



Lab 7 – App Structure

Exploring App Structure and Navigation

*Today, we're going to embark on a **mouth-watering** journey into app development, as we create a **delightful** Bakery app with a variety of Sweet Treats. Don't get too hungry as we concoct a scrumptious Bakery app showcasing a tempting array of sweet items such as cakes... pies... mMMmmmm it's making me hungry just thinking about it... What was I saying? Oh, that's right. After today's lab, we will have whipped up an app with a scrumptious user experience that will leave users hungry for more...*

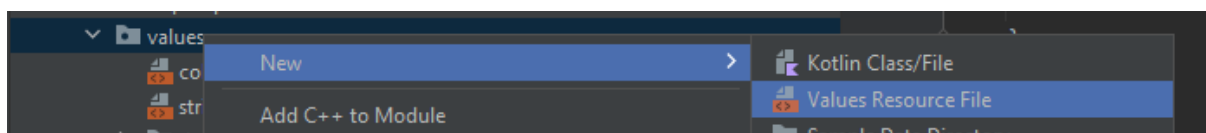
Alright all jokes aside, today we're going to be challenging the skills we've learned so far in order to create a multi-activity app. This app will pull resources from a string-array in order to create a ListView, explain how we can use the ArrayAdapter as a bridge between different datasources, and finally let us have different content load dynamically. This will let us create a Bakery app, where we can browse different categories to see all the different products on offer. Let's get started!

App making stuff, I guess

Let's first work our way through creating a basic app that can pull items from lists – similar to what we have in previous sessions.

Go ahead and create a new app following the settings we've used so far, matching the name of the lab at the top of the page. Remember to target API 26.

As we're making a Bakery, we need a bunch of categories of products our Bakery might sell. We're going to create a string resource called **categories.xml** to hold this information like we did a couple of labs ago:



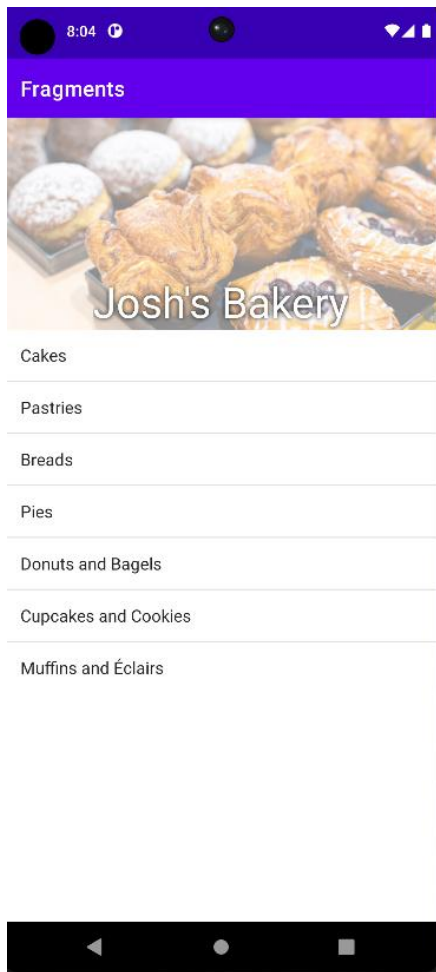
For this to work, we'll have all of our categories held within a **string-array** called products with each item using the **item** tag:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string-array name="products">
        <item>Chocolate Cake</item>
    </string-array>
</resources>
```

Now **add in some categories** of your choosing that the Bakery will sell products of. You shouldn't need too many (it'll add more work the more you add, so maybe stick to no more than **3 categories**), but here's mine to give you some ideas:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string-array name="product_categories">
        <item>Cakes</item>
        <item>Pastries</item>
        <item>Breads</item>
        <item>Pies</item>
        <item>Donuts and Bagels</item>
        <item>Cupcakes and Cookies</item>
        <item>Muffins and Éclairs</item>
    </string-array>
</resources>
```

Now let's set up our **activity_main.xml** layout. We'll be using **LinearLayout** as our base layout, along with changing our app to look a bit like the following:



In order to accomplish this, I've used:

- A **FrameLayout** that holds an **ImageView** and a **TextView** component – this allows me to have my text and image overlayed on top of each other.
- The **ImageView** has a maximum height of 200dp, with `scaleType` set to `centerCrop` so it always floats at the top of the screen
- The **TextView** has `textAppearance` set to `@style/TextAppearance.AppCompat.Large` so it appears in larger text
- Outside of the **FrameLayout**, I've got a **ListView** that fills the rest of the screen. It uses the **entries** attribute to pull all of the products I made earlier.

Try your best to do this yourself, and then review my code on the next page before continuing:

activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    tools:context=".MainActivity">

    <FrameLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content">

        <ImageView
            android:layout_width="match_parent"
            android:layout_height="200dp"
            android:layout_gravity="center_horizontal"
            android:scaleType="centerCrop"
            android:src="@drawable/bakery" />
        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:textColor="@color/white"
            android:shadowRadius="10"
            android:shadowColor="@color/black"
            android:layout_gravity="bottom|center_horizontal"
            android:textSize="40sp"

            android:textAppearance="@style/TextAppearance.AppCompat.Large"
            android:text="@string/bakery_title" />

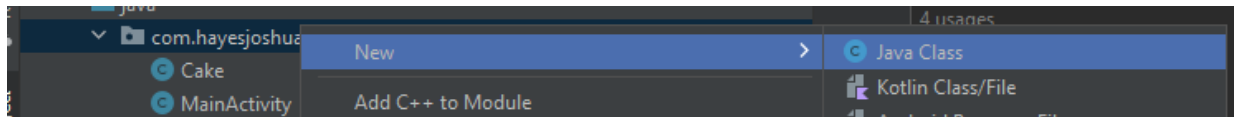
    </FrameLayout>

    <ListView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:id="@+id/list_options"
        android:entries="@array/product_categories" />

</LinearLayout>
```

Now we need a new Java class that is going to hold the information about different products that we hold in our store. For now, I'm going to start with one that will hold information about the **Cake** category – but feel free to substitute that name for another category.

Unlike every time we've created a new class in the past, this time we aren't going to create an activity with it. Right-click on the folder with your **app package name** on the left and select **New > Java Class**. I'm calling mine **Cake**, then hit enter.



In this new class, go ahead and add a **private string variable** for a name and description, as well as a **private integer** to reference an image ID:

```
public class Cake {  
    private String name;  
    private String description;  
    private int imageResource;  
}
```

Now add an array called **cakes** that will hold many **Cake** objects. We want this to be the only instance of this variable, and not be able to be changed after launch – so set it to **public static final**.

```
public static final Cake[] cakes;
```

Let's now add a constructor that will take in a cake name, description and image ID and set our variables we made above.

```
private Cake(String name, String description, int  
imageResource){  
    this.name = name;  
    this.description = description;  
    this.imageResource = imageResource;  
}
```

Add in some products that match your category and assign it to your array where you've declared it. You'll need a name, a description, and an image for each to attach to it. Here's what mine looks like in the cake category:

```
public static final Cake[] cakes = {  
    new Cake( "Sponge", "Sponge cake is a light cake made with  
eggs, flour and sugar, sometimes leavened with baking powder.",  
R.drawable.sponge ),
```

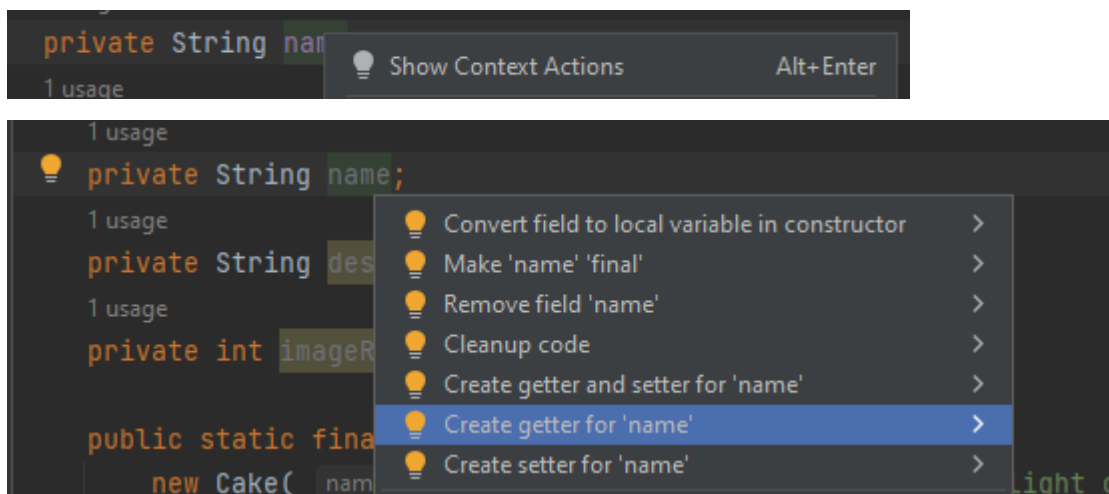
```

        new Cake( "Red Velvet", "Red velvet cake is a delicacy
        originating from the Victorian Era.", R.drawable.red_velvet ),
        new Cake( "Carrot Cake", "Carrot cake is cake that contains
        carrots mixed into the batter.", R.drawable.carrot )
    };

```

We want our name, description and image resource to all be private variables, but still readable outside of our class. We don't want other classes to be able to modify our variables though. In Java, we can do this using a dedicated **get()** method for our variables. In most languages, this concept is referred to as **Encapsulation**.

Add a **get()** method for our name, description and image resource variables, either by typing them out or right-clicking, selecting **Show Context Actions** > **Create getter for 'variable'**



```

public String getName() {
    return name;
}

public String getDescription() {
    return description;
}

public int getImageResource() {
    return imageResource;
}

```

Finally, add a **toString()** override method that returns our current name:

```

@NonNull
@Override
public String toString() {
    return this.name;
}

```

```
}
```

That's all we need for this file for now.

Head to **MainActivity.java** – we're going to be using something called a **AdapterView**.

An **AdapterView** is a special type of view that is used to display a collection of data items. We can use the AdapterView with either the ListView, GridView (not covered yet), and Spinner (covered in lab 5). These views are designed to efficiently display and manage large sets of data while providing a clean and intuitive user interface.

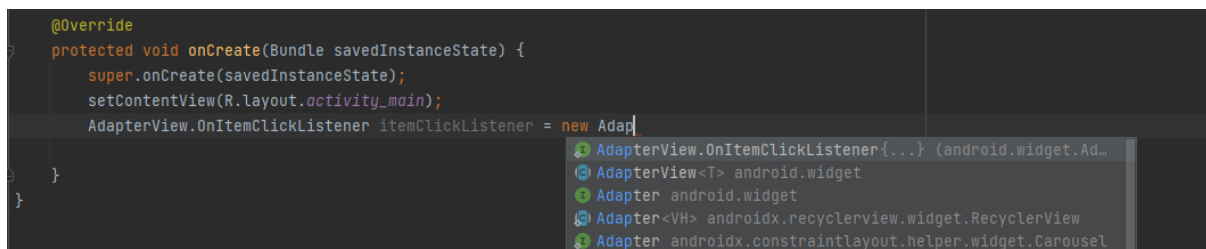
Think of an AdapterView as a container that automatically creates and manages child views (items) based on the data it receives. The data is usually provided to the AdapterView using an Adapter, which acts as a bridge between the data source and the AdapterView itself.

The Adapter's main responsibility is to convert each data item into a corresponding view that can be displayed within the AdapterView. This process is often referred to as "binding" the data to the views. The AdapterView will then handle user interactions, such as scrolling or item selection, and automatically update the displayed views as needed.

To get started, add an import statement for an **AdapterView**

```
import android.widget.AdapterView;
```

Within our **onCreate** method, add the following code and hit enter – to generate an **onItemClick** method within our adapter view:



```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    AdapterView.OnItemClickListener itemClickListener = new Adap
```

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    AdapterView.OnItemClickListener itemClickListener = new
    AdapterView.OnItemClickListener() {
        @Override
        public void onItemClick(AdapterView<?> adapterView, View
        view, int i, long l) {

        }
    };
}
```

Modify the name of the variables coming into our **onItemClick** method to a **position** and an **id**, so they make more sense later on:

```
public void onItemClick(AdapterView<?> adapterView, View view,
int position, long id)
```

Within this method, we want our code to run an intent depending on the position we are in our adapter view. This position will be determined depending on where our item is within the array we made earlier.

Within my categories.xml file, Cakes is the first item in the list (arrays are indexed at 0), so I will put in an if() statement that checks if I'm at position 0 – if I am, it will create an intent and run it.

```
@Override
public void onItemClick(AdapterView<?> adapterView, View view,
int position, long id) {
    if(position == 0){
        Intent intent = new Intent(MainActivity.this,
CakeCategoryActivity.class);
        startActivity(intent);
    }
}
```

You'll get an error saying that **CakeCategoryActivity** doesn't exist, don't worry – we'll make that in a second. Also make sure you add in an import statement for your intent:

```
import android.content.Intent;
```

At the bottom of our **onCreate** method, all that is left to do is get a reference to our ListView we made earlier and set the onItemClickListener. This is how your MainActivity should look:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    AdapterView.OnItemClickListener itemClickListener = new
AdapterView.OnItemClickListener() {
    @Override
    public void onItemClick(AdapterView<?> adapterView, View
view, int position, long id) {
        if(position == 0){
```



```

        Intent intent = new Intent(MainActivity.this,
CakeCategoryActivity.class);
        startActivity(intent);
    }
}
};
ListView listView = findViewById(R.id.list_options);
listView.setOnItemClickListener(itemClickListener);
}

```

With our code completed, our **OnItemClickListener** that is assigned to our ListView will run whenever we select an item in our list. Let's go ahead and create that activity we mentioned earlier:

Create a new **Empty Activity** called **CakeCategoryActivity** and modify it to look something like the following:

```

<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    tools:context=".CakeCategoryActivity">
    <ListView
        android:id="@+id/list_cakes"
        android:layout_width="match_parent"
        android:layout_height="wrap_content" />
</LinearLayout>

```

Notice how our ListView here doesn't have an entries attribute – this is because we're going to dynamically fetch these entries that we want to display.

To link our list of cakes to our ListView, we need to use the **ArrayAdapter** again. Remember, Adapters act as a bridge between our data sources and our ListView.

Update the **onCreate** method within **CakeCategoryActivity** to include the following line:

```

setContentView(R.layout.activity_cake_category);
ArrayAdapter<Cake> listAdapter = new ArrayAdapter<>(this,
    android.R.layout.simple_list_item_1, Cake.cakes);

```

This array adaptor gets a reference to our current Activity, a reference to our Cake object and also a reference to **simple_list_item_1**.

In this use case, **simple_list_item_1** is a predefined Android layout resource that represents a simple, single-line list item view. It is used as the layout for each item in the ArrayAdapter, which is responsible for creating and managing the views for the individual items within the AdapterView (our ListView).

When you create an ArrayAdapter with the android.R.layout.simple_list_item_1 layout, you are telling the ArrayAdapter to use the built-in Android layout for each item in the list. This layout consists of a single TextView with some default styling.

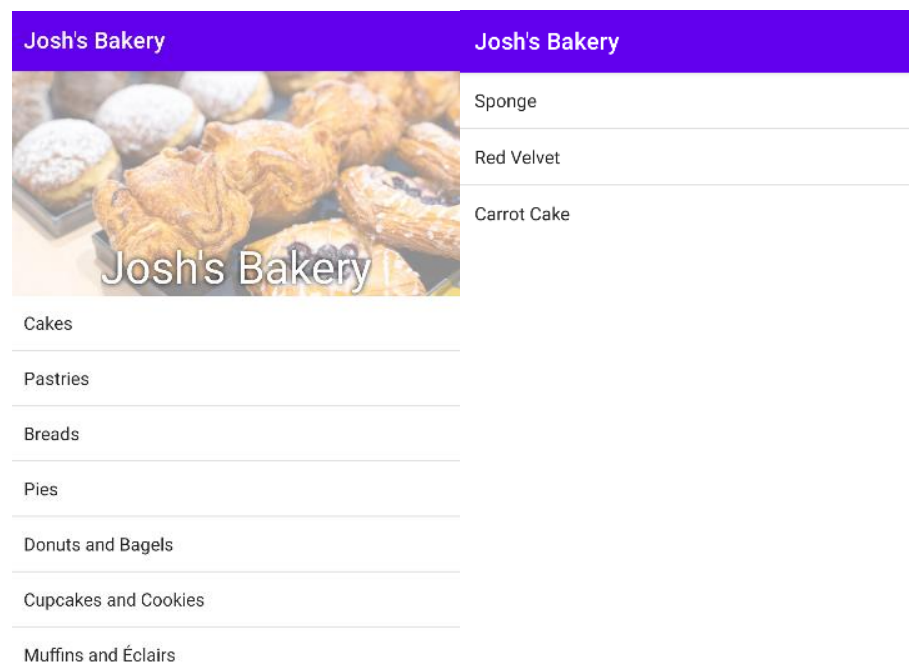
The ArrayAdapter will use this layout to create a view for each Cake object in the Cake.cakes array. It will automatically set the text of the TextView to the string representation of the Cake object, which we've done by calling the toString() method on the Cake object.

If that doesn't make sense – don't worry, you'll see it in action in a second.

Add in the **following two lines** to finally get a reference to the list we just made, and set our adaptor to the one we just made:

```
ListView listCakes = findViewById(R.id.list_cakes);  
listCakes.setAdapter(listAdapter);
```

That's our code done. Let's **test our app** by running it and clicking on Cakes.



We should now get a list of all our cakes. Now we just need a details activity that can show the name of our cake, the picture of the cake and finally it's description. For this to work, we

need to have an `onClick` listener on `Cake` in a similar way to how we did it earlier. Update your `onCreate` method within our **CakeCategoryActivity** to look a bit like the following:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_cake_category);
    ArrayAdapter<Cake> listAdapter = new ArrayAdapter<>(this,
    android.R.layout.simple_list_item_1, Cake.cakes);
    ListView listCakes = findViewById(R.id.list_cakes);
    listCakes.setAdapter(listAdapter);

    AdapterView.OnItemClickListener itemClickListener = new
    AdapterView.OnItemClickListener() {
        @Override
        public void onItemClick(AdapterView<?> adapterView, View
    view, int position, long id) {
            Intent intent = new
    Intent(CakeCategoryActivity.this, CakeActivity.class);
            intent.putExtra(CakeActivity.EXTRA_CAKEID, (int)
    id);
            startActivity(intent);
        }
    };
    listCakes.setOnItemClickListener(itemClickListener);
}
```

Don't forget the import statement for our Intents – we'll now create the **CakeActivity** and update its layout to look a bit like this (feel free to tweak it how you'd like it to look):

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    tools:context=".CakeActivity">
    <FrameLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content">
        <ImageView
```

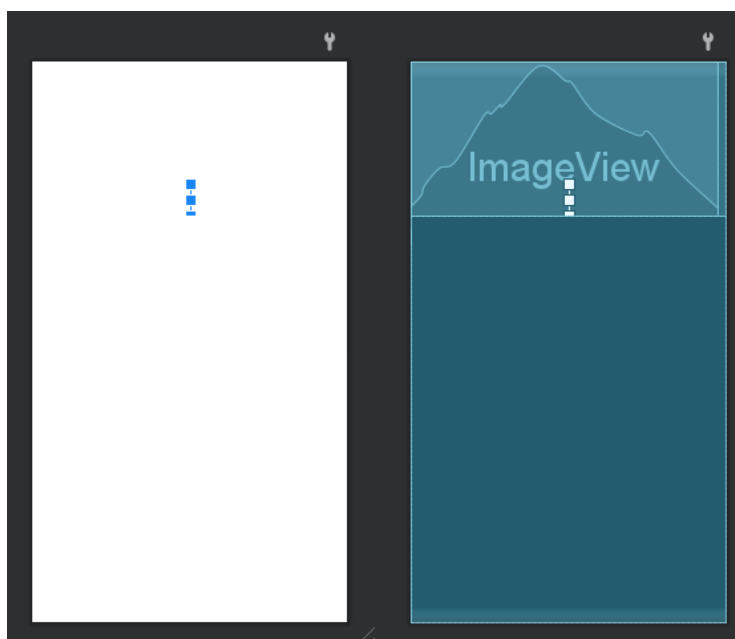
```

        android:id="@+id/image_cake"
        android:layout_width="400dp"
        android:layout_height="200dp"
        android:scaleType="centerCrop" />
    <TextView
        android:id="@+id/text_name"
        android:textSize="30sp"
        android:textColor="#FFFFFF"
        android:shadowColor="#000000"
        android:shadowRadius="1"
        android:shadowDx="2"
        android:shadowDy="2"

        android:textAppearance="@style/TextAppearance.AppCompat.Large"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="bottom|center_horizontal" />
</FrameLayout>
<TextView
    android:id="@+id/text_description"
    android:layout_marginTop="20dp"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />
</LinearLayout>

```

You won't see too much in our layouts, as we will be updating our code to dynamically populate these views



Update our **CakeActivity.java** file to include a **static final String** like the following:

```
public static final String EXTRA_CAKEID = "cakeID";
```

After our code within onCreate, we need to get a reference to the intent that was passed to this class, along with the id we passed into it.

```
Intent intent = getIntent();  
int id = intent.getIntExtra(EXTRA_CAKEID, 0);
```

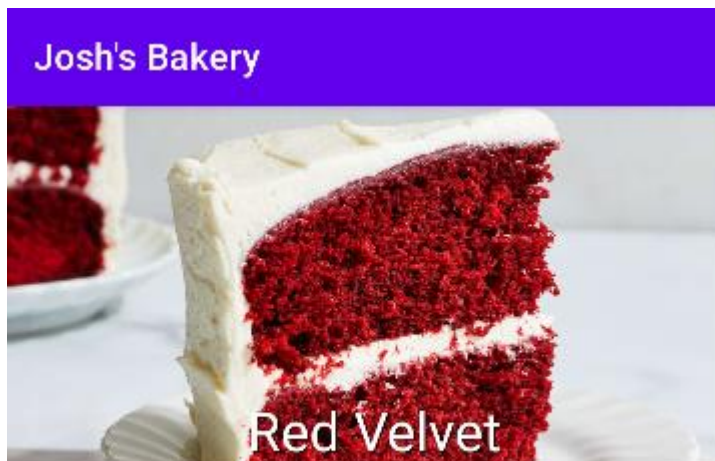
With this ID, we can now get a reference to the Cake in question from our Cake array:

```
Cake cake = Cake.cakes[id];
```

Now we need a reference to the objects in our layout so we can set them to the values in our cake object:

```
ImageView imageView = findViewById( R.id.image_cake );  
TextView name = findViewById( R.id.text_name );  
TextView description = findViewById( R.id.text_description );  
imageView.setImageResource( cake.getImageResource() );  
name.setText( cake.getName() );  
description.setText( cake.getDescription() );
```

Now if we run our app, we should be able to see the details of each cake:



Red velvet cake is a delicacy originating from the Victorian Era.

Challenge

It's now time to implement a category and details view for your other categories using everything we've learned so far. Here's a checklist to help you keep on track.

- First create a Vanilla Java class to contain information on an item
- ListView's onItemClick() in MainActivity starts the activity
- Make a new category activity with a ListView and use an ArrayAdapter to link the items array to the ListView
- Make OnItemClickListener for the ListView in the category activity that starts the detail activity
- Make the detail activity that fetches data from the array
- Update views on the detail activity

If you get stuck, just scroll up and follow the instructions you followed for the Cake. It should be almost identical.

Well done, you've completed the lab! In our next lab, we'll be looking at a concept called **Fragments** that allow us to dynamically create interfaces. You are more than welcome to spend some time over the break researching this topic, we'll touch on it when we return!

Submission

When submitting your lab file, ensure that you have put the entire contents of your app's folder inside a **.zip** file. This will be in the location that you specified at the beginning of today's lab. It should be called **App Structure**.

Now rename your **.zip** file to

[YourStudentID]_IT617_Lab7.zip

We'll be using this naming convention for all of our labs moving forward.

Finally, upload this **.zip** to Blackboard.

If you have any issues getting a lab completed or have been away sick for a length of time, please ensure that you send me an email to explain the situation before the due date. If you do not email me, you may receive a 0.