

Lecture #7

Reactive Programming

Mobile Applications 2019-2020

Why Reactive?

- Unless you can model your entire system synchronously, a single asynchronous source breaks imperative programming.



Why Reactive?

```
val um: UserManager = UserManagerImpl()  
Logd(um.getUser())
```

```
um.setName("John Doe")  
Logd(um.getUser()): User  
}  
fun setName(name: String)  
fun setAge(age: Int)  
}
```



Why Reactive?

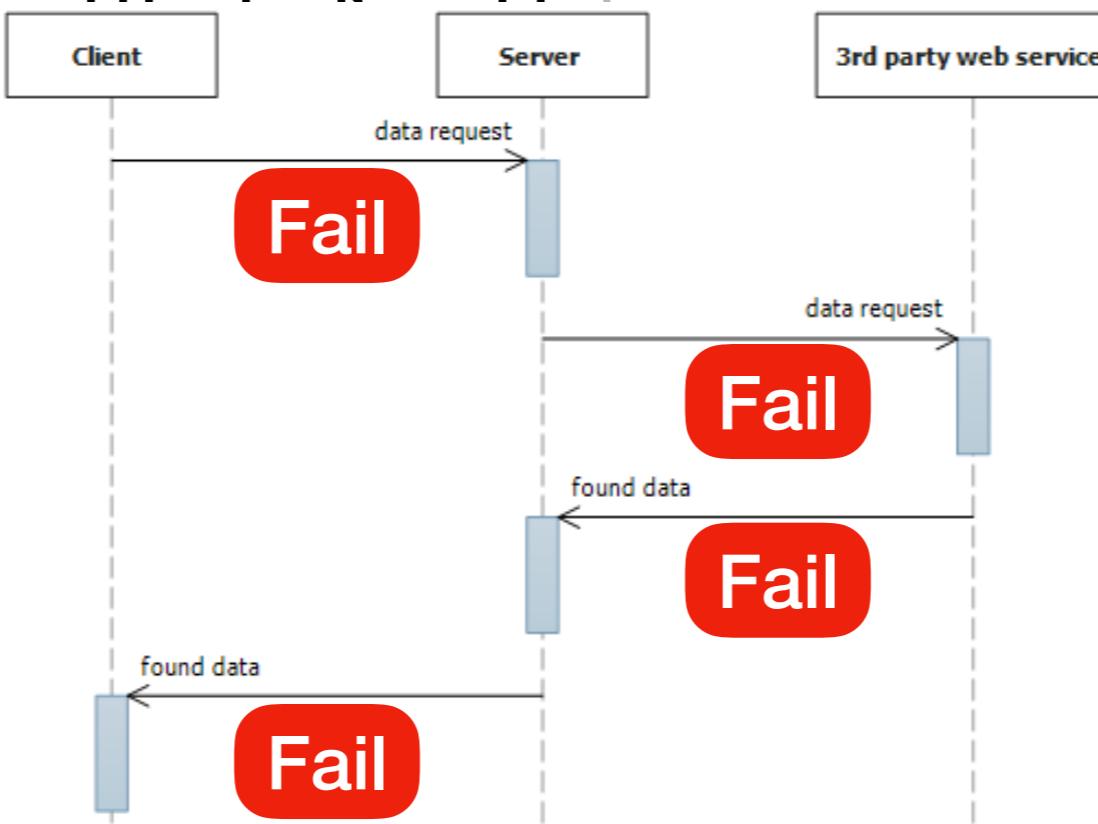
```
interface UserManager {  
    fun getUser(): User  
    fun setName(name: String)  
    fun setAge(age: Int)  
}
```

```
um.setName("John Doe")  
logd(um.getUser())
```



Why Reactive?

```
val um: UserManagerV2 = UserManagerV2Impl()  
interface UserManagerV2 {  
    fun getUser(): User  
    fun setName(name: String, callback: Runnable)  
    fun getName(): String  
}  
}
```



Why Reactive?

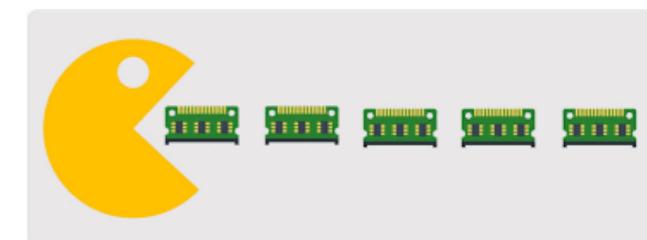
```
um.setName("John Doe", object : UserManagerV3.Listener {  
    override fun success(user: User) {  
        um.setAge(42, object : UserManagerV3.Listener {  
            override fun success(user: User) {  
                logd(user)  
            }  
        }  
    }  
    override fun failed(error: UserException) {  
        loge("Unable to update the user details", error)  
    }  
}  
val userManagerV3: UserManagerV3Impl =  
    UserManagerV3Impl(  
        userManagerV3Listener = object :  
            Listener {  
            override fun success(user: User) {  
                logd("User updated successfully", user)  
            }  
            override fun failed(error: UserException) {  
                loge("Unable to update the user details", error)  
            }  
        }  
    )
```





Why Reactive?

```
class UserActivity : AppCompatActivity() {  
    val um: UserManager = UserManagerImpl()  
    override fun onCreate(savedInstanceState: Bundle?) {  
        um.setName("John Doe", object : UserManager.Listener {  
            override fun success(user: User) {  
                um.setAge(42, user, object : UserManager.Listener {  
                    override fun success(user: User) {  
                        um.setAge(42, user, object : UserManager.Listener {  
                            override fun success(user: User) {  
                                um.setName("John Doe", user, object : UserManager.Listener {  
                                    override fun success(user: User) {  
                                        if (!isDestroyed) {  
                                            updateUserDetails()  
                                        }  
                                    }  
                                }  
                            }  
                        }  
                    }  
                }  
            }  
        }  
    }  
    override fun failed(error: UserException) {  
        loge("Unable to update the user details", error)  
    }  
}  
override fun failed(error: UserException) {  
    loge("Unable to update the user details", error)  
}  
override fun failed(error: UserException) {  
    loge("Unable to update the user details", error)  
}
```

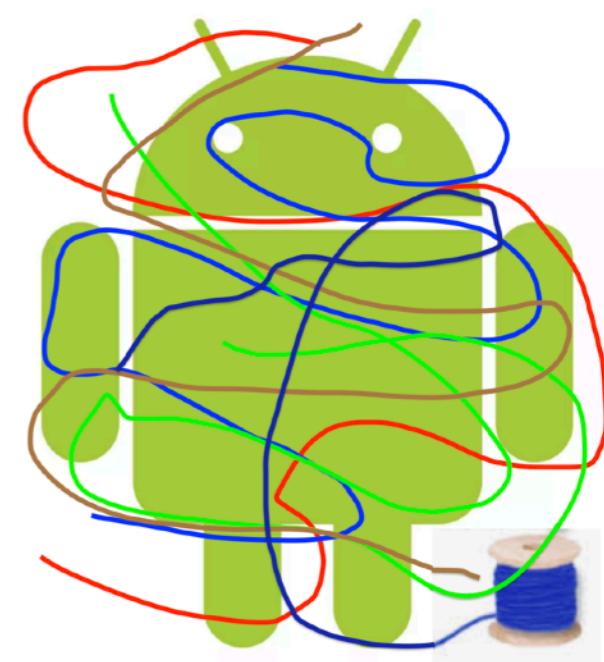


"A small leak will sink a great ship."
Benjamin Franklin

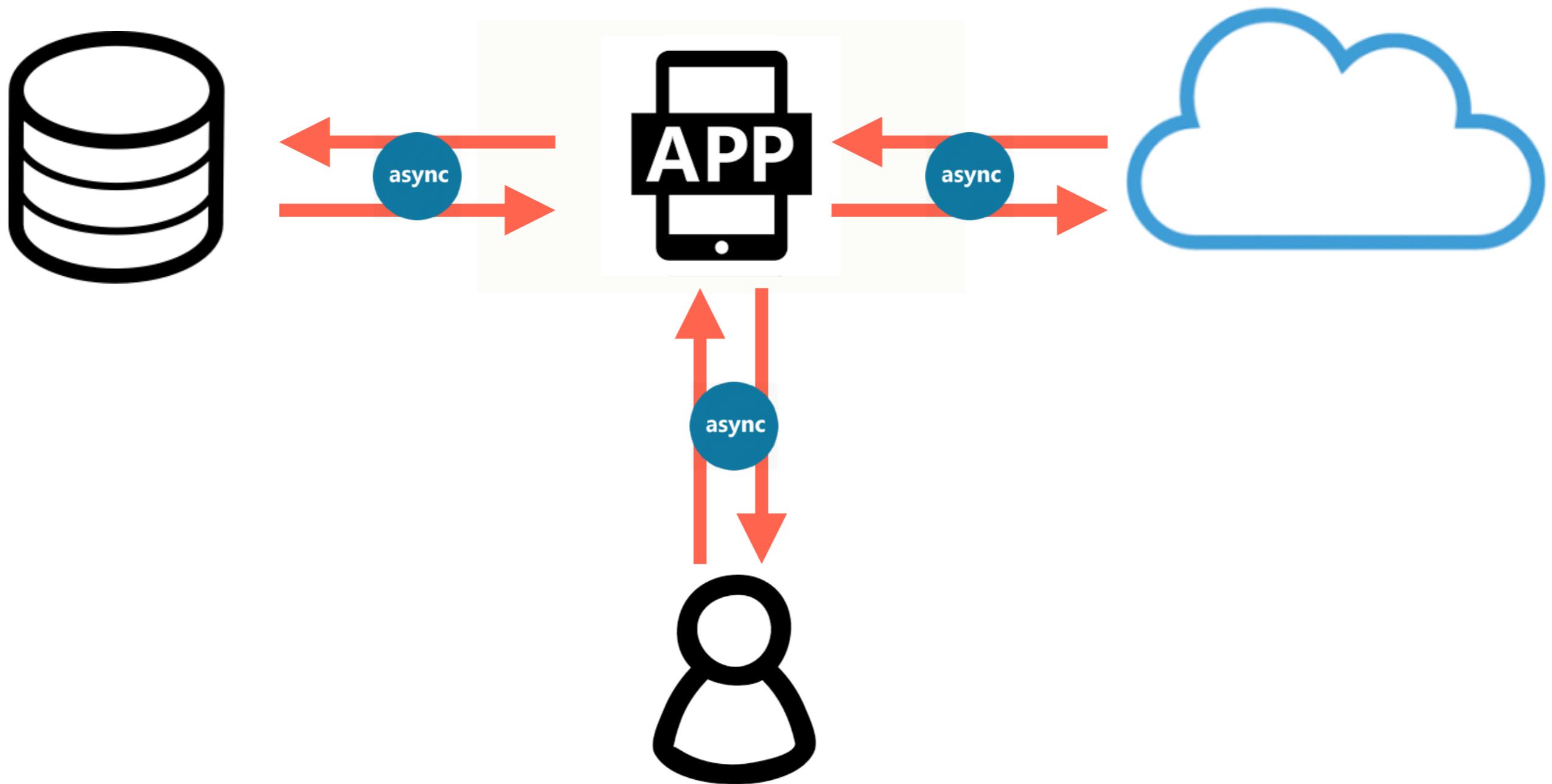


Why Reactive?

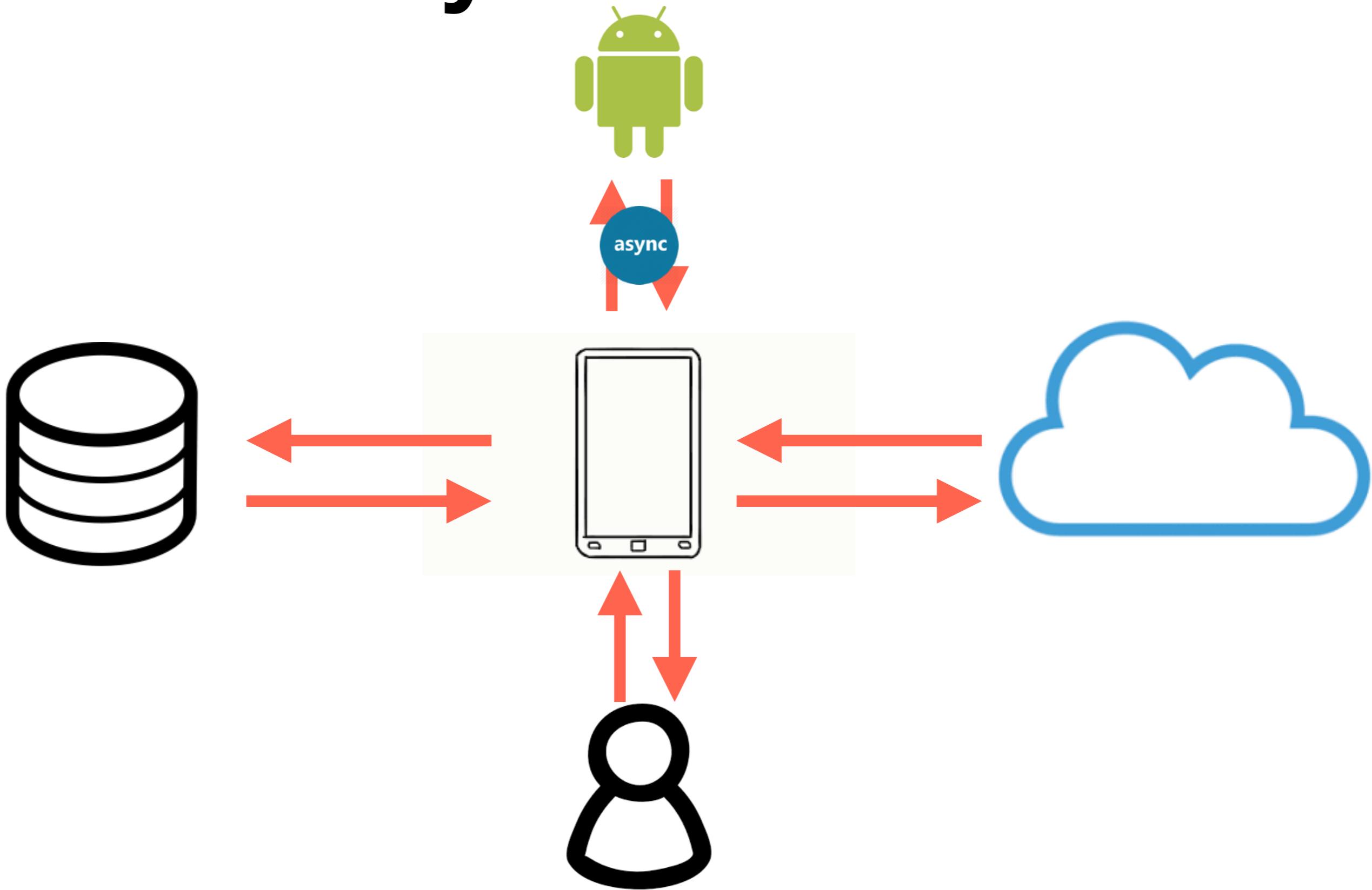
```
class UserActivity : AppCompatActivity() {  
    val um: UserManager = UserManagerImpl()  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        setContentView(R.layout.activity_user)  
        um.setName("John Doe", object : UserManager.Listener {  
            override fun success(user: User) {  
                um.setAge(42, object : UserManager.Listener {  
                    override fun success(user: User) {  
                        runOnUiThread {  
                            if (!textView.isDestroyed) user.name  
                            } textView.text = user.name  
                            Log.d("User", user.name)  
                        } }  
            override fun failed(error: UserException) {  
                override("Unable to update the user details", error)  
            }  
        })  
    }  
    override fun failed(error: UserException) {  
        //..Log("Unable to update the user details", error)  
    }  
}
```



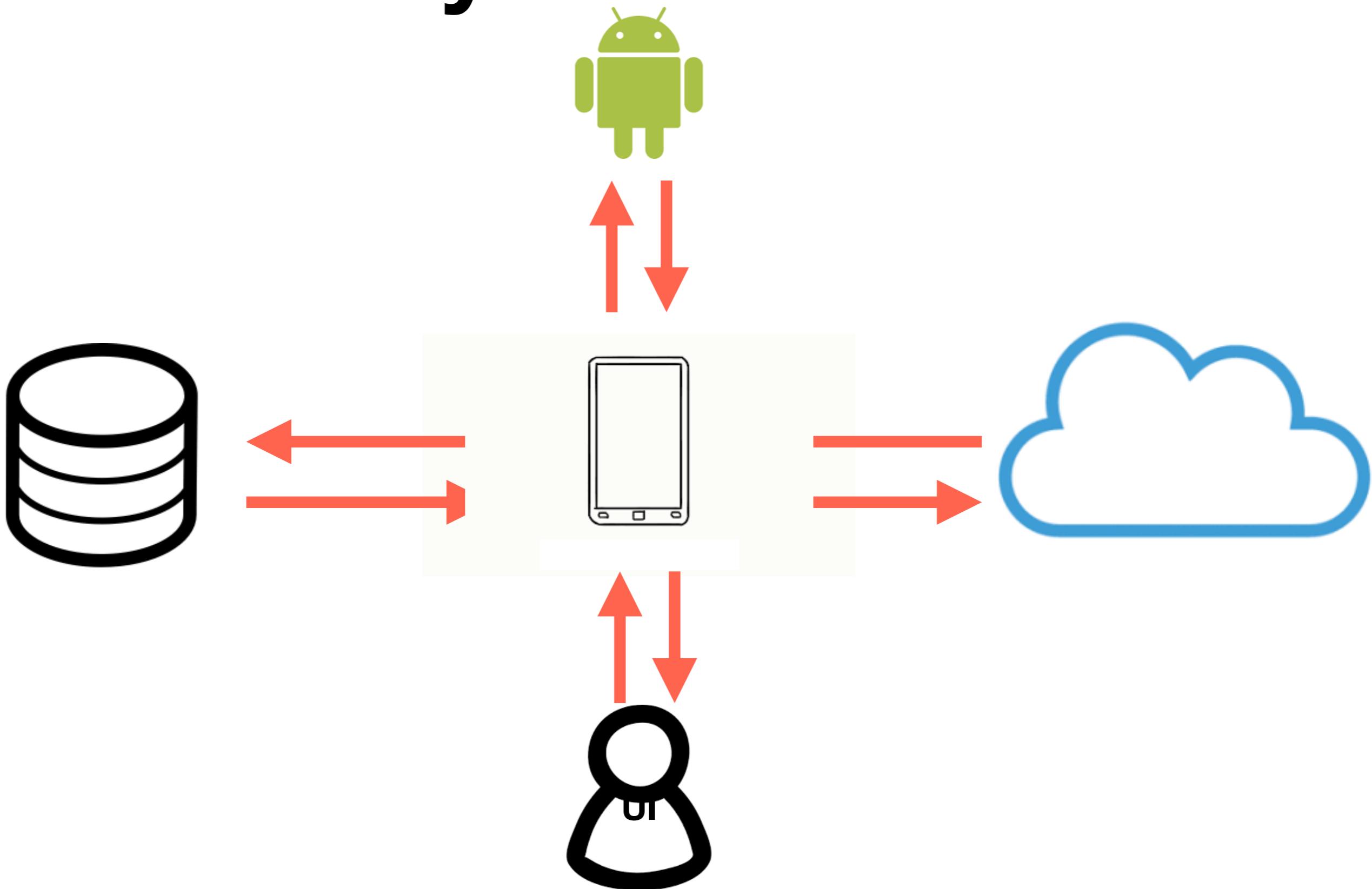
Why Reactive?



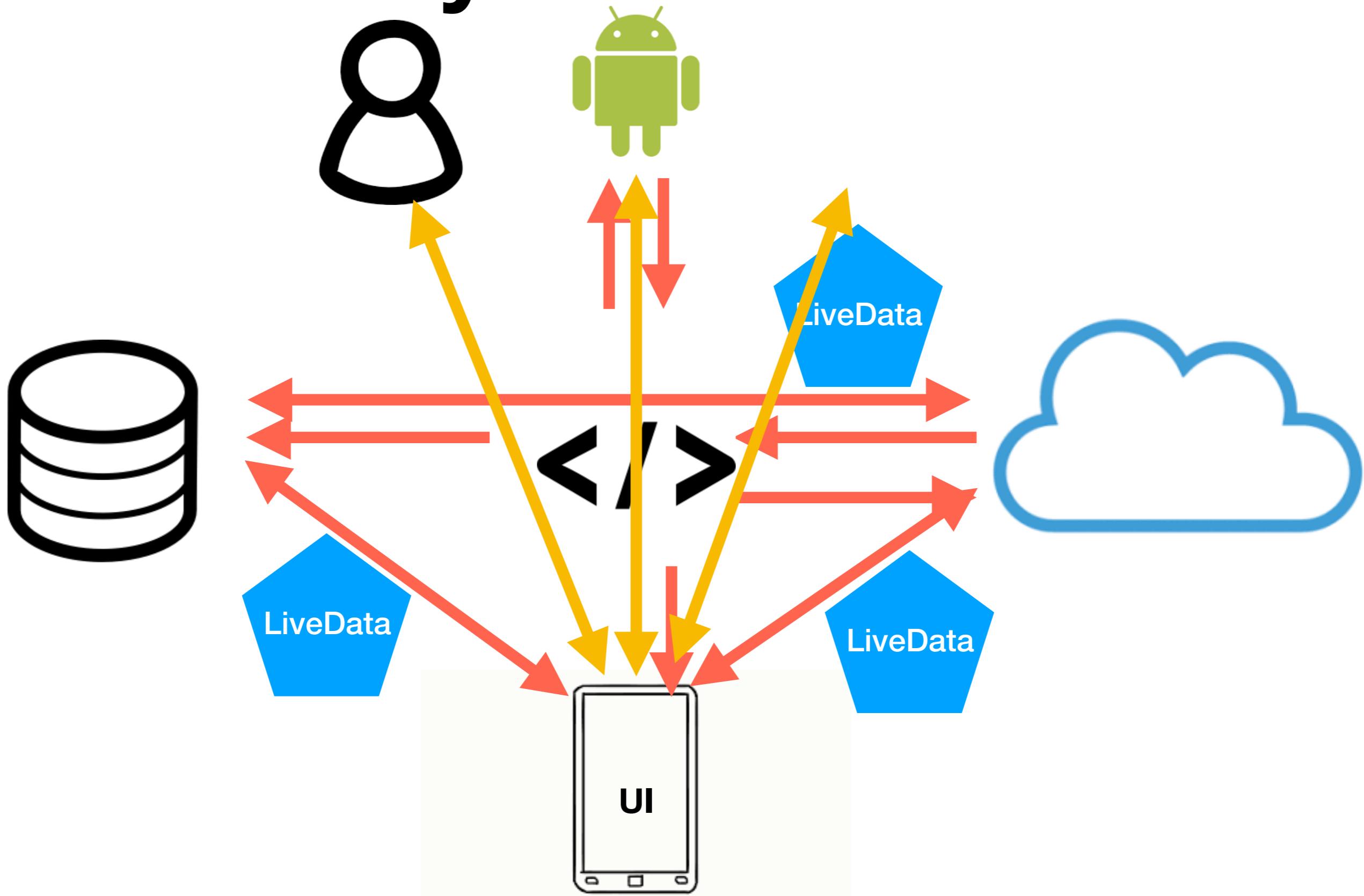
Why Reactive?



Why Reactive?



Why Reactive?



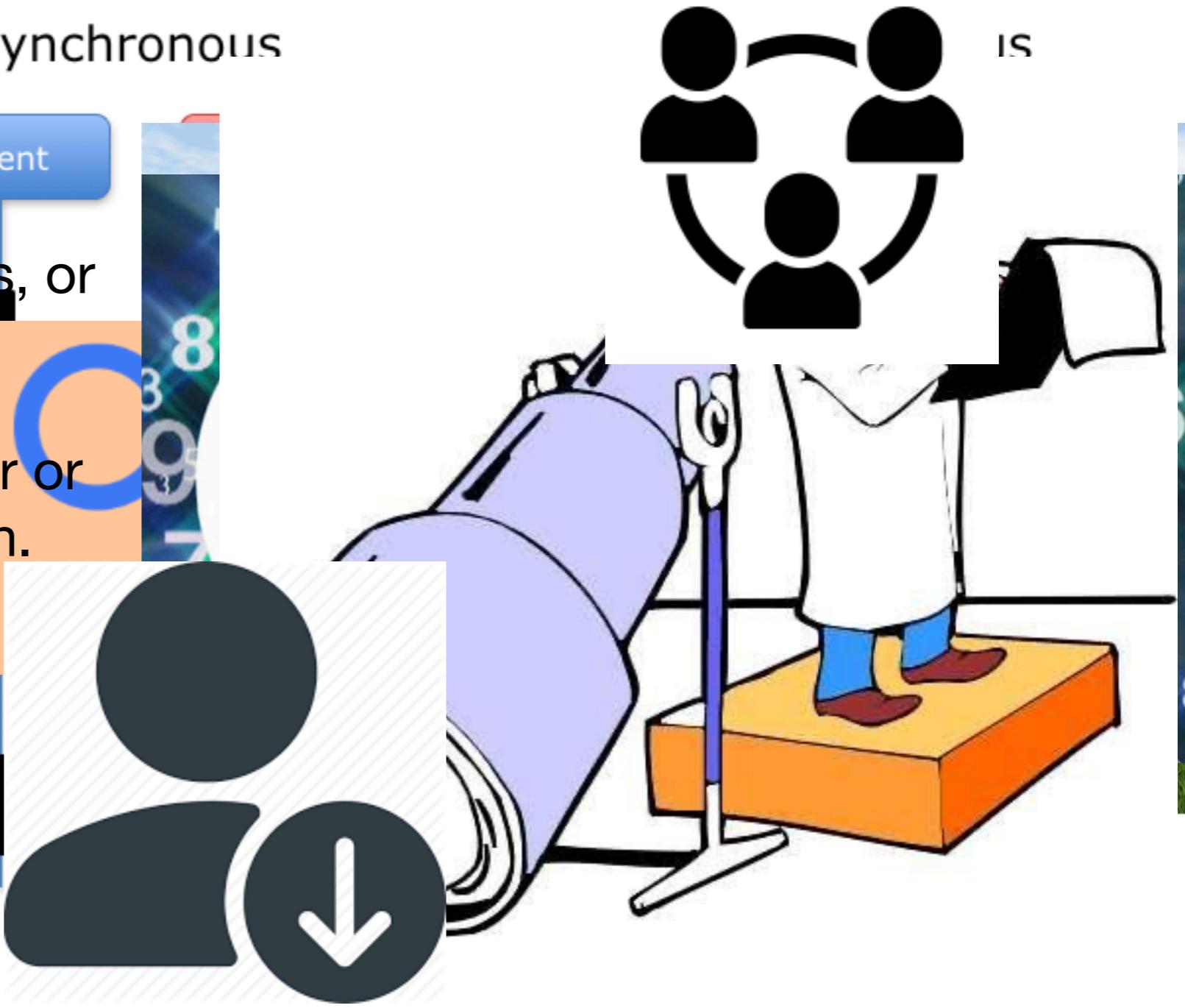
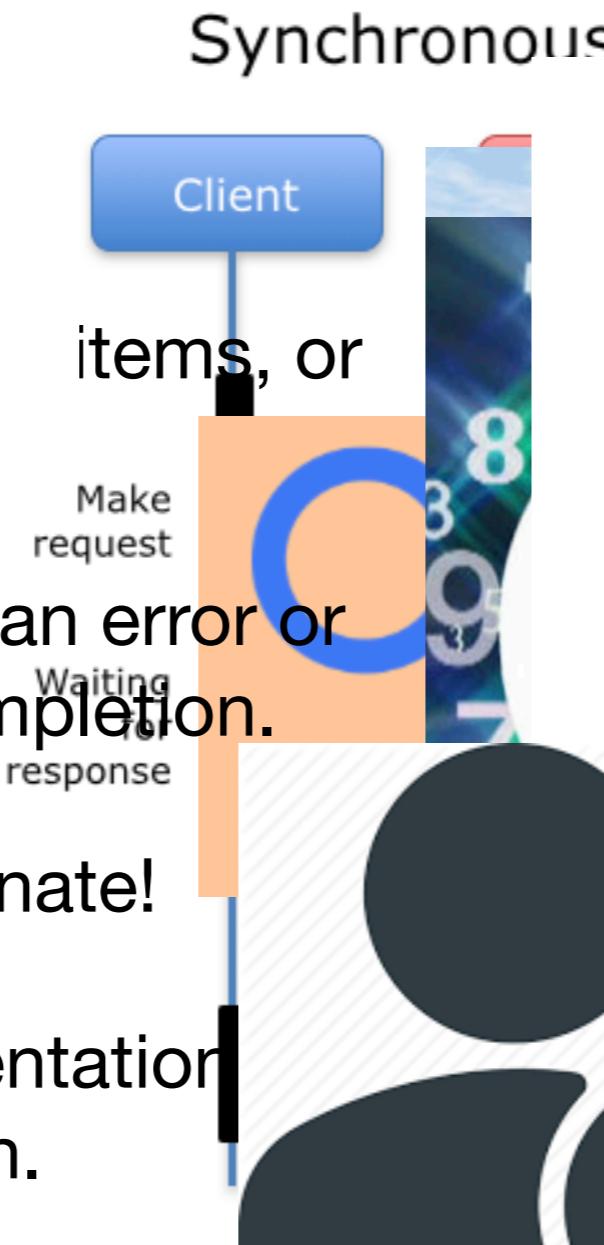
Rx{Java|Kotlin|Swift|Dart}

- A set of classes for representing sources of data.
- A set of classes for listening to data sources.
- A set of methods for modifying and composing the data.



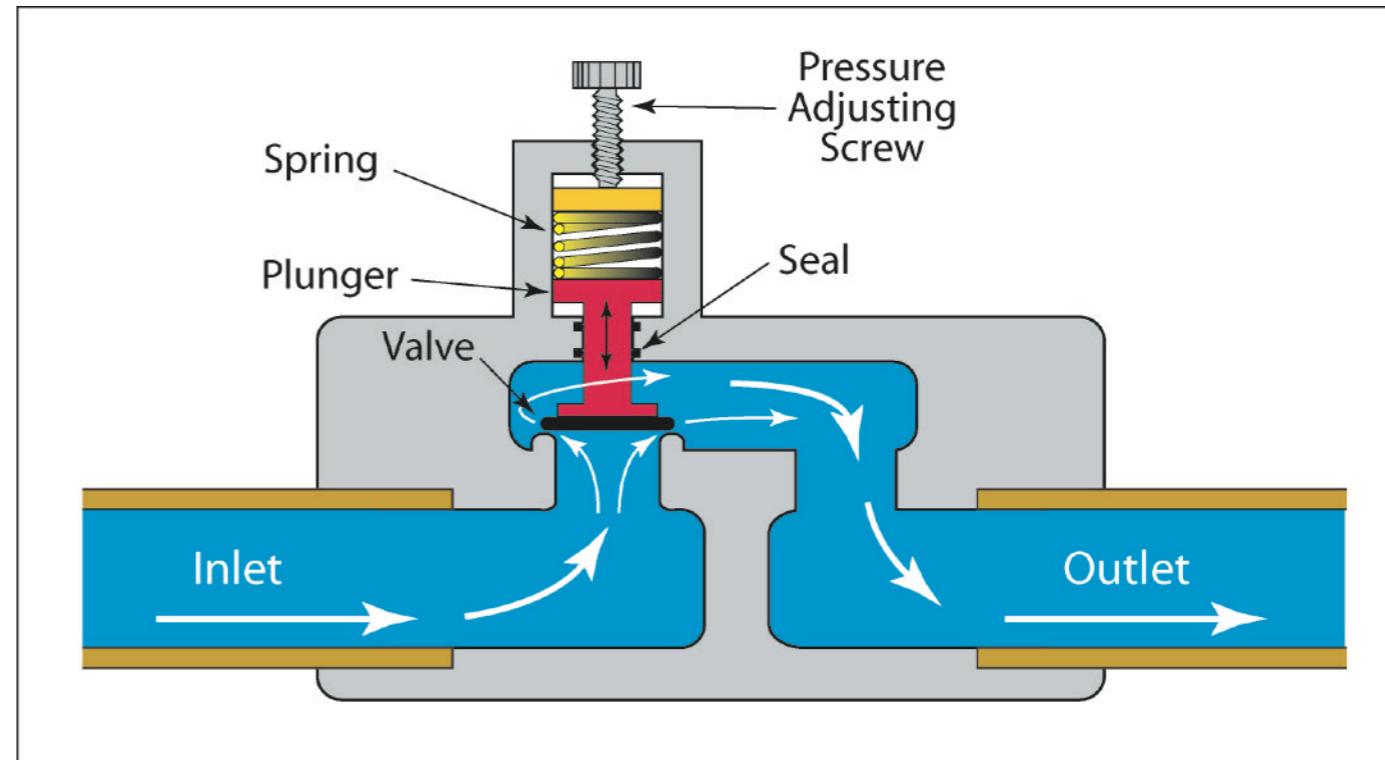
Sources

- Synchronous
- Synchronous
- Terminates with an error or succeeds to completion.
- May never terminate!
- Just an implementation of the Observer pattern.



Sources

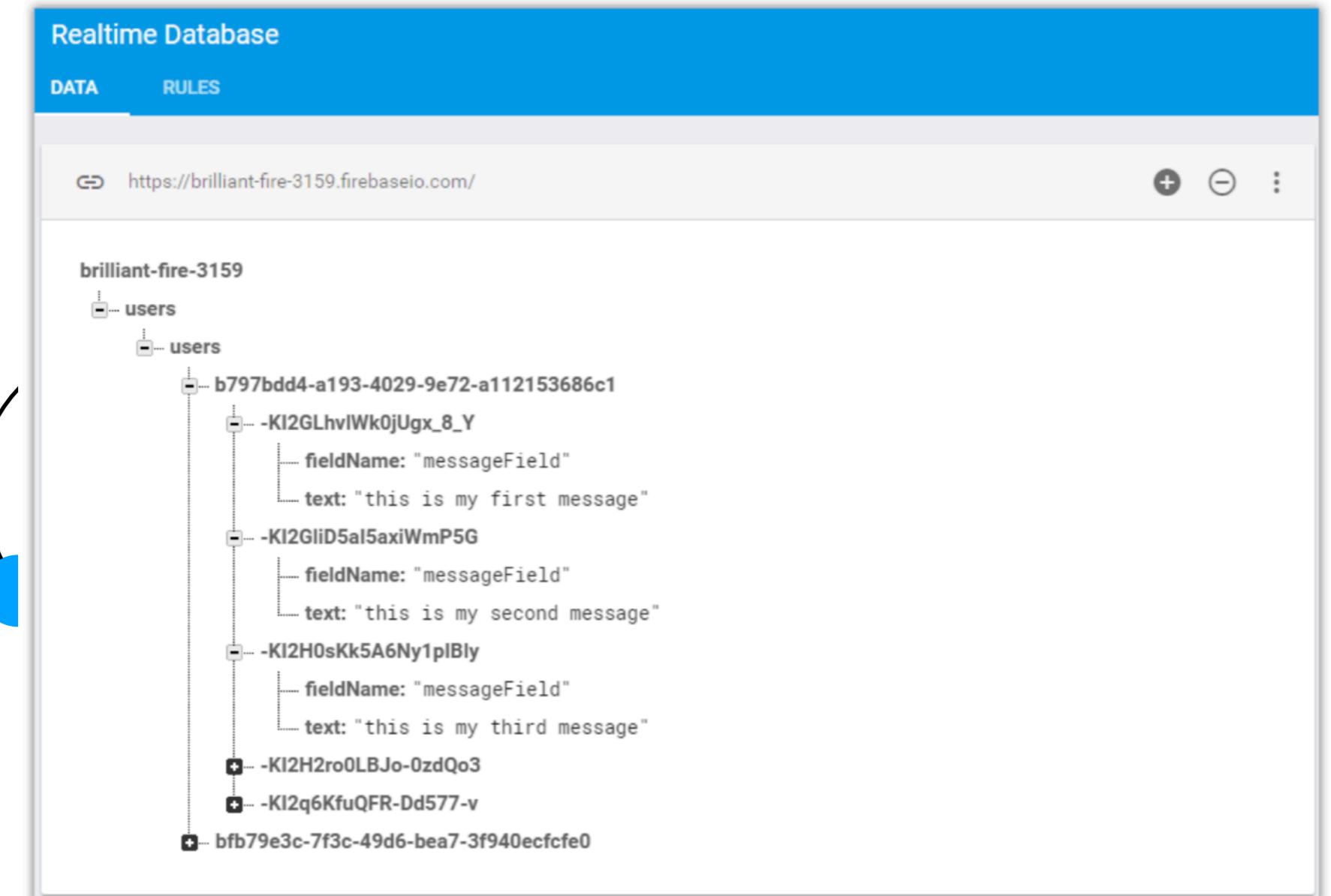
- Observable<T>
- Observable<T>
 - Emits 0 to N items.
 - Terminates with complete or error.
 - Terminates with complete or error.
- Observable<T> does not have backpressure.
- Flowable<T>
 - Emits 0 to N items.
 - Emits 0 to N items.
 - Terminates with complete or error.
 - Terminates with complete or error.
 - Has backpressure.



Flowable vs. Observable

```
val events: Observable<MotionEvent> = RxView.touches(paintView);
```

```
val users: Observable<List<User>> = db.createQuery(User::class.java).select(*).from();
```



Flowable vs. Observable

`Observable<MotionEvent>`

```
interface Observer<T>{
    fun onNext(t: T)
    fun onComplete()
    fun onError(e: Throwable)
    fun onSubscribe(d: Disposable)
}
```

```
interface Disposable{
    fun dispose()
}
```

`Flowable<User>`

```
interface Subscriber<T>{
    fun onNext(t: T)
    fun onComplete()
    fun onError(e: Throwable)
    fun onSubscribe(s: Subscription)
}
```

```
interface Subscription{
    fun cancel()
    fun request(r: Long)
}
```

Sources

```
interface UserManager {  
    fun getUser(): Observable<User>  
    fun setName(name: String)  
    fun setAge(age: Int)  
}
```

Specialized Sources

- Encoding subsets of Observable into the type system:

- Single



- Either succeeds with an item or an error.

- No backpressure support.

- Completable

Runnable

- Either completes or errors. Has no items!

- No backpressure support.

- Maybe

Optional

- Either succeeds with an item, completes with no items, or error.

- No backpressure support.



Sources

```
interface UserManager {  
    fun getUser(): Observable<User>  
    fun setName(name: String): Completable  
    fun setAge(age: Int): Completable  
}
```

Creating Sources

```
Flowable.just("Hello")
Flowable.just("Hello", "World")
```

```
Observable.just("Hello")
Observable.just("Hello", "World")
```

```
Maybe.just("Hello")
List.of("Hello", "World")
val list = array.toList()
Single.just("Hello")
Flowable.fromArray(array)
Flowable.fromIterable(list)
Observable.fromCallable {
  Observable.fromArray(array)
  Observable.fromIterable(list)
}
```

Creating Sources

```
val url = "https://example.com"  
  
val request = Request.Builder().url(url).build()  
val client = OkHttpClient()  
  
Observable.fromCallable {  
    client.newCall(request).execute()  
}
```

Create Sources

```
Observable.create<String> {  
    it.onNext("Hello")  
    it.onNext("World")  
}    it.onComplete()  
}
```

```
Observable.create(ObservableOnSubscribe<String> {  
    it.onNext("Hello")  
    it.onComplete()  
})
```

```
Observable.create(ObservableOnSubscribe<String>(  
    function = fun(it: ObservableEmitter<String>) {  
        it.onNext("Hello")  
        it.onComplete()  
    }))
```

Create Sources

```
val request = Request.Builder().url(url).build()
val client = OkHttpClient()

    Observable.create<View> {
        Observable.create<String> {
            it.setCancellable { textView.setOnClickListener(null) }
            val call = client.newCall(request)
            textView.setOnClickListener { v -> it.onNext(v) }
            it.setCancellable { call.cancel() }
            call.enqueue(object : Callback {
                override fun onFailure(call: Call, e: IOException) {
                    it.onError(e)
                }

                override fun onResponse(call: Call, response: Response) {
                    it.onNext(response.body().toString())
                    it.onComplete()
                }
            })
        }
    }
}
```

Observing Sources

Observable<MotionEvent>

```
interface Observer<T>{
    fun onNext(t: T)
    fun onComplete()
    fun onError(t: Throwable)
    fun onSubscribe(d: Disposable)
}
```

Flowable<User>

```
interface Subscriber<T>{
    fun onNext(t: T)
    fun onComplete()
    fun onError(t: Throwable)
    fun onSubscribe(s: Subscription)
}
```

```
interface Disposable{
    fun dispose()
}
```

```
interface Subscription{
    fun cancel()
    fun request(r: Long)
}
```

Observing Sources

```
val observable: Observable<String> = Observable.just("Hello")  
  
val disposable = observable.subscribeWith(object: DisposableObserver<String>() {  
    override fun onComplete() {  
        //... observable.subscribe(object: DisposableObserver<String>(){  
            override fun onComplete() {  
                //...  
            }  
            override fun onNext(t: String) {  
                //...  
            }  
            override fun onNext(t: String) {  
                //...  
            }  
            override fun onError(e: Throwable) {  
                //...  
            }  
            override fun onError(e: Throwable) {  
                //...  
            }  
        })  
        observable.subscribe(observer)  
    }  
})  
disposable.dispose()  
observer.dispose()  
How to we dispose?
```

Rx{Java|Kotlin|Swift|Dart}

- A set of classes for representing sources of data.
- A set of classes for listening to data sources.
- A set of methods for modifying and composing the data and composing the data.



Operators



- Manipulate or combine data in some way.
- Manipulate threading in some way.
- Manipulate emissions in some way.

Operators

```
val greeting = Observable.just("Hello")
val yelling = greeting.map{it.toUpperCase() }
```

Operators

```
class UserActivity : AppCompatActivity() {  
    val userManager: UserManager by lazy { AppUserManagerImpl() }  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        setContentView(R.layout.activity_user)  
        userManager.setName("John Doe", object : UserManager.Listener {  
            override fun onSuccess(user: User) {  
                userManager.setAge(42, object : UserManager.Listener {  
                    override fun onSuccess(user: User) {  
                        runOnUiThread {  
                            if (!isDestroyed) {  
                                textView.text = user.name  
                            }  
                        }  
                    }  
                }  
            }  
        })  
        override fun onFailure(error: UserException) {  
            Log.e("Unable to update the user details", error.message)  
        }  
    }  
}
```

Operators

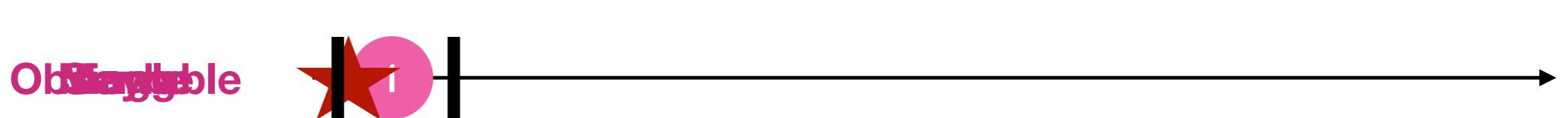
```
val url = "https://example.com"
val user: Observable<User> = um.getUser()
    val mainThreadUser = user.observeOn(AndroidSchedulers.mainThread())
val request = Request.Builder().url(url).build()
val client = OkHttpClient()

val response = Observable.fromCallable { client.newCall(request).execute() }
    .subscribeOn(Schedulers.io()).map { it.body()?.string() }
    .flatMap { Observable.fromArray(it.split(" ")) }
    .observeOn(AndroidSchedulers.mainThread())
```

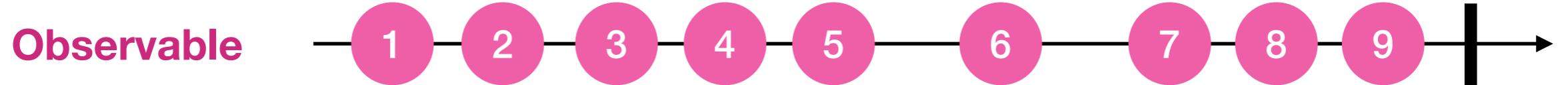
Operators



firstElement()



Operators



ignoreElements()



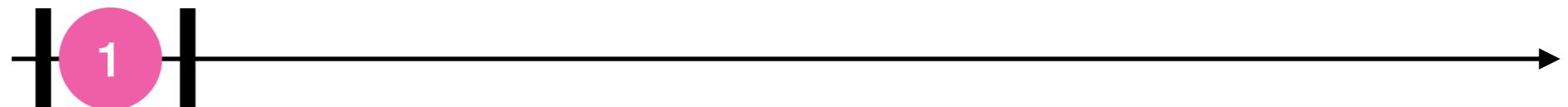
Operators

Flowable



firstElement()

Single



Operators



ignoreElements()



Being Reactive

```
// onCreate
val disposables = CompositeDisposable()
disposables.add(user.getUser()
    .observeOn(AndroidSchedulers.mainThread())
    .subscribeWith(object: DisposableObserver<User>{
        override fun onNext(user: User) {
            textView.text = user.toString()
        }
        override fun onError(e: Throwable) {
        }
        override fun onComplete() {
        }
    })
))

// onDestroy
disposable.dispose()
```

DEMO

Dependencies

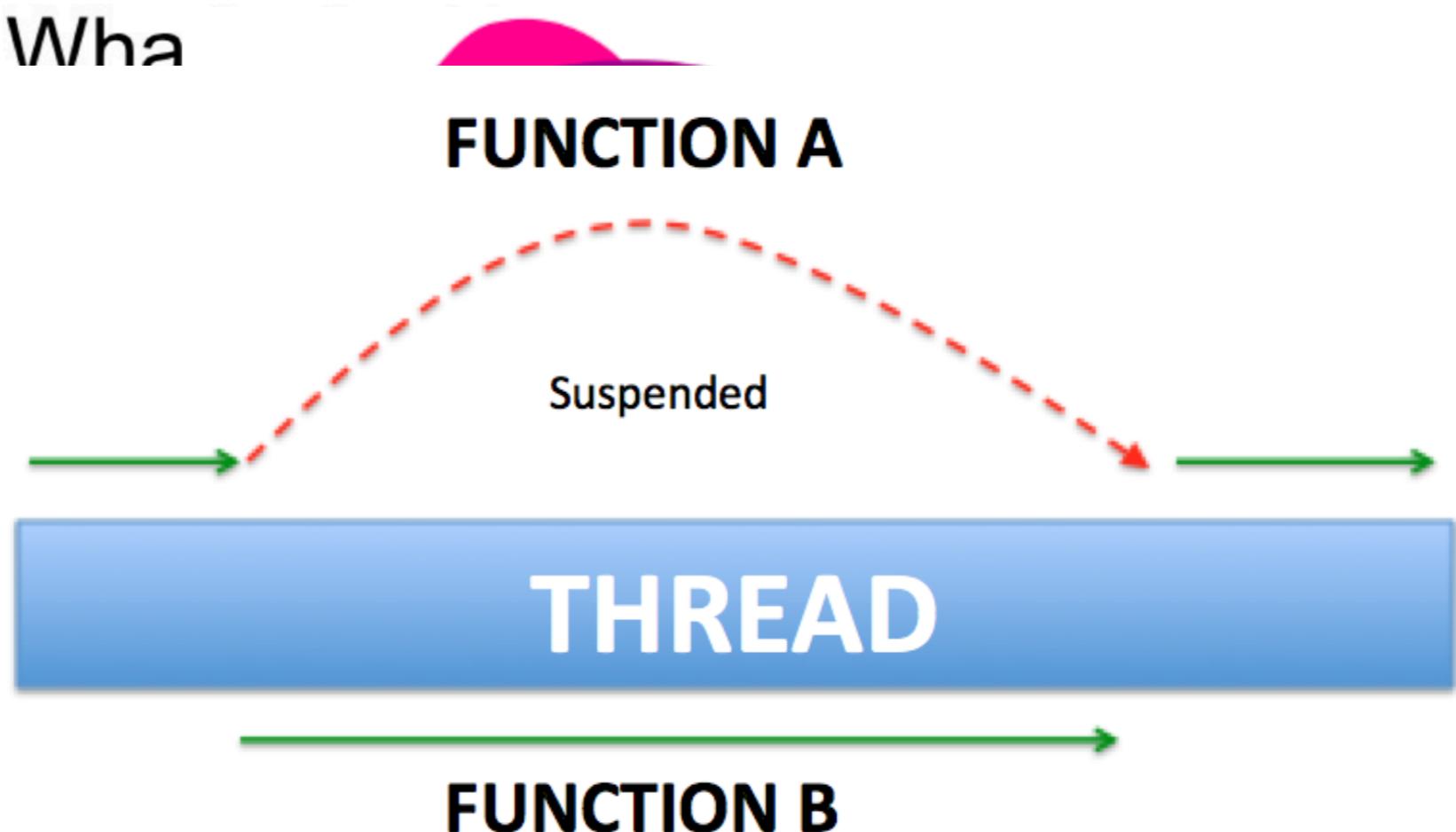
```
dependencies {  
    implementation 'io.reactivex.rxjava2:rxkotlin:2.2.0'  
    implementation 'io.reactivex.rxjava2:rxandroid:2.0.1'  
}
```

Options

- Callbacks
- Futures
- Promises
- Rx
- Coroutines

```
describe('.totalValue', function(){  
    it('should calculate the total value of items in a space', function(done){  
        ...  
    });  
});
```

What's



```
    boo]  
}  
...  
});  
});
```

Coroutines

```
fun requestToken(): Token {  
    // make a token request and waits  
    reblocks thread while waiting for result  
    return token  
}  
  
fun createPost(token: Token, item: Item): Post {  
    fun logtePost(token: $token, item: $item) Post {  
        LogndBoštěngetem $itemheusengetokenwaits  
        reendsposte item to the server and waits  
        return post  
    }  
}  
  
fun processPost(post: Post) {  
    fun processPost(post: Post) {  
        Logpřečešeníhoapost  
        log("Processing a $post")  
    }  
}  
  
fun postItem(item: Item) {  
    fun postItem(item: Item) {  
        val token==createBlokken, item)  
        val processPost(createPost(token, item)  
        processPost(post)  
    }  
}
```



Coroutines

```
suspend fun requestToken(): Token {  
    // make a token request and suspends  
    return token  
}  
  
suspend fun createPost(token: Token, item: Item): Post {  
    log("Posting an $item using $token")  
    //sends the item to the server and waits  
    return post  
}  
  
fun processPost(post: Post) {  
    // processing the post  
    log("Processing a $post")  
}  
  
suspend fun postItem(item: Item) {  
    val token = requestToken()  
    val post = createPost(token, item)  
    processPost(post)  
}
```



Bonuses

Regular exception handling:

```
->   for ((token, item) in list) {  
    createPost(token, item)  
} catch (e: BadTokenException) {  
    // ...  
}
```

Regular higher-order function:

```
->   file.readLines().foreach { line ->  
    createPost(token, line.toItem())  
}
```

Any of: **foreach**, **let**, **apply**, **repeat**, **filter**, **map**, **use**, etc.

Builders

```
suspend fun requestToken(): Token {  
    // make a token request and suspends  
    return token  
}  
  
suspend fun createPost(token: Token, item: Item): Post {  
    logd("Posting an $item using $token")  
    //sends the item to the server and waits  
    return post  
}
```

```
fun processPost(post: Post) {  
    // processing the post  
    logd("Processing a $post")  
}
```

```
suspend fun item(item: Item): Item {  
    GlobalScope.launch(Coroutines.Main) {  
        val token = requestToken()  
        val post = createPost(token, item)  
        processPost(post)  
    }  
}
```

Suspend function



'requestToken' should be called only from a coroutine or another suspend function



Builders

```
public fun CoroutineScope.launch(
    context: CoroutineContext = EmptyCoroutineContext,
    start: CoroutineStart = CoroutineStart.DEFAULT,
    block: suspend CoroutineScope.() -> Unit
): Job {
    val newContext = newCoroutineContext(context)
    val coroutine = if (start.isLazy)
        LazyStandaloneCoroutine(newContext, block) else
        StandaloneCoroutine(newContext, active = true)
    coroutine.start(start, coroutine, block)
    return coroutine
}
```

DEMO

Dependencies

```
dependencies {  
    implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-core:1.3.2'  
    implementation "org.jetbrains.kotlinx:kotlinx-coroutines-android:1.3.0"  
}
```

Usage

```
class MyViewModel : ViewModel() {  
    private val _result = MutableLiveData<String>()  
    val result: LiveData<String> = _result
```

```
    init {  
        viewModelScope.launch {  
            val computationalResult = doComputation()  
            _result.value = computationalResult  
        }  
    }  
}
```

implementation "androidx.lifecycle:lifecycle-viewmodel-ktx:2.2.0-rc01"

Usage

```
class MyNewViewModel : ViewModel() {  
    val result = liveData {  
        emit(doComputation())  
    }  
}
```



implementation "androidx.lifecycle:lifecycle-livedata-ktx:2.2.0-rc01"

Flow

```
suspend fun foo(): List<Int> {  
    -> delay(1000) // pretend we are doing something asynchronous here  
    return listOf(1, 2, 3)  
}
```

Output:

```
fun main() = runBlocking<Unit> {  
    -> foo().forEach { value -> println(value) }  
}
```

1
2
3

Flow

```
fun foo(): Flow<Int> = flow { // flow builder
    for (i in 1..3) {
        
        delay(100) // pretend we are doing something useful here
        emit(i) // emit next value
    }
}

fun main() = runBlocking<Unit> {
    // Launch a concurrent coroutine to check if the main thread is blocked
    launch {
        for (k in 1..3) {
            println("I'm not blocked $k")
            
            delay(100)
        }
    }
    // Collect the flow
    
    foo().collect { value -> println(value) }
}
```

Output:

```
I'm not blocked 1
1
I'm not blocked 2
2
I'm not blocked 3
3
```

Flows are cold

```
fun foo(): Flow<Int> = flow {  
    println("Flow started")  
    for (i in 1..3) {  
        delay(100)  
        emit(i)  
    }  
}  
  
fun main() = runBlocking<Unit> {  
    println("Calling foo...")  
    val flow = foo()  
    println("Calling collect 1.")  
    flow.collect { value -> println(value) }  
    println("Calling collect again...")  
    flow.collect { value -> println(value) }  
}
```

Output:

Calling foo...
Calling collect...
Flow started
1
Calling collect again...
Flow started
1
Calling collect again...

Flow Cancellation

```
fun fooCancellation(): Flow<Int> = flow {  
    for (i in 1..3) {  
        delay(100)  
        println("Emitting $i")  
        emit(i)  
    }  
}
```

Output:

```
Emitting 1  
1  
Emitting 2  
2  
Done
```

```
fun main() = runBlocking<Unit> {  
    withTimeoutOrNull(250) { // Timeout after 250ms  
        fooCancellation().collect { value -> println(value) }  
    }  
    println("Done")  
}
```

Flow Operators

```
suspend fun performRequest(request: Int): String {  
    -$→ delay(1000) // imitate long-running asynchronous work  
    return "response $request"  
}
```

Output:

```
fun main() = runBlocking<Unit> {  
    (1..3).asFlow() // a flow of requests  
    -$→ .map { request -> performRequest(request) }  
    -$→ .collect { response -> println(response) }  
}
```

response 1
response 2
response 3

Transform Operator

```
suspend fun performRequest(request: Int): String {  
    -$→ delay(1000) // imitate long-running asynchronous work  
    return "response $request"  
}  
  
(1..3).asFlow() // a flow of requests  
    .transform { request ->  
        -$→ emit("Making request $request")  
        -$→ emit(performRequest(request))  
    }  
    -$→ .collect { response -> println(response) }  
        output:
```

Making request 1

response 1

Making request 2

response 2

Making request 3

response 3

Size-limiting Operators

```
fun numbers(): Flow<Int> = flow {  
    try {  
        emit(1)  
        emit(2)  
        println("This line will not execute")  
        emit(3)  
    } finally {  
        println("Finally in numbers")  
    }  
}
```

`@ExperimentalCoroutinesApi`

```
fun main() = runBlocking<Unit> {  
    numbers()  
        .take(2) // take only the first two  
        .collect { value -> println(value) }  
}
```

Output:

1
2

Finally in numbers

Terminal Flow Operators

```
@ExperimentalCoroutinesApi
```

```
fun main() = runBlocking<Unit> {  
    val sum = (1..5).asFlow()  
        .map { it * it } // squares of numbers from 1 to 5  
        .reduce { a, b -> a + b } // sum them (terminal operator)  
    println(sum)  
}
```



Output: 55

Lecture outcomes

- Understand the reactive programming (rx) concepts.
- Use rx to re-write the application logic.
- Design a real time application logic against a real time backend.
- Understand coroutines and flow.

