Pázmány Péter Catholic University
Faculty of Information Technology and Bionics

Android Development

Wear, Auto, ML Kit

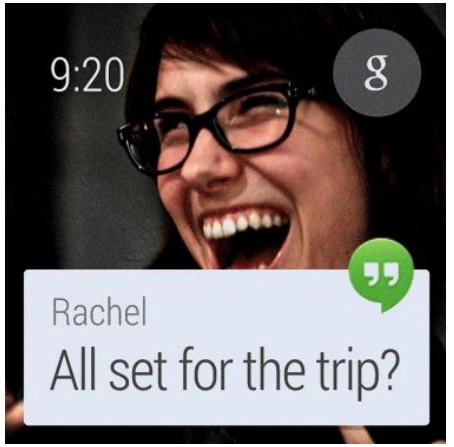
Pázmány Péter Catholic University
Faculty of Information Technology and Bionics

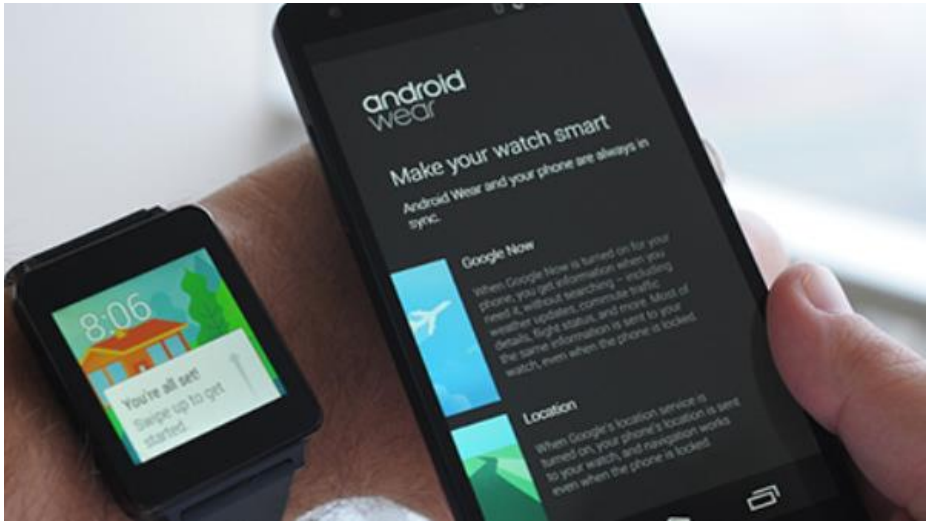
Wear OS by Google

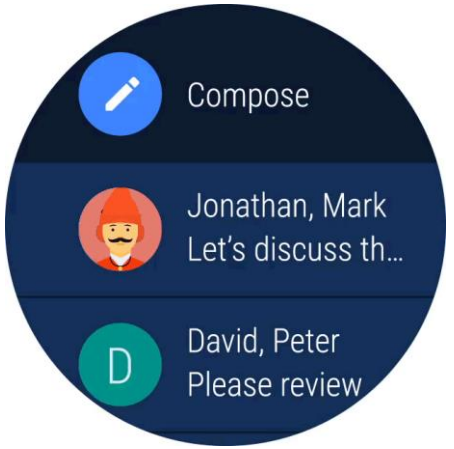
Android Wear

Wear OS

- Applications running on wearable devices
  - Introduced in 2014 (Android Wear)
  - 2017: Android Wear 2.0
  - 2018: Rebranded: Wear OS





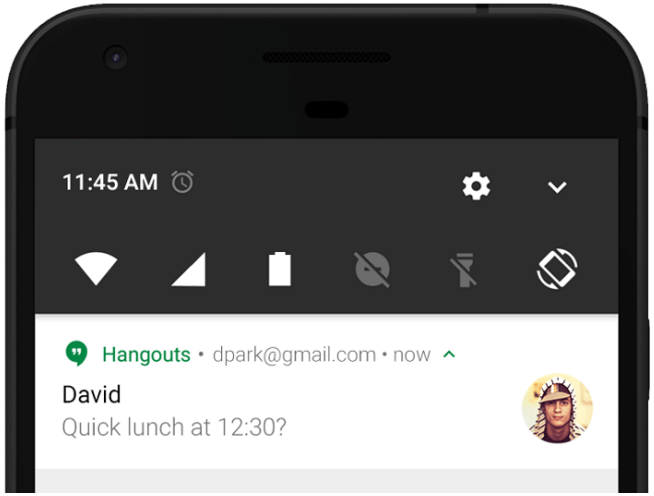


Different methods

- The simplest application is also capable of utilizing the possibilities of Wearable devices
  - Extension of notifications
- Complete, „standalone" applications for devices
  - It is often part of the application running on the phone
- Watchfaces

Extension of existing applications

- Notifications on watches have the same structure as notifications on phones.
  - Wear OS provides APIs for adding wearable-specific features to notifications.
  - When you issue a notification from your app, each notification appears as a card on the Notification Stream.



Extension of existing applications

- Further capabilities of notifications
  - Using `Notification.Builder`
  - `NotificationCompat.Builder` (JetPack, AndroidX library)

```
dependencies {
    implementation 'androidx.core:core:1.2.0'
}
```

- Reminder

```
var builder = NotificationCompat.Builder(this,
CHANNEL_ID)
        .setSmallIcon(R.drawable.notification_icon)
        .setContentTitle(textTitle)
        .setContentText(textContent)

.setPriority(NotificationCompat.PRIORITY_DEFAULT)
```

Reminder continued

```
private fun createNotificationChannel() {

   if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.O) {
        val name = getString(R.string.channel_name)
        val descriptionText = getString(R.string.channel_description)
        val importance = NotificationManager.IMPORTANCE_DEFAULT
        val channel = NotificationChannel(CHANNEL_ID, name, importance).apply
        {
            description = descriptionText

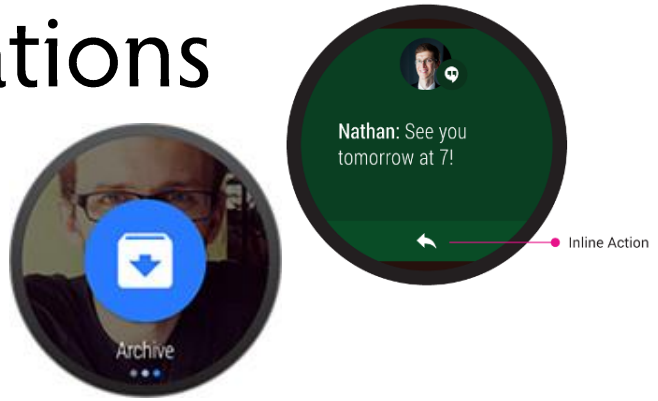
        }


    val notificationManager: NotificationManager =
        getSystemService(Context.NOTIFICATION_SERVICE) as NotificationManager
        notificationManager.createNotificationChannel(channel)

    }

}
```

Extension of existing applications

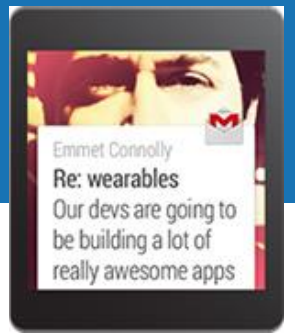


- Adding action button
  - Using builder
    - `.addAction(R.drawable.ic_map, getString(R.string.map), mapPendingIntent)`
- Specific commands
  - An Intent

```
val mapPendingIntent = Intent(Intent.ACTION_VIEW).let { mapIntent ->
    mapIntent.data = Uri.parse("geo:0,0?q=" + Uri.encode(location))
    PendingIntent.getActivity(this, 0, mapIntent, 0)
}
```

  - Extension (inline Action)

```
val actionExtender = NotificationCompat.Action.WearableExtender()
        .setHintLaunchesActivity(true)
        .setHintDisplayActionInline(true)
wearableExtender.addAction(actionBuilder.extend(actionExtender).build())
```



Extension of existing applications

```
var notification = NotificationCompat.Builder(context, CHANNEL_ID)
        .setSmallIcon(R.drawable.new_mail)
        .setContentTitle(emailObject.getSenderName())
        .setContentText(emailObject.getSubject())
        .setLargeIcon(emailObject.getSenderAvatar())
        .setStyle(NotificationCompat.BigTextStyle()
                .bigText(emailObject.getSubjectAndSnippet()))
        .build()

var notification = NotificationCompat.Builder(context, CHANNEL_ID)
        .setSmallIcon(R.drawable.new_mail)
        .setContentTitle("5 New mails from " + sender.toString())
        .setContentText(subject)
        .setLargeIcon(aBitmap)
        .setStyle(NotificationCompat.InboxStyle()
                .addLine(messageSnippet1)
                .addLine(messageSnippet2))
        .build()
```

Smart reply

```
val replyPendingIntent = Intent(this,
ReplyActivity::class.java).let { replyIntent ->
  PendingIntent.getActivity(this, 0, replyIntent,
  PendingIntent.FLAG_UPDATE_CURRENT)
}

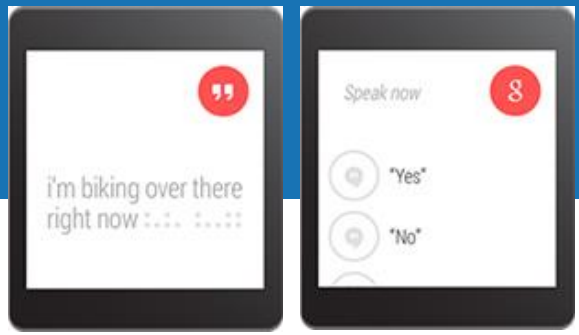
val action = NotificationCompat.Action.Builder(
  R.drawable.ic_reply_icon,
    getString(R.string.label),
    replyPendingIntent)
  .addRemoteInput(remoteInput)
  .setAllowGeneratedReplies(true)
  .build()
```

Smart reply

```
val noti = NotificationCompat.Builder(context, channelId)
 .setContentTitle("${messages.size} new messages with $sender")
 .setContentText(subject)
 .setSmallIcon(R.drawable.new_message)
 .setLargeIcon(aBitmap)

 .setStyle(
     NotificationCompat.MessagingStyle(
       resources.getString(R.string.reply_name))
       .addMessage(messages[0].text,
          messages[0].time, messages[0].sender)
       .addMessage(messages[1].text,
          messages[1].time, messages[1].sender)
     )

 .extend(NotificationCompat.WearableExtender()
    .addAction(action)).build()
```

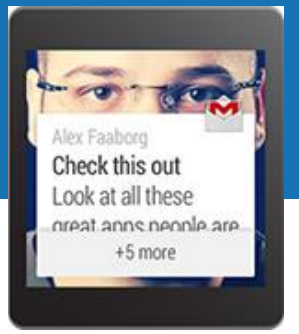


Receiving voice command

- Wearable devices are equipped with microphones, thus voice control can be implemented
  - Receiving arbitrary text

```
const val EXTRA_VOICE_REPLY = "extra_voice_reply"
...
val remoteInput = resources.getString(R.string.reply_label).let { replyLabel
->
    RemoteInput.Builder(EXTRA_VOICE_REPLY)
            .setLabel(replyLabel)
            .build()
}
```

  - Defining a set of possible texts
    -
```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string-array name="reply_choices">
        <item>Yes</item>
        <item>No</item>
        <item>Maybe</item>
    </string-array>
</resources>
```
    - `.setChoices(replyChoices)`



Grouping

- Defining a group
  - `setGroup(GROUP_KEY_WORK_EMAIL)`
- Creating a summary for the group

```
val summaryNotification =
NotificationCompat.Builder(this@MainActivity, CHANNEL_ID)
        .setContentTitle(emailObject.getSummary())
        .setContentText("Two new messages")
        .setSmallIcon(R.drawable.ic_notify_summary_status)
        .setStyle(NotificationCompat.InboxStyle()
                .addLine("Alex Faarborg Check this out")
                .addLine("Jeff Chang Launch Party")
                .setBigContentTitle("2 new messages")
                .setSummaryText("janedoe@example.com"))
        .setGroup(GROUP_KEY_WORK_EMAIL)
        .setGroupSummary(true)
        .build()
```

Standalone application

- Wear OS apps run directly on a watch
  - giving you access to hardware such as sensors and the GPU
  - apps are similar to other apps that use the Android SDK
  - differ in design and functionality
- Define an app as a Wear app

```
<manifest>
...
  <uses-feature android:name="android.hardware.type.watch" />
...
</manifest>
```

Standalone application

- A Wear app should work independently of a phone app
  - allowing users the greatest flexibility in their choice of phones
- But, a watch app can be
  - Completely independent of a phone app
  - Semi-independent
    - a phone app is not required and would provide only optional features
  - Dependent on a phone app

```
<application>
...
  <meta-data
    android:name="com.google.android.wearable.standalone"
    android:value="true" />
...
</application>
```

Standalone application

- What is required
  - Appropriate SDK
  - AVD
  - Project
    - Activity for mobile device
    - Activity for wearable device
  - Starting a wearable application is similar to the mobile application
    - Run
    - adb install

Standalone application

- The following are some of the differences between phone and watch apps:
  - Watch apps use watch-specific APIs, where applicable (e.g., for circular layouts, wearable drawers, ambient mode, etc.).
  - Watch apps contain functionality appropriate to a watch.
  - Watch apps can access many standard Android APIs, but don't support the following:
    - android.webkit
    - android.print
    - android.app.backup
    - android.appwidget
    - android.hardware.usb
  - You can check if a watch supports a feature by calling hasSystemFeature() before using an API.

Standalone application

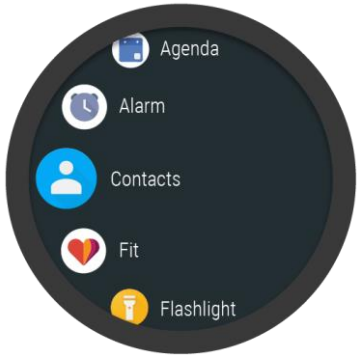
- Libraries
  - AndroidX, for cases where, backward compatibility is required
  - Wearable Data Layer API
  - Play Services
  - Wearable UI support lib

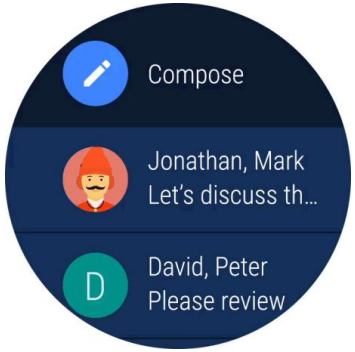
Standalone application

- The Wear UI Library includes (but is not limited to) the following classes.
  - BoxInsetLayout
    - A layout that applies insets for round screens.
  - SwipeDismissFrameLayout
    - A layout that enables a user to dismiss any view by swiping on the screen from left to right.
  - WearableRecyclerView
    - A view that provides a curved layout, such as the layout used for the main Wear application launcher.
  - AmbientModeSupport.
    - A class used to provide support for ambient mode.

Additional views

- Creating a Curved Layout
  - To create a curved layout for scrollable items in your wearable app
    - Use WearableRecyclerView.
    - Use the WearableRecyclerView.setLayoutManager() method to set the offsetting logic.
- Navigation drawer
  - The navigation drawer lets users switch between views of your app, similar to the navigation drawer on a phone.
  - Single page navigation drawer
    - You can present the contents of a navigation drawer in a single page or as multiple pages
- Action Drawer
  - The action drawer provides easy access to common actions in the app.





Drawer layout

- To add an action or a navigation drawer to your app, declare a user interface with a WearableDrawerLayout object as the root view of the layout.

```
<androidx.wear.widget.drawer.WearableDrawerLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:id="@+id/drawer_layout"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
    <ScrollView
        android:id="@+id/content"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:nestedScrollingEnabled="true">
        <LinearLayout
            android:id="@+id/linear_layout"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:orientation="vertical" />
    </ScrollView>
```

Drawer layout

- To add an action or a navigation drawer to your app, declare a user interface with a WearableDrawerLayout object as the root view of the layout.

```
<android.support.wear.widget.drawer.WearableNavigationDrawerView
    android:id="@+id/top_navigation_drawer"
    android:layout_width="match_parent"
    android:layout_height="match_parent"/>
<android.support.wear.widget.drawer.WearableActionDrawerView
    android:id="@+id/bottom_action_drawer"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    app:actionMenu="@menu/action_drawer_menu"/>
</androidx.wear.widget.drawer.WearableDrawerLayout>
```

Packaging

- Differences between wear 1 and wear 2 apps
- Android Wear 1.X
  - Must be embedded to the Phone Application
  - APK inside the main APK
  - Same key must be used to sign

```
dependencies {
    compile 'com.google.android.gms:play-services-wearable:10.0.1'
    compile 'com.android.support:support-compat:25.1.0'
    wearApp project(':wearable')
}
```

Packaging

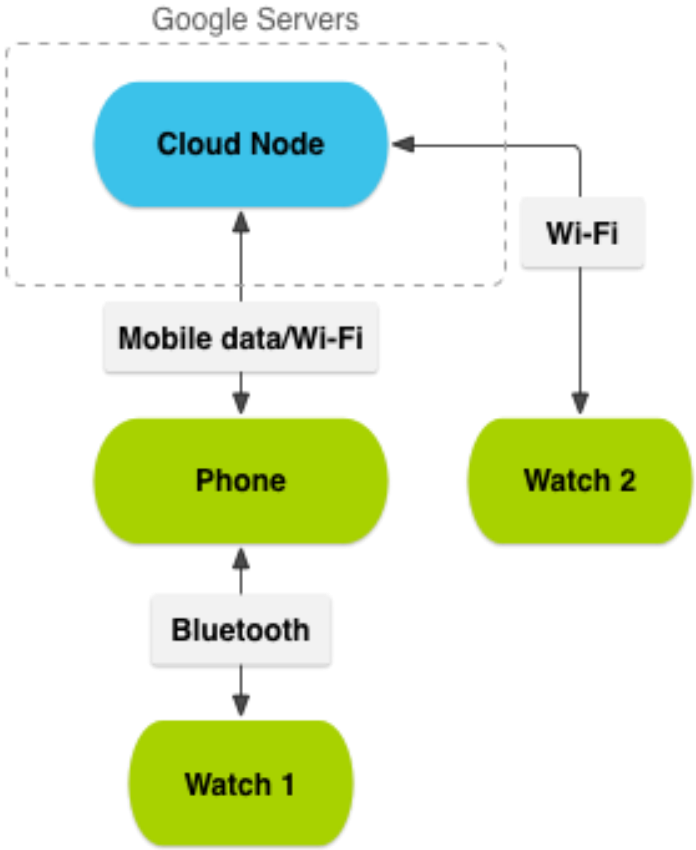
- Android Wear 2.X and Wear OS
  - Can be distributed separately
    - It can be uploaded as a normal application
    - It will appear in the on-watch Play Store
  - Wear application as a companion application
    - Multi-APK delivery method if wear application is related with a phone application
    - Notification about the wear application will be displayed
      - Automatic update

Debugging

- Wearable can be debugged via
  - USB
  - Wi-Fi, directly
  - Bluetooth, through the host device
- All of them has to be enabled firs
  - And developer mode also has to be enabled
    - As on the phone

Data synchronization

- Data synchronization is an essential part of the wearable device
  - The framework supports the synchronization between more device
  - Sending data, messages
  - For services as well



Sending and Syncing Data

- Capabilities
  - Data Item
    - A DataItem provides data storage with automatic syncing between the handheld and wearable.
  - Asset
    - Asset objects are for sending binary blobs of data, such as images
    - Assets are attached to data items and the system automatically sends
      - conserving Bluetooth bandwidth by caching large assets to avoid re-transmission.
  - Message
    - A MessageClient can send messages
      - good for remote procedure calls (RPC),
      - for one-way requests or for a request/response communication model.

Sending and Syncing Data

- Capabilities
  - Channel
    - You can use a ChannelClient to transfer large entities
      - Transfer large data files between two or more connected devices saves disk space over DataClient
      - Reliably send the files.
      - Transfer streamed data
  - WearableListenerService (for services)
    - Extending WearableListenerService lets you listen for important data layer events in a service.

Sending and Syncing Data

- Capabilities
  - OnDataChangedListener (for foreground activities)
    - lets you listen for important data layer events when an activity is in the foreground.
- These APIs are designed for communication between handhelds and wearables
  - Should only be used
- With Android Wear 2.0, a watch can communicate with a network directly, without access to an Android or iOS phone

Network Access

- Android Wear apps can make network request
  - When a watch has a Bluetooth connection to a phone, the watch's network traffic generally is proxied through the phone
  - But when a phone is unavailable, Wi-Fi and cellular networks are used, depending on the hardware
- The Android Wear platform manages network connectivity. The platform chooses the default, active network by balancing two factors:
  - The need for long battery life
  - The need for network bandwidth
- Acquiring a High-Bandwidth Network
  - You can use the ConnectivityManager class to check if an active network exists and has enough bandwidth
- Launching the Wi-Fi Settings Activity

```
context.startActivity(Intent("com.google.android.clockwork.settings.connectivity.wifi.ADD_NETWORK_SETTINGS"))
```

Accessing the data layer

Using GoogleApiClient

```
val dataClient: DataClient = Wearable.getDataClient(context)


val dataClient: DataClient =
   Wearable.WearableOptions.Builder().setLooper(myLooper).build()
      .let { options ->
          Wearable.getDataClient(context, options)
      }
```

DataItem

- A `DataItem` has two parts
  - `Payload` and `Path`
  - The payload is a byte array, maximized in 100 kB
  - The path resembles to an absolute path in a file system: `/path/to/data`
- It is not instantiated directly
  - `PutDataRequest` by giving the path
  - `setData()` sets the data
  - `DataApi.putDataItem()` creates the `DataItem`
- Instead of byte array a `DataMap` also can be used

DataMap

```
private const val COUNT_KEY = "com.example.key.count"

class MainActivity : Activity() {

    private lateinit var dataClient: DataClient
    private var count = 0
    ...
    // Create a data map and put data in it
    private fun increaseCounter() {
        val putDataReq: PutDataRequest =
          PutDataMapRequest.create("/count").run {
            dataMap.putInt(COUNT_KEY, count++)
            asPutDataRequest()
        }
        val putDataTask: Task<DataItem> =
           dataClient.putDataItem(putDataReq)
    }
    ...
}
```

Listener

```
private const val COUNT_KEY = "com.example.key.count"
class MainActivity : Activity(),
DataClient.OnDataChangedListener {

    private var count = 0

    override fun onResume() {
        super.onResume()
        Wearable.getDataClient(this).addListener(this)
    }

    override fun onPause() {
        super.onPause()
        Wearable.getDataClient(this).removeListener(this)
    }
```

Listener

```
    override fun onDataChanged(dataEvents: DataEventBuffer) {
        dataEvents.forEach { event ->
            if (event.type == DataEvent.TYPE_CHANGED) {
                event.dataItem.also { item ->
                    if (item.uri.path.compareTo("/count") == 0) {
                        DataMapItem.fromDataItem(item).dataMap.apply {
                            updateCount(getInt(COUNT_KEY))
                        }
                    }
                }
            } else if (event.type == DataEvent.TYPE_DELETED) {
                …
            }
        }
    }

    private fun updateCount(int: Int) { ... }

}
```

Sending messages

- A message with arbitrary content can be sent
  - It can be used for calling procedures, or starting Activities
    - Long calculations performed on the handheld
- Advertise capabilities
  - To launch an activity on a handheld device from a wearable device, use the MessageApi class to send the request
  - `res/values/wear.xml`
    - ```
      <resources>
          <string-array name="android_wear_capabilities">
              <item>voice_transcription</item>
          </string-array>
      </resources>
      ```
      ```

Retrieve the nodes with the required capabilities

```
private const val VOICE_TRANSCRIPTION_CAPABILITY_NAME = "voice_transcription"
...
private fun setupVoiceTranscription() {
    val capabilityInfo: CapabilityInfo = Tasks.await(
            Wearable.getCapabilityClient(context)
                    .getCapability(
                            VOICE_TRANSCRIPTION_CAPABILITY_NAME,
                            CapabilityClient.FILTER_REACHABLE
                    )
    )
    // capabilityInfo has the reachable nodes with the transcription capability
    updateTranscriptionCapability(capabilityInfo)
}
```

Retrieve the nodes with the required capabilities

```
private var transcriptionNodeId: String? = null

private fun updateTranscriptionCapability(capabilityInfo: CapabilityInfo) {
    transcriptionNodeId = pickBestNodeId(capabilityInfo.nodes)
}

private fun pickBestNodeId(nodes: Set<Node>): String? {
    // Find a nearby node or pick one arbitrarily
    return nodes.firstOrNull { it.isNearby }?.id ?: nodes.firstOrNull()?.id
}


private fun getNodes(): Collection<String> {
    return Tasks.await(Wearable.getNodeClient(context).connectedNodes).map { it.id }
}
```

Sending the message

```
const val VOICE_TRANSCRIPTION_MESSAGE_PATH = "/voice_transcription"
...
private fun requestTranscription(voiceData: ByteArray) {
    transcriptionNodeId?.also { nodeId ->
        val sendTask: Task<*> = Wearable.getMessageClient(context).sendMessage(
                nodeId,
                VOICE_TRANSCRIPTION_MESSAGE_PATH,
                voiceData
        ).apply {
            addOnSuccessListener { ... }
            addOnFailureListener { ... }
        }
    }
}
```

Receiving the message

```
fun onMessageReceived(messageEvent: MessageEvent) {
    if (messageEvent.path == VOICE_TRANSCRIPTION_MESSAGE_PATH) {
        val startIntent = Intent(this, MainActivity::class.java).apply {
            addFlags(Intent.FLAG_ACTIVITY_NEW_TASK)
            putExtra("VOICE_DATA", messageEvent.data)
        }
        startActivity(this, startIntent)
    }
}
```

Auto

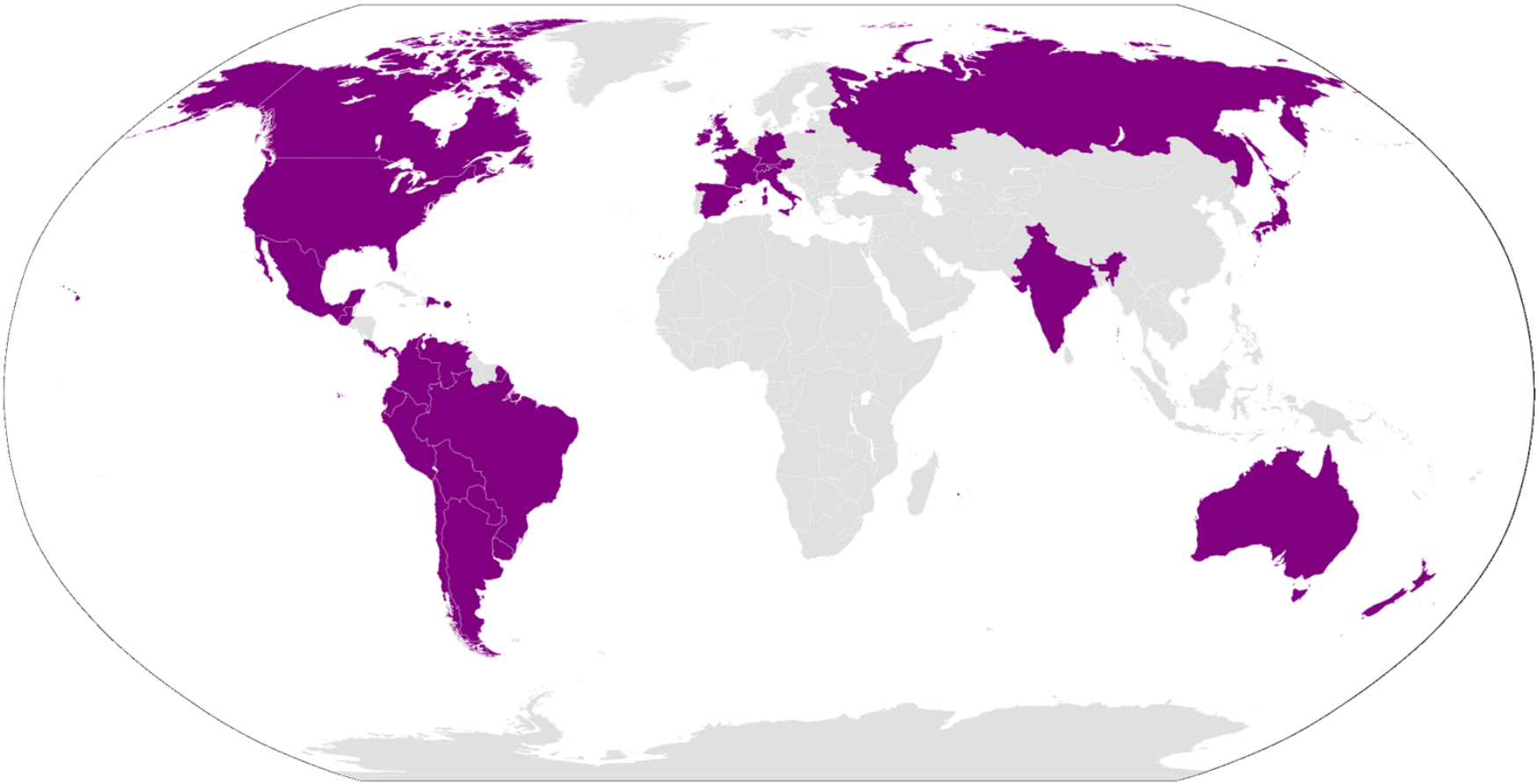
Auto

- Introduced with the Android Lollipop
  - Compatible functions with the onboard (navigation, radio, multimedia) device of the car
- Functions
  - GPS and navigation
  - Music playing and controlling
  - Phone services
  - SMS writing and reading (Voice command and TTS)
  - Searching and browsing on web
- Similar developing and connecting as introduces in Wear API

Auto

- Supported hardware
  - GPS
  - Steering wheel
  - Sound system
  - Speed measurements
  - Using antennae
  - Status of the car
- Partners
  - Abarth, Acura, Alfa Romeo, Audi, Bentley, Chevrolet, Chrysler, Dodge, Fiat, Ford, Honda, Hyundai, Infiniti, Jeep, Kia, Maserati, Mazda, Mitsubishi, Nissan, Opel, RAM, Renault, SEAT, Škoda, Subaru, Suzuki, Volkswagen, Volvo
  - Kenwood, Pioneer

Accessible



Pázmány Péter Catholic University
Faculty of Information Technology and Bionics

NNAPI

Android Neural Networks API

NNAPI

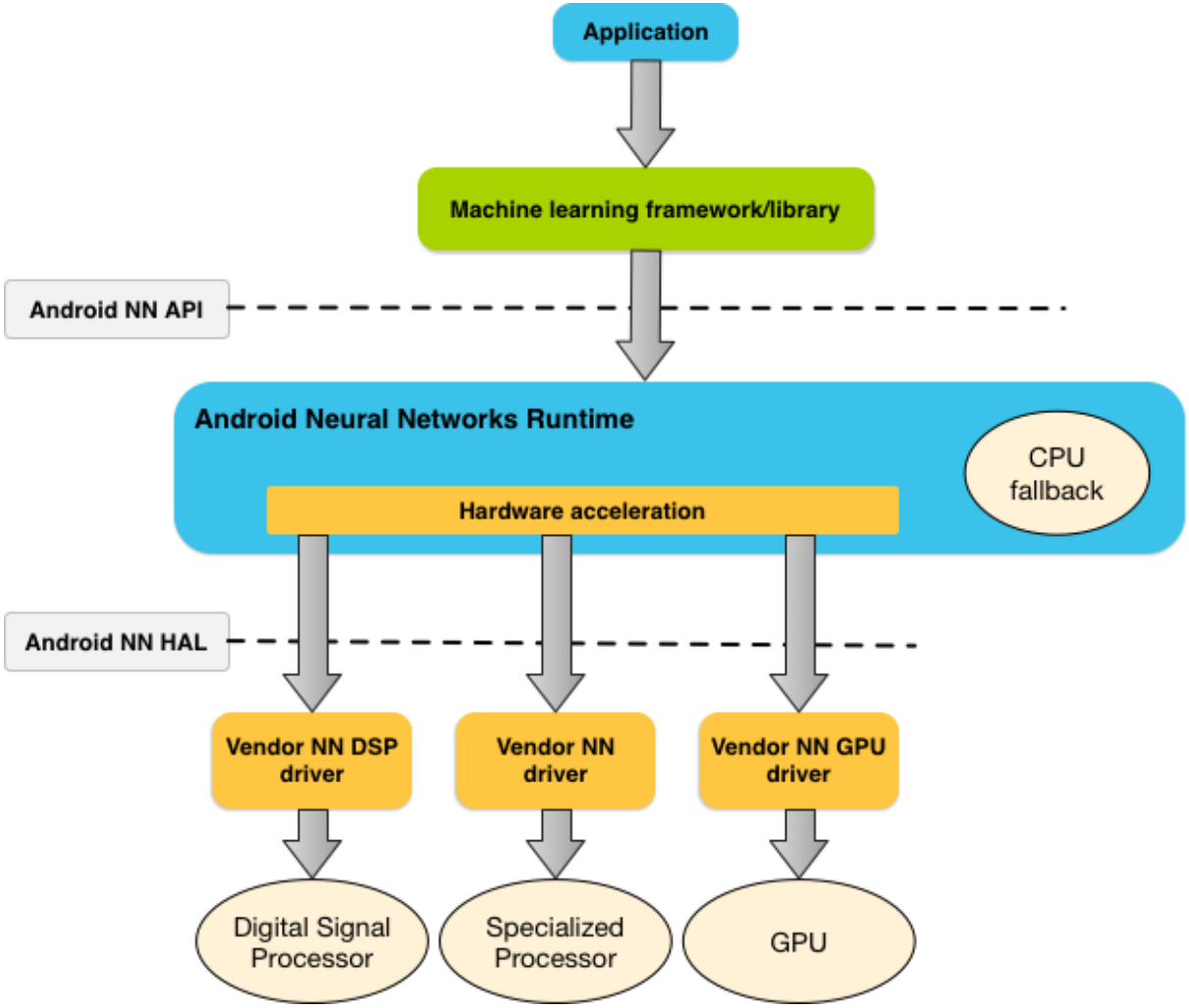
- The Android Neural Networks API (NNAPI) is an Android C API designed for running computationally intensive operations for machine learning on Android devices.
- NNAPI is designed to provide a base layer of functionality for higher-level machine learning frameworks, such as
  - TensorFlow Lite
  - Caffe2,
- The API is available on all Android devices running Android 8.1 (API level 27) or higher.

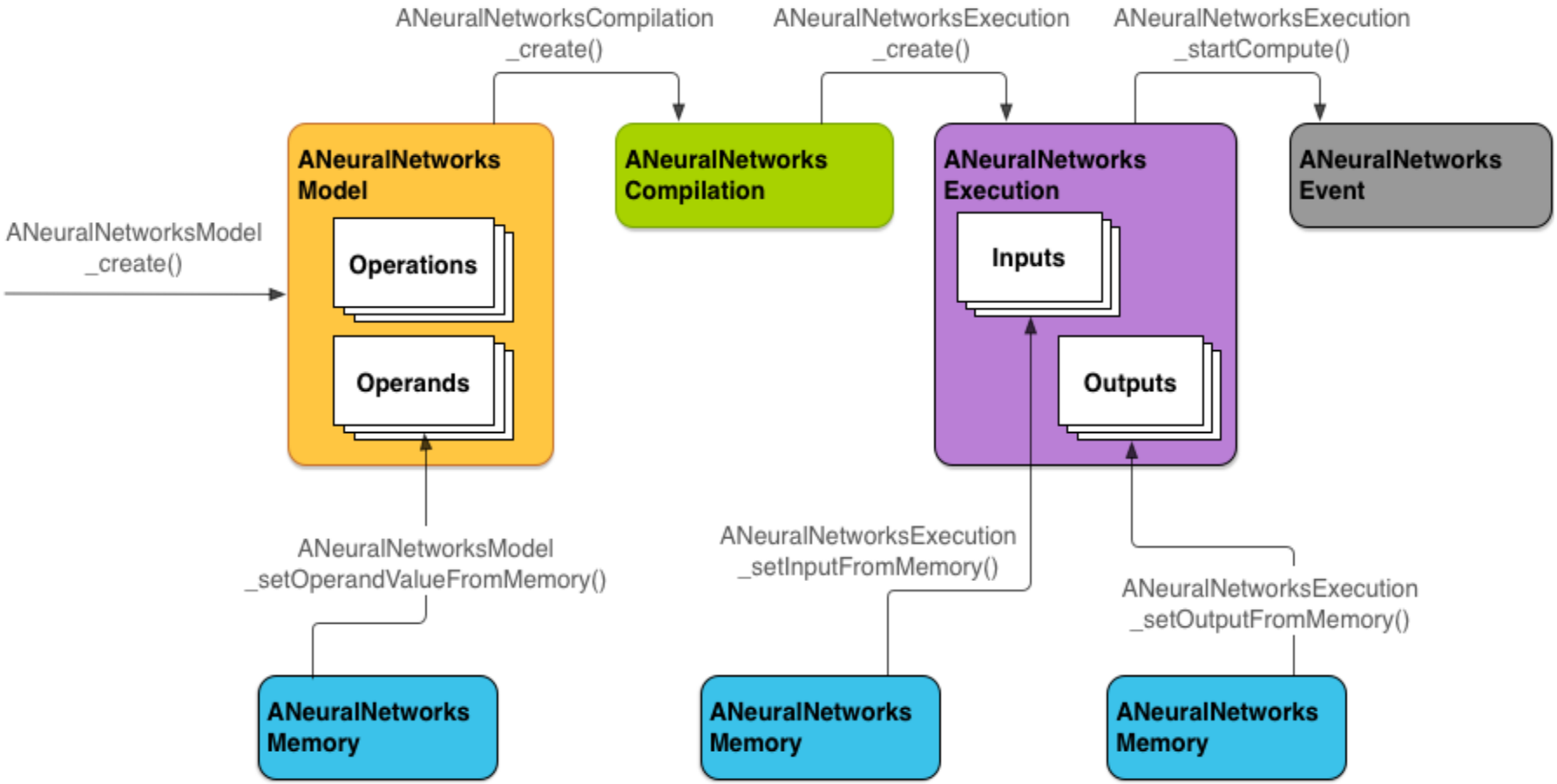
NNAPI

- On-device inferencing has many benefits:
  - Latency: You don't need to send a request over a network connection and wait for a response.
    - For example, this can be critical for video applications that process successive frames coming from a camera.
  - Availability: The application runs even when outside of network coverage.
  - Speed: New hardware that is specific to neural network processing provides significantly faster computation than a general-purpose CPU, alone.
  - Privacy: The data does not leave the Android device.
  - Cost: No server farm is needed when all the computations are performed on the Android device.

NNAPI

- There are also trade-offs that a developer should keep in mind:
  - System utilization: Evaluating neural networks involves a lot of computation, which could increase battery power usage.
    - You should consider monitoring the battery health if this is a concern for your app, especially for long-running computations.
  - Application size: Pay attention to the size of your models.
    - Models may take up multiple megabytes of space.
    - If bundling large models in your APK would unduly impact your users, you may want to consider downloading the models after app installation, using smaller models, or running your computations in the cloud.
    - NNAPI does not provide functionality for running models in the cloud.


Application
Machine learning framework/library
Android NN API
Android Neural Networks Runtime
CPU fallback
Hardware acceleration
Android NN HAL
Vendor NN DSP driver
Vendor NN driver
Vendor NN GPU driver
Digital Signal Processor
Specialized Processor
GPU


ANeuralNetworksCompilation
_create()
ANeuralNetworksExecution
_create()
ANeuralNetworksExecution
_startCompute()
ANeuralNetworksModel
_create()
ANeuralNetworks
Model
Operations
Operands
ANeuralNetworks
Compilation
ANeuralNetworks
Execution
Inputs
Outputs
ANeuralNetworks
Event
ANeuralNetworksModel
_setOperandValueFromMemory()
ANeuralNetworksExecution
_setInputFromMemory()
ANeuralNetworksExecution
_setOutputFromMemory()
ANeuralNetworks
Memory
ANeuralNetworks
Memory
ANeuralNetworks
Memory

ML kit

Firebase

ML kit

- ML Kit is a mobile SDK that brings Google's machine learning expertise to Android apps in a powerful yet easy-to-use package.
- Whether you're new or experienced in machine learning, you can implement the functionality you need in just a few lines of code.
- There's no need to have deep knowledge of neural networks or model optimization to get started.
  - On the other hand, if you are an experienced ML developer, ML Kit provides convenient APIs that help you use your custom TensorFlow Lite models in your mobile apps.

ML features

Feature	On-device	Cloud
Text recognition	X	X
Face detection	X	
Barcode scanning	X	
Image labeling	X	X
Object detection & tracking	X	
Landmark recognition		X
Language identification	X	
Translation	X	
Smart Reply	X	
AutoML model inference	X	
Custom model inference	X	

ML text recognition

- With ML Kit's text recognition APIs, you can recognize text in any Latin-based language (and more, with Cloud-based text recognition).
- Text recognition can automate tedious data entry for credit cards, receipts, and business cards, or help organize photos.
  - With the Cloud-based API, you can extract text from documents, which you can use to increase accessibility or translate documents.
  - Apps can even keep track of real-world objects, such as by reading the numbers on trains.

ML text recognition

Text
Please
Bounding Polygon
(355, 371) (471, 371) (471, 415) (355, 415)

Text
Walk on the Grass
Bounding Polygon
(388, 424) (711, 384) (718, 439) (395, 480)

Text
Metropolitan Toronto Parks
Bounding Polygon
(361, 532) (651, 489) (656, 521) (365, 563)

ML face detection

- With ML Kit's face detection API, you can detect faces in an image and identify key facial features.
- With face detection, you can get the information you need to perform tasks like embellishing selfies and portraits, or generating avatars from a user's photo.
  - Because ML Kit can perform face detection in real time, you can use it in applications like video chat or games that respond to the player's expressions.

ML face detection

Face 1 of 3		
Bounding polygon	(884.880004882812, 149.546676635742), (1030.77197265625, 149.546676635742), (1030.77197265625, 329.660278320312), (884.880004882812, 329.660278320312)	
Angles of rotation	Y: -14.054030418395996, Z: -55.007488250732422	
Tracking ID	2	
Facial landmarks	Left eye	(945.869323730469, 211.867126464844)
	Right eye	(971.579467773438, 247.257247924805)
	Bottom of mouth	(907.756591796875, 259.714477539062)
	... etc.	
Feature probabilities	Smiling	0.8897916674613952
	Left eye open	0.9863588893786072
	Right eye open	0.9925832338631153

ML barcode scan

- With ML Kit's barcode scanning API, you can read data encoded using most standard barcode formats.
- Barcodes are a convenient way to pass information from the real world to your app.
  - In particular, when using 2D formats such as QR code, you can encode structured data such as contact information or WiFi network credentials.
  - Because ML Kit can automatically recognize and parse this data, your app can respond intelligently when a user scans a barcode.

ML barcode scan

Result
Corners
(49,125), (172,125), (172,160), (49,160)
Raw value
2404105001722

ML image labeling

- With ML Kit's image labeling APIs, you can recognize entities in an image without having to provide any additional contextual metadata, using either an on-device API or a cloud-based API.
- Image labeling gives you insight into the content of images.
  - When you use the API, you get a list of the entities that were recognized: people, things, places, activities, and so on.
  - Each label found comes with a score that indicates the confidence the ML model has in its relevance.
  - With this information, you can perform tasks such as automatic metadata generation and content moderation.

ML image labeling

On-device	Cloud
Description	Description
Stadium	sport venue
Knowledge Graph entity ID	Knowledge Graph entity ID
/m/019cfy	/m/0bmgjqz
Confidence	Confidence
0.9205354	0.9860726
Description	Description
Sports	player
Knowledge Graph entity ID	Knowledge Graph entity ID
/m/06ntj	/m/02vzx9
Confidence	Confidence
0.7531109	0.9797604

ML landmark recognition

- With ML Kit's landmark recognition API, you can recognize well-known landmarks in an image.
- When you pass an image to this API, you get the landmarks that were recognized in it, along with each landmark's geographic coordinates and the region of the image the landmark was found.
- You can use this information to automatically generate image metadata, create individualized experiences for users based on the content they share, and more.

ML landmark recognition

Result	
Description	Brugge
Geographic Coordinates	51.207367, 3.226933
Knowledge Graph entity ID	/m/0drjd2
Bounding Polygon	(20, 342), (651, 342), (651, 798), (20, 798)
Confidence Score	0.77150935

ML custom model

- If you're an experienced ML developer and ML Kit's pre-built models don't meet your needs, you can use a custom TensorFlow Lite model with ML Kit.
- Host your TensorFlow Lite models using Firebase or package them with your app.
  - Then, use the ML Kit SDK to perform inference using the best-available version of your custom model.
  - If you host your model with Firebase, ML Kit automatically updates your users with the latest version.

Homework

- Create an application to read barcodes
  - Use the builtin ML capabilities
  - Store the history of the recognized barcodes
    - Date
    - Information
    - Type of the code
  - Save the information to file in JSON format
    - Use the GSON library

More on API

Next week