twitch.tv/dancojocar
youtube.com/dancojocar

Lecture #5
Coroutines & Reactive Programming

Mobile Applications
Fall 2024

Why Reactive?

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

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())
```

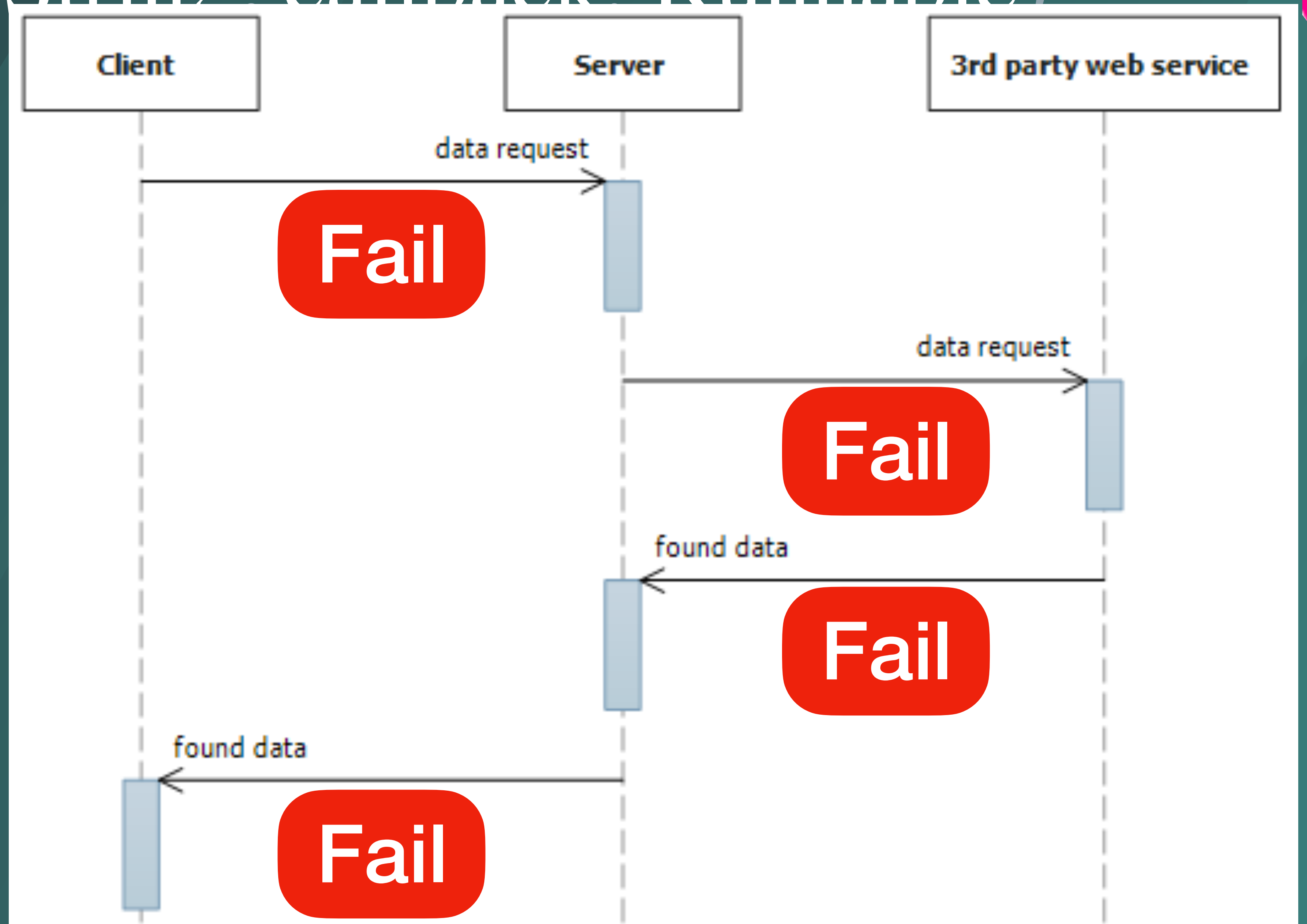
Why Reactive?

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

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

Why Reactive?

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

```
val um: UserManagerV2 = UserManagerV2Impl()
log(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)
  }
}
```

```kotlin
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)
    }
})
```

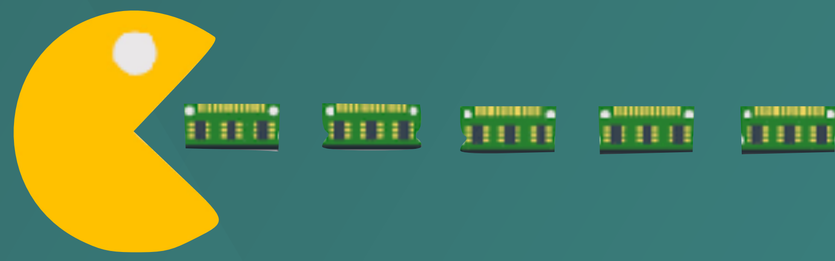
```kotlin
val um: UserManagerV3 = UserManagerV3Impl()
logd(um.getUser())

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)
    }
})
```

```
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)
    }
})

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)
            }
        })
    }
```

```
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?

```
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)
            }
        })
    }
}
```



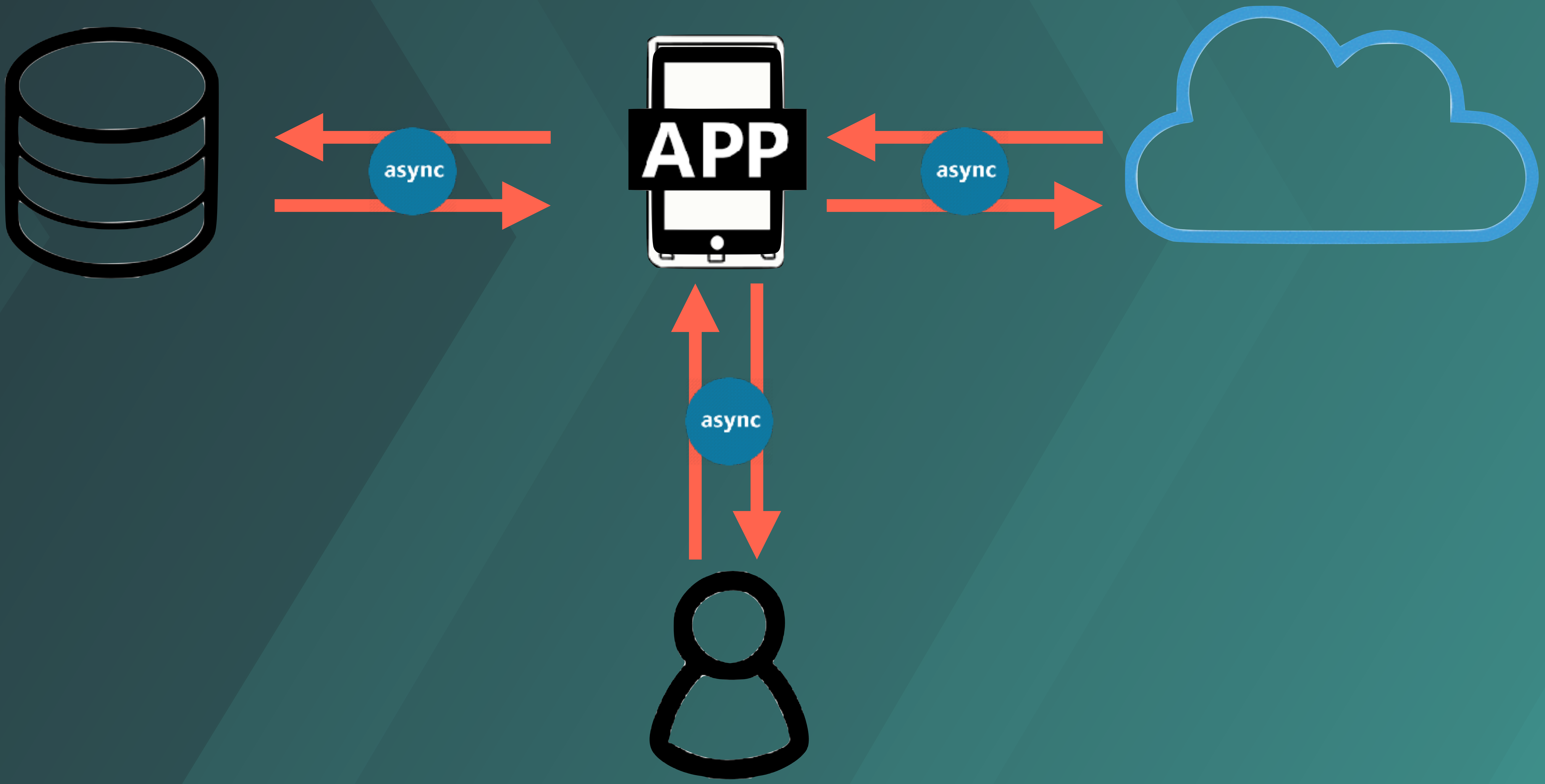
"A small leak will sink a great ship."
Benjamin Franklin

https://github.com/square/leakcanary

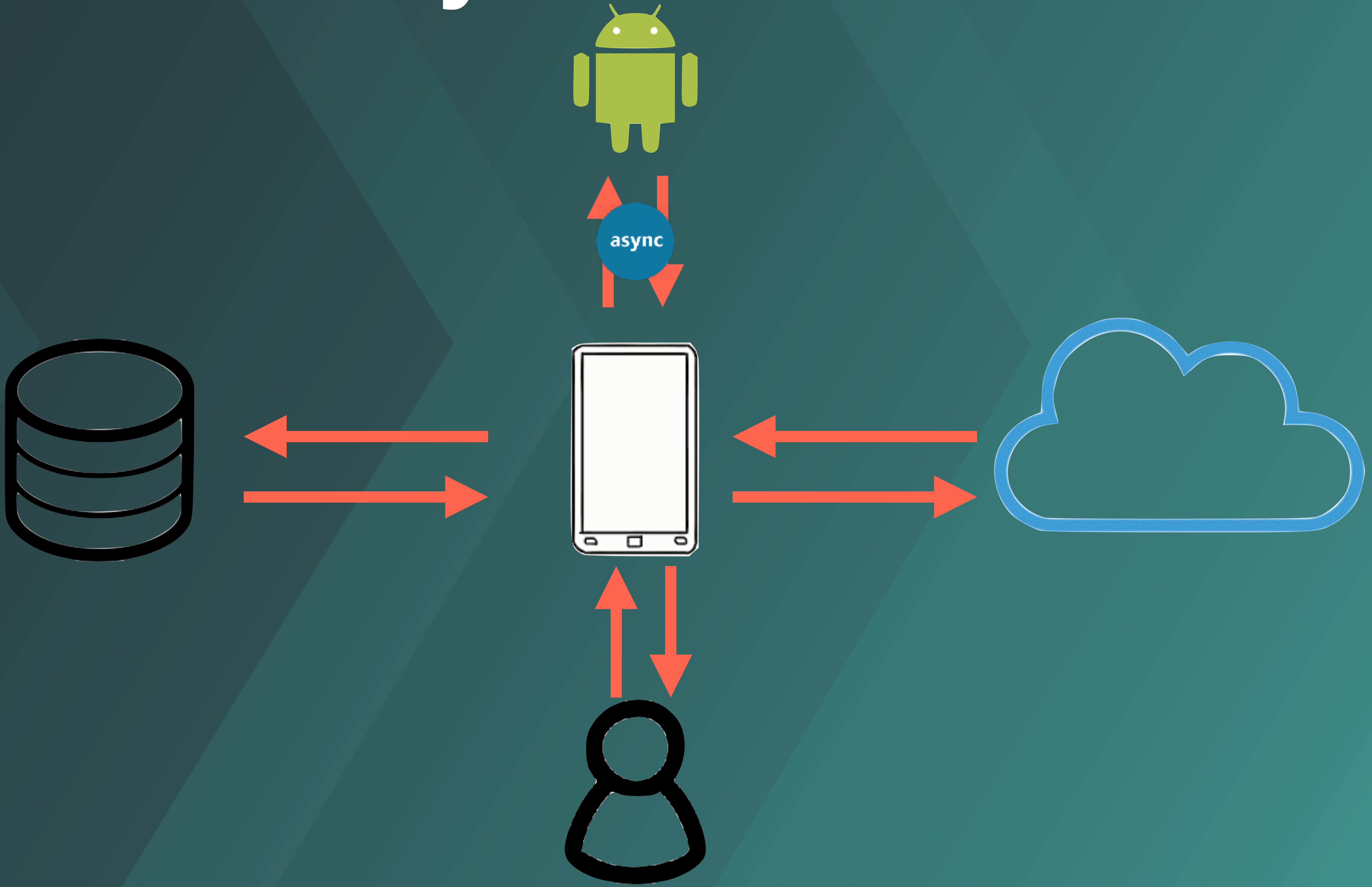
Why Reactive?

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

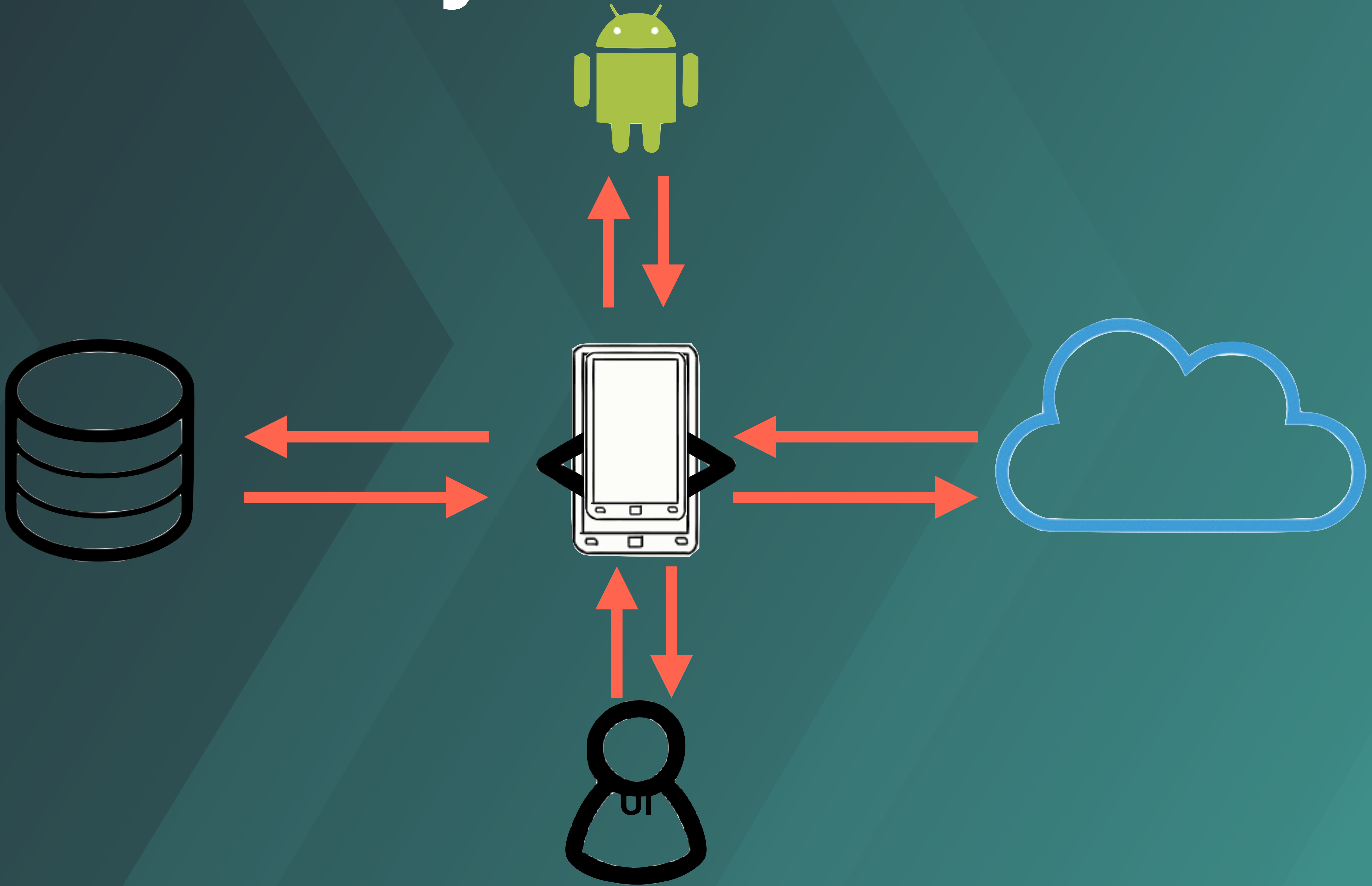
Why Reactive?



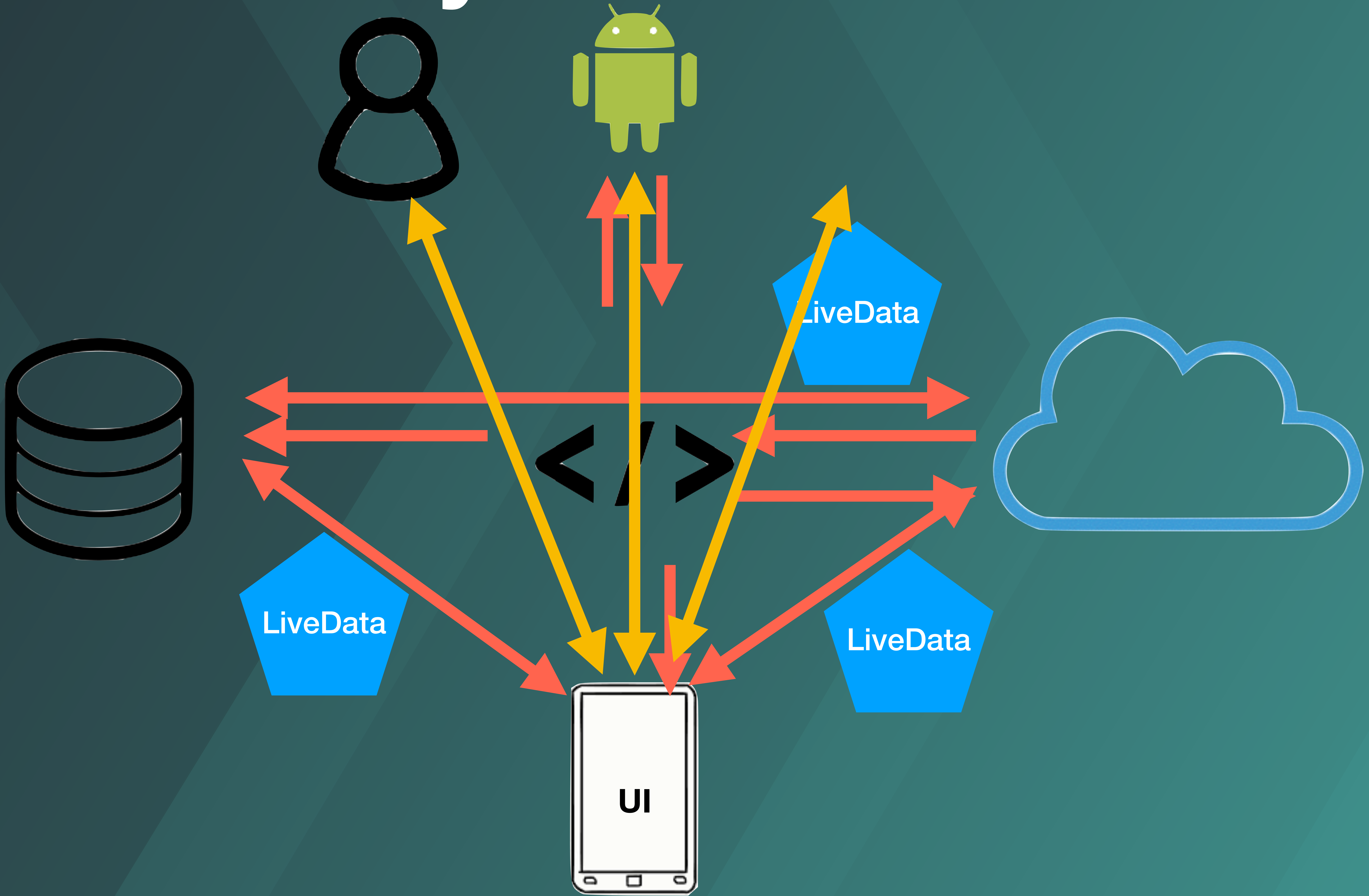
Why Reactive?



Why Reactive?



Why Reactive?



Rx{Java|Kotlin|Swift|Dart}

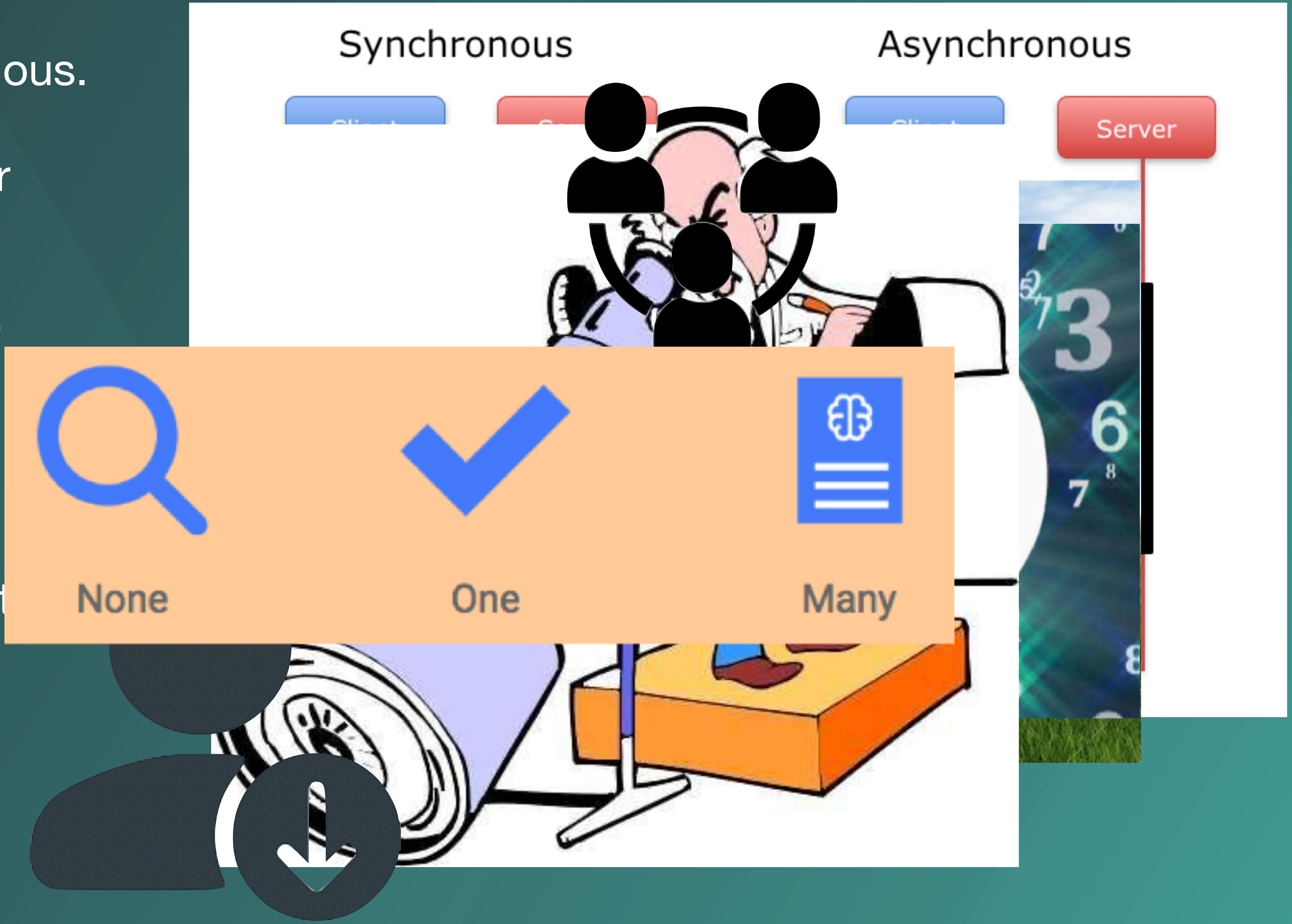
- A set of classes for representing sources of data.
- A set of classes for listening to data sources.
- A set of methods for modifying and composing the data.

RxSwift

http://reactivex.io/

Sources

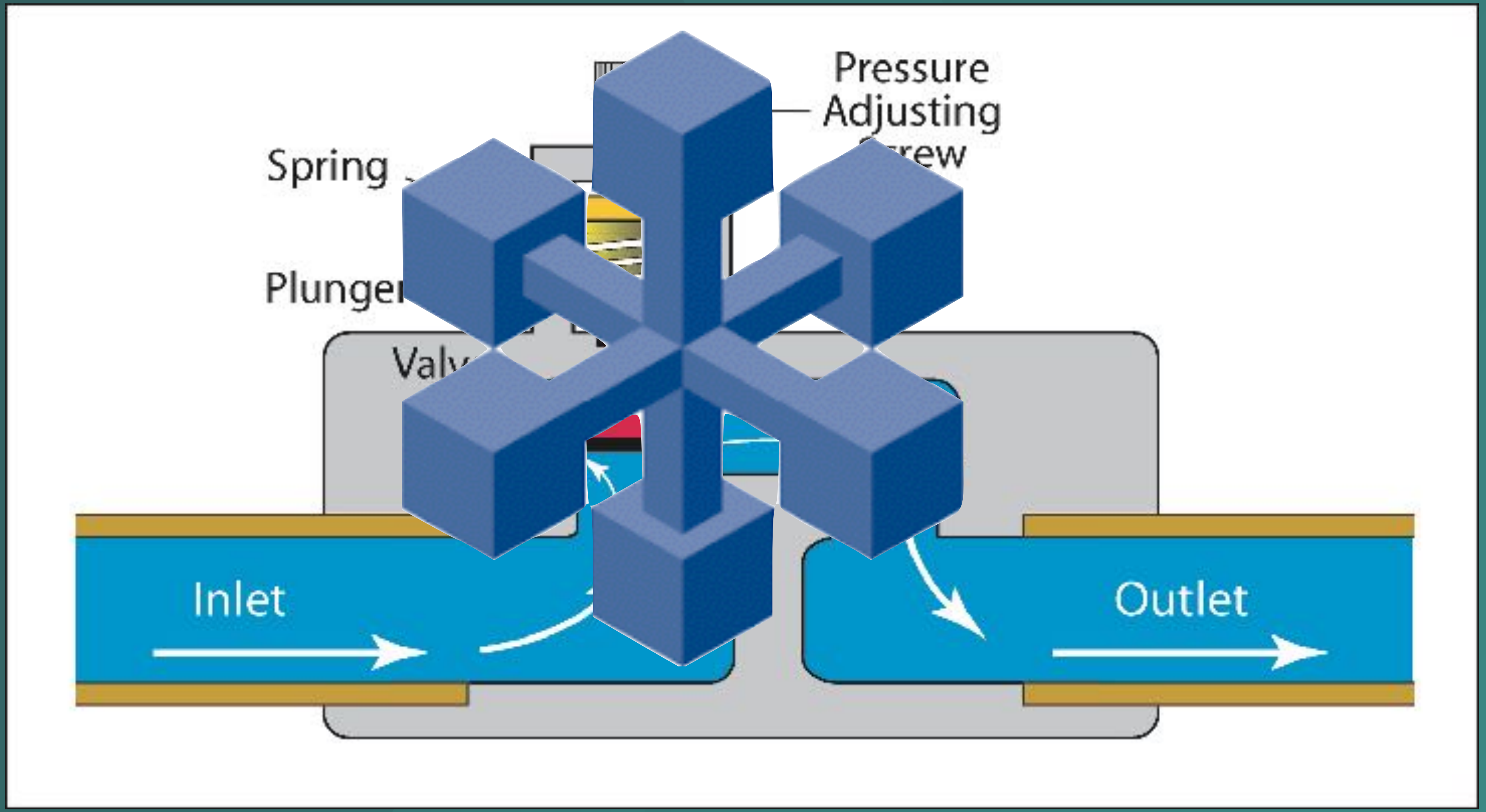
- Synchronous or asynchronous.
- Single item, many items, or empty.
- Terminates with an error or succeeds to completion.
- May never terminate!
- Just an implementation of t
Observer pattern.



Sources

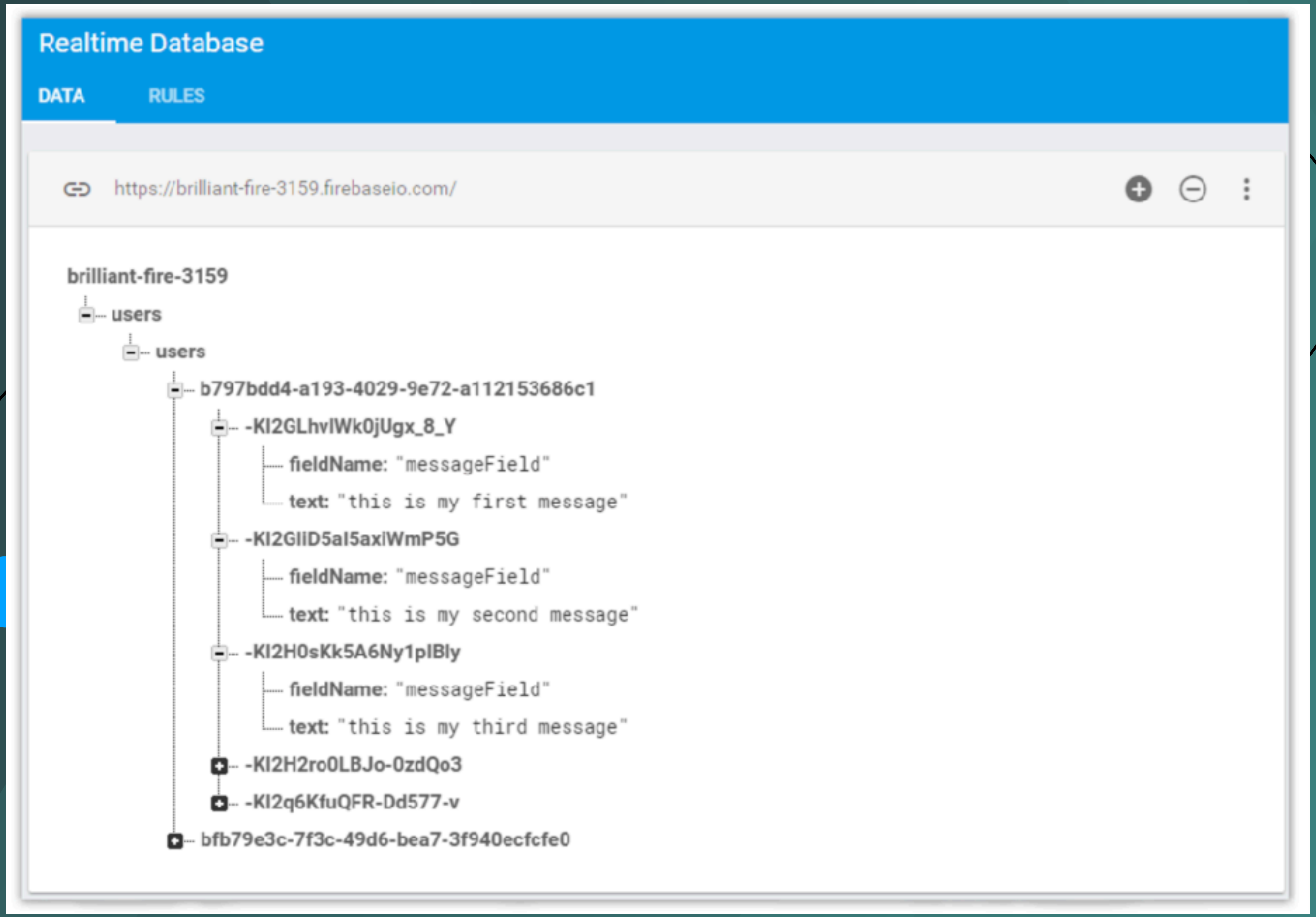
- Observable<T>
  - Emits 0 to N items.
  - Terminates with complete or error.
  - Does not have backpressure.
- Flowable<T>
  - Emits 0 to N items.
  - Terminates with complete or error.
  - Has backpressure.

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



Flowable vs. Observable

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

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



Realtime Database

DATA RULES

https://brilliant-fire-3159.firebaseio.com/

```
brilliant-fire-3159
 ⊟- users
    ⊟- users
       ⊟- b797bdd4-a193-4029-9e72-a112153686c1
          ⊟- -KI2GLhvIWk0jUgx_8_Y
               fieldName: "messageField"
               text: "this is my first message"
          ⊟- -KI2GlID5aI5axIWmP5G
               fieldName: "messageField"
               text: "this is my second message"
          ⊟- -KI2H0sKk5A6Ny1pIBly
               fieldName: "messageField"
               text: "this is my third message"
          ⊞- -KI2H2rc0LBJo-0zdQo3
          ⊞- -KI2q6KfuQFR-Dd577-v
       ⊞- bfb79e3c-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)
}

interface Disposable{
    fun dispose()
}
```

Flowable<User>

```
interface Subscriber<T>{
    fun onNext(t: T)
    fun onComplete()
    fun onError(t: 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 scalar
    - 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")

Single.just("Hello")
```

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

Flowable.fromArray(array)
Flowable.fromIterable(list)
Observable.fromCallable {
    setName()
}
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.onComplete()
}
```

```
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

```
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)
}
```

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

`Flowable<User>`

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

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

Observing Sources

```
val observable: Observable<String> = Observable.just("Hello")
val observer = object : DisposableObserver<String>() {
val disposable = observable.subscribeWith(object : DisposableObserver<String>() {
    override fun onComplete() {
        //...
    }
    override fun onNext(t: String) {
        //...
    }
    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.

- 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()
```

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

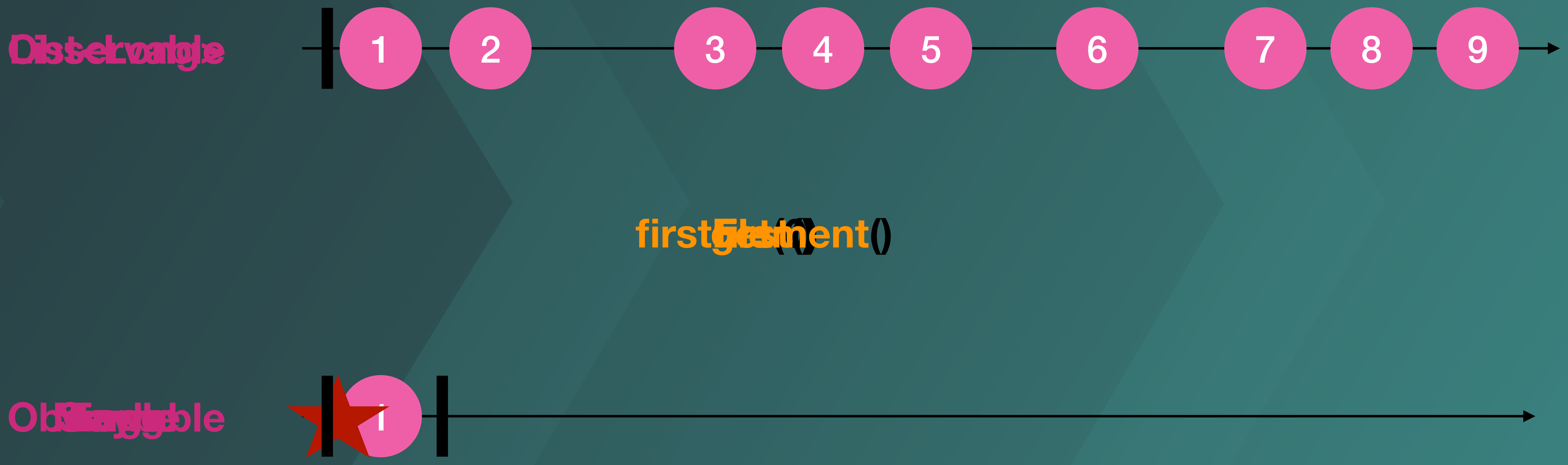
Operators

```
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) {
                        loge("Unable to update the user details", error)
                    }
                })
            }
            //...
        })
    }
}
```

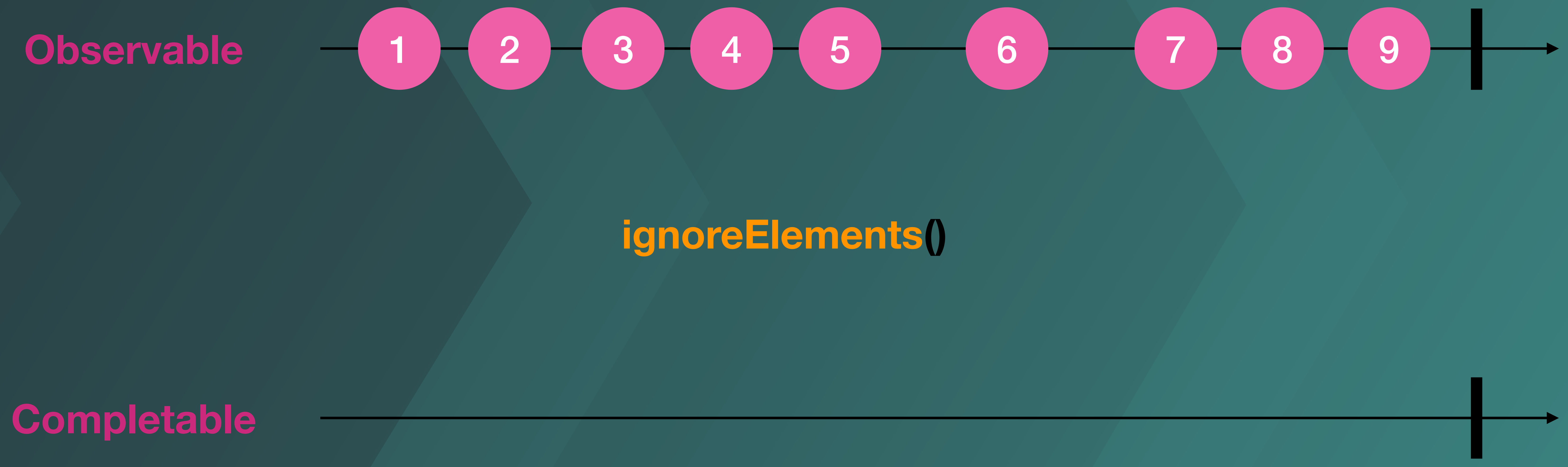
Operators

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

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

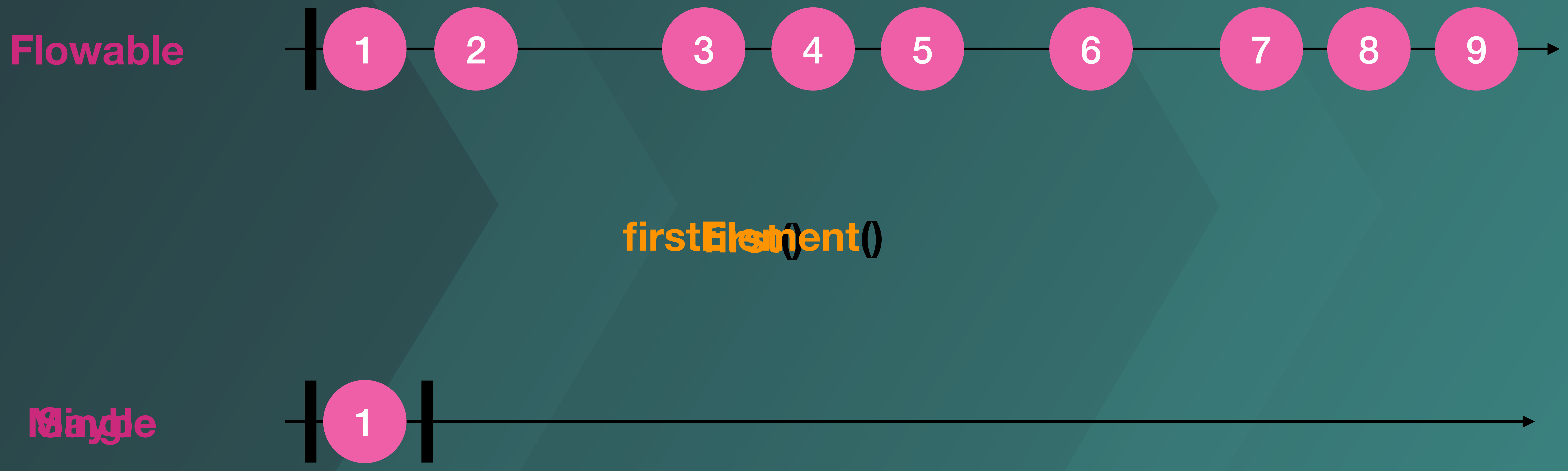
Operators



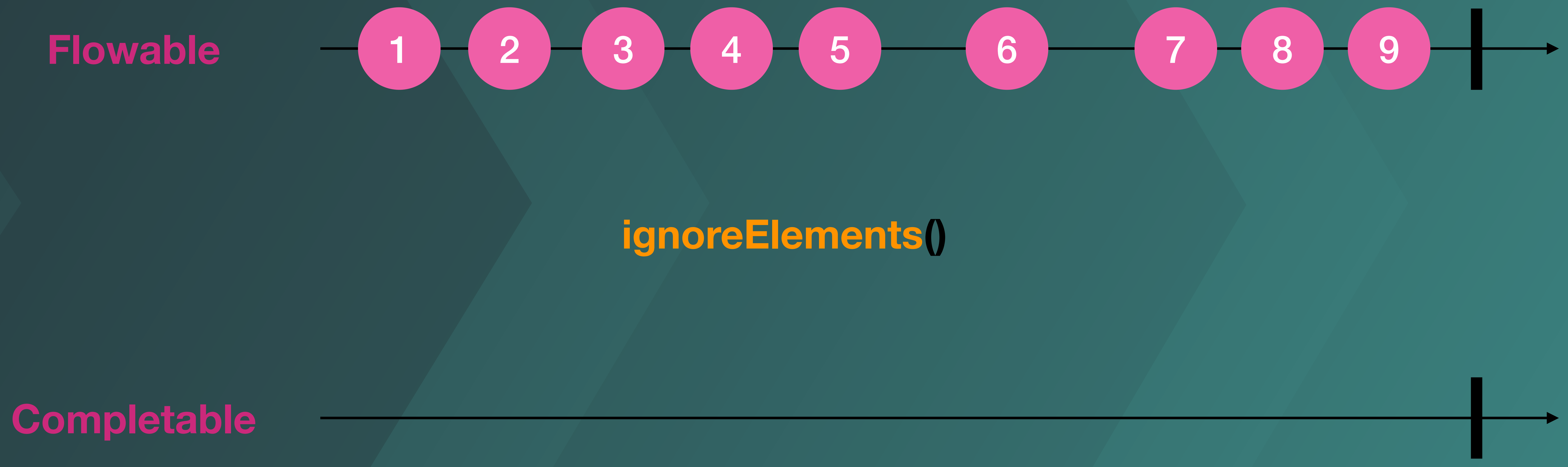
Operators



Operators



Operators



http://reactivex.io/documentation/operators.html

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()
```

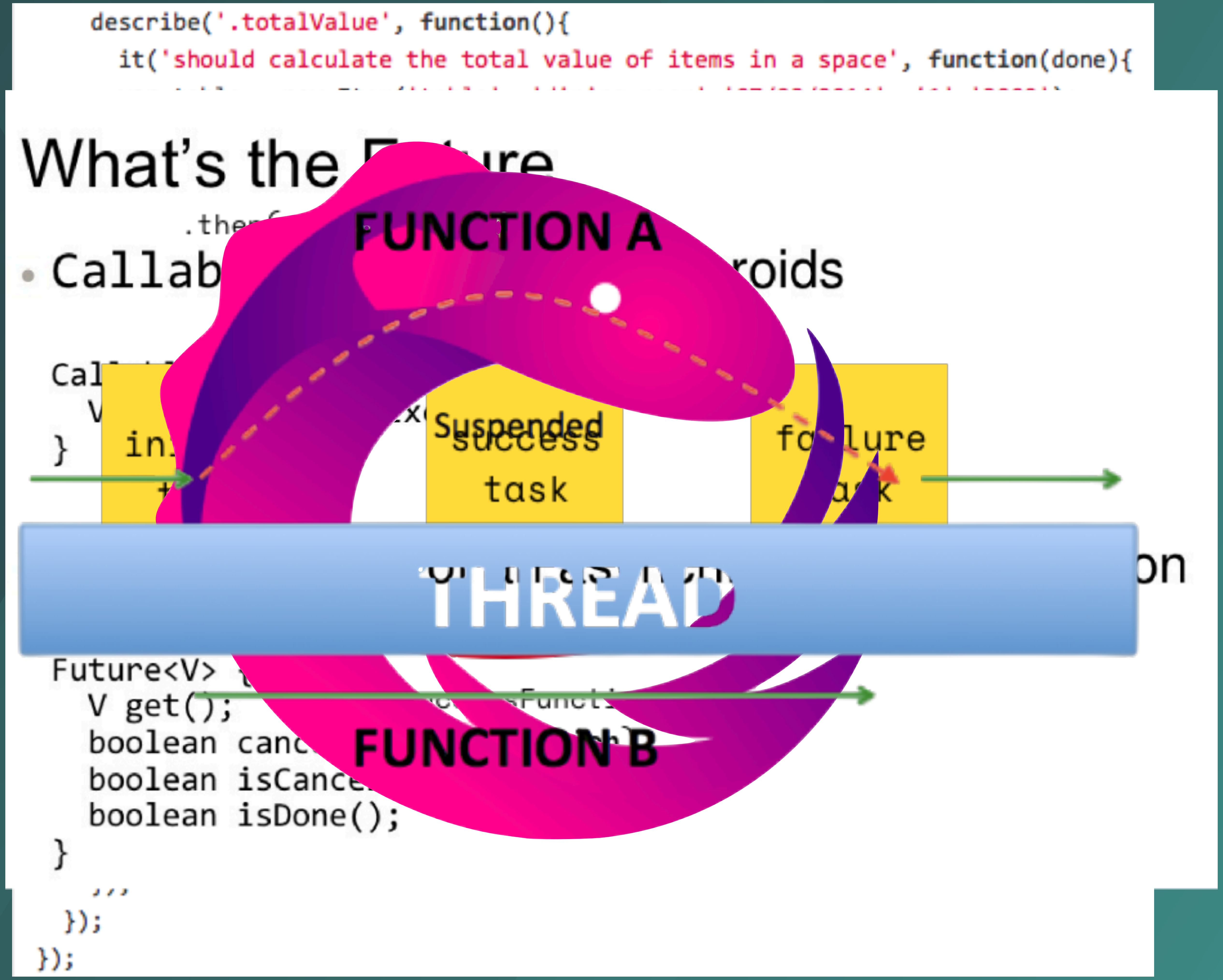
Dependencies

DEMO

```
allprojects {
    repositories {
        maven { url "https://oss.jfrog.org/libs-snapshot" }
    }
}

dependencies {
   implementation 'io.reactivex.rxjava3:rxandroid:3.0.0'
    // Because RxAndroid releases are few and far between, it is recommended you also
    // explicitly depend on RxJava's latest version for bug fixes and new features.
    // (see https://github.com/ReactiveX/RxJava/releases for latest 3.x.x version)
    implementation 'io.reactivex.rxjava3:rxjava:3.1.5'
}
```

Options

- Callbacks
- Futures
- Promises
- Rx
- Coroutines



https://kotlinlang.org/docs/reference/coroutines-overview.html

Coroutines

```
fun requestToken(): Token {
    // make a token request and waits
    // 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
}

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)
}
```

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

```kotlin
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)
}
```

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

Suspend function 're stToken' 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) {
    viewModelScope.launch {
      val token = requestToken()
      val post = createPost(token, item)
      processPost(post)
    }
}
```

Builders

```kotlin
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
}
```

Dependencies

DEMO

```
dependencies {
  implementation “org.jetbrains.kotlinx:kotlinx-coroutines-android:<version>”
}
```

Usage

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

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

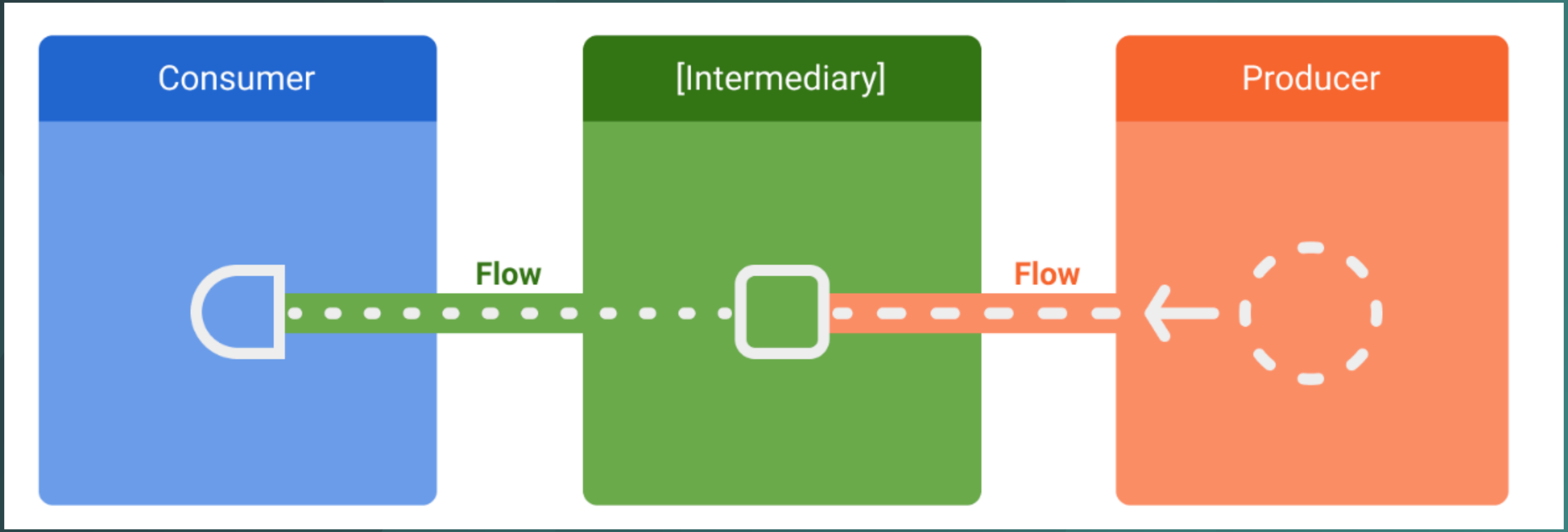
```
implementation “androidx.lifecycle:lifecycle-viewmodel-ktx:<version>”
```

Usage

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

implementation “androidx.lifecycle:lifecycle-livedata-ktx:<version>”

Flow



Flow

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

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

```
Output:

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...")
    flow.collect { value -> println(value) }
    println("Calling collect again...")
    flow.collect { value -> println(value) }
}
```

```
Output:

Calling foo...
Calling collect...
Flow started
1
2
3
Calling collect again...
Flow started
1
2
3
```

Flow Cancellation

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

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

```
Output:

Emitting 1
1
Emitting 2
2
Done
```

Flow Operators

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

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

Output:

```
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

DEMO

```
@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
```

kotlinlang.org/docs/reference/coroutines/flow.html

Room with Flow

```
@Dao
abstract class ExampleDao {
    @Query("SELECT * FROM Example")
    abstract fun getExamples(): Flow<List<Example>>
}
```

Creating a Custom Flow

```
class NewsRemoteDataSource(
    private val newsApi: NewsApi,
    private val refreshIntervalMs: Long = 5000
) {
    val latestNews: Flow<List<ArticleHeadline>> = flow {
        while(true) {
            val latestNews = newsApi.fetchLatestNews()
            emit(latestNews) // Emits the result of the request to the flow
            delay(refreshIntervalMs) // Suspends the coroutine for some time
        }
    }
}

// Interface that provides a way to make network requests with suspend functions
interface NewsApi {
    suspend fun fetchLatestNews(): List<ArticleHeadline>
}
```

Creating a Custom Flow

```
class NewsRemoteDataSource(
    private val newsApi: NewsApi,
    private val refreshIntervalMs: Long = 5000
) {
    val latestNews: Flow<List<ArticleHeadline>> = flow {
        while(true) {
            val latestNews = newsApi.fetchLatestNews()
            emit(latestNews) // Emits the result of the request to the flow
            delay(refreshIntervalMs) // Suspends the coroutine for some time
        }
    }
}

// Interface that provides a way to make network requests with suspend functions
interface NewsApi {
    suspend fun fetchLatestNews(): List<ArticleHeadline>
}
```

Collecting from a flow

```
class NewsRepository(
    private val newsRemoteDataSource: NewsRemoteDataSource,
    private val userData: UserData
) {
    /**
     * Returns the favorite latest news applying transformations on the flow.
     * These operations are lazy and don't trigger the flow. They just transform
     * the current value emitted by the flow at that point in time.
     */
    val favoriteLatestNews: Flow<List<ArticleHeadline>> =
        newsRemoteDataSource.latestNews
            // Intermediate operation to filter the list of favorite topics
            .map { news -> news.filter { userData.isFavoriteTopic(it) } }
            // Intermediate operation to save the latest news in the cache
            .onEach { news -> saveInCache(news) }
}
```

Collecting from a flow

```
class LatestNewsViewModel(
    private val newsRepository: NewsRepository
) : ViewModel() {
   init {
        viewModelScope.launch {
            // Trigger the flow and consume its elements using collect
            newsRepository.favoriteLatestNews.collect { favoriteNews ->
                // Update View with the latest favorite news
            }
        }
    }
}
```

Catching unexpected exceptions

```
class LatestNewsViewModel(
    private val newsRepository: NewsRepository
) : ViewModel() {
   init {
        viewModelScope.launch {
            newsRepository.favoriteLatestNews
                // Intermediate catch operator. If an exception is thrown,
                // catch and update the UI
                .catch { exception -> notifyError(exception) }
                .collect { favoriteNews ->
                    // Update View with the latest favorite news
                }
        }
    }
}
```

Catching unexpected exceptions

```
class LatestNewsViewModel(
    private val newsRepository: NewsRepository
) : ViewModel() {
   init {
        viewModelScope.launch {
          newsRepository.favoriteLatestNews
              // Intermediate catch operator. If an exception is thrown,
              // catch and update the UI
              .catch { exception -> notifyError(exception) }
              .collect { favoriteNews ->
                  // Update View with the latest favorite news
              }
        }
    }
}
```

Upstream Flow

Catching unexpected exceptions

```
class LatestNewsViewModel(
    private val newsRepository: NewsRepository
) : ViewModel() {
   init {
        viewModelScope.launch {
            newsRepository.favoriteLatestNews
                // Intermediate catch operator. If an exception is thrown,
                // catch and update the UI
                .catch { exception -> notifyError(exception) }
                .collect { favoriteNews ->
                    // Update View with the latest favorite news
                }
        }
    }
}
```

Upstream Flow

Downstream Flow

Catching unexpected exceptions

```
class LatestNewsViewModel(
    private val newsRepository: NewsRepository
) : ViewModel() {
   init {
        viewModelScope.launch {
            newsRepository.favoriteLatestNews
                // Intermediate catch operator. If an exception is thrown,
                // catch and update the UI
                .catch { exception -> notifyError(exception) }
                .collect { favoriteNews ->
                    // Update View with the latest favorite news
                }
        }
    }
}
```

Upstream Flow

Downstream Flow

Catching unexpected exceptions

```
class NewsRepository(...) {
    val favoriteLatestNews: Flow<List<ArticleHeadline>> =
        newsRemoteDataSource.latestNews
            .map { news -> news.filter { userData.isFavoriteTopic(it) } }
            .onEach { news -> saveInCache(news) }
            // If an error happens, emit the last cached values
            .catch { exception -> emit(lastCachedNews()) }
}
```

Catching unexpected exceptions

```
class NewsRepository(...) {
    val favoriteLatestNews: Flow<List<ArticleHeadline>> =
        newsRemoteDataSource.latestNews
            .map { news -> news.filter { userData.isFavoriteTopic(it) } }
            .onEach { news -> saveInCache(news) }
            // If an error happens, emit the last cached values
            .catch { exception -> emit(lastCachedNews()) }
}
```

Collecting Flows

```
userMessages.collect { messages ->
  listAdapter.submitList(messages)
}
```

Collecting Flows

```
userMessages.collect { messages ->
  listAdapter.submitList(messages)
}
```

Cold Flows

```
userMessages.collect { messages ->
  listAdapter.submitList(messages)
}
```

Cold Flows

```
userMessages.collect { messages ->
  listAdapter.submitList(messages)
}
```

Collect Flows from UI

Collect items when needed

- Lifecycle-aware alternatives
- androidx.lifecycle:lifecycle-livedata-ktx
  Flow<T>.asLiveData(): LiveData
- androidx.lifecycle:lifecycle-runtime-ktx
  Lifecycle. repeatOnLifecycle(state)
  Flow<T>. flowWithLifecycle (lifecycle, state)

Collect Flows from UI

```
// import androidx. lifecycle.asLiveData
class MessagesViewModel(repository: MessagesRepository) : ViewModel() {
  val userMessages = repository userMessages.asLiveData ()
  •••
}
```

Collect Flows from UI

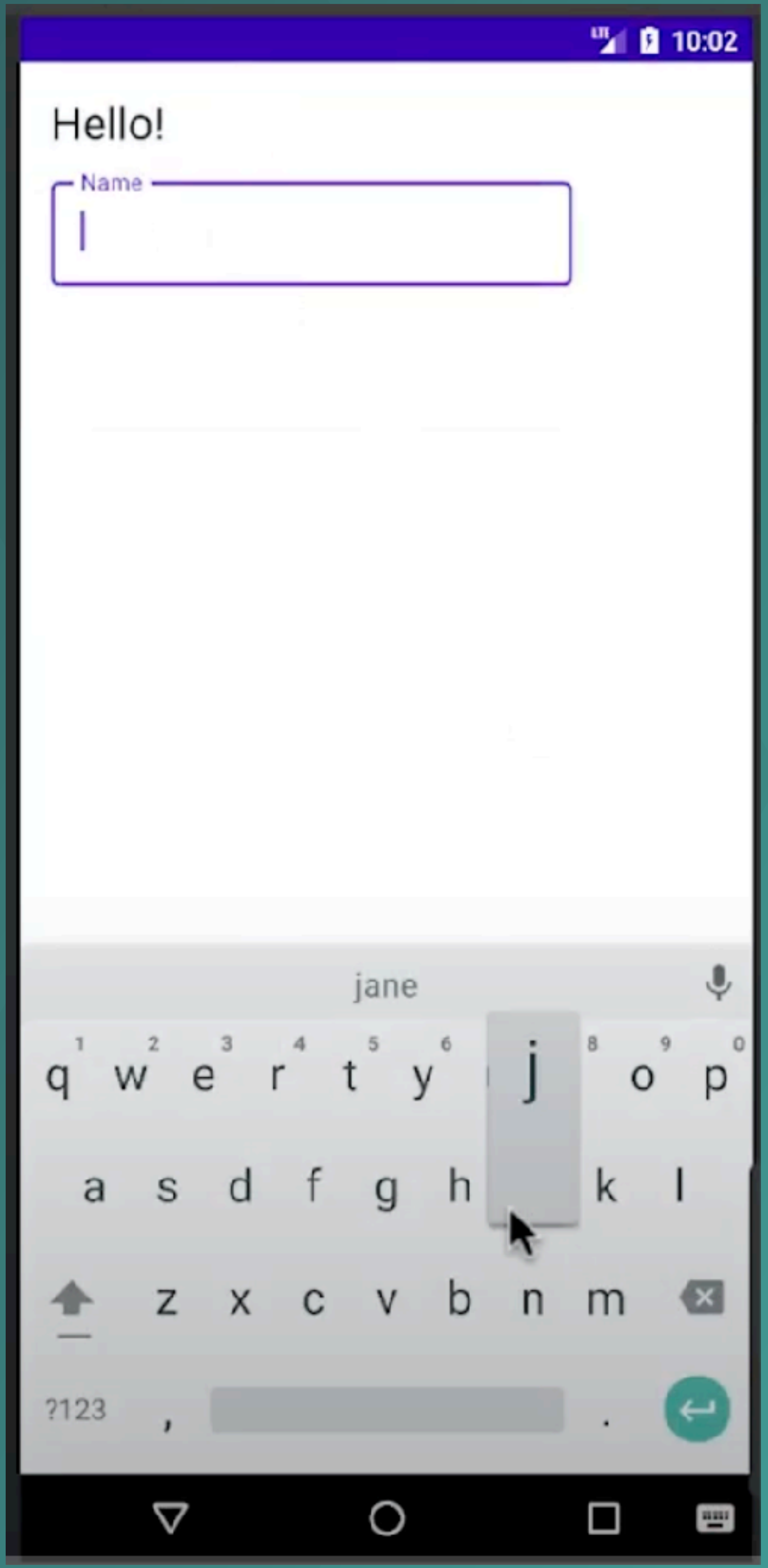
```
// import androidx. lifecycle. repeatOnLifecycle
class MessagesActivity : AppCompatActivity() {
 override fun onCreate(savedInstanceState: Bundle?) {
   ...
   lifecycleScope. launch {
     repeatOnLifecycle(Lifecycle.State.STARTED){
       viewModel.userMessages.collect { message ->
        listAdapter.submitList(message)
       }
      }
     }
    }
  }
}
```

Compose State

```
@Composable
private fun HelloContent() {
    Column(modifier = Modifier.padding(16.dp)) {
        Text(
            text = "Hello!",
            modifier = Modifier.padding(bottom = 8.dp),
            style = MaterialTheme.typography.bodyMedium
        )
        OutlinedTextField(
            value = "",
            onValueChange = { },
            label = { Text("Name") }
        )
    }
}
```

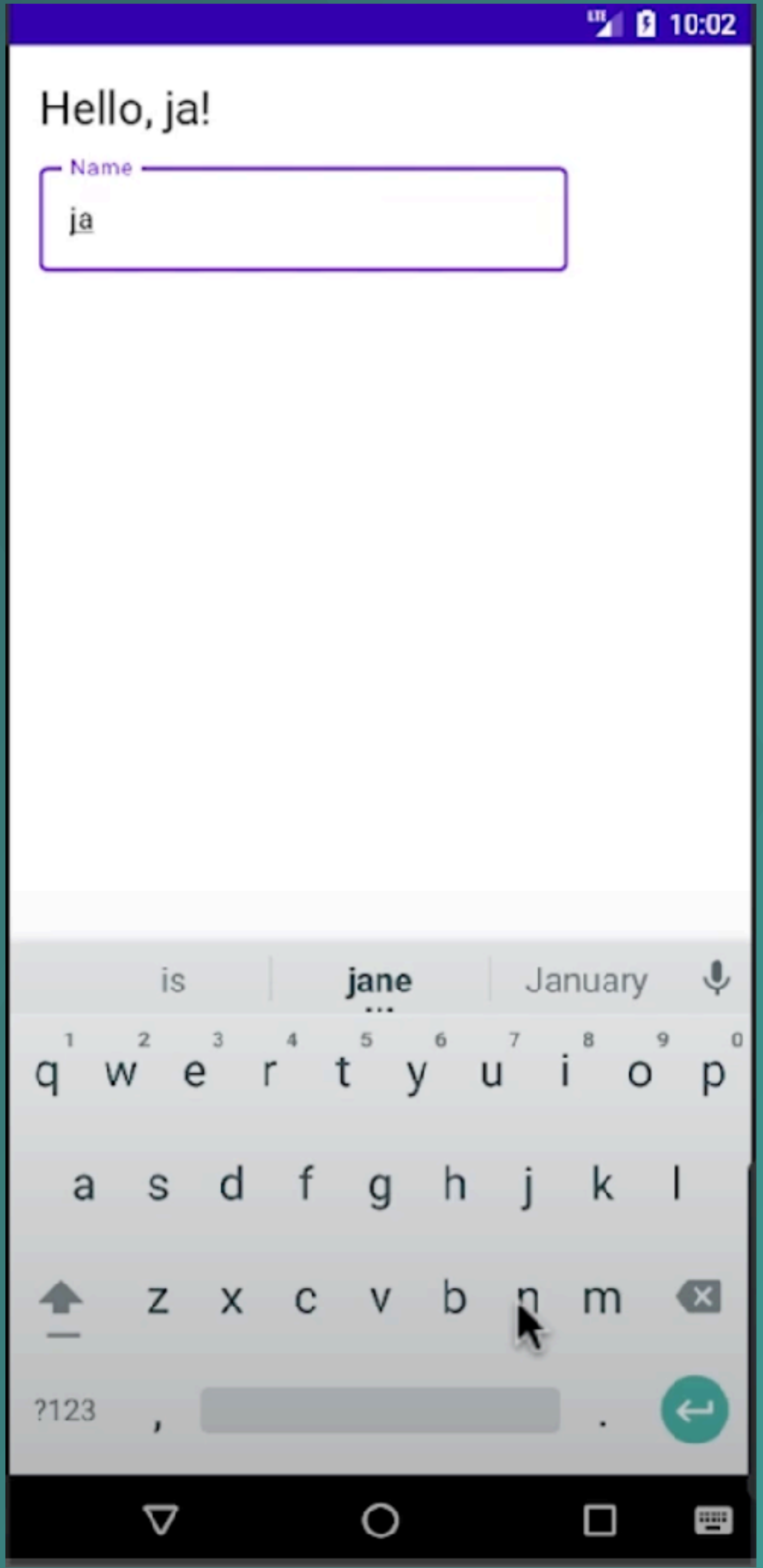
Compose State

```
@Composable
private fun HelloContent() {
    Column(modifier = Modifier.padding(16.dp)) {
        Text(
            text = "Hello!",
            modifier = Modifier.padding(bottom = 8.dp),
            style = MaterialTheme.typography.bodyMedium
        )
        OutlinedTextField(
            value = "",
            onValueChange = { },
            label = { Text("Name") }
        )
    }
}
```



Compose State

```
@Composable
fun HelloContent() {
  Column(modifier = Modifier.padding(16.dp)) {
    var name by mutableStateOf("")
    if (name.isNotEmpty()) {
      Text(
        text = "Hello, $name!",
        modifier = Modifier.padding(bottom = 8.dp),
        style = MaterialTheme.typography.bodyMedium
      )
    }
    OutlinedTextField(
      value = name,
      onValueChange = { name = it },
      label = { Text("Name") }
    )
  }
}
```

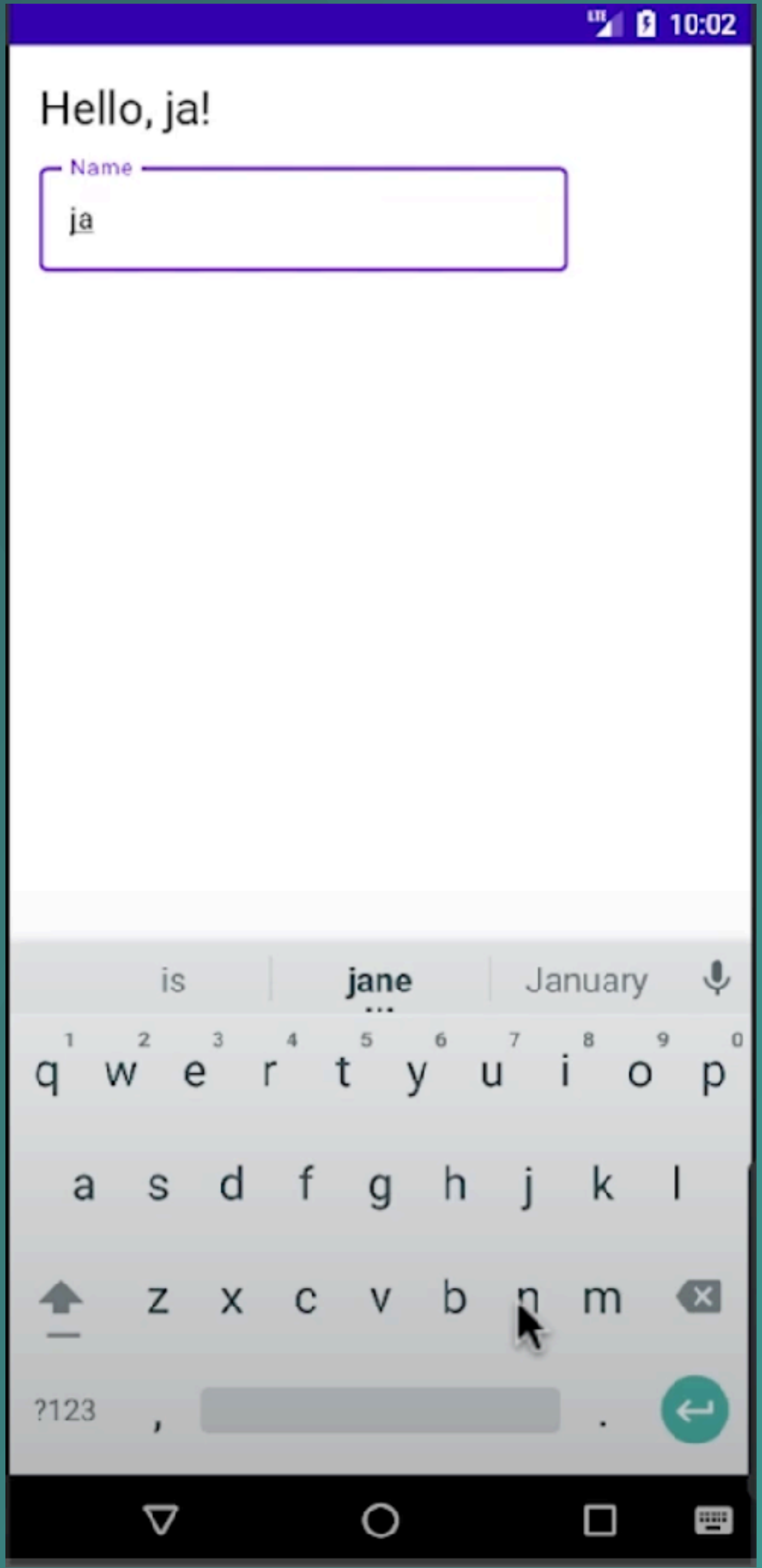

Hello, ja!

Name

ja

Compose State

```
@Composable
fun HelloContent() {
    Column(modifier = Modifier.padding(16.dp)) {
        var name by remember { mutableStateOf("") }
        if (name.isNotEmpty()) {
            Text(
                text = "Hello, $name!",
                modifier = Modifier.padding(bottom = 8.dp),
                style = MaterialTheme.typography.bodyMedium
            )
        }
        OutlinedTextField(
            value = name,
            onValueChange = { name = it },
            label = { Text("Name") }
        )
    }
}
```



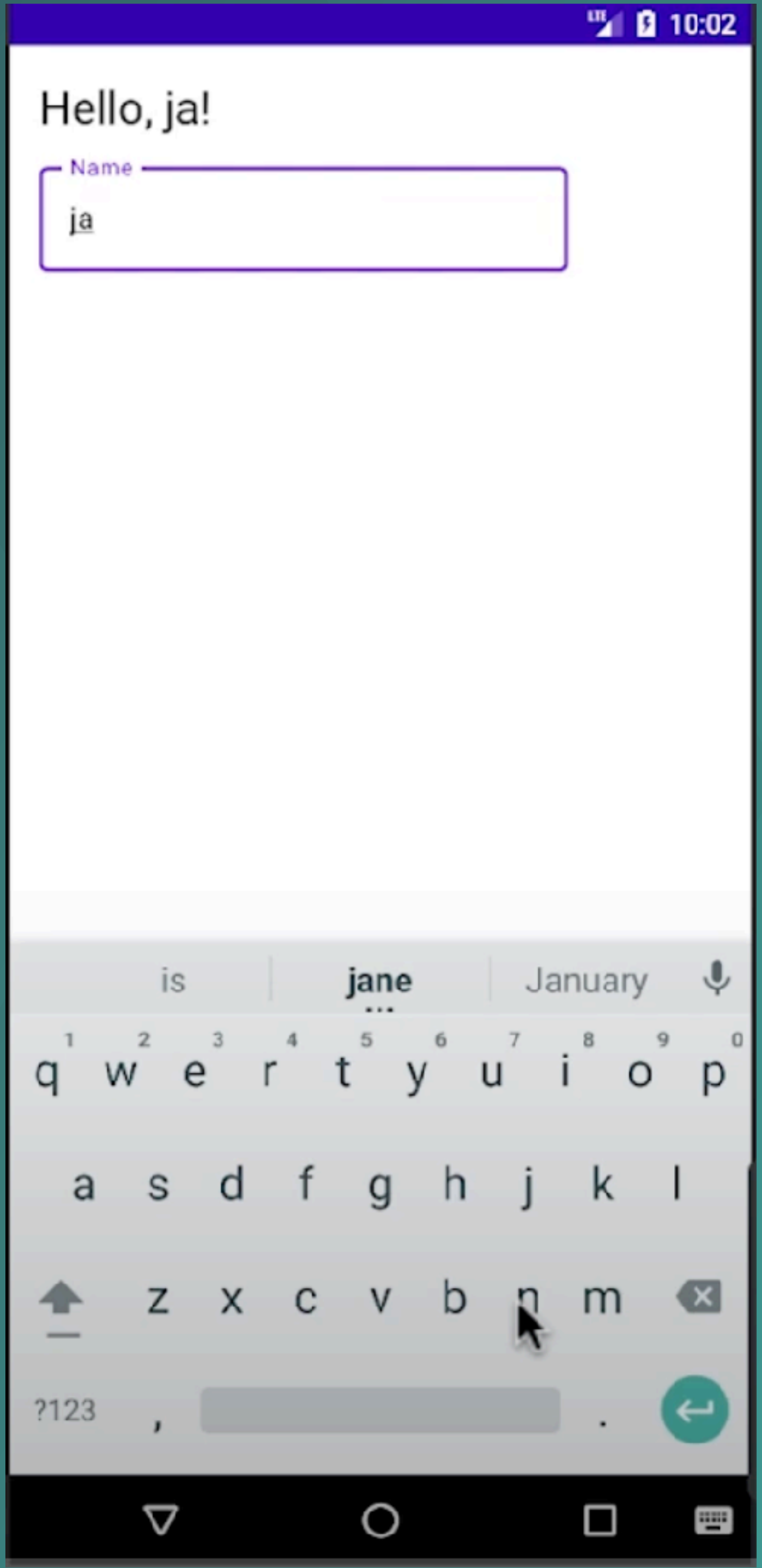
Hello, ja!

Name

ja

Compose State

```
@Composable
fun HelloContent() {
    Column(modifier = Modifier.padding(16.dp)) {
        var name by rememberSaveable { mutableStateOf("") }
        if (name.isNotEmpty()) {
            Text(
                text = "Hello, $name!",
                modifier = Modifier.padding(bottom = 8.dp),
                style = MaterialTheme.typography.bodyMedium
            )
        }
        OutlinedTextField(
            value = name,
            onValueChange = { name = it },
            label = { Text("Name") }
        )
    }
}
```



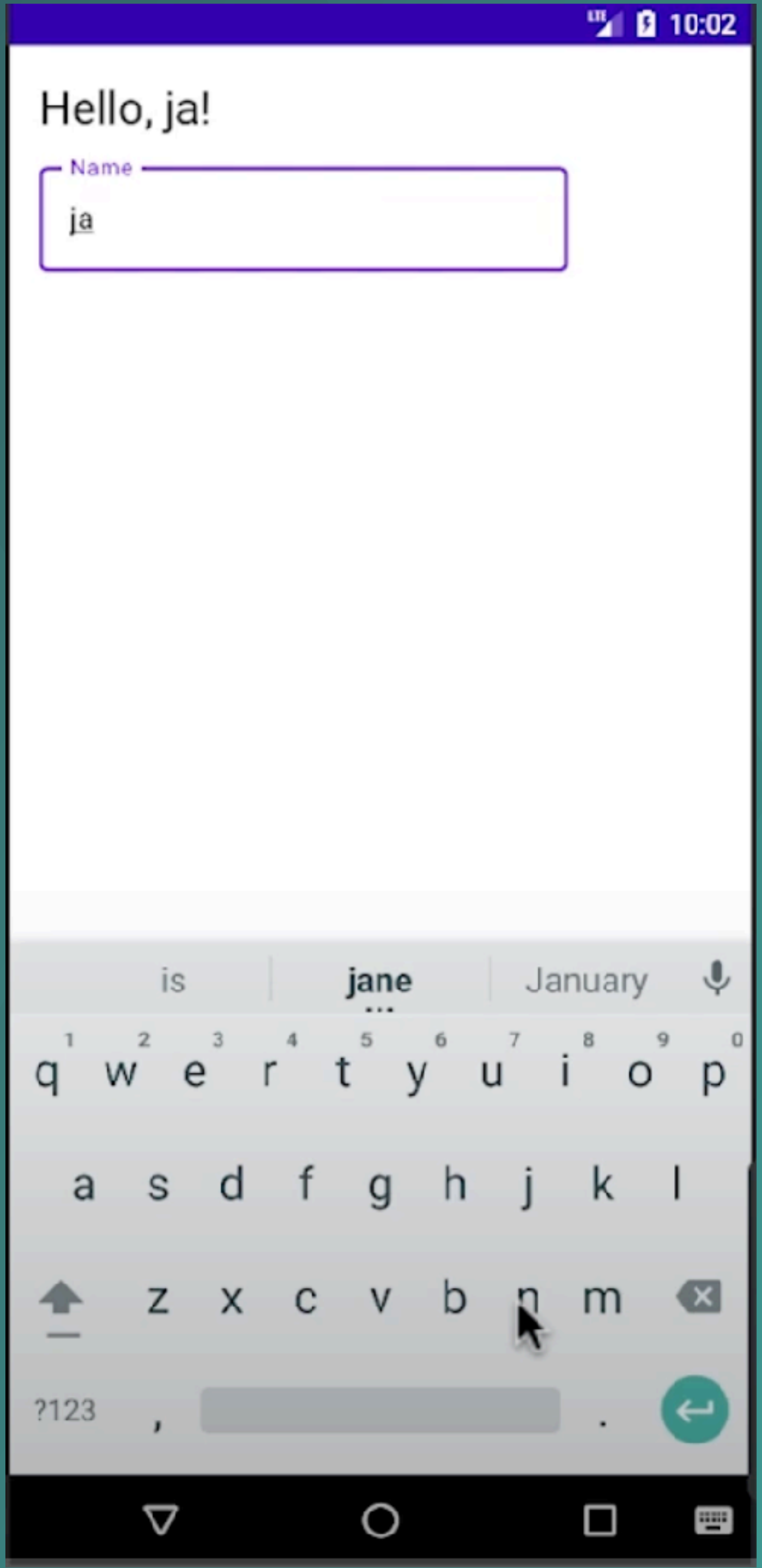
Hello, ja!

Name

ja

Compose State

```
@Composable
fun HelloScreen() {
    var name by rememberSaveable { mutableStateOf("") }
    HelloContent(name = name, onNameChange = { name = it })
}

@Composable
fun HelloContent(name: String, onNameChange: (String) -> Unit) {
    Column(modifier = Modifier.padding(16.dp)) {
        Text(
            text = "Hello, $name",
            modifier = Modifier.padding(bottom = 8.dp),
            style = MaterialTheme.typography.bodyMedium
        )
        OutlinedTextField(value = name, onValueChange = onNameChange,
                          label = { Text("Name") })
    }
}
```

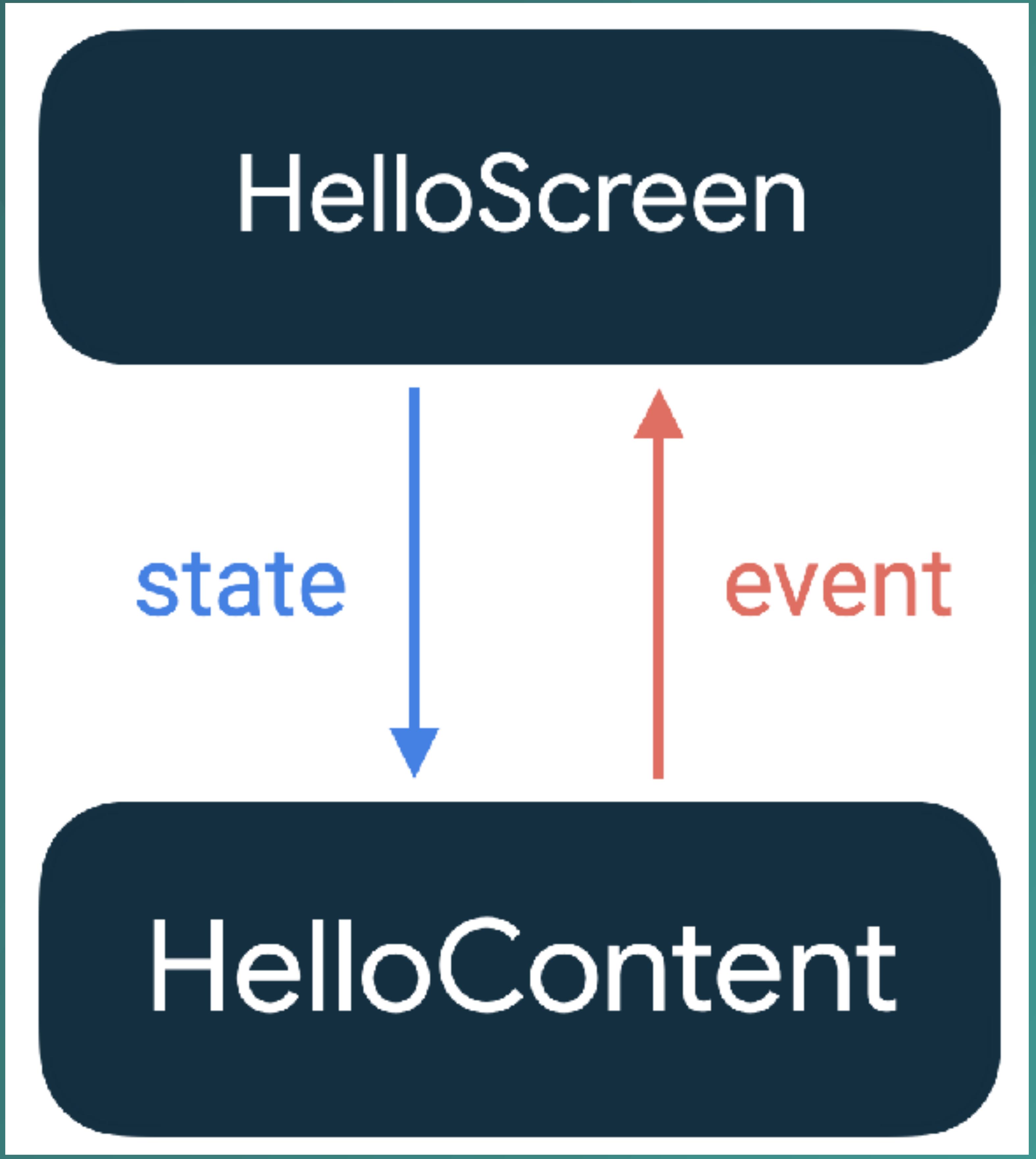

Hello, ja!

Name

ja

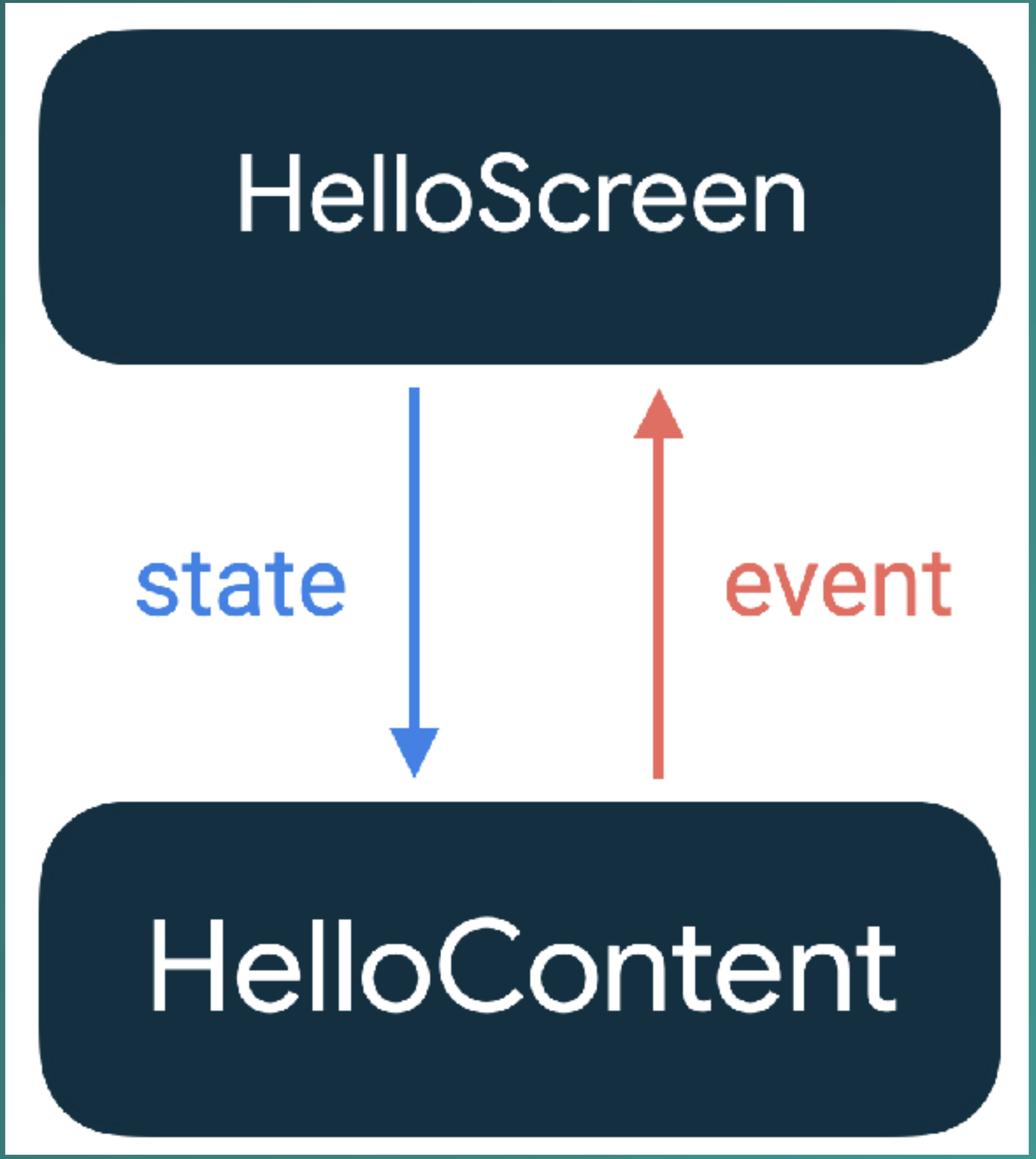
State Hoisting

```
@Composable
fun HelloScreen() {
    var name by rememberSaveable { mutableStateOf("") }
    HelloContent(name = name, onNameChange = { name = it })
}

@Composable
fun HelloContent(name: String, onNameChange: (String) -> Unit) {
    Column(modifier = Modifier.padding(16.dp)) {
        Text(
            text = "Hello, $name",
            modifier = Modifier.padding(bottom = 8.dp),
            style = MaterialTheme.typography.bodyMedium
        )
        OutlinedTextField(value = name, onValueChange = onNameChange,
                          label = { Text("Name") })
    }
}
```



State Hoisting

```
@Composable
fun HelloScreen() {
    var name by rememberSaveable { mutableStateOf("") }
    HelloContent(name = name, onNameChange = { name = it })
}

@Composable
fun HelloContent(name: String, onNameChange: (String) -> Unit) {
    Column(modifier = Modifier.padding(16.dp)) {
        Text(
            text = "Hello, $name",
            modifier = Modifier.padding(bottom = 8.dp),
            style = MaterialTheme.typography.bodyMedium
        )
        OutlinedTextField(value = name, onValueChange = onNameChange,
                          label = { Text("Name") })
    }
}
```



State with LiveData

```
class HelloViewModel: ViewModel(){
  private val _name = MutableLiveData("")
  val name: LiveData<String> = _name

  fun onNameChange(newName: String){
    _name.value = newName
  }
}

@Composable
fun HelloScreen(helloViewModel: HelloViewModel = viewModel()) {
    var name by helloViewModel.name.observeAsState("")
    HelloContent(name = name, onNameChange = {helloViewModel.onNameChange(it)})
}
```

https://developer.android.com/develop/ui/compose/state

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.