

Тестирование и отладка приложений

Введение

Тестирование и отладка являются критически важными этапами в процессе разработки мобильных приложений. Эти процессы не только помогают выявлять и исправлять ошибки, но также способствуют повышению производительности и обеспечению безопасности приложения. Далее рассмотрим ключевые методы и инструменты для тестирования и отладки, а также коснемся профилирования приложений и важных аспектов их безопасности.





Виды тестирования в Android

В Android существуют различные виды тестирования, которые помогают обеспечить качество и надежность приложения:

Юнит-тестирование (Unit Testing)

Проверяет отдельные модули или функции приложения, используя такие инструменты, как JUnit и Mockito.

Интеграционное тестирование (Integration Testing)

Проверяет взаимодействие между различными модулями приложения.

UI-тестирование (UI Testing)

Проверяет пользовательский интерфейс и взаимодействие пользователя с приложением. Инструменты: Espresso, UI Automator.

Пример: Unit-теста

AverageCalculator.kt

Kotlin

```
class AverageCalculator {  
    fun calculateAverage(numbers: List<Double>): Double? {  
        if (numbers.isEmpty()) return null  
        return numbers.sum() / numbers.size  
    }  
}
```

AverageCalculatorTest.kt

Kotlin

```
class AverageCalculatorTest {  
    private val calculator = AverageCalculator()  
  
    @Test  
    fun testCalculateAverage_withNonEmptyList() {  
        val numbers = listOf(10.0, 20.0, 30.0)  
        val result = calculator.calculateAverage(numbers)  
        assertEquals(20.0, result, 0.001)  
    }  
  
    @Test  
    fun testCalculateAverage_withEmptyList() {  
        val numbers = emptyList<Double>()  
        val result = calculator.calculateAverage(numbers)  
        assertNull(result)  
    }  
  
    @Test  
    fun testCalculateAverage_withSingleElement() {  
        val numbers = listOf(42.0)  
        val result = calculator.calculateAverage(numbers)  
        assertEquals(42.0, result, 0.001)  
    }  
}
```

Пример: UI-теста (XML & Activity)

MainActivity.kt

Kotlin

```
class MainActivity : AppCompatActivity() {  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        setContentView(R.layout.activity_main)  
  
        val editTextUsername = findViewById<EditText>(R.id.editTextUsername)  
        val buttonSubmit = findViewById<Button>(R.id.buttonSubmit)  
        val textViewGreeting = findViewById<TextView>(R.id.textViewGreeting)  
  
        buttonSubmit.setOnClickListener {  
            val username = editTextUsername.text.toString()  
            textViewGreeting.text = "Hello, $username!"  
        }  
    }  
}
```

activity_main.xml

XML

```
<LinearLayout  
    xmlns:android="http://schemas.android.com/apk/res/android"  
    android:layout_width="match_parent"  
    android:layout_height="match_parent"  
    android:orientation="vertical"  
    android:padding="16dp">  
  
    <EditText  
        android:id="@+id/editTextUsername"  
        android:layout_width="match_parent"  
        android:layout_height="wrap_content"  
        android:hint="Enter your name"/>  
  
    <Button  
        android:id="@+id/buttonSubmit"  
        android:layout_width="wrap_content"  
        android:layout_height="wrap_content"  
        android:text="Submit"/>  
  
    <TextView  
        android:id="@+id/textViewGreeting"  
        android:layout_width="match_parent"  
        android:layout_height="wrap_content"  
        android:textSize="22sp"/>  
  
</LinearLayout>
```

Пример: UI-теста

MainActivityUITest.kt

Kotlin

```
@RunWith(AndroidJUnit4::class)
class MainActivityUITest {
    @get:Rule
    var activityRule = ActivityScenarioRule(MainActivity::class.java)

    @Test
    fun greetUser_correctly() {
        // Вводим имя "John" в EditText
        onView(withId(R.id.editTextUsername)).perform(typeText("John"))

        // Нажимаем на кнопку "Submit"
        onView(withId(R.id.buttonSubmit)).perform(click())

        // Проверяем, что TextView содержит текст "Hello, John!"
        onView(withId(R.id.textViewGreeting)).check(matches(withText("Hello, John!")))
    }
}
```

Пример: UI-теста (Compose screen)

GreetingScreen.kt

Kotlin

```
@Composable
fun GreetingScreen() {
    var name by remember { mutableStateOf("") }
    var greeting by remember { mutableStateOf("") }

    Column(modifier = Modifier.padding(16.dp)) {
        TextField(
            value = name,
            onValueChange = { name = it },
            label = { Text("Enter your name") },
            modifier = Modifier.fillMaxWidth()
        )
        Button(
            onClick = { greeting = "Hello, $name!" },
            modifier = Modifier.padding(top = 8.dp)
        ) {
            Text("Submit")
        }
        if (greeting.isNotEmpty()) {
            Text(
                text = greeting,
                modifier = Modifier.padding(top = 8.dp),
                style = MaterialTheme.typography.h6
            )
        }
    }
}
```

Пример: UI-теста

GreetingScreenTest.kt

Kotlin

```
class GreetingScreenTest {

    @get:Rule
    val composeTestRule = createComposeRule()

    @Test
    fun greetUser_displaysGreeting() {
        composeTestRule.setContent {
            GreetingScreen()
        }

        // Вводим имя "John" в текстовое поле
        composeTestRule.onNodeWithText("Enter your name").performTextInput("John")

        // Нажимаем на кнопку "Submit"
        composeTestRule.onNodeWithText("Submit").performClick()


        // Проверяем, что на экране отображается текст "Hello, John!"
        composeTestRule.onNodeWithText("Hello, John!").assertIsDisplayed()
    }
}
```


Отладка (Debugger)

Отладчик в Android Studio является мощным инструментом, который позволяет разработчикам детализировано анализировать выполнение кода приложения.

С его помощью можно:

- установить breakpoints для временной приостановки выполнения программы
- пошагово проследить выполнение кода
- проверить и изменить значения переменных
- и многое другое



Возможности отладчика

Точки останова (Breakpoints)

Устанавливаются в ключевых местах кода для остановки выполнения и анализа состояния приложения.

1

Пошаговое выполнение (Stepping):

Позволяет построчно выполнить код, входить в функции или выходить из них, что помогает точно локализовать источники ошибок.

2

Просмотр переменных:

В реальном времени можно просматривать и изменять значения переменных, что помогает понять, как данные изменяются в процессе выполнения.

3

Анализ стека вызовов (Call Stack):

Отображает последовательность вызовов функций и методов, помогая разработчикам понять поток выполнения приложения.

4

Начало отладки

Перед началом процесса отладки кода необходимо:

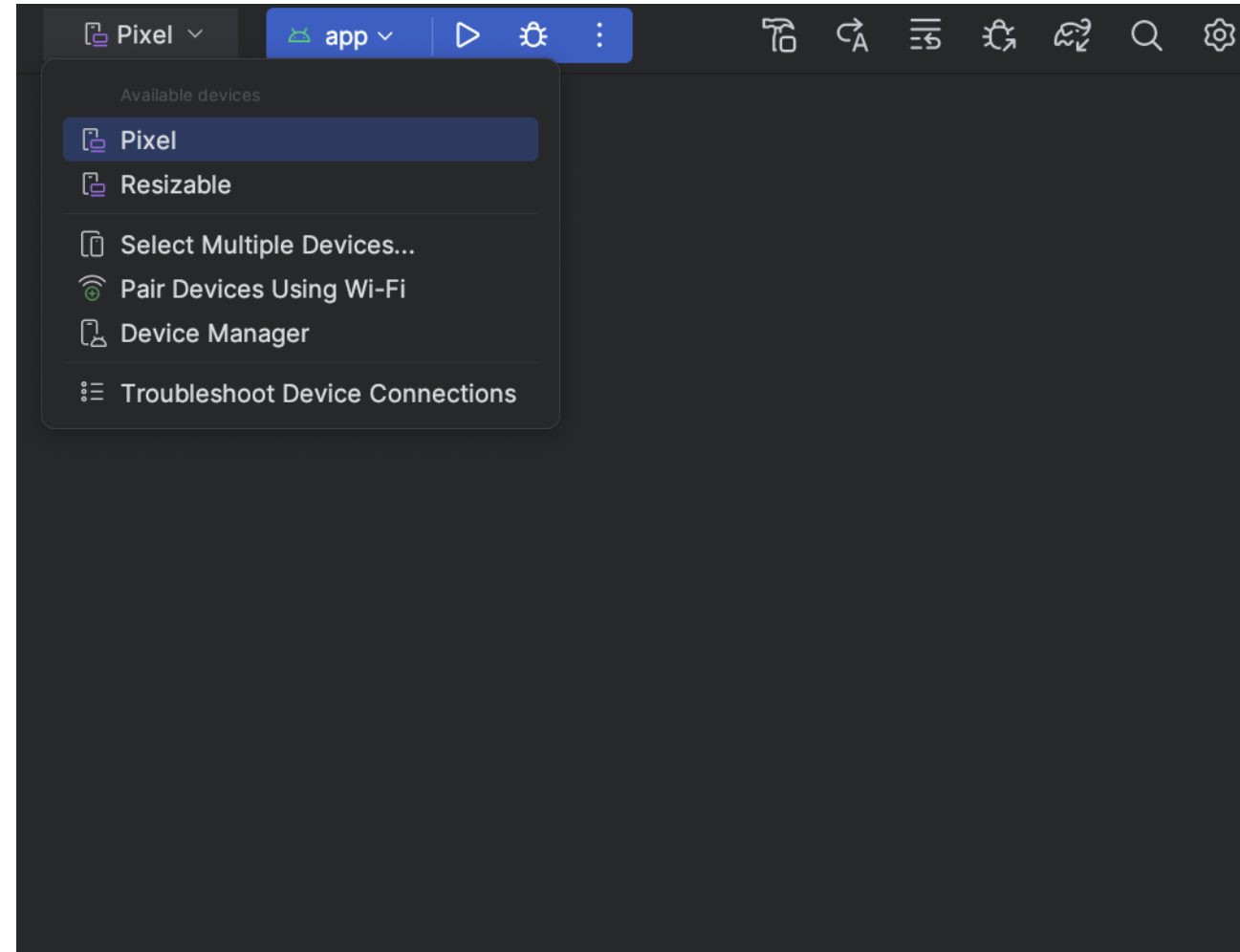
1. Подготовить устройство и проект:

- Включить отладку на устройстве (в developers options)
- Включить флаг debuggable в проекте

2. Выбрать устройство или эмулятор;

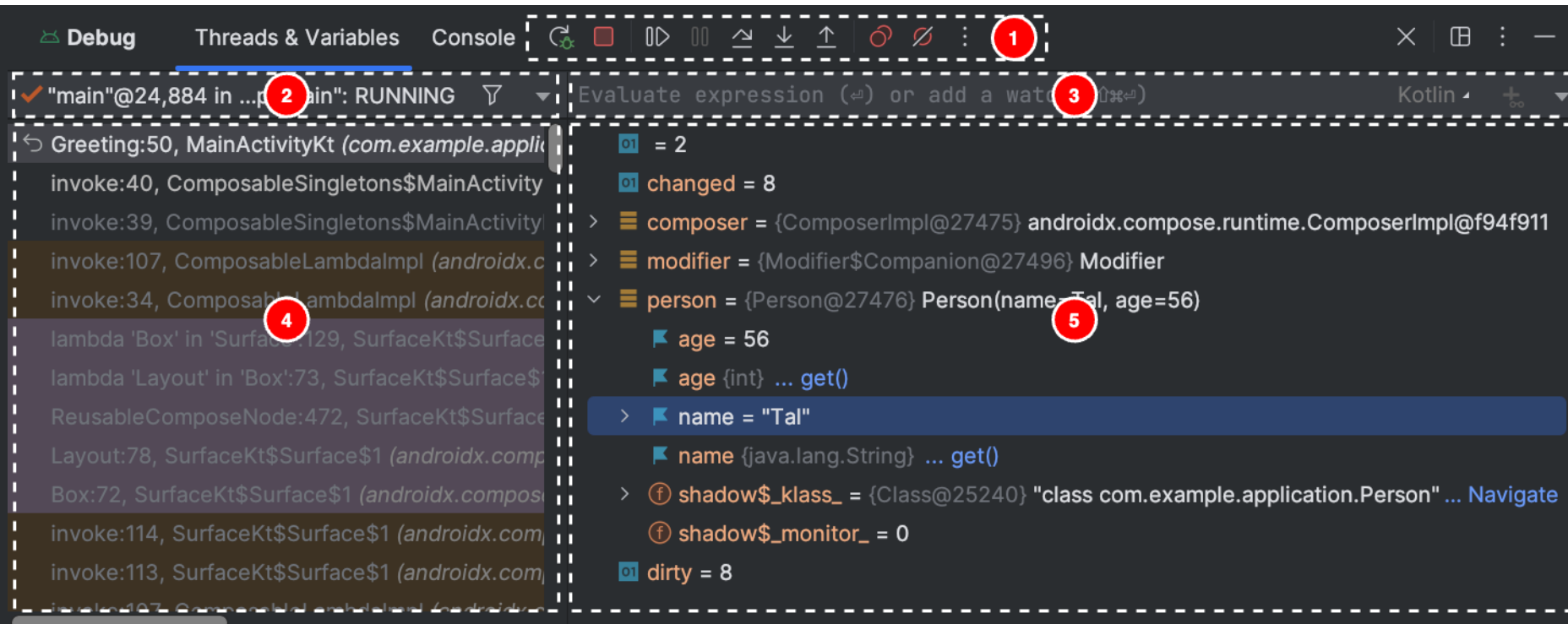
3. Поставить *breakpoints* в коде;

4. Воспользоваться приложением (дойти до определенного участка кода)



Окно отладки

1. Панель инструментов и навигации
2. Селектор потоков
3. Вычисления и наблюдение за переменными
4. Стек вызовов
5. Переменные



Профилировщик (Profiler)

Android Profiler — это инструмент для анализа производительности приложения. Он позволяет измерять показатели в реальном времени.

Основные функции:

CPU Profiler: Анализ производительности процессора, выявление «узких мест», оптимизация алгоритмов.

Memory Profiler: Мониторинг использования памяти, выявление утечек памяти, анализ сборки мусора (GC).

Network Profiler: Анализ сетевой активности приложения, измерение скорости загрузки и передачи данных.

Energy Profiler: Анализ потребления энергии приложением, что важно для оптимизации времени работы устройства от батареи.




Android Profiler

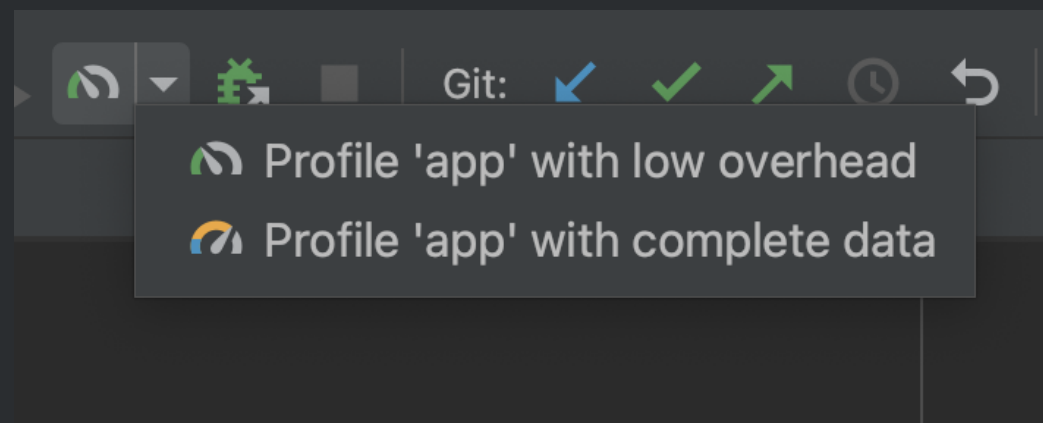
Подготовка к профилированию

Для запуска профилировщика нужно выполнить ряд требований:

1. Наличие устройства или эмулятора уровня *API 29* или выше
2. Наличие *Google Play* на устройстве или эмуляторе

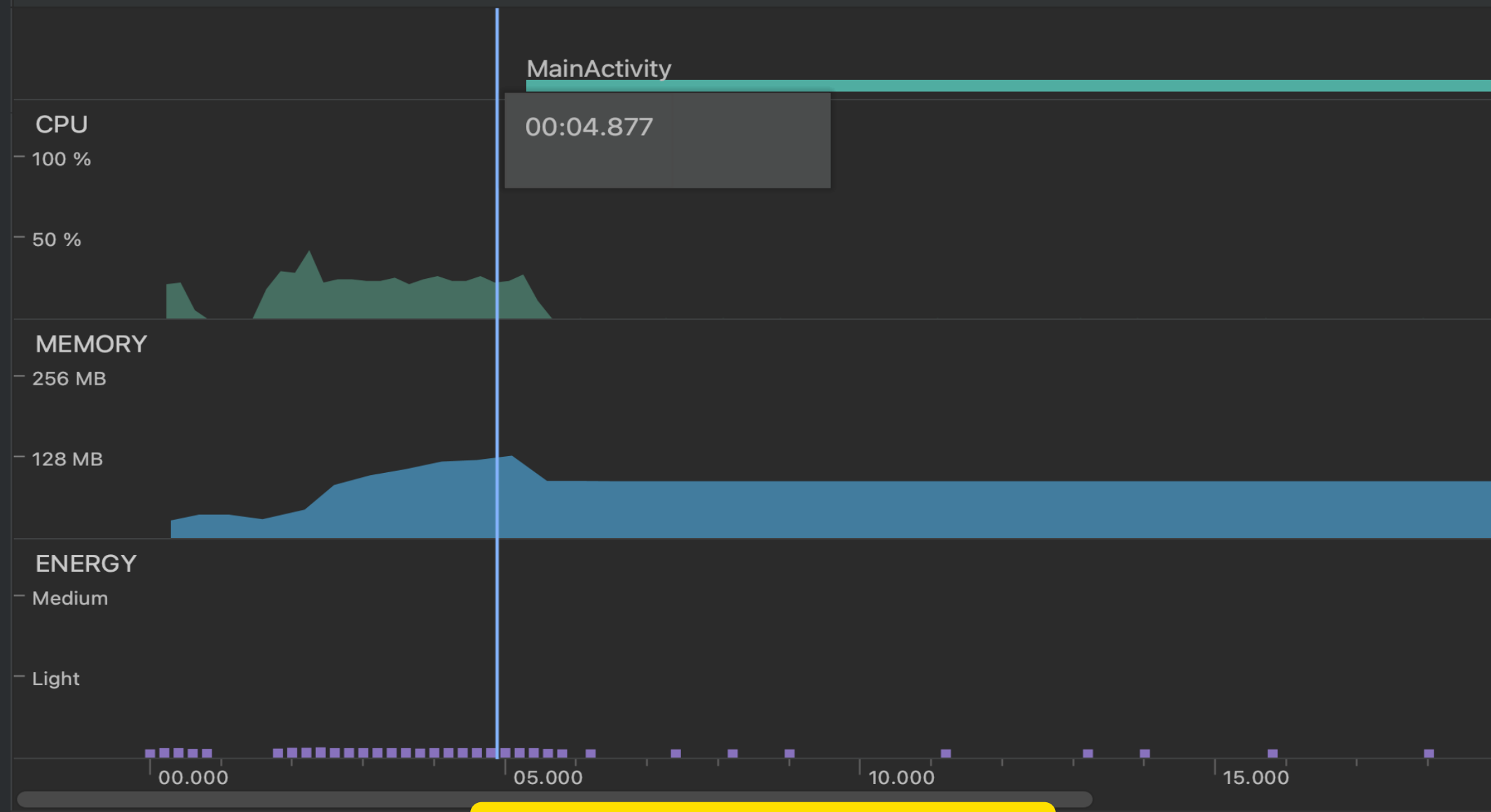
Запускается профилировщик по иконке  в тулбаре. С выбором необходимой опции:

- **low overhead** запускает профилировщики CPU и Memory
- **complete data** запускают профилировщики CPU, Memory и энергопотребления

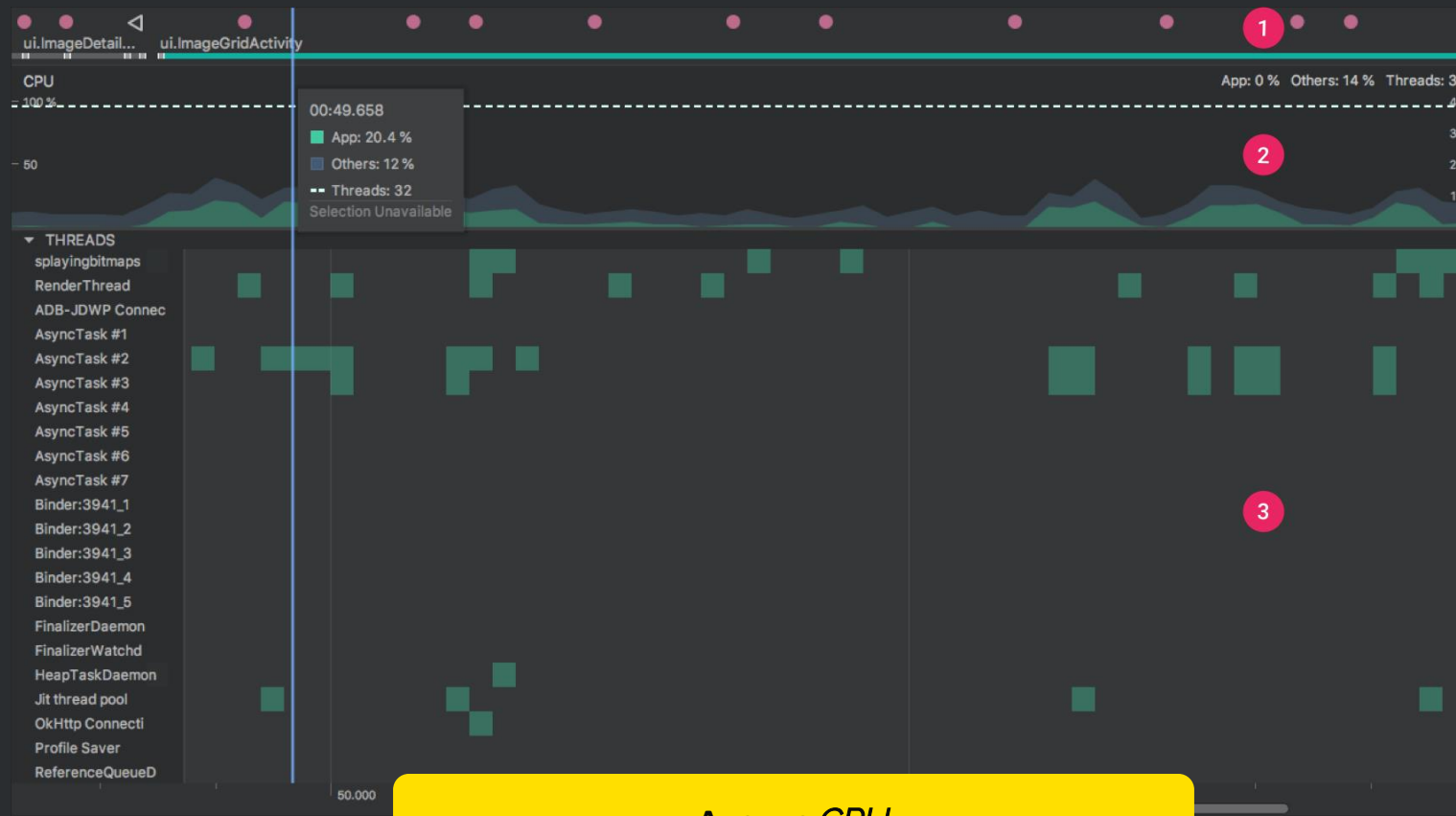


(Google Pixel_4_API_30)

Profiling with complete data. This does not represent app performance in production. Consider profiling with low overhead.



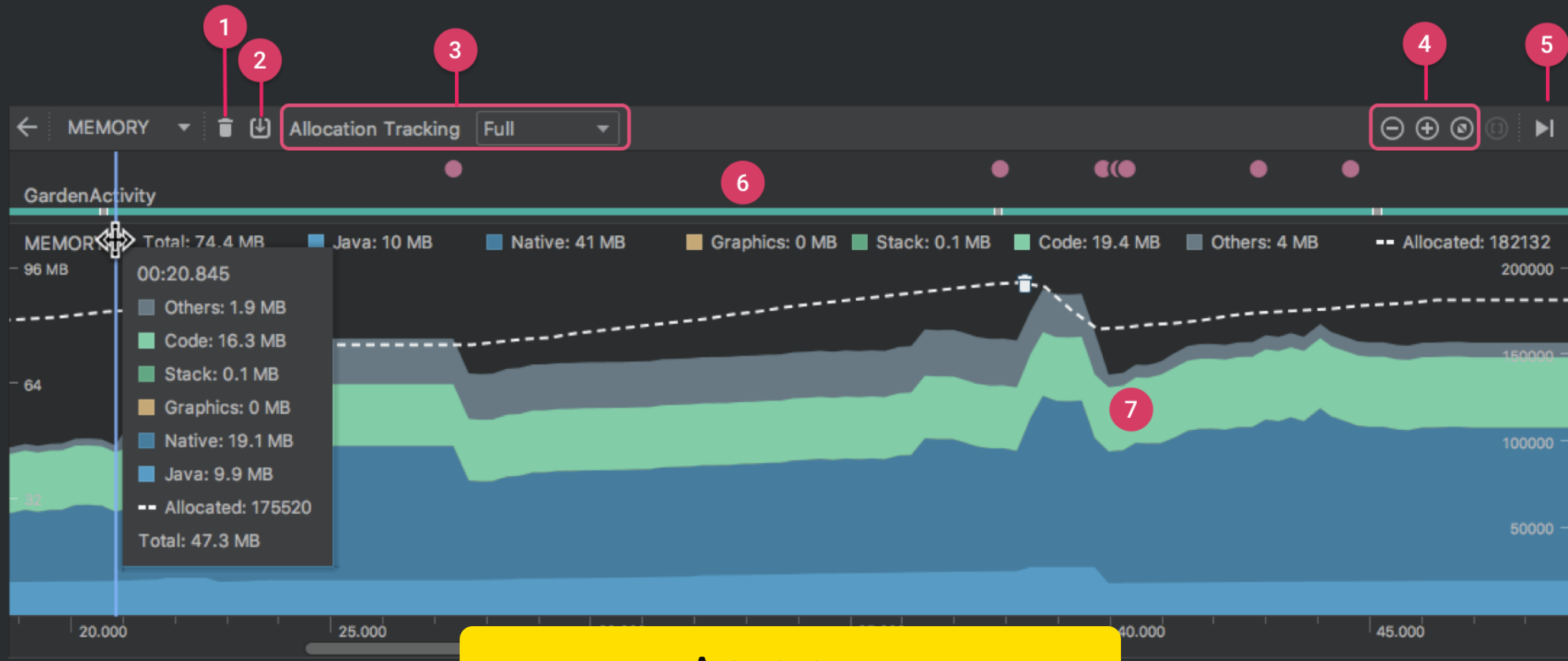
Окно профилировщика



1. **Шкала событий:** показывает действия в вашем приложении (жизненный цикл и взаимодействие пользователя с устройством)

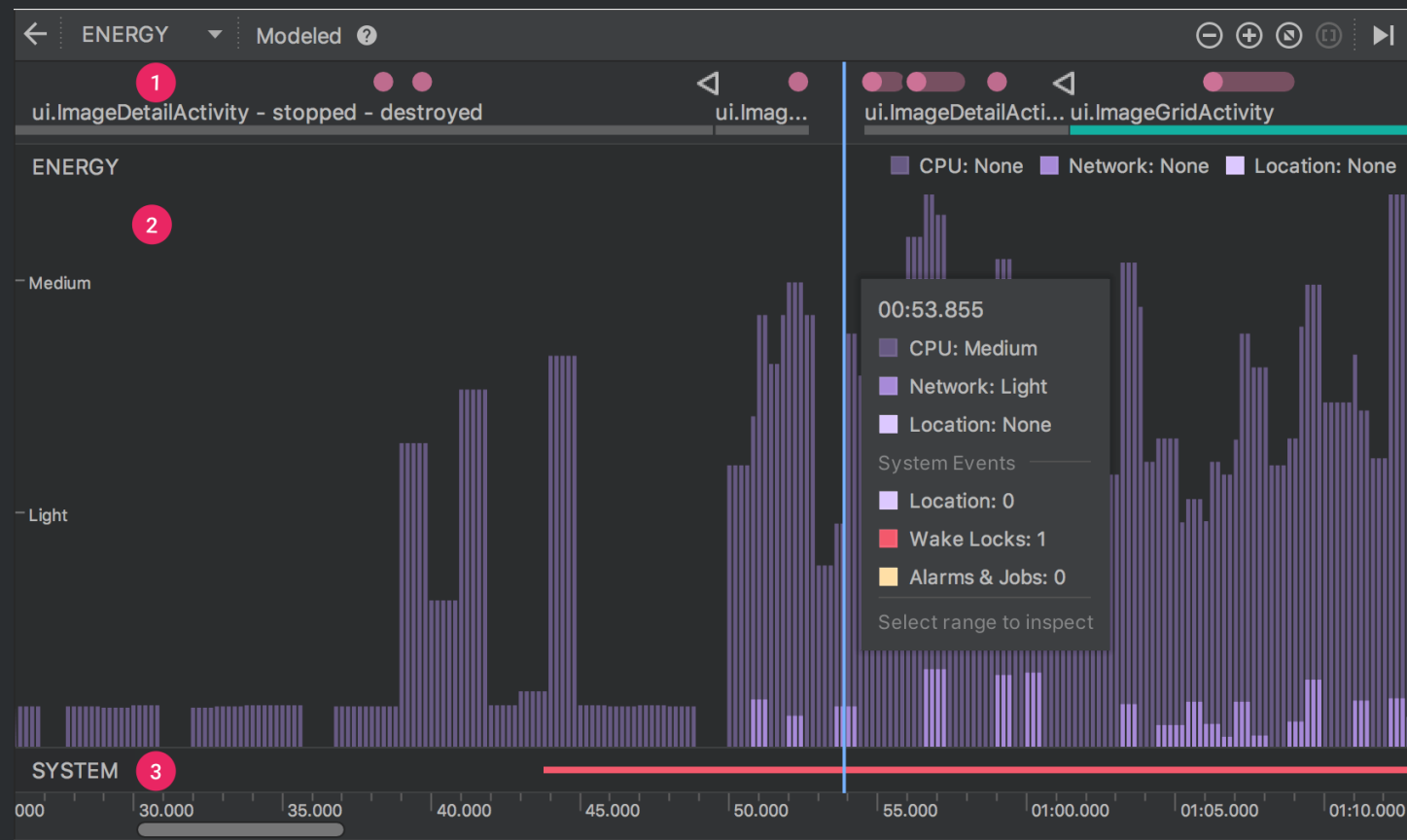
2. **Шкала ЦП:** показывает использование ЦП в реальном времени вашим приложением — в процентах от общего доступного времени ЦП

3. **Шкала активности потоков:** перечисляет каждый поток, принадлежащий процессу приложения и указывает его активность с помощью различных цветов



1. Принудительная очистка мусора (GC)
2. Захват дампа памяти
3. Меню выбора частоты распределение памяти
4. Увеличения/уменьшения масштаба шкалы.

5. Переход к текущим данным памяти.
6. Шкала событий, на которой показаны состояния активности, события пользовательского ввода и события поворота экрана.
7. Шкала использования памяти



Анализ потребления энергии

1. Шкала событий: показывает события жизненного цикла и взаимодействие пользователя с устройством

2. Шкала энергопотребления: показывает предполагаемое потребление энергии вашим приложением.

3. Шкала системы: показывает системные события, которые могут повлиять на потребление энергии.



True Engineering
630128, г. Новосибирск,
ул. Кутателадзе, 4г
(383) 363-33-51, 363-33-50
info@trueengineering.ru
trueengineering.ru

**Новосибирский
Государственный
Университет**