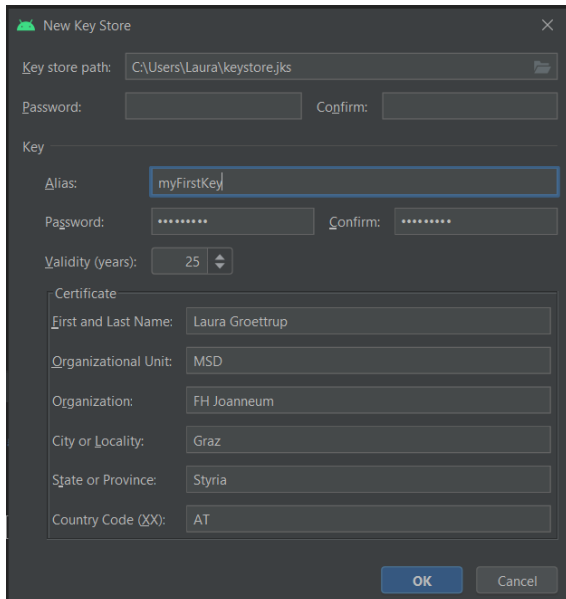


Continuous Delivery Ex 4 Android Build

Von Laura Gröttrup, MSD18, WS20/21

1) Create a signing config

Zuerst muss ein Keystore erstellt werden. Diesen habe ich mit der Android Studio UI unter „Build -> Generate Signed Bundle/APK -> APK -> Key Store Path -> Create new -> “ erstellt. In der gleichen Maske in der ich auch meinen Keystore finalisiert habe, habe ich auch direkt einen Key erstellt.



Nun da ich einen Key in meinem Keystore habe, bearbeitete ich die build.gradle folgendermaßen, damit meine APKs beim release-build signiert werden (im Debug werden sie automatisch von Android signiert):

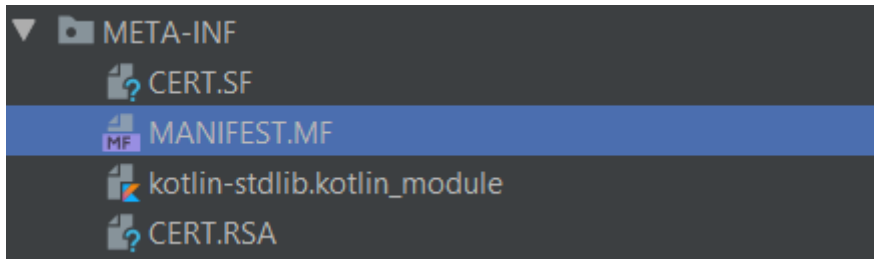
```
signingConfigs {
    release {
        storeFile = file('C:\\Users\\Laura\\keystore.jks')
        storePassword = file('keystore.txt').getText('UTF-8')
        keyAlias "myFirstKey"
        keyPassword = file('key.txt').getText('UTF-8')
        v1SigningEnabled true
        v2SigningEnabled true
    }
}
buildTypes {
    release {
        minifyEnabled false
        proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rules.pro'
        signingConfig signingConfigs.release
    }
}
```

Die Passwörter für den Key sowie den Keystore lese ich mit der Methode .getText() aus im Projekt liegenden Text-Dateien aus.

1b) Build the APK and verify the signature

Mittels des Befehles „gradlew assembleRelease“ Kann nun eine Release-APK erstellt werden.

Analisiert man nun die APK, fallen die Dateien CERT.SF und CERT.RSA im META-INF auf. Diese gehören zu der v1 Signierung unserer App.



Betrachten wir aber die APK mit dem apksigner und dem Befehl „apksigner verify -v <apkpath.apk>“, wir sehen nur dass das v2 Scheme verwendet wird. Das liegt daran dass wir bei unserem Projekt eine minSdkVersion von 24 verwenden, und bei allen Projekten mit Level 24 aufwärts die v2 Signatur verwendet wird.

```
C:\Users\Laura\AppData\Local\Android\Sdk\build-tools\29.0.3>apksigner verify -v C:\Users\Laura\Documents\Universitaet\CI\ex4_groettrup\stopwatch-ex4\app\build\outputs\apk\free\release\app-free-release.apk
Verifies
Verified using v1 scheme (JAR signing): false
Verified using v2 scheme (APK Signature Scheme v2): true
Verified using v3 scheme (APK Signature Scheme v3): false
Number of signers: 1
C:\Users\Laura\AppData\Local\Android\Sdk\build-tools\29.0.3>
```

2) Modify the debug build

Als nächstes wollen wir einen Parameter, die Geschwindigkeit der Stoppuhr, für die Debug-APK viermal so schnell setzen wie in der Release-APK.

Dafür setzen wir folgendermaßen ein neues Build Configurations Feld:

```
defaultConfig {
    applicationId "at.fhj.msd.stopwatch"
    minSdkVersion 24
    targetSdkVersion 29
    versionCode 1
    versionName "1.0"
    buildConfigField "int", "SPEED_FACTOR", "1"

    testInstrumentationRunner "androidx.test.runner.AndroidJUnitRunner"
}
```

Nun gibt es einen Parameter SPEED_FACTOR, den man über die “BuildConfig” erreichen kann. Dieser ist Standardmässig auf 1 gesetzt. Für die Debug-Version setzen erhöhen wir ihn um Faktor 4.

```

buildTypes {
    release {
        minifyEnabled false
        proguardFiles getDefaultProguardFile('proguard-android.txt'), 'proguard-rules.pro'
        signingConfig signingConfigs.release
    }
    debug {
        buildConfigField "int", "SPEED_FACTOR", "4"
    }
}

```

Schlussendlich nutzen wir diesen neu angelegten Parameter in unserem Programm als Variablenbelegung im StopWatchViewModel.

```

class StopWatchViewModel : ViewModel() {

    private val timer: ScheduledExecutorService =
        Executors.newScheduledThreadPool(1)
    private val lapTimesList = mutableListOf<LapTime>()
    private var timerTask: ScheduledFuture<*>? = null

    val isRunning = MutableLiveData<Boolean>(false)
    val speedFactor = BuildConfig.SPEED_FACTOR //use this to speed up the clock
    for easier debugging
}

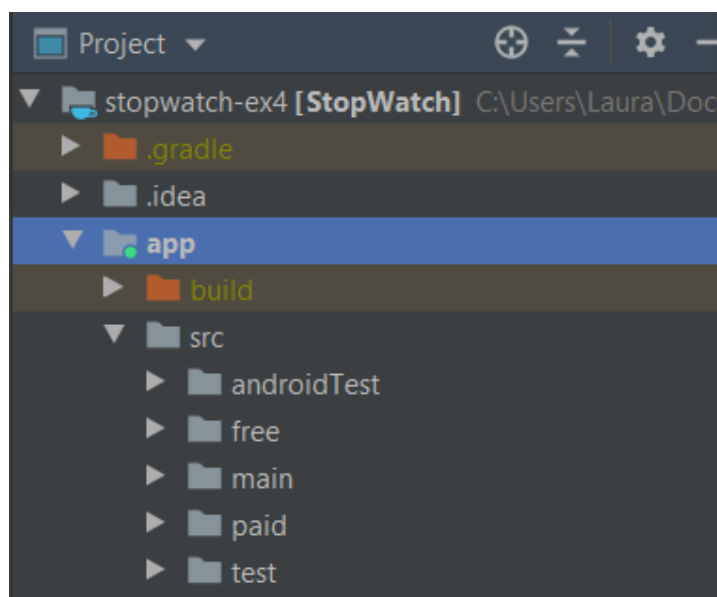
```

Bilden wir nun beide APK-Versionen und lassen sie nebeneinander laufen, fällt auf das die Uhr in der Debugversion deutlich schneller ist als in der Release Version.

3) Create an app flavor

Im nächsten Schritt wollen wir zwei verschiedene Appversionen bereitstellen: Einmal mit und einmal ohne Werbung, die durch einen Surfix im Namen unterschieden werden können.

Da wir unterschiedliche Ressourcen in diesen Apps brauchen werden, einmal ein Layout mit und einmal ohne Werbung, legen wir zwei neue Ordner „free“ und „paid“ in unserem Sourcefolder an.



Wir kopieren den Ressourcenfolder aus Main in beide dieser Ordner und löschen ihn aus den Main Ordner da wir dorthin den jeweils passenden Ressourcen Ordner rüber kopieren werden. Im Paid-Ordner bearbeiten wir das `fragment_stopwatch.xml` Layout sodass es keine Werbung mehr anzeigt.

Nun legen wir zwei verschiedene Flavors in unserer `build.gradle` an, mit dem entsprechenden Surfix für beide Versionen.

```
flavorDimensions 'version'
productFlavors {
    paid {
        dimension 'version'
        applicationIdSuffix '.paid'
    }
    free {
        dimension 'version'
        applicationIdSuffix '.free'
    }
}
```

Für alle möglichen Kombinationen unserer Build Variants definieren wir nun, welcher Ressource Ordner verwendet werden soll. Dieser wird im Buildprozess in den main Ordner kopiert, woraus die APK schlussendlich erstellt wird.

```
sourceSets {
    paidRelease { res.srcDirs = ['src/paid/res', 'src/main/res'] }
    paidDebug { res.srcDirs = ['src/paid/res', 'src/main/res'] }
    freeRelease { res.srcDirs = ['src/free/res', 'src/main/res'] }
    freeDebug { res.srcDirs = ['src/free/res', 'src/main/res'] }
}
```

Wenn wir nun unser Projekt bauen, werden vier APKs erstellt.

4) Add Unit tests

Schlussendlich erstellen wir noch ein paar Unit Tests, die unsere `StopWatchViewModel` Klasse testen sollen.

Mein erster Test überprüft, ob Runden in der Liste gespeichert werden wenn die Methode `takeLapTime()` aufgerufen wird, und ob alle Einträge mit der Methode `clearLaps()` gelöscht werden.

```
@Test
fun testLapTimeLists(){
    viewModel.toggleStartStop()
    viewModel.takeLapTime()
    viewModel.takeLapTime()
    viewModel.toggleStartStop()
    assertEquals(viewModel.lapTimes.getOrAwaitValue().size, 2)
    viewModel.clearLaps()
    assertEquals(viewModel.lapTimes.getOrAwaitValue().size, 0)
}
```

Mein zweiter Test überprüft, ob sich die Zeit verändert wenn die Stopuhr mit toggleStartStop() gestartet wird, und ob die Zeit mit der Methode reset() wieder auf den ursprünglichen Wert von 00:00.00 gesetzt wird.

```
@Test
fun testTakingTimeReset(){
    viewModel.toggleStartStop()
    Thread.sleep(400L)
    viewModel.toggleStartStop()
    assertEquals(viewModel.elapsedTimeString.getOrAwaitValue(), "00:00.00")
    viewModel.reset()
    assertEquals(viewModel.elapsedTimeString.getOrAwaitValue(), "00:00.00")
}
```