class

```
25      protected ProductEntry(Parcel in) {
26          quantity = in.readInt();
27          isChecked = in.readByte() != 0;
28          product = (Product) in.readValue(Product.class.getClassLoader());
29      }
30
31      @Override
32      public void writeToParcel(Parcel dest, int flags) {
33          dest.writeInt(quantity);
34          dest.writeByte((byte) (isChecked ? 1 : 0));
35          dest.writeValue(product);
36      }
37
38      @Override
39      public int describeContents() { return 0; }
42
43      public static final Creator<ProductEntry> CREATOR = new Creator<ProductEntry>() {
44          @Override
45          public ProductEntry createFromParcel(Parcel in) { return new ProductEntry(in); }
48
49          @Override
50          public ProductEntry[] newArray(int size) { return new ProductEntry[size]; }
53      };
```

Figure 30. *Parcelable* implementation in *ProductEntry* class

Furthermore, we also needed to fix *Parcelable* implementation in *ShoppingList* class, so it correctly sends custom classes data (Figure 31).

```
protected ShoppingList(Parcel in) {
    title = in.readString();
    itemCount = in.readInt();
    if (in.readByte() == 0x01) {
        productList = new ArrayList<ProductEntry>();
        in.readList(productList, ProductEntry.class.getClassLoader());
    } else {
        productList = null;
    }
}

public static final Creator<ShoppingList> CREATOR = new Creator<ShoppingList>() {
    @Override
    public ShoppingList createFromParcel(Parcel in) { return new ShoppingList(in); }

    @Override
    public ShoppingList[] newArray(int size) { return new ShoppingList[size]; }
};

@Override
public int describeContents() { return 0; }

@Override
public void writeToParcel(Parcel dest, int flags) {
    dest.writeString(title);
    dest.writeInt(itemCount);
    if (productList == null)
    {
        dest.writeByte((byte) (0x00));
    }
    else
    {
        dest.writeByte((byte) (0x01));
        dest.writeList(productList);
    }

}
```

Figure 31. *Parcelable* implementation in *ShoppingList* class

Task #9. List saves changes when switching between activities

For this task, we slightly changed how the list activity is called from *ListFragment.java* class and how the *ListActivity.java* finishes. First, we start the activity with *startActivityForResult()* method, which indicates that we will want to receive changes from within the activity. The changes are then processed inside *onActivityResult()* method (Figure 32).

```
113     @Override
114     public void onListClick(int position) {
115         activeIndex = position;
116
117         Intent intent = new Intent(getContext(), ListActivity.class);
118         intent.putExtra( name: "list_object", shoppingList.get(position));
119         startActivityForResult(intent,  requestCode: 101, ActivityOptions.makeSceneTransitionAnimation(getActivity()).toBundle());
120     }
121
122     @Override
123     public void onActivityResult(int requestCode, int resultCode, @Nullable Intent data) {
124         ShoppingList resultList = data.getParcelableExtra( name: "list_object");
125         if (requestCode == 101 && resultCode == Activity.RESULT_OK)
126         {
127             shoppingList.set(activeIndex, resultList);
128             shoppingList.get(activeIndex).itemCount = resultList.itemCount;
129             adapter.UpdateList(shoppingList);
130         }
131     }
```

Figure 32. *onListClick()* and *onActivityResult()* methods

Then, inside the *ListActivity*, when we either press the back button inside the application, or the android back button, we call *onBackPressed()* method, which adds the activity data with all the changes (if there were made any) into a new intent and sends it back to the main activity (*onActivityResult()* method) (Figure 33).

```
196     @Override
197     public boolean onOptionsItemSelected(@NonNull MenuItem item) {
198         switch (item.getItemId())
199         {
200             case android.R.id.home:
201                 onBackPressed();
202                 return true;
203             case R.id.appBarAddNew:
204                 openAddProductDialogue();
205                 return true;
206         }
207         return super.onOptionsItemSelected(item);
208     }
209
210     @Override
211     public void onBackPressed()
212     {
213         Intent resultIntent = new Intent();
214         resultIntent.putExtra( name: "list_object", data);
215         setResult(Activity.RESULT_OK, resultIntent);
216         this.finishAfterTransition();
217     }
```

Figure 33. *onBackPressed()* method in *ListActivity* class

Task #10. Lists display and update the item count

Now that the changes to lists are being saved, by simply applying the *itemCount* value to the *TextField* attribute of the item inside *onBindViewHolder()* method in *ShoppingListAdapter* (Figure 34), the item count is being changed during runtime (Figure 35).

```
109     @Override
110     public void onBindViewHolder(@NonNull ShoppingListViewHolder holder, int position) {
111         ShoppingList list = shoppingList.get(position);
112
113         holder.textViewList.setText(list.title);
114         holder.textViewCount.setText(String.valueOf(list.itemCount));
```

Figure 34. Implementation of item count change

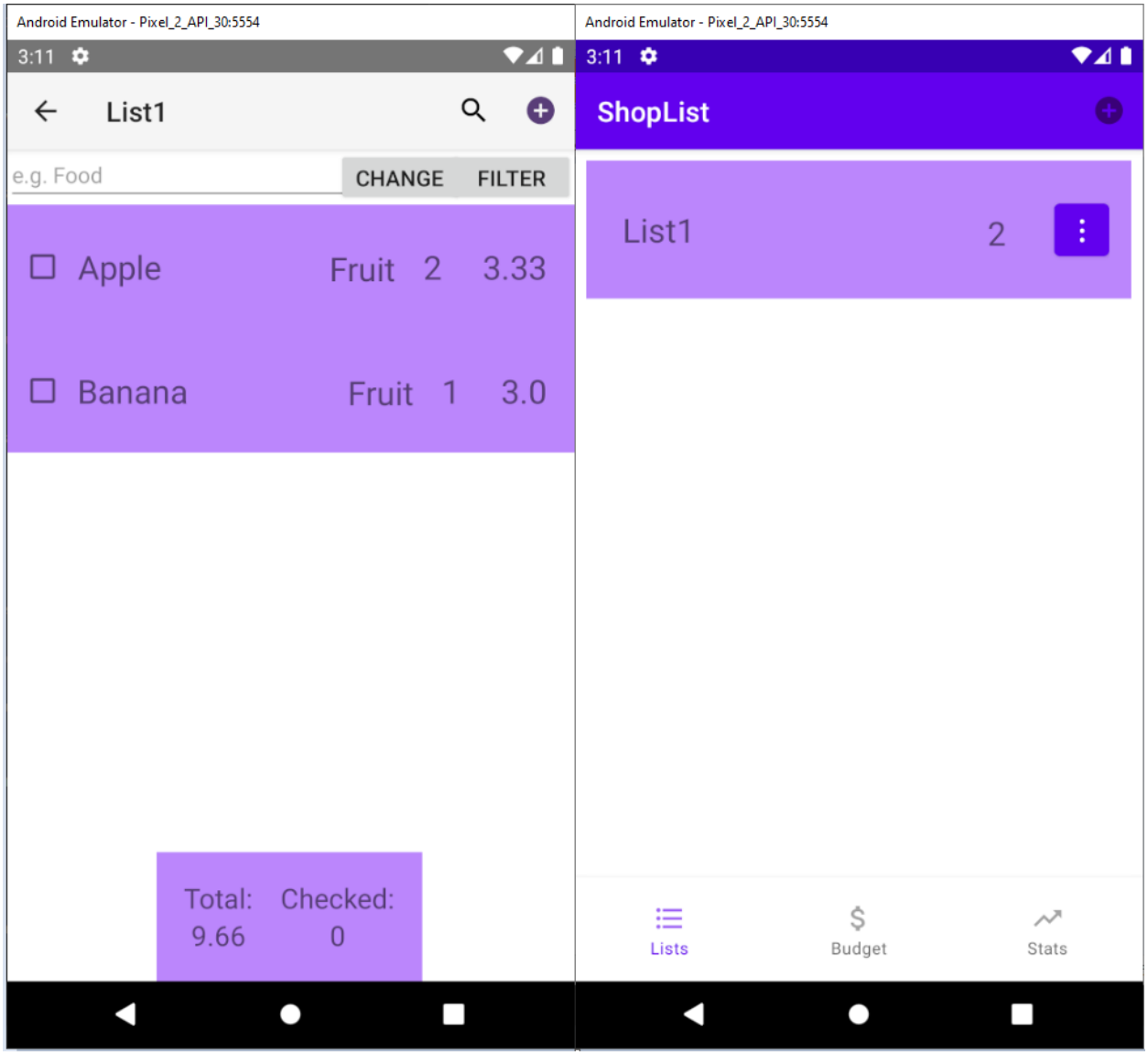


Figure 35. How example list looks and show the item count

Task #11. Add edit and remove product functionality

For this task, we created a new dialogue class called *EditProductDialogue*. An edit box pops out whenever user holds down on a product and you can either edit current values or remove the item entirely (Figure 36).

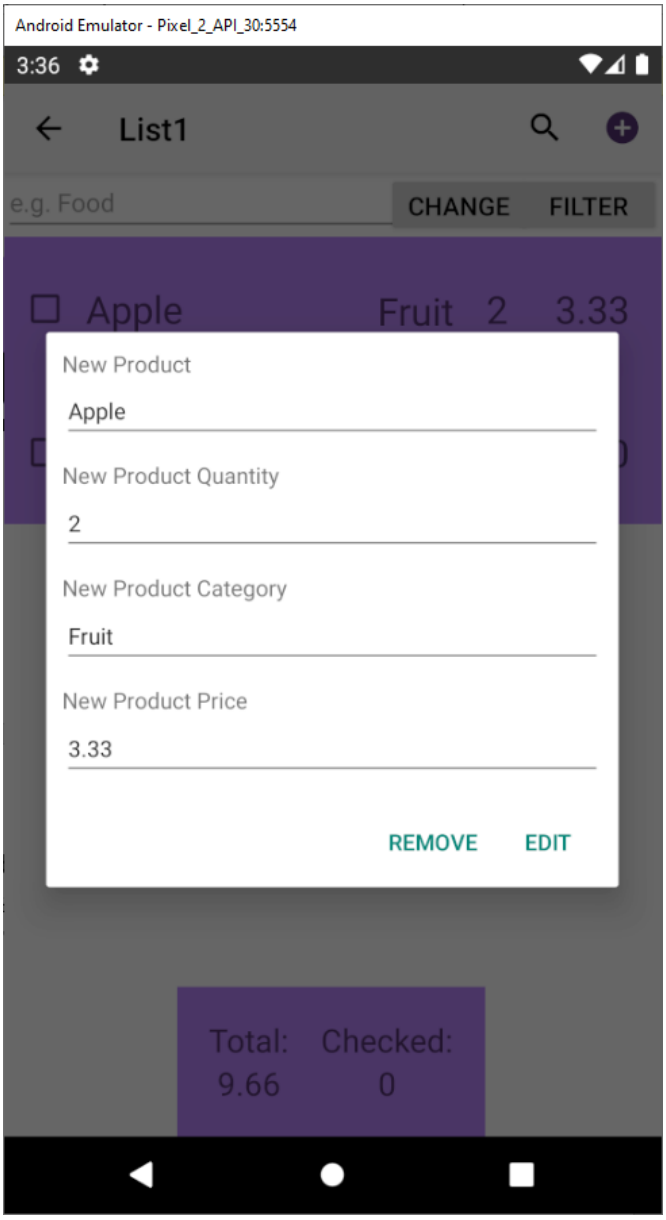


Figure 36. Example of *EditProductDialogue* popup

Main functionality of the dialogue class can be seen in Figure 37 and functionality of the edit, remove actions can be seen in Figure 38, which are implemented in *ListActivity* class.

```java
public Dialog onCreateDialog(@Nullable Bundle savedInstanceState) {
    AlertDialog.Builder builder = new AlertDialog.Builder(getActivity());

    LayoutInflater inflater = getActivity().getLayoutInflater();
    View view = inflater.inflate(R.layout.layout_new_product,  root: null);

    builder.setView(view)
            .setNegativeButton( text: "Remove", new DialogInterface.OnClickListener() {
                @Override
                public void onClick(DialogInterface dialog, int which)
                {
                    listener.removeProduct();
                }
            })
            .setPositiveButton( text: "Edit", new DialogInterface.OnClickListener() {
                @Override
                public void onClick(DialogInterface dialog, int which) {
                    productName = editTextProductName.getText().toString();
                    productCategory = editTextProductCategory.getText().toString();
                    productPrice = Float.parseFloat(editTextProductPrice.getText().toString());
                    productQuantity = Integer.parseInt(editTextProductQuantity.getText().toString());

                    if (productName.equals("") || productCategory.equals("") || productPrice <= 0f || productQuantity <= 0)
                    { Toast.makeText(getContext(),  text: "Field is empty or incorrect", Toast.LENGTH_SHORT).show(); }
                    else
                    { listener.editProduct(productName, productCategory, productPrice, productQuantity); }
                }
            });

    editTextProductName = view.findViewById(R.id.etNewProductName);
    editTextProductName.setText(productName);
    editTextProductCategory = view.findViewById(R.id.etNewProductCategory);
    editTextProductCategory.setText(productCategory);
    editTextProductPrice = view.findViewById(R.id.etNewProductPrice);
    editTextProductPrice.setText(String.valueOf(productPrice));
    editTextProductQuantity = view.findViewById(R.id.etNewProductQuantity);
    editTextProductQuantity.setText(String.valueOf(productQuantity));

    return builder.create();
}
```

Figure 37. *EditProductDialogue* main functionality method

```java
@Override
public void onProductClick(int position) { onCheck(position); }

@Override
public void onProductLongClick(int position) {
    // Implement menu to remove the product or edit
    editIndex = position;

    EditProductDialogue editProdDial = new EditProductDialogue(data.productList.get(editIndex));
    editProdDial.show(getSupportFragmentManager(),  tag: "editProduct");
}

@Override
public void editProduct(String productName, String productCategory, float productPrice, int productQuantity)
{
    ProductEntry e = data.productList.get(editIndex);
    e.product.name = productName;
    e.product.category = productCategory;
    e.product.price = productPrice;
    e.quantity = productQuantity;
    data.EditProduct(e, editIndex);
    adapter.notifyItemChanged(editIndex);
    CalculateCosts(data.productList);
}

@Override
public void removeProduct()
{
    data.RemoveProduct(editIndex);
    adapter.notifyItemRemoved(editIndex);
    CalculateCosts(data.productList);
}
```

Figure 38. Implementation of edit, remove and dialogue popup actions

Task #12. Add the ability to check total cost of a list without entering it

For this task, we created a method called *GetListCost()* in *ShoppingList* class, which calculates cost of every product of the list and then returns it (Figure 39).

```
92      public float GetListCost()
93      {
94          totalCost = 0;
95          for (ProductEntry e : productList)
96          {
97              totalCost += e.getCost();
98          }
99          return totalCost;
100     }
```

Figure 39. *GetListCost()* method in *ShoppingList* class

Later, whenever user holds his finger on a specific list, a *Toast* notification will appear indicating the cost of the list (Figure 40).

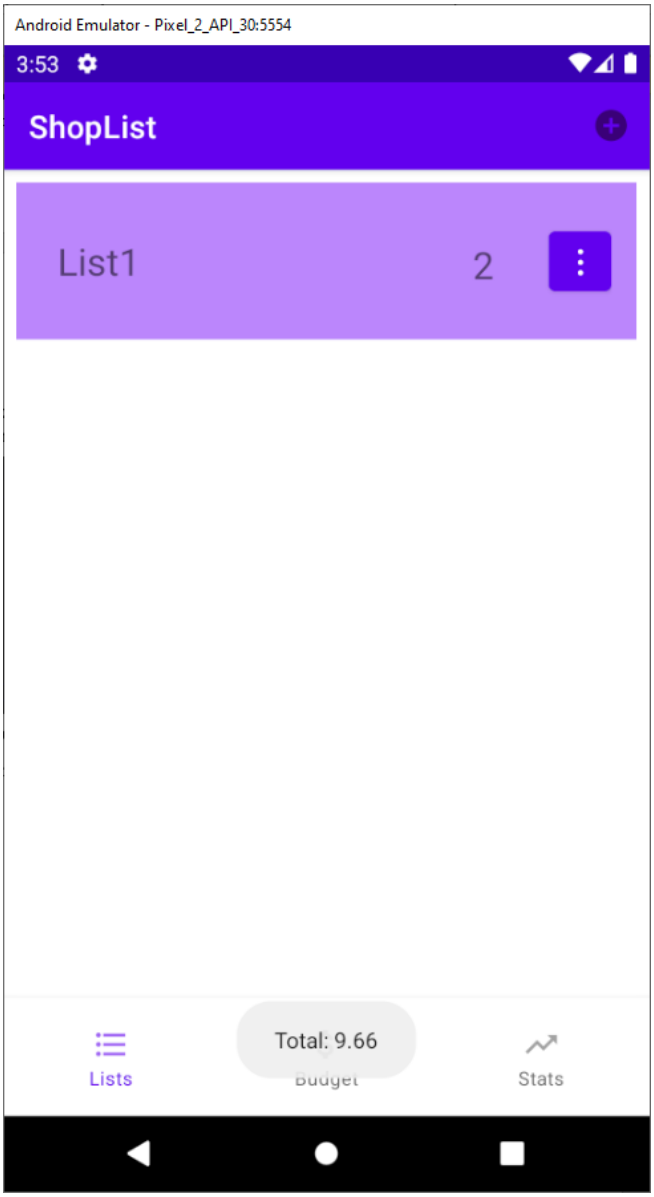


Figure 40. Example of entire list cost check functionality

Task #13. Implement list budget functionality

For this task, we updated the layout of the budget fragment, so you could specify your allowance (Figure 41).

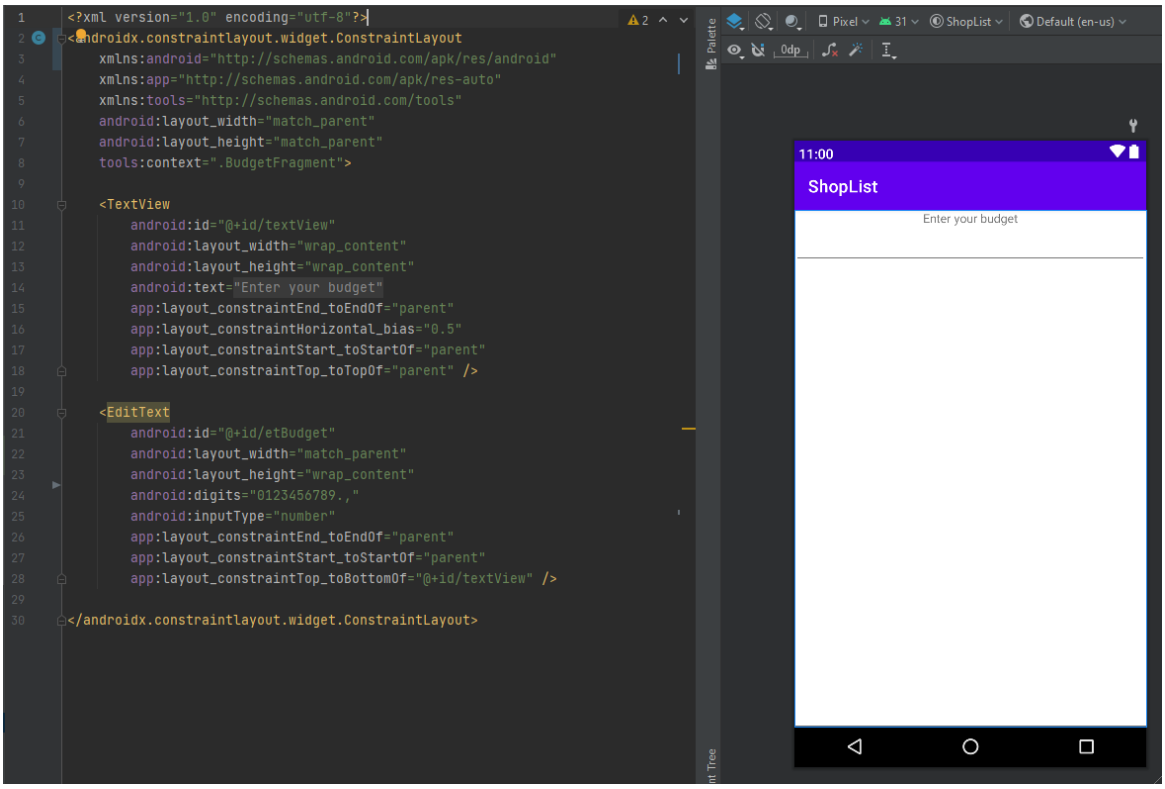


Figure 41. Layout of the budget fragment

Inside *BudgetFragment* class, the budget is changed whenever the *EditText* field changes. This is handled inside *afterTextChanged()* method (Figure 42) which calls *MainActivity* static method called *setBudgetLimit()*, which simply changes the budget variable.

Whenever a list exceeds the specified limit (if budget limit is not zero), the list item will be tinted red and when entering the list or adding more products, user will be shown a *Toast* notification, indicating the current limit (Figure 43).

The color change is implemented inside *ShoppingListAdapter* class, where it checks whether a list exceeds the budget. If it does, then it changes the background color, if it doesn‘t, it reverts the color to the default (purple) color (Figure 44).

The *Toast* is implemented inside *ListActivity* class‘s *CalculateCosts()* method, so whenever a product is added, edited or removed, it checks whether the list exceeds the budget limit (Figure 45).

```java
    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
                             Bundle savedInstanceState) {
        View fragView = inflater.inflate(R.layout.fragment_budget, container,  attachToRoot: false);

        etBudget = fragView.findViewById(R.id.etBudget);
        etBudget.setText(String.valueOf(MainActivity.budgetLimit));

        etBudget.addTextChangedListener(new TextWatcher() {
            @Override
            public void beforeTextChanged(CharSequence s, int start, int count, int after) {

            }

            @Override
            public void onTextChanged(CharSequence s, int start, int before, int count) {

            }

            @Override
            public void afterTextChanged(Editable s) {
                if (!etBudget.getText().equals("") && etBudget.getText().length() > 0)
                {
                    MainActivity.setBudgetLimit(Float.parseFloat(etBudget.getText().toString()));
                }
                else
                {
                    MainActivity.setBudgetLimit(0);
                }
            }
        });

        return fragView;
    }
}
```

Figure 42. *BudgetFragment* class main functionality

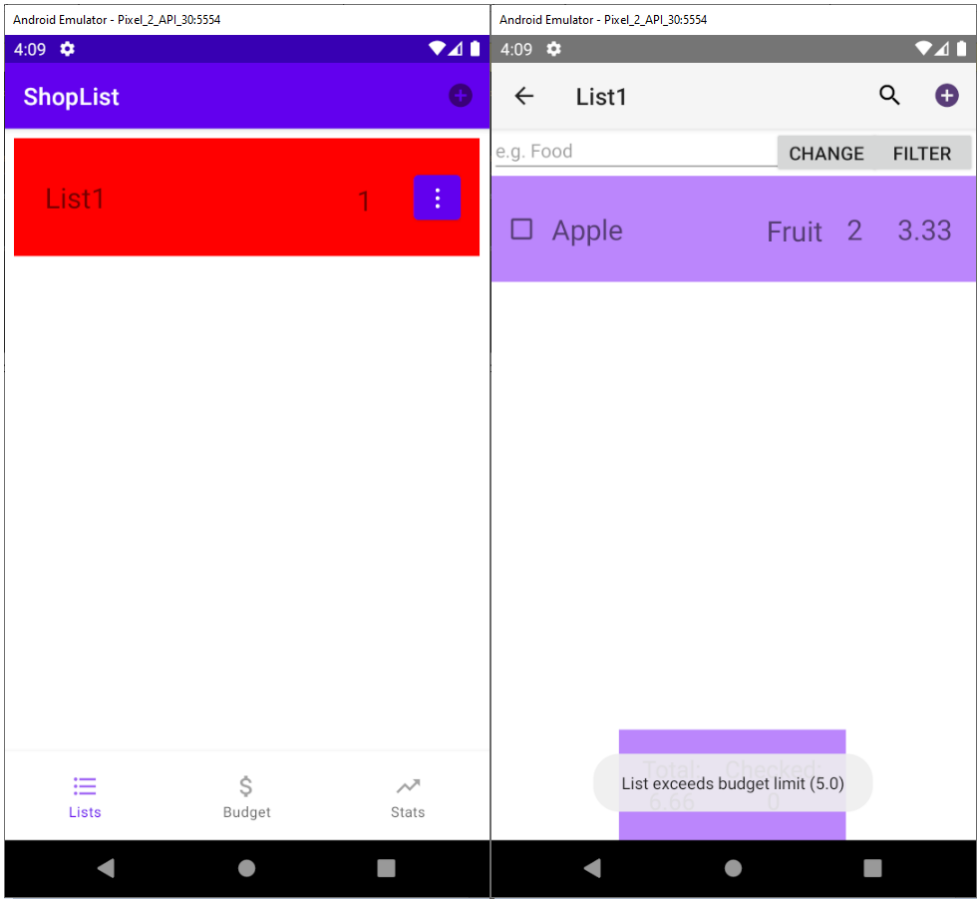


Figure 43. Example of budget functionality when budget is set to 5

```
109        @Override
110        public void onBindViewHolder(@NonNull ShoppingListViewHolder holder, int position) {
111            ShoppingList list = shoppingList.get(position);
112            //Checks if list exceeds budget
113            if (MainActivity.budgetLimit > 0 && list.GetListCost() > MainActivity.budgetLimit)
114            {
115                holder.view.setBackgroundColor(Color.RED);
116            }
117            else
118            {
119                holder.view.setBackgroundColor(holder.defaultColor);
120            }
```

Figure 44. Implementation of color change when list exceeds budget limit

```
166 @      public void CalculateCosts(List<ProductEntry> list)
167        {
168            float total = 0, checked = 0;
169            for (ProductEntry e : list)
170            {
171                total += e.getCost();
172                if (e.isChecked) checked += e.getCost();
173            }
174            DecimalFormat frmt = new DecimalFormat( pattern: "#.##");
175            tvTotal.setText(frmt.format(total));
176            tvChecked.setText(frmt.format(checked));
177
178            if (MainActivity.budgetLimit > 0 && total > MainActivity.budgetLimit)
179            {
180                Toast.makeText( context: this,  text: "List exceeds budget limit (" + MainActivity.budgetLimit + ")", Toast.LENGTH_SHORT).show();
181            }
182        }
```

Figure 45. Implementation of *Toast* popup when list exceeds budget limit

Task #14. Implement a reminder after specified time

For this task, we updated the list menu layout with a “Remind” option. Whenever user selects it, a new dialogue box will pop up where he can specify the time after which a notification will pop up reminding him of the list (Figures 46).

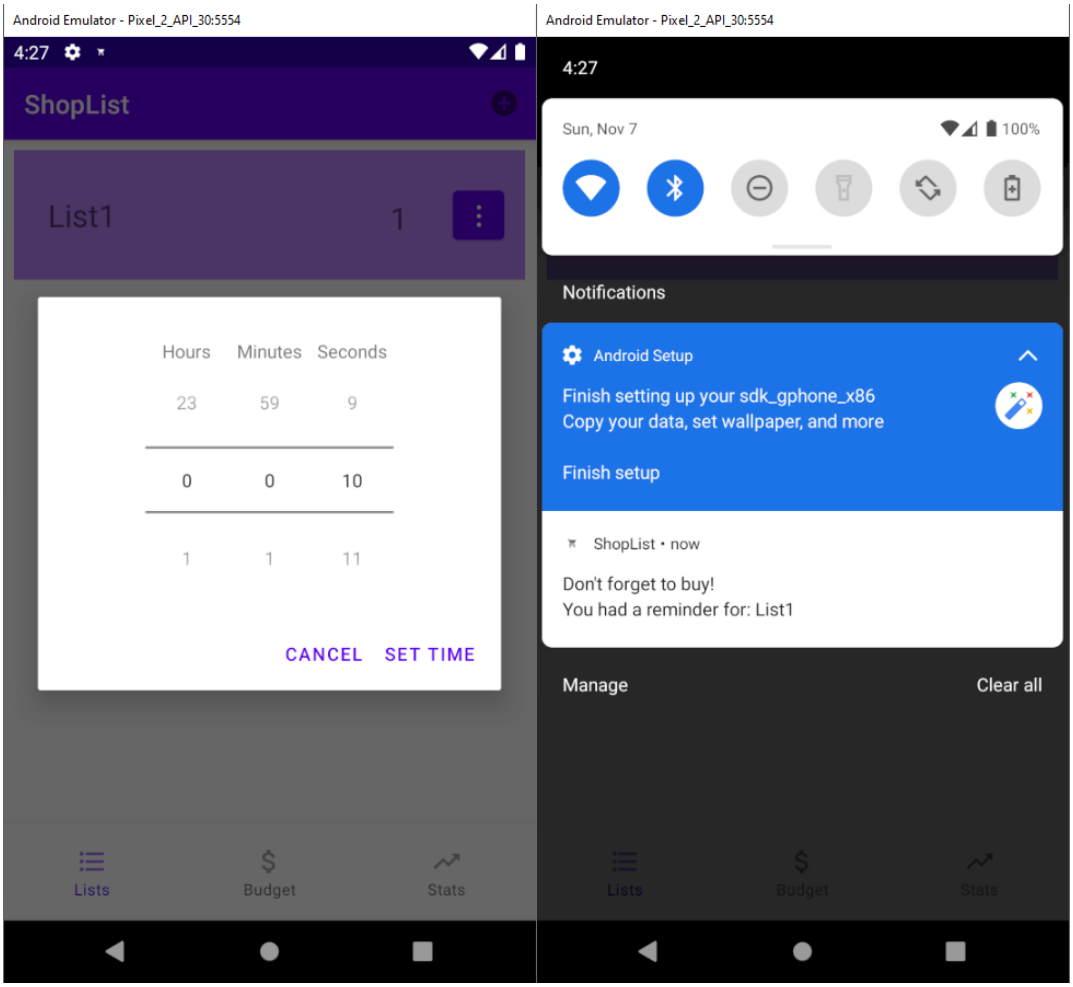
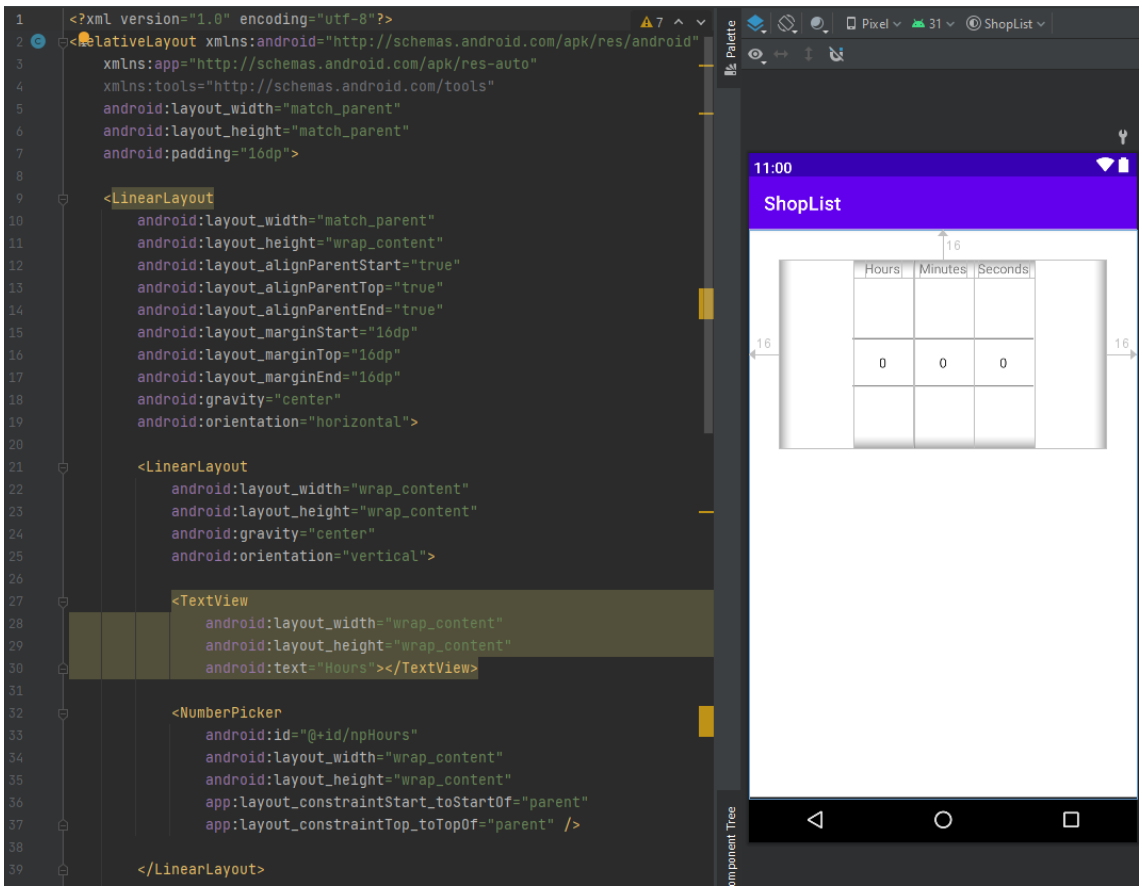


Figure 46. Example functionality of the reminder system

A fragment of the layout for the dialogue box can be seen in Figure 47. *TimePickerDialogue* class is responsible for creating the dialogue box. Main functionality of the class can be seen in Figure 48.



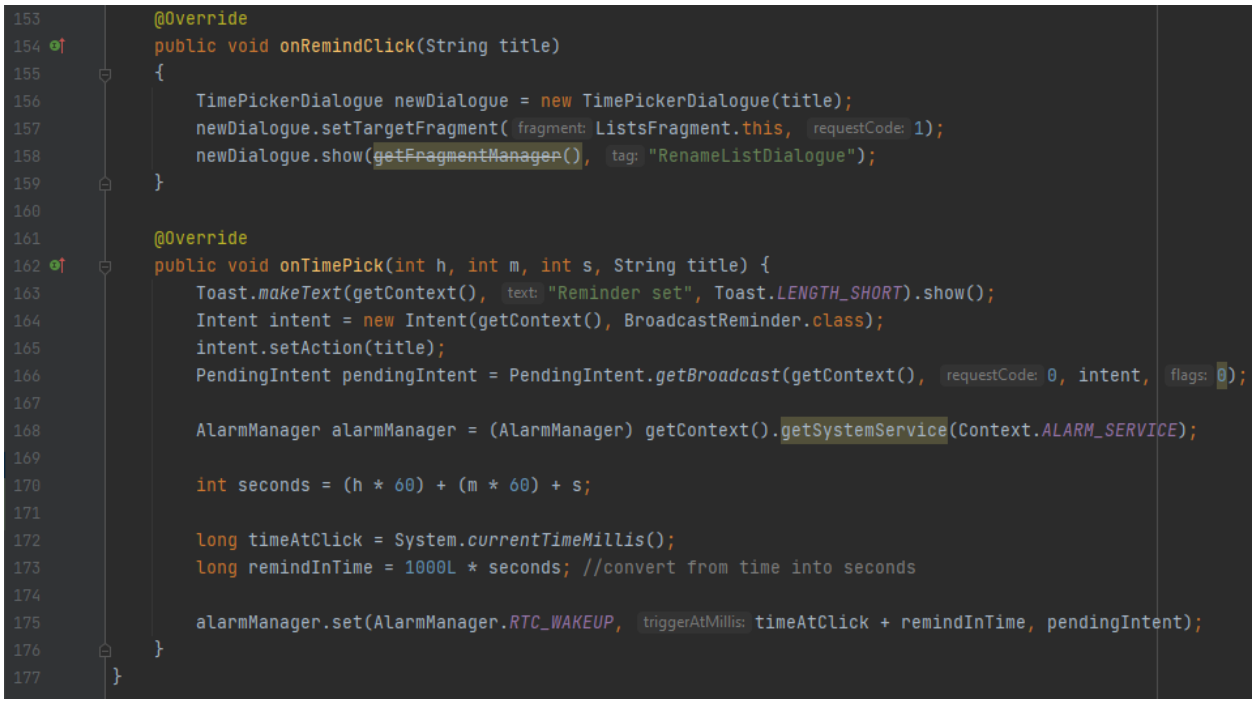

```xml
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="16dp">

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_alignParentStart="true"
        android:layout_alignParentTop="true"
        android:layout_alignParentEnd="true"
        android:layout_marginStart="16dp"
        android:layout_marginTop="16dp"
        android:layout_marginEnd="16dp"
        android:gravity="center"
        android:orientation="horizontal">

        <LinearLayout
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:gravity="center"
            android:orientation="vertical">

            <TextView
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:text="Hours"></TextView>

            <NumberPicker
                android:id="@+id/npHours"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                app:layout_constraintStart_toStartOf="parent"
                app:layout_constraintTop_toTopOf="parent" />

        </LinearLayout>
```

Figure 47. Small fragment of the time picking dialogue box layout

```java
    public Dialog onCreateDialog(@Nullable Bundle savedInstanceState) {
        AlertDialog.Builder builder = new AlertDialog.Builder(getActivity());

        LayoutInflater inflater = getActivity().getLayoutInflater();
        View view = inflater.inflate(R.layout.layout_pick_time, root: null);

        builder.setView(view)
                .setNegativeButton( text: "Cancel", new DialogInterface.OnClickListener() {
                    @Override
                    public void onClick(DialogInterface dialog, int which) {

                    }
                })
                .setPositiveButton( text: "Set time", new DialogInterface.OnClickListener() {
                    @Override
                    public void onClick(DialogInterface dialog, int which) {
                        hours = npHour.getValue();
                        minutes = npMinute.getValue();
                        seconds = npSecond.getValue();
                        listener.onTimePick(hours, minutes, seconds, listTitle);
                    }
                });

        npHour = view.findViewById(R.id.npHours);
        npHour.setMaxValue(23);
        npMinute = view.findViewById(R.id.npMinutes);
        npMinute.setMaxValue(59);
        npSecond = view.findViewById(R.id.npSeconds);
        npSecond.setMaxValue(59);

        return builder.create();
    }


}
```

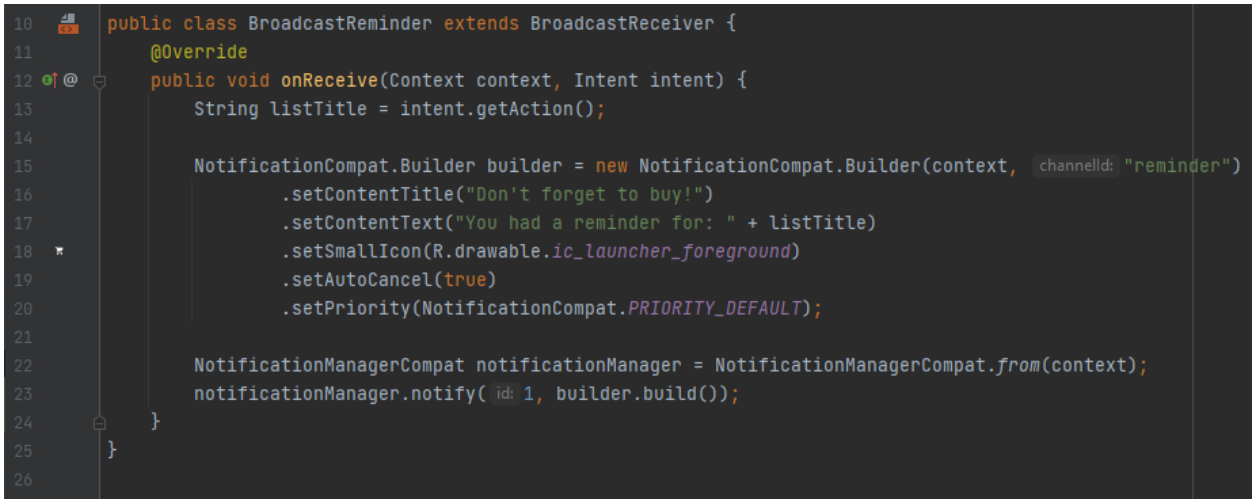
Figure 48. Main functionality part of the *TimePickerDialogue* class

Whenever the time is set, *onTimePick()* method is called inside *ListsFragment* class (Figure 49), which sets up the notification using a custom *BroadcastReminder* class (Figure 50).



```
    @Override
    public void onRemindClick(String title)
    {
        TimePickerDialogue newDialogue = new TimePickerDialogue(title);
        newDialogue.setTargetFragment( fragment: ListsFragment.this,  requestCode: 1);
        newDialogue.show(getFragmentManager(),  tag: "RenameListDialogue");
    }

    @Override
    public void onTimePick(int h, int m, int s, String title) {
        Toast.makeText(getContext(),  text: "Reminder set", Toast.LENGTH_SHORT).show();
        Intent intent = new Intent(getContext(), BroadcastReminder.class);
        intent.setAction(title);
        PendingIntent pendingIntent = PendingIntent.getBroadcast(getContext(),  requestCode: 0, intent,  flags: 0);

        AlarmManager alarmManager = (AlarmManager) getContext().getSystemService(Context.ALARM_SERVICE);

        int seconds = (h * 60) + (m * 60) + s;

        long timeAtClick = System.currentTimeMillis();
        long remindInTime = 1000L * seconds; //convert from time into seconds

        alarmManager.set(AlarmManager.RTC_WAKEUP,  triggerAtMillis: timeAtClick + remindInTime, pendingIntent);
    }
}
```

Figure 49. *onTimePick()* method inside *ListsFragment* class



```
public class BroadcastReminder extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent) {
        String listTitle = intent.getAction();

        NotificationCompat.Builder builder = new NotificationCompat.Builder(context,  channelId: "reminder")
                .setContentTitle("Don't forget to buy!")
                .setContentText("You had a reminder for: " + listTitle)
                .setSmallIcon(R.drawable.ic_launcher_foreground)
                .setAutoCancel(true)
                .setPriority(NotificationCompat.PRIORITY_DEFAULT);

        NotificationManagerCompat notificationManager = NotificationManagerCompat.from(context);
        notificationManager.notify( id: 1, builder.build());
    }
}
```

Figure 50. Custom *BroadcastReminder* class for creating a notification

Task #15. Add some interactive and transitional animations

For this task we added a transition animation when going into a list and returning from it (The list slides up from the bottom on enter, and slides down when exiting). Implementation is inside *ListActivity* class and can be seen in Figure 51.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    getWindow().requestFeature(Window.FEATURE_ACTIVITY_TRANSITIONS);
    getWindow().setEnterTransition(new Slide());
    getWindow().setExitTransition(new Slide(Gravity.TOP));
```

Figure 51. Implementation of transition animation between activities

We also added a simple transition between three main fragments. A newly selected fragment slides in from the side while the old fragment fades out. Implementation is in *MainActivity* class and can be seen in Figure 52.

```
private void setCurrentFragment(Fragment frag)
{
    getSupportFragmentManager().beginTransaction()
            .setCustomAnimations(
                    R.anim.slide_in,
                    R.anim.fade_out)
            .replace(R.id.frameLayoutFragment, frag)
            .commit();
}
```

Figure 52. Implementation of transition animation between menu fragments

Lastly, we added a simple transparency animation when clicking on list or product items. The implementations are inside their own *onCreateViewHolder()* methods using *onTouchListener* (Figure 53).

```
public ShoppingListViewHolder onCreateViewHolder(@NonNull ViewGroup parent, int viewType) {
    View view = LayoutInflater.from(parent.getContext()).inflate(R.layout.item_shopping_list, parent, attachToRoot: false);

    view.setOnTouchListener(new View.OnTouchListener() {
        @Override
        public boolean onTouch(View v, MotionEvent event) {
            switch (event.getAction())
            {
                case MotionEvent.ACTION_DOWN:
                {
                    v.animate().alpha(0.5f).setDuration(200).start();
                    break;
                }
                default:
                    v.animate().alpha(1f).setDuration(200).start();
            }
            return false;
        }
    });

    return new ShoppingListViewHolder(view, mOnListListener);
}
```

Figure 53. Implementation of transparency animation on list items

Task #16. Allow connection to a local database for retrieving data

For this task we tried to establish a separate class environment where we could realize all of our database functions. In this case we named the database class DBHelper (Figure 54), it will be used in the future to help establish connections with the database and help retrieve data from it as well as add to it and delete from it. [4]

```
package com.example.shoplist;

import ...

public class DBHelper  extends SQLiteOpenHelper {

    public static final String PRODUCT_TABLE = "PRODUCT_TABLE";
    public static final String COLUMN_ID = "ID";
    public static final String COLUMN_PRODUCT_NAME = "PRODUCT_NAME";
    public static final String COLUMN_PRODUCT_CATEGORY = "PRODUCT_CATEGORY";
    public static final String COLUMN_PRODUCT_PRICE = "PRODUCT_PRICE";
    public static final String COLUMN_PRODUCT_QUANTITY = "PRODUCT_QUANTITY";
    public static final String COLUMN_PRODUCT_CHECKED = "PRODUCT_CHECKED";
    public static final String COLUMN_PRODUCT_LIST_TITLE = "PRODUCT_LIST_TITLE";
    public static final String COLUMN_PRODUCT_LIST_COUNT = "PRODUCT_LIST_COUNT";


    public DBHelper(@Nullable Context context) { super(context, name: "shoppingList.db", factory: null, version: 1); }

    @Override
    public void onCreate(SQLiteDatabase db) {
        String createTable = "CREATE TABLE " + PRODUCT_TABLE + " (" + COLUMN_ID + " INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL, " + COLUMN_PRODUCT_NAME + ", " +
                COLUMN_PRODUCT_CATEGORY + ", " + COLUMN_PRODUCT_PRICE + ", " + COLUMN_PRODUCT_QUANTITY + ", " + COLUMN_PRODUCT_CHECKED + ", "
                + COLUMN_PRODUCT_LIST_TITLE + ", " + COLUMN_PRODUCT_LIST_COUNT + ")";

        db.execSQL(createTable);

    }
```

Figure 54. Basic body of the DBHelper class and *onCreate()* method

In this class we mark what we need in order to make the database useful for the project. Since our shopping lists are made up of a ProductEntry list and the ProductEntry class is made up of another Product class we needed to make sure we have all the necessary variables to reach any variable at any time without the use of addiontal databases.

Also, so that we would have a strict numbered list of products we have decided to add an ID int variable to the Product class so we could track the products via ID instead of indexes and other ways. (Figure 55)

```java
public class Product implements Parcelable {
    int id;
    String name;
    String category;
    float price;

    public Product()
    {
        int id = 0;
        name = "";
        category = "";
        price = 0f;
    }

    public Product(String name, String category, float price)
    {
        this.name = name;
        this.category = category;
        this.price = price;
    }
    public Product(int id, String name, String category, float price)
    {
        this.id = id;
        this.name = name;
        this.category = category;
        this.price = price;
    }
```

Figure 55 Modified Product class with ID variable

Task #17. Allow adding objects to a database via function

To be able to make use of the database one of the most important things is to be able to add new information to it. For this reason we have created a simple function that would go through an entire ShoppingList class, find all the ProductEntry classes and use them to put in the required variables for the database. (Figure 56)

```
public boolean addToDatabase(ShoppingList shoppingList)
{
    SQLiteDatabase db = this.getWritableDatabase();
    ContentValues cv = new ContentValues();

    for (ProductEntry prodEntry:
         shoppingList.productList)
    {
        cv.put(COLUMN_PRODUCT_NAME, prodEntry.product.getName());
        cv.put(COLUMN_PRODUCT_CATEGORY, prodEntry.product.getCategory());
        cv.put(COLUMN_PRODUCT_PRICE, prodEntry.product.getPrice());
        cv.put(COLUMN_PRODUCT_QUANTITY, prodEntry.quantity);
        cv.put(COLUMN_PRODUCT_CHECKED, prodEntry.isChecked);
        cv.put(COLUMN_PRODUCT_LIST_TITLE, shoppingList.title);
        cv.put(COLUMN_PRODUCT_LIST_COUNT, shoppingList.productList.size());
    }

    long insert = db.insert(PRODUCT_TABLE,  nullColumnHack: null, cv);
    if (insert == -1) { return false; }
    else { return true; }
}
```

Figure 56. *addToDatabase()* method that allows the user to add variables to the database

At the end of the function the insert value determines whether the data was added successfully or not. If the insert value is -1 then the insertion of data failed, if it is any other number then it has successfully been complete. Also, it is important to point out that the ID variable is not mentioned here as it is set to AutoIncrement in the main body of the database class.

Task #18. Allow retrieving data from the database for display

Just as important as inserting data it is important that we would be able to see it as well. In this case the data is being exported as a ShoppingList class where all of the variables are stored. Here we run a query through the database to select all the columns and their variables from the table via Cursor. This cursor moves from the start of the table and reads every line. After this the recorded data is set into bigger list classes, this is done until there are no more columns in the table and the end result is returned as a ShoppingList class. (Figure 57)

```
public ShoppingList getAll()
{
    String listTitle = "";
    int listCount = 0;
    List<ProductEntry> returnProducts = new ArrayList<>();

    String query = "SELECT * FROM " + PRODUCT_TABLE;
    SQLiteDatabase db = this.getReadableDatabase();

    Cursor cursor = db.rawQuery(query,  selectionArgs: null);

    if (cursor.moveToFirst())
    {
        do {
            int productID = cursor.getInt( i: 0);
            String productName = cursor.getString( i: 1);
            String productCategory = cursor.getString( i: 2);
            float productPrice = cursor.getFloat( i: 3);
            int productQuantity = cursor.getInt( i: 4);
            boolean productChecked = cursor.getInt( i: 5) == 1 ? true: false;
            listTitle = cursor.getString( i: 6);
            listCount = cursor.getInt( i: 7);

            ProductEntry newEntry = new ProductEntry(new Product(productID, productName,
                    productCategory, productPrice), productQuantity, productChecked);
            returnProducts.add(newEntry);

        } while (cursor.moveToNext());

    }
    ShoppingList shopList = new ShoppingList(listTitle, returnProducts);
    cursor.close();
    db.close();
    return shopList;
}
```

Figure 57. *getAll()* method from the database class where all the data of the database is gotten for display

Task #19. Allow deleting data inside the database

To not overcrowd the database it is important to be able to delete data as well. We have devised a simple function to handle this. The method finds the cell with the ID that is provided as an index and removes it instantly via SQL. This method works the same way as the two before it so there is not much to comment about the functionality of it. (Figure 58)

```
public boolean deleteProductFromDatabase(int index)
{
    SQLiteDatabase db = this.getWritableDatabase();
    String query = "DELETE FROM " + PRODUCT_TABLE + " WHERE " + COLUMN_ID + " = " + index;

    Cursor cursor = db.rawQuery(query,  selectionArgs: null);
    if (cursor.moveToFirst())
        return true;
    else return false;

}
```

Figure 58. *deleteProductFromDatabase()* method that deletes parts of the table depending on the index

Task #20. Integrate the database functions with the already existing classes

To make everything work we had to replace the current code that was written with Intents and other ways of transferring data to taking all the information from the database where it is all stored. The *addToDatabase()* method was used in the *applyProduct()* method so that when a new product is added it would be added to the database instead. (Figure 59) Before making the changes there is an *if* statement that clarifies if the list where we would like to add the Product is in the same list as we are currently in. After that we add everything we need and display a small text notification to let the user know the data was saved to the database.

```
@Override
public void applyProduct(String listTitle, String productName, String productCategory, float productPrice, int productQuantity)
{
    List<ProductEntry> listProd = new ArrayList<>();
    Product prod = new Product(productName, productCategory, productPrice);

    DBHelper dbHelper = new DBHelper( context: this);
    if (listTitle.equals(data.title))
    {
        listProd.add(new ProductEntry(prod, productQuantity, isChecked: false));
        boolean worked = dbHelper.addToDatabase(new ShoppingList(listTitle, listProd));
        Toast.makeText( context: this, text: "Added new values to database " , Toast.LENGTH_SHORT).show();
        adapter.notifyItemInserted( position: data.productList.size() - 1);
        CalculateCosts(data.productList);
    }
}
```

Figure 59. database method used in another function to add more objects to the database

Now instead of using the Parcable class we use the database helper we created and simply import the data from the database straight to the empty Shopping List variable we had created before. (Figure 60)

```
DBHelper dbHelper = new DBHelper( context: this);
data = dbHelper.getAll();
```

Figure 60. Real use of database helper to retrieve all the database values

We also modified the *removeProduct()* method to be used with a database so that values would be deleted from the database instead of just for show.

```
@Override
public void removeProduct()
{
    DBHelper dbHelper = new DBHelper( context: this);
    dbHelper.deleteProductFromDatabase(editIndex);
    adapter.notifyItemRemoved(editIndex);
    CalculateCosts(data.productList);
}
```

Figure 61. *removeProduct()* method with database integration

Task #21. Use new tools and options to find database file location

In order to know for sure that the database was edited correctly and there are no bugs we double checked it with a software program called *DB Browser for SQLite.* In this program we were able to open .db files and see all the information we needed as well as edit anything that wasn't desired. To find the .db file of the application we had to navigate through these menus *View -> Tool Windows -> Device File Explorer*, then find the file in the rest of the hierarchy. With the help of this additional software we were able to closely examine and watch the changes made in the database. (Figure 62)

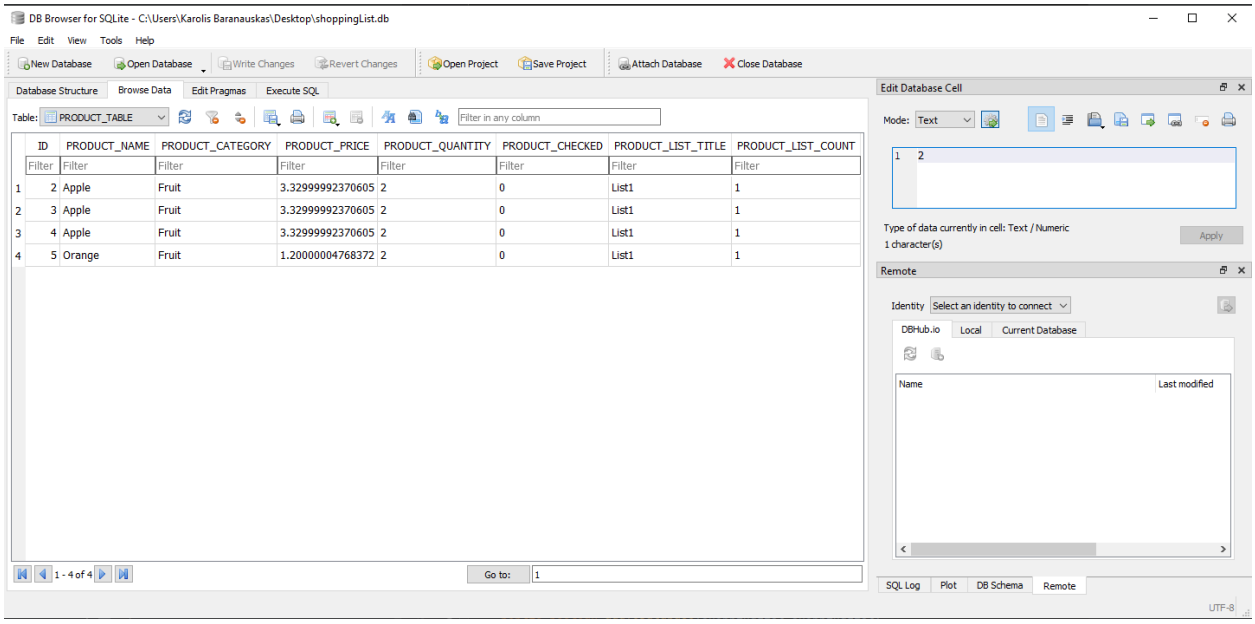


Figure 62. Screenshot of the *DB Browser for SQLite* software

Defense Task #3. Implement value rate converter through JSON

For this task, we registered to [Foreign exchange rates and currency conversion API (fixer.io)] and got latest value exchanges rates in a .json format. Then inside *BudgetFragment* class, we implemented a simple json converter and multiplied our budget from *EUR* to one different rate value (*AED*) (Figures 63 and 64).

```
public static final String ParsingData = "{\"data\":{\"success\":\"true\",\"timestamp\":\"1636354863\",\"base\":\"EUR\",\"date\":\"2021-11-08\",\"rate\":\"4.248225\"}}";
float conv;

public BudgetFragment() {

}


@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);


}

@Override
public View onCreateView(LayoutInflater inflater, ViewGroup container,
                         Bundle savedInstanceState) {
    View fragView = inflater.inflate(R.layout.fragment_budget, container, attachToRoot: false);

    etBudget = fragView.findViewById(R.id.etBudget);
    etBudget.setText(String.valueOf(MainActivity.budgetLimit));

    textView1 = fragView.findViewById(R.id.tvRate);
    tvConv = fragView.findViewById(R.id.tvCoverted);
    try
    {

        JSONObject jsonObject = new JSONObject(ParsingData).getJSONObject("data");
        String success = jsonObject.getString( name: "success");
        String timestamp = jsonObject.getString( name: "timestamp");
        String base = jsonObject.getString( name: "base");
        String date = jsonObject.getString( name: "date");
        String rate = jsonObject.getString( name: "rate");

        String str = "Success:" + success + "\nstamp:" + timestamp + "\nbase:" + base + "\ndate:" + date + "\nrate:" + rate;
        textView1.setText(str);
        conv = Float.valueOf(rate) * MainActivity.budgetLimit;
    }
    catch (JSONException e) {}
```

Figure 63. Implementation of JSON converter

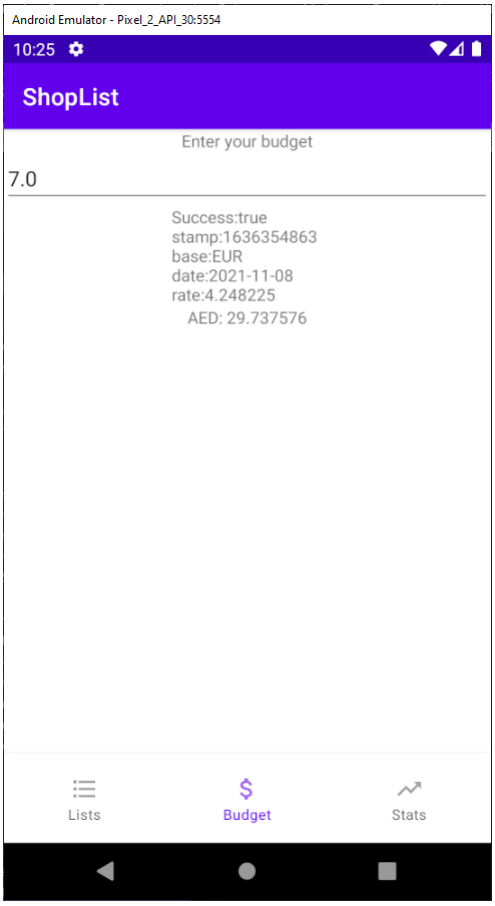


Figure 64. Example of value converting

Task #22. Update budget fragment layout

For this task we fixed the layout of the budget fragment after defense. A spinner was added via which you can select a currency to convert euros into. Under it the conversion result is shown (see figure 65).

Figure 65. New budget fragment layout

Task #23. Fix parsing from JSON file and conversion in Budget fragment

To fix the parsing, we made an ArrayList and a List to save currency rate data into local variables. While parsing the *JSON* file, those values are being added to those variables and the spinner layout element (to display the value and its rate) (see Figure 66).

```
private void parseJSON()
{
    try
    {

        JSONObject jsonObject = new JSONObject(readJSON());
        String success = jsonObject.getString( name: "success");
        String timestamp = jsonObject.getString( name: "timestamp");
        String base = jsonObject.getString( name: "base");
        String date = jsonObject.getString( name: "date");
        JSONObject ratesObject = jsonObject.getJSONObject("rates");

        String str = "Success:" + success + "\nstamp:" + timestamp + "\nbase:" + base + "\ndate:" + date;
        Iterator<String> iterator = ratesObject.keys();
        List<String> spinnerValues = new ArrayList<>();
        while (iterator.hasNext())
        {
            String key = iterator.next();
            Float value = Float.valueOf(ratesObject.getString(key));
            HashMap<String, Float> pair = new HashMap<String, Float>();
            pair.put(key, value);
            currencies.add(key);


            spinnerValues.add(key + ": " + String.format("%.2f", value));
            rates.add(pair);
        }
        spinnerAdapter = new ArrayAdapter<String>(getContext(), android.R.layout.simple_spinner_item, spinnerValues);
        spinnerAdapter.setDropDownViewResource(android.R.layout.simple_spinner_dropdown_item);
    }
    catch (JSONException e) {}
}
```

Figure 66. Fixed parsing of the JSON file

After selecting a element inside the spinner, the *onItemSelected* method is called which find the selected rate and converts it from euros. Then it displays the conversion result under the spinner (Figure 67).

```
@Override
public View onCreateView(LayoutInflater inflater, ViewGroup container,
                         Bundle savedInstanceState) {
    View fragView = inflater.inflate(R.layout.fragment_budget, container,  attachToRoot: false);

    //reads from the "currencies" JSON file and parses currency/rate pairs
    parseJSON();
    spRates = fragView.findViewById(R.id.spinnerRates);
    spRates.setAdapter(spinnerAdapter);
    spRates.setOnItemSelectedListener(new AdapterView.OnItemSelectedListener() {
        @Override
        public void onItemSelected(AdapterView<?> parent, View view, int position, long id) {
            currencyIndex = position;
            conv = MainActivity.budgetLimit * rates.get(position).get(currencies.get(position));
            tvConv.setText(currencies.get(currencyIndex) + ": " + String.valueOf(conv));
        }

        @Override
        public void onNothingSelected(AdapterView<?> parent) { currencyIndex = 0; }
    });
```

Figure 67. Fixed conversion of currencies

Task #24. Get current location of the device

For this task, we changed the stats fragment into map fragment and did everything in *MapFragment.java* class. First, we needed to set few permissions in the *AndroidManifest.xml* file. Those permisions are *ACCESS_FINE_LOCATION* and *ACCESS_COARSE_LOCATION* (see figure 68).

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.shoplist">

    <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
    <uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />
    <uses-permission android:name="android.permission.INTERNET" />
```

Figure 68. Needed location permissions

Then we need to ask the user for location permissions and see if he has location enabled (Figure 69).

```
private LocationCallback locCallback = new LocationCallback() {

    @Override
    public void onLocationResult(LocationResult locationResult) {
        Location lastLoc = locationResult.getLastLocation();
        updateLocation(lastLoc);
    }
};

private boolean checkPermissions() {
    return ActivityCompat.checkSelfPermission(getContext(), Manifest.permission.ACCESS_COARSE_LOCATION) == PackageManager.PERMISSION_GRANTED &&
            ActivityCompat.checkSelfPermission(getContext(), Manifest.permission.ACCESS_FINE_LOCATION) == PackageManager.PERMISSION_GRANTED &&
            ActivityCompat.checkSelfPermission(getContext(), Manifest.permission.INTERNET) == PackageManager.PERMISSION_GRANTED;
}

private void requestPermissions() {
    ActivityCompat.requestPermissions(getActivity(), new String[]{
            Manifest.permission.ACCESS_COARSE_LOCATION,
            Manifest.permission.ACCESS_FINE_LOCATION,
            Manifest.permission.INTERNET}, requestCode: 1);
}

private boolean isLocationEnabled() {
    LocationManager locationManager = (LocationManager) getActivity().getSystemService(Context.LOCATION_SERVICE);
    return locationManager.isProviderEnabled(LocationManager.GPS_PROVIDER);
}

@Override
public void
onRequestPermissionsResult(int requestCode, @NonNull String[] permissions, @NonNull int[] grantResults) {
    super.onRequestPermissionsResult(requestCode, permissions, grantResults);

    if (requestCode == 1) {
        if (grantResults.length > 0 && grantResults[0] == PackageManager.PERMISSION_GRANTED) {
            getLocation();
        }
    }
}
```

Figure 69. Permission checking

If everything is good to go, we receive the *Location* object from which we can take device‘s latitude and longitude and save them to local variables (Figures 70 and 71).

```java
@SuppressLint("MissingPermission")
private void getLocation() {

    if (checkPermissions()) {
        if (isLocationEnabled()) {

            locClient.getLastLocation().addOnCompleteListener(new OnCompleteListener<Location>() {
                @Override
                public void onComplete(@NonNull Task<Location> task) {
                    Location location = task.getResult();
                    if (location == null) {
                        LocationRequest locRequest = new LocationRequest();
                        locRequest.setPriority(LocationRequest.PRIORITY_HIGH_ACCURACY);
                        locRequest.setInterval(5);
                        locRequest.setFastestInterval(0);
                        locRequest.setNumUpdates(1);

                        locClient = LocationServices.getFusedLocationProviderClient(getContext());
                        locClient.requestLocationUpdates(locRequest, locCallback, Looper.myLooper());
                    } else {
                        updateLocation(location);
                    }
                }
            });
        } else {
            Toast.makeText(getContext(), text: "Please turn on your location...", Toast.LENGTH_LONG).show();
            Intent intent = new Intent(Settings.ACTION_LOCATION_SOURCE_SETTINGS);
            startActivity(intent);
        }
    } else {
        requestPermissions();
    }
}
```

Figure 70. Code for getting current device's location

```java
public void updateLocation(@NonNull Location location) {
    latitude = String.valueOf(location.getLatitude());
    longitude = String.valueOf(location.getLongitude());
```

Figure 71. Getting the latitude and longitude

Task #25. Display device's location in Google Maps API

For this task, we needed to register in *Google Cloud Platform* from which we got a unique key to use *Maps* SDK. Then, in the *build.gradle* file, we needed to implement *play services* for locations and maps. Also, in the manifest, we needed to write the unique key in the meta-data (Figure 72).

```
<meta-data
    android:name="com.google.android.geo.API_KEY"
    android:value="@string/google_maps_key" />
```

Figure 72. Meta data for google maps in the manifest

We get a reference to the map fragment in *onViewCreated* and later create the map in the *updateLocation* method, if the reference is not null. When the map is ready, inside *onMapReady* method marker is placed at the current location, it zooms closer to it and pops a *Toast* indicating device's location (Figures 73 and 74).

```
@Override
public void onResume() {
    super.onResume();

    //Update location
    getLocation();
}

private OnMapReadyCallback callback = new OnMapReadyCallback() {
    @Override
    public void onMapReady(GoogleMap googleMap) {
        LatLng loc = new LatLng(Double.valueOf(latitude), Double.valueOf(longitude));
        googleMap.addMarker(new MarkerOptions().position(loc).title("Your location marker").icon(BitmapDescriptorFactory.defaultMarker( hue: 100)));
        googleMap.animateCamera(CameraUpdateFactory.newLatLngZoom(loc,  zoom: 12f));
        Toast.makeText(getContext(),  text: "Your latitude: " + latitude + ", longitude: " + longitude, Toast.LENGTH_LONG).show();

    }
};

@Override
public void onViewCreated(@NonNull View view, @Nullable Bundle savedInstanceState) {
    super.onViewCreated(view, savedInstanceState);
    mapFragment = (SupportMapFragment) getChildFragmentManager().findFragmentById(R.id.map);

}

@Override
public View onCreateView(LayoutInflater inflater, ViewGroup container,
                         Bundle savedInstanceState) {

    View fragView = inflater.inflate(R.layout.fragment_map, container,  attachToRoot: false);

    locClient = LocationServices.getFusedLocationProviderClient(getContext());

    getLocation();


    return fragView;
}
```

Figure 73. Creating the map and marker

```
public void updateLocation(@NonNull Location location) {
    latitude = String.valueOf(location.getLatitude());
    longitude = String.valueOf(location.getLongitude());

    if (mapFragment != null) {
        mapFragment.getMapAsync(callback);
    }
}
```

Figure 74. Loading the map after getting location

You can see the result in Figure 75.

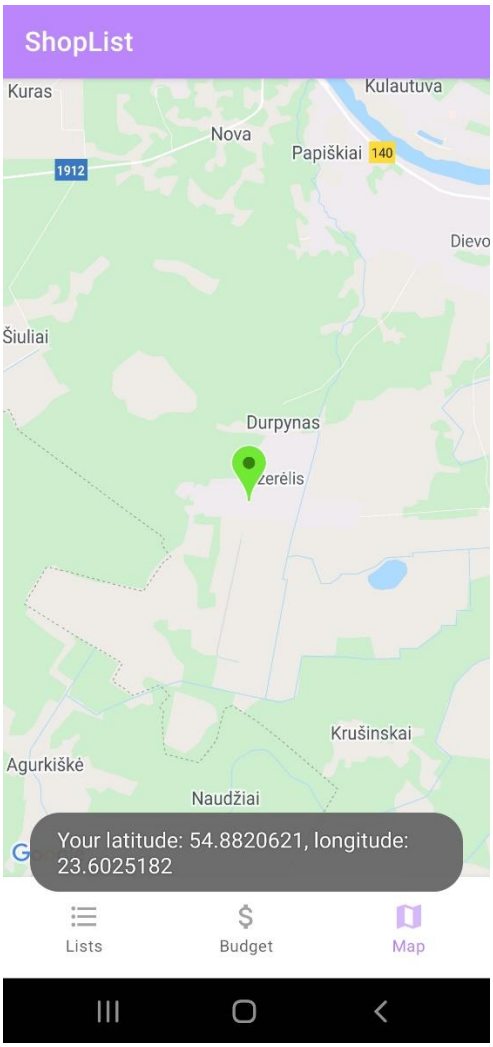


Figure 75. Example of how it looks after opening Map fragment

Task #25. Get and display information from Places API

For this task, we needed to implement *Places* API. We were able to use the same unique key. After getting the information, we saved it into a JSON file which we parsed it by calling *parseJSON* method inside *updateLocation* (Figure 76).

```
private void parseJSON()
{
    try
    {

        //JSONObject jsonObject = new JSONObject(readJSONfromURL());
        JSONObject jsonObject = new JSONObject(readJSONfromFile());

        JSONArray placesArray = jsonObject.getJSONArray( name: "results");

        for (int i = 0; i < placesArray.length(); i++)
        {
            JSONObject placesObject = placesArray.getJSONObject(i);
            JSONObject geometryObject = placesObject.getJSONObject("geometry");
            JSONObject locationObject = geometryObject.getJSONObject("location");

            String name = placesObject.getString( name: "name");
            String lat = locationObject.getString( name: "lat");
            String lng = locationObject.getString( name: "lng");

            Place p = new Place(name, lat, lng);
            places.add(p);
        }
    }
    catch (JSONException e) {}
}
```

Figure 76. Parsing of places data

We also made a custom *Place* class to help us hold the needed data (Figure 77).

```
class Place
{
    String name;
    String placeLat;
    String placeLong;

    public Place(String n, String pLat, String pLong)
    {
        name = n;
        placeLat = pLat;
        placeLong = pLong;
    }
}
```

Figure 77. Custom class for places data

Later, in the *onMapReady* method we create markers for the nearby places and display them on the map (Figure 78 and 79). If you click the marker, it also displays the name of the location.

```
private OnMapReadyCallback callback = new OnMapReadyCallback() {
    @Override
    public void onMapReady(GoogleMap googleMap) {
        LatLng loc = new LatLng(Double.valueOf(latitude), Double.valueOf(longitude));
        googleMap.addMarker(new MarkerOptions().position(loc).title("Your location marker").icon(BitmapDescriptorFactory.defaultMarker( hue: 100)));

        // For nearby locations, need to just loop through them and create markers
        for (Place p : places)
        {
            LatLng place = new LatLng(Double.valueOf(p.placeLat), Double.valueOf(p.placeLong));
            googleMap.addMarker(new MarkerOptions().position(place).title(p.name));
        }

        googleMap.animateCamera(CameraUpdateFactory.newLatLngZoom(loc,  zoom: 12f));
        Toast.makeText(getContext(),  text: "Your latitude: " + latitude + ", longitude: " + longitude, Toast.LENGTH_LONG).show();

    }
};
```

Figure 78. Creating markers for obtained places

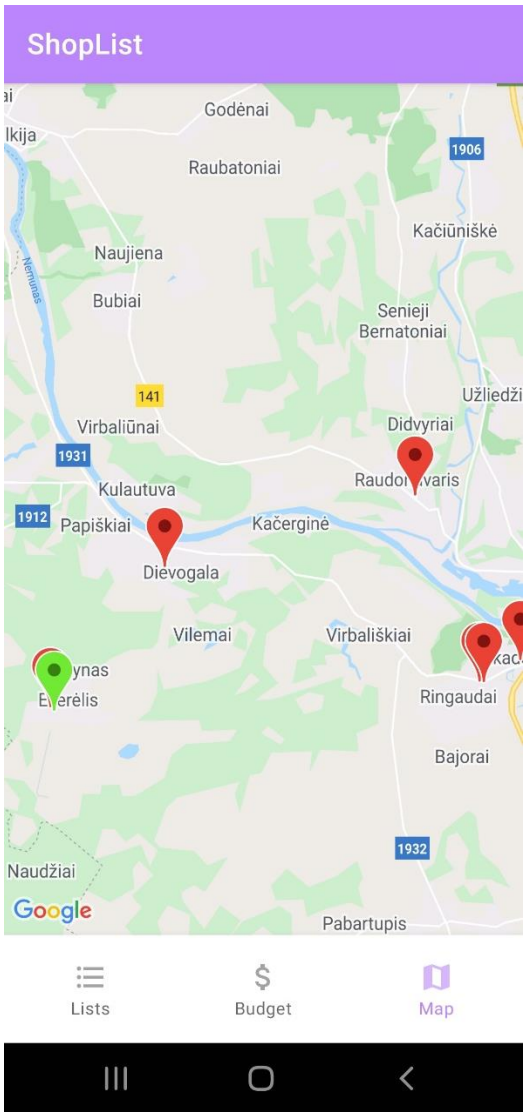


Figure 79. Example of how the additional markers are displayed

Task #26. Save/Read list data to/from external file

For this task, we firstly needed to deal with permissions to read/write to external storage. We updated the manifest file (Figure 80) and handled everything in *ListsFragment.java* class (Figure 81).

```
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE" />
```

Figure 80. Permissions for reading and writing to external storage

```
private boolean checkPermissions() {
    return ActivityCompat.checkSelfPermission(getContext(), Manifest.permission.WRITE_EXTERNAL_STORAGE) == PackageManager.PERMISSION_GRANTED &&
            ActivityCompat.checkSelfPermission(getContext(), Manifest.permission.READ_EXTERNAL_STORAGE) == PackageManager.PERMISSION_GRANTED;
}

private void managePermissions()
{
    ActivityCompat.requestPermissions(getActivity(), new String[]{
            Manifest.permission.WRITE_EXTERNAL_STORAGE,
            Manifest.permission.READ_EXTERNAL_STORAGE}, requestCode: 2);
}

@Override
public void
onRequestPermissionsResult(int requestCode, @NonNull String[] permissions, @NonNull int[] grantResults) {
    super.onRequestPermissionsResult(requestCode, permissions, grantResults);

    if (requestCode == 2) {
        if (grantResults.length > 0 && grantResults[0] == PackageManager.PERMISSION_GRANTED) {
            readFromFile();
        }
    }
}
```

Figure 81. Reading and writing permission handling code

When everything is good with permissions, data from the file (if it exists) is read (Figure 82). After reading the data, data is parsed inside another method (Figure 83). Appropriate objects are created inside that method, the main shopping list object is updated and after everything is done, adapter is updated so the changes are seen in the app (Figure 84).

```
private void readFromFile()
{
    dataDir = getActivity().getFilesDir();
    if (checkPermissions())
    {
        FileReader fr = null;
        File externalFile = new File(dataDir, filename);
        StringBuilder strBuilder = new StringBuilder();
        try {
            fr = new FileReader(externalFile);
            BufferedReader br = new BufferedReader(fr);
            String line = br.readLine();
            while (line != null)
            {
                strBuilder.append(line).append('\n');
                line = br.readLine();
            }
            fr.close();

        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }
        Log.i( tag: "DATAFROMFILE", strBuilder.toString());
        parseData(strBuilder.toString());
    }
    else
    {
        managePermissions();
    }

}
```

Figure 82. *readFromFile()* method

```
private void parseData(String data)
{
    String[] tokens = data.split( regex: "\\r?\\n");
    if (android.os.Build.VERSION.SDK_INT >= android.os.Build.VERSION_CODES.N) {
        Iterator<String> iterator = Arrays.stream(tokens).iterator();
        int listCount = Integer.valueOf(iterator.next());
        for (int i = 0; i < listCount; i++)
        {
            String listData = iterator.next();
            String[] listDataTokens = listData.split( regex: ";");
            String listTitle = listDataTokens[0];
            int productCount = Integer.valueOf(listDataTokens[1]);

            ShoppingList list = new ShoppingList(listTitle);

            for (int j = 0; j < productCount; j++)
            {
                String productData = iterator.next();
                String[] productDataTokens = productData.split( regex: ";");
                String productName = productDataTokens[0];
                String productCategory = productDataTokens[1];
                Float productPrice = Float.valueOf(productDataTokens[2]);
                int productQuantity = Integer.valueOf(productDataTokens[3]);
                boolean productIsChecked = Boolean.valueOf(productDataTokens[4]);

                Product p = new Product(productName, productCategory,productPrice);
                list.AddProduct(p, productQuantity, productIsChecked);
            }
            shoppingList.add(list);
        }
    }
    adapter.UpdateList(shoppingList);
}
```

Figure 83. *parseData()* method

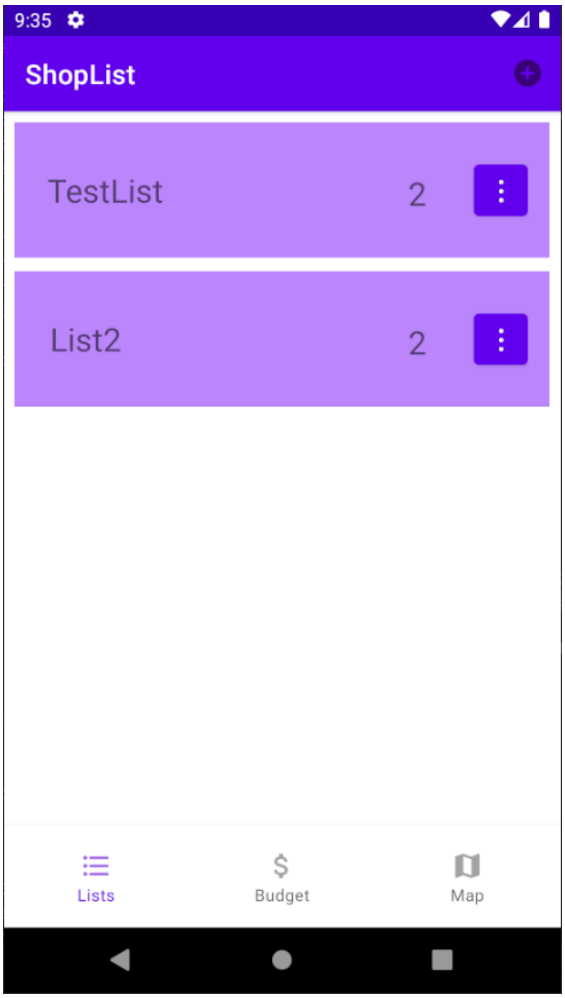


Figure 84. Loaded data in the app

Saving happens whenever user adds, renames, deletes a list or returns from a list. In all of these cases, a method called *saveToFile* is called, which writes all the needed information about every list (Figure 85).

```
private void saveToFile(List<ShoppingList> shoppingList)
{
    String data = "";
    data += shoppingList.size() + "\n";
    for(ShoppingList list : shoppingList)
    {
        data += list.toString();
    }
    Log.i( tag: "ListData", data);

    String storageState = Environment.getExternalStorageState();
    if (storageState.equals(Environment.MEDIA_MOUNTED))
    {
        try
        {
            File externalFile = new File(dataDir, filename);
            if (externalFile.exists())
            {
                externalFile.delete();
            }
            externalFile.createNewFile();

            Log.i( tag: "DIR", externalFile.toString());
            FileOutputStream fos = null;

            fos = new FileOutputStream(externalFile);
            fos.write(data.getBytes());
            fos.close();
        }
        catch (FileNotFoundException e)
        {
            e.printStackTrace();
        }
        catch (IOException e)
        {
            e.printStackTrace();
        }
    }
}
```

Figure 85. *saveToFile()* method

To make the saving slightly easier, we overridden the *toString()* method. It returns all the needed data in a compact form for easy saving (Figure 86). An example of saved data file can be seen in figure 87.

```
@Override
public String toString()
{
    String listData = "";

    listData += title + ";" + itemCount + "\n";
    for (ProductEntry entry : productList)
    {
        Product p = entry.product;
        listData += p.name + ";" + p.category + ";" + p.price + ";" + entry.quantity + ";" + entry.isChecked + "\n";
    }

    return listData;
}
```

Figure 86. Overridden *toString()* method inside *ShoppingList.java* class

```
2
TestList;2
Bread;Bread;3.0;2;false
Apple;Apple;1.0;1;true
List2;2
Tea;Beverage;2.0;2;false
Bread;Bread;3.0;2;false
```

Figure 87. Example of the saved data file

Task #27. Add another fragment to the navigation bar

It was planned before to add another one last part to the lower navigation bar. This last part was to be a Camera implementation that would use the camera and save the data inside a database in the phone. To realize this we first had to create a new Activity class where we could complete a part of these actions (Figure 88).

```
public class PictureActivity extends AppCompatActivity
{

    FrameLayout frameLayout;

    @Override
    protected void onCreate(@Nullable Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_pictures);

        loadFragment(new PictureFragment(),  bool: false);
    }

    public void loadFragment(Fragment fragment, Boolean bool)
    {
        FragmentTransaction transaction = getSupportFragmentManager().beginTransaction();
        transaction.replace(R.id.frameLayout, fragment);

        if (bool)
        {
            transaction.addToBackStack(null);
        }
        transaction.commit();
    }

}
```

Figure 88. Activity class for the picture part

After having written down the main activity where the actions will be made we can move on to the more complex part.

Since the application is based on Fragments we will have to realize the fragment part of the class as well. We have done this in the PictureFragment class (Figure 89).

```
public class PictureFragment extends Fragment
{
    private static final int CAMERA_REQUEST = 1888;
    TextView text,text1;
    private static final int MY_CAMERA_PERMISSION_CODE = 100;
    String photo;
    DBHelper databaseHandler;
    private SQLiteDatabase db;
    Bitmap theImage;

    @Nullable
    @Override
    public View onCreateView(@NonNull LayoutInflater inflater, @Nullable ViewGroup container, @Nullable Bundle savedInstanceState)
    {
        View view = inflater.inflate(R.layout.fragment_pictures,container, attachToRoot: false);

        text = view.findViewById(R.id.saveImageText);
        text1 = view.findViewById(R.id.ViewImageText);

        databaseHandler = new DBHelper(getContext());
        text.setOnClickListener(new View.OnClickListener(){
            @RequiresApi(api = Build.VERSION_CODES.M)
            @Override
            public void onClick(View v)
            {
                if (getActivity().checkSelfPermission(Manifest.permission.CAMERA) != PackageManager.PERMISSION_GRANTED)
                {
                    requestPermissions(new String[]{Manifest.permission.CAMERA}, MY_CAMERA_PERMISSION_CODE);
                }
                else
                {
                    Intent cameraIntent = new Intent(MediaStore.ACTION_IMAGE_CAPTURE);
                    startActivityForResult(cameraIntent, CAMERA_REQUEST);
                }
            }
        });
        text1.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v)
            {
                ((PictureActivity) getActivity()).loadFragment(new PictureStoreDatabaseFragment(),  book: true);
            }
        });
```

Figure 89. PictureFragment class realization

With the help of this class we will be able to get the permissions to use the camera, which are required, encode a String into a Bitmap and set data into a database.

Task #28. Add a button to display all pictures saved

Now that we have more essentials we can move on to more useful classes. With this other Fragment we will be able to separately display all the saved photos on the special screen. After tapping the button View Image the new Fragment class initializes and gets the pictures (Figure 90).

```java
public class PictureStoreDatabaseFragment extends Fragment
{
    RecyclerView recyclerView;
    private DBHelper myDatabase;
    private SQLiteDatabase db;
    private ArrayList singleRowArrayList;
    private Picture singleRow;
    String image;
    int uid;
    Cursor cursor;

    @Nullable
    @Override
    public View onCreateView(@NonNull LayoutInflater inflater, @Nullable ViewGroup container, @Nullable Bundle savedInstanceState) {
        View view = inflater.inflate(R.layout.stored_pictures,container, attachToRoot: false);
        recyclerView = view.findViewById(R.id.recyclerview);
        myDatabase = new DBHelper(getContext());
        db = myDatabase.getWritableDatabase();
        setData();
        return view;
    }
    private void setData() {
        db = myDatabase.getWritableDatabase();
        RecyclerView.LayoutManager layoutManager = new LinearLayoutManager(getContext());
        recyclerView.setLayoutManager(layoutManager);
        singleRowArrayList = new ArrayList<>();
        String[] columns = {DBHelper.KEY_ID, DBHelper.KEY_IMG_URL};
        cursor = db.query(DBHelper.TABLE_NAME, columns, selection: null, selectionArgs: null, groupBy: null, having: null, orderBy: null);
        while (cursor.moveToNext()) {
            int index1 = cursor.getColumnIndex(DBHelper.KEY_ID);
            int index2 = cursor.getColumnIndex(DBHelper.KEY_IMG_URL);
            uid = cursor.getInt(index1);
            image = cursor.getString(index2);
            singleRow = new Picture(image,uid);
            singleRowArrayList.add(singleRow);
        }
        if (singleRowArrayList.size()==0){
            recyclerView.setVisibility(View.GONE);
        }else {
            DBAdapter localDataBaseResponse = new DBAdapter(getContext(), singleRowArrayList, db, myDatabase);
            recyclerView.setAdapter(localDataBaseResponse);
        }
    }
```

Figure 90. Picture viewing Fragment code snippet

This Fragment uses the database so we have to create a helper class for making sure the database is being used correctly. We did this with the DBAdapter class, it will assist us with displaying and getting database elements, as well as deleting other elements. Now the picture that gets taken is displayed on the screen immediately so that the user can be sure it was added successfully (Figure 91).

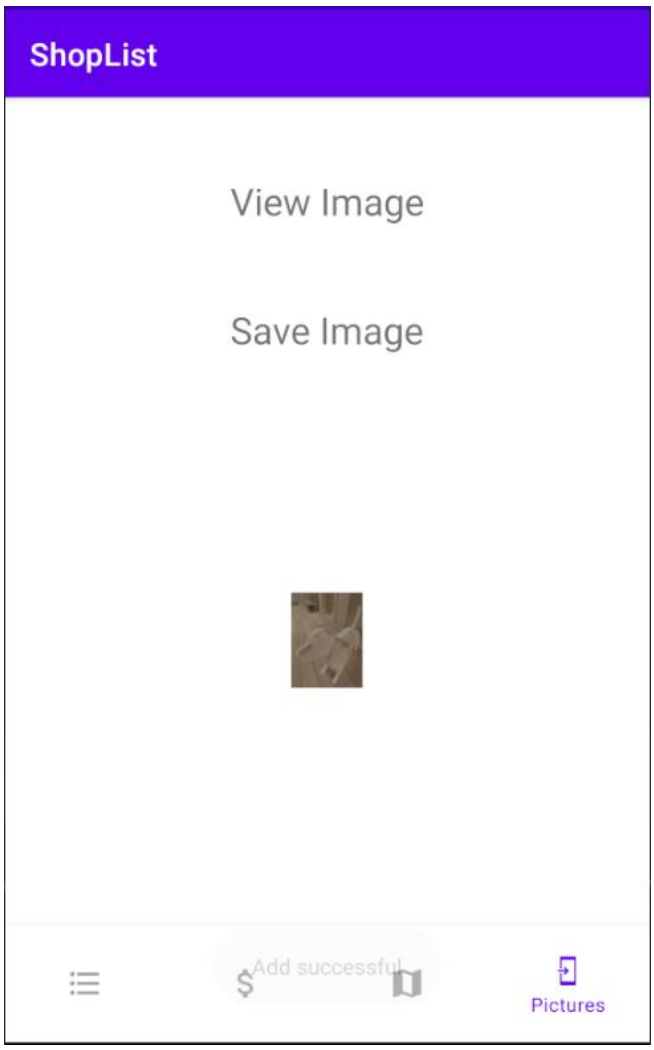


Figure 91. Saved picture displayed.

Task #29. Add database functionality to pictures and save them in the database

With the help of the DBAdapter class we were able to add a lot of functionality to the application making it more versatile. Though, we did run into troubles as we did want our own type of RecyclerView to be called so that it would display the pictures we had to rewrite the class inside (Figure 92).

```
class MyViewHolder extends RecyclerView.ViewHolder {
    ImageView newsImage, delete;

    public MyViewHolder(@NonNull View itemView) {
        super(itemView);
        newsImage = (ImageView) itemView.findViewById(R.id.newImage);
        delete = (ImageView) itemView.findViewById(R.id.deleteImage);
    }
}
```

Figure 92. Our own ViewHolder class with special parameters.

With this special class and a bunch of other functions we were able to realize the DBAdapter class which utilizes the database to work with the data inside it. We did run into trouble with override methods, but this wasn‘t an issue after we moved one of them to a different interface entirely (Figure 93).

```
public class DBAdapter extends RecyclerView.Adapter implements DBAdapterExtra {
    Context context;
    ArrayList singleRowArrayList;
    SQLiteDatabase db;
    DBHelper myDatabase;

    public DBAdapter(Context context, ArrayList singleRowArrayList, SQLiteDatabase db, DBHelper myDatabase) {
        this.context = context;
        this.singleRowArrayList = singleRowArrayList;
        this.db = db;
        this.myDatabase = myDatabase;
    }

    @NonNull
    @Override
    public MyViewHolder onCreateViewHolder(@NonNull ViewGroup viewGroup, int i) {
        View view = LayoutInflater.from(context).inflate(R.layout.layout_pictures,  root: null);
        MyViewHolder myViewHolder = new MyViewHolder(view);
        return myViewHolder;
    }

    @Override
    public void onBindViewHolder(@NonNull RecyclerView.ViewHolder holder, int position)
    {
        MyViewHolder myViewHolder = new MyViewHolder(holder.itemView);
        myViewHolder.newsImage.setImageBitmap(getBitmapFromEncodedString(singleRowArrayList.get(position).toString()));
        myViewHolder.delete.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v)
            {
                deletedata(myViewHolder.getAdapterPosition(), singleRowArrayList);
            }
        });

    }
```

Figure 93. Code snippet from DBAdapter class and it‘s methods

One of the most important things this class does is get the Bitmap of an Encoded String. This function turns the String of chars into a picture for us to view on the phone (Figure 94).

```
private Bitmap getBitmapFromEncodedString(String encodedString) {

    byte[] arr = Base64.decode(encodedString, Base64.URL_SAFE);

    Bitmap img = BitmapFactory.decodeByteArray(arr,  offset: 0, arr.length);

    return img;
}
```

Figure 94. Function to convert String to Bitmap

Defense Task #4. Display the full size image

For this task, before creating the camera intent, we add some information to *ContentValues* (Figure 95). This lets us save the image to external storage after taking the picture. Later, we load the image from external storage (and not the thumbnail anymore) and display it below the *SaveImage* button (Figures 96 and 97).

```
text.setOnClickListener(new View.OnClickListener(){
    @RequiresApi(api = Build.VERSION_CODES.M)
    @Override
    public void onClick(View v)
    {
        if (getActivity().checkSelfPermission(Manifest.permission.CAMERA) != PackageManager.PERMISSION_GRANTED)
        {
            requestPermissions(new String[]{Manifest.permission.CAMERA}, MY_CAMERA_PERMISSION_CODE);
        }
        else
        {
            values = new ContentValues();
            values.put(MediaStore.Images.Media.TITLE, "New Picture");
            values.put(MediaStore.Images.Media.DESCRIPTION, "From your Camera");
            imageUri = getContext().getContentResolver().insert(
                    MediaStore.Images.Media.EXTERNAL_CONTENT_URI, values);
            Intent intent = new Intent(MediaStore.ACTION_IMAGE_CAPTURE);
            intent.putExtra(MediaStore.EXTRA_OUTPUT, imageUri);
            startActivityForResult(intent, CAMERA_REQUEST);

        }
    }
});
```

Figure 95. Metadata of *ContentValues*

```
@Override
public void onActivityResult(int requestCode, int resultCode, @Nullable Intent data) {
    if (requestCode == CAMERA_REQUEST && resultCode == Activity.RESULT_OK)
    {
        try {
            theImage = (Bitmap) MediaStore.Images.Media.getBitmap(getContext().getContentResolver(), imageUri);
        } catch (IOException e) {
            e.printStackTrace();
        }

        iv.setImageBitmap(theImage);
        photo=getEncodedString(theImage);
        setDataToDataBase();

    }
}
```

Figure 96. Loading the full size picture from external storage

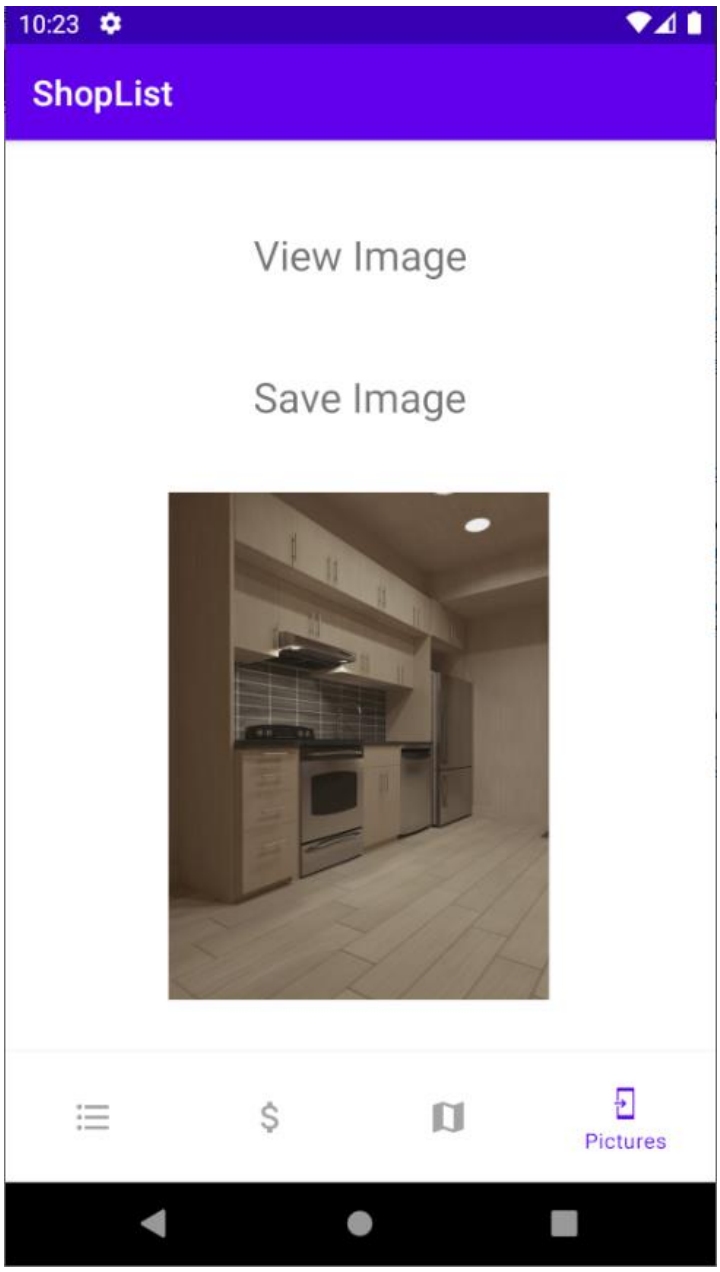


Figure 97. Example of higher quality picture

Reference list

1. Philipp Lackner. *Android Fundamentals for Beginners*. 2020 [žiūrėta 2021 m. spalio 7 d.]. Prieiga per: https://www.youtube.com/playlist?list=PLQkwcJG4YTCTq1raTb5iMuxnEB06J1VHX
2. Google Developers. *Create a Notification.* 2021 [žiūrėta 2021 m. spalio 24 d.]. Prieiga per: https://developer.android.com/training/notify-user/build-notfication.
3. Google Developers. *Navigate between fragments using animations*. 2021 [žiūrėta 2021 m. Lapkričio 6 d.]. Prieiga per: https://developer.android.com/guide/fragments/animate.
4. Shad Sluiter. *SQLite Database for Android – Full Course.* 2020 [žiūrėta 2021 m. lapkričio 6d.] Prieiga per: https://www.youtube.com/watch?v=312RhjfetP8
5. W3 Schools. *JSON Syntax*. 1999–2021 [žiūrėta 2021 m. lapkričio 19d.]. Prieiga per: https://www.w3schools.com/js/js_json_syntax.asp
6. Bean Jordan. *How to Use the Google Places API for Location Analysis and More*. 2021 [žiūrėta 2021 m. gruodžio 3d.]. Prieiga per: https://towardsdatascience.com/how-to-use-the-google-places-api-for-location-analysis-and-more-17e48f8f25b1
7. Google Developers. *Markers*. 2021 [žiūrėta 2021 m. gruodžio 4 d.]. Prieiga per: https://developers.google.com/maps/documentation/android-sdk/marker
8. GeeksforGeeks. *How to get user location in Android*. 2021 [žiūrėta 2021 m. gruodžio 4 d.]. Prieiga per: https://www.geeksforgeeks.org/how-to-get-user-location-in-android/
9. Sandip Bhattacharya. *Read/Write Text File from/to Android External Storage*. 2020 [žiūrėta 2021 m. gruodžio 12 d.]. Prieiga per: https://www.youtube.com/watch?v=JUlZYddw03o
10. AbhiAndroid. *Camera Tutorial With Example In Android Studio [Step by Step]* [žiūrėta 2021 m. gruodžio 12 d.]. Prieiga per: https://abhiandroid.com/programming/camera#1_Using_Camera_By_Using_Camera_Application