

Lecture #7

Reactive Programming

Mobile Applications 2018-2019

Why Reactive?

- Unless you can model your entire system synchronously ...



Why Reactive?

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



Why Reactive?

```
interface UserManager {  
    fun getUser(): User  
}
```



Why Reactive?

```
interface UserManager {  
    fun 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)  
}  
  
val um: UserManager = UserManagerImpl()  
Logd(um.getUser())
```



Why Reactive?

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

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

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



Why Reactive?

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

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

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



Why Reactive?

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



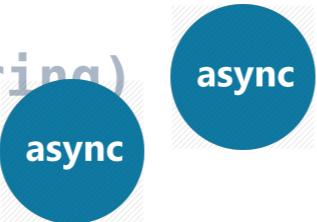
Why Reactive?

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



Why Reactive?

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

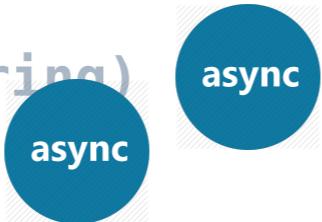


```
um.setName("John Doe")  
logd(um.getUser())
```

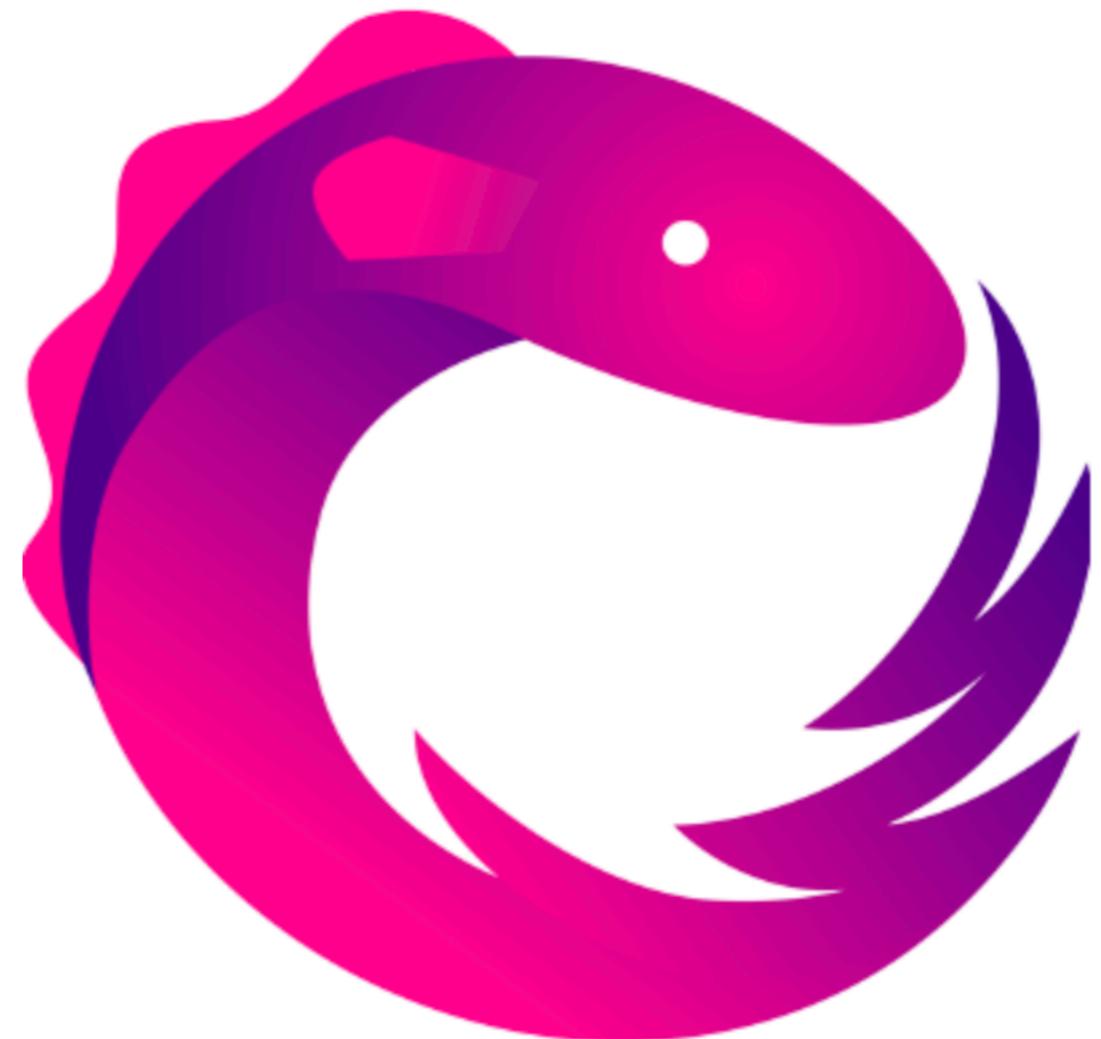


Why Reactive?

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

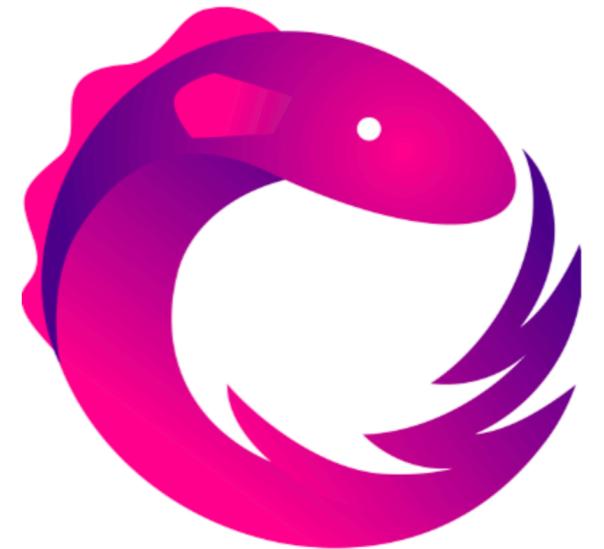


```
um.setName("John Doe")  
logd(um.getUser())
```



Why Reactive?

```
interface UserManagerV2 {  
    fun getUser(): User  
    fun setName(name: String, callback: Runnable)  
    fun setAge(age: Int, callback: Runnable)  
}
```



Why Reactive?

```
interface UserManagerV2 {  
    fun getUser(): User  
    fun setName(name: String, callback: Runnable)  
    fun setAge(age: Int, callback: Runnable)  
}
```

```
val um: UserManagerV2 = UserManagerV2Impl()  
logd(um.getUser())
```



Why Reactive?

```
interface UserManagerV2 {  
    fun getUser(): User  
    fun setName(name: String, callback: Runnable)  
    fun setAge(age: Int, callback: Runnable)  
}
```

```
val um: UserManagerV2 = UserManagerV2Impl()  
logd(um.getUser())
```

```
um.setName("John Doe", Runnable {  
    logd(um.getUser())  
})
```

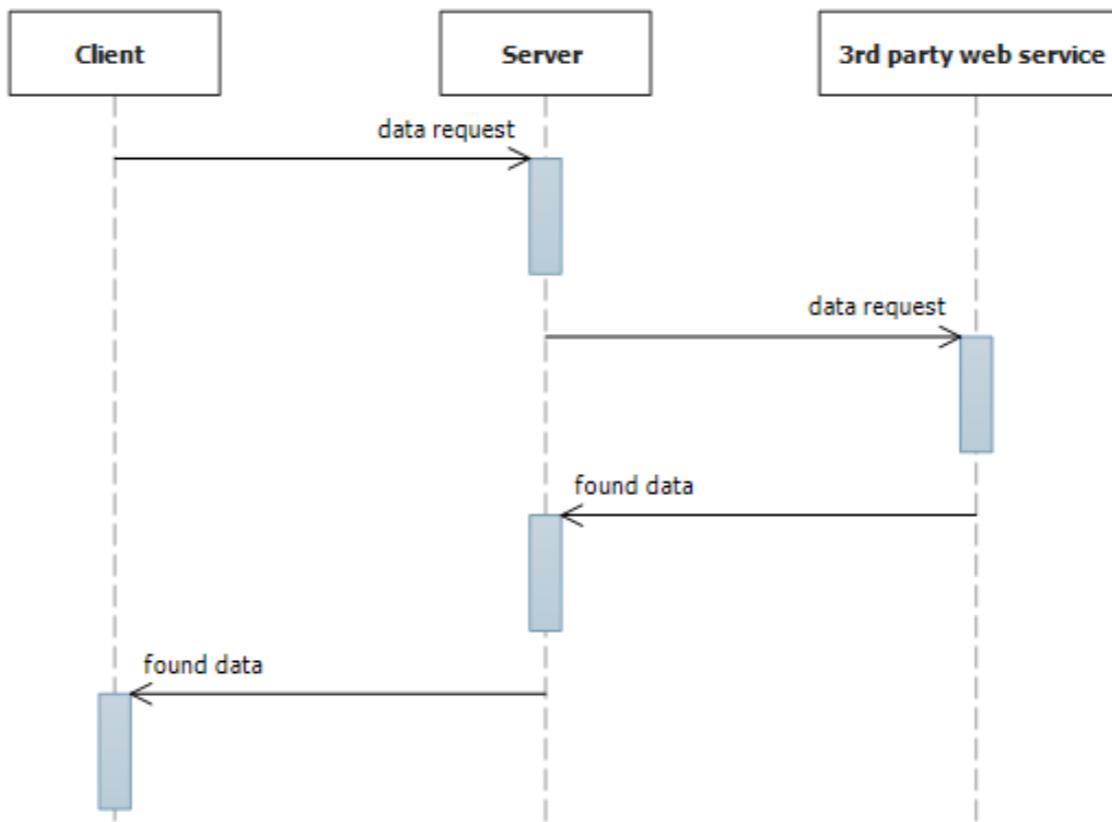


Why Reactive?

```
interface UserManagerV2 {  
    fun getUser(): User  
    fun setName(name: String, callback: Runnable)  
    fun setAge(age: Int, callback: Runnable)  
}
```

```
val um: UserManagerV2 = UserManagerV2Impl()  
logd(um.getUser())
```

```
um.setName("John Doe", Runnable {  
    logd(um.getUser())  
})
```

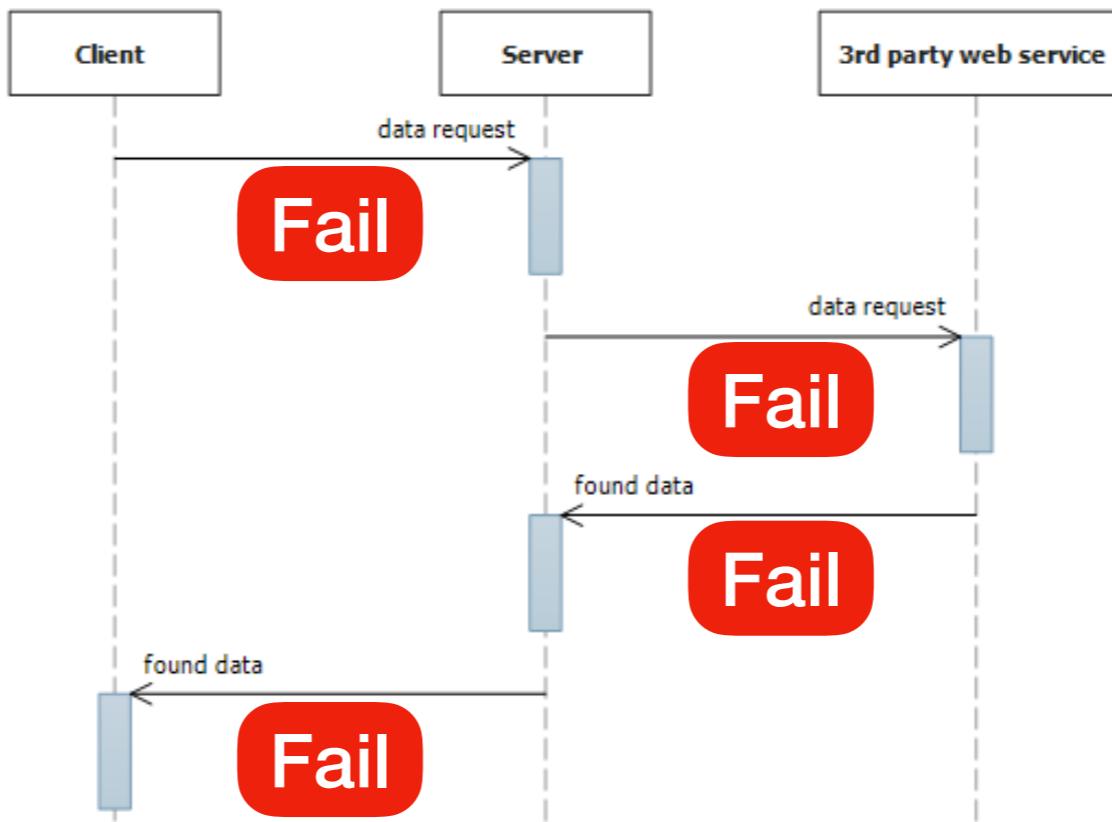


Why Reactive?

```
interface UserManagerV2 {  
    fun getUser(): User  
    fun setName(name: String, callback: Runnable)  
    fun setAge(age: Int, callback: Runnable)  
}
```

```
val um: UserManagerV2 = UserManagerV2Impl()  
logd(um.getUser())
```

```
um.setName("John Doe", Runnable {  
    logd(um.getUser())  
})
```



Why Reactive?

```
interface UserManagerV3 {  
    fun getUser(): User  
    fun setName(name: String, listener: Listener): void  
    fun setAge(age: Int, listener: Listener): void  
  
    interface Listener {  
        fun success(user: User)  
        fun failed(error: UserException)  
    }  
}
```



Why Reactive?

```
interface UserManagerV3 {  
    fun getUser(): User  
    fun setName(name: String, listener: Listener): void  
    fun setAge(age: Int, listener: Listener): void  
  
    interface Listener {  
        fun success(user: User)  
        fun failed(error: UserException)  
    }  
}  
  
val um: UserManagerV3 = UserManagerV3Impl()  
logd(um.getUser())  
  
um.setName("John Doe", object : UserManagerV3.Listener {  
    override fun success(user: User) {  
        logd(user)  
    }  
  
    override fun failed(error: UserException) {  
        loge("Unable to update the user details", error)  
    }  
})
```



Why Reactive?

```
val um: UserManagerV3 = UserManagerV3Impl()
logd(um.getUser())

um.setName("John Doe", object : UserManagerV3.Listener {
    override fun success(user: User) {
        logd(user)
    }

    override fun failed(error: UserException) {
        loge("Unable to update the user details", error)
    }
})

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)
    }
})
```



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)  
            }  
        })  
    }  
  
    override fun failed(error: UserException) {  
        loge("Unable to update the user details", error)  
    }  
})
```





Why Reactive?

```
um.setName("John Doe", object : UserManager.Listener {  
    override fun success(user: User) {  
        um.setAge(42, object : UserManager.Listener {  
            override fun success(user: User) {  
                logd(user)  
            }  
  
            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)  
    }  
})
```



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) {  
                        logd(user)  
                    }  
  
                    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)  
            }  
        })  
    }  
}
```



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) {  
                        textView.text = user.name  
                        Logd(user)  
                    }  
                    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)  
            }  
        })  
    }  
}
```



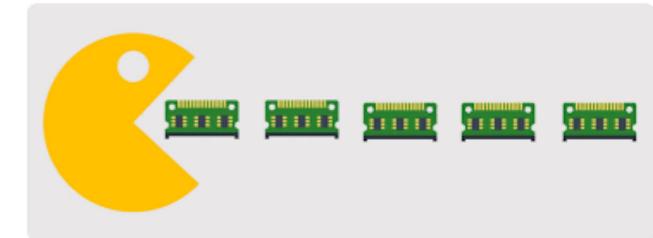
Why Reactive?

```
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) {  
                        if (!isDestroyed) {  
                            textView.text = user.name  
                        }  
                        logd(user)  
                    }  
                    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)  
            }  
        })  
    }  
}
```



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) {  
                        if (!isDestroyed) {  
                            textView.text = user.name  
                        }  
                        logd(user)  
                    }  
                    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)  
            }  
        })  
    }  
}
```

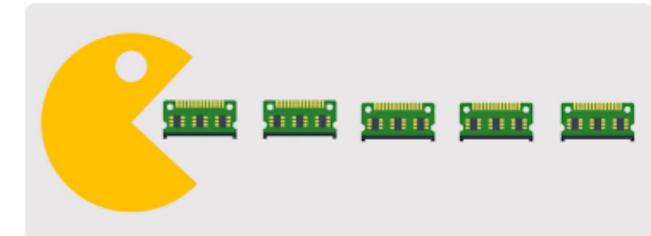


“A small leak will sink a great ship.”
Benjamin Franklin



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) {  
                        if (!isDestroyed) {  
                            textView.text = user.name  
                        }  
                        logd(user)  
                    }  
                    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) {  
                        if (!isDestroyed) {  
                            textView.text = user.name  
                        }  
                        logd(user)  
                    }  
                    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)  
            }  
        })  
    }  
}
```





Why Reactive?

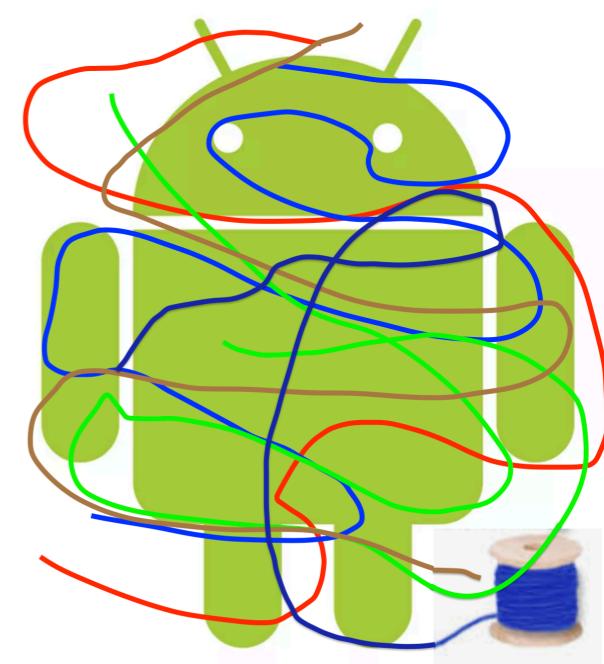
```
class UserActivity : AppCompatActivity() {  
    val um: UserManager = UserManagerImpl()  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        setContentView(R.layout.activity_user)  
        um.setName("John Doe", object : UserManager.Listener {  
            override fun success(user: User) {  
                um.setAge(42, object : UserManager.Listener {  
                    override fun success(user: User) {  
                        if (!isDestroyed) {  
                            textView.text = user.name  
                        }  
                        Logd(user)  
                    }  
                    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)  
            }  
        })  
    }  
}
```



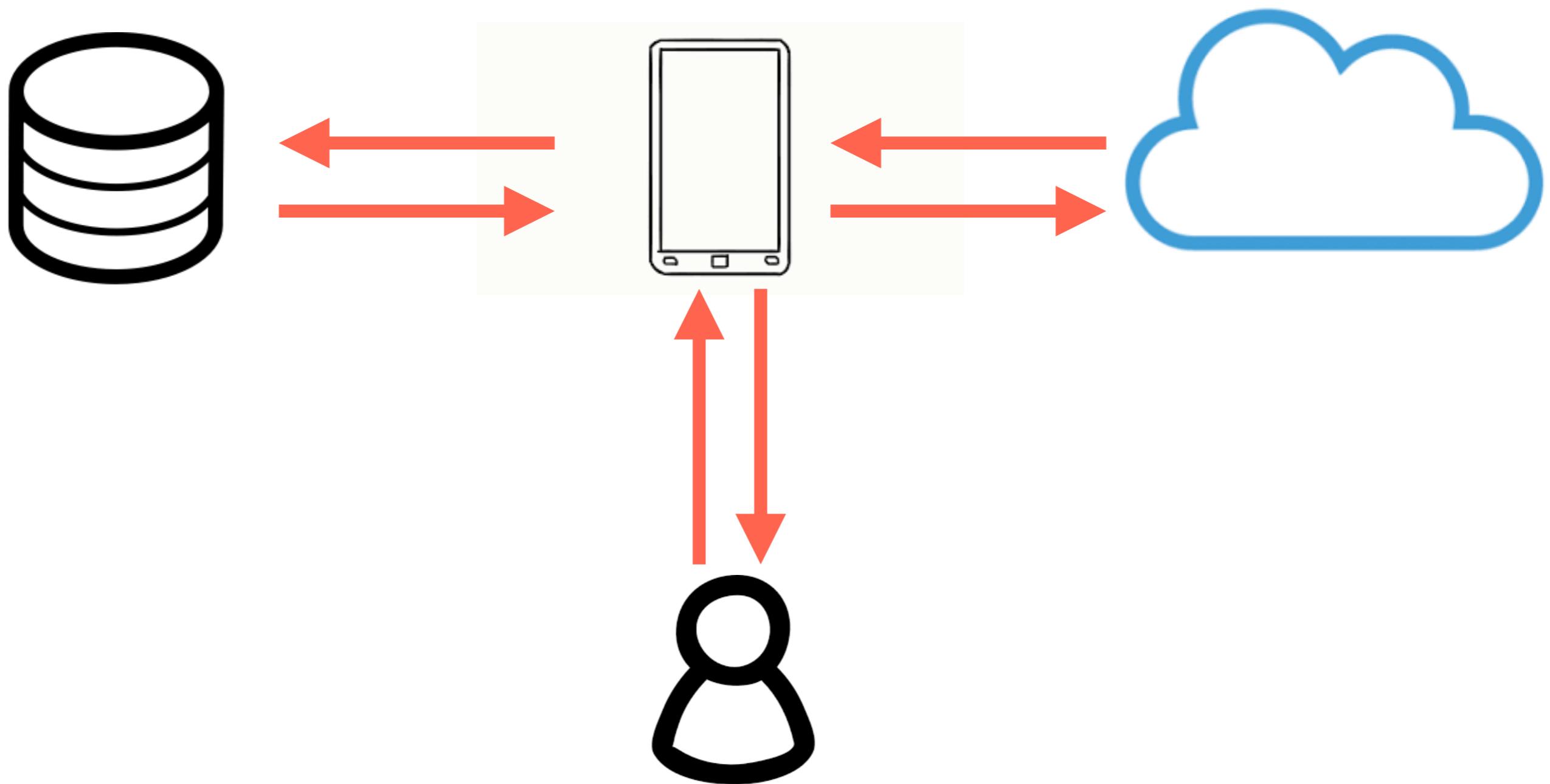


Why Reactive?

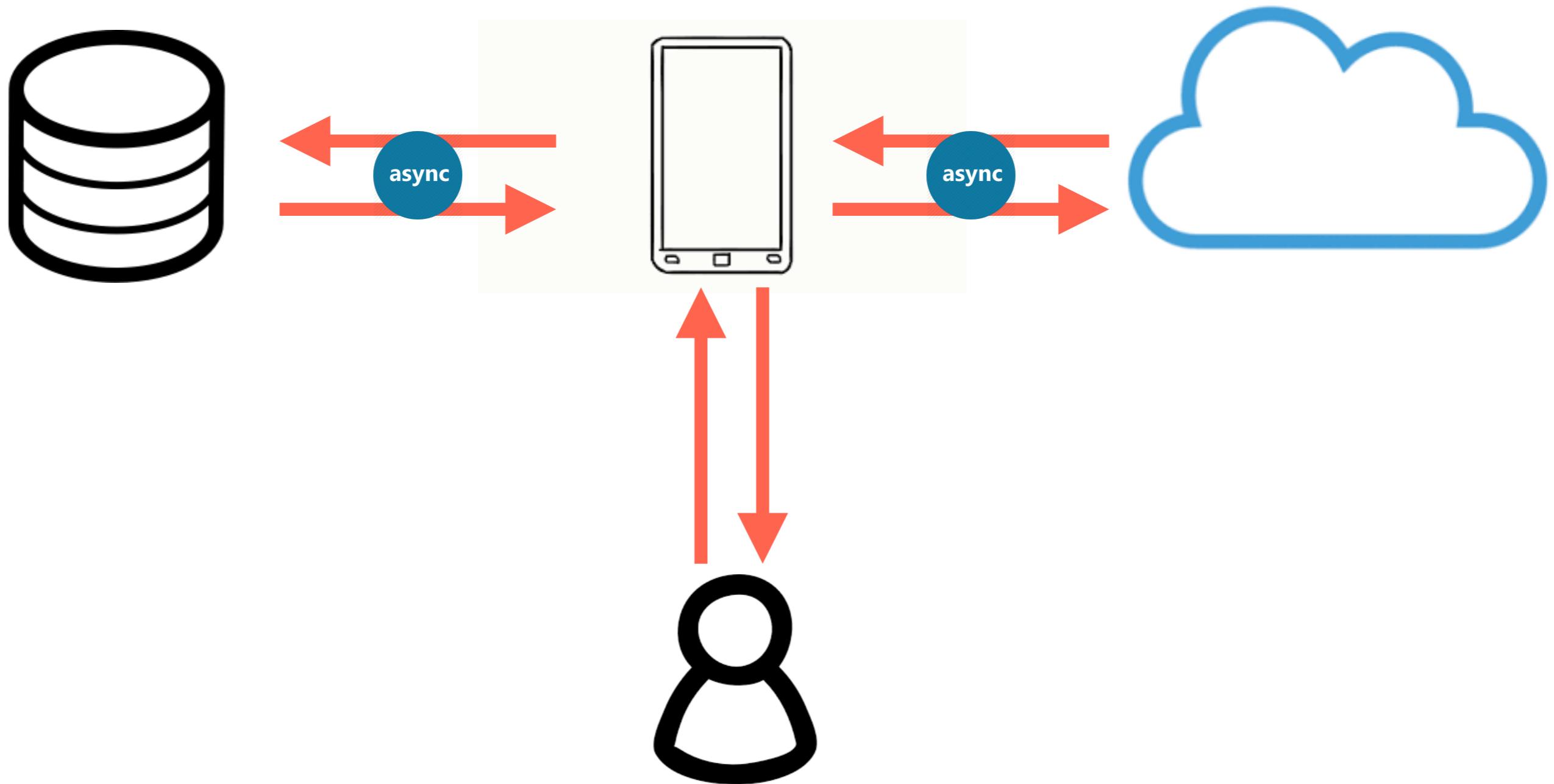
```
class UserActivity : AppCompatActivity() {  
    val um: UserManager = UserManagerImpl()  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
        setContentView(R.layout.activity_user)  
        um.setName("John Doe", object : UserManager.Listener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



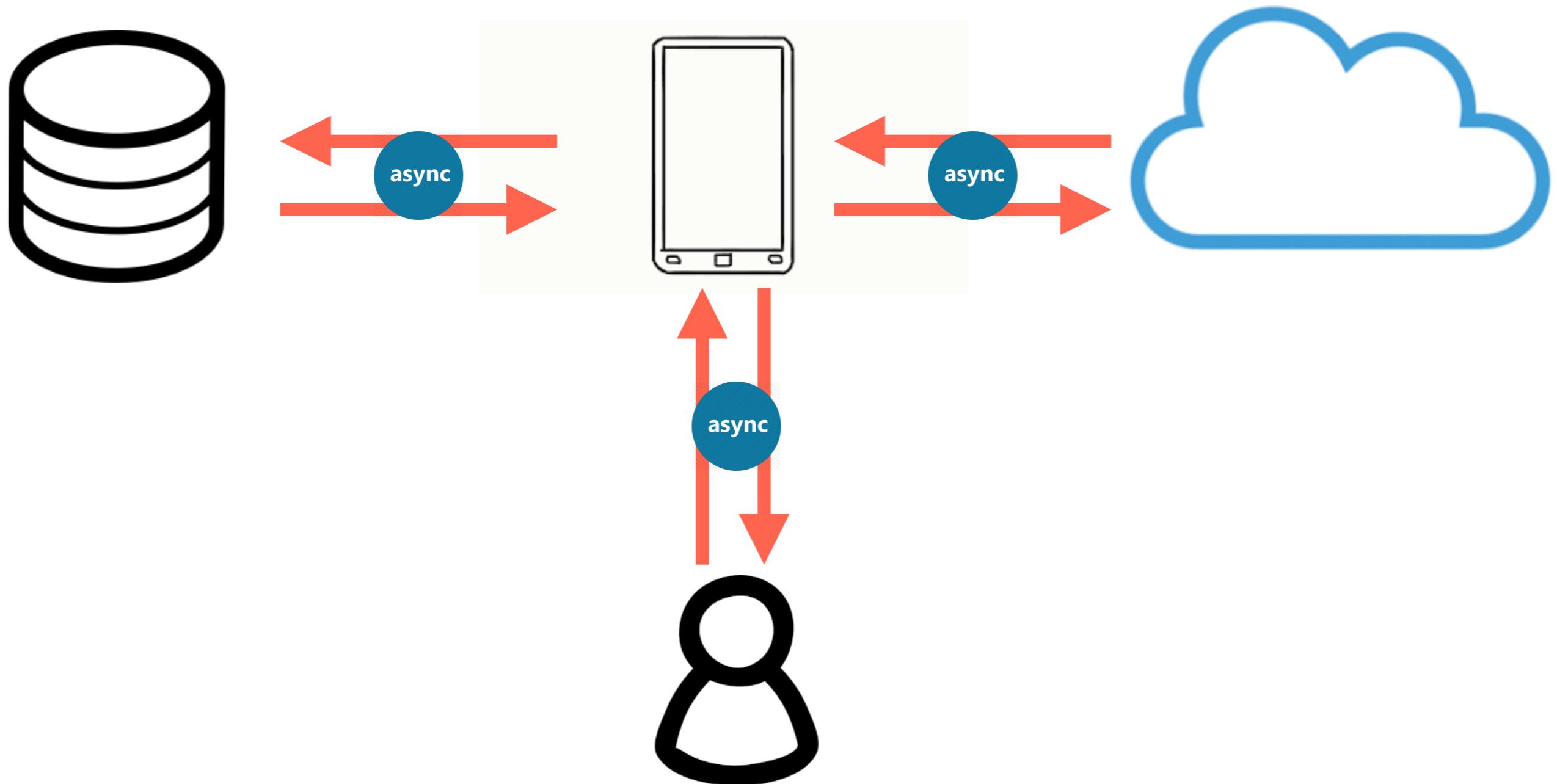
Why Reactive?



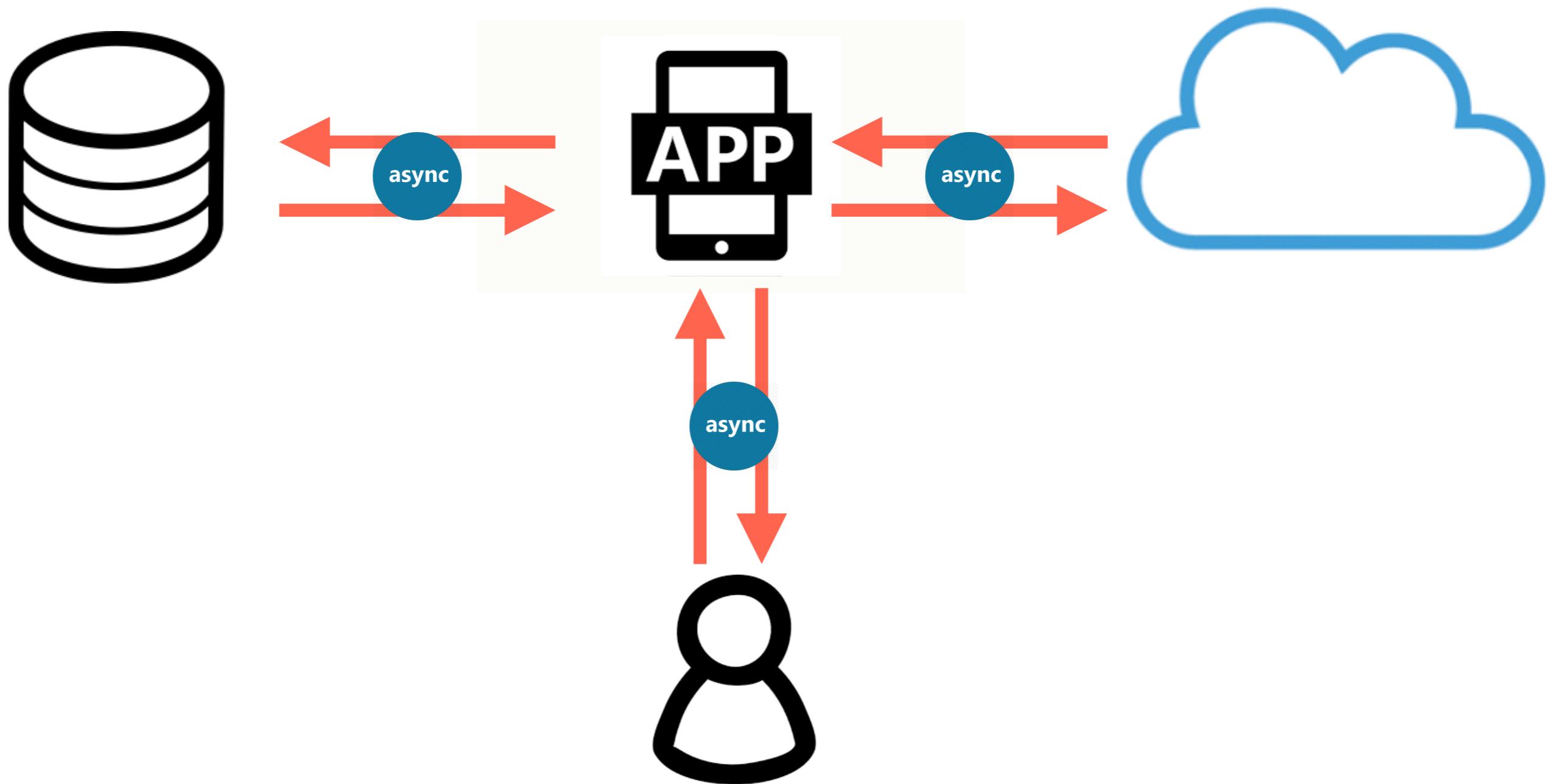
Why Reactive?



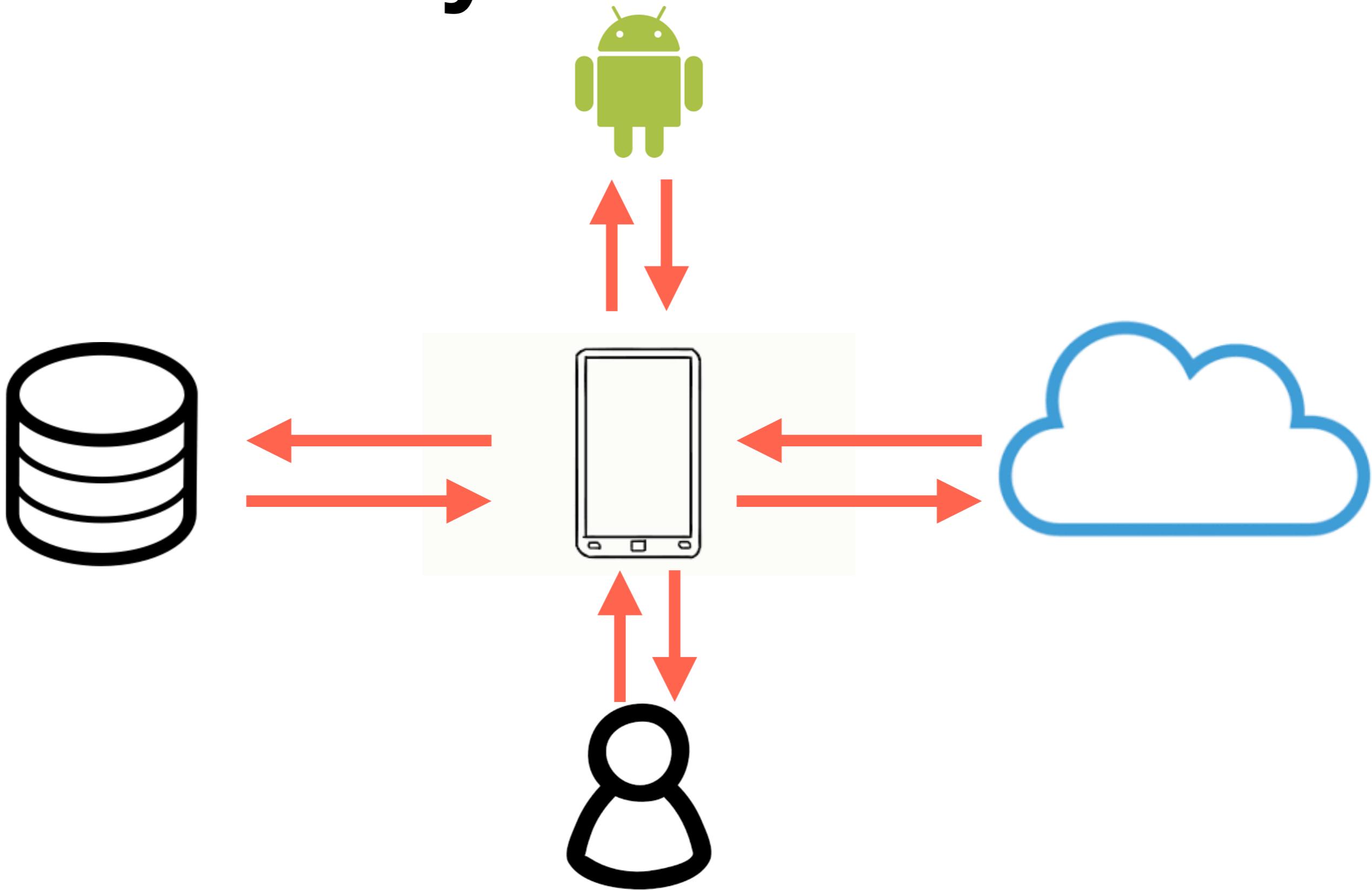
Why Reactive?



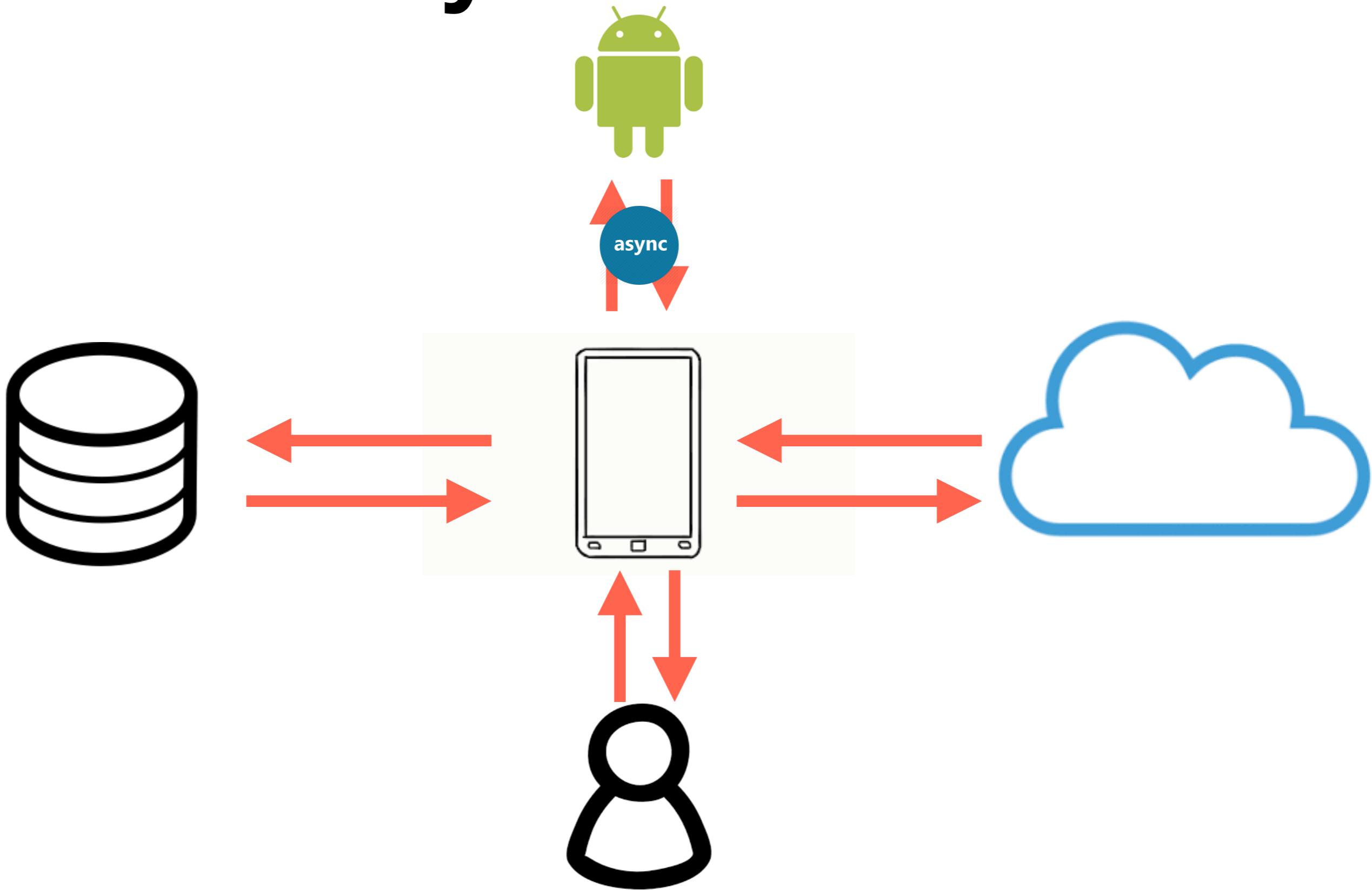
Why Reactive?



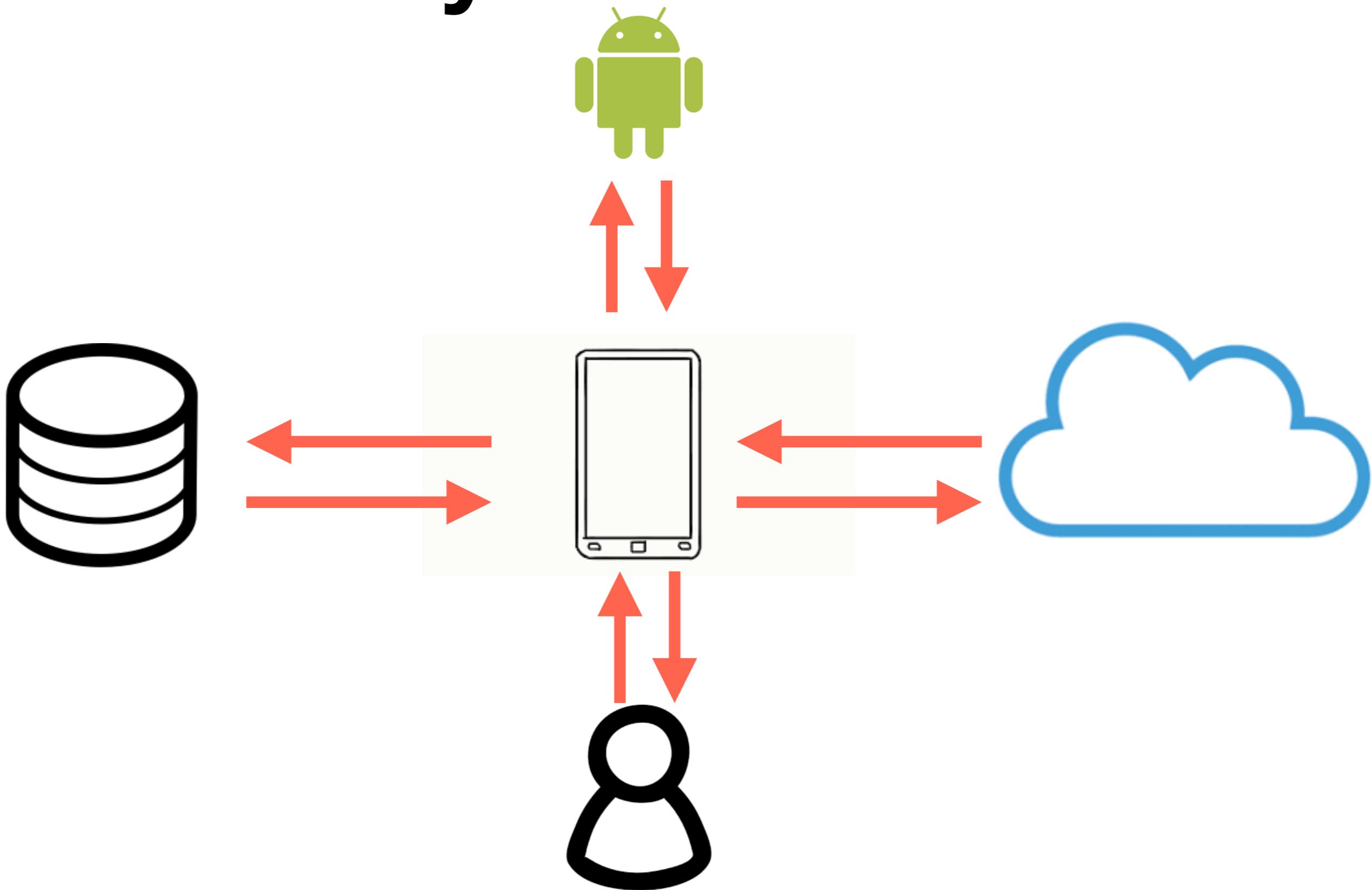
Why Reactive?



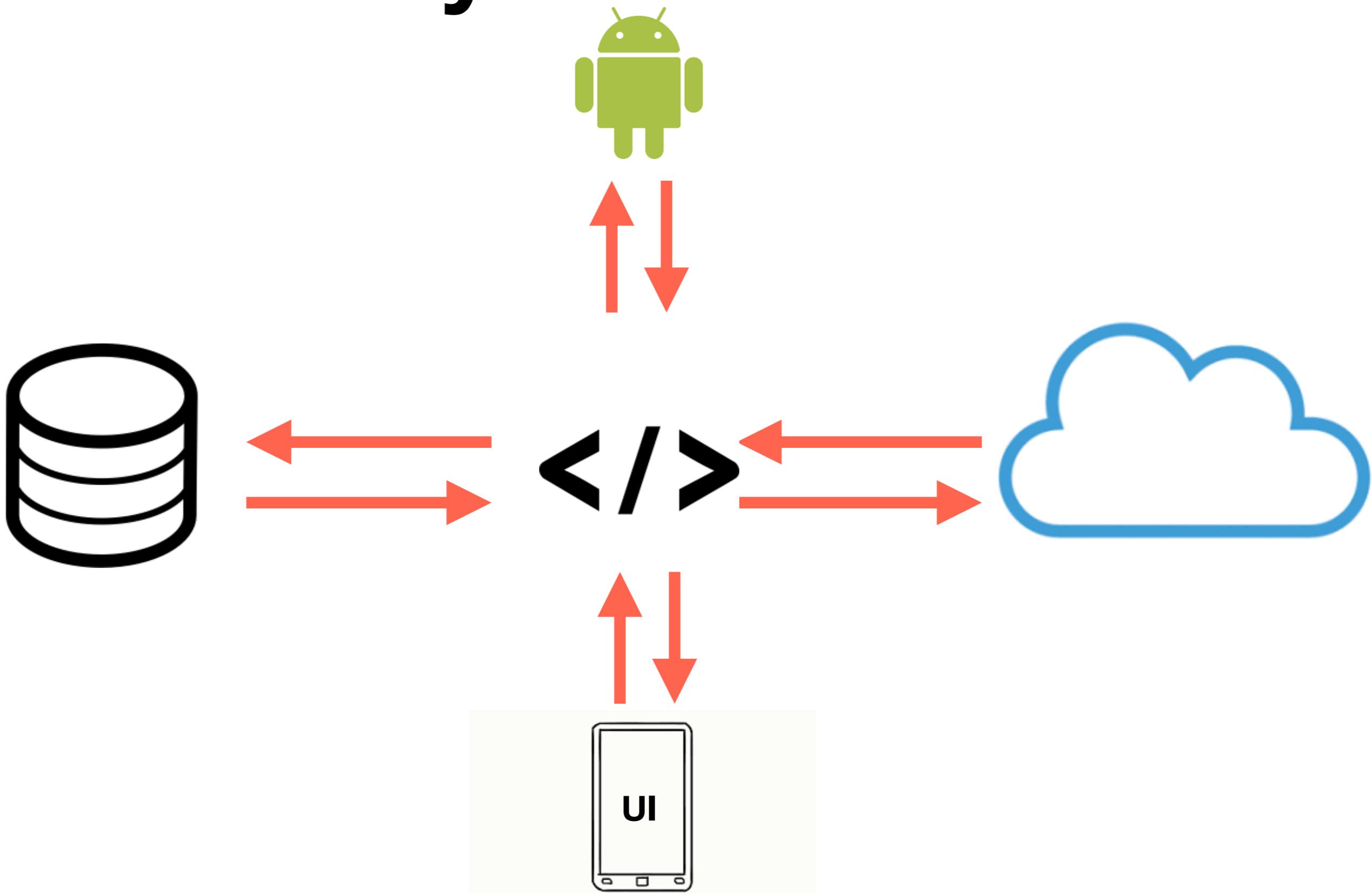
Why Reactive?



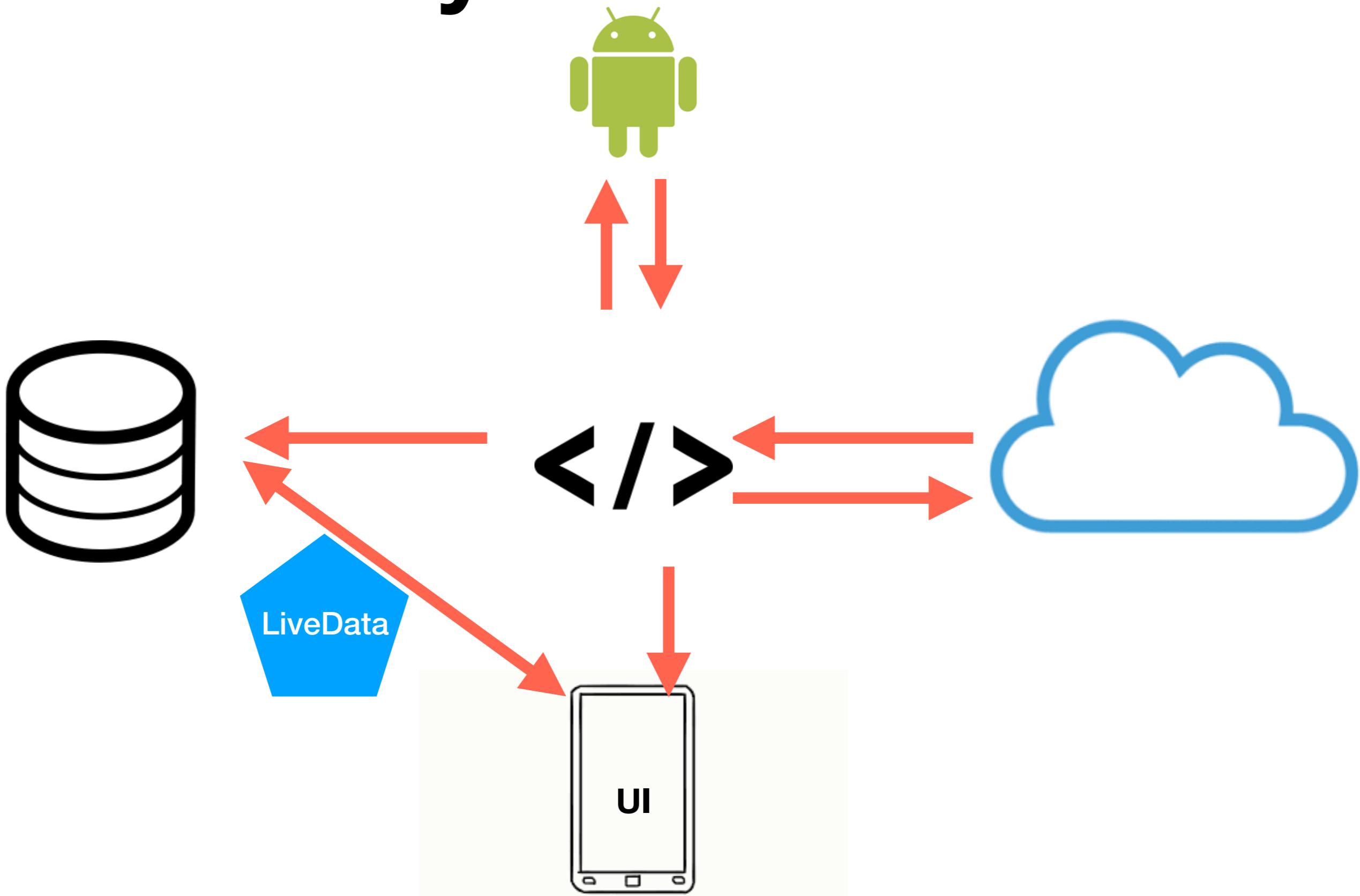
Why Reactive?



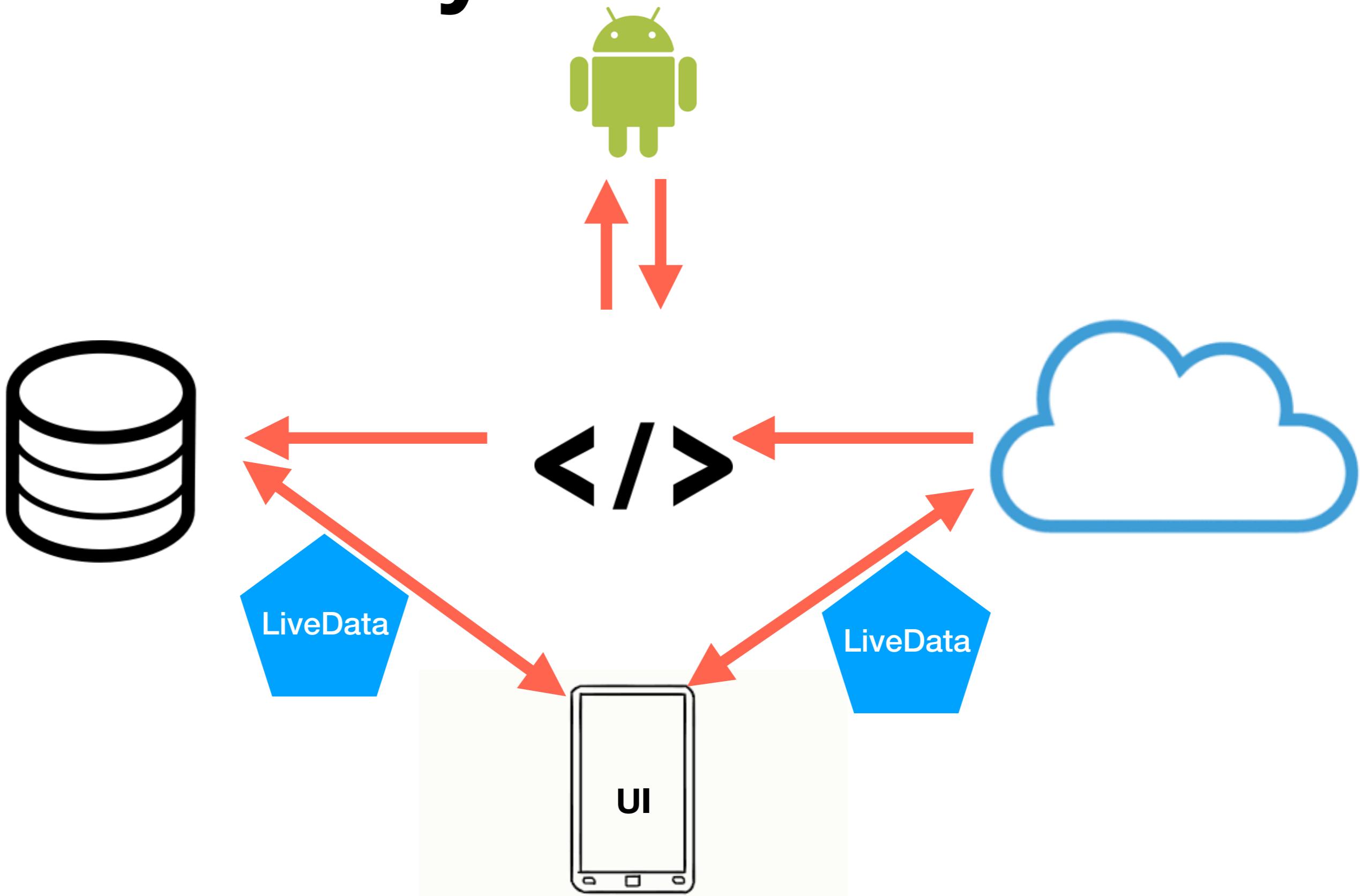
Why Reactive?



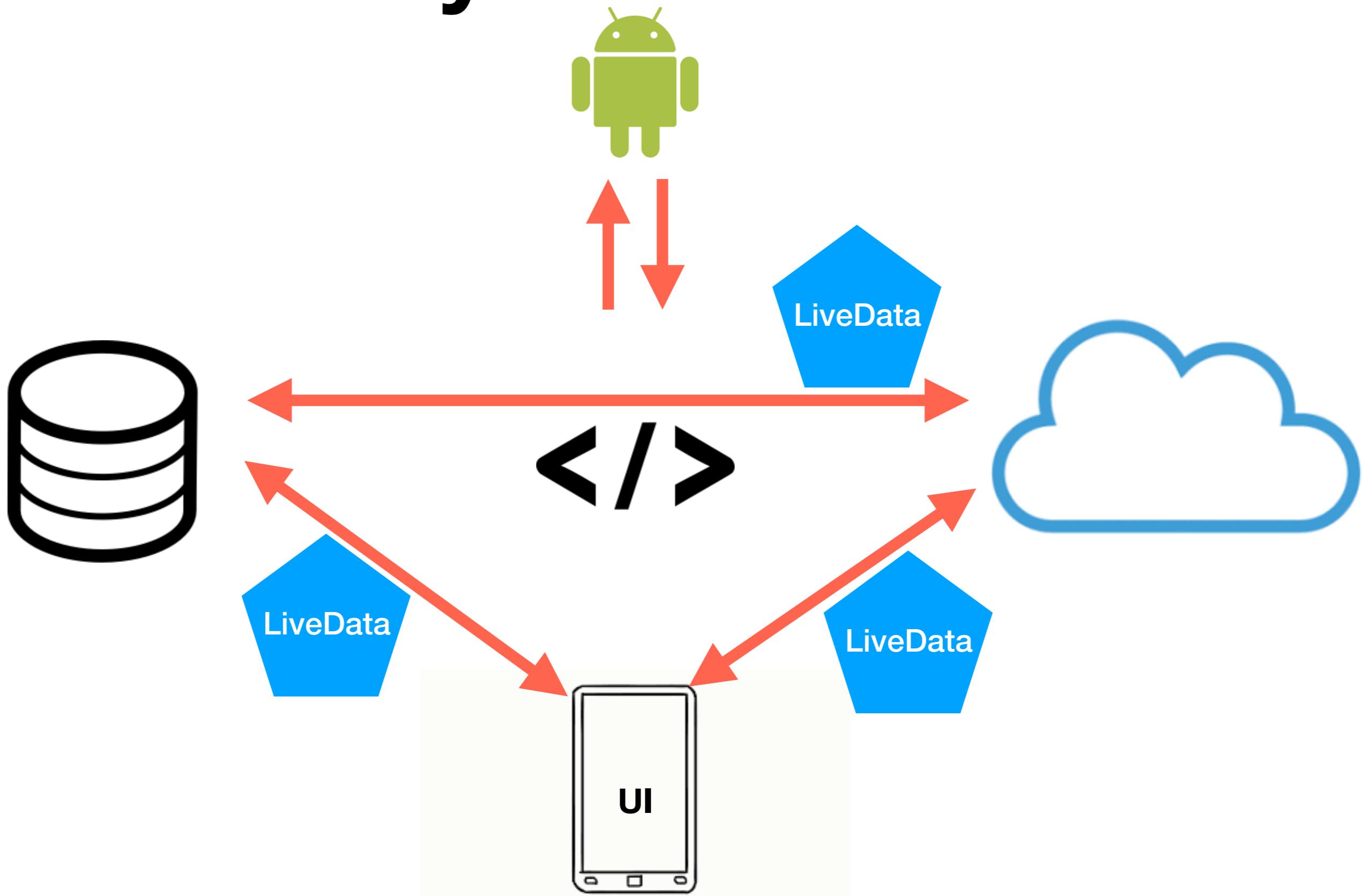
Why Reactive?



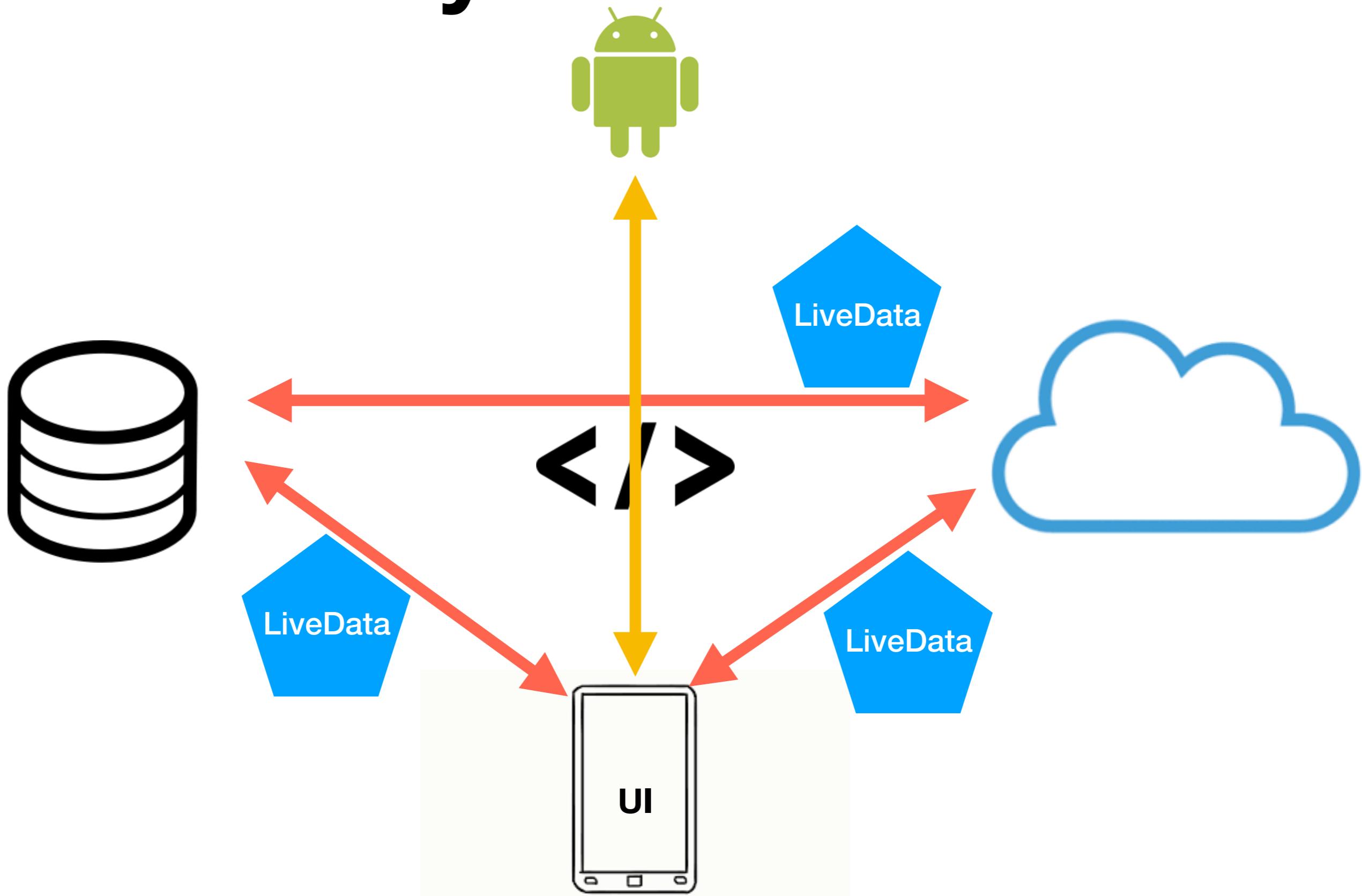
Why Reactive?



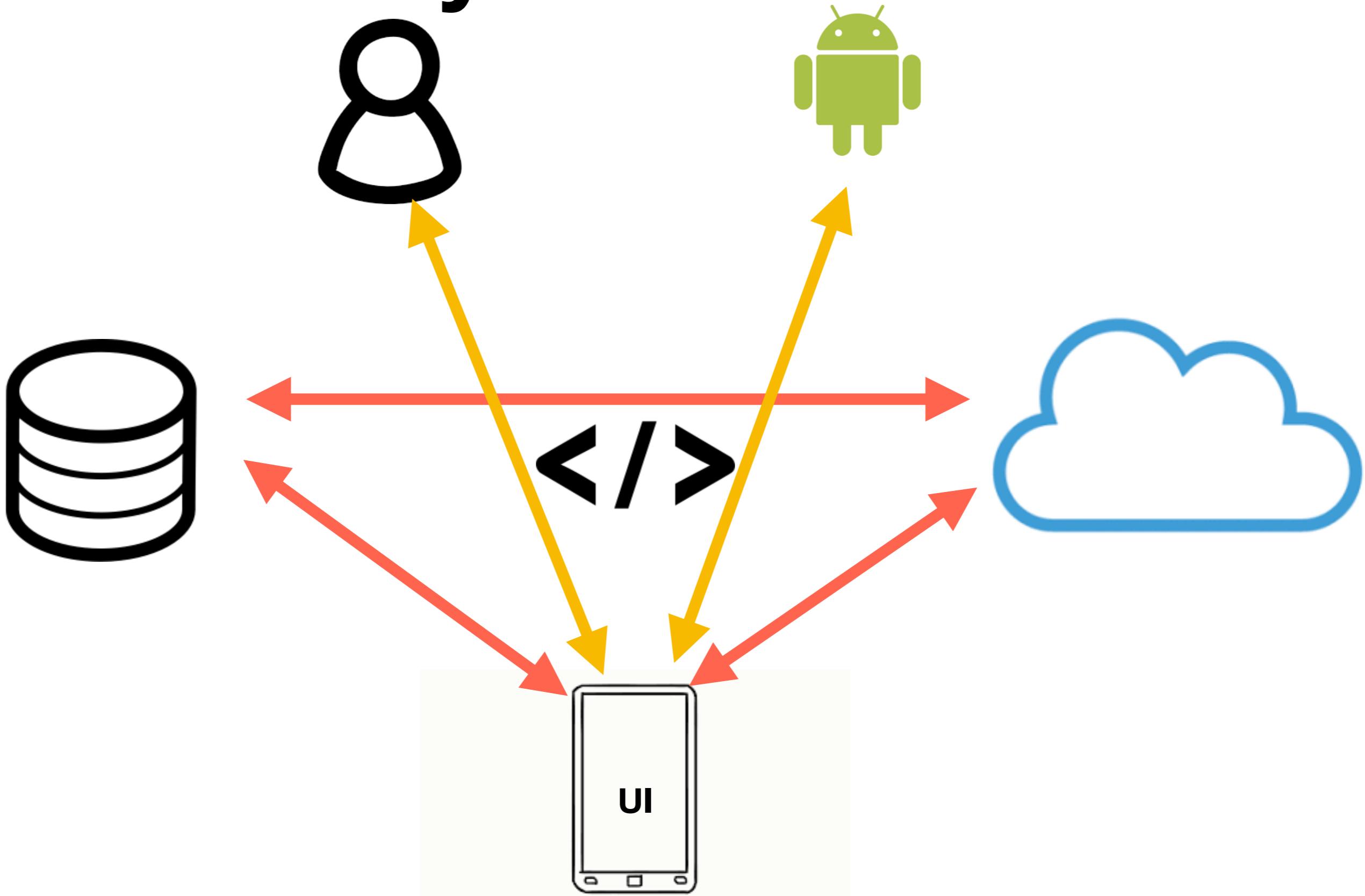
Why Reactive?



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

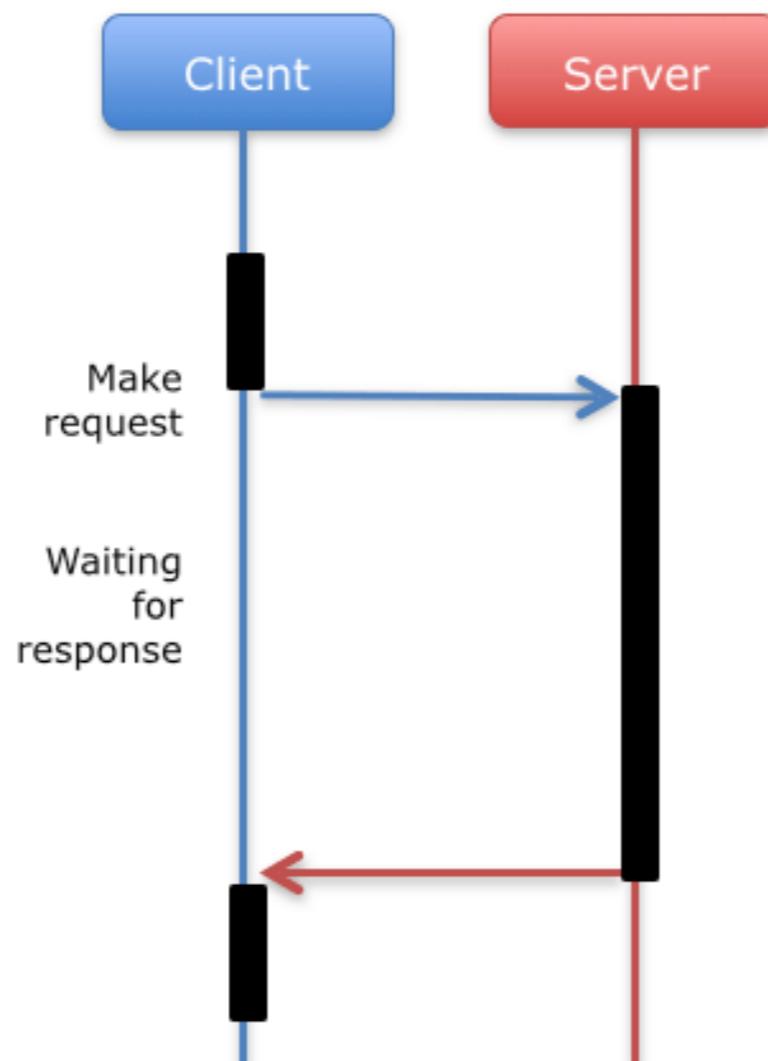
Sources



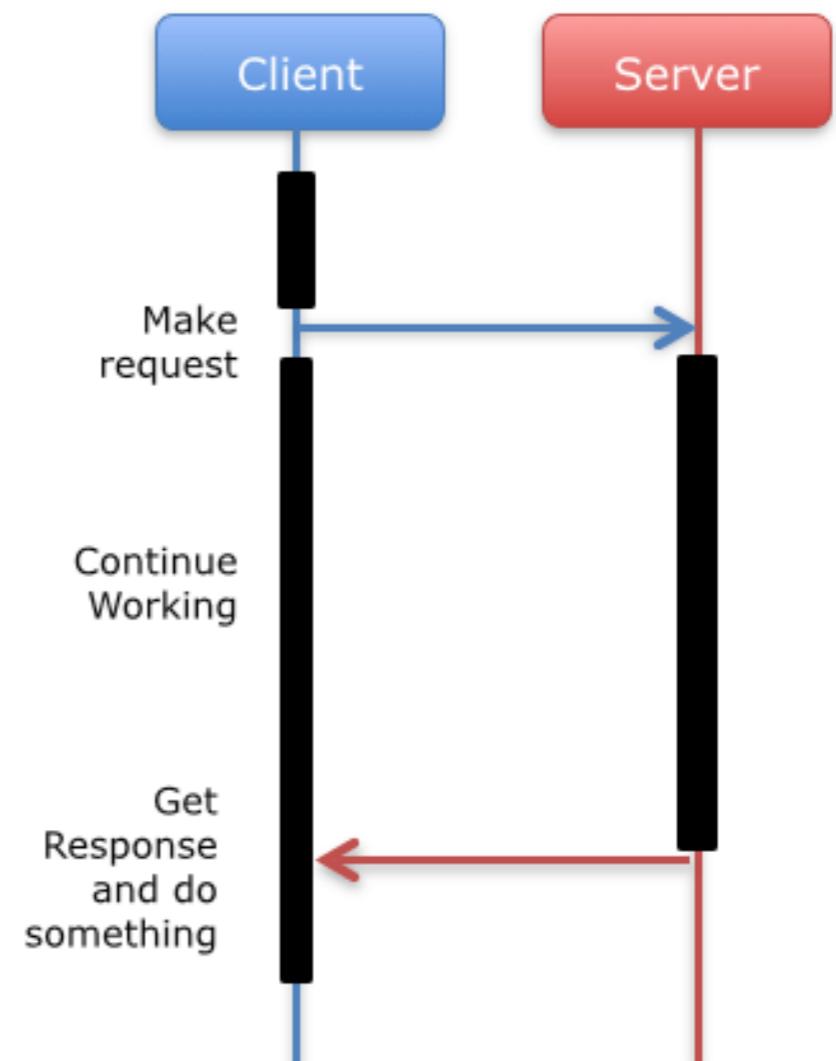
Sources

- Synchronous or asynchronous.

Synchronous

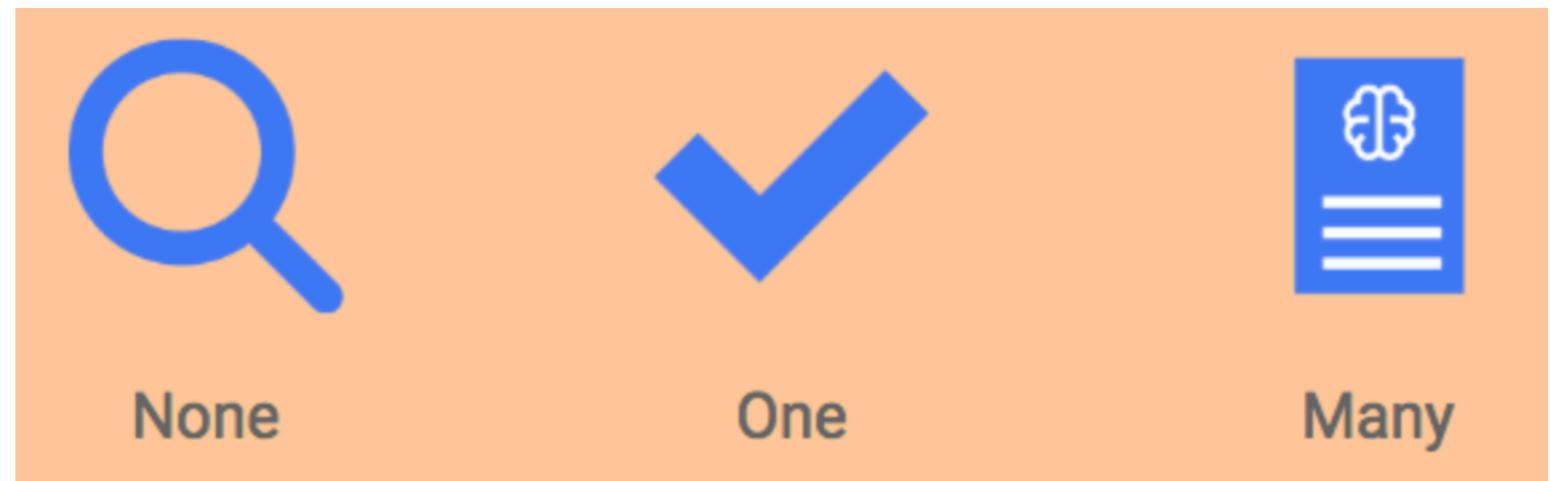


Asynchronous



Sources

- Synchronous or asynchronous.
- Single item, many items, or empty.



Sources

- Synchronous or asynchronous.
- Single item, many items, or empty.
- Terminates with an error or succeeds to completion.



Sources

- Synchronous or asynchronous.
- Single item, many items, or empty.
- Terminates with an error or succeeds to completion.
- May never terminate!



Sources

- Synchronous or asynchronous.
- Single item, many items, or empty.
- Terminates with an error or succeeds to completion.
- May never terminate!
- Just an implementation of the Observer pattern.



Sources

- Observable<T>
- Flowable<T>



Sources

- Observable<T>
 - Emits 0 to N items.
 - Terminates with complete or error.
- Flowable<T>
 - Emits 0 to N items.
 - Terminates with complete or error.



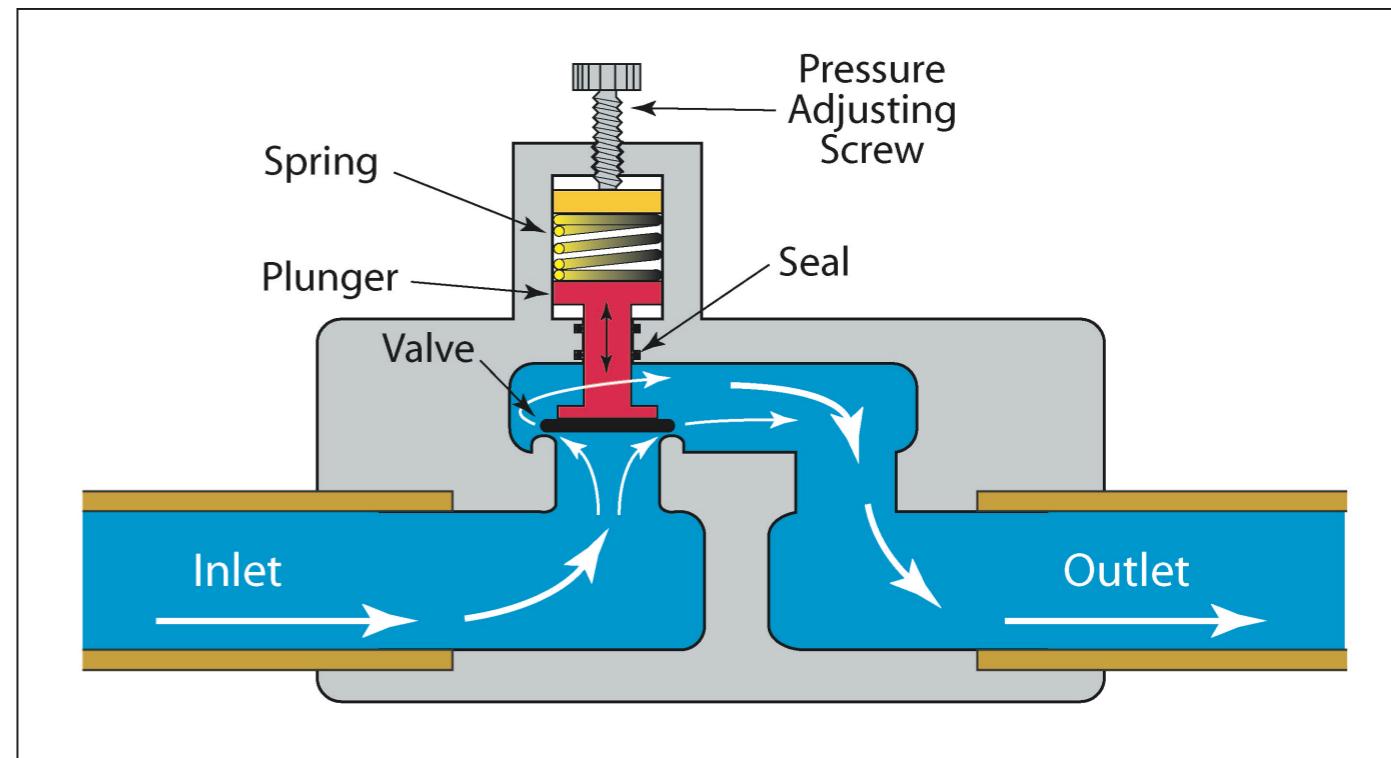
Sources

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



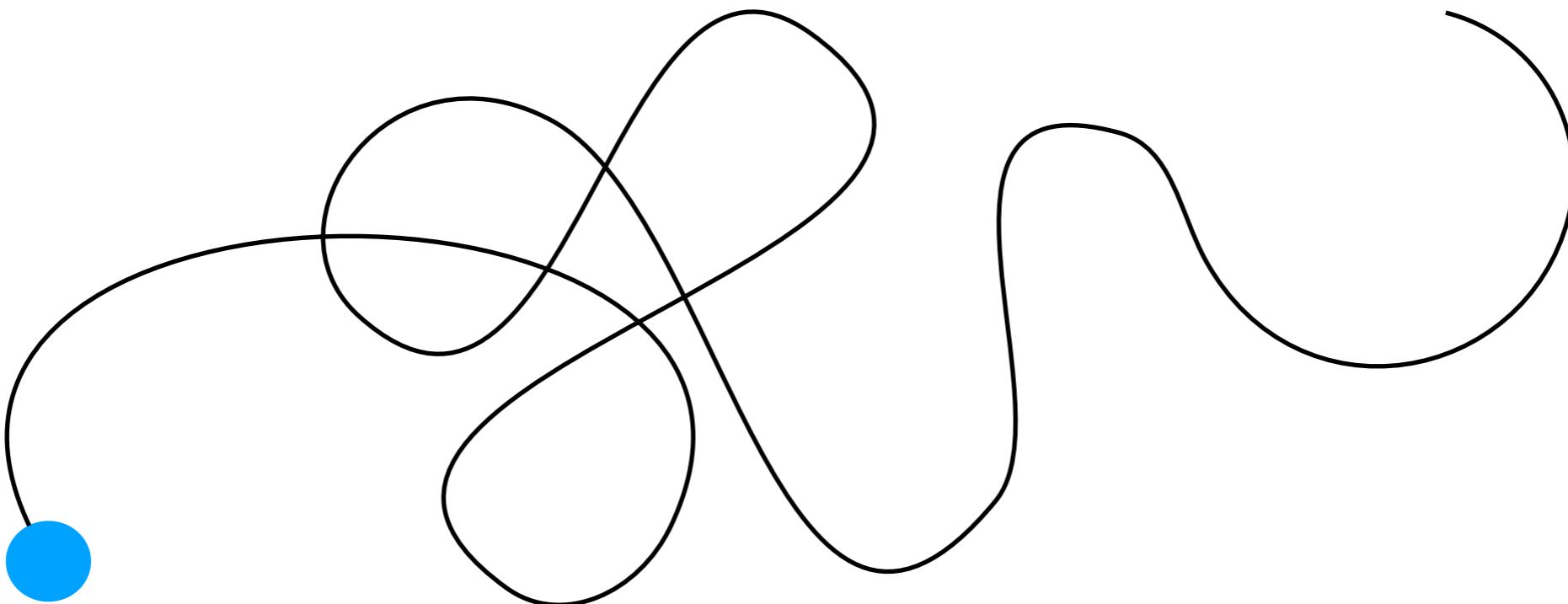
Sources

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



Flowable vs. Observable

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



Flowable vs. Observable

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

```
val users: Observable<User> = db.query("select * from ...");
```

The screenshot shows the Firebase Realtime Database interface. The database path is `brilliant-fire-3159/users/users`. The data structure is as follows:

- `b797bdd4-a193-4029-9e72-a112153686c1`:
 - `-KI2GLhvlWk0jUgx_8_Y`:
 - `fieldName: "messageField"`
 - `text: "this is my first message"`
 - `-KI2GliD5al5axiWmP5G`:
 - `fieldName: "messageField"`
 - `text: "this is my second message"`
 - `-KI2H0sKk5A6Ny1plBly`:
 - `fieldName: "messageField"`
 - `text: "this is my third message"`
 - `+ -KI2H2ro0LBJo-0zdQo3`
 - `+ -KI2q6KfuQFR-Dd577-v`
- `fbf79e3c-7f3c-49d6-bea7-3f940ecfcfe0`

Flowable vs. Observable

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

```
val users: Flowable<User> = db.query("select * from ...");
```

The screenshot shows the Firebase Realtime Database interface. The database path is `brilliant-fire-3159/users/users`. The data structure is as follows:

- `b797bdd4-a193-4029-9e72-a112153686c1`:
 - `-KI2GLhvlWk0jUgx_8_Y`:
 - `fieldName: "messageField"`
 - `text: "this is my first message"`
 - `-KI2GliD5al5axiWmP5G`:
 - `fieldName: "messageField"`
 - `text: "this is my second message"`
 - `-KI2H0sKk5A6Ny1plBly`:
 - `fieldName: "messageField"`
 - `text: "this is my third message"`
 - `+ -KI2H2ro0LBJo-0zdQo3`
 - `+ -KI2q6KfuQFR-Dd577-v`
- `fbf79e3c-7f3c-49d6-bea7-3f940ecfcfe0`

Flowable vs. Observable

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)
}
```

Flowable vs. Observable

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)
}
```

Flowable vs. Observable

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)
}
```

Flowable vs. Observable

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)
}
```

Flowable vs. Observable

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)
}
```

Sources

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

Sources

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

Specialized Sources

Specialized Sources

- Encoding subsets of Observable into the type system:

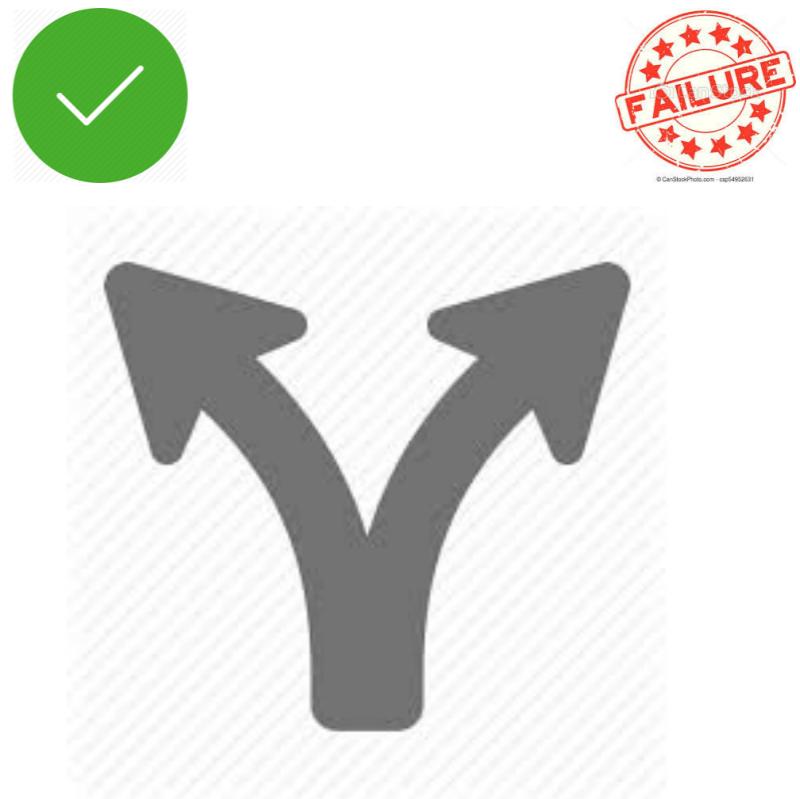
Specialized Sources

- Encoding subsets of Observable into the type system:
 - Single
 - Either succeeds with an item or an error.
 - No backpressure support.



Specialized Sources

- Encoding subsets of Observable into the type system:
 - Single
 - Either succeeds with an item or an error.
 - No backpressure support.
 - Completable
 - Either completes or errors. Has no items!
 - No backpressure support.



Specialized Sources

- Encoding subsets of Observable into the type system:
 - Single
 - Either succeeds with an item or an error.
 - No backpressure support.
 - Completable
 - Either completes or errors. Has no items!
 - No backpressure support.
 - Maybe
 - Either succeeds with an item, completes with no items, or error.
 - No backpressure support.



Specialized Sources

- Encoding subsets of Observable into the type system:

- Single



- Either succeeds with an item or an error.

- No backpressure support.

- Completable

- Either completes or errors. Has no items!

- No backpressure support.

- Maybe

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

- No backpressure support.



Specialized Sources

- Encoding subsets of Observable into the type system:

- Single



- Either succeeds with an item or an error.

- No backpressure support.

- Completable



- Either completes or errors. Has no items!

- No backpressure support.

- Maybe

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

- No backpressure support.



Specialized Sources

- Encoding subsets of Observable into the type system:

- Single



- Either succeeds with an item or an error.

- No backpressure support.

- Completable



- Either completes or errors. Has no items!

- No backpressure support.

- Maybe



- 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)  
    fun setAge(age: Int)  
}
```

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")
```

```
Single.just("Hello")
```

Creating Sources

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

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

```
Maybe.just("Hello")
```

```
Single.just("Hello")
```

```
val array = arrayListOf("Hello", "World")
val list = array.toList()
```

```
Flowable.fromArray(array)
Flowable.fromIterable(list)
```

```
Observable.fromArray(array)
Observable.fromIterable(list)
```

Creating Sources

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

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

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

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

```
Maybe.just("Hello")
```

```
Single.just("Hello")
```

```
val array = arrayListOf("Hello", "World")
val list = array.toList()
```

```
Flowable.fromArray(array)
```

```
Flowable.fromIterable(list)
```

```
Observable.fromArray(array)
```

```
Observable.fromIterable(list)
```

```
Observable.fromCallable {
    setName()
}
```

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

Create Sources

```
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

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

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

Create Sources

```
Observable.create<View> {
    it.setCancellable { textView.setOnClickListener(null) }
    textView.setOnClickListener { v -> it.onNext(v) }
}
```

Create Sources

```
Observable.create<View> {
    it.setCancellable { textView.setOnClickListener(null) }
    textView.setOnClickListener { v -> it.onNext(v) }
}

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

Observable.create<String> {
    val call = client.newCall(request)
    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

Observing Sources

```
val observable: Observable<String> = Observable.just("Hello")

observable.subscribe(object: DisposableObserver<String>(){
    override fun onComplete() {
        //...
    }

    override fun onNext(t: String) {
        //...
    }

    override fun onError(e: Throwable) {
        //...
    }
})
```

Observing Sources

```
val observable: Observable<String> = Observable.just("Hello")

observable.subscribe(object: DisposableObserver<String>(){
    override fun onComplete() {
        //...
    }

    override fun onNext(t: String) {
        //...
    }

    override fun onError(e: Throwable) {
        //...
    }
})
```

How to we dispose?

Observing Sources

```
val observable: Observable<String> = Observable.just("Hello")

val observer = object : DisposableObserver<String>() {
    override fun onComplete() {
        //...
    }

    override fun onNext(t: String) {
        //...
    }

    override fun onError(e: Throwable) {
        //...
    }
}
observable.subscribe(observer)

observer.dispose()
```

Observing Sources

```
val observable: Observable<String> = Observable.just("Hello")

val disposable = observable.subscribeWith(object : DisposableObserver<String>() {
    override fun onComplete() {
        //...
    }

    override fun onNext(t: String) {
        //...
    }

    override fun onError(e: Throwable) {
        //...
    }
})

disposable.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.



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.**



Operators



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

Operators

```
val greeting = "Hello"  
val yelling = greeting.toUpperCase()
```

Operators

```
val greeting = Observable.just("Hello")
val yelling = greeting.toUpperCase()
```

Operators

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

Operators

```
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 (!isDestroyed) {  
                                textView.text = user.name  
                            }  
                        }  
                    }  
                }  
            }  
            override fun failed(error: UserException) {  
                Log.e("Unable to update the user details", error)  
            }  
        })  
    }  
    //...  
}
```

Operators

```
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 (!isDestroyed) {  
                                textView.text = user.name  
                            }  
                        }  
                    }  
                }  
            }  
            override fun failed(error: UserException) {  
                loge("Unable to update the user details", error)  
            }  
        })  
        //...  
    }  
}
```

Operators

```
val user: Observable<User> = um.getUser()  
val mainThreadUser = user.observeOn(AndroidSchedulers.mainThread())
```

Operators

```
val user: Observable<User> = um.getUser()
val mainThreadUser = user.observeOn(AndroidSchedulers.mainThread())

val url = "https://example.com"

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

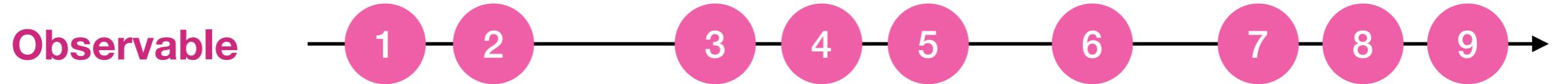
Observable



Observable

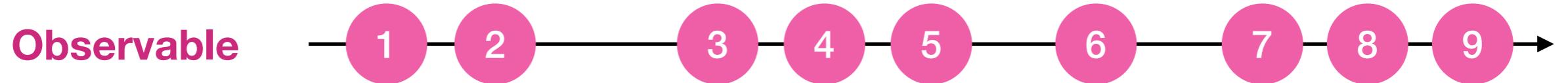


Operators

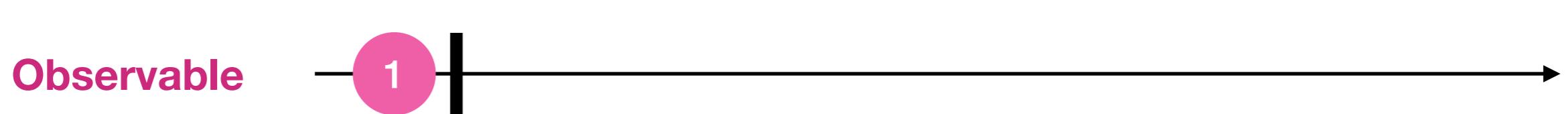


Observable —————→

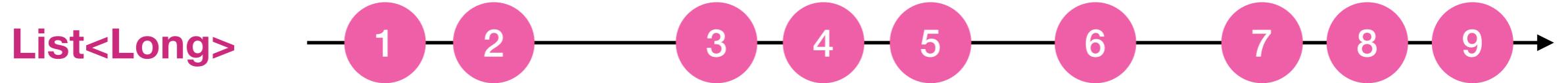
Operators



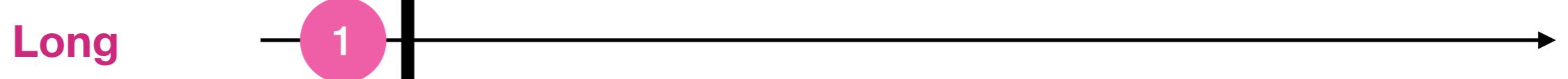
first()



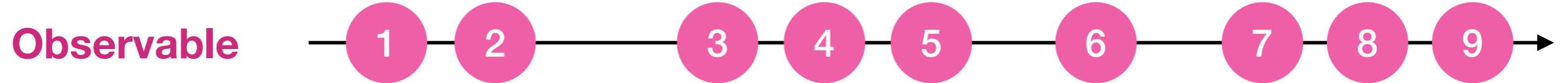
Operators



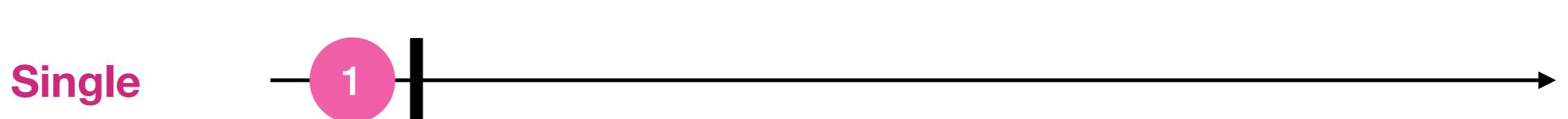
get(0)



Operators

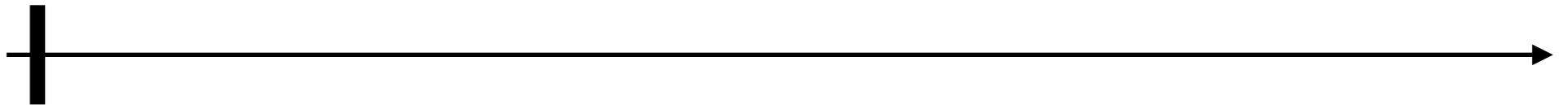


first()



Operators

Observable



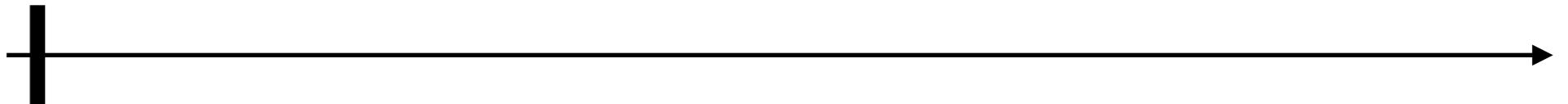
first()

Single



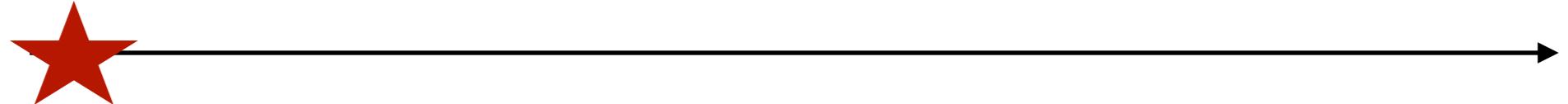
Operators

Observable



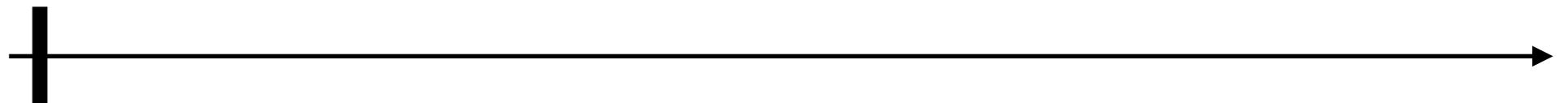
`first()`

Single



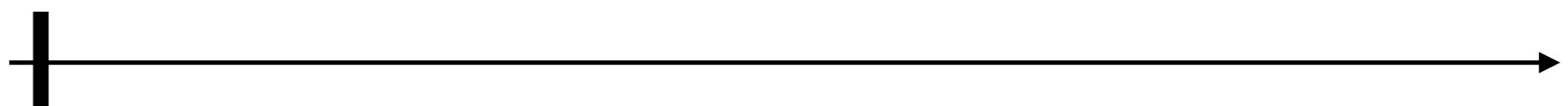
Operators

Observable

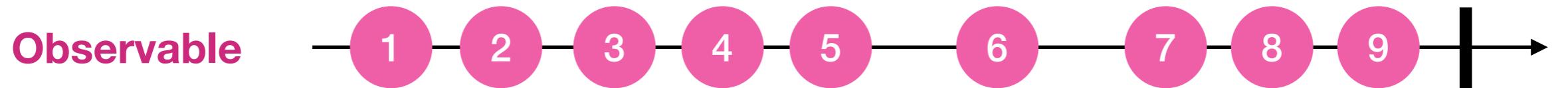


`firstElement()`

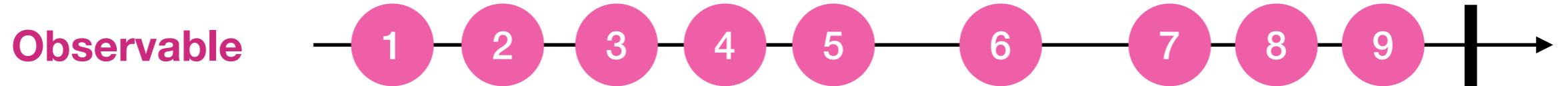
Maybe



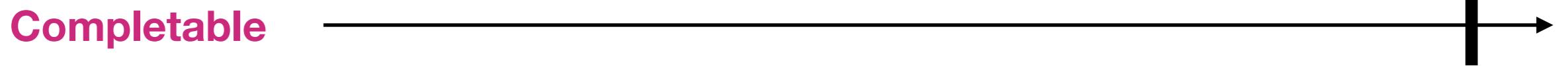
Operators



Operators



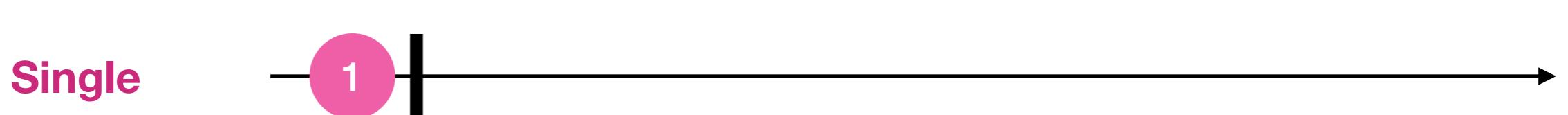
ignoreElements()



Operators

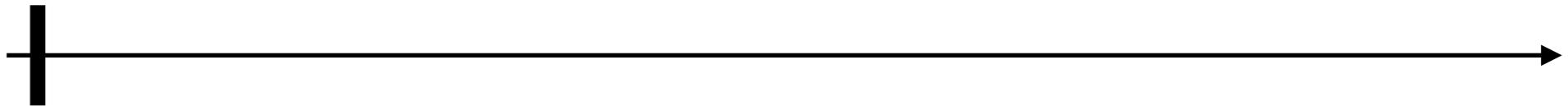


first()



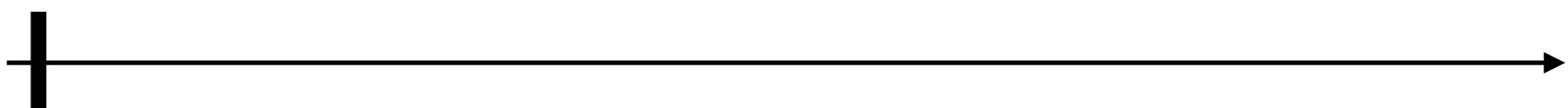
Operators

Flowable



`firstElement()`

Maybe



Operators



`ignoreElements()`



Operators



ignoreElements()



Being Reactive

```
um.getUser()
    .observeOn(AndroidSchedulers.mainThread())
    .subscribeWith(object: DisposableObserver<User>(){
        override fun onNext(user: User) {
        }

        override fun onError(e: Throwable) {
        }

        override fun onComplete() {
        }
    })
}
```

Being Reactive

```
um.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() {
        }
    })
}
```

Being Reactive

```
// onCreate
val disposables = CompositeDisposable()
disposables.add(um.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

Options

- Callbacks

```
describe('.totalValue', function(){
  it('should calculate the total value of items in a space', function(done){
    var table = new Item('table', 'dining room', '07/23/2014', '1', '3000');
    var chair = new Item('chair', 'living room', '07/23/2014', '3', '300');
    var couch = new Item('couch', 'living room', '07/23/2014', '2', '1100');
    var chair2 = new Item('chair', 'dining room', '07/23/2014', '4', '500');
    var bed = new Item('bed', 'bed room', '07/23/2014', '1', '2000');

    table.save(function(){
      chair.save(function(){
        couch.save(function(){
          chair2.save(function(){
            bed.save(function(){
              Item.totalValue({room: 'dining room'}, function(totalValue){
                expect(totalValue).to.equal(5000);
                done();
              });
            });
          });
        });
      });
    });
  });
});
```

Options

What's the Future

- Callbacks
- Futures

- Callable – Runnable on steroids

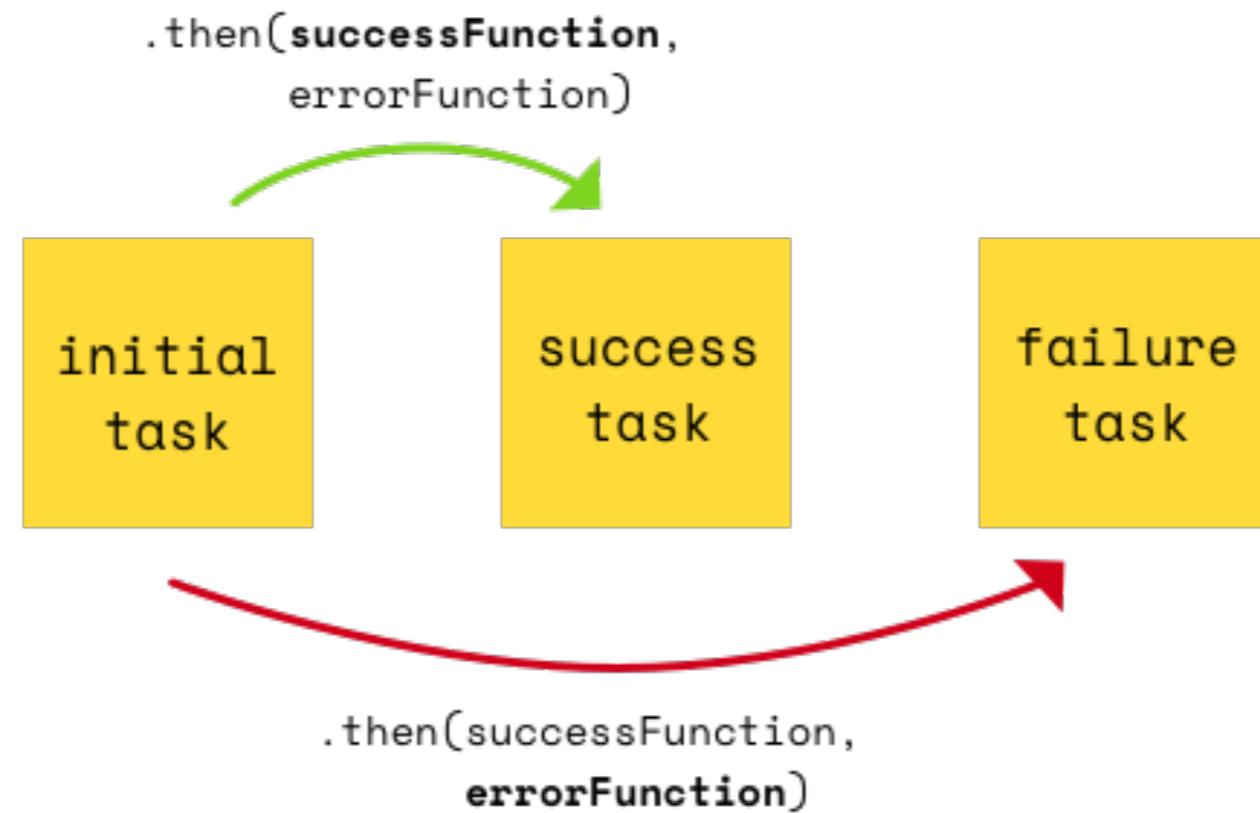
```
Callable<V> {  
    V call() throws Exception;  
}
```

- Future – result of an asynchronous computation

```
Future<V> {  
    V get();  
    boolean cancel();  
    boolean isCancelled();  
    boolean isDone();  
}
```

Options

- Callbacks
- Futures
- Promises



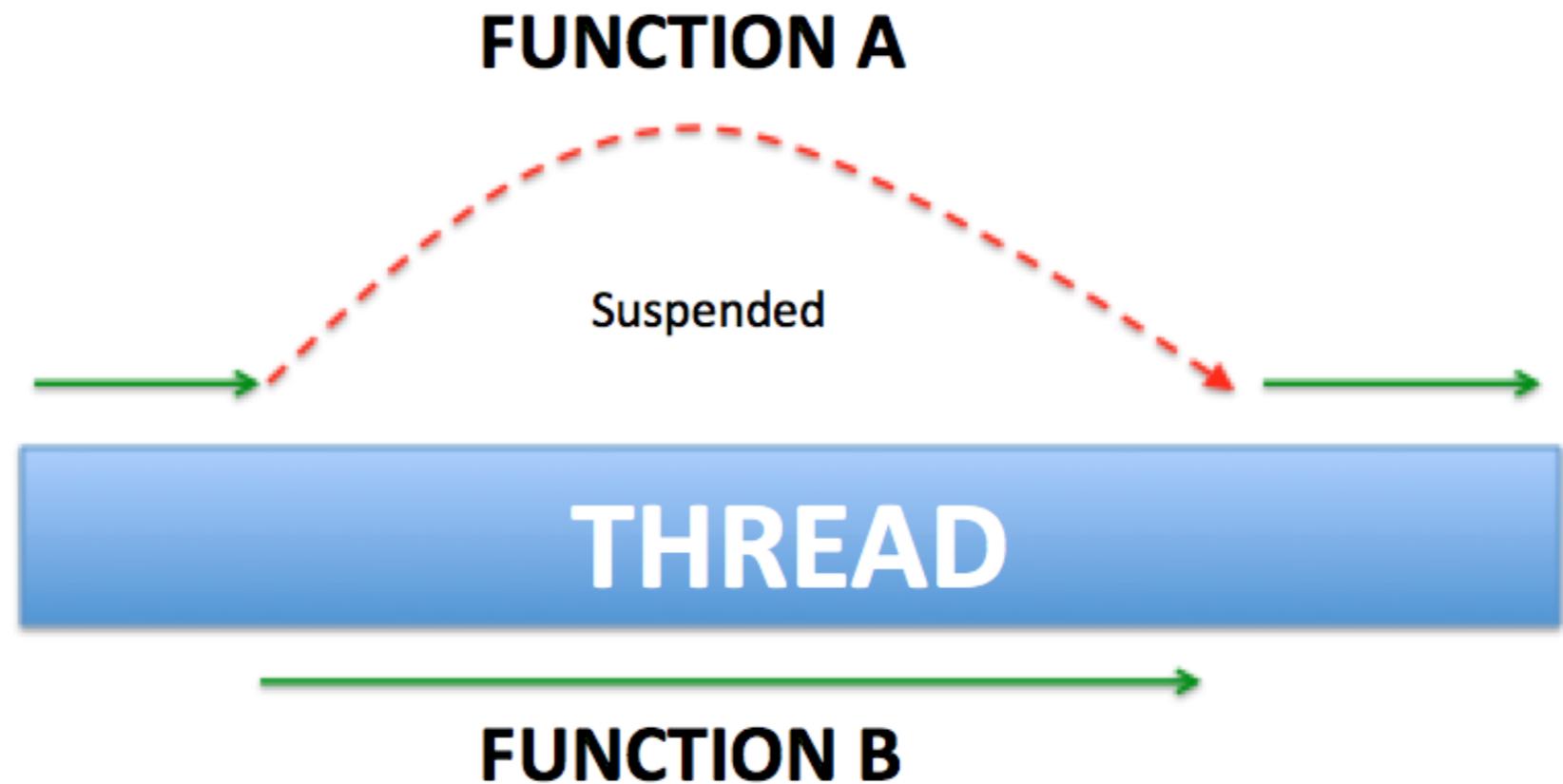
Options

- Callbacks
- Futures
- Promises
- Rx



Options

- Callbacks
- Futures
- Promises
- Rx
- Coroutines



Coroutines

```
fun requestToken(): Token {  
    // make a token request and waits  
    return token  
}  
  
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")  
}  
  
fun postItem(item: Item) {  
    val token = requestToken()  
    val post = createPost(token, item)  
    processPost(post)  
}
```

Coroutines

```
fun requestToken(): Token {  
    // make a token request  
    // block the thread while waiting for result  
    return token  
}  
  
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")  
}  
  
fun postItem(item: Item) {  
    val token = requestToken()  
    val post = createPost(token, item)  
    processPost(post)  
}
```

Coroutines



```
fun requestToken(): Token {  
    // make a token request  
    // block the thread while waiting for result  
    return token  
}  
  
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")  
}  
  
fun postItem(item: Item) {  
    val token = requestToken()  
    val post = createPost(token, item)  
    processPost(post)  
}
```

Coroutines

```
suspend fun requestToken(): Token {  
    // make a token request and suspends  
    return token  
}  
  
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")  
}  
  
fun postItem(item: Item) {  
    val token = requestToken()  
    val post = 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 {  
    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")  
}  
  
fun postItem(item: Item) {  
    val token = requestToken()  
    val post = 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 {  
    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 postItem(item: Item) {  
    val token = requestToken()  
    val post = createPost(token, item)  
    processPost(post)  
}
```



Bonuses

Regular loops:

```
for ((token, item) in list) {  
    createPost(token, item)  
}
```

Bonuses

Regular loops:

```
 for ((token, item) in list) {  
    createPost(token, item)  
}
```

Bonuses

Regular loops:

```
for ((token, item) in list) {  
    -$→ createPost(token, item)  
}
```

Regular exception handling:

```
try {  
    -$→ createPost(token, item)  
} catch (e: BadTokenException) {  
    // ...  
}
```

Bonuses

Regular loops:

```
for ((token, item) in list) {  
    -$→ createPost(token, item)  
}
```

Regular exception handling:

```
try {  
    -$→ createPost(token, item)  
} catch (e: BadTokenException) {  
    // ...  
}
```

Regular higher-order function:

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

Bonuses

Regular loops:

```
for ((token, item) in list) {  
    createPost(token, item)  
}
```

Regular exception handling:

```
try {  
    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 postItem(item: Item) {  
    val token = requestToken()  
    val post = createPost(token, item)  
    processPost(post)  
}
```



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")  
}  
  
fun postItem(item: Item) {  
    val token = requestToken()  
    val post = createPost(token, item)  
    processPost(post)  
}
```

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")  
}  
  
fun postItem(item: Item) {  
    val token = requestToken()  
    createPost(token, item)  
}
```

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

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



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



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:1.0.1'  
    implementation "org.jetbrains.kotlinx:kotlinx-coroutines-android:1.0.1"  
}
```

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.

