# Why Kotlin needs much fewer frameworks

**Kotland 2021 - Anton Keks @antonkeks**

# Kotlin Server-Side

Kotlin was built for in-house development of IntelliJ IDEA

Android developers took over

Server-side is currently neglected, but steadily rising

# Kotlin on Client-Side

Tried, not yet:

- Compiles to ES5 (hard to use in e.g. web components)
- More difficult interop - hard to make “implementation detail”
- Difficult to use any UI framework other than React
- Huge bundle with stdlib

Waiting for easier WebAssembly target, maybe Compose for Web

# Kotlin "killer" features

- Nullability declarations and compile-time checks
- DRY
  - Type inference: no need to specify types twice
  - Constructor & field declarations together (+default values)
- Easy immutable data classes
- Lambda with receiver, enabling builders and what not
- Extension functions
  - Incl. idiomatic built-ins: .apply, .also, .let, etc
- Reified generics
- Type-safe reflection

# Avoiding too many dependencies

- Fewer - lighter, easier upgrades, etc
- Any dependency should be carefully selected
  - Does it add more value than problems/configuration/etc?
  - Will it be easy to replace if needed?
  - Can you implement the same in ~30 lines of code?
  - You control their usage, avoid them to control your code
  - You need to understand how your deps work in detail
- One big framework that calls your code and handles lots of things is bad - impossible to replace later
- Think of deps as “puppet theater”

# My project template

github.com/angryziber/

## kotlin-jooby-svelte-template

Used for 3 real-world customer projects already

I start with it, but modify for each project needs

# JVM - Dependency #1

- Kotlin works and interops really well on the JVM
- Version is not that important, but newer is better
  - E.g. Java 15 finally has better NullPointerExceptions
- JDK has lots of built-in APIs
- My dream is to switch to Kotlin/Native some day, but still few libraries for simple things

Cons:

- Slower startup (not a problem for server-side)
- High memory footprint

# Jooby/Kooby - dependency #2

- We need a HTTP server - right?
- Lightweight enough and simple to use
- Like Spark Java, but more extensible and less static
- You write your own main(), seems easy to replace/extend
- Compile-time MVC routes (easy to unit-test)
- Coroutine support for sane async programming
- Had to do a few PRs, but that’s normal :-)

Disclaimer: maybe I will try using just Undertow in future

# Dependency injection

- DI (IoC) makes deps an implementation detail
- Spring/Guice/Dagger/Koin/etc? - all need lots of config
  - Mostly big and bloated, implement unnecessary features
  - Java frameworks suffer from **Annotation Abuse**
- Kotlin makes constructor injection DRY (unlike Java)
  - All classes are easy to test in isolation
  - Compiler will not allow to use incomplete instances
  - Dependency graph can be created by hand in main()
  - Also really easy to implement by recursively creating class instances using reflection: AutoCreatingServiceRegistry
- **Bonus:** reified generics for **val service: Service = app.require()**

# DB Access - ORM

- Hibernate/JOOQ/Sql2o/etc?
  - Big, verbose, slow to initialize, lots of fighting
- Why not build extensions for java.sql.DataSource?
- Meet JdbcExtensions (usage samples)
  - **db.insert()**, **db.query()**
  - **rs.getLocalDate()**, etc
  - Every project has different needs, I modify them accordingly
- All entities have **val id = UUID.randomUUID()**
- **Bonus:** reflection-based **rs.fromValues()** and **e.toValues()**

# DB Access - migrations

- Liquibase - dependency #3
- Simple to use and well isolated
- Not hard to reimplement as well
  - E.g. initial data as **db.insert(TestData.user)**
  - I have written a simple migration tool for MongoDB

# Transaction handling

- Transaction
- Handling is automatic at boundaries
  - Per request + per coroutine
  - Lazy; commit on success, rollback on failure
  - RequestTransactionHandler
  - TransactionCoroutineContext

# Bonus: Repository integration tests

- I used to prefer H2 DB for integration tests
  - Better to use a real DB in Docker (if not Oracle)
  - Extend DBTest and go (after **docker-compose up db**)
  - Migration at start + auto-rollback after each test
- Others are unit tests
  - Using pre-created entities from TestData
  - **.copy(field = OTHER)** produces modified entities

# Java async HttpClient with coroutines

- Java 11+ HttpClient is well-optimized and built-in
  - Has horrible “modern” Java API with builders and optionals
  - Inconvenient to mock in unit tests, also in Java
- Json - to make Jackson API nicer (dependency #4)
- JsonHttpClient - small mockable and extensible wrapper
  - http.get(), http.post(), etc (usage examples)
  - Async by default (via coroutine await())

# jvm2dts

- My dream of reusing model classes on client-side
  - Easy in full-stack TypeScript project
  - Would be cool to have Kotlin on the UI without the cons
- For now, jvm2dts can generate TypeScript interfaces from Kotlin data classes for type-checking on client-side!

# Cache

- Memcached/Redis/etc? Maybe something simpler first
- Cache - simple in-memory cache with expiration
- usage examples

# E2E tests with Selenide

- Unit tests are fast, but not enough
- It’s good to end-to-end test your app in the same language using the same APIs
  - E.g. you drive app, and can run **db.insert()**, **db.query()**
  - Or even **repository.save(TestData.user.copy(...))**
- E2ETest

# Thanks!

github.com/angryziber/

## kotlin-jooby-svelte-template

Was too quick to follow?

Examine the linked code at your own pace!

At Codeborne we are hiring!