Loading Data From the Internet

This lesson shows how to load and display the data from the internet.

WE'LL COVER THE FOLLOWING

Overview

Most of the mobile applications show data that is loaded from the backend web service. The most popular way of communication is via REST, and the most popular format of communication is JSON.

In our Travel Blog Application, we are going to load data from the Internet via simple HTTP GET request, parse a JSON file and display the data from the JSON file in a user-friendly manner.

Dependencies

While it's possible to use a Java HTTP client to load data from the Internet, on Android most of the projects use a popular, open-source OkHttp developed by square.

To convert the JSON objects to Java objects we are going to use Gson library, which works nicely with OkHttp.

Let's add the new library dependencies to the *app/build.gradle* file in the `dependencies` section.

```
dependencies {
    // ui
    implementation 'androidx.appcompat:appcompat:1.1.0'

    implementation 'androidx.constraintlayout:constraintlayout:1.1.3'
    implementation 'com.google.android.material:material:1.1.0-alpha10'
    implementation 'com.github.bumptech.glide:glide:4.10.0'
    implementation 'com.squareup.okhttp3:okhttp:4.2.1'
    implementation 'com.google.code.gson:gson:2.8.6'
}
```

build.gradle

Data models

Before creating HTTP client, let’s take a look into the JSON which we are going to receive and create corresponding model classes.

```
{
  "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": "Australia is one of the most popular travel destinations in the world."
    }
  ]
}
```

The JSON has a `data` element which is an array of blog articles. Every blog article contains some information like `title`, `description`, `date` along with nested `author` object. Let’s try to represent this via Java model classes and put them into a separate package `com.travelblog.http`.

This is an object to hold an array of blog articles.

```
public class BlogData {

    private List<Blog> data;

    public List<Blog> getData() {
        return data != null ? data : new ArrayList<>();
    }
}
```

This is an object to hold blog article information

This is an object to hold blog article information.

```
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;

    public String getTitle() {
        return title;
    }

    public String getDate() {
        return date;
    }

    public String getImage() {
        return image;
    }

    public String getDescription() {
        return description;
    }

    public int getViews() {
        return views;
    }

    public float getRating() {
        return rating;
    }

    public Author getAuthor() {
        return author;
    }

    public void setAuthor(Author author) {
        this.author = author;
    }

    public String getId() {
        return id;
    }
}
```

This is an object to hold blog author information.

```
public class Author {

    private String name;
    private String avatar;
    public String getName() {
        return name;
    }


    public String getAvatar() {
        return avatar;
    }
}
```

HTTP client

Now that our models are ready, let's create the `BlogHttpClient` class which will be responsible for making network calls.

We should have only one instance of `BlogHttpClient` across the application. That's why the `BlogHttpClient` class is declared as `final` and the constructor is `private`. To access the object instance `BlogHttpClient#INSTANCE` can be used.

```
public final class BlogHttpClient {

    public static final BlogHttpClient INSTANCE = new BlogHttpClient();

    private static final String BASE_URL =
        "https://bitbucket.org/dmytrodanylyk/travel-blog-resources/raw/";
    private static final String BLOG_ARTICLES_URL =
        BASE_URL + "8550ef2064bf14fcf3b9ff322287a2e056c7e153/blog_articles.json";

    private BlogHttpClient() {
        ...
    }
}
```

To make a network, call we need to declare several fields:

- `Executor` to run network calls on the background thread, because we should not block UI thread
- `OkHttpClient` to make network calls
- `Gson` to parse JSON data

```
public final class BlogHttpClient {
    ...
    private Executor executor;
    private OkHttpClient client;
    private Gson gson;

    private BlogHttpClient() {
        executor = Executors.newFixedThreadPool(4);
        client = new OkHttpClient();
        gson = new Gson();
    }
    }
```

> Android has a concept of UI thread *(also known as "main thread")*; it's the only thread that you can use to touch the UI components. All heavy works should be delegated to the background thread to keep the user interface responsive. When the work on the background thread is completed, we can use the method of the activity `runOnUiThread` to switch from the background to UI thread.

Let's declare a `loadBlogArticles` method in the `BlogHttpClient` and try to understand what we are doing here:

- (1) create a `Request` object which defines the type of request and URL
- (2) use `Executor` to execute code on the background thread
- (3) execute an `OkHttpClient` request to get the `Response` and `ResponseBody`
- (4) if `ResponseBody` is not null, we use `Gson` to parse the JSON by providing JSON String object and class of the result data `BlogData.class`
- (5) the whole network call section is wrapped by `try/catch` to catch the exception and log it via `Log.e` method

```
public void loadBlogArticles() {
    Request request = new Request.Builder() // 1
            .get()
            .url(BLOG_ARTICLES_URL)
            .build();

    executor.execute(() -> { // 2
        try {
            Response response = client.newCall(request).execute(); // 3
            ResponseBody responseBody = response.body();
            if (responseBody != null) { // 4
                String json = responseBody.string();
                BlogData blogData = gson.fromJson(json, BlogData.class);
                if (blogData != null) {
                    // success blogData.getData()
                    return;
                }
            }
        } catch (IOException e) { // 5
            Log.e("BlogHttpClient", "Error loading blog articles", e);
        }
        // error
    });
}
```

The final step is to add a callback listener `BlogArticlesCallback` to deliver the result to the caller class. We basically need `onSuccess` method to deliver a list

of blog articles and `onError` method to deliver the error result.

```
public interface BlogArticlesCallback {
    void onSuccess(List<Blog> blogList);
    void onError();
}
```

Caller class should pass `BlogArticlesCallback` as a parameter to `loadBlogArticles` method, which we can use to execute `onSuccess` or `onError` method.

```
public void loadBlogArticles(BlogArticlesCallback callback) {
    Request request = new Request.Builder()
            .get()
            .url(BLOG_ARTICLES_URL)
            .build();

    executor.execute(() -> {
        try {
            Response response = client.newCall(request).execute();
            ResponseBody responseBody = response.body();
            if (responseBody != null) {
                String json = responseBody.string();
                BlogData blogData = gson.fromJson(json, BlogData.class);
                if (blogData != null) {
                    callback.onSuccess(blogData.getData());
                    return;
                }
            }
        } catch (IOException e) {
            Log.e("BlogHttpClient", "Error loading blog articles", e);
        }
        callback.onError();
    });
}
```

Loading data from the Internet

Now we can use `BlogHttpClient` in the `BlogDetailsActivity` to load and display the data from the Internet.

Start by refactoring `BlogDetailsActivity` a bit:

- remove setting hard-coded data
- create global fields for views which we are going to use later
- call the `loadData` method at the very end

```
public class BlogDetailsActivity extends AppCompatActivity {

    private TextView textTitle;
    private TextView textDate;
    private TextView textAuthor;
    private TextView textRating;
    private TextView textDescription;
    private TextView textViews;
    private RatingBar ratingBar;
    private ImageView imageAvatar;
    private ImageView imageMain;

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

        imageMain = findViewById(R.id.imageMain);
        imageAvatar = findViewById(R.id.imageAvatar);

        ImageView imageBack = findViewById(R.id.imageBack);
        imageBack.setOnClickListener(v -> finish());

        textTitle = findViewById(R.id.textTitle);
        textDate = findViewById(R.id.textDate);
        textAuthor = findViewById(R.id.textAuthor);
        textRating = findViewById(R.id.textRating);
        textViews = findViewById(R.id.textViews);
        textDescription = findViewById(R.id.textDescription);
        ratingBar = findViewById(R.id.ratingBar);

        // start data loading
        loadData()
    }
}
```

It’s time to implement the `loadData` method:

- (1) get the instance of HTTP client using `BlogHttpClient.INSTANCE` method
- (2) call `loadBlogArticles` method and pass `BlogArticlesCallback`
- (3) inside the `onSuccess` method we are still on the background thread, use `runOnUiThread` to switch to the UI thread
- (4) call `showData` method and pass a first blog article

```
private void loadData() {
    BlogHttpClient.INSTANCE.loadBlogArticles(new BlogArticlesCallback() { // 1, 2
        @Override
        public void onSuccess(List<Blog> blogList) { // 3
            runOnUiThread(() -> showData(blogList.get(0))); // 4
        }
        @Override
        public void onError() {
            // handle error

        }
    });
}
```

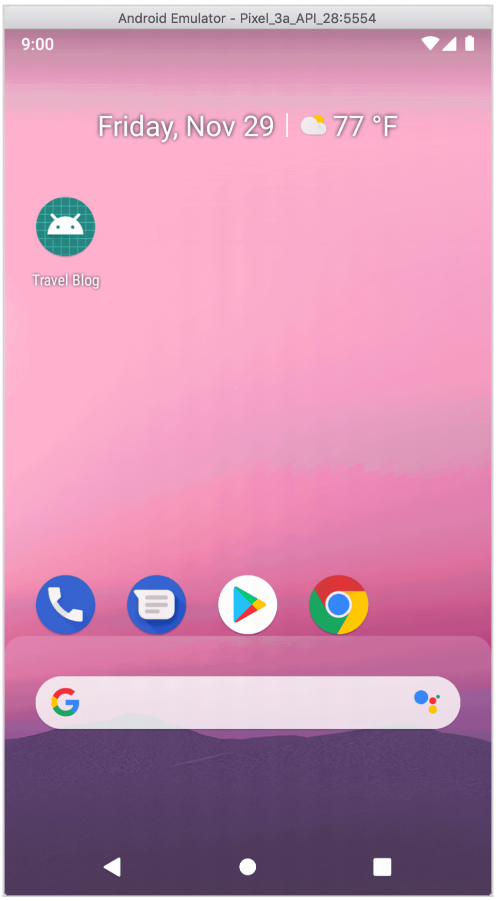
Finally, we can implement the `showData` method to display the data to the user using `Blog` properties.

```
private void showData(Blog blog) {
    textTitle.setText(blog.getTitle());
    textDate.setText(blog.getDate());
    textAuthor.setText(blog.getAuthor().getName());
    textRating.setText(String.valueOf(blog.getRating()));
    textViews.setText(String.format("(%d views)", blog.getViews()));
    textDescription.setText(blog.getDescription());
    ratingBar.setRating(blog.getRating());

    Glide.with(this)
            .load(blog.getImage())
            .transition(DrawableTransitionOptions.withCrossFade())
            .into(imageMain);

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

Now, when we launch the application, all data will be loaded from the internet.



Hit the *run* button to try it yourself.

```
package com.travelblog.http;

public class Author {

    private String name;
    private String avatar;

    public String getName() {
        return name;
    }

    public String getAvatar() {
        return avatar;
    }
}
```

The next lesson will introduce how to add a loading indicator when blog data is loading.