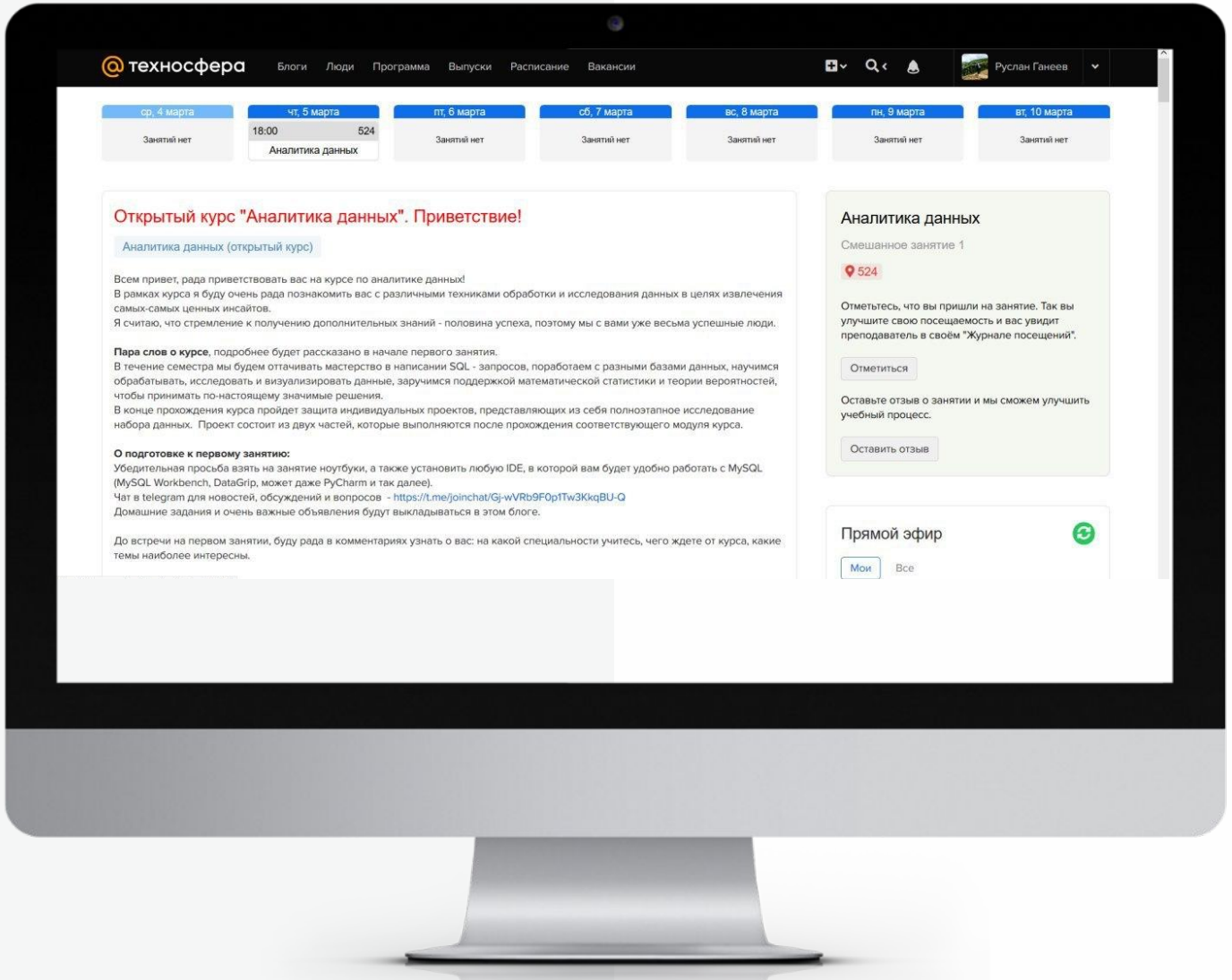
x mail.ru group

Основные компоненты приложения

Клещин Никита



Напоминание отметиться на портале

И еще раз

Что помним?

- Fragments
- Lifecycle
- States

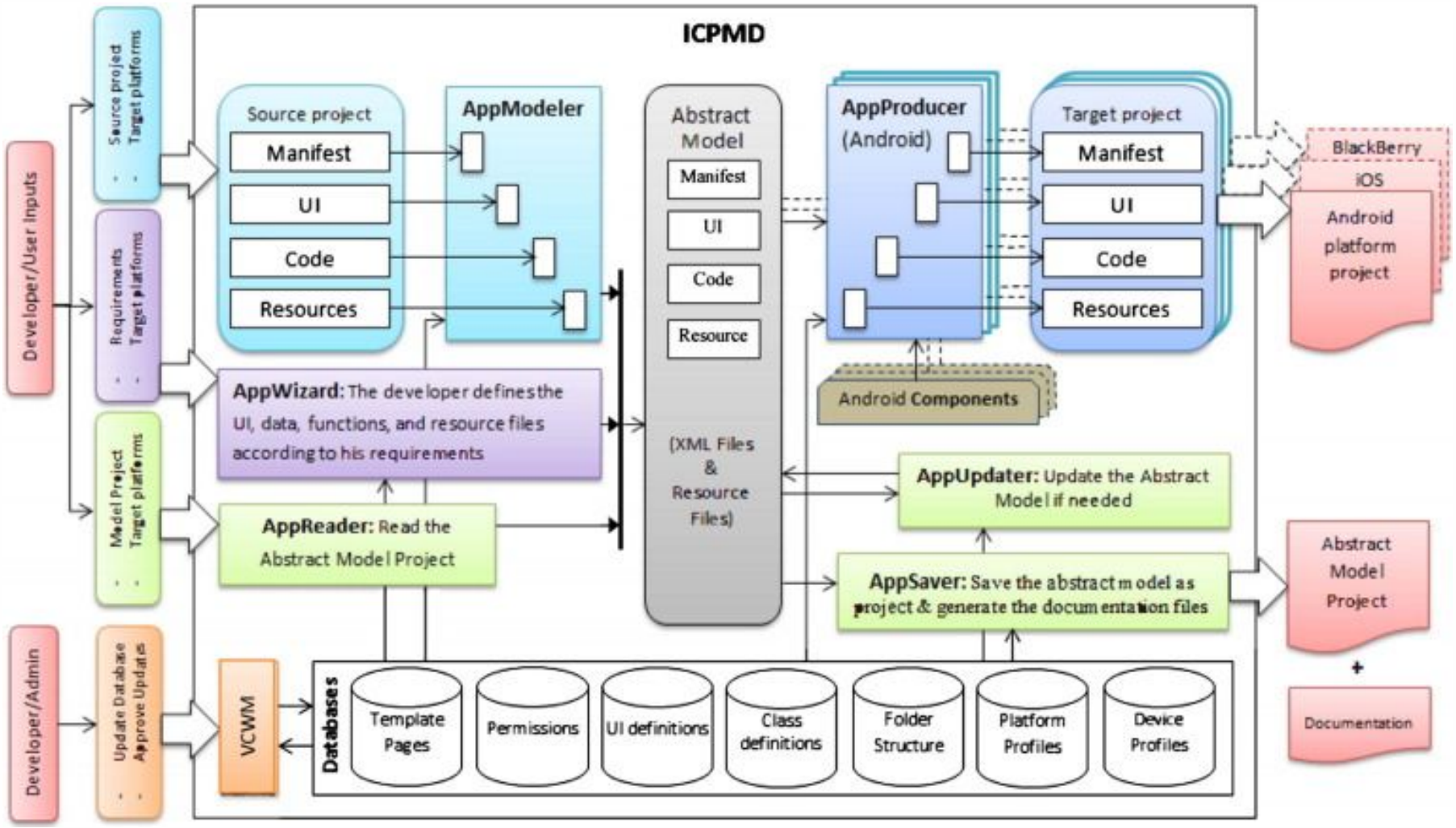
Что уже сделали?

- Как ДЗ?
- Как ТЗ?
- Сэмплы с ДЗ помогают?

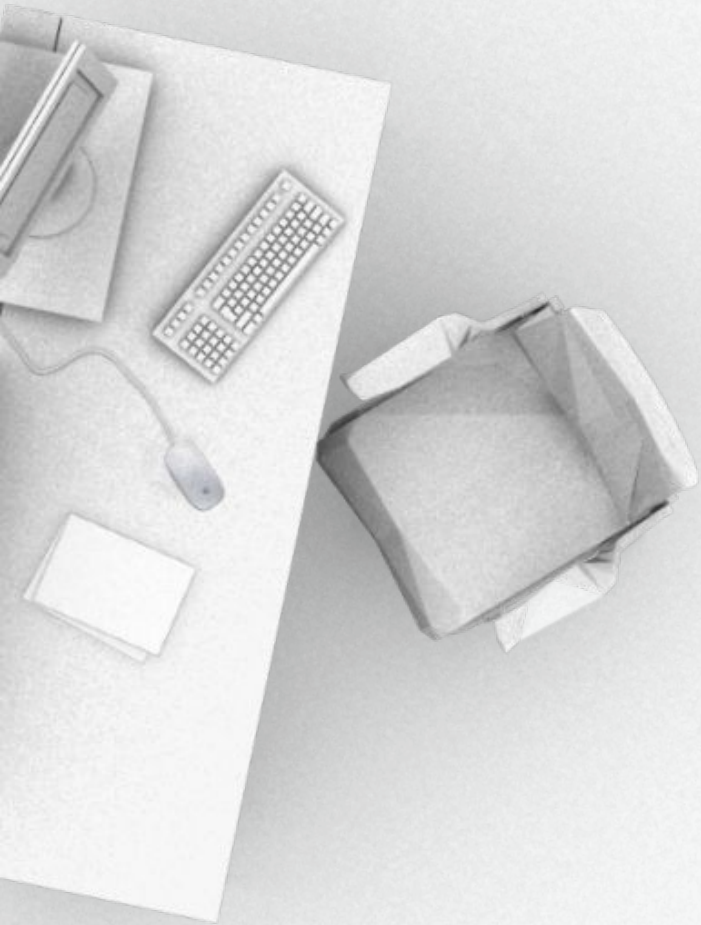
Содержание занятия

1. Приложение
2. Activity
3. Service
4. BroadcastReceiver
5. ContentProvider

Ну как, готовы запоминать схему?:)



Не, количество компонентов не изменилось

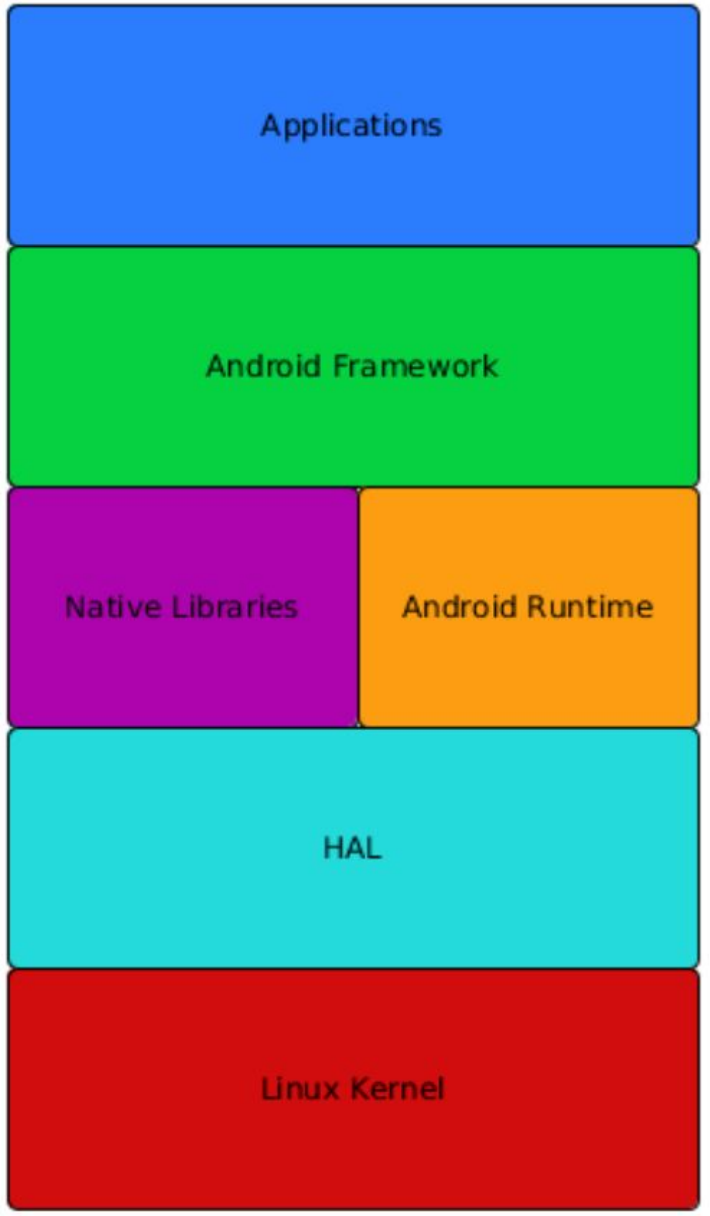


Приложение

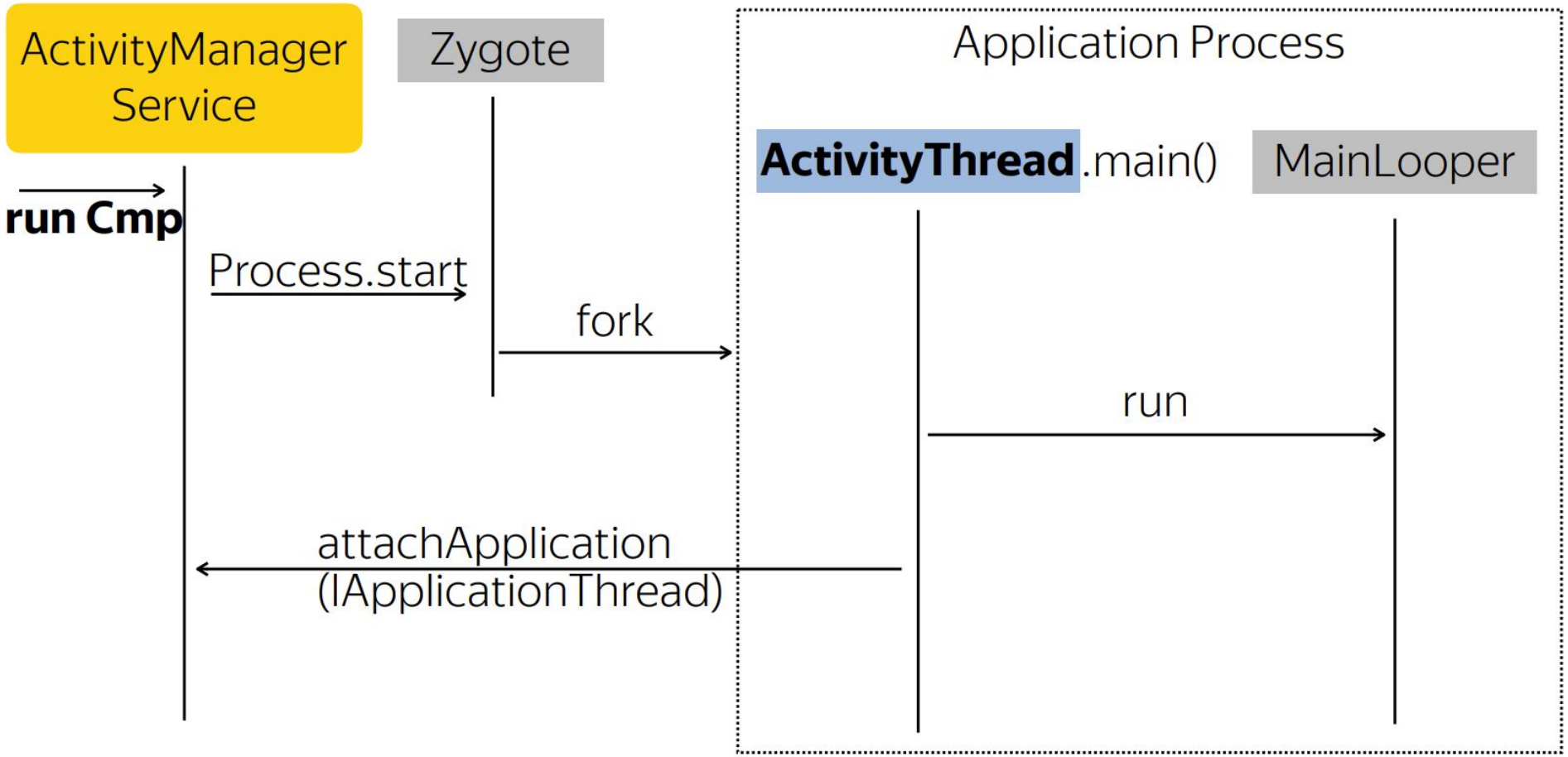
Application

Что это?

- ПО, которое выполняет определенные функции;
- Приложение на **Android** может выполнять работу в нескольких процессах;
- Процесс - отдельное, виртуальное адресное пространство со своими ресурсами и загруженными инструкциями;
- У каждого процесса есть **Application** - основной класс для поддержания глобального состояния приложения;
- Обычное приложение может сделать самоубийство (*Runtime.getRuntime().exit(0)*);
- Но не нельзя убить чужое приложение:)



Bind Application



Application

- Наследник **Context**;
- Есть свои методы, для “жизненного цикла”:
  - **attachBaseContext()** - присоединить базовый контекст
  - **onCreate()** - класс создан;
  - **onConfigurationChanged()** - изменена конфигурация приложения. Работает не так, как в Activity;
  - **onLowMemory()** - система уведомляет, что у нее мало памяти;
  - **onTrimMemory()** - намек на то, что надо освободить память;
  - **registerActivityLifecycleCallbacks()** - подписаться на изменение **Activity**
  - **onTerminate()** - только в эмуляторах!

```java
public class ConnectivityApplication extends Application {
    @Override
    protected void attachBaseContext(Context base) {
        final Context wrapped = new MyContextWrapper(base);

        super.attachBaseContext(wrapped);
    }

    @Override
    public void onCreate() {
        super.onCreate();

        ServiceLocator.initialize(this);
    }
}
```

```kotlin
class ConnectivityApplication : Application() {
    override fun attachBaseContext(base: Context) {
        final Context wrapped = new MyContextWrapper(base);

        super.attachBaseContext(wrapped)
    }

    override fun onCreate() {
        super.onCreate()

        ServiceLocator.initialize(this)
    }
}
```

Manifest

Содержит важную информацию для системы:

- Задает имя пакета;
- Ограничения по API;
- Ограничения по устройствам;
- Описывает компоненты приложения;
- Описывает, какие разрешения нужны приложению;
- Объявляет разрешения для своих компонентов.

```
<manifest
    xmlns:android="http://schemas.android.com/apk/res/android"
    package="ru.hse.lection04"
    >


    <uses-permission ...
    ...


    <application
        android:name=".ConnectivityApplication"
        android:allowBackup="true"
        android:label="@string/app_name"
        android:icon="@mipmap/ic_launcher"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:theme="@style/AppTheme"
        >

        <activity ...

        <service ...

        <provider ...

        <receiver ...

        ...
    </application>
</manifest>
```

Структура

```
<manifest>
    <uses-permission />
    <permission />
    <permission-tree />
    <permission-group />
    <instrumentation />
    <uses-sdk />
    <uses-configuration />
    <uses-feature />
    <supports-screens />
    <compatible-screens />
    <supports-gl-texture />

    <application> ... </application>
</manifest>
```

```
<application>
    <activity>
        <intent-filter>
            <action />
            <category />
            <data />
        </intent-filter>
        <meta-data />
    </activity>

    <activity-alias>
        <intent-filter> . . . </intent-filter>
        <meta-data />
    </activity-alias>

    <service>
        <intent-filter> . . . </intent-filter>
        <meta-data/>
    </service>

    <receiver>
        <intent-filter> . . . </intent-filter>
        <meta-data />
    </receiver>

    <provider>
        <grant-uri-permission />
        <meta-data />
        <path-permission />
    </provider>

    <uses-library />

</application>
```

Ограничения

```
android {
    compileSdkVersion 30
    buildToolsVersion "30.0.2"

    defaultConfig {
        applicationId "ru.hse.lection04"
        minSdkVersion 23
        targetSdkVersion 30
        versionCode 1
        versionName "1.0"

        ...
    }

    ...
}
```

```
<manifest
    package="string"
    android:sharedUserId="string"
    android:sharedUserLabel="string resource"
    android:versionCode="integer"
    android:versionName="string"
    android:installLocation=["auto" | "internalOnly" | "preferExternal"]
    >

<uses-sdk
    android:minSdkVersion="integer"
    android:targetSdkVersion="integer"
    android:maxSdkVersion="integer"
    />

<uses-configuration
    android:reqFiveWayNav=["true" | "false"]
    android:reqHardKeyboard=["true" | "false"]
    android:reqKeyboardType=["undefined" | "nokeys" | "qwerty" | "twelvekey"]
    android:reqNavigation=["undefined" | "nonav" | "dpad" | "trackball" | "wheel"]
    android:reqTouchScreen=["undefined" | "notouch" | "stylus" | "finger"]
    />

<uses-feature
    android:name="string"
    android:required=["true" | "false"]
    android:glEsVersion="integer"
    />

<supports-screens
    android:resizeable=["true"| "false"]
    android:smallScreens=["true" | "false"]
    android:normalScreens=["true" | "false"]
    android:largeScreens=["true" | "false"]
    android:xlargeScreens=["true" | "false"]
    android:anyDensity=["true" | "false"]
    android:requiresSmallestWidthDp="integer"
    android:compatibleWidthLimitDp="integer"
    android:largestWidthLimitDp="integer"
/>
```

Разрешения (permission)

- **uses-permission** - какое разрешение необходимо;
- **permission** - задать новое разрешение;
- **permission-group** - связывает permission логически в одну группу
- **permission-tree** - Объявление пространство имен разрешений;

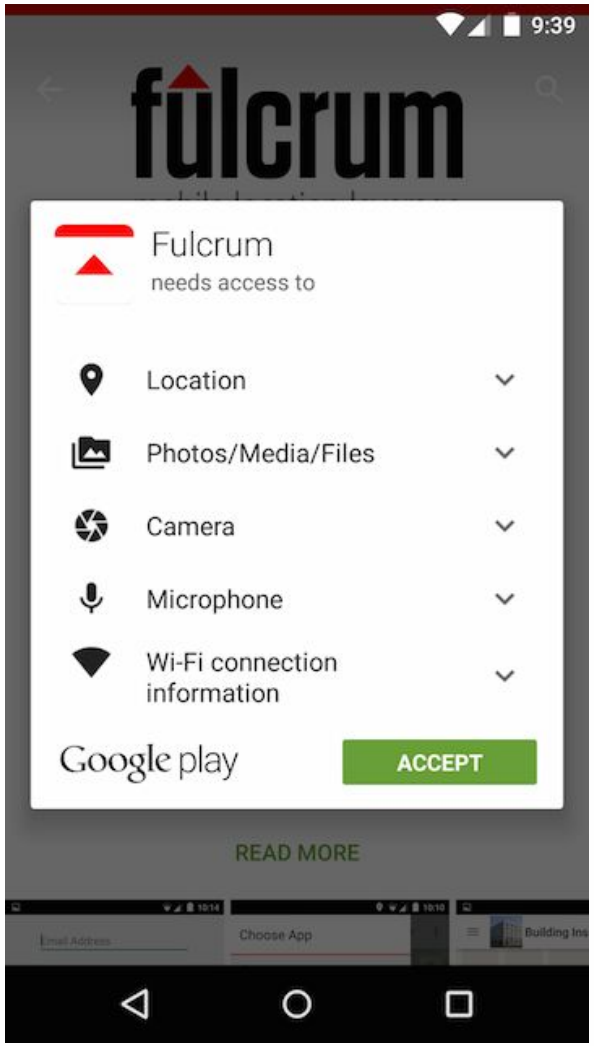
Начиная с **Android API 23** если **permission** имеет категорию **dangerous**, то для его получения надо использовать механику “**Runtime Permission**”

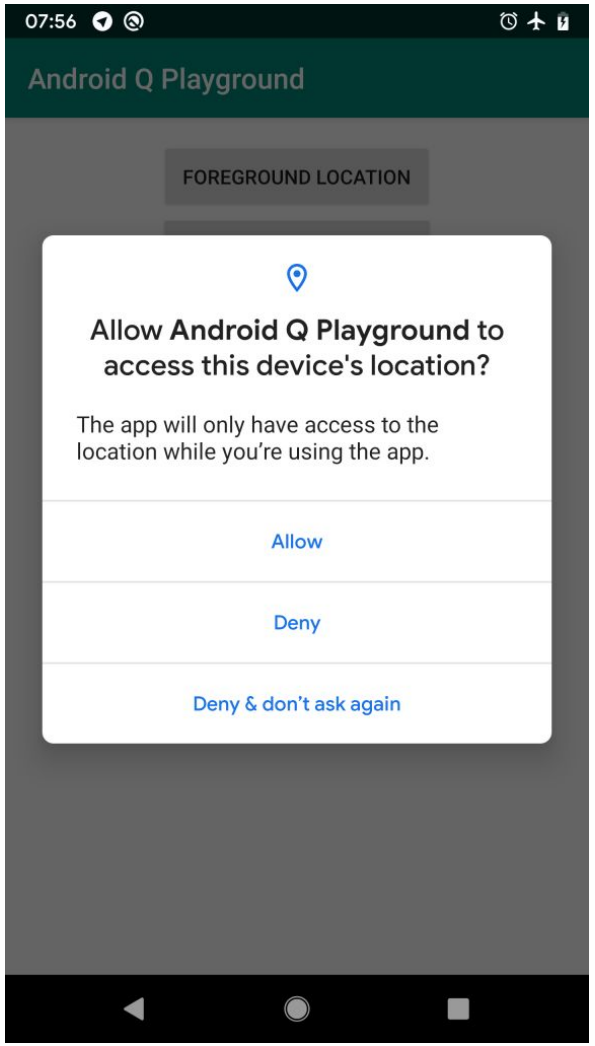
```
<permission
    android:name="com.example.project.DEBIT_ACCT"

    android:permissionGroup="string"
    android:description="string resource"
    android:icon="drawable resource"
    android:label="string resource"
    android:protectionLevel=["normal" | "dangerous" | "signature"]
    />

<uses-permission
    android:name="com.example.project.DEBIT_ACCT"
    android:maxSdkVersion="19"
    />

<permission-group
    android:name="string"
    android:description="string resource"
    android:icon="drawable resource"
    android:label="string resource"
    />

<permission-tree
    android:name="string"
    android:icon="drawable resource"
    android:label="string resource"
    />

...
<application . . .>
    <activity
        android:name=".FreneticActivity"
        android:permission="com.example.project.DEBIT_ACCT"
        ...
</application>
```

API 23 - введение Runtime Permission

Это означает что теперь опасные разрешения автоматические не предоставляются системой при установке. Разработчик должен будет предусмотреть механизм их запроса в коде. Желательно - прямо перед использованием.

При этом, пользователь может отозвать предоставленное разрешение через настройки приложения. Надо учитывать этот момент, когда происходит возврат в приложение





<application>

Внутри данного тэга описываем компоненты.

meta-data - Key-Value элемент. Данные сюда могут быть вписаны во время компиляции, а потом мы их можем достать при работе приложения через **PackageItemInfo.metaData**.

uses-library - Задает общую библиотеку, с которой должно быть связано приложение. Это аффектит на установку.

Про тэги компонентов - разберем позже.

```
<application
    android:name=".ConnectivityApplication"
    android:allowBackup="true"
    android:label="@string/app_name"
    android:icon="@mipmap/ic_launcher"
    android:roundIcon="@mipmap/ic_launcher_round"
    android:theme="@style/AppTheme"

    ...
    >

    <activity ...

    <activity-alias ...

    <service ...

    <provider ...

    <receiver ...

    <meta-data ...

    <uses-library ...

</application>
```

<intent-filter>

По сути - это триггер на определенные намерения.

Компонент может иметь несколько **intent-filter**, с разным набором параметров.

Прикрепляется к **Activity**, **Service** и **BroadcastReceiver**.

action - типо действия

category - дополнительная информация об исполнении

data - данные намерения. **Uri** или **MimeType**

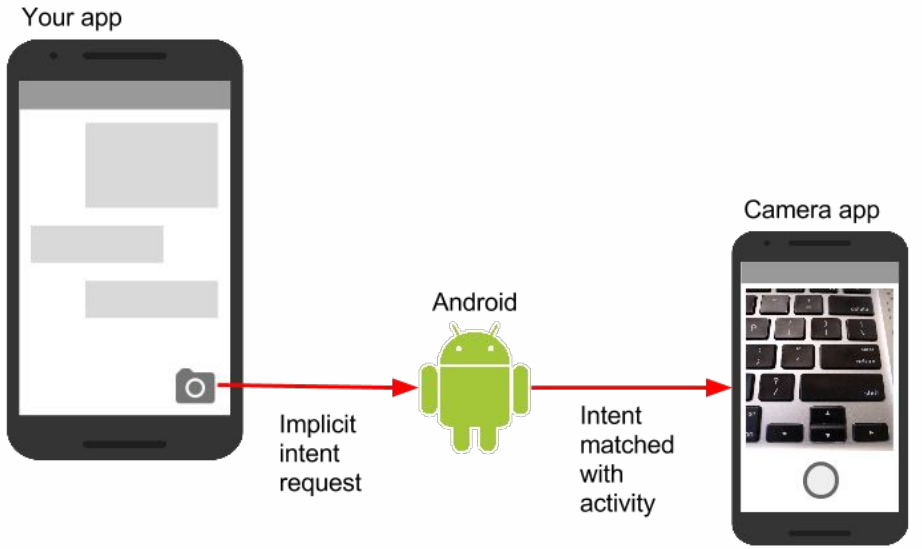
```
<intent-filter
    android:icon="drawable resource"
    android:label="string resource"
    android:priority="integer"
    >
        // action - обязательно
        <action android:name="android.intent.action.MAIN" />

        <category android:name="android.intent.category.LAUNCHER" />

        <data
            android:scheme="string"
            android:host="string"
            android:port="string"
            android:path="string"
            android:pathPattern="string"
            android:pathPrefix="string"
            android:mimeType="string"
            />
</intent-filter>
```

Intent

Нужен для запуска **Activity**, **Service** или **BroadcastReceiver**

Explicit intent - Явное “намерение”. Указываем класс, к которому хотим обратиться.

Implicit intent - Неявное “намерение”. Указываем данные, а далее система собирает список обработчиков




```
1.  new Intent(Intent.ACTION_VIEW, Uri.parse(url));
2.  new Intent(FirstActivity.this, SecondActivity.class);
3.  new Intent(Intent.ACTION_SEND)
        .setType("image/png")
        .putExtra(Intent.EXTRA_STREAM, screenshotUri);
4.  new Intent(Intent.ACTION_CALL)
        .setData(Uri.parse("tel:555-555-5555"));
```

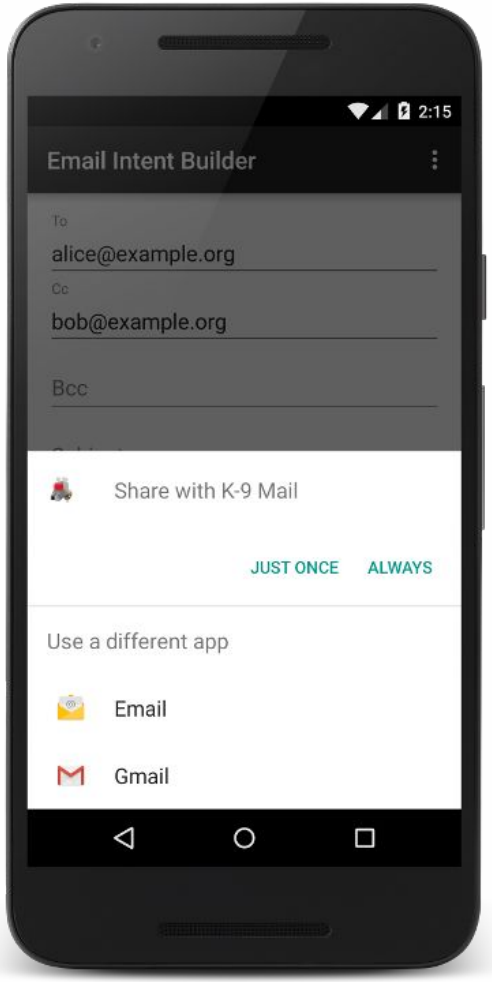
Конструирование намерения

У класса **Intent** большая вариативность конструкторов. Но эти же параметры можно сетить отдельно:

- **setAction()** - установить действие
- **addCategory()** - установить категорию
- **setData()** - установить Uri-данные
- **setType()** - установить тип данных

Также, вспомогательные методы

- **putExtras()** - положить данные в виде **Bundle**
- **setFlags()** - флаги для коррекции запуска (обычно у **Activity**)
- **setPackage()** - фильтр по имени пакета приложения
- и т.п. ...



Activity

Опять он!

Что это?

Один из базовых компонентов **Android**

Отвечает за интерфейс приложения и взаимодействие с ним.

Есть свой жизненный цикл и состояния.

Чтобы запустить **Activity**, требуется использовать метод из **Context**:

- **Context.startActivity()** - простой запуск
- **Context.startActivityForResult()** - если ожидаем возврат

```java
public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        setContentView(R.layout.activity_main);
    }
}
```

```kotlin
class MainActivity : AppCompatActivity() {

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

        setContentView(R.layout.activity_main)
    }
}
```

Указание в Manifest

<activity> - тэг для описания **Activity** в манифесте:

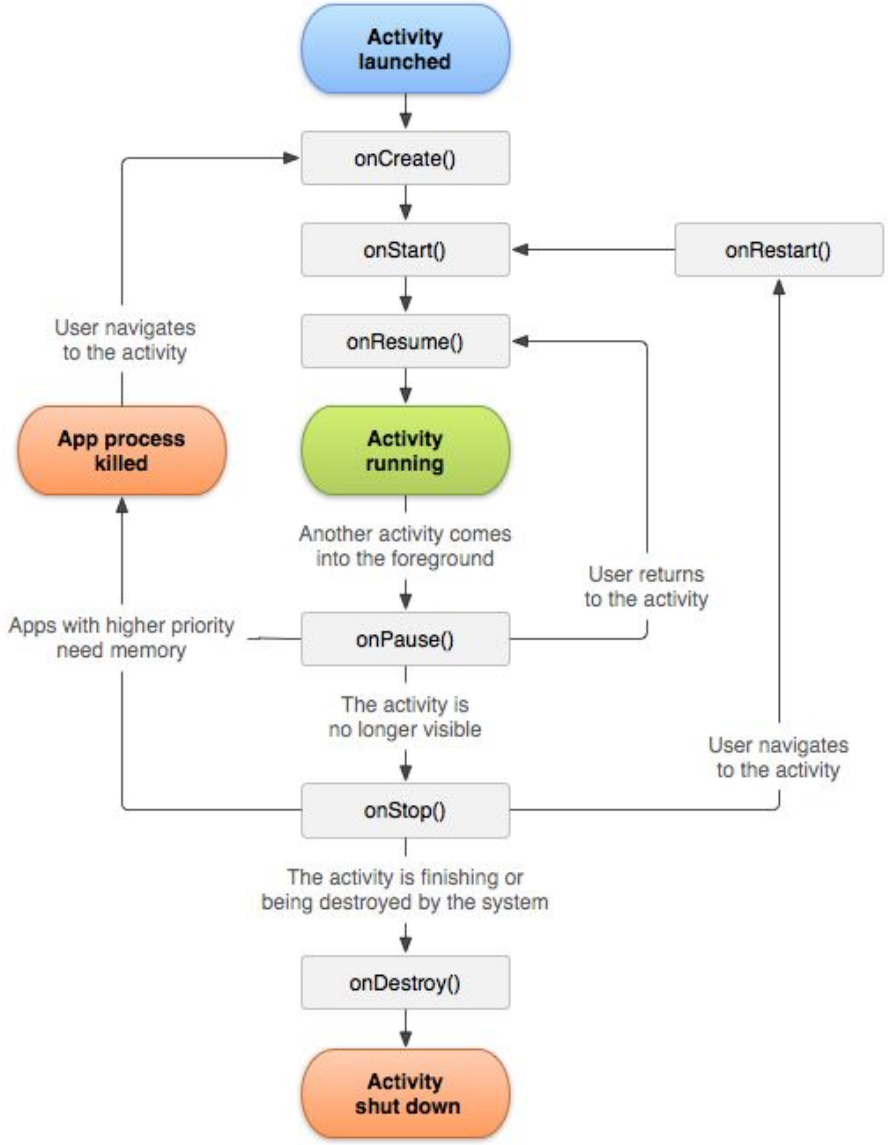
- **name** - путь до класса **Activity**
- **theme** - собственная тема, если она должна отличаться от основной темы
- и еще много других параметров...

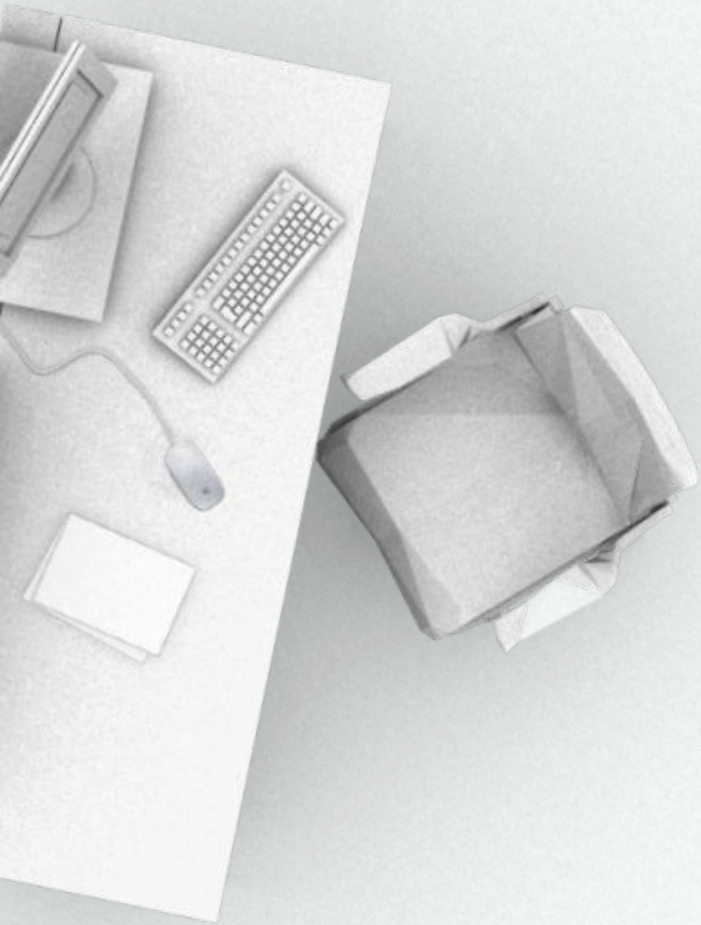
<activity-alias> - линк на **<activity>**. Единственная его функция - это “красиво” разделить саму **Activity** и точку входа в него.

```
<activity
    android:name=".activities.SplashActivity"
    android:screenOrientation="sensorPortrait"
    android:theme="@style/AppTheme.Splash"
    />


<activity-alias
    android:name="Launcher"
    android:targetActivity=".activities.SplashActivity"
    >
    <intent-filter>
        <action android:name="android.intent.action.VIEW" />

        <category android:name="android.intent.category.DEFAULT" />
        <category android:name="android.intent.category.BROWSABLE" />

        <data android:scheme="example" />
    </intent-filter>

    <intent-filter>
        <action android:name="com.example.MESSAGE" />

        <category android:name="android.intent.category.DEFAULT" />
    </intent-filter>

    <intent-filter>
        <action android:name="android.intent.action.MAIN" />

        <category android:name="android.intent.category.DEFAULT" />
        <category android:name="android.intent.category.LAUNCHER" />
        <category
android:name="miui.intent.category.SYSAPP_RECOMMEND" />
    </intent-filter>
</activity-alias>
```

Жизненный цикл:)

- **onCreate(Bundle savedState)** - Инициализируемся
- **onStart()** - Подготовка к отображению
- **onResume()** - Можно взаимодействовать
- **... PROFIT!**
- **onSaveInstanceState(Bundle outState)** - Сохраняемся
- **onPause()** - Потеряли фокус
- **onStop()** - Невидимы для пользователя
- **onDestroy()** - Пока-пока!





Service

Фоновая работа и не только!

Что это?

Обеспечивает работоспособность и “видимость” для системы вашего приложения, даже после уничтожения визуальной части.

В основном используется для “длительной” работы, когда ее надо будет продолжить в фоновом режиме.

В случае отсутствия визуальной части приложения, активный **Service** повышает приоритет вашего приложения, чтобы его не убила система.

Service выполняет работу в главном потоке. А его наследник, **IntentService**, в асинхронном потоке.

```java
public class ConnectivityService extends Service {
    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
        return super.onStartCommand(intent, flags, startId);
    }

    @Nullable
    @Override
    public IBinder onBind(Intent intent) {
        return null;
    }
}
```

```kotlin
class ConnectivityService : Service() {
    override fun onStartCommand(intent: Intent?, flags: Int,
startId: Int): Int {
        return super.onStartCommand(intent, flags, startId)
    }

    override fun onBind(intent: Intent): IBinder? {
        return null
    }

    ...
}
```

Указание в Manifest

<service> - тэг для описания **Service** в манифесте:

- **name** - путь до класса Service
- **process** - если указан, то сервис будет запущен в другом процессе
- **exported** - можно использовать этот компонент другим приложениям
- и многое другое ...

Указание **<intent-filter>** в манифесте, позволяет запустить сервис “не явно”, а значит и сделать доступ к сервису извне проще.

```
<service
    android:name=".businesslayer.slices.AsyncSliceService"
    android:permission="android.permission.BIND_JOB_SERVICE"
    android:exported="false"
    android:process=":processname"
    >
    <intent-filter>
        <action android:name="com.example.ASYNC_SLICE"/>
    </intent-filter>
</service>
```

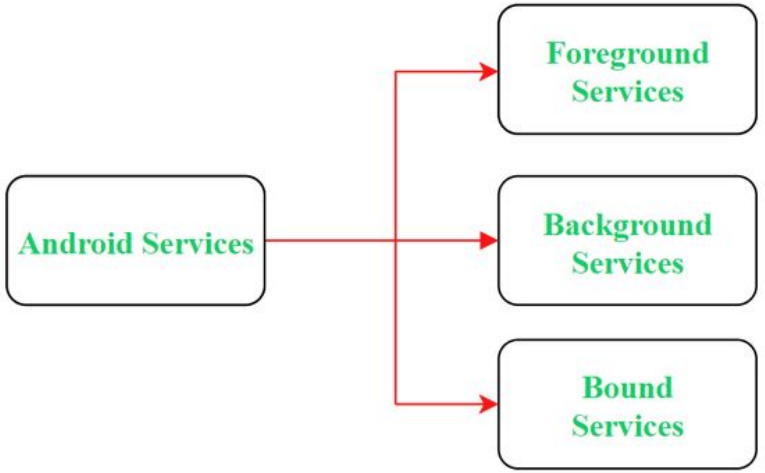
Сервисы делят на 3 вида

Service - один из тех компонентов, работу которого ограничивают. Поэтому следует учитывать, как надо стартовать сервис.

Background Service - не видим пользователю. При закрытии всех **Activity** этот сервис будет уничтожен. Стартуется при помощи **Context.startService(Intent)**. Стопить работу надо самостоятельно.

Foreground Service - видим пользователю при помощи уведомления в статусбаре. Из-за того что видим, система позволяет ему работать без визуальной части. Стартуется при помощи **ContextCompat.startForegroundService(Intent)** и установить уведомление. Стопить работу надо самостоятельно.

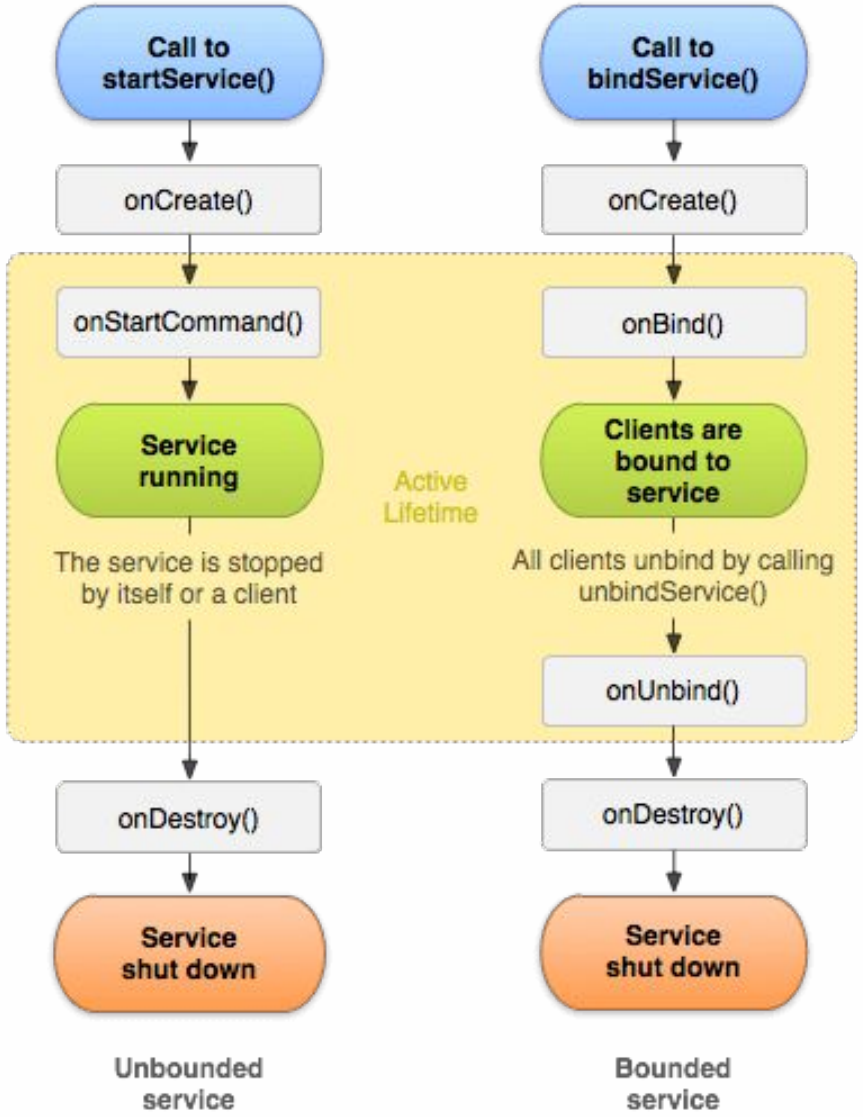
Bound Service - не видим пользователю. Стартуется сервис при помощи **Context.bindService()**. Сервис живет до тех пор, пока у него есть “подписчики”.



Жизненный цикл

В зависимости от вида старта, цикл немного отличается:

- **onCreate()** - инициализация сервиса;
- **onStartCommand()** - в сервис пришел **Intent**. Не обязательно первый - ими можно не только стартовать сервисы, но и взаимодействовать с ними;
- **onBind()** - отдать канал связи **IBinder**. Если привязывается компонент из другого процесса - надо использовать **AIDL** интерфейс.
- **onUnbind()** - от сервиса отвязался компонент.
- **onDestroy()** - уничтожение



Немного про onStartCommand()

Этот метод возвращает один из следующих флагов.

- **START_NOT_STICKY** - если система убьет сервис, она его не перезапустит;
- **START_STICKY / START_STICKY_COMPATIBILITY** - если система убивает рабочий сервис, она его оживит, но не передаст в него **Intent**;
- **START_REDELIVER_INTENT** - если система убивает рабочий сервис, она его оживит, и вернет в него **Intent**, после которого не было завершения сервиса.

Старт Foreground Service

```java
public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        setContentView(R.layout.activity_main);

        final Intent intent = ConnectivityService.newInstance(this);
        ContextCompat.startForegroundService(this, intent);
    }
}
```

```java
public class ConnectivityService extends Service {
    @Override
    public void onCreate() {
        super.onCreate();

        final Notification notification = buildNotification();
        startForeground(FOREGROUND_ID, notification);
    }
}
```

```kotlin
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        setContentView(R.layout.activity_main)

        val intent = ConnectivityService.Companion.newInstance(this)
        ContextCompat.startForegroundService(this, intent)
    }
}
```

```kotlin
class ConnectivityService : Service() {
    override fun onCreate() {
        super.onCreate()

        val notification = buildNotification()
        startForeground(FOREGROUND_ID, notification)
    }
}
```

BroadcastReceiver

“Широковещательные сообщения”

Что это?

Базовый класс для получения “вещаний” от системы.

Вещания могут быть разные - от полной загрузки устройства, до информации о том, что произошел один тик на внутренних часах. Также можно создавать вещания самому.

Исполнение метода **onReceive** должно занимать как можно меньше времени, поэтому зачастую ресивер используется для получения сигнала, чтобы инициализировать исполнение в другом компоненте. Можно попробовать **goAsync()**.

Регистрацию разделяют на статическую (в **Manifest**) и динамическую (в коде).

```java
public class BootReceiver extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent) {
        // TODO
    }
}
```

```kotlin
class BootReceiver : BroadcastReceiver() {
    override fun onReceive(context: Context, intent: Intent) {
        // TODO
    }
}
```

Указание в Manifest

<receiver> - Указание **BroadcastReceiver** в манифесте - делает ему “статическую” регистрацию.

Такой вид регистрации позволяет подписаться только на те события, которые отрабатывают не часто (например - полная загрузка устройства).

- **name** - путь до класса **BroadcastReceiver**
- **<intent-filter>** - указать триггер, на который должен запуститься ресивер

Перед регистрированием ресивера - надо будет уточнить, какая регистрация для него доступна.

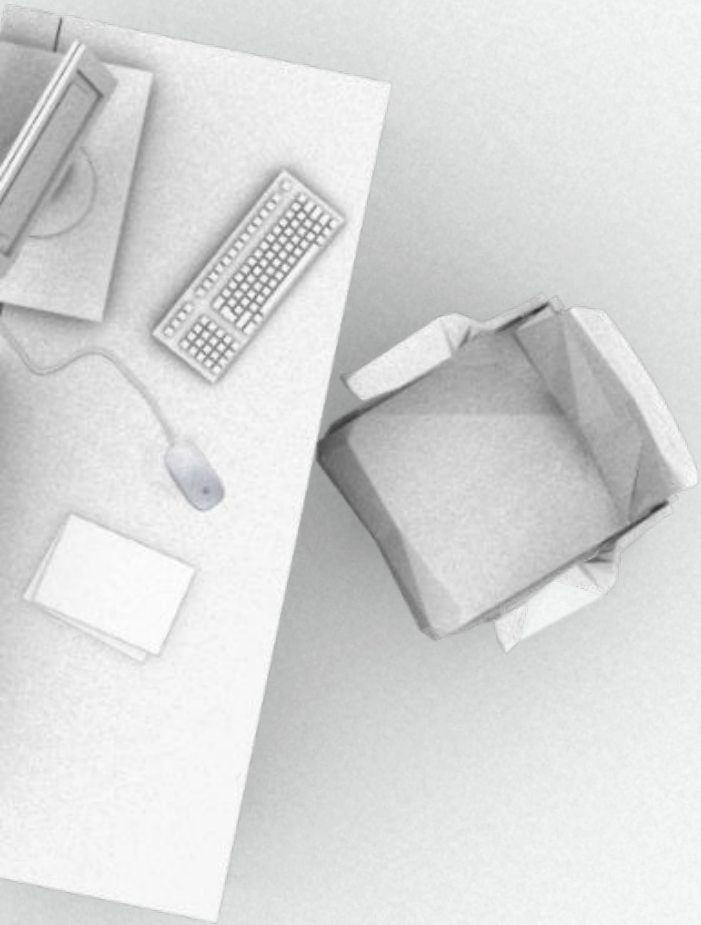
```
<receiver
    android:name=".presentationlayer.BootReceiver"
    android:enabled="true"
    >
    <intent-filter>
        <action android:name="android.intent.action.BOOT_COMPLETED" />
        <action android:name="android.intent.action.QUICKBOOT_POWERON" />
    </intent-filter>
</receiver>
```

Динамическая регистрация

Происходит в коде. Поэтому следует понимать когда его регистрировать и когда отписываться.

- **Context.registerReceiver()** - зарегистрировать ресивер;
- **Context.unregisterReceiver()** - отписать ресивер;
- **Context.sendOrderedBroadcast()** - отправить сообщение ресиверам поочередно, а не одновременно;
- **Context.send(remove)StickyBroadcast()** - устарел с 21 API. “Прилипшее” вещание, висит, и его данные получит любой новый подписчик.

Динамически можно зарегистрироваться почти на любые события, но тоже есть ограничения. Например, событие “Полная загрузка устройства” регистрировать в коде - смысла нет:)



ContentProvider

Обертка над данными

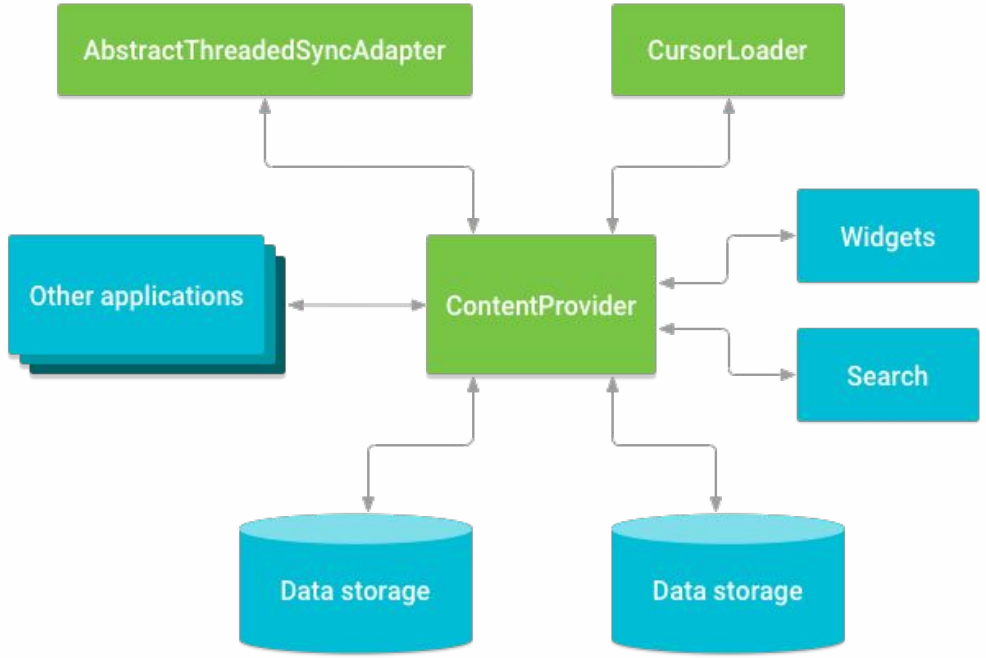
Что это?

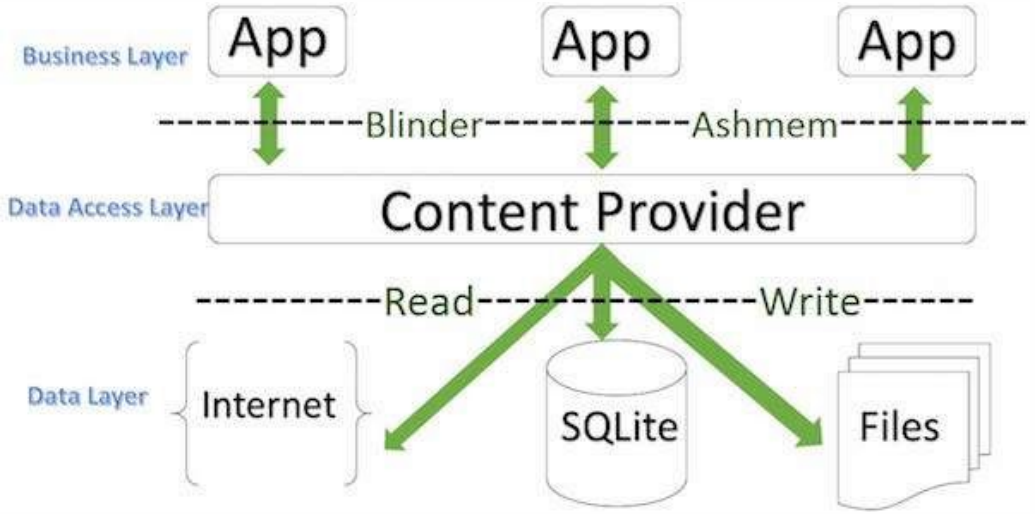
Интерфейс для предоставления доступа к данным приложения. Провайдер сам обеспечивает безопасность данных и безопасную работу из разных процессов.

Как данные хранятся на самом деле - значение не имеет.

Работа с провайдером происходит через **ContentResolver**.

Обычный родительский класс - **ContentProvider**. Но для доступов к хранилищам **SAF** возможно потребуется наследоваться от **DocumentsProvider**.





Указание в Manifest

<provider> - объявление компонента.

- **name** - путь до класса ContentProvider;
- **authorities** - перечисление “URI authorities”, которые поддерживает провайдер.
- и много чего другого...

Если есть наследование от **DocumentsProvider**, то потребуется **<intent-filter>**.

```
<provider
    android:authorities="ru.hse.lection04.log"
    android:name=".datalayer.LogDataAccessor"
    android:exported="true"
    />
```

Имплементация

Для работы провайдера надо будет имплементировать несколько методов:

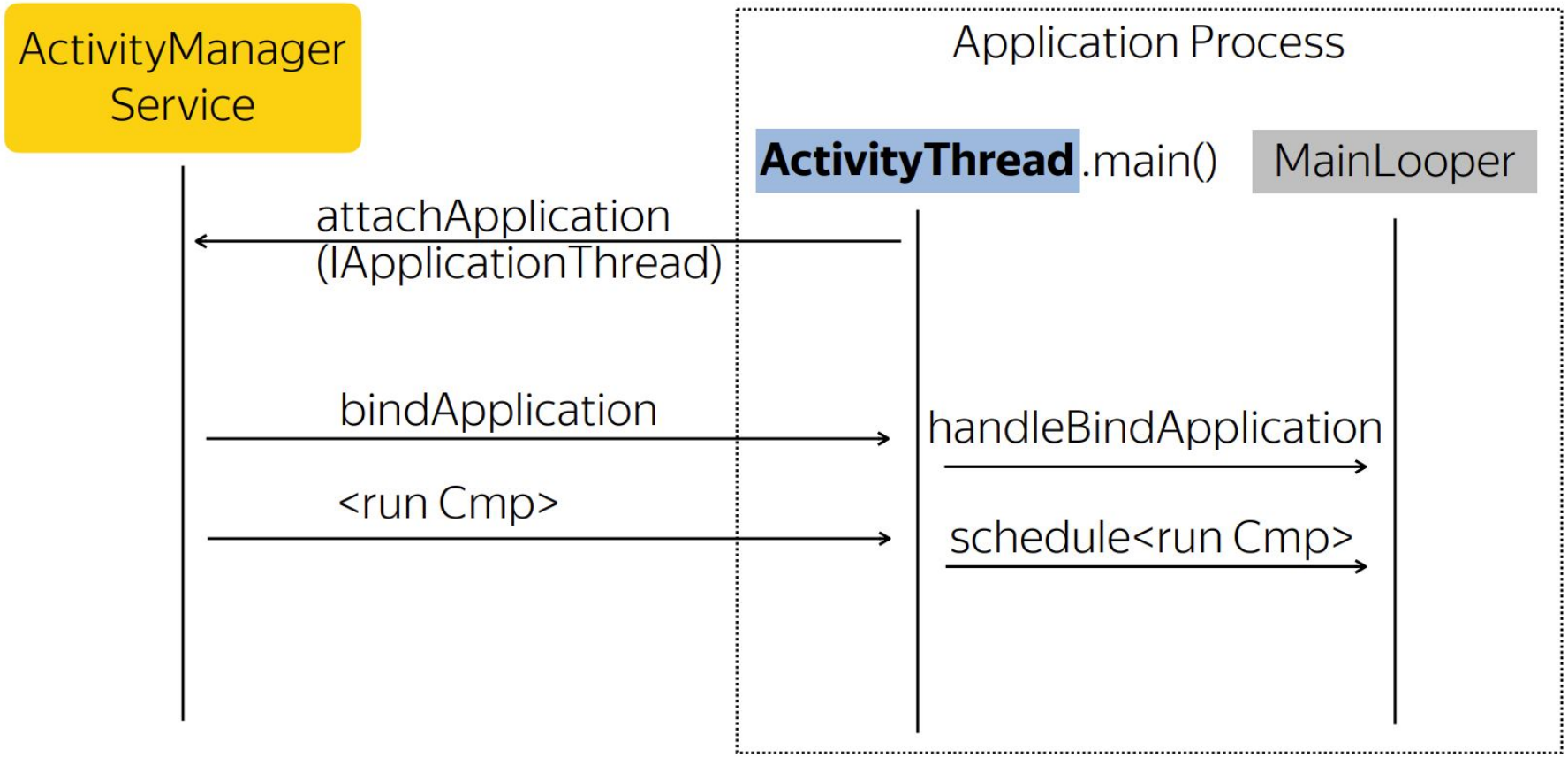
- **onCreate()** - инициализация;
- **getType()** - вернуть тип данных, соответствующего URI контента;
- **insert()** - добавить запись;
- **delete()** - удалить запись;
- **update()** - обновить запись;
- **query()** - выполнить запрос и вернуть результат.

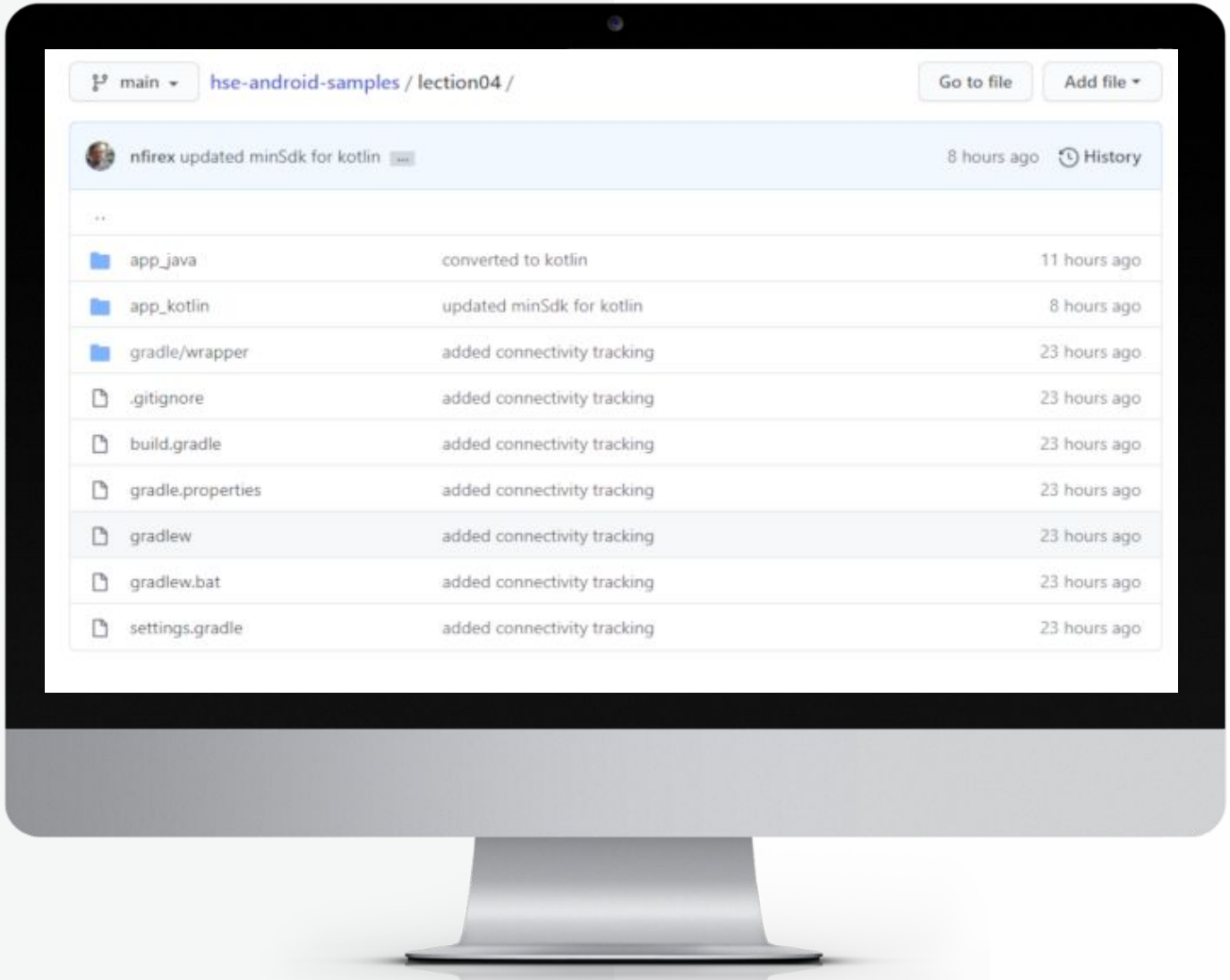
Удобнее всего писать логику над **SQLite**, конечно. Но если вы используете что-то другое, а данные надо возвращать в виде **Cursor** - можно использовать **MatrixCursor**.


```
public class LogDataAccessor extends ContentProvider {
    @Override
    public boolean onCreate() {
        return true;
    }

    @Nullable
    @Override
    public String getType(@NonNull Uri uri) {

    }

    @Nullable
    @Override
    public Uri insert(
        @NonNull Uri uri
        , @Nullable ContentValues values
    ) {

    }

    @Override
    public int delete(
        @NonNull Uri uri
        , @Nullable String selection
        , @Nullable String[] selectionArgs
    ) {

    }

    @Override
    public int update(
        @NonNull Uri uri
        , @Nullable ContentValues values
        , @Nullable String selection
        , @Nullable String[] selectionArgs
    ) {

    }

    @Nullable
    @Override
    public Cursor query(
        @NonNull Uri uri
        , @Nullable String[] projection
        , @Nullable String selection
        , @Nullable String[] selectionArgs
        , @Nullable String sortOrder
    ) {

    }
}
```

Особенность ContentProvider ;)

Он инициализируется до **Application.onCreate()**

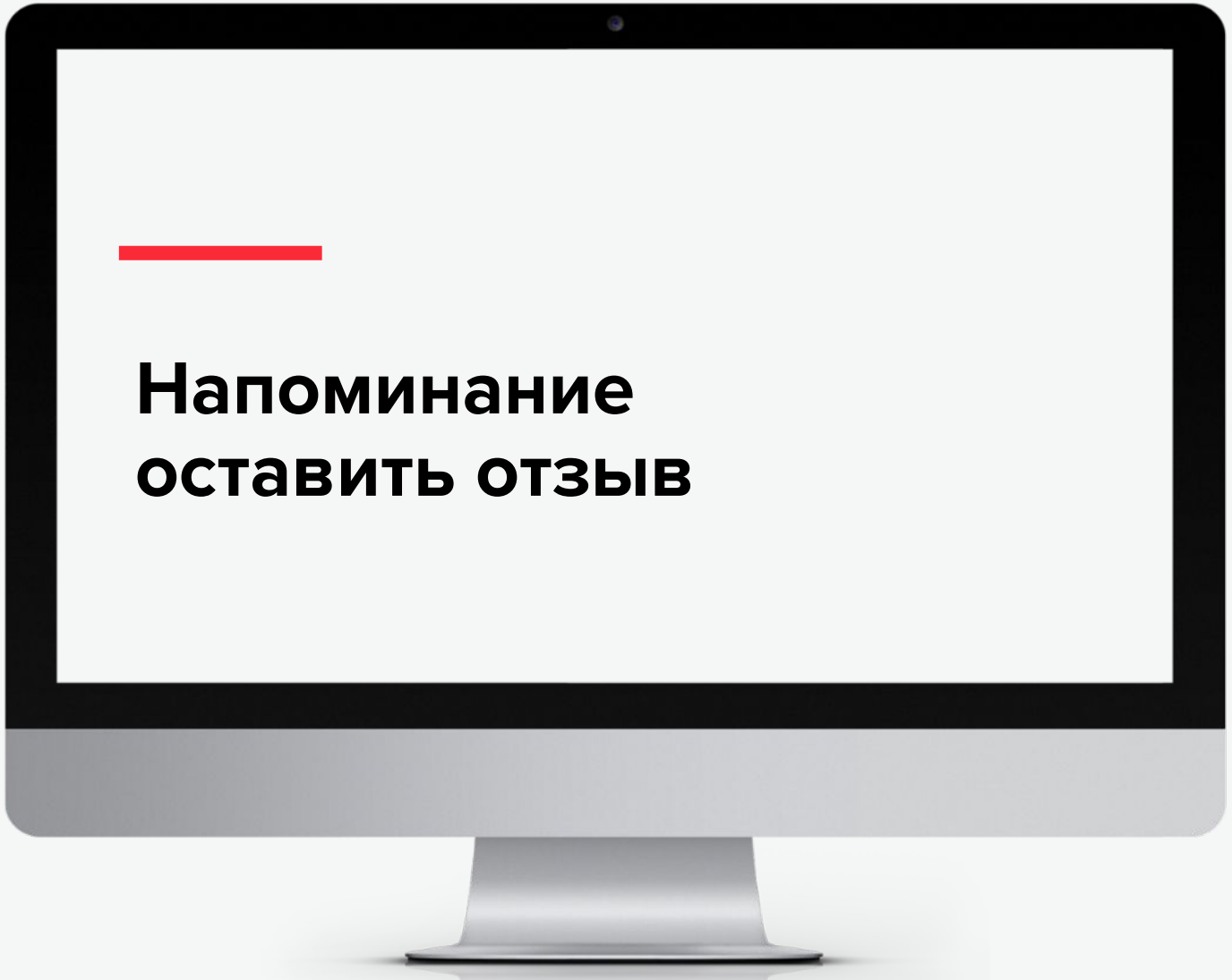




Fix a Bug?

В репозитории есть ошибка в имплементации логики получения информации о состоянии соединения.

Меняю **Pull Request** с фиксом на доп баллы.


Напоминание
оставить отзыв

СПАСИБО
ЗА ВНИМАНИЕ