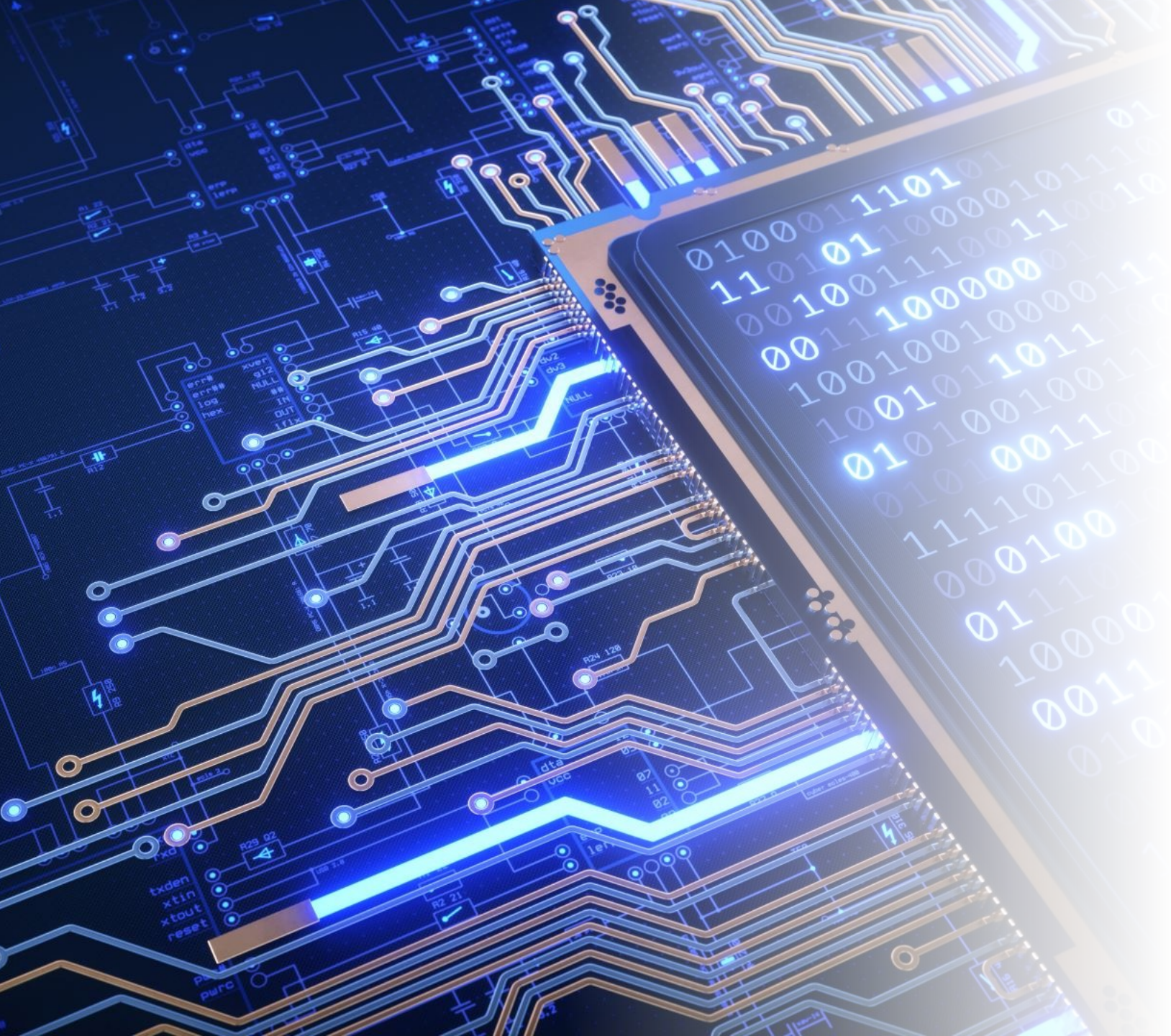


EEP 523: MOBILE APPLICATIONS FOR SENSING AND CONTROL

SUMMER 2021
Lecture 7

Tamara Bonaci

tbonaci@uw.edu

Agenda

- Review:
  - Deep Learning Workflow
  - Introduction to Arduino
- Controlling Arduino with Android using:
  - Wi-FI
  - BLE
- Long term data storage: **Databases**
  - Local: Room
  - Remote: Firebase

- **2D Graphics**: Canvas/View class
- Animations/MotionLayout

- (Web-based content)

- (Web services)

Review: Deep Learning Workflow in 7 steps

1. Decide on a goal
2. Collect a dataset
3. Design a model architecture
4. Train the model
5. Convert the model
6. Run inference
7. Evaluate and troubleshoot

Review: How do We Program an Arduino?

- Arduino language is basically a set **of C/C++ functions** that can be called from your code.
- Your sketch undergoes minor changes (e.g., automatic generation of function prototypes) and then is passed directly to a C/C++ compiler (avr-g++).

Review: How do We Program an Arduino?

The Arduino IDE supports the languages **C and C++** using special rules of code structuring.

A minimal Arduino C/C++ program consist of only two functions:

setup():

- Called once when a sketch starts after power-up or reset.
- Used to initialize variables, input and output pin modes, and other libraries needed in the sketch.

loop()

- After setup() function ends, the loop() function is executed **repeatedly** in the main program.
- Controls the board until the board is powered off or is reset.

Arduino Programming Language

Functions

Digital I/O
- digitalRead()
- digitalWrite()
- pinMode()

Analog I/O
- analogRead()
- analogReference()
- analogWrite()

Zero, Due & MKR Family
- analogReadResolution()
- analogWriteResolution()

Advanced I/O
- noTone()
- pulseIn()
- pulseInLong()
- shiftIn()
- shiftOut()
- tone()

Time
- delay()
- delayMicroseconds()
- micros()
- millis()

Math
- abs()
- constrain()
- map()
- max()
- min()
- pow()
- sq()
- sqrt()

Trigonometry
- cos()
- sin()
- tan()

Characters
- isAlpha()
- isAlphaNumeric()
- isAscii()
- isControl()
- isDigit()
- isGraph()
- isHexadecimalDigit()
- isLowerCase()
- isPrintable()
- isPunct()
- isSpace()
- isUpperCase()

Random Numbers
- random()
- randomSeed()

Bits and Bytes
- bit()
- bitClear()
- bitRead()
- bitSet()
- bitWrite()
- highByte()
- lowByte()

External Interrupts
- attachInterrupt()
- detachInterrupt()

Interrupts
- interrupts()
- noInterrupts()

Communication
- Serial
- Stream

USB
- Keyboard
- Mouse

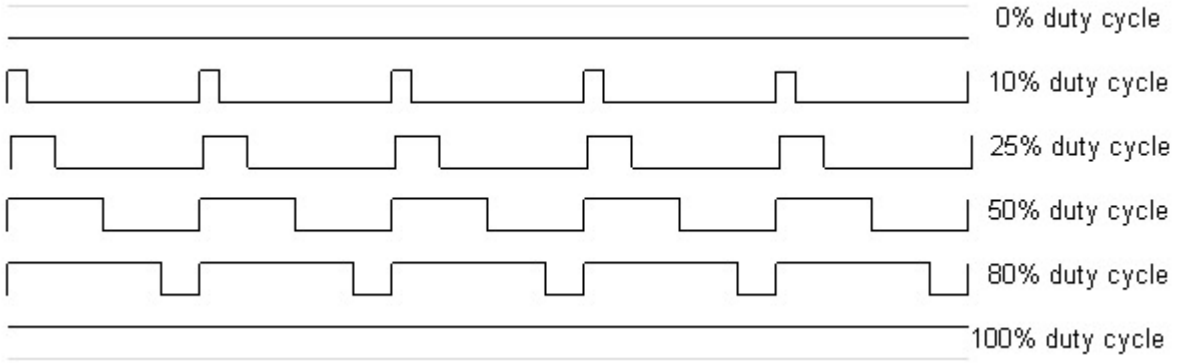
Review: Interrupts

Syntax

```
attachInterrupt(digitalPinToInterrupt(pin), ISR, mode);  //(recommended)
```

- digitalPinToInterrupt(pin): translate the actual digital pin to the specific interrupt number. For example, if you connect to pin 3, use digitalPinToInterrupt(3)
- ISR: to call when the interrupt occurs; this function takes no parameters and returns nothing. This function is sometimes referred to as an interrupt service routine.
- mode: defines when the interrupt should be triggered. Four constants are predefined as valid values:
  LOW to trigger the interrupt whenever the pin is low,
  CHANGE to trigger the interrupt whenever the pin changes value
  RISING to trigger when the pin goes from low to high,
  FALLING for when the pin goes from high to low.
  HIGH to trigger the interrupt whenever the pin is high. (some boards only)

Review: Pulse-Width Mouldation (PWM)



PWM signal is a digital square wave, where the frequency is constant, but that fraction of the time the signal is on (the duty cycle) can be varied between 0 and 100%

```
analogWrite(pin, dutyCycle)
```

Review: PWM with Arduino

1. **AnalogWrite:**

```
analogWrite(pin, dutyCycle)
```

   + dutyCycle from 0 to 255
   + simple interface to hardware PWM
   - No control over frequency

2. **Time delays**:
   repeatedly turning the pin on and off for the desired times
   + Use any digital output pin, full control of duty cycle and frequency
   - Any interrupts will affect the timing
   - Can’t leave the output running while the processer does something else

3. **Microcontroller timers**:
   by manipulating the chip's timer registers directly, you can obtain more control than the analogWrite function provides.

Android BLE Connectivity

EE P 523, Lecture 7

Android Connectivity

- **Bluetooth**
- **Bluetooth Low Energy (BLE)**
- **Wi-Fi**
- **Near Field Communication (NFC)**

Bluetooth

- Used to transmit data that can be used for interactive services such as audio, messaging, and telephony.

Bluetooth + Low Energy

BLE: Sensors, remote control

BT: audio streaming

Bluetooth Low Energy (BLE)

- Provides significantly lower power consumption.
- This allows Android apps to communicate with BLE devices that have stricter power requirements, such as **proximity sensors, heart rate monitors, and fitness devices**.

Bluetooth + Low Energy

- Bluetooth Low Energy (BLE), available in Android 4.3 and later, creates short connections between devices to transfer bursts of data.
- BLE remains in **sleep mode when not connected**.
- Lower bandwidth and reduced power consumption compared to Classic Bluetooth.
- It is ideal for applications such as a heart-rate monitor or a wireless keyboard.
- To use BLE, devices need to have a **chipset that supports BLE.**

BLE

- Modulation rate: 1Mbps
  (In practice, you can expect between 5-10 KB per second, depending on the limitations of the devices used.

- Typical operating range is closer **to 2 to 5 meters**.

- The higher the range the more battery consumption

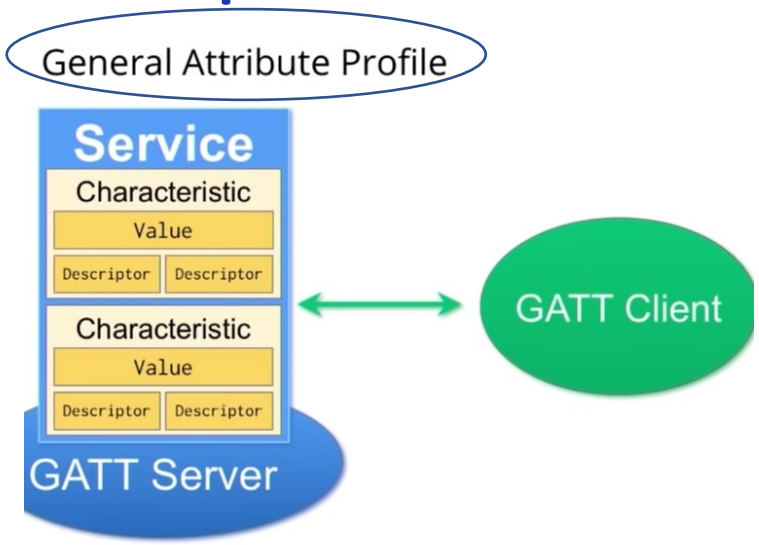
- **Bluetooth Device Address - a 48-bit number which uniquely identifies a device among peers.**

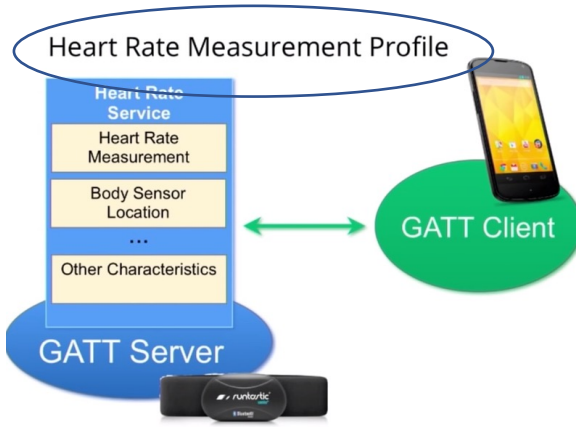
  You can think of a BDA as something similar to the MAC address in IP.

BLE Packets

A BLE device can talk to nearby devices in one of two ways: **Broadcasting** and **Connections**.

BLE Concepts for Android

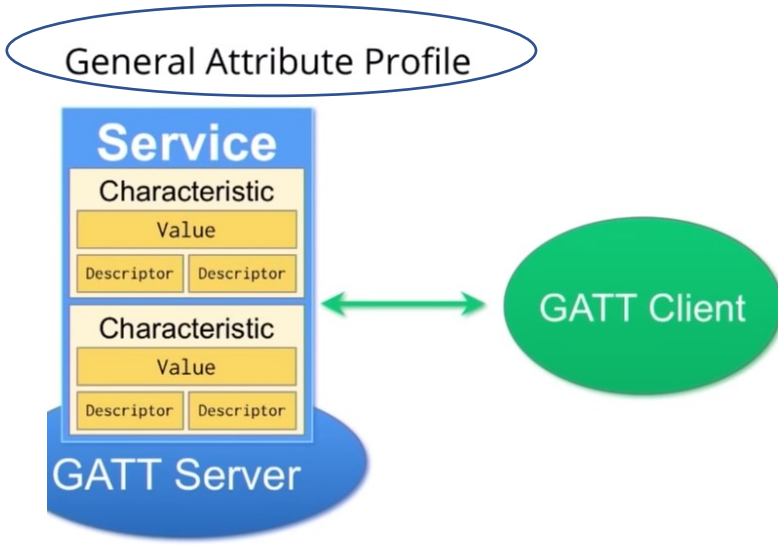


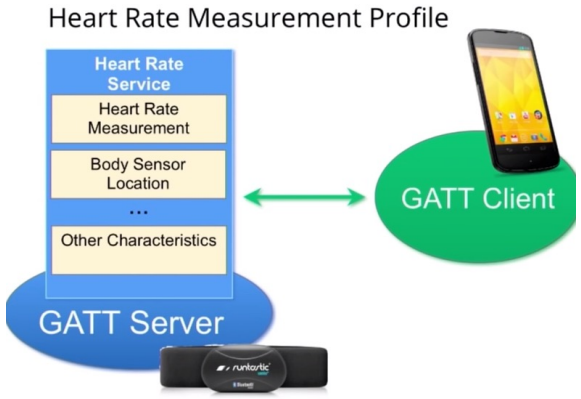


Generic Attribute Profile (GATT)

- General specification for sending and receiving short pieces of data known as "attributes" over a BLE link.
- All current Low Energy application profiles are based on GATT.
- A profile is a specification for how a device works in a particular application.
- A device can implement more than one profile. For example, a device could contain a heart rate monitor and a battery level detector.

Bluetooth Low Energy

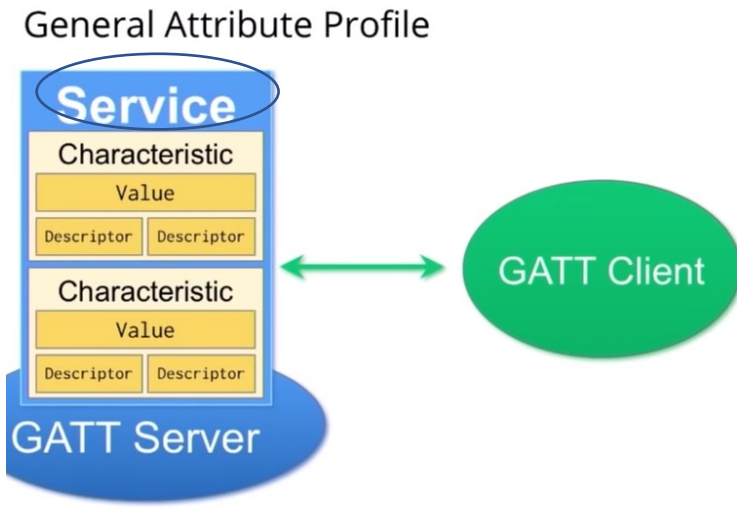




Attribute Protocol (ATT)—GATT is built on top of the Attribute Protocol (ATT).

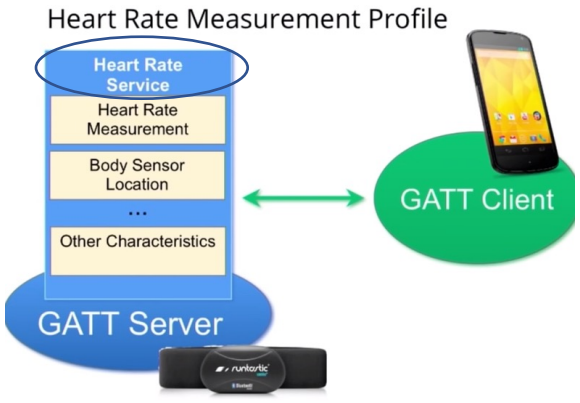
- Each attribute is uniquely identified by a Universally Unique Identifier (UUID), which is a standardized 128-bit format for a string ID used to uniquely identify information.

Bluetooth Low Energy

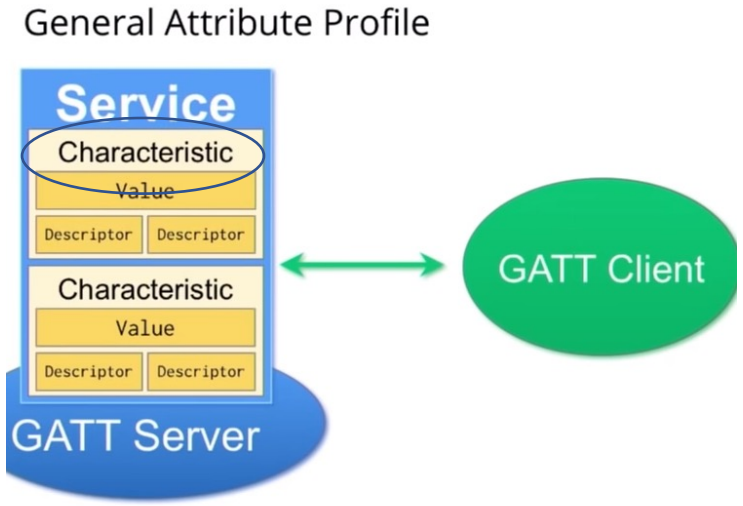


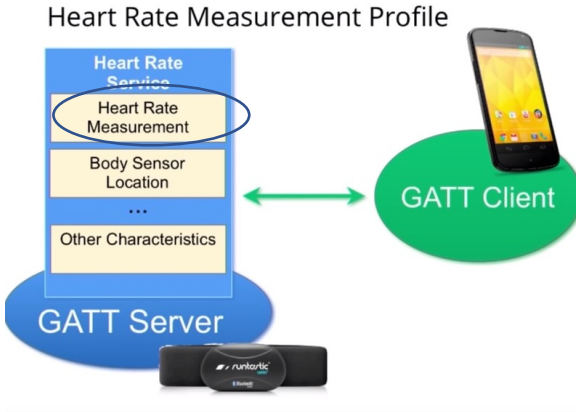
Service

- **C**ollection of characteristics.
- For example, you could have a service called "Heart Rate Monitor" that includes characteristics such as "heart rate measurement."



Bluetooth Low Energy





- **Characteristic**
  A characteristic can be thought of as a type, analogous to a class.

- **Descriptor**
  Attributes that describe a characteristic value.

For example, a descriptor might specify a human-readable description, an acceptable range for a characteristic's value, or a unit of measure that is specific to a characteristic's value.

BLE in Android

1. Request Permission
2. Set Up BLE
3. Find BLE devices
4. Connect to a GATT Server
5. Read BLE attributes
6. Receive GATT notifications
7. Close the Client App

1. Request BLE Permission

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

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

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

<uses-feature
android:name="android.hardware.bluetooth_le"android:required="true"/>
```

```
// Check permissions in the onCreate of your main Activity
ActivityCompat.requestPermissions(this,
    arrayOf( Manifest.permission.BLUETOOTH, Manifest.permission.BLUETOOTH_ADMIN,
Manifest.permission.ACCESS_FINE_LOCATION, Manifest.permission.ACCESS_COARSE_LOCATION), 1)
```

2. Setup BLE

- Get the **Bluetooth Adapter**
  - **T**here's one Bluetooth adapter for the entire system, and your application can interact with it using this object.

```
private val adapter: BluetoothAdapter
```

- **Enable Bluetooth**

```
val adapter: BluetoothAdapter?
adapter = BluetoothAdapter.getDefaultAdapter()
if (adapter != null) {
   if (!adapter.isEnabled) {
      val enableBtIntent = Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE)
      startActivityForResult(enableBtIntent, REQUEST_ENABLE_BT)

   }     }
```

3. Find BLE devices

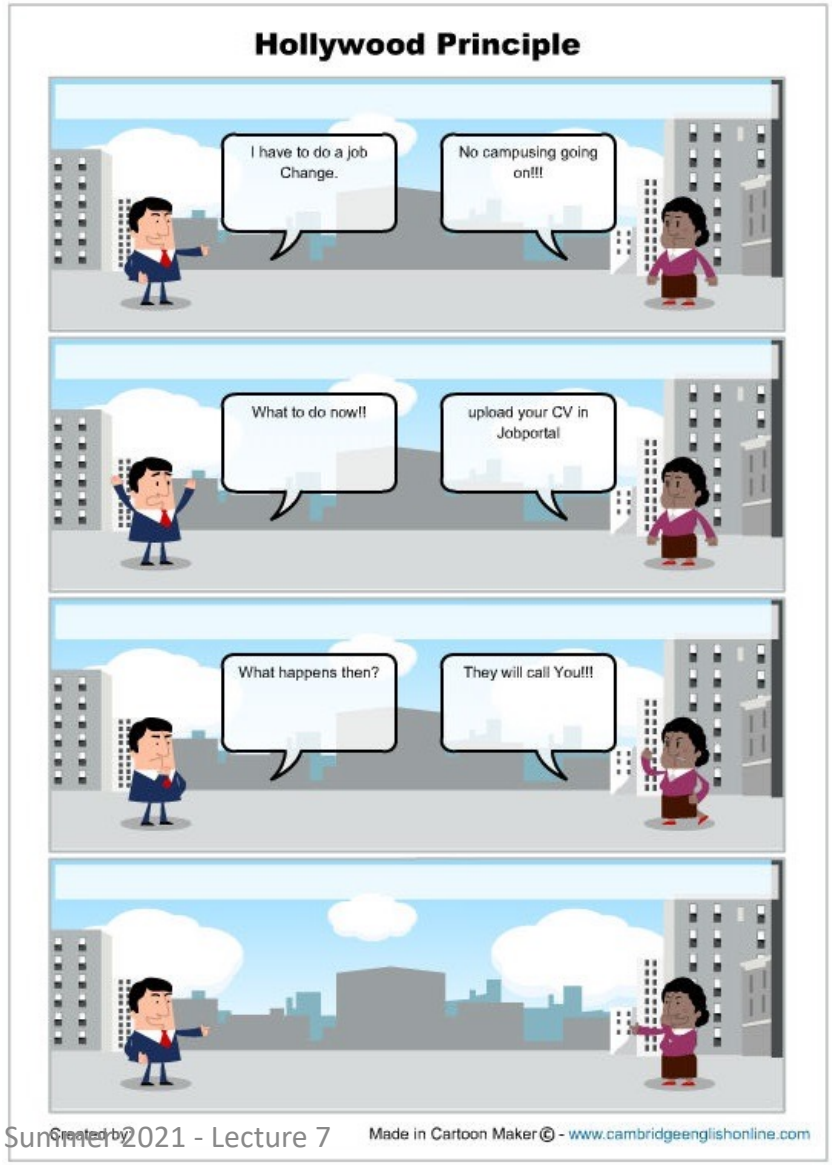
- To find BLE devices, you use the startLeScan() method.
- This method takes a BluetoothAdapter.LeScanCallbackas a parameter.
- You must implement this callback, because that is how scan results are returned.

Because **scanning is battery-intensive**, you should observe the following guidelines:

- As soon as you find the desired device, stop scanning.
- Never scan on a loop, and set a time limit on your scan. A device that was previously available may have moved out of range, **and continuing to scan drains the battery**.

Callback functions

- The concept of callbacks is to *inform a class synchronous / asynchronous if some work in another class is done*.



3. Find BLE devices

```
fun connect(v: View) {
    startScan()
}
```

```
fun connectFirstAvailable() {
    // Disconnect to any connected device.
    disconnect()
    // Stop any in progress device scan.
    stopScan()
    // Start scan and connect to first available device.
    connectFirst = true
    startScan()
}
```

```
fun startScan() {
    if (adapter != null) {
        if (!adapter.isEnabled) {
            val enableBtIntent = Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE)
            context.startActivity(enableBtIntent)
        }
        adapter.startLeScan(this)
    }
}
```

4. Conect to a GATT server

- The first step in interacting with a BLE device is connecting to it— more specifically, connecting to the GATT server on the device.

- To connect to a GATT server on a BLE device, you use the connectGatt() method.

This method takes three parameters:

- a Context object,
- autoConnect (boolean indicating whether to automatically connect to the BLE device as soon as it becomes available),
- a reference to a BluetoothGattCallback:

5. Read BLE atrributes

- Once your Android app has connected to a GATT server and discovered services, it can read and write attributes, where supported.

1. Loops through available GATT Services.

2. Loops through available Characteristics

6. Receive GATT notifications

It's common for BLE apps to ask to be notified when a particular characteristic changes on the device.

```
// Setup notifications on RX characteristic changes (i.e. data received).
// First call setCharacteristicNotification to enable notification.
if (!gatt.setCharacteristicNotification(rx, true)) {
    // Stop if the characteristic notification setup failed.
    Log.e("BLE", "characteristic notification setup failed")
    connectFailure()
    return
}
```

Once notifications are enabled for a characteristic, an onCharacteristicChanged() callback is triggered if the characteristic changes on the remote device:

```
override fun onCharacteristicChanged(gatt: BluetoothGatt, characteristic:
BluetoothGattCharacteristic) {
    super.onCharacteristicChanged(gatt, characteristic)
    notifyOnReceive(this, characteristic)
}
```

7. Close the Client App

Once your app has finished using a BLE device, it should call close() so the system can release resources appropriately:

```
fun close() {
    bluetoothGatt?.close()
    bluetoothGatt = null
}
```

```
else if (newState == BluetoothGatt.STATE_DISCONNECTED) {
    // Disconnected, notify callbacks of disconnection.
    rx = null
    tx = null
    notifyOnDisconnected(this)
}
```

Callback functions

- Class A calls Class B to get some work done in a Thread.
- If the Thread finished the work, it will inform Class A over the callback and provide the results. So there is no need for polling or something. You will get the results as soon as they are available.

In the BLE adapter Class

```
private val callbacks: WeakHashMap<Callback, Any>
```

```
notifyOnConnected(this)
```

```
private fun notifyOnConnected(uart: BLE) {
    for (cb in callbacks.keys) {
        cb?.onConnected(uart)
    }
}
```

In the Main Activity

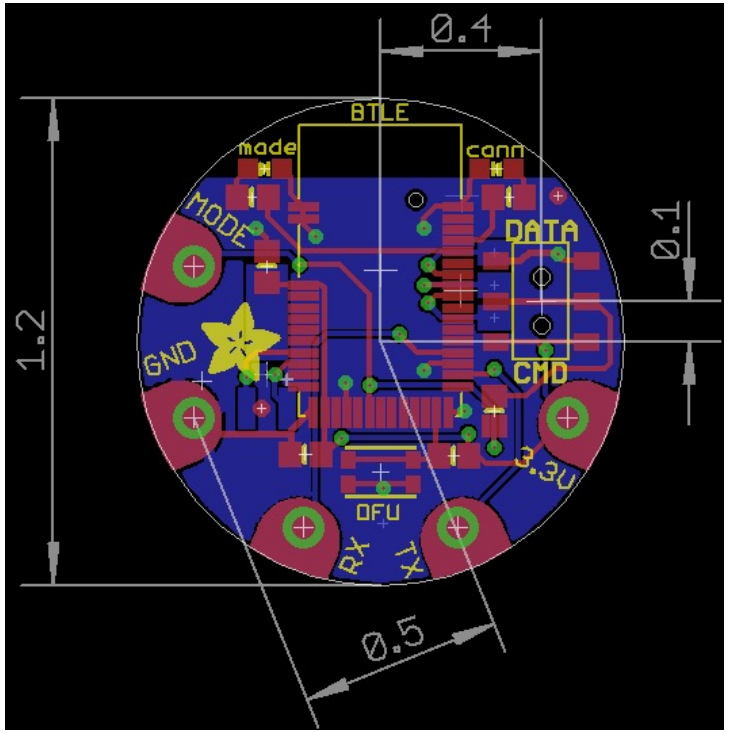
```
override fun onConnected(ble: BLE)
{
    writeLine("Connected!")

}
```

Android-Arduino Interaction

EE P 523, Lecture 7

Connecting Android with Arduino

Flora Wearable Bluefruit LE Module



- ARM Cortex M0 core running at 16MHz
- 256KB flash memory
- 32KB SRAM
- Transport: **UART @ 9600 baud**
- Bootloader with support for safe OTA (Over The Air) firmware updates
- Easy AT command set to get up and running quickly
- Diameter: 30.5mm / 1.2"
- Height: 4mm / 0.16"
- Weight: 2.5g

Universal Asynchronous Receiver Transmitter (UART) (link)

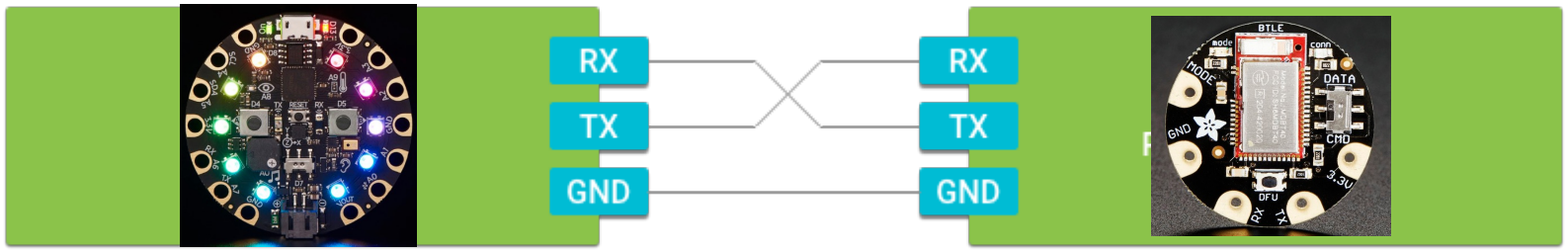
Generic interface for exchanging raw data with a peripheral device.

- ***Universal***: both the data transfer speed and data byte format are configurable.
- ***Asynchronous:*** there are no clock signals present to synchronize the data transfer between the two devices.

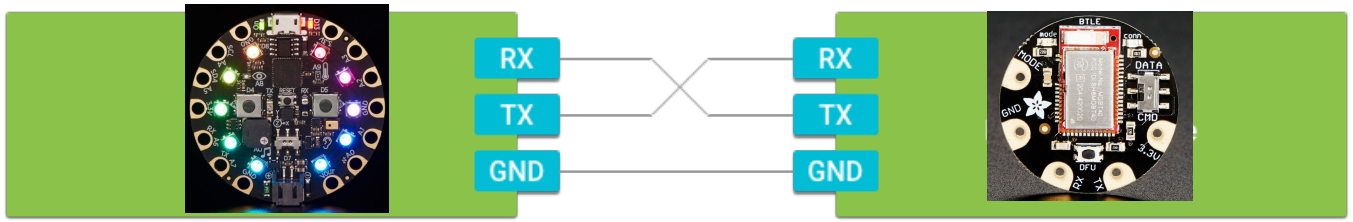
- The device hardware collects all incoming data in a first-in first-out (FIFO) buffer until read by your app.

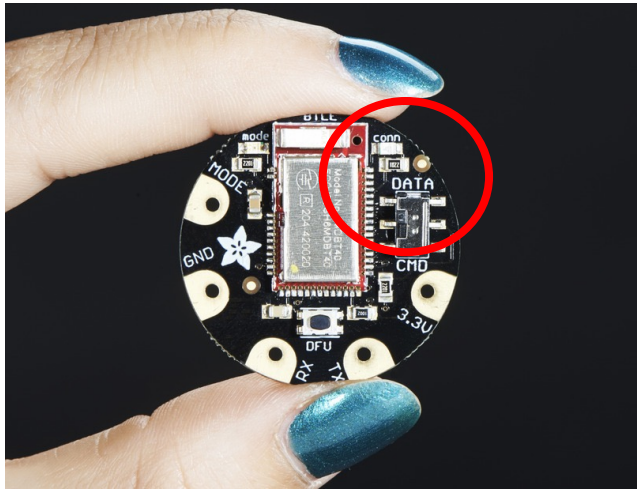
- UART data transfer is ***full-duplex***, meaning data can be sent and received at the same time.

- Typically, faster than I^2C, but the lack of a shared clock means that both devices must agree on a common data transfer rate that each device can adhere to independently with minimal timing error



Universal Asynchronous Receiver Transmitter UART

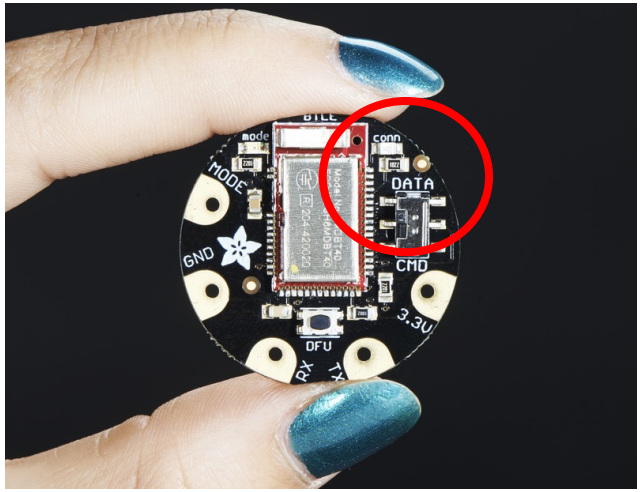




Don't forget, set the Bluefruit LE module switch to the DATA position!

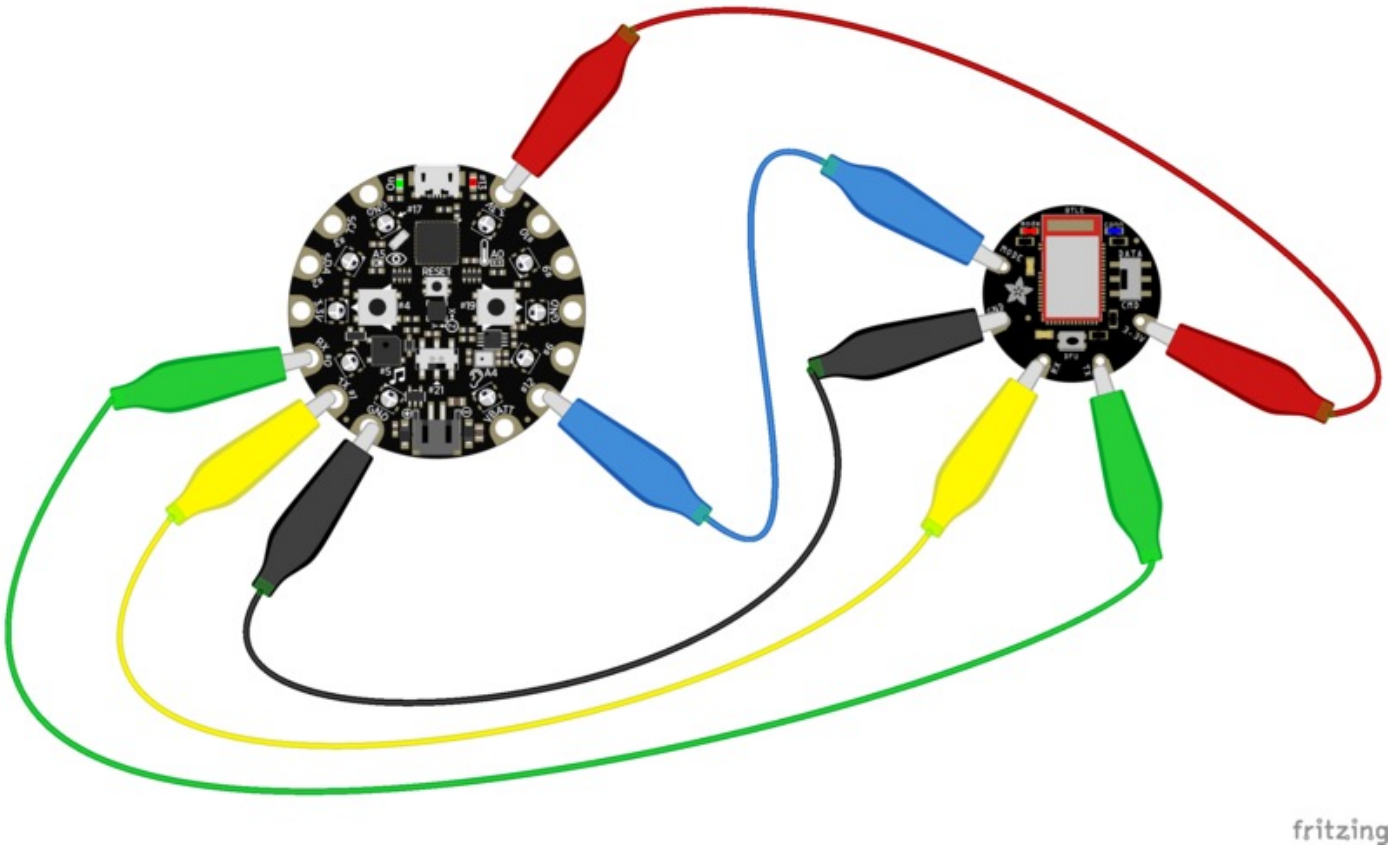
Command Mode

- Enter a variety of Hayes AT style commands to configure the device or retrieve basic information about the module of BLE connection(set the mode selection switch to **CMD** or setting the MODE pin to a high voltage)



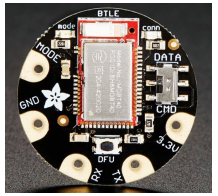
Don't forget, set the Bluefruit LE module switch to the DATA position!

Flora Wearable Bluefruit LE Module



UART UUID for BLE shield

```
companion object {

    // UUIDs for UART service and associated characteristics.
    var UART_UUID = UUID.fromString("6E400001-B5A3-F393-E0A9-E50E24DCCA9E")
    var TX_UUID = UUID.fromString("6E400002-B5A3-F393-E0A9-E50E24DCCA9E")
    var RX_UUID = UUID.fromString("6E400003-B5A3-F393-E0A9-E50E24DCCA9E")

    // UUID for the UART BTLE client characteristic which is necessary for
notifications.
    var CLIENT_UUID = UUID.fromString("00002902-0000-1000-8000-00805f9b34fb")

    // UUIDs for the Device Information service and associated characteristics.
    var DIS_UUID = UUID.fromString("0000180a-0000-1000-8000-00805f9b34fb")
    var DIS_MANUF_UUID = UUID.fromString("00002a29-0000-1000-8000-00805f9b34fb")
    var DIS_MODEL_UUID = UUID.fromString("00002a24-0000-1000-8000-00805f9b34fb")
    var DIS_HWREV_UUID = UUID.fromString("00002a26-0000-1000-8000-00805f9b34fb")
    var DIS_SWREV_UUID = UUID.fromString("00002a28-0000-1000-8000-00805f9b34fb")

}
```



https://infocenter.nordicsemi.com/index.jsp?topic=%2Fcom.nordic.infocenter.sdk5.v15.0.0%2Fble_sdk_app_nus_eval.html

Nordic Semiconductors

Android -> Arduino

In the Main Activity

```
ble!!.send("red")
```

```
// Send data to connected UART device.
fun send(data: ByteArray?) {
    if (tx == null || data == null || data.size == 0)
{
        // Do nothing if there is no connection or
message to send.
        return
    }
    // Update TX characteristic value.  Note the
setValue overload that takes a byte array must be
used.
    tx!!.value = data
    //        writeInProgress = true; // Set the
write in progress flag
    Log.d("BLE", "writing")
    gatt!!.writeCharacteristic(tx)
    try {
        writeInProgress.acquire()
    } catch (e: InterruptedException) {
        e.printStackTrace()
    }

}
```

```
int c = ble.read();
Serial.print((char)c); // Show
in the serial monitor

received += (char)c;

if(red == received){

      //Turn the LED red light

}
```

Android < - Arduino

```
sensorTemp = CircuitPlayground.temperature();

//Send data to Android Device
   char output[8];
   String data = "";
   data += sensorTemp;
   data.toCharArray(output,8);
   ble.print(data);
```

```
override fun onReceive(ble: BLE, rx: BluetoothGattCharacteristic) {
    writeLine("Received value: " + rx.getStringValue(0))

}
```

Change BLE shield name: Android Side

In a scenario with multiple BLE active with the same name:

```
// Get Bluetooth
ble = BLEControl(applicationContext, DEVICE_NAME)
```

```
companion object {
    private val DEVICE_NAME = "YodaBest"
    private val REQUEST_ENABLE_BT = 0
}
```

Change BLE shield name: Arduino Side

In a scenario with multiple BLE active with the same name:

```
#define NEW_DEVICE_NAME "PMP590 is awesome"
```

1. Factory reset to make sure everything is in a known state Apply the new name

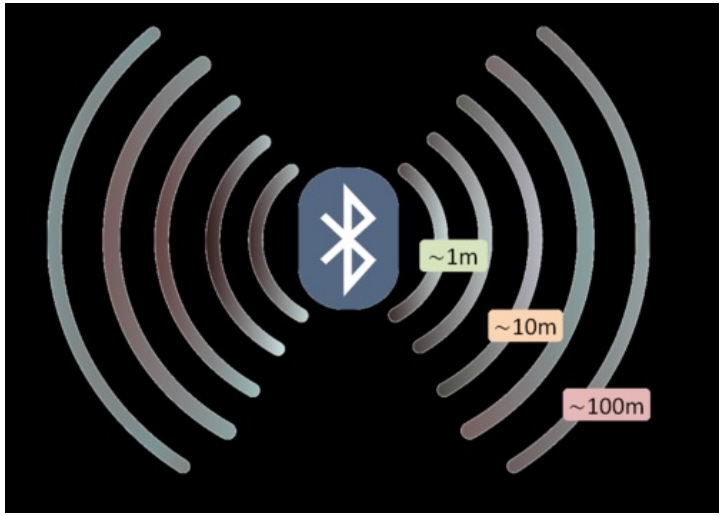
```
if ( ! ble.factoryReset() ){
      error(F("Couldn't factory reset"));
    }
```

2. Apply new name and wait for ok (see code on Arduino IDE)

Received Signal Strenght Indication (RSSI)

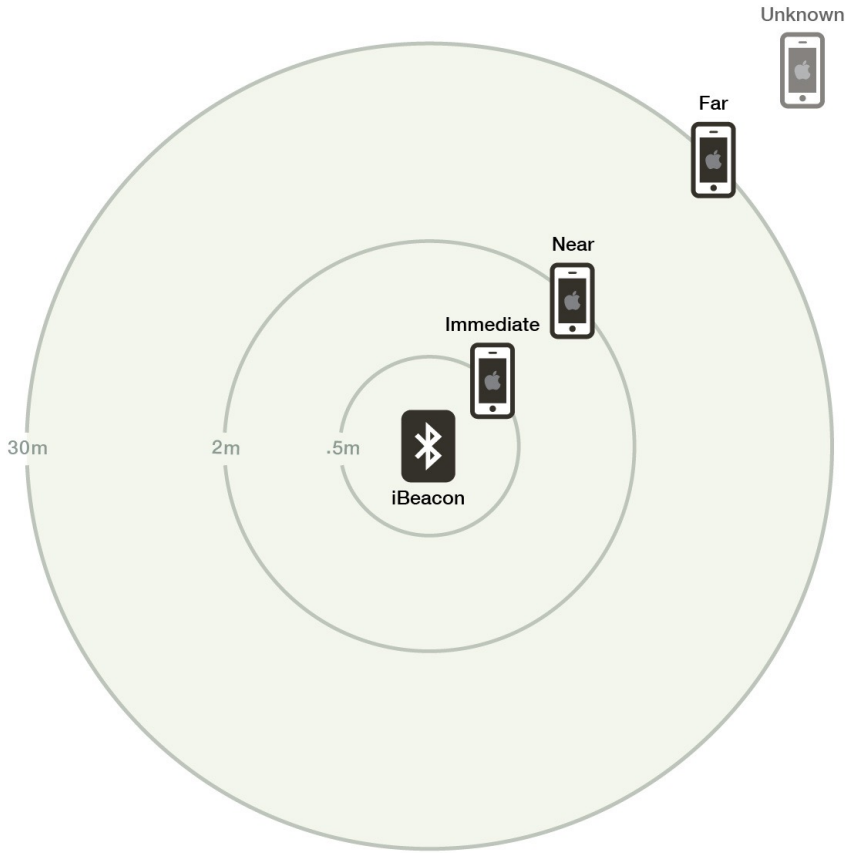
- RSSI stands for Received Signal Strength Indicator. It is the strength of the beacon's signal as seen on the receiving device, e.g., a smartphone. **The signal strength depends on distance and Broadcasting Power value**. At maximum Broadcasting Power (+4 dBm) the RSSI ranges from -26 (a few inch

- Measured Power: 1 Meter RSSI
- es) to -100 (40-50 m distance).



RSSI: Distance phone-arduino

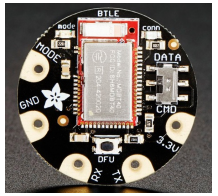
$$Distance = 10^{\left(\frac{Reference\ power - RSSI}{10 * N}\right)}$$



BLE shield

Datasheet

https://cdn-shop.adafruit.com/product-files/2267/MDBT40-P256R.pdf



1.2 Features

. 2.4GHZ transceiver

. -93dbm sensitivity in Bluetooth low energy mode

. TX Power -20 to +4dbm

. RSSI (1db resolution)

. ARM Cortex – M0 32 bit processor

. Serial Wire Debug (SWD)

. S100 series SoftDevice ready

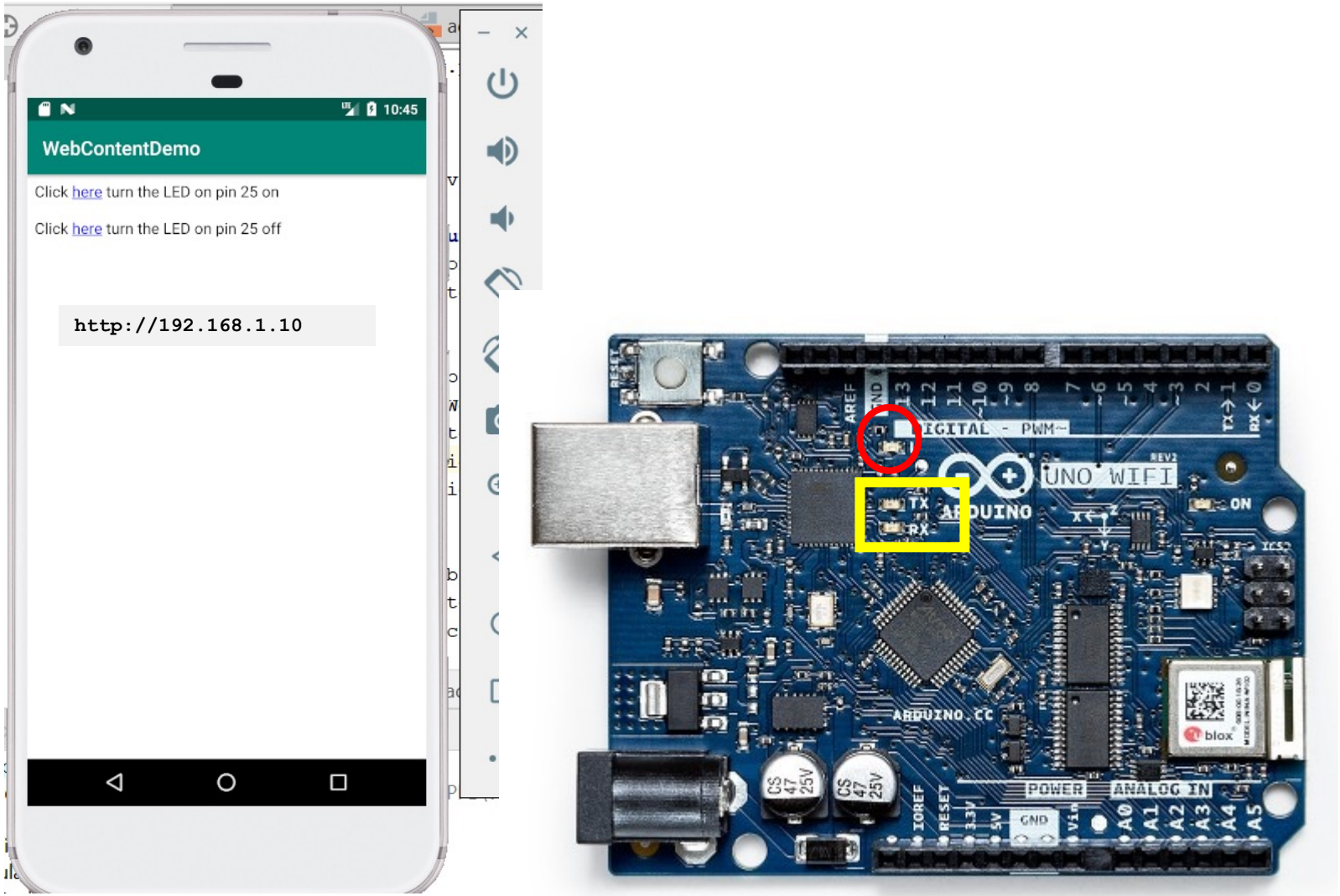
RSSI: Distance phone-arduino

BLEControl.kt class

```
fun getRSSI(){
    gatt!!.readRemoteRssi()
}
override fun onReadRemoteRssi(gatt: BluetoothGatt, rssi: Int, status:Int) {
    super.onReadRemoteRssi(gatt, rssi, status)
    if (status == BluetoothGatt.GATT_SUCCESS) {
        mRSSI = rssi
        notifyRSSIread(this, mRSSI)
    } else {

    }
}
```

```
private fun notifyRSSIread(uart:BLEControl,rssi:Int){
    for (cb in callbacks.keys){
        cb?.onRSSIread(uart,rssi)
    }
}
```

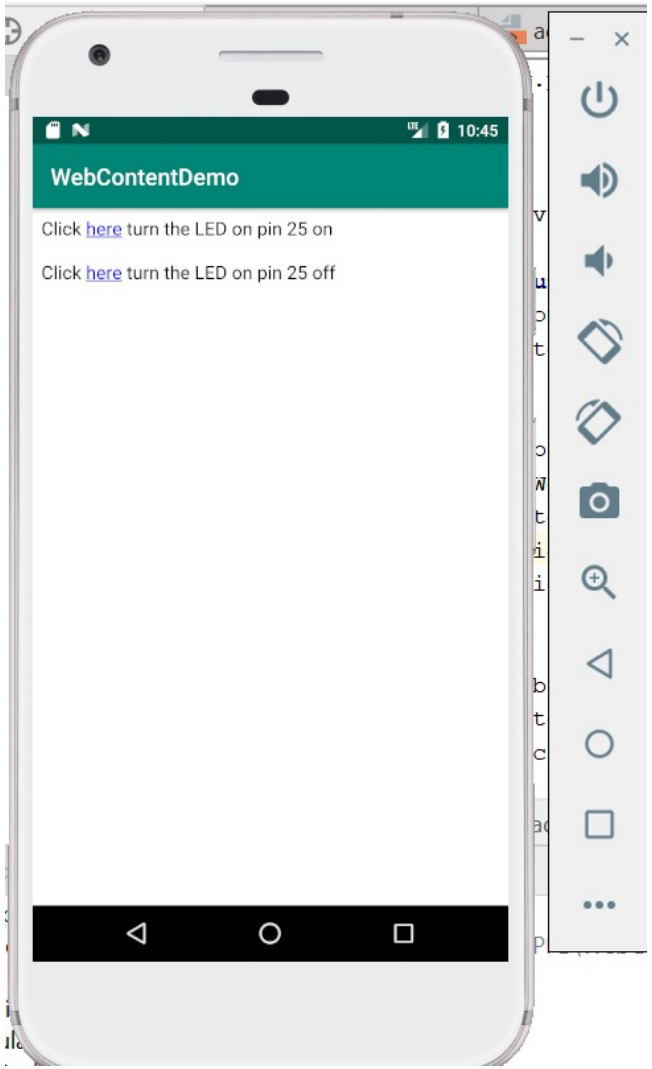
Connecting Arduino to WiFi



Connecting Arduino to WiFi

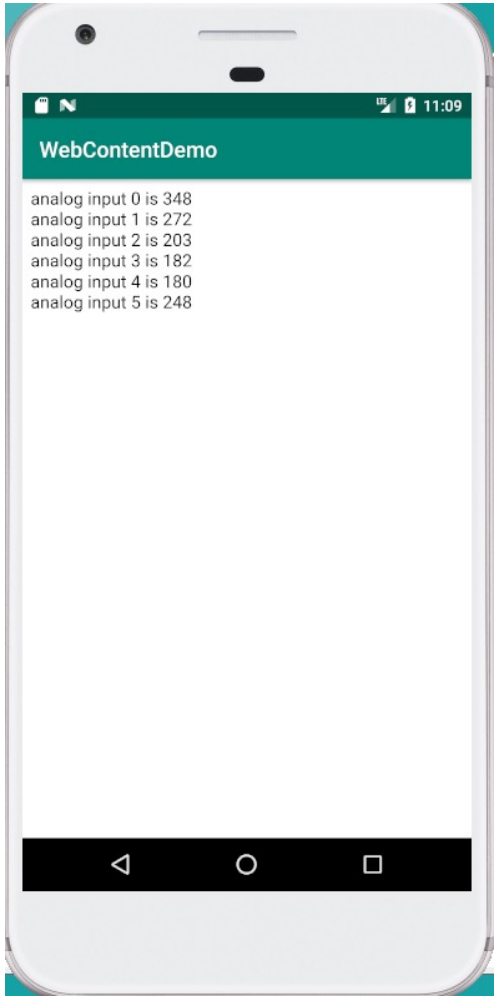

```
// if the current line is blank, you got two newline characters in a row.
        // that's the end of the client HTTP request, so send a response:
        if (currentLine.length() == 0) {
          // HTTP headers always start with a response code (e.g. HTTP/1.1 200 OK)
          // and a content-type so the client knows what's coming, then a blank line:
          client.println("HTTP/1.1 200 OK");
          client.println("Content-type:text/html");
          client.println();

          // the content of the HTTP response follows the header:
          client.println();
          client.println("<br>");
          client.print("Click <a href=\"/H\">here</a> turn the LED on pin 25 on<br>");
          client.println("<br>");
          client.print("Click <a href=\"/L\">here</a> turn the LED on pin 25 off<br>");

          // The HTTP response ends with another blank line:
          client.println();
          // break out of the while loop:
          break;
        }
```

Connecting Arduino to WiFi




```
// Check to see if the client request was "GET /H" or "GET /L":
      if (currentLine.endsWith("GET /H")) {
        digitalWrite(25, HIGH); // GET /H turns the LED on

      }
      if (currentLine.endsWith("GET /L")) {
        digitalWrite(25, LOW); // GET /L turns the LED off
      }
```

Connecting Arduino to WiFi




```
for (int analogChannel = 0; analogChannel < 6;
analogChannel++) {
            int sensorReading = analogRead(analogChannel);
            client.print("analog input ");
            client.print(analogChannel);
            client.print(" is ");
            client.print(sensorReading);
            client.println("<br />");
          }
```

Connecting Arduino to WiFi

client disonnected
new client
GET /favicon.ico HTTP/1.1
Host: 192.168.1.10
Connection: keep-alive
User-Agent: Mozilla/5.0 (Linux; Android 7.0; Android SDK built for x86 Build/NYC; wv) AppleWebKit/537.36 (KHTML, like Gecko) Version/4.0 Chrome/51.0.2704.90 Mobile Safari/537.36
Accept: */*
Referer: http://192.168.1.10/H
Accept-Encoding: gzip, deflate
Accept-Language: en-US
X-Requested-With: edu.uw.pmpee590.webcontentdemo

client disonnected

NFC protocol

- NFC is a branch of High-Frequency (HF) RFID, and both operate at the 13.56 MHz frequency.
- NFC is designed to be a secure form of data exchange, and an NFC device is capable of being both an NFC reader and an NFC tag.
- This unique feature allows NFC devices to communicate peer-to-peer

NFC protocol

- As a finely honed version of HF RFID, near-field communication devices have taken advantage of the short read range limitations of its radio frequency.

- Because NFC devices must be in close proximity to each other, usually no more than a few centimeters, it has become a popular choice for secure communication between consumer devices such as smartphones.

NFC in Android

Android framework APIs are based around a NFC Forum standard called NDEF (NFC Data Exchange Format).

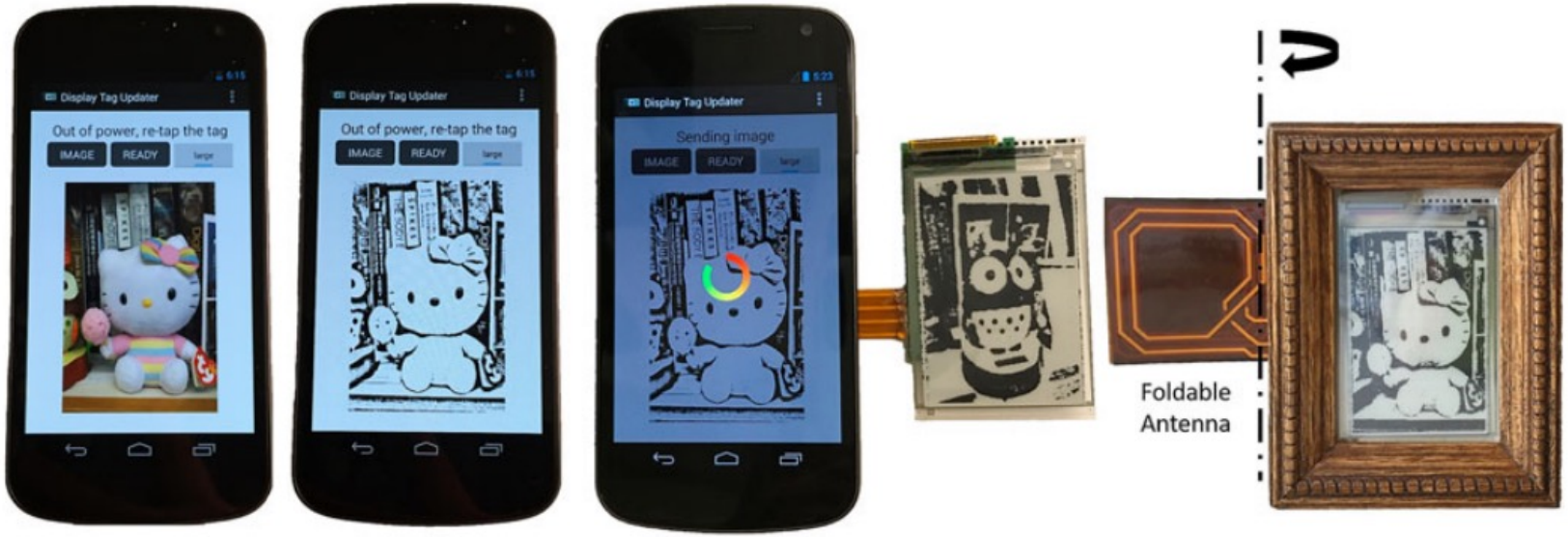
Android-powered devices with NFC simultaneously support three main modes of operation:

1.Reader/writer mode
NFC device read and/or write passive NFC tags and stickers.

2. P2P mode
NFC device exchange data with other NFC peers;

3.Card emulation mode,
NFC device itself to act as an NFC card.
The emulated NFC card can then be accessed by an external NFC reader, such as an NFC point-of-sale terminal.

NFC WISP



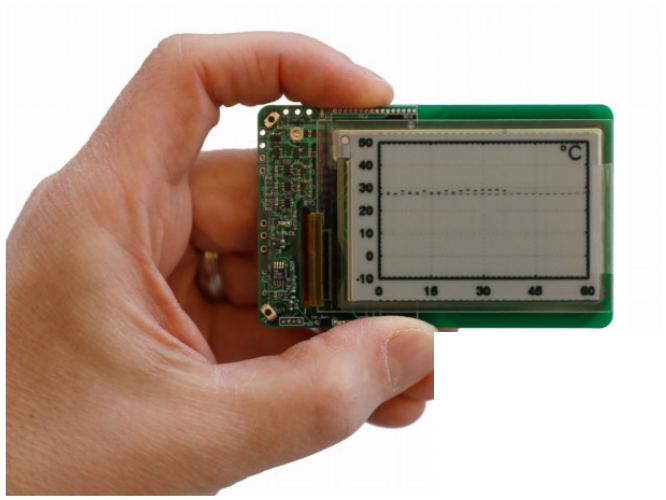
Step 1:
Choose photo and E-ink type

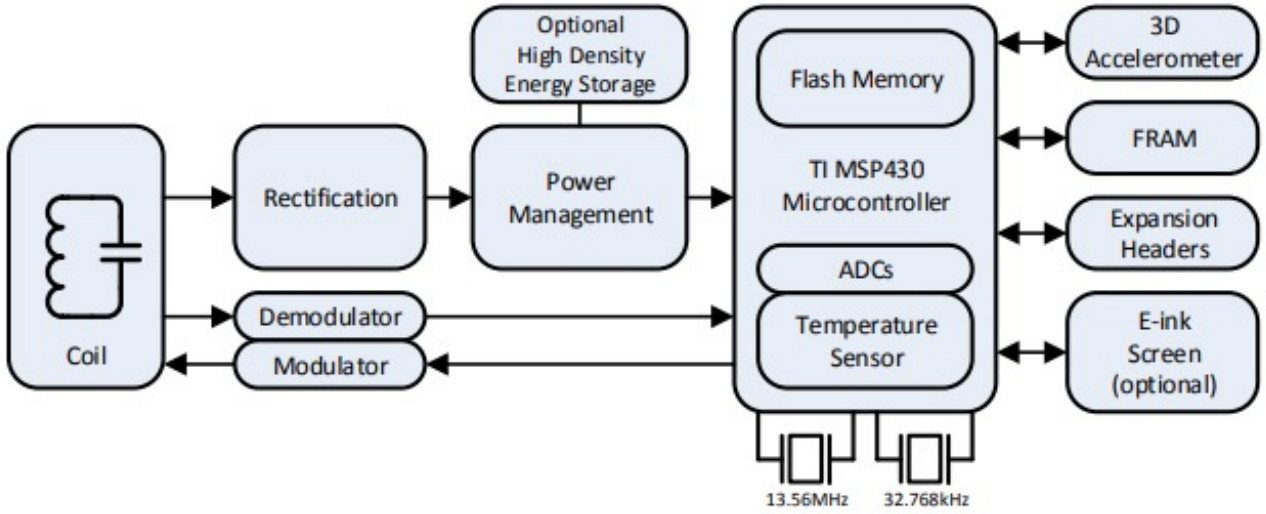
Step 2:
Pinch & crop photo and convert

Step 3:
Tap to NFC-WISP's antenna and wait for image updating within 3s

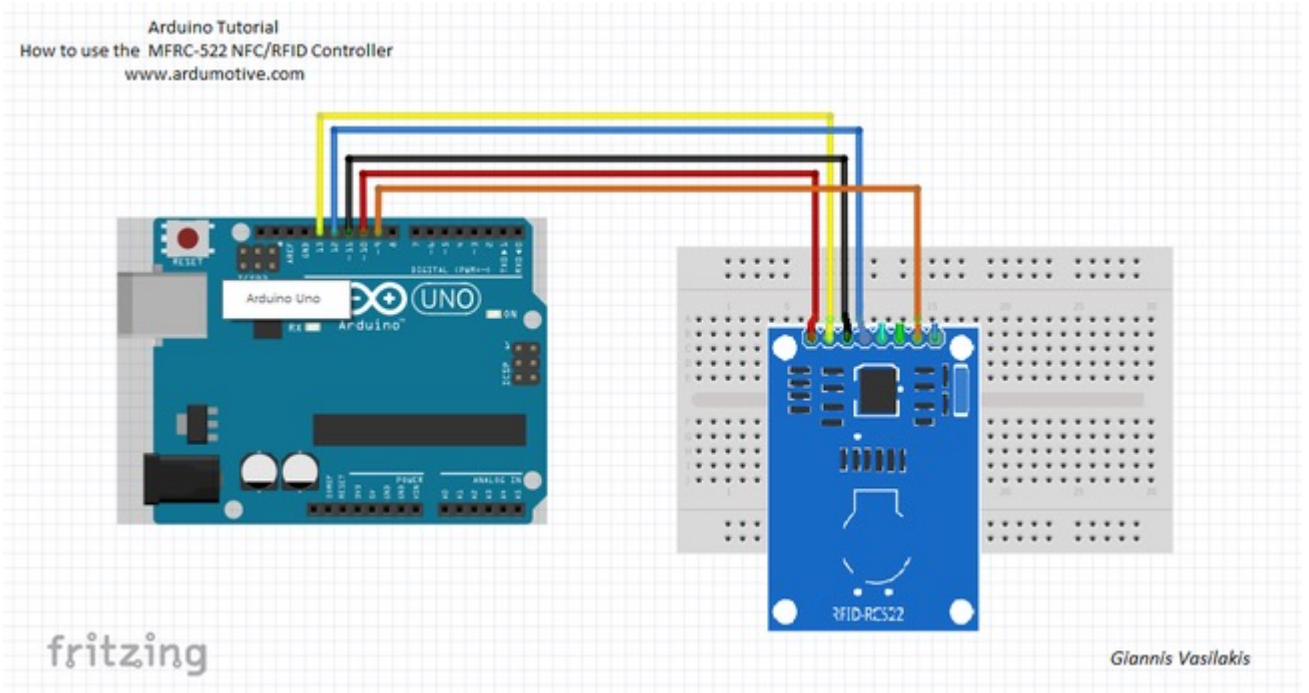
Application example:
Battery-free digital frame

NFC WISP




Optional
High Density
Energy Storage
Flash Memory
3D
Accelerometer
Rectification
Power
Management
TI MSP430
Microcontroller
FRAM
Expansion
Headers
ADCs
Coil
Demodulator
Temperature
Sensor
E-ink
Screen
(optional)
Modulator
13.56MHz
32.768kHz

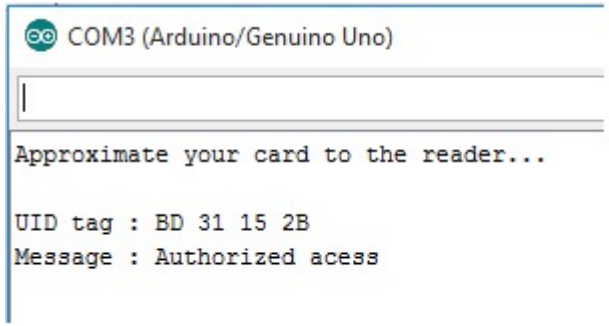
NFC in Arduino



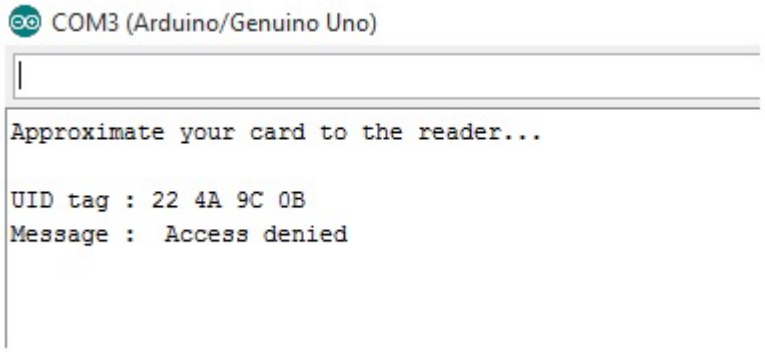
- The microcontroller and card reader uses SPI for communication
- The card reader and the tags communicate using a 13.56MHz electromagnetic field. (ISO 14443A standart tags)

NFC in Arduino

Approximate the card you've chosen to give access and you'll see:



If you approximate another tag with another UID, the denial message will show up:



App Data Storage and Databases

EE P 523, Lecture 7

Where to Save Non-UI data?

- We learned how to persist transient UI state data across rotation and process death using `ViewModel`, and saved instance state.
- These two approaches are great for small amounts of data tied to the UI.
- However, these approaches should not be used for storing **non-UI data or** data that is not tied to an activity or fragment instance and **needs to persist** regardless of UI state.

Where to Save Non-UI data?

- Shared Preferences
- Internal storage file
- (Strings.xml)
- Database
  - Local: SQLite + Room
  - Remote: Firebase Firestore

Database

- **Organized collection of data**, generally stored and accessed electronically from a computer system.

- Database is a Collection of one or more tables along with support for efficient operations (common: create, read, update, delete, fast search)
  - Think of an excel worksheet as a table

Where is the Data??

A database may be located in many places:

- Within your Android device (local database)
- On a remote server
- Spread though many remote servers (“in the cloud”)

Why use a Database?

powerful: can search, filter, combine data from many sources

fast: can search/filter a database very quickly compared to a file

big: scale well up to very large data sizes

safe: built-in mechanisms for failure recovery (transactions)

multi-user: concurrency features let many users view/edit data at same time

abstract: layer of abstraction between stored data and app(s)

common syntax: database programs use same SQL commands

Local Databases

EE P 523, Lecture 7

Remote vs. Local Databases

Remote

- Scalability
- Data Handling
- Synchronize data across all users
- Scalable

Local

- Offline access
- Speed
- “Free”
- Not rely on a 3rd party

Some DB software

Oracle

Microsoft

- **SQL Server** (powerful)
- **Access** (simple)

- **PostgreSQL**
  - powerful/complex free open-source database system

- **SQLite**
  - transportable, lightweight free open-source database system

- **MySQL**
  - simple free open-source database system
  - many servers run "LAMP" (Linux, Apache, MySQL, and PHP)
  - Wikipedia is run on PHP and MySQL

Taking to a Database

- **Structured Query Language** (SQL): a language for searching and updating a database
  - a standard syntax that is used by all database software (*with minor incompatibilities)*
  - generally, case-insensitive

- aggregate functions
- ALTER TABLE
- ANALYZE
- ATTACH DATABASE
- BEGIN TRANSACTION
- comment
- COMMIT TRANSACTION
- core functions
- CREATE INDEX
- CREATE TABLE
- CREATE TRIGGER
- CREATE VIEW
- CREATE VIRTUAL TABLE
- date and time functions
- DELETE
- DETACH DATABASE
- DROP INDEX
- DROP TABLE
- DROP TRIGGER
- DROP VIEW
- END TRANSACTION
- EXPLAIN
- expression
- INDEXED BY
- INSERT
- keywords
- ON CONFLICT clause
- PRAGMA
- REINDEX
- RELEASE SAVEPOINT
- REPLACE
- ROLLBACK TRANSACTION
- SAVEPOINT
- SELECT
- UPDATE
- UPSERT
- VACUUM
- WITH clause

SQL -> SQLlite

- SQLite is an open-source **relational database**, like MySQL or PostgreSQL.
- SQL, short for Structured Query Language, is a standard language used for interacting with databases.
- Unlike other databases, SQLite stores its data in simple files you can read and write using the SQLite library.
- For the most part, you do not need to know or care about SQLite when using Room

SQL Example

```
INSERT INTO `category` VALUES (2,'Climbing','climbing.png');
```

Category		
id	description	image
1	Hiking	hiking.png
2	Climbing	climbing.png

```
SELECT description FROM article WHERE price < 50;
```

Article				
id	description	price	image	categoryid
3	Climbing shoes	100	cs.png	2
6	Long rope	100	cr.png	2
7	Carabiner	40	c.png	2

Android SQLite Database API

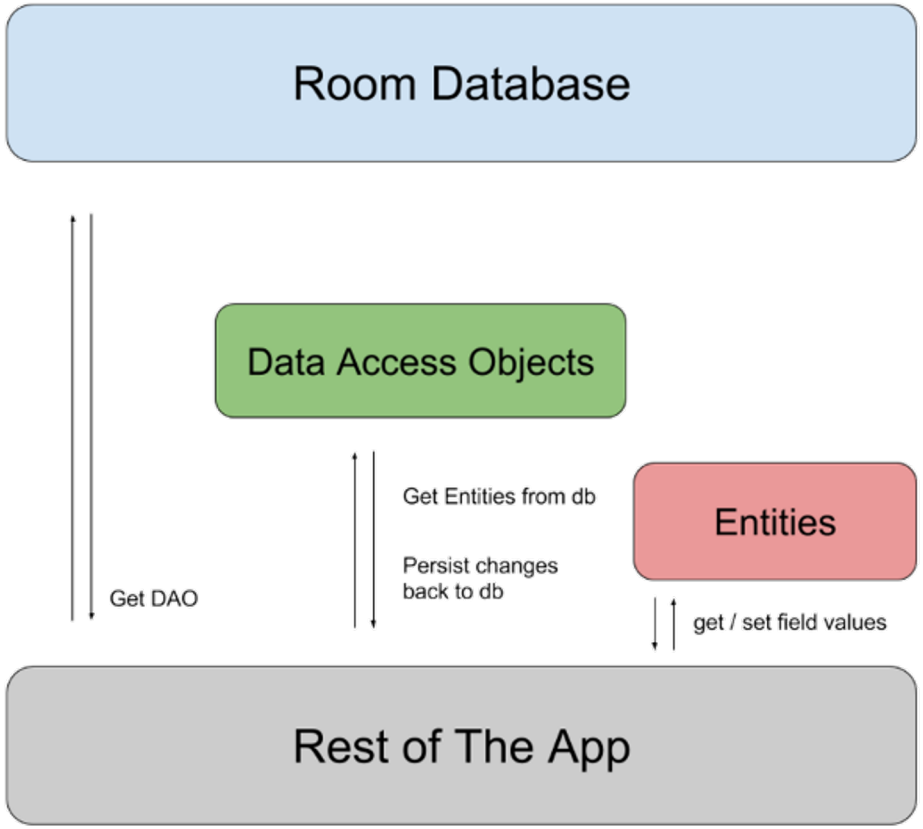
- Saving data to a database is ideal for repeating or structured data, such as contact information.

  - Android APIs are powerful, but they are fairly low-level and require a great deal of time and effort to use:
  - There is no compile-time verification of raw SQL queries. As your data graph changes, you need to update the affected SQL queries manually. This process can be time consuming and error prone.
  - You need to use lots of boilerplate code to convert between SQL queries and data objects.

Google highly recommended using the **Room Persistence Library** as an abstraction layer for accessing information in your app's SQLite databases.

What is Room?

Persistence **library provides an abstraction layer over SQLite** to allow for more robust database access while harnessing the full power of SQLite.

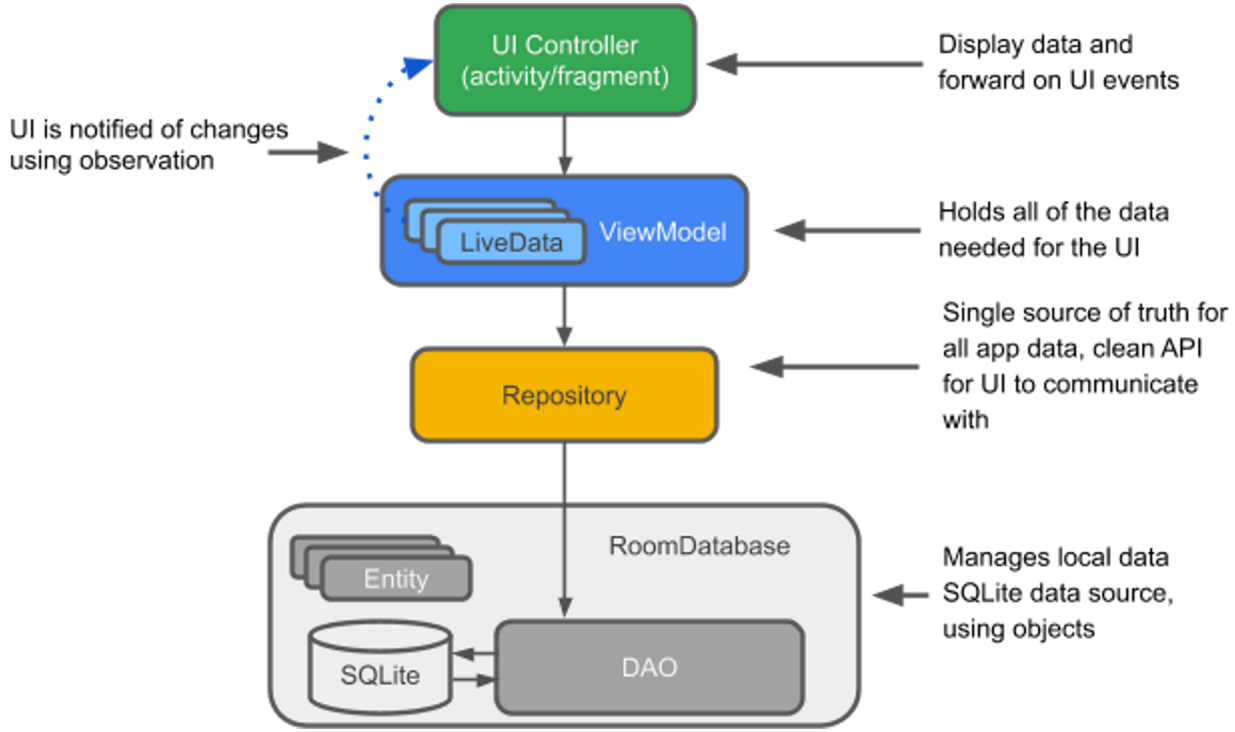


Room Database

What is a Room database**?**

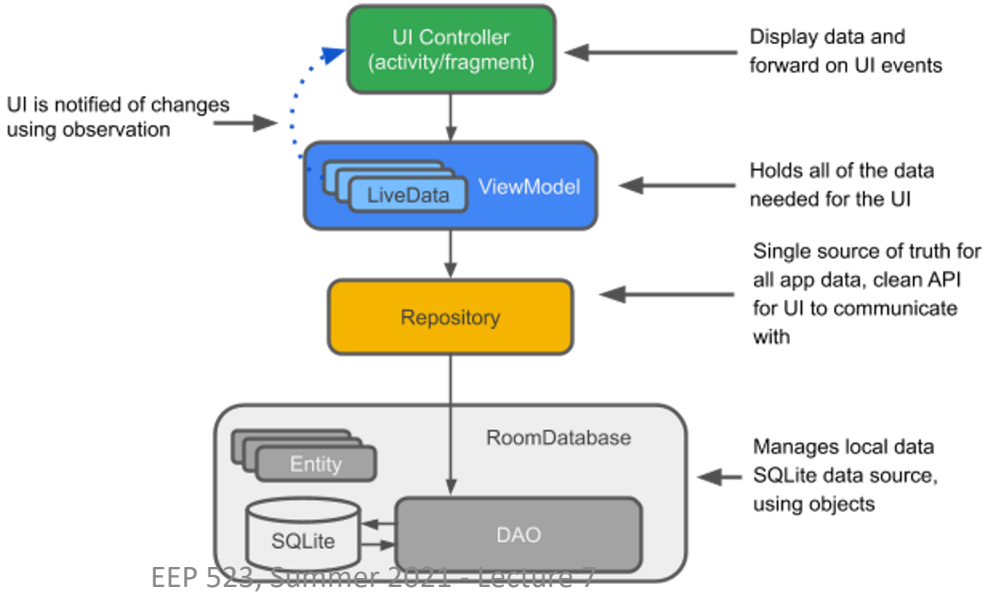
- Room is a database layer on top of an SQLite database.
- Room takes care of mundane tasks that you used to handle with an SQLiteOpenHelper.
- Room uses the DAO to issue queries to its database.
- By default, to avoid poor UI performance, Room doesn't allow you to issue queries on the main thread. When Room queries return LiveData, the queries are automatically run asynchronously on a background thread.
- Room provides compile-time checks of SQLite statements.

Room DB with a View



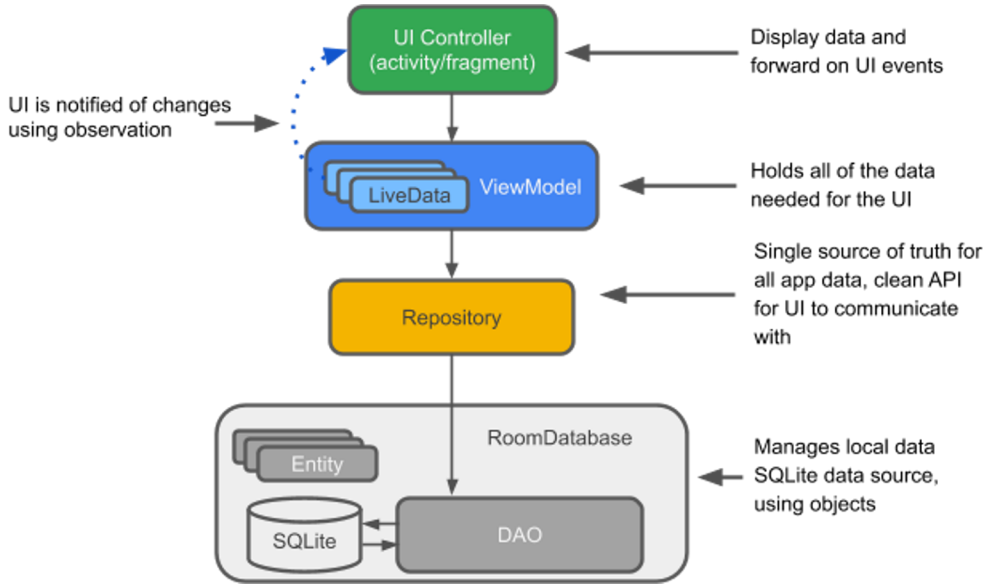
Android Local Database with Room

- **Entity:** Annotated class that describes a database table when working with Room.
- **SQLite database:** On device storage. Room persistence library creates and maintains this database for you.
- **DAO:** Data access object. A mapping of SQL queries to functions. When you use a DAO, you call the methods, and Room takes care of the rest.
- **Room database:** Simplifies database work and serves as an access point to the underlying SQLite database



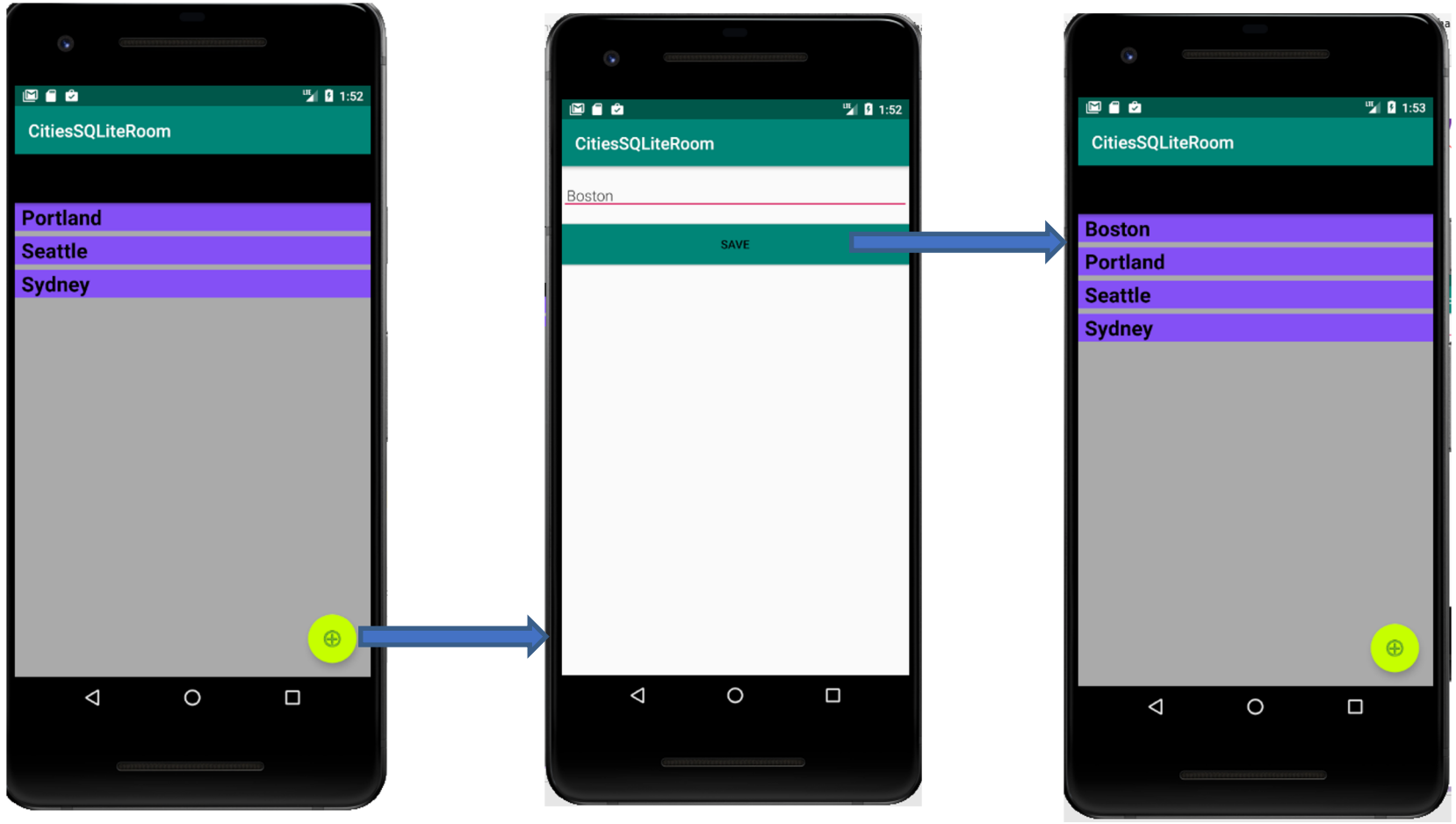
Android Local Database with Room

- **Repository:** Used to manage multiple data sources.
- **ViewModel:** Acts as a communication center between the Repository (data) and the UI. UI no longer needs to worry about origin of the data either.
- **LiveData:** A data holder class that can be observed. Always holds/caches latest version of data. Notifies its observers when the data has changed.

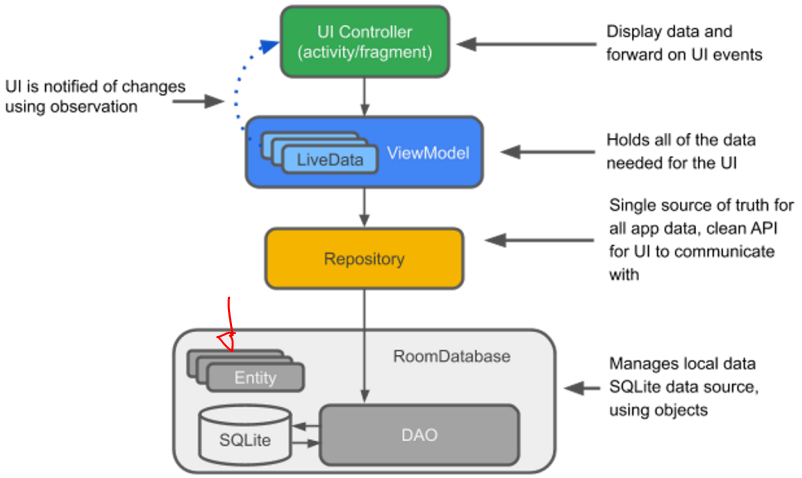


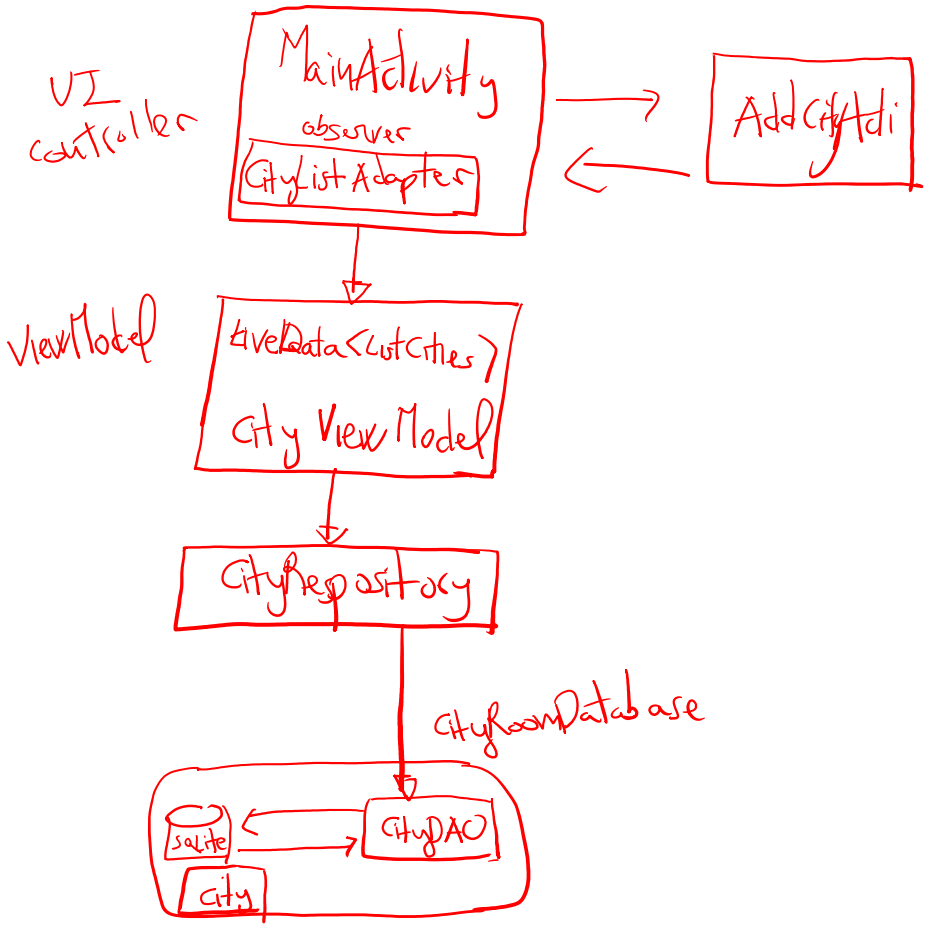
Android Local Database with Room

Example ***MyCitiesRoomSQLite***



Example App:CitiesSQLiteRoom





Android Local Database with Room

In the app's build.gradle file

Example ***MyCitiesRoomSQLite***

```
apply plugin: 'kotlin-kapt'
```

```
dependencies {
 def room_version = "2.2.5"

 implementation "androidx.room:room-runtime:$room_version"
 kapt "androidx.room:room-compiler:$room_version"

 implementation "androidx.room:room-ktx:$room_version"
```

Example App: CitiesSQLiteRoom

Example ***MyCitiesRoomSQLite***

The components of the app are:

- **MainActivity**: displays words in a list using a RecyclerView and the CityListAdapter. In the MainActivity, there is an Observer that observes the words LiveData from the database and is notified when they change.
- **NewCityActivity**: adds a new city to the list.
- **CityViewModel**(*): provides methods for accessing the data layer, and it returns LiveData so that MainActivity can set up the observer relationship.
- **LiveData**<List<City>>: Makes possible the automatic updates in the UI components
- **Repository**: manages one or more data sources. The Repository exposes methods for the ViewModel to interact with the underlying data provider. In this app, that backend is a Room database.
- **Room**: is a wrapper around and implements a SQLite database. Room does a lot of work for you that you used to have to do yourself.
- **DAO**: maps method calls to database queries, so that when the Repository calls a method such as getAllCities(), Room can execute **SELECT * from city_table ORDER BY word ASC.**
- **City**: is the entity class that contains a single work.

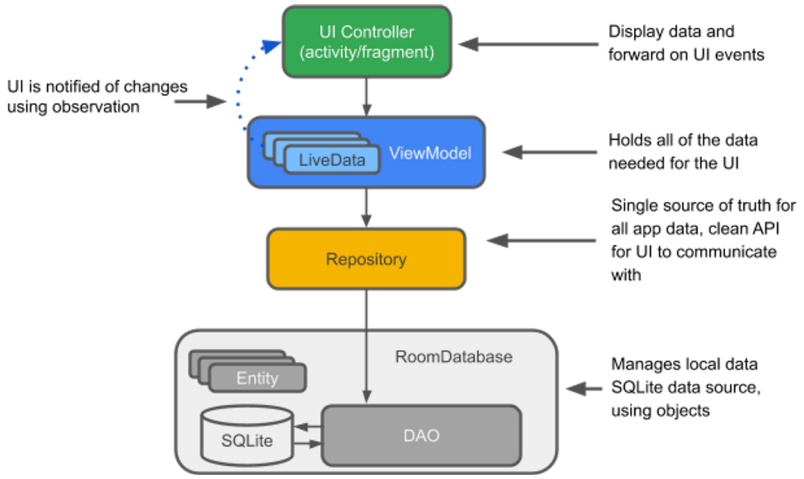
Android Local Database with Room

There are three steps to creating a database with Room:

1. Annotating your model class to make it a database entity

1. Creating the class that will represent the database itself

1. Creating a type converter so that your database can handle your model data



1. Defining Entities

- Room structures the database tables for your application based on the entities you define.

- Entities are model classes you create, annotated with the
- @Entity annotation.

- Room will create a database table for any class with that annotation.

```
@Database(entities = [City::class], version = 1)
abstract class CityRoomDatabase : RoomDatabase() {

}
```

2.1 Create the DataBase Class

2.2 Create the DAO

- A DAO is an interface that contains functions for each database operation you want to perform

- The `@Dao` annotation lets Room know that **CityDAO** is one of your data access objects. When you hook **CityDAO** up to your database class, Room will generate implementations of the functions you add to this interface.

```
@Dao
interface CityDAO {
    @Query("SELECT * from city ORDER BY cityName ASC")
    fun getAlphabetizedCities(): LiveData<List<City>>

    @Insert
    suspend fun insert(city: City)

    @Query("DELETE FROM city")
    fun deleteAll()

    @Query("SELECT * from city LIMIT 1")
    fun getAnyCity(): Array<City?>?

    @Delete
    fun deleteCity(city: City?)
}
```

3. Create the type Converter (optional)

- Room is able to store primitive types with ease in the underlying SQLite database tables, but other types will cause issues.
- If your class relies on the **Date** and **UUID** objects: Room does not know how to store by default.
- To tell Room how to convert your data types, you specify a type converter.
- A type converter tells Room how to convert a specific type to the format it needs to store in the database.
- You will need two functions, which you will annotate with @TypeConverter, for each type:
  - tell Room how to convert the type to store it in the database,
  - tell Room how to convert from the database representation back to the original type.

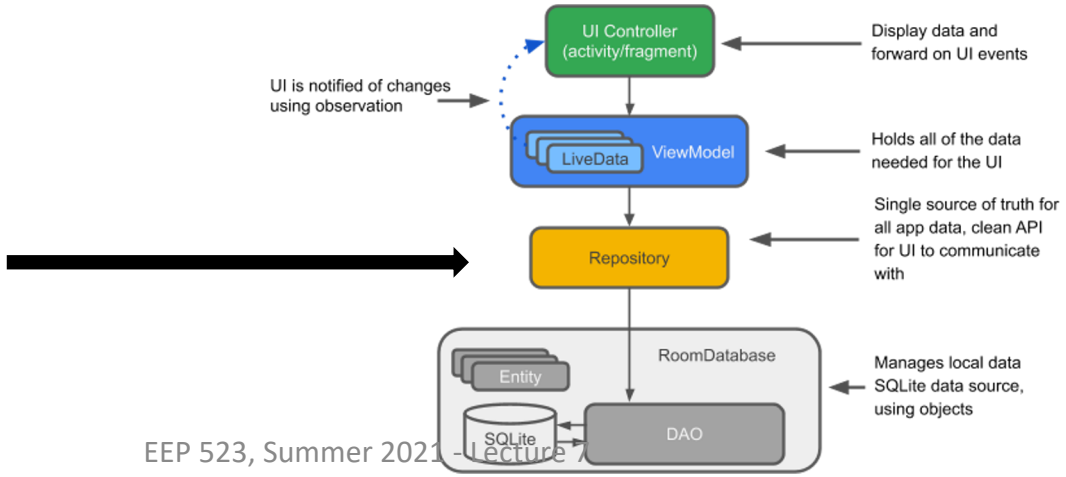
3. Create the type Converter: Example

```
class CityTypeConverters {
    @TypeConverter
    fun fromDate(date: Date?): Long? {
        return date?.time
    }

    @TypeConverter
    fun toDate(millisSinceEpoch: Long?): Date? {
        return millisSinceEpoch?.let {
            Date(it)
        }
    }

    @TypeConverter
    fun toUUID(uuid: String?): UUID? {
        return UUID.fromString(uuid)
    }

    @TypeConverter
    fun fromUUID(uuid: UUID?): String? {
        return uuid?.toString()
    }
}
```

```
@Database(entities = [ City::class ], version=1)
@TypeConverters(CityTypeConverters::class)


abstract class CityeDatabase : RoomDatabase() {
```

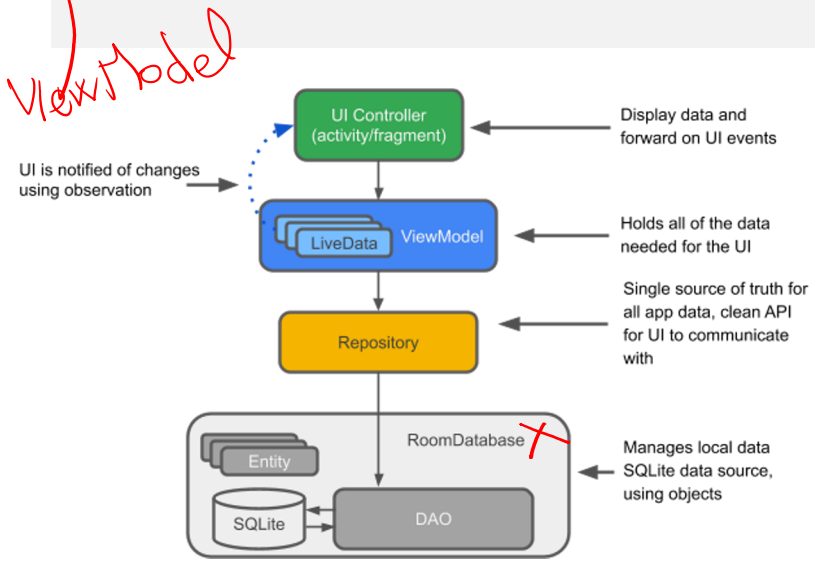
Accessing the DB using the *Repository* Pattern

- A repository class encapsulates the logic for accessing data from a single source or a set of sources.
- It determines how to fetch and store a particular set of data, whether locally in a database or from a remote server.
- Your UI code will request all the data from the repository, because the UI does not care how the data is actually stored or fetched.
- Those are implementation details of the repository itself.



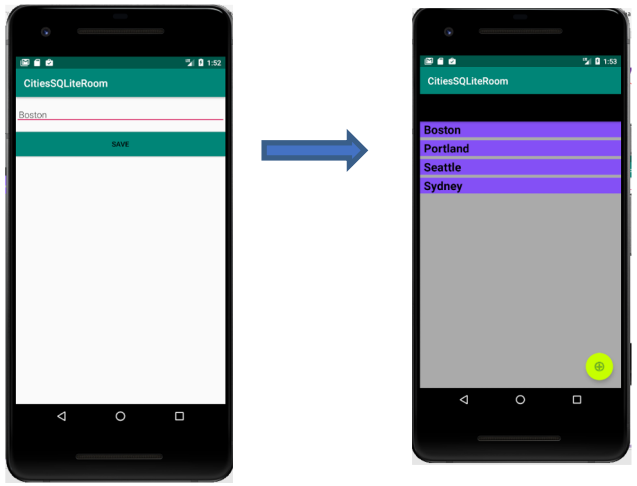
Accessing the DB using the Repository Pattern

```
class CityRepository(private val cityDao: CityDAO) {


    val allCities: LiveData<List<City>> =
cityDao.getAlphabetizedCities()

    @WorkerThread
    suspend fun insert(city: City) {
        cityDao.insert(city)
    }
}
```




```
@Dao
interface CityDAO {

    @Query("SELECT * from city ORDER BY
cityName ASC")
    fun getAlphabetizedCities():
LiveData<List<City>>

    @Insert
    suspend fun insert(city: City)


    @Query("DELETE FROM city")
    fun deleteAll()

    @Query("SELECT * from city LIMIT 1")
    fun getAnyCity(): Array<City?>?

}
```

Add new city to the Database




```
override fun onActivityResult(requestCode: Int, resultCode: Int, intentData: Intent?) {
    super.onActivityResult(requestCode, resultCode, intentData)

    if (requestCode == newWordActivityRequestCode && resultCode == Activity.RESULT_OK) {
        intentData?.let { data ->
            val city = City(data.getStringExtra(NewCityActivity.EXTRA_REPLY))
            cityViewModel.insert(city)
        }
    } else {
      }
}
```

Initialize DB

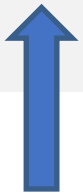
CityRoomDatabase.kt

```
/**
    * If you want to initialize the dab with these values
    */
   suspend fun populateDatabase(cityDao: CityDAO) {
       // Start the app with a clean database every time.
       // Not needed if you only populate on creation.
      // cityDao.deleteAll()
       if (cityDao.getAnyCity()!!.size < 1) {
           var city = City("Sydney")
           cityDao.insert(city)
           city = City("Seattle")
           cityDao.insert(city)
           city = City("Portland")
           cityDao.insert(city)
       }
```

Remove city with swipe:

MainActivity.kt

```
//THIS IS THE CODE TO REMOVE AN ITEM FROM THE DATABASE AND THE LIST WHEN THE ITEM IS SWIPED
fun removeCityOnswipe(){
    val itemTouchCallback = object : ItemTouchHelper.SimpleCallback(0, ItemTouchHelper.LEFT or
ItemTouchHelper.RIGHT) {

        override fun onSwiped(viewHolder: RecyclerView.ViewHolder, swipeDir: Int) {
            val position = viewHolder.adapterPosition
            val myCity: City? = adapter.getCityAtPosition(position)

            // Delete the word
            cityViewModel.deleteCity(myCity!!)
        }
    }
```



Solution

MainActivity.kt

```
//THIS IS THE CODE TO REMOVE AN ITEM FROM THE DATABASE AND THE LIST WHEN THE ITEM IS SWIPED
fun removeCityOnswipe(){
    val itemTouchCallback = object : ItemTouchHelper.SimpleCallback(0, ItemTouchHelper.LEFT or ItemTouchHelper.RIGHT) {

        override fun onSwiped(viewHolder: RecyclerView.ViewHolder, swipeDir: Int) {
            val position = viewHolder.adapterPosition
            val myCity: City? = adapter.getCityAtPosition(position)

            // Delete the word
            cityViewModel.deleteCity(myCity!!)
        }
    }
```

CityViewModel.kt

```
fun deleteCity(city:City) = viewModelScope.launch(Dispatchers.IO) {
    repository.deleteCity(city)
}
```

Solution

```
class CityRepository(private val cityDao: CityDAO) {


    val allCities: LiveData<List<City>> =
cityDao.getAlphabetizedCities()

    @WorkerThread
     suspend fun insert(city: City) {
        cityDao.insert(city)
    }

    @WorkerThread
     fun deleteCity(city:City) {
        cityDao.deleteCity(city)
    }
}
```

```
@Dao
interface CityDAO {

    @Query("SELECT * from city ORDER BY
cityName ASC")
    fun getAlphabetizedCities():
LiveData<List<City>>

    @Insert
    suspend fun insert(city: City)

    @Delete
    fun deleteCity(city: City?)


    @Query("DELETE FROM city")
    fun deleteAll()

    @Query("SELECT * from city LIMIT 1")
    fun getAnyCity(): Array<City?>?

}
```

Remote Databases

EE P 523, Lecture 7

Remote Databases: Firebase

https://www.youtube.com/watch?v=x8qTEMkZCPs&list=PLOU2XLYxmsILVTiOlMJdo7RQS55jYhsMi&index=87&t=310s

Remote Databases: Firebase

- Firebase is a mobile and web application development platform developed by Firebase, Inc. in 2011, then acquired by Google in 2014. As of October 2018, the Firebase platform has 18 products, which are used by 1.5 million apps

- Firebase evolved from Envolve, a prior startup founded by James Tamplin and Andrew Lee in 2011.

https://firebase.google.com/pricing?authuser=0

Remote Databases

Firebase offers two cloud-based, client-accessible database solutions that support **real-time** data syncing:

- **Cloud Firestore** is Firebase's newest database for mobile app development. It builds on the successes of the Realtime Database with a new, more intuitive data model. Cloud Firestore also features richer, faster queries and scales further than the Realtime Database.

- **Realtime Database** is Firebase's original database. It's an efficient, low-latency solution for mobile apps that require synced states across clients in realtime.

Both solutions offer:

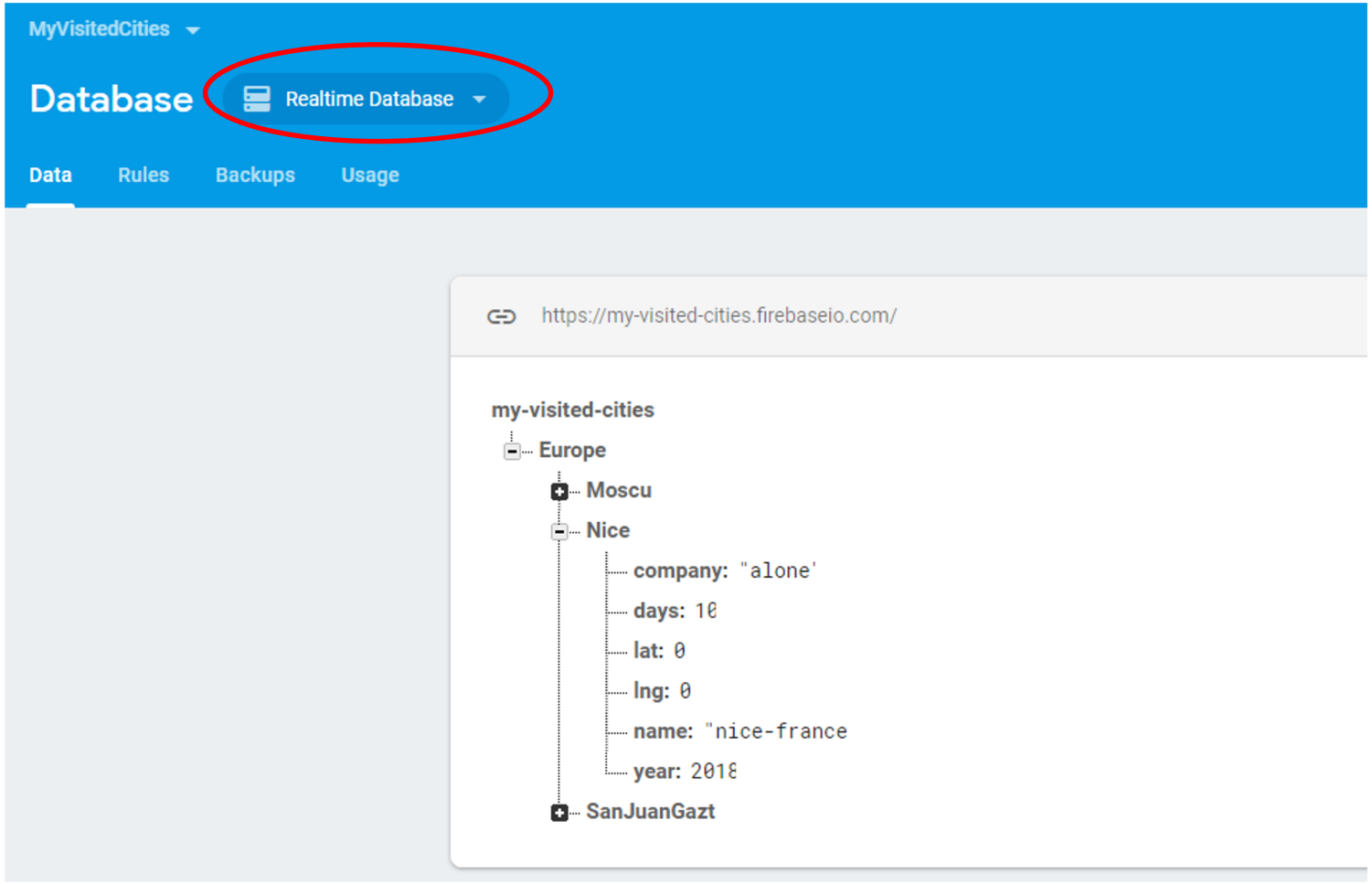
Client-first SDKs, with no servers to deploy and maintain Realtime updates

Free tier, then pay for what you use

RealTime VS Firestore DB

	REAL TIME	FIRESTORE
Data Model	One large JSON tree	**Collections of documents**
Offline support	yes	yes
queries		
Scalabitliy	Scaling requires sharding.	automatic
Querying	Queries can sort *or* filter on a property, but not both.	You can chain filters and combine filtering and sorting on a property in a single query.

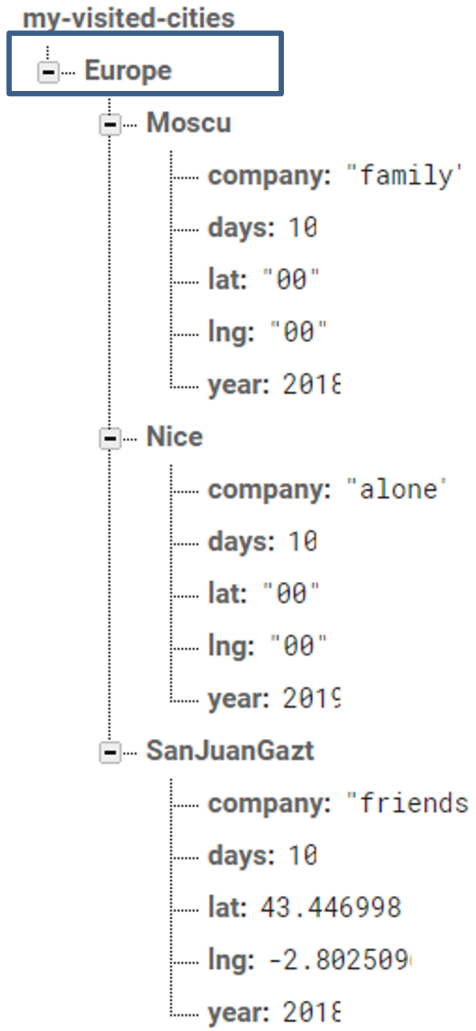
Firebase: Realtime DB



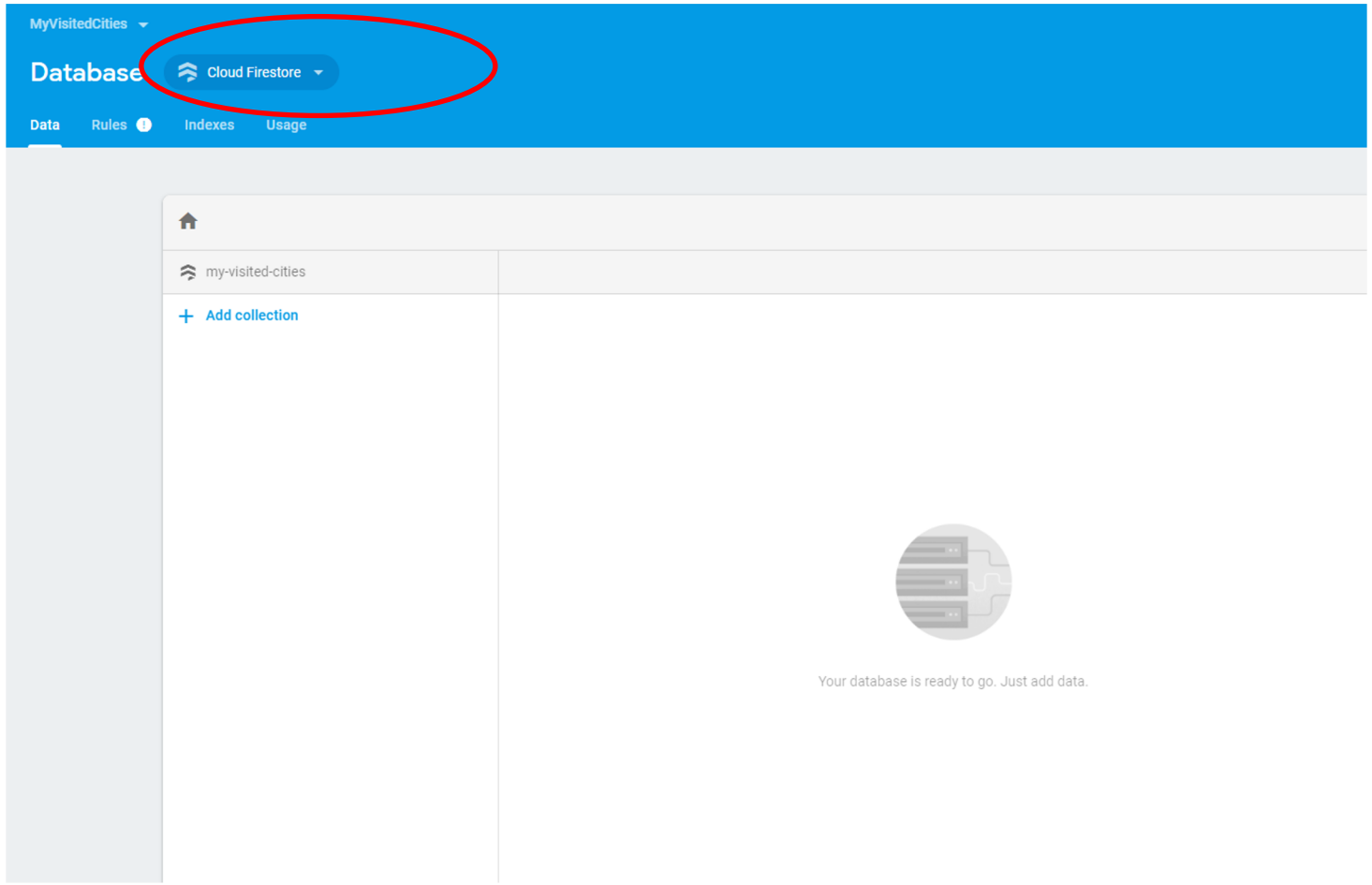
Retrieve data from *Realtime* DB

1. Grab the Firebase object for that data
2. bind an event to handle.
3. it will be notified initially, and on stage changes

```
fun retrieveChild(name:String){
    var fb = FirebaseDatabase.getInstance().reference
    val mdata = fb.child("Europe/$name")
    mdata.addValueEventListener(object:ValueEventListener{

        override fun onDataChange(data: DataSnapshot) {
            val mcity = data.getValue(City::class.java)
            val mlatitude = mcity!!.lat!!
            val text = "Latitude for $name is: ${mlatitude}"
            val duration = Toast.LENGTH_SHORT
            Toast.makeText(applicationContext, text, duration).show()
          }

        override fun onCancelled(databaseError: DatabaseError) {
            // Getting Post failed, log a message
            //Log.w(TAG, "loadPost:onCancelled", databaseError.toException())
            // ...
        }
    })

}
```



Firebase: Cloud Firestore

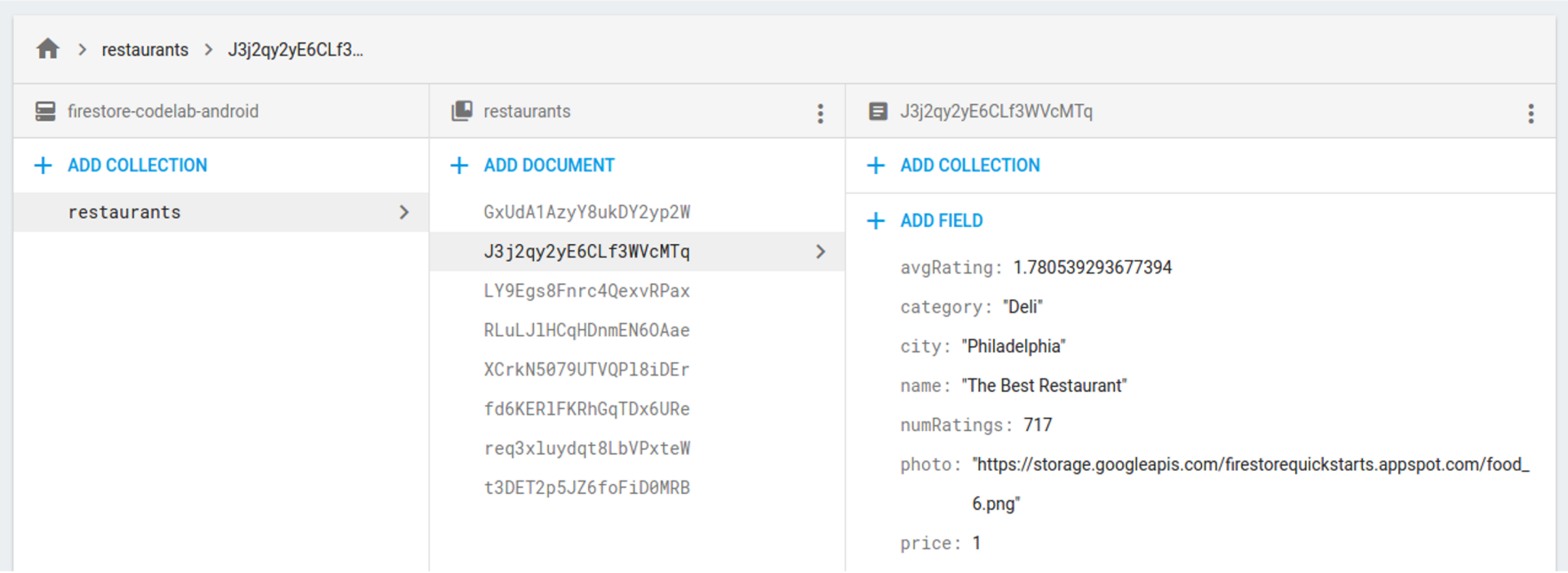


Firebase: Cloud Firestore

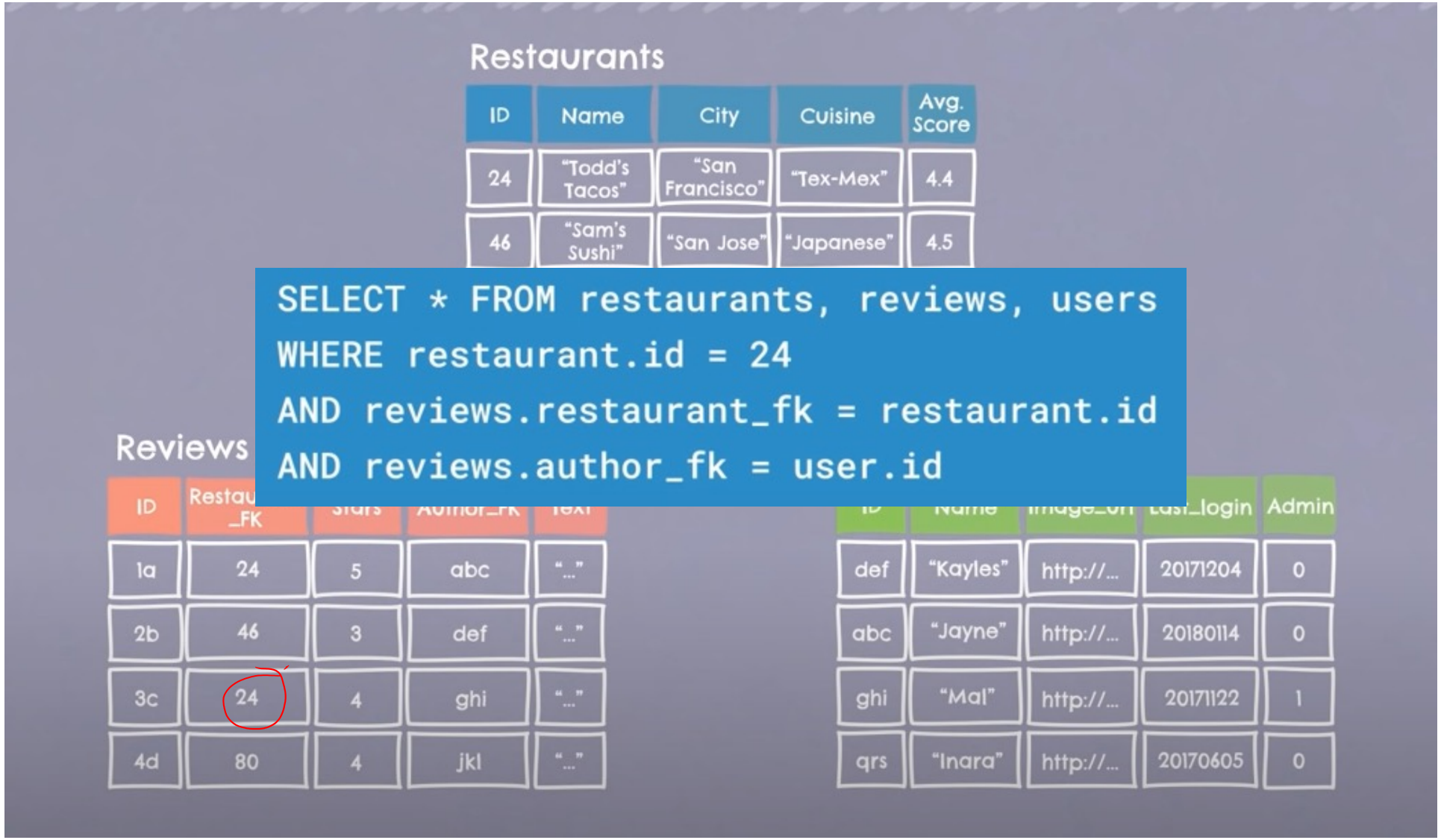
- On January 31st 2019, officially brought out of beta, making it an official product of the Firebase line-up.

- Successor to Firebase's original databasing system, Real-time Database, and allows for nested documents and fields rather than the tree-view provided in the Real-time Database.

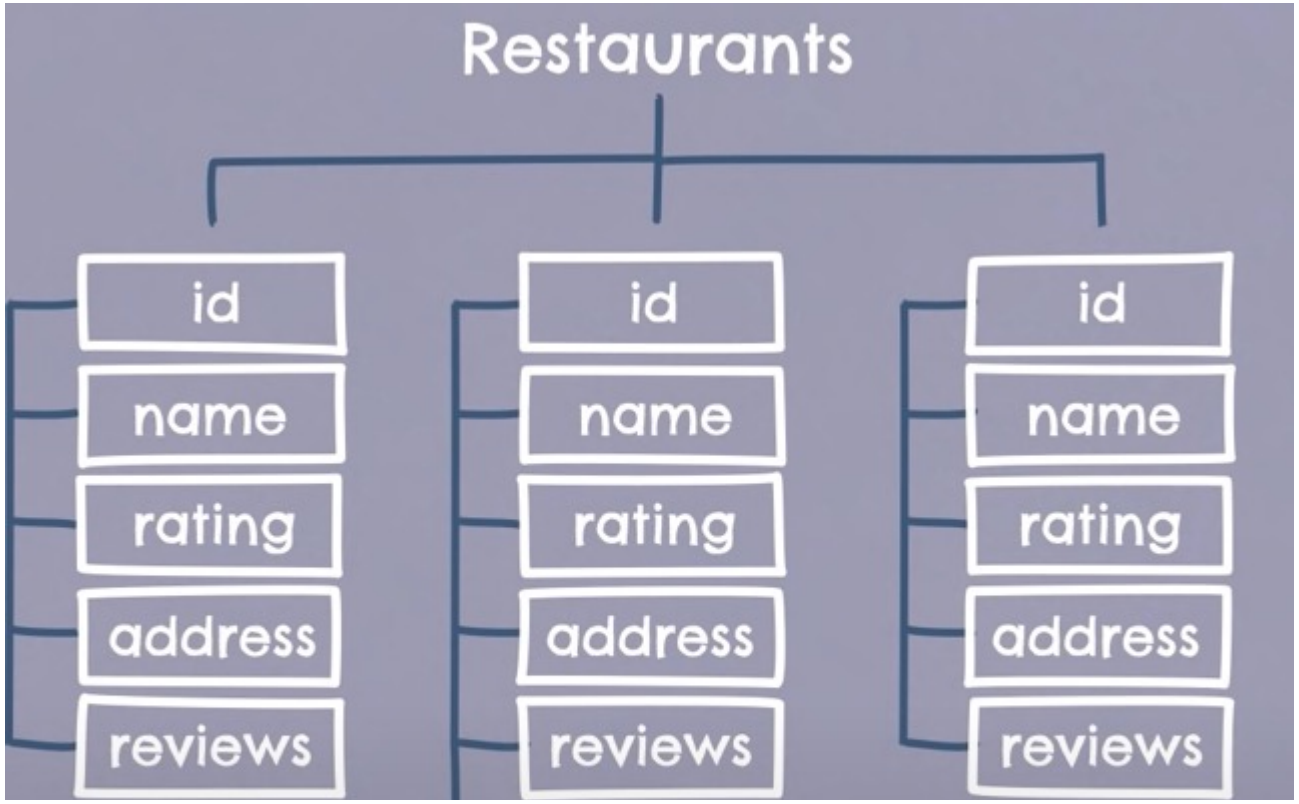
Firebase: Cloud Firestore



From Relational Databases …..



…. to schemaless databses



Firebase Data Model

Collections

Documents live in collections, which are simply containers for documents. For example, you could have a `users` collection to contain your various users, each represented by a document:

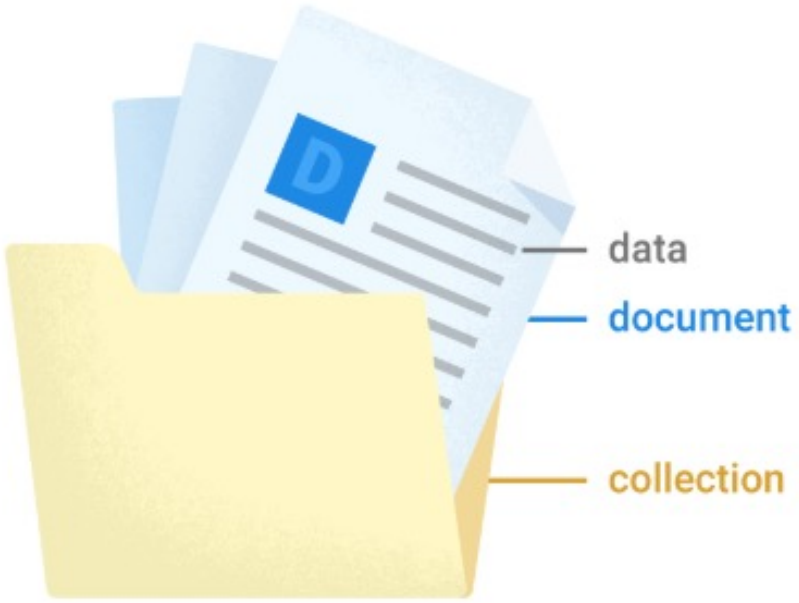
users

alovelace

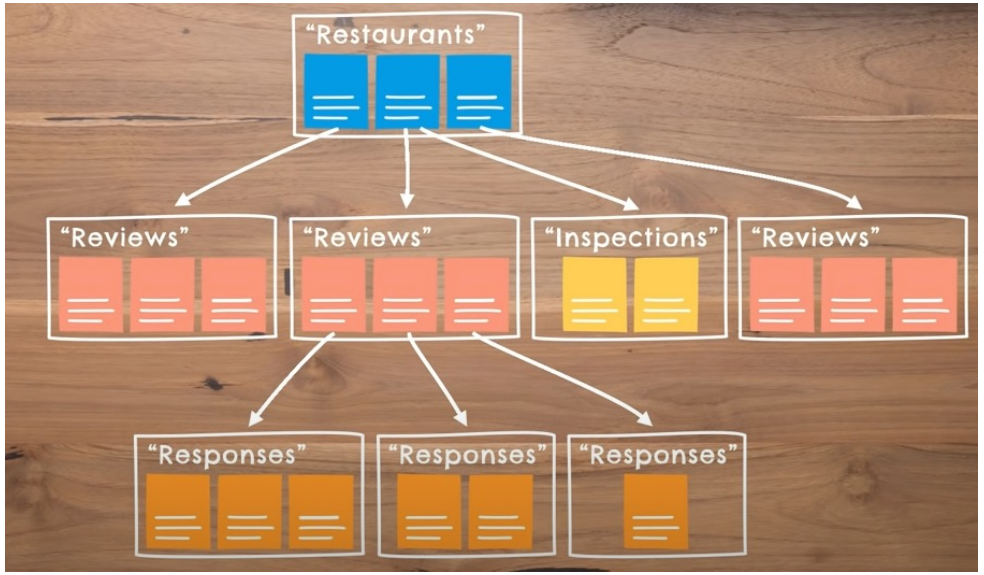
```
first : "Ada"
last : "Lovelace"
born : 1815
```

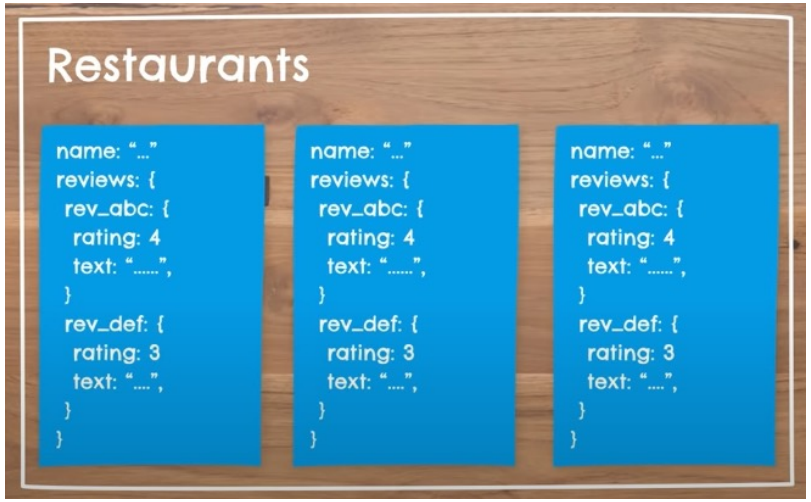
aturing

```
first : "Alan"
last : "Turing"
born : 1912
```



How to structure the data?





Rules for structuring the data

Rule #1: Documents have limits: 1Mb/document, 20,000 fields

Rule #2: You can’t retrieve a partial document

Rule #3: Queries are shallow

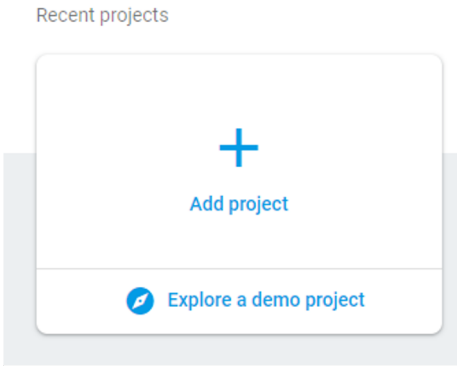
Rule #4: Billed by the number of reads and writes you perform

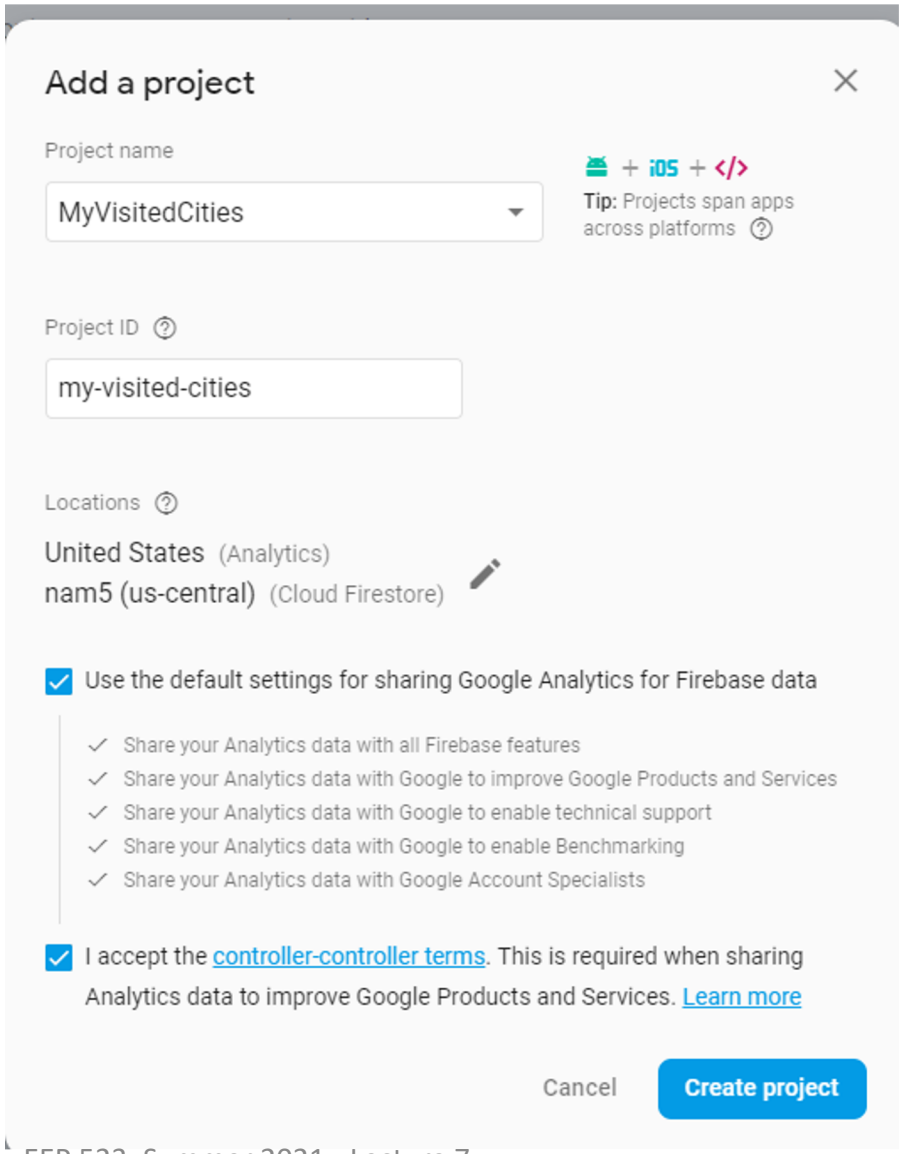
Rule #5: Queries find documents in collections

Firestore in your Android App

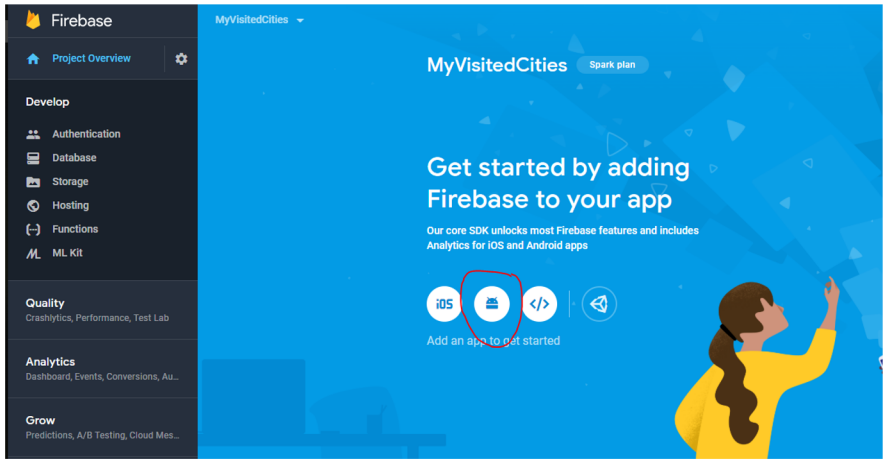
1. Go to Firebase online and create a new account
2. Add new project
3. Create and Android App where you are going to use Firebase
4. Add Firebase to your Android App
5. Connect your App to Firebase
   In Android IDE, go to Tools -> Firebase -> Firestore (or RealTime)
   Follow the steps
6. Go to Firebase online, and click on Database on the left panel, and create a database (select Firestore or RealTime)
7. All a collection to your database

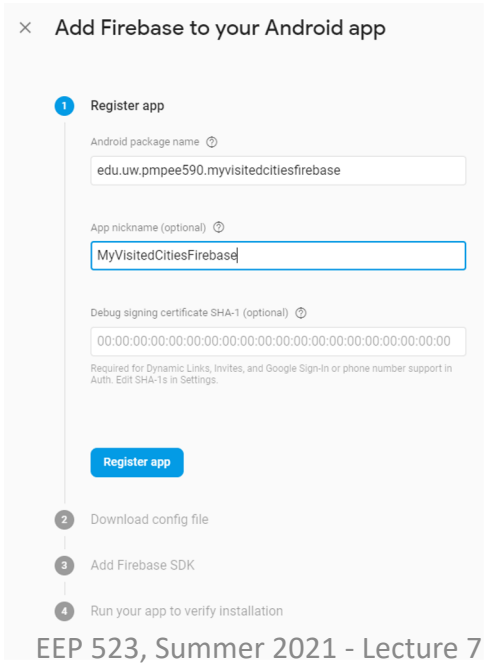
2. Add project



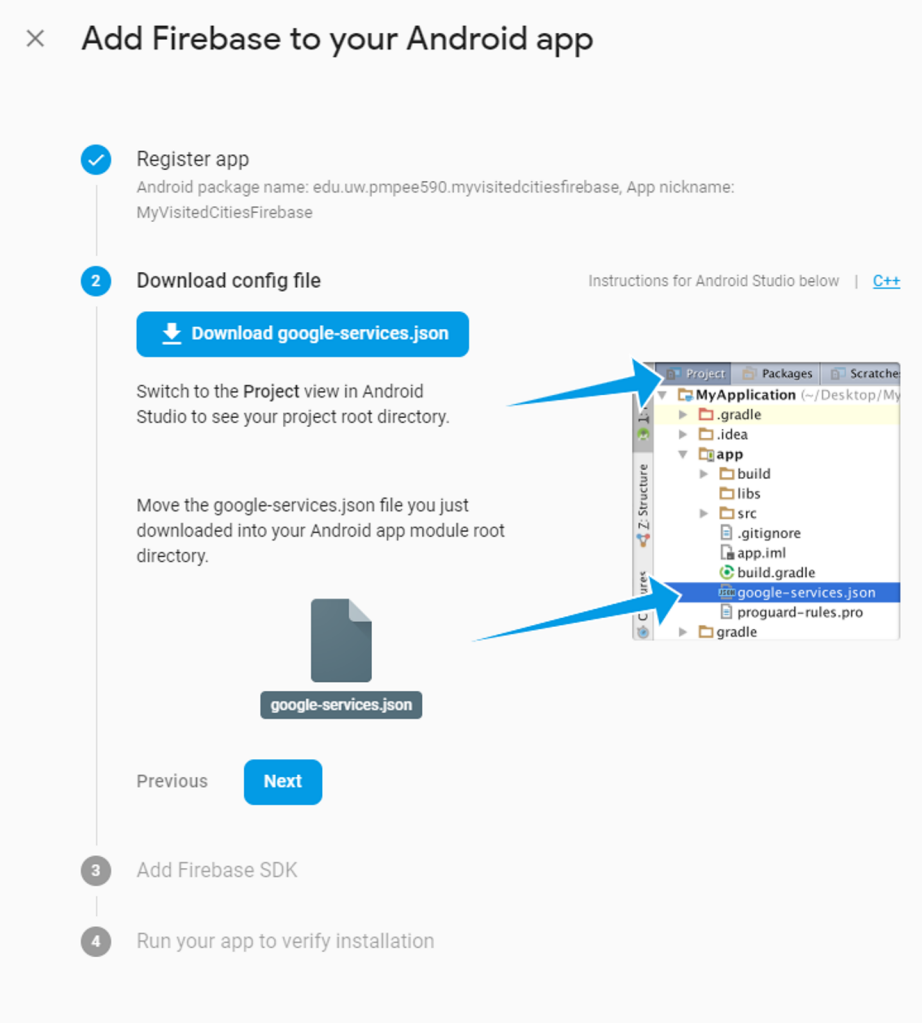


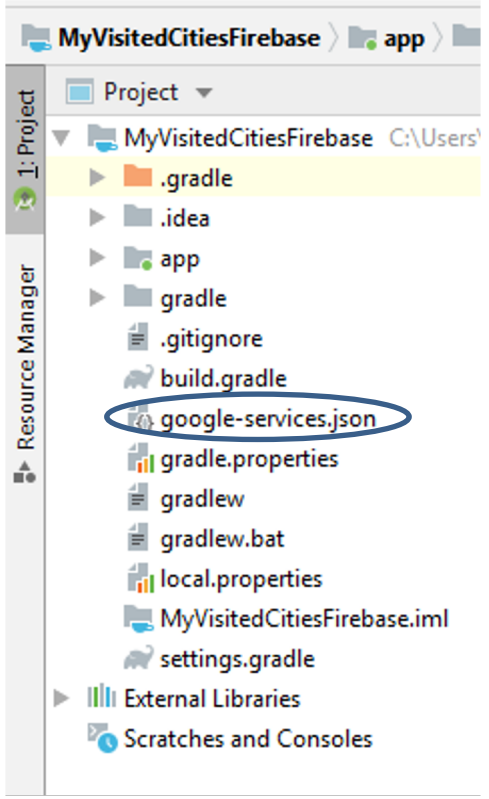
4. Add App to Firebase project



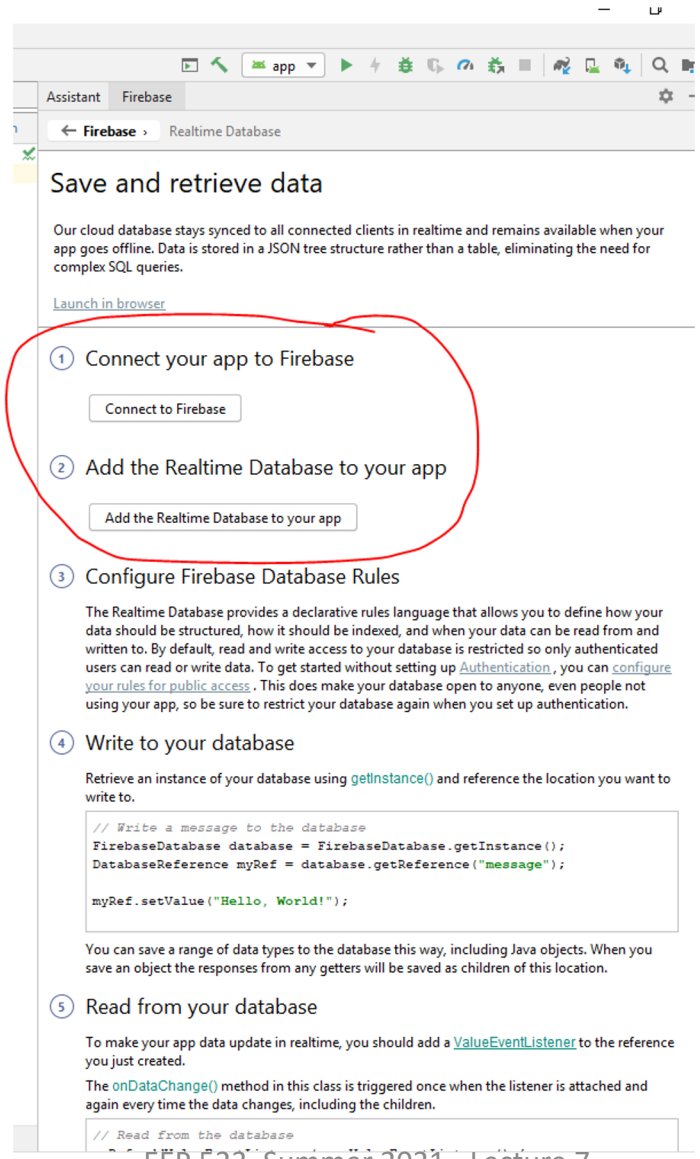


4. Add App to Firebase project

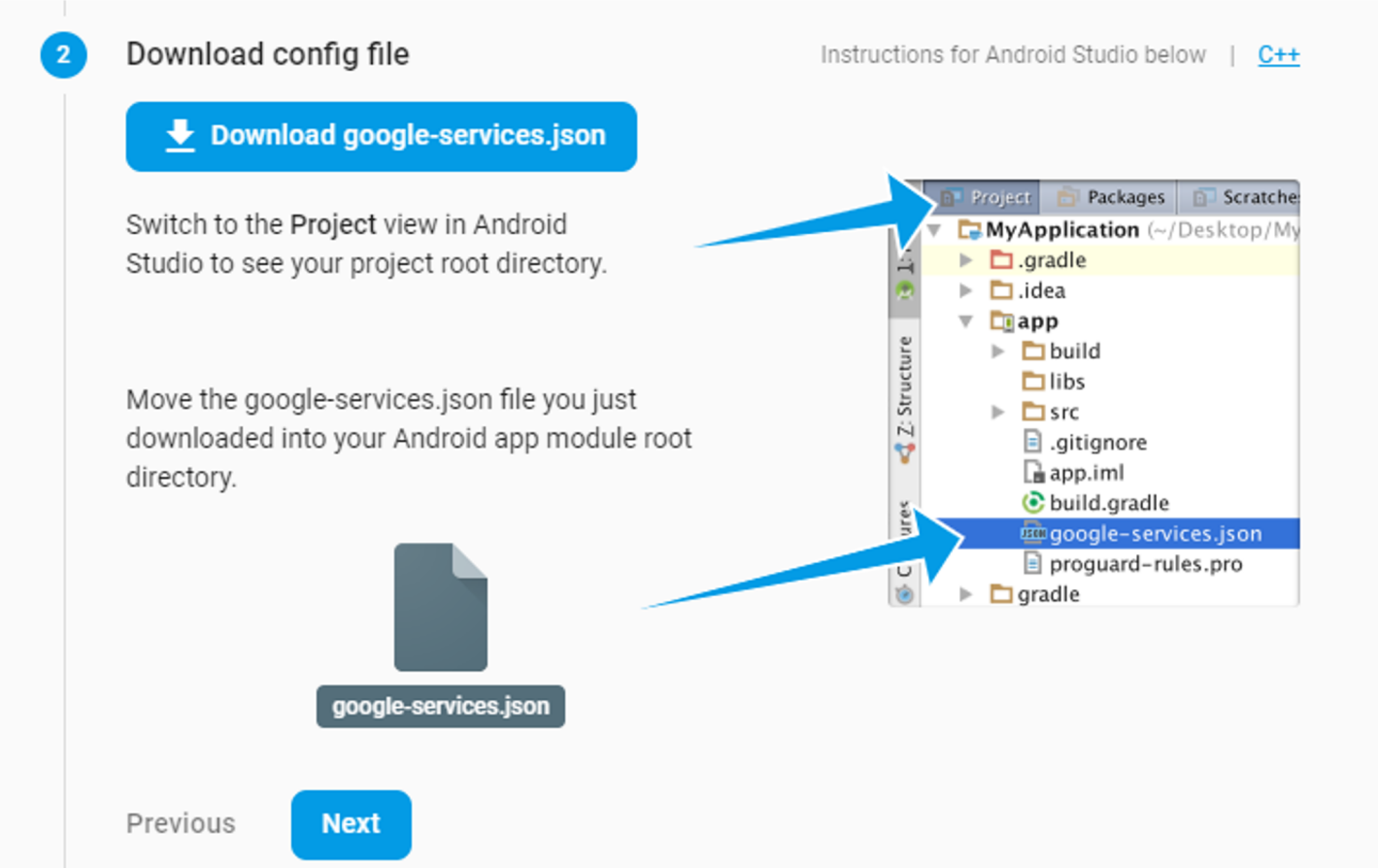




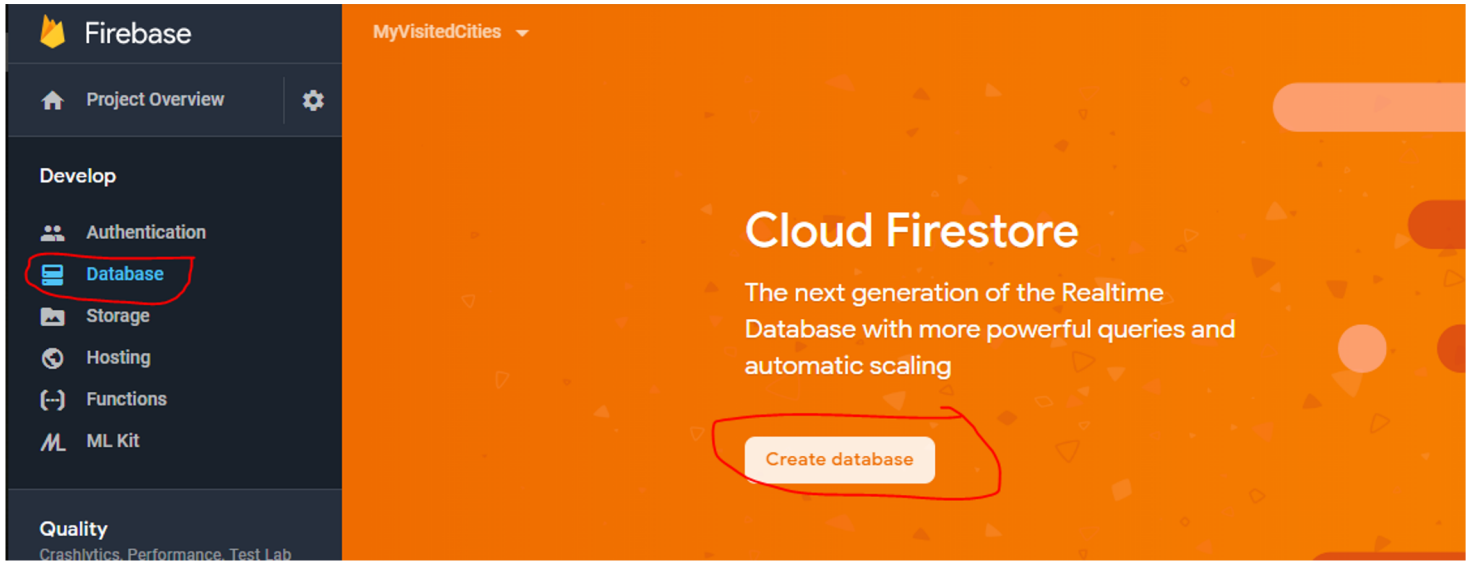
Add Firebase to you Android App

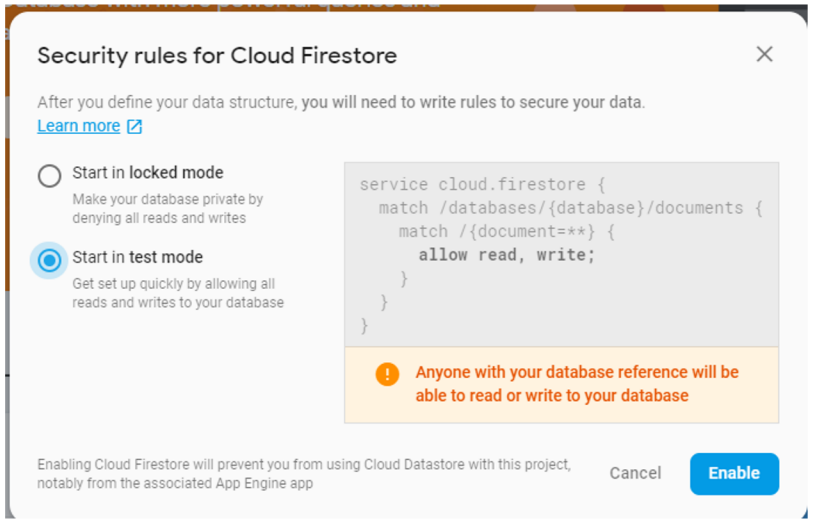


Add Firebase to you Android App

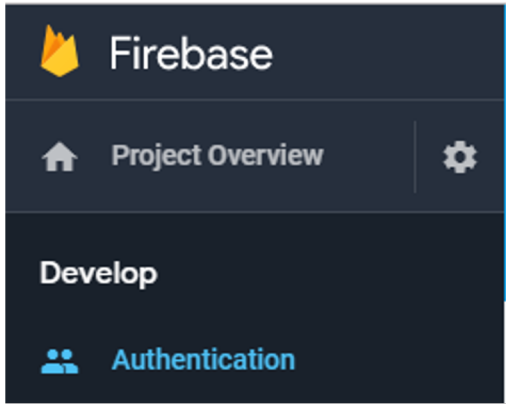


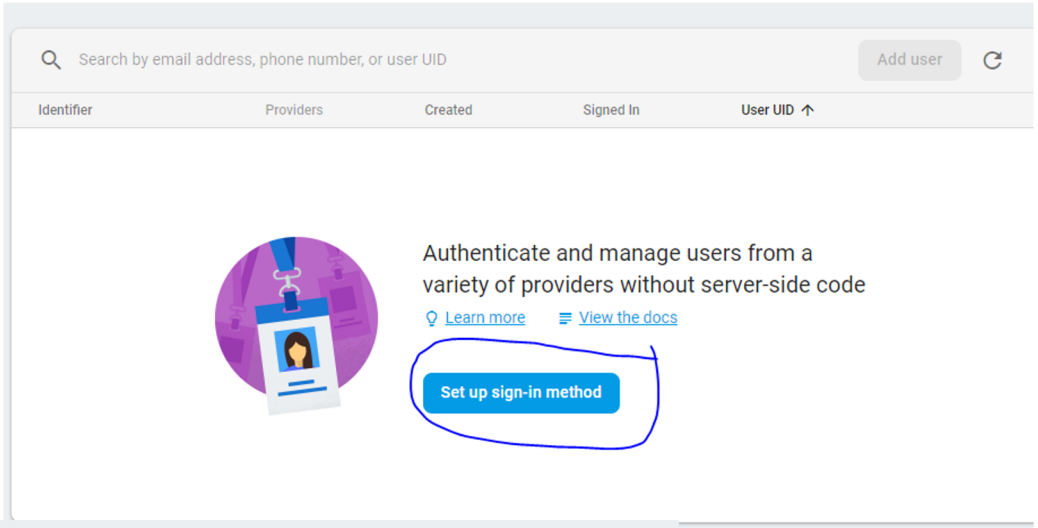
Create Real Time/ Firestore

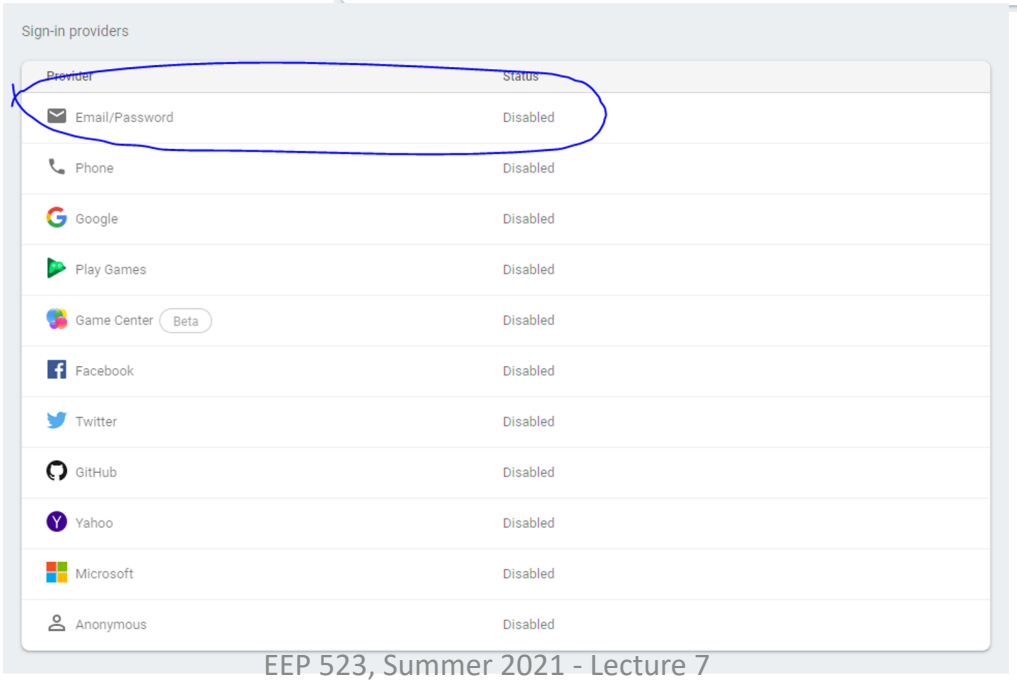




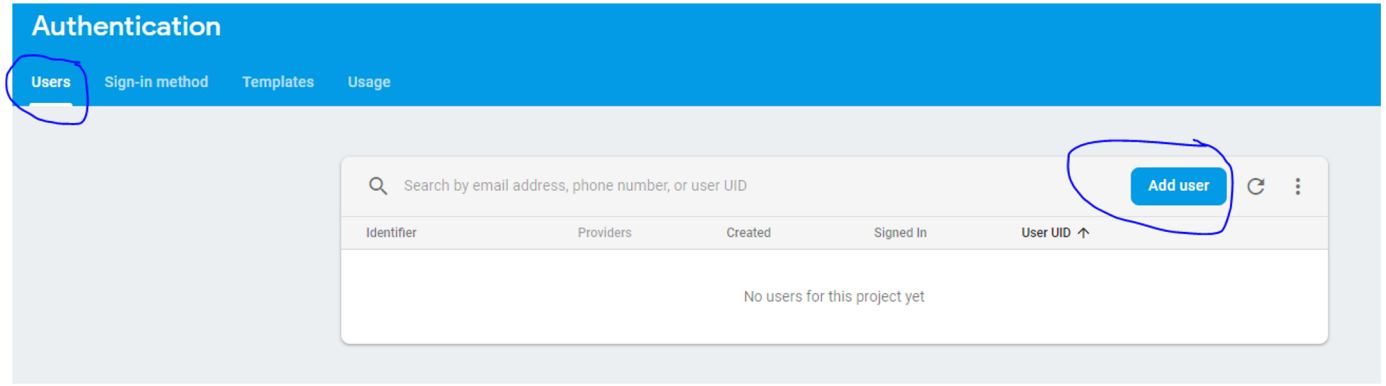
Authentication

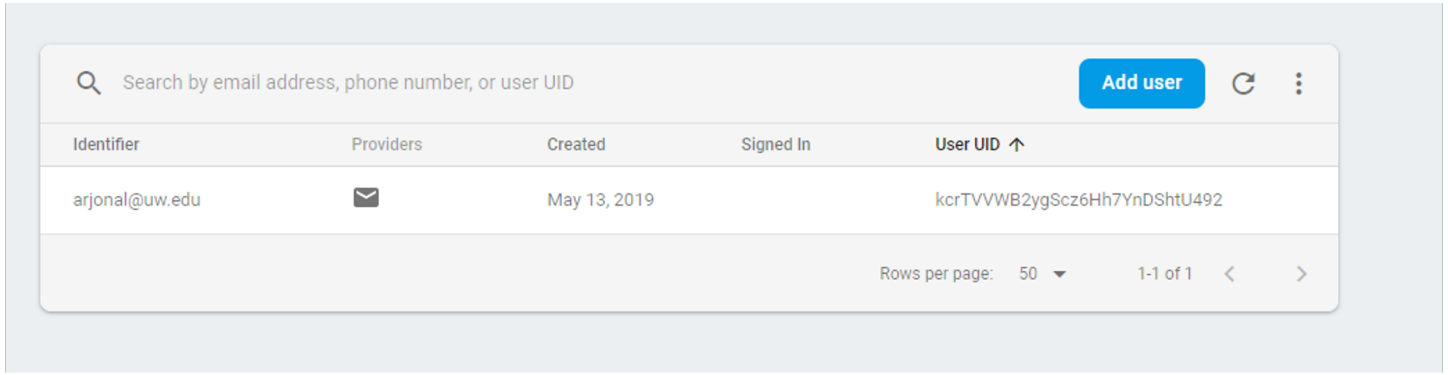






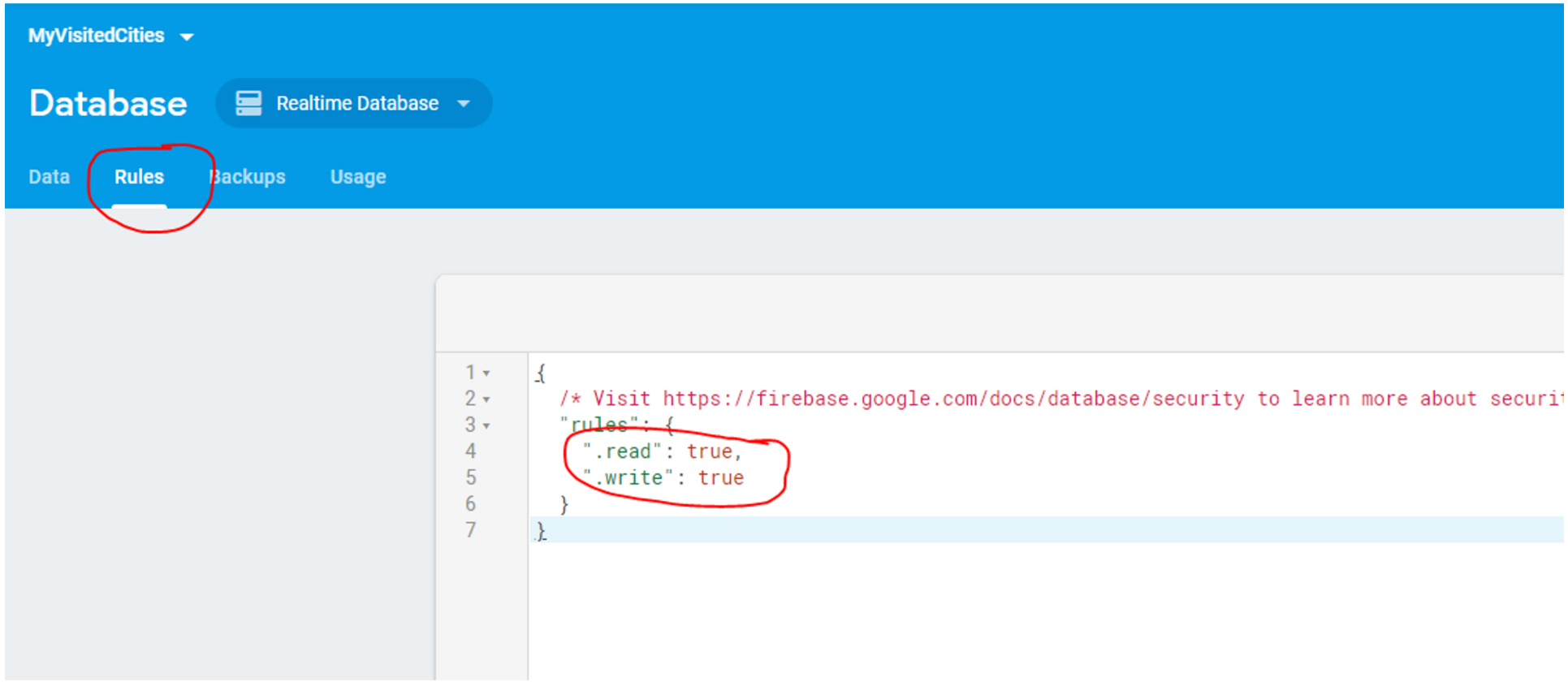
Authentication





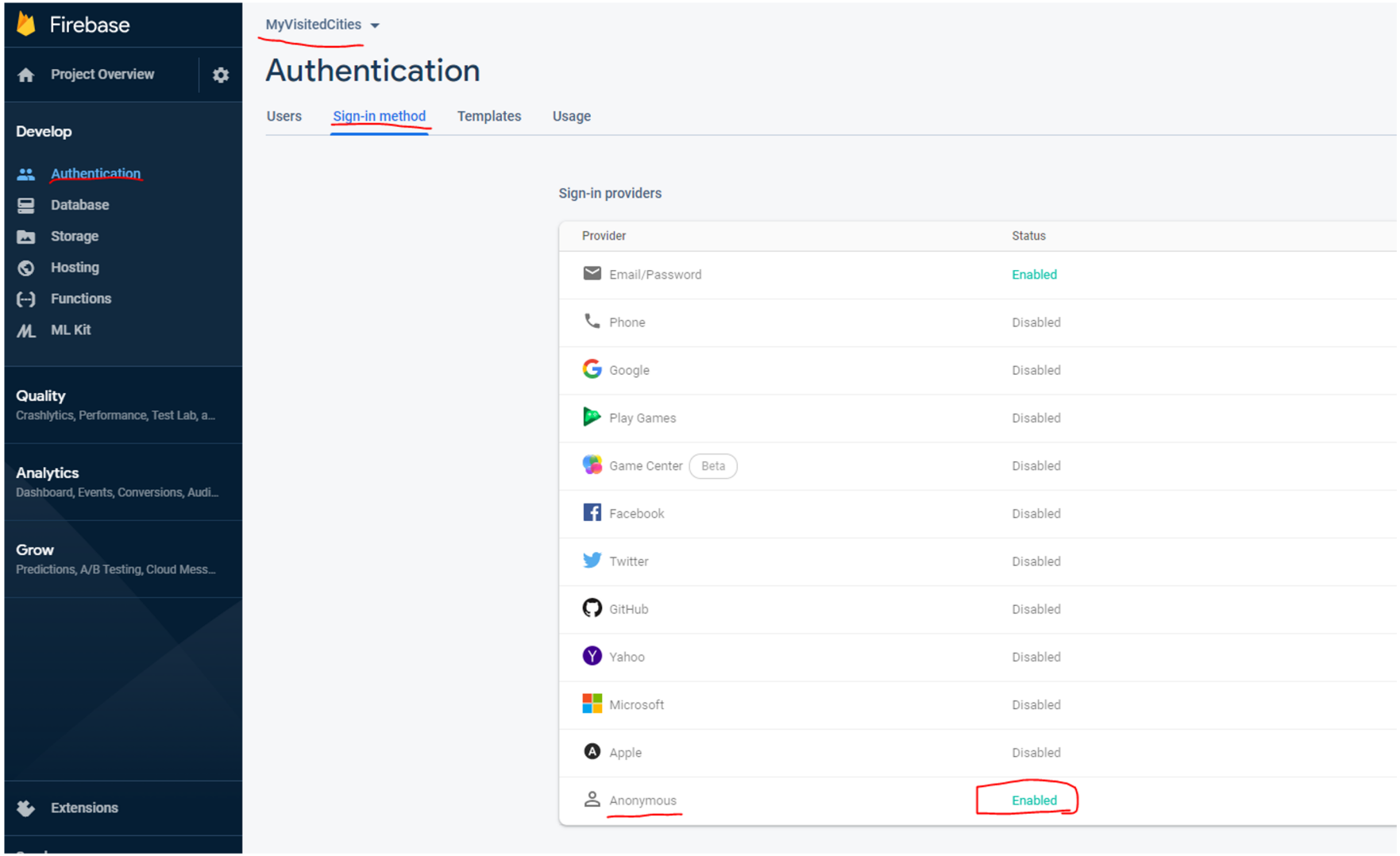

```
implementation 'com.google.firebase:firebase-core:16.0.7'
implementation 'com.google.firebase:firebase-auth:16.1.0'
```

Authentication




```
implementation 'com.google.firebase:firebase-core:16.0.7'
implementation 'com.google.firebase:firebase-auth:16.1.0'
```

Authentication: anonymous



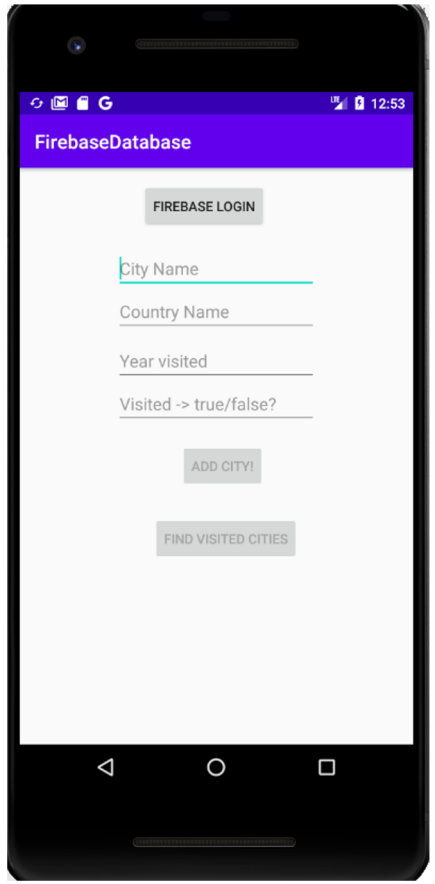
Authentication from App

```
companion object {
    const val FIREBASE_USERNAME = ""
    const val FIREBASE_PASSWORD = ""
}
```

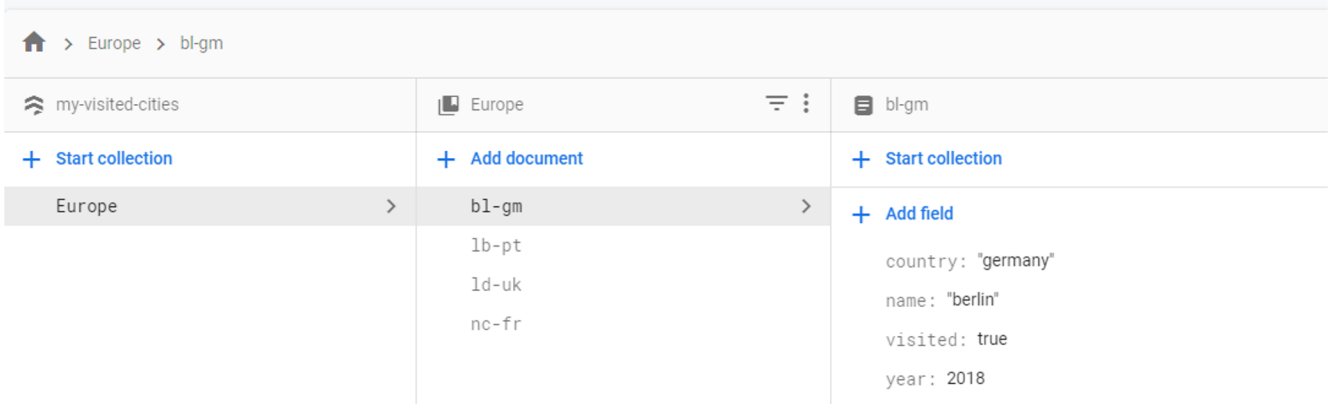
```
private lateinit var mAuth: FirebaseAuth

//private lateinit var database: DatabaseReference
var database = FirebaseDatabase.getInstance().reference
```

```
    //Connect to the Firebase RealTime DataBase
    fun connectFirebaseRTDB(v: View){
        //Connect to firebase

        FirebaseApp.initializeApp(this)
        mAuth = FirebaseAuth.getInstance()
        mAuth.signInWithEmailAndPassword(FIREBASE_USERNAME, FIREBASE_PASSWORD)

}
```

Example App with Firestore



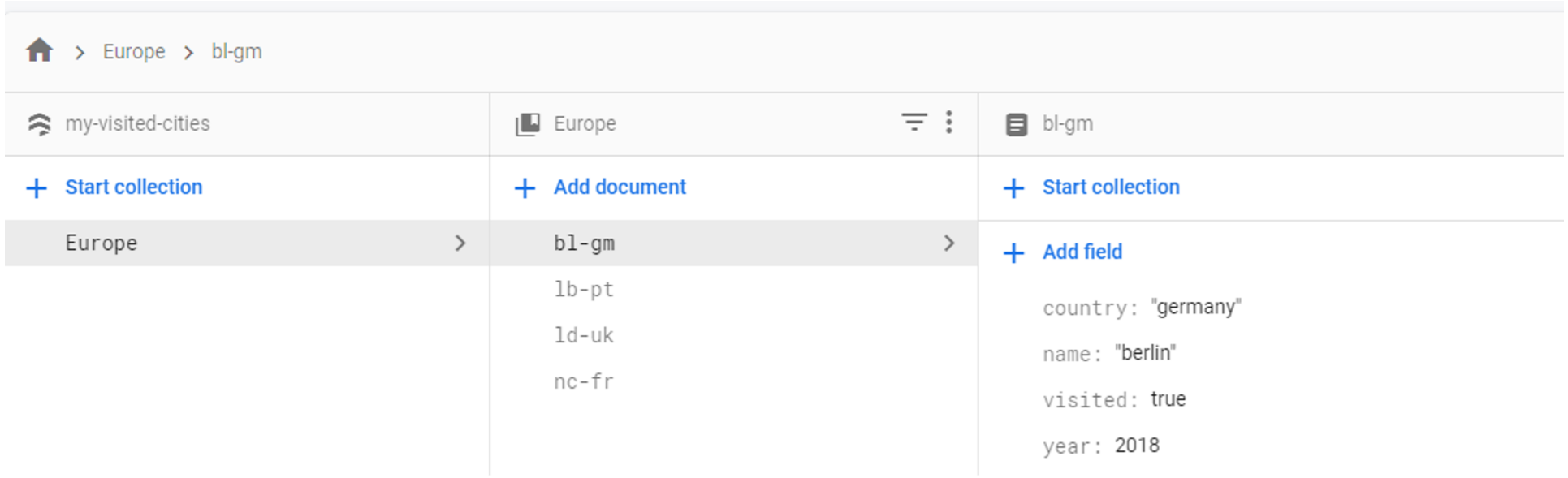
Example in Canvas: ***FirebaseDatabase***



Retrieve data from *Firestore*

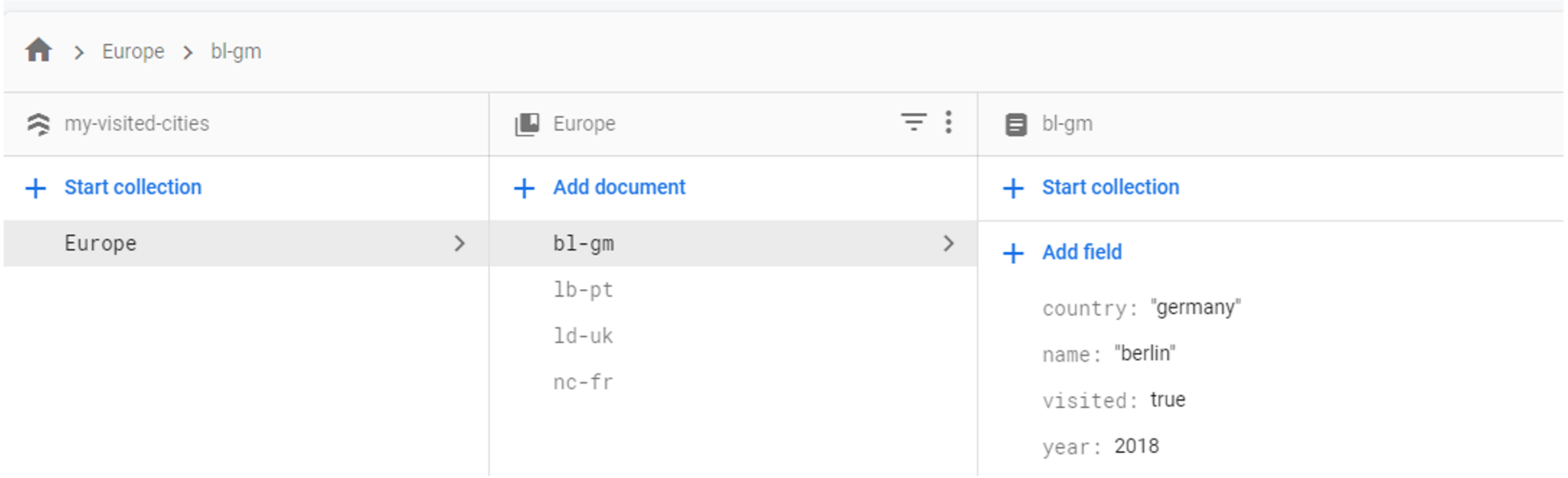
There are two ways to retrieve data stored in Cloud Firestore. Either of these methods can be used with documents, collections of documents, or the results of queries:

1. Call a method to get the data.
2. Set a listener to receive data-change events.



1. Get the data once

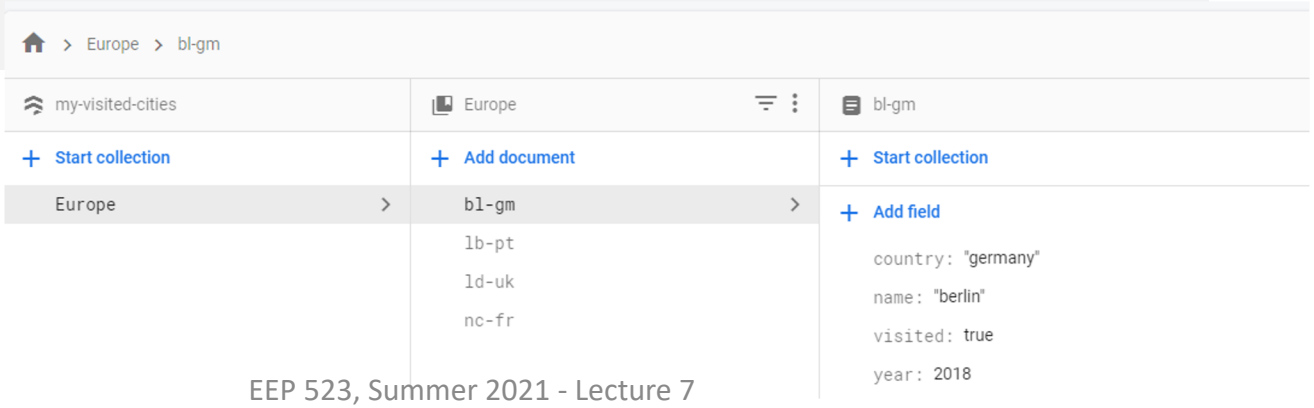
```
fun readDataExample(){
    val cityReference = db.collection("Europe").document("nc-fr")
    cityReference.get().addOnSuccessListener { documentSnapshot ->
        val newCity = documentSnapshot.toObject(City::class.java)

        var cityName = newCity!!.name
       Toast.makeText(applicationContext, "$cityName", Toast.LENGTH_SHORT).show()
    }
}
```



2. Listen for RealTime changes

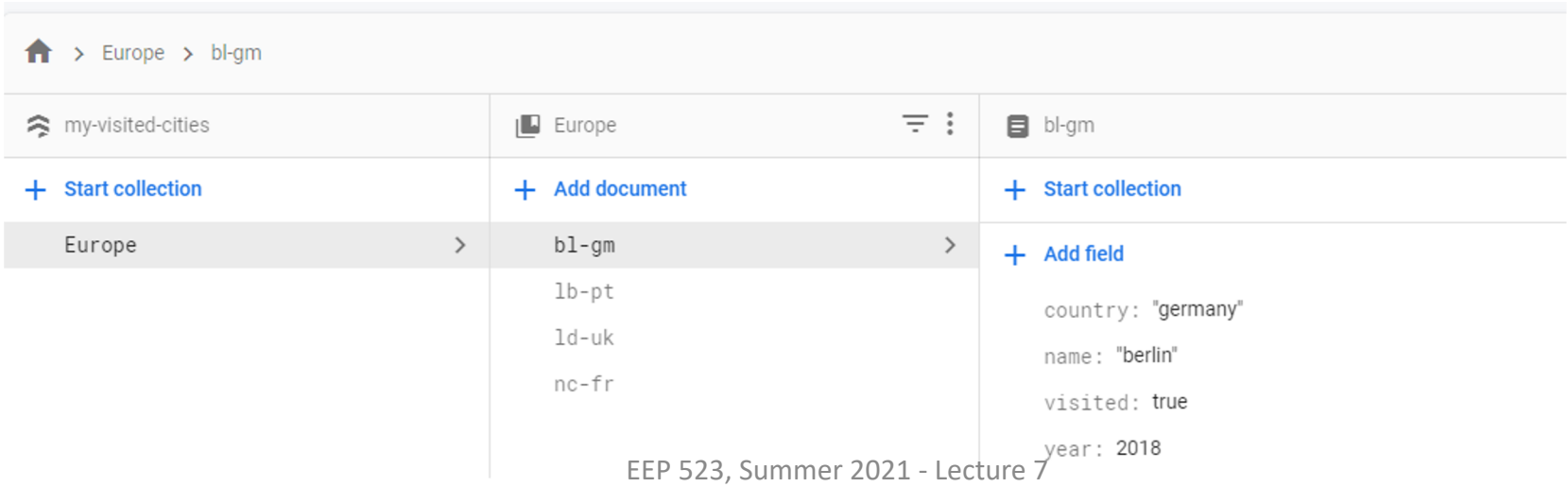
You can listen to a document with the **onSnapshot()** method. An initial call using the callback you provide creates a document snapshot immediately with the current contents of the single document. Then, each time the contents change, another call updates the document snapshot.

```
fun listenForRTChanagesExample(){
    val cityReference = db.collection("Europe").document("nc-fr")
    cityReference.addSnapshotListener { snapshot, e ->
        if (e != null) {
            return@addSnapshotListener
        }
        val source = if (snapshot != null && snapshot.metadata.hasPendingWrites())
            "Local"
        else
            "Server"
        if (snapshot != null && snapshot.exists()) {
            Toast.makeText(this,"change in nice document",Toast.LENGTH_LONG).show()
            Log.d("tag", "$source data: ${snapshot.data}")
        } else {
           Log.d("tag", "$source data: null")
        }
    }
}
```



Filter *visited* cities

```
////////////////////////////////////////////////////////////////////////////
//Filter cities visited = true
fun findCityYear(v: View){
    db.collection("Europe")
        .whereEqualTo("visited", true)
        .get()
        .addOnSuccessListener { documents ->
            for (document in documents) {
                var theName = document.get("name")
               Log.d("tag", "${document.id} => ${document.data}")

                resultView.append(theName.toString() + ", ")

            }
        }
        .addOnFailureListener { exception ->
            Log.w("tag", "Error getting documents: ", exception)
        }

}
```

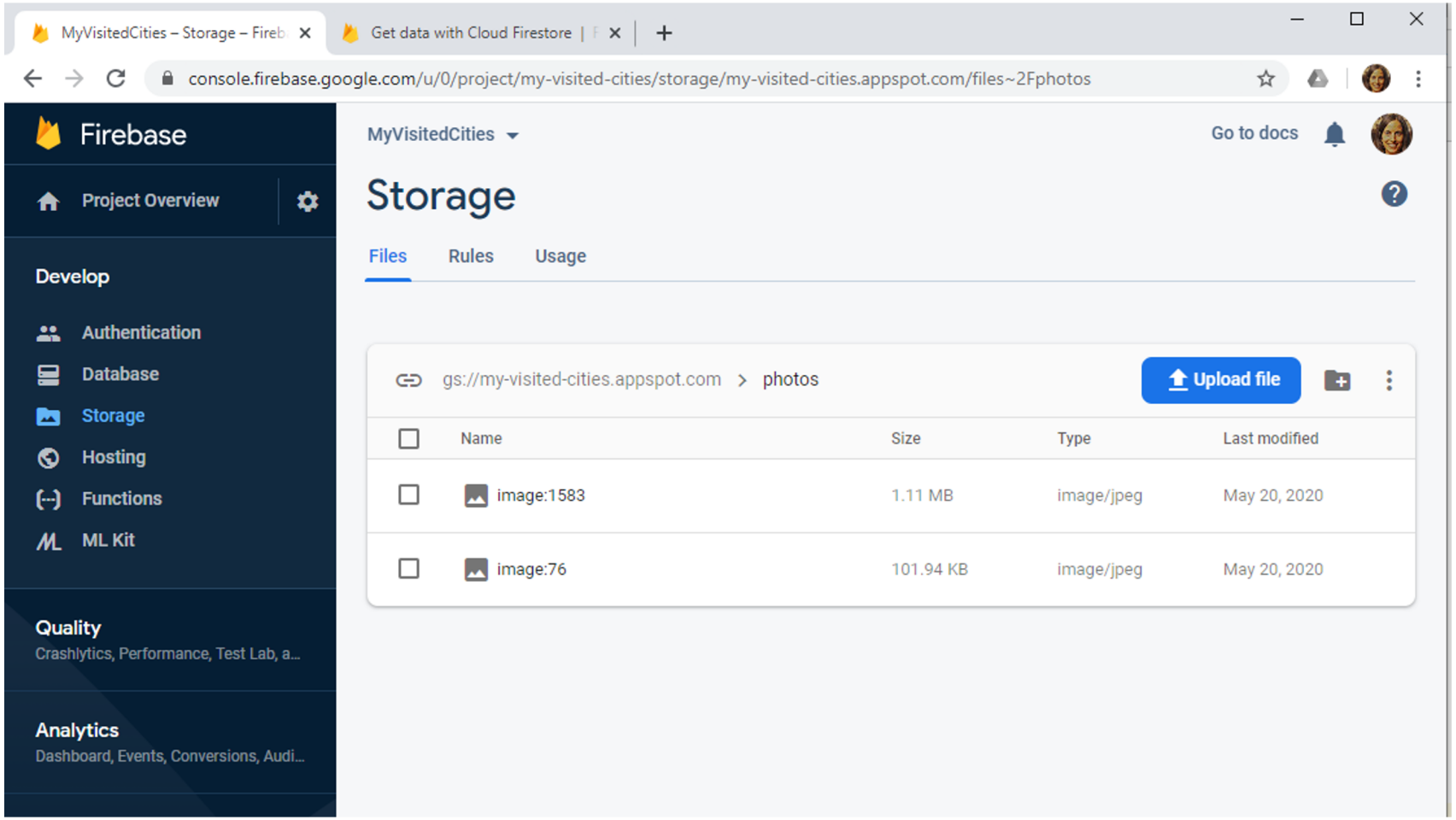


Save/Restore Pictures

If the picture is in the phone gallery, you can use the DB to just save the URI (must be unique)

```
UUID.randomUUID().toString()) // unique ID
```

Save/Restore pictures with Firebase Storage



Upload Picture File

Example: ***FirebaseUpDownLoad***

1. Create a service to upload the picture
2. From the main activity, select the picture, and start the service with an intent passing the picture URI
3. Upload the picture in the /photos path: refer to

```
uploadFromUri(fileUri: Uri) {
}
```

```
// Get a reference to store file at photos/<FILENAME>.jpg
fileUri.lastPathSegment?.let {
    val photoRef = storageRef.child("photos")
        .child(it)

    // Upload file to Firebase Storage
    Log.d(TAG, "uploadFromUri:dst:" + photoRef.path)
    photoRef.putFile(fileUri).addOnProgressListener { taskSnapshot ->

    }
```

Upload Picture File

4. (optional) send a broadcast receiver to the MainActivity to notify success upload

```
broadcastUploadFinished(downloadUri, fileUri)
```

Example: ***FirebaseUpDownLoad***

```
private fun broadcastUploadFinished(downloadUrl: Uri?, fileUri: Uri?): Boolean {
    val success = downloadUrl != null
    val action = if (success) UPLOAD_COMPLETED else UPLOAD_ERROR
    val broadcast = Intent(action)
        .putExtra(EXTRA_DOWNLOAD_URL, downloadUrl)
        .putExtra(EXTRA_FILE_URI, fileUri)
    return LocalBroadcastManager.getInstance(applicationContext)
        .sendBroadcast(broadcast)
}
```

Upload Picture File

Example: ***FirebaseUpDownLoad***

4. (optional) Receive the Broadcast message in the MainActivity

```
// Local broadcast receiver
broadcastReceiver = object : BroadcastReceiver() {
    override fun onReceive(context: Context, intent: Intent) {
        Log.d(TAG, "onReceive:$intent")
        hideProgressBar()
        when (intent.action) {
            ServiceUpload.UPLOAD_COMPLETED, ServiceUpload.UPLOAD_ERROR ->
     onUploadResultIntent(intent)
        }
    }
}



private fun onUploadResultIntent(intent: Intent) {
    // Got a new intent from MyUploadService with a success or failure
    downloadUrl = intent.getParcelableExtra(ServiceUpload.EXTRA_DOWNLOAD_URL)
    fileUri = intent.getParcelableExtra(ServiceUpload.EXTRA_FILE_URI)
    if (downloadUrl != null) {
        pictureDownloadUri.text = downloadUrl.toString()}
}
```

Download picture with Glide

Example: ***FirebaseUpDownLoad***

```
private lateinit var storageRef: StorageReference

private fun beginDownload() {
    // Download directly from StorageReference using Glide
    storageRef = Firebase.storage.reference
    // Create a reference with an initial file path and name
    val pathReference = storageRef.child("photos/image:1204")


    Glide.with(this /* context */)
        .load(pathReference)
        .into(imageView)
}
```

Download Picture with Glide

activity_main.xml | MainActivity.kt | MyAppGlideModule.java | AndroidManifest.xml | ServiceUpload.k

```java
package edu.uw.eep523.firebasepicturesupdownload;

import android.content.Context;

import com.bumptech.glide.Glide;
import com.bumptech.glide.Registry;
import com.bumptech.glide.annotation.GlideModule;
import com.bumptech.glide.module.AppGlideModule;
import com.firebase.ui.storage.images.FirebaseImageLoader;
import com.google.firebase.storage.StorageReference;

import java.io.InputStream;

@GlideModule
public class MyAppGlideModule extends AppGlideModule {

    @Override
    public void registerComponents(Context context, Glide glide, Registry registry) {
        // Register FirebaseImageLoader to handle StorageReference
        registry.append(StorageReference.class, InputStream.class,
                new FirebaseImageLoader.Factory());
    }
}
```

Anonymous Sign-In

Example: ***FirebaseUpDownLoad***

```
private fun signInAnonymously() {
    // Sign in anonymously. Authentication is required to read or write from
Firebase Storage.
    showProgressBar(getString(R.string.progress_auth))
    auth.signInAnonymously()
        .addOnSuccessListener(this) { authResult ->
            Log.d(TAG, "signInAnonymously:SUCCESS")
            hideProgressBar()
            updateUI(authResult.user)
        }
        .addOnFailureListener(this) { exception ->
            Log.e(TAG, "signInAnonymously:FAILURE", exception)
            hideProgressBar()
            updateUI(null)
        }
}
```

Access data offline

x Not online- 1st-class Platform
✓ Yes: offline tolerant

- Offline writes: exponential back-off
- Offline reads: cached data (how big is the cache?)

Android Web Based Content

EE P 523, Lecture 7

Android Web-based content

- You shouldn't develop an Android app simply as a means to view your website.
- Rather, the web pages you embed in your app should be designed especially for that environment

Best Practices

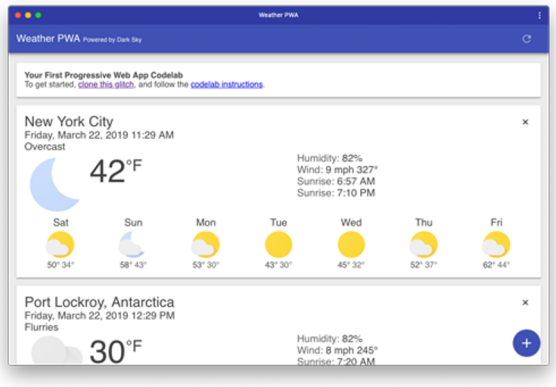
1. Redirect mobile devices to a dedicated mobile version of your website

1. **Use a vertical linear layout**
   Avoid the need for the user to scroll left and right while navigating your web page.
   Scrolling up and down is easier for the user and makes your web page simpler.

3. **Set the layout height and width to** match_parent
   ensures that your app's views are sized correctly.
   Discourage setting the height to wrap_content

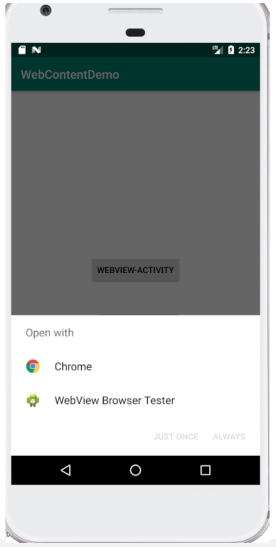
Android Web-based content

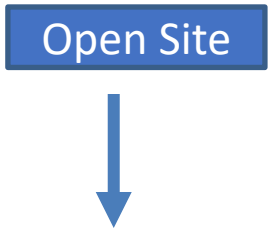
- Progressive web app (PWA): send users to a mobile site.
  - **HTML, CSS, JavaScript**
  - Installable, app-like experience on desktop and mobile that are built and **delivered directly via the web**.
  - They're web apps that are **fast and reliable**.
  - They're web apps that work in **any browser**.



Android Web-based content

- Send an intent to installed web browsers:
  display third-party web content

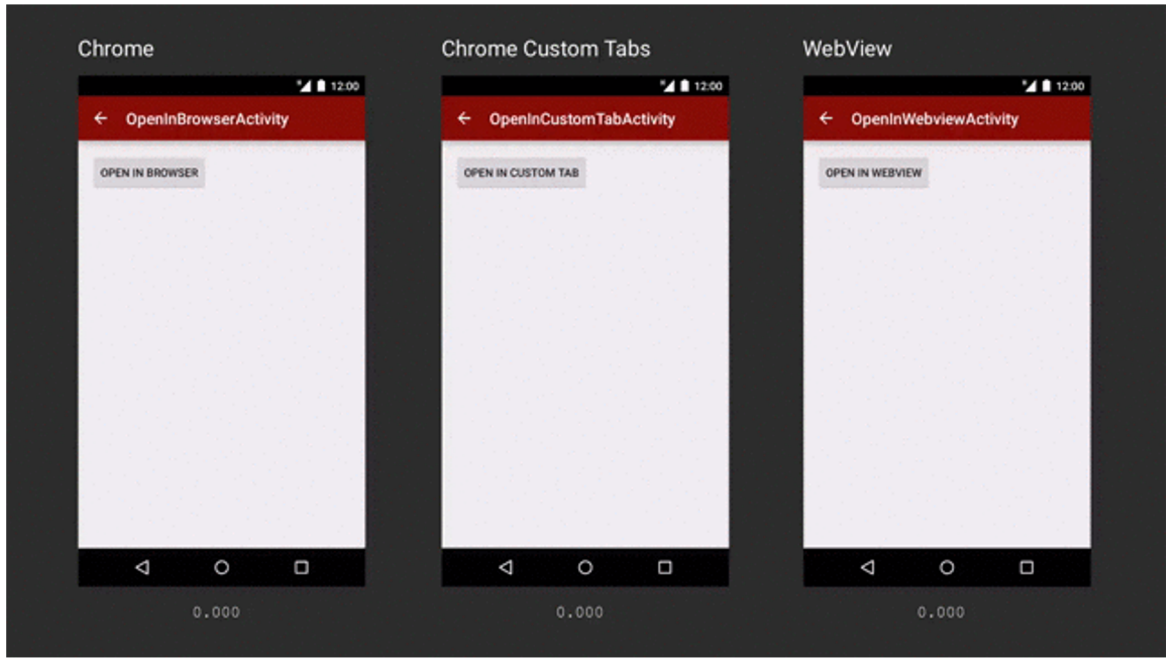





```
fun openWebPage(v : View) {
    val webpage: Uri = Uri.parse("https://www.hbo.com/game-of-thrones/cast-and-crew")
    val intent = Intent(Intent.ACTION_VIEW, webpage)
    if (intent.resolveActivity(packageManager) != null) {
        startActivity(intent)
    }
}
```

Web Based Content

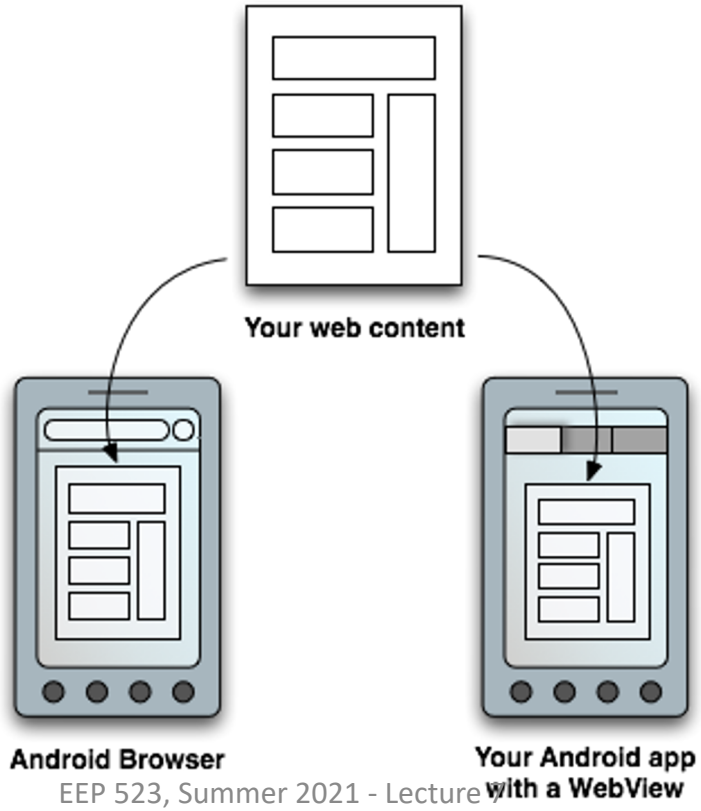
- Chrome Custom Tabs (link):
  - avoid leaving your app to open the browser
  - customize the browser's UI



WebView

You can make your web content available to users in two ways:

1. Traditional web browser
2. Android application, by including a **WebView** in the layout.



Chrome Custom Tabs or WebView?

- The WebView is good solution if you are hosting your own content inside your app.
- If your app directs people to URLs outside your domain, it is recommended that you use **Chrome Custom Tabs**

Chrome Custom Tabs

- **Simple to implement**. No need to build code to manage requests, permission grants or cookie stores.
- **UI customization**:
  - Toolbar color
  - Action button
  - Custom menu items
  - Custom in/out animations
  - Bottom toolbar
- Navigation awareness: the browser delivers a callback to the application upon an external navigation.
- Security: the browser uses **Google's Safe Browsing** to protect the user and the device from dangerous sites.

Chrome Custom Tabs or WebView?

- Performance optimization:
  - Pre-warming of the Browser in the background, while avoiding stealing resources from the application.
  - Providing a likely URL in advance to the browser, which may perform speculative work, speeding up page load time.

- **Lifecycle management**: the browser prevents the application from being evicted by the system while on top of it, by raising its importance to the "foreground" level.

- Shared cookie jar and permissions model so users don't have to log in to sites they are already connected to, or re-grant permissions they have already granted.

- If the user has turned on Data Saver, they will still benefit from it.

- **Synchronized AutoComplete** across devices for better form completion.

- **Quickly return to app with a single tap.**

WebView

- Extension of Android's View class
- Allows you to display **web pages as a part of your activity layout**.
- It does *not* include any features of a fully developed web browser, such as navigation controls or an address bar.
- All that WebView does, by default, is show a web page.

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
          package="edu.uw.pmpee590.webviewdemo">


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

Adding a WebView to your App

1. Insert a View in the xml

In your activity xml file

```
<WebView
        android:id="@+id/mwebview"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
/>
```

In your activity kotlin class

```
val myWebView_xml: WebView = findViewById(R.id.mwebview)
myWebView_xml.loadUrl("https://www.youtube.com/watch?v=rlR4PJn8b8I#action=share")
```

Adding a WebView to your App

2. At runtime

```
val myWebView = WebView(applicationContext)
setContentView(myWebView)
myWebView.loadUrl("https://www.hbo.com/game-of-thrones/cast-and-crew")
myWebView.webViewClient = WebViewClient() // To load in the app the content
when the user click on links
```

RESTful Web Service

EE P 523, Lecture 7

Web Services

- Standard for exchanging information between different types of applications irrespective of language and platform.
- For example, an android application can interact with java or .net application using web services.
- In practice, a web service commonly provides an object-oriented **web-based interface to a database server**, utilized for example by another web server, or by a mobile app, that provides a user interface to the end user.

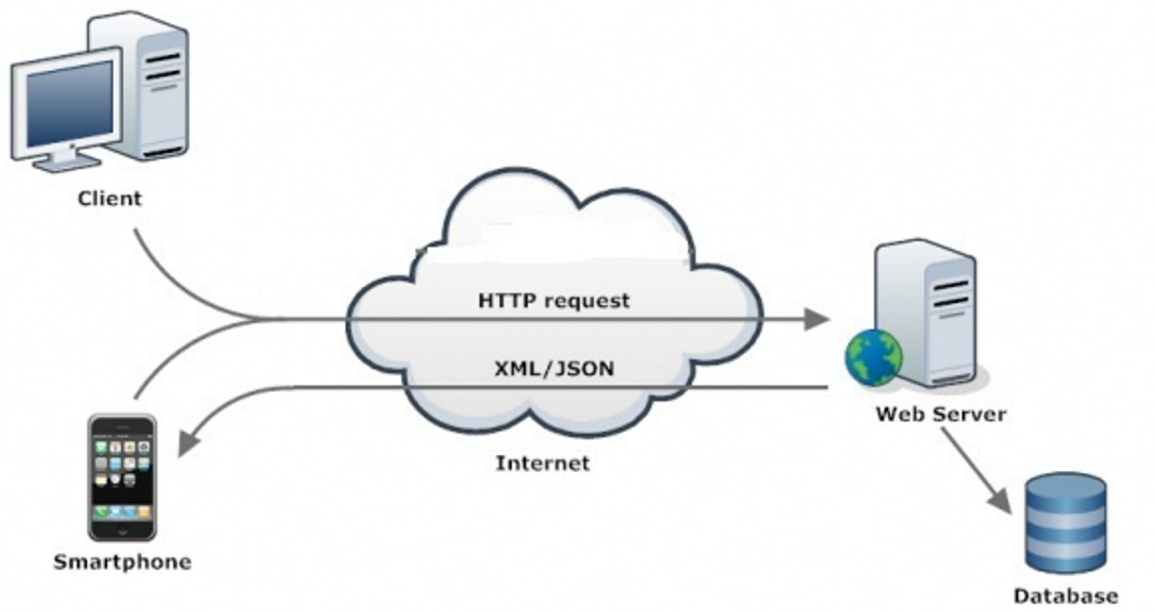
Web Services

A complete web service is, therefore, any service that

- Is available over the Internet or private (intranet) networks
- Uses a standardized XML messaging system
- Is not tied to any one operating system or programming language
- Is self-describing via a common XML grammar
- Is discoverable via a simple find mechanism

Web Services

Request data via URLs with parameters, and get the data returned as a response

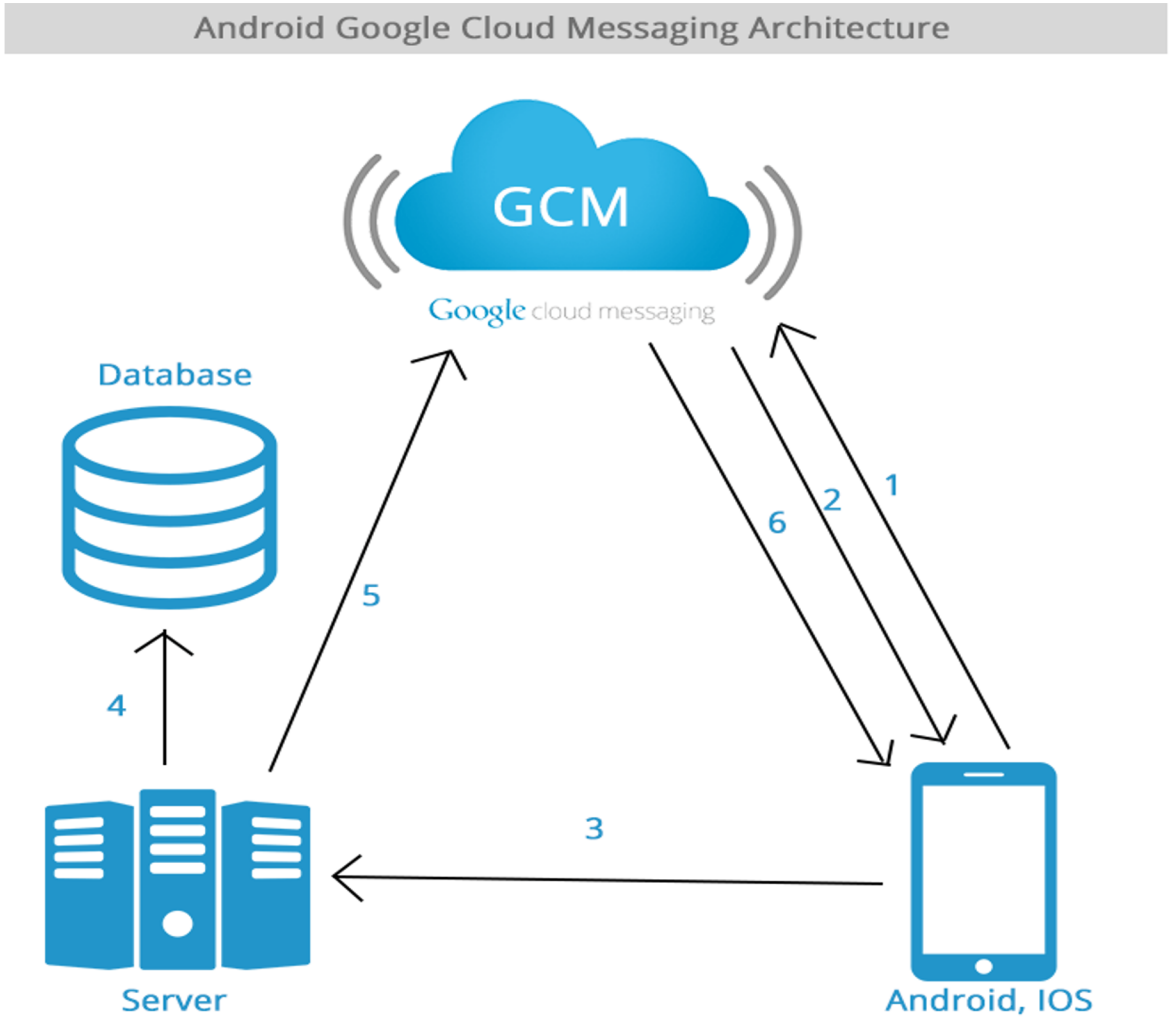


Example: AWS

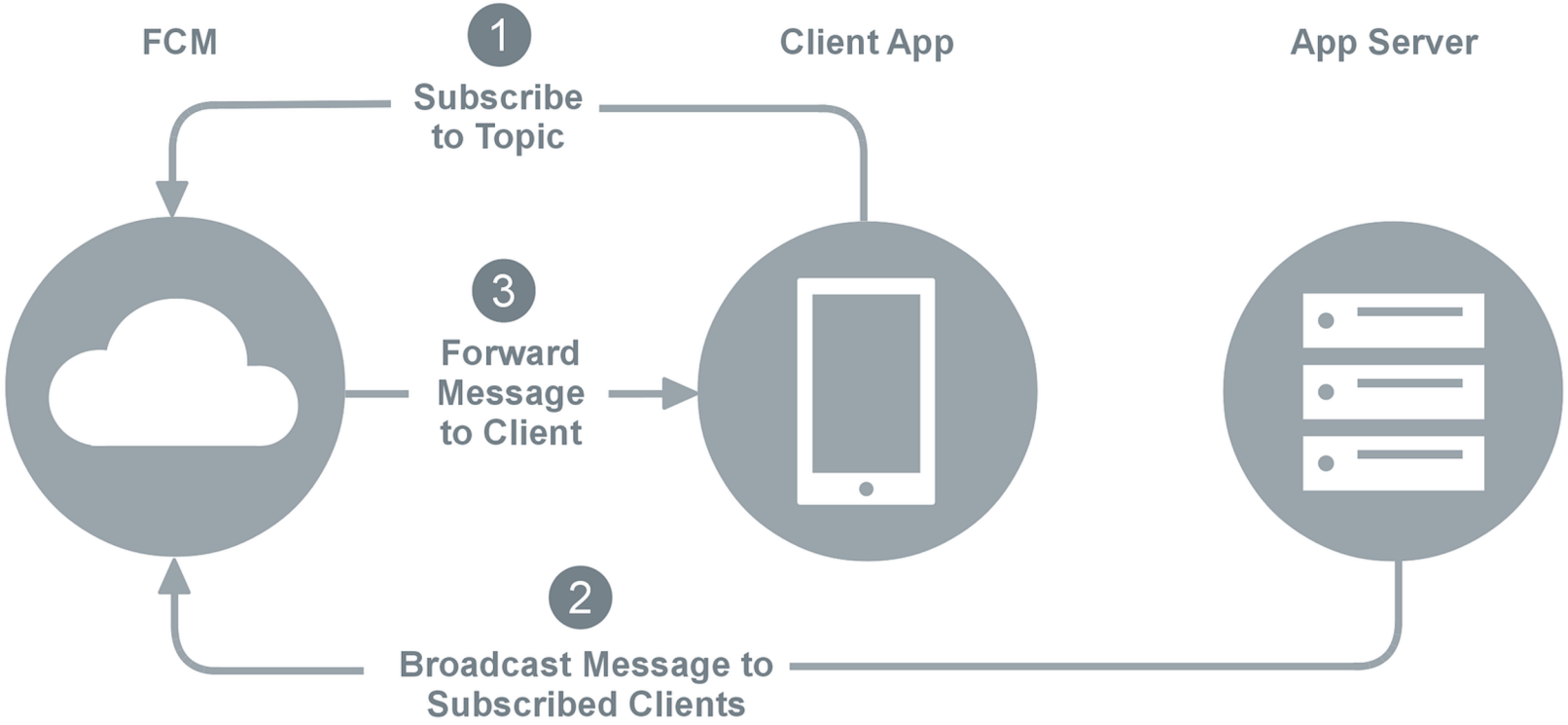
Amazon Web Services (**AWS)** provides on-demand cloud computing platforms

- to individuals, companies and governments, on a metered pay-as-you-go basis.
- In aggregate, these cloud computing web services provide a set of primitive, abstract technical infrastructure and distributed computing building blocks and tools.
- One of these services is Amazon Elastic Compute Cloud, which allows users to have at their disposal a virtual cluster of computers, available all the time, through the Internet.

Push Notifications



Push Notifications: Firebase Cloud Messaging



https://firebase.google.com/docs/cloud-messaging

What is REST?

Representational State Transfer (**REST**) is a software architectural style that defines a set of constraints to be used for creating Web services.

- Uses XML or JSON over HTTP

  - POST: create a resource on the server
  - GET: retrieve a resource
  - PUT: change the state of a resource or to update it
  - DELETE: remove or delete a resource

JSON (JavaScript Object Notation)

- Lightweight format for **storing and transporting data**
- Transmit data objects consisting of attribute–value pairs and array data types (or any other serializable value).
- It is a very common data format used for asynchronous browser–server communication, including as a replacement for XML in some AJAX-style systems.
- JSON is a language-independent data format

```
{
    "id": 1,
    "description": "Hiking"
}
```

JSON (JavaScript Object Notation)

Kotlin

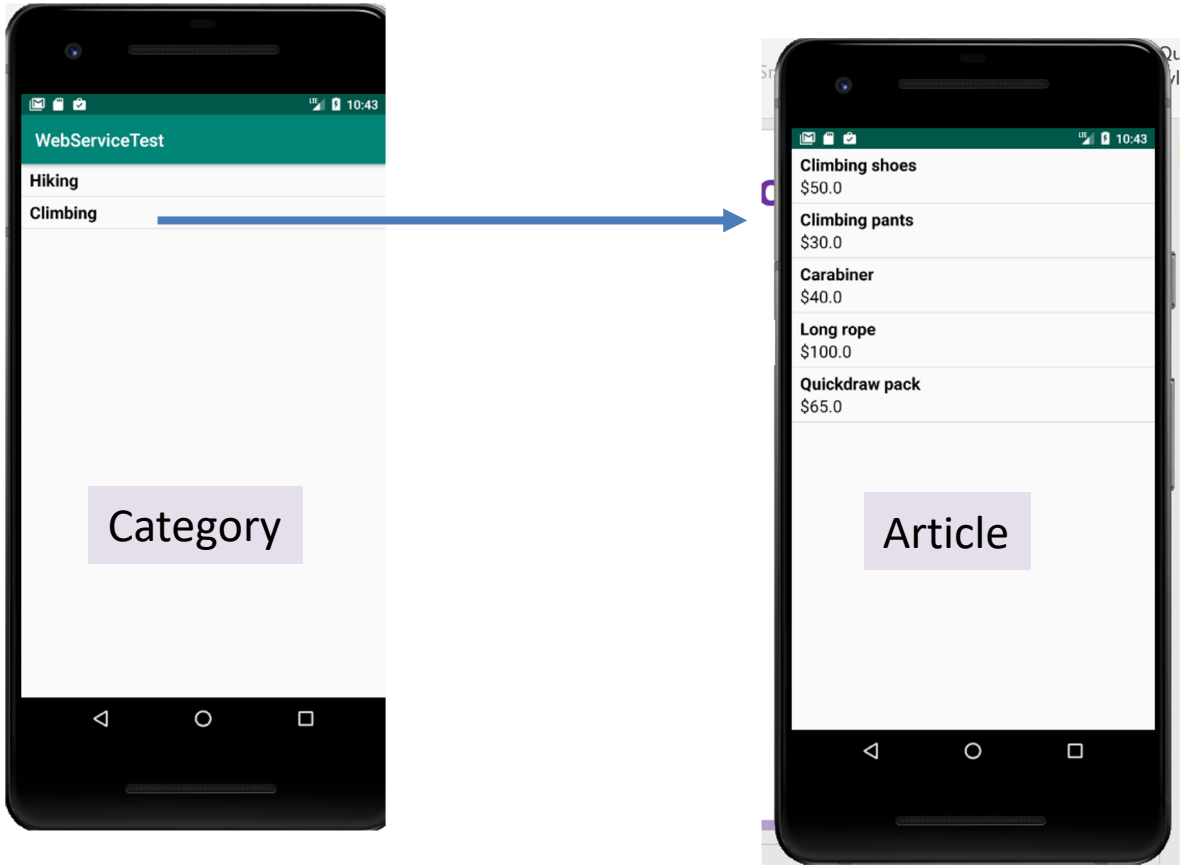
```kotlin
class Category {
    var id: Int = 0
    var description: String? = null
```

JSON

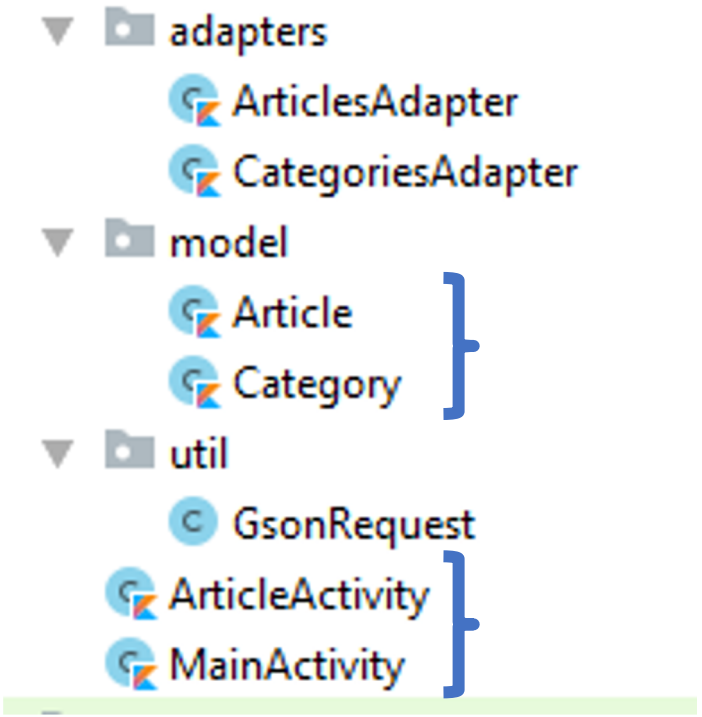
```json
{
    "id": 1,
    "description": "Hiking"
}
```

```json
[
    {
        "id": 1,
        "description": "Hiking"
    },
    {
        "id": 2,
        "description": "Climbing"
    }
]
```

Example: Sports Shop



Web Service Example




```
implementation 'com.google.code.gson:gson:2.8.5'
implementation 'com.android.volley:volley:1.1.1'
```

Where is the Data?

```
BEGIN TRANSACTION;

CREATE TABLE IF NOT EXISTS `category` (
      `categoryid`     INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,
      `description`    TEXT,
      `image`    TEXT
);
INSERT INTO `category` VALUES (1,'Hiking','hiking.png');
INSERT INTO `category` VALUES (2,'Climbing','climbing.png');
```

Table for Category

```
CREATE TABLE IF NOT EXISTS `article` (
      `articleid`      INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,
      `description`    TEXT,
      `price`    REAL,
      `image`    TEXT,
      `categoryid`     INTEGER,
      FOREIGN KEY(`categoryid`) REFERENCES `category`(`categoryid`)
);
INSERT INTO `article` VALUES (1,'Hiking pants',20.0,'hiking_pants.png',1);
INSERT INTO `article` VALUES (2,'Boots',50.0,'hiking_boots.png',1);
INSERT INTO `article` VALUES (3,'Climbing shoes',50.0,'climbing_shoes.png',2);
INSERT INTO `article` VALUES (4,'Climbing pants',30.0,'climbing_pants.png',2);
INSERT INTO `article` VALUES (5,'Carabiner',40.0,'climbing_carabiner.png',2);
INSERT INTO `article` VALUES (6,'Long rope',100.0,'climbing_rope.png',2);
INSERT INTO `article` VALUES (7,'Quickdraw pack',65.0,'climbing_quickdraw.png',2);

COMMIT;
```

Table for Article

Web Server

https://www.digitalocean.com/products/linux-distribution/ubuntu/

Standard Droplets

Balanced virtual machines with a healthy amount of memory tuned to host and scale applications like blogs, web applications, testing / stagin environments, in-memory caching and databases.

MEMORY	VCPUS	SSD DISK	TRANSFER	PRICE
1 GB	1 vCPU	25 GB	1 TB	$5/mo $0.007/hr
2 GB	1 vCPU	50 GB	2 TB	$10/mo $0.015/hr
3 GB	1 vCPU	60 GB	3 TB	$15/mo $0.022/hr
2 GB	2 vCPUs	60 GB	3 TB	$15/mo $0.022/hr
1 GB	3 vCPUs	60 GB	3 TB	$15/mo $0.022/hr
4 GB	2 vCPUs	80 GB	4 TB	$20/mo $0.030/hr

Example: SportsShop Web Service

- **MainActivity**: **GET** Categories data from the server and fill in the ListView

```
lateinit var  categories: Array<Category>

override fun onCreate(savedInstanceState: Bundle?) {

    val queue = Volley.newRequestQueue(this)

    val myReq = GsonRequest(Method.GET, getString(R.string.server_url) +
"/categories/",    Array<Category>::class.java,
            createMyReqSuccessListener(), createMyReqErrorListener())

    queue.add(myReq)

    val categoriesLV = findViewById<View>(R.id.categories_listview) as ListView
    categoriesLV.onItemClickListener = this
}
```

```
<ListView
    android:id="@+id/categories_listview"
    android:layout_width="match_parent"
    android:layout_height="match_parent"></ListView>
```

Example: Sports Shop

- **MainActivity:** on Category click: Start ArticleActivity put the id of the selected category in the Intent

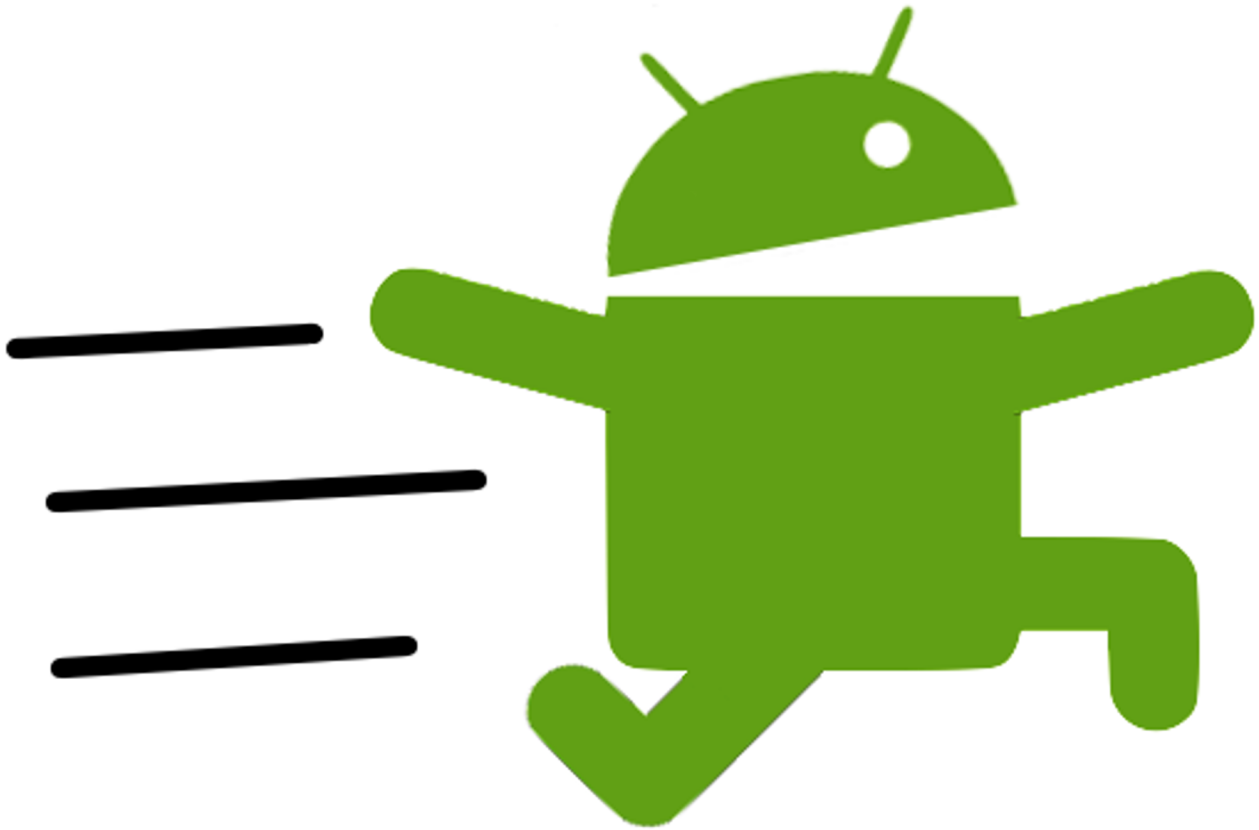
```
override fun onItemClick(adapterView: AdapterView<*>, view: View, i: Int, l: Long) {
    val intent = Intent(baseContext, ArticleActivity::class.java)
    intent.putExtra(getString(R.string.selected_category_id_key), categories[i].id)
    startActivity(intent)
}
```

Example: Sports Shop

- **ArticleActivity:** **GET** the category id from the intent, and fill in the ListView

```
   override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_article)
        val queue = Volley.newRequestQueue(this)

        val categoryId = intent.extras!!.getInt(getString(R.string.selected_category_id_key), 0)

        val myReq = GsonRequest(Request.Method.GET, getString(R.string.server_url) +
"/categories/" + categoryId + "/articles", Array<Article>::class.java,
                createMyReqSuccessListener(), createMyReqErrorListener())

        queue.add(myReq)
    }
```

```
private fun createMyReqSuccessListener(): Response.Listener<Array<Article>> {
    return Response.Listener { response ->
        val articlesLV = findViewById<View>(R.id.articles_listview) as ListView

        val adapter = ArticlesAdapter(baseContext, response)
        articlesLV.adapter = adapter
}
}
```



Your Questions

?