Displaying Data From Code

In this lesson, we will implement the recycler view adapter to render a list of blog articles.

WE'LL COVER THE FOLLOWING

- Loading data from the Internet
- List Adapter
- Connecting the dots

Loading data from the Internet

During the previous lessons, we created a `BlogHttpClient` which loads a list of blog articles. Let's modify our code to support the new JSON format.

Old JSON format:

```
{
  "data": [
    {
      "title": "G'day from Sydney",
      "date": "August 2, 2019",
      "views": 2687,
      "rating": 4.4,
      "image": "https://bitbucket.org/dmytrodanylyk/travel-blog-resources/raw/3436e16367c8ec2
      "author": {
        "name": "Grayson Wells",
        "avatar": "https://bitbucket.org/dmytrodanylyk/travel-blog-resources/raw/3436e16367c8
      },
      "description": "G'day mate! Welcome to Sydney, where you come for the ..."
    }
}
```

The main changes in the new JSON format are:

- we have `id` attribute
- the `image` and `avatar` now contain a relative path

```
{
  "data": [
    {
      "id": 1,

      "title": "G'day from Sydney",
      "date": "August 2, 2019",
      "views": 2687,
      "rating": 4.4,
      "image": "/images/sydney_image.jpg",
      "author": {
        "name": "Grayson Wells",
        "avatar": "/avatars/avatar1.jpg"
      },
      "description": "G'day mate! Welcome to Sydney, where you come for the ..."
    }
  ]
}
```

In the `BlogHttpClient`, we need to change the URL to the following:

```
  public static final String BASE_URL = "https://bitbucket.org/dmytrodanylyk/travel-blog-reso
  public static final String PATH = "/raw/3eede691af3e8ff795bf6d31effb873d484877be";
  private static final String BLOG_ARTICLES_URL = BASE_URL + PATH + "/blog_articles.json";
```

BlogHttpClient

This gives us the ability to concatenate `BASE_URL` with the relative path of blog `image` or author `avatar`. Let’s add additional get methods to `Author` class and `Blog` class which are going to return full image URLs.

```
public class Author {
    ...
    private String avatar;

    public String getName() {
        return name;
    }

    public String getAvatarURL() {
        return BlogHttpClient.BASE_URL + BlogHttpClient.PATH + getAvatar();
    }
}

public class Blog {

    ...
    private String image;

    public String getImage() {
        return image;
    }

    public String getImageURL() {
        return BlogHttpClient.BASE_URL + BlogHttpClient.PATH + getImage();
    }
}
```

Now, we can modify `MainActivity` code and add a blog loading code similar to what we have in `BlogDetailsActivity`.

```
public class MainActivity extends AppCompatActivity {

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

        loadData();
    }

    private void loadData() {
        BlogHttpClient.INSTANCE.loadBlogArticles(new BlogArticlesCallback() {
            @Override
            public void onSuccess(List<Blog> blogList) {
                runOnUiThread(() -> {
                    // TODO show data
                });
            }

            @Override
            public void onError() {
                runOnUiThread(() -> {
                    showErrorSnackbar();
                });
            }
        });
    }

    private void showErrorSnackbar() {
        View rootView = findViewById(android.R.id.content);
        Snackbar snackbar = Snackbar.make(rootView, "Error during loading blog articles", Sna
        snackbar.setActionTextColor(getResources().getColor(R.color.orange500));
        snackbar.setAction("Retry", v -> {
            loadData();
            snackbar.dismiss();
        });
        snackbar.show();
    }
}
```

List Adapter

In order for `RecyclerView` to render and reuse the list of items, we need to use one of the implementations of `RecyclerView.Adapter` class.

Let's create a new package `com.travelblog.adapter` and put a new class `MainAdapter` there. We are going to use `ListAdapter` which is the implementation of `RecyclerView.Adapter`, so let's make our `MainAdapter` extend `ListAdapter` and go over through the recycler view adapter concept.

```
public class MainAdapter extends
                ListAdapter<Blog, MainAdapter.MainViewHolder> { //1

    @NonNull
    @Override
    public MainViewHolder onCreateViewHolder(@NonNull ViewGroup parent,
                                              int viewType) {
        // 2
    }

    @Override
    public void onBindViewHolder(MainViewHolder holder, int position) {
        // 3
    }

    static class MainViewHolder extends RecyclerView.ViewHolder {
        // 4
    }
}
```

Whenever we extend `ListAdapter`, we need to specify two generic types: the model and the view holder (1). In our case the model is `Blog` and the view holder object is `MainViewHolder` (4). The view holder class is going to contain references to our list item views because trying to find them every time when we need to render something cost time. In other words, the view holder is just an optimization.

There are two main methods which we need to override:

- `onCreateViewHolder` (2) this method is called **only a certain amount** of times when recycler view needs a new view holder object
- `onBindViewHolder` (3) this method is called **every** time when we need to render a list item, or in other words bind the *model* to the *view holder*

The general algorithm of how recycler view works:

- executes `onCreateViewHolder` the number of times which is equal to the number of list items which can appear on the screen at the same time
- executes `onBindViewHolder` every time we need to render list item
- when a user scrolls the list, views which are not visible any more get reused with a corresponding view holder

Now, we can implement the adapter logic. First, let's add `MainViewHolder` constructor and bind the views to the Java fields. Second, we need `bindTo` method which has Blog model as a parameter. The bindTo method will bind the data from the model to the view.

```
public class MainAdapter extends ListAdapter<Blog, MainAdapter.MainViewHolder> {
    ...
    static class MainViewHolder extends RecyclerView.ViewHolder {

        private TextView textTitle;
        private TextView textDate;
        private ImageView imageAvatar;

        MainViewHolder(@NonNull View itemView) {
            super(itemView);
            textTitle = itemView.findViewById(R.id.textTitle);
            textDate = itemView.findViewById(R.id.textDate);
            imageAvatar = itemView.findViewById(R.id.imageAvatar);
        }

        void bindTo(Blog blog) {
            textTitle.setText(blog.getTitle());
            textDate.setText(blog.getDate());

            Glide.with(itemView)
                    .load(blog.getAuthor().getAvatarURL())
                    .transform(new CircleCrop())
                    .transition(DrawableTransitionOptions.withCrossFade())
                    .into(imageAvatar);
        }
    }
}
```

Once the view holder is implemented, add the logic to `onCreateViewHolder` and `onBindViewHolder`.

In the `onCreateViewHolder` method, we can create the `LayoutInflater` object, use it to inflate the `item_main.xml` layout and pass it to the `MainViewHolder`.

In the `onBindViewHolder` method, we can simply get the current `Blog` object via `getItem` method and bind it to the `MainViewHolder`.

```
public class MainAdapter extends ListAdapter<Blog, MainAdapter.MainViewHolder> {

    @NonNull
    @Override
    public MainViewHolder onCreateViewHolder(@NonNull ViewGroup parent,
                                             int viewType) {
        LayoutInflater inflater = LayoutInflater.from(parent.getContext());
        View view = inflater.inflate(R.layout.item_main, parent, false);
        return new MainViewHolder(view);
    }
    @Override
    public void onBindViewHolder(MainViewHolder holder, int position) {

        holder.bindTo(getItem(position));
    }

    static class MainViewHolder extends RecyclerView.ViewHolder {
        ...
    }
}
```

There is one more thing which we need to implement in order for the `ListAdapter` to work correctly - `DiffUtil.ItemCallback`. This is a special callback that is used by the adapter to figure out if the items are the same and if the item content is the same for further difference calculation to re-render only items that have changed.

The `DiffUtil.ItemCallback` implementation is straightforward, it has two methods:

- `areItemsTheSame` where we compare id of the old item with an id of the new item
- `areContentsTheSame` where we compare data of old item with data of new item

In order for the adapter to use the `DiffUtil.ItemCallback`, we need to pass it to the constructor.

```
public class MainAdapter extends ListAdapter<Blog, MainAdapter.MainViewHolder> {

    public MainAdapter() {
        super(DIFF_CALLBACK);
    }

    ...

    private static final DiffUtil.ItemCallback<Blog> DIFF_CALLBACK =
            new DiffUtil.ItemCallback<Blog>() {
                @Override
                public boolean areItemsTheSame(@NonNull Blog oldData,
                                                @NonNull Blog newData) {
                    return oldData.getId().equals(newData.getId());
                }

                @Override
                public boolean areContentsTheSame(@NonNull Blog oldData,
                                                  @NonNull Blog newData) {
                    return oldData.equals(newData);
                }
            };
}
```

```
}
```

Note: Since we used the `equals` method to compare `Blog` items, we need to overwrite this method.

```
public class Blog {

    private String id;
    private Author author;
    private String title;
    private String date;
    private String image;
    private String description;
    private int views;
    private float rating;

    ...

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Blog blog = (Blog) o;
        return views == blog.views &&
                Float.compare(blog.rating, rating) == 0 &&
                Objects.equals(id, blog.id) &&
                Objects.equals(author, blog.author) &&
                Objects.equals(title, blog.title) &&
                Objects.equals(date, blog.date) &&
                Objects.equals(image, blog.image) &&
                Objects.equals(description, blog.description);
    }

    @Override
    public int hashCode() {
        return Objects.hash(id, author, title, date, image, description, views, rating);
    }
}

public class Author {

    private String name;
    private String avatar;

    ...

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Author author = (Author) o;
        return Objects.equals(name, author.name) &&
                Objects.equals(avatar, author.avatar);
    }

    @Override
    public int hashCode() {
        return Objects.hash(name, avatar);
    }
}
```

Connecting the dots

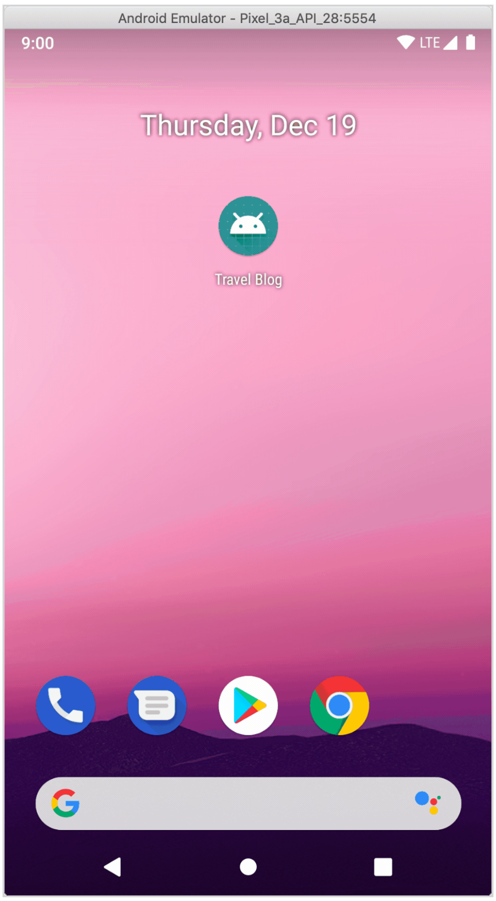
Now that we have the data and the adapter, we can connect everything in the `MainActivity`.

Create the `MainAdapter` and save it into the Java field so we can use it later. Bind the `RecyclerView` from XML to Java objects via `findViewById` method and set the adapter. We also need to set the layout manager to `LinearLayoutManager`, because `RecyclerView` can also render a grid layout.

Finally, in the `loadData` method, when the data is successfully loaded, set the blog list to the adapter view `submitList` method.

```
public class MainActivity extends AppCompatActivity {

    private MainAdapter adapter;

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

        adapter = new MainAdapter();

        RecyclerView recyclerView = findViewById(R.id.recyclerView);
        recyclerView.setLayoutManager(new LinearLayoutManager(this));
        recyclerView.setAdapter(adapter);

        loadData();
    }

    private void loadData() {
        BlogHttpClient.INSTANCE.loadBlogArticles(new BlogArticlesCallback() {
            @Override
            public void onSuccess(List<Blog> blogList) {
                runOnUiThread(() -> {
                    adapter.submitList(blogList);
                });
            }

            @Override
            public void onError() {
                runOnUiThread(() -> {
                    showErrorSnackbar();
                });
            }
        });
    }

    ...
}
```

Time to see what we have done in action! When we launch the application, `MainActivity` should load and display a list of blog articles.



Hit the *run* button to try it yourself.

```
package com.travelblog.adapter;

import android.view.*;
import android.widget.*;

import androidx.annotation.*;
import androidx.recyclerview.widget.ListAdapter;
import androidx.recyclerview.widget.*;

import com.bumptech.glide.*;
import com.bumptech.glide.load.resource.bitmap.*;
import com.bumptech.glide.load.resource.drawable.*;
import com.travelblog.R;
import com.travelblog.http.*;

public class MainAdapter extends ListAdapter<Blog, MainAdapter.MainViewHolder> {

    public MainAdapter() {
        super(DIFF_CALLBACK);
    }

    @NonNull
    @Override
    public MainViewHolder onCreateViewHolder(@NonNull ViewGroup parent, int viewType) {
        LayoutInflater inflater = LayoutInflater.from(parent.getContext());
        View view = inflater.inflate(R.layout.item_main, parent, false);
        return new MainViewHolder(view);
    }

    @Override
    public void onBindViewHolder(MainViewHolder holder, int position) {
        holder.bindTo(getItem(position));
    }

    static class MainViewHolder extends RecyclerView.ViewHolder {

        private TextView textTitle;
        private TextView textDate;
        private ImageView imageAvatar;

        MainViewHolder(@NonNull View itemView) {
            super(itemView);
            textTitle = itemView.findViewById(R.id.textTitle);
            textDate = itemView.findViewById(R.id.textDate);
            imageAvatar = itemView.findViewById(R.id.imageAvatar);
        }

        void bindTo(Blog blog) {
            textTitle.setText(blog.getTitle());
            textDate.setText(blog.getDate());

            Glide.with(itemView)
                    .load(blog.getAuthor().getAvatarURL())
                    .transform(new CircleCrop())
                    .transition(DrawableTransitionOptions.withCrossFade())
                    .into(imageAvatar);
        }

    }

    private static final DiffUtil.ItemCallback<Blog> DIFF_CALLBACK =
            new DiffUtil.ItemCallback<Blog>() {
                @Override
                public boolean areItemsTheSame(@NonNull Blog oldData,
                                               @NonNull Blog newData) {
                    return oldData.getId().equals(newData.getId());
                }

                @Override
                public boolean areContentsTheSame(@NonNull Blog oldData,
                                                  @NonNull Blog newData) {
                    return oldData.equals(newData);
                }
            };
}
```

The next lesson will discuss how to add a pull-to-refresh functionality.