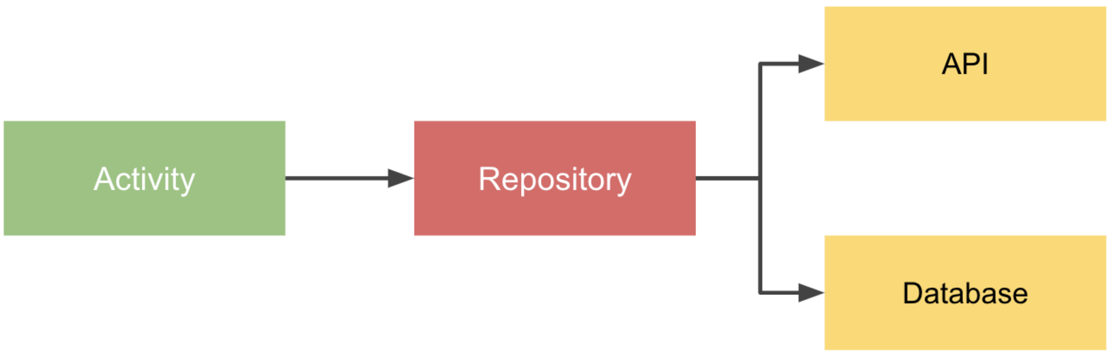
# Repository Pattern

This lesson will go over the implementation of the Repository pattern.



The *Repository* pattern helps us to abstract away the way we retrieve the data. The *Repository* provides a clean API so that the rest of the app can retrieve data easily.

Before implementing the *Repository* pattern, let’s re-factor `BlogHttpClient`. Since we are not going to use the `BlogHttpClient` directly inside activity, we can remove the listener from the `loadBlogArticles` method and make it blocking call instead.

After Before

```
public List<Blog> loadBlogArticles() {
    ...
}
```

Here is what the `loadBlogArticles` method looks after re-factoring.

After Before

```
public List<Blog> loadBlogArticles() {
```

```
public List<Blog> loadBlogArticles() {
    Request request = new Request.Builder()
            .get()

            .url(BLOG_ARTICLES_URL)
            .build();

    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) {
                return blogData.getData();
            }
        }
    } catch (IOException e) {
        Log.e("BlogHttpClient", "Error loading blog articles", e);
    }
    return null;
}
```

Now, create the `BlogRepository` class in the `com.travelblog.repository` package. In the constructor, we need to initialize three properties:

- `BlogHttpClient` to get the data from the network
- `AppDatabase` to get the data from the database
- `Executor` to perform work on the background thread

```
public class BlogRepository {

    private BlogHttpClient httpClient;
    private AppDatabase database;
    private Executor executor;

    public BlogRepository(Context context) {
        httpClient = BlogHttpClient.INSTANCE;
        database = DatabaseProvider.getInstance(context.getApplicationContext());
        executor = Executors.newSingleThreadExecutor();
    }
}
```

It’s time to implement the `loadDataFromDatabase` and `loadDataFromNetwork` method.

The implementation of `loadDataFromDatabase` method is pretty simple. We use `executor` to perform work on the background thread and use the `database.blogDao().getAll()` method to get a list of blog articles from the database.

```
public void loadDataFromDatabase(DataFromDatabaseCallback callback) {
    executor.execute(() -> callback.onSuccess(database.blogDao().getAll()));

}
```

Because the `loadDataFromDatabase` method delivers the result asynchronously, we also need to define a callback.

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

The implementation of `loadDataFromNetwork` is a bit more complicated. This method also delivers the result asynchronously, so let's define the callback first. Unlike the `DataFromDatabaseCallback` the `DataFromNetworkCallback` may fail, so we have `onSuccess` and `onError` methods.

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

Similarly to the `loadDataFromDatabase` we use executor to perform work on the background thread.

Now, when the blog list is *null*, we trigger `onError` callback *(1)*.

When the list is *not null*, we delete all the data from the database *(2)*, save the new data to the database *(3)* and deliver the data via `onSuccess` method *(4)*.

```
public void loadDataFromNetwork(DataFromNetworkCallback callback) {
    executor.execute(() -> {
        List<Blog> blogList = httpClient.loadBlogArticles();
        if (blogList == null) {
            callback.onError(); // 1
        } else {
            BlogDAO blogDAO = database.blogDao();
            blogDAO.deleteAll(); // 2
            blogDAO.insertAll(blogList); // 3
            callback.onSuccess(blogList); // 4
        }
    });
}
```

In the next lesson, we will use the `BlogRepository` object in the `MainActivity` to finish the implementation of offline support to finish the implementation of offline support.