UI RÉACTIVES AVEC KOTLIN FLOW

<MonkeyPatch/>

Experts dans les technologies Web, Mobile et Big Data

Alexandre Delattre
@alexandre_del31

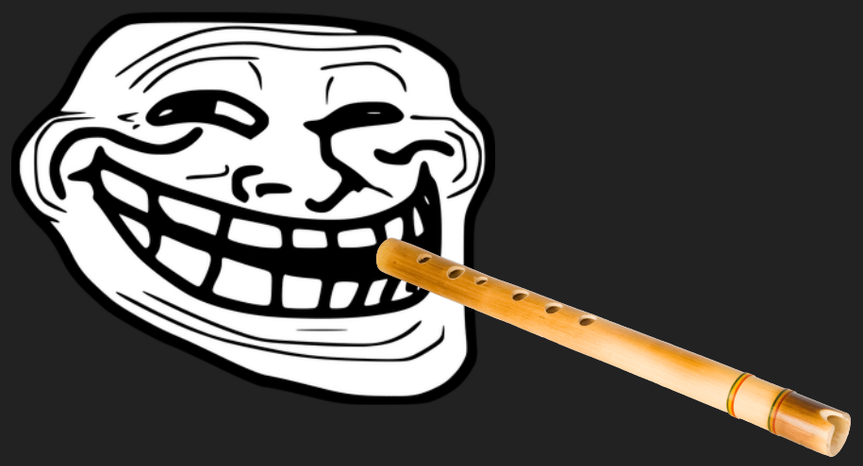
BACKGROUND

<MonkeyPatch/>
Experts dans les technologies Web, Mobile et Big Data

- 2010 : Projet de fin d'étude & premiers développement sous Android 2+
- 2015 : Portage de l'application KissMyShoe sous iOS avec Swift
- 2019 : Réalisation de Konfetti en Kotlin Multiplatform
- En 2020 : Premiers projets KMP « production-grade »

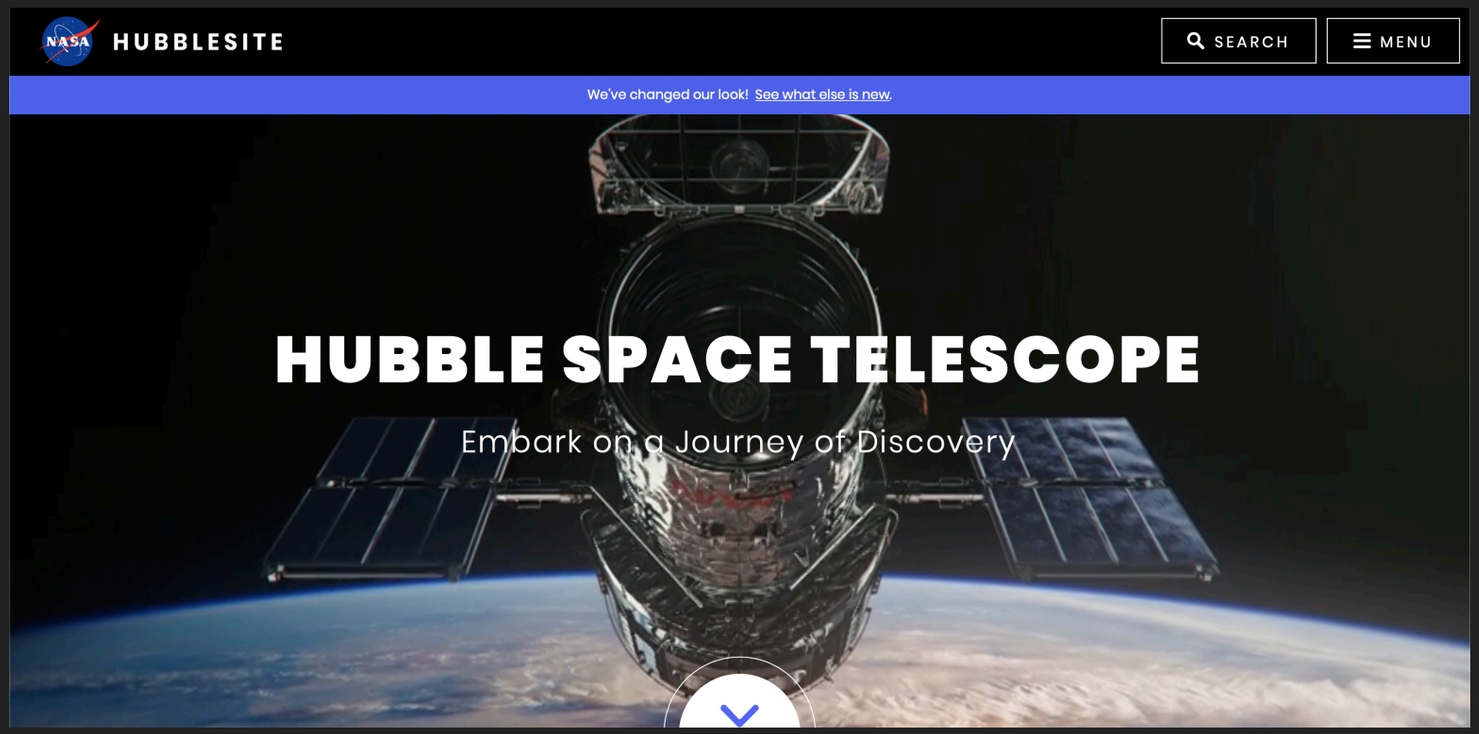
CE QUE NOUS ALLONS VOIR AUJOURD'HUI

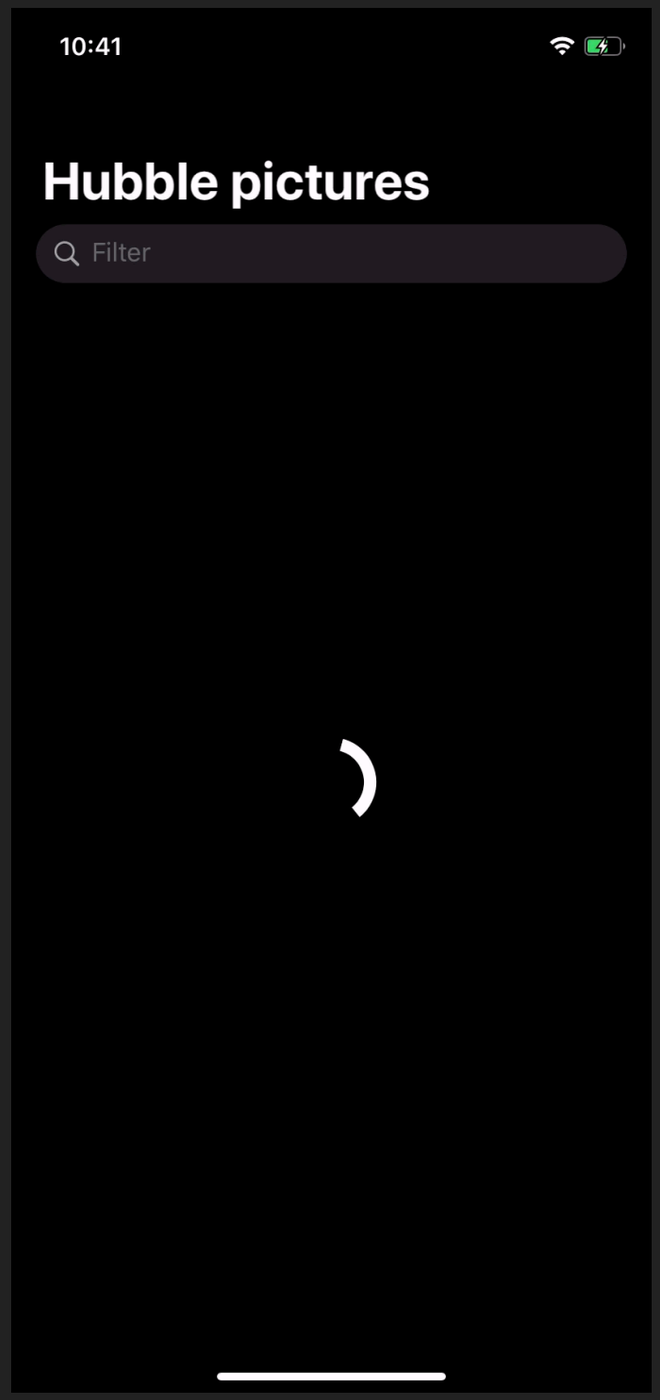
- Pourquoi les frameworks d'UI réactifs sont l'avenir du mobile natif
- Pourquoi Kotlin multiplatform et l'approche réactive Flow vont nous aider à mutualiser du code robuste.
- Pourquoi Flutter sera toujours à la traîne sous iOS



CE QUE NOUS ALLONS VOIR AUJOURD'HUI

- Nous allons construire une app multi-plateforme pour parcourir les photos prises par Hubble 🛰️





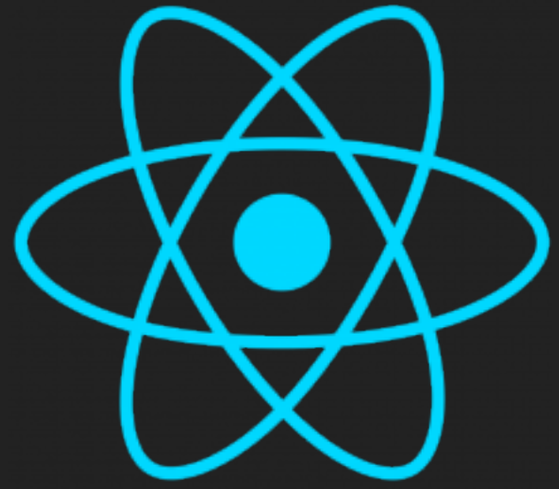
LE PLAN

- Introduction
- Reactive UI
- Kotlin Multiplatform - Flow
- L’app

UN ECOSYSTÈME MOBILE NATIF EN PLEINE MUTATION

… des approches « concurrentes » de plus en plus intéressantes

Flutter

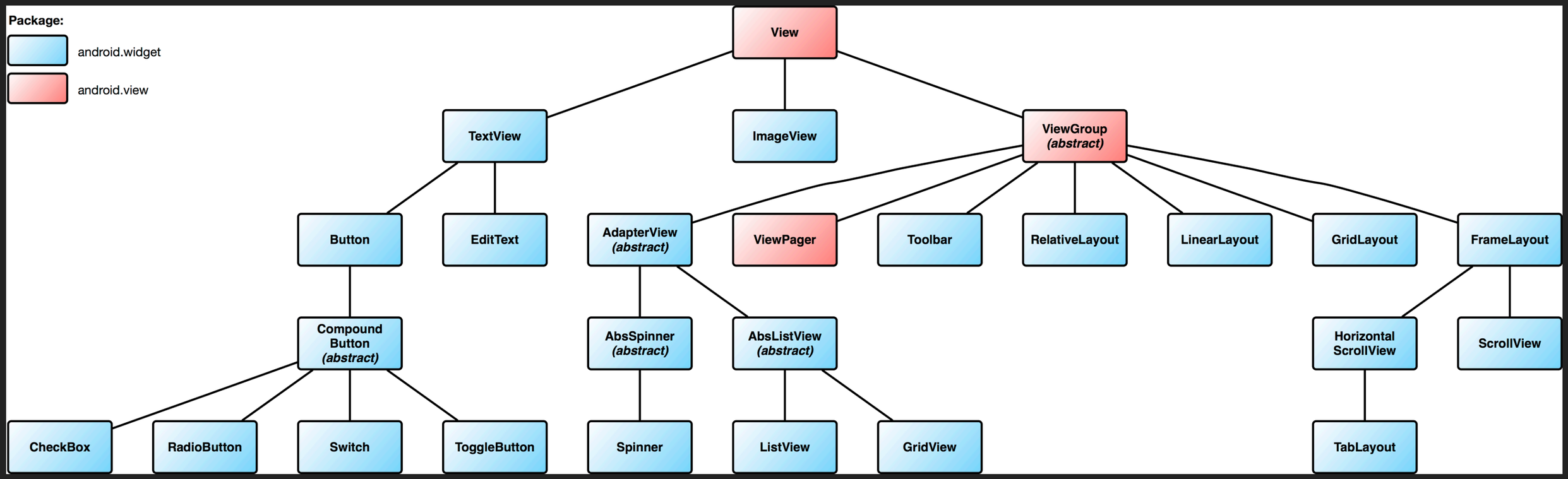


INTRODUCTION

- UIKit (2008)
- Basé sur Cocoa Touch (2001)

- Android 1.0 (2008)
- Développement démarré en 2003
- La classe View fait plus de 30k lignes de code !

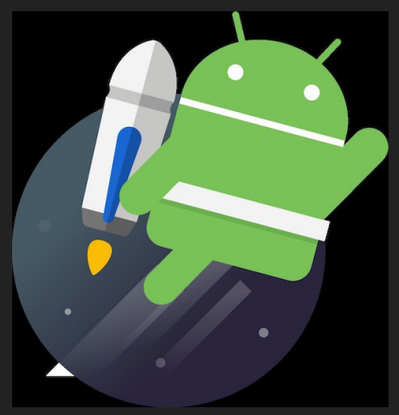
INTRODUCTION




COMPOSITION OVER INHERITANCE
PREFER YOU MUST
memegenerator.net

INTRODUCTION

SwiftUI
(iOS 13)



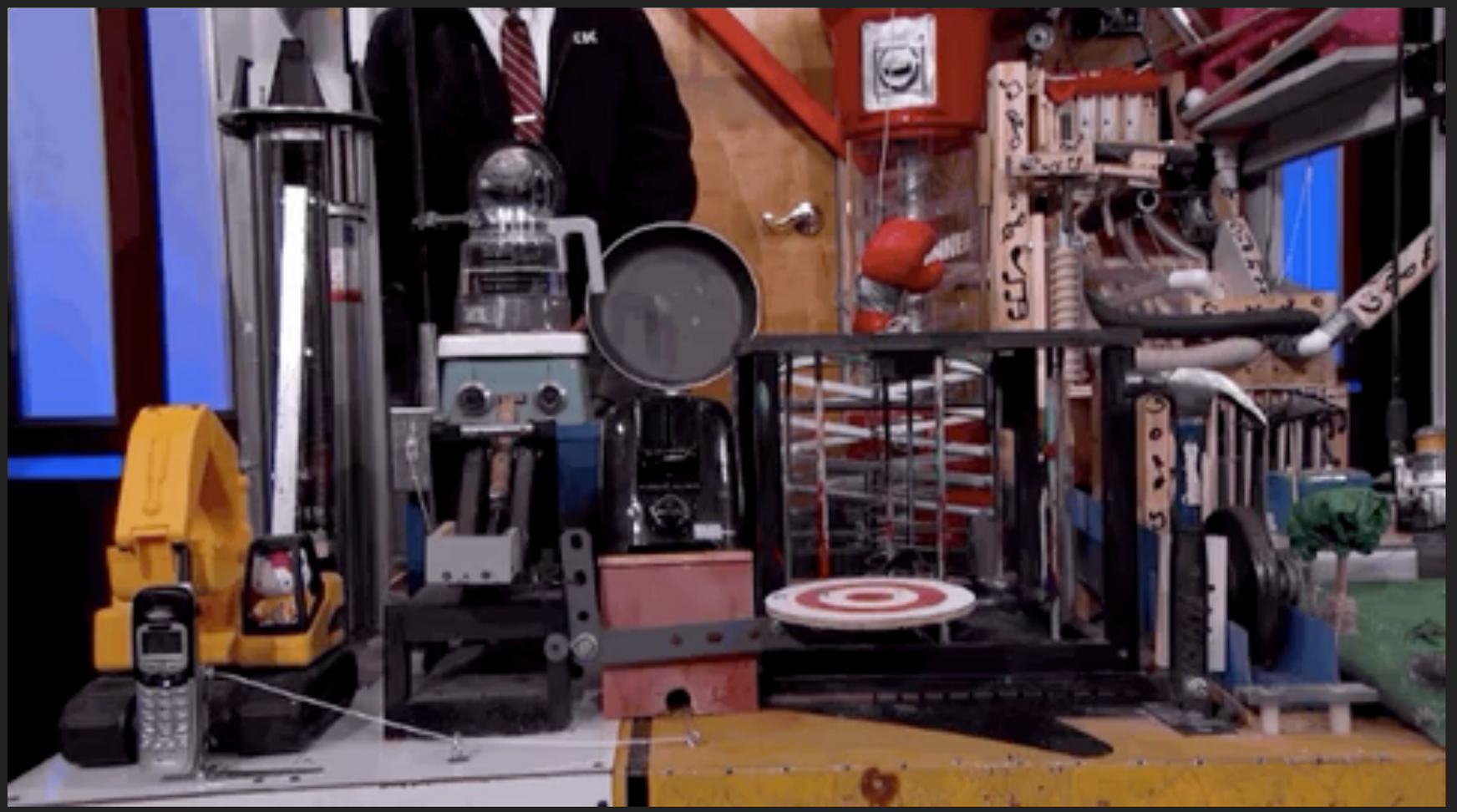
Jetpack Compose
(alpha)

REACTIVE EXTENSION FOR THE WIN 💪

- RxJava et RxSwift sont utilisés dans deux nombreux projets mobiles
- Coté backend, l'approche réactive est aussi en plein essor (i.e. WebFlux)
- Apple a adopté officiellement cette approche avec Combine.
- Kotlin adopte cette approche avec Flow

REACTIVE EXTENSION FOR THE WIN 💪

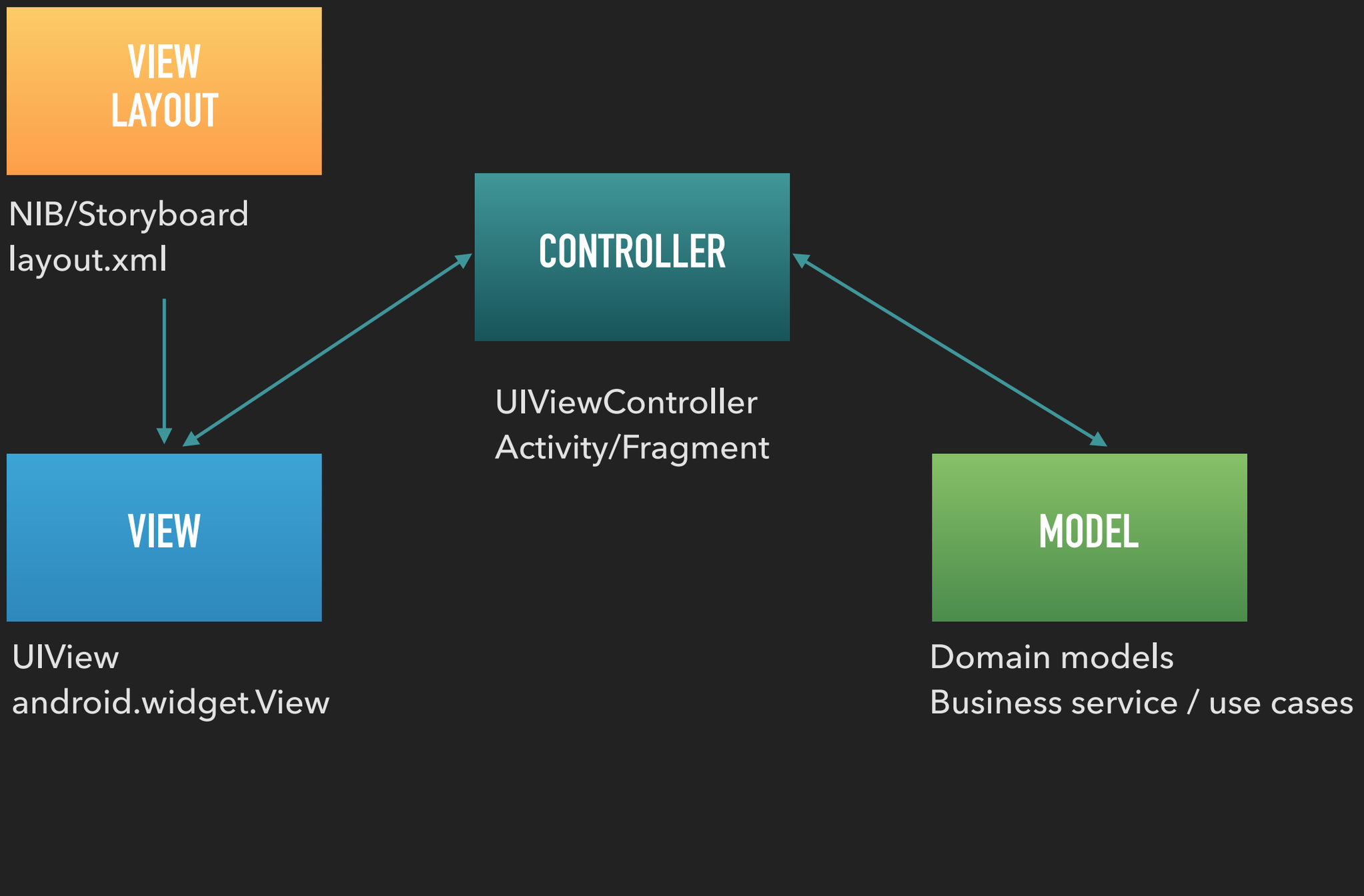
- Attention à la complexité !!!



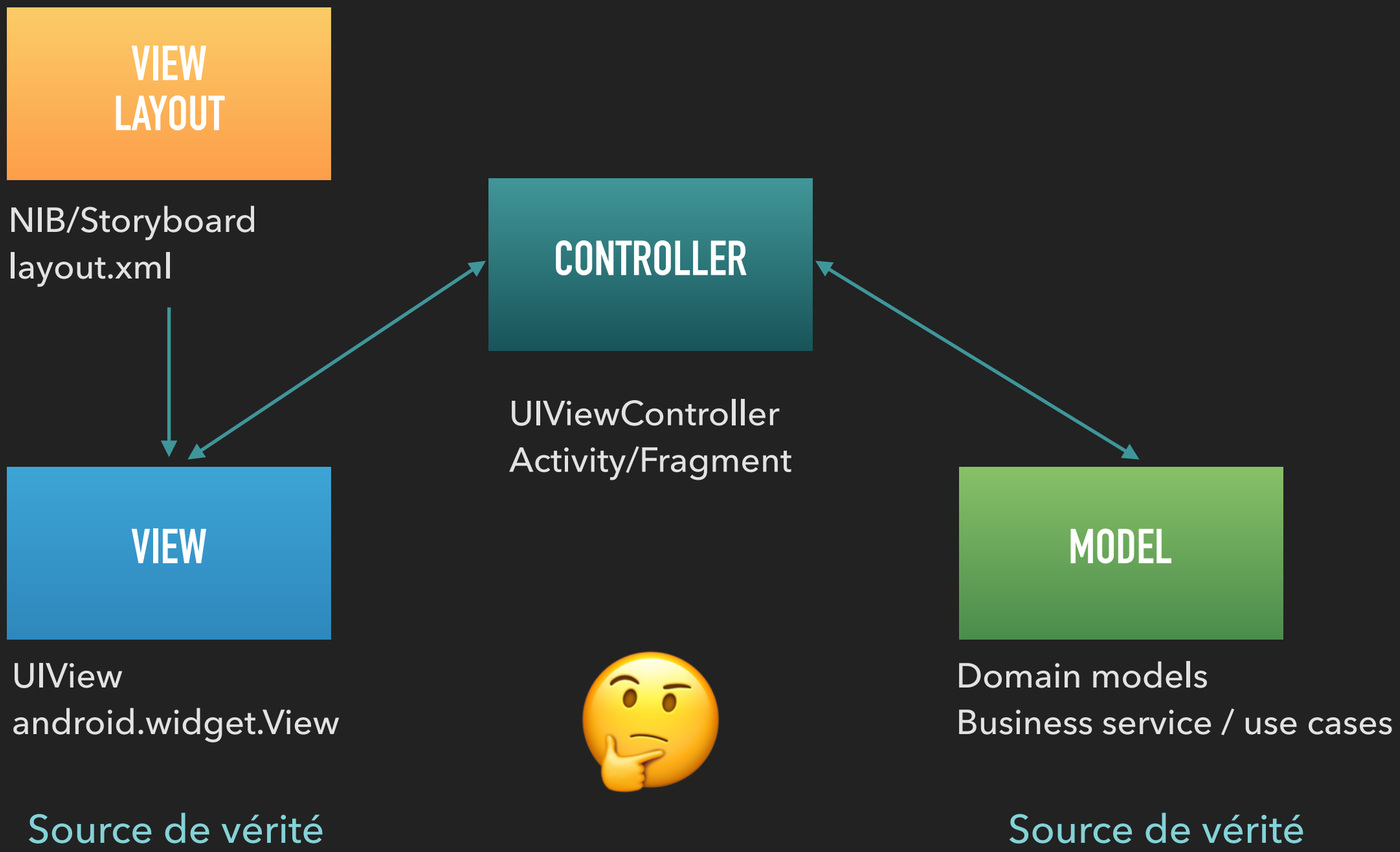
PARTIE 1

REACTIVE UI

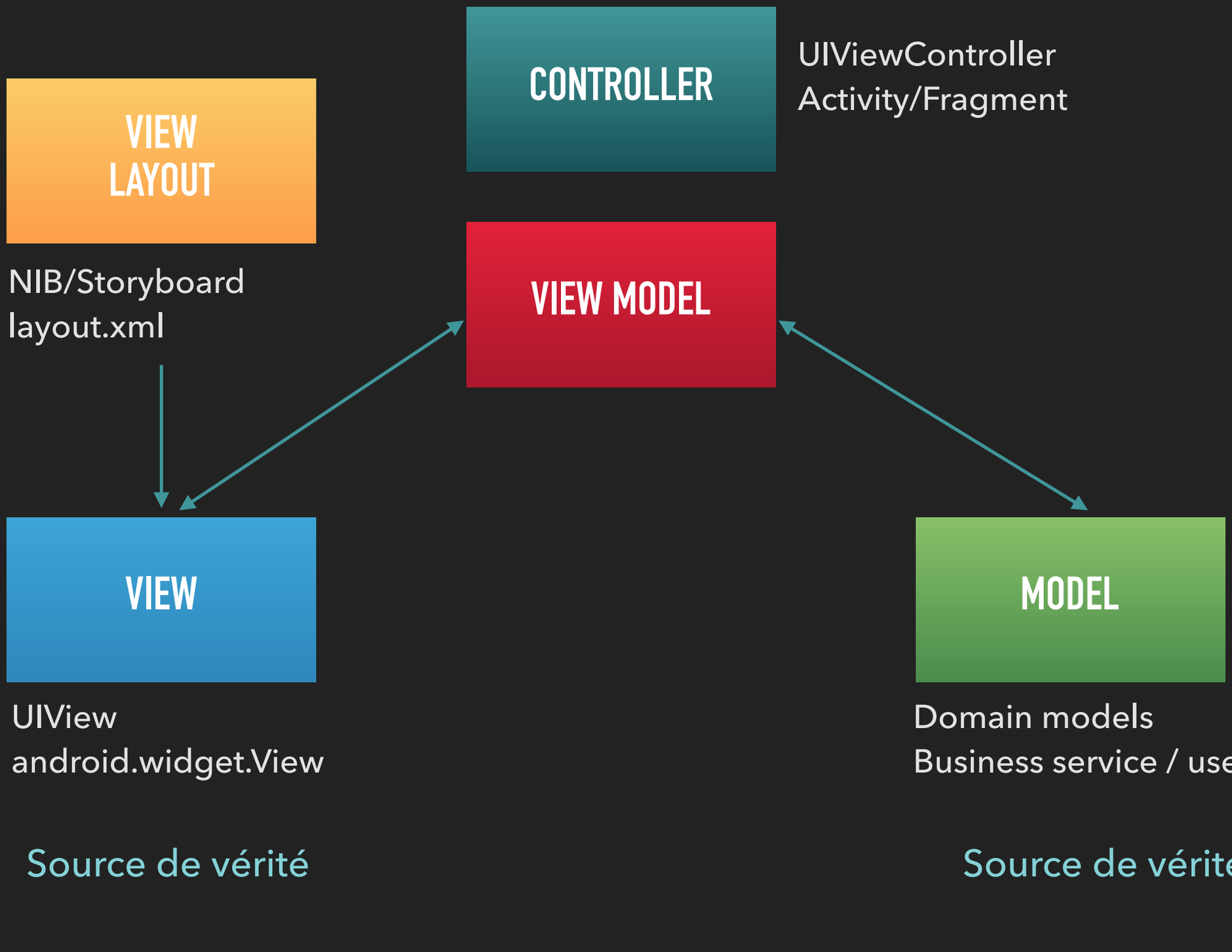
PLAIN-OLD MVC



PLAIN-OLD MVC



PLAIN-OLD MVVM ?

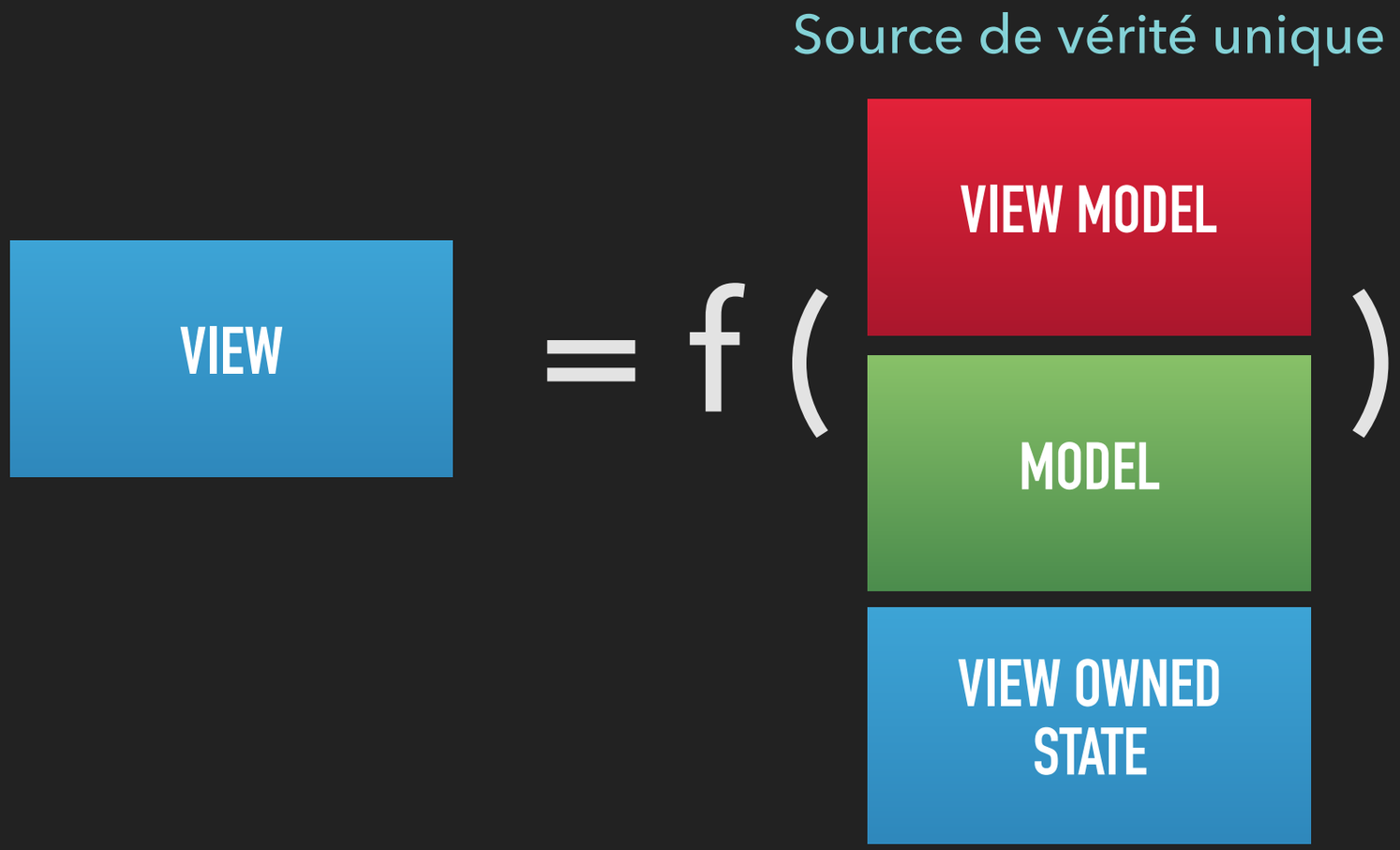


DECLARATIVE/REACTIVE APPROACH

Source de vérité unique

Domain models
Business service / use cases

DECLARATIVE/REACTIVE APPROACH



UN EXEMPLE EN PSEUDO-CODE

```
class SomeActivity(
    private val viewModel: ViewModel
) {
    lateinit var view: View

    fun create() {
        view = loadViewFromXml()
        view.loader.visible = true
        view.list.visible = false

        viewModel.observe { newData ->
            update(newData)
        }
    }

    fun update(data: Data) {
        view.loader.visible = false
        view.list.visible = true
        view.list.adapter = Adapter(data)
    }
}
```

UN EXEMPLE EN PSEUDO-CODE

```
@Compose
fun SomeView(viewModel: ViewModel) {
    // SomeView est recomposé à chaque évolution de la LiveData
    val data = +observe(viewModel.liveData)
    data?.let {
        DataList(data)
    } ?: ProgressIndicator()
}

@Compose fun ProgressIndicator() {}

@Compose fun DataList(data: Data) {}
```

UN EXEMPLE EN PSEUDO-CODE

```
struct SomeView: View {
    @ObservedObject var viewModel: ViewModel

    var body: some View {
        ZStack {
            if viewModel.data == nil {
                ProgressIndicator()
            } else {
                DataList(data: viewModel.data!)
            }
        }
    }
}

struct ProgressIndicator: View { }

struct DataList: View { }
```

DESCRIBING THE STATE OF THE UI RIGHT NOW, FOR ANY VALUE OF NOW

Adam Perry

COMMENT ÇA MARCHE ?

- On reconstruit l'arbre de vue (ou un sous ensemble) à chaque changement des sources de vérités
- Est-ce couteux 💸💸💸?
- Non, car l'arbre de vue est une description légère de la vue
- Le runtime (SwiftUI/Compose) fait un différentiel de l'arbre de vue, afin de dessiner et animer la vue graphique réelle

BÉNÉFICES

- 1 seul source de vérité pour chaque data
  - Moins de bugs
  - Code plus simple à écrire/lire
- 1 seul source pour décrire le layout UI
- Composants plus facilement réutilisables/refactorables
- Approche « trait » plus flexible
- Previews ❤️

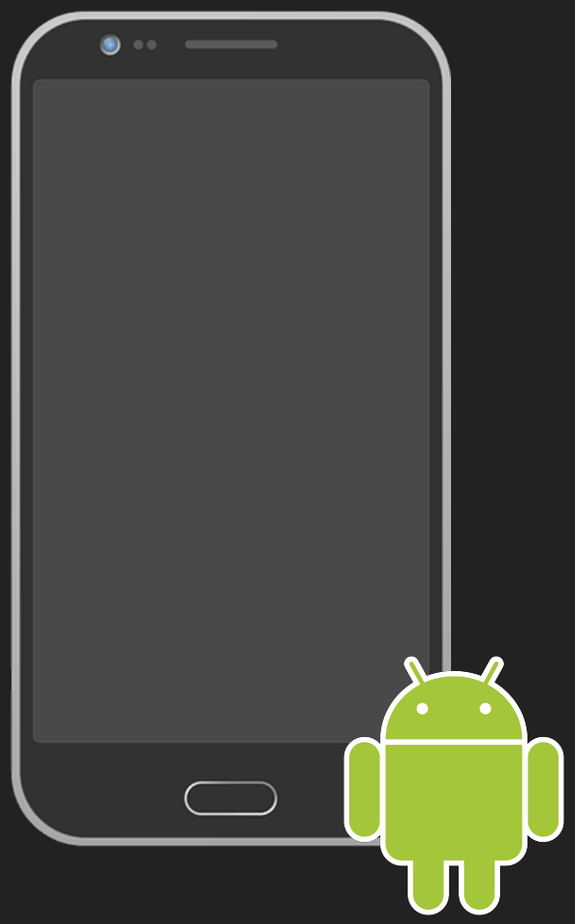
PARTIE 2

KOTLIN MULTIPLATFORM

KOTLIN, UN LANGAGE PRAGMATIQUE

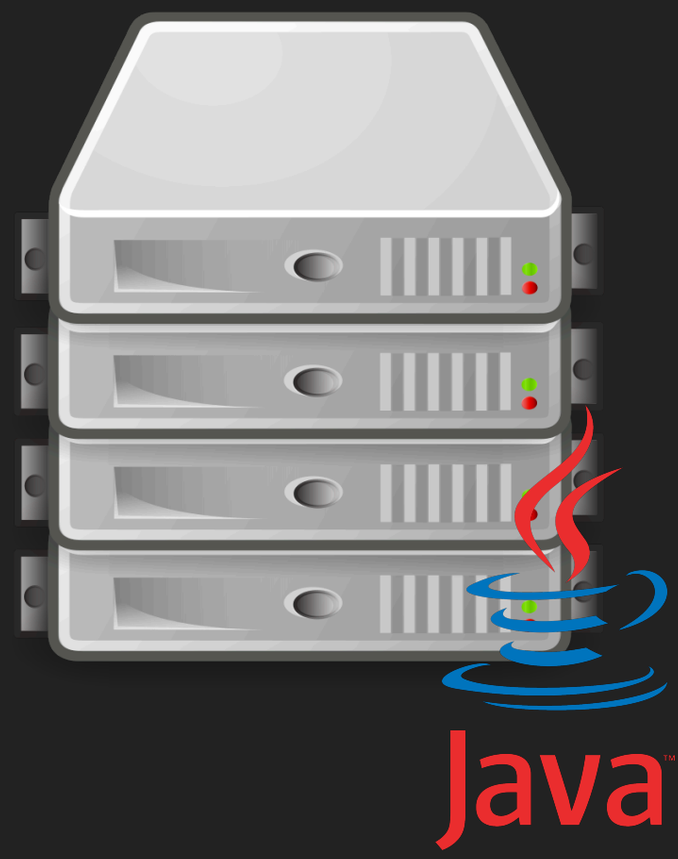
- Lisibilité
- Réutilisabilité
- Interopérabilité
- Sûreté
- Tooling

KOTLIN UN LANGAGE MULTI-PLATEFORME



Kotlin/JVM

Kotlin/Native



Kotlin/JVM

STRUCTURE D'UN PROJET MULTIPLATEFORME

- Un projet Gradle contenant plusieurs source sets.

```
src/
   commonMain/ : common code
   commonTest/ : common test code
   iosMain/    : iOS specific code
   iosTest/    : iOS specific test code
   androidMain/: Android specific code
   androidTest/: Android specific test code
```

Code commun

Code spécifique

CODE MULTIPLATEFORME

commonMain/

```
expect fun getCurrentTimeMillis(): Long
```

androidMain/

```
actual fun getCurrentTimeMillis() = System.currentTimeMillis()
```

iosMain/

```
actual fun getCurrentTimeMillis(): Long =
    NSDate.date().timeIntervalSince1970.toLong()
```

UN PETIT MOT SUR LES COROUTINES

- Les coroutines sont des « threads » légers
- Il est possible de scheduler un grand nombre de coroutines sur un faible nombre de thread
- La notion de « concurrence structurée » permet de hiérarchiser les coroutines entre elle et de gérer l'annulation de manière prévisible
- Les coroutines peuvent être utilisées directement, mais elles sont aussi une excellente fondation pour des abstractions de plus haut-niveau : Reactive / Acteurs / FP

UN PETIT MOT SUR LES COROUTINES

MonkeyTech Days

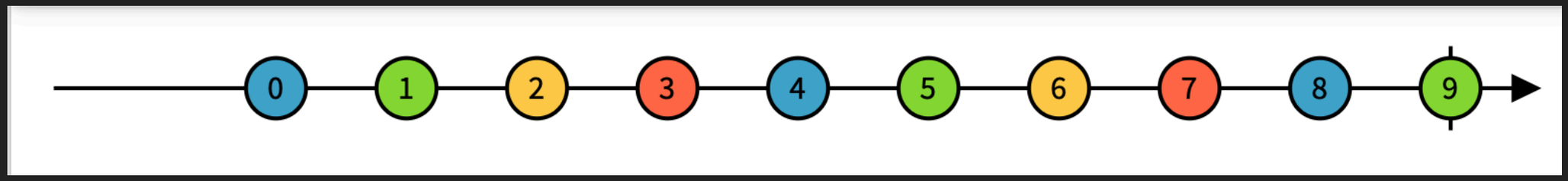
Quasar vs Kotlin Coroutines

13 Mars 2020

https://www.meetup.com/fr-FR/MonkeyTechDays/

FLOW

- A l'instar des ReactiveStreams, un **`Flow<T>`** définit une sequence asynchrone de valeurs de type T



FLOW

- **`Flow<T>`** a été introduit en « experimental » dans la librairie **kotlinx.coroutines**
- Flow<T> supporte la back-pressure. L'implémentation est beaucoup plus simple que pour les Reactive Streams.
- **`Flow<T>`** n'implémente pas directement l'interface Publisher des Reactive Streams

FLOW

- Pour consommer le **`Flow`**, une coroutine le collecte à l'aide d'un opérateur terminal
- Pour arrêter la production, il suffit d'annuler la coroutine consommatrice.

```
fun main() = runBlocking {
    // On lance une coroutine de collecte
    // en fond
    val collectJob = launch {
        sampleFlow.collect {
            println("Received $it")
        }
    }

    // Notre coroutine attend 2s avant
    // d'annuler la collecte
    delay(2000)
    collectJob.cancel()
}
```

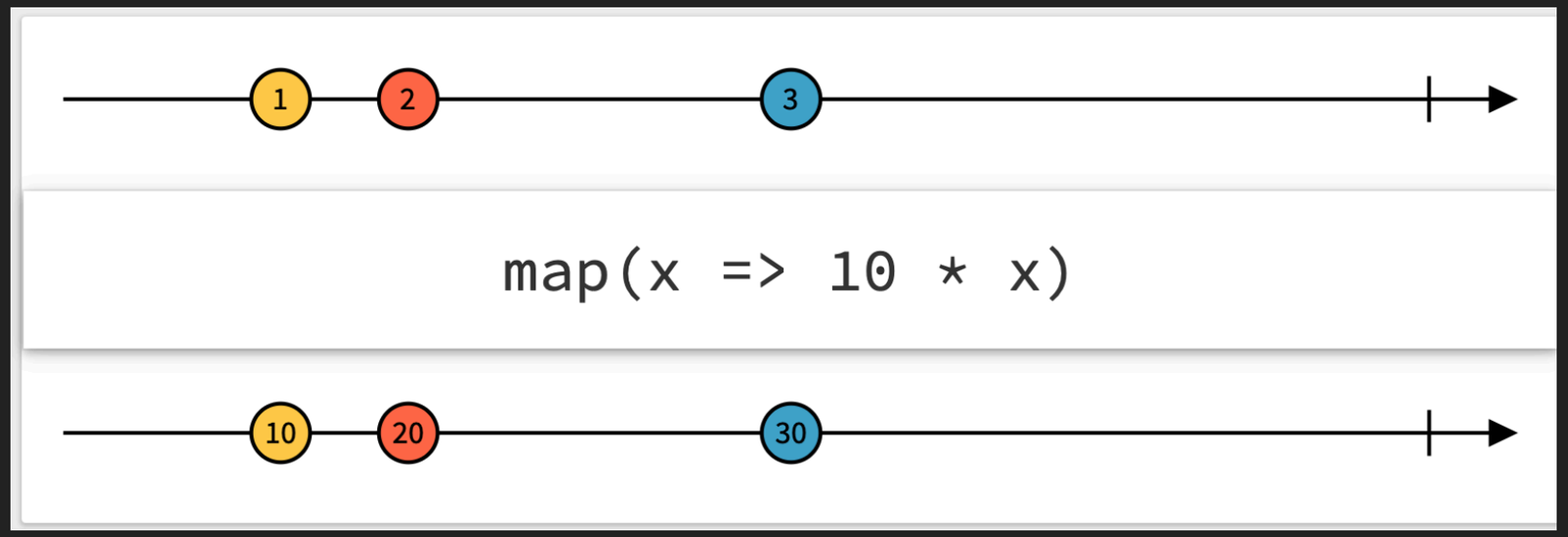
COLD FLOW / HOT CHANNELS

- Un **`Flow<T>`** est un flux froid, c.a.d
  - Il est possible de le collecter plusieurs fois
  - Chaque consommation, va donner lieu à une emission unique
- Si le **`Flow`** « contient » un effet de bord :
  - L'effet de bord sera reproduit à chaque consommation
  - Des valeurs uniques seront envoyés à chaque consommateur
- Si l'on a besoin d'un Flux chaud (par ex. pour partager les valeurs) il faut transformer le **`Flow`** en **`Channel`**.

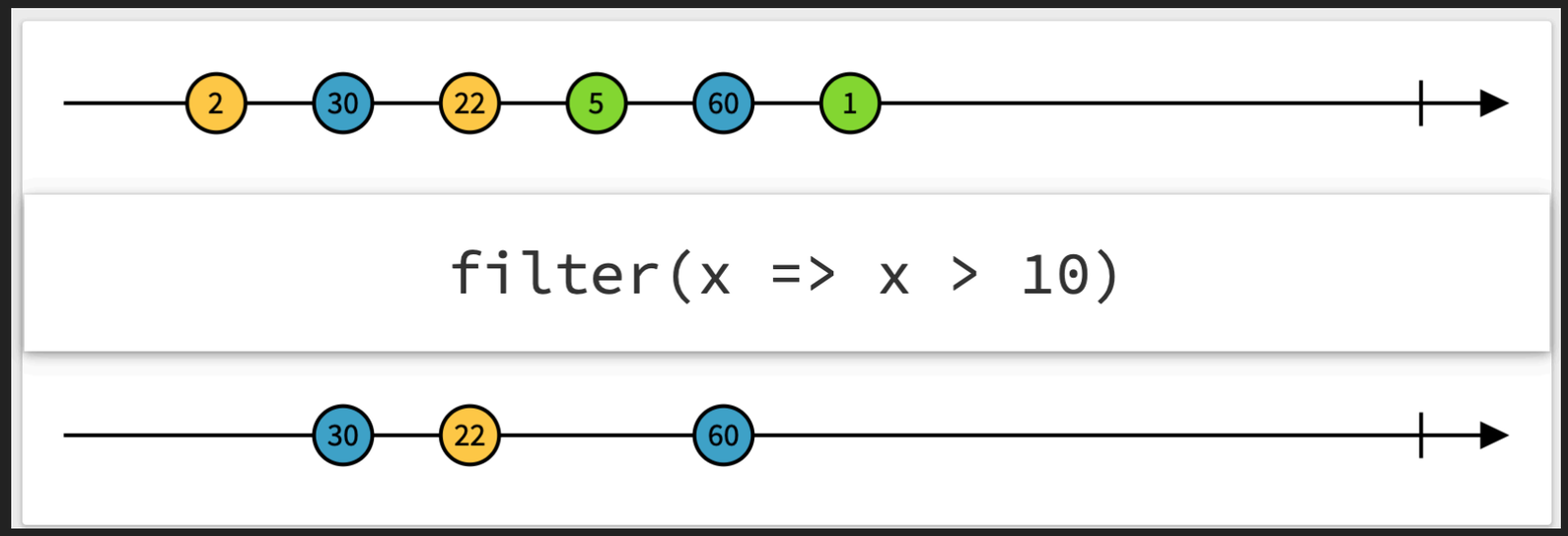
A MONAD IS JUST A MONOID IN THE CATEGORY OF ENDOFUNCTOR

OPÉRATEURS

- Map

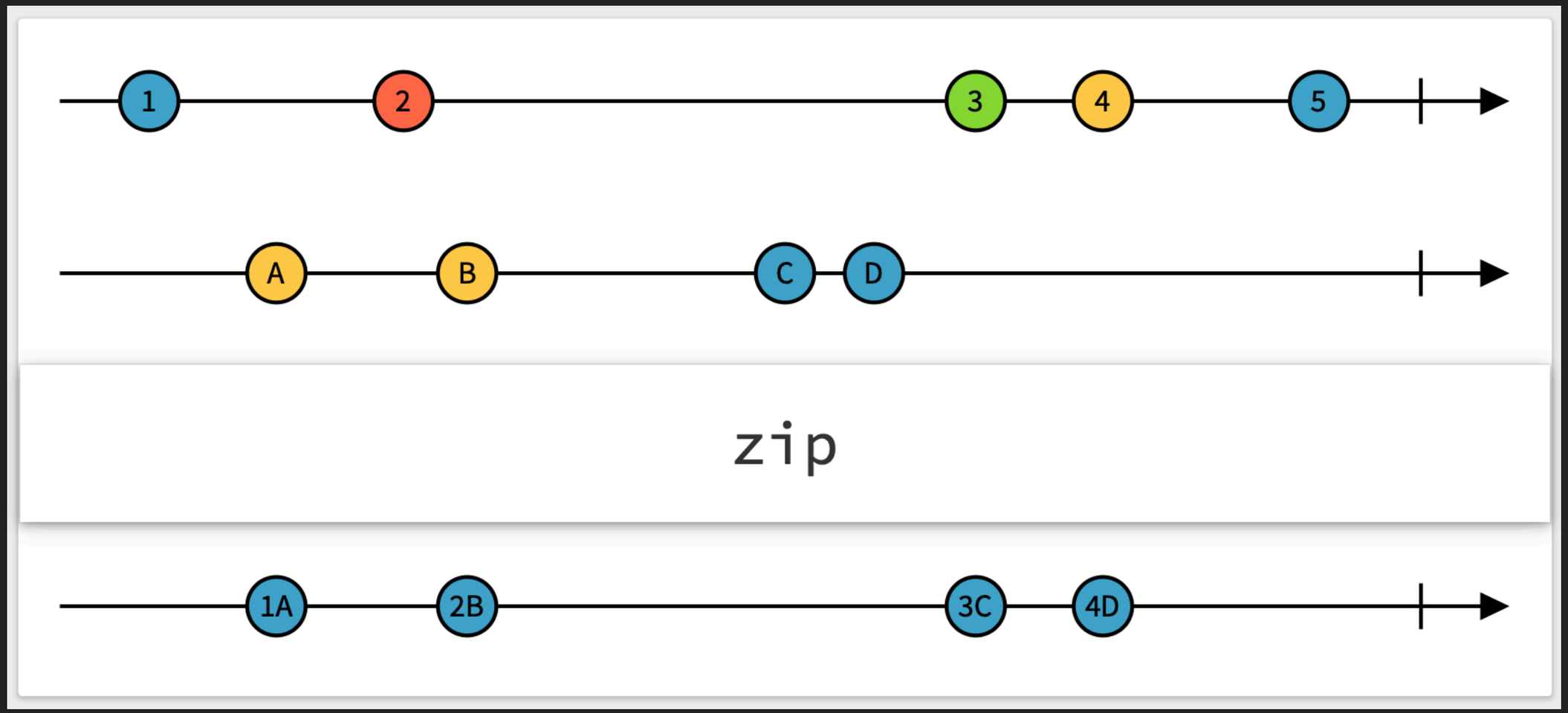


- Filter



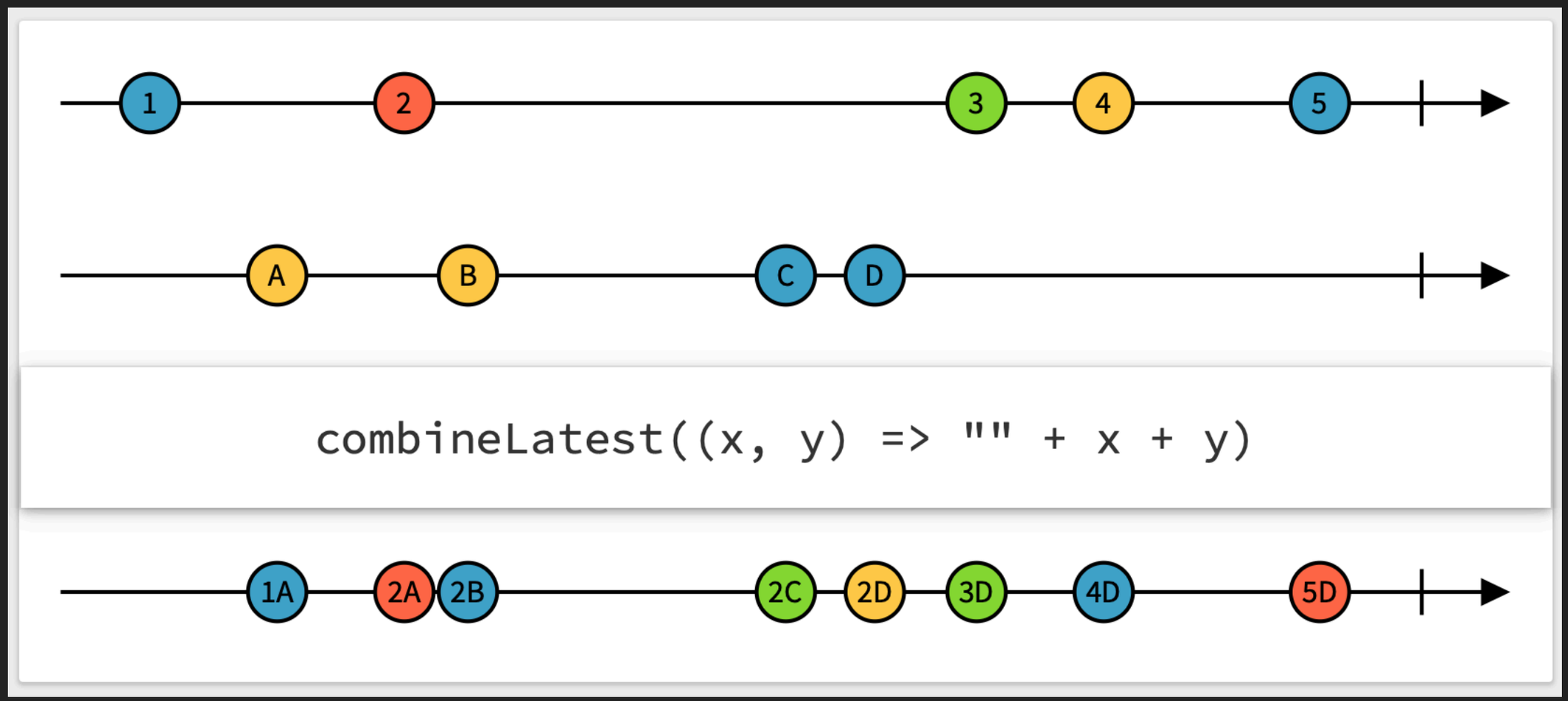
OPÉRATEURS

- Zip

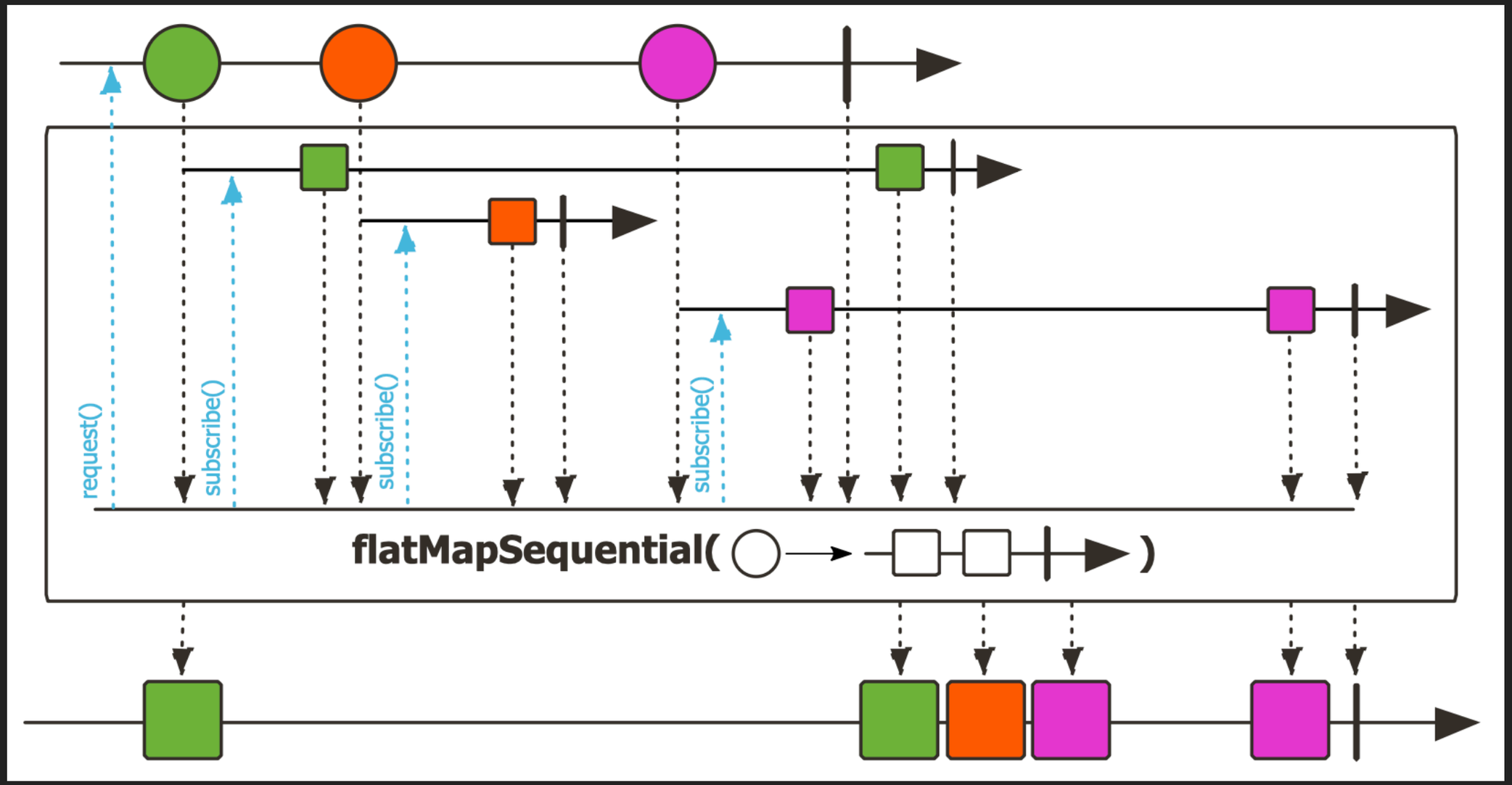


OPÉRATEURS

- Combine



OPÉRATEURS CUSTOM



OPÉRATEURS CUSTOM

```kotlin
@ExperimentalCoroutinesApi
fun <T, U> Flow<T>.flatMapSequential(
    concurrency: Int = DEFAULT_CONCURRENCY,
    prefetch: Int = 8,
    block: suspend (T) -> Flow<U>
): Flow<U> = channelFlow {
    val channel = Channel<Channel<U>>(concurrency)

    launch {
        collect {
            val subFlow = block(it)
            val subChannel = Channel<U>(prefetch)
            channel.send(subChannel)
            launch {
                subFlow.collect {
                    subChannel.send(it)
                }
                subChannel.close()
            }
        }

        channel.close()
    }

    channel.consumeEach { subChannel ->
        subChannel.consumeEach {
            send(it)
        }
    }
}
```

~ 30 lignes de code

OPÉRATEURS CUSTOM

```
/**
 * Maps each upstream value into a Publisher and concatenates them into one
 * sequence of items.
 *
 * @param <T> the source value type
 * @param <R> the output value type
 * @see <a href="https://github.com/reactor/reactive-streams-commons">Reactive-Streams-Commons</a>
 */
final class FluxMergeSequential<T, R> extends InternalFluxOperator<T, R> {

	final ErrorMode errorMode;

	final Function<? super T, ? extends Publisher<? extends R>> mapper;

	final int maxConcurrency;

	final int prefetch;

	final Supplier<Queue<MergeSequentialInner<R>>> queueSupplier;

	FluxMergeSequential(Flux<? extends T> source,
			Function<? super T, ? extends Publisher<? extends R>> mapper,
			int maxConcurrency, int prefetch, ErrorMode errorMode) {
		this(source, mapper, maxConcurrency, prefetch, errorMode,
				Queues.get(Math.max(prefetch, maxConcurrency)));
	}

	//for testing purpose
	FluxMergeSequential(Flux<? extends T> source,
			Function<? super T, ? extends Publisher<? extends R>> mapper,
			int maxConcurrency, int prefetch, ErrorMode errorMode,
			Supplier<Queue<MergeSequentialInner<R>>> queueSupplier) {
		super(source);
		if (prefetch <= 0) {
			throw new IllegalArgumentException("prefetch > 0 required but it was " + prefetch);
		}
		if (maxConcurrency <= 0) {
			throw new IllegalArgumentException("maxConcurrency > 0 required but it was " + maxConcurrency);
		}
		this.mapper = Objects.requireNonNull(mapper, "mapper");
		this.maxConcurrency = maxConcurrency;
		this.prefetch = prefetch;
		this.errorMode = errorMode;
		this.queueSupplier = queueSupplier;
	}

	@Override
	public CoreSubscriber<? super T> subscribeOrReturn(CoreSubscriber<? super R> actual) {
		//for now mergeSequential doesn't support onErrorContinue, so the scalar version shouldn't either
		if (FluxFlatMap.trySubscribeScalarMap(source, actual, mapper, false, false)) {
			return null;
		}

		return new MergeSequentialMain<T, R>(actual,
				mapper,
				maxConcurrency,
				prefetch,
				errorMode,
				queueSupplier);
	}

	static final class MergeSequentialMain<T, R> implements InnerOperator<T, R> {

		/** the mapper giving the inner publisher for each source value */
		final Function<? super T, ? extends Publisher<? extends R>> mapper;

		/** how many eagerly subscribed inner stream at a time, at most */
		final int maxConcurrency;

		/** request size for inner subscribers (size of the inner queues) */
		final int prefetch;

		final Queue<MergeSequentialInner<R>> subscribers;
```

~ 600 lignes de code

😱 😱 😱

PARTIE 3

L'APP

UN MOT SUR L'ARCHITECTURE

UI

SwiftUI
+
Compose

VIEWMODEL

USE CASE

ENTITY

REPOSITORY

REST CLIENT

Kotlin Multiplatform
Code partagé

UN MOT SUR L'ARCHITECTURE

UI	VIEWMODEL	USE CASE ENTITY	REPOSITORY	REST CLIENT
SwiftUI + Compose	KLiveData	Flow	Ktor HTTP client + coroutines	

LIVE CODE

CONCLUSION

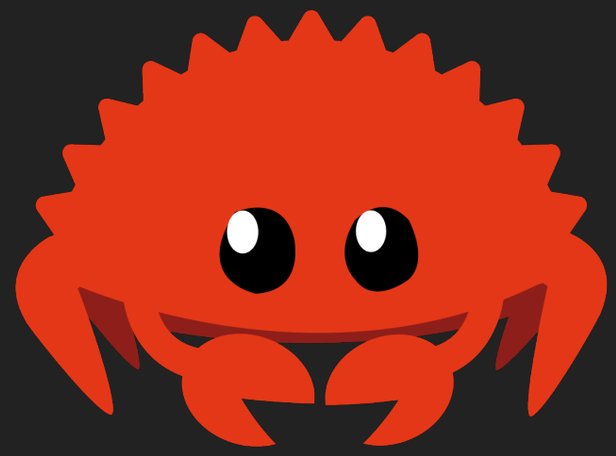
- The good
  - Très prometteur et utile à l'avenir
  - ❤️ l'approche fonctionnelle + déclarative + pragmatique

CONCLUSION

- The bad
  - Compose en early alpha
  - Boilerplate. Nécessite de connaitre/ jongler avec beaucoup de technos
- KLiveData + Flow => DataFlow

CONCLUSION

- The ugly
  - Le modèle mémoire/multithreading de Kotlin native
  - La gestion des dépendances binaires de Kotlin native



Kotlin Native ?

I ❤️  otlin Native

QUESTIONS ?