FIAP ON

TÓPICOS AVANÇADOS DE DESENVOLVIMENTO ANDROID

DISPOSITIVOS *VESTÍVEIS E WEAR OS*

RICARDO DA SILVA OGLIARI

5

LISTA DE FIGURAS

LISTA DE CÓDIGOS-FONTE

SUMÁRIO

5 DISPOSITIVOS VESTÍVEIS E WEAR OS

Wear OS é a especificação do Android para aparelhos vestíveis, os populares wearables. Dentro desse tipo de equipamento temos, de forma clara, um conhecimento e introdução maior nos relógios inteligentes, os smartwatches.

5.1 História

Os primeiros relógios inteligentes com o Wear OS surgiram em 2014. Naquele momento, o mercado nos apresentava três aparelhos: Moto 360, LG G Watch e Samsung Gear Live. Na Figura a seguir podemos ver um Gear Live 1.

Figura 5.1 – Samsung Gear Live 1
Fonte: Wikipedia Gear Live (2020)

Os primeiros aparelhos com Wear OS 1 não tiveram muito sucesso no mercado. Um bom exemplo é a mudança que a própria Samsung fez, adotando o sistema operacional Tizen no Gear Live 2. Porém a própria empresa voltou a adotar o Wear OS 2.0 na sequência.

O desenvolvimento para o Wear OS 1.0 também tinha algumas limitações que foram resolvidas com a versão mais nova. Por exemplo, agora podemos criar uma aplicação para relógio totalmente independente do smartphone, inclusive tendo sua própria versão na Play Store ou ainda fazendo requisições sobre o protocolo HTTP,

de forma totalmente autônoma. A API da biblioteca também amadureceu sensivelmente, fornecendo meios para trabalhar com as limitações que o hardware do relógio apresenta.

Atualmente, não é possível encontrar uma lista de quais aparelhos suportam a Wear OS 2.0, no site oficial de desenvolvedores Android. Porém encontramos uma lista extensa de fabricantes parceiros, na qual é possível encontrar fabricantes de eletrônicos (como Intel e Samsung), fabricantes renomados de relógios (como Cassio e Mont Blanc), operadoras de telefonia celular (como T-Mobile) e marca de luxo (como Louis Vuitton, Guess e Hugo Boss).

5.2 Ambiente de Desenvolvimento

O Android Studio fornece todas as ferramentas necessárias para o desenvolvimento Android, em todas as plataformas possíveis; e, para Wear OS, não foge à regra. Não há grande diferença em relação ao ambiente de desenvolvimento, apenas o necessário de conhecimento exigido para programação Android.

Uma diferença sutil, esteja na criação de um emulador para testes, o Android Virtual Device caso um smartwatch físico não esteja disponível. Ao entrar na janela de criação do dispositivo Android virtual, é recomendável criar duas versões, um para tela quadrada e outra para tela circular do relógio. Veremos adiante que existem detalhes importantes em relação a essa diferença de layout, e que devem ser consideradas.

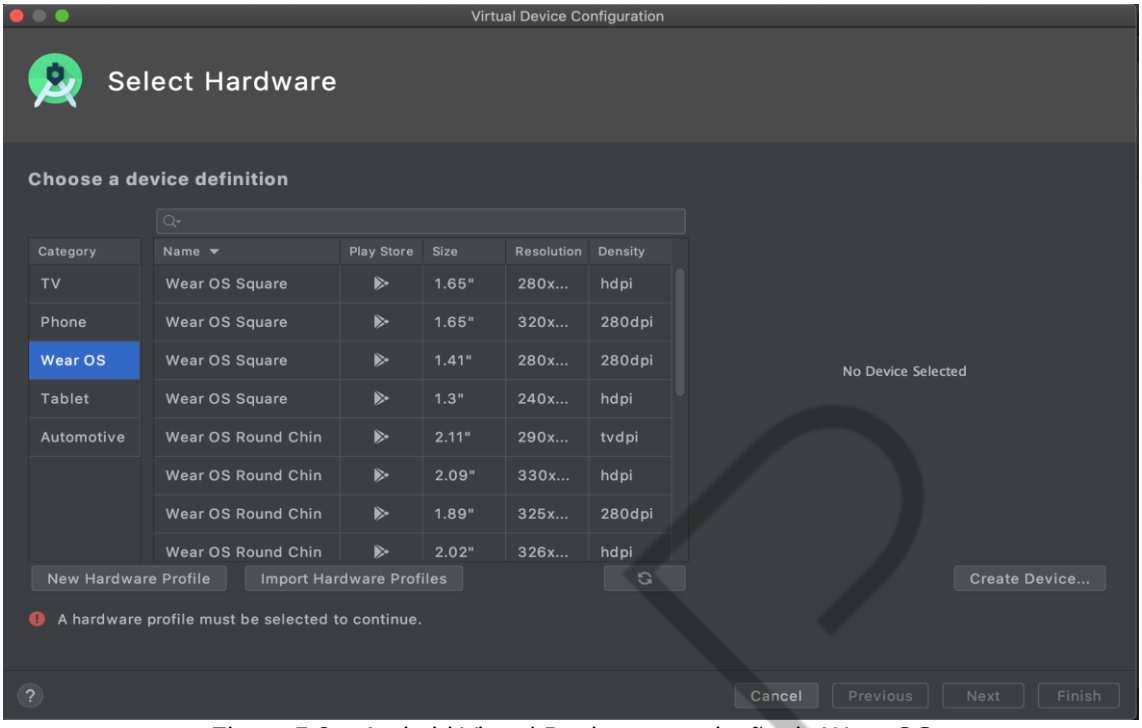

Figura 5.2 – Android Virtual Device para criação de Wear OS
Fonte: Elaborado pelo autor (2020)

5.3 Desenvolvimento

5.3.1 Aplicativo Proposto

Vamos criar uma aplicação reunindo alguns detalhes de implementação, como: ativação de timer, chamadas de ações, posicionamento por GPS e notificações.

Esses casos de uso podem ser explorados em outras aplicações, por exemplo: podemos usar a detecção de localização por GPS para avisar ao usuário que ele pode estar numa região com algum perigo (notificação), e que o uso de equipamentos de proteção seja obrigatório, necessitando sua confirmação (chamada de ação).

Mostraremos essas opções em uma lista, já condizentes com o layout do relógio (quadrado ou não). Veja, na figura a seguir, nossa meta ao final deste projeto.

Figura 5.3 – Aplicativo proposto em dois layouts de relógio diferentes
Fonte: Elaborado pelo autor (2020)

Nossos principais recursos são:

- Timer: opção de confirmação de ação com um timer. Muito comum, devido à usabilidade limitada de um relógio.
- Confirmação de sucesso, confirmação de falha e confirmação On Phone: a API do Wear OS já fornece três telas de confirmação padrão, de sucesso, de falha e de indicação para completar a ação no smartphone.
- Minha posição: vamos mostrar a posição geográfica (GPS) do relógio.
- Entrada de áudio: para requisitar uma entrada de informação através de áudio.
- Chamada HTTP: uma chamada GET, apenas para integrações com APIs externas.

5.3.2 Criando o projeto no Android Studio

O uso do Android Studio é análogo ao desenvolvimento de um aplicativo regular para smartphone. Porém existem algumas diferenças sutis. Logo na primeira tela do wizard de criação da aplicação, devemos ter cuidado para selecionar a opção “Wear OS”, sendo que a opção padrão é “Phone and Tablet”. Veja, na Figura “Wizard de

criação de aplicativo Wear OS. Escolha de Activity", a escolha do tipo de Activity é indiferente neste momento. Mesmo assim, se o leitor desejar seguir exatamente os passos do texto, escolha a opção "Blank Activity".

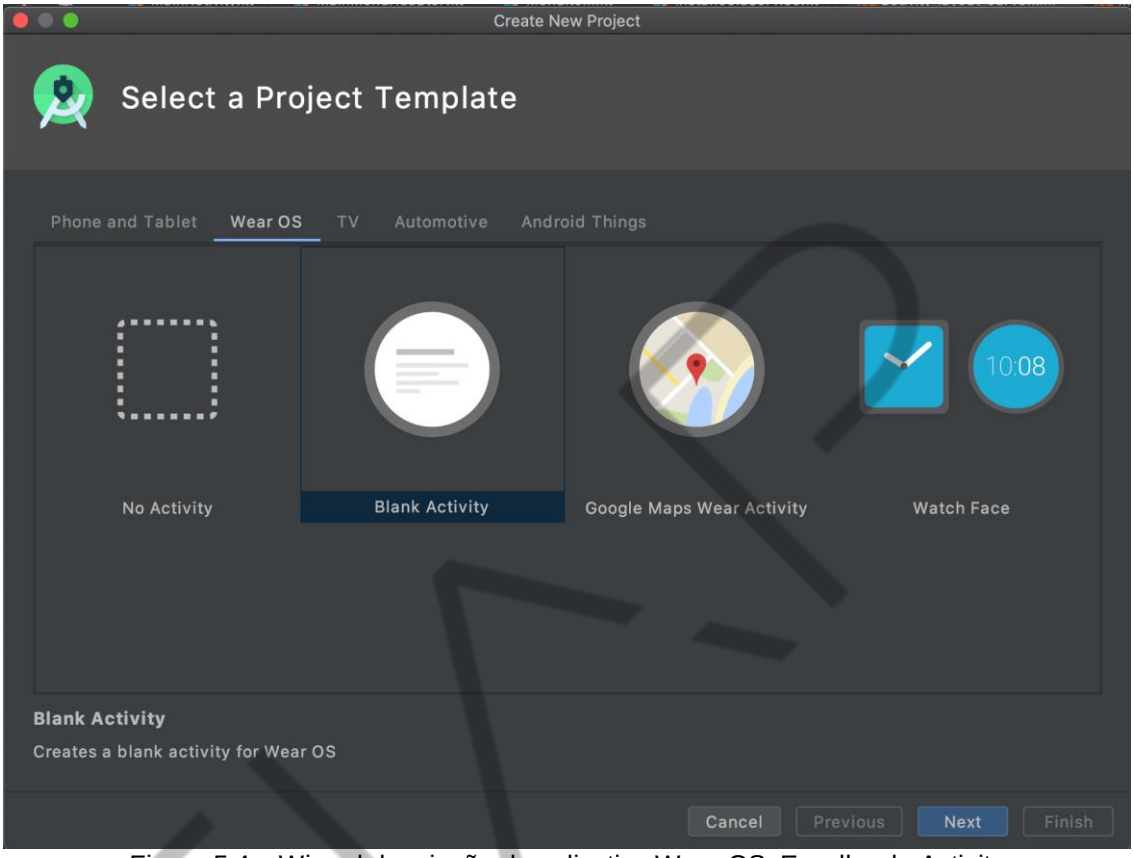

Figura 5.4 – Wizard de criação de aplicativo Wear OS. Escolha de Activity
Fonte: Elaborado pelo autor (2020)

A grande maioria dos conceitos comuns no desenvolvimento Android será utilizada aqui também. Isso traz um aproveitamento interessante para quem já desenvolve aplicações para smartphones, usando Android e sua pilha de API e ferramentas.

No próximo passo do wizard, temos um detalhe muito importante; além das opções já comuns, como: nome do projeto, nome do pacote, localização do projeto, linguagem e versão mínima do SDK, temos também a opção "Pair With Empty Phone App" (como mostra a figura abaixo). Se a opção for marcada, o aplicativo do Wear OS será pareado com um smartphone e veremos dois módulos sendo criados no projeto: um para relógio e outro para smartphone.

Nas versões mais recentes do Wear OS é possível criar uma aplicação para relógio independente do smartphone. Aliás, é essa a opção que desejamos, logo, mantenha o checkbox desmarcado e clique em "Finish".

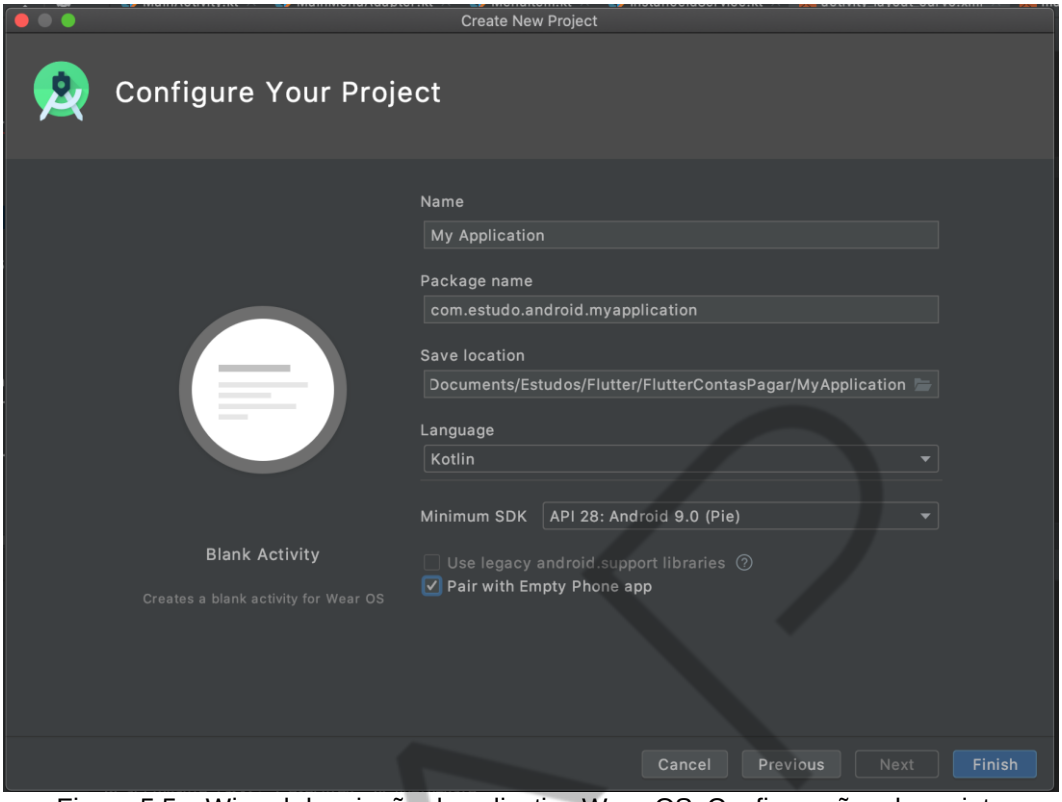

Figura 5.5 – Wizard de criação de aplicativo Wear OS. Configurações do projeto
Fonte: Elaborado pelo autor (2020)

5.3.3 Estudando o código padrão

A primeira novidade que vamos estudar está no *build.gradle,* específico do módulo app. Navegue até o final do arquivo, mais especificamente na parte de dependências (*dependencies*). Podemos ver no próximo código-fonte que os três pontos são as dependências mais comuns. Nosso foco estará no *androidx.wear:wear:1.0.0* e no *com.google.android.wearable:2.7.0.*

```
dependencies {
    ...


    implementation 'androidx.wear:wear:1.0.0'
    compileOnly 'com.google.android.wearable:wearable:2.7.0'
}
```

Código-fonte 5.1 – dependências no build.gradle
Fonte: Elaborado pelo autor (2019)

Com essas implementações, podemos utilizar Views específicas criadas pensando no ambiente proposto de relógios com diferenças de layouts. Vamos agora

para o *activity_main.xml* e estudar a primeira delas. Este arquivo de layout é mostrado no Código-fonte a seguir.

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.wear.widget.BoxInsetLayout
xmlns:android="http://schemas.android.com/apk/res/android"
   xmlns:app="http://schemas.android.com/apk/res-auto"
   xmlns:tools="http://schemas.android.com/tools"
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   android:background="@color/dark_grey"
   android:padding="@dimen/box_inset_layout_padding"
   tools:context=".MainActivity"
   tools:deviceIds="wear">

   <FrameLayout
       android:layout_width="match_parent"
       android:layout_height="match_parent"
       android:padding="@dimen/inner_frame_layout_padding"
       app:boxedEdges="all">

       <TextView
           android:id="@+id/text"
           android:layout_width="wrap_content"
           android:layout_height="wrap_content"
           android:text="@string/hello_world" />

   </FrameLayout>
</androidx.wear.widget.BoxInsetLayout>
```

Código-fonte 5.2 – activity_main.xml
Fonte: Elaborado pelo autor (2019)

Antes de começarmos a compreender o código, é importante levantar um questionamento: é possível usar todos os layouts comuns do Android no Wear OS, como o LinearLayout, o FrameLayout, o ConstraintsLayout e todos os demais? Sim, com certeza. Mas isso implica em algum complicador? Mesma resposta, sim! Isso acontece porque relógios podem ser curvos ou quadrados; sendo assim, usando um simples LinearLayout podemos ter uma visão errônea em telas arredondadas, como o Moto 360, por exemplo.

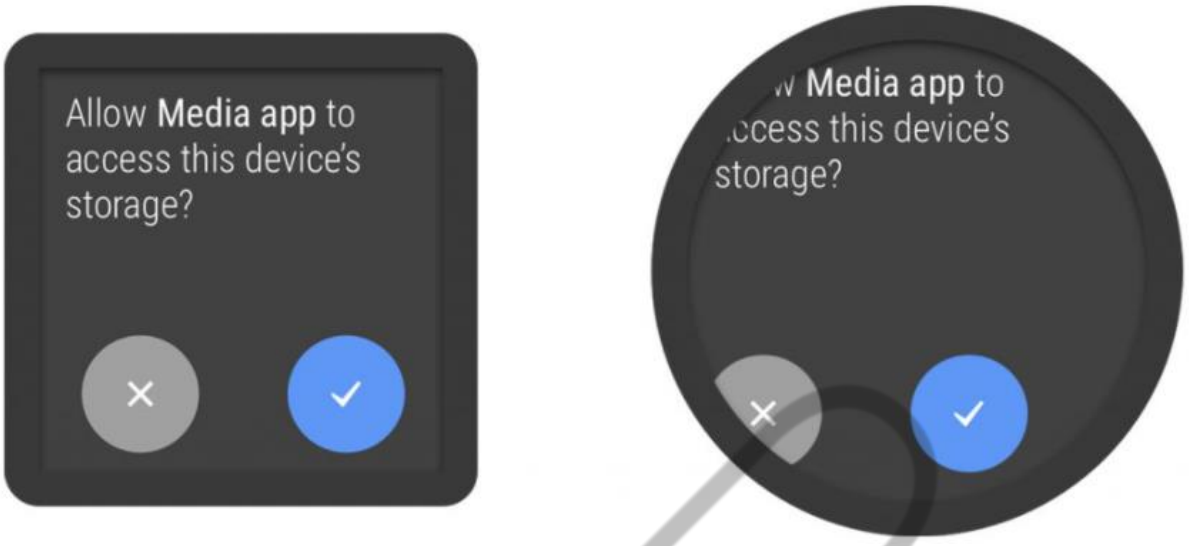

Figura 5.6 – Problemas com layouts comuns da Android API
Fonte: Android Layout Training (2020)

Agora, o *BoxInsetLayout* fará total sentido para aplicações deste tipo. Ele faz parte da API do Wear OS e já trabalha com essa diferença de layout, evitando problemas ou complicações que poderíamos ter com diferentes tipos de form factor de relógios. Claro que ainda podemos usar Constraints, Linear, Frame e afins, mas eles estão aninhados como filhos do BoxInsetLayout. Além disso, perceba que no *FrameLayout* temos uma propriedade *app:boxesEdges* com o valor *all.*

O valor all refere-se ao controle do BoxInsetLayout em todos os quatro lados do dispositivo. É possível o desenvolvedor usar somente em um dos lados, mudando o valor dessa propriedade para *left, right, top* ou *bottom.*

5.3.4 Usando o RecyclerView do Wear OS

O uso de *RecyclerView* em aplicativos para smartphone é muito comum, afinal, mostrar uma lista de opções é uma funcionalidade bastante utilizada na maioria das soluções. E no Wear OS, nós temos uma especialização desta classe, porém ela já trata as diferenças de layouts dos relógios. E o mais importante é que todo o conhecimento do *RecyclerView,* tradicional será utilizado aqui.

Neste tipo de layout temos uma listagem de itens que são, na sua menor parte, também uma porção de layout que será repetido de 0 até n vezes. E no Wear OS isso se mantém. Dessa forma, vamos criar um arquivo de layout que será, justamente, a representação desse layout que será utilizado em cada item da nossa futura lista.

Crie um layout, chamado *main_menu_item.xml.*

```
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
   android:id="@+id/menu_container"
   xmlns:app="http://schemas.android.com/apk/res-auto"
   xmlns:tools="http://schemas.android.com/tools"
   android:layout_width="match_parent"
   android:layout_height="wrap_content"
   android:orientation="horizontal"
   android:paddingBottom="5dp"
   android:gravity="center_vertical"
   android:paddingTop="5dp">

   <ImageView
       android:id="@+id/menu_icon"
       android:layout_width="wrap_content"
       android:layout_height="wrap_content"
       android:layout_marginRight="5dp"
       android:src="@drawable/ic_right"
       android:layout_alignParentLeft="true"
       android:layout_alignParentStart="true"
       android:layout_centerVertical="true"
       tools:ignore="HardcodedText" />

   <TextView
       android:id="@+id/menu_item"
       android:layout_width="wrap_content"
       android:layout_height="wrap_content"
       android:layout_centerVertical="true"
       android:text="Menu Item"
       android:textColor="#000000"
       android:textSize="14sp"
       tools:ignore="HardcodedText" />
</LinearLayout>
```

Código-fonte 5.3 – arquivo main_menu_item
Fonte: Elaborado pelo autor (2020)

Perceba que estamos usando as mesmas Views e ViewGroups já comuns no desenvolvimento Android, além de estarmos usando também as propriedades já tradicionais e extremamente conhecidas, como o layout_width e o ayout_height Somente um detalhe: o ic_right é um arquivo vetorial que foi baixado no site Material Icons (https://materialdesignicons.com/).

Agora vamos para o layout da nossa única tela. Veja, no Código-fonte a seguir, o código do *activity_main.xml*.

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.wear.widget.WearableRecyclerView
```

```
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/wearableRecyclerView"
    android:background="@color/white"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:padding="15dp"
    android:scrollbars="vertical" />
```

Código-fonte 5.4 – arquivo activity_main
Fonte: Elaborado pelo autor (2020)

O XML acima tem dois pontos que merecem nossa atenção. Para ter acesso aos benefícios do RecyclerView do Wear OS tome cuidado para usar a classe marcada em negrito, a *WearableRecyclerView.* Outro detalhe, no padding que configuramos como *15dp*, em telas arredondadas, ele não terá efeito. Isso acontece porque o controle do próprio layout já tem um espaçamento maior, justamente para garantir que nosso layout não quebrará nas diferenças entre telas quadradas e arredondadas.

No início, quando apresentamos o aplicativo a ser desenvolvido, notamos que nossa lista de ações é simples, contendo apenas um texto. Porém, já pensando em listas mais complexas e também para uma boa separação, vamos criar uma classe *MenuItem* com apenas uma propriedade, o texto que será exibido na tela. Veja, no Código-fonte “classe MenuItem”, como é o código da referida classe:

```
package com.estudo.android.exwearosfiap

data class MenuItem  (var text: String? = null)
```

Código-fonte 5.5 – classe MenuItem
Fonte: Elaborado pelo autor (2020)

Mais uma coincidência com o RecyclerView tradicional: a da necessidade de um adapter. Veja, no Código-fonte a seguir, o código-fonte da “classe MainMenuAdapter”, e que não temos nada específico da Wear OS.

```
package com.estudo.android.exwearosfiap

import android.content.Context
import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import android.widget.LinearLayout
import android.widget.TextView
import androidx.recyclerview.widget.RecyclerView
```

```
class MainMenuAdapter(
    context: Context,
    dataArgs: ArrayList<MenuItem>,
    callback: AdapterCallback?
) :
    RecyclerView.Adapter<MainMenuAdapter.RecyclerViewHolder>()
{
    private var dataSource = ArrayList<MenuItem>()

    interface AdapterCallback {
        fun onItemClicked(menuPosition: Int?)
    }

    private val callback: AdapterCallback?
    private val context: Context

    override fun onCreateViewHolder(
        parent: ViewGroup,
        viewType: Int
    ): RecyclerViewHolder {
        val view: View =
LayoutInflater.from(parent.context).inflate(R.layout.main_men
u_item, parent, false)
        return RecyclerViewHolder(view)
    }

    class RecyclerViewHolder(view: View) :
RecyclerView.ViewHolder(view) {
        var menuContainer: LinearLayout
        var menuItem: TextView

        init {
            menuContainer =
view.findViewById(R.id.menu_container)
            menuItem = view.findViewById(R.id.menu_item)
        }
    }

    override fun onBindViewHolder(
        holder: RecyclerViewHolder,
        position: Int
    ) {
        val data_provider = dataSource[position]
        holder.menuItem.setText(data_provider.text)
        holder.menuContainer.setOnClickListener {
callback?.onItemClicked(position) }
    }

    override fun getItemCount(): Int {
        return dataSource.size
    }
```

```
    init {
        this.context = context
        dataSource = dataArgs
        this.callback = callback
    }
}
```

Código-fonte 5.6 – classe MainMenuAdapter
Fonte: Elaborado pelo autor (2020)

Agora já temos tudo o que precisamos para o Recycler: o layout dos itens específicos e o adaptador. Podemos então finalizar as configurações no MainActivity. No Código-fonte a seguir, temos a nova implementação desta classe.

```
package com.estudo.android.exwearosfiap

import android.os.Bundle
import android.support.wearable.activity.WearableActivity
import androidx.wear.widget.WearableLinearLayoutManager
import kotlinx.android.synthetic.main.activity_main.*

class MainActivity : WearableActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        // Enables Always-on
        setAmbientEnabled()

        wearableRecyclerView.apply {
            layoutManager =
WearableLinearLayoutManager(this@MainActivity)

            val menuItems: ArrayList<MenuItem> = ArrayList()
            menuItems.add(MenuItem("Timer"))
            menuItems.add(MenuItem("Confirmação de Sucesso"))
            menuItems.add(MenuItem("Confirmação de Falha"))
            menuItems.add(MenuItem("Confirmação On Phone"))
            menuItems.add(MenuItem("Minha Posição"))
            menuItems.add(MenuItem("Chamada HTTP"))

            setAdapter(
                MainMenuAdapter(this@MainActivity, menuItems,
object : MainMenuAdapter.AdapterCallback {
                    override fun onItemClicked(menuPosition:
Int?) {

                    }
                }))
```

```
            }
        }
}
```

Código-fonte 5.7 – classe MainActivity
Fonte: Elaborado pelo autor (2020)

Novamente temos o padrão dos códigos anteriores. À primeira vista, temos um código extremamente semelhante ao desenvolvimento para Android Smartphone. Porém, perceba que a classe herda de *WearableActivity* e não de *AppCompatActivity*. No entanto, temos uma mudança ainda maior gerada justamente por essa herança diferenciada.

Em um smartwatch temos o modo ambiente que define uma Activity, que ficará visível para o usuário todo o tempo. Mas ele tem um modo de economia de energia, justamente o modo ambiente. É muito comum, nas aplicações, termos dois layouts diferentes para estes dois modos de uso diferentes. Em nosso exemplo, não vamos chegar a programar isso, mas ativamos o modo ambiente através do método *setAmbientEnabled.*

Por fim, veja que usamos uma classe *WearableLinearLayoutManager* e não a *LinearLayoutManager,* comum no desenvolvimento para smartphone. A resposta ao clique nos itens da lista ainda não foi programada. Esse será o próximo passo que codificaremos.

5.3.5 Definindo uma tela de Timer

Chegamos então no seguinte status: nosso aplicativo já mostra uma listagem completa e funcional, exceto pelo fato de não termos nenhuma ação no clique nas opções da lista. Veja na Figura “Aplicação neste momento. Funcional, mas sem resposta a eventos” como está a nossa aplicação

Figura 5.7 – Aplicação neste momento. Funcional, mas sem resposta a eventos
Fonte: Android Layout Training (2020)

Vamos começar nossa primeira tratativa pelo timer. Na Figura Timer que vamos construir, vemos qual é a nossa implementação alvo. Devido à natureza do relógio e à limitação de iterações com o usuário, uma confirmação de ação por timer é muito comum. A ideia é programar um tempo para que o usuário possa cancelar uma ação. Caso o tempo passe sem ação nenhuma, entende-se que o usuário quer confirmar aquela ação.

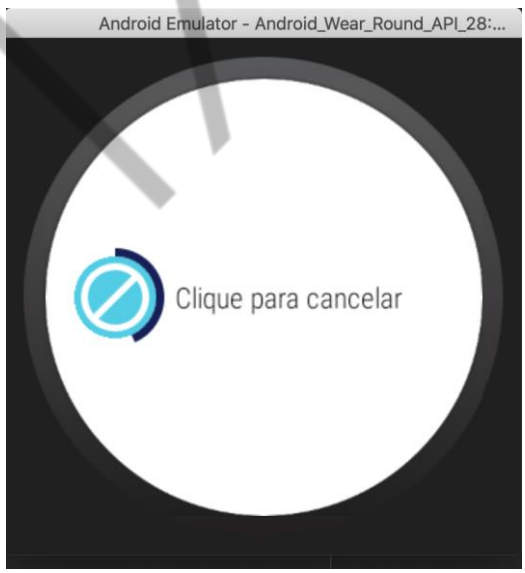

Figura 5.8 – Timer que iremos construir
Fonte: Android Layout Training (2020)

O primeiro passo será criar um novo arquivo de layout, que chamaremos de *activity_confirm_layout.* Veja o Código-fonte “activity_confirm_screen.xml” com a implementação desse layout.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
   xmlns:app="http://schemas.android.com/apk/res-auto"
   xmlns:tools="http://schemas.android.com/tools"
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   android:orientation="vertical"
   android:padding="10dp"
   android:gravity="center_vertical"
   android:background="@color/white"
   tools:context=".ConfirmScreenActivity">

   <LinearLayout
       android:gravity="center_vertical"
       android:orientation="horizontal"
       android:layout_width="match_parent"
       android:layout_height="wrap_content">

       <androidx.wear.widget.CircularProgressLayout
           android:id="@+id/circular_progress"
           android:layout_width="wrap_content"
           android:layout_height="wrap_content"
           android:padding="4dp"
           android:layout_marginRight="5dp"
           app:backgroundColor="@color/lightblue"
           app:colorSchemeColors="@color/darkblue"
           app:strokeWidth="4dp">
           <ImageView
               android:tint="@color/white"
               android:src="@drawable/ic_cancel"
               android:layout_width="40dp"
               android:layout_height="40dp" />
       </androidx.wear.widget.CircularProgressLayout>
       <TextView
           android:layout_width="wrap_content"
           android:layout_height="wrap_content"
           android:textColor="@color/black"
           android:text="Clique para cancelar"/>

   </LinearLayout>

</LinearLayout>
```

Código-fonte 5.8 – activity_confirm_screen.xml
Fonte: Elaborado pelo autor (2020)

Vamos diretamente aos detalhes. O *CircularProgressLayout* é o elemento que apresenta uma animação de contagem de tempo e, em seu interior, podemos mostrar qualquer elemento visual. No exemplo, estamos mostrando uma imagem vetorial,

também encontrada no site, já mencionado, do Material Icons. No próprio elemento circular temos três propriedades primordiais para seu funcionamento:

- backgroundColor: cor que define o fundo do elemento filho de *CircularProgressLayout.* Para que essa cor possa ser vista, colocamos transparência total no fundo da imagem ic_cancel, além de mudar o *foreground* da imagem para branca.
- colorSchemeColor: definição da cor que preencherá a linha que circunda o filho de *CircularProgressLayout.* No exemplo é o tom de azul mais forte.
- strokeWidth: largura da linha que circunda o filho de *CircularProgressLayout.*

Depois do layout criado, vamos à codificação de tela. Veja, no Código-fonte a seguir, a classe *ConfirmScreenActivity.*

```
<?xml version="1.0" encoding="utf-8"?>
package com.estudo.android.exwearosfiap

import android.os.Bundle
import android.support.wearable.activity.WearableActivity
import android.view.View
import androidx.wear.widget.CircularProgressLayout
import
kotlinx.android.synthetic.main.activity_confirm_screen.*

class ConfirmScreenActivity : WearableActivity(),
CircularProgressLayout.OnTimerFinishedListener,
   View.OnClickListener {

   override fun onCreate(savedInstanceState: Bundle?) {
       super.onCreate(savedInstanceState)
       setContentView(R.layout.activity_confirm_screen)

       setAmbientEnabled()

       circular_progress.apply {
           onTimerFinishedListener =
this@ConfirmScreenActivity
           setOnClickListener(this@ConfirmScreenActivity)

           totalTime = 2000
           startTimer()
       }
   }
```

```
    override fun onTimerFinished(layout:
CircularProgressLayout?) {}

    override fun onClick(view: View?) {
        circular_progress?.stopTimer()
    }
}
```

Código-fonte 5.9 – classe ConfirmScreenActrivity
Fonte: Elaborado pelo autor (2020)

Além do conhecimento que já adquirimos sobre o modo ambiente e a herança de *WearableActivity,* os códigos em negrito são essenciais para entender o *CircularProgressLayout.*

Dentro da função *apply* para o id do nosso *CircularProgressLayout* estamos, inicialmente, passando a função que preenche o *onTimerFinishedListener.* Esse listener será chamado quando o tempo de espera for concluído sem ação de cancelamento do usuário e, dessa forma, podemos efetuar definitivamente aquela ação que precisava ser confirmada.

Na sequência, estamos passando para o *setOnClickListener* a implementação da própria classe. Isso é possível porque a *ConfirmScreenActivity* implementa a interface *View.OnClickListener* e, consequentemente, também sobrescreve o método *onClick.* Esse método será chamado se o click para cancelar a ação for efetuado. Em no nosso exemplo apenas paramos o contador do timer.

Posteriormente, analisamos o *totalTime,* que indica qual o tempo máximo de espera ou qual o tempo limite que a animação vai levar para preencher os 360 graus que circundam o filho de *CircularProgressLayout.* Um lembrete: este tempo é em milissegundos (no nosso caso, são 2 segundos ao todo). Assim, iniciamos a contagem do timer com a chamada ao método *startTimer.*

E, para finalizar a parte de chamada ao timer, precisamos apenas de algumas alterações na MainActivity. Veja o Código-fonte “Chamando a tela ConfirmScreenActrivity”.

A mudança ocorreu apenas na resposta ao clique em uma opção do RecyclerView. Estamos verificando o índice da posição do menu, se ele for igual a 0 (zero), chamamos a função *timer.* Esta, por sua vez, apenas chama a tela *ConfirmScreenActivity* com a tão conhecida *Intent,* tão usada no desenvolvimento para todas as plataformas Android.

```
override fun onCreate(savedInstanceState: Bundle?) {
    ...

    wearableRecyclerView.apply {
        ...

        setAdapter(
            MainMenuAdapter(this@MainActivity, menuItems,
object : MainMenuAdapter.AdapterCallback {
                override fun onItemClicked(menuPosition: Int?)
{
                    when (menuPosition) {
                        0 -> timer()
                    }
                }
            }))
    }
}

fun timer(){
    val intent = Intent(this,
ConfirmScreenActivity::class.java);
    startActivity(intent)
}
```

Código-fonte 5.10 – Chamando a tela ConfirmScreenActrivity
Fonte: Elaborado pelo autor (2020)

5.3.6 Trabalhando com avisos

A API do Wear OS nos fornece uma tela padronizada para emitir alertas animados aos usuários. Nas Figuras a seguir, podemos ver os avisos de sucesso, falha e ação para ser completada no smartphone.

Figura 5.9 – Aviso de sucesso
Fonte: Elaborado pelo autor (2020)

Figura 5.10 – Aviso de falha
Fonte: Elaborado pelo autor (2020)

Figura 5.11 – Abrir no phone
Fonte: Elaborado pelo autor (2020)

Outra boa notícia: nós já temos uma *Activity* pronta para uso diretamente na API. Basta declará-la no manifesto da aplicação, colocando como filho da tag *application* as seguintes tags:

```
<activity
   android:name="androidx.wear.activity.ConfirmationActivity">
</activity>
```

Em seguida, trataremos o evento de clique nos menus correspondentes, chamando a ação adequada. Veja o Código-fonte “Uso da ConfirmationActivity”.

```
override fun onCreate(savedInstanceState: Bundle?) {

   when (menuPosition) {
       0 -> timer()
       1 -> successConfirmation()
       2 -> failureConfirmation()
       3 -> openOnPhoneConfirmation()
   }
}

fun showConfirmation(extra: String = "Mensagem Extra", type:
Int){
   val intent = Intent(this,
ConfirmationActivity::class.java).apply {
       putExtra(ConfirmationActivity.EXTRA_ANIMATION_TYPE,
type)
       putExtra(ConfirmationActivity.EXTRA_MESSAGE, extra)
   }
   startActivity(intent)
}

fun successConfirmation(extra: String = "Mensagem Extra"){
   showConfirmation(extra,
ConfirmationActivity.SUCCESS_ANIMATION)
}

fun failureConfirmation(extra: String = "Mensagem Extra"){
   showConfirmation(extra,
ConfirmationActivity.FAILURE_ANIMATION)
}

fun openOnPhoneConfirmation(extra: String = "Mensagem
Extra"){
   showConfirmation(extra,
ConfirmationActivity.OPEN_ON_PHONE_ANIMATION)
}
```

Código-fonte 5.11 – uso da ConfirmationActivity
Fonte: Elaborado pelo autor (2020)

O principal método aqui é o *showConfirmation*, e é neste ponto que estamos chamando a *ConfirmationActivity.* Apenas dois parâmetros extras são necessários: o *EXTRA_ANIMATION_TYPE,* que define seu tipo, dentro de três constantes possíveis, *SUCCESS, FAILURE ou OPEN_OH_PHONE* e o *EXTRA_MESSAGE* que é o texto que fica abaixo da animação.

5.4 Mostrando o endereço

No Wear OS 2.0 é possível que a aplicação, de maneira independente, consulte dados de geoposicionamento independentes do smartphone. Mesmo assim, a forma de programar se mantém extremamente parecida.

Inicialmente, precisamos declarar as permissões de usuário no manifesto da aplicação. Sendo assim, adicione as duas linhas seguintes no manifesto como filhos da tag raiz, *manifest.*

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

Em uma aplicação real, não é necessário declarar as duas opções porque o COARSE_LOCATION se baseia em método de posicionamento baseado em rede, como triangulação de antenas - estação rádio base - (da rede da operadora de telefonia móvel) e o FINE_LOCATION procura usar métodos finos de posicionamento, com grau de precisão e custo computacional maiores (a bateria também).

O próximo arquivo editado será o *build.gradle* do módulo *app.* Adicionaremos mais uma linha nas dependências do projeto. Veja na Figura a seguir a última linha de *implementation.*

```
dependencies {
   ...

   implementation 'com.google.android.gms:play-services-
location:17.0.0'
}
```

Código-fonte 5.12 – implementation do Google Location API
Fonte: Elaborado pelo autor (2020)

Pronto, agora estamos preparados para nos debruçar sobre a MainActivity e responder ao item de posicionamento no RecyclerView. Vamos ao Código-fonte “Implementation do Google Location API”.

```
override fun onCreate(savedInstanceState: Bundle?) {

   when (menuPosition) {
       0 -> timer()
       1 -> successConfirmation()
       2 -> failureConfirmation()
       3 -> openOnPhoneConfirmation()
       4 -> myPosition()
   }

}

fun myPosition(){
   if (hasGps()) {
       if (ActivityCompat.checkSelfPermission(
               this,
               Manifest.permission.ACCESS_FINE_LOCATION
           ) != PackageManager.PERMISSION_GRANTED
       ) {
           requestPermissions(

arrayOf(Manifest.permission.ACCESS_FINE_LOCATION),
               10
           )
       } else {
           val fusedLocationClient =
LocationServices.getFusedLocationProviderClient(this)

fusedLocationClient.lastLocation.addOnSuccessListener {
location: Location? ->
               location?.let {location ->
                   val address =
Geocoder(this).getFromLocation(location.latitude,
location.longitude, 1).get(0)
                   Toast.makeText(this,
address.getAddressLine(0), Toast.LENGTH_LONG).show()
               }
           }
       }
   }
}

private fun hasGps(): Boolean =
packageManager.hasSystemFeature(PackageManager.FEATURE_LOCATI
ON_GPS)
```

Código-fonte 5.13 – Implementation do Google Location API
Fonte: Elaborado pelo autor (2020)

O método *myPosition* começa fazendo uma chamada a outro método, o *hasGps.* O hasGps apenas usa o gerenciador de pacotes e seu método *hasSystemFeature* para checar se o hardware adjacente tem suporte à localização por GPS. Seu retorno é um valor booleano que dependerá da resposta do aparelho em questão.

Voltando ao *myPosition,* se o primeiro teste lógico é atendido, entramos em outra lógica booleana, conferindo se o usuário já nos deu permissão para acessar os dados de geoposicionamento. Caso a permissão ainda não exista, acionamos o *requestPermissions,* passando, por parâmetro, uma lista de Strings com todas as permissões desejadas.

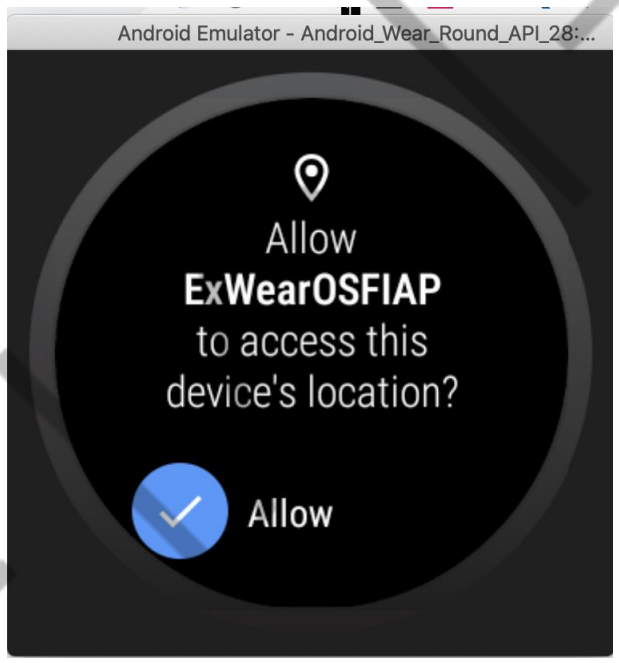

Figura 5.12 – Permissão para dados de geoposicionamento
Fonte: Elaborado pelo autor (2020)

Caso a aplicação já tenha permissão, criamos uma instância de *FusedLocationProviderClient.* Essa classe nos dá acesso a uma *Task* chamada *lastLocation.* Como estamos trabalhando com uma tarefa assíncrona, adicionamos o *addOnSuccessListener,* dessa forma, interceptamos o retorno de sucesso e recuperamos a instância de *Location* passada por parâmetro.

Porém, com a localização, só temos acesso aos dados de latitude e longitude, mas não a um dado visualmente aceitável para nosso usuário, como rua, cidade, bairro e assim por diante. Para resolver esse problema, usamos a classe *Geocoder* com seu método *getFromLocation.* Os parâmetros desses métodos são justamente

latitude e longitude e, o retorno, uma listagem de instância de *Address,* as quais fornecem todos os dados associados a um endereço.

Dessa forma, só anexamos esse resultado a uma notificação *toast.* Sim, exatamente o mesmo *Toast* que também é extensamente usado pelos desenvolvedores Android.

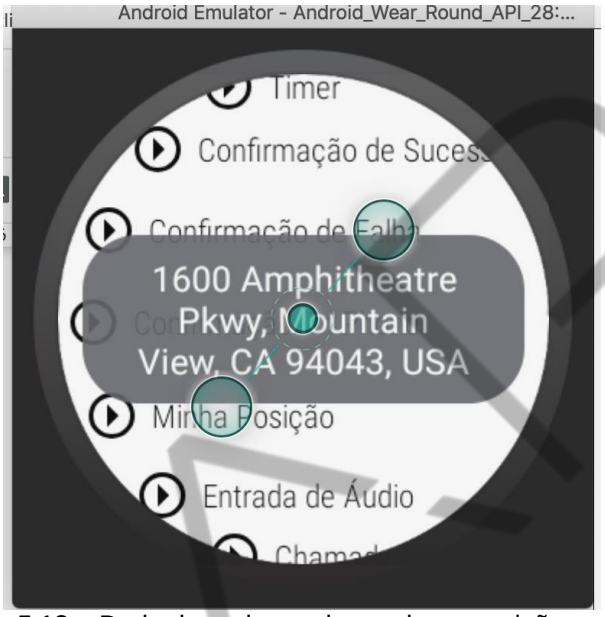

Figura 5.13 – Dado de endereço baseado na posição corrente
Fonte: Elaborado pelo autor (2020)

5.5 Requisições HTTP

Neste passo, vamos fazer uma chamada simples a um serviço HTTP prototipado (mock), apenas para mostrar a viabilidade dessa funcionalidade presente nas aplicações Wear OS.

Para criar seus dados prototipados será necessário criar uma conta,
de forma gratuita, no site https://app.fakejson.com/.
Ao entrar no site, a página apresenta um menu lateral com uma opção token.
Basta pegar o valor do token e usar como parte da URL
https://app.fakejson.com/q/<seu_token>.
Por exemplo, usamos neste projeto a seguinte URL:
https://app.fakejson.com/q/Lpg0g0V6?token=9s0m5BF14JecTED1KL7rrQ

O primeiro passo será adicionar mais uma implementação de dependência no *bluid.gradle* do módulo app. Siga como fizemos algumas vezes durante este texto, apenas adicionando a linha abaixo no objeto *dependencies:*

```
implementation 'com.android.volley:volley:1.1.1'
```

Também será necessário adicionar três linhas de permissão de usuário no manifesto da aplicação.

```
<uses-permission android:name="android.permission.INTERNET" />
<uses-permission
android:name="android.permission.CHANGE_NETWORK_STATE" />
<uses-permission
android:name="android.permission.WRITE_SETTINGS"
        tools:ignore="ProtectedPermissions" />
```

Na *MainActivity,* adicione duas variáveis: a *connectivityManager* e a *request.* Ambas podem ser vistas no Código-fonte “Novas variáveis para a MainActivity”. A primeira serve como um gerenciador de conectividade do dispositivo, pois precisamos dele para requisitar acesso à rede. O *NetworkRequest* define detalhes de tipo de transporte e capacidade da rede a serem utilizados.

```
private var connectivityManager: ConnectivityManager? = null

val request: NetworkRequest = NetworkRequest.Builder().run {
   addTransportType(NetworkCapabilities.TRANSPORT_WIFI)
   addTransportType(NetworkCapabilities.TRANSPORT_CELLULAR)

addCapability(NetworkCapabilities.NET_CAPABILITY_NOT_METERED)
   addCapability(NetworkCapabilities.NET_CAPABILITY_INTERNET)
   build()
}
```

Código-fonte 5.14 – novas variáveis para a MainActivity
Fonte: Elaborado pelo autor (2020)

Ainda na MainActivity, precisamos de algumas alterações também no *onCreate,* como podemos ver no Código-fonte a seguir. A primeira mudança foi a instanciação do *connectivityManager,* através de uma chamada ao *getSystemService,* que retorna um serviço relacionado especificamente com o sistema operacional. E, também, adicionamos a chamada ao método *httpRequest,* quando o item de menu coma posição 5 for acionado.

```
override fun onCreate(savedInstanceState: Bundle?) {
   super.onCreate(savedInstanceState)
   setContentView(R.layout.activity_layout_curvo)

   connectivityManager =
getSystemService(Context.CONNECTIVITY_SERVICE) as
ConnectivityManager

   wearableRecyclerView.apply {
       ...
       setAdapter(
           MainMenuAdapter(this@MainActivity, menuItems,
object : MainMenuAdapter.AdapterCallback {
           override fun onItemClicked(menuPosition: Int?) {
               when (menuPosition) {
                   ...
                   5 -> httpRequest()
               }
           }
       }))
   }
}
```

Código-fonte 5.15 – instância de connectivityManager
Fonte: Elaborado pelo autor (2020)

O Código-fonte “Conjunto de método para requisição HTTP”, nos mostra exatamente o método *httpRequest* e suas dependências. Se o *connectivityManager* não estiver nulo, o método *requestNetwork* é acionado. Lembre-se: criamos o *request* como variável global da classe, fizemos isso poucas páginas atrás. O *networkCallback* é mostrado na mesma listagem.

É com o *networkCallback* que a maior parte do trabalho é feito. Este retorno tem a sobrescrita de dois métodos. Podemos ter uma rede disponível (chamada ao método *onAvailable*) ou uma rede indisponível (chamada ao método *onUnavailable*).

Se a rede estiver disponível, criamos uma constante URL que apenas reflita o destino da chamada HTTP (faremos na sequência). No nosso caso, usamos um serviço web chamado *fakejson* para mockar retornos de API.

Na sequência, criamos uma instância de *JsonRequest* e passamos os seguintes parâmetros: método HTTP, URL de destino, JSON que será enviado na requisição (por isso o valor nulo é uma requisição muito simples), resposta no caso de sucesso e resposta no caso de erro. No caso de sucesso, apenas mandamos para o método que mostra a tela de confirmação de sucesso e, por parâmetro, mandamos a string com a propriedade *user_name* do json recebido por retorno.

Por fim, este mesmo *jsonRequest* é colocado em uma fila de requisições do *Volley,* e será gerenciado por esta mesma biblioteca.

```
private fun httpRequest() {
   connectivityManager?.requestNetwork(request,
networkCallback)
}

private fun getQueue(): RequestQueue{
   return Volley.newRequestQueue(this)
}

val networkCallback = object :
ConnectivityManager.NetworkCallback() {
   override fun onAvailable(network: Network) {
       val url = "https://app.fakejson.com/q/my_token"

       val jsonObjectRequest = JsonObjectRequest(
           Request.Method.GET,
           url,
           null,
           Response.Listener { response ->
               successConfirmation(extra =
response.getString("user_name"))
           },
           Response.ErrorListener { error ->
               failureConfirmation(extra = "Problema na
requisição")
           }
       )

       getQueue().add(jsonObjectRequest)
   }

   override fun onUnavailable() {
       super.onUnavailable()
   }
}
```

Código-fonte 5.16 – Conjunto de método para requisição HTTP
Fonte: Elaborado pelo autor (2020)

O resultado final, em caso de sucesso, pode ser visto na Figura “Retorno de sucesso na chamada HTTP”.

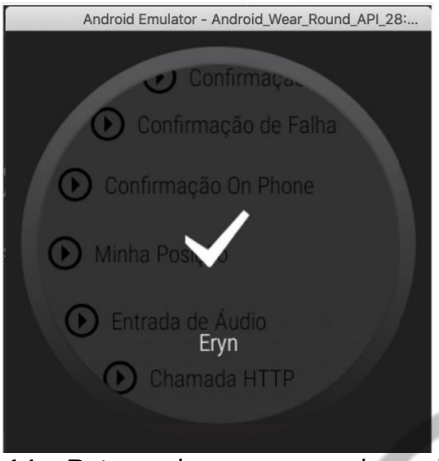

Figura 5.14 – Retorno de sucesso na chamada HTTP
Fonte: Elaborado pelo autor (2020)

5.6 Firebase e Push Notifications

Assim como nos smartphones, também é possível enviar push notifications diretamente para os dispositivos vestíveis. Isso é bastante útil, pois já estamos vivendo em um momento em que os smartwatches não dependem exclusivamente dos smartphones, recebendo de forma independente suas próprias notificações.

O primeiro passo para a integração com o *Firebase Cloud Messaging* é acessar o menu "*tools" > "Firebase"* no Android Studio. Nesse momento, a IDE nos apresenta um assistente no lado direito, com acesso fácil para configuração de diversos serviços presentes no Firebase (veja na figura seguinte).

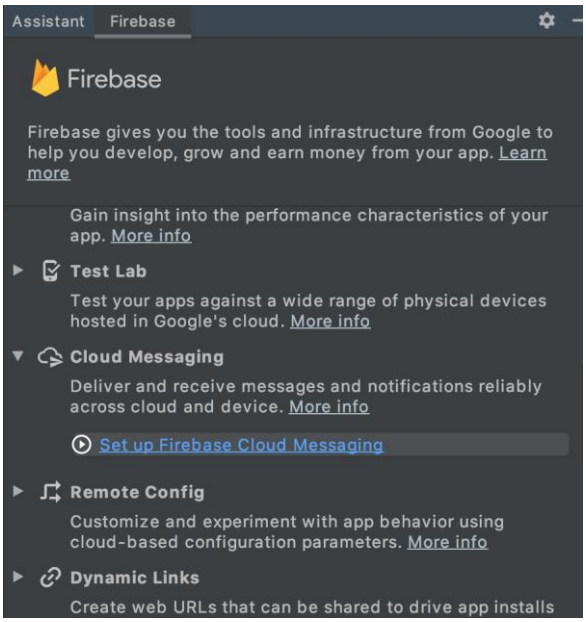

Figura 5.15 – Firebase Assistant do Android Studio
Fonte: Elaborado pelo autor (2020)

Navegue até a opção *Cloud Messaging.* Um texto explicativo é exibido, seguido da opção *Set Up Firebase Cloud Messaging.* Clique nessa opção e então veremos um passo a passo do que é necessário fazer para a correta configuração. Veja na a seguir.

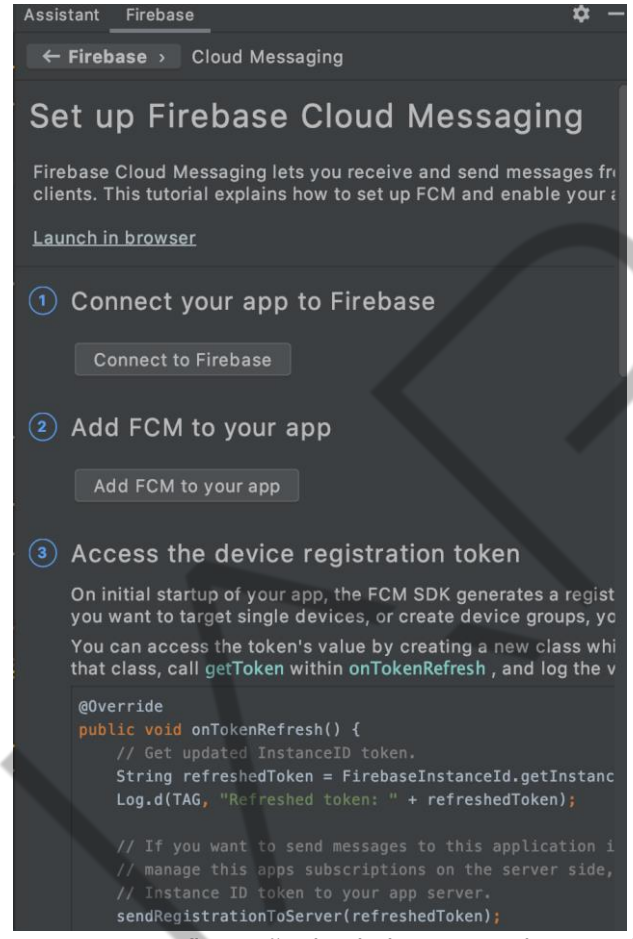

Figura 5.16 – Configuração do Firebase CLoud Messaging
Fonte: Elaborado pelo autor (2020)

O primeiro passo é clicar na opção *Connect to Firebase.* Ao fazer isso, você será encaminhado para a página de console do Firebase. Entre no seu console da cloud Firebase (https://firebase.google.com) e crie um projeto, "*Adicionar Projeto".*

Novamente teremos um wizard em nosso processo. Inicialmente, será necessário informar o nome do projeto, lembrando que estamos criando um projeto no console do Firebase, na nuvem e nos servidores do Google. Ainda não temos nenhuma ligação entre o projeto Wear OS e o Firebase em nossa máquina.

O nome é de livre escolha. Não existe nenhuma regra ou obrigatoriedade de informar o mesmo nome do projeto que nomeamos o aplicativo, sendo desenvolvido em nossa máquina. Porém é muito comum usarmos o mesmo nome para ficar fácil de

entender, futuramente, que ambos fazem parte do mesmo sistema computacional como um todo.

O segundo passo do wizard permite a utilização do Google Analytics no projeto que está sendo criado. Não vamos utilizar nada relacionado a isso, sendo assim, desmarque a opção *Ativar o google analytics neste projeto,* como mostrado a seguir.

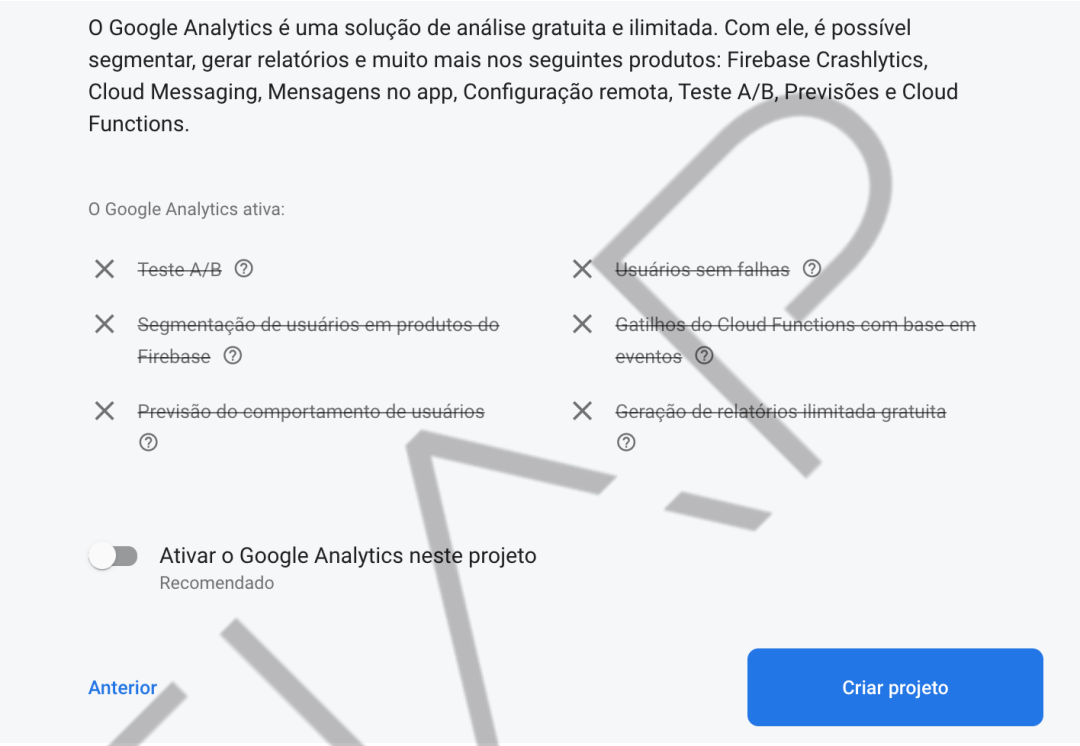

Figura 5.17 – Google Analytics desabilitado
Fonte: Elaborado pelo autor (2020)

Nesse mesmo wizard, clique no botão *Criar Projeto.* Se o procedimento for realizado com sucesso, você deve visualizar uma mensagem "*Seu novo projeto está pronto".* E, logo abaixo, temos a opção *Continuar* novamente.

Quando entramos no projeto, logo percebemos que alguma ação está sendo feita. No final, teremos uma mensagem de sucesso semelhante à Mensagem de app criado com sucesso:

Figura 5.18 – Mensagem de app criado com sucesso
Fonte: Elaborado pelo autor (2020)

E na mensagem de sucesso temos a opção *Conectar.* Ao selecioná-la, faremos a conexão entre o projeto Wear OS que estamos desenvolvendo em nossa máquina e o projeto recém-criado no console do Firebase, presente na nuvem e nos diversos servidores Google.

Depois que a conexão for efetuada com sucesso, o Firebase será ainda mais claro, mostrando uma outra página com o aviso: “Your Android Studio project is connected to your Firebase Android app. You can now use Firebase in your project! Go back to Android Studio to start using one of the Firebase SDKs”.

Nesse momento, podemos voltar ao Android Studio e continuar com o assistente do Firebase. Logo você perceberá que a opção *Connect your app to Firebase* removeu o botão de ação e mostrou um texto, em tom de verde, com a mensagem *Connected.*

Agora nos resta ir para o passo 2. Ele está intitulado no assistente como *Add FCM to your app* e o botão logo abaixo tem o rótulo igual que, ao ser clicado, um diálogo é apresentado com as alterações que são necessárias nos arquivos do *build.gradle.* Podemos copiar as informações e fazermos manualmente ou, de maneira mais simples, apenas clicar na opção *Accept Changes* e tudo é feito de forma automática pela IDE.

O Android Studio e seu assistente também nos fornecem uma ajuda sobre a codificação. Veja que no passo 3, nomeado como *Access the device registration token,* ele nos pede para criar uma classe já definindo sua herança e as sobrescritas necessárias.

Esse token não será usado pela nossa aplicação neste momento. Ele é usado quando a notificação push a ser enviada é direcionada a apenas um dispositivo ou a um grupo deles. Claro que isso é muito comum no desenvolvimento no mundo real, porém, como nosso objetivo aqui é mostrar a possibilidade de uso do FCM (Firebase Cloud Messaging) no Wear OS, vamos trabalhar da forma mais simples.

Vamos passar para o passo 4 do assistente, nomeado com *Handle messages.* Seguindo as orientações, clique com o botão direito do mouse no pacote principal da aplicação e navegue pelo menu *New* > *Service* > *Service.* Nomeie a classe como *HandleMessages,* desmarque os check boxes e escolha *Kotlin* como linguagem fonte:

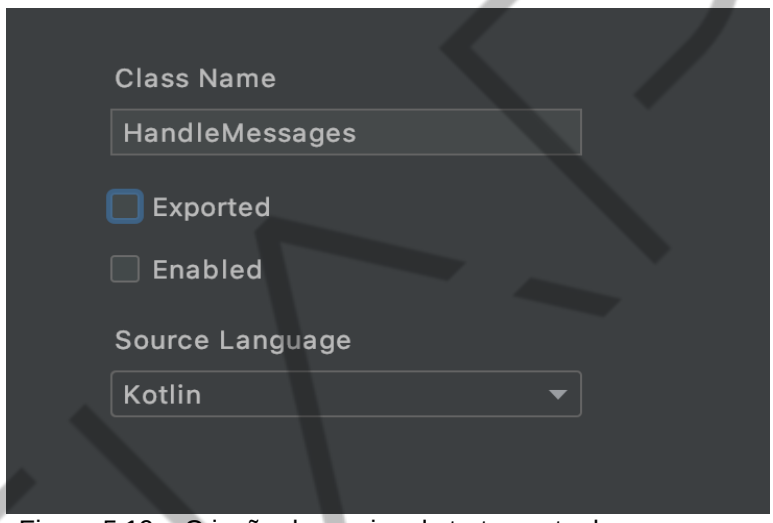

Figura 5.19 – Criação do serviço de tratamento de mensagens
Fonte: Elaborado pelo autor (2020)

Agora vá para o arquivo de manifesto da aplicação e procure pelo serviço recém-criado, será a única tag *service* presente. Precisamos passar um filtro de intenção para interceptarmos o evento de mensagem e, também, podemos remover as propriedades de *enabled* e *exported.* Veja, no Código-fonte a seguir, como ele fica:

```
<service
   android:name=".HandleMessages">
   <intent-filter>
       <action
android:name="com.google.firebase.MESSAGING_EVENT"/>
   </intent-filter>
</service>
```

Código-fonte 5.17 – Alterações no serviço declarado no manifesto
Fonte: Elaborado pelo autor (2020)

Para finalizarmos a parte de códigos, vamos às alterações na classe *HandleMessages,* que já estão visíveis no Código-fonte “Alterações no serviço declarado no manifesto”. O primeiro ponto de muita atenção é a herança da classe,

que não deve ser filha de *Service,* mas sim de *FirebaseMessagingService.* A sobrescrita do *onMessageReceived* é exatamente o local em que a mensagem push recebida será entregue.

Com o *RemoteMessage* em mãos, estamos apenas apresentando o conteúdo da propriedade *place* em uma notificação *toast.* O *Looper* que cerca essa ação é necessário como uma determinação da API para podermos lançar o toast em um serviço.

```
import android.os.Looper
import android.widget.Toast
import com.google.firebase.messaging.FirebaseMessagingService
import com.google.firebase.messaging.RemoteMessage

class HandleMessages : FirebaseMessagingService() {

   override fun onMessageReceived(remoteMessage:
RemoteMessage) {
       if (remoteMessage.data.size > 0) {
           Looper.prepare()

           Toast.makeText(this, remoteMessage.data["place"],
Toast.LENGTH_LONG).show()

           Looper.loop()
       }
   }
}
```

Código-fonte 5.18 – Alterações no serviço declarado no manifesto
Fonte: Elaborado pelo autor (2020)

Agora uma pergunta: como vou fazer o teste do push notification? Será necessário implementar toda a parte do servidor para atuar como *sender* da mensagem? A resposta é não. Na própria página de console do Firebase podemos fazer os primeiros testes de envio de mensagens.

Volte para o console do Firebase (https://console.firebase.google.com) e acesse o projeto que criamos anteriormente. Na dashboard da página veja o menu à esquerda, em que temos a listagem de todos os serviços oferecidos pelo Firebase. Deslize a página até encontrar o menu *Amplair* e clique na opção *Cloud Messaging.*

Logo no topo da página carregada temos um pequeno texto explicativo, seguido por um botão com o texto *Send yout first message.* Clique nele.

Nos passos de envio da notificação, podemos começar informando um texto *teste* no *Título* e *Texto da Notificação.* Na sequência, temos o passo de segmentação. Deixe o primeiro combo marcado com *app* e, do seu lado direito, clique no combo, escolhendo a aplicação que conectamos a este projeto Firebase, como mostra a seguir. Na sequência, defina a programação para *Agora.*

Figura 5.20 – Configuração da segmentação da notificação
Fonte: Elaborado pelo autor (2020)

Somente na última etapa, no passo 4, nomeado com *Outras Opções,* faremos uma mudança um pouco maior. Nos *dados personalizados,* adicione uma chave *place,* seguido do valor que preferir. No meu exemplo (Figura “Configuração de envio de dados personalizados”) coloquei: *J.A Arquitetura e Urbanismo*

Para finalizar, basta clicar no botão *Revisar* e, em seguida, clicar no botão *Publicar.*

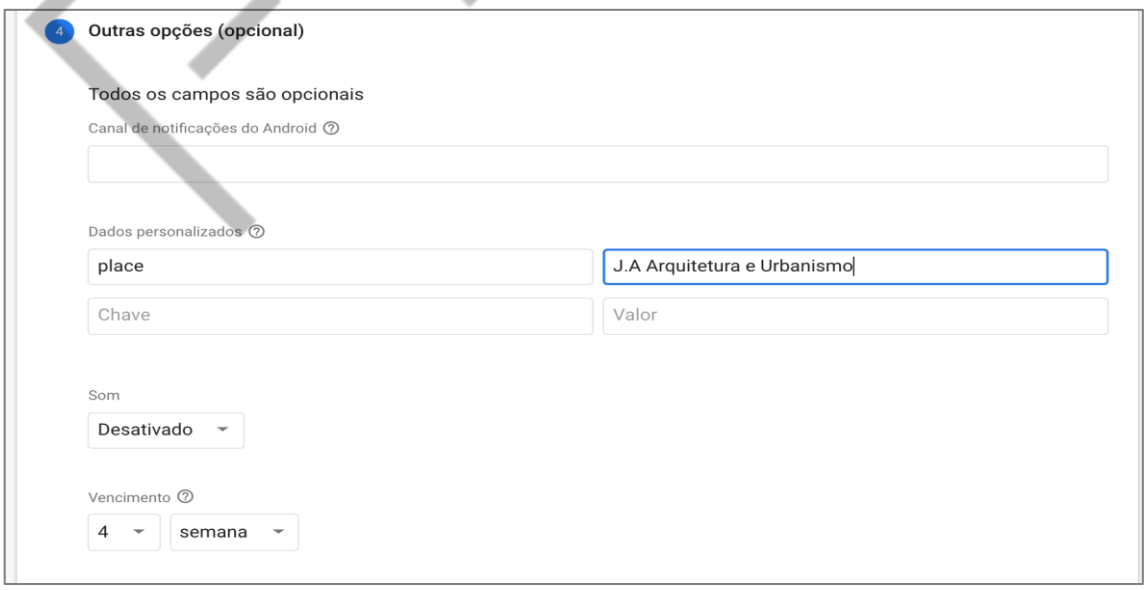

Figura 5.21 – Configuração de envio de dados personalizados
Fonte: Elaborado pelo autor (2020)

CONCLUSÃO

Neste capítulo, foi possível conhecer os principais aspectos sobre o desenvolvimento com Wear OS e aplicativos para smartwatches.

Desenvolvedores Android podem utilizar grande parte do conhecimento em outras plataformas, dado a convergência do ecossistema Android. Todos os conceitos básicos estão presentes, como telas, permissões, intenções, geração de pacotes para as lojas etc.

Apesar de suas especificidades, principalmente no que diz respeito à interface gráfica e à forma de utilizar o aparelho, a curva de aprendizado não é acentuada. Sendo assim, espero que você tenha tido uma boa base do que é possível criar para os relógios inteligentes com Android embarcado.

Você também pode acessar o projeto completo utilizado neste capítulo, no link do GitHub: https://github.com/FIAPON/IntroducaoWearOS.

REFERÊNCIAS

ANDROID DEVELOPERS. [s.d.]. Disponível em: <https://developer.android.com/training/wearables/ui/layouts>. Acesso em: 17 set. 2020.

MATERIAL DESIGN ICONS. [s.d.]. Disponível em: https://materialdesignicons.com/. Acesso em: 17 set. 2020.

STAT COUNTER. **Mobile Operating System Market Share Worldwide**. [s.d.]. Disponível em: <https://gs.statcounter.com/os-market-share/mobile/worldwide>. Acesso em: 17 set. 2020.