

Подготовка к публикации



APK (Android Package)

Формат файла, используемый для распространения и установки приложений на устройствах с операционной системой Android.

По своей сути, APK-файл - ZIP архив, содержащий все необходимые компоненты приложения: скомпилированный код (.dex файлы), ресурсы (например, изображения, звуки), манифест приложения, а также метаданные.



Пример содержимого APK

com.example.my.test.mytestapplicationxml (Version Name: 1.0, Version Code: 1)

APK size: 5,6 MB, Download Size: 4,6 MB

Compare with previous APK...

File	Raw File Size	Download Size	% of Total Download Size
classes.dex	4 MB	3,6 MB	84,7% <div></div>
> res	325,7 KB	317,6 KB	7,3% <div></div>
resources.arsc	1019,7 KB	220,3 KB	5,1% <div></div>
classes2.dex	126 KB	112,4 KB	2,6% <div></div>
> kotlin	10,2 KB	10,1 KB	0,2%
classes3.dex	2,1 KB	2 KB	0%
AndroidManifest.xml	1,6 KB	1,6 KB	0%
> META-INF	809 B	879 B	0%
DebugProbesKt.bin	782 B	782 B	0%

Load Proguard mappings...

This dex file defines 6411 classes with 49691 methods, and references 60550 methods.

Class	Defined Methods	Referenced Methods	Size
> androidx	24604	26759	2,4 MB
> com	10610	11759	896 KB
> kotlin	9931	10966	922,6 KB
> android	82	4850	45,8 KB
> kotlinx	4441	4676	437,5 KB
> java		1490	12,1 KB
> org	23	45	2,8 KB
> sun		2	16 B
> float[]		1	8 B
> int[]		1	8 B
> long[]		1	8 B

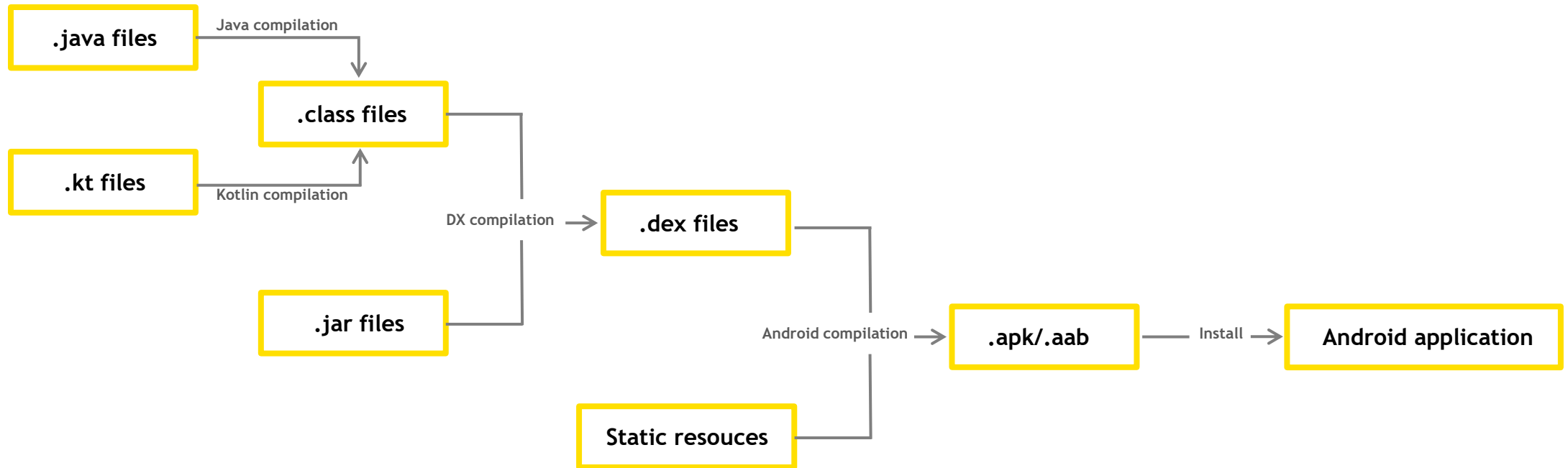
AAB (Android App Bundle)

Представил в 2018 году в качестве альтернативы традиционным APK-файлам для публикации приложений в сторсах, как более эффективный и гибкий способ упаковки приложения Android.

Основные особенности формата AAB:

1. **Динамическая доставка (Dynamic Delivery)** — позволяет загружать части приложения по мере необходимости. Например, пользователь может загрузить основной функционал приложения сразу, а дополнительные функции — позже, по запросу.
2. **Оптимизация для устройства** — скомпилированный код и ресурсы разделены на отдельные модули, в отличие от традиционного APK, который включает в себя весь код и ресурсы для всех возможных конфигураций устройства. AAB-файлы позволяют генерировать APK файлы, оптимизированные для конкретные устройства, пользователю загружается только необходимый код и ресурсы, подходящие для его устройства, что уменьшает размер установочного файла.
3. **Невозможность прямой установки** — важно отметить, что AAB-файлы не могут быть установлены непосредственно на устройства пользователей, как это возможно с APK-файлами. Они предназначены исключительно для публикации и генерации APK-файлов.

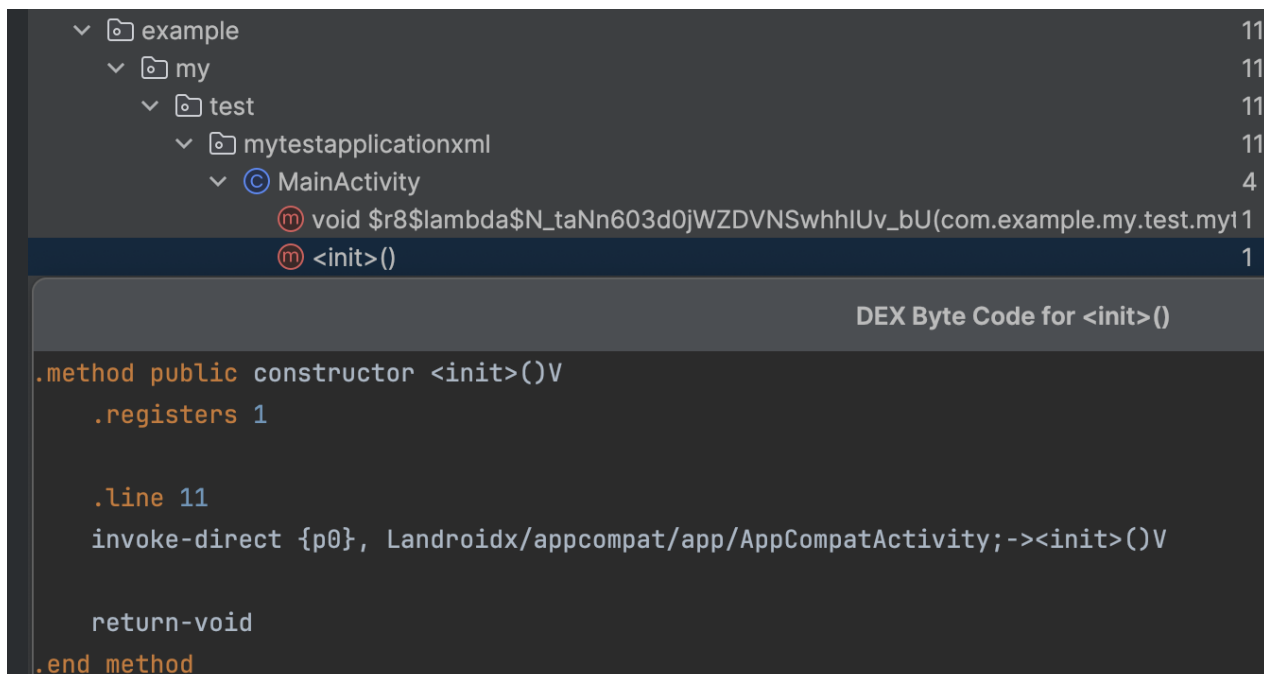
Процесс сборки



.dex (Dalvik executable)

Результат компиляции кода приложения. Формат оптимизирован для выполнения в Android среде.

Байт-код содержит низкоуровневые инструкции, которые могут эффективно выполняться в Dalvik или ART. Обычно включает в себя инструкции по загрузке и хранению данных, вызовы методов, операторы управления потока выполнения и т.д.



```

example 11
└─ my 11
   └─ test 11
      └─ mytestapplication.xml 11
         └─ MainActivity 4
            void $r8$lambda$N_taNn603d0jWZDVNSwhhIUv_bU(com.example.my.test.my1 1
            <init>() 1

```

DEX Byte Code for <init>()

```

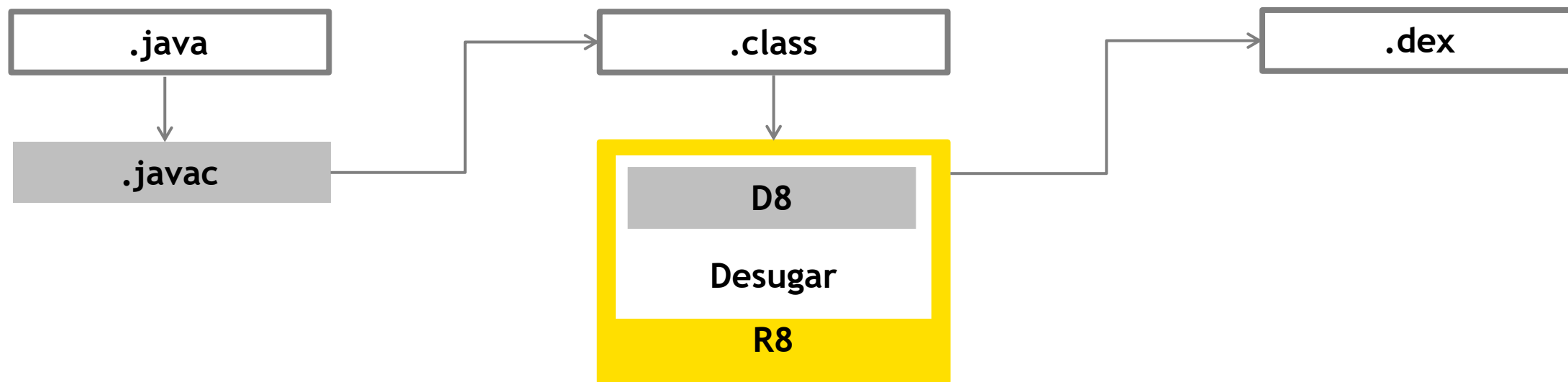
.method public constructor <init>()V
    .registers 1

    .line 11
    invoke-direct {p0}, Landroidx/appcompat/app/AppCompatActivity; -> <init>()V

    return-void
.end method

```

Процесс сборки



D8 - компилятор по умолчанию, с версии AGP 3.2, вместо DX.

R8 - оптимизатор по умолчанию, с версии AGP 3.4, вместо ProGuard.

Функции:

1. Уменьшение занимаемой памяти
2. Повышение скорости работы
3. Повышение безопасности

Процесс сборки

isDebuggable - выключение возможности отладки.

isMinifyEnabled - включает удаление неиспользуемого кода, обфускацию и оптимизацию для кода приложения и зависимого кода.

isShrinkResources - включает удаление неиспользуемых ресурсов для кода приложения и зависимого кода.

getDefaultProguardFile - добавляет стандартные правила ProGuard, генерируемые AGP, специфичные для android приложений.

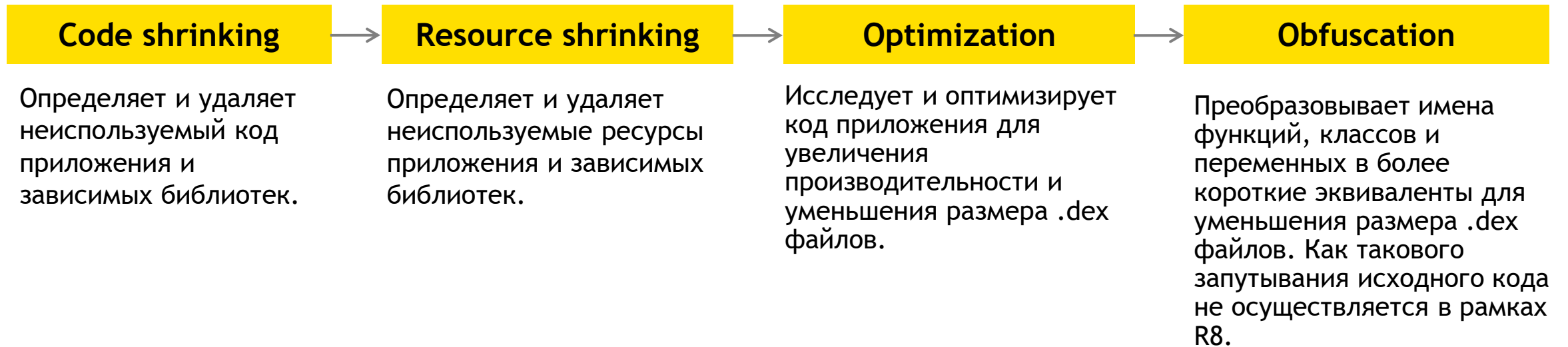
- **proguard-android-optimize.txt** - файл с инструкциями оптимизации.
- **proguard-android.txt** - файл без инструкций оптимизаций.

«**proguard-rules.pro**» - добавляет локальный, кастомный файл с правилами ProGuard, специфичные для приложения.

```
buildTypes {  
    release {  
        isDebuggable = false  
        isMinifyEnabled = true  
        isShrinkResources = true
```

```
    proguardFiles(getDefaultProguardFile("proguard-  
        android-optimize.txt"), "proguard-rules.pro")  
    }  
}
```


Процесс сборки (R8)



`androidx.appcompat.app.ActionBarDrawerToggle$DelegateProvider` -> `a.a.a.b:`

`androidx.appcompat.app.AlertController` -> `androidx.appcompat.app.AlertController:`

`android.content.Context mContext` -> `a`

Keep rules

Допустим, есть класс, который напрямую не используется в коде (Например, при рефлексии):

Чтобы сохранить класс от удаления и переименования нужно:

1. Либо пометить его аннотацией `@Keep`:

```
rules
Kotlin

import androidx.annotation.Keep

@Keep
class SomeApiResponseClass(...)
```

2. Либо прописать правило в файле `proguard-rules.pro` для конкретного класса:

```
-keep class
com.example.my.test.mytestapplicatio
nxml.data.network.models.SomeApiRe
sponseClass { *;}
```

3. Либо так, если хотим сохранить все классы лежащие во всем проекте в папках `data/network/models`:

```
-keep class
com.example.my.test.mytestapplicati
onxml**.data.network.models.** { *; }
```

rules

Kotlin

```
package com.example.my.test.mytestapplicationxml.data.network.models

class SomApiResponseClass(
    val id: Int,
    val value: String,
)
```

Keep rules

Если же имя класса нам необязательно, но важны поля, чтобы, например, правильно распарсить ответ от сервера, то можно поля пометить специальной аннотацией из библиотеки для сериализации и разрешить обфускацию полей.

Prguard-rule.pro:

```
-keepclassmembers,allowobfuscation class  
com.example.my.test.mytestapplicationxml.** {  
    @kotlinx.serialization.SerialName <fields>;  
}
```

rules

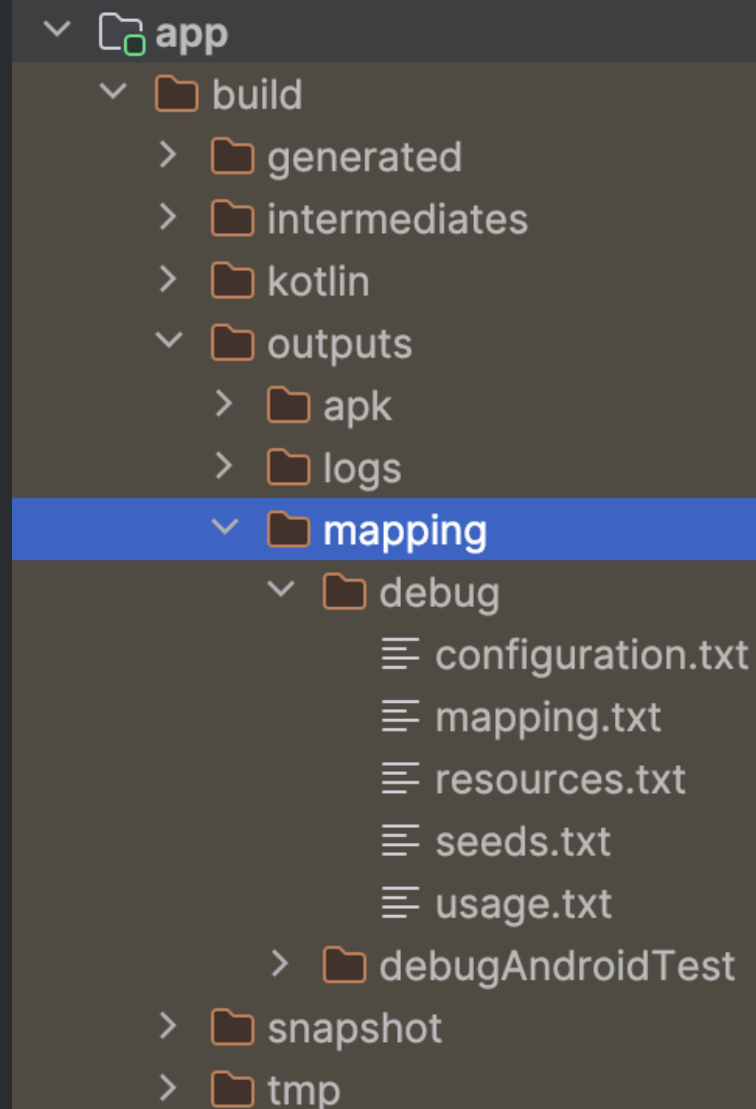
Kotlin

```
import kotlinx.serialization.SerialName  
import kotlinx.serialization.Serializable  
  
@Serializable  
class SomeApiResponseClass(  
    @SerialName("id")  
    val id: Int,  
    @SerialName("value")  
    val value: String,  
)
```

outputs/mapping

При сборке приложения, на основе установленных параметров, генерируются дополнительные файлы в каталоге:

build/outputs/mapping/<build_type>



mapping/ configuration.txt

В файле собраны правила конфигурации ProGuard со всего проекта, примененные в процессе оптимизации.

Зависимые библиотеки так же могут содержать правила ProGuard, нужно быть с этим осторожным, так как эти правила применяются ко всему проекту (Например, библиотека может отключить оптимизации).

```
...
# Optimizations: If you don't want to optimize, use
the proguard-android.txt configuration file
# instead of this one, which turns off the
optimization flags.
-allowaccessmodification

# Preserve some attributes that may be required for
reflection.
-keepattributes AnnotationDefault,
                EnclosingMethod,
                InnerClasses,
                RuntimeVisibleAnnotations,
                RuntimeVisibleParameterAnnotations,
                RuntimeVisibleTypeAnnotations,
                Signature

-keep public class
com.google.vending.licensing.ILicensingService
-keep public class
com.android.vending.licensing.ILicensingService
...
```

mapping/ seeds.txt

Описывает входные точки приложения, определенные при работе R8.

```
...
androidx.lifecycle.LegacySavedStateHandleController$tryToAdd
Recreator$1
androidx.startup.InitializationProvider
android.support.v4.app.RemoteActionCompatParcelizer
com.example.my.test.mytestapplicationxml.MainActivity
androidx.activity.ImmLeaksCleaner
com.google.android.material.search.SearchView$Behavior
com.example.my.test.mytestapplicationxml.SecondActivity
androidx.annotation.Keep
...
```

mapping/ usage.txt

Описывает удаленный код в
процессе оптимизации.

```
...
androidx.viewpager2.widget.ViewPager2$Smooth
hScrollToPosition
androidx.viewpager2.widget.ViewPager2
com.example.my.test.mytestapplicationxml.Pa
ramsClass:
    public void <init>()
    public final java.lang.String
getParamA()
    public final int getParamB()
com.example.my.test.mytestapplicationxml.R$
color
com.example.my.test.mytestapplicationxml.R$
drawable
...
```

mapping/usage.txt

Kotlin

```
class ParamsClass(
    val paramA: String = "", val paramB: Int = 0,
) {

    fun getParams(): String {
        throw IllegalStateException("missing params")
    }
}

println(ParamsClass().getParams())
```

mapping/ resources.txt

Описывает информацию по ресурсам, какие используются, какие нет.

```
...
@string/side_sheet_accessibility_pane_title :
reachable=true
@string/side_sheet_behavior : reachable=false
@string/status_bar_notification_info_overflow :
reachable=false
@string/unused_string : reachable=false
@style/AlertDialog_AppCompat : reachable=false
    @style/Base_AlertDialog_AppCompat
@style/AlertDialog_AppCompat_Light :
reachable=false
    @style/Base_AlertDialog_AppCompat_Light
@style/Animation_AppCompat_Dialog :
reachable=false
...
```




mapping/ mapping.txt

Генерируется R8 при оптимизации кода. Используется, чтобы конвертировать stack trace в исходный вид.

Рекомендуется сохранять файл на каждый релиз (файл перезаписывается при новых сборках), чтобы иметь возможность расшифровать вывод.

Для каждой версии приложения в Google Play можно приложить соответствующий mapping.txt, чтобы в Play Console видеть расшифрованный вывод репортов.

mapping/mapping.txt

```
com.example.my.test.mytestapplicationxml.ParamsClass -> R$$$REMOVED$$$CLASS$$$102:
# {"id":"sourceFile","fileName":"MainActivity.kt"}
com.example.my.test.mytestapplicationxml.SecondActivity -> com.example.my.test.mytestapplicationxml.SecondActivity:
# {"id":"sourceFile","fileName":"MainActivity.kt"}
    6:9:void
com.example.my.test.mytestapplicationxml.ParamsClass.<init>(java.lang.String,int,int,kotlin.jvm.internal.DefaultConstructorMarker):0:0 -> onCreate
    6:9:void onCreate(android.os.Bundle):0 -> onCreate
    10:14:void com.example.my.test.mytestapplicationxml.ParamsClass.<init>(java.lang.String,int):0:0 -> onCreate
    10:14:void
com.example.my.test.mytestapplicationxml.ParamsClass.<init>(java.lang.String,int,int,kotlin.jvm.internal.DefaultConstructorMarker):0 -> onCreate
    10:14:void onCreate(android.os.Bundle):0 -> onCreate
    15:22:java.lang.String com.example.my.test.mytestapplicationxml.ParamsClass.getParams():0:0 -> onCreate
    15:22:void onCreate(android.os.Bundle):0 -> onCreate
com.example.my.test.mytestapplicationxml.data.network.models.SomeApiResponseClass ->
com.example.my.test.mytestapplicationxml.data.network.models.SomeApiResponseClass:
# {"id":"sourceFile","fileName":"SomeApiResponseClass.kt"}
com.google.android.material.R$styleable -> j0.a:
    int[] TextInputLayout -> A
    int[] ThemeEnforcement -> B
    int[] BottomSheetBehavior_Layout -> a
```

Merged Manifest

В процессе сборки apk создается объединенный AndroidManifest.xml, в котором содержится информация из всех манифестов приложения и зависимых библиотек.

Потенциально, зависимые библиотеки могут подкинуть компоненты, которые вашему приложению не нужны или запрашивать опасные разрешения, влияющие на приватную информацию пользователя.

В таких случаях можно воспользоваться Merge rule markers:

1. Манифест из библиотеки:

```
<activity-alias
android:name="com.example.alias">
  <meta-data android:name="cow"
    android:value="@string/moo"/>
</activity-alias>
```

2. Переопределение компонента:

```
<activity-alias android:name="com.example.alias">
  <meta-data android:name="cow"
    tools:node="remove"/>
</activity-alias>
```

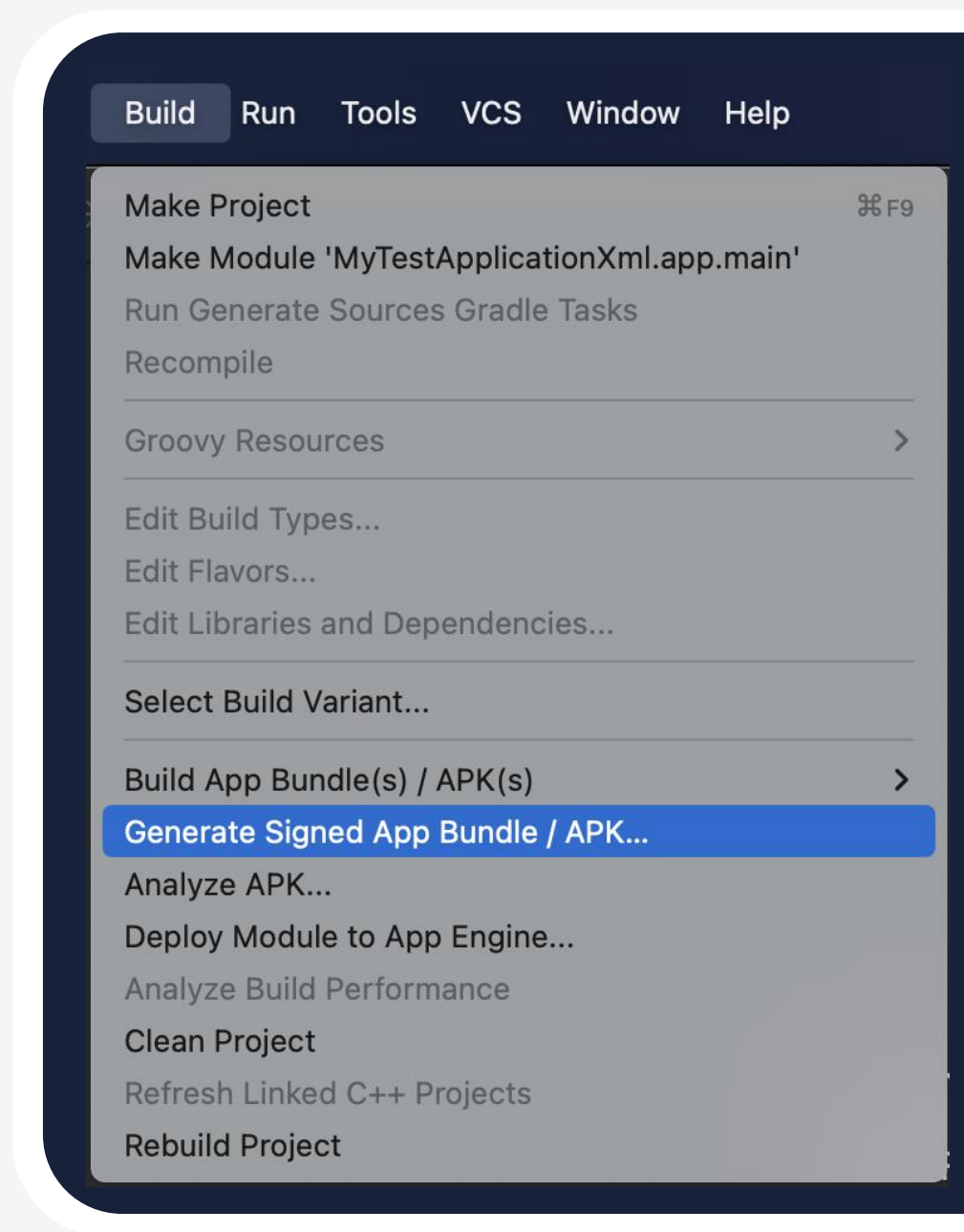
3. Объединенный манифест:

```
<activity-alias android:name="com.example.alias">
</activity-alias>
```

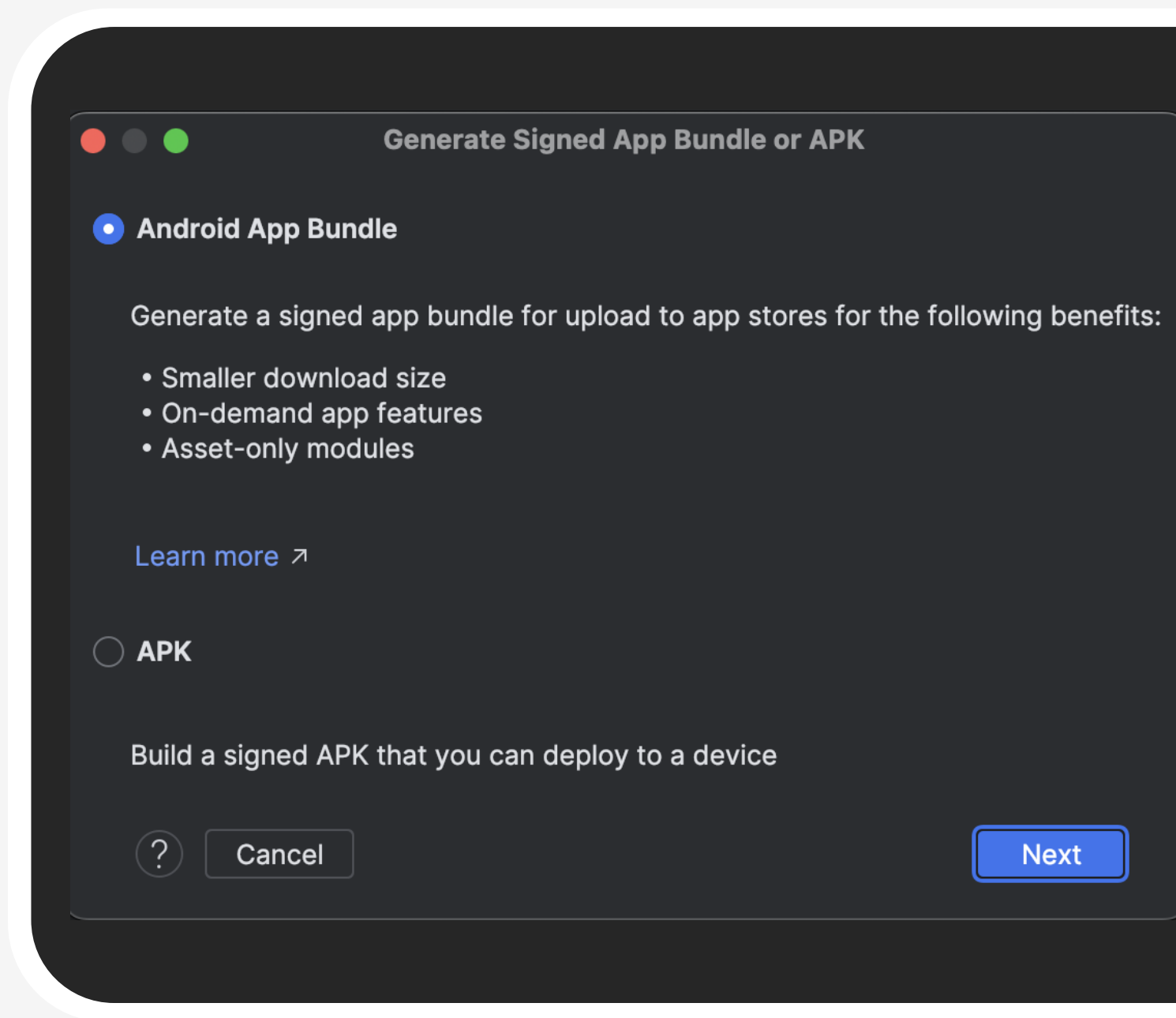
Подготовка к релизу

1. Настроить иконку приложения (атрибут **icon** и **roundIcon** в блоке **application**)
2. Подобрать оптимальный **applicationId**, сопровождающий приложения в дальнейшем (**applicationId** нельзя сменить).
3. Выключить отладочную информацию, включить оптимизацию (см. [слайд 8](#)).
4. Убрать опасные логи из приложения, либо обернуть в условие **if (BuildConfig.DEBUG)**.
5. Проверить **manifest** файл, что объявлены все нужные разрешения и удалить лишние **<uses-permission>**.
6. Обновить значения **versionCode** и **versionName**.
7. Добавить, при необходимости, пользовательское соглашение (**End-user license agreement**).
8. Подготовить описание, видео и скриншоты приложения при публикации.
9. Подписать приложение.

Подпись приложения

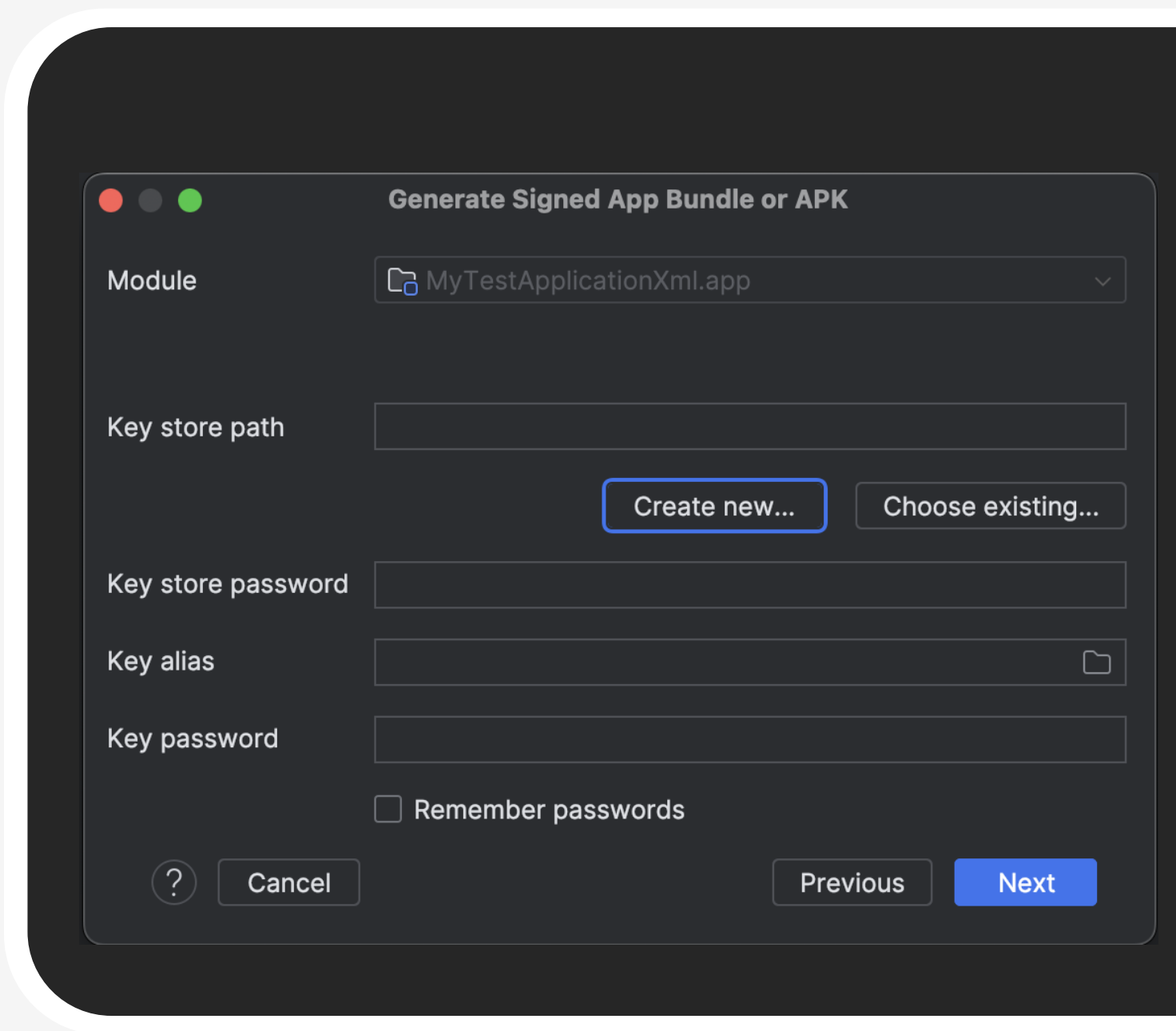


Подпись приложения



Подпись приложения

Подписать aab можно либо с помощью существующего ключа, либо создать новый и подписать им.



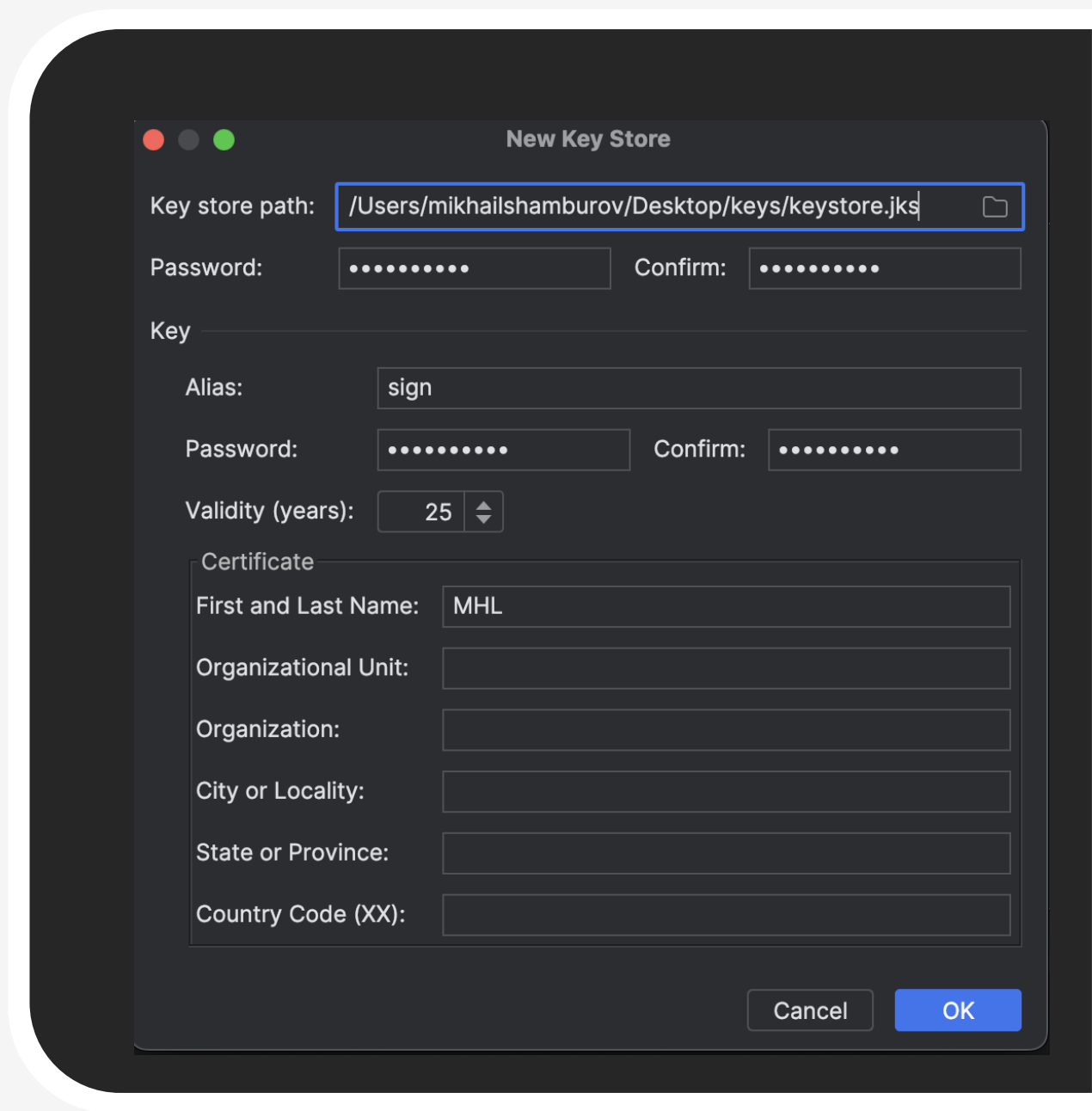
Подпись приложения

.jks (Java key store) - бинарный файл хранящий приватные ключи и сертификаты.

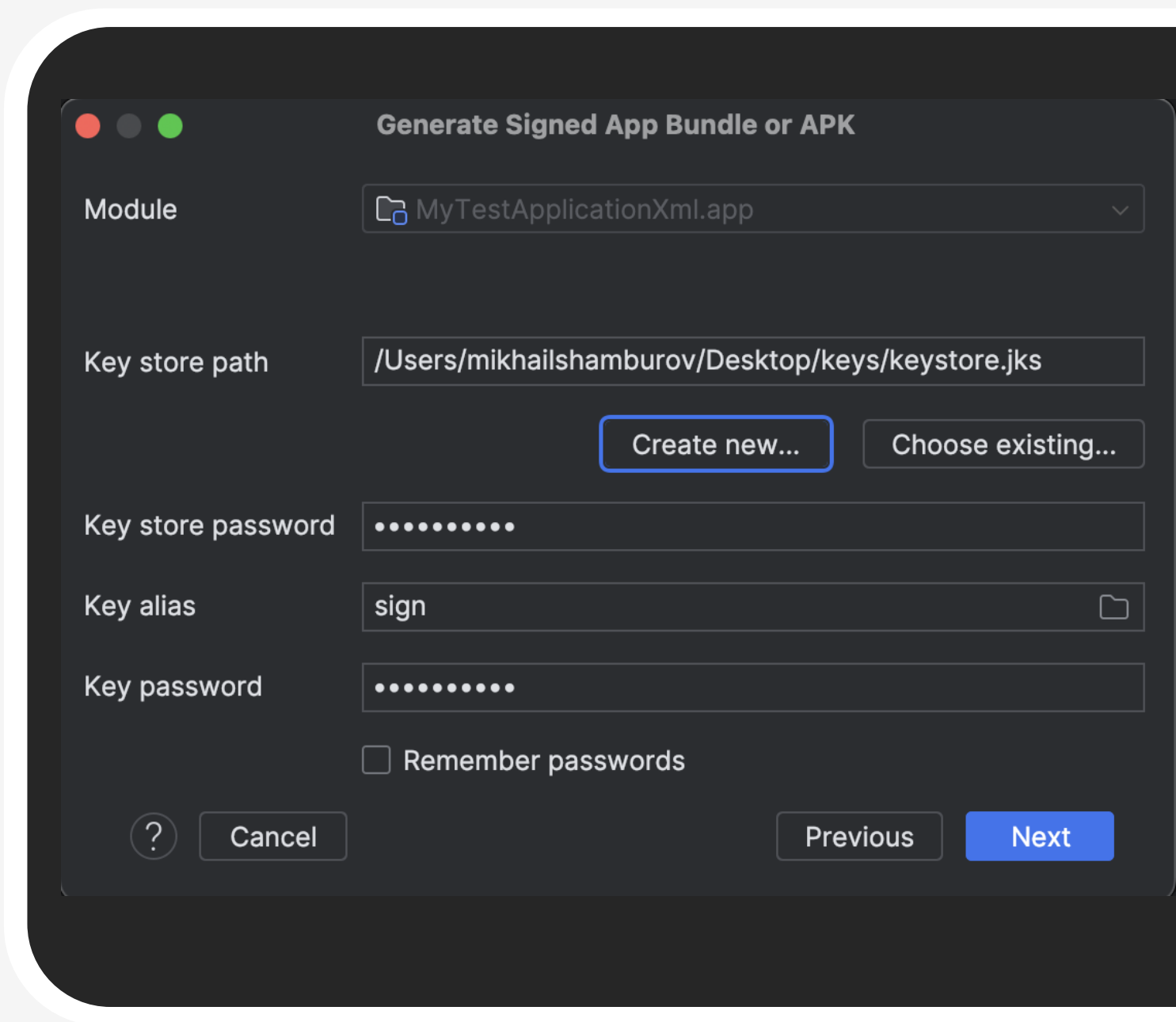
Само хранилище и каждый ключ могут быть защищены собственным паролем.

Дополнительно, нужно указать срок валидности ключа в течении которого им можно будет подписывать приложение (Минимальное значение 25 лет).

И указать информацию о владельце приложения в разделе сертификата. Эта информация не отображается напрямую в приложении, но включается в информацию сертификата, как часть арк файла.



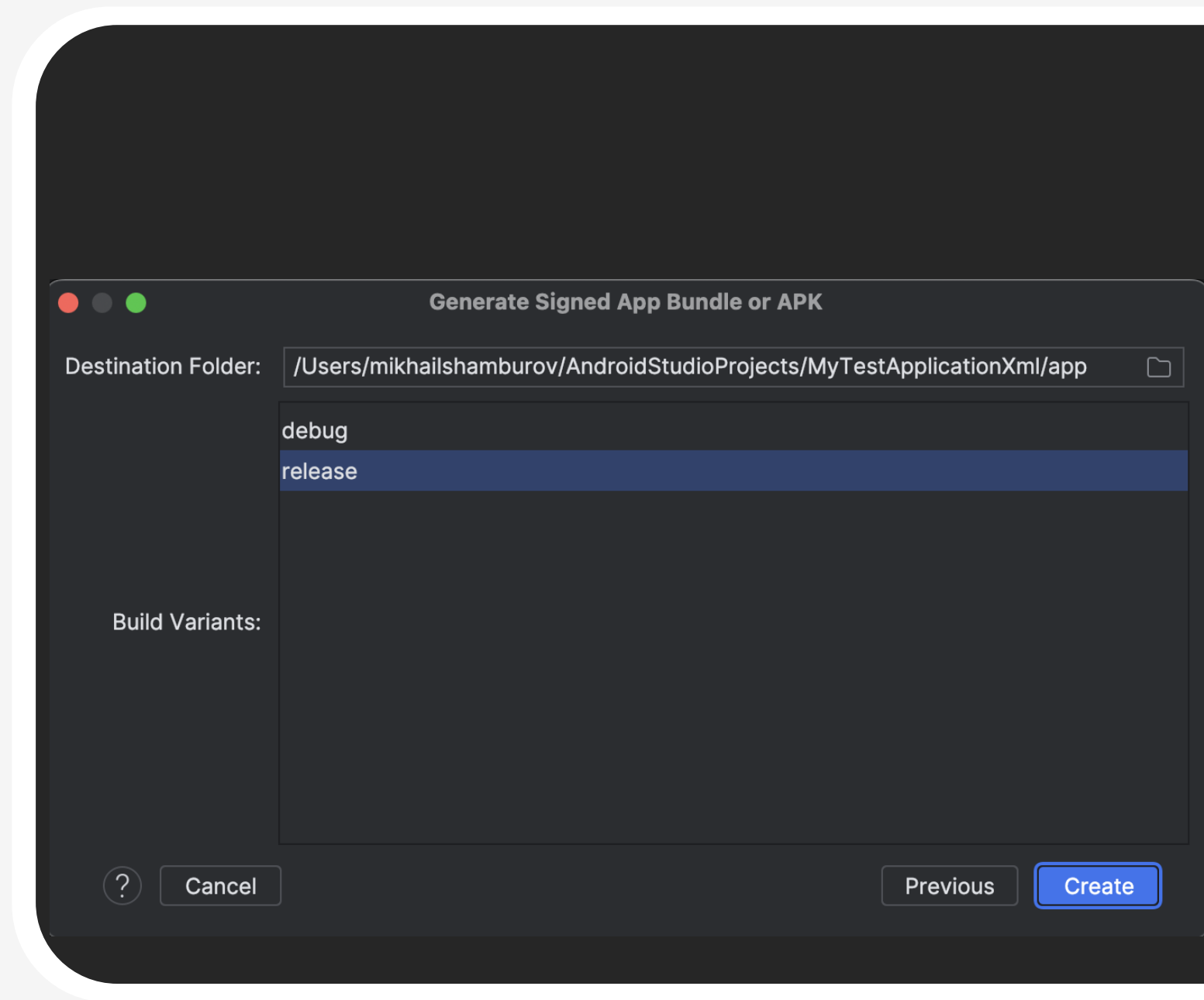
Подпись приложения



The image shows a screenshot of the 'Generate Signed App Bundle or APK' dialog box in Android Studio. The dialog is dark-themed and contains the following fields and controls:

- Module:** A dropdown menu showing 'MyTestApplicationXml.app'.
- Key store path:** A text field containing '/Users/mikhailshamburov/Desktop/keys/keystore.jks'.
- Key store password:** A password field with 10 dots.
- Key alias:** A text field containing 'sign'.
- Key password:** A password field with 10 dots.
- Remember passwords:** An unchecked checkbox.
- Buttons:** 'Create new...' (highlighted with a blue border), 'Choose existing...', '?', 'Cancel', 'Previous', and 'Next' (highlighted in blue).

Подпись приложения

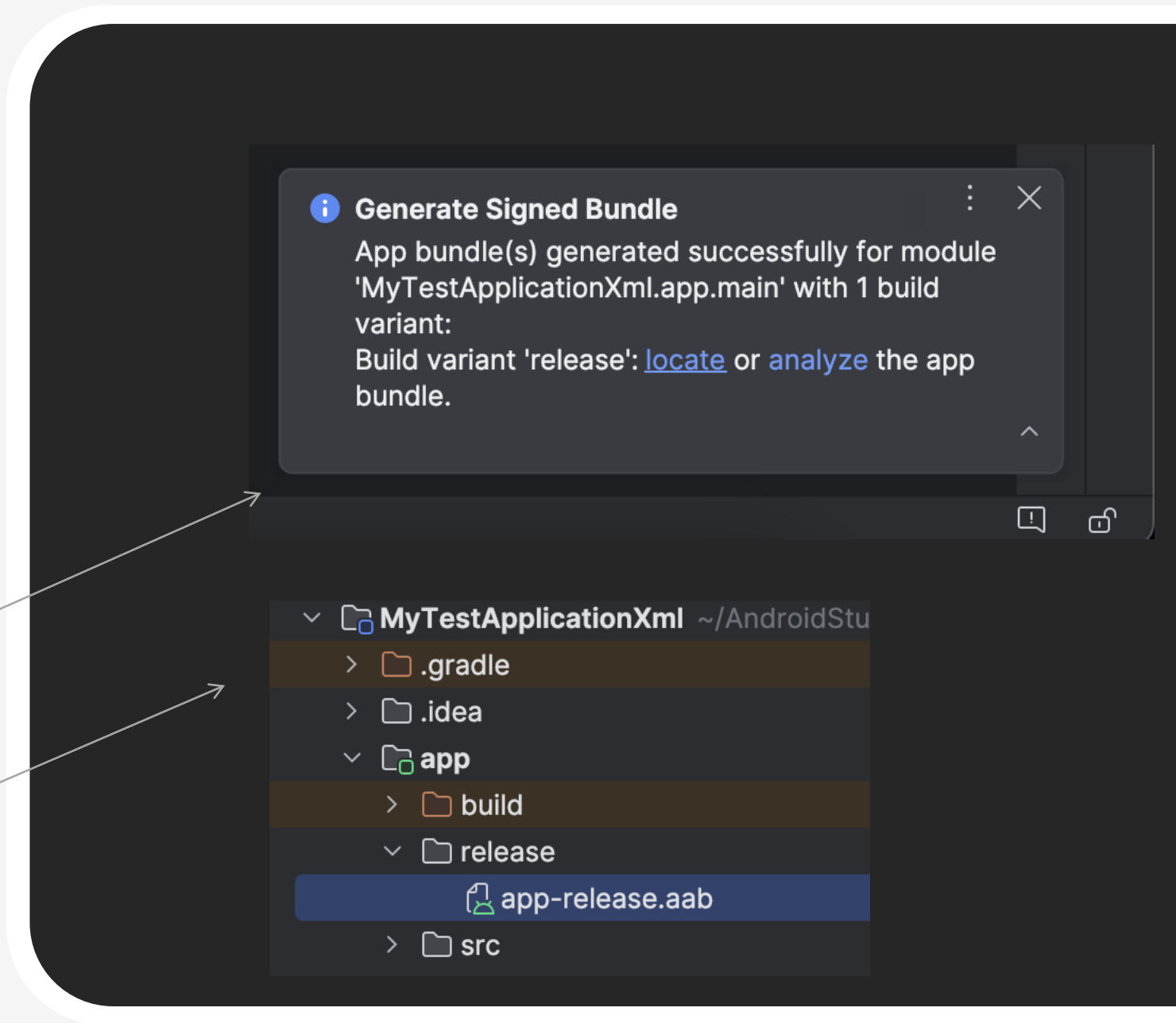


Подпись приложения

После выполнения всех этапов подписи приложения сгенерируется подписанный aab файл.

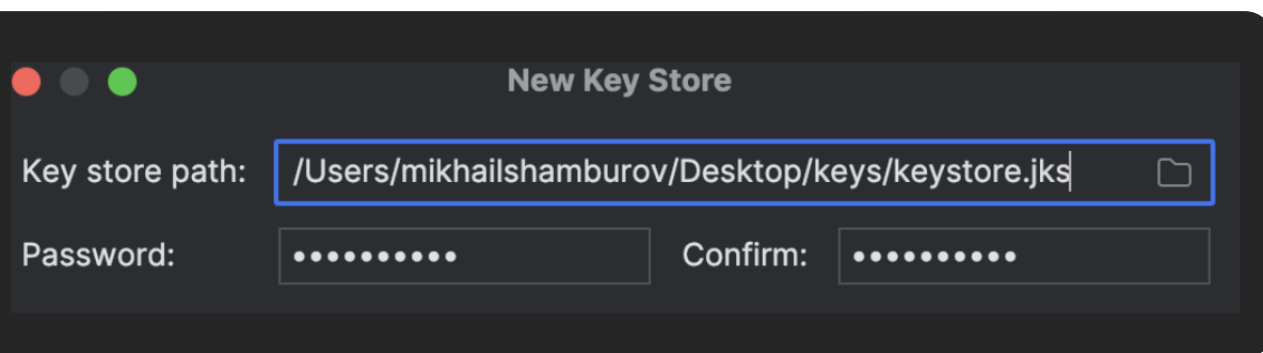
Расположение файла можно узнать кликнув на `locate` в сплывающем окне.

Либо найти в папке `app/release/`



Подпись приложения

Мы так же получили сгенерированный ключ, которым и был подписан наш aab файл. Находится он по пути, который мы указывали при создании ключа.



В Google Play (GP) этот ключ будет являться ключем загрузки.

Впоследствии, когда приложение будет загружено в стор, то стор сформирует различные .apk файлы и подпишет их ключом подписи приложения. Ключ подписи сформируется на стороне GP, либо вы сами его предоставите. (Процесс может отличаться в разныхсторах).

Арк файлы подписанные ключом подписи будут отдаваться пользователям на скачивание и установку.

Два разных ключа подписи используется для повышения безопасности.

Потеряв ключ загрузки, его можно восстановить проще, чем ключ подписи, потому что ключ загрузки отвечает только за подпись aab файла при загрузке в стор.

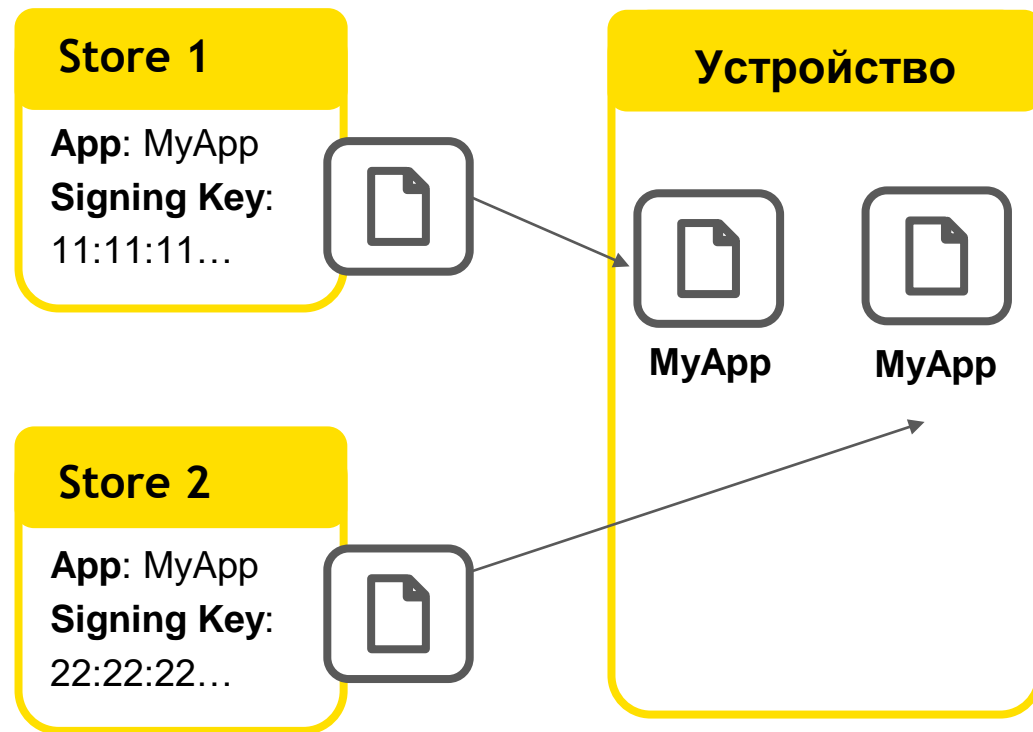
Следует **хранить** сгенерированные файлы загрузки и подписи **в безопасном месте**, чтобы никто другой не смог воспользоваться вашими ключами.

Подпись приложения

Нужно быть осторожным с ключом подписи приложения.

Если в один стор было загружено приложение с одним ключом подписи, а в другой стор с другим ключом подписи, то, при установке из этих 2-х источников, будет установлено 2 копии вашего приложения.

Поэтому, чтобы поддерживать совместимость между сторками, приложение нужно подписывать одним и тем же ключом подписи.





True Engineering

630128, г. Новосибирск,
ул. Кутателадзе, 4г

(383) 363-33-51, 363-33-50
info@trueengineering.ru
trueengineering.ru

Новосибирский
Государственный
Университет