

Фоновая работа, уведомления

Клещин Никита



A large, bright cyan cross shape is centered on a white background. The cross is composed of four thick, rounded arms that extend towards the corners of the frame. The text "Вспомним что было" is written in black, sans-serif font across the center of the cross.

Вспомним что было



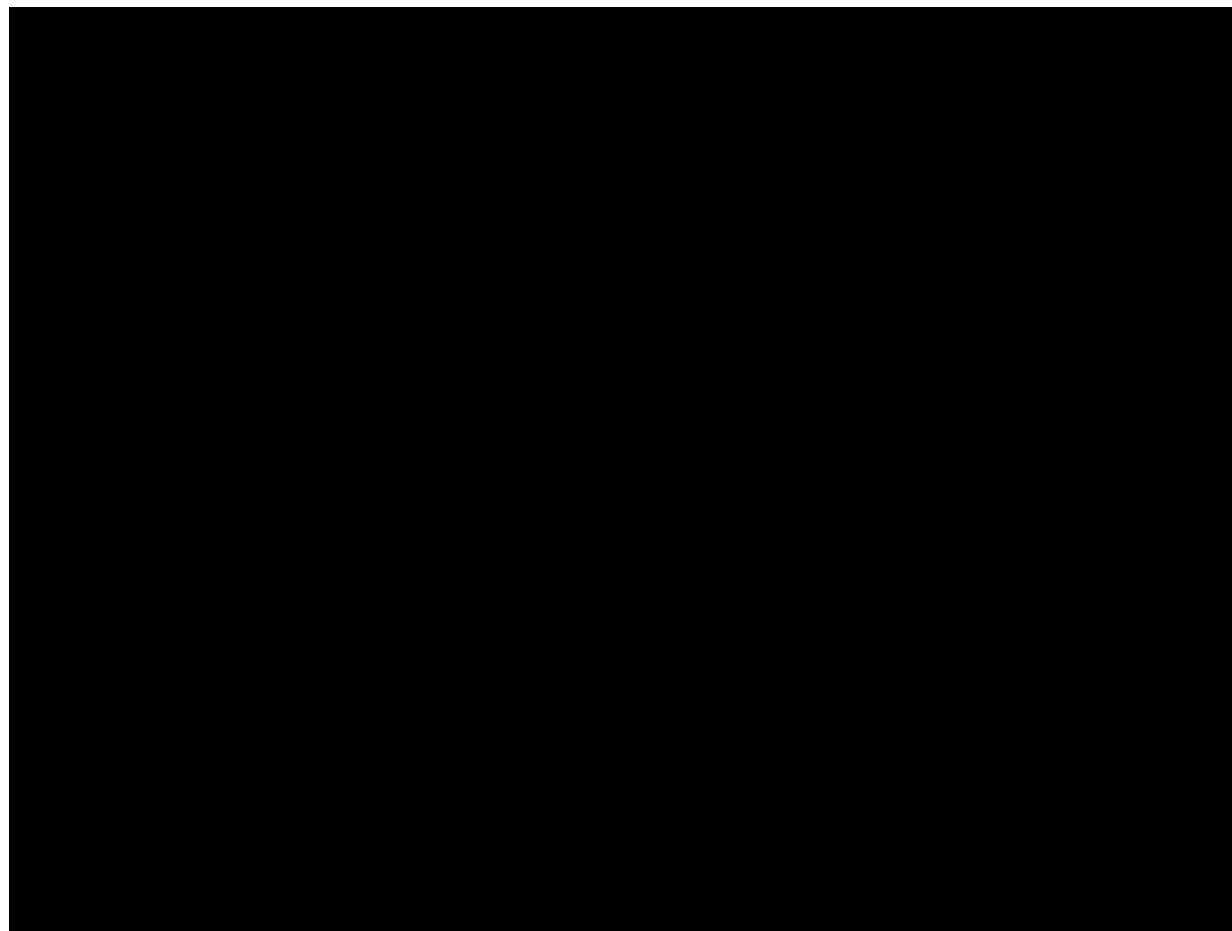
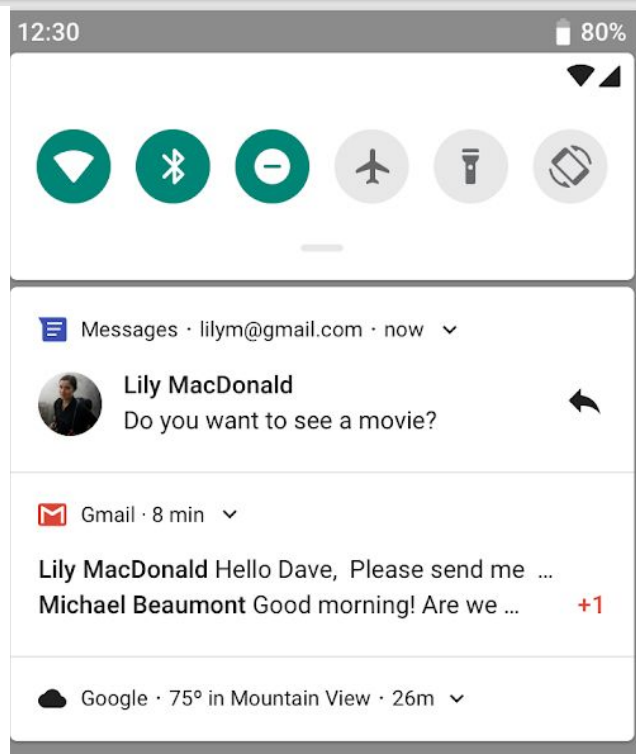
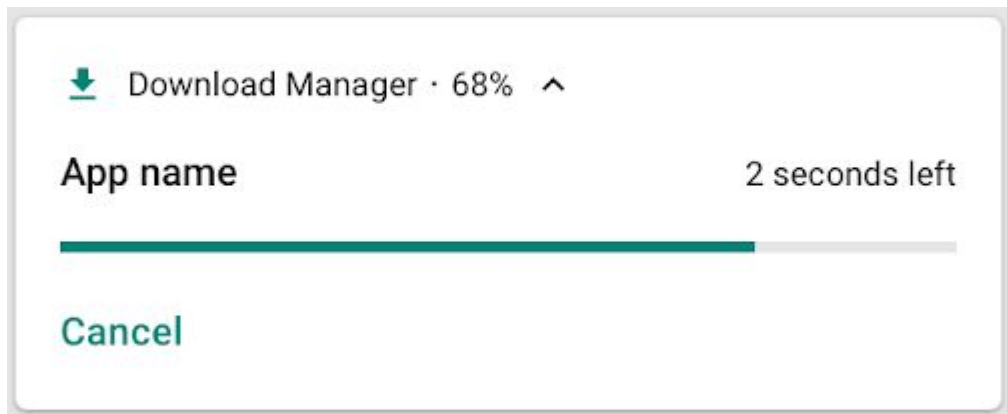
Содержание занятия

1. Notification
2. Service
3. JobScheduler
4. WorkManager
5. AlarmManager
6. ???

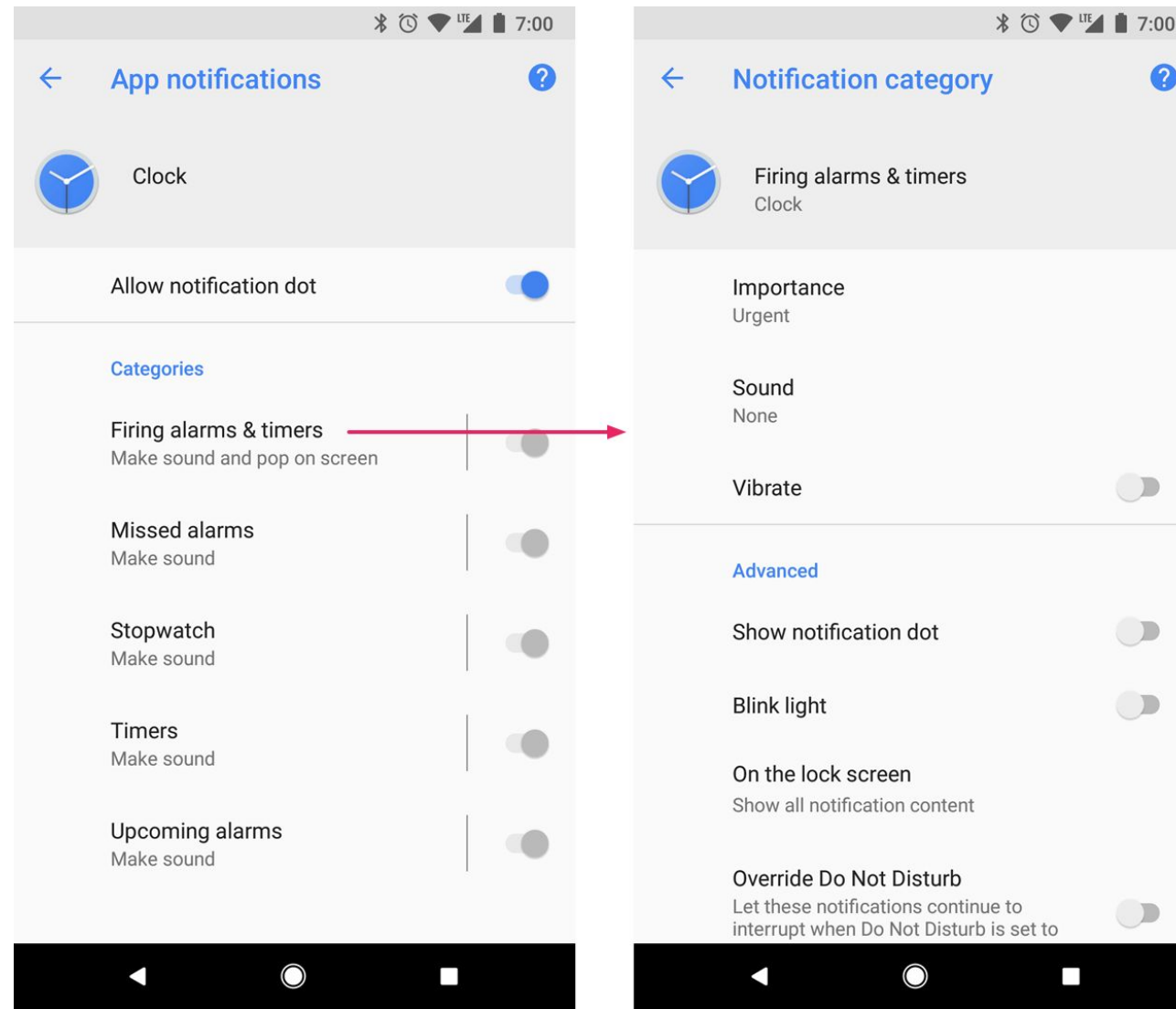
Notification



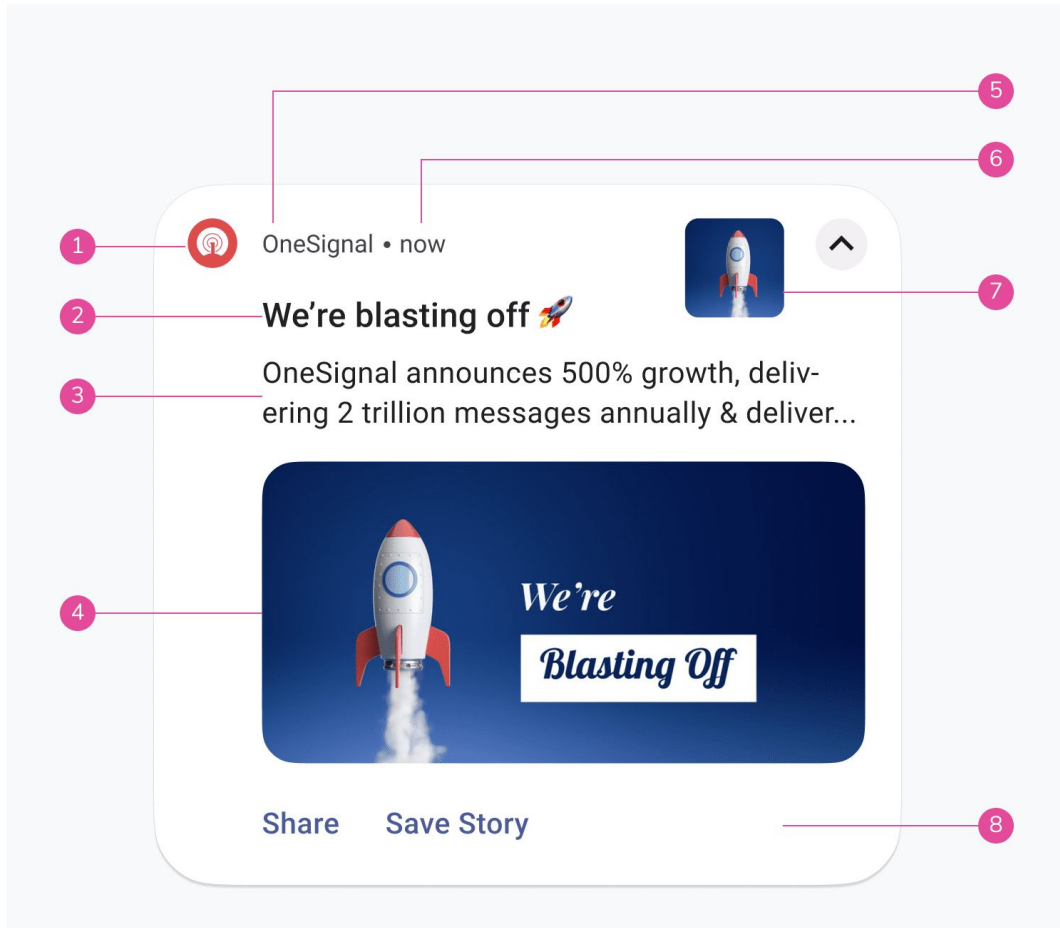
Уведомления



Каналы (начиная с 8 Android)



Анатомия (с 12 Android)



1. Small Icon (24x24 - 96x96, PNG. Белого цвета с прозрачным фоном)
2. Title (до 50 символов)
3. Body (до 50 символов)
4. Large Picture (1440x720 или соотношение сторон 2:1. PNG, JPG, GIF (без анимации))
5. App Name (не меняется)
6. Time (время получения)
7. Icon (192x192 или больше. PNG, JPG, GIF (без анимации))
8. Action Buttons (до трех штук)

Стили уведомлений

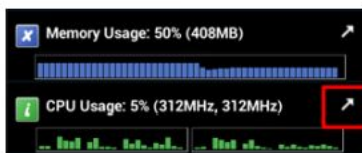
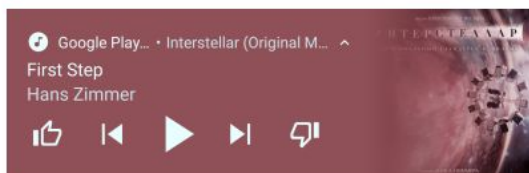
BigTextStyle

 NYTimes • 4h ^

NYTimes

With 130-m.p.h. winds and deadly force, the eye of Hurricane Irma has begun passing over the Florida Keys

MediaStyle



CustomView

InboxStyle

 Gmail • bogdan247@gmail.com • 11m v

scottadamgordon Samsung's DJ Koh: here's why the Note...

scottadamgordon IFA preview update [News]

Bloomberg Technology No more whole paychecks

BigPictureStyle

 Скриншот ^

Скриншот сохранен

Нажмите, чтобы просмотреть

Закат: 16:05

Влажность: 90%

ТЕМП. ОСАДКИ ВЕТЕР ВЛАЖН.



ОТПРАВИТЬ УДАЛИТЬ

*-Compat классы

Чтобы самостоятельно не учитывать версии Android при создании разных компонентов (ведь у них может очень сильно изменяться API), есть библиотеки для обеспечения обратной совместимости. Обычно у них всегда есть постфикс `Compat`. Для работы с уведомлениями лучше использовать их, для сокращения кода:

- **`NotificationManagerCompat.from(context)`**
- **`NotificationCompat.Builder(context, channelId)`**
- **`NotificationChannelGroupCompat.Builder(id)`**
- **`NotificationChannelCompat.Builder(id, importance)`**

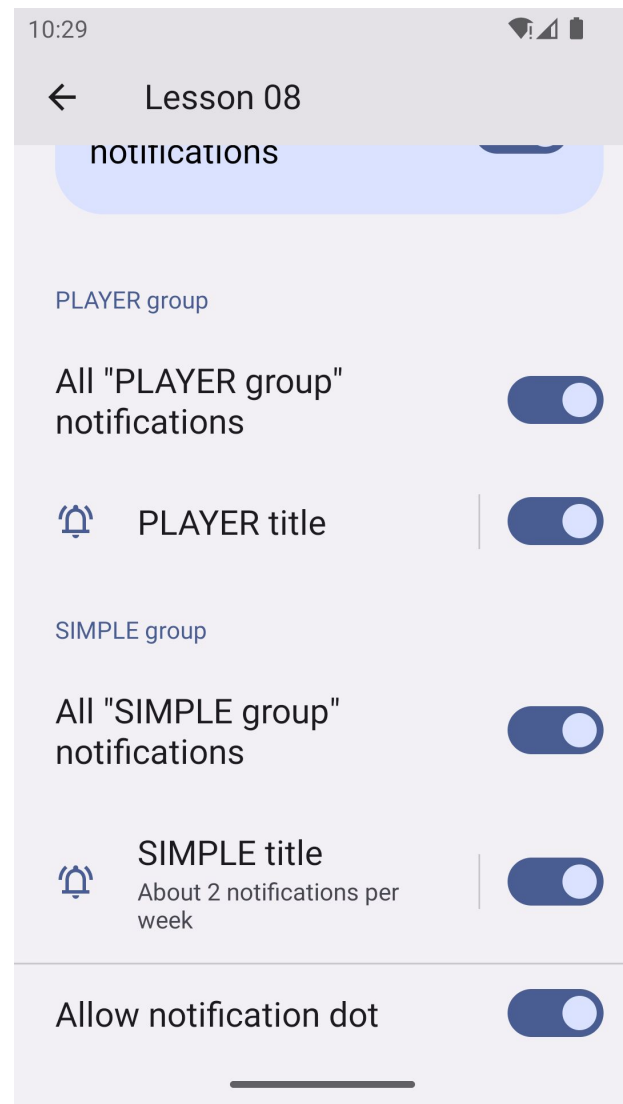
Создать группы (не обязательно)

```
val manager = NotificationManagerCompat.from(context)

// Получить группу
if (manager.getNotificationChannelGroup(id) != null) {
    return
}

val builder = NotificationChannelGroupCompat.Builder(id).apply {
    setName(title)
}

// Создать группу
manager.createNotificationChannelGroup(builder.build())
...
// Удалить группу
manager.deleteNotificationChannelGroup(id)
```



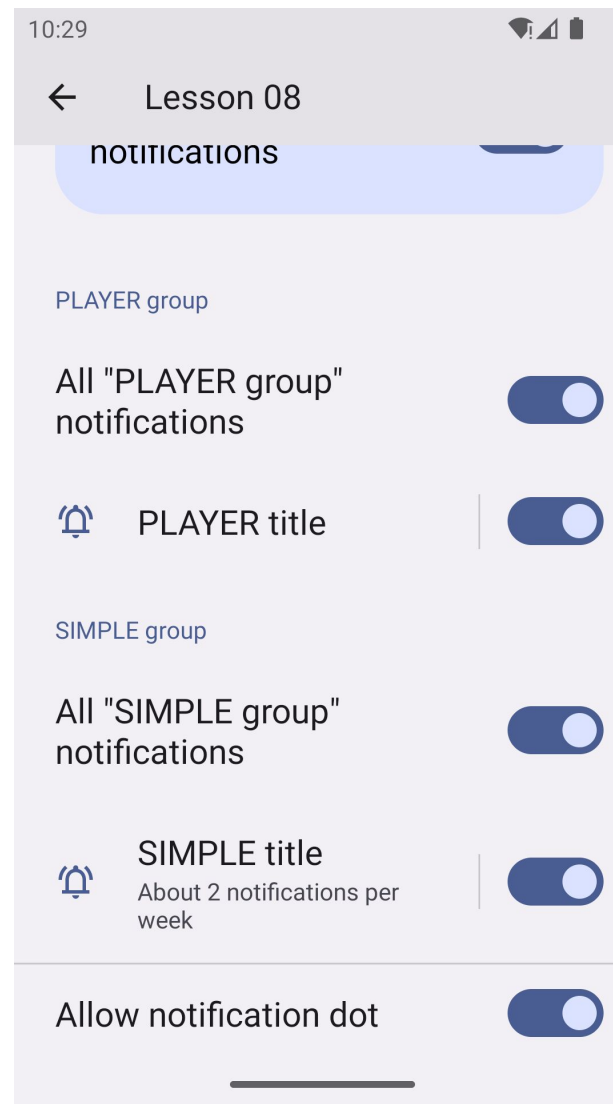
Создать каналы

```
val manager = NotificationManagerCompat.from(context)

// Получить канал
if (manager.getNotificationChannel(id) != null) {
    return
}

val builder = NotificationChannelCompat.Builder(id, importance).apply {
    setName(title)
    setDescription(description)
    setGroup(groupId)
}

// Создать канал
manager.createNotificationChannel(builder.build())
...
// Удалить канал
manager.deleteNotificationChannel(id)
```



Получить разрешение

// С Android 13 нужно запрашивать разрешение на показ уведомлений

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

```
class MainActivity : AppCompatActivity() {
    protected val notificationPermissionLauncher = registerForActivityResult(
        ActivityResultContracts.RequestPermission(),
        ::proceedNotificationPermission
    )

    fun requestPermission(activity: Activity) {
        if (Build.VERSION.SDK_INT < Build.VERSION_CODES.TIRAMISU) {
            // Текущая версия Android не требует разрешение для показа уведомлений
            return
        }

        if (ContextCompat.checkSelfPermission(activity, Manifest.permission.POST_NOTIFICATIONS) == PackageManager.PERMISSION_GRANTED) {
            // Разрешение уже выдано
            return
        }

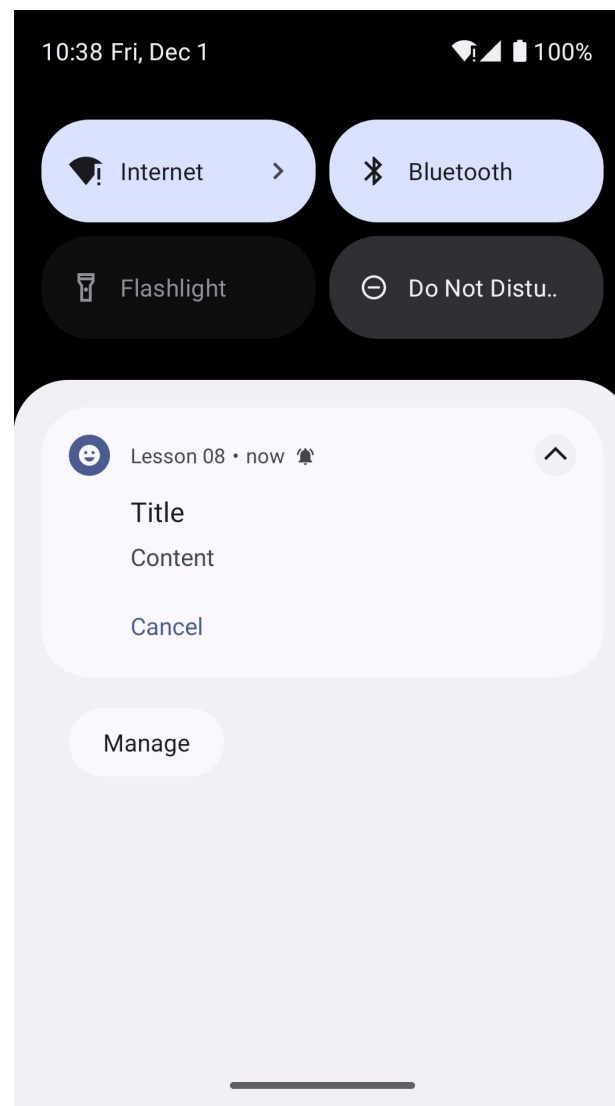
        if (ActivityCompat.shouldShowRequestPermissionRationale(activity, Manifest.permission.POST_NOTIFICATIONS)) {
            // Означает что пользователь уже видел предложение дать разрешение, но отказался.
            // Google рекомендует не донимать пользователя таким вопросом
            return
        }

        notificationPermissionLauncher.launch(Manifest.permission.POST_NOTIFICATIONS)
    }

    protected fun proceedNotificationPermission(isGranted: Boolean) {
        ...
    }
}
```

Показать уведомление

```
if (ActivityCompat.checkSelfPermission(context, Manifest.permission.POST_NOTIFICATIONS) != PackageManager.PERMISSION_GRANTED) {  
    // Разрешения нет  
    return  
}  
  
val builder = NotificationCompat.Builder(context, channelId).apply {  
    priority = NotificationCompat.PRIORITY_DEFAULT  
  
    setSmallIcon(R.mipmap.ic_notification)  
    setContentTitle("Title")  
    setContentText("Content")  
    setGroup(groupId)  
  
    // Не "смахиваемый"  
    setOngoing(true)  
  
    // Добавить кнопку  
    val actionIntent: Intent = ...  
    val actionPendingIntent: PendingIntent = ...  
    addAction(0, actionTitle, actionPendingIntent)  
}  
  
val manager = NotificationManagerCompat.from(context)  
  
// Показать уведомление  
manager.notify(notificationId, builder.build())  
...  
// Отменить уведомление  
manager.cancel(notificationId)  
manager.cancelAll()
```



Важность

User-visible importance level	Importance (Android 8.0 and higher)	Priority (Android 7.1 and lower)
Urgent Makes a sound and appears as a heads-up notification	<code>IMPORTANCE_HIGH</code>	<code>PRIORITY_HIGH</code> or <code>PRIORITY_MAX</code>
High Makes a sound	<code>IMPORTANCE_DEFAULT</code>	<code>PRIORITY_DEFAULT</code>
Medium No sound	<code>IMPORTANCE_LOW</code>	<code>PRIORITY_LOW</code>
Low No sound and does not appear in the status bar	<code>IMPORTANCE_MIN</code>	<code>PRIORITY_MIN</code>

PendingIntent?

Обертка для **Intent**. **PendingIntent** запустит Intent в каком-то будущем, при наступлении определенных событий.

Частый случай применения такого объекта - это действия в уведомлениях.

Варианты создания PendingIntent:

- **PendingIntent.getActivity** - запуск Activity
- **PendingIntent.getActivities** - то же, что и прошлый, но для нескольких Activity
- **PendingIntent.getBroadcast** - триггер для BroadcastReceiver
- **PendingIntent.getService/getForegroundService** - триггер для Service

Где

- **context** - (Как думаете, это что?)
- **requestCode** - Приватный ключ для того, что бы различать PendingIntent
- **intent** - (Как думаете, это что?)
- **flags** - флаги, настраивающие поведение PendingIntent. Можно указывать несколько при помощи бинарной операции (например, PendingIntent.FLAG_IMMUTABLE or PendingIntent.FLAG_UPDATE_CURRENT)

BroadcastReceiver

Основной компонент приложения. Используется для подписки на какие-то уведомления в системе. Также, может использоваться для общения компонентов внутри приложения.

От версии к версии Android - ужимаются в правах:)

И при подписи на какие-то события, надо внимательно читать, где следует регистрироваться

IntentFilter - это условия срабатывания триггера (можно заметить, что такой фильтр есть и у **Activity** и у **Service**).

```
// Обязательная регистрация в манифесте
<receiver android:name=".MyBroadcastReceiver" android:exported="false">
```

```
    // Фильтр нужен, если планируем получить события извне
    <intent-filter>
        <action android:name="APP_SPECIFIC_BROADCAST" />
    </intent-filter>
</receiver>
```

```
// Подписаться в коде
val receiver = ActionBroadcastReceiver()
val filter = IntentFilter(Intent.ACTION_TIME_TICK)
ContextCompat.registerReceiver(
    context,
    receiver,
    filter,
    ContextCompat.RECEIVER_EXPORTED
)
```

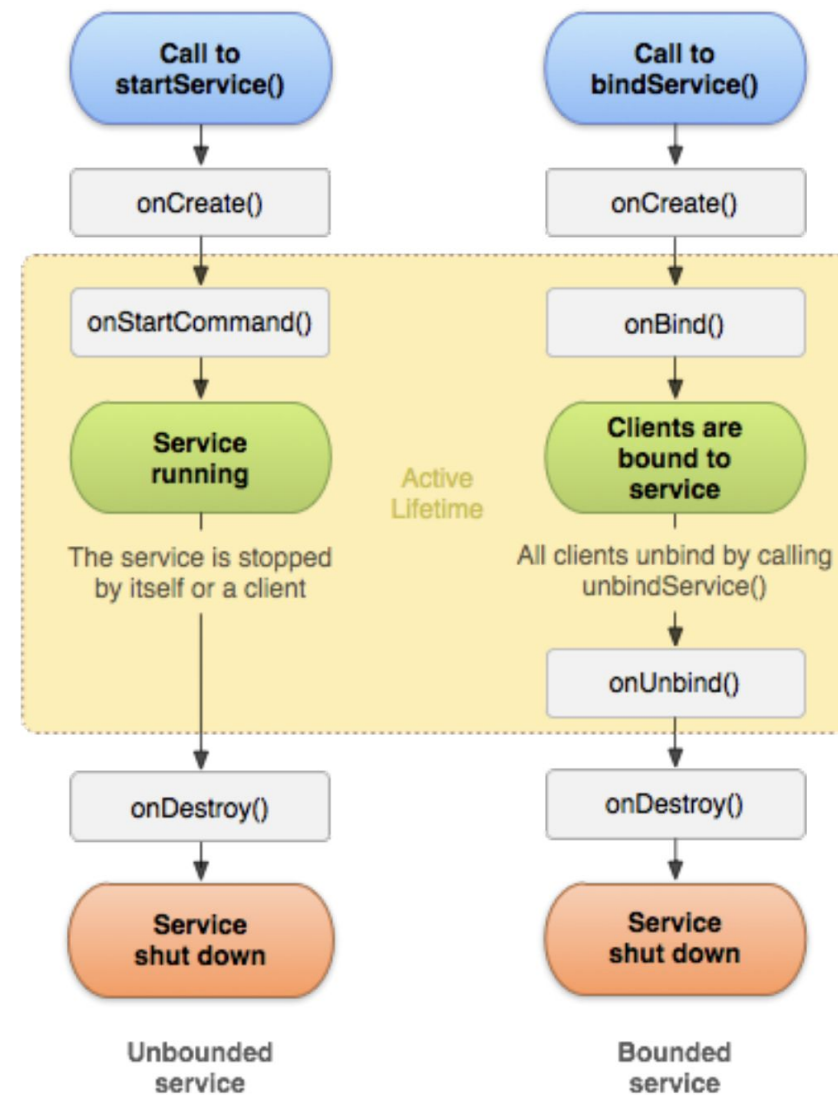
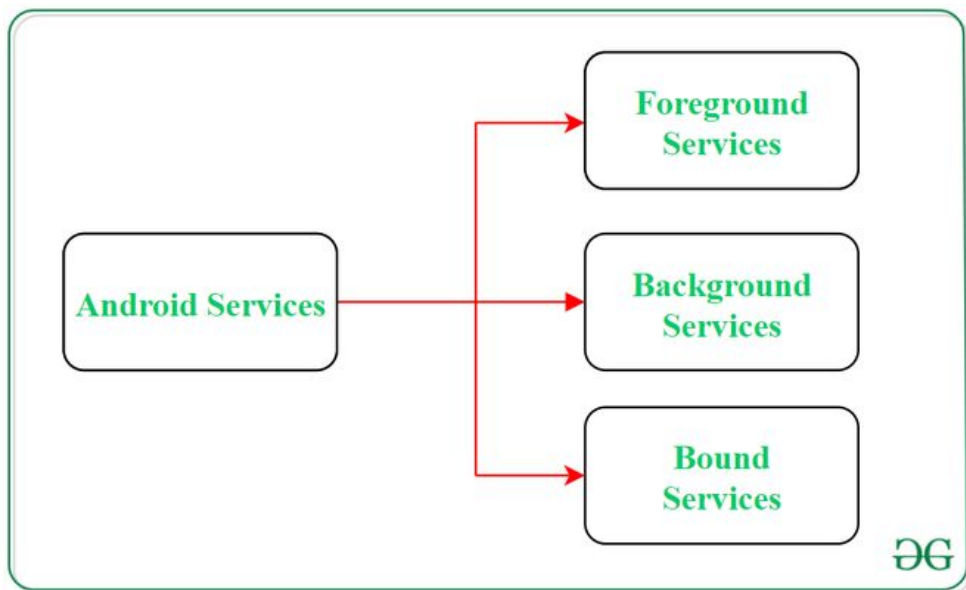
```
// Отписаться в коде
Context.unregisterReceiver(receiver)
```


Service



Сервисы

- Основной компонент приложения
- По-умолчанию работает в **главном** потоке
- Повышает приоритет процесса
- Может быть Background/Foreground/Bounded ???



Background Service

Сервис, который стартуется при помощи

- **Context.startService()**

Команду startService также можно использовать для отправки сообщений в сервис.

После старта он живет до тех пор пока не будет остановлен вручную, при помощи

- **Context.stopService()** вне сервиса
- **Service.stopSelf()** из сервиса

У этого сервиса в поздних версиях Android есть сильное ограничение - он не может жить без визуальной части приложения (система его будет убивать).

// Сервис необходимо регистрировать в манифесте

```
<service
    android:name=".BackgroundService"
    android:enabled="true"
/>
```

```
fun sendCommand(context: Context, command: String, data: String?) {
    val intent = Intent(context, BackgroundService::class.java).apply {
        putExtra(COMMAND, command)
        putExtra(DATA, data)
    }

    context.startService(intent)
}

class BackgroundService : Service() {
    override fun onStartCommand(intent: Intent?, flags: Int, startId: Int): Int {
        val command = intent?.getStringExtra(COMMAND)
        val data = intent?.getStringExtra(DATA)
        handleCommand(command, data)

        return START_NOT_STICKY
    }

    protected fun handleCommand(command: String, data: String?) {
        ...

        stopSelf()
    }
    ...
}
```

Foreground Service

Сервис, который может жить без визуальной части вашего приложения (система не будет его убивать просто так). Этот сервис должен быть видим для пользователя за счет уведомления, который сервис должен будет показать при запуске (надо проставить в течении 5 секунд после запуска! Иначе он будет убит):

- **ServiceCompat.startForeground()**

Этот сервис стартуется при помощи:

- **ContextCompat.startForegroundService()**

Останавливается сервис при помощи тех же команд что и Background Service.

Еще одно отличие этого вида сервиса - это тип его использования - foregroundServiceType и разрешения

```
// Сервис регистрируется в манифесте чуть сложнее
<uses-permission android:name="android.permission.FOREGROUND_SERVICE" />
```

```
// И возможно еще потребуется добавить uses-permission, в зависимости от
типа задач вашего сервиса (это отдельно надо будет смотреть в
документации)
```

```
<service
    android:name=".ForegroundService"
    android:foregroundServiceType="mediaPlayback"
    android:enabled="true"
/>
```

```
fun executeCommand(context: Context, command: String) {
    val intent = Intent(context, ForegroundService::class.java).apply {
        putExtra(COMMAND, command)
    }

    ContextCompat.startForegroundService(context, intent)
}

class ForegroundService : Service() {
    override fun onCreate() {
        super.onCreate()

        ...
        ServiceCompat.startForeground(this, id, ongoingNotification, type)
    }
}
```

Правила onStartCommand

Обработка Intent сервисом необходимо делать в методе onStartCommand.

Этот метод должен будет вернуть значение, которое говорит о правилах рестарта этого сервиса.

Метод onStartCommand так же может сказать как была вызвана эта команда, при помощи параметра **flag**:

- **0** - ??? описания нет, но это обычный старт. Считается что скорее всего из-за бага он никогда не приходит, а вместо него будет **START_FLAG_RETRY**
- **START_FLAG_RETRY** - указывает на то, что сервис повторно запущен после непредвиденного завершения работы; передается в том случае, если ранее сервис работал в режиме **START_STICKY**
- **START_FLAG_REDELIVERY** - указывает на то, что параметр Intent повторно передан при принудительном завершении работы сервиса перед явным вызовом метода **stopSelf()**

Результаты метода onStartCommand:

START_NOT_STICKY/START_STICKY_COMPATIBILITY

– сервис не будет перезапущен после того, как был убит системой

START_STICKY – сервис будет перезапущен после того, как был убит системой

START_REDELIVER_INTENT – сервис будет перезапущен после того, как был убит системой. Кроме этого, сервис снова получит все вызовы **startService**, которые не были завершены методом **stopSelf(startId)**.

Bound Service

Сервис, который стартуется при помощи метода

- **Context.bindService**

После такого старта, при помощи класса:

- **ServiceConnection**

Можно получить объект

- **Binder**

При помощи которого можно общаться с сервисом напрямую (без команд **startService**).

Жизненный цикл этого сервиса отличается от прошлых

- сервис умрет, когда от него отсоединяется все подписчики (Binder), при помощи метода

- **Context.unbindService**

```
// Сервис все равно надо будет регистрировать в манифесте
```

```
<service
    android:name=".BoundService"
    android:enabled="true"
/>
```

```
fun connect() {
    val intent = Intent(context, BoundService::class.java)
    context.bindService(intent, this, Context.BIND_AUTO_CREATE)
}
```

```
fun disconnect() {
    context.unbindService(this)
}
```

```
class BoundService : Service() {
    protected val binder by lazy { Controller() }

    override fun onBind(intent: Intent) = binder

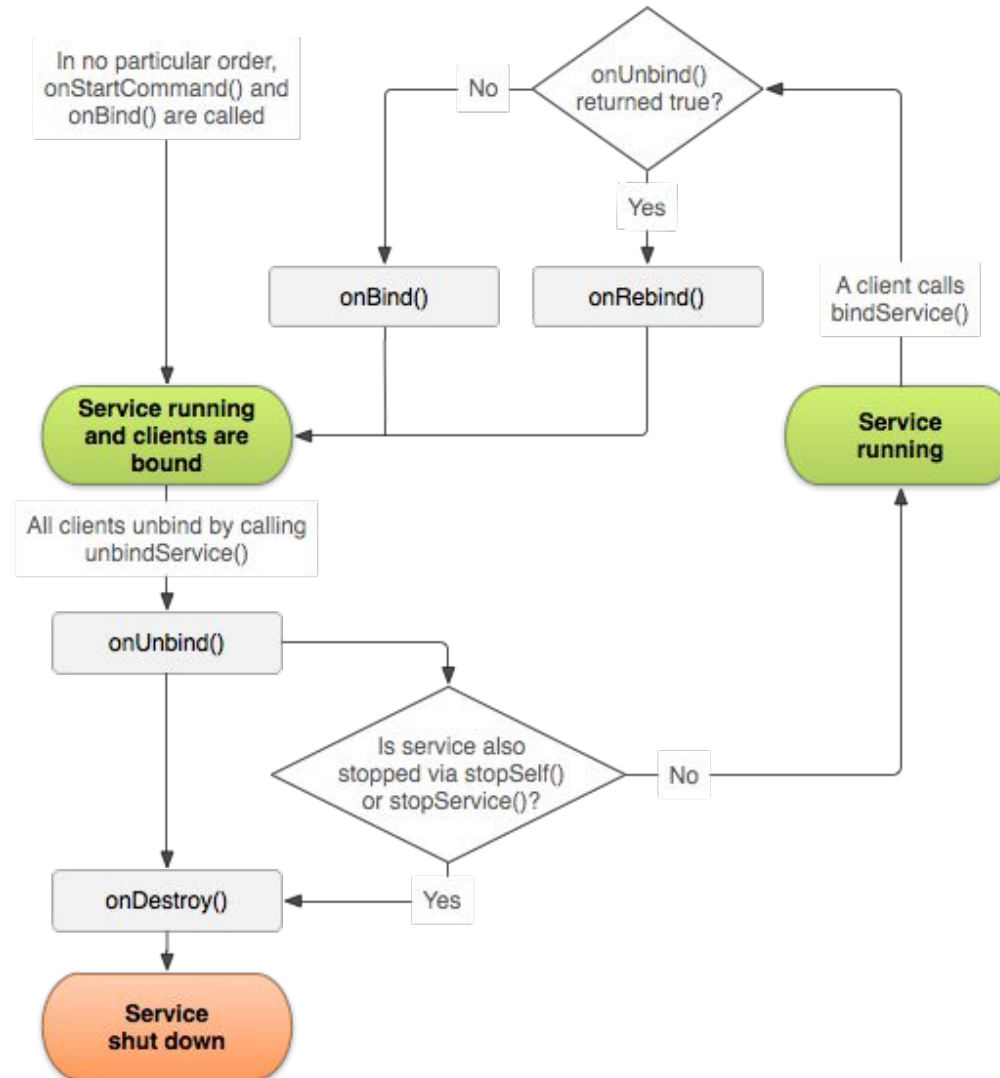
    inner class Controller : Binder() {
        ...
    }
}
```

```
class Connection(protected val context: Context): ServiceConnection {
    protected var binder: Controller? = null

    // Сервис подключен
    override fun onServiceConnected(name: ComponentName?, service: IBinder?) {
        binder = service as? Controller
    }

    // Сервис отключился (по каким-то причинам, без unbindService)
    override fun onServiceDisconnected(name: ComponentName?) {
    }
}
```

Started + Bound



Сервис и процессы

По умолчанию сервис работает в главном процессе. А это значит, что если у него внутри будет неотловленная ошибка, то она убьет приложение.

В каких-то случаях будет необходимость запускать сервис в отдельных процессах. Этим можно управлять при регистрации класса **Service** в манифесте при помощи параметра **process**.

Если сервис обычный (Background, Foreground), то механика работы с ним не меняется.

Если же ваш сервис Bound Service, то обычный Binder уже не подойдет - придется описать его через .aidl файл, и учесть ограничения на передачу данных (можно передавать **базовые типы** и **Parcelable**). И инициализировать Binder через **IRemoteService.Stub**.

```
<service
    android:name=".ForegroundService"
    android:foregroundServiceType="mediaPlayback"
    android:enabled="true"
    android:exported="true"
    android:process=":player"
>
<intent-filter>

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

// IRemoteService.aidl
package com.example.android;

// Declare any non-default types here with import statements.

/** Example service interface */
interface IRemoteService {
    /** Request the process ID of this service. */
    int getPid();

    /** Demonstrates some basic types that you can use as parameters
     * and return values in AIDL.
     */
    void basicTypes(int anInt, long aLong, boolean aBoolean, float aFloat, double
aDouble, String aString);
}

private val binder = object : IRemoteService.Stub() {
    override fun getPid(): Int = Process.myPid()

    override fun basicTypes(anInt: Int, aLong: Long, aBoolean: Boolean, aFloat:
Float, aDouble: Double, aString: String) {
        // Does nothing.
    }
}
```


JobScheduler



IntentService

Ранее, для выполнения коротких задач, в асинхронном потоке (Обычный сервис же выполняется в главном потоке), использовался `IntentService`. Его надо было регистрировать в манифесте так же, как и обычный сервис

Его логика работы была простая - сервис запускался и начинал выполнять работу. Если во время его работы ему прилетали новые команды, то они добавлялись в очередь и исполнялись по порядку.

По окончании работы - сервис каким-то способом уведомлял слушателя (зачастую, при помощи `BroadcastReceiver`), и сам себя останавливал.

Этот класс уже считается устаревшим с Android API 30.

На его смену пришел `JobScheduler`.

```
/// Для работы с Intent надо было использовать метод onHandleIntent
@Override
protected void onHandleIntent(Intent intent) {
    if (intent != null) {
        final String action = intent.getAction();
        if (ACTION_WRITE_EXERCISE.equals(action)) {
            handleActionWriteExercise();
        }
    }
}
```

JobScheduler

Планировщик задач...

Сами задачи обрабатываются

- **JobService**

Данные для выполнения задач необходимо создать

- **JobInfo.Builder**

Сам **JobInfo** можно наполнить разными условиями, которые будут триггерить выполнения задачи.

После создания **JobInfo** его можно добавить в список на исполнение при помощи команды

JobScheduler.schedule. А отменить выполнение при помощи **JobScheduler.cancel**. Шедулер можно получить при помощи **Context**:

- `Context.getSystemService(Context.JOB_SCHEDULER_SERVICE)` as **JobScheduler**

Несмотря на то что описывают “как легко он решает ваши проблемы”... Могу сказать что придется попотеть, чтобы заставить его работать как надо -_-

// Да, это тоже сервис)

```
<service
    android:name=".JobSchedulerService"
    android:permission="android.permission.BIND_JOB_SERVICE"
    android:exported="true"
/>
```

// Для работы необходимо переопределить **onStartJob** и **onStopJob**

```
class JobSchedulerService: JobService() {
    // Вызывается, когда триггер на исполнение сработал
    override fun onStartJob(params: JobParameters?): Boolean {
        ...
    }

    // Вызывается, когда исполнять задачу больше не надо
    override fun onStopJob(params: JobParameters?): Boolean {
        ...
    }
}
```

```
fun scheduleSimpleJob(context: Context) {
    val component = ComponentName(context, JobSchedulerService::class.java)
    val builder = JobInfo.Builder(jobId, component).apply {
        ...
    }

    val job = builder.build()
    val scheduler = context
        .getSystemService(Context.JOB_SCHEDULER_SERVICE) as JobScheduler

    scheduler.schedule(job)
}
```

```
fun unscheduleSimpleJob(context: Context) {
    val scheduler = context
        .getSystemService(Context.JOB_SCHEDULER_SERVICE) as JobScheduler

    scheduler.cancel(jobId)
}
```

WorkManager



А теперь забыли про
JobScheduler:)

WorkManager из Jetpack

WorkManager - фреймворк.

Позволяет запускать фоновые задачи последовательно или параллельно, передавать в них данные, получать из них результат, отслеживать статус выполнения и запускать только при соблюдении заданных условий.

Но, тоже есть нюансы с работой.

Распространенное заблуждение - это как периодически может исполняться задача (МИНИМУМ 15 МИНУТ, это уже почти мем).

```
implementation 'androidx.work:work-runtime-ktx:2.9.0'

fun schedule(context: Context, startTime: Long, scope: CoroutineScope) {
    val uuid = UUID.randomUUID()

    // Данные для работы
    val dataBuilder = Data.Builder().apply {
        putLong(EXTRAS_START_TIME, startTime)
    }

    // Условия для срабатывания
    val constraints = Constraints.Builder().apply {
        setRequiredNetworkType(NetworkType.CONNECTED)
    }

    val requestBuilder = OneTimeWorkRequest.Builder(TimerWork::class.java).apply {
        ...
    }

    // Запланировать работу
    val manager = WorkManager.getInstance(context)
    manager.enqueue(requestBuilder.build())

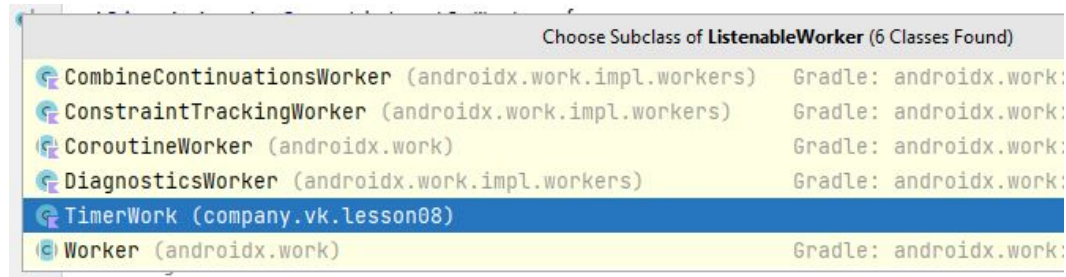
    // Если хотим подписаться на прогресс или прогрессы
    manager.getWorkInfoByIdFlow(uuid)
        .onEach { info ->
            ...
        }
        .launchIn(scope)
}

// Отменить выполнение работы
fun cancel(context: Context) {
    WorkManager.getInstance(context).cancelAllWorkByTag(TAG)
}
```

Работа распределяется по Worker-ам

Воркеры выполняют отдельные задач.

Есть какие-то количество базовых воркеров



```
class TimerWork(context: Context, params: WorkerParameters)
    : CoroutineWorker(context, params) {

    override suspend fun doWork(): Result {
        // Если хотим рассказать про прогресс
        setProgress(builder.build())

        // Выставляем результат работы
        return Result.success()
    }
}
```

Перед работой, воркеры оборачиваются в Request:

- **OneTimeWorkRequest** - задача будет выполнена один раз
- **PeriodicWorkRequest** - задача будет выполнена многократно

AlarmManager



AlarmManager - 1

Служба для отправки разовых/повторяющихся сообщений в **заданное** время. Подходит для планировщиков, будильников, периодических проверок. Способно пробудить устройство, но после перезапуска устройства - задачи отменяются.

AlarmManager получается при помощи Context

- `getSystemService(Context.ALARM_SERVICE)` as `AlarmManager`

Установка “будильника” происходит одним из следующих методов:

- **set/setExact/setAlarmClock** - Одноразовые срабатывания
- **setExactAndAllowWhileIdle/setAndAllowWhileIdle** - Срабатывания, даже в сберегающем режиме
- **setRepeating/setInexactRepeating** - Повторяющиеся срабатывания

так же, есть **AlarmManagerCompat**, но он тут уже скорее как хелпер-класс, методы которого требуют `AlarmManager` для работы.

Объект в поле **operation** в установке будильника - это **PendingIntent**, который может принадлежать **Activity/Activities**, **Service** или **BroadcastReceiver**.

Значения для **typeOne**:

- **ELAPSED_REALTIME** - учет времени идет с момента запуска устройства.
- **ELAPSED_REALTIME_WAKEUP** - то же, что и прошлое, но выводит из режима ожидания
- **RTC** - запуск в точное время
- **RTC_WAKEUP** - то же, что и прошлое, но выводит из режима ожидания

AlarmManager - 2

Отменить можно при помощи

- **AlarmManager.cancel()**
- **AlarmManager.cancelAll()**

```
fun schedule(context: Context) {  
    val alarmManager = context.getSystemService(Context.ALARM_SERVICE) as AlarmManager  
    val operation = operation(context)  
  
    alarmManager.setAndAllowWhileIdle()  
    alarmManager.setRepeating(AlarmManager.RTC_WAKEUP, exactTime, period, operation)  
}  
  
fun cancel(context: Context) {  
    val alarmManager = context.getSystemService(Context.ALARM_SERVICE) as AlarmManager  
    val operation = operation(context)  
  
    alarmManager.cancel(operation)  
}  
  
private fun operation(context: Context): PendingIntent {  
    val intent = Intent(context, AlarmReceiver::class.java)  
    val pendingIntent = PendingIntent.getBroadcast(context, 0, intent, PendingIntent.FLAG_MUTABLE)  
  
    return pendingIntent  
}
```

Всегда есть какое-то НО

```
// А как же точные вызовы....  
AlarmManager system_server W Suspiciously short interval 2000 millis; expanding to 60 seconds
```

Если часто обращаетесь к нему... Можно получить письмо счастья от GooglePlay

???





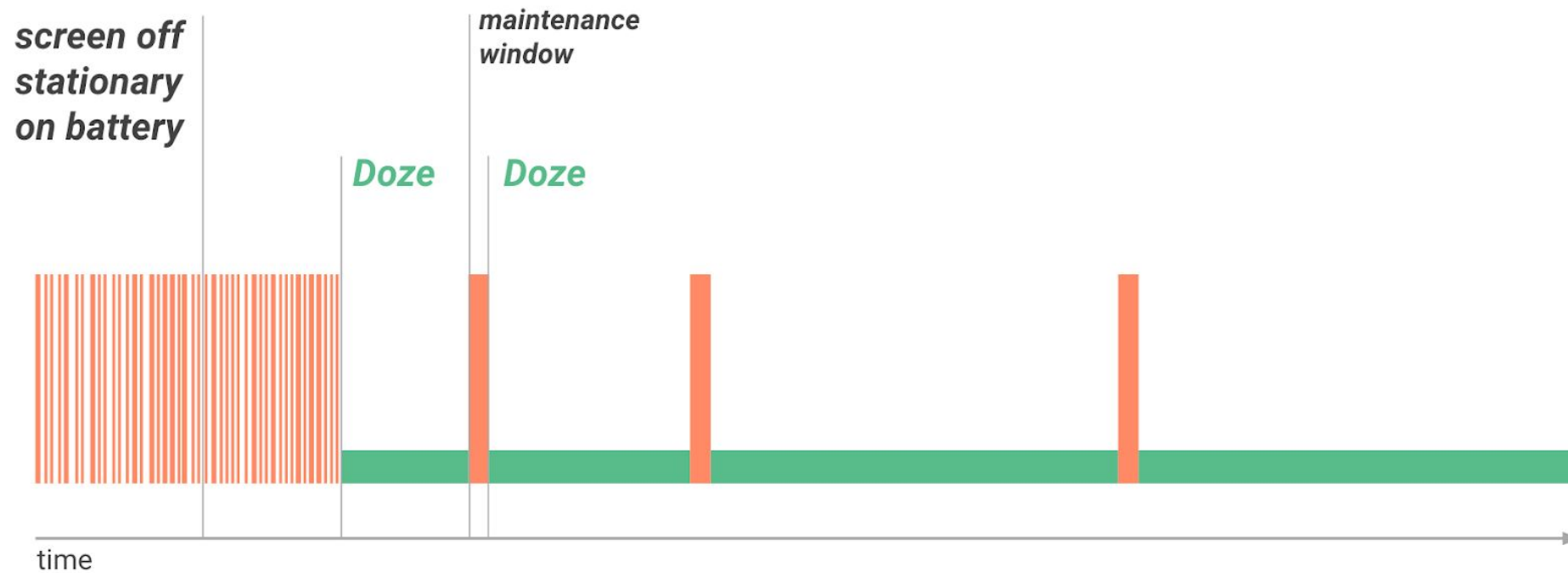
Doze Mode & App Standby (Android 6.0)

Doze mode - режим, в который переходят все приложения, если устройство не подключено к зарядке и неподвижно в течение длительного промежутка времени;

App Standby - режим, в который переходят отдельно взятые приложения, которыми давно не пользовался пользователь.

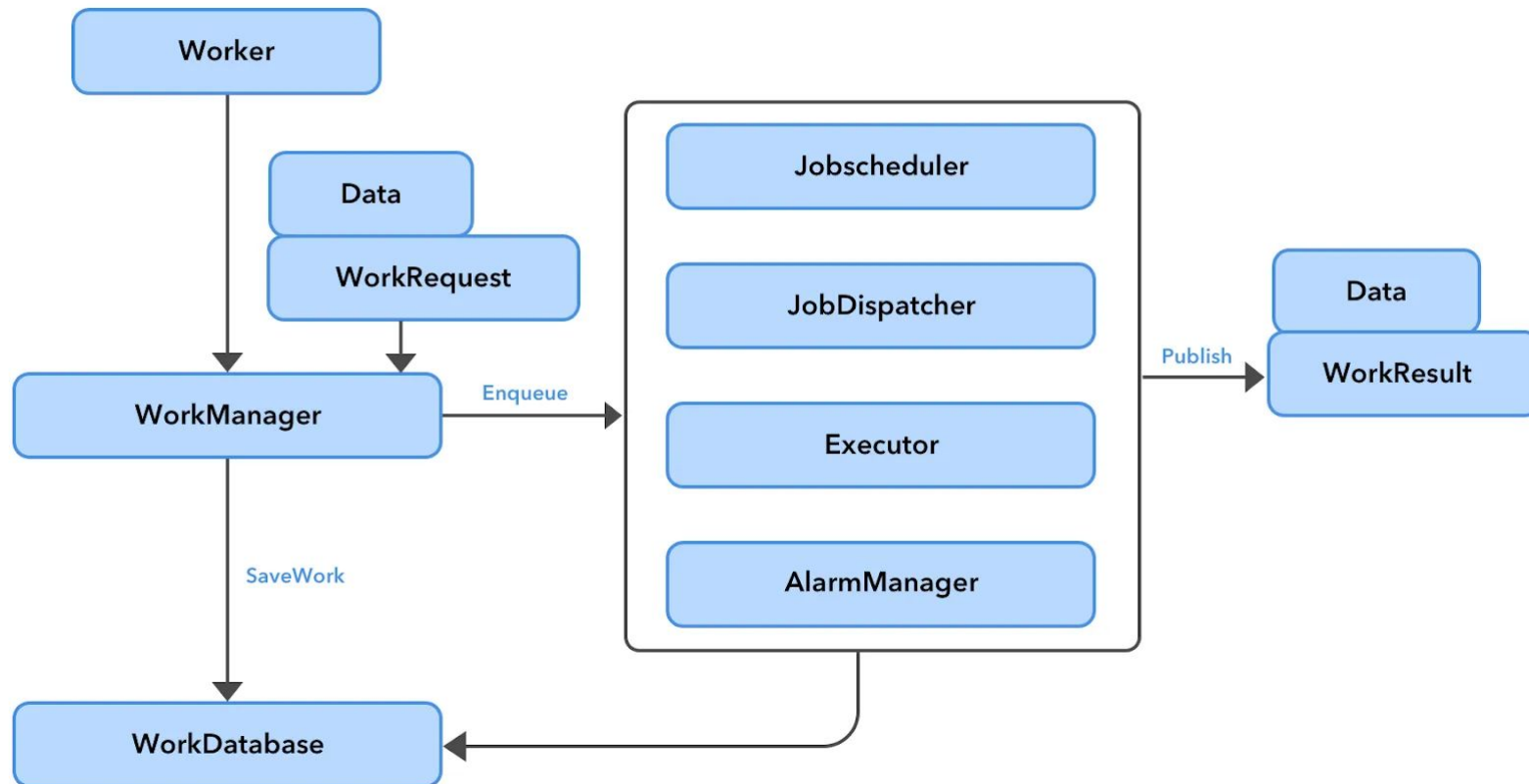


Работа приложений при Doze Mode

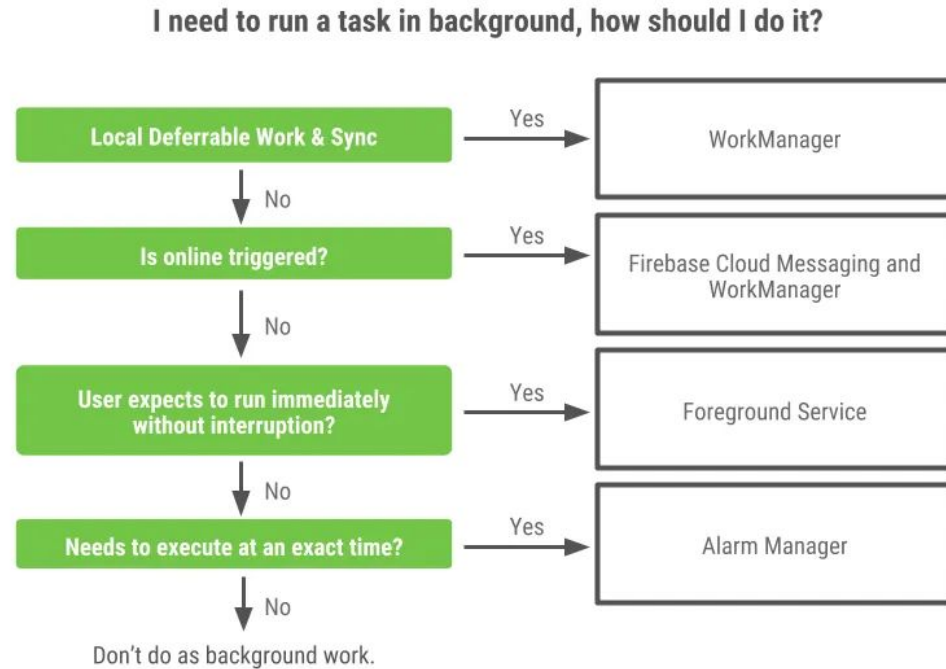


Архитектура WorkManager

WorkManager



Что выбрать для работы?



(картинка актуальна на март 2023 года)

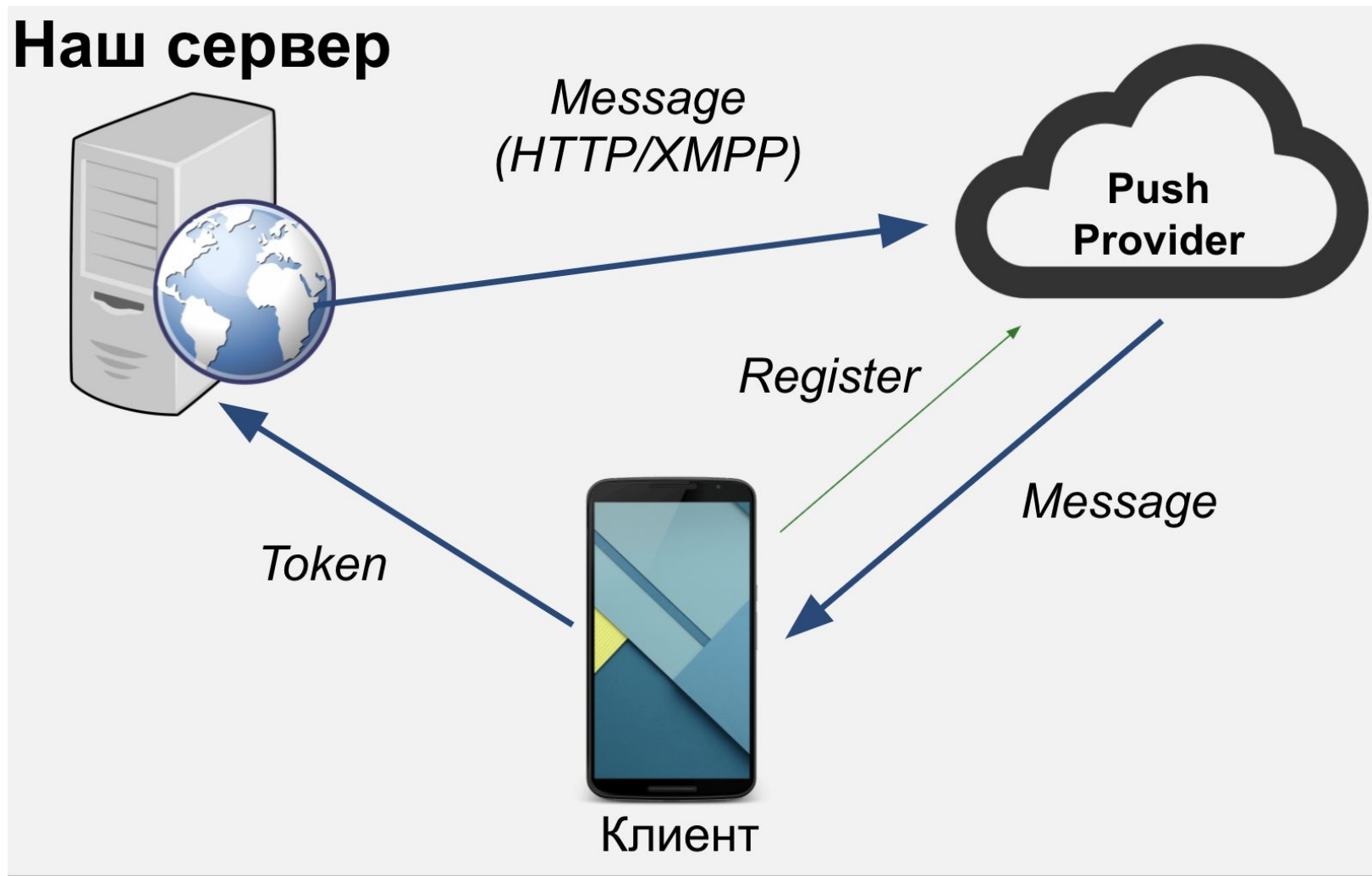
В связи с улучшением безопасности системы, изменением API компонентов, оптимизации потребления ресурсов, надо периодически ознакамливаться с ограничениями.

По прошлому слайду может показаться что WorkManager решает большую часть запросов. Но надо помнить, что это фреймворк, и местами он может быть избыточным.


В лекции упор сделан был в основном на внутренние механизмы Android для работы, чтобы понимать механики, которые могут использовать разные фреймворки (не только WorkManager).

А что если, работа
была сделана на
бэкенде? 0_o

Push



FCM или HMS

 **Analytics**

Dashboard

Realtime

Events

Conversions

Audiences

Funnels

Custom Definitions

Latest Release

Retention

DebugView

Engage

Predictions

SAK Sample ▾

Cloud Messaging

Notifications Reports

!

 Conversion measurement and most targeting options require Google Analytics, which is currently not enabled for this project. Contact your project owner.

🔍 Search by campaign name, message content, campaign status, and platform

Create experiment

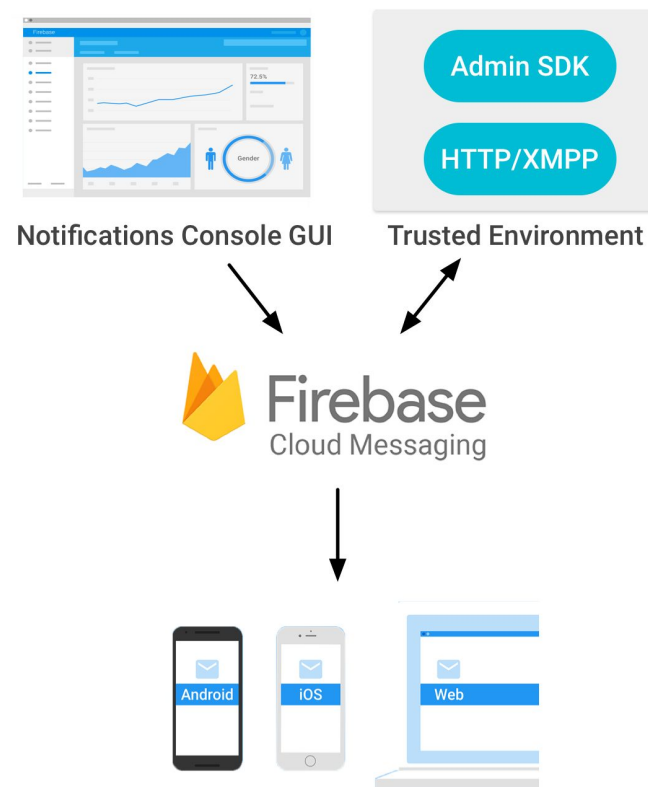
New notification

Notification	Status ?	Platform	Start / Send	End	Sends	Opens
▶ Тест	✓ Completed	🤖	Jan 26, 2021 8:40 PM	—	<1000	0%
↻ 0/10 Recurring notifications ?						

Firebase

- Создать приложение в консоли:
<https://console.firebase.google.com/>
- Настроить приложение в консоли
- Скачать файл google-services.json и положить его в директорию app
- Добавить Firebase SDK в build.gradle
- Создать сервис и добавить в AndroidManifest

...либо воспользоваться Firebase Assistant
(Tools → Firebase)



- Устройство всегда держит коннект к Google серверам
- Отправка происходит через бекенд приложения или из консоли Firebase
- Есть девайсы без FCM. Есть аналоги - HMS от Huawei

Домашнее задание №3

Как выбрать экраны?

- Четко определите какая функциональность есть на экранах. Главное, чтобы один из экранов был “тяжелым” по функциональности.

Если экранов меньше количества человек?

- Придется делать разным людям одинаковые экраны, но надо будет делать в своем стиле:)

Как будет происходить проверка?

- Преподаватели и менторы будут проверять сразу у группы, которая сформирована была на треке проектирования интерфейсов.
- После проверки реализации, будет даваться доп задание исходя из того, что сделал каждый человек.
- Задания будут даваться на основе прошедшего курса.
- На защите можно будет пользоваться чем угодно. Только не просите код писать за вас:)

kotlin level 1

Домашнее задание №3

“Верстка”

Постановка задачи

Из домашнего задания по интерфейсам, требуется реализовать минимум 2 экрана. Для наполнения экрана требуется использовать собственные объекты с данными, которые потом мапятся на поля верстки.

Ограничения и требования

1. Требуется использовать Fragment.
2. Приложение не должно содержать хардкод.
3. Приложение должно использовать ресурсы(resources) для работы
4. В коде можно оставлять комментарии, но в конечной версии нельзя оставлять Log
5. Между реализованными экранами должны быть переходы, или любой другой способ попасть на эти экраны после запуска приложения.
6. Данные экранов нельзя зашивать в верстке, они должны мапиться в верстку при помощи объектов с данными
7. Данные поставляться на экраны должны через не-синхронные вызовы, и задержка для получения данных должна быть в 2 секунды и иметь возможность ее увеличить (в коде).
8. Экраны или блоки с данными, должны поддерживать состояние загрузки.

Что будет плюсом

1. Поддержка состояния ошибки всеми экранами и блоками
2. Поддержка пустого состояния блоками с данными и экранами
3. Реализация больше трех экранов
4. Использование API для получения данных для экранов.

Спасибо за
внимание!

