LAB 8 – Android Room Database and Repository

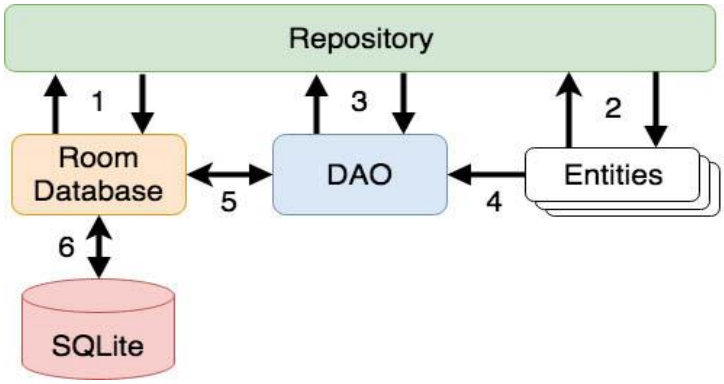
At the end of this lab, students should be able to:

- Make use of the Android TableLayout and TableRow views to design the user interface layout for the Room database example application.
- Demonstrate how to implement SQLite-based database storage using the Room persistence library.
- Make use of all of the elements covered in the Android Room

The TableLayout and TableRow Layout Views

Thee purpose of the TableLayout container view is to allow user interface elements to be organized on the screen in a table format consisting of rows and columns. Each row within a TableLayout is occupied by a TableRow instance which, in turn, is divided into cells, with each cell containing a single child view (which may itself be a container with multiple view children).

Key Elements of Room Database Persistence



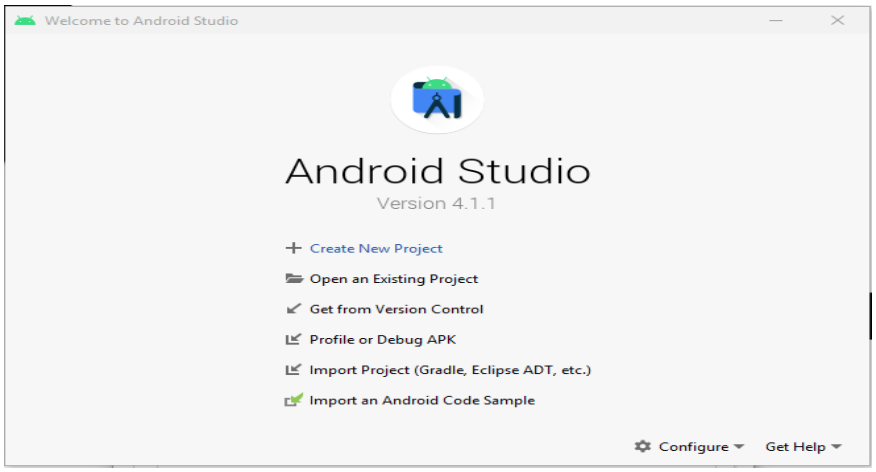
This architecture diagram illustrates the way in which these different elements interact to provide Room-based database storage within an Android app. The numbers mean:

1. The repository interacts with the Room Database to get a database instance which, in turn, is used to obtain references to DAO instances.
2. The repository creates entity instances and configures them with data before passing them to the DAO for use in search and insertion operations.
3. The repository calls methods on the DAO passing through entities to be inserted into the database and receives entity instances back in response to search queries.
4. When a DAO has results to return to the repository it packages those results into entity objects.
5. The DAO interacts with the Room Database to initiate database operations and handle results.
6. The Room Database handles all of the low level interactions with the underlying SQLite database, submitting queries and receiving results.

Android Room Database Project- An Example

a) Creating the Room Database Project

i. Go to File -> New Project



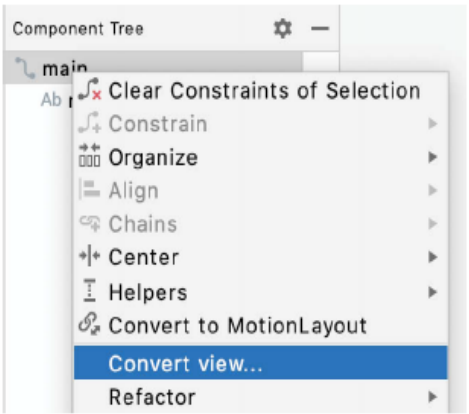
ii. Choose **Fragment + ViewModel** template, then click Next

iii. Enter *RoomDemo* into the Name field and specify com.ebookfrenzy.roomdemo as the package name.
Change the Minimum API level setting to API26: Android 8.0(Oreo) and the Language menu to Java.
Click Finish.

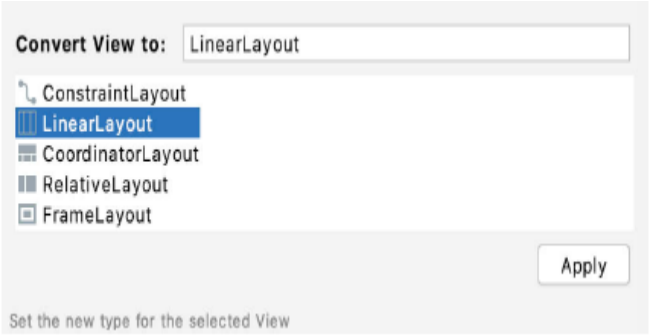
b) Converting to a LinearLayout

Locate the *main_fragment.xml* file in the Project tool window and double click on it to load it into the layout editor. By default, Android Studio has used a ConstraintLayout as the root layout element in the user interface.

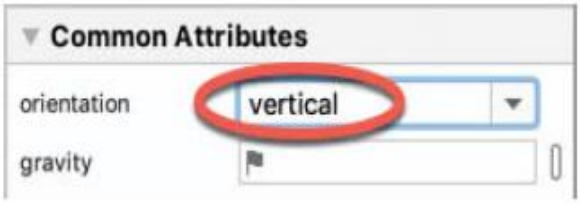
i. This needs to be converted to a vertically oriented LinearLayout. With the Layout Editor tool in Design mode, locate the *main* ConstraintLayout component in the Component tree and right-click on it to display the menu shown in figure below and select the *Convert View...*option:



ii. In the resulting dialog in the below figure select the option to convert to a LinearLayout before clicking on the Apply button.



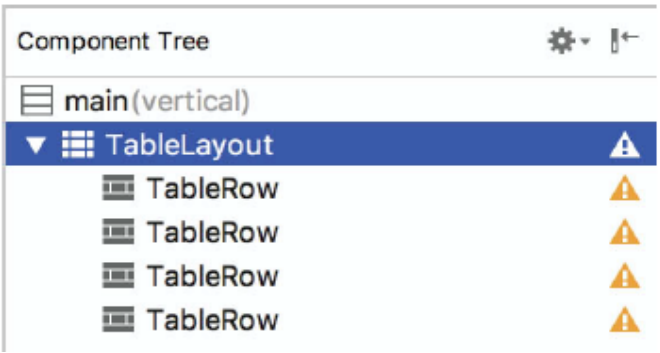
iii. By default, the layout editor will have converted the ConstraintLayout to a horizontal LinearLayout so select the layout component in the Component Tree window, refer to the Attributes tool window and change the orientation property to *vertical*:



iv. With the conversion complete, select and delete the default TextView widget from the layout.

c) Adding the TableLayout to the User Interface

i. Remaining in the main_fragment.xml file and referring to the Layouts category of the Palette, drag and drop a TableLayout view so that it is positioned at the top of the LinearLayout canvas area. Once these initial steps are complete, the Component Tree for the layout should resemble that shown in figure below:

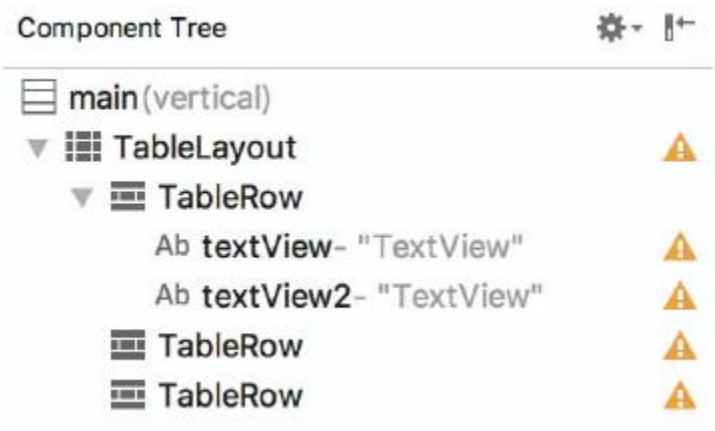


ii. Clearly, Android Studio has automatically added four TableRow instances to the TableLayout. Since only three rows are required for this example, select and delete the fourth TableRow instance.

iii. With the TableLayout selected, use the Attributes tool window to change the layout_height property to *wrap_content* and layout_width to *match_parent*.

d) Configuring the TableRows

i. From within the *Text* section of the palette, drag and drop two TextView objects onto the uppermost TableRow entry in the Component Tree.

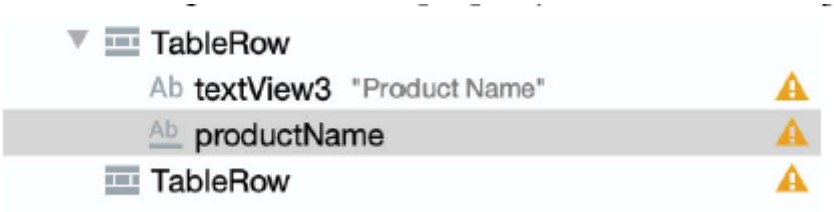


ii. Select the left most TextView within the screen layout and, in the Attributes tool window, change the text property to “Product ID”. Repeat this step for the right most TextView, this time changing the text to “Not assigned” and specifying an *ID* value of *productID*.

iii. Drag and drop another TextView widget onto the second TableRow entry in the Component Tree and change the text on the view to read “Product Name”.

iv. Locate the Plain Text object in the palette and drag and drop it so that it is positioned beneath the Product Name TextView within the Component Tree as outlined in figure below.

v. With the EditText object selected, change the inputType property from textPersonName to none, delete the “Name” string from the text property and set the ID to productName.



vi. Drag and drop another TextView and a Number (Decimal) Text Field onto the third TableRow so that the TextView is positioned above the EditText in the hierarchy.

vii. Change the text on the TextView to Product Quantity and the ID of the EditText object *to productQuantity*.

viii. Shift click to select all of the widgets in the layout as shown in figure below, and use the Attributes tool window to set the textSize property on all of the objects to 18sp:

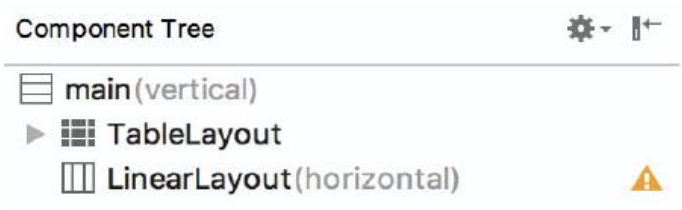
ix. Before proceeding, be sure to extract all of the text properties added in the above steps to string resources.

e) Adding the Button Bar to the Layout

The next step is to add a LinearLayout (Horizontal) view to the parent LinearLayout view, positioned immediately below the TableLayout view.

i. Begin by clicking on the small disclosure arrow to the left of the TableLayout entry in the Component Tree so that the TableRows are folded away from view.

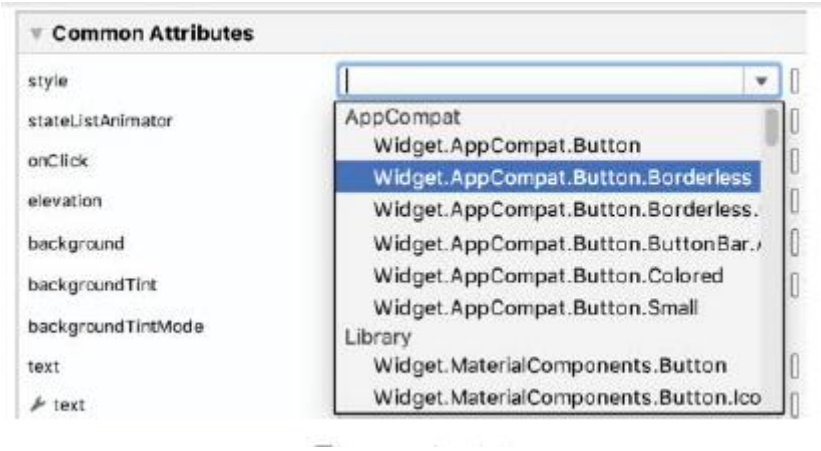
ii. Drag a *LinearLayout (horizontal)* instance from the *Layouts* section of the Layout Editor palette, drop it immediately beneath the TableLayout entry in the Component Tree panel and change the layout_height property to *wrap_content*:



iii. Drag and drop three Button objects onto the new LinearLayout and assign string resources for each button that read “Add”, “Find” and “Delete” respectively. Buttons in this type of button bar arrangement should generally be displayed with a borderless style.

iv. For each button, use the Attributes tool window to change the *style* setting to *Widget.AppCompat.Button.Borderless* and the *textColor* attribute to *@color/black.*

v. Change the IDs for the buttons to *addButton, findButton* and *deleteButton* respectively.



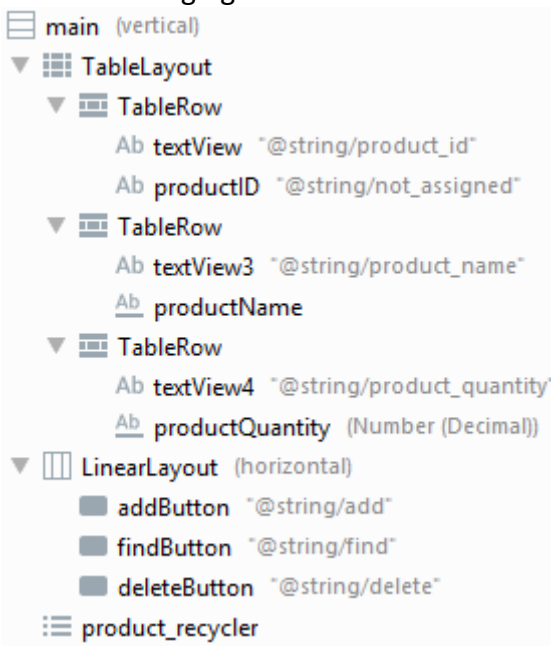
vi. With the new horizontal LinearLayout view selected in the Component Tree change the gravity property to *center_horizontal* so that the buttons are centered horizontally within the display.

f) Adding the RecyclerView

i. In the Component Tree, click on the disclosure arrow to the right of the newly added horizontal LinearLayout entry to fold all of the children from view.

ii. From the Containers section of the Palette, drag a RecyclerView instance and drop it onto the Component Tree so that it positioned beneath the button bar LinearLayout as shown in the next figure. Take care to ensure the RecyclerView is added as a direct child of the parent vertical LinearLayout view and not as a child of the horizontal button bar LinearLayout.

iii. With the RecyclerView selected in the layout, change the ID of the view to *product_recycler* and set the layout_height property to *match_parent*. Before proceeding, check that the hierarchy of the layout in the Component Tree panel matches that shown in the following figure:



g) Adjusting the Layout Margins

i. Begin by clicking on the first TableRow entry in the Component Tree panel so that it is selected. Hold down the Cmd/Ctrl-key on the keyboard and click on the second and third TableRows, the horizontal LinearLayout and the RecyclerView so that all five items are selected. In the Attributes panel, locate the *layout_margin* attributes category and, once located, change the value to 10dp as shown in figure below:

ii. With margins set, the user interface should appear as illustrated in the following figure.

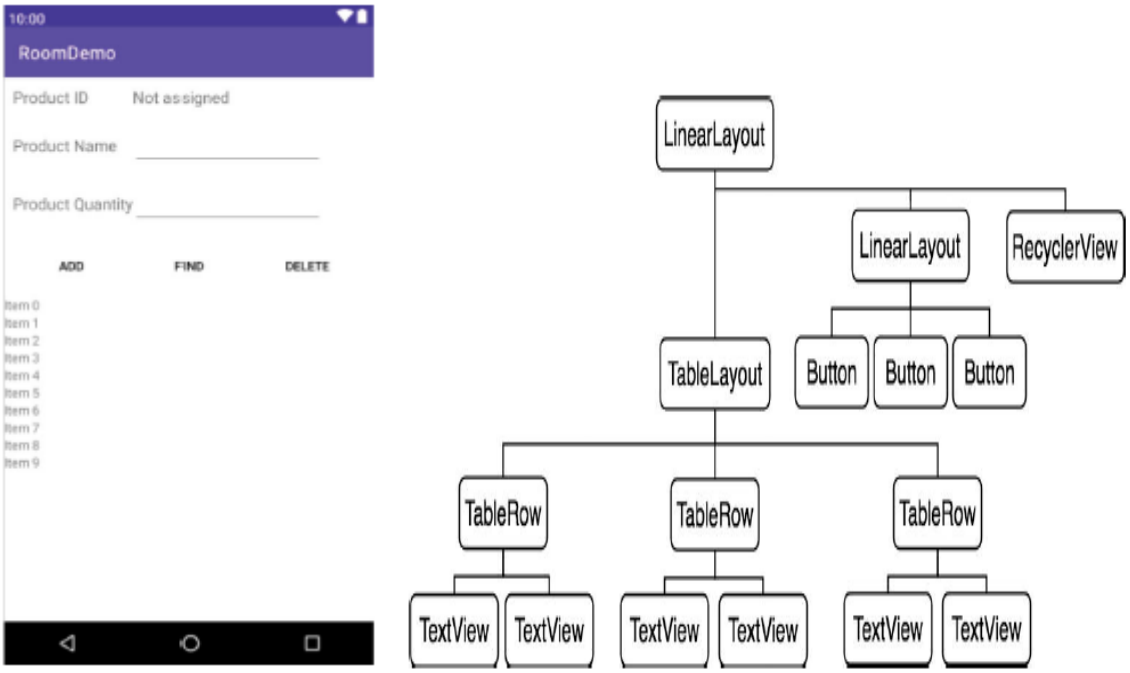


Figure: Hierarchy view of the layout

The user interface layout created was the first step in creating a rudimentary inventory app designed to store the names and quantities of products.
When completed, the app will provide the ability to add, delete and search for database entries while also displaying a scrollable list of all products currently stored in the database. This product list will update automatically as database entries are added or deleted.

h) Modifying the Build Configuration

i. Before adding any new classes to the project, the first step is to add some additional libraries to the build configuration, including the Room persistence library. Locate and edit the module level *build.gradle* file (*app -> Gradle Scripts ->build.gradle (Module: RoomDemo.app*)) and modify it as follows:

```
dependencies {
.
.
implementation "androidx.recyclerview:recyclerview:1.1.0"
implementation "androidx.room:room-runtime:2.2.5"
implementation "androidx.fragment:fragment-ktx:1.2.5"
annotationProcessor "androidx.room:room-compiler:2.2.5"
.
.
}
```

i) Building the Entity

Begin by creating the entity which defines the schema for the database table. The entity will consist of an integer for the product id, a string column to hold the product name and another integer value to store the quantity. The product id column will serve as the primary key and will be auto-generated. Table below summarizes the structure of the entity:

Column	Data Type
productid	Integer / Primary Key / Auto Increment
productname	String
productquantity	Integer

i. Add a class file for the entity by right clicking on the *app -> java ->com.ebookfrenzy.roomdemo* entry in the Project tool window and selecting the *New -> Java Class* menu option. In the new class dialog, name the class *Product* and press Enter to generate the file. When the *Product.java* file opens in the editor, modify it so that it reads as follows:

```
public class Product {
private int id;
private String name;
private int quantity;
public Product(String name, int quantity) {
this.id = id;
this.name = name;
this.quantity = quantity;
}
public int getId() {
return this.id;
}
public String getName() {
return this.name;
}
public int getQuantity() {
return this.quantity;
}
public void setId(int id) {
this.id = id;
}
public void setName(String name) {
this.name = name;
}
public void setQuantity(int quantity) {
this.quantity = quantity;
}
}
```

ii. The class now has variables for the database table columns and matching getter and setter methods. Of course this class does not become an entity until it has been annotated. With the class file still open in the editor, add annotations and corresponding import statements:

```
package com.ebookfrenzy.roomdemo;
import androidx.annotation.NonNull;
import androidx.room.ColumnInfo;
import androidx.room.Entity;
import androidx.room.PrimaryKey;
@Entity(tableName = "products")
public class Product {
@PrimaryKey(autoGenerate = true)
@ColumnInfo(name = "productId")
private int id;
@ColumnInfo(name = "productName")
private String name;
private int quantity;
.
.
}
```

These annotations declare this as the entity for a table named products and assigns column names for both the *id* and *name* variables. The id column is also configured to be the primary key and auto-generated. Since a primary key can never be null, the @NonNull annotation is also applied. Since it will not be necessary to reference the quantity column in SQL queries, a column name has not been assigned to the *quantity* variable.

j) Creating the Data Access Object

i. With the product entity defined, the next step is to create the DAO interface. Referring once again to the Project tool window, right-click on the *app -> java ->com.ebookfrenzy.roomdemo* entry and select the New -> Java Class menu option. In the new class dialog, enter *ProductDao* into the Name field and select *Interface* from the list as highlighted in figure below:

ii. Press Enter to generate the new interface and, with the *ProductDao.java* file loaded into the code editor, make the following changes:

```
package com.ebookfrenzy.roomdemo;
import androidx.lifecycle.LiveData;
import androidx.room.Dao;
import androidx.room.Insert;
import androidx.room.Query;
import java.util.List;
@Dao
```

```
public interface ProductDao {
   @Insert
   void insertProduct(Product product);
   @Query("SELECT * FROM products WHERE productName = :name")
   List<Product> findProduct(String name);
   @Query("DELETE FROM products WHERE productName = :name")
   void deleteProduct(String name);
   @Query("SELECT * FROM products")
   LiveData<List<Product>> getAllProducts();
}
```

The DAO implements methods to insert, find and delete records from the products database. The insertion method is passed a Product entity object containing the data to be stored while the methods to find and delete records are passed a string containing the name of the product on which to perform the operation. The *getAllProducts()* method returns a LiveData object containing all of the records within the database. This method will be used to keep the RecyclerView product list in the user interface layout synchronized with the database.

k) Adding the Room Database

The last task before adding the repository to the project is to implement the Room Database instance. Add a new class to the project named *ProductRoomDatabase*, this time with the Class option selected. Once the file has been generated, modify it as follows:

```
package com.ebookfrenzy.roomdemo;
import android.content.Context;
import androidx.room.Database;
import androidx.room.Room;
import androidx.room.RoomDatabase;
@Database(entities = {Product.class}, version = 1)
public abstract class ProductRoomDatabase extends RoomDatabase {
public abstract ProductDao productDao();
private static ProductRoomDatabase INSTANCE;
static ProductRoomDatabase getDatabase(final Context context)
{
if (INSTANCE == null) {
synchronized (ProductRoomDatabase.class) {
if (INSTANCE == null) {
INSTANCE =
Room.databaseBuilder(context.getApplicationContext(),
ProductRoomDatabase.class,
"product_database").build
();
}
}
}
return INSTANCE;
} }
```

l) Adding the Repository

i. Add a new class named *ProductRepository* to the project, with the *Class* option selected. The repository class will be responsible for interacting with the Room database on behalf of the ViewModel and will need to provide methods that use the DAO to insert, delete and query product records. With the exception of the *getAllProducts()* DAO method (which returns a LiveData object) these database operations will need to be performed on separate threads from the main thread.

ii. Remaining within the *ProductRepository.java* file, add the code for a handler to return the search results to the repository thread:

```
package com.ebookfrenzy.roomdemo;
import android.os.Handler;
import android.os.Looper;
import android.os.Message;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import androidx.lifecycle.MutableLiveData;
import java.util.List;
public class ProductRepository {
private final MutableLiveData<List<Product>> searchResults =
new MutableLiveData<>();
private List<Product> results;
Handler handler = new Handler(Looper.getMainLooper()) {
@Override public void handleMessage(Message msg) {
searchResults.setValue(results);
}
};
}
```

The above declares a MutableLiveData variable named *searchResults* into which the results of a search operation are stored whenever an asynchronous search task completes (later in the tutorial, an observer within the ViewModel will monitor this live data object).

The repository class now needs to provide some methods that can be called by the ViewModel to initiate these operations.

In order to be able to do this, however, the repository needs to obtain the DAO reference via a ProductRoomDatabase instance.

iii. Add a constructor method to the ProductRepository class to perform these tasks:

```
.
.
import android.app.Application;
.
.
public class ProductRepository {
private final MutableLiveData<List<Product>> searchResults =
new MutableLiveData<>();
private List<Product> results;
private final ProductDao productDao;
public ProductRepository(Application application) {
ProductRoomDatabase db;
db = ProductRoomDatabase.getDatabase(application);
productDao = db.productDao();
}
.
.
```

iv. With a reference to DAO stored, the methods are ready to be added to the class file:

```
public void insertProduct(Product newproduct) {
ExecutorService executor = Executors.newSingleThreadExecutor();
executor.submit(() -> {
productDao.insertProduct(newproduct);
});
executor.shutdown();
}
```

```
        public void deleteProduct(String name) {
        ExecutorService executor = Executors.newSingleThreadExecutor();
        executor.submit(() -> {
        productDao.deleteProduct(name);
        });
        executor.shutdown();
        }
        public void findProduct(String name) {
        ExecutorService executor = Executors.newSingleThreadExecutor();
        executor.submit(() -> {
        results = productDao.findProduct(name);
        handler.sendEmptyMessage(0);
        });
        executor.shutdown();
}
```

In the cases of the insertion and deletion methods, the appropriate new threads are created and used to perform the corresponding database operation. In the case of the *findProduct()* method, a message is sent to the handler indicating that new results are available.

One final task remains to complete the repository class. The RecyclerView in the user interface layout will need to be able to keep up to date the current list of products stored in the database. The ProductDao class already includes a method named *getAllProducts()* which uses a SQL query to select all of the database records and return them wrapped in a LiveData object. The repository needs to call this method once on initialization and store the result within a LiveData object that can be observed by the ViewModel and, in turn, by the UI controller. Once this has been set up, each time a change occurs to the database table the UI controller observer will be notified and the RecyclerView can be updated with the latest product list.

v. Remaining within the *ProductRepository.java* file, add a LiveData variable and call to the DAO *getAllProducts()* method within the constructor:

```
.
.
import androidx.lifecycle.LiveData;
..
public class ProductRepository {
private final MutableLiveData<List<Product>> searchResults =
new MutableLiveData<>();
private List<Product> results;
private final LiveData<List<Product>> allProducts;
private final ProductDao productDao;
public ProductRepository(Application application) {
ProductRoomDatabase db;
db = ProductRoomDatabase.getDatabase(application);
productDao = db.productDao();
allProducts = productDao.getAllProducts();
}
.
.}
```

vi. To complete the repository, add methods that the ViewModel can call to obtain references to the *allProducts* and *searchResults* live data objects:

```
public LiveData<List<Product>> getAllProducts() {
return allProducts;
}
public MutableLiveData<List<Product>> getSearchResults() {
return searchResults;
}
```

m) Modifying the ViewModel

The ViewModel is responsible for creating an instance of the repository and for providing methods and LiveData objects that can be utilized by the UI controller to keep the user interface synchronized with the underlying database.
As implemented in *ProductRepository.java*, the repository constructor requires access to the application context in order to be able to get a Room Database instance. To make the application context accessible within the ViewModel so that it can be passed to the repository, the ViewModel needs to subclassAndroidViewModel instead of ViewModel.

i. Begin, therefore, by editing the *MainViewModel.java* file (located in the Project tool window under *app -> java -> com.ebookfrenzy.roomdemo -> ui.main*) and changing the class to extend AndroidViewModel and to implement the default constructor:

```
package com.ebookfrenzy.roomdemo.ui.main;
import android.app.Application;
import androidx.lifecycle.AndroidViewModel;
import androidx.lifecycle.LiveData;
import androidx.lifecycle.MutableLiveData;
import com.ebookfrenzy.roomdemo.Product;
import com.ebookfrenzy.roomdemo.ProductRepository;
import java.util.List;
import androidx.lifecycle.ViewModel;
public class MainViewModel extends AndroidViewModel
{
private ProductRepository repository;
private LiveData<List<Product>> allProducts;
private MutableLiveData<List<Product>> searchResults;
public MainViewModel (Application application) {
super(application);
repository = new ProductRepository(application);
allProducts = repository.getAllProducts();
searchResults = repository.getSearchResults();
}
}
```

The constructor essentially creates a repository instance and then uses it to get references to the results and live data objects so that they can be observed by the UI controller.

ii. All that now remains within the ViewModel is to implement the methods that will be called from within the UI controller in response to button clicks and when setting up observers on the LiveData objects:

```
MutableLiveData<List<Product>> getSearchResults() {
return searchResults;
}
LiveData<List<Product>> getAllProducts() {
return allProducts;
}
public void insertProduct(Product product) {
repository.insertProduct(product);
}
public void findProduct(String name) {
repository.findProduct(name);
}
public void deleteProduct(String name) {
repository.deleteProduct(name);
}
```

n) Creating the Product Item Layout

The name of each product in the database will appear within the RecyclerView list in the main user interface. This will require a layout resource file containing a TextView to be used for each row in the list.

i. Add this file now by right-clicking on the *app -> res -> layout* entry in the Project tool window and selecting the *New -> Layout resource file* menu option. Name the file *product_list_item* and change the root element to LinearLayout before clicking on OK to create the file and load it into the layout editor. With the layout editor in Design mode, drag a TextView object from the palette onto the layout where it will appear by default at the top of the layout:

ii. With the TextView selected in the layout, use the Attributes tool window to set the ID of the view to *product_row* and the layout_height to 30dp. Select the LinearLayout entry in the Component Tree window and set the layout_height attribute to *wrap_content*.

o) Adding the RecyclerView Adapter

A RecyclerView instance requires an adapter class to provide the data to be displayed.

i. Add this class now by right clicking on the *app -> java -> com.ebookfrenzy.roomdemo -> ui.main* entry in the Project tool window and selecting the *New -> Java Class...* menu. In the dialog, name the class *ProductListAdapter*. With the resulting *ProductListAdapter.java* class loaded into the editor, implement the class as follows:

```
package com.ebookfrenzy.roomdemo.ui.main;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.TextView;
import com.ebookfrenzy.roomdemo.R;
import androidx.recyclerview.widget.RecyclerView;
import androidx.annotation.NonNull;
import com.ebookfrenzy.roomdemo.Product;
import java.util.List;
public class ProductListAdapter
extends RecyclerView.Adapter<ProductListAdapter.ViewHolder> {
private final int productItemLayout;
private List<Product> productList;
public ProductListAdapter(int layoutId) {
productItemLayout = layoutId;
}
public void setProductList(List<Product> products) {
productList = products;
notifyDataSetChanged();
}
@Override
public int getItemCount() {
return productList == null ? 0 : productList.size();
 }
```

```
@NonNull
@Override
public ViewHolder onCreateViewHolder(ViewGroup parent, int
viewType) {
View view = LayoutInflater.from(
parent.getContext()).inflate(productItemLayout, parent, false);
return new ViewHolder(view);
}
@Override
public void onBindViewHolder(final ViewHolder holder, final int
listPosition) {
TextView item = holder.item;
item.setText(productList.get(listPosition).getName());
}
static class ViewHolder extends RecyclerView.ViewHolder {
TextView item;
ViewHolder(View itemView) {
super(itemView);
item = itemView.findViewById(R.id.product_row);
}
}
}
```

p) Preparing the Main Fragment

The last remaining component to modify is the MainFragment class which needs to configure listeners on the Button views and observers on the live data objects located in the ViewModel class. Before adding this code, some preparation work needs to be performed to add some imports, variables and to obtain references to view ids. Edit the *MainFragment.java* file and modify it as follows:

```
.
.
import android.widget.EditText;
import android.widget.TextView;
.
.
public class MainFragment extends Fragment {
private MainViewModel mViewModel;
private ProductListAdapter adapter;
private TextView productId;
private EditText productName;
private EditText productQuantity;
..
@Override
public void onActivityCreated(@Nullable Bundle savedInstanceState) {
super.onActivityCreated(savedInstanceState);
mViewModel = new ViewModelProvider(this).get(MainViewModel.class);
productId = getView().findViewById(R.id.productID);
productName = getView().findViewById(R.id.productName);
productQuantity = getView().findViewById(R.id.productQuantity);
listenerSetup();
observerSetup();
recyclerSetup();
}
.
.
}
```

At various stages in the code, the app will need to clear the product information displayed in the user interface. To avoid code repetition, add the following *clearFields()* convenience function:

```
private void clearFields() {
productId.setText("");
productName.setText("");
productQuantity.setText("");
}
```

Before the app can be built and tested, the three setup methods called from the *onActivityCreated()* method above need to be added to the class.

q) Adding the Button Listener

The user interface layout for the main fragment contains three buttons each of which needs to perform a specific task when clicked by user. Edit the *MainFragment.java* file and add the *listenerSetup()* method:

```
..
import android.widget.Button;
import com.ebookfrenzy.roomdemo.Product;
..
private void listenerSetup() {
Button addButton = getView().findViewById(R.id.addButton);
Button findButton = getView().findViewById(R.id.findButton);
Button deleteButton = getView().findViewById(R.id.deleteButton);
addButton.setOnClickListener(new View.OnClickListener() {
@Override
public void onClick(View view) {
String name = productName.getText().toString();
String quantity = productQuantity.getText().toString();
if (!name.equals("") && !quantity.equals("")) {
Product product = new Product(name,
Integer.parseInt(quantity));
mViewModel.insertProduct(product);
clearFields();
} else {
productId.setText("Incomplete information");
}}
});
findButton.setOnClickListener(new View.OnClickListener() {
@Override
public void onClick(View view) {
mViewModel.findProduct(productName.getText().toString());
}
});
deleteButton.setOnClickListener(new View.OnClickListener() {
@Override
public void onClick(View view) {
mViewModel.deleteProduct(productName.getText().toString());
clearFields();
}
});}
..}
```

The addButton listener performs some basic validation to ensure that the user has entered both a product name and quantity and uses this data to create a new Product entity object (note that the quantity string is converted to an integer to match the entity data type). The ViewModel *insertProduct()* method is then called and passed the Product object before the fields are cleared.
The findButton and deleteButton listeners pass the product name to either the ViewModel *findProduct()* or *deleteProduct()* method.

r) Adding LiveData Observers

The user interface now needs to add observers to remain synchronized with the *searchResults* and *allProducts* live data objects within the ViewModel.

i. Remaining in the *Mainfragment.java* file, implement the observer setup method as follows:

```
.
.
import androidx.lifecycle.Observer;
import java.util.List;
import java.util.Locale;
.
.
private void observerSetup() {
mViewModel.getAllProducts().observe(getViewLifecycleOwner(),
new Observer<List<Product>>() {
public void onChanged(@Nullable final List<Product> products) {
adapter.setProductList(products);
}
});
mViewModel.getSearchResults().observe(getViewLifecycleOwner(),
new Observer<List<Product>>() {
@Override
public void onChanged(@Nullable final List<Product> products) {
if (products.size() > 0) {
productId.setText(String.format(Locale.US, "%d",
products.get(0).getId()));
productName.setText(products.get(0).getName());
productQuantity.setText(String.format(Locale.US, "%d",
products.get(0).getQuantity()));
} else {
productId.setText("No Match");
}
}
});
}
.
.
}
```

The “all products” observer simply passes the current list of products to the *setProductList()* method of the RecyclerAdapter where the displayed list will be updated.
The “search results” observer checks that at least one matching result has been located in the database, extracts the first matching Product entity object from the list, gets the data from the object, converts it where necessary and assigns it to the TextView and EditText views in the layout. If the product search failed, the user is notified via a message displayed on the product ID TextView.

s) Initializing The RecyclerView

Add the final setup method to initialize and configure the RecyclerView and adapter as follows:

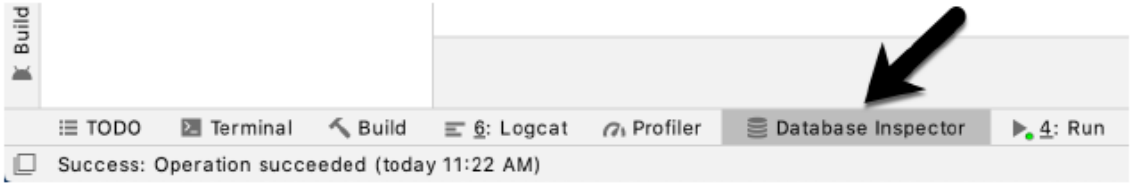
```
.
.
import androidx.recyclerview.widget.LinearLayoutManager;
import androidx.recyclerview.widget.RecyclerView;


..
private void recyclerSetup() {
RecyclerView recyclerView;
adapter = new ProductListAdapter(R.layout.product_list_item);
recyclerView = getView().findViewById(R.id.product_recycler);
recyclerView.setLayoutManager(new LinearLayoutManager(getContext()));
recyclerView.setAdapter(adapter);
}
.
.
}
```

t) Testing the RoomDemo App

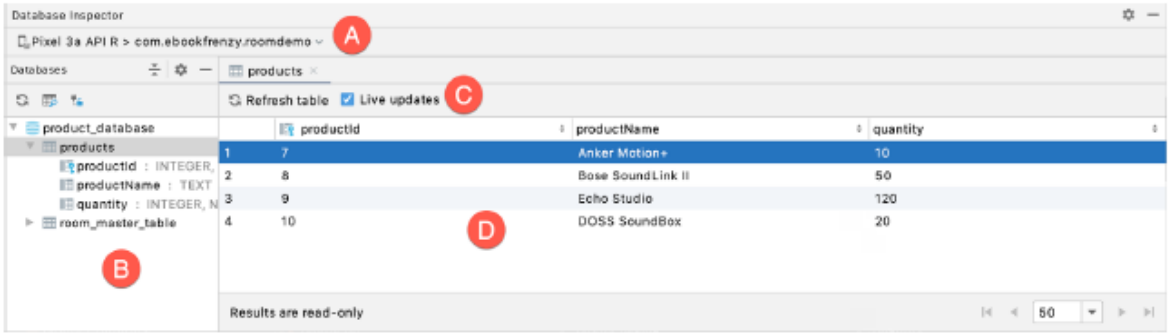
Compile and run the app on a device or emulator, add some products and make sure that they appear automatically in the RecyclerView. Perform a search for an existing product and verify that the product ID and quantity fields update accordingly. Finally, enter the name for an existing product, delete it from the database and confirm that it is removed from the RecyclerView product list.

u) Using The Database Inspector

The Database Inspector tool window may be used to inspect the content of Room databases associated with a running app and to perform minor data changes. After adding some database records using the RoomDemo app, display the Database Inspector tool window using the tab button indicated in figure below:



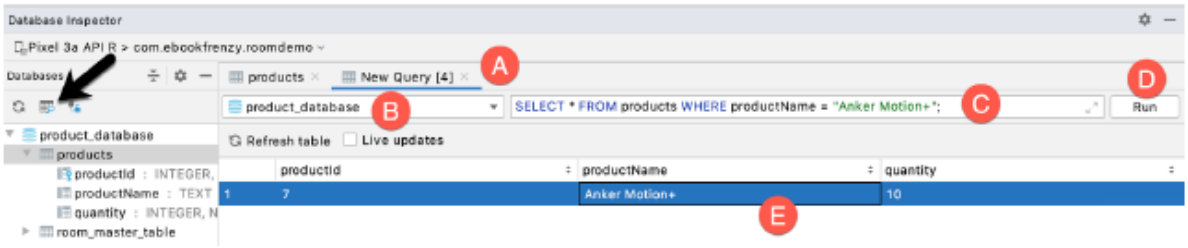
From within the inspector window, select the running app from the menu marked A in the following figure



From the Databases panel (B) double-click on the *products* table to view the table rows currently stored in the database. Enable the *Live updates* option (C) and then use the running app to add more records to the database.

Note that the Database Inspector updates the table data (D) in real-time to reflect the changes. Turn off Live updates so that the table is no longer read only, double-click on the quantity cell for a table row and change the value before pressing the keyboard Enter key. Return to the running app and search for the product to confirm the change made to the quantity in the inspector was saved to the database table.

Finally, click on the table query button (indicated by the arrow in figure below) to display a new query tab (A), make sure that *product_database* is selected (B) and enter a SQL statement into the query text field (C) and click the Run button(D):



The list of rows should update to reflect the results of the SQL query (E).

Exercise:

1. Please add another column for price (you may need to edit the xml file for the interface as well as java file for the coding).
2. Key in a new data including the price.

3. Screenshot an example when you search for the product and upload in MMLS.