# CS 370 Lab 4: RecyclerView

This lab is about displaying a list of items, in a memory-efficient manner.

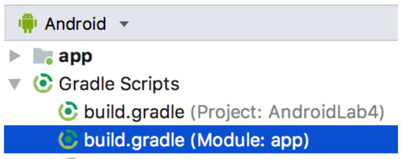
**Obtain Code**

1. Ensure that the lab workstation is booted into OS X
2. Create a new folder on your desktop called Repositories
3. Open a terminal session (Applications -> Utilities -> Terminal)
    - This is just a terminal window! You **don't** need to log in to blue
4. Ensure there are no other default accounts in the OSX keychain
    a. Keychain management instructions: https://kb.wisc.edu/helpdesk/page.php?id=2197
    b. Search for any *github.com* entries and remove them
5. Using the command line, change directory to the Repositories folder you just created
6. Clone the repository: **git clone https://github.com/SSU-CS370-F18/AndroidLab4.git**
7. Change directory to the AndroidLab4 folder you just cloned
8. Branch the repository using a branch name of **lastnamefirstname-370H4**
9. Open Android Studio, and open Lab 4

**Part 1: Parsing with Gson**

Gson is Google's JSON parsing library. If your models are configured in a way that matches the JSON you are parsing, Gson can automatically create your models and fill them with data.

1. Look in the Gradle Scripts directory, open the app build.gradle file


Android
app
Gradle Scripts
build.gradle (Project: AndroidLab4)
build.gradle (Module: app)

2. The *dependencies { }* block is where all external libraries our app needs are listed.
    a. Uncomment the *gson* dependency, and rebuild the project. This will download the Gson files from the gradle server, and make it available for use in the app.
3. To use Gson, we need to make sure our Java models match the response structure.
4. Look in the <u>model</u> directory. You should see *RecipeModel* and *RecipeResponse*.

5. Open *RecipeModel*. Compare the class variables to the fields in the JSON sample below
    a. the variable names match the JSON field names (the keys)
    b. the variable datatypes match the JSON field types (the values)
        i. String for values in "quotes"
        ii. int for numeric values
        iii. List<String> for a list of [ "strings" ]
    c.
6. Open *RecipeResponse*.
    a. there is only one class variable – *recipes*. Notice that there's no corresponding field name in the JSON sample.
7. Add the *SerializedName* annotation to the *recipes* variable. In the JSON sample, find the field name we want to use to get the list of RecipeModels and use that here.

```
@SerializedName("json_key_name")
private List<RecipeModel> recipes;
```

This annotation signals to Gson that the variable has a different Java name than what can be found in JSON. Without this, it would search for a *recipes* field in the JSON (and fail).

Now that the models are ready, we can use Gson to fill them with data.

8. Open *RecipeSearchAsyncTask*
9. This is almost the same as Lab 3, but notice the three differences:
    a. the HttpUrl and Request builders use the chain method style
    b. there's a new query parameter: *maxResult*. We'll get more results with this.
    c. the `*RESULT*` field of the AsyncTask is slightly different – it's a *List*
10. Open *RecipeParser*. The return type is changed here as well, to a *List* of *RecipeModel*
11. This is where we use Gson to parse our response
    a. Create a new instance of Gson
    b. Parse the response
    c. Get the list of recipe models from the response model
    d. Return the list

```
Gson gson = new Gson();
RecipeResponse response = gson.fromJson(json, RecipeResponse.class);
List<RecipeModel> recipes = response.getRecipes();
```

## Part 2: Recycling a List with RecyclerView

When you have more objects than will fit on the screen, you need a list view.

When you have a lot of objects, you need to manage your memory.

A RecyclerView lets you display large amounts of data without using memory on objects that aren't being displayed. There are three parts to a simple RecyclerView:

- The RecyclerView

  The overall container for the list. This goes in the layout file. Empty to start with, and will fill itself with views full of data that you provide.
- The ViewHolder

  Each ViewHolder is in charge of displaying a single item with a view. The RecyclerView creates only as many ViewHolders as are needed to display the on-screen portion of the content, plus a few extra. As the list scrolls, the off-screen ViewHolders get rebinded with the data which is scrolling onto the screen.
- The Adapter

  The ViewHolder objects are managed by an adapter, which creates them as needed. The adapter also binds the ViewHolders to their data. It uses the ViewHolder's position to determine what the contents should be, based on its list position.

12. Open *res/layout/recycler_list_item.xml*. This is the layout that we will use for each entry in the list. There's two TextViews here: a name and a rating.
13. Open *RecipeViewHolder*. This class represents one entry in the list. It has two TextView variables that will connect to the TextViews in *recycler_list_item* layout.
14. In the constructor, after calling the super class, bind the layout TextViews to the variables.
    a. Since this class is not a subclass of Activity, we have to use the View that was passed as an argument to access the *findViewById* function:
    ```
    itemView.findViewById(…);
    ```
15. The *bindView* method will get called when we attach data to this ViewHolder. This method is where we update the views with new data.
    a. Get the *recipeName* from the model and set it on the *itemNameTextView*
    b. Get the *rating* from the model and set it on the *itemRatingTextView*

    ```
    String.format("Rating: %i / 5", variable)  // (int)variable replaces %i
    ```

16. Open *RecipeViewAdapter*. This class controls the creation and reassignment of ViewHolders. Notice the member variable storing a List of RecipeModels – this is all the data for the entire list. The RecyclerView only creates enough ViewHolders for what can be seen on screen, and will reuse them with different data as the user scrolls through the list.
17. Look at the *onCreateViewHolder* method. This is called to create a new ViewHolder. Here we take the *recycler_list_item* layout and inflate it (create objects from the design), then create and return a ViewHolder.
18. The *onBindViewHolder* method is called when we need to change the data in a particular ViewHolder. The *position* argument tells us which data to put in the ViewHolder.
    a. Get the model at the specified position in the list
    b. Call the RecipeViewHolder.bindView method to

    ```
    RecipeModel model = recipeCollection.get(position);
    recipeViewHolder.bindView(model);
    ```
19. The *getItemCount* function is called to determine the size of the list. Finish this by returning the number of items in the *recipeCollection*
    a. in Java, use *listname.size()* to get the length of a list
20. Open *RecyclerViewActivity*. The last thing we need to do is setup the RecyclerView and connect the adapter to it.
21. Find these two lines of code:

    ```
    RecyclerView.LayoutManager layoutManager = new LinearLayoutManager(getBaseContext());
    recyclerView.setLayoutManager(layoutManager);
    ```

    These lines create a new LinearLayoutManager that will simply order items top-to-bottom.
22. In the *RecipeCallbackListener* callback (defined in the button clicklistener), we are no longer updating a single TextView with one model's information – we will fill the RecyclerView with data and it will take care of displaying it for us (using the adapter and view holder).
    a. Create a new *RecipeViewAdapter* with the list of items we get in the callback.
    b. Pass the adapter to the RecyclerView.

    ```
    RecipeViewAdapter adapter = new RecipeViewAdapter(models);
    recyclerView.setAdapter(adapter);
    ```
23. If your app is running correctly, you should be able to enter a search term, click the button, and see a list of results displayed. If not, revisit the above steps and make sure you've followed all the instructions.

24. When your app runs properly, execute the following commands to commit your changes to your branch:
    a. git add .
    b. git commit -m “working code complete”
    c. git push origin yourbranchname

    Your code branch is now saved and committed to the git repository.

This completes Lab 4.

For bonus points…

**Image Loading**

Use a library to asynchronously load images into your app.

In the lab, we get the url for an image associated with a recipe. In order to get the actual image, we would have to execute another asynchronous call on another background thread to download the image, and then update an ImageView on the Main/UI thread. This is much more painful to do by hand. We will use a library instead.

1. Look up an image-loading library for Android
    a. Glide and Picasso are common ones
2. Add it to the app/build.gradle file, in the *dependencies* segment

```
// example: Glide or Picasso. you only need one, and doesn’t have to be these
implementation "com.github.bumptech.glide:glide:4.8.0"
implementation "com.squareup.picasso:picasso:2.71828"
```

3. Change the *recycler_list_item* layout to include an ImageView. Make sure it has an *id*. Position it so the image is on the left, and the name and rating are on the right stacked vertically (like they are initially).
4. Add code to the RecyclerViewHolder’s *bindView* method to start loading the image, using the first link in smallRecipeUrls for each RecipeModel.
5. Commit and push your code. Use “bonus – image loading” as your commit message