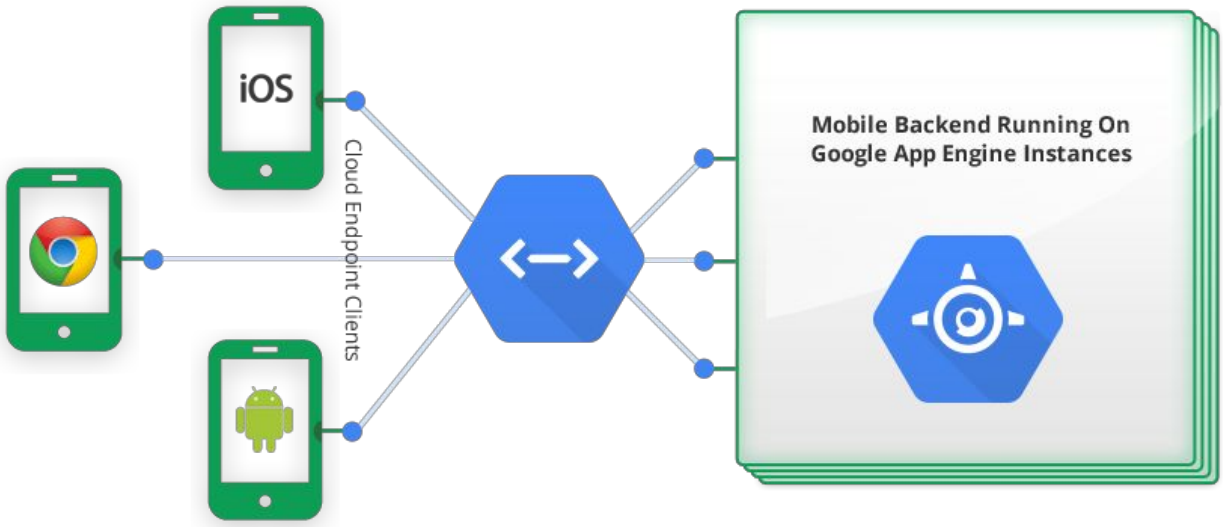
Lab 7 - Google Cloud Services

Google cloud provides a rich set of services centred around remote computation, storage, data analytics, and messaging, among other functionalities. The platform is quite extensive, and we are going to cover only a small part of it in the lab. You should, however, find out more about the affordances of Google cloud platform at https://cloud.google.com/
In this lab we will concentrate on:

- Cloud Endpoints - a means of creating an API and API-libraries for your cloud-side computation

Time permitting, we will also talk about Google Cloud Messaging that enables asynchronous communication between the cloud hosting your backend code and remote clients (e.g. Android devices running your mobile app)

Cloud Endpoints



Endpoints are used to expose the server-side functionality to the clients through an API that is created according to the developer's (your!) specification. The process of creating and linking your app through Endpoints goes as follows:

- Create your backend API project (using Maven to do this is the easiest method), then write your API backend code.
- Annotate your API backend code, so classes and client libraries can be generated from it.
- Generate the client library using Maven, or alternatively, the endpoints.sh command line tool.
- Write your client app, using the client library when making calls to the API backend.

Thankfully, Android Studio automates a large portion of the above process.

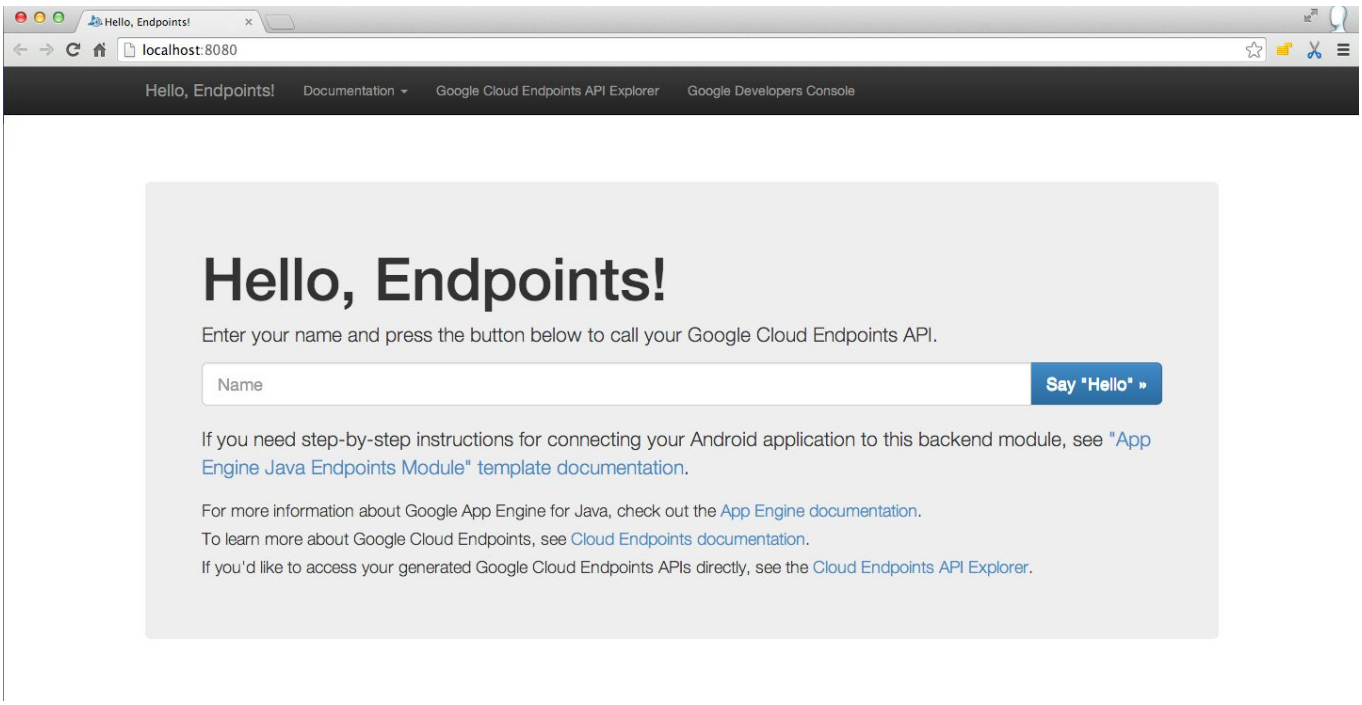
We will start with a simple example in which we take two numbers from a mobile device, perform remote computation (multiplies them) in the cloud and send the result back. This example is loosely based on the Simple Hello Endpoints from the Google documentation.

Start a new project in Android Studio, the one with a blank activity is fine. Now, add a new Google Cloud module (New->NewModule). Ensure that you add a new **Endpoints** module called backend. This backend module is created with some boilerplate code (basically, an example) -- **MyEndpoint** and **MyBean** classes are created. **MyBean** is an example of data that is transferred via endpoints, these often conform to the JavaBeans standard (http://www.oracle.com/technetwork/java/javase/documentation/spec-136004.html). **MyEndpoint** is more interesting as it comes with some annotation (notice `@Api`). During the compilation this annotation will be used to expose backend functions, through a library, to the mobile app. Spend some time until you understand what each of the annotation tags means.

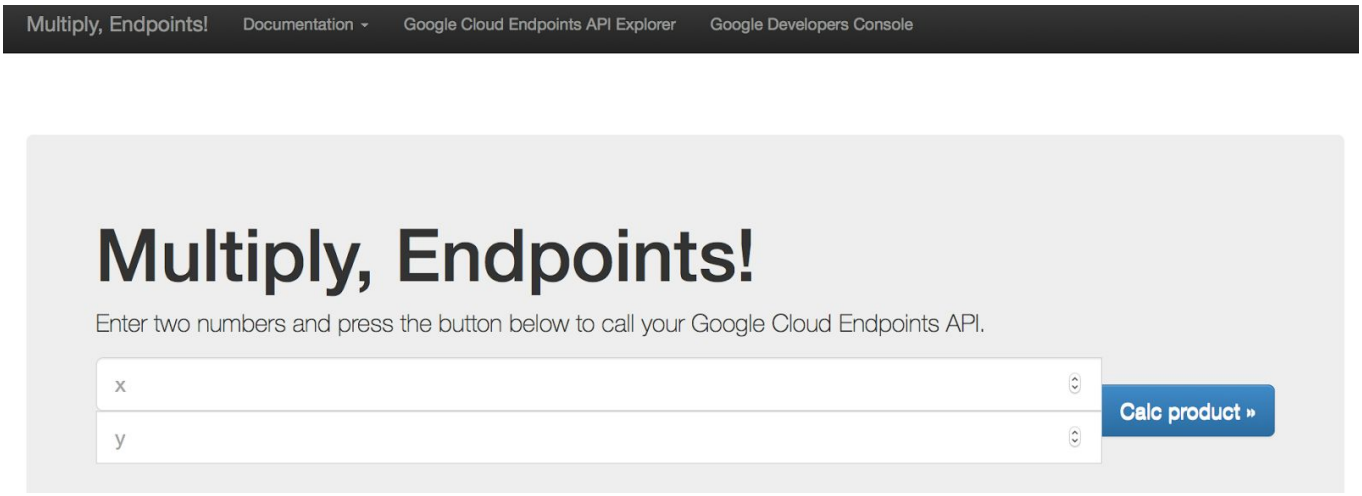
Let's build our own models and APIs based on the example code. Create two new packages "`apis`" and "`models`". We will put our endpoint code in "`apis`" and data representation classes in "`models`" (remember the Model-View-Controller paradigm). In "`models`" create **ResultMul** class. Make it a simple JavaBean with a single integer field holding the result of a multiplication and implement get/set methods for accessing and setting the result. In "`apis`" create **ResultMulEndpoint** class that has a method for multiplying two integer numbers and returns a **ResultMul** object with the result:

```
public ResultMul multiply(@Named("x") int x, @Named("y") int y);
```

Ensure that you have Java SDK installed, and then test your backend code directly from AndroidStudio. Note that "backend" runnable configuration has been created for you automatically. Running this configuration will run the backend code on your local machine, as if it is running on Google Cloud. Try it, and open the browser navigating to `localhost:8080`. You should get something like this:



When you created a new endpoints module an html file got created as well: `index.html` in `webapp` directory. Open the file and modify it, so that the end result looks like this:



It is crucial that you modify the JavaScript code so that “multiply” is called with x and y as parameters:

```
var x = document.getElementById('xInput').value;
var y = document.getElementById('yInput').value;
gapi.client.myApi.multiply({'x': x, 'y':y}).execute(...
```

Now you can test your backend by putting two integers in place of x and y and calling “calc product”. If everything works well, we can continue with developing the mobile app.

Open your “app” module’s build.gradle file. Notice:

```
  compile project(path: ':backend', configuration: 'android-endpoints')
```

This links your Android code with the automatically-generated strongly-typed client libraries of your API. You can call your remote code directly from an Android app!

In **MainActivity** create two textboxes with numerical input, and one “multiply” button. When a user clicks on the button, the data should be sent to the server for multiplication. Any operations that use network connectivity should be performed on the background thread, otherwise our UI thread may hang prompting the user to close our app. Thus, create an AsyncTask **EndpointsAsyncTask** and make it take the two number and present a toast with the result back to the user.

The core of the work is done in the background, thus the task’s `doInBackground` method should look like this:

```
@Override
protected Integer doInBackground(Pair<Integer, Integer>... params) {
  if (myApiService == null) {  // Only do this once
    MyApi.Builder builder = new MyApi.Builder(AndroidHttp.newCompatibleTransport(),
      new AndroidJsonFactory(), null)
                .setRootUrl("http://10.0.2.2:8080/_ah/api/")
                .setGoogleClientRequestInitializer(new
GoogleClientRequestInitializer() {
                    @Override
                    public void initialize(AbstractGoogleClientRequest<?>
abstractGoogleClientRequest) throws IOException {
```

```
            abstractGoogleClientRequest.setDisableGZipContent(true);
            }
        });
       myApiService = builder.build();
    }

    int x= params[0].first;
    int y = params[0].second;

    try {
          return myApiService.multiply(x,y).execute().getData();
    } catch (IOException e) {
          return -1;
    }
}
```

Note how **MyApi** class got imported from the library that links with your backed code.

In `onPostExecute` add code to show a **Toast** with the result from the server.

Finally, call the task from your **MainActivity** with the values from **EditText** boxes (x, y) passed to it:
`new EndpointsAsyncTask().execute(new Pair<Integer, Integer>(x, y);`

Google Cloud Messaging (GCM)

Thanks to Google Play Services GCM can be used for asynchronous communication between the cloud and your app. To use GCM in your app you need to register the app, so that you can obtain the API key, and you need to add the necessary permissions to the manifest. A part of the process can be automated by obtaining your app's configuration file from the cloud services. Just follow the instructions here: https://developers.google.com/cloud-messaging/android/client

Android Google Play Services provide two useful classes GCMReceiver and GcmListenerService for listening and handling GCM messages. You can find out more by examining the example provided here: https://github.com/googlesamples/google-services.git

Once you master this example, try to implement GCM between your server and the client. Have the client send two integers, and then, after a random delay on the server, send the result back via GCM. On the client, instead of the toast, deliver a notification in the notification bar. Clicking on the notification should pop up an Activity with the result.

Happy coding!