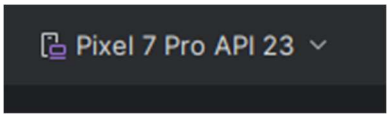
Android Development

Version

We are developing on android marshmallow. For the application to run locally the device must be running API 23. While this is an older version of android, newer APIs block requests and responses from addresses that are not secured with HTTPS. Since we do not have an SSL certificate, we can only send requests through HTTP

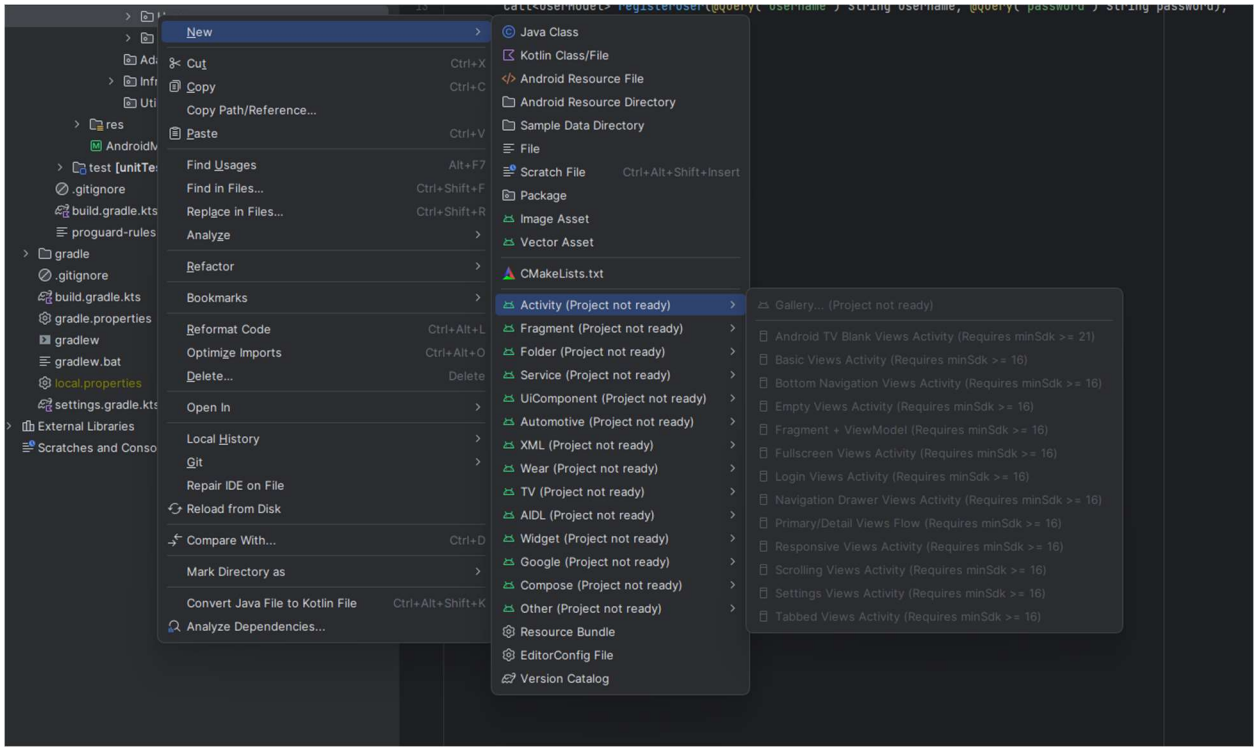


Development

Newer versions of android use a coding language called Kotlin, however we are all most familiar with Java, so we are using that. If you are trying to find a solution to a problem online, be sure that you are implementing a java solution, not a Kotlin one.

Activities

Each screen in android is called an “Activity”. You can create an activity by right clicking the package you want to add it to, selecting New -> Activity -> and selecting one of the options. My favorite option is Empty Views Activity.



Upon creation of an activity, a java class will be created that outlines the logic and behavior of the screen, and an XML file will be created in app/src/main/res/layout that outlines the appearance of the screen.

Layouts

Layouts are written in XML, which has a similar structure to HTML. Each component is called a tag, and each individual component on a screen is typically one tag.

There are layout tags that act as containers for more functional tags such as buttons. Each Layout file must have a root layout tag. A common layout tag to use as the root is Linear Layout. Tags have modifiers called attributes that have a wide variety of uses such as specifying the size or color of a component.

```
<?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"
        tools:context=".Activities.LoginRegister.LoginActivity"
        android:orientation="vertical">
```

This is the root tag for the login page. The tools:context attribute is used to bind the XML file to its corresponding java class. If a tag ends with an angle bracket, then it is considered open, and must be closed. This is done by either adding a / before the bracket, or closing tag. The closing tag here would be <LinearLayout/>.

Any additional tags, such as buttons, would be located between the opening and closing tags.

Some useful layouts are LinearLayouts, which arrange components either horizontally or vertically, FrameLayouts which act as basic containers with minimal functionality, and GridLayouts, which allow you to specify a number of rows and columns that can then be populated with UI elements.

```
<!--     EVENT TIME SELECTION     -->
<GridLayout
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:rowCount="1"
    android:columnCount="2"
    android:layout_gravity="center"
    android:paddingTop="10dp">

    <!--     START TIME     -->
    <LinearLayout
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:orientation="horizontal"
        android:layout_column="0"
        android:layout_row="0"
        android:id="@+id/pick_start_time">
        <LinearLayout
```

This is a grid layout with 1 row and 2 columns. It contains a linear layout in cell 0,0 as specified in the layout_row and layout_column attributes.

UI Elements

Like layouts, UI elements are defined using tags. Some useful UI elements are buttons, spinners (dropdown menus), ImageViews (which can display pictures), TextViews, EditText (allow a user to input text, and ListViews (can display collections of data).

```
    <ListView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:id="@+id/eventListView"
        android:isScrollContainer="true"/>
    <TextView
        android:layout_width="match_parent"
        android:layout_height="0dip"
        android:layout_weight="0.92"
        android:layout_gravity="center"
        android:gravity="center"
        android:textSize="30sp"
        android:id="@+id/list_empty"
        android:visibility="gone"
        android:text="There are no events scheduled."/>
```

One attribute that is consistent across all tags is android:id. Giving a tag an id allows the java class to interact with that UI element.

The appearance of basic UI elements can be heavily modified by defining custom drawable files for them. This works like a CSS file in HTML.

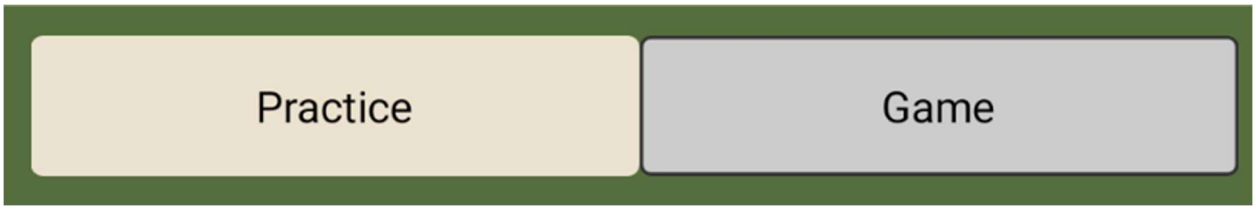
```xml
<?xml version="1.0" encoding="utf-8"?>
<selector xmlns:android="http://schemas.android.com/apk/res/android" >

    <item android:state_pressed="true" >
        <shape android:shape="rectangle" >
            <corners android:radius="3dip" />
            <stroke android:width="1dip" android:color="#333333" />
            <solid android:color="#cccccc" />
        </shape>
    </item>

    <item android:state_checked="true">
        <shape android:shape="rectangle" >
            <corners android:radius="3dip" />
            <stroke android:width="1dip" android:color="#333333" />
            <solid android:color="#cccccc" />
        </shape>
    </item>

    <item>
        <shape android:shape="rectangle"  >
            <corners android:radius="3dip" />
            <stroke android:width="1dip" android:color="@color/calendar_light_colour" />
            <solid android:color="@color/calendar_light_colour" />
        </shape>
    </item>
</selector>
```

This file changes a radio button from looking like a list of options with check boxes to something that looks like a regular set of square buttons.




```
<RadioGroup
    android:id="@+id/event_type"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:gravity="center_horizontal"
    android:orientation="horizontal"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent">
    <RadioButton
        android:id="@+id/practice"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:background="@drawable/drawable_radio_button"
        android:button="@android:color/transparent"
        android:checked="true"
        android:text="Practice"
        android:layout_weight="0.5"
        android:textAlignment="center"/>
    <RadioButton
        android:id="@+id/game"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:background="@drawable/drawable_radio_button"
        android:button="@android:color/transparent"
        android:checked="true"
        android:text="Game"
        android:textAlignment="center"
        android:layout_weight="0.5"/>
</RadioGroup>
```

Adapters

Adapters can be used to tell the application how to display java objects on a screen. For example, if we have a list of User data models, and we want to display their userIds and names, we can create an adapter that binds those values in the data model to some UI element.

```xml
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="horizontal"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:clickable="false">

    <!--     PLAYER NUMBER     -->
    <TextView
        android:layout_width="0dip"
        android:layout_height="wrap_content"
        android:layout_weight="1"
        android:paddingBottom="15dp"
        android:paddingTop="15dp"
        android:paddingStart="20dp"
        android:focusable="false"
        android:focusableInTouchMode="false"
        android:textSize="25sp"
        android:id="@+id/playerNumber"/>

    <!--    PLAYER FULL NAME     -->
    <TextView
        android:layout_width="0dip"
        android:layout_height="wrap_content"
        android:layout_weight="3"
        android:paddingBottom="15dp"
        android:paddingTop="15dp"
        android:textSize="25sp"
        android:focusable="false"
        android:focusableInTouchMode="false"
        android:id="@+id/playerFullName"/>
</LinearLayout>
```

This XML represents one row in a ListView. There is java code that will be covered later that binds these two UI elements to a value in a data model.

Fragments

A fragment is similar to an activity, but it can be implemented as a UI element, rather than an entire screen. This means that fragments can be reused across multiple activities. This is useful if you

have multiple screens that exist in different contexts, but they all display similar things. Some apps are entirely made of fragments, and use code to switch between them in a single main activity.

A fragment can be created from the same menu as an activity, just selecting fragment instead of activity.

A fragment is implemented as a single tag, and the layout attribute points it to the correct layout for the fragment. The fragment has its own layout file that works in the exact same way as an activity, and also has a java class.

```
<fragment
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:name="com.cosc3p97.coachwizard.Activities.Calender.DaySchedule"
    tools:layout="@layout/fragment_day_schedule"
    android:id="@+id/day_schedule" />
```

Activity Classes

Each layout has a tag that binds it to a java class. This class represents all of the behaviour of the UI.

```
public class LoginActivity extends AppCompatActivity {
```

Upon creation of a new activity, a new class will be created that extends AppCompactActivity. It will also contain a method called onCreate, which will always be executed after the activity is created. This method is used to do any initial set up to the activity, such as populating the layout with data pulled from the database.

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

If an activity has already been created, but a user has navigated back to it, then the method onResume is executed, rather than onCreate. This is useful if you want to refresh a UI when you navigate to it.

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

Starting an Activity

```
2 usages
private void homeActivity(int Id) {
    Intent i = new Intent( packageContext: LoginActivity.this, HomeActivity.class);
    Bundle param = new Bundle();
    param.putInt("userId", user.getId());
    i.putExtras(param);
    startActivity(i);
    finish();
}
```

This is how a new activity is created. The intent contains the current activity, and the class of the activity that we want navigate to. A bundle is used to pass parameters into the new activity. Here we store the Id of the current user so that it can be used in the next activity to find that user in the database. These parameters contain a name and a value. StartActivity is used to navigate to the new activity, changing what is displayed on the screen. We use finish() to dispose of the current activity. If the current activity is disposed of, then pressing the back button will not take you back to it, so it is useful to remove that part in some cases.

```
intent = getIntent();
userId = intent.getIntExtra( name: "Id",  defaultValue: 0);
```

This is how we access the parameters that are passed into an activity.

Events

Events are used to trigger a method to execute when something happens in the UI, such as when a button is clicked.

```
Button registerBtn = findViewById(R.id.register_button);
registerBtn.setOnClickListener(v -> register());
```

Each UI element has its own class, so we can locate the UI element using its Id attribute, and assigning it to a java object. We can then set attributes for it in the java code, such as assigning the method register() to execute when it detects a click.

Adapters Cont.

```java
public class PlayerRowAdapter extends ArrayAdapter<Player> {

    // invoke the suitable constructor of the ArrayAdapter class
    1 usage
    public PlayerRowAdapter(@NonNull Context context, ArrayList<Player> playerList) {
        super(context, resource: 0, playerList);
    }//end constructor

    @NonNull
    @Override
    public View getView(int position, @Nullable View convertView, @NonNull ViewGroup parent) {

        // get view
        View currentItemView = convertView;

        // if the view is null, inflate a new one of player row
        if (currentItemView == null) {
            currentItemView = LayoutInflater.from(getContext()).inflate(R.layout.adapter_player_row, parent, attachToRoot: false);
        }

        // get the position of the view from the ArrayAdapter
        Player currentPlayer = getItem(position);

        // Assign value retrieved from player model to text view in player row
        TextView playerNum = currentItemView.findViewById(R.id.playerNumber);
        playerNum.setText(String.valueOf(currentPlayer.getPlayerNumber()));

        // Assign value retrieved from player model to text view in player row
        TextView playerFullName = currentItemView.findViewById(R.id.playerFullName);
        playerFullName.setText(currentPlayer.getFullName());

        // then return the recyclable view
        return currentItemView;
    }
}
```

Each adapter will look very similar to this. It is created with a constructor that takes a list of objects, and then getView tells the view what UI elements to bind what variables to. Notice that the class extends ArrayAdapter<Player>. In this case, player is the data model we are trying to display.

```java
PlayerRowAdapter playerRowAdapter = new PlayerRowAdapter( context: this, (ArrayList<Player>) players);
playerListView.setChoiceMode(ListView.CHOICE_MODE_SINGLE);
playerListView.setAdapter(playerRowAdapter);
playerListView.setOnItemClickListener(new AdapterView.OnItemClickListener() {

    @Override
    public void onItemClick(AdapterView<?> parent, View view,
                            int position, long id) {
        Player selectedPlayer = (Player) playerListView.getItemAtPosition(position);
        openPlayer(selectedPlayer.getId());
    }
});
```

This creates a new instance of the adapter with a list of data models, and binds it to a ListView in the activity. It is also binding the click event of a row in the list to the method openPlayer, and passing in the clicked player as a parameter.

Fragments

Fragments are very similar to activities, but extend Fragment rather than AppCompactActivity

```
public class DaySchedule extends Fragment {
```

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

    coachDb = CoachDb.getInstance(getActivity());

}
@Override
public View onCreateView(LayoutInflater inflater, ViewGroup container,
                         Bundle savedInstanceState) {
    // Inflate the layout for this fragment
    return inflater.inflate(R.layout.fragment_day_schedule, container, attachToRoot: false);
}
```

They contain an onCreate, but also an onCreateView. If you want a parent activity to communicate with a fragment that it contains, such as to tell the fragment to update its content, you would locate it similarly to how you would locate any other UI element, and then you can call a method on that fragment

```
DaySchedule schedule = (DaySchedule) getSupportFragmentManager().findFragmentById(R.id.day_schedule);
schedule.loadEventsAsync(selectedDate);            }
```

A new object is created that is the same type as the fragment, located by its id in the activity, and then a method is called that exists in the fragment.

Retrofit and The API Server

As we have discussed, we are building an application that allows users to view data that will be stored on a database. For this to work in a production environment, there would have to be a single database that is accessible to everyone. To achieve this, we are creating a Representational State Transfer API (REST API) in addition to a user interface. This API would be constantly running in some sort of cloud server, and any user can make requests through it to communicate with the database. A REST API allows us to use standard HTTP methods such as GET, POST, and PUT. Since we are not

have an application in production, we must simulate this by hosting the API server and database locally.

HTTP requests use JSON to handle data that is being transferred through them. Since JSON is a standard structure, we can use it to handle data that comes from different sources, even if those sources use different technologies. We are using Java to develop or user interface, and C# to program our API server. Retrofit is a library that facilitates the conversion of java classes to JSON and JSON to java classes. It also handles the majority of the work when sending and receiving HTTP requests and responses.

Getting Data from The Database

In order to interact with the database, such as when a user logs in, the UI must communicate with the database. This is done by sending a request using some URL.

```
"http://10.0.2.2:14509/";
```

This is the URL of the API server. As long as it is running locally a request will go through to it this URL will allow a request from the UI to trigger a method on the API server. The API server has special classes called controllers, which act as entry points for the requests. The controllers all contain methods that have path that can be used to execute this method.
http://10.0.2.2:14509/api/login/login-user?username=testUser&password=testPass

http://10.0.2.2:14509/ is the address of the API server, and tells the request to go to it

api/login/login-user tells the request which method on the API server to execute

?username=testUser&password=testPass indicates the values of the parameters passed into the method on the API server.

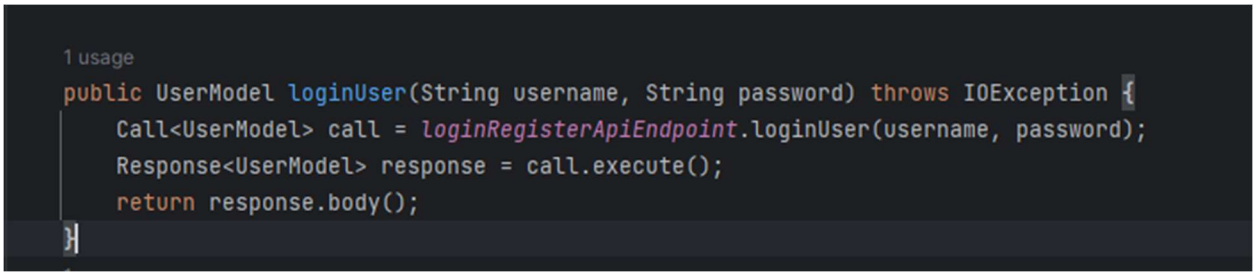
There are three main components that work together to allow the UI to talk with the API server.

SPWebApiRepository acts as the connection to the server. It is a singleton class and can be accessed using getInstance(). It contains the base address

Each type of data that we want to access will have an associated EndPoints interface. These are special interfaces contain methods that define the paths for each method on the API server.

```
4 usages
public interface LoginRegisterApiEndpoints {
        1 usage
        @GET("api/login/login-user")
        Call<UserModel> loginUser(@Query("username") String username, @Query("password") String password);
```

Each endpoint interface will have an associated client class. These client classes contain methods that call the endpoint methods convert the JSON sent as responses to java objects.



Any activity that requires data from the database will need to get it by calling a method in a client. I would suggest storing an instance of any clients that are needed as private variables in the activity, and initializing them in OnCreate.

SPWebApiRepository contains an instance of every client, and can be used in the activities to access those instances. Once you have an api client, you can call methods in it, and it will return full java objects for you to work with in the activity.