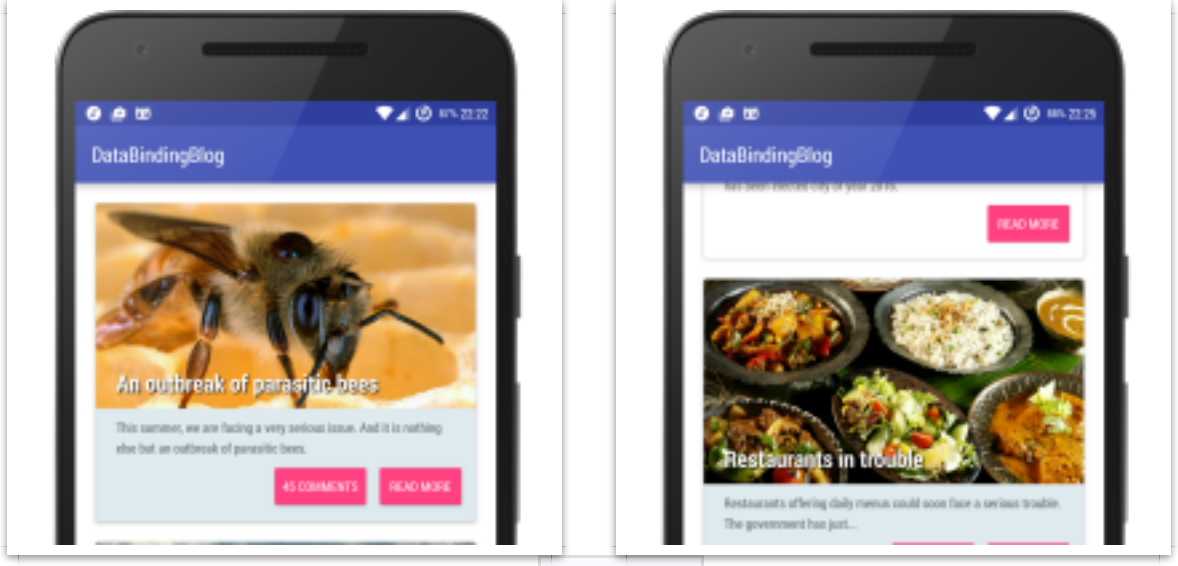
Android MVVM pattern

August 28, 2016 by Milan Bárta

We shall demonstrate the usage of Android MVVM pattern on the example application from my previous post on data binding. In short, the application displays a list of article items each containing a featured image of the article, its title, excerpt and two buttons navigating to hypothetical article comments and detail.



We modify the project by simplifying the `Article` model data class and creating a new ViewModel class acting as a bridge between the Model and View. The holding Activity doesn't change much too:

```
public class MainActivity extends AppCompatActivity {

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

        MainActivityBinding binding = DataBindingUtil.setContentView(thi
        RecyclerView.LayoutManager layoutManager = new LinearLayoutManag
        binding.contactList.setLayoutManager(layoutManager);

        List<Article> articles = new ArrayList<>();
        /* data filling */

        ArticleAdapter adapter = new ArticleAdapter(articles, this);
        binding.contactList.setAdapter(adapter);
    }
}
```

Adapter

We implement an adapter for `RecyclerView` that uses data binding for each of its items.

```
public class ArticleAdapter extends RecyclerView.Adapter<ArticleAdapter

    private List<Article> mArticles;
    private Context mContext;

    public ArticleAdapter(List<Article> mArticles, Context mContext) {
        this.mArticles = mArticles;
        this.mContext = mContext;
    }

    @Override
```

```
    public BindingHolder onCreateViewHolder(ViewGroup parent, int viewTy
        ArticleItemBinding binding = DataBindingUtil.inflate(
                LayoutInflater.from(parent.getContext()),
                R.layout.article_item, parent, false);

        return new BindingHolder(binding);
    }

    @Override
    public void onBindViewHolder(BindingHolder holder, int position) {
        ArticleItemBinding binding = holder.binding;
        binding.setAvm(new ArticleViewModel(mArticles.get(position), mCo
    }

    @Override
    public int getItemCount() {
        return mArticles.size();
    }

    public static class BindingHolder extends RecyclerView.ViewHolder {
        private ArticleItemBinding binding;

        public BindingHolder(ArticleItemBinding binding) {
            super(binding.contactCard);
            this.binding = binding;
        }
    }
}
```

Model

With the introduction of ViewModel, the `Article` data model class becomes lighter and loses the View-Model binding logic. Now it's only a POJO (Plain Old Java Object) with a constructors and getter and setter methods.

```
public class Article {

    private String title;
    private String excerpt;
    private boolean highlight;
    private String imageUrl;
    private int commentsNumber;
    private boolean read;

    /* constructor */
    /* getters and setters */
}
```

ViewModel

The ViewModel class acts here as the middle man and communicates with both Model (in our case `Article` object) and View (defined by the layout XML file). It implements the `Observable` interface by extending the `BaseObservable` class and all the View-Model binding logic has moved into its code from the `Article` class:

```
public class ArticleViewModel extends BaseObservable {

    private Article mArticle;
    private Context mContext;

    public ArticleViewModel(Article mArticle, Context mContext) {
        this.mArticle = mArticle;
        this.mContext = mContext;
    }

    @Bindable
    public String getTitle() {
        return mArticle.getTitle();
    }

    public void setTitle(String title) {
        mArticle.setTitle(title);
        notifyPropertyChanged(BR.title);
    }

    public int getCardBackgroundColor() {
        return mArticle.isHighlight() ?
                ContextCompat.getColor(mContext, R.color.highlight) :
                Color.parseColor("#ffffffff");
    }

    public int getCommentsButtonVisibility() {
        return mArticle.getCommentsNumber() == 0 ?
                View.GONE : View.VISIBLE;
    }

    public int getCommentsNumber() {
        return mArticle.getCommentsNumber();
    }

    public String getExcerpt() {
        return mArticle.getExcerpt();
    }

    public String getImageUrl() {
        return mArticle.getImageUrl();
    }

    @BindingAdapter({"image"})
    public static void loadImage(ImageView view, String url) {
        Glide.with(view.getContext()).load(url).centerCrop().into(view);
    }
```

```
48
49      public void setRead(boolean read) {
50          // change title of already read article:
51          if (read && !mArticle.isRead()) {
52              setTitle("READ: " + getTitle());
53          }
54
55          mArticle.setRead(read);
56      }
57
58      public View.OnClickListener onReadMoreClicked() {
59          return new View.OnClickListener() {
60              @Override
61              public void onClick(View view) {
62                  Toast.makeText(view.getContext(), "Opens article detail",
63                  setRead(true);
64              }
65          };
66      }
67
68      public View.OnClickListener onCommentsClicked() {
69          return new View.OnClickListener() {
70              @Override
71              public void onClick(View view) {
72                  Toast.makeText(view.getContext(), "Opens comments detail'
73              }
74          };
75      }
76  }
```

The clearer separation of View and Model layers in Android MVVM pattern can be observed for example in the `getCommentsButtonVisibility` method. Previously, the button visibility logic has been a part of the View (defined in XML). Now the visibility is decided upon in ViewModel and can be easily refactored and tested. Additionally, we no longer have to reference the `View` class as a variable from the layout file.

View (Layout XML files)

The only object the View layer has access to is now the ViewModel. Only through the ViewModel can the View get the data to present to the user. All the view logic (such as the card background colour logic for highlighted articles or button visibility) has been moved to the ViewModel class and View only calls the appropriate methods to get the result to show to the user.

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <layout xmlns:android="http://schemas.android.com/apk/res/android"
3      xmlns:app="http://schemas.android.com/apk/res-auto">
4
```

```
    <data>
        <variable
            name="avm"
            type="com.example.databindingblog.ArticleViewModel" />
    </data>

    <android.support.v7.widget.CardView
        android:id="@+id/contact_card"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_marginLeft="20dp"
        android:layout_marginRight="20dp"
        android:layout_marginTop="20dp"
        app:cardBackgroundColor="@{avm.cardBackgroundColor}"
        app:cardCornerRadius="3dp"
        app:cardElevation="3dp">

        <RelativeLayout
            android:layout_width="match_parent"
            android:layout_height="match_parent">

            <ImageView
                android:id="@+id/image"
                android:layout_width="match_parent"
                android:layout_height="200dp"
                android:layout_alignParentTop="true"
                app:image="@{avm.imageUrl}" />

            <TextView
                android:id="@+id/title"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:layout_alignBottom="@+id/image"
                android:layout_alignStart="@+id/image"
                android:layout_marginBottom="10dp"
                android:layout_marginEnd="20dp"
                android:layout_marginStart="20dp"
                android:ellipsize="end"
                android:lines="1"
                android:shadowColor="@android:color/black"
                android:shadowDx="4"
                android:shadowDy="4"
                android:shadowRadius="4"
                android:text="@{avm.title}"
                android:textColor="@android:color/white"
                android:textSize="25sp"
                android:textStyle="bold" />

            <TextView
                android:id="@+id/excerpt"
                android:layout_width="wrap_content"
                android:layout_height="wrap_content"
                android:layout_alignStart="@+id/image"
                android:layout_below="@+id/image"
```

```
            android:layout_marginBottom="5dp"
            android:layout_marginLeft="20dp"
            android:layout_marginRight="20dp"
            android:layout_marginTop="10dp"
            android:lineSpacingMultiplier="1.2"
            android:text="@{avm.excerpt}"
            android:textAppearance="?android:attr/textAppearanceSmall

        <Button
            android:id="@+id/read_more"
            style="@style/Widget.AppCompat.Button.Colored"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_alignParentEnd="true"
            android:layout_below="@+id/excerpt"
            android:layout_marginBottom="10dp"
            android:layout_marginEnd="10dp"
            android:onClick="@{avm.onReadMoreClicked}"
            android:padding="10dp"
            android:text="Read more" />

        <Button
            android:id="@+id/comments"
            style="@style/Widget.AppCompat.Button.Colored"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_below="@+id/excerpt"
            android:layout_marginBottom="10dp"
            android:layout_marginEnd="5dp"
            android:layout_toStartOf="@+id/read_more"
            android:onClick="@{avm.onCommentsClicked}"
            android:text="@{@plurals/numberOfComments(avm.commentsNum
            android:visibility="@{avm.commentsButtonVisibility}" />

    </RelativeLayout>
</android.support.v7.widget.CardView>
</layout>
```

Conclusion

To conclude, if you decide to go with data binding (see my post on Android data binding) and plan to use it in more complex project, using the Android MVVM pattern is definitely the right way to go. It clearly separates the View and Model layers by introducing the ViewModel middle-man containing the view logic. Not only does the code become better structured, but it will also be much easier to test.

Here, you can download the whole example application project.