Część 6: Menedżer pracy

Użyj interfejsu WorkManager API systemu Android Jetpack, aby zaplanować niezbędne prace w tle, takie jak tworzenie kopii zapasowych danych lub pobieranie nowej zawartości, które działają nawet po zamknięciu aplikacji lub ponownym uruchomieniu urządzenia.

Spis treści

Zaplanuj zadania z WorkManager

Dowiedz się, kiedy i jak korzystać z WorkManager, interfejsu API obsługującego pracę w tle, która musi działać niezależnie od tego, czy proces aplikacji jest nadal uruchomiony.

Praca w tle z WorkManager

1. Wstęp

W systemie Android istnieje wiele opcji odroczenia pracy w tle. To laboratorium kodowania obejmuje WorkManager , kompatybilną wstecz, elastyczną i prostą bibliotekę do odroczenia pracy w tle. WorkManager jest zalecanym harmonogramem zadań na Androida do odroczenia pracy, z gwarancją wykonania.

Co to jest WorkManager

WorkManager jest częścią Android Jetpack i komponentem architektury do pracy w tle, która wymaga połączenia oportunistycznego i gwarantowanego wykonania. Oportunistyczne wykonanie oznacza, że WorkManager wykona Twoją pracę w tle tak szybko, jak to możliwe. Gwarantowane wykonanie oznacza, że WorkManager zadba o logikę rozpoczęcia pracy w różnych sytuacjach, nawet jeśli opuścisz swoją aplikację.

WorkManager to niezwykle elastyczna biblioteka, która ma wiele dodatkowych korzyści. Obejmują one:

- Obsługa zarówno asynchronicznych zadań jednorazowych, jak i okresowych
- Obsługa ograniczeń, takich jak warunki sieciowe, przestrzeń dyskowa i stan ładowania
- Łączenie złożonych żądań pracy, w tym równoległe prowadzenie prac
- Dane wyjściowe z jednego żądania pracy używane jako dane wejściowe dla następnego
- Obsługa zgodności poziomu API z powrotem do poziomu API 14 (patrz uwaga)
- Praca z usługami Google Play lub bez
- Postępowanie zgodnie z najlepszymi praktykami dotyczącymi zdrowia systemu
- Obsługa LiveData w celu łatwego wyświetlania stanu żądania pracy w interfejsie użytkownika

Notatka:

WorkManager znajduje się na szczycie kilku interfejsów API, takich jak JobSchedulerі AlarmManager. WorkManager wybiera odpowiednie API do użycia na podstawie warunków, takich jak poziom API urządzenia użytkownika. Aby dowiedzieć się więcej, zapoznaj się z dokumentacją WorkManager .

Kiedy używać WorkManagera

Biblioteka WorkManager to dobry wybór w przypadku zadań, które są przydatne do wykonania, nawet jeśli użytkownik odchodzi od konkretnego ekranu lub aplikacji.

Kilka przykładów zadań, które są dobrym wykorzystaniem WorkManagera:

- Przesyłanie dzienników
- Stosowanie filtrów do obrazów i zapisywanie obrazu
- Okresowa synchronizacja danych lokalnych z siecią

WorkManager oferuje gwarantowaną realizację i nie wszystkie zadania tego wymagają. W związku z tym nie jest to uniwersalny sposób na uruchamianie każdego zadania poza głównym wątkiem. Aby uzyskać więcej informacji o tym, kiedy używać programu WorkManager, zapoznaj się z Przewodnikiem po przetwarzaniu w tle .

Co zbudujesz

W dzisiejszych czasach smartfony są prawie za dobre w robieniu zdjęć. Dawno minęły czasy, gdy fotograf mógł zrobić niezawodne, niewyraźne zdjęcie czegoś tajemniczego.

W tym ćwiczeniu z kodowania będziesz pracować nad Blur-O-Matic, aplikacją, która rozmywa zdjęcia i zapisuje wyniki w pliku. Czy to był potwór z Loch Ness czy łódź podwodna evelopera ? Dzięki Blur-O-Matic nikt nigdy się nie dowie.

Czego się nauczysz

- Dodanie WorkManagera do projektu
- Planowanie prostego zadania
- Parametry wejściowe i wyjściowe
- Praca łańcuchowa
- Wyjątkowa praca
- Wyświetlanie statusu pracy w interfejsie użytkownika
- Anulowanie pracy

- Ograniczenia pracy

Co będziesz potrzebował

- Najnowsza stabilna wersja Android Studio
- Powinieneś także znać LiveDatai ViewModel. Jeśli jesteś nowicjuszem w tych klasach, zapoznaj się z ćwiczeniami Codelab obsługującymi cykl życia systemu Android (w szczególności dla ViewModel i LiveData) lub Room with a View Codelab (wprowadzenie do komponentów architektury).

2. Przygotowanie

Krok 1 - Pobierz kod

Kliknij poniższy link, aby pobrać cały kod do tego ćwiczenia z programowania:

file_downloadPobierz kod startowy

Lub jeśli wolisz, możesz sklonować laboratorium kodowania WorkManager z GitHub:

```
$ git clone -b start_kotlin https://github.com/googlecodelabs/android-workmanager
```

Notatka:

Ten przykładowy kod używa powiązania widoku do interakcji z widokami zamiast findViewById.

Aby dowiedzieć się więcej, zapoznaj się z dokumentacją View Binding .

Krok 2 — Uruchom aplikację

Uruchom aplikację. Powinieneś zobaczyć następujący ekran:

Na ekranie powinny znajdować się przyciski radiowe, za pomocą których możesz wybrać stopień rozmycia obrazu. Naciśnięcie przycisku **Go** w końcu rozmyje i zapisze obraz.

Na razie aplikacja nie stosuje żadnego rozmycia.

Kod startowy zawiera:

- `WorkerUtils`**:** Ta klasa zawiera kod do rzeczywistego rozmycia obrazu oraz kilka wygodnych metod, których użyjesz później do wyświetlenia `Notifications`, zapisania mapy bitowej do pliku i spowolnienia aplikacji.
- *`BlurActivity`:* * Aktywność, która pokazuje obraz i zawiera przyciski radiowe do wyboru poziomu rozmycia.
- *`BlurViewModel`:* * Ten model widoku przechowuje wszystkie dane potrzebne do wyświetlenia `BlurActivity`. Będzie to również klasa, od której rozpoczniesz pracę w tle za pomocą WorkManagera.
- `Constants`**:** Klasa statyczna z pewnymi stałymi, których będziesz używać podczas ćwiczeń z programowania.
- `res/activity_blur.xml`: pliki układu dla **.**`BlurActivity`

***** To jedyne pliki, w których będziesz pisać kod.

3. Dodaj WorkManager do swojej aplikacji

`WorkManager`wymaga zależności Gradle poniżej. **Zostały one już zawarte** w plikach kompilacji:

app/build.gradle

```
dependencies {
   // WorkManager dependency
   implementation "androidx.work:work-runtime-ktx:$versions.work"
}
```

Powinieneś pobrać najnowszą stabilną wersję `work-runtime-ktx`stąd i umieścić poprawną wersję. W tej chwili najnowsza wersja to:

build.gradle

```
versions.work = "2.7.1"
```

Jeśli aktualizujesz swoją wersję do nowszej, upewnij się, że wykonałeś **Synchronizuj teraz** , aby zsynchronizować swój projekt ze zmienionymi plikami Gradle.

4. Złóż swoje pierwsze zlecenie pracy

W tym kroku zrobisz zdjęcie w `res/drawable`folderze o nazwie `android_cupcake.png`i uruchomisz na nim kilka funkcji w tle. Te funkcje spowodują rozmycie obrazu i zapisanie go w pliku tymczasowym.

Podstawy WorkManagera

Jest kilka klas WorkManagera, o których musisz wiedzieć:

- `Worker`: Tutaj umieszczasz w tle kod pracy, którą chcesz wykonać. Rozszerzysz tę klasę i nadpiszesz `doWork()`metodę.
- `WorkRequest`**:** To jest prośba o wykonanie jakiejś pracy. Przekażesz swoje `Worker`jako część tworzenia swojego `WorkRequest`. Podczas tworzenia `WorkRequest`możesz również określić takie rzeczy, jak `Constraints`kiedy `Worker`powinien działać.
- `WorkManager`**:** Te zajęcia faktycznie planują Twoje zajęcia `WorkRequest`i sprawiają, że są uruchamiane. Planuje `WorkRequest`je w sposób, który rozkłada obciążenie zasobów systemowych, przy jednoczesnym przestrzeganiu określonych przez użytkownika ograniczeń.

W twoim przypadku zdefiniujesz nowy `BlurWorker`, który będzie zawierał kod do rozmycia obrazu. Po kliknięciu przycisku **Przejdź**`WorkRequest` , a jest tworzone, a następnie umieszczane w kolejce przez `WorkManager`.

Krok 1 - Stwórz BlurWorker

W pakiecie `workers`stwórz nową klasę Kotlin o nazwie `BlurWorker`.

Krok 2 - Dodaj konstruktora

Dodaj zależność do `Worker`dla `BlurWorker`klasy:

```
class BlurWorker(ctx: Context, params: WorkerParameters) : Worker(ctx, params) {
}
```

Krok 3 - Zastąp i zaimplementuj doWork()

Twój `Worker`obraz zamazuje obraz babeczki.

Aby lepiej zobaczyć, kiedy praca jest wykonywana, użyjesz WorkerUtil's `makeStatusNotification()`. Ta metoda pozwoli Ci łatwo wyświetlić baner powiadomień u góry ekranu.

Zastąp `doWork()`metodę, a następnie zaimplementuj następujące. Możesz odwołać się do wypełnionego kodu na końcu sekcji:

1. Uzyskaj `Context`dzwoniąc do `applicationContext`nieruchomości. Przypisz go do nowego `val`nazwanego `appContext`. Będzie to potrzebne do różnych manipulacji mapami bitowymi, które zamierzasz zrobić.
2. Wyświetl powiadomienie o stanie za pomocą funkcji, `makeStatusNotification`aby powiadomić użytkownika o rozmyciu obrazu.
3. Utwórz `Bitmap`z obrazka babeczki:

```
val picture = BitmapFactory.decodeResource(
        appContext.resources,
        R.drawable.android_cupcake)
```

4. Uzyskaj niewyraźną wersję mapy bitowej, wywołując `blurBitmap`metodę z `WorkerUtils`.

5. Zapisz tę bitmapę do pliku tymczasowego, wywołując `writeBitmapToFile`metodę z `WorkerUtils`. Pamiętaj, aby zapisać zwrócony identyfikator URI w zmiennej lokalnej.
6. Wykonaj powiadomienie wyświetlające identyfikator URI, wywołując `makeStatusNotification`metodę z `WorkerUtils`.
7. Powrót `Result.success()`.
8. Umieść kod z kroków 3-6 w instrukcji try/catch. Złap ogólny `Throwable`.
9. W instrukcji catch wydrukuj komunikat o błędzie za pomocą instrukcji Log: `Log.e(TAG, "Error applying blur")`.
10. W instrukcji catch , a następnie wróć `Result.failure()`.

Poniżej znajduje się wypełniony kod dla tego kroku.

BlurWorker. **kt**

```
package com.example.background.workers

import android.content.Context
import android.graphics.BitmapFactory
import android.util.Log
import androidx.work.Worker
import androidx.work.WorkerParameters
import com.example.background.R

private const val TAG = "BlurWorker"
class BlurWorker(ctx: Context, params: WorkerParameters) : Worker(ctx, params) {

    override fun doWork(): Result {
        val appContext = applicationContext

        makeStatusNotification("Blurring image", appContext)

        return try {
            val picture = BitmapFactory.decodeResource(
                    appContext.resources,
                    R.drawable.android_cupcake)

            val output = blurBitmap(picture, appContext)

            // Write bitmap to a temp file
            val outputUri = writeBitmapToFile(appContext, output)

            makeStatusNotification("Output is $outputUri", appContext)

            Result.success()
        } catch (throwable: Throwable) {
            Log.e(TAG, "Error applying blur")
            Result.failure()
```

```
        }
    }
}
```

Krok 4 — Pobierz WorkManager w ViewModel

Utwórz zmienną klasy dla `WorkManager`instancji w swoim `ViewModel`:

BlurViewModel.kt

```
private val workManager = WorkManager.getInstance(application)
```

Krok 5 - Dodaj WorkRequest do kolejki WorkManager

W porządku, czas zrobić `WorkRequest`i powiedzieć WorkManagerowi, żeby go uruchomił. Istnieją dwa rodzaje `WorkRequest`s:

- `OneTimeWorkRequest`**:** A `WorkRequest`, które zostanie wykonane tylko raz.
- `PeriodicWorkRequest`**:** A `WorkRequest`, które będzie się powtarzać w cyklu.

Chcemy, aby obraz był zamazany tylko raz, gdy klikniemy przycisk **Przejdź .** Metoda `applyBlur`jest wywoływana po kliknięciu przycisku **Go** , więc utwórz `OneTimeWorkRequest`stamtąd `BlurWorker`. Następnie, używając swojej `WorkManager`instancji, umieść w kolejce swoje`WorkRequest`.

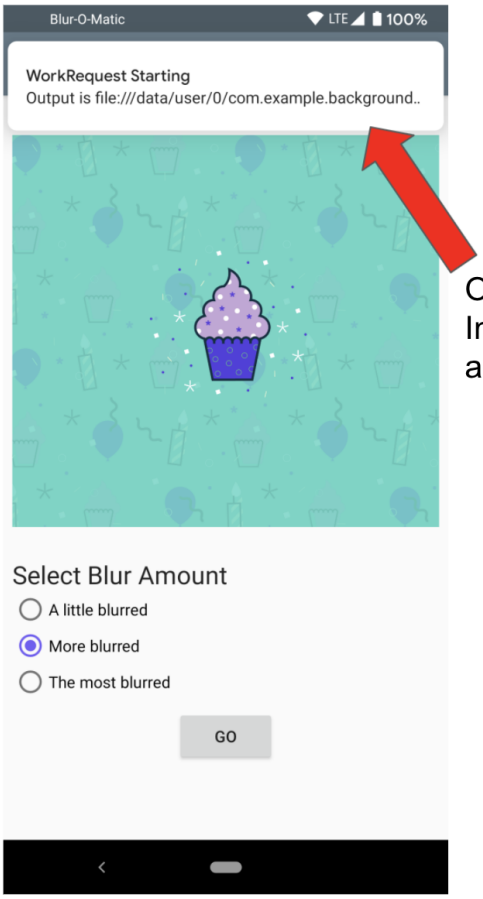
Dodaj następujący wiersz kodu do `BlurViewModel's` `applyBlur()`metody:

BlurViewModel.kt

```
internal fun applyBlur(blurLevel: Int) {
    workManager.enqueue(OneTimeWorkRequest.from(BlurWorker::class.java))
}
```

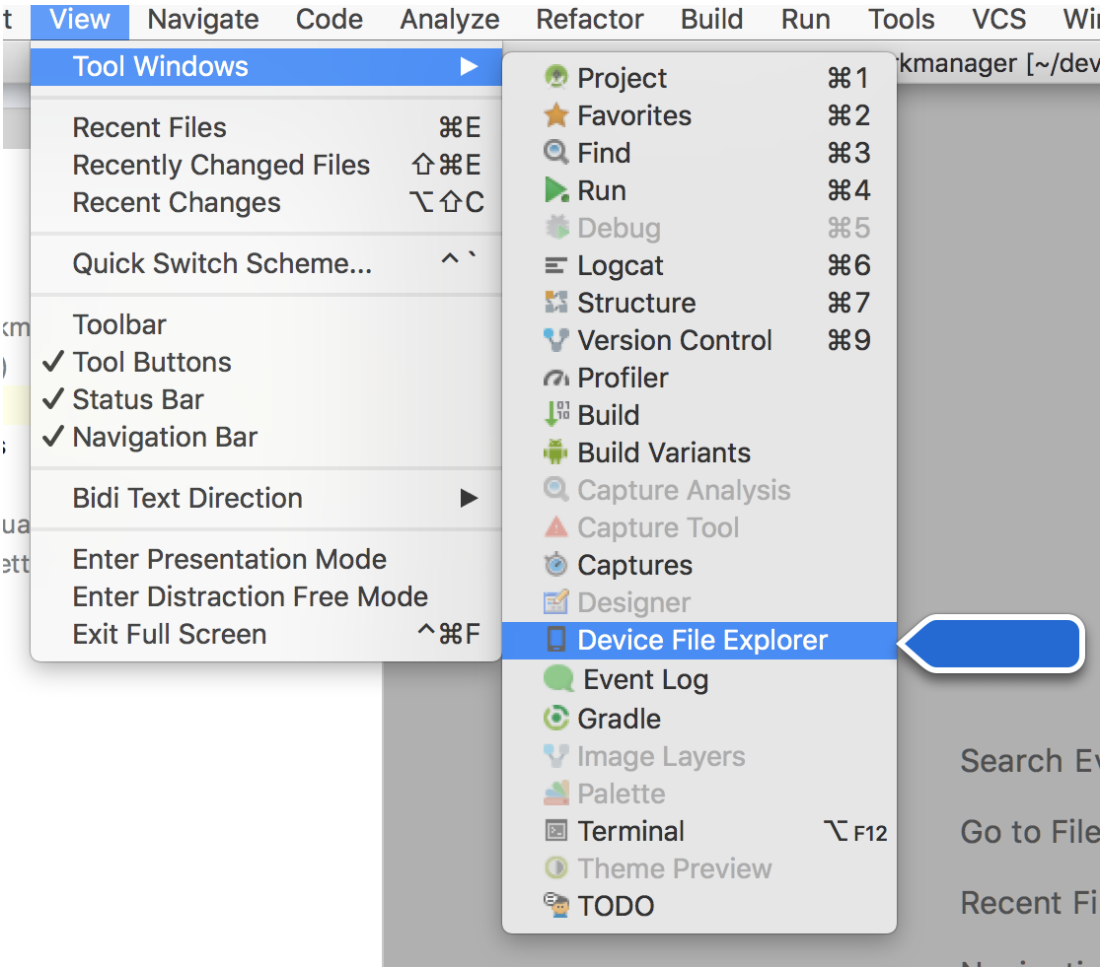
Krok 6 - Uruchom swój kod!

Uruchom swój kod. Powinien się skompilować i powinieneś zobaczyć powiadomienie po naciśnięciu przycisku **Go** . Pamiętaj, że aby zobaczyć bardziej rozmyte wyniki, wybierz opcję „Bardziej rozmyte” lub „Najbardziej rozmyte”.

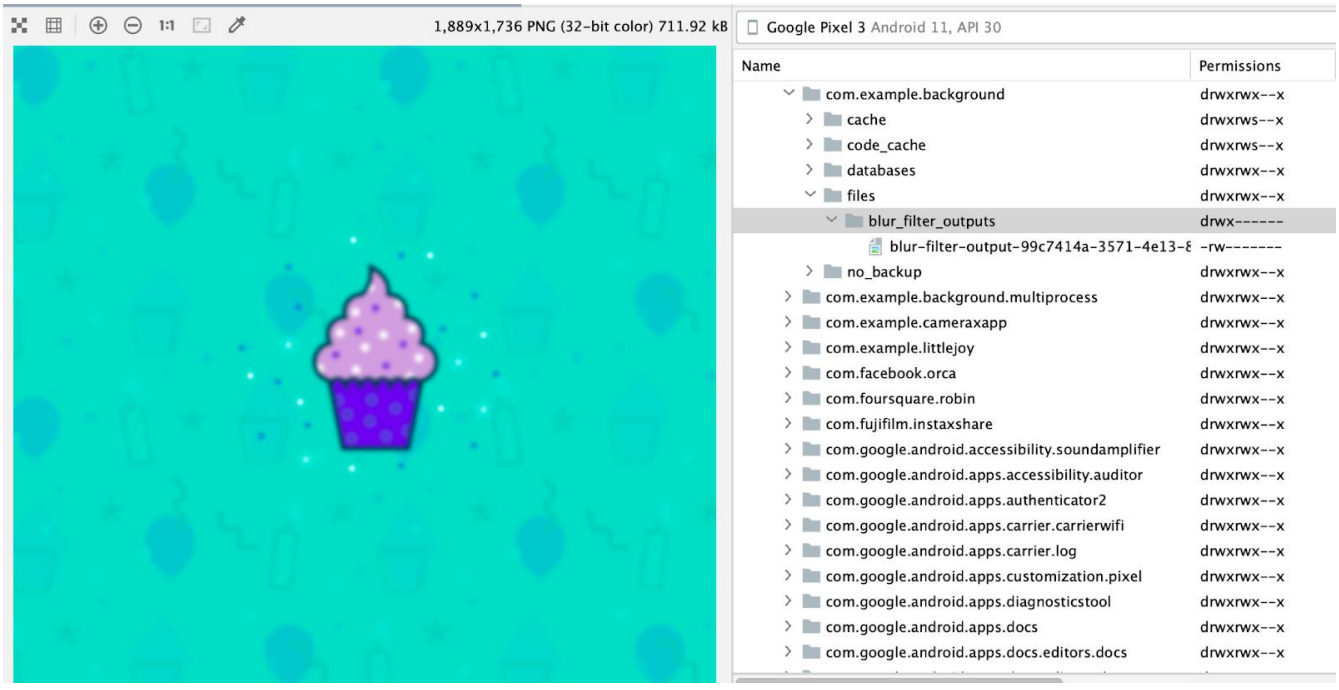


Output URI to look for blurred image
In Device File Explorer,
as described in the next step.

Aby potwierdzić, że obraz został pomyślnie zamazany, możesz otworzyć **Eksplorator plików urządzenia** w Android Studio:



Następnie przejdź do **danych > dane > com.example.background > pliki > blur_filter_outputs > <URI>** i potwierdź, że babeczka rzeczywiście została zamazana:



5. Dodaj wejście i wyjście

Rozmycie zasobu obrazu w katalogu zasobów jest dobre i dobre, ale aby Blur-O-Matic naprawdę była rewolucyjną aplikacją do edycji obrazów, jaką ma być, powinieneś pozwolić użytkownikowi rozmazać obraz, który widzi na ekranie, a następnie móc aby pokazać im rozmazany wynik.

Aby to zrobić, dostarczymy identyfikator URI obrazu babeczki wyświetlanego jako **dane wejściowe** do naszego `WorkRequest`wyświetlanego obrazu, a następnie użyjemy danych **wyjściowych** naszego WorkRequest, aby wyświetlić końcowy rozmyty obraz.

Krok 1 — Utwórz obiekt wprowadzania danych

Wejście i wyjście jest przekazywane przez `Data`obiekty. `Data`obiekty to lekkie kontenery dla par klucz/wartość. Są przeznaczone do przechowywania **niewielkiej** ilości danych, które mogą przechodzić do iz `WorkRequest`s.

Przekażesz identyfikator URI obrazu użytkownika do pakietu. Ten identyfikator URI jest przechowywany w zmiennej o nazwie `imageUri`.

W `BlurViewModel`programie utwórz prywatną metodę o nazwie `createInputDataForUri`. Ta metoda powinna:

1. Utwórz `Data.Builder`obiekt. Importuj `androidx.work.Data`na żądanie.
2. Jeśli `imageUri`jest wartością inną niż null `URI`, dodaj ją do `Data`obiektu przy użyciu `putString`metody. Ta metoda przyjmuje klucz i wartość. Możesz użyć stałej String `KEY_IMAGE_URI`z `Constants`klasy.
3. Wezwij `build()`obiekt `Data.Builder`, aby zrobić swój `Data`obiekt i zwróć go.

Poniżej znajduje się ukończona `createInputDataForUri`metoda:

BlurViewModel.kt

```
/**
 * Creates the input data bundle which includes the Uri to operate on
 * @return Data which contains the Image Uri as a String
 */
private fun createInputDataForUri(): Data {
    val builder = Data.Builder()
    imageUri?.let {
        builder.putString(KEY_IMAGE_URI, imageUri.toString())
    }
    return builder.build()
}
```

Krok 2 — Przekaż obiekt Data do WorkRequest

Zamierzasz zmienić `applyBlur`metodę `BlurViewModel`tak, aby:

1. Tworzy nowy `OneTimeWorkRequestBuilder`.
2. Wywołania `setInputData`, przekazując wynik z `createInputDataForUri`.
3. Buduje `OneTimeWorkRequest`.

4. Umieszcza żądanie pracy w kolejce przy użyciu `WorkManager`żądania, dzięki czemu praca zostanie zaplanowana do uruchomienia.

Poniżej znajduje się ukończona `applyBlur`metoda:

BlurViewModel.kt

```
internal fun applyBlur(blurLevel: Int) {
    val blurRequest = OneTimeWorkRequestBuilder<BlurWorker>()
            .setInputData(createInputDataForUri())
            .build()

    workManager.enqueue(blurRequest)
}
```

Krok 3 — Zaktualizuj funkcję doWork() BlurWorkera, aby uzyskać dane wejściowe

Teraz zaktualizujmy metodę `BlurWorker`, `doWork()`aby uzyskać identyfikator URI, który przekazaliśmy z `Data`obiektu:

BlurWorker.kt

```
override fun doWork(): Result {
    val appContext = applicationContext

    // ADD THIS LINE
    val resourceUri = inputData.getString(KEY_IMAGE_URI)
    // ... rest of doWork()
}
```

Krok 4 — Rozmyj podany identyfikator URI

Za pomocą identyfikatora URI rozmyjmy teraz obraz babeczki na ekranie.

1. Usuń poprzedni kod, który pobierał zasób obrazu.

   `val picture = BitmapFactory.decodeResource(appContext.`*`resources`*`, R.drawable.`*`android_cupcake`*`)`

2. Sprawdź, czy `resourceUri`uzyskany z tego `Data`, który został przekazany, nie jest pusty.
3. Przypisz `picture`zmienną jako obraz, który został przekazany w następujący sposób:

   `val picture = BitmapFactory.decodeStream(`

   `appContext.`*`contentResolver`*`.`

   `openInputStream(Uri.parse(resourceUri)))`

BlurWorker.kt

```
override fun doWork(): Result {
    val appContext = applicationContext

    val resourceUri = inputData.getString(KEY_IMAGE_URI)

    makeStatusNotification("Blurring image", appContext)

    return try {
        // REMOVE THIS
        //    val picture = BitmapFactory.decodeResource(
        //            appContext.resources,
        //            R.drawable.android_cupcake)

        if (TextUtils.isEmpty(resourceUri)) {
            Log.e(TAG, "Invalid input uri")
            throw IllegalArgumentException("Invalid input uri")
        }

        val resolver = appContext.contentResolver

        val picture = BitmapFactory.decodeStream(
                resolver.openInputStream(Uri.parse(resourceUri)))

        val output = blurBitmap(picture, appContext)

        // Write bitmap to a temp file
        val outputUri = writeBitmapToFile(appContext, output)

        Result.success()
    } catch (throwable: Throwable) {
        Log.e(TAG, "Error applying blur")
        throwable.printStackTrace()
        Result.failure()
    }
}
```

Krok 5 — wyślij tymczasowy identyfikator URI

To już koniec procesu roboczego i możesz zwrócić wyjściowy identyfikator URI w `Result.success()`. Podaj wyjściowy identyfikator URI jako dane **wyjściowe** , aby ten tymczasowy obraz był łatwo dostępny dla innych pracowników do dalszych operacji. Przyda się to w następnym rozdziale, kiedy tworzysz łańcuch pracowników. Aby to zrobić:

1. Utwórz nowy `Data`, tak jak w przypadku danych wejściowych, i zapisz `outputUri`jako `String`. Użyj tego samego klawisza, `KEY_IMAGE_URI`.
2. Zwróć to do WorkManagera za pomocą `Result.success(Data outputData)`metody.

BlurWorker.kt

Zmodyfikuj `Result.success()`linię w `doWork()`:

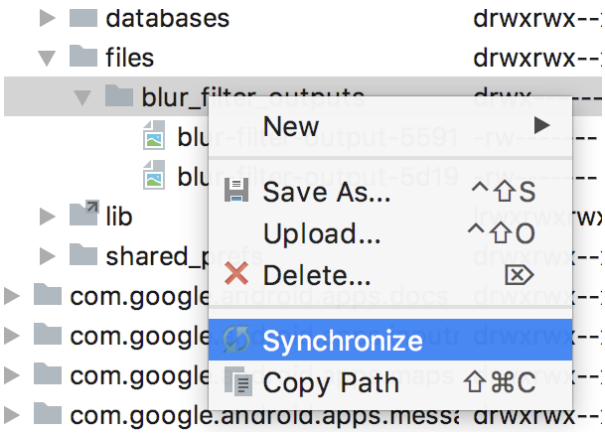
```
val outputData = workDataOf(KEY_IMAGE_URI to outputUri.toString())

Result.success(outputData)
```

Krok 6 — Uruchom aplikację

W tym momencie powinieneś uruchomić swoją aplikację. Powinien się skompilować i zachowywać tak samo, jak widać rozmazany obraz za pomocą Eksploratora plików urządzenia, ale jeszcze nie na ekranie.

Aby sprawdzić, czy nie ma innego rozmytego obrazu, możesz otworzyć **Eksplorator plików urządzenia** w Android Studio i przejść do **data/data/com.example.background/files/blur_filter_outputs/<URI>** , tak jak w poprzednim kroku.

Pamiętaj, że może być konieczna **synchronizacja** , aby zobaczyć swoje obrazy:



Świetna robota! Zamazałeś obraz wejściowy za pomocą `WorkManager`!

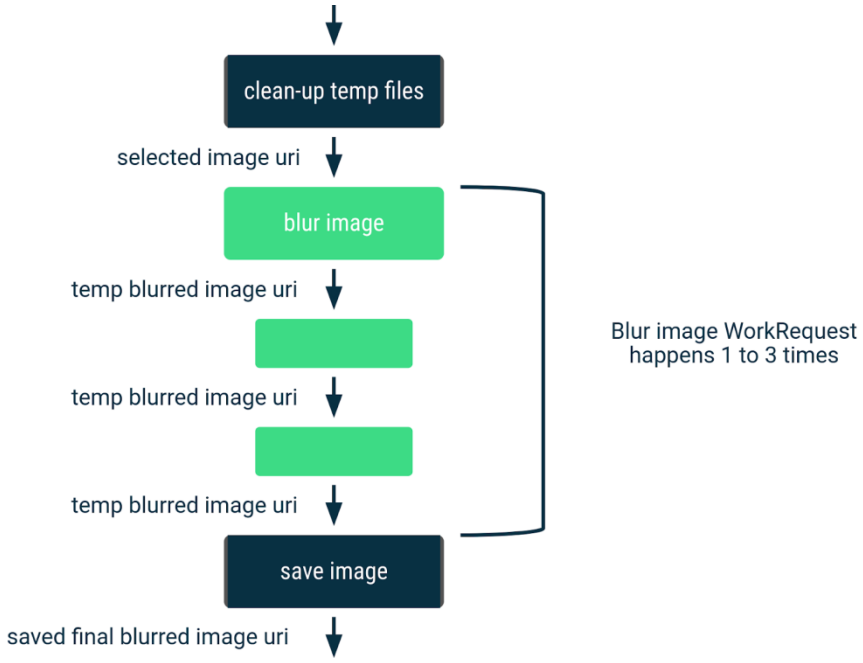
6. Połącz swoją pracę

W tej chwili wykonujesz jedno zadanie robocze: zamazujesz obraz. To świetny pierwszy krok, ale brakuje niektórych podstawowych funkcji:

- Nie czyści plików tymczasowych.
- W rzeczywistości nie zapisuje obrazu w trwałym pliku.
- Zawsze rozmywa obraz w takim samym stopniu.

Aby dodać tę funkcjonalność, użyjemy łańcucha pracy WorkManager.

WorkManager umożliwia tworzenie oddzielnych `WorkerRequest`s, które działają w kolejności lub równolegle. W tym kroku utworzysz łańcuch pracy, który wygląda tak:



Litery `WorkRequest`są reprezentowane jako pola.

Inną naprawdę fajną cechą łączenia jest to, że wyjście jednego `WorkRequest`staje się wejściem następnego `WorkRequest`w łańcuchu. Dane wejściowe i wyjściowe, które są przekazywane między nimi, `WorkRequest`są wyświetlane jako niebieski tekst.

Krok 1 — Utwórz czyszczenie i uratuj pracowników

Najpierw zdefiniujesz wszystkie `Worker`potrzebne klasy. Masz już `Worker`do rozmycia obrazu, ale potrzebujesz też, `Worker`który czyści pliki tymczasowe i `Worker`który zapisuje obraz na stałe.

Utwórz w pakiecie dwie nowe klasy `workers`, które rozszerzają `Worker`.

Pierwszy powinien być wywołany `CleanupWorker`, drugi powinien zostać wywołany `SaveImageToFileWorker`.

Krok 2 — spraw, aby rozszerzył się Worker

Rozszerz `CleanupWorker`klasę z `Worker`klasy. Dodaj wymagane parametry konstruktora.

```
class CleanupWorker(ctx: Context, params: WorkerParameters) : Worker(ctx, params) {
}
```

Krok 3 — Zastąp i zaimplementuj metodę doWork() dla CleanupWorker

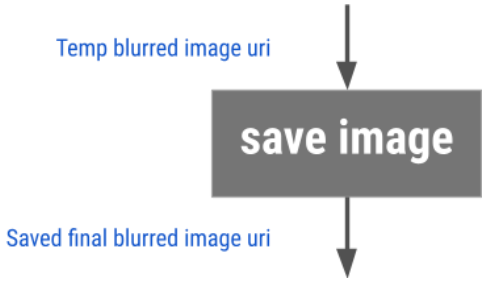
`CleanupWorker`nie musi pobierać żadnych danych wejściowych ani przekazywać żadnych danych wyjściowych. Zawsze usuwa pliki tymczasowe, jeśli istnieją. Ponieważ manipulacja plikami jest poza zakresem tego ćwiczenia z programowania, możesz skopiować kod `CleanupWorker`poniżej:

CleanupWorker.kt

```
package com.example.background.workers

import android.content.Context
import android.util.Log
import androidx.work.Worker
import androidx.work.WorkerParameters
import com.example.background.OUTPUT_PATH
import java.io.File

/**
 * Cleans up temporary files generated during blurring process
 */
private const val TAG = "CleanupWorker"
class CleanupWorker(ctx: Context, params: WorkerParameters) : Worker(ctx, params) {

    override fun doWork(): Result {
        // Makes a notification when the work starts and slows down the work so that
        // it's easier to see each WorkRequest start, even on emulated devices
        makeStatusNotification("Cleaning up old temporary files", applicationContext)
        sleep()

        return try {
            val outputDirectory = File(applicationContext.filesDir, OUTPUT_PATH)
            if (outputDirectory.exists()) {
                val entries = outputDirectory.listFiles()
                if (entries != null) {
                    for (entry in entries) {
                        val name = entry.name
                        if (name.isNotEmpty() && name.endsWith(".png")) {
                            val deleted = entry.delete()
                            Log.i(TAG, "Deleted $name - $deleted")
                        }
                    }
                }
            }
            Result.success()
        } catch (exception: Exception) {
            exception.printStackTrace()
            Result.failure()
        }
    }
}
```

Krok 4 — Zastąp i zaimplementuj metodę doWork() dla SaveImageToFileWorker

`SaveImageToFileWorker`przyjmie dane wejściowe i wyjściowe. Dane wejściowe to `String`tymczasowo zamazany identyfikator URI obrazu przechowywany z kluczem `KEY_IMAGE_URI`. Wyjściem będzie również `String`URI zapisanego zamazanego obrazu przechowywanego z kluczem `KEY_IMAGE_URI`.



Ponieważ nadal nie jest to laboratorium programowania dotyczące manipulacji plikami, kod znajduje się poniżej. Zwróć uwagę, jak wartości `resourceUri`i `output`są pobierane za pomocą klucza `KEY_IMAGE_URI`. Jest to bardzo podobne do kodu, który napisałeś w ostatnim kroku dla danych wejściowych i wyjściowych (używa wszystkich tych samych klawiszy).

SaveImageToFileWorker.kt

```
package com.example.background.workers

import android.content.Context
import android.graphics.BitmapFactory
import android.net.Uri
import android.provider.MediaStore
import android.util.Log
import androidx.work.workDataOf
import androidx.work.Worker
import androidx.work.WorkerParameters
import com.example.background.KEY_IMAGE_URI
import java.text.SimpleDateFormat
import java.util.Date
import java.util.Locale

/**
 * Saves the image to a permanent file
 */
private const val TAG = "SaveImageToFileWorker"
class SaveImageToFileWorker(ctx: Context, params: WorkerParameters) : Worker(ctx, params) {

    private val title = "Blurred Image"
    private val dateFormatter = SimpleDateFormat(
            "yyyy.MM.dd 'at' HH:mm:ss z",
            Locale.getDefault()
    )
```

```
    override fun doWork(): Result {
        // Makes a notification when the work starts and slows down the work so that
        // it's easier to see each WorkRequest start, even on emulated devices
        makeStatusNotification("Saving image", applicationContext)
        sleep()

        val resolver = applicationContext.contentResolver
        return try {
            val resourceUri = inputData.getString(KEY_IMAGE_URI)
            val bitmap = BitmapFactory.decodeStream(
                    resolver.openInputStream(Uri.parse(resourceUri)))
            val imageUrl = MediaStore.Images.Media.insertImage(
                    resolver, bitmap, title, dateFormatter.format(Date()))
            if (!imageUrl.isNullOrEmpty()) {
                val output = workDataOf(KEY_IMAGE_URI to imageUrl)

                Result.success(output)
            } else {
                Log.e(TAG, "Writing to MediaStore failed")
                Result.failure()
            }
        } catch (exception: Exception) {
            exception.printStackTrace()
            Result.failure()
        }
    }
}
```

Krok 5 — Zmodyfikuj powiadomienie BlurWorker

Teraz, gdy mamy łańcuch `Worker`s dbających o zapisanie obrazu w odpowiednim folderze, możemy spowolnić pracę, używając `sleep()`metody zdefiniowanej w `WorkerUtils`klasie, aby łatwiej było zobaczyć każdy `WorkRequest`start, nawet na emulowanych urządzeniach. Ostateczna wersja `BlurWorker`stała się:

BlurWorker.kt

```
class BlurWorker(ctx: Context, params: WorkerParameters) : Worker(ctx, params) {

override fun doWork(): Result {
    val appContext = applicationContext

    val resourceUri = inputData.getString(KEY_IMAGE_URI)

    makeStatusNotification("Blurring image", appContext)
```

```
    // ADD THIS TO SLOW DOWN THE WORKER
    sleep()
    // ^^^^

    return try {
        if (TextUtils.isEmpty(resourceUri)) {
            Timber.e("Invalid input uri")
            throw IllegalArgumentException("Invalid input uri")
        }

        val resolver = appContext.contentResolver

        val picture = BitmapFactory.decodeStream(
                resolver.openInputStream(Uri.parse(resourceUri)))

        val output = blurBitmap(picture, appContext)

        // Write bitmap to a temp file
        val outputUri = writeBitmapToFile(appContext, output)

        val outputData = workDataOf(KEY_IMAGE_URI to outputUri.toString())

        Result.success(outputData)
    } catch (throwable: Throwable) {
        throwable.printStackTrace()
        Result.failure()
    }
}
```

Krok 6 — Utwórz łańcuch WorkRequest

Musisz zmodyfikować metodę, aby wykonać łańcuch `BlurViewModel`s zamiast tylko jednego. Obecnie kod wygląda tak:`applyBlurWorkRequest`

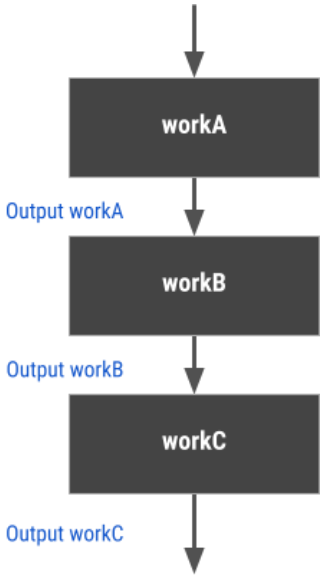
BlurViewModel.kt

```
val blurRequest = OneTimeWorkRequestBuilder<BlurWorker>()
        .setInputData(createInputDataForUri())
        .build()

workManager.enqueue(blurRequest)
```

Zamiast dzwonić `workManager.enqueue()`, zadzwoń `workManager.beginWith()`. Zwraca a [WorkContinuation], który definiuje łańcuch `WorkRequest`s. Możesz dodać do tego łańcucha żądań pracy, wywołując `then()`metodę, na przykład, jeśli masz trzy `WorkRequest`obiekty, `workA`, `workB`i `workC`, możesz wykonać następujące czynności:

```
// Example code, don't copy to the project
val continuation = workManager.beginWith(workA)

continuation.then(workB) // FYI, then() returns a new WorkContinuation instance
        .then(workC)
        .enqueue() // Enqueues the WorkContinuation which is a chain of work
```

Spowoduje to utworzenie i uruchomienie następującego łańcucha WorkRequests:



Utwórz łańcuch `CleanupWorker` `WorkRequest`,
a `BlurImage` `WorkRequest`i `SaveImageToFile` `WorkRequest`w `applyBlur`. Wprowadź dane do `BlurImage` `WorkRequest`.

Kod do tego znajduje się poniżej:

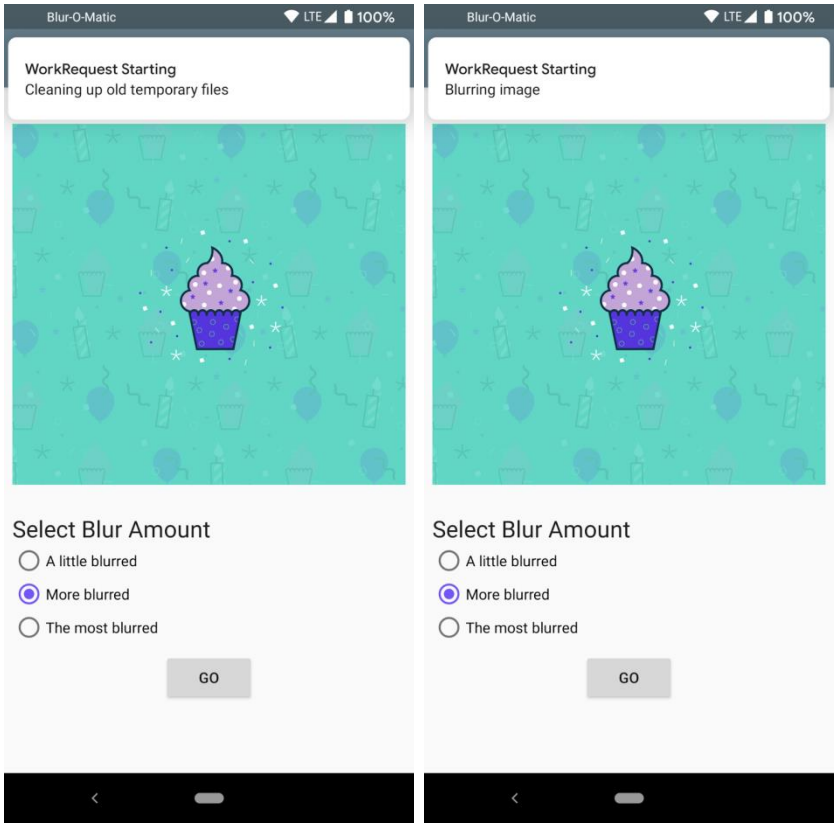
BlurViewModel.kt

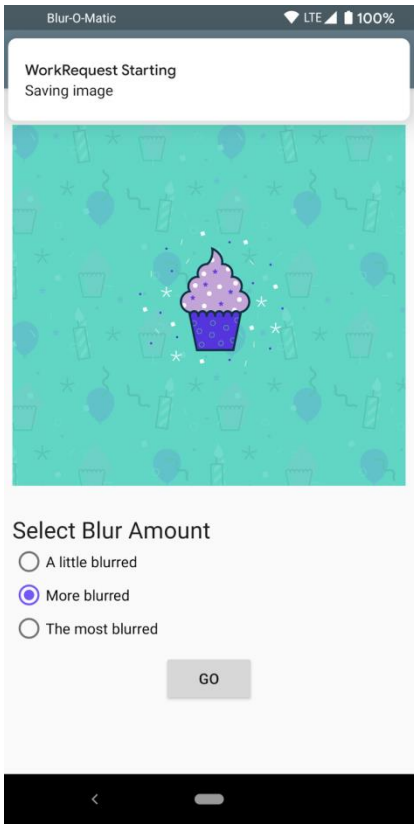
```
internal fun applyBlur(blurLevel: Int) {
    // Add WorkRequest to Cleanup temporary images
    var continuation = workManager
            .beginWith(OneTimeWorkRequest
            .from(CleanupWorker::class.java))

    // Add WorkRequest to blur the image
    val blurRequest = OneTimeWorkRequest.Builder(BlurWorker::class.java)
            .setInputData(createInputDataForUri())
            .build()

    continuation = continuation.then(blurRequest)

    // Add WorkRequest to save the image to the filesystem
    val save = OneTimeWorkRequest.Builder(SaveImageToFileWorker::class.java).build()
```

```
    continuation = continuation.then(save)

    // Actually start the work
    continuation.enqueue()
}
```

To powinno się **skompilować** i **uruchomić** . Powinieneś teraz być w stanie nacisnąć przycisk **Go** i zobaczyć powiadomienia, gdy różni pracownicy wykonują. Nadal będziesz mógł zobaczyć rozmazany obraz w Eksploratorze plików urządzenia, a w nadchodzącym kroku dodasz dodatkowy przycisk, aby użytkownicy mogli zobaczyć rozmazany obraz na urządzeniu.

Na poniższych zrzutach ekranu zauważysz, że powiadomienia wyświetlają, który pracownik jest aktualnie uruchomiony.





Krok 7 - Powtórz BlurWorker

Czas dodać możliwość rozmycia obrazu o różne wartości. Weź `blurLevel`przekazany parametr `applyBlur`i dodaj tyle `WorkRequest`operacji rozmycia do łańcucha. Tylko pierwsze `WorkRequest`potrzeby i powinny pobierać dane wejściowe uri.

Zauważ, że jest to nieco wymyślone do celów edukacyjnych. Trzykrotne wywołanie naszego kodu rozmycia jest mniej wydajne niż `BlurWorker`przyjęcie danych wejściowych, które kontrolują „poziom" rozmycia. Ale pozwala nam to pokazać elastyczność łączenia WorkManagera.

Wypróbuj sam, a następnie porównaj z poniższym kodem:

BlurViewModel.kt

```
internal fun applyBlur(blurLevel: Int) {
    // Add WorkRequest to Cleanup temporary images
    var continuation = workManager
            .beginWith(OneTimeWorkRequest
            .from(CleanupWorker::class.java))

    // Add WorkRequests to blur the image the number of times requested
    for (i in 0 until blurLevel) {
        val blurBuilder = OneTimeWorkRequestBuilder<BlurWorker>()

        // Input the Uri if this is the first blur operation
        // After the first blur operation the input will be the output of previous
```

```
        // blur operations.
        if (i == 0) {
            blurBuilder.setInputData(createInputDataForUri())
        }

        continuation = continuation.then(blurBuilder.build())
    }

    // Add WorkRequest to save the image to the filesystem
    val save = OneTimeWorkRequestBuilder<SaveImageToFileWorker>()
            .build()

    continuation = continuation.then(save)

    // Actually start the work
    continuation.enqueue()
}
```

Otwórz eksplorator plików urządzenia, aby zobaczyć rozmyte obrazy. Zwróć uwagę, że folder wyjściowy zawiera wiele rozmytych obrazów, obrazy, które są na pośrednich etapach rozmycia, oraz końcowy obraz, który wyświetla rozmyty obraz na podstawie wybranej wielkości rozmycia.

Znakomita "praca"! Teraz możesz rozmazać obraz tak bardzo lub tak mało, jak chcesz! Jakie tajemnicze!

7. Zapewnij wyjątkową pracę

Teraz, gdy już używałeś łańcuchów, nadszedł czas, aby zająć się inną potężną funkcją WorkManagera - **unikalnymi łańcuchami pracy** .
Czasami potrzebujesz tylko jednego łańcucha pracy na raz. Na przykład, być może masz łańcuch roboczy, który synchronizuje dane lokalne z serwerem — prawdopodobnie chcesz, aby pierwsza synchronizacja danych zakończyła się przed rozpoczęciem nowej. Aby to zrobić, użyjesz `beginUniqueWork` zamiast `beginWith`; i podajesz unikalną `String` nazwę. Nadaje to nazwę **całemu** łańcuchowi żądań pracy, dzięki czemu można się do nich odwoływać i razem wysyłać zapytania.
Upewnij się, że Twój łańcuch pracy nad rozmyciem pliku jest unikalny, używając `beginUniqueWork`. Podaj `IMAGE_MANIPULATION_WORK_NAME` jako klucz. Musisz także zdać `ExistingWorkPolicy`. Masz do wyboru `REPLACE`, `KEEP` lub `APPEND`.

Użyjesz, `REPLACE` ponieważ jeśli użytkownik zdecyduje się zamazać inny obraz przed zakończeniem bieżącego, chcemy zatrzymać bieżący i zacząć zamazywać nowy obraz.

Poniżej znajduje się kod umożliwiający rozpoczęcie unikalnej kontynuacji pracy:

BlurViewModel.kt

```
// REPLACE THIS CODE:
// var continuation = workManager
//         .beginWith(OneTimeWorkRequest
```

```
//          .from(CleanupWorker::class.java))
// WITH
var continuation = workManager
    .beginUniqueWork(
        IMAGE_MANIPULATION_WORK_NAME,
        ExistingWorkPolicy.REPLACE,
        OneTimeWorkRequest.from(CleanupWorker::class.java)
    )
```

Blur-O-Matic będzie teraz rozmazywać tylko jeden obraz na raz.

8. Oznacz i wyświetl status pracy

Ta sekcja intensywnie korzysta z LiveData , więc aby w pełni zrozumieć, co się dzieje, powinieneś znać LiveData. LiveData to obserwowalny, świadomy cykl życia posiadacz danych. Jeśli po raz pierwszy pracujesz z LiveData lub obiektami obserwowalnymi , możesz zapoznać się z dokumentacją lub modułami Codelab uwzględniającymi cykl życia systemu Android .

Następną dużą zmianą, którą zrobisz, jest zmiana tego, co jest wyświetlane w aplikacji podczas wykonywania Pracy.

Możesz uzyskać status dowolnego `WorkRequest`poprzez uzyskanie a `LiveData`, które przechowuje `WorkInfo`obiekt. `WorkInfo`to obiekt, który zawiera szczegółowe informacje o aktualnym stanie a `WorkRequest`, w tym:

- Czy praca to `BLOCKED`, `CANCELLED`, `ENQUEUED`, `FAILED`, `RUNNING`czy `SUCCEEDED`.
- Po `WorkRequest`zakończeniu wszelkie dane wyjściowe z pracy.

Poniższa tabela przedstawia trzy różne sposoby zdobywania obiektów i ich działania `LiveData<WorkInfo>`.`LiveData<List<WorkInfo>>`

Rodzaj	**Metoda WorkManagera**	**Opis**
Uzyskaj pracę za pomocą **identyfikatora**	getWorkInfoByIdLiveData	Każdy WorkRequestposiada unikalny identyfikator wygenerowany przez WorkManager; możesz użyć tego, aby uzyskać jeden LiveData dokładnie za to WorkRequest.
Zdobądź pracę, używając **unikalnej nazwy sieci**	getWorkInfosForUniqueWorkLiveData	Jak właśnie zauważyłeś, WorkRequests może być częścią wyjątkowego łańcucha. Zwraca się to dla wszystkich prac w jednym, unikalnym łańcuchu .LiveData<list ></list WorkRequests
Uzyskaj pracę za pomocą **tagu**	getWorkInfosByTagLiveData	Na koniec możesz opcjonalnie oznaczyć dowolne WorkRequest za pomocą ciągu. Możesz oznaczyć wiele WorkRequests tym samym tagiem, aby je

powiązać. Zwraca to dla dowolnego pojedynczego tagu.LiveData<list
></list

Będziesz otagowywać SaveImageToFileWorker WorkRequest, aby można było je uzyskać za pomocą getWorkInfosByTag. Do oznaczenia swojej pracy użyjesz tagu zamiast identyfikatora WorkManager, ponieważ jeśli użytkownik zamazuje wiele obrazów, wszystkie zapisane obrazy WorkRequestbędą miały ten sam tag, ale *nie* ten sam identyfikator. Również jesteś w stanie wybrać tag.

Nie użyłbyś getWorkInfosForUniqueWork, ponieważ zwróciłoby to również WorkInfodla wszystkich rozmyć WorkRequesti czyszczenia WorkRequest; znalezienie zapisanego obrazu wymagałoby dodatkowej logiki WorkRequest.

Krok 1 - Oznacz swoją pracę

W applyBlurprogramie , podczas tworzenia SaveImageToFileWorkertagu , otaguj swoją pracę za pomocą Stringstałej TAG_OUTPUT:

BlurViewModel.kt

```
val save = OneTimeWorkRequestBuilder<SaveImageToFileWorker>()
        .addTag(TAG_OUTPUT) // <-- ADD THIS
        .build()
```

Krok 2 — Pobierz WorkInfo

Teraz, gdy już otagowałeś pracę, możesz uzyskać WorkInfo:

1. W BlurViewModelprogramie zadeklaruj nową zmienną klasy o nazwie outputWorkInfosaLiveData<List<WorkInfo>>
2. Dodaj blok BlurViewModelstartowy, aby uzyskać WorkInfoużycieWorkManager.getWorkInfosByTagLiveData

Potrzebny kod znajduje się poniżej:

BlurViewModel.kt

```
// New instance variable for the WorkInfo
internal val outputWorkInfos: LiveData<List<WorkInfo>>

// Modify the existing init block in the BlurViewModel class to this:
init {
    imageUri = getImageUri(application.applicationContext)
    // This transformation makes sure that whenever the current work Id changes the WorkInfo
    // the UI is listening to changes
    outputWorkInfos = workManager.getWorkInfosByTagLiveData(TAG_OUTPUT)
}
```

Krok 3 — Wyświetl informacje robocze

Teraz, gdy masz już `LiveData`swoje `WorkInfo`, możesz to zaobserwować w `BlurActivity`. W obserwatorze:

1. Sprawdź, czy lista `WorkInfo`nie jest pusta i czy nie zawiera żadnych `WorkInfo`obiektów - jeśli nie to przycisk **Idź** nie został jeszcze kliknięty, więc wróć.
2. Zdobądź pierwszy `WorkInfo`na liście; zawsze będzie `WorkInfo`oznaczony tylko jeden `TAG_OUTPUT`, ponieważ sprawiliśmy, że łańcuch pracy jest wyjątkowy.
3. Sprawdź, czy status pracy jest zakończony, używając `workInfo.state.isFinished`.
4. Jeśli to nie koniec, wywołaj, `showWorkInProgress()`który ukrywa przycisk **Go** i wyświetla przycisk **Anuluj pracę** i pasek postępu.
5. Po zakończeniu wywołania `showWorkFinished()`, które ukrywa przycisk **Anuluj pracę** i pasek postępu oraz wyświetla przycisk **Przejdź** .

Oto kod:

Uwaga: importuj `androidx.lifecycle.Observer`na żądanie.

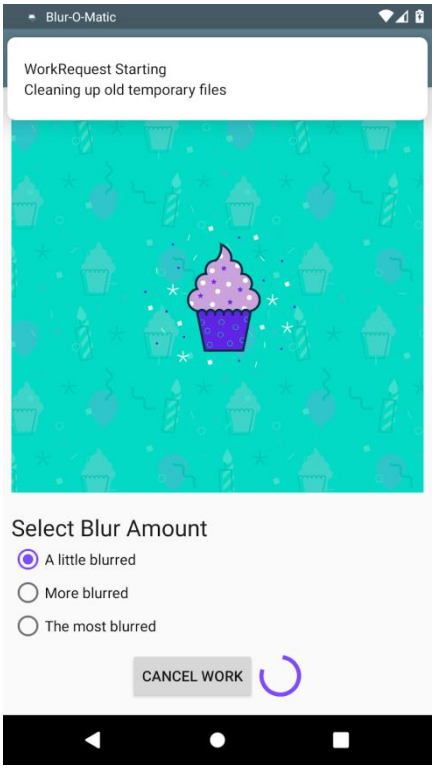
BlurActivity.kt

```
override fun onCreate(savedInstanceState: Bundle?) {
    ...
    // Observe work status, added in onCreate()
    viewModel.outputWorkInfos.observe(this, workInfosObserver())
}

// Define the observer function
private fun workInfosObserver(): Observer<List<WorkInfo>> {
    return Observer { listOfWorkInfo ->

        // Note that these next few lines grab a single WorkInfo if it exists
        // This code could be in a Transformation in the ViewModel; they are included here
        // so that the entire process of displaying a WorkInfo is in one location.

        // If there are no matching work info, do nothing
        if (listOfWorkInfo.isNullOrEmpty()) {
            return@Observer
        }

        // We only care about the one output status.
        // Every continuation has only one worker tagged TAG_OUTPUT
        val workInfo = listOfWorkInfo[0]

        if (workInfo.state.isFinished) {
            showWorkFinished()
        } else {
            showWorkInProgress()
        }
```

```
    }
}
```

Krok 4 — Uruchom aplikację

Uruchom swoją aplikację - powinna się skompilować i uruchomić, a teraz pokazywać pasek postępu, gdy działa, a także przycisk anulowania:



9. Pokaż ostateczny wynik

Każdy `WorkInfo`ma również `getOutputData`metodę, która pozwala uzyskać `Data`obiekt wyjściowy z końcowym zapisanym obrazem. W Kotlinie możesz uzyskać dostęp do tej metody za pomocą zmiennej generowanej przez język: `outputData`. Wyświetlajmy przycisk z napisem **Zobacz plik** , gdy jest gotowy do wyświetlenia zamazany obraz.

Krok 1 — Utwórz przycisk „Zobacz plik”

W `activity_blur.xml`układzie jest już ukryty przycisk. Jest w `BlurActivity`i nazywa się `outputButton`.

W `BlurActivity`, wewnątrz `onCreate()`skonfiguruj odbiornik kliknięć dla tego przycisku. Powinien uzyskać identyfikator URI, a następnie otworzyć działanie, aby wyświetlić ten identyfikator URI. Możesz użyć poniższego kodu:

BlurActivity.kt

```
override fun onCreate(savedInstanceState: Bundle?) {
  // Setup view output image file button
  binding.seeFileButton.setOnClickListener {
     viewModel.outputUri?.let { currentUri ->
        val actionView = Intent(Intent.ACTION_VIEW, currentUri)
        actionView.resolveActivity(packageManager)?.run {
           startActivity(actionView)
        }
     }
  }
}
```

Krok 2 — Ustaw identyfikator URI i pokaż przycisk

Jest kilka ostatnich poprawek, które musisz zastosować do `WorkInfo`obserwatora, aby to zadziałało (gra słów nie jest przeznaczona):

1. Jeśli `WorkInfo`zakończy się, pobierz dane wyjściowe, używając `workInfo.outputData`.
2. Następnie pobierz wyjściowy URI, pamiętaj, że jest on przechowywany z `Constants.KEY_IMAGE_URI`kluczem.
3. Następnie, jeśli identyfikator URI nie jest pusty, zostanie poprawnie zapisany; pokaż `outputButton`i wywołaj `setOutputUri`model widoku z uri.

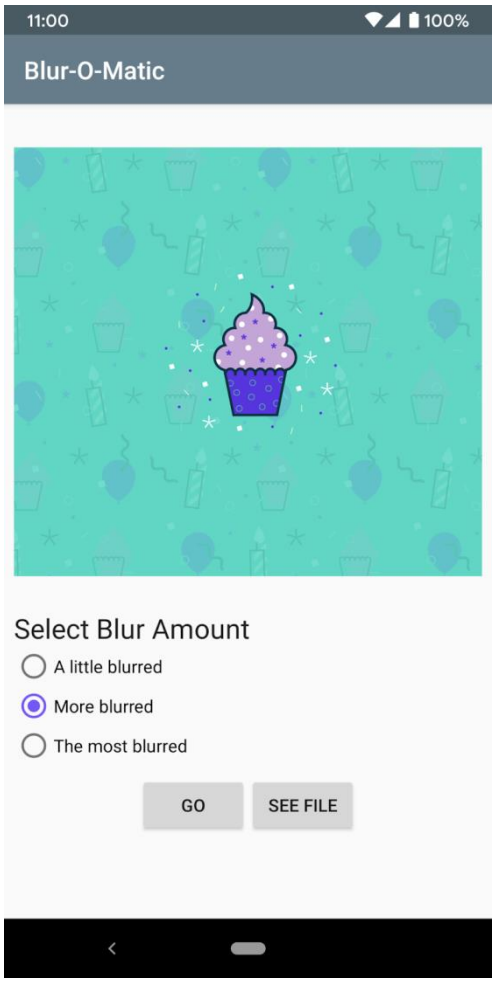
BlurActivity.kt

```
private fun workInfosObserver(): Observer<List<WorkInfo>> {
   return Observer { listOfWorkInfo ->

      // Note that these next few lines grab a single WorkInfo if it exists
      // This code could be in a Transformation in the ViewModel; they are included here
      // so that the entire process of displaying a WorkInfo is in one location.

      // If there are no matching work info, do nothing
      if (listOfWorkInfo.isNullOrEmpty()) {
         return@Observer
      }

      // We only care about the one output status.
      // Every continuation has only one worker tagged TAG_OUTPUT
      val workInfo = listOfWorkInfo[0]

      if (workInfo.state.isFinished) {
         showWorkFinished()
```

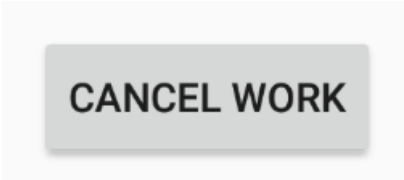
```
            // Normally this processing, which is not directly related to drawing views on
            // screen would be in the ViewModel. For simplicity we are keeping it here.
            val outputImageUri = workInfo.outputData.getString(KEY_IMAGE_URI)

            // If there is an output file show "See File" button
            if (!outputImageUri.isNullOrEmpty()) {
                viewModel.setOutputUri(outputImageUri)
                binding.seeFileButton.visibility = View.VISIBLE
            }
        } else {
            showWorkInProgress()
        }
    }
}
```

Krok 3 — Uruchom swój kod

Uruchom swój kod. Powinieneś zobaczyć nowy, klikalny przycisk **Zobacz plik** , który przeniesie Cię do pliku wyjściowego:



10. Anuluj pracę



Dodałeś ten przycisk **Anuluj pracę** , więc dodajmy kod, aby coś zrobił. Dzięki WorkManager możesz anulować pracę za pomocą identyfikatora, tagu i unikalnej nazwy łańcucha.

W takim przypadku będziesz chciał anulować pracę według unikalnej nazwy łańcucha, ponieważ chcesz anulować całą pracę w łańcuchu, a nie tylko konkretny krok.

Krok 1 - Anuluj pracę według nazwy

W programie `BlurViewModel`dodaj nową metodę wywoływaną, `cancelWork()`aby anulować unikatową pracę. Wewnątrz wywołania funkcji `cancelUniqueWork`na `workManager`, przekaż znacznik `IMAGE_MANIPULATION_WORK_NAME`.

BlurViewModel.kt

```
internal fun cancelWork() {
    workManager.cancelUniqueWork(IMAGE_MANIPULATION_WORK_NAME)
```

}

Krok 2 — Metoda anulowania połączenia

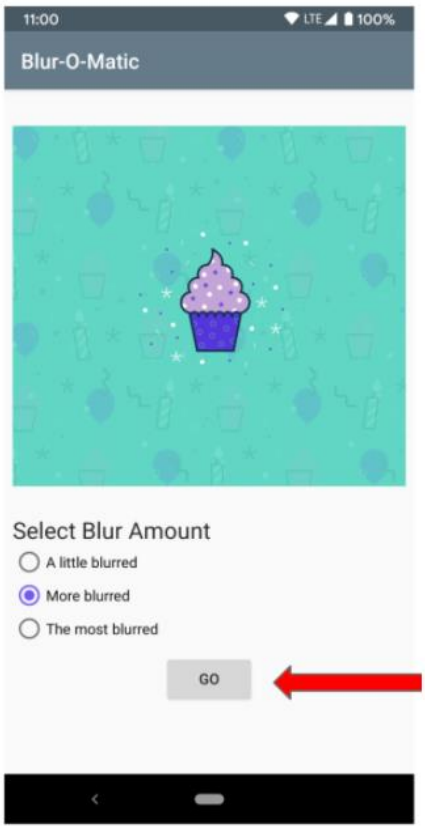
Następnie podłącz przycisk, `cancelButton`aby zadzwonić `cancelWork`:

BlurActivity.kt

```
// In onCreate()
// Hookup the Cancel button
binding.cancelButton.setOnClickListener { viewModel.cancelWork() }
```

Krok 3 - Uruchom i anuluj pracę

Uruchom swoją aplikację. Powinien się dobrze skompilować. Rozpocznij zamazywanie obrazu, a następnie kliknij przycisk Anuluj. Cały łańcuch jest anulowany!



Zauważ, że teraz jest tylko przycisk START po anulowaniu pracy, ponieważ WorkState nie jest już w stanie ZAKOŃCZONY.

11. Ograniczenia pracy

Last but not least, `WorkManager` obsługuje `Constraints`. W przypadku Blur-O-Matic użyjesz ograniczenia, że urządzenie musi się ładować. Oznacza to, że Twoje żądanie pracy będzie działać tylko wtedy, gdy urządzenie się ładuje.

Krok 1 — Utwórz i dodaj ograniczenie ładowania

Aby utworzyć `Constraints` obiekt, użyj `Constraints.Builder`. Następnie ustaw żądane ograniczenia i dodaj je do `WorkRequest` metody, `setRequiresCharging()` jak pokazano poniżej:

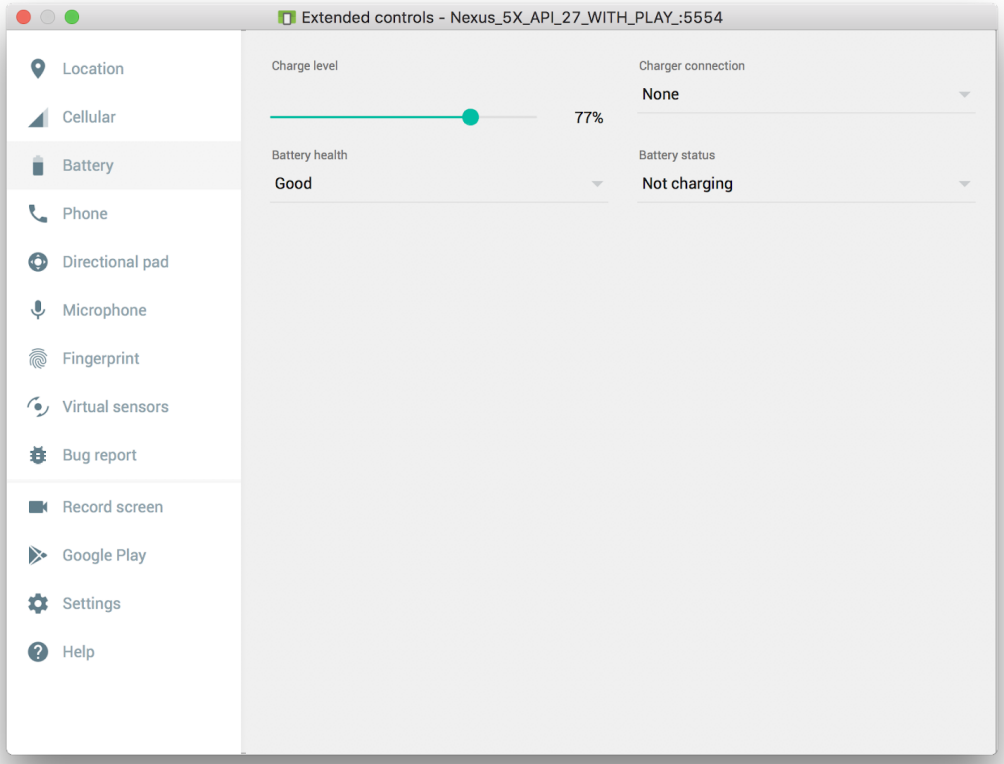
Importuj `androidx.work.Constraints` na żądanie.

BlurViewModel.kt

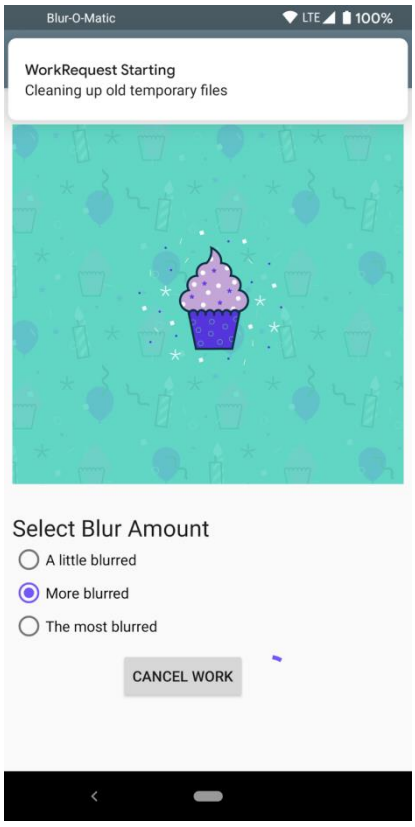
```
// Put this inside the applyBlur() function, above the save work request.
// Create charging constraint
val constraints = Constraints.Builder()
        .setRequiresCharging(true)
        .build()

// Add WorkRequest to save the image to the filesystem
val save = OneTimeWorkRequestBuilder<SaveImageToFileWorker>()
        .setConstraints(constraints)
        .addTag(TAG_OUTPUT)
        .build()
continuation = continuation.then(save)

// Actually start the work
continuation.enqueue()
```

Krok 2 — Przetestuj za pomocą emulatora lub urządzenia

Teraz możesz uruchomić Blur-O-Matic. Jeśli korzystasz z urządzenia, możesz je odłączyć lub podłączyć. W emulatorze możesz zmienić stan ładowania w oknie **Rozszerzone sterowanie** :



Gdy urządzenie się nie ładuje, powinno zawiesić `SaveImageToFileWorker`, jego wykonywanie dopiero po podłączeniu.



Kolejnym dobrym ograniczeniem do dodania do Blur-O-Matic byłoby setRequiresStorageNotLowograniczenie podczas zapisywania. Aby zobaczyć pełną listę opcji ograniczeń, zapoznaj się z Constraints.Builderodnośnikiem.

12. Gratulacje

Gratulacje! Ukończyłeś aplikację Blur-O-Matic i dowiedziałeś się o:

- Dodawanie WorkManagera do projektu
- PlanowanieOneTimeWorkRequest
- Parametry wejściowe i wyjściowe
- Łączenie prac razem WorkRequests
- Nazywanie unikalnych WorkRequestłańcuchów
- Tagowanie WorkRequests
- Wyświetlanie WorkInfow interfejsie użytkownika
- Anulowanie WorkRequests
- Dodawanie ograniczeń doWorkRequest

Wspaniała robota"! Aby zobaczyć końcowy stan kodu i wszystkie zmiany, sprawdź:

file_downloadPobierz ostateczny kod

Lub, jeśli wolisz, możesz sklonować laboratorium kodowania WorkManagera z GitHub:

```
$ git clone https://github.com/googlecodelabs/android-workmanager
```

WorkManager obsługuje znacznie więcej, niż moglibyśmy omówić w tym laboratorium, w tym pracę powtarzalną, bibliotekę wsparcia testowania, żądania pracy równoległej i łączenie danych wejściowych. Aby dowiedzieć się więcej, przejdź do dokumentacji WorkManager lub przejdź do Advanced WorkManager codelab .

Projekt: Nawadniaj mnie! aplikacja

1. Zanim zaczniesz

To laboratorium programowania przedstawia nową aplikację o nazwie Water Me, którą zbudujesz samodzielnie. To laboratorium kodowania poprowadzi Cię przez kolejne kroki, aby ukończyć projekt aplikacji Water Me, w tym konfigurację projektu i testowanie w Android Studio.

Warunki wstępne

- Ten projekt jest przeznaczony dla studentów, którzy ukończyli 6. część kursu Android Basics in Kotlin.

Co zbudujesz

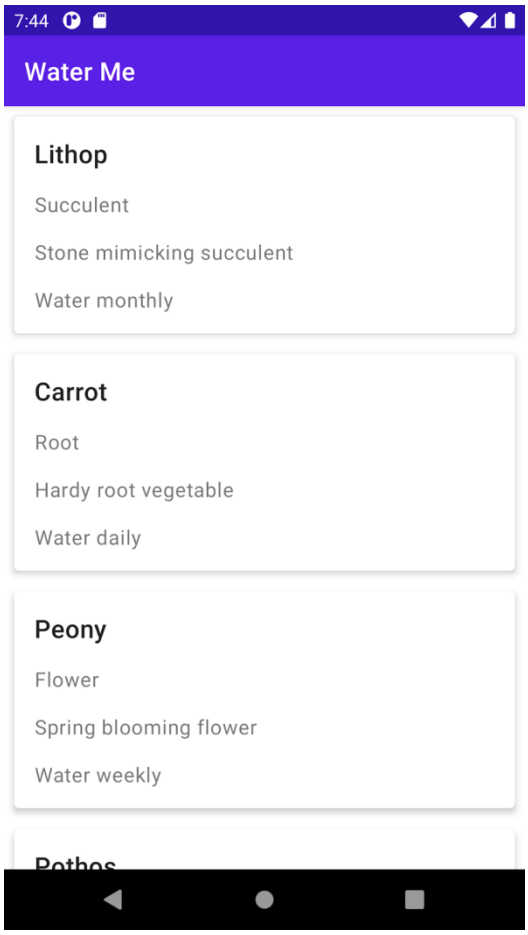
- Zaplanuj powiadomienie za pomocą niestandardowego `Worker`w istniejącej aplikacji.

Co będziesz potrzebował

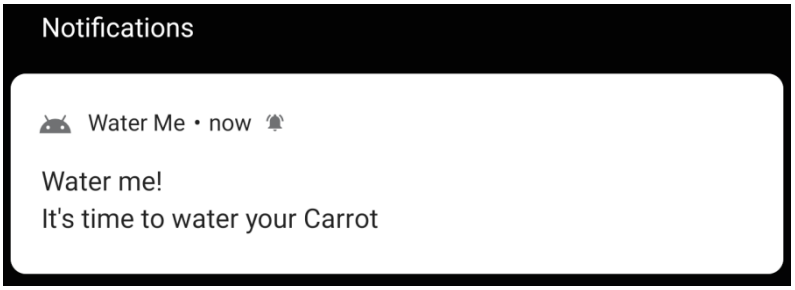
- Komputer z zainstalowanym Android Studio.

2. Zakończony przegląd aplikacji

Woda ja! Aplikacja składa się z listy roślin, informacji na ich temat oraz opisu częstotliwości podlewania każdej z nich. Dla każdej z tych roślin ukończona aplikacja zaplanuje przypomnienie, kiedy należy je podlewać.



Przypomnienia będą wyświetlane jako powiadomienia na urządzeniu, nawet jeśli Water Me! aplikacja nie działa. Dotknięcie powiadomienia uruchamia Water Me! aplikacja.



Aby ta funkcja działała, Twoim zadaniem jest zaplanowanie zadania w tle przy użyciu niestandardowej `Worker`opcji wyświetlającej powiadomienie.

3. Rozpocznij

Pobierz kod projektu

Zauważ, że nazwa folderu to `android-basics-kotlin-water-me-app`. Wybierz ten folder podczas otwierania projektu w Android Studio.

Adres URL kodu startowego:

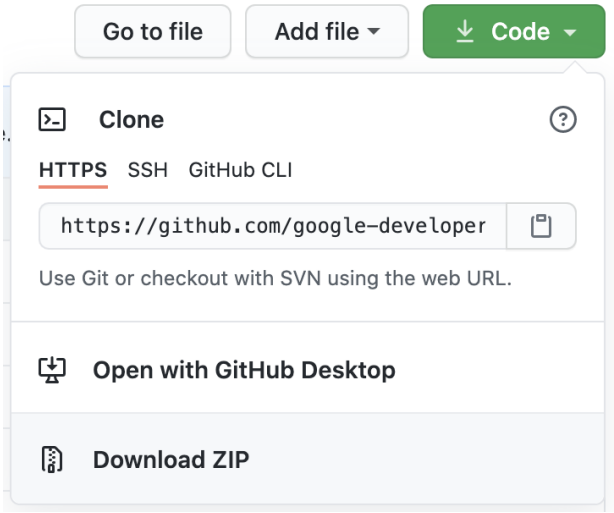
https://github.com/google-developer-training/android-basics-kotlin-water-me-app/tree/main

Nazwa oddziału z kodem startowym:main

Aby uzyskać kod tego ćwiczenia z programowania i otworzyć go w Android Studio, wykonaj następujące czynności.

Pobierz kod

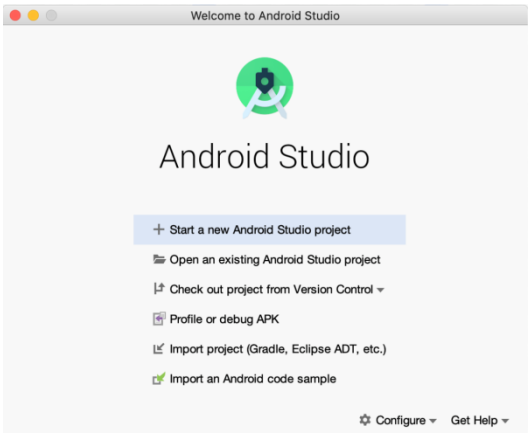
1. Kliknij podany adres URL. Spowoduje to otwarcie strony GitHub projektu w przeglądarce.
2. Na stronie GitHub projektu kliknij przycisk **Kod** , który wyświetli okno dialogowe.



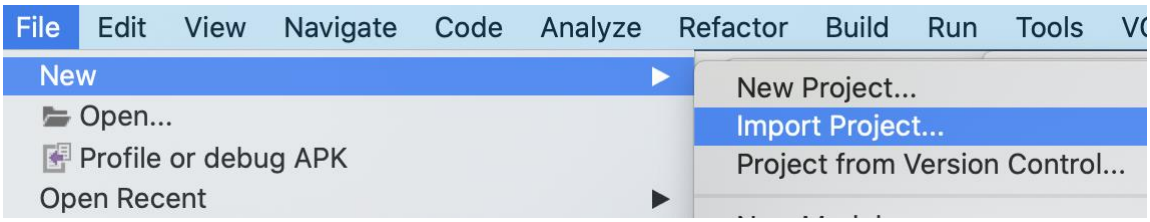
3. W oknie dialogowym kliknij przycisk **Pobierz ZIP** , aby zapisać projekt na swoim komputerze. Poczekaj na zakończenie pobierania.
4. Znajdź plik na swoim komputerze (prawdopodobnie w folderze **Pobrane).**
5. Kliknij dwukrotnie plik ZIP, aby go rozpakować. Spowoduje to utworzenie nowego folderu zawierającego pliki projektu.

Otwórz projekt w Android Studio

1. Uruchom Android Studio.
2. W oknie **Witamy w Android Studio** kliknij **Otwórz istniejący projekt Android Studio.**



Uwaga: jeśli Android Studio jest już otwarte, wybierz opcję menu **Plik** > **Nowy** > **Importuj projekt** .



3. W oknie dialogowym **Importuj projekt** przejdź do miejsca, w którym znajduje się rozpakowany folder projektu (prawdopodobnie w folderze **Pobrane**).
4. Kliknij dwukrotnie ten folder projektu.
5. Poczekaj, aż Android Studio otworzy projekt.
6. Kliknij przycisk **Uruchom** , aby skompilować i uruchomić aplikację. Upewnij się, że kompiluje się zgodnie z oczekiwaniami.
7. Przeglądaj pliki projektu w oknie narzędzia **Projekt** , aby zobaczyć, jak skonfigurowana jest aplikacja.

4. Zaplanuj powiadomienia za pomocą WorkManager

Cała funkcjonalność Water Me! aplikacja jest już zaimplementowana, z wyjątkiem części do zaplanowania i powiadomienia. Kod do wyświetlenia powiadomienia znajduje się w `WaterReminderWorker.kt`(w pakiecie **worker).** Dzieje się tak w `doWork()`metodzie `Worker`klasy niestandardowej. Ponieważ powiadomienia mogą być nowym tematem, ten kod jest już zaimplementowany.

```
override fun doWork(): Result {
    val intent = Intent(applicationContext, MainActivity::class.java).apply {
        flags = Intent.FLAG_ACTIVITY_NEW_TASK or Intent.FLAG_ACTIVITY_CLEAR_TASK
    }

    val pendingIntent: PendingIntent = PendingIntent
        .getActivity(applicationContext, 0, intent, 0)

    val plantName = inputData.getString(nameKey)

    val builder = NotificationCompat.Builder(applicationContext, BaseApplication.CHANNEL_ID)
        .setSmallIcon(R.drawable.ic_android_black_24dp)
        .setContentTitle("Water me!")
        .setContentText("It's time to water your $plantName")
        .setPriority(NotificationCompat.PRIORITY_HIGH)
        .setContentIntent(pendingIntent)
        .setAutoCancel(true)

    with(NotificationManagerCompat.from(applicationContext)) {
```

```
            notify(notificationId, builder.build())
        }

        return Result.success()
    }
```

Twoim zadaniem jest stworzenie `OneTimeWorkRequest`metody, która wywoła tę metodę z poprawnymi parametrami z `PlantViewModel`.

Twórz zlecenia pracy.

Aby zaplanować powiadomienie, musisz zaimplementować `scheduleReminder()`metodę w `PlantViewModel.kt`.

1. Utwórz zmienną o nazwie `data`using `Data.Builder`. Dane powinny składać się z pojedynczej wartości ciągu, gdzie `WaterReminder.Worker.nameKey`jest kluczem, a `plantName`przekazana `scheduleReminder()`wartość jest wartością.
2. Utwórz jednorazowe żądanie pracy za pomocą `WaterReminderWorker`, używając `delay`i `unit`przekazanego do `scheduleReminder()`funkcji i ustawiając dane wejściowe na utworzoną `data`zmienną.
3. Wywołaj metodę `workManager`, `enqueueUniqueWork()`przekazując nazwę rośliny, używając `REPLACE`jako `ExistingWorkPolicy`, oraz żądanie pracy.

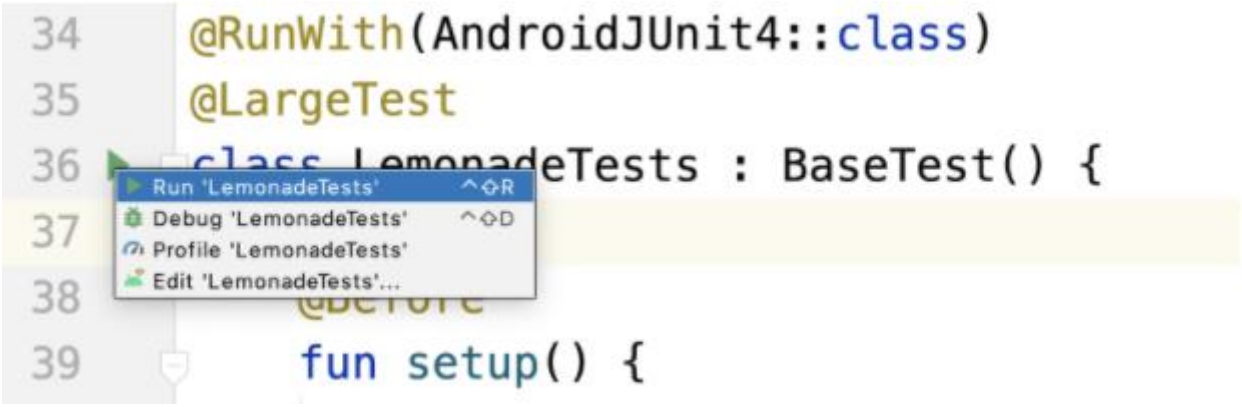
Twoja aplikacja powinna teraz działać zgodnie z oczekiwaniami. Ponieważ pojawienie się każdego przypomnienia zajmie dużo czasu, zalecamy przeprowadzenie dołączonych testów, aby sprawdzić, czy powiadomienie działa zgodnie z oczekiwaniami.

5. Instrukcje testowania

Przeprowadzanie testów

Aby uruchomić testy, możesz wykonać jedną z poniższych czynności.

W przypadku pojedynczego przypadku testowego otwórz klasę przypadku testowego i kliknij zieloną strzałkę po lewej stronie deklaracji klasy. Następnie możesz wybrać z menu opcję Uruchom. Spowoduje to uruchomienie wszystkich testów w przypadku testowym.



Często będziesz chciał uruchomić tylko jeden test, na przykład, jeśli tylko jeden test się nie powiedzie, a pozostałe testy zakończą się pomyślnie. Pojedynczy test można uruchomić tak samo, jak cały przypadek testowy. Użyj zielonej strzałki i wybierz opcję **Uruchom** .