Learn Android MVP Pattern By Example

Val Okafor, 9 May 2016 CPOL

Rate this:

★★★★★ 4.69 (11 votes)

In this tutorial, I will share a practical example of Model View Presenter (MVP) pattern in Android development. MVP is a design pattern and when it comes to Android development some of the examples available online are heavy on theories.

In this tutorial, I will share a practical example of Model View Presenter (MVP) pattern in Android development. MVP is a design pattern and when it comes to Android development some of the examples available online are heavy on theories. And in theory Android MVP pattern is simple, it advocates separating business and persistence logic out of the Activity and Fragment. That is a noble goal, the only challenge is that in Android development we do not do MainActivity = new Activity().

The consideration of, and the handling of Activity/Fragment life cycles and other unique Android way of life that we have to deal with makes it hard to adopt generic design patterns into Android development. However with MVP, it has been my experience that even a small attempt in adopting Android MVP pattern provides more flexibility and ease of maintainability over non-pattern based Android development.

This tutorial is a continuation of my tutorial on Dependency Injection with Dagger 2. If you did not read that tutorial, I encourage you to do so as that will make it easy for you to understand the rest of this tutorial.

Android Model View Presenter Pattern Defined

To understand Android MVP pattern we first need to understand the problem it is trying to solve. The goal of MVP in Android is to separate the known from the unknown. Let me explain this with the classic Hello World and the tried and true TextView. If I have an app that all it needs to do is to display Hello World, then I can accomplish this app like this.

Hide Copy Code

```
public class MainActivity extends AppCompatActivity {

    private TextView mTextViewGreeting;

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

        mTextViewGreeting = (TextView) findViewById(R.id.greeting_text_view);
        mTextViewGreeting.setText("Hello World");

    }

}
```

In this scenario, any mention of design pattern or testing will be over engineering. Because TextView is a known quantity, we do not need to test it, it works given the correct input. However if business requirements change, and I am now asked to add some personalization to my greetings, then I can further update my Hello World to greet like this:

Hide Shrink ▲ Copy Code

```
public class MainActivity extends AppCompatActivity {

    private TextView mTextViewGreeting;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar);
        setSupportActionBar(toolbar);

        mTextViewGreeting = (TextView) findViewById(R.id.greeting_text_view);
        //mTextViewGreeting.setText("Hello World");
        mTextViewGreeting.setText(getSalute());

    }

    private String getSalute() {
        int hour = Calendar.getInstance().get(Calendar.HOUR_OF_DAY);


        String greeting = "Hi";
        if (hour > 0 && hour <= 12)
            greeting = "Good Morning";
        else
        if (hour > 12 && hour <= 18)
            greeting = "Good Afternoon";
        else
        if (hour > 18 && hour <= 21)
            greeting = "Good Evening";
        else
        if (hour > 21 && hour <= 24)
            greeting = "Good Night";

        return greeting;
    }
}
```

There are few things wrong with the approach used above:

1. We are making the Activity which already have a hundred and one things to worry about to now also own our getSalute() logic.
2. We cannot test our logic independent of the Activity
3. And if business requirements change, as it often does, we will continue to couple untested, unknown, unproven logic within the Activity.

Enter MVP – Android MVP pattern is a design approach that separates your programming logic from the Activity and the Fragment. With this approach, the Activity or the Fragment is the View and the Java class holding the logic is the Presenter.

Applying Android MVP to the Hello World example, we can create a standard Java class called Presenter, and this class will have a method called setGreetings(String greetings). Then somehow we need to connect this POJO class and the Activity (via Contract) such that the Presenter can ask the Activity to set greeting at the appropriate time. This way the Activity no longer has to know or worry about how the greeting is constructed.

Android MVP By Example

I will use a list of products to demonstrate Android MVP pattern. This list of products is part of the sample Android shopping cart app which I demonstrated in the Dagger 2 tutorial. Here are the steps to display a list of products using MVP.

Step 1 – Create the Contract: The Contract is the glue that binds the participating components in MVP pattern together. It is nothing but the good old Java interface that defines the responsibility of the Model, the View and the Presenter. I like to create these three interfaces within one class file because it makes for better code structure. Here is what the contract for product list may look like.

Hide Shrink ▲ Copy Code

```
public class ProductListContract {
    public interface View {
        void showProducts(List&lt;Product&gt; products);

        void showAddProductForm();

        void showEditProductForm(Product product);

        void showDeleteProductPrompt(Product product);

        void showGoogleSearch(Product product);

        void showEmptyText();

        void hideEmptyText();


        void showMessage(String message);

    }

    public interface Actions {
        void loadProducts();

        void onAddProductButtonClicked();

        void onAddToCartButtonClicked(Product product);

        Product getProduct(long id);

        void addProduct(Product product);

        void onDeleteProductButtonClicked(Product product);

        void deleteProduct(Product product);

        void onEditProductButtonClicked(Product product);

        void updateProduct(Product product);

    }

    public interface Repository {
        List&lt;Product&gt; getAllProducts();

        Product getProductById(long id);

        void deleteProduct(Product product);

        void addProduct(Product product);

        void updateProduct(Product product);

    }
}
```

The above class file has three interfaces within one interface. It is OK to create three separate class file to accomplish the same goal, I prefer the above approach. Let's see what each interface does.

1. **View** – this defines the methods that the concrete View aka Fragment will implement. This way you can proceed to create and test the Presenter without worrying about Android-specific components such as Context.
2. **Actions** – this defines the methods that the concrete Presenter class will implement. Also known as user actions, this is where the business logic for the app is defined.
3. **Repository** – this defines the methods that the concrete persistence class will implement. This way the Presenter does not need to be concerned about how data is persisted.

Step 2 – Create Presenter Class: Create a package called presenter, and in this package add a standard Java class called ProductListPresenter.java

Step 3 – Update Dagger Component: Since the Presenter is a POJO class, the components that we need in this class such as the concrete implementation of the View, Repository, Shopping Cart, etc has to be passed to us. We can either receive these components via the constructor or we can have them be injected using Dagger 2. We will use the later approach.

We, therefore, need to update the AppComponent interface in the dagger package to indicate that the Presenter class is an injectable target like this.

Hide Copy Code

```
public interface AppComponent {
    void inject(ProductListener presenter);
    void inject(MainActivity activity);
    void inject(ProductListPresenter presenter);
}
```

Step 3 – Create Persistence Module: We need to let Dagger 2 know which concrete class will be used to satisfy the dependency on the Repository interface. This concrete class can use either in-memory data, SQLite, Realm Database, etc but for this tutorial, we will just create an empty in-memory class called TempRepositoty.java. I cover the different types of concrete persistence in my course ProntoSQLite.

After we create the TempRepository, we then need to create a Dagger 2 module called Persistence Module and it will look like this.

Hide Copy Code

```
@Module
public class PersistenceModule {
    @Provides
    public ProductListContract.Repository providesProductRepository(Context context){
        return new TempRepository();
    }

}
```

We also need to update the AppComponent to add the newly created module like this:

Hide Copy Code

```
@Singleton
@Component(
        modules = {
                AppModule.class,
                ShoppingCartModule.class,
                PersistenceModule.class
        }
)
public interface AppComponent {
    void inject(ProductListener presenter);
    void inject(MainActivity activity);
    void inject(ProductListPresenter presenter);
}
```

Step 4 – Implement the Presenter: We can now implement the Presenter like this. Notice that we require the View to be passed in via the constructor.

Hide Shrink ▲ Copy Code

```
public class ProductListPresenter implements ProductListContract.Actions{
    private final ProductListContract.View mView;
    @Inject ProductListContract.Repository mRepository;
    @Inject ShoppingCart mCart;



    public ProductListPresenter(ProductListContract.View mView) {
        this.mView = mView;
        ProntoShopApplication.getInstance().getAppComponent().inject(this);
        ProntoShopApplication.getInstance().getAppComponent().inject(this);
    }


    @Override
    public void loadProducts() {
        List<Product> availableProducts = mRepository.getAllProducts();
        if (availableProducts != null && availableProducts.size() > 0){
            mView.hideEmptyText();
            mView.showProducts(availableProducts);

        }else {
            mView.showEmptyText();
        }

    }

    @Override
    public void onAddProductButtonClicked() {
        mView.showAddProductForm();
    }

    @Override

    public void onAddToCartButtonClicked(Product product) {
        //perform add to checkout button here
        LineItem item = new LineItem(product, 1);
        mCart.addItemToCart(item);
    }

    @Override
    public Product getProduct(long id) {
        return mRepository.getProductById(id);
    }

    @Override
    public void addProduct(Product product) {
        mRepository.addProduct(product);
    }

    @Override
    public void onDeleteProductButtonClicked(Product product) {
        mView.showDeleteProductPrompt(product);
    }

    @Override
    public void deleteProduct(Product product) {
        mRepository.deleteProduct(product);
        loadProducts();
    }

    @Override
    public void onEditProductButtonClicked(Product product) {
        //mView.showEditProductForm(product);
        mView.showEditProductForm(product);
    }

    @Override
    public void updateProduct(Product product) {
        mRepository.updateProduct(product);    }


}
```

As you can see from the code above, Presenter does not need to know who the View is, and because of this, you can refine, refactor and test your Presenter logic independently. When business requirements change, you just define more methods in the Contract and then have the respective classes to implement them.

Step 5 – Implement the View: The View can be implemented in the Activity of the Fragment, however for a number of reasons including handling of configuration changes it is better to implement the View with Fragment. We can add a Fragment named ProductListFragment.java and here is how we can have this Fragment participate in the Android MVP pattern.

Hide Shrink Copy Code

```
public class ProductListFragment extends Fragment implements ProductListContract.View{

    private View mRootView;
    private ProductListContract.Actions mPresenter;

    public ProductListFragment() {
        // Required empty public constructor
    }


    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
                             Bundle savedInstanceState) {
        // Inflate the layout for this fragment
        mRootView = inflater.inflate(R.layout.fragment_product_list, container, false);

        //Instantiate the Presenter
        mPresenter = new ProductListPresenter(this);

        //Initialize Adapter and Recyclyer View here
        //With an empty list of products
        return mRootView;
    }


    @Override
    public void onResume() {
        super.onResume();
        //Ask the Presenter to load the list of products
        mPresenter.loadProducts();
    }

    @Override
    public void showProducts(List<Product> products) {
        //The Presenter returns the list of products here
        //call the replace data method in the Adapter
    }

    @Override
    public void showAddProductForm() {
        //startActivity(new Intent(Add Product Activity or Fragment))
    }

    @Override
    public void showEditProductForm(Product product) {
        //startActivity(new Intent(Edit Product Activity or Fragment))
        //Pass in the product to be edited
    }

    @Override
    public void showDeleteProductPrompt(Product product) {

    }


    @Override
    public void showEmptyText() {
        //Hide RecyclerView
        //Show a TextView showing not items in the list
    }

    @Override
    public void hideEmptyText() {
        //Hide the TextView showing no items in the list
        //Show RecyclerView
    }

    @Override
    public void showMessage(String message) {
        //Show a Toast or Snackbar with the message from the
        //Presenter such as Item added, item deleted, etc
    }
}
```

As you can see from the View implementation above, the View is just focused on displaying information passed to it from the Presenter. It is not involved with any decision. If the View receive a button click event, it tells the Presenter wich button was clicked. The Presenter decides what to do and if the response to that click event requires View participation then the Presenter calls the appropriate method in the View.

What of the Adapter? – The adapter is a component of the View and should not directly participate in the MVP pattern. The Fragment works with the Adapter. We will need to create a method in the adapter that accepts a list of the items we want to display. We can then call this method from the Fragment and pass it the list of items the Presenter passed to the Fragment and the Adapter will repaint itself.

Handle Configuration Changes

The MVP pattern should not replace the way you handle configuration changes in your app. It helps if you do not try to swim against the tide – aka keep it simple. As with most things in programming, there is not a consensus as to how best to handle configuration changes with the Presenter in Android MVP Pattern. Here are some of the opinions on this issue.

1. @czyrus advocates using a few options including using Loaders as in the Loader Framework.
2. +BradCampbellNZ – advocates using a library that survives configuration changes.
3. @nbarraille – advocates retaining an instance of the Presenter during configuration change since the Presenter.
4. @luszczuk – goes through a list of some pros and cons of various alternatives in this post on how you can handle configuration change however I am not sure I understand the position he is advocation.

And there are many alternatives that you can find on the internet; however, some of them are on the borderline of over engineering. I will advocate keeping the Presenter as light weight as possible and just recreate it within each instance of the Fragment. Use a dependency injection framework like Dagger 2 to keep singleton instance of the Repository where your AsyncTasks run with a Context that is not tied to the Fragment or Activity. If your solution has a need of the Context in your Presenter then you may be complicating things already. Also, if your View has a dependency on the model then you maybe be defeating the purpose of the Android MVP pattern.

Conclusion
<form action="//valokafor.us4.list-manage.com/subscribe/post?u=0600ce94a59d7720819aa3dd8&id=6e5492cf7d" class="frm" method="post"><input type="text" style="position: absolute !important; left: -5000px !important;" name="b_0600ce94a59d7720819aa3dd8_6e5492cf7d" value="" /><input class="" type="text" placeholder="Email Address" name="EMAIL" /><input class="" type="text" placeholder="Name" name="FNAME" />
<input type='submit' value="Get More Articles Like This">
</form>

Android MVP pattern is a design pattern that encourages the separation of concern in developing Android apps. An MVP designed app is loosely coupled, easy to maintain, test and extend. I have a video training course where I show a more practical example of MVP.

The sample source code for this tutorial can be found @ Github.