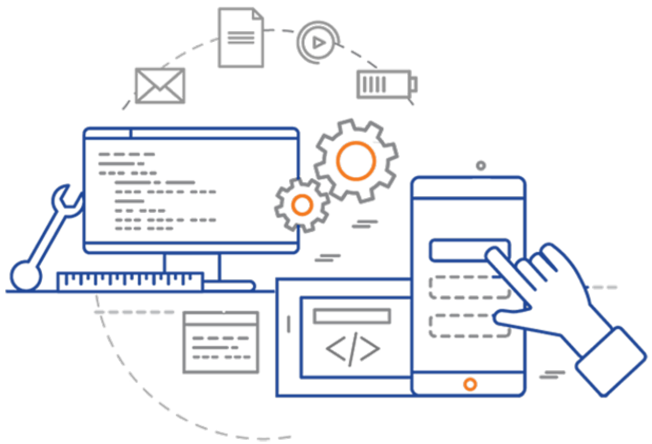


CS 398: Application Development

Evaluation

Non-functional requirements; Measuring and logging data.

Non-Functional Requirements

To this point, we've focused on testing the functional requirements and the correctness of our results.

However, there's other criteria: the non-functional requirements (NFRs) or properties of our system that we defined during the requirements gathering phase.

NFRs are defined as significant, *measurable* characteristics of a system. e.g.

- **Processing speed**: how long particular computations or functions take to execute.
- **Volume of data**: images processed, calculations performed and so on.
- **Network performance**: how long transmitting, receiving data over a network takes.
- **Memory consumed**: peak, and average memory consumption.
- **Startup time**: how long it takes to launch.

Collecting Data

System metrics

The Kotlin and JDK libraries contains a number of useful functions that can help us collect system information.

Package	Function	Purpose
kotlin.system	exitProcess()	Terminate the currently running process.
kotlin.system	getTimeMillis()	Gets system time; subtract two points to get elapsed time in milliseconds.
kotlin.system	measureTimeMillis { }	Executes the given block (lambda) and returns elapsed time in milliseconds.
java.lang	Runtime.getRuntime().freeMemory()	Free memory for JVM.
java.lang	Runtime.getRuntime().totalMemory()	Total available memory for JVM.
java.io	File("/").freeSpace	Free bytes in a directory.

Typically you want to capture data before and after a critical operation, and then compare them to determine the cost of that operation (e.g. time to execute, memory consumed).

If you are measuring time, the `measureTimeMillis` function from the `kotlin-stdlib` is particularly helpful since it takes a lambda argument which is the block of code to execute.

```kotlin
fun performLengthyComputation() {
    // a lot of processing goes on here
}

// manual
val start = getTimeMillis()
performLengthyComputation()
val end = getTimeMillis()
println("The elapsed time in milliseconds is ${end-start}")

// using kotlin-stdlib
val elapsed = measureTimeMillis { performLengthyComputation() }
println("The elapsed time in milliseconds is $elapsed")
```

Tips

You are going to want to measure everything that you listed in your non-functional requirements.

- You need to exercise the functions that you want to measure: run the application, and interact with the features that you care about testing.
- If the functionality that you want to measure spans many classes or functions, you may need to write a top-level function to encapsulate that behaviour so that it's easier to measure.
    - Milliseconds is probably the most coarse-grained you want to measure; there are also functions to measure microseconds but your tests will not be accurate to that level.
- Keep in mind that your system has other things running that will consume memory, drive space and so on. It is difficult/impossible to completely isolate your application.
    - Test multiple times, and collect data from each attempt - average the results to get a more accurate estimate the value that you are collecting, which will reduce the impact of other applications that may be running on the system and consuming resources.

Logging Data

Logging Data

You will want to gather the information in a text file or database, so that you can examine and analyze it later.

You should record each data point that you collect as a single row in your log file. If you need to analyze it, this makes it much easier.

Each row should contain at least the following:

- **Timestamp**: Measure events and record time in milliseconds.
- **Event type**: this is more useful if you use logging for debugging purposes (where you can setup INFO, WARNING, EXCEPTION categories of messages). Use a default option here e.g. INFO.
- **Description**: Include a description of what your are logging. e.g. "time to execute bigFunction()"
- **Value**: Capture the numeric value in question. e.g. time in milliseconds.

```
var file: FileWriter? = null
var writer: BufferedWriter? = null

data class Entry(val type: TYPE, val description: String, val value: Int) {
   override fun toString(): String = “${getCurrentDateTime().toString()}, $type, $description, $value"
}
enum class TYPE { DEBUG, LOG}

fun Date.toString(format: String): String = SimpleDateFormat(format, Locale.getDefault()).format(this)
fun getCurrentDateTime(): Date = Calendar.getInstance().time

fun open(filename: String) {
    file = FileWriter(filename)
    writer = BufferedWriter(file)
}

fun debug(description: String, value: Int) = save(Entry(TYPE.DEBUG, description, value))
fun log(description: String, value: Int) = save(Entry(TYPE.LOG, description, value))
private fun save(entry: Entry) = writer?.write(entry.toString() + "\n")

fun close() {
    writer?.close()
    file?.close()
}
```

Logging Classes

Kotlin does not have a built-in logging class, but there are some options:

- Android has a Log class which is extremely robust and integrates with the development environment.
- Desktop/JDK users can use the Java Logging API. Although it's a Java library, its easy to use and compatible with Kotlin.
- There are also third-party solutions like Log4J that are popular for commercial environments.

```
import java.util.logging.*

object LoggerExample {
    private val LOGGER = Logger.getLogger(LoggerExample::class.java.getName())

    @JvmStatic
    fun main(args: Array<String>) {
        // Set handlers for both console and log file
        val consoleHandler: Handler? = ConsoleHandler()
        val fileHandler: Handler? = FileHandler("./example.log")
        LOGGER.addHandler(consoleHandler)
        LOGGER.addHandler(fileHandler)

        // Set to filter what gets logged at each destination
        consoleHandler?.setLevel(Level.ALL)
        fileHandler?.setLevel(Level.ALL)
        LOGGER.level = Level.ALL

        // Log some data
        LOGGER.info("Information")
        LOGGER.config("Configuration")
        LOGGER.warning("Warning")

        // Console handler removed
        LOGGER.removeHandler(consoleHandler)

        // This should still go to the log file
        LOGGER.log(Level.FINE, "Finer logged without console")
    }
}
```

Multiple outputs

Filter events

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE log SYSTEM "logger.dtd">
<log>
<record>
  <date>2022-03-08T22:13:55.289391Z</date>
  <millis>1646777635289</millis>
  <nanos>391000</nanos>
  <sequence>0</sequence>
  <logger>LoggerExample</logger>
  <level>INFO</level>
  <class>LoggerExample</class>
  <method>main</method>
  <thread>1</thread>
  <message>Information</message>
</record>
</log>
```

XML is the default for this library.

Profiling

Real-time Profiling

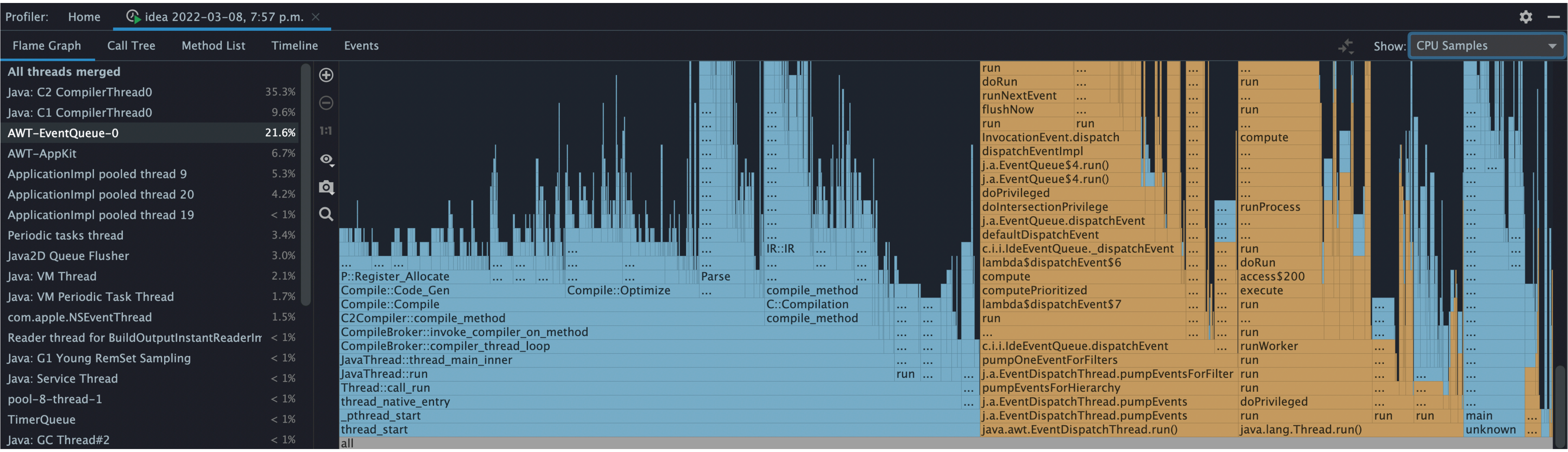
Finally, IntelliJ IDEA and Android Studio both include real-time profiling tools that can help you detect performance issues.

To launch the profiler in IntelliJ IDEA:

- Run - Run
- Run - Attach Profiler to Process...

This will launch your application.

IntelliJ IDEA will collect data from your application as it's running. Stop it after you have exercised your functions “enough times”.



All threads merged	
Java: C2 CompilerThread0	35.3%
Java: C1 CompilerThread0	9.6%
AWT-EventQueue-0	21.6%
AWT-AppKit	6.7%
ApplicationImpl pooled thread 9	5.3%
ApplicationImpl pooled thread 20	4.2%
ApplicationImpl pooled thread 19	< 1%
Periodic tasks thread	3.4%
Java2D Queue Flusher	3.0%
Java: VM Thread	2.1%
Java: VM Periodic Task Thread	1.7%
com.apple.NSEventThread	1.5%
Reader thread for BuildOutputInstantReaderIm	< 1%
Java: G1 Young RemSet Sampling	< 1%
Java: Service Thread	< 1%
pool-8-thread-1	< 1%
TimerQueue	< 1%
Java: GC Thread#2	< 1%

Woah.