Mobile Application Development

Produced by

David Drohan (ddrohan@wit.ie)

Department of Computing & Mathematics
Waterford Institute of Technology
http://www.wit.ie

Waterford Institute *of* Technology
INSTITIÚID TEICNEOLAÍOCHTA PHORT LÁIRGE

Android & Retrofit

A type-safe HTTP client for Android & Java

Retrofit

A type-safe **HTTP client** for Android and Java

Introduction

Retrofit turns your HTTP API into a Java interface.

```
public interface GitHubService {
  @GET("users/{user}/repos")
  Call<List<Repo>> listRepos(@Path("user") String user);
}
```

The `Retrofit` class generates an implementation of the `GitHubService` interface.

```
Retrofit retrofit = new Retrofit.Builder()
    .baseUrl("https://api.github.com/")
    .build();

GitHubService service = retrofit.create(GitHubService.class);
```

Each `Call` from the created `GitHubService` can make a synchronous or asynchronous HTTP request to the remote webserver.

```
Call<List<Repo>> repos = service.listRepos("octocat");
```

Agenda & Goals

- Investigate the use of Retrofit in App Development
- Be aware of the different Retrofit Annotations and Classes and how, when and where to use them
- Revisit Java interfaces
- Understand how to integrate Retrofit into an Android App
- Refactor our CoffeeMate Case Study

What is it?

- Retrofit is a Java Library that turns your REST API into a Java interface
- Simplifies HTTP communication by turning remote APIs into declarative, type-safe interfaces
- Developed by Square (Jake Wharton)
- Retrofit is one of the most popular HTTP Client Library for Android as a result of its simplicity and its great performance compared to the others (next slide)
- Retrofit makes use of OkHttp (from the same developer) to handle network requests.

Retrofit Performance Analysis

	One Discussion	Dashboard (7 requests)	25 Discussions
AsyncTask	941 ms	4,539 ms	13,957 ms
Volley	560 ms	2,202 ms	4,275 ms
Retrofit	312 ms	889 ms	1,059 ms

Why Use it?

- Developing your own type-safe HTTP library to interface with a REST API can be a real pain: you have to handle many functionalities such as making connections, caching, retrying failed requests, threading, response parsing, error handling, and more.
- Retrofit, on the other hand, is very well planned, documented, and tested—a battle-tested library that will save you a lot of precious time and headaches.

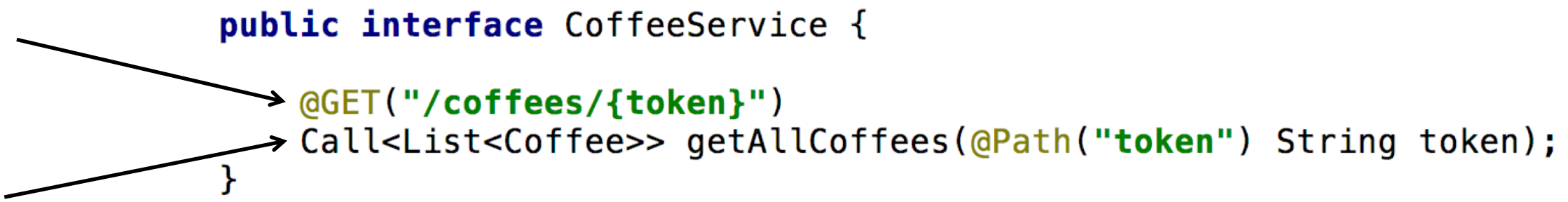
The Basics

- Again, Retrofit2 is a flexible library that uses annotated interfaces to create REST calls. To get started, let's look at our CoffeeMate example that makes a GET request for Coffees.
- Here's the Coffee class we're using:
- Much simpler if field names matches server model (but doesn't have to, see later)

```
public class Coffee
{
    public String name;
    public String shop;
    public double rating;
    public double price;
    public boolean favourite;

    public String _id;
    public String usertoken;
    public String address;
    public String googlephoto;
    public Marker marker = new Marker();


    public Coffee() {...}

    public Coffee(String name, String shop, double rating,
                  double price, boolean fav, String token,
                  double lat, double lng, String path)
    {...}

    @Override
    public String toString() {...}
}
```

The Service interface *

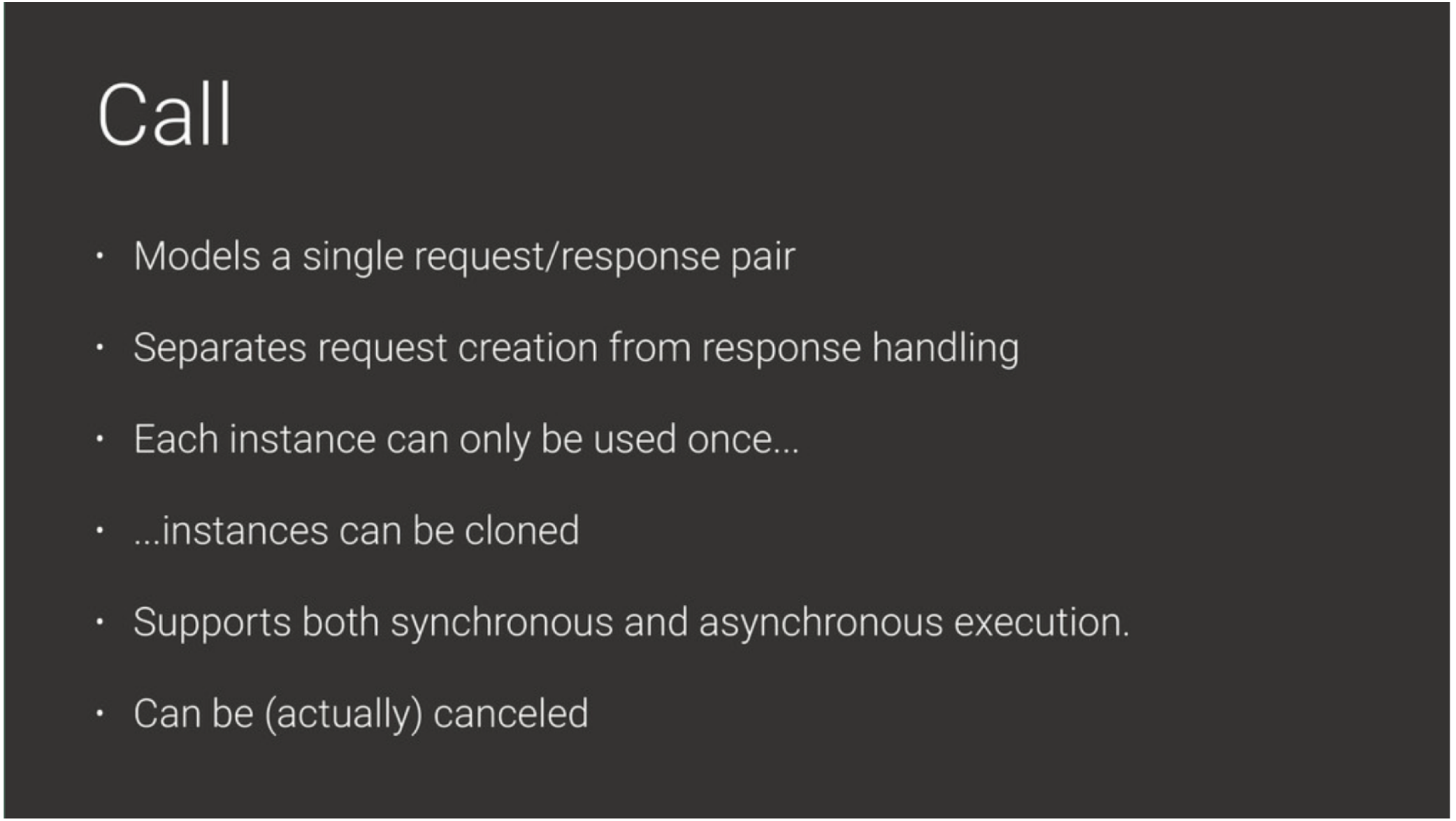
- Once we've defined the class, we can make a service interface to handle our API. A **GET** request to load all Coffees could look something like this:



```
public interface CoffeeService {

    @GET("/coffees/{token}")
    Call<List<Coffee>> getAllCoffees(@Path("token") String token);
}
```

- Note that the @GET annotation takes the endpoint we wish to hit. As you can see, an implementation of this interface will return a Call object containing a list of Coffees.

Retrofit 2.0 introduced a new **Call** class (it also exists in OkHttp package).



(screen is taken from Jack Whartons presentation during Droidcon NYC 2015, available here)

Beyond **GET** – other types of Calls

- Retrofit2 is not limited to GET requests. You may specify other REST methods using the appropriate annotations (such as @POST, @PUT and @DELETE).
- Here's another version of our **CoffeeService** interface (with full CRUD support)

```
public interface CoffeeService {

    @GET("/coffees/{token}")
    Call<List<Coffee>> getAllCoffees(@Path("token") String token);

    @POST("/coffees/{token}")
    Call<Coffee> createCoffee(@Path("token") String token,
                              @Body Coffee coffee);

    @GET("/coffees/{token}/{id}")
    Call<Coffee> retrieveCoffee(@Path("token") String token,
                                @Path("id") String id);

    @PUT("/coffees/{token}/{id}")
    Call<Coffee> updateCoffee(@Path("token") String token,
                              @Path("id") String id,
                              @Body Coffee coffee);

    @DELETE("/coffees/{token}/{id}")
    Call<List<Coffee>> deleteCoffee(@Path("token") String token,
                              @Path("id") String id);

}
```

Calling the API

- So how do we use this interface to make requests to the API?
- Use Retrofit2 to create an implementation of the above interface, and then call the desired method.
- Retrofit2 supports a number of converters used to map Java objects to the data format your server expects (JSON, XML, etc). we'll be using the Gson converter.

- Gson: `com.squareup.retrofit2:converter-gson`
- Jackson: `com.squareup.retrofit2:converter-jackson`
- Moshi: `com.squareup.retrofit2:converter-moshi`
- Protobuf: `com.squareup.retrofit2:converter-protobuf`
- Wire: `com.squareup.retrofit2:converter-wire`
- Simple XML: `com.squareup.retrofit2:converter-simplexml`
- Scalars (primitives, boxed, and String): `com.squareup.retrofit2:converter-scalars`

Aside - CoffeeMate Android Client

- **coffeemateweb** api endpoints

```
{ method: 'GET', path: '/coffees/{token}', config: CoffeeApi.findAll },
{ method: 'GET', path: '/coffees/{token}/{id}', config: CoffeeApi.findById },
{ method: 'POST', path: '/coffees/{token} ', config: CoffeeApi.addCoffee },
{ method: 'PUT', path: '/coffees/{token}/{id}', config: CoffeeApi.updateCoffee },
{ method: 'DELETE', path: /coffees/{token}/{id}', config: CoffeeApi.deleteCoffee }
```

- Use **CoffeeService** for
  - Adding / Updating / Deleting a Coffee
  - Listing All Coffees
  - Finding a single Coffee

Android Networking (Using Retrofit) in CoffeeMate.5.1

Steps to integrate Retrofit into your App

1. Set up your Project Dependencies & Permissions
2. Create Interface for API and declare methods for each REST Call, specifying method type using Annotations - @GET, @POST, @PUT, etc. For parameters use - @Path, @Query, @Body
3. Use **Retrofit** to build the service client
4. Make the REST Calls as necessary using the relevant Callback mechanism

1. Project Dependencies & Permissions *

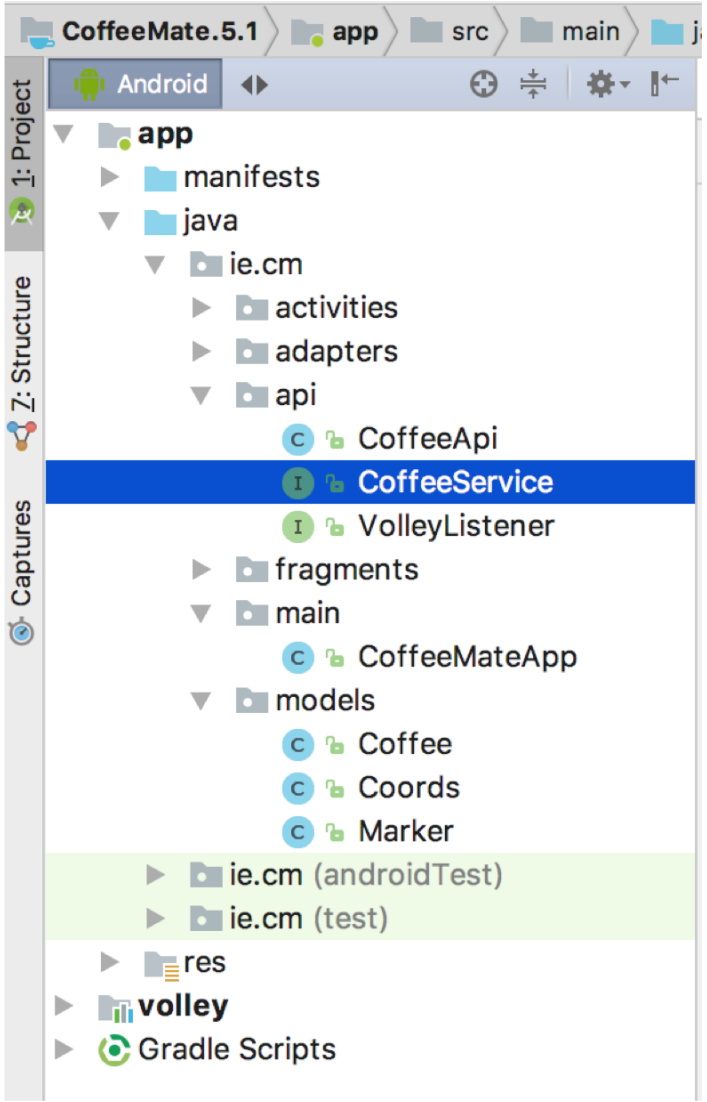
- Add the required dependencies to your build.gradle

```
dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    compile 'com.android.support:appcompat-v7:26.1.0'
    compile 'com.android.support:support-v4:26.1.0'
    compile 'com.android.support:design:26.1.0'
    compile 'com.makeramen:roundedimageview:2.2.1'
    compile 'com.android.support.constraint:constraint-layout:1.0.2'
    testCompile 'junit:junit:4.12'
    compile project(':volley')
    compile 'com.google.code.gson:gson:2.8.2'
    compile 'com.google.android.gms:play-services-auth:11.8.0'

    compile 'com.squareup.retrofit2:retrofit:2.1.0'
    compile 'com.squareup.retrofit2:converter-gson:2.1.0'
    compile 'com.squareup.okhttp3:okhttp:3.7.0'
}
```

- And the necessary permissions to your manifest – BEFORE/OUTSIDE the application tag

```
<uses-permission android:name="android.permission.INTERNET"/>
```

2. Create interface for API




```java
public interface CoffeeService {

    @GET("/coffees/{token}")
    Call<List<Coffee>> getAllCoffees(@Path("token") String token);

    @POST("/coffees/{token}")
    Call<Coffee> createCoffee(@Path("token") String token,
                              @Body Coffee coffee);

    @GET("/coffees/{token}/{id}")
    Call<Coffee> retrieveCoffee(@Path("token") String token,
                                @Path("id") String id);

    @PUT("/coffees/{token}/{id}")
    Call<Coffee> updateCoffee(@Path("token") String token,
                              @Path("id") String id,
                              @Body Coffee coffee);

    @DELETE("/coffees/{token}/{id}")
    Call<List<Coffee>> deleteCoffee(@Path("token") String token,
                              @Path("id") String id);
}
```

3. Build Service Client - CoffeeMateApp *

- Implement the necessary interface & variables

Our **`CoffeeService`** instance

```
public class CoffeeMateApp extends Application
{
    //...
    public CoffeeService coffeeService;

    //...

    public String serviceURL = "http://coffeemateweb.herokuapp.com";
```

Note the **`serviceURL`**

3. Build Service Client - CoffeeMateApp *

- Create the proxy service '**coffeeService**', with the appropriate Gson parsers

```
@Override
public void onCreate() {
    super.onCreate();

    //...
    Gson gson = new GsonBuilder().create();

    OkHttpClient okHttpClient = new OkHttpClient.Builder()
            .connectTimeout( timeout: 30, TimeUnit.SECONDS)
            .writeTimeout( timeout: 30, TimeUnit.SECONDS)
            .readTimeout( timeout: 30, TimeUnit.SECONDS)
            .build();


    Retrofit retrofit = new Retrofit.Builder()
            .baseUrl(serviceURL)
            .addConverterFactory(GsonConverterFactory.create(gson))
            .client(okHttpClient)
            .build();

    coffeeService = retrofit.create(CoffeeService.class);

    Log.v( tag: "coffeemate",  msg: "Coffee Service Created");
}
```

3. Build Service Client - CoffeeMateApp *

Gson for converting our JSON

OkHttpClient for communication timeouts (optional)

Retrofit.Builder to create an instance of our interface

```
@Override
public void onCreate() {
    super.onCreate();

    //...
    Gson gson = new GsonBuilder().create();

    OkHttpClient okHttpClient = new OkHttpClient.Builder()
            .connectTimeout( timeout: 30, TimeUnit.SECONDS)
            .writeTimeout( timeout: 30, TimeUnit.SECONDS)
            .readTimeout( timeout: 30, TimeUnit.SECONDS)
            .build();

    Retrofit retrofit = new Retrofit.Builder()
            .baseUrl(serviceURL)
            .addConverterFactory(GsonConverterFactory.create(gson))
            .client(okHttpClient)
            .build();

    coffeeService = retrofit.create(CoffeeService.class);

    Log.v( tag: "coffeemate",  msg: "Coffee Service Created");
}
```

4. Calling the API - CoffeeFragment *

- Implement the necessary interface & variables

```
public class CoffeeFragment  extends Fragment implements AdapterView.OnItemClickListener,
                                                          View.OnClickListener,
                                                          AbsListView.MultiChoiceModeListener,
                                                          Callback<List<Coffee>>

{
  //...

  public          Call<List<Coffee>>  call;

  public CoffeeMateApp app = CoffeeMateApp.getInstance();
```

Note the **Callback** interface

the **call** object, for making the requests

4. CoffeeFragment – `onResume()` *

```
@Override
public void onResume() {
  super.onResume();

  //...

  call = (Call<List<Coffee>>) app.coffeeService.getAllCoffees(app.googleToken);
  call.enqueue( callback: this);

}
```

`enqueue()` allows for asynchronous callback to our service

4. CoffeeFragment – `onResponse()` *

- Triggered on a successful call to the API
- Takes 2 parameters
  - The Call object
  - The expected Response object
- *Converted* JSON result stored in response.body()

```
@Override
public void onResponse(Call<List<Coffee>> call, Response<List<Coffee>> response) {
    app.coffeeList = response.body();

    listAdapter = new CoffeeListAdapter(getActivity(), deleteListener: this, app.coffeeList);
    coffeeFilter = new CoffeeFilter(app.coffeeList, filterText: "all",listAdapter);
    setListView(listView);

    //...
}
```

4. CoffeeFragment – `onFailure()` *

- Triggered on an unsuccessful call to the API
- Takes 2 parameters
  - The Call object
  - A Throwable object containing error info
- Probably should inform user of what's happened

```
@Override
public void onFailure(Call<List<Coffee>> call, Throwable t) {
  Toast.makeText(getActivity(),
          text: "Coffee Service Unavailable. Try again later",
          Toast.LENGTH_LONG).show();
}
```

Add UseCase *

```
public class AddFragment extends Fragment implements View.OnClickListener,
                                                      Callback<Coffee> {

    private TextView    titleBar;
    private String      coffeeName, coffeeShop;
    private double      coffeePrice, ratingValue;
    private EditText    name, shop, price;
    private RatingBar   ratingBar;
    public CoffeeMateApp app = CoffeeMateApp.getInstance();

    public Call<Coffee> callCreate;

    public AddFragment() {...}

    public static AddFragment newInstance() {...}

    @Override
    public void onCreate(Bundle savedInstanceState) {...}

    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
                             Bundle savedInstanceState) {...}

    @Override
    public void onResume() {...}
```

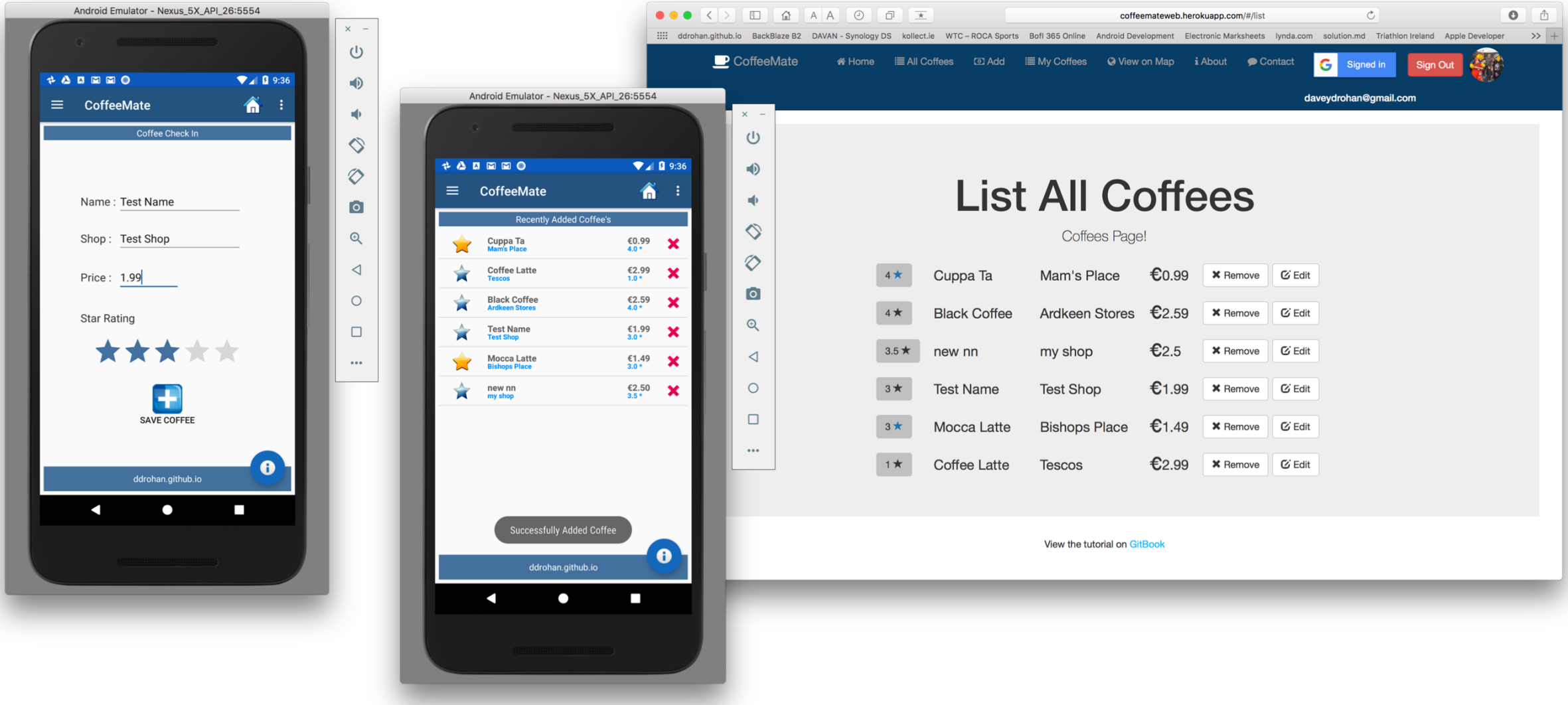
Callback and **Call** references

Add UseCase *

```
public void onClick(View v) {

    coffeeName = name.getText().toString();
    coffeeShop = shop.getText().toString();
    try {
        coffeePrice = Double.parseDouble(price.getText().toString());
    } catch (NumberFormatException e) {
        coffeePrice = 0.0;
    }
    ratingValue = ratingBar.getRating();

    if ((coffeeName.length() > 0) && (coffeeShop.length() > 0)
            && (price.length() > 0)) {
        Coffee c = new Coffee(coffeeName, coffeeShop, ratingValue,
                coffeePrice, fav: false,app.googleToken, lat: 0, lng: 0,app.googlePhotoURL);

        //app.coffeeList.add(c);
        callCreate = app.coffeeService.createCoffee(app.googleToken,c);
        callCreate.enqueue( callback: this);

    } else
        Toast.makeText(
                getActivity(),
                text: "You must Enter Something for "
                        + "\'Name\', \'Shop\' and \'Price\'",
                Toast.LENGTH_SHORT).show();
}
```

Creating the **`Call`** and triggering the **`Callback`**

CoffeeMateWeb + Mobile



Anonymous Callbacks *

Anonymous Callback allows for multiple calls in same class

```
Call<Coffee> callUpdate = app.coffeeService.updateCoffee(app.googleToken,
                                                         aCoffee._id,
                                                         aCoffee);

callUpdate.enqueue(new Callback<Coffee>() {
    @Override
    public void onResponse(Call<Coffee> call, Response<Coffee> response) {
        Toast.makeText(context, text: "Successfully Updated Coffee" ,
                Toast.LENGTH_LONG).show();
    }

    @Override
    public void onFailure(Call<Coffee> call, Throwable t) {
        Toast.makeText(context, text: "Unable to Update Coffee" ,
                Toast.LENGTH_LONG).show();
    }
});
```

Delete UseCase *

```
public void onCoffeeDelete(final Coffee coffee)
{
  String stringName = coffee.name;
  AlertDialog.Builder builder = new AlertDialog.Builder(getActivity());
  builder.setMessage("Are you sure you want to Delete the \'Coffee\' " + stringName + "?");
  builder.setCancelable(false);

  builder.setPositiveButton( text: "Yes", new DialogInterface.OnClickListener()
  {
    public void onClick(DialogInterface dialog, int id)
    {
      deleteACoffee(coffee._id);
      getAllCoffees();
    }
  }).setNegativeButton( text: "No", new DialogInterface.OnClickListener()
  {
    public void onClick(DialogInterface dialog, int id) { dialog.cancel(); }
  });
  AlertDialog alert = builder.create();
  alert.show();
}
```

Anonymous Callbacks
multiple calls here

Delete UseCase *

Anonymous Callback

```
public void deleteACoffee(String id) {
  Call<List<Coffee>> callDelete = app.coffeeService.deleteCoffee(app.googleToken,id);
  callDelete.enqueue(new Callback<List<Coffee>>() {
    @Override
    public void onResponse(Call<List<Coffee>> call, Response<List<Coffee>> response) {
      Toast.makeText(getActivity(), text: "Successfully Deleted Coffee " ,
              Toast.LENGTH_LONG).show();
    }

    @Override
    public void onFailure(Call<List<Coffee>> call, Throwable t) {
      Toast.makeText(getActivity(), text: "Unable to Delete Coffee : " + "ERROR: " +
              t.getMessage(),Toast.LENGTH_LONG).show();
    }
  });
}
```

Standard Callback

```
public void getAllCoffees() {
  callRetrieve = app.coffeeService.getAllCoffees(app.googleToken);
  callRetrieve.enqueue( callback: this);
}
```

Bridging the Gap Between Your Code & Your API

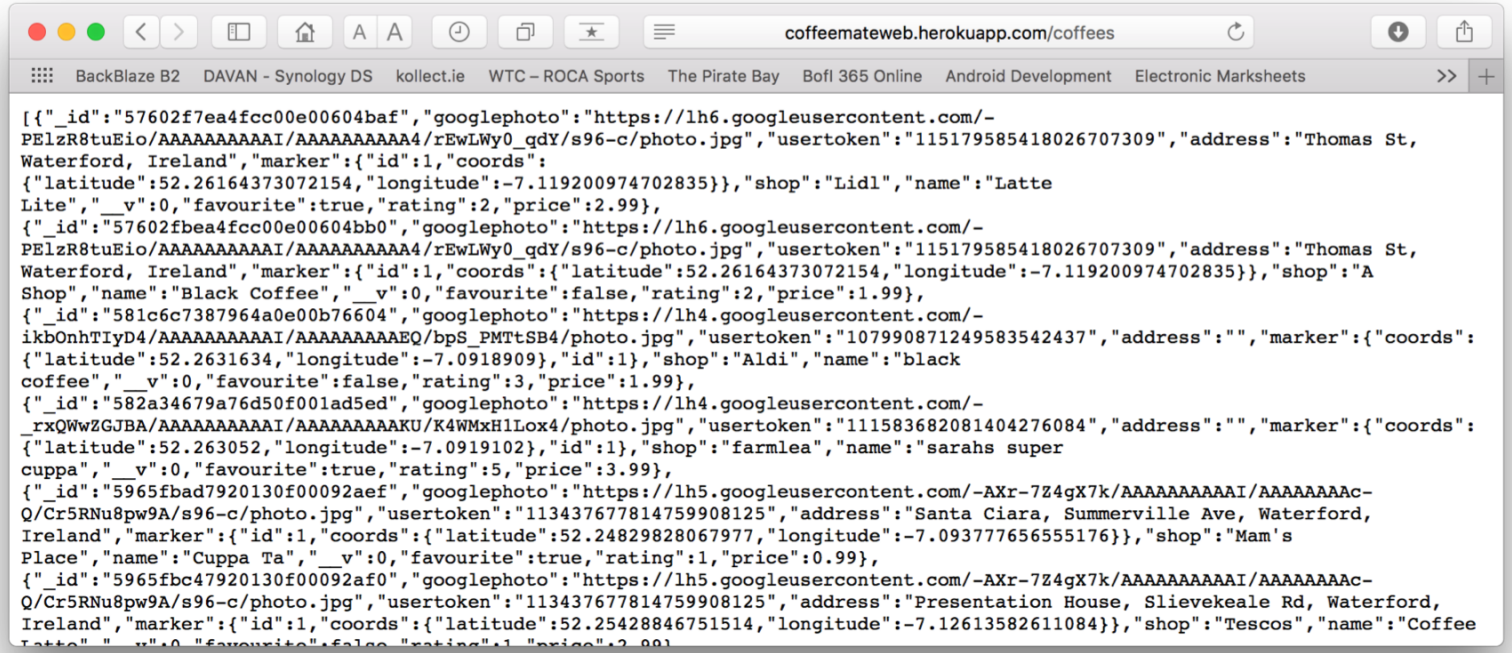
Variable Names

- In the previous examples, we assumed that there was an exact mapping of instance variable names between the Coffee class and the server. This will often not be the case, especially if your server uses a different spelling convention than your Android app.
- For example, if you use a Rails server, you will probably be returning data using `snake_case`, while your Java probably uses `camelCase`. If we add a `dateCreated` to the Coffee class, we may be receiving it from the server as `date_created`.
- To create this mapping, use the @SerializedName annotation on the instance variable. For example:

```
@SerializedName("date_created")
private Date dateCreated;
```

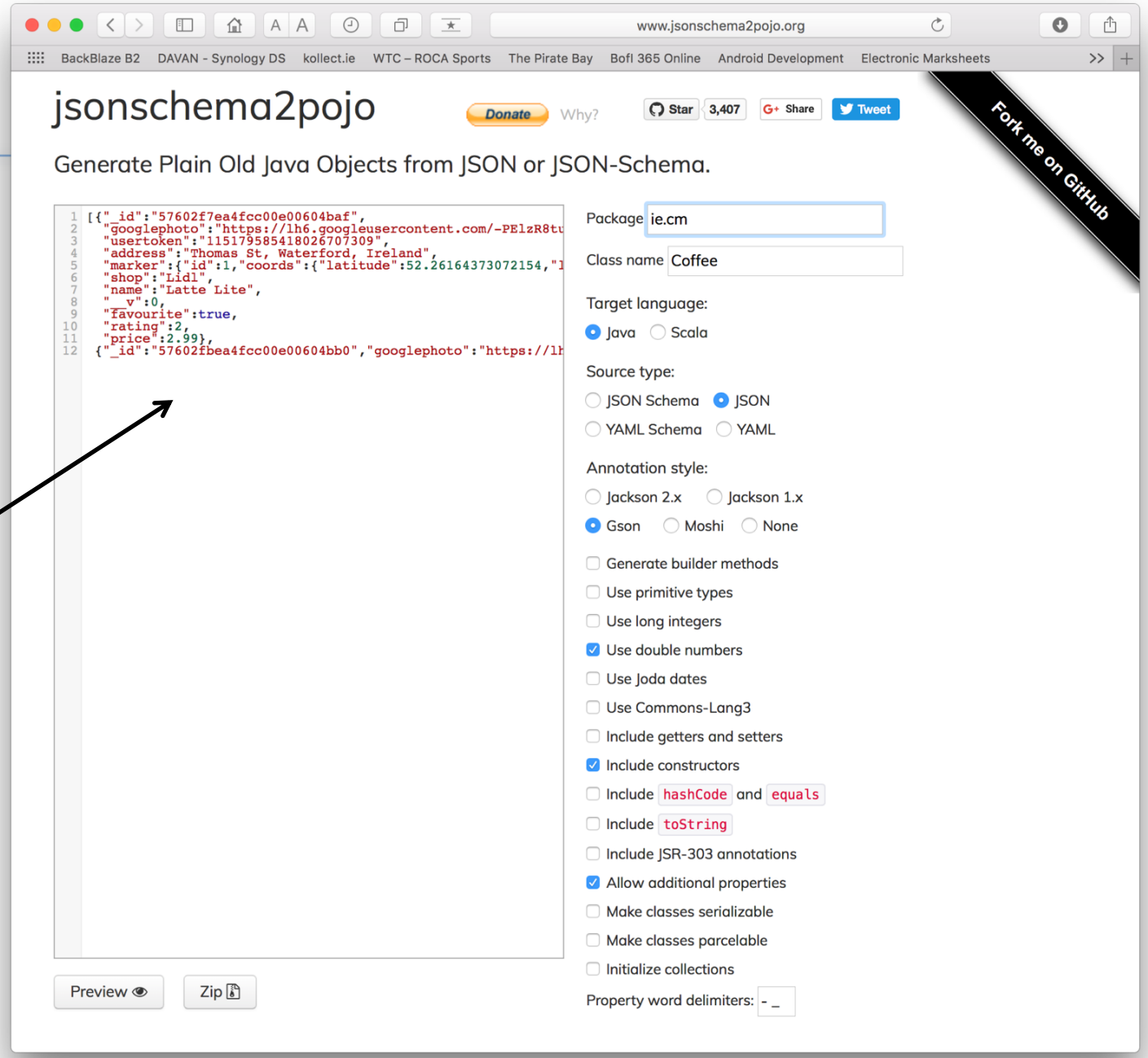
Bridging the Gap Between Your Code & Your API

- You can also create your models automatically from your JSON response data by leveraging a very useful tool: jsonschema2pojo - http://www.jsonschema2pojo.org
- Grab your JSON string, visit the above link and paste it in, like so



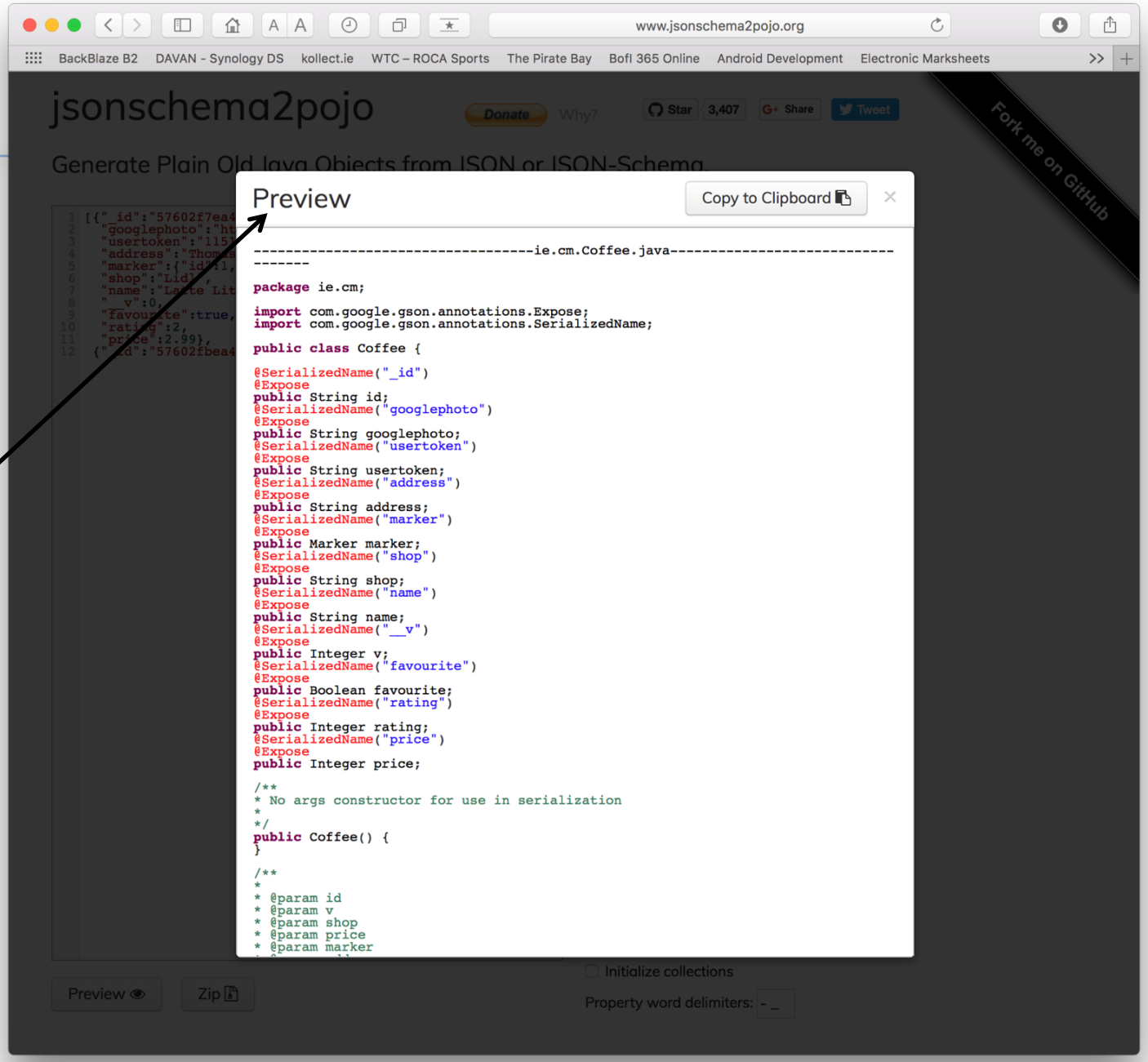
Bridging the Gap Between Your Code & Your API

Your JSON goes here



Bridging the Gap Between Your Code & Your API

Your annotated class



Bridging the Gap Between Your Code & Your API

Date Formats

- Another potential disconnect between the app and the server is the way they represent date objects.
- For instance, Rails may send a date to your app in the format "`yyyy-MM-dd'T'HH:mm:ss`" which Gson will not be able to convert to a Java **Date**. We have to explicitly tell the converter how to perform this conversion. In this case, we can alter the Gson builder call to look like this:

```
Gson gson = new GsonBuilder()
                .setDateFormat("yyyy-MM-dd'T'HH:mm:ss")
                .create();
```

Additional Info - Headers

- If you wish to add headers to your calls, you can annotate your method or arguments to indicate this.
- For instance, if we wanted to add a content type and a specific user's authentication token, we could do something like this:

```
@POST("/coffees")
@Headers({"Content-Type: application/json"})
Call<Coffee> createCoffee(@Body Coffee coffee,
                          @Header("Authorization") String token);
```

Full List of Retrofit Annotations

- ❑ @POST
- ❑ @PUT
- ❑ @GET
- ❑ @DELETE
- ❑ @Header
- ❑ @PATCH
- ❑ @Path
- ❑ @Query
- ❑ @Body

References

- http://square.github.io/retrofit/
- https://spin.atomicobject.com/2017/01/03/android-rest-calls-retrofit2/
- https://code.tutsplus.com/tutorials/getting-started-with-retrofit-2--cms-27792


Thanks!