How to cook a well done MVI for Android

Sergey Ryabov

- Android Engineer & Mobile Consultant
- Kotlin User Group SPb
- Android Academy SPb & Msk
- Bla-bla-bla
- Digital Nomad

DataArt
PUMA

PROBLEMS OF A MODERN APP

PROBLEMS OF A MODERN APP

- Lots of asynchronicity: REST, WebSockets, Pushes, …

PROBLEMS OF A MODERN APP

- Lots of asynchronicity: REST, WebSockets, Pushes, …
- State updates from random places

PROBLEMS OF A MODERN APP

- Lots of asynchronicity: REST, WebSockets, Pushes, …
- State updates from random places
- Big size

PROBLEMS OF A MODERN APP

- Lots of asynchronicity: REST, WebSockets, Pushes, …
- State updates from random places
- Big size
- Async * Size => Something changes Somewhere and it all F***ed up

PROBLEMS OF A MODERN APP

- Lots of asynchronicity: REST, WebSockets, Pushes, …
- State updates from random places
- Big size
- Async * Size => Something changes Somewhere and it all F***ed up
- Pain In The Ass when looking for “where it all started to go wrong”

COMMON STATE

COMMON STATE

- Single source of truth

COMMON STATE

- Single source of truth
- Easy to check at any particular time

COMMON STATE

- Single source of truth
- Easy to check at any particular time
- Clear sequence of changes

COMMON STATE

- Single source of truth
- Easy to check at any particular time
- Clear sequence of changes
- Easy to find the cause of the change

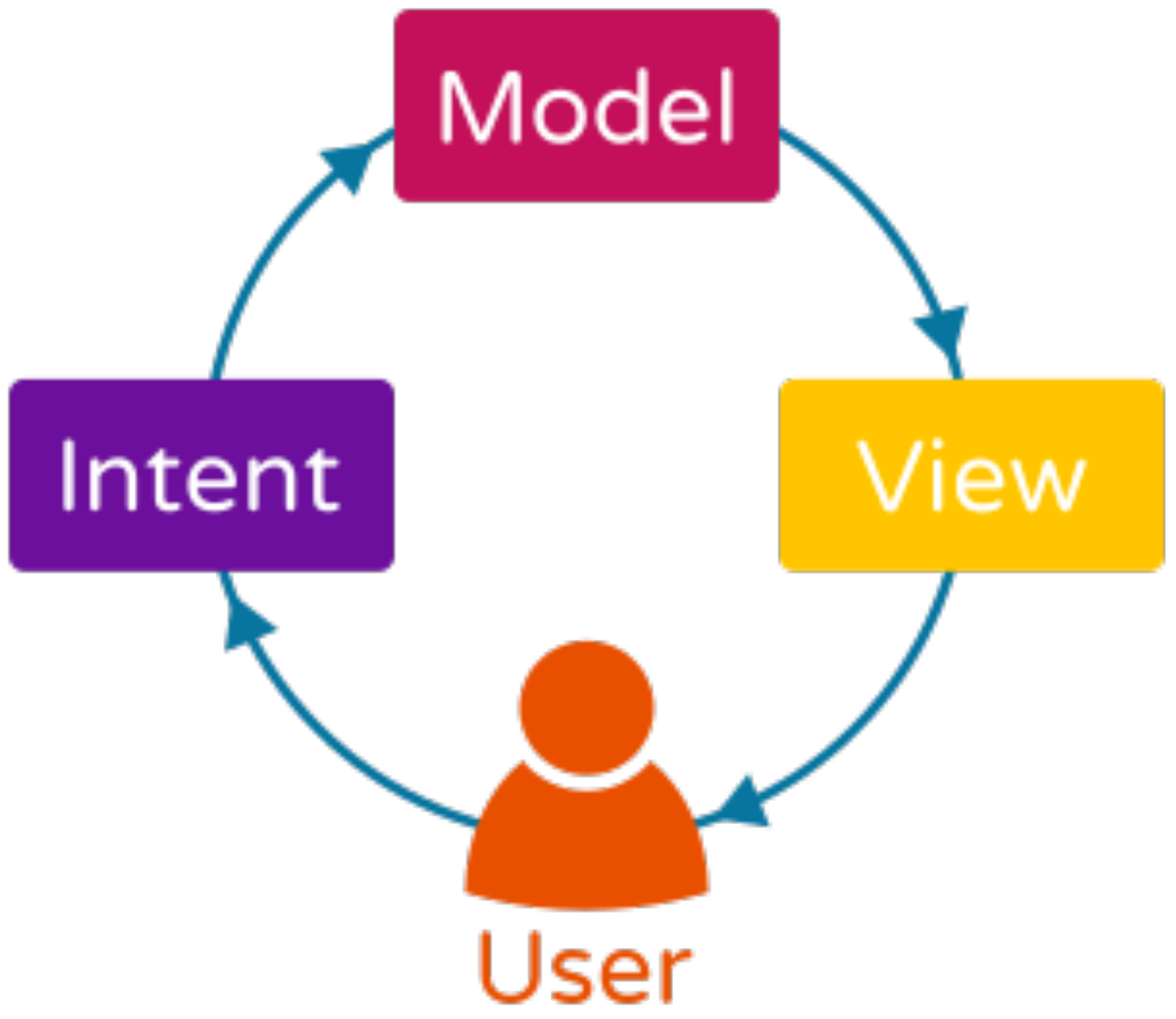
COMMON STATE

- Single source of truth
- Easy to check at any particular time
- Clear sequence of changes
- Easy to find the cause of the change
- Easy decoupled testing

COMMON STATE

Benefits are clear, but the arch implementation is still hard

UNIDIRECTIONAL DATA FLOW



UNIDIRECTIONAL DATA FLOW

```
view( model( intent() ) )
```

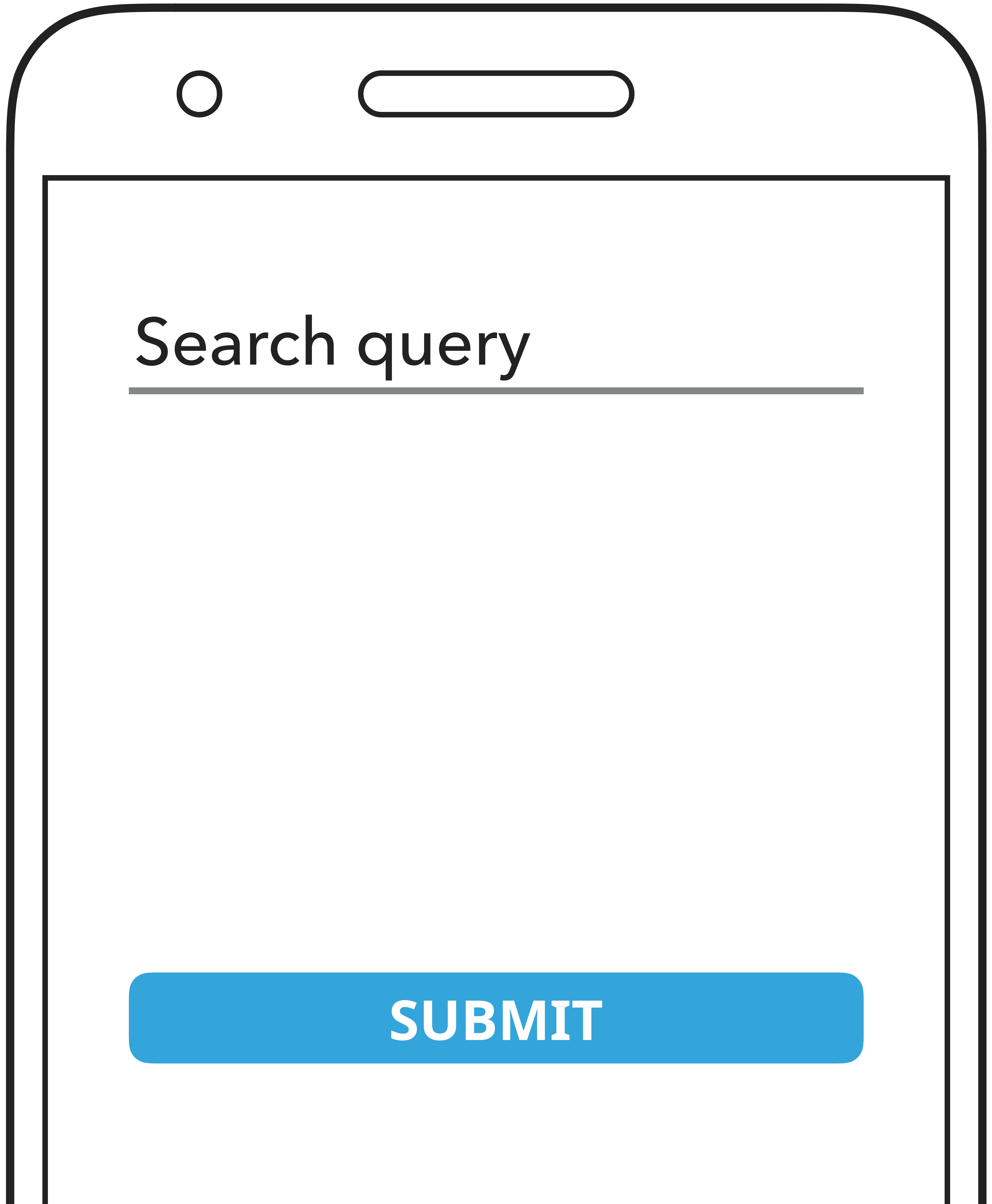
UNIDIRECTIONAL DATA FLOW

```
render( state( actions() ) )
```

Managing State with RxJava

Jake Wharton

REACTIVE STATE



REACTIVE STATE

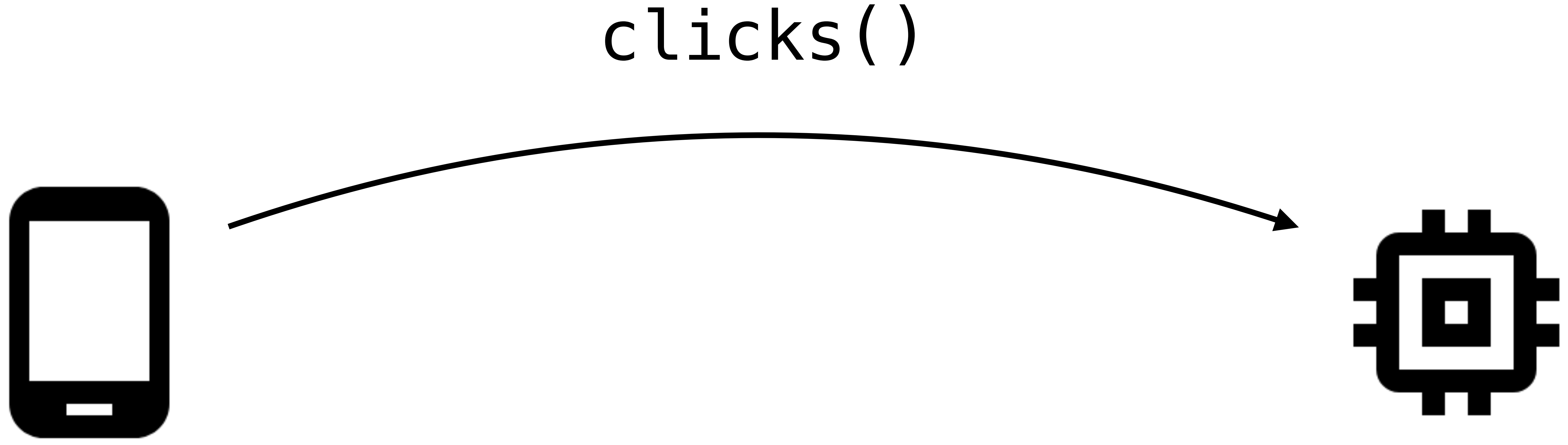
```
submitBtn.clicks()
    .doOnNext {
      submitBtn.isEnabled = false
      progressView.visibility = VISIBLE
    }
    .flatMap { api.search(searchView.text.toString()) }
    .observeOn(uiScheduler)
    .doOnNext { progressView.visibility = GONE }
    .subscribe(
        { data -> showData(data) },
        {
          submitBtn.isEnabled = true
          toast("Search failed")
        }
    )
```

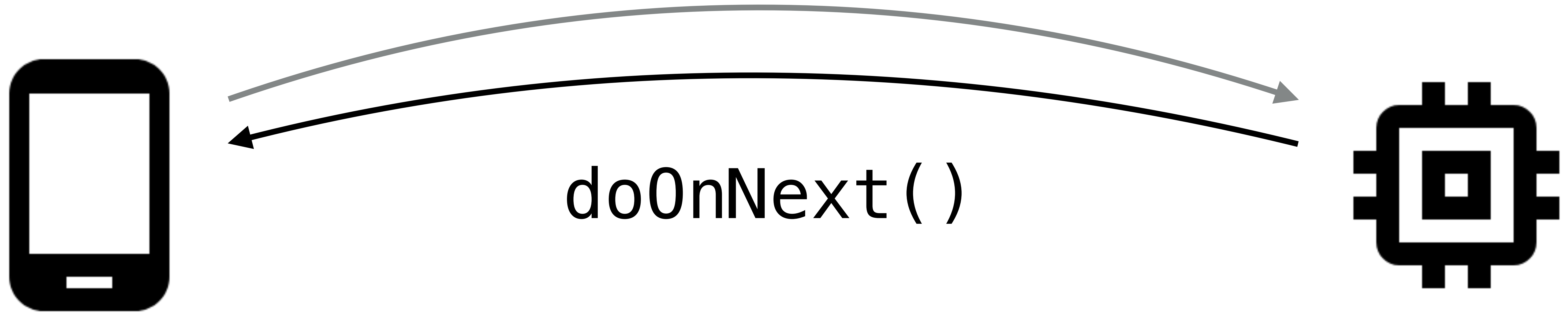
REACTIVE STATE

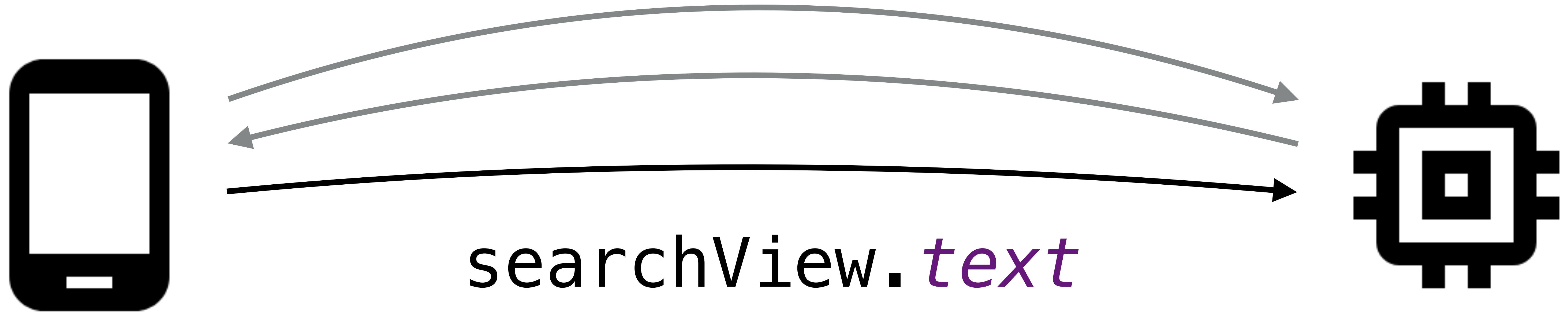
```
submitBtn.clicks()
    .doOnNext {
      submitBtn.isEnabled = false
      progressView.visibility = VISIBLE
    }
    .flatMap { api.search(searchView.text.toString()) }
    .observeOn(uiScheduler)
    .doOnNext { progressView.visibility = GONE }
    .subscribe(
        { data -> showData(data) },
        {
          submitBtn.isEnabled = true
          toast("Search failed")
        }
    )
```

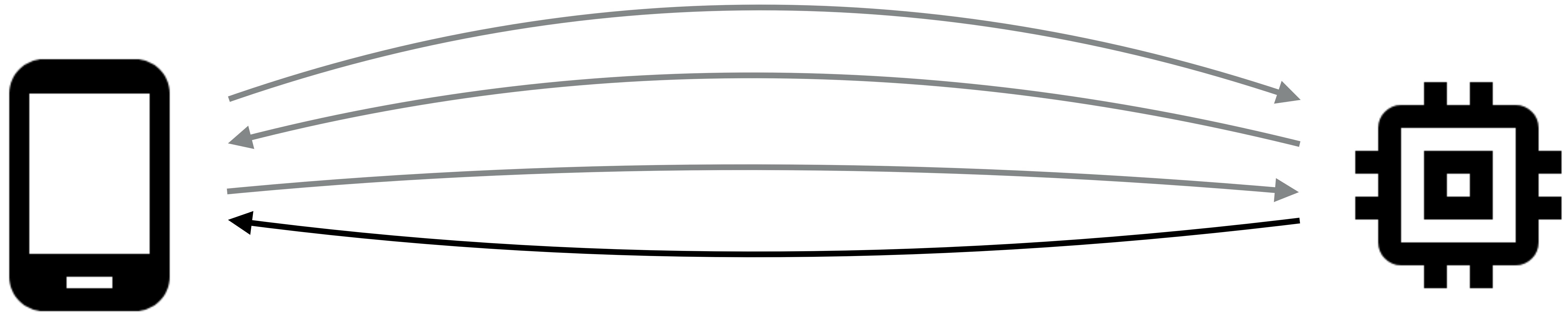
REACTIVE STATE

```
submitBtn.clicks()
    .doOnNext {
      submitBtn.isEnabled = false
      progressView.visibility = VISIBLE
    }
    .flatMap { api.search(searchView.text.toString()) }
    .observeOn(uiScheduler)
    .doOnNext { progressView.visibility = GONE }
    .subscribe(
        { data -> showData(data) },
        {
          submitBtn.isEnabled = true
          toast("Search failed")
        }
    )
```

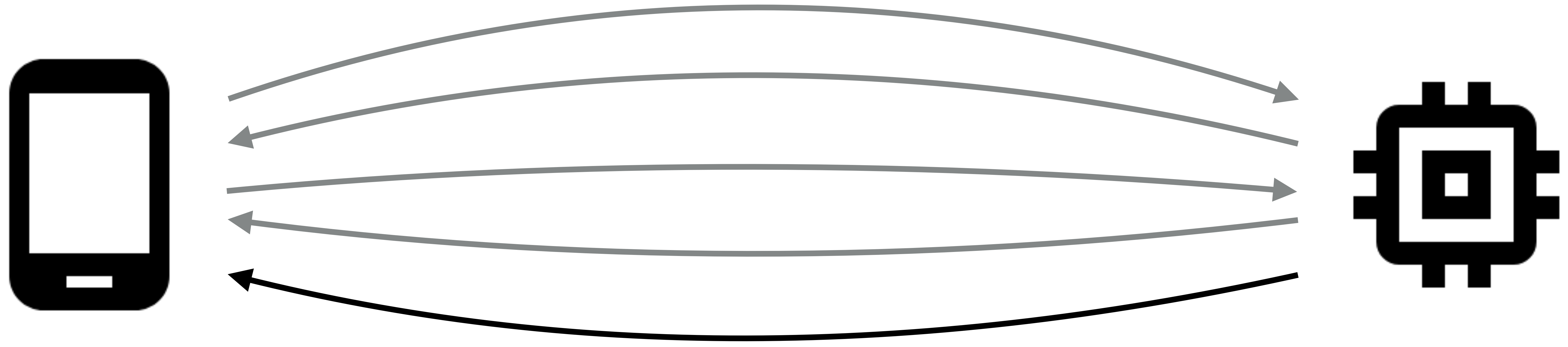

clicks()


doOnNext()

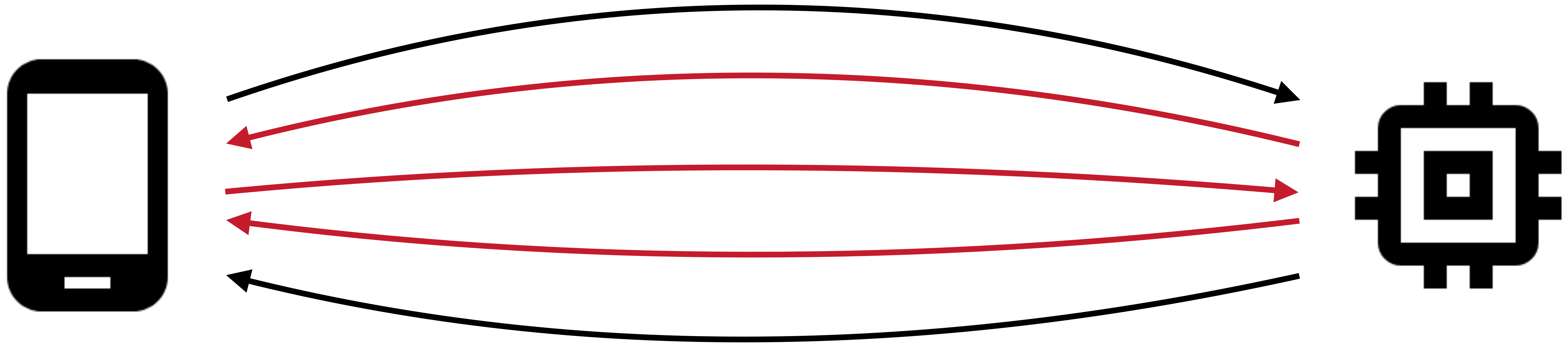

searchView.*text*



`doOnNext()`



`subscribe()`



REACTIVE STATE

```
sealed class UiAction {
  class SearchAction(val query: String) : UiAction()
}


sealed class UiState {
  object Loading : UiState()
  class Success(val data: Data) : UiState()
  class Failure(val error: Throwable) : UiState()
}
```

REACTIVE STATE

```
submitBtn.clicks()
    .map { SearchAction(searchView.text.toString()) }
    .flatMap { action ->
      api.search(action.query)
          .map<UiState> { result -> Success(result) }
          .onErrorReturn { e -> Failure(e) }
          .observeOn(uiScheduler)
          .startWith(Loading)
    }
    .subscribe { state ->
      submitBtn.isEnabled = state !is Loading
      progressView.visibility = if (state is Loading) VISIBLE else GONE
      when (state) {
        is Success -> showData(state.data)
        is Failure -> toast("Search failed")
      }
    }
```

REACTIVE STATE

```
val actions = submitBtn.clicks()
    .map { SearchAction(searchView.text.toString()) }

actions.flatMap { action ->
      api.search(action.query)
          .map<UiState> { result -> Success(result) }
          .onErrorReturn { e -> Failure(e) }
          .observeOn(uiScheduler)
          .startWith(Loading)
    }
    .subscribe { state ->
      submitBtn.isEnabled = state !is Loading
      progressView.visibility = if (state is Loading) VISIBLE else GONE
      when (state) {
        is Success -> showData(state.data)
        is Failure -> toast("Search failed")
      }
    }
```

REACTIVE STATE

```kotlin
val actions = submitBtn.clicks()
    .map { SearchAction(searchView.text.toString()) }

val states = actions.flatMap { action ->
      api.search(action.query)
          .map<UiState> { result -> Success(result) }
          .onErrorReturn { e -> Failure(e) }
          .observeOn(uiScheduler)
          .startWith(Loading)
    }

states.subscribe { state ->
      submitBtn.isEnabled = state !is Loading
      progressView.visibility = if (state is Loading) VISIBLE else GONE
      when (state) {
        is Success -> showData(state.data)
        is Failure -> toast("Search failed")
      }
    }
```

REACTIVE STATE

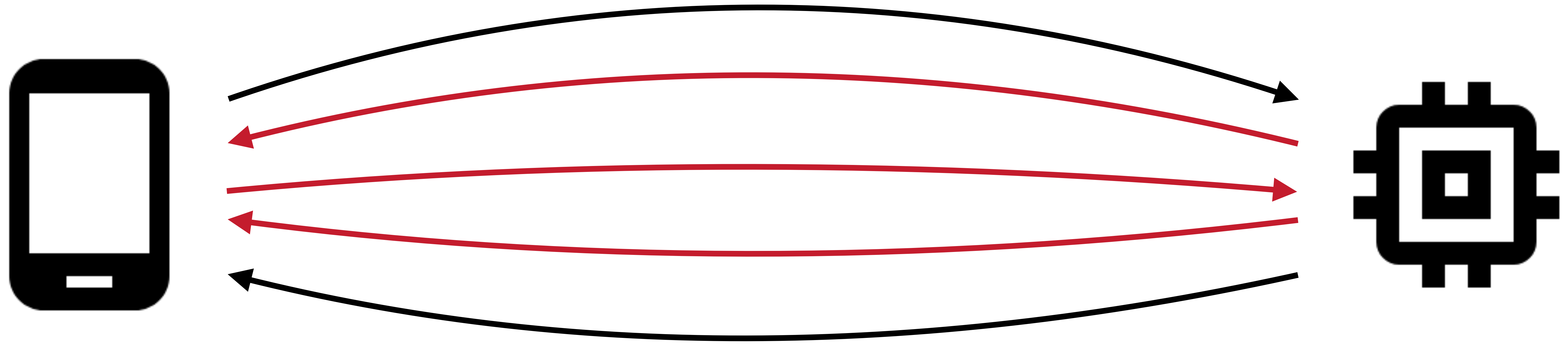
```kotlin
val actions = submitBtn.clicks()
    .map { SearchAction(searchView.text.toString()) }

val states = actions.flatMap { action ->
      api.search(action.query)
          .map<UiState> { result -> Success(result) }
          .onErrorReturn { e -> Failure(e) }
          .observeOn(uiScheduler)
          .startWith(Loading)
    }

states.subscribe(::render)

private fun render(state: UiState) {
  submitBtn.isEnabled = state !is Loading
  progressView.visibility = if (state is Loading) VISIBLE else GONE
  when (state) {
    is Success -> finish()
    is Failure -> toast("Search failed")
  }
}
```

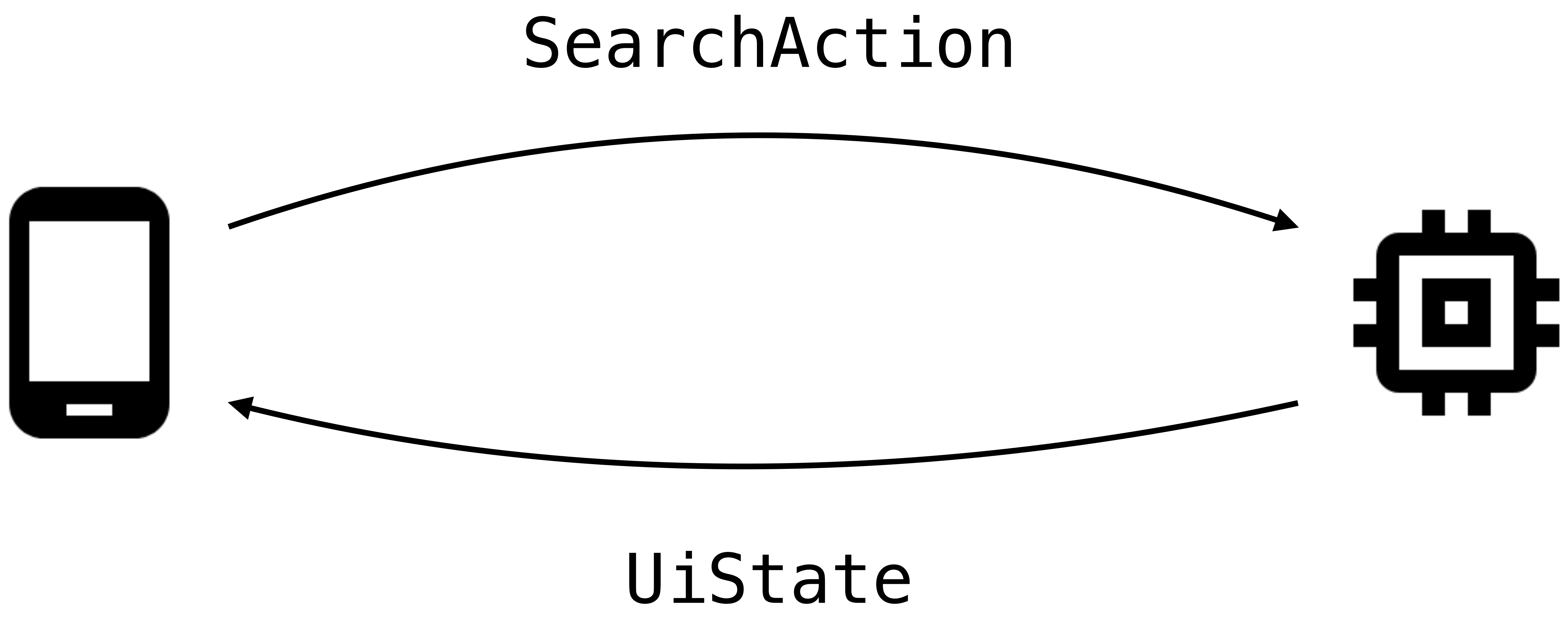
REACTIVE STATE

```
class SearchComponent(private val api: Api, val uiScheduler: Scheduler) {
  fun bind(actions: Observable<SearchAction>): Observable<UiState> {

    return actions.flatMap { action ->
      api.search(action.query)
          .map<UiState> { result -> Success(result) }
          .onErrorReturn { e -> Failure(e) }
          .observeOn(uiScheduler)
          .startWith(Loading)
    }

  }
}
```

REACTIVE STATE

```
class SearchComponent(private val api: Api, val uiScheduler: Scheduler) {
  fun bind(actions: Observable<SearchAction>): Observable<UiState> {

    return actions.flatMap { action ->
      api.search(action.query)
          .map<UiState> { result -> Success(result) }
          .onErrorReturn { e -> Failure(e) }
          .observeOn(uiScheduler)
          .startWith(Loading)
    }

  }
}

// View part
searchComponent.bind(actions).subscribe(::render)
```




SearchAction
UiState

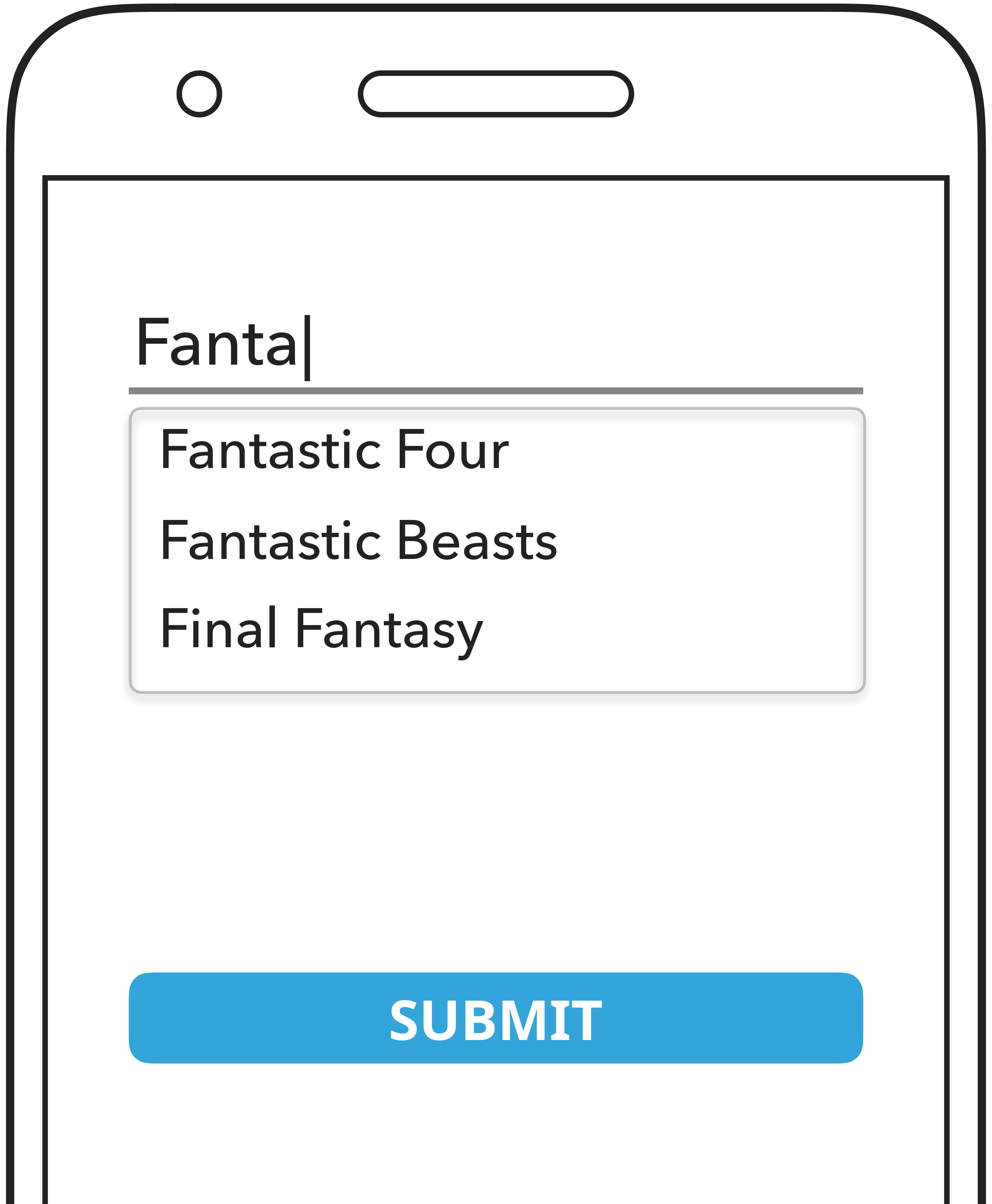
```
searchComponent.bind(actions).subscribe(::render)
searchComponent.bind(actions).subscribe(::render)

render( state( actions() ) )
```

What if we have more complex logic?

What if we have more complex logic?

How about two network calls?

REACTIVE STATE



```
sealed class UiAction : Action {
  class SearchAction(val query: String) : UiAction()
}
```

```kotlin
sealed class UiAction : Action {
  class SearchAction(val query: String) : UiAction()
  class LoadSuggestionsAction(val query: String) : UiAction()
}
```

```kotlin
sealed class UiState {
  object Loading : UiState()
  class Success(val data: Data) : UiState()
  class Failure(val error: Throwable) : UiState()
}
```

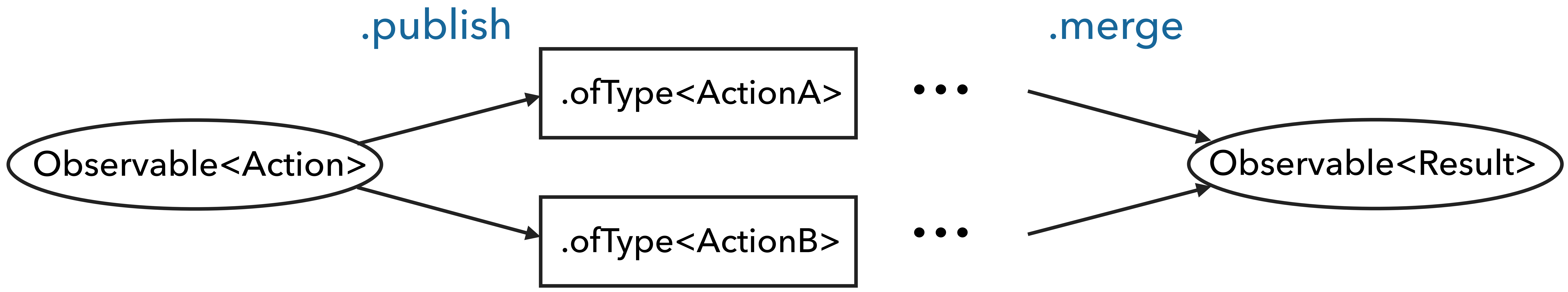
```kotlin
class UiState(
    val loading: Boolean = false,
    val data: String? = null,
    val error: Throwable? = null,
    val suggestions: List<String>? = null
)
```

```kotlin
sealed class InternalAction : Action {
  object SearchLoadingAction : InternalAction()
  class SearchSuccessAction(val data: String) : InternalAction()
  class SearchFailureAction(val error: Throwable) : InternalAction()
  class SuggestionsLoadedAction(val suggestions: List<String>) : InternalAction()
}
```

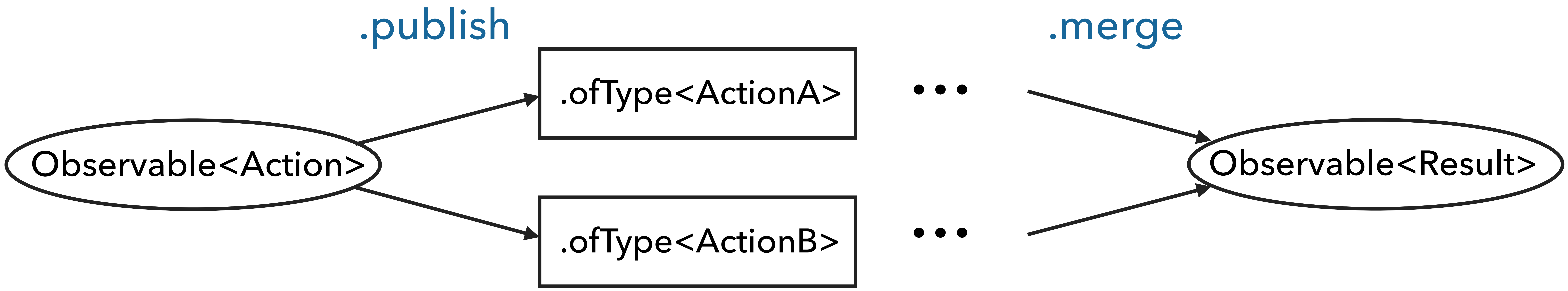
```
sealed class InternalAction : Action {
  object SearchLoadingAction : InternalAction()
  class SearchSuccessAction(val data: String) : InternalAction()
  class SearchFailureAction(val error: Throwable) : InternalAction()
  class SuggestionsLoadedAction(val suggestions: List<String>) : InternalAction()
}
```

```kotlin
fun bind(actions: Observable<Action>): Observable<UiState> {
   ...
}
```

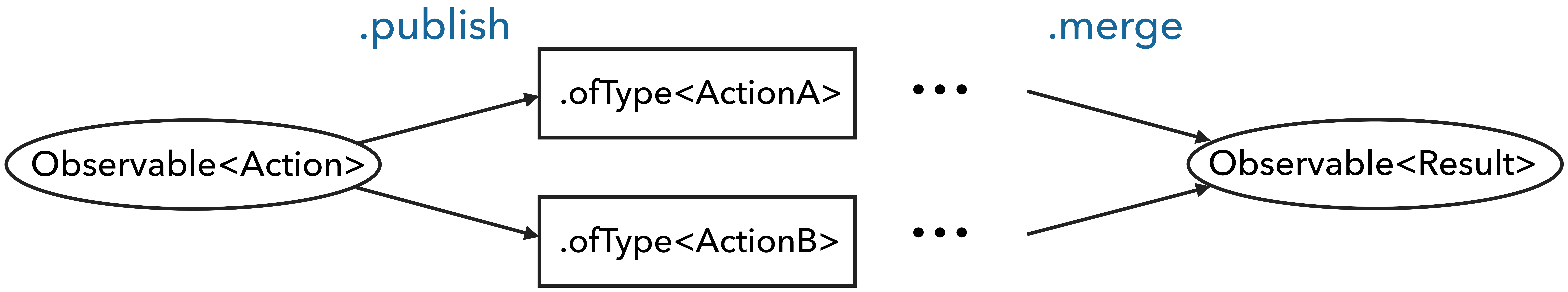
```
fun bind(actions: Observable<Action>): Observable<UiState> {
    ...
}
```



```
fun bind(actions: Observable<Action>): Observable<UiState> {
  return actions.publish { shared ->
        Observable.merge<Action>()
      }
}
```



```
fun bind(actions: Observable<Action>): Observable<UiState> {
  return actions.publish { shared ->
        Observable.merge<Action>(
           bind(shared.ofType<SearchAction>()),
           bind(shared.ofType<LoadSuggestionsAction>()))
      }
}
```



```kotlin
fun bind(actions: Observable<SearchAction>): Observable<InternalAction> {
  return actions.flatMap { action ->
    api.search(action.query)
        .map { result -> SearchSuccessAction(result) }
        .onErrorReturn { e -> SearchFailureAction(e) }
        .observeOn(uiScheduler)
        .startWith(SearchLoadingAction)
  }
}

fun bind(actions: Observable<LoadSuggestionsAction>): Observable<InternalAction> {
  return actions.flatMap { action ->
    api.suggestions(action.query)
        .onErrorReturnItem(emptyList())
        .map { result -> SuggestionsLoadedAction(result) }
        .observeOn(uiScheduler)
  }
}
```

```
fun bind(actions: Observable<SearchAction>): Observable<InternalAction> {
  return actions.flatMap { action ->
    api.search(action.query)
        .map { result -> SearchSuccessAction(result) }
        .onErrorReturn { e -> SearchFailureAction(e) }
        .observeOn(uiScheduler)
        .startWith(SearchLoadingAction)
  }
}

fun bind(actions: Observable<LoadSuggestionsAction>): Observable<InternalAction> {
  return actions.flatMap { action ->
    api.suggestions(action.query)
        .onErrorReturnItem(emptyList())
        .map { result -> SuggestionsLoadedAction(result) }
        .observeOn(uiScheduler)
  }
}
```

```kotlin
fun bind(actions: Observable<Action>): Observable<UiState> {
  return actions.publish { shared ->
      Observable.merge<Action>(
           bind(shared.ofType<SearchAction>()),
           bind(shared.ofType<LoadSuggestionsAction>()))
      }
}
```

```
fun bind(actions: Observable<Action>): Observable<UiState> {
  return actions.publish { shared ->
      Observable.merge<Action>(
           bind(shared.ofType<SearchAction>()),
           bind(shared.ofType<LoadSuggestionsAction>()))
      }
      .scan(UiState()) { state, action ->
       ...
      }
}
```

```
            bind(shared.ofType<LoadSuggestionsAction>()))
        }
        .scan(UiState()) { state, action ->
          when (action) {
            SearchLoadingAction -> state.copy(
                loading = true,
                error = null,
                suggestions = null)

            is SearchSuccessAction -> state.copy(
                loading = false,
                data = newData,
                error = null,
                suggestions = null)

            is SearchFailureAction -> state.copy(
                loading = false,
                error = action.error)

            is SuggestionsLoadedAction -> state.copy(
                suggestions = action.suggestions)

            is SearchAction, is LoadSuggestionsAction -> state
          }
        }
  }
```

```
            bind(shared.ofType<LoadSuggestionsAction>()))
        }
        .scan(UiState()) { state, action ->
          when (action) {
            SearchLoadingAction -> state.copy(
                loading = true,
                error = null,
                suggestions = null)

            is SearchSuccessAction -> state.copy(
                loading = false,
                data = newData,
                error = null,
                suggestions = null)

            is SearchFailureAction -> state.copy(
                loading = false,
                error = action.error)

            is SuggestionsLoadedAction -> state.copy(
                suggestions = action.suggestions)

            is SearchAction, is LoadSuggestionsAction -> state
          }
        }
}
```

```
           bind(shared.ofType<LoadSuggestionsAction>()))
      }
      .scan(UiState()) { state, action ->
        when (action) {
          SearchLoadingAction -> state.copy(
              loading = true,
              error = null,
              suggestions = null)

          is SearchSuccessAction -> state.copy(
              loading = false,
              data = newData,
              error = null,
              suggestions = null)

          is SearchFailureAction -> state.copy(
              loading = false,
              error = action.error)

          is SuggestionsLoadedAction -> state.copy(
              suggestions = action.suggestions)

          is SearchAction, is LoadSuggestionsAction -> state
        }
      }
}
```

```
           bind(shared.ofType<LoadSuggestionsAction>()))
      }
      .scan(UiState()) { state, action ->
        when (action) {
          SearchLoadingAction -> state.copy(
              loading = true,
              error = null,
              suggestions = null)

          is SearchSuccessAction -> state.copy(
              loading = false,
              data = newData,
              error = null,
              suggestions = null)

          is SearchFailureAction -> state.copy(
              loading = false,
              error = action.error)

          is SuggestionsLoadedAction -> state.copy(
              suggestions = action.suggestions)

          is SearchAction, is LoadSuggestionsAction -> state
        }
      }
}
```

```
           bind(shared.ofType<LoadSuggestionsAction>()))
      }
      .scan(UiState()) { state, action ->
        when (action) {
          SearchLoadingAction -> state.copy(
              loading = true,
              error = null,
              suggestions = null)

          is SearchSuccessAction -> state.copy(
              loading = false,
              data = newData,
              error = null,
              suggestions = null)

          is SearchFailureAction -> state.copy(
              loading = false,
              error = action.error)

          is SuggestionsLoadedAction -> state.copy(
              suggestions = action.suggestions)

          is SearchAction, is LoadSuggestionsAction -> state
        }
      }
}
```

```
           bind(shared.ofType<LoadSuggestionsAction>()))
      }
      .scan(UiState()) { state, action ->
        when (action) {
          SearchLoadingAction -> state.copy(
              loading = true,
              error = null,
              suggestions = null)

          is SearchSuccessAction -> state.copy(
              loading = false,
              data = newData,
              error = null,
              suggestions = null)

          is SearchFailureAction -> state.copy(
              loading = false,
              error = action.error)

          is SuggestionsLoadedAction -> state.copy(
              suggestions = action.suggestions)

          is SearchAction, is LoadSuggestionsAction -> state
        }
      }
}
```

REACTIVE STATE

```
class SearchComponent(private val api: Api, private val uiScheduler: Scheduler) {

  fun bind(actions: Observable<Action>): Observable<UiState> {
    return actions.publish { shared ->
          Observable.merge<Action>(
              bind(shared.ofType<SearchAction>()),
              bind(shared.ofType<LoadSuggestionsAction>()))
        }
        .scan(UiState()) { state, action ->
          when (action) {
            SearchLoadingAction                       -> state.copy(...)
            is SearchSuccessAction                    -> state.copy(...)
            is SearchFailureAction                    -> state.copy(...)
            is SuggestionsLoadedAction                -> state.copy(...)
            is SearchAction, is LoadSuggestionsAction -> state
          }
        }
  }
}
```

UNIDIRECTIONAL DATA FLOW

UNIDIRECTIONAL DATA FLOW

- Redux

UNIDIRECTIONAL DATA FLOW

- Redux
- Cycle.js

UNIDIRECTIONAL DATA FLOW

- Redux
- Cycle.js
- Flux

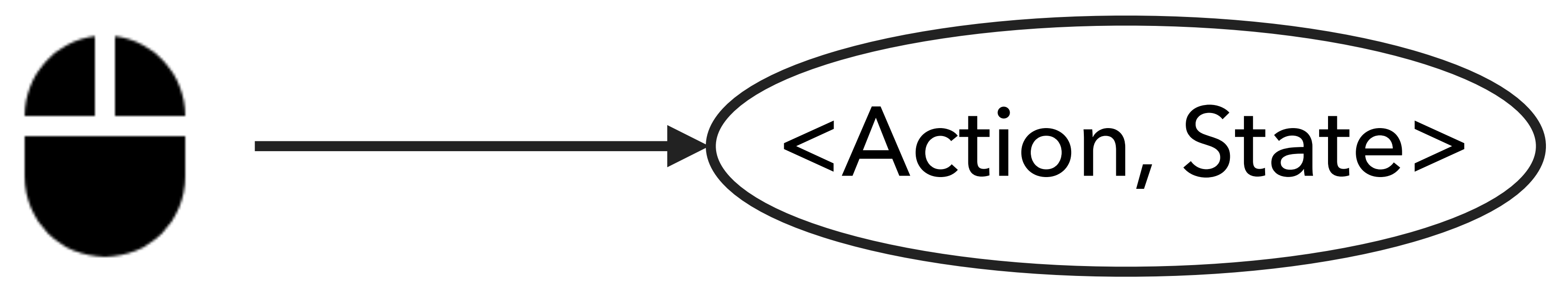
UNIDIRECTIONAL DATA FLOW

- Redux
- Cycle.js
- Flux
- Elm

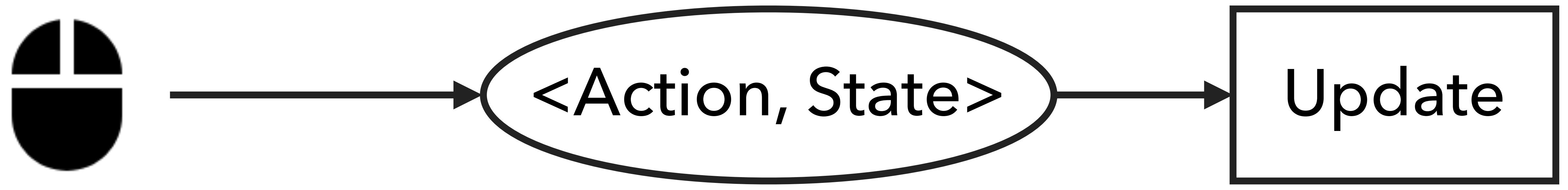
ELM COMPONENT

ELM COMPONENT

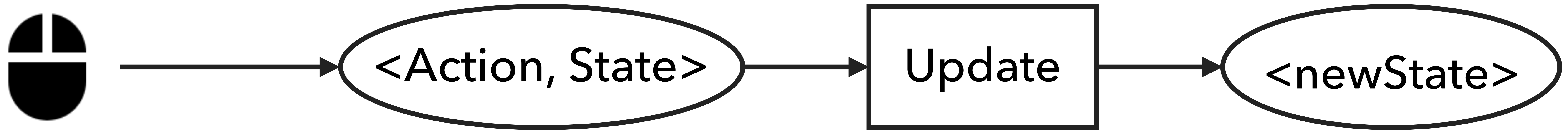
ELM COMPONENT



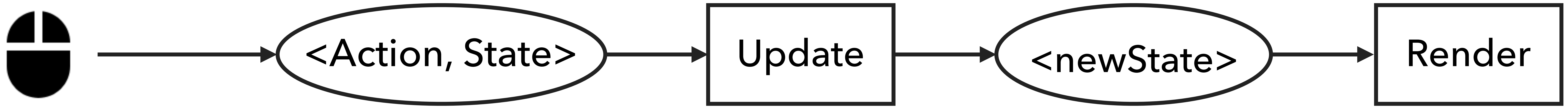
ELM COMPONENT



ELM COMPONENT



ELM COMPONENT



ELM COMPONENT

<Action, State>
Update
<newState>
Render

ELM COMPONENT

<Action, State>
Update
<newState>
Render
<SideEffect>

ELM COMPONENT

<Action, State>
Update
<newState>
Render
<SideEffect>
Call

ELM COMPONENT

<Action, State>
Update
<newState>
Render
<SideEffect>
Call
<Action>

ELM COMPONENT

<Action, State>
Update
<newState>
Render
<SideEffect>
Call
<Action>

ELM COMPONENT

<Action, State>
Update
<newState>
Render
<SideEffect>
Call
<Action>

ELM

ELM

- `Update`

ELM

- Update
- Call

ELM

- `Update`
- `Call`
- `Component`

ELM - REDUX

- `Update      ->  Reducer`
- `Call`
- `Component`

ELM - REDUX

- `Update     ->  Reducer`
- `Call       ->  Middleware`
- `Component`

ELM - REDUX

- `Update     ->  Reducer`
- `Call       ->  Middleware`
- `Component  ->  Store`

REACTIVE STATE

```
class SearchComponent(private val api: Api, private val uiScheduler: Scheduler) {

  fun bind(actions: Observable<Action>): Observable<UiState> {
    return actions.publish { shared ->
          Observable.merge<Action>(
              bind(shared.ofType<SearchAction>()),
              bind(shared.ofType<LoadSuggestionsAction>()))
        }
        .scan(UiState()) { state, action ->
          when (action) {
            SearchLoadingAction                            -> state.copy(...)
            is SearchSuccessAction                         -> state.copy(...)
            is SearchFailureAction                         -> state.copy(...)
            is SuggestionsLoadedAction                     -> state.copy(...)
            is SearchAction, is LoadSuggestionsAction -> state
          }
        }
  }
}
```

REACTIVE STATE

```
class SearchComponent(private val api: Api, private val uiScheduler: Scheduler) {

  fun bind(actions: Observable<Action>): Observable<UiState> {
    return actions.publish { shared ->
          Observable.merge<Action>(
              bind(shared.ofType<SearchAction>()),
              bind(shared.ofType<LoadSuggestionsAction>()))
        }
        .scan(UiState()) { state, action ->
          when (action) {
            SearchLoadingAction                              -> state.copy(...)
            is SearchSuccessAction                           -> state.copy(...)
            is SearchFailureAction                           -> state.copy(...)
            is SuggestionsLoadedAction                       -> state.copy(...)
            is SearchAction, is LoadSuggestionsAction -> state
          }
        }
  }
}
```

Reducer

REACTIVE STATE

```
class SearchComponent(private val api: Api, private val uiScheduler: Scheduler) {

  fun bind(actions: Observable<Action>): Observable<UiState> {
    return actions.publish { shared ->
          Observable.merge<Action>(                             Middleware
              bind(shared.ofType<SearchAction>()),
              bind(shared.ofType<LoadSuggestionsAction>()))
        }
        .scan(UiState()) { state, action ->                        Reducer
          when (action) {
            SearchLoadingAction                         -> state.copy(...)
            is SearchSuccessAction                      -> state.copy(...)
            is SearchFailureAction                      -> state.copy(...)
            is SuggestionsLoadedAction                  -> state.copy(...)
            is SearchAction, is LoadSuggestionsAction -> state
          }
        }
  }
}
```

REACTIVE STATE

Store

```
class SearchComponent(private val api: Api, private val uiScheduler: Scheduler) {

  fun bind(actions: Observable<Action>): Observable<UiState> {
    return actions.publish { shared ->
          Observable.merge<Action>(                                Middleware
              bind(shared.ofType<SearchAction>()),
              bind(shared.ofType<LoadSuggestionsAction>()))
        }
        .scan(UiState()) { state, action ->                        Reducer
          when (action) {
            SearchLoadingAction                          -> state.copy(...)
            is SearchSuccessAction                       -> state.copy(...)
            is SearchFailureAction                       -> state.copy(...)
            is SuggestionsLoadedAction                   -> state.copy(...)
            is SearchAction, is LoadSuggestionsAction -> state
          }
        }
  }
}
```

REDUCER

```kotlin
interface Reducer<S, A> {
  fun reduce(state: S, action: A): S
}
```

REDUCER

```
class SearchReducer : Reducer<UiState, Action> {
  override fun reduce(state: UiState, action: Action): UiState {
    return when (action) {
      SearchLoadingAction                           -> state.copy(...)
      is SearchSuccessAction                        -> state.copy(...)
      is SearchFailureAction                        -> state.copy(...)
      is SuggestionsLoadedAction                    -> state.copy(...)
      is SearchAction, is LoadSuggestionsAction -> state
    }
  }
}
```

MIDDLEWARE

```kotlin
interface Middleware<A, S> {
  fun bind(actions: Observable<A>, state: Observable<S>): Observable<A>
}
```

MIDDLEWARE

```
class SearchMiddleware(val api: Api) : Middleware<Action, UiState> {

  override fun bind(actions: Observable<Action>): Observable<Action> {
    return actions.ofType<SearchAction>()
        .flatMap { action ->
          api.search(action.query)
              .map<InternalAction> { result -> SearchSuccessAction(result) }
              .onErrorReturn { e -> SearchFailureAction(e) }
              .startWith(SearchLoadingAction)
        }
  }
}
```

REACTIVE STATE

```
class SearchMiddleware(val api: Api) : Middleware<Action, UiState> {

  override fun bind(actions: Observable<Action>, state: Observable<UiState>)
      : Observable<Action> {
    return actions.ofType<SearchAction>()
        .withLatestFrom(state) { action, currentState -> action to currentState }
        .flatMap { (action, state) ->
          api.search(action.query)
              .map<InternalAction> { result -> SearchSuccessAction(result) }
              .onErrorReturn { e -> SearchFailureAction(e) }
              .startWith(SearchLoadingAction)
        }
  }
}
```

REACTIVE STATE

```
class SearchMiddleware(val api: Api) : Middleware<Action, UiState> {

  override fun bind(actions: Observable<Action>, state: Observable<UiState>)
      : Observable<Action> {
    return actions.ofType<SearchAction>()
        .withLatestFrom(state) { action, currentState -> action to currentState }
        .flatMap { (action, state) ->
          api.search(action.query)
              .map<InternalAction> { result -> SearchSuccessAction(result) }
              .onErrorReturn { e -> SearchFailureAction(e) }
              .startWith(SearchLoadingAction)
        }
  }
}
```

STORE – SEARCH COMPONENT

STORE - SEARCH COMPONENT

```
private val state = BehaviorRelay.createDefault<UiState>(UiState())
private val actions = PublishRelay.create<Action>()
```

STORE – SEARCH COMPONENT

```kotlin
private val state = BehaviorRelay.createDefault<UiState>(UiState())
private val actions = PublishRelay.create<Action>()

fun wire(): Disposable {
  val disposable = CompositeDisposable()

  disposable += actions
      .withLatestFrom(state) { action, state ->
        SearchReducer().reduce(state, action)
      }
      .subscribe(state::accept)

  return disposable
}
```

STORE - SEARCH COMPONENT

```
private val state = BehaviorRelay.createDefault<UiState>(UiState())
private val actions = PublishRelay.create<Action>()

fun wire(): Disposable {
  val disposable = CompositeDisposable()

  disposable += actions
      .withLatestFrom(state) { action, state ->
        SearchReducer().reduce(state, action)
      }
      .distinctUntilChanged()
      .subscribe(state::accept)

  return disposable
}
```

STORE - SEARCH COMPONENT

```
private val state = BehaviorRelay.createDefault<UiState>(UiState())
private val actions = PublishRelay.create<Action>()

fun wire(): Disposable {
  val disposable = CompositeDisposable()

  disposable += actions
      .withLatestFrom(state) { action, state ->
        SearchReducer().reduce(state, action)
      }
      .distinctUntilChanged()
      .subscribe(state::accept)

  disposable += Observable.merge<Action>(
      SearchMiddleware(api).bind(actions, state),
      SuggestionsMiddleware(api).bind(actions, state)
  ).subscribe(actions::accept)

  return disposable
}
```

STORE – SEARCH COMPONENT

```
private val state = BehaviorRelay.createDefault<UiState>(UiState())
private val actions = PublishRelay.create<Action>()

fun wire(): Disposable {
  val disposable = CompositeDisposable()

  disposable += actions
      .withLatestFrom(state) { action, state ->
        SearchReducer().reduce(state, action)
      }
      .distinctUntilChanged()
      .subscribe(state::accept)

  disposable += Observable.merge<Action>(
      SearchMiddleware(api).bind(actions, state),
      SuggestionsMiddleware(api).bind(actions, state)
  ).subscribe(actions::accept)

  return disposable
}
```

STORE

```
fun bind(actions: Observable<Action>, render: (UiState) -> Unit): Disposable {
  val disposable = CompositeDisposable()
  disposable += state.observeOn(uiScheduler).subscribe(render)
  disposable += actions.subscribe(actions::accept)
  return disposable
}
```

STORE

```
fun bind(actions: Observable<Action>, render: (UiState) -> Unit): Disposable {
  val disposable = CompositeDisposable()
  disposable += state.observeOn(uiScheduler).subscribe(render)
  disposable += actions.subscribe(actions::accept)
  return disposable
}
```

STORE

```kotlin
interface MviView<A, S> {
  val actions: Observable<A>
  fun render(state: S)
}


// SearchComponent
fun bind(actions: Observable<Action>, render: (UiState) -> Unit): Disposable {
  val disposable = CompositeDisposable()
  disposable += state.observeOn(uiScheduler).subscribe(render)
  disposable += actions.subscribe(actions::accept)
  return disposable
}
```

STORE

```
interface MviView<A, S> {
  val actions: Observable<A>
  fun render(state: S)
}


// SearchComponent
fun bind(view: MviView<Action, UiState>): Disposable {
  val disposable = CompositeDisposable()
  disposable += state.observeOn(uiScheduler).subscribe(view::render)
  disposable += view.actions.subscribe(actions::accept)
  return disposable
}
```

STORE

```kotlin
interface MviView<A, S> {
  val actions: Observable<A>
  fun render(state: S)
}


// SearchComponent
fun bind(view: MviView<Action, UiState>): Disposable {
  val disposable = CompositeDisposable()
  disposable += state.observeOn(uiScheduler).subscribe(view::render)
  disposable += view.actions.subscribe(actions::accept)
  return disposable
}
```

STORE

```
private val state = BehaviorRelay.createDefault<UiState>(UiState())
private val actions = PublishRelay.create<Action>()

fun wire(): Disposable {
  val disposable = CompositeDisposable()

  disposable += actions
      .withLatestFrom(state) { action, state ->
        SearchReducer().reduce(state, action)
      }
      .distinctUntilChanged()
      .subscribe(state::accept)

  disposable += Observable.merge<Action>(
      SearchMiddleware(api).bind(actions, state),
      SuggestionsMiddleware(api).bind(actions, state)
  ).subscribe(actions::accept)

  return disposable
}
```

STORE

```
private val state = BehaviorRelay.createDefault<UiState>(UiState())
private val actions = PublishRelay.create<Action>()

fun wire(): Disposable {
  val disposable = CompositeDisposable()

  disposable += actions
      .withLatestFrom(state) { action, state ->
        SearchReducer().reduce(state, action)
      }
      .distinctUntilChanged()
      .subscribe(state::accept)

  disposable += Observable.merge<Action>(
      SearchMiddleware(api).bind(actions, state),
      SuggestionsMiddleware(api).bind(actions, state)
  ).subscribe(actions::accept)

  return disposable
}
```

STORE

```
private val state = BehaviorRelay.createDefault<UiState>(initialState)
private val actions = PublishRelay.create<Action>()

fun wire(): Disposable {
  val disposable = CompositeDisposable()

  disposable += actions
      .withLatestFrom(state) { action, state ->
        reducer.reduce(state, action)
      }
      .distinctUntilChanged()
      .subscribe(state::accept)

  disposable += Observable.merge<Action>(
      middlewares.map { it.bind(actions, state) }
  ).subscribe(actions::accept)

  return disposable
}
```

STORE

```kotlin
class SearchComponent(
    private val reducer: Reducer<UiState, Action>,
    private val middlewares: List<Middleware<Action, UiState>>,
    private val initialState: UiState
)
```

STORE

```
class Store<A, S>(
    private val reducer: Reducer<S, A>,
    private val middlewares: List<Middleware<A, S>>,
    private val initialState: S
)
```

CORE

```kotlin
interface MviView<A, S> {
  val actions: Observable<A>
  fun render(state: S)
}

interface Reducer<S, A> {
  fun reduce(state: S, action: A): S
}

interface Middleware<A, S> {
  fun bind(actions: Observable<A>, state: Observable<S>): Observable<A>
}

class Store<A, S>(
    private val reducer: Reducer<S, A>,
    private val middlewares: List<Middleware<A, S>>,
    private val initialState: S
) {
  fun wire(): Disposable {}
  fun bind(view: MviView<Action, UiState>): Disposable {}
}
```

BIND IT ALL!

BIND IT ALL!

- Android Arch ViewModels

BIND IT ALL!

- Android Arch ViewModels
- DI Scopes

BIND IT ALL!

- Android Arch ViewModels
- DI Scopes
- Others

BIND IT ALL!

```
class SearchViewModel<A, S>
@Inject constructor (private val store: Store<A, S>) : ViewModel() {

  private val wiring = store.wire()
  private var viewBinding: Disposable? = null

  override fun onCleared() {
    wiring.dispose()
  }

  fun bind(view: MviView<A, S>) {
    viewBinding = store.bind(view)
  }

  fun unbind() {
    viewBinding?.dispose()
  }
}
```

BIND IT ALL!

```
class SearchViewModel<A, S>
@Inject constructor (private val store: Store<A, S>) : ViewModel() {

  private val wiring = store.wire()
  private var viewBinding: Disposable? = null

  override fun onCleared() {
    wiring.dispose()
  }

  fun bind(view: MviView<A, S>) {
    viewBinding = store.bind(view)
  }

  fun unbind() {
    viewBinding?.dispose()
  }
}
```

BIND IT ALL!

```
class SearchViewModel<A, S>
@Inject constructor (private val store: Store<A, S>) : ViewModel() {

  private val wiring = store.wire()
  private var viewBinding: Disposable? = null

  override fun onCleared() {
    wiring.dispose()
  }

  fun bind(view: MviView<A, S>) {
    viewBinding = store.bind(view)
  }

  fun unbind() {
    viewBinding?.dispose()
  }
}
```

BIND IT ALL!

```kotlin
class SearchViewModel<A, S>
@Inject constructor (private val store: Store<A, S>) : ViewModel() {

  private val wiring = store.wire()
  private var viewBinding: Disposable? = null

  override fun onCleared() {
    wiring.dispose()
  }

  fun bind(view: MviView<A, S>) {
    viewBinding = store.bind(view)
  }

  fun unbind() {
    viewBinding?.dispose()
  }
}
```

OUR UNIDIRECTIONAL DATA FLOW

OUR UNIDIRECTIONAL DATA FLOW

Reducer
::render
Middleware

OUR UNIDIRECTIONAL DATA FLOW

SearchAction
Reducer
::render
Middleware

OUR UNIDIRECTIONAL DATA FLOW

< SearchAction, State.Empty >
Reducer
::render
Middleware

OUR UNIDIRECTIONAL DATA FLOW

<SearchAction, State.Empty>
State.Empty
Reducer
::render
Middleware

OUR UNIDIRECTIONAL DATA FLOW

< SearchAction, State.Empty >
State.Empty
Reducer
::render
Middleware
SearchLoadingAction

OUR UNIDIRECTIONAL DATA FLOW

<SearchLoadingAction , State.Empty>
Reducer
::render
Middleware

OUR UNIDIRECTIONAL DATA FLOW

<SearchLoadingAction , State.Empty>
State(loading = true)
Reducer
::render
Middleware

OUR UNIDIRECTIONAL DATA FLOW

<SearchLoadingAction,
State(loading = true)>
Reducer
::render
Middleware

OUR UNIDIRECTIONAL DATA FLOW

<SearchLoadingAction,
State(loading = true)>
Reducer
::render
Middleware

OUR UNIDIRECTIONAL DATA FLOW

<SearchLoadingAction,
State(loading = true)>
Reducer
::render
Middleware
SearchSuccessAction

OUR UNIDIRECTIONAL DATA FLOW

< SearchSuccessAction ,
State(loading = true)>
Reducer
::render
Middleware

OUR UNIDIRECTIONAL DATA FLOW

< SearchSuccessAction ,
State(loading = true)>
Reducer
State(loading = false,
data = newData)
::render
Middleware

BACK TO REALITY

BACK TO REALITY

- Rendering Overflow

ANY PROBLEMS WITH UI?

ANY PROBLEMS WITH UI?

- Lots of State updates

ANY PROBLEMS WITH UI?

- Lots of State updates
- May cause lots of redundant UI rerendering

ANY PROBLEMS WITH UI?

- Lots of State updates
- May cause lots of redundant UI rerendering
- …

ANY PROBLEMS WITH UI?

- Lots of State updates
- May cause lots of redundant UI rerendering
- …
- Domic!

WHAT IS DOMIC

WHAT IS DOMIC

- Diffing changes

WHAT IS DOMIC

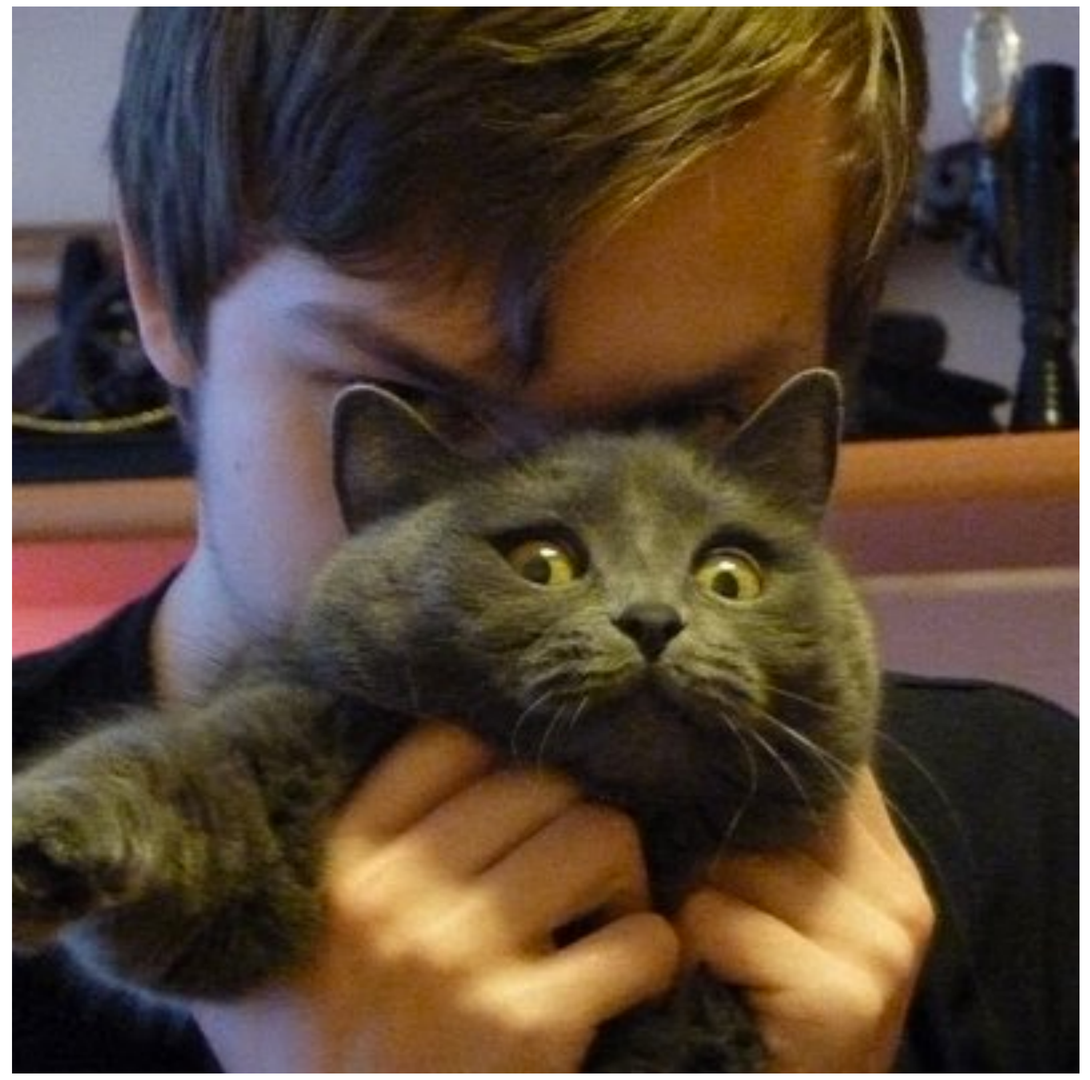
- Diffing changes
- Same Android Widgets

WHAT IS DOMIC

- Diffing changes
- Same Android Widgets
- Threading

WHAT IS DOMIC

- Diffing changes
- Same Android Widgets
- Threading



BACK TO REALITY

- Rendering Overflow
- Testing

TESTING

TESTING VIEW

```kotlin
// In
val observer = TestObserver.create<UiAction>()
realView.actions.subscribe(observer)

onView(withId(R.id.search_edit)).perform(typeText("Query"))
onView(withId(R.id.submit_btn)).perform(click())
observer.assertValue(SearchAction("Query"))



// Out
val uiState = UiState(loading = false, data = "TestData")
realView.render(uiState)
takeScreenshot()
```

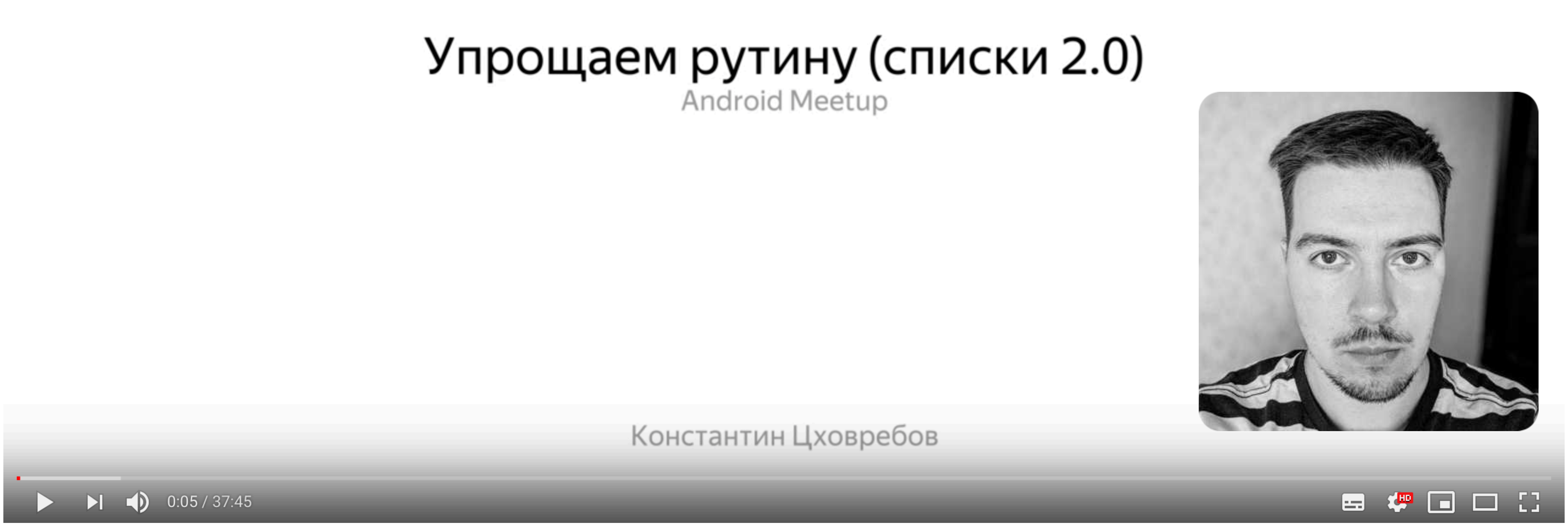
TESTING LOGIC

```
val viewModel = provide<SearchViewModel<Action, UiState>>

viewModel.bind(fakeView)
actions.onNext(SearchAction("Query"))
states.assertValue(UiState(loading = true))

viewModel.unbind()
actions.onNext(SearchAction("AnotherQuery"))
states.assertNoValues()
```

BACK TO REALITY

- Rendering Overflow
- Testing
- Paging

PAGING


Упрощаем рутину (списки 2.0)

Android Meetup

Константин Цховребов

youtu.be/h5afEeuI0GQ

BACK TO REALITY

- Rendering Overflow
- Testing
- Paging
- SingleLiveEvents

SINGLE LIVE EVENTS

SINGLE LIVE EVENTS

<Action, State>
Update
<newState>
Render
<SideEffect>
Call
<Action>

SINGLE LIVE EVENTS

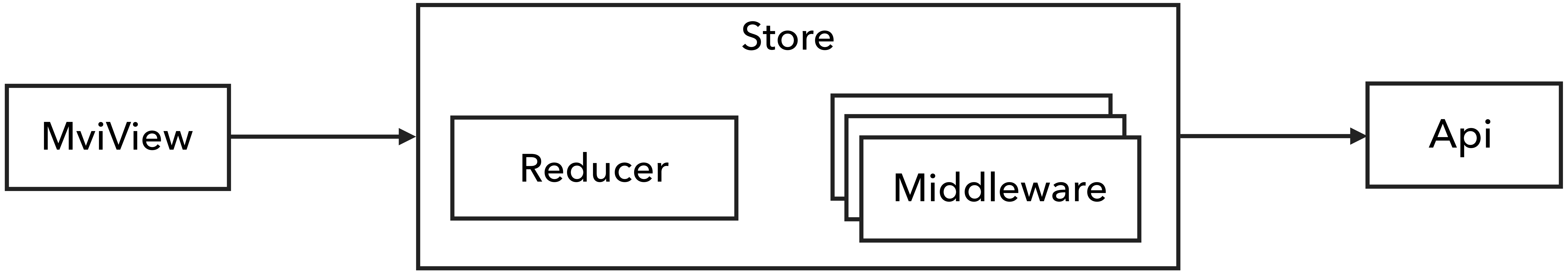
<Action, State>
Update
<newState>
Render
<SideEffect>
Call
<Action>
Subscriptions

BACK TO REALITY

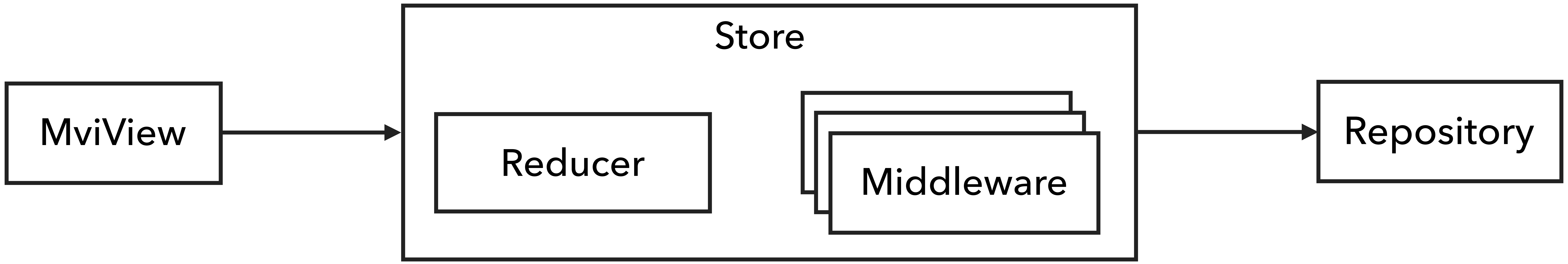
- Rendering Overflow
- Testing
- Paging
- SingleLiveEvents
- Clean Architecture

DO YOU EVEN CLEAN?

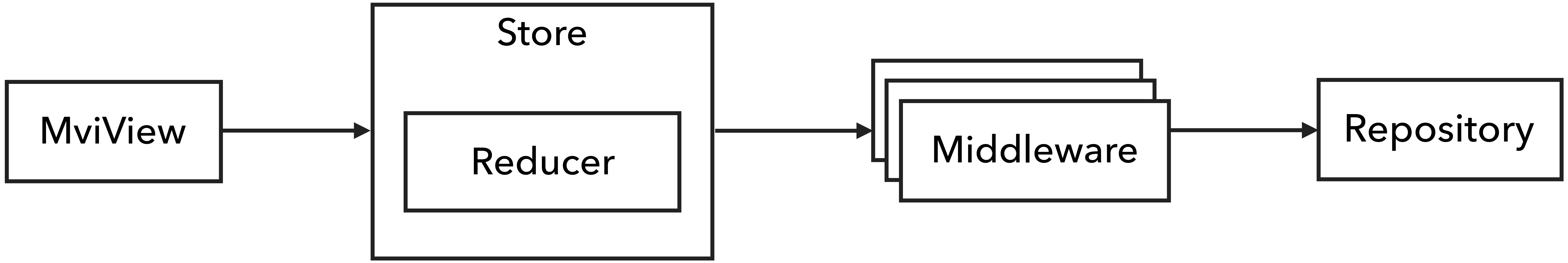
DO YOU EVEN CLEAN?



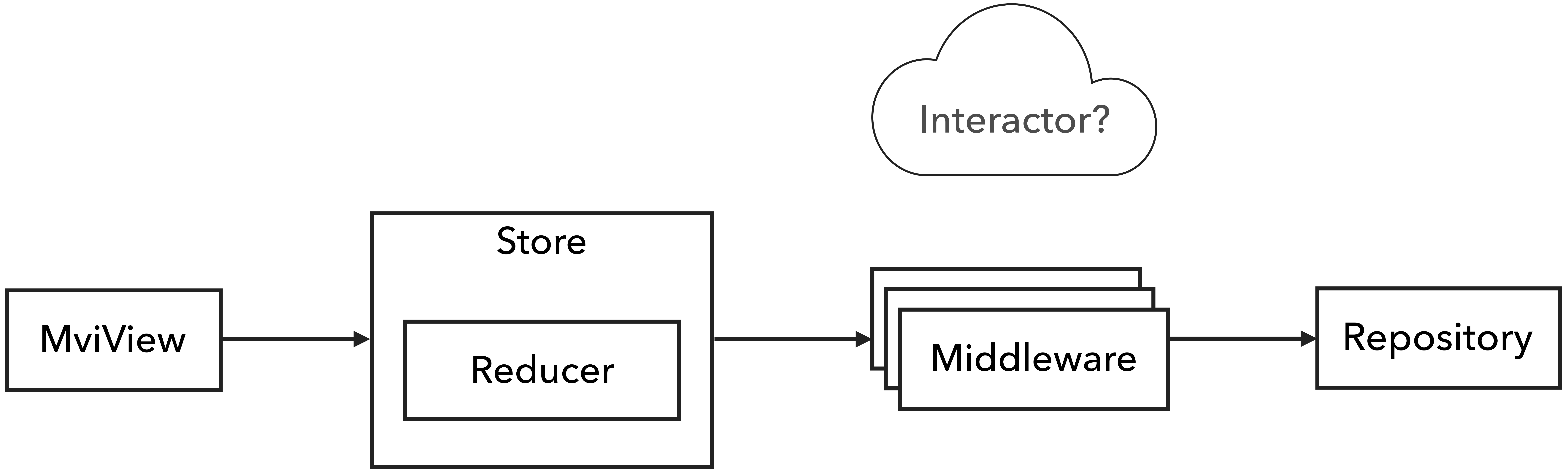
DO YOU EVEN CLEAN?



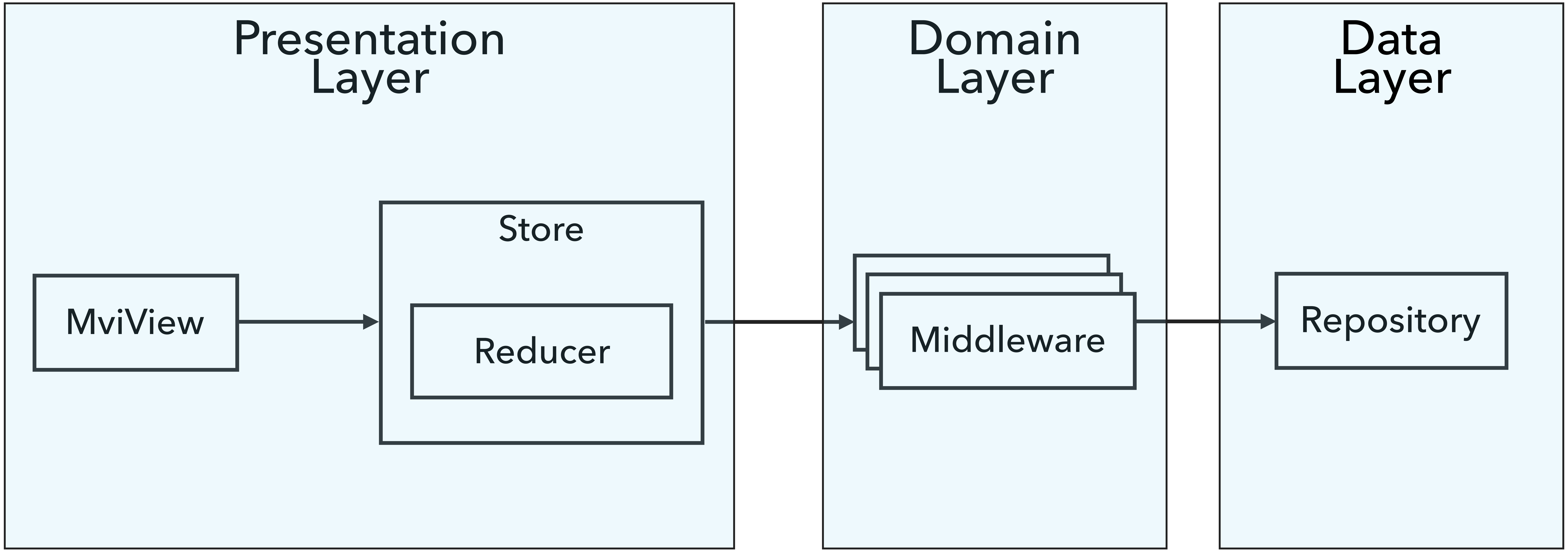
DO YOU EVEN CLEAN?



DO YOU EVEN CLEAN?



DO YOU EVEN CLEAN?



TO WRAP UP

TO WRAP UP

- MVI is about presentation logic

TO WRAP UP

- MVI is about presentation logic
- Reactive flow

TO WRAP UP

- MVI is about presentation logic
- Reactive flow
- Sealed classes for Actions & States

TO WRAP UP

- MVI is about presentation logic
- Reactive flow
- Sealed classes for Actions & States
- Kotlin multiplatform usage options

TO WRAP UP

- MVI is about presentation logic
- Reactive flow
- Sealed classes for Actions & States
- Kotlin multiplatform usage options
  - RxKotlin (for real)

TO WRAP UP

- MVI is about presentation logic
- Reactive flow
- Sealed classes for Actions & States
- Kotlin multiplatform usage options
  - RxKotlin (for real)
  - Reagent-like

TO WRAP UP

- MVI is about presentation logic
- Reactive flow
- Sealed classes for Actions & States
- Kotlin multiplatform usage options
  - RxKotlin (for real)
  - Reagent-like
- Time Travel Debugger

UNIDIRECTIONAL DATA FLOW LIBS

UNIDIRECTIONAL DATA FLOW LIBS

- RxRedux / Freeletics   github.com/freeletics/RxRedux

UNIDIRECTIONAL DATA FLOW LIBS

- RxRedux / Freeletics github.com/freeletics/RxRedux
- Mobius / Spotify github.com/spotify/mobius

UNIDIRECTIONAL DATA FLOW LIBS

- RxRedux / Freeletics github.com/freeletics/RxRedux
- Mobius / Spotify github.com/spotify/mobius
- MvRx / AirBnb github.com/airbnb/MvRx

UNIDIRECTIONAL DATA FLOW LIBS

- RxRedux / Freeletics github.com/freeletics/RxRedux
- Mobius / Spotify github.com/spotify/mobius
- MvRx / AirBnb github.com/airbnb/MvRx
- MVICore / Badoo github.com/badoo/MVICore

UNIDIRECTIONAL DATA FLOW LIBS

- RxRedux / Freeletics github.com/freeletics/RxRedux
- Mobius / Spotify github.com/spotify/mobius
- MvRx / AirBnb github.com/airbnb/MvRx
- MVICore / Badoo github.com/badoo/MVICore
- Grox / Groupon github.com/groupon/grox

UNIDIRECTIONAL DATA FLOW LIBS

- RxRedux / Freeletics github.com/freeletics/RxRedux
- Mobius / Spotify github.com/spotify/mobius
- MvRx / AirBnb github.com/airbnb/MvRx
- MVICore / Badoo github.com/badoo/MVICore
- Grox / Groupon github.com/groupon/grox
- Suas / Zendesk github.com/zendesk/Suas-Android

MVICORE

MVICORE

- `Wish`

MVICORE

- `Wish`
- `Effect`

MVICORE

- `Wish`
- `Effect`
- `Actor`

MVICORE

- Wish
- Effect
- Actor
- News

MVICORE

- `Wish`
- `Effect`
- `Actor`
- `News`
- `Feature`

MVICORE

- `Wish     ->  Action`
- `Effect`
- `Actor`
- `News`
- `Feature`

MVICORE

- Wish -> Action
- Effect -> Internal Action
- Actor
- News
- Feature

MVICORE

Wish	->	Action
Effect	->	Internal Action
Actor	->	Middleware
News		
Feature		

MVICORE

- Wish -> Action
- Effect -> Internal Action
- Actor -> Middleware
- News -> Subscriptions
- Feature

MVICORE

Wish	->	Action
Effect	->	Internal Action
Actor	->	Middleware
News	->	Subscriptions
Feature	->	Store

LINKS

- **Managing State with RxJava** by Jake Wharton
  youtu.be/0IKHxjkgop4
- **The Reactive Workflow Pattern** by Ray Ryan
  youtu.be/mvBVkU2mCF4
- **Domic – Reactive Virtual DOM** by Artem Zinnatullin
  youtu.be/Ce6phlHfKR8
- **Domic repo**
  github.com/lyft/domic
- **Reagent repo**
  github.com/JakeWharton/Reagent

How to cook a well done MVI for Android

Sergey Ryabov
@colriot