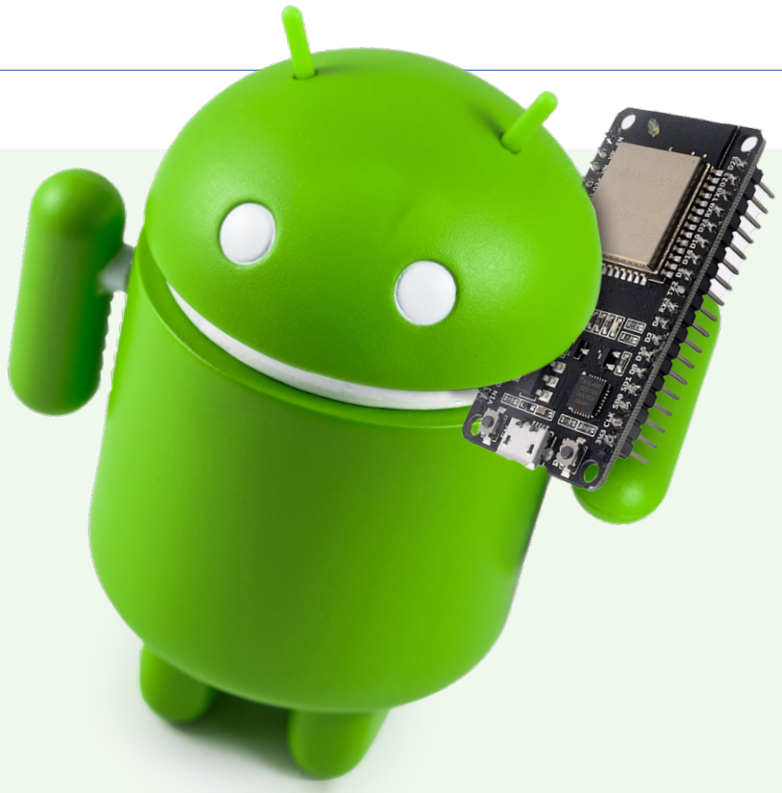
Met Android-smartphone, **spreek ik met ESP32?**



praktisch project met behulp van de Android WiFi-API

Tam Hanna (Hongarije)

Ontwikkelaars van Android-apps weten dat de code die nodig is om verbinding te maken met een WiFi-netwerk niet triviaal is, vooral wanneer je het hele proces voor de eindgebruiker zo soepel en vlot mogelijk wilt laten verlopen. Software-ontwikkelaars moeten er bovendien voor zorgen dat hun code compatibel is met verschillende Android-versies en op de hoogte zijn van de door Google vereiste beveiligingsmaatregelen, zodat de app probleemloos functioneert. Dit artikel wil je helpen een weg te vinden door deze jungle.

Als je je geen zorgen maakt over het relatief hoge energieverbruik, dan is WiFi de radiostandaard bij uitstek, vooral voor kleinere projecten. Bluetooth komt niet in aanmerking vanwege de certificerings- en licentiekosten van Bluetooth SIG. Communicatie kan plaatsvinden via een REST interface-architectuur, en WiFi-zenders zijn zowat overal verkrijgbaar.

Het is geen verrassing dat kleine ESP32-ontwikkelboards – met hun ingebouwde WiFi access points – zo populair zijn bij ontwikkelaars van Android-toepassingen.

Het Android OS is zo ontworpen dat de ervaring van de gebruikersinterface voorrang krijgt op de reactietijd op externe stimuli. Deze hoge latentie maakt het ongeschikt voor het besturen van tijdkritische gebeurtenissen. Android is uitgerust met uitgebreide GUI-stacks, en voor veeleisende taken zoals het weergeven van grafieken zijn er bibliotheken die via Gradle geladen kunnen worden.

In het geval dat een systeem een snelle reactie op real-time gebeurtenissen nodig heeft en ook UI-vereisten heeft, lijkt het logisch om de verwerking te verdelen zoals geschetst in **figuur 1**.

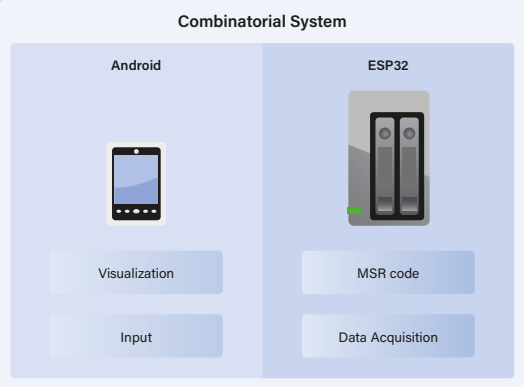
Vaak is de zwakke schakel in de keten de eindgebruiker, die uiteindelijk waarden in het systeem moet invoeren. Als de eindgebruiker 'handmatig' een verbinding met een bepaald WiFi-netwerk tot stand moet brengen, wordt uw product-ondersteuningsteam gegarandeerd overspoeld met telefoontjes van gefrustreerde klanten.

Dat zou vlotter kunnen als een automatische aansluiting op een netwerk tot stand zou kunnen worden gebracht. Google maakt zich echter zorgen over de functionele veiligheid en de bescherming van persoonsgegevens bij een dergelijke techniek.

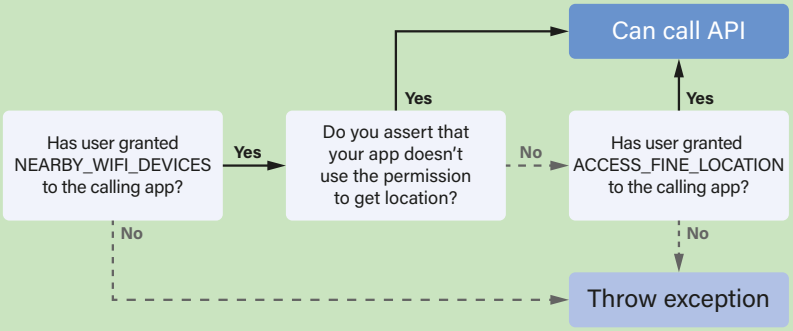
Wat willen we?

Het doel van het volgende artikel is het beschrijven van de Android-code die nodig is om een communicatieverbinding tot stand te brengen met een WiFi-doelnetwerk. Een ESP32 draait SoftAP om een remote WiFi access point op te zetten, waarmee het Android-apparaat verbinding kan maken. In de praktijk kan deze code ook worden gebruikt met veel andere controllers met WiFi-capaciteit (zie de download op [1]).

Dit artikel is gebaseerd op een project dat de auteur ontwikkelde voor een klant. Vrijwel elk ESP32-board dat een variant van het voorbeeldprogramma *esp-idf/examples/wifi/getting_started/* (opgenomen in de ESP-IDF [2]) draait, geconfigureerd als toegangspunt, zou ook geschikt zijn. We veronderstellen dat de lezer ervaring heeft met de ESP32-ontwikkelomgeving.



Figuur 1. De Android-smartphone verzorgt de GUI, terwijl de ESP32 verantwoordelijk is voor de hardware-communicatie.



Figuur 2. Voor draadloze communicatie onder Android 13 kunnen extra machtigingen nodig zijn (zie afbeelding bij [4]).

Aan de Android-kant moet je twee smartphones hebben: één met een Android-versie ≥ 10, en één met Android 9 of eerder. Android Studio [3] dient als ontwikkelomgeving. De ruimte is hier beperkt, dus ik ga ervan uit dat de lezer enige ervaring heeft met de Android IDE.

Configuratie

De compilatie-configuratie van een Android-project is een vrij ingewikkeld proces. Meestal is er een versie van het *build.gradle*-bestand dat bij de *app*-module hoort, zodat alle noodzakelijke instellingen en afhankelijkheden van de app worden ingesteld in Android Studio. Dit bestand bevat typisch statements zoals:

```
android {
    compileSdkVersion 29
    buildToolsVersion "30.0.1"
    defaultConfig {
        minSdkVersion 26
        targetSdkVersion 31
```

Met het veld `minSdkVersion` kan de ontwikkelaar de minimale versie van het Android-besturingssysteem opgeven voor het doel- of eindapparaat waarmee de app compatibel is. De waarde 26 verwijst naar Android 8 (Oreo). `compileSdkVersion` specificeert de versie van de Software Development Kit waarmee de app zal worden gecompileerd, en `buildToolsVersion` geeft aan welke versie van de Android build tools zal worden gebruikt om het project te compileren. De regel `targetSdkVersion` specificeert de versie van het Android OS waarvoor een app wordt ontwikkeld.
Dit onderscheid, dat op het eerste gezicht ingewikkeld en verwarrend klinkt, is nodig omdat het 'gedrag' van veel Android API's verandert afhankelijk van de OS-versie waaronder het draait.
Als `targetSdkVersion` is ingesteld op een bepaalde waarde, gaat het besturingssysteem ervan uit dat de ontwikkelaar bij het ontwerp van de app rekening heeft gehouden met alle wijzigingen die in deze versie zijn geïntroduceerd – als de waarde kleiner is, wordt de compatibiliteitsmodus geactiveerd.
Helaas heeft Google strikte richtlijnen opgesteld voor de waarden in `targetSdkVersion`. Als een app is ontworpen om alleen op oudere OS-versies te draaien, zal Android Studio je tijdens het compileren waarschuwen met een bericht in de vorm van 'Google Play requires that apps target API level 30 or higher', wat aangeeft dat de Play Store-backend zal weigeren de gegenereerde APK te uploaden. In de niet al te verre toekomst wil Google 'zeer oude' apps geleidelijk uit de Play Store verwijderen.
Als ontwikkelaar moet je je bewust zijn van deze beperkingen en mogelijke valkuilen.

Machtigingen

Je kunt de wereld van Android ruwweg verdelen in de tijd voor en de tijd na het nieuwe permission management system en/of Storage Access Framework: in de goede oude tijd waren ontwikkelaars over het algemeen vrij om te doen wat ze wilden, maar toen ging Google serieus werk maken van de beveiliging van persoonsgegevens.
In het geval van WiFi is de situatie netelig, aangezien informatie over de standplaats kan worden afgeleid uit de kennis van de draadloze netwerken in de omgeving (de eerste iPod touch van Apple, zonder GPS of mobiele telefoon, kon bijvoorbeeld op deze manier zijn locatie bepalen).
Het is nu nodig om de volgende declaraties in het manifest-bestand van de Android-app te specificeren om te verzoeken dat de app bepaalde machtigingen krijgt.

```
<uses-permission android:name=
"android.permission.ACCESS_FINE_LOCATION" />
<uses-permission android:name=
"android.permission.ACCESS_COARSE_LOCATION" />
<uses-permission android:name=
"android.permission.ACCESS_WIFI_STATE" />
<uses-permission android:name=
"android.permission.CHANGE_WIFI_STATE" />
<uses-permission android:name=
"android.permission.CHANGE_NETWORK_STATE" />
<uses-permission android:name=
"android.permission.ACCESS_NETWORK_STATE" />
```

Als je Android 13 in je toepassing wilt ondersteunen, moet je een extra wijziging aanbrengen: in het geval van WLAN werkt de Android 13 machtigingslogica zoals geschetst in **figuur 2**.
Het is nu nodig om de volgende declaraties in het manifest-bestand van de Android-app te specificeren om te verzoeken dat de app bepaalde machtigingen krijgt.

```
<uses-permission android:name=
"android.permission.NEARBY_WIFI_DEVICES"
android:usesPermissionFlags="neverForLocation" />
```

De `android.permission.NEARBY_WIFI_DEVICES` machtiging is een verzoek van de app-ontwikkelaar om de app toe te staan te scannen naar en toegang te krijgen tot alle WiFi-apparaten in de buurt, terwijl `android:usesPermissionFlags="neverForLocation"` een speciaal regime is dat een 'nooit voor locatie'-vlag toekent aan de app. Dit verzekert dat de app WiFi-informatie alleen gebruikt voor configuratiedoeleinden en niet voor locatie-informatie. In dit geval is de machtiging `android.permission.ACCESS_FINE_LOCATION` niet vereist onder Android 13.
Eerdere versies van het besturingssysteem vereisen deze machtiging, in welk geval een configuratie zoals deze wordt aanbevolen:

```
<uses-permission android:name=
"android.permission.ACCESS_FINE_LOCATION"
android:maxSdkVersion="32" />
```

In deze context garandeert het attribuut `maxSdkVersion` dat de toestemmingsdeclaratie alleen door eerdere Android-versies wordt herkend.

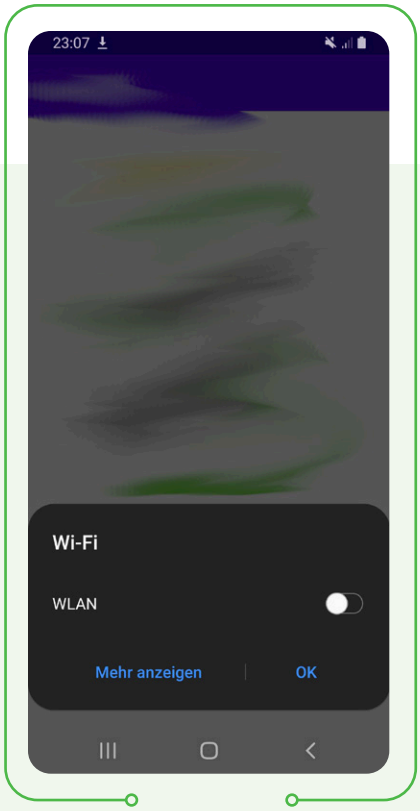
Tweesporen-ontwikkeling

Bij het instellen van de *Activity* die verantwoordelijk is voor het 'scannen' van de omgeving naar een bruikbaar WiFi-eindpunt, moeten we eerst controleren of de WiFi van de telefoon is ingeschakeld:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
...
if (android.os.Build.VERSION.SDK_INT <
       Build.VERSION_CODES.Q) {
    wifiManager =
        (WifiManager) getApplicationContext()
       .getSystemService(Context.WIFI_SERVICE);
    if (!wifiManager.isWifiEnabled()) {
      wifiManager.setWifiEnabled(true);
    }
  }
```

Als we te maken hebben met een oudere versie van Android die niet wordt beïnvloed door API-wijzigingen, kan ons programma de WiFi van de telefoon inschakelen zonder om toestemming van de gebruiker te vragen. Helaas geldt dit niet voor nieuwere versies van Android – bij deze versies is toestemming van de gebruiker vereist.
Om in runtime onderscheid te maken tussen de twee varianten, vertrouwen we op de constante `android.os.Build.VERSION.SDK_INT`. Deze verwijst naar de API-versie die wordt ondersteund door de firmware van de huidige telefoon.

Om WiFi daadwerkelijk in te schakelen, moeten we een *intent* sturen van het type `Settings.Panel.ACTION_WIFI`:



Figuur 3. Google vraagt uitdrukkelijke toestemming van de gebruiker.

```
else {
    wifiManager =
       (WifiManager) getApplicationContext().
       getSystemService(Context.WIFI_SERVICE);
    if (!wifiManager.isWifiEnabled()) {
        Toast toast =
           Toast.makeText(getApplicationContext(),
          "PLEASE enable WiFi so that we can connect!",
           Toast.LENGTH_LONG);
        toast.show();
        Intent panelIntent =
            new Intent(Settings.Panel.ACTION_WIFI);
        this.startActivityForResult(panelIntent,
           ICY_WIFI_REQCODE);
    }
}
```

Je wordt voor alle moeite beloond met het bericht in **figuur 3** dat in runtime verschijnt. De gebruiker moet de knop omzetten om het activeringsproces te autoriseren.
Het hier afgedrukte fragment vuurt een *toast* af naar de gebruiker voordat de *Intent* wordt verzonden, met de mededeling dat WiFi moet worden geactiveerd.
Indien geen actie wordt ondernomen, kan het nodig zijn de gebruiker eraan te herinneren dat hij WiFi moet activeren.
Voor Android-beginners kan het vreemd lijken dat `ICY_WIFI_REQCODE` wordt doorgegeven aan `startActivityForResult()`, maar dit is een correlatiecode waarmee het ontvangende eindpunt de trigger kan identificeren die verantwoordelijk is voor het inkomende verzoek. Correlatiecodes zijn normale integers waaraan het besturingssysteem

Goud van oud

Als je klaar bent met je eerste smartphone-app, moet je eens even teruggaan in de tijd en de basisprincipes bekijken die Jeff Hawkins gebruikte om Palm OS te maken, dat draaide op Palm- en Palm Pilot-toestellen. Ook al zijn ze nu verouderd, zijn filosofie bij het ontwerpen van Palm OS is interessant. *Zen of Palm* is niet erg lang, en je kunt een PDF-versie vinden op [5].

geen significante waarde toekent. De auteur declareert ze in het programma volgens het volgende schema; het enige dat telt is dat de gegenereerde code uniek is.

```
public class MainActivity extends
AppCompatActivity implements
AdapterView.OnItemClickListener {
    private final int MY_PERMISSIONS_
       ACCESS_COARSE_LOCATION = 1;
    int ICY_WIFI_REQCODE = 4242;
```

De volgende taak is het eigenlijke WiFi-scanproces te starten. Door ruimtegebrek kunnen we niet in de Android GUI-stack duiken, maar we vertrouwen erop dat je een manier zult vinden om de `getWifi()`-methode te activeren.
De eerste stap in de verwerking van standplaatsinformatie is controleren of we al access-machtiging hebben tot de `Manifest.permission.ACCESS_FINE_LOCATION`. Onder Android 13 moet je ook de nieuwe machtiging opnemen in de analysecode:

```
private void getWifi() {
    if (ContextCompat.
       checkSelfPermission(MainActivity.this,
       Manifest.permission.ACCESS_FINE_LOCATION) !=
       PackageManager.PERMISSION_GRANTED) {
    ActivityCompat.
         requestPermissions(MainActivity.this,
         new String[],
         MY_PERMISSIONS_ACCESS_COARSE_LOCATION);
    }
```

Vergeet niet dat, voor Android 6.0 en hoger, 'gevoelige' machtigingen alleen door het besturingssysteem worden ingeschakeld als de gebruiker daar uitdrukkelijk mee instemt. Hiervoor worden dialogen gebruikt zoals die in **figuur 4**.
Als we de toestemming al hebben gegeven, is de volgende stap nog een versiewissel. Als we te maken hebben met een eerdere versie van Android, roepen we direct de methode `getWifiWorkerOld()` aan:

```
    else {
        if (android.os.Build.VERSION.SDK_INT <
            Build.VERSION_CODES.Q) {
          getWifiWorkerOld();
        }
```

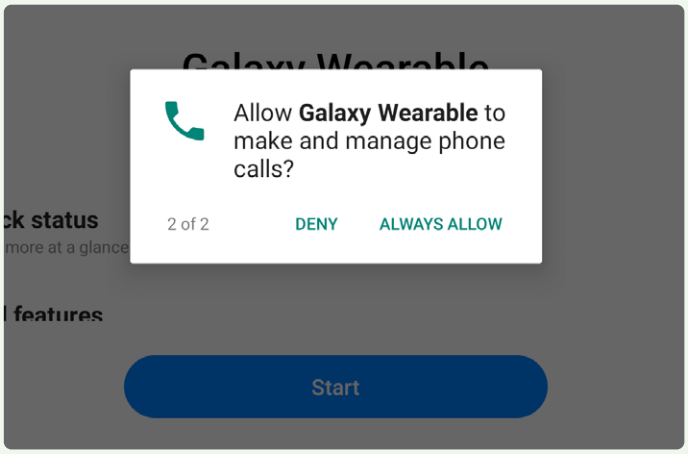
In het geval van een nieuwe telefoon wordt, om een crash te voorkomen, op dit punt de voeding van de WiFi-zender opnieuw gecontroleerd, omdat het denkbaar is dat gebruikers WiFi uitschakelen voordat de methode actief wordt.

```
    else {
        if (wifiManager.isWifiEnabled()) {
            getWifiWorkerOld();
        }
        else {
            Toast toast =
               Toast.makeText(getApplicationContext(),
              "PLEASE enable WiFi so that we can connect!",
               Toast.LENGTH_LONG);
           toast.show();
           Intent panelIntent =
               new Intent(Settings.Panel.ACTION_WIFI);
           this.startActivityForResult(panelIntent,
               ICY_WIFI_REQCODE);
        }
    }
```

Zodra we met succes de voedingstoestand van de zender hebben gecontroleerd, gaan we over tot het activeren van het scanproces met behulp van deze code:

```
private void getWifiWorkerOld() {
    ...
    wifiManager.startScan();
}
```

`wifiManager` is een systeemklasse die we verkregen bij het activeren van de activiteit. De methode `startScan()` zorgt vervolgens voor het starten van het scanproces.
Het terugzenden van de informatie die door de scanrun wordt opgevangen, gebeurt via *Broadcast*. Voor de ontvangst ervan hebben we een *BroadcastReceiver* nodig. Omdat we die niet willen aanmaken in het manifest-bestand, schrijven we hem in plaats daarvan in de methode `onPostResume()` van de activiteit:



Figuur 4. Let op het nieuwe machtigingensysteem.

```
@Override
protected void onPostResume() {
    super.onPostResume();
    receiverWifi =
       new WifiReceiver(wifiManager, wifiList, this);
    IntentFilter intentFilter =
       new IntentFilter();
    intentFilter.addAction
    (WifiManager.SCAN_RESULTS_AVAILABLE_ACTION);
    registerReceiver(receiverWifi, intentFilter);
}
```

In het belang van de compliantie met de Play Store is het belangrijk dat handmatig geregistreerde *BroadcastReceivers* ook door het besturingssysteem worden afgemeld. Dit kan het gemakkelijkst worden gedaan met behulp van een van de levenscyclusmethoden van de activiteit. Ik heb ervoor gekozen hier de methode `onPause()` te gebruiken:

```
@Override
protected void onPause() {
    super.onPause();
    unregisterReceiver(receiverWifi);
}
```

Broadcast Receiver voor gegevensverwerking

Aangezien de ontvangst van netwerkinformatie die door het WiFi scanproces wordt geretourneerd, gebeurt via het broadcast-systeem van Android, moeten we de klasse `BroadcastReceiver` implementeren die hierboven bij het besturingssysteem is geregistreerd.
BroadcastReceivers in Android hebben de vorm van klassen die zijn afgeleid van de `BroadcastReceiver` hoofdklasse. In ons voorbeeld – de GUI-variabelen negerend – ziet de 'header' er als volgt uit (we drukken de constructor die nodig is om de velden te vullen hier niet af):

```
class WifiReceiver extends BroadcastReceiver {
    WifiManager wifiManager;
    ListView wifiDeviceList;
    MainActivity myParentAct;
```

De belangrijke vraag is nu hoe de netwerkinformatie verwerkt moet worden die terugkomt van de WiFi-zenders. Het antwoord is te vinden in de methode `onReceive()`, die het besturingssysteem activeert wanneer de uitgezonden informatie, die wordt geleverd in de vorm van Intents, arriveert.

```
public void onReceive(Context context, Intent intent) {
        String action = intent.getAction();
        if (WifiManager.
           SCAN_RESULTS_AVAILABLE_ACTION.equals(action)) {
          List<ScanResult> wifiList =
              wifiManager.getScanResults();
          ArrayList<String> deviceList =
              new ArrayList<>();
          for (ScanResult scanResult : wifiList) {
              if(scanResult.SSID.
                  contains("NAMEOFTHING"))
                  deviceList.add(scanResult.SSID );
          }
          myParentAct.myArrayAdapter=
             new ArrayAdapter(context,
               android.R.layout.simple_list_item_1,
               deviceList.toArray());
           wifiDeviceList.
               setAdapter(myParentAct.myArrayAdapter);
       }
    }
}
```

In de eerste stap roept de code `intent.getAction()` aan om de naam van de Intent te controleren. *Broadcast Receiver* kan theoretisch elke intent bevatten; door te controleren aan de hand van de string `WifiManager.SCAN_RESULTS_AVAILABLE_ACTION`, zorgen we ervoor dat we te maken hebben met een 'compatibele' intent.
Aangezien het codevoorbeeld waaraan de auteur de volgende fragmenten ontleende een lijst gebruikt voor de weergave, is in de volgende stap enige aanpassing vereist. Hoe dan ook, de zin van het proces is de populatie van de klasse `deviceList`.
In een echte toepassing zou op dit punt een lijst worden gevuld met informatie, maar je kunt natuurlijk op dit punt anders te werk gaan als je slechts een enkel WiFi-doelnetwerk als geldig beschouwt.
Een voorbeeld van de andere aanpak is een apparaat dat tijdens de configuratie contact maakt met zijn basistoepassing en dit aangeeft met een specifieke WiFi-naam. In dat geval zouden we direct na het detecteren van dat netwerk verder kunnen gaan.

Een nette verbindingssetup

Gebruikers houden niet van lange wachttijden bij smartphone-toepassingen. Daarom is het zinvol voortgangsbalken of soortgelijke gadgets te tonen die de gebruiker verzekeren dat de toepassing niet is vastgelopen.
In de client-toepassing die als basis dient voor dit artikel, is dit geïmplementeerd in de vorm van een speciale activity: deze toont een bedrijfslogo en een voortgangsbalk, en zorgt zo voor enige geruststelling. Hij wordt geactiveerd door een intent te versturen:

```
public void onItemClick(AdapterView<?>
adapterView, View view, int pos, long anID) {
    String y = (String) myArrayAdapter.getItem(pos);
    Intent intent =
      new Intent(MainActivity.this,ConnActivity.class);
    intent.putExtra("WIFINAME", y);
    startActivity(intent);
}
```

In 'Code Behind' begint het met het declareren van een *AsyncTask* die een aanwezigheidstest uitvoert. Dit is een verificatie (hier niet beschreven) dat de connection peer deel uitmaakt van het client-ecosysteem:

Pas op voor de Google-blokkade

Vanaf Android 11 heeft Google 'eenmalige machtigingen' ingevoerd, waarmee de gebruiker een app eenmalig toegang kan verlenen tot een bepaalde functie of gegevens. Dit verandert ook de manier waarop apps op de achtergrond toestemming vragen, waardoor sommige verzoeken kunnen worden geblokkeerd. De redenering hierachter is om gebruikers meer controle over hun gegevens te geven en de privacy op Android te verbeteren. Ontwikkelaars moeten zich bewust zijn van deze potentiële problemen.

```
public class ConnActivity extends AppCompatActivity {
    PresenceTestAsynctask aT;
    ConnActivity mySelf;
```

In `onCreate()` begint de verbindingsopbouw zonder verdere actie van de gebruiker. De eerste officiële actie is het configureren van enkele member-variabelen en het aanroepen van de `getIntent().getExtras()` methodes om de informatie – eerder 'verpakt' door Intent – toegankelijk te maken met de WiFi-naam en andere ondersteunende gegevens:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_conn);
    mySelf=this;
    Bundle b = getIntent().getExtras();
    String networkPass = "sdsds";
```

De volgende stap is nog een versiewissel die de 'versiestatus' van de doel-hardware controleert.
Als we te maken hebben met een eerdere versie van Android, is de volgende stap de start van de verbindingsopbouw volgens de onderstaande code:

```
if (android.os.Build.VERSION.SDK_INT <
Build.VERSION_CODES.Q) {
    WifiConfiguration conf =
       new WifiConfiguration();
    conf.SSID = "\"" +
      b.getString("WIFINAME") + "\"";
    conf.preSharedKey = "\"" + networkPass + "\"";
    WifiManager wifiManager =
       (WifiManager) getApplicationContext().
       getSystemService(Context.WIFI_SERVICE);
    wifiManager.addNetwork(conf);
```

De methode `addNetwork()` neemt een object van het type `WifiConfiguration` over. Diens taak is het leveren van een complete WLAN-informatieset, waarmee de telefoon vervolgens verbinding kan maken.
In onze laatste handeling moeten we de verbinding opzetten volgens het volgende schema, wat gebeurt met drie methoden: `disconnect()`, `enableNetwork()` en `reconnect()`. Daarna volgt de activering van de AsyncTask – als het externe station met succes is geauthenticeerd, begint de clienttoepassing met de eigenlijke configuratie van de aangesloten hardware:

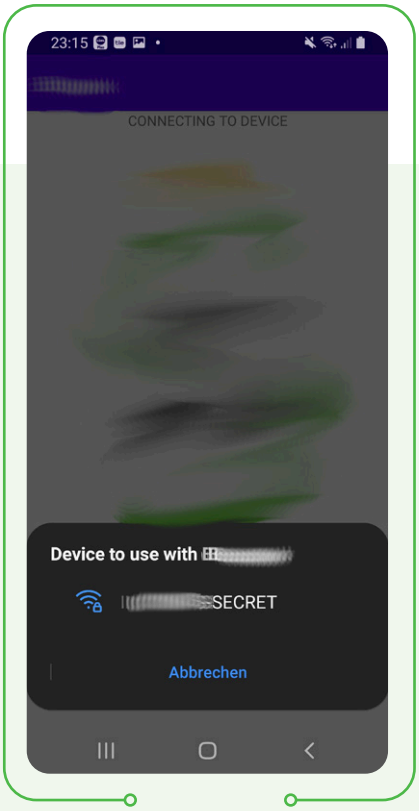
```
//Can only come here from permission granted state
@SuppressLint("MissingPermission")
List<WifiConfiguration> list =
   wifiManager.getConfiguredNetworks();
for (WifiConfiguration i : list) {
    if (i.SSID != null && i.SSID.equals("\"" +
        b.getString("WIFINAME") + "\"")) {
      final ConnectivityManager connectivityManager =
        (ConnectivityManager)getApplicationContext().
        getSystemService(Context.CONNECTIVITY_SERVICE);
      ConnectivityManager.NetworkCallback
          networkCallback =
          new ConnectivityManager.NetworkCallback() {
        @Override
       public void onAvailable(@NonNull Network network) {
          super.onAvailable(network);
          //ONLY IF ANDROID 9
          if (android.os.Build.VERSION.SDK_INT ==
               Build.VERSION_CODES.Q) {
            connectivityManager.
                 bindProcessToNetwork(network);
          }
        }
    };
    connectivityManager.
       registerDefaultNetworkCallback(networkCallback);

    wifiManager.disconnect();
    wifiManager.enableNetwork(i.networkId, true);
    wifiManager.reconnect();
    aT = new PresenceTestAsynctask
        (getApplicationContext());
    aT.myAct=mySelf;
    aT.execute("...");
    break;
  }
}
```

Bij nieuwere versies van Android is op dit punt opnieuw interventie van de gebruiker vereist. Het werk begint met het aanmaken van een `NetworkSpecifier`-object zoals dit:

```
}else{
    final NetworkSpecifier specifier =
        new WifiNetworkSpecifier.Builder()
        .setSsidPattern(new PatternMatcher(
          b.getString("WIFINAME"),
          PatternMatcher.PATTERN_PREFIX))
        .setWpa2Passphrase(networkPass)
        .build();
```

`NetworkSpecifier`-objecten zijn een soort zoekfilter die de Android GUI helpt draadloze netwerk-peers te identificeren die relevant zijn voor het specifieke gebruiksgeval.



Figuur 5. Android Q vraagt de gebruiker alvorens een verbinding tot stand te brengen.

De volgende stap is het bouwen van een `NetworkRequest`-element, hetgeen als volgt gebeurt:

```
final NetworkRequest request =
    new NetworkRequest.Builder()
        .addTransportType(NetworkCapabilities.
            TRANSPORT_WIFI)
        .removeCapability(NetworkCapabilities.
            NET_CAPABILITY_INTERNET)
        .setNetworkSpecifier(specifier)
        .build();
```

De eigenlijke setup van de verbinding vindt plaats zoals getoond in **figuur 5**. De gebruiker moet de naam invoeren van het WiFi-netwerk dat het configuratiedoel wordt.
Een neveneffect van deze procedure is dat de WiFi-verbinding asynchroon tot stand moet worden gebracht. Feedback over succes en mislukking wordt gegeven via een callback die moet worden aangemaakt in de toepassing die verantwoordelijk is voor de activering:

```
final ConnectivityManager connectivityManager =
(ConnectivityManager)getApplicationContext().
getSystemService(Context.CONNECTIVITY_SERVICE);
final ConnectivityManager.NetworkCallback
      networkCallback =
      new ConnectivityManager.NetworkCallback() {
    @Override
    public void onAvailable(@NonNull Network network) {
       super.onAvailable(network);
          connectivityManager.
              bindProcessToNetwork(network);
          aT = new PresenceTestAsynctask
              (getApplicationContext());
          aT.myAct=mySelf;
          aT.execute(". . .");
       }
       @Override
       public void onUnavailable() {
          super.onUnavailable();
       }
};
```

Als de methode `onAvailable()` wordt aangeroepen, is de verbinding met succes tot stand gebracht. In dat geval start de toepassing de AsyncTask opnieuw om de remote end te authenticeren en indien nodig aanvullende acties uit te voeren.
De laatste actie van onze scanmethode is dan het aanroepen van `requestNetwork()`, dat de hierboven getoonde gebruikersinterface activeert.

```
            connectivityManager.
              requestNetwork(request, networkCallback);
        }
    }
}
```

Is het al die moeite waard?

Er zijn vaak veel manieren om een engineering-probleem op te lossen. Het is waar dat real-time besturingssystemen – ik denk hier aan het Azure RTOS – samen met hun ingebedde GUI-stacks en draaiend op een microcontroller, inderdaad krachtige grafische interfaces kunnen realiseren in een single-device oplossing die de meeste geavanceerde gebruikssituaties zal dekken.
Om dezelfde functionaliteit te bereiken als met de hier beschreven Android-configuratie, zouden hogere hardwarekosten ontstaan – er komt geen touchscreen voor de GUI, aangezien de bijbehorende logica, zoals een framebuffergeheugen, teveel onderdelen zou vergen. Bovendien moeten we denken aan de tijd en moeite die nodig zijn om het systeem te ontwikkelen: als je bijvoorbeeld de overdracht van beelden en sensormetingen per e-mail zou willen implementeren in een real-time besturingssysteem, moet je een paar dagen uittrekken voor het coderen en je ervan bewust zijn dat de extra belasting gevolgen kan hebben voor de systeemlatentie. De Arduino-combinatie biedt een gelikte GUI op een apparaat dat de meesten van ons al bij zich dragen. Het idee om je telefoon tevoorschijn te halen en aan te sluiten op een lokaal toegangspunt, om wat voor reden dan ook, is best wel cool.
Als je dit project wilt gaan ontwikkelen, hoop ik dat sommige van de hier beschreven experimenten en tweaks je op weg door de jungle zullen helpen. Als je succes hebt of betere oplossingen vindt, horen we graag hoe het je vergaat; schrijf ons!

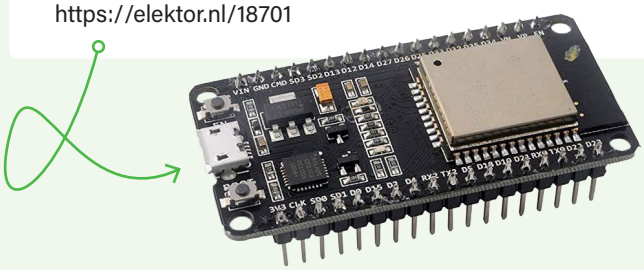
210229-03

Vragen of opmerkingen?

Hebt u technische vragen of opmerkingen naar aanleiding van dit artikel? Stuur een e-mail naar de auteur via tamhan@tamoggemon.com of naar de redactie van Elektor via redactie@elektor.com.

Gerelateerde producten

- **ESP32-PICO-Kit V4 (with female headers) (SKU 20323)**
  https://elektor.nl/20323
- **ESP32-DevKitC-32D (SKU 18701)**
  https://elektor.nl/18701



WEBLINKS

[1] De webpagina van dit artikel: https://elektormagazine.nl/210229-03
[2] ESP-IDF: https://idf.espressif.com/
[3] Android Studio: https://developer.android.com/studio
[4] Android Documentation: Request permission to access nearby Wi-Fi devices: https://elektor.link/AndroidWiFiPermission
[5] Zen of Palm (2003) [PDF]: https://cs.uml.edu/~fredm/courses/91.308-fall05/palm/zenofpalm.pdf