IEMS5722
Mobile Network Programming and Distributed Server Architecture

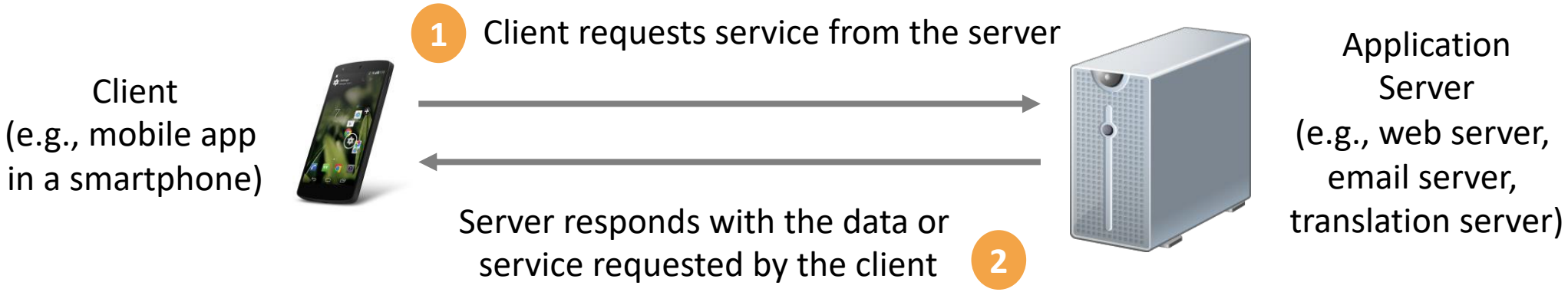
Lecture 10

Web Sockets for Real-time Communications

WebSocket

Limitations of HTTP (Recap)

- All of the examples we have gone through in network programming so far can be regarded as using the "**pull**" method
- Communication is always **initiated by the client**
- Client "pulls" data or services from the server **when necessary** (e.g. when the user launches the app, or presses a button)



Limitations of HTTP (Recap)

- **HTTP** is a **pull-based** protocol
  - Users browse the Web and **actively** decide which Website to browse, which link to follow, etc.
  - An **effective** and **economical** way (each user chooses what they need)
  - However, if some resources are regularly requested, the pull model can put heavy load on the server

Implementing Push (Recap)

- The **World Wide Web**, and in particular the **HTTP protocol**, is designed for "**pull**", and additional engineering is required to implement push on the Web
- Some ways to "**emulate**" push on the Web
  - Polling (periodic pull)
  - Comet Model
  - BOSH
  - WebSockets

WebSocket

- HTTP is **half-duplex**: only one side can send data at a time, like using walkie-talkies
- WebSocket is a protocol providing **full-duplex** communications channels between two computers over a **TCP connection**
- Designed to be implemented in **Web browsers** and **Web servers**
- Communications are done over **TCP port 80**
  (can be used in secured computing environments)
- Supported since IE 10, Firefox 11, Chrome 16, Safari 6, Opera 12.10, Android Browser 4.4.

Reference:
https://tools.ietf.org/html/rfc6455
https://www.websocket.org/

WebSocket

- A **persistent** connections between a Web browser (or a mobile app) and a server
- Both sides can send out data to the other side at any time
- Why WebSocket?
  - Lower latency (avoid TCP handshaking)
  - Smaller overhead (only 2 bytes per message)
  - Less unnecessary communication (data is only sent whenever needed)

WebSocket

- WebSocket is part of the **HTML5** standard
  - Supported in latest versions of major Web browsers
  - Simple API in JavaScript
  - Libraries also available on iOS and Android
- Try HVBRD using your phone and computer together: https://experiments.withgoogle.com/hvbrd

```
var host = 'ws://localhost:8000/example';
var socket = new WebSocket(host);

socket.onopen = function() {
    console.log('Socket opened');
    socket.send('Hello server!');
}

socket.onmessage = function(msg) {
    console.log('Server says: ' + msg);
}

socket.onclose = function() {
    console.log('Socket closed');
}
```

WebSocket

- Particularly useful when you would like to develop applications such as:
  - Real-time multiplayer games
  - Chatrooms
  - Real-time news feed
  - Collaborative apps (e.g. consider something like Google Documents)
  - Live commenting
  - …

WebSocket

- **Design principles** of WebSocket
  - An additional layer on top of TCP
  - Enable bi-directional communication between client and servers
  - Support low-latency apps without HTTP overhead
  - Web origin-based security model for browsers
  - Support multiple server-side endpoints

WebSocket

- How does WebSockets work?
- WebSocket handshake
  - Client sends a regular HTTP request to the server with an “upgrade” header field

```
GET /chat HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Sec-WebSocket-Version: 13
```

  - If server supports WebSocket, it sends back a response with the upgrade header field

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: HSmrc0sMlYUkAGmm5OPpG2HaGWk=
```

WebSocket

- Once the handshake is completed:
  - The initial HTTP connection will be replaced by the **WebSocket connection** (using the same underlying TCP/IP connection)
  - Both the client and the server can now start sending data to the other side
  - Data are transferred in **frames**
    - Messages (payload) will be reconstructed once all frames are received
    - Because of the established WebSocket connection, much less overhead will be incurred on the message being transmitted
- Check whether a browser supports WebSocket: https://caniuse.com/#search=websocket

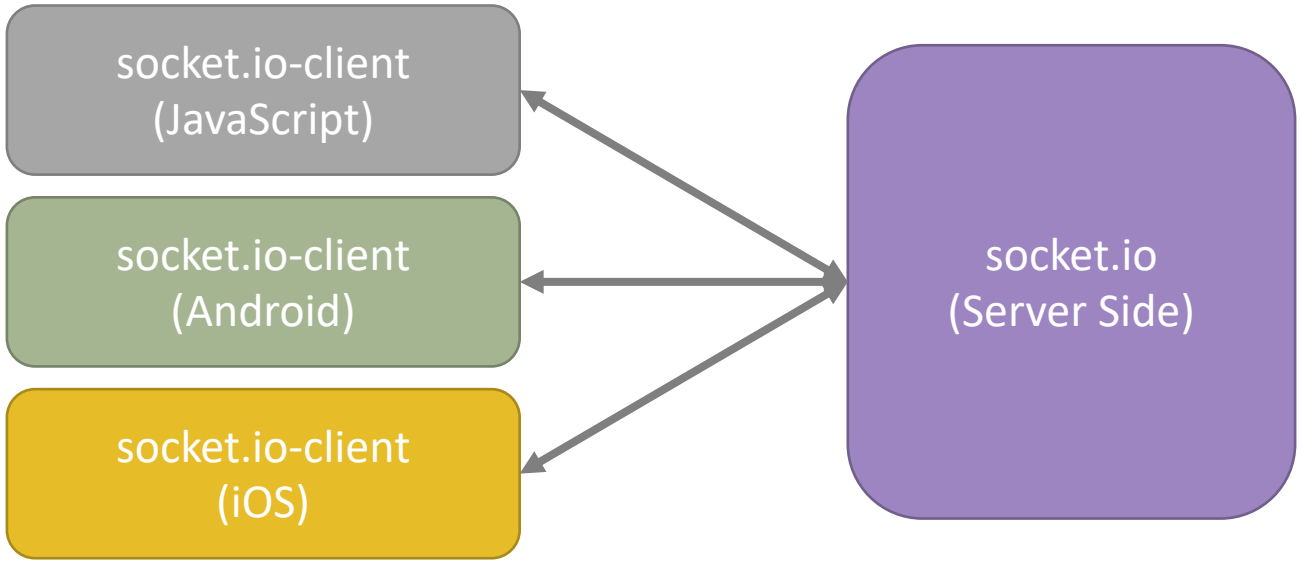
socket.io

socket.io

- A library based on **Node.js** for real time, bi-directional communication between a server and one or multiple clients
- Using **WebSocket** to perform data communication (fall back to older solutions when WebSocket is not supported)
- Originally written for Node.js on the server side and JavaScript on the client side, there are now libraries for Python, Android and iOS
- Official Website: https://socket.io/

socket.io

- socket.io has two parts: 1) Server and 2) Client
- Client libraries are available in JavaScript (Web), Android and iOS
- Allow you to build real-time apps across multiple platforms



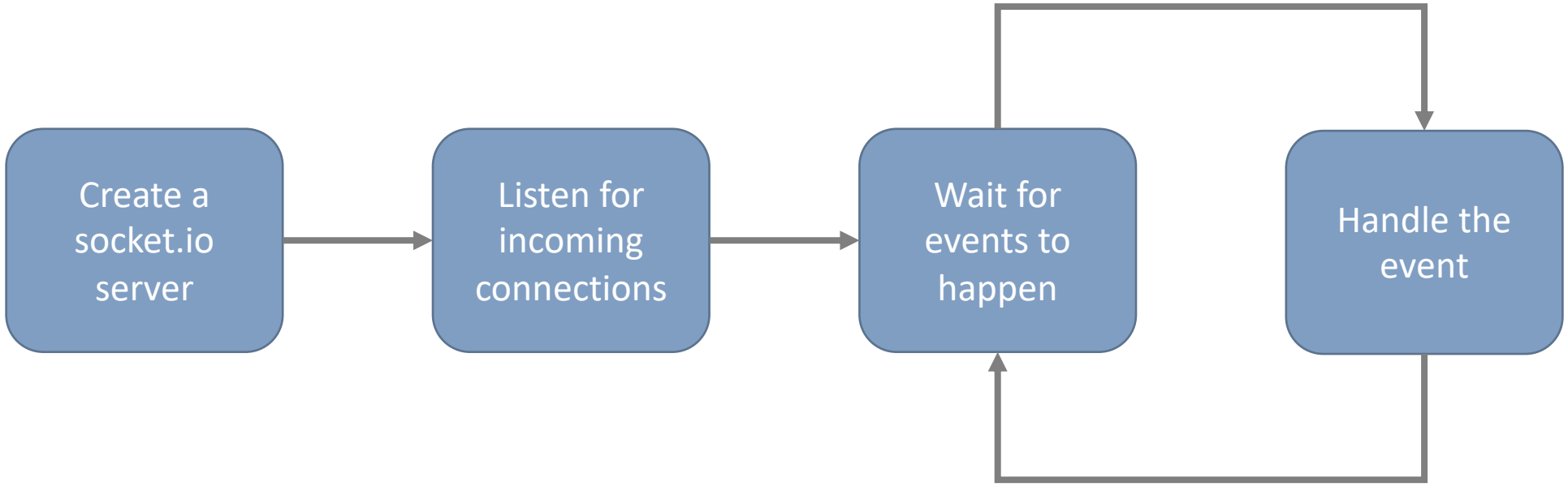
Reference:
iOS Client: https://socket.io/blog/socket-io-on-ios/
Android Client: https://socket.io/blog/native-socket-io-and-android/

socket.io

- **Event-driven**
  - Once connected, the server and client can communicate with each other by triggering or "**emitting**" events
  - Create **callback functions** to carry out different actions whenever some events happens

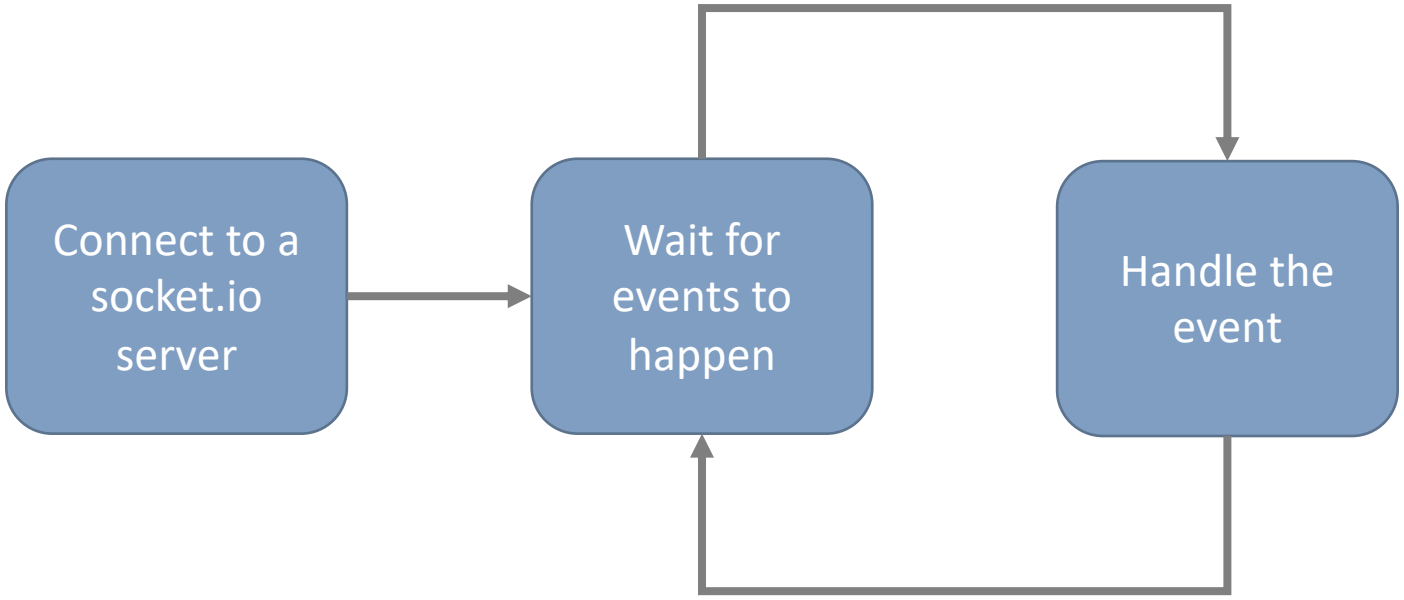
socket.io

- Flowchart of a server-side program



socket.io

- Flowchart of a client-side program



Flask-SocketIO

Flask-SocketIO

- A module that allows you to use socket.io in Flask applications https://flask-socketio.readthedocs.io/
- Install using the following command:

```
$ sudo pip3 install flask-socketio
```

- You will also need the concurrent network library Eventlet as well (http://eventlet.net/)

```
$ sudo pip3 install eventlet
```

Flask-SocketIO

- Initialization

```
from flask import Flask, render_template
from flask_socketio import SocketIO

app = Flask(__name__)
app.config['SECRET_KEY'] = 'secret!'
socketio = SocketIO(app)

if __name__ == '__main__':
    socketio.run(app)
```

You need to provide a secret key for Flask to encrypt data for user sessions

Setup the app to use functions of socket.io

Flask-SocketIO

- Remember we mentioned that in socket.io all communications are based on events.
- We will have to define handlers of events in our Flask application
- For example:

```
@socketio.on('message')
def handle_message(message):
    # Action to be performed when a message is received

@socketio.on('my event')
def handle_my_custom_event(arg1, arg2, arg3):
    # Action to be performed when a custom event is received
```

Event names:
reserved and custom

Events

- Both the server and the client can generate events, and if the other side has a handler of that event, the handler will be invoked to carry out some actions
- In Flask-SocketIO, there are several different types of events
  - **Special events** (‘connect’, ‘disconnect’, ‘join’, ‘leave’)
  - **Unnamed events** (‘message’ or ‘json’)
  - **Custom events** (a name of your choice, e.g. ‘my event’)

Events

- Connection events

```
# Invoked whenever a client is connected
@socketio.on('connect')
def connect_handler():
    print('Client connected')

# Invoked whenever a client is disconnected
@socketio.on('disconnect')
def disconnect_handler():
    print('Client disconnected')
```

Events

- Unnamed events

```
# Invoked whenever an unnamed event happens
@socketio.on('message')
def message_handler(message):
    print('Message received: ' + message)

# Invoked whenever an unnamed event happens (with JSON data)
@socketio.on('json')
def json_handler(json):
    print('JSON received: ' + str(json))
```

Events

- Custom events
  - Message data can be string, bytes, int or JSON

```
@socketio.on('my event 1')
def my_event_1_handler(json):
    print('JSON received: ' + str(json))
```

  - Also support multiple arguments

```
@socketio.on('my event 2')
def my_event_2_handler(arg1, arg2, arg3):
    print('Arguments received: ' + arg1 + arg2 + arg3)
```

Events

- The server can send messages to clients by using **send()** or **emit()** functions
- The **send()** function is for sending unnamed events
- The **emit()** function is for sending custom named events

```
from flask_socketio import send, emit

@socketio.on('message')
def message_handler(message):
    send(message)

@socketio.on('json')
def json_handler(json):
    send(json, json=True)

@socketio.on('my event 1')
def my_event_1_handler(json):
    emit('my response', json)
```

Broadcast

- Normally, **send** and **emit** only send message to a single client
- Broadcasting allows you to send a message to all clients who are connected to the server
- For example, in a multi-player game, one user performs an action, and you want this to be known to all other users

```
@socketio.on('my event 1')
def my_event_1_handler(data):
    emit('my response', data, broadcast=True)
```

Set “broadcast=True” when emitting a message to broadcast to all connected clients

Rooms

- In some applications, users may interact with only a **subset** of other users
- Examples:
  - Chat application with multiple rooms
  - Multiplayer games (multiple game boards, game rooms, etc.)
  - …

```
from flask_socketio import join_room, leave_room

@socketio.on('join')
def on_join(data):
    username = data['username']
    room = data['room']
    join_room(room)
    send(username + ' has entered the room.', room=room)

@socketio.on('leave')
def on_leave(data):
    username = data['username']
    room = data['room'] leave_room(room)
    send(username + ' has left the room.', room=room)
```

Deploying Flask-SocketIO

- If you have "**eventlet**" installed, you can use the embedded server which is production-ready
- Invoked by **socketio.run(app, host="0.0.0.0", port=5000)**
- Sample supervisor configuration files:

```
[program:iems5722_socketio]
command = python3 flask-socketio.py
directory = /home/ubuntu/iems5722
user = ubuntu
autostart = true
autorestart = true
stdout_logfile = /home/ubuntu/iems5722/flask-socketio_app.log
redirect_stderr = true
```

Deploying Flask-SocketIO

- If you would like to deploy a single Flask app including both your APIs and the socket.io event handlers, use Gunicorn.
- For example:

```
[program:iems5722]
command = gunicorn –k eventlet -b localhost:8000 -w 1 iems5722:app
directory = /home/ubuntu/iems5722
user = ubuntu
autostart = true
autorestart = true
stdout_logfile = /home/ubuntu/iems5722/iems5722_app.log
redirect_stderr = true
```

What can you do with socket.io?

- Real-time chatrooms
- Real-time multiplayer games
- Real-time event broadcasting (e.g. real-time results of a competition or an event)
- And a snake game across multiple screens! https://www.youtube.com/watch?v=yUcRsU0CurQ

socket.io on Android

socket.io on Android

- A socket.io client library for Java and Android
- Source code and set-up guide: https://github.com/socketio/socket.io-client-java
- To use the library in your Android project, add the following into the **dependencies** to your **build.gradle** file under the **app** directory

```
dependencies {
    ...
    implementation ('io.socket:socket.io-client:1.0.0') {
        // excluding org.json which is provided by Android exclude group:
        exclude group: 'org.json', module: 'json'
    }
}
```

Reference:
https://github.com/socketio/socket.io-client-java
https://socket.io/blog/native-socket-io-and-android/

socket.io on Android

- Creating a socket.io client in Android

```
import io.socket.client.Socket;
private Socket socket;

@Override
protected void onCreate(Bundle savedInstanceState) {
    ...
    try {
        socket = IO.socket("http://chat.socket.io");
    } catch (URISyntaxException e) {
         e.printStackTrace();
    }
    socket.on(Socket.EVENT_CONNECT_ERROR, onConnectError);
    socket.on(Socket.EVENT_CONNECT_TIMEOUT, onConnectError);
    socket.on(Socket.EVENT_CONNECT, onConnectSuccess);
    socket.on("my event", onMyEvent);
    socket.connect();
}
```

The URL of the socket.io server

Callback functions (handlers) for different events

Initiating a connection to the server

socket.io on Android

- Creating event handlers/listeners:

```
private Emitter.Listener onConnectError = new Emitter.Listener() {
    @Override
    public void call(Object... args) {
        JSONObject data = (JSONObject) args[0];
        ...
    }
};
```

There can be multiple parameters passed to this function (depending on the server). If it is JSON, cast it to a JSONObject or JSONArray.

Must override this "call()" function in your listener

This listener function is called on a new thread, not the UI thread. You should not manipulate the UI components here.

socket.io on Android

- Sending string message back to the server.

```
socket.emit("my event", "data");
```

- You can also send JSON objects using the emit function.

```
JSONObject json = new JSONObject();
json.put("username", "Marco");
json.put("message", "Hello!");
socket.emit("my event", json);
```

socket.io on Android

- Like many other things on Android, you need to manage the socket's life cycle. When the user leaves the activity, you should disconnect.

```
@Override
protected void onDestroy() {
    socket.disconnect();
    socket.off();

    super.onDestroy();
}
```

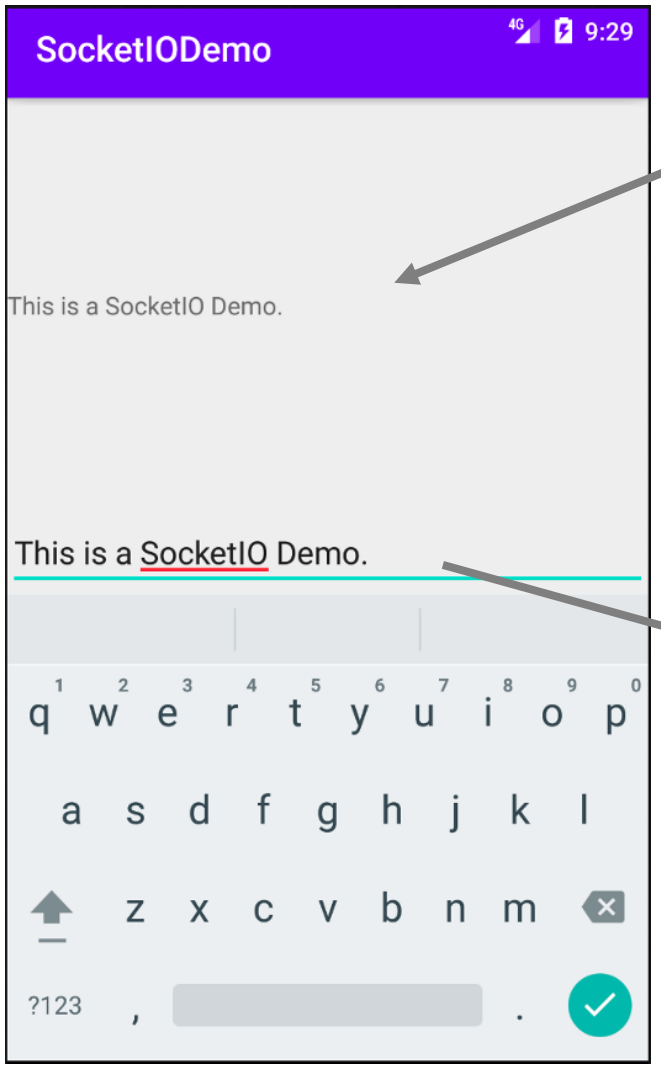
Calling "off()" to remove listeners.

SocketIO Demo: Real-time Typing Update

SocketIO Demo

A TextView for displaying server output

An EditText for obtaining user input



Client receives the event and updates the TextView.

Server receives the event and emits a broadcast event to all connected clients.

When the text is changed, an event is emitted to the server.

SocketIO Demo – Server Application

```python
from flask import Flask, render_template
from flask_socketio import SocketIO, emit

app = Flask(__name__)
app.config['SECRET_KEY'] = 'iems5722'
socketio = SocketIO(app)

@socketio.on('text')
def update_handler(json):
    emit('update', {'text': json['text']}, broadcast=True)

if __name__ == '__main__':
    socketio.run(app, host='0.0.0.0', port=5000)
```

SocketIO Demo – Android Client Activity (1)

```
package hk.edu.cuhk.ie.iems5722.socketiodemo;

import androidx.appcompat.app.AppCompatActivity;

import android.os.Bundle;
import android.text.Editable;
import android.text.TextWatcher;
import android.widget.EditText;
import android.widget.TextView;

import org.json.JSONException;
import org.json.JSONObject;

import java.net.URISyntaxException;

import io.socket.client.IO;
import io.socket.client.Socket;
import io.socket.emitter.Emitter;

public class MainActivity extends AppCompatActivity {
    private Socket socket;
    private TextView output;
    private EditText input;
```

SocketIO Demo – Android Client Activity (2)

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

        output = (TextView) findViewById(R.id.textViewOutput);
        input = (EditText) findViewById(R.id.editTextInput);

        try {
            socket = IO.socket("http://localhost:5000/");
            socket.on(Socket.EVENT_CONNECT, onConnectSuccess);
            socket.on("update", onTextUpdate);
            socket.connect();
        } catch (URISyntaxException e) {
            e.printStackTrace();
        }
```

SocketIO Demo – Android Client Activity (3)

```
        input.addTextChangedListener(new TextWatcher() {
            @Override
            public void beforeTextChanged(CharSequence s, int start, int count, int after) {
            }

            @Override
            public void onTextChanged(CharSequence s, int start, int before, int count) {
            }

            @Override
            public void afterTextChanged(Editable s) {
                if (socket != null) {
                    try {
                        JSONObject json = new JSONObject();
                        json.put("text", s.toString());
                        socket.emit("text", json);
                    } catch (JSONException e) {
                        e.printStackTrace();
                    }
                }
            }
        });
    }
```

SocketIO Demo – Android Client Activity (4)

```
@Override
protected void onDestroy() {
    if (socket != null) {
        socket.disconnect();
        socket.off();
    }

    super.onDestroy();
}

private Emitter.Listener onConnectSuccess = new Emitter.Listener() {
    @Override
    public void call(Object... args) {
        runOnUiThread(new Runnable() {
            @Override
            public void run() {
                output.setText("Connected");
            }
        });
    }
};
```

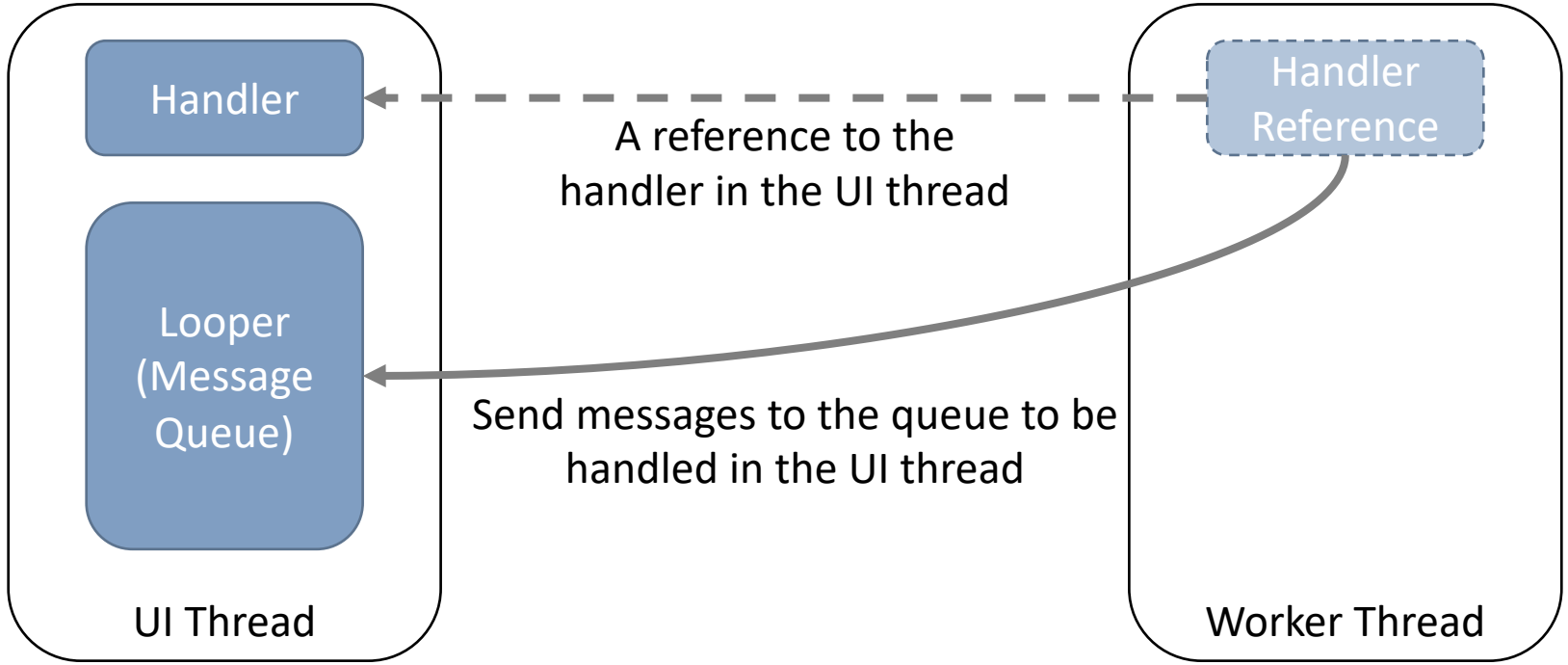
SocketIO Demo – Android Client Activity (5)

```
    private Emitter.Listener onTextUpdate = new Emitter.Listener() {
        @Override
        public void call(Object... args) {
            try {
                JSONObject data = (JSONObject) args[0];
                final String text = data.getString("text");
                runOnUiThread(new Runnable() {
                    @Override
                    public void run() {
                        output.setText(text);
                    }
                });
            } catch (JSONException e) {
                e.printStackTrace();
            }
        }
    };
}
```

Multithreading and Handlers

Handlers

- **Handler** is a component in the Android system for managing threads. A **Handler** object:
  - Is associated with a particular thread and its message queue called “looper”
  - Receives messages and runs code to handle the message



Handlers

- A **handler** will be associated with the **thread** in which it is created
- You need to override the **handleMessage()** method to specify the action(s) to be taken when a message is received

```
public class MainActivity extends AppCompatActivity {
    private static class MainHandler extends Handler {
        @Override
        public void handleMessage(Message msg) {
            ...
        }
    }

    MainHandler handler = new MainHandler();
    ...
}
```

Manipulation of UI components can be done here, as this will be performed on the UI thread

Handlers

- When you want to ask the UI thread to perform something, send a message to its handler.

```
...
Runnable runnable = new Runnable() {
    public void run() {
        String data = "Message to send.";
        Message message = handler.obtainMessage(ACTION_CODE, data);
        message.sendToTarget();
    }
};
...
```

Handlers

- A message can be created by using the **Handler.obtainMessage(...)** method.
- You can supply up to four parameters to this method.

```
obtainMessage()
obtainMessage(int what)
obtainMessage(int what, Object obj)
obtainMessage(int what, int arg1, int arg2)
obtainMessage(int what, int arg1, int arg2, Object obj)
```

A user-defined code for identifying the type of the message

Integer parameters to the handler

Arbitrary object to the handler

Reference: https://developer.android.com/reference/android/os/Handler

Handlers

- To extract parameters from the message in the **handleMessage()** method:

```
...
private static class MainHandler extends Handler {
    @Override
    public void handleMessage(Message msg) {
        if (msg.what == ACTION_CODE_1) {
            int arg1 = msg.arg1;
            int arg2 = msg.arg2;
            MyObject obj = (MyObject) msg.obj;
        } else if (msg.what == ACTION_CODE_2) {
            ...
        }
        ...
    }
}
...
```

Reference: https://developer.android.com/reference/android/os/Message

Embedding Data in Messages

- You can also embed data in the message using the **setData()** method

```
...
Message msg = new Message();
Bundle data = new Bundle();
data.putString("PARAM_1", "string");
data.putInt("PARAM_2", 5722);
...
msg.setData(data);
msg.what = ACTION_CODE;
handler.sendMessage(msg);
...
```

- Then in the **handleMessage()** method you can retrieve the data like this:

```
String param_1 = msg.getData().getString("PARAM_1");
```

SocketIO Demo – Modified Android Activity (1)

```
...

import android.os.Handler;
import android.os.Message;

import java.lang.ref.WeakReference;

public class MainActivity extends AppCompatActivity {
    ...
    private static final int ACTION_CONNECTED = 1;
    private static final int ACTION_UPDATE = 2;

    private MainHandler handler = new MainHandler(this);

    private static class MainHandler extends Handler {
        private final WeakReference<MainActivity> mainActivity;
        MainHandler(MainActivity activity) {
            this.mainActivity = new WeakReference<>(activity);
        }
```

SocketIO Demo – Modified Android Activity (2)

```
        @Override
        public void handleMessage(Message msg) {
            MainActivity activity = mainActivity.get();

            switch (msg.what) {
                case ACTION_CONNECTED:
                    activity.output.setText((String) msg.obj);
                    break;

                case ACTION_UPDATE:
                    activity.output.setText((String) msg.obj);
                    break;

                default:
                    super.handleMessage(msg);
            }
        }
    }
```

SocketIO Demo – Modified Android Activity (3)

```
    private Emitter.Listener onConnectSuccess = new Emitter.Listener() {
        @Override
        public void call(Object... args) {
            Message msg = handler.obtainMessage(ACTION_CONNECTED, "Connected");
            msg.sendToTarget();
        }
    };

    private Emitter.Listener onTextUpdate = new Emitter.Listener() {
        @Override
        public void call(Object... args) {
            try {
                JSONObject data = (JSONObject) args[0];
                String text = data.getString("text");
                Message msg = handler.obtainMessage(ACTION_UPDATE, text);
                msg.sendToTarget();
            } catch (JSONException e) {
                e.printStackTrace();
            }
        }
    };
}
```

Authentication & Authorization

User Authentication

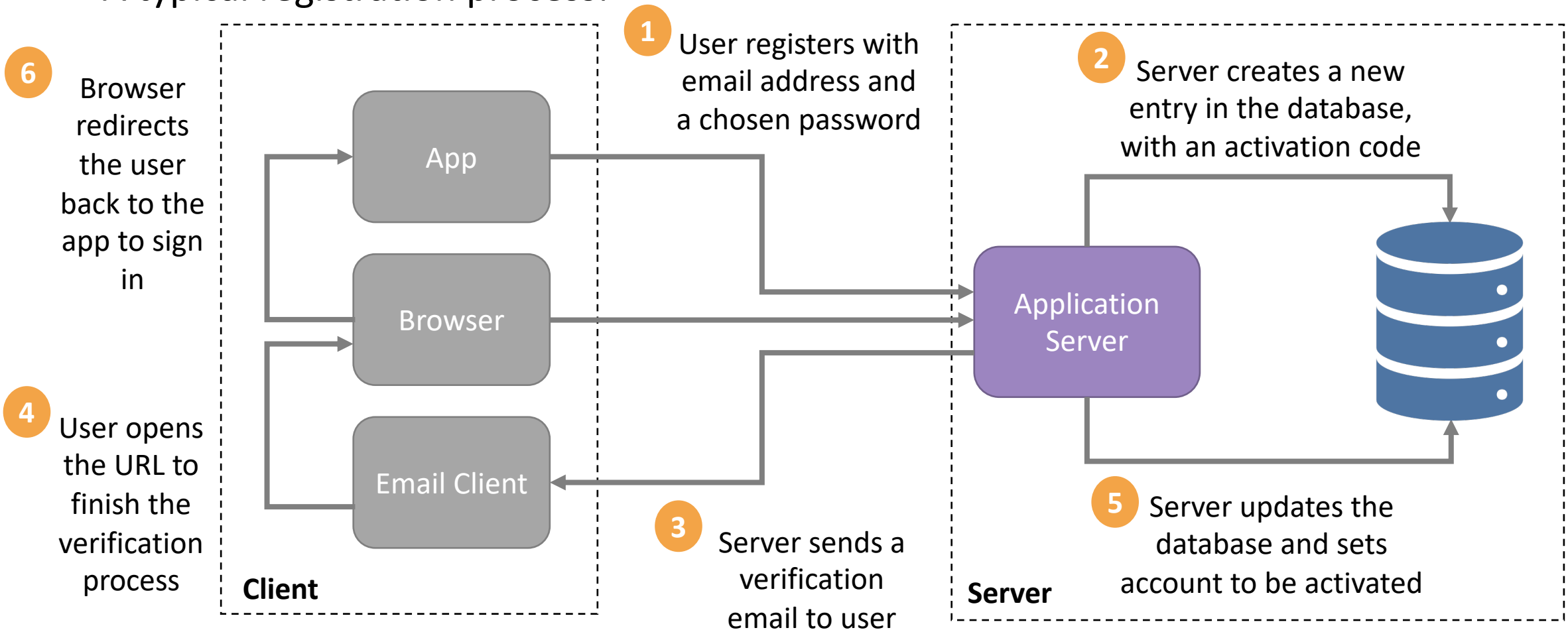
- It is very likely that you would like your users to create an account in your app, and sign in to use its services
- Reasons:
  - It is necessary; your app may need to uniquely identify every user
  - Track user's usage of the app
  - Allow users to retrieve their data on different devices
  - Present more personalized information and services
  - …

User Authentication

- How should you implement your system to enable user authentication in your app?
- Basic functions
  - **Register** an account by email (or phone number)
  - **Validate** user's email address by sending him/her an activation email
  - **Sign in** using email (or username) and chosen password
  - **Authenticate** the user whenever the app makes API requests

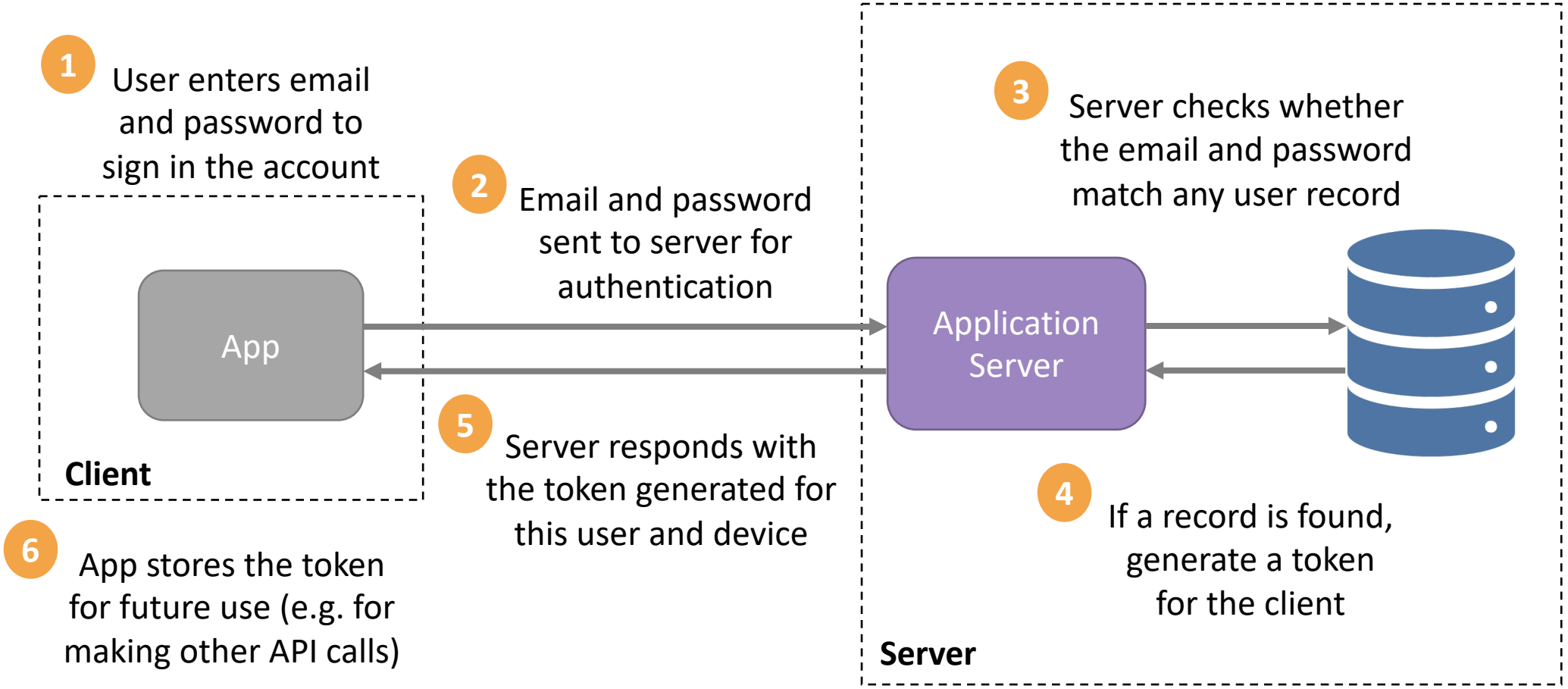
Registration

- A typical registration process:



Sign-In

- A typical sign-in process:



User Authentication

- Best practices for privacy and security concerns
  - **Do not store** the password clear text in your database, **hash** it with a **salt** before sending it out from the app (e.g. using MD5 or SHA1, or more secure ones)
  - **Do not store** user password in the device
  - Use **HTTPS** whenever possible
  - **Validate** user's input (e.g. email and password), before sending them to the server

Reference: https://developer.android.com/training/articles/security-tips

Authorization

- When using third party libraries, SDKs or APIs in your app, usually it requires the user to **authorize** your app to use data in another application
- Example:
  - Retrieve your **friend list** from Facebook
  - Retrieve your **name** and **email address** from Google
  - Retrieve your **posts** from Twitter or Weibo

Authorization

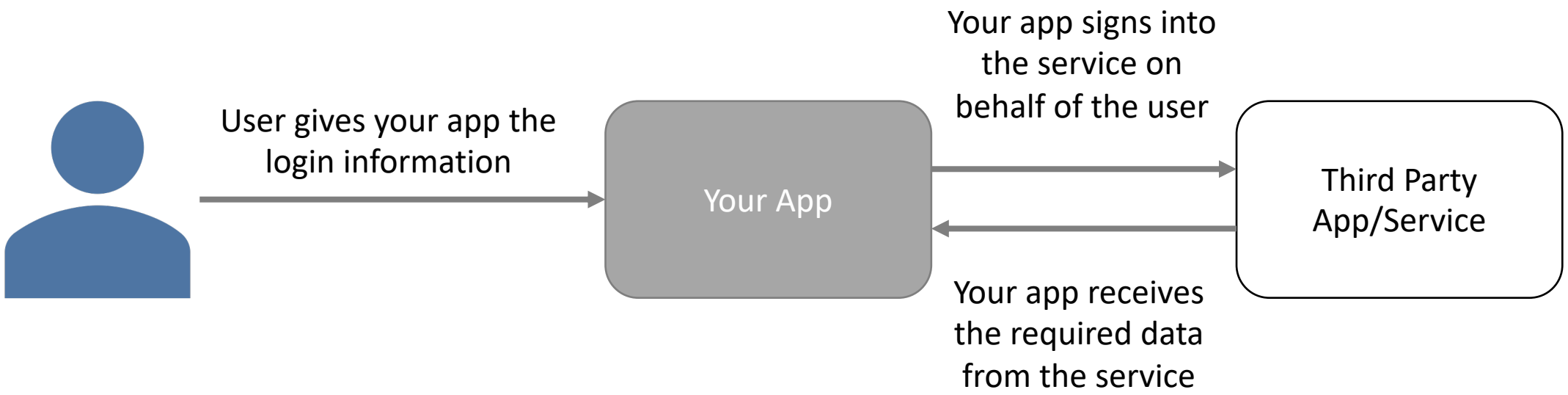
- For most Internet services, SDKs for Android and iOS are available for integrating sign-in process in your app:
  - Facebook https://developers.facebook.com/docs/facebook-login/
  - Twitter https://developer.twitter.com/en/docs/basics/authentication/guides/log-in-with-twitter
  - Google https://developers.google.com/identity/sign-in/android
  - Wechat https://developers.weixin.qq.com/doc/oplatform/en/Mobile_App/WeChat_Login/Development_Guide.html

Authorization

- For asking for authorization to use data in another app, you should always use their latest SDK whenever possible
- These SDKs wrapped the process of asking authorization from the user
- Most of services perform authorization using **OAuth 2.0 (Open Authorization Version 2.0)**
- It is unlikely (but not impossible) that you will have to implement the flow by yourself, so let's take a look at it

Open Authorization (OAuth)

- To access a user's account and the data within, you need to have the user's username and password
- Does the following make sense?



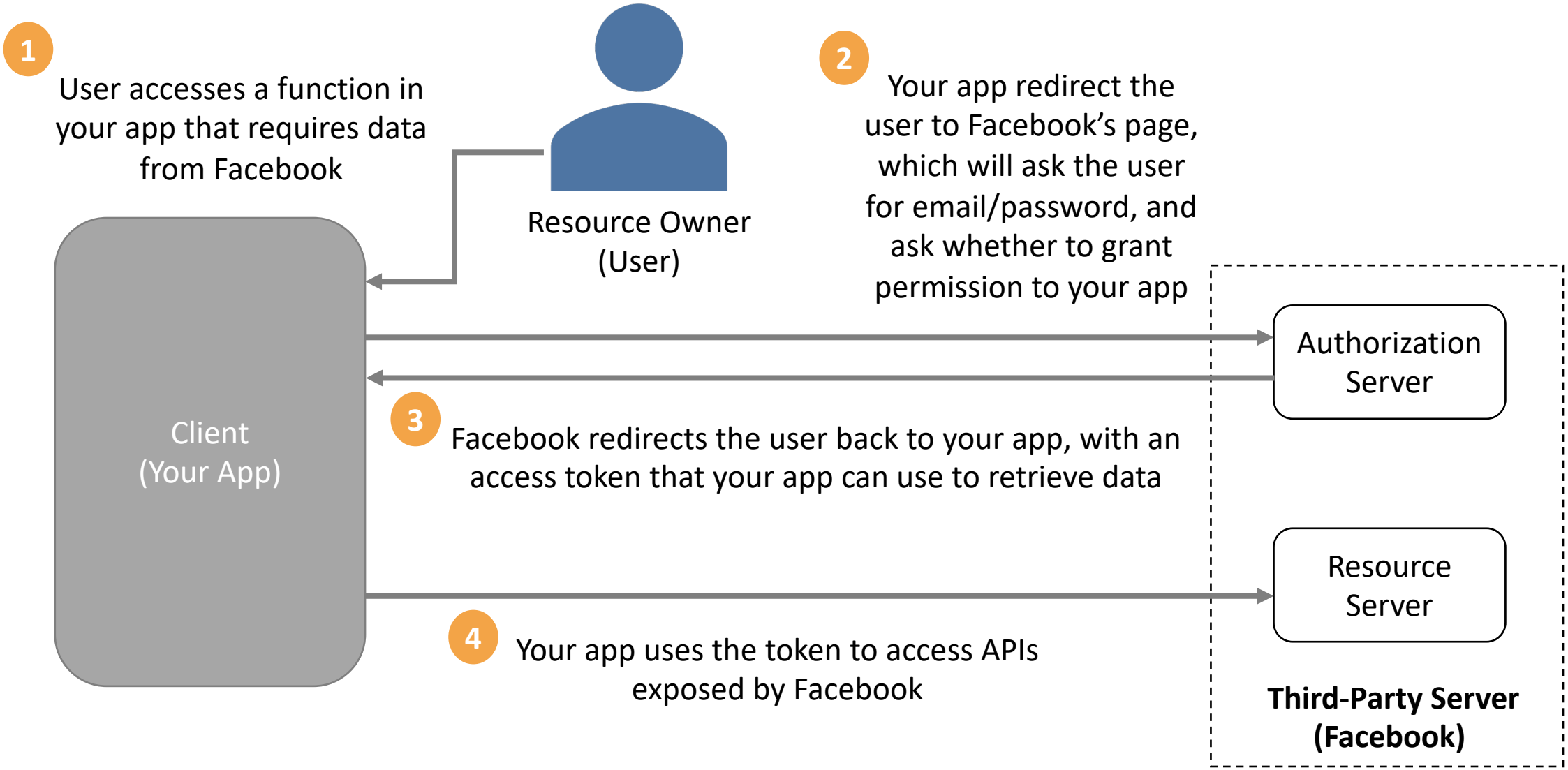
Open Authorization (OAuth)

- Never ask users to “give away” their password
- OAuth allows you to get access to user’s data in another service without the need to ask the user for his user account
- Let’s consider a scenario:
  - You have developed a social app, and you want to access the user’s friend list in Facebook, so that you can build up the social network in your app quickly

Open Authorization (OAuth)

- In OAuth, we have three entities:
  - **Resource Owner**
    - This is the user who is using your app
  - **Resource and Authorization Server**
    - In our case this is Facebook's server, which will authorize your app and serve the user's data to your app
  - **Client**
    - Your app, the application that the user is using

Open Authorization (OAuth)



Best Practices in Android Programming

References

- Device compatibility overview
  https://developer.android.com/guide/practices/compatibility
- Best Practices – Performance and Power
  https://developer.android.com/topic/performance
- Privacy best practices
  https://developer.android.com/privacy/best-practices
- App security best practices
  https://developer.android.com/topic/security/best-practices

End of Lecture 10