You

did

WHAT?!?

What

were

you

THINKING?!?!?!?

Go on…

This is a safe place

Lessons learned building a build system

Cédric Beust

cedric@beust.com

VP of Engineering at Samsung SmartThings

Table of contents

1. Why? WHY?
2. Whaaaaaat?
3. But… how?

Why? WHY?

What led to Kobalt?

- Dissatisfaction with existing build tools
- Felt the need to be in control of my build tool
- Envisioned I might encounter interesting challenges along the way

… but really

- Gave me a good excuse to write a lot of Kotlin

Whaaaaaat?

What is Kobalt?

```
val VERSION = “1.52”

val jcommander = project {
    name = "jcommander"
    group = "com.beust"
    artifactId = name
    version = VERSION

    dependenciesTest {
        compile("org.testng:testng:")
    }

    assemble {
        mavenJars {}
    }

    bintray {
        publish = false
    }
}
```

Design goals

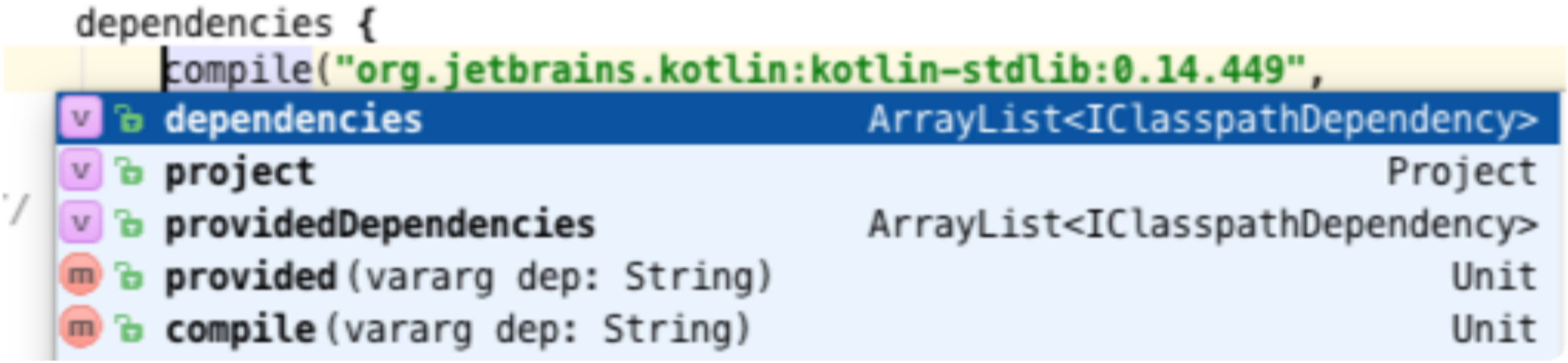
- Written 100% in Kotlin, core and build file

- A set of features coming up in the next slides

Build file

- 100% valid Kotlin code (type safe builders)
- Use Kotlin mechanisms for everything (e.g. profiles)
- Emphasis on making the build file syntax intuitive
- Maven repo information
- Reuse a lot of ideas from Gradle, invent a few new ones

Auto completion



```
dependencies {
    compile("org.jetbrains.kotlin:kotlin-stdlib:0.14.449",
```

```
v  dependencies                          ArrayList<IClasspathDependency>
v  project                                                       Project
v  providedDependencies                  ArrayList<IClasspathDependency>
m  provided (vararg dep: String)                                    Unit
m  compile (vararg dep: String)                                     Unit
```

Incremental tasks

```
───── kobalt:compile
  Kotlin 1.1.2 compiling 182 files
  Actual files that needed to be compiled: 7
───── kobalt:assemble
  Output and input hashes identical, skipping this task
```

Note: build tasks are opaque, individual tasks can additionally implement incremental runs on their own

Parallel builds

```
PARALLEL BUILD SUCCESSFUL (25 SECONDS),
sequential build would have taken 43 seconds
```

Profiles

```
val experimental = false

val p = project {
    name = if (experimental) "project-exp" else "project"
    version = "1.3"
```

Enabling profiles:

```
$ ./kobaltw --profiles experimental assemble
```

Easy project dependencies

```
val lib1 = project {
  name = “network”
  // …
}

val lib2 = project {
  name = “authentication”
  // …
}

val mainApp = project(lib1, lib2) { …
```

Plug-in architecture

- Statically typed
- Extension points (similar to Eclipse, IDEA)
- Clearly separates Kobalt’s core from plug-ins
- Hollywood principle

Other features

- Multiple testing frameworks (TestNG, JUnit 4, JUnit 5, Spock, …)IDEA plug-in
- Built-in Maven repo uploads
- Templates
- ASCII art and animations
- Variants and flavors
- Multi language
- Tasks inside the build file
- Self updating
- Version checks:

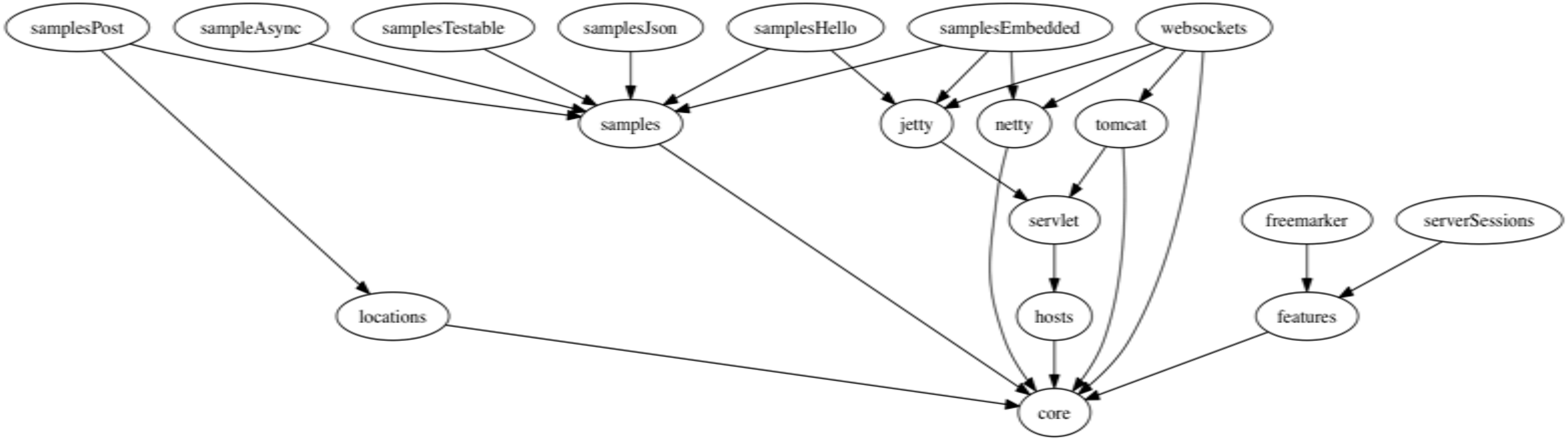
```
$ ./kobaltw --checkVersions
New versions found:
        org.testng:testng:6.12
        org.jetbrains.kotlin:kotlin-test:1.1.51
        com.squareup.okhttp:okhttp:3.9.0
```

But… how?

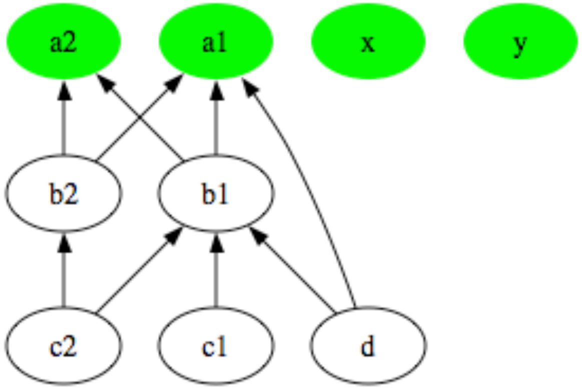
Parallel builds with DynamicGraph

Efficient parallelism for graph processing

Example project (ktor)

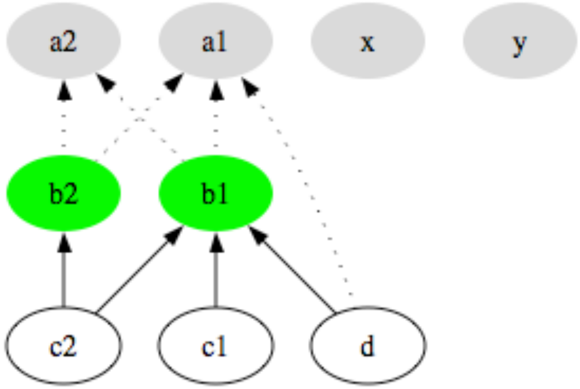


Topological sort



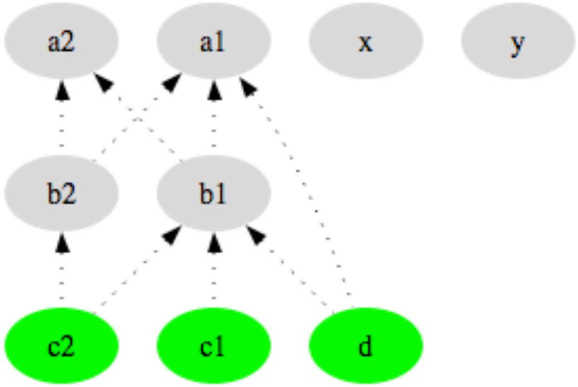
a2, a1, x, y

Topological sort



a2, a1, x, y, b2, b1

Topological sort

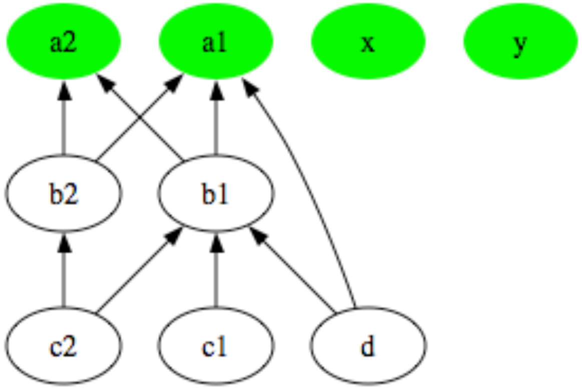


(a2, a1, x, y), (b2, b1), (c2, c1, d)

Single threaded

Order of invocation:

a2, a1, x, y, b2, b1, c2, c1, d



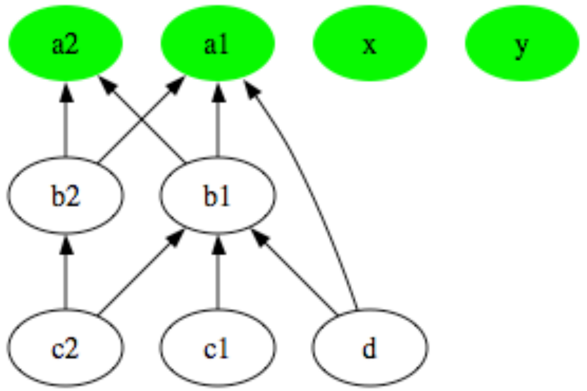
Multithreaded (naïve)

Two thread pools:

- One for free nodes (n threads)
- One for dependent nodes (1 thread)

Free nodes: x, y

Dependent nodes: a2, a1, b2, b1, c2, c1, d



Multithreaded (DynamicGraph)

One thread pool (n threads)

Recalculate free nodes at each completion

Launch a1, a2, x, y

(a2 completes)

(a1 completes)

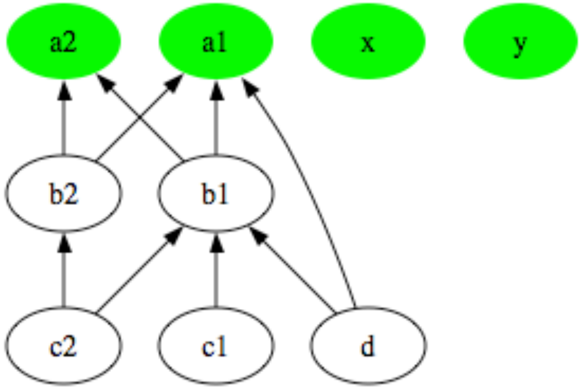
Launch b2, b1

(b1 completes)

Launch c, d

(b2 completes)

Launch c2



DynamicGraph algorithm

```
freeNodes = graph.freeNodes
do {
  schedule each free node in the thread pool
  wait for the next node to complete
  remove that node from the graph
  freeNodes = graph.freeNodes
} while (! freeNodes.isEmpty)
```

- Not shown: cycle handling, waiting for completion, time outs, ...
- No need for each task to explicitly wait for its dependents
- `DynamicGraph` and `DynamicGraphExecutor` are completely generic

DynamicGraph in action

```
╔══════════════════════════════════════════════════════════════════════════════════════════════════════
║  Time (sec) ║ Thread 39           ║ Thread 40           ║ Thread 41           ║ Thread 42           ║
╠═════════════════════════════════════════════════════════════════════════════════════════════════════
║  0          ║ core                ║                     ║                     ║                     ║
║  45         ║ core (45)           ║                     ║                     ║                     ║
║  45         ║                     ║ ktor-locations      ║                     ║                     ║
║  45         ║                     ║                     ║ ktor-netty          ║                     ║
║  45         ║                     ║                     ║                     ║ ktor-samples        ║
║  45         ║ ktor-hosts          ║                     ║                     ║                     ║
║  45         ║                     ║                     ║                     ║                     ║
║  45         ║ ktor-hosts (0)      ║                     ║                     ║                     ║
║  45         ║ ktor-servlet        ║                     ║                     ║                     ║
║  45         ║                     ║                     ║                     ║                     ║
║  45         ║                     ║                     ║                     ║ ktor-samples (0)    ║
║  45         ║                     ║                     ║                     ║ ktor-freemarker     ║
...
╚═════════════════════════════════════════════════════════════════════════════════════════════════════╝

PARALLEL BUILD SUCCESSFUL (68 seconds, sequential build would have taken 97 seconds)
```

Parallel logging in Kobalt

The problem

"How to reconcile parallel execution with sequential logging?"

```
            _  __          _               _   _
           | |/ /   ___   | |__     __ _  | | | |_
           | ' /   / _ \  | '_ \   / _` | | | | __|
           | . \  | (_) | | |_) | | (_| | | | | |_
           |_|\_\  \___/  |_.__/   \__,_| |_|  \__|  1.0.90
Parallel build starting
     ╔═════════════════════════╗
     ║ Building kobalt-wrapper ║
     ╚═════════════════════════╝
───── kobalt-wrapper:compile
───── kobalt-wrapper:copyVersionForWrapper
───── kobalt-wrapper:assemble
     ╔════════════════════════════╗
     ║ Building kobalt-plugin-api ║
     ╚════════════════════════════╝
───── kobalt-plugin-api:compile
───── kobalt-plugin-api:copyVersionForWrapper
───── kobalt-plugin-api:assemble
  Created modules\kobalt-plugin-api\kobaltBuild\libs\kobalt-plugin-api-1.0.90.pom
  Created .\kobaltBuild\libs\kobalt-1.0.90.zip


          ╔═════════════════════════════════════════════════════════╗
          ║  Project                     ║ Build status║ Time     ║
          ╠═════════════════════════════════════════════════════════╣
          ║  kobalt-wrapper              ║ SUCCESS     ║ 0.06     ║
          ║  kobalt-plugin-api           ║ SUCCESS     ║ 5.39     ║
          ║  kobalt                      ║ SUCCESS     ║ 13.05    ║
          ╚═════════════════════════════════════════════════════════╝

PARALLEL BUILD SUCCESSFUL (18 SECONDS), sequential build would have taken 25 seconds
```

Incremental tasks in Kobalt

The problem

“If a task is run twice in a row, it should be skipped the second time.”

Constraints:

- Tasks are generic, not necessarily file based.
- Need to apply to the transitive closure of tasks.

The (current) solution:

- Input and output hashes.

Ad hoc polymorphism

Example: a JSON library

Provides JsonObject

```
interface JsonObject {
    fun toJson() : String
}
```

And an API to manipulate these objects

```
fun prettyPrint(jo: JsonObject) = ...
```

Example: a JSON library

You provide implementations through inheritance:

```
class Account : JsonObject {
   fun override toJson() : String { ... }
}
```

Cons:

- Forces inheritance.
- Ties business logic to orthogonal concerns.
- What if you can’t modify `Account`?

Example: a JSON library

You provide implementations as extension functions:

```
fun Account.toJson() : JsonObject = …
```

Cons:

- Less transparent: `prettyPrint(account.toJson())`

Pros:

- Doesn’t pollute your business classes (separation of concerns)

⇒ Poor man ad hoc polymorphism

Example: persistence

Version 1:

```
 fun persist(person: Person) {
     db.save(person.id, person)
}
```

Version 2:

```
interface HasId {
    val id : Id
}

class Person : HasId {
    override val id: Id get() = ...
}

fun persist(o: HasId) { ... }
```

Example: persistence

Version 3:

```
 fun <T> persist(o: T, toId: (T) -> Id) {
    db.save(o, toId(o))
}

// Persist a Person: easy since Person implements HasId
persist(person, { person -> person.id })

// Persist an Account: need to get an id some other way
persist(account, { account -> getAnIdForAccountSomehow(account) }
```

Example: persistence

Again:

```
fun <T> persist(o: T, toId: (T) -> Id) {
    db.save(o, toId(o))
}
```

- Detached from your classes.
- Completely generic. No `Account`, no `Person`, no common base type. Works “for all” types.

Depends less on types, more on functions (but still statically typed!).

Ad hoc polymorphism in a nutshell

- Move away from types (nominal types), put emphasis on functions

- For Kotlin, a step toward type classes

- For more information, see KEEP #87 “Type Classes as extensions in Kotlin” by Raul Raja

Wrapping up

Kobalt:

- http://beust.com/kobalt
- http://github.com/cbeust/kobalt

SAMSUNG

SmartThings™

We’re hiring Kotlin Android developers!

Questions?